Julia 中文文档

Julia 中文社区

May 10, 2025

Contents

Contents ii

I 主页 1

1 Julia 1.10 中文文档 2
- 1.1 鸣谢 2
- 1.2 其它中文资料 2
- 1.3 简介 3
- 1.4 Julia 与其他语言的比较 3

II 手册 5

2 入门 6
- 2.1 资源 7

3 变量 8
- 3.1 合法的变量名 9
- 3.2 赋值表达式与赋值和修改的区别 10
- 3.3 命名规范 11

4 整数和浮点数 13
- 4.1 整数 14
- 4.2 浮点数 17
- 4.3 任意精度算术 22
- 4.4 数值字面量系数 23
- 4.5 零和一的字面量 25

5 数学运算和初等函数 26
- 5.1 算术运算符 26
- 5.2 布尔运算符 27
- 5.3 位运算符 27
- 5.4 复合赋值运算符 28
- 5.5 向量化“点”运算符 29
- 5.6 数值比较 29
- 5.7 运算符的优先级与结合性 32
- 5.8 数值转换 34

6 复数和有理数 37
- 6.1 复数 37
- 6.2 有理数 40

7 字符串 43
- 7.1 字符 43
- 7.2 字符串基础 45
- 7.3 Unicode 和 UTF-8 47
- 7.4 拼接 50
- 7.5 插值 51

7.6 三引号字符串字面量 . . . 52
7.7 常见操作 . . . 54
7.8 正则表达式 . . . 55
7.9 字节数组字面量 . . . 60
7.10 版本号字面量 . . . 61
7.11 原始字符串字面量 . . . 62

8 函数 63

8.1 参数传递行为 . . . 64
8.2 参数类型声明 . . . 65
8.3 return 关键字 . . . 65
8.4 操作符也是函数 . . . 67
8.5 具有特殊名称的操作符 . . . 67
8.6 匿名函数 . . . 68
8.7 元组 . . . 69
8.8 具名元组 . . . 69
8.9 解构赋值和多返回值 . . . 70
8.10 Property destructuring . . . 72
8.11 参数解构 . . . 73
8.12 变参函数 . . . 73
8.13 可选参数 . . . 75
8.14 关键字参数 . . . 76
8.15 默认值作用域的计算 . . . 77
8.16 函数参数中的 Do 结构 . . . 78
8.17 函数的复合与链式调用 . . . 79
8.18 向量化函数的点语法 . . . 80
8.19 更多阅读 . . . 82

9 流程控制 83

9.1 复合表达式 . . . 83
9.2 条件表达式 . . . 84
9.3 短路求值 . . . 87
9.4 重复执行：循环 . . . 89
9.5 异常处理 . . . 92
9.6 Tasks 任务（或协程） . . . 97

10 变量作用域 98

10.1 全局作用域 . . . 99
10.2 局部作用域 . . . 99
10.3 常量 . . . 108
10.4 带类型的全局变量 . . . 111

11 类型 112

11.1 类型声明 . . . 113
11.2 抽象类型 . . . 114
11.3 原始类型 . . . 116
11.4 复合类型 . . . 116
11.5 可变复合类型 . . . 118
11.6 已声明的类型 . . . 120
11.7 类型共用体 . . . 120
11.8 参数类型 . . . 121
11.9 UnionAll 类型 . . . 128
11.10 单例类型 . . . 129
11.11 Types of functions . . . 130
11.12 Type{T} 类型选择器 . . . 131
11.13 类型别名 . . . 132

11.14 类型操作 . . . 133
11.15 自定义 pretty-printing . . . 134
11.16 值类型 . . . 136

12 方法 138

12.1 定义方法 . . . 138
12.2 Method specializations . . . 142
12.3 方法歧义 . . . 143
12.4 参数方法 . . . 144
12.5 重定义方法 . . . 146
12.6 使用参数方法设计样式 . . . 148
12.7 参数化约束的可变参数方法 . . . 151
12.8 可选参数和关键字的参数的注意事项 . . . 152
12.9 类函数对象 . . . 153
12.10 空泛型函数 . . . 153
12.11 方法设计与避免歧义 . . . 153
12.12 Defining methods in local scope . . . 156

13 构造函数 158

13.1 外部构造方法 . . . 158
13.2 内部构造方法 . . . 159
13.3 不完整初始化 . . . 160
13.4 参数类型的构造函数 . . . 162
13.5 示例学习：有理数 . . . 164
13.6 仅外部的构造函数 . . . 166

14 类型转换和类型提升 168

14.1 类型转换 . . . 168
14.2 类型提升 . . . 171

15 接口 174

15.1 迭代 . . . 174
15.2 Indexing . . . 176
15.3 抽象数组 . . . 177
15.4 等步长数组 . . . 181
15.5 自定义广播 . . . 181
15.6 Instance Properties . . . 186

16 模块 189

16.1 命名空间管理 . . . 190
16.2 子模块和相对路径 . . . 195
16.3 模块初始化和预编译 . . . 196

17 文档 200

17.1 Accessing Documentation . . . 200
17.2 编写文档 . . . 200
17.3 函数与方法 . . . 204
17.4 进阶用法 . . . 204
17.5 语法指南 . . . 206

18 元编程 211

18.1 程序表示 . . . 211
18.2 表达式与求值 . . . 213
18.3 宏 . . . 218
18.4 代码生成 . . . 226
18.5 非标准字符串字面量 . . . 227
18.6 生成函数 . . . 228

19 一维和多维数组 235

19.1 基本函数 . . . 235

19.2 构造和初始化 . . . 236
19.3 数组常量 . . . 237
19.4 数组推导 . . . 243
19.5 生成器表达式 . . . 244
19.6 索引 . . . 245
19.7 索引赋值 . . . 247
19.8 支持的索引类型 . . . 247
19.9 迭代 . . . 252
19.10 Array traits . . . 253
19.11 数组和向量化的算子与函数 . . . 253
19.12 广播 . . . 254
19.13 实现 . . . 255
20 缺失值 257
20.1 缺失值的传播 . . . 257
20.2 相等和比较运算符 . . . 257
20.3 逻辑运算符 . . . 258
20.4 流程控制和短路运算符 . . . 260
20.5 包含缺失值的数组 . . . 260
20.6 跳过缺失值 . . . 261
20.7 数组上的逻辑运算 . . . 263
21 网络和流 264
21.1 基础流 I/O . . . 264
21.2 文本 I/O . . . 265
21.3 IO 输出的上下文信息 . . . 266
21.4 使用文件 . . . 266
21.5 一个简单的 TCP 示例 . . . 267
21.6 解析 IP 地址 . . . 269
21.7 异步 I/O . . . 269
21.8 Multicast . . . 270
22 并行计算 272
23 异步编程 273
23.1 基本 Task 操作 . . . 273
23.2 在 Channel 中进行通信 . . . 274
23.3 更多任务操作 . . . 278
23.4 Task 和事件 . . . 278
24 多线程 279
24.1 启用 Julia 多线程 . . . 279
24.2 Threadpools . . . 280
24.3 Communication and synchronization . . . 281
24.4 @threads 宏 . . . 282
24.5 原子操作 . . . 284
24.6 field 粒度的原子操作 . . . 285
24.7 副作用和可变的函数参数 . . . 286
24.8 @threadcall . . . 286
24.9 注意！ . . . 286
24.10 Task Migration . . . 286
24.11 终结器的安全使用 . . . 287
25 多进程和分布式计算 288
25.1 访问代码以及加载库 . . . 290
25.2 启动和管理 worker 进程 . . . 291
25.3 数据转移 . . . 292
25.4 全局变量 . . . 293

25.5 并行的 Map 和 Loop . . . 294
25.6 远程引用和 AbstractChannel . . . 296
25.7 Channel 和 RemoteChannel . . . 296
25.8 本地调用 . . . 298
25.9 共享数组 . . . 299
25.10 集群管理器 . . . 303
25.11 指定网络拓补结构（实验性功能） . . . 308
25.12 一些值得关注的外部库 . . . 308

26 运行外部程序 312

26.1 插值 . . . 313
26.2 引用 . . . 315
26.3 管道 . . . 316
26.4 `Cmd` 对象 . . . 318

27 调用 C 和 Fortran 代码 319

27.1 创建和 C 兼容的 Julia 函数指针 . . . 321
27.2 将 C 类型映射到 Julia . . . 323

28 将 C 函数映射到 Julia 330

28.1 C 包装器示例 . . . 332
28.2 Fortran 包装器示例 . . . 333
28.3 垃圾回收安全 . . . 334
28.4 非常数函数规范 . . . 334
28.5 间接调用 . . . 334
28.6 cfunction 闭包 . . . 335
28.7 关闭库 . . . 335
28.8 Variadic function calls . . . 336
28.9 `ccall` interface . . . 336
28.10 Calling Convention . . . 336
28.11 访问全局变量 . . . 337
28.12 通过指针来访问数据 . . . 337
28.13 线程安全 . . . 338
28.14 关于 Callbacks 的更多内容 . . . 338
28.15 C++ . . . 338

29 处理操作系统差异 339

30 环境变量 340

30.1 文件位置 . . . 340
30.2 Pkg.jl . . . 343
30.3 Network transport . . . 344
30.4 External applications . . . 345
30.5 并行 . . . 345
30.6 REPL 格式化输出 . . . 346
30.7 System and Package Image Building . . . 347
30.8 Debugging and profiling . . . 347

31 嵌入 Julia 350

31.1 高级别嵌入 . . . 350
31.2 在 Windows 使用 Visual Studio 进行高级别嵌入 . . . 352
31.3 转换类型 . . . 353
31.4 调用 Julia 函数 . . . 353
31.5 内存管理 . . . 354
31.6 使用数组 . . . 357
31.7 异常 . . . 359

32 代码加载 363

32.1 定义 . . . 363

32.2 包的联合生态 . . . 364
32.3 环境（Environments） . . . 364
32.4 总结 . . . 373
33 性能分析 374
33.1 基本用法 . . . 374
33.2 结果累积和清空 . . . 377
33.3 用于控制性能分析结果显示的选项 . . . 377
33.4 配置 . . . 378
33.5 内存分配分析 . . . 379
33.6 外部性能分析 . . . 380
34 栈跟踪 382
34.1 查看栈跟踪 . . . 382
34.2 抽取有用信息 . . . 383
34.3 错误处理 . . . 384
34.4 异常栈与 `current_exceptions` . . . 385
34.5 `stacktrace` 与 `backtrace` 的比较 . . . 386
35 性能建议 388
35.1 影响性能的关键代码应该在函数内部 . . . 388
35.2 Avoid untyped global variables . . . 388
35.3 避免全局变量 . . . 388
35.4 使用 `@time` 评估性能以及注意内存分配 . . . 389
35.5 工具 . . . 391
35.6 避免使用抽象类型参数的容器 . . . 391
35.7 添加类型声明 . . . 392
35.8 将函数拆分为多个定义 . . . 398
35.9 编写「类型稳定的」函数 . . . 398
35.10 避免更改变量类型 . . . 399
35.11 分离核心函数（又称为函数屏障） . . . 399
35.12 具有值作为参数的类型 . . . 400
35.13 滥用多重派发的危险（也就是更多关于以值作为参数的类型） . . . 401
35.14 沿列按内存顺序访问数组 . . . 402
35.15 输出预分配 . . . 404
35.16 点语法：融合向量化操作 . . . 405
35.17 考虑对切片使用视图 . . . 405
35.18 复制数据不总是坏的 . . . 406
35.19 使用 StaticArrays.jl 进行小型固定大小的向量/矩阵运算 . . . 406
35.20 避免 I/O 中的字符串插值 . . . 407
35.21 并发执行时优化网络 I/O . . . 407
35.22 修复过期警告 . . . 408
35.23 小技巧 . . . 408
35.24 性能标注 . . . 408
35.25 将次正规数视为零 . . . 411
35.26 `@code_warntype` . . . 412
35.27 被捕获变量的性能 . . . 413
35.28 Multithreading and linear algebra . . . 415
35.29 Alternative linear algebra backends . . . 415
36 工作流程建议 416
36.1 基于 REPL 的工作流程 . . . 416
36.2 基于浏览器的工作流程 . . . 417
36.3 基于 Revise 的工作流程 . . . 417
37 代码风格指南 419
37.1 缩进 . . . 419

37.2 写函数，而不是仅仅写脚本 . . . 419
37.3 类型不要写得过于具体 . . . 419
37.4 让调用者处理多余的参数多样性 . . . 420
37.5 在修改其参数的函数名称后加！ . . . 420
37.6 避免使用奇怪的 Union 类型 . . . 421
37.7 避免复杂的容器类型 . . . 421
37.8 方法导出优先于直接字段访问 . . . 421
37.9 Use naming conventions consistent with Julia base/ . . . 422
37.10 使用和 Julia base/ 文件夹中的代码一致的命名习惯 . . . 422
37.11 使用与 Julia Base 中的函数类似的参数顺序 . . . 423
37.12 不要过度使用 try-catch . . . 423
37.13 不要给条件语句加括号 . . . 423
37.14 不要过度使用 424
37.15 不要使用不必要的静态参数 . . . 424
37.16 避免判断变量是实例还是类型的混乱 . . . 424
37.17 不要过度使用宏 . . . 424
37.18 不要把不安全的操作暴露在接口层 . . . 425
37.19 不要重载基础容器类型的方法 . . . 425
37.20 避免类型盗版 . . . 425
37.21 注意类型相等 . . . 426
37.22 不要写 x->f(x) . . . 426
37.23 尽可能避免使用浮点数作为通用代码的字面量 . . . 426
38 常见问题 428
38.1 概述 . . . 428
38.2 公共 API . . . 429
38.3 Sessions and the REPL . . . 429
38.4 会话和 REPL . . . 429
38.5 脚本 . . . 430
38.6 Variables and Assignments . . . 431
38.7 函数 . . . 432
38.8 类型，类型声明和构造函数 . . . 435
38.9 “method not matched”故障排除：参数类型不变性和 MethodError . . . 441
38.10 包和模块 . . . 443
38.11 空值与缺失值 . . . 443
38.12 内存 . . . 443
38.13 异步 IO 与并发同步写入 . . . 444
38.14 数组 . . . 445
38.15 计算集群 . . . 446
38.16 Julia 版本发布 . . . 446
39 与其他语言的显著差异 448
39.1 与 MATLAB 的显著差异 . . . 448
39.2 与 R 的显著差异 . . . 450
39.3 与 Python 的显著差异 . . . 452
39.4 与 C/C++ 的显著差异 . . . 454
39.5 与 Common Lisp 的显著差异 . . . 458
40 Unicode 输入表 460
41 Command-line Interface 461
41.1 Using arguments inside scripts . . . 461
41.2 Parallel mode . . . 461
41.3 Startup file . . . 462
41.4 Command-line switches for Julia . . . 462

III Base **464**

42 基本功能 **465**

42.1 介绍 . . . 465
42.2 Getting Around . . . 465
42.3 Keywords . . . 471
42.4 Standard Modules . . . 492
42.5 Base Submodules . . . 492
42.6 All Objects . . . 494
42.7 Properties of Types . . . 511
42.8 Special Types . . . 528
42.9 Generic Functions . . . 540
42.10 Syntax . . . 549
42.11 Missing Values . . . 568
42.12 System . . . 571
42.13 Versioning . . . 587
42.14 Errors . . . 588
42.15 Events . . . 599
42.16 Reflection . . . 601
42.17 Code loading . . . 607
42.18 Internals . . . 608
42.19 Meta . . . 615

43 集合和数据结构 **619**

43.1 迭代 . . . 619
43.2 构造函数和类型 . . . 621
43.3 通用集合 . . . 623
43.4 可迭代集合 . . . 625
43.5 可索引集合 . . . 675
43.6 字典 . . . 677
43.7 类似 Set 的集合 . . . 691
43.8 双端队列 . . . 698
43.9 集合相关的实用工具 . . . 708

44 数学相关 **710**

44.1 数学运算符 . . . 710
44.2 数学函数 . . . 740
44.3 自定义二元运算符 . . . 786

45 标准数值类型 **787**

45.1 数据格式 . . . 792
45.2 常用数值函数和常量 . . . 800
45.3 BigFloats and BigInts . . . 815

46 字符串 **819**

47 数组 **865**

47.1 构造函数与类型 . . . 865
47.2 基础函数 . . . 881
47.3 广播与矢量化 . . . 887
47.4 索引与赋值 . . . 892
47.5 Views (SubArrays 以及其它 view 类型) . . . 901
47.6 拼接与排列 . . . 909
47.7 数组函数 . . . 929
47.8 组合学 . . . 943

48 Tasks **949**

48.1 Scheduling . . . 954
48.2 Synchronization . . . 956

48.3 Channels . . . 963
48.4 Low-level synchronization using schedule and wait . . . 969
49 Multi-Threading 971
49.1 原子操作 . . . 976
49.2 ccall using a libuv threadpool (Experimental) . . . 985
49.3 Low-level synchronization primitives . . . 985
50 常量 986
51 文件系统 990
52 I/O 与网络 1013
52.1 通用 I/O . . . 1013
52.2 文本 I/O . . . 1036
52.3 多媒体 I/O . . . 1043
52.4 网络 I/O . . . 1048
53 运算符与记号 1050
54 排序及相关函数 1052
54.1 排序函数 . . . 1054
54.2 排列顺序相关的函数 . . . 1063
54.3 排序算法 . . . 1069
54.4 Alternate Orderings . . . 1070
55 迭代相关 1072
56 反射与自我检查 1085
56.1 模块绑定 . . . 1085
56.2 DateType 字段 . . . 1085
56.3 Subtypes . . . 1086
56.4 DataType 布局 . . . 1086
56.5 函数方法 . . . 1086
56.6 扩展和更底层 . . . 1086
56.7 中间表示和编译后表示 . . . 1087
57 C 接口 1088
58 LLVM 接口 1104
59 C 标准库 1105
60 堆栈跟踪 1109
61 SIMD 支持 1111

IV 标准库 1112
62 ArgTools 1113
62.1 Argument Handling . . . 1113
62.2 Function Testing . . . 1115
63 Artifacts 1117
64 Base64 1119
65 CRC32c 1122
66 日期 1123
66.1 构造函数 . . . 1123
66.2 Durations/Comparisons . . . 1125
66.3 Accessor Functions . . . 1127
66.4 Query Functions . . . 1128
66.5 TimeType-Period Arithmetic . . . 1129
66.6 Adjuster Functions . . . 1130
66.7 Period Types . . . 1132
66.8 Rounding . . . 1133

67 API reference 1135
67.1 Dates and Time Types . . . 1135
67.2 Dates Functions . . . 1137
68 分隔符文件 1164
69 Distributed Computing 1169
69.1 Cluster Manager Interface . . . 1187
70 Downloads 1191
71 文件相关事件 1195
72 Pidfile 1197
72.1 Primary Functions . . . 1197
72.2 Helper Functions . . . 1198
73 Future 1200
74 Interactive Utilities 1201
75 Lazy Artifacts 1210
76 LibCURL 1211
77 LibGit2 1212
78 动态链接器 1257
79 Linear Algebra 1261
79.1 特殊矩阵 . . . 1263
79.2 Matrix factorizations . . . 1266
79.3 Orthogonal matrices (`AbstractQ`) . . . 1266
79.4 Standard functions . . . 1268
79.5 Low-level matrix operations . . . 1369
79.6 BLAS functions . . . 1374
79.7 LAPACK functions . . . 1387
80 日志记录 1405
80.1 日志事件结构 . . . 1406
80.2 Processing log events . . . 1406
80.3 Testing log events . . . 1408
80.4 Environment variables . . . 1408
80.5 Examples . . . 1408
80.6 Reference . . . 1409
81 Markdown 1416
81.1 内联元素 . . . 1416
81.2 Toplevel elements . . . 1418
81.3 Markdown Syntax Extensions . . . 1422
82 内存映射 I/O 1424
83 Network Options 1427
84 Pkg 1431
85 Printf 1435
86 性能分析 1437
86.1 CPU Profiling . . . 1437
86.2 Via `@profile` . . . 1437
86.3 Triggered During Execution . . . 1437
86.4 Reference . . . 1438
86.5 Memory profiling . . . 1441
86.6 Heap Snapshots . . . 1443
87 The Julia REPL 1444
87.1 不同的提示符模式 . . . 1444
87.2 Key bindings . . . 1448
87.3 Tab completion . . . 1449
87.4 Customizing Colors . . . 1452

87.5 Changing the contextual module which is active at the REPL . . . 1453
87.6 Numbered prompt . . . 1454
87.7 TerminalMenus . . . 1455
87.8 References . . . 1457
88 随机数 1464
88.1 Random numbers module . . . 1465
88.2 Random generation functions . . . 1465
88.3 Subsequences, permutations and shuffling . . . 1469
88.4 Generators (creation and seeding) . . . 1472
88.5 Hooking into the Random API . . . 1476
89 Reproducibility 1483
90 SHA 1484
90.1 SHA functions . . . 1484
90.2 Working with context . . . 1488
90.3 HMAC functions . . . 1491
91 序列化 1497
92 共享数组 1499
93 套接字 1502
94 稀疏数组 1510
94.1 压缩稀疏列（CSC）稀疏矩阵存储 . . . 1510
94.2 稀疏向量储存 . . . 1511
94.3 稀疏向量与矩阵构造函数 . . . 1511
94.4 稀疏矩阵的操作 . . . 1513
94.5 Correspondence of dense and sparse methods . . . 1513
95 SparseArrays API 1514
96 Noteworthy external packages 1530
97 统计 1531
98 TOML 1541
98.1 Parsing TOML data . . . 1541
98.2 Exporting data to TOML file . . . 1542
98.3 References . . . 1543
99 Tar 1545
100 单元测试 1550
100.1 测试 Julia Base 库 . . . 1550
100.2 基本的单元测试 . . . 1550
100.3 Working with Test Sets . . . 1553
100.4 Testing Log Statements . . . 1557
100.5 Other Test Macros . . . 1560
100.6 Broken Tests . . . 1562
100.7 Test result types . . . 1563
100.8 Creating Custom `AbstractTestSet` Types . . . 1564
100.9 Test utilities . . . 1566
100.10 Workflow for Testing Packages . . . 1568
101 UUIDs 1571
102 Unicode 1573

V 开发者文档 1578
103 Documentation of Julia’s Internals 1579
103.1 Julia 运行时的初始化 . . . 1579
103.2 Julia 的 AST . . . 1582
103.3 More about types . . . 1595

103.4 Memory layout of Julia Objects . . . 1602
103.5 Julia 代码的 eval . . . 1605
103.6 Calling Conventions . . . 1610
103.7 本机代码生成过程的高级概述 . . . 1611
103.8 Julia 函数 . . . 1613
103.9 笛卡尔 . . . 1617
103.10 Talking to the compiler (the `:meta` mechanism) . . . 1622
103.11 子数组 . . . 1623
103.12 isbits Union Optimizations . . . 1627
103.13 System Image Building . . . 1628
103.14 Package Images . . . 1630
103.15 Working with LLVM . . . 1631
103.16 printf() and stdio in the Julia runtime . . . 1638
103.17 边界检查 . . . 1640
103.18 Proper maintenance and care of multi-threading locks . . . 1642
103.19 Arrays with custom indices . . . 1646
103.20 Module loading . . . 1650
103.21 类型推导 . . . 1650
103.22 Julia SSA-form IR . . . 1652
103.23 `EscapeAnalysis` . . . 1655
103.24 Static analyzer annotations for GC correctness in C code . . . 1668
103.25 Garbage Collection in Julia . . . 1673
103.26 Fixing precompilation hangs due to open tasks or IO . . . 1675

104 Developing/debugging Julia's C code 1678

104.1 报告和分析崩溃（段错误） . . . 1678
104.2 gdb 调试提示 . . . 1680
104.3 在 Julia 中使用 Valgrind . . . 1685
104.4 External Profiler Support . . . 1686
104.5 Sanitizer support . . . 1689
104.6 Instrumenting Julia with DTrace, and bpftrace . . . 1691

105 Building Julia 1698

105.1 Building Julia (Detailed) . . . 1698
105.2 Linux . . . 1706
105.3 macOS . . . 1707
105.4 Windows . . . 1707
105.5 tools . . . 1710
105.6 For 64 bit Julia, install x86_64 . . . 1710
105.7 For 32 bit Julia, install i686 . . . 1710
105.8 FreeBSD . . . 1712
105.9 ARM (Linux) . . . 1712
105.10 Binary distributions . . . 1714
105.11 Point releasing 101 . . . 1716

Part I

主页

Chapter 1

Julia 1.10 中文文档

欢迎来到 Julia 1.10 中文文档 (PDF 版本)!

请先阅读 Julia 1.0 正式发布博文 以获得对这门语言的总体概观。我们推荐刚刚开始学习 Julia 语言的朋友阅读中文社区提供的 Julia 入门指引，也推荐你在中文论坛对遇到的问题进行提问。

镜像加速

使用镜像站来加速下载几乎是每个国内用户都需要了解的事情，关于镜像站的使用说明及汇总可以在 Julia PkgServer 镜像服务及镜像站索引 中可以看到。

关于中文文档

Julia 语言相关的本地化工作是一个由社区驱动的开源项目 JuliaZH.jl，旨在方便 Julia 的中文用户。我们目前使用 Transifex 作为翻译平台。翻译工作正在进行，有任何疑问或建议请到社区论坛文档区反馈。若有意参与翻译工作，请参考翻译指南。

1.1 鸣谢

特别感谢集智俱乐部对Julia 中文文档和Julia 中文论坛相关服务器资源的赞助！

❤❤❤ 如果您或您所属的组织有意资助 Julia 中文社区的发展，欢迎前往论坛的社区板块发帖咨询。❤❤❤

1.2 其它中文资料

通用教程类：

- 北大李东风 Julia 语言入门
- Julia Roadmap 入门
- Julia Data Science 中文版

专题类：

- JuliaLogging 文档中文版

- Julia Pkg.jl 中文版
- 每个科研工作者在写高性能代码时都需了解的硬件知识

1.3 简介

科学计算对性能一直有着最高的需求，但目前各领域的专家却大量使用较慢的动态语言来开展他们的日常工作。偏爱动态语言有很多很好的理由，因此我们不会舍弃动态的特性。幸运的是，现代编程语言设计与编译器技术可以大大消除性能折衷（trade-off），并提供有足够生产力的单一环境进行原型设计，而且能高效地部署性能密集型应用程序。Julia 语言在这其中扮演了这样一个角色：它是一门灵活的动态语言，适合用于科学计算和数值计算，并且性能可与传统的静态类型语言媲美。

由于 Julia 的编译器和其它语言比如 Python 或 R 的解释器有所不同，一开始你可能发现 Julia 的性能并不是很突出。如果你觉得速度有点慢，我们强烈建议在尝试其他功能前，先读一读文档中的提高性能的窍门。在理解了 Julia 的运作方式后，写出和 C 一样快的代码对你而言就是小菜一碟。

1.4 Julia 与其他语言的比较

Julia 拥有可选类型标注和多重派发这两个特性，同时还拥有很棒的性能。这些都得归功于使用 LLVM 实现的类型推导和即时编译（JIT）技术 (和 可选的提前编译（AOT）)。Julia 是一门支持过程式、函数式和面向对象的多范式语言。它像 R、MATLAB 和 Python 一样简单，在高级数值计算方面有丰富的表现力，并且支持通用编程。为了实现这个目标，Julia 以数学编程语言（mathematical programming languages）为基础，同时也参考了不少流行的动态语言，例如 Lisp、Perl、Python、Lua、和 Ruby。

Julia 与传统动态语言最重要的区别是：

- 核心语言很小：标准库是用 Julia 自身写的，包括整数运算这样的基础运算
- 丰富的基础类型：既可用于定义和描述对象，也可用于做可选的类型标注
- 通过多重派发，可以根据类型的不同，来调用同名函数的不同实现
- 为不同的参数类型，自动生成高效、专用的代码
- 接近 C 语言的性能

尽管人们有时会说动态语言是“无类型的”，但实际上绝对不是这样的：每一个对象都有一个类型，无论它是基础的类型（primitive）还是用户自定义的类型。大多数的动态语言都缺乏类型声明，这意味着程序员无法告诉编译器值的类型，也就无法显式地讨论类型。另一方面，在静态语言中，往往必须标注对象的类型。但类型只在编译期才存在，而无法在运行时进行操作和表达。而在 Julia 中，类型本身是运行时的对象，并可用于向编译器传达信息。

是什么让 Julia 成为 Julia？

类型系统和多重派发是 Julia 语言最主要的特征，但一般不需要显式地手动标注或使用：函数通过函数名称和不同类型参数的组合进行定义，在调用时会派发到最接近（most specific）的定义上去。这样的编程模型非常适合数学化的编程，尤其是在传统的面向对象派发中，一些函数的第一个变量理论上并不“拥有”这样一个操作时。在 Julia 中运算符只是函数的一个特殊标记——例如，为用户定义的新类型添加加法运算，你只要为 + 函数定义一个新的方法就可以了。已有的代码就可以无缝接入这个新的类型。

Julia 在设计之初就非常看重性能，再加上它的动态类型推导（可以被可选的类型标注增强），使得 Julia 的计算性能超过了其它的动态语言，甚至能够与静态编译语言竞争。对于大型数值问题，速度一直都是，也一直会是一个重要的关注点：在过去的几十年里，需要处理的数据量很容易与摩尔定律保持同步。

Julia 的优势

Julia 的目标是创建一个前所未有的集易用、强大、高效于一体的语言。除此之外，Julia 还拥有以下优势：

- 采用 MIT 许可证：免费又开源
- 用户自定义类型的速度与兼容性和内建类型一样好
- 无需特意编写向量化的代码：非向量化的代码就很快
- 为并行计算和分布式计算设计
- 轻量级的“绿色”线程：协程
- 低调又牛逼的类型系统
- 优雅、可扩展的类型转换和类型提升
- 对 Unicode 的有效支持，包括但不限于 UTF-8
- 直接调用 C 函数，无需封装或调用特别的 API
- 像 Shell 一样强大的管理其他进程的能力
- 像 Lisp 一样的宏和其他元编程工具

Part II

手册

Chapter 2

入门

无论是使用预编译好的二进制程序，还是自己从源码编译，安装 Julia 都是一件很简单的事情。请按照 https://julialang.org/downloads/ 的提示来下载并安装 Julia。

如果你是从下面的某一种语言切换到 Julia 的话，那么你应该首先阅读与这些语言有显著差异的那一部分 MATLAB, R, Python, C/C++ or Common Lisp. 这将帮助你避免一些常见的编程陷阱，因为 Julia 在许多微妙的方面与这些语言不同。

启动一个交互式会话（也叫 REPL）是学习和尝试 Julia 最简单的方法。双击 Julia 的可执行文件或是从命令行运行 julia 就可以启动：

```
$ julia

               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.10.9 (2025-03-10)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |


julia> 1 + 2
3

julia> ans
3
```

输入 CTRL-D（同时按 Ctrl 键和 d 键）或 exit() 便可以退出交互式会话。在交互式模式中，julia 会显示一条横幅并提示用户输入。一旦用户输入了一段完整的代码（表达式），例如 1 + 2，然后按回车，交互式会话就会执行这段代码，并将结果显示出来。如果输入的代码以分号结尾，那么结果将不会显示出来。然而不管结果显示与否，变量 ans 总会存储上一次执行代码的结果，需要注意的是，变量 ans 只在交互式会话中才有。

在交互式会话中，要运行写在源文件 file.jl 中的代码，只需输入 include("file.jl")。

如果想以非交互的方式执行文件中的代码，可以把文件名作为 julia 命令的第一个参数：

```
$ julia script.jl
```

您可以向 Julia 和您的程序 `script.jl` 传递额外的参数。所有可用选项的详细列表请参见 命令行界面。

2.1 资源

除了本手册以外，官方网站还提供了一个有用的 **学习资源列表** 来帮助新用户学习 Julia。

您可以切换到帮助模式，将 REPL 当作一种学习资源。在空的 `julia>` 提示符下按 `?` 即可切换到帮助模式。在帮助模式下输入关键字或函数名，就能获取相关文档和示例。对于大多数函数或其他对象也是如此！

```
help?> begin
search: begin disable_sigint reenable_sigint

  begin

  begin...end denotes a block of code.
```

如果已经对 Julia 有所了解，可以先看 性能提示 和 工作流程提示。

Chapter 3

变量

Julia 语言中，变量是与某个值相关联（或绑定）的名字。你可以用它来保存一个值（例如某些计算得到的结果），供之后的代码使用。例如：

```
# 将 10 赋值给变量 x
julia> x = 10
10

# 使用 x 的值做计算
julia> x + 1
11

# 重新给 x 赋值
julia> x = 1 + 1
2

# 也可以给 x 赋其它类型的值，比如字符串文本
julia> x = "Hello World!"
"Hello World!"
```

Julia 提供了非常灵活的变量命名策略。变量名是大小写敏感的，且不包含语义，意思是说，Julia 不会根据变量的名字来区别对待它们。（译者注：Julia **不会**自动将全大写的变量识别为常量，也**不会**将有特定前后缀的变量自动识别为某种特定类型的变量，即不会根据变量名字，自动判断变量的任何属性。）

```
julia> x = 1.0
1.0

julia> y = -3
-3

julia> Z = "My string"
"My string"

julia> customary_phrase = "Hello world!"
"Hello world!"

julia> UniversalDeclarationOfHumanRightsStart = "人人生而自由，在尊严和权利上一律平等。"
"人人生而自由，在尊严和权利上一律平等。"
```

你还可以使用 UTF-8 编码的 Unicode 字符作为变量名：

```
julia> δ = 0.00001
1.0e-5

julia> 안녕하세요 = "Hello"
"Hello"
```

在 Julia REPL 和一些其它 Julia 的编辑器中，很多 Unicode 数学符号可以使用反斜杠加 LaTeX 符号接 tab 健打出。例如：变量名 δ 可以通过 `\delta`-*tab* 来输入，甚至可以用 `\alpha`-*tab*-`\hat`-*tab*-`\^(2)`-*tab* 来输入 $\hat{\alpha}^{(2)}$ 这种复杂的变量名。（如果你在某个地方发现了一个不知道怎么输入的符号，比如在别人的代码里，输入? 接着复制那个符号，REPL 的帮助功能会告诉你输入方法。）

如果有需要的话，Julia 甚至允许你重定义内置常量和函数。（这样做可能引发潜在的混淆，所以并不推荐）

```
julia> pi = 3
3

julia> pi
3

julia> sqrt = 4
4
```

然而，如果你试图重定义一个已经在使用中的内置常量或函数，Julia 会报错：

```
julia> pi
π = 3.1415926535897...

julia> pi = 3
ERROR: cannot assign a value to imported variable Base.pi from module Main

julia> sqrt(100)
10.0

julia> sqrt = 4
ERROR: cannot assign a value to imported variable Base.sqrt from module Main
```

3.1 合法的变量名

变量名字必须以英文字母（A-Z 或 a-z）、下划线或编码大于 00A0 的 Unicode 字符的一个子集开头。具体来说指的是，Unicode 字符分类中的 Lu/Ll/Lt/Lm/Lo/Nl（字母）、Sc/So（货币和其他符号）以及一些其它像字母的符号（例如 Sm 类别数学符号中的一部分）。变量名的非首字符还允许使用惊叹号 `!`、数字（包括 0-9 和其他 Nd/No 类别中的 Unicode 字符）以及其它 Unicode 字符：变音符号和其他修改标记（Mn/Mc/Me/Sk 类别）、标点和连接符（Pc 类别）、引号和少许其他字符。

像 `+` 这样的运算符也是合法的标识符，但是它们会被特别地解析。在一些上下文中，运算符可以像变量一样使用，比如 `(+)` 表示加函数，语句 `(+) = f` 会把它重新赋值。大部分 Unicode 中缀运算符（Sm 类别），像 `⊕`，会被解析成真正的中缀运算符，并且支持用户自定义方法（举个例子，你可以使用语句 `const ⊗ = kron` 将 `⊗` 定义为中缀的 Kronecker 积）。运算符也可以使用修改标记、引号和上标/下标进行加缀，例如 $\hat{+}_a''$ 被解析成一个与 `+` 具有相同优先级的中缀运算符。以下标/上标字母结尾的运

算符与后续变量名之间需要一个空格。举个例子，如果 +ᵃ 是一个运算符，那么 +ᵃx 应该被写为 +ᵃ x，以区分表达式 + ᵃx ，其中 ᵃx 是变量名。

一类特定的变量名是只包含下划线的变量名。这些标识符只能赋值，不能用于给其他变量赋值。这些标识符只能赋值，赋值后会立即丢弃，因此不能用于为其他变量赋值。严格来说，它们只能用作左值 (rvalues) 而不能作右值。

```
julia> x, ___ = size([2 2; 1 1])
(2, 2)

julia> y = ___
ERROR: syntax: all-underscore identifier used as rvalue

julia> println(___)
ERROR: syntax: all-underscore identifier used as rvalue
```

唯一明确禁止使用的变量名是内置的关键字：

```
julia> else = false
ERROR: syntax: unexpected "else"

julia> try = "No"
ERROR: syntax: unexpected "="
```

一些 Unicode 字符在标识符中被视为等价的。输入 Unicode 组合字符的不同方式被视为等价的（例如：重音符号）。（具体来说，Julia 标识符遵循 NFC 规范）。

Julia 还包含一些非标准的等价关系，适用于那些视觉上相似、且可以通过某些输入方法轻松输入的字符。

- Unicode 字符 ɛ（U+025B: Latin small letter open e）和 µ（U+00B5: micro sign）被视为与相应的希腊字母等价。
- 中间点 ·（U+00B7）和希腊间隔点 ·（U+0387）都被视为等同于数学点运算符 ⋅（U+22C5）。
- 减号 −（U+2212）被视为与连字符减号 -（U+002D）等价。

3.2 赋值表达式与赋值和修改的区别

赋值 variable = value 将名称 variable " **绑定**" 到右侧计算得到的 value 上，整个赋值被 Julia 视为一个等于右侧 value 的表达式。这意味着赋值可以被链式使用（同一个 value 可以通过 variable1 = variable2 = value 赋值给多个变量）或在其他表达式中使用，这也是为什么它们的结果在 REPL 中显示为右值（RHS, right-hand side）。（一般来说，REPL 会显示你所计算的任何表达式的值）

例如：这里 b = 2+2 的值 4 被用于另一个算术运算和赋值。

```
julia> a = (b = 2+2) + 3
7

julia> a
7

julia> b
4
```

一个常见的混淆点是赋值（给一个值赋予新的" 名称"）和修改（改变一个值）之间的区别。如果你先执行 a = 2 然后执行 a = 3，你改变的是" 名称" a 使其指向新值 3。你并没有改变数字 2，所以 2+2 仍然会得到 4 而不是 6！

当处理可变类型如数组时，这种区别变得更加明显，因为数组的内容可以被改变：

```
julia> a = [1,2,3]  # 一个包含 3 个整数的数组
3-element Vector{Int64}:
 1
 2
 3

julia> b = a   # b 和 a 是同一个数组的名称!
3-element Vector{Int64}:
 1
 2
 3
```

这里，b = a 这行代码并不会复制数组 a，它只是将名称 b 绑定到同一个数组 a 上：b 和 a 都" 指向" 内存中的同一个数组 [1,2,3]。

相比之下，赋值 a[i] = value 改变了数组的内容，且修改后的数组可以通过名称 a 和 b 访问：

```
julia> a[1] = 42     # 修改第一个元素
42

julia> a = 3.14159   # a 现在指向另一个不同的对象
3.14159

julia> b   # b 指向的是经过修改的原始数组对象
3-element Vector{Int64}:
 42
  2
  3
```

也就是说：a[i] = value（setindex! 的别名）修改了内存中已存在的数组对象，可以通过 a 或 b 访问该对象。随后设置 a = 3.14159 并不会改变这个数组，它只是将 a 绑定到一个不同的对象上；原始数组仍然可以通过 b 访问。另一种常见的修改现有对象的语法是 a.field = value（setproperty! 的别名），它可以用来修改 mutable struct。

当你在 Julia 中调用函数时，它的行为就像你将参数值赋值给与函数参数对应的新变量名一样，这在参数传递行为中有所讨论。（按照惯例，修改一个或多个输入参数的函数名以 ! 结尾。）

3.3 命名规范

虽然 Julia 语言对合法名字的限制非常少，但是遵循以下这些命名规范是非常有用的：

- 变量的名字采用小写。
- 使用下划线（'_'）来分隔名字中的单词，但是不鼓励使用下划线，除非在不使用下划线时名字会非常难读。
- 类型 (Type) 和模块（Module）的名字使用大写字母开头，并且用大写字母而不是用下划线分隔单词。

- 函数（`function`）和宏（`macro`）的名字使用小写，不使用下划线。
- 会对输入参数进行更改的函数要使用！结尾。这些函数有时叫做“mutating”或“in-place”函数，因为它们在被调用后会修改他们的输入参数的内容而不仅仅只是返回一个值。

关于命名规范的更多信息，可查看代码风格指南。

Chapter 4

整数和浮点数

整数和浮点值是算术和计算的基础。这些数值的内置表示被称作原始数值类型（numeric primitive），且整数和浮点数在代码中作为立即数时称作数值字面量（numeric literal）。例如，`1` 是个整型字面量，`1.0` 是个浮点型字面量，它们在内存中作为对象的二进制表示就是原始数值类型。

Julia 提供了很丰富的原始数值类型，并基于它们定义了一整套算术操作，还提供按位运算符以及一些标准数学函数。这些函数能够直接映射到现代计算机原生支持的数值类型及运算上，因此 Julia 可以充分地利用运算资源。此外，Julia 还为任意精度算术提供了软件支持，对于无法使用原生硬件表示的数值类型，Julia 也能够高效地处理其数值运算。当然，这需要相对的牺牲一些性能。

以下是 Julia 的原始数值类型：

- **整数类型**：

类型	带符号？	比特数	最小值	最大值
`Int8`	✓	8	-2^7	2^7 - 1
`UInt8`		8	0	2^8 - 1
`Int16`	✓	16	-2^15	2^15 - 1
`UInt16`		16	0	2^16 - 1
`Int32`	✓	32	-2^31	2^31 - 1
`UInt32`		32	0	2^32 - 1
`Int64`	✓	64	-2^63	2^63 - 1
`UInt64`		64	0	2^64 - 1
`Int128`	✓	128	-2^127	2^127 - 1
`UInt128`		128	0	2^128 - 1
`Bool`	N/A	8	false (0)	true (1)

- **浮点类型:**

类型	精度	比特数
`Float16`	半精度	16
`Float32`	单精度	32
`Float64`	双精度	64

此外，对复数和有理数的完整支持是在这些原始数据类型之上建立起来的。多亏了 Julia 有一个很灵活的、用户可扩展的类型提升系统，所有的数值类型都无需显式转换就可以很自然地相互进行运算。

4.1 整数

整数字面量以标准形式表示：

```
julia> 1
1

julia> 1234
1234
```

整型字面量的默认类型取决于目标系统是 32 位还是 64 位架构：

```
# 32 位系统:
julia> typeof(1)
Int32

# 64 位系统:
julia> typeof(1)
Int64
```

Julia 的内置变量 `Sys.WORD_SIZE` 表明了目标系统是 32 位还是 64 位架构：

```
# 32 位系统:
julia> Sys.WORD_SIZE
32

# 64 位系统:
julia> Sys.WORD_SIZE
64
```

Julia 也定义了 `Int` 与 `UInt` 类型，它们分别是系统有符号和无符号的原生整数类型的别名。

```
# 32 位系统:
julia> Int
Int32
julia> UInt
UInt32

# 64 位系统:
julia> Int
Int64
julia> UInt
UInt64
```

那些超过 32 位表示范围的大整数，如果能用 64 位表示，那么无论是什么系统都会用 64 位表示：

```
# 32 位或 64 位系统:
julia> typeof(3000000000)
Int64
```

无符号整数会通过 `0x` 前缀以及十六进制数 `0-9a-f` 来输入和输出（输入也可以使用大写的 `A-F`）。无符号值的位数取决于十六进制数字使用的数量：

```
julia> x = 0x1
0x01

julia> typeof(x)
UInt8

julia> x = 0x123
0x0123

julia> typeof(x)
UInt16

julia> x = 0x1234567
0x01234567

julia> typeof(x)
UInt32

julia> x = 0x123456789abcdef
0x0123456789abcdef

julia> typeof(x)
UInt64

julia> x = 0x11112222333344445555666677778888
0x11112222333344445555666677778888

julia> typeof(x)
UInt128
```

采用这种做法是因为，当人们使用无符号十六进制字面量表示整数值的时候，通常会用它们来表示一个固定的数值字节序列，而不仅仅是个整数值。

二进制和八进制字面量也是支持的：

```
julia> x = 0b10
0x02

julia> typeof(x)
UInt8

julia> x = 0o010
0x08

julia> typeof(x)
UInt8

julia> x = 0x00000000000000001111222233334444
0x00000000000000001111222233334444

julia> typeof(x)
UInt128
```

与十六进制字面量一样，二进制、八进制和十六进制的字面量都会产生无符号的整数类型。当字面量不是开头全是 0 时，它们二进制数据项的位数会是最少需要的位数。在前导为零的情况下，大小由具有相同长度但前导数字为 1 的字面量所需的最小大小决定。这意味着：

- `0x1` 和 `0x12` 是 `UInt8` 字面量
- `0x123` 和 `0x1234` 是 `UInt16` 字面量
- `0x12345` 和 `0x12345678` 是 `UInt32` 字面量
- `0x123456789` 和 `0x1234567890adcdef` 是 `UInt64` 字符串字面量

即使是不影响数值的前导零，它们也会影响字面量的类型。因此 `0x01` 是 `UInt8` 而 `0x0001` 是 `UInt16`。

这允许用户控制字面量的类型和大小。

无符号字面量（以 `0x` 开头）如果编码的整数太大而无法表示为 `UInt128` 值时，将会构造 `BigInt` 值。这不是无符号类型，但它是唯一一个足够大能表示如此大整数值的内置类型。

二进制、八进制和十六进制的字面量前面加一个负号 `-`，这样可以产生一个和原字面量有着同样位数而值为原数的补码的数（二补数）：

```
julia> -0x2
0xfe

julia> -0x0002
0xfffe
```

整型等原始数值类型的最小和最大可表示的值可用 `typemin` 和 `typemax` 函数得到：

```
julia> (typemin(Int32), typemax(Int32))
(-2147483648, 2147483647)

julia> for T in [Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128]
           println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
       end
   Int8: [-128,127]
  Int16: [-32768,32767]
  Int32: [-2147483648,2147483647]
  Int64: [-9223372036854775808,9223372036854775807]
 Int128: [-170141183460469231731687303715884105728,170141183460469231731687303715884105727]
  UInt8: [0,255]
 UInt16: [0,65535]
 UInt32: [0,4294967295]
 UInt64: [0,18446744073709551615]
UInt128: [0,340282366920938463463374607431768211455]
```

`typemin` 和 `typemax` 返回的值的类型总与所给参数的类型相同。（上面的表达式用了一些目前还没有介绍的功能，包括 for 循环、字符串和字符串插值，但这对于已有一些编程经验的用户应该是很容易理解的。）

溢出行为

在 Julia 里，超出一个类型可表示的最大值会导致环绕 (wraparound) 行为：

```
julia> x = typemax(Int64)
9223372036854775807

julia> x + 1
-9223372036854775808

julia> x + 1 == typemin(Int64)
true
```

因此，Julia 的整数算术实际上是模算数的一种形式，它反映了现代计算机实现底层算术的特点。在可能有溢出产生的程序中，对最值边界出现循环进行显式检查是必要的。否则，推荐使用任意精度算术中的 BigInt 类型作为替代。

下面是溢出行为的一个例子以及如何解决溢出：

```
julia> 10^19
-8446744073709551616

julia> big(10)^19
10000000000000000000
```

除法错误

div 函数的整数除法有两种异常情况：除以零，以及使用 -1 去除最小的负数（typemin）。这两种情况都会抛出一个 DivideError 错误。rem 取余函数和 mod 取模函数在除零时抛出 DivideError 错误。

4.2 浮点数

浮点数字面量以标准格式表示，必要时可使用E-表示法。

```
julia> 1.0
1.0

julia> 1.
1.0

julia> 0.5
0.5

julia> .5
0.5

julia> -1.23
-1.23

julia> 1e10
1.0e10

julia> 2.5e-4
0.00025
```

上面的结果都是 Float64 类型的值。使用 f 替代 e 可以得到 Float32 类型的字面量：

```
julia> x = 0.5f0
0.5f0

julia> typeof(x)
Float32

julia> 2.5f-4
0.00025f0
```

数值可以很容易地转换为 Float32 类型：

```
julia> x = Float32(-1.5)
-1.5f0

julia> typeof(x)
Float32
```

也存在十六进制的浮点数字面量，但只适用于 Float64 类型的值。一般使用 p 前缀及以 2 为底的指数来表示：

```
julia> 0x1p0
1.0

julia> 0x1.8p3
12.0

julia> x = 0x.4p-1
0.125

julia> typeof(x)
Float64
```

Julia 也支持半精度浮点数（Float16），但它们是使用 Float32 进行软件模拟实现的。

```
julia> sizeof(Float16(4.))
2

julia> 2*Float16(4.)
Float16(8.0)
```

下划线 _ 可用作数字分隔符：

```
julia> 10_000, 0.000_000_005, 0xdead_beef, 0b1011_0010
(10000, 5.0e-9, 0xdeadbeef, 0xb2)
```

浮点数中的零

浮点数有两种零，正零和负零。它们相互相等但有着不同的二进制表示，可以使用 bitstring 函数来查看：

```
julia> 0.0 == -0.0
true

julia> bitstring(0.0)
"0000000000000000000000000000000000000000000000000000000000000000"

julia> bitstring(-0.0)
"1000000000000000000000000000000000000000000000000000000000000000"
```

特殊的浮点值

有三种特定的标准浮点值不和实数轴上任何一点对应：

Float16	Float32	Float64	名称	描述
Inf16	Inf32	Inf	正无穷	一个大于所有有限浮点数的数
-Inf16	-Inf32	-Inf	负无穷	一个小于所有有限浮点数的数
NaN16	NaN32	NaN	不是数（Not a Number）	一个不和任何浮点值（包括自己）相等（==）的值

对于这些非有限浮点值相互之间以及关于其它浮点值的顺序的更多讨论，请参见数值比较。根据 IEEE 754 标准，这些浮点值是某些算术运算的结果：

```
julia> 1/Inf
0.0

julia> 1/0
Inf

julia> -5/0
-Inf

julia> 0.000001/0
Inf

julia> 0/0
NaN

julia> 500 + Inf
Inf

julia> 500 - Inf
-Inf

julia> Inf + Inf
Inf

julia> Inf - Inf
NaN

julia> Inf * Inf
Inf

julia> Inf / Inf
```

```
NaN

julia> 0 * Inf
NaN

julia> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

`typemin` 和 `typemax` 函数同样适用于浮点类型：

```
julia> (typemin(Float16),typemax(Float16))
(-Inf16, Inf16)

julia> (typemin(Float32),typemax(Float32))
(-Inf32, Inf32)

julia> (typemin(Float64),typemax(Float64))
(-Inf, Inf)
```

机器精度

大多数实数都无法用浮点数准确地表示，因此有必要知道两个相邻可表示的浮点数间的距离。它通常被叫做机器精度。

Julia 提供了 `eps` 函数，它可以给出 `1.0` 与下一个 Julia 能表示的浮点数之间的差值：

```
julia> eps(Float32)
1.1920929f-7

julia> eps(Float64)
2.220446049250313e-16

julia> eps() # 与 eps(Float64) 相同
2.220446049250313e-16
```

这些值分别是 `Float32` 中的 `2.0^-23` 和 `Float64` 中的 `2.0^-52`。`eps` 函数也可以接受一个浮点值作为参数，然后给出这个值与下一个可表示的浮点数值之间的绝对差。也就是说，`eps(x)` 产生一个和 `x` 类型相同的值，并且 `x + eps(x)` 恰好是比 `x` 更大的下一个可表示的浮点值：

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(1000.)
```

```
1.1368683772161603e-13

julia> eps(1e-27)
1.793662034335766e-43

julia> eps(0.0)
5.0e-324
```

两个相邻可表示的浮点数之间的距离并不是常数，数值越小，间距越小，数值越大，间距越大。换句话说，可表示的浮点数在实数轴上的零点附近最稠密，并沿着远离零点的方向以指数型的速度变得越来越稀疏。根据定义，`eps(1.0)` 与 `eps(Float64)` 相等，因为 `1.0` 是个 64 位浮点值。

Julia 也提供了 `nextfloat` 和 `prevfloat` 两个函数分别返回基于参数的下一个更大或更小的可表示的浮点数：

```
julia> x = 1.25f0
1.25f0

julia> nextfloat(x)
1.2500001f0

julia> prevfloat(x)
1.2499999f0

julia> bitstring(prevfloat(x))
"00111111100111111111111111111111"

julia> bitstring(x)
"00111111101000000000000000000000"

julia> bitstring(nextfloat(x))
"00111111101000000000000000000001"
```

这个例子体现了一般原则，即相邻可表示的浮点数也有着相邻的二进制整数表示。

舍入模式

一个数如果没有精确的浮点表示，就必须被舍入到一个合适的可表示的值。然而，如果想的话，可以根据舍入模式改变舍入的方式，如 IEEE 754 标准 所述。

Julia 所使用的默认模式总是 `RoundNearest`，指舍入到最接近的可表示的值，这个被舍入的值会使用尽量少的有效位数。

背景知识与参考文献

浮点算术带来了很多微妙之处，它们可能对于那些不熟悉底层实现细节的用户会是很出人意料的。然而，这些微妙之处在大部分科学计算的书籍中以及以下的参考资料中都有详细介绍:

- 浮点数算术的权威指南是 IEEE 754-2008 标准；然而这篇标准在网上无法免费获得。
- 关于浮点数是如何表示的，想要一个简单而明白的介绍的话，可以看 John D. Cook 的文章 以及他关于从这种表示与实数理想的抽象化的差别中产生的一些问题的介绍

- 同样推荐 Bruce Dawson 的一系列关于浮点数的博客文章。
- 想要一个对浮点数和使用浮点数计算时产生的数值精度问题的极好的、有深度的讨论，可以参见 David Goldberg 的文章 What Every Computer Scientist Should Know About Floating-Point Arithmetic。
- 更多延伸文档，包括浮点数的历史、基础理论、问题以及数值计算中很多其它主题的讨论，可以参见 William Kahan 的写作集。他以“浮点数之父”闻名。特别感兴趣的话可以看 An Interview with the Old Man of Floating-Point。

4.3 任意精度算术

为了允许使用任意精度的整数与浮点数，Julia 分别包装了 GNU Multiple Precision Arithmetic Library (GMP) 以及 GNU MPFR Library。Julia 中的 `BigInt` 与 `BigFloat` 两种类型分别提供了任意精度的整数和浮点数。

可以使用构造函数从原始数据类型创建任意精度的整数和浮点数，或是使用字符串字面量 `@big_str`、`parse`函数从 `AbstractString` 类型来创建。当创建的整型字面量过大而无法被其它内置的整数类型表示时，也会被输入为 `BigInt` 类型。注意到 Julia 的 `Base` 中并没有无符号的任意精度的整型（`BigInt` 在多数情况下已经够用了），（除十进制外）也可以使用十六进制、八进制、二进制的字面量。

得益于 Julia 的类型提升和类型转换机制，一旦被创建，它们就可以像其它数值类型一样参与算术运算。

```
julia> BigInt(typemax(Int64)) + 1
9223372036854775808

julia> big"123456789012345678901234567890" + 1
123456789012345678901234567891

julia> parse(BigInt, "123456789012345678901234567890") + 1
123456789012345678901234567891

julia> string(big"2"^200, base=16)
"100000000000000000000000000000000000000000000000000"

julia> 0x100000000000000000000000000000000-1 == typemax(UInt128)
true

julia> 0x000000000000000000000000000000000
0

julia> typeof(ans)
BigInt

julia> big"1.23456789012345678901"
1.234567890123456789010000000000000000000000000000000000000000000000000000004

julia> parse(BigFloat, "1.23456789012345678901")
1.234567890123456789010000000000000000000000000000000000000000000000000000004

julia> BigFloat(2.0^66) / 3
2.459565876494606882133333333333333333333333333333333333333333333333333333344e+19

julia> factorial(BigInt(40))
815915283247897734345611269596115894272000000000
```

然而，上面的原始类型与 `BigInt`/`BigFloat` 之间的类型提升并不是自动的，需要明确地指定：

```
julia> x = typemin(Int64)
-9223372036854775808

julia> x = x - 1
9223372036854775807

julia> typeof(x)
Int64

julia> y = BigInt(typemin(Int64))
-9223372036854775808

julia> y = y - 1
-9223372036854775809

julia> typeof(y)
BigInt
```

`BigFloat` 的默认精度（有效数字的位数）和舍入模式可以通过调用 `setprecision` 和 `setrounding` 来全局地改变，所有之后的计算都会根据这些改变进行。还有一种方法，可以使用同样的函数以及 do-block 来只在运行一个特定代码块时改变精度和舍入模式：

```
julia> setrounding(BigFloat, RoundUp) do
           BigFloat(1) + parse(BigFloat, "0.1")
       end
1.100000000000000000000000000000000000000000000000000000000000000000000000000003

julia> setrounding(BigFloat, RoundDown) do
           BigFloat(1) + parse(BigFloat, "0.1")
       end
1.099999999999999999999999999999999999999999999999999999999999999999999999999986

julia> setprecision(40) do
           BigFloat(1) + parse(BigFloat, "0.1")
       end
1.1000000000004
```

4.4 数值字面量系数

为了让常见的数值公式和表达式更清楚，Julia 允许变量直接跟在一个数值字面量后，暗指乘法。这可以让写多项式变得很清楚：

```
julia> x = 3
3

julia> 2x^2 - 3x + 1
10

julia> 1.5x^2 - .5x + 1
13.0
```

也会让写指数函数变得更加优雅：

```
julia> 2^2x
64
```

数值字面量系数的优先级跟一元运算符相同，比如说取相反数。所以 `2^3x` 会被解析成 `2^(3x)`，而 `2x^3` 会被解析成 `2*(x^3)`。

数值字面量也能作为被括号表达式的系数：

```
julia> 2(x-1)^2 - 3(x-1) + 1
3
```

Note

用于隐式乘法的数值字面量系数的优先级高于其它的二元运算符，例如乘法（`*`）和除法（`/`、`\` 以及 `//`）。这意味着，比如说，`1 / 2im` 等于 `-0.5im` 以及 `6 // 2(2+1)` 等于 `1 // 1`。

此外，括号表达式可以被用作变量的系数，暗指表达式与变量相乘：

```
julia> (x-1)x
6
```

但是，无论是把两个括号表达式并列，还是把变量放在括号表达式之前，都不会被用作暗指乘法：

```
julia> (x-1)(x+1)
ERROR: MethodError: objects of type Int64 are not callable

julia> x(x+1)
ERROR: MethodError: objects of type Int64 are not callable
```

这两种表达式都会被解释成函数调用：所有不是数值字面量的表达式，后面紧跟一个括号，就会被解释成使用括号内的值来调用函数（更多关于函数的信息请参见函数）。因此，在这两种情况中，都会因为左边的值并不是函数而产生错误。

上述的语法糖显著地降低了在写普通数学公式时的视觉干扰。注意数值字面量系数和后面用来相乘的标识符或括号表达式之间不能有空格。

语法冲突

并列的字面量系数语法可能和两种数值字面量语法产生冲突：十六进制、八进制、二进制整数字面量以及浮点字面量的工程表示法。下面是几种会产生语法冲突的情况：

- 十六进制整数字面量 `0xff` 可能被解释成数值字面量 `0` 乘以变量 `xff`。类似的，像 `0o777` 或 `0b01001010` 使用八进制或二进制表示法也会形成冲突。
- 浮点字面量表达式 `1e10` 可以被解释成数值字面量 `1` 乘以变量 `e10`，与之等价的 `E`-表示法也存在类似的情况。
- 32-bit 的浮点数字面量 `1.5f22` 被解释成数值字面量 `1.5` 乘以变量 `f22`。

在这些所有的情况中，歧义都优先解释为数值字面量：

- `0x` /`0o`/`0b` 开头的表达式总是十六进制/八进制/二进制字面量。
- 数值开头跟着 `e` 和 `E` 的表达式总是浮点字面量。
- 数值开头跟着 `f` 的表达式总是 32-bit 浮点字面量。

由于历史原因 `E` 和 `e` 在数值字面量上是等价的，与之不同的是，`F` 只是一个行为和 `f` 不同的字母。因此开头为 `F` 的表达式将会被解析为一个数值字面量乘以一个变量，例如 `1.5F22` 等价于 `1.5 * F22`。

4.5 零和一的字面量

Julia 提供了 0 和 1 的字面量函数，可以返回特定类型或所给变量的类型。

函数	描述
`zero(x)`	`x` 类型或变量 `x` 的类型的零字面量
`one(x)`	`x` 类型或变量 `x` 的类型的一字面量

这些函数在数值比较中可以用来避免不必要的类型转换带来的开销。

例如：

```
julia> zero(Float32)
0.0f0

julia> zero(1.0)
0.0

julia> one(Int32)
1

julia> one(BigFloat)
1.0
```

Chapter 5

数学运算和初等函数

Julia 为它所有的基础数值类型，提供了整套的基础算术和位运算，也提供了一套高效、可移植的标准数学函数。

5.1 算术运算符

以下算术运算符支持所有的原始数值类型：

表达式	名称	描述
`+x`	一元加法运算符	全等操作
`-x`	一元减法运算符	将值变为其相反数
`x + y`	二元加法运算符	执行加法
`x - y`	二元减法运算符	执行减法
`x * y`	乘法运算符	执行乘法
`x / y`	除法运算符	执行除法
`x ÷ y`	整除	取 x / y 的整数部分
`x \ y`	反向除法	等价于 `y / x`
`x ^ y`	幂操作符	x 的 y 次幂
`x % y`	取余	等价于 `rem(x,y)`

除了优先级比二元操作符高以外，直接放在标识符或括号前的数字，如 `2x` 或 `2(x+y)` 还会被视为乘法。详见数值字面量系数。

Julia 的类型提升系统使得混合参数类型上的代数运算也能顺其自然的工作，请参考类型提升系统来了解更多内容。

符号 ÷ 可以通过输入 `\div<tab>` 到 REPL 或 Julia IDE 的方式来打出. 更多信息参见 Unicode 输入表。

这里是使用算术运算符的一些简单例子：

```
julia> 1 + 2 + 3
6

julia> 1 - 2
-1

julia> 3*2/12
0.5
```

习惯上我们会把优先运算的操作符紧邻操作数，比如 `-x + 2` 表示先要给 `x` 取反，然后再加 `2` 。

在乘法操作中，`false` 被视作 零。

```
julia> NaN * false
0.0

julia> false * Inf
0.0
```

这在已知某些量为零时，可以避免 `NaN` 的传播。详细的动机参见：Knuth (1992)。

5.2 布尔运算符

`Bool` 类型支持以下布尔运算符：

表达式	名称
`!x`	否定
`x && y`	短路与
`x || y`	短路或

否定将 `true` 更改为 `false`，反之亦然。链接页面上解释了逻辑短路。

请注意，`Bool` 是一个整数类型，所有常用的类型提升规则和数字运算符仍然对它适用。

5.3 位运算符

所有原始整数类型都支持以下位运算符：

表达式	名称
`~x`	按位取反
`x & y`	按位与
`x | y`	按位或
`x ⊻ y`	按位异或（逻辑异或）
`x ⊼ y`	按位与（非与）
`x ⊽ y`	按位或（非或）
`x >>> y`	逻辑右移
`x >> y`	算术右移
`x << y`	逻辑/算术左移

以下是位运算符的一些示例：

```
julia> ~123
-124

julia> 123 & 234
106

julia> 123 | 234
251
```

```
julia> 123 ⊻ 234
145

julia> xor(123, 234)
145

julia> nand(123, 123)
-124

julia> 123 ⊼ 123
-124

julia> nor(123, 124)
-128

julia> 123 ⊽ 124
-128

julia> ~UInt32(123)
0xffffff84

julia> ~UInt8(123)
0x84
```

5.4 复合赋值运算符

每一个二元运算符和位运算符都可以给左操作数复合赋值：方法是把 `=` 直接放在二元运算符后面。比如，`x += 3` 等价于 `x = x + 3` 。

```
julia> x = 1
1

julia> x += 3
4

julia> x
4
```

二元运算符和位运算符的复合赋值操作符有下面几种：

```
+=  -=  *=  /=  \=  ÷=  %=  ^=  &=  |=  ⊻=  >>>=  >>=  <<=
```

Note

复合赋值后会把变量重新绑定到左操作数上，所以变量的类型可能会改变。

```
julia> x = 0x01; typeof(x)
UInt8

julia> x *= 2 # 与 x = x * 2 相同
2

julia> typeof(x)
Int64
```

5.5 向量化“点”运算符

Julia 中，**每个**二元运算符都有一个“点”运算符与之对应，例如 ^ 就有对应的 .^ 存在。这个对应的 .^ 被 Julia **自动地**定义为逐元素地执行 ^ 运算。比如 [1,2,3] ^ 3 是非法的，因为数学上没有给（长宽不一样的）数组的立方下过定义。但是 [1,2,3] .^ 3 在 Julia 里是合法的，它会逐元素地执行 ^ 运算（或称向量化运算），得到 [1^3, 2^3, 3^3]。类似地，! 或 √ 这样的一元运算符，也都有一个对应的 .√ 用于执行逐元素运算。

```
julia> [1,2,3] .^ 3
3-element Vector{Int64}:
  1
  8
 27
```

更确切地说，a .^b 被解析为 “点运算”调用 (^).(a,b)，这会执行 广播 操作：该操作能结合数组和标量、相同大小的数组（进行元素之间的运算），甚至不同形状的数组（例如行、列向量结合生成矩阵）。此外，就像所有向量化的点运算调用一样，这些点运算符是融合的。例如，在计算关于数组 A 的表达式 2 .* A.^2 .+ sin.(A)（或者等价地，使用@. 宏，@. 2A^2 + sin(A)），Julia 只对 A 进行做一次循环，遍历 A 中的每个元素 a 并计算 2a^2 + sin(a)。特别的，类似 f.(g.(x)) 的嵌套点运算调用也是融合的，并且“相邻的”二元运算符表达式 x .+ 3 .* x.^2 可以等价转换为嵌套 dot 调用：(+).(x, (*).(3, (^).(x, 2)))。

除了点运算符，我们还有逐点赋值运算符，类似 a .+= b（或者 @. a += b）会被解析成 a .= a .+ b，这里的 .= 是一个**融合的** in-place 运算，更多信息请查看 dot 文档)。

这个点语法，也能用在用户自定义的运算符上。例如，通过定义 ⊗(A,B) = kron(A,B) 可以为 Kronecker 积（kron）提供一个方便的中缀语法 A ⊗ B，那么配合点语法 [A,B] .⊗ [C,D] 就等价于 [A⊗C, B⊗D]。

将点运算符用于数值字面量可能会导致歧义。例如，1.+x 到底是表示 1. + x 还是 1 .+ x？这会令人疑惑。因此不允许使用这种语法，遇到这种情况时，必须明确地用空格消除歧义。

5.6 数值比较

标准的比较操作对所有原始数值类型有定义：

下面是一些简单的例子：

操作符	名称
==	相等
!=, ≠	不等
<	小于
<=, ≤	小于等于
>	大于
>=, ≥	大于等于

```
julia> 1 == 1
true

julia> 1 == 2
false

julia> 1 != 2
true

julia> 1 == 1.0
true

julia> 1 < 2
true

julia> 1.0 > 3
false

julia> 1 >= 1.0
true

julia> -1 <= 1
true

julia> -1 <= -1
true

julia> -1 <= -2
false

julia> 3 < -0.5
false
```

整数的比较方式是标准的按位比较，而浮点数的比较方式则遵循 IEEE 754 标准。

- 有限数的大小顺序，和我们所熟知的相同。
- `+0` 等于但不大于 `-0`.
- `Inf` 等于自身，并且大于除了 `NaN` 外的所有数。
- `-Inf` 等于自身，并且小于除了 `NaN` 外的所有数。
- `NaN` 不等于、不小于且不大于任何数值，包括它自己。

NaN 不等于它自己这一点可能会令人感到惊奇，所以需要注意：

```
julia> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

当你将 NaN 和 数组 一起连用时，你就会感到头疼：

```
julia> [1 NaN] == [1 NaN]
false
```

为此，Julia 给这些特别的数提供了下面几个额外的测试函数。这些函数在某些情况下很有用处，比如在做 hash 比较时。

函数	测试是否满足如下性质
isequal(x, y)	x 与 y 是完全相同的
isfinite(x)	x 是有限大的数字
isinf(x)	x 是（正/负）无穷大
isnan(x)	x 是 NaN

isequal 认为 NaN 之间是相等的：

```
julia> isequal(NaN, NaN)
true

julia> isequal([1 NaN], [1 NaN])
true

julia> isequal(NaN, NaN32)
true
```

isequal 也能用来区分带符号的零：

```
julia> -0.0 == 0.0
true

julia> isequal(-0.0, 0.0)
false
```

有符号整数、无符号整数以及浮点数之间的混合类型比较是很棘手的。开发者费了很大精力来确保 Julia 在这个问题上做的是正确的。

对于其它类型，`isequal` 会默认调用 `==`，所以如果你想给自己的类型定义相等，那么就只需要为 `==` 增加一个方法。如果你想定义一个你自己的相等函数，你可能需要定义一个对应的 `hash` 方法，用于确保 `isequal(x,y)` 隐含着 `hash(x) == hash(y)`。

链式比较

与其他多数语言不同，就像 notable exception of Python 一样，Julia 允许链式比较：

```
julia> 1 < 2 <= 2 < 3 == 3 > 2 >= 1 == 1 < 3 != 5
true
```

链式比较在写数值代码时特别方便，它使用 && 运算符比较标量，数组则用 & 进行按元素比较。比如，`0 .< A .< 1` 会得到一个 boolean 数组，如果 A 的元素都在 0 和 1 之间则数组元素就都是 true。

注意链式比较的执行顺序：

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> v(1) < v(2) <= v(3)
2
1
3
true

julia> v(1) > v(2) <= v(3)
2
1
false
```

中间的表达式只会计算一次，而如果写成 `v(1) < v(2) && v(2) <= v(3)` 是计算了两次的。然而，链式比较中的顺序是不确定的。强烈建议不要在表达式中使用有副作用（比如 printing）的函数。如果的确需要，请使用短路运算符 &&（请参考短路求值）。

初等函数

Julia 提供了强大的数学函数和运算符集合。这些数学运算定义在各种合理的数值上，包括整型、浮点数、分数和复数，只要这些定义有数学意义就行。

而且，和其它 Julia 函数一样，这些函数也能通过 点语法 `f.(A)` 以 “向量化” 的方式作用于数组和其它集合上。比如，`sin.(A)` 会计算 A 中每个元素的 sin 值。

5.7 运算符的优先级与结合性

从高到低，Julia 运算符的优先级与结合性为：

[1]一元运算符 + 和 - 需要显式调用，即给它们的参数加上括号，以免和 ++ 等运算符混淆。其它一元运算符的混合使用都被解析为右结合的，比如 `√√-a` 解析为 `√(√(-a))`。

[2]运算符 +、++ 和 * 是非结合性的。`a + b + c` 被解析为 `+(a, b, c)` 而不是 `+(+(a, b), c)`。然而，`+(a, b, c, d...)` 和 `*(a, b, c, d...)` 的回退方法默认为左结合求值。

分类	运算符	结合性
语法	. followed by ::	左结合
幂运算	^	右结合
一元运算符	+ - √	右结合[1]
移位运算	<< >> >>>	左结合
除法	//	左结合
乘法	* / % & \ ÷	左结合[2]
加法	+ - | ⊻	左结合[2]
语法	: ..	左结合
语法	|>	左结合
语法	<|	右结合
比较	> < >= <= == === != !== <:	无结合性
流程控制	&& followed by || followed by ?	右结合
Pair 操作	=>	右结合
赋值	= += -= *= /= //= \= ^= ÷= %= |= &= ⊻= <<= >>= >>>=	右结合

想查看 每个 Julia 运算符的优先级，可以参考这个文件：src/julia-parser.scm。注意到有一些运算符在 Base 模块中没有定义但是可能是在标准库、包或者用户代码中定义的。

你也可以通过内置函数 Base.operator_precedence 查看任何给定运算符的优先级数值，数值越大优先级越高：

```
julia> Base.operator_precedence(:+), Base.operator_precedence(:*), Base.operator_precedence(:.)
(11, 12, 17)

julia> Base.operator_precedence(:sin), Base.operator_precedence(:+=),
↪  Base.operator_precedence(:(=))  # (注意：等号前后必须有括号 `:(=)`)
(0, 1, 1)
```

另外，内置函数 Base.operator_associativity 可以返回运算符结合性的符号表示：

```
julia> Base.operator_associativity(:-), Base.operator_associativity(:+),
↪  Base.operator_associativity(:^)
(:left, :none, :right)

julia> Base.operator_associativity(:⊗), Base.operator_associativity(:sin),
↪  Base.operator_associativity(:→)
(:left, :none, :right)
```

注意诸如 :sin 这样的符号返回优先级 0，此值代表无效的运算符或非最低优先级运算符。类似地，它们的结合性被认为是 :none。

数字字面量系数，例如 2x 被视为比任何其他二元运算具有更高优先级的乘法，除了 ^，指数计算具有更高的优先级。

```
julia> x = 3; 2x^2
18

julia> x = 3; 2^2x
64
```

并列解析就像一元运算符，它在指数周围具有相同的自然不对称性：-x^y 和 2x^y 解析为 -(x^y) 和 2(x^y) 而 x^-y 和 x^2y 解析为 x^(-y) 和 x^(2y)。

5.8 数值转换

Julia 支持三种数值转换，它们在处理不精确转换上有所不同。

- T(x) 和 convert(T,x) 都会把 x 转换为 T 类型。
  - 如果 T 是浮点类型，转换的结果就是最近的可表示值，可能会是正负无穷大。
  - 如果 T 为整数类型，当 x 不能由 T 类型表示时，会抛出 InexactError。
- x % T 将整数 x 转换为整型 T，与 x 模 2^n 的结果一致，其中 n 是 T 的位数。换句话说，在二进制表示下被截掉了一部分。
- 舍入函数 接收一个 T 类型的可选参数。比如，round(Int,x) 是 Int(round(x)) 的简写版。

下面的例子展示了不同的形式

```
julia> Int8(127)
127

julia> Int8(128)
ERROR: InexactError: trunc(Int8, 128)
Stacktrace:
[...]

julia> Int8(127.0)
127

julia> Int8(3.14)
ERROR: InexactError: Int8(3.14)
Stacktrace:
[...]

julia> Int8(128.0)
ERROR: InexactError: Int8(128.0)
Stacktrace:
[...]

julia> 127 % Int8
127

julia> 128 % Int8
-128

julia> round(Int8,127.4)
127

julia> round(Int8,127.6)
ERROR: InexactError: trunc(Int8, 128.0)
Stacktrace:
[...]
```

请参考类型转换与类型提升一节来定义你自己的类型转换和提升规则。

舍入函数

函数	描述	返回类型
round(x)	x 舍到最接近的整数	typeof(x)
round(T, x)	x 舍到最接近的整数	T
floor(x)	x 向 -Inf 舍入	typeof(x)
floor(T, x)	x 向 -Inf 舍入	T
ceil(x)	x 向 +Inf 方向取整	typeof(x)
ceil(T, x)	x 向 +Inf 方向取整	T
trunc(x)	x 向 0 取整	typeof(x)
trunc(T, x)	x 向 0 取整	T

除法函数

函数	描述
div(x,y), x÷y	截断除法；商向零近似
fld(x,y)	向下取整除法；商向 -Inf 近似
cld(x,y)	向上取整除法；商向 +Inf 近似
rem(x,y), x%y	取余；满足 x == div(x,y)*y + rem(x,y)；符号与 x 一致
mod(x,y)	取模；满足 x == fld(x,y)*y + mod(x,y)；符号与 y 一致
mod1(x,y)	偏移 1 的 mod；若 y>0，则返回 r∈(0,y]，若 y<0，则 r∈[y,0) 且满足 mod(r, y) == mod(x, y)
mod2pi(x)	对 2pi 取模；0 <= mod2pi(x) < 2pi
divrem(x,y)	返回 (div(x,y),rem(x,y))
fldmod(x,y)	返回 (fld(x,y),mod(x,y))
gcd(x,y...)	x, y,... 的最大公约数
lcm(x,y...)	x, y,... 的最小公倍数

符号和绝对值函数

函数	描述
abs(x)	x 的模
abs2(x)	x 的模的平方
sign(x)	表示 x 的符号，返回 -1，0，或 +1
signbit(x)	表示符号位是 true 或 false
copysign(x,y)	返回一个数，其值等于 x 的模，符号与 y 一致
flipsign(x,y)	返回一个数，其值等于 x 的模，符号与 x*y 一致

幂、对数与平方根

想大概了解一下为什么诸如 hypot、expm1和 log1p 函数是必要和有用的，可以看一下 John D. Cook 关于这些主题的两篇优秀博文：expm1, log1p, erfc，和 hypot。

三角和双曲函数

所有标准的三角和双曲函数也都已经定义了：

函数	描述
`sqrt(x)`, $\sqrt{x}$	x 的平方根
`cbrt(x)`, $\sqrt[3]{x}$	x 的立方根
`hypot(x,y)`	当直角边的长度为 x 和 y 时，直角三角形斜边的长度
`exp(x)`	自然指数函数在 x 处的值
`expm1(x)`	当 x 接近 0 时的 `exp(x)-1` 的精确值
`ldexp(x,n)`	`x*2^n` 的高效算法，`n` 为整数
`log(x)`	x 的自然对数
`log(b,x)`	以 b 为底 x 的对数
`log2(x)`	以 2 为底 x 的对数
`log10(x)`	以 10 为底 x 的对数
`log1p(x)`	当 x 接近 0 时的 `log(1+x)` 的精确值
`exponent(x)`	x 的二进制指数
`significand(x)`	浮点数 x 的二进制有效数（也就是尾数）

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
asinh  acosh  atanh  acoth  asech  acsch
sinc   cosc
```

所有这些函数都是单参数函数，不过 `atan` 也可以接收两个参数来表示传统的 `atan2` 函数。

另外，`sinpi(x)` 和 `cospi(x)` 分别用来对 `sin(pi*x)` 和 `cos(pi*x)` 进行更精确的计算。

要计算角度而非弧度的三角函数，以 d 做后缀。比如，`sind(x)` 计算 x 的 sine 值，其中 x 是一个角度值。下面是角度变量的三角函数完整列表：

```
sind   cosd   tand   cotd   secd   cscd
asind  acosd  atand  acotd  asecd  acscd
```

特殊函数

SpecialFunctions.jl 提供了许多其他的特殊数学函数。

Chapter 6

复数和有理数

Julia 语言包含了预定义的复数和有理数类型，并且支持它们的各种标准数学运算和初等函数。由于也定义了复数与分数的类型转换与类型提升，因此对预定义数值类型（无论是原始的还是复合的）的任意组合进行的操作都会表现得如预期的一样。

6.1 复数

全局常量 `im` 被绑定到复数 *i*，表示 -1 的主平方根。（不应使用数学家习惯的 `i` 或工程师习惯的 `j` 来表示此全局常量，因为它们是非常常用的索引变量名）由于 Julia 允许数值字面量作为数值字面量系数，这种绑定就足以为复数提供很方便的语法，类似于传统的数学记法：

```
julia> 1+2im
1 + 2im
```

你可以对复数进行各种标准算术操作：

```
julia> (1 + 2im)*(2 - 3im)
8 + 1im

julia> (1 + 2im)/(1 - 2im)
-0.6 + 0.8im

julia> (1 + 2im) + (1 - 2im)
2 + 0im

julia> (-3 + 2im) - (5 - 1im)
-8 + 3im

julia> (-1 + 2im)^2
-3 - 4im

julia> (-1 + 2im)^2.5
2.729624464784009 - 6.9606644595719im

julia> (-1 + 2im)^(1 + 1im)
-0.27910381075826657 + 0.08708053414102428im

julia> 3(2 - 5im)
6 - 15im
```

```
julia> 3(2 - 5im)^2
-63 - 60im

julia> 3(2 - 5im)^-1.0
0.20689655172413793 + 0.5172413793103449im
```

类型提升机制也确保你可以使用不同类型的操作数的组合:

```
julia> 2(1 - 1im)
2 - 2im

julia> (2 + 3im) - 1
1 + 3im

julia> (1 + 2im) + 0.5
1.5 + 2.0im

julia> (2 + 3im) - 0.5im
2.0 + 2.5im

julia> 0.75(1 + 2im)
0.75 + 1.5im

julia> (2 + 3im) / 2
1.0 + 1.5im

julia> (1 - 3im) / (2 + 2im)
-0.5 - 1.0im

julia> 2im^2
-2 + 0im

julia> 1 + 3/4im
1.0 - 0.75im
```

注意 `3/4im == 3/(4*im) == -(3/4*im)`，因为系数比除法的优先级更高。

Julia 提供了一些操作复数的标准函数:

```
julia> z = 1 + 2im
1 + 2im

julia> real(1 + 2im) # z 的实部
1

julia> imag(1 + 2im) # z 的虚部
2

julia> conj(1 + 2im) # z 的复共轭
1 - 2im

julia> abs(1 + 2im) # z 的绝对值
2.23606797749979
```

```
julia> abs2(1 + 2im) # 取平方后的绝对值
5

julia> angle(1 + 2im) # 以弧度为单位的相位角
1.1071487177940904
```

按照惯例，复数的绝对值（abs）是从零点到它的距离。abs2 给出绝对值的平方，作用于复数上时非常有用，因为它避免了取平方根。angle 返回以弧度为单位的相位角（也被称为辐角函数）。所有其它的初等函数在复数上也都有完整的定义：

```
julia> sqrt(1im)
0.7071067811865476 + 0.7071067811865475im

julia> sqrt(1 + 2im)
1.272019649514069 + 0.7861513777574233im

julia> cos(1 + 2im)
2.0327230070196656 - 3.0518977991517997im

julia> exp(1 + 2im)
-1.1312043837568135 + 2.4717266720048188im

julia> sinh(1 + 2im)
-0.4890562590412937 + 1.4031192506220405im
```

注意数学函数通常应用于实数就返回实数值，应用于复数就返回复数值。例如，当 sqrt 应用于 -1 与 -1 + 0im 会有不同的表现，虽然 -1 == -1 + 0im：

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex result if called with
↪  a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]

julia> sqrt(-1 + 0im)
0.0 + 1.0im
```

从变量构建复数时，文本型数值系数记法不再适用。相反地，乘法必须显式地写出：

```
julia> a = 1; b = 2; a + b*im
1 + 2im
```

然而，我们**并不**推荐这样做，而应改为使用更高效的 complex 函数直接通过实部与虚部构建一个复数值：

```
julia> a = 1; b = 2; complex(a, b)
1 + 2im
```

这种构建避免了乘法和加法操作。

Inf 和 NaN 可能出现在复数的实部和虚部，正如特殊的浮点值章节所描述的：

```
julia> 1 + Inf*im
1.0 + Inf*im

julia> 1 + NaN*im
1.0 + NaN*im
```

6.2 有理数

Julia 有一个用于表示整数精确比值的分数类型。分数通过 // 运算符构建：

```
julia> 2//3
2//3
```

如果一个分数的分子和分母含有公因子，它们会被约分到最简形式且分母非负：

```
julia> 6//9
2//3

julia> -4//8
-1//2

julia> 5//-15
-1//3

julia> -4//-12
1//3
```

整数比值的这种标准化形式是唯一的，所以分数值的相等性可由校验分子与分母都相等来测试。分数值的标准化分子和分母可以使用 numerator 和 denominator 函数得到：

```
julia> numerator(2//3)
2

julia> denominator(2//3)
3
```

分子和分母的直接比较通常是不必要的，因为标准算术和比较操作对分数值也有定义：

```
julia> 2//3 == 6//9
true

julia> 2//3 == 9//27
false

julia> 3//7 < 1//2
true
```

```
julia> 3//4 > 2//3
true

julia> 2//4 + 1//6
2//3

julia> 5//12 - 1//4
1//6

julia> 5//8 * 3//12
5//32

julia> 6//5 / 10//7
21//25
```

分数可以很容易地转换成浮点数：

```
julia> float(3//4)
0.75
```

对于任何整数值 `a` 和 `b`，从有理数到浮点数的转换都要遵守以下特性，但 `a==0 && b <= 0` 时除外：

```
julia> a = 1; b = 2;

julia> isequal(float(a//b), a/b)
true

julia> a, b = 0, 0
(0, 0)

julia> float(a//b)
ERROR: ArgumentError: invalid rational: zero(Int64)//zero(Int64)
Stacktrace:
[...]

julia> a/b
NaN

julia> a, b = 0, -1
(0, -1)

julia> float(a//b), a/b
(0.0, -0.0)
```

Julia 接受构建无穷分数值：

```
julia> 5//0
1//0

julia> x = -3//0
-1//0
```

```
julia> typeof(x)
Rational{Int64}
```

但不接受试图构建一个 `NaN` 分数值:

```
julia> 0//0
ERROR: ArgumentError: invalid rational: zero(Int64)//zero(Int64)
Stacktrace:
[...]
```

像往常一样，类型提升系统使得分数可以轻松地同其它数值类型进行交互:

```
julia> 3//5 + 1
8//5

julia> 3//5 - 0.5
0.09999999999999998

julia> 2//7 * (1 + 2im)
2//7 + 4//7*im

julia> 2//7 * (1.5 + 2im)
0.42857142857142855 + 0.5714285714285714im

julia> 3//2 / (1 + 2im)
3//10 - 3//5*im

julia> 1//2 + 2im
1//2 + 2//1*im

julia> 1 + 2//3im
1//1 - 2//3*im

julia> 0.5 == 1//2
true

julia> 0.33 == 1//3
false

julia> 0.33 < 1//3
true

julia> 1//3 - 0.33
0.0033333333333332993
```

Chapter 7

字符串

字符串是由有限个字符组成的序列。而字符在英文中一般包括字母 A,B, C 等、数字和常用的标点符号。这些字符由 ASCII 标准统一标准化并且与 0 到 127 范围内的整数一一对应。当然，还有很多非英文字符，包括 ASCII 字符在注音或其他方面的变体，例如西里尔字母和希腊字母，以及与 ASCII 和英文均完全无关的字母系统，包括阿拉伯语，中文，希伯来语，印度语，日本语，和韩语。Unicode 标准对这些复杂的字符做了统一的定义，是一种大家普遍接受标准。根据需求，写代码时可以忽略这种复杂性而只处理 ASCII 字符，也可针对可能出现的非 ASCII 文本而处理所有的字符或编码。Julia 可以简单高效地处理纯粹的 ASCII 文本以及 Unicode 文本。甚至，在 Julia 中用 C 语言风格的代码来处理 ASCII 字符串，可以在不失性能和易读性的前提下达到预期效果。当遇到非 ASCII 文本时，Julia 会优雅明确地提示错误信息而不是引入乱码。这时，直接修改代码使其可以处理非 ASCII 数据即可。

关于 Julia 的字符串类型有一些值得注意的高级特性：

- Julia 中用于字符串（和字符串字面量）的内置具体类型是 `String`。它支持全部 Unicode 字符通过 UTF-8 编码。（`transcode` 函数是提供 Unicode 编码和其他编码转换的函数。）
- 所有的字符串类型都是抽象类型 `AbstractString` 的子类型，而一些外部包定义了别的 `AbstractString` 子类型（例如为其它的编码定义的子类型）。若要定义需要字符串参数的函数，你应当声明此类型为 `AbstractString` 来让这函数接受任何字符串类型。
- 类似 C 和 Java，但是和大多数动态语言不同的是，Julia 有优秀的表示单字符的类型，即 `AbstractChar`。`Char` 是 `AbstractChar` 的内置子类型，它能表示任何 Unicode 字符的 32 位原始类型（基于 UTF-8 编码）。
- 如 Java 中那样，字符串不可改——任何 `AbstractString` 对象的值不可改变。若要构造不同的字符串值，应当从其它字符串的部分构造一个新的字符串。
- 从概念上讲，字符串是从索引到字符的部分函数：对于某些索引值，它不返回字符值，而是引发异常。这允许通过编码表示形式的字节索引来实现高效的字符串索引，而不是通过字符索引——它不能简单高效地实现可变宽度的 Unicode 字符串编码。

7.1 字符

`Char` 类型的值代表单个字符：它只是带有特殊文本表示法和适当算术行为的 32 位原始类型，不能转化为代表 Unicode 代码 的数值。(Julia 的包可能会定义别的 `AbstractChar` 子类型，比如当为了优化对其它 字符编码 的操作时）`Char` 类型的值以这样的方式输入和显示。

(请注意，字符字面量是用单引号分隔的，而不是双引号)：

```
julia> c = 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> typeof(c)
Char
```

你可以轻松地将 Char 转换为其对应的整数值，即 Unicode 代码：

```
julia> c = Int('x')
120

julia> typeof(c)
Int64
```

在 32 位的计算机中，typeof(c) 将显示为 Int32。你可以轻松地将一个整数值转回 Char：

```
julia> Char(120)
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

并非所有的整数值都是有效的 Unicode 代码，但是为了性能，Char 的转化不会检查每个值是否有效。如果你想检查每个转换的值是否为有效值，请使用 isvalid 函数：

```
julia> Char(0x110000)
'\U110000': Unicode U+110000 (category In: Invalid, too high)

julia> isvalid(Char, 0x110000)
false
```

目前，有效的 Unicode 码点为，从 U+0000 至 U+D7FF，以及从 U+E000 至 U+10FFFF。它们还未全部被赋予明确的含义，也还没必要能被程序识别；然而，所有的这些值都被认为是有效的 Unicode 字符。

你可以在单引号中输入任何 Unicode 字符，通过使用 \u 加上至多 4 个十六进制数字或者 \U 加上至多 8 个十六进制数（最长的有效值也只需要 6 个）：

```
julia> '\u0'
'\0': ASCII/Unicode U+0000 (category Cc: Other, control)

julia> '\u78'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> '\u2200'
'∀': Unicode U+2200 (category Sm: Symbol, math)

julia> '\U10ffff'
'\U10ffff': Unicode U+10FFFF (category Cn: Other, not assigned)
```

Julia 使用系统默认的区域和语言设置来确定，哪些字符可以被正确显示，哪些需要用 \u 或 \U 的转义来显示。除 Unicode 转义格式之外，还可以使用所有的传统 C 语言转义输入形式：

```
julia> Int('\0')
0

julia> Int('\t')
9

julia> Int('\n')
10

julia> Int('\e')
27

julia> Int('\x7f')
127

julia> Int('\177')
127
```

你可以对 `Char` 的值进行比较和有限的算术运算：

```
julia> 'A' < 'a'
true

julia> 'A' <= 'a' <= 'Z'
false

julia> 'A' <= 'X' <= 'Z'
true

julia> 'x' - 'a'
23

julia> 'A' + 1
'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
```

7.2 字符串基础

字符串字面量由双引号或三重双引号（非单引号）分隔：

```
julia> str = "Hello, world.\n"
"Hello, world.\n"

julia> """Contains "quote" characters"""
"Contains \"quote\" characters"
```

字符串中的长行可以通过在换行符前加反斜杠 (\) 来分隔：

```
julia> "This is a long \
       line"
"This is a long line"
```

如果要字符串中提取字符，可以对其进行索引：

```
julia> str[begin]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> str[1]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> str[6]
',': ASCII/Unicode U+002C (category Po: Punctuation, other)

julia> str[end]
'\n': ASCII/Unicode U+000A (category Cc: Other, control)
```

许多的 Julia 对象，包括字符串，都可以用整数进行索引。第一个元素的索引（字符串的第一个字符）由 `firstindex(str)` 返回，最后一个元素（字符）的索引由 `lastindex(str)` 返回。关键字 `begin` 和 `end` 可以在索引操作中使用，它们分别表示给定维度上的第一个和最后一个索引。字符串索引就像 Julia 中的大多数索引一样，是从 1 开始的：对于任何 `AbstractString`，`firstindex` 方法总是返回 `1`。但是，下面我们将会看到，对于一个字符串来说 `lastindex(str)` 和 `length(str)` 的结果不一定相同，因为 Unicode 字符可能由多个编码单元组成。

你可以用 `end` 进行算术以及其它操作，就像一个普通值一样：

```
julia> str[end-1]
'.': ASCII/Unicode U+002E (category Po: Punctuation, other)

julia> str[end÷2]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

使用小于 `begin` (1) 或大于 `end` 的索引会引发错误：

```
julia> str[begin-1]
ERROR: BoundsError: attempt to access 14-codeunit String at index [0]
[...]

julia> str[end+1]
ERROR: BoundsError: attempt to access 14-codeunit String at index [15]
[...]
```

你也可以用范围索引来提取子字符串：

```
julia> str[4:9]
"lo, wo"
```

注意，表达式 `str[k]` 和 `str[k:k]` 不会给出相同的结果：

```
julia> str[6]
',': ASCII/Unicode U+002C (category Po: Punctuation, other)

julia> str[6:6]
","
```

前者是一个 Char 类型的单个字符，而后者是一个恰好只包含一个字符的字符串。在 Julia 中，这些是不同的。

范围索引复制原始字符串的选定部分。此外，可以使用类型 SubString，将视图创建为字符串，更简单地说，在代码块上使用 @views 宏会将所有字符串切片转换为子字符串。

例如：

```
julia> str = "long string"
"long string"

julia> substr = SubString(str, 1, 4)
"long"

julia> typeof(substr)
SubString{String}

julia> @views typeof(str[1:4]) # @views converts slices to SubStrings
SubString{String}
```

几个标准函数，像 chop, chomp 或者 strip 都会返回一个 SubString。

7.3 Unicode 和 UTF-8

Julia 完全支持 Unicode 字符和字符串。如上所述，在字符字面量中，Unicode 代码可以用 Unicode \u 和 \U 转义序列表示，也可以用所有标准 C 转义序列表示。这些同样可以用来写字符串字面量：

```
julia> s = "\u2200 x \u2203 y"
"∀ x ∃ y"
```

这些 Unicode 字符是作为转义还是特殊字符显示，取决于你终端的语言环境设置以及它对 Unicode 的支持。字符串字面量用 UTF-8 编码。UTF-8 是一种可变长度的编码，也就是说并非所有字符都以相同的字节数（code units）编码。在 UTF-8 中，ASCII 字符（小于 0x80(128) 的那些）如它们在 ASCII 中一样使用单字节编码；而 0x80 及以上的字符使用最多 4 个字节编码。

在 Julia 中字符串索引指的是代码单元（对于 UTF-8 来说等同于字节/byte），固定宽度的构建块用于编码任意字符（code point）。这意味着并非每个索引到 UTF-8 字符串的字节都必须是一个字符的有效索引。如果在这种无效字节索引处索引字符串，将会报错：

```
julia> s[1]
'∀': Unicode U+2200 (category Sm: Symbol, math)

julia> s[2]
ERROR: StringIndexError: invalid index [2], valid nearby indices [1]=>'∀', [4]=>' '
Stacktrace:
[...]

julia> s[3]
ERROR: StringIndexError: invalid index [3], valid nearby indices [1]=>'∀', [4]=>' '
Stacktrace:
[...]

julia> s[4]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

在这种情况下，字符 ∀ 是一个三字节字符，因此索引 2 和 3 都是无效的，而下一个字符的索引是 4；这个接下来的有效索引可以用 nextind(s,1) 来计算，再接下来的用 nextind(s,4)，依此类推。

如果倒数第二个字符是多字节字符，由于 end 总是集合中最后一个有效索引，这时 end-1 将会是无效索引。

```
julia> s[end-1]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)

julia> s[end-2]
ERROR: StringIndexError: invalid index [9], valid nearby indices [7]=>'∃', [10]=>' '
Stacktrace:
[...]

julia> s[prevind(s, end, 2)]
'∃': Unicode U+2203 (category Sm: Symbol, math)
```

第一种情况可以，因为最后一个字符 y 和空格都是一字节的字符，而 end-2 索引到中间的 ∃ 由多字节表示。正确的方法是使用 prevind(s, lastindex(s), 2)，或者，如果你使用该值来索引 s，则可以写为 s[prevind(s, end, 2)]，end 展开为 lastindex(s)。

使用范围索引提取子字符串也需要有效的字节索引，不然就会抛出错误：

```
julia> s[1:1]
"∀"

julia> s[1:2]
ERROR: StringIndexError: invalid index [2], valid nearby indices [1]=>'∀', [4]=>' '
Stacktrace:
[...]

julia> s[1:4]
"∀ "
```

由于可变长度的编码，字符串中的字符数（由 length(s) 给出）并不总是等于最后一个索引的数字。如果你从 1 到 lastindex(s) 迭代并索引到 s，未报错时返回的字符序列是包含字符串 s 的字符序列。所以，length(s) <= lastindex(s)，这是因为字符串中的每个字符必须有它自己的索引。下面是对 s 的字符进行迭代的一个冗长而低效的方式：

```
julia> for i = firstindex(s):lastindex(s)
           try
               println(s[i])
           catch
               # ignore the index error
           end
       end
∀

x

∃

y
```

空行上面其实是有空格的。幸运的是，上面的笨拙写法不是对字符串中字符进行迭代所必须的——因为你只需把字符串本身用作迭代对象，而不需要额外处理：

```
julia> for c in s
           println(c)
       end
∀

x

∃

y
```

如果需要为字符串获取有效索引，可以使用 nextind 和 prevind 函数递增/递减到下一个/前一个有效索引，如前所述。你也可以使用 eachindex 函数迭代有效的字符索引：

```
julia> collect(eachindex(s))
7-element Vector{Int64}:
  1
  4
  5
  6
  7
 10
 11
```

要访问编码的原始代码单位（UTF-8 的字节），可以使用 codeunit(s,i)函数，其中索引 i 从 1 连续运行到 ncodeunits(s)。codeunits(s) 函数返回一个 AbstractVector{UInt8} 包装器，允许您以数组的形式访问这些原始代码单元（字节）。

Julia 中的字符串可以包含无效的 UTF-8 代码单元序列。这个惯例允许把任何字序列当作 String。在这种情形下的一个规则是，当从左到右解析代码单元序列时，字符由匹配下面开头位模式之一的最长的 8 位代码单元序列组成（每个 x 可以是 0 或者 1）：

- 0xxxxxxx;
- 110xxxxx 10xxxxxx;
- 1110xxxx 10xxxxxx 10xxxxxx;
- 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx;
- 10xxxxxx;
- 11111xxx.

特别地，这意味着过长和过高的代码单元序列及其前缀将被视为单个无效字符，而不是多个无效字符。这个规则最好用一个例子来解释：

```
julia> s = "\xc0\xa0\xe2\x88\xe2|"
"\xc0\xa0\xe2\x88\xe2|"

julia> foreach(display, s)
'\xc0\xa0': [overlong] ASCII/Unicode U+0020 (category Zs: Separator, space)
'\xe2\x88': Malformed UTF-8 (category Ma: Malformed, bad data)
'\xe2': Malformed UTF-8 (category Ma: Malformed, bad data)
'|': ASCII/Unicode U+007C (category Sm: Symbol, math)

julia> isvalid.(collect(s))
4-element BitArray{1}:
 0
 0
 0
 1

julia> s2 = "\xf7\xbf\xbf\xbf"
"\U1fffff"

julia> foreach(display, s2)
'\U1fffff': Unicode U+1FFFFF (category In: Invalid, too high)
```

我们可以看到字符串 s 中的前两个代码单元形成了一个过长的空格字符编码。这是无效的，但是在字符串中作为单个字符是可以接受的。接下来的两个代码单元形成了一个有效的 3 位 UTF-8 序列开头。然而，第五个代码单元 \xe2 不是它的有效延续，所以代码单元 3 和 4 在这个字符串中也被解释为格式错误的字符。同理，由于 | 不是它的有效延续，代码单元 5 形成了一个格式错误的字符。最后字符串 s2 包含了一个太高的代码。

Julia 默认使用 UTF-8 编码，对于新编码的支持可以通过包加上。例如，LegacyStrings.jl 包实现了 UTF16String 和 UTF32String 类型。关于其它编码的额外讨论以及如何实现对它们的支持暂时超过了这篇文档的讨论范围。UTF-8 编码相关问题的进一步讨论参见下面的字节数组字面量章节。transcode 函数可在各种 UTF-xx 编码之间转换，主要用于外部数据和包。

7.4 拼接

最常见最有用的字符串操作是级联：

```
julia> greet = "Hello"
"Hello"

julia> whom = "world"
"world"

julia> string(greet, ", ", whom, ".\n")
"Hello, world.\n"
```

意识到像对无效 UTF-8 字符进行级联这样的潜在危险情形是非常重要的。生成的字符串可能会包含和输入字符串不同的字符，并且其中字符的数目也可能少于被级联字符串中字符数目之和，例如：

```
julia> a, b = "\xe2\x88", "\x80"
("\xe2\x88", "\x80")
```

```
julia> c = string(a, b)
"∀"

julia> collect.([a, b, c])
3-element Vector{Vector{Char}}:
 ['\xe2\x88']
 ['\x80']
 ['∀']

julia> length.([a, b, c])
3-element Vector{Int64}:
 1
 1
 1
```

这种情形只可能发生于无效 UTF-8 字符串上。对于有效 UTF-8 字符串，级联保留字符串中的所有字符和字符串的总长度。

Julia 也提供 * 用于字符串级联：

```
julia> greet * ", " * whom * ".\n"
"Hello, world.\n"
```

尽管对于提供 + 函数用于字符串拼接的语言使用者而言，* 似乎是一个令人惊讶的选择，但 * 的这种用法在数学中早有先例，尤其是在抽象代数中。

在数学上，+ 通常表示可交换运算（*commutative* operation）——运算对象的顺序不重要。一个例子是矩阵加法：对于任何形状相同的矩阵 A 和 B，都有 A + B == B + A。与之相反，* 通常表示不可交换运算——运算对象的顺序很重要。例如，对于矩阵乘法，一般 A * B != B * A。同矩阵乘法类似，字符串拼接是不可交换的：greet * whom != whom * greet。在这一点上，对于插入字符串的拼接操作，* 是一个自然而然的选择，与它在数学中的用法一致。

更确切地说，有限长度字符串集合 S 和字符串拼接操作 * 构成了一个自由幺半群 $(S, *)$。该集合的单位元是空字符串，""。当一个自由幺半群不是交换的时，它的运算通常表示为 \cdot，*，或者类似的符号，而非暗示交换性的 +。

7.5 插值

拼接构造字符串的方式有时有些麻烦。为了减少对于 string 的冗余调用或者重复地做乘法，Julia 允许像 Perl 中一样使用 $ 对字符串字面量进行插值：

```
julia> greet = "Hello"; whom = "world";

julia> "$greet, $whom.\n"
"Hello, world.\n"
```

这更易读更方便，而且等效于上面的字符串拼接——系统把这个显然一行的字符串字面量重写成带参数的字符串字面量拼接 string(greet, ", ", whom, ".\n")。

在 $ 之后最短的完整表达式被视为插入其值于字符串中的表达式。因此，你可以用括号向字符串中插入任何表达式：

```
julia> "1 + 2 = $(1 + 2)"
"1 + 2 = 3"
```

拼接和插值都调用 string 以转换对象为字符串形式。然而，string 实际上仅仅返回了 print 的输出，因此，新的类型应该添加 print 或 show 方法，而不是 string 方法。

多数非 AbstractString 对象被转换为和它们作为文本表达式输入的方式密切对应的字符串：

```
julia> v = [1,2,3]
3-element Vector{Int64}:
 1
 2
 3

julia> "v: $v"
"v: [1, 2, 3]"
```

string 是 AbstractString 和 AbstractChar 值的标识，所以它们作为自身被插入字符串，无需引用，无需转义：

```
julia> c = 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> "hi, $c"
"hi, x"
```

若要在字符串字面量中包含文本 $，就用反斜杠转义：

```
julia> print("I have \$100 in my account.\n")
I have $100 in my account.
```

7.6 三引号字符串字面量

当使用三引号（"""..."""）创建字符串时，它们有一些在创建更长文本块时可能用到的特殊行为。

首先，三引号字符串也被反缩进到最小缩进线的水平。这在定义包含缩进的字符串时很有用。例如：

```
julia> str = """
           Hello,
           world.
         """
"  Hello,\n  world.\n"
```

在这里，后三引号 """ 前面的最后一（空）行设置了缩进级别。

反缩进级别被确定为所有行中空格或制表符的最大公共起始序列，不包括前三引号 """ 后面的一行以及只包含空格或制表符的行（总包含结尾 """ 的行）。那么对于所有不包括前三引号 """ 后面文本的行而言，公共起始序列就被移除了（包括只含空格和制表符而以此序列开始的行），例如：

```
julia> """    This
         is
           a test"""
"    This\nis\n  a test"
```

接下来，如果前三引号 """ 后面紧跟换行符，那么换行符就从生成的字符串中被剥离。

```
"""hello"""
```

等价于

```
"""
hello"""
```

但是

```
"""

hello"""
```

将在开头包含一个文本换行符。

换行符的移除是在反缩进之后进行的。例如：

```
julia> """
         Hello,
         world."""
"Hello,\nworld."
```

如果使用反斜杠消除换行符，下一行的缩进也会被消除：

```
julia> """
         Averylong\
         word"""
"Averylongword"
```

Trailing whitespace is left unaltered.

```
julia> """
         Averylong\
         word"""
"Averylongword"
```

尾随空格保持不变。

三引号字符串字面量可不带转义地包含 " 符号。

注意，无论是用单引号还是三引号，在文本字符串中换行符都会生成一个换行 (LF) 字符 `\n`，即使你的编辑器使用回车组合符 `\r` (CR) 或 CRLF 来结束行。为了在字符串中包含 CR，总是应该使用显式转义符 `\r`；比如，可以输入文本字符串 `"a CRLF line ending\r\n"`。

7.7 常见操作

你可以使用标准的比较操作符按照字典顺序比较字符串：

```
julia> "abracadabra" < "xylophone"
true

julia> "abracadabra" == "xylophone"
false

julia> "Hello, world." != "Goodbye, world."
true

julia> "1 + 2 = 3" == "1 + 2 = $(1 + 2)"
true
```

你可以使用 `findfirst` 与 `findlast` 函数搜索特定字符的索引：

```
julia> findfirst('o', "xylophone")
4

julia> findlast('o', "xylophone")
7

julia> findfirst('z', "xylophone")
```

你可以带上第三个参数，用 `findnext` 与 `findprev` 函数来在给定偏移量处搜索字符：

```
julia> findnext('o', "xylophone", 1)
4

julia> findnext('o', "xylophone", 5)
7

julia> findprev('o', "xylophone", 5)
4

julia> findnext('o', "xylophone", 8)
```

你可以用 `occursin` 函数检查在字符串中某子字符串可否找到。

```
julia> occursin("world", "Hello, world.")
true

julia> occursin("o", "Xylophon")
true

julia> occursin("a", "Xylophon")
false

julia> occursin('o', "Xylophon")
true
```

最后那个例子表明 occursin 也可用于搜寻字符字面量。

另外还有两个方便的字符串函数 repeat 和 join：

```
julia> repeat(".:Z:.", 10)
".:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:..:Z:."

julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
"apples, bananas and pineapples"
```

其它有用的函数还包括：

- firstindex(str) 给出可用来索引到 str 的最小（字节）索引（对字符串来说这总是 1，对于别的容器来说却不一定如此）。
- lastindex(str) 给出可用来索引到 str 的最大（字节）索引。
- length(str)，str 中的字符个数。
- length(str, i, j)，str 中从 i 到 j 的有效字符索引个数。
- ncodeunits(str)，字符串中代码单元（码元）的数目。
- codeunit(str, i) 给出在字符串 str 中索引为 i 的代码单元值。
- thisind(str, i)，给定一个字符串的任意索引，查找索引点所在的首个索引。
- nextind(str, i, n=1) 查找在索引 i 之后第 n 个字符的开头。
- prevind(str, i, n=1) 查找在索引 i 之前第 n 个字符的开始。

有些情况下，你想构造一个字符串或使用字符串语义，但标准字符串结构的行为并不完全符合需求。对于这类情况，Julia 提供了非标准字符串字面量。非标准字符串字面量看起来像常规的双引号字符串字面量，但前面会紧接着一个标识符，其行为可能与普通字符串字面量不同。

如下所述：正则表达式、字节数组字面量和版本号字面量，都是非标准字符串字面量的一些例子。用户和包也可以定义新的非标准字符串字面量。更多文档在元编程部分提供。

7.8 正则表达式

有时你要寻找的不是确切的字符串，而是特定的模式。例如，假设你正试图从一个大型文本文件中提取单个日期。你不知道那个日期是什么（这就是为什么你要搜索它），但你知道它看起来像 YYYY-MM-DD。正则表达式允许你指定这些模式并搜索它们。

Julia 使用由 PCRE 库提供的 Perl 兼容正则表达式（regexes）的版本 2（更多详情请参见 PCRE2 语法描述）。

正则表达式与字符串有两种关联方式：明显的联系是正则表达式用于在字符串中查找常规模式；另一个联系是正则表达式本身作为字符串输入，被解析成一个状态机，可用于高效地在字符串中搜索模式。

在 Julia 中，正则表达式使用以 r 开头的各种标识符作为前缀的非标准字符串字面量输入。最基本的正则表达式字面量，没有打开任何选项，只使用 r"..."：

```
julia> re = r"^\s*(?:#|$)"
r"^\s*(?:#|$)"

julia> typeof(re)
Regex
```

若要检查正则表达式是否匹配某字符串，就用 `occursin`：

```
julia> occursin(r"^\s*(?:#|$)", "not a comment")
false

julia> occursin(r"^\s*(?:#|$)", "# a comment")
true
```

可以看到，`occursin` 只返回正确或错误，表明给定正则表达式是否在该字符串中出现。然而，通常我们不只想知道字符串是否匹配，更想了解它是如何匹配的。要捕获匹配的信息，可以改用 `match` 函数：

```
julia> match(r"^\s*(?:#|$)", "not a comment")

julia> match(r"^\s*(?:#|$)", "# a comment")
RegexMatch("#")
```

若正则表达式与给定字符串不匹配，`match` 返回 `nothing`——在交互式提示框中不打印任何东西的特殊值。除了不打印，它是一个完全正常的值，这可以用程序来测试：

```
m = match(r"^\s*(?:#|$)", line)
if m === nothing
    println("not a comment")
else
    println("blank or comment")
end
```

如果正则表达式匹配，`match` 的返回值是一个 `RegexMatch` 对象。这些对象记录了表达式是如何匹配的，包括该模式匹配的子字符串和任何可能被捕获的子字符串。上面的例子仅仅捕获了匹配的部分子字符串，但也许我们想要捕获的是注释字符后面的任何非空文本。我们可以这样做：

```
julia> m = match(r"^\s*(?:#\s*(.*?)\s*$|$)", "# a comment ")
RegexMatch("# a comment ", 1="a comment")
```

当调用 `match` 时，你可以选择指定开始搜索的索引。例如：

```
julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",1)
RegexMatch("1")

julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",6)
RegexMatch("2")

julia> m = match(r"[0-9]","aaaa1aaaa2aaaa3",11)
RegexMatch("3")
```

你可以从 RegexMatch 对象中提取如下信息：

- 匹配的整个子字符串：m.match
- 作为字符串数组捕获的子字符串：m.captures
- 整个匹配开始处的偏移：m.offset
- 作为向量的捕获子字符串的偏移：m.offsets

当捕获不匹配时，m.captures 在该处不再包含一个子字符串，而是 什么也不包含；此外，m.offsets 的偏移量为 0（回想一下，Julia 的索引是从 1 开始的，因此字符串的零偏移是无效的）。下面是两个有些牵强的例子：

```
julia> m = match(r"(a|b)(c)?(d)", "acd")
RegexMatch("acd", 1="a", 2="c", 3="d")

julia> m.match
"acd"

julia> m.captures
3-element Vector{Union{Nothing, SubString{String}}}:
 "a"
 "c"
 "d"

julia> m.offset
1

julia> m.offsets
3-element Vector{Int64}:
 1
 2
 3

julia> m = match(r"(a|b)(c)?(d)", "ad")
RegexMatch("ad", 1="a", 2=nothing, 3="d")

julia> m.match
"ad"

julia> m.captures
3-element Vector{Union{Nothing, SubString{String}}}:
 "a"
 nothing
 "d"

julia> m.offset
1

julia> m.offsets
3-element Vector{Int64}:
 1
 0
 2
```

让捕获作为数组返回是很方便的，这样就可以用解构语法把它们和局域变量绑定起来。为了方便，RegexMatch 对象实现了传递到 captures 字段的迭代器方法，因此您可以直接解构匹配对象：

```
julia> first, second, third = m; first
"a"
```

通过使用捕获组的编号或名称对 RegexMatch 对象进行索引，也可实现对捕获的访问：

```
julia> m=match(r"(?<hour>\d+):(?<minute>\d+)","12:45")
RegexMatch("12:45", hour="12", minute="45")

julia> m[:minute]
"45"

julia> m[2]
"45"
```

使用 replace 时利用 \n 引用第 n 个捕获组和给替换字符串加上 s 的前缀，可以实现替换字符串中对捕获的引用。捕获组 0 指的是整个匹配对象。可在替换中用 \g<groupname> 对命名捕获组进行引用。例如：

```
julia> replace("first second", r"(\w+) (?<agroup>\w+)" => s"\g<agroup> \1")
"second first"
```

为明确起见，编号捕获组也可用 \g<n> 进行引用，例如：

```
julia> replace("a", r"." => s"\g<0>1")
"a1"
```

你可以在后双引号的后面加上 i, m, s 和 x 等标志对正则表达式进行修改。这些标志和 Perl 里面的含义一样，详见以下对 perlre 手册的摘录：

```
i   不区分大小写的模式匹配。

    若区域设置规则有效，则小于 255 码位的大小写映射将取自当前区域设置，
    更大代码点的部分取自 Unicode 规则。
    然而，跨越 Unicode 规则和非 Unicode 规则边界 (ords 255/256) 时的匹配将失败。

m   将字符串视为多行。
    也就是说，将 "^" 和 "$"，使其从匹配字符串的开头和结尾，
    变为匹配字符串中任意一行的开头或结尾。

s   将字符串视为单行。
    也就是说，将 "." 以匹配任何字符，即使是通常不能匹配的换行符。

    同时使用时 r""ms，它们让 "." 匹配任何字符，
    也同时支持用 "^" 和 "$" 分别匹配换行后和换行前的字符串。

x   让正则表达式解析器忽略大部分空白：既不是反斜杠也不属于字符类的空白。
    它可以用来把正则表达式分解成更易读的部分。
    和普通代码中一样 `#` 字符用于开始注释。
```

例如，下面的正则表达式已打开所有三个标志：

```
julia> r"a+.*b+.*?d$"ism
r"a+.*b+.*?d$"ims

julia> match(r"a+.*b+.*?d$"ism, "Goodbye,\nOh, angry,\nBad world\n")
RegexMatch("angry,\nBad world")
```

r"..." 文本的构造没有插值和转义（除了引号 " 仍然需要转义）。下面例子展示了它和标准字符串字面量之间的差别：

```
julia> x = 10
10

julia> r"$x"
r"$x"

julia> "$x"
"10"

julia> r"\x"
r"\x"

julia> "\x"
ERROR: syntax: invalid escape sequence
```

Julia 也支持 r"""...""" 形式的三引号正则表达式字符串（或许便于处理包含引号和换行符的正则表达式）。

Regex() 构造函数可以用于以编程方式创建合法的正则表达式字符串。这允许在构造正则表达式字符串时使用字符串变量的内容和其他字符串操作。上面的任何正则表达式代码可以在 Regex() 的单字符串参数中使用。下面是一些例子：

```
julia> using Dates

julia> d = Date(1962,7,10)
1962-07-10

julia> regex_d = Regex("Day " * string(day(d)))
r"Day 10"

julia> match(regex_d, "It happened on Day 10")
RegexMatch("Day 10")

julia> name = "Jon"
"Jon"

julia> regex_name = Regex("[\"( ]\\Q$name\\E[\") ]")  # interpolate value of name
r"[\"( ]\QJon\E[\") ]"

julia> match(regex_name, " Jon ")
RegexMatch(" Jon ")
```

```
julia> match(regex_name, "[Jon]") === nothing
true
```

注意 \Q...\E 转义序列的使用。\Q 和 \E 之间的所有字符都被解释为字符字面量（在字符串插值之后）。This is convenient for matching characters that would otherwise be regex metacharacters. However, caution is needed when using this feature together with string interpolation, since the interpolated string might itself contain the \E sequence, unexpectedly terminating literal matching. User inputs need to be sanitized before inclusion in a regex.

7.9 字节数组字面量

另一个有用的非标准字符串字面量是字节数组字面量：b"..."。这种形式使你能够用字符串表示法来表达只读字面量字节数组，也即 UInt8 值的数组。字节数组字面量的规则如下：

- ASCII 字符和 ASCII 转义生成单个字节。
- \x 和八进制转义序列生成与转义值对应的字节。
- Unicode 转义序列生成编码 UTF-8 中该代码点的字节序列。

这些规则有一些重叠，这是因为 \x 的行为和小于 0x80(128) 的八进制转义被前两个规则同时包括了；但这两个规则又是一致的。通过这些规则可以方便地同时使用 ASCII 字符，任意字节值，以及 UTF-8 序列来生成字节数组。下面是一个用到全部三个规则的例子：

```
julia> b"DATA\xff\u2200"
8-element Base.CodeUnits{UInt8, String}:
 0x44
 0x41
 0x54
 0x41
 0xff
 0xe2
 0x88
 0x80
```

其中，ASCII 字符串"DATA" 对应于字节 68, 65, 84, 65。\xff 生成单个字节 255。Unicode 转义 \u2200 在 UTF-8 中被编码为三个字节 226, 136, 128。注意生成的字节数组不对应任何有效 UTF-8 字符串。

```
julia> isvalid("DATA\xff\u2200")
false
```

正如前面所述，CodeUnits{UInt8,String} 类型的行为类似于只读 UInt8 数组。如果需要标准数组，你可使用 Vector{UInt8} 进行转换。

```
julia> x = b"123"
3-element Base.CodeUnits{UInt8, String}:
 0x31
 0x32
```

```
 0x33

julia> x[1]
0x31

julia> x[1] = 0x32
ERROR: CanonicalIndexError: setindex! not defined for Base.CodeUnits{UInt8, String}
[...]

julia> Vector{UInt8}(x)
3-element Vector{UInt8}:
 0x31
 0x32
 0x33
```

同时，要注意到 `\xff` 和 `\uff` 之间的显著差别：前面的转义序列编码为字节 *255*，而后者代表 代码 *255*，它在 UTF-8 中编码为两个字节：

```
julia> b"\xff"
1-element Base.CodeUnits{UInt8, String}:
 0xff

julia> b"\uff"
2-element Base.CodeUnits{UInt8, String}:
 0xc3
 0xbf
```

字符字面量也用到了相同的行为。

对于小于 `\u80` 的代码，每个代码的 UTF-8 编码恰好只是由相应 `\x` 转义产生的单个字节，因此忽略两者的差别无伤大雅。然而，从 `x80` 到 `\xff` 的转义比起从 `u80` 到 `\uff` 的转义来，就有一个主要的差别：前者都只编码为一个字节，它没有形成任何有效 UTF-8 数据，除非它后面有非常特殊的连接字节；而后者则都代表 2 字节编码的 Unicode 代码。

如果这些还是太难理解，试着读一下 " 每个软件开发人员绝对必须知道的最基础 Unicode 和字符集知识"。它是一个优质的 Unicode 和 UTF-8 指南，或许能帮助解除一些这方面的疑惑。

7.10 版本号字面量

版本号很容易用 `v"..."` 形式的非标准字符串字面量表示。版本号字面量生成遵循语义版本规范的 `VersionNumber` 对象，因此由主、次、补丁号构成，后跟预发行 (pre-release) 和生成阿尔法数注释（build alpha-numeric）。例如，`v"0.2.1-rc1+win64"` 可分为主版本号 `0`，次版本号 `2`，补丁版本号 `1`，预发行版号 `rc1`，以及生成版本 `win64`。输入版本字面量时，除了主版本号以外所有内容都是可选的，因此 `v"0.2"` 等效于 `v"0.2.0"`（预发行号和生成注释为空），`v"2"` 等效于 `v"2.0.0"`，等等。

`VersionNumber` 对象在轻松正确地比较两个（或更多）版本时非常有用。例如，常数 `VERSION` 把 Julia 的版本号保留为一个 `VersionNumber` 对象，因此可以像下面这样用简单的声明定义一些特定版本的行为：

```
if v"0.2" <= VERSION < v"0.3-"
    # 针对 0.2 发行版系列做些事情
end
```

注意在上例中用到了非标准版本号 v"0.3-"，其中有尾随符 -：这个符号是 Julia 标准的扩展，它可以用来表明低于任何 0.3 发行版的版本，包括所有的预发行版。所以上例中代码只能在稳定版本 0.2 上运行，而不能在 v"0.3.0-rc1" 这样的版本上运行。为了支持非稳定（即预发行）的 0.2 版本，下限检查应像这样应该改为：v"0.2-" <= VERSION。

另一个非标准版本规范扩展使得能够使用 + 来表示生成版本的上限，例如 VERSION > v"0.2-rc1+" 可以用来表示任意高于 0.2-rc1 和其任意生成版本的版本：它对 v"0.2-rc1+win64" 返回 false 而对 v"0.2-rc2" 返回 true。

在比较中使用这样的特殊版本是个好办法（特别是，总是应该对高版本使用尾随 -，除非有好理由不这样），但它们不应该被用作任何内容的实际版本，因为它们在语义版本控制方案中无效。

除了用于定义常数 VERSION，VersionNumber 对象在 Pkg 模块应用广泛，常用于指定软件包的版本及其依赖。

7.11 原始字符串字面量

无插值和非转义的原始字符串可用 raw"..." 形式的非标准字符串字面量表示。原始字符串字面量生成普通的 String 对象，它无需插值和非转义地包含和输入完全一样的封闭式内容。这对于包含其他语言中使用" 或 \" 作为特殊字符的代码或标记的字符串很有用。

例外的是，引号仍必须转义，例如 raw"\"" 等效于 "\""。为了能够表达所有字符串，反斜杠也必须转义，不过只是当它刚好出现在引号前面时。

```
julia> println(raw"\\ \\\"")
\\ \"
```

请注意，前两个反斜杠在输出中逐字显示，这是因为它们不是在引号前面。然而，接下来的一个反斜杠字符转义了后面的一个反斜杠；又由于这些反斜杠出现在引号前面，最后一个反斜杠转义了一个引号。

Chapter 8

函数

在 Julia 里，函数是将参数值组成的元组映射到返回值的一个对象。Julia 的函数不是纯粹的数学函数，因为这些函数可以改变程序的全局状态并且可能受其影响。在 Julia 中定义函数的基本语法是：

```
julia> function f(x,y)
           x + y
       end
f (generic function with 1 method)
```

这个函数接收两个参数 x 和 y 并返回最后一个表达式的值，这里是 x + y。

在 Julia 中定义函数还有第二种更简洁的语法。上述的传统函数声明语法等效于以下紧凑性的“赋值形式”：

```
julia> f(x,y) = x + y
f (generic function with 1 method)
```

尽管函数可以是复合表达式（见 复合表达式），但在赋值形式下，函数体必须是一个一行的表达式。简短的函数定义在 Julia 中是很常见的。非常惯用的短函数语法大大减少了打字和视觉方面的干扰。

使用传统的括号语法调用函数：

```
julia> f(2,3)
5
```

没有括号时，表达式 f 指的是函数对象，可以像任何值一样被传递：

```
julia> g = f;

julia> g(2,3)
5
```

和变量名一样，Unicode 字符也可以用作函数名：

```
julia> ∑(x,y) = x + y
∑ (generic function with 1 method)
```

```
julia> ∑(2, 3)
5
```

8.1 参数传递行为

Julia 函数参数遵循一种有时被称为” 按共享传递 (pass-by-sharing)” 的约定，这意味着当值被传递给函数时不会被复制。函数参数本身作为新的变量绑定（可以引用值的新” 名称”），很像赋值 argument_name = argument_value，因此它们所引用的对象与传递的值相同。在函数内对可变值（如 Array）所做的修改对调用者是可见的。（这与在 Scheme、大多数 Lisp、Python、Ruby 和 Perl 等其他动态语言中的行为相同）

例如，在此函数中：

```
function f(x, y)
    x[1] = 42    # 修改 x
    y = 7 + y    # y 获得新绑定，并非修改
    return y
end
```

语句 x[1] = 42 改变了对象 x，因此这个变化将在调用者为此参数传递的数组中可见。另一方面，赋值 y = 7 + y 改变了绑定（” 名称”）y 使其引用新值 7 + y，而不是修改 y 所引用的原始对象。因此不会改变调用者传递的相应参数。

这可以通过调用 f(x, y) 看出来：

```
julia> a = [4,5,6]
3-element Vector{Int64}:
 4
 5
 6

julia> b = 3
3

julia> f(a, b)  # 返回 7 + b == 10
10

julia> a  # a[1] 由 f 改为 42
3-element Vector{Int64}:
 42
  5
  6

julia> b  # 未更改
3
```

Julia 中的一个常见约定（不是语法要求），作为在调用处的提醒，这样的函数通常会被命名为：f!(x, y) 而不是 f(x, y)，表明至少有一个参数（通常是第一个）正在被修改。

参数之间共享内存

当被修改的参数与另一个参数共享内存时，修改函数的行为可能会出乎意料，这种情况被称为别名（例如，当一个参数是另一个参数的视图时）。除非函数文档字符串明确指出别名会产生预期的结果，否则调用者有责任确保在这类输入上的正确行为。

8.2 参数类型声明

您可以通过将 ::TypeName 附加到参数名称来声明函数参数的类型，就像 Julia 中的 类型声明 一样。例如，以下函数递归计算 斐波那契数列：

```
fib(n::Integer) = n ≤ 2 ? one(n) : fib(n-1) + fib(n-2)
```

并且 ::Integer 规范意味着它只有在 n 是 抽象 Integer 类型的子类型时才可调用。

参数类型声明**通常对性能没有影响**：无论声明什么参数类型（如果有），Julia 都会为实际参数类型编译函数的特例版本。例如，调用 fib(1) 将触发专门为 Int 参数优化的特例化的 fib 的编译，它会在 fib(7) 或 fib(15) 调用时重新使用。（参数类型声明不触发额外的编译器特化的情况很少；请参阅：注意 Julia 何时不触发特例化。）

在 Julia 中声明参数类型的最常见原因是：

- **派发**：如 方法 中所述，对于不同的参数类型，你可以有不同版本（"方法"）的函数，在这种情况下，参数类型用于确定调用哪个版本的函数。例如，你可以使用 Binet 公式 实现一个完全不同的算法 fib(x::Number) = ...，该算法扩展为了非整数值，适用于任何 Number 类型。
- **正确性**：如果函数只为某些参数类型返回正确的结果，则类型声明会很有用。例如，如果我们省略参数类型并写成 fib(n) = n ≤ 2 ? one(n) : fib(n-1) + fib(n-2)，然后 fib(1.5) 会默默地给我们无意义的答案 1.0。
- **清晰性**：类型声明可以作为一种关于预期参数的文档形式。

但是，**过分限制参数类型是常见的错误**，这会不必要地限制函数的适用性，并防止它在未预料到的情况下被重用。例如，上面的 fib(n::Integer) 函数同样适用于 Int 参数（机器整数）和 BigInt 任意精度整数（参见 BigFloats 和 BigInts），这样十分有效，因为斐波那契数以指数方式快速增长，并且会迅速溢出任何固定精度类型，如 Int（参见 溢出行为）。但是，如果我们将函数声明为 fib(n::Int)，那么 BigInt 的应用就会被阻止。通常，应该对参数使用最通用的适用抽象类型，并且**如有不确定，就省略参数类型**。如果有必要，你可以随时添加参数类型规范，并且不会因为省略它们而牺牲性能或功能。

8.3 return 关键字

函数返回的值是最后计算的表达式的值，默认情况下，它是函数定义主体中的最后一个表达式。在上一小节的示例函数 f 中，返回值是表达式的 x + y 值。与在 C 语言和大多数其他命令式或函数式语言中一样，return 关键字会让函数立即返回，从而提供返回值的表达式：

```
function g(x,y)
    return x * y
    x + y
end
```

由于函数定义可以输入到交互式会话中，因此可以很容易的比较这些定义：

```
julia> f(x,y) = x + y
f (generic function with 1 method)

julia> function g(x,y)
           return x * y
           x + y
       end
g (generic function with 1 method)

julia> f(2,3)
5

julia> g(2,3)
6
```

当然，在一个单纯的线性执行的函数体内，例如 g，使用 return 是没有意义的，因为表达式 x + y 永远不会被执行到，我们可以简单地把 x * y 写为最后一个表达式从而省略掉 return。然而在使用其他控制流程的函数体内，return 却是有用的。例如，在计算两条边长分别为 x 和 y 的三角形的斜边长度时可以避免溢出：

```
julia> function hypot(x,y)
           x = abs(x)
           y = abs(y)
           if x > y
               r = y/x
               return x*sqrt(1+r*r)
           end
           if y == 0
               return zero(x)
           end
           r = x/y
           return y*sqrt(1+r*r)
       end
hypot (generic function with 1 method)

julia> hypot(3, 4)
5.0
```

这个函数有三个可能的返回处，返回三个不同表达式的值，具体取决于 x 和 y 的值。最后一行的 return 可以省略，因为它是最后一个表达式。

返回类型

也可以使用 :: 运算符在函数声明中指定返回类型。这可以将返回值转换为指定的类型。

```
julia> function g(x, y)::Int8
           return x * y
       end;

julia> typeof(g(1, 2))
Int8
```

这个函数将忽略 x 和 y 的类型，返回 Int8 类型的值。有关返回类型的更多信息，请参见类型声明。

返回类型声明在 Julia 中**很少使用**：通常，你应该编写“类型稳定”的函数，Julia 的编译器可以在其中自动推断返回类型。更多信息请参阅 性能提示 一章。

返回 nothing

对于不需要任何返回值的函数（只用来产生副作用的函数），Julia 中的写法为返回值nothing:

```
function printx(x)
    println("x = $x")
    return nothing
end
```

这在某种意义上是一个“惯例”，在 Julia 中 nothing 不是一个关键字，而是 Nothing 类型的一个单例（singleton）。也许你已经注意到 printx 函数有点不自然，因为 println 实际上已经会返回 nothing，所以 return 语句是多余的。

有两种比 return nothing 更短的写法：一种是直接写 return 这会隐式的返回 nothing。另一种是在函数的会后一行写上 nothing，因为函数会隐式的返回最后一个表达式的值。三种写法使用哪一种取决于代码风格的偏好。

8.4 操作符也是函数

在 Julia 中，大多数操作符只不过是支持特殊语法的函数（&& 和 || 等具有特殊评估语义的操作符除外，他们不能是函数，因为短路求值要求在计算整个表达式的值之前不计算每个操作数）。因此，您也可以使用带括号的参数列表来使用它们，就和任何其他函数一样：

```
julia> 1 + 2 + 3
6

julia> +(1,2,3)
6
```

中缀表达式和函数形式完全等价。——事实上，前一种形式会被编译器转换为函数调用。这也意味着你可以对操作符，例如 + 和 * ，进行赋值和传参，就像其它函数传参一样。

```
julia> f = +;

julia> f(1,2,3)
6
```

然而，函数以 f 命名时不再支持中缀表达式。

8.5 具有特殊名称的操作符

有一些特殊的表达式对应的函数调用没有显示的函数名称，它们是：

Note that expressions similar to [A; B;; C; D;; ...] but with more than two consecutive ; also correspond to hvncat calls.

表达式	函数调用
`[A B C ...]`	hcat
`[A; B; C; ...]`	vcat
`[A B; C D; ...]`	hvcat
`[A; B;; C; D;; ...]`	hvncat
`A'`	adjoint
`A[i]`	getindex
`A[i] = x`	setindex!
`A.n`	getproperty
`A.n = x`	setproperty!

8.6 匿名函数

函数在 Julia 里是一等公民：可以指定给变量，并使用标准函数调用语法通过被指定的变量调用。函数可以用作参数，也可以当作返回值。函数也可以不带函数名称地匿名创建，使用语法如下：

```
julia> x -> x^2 + 2x - 1
#1 (generic function with 1 method)

julia> function (x)
           x^2 + 2x - 1
       end
#3 (generic function with 1 method)
```

这样就创建了一个接受一个参数 `x` 并返回当前值的多项式 `x^2+2x-1` 的函数。注意结果是个泛型函数，但是带了编译器生成的连续编号的名字。

匿名函数最主要的用法是传递给接收函数作为参数的函数。一个经典的例子是 `map`，为数组的每个元素应用一次函数，然后返回一个包含结果值的新数组：

```
julia> map(round, [1.2, 3.5, 1.7])
3-element Vector{Float64}:
 1.0
 4.0
 2.0
```

如果做为第一个参数传递给 `map` 的转换函数已经存在，那直接使用函数名称是没问题的。但是通常要使用的函数还没有定义好，这样使用匿名函数就更加方便：

```
julia> map(x -> x^2 + 2x - 1, [1, 3, -1])
3-element Vector{Int64}:
  2
 14
 -2
```

接受多个参数的匿名函数写法可以使用语法 `(x,y,z)->2x+y-z`，而无参匿名函数写作 `()->3` 。无参函数的这种写法看起来可能有些奇怪，不过它对于延迟计算很有必要。这种用法会把代码块包进一个无参函数中，后续把它当做 `f` 调用。

例如，考虑对 `get` 的调用：

```
get(dict, key) do
    # default value calculated here
    time()
end
```

上面的代码等效于使用包含代码的匿名函数调用 `get`。被包围在 do 和 end 之间，如下所示

```
get(()->time(), dict, key)
```

这里对 `time` 的调用，被包裹了它的一个无参数的匿名函数延迟了。只有当请求的键不在 `dict` 中时，才会调用该函数。

8.7 元组

Julia 有一个和函数参数与返回值密切相关的内置数据结构叫做元组（*tuple*）。一个元组是一个固定长度的容器，可以容纳任何值，但不可以被修改 (是 *immutable* 的)。元组通过圆括号和逗号来构造，其内容可以通过索引来访问：

```
julia> (1, 1+1)
(1, 2)

julia> (1,)
(1,)

julia> x = (0.0, "hello", 6*7)
(0.0, "hello", 42)

julia> x[2]
"hello"
```

注意，长度为 1 的元组必须使用逗号 `(1,)`，而 `(1)` 只是一个带括号的值。`()` 表示空元组（长度为 0）。

8.8 具名元组

元组的元素可以有名字，这时候就有了具名元组：

```
julia> x = (a=2, b=1+2)
(a = 2, b = 3)

julia> x[1]
2

julia> x.a
2
```

除了常规索引语法（`x[1]` 或 `x[:a]`）外，还可以使用点语法 `(x.a)`，通过名称访问已命名元组的字段。

8.9 解构赋值和多返回值

逗号分隔的变量列表（可选地用括号括起来）可以出现在赋值的左侧：右侧的值通过迭代并依次赋值给每个变量来进行解构：

```
julia> (a,b,c) = 1:3
1:3

julia> b
2
```

右边的值应该是一个至少与左边的变量数量一样长的迭代器（参见迭代接口）（迭代器的任何多余元素会被忽略）。

可用于通过返回元组或其他可迭代值从函数返回多个值。例如，以下函数返回两个值：

```
julia> function foo(a,b)
           a+b, a*b
       end
foo (generic function with 1 method)
```

如果你在交互式会话中调用它且不把返回值赋值给任何变量，你会看到返回的元组：

```
julia> foo(2,3)
(5, 6)
```

解构赋值将每个值提取到一个变量中：

```
julia> x, y = foo(2,3)
(5, 6)

julia> x
5

julia> y
6
```

另一个常见用途是交换变量：

```
julia> y, x = x, y
(5, 6)

julia> x
6

julia> y
5
```

如果只需要迭代器元素的一个子集，一个常见的惯例是将忽略的元素分配给一个只包含下划线 _ 的变量（这是一个无效的变量名，请参阅 合法的变量名）：

```
julia> _, _, _, d = 1:10
1:10

julia> d
4
```

其他有效的左侧表达式可以用作赋值列表的元素，它们将调用 `setindex!` 或 `setproperty!`，或者递归地解构迭代器的各个元素：

```
julia> X = zeros(3);

julia> X[1], (a,b) = (1, (2, 3))
(1, (2, 3))

julia> X
3-element Vector{Float64}:
 1.0
 0.0
 0.0

julia> a
2

julia> b
3
```

Julia 1.6

带 ... 的赋值需要 Julia 1.6

如果赋值列表中的最后一个符号后缀为 ...（称为 *slurping*），那么它将被分配给右侧迭代器剩余元素的集合或其惰性迭代器：

```
julia> a, b... = "hello"
"hello"

julia> a
'h': ASCII/Unicode U+0068 (category Ll: Letter, lowercase)

julia> b
"ello"

julia> a, b... = Iterators.map(abs2, 1:4)
Base.Generator{UnitRange{Int64}, typeof(abs2)}(abs2, 1:4)

julia> a
1

julia> b
Base.Iterators.Rest{Base.Generator{UnitRange{Int64}, typeof(abs2)},
↪  Int64}(Base.Generator{UnitRange{Int64}, typeof(abs2)}(abs2, 1:4), 1)
```

有关特定迭代器的精确处理和自定义的详细信息，请参阅 `Base.rest`。

> **Julia 1.9**
>
> `...` in non-final position of an assignment requires Julia 1.9

Slurping in assignments can also occur in any other position. As opposed to slurping the end of a collection however, this will always be eager.

```
julia> a, b..., c = 1:5
1:5

julia> a
1

julia> b
3-element Vector{Int64}:
 2
 3
 4

julia> c
5

julia> front..., tail = "Hi!"
"Hi!"

julia> front
"Hi"

julia> tail
'!': ASCII/Unicode U+0021 (category Po: Punctuation, other)
```

这是通过函数 `Base.split_rest` 实现的。

请注意，对于变参函数定义，收集参数（slurping）仍然只允许在最后位置使用。这不适用于单参数解构，因为它不影响方法分派：

```
julia> f(x..., y) = x
ERROR: syntax: invalid "..." on non-final argument
Stacktrace:
[...]

julia> f((x..., y)) = x
f (generic function with 1 method)

julia> f((1, 2, 3))
(1, 2)
```

8.10 Property destructuring

除了基于迭代的解构外，赋值的右侧也可以使用属性名称进行解构。这遵循具名元组的语法，通过使用 `getproperty` 将右侧赋值中具有相同名称的属性赋给左侧的每个变量：

```
julia> (; b, a) = (a=1, b=2, c=3)
(a = 1, b = 2, c = 3)

julia> a
1

julia> b
2
```

8.11 参数解构

析构特性也可以被用在函数参数中。如果一个函数的参数被写成了元组形式 (如 `(x, y)`) 而不是简单的符号，那么一个赋值运算 `(x, y) = argument` 将会被默认插入：

```
julia> minmax(x, y) = (y < x) ? (y, x) : (x, y)

julia> gap((min, max)) = max - min

julia> gap(minmax(10, 2))
8
```

注意在定义函数 `gap` 时额外的括号。没有它们，`gap` 函数将会是一个双参数函数，这个例子也会无法正常运行。

同样，属性解构也可用于函数参数：

```
julia> foo((; x, y)) = x + y
foo (generic function with 1 method)

julia> foo((x=1, y=2))
3

julia> struct A
           x
           y
       end

julia> foo(A(3, 4))
7
```

对于匿名函数，解构单个元组需要一个额外的逗号：

```
julia> map(((x,y),) -> x + y, [(1,2), (3,4)])
2-element Array{Int64,1}:
 3
 7
```

8.12 变参函数

定义有任意个参数的函数会带来很多便利。这类函数通常被称为"变参"函数，即"参数数量可变"的简称。你可以通过在最后一个参数后增加省略号来定义一个变参函数:

```
julia> bar(a,b,x...) = (a,b,x)
bar (generic function with 1 method)
```

变量 a 和 b 和以前一样被绑定给前两个参数，后面的参数整个做为迭代集合被绑定到变量 x 上:

```
julia> bar(1,2)
(1, 2, ())

julia> bar(1,2,3)
(1, 2, (3,))

julia> bar(1, 2, 3, 4)
(1, 2, (3, 4))

julia> bar(1,2,3,4,5,6)
(1, 2, (3, 4, 5, 6))
```

在所有这些情况下，x 被绑定到传递给 bar 的尾随值的元组。

也可以限制可以传递给函数的参数的数量，这部分内容稍后在 参数化约束的可变参数方法 中讨论。

另一方面，将可迭代集中包含的值拆解为单独的参数进行函数调用通常很方便。要实现这一点，需要在函数调用中额外使用 ... 而不仅仅只是变量：

```
julia> x = (3, 4)
(3, 4)

julia> bar(1,2,x...)
(1, 2, (3, 4))
```

在这个情况下一组值会被精确切片成一个可变参数调用，这里参数的数量是可变的。但是并不需要成为这种情况：

```
julia> x = (2, 3, 4)
(2, 3, 4)

julia> bar(1,x...)
(1, 2, (3, 4))

julia> x = (1, 2, 3, 4)
(1, 2, 3, 4)

julia> bar(x...)
(1, 2, (3, 4))
```

进一步，拆解给函数调用中的可迭代对象不需要是个元组：

```
julia> x = [3,4]
2-element Vector{Int64}:
 3
 4
```

```
julia> bar(1,2,x...)
(1, 2, (3, 4))

julia> x = [1,2,3,4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> bar(x...)
(1, 2, (3, 4))
```

此外，参数被放入的函数不一定是可变参数函数（尽管经常是）：

```
julia> baz(a,b) = a + b;

julia> args = [1,2]
2-element Vector{Int64}:
 1
 2

julia> baz(args...)
3

julia> args = [1,2,3]
3-element Vector{Int64}:
 1
 2
 3

julia> baz(args...)
ERROR: MethodError: no method matching baz(::Int64, ::Int64, ::Int64)

Closest candidates are:
  baz(::Any, ::Any)
   @ Main none:1

Stacktrace:
[...]
```

正如你所见，如果要拆解的容器（比如元组或数组）元素数量不匹配就会报错，和直接给多个参数报错一样。

8.13 可选参数

在很多情况下，函数参数有合理的默认值，因此也许不需要显式地传递。例如，`Dates` 模块中的 `Date(y, [m, d])` 函数对于给定的年（year）`y`、月（mouth）`m`、日（data）`d` 构造了 `Date` 类型。但是，`m` 和 `d` 参数都是可选的，默认值都是 `1`。这行为可以简述为：

```
julia> using Dates

julia> function date(y::Int64, m::Int64=1, d::Int64=1)
           err = Dates.validargs(Date, y, m, d)
           err === nothing || throw(err)
           return Date(Dates.UTD(Dates.totaldays(y, m, d)))
       end
date (generic function with 3 methods)
```

注意，这个定义调用了 Date 函数的另一个方法，该方法带有一个 UTInstant{Day} 类型的参数。

通过此定义，函数调用时可以带有一个、两个或三个参数，并且在只有一个或两个参数被指定时后，自动传递 1 为未指定参数值：

```
julia> date(2000, 12, 12)
2000-12-12

julia> date(2000, 12)
2000-12-01

julia> date(2000)
2000-01-01
```

可选参数实际上只是一种方便的语法，用于编写多种具有不同数量参数的方法定义（请参阅 可选参数和关键字的参数的注意事项）。这可通过调用 methods 函数来检查我们的 date 函数示例。

```
julia> methods(date)
# 3 methods for generic function "date":
[1] date(y::Int64) in Main at REPL[1]:1
[2] date(y::Int64, m::Int64) in Main at REPL[1]:1
[3] date(y::Int64, m::Int64, d::Int64) in Main at REPL[1]:1
```

8.14 关键字参数

某些函数需要大量参数，或者具有大量行为。记住如何调用这样的函数可能很困难。关键字参数允许通过名称而不是仅通过位置来识别参数，使得这些复杂接口易于使用和扩展。

例如，考虑绘制一条线的函数 plot。这个函数可能有很多选项，用来控制线条的样式、宽度、颜色等。如果它接受关键字参数，一个可行的调用可能看起来像 plot(x, y, width=2)，这里我们仅指定线的宽度。请注意，这样做有两个目的。调用更可读，因为我们能以其意义标记参数。也使得大量参数的任意子集都能以任意次序传递。

具有关键字参数的函数在签名中使用分号定义：

```
function plot(x, y; style="solid", width=1, color="black")
    ###
end
```

在函数调用时，分号是可选的：可以调用 plot(x, y, width=2) 或 plot(x, y; width=2)，但前者的风格更为常见。显式的分号只有在传递可变参数或下文中描述的需计算的关键字时是必要的。

关键字参数的默认值只在必需时求值（当相应的关键字参数没有被传入），并且按从左到右的顺序求值，因为默认值的表达式可能会参照先前的关键字参数。

关键字参数的类型可以通过如下的方式显式指定：

```
function f(;x::Int=1)
    ###
end
```

关键字参数也可以在变参函数中使用：

```
function plot(x...; style="solid")
    ###
end
```

附加的关键字参数可用 ... 收集，正如在变参函数中：

```
function f(x; y=0, kwargs...)
    ###
end
```

在 f 中，kwargs 将是一个在命名元组上的不可变键值迭代器。具名元组（以及带有 Symbol 键的字典, and other iterators yielding two-value collections with symbol as first values）可以在调用中使用分号作为关键字参数传递，例如 f(x, z=1; kwargs...)。

如果一个关键字参数在方法定义中未指定默认值，那么它就是必需的：如果调用者没有为其赋值，那么将会抛出一个 UndefKeywordError 异常：

```
function f(x; y)
    ###
end
f(3, y=5) # ok, y is assigned
f(3)      # throws UndefKeywordError(:y)
```

在分号后也可传递 key => value 表达式。例如，plot(x, y; :width => 2) 等价于 plot(x, y, width=2)。当关键字名称需要在运行时被计算时，这就很实用了。

当分号后出现裸标识符或点表达式时，标识符或字段名称隐含关键字参数名称。例如 plot(x, y; width) 等价于 plot(x, y; width=width)，plot(x, y; options.width) 等价于 plot(x, y; width=options.width)。

可选参数的性质使得可以多次指定同一参数的值。例如，在调用 plot(x, y; options..., width=2) 的过程中，options 结构也能包含一个 width 的值。在这种情况下，最右边的值优先级最高；在此例中，width 的值可以确定是 2。但是，显式地多次指定同一参数的值是不允许的，例如 plot(x, y, width=2, width=3)，这会导致语法错误。

8.15 默认值作用域的计算

当计算可选和关键字参数的默认值表达式时，只有先前的参数才在作用域内。例如，给出以下定义：

```
function f(x, a=b, b=1)
    ###
end
```

a=b 中的 b 指的是外部作用域内的 b，而不是后续参数中的 b。

8.16 函数参数中的 Do 结构

把函数作为参数传递给其他函数是一种强大的技术，但它的语法并不总是很方便。当函数参数占据多行时，这样的调用便特别难以编写。例如，考虑在具有多种情况的函数上调用 map：

```
map(x->begin
           if x < 0 && iseven(x)
               return 0
           elseif x == 0
               return 1
           else
               return x
           end
       end,
    [A, B, C])
```

Julia 提供了一个保留字 do，用于更清楚地重写此代码：

```
map([A, B, C]) do x
    if x < 0 && iseven(x)
        return 0
    elseif x == 0
        return 1
    else
        return x
    end
end
```

do x 语法创建一个带有参数 x 的匿名函数，并将其作为第一个参数传递给 map。类似地，do a,b 将创建一个有两个参数的匿名函数。请注意，do (a,b) 将创建一个单参数匿名函数，其参数是一个要解构的元组。Note that do (a,b) would create a one-argument anonymous function, whose argument is a tuple to be deconstructed. 一个简单的 do 会声明接下来是一个形式为 () -> ... 的匿名函数。

这些参数如何初始化取决于「外部」函数；在这里，map 将会依次将 x 设置为 A、B、C，再分别调用调用匿名函数，正如在 map(func, [A, B, C]) 语法中所发生的。

这种语法使得更容易使用函数来有效地扩展语言，因为调用看起来就像普通代码块。有许多可能的用法与 map 完全不同，比如管理系统状态。例如，有一个版本的 open 可以通过运行代码来确保已经打开的文件最终会被关闭：

```
open("outfile", "w") do io
    write(io, data)
end
```

这是通过以下定义实现的：

```
function open(f::Function, args...)
    io = open(args...)
    try
        f(io)
    finally
        close(io)
    end
end
```

在这里，open 首先打开要写入的文件，接着将结果输出流传递给你在 do ... end 代码快中定义的匿名函数。在你的函数退出后，open 将确保流被正确关闭，无论你的函数是正常退出还是抛出了一个异常（try/finally 结构会在 流程控制 中描述）。

使用 do 代码块语法时，查阅文档或实现有助于了解用户函数的参数是如何初始化的。

类似于其他的内部函数，do 代码块也可以 "捕获" 上一个作用域的变量。例如，上一个 open...do 的例子中变量 data 是从外部作用域捕获的。捕获变量可能会给性能优化带来挑战，详见 性能建议。

8.17 函数的复合与链式调用

Julia 中的多个函数可以用函数复合或管道连接（链式调用）组合起来。

函数的复合指的是把多个函数绑定到一起，然后作用于最先调用那个函数的参数。你可以使用函数复合运算符 (∘) 来组合函数，这样一来 (f ∘ g)(args...) 就等价于 f(g(args...)).

你可以在 REPL 和合理配置的编辑器中用 \circ<tab> 输入函数复合运算符。

例如，sqrt 和 + 可以用下面这种方式组合：

```
julia> (sqrt ∘ +)(3, 6)
3.0
```

这个语句先把数字相加，再对结果求平方根。

下一个例子组合了三个函数并把新函数作用到一个字符串组成的数组上：

```
julia> map(first ∘ reverse ∘ uppercase, split("you can compose functions like this"))
6-element Vector{Char}:
 'U': ASCII/Unicode U+0055 (category Lu: Letter, uppercase)
 'N': ASCII/Unicode U+004E (category Lu: Letter, uppercase)
 'E': ASCII/Unicode U+0045 (category Lu: Letter, uppercase)
 'S': ASCII/Unicode U+0053 (category Lu: Letter, uppercase)
 'E': ASCII/Unicode U+0045 (category Lu: Letter, uppercase)
 'S': ASCII/Unicode U+0053 (category Lu: Letter, uppercase)
```

函数的链式调用（有时也称 "使用管道" 把数据送到一系列函数中去）指的是把一个函数作用到前一个函数的输出上：

```
julia> 1:10 |> sum |> sqrt
7.416198487095663
```

在这里，sum 函数求出的和被传递到 sqrt 函数作为参数。等价的函数复合写法是：

```
julia> (sqrt ∘ sum)(1:10)
7.416198487095663
```

管道运算符还可以和广播一起使用（.|>），这提供了一个有用的链式调用/管道 + 向量化运算的组合语法（接下来将描述）。

```
julia> ["a", "list", "of", "strings"] .|> [uppercase, reverse, titlecase, length]
4-element Vector{Any}:
  "A"
  "tsil"
  "Of"
 7
```

当将管道与匿名函数结合使用时，如果不希望后续的管道被解析为匿名函数体的一部分，必须使用括号。比较：

```
julia> 1:3 .|> (x -> x^2) |> sum |> sqrt
3.7416573867739413

julia> 1:3 .|> x -> x^2 |> sum |> sqrt
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

8.18 向量化函数的点语法

在科学计算语言中，通常会有函数的「向量化」版本，它简单地将给定函数 f(x) 作用于数组 A 的每个元素，接着通过 f(A) 生成一个新数组。这种语法便于数据处理，但在其它语言中，向量化通常也是性能所需要的：如果循环很慢，函数的「向量化」版本可以调用由低级语言编写的、快速的库代码。在 Julia 中，向量化函数不是性能所必需的，实际上编写自己的循环通常也是有益的（请参阅 Performance Tips），但它们仍然很方便。因此，任何 Julia 函数 f 能够以元素方式作用于任何数组（或者其它集合），这通过语法 f.(A) 实现。例如，sin 可以作用于向量 A 中的所有元素，如下所示：

```
julia> A = [1.0, 2.0, 3.0]
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> sin.(A)
3-element Vector{Float64}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

当然，你如果为 f 编写了一个专门的「向量化」方法，例如通过 f(A::AbstractArray) = map(f, A)，可以省略点号，这和 f.(A) 一样高效。但这种方法要求你事先决定要进行向量化的函数。

更一般地，f.(args...) 实际上等价于 broadcast(f, args...)，它允许你操作多个数组（甚至是不同形状的），或是数组和标量的混合（请参阅 Broadcasting）。例如，如果有 f(x,y) = 3x + 4y，那

么 f.(pi,A) 将为 A 中的每个 a 返回一个由 f(pi,a) 组成的新数组，而 f.(vector1,vector2) 将为每个索引 i 返回一个由 f(vector1[i],vector2[i]) 组成的新向量（如果向量具有不同的长度则会抛出异常）。

```
julia> f(x,y) = 3x + 4y;

julia> A = [1.0, 2.0, 3.0];

julia> B = [4.0, 5.0, 6.0];

julia> f.(pi, A)
3-element Vector{Float64}:
 13.42477796076938
 17.42477796076938
 21.42477796076938

julia> f.(A, B)
3-element Vector{Float64}:
 19.0
 26.0
 33.0
```

关键字参数不会被广播，而是简单地传递给函数的每次调用。例如，round.(x, digits=3) 等价于 broadcast(x -> round(x, digits=3), x)。

此外，嵌套的 f.(args...) 调用会被融合到一个 broadcast 循环中。例如，sin.(cos.(X)) 等价于 broadcast(x -> sin(cos(x)), X)，类似于 [sin(cos(x)) for x in X]：在 X 上只有一个循环，并且只为结果分配了一个数组。[相反，在典型的「向量化」语言中，sin(cos(X)) 首先会为 tmp=cos(X) 分配第一个临时数组，然后在单独的循环中计算 sin(tmp)，再分配第二个数组。] 这种循环融合不是可能发生也可能不发生的编译器优化，只要遇到了嵌套的 f.(args...) 调用，它就是一个语法保证。技术上，一旦遇到「非点」函数调用，融合就会停止；例如，在 sin.(sort(cos.(X))) 中，由于插入的 sort 函数，sin 和 cos 无法被合并。

最后，最大效率通常在向量化操作的输出数组被预分配时实现，这样重复调用就不会一次又一次地为结果分配新数组（请参阅 输出预分配）。一个方便的语法是 X .= ...，它等价于 broadcast!(identity, X, ...)，除了上面提到的，broadcast! 循环可与任何嵌套的「点」调用融合。例如，X .= sin.(Y) 等价于 broadcast!(sin, X, Y)，用 sin.(Y) in-place 覆盖 X。如果左边是数组索引表达式，例如 X[2:end] .= sin.(Y)，那就将 broadcast! 转换在一个 view 上，例如 broadcast!(sin, view(X, 2:lastindex(X)), Y)，这样左侧就被 in-place 更新了。

由于在表达式中为许多操作和函数调用添加点可能很乏味并导致难以阅读的代码，宏 @. 用于将表达式中的每个函数调用、操作和赋值转换为「点」版本。

```
julia> Y = [1.0, 2.0, 3.0, 4.0];

julia> X = similar(Y); # pre-allocate output array

julia> @. X = sin(cos(Y)) # equivalent to X .= sin.(cos.(Y))
4-element Vector{Float64}:
  0.5143952585235492
 -0.4042391538522658
 -0.8360218615377305
 -0.6080830096407656
```

像 `.+` 这样的二元（或一元）运算符使用相同的机制进行管理：它们等价于 `broadcast` 调用且可与其它嵌套的「点」调用融合。`X .+= Y` 等等价于 `X .= X .+ Y`，结果为一个融合的 in-place 赋值；另见 dot operators。

您也可以使用 `|>` 将点操作与函数链组合在一起，如本例所示：

```
julia> 1:5 .|> [x->x^2, inv, x->2*x, -, isodd]
5-element Vector{Real}:
    1
    0.5
    6
   -4
 true
```

8.19 更多阅读

我们应该在这里提到，这远不是定义函数的完整图景。Julia 拥有一个复杂的类型系统并且允许对参数类型进行多重分派。这里给出的示例都没有为它们的参数提供任何类型注释，意味着它们可以作用于任何类型的参数。类型系统在类型中描述，而方法则描述了根据运行时参数类型上的多重分派所选择的方法定义函数。

Chapter 9

流程控制

Julia 提供了大量的流程控制构件：

- 复合表达式：begin 和 ;。
- 条件表达式：if-elseif-else 和 ?: (三元运算符)。
- 短路求值：逻辑运算符 && （与）和 || （或），以及链式比较。
- 重复执行：循环：while 和 for。
- 异常处理：try-catch、error 和 throw。
- Task（协程）：yieldto。

前五个流程控制机制是高级编程语言的标准。Task 不是那么的标准：它提供了非局部的流程控制，这使得在暂时挂起的计算任务之间进行切换成为可能。这是一个功能强大的构件：Julia 中的异常处理和协同多任务都是通过 Task 实现的。虽然日常编程并不需要直接使用 Task，但某些问题用 Task 处理会更加简单。

9.1 复合表达式

有时一个表达式能够有序地计算若干子表达式，并返回最后一个子表达式的值作为它的值是很方便的。Julia 有两个组件来完成这个：begin 代码块和 ; 链。这两个复合表达式组件的值都是最后一个子表达式的值。下面是一个 begin 代码块的例子：

```
julia> z = begin
           x = 1
           y = 2
           x + y
       end
3
```

因为这些是非常简短的表达式，它们可以简单地被放到一行里，这也是 ; 链的由来：

```
julia> z = (x = 1; y = 2; x + y)
3
```

这个语法在定义简洁的单行函数的时候特别有用，参见函数。尽管很典型，但是并不要求 begin 代码块是多行的，或者 ; 链是单行的：

```
julia> begin x = 1; y = 2; x + y end
3

julia> (x = 1;
        y = 2;
        x + y)
3
```

9.2 条件表达式

条件表达式（Conditional evaluation）可以根据布尔表达式的值，让部分代码被执行或者不被执行。下面是对 if-elseif-else 条件语法的分析：

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

如果表达式 x < y 是 true，那么对应的代码块会被执行；否则判断条件表达式 x > y，如果它是 true，则执行对应的代码块；如果没有表达式是 true，则执行 else 代码块。下面是一个例子：

```
julia> function test(x, y)
           if x < y
               println("x is less than y")
           elseif x > y
               println("x is greater than y")
           else
               println("x is equal to y")
           end
       end
test (generic function with 1 method)

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

elseif 和 else 代码块是可选的，并且可以使用任意多个 elseif 代码块。if-elseif-else 组件中的第一个条件表达式为 true 时，其他条件表达式才会被执行，当对应的代码块被执行后，其余的表达式或者代码块将不会被执行。

if 代码块是"有渗漏的"，也就是说它们不会引入局部作用域。这意味着在 if 语句中新定义的变量依然可以在 if 代码块之后使用，尽管这些变量没有在 if 语句之前定义过。所以，我们可以将上面的 test 函数定义为

```
julia> function test(x,y)
           if x < y
               relation = "less than"
           elseif x == y
               relation = "equal to"
           else
               relation = "greater than"
           end
           println("x is ", relation, " y.")
       end
test (generic function with 1 method)

julia> test(2, 1)
x is greater than y.
```

变量 relation 是在 if 代码块内部声明的，但可以在外部使用。然而，在利用这种行为的时候，要保证变量在所有的分支下都进行了定义。对上述函数做如下修改会导致运行时错误

```
julia> function test(x,y)
           if x < y
               relation = "less than"
           elseif x == y
               relation = "equal to"
           end
           println("x is ", relation, " y.")
       end
test (generic function with 1 method)

julia> test(1,2)
x is less than y.

julia> test(2,1)
ERROR: UndefVarError: `relation` not defined
Stacktrace:
 [1] test(::Int64, ::Int64) at ./none:7
```

if 代码块也会返回一个值，这可能对于一些从其他语言转过来的用户来说不是很直观。这个返回值就是被执行的分支中最后一个被执行的语句的返回值。所以

```
julia> x = 3
3

julia> if x > 0
           "positive!"
       else
           "negative..."
       end
"positive!"
```

需要注意的是，在 Julia 中，经常会用短路求值来表示非常短的条件表达式（单行），这会在下一节中介绍。

与 C, MATLAB, Perl, Python，以及 Ruby 不同，但跟 Java，还有一些别的严谨的类型语言类似：一个条件表达式的值如果不是 true 或者 false 的话，会返回错误：

```
julia> if 1
           println("true")
       end
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

这个错误是说，条件判断结果的类型：Int64 是错的，而不是期望的 Bool。

所谓的" 三元运算符", ?:，很类似 if-elseif-else 语法，它用于选择性获取单个表达式的值，而不是选择性执行大段的代码块。它因在很多语言中是唯一一个有三个操作数的运算符而得名：

```
a ? b : c
```

在 ? 之前的表达式 a, 是一个条件表达式，如果条件 a 是 true，三元运算符计算在 : 之前的表达式 b；如果条件 a 是 false，则执行 : 后面的表达式 c。注意，? 和 : 旁边的空格是强制的，像 a?b:c 这种表达式不是一个有效的三元表达式（但在? 和 : 之后的换行是允许的）。

理解这种行为的最简单方式是看一个实际的例子。在前一个例子中，虽然在三个分支中都有调用 println，但实质上是选择打印哪一个字符串。在这种情况下，我们可以用三元运算符更紧凑地改写。为了简明，我们先尝试只有两个分支的版本：

```
julia> x = 1; y = 2;

julia> println(x < y ? "less than" : "not less than")
less than

julia> x = 1; y = 0;

julia> println(x < y ? "less than" : "not less than")
not less than
```

如果表达式 x < y 为真，整个三元运算符会执行字符串 "less than"，否则执行字符串 "not less than"。原本的三个分支的例子需要链式嵌套使用三元运算符：

```
julia> test(x, y) = println(x < y ? "x is less than y"    :
                            x > y ? "x is greater than y" : "x is equal to y")
test (generic function with 1 method)

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

为了方便链式传值，运算符从右到左连接到一起。

重要地是，与 if-elseif-else 类似，: 之前和之后的表达式只有在条件表达式为 true 或者 false 时才会被相应地执行：

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> 1 < 2 ? v("yes") : v("no")
yes
"yes"

julia> 1 > 2 ? v("yes") : v("no")
no
"no"
```

9.3 短路求值

Julia 中的 && 和 || 运算符分别对应于逻辑“与”和“或”操作，并通常都这样使用。但是，它们具有逻辑短路的特殊性质：不一定评估其第二个参数，下面会详细介绍。（也有按位 & 和 | 运算符可用作逻辑“与”和“或”的无短路行为，但要注意 & 和 | 的评估时的优先级高于 && 和 || 。）

短路求值与条件求值非常相似。这种行为在大多数具有 && 和 || 布尔运算符的命令式编程语言中都可以找到：在一系列由这些运算符连接的布尔表达式中，为了得到整个链的最终布尔值，仅仅只有最小数量的表达式被计算。一些语言（如 Python）将它们称为 and（&&）和 or（||）。更准确地说，这意味着：

- 在表达式 a && b 中，子表达式 b 仅当 a 为 true 的时候才会被执行。
- 在表达式 a || b 中，子表达式 b 仅在 a 为 false 的时候才会被执行。

这里的原因是：如果 a 是 false，那么无论 b 的值是多少，a && b 一定是 false。同理，如果 a 是 true，那么无论 b 的值是多少，a || b 的值一定是 true。&& 和 || 都依赖于右边，但是 && 比 || 有更高的优先级。我们可以简单地测试一下这个行为：

```
julia> t(x) = (println(x); true)
t (generic function with 1 method)

julia> f(x) = (println(x); false)
f (generic function with 1 method)

julia> t(1) && t(2)
1
2
true

julia> t(1) && f(2)
1
2
false

julia> f(1) && t(2)
1
```

```
false

julia> f(1) && f(2)
1
false

julia> t(1) || t(2)
1
true

julia> t(1) || f(2)
1
true

julia> f(1) || t(2)
1
2
true

julia> f(1) || f(2)
1
2
false
```

你可以用同样的方式测试不同 && 和 || 运算符的组合条件下的关联和优先级。

这种行为在 Julia 中经常被用来作为简短 `if` 语句的替代。可以用 `<cond> && <statement>` (可读为: <cond> *and then* <statement>) 来替换 `if <cond> <statement> end`。类似的，可以用 `<cond> || <statement>` (可读为: <cond> *or else* <statement>) 来替换 `if ! <cond> <statement> end`.

例如，可以像这样定义递归阶乘：

```
julia> function fact(n::Int)
           n >= 0 || error("n must be non-negative")
           n == 0 && return 1
           n * fact(n-1)
       end
fact (generic function with 1 method)

julia> fact(5)
120

julia> fact(0)
1

julia> fact(-1)
ERROR: n must be non-negative
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] fact(::Int64) at ./none:2
 [3] top-level scope
```

无短路求值的布尔运算可以用位布尔运算符来完成，见数学运算和初等函数：& 和 |。这些是普通的函数，同时也刚好支持中缀运算符语法，但总是会计算它们的所有参数：

```
julia> f(1) & t(2)
1
2
false

julia> t(1) | t(2)
1
2
true
```

与 if, elseif 或者三元运算符中的条件表达式相同，&& 或者 || 的操作数必须是布尔值（true 或者 false）。在链式嵌套的条件表达式中，除最后一项外，使用非布尔值会导致错误：

```
julia> 1 && true
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

但在链的末尾允许使用任意类型的表达式，此表达式会根据前面的条件被执行并返回：

```
julia> true && (x = (1, 2, 3))
(1, 2, 3)

julia> false && (x = (1, 2, 3))
false
```

9.4 重复执行：循环

有两个用于重复执行表达式的组件：while 循环和 for 循环。下面是一个 while 循环的例子：

```
julia> i = 1;

julia> while i <= 3
           println(i)
           global i += 1
       end
1
2
3
```

while 循环会执行条件表达式（例子中为 i <= 5），只要它为 true，就一直执行 while 循环的主体部分。当 while 循环第一次执行时，如果条件表达式为 false，那么主体代码就一次也不会被执行。

for 循环使得常见的重复执行代码写起来更容易。像之前 while 循环中用到的向上和向下计数是可以用 for 循环更简明地表达：

```
julia> for i = 1:3
           println(i)
       end
1
2
3
```

这里的 1:3 是一个范围对象，代表数字 1, 2, 3 的序列。for 循环在这些值之中迭代，对每一个变量 i 进行赋值。for 循环与之前 while 循环的一个非常重要区别是作用域，即变量的可见性。for 循环总是在其主体中引入一个新的迭代变量，无论在外部作用域中是否存在同名变量。这意味着一方面 i 不需要在循环之前声明。另一方面，它在循环外部不可见，同名的外部变量也不会受到影响。你需要一个新的交互式会话实例或者一个不同的变量.

```
julia> for j = 1:3
           println(j)
       end
1
2
3

julia> j
ERROR: UndefVarError: `j` not defined
```

```
julia> j = 0;

julia> for j = 1:3
           println(j)
       end
1
2
3

julia> j
0
```

使用 for outer 可以修改后一种行为并重用现有的局部变量。

参见变量作用域获取关于变量作用域的详细解释，以及 outer 及其在 Julia 中的工作方式。

一般来说，for 循环组件可以用于迭代任一个容器。在这种情况下，相比 =，另外的（但完全相同）关键字 in 或者 ∈ 则更常用，因为它使得代码更清晰：

```
julia> for i in [1,4,0]
           println(i)
       end
1
4
0

julia> for s ∈ ["foo","bar","baz"]
           println(s)
       end
foo
bar
baz
```

在手册后面的章节中会介绍和讨论各种不同的迭代容器（比如，多维数组）。

为了方便，我们可能会在测试条件不成立之前终止一个 while 循环，或者在访问到迭代对象的结尾之前停止一个 for 循环，这可以用关键字 break 来完成：

```
julia> i = 1;

julia> while true
           println(i)
           if i >= 3
               break
           end
           global i += 1
       end
1
2
3

julia> for j = 1:1000
           println(j)
           if j >= 3
               break
           end
       end
1
2
3
```

没有关键字 `break` 的话，上面的 `while` 循环永远不会自己结束，而 `for` 循环会迭代到 1000，这些循环都可以使用 `break` 来提前结束。

在某些场景下，需要直接结束此次迭代，并立刻进入下次迭代，`continue` 关键字可以用来完成此功能：

```
julia> for i = 1:10
           if i % 3 != 0
               continue
           end
           println(i)
       end
3
6
9
```

这是一个有点做作的例子，因为我们可以通过否定这个条件，把 `println` 调用放到 `if` 代码块里来更简洁的实现同样的功能。在实际应用中，在 `continue` 后面还会有更多的代码要运行，并且调用 `continue` 的地方可能会有多个。

多个嵌套的 `for` 循环可以合并到一个外部循环，可以用来创建其迭代对象的笛卡尔积：

```
julia> for i = 1:2, j = 3:4
           println((i, j))
       end
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

有了这个语法，迭代变量依然可以正常使用循环变量来进行索引，例如 `for i = 1:n, j = 1:i` 是合法的，但是在一个循环里面使用 `break` 语句则会跳出整个嵌套循环，不仅仅是内层循环。每次内层循环运行的时候，变量（`i` 和 `j`）会被赋值为他们当前的迭代变量值。所以对 `i` 的赋值对于接下来的迭代是不可见的：

```
julia> for i = 1:2, j = 3:4
           println((i, j))
           i = 0
       end
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

如果这个例子给每个变量一个关键字 `for` 来重写，那么输出会不一样：第二个和第四个变量包含 `0`。

可以使用 `zip` 在单个 `for` 循环中同时迭代多个容器：

```
julia> for (j, k) in zip([1 2 3], [4 5 6 7])
           println((j,k))
       end
(1, 4)
(2, 5)
(3, 6)
```

使用 `zip` 将创建一个迭代器，它是一个包含传递给它的容器的子迭代器的元组。`zip` 迭代器将按顺序迭代所有子迭代器，在 `for` 循环的第 i 次迭代中选择每个子迭代器的第 i 个元素。一旦任何子迭代器用完，`for` 循环就会停止。

9.5 异常处理

当一个意外条件发生时，一个函数可能无法向调用者返回一个合理的值。在这种情况下，最好让意外条件终止程序并打印出调试的错误信息，或者根据程序员预先提供的异常处理代码来采取恰当的措施。

内置的 Exception

当一个意外的情况发生时，会抛出 `Exception`。下面列出的内置 `Exception` 都会中断正常的控制流程。

例如，当输入参数为负实数时，`sqrt` 函数会抛出一个 `DomainError`：

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex result if called with
↪  a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

你可能需要根据下面的方式来定义你自己的异常：

Exception
ArgumentError
BoundsError
CompositeException
DimensionMismatch
DivideError
DomainError
EOFError
ErrorException
InexactError
InitError
InterruptException
InvalidStateException
KeyError
LoadError
OutOfMemoryError
ReadOnlyMemoryError
RemoteException
MethodError
OverflowError
Meta.ParseError
SystemError
TypeError
UndefRefError
UndefVarError
StringIndexError

```
julia> struct MyCustomException <: Exception end
```

throw 函数

我们可以用 throw 显式地创建异常。例如，若一个函数只对非负数有定义，当输入参数是负数的时候，可以用 throw 抛出一个 DomainError。

```
julia> f(x) = x>=0 ? exp(-x) : throw(DomainError(x, "argument must be nonnegative"))
f (generic function with 1 method)

julia> f(1)
0.36787944117144233

julia> f(-1)
ERROR: DomainError with -1:
argument must be nonnegative
Stacktrace:
 [1] f(::Int64) at ./none:1
```

注意 DomainError 后面不接括号的话不是一个异常，而是一个异常类型。我们需要调用它来获得一个 Exception 对象：

```
julia> typeof(DomainError(nothing)) <: Exception
true

julia> typeof(DomainError) <: Exception
false
```

另外，一些异常类型会接受一个或多个参数来进行错误报告：

```
julia> throw(UndefVarError(:x))
ERROR: UndefVarError: `x` not defined
```

我们可以仿照 `UndefVarError` 的写法，用自定义异常类型来轻松实现这个机制：

```
julia> struct MyUndefVarError <: Exception
           var::Symbol
       end

julia> Base.showerror(io::IO, e::MyUndefVarError) = print(io, e.var, " not defined")
```

Note

错误信息的第一个单词最好用小写。例如：

`size(A) == size(B) || throw(DimensionMismatch("size of A not equal to size of B"))`

就比

`size(A) == size(B) || throw(DimensionMismatch("Size of A not equal to size of B"))`.

更好。

但是，有时保留大写首字母是有意义的，例如函数的参数就是大写字母时：

`size(A,1) == size(B,2) || throw(DimensionMismatch("A has first dimension..."))`.

错误

我们可以用 `error` 函数生成一个 `ErrorException` 来中断正常的控制流程。

假设我们希望在计算负数的平方根时让程序立即停止执行。为了实现它，我们可以定义一个挑剔的 `sqrt` 函数，当它的参数是负数时，产生一个错误：

```
julia> fussy_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not allowed")
fussy_sqrt (generic function with 1 method)

julia> fussy_sqrt(2)
1.4142135623730951

julia> fussy_sqrt(-1)
ERROR: negative x not allowed
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
```

```
 [2] fussy_sqrt(::Int64) at ./none:1
 [3] top-level scope
```

如果另一个函数调用 fussy_sqrt 和一个负数, 它会立马返回，在交互会话中显示错误信息，而不会继续执行调用的函数：

```
julia> function verbose_fussy_sqrt(x)
           println("before fussy_sqrt")
           r = fussy_sqrt(x)
           println("after fussy_sqrt")
           return r
       end
verbose_fussy_sqrt (generic function with 1 method)

julia> verbose_fussy_sqrt(2)
before fussy_sqrt
after fussy_sqrt
1.4142135623730951

julia> verbose_fussy_sqrt(-1)
before fussy_sqrt
ERROR: negative x not allowed
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] fussy_sqrt at ./none:1 [inlined]
 [3] verbose_fussy_sqrt(::Int64) at ./none:3
 [4] top-level scope
```

try/catch 语句

通过 try / catch 语句，可以测试 Exception 并优雅处理可能会破坏应用程序的事情。例如，在下面的代码中，平方根函数会引发异常。通过在其周围放置 try / catch 块可以缓解。您可以选择如何处理此异常，无论是记录它，返回占位符值还是就像下面仅打印一句话。要注意的是在决定如何处理异常时，使用 try / catch 块比使用条件分支处理要慢得多。以下是使用 try / catch 块处理异常的更多示例：

```
julia> try
sqrt("ten")
catch e
println("You should have entered a numeric value")
end
You should have entered a numeric value
```

try/catch 语句允许保存 Exception 到一个变量中。在下面这个做作的例子中，如果 x 是可索引的，则计算 x 的第二项的平方根，否则就假设 x 是一个实数，并返回它的平方根：

```
julia> sqrt_second(x) = try
           sqrt(x[2])
       catch y
           if isa(y, DomainError)
               sqrt(complex(x[2], 0))
```

```
           elseif isa(y, BoundsError)
               sqrt(x)
           end
       end
sqrt_second (generic function with 1 method)

julia> sqrt_second([1 4])
2.0

julia> sqrt_second([1 -4])
0.0 + 2.0im

julia> sqrt_second(9)
3.0

julia> sqrt_second(-9)
ERROR: DomainError with -9.0:
sqrt was called with a negative real argument but will only return a complex result if called with
↪  a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

注意 catch 后面的字符会被一直认为是异常的名字，所以在写 try/catch 单行表达式时，需要特别小心。下面的代码**不会**在错误的情况下返回 x 的值：

```
try bad() catch x end
```

正确的做法是在 catch 后添加一个分号或者直接换行：

```
try bad() catch; x end

try bad()
catch
    x
end
```

try/catch 结构的强大之处在于，能够立即将深度嵌套的计算，展开到调用函数栈中的更高层级。在某些情况下，虽然没有发生错误，但展开栈并将值传递到更高层级的能力是很有用的。

Julia 提供了 rethrow、backtrace、catch_backtrace 和 current_exceptions 函数用于更高级的错误处理。

else 子句

Julia 1.8

此功能至少需要 Julia 1.8

在某些情况下，我们可能不仅想要适当地处理错误情况，还想要在 try 代码块成功执行时运行一些代码。为此，可以在 catch 代码块之后指定一个 else 子句，该子句会在之前没有抛出错误时运行。

与将此代码直接包含在 try 代码块中相比，这种方式的优势在于任何进一步的错误不会被 catch 子句静默捕获。

```
local x
try
    x = read("file", String)
catch
    # handle read errors
else
    # do something with x
end
```

Note

try、catch、else 和 finally 子句各自引入了自己的作用域块，所以如果一个变量仅在 try 块中定义，它就不能被 else 或 finally 子句访问：

```
julia> try
           foo = 1
       catch
       else
           foo
       end
ERROR: UndefVarError: `foo` not defined
```

在 try 块外使用 local 关键字 可以使变量在外部作用域的任何位置都可访问。

finally 子句

在进行状态改变或者使用类似文件的资源的编程时，经常需要在代码结束的时候进行必要的清理工作（比如关闭文件）。由于异常会使得部分代码块在正常结束之前退出，所以可能会让上述工作变得复杂。finally 关键字提供了一种方式，无论代码块是如何退出的，都能够让代码块在退出时运行某段代码。

这里是一个确保一个打开的文件被关闭的例子：

```
f = open("file")
try
    # 处理文件 f
finally
    close(f)
end
```

当控制流离开 try 代码块（例如，遇到 return，或者正常结束），close(f) 就会被执行。如果 try 代码块由于异常退出，这个异常会继续传递。catch 代码块可以和 try 还有 finally 配合使用。这时 finally 代码块会在 catch 处理错误之后才运行。

9.6 Tasks 任务（或协程）

Task 是一种允许计算以更灵活的方式被中断或者恢复的流程控制特性。我们提及它只是为了说明的完整性；详细的介绍参见：异步编程。

Chapter 10

变量作用域

变量的 **作用域**是代码的一个区域，在这个区域中这个变量是可访问的。给变量划分作用域有助于解决变量命名冲突。这个概念是符合直觉的：两个函数可能同时都有叫做 x 的参量，而这两个 x 并不指向同一个东西。相似地，也有很多其他的情况，代码的不同块会使用同样名字，但并不指向同一个东西。相同的变量名是否指向同一个东西的规则被称为作用域规则；这一节会详细地把这个规则讲清楚。

语言中的某些结构会引入作用域块，这是可以成为一些变量集合的作用域的代码区域。一个变量的作用域不是源代码行的任意集合；相反，它始终与这些块之一关系密切。在 Julia 中主要有两种作用域，全局作用域与 局部作用域，后者可以嵌套。在 Julia 中还存在引入“硬作用域”的构造和只引入“软作用域”的构造之间的区别，这影响到是否允许以相同的名称遮蔽全局变量。

作用域结构

引入作用域块的结构有：

结构	作用域类型	允许使用在
module, baremodule	全局	全局
struct	局部（软）	全局
for, while, try	局部（软）	global, local
macro	局部（硬）	全局
函数，do 语句块, let语句块, 数组推导, 生成器	局部（硬）	global, local

值得注意的是，这个表内没有的是 begin 块和if 块，这两个块**不会**引进新的作用域块。这两种作用域遵循的规则有点不一样，会在下面解释。

Julia 使用词法作用域，也就是说一个函数的作用域不继承自调用了函数的调用者作用域，而继承自该函数定义处作用域。举例如下，foo 中的 x 指向模块 Bar 的全局作用域中 x。

```
julia> module Bar
           x = 1
           foo() = x
       end;
```

而非调用了 foo 的作用域中的 x：

```
julia> import .Bar
```

```
julia> x = -1;

julia> Bar.foo()
1
```

因此**词法作用域**意味着，某段代码内某变量的指向只从它出现之处就可以推断出来，而不依赖于程序的执行方式。在嵌套的作用域结构里，内层作用域能”看“到所有外层作用域内变量。相对地，外层作用域不能看到内层作用域的变量。

10.1 全局作用域

每个模块会引进一个新全局作用域，与其他所有模块的全局作用域分开；无所不包的全局作用域不存在。模块可以把其他模块的变量引入到它的作用域中，通过using 或者 import语句或者通过点符号这种有资格的通路，也就是说每个模块都是所谓的命名空间或者关联着含值的名字的第一类数据结构。

如果顶层表达式包含带有关键字 `local` 的变量声明，那么该变量在该表达式外部是不可访问的。表达式内部的变量不会影响同名的全局变量。例如，在顶层的 `begin` 或 `if` 块中声明 `local x`：

```
julia> x = 1
       begin
           local x = 0
           @show x
       end
       @show x;
x = 0
x = 1
```

注意交互式提示行（即 REPL）是在模块 `Main` 的全局作用域中。

10.2 局部作用域

大多数代码块都会引入一个新的局部作用域（完整列表请参见上面的表格）。如果这样的代码块在语法上嵌套在另一个局部作用域内，则它创建的作用域嵌套在它所出现的所有局部作用域内，而这些作用域最终都嵌套在代码被求值的模块的全局作用域内。外部作用域中的变量对于它们所包含的任何作用域都是可见的，这意味着它们可以在内部作用域中被读取和写入，除非存在一个同名的局部变量”遮蔽”了同名的外部变量。即使外部局部变量是在内部块之后（在文本上位于下方）声明的，这也是成立的。当我们说一个变量在给定作用域中”存在”时，这意味着该名称的变量存在于当前作用域嵌套其中的任何作用域中，包括当前作用域。

一些编程语言需要在使用新变量之前显式声明它们。显式声明也适用于 Julia：在任何局部作用域中，编写 `local x` 都会在该作用域中声明一个新的局部变量，无论外部作用域中是否已经存在名为 x 的变量。像这样声明每个新变量有点冗长乏味，但是，与许多其他语言一样，Julia 考虑对不存在的变量名称进行赋值以隐式声明该变量。如果当前作用域是全局的，则新变量是全局的；如果当前作用域是局部的，则新变量对最内部的局部作用域是局部的，并且在该作用域内可见，但在该作用域外不可见。如果你给现有的局部变量赋值，它总是更新现有的局部变量：你只能通过使用 `local` 关键字在嵌套范围内显式声明新的局部变量来隐藏原局部变量。特别是，这适用于在内部函数中分配的变量，这可能会让来自 Python 的用户感到惊讶，其中内部函数中的赋值会创建一个新的局部变量，除非该变量被明确声明为非局部变量。

大多数情况下，这是非常直观的，但与许多直觉行为一样，细节比人们天真地想象的要微妙得多。

当 x = <value> 出现在某局部作用域，Julia 根据赋值表达式出现位置、x 在此处已经引用的内容，采取如下规则确定表达式的意义：

1. **现存的局部变量**：如果 x **已经是一个局部变量**，那现存的局部变量 x 将被赋值；
2. ** 硬作用域：如果 x 还不是局部变量并且赋值发生的作用域结构是硬作用域（即在 `let` 语句块、函数体、宏、推导式或生成器中），则会在赋值作用域中创建一个名为 x 的新局部变量；
3. **软作用域**：如果 x **并非已经是局部变量**，并且所有包含此次赋值的作用域结构是软作用域（循环、`try/catch` 块、或者 `struct` 块），最后行为取决于全局变量 x 是否被定义：
   - 如果全局变量 x 是**未定义**，最终此次赋值会在该作用域创建一个名为 x 的新局部变量；
   - 如果全局变量 x 是**已定义**，此次赋值会被认为是有歧义的：
     * 在**非交互**的上下文（文件、eval）中，会打印一个有歧义警告，同时创建一个新局部变量；
     * 在**交互**的上下文（REPL, notebooks）中，会向全局变量 x 赋值。

你或许注意到，当某隐性局部变量（比如未经 `local x` 声明）遮掩某全局变量，非交互的上下文中硬作用域和软作用域有相同行为，除了会输出警告。方便起见，交互的上下文遵从一套更复杂的启发式规则。下面的例子将会深入讲解。

既然你知道这个规则，那就看看一些例子。每个例子都是一个新的 REPL 会话中进行的，因此每个片段中唯一的全局变量就是在该代码块中分配的全局变量。

我们将从一个良好且明确的情况开始——在一个硬作用域内赋值，在这个情况下是一个函数体，当同名的局部变量不存在时：

```
julia> function greet()
           x = "hello" # new local
           println(x)
       end
greet (generic function with 1 method)

julia> greet()
hello

julia> x # global
ERROR: UndefVarError: `x` not defined
```

在 `greet` 函数内部，赋值 `x = "hello"` 导致 x 成为函数作用域中的一个新局部变量。有两个相关的事实：赋值发生在局部作用域内，并且没有现有的局部 x 变量。由于 x 是局部的，所以是否存在名为 x 的全局变量并不重要。例如，我们在定义和调用 `greet` 之前定义了 `x = 123`：

```
julia> x = 123 # global
123

julia> function greet()
           x = "hello" # new local
           println(x)
       end
greet (generic function with 1 method)

julia> greet()
```

```
hello

julia> x # global
123
```

由于 greet 中的 x 是局部的，全局 x 的值（或缺少值）不会受到调用 greet 的影响。硬作用域规则不关心名为 x 的全局变量是否存在：在硬作用域中对 x 的赋值是局部的（除非 x 被声明为全局的）。

我们将考虑的下一个明确的情况是已经有一个名为 x 的局部变量，在这种情况下，x = 1 总是赋值给这个现有的局部 x。无论赋值发生在同一局部作用域、同一函数体的内部局部作用域，还是嵌套在另一个函数内部的函数体（也称为 闭包）。

我们将使用 sum_to 函数，它计算从 1 到 n 的整数之和，例如：

```
function sum_to(n)
    s = 0 # new local
    for i = 1:n
        s = s + i # assign existing local
    end
    return s # same local
end
```

与前面的示例一样，在 sum_to 函数先对 s 的第一次赋值导致 s 成为函数体中的一个新局部变量。for 循环在函数作用域内有自己的内部局部作用域。在 s = s + i 出现的地方，s 已经是一个局部变量，所以赋值更新了现有的 s 而不是创建一个新的局部变量。我们可以通过在 REPL 中调用 sum_to 来测试：

```
julia> function sum_to(n)
           s = 0 # new local
           for i = 1:n
               s = s + i # assign existing local
           end
           return s # same local
       end
sum_to (generic function with 1 method)

julia> sum_to(10)
55

julia> s # global
ERROR: UndefVarError: `s` not defined
```

由于 s 是函数 sum_to 的局部变量，调用该函数对全局变量 s 没有影响。我们还可以看到，for 循环中的更新 s = s + i 必须更新由初始化 s = 0 创建的相同 s，因为我们得到了整数 1 到 10 的正确总和 55。

让我们通过编写一个稍微详细一点的变体来深入了解一下 for 循环体有自己的作用域，我们将其称为 sum_to_def，其中，在更新 s 之前，我们将和 s + i 保存在一个变量中 t：

```
julia> function sum_to_def(n)
           s = 0 # new local
           for i = 1:n
```

```
            t = s + i # new local `t`
            s = t # assign existing local `s`
        end
        return s, @isdefined(t)
    end
sum_to_def (generic function with 1 method)

julia> sum_to_def(10)
(55, false)
```

这个版本像先前一样返回 s，但它也使用 @isdefined 宏返回一个布尔值，指示是否在函数的最外层局部作用域中定义了一个名为 t 的局部变量。正如你所看到的，在 for 循环体之外没有定义 t。这又是因为硬作用域规则：由于对 t 的赋值发生在一个函数内部，这引入了一个硬作用域，赋值导致 t 在它出现的局部作用域中成为一个新的局部变量，即循环体内部。即使有一个名为 t 的全局变量，它也没有任何区别——硬作用域规则不受全局作用域中的任何内容的影响。

请注意，for 循环体的局部作用域与内部函数的局部作用域没有区别。这意味着我们可以重写此示例，以便将循环体实现为对内部辅助函数的调用，并且其行为方式相同：

```
julia> function sum_to_def_closure(n)
           function loop_body(i)
               t = s + i # new local `t`
               s = t # assign same local `s` as below
           end
           s = 0 # new local
           for i = 1:n
               loop_body(i)
           end
           return s, @isdefined(t)
       end
sum_to_def_closure (generic function with 1 method)

julia> sum_to_def_closure(10)
(55, false)
```

这个例子说明了几个要点：

1. 内部函数作用域就像任何其他嵌套的局部作用域一样。特别是，如果一个变量已经是内部函数之外的局部变量，并且你在内部函数中为其赋值，则外部局部变量会被更新。
2. 外部的局部变量的定义是否发生在更新位置的下方并不重要，规则保持不变。在解析内部的局部变量含义之前，解析整个封闭局部作用域并确定其局部变量。

这种设计意味着你通常可以将代码移入或移出内部函数而不改变其含义，这给使用闭包语言中的许多常见习语提供了便利。（参见 do blocks）。

让我们继续讨论软作用域规则涵盖的一些更模糊的情况。我们将通过将 greet 和 sum_to_def 函数的主体提取到软作用域上下文中来探索这一点。首先，让我们将 greet 的主体放在一个 for 循环中——它是软的，而不是硬的——并在 REPL 中运行：

```
julia> for i = 1:3
           x = "hello" # new local
           println(x)
       end
hello
hello
hello

julia> x
ERROR: UndefVarError: `x` not defined
```

由于在执行 `for` 循环时未定义全局变量 `x`，因此软作用域规则的第一个子句适用，并且 `x` 被创建为 `for` 循环内的局部变量，因此循环执行完后全局变量 `x` 一直没有定义。接下来，让我们考虑提取到全局作用域内的 `sum_to_def` 的函数体，将其参数固定为 `n = 10`

```
s = 0
for i = 1:10
    t = s + i
    s = t
end
s
@isdefined(t)
```

这段代码有什么作用？提示：这是一个小把戏。答案是"视情况而定"。如果此代码以交互方式输入，则其行为方式与在函数体中的行为方式相同。但是如果代码出现在文件中，它会打印一个歧义警告并抛出一个未定义的变量错误。让我们先看看它在 REPL 中的情况：

```
julia> s = 0 # global
0

julia> for i = 1:10
           t = s + i # new local `t`
           s = t # assign global `s`
       end

julia> s # global
55

julia> @isdefined(t) # global
false
```

REPL 内行为接近于函数体内，决定循环内部的赋值是分配给一个全局变量还是创建新的局部变量，取决于是否定义了具有该名称的全局变量。如果存在同名的全局变量，则赋值会更新它。如果不存在全局变量，则赋值会创建一个新的局部变量。在这个例子中，我们看到两种情况都在起作用：

- 没有名为 `t` 的全局变量，因此 `t = s + i` 创建了一个新的 `t`，它是 `for` 循环的局部变量；
- 有一个名为 `s` 的全局变量，因此将 `s = t` 赋值给它。

第二个情况解释了为什么循环的执行会改变 `s` 的全局值，第一个情况解释了为什么在循环执行后 `t` 仍未定义。现在，让我们尝试运行相同的代码，就像它在文件中一样：

```
julia> code = """
       s = 0 # global
       for i = 1:10
           t = s + i # new local `t`
           s = t # new local `s` with warning
       end
       s, # global
       @isdefined(t) # global
       """;

julia> include_string(Main, code)
┌ Warning: Assignment to `s` in soft scope is ambiguous because a global variable by the same name
↪   exists: `s` will be treated as a new local. Disambiguate by using `local s` to suppress this
↪   warning or `global s` to assign to the existing global variable.
└ @ string:4
ERROR: LoadError: UndefVarError: `s` not defined
```

这里我们使用 include_string 来评估 code，就好像它是文件的内容一样。我们也可以将 code 保存到一个文件中，然后对该文件调用 include——结果是一样的。如你所见，这与在 REPL 中评估相同代码的行为完全不同。让我们分解一下这里发生的事情：

- 在循环运行之前，全局 s 被定义为值 0
- 赋值 s = t 发生在软作用域中——任何函数体或其他硬作用域结构之外的 for 循环
- 因此软作用域规则的第二个子句适用，并且分配不明确，因此发出警告
- 继续执行，使 s 成为 for 循环体中的局部作用域
- 由于 s 是 for 循环的局部变量，所以在计算 t = s + i 时它是未定义的，从而导致错误
- 求值到此就结束了，但如果到了 s 和 @isdefined(t)，它将返回 0 和 false。

这展示了作用域的一些重要方面：在一个作用域中，每个变量只能有一个含义，而该含义的确定与表达式的顺序无关。循环中表达式 s = t 的存在导致 s 在循环中是局部的，这意味着当它出现在 t = s + i 的右侧时它也是局部的，即使该表达式首先出现并首先计算。有人可能会想象循环第一行上的 s 可以是全局的，而循环第二行上的 s 是局部的，但这是不可能的，因为这两行在同一个作用域块中并且每个变量在给定的作用域内只能有一种含义。

在软作用域

我们现在已经涵盖了所有局部作用域规则，但在结束本节之前，也许应该说几句关于为什么在交互式和非交互式上下文中处理模糊软作用域的情况不同。人们可以问两个明显的问题：

1. 为什么不都像 REPL 那样？
2. 为什么不都表现得像在文件中那样？并跳过警告？

在 Julia ≤ 0.6 的版本中，所有全局作用域确实像当前的 REPL 一样工作：当 x = <value> 发生在循环中（或 try/catch，struct 内）但在函数体（或 let 语句块或推导式）之外时，它根据是否定义了一个名为 x 的全局变量来决定 x 是否应该是循环的局部变量。这种行为具有直观和方便的优点，因为它尽可能接近函数体内部的行为。特别是，当尝试调试函数时，它可以轻松地在函数体和 REPL 之间来回移动代码。但是，它有一些缺点。首先，这是一种相当复杂的行为：多年来，许多人对这种行

为感到困惑，并抱怨说它既复杂又难以解释和理解。这是有道理的。其次，可以说更糟的是，它不利于“大规模”编程。当你在这样的地方看到一小段代码时，很清楚发生了什么：

```
s = 0
for i = 1:10
    s += i
end
```

显然，代码的意图是修改现有的全局变量 `s`。这还能是什么意思？然而，并非所有现实世界的代码都如此简短或清晰。我们发现像下面这样的代码经常出现：

```
x = 123

# much later
# maybe in a different file

for i = 1:10
    x = "hello"
    println(x)
end

# much later
# maybe in yet another file
# or maybe back in the first one where `x = 123`

y = x + 234
```

我们非常不清楚这里应该发生什么。由于 `x + "hello"` 是一个方法错误，似乎意图是让 `x` 在 `for` 循环中是局部的。但是运行时值和碰巧存在的方法不能用于确定变量的范围。对于 Julia ≤ 0.6 的行为，尤其令人担忧的是，有人可能先编写了 `for` 循环，让它工作得很好，但后来当其他人在远处添加了一个新的全局时——可能是在不同的文件——代码突然改变了含义，要么中断，要么更糟糕的是，默默地做执行了错误的命令。这种“幽灵般的远距离动作”是好的编程语言设计应该防止的。

因此，在 Julia 1.0 中，我们简化了作用域的规则：在任何局部作用域中，对一个还不是局部变量的名称进行赋值会创建一个新的局部变量。这完全消除了软作用域的概念，并消除了幽灵行为的可能性。由于移除了软作用域，我们发现并修复了大量错误，证明我们选择摆脱它是正确的。我们有很多的欣喜！嗯，不，不是真的。因为有些人很生气，他们现在不得不写：

```
s = 0
for i = 1:10
    global s += i
end
```

你看到那里的 `global` 注解了吗？非常令人讨厌。显然，这种情况是不能容忍的。但更严重的是，这种需要 `global` 顶层代码的情况有两个主要问题：

1. 从函数体内部复制和粘贴代码到 REPL 来 debug 不再方便——你必须加上 `global` 注释，然后把它删了再复制回去。

2. 初学者编写这种代码往往不会加 `global`，并且不知道为什么他们的代码不起作用 - 他们得到的错误是 `s` 未定义，这似乎并没有启发犯错的人。

从 Julia 1.5 开始，此代码在 REPL 或 Jupyter 笔记本（就像 Julia 0.6）等交互式上下文中无需 `global` 注解即可正确执行，同时，在文件和其他非交互式上下文中，它会打印出以下非常直接的警告：

> 在软作用域中对 s 的赋值是不明确的，因为存在同名的全局变量：s 将被视为新的局部变量。通过使用 `local s` 来消除此警告或使用 `global s` 赋值给现有的全局变量来消除歧义。

这解决了这两个问题，同时保留了 1.0 行为的“大规模编程”好处：全局变量对可能很远的代码的含义没有幽灵般的影响；在 REPL 复制粘贴调试工作，初学者没有任何问题；任何时候有人忘记 `global` 注解或不小心用软作用域中的局部变量遮蔽了现有的全局变量，这无论如何都会令人困惑，他们会得到一个很好的明确警告。

这种设计的一个重要特点是，在没有警告的情况下在文件中执行的任何代码在新的 REPL 中的行为方式相同。另一方面，如果您使用 REPL 会话并将其保存到文件中，如果它的行为与 REPL 中的行为不同，那么您将收到警告。

Let 块

`let` 语句创建一个新的 硬作用域块（见上文）并在每次运行时引入新的变量绑定。变量不必立即分配：

```
julia> var1 = let x
           for i in 1:5
               (i == 4) && (x = i; break)
           end
           x
       end
4
```

赋值可能会为现有值地址重新分配一个新值，而 `let` 总是会创建一个新地址。这种差异通常并不重要，并且只有在通过闭包超出其作用域的变量的情况下才能检测到。`let` 语法接受以逗号分隔的一系列赋值和变量名：

```
julia> x, y, z = -1, -1, -1;

julia> let x = 1, z
           println("x: $x, y: $y") # x is local variable, y the global
           println("z: $z") # errors as z has not been assigned yet but is local
       end
x: 1, y: -1
ERROR: UndefVarError: `z` not defined
```

赋值将按次序执行：作用域右侧先于左侧引入新变量前被执行。这使得类似 `let x = x` 的写法是有意义的，因为这两个 x 变量并不一样，拥有不同存储位置。`let` 的行为在如下例子中是必要的：

```
julia> Fs = Vector{Any}(undef, 2); i = 1;

julia> while i <= 2
           Fs[i] = ()->i
           global i += 1
       end

julia> Fs[1]()
```

```
3

julia> Fs[2]()
3
```

在这里，我们创建并存储了两个返回变量 `i` 的闭包。但是因为始终是同一个变量 `i`，所以这两个闭包行为是相同的。我们可以使用 `let` 为 `i` 创建新绑定：

```
julia> Fs = Vector{Any}(undef, 2); i = 1;

julia> while i <= 2
           let i = i
               Fs[i] = ()->i
           end
           global i += 1
       end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

由于 `begin` 结构不会引入新的作用域，使用零参数 `let` 来引入一个新的作用域块而不立即创建任何新的绑定是很有用的：

```
julia> let
           local x = 1
           let
               local x = 2
           end
           x
       end
1
```

由于 `let` 引入了一个新的作用域块，内部局部变量 `x` 与外部局部变量 `x` 是一个不同的变量。这个特定的例子相当于：

```
julia> let x = 1
           let x = 2
           end
           x
       end
1
```

循环和数组推导

对于循环和数组推导：在其内部作用域中引入的新变量在每次循环迭代中都会被新分配一块内存，如同被 `let` 块包围。

```
julia> Fs = Vector{Any}(undef, 2);

julia> for j = 1:2
           Fs[j] = ()->j
       end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

for 循环或者推导式的迭代变量始终是个新变量：

```
julia> function f()
           i = 0
           for i = 1:3
               # empty
           end
           return i
       end;

julia> f()
0
```

但是偶然地，把一个已有的局部变量作为迭代变量也是有用的。添加关键字 outer 就能方便地做到：

```
julia> function f()
           i = 0
           for outer i = 1:3
               # empty
           end
           return i
       end;

julia> f()
3
```

10.3 常量

变量普遍地用于命名一个特定、不变的值。这些变量只被赋值一次。向编译器传递 const 关键字，即可声明这个意图：

```
julia> const e  = 2.71828182845904523536;

julia> const pi = 3.14159265358979323846;
```

单个 const 关键字能同时声明多个变量：

```
julia> const a, b = 1, 2
(1, 2)
```

const 声明只应使用在全局作用域中的全局变量。因为全局变量的值（甚至类型）可以随时改变，编译器很难优化包含全局变量的代码。而用 const 声明一个不变的全局变量，就能处理这个问题。

局部常量却大有不同。编译器能够自动确定一个局部变量什么时候是不变的，所以局部常量声明是不必要的，其实现在也并不支持。

一些特殊的顶层赋值，比如用了 function 和 structure 关键字，默认就是常量。

注意 const 只会影响变量绑定；变量可能会绑定到一个可变的对象上（比如一个数组）使得其仍然能被改变。另外当尝试给一个声明为常量的变量赋值时，可能出现下列情景：

- 如果新赋值的类型与原常量类型不一样，会扔出一个错误：

```
julia> const x = 1.0
1.0

julia> x = 1
ERROR: invalid redefinition of constant x
```

- 如果新赋值的类型与原常量一样，会打印一个警告：

```
julia> const y = 1.0
1.0

julia> y = 2.0
WARNING: redefinition of constant y. This may fail, cause incorrect answers, or produce other
↪  errors.
2.0
```

- 如果赋值不导致原变量值变化，则不会给出任何信息：

```
julia> const z = 100
100

julia> z = 100
100
```

最后一条规则也适用于不可变对象，即使变量绑定的地址改变了，例如：

```
julia> const s1 = "1"
"1"

julia> s2 = "1"
"1"
```

```
julia> pointer.([s1, s2], 1)
2-element Array{Ptr{UInt8},1}:
 Ptr{UInt8} @0x00000000132c9638
 Ptr{UInt8} @0x0000000013dd3d18

julia> s1 = s2
"1"

julia> pointer.([s1, s2], 1)
2-element Array{Ptr{UInt8},1}:
 Ptr{UInt8} @0x0000000013dd3d18
 Ptr{UInt8} @0x0000000013dd3d18
```

然而对于可变对象，警告会如预期出现：

```
julia> const a = [1]
1-element Vector{Int64}:
 1

julia> a = [1]
WARNING: redefinition of constant a. This may fail, cause incorrect answers, or produce other
↪  errors.
1-element Vector{Int64}:
 1
```

注意，虽然有时是可能更改常量的值，但是十分不推荐这样做。这样做仅仅是为了便于交互式使用。更改常量可引发多种问题或者非预期行为。举个例子，如果一个方法引用了一个常量并且在常量被更改前已经被编译了，那么该函数很有可能继续使用旧值：

```
julia> const x = 1
1

julia> f() = x
f (generic function with 1 method)

julia> f()
1

julia> x = 2
WARNING: redefinition of constant x. This may fail, cause incorrect answers, or produce other
↪  errors.
2

julia> f()
1
```

10.4 带类型的全局变量

Julia 1.8

使用带类型的全局变量至少需要 Julia 1.8

与声明为常量类似，全局绑定也可以被声明为始终具有固定类型。这可以通过不分配实际值的语法 `global x::T` 来完成，或者在赋值时使用 `x::T = 123` 的形式。

```
julia> x::Float64 = 2.718
2.718

julia> f() = x
f (generic function with 1 method)

julia> Base.return_types(f)
1-element Vector{Any}:
 Float64
```

对全局赋值时，Julia 会首先尝试使用 `convert` 将其转换为合适的类型：

```
julia> global y::Int

julia> y = 1.0
1.0

julia> y
1

julia> y = 3.14
ERROR: InexactError: Int64(3.14)
Stacktrace:
[...]
```

类型不必是具体的，但使用抽象类型的注解通常对性能没有什么好处。

一旦全局已被赋值或其类型已被设置，则不允许绑定类型：

```
julia> x = 1
1

julia> global x::Int
ERROR: cannot set type for global x. It already has a value or is already set to a different type.
Stacktrace:
[...]
```

Chapter 11

类型

通常，我们把程序语言中的类型系统划分成两类：静态类型和动态类型。对于静态类型系统，在程序运行之前，我们就可计算每一个表达式的类型。而对于动态类型系统，我们只有通过运行那个程序，得到表达式具体的值，才能确定其具体的类型。通过让编写的代码无需在编译时知道值的确切类型，面向对象允许静态类型语言具有一定的灵活性。可以编写在不同类型上都能运行的代码的能力被称为多态。在经典的动态类型语言中，所有的代码都是多态的，这意味着这些代码对于其中值的类型没有约束，除非在代码中去具体的判断一个值的类型，或者对对象做一些它不支持的操作。

Julia 类型系统是动态的，但由于允许指出某些变量具有特定类型，因此占有静态类型系统的一些优势。这对于生成高效的代码非常有帮助，但更重要的是，它允许针对函数参数类型的方法派发与语言深度集成。方法派发将在方法中详细探讨，但它根植于此处提供的类型系统。

在类型被省略时，Julia 的默认行为是允许值为任何类型。因此，可以编写许多有用的 Julia 函数，而无需显式使用类型。然而，当需要额外的表达力时，很容易逐渐将显式的类型注释引入先前的「无类型」代码中。添加类型注释主要有三个目的：利用 Julia 强大的多重派发机制、提高代码可读性以及捕获程序错误。

用类型系统的术语描述，Julia 是动态（dynamic）、主格（nominative）和参数（parametric）的。泛型可以被参数化，并且类型之间的层次关系可以被显式地声明，而不是隐含地通过兼容的结构。Julia 类型系统的一个特别显著的特征是具体类型相互之间不能是子类型：所有具体类型都是最终的，并且超类只能是抽象类型。虽然这乍一看可能过于严格，但它有许多益处，且缺点却少得出奇。事实证明，能够继承行为比继承结构更重要，同时继承两者在传统的面向对象语言中导致了重大困难。Julia 类型系统的其它高级方面应当在先言明：

- 对象值和非对象值之间没有分别：Julia 中的所有值都是具有类型的真实对象且其类型属于一个单独的、完全连通的类型图，该类型图的所有节点作为类型一样都是头等的。
- 「编译期类型」是没有任何意义的概念：变量所具有的唯一类型是程序运行时的实际类型。这在面向对象被称为「运行时类型」，其中静态编译和多态的组合使得这种区别变得显著。
- 只有值，而不是变量，有类型——变量只是绑定到值的名称，尽管为了简单起见，我们可以说“变量的类型”作为“变量所引用的值的类型”的简写。
- 抽象类型和具体类型都可以通过其它类型进行参数化。它们的参数化还可通过符号、使得 `isbits` 返回 true 的任意类型的值（实质上，也就是像数字或布尔变量这样的东西，存储方式像 C 类型或不包含指向其它对象的指针的 `struct`）和其元组。类型参数在不需要被引用或限制时可以省略。

Julia 的类型系统设计得强大而富有表现力，却清晰、直观且不引人注目。许多 Julia 程序员可能从未感觉需要编写明确使用类型的代码。但是，某些场景的编程可通过声明类型变得更加清晰、简单、快速和稳健。

11.1 类型声明

`::` 运算符可以用来在程序中给表达式和变量附加类型注释。这有两个主要原因：

1. 作为断言，帮助程序确认能是否正常运行，
2. 给编译器提供额外的类型信息，在一些情况下这可以提升程序性能。

置于到计算值的表达式后面时，`::` 操作符读作「是 ······ 的实例（is an instance of)」。在任何地方都可以用它来断言左侧表达式的值是右侧类型的实例。当右侧类型是具体类型时，左侧的值必须能够以该类型作为其实现——回想一下，所有具体类型都是最终的，因此没有任何实现是任何其它具体类型的子类型。当右侧类型是抽象类型时，值是由该抽象类型子类型中的某个具体类型实现的才能满足该断言。如果类型断言非真，抛出一个异常，否则返回左侧的值：

```
julia> (1+2)::AbstractFloat
ERROR: TypeError: in typeassert, expected AbstractFloat, got a value of type Int64

julia> (1+2)::Int
3
```

这将允许类型断言作用在任意表达式上。

置于赋值语句左侧的变量之后，或作为 `local` 声明的一部分时，`::` 操作符的意义有所不同：它声明变量始终具有指定的类型，就像静态类型语言（如 C）中的类型声明。每个被赋给该变量的值都将使用 `convert` 转换为被声明的类型：

```
julia> function foo()
           x::Int8 = 100
           x
       end
foo (generic function with 1 method)

julia> x = foo()
100

julia> typeof(x)
Int8
```

这个特性对避免特定的性能「陷阱」很有帮助，比如给一个变量赋值时意外地更改了其类型。

此「声明」行为仅发生在特定上下文中：

```
local x::Int8  # in a local declaration
x::Int8 = 10   # as the left-hand side of an assignment
```

and applies to the whole current scope, even before the declaration.

As of Julia 1.8, type declarations can now be used in global scope i.e. type annotations can be added to global variables to make accessing them type stable.

```
julia> x::Int = 10
10

julia> x = 3.5
ERROR: InexactError: Int64(3.5)

julia> function foo(y)
           global x = 15.8    # throws an error when foo is called
           return x + y
       end
foo (generic function with 1 method)

julia> foo(10)
ERROR: InexactError: Int64(15.8)
```

声明也可以附加到函数定义：

```
function sinc(x)::Float64
    if x == 0
        return 1
    end
    return sin(pi*x)/(pi*x)
end
```

从函数返回时就如同给一个已被声明类型的变量赋值：返回值始终会被转换为 Float64。

11.2 抽象类型

抽象类型不能实例化，只能作为类型图中的节点使用，从而描述相关具体类型的集，即那些作为其后代的具体类型。即便抽象类型没有实例，由于它们是类型系统的主干，故我们首先从抽象类型谈起：抽象类型形成了概念的层次结构，这使得 Julia 的类型系统不只是对象实现的集合。

回想一下，在整数和浮点数中，我们介绍了各种数值的具体类型：Int8、UInt8、Int16、UInt16、Int32、UInt32、Int64、UInt64、Int128、UInt128、Float16、Float32 和 Float64。尽管 Int8、Int16、Int32、Int64 和 Int128 具有不同的表示大小，但都具有共同的特征，即它们都是带符号的整数类型。类似地，UInt8、UInt16、UInt32、UInt64 和 UInt128 都是无符号整数类型，而 Float16、Float32 和 Float64 是不同的浮点数类型而非整数类型。一段代码只对某些类型有意义是很常见的，比如，只在其参数是某种类型的整数，而不真正取决于特定类型的整数时有意义。例如，最大公分母算法适用于所有类型的整数，但不适用于浮点数。抽象类型允许构造类型的层次结构，这给具体类型提供了可以适应的环境。例如，你可以轻松地为任何类型的整数编程，而不用将算法限制为某种特殊类型的整数。

抽象类型可以由 abstract type 关键字来声明。声明抽象类型的一般语法是：

```
abstract type «name» end
abstract type «name» <: «supertype» end
```

该 abstract type 关键字引入了一个新的抽象类型，«name» 为其名称。此名称后面可以跟 <: 和一个已存在的类型，表示新声明的抽象类型是此「父」类型的子类型。

如果没有给出超类型，则默认超类型为 Any——一个已经定义好的抽象类型，所有对象都是 Any 的实例并且所有类型都是 Any 的子类型。在类型理论中，Any 通常称为「top」，因为它位于类型图的顶

点。Julia 还有一个预定义了的抽象「bottom」类型，在类型图的最低点，写成 Union{}。这与 Any 完全相反：任何对象都不是 Union{} 的实例，所有的类型都是 Union{} 的超类型。

让我们考虑一些构成 Julia 数值类型层次结构的抽象类型：

```
abstract type Number end
abstract type Real          <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer       <: Real end
abstract type Signed        <: Integer end
abstract type Unsigned      <: Integer end
```

Number 类型为 Any 类型的直接子类型，并且 Real 为它的子类型。接下来，Real 有两个子类型（它还有更多的子类型，但这里只展示了两个，稍后将会看到其它的子类型）：Integer 和 AbstractFloat，将世界分为整数的表示和实数的表示。实数的表示当然包括浮点类型，但也包括其他类型，例如有理数。因此，AbstractFloat 是一个 Real 的子类型，仅包括实数的浮点表示。整数被进一步细分为 Signed 和 Unsigned 两类。

<: 运算符的通常意义为「是 ······ 的子类型（is a subtype of）」，可以用在声明中，声明右侧类型是新声明类型的直接超类型；也可以在表达式中用作子类型运算符，在其左操作数为其右操作数的子类型时返回 true：

```
julia> Integer <: Number
true

julia> Integer <: AbstractFloat
false
```

抽象类型的一个重要用途是为具体类型提供默认实现。举个简单的例子，考虑：

```
function myplus(x,y)
    x+y
end
```

首先需要注意的是上述的参数声明等价于 x::Any 和 y::Any。当函数被调用时，例如 myplus(2,5)，派发器会选择与给定参数相匹配的名称为 myplus 的最具体方法。（有关多重派发的更多信息，请参阅方法。）

假设没有找到比上述方法更具体的方法，Julia 则会基于上面给出的泛型函数，在内部定义并编译一个名为 myplus 的方法，专门用于处理两个 Int 参数，即它隐式地定义并编译：

```
function myplus(x::Int,y::Int)
    x+y
end
```

最后，调用这个具体的方法。

因此，抽象类型允许程序员编写泛型函数，泛型函数可以通过许多具体类型的组合用作默认方法。多重派发使得程序员可以完全控制是使用默认方法还是更具体的方法。

需要注意的重点是，即使程序员依赖参数为抽象类型的函数，性能也不会有任何损失，因为它会针对每个调用它的参数元组的具体类型重新编译。（但在函数参数是抽象类型的容器的情况下，可能存在性能问题；请参阅性能建议。）

11.3 原始类型

Warning

通常情况下更建议在新的复合类型中封装现有的原始类型，而不是重新定义自己的原始类型。

这个功能的存在是为了允许 Julia 能引导受 LLVM 支持的标准基本类型。一旦一些标准类型被定义，就不需要再定义更多了。

原始类型是具体类型，其数据是由简单的位组成。原始类型的经典示例是整数和浮点数。与大多数语言不同，Julia 允许你声明自己的原始类型，而不是只提供一组固定的内置原始类型。实际上，标准原始类型都是在语言本身中定义的：

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char <: AbstractChar 32 end

primitive type Int8    <: Signed   8 end
primitive type UInt8   <: Unsigned 8 end
primitive type Int16   <: Signed   16 end
primitive type UInt16  <: Unsigned 16 end
primitive type Int32   <: Signed   32 end
primitive type UInt32  <: Unsigned 32 end
primitive type Int64   <: Signed   64 end
primitive type UInt64  <: Unsigned 64 end
primitive type Int128  <: Signed   128 end
primitive type UInt128 <: Unsigned 128 end
```

声明原始类型的一般语法是：

```
primitive type «name» «bits» end
primitive type «name» <: «supertype» «bits» end
```

bits 的数值表示该类型需要多少存储空间，name 为新类型指定名称。可以选择将一个原始类型声明为某个超类型的子类型。如果省略超类型，则默认 Any 为其直接超类型。上述声明中意味着 `Bool` 类型需要 8 位来储存，并且直接超类型为 `Integer`。目前支持的大小只能是 8 位的倍数，不然你就会遇到 LLVM 的 bug。因此，布尔值虽然确实只需要一位，但不能声明为小于 8 位的值。

`Bool`，`Int8` 和 `UInt8` 类型都具有相同的表现形式：它们都是 8 位内存块。然而，由于 Julia 的类型系统是主格的，它们尽管具有相同的结构，但不是通用的。它们之间的一个根本区别是它们具有不同的超类型：`Bool` 的直接超类型是 `Integer`、`Int8` 的是 `Signed` 而 `UInt8` 的是 `Unsigned`。`Bool`，`Int8` 和 `UInt8` 的所有其它差异是行为上的——定义函数的方式在这些类型的对象作为参数给定时起作用。这也是为什么主格的类型系统是必须的：如果结构确定类型，类型决定行为，就不可能使 `Bool` 的行为与 `Int8` 或 `UInt8` 有任何不同。

11.4 复合类型

复合类型在各种语言中被称为 record、struct 和 object。复合类型是命名字段的集合，其实例可以视

为单个值。复合类型在许多语言中是唯一一种用户可定义的类型，也是 Julia 中最常用的用户定义类型。

在主流的面向对象语言中，比如 C++、Java、Python 和 Ruby，复合类型也具有与它们相关的命名函数，并且该组合称为「对象」。在纯粹的面向对象语言中，例如 Ruby 或 Smalltalk，所有值都是对象，无论它们是否为复合类型。在不太纯粹的面向对象语言中，包括 C++ 和 Java，一些值，比如整数和浮点值，不是对象，而用户定义的复合类型是具有相关方法的真实对象。在 Julia 中，所有值都是对象，但函数不与它们操作的对象捆绑在一起。这是必要的，因为 Julia 通过多重派发选择函数使用的方法，这意味着在选择方法时考虑所有函数参数的类型，而不仅仅是第一个（有关方法和派发的更多信息，请参阅方法）。因此，函数仅仅「属于」它们的第一个参数是不合适的。将方法组织到函数对象中而不是在每个对象「内部」命名方法最终成为语言设计中一个非常有益的方面。

`struct` 关键字与复合类型一起引入，后跟一个字段名称的块，可选择使用 `::` 运算符注释类型：

```
julia> struct Foo
           bar
           baz::Int
           qux::Float64
       end
```

没有类型注释的字段默认为 `Any` 类型，所以可以包含任何类型的值。

类型为 `Foo` 的新对象通过将 `Foo` 类型对象像函数一样应用于其字段的值来创建：

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.", 23, 1.5)

julia> typeof(foo)
Foo
```

像函数一样使用的类型称为**构造函数**。有两个构造函数已被自动生成（这些构造函数称为**默认构造函数**）。其中一个接受任何参数并调用 `convert` 函数将它其转换为字段的类型，另一个接受与字段类型完全匹配的参数。两者都生成的原因是，这使得更容易添加新定义而不会在无意中替换默认构造函数。

由于 `bar` 字段在类型上不受限制，因此任何值都可以。但是 `baz` 的值必须可转换为 `Int` 类型：

```
julia> Foo((), 23.5, 1)
ERROR: InexactError: Int64(23.5)
Stacktrace:
[...]
```

可以使用 `fieldnames` 函数找到字段名称列表。

```
julia> fieldnames(Foo)
(:bar, :baz, :qux)
```

可以使用传统的 `foo.bar` 表示法访问复合对象的字段值：

```
julia> foo.bar
"Hello, world."
```

```
julia> foo.baz
23

julia> foo.qux
1.5
```

用 struct 声明的复合对象是不可变的；创建后不能修改。乍一看这似乎很奇怪，但它有几个优点：

- 它可以更高效。某些 struct 可以被高效地打包到数组中，并且在某些情况下，编译器可以避免完全分配不可变对象。
- 不可能违反类型构造函数提供的不变性。
- 使用不可变对象的代码更容易推理。

不可变对象可以包含可变对象（比如数组）作为字段。那些被包含的对象将保持可变；只是不可变对象本身的字段不能更改为指向不同的对象。

如果需要，可以使用关键字 mutable struct 声明可变复合对象，这将在下一节中讨论。

如果一个不可变结构的所有字段都是不可区分的（===），那么包含这些字段的两个不可变值也是不可区分的：

```
julia> struct X
           a::Int
           b::Float64
       end

julia> X(1, 2) === X(1, 2)
true
```

关于如何构造复合类型的实例还有很多要说的，但这种讨论依赖于参数类型和方法，并且这是非常重要的，应该在专门的章节中讨论：构造函数。

For many user-defined types X, you may want to define a method Base.broadcastable(x::X) = Ref(x) so that instances of that type act as 0-dimensional ”scalars” for broadcasting.

11.5 可变复合类型

如果使用 mutable struct 而不是 struct 声明复合类型，则它的实例可以被修改：

```
julia> mutable struct Bar
           baz
           qux::Float64
       end

julia> bar = Bar("Hello", 1.5);

julia> bar.qux = 2.0
2.0

julia> bar.baz = 1//2
1//2
```

An extra interface between the fields and the user can be provided through Instance Properties. This grants more control on what can be accessed and modified using the `bar.baz` notation.

为了支持修改，这种对象通常分配在堆上，并且具有稳定的内存地址。可变对象就像一个小容器，随着时间的推移，可能保持不同的值，因此只能通过其地址可靠地识别。相反地，不可变类型的实例与特定字段值相关——仅字段值就告诉你该对象的所有内容。在决定是否使类型为可变类型时，请询问具有相同字段值的两个实例是否被视为相同，或者它们是否可能需要随时间独立更改。如果它们被认为是相同的，该类型就应该是不可变的。

总结一下，Julia 的两个基本属性定义了不变性：

- 不允许修改不可变类型的值。
  - 对于位类型，这意味着值的位模式一旦设置将不再改变，并且该值是位类型的标识。
  - 对于复合类型，这意味着其字段值的标识将不再改变。当字段是位类型时，这意味着它们的位将不再改变，对于其值是可变类型（如数组）的字段，这意味着字段将始终引用相同的可变值，尽管该可变值的内容本身可能被修改。
- 具有不可变类型的对象可以被编译器自由复制，因为其不可变性使得不可能以编程方式区分原始对象和副本。
  - 特别地，这意味着足够小的不可变值（如整数和浮点数）通常在寄存器（或栈分配）中传递给函数。
  - 另一方面，可变值是堆分配的，并作为指向堆分配值的指针传递给函数，除非编译器确定没有办法知道这不是正在发生的事情。

In cases where one or more fields of an otherwise mutable struct is known to be immutable, one can declare these fields as such using `const` as shown below. This enables some, but not all of the optimizations of immutable structs, and can be used to enforce invariants on the particular fields marked as `const`.

Julia 1.8

`const` annotating fields of mutable structs requires at least Julia 1.8.

```
julia> mutable struct Baz
           a::Int
           const b::Float64
       end

julia> baz = Baz(1, 1.5);

julia> baz.a = 2
2

julia> baz.b = 2.0
ERROR: setfield!: const field .b of type Baz cannot be changed
[...]
```

11.6 已声明的类型

前面章节中讨论的三种类型（抽象、原始、复合）实际上都是密切相关的。它们共有相同的关键属性：

- 它们都是显式声明的。
- 它们都具有名称。
- 它们都已经显式声明超类型。
- 它们可以有参数。

由于这些共有属性，它们在内部表现为相同概念 DataType 的实例，其是任何这些类型的类型：

```
julia> typeof(Real)
DataType

julia> typeof(Int)
DataType
```

DataType 可以是抽象的或具体的。它如果是具体的，就具有指定的大小、存储布局和字段名称（可选）。因此，原始类型是具有非零大小的 DataType，但没有字段名称。复合类型是具有字段名称或者为空（大小为零）的 DataType。

每一个具体的值在系统里都是某个 DataType 的实例。

11.7 类型共用体

类型共用体是一种特殊的抽象类型，它包含作为对象的任何参数类型的所有实例，使用特殊Union关键字构造：

```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}

julia> 1 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"

julia> 1.0 :: IntOrString
ERROR: TypeError: in typeassert, expected Union{Int64, AbstractString}, got a value of type Float64
```

许多语言都有内建的共用体结构来推导类型；Julia 简单地将它暴露给程序员。Julia 编译器能在 Union 类型只具有少量类型[1]的情况下生成高效的代码，方法是为每个可能类型的不同分支都生成专用代码。

Union 类型的一种特别有用的情况是 Union{T, Nothing}，其中 T 可以是任何类型，Nothing 是单态类型，其唯一实例是对象 nothing。此模式是其它语言中 Nullable、Option 或 Maybe 类型在 Julia 的等价。通过将函数参数或字段声明为 Union{T, Nothing}，可以将其设置为类型为 T 的值，或者 nothing 来表示没有值。有关详细信息，请参阅常见问题的此条目。

11.8 参数类型

Julia 类型系统的一个重要和强大的特征是它是参数的：类型可以接受参数，因此类型声明实际上引入了一整套新类型——每一个参数值的可能组合引入一个新类型。许多语言支持某种版本的泛型编程，其中，可以指定操作泛型的数据结构和算法，而无需指定所涉及的确切类型。例如，某些形式的泛型编程存在于 ML、Haskell、Ada、Eiffel、C++、Java、C#、F#、和 Scala 中，这只是其中的一些例子。这些语言中的一些支持真正的参数多态（例如 ML、Haskell、Scala），而其它语言基于模板的泛型编程风格（例如 C++、Java）。由于在不同语言中有多种不同种类的泛型编程和参数类型，我们甚至不会尝试将 Julia 的参数类型与其它语言的进行比较，而是专注于解释 Julia 系统本身。然而，我们将注意到，因为 Julia 是动态类型语言并且不需要在编译时做出所有类型决定，所以许多在静态参数类型系统中遇到的传统困难可以被相对容易地处理。

所有已声明的类型（DataType 类型）都可被参数化，在每种情况下都使用一样的语法。我们将按一下顺序讨论它们：首先是参数复合类型，接着是参数抽象类型，最后是参数原始类型。

参数复合类型

类型参数在类型名称后引入，用大括号扩起来：

```
julia> struct Point{T}
           x::T
           y::T
       end
```

此声明定义了一个新的参数类型，Point{T}，拥有类型为 T 的两个「坐标」。有人可能会问 T 是什么？嗯，这恰恰是参数类型的重点：它可以是任何类型（或者任何位类型值，虽然它实际上在这里显然用作类型）。Point{Float64} 是一个具体类型，该类型等价于通过用 Float64 替换 Point 的定义中的 T 所定义的类型。因此，单独这一个声明实际上声明了无限个类型：Point{Float64}，Point{AbstractString}，Point{Int64}，等等。这些类型中的每一个类型现在都是可用的具体类型：

```
julia> Point{Float64}
Point{Float64}

julia> Point{AbstractString}
Point{AbstractString}
```

Point{Float64} 类型是坐标为 64 位浮点值的点，而 Point{AbstractString} 类型是「坐标」为字符串对象（请参阅 Strings）的「点」。

Point 本身也是一个有效的类型对象，包括所有实例 Point{Float64}、Point{AbstractString} 等作为子类型：

```
julia> Point{Float64} <: Point
true

julia> Point{AbstractString} <: Point
true
```

当然，其他类型不是它的子类型：

```
julia> Float64 <: Point
false

julia> AbstractString <: Point
false
```

Point 不同 T 值所声明的具体类型之间，不能互相作为子类型：

```
julia> Point{Float64} <: Point{Int64}
false

julia> Point{Float64} <: Point{Real}
false
```

> **Warning**
>
> 最后一点非常重要：即使 `Float64 <: Real` **也没有** `Point{Float64} <: Point{Real}`。

换成类型理论说法，Julia 的类型参数是不变的，而不是协变的（或甚至是逆变的）。这是出于实际原因：虽然任何 Point{Float64} 的实例在概念上也可能像是 Point{Real} 的实例，但这两种类型在内存中有不同的表示：

- Point{Float64} 的实例可以紧凑而高效地表示为一对 64 位立即数；
- Point{Real} 的实例必须能够保存任何一对 Real 的实例。由于 Real 实例的对象可以具有任意的大小和结构，Point{Real} 的实例实际上必须表示为一对指向单独分配的 Real 对象的指针。

在数组的情况下，能够以立即数存储 Point{Float64} 对象会极大地提高效率：Array{Float64} 可以存储为一段 64 位浮点值组成的连续内存块，而 Array{Real} 必须是一个由指向单独分配的 Real 的指针组成的数组——这可能是 boxed 64 位浮点值，但也可能是任意庞大和复杂的对象，且其被声明为 Real 抽象类型的表示。

由于 Point{Float64} 不是 Point{Real} 的子类型，下面的方法不适用于类型为 Point{Float64} 的参数：

```
function norm(p::Point{Real})
    sqrt(p.x^2 + p.y^2)
end
```

一种正确的方法来定义一个接受类型的所有参数的方法，Point{T} 其中 T 是一个子类型Real：

```
function norm(p::Point{<:Real})
    sqrt(p.x^2 + p.y^2)
end
```

（等效地，另一种定义方法 function norm(p::Point{T} where T<:Real) 或 function norm(p::Point{T}) where T<:Real；查看 UnionAll 类型。）

稍后将在方法中讨论更多示例。

如何构造一个 Point 对象？可以为复合类型定义自定义的构造函数，这将在构造函数中详细讨论，但在没有任何特别的构造函数声明的情况下，有两种默认方式可以创建新的复合对象，一种是显式地给出类型参数，另一种是通过传给对象构造函数的参数隐式地推断出。

由于 Point{Float64} 类型等价于在 Point 声明时用 Float64 替换 T 得到的具体类型，它可以相应地作为构造函数使用：

```
julia> p = Point{Float64}(1.0, 2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(p)
Point{Float64}
```

对于默认的构造函数，必须为每个字段提供一个参数：

```
julia> Point{Float64}(1.0)
ERROR: MethodError: no method matching Point{Float64}(::Float64)
[...]

julia> Point{Float64}(1.0,2.0,3.0)
ERROR: MethodError: no method matching Point{Float64}(::Float64, ::Float64, ::Float64)
[...]
```

参数类型只生成一个默认的构造函数，因为它无法覆盖。这个构造函数接受任何参数并将它们转换为字段的类型。

许多情况下，没有必要提供想要构造的 Point 对象的类型，因为构造函数调用参数的类型已经隐式地提供了类型信息。因此，你也可以将 Point 本身用作构造函数，前提是参数类型 T 的隐含值是明确的：

```
julia> p1 = Point(1.0,2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(p1)
Point{Float64}

julia> p2 = Point(1,2)
Point{Int64}(1, 2)

julia> typeof(p2)
Point{Int64}
```

在 Point 的例子中，当且仅当 Point 的两个参数类型相同时，T 的类型才确实是隐含的。如果不是这种情况，构造函数将失败并出现 MethodError：

```
julia> Point(1,2.5)
ERROR: MethodError: no method matching Point(::Int64, ::Float64)

Closest candidates are:
  Point(::T, !Matched::T) where T
   @ Main none:2
```

```
Stacktrace:
[...]
```

可以定义适当处理此类混合情况的函数构造方法，将在后面的构造函数中讨论。

参数抽象类型

参数抽象类型声明以非常相似的方式声明了一族抽象类型：

```
julia> abstract type Pointy{T} end
```

在此声明中，对于每个类型或整数值 T，Pointy{T} 都是不同的抽象类型。与参数复合类型一样，每个此类型的实例都是 Pointy 的子类型：

```
julia> Pointy{Int64} <: Pointy
true

julia> Pointy{1} <: Pointy
true
```

参数抽象类型是不变的，就像参数复合类型：

```
julia> Pointy{Float64} <: Pointy{Real}
false

julia> Pointy{Real} <: Pointy{Float64}
false
```

符号 Pointy{<:Real} 可用于表示协变类型的 Julia 类似物，而 Pointy{>:Int} 类似于逆变类型，但从技术上讲，它们都代表了类型的集合（参见 UnionAll 类型）。

```
julia> Pointy{Float64} <: Pointy{<:Real}
true

julia> Pointy{Real} <: Pointy{>:Int}
true
```

正如之前的普通抽象类型用于在具体类型上创建实用的类型层次结构一样，参数抽象类型在参数复合类型上具有相同的用途。例如，我们可以将 Point{T} 声明为 Pointy{T} 的子类型，如下所示：

```
julia> struct Point{T} <: Pointy{T}
           x::T
           y::T
       end
```

鉴于此类声明，对每个 T，都有 Point{T} 是 Pointy{T} 的子类型：

```
julia> Point{Float64} <: Pointy{Float64}
true

julia> Point{Real} <: Pointy{Real}
true

julia> Point{AbstractString} <: Pointy{AbstractString}
true
```

下面的关系依然不变：

```
julia> Point{Float64} <: Pointy{Real}
false

julia> Point{Float64} <: Pointy{<:Real}
true
```

参数抽象类型（比如 Pointy）的用途是什么？考虑一下如果点都在对角线 $x = y$ 上，那我们创建的点的实现可以只有一个坐标：

```
julia> struct DiagPoint{T} <: Pointy{T}
           x::T
       end
```

现在，Point{Float64} 和 DiagPoint{Float64} 都是抽象 Pointy{Float64} 的实现，每个类型 T 的其它可能选择与之类似。这允许对被所有 Pointy 对象共享的公共接口进行编程，接口都由 Point 和 DiagPoint 实现。但是，直到我们在下一节方法中引入方法和分派前，这无法完全证明。

有时，类型参数取遍所有可能类型也许是无意义的。在这种情况下，可以像这样约束 T 的范围：

```
julia> abstract type Pointy{T<:Real} end
```

在这样的声明中，可以使用任何 Real 的子类型替换 T，但不能使用不是 Real 子类型的类型：

```
julia> Pointy{Float64}
Pointy{Float64}

julia> Pointy{Real}
Pointy{Real}

julia> Pointy{AbstractString}
ERROR: TypeError: in Pointy, in T, expected T<:Real, got Type{AbstractString}

julia> Pointy{1}
ERROR: TypeError: in Pointy, in T, expected T<:Real, got a value of type Int64
```

参数化复合类型的类型参数可用相同的方式限制：

```
struct Point{T<:Real} <: Pointy{T}
    x::T
    y::T
end
```

这里给出了一个真实示例，展示了所有这些参数类型机制如何发挥作用，下面是 Julia 的不可变类型 `Rational` 的实际定义（除了我们为了简单起见省略了的构造函数），用来表示准确的整数比例：

```
struct Rational{T<:Integer} <: Real
    num::T
    den::T
end
```

只有接受整数值的比例才是有意义的，因此参数类型 `T` 被限制为 `Integer` 的子类型，又整数的比例代表实数轴上的值，因此任何 `Rational` 都是抽象 `Real` 的实现。

元组类型

元组类型是函数参数的抽象化——不带函数本身。函数参数的突出特征是它们的顺序和类型。因此，元组类型类似于参数化的不可变类型，其中每个参数都是一个字段的类型。例如，二元元组类型类似于以下不可变类型：

```
struct Tuple2{A,B}
    a::A
    b::B
end
```

然而，有三个主要差异：

- 元组类型可以具有任意数量的参数。
- 元组类型的参数是**协变的**（covariant）：`Tuple{Int}` 是 `Tuple{Any}` 的子类型。因此，`Tuple{Any}` 被认为是一种抽象类型，且元组类型只有在它们的参数都是具体类型时才是具体类型。
- 元组没有字段名称; 字段只能通过索引访问。

元组值用括号和逗号书写。构造元组时，会根据需要生成适当的元组类型：

```
julia> typeof((1,"foo",2.5))
Tuple{Int64, String, Float64}
```

请注意协变性的含义：

```
julia> Tuple{Int,AbstractString} <: Tuple{Real,Any}
true

julia> Tuple{Int,AbstractString} <: Tuple{Real,Real}
false

julia> Tuple{Int,AbstractString} <: Tuple{Real,}
false
```

直观地，这对应于函数参数的类型是函数签名（当函数签名匹配时）的子类型。

变参元组类型

元组类型的最后一个参数可以是特殊值 Vararg，它表示任意数量的尾随参数：

```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString, Vararg{Int64}}

julia> isa(("1",), mytupletype)
true

julia> isa(("1",1), mytupletype)
true

julia> isa(("1",1,2), mytupletype)
true

julia> isa(("1",1,2,3.0), mytupletype)
false
```

此外，Vararg{T} 对应于零个或更多的类型为 T 的元素。变参元组类型被用来表示变参方法接受的参数（请参阅变参函数）。

特殊值 Vararg{T,N}（当用作元组类型的最后一个参数时）正好对应于类型为 T 的 N 个元素。NTuple{N,T} 是 Tuple{Vararg{T,N}} 的一个方便的别名，即一个包含正好包含 T 类型的 N 个元素的元组类型。

具名元组类型

具名元组是 NamedTuple 类型的实例，该类型有两个参数：一个给出字段名称的符号元组，和一个给出字段类型的元组类型。For convenience, NamedTuple types are printed using the @NamedTuple macro which provides a convenient struct-like syntax for declaring these types via key::Type declarations, where an omitted ::Type corresponds to ::Any.

```
julia> typeof((a=1,b="hello")) # prints in macro form
@NamedTuple{a::Int64, b::String}

julia> NamedTuple{(:a, :b), Tuple{Int64, String}} # long form of the type
@NamedTuple{a::Int64, b::String}
```

The begin ... end form of the @NamedTuple macro allows the declarations to be split across multiple lines (similar to a struct declaration), but is otherwise equivalent:

```
julia> @NamedTuple begin
           a::Int
           b::String
       end
@NamedTuple{a::Int64, b::String}
```

NamedTuple 类型可以用作构造函数，接受一个单独的元组作为参数。构造出来的 NamedTuple 类型可以是具体类型，如果参数都被指定，也可以是只由字段名称所指定的类型：

```
julia> @NamedTuple{a::Float32,b::String}((1, ""))
(a = 1.0f0, b = "")

julia> NamedTuple{(:a, :b)}((1, ""))
(a = 1, b = "")
```

如果指定了字段类型，参数会被转换。否则，就直接使用参数的类型。

参数原始类型

原始类型也可以参数化声明，例如，指针都能表示为原始类型，其在 Julia 中以如下方式声明：

```
# 32-bit system:
primitive type Ptr{T} 32 end

# 64-bit system:
primitive type Ptr{T} 64 end
```

与典型的参数复合类型相比，此声明中略显奇怪的特点是类型参数 T 并未在类型本身的定义里使用——它实际上只是一个抽象的标记，定义了一整族具有相同结构的类型，类型间仅由它们的类型参数来区分。因此，Ptr{Float64} 和 Ptr{Int64} 是不同的类型，就算它们具有相同的表示。当然，所有特定的指针类型都是总类型 Ptr 的子类型：

```
julia> Ptr{Float64} <: Ptr
true

julia> Ptr{Int64} <: Ptr
true
```

11.9 UnionAll 类型

我们已经说过，像 Ptr 这样的参数类型可充当它所有实例（Ptr{Int64} 等）的超类型。这是如何办到的？Ptr 本身不能是普通的数据类型，因为在不知道引用数据的类型时，该类型显然不能用于存储器操作。答案是 Ptr（或其它参数类型像 Array）是一种不同种类的类型，称为 UnionAll 类型。这种类型表示某些参数的所有值的类型的迭代并集。

UnionAll 类型通常使用关键字 where 编写。例如，Ptr 可以更精确地写为 Ptr{T} where T，也就是对于 T 的某些值，所有类型为 Ptr{T} 的值。在这种情况下，参数 T 也常被称为「类型变量」，因为它就像一个取值范围为类型的变量。每个 where 只引入一个类型变量，因此在具有多个参数的类型中这些表达式会被嵌套，例如 Array{T,N} where N where T。

类型应用语法 A{B,C} 要求 A 是个 UnionAll 类型，并先代入 B 作为 A 中最外层的类型变量。结果应该是另一个 UnionAll 类型，然后再将 C 代入。所以 A{B,C} 等价于 A{B}{C}。这解释了为什么可以部分实例化一个类型，比如 Array{Float64}：第一个参数已经被固定，但第二个参数仍取遍所有可能值。通过使用 where 语法，任何参数子集都能被固定。例如，所有一维数组的类型可以写为 Array{T,1} where T。

类型变量可以用子类型关系来加以限制。Array{T} where T<:Integer 指的是元素类型是某种 Integer 的所有数组。语法 Array{<:Integer} 是 Array{T} where T<:Integer 的便捷的缩写。类型变量可同时具有上下界。Array{T} where Int<:T<:Number 指的是元素类型为能够包含 Int 的 Number 的所有数组

(因为 T 至少和 Int 一样大)。语法 where T>:Int 也能用来只指定类型变量的下界，且 Array{>:Int} 等价于 Array{T} where T>:Int。

由于 where 表达式可以嵌套，类型变量界可以引用更外层的类型变量。比如 Tuple{T,Array{S}} where S<:AbstractArray{T} where T<:Real 指的是二元元组，其第一个元素是某个 Real，而第二个元素是数组的数组 Array，其包含的内部数组的元素类型由元组的第一个元素类型决定。

where 关键字本身可以嵌套在更复杂的声明里。例如，考虑由以下声明创建的两个类型：

```
julia> const T1 = Array{Array{T, 1} where T, 1}
Vector{Vector} (alias for Array{Array{T, 1} where T, 1})

julia> const T2 = Array{Array{T, 1}, 1} where T
Array{Vector{T}, 1} where T
```

类型 T1 定义了由一维数组组成的一维数组；每个内部数组由相同类型的对象组成，但此类型对于不同内部数组可以不同。另一方面，类型 T2 定义了由一维数组组成的一维数组，其中的每个内部数组必须具有相同的类型。请注意，T2 是个抽象类型，比如 Array{Array{Int,1},1} <: T2，而 T1 是个具体类型。因此，T1 可由零参数构造函数 a=T1() 构造，但 T2 不行。

命名此类型有一种方便的语法，类似于函数定义语法的简短形式：

```
Vector{T} = Array{T, 1}
```

这等价于 const Vector = Array{T,1} where T。编写 Vector{Float64} 等价于编写 Array{Float64,1}，总类型 Vector 具有所有 Array 对象的实例，其中 Array 对象的第二个参数——数组维数——是 1，而不考虑元素类型是什么。在参数类型必须总被完整指定的语言中，这不是特别有用，但在 Julia 中，这允许只编写 Vector 来表示包含任何元素类型的所有一维密集数组的抽象类型。

11.10 单例类型

没有字段的不可变复合类型称为 单例类型。正式地，如果

1. T 是一个不可变的复合类型（即用 struct 定义），
2. a isa T && b isa T 暗含 a === b，

那么 T 是单例类型。[2] Base.issingletontype 可以用来检查一个类型是否是单例类型。抽象类型 不能通过构造成为单例类型。

根据定义，此类类型只能有一个实例：

```
julia> struct NoFields
       end

julia> NoFields() === NoFields()
true

julia> Base.issingletontype(NoFields)
true
```

=== 函数确认 NoFields 的构造实例实际上是一个且相同的。

当上述条件成立时，参数类型可以是单例类型。例如，

```
julia> struct NoFieldsParam{T}
       end

julia> Base.issingletontype(NoFieldsParam) # Can't be a singleton type ...
false

julia> NoFieldsParam{Int}() isa NoFieldsParam # ... because it has ...
true

julia> NoFieldsParam{Bool}() isa NoFieldsParam # ... multiple instances.
true

julia> Base.issingletontype(NoFieldsParam{Int}) # Parametrized, it is a singleton.
true

julia> NoFieldsParam{Int}() === NoFieldsParam{Int}()
true
```

11.11 Types of functions

Each function has its own type, which is a subtype of `Function`.

```
julia> foo41(x) = x + 1
foo41 (generic function with 1 method)

julia> typeof(foo41)
typeof(foo41) (singleton type of function foo41, subtype of Function)
```

Note how `typeof(foo41)` prints as itself. This is merely a convention for printing, as it is a first-class object that can be used like any other value:

```
julia> T = typeof(foo41)
typeof(foo41) (singleton type of function foo41, subtype of Function)

julia> T <: Function
true
```

Types of functions defined at top-level are singletons. When necessary, you can compare them with ===.

Closures also have their own type, which is usually printed with names that end in `#<number>`. Names and types for functions defined at different locations are distinct, but not guaranteed to be printed the same way across sessions.

```
julia> typeof(x -> x + 1)
var"#9#10"
```

Types of closures are not necessarily singletons.

```
julia> addy(y) = x -> x + y
addy (generic function with 1 method)

julia> typeof(addy(1)) === typeof(addy(2))
true

julia> addy(1) === addy(2)
false

julia> Base.issingletontype(typeof(addy(1)))
false
```

11.12 Type{T} 类型选择器

对于每个类型 T，Type{T} 是一个抽象的参数类型，它的唯一实例是对象 T。在我们讨论 参数方法 和 类型转换 之前，很难解释这个构造的效用，但简而言之，它允许人们专门针对特定类型的函数行为作为值。这对于编写其行为取决于作为显式参数给出的类型而不是由其参数之一的类型隐含的类型的方法（尤其是参数方法）很有用。

由于定义有点难理解，我们来看一些例子：

```
julia> isa(Float64, Type{Float64})
true

julia> isa(Real, Type{Float64})
false

julia> isa(Real, Type{Real})
true

julia> isa(Float64, Type{Real})
false
```

换句话说，isa(A, Type{B}) 当且仅当 A 和 B 是同一个对象并且该对象是一个类型时才为真。

特别的，由于参数类型是 不变量，我们有

```
julia> struct TypeParamExample{T}
           x::T
       end

julia> TypeParamExample isa Type{TypeParamExample}
true

julia> TypeParamExample{Int} isa Type{TypeParamExample}
false

julia> TypeParamExample{Int} isa Type{TypeParamExample{Int}}
true
```

如果没有参数，Type 只是一个抽象类型，所有类型对象都是其实例：

```
julia> isa(Type{Float64}, Type)
true

julia> isa(Float64, Type)
true

julia> isa(Real, Type)
true
```

不是类型的对象不是 Type 的实例：

```
julia> isa(1, Type)
false

julia> isa("foo", Type)
false
```

虽然 Type 与任何其他抽象参数类型一样是 Julia 类型层次结构的一部分，但它并不常用在方法签名之外，除非在某些特殊情况下。Type 的另一个重要用法是使不太精确的字段类型更加清晰，例如DataType 在下面的示例中，默认构造函数可能会导致依赖精确包装类型的代码出现性能问题（类似于 [抽象类型参数](@ref man-performance-abstract-container)）。

```
julia> struct WrapType{T}
       value::T
       end

julia> WrapType(Float64) # default constructor, note DataType
WrapType{DataType}(Float64)

julia> WrapType(::Type{T}) where T = WrapType{Type{T}}(T)
WrapType

julia> WrapType(Float64) # sharpened constructor, note more precise Type{Float64}
WrapType{Type{Float64}}(Float64)
```

11.13 类型别名

有时为一个已经可表达的类型引入新名称是很方便的。这可通过一个简单的赋值语句完成。例如，UInt 是 UInt32 或 UInt64 的别名，因为它的大小与系统上的指针大小是相适应的。

```
# 32-bit system:
julia> UInt
UInt32

# 64-bit system:
julia> UInt
UInt64
```

这是由 base/boot.jl 中以下代码实现的：

```
if Int === Int64
    const UInt = UInt64
else
    const UInt = UInt32
end
```

当然，这依赖于 Int 的别名，但它被预定义成正确的类型——Int32 或 Int64。

（注意，与 Int 不同，Float 不作为特定大小的 AbstractFloat 类型的别名而存在。与整数寄存器不同，浮点数寄存器大小由 IEEE-754 标准指定，而 Int 的大小反映了该机器上本地指针的大小。）

11.14 类型操作

因为 Julia 中的类型本身就是对象，所以一般的函数可以对它们进行操作。已经引入了一些对于使用或探索类型特别有用的函数，例如 <: 运算符，它表示其左操作数是否为其右操作数的子类型。

isa 函数测试对象是否具有给定类型并返回 true 或 false：

```
julia> isa(1, Int)
true

julia> isa(1, AbstractFloat)
false
```

在文档示例中随处可见的 typeof 函数返回其参数的类型。如上所述，因为类型都是对象，所以它们也有类型，我们可以询问它们的类型：

```
julia> typeof(Rational{Int})
DataType

julia> typeof(Union{Real,String})
Union
```

如果我们重复这个过程会怎样？一个类型的类型是什么？碰巧，每个类型都是复合值，因此都具有 DataType 类型：

```
julia> typeof(DataType)
DataType

julia> typeof(Union)
DataType
```

DataType 是它自己的类型。

另一个适用于某些类型的操作是 supertype，它显示了类型的超类型。只有已声明的类型（DataType）才有明确的超类型：

```
julia> supertype(Float64)
AbstractFloat
```

```
julia> supertype(Number)
Any

julia> supertype(AbstractString)
Any

julia> supertype(Any)
Any
```

如果将 supertype 应用于其它类型对象（或非类型对象），则会引发 MethodError：

```
julia> supertype(Union{Float64,Int64})
ERROR: MethodError: no method matching supertype(::Type{Union{Float64, Int64}})
Closest candidates are:
[...]
```

11.15 自定义 pretty-printing

通常，人们会想要自定义显示类型实例的方式。这可通过重载 show 函数来完成。举个例子，假设我们定义一个类型来表示极坐标形式的复数：

```
julia> struct Polar{T<:Real} <: Number
           r::T
           Θ::T
       end

julia> Polar(r::Real,Θ::Real) = Polar(promote(r,Θ)...)
Polar
```

在这里，我们添加了一个自定义的构造函数，这样就可以接受不同 Real 类型的参数并将它们类型提升为共同类型（请参阅构造函数和类型转换和类型提升）。（当然，为了让它表现地像个 Number，我们需要定义许多其它方法，例如 +、*、one、zero 及类型提升规则等。）默认情况下，此类型的实例只是相当简单地显示有关类型名称和字段值的信息，比如，Polar{Float64}(3.0,4.0)。

如果我们希望它显示为 3.0 * exp(4.0im)，我们可定义以下方法来将对象打印到给定的输出对象 io（其代表文件、终端、及缓冲区等；请参阅网络和流）：

```
julia> Base.show(io::IO, z::Polar) = print(io, z.r, " * exp(", z.Θ, "im)")
```

Polar 对象的输出可以被更精细地控制。特别是，人们有时想要🄵嗦的多行打印格式，用于在 REPL 和其它交互式环境中显示单个对象，以及一个更紧凑的单行格式，用于 print 函数或在作为其它对象（比如一个数组）的部分是显示该对象。虽然在两种情况下默认都会调用 show(io, z) 函数，你仍可以定义一个不同的多行格式来显示单个对象，这通过重载三参数形式的 show 函数，该函数接收 text/plain MIME 类型（请参阅 多媒体 I/O）作为它的第二个参数，举个例子：

```
julia> Base.show(io::IO, ::MIME"text/plain", z::Polar{T}) where{T} =
           print(io, "Polar{$T} complex number:\n   ", z)
```

（请注意 print(..., z) 在这里调用的是双参数的 show(io, z) 方法。）这导致：

```
julia> Polar(3, 4.0)
Polar{Float64} complex number:
   3.0 * exp(4.0im)

julia> [Polar(3, 4.0), Polar(4.0,5.3)]
2-element Vector{Polar{Float64}}:
 3.0 * exp(4.0im)
 4.0 * exp(5.3im)
```

其中单行格式的 show(io, z) 仍用于由 Polar 值组成的数组。从技术上讲，REPL 调用 display(z) 来显示单行的执行结果，其默认为 show(stdout, MIME("text/plain"), z)，而后者又默认为 show(stdout, z)，但是你不应该定义新的 display 方法，除非你正在定义新的多媒体显示管理器（请参阅多媒体 I/O）。

此外，你还可以为其它 MIME 类型定义 show 方法，以便在支持的环境（比如 IJulia）中实现更丰富的对象显示（HTML、图像等）。例如，我们可以定义 Polar 对象的 HTML 显示格式，使其带有上标和斜体：

```
julia> Base.show(io::IO, ::MIME"text/html", z::Polar{T}) where {T} =
           println(io, "<code>Polar{$T}</code> complex number: ",
                   z.r, " <i>e</i><sup>", z.Θ, " <i>i</i></sup>")
```

之后会在支持 HTML 显示的环境中自动使用 HTML 显示 Polar 对象，但你也可以手动调用 show 来获取 HTML 输出：

```
julia> show(stdout, "text/html", Polar(3.0,4.0))
<code>Polar{Float64}</code> complex number: 3.0 <i>e</i><sup>4.0 <i>i</i></sup>
```

根据经验，单行 show 方法应为创建的显示对象打印有效的 Julia 表达式。当这个 show 方法包含中缀运算符时，比如上面的 Polar 的单行 show 方法里的乘法运算符（*），在作为另一个对象的部分打印时，它可能无法被正确解析。要查看此问题，请考虑下面的表达式对象（请参阅程序表示），它代表 Polar 类型的特定实例的平方：

```
julia> a = Polar(3, 4.0)
Polar{Float64} complex number:
   3.0 * exp(4.0im)

julia> print(:($a^2))
3.0 * exp(4.0im) ^ 2
```

因为运算符 ^ 的优先级高于 *（请参阅运算符的优先级与结合性），所以此输出错误地表示了表达式 a ^ 2，而该表达式等价于 (3.0 * exp(4.0im)) ^ 2。为了解决这个问题，我们必须为 Base.show_unquoted(io::IO, z::Polar, indent::Int, precedence::Int) 创建一个自定义方法，在打印时，表达式对象会在内部调用它：

```
julia> function Base.show_unquoted(io::IO, z::Polar, ::Int, precedence::Int)
           if Base.operator_precedence(:*) <= precedence
               print(io, "(")
               show(io, z)
               print(io, ")")
           else
```

```
            show(io, z)
        end
    end

julia> :($a^2)
:((3.0 * exp(4.0im)) ^ 2)
```

当正在调用的运算符的优先级大于等于乘法的优先级时，上面定义的方法会在 `show` 调用的两侧加上括号。这个检查允许，在没有括号时也可被正确解析的表达式（例如 `:($a + 2)` 和 `:($a == 2)`），在打印时省略括号：

```
julia> :($a + 2)
:(3.0 * exp(4.0im) + 2)

julia> :($a == 2)
:(3.0 * exp(4.0im) == 2)
```

在某些情况下，根据上下文调整 `show` 方法的行为是很有用的。这可通过 `IOContext` 类型实现，它允许同时传递上下文属性和封装后的 IO 流。例如，我们可以在 `:compact` 属性设置为 `true` 时创建一个更短的表示，而在该属性为 `false` 或不存在时返回长的表示：

```
julia> function Base.show(io::IO, z::Polar)
           if get(io, :compact, false)::Bool
               print(io, z.r, "∠", z.θ, "im")
           else
               print(io, z.r, " * exp(", z.θ, "im)")
           end
       end
```

当传入的 IO 流是设置了 `:compact`（译注：该属性还应当设置为 `true`）属性的 `IOContext` 对象时，新的紧凑表示将被使用。特别地，当打印具有多列的数组（由于水平空间有限）时就是这种情况：

```
julia> show(IOContext(stdout, :compact=>true), Polar(3, 4.0))
3.0∠4.0im

julia> [Polar(3, 4.0) Polar(4.0,5.3)]
1×2 Matrix{Polar{Float64}}:
 3.0∠4.0im  4.0∠5.3im
```

有关调整打印效果的常用属性列表，请参阅文档 `IOContext`。

11.16 值类型

在 Julia 中，你无法根据诸如 `true` 或 `false` 之类的值进行分派。然而，你可以根据参数类型进行分派，Julia 允许你包含「plain bits」值（类型、符号、整数、浮点数和元组等）作为类型参数。`Array{T,N}` 里的维度参数就是一个常见的例子，在这里 `T` 是类型（比如 `Float64`），而 `N` 只是个 `Int`。

你可以创建把值作为参数的自定义类型，并使用它们控制自定义类型的分派。为了说明这个想法，让我们引入参数类型 `Val{x}` 和构造函数 `Val(x) = Val{x}()`，在不需要更精细的层次结构时，这是利用此技巧的一种习惯的方式。

Val 的定义为：

```
julia> struct Val{x}
       end

julia> Val(x) = Val{x}()
Val
```

Val 的实现就只需要这些。一些 Julia 标准库里的函数接收 Val 的实例作为参数，你也可以使用它来编写你自己的函数，例如：

```
julia> firstlast(::Val{true}) = "First"
firstlast (generic function with 1 method)

julia> firstlast(::Val{false}) = "Last"
firstlast (generic function with 2 methods)

julia> firstlast(Val(true))
"First"

julia> firstlast(Val(false))
"Last"
```

为了保证 Julia 的一致性，调用处应当始终传递 Val **实例**而不是 **类型**，也就是使用 foo(Val(:bar)) 而不是 foo(Val{:bar})。

值得注意的是，参数「值」类型非常容易被误用，包括 Val；在不适用的情形下，你很容易使代码性能变得更糟糕。特别是，你可能永远都不会想要写出如上所示的代码。有关 Val 的正确（和不正确）使用的更多信息，请阅读性能建议中更广泛的讨论。

[1]「少数」由常数 max_union_splitting 定义，目前默认为 4。

[2]一些流行的编程语言具有单例类型，包括 Haskell、Scala 和 Ruby。

Chapter 12

方法

我们回想一下，在函数中我们知道函数是这么一个对象，它把一组参数映射成一个返回值，或者当没有办法返回恰当的值时扔出一个异常。具有相同概念的函数或者运算，经常会根据参数类型的不同而进行有很大差异的实现：两个整数的加法与两个浮点数的加法是相当不一样的，整数与浮点数之间的加法也不一样。除了它们实现上的不同，这些运算都归在" 加法" 这么一个广义的概念之下，因此在 Julia 中这些行为都属于同一个对象：+ 函数。

为了让对同样的概念使用许多不同的实现这件事更顺畅，函数没有必要马上全部都被定义，反而应该是一块一块地定义，为特定的参数类型和数量的组合提供指定的行为。对于一个函数的一个可能行为的定义叫做方法。直到这里，我们只展示了那些只定了一个方法的，对参数的所有类型都适用的函数。但是方法定义的特征是不仅能表明参数的数量，也能表明参数的类型，并且能提供多个方法定义。当一个函数被应用于特殊的一组参数时，能用于这一组参数的最特定的方法会被使用。所以，函数的全体行为是他的不同的方法定义的行为的组合。如果这个组合被设计得好，即使方法们的实现之间会很不一样，函数的外部行为也会显得无缝而自洽。

当一个函数被应用时执行方法的选择被称为 **分派**。Julia 允许分派过程基于给定的参数个数和所有参数的类型来选择调用函数的哪个方法。这与传统的面对对象的语言不一样，面对对象语言的分派只基于第一参数，经常有特殊的参数语法，并且有时是暗含而非显式写成一个参数。[1] 使用函数的所有参数，而非只用第一个，来决定调用哪个方法被称为多重分派。多重分派对于数学代码来说特别有用，人工地将运算视为对于其中一个参数的属于程度比其他所有的参数都强的这个概念对于数学代码是几乎没有意义的：`x + y` 中的加法运算对 `x` 的属于程度比对 `y` 更强？一个数学运算符的实现普遍基于它所有的参数的类型。即使跳出数学运算，多重分派是对于结构和组织程序来说也是一个强大而方便的范式。

> **Note**
>
> 本章中的所有示例都假定是为 相同模块中的函数定义模块。如果你想给 另一个模块中的函数添加方法，你必须 `import` 它或使用模块名称限定的名称。请参阅有关 命名空间管理 的部分。

12.1 定义方法

直到这里，在我们的例子中，我们定义的函数只有一个不限制参数类型的方法。这种函数的行为就与传统动态类型语言中的函数一样。不过，我们已经在没有意识到的情况下已经使用了多重分派和

[1] In C++ or Java, for example, in a method call like `obj.meth(arg1,arg2)`, the object obj "receives" the method call and is implicitly passed to the method via the `this` keyword, rather than as an explicit method argument. When the current `this` object is the receiver of a method call, it can be omitted altogether, writing just `meth(arg1,arg2)`, with `this` implied as the receiving object.

方法：所有 Julia 标准函数和运算符，就像之前提到的 + 函数，都根据参数的类型和数量的不同组合而定义了大量方法。

当定义一个函数时，可以根据需要使用在复合类型中介绍的 :: 类型断言运算符来限制参数类型，

```
julia> f(x::Float64, y::Float64) = 2x + y
f (generic function with 1 method)
```

这个函数只在 x 和 y 的类型都是 Float64 的情况下才会被调用：

```
julia> f(2.0, 3.0)
7.0
```

用其它任意的参数类型则会导致 MethodError：

```
julia> f(2.0, 3)
ERROR: MethodError: no method matching f(::Float64, ::Int64)

Closest candidates are:
  f(::Float64, !Matched::Float64)
   @ Main none:1

Stacktrace:
[...]

julia> f(Float32(2.0), 3.0)
ERROR: MethodError: no method matching f(::Float32, ::Float64)

Closest candidates are:
  f(!Matched::Float64, ::Float64)
   @ Main none:1

Stacktrace:
[...]

julia> f(2.0, "3.0")
ERROR: MethodError: no method matching f(::Float64, ::String)

Closest candidates are:
  f(::Float64, !Matched::Float64)
   @ Main none:1

Stacktrace:
[...]

julia> f("2.0", "3.0")
ERROR: MethodError: no method matching f(::String, ::String)
```

如同你所看到的，参数必须精确地是 Float64 类型。其它数字类型，比如整数或者 32 位浮点数值，都不会自动转化成 64 位浮点数，字符串也不会解析成数字。由于 Float64 是一个具体类型，且在 Julia 中具体类型无法拥有子类，所以这种定义方式只能适用于函数的输入类型精确地是 Float64 的情况，但一个常见的做法是用抽象类型来定义通用的方法：

```
julia> f(x::Number, y::Number) = 2x - y
f (generic function with 2 methods)

julia> f(2.0, 3)
1.0
```

用上面这种方式定义的方法可以接收任意一对 Number 的实例参数，且它们不需要是同一类型的，只要求都是数值。如何根据不同的类型来做相应的处理就可以委托给表达式 2x - y 中的代数运算。

为了定义一个有多个方法的函数，只需简单定义这个函数多次，使用不同的参数数量和类型。函数的第一个方法定义会建立这个函数对象，后续的方法定义会添加新的方法到存在的函数对象中去。当函数被应用时，最符合参数的数量和类型的特定方法会被执行。所以，上面的两个方法定义在一起定义了函数 f 对于所有的一对虚拟类型 Number 实例的行为–但是针对一对Float64值有不同的行为。如果一个参数是 64 位浮点数而另一个不是，f(Float64,Float64) 方法不会被调用，而一定使用更加通用的 f(Number,Number) 方法：

```
julia> f(2.0, 3.0)
7.0

julia> f(2, 3.0)
1.0

julia> f(2.0, 3)
1.0

julia> f(2, 3)
1
```

2x + y 定义只用于第一个情况，2x - y 定义用于其他的情况。没有使用任何自动的函数参数的指派或者类型转换：Julia 中的所有转换都不是 magic 的，都是完全显式的。然而类型转换和类型提升显示了足够先进的技术的智能应用能够与 magic 不可分辨到什么程度。[2] 对于非数字值，和比两个参数更多或者更少的情况，函数 f 并没有定义，应用会导致 MethodError：

```
julia> f("foo", 3)
ERROR: MethodError: no method matching f(::String, ::Int64)

Closest candidates are:
  f(!Matched::Number, ::Number)
   @ Main none:1

Stacktrace:
[...]

julia> f()
ERROR: MethodError: no method matching f()

Closest candidates are:
  f(!Matched::Float64, !Matched::Float64)
   @ Main none:1
  f(!Matched::Number, !Matched::Number)
   @ Main none:1
```

```
Stacktrace:
[...]
```

可以简单地看到对于函数存在哪些方法，通过在交互式会话中键入函数对象本身：

```
julia> f
f (generic function with 2 methods)
```

这个输出展示了 f 有两个方法。为了找到这些方法的前面，使用methods函数：

```
julia> methods(f)
# 2 methods for generic function "f" from Main:
 [1] f(x::Float64, y::Float64)
     @ none:1
 [2] f(x::Number, y::Number)
     @ none:1
```

这表示 f 有两个方法，一个接受两个 Float64 参数一个接受两个 Number 类型的参数。它也显示了这些方法定义所在的文件和行数：因为这些方法是在 REPL 中定义的，我们得到了表面上的行数 none:1.

没有:: 的类型声明，方法参数的类型默认为 Any，这就意味着没有约束，因为 Julia 中的所有的值都是抽象类型 Any 的实例。所以，我们可以为 f 定义一个接受所有的方法，像这样：

```
julia> f(x,y) = println("Whoa there, Nelly.")
f (generic function with 3 methods)

julia> methods(f)
# 3 methods for generic function "f" from Main:
 [1] f(x::Float64, y::Float64)
     @ none:1
 [2] f(x::Number, y::Number)
     @ none:1
 [3] f(x, y)
     @ none:1

julia> f("foo", 1)
Whoa there, Nelly.
```

这个接受所有参数类型的方法比其他的对一对参数值的其他任意可能的方法定义更不专用。所以他只会被没有其他方法定义应用的一对参数调用。

注意到第三个方法的签名中并没有指定参数 x 和 y 的类型。它是 f(x::Any, y::Any) 的简写。

尽管这看起来很简单，但对值类型的多重派发可能是 Julia 语言最强大和最核心的特性。核心运算通常有几十种方法：

```
julia> methods(+)
# 180 methods for generic function "+":
[1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:227
[2] +(x::Bool, y::Bool) in Base at bool.jl:89
```

```
[3] +(x::Bool) in Base at bool.jl:86
[4] +(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:96
[5] +(x::Bool, z::Complex) in Base at complex.jl:234
[6] +(a::Float16, b::Float16) in Base at float.jl:373
[7] +(x::Float32, y::Float32) in Base at float.jl:375
[8] +(x::Float64, y::Float64) in Base at float.jl:376
[9] +(z::Complex{Bool}, x::Bool) in Base at complex.jl:228
[10] +(z::Complex{Bool}, x::Real) in Base at complex.jl:242
[11] +(x::Char, y::Integer) in Base at char.jl:40
[12] +(c::BigInt, x::BigFloat) in Base.MPFR at mpfr.jl:307
[13] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) in Base.GMP at gmp.jl:392
[14] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt) in Base.GMP at gmp.jl:391
[15] +(a::BigInt, b::BigInt, c::BigInt) in Base.GMP at gmp.jl:390
[16] +(x::BigInt, y::BigInt) in Base.GMP at gmp.jl:361
[17] +(x::BigInt, c::Union{UInt16, UInt32, UInt64, UInt8}) in Base.GMP at gmp.jl:398
...
[180] +(a, b, c, xs...) in Base at operators.jl:424
```

Multiple dispatch together with the flexible parametric type system give Julia its ability to abstractly express high-level algorithms decoupled from implementation details.

12.2 Method specializations

When you create multiple methods of the same function, this is sometimes called "specialization." In this case, you're specializing the *function* by adding additional methods to it: each new method is a new specialization of the function. As shown above, these specializations are returned by `methods`.

There's another kind of specialization that occurs without programmer intervention: Julia's compiler can automatically specialize the *method* for the specific argument types used. Such specializations are *not* listed by `methods`, as this doesn't create new `Method`s, but tools like `@code_typed` allow you to inspect such specializations.

For example, if you create a method

```
mysum(x::Real, y::Real) = x + y
```

you've given the function `mysum` one new method (possibly its only method), and that method takes any pair of `Real` number inputs. But if you then execute

```
julia> mysum(1, 2)
3

julia> mysum(1.0, 2.0)
3.0
```

Julia will compile `mysum` twice, once for `x::Int, y::Int` and again for `x::Float64, y::Float64`. The point of compiling twice is performance: the methods that get called for `+` (which `mysum` uses) vary depending on the specific types of `x` and `y`, and by compiling different specializations Julia can do all the method lookup ahead of time. This allows the program to run much more quickly, since it does not have to bother with method lookup while it is running. Julia's automatic specialization allows you to write generic algorithms and expect that the compiler will generate efficient, specialized code to handle each case you need.

In cases where the number of potential specializations might be effectively unlimited, Julia may avoid this default specialization. See Be aware of when Julia avoids specializing for more information.

12.3 方法歧义

在一系列的函数方法定义时有可能没有单独的最专用的方法能适用于参数的某些组合：

```
julia> g(x::Float64, y) = 2x + y
g (generic function with 1 method)

julia> g(x, y::Float64) = x + 2y
g (generic function with 2 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
ERROR: MethodError: g(::Float64, ::Float64) is ambiguous.

Candidates:
  g(x, y::Float64)
    @ Main none:1
  g(x::Float64, y)
    @ Main none:1

Possible fix, define
  g(::Float64, ::Float64)

Stacktrace:
[...]
```

这里 `g(2.0,3.0)` 的调用使用 `g(Float64, Any)` 和 `g(Any, Float64)` 都能处理，并且两个都不更加专用。在这样的情况下，Julia 会扔出`MethodError`而非任意选择一个方法。你可以通过对交叉情况指定一个合适的方法来避免方法歧义：

```
julia> g(x::Float64, y::Float64) = 2x + 2y
g (generic function with 3 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
10.0
```

建议先定义没有歧义的方法，因为不这样的话，歧义就会存在，即使是暂时性的，直到更加专用的方法被定义。

在更加复杂的情况下，解决方法歧义会会涉及到设计的某一个元素；这个主题将会在下面进行进一步的探索。

12.4 参数方法

方法定义可以视需要存在限定特征的类型参数：

```
julia> same_type(x::T, y::T) where {T} = true
same_type (generic function with 1 method)

julia> same_type(x,y) = false
same_type (generic function with 2 methods)
```

第一个方法应用于两个参数都是同一个具体类型时，不管类型是什么，而第二个方法接受一切，涉及其他所有情况。所以，总得来说，这个定义了一个布尔函数来检查两个参数是否是同样的类型：

```
julia> same_type(1, 2)
true

julia> same_type(1, 2.0)
false

julia> same_type(1.0, 2.0)
true

julia> same_type("foo", 2.0)
false

julia> same_type("foo", "bar")
true

julia> same_type(Int32(1), Int64(2))
false
```

这样的定义对应着那些类型签名是 `UnionAll` 类型的方法（参见 UnionAll 类型）。

在 Julia 中这种通过分派进行函数行为的定义是十分常见的，甚至是惯用的。方法类型参数并不局限于用作参数的类型：他们可以用在任意地方，只要值会在函数或者函数体的特征中。这里有个例子，例子中方法类型参数 `T` 用作方法特征中的参数类型 `Vector{T}` 的类型参数：

```
julia> myappend(v::Vector{T}, x::T) where {T} = [v..., x]
myappend (generic function with 1 method)

julia> myappend([1,2,3],4)
4-element Vector{Int64}:
 1
 2
 3
 4

julia> myappend([1,2,3],2.5)
ERROR: MethodError: no method matching myappend(::Vector{Int64}, ::Float64)
```

```
Closest candidates are:
  myappend(::Vector{T}, !Matched::T) where T
   @ Main none:1

Stacktrace:
[...]

julia> myappend([1.0,2.0,3.0],4.0)
4-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0

julia> myappend([1.0,2.0,3.0],4)
ERROR: MethodError: no method matching myappend(::Vector{Float64}, ::Int64)

Closest candidates are:
  myappend(::Vector{T}, !Matched::T) where T
   @ Main none:1

Stacktrace:
[...]
```

如你所看到的，追加的元素的类型必须匹配它追加到的向量的元素类型，否则会引起MethodError。在下面的例子中，方法类型参量 T 用作返回值：

```
julia> mytypeof(x::T) where {T} = T
mytypeof (generic function with 1 method)

julia> mytypeof(1)
Int64

julia> mytypeof(1.0)
Float64
```

就像你能在类型声明时通过类型参数对子类型进行约束一样（参见参数类型），你也可以约束方法的类型参数：

```
julia> same_type_numeric(x::T, y::T) where {T<:Number} = true
same_type_numeric (generic function with 1 method)

julia> same_type_numeric(x::Number, y::Number) = false
same_type_numeric (generic function with 2 methods)

julia> same_type_numeric(1, 2)
true

julia> same_type_numeric(1, 2.0)
false

julia> same_type_numeric(1.0, 2.0)
true
```

```
julia> same_type_numeric("foo", 2.0)
ERROR: MethodError: no method matching same_type_numeric(::String, ::Float64)

Closest candidates are:
  same_type_numeric(!Matched::T, ::T) where T<:Number
   @ Main none:1
  same_type_numeric(!Matched::Number, ::Number)
   @ Main none:1

Stacktrace:
[...]

julia> same_type_numeric("foo", "bar")
ERROR: MethodError: no method matching same_type_numeric(::String, ::String)

julia> same_type_numeric(Int32(1), Int64(2))
false
```

same_type_numeric 函数的行为与上面定义的 same_type 函数基本相似，但是它只对一对数定义。

参数方法允许与 where 表达式同样的语法用来写类型（参见 UnionAll 类型）。如果只有一个参数，封闭的大括号（在 where {T} 中）可以省略，但是为了清楚起见推荐写上。多个参数可以使用逗号隔开，例如 where {T, S <: Real}，或者使用嵌套的 where 来写，例如 where S<:Real where T。

12.5 重定义方法

当重定义一个方法或者增加一个方法时，知道这个变化不会立即生效很重要。这是 Julia 能够静态推断和编译代码使其运行很快而没有惯常的 JIT 技巧和额外开销的关键。实际上，任意新的方法定义不会对当前运行环境可见，包括 Tasks 和线程（和所有的之前定义的 @generated 函数）。让我们通过一个例子说明这意味着什么：

```
julia> function tryeval()
           @eval newfun() = 1
           newfun()
       end
tryeval (generic function with 1 method)

julia> tryeval()
ERROR: MethodError: no method matching newfun()
The applicable method may be too new: running in world age xxxx1, while current world is xxxx2.
Closest candidates are:
  newfun() at none:1 (method too new to be called from this world context.)
 in tryeval() at none:1
 ...

julia> newfun()
1
```

在这个例子中看到 newfun 的新定义已经被创建，但是并不能立即调用。新的全局变量立即对 tryeval 函数可见，所以你可以写 return newfun（没有小括号）。但是你，你的调用器，和他们调用的函数等等都不能调用这个新的方法定义！

但是这里有个例外：之后的在 *REPL* 中的 newfun 的调用会按照预期工作，能够见到并调用 newfun 的新定义。

但是，之后的 tryeval 的调用将会继续看到 newfun 的定义，因为该定义位于 *REPL* 的前一个语句中并因此在之后的 tryeval 的调用之前。

你可以试试这个来让自己了解这是如何工作的。

这个行为的实现通过一个「world age 计数器」。这个单调递增的值会跟踪每个方法定义操作。此计数器允许用单个数字描述「对于给定运行时环境可见的方法定义集」，或者说「world age」。它还允许仅仅通过其序数值来比较在两个 world 中可用的方法。在上例中，我们看到（方法 newfun 所存在的）「current world」比局部于任务的「runtime world」大一，后者在 tryeval 开始执行时是固定的。

有时规避这个是必要的（例如，如果你在实现上面的 REPL）。幸运的是这里有个简单地解决方法：使用Base.invokelatest调用函数：

```
julia> function tryeval2()
           @eval newfun2() = 2
           Base.invokelatest(newfun2)
       end
tryeval2 (generic function with 1 method)

julia> tryeval2()
2
```

最后，让我们看一些这个规则生效的更复杂的例子。定义一个函数 f(x)，最开始有一个方法：

```
julia> f(x) = "original definition"
f (generic function with 1 method)
```

开始一些使用 f(x) 的运算：

```
julia> g(x) = f(x)
g (generic function with 1 method)

julia> t = @async f(wait()); yield();
```

现在我们给 f(x) 加上一些新的方法：

```
julia> f(x::Int) = "definition for Int"
f (generic function with 2 methods)

julia> f(x::Type{Int}) = "definition for Type{Int}"
f (generic function with 3 methods)
```

比较一下这些结果如何不同：

```
julia> f(1)
"definition for Int"

julia> g(1)
"definition for Int"
```

```
julia> fetch(schedule(t, 1))
"original definition"

julia> t = @async f(wait()); yield();

julia> fetch(schedule(t, 1))
"definition for Int"
```

12.6 使用参数方法设计样式

虽然复杂的分派逻辑对于性能或者可用性并不是必须的，但是有时这是表达某些算法的最好的方法。这里有一些常见的设计样式，在以这个方法使用分派时有时会出现。

从超类型中提取出类型参数

以下是一个正确的代码模板，用于返回具有明确定义的元素类型的 `AbstractArray` 的任意子类型的元素类型 `T`：

```
abstract type AbstractArray{T, N} end
eltype(::Type{<:AbstractArray{T}}) where {T} = T
```

using so-called triangular dispatch. Note that `UnionAll` types, for example `eltype(AbstractArray{T} where T <: Integer)`, do not match the above method. The implementation of `eltype` in Base adds a fallback method to Any for such cases.

一个常见的错误是试着使用内省来得到元素类型：

```
eltype_wrong(::Type{A}) where {A<:AbstractArray} = A.parameters[1]
```

但是创建一个这个方法会失败的情况不难：

```
struct BitVector <: AbstractArray{Bool, 1}; end
```

这里我们已经创建了一个没有参数的类型 `BitVector`，但是元素类型已经完全指定了，`T` 等于 `Bool`！

另一个错误是尝试使用 `supertype` 沿着类型层次结构向上走：

```
eltype_wrong(::Type{AbstractArray{T}}) where {T} = T
eltype_wrong(::Type{AbstractArray{T, N}}) where {T, N} = T
eltype_wrong(::Type{A}) where {A<:AbstractArray} = eltype_wrong(supertype(A))
```

While this works for declared types, it fails for types without supertypes:

```
julia> eltype_wrong(Union{AbstractArray{Int}, AbstractArray{Float64}})
ERROR: MethodError: no method matching supertype(::Type{Union{AbstractArray{Float64,N} where N,
↪  AbstractArray{Int64,N} where N}})
Closest candidates are:
  supertype(::DataType) at operators.jl:43
  supertype(::UnionAll) at operators.jl:48
```

用不同的类型参数构建相似的类型

当构建通用代码时，通常需要创建一些类似对象，在类型的布局上有一些变化，这就也让类型参数的变化变得必要。例如，你会有一些任意元素类型的抽象数组，想使用特定的元素类型来编写你基于它的计算。你必须实现为每个 `AbstractArray{T}` 的子类型实现方法，这些方法描述了如何计算类型转换。从一个子类型转化成拥有一个不同参数的另一个子类型的通用方法在这里不存在。

`AbstractArray` 的子类型典型情况下会实现两个方法来完成这个：一个方法把输入输入转换成特定的 `AbstractArray{T,N}` 抽象类型的子类型；一个方法用特定的元素类型构建一个新的未初始化的数组。这些的样例实现可以在 Julia Base 里面找到。这里是一个基础的样例使用，保证 `input` 与 `output` 是同一种类型：

```
input = convert(AbstractArray{Eltype}, input)
output = similar(input, Eltype)
```

作为这个的扩展，在算法需要输入数组的拷贝的情况下，`convert`使无法胜任的，因为返回值可能只是原始输入的别名。把`similar`（构建输出数组）和`copyto!`（用输入数据填满）结合起来是需要给出输入参数的可变拷贝的一个范用方法：

```
copy_with_eltype(input, Eltype) = copyto!(similar(input, Eltype), input)
```

迭代分派

为了分派一个多层的参数参量列表，将每一层分派分开到不同的函数中常常是最好的。这可能听起来跟单分派的方法相似，但是你会在下面见到，这个更加灵活。

例如，尝试按照数组的元素类型进行分派常常会引起歧义。相反地，常见的代码会首先按照容易类型分派，然后基于 eltype 递归到更加更加专用的方法。在大部分情况下，算法会很方便地就屈从与这个分层方法，在其他情况下，这种严苛的工作必须手动解决。这个分派分支能被观察到，例如在两个矩阵的加法的逻辑中：

```
# 首先分派选择了逐元素相加的 map 算法。
+(a::Matrix, b::Matrix) = map(+, a, b)
# 然后分派处理了每个元素然后选择了计算的
# 恰当的常见元素类型。
+(a, b) = +(promote(a, b)...)
# 一旦元素有了相同类型，它们就可以相加。
# 例如，通过处理器暴露出的原始运算。
+(a::Float64, b::Float64) = Core.add(a, b)
```

基于 Trait 的分派

对于上面的可迭代分派的一个自然扩展是给方法选择加一个内涵层，这个层允许按照那些与类型层级定义的集合相独立的类型的集合来分派。我们可以通过写出问题中的类型的一个 `Union` 来创建这个一个集合，但是这不能够扩展，因为 `Union` 类型在创建之后无法改变。但是这么一个可扩展的集合可以通过一个叫做"Holy-trait"的一个设计样式来实现。

这个样式是通过定义一个范用函数来实现，这个函数为函数参数可能属于的每个 trait 集合都计算出不同的单例值（或者类型）。如果这个函数是单纯的，这与通常的分派对于性能没有任何影响。

上一部分中的示例掩盖了 `map` 和 `promote` 的实现细节，这两个都是依据 trait 来进行运算的。在迭代矩阵时，例如在 `map` 的实现中，一个重要的问题是使用什么顺序遍历数据。当 `AbstractArray` 子类型实现 `Base.IndexStyle` trait 时，`map` 等其他函数可以根据此信息进行派发以选择最佳算法（请参阅

[抽象数组接口](@ref man-interface-array))。这意味着每个子类型不需要实现 map 的自定义版本，因为通用定义 +trait 类将使系统能够选择最快的版本。下面是 map 的一个简单实现，说明了基于 trait 的调度：

```
map(f, a::AbstractArray, b::AbstractArray) = map(Base.IndexStyle(a, b), f, a, b)
# generic implementation:
map(::Base.IndexCartesian, f, a::AbstractArray, b::AbstractArray) = ...
# linear-indexing implementation (faster)
map(::Base.IndexLinear, f, a::AbstractArray, b::AbstractArray) = ...
```

这个基于 trait 的方法也出现在promote机制中，被标量 + 使用。它使用了promote_type，这在知道两个计算对象的类型的情况下返回计算这个运算的最佳的常用类型。这就使得我们不用为每一对可能的类型参数实现每一个函数，而把问题简化为对于每个类型实现一个类型转换运算这样一个小很多的问题，还有一个优选的逐对的类型提升规则的表格。

输出类型计算

基于 trait 的类型提升的讨论可以过渡到我们的下一个设计样式：为矩阵运算计算输出元素类型。

为了实现像加法这样的原始运算，我们使用promote_type函数来计算想要的输出类型。(像之前一样，我们在 + 调用中的 promote 调用中见到了这个工作)。

对于矩阵的更加复杂的函数，对于更加复杂的运算符序列来计算预期的返回类型是必要的。这经常按下列步骤进行：

1. 编写一个小函数 op 来表示算法核心中使用的运算的集合。
2. 使用 promote_op(op, argument_types...) 计算结果矩阵的元素类型 R，这里 argument_types 是通过应用到每个输入数组的 eltype 计算的。
3. 创建类似于 similar(R, dims) 的输出矩阵，这里 dims 是输出矩阵的预期维度数。

作为一个更加具体的例子，一个范用的方阵乘法的伪代码是：

```
function matmul(a::AbstractMatrix, b::AbstractMatrix)
    op = (ai, bi) -> ai * bi + ai * bi

    ## this is insufficient because it assumes `one(eltype(a))` is constructable:
    # R = typeof(op(one(eltype(a)), one(eltype(b))))

    ## this fails because it assumes `a[1]` exists and is representative of all elements of the
    ↪  array
    # R = typeof(op(a[1], b[1]))

    ## this is incorrect because it assumes that `+` calls `promote_type`
    ## but this is not true for some types, such as Bool:
    # R = promote_type(ai, bi)

    # this is wrong, since depending on the return value
    # of type-inference is very brittle (as well as not being optimizable):
    # R = Base.return_types(op, (eltype(a), eltype(b)))

    ## but, finally, this works:
    R = promote_op(op, eltype(a), eltype(b))
```

```
    ## although sometimes it may give a larger type than desired
    ## it will always give a correct type

    output = similar(b, R, (size(a, 1), size(b, 2)))
    if size(a, 2) > 0
        for j in 1:size(b, 2)
            for i in 1:size(a, 1)
                ## here we don't use `ab = zero(R)`,
                ## since `R` might be `Any` and `zero(Any)` is not defined
                ## we also must declare `ab::R` to make the type of `ab` constant in the loop,
                ## since it is possible that typeof(a * b) != typeof(a * b + a * b) == R
                ab::R = a[i, 1] * b[1, j]
                for k in 2:size(a, 2)
                    ab += a[i, k] * b[k, j]
                end
                output[i, j] = ab
            end
        end
    end
    return output
end
```

分离转换和内核逻辑

能有效减少编译时间和测试复杂度的一个方法是将预期的类型和计算转换的逻辑隔离。这会让编译器将与大型内核的其他部分相独立的类型转换逻辑特别化并内联。

将更大的类型类转换成被算法实际支持的特定参数类是一个常见的设计样式：

```
complexfunction(arg::Int) = ...
complexfunction(arg::Any) = complexfunction(convert(Int, arg))

matmul(a::T, b::T) = ...
matmul(a, b) = matmul(promote(a, b)...)
```

12.7 参数化约束的可变参数方法

函数参数也可以用于约束应用于”可变参数”函数（变参函数）的参数的数量。Vararg{T,N} 可用于表明这么一个约束。举个例子：

```
julia> bar(a,b,x::Vararg{Any,2}) = (a,b,x)
bar (generic function with 1 method)

julia> bar(1,2,3)
ERROR: MethodError: no method matching bar(::Int64, ::Int64, ::Int64)

Closest candidates are:
  bar(::Any, ::Any, ::Any, !Matched::Any)
   @ Main none:1

Stacktrace:
[...]
```

```
julia> bar(1,2,3,4)
(1, 2, (3, 4))

julia> bar(1,2,3,4,5)
ERROR: MethodError: no method matching bar(::Int64, ::Int64, ::Int64, ::Int64, ::Int64)

Closest candidates are:
  bar(::Any, ::Any, ::Any, ::Any)
   @ Main none:1

Stacktrace:
[...]
```

更加有用的是，用一个参数就约束可变参数的方法是可能的。例如：

```
function getindex(A::AbstractArray{T,N}, indices::Vararg{Number,N}) where {T,N}
```

只会在 `indices` 的个数与数组的维数相同时才会调用。

当只有提供的参数的类型需要被约束时，`Vararg{T}` 可以写成 `T...`。例如 `f(x::Int...) = x` 是 `f(x::Vararg{Int}) = x` 的简便写法。

12.8 可选参数和关键字的参数的注意事项

与在函数中简要提到的一样，可选参数是使用多方法定义语法来实现的。例如，这个定义：

```
f(a=1,b=2) = a+2b
```

翻译成下列三个方法：

```
f(a,b) = a+2b
f(a) = f(a,2)
f() = f(1,2)
```

这就意味着调用 `f()` 等于调用 `f(1,2)`。在这个情况下结果是 `5`，因为 `f(1,2)` 使用的是上面 `f` 的第一个方法。但是，不总是需要是这种情况。如果你定义了第四个对于整数更加专用的方法：

```
f(a::Int,b::Int) = a-2b
```

此时 `f()` 和 `f(1,2)` 的结果都是`-3`。换句话说，可选参数只与函数捆绑，而不是函数的任意一个特定的方法。这个决定于使用的方法的可选参数的类型。当可选参数是用全局变量的形式定义时，可选参数的类型甚至会在运行时改变。

关键字参数与普通的位置参数的行为很不一样。特别地，他们不参与到方法分派中。方法只基于位置参数分派，在匹配得方法确定之后关键字参数才会被处理。

12.9 类函数对象

方法与类型相关，所以可以通过给类型加方法使得任意一个 Julia 类型变得”可被调用”。（这个”可调用”的对象有时称为”函子”。）

例如，你可以定义一个类型，存储着多项式的系数，但是行为像是一个函数，可以为多项式求值：

```
julia> struct Polynomial{R}
           coeffs::Vector{R}
       end

julia> function (p::Polynomial)(x)
           v = p.coeffs[end]
           for i = (length(p.coeffs)-1):-1:1
               v = v*x + p.coeffs[i]
           end
           return v
       end

julia> (p::Polynomial)() = p(5)
```

注意函数是通过类型而非名字来指定的。如同普通函数一样这里有一个简洁的语法形式。在函数体内，p 会指向被调用的对象。Polynomial 会按如下方式使用：

```
julia> p = Polynomial([1,10,100])
Polynomial{Int64}([1, 10, 100])

julia> p(3)
931

julia> p()
2551
```

这个机制也是 Julia 中类型构造函数和闭包（指向其环境的内部函数）的工作原理。

12.10 空泛型函数

有时引入一个没有添加方法的范用函数是有用的。这会用于分离实现与接口定义。这也可为了文档或者代码可读性。为了这个的语法是没有参数组的一个空函数块：

```
function emptyfunc end
```

12.11 方法设计与避免歧义

Julia 的方法多态性是其最有力的特性之一，利用这个功能会带来设计上的挑战。特别地，在更加复杂的方法层级中出现歧义不能说不常见。

在上面我们曾经指出我们可以像这样解决歧义

```
f(x, y::Int) = 1
f(x::Int, y) = 2
```

靠定义一个方法

```
f(x::Int, y::Int) = 3
```

这通常是正确的方案；然而，在某些情况下，盲目地遵循这一建议可能会适得其反。特别是，泛型函数的方法越多，产生歧义的可能性就越大。当方法层次结构变得比这个简单的示例更复杂时，仔细考虑替代策略可能是值得的。

下面我们会讨论特别的一些挑战和解决这些挑战的一些可选方法。

元组和 N 元组参数

`Tuple`（和 `NTuple`）参数会带来特别的挑战。例如，

```
f(x::NTuple{N,Int}) where {N} = 1
f(x::NTuple{N,Float64}) where {N} = 2
```

是有歧义的，因为存在 `N == 0` 的可能性：没有元素去确定 `Int` 还是 `Float64` 变体应该被调用。为了解决歧义，一个方法是为空元组定义方法：

```
f(x::Tuple{}) = 3
```

作为一种选择，对于其中一个方法之外的所有的方法可以坚持元组中至少有一个元素：

```
f(x::NTuple{N,Int}) where {N} = 1           # this is the fallback
f(x::Tuple{Float64, Vararg{Float64}}) = 2   # this requires at least one Float64
```

正交化你的设计

当你打算根据两个或更多的参数进行分派时，考虑一下，一个「包裹」函数是否会让设计简单一些。举个例子，与其编写多变量：

```
f(x::A, y::A) = ...
f(x::A, y::B) = ...
f(x::B, y::A) = ...
f(x::B, y::B) = ...
```

不如考虑定义

```
f(x::A, y::A) = ...
f(x, y) = f(g(x), g(y))
```

这里 `g` 把参数转变为类型 `A`。这是更加普遍的正交设计原理的一个特别特殊的例子，在正交设计中不同的概念被分配到不同的方法中去。这里 `g` 最可能需要一个 fallback 定义

```
g(x::A) = x
```

一个相关的方案使用 promote 来把 x 和 y 变成常见的类型:

```
f(x::T, y::T) where {T} = ...
f(x, y) = f(promote(x, y)...)
```

这个设计的一个隐患是: 如果没有合适的把 x 和 y 转换到同样类型的类型提升方法，第二个方法就可能无限自递归然后引发堆溢出。

一次只根据一个参数分派

如果你你需要根据多个参数进行分派，并且有太多的为了能定义所有可能的变量而存在的组合，而存在很多回退函数，你可以考虑引入" 名字级联"，这里（例如）你根据第一个参数分配然后调用一个内部的方法:

```
f(x::A, y) = _fA(x, y)
f(x::B, y) = _fB(x, y)
```

接着内部方法 _fA 和 _fB 可以根据 y 进行分派，而不考虑有关 x 的歧义存在。

需要意识到这个方案至少有一个主要的缺点: 在很多情况下，用户没有办法通过进一步定义你的输出函数 f 的具体行为来进一步定制 f 的行为。相反，他们需要去定义你的内部方法 _fA 和 _fB 的具体行为，这会模糊输出方法和内部方法之间的界线。

抽象容器与元素类型

在可能的情况下要试图避免定义根据抽象容器的具体元素类型来分派的方法。举个例子，

```
-(A::AbstractArray{T}, b::Date) where {T<:Date}
```

会引起歧义，当定义了这个方法:

```
-(A::MyArrayType{T}, b::T) where {T}
```

最好的方法是不要定义这些方法中的任何一个。相反，使用范用方法-(A::AbstractArray, b) 并确认这个方法是使用分别对于每个容器类型和元素类型都是适用的通用调用 (像 similar 和-) 实现的。这只是建议正交化你的方法的一个更加复杂的变种而已。

当这个方法不可行时，这就值得与其他开发者开始讨论如果解决歧义; 只是因为一个函数先定义并不总是意味着他不能改变或者被移除。作为最后一个手段，开发者可以定义" 创可贴" 方法

```
-(A::MyArrayType{T}, b::Date) where {T<:Date} = ...
```

可以暴力解决歧义。

与默认参数的复杂方法" 级联"

如果你定义了提供默认的方法" 级联"，要小心去掉对应着潜在默认的任何参数。例如，假设你在写一个数字过滤算法，你有一个通过应用 padding 来出来信号的边的方法:

```
function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel)  # now perform the "real" computation
end
```

这会与提供默认 padding 的方法产生冲突:

```
myfilter(A, kernel) = myfilter(A, kernel, Replicate()) # replicate the edge by default
```

这两个方法一起会生成无限的递归，A 会不断变大。

更好的设计是像这样定义你的调用层级:

```
struct NoPad end  # indicate that no padding is desired, or that it's already applied

myfilter(A, kernel) = myfilter(A, kernel, Replicate())  # default boundary conditions

function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel, NoPad())  # indicate the new boundary conditions
end

# other padding methods go here

function myfilter(A, kernel, ::NoPad)
    # Here's the "real" implementation of the core computation
end
```

NoPad 被置于与其他 padding 类型一致的参数位置上，这保持了分派层级的良好组织，同时降低了歧义的可能性。而且，它扩展了「公开」的 myfilter 接口：想要显式控制 padding 的用户可以直接调用 NoPad 变量。

12.12 Defining methods in local scope

You can define methods within a local scope, for example

```
julia> function f(x)
           g(y::Int) = y + x
           g(y) = y - x
           g
       end
f (generic function with 1 method)

julia> h = f(3);

julia> h(4)
7

julia> h(4.0)
1.0
```

However, you should *not* define local methods conditionally or subject to control flow, as in

```
function f2(inc)
    if inc
        g(x) = x + 1
    else
        g(x) = x - 1
    end
end

function f3()
    function g end
    return g
    g() = 0
end
```

as it is not clear what function will end up getting defined. In the future, it might be an error to define local methods in this manner.

For cases like this use anonymous functions instead:

```
function f2(inc)
    g = if inc
        x -> x + 1
    else
        x -> x - 1
    end
end
```

[2] Arthur C. Clarke, *Profiles of the Future* (1961): Clarke's Third Law.

Chapter 13

构造函数

构造函数 [1] 是用来创建新对象的函数–确切地说，它创建的是复合类型的实例。在 Julia 中，类型对象也同时充当构造函数的角色：可以用类名加参数元组的方式像函数调用一样来创建新实例。这一点在介绍复合类型（Composite Types）时已经大致谈过了。例如：

```
julia> struct Foo
           bar
           baz
       end

julia> foo = Foo(1, 2)
Foo(1, 2)

julia> foo.bar
1

julia> foo.baz
2
```

对很多类型来说，通过给所有字段赋值来创建新对象的这种方式就足以用于产生新实例了。然而，在某些情形下，创建复合对象需要更多的功能。有时必须通过检查或转化参数来确保固有属性不变。递归数据结构，特别是那些可能引用自身的数据结构，它们通常不能被干净地构造，而是需要首先被不完整地构造，然后再通过编程的方式完成补全。为了方便，有时需要用较少的参数或者不同类型的参数来创建对象，Julia 的对象构造系统解决了所有这些问题。

13.1 外部构造方法

构造函数与 Julia 中的其他任何函数一样，其整体行为由其各个方法的组合行为定义。因此，只要定义新方法就可以向构造函数添加功能。例如，假设你想为 Foo 对象添加一个构造方法，该方法只接受一个参数并其作为 bar 和 baz 的值。这很简单：

```
julia> Foo(x) = Foo(x,x)
Foo
```

[1]命名法：虽然术语「构造函数」通常是指用于构造类型对象的函数全体，但通常会略微滥用术语将特定的构造方法称为「构造函数」。在这种情况下，通常可以从上下文中清楚地辨别出术语表示的是「构造方法」而不是「构造函数」，尤其是在讨论某个特别的「构造方法」的时候。

```
julia> Foo(1)
Foo(1, 1)
```

你也可以为 Foo 添加新的零参数构造方法，它为 bar 和 baz 提供默认值：

```
julia> Foo() = Foo(0)
Foo

julia> Foo()
Foo(0, 0)
```

这里零参数构造方法会调用单参数构造方法，单参数构造方法又调用了自动提供默认值的双参数构造方法。上面附加的这类构造方法，它们的声明方式与普通的方法一样，像这样的构造方法被称为**外部**构造方法，下文很快就会揭示这样称呼的原因。外部构造方法只能通过调用其他构造方法来创建新实例，比如自动提供默认值的构造方法。

13.2 内部构造方法

尽管外部构造方法可以成功地为构造对象提供了额外的便利，但它无法解决另外两个在本章导言里提到的问题：确保固有属性不变和允许创建自引用对象。因此，我们需要**内部**构造方法。内部构造方法和外部构造方法很相像，但有两点不同：

1. 内部构造方法在类型声明代码块的内部，而不是和普通方法一样在外部。
2. 内部构造方法能够访问一个特殊的局部函数 new，此函数能够创建该类型的对象。

例如，假设你要声明一个保存一对实数的类型，但要约束第一个数不大于第二个数。你可以像这样声明它：

```
julia> struct OrderedPair
           x::Real
           y::Real
           OrderedPair(x,y) = x > y ? error("out of order") : new(x,y)
       end
```

现在 OrderedPair 对象只能在 x <= y 时被成功构造：

```
julia> OrderedPair(1, 2)
OrderedPair(1, 2)

julia> OrderedPair(2,1)
ERROR: out of order
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] OrderedPair(::Int64, ::Int64) at ./none:4
 [3] top-level scope
```

如果类型被声明为 mutable，你可以直接更改字段值来打破这个固有属性，然而，在未经允许的情况下，随意摆弄对象的内核一般都是不好的行为。你（或者其他人）可以在以后任何时候提供额外的

外部构造方法，但一旦类型被声明了，就没有办法来添加更多的内部构造方法了。由于外部构造方法只能通过调用其它的构造方法来创建对象，所以最终构造对象的一定是某个内部构造函数。这保证了已声明类型的对象必须通过调用该类型的内部构造方法才得已存在，从而在某种程度上保证了类型的固有属性。

只要定义了任何一个内部构造方法，Julia 就不会再提供默认的构造方法：它会假定你已经为自己提供了所需的所有内部构造方法。默认构造方法等效于一个你自己编写的内部构造函数，该函数将所有成员作为参数（如果相应的字段具有类型，则约束为正确的类型），并将它们传递给 `new`，最后返回结果对象：

```
julia> struct Foo
           bar
           baz
           Foo(bar,baz) = new(bar,baz)
       end
```

这个声明与前面没有显式内部构造方法的 `Foo` 类型的定义效果相同。以下两个类型是等价的–一个具有默认构造方法，另一个具有显式构造方法：

```
julia> struct T1
           x::Int64
       end

julia> struct T2
           x::Int64
           T2(x) = new(x)
       end

julia> T1(1)
T1(1)

julia> T2(1)
T2(1)

julia> T1(1.0)
T1(1)

julia> T2(1.0)
T2(1)
```

提供尽可能少的内部构造方法是一种良好的形式：仅在需要显式地处理所有参数，以及强制执行必要的错误检查和转换时候才使用内部构造。其它用于提供便利的构造方法，比如提供默认值或辅助转换，应该定义为外部构造函数，然后再通过调用内部构造函数来执行繁重的工作。这种解耦是很自然的。

13.3 不完整初始化

最后一个还没提到的问题是，如何构造具有自引用的对象，更广义地来说是构造递归数据结构。由于这其中的困难并不是那么显而易见，这里我们来简单解释一下，考虑如下的递归类型声明：

```
julia> mutable struct SelfReferential
           obj::SelfReferential
       end
```

这种类型可能看起来没什么大不了，直到我们考虑如何来构造它的实例。如果 a 是 SelfReferential 的一个实例，则第二个实例可以用如下的调用来创建：

```
julia> b = SelfReferential(a)
```

但是，当没有实例存在的情况下，即没有可以传递给 obj 成员变量的有效值时，如何构造第一个实例？唯一的解决方案是允许使用未初始化的 obj 成员来创建一个未完全初始化的 SelfReferential 实例，并使用该不完整的实例作为另一个实例的 obj 成员的有效值，例如，它本身。

为了允许创建未完全初始化的对象，Julia 允许使用少于该类型成员数的参数来调用 new 函数，并返回一个具有某个未初始化成员的对象。然后，内部构造函数可以使用不完整的对象，在返回之前完成初始化。例如，我们在定义 SelfReferential 类型时采用了另一个方法，使用零参数内部构造函数来返回一个实例，此实例的 obj 成员指向其自身：

```
julia> mutable struct SelfReferential
           obj::SelfReferential
           SelfReferential() = (x = new(); x.obj = x)
       end
```

我们可以验证这一构造函数有效性，且由其构造的对象确实是自引用的：

```
julia> x = SelfReferential();

julia> x === x
true

julia> x === x.obj
true

julia> x === x.obj.obj
true
```

虽然从一个内部构造函数中返回一个完全初始化的对象是很好的，但是也可以返回未完全初始化的对象：

```
julia> mutable struct Incomplete
           data
           Incomplete() = new()
       end

julia> z = Incomplete();
```

尽管允许创建含有未初始化成员的对象，然而任何对未初始化引用的访问都会立即报错：

```
julia> z.data
ERROR: UndefRefError: access to undefined reference
```

这避免了不断地检测 null 值的需要。然而，并不是所有的对象成员都是引用。Julia 会将一些类型当作纯数据（"plain data"），这意味着它们的数据是自包含的，并且没有引用其它对象。The plain data

types consist of primitive types (e.g. Int) and immutable structs of other plain data types (see also: isbits, isbitstype). The initial contents of a plain data type is undefined:

```
julia> struct HasPlain
           n::Int
           HasPlain() = new()
       end

julia> HasPlain()
HasPlain(438103441441)
```

由纯数据组成的数组也具有一样的行为。

在内部构造函数中，你可以将不完整的对象传递给其它函数来委托其补全构造：

```
julia> mutable struct Lazy
           data
           Lazy(v) = complete_me(new(), v)
       end
```

与构造函数返回的不完整对象一样，如果 complete_me 或其任何被调用者尝试在初始化之前访问 Lazy 对象的 data 字段，就会立刻报错。

13.4 参数类型的构造函数

参数类型的存在为构造函数增加了更多的复杂性。首先，让我们回顾一下参数类型。在默认情况下，我们可以用两种方法来实例化参数复合类型，一种是显式地提供类型参数，另一种是让 Julia 根据构造函数输入参数的类型来隐式地推导类型参数。这里有一些例子：

```
julia> struct Point{T<:Real}
           x::T
           y::T
       end

julia> Point(1,2) ## 隐式的 T ##
Point{Int64}(1, 2)

julia> Point(1.0,2.5) ## 隐式的 T ##
Point{Float64}(1.0, 2.5)

julia> Point(1,2.5) ## 隐式的 T ##
ERROR: MethodError: no method matching Point(::Int64, ::Float64)
Closest candidates are:
  Point(::T, ::T) where T<:Real at none:2

julia> Point{Int64}(1, 2) ## 显式的 T ##
Point{Int64}(1, 2)

julia> Point{Int64}(1.0,2.5) ## 显式的 T ##
ERROR: InexactError: Int64(2.5)
Stacktrace:
[...]
```

```
julia> Point{Float64}(1.0, 2.5) ## 显式的 T ##
Point{Float64}(1.0, 2.5)

julia> Point{Float64}(1,2) ## 显式的 T ##
Point{Float64}(1.0, 2.0)
```

就像你看到的那样,用类型参数显式地调用构造函数,其参数会被转换为指定的类型:Point{Int64}(1,2) 可以正常工作,但是 Point{Int64}(1.0,2.5) 则会在将 2.5 转换为 Int64 的时候报一个 InexactError。当类型是从构造函数的参数隐式推导出来的时候,比如在例子 Point(1,2) 中,输入参数的类型必须一致,否则就无法确定 T 是什么,但 Point 的构造函数仍可以适配任意同类型的实数对。

实际上,这里的 Point, Point{Float64} 以及 Point{Int64} 是不同的构造函数。Point{T} 表示对于每个类型 T 都存在一个不同的构造函数。如果不显式提供内部构造函数,在声明复合类型 Point{T<:Real} 的时候,Julia 会对每个满足 T<:Real 条件的类型都提供一个默认的内部构造函数 Point{T},它们的行为与非参数类型的默认内部构造函数一致。Julia 同时也会提供了一个通用的外部构造函数 Point,用于适配任意同类型的实数对。Julia 默认提供的构造函数等价于下面这种显式的声明:

```
julia> struct Point{T<:Real}
           x::T
           y::T
           Point{T}(x,y) where {T<:Real} = new(x,y)
       end

julia> Point(x::T, y::T) where {T<:Real} = Point{T}(x,y);
```

注意,每个构造函数定义的方式与调用它们的方式是一样的。调用 Point{Int64}(1,2) 会触发 struct 块内部的 Point{T}(x,y)。另一方面,外部构造函数声明的 Point 构造函数只会被同类型的实数对触发,它使得我们可以直接以 Point(1,2) 和 Point(1.0,2.5) 这种方式来创建实例,而不需要显示地使用类型参数。由于此方法的声明方式已经对输入参数的类型施加了约束,像 Point(1,2.5) 这种调用自然会导致"no method" 错误。

假如我们想让 Point(1,2.5) 这种调用方式正常工作,比如,通过将整数 1 自动「提升」为浮点数 1.0,最简单的方法是像下面这样定义一个额外的外部构造函数:

```
julia> Point(x::Int64, y::Float64) = Point(convert(Float64,x),y);
```

此方法使用 convert 函数将 x 显式转换为 Float64,然后在两个参数都是 Float64 的情况下使用通用的构造函数。通过这个方法定义,以前的报MethodError的代码现在可以成功地创建一个类型为 Point{Float64} 的点:

```
julia> p = Point(1,2.5)
Point{Float64}(1.0, 2.5)

julia> typeof(p)
Point{Float64}
```

然而,其它类似的调用依然有问题:

```
julia> Point(1.5,2)
ERROR: MethodError: no method matching Point(::Float64, ::Int64)

Closest candidates are:
  Point(::T, !Matched::T) where T<:Real
   @ Main none:1

Stacktrace:
[...]
```

如果你想要找到一种方法可以使类似的调用都可以正常工作，请参阅类型转换与类型提升。这里稍稍“剧透”一下，我们可以利用下面的这个外部构造函数来满足需求，无论输入参数的类型如何，它都可以触发通用的 `Point` 构造函数：

```
julia> Point(x::Real, y::Real) = Point(promote(x,y)...);
```

这里的 `promote` 函数会将它的输入转化为同一类型，在此例中是 `Float64`。定义了这个方法，`Point` 构造函数会自动提升输入参数的类型，且提升机制与算术运算符相同，比如 `+`，因此对所有的实数输入参数都适用：

```
julia> Point(1.5,2)
Point{Float64}(1.5, 2.0)

julia> Point(1,1//2)
Point{Rational{Int64}}(1//1, 1//2)

julia> Point(1.0,1//2)
Point{Float64}(1.0, 0.5)
```

因此，虽然 Julia 中默认提供的隐式类型参数构造函数相当严格，但可以很容易地使它们以更轻松且明智的方式运行。此外，由于构造函数可以利用类型系统、方法和多重派发的所有功能，因此定义复杂的行为通常非常简单。

13.5 示例学习：有理数

也许将所有这些部分联系在一起的最佳方法是展示参数复合类型及其构造方法的真实示例。为此，我们实现了自己的有理数类型 `OurRational`，类似于 Julia 的内置 `Rational` 类型，定义在 [rational.jl](https://github.com/JuliaLang /julia/blob/master/base/rational.jl)：

```
julia> struct OurRational{T<:Integer} <: Real
           num::T
           den::T
           function OurRational{T}(num::T, den::T) where T<:Integer
               if num == 0 && den == 0
                    error("invalid rational: 0//0")
               end
               num = flipsign(num, den)
               den = flipsign(den, den)
               g = gcd(num, den)
               num = div(num, g)
```

```
                den = div(den, g)
                new(num, den)
            end
        end

julia> OurRational(n::T, d::T) where {T<:Integer} = OurRational{T}(n,d)
OurRational

julia> OurRational(n::Integer, d::Integer) = OurRational(promote(n,d)...)
OurRational

julia> OurRational(n::Integer) = OurRational(n,one(n))
OurRational

julia> ⊘(n::Integer, d::Integer) = OurRational(n,d)
⊘ (generic function with 1 method)

julia> ⊘(x::OurRational, y::Integer) = x.num ⊘ (x.den*y)
⊘ (generic function with 2 methods)

julia> ⊘(x::Integer, y::OurRational) = (x*y.den) ⊘ y.num
⊘ (generic function with 3 methods)

julia> ⊘(x::Complex, y::Real) = complex(real(x) ⊘ y, imag(x) ⊘ y)
⊘ (generic function with 4 methods)

julia> ⊘(x::Real, y::Complex) = (x*y') ⊘ real(y*y')
⊘ (generic function with 5 methods)

julia> function ⊘(x::Complex, y::Complex)
           xy = x*y'
           yy = real(y*y')
           complex(real(xy) ⊘ yy, imag(xy) ⊘ yy)
       end
⊘ (generic function with 6 methods)
```

第一行–`struct OurRational{T<:Integer} <: Real` –声明了 `OurRational` 会接收一个整数类型的类型参数，且它自己属于实数类型。它声明了两个成员：`num::T` 和 `den::T`。这表明一个 `OurRational{T}` 的实例中会包含一对整数，且类型为 `T`，其中一个表示分子，另一个表示分母。

Now things get interesting. `OurRational` has a single inner constructor method which checks that `num` and `den` aren't both zero and ensures that every rational is constructed in "lowest terms" with a non-negative denominator. This is accomplished by first flipping the signs of numerator and denominator if the denominator is negative. Then, both are divided by their greatest common divisor (`gcd` always returns a non-negative number, regardless of the sign of its arguments). Because this is the only inner constructor for `OurRational`, we can be certain that `OurRational` objects are always constructed in this normalized form.

为了方便，`OurRational` 也提供了一些其它的外部构造函数。第一个外部构造函数是“标准的”通用构造函数，当分子和分母的类型一致时，它就可以推导出类型参数 `T`。第二个外部构造函数可以用于分子和分母的类型不一致的情景，它会将分子和分母的类型提升至一个共同的类型，然后再委托第一个外部构造函数进行构造。第三个构造函数会将一个整数转化为分数，方法是将 `1` 当作分母。

在定义了外部构造函数之后，我们为 `⊘` 算符定义了一系列的方法，之后就可以使用 `⊘` 算符来写分数，（比如 `1 ⊘ 2`）。Julia 的 `Rational` 类型采用的是 `//` 算符。在做上述定义之前，`⊘` 是一个无意的且未被定义的算符。定义之后，它的行为与在 有理数 一节中描述的一致——注意它的所有行为都是那

短短几行定义的。第一个也是最基础的定义只是将 a ⊘ b 中的 a 和 b 当作参数传递给 OurRational 的构造函数来实例化 OurRational，这要求 a 和 b 分别都是整数。在 ⊘ 的某个操作数已经是分数的情况下，我们采用了一个有点不一样的方法来构建新的分数，这实际上等价于用分数除以一个整数。最后，我们也可以让 ⊘ 作用于复数，用来创建一个类型为 Complex{<:OurRational} 的对象——即一个实部和虚部都是分数的复数：

```
julia> z = (1 + 2im) ⊘ (1 - 2im);

julia> typeof(z)
Complex{OurRational{Int64}}

julia> typeof(z) <: Complex{<:OurRational}
true
```

因此，尽管 ⊘ 算符通常会返回一个 OurRational 的实例，但倘若其中一个操作数是复整数，那么就会返回 Complex{<:OurRational}。感兴趣的话可以读一读 rational.jl：它实现了一个完整的 Julia 基本类型，但却非常的简短，而且是自恰的。

13.6 仅外部的构造函数

正如我们所看到的，典型的参数类型都有一个内部构造函数，它仅在全部的类型参数都已知的情况下才会被调用。例如，可以用 Point{Int} 调用，但 Point 就不行。我们可以选择性的添加外部构造函数来自动推导并添加类型参数，比如，调用 Point(1,2) 来构造 Point{Int}。外部构造函数调用内部构造函数来实际创建实例。然而，在某些情况下，我们可能并不想要内部构造函数，从而达到禁止手动指定类型参数的目的。

例如，假设我们要定义一个类型用于存储向量以及其累加和：

```
julia> struct SummedArray{T<:Number,S<:Number}
           data::Vector{T}
           sum::S
       end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
SummedArray{Int32, Int32}(Int32[1, 2, 3], 6)
```

问题在于我们想让 S 的类型始终比 T 大，这样做是为了确保累加过程不会丢失信息。例如，当 T 是 Int32 时，我们想让 S 是 Int64。所以我们想要一种接口来禁止用户创建像 SummedArray{Int32,Int32} 这种类型的实例。一种实现方式是只提供一个 SummedArray 构造函数，当需要将其放入 struct-block 中，从而不让 Julia 提供默认的构造函数：

```
julia> struct SummedArray{T<:Number,S<:Number}
           data::Vector{T}
           sum::S
           function SummedArray(a::Vector{T}) where T
               S = widen(T)
               new{T,S}(a, sum(S, a))
           end
       end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
ERROR: MethodError: no method matching SummedArray(::Vector{Int32}, ::Int32)
```

```
Closest candidates are:
  SummedArray(::Vector{T}) where T
   @ Main none:4

Stacktrace:
[...]
```

此构造函数将会被 `SummedArray(a)` 这种写法触发。`new{T,S}` 的这种写法允许指定待构建类型的参数，也就是说调用它会返回一个 `SummedArray{T,S}` 的实例。`new{T,S}` 也可以用于其它构造函数的定义中，但为了方便，Julia 会根据正在构造的类型自动推导出 `new{}` 花括号里的参数（如果可行的话）。

Chapter 14

类型转换和类型提升

Julia 有一个提升系统，可以将数学运算符的参数提升为通用类型，如在前面章节中提到的整数和浮点数、数学运算和初等函数、类型和方法。在本节中，我们将解释类型提升系统如何工作，以及如何将其扩展到新的类型，并将其应用于除内置数学运算符之外的其他函数。传统上，编程语言在参数的类型提升上分为两大阵营:

- **内置数学类型和运算符的自动类型提升**。大多数语言中，内置数值类型，当作为带有中缀语法的算术运算符的操作数时，例如 +、-、* 和 / 将自动提升为通用类型，以产生预期的结果。举例来说，C、Java、Perl 和 Python，都将 `1 + 1.5` 的和作为浮点值 `2.5`，即使 + 的一个操作数是整数。这些系统非常方便且设计得足够精细，以至于它对于程序员来讲通常是不可见的：在编写这样的表达式时，几乎没有人有意识地想到这种类型提升，但编译器和解释器必须在相加前执行转换，因为整数和浮点值无法按原样相加。因此，这种自动类型转换的复杂规则不可避免地是这些语言的规范和实现的一部分。
- **没有自动类型提升**。这个阵营包括 Ada 和 ML——非常「严格的」静态类型语言。在这些语言中，每个类型转换都必须由程序员明确指定。因此，示例表达式 `1 + 1.5` 在 Ada 和 ML 中都会导致编译错误。相反地，必须编写 `real(1) + 1.5`，来在执行加法前将整数 `1` 显式转换为浮点值。然而，处处都显式转换是如此地不方便，以至于连 Ada 也有一定程度的自动类型转换：整数字面量被类型提升为预期的整数类型，浮点字面量同样被类型提升为适当的浮点类型。

在某种意义上，Julia 属于「无自动类型提升」类别：数学操作符只是具有特殊语法的函数，函数的参数永远不会自动转换。然而，人们可能会发现数学运算能应用于各种混合的参数类型，但这只是多态的多重分派的极端情况——这是 Julia 的分派和类型系统特别适合处理的情况。数学操作数的「自动」类型提升只是作为一个特殊的应用出现：Julia 带有预定义的数学运算符的 catch-all 分派规则，其在某些操作数类型的组合没有特定实现时调用。这些 catch-all 分派规则首先使用用户可定义的类型提升规则将所有操作数提升到一个通用的类型，然后针对结果值（现在已属于相同类型）调用相关运算符的特定实现。用户定义的类型可简单地加入这个类型提升系统，这需要先定义与其它类型进行相互类型转换的方法，接着提供一些类型提升规则来定义与其它类型混合时应该提升到什么类型。

14.1 类型转换

获取某种类型 `T` 的值的标准方法是调用该类型的构造函数 `T(x)`。但是，有些情况下，在程序员没有明确要求时，仍将值从一种类型转换为另一种类型是很方便的。其中一个例子是将值赋给一个数组：假设 `A` 是个 `Vector{Float64}`，表达式 `A[1] = 2` 执行时应该自动将 `2` 从 `Int` 转换为 `Float`，并将结果存储在该数组中。这通过 `convert` 函数完成。

`convert` 函数通常接受两个参数：第一个是类型对象，第二个是需要转换为该类型的值。返回的是已转换后的值。理解这个函数最简单的办法就是尝试：

```
julia> x = 12
12

julia> typeof(x)
Int64

julia> xu = convert(UInt8, x)
0x0c

julia> typeof(xu)
UInt8

julia> xf = convert(AbstractFloat, x)
12.0

julia> typeof(xf)
Float64

julia> a = Any[1 2 3; 4 5 6]
2×3 Matrix{Any}:
 1  2  3
 4  5  6

julia> convert(Array{Float64}, a)
2×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
```

类型转换并不总是可行的，有时 convert 函数并不知道该如何执行所请求的类型转换就会抛出 MethodError 错误。例如下例：

```
julia> convert(AbstractFloat, "foo")
ERROR: MethodError: Cannot `convert` an object of type String to an object of type AbstractFloat
[...]
```

一些语言考虑将解析字符串为数字或格式化数字为字符串来进行转换（许多动态语言甚至会自动执行转换），但 Julia 不会：尽管某些字符串可以解析为数字，但大多数字符串都不是有效的数字表示形式，只有非常有限的子集才是。因此，在 Julia 中，必须使用专用的 parse 函数来执行此操作，这使其更加明确。

什么时候使用 convert 函数?

构造以下语言结构时需要调用 convert 函数：

- 对一个数组赋值会转换为数组元素的类型。
- 对一个对象的字段赋值会转换为已声明的字段类型。
- 使用 new 构造对象会转换为该对象已声明的字段类型。
- 对已声明类型的变量赋值（例如 local x::T）会转换为该类型。
- 已声明返回类型的函数会转换其返回值为该类型。
- 把值传递给 ccall 会将其转换为相应参数的类型。

类型转换与构造

注意到 `convert(T, x)` 的行为似乎与 `T(x)` 几乎相同，它的确通常是这样。但是，有一个关键的语义差别：因为 `convert` 能被隐式调用，所以它的方法仅限于被认为是「安全」或「意料之内」的情况。`convert` 只会在表示事物的相同基本种类的类型之间进行转换（例如，不同的数字表示和不同的字符串编码）。它通常也是无损的；将值转换为其它类型并再次转换回去应该产生完全相同的值。

这是四种一般的构造函数与 `convert` 不同的情况：

与其参数类型无关的类型的构造函数

一些构造函数没有体现「转换」的概念。例如，`Timer(2)` 创建一个时长 2 秒的定时器，它实际上并不是从整数到定时器的「转换」。

可变的集合

如果 `x` 类型已经为 `T`，`convert(T, x)` 应该返回原本的 `x`。相反地，如果 `T` 是一个可变的集合类型，那么 `T(x)` 应该总是创建一个新的集合（从 `x` 复制元素）。

封装器类型

对于某些「封装」其它值的类型，构造函数可能会将其参数封装在一个新对象中，即使它已经是所请求的类型。例如，用 `Some(x)` 表示封装了一个 `x` 值（在上下文中，其结果可能是一个 `Some` 或 `nothing`）。但是，`x` 本身可能是对象 `Some(y)`，在这种情况下，结果为 `Some(Some(y))`，封装了两层。然而，`convert(Some, x)` 只会返回 `x`，因为它已经是 `Some` 的实例了。

不返回自身类型的实例的构造函数

在极少见的情况下，构造函数 `T(x)` 返回一个类型不为 `T` 的对象是有意义的。如果封装器类型是它自身的反转（例如 `Flip(Flip(x)) === x`），或者在重构库时为了支持某个旧的调用语法以实现向后兼容，则可能发生这种情况。但是，`convert(T, x)` 应该总是返回一个类型为 `T` 的值。

定义新的类型转换

在定义新类型时，最初创建它的所有方法都应定义为构造函数。如果隐式类型转换很明显是有用的，并且某些构造函数满足上面的「安全」标准，那么可以考虑添加 `convert` 方法。这些方法通常非常简单，因为它们只需要调用适当的构造函数。此类定义可能会像这样：

```
convert(::Type{MyType}, x) = MyType(x)
```

这个方法的第一个参数的类型是Type{MyType}，它的唯一实例是 `MyType`。因此，仅当第一个参数是类型值 `MyType` 时才会调用此方法。注意第一个参数使用的语法：在`::` 符号之前省略参数名称，只给出类型。这是 Julia 中指定类型但不需要通过名称引用其值的函数参数的语法。

某些抽象类型的所有实例默认都被认为是「足够相似的」，在 Julia Base 中也提供了通用的 `convert` 定义。例如，这个定义声明通过调用单参数构造函数将任何 `Number` 类型 `convert` 为其它任何 `Number` 类型是有效的：

```
convert(::Type{T}, x::Number) where {T<:Number} = T(x)::T
```

这意味着新的 `Number` 类型只需要定义构造函数，因为此定义将为它们处理 `convert`。在参数已经是所请求的类型的情况下，用恒同变换来处理 `convert`。

```
convert(::Type{T}, x::T) where {T<:Number} = x
```

AbstractString、AbstractArray 和 AbstractDict 也存在类似的定义。

14.2 类型提升

类型提升是指将一组混合类型的值转换为单个通用类型。尽管不是绝对必要的，但一般暗示被转换的值的通用类型可以忠实地表示所有原始值。此意义下，术语「类型提升」是合适的，因为值被转换为「更大」的类型——即能用一个通用类型表示所有输入值的类型。但重要的是，不要将它与面向对象（结构）超类或 Julia 的抽象超类型混淆：类型提升与类型层次结构无关，而与备选的表示之间的转换有关。例如，尽管每个 Int32 值可以表示为 Float64 值，但 Int32 不是 Float64 的子类型。

在 Julia 中，类型提升到一个通用的「更大」类型的操作是通过 promote 函数执行的，该函数接受任意数量的参数，并返回由相同数量的值组成的元组，值会被转换为一个通用类型，或在无法类型提升时抛出异常。类型提升的最常见用途是将数字参数转换为通用类型：

```
julia> promote(1, 2.5)
(1.0, 2.5)

julia> promote(1, 2.5, 3)
(1.0, 2.5, 3.0)

julia> promote(2, 3//4)
(2//1, 3//4)

julia> promote(1, 2.5, 3, 3//4)
(1.0, 2.5, 3.0, 0.75)

julia> promote(1.5, im)
(1.5 + 0.0im, 0.0 + 1.0im)

julia> promote(1 + 2im, 3//4)
(1//1 + 2//1*im, 3//4 + 0//1*im)
```

Floating-point values are promoted to the largest of the floating-point argument types. Integer values are promoted to the largest of the integer argument types. If the types are the same size but differ in signedness, the unsigned type is chosen. Mixtures of integers and floating-point values are promoted to a floating-point type big enough to hold all the values. Integers mixed with rationals are promoted to rationals. Rationals mixed with floats are promoted to floats. Complex values mixed with real values are promoted to the appropriate kind of complex value.

这就是使用类型提升的全部内容。剩下的只是聪明的应用，最典型的「聪明」应用是数值操作（如 +、-、* 和 /）的 catch-all 方法的定义。以下是在 promotion.jl 中给出的几个 catch-all 方法的定义：

```
+(x::Number, y::Number) = +(promote(x,y)...)
-(x::Number, y::Number) = -(promote(x,y)...)
*(x::Number, y::Number) = *(promote(x,y)...)
/(x::Number, y::Number) = /(promote(x,y)...)
```

这些方法的定义表明，如果没有更特殊的规则来加、减、乘及除一对数值，则将这些值提升为通用类型并再试一次。这就是它的全部内容：在其它任何地方都不需要为数值操作担心到通用数值类型的

类型提升——它会自动进行。许多算术和数学函数的 catch-all 类型提升方法的定义在 promotion.jl 中，但除此之外，Julia Base 中几乎不再需要调用 promote。promote 最常用于外部构造方法中，为了更方便，可允许使用混合类型的构造函数调用委托给一个内部构造函数，并将字段提升为适当的通用类型。例如，回想一下，rational.jl 提供了以下外部构造方法：

```
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)
```

这允许像下面这样的调用正常工作：

```
julia> x = Rational(Int8(15),Int32(-5))
-3//1

julia> typeof(x)
Rational{Int32}
```

对于大多数用户定义的类型，最好要求程序员明确地向构造函数提供期待的类型，但有时，尤其是对于数值问题，自动进行类型提升会很方便。

定义类型提升规则

虽然原则上可以直接为 promote 函数定义方法，但这需要为参数类型的所有可能排列下许多冗余的定义。相反地，promote 的行为是根据名为 promote_rule 的辅助函数定义的，该辅助函数可以为其提供方法。promote_rule 函数接受一对类型对象并返回另一个类型对象，这样参数类型的实例会被提升为被返回的类型。因此，通过定义规则：

```
promote_rule(::Type{Float64}, ::Type{Float32}) = Float64
```

声明当同时类型提升 64 位和 32 位浮点值时，它们应该被类型提升为 64 位浮点数。但是，提升类型不需要是参数类型之一；例如，在 Julia Base 中有以下类型提升规则：

```
promote_rule(::Type{BigInt}, ::Type{Float64}) = BigFloat
promote_rule(::Type{BigInt}, ::Type{Int8}) = BigInt
```

在后一种情况下，输出类型是 BigInt，因为 BigInt 是唯一一个足以容纳任意精度整数运算结果的类型。还要注意，不需要同时定义 promote_rule(::Type{A}, ::Type{B}) 和 promote_rule(::Type{B}, ::Type{A})——对称性隐含在类型提升过程中使用 promote_rule 的方式。

以 promote_rule 函数为基础定义了 promote_type 函数，在给定任意数量的类型对象时，它返回这些值作为 promote 的参数应被提升的通用类型。因此，如果想知道在没有实际值情况下，具有确定类型的一些值会被类型提升为什么类型，可以使用 promote_type：

```
julia> promote_type(Int8, Int64)
Int64
```

Note that we do **not** overload promote_type directly: we overload promote_rule instead. promote_type uses promote_rule, and adds the symmetry. Overloading it directly can cause ambiguity errors. We overload promote_rule to define how things should be promoted, and we use promote_type to query that.

在内部，promote_type 在 promote 中用于确定参数值应被转换为什么类型以便进行类型提升。好奇的读者可以阅读 promotion.jl，该文件用大概 35 行定义了完整的类型提升规则。

Internally, promote_type is used inside of promote to determine what type argument values should be converted to for promotion. The curious reader can read the code in promotion.jl, which defines the complete promotion mechanism in about 35 lines.

案例研究：有理数的类型提升

最后，我们来完成关于 Julia 的有理数类型的案例研究，该案例通过以下类型提升规则相对复杂地使用了类型提升机制：

```
promote_rule(::Type{Rational{T}}, ::Type{S}) where {T<:Integer,S<:Integer} =
↪   Rational{promote_type(T,S)}
promote_rule(::Type{Rational{T}}, ::Type{Rational{S}}) where {T<:Integer,S<:Integer} =
↪   Rational{promote_type(T,S)}
promote_rule(::Type{Rational{T}}, ::Type{S}) where {T<:Integer,S<:AbstractFloat} =
↪   promote_type(T,S)
```

第一条规则说，使用其它整数类型类型提升有理数类型会得到个有理数类型，其分子/分母类型是使用其它整数类型提升该有理数分子/分母类型的结果。第二条规则将相同的逻辑应用于两种不同的有理数类型，它们进行类型提升会得到有理数类型，其分子/分母类型是它们各自的分子/分母类型进行提升的结果。第三个也是最后一个规则规定，使用浮点数类型提升有理数类型与使用该浮点数类型提升其分子/分母类型会产生相同的类型。

这一小部分的类型提升规则，连同该类型的构造函数和数字的默认 convert 方法，便足以使有理数与 Julia 的其它数值类型——整数、浮点数和复数——完全自然地互操作。通过以相同的方式提供类型转换方法和类型提升规则，任何用户定义的数值类型都可像 Julia 的预定义数值类型一样自然地进行互操作。

Chapter 15

接口

Julia 的很多能力和扩展性都来自于一些非正式的接口。通过为自定义的类型扩展一些特定的方法，自定义类型的对象不但获得那些方法的功能，而且也能够用于其它的基于那些行为而定义的通用方法中。

15.1 迭代

必需方法		简短描述
`iterate(iter)`		通常返回由第一项及其初始状态组成的元组，但如果为空，则返回 `nothing`
`iterate(iter, state)`		通常返回由下一项及其状态组成的元组，或者在没有下一项存在时返回 `nothing`。
重要可选方法	**默认定义**	**简短描述**
`Base.IteratorSize(IterType)`	`Base.HasLength()`	`Base.HasLength()`，`Base.HasShape{N}()`，`Base.IsInfinite()` 或者 `Base.SizeUnknown()` 中合适的一个
`Base.IteratorEltype(IterType)`	`Base.HasEltype()`	`Base.EltypeUnknown()` 或 `Base.HasEltype()` 中合适的一个
`eltype(IterType)`	`Any`	由 `iterate()` 返回元组中第一项的类型。
`length(iter)`	(未定义)	项数，如果已知
`size(iter, [dim])`	(未定义)	在各个维度上项数，如果已知

由 `IteratorSize(IterType)` 返回的值	必需方法
`Base.HasLength()`	`length(iter)`
`Base.HasShape{N}()`	`length(iter)` 和 `size(iter, [dim])`
`Base.IsInfinite()`	(无)
`Base.SizeUnknown()`	(无)

由 `IteratorEltype(IterType)` 返回的值	必需方法
`Base.HasEltype()`	`eltype(IterType)`
`Base.EltypeUnknown()`	(*none*)

顺序迭代由 `iterate` 函数实现。Julia 的迭代器可以从对象外部跟踪迭代状态，而不是在迭代过程中改变对象本身。迭代过程中的返回一个包含了当前迭代值及其状态的元组，或者在没有元素存在的情况下返回 `nothing`。状态对象将在下一次迭代时传递回 iterate 函数，并且通常被认为是可迭代对象的私有实现细节。

任何定义了这个函数的对象都是可迭代的，并且可以被应用到许多依赖迭代的函数上 。也可以直接被应用到 for 循环中，因为根据语法：

```
for item in iter   # or  "for item = iter"
    # body
end
```

以上代码被解释为：

```
next = iterate(iter)
while next !== nothing
    (item, state) = next
    # body
    next = iterate(iter, state)
end
```

举一个简单的例子：一组定长数据的平方数迭代序列：

```
julia> struct Squares
           count::Int
       end

julia> Base.iterate(S::Squares, state=1) = state > S.count ? nothing : (state*state, state+1)
```

仅仅定义了 iterate 函数的 Squares 类型就已经很强大了。我们现在可以迭代所有的元素了：

```
julia> for item in Squares(7)
           println(item)
       end
1
4
9
16
25
36
49
```

我们可以利用许多内置方法来处理迭代： in 或 sum。

```
julia> 25 in Squares(10)
true

julia> sum(Squares(100))
338350
```

我们可以扩展一些其它的方法，为 Julia 提供有关此可迭代集合的更多信息。我们知道 Squares 序列中的元素总是 Int 型的。通过扩展 eltype 方法，我们可以给 Julia 更多信息来帮助其在更复杂的方法中生成更具体的代码。我们同时也知道该序列中的元素数目，故同样地也可以扩展 length：

```
julia> Base.eltype(::Type{Squares}) = Int # Note that this is defined for the type

julia> Base.length(S::Squares) = S.count
```

现在，当我们让 Julia 去 `collect` 所有元素到一个数组中时，Julia 可以预分配一个适当大小的 `Vector{Int}`，而不是朴素地 `push!` 每一个元素到 `Vector{Any}`：

```
julia> collect(Squares(4))
4-element Vector{Int64}:
  1
  4
  9
 16
```

尽管大多时候我们都可以依赖一些通用的实现，但某些时候，如果我们知道一个更简单的算法，可以用其扩展具体方法。例如，计算平方和有公式，因此可以扩展出一个更高效的解法来替代通用方法：

```
julia> Base.sum(S::Squares) = (n = S.count; return n*(n+1)*(2n+1)÷6)

julia> sum(Squares(1803))
1955361914
```

这种模式在 Julia Base 中很常见，一些必须实现的方法构成了一个小的集合，从而定义出一个非正式的接口，用于实现一些更加炫酷的操作。某些应用场景中，一些类型有更高效的算法，故可以扩展出额外的专用方法。

能以逆序迭代集合也很有用，这可由 `Iterators.reverse(iterator)` 迭代实现。但是，为了实际支持逆序迭代，迭代器类型 `T` 需要为 `Iterators.Reverse{T}` 实现 `iterate`。（给定 `r::Iterators.Reverse{T}`，类型 `T` 的底层迭代器是 `r.itr`。）在我们的 `Squares` 示例中，我们可以实现 `Iterators.Reverse{Squares}` 方法：

```
julia> Base.iterate(rS::Iterators.Reverse{Squares}, state=rS.itr.count) = state < 1 ? nothing :
↪  (state*state, state-1)

julia> collect(Iterators.reverse(Squares(4)))
4-element Vector{Int64}:
 16
  9
  4
  1
```

15.2 Indexing

For the `Squares` iterable above, we can easily compute the `i`th element of the sequence by squaring it. We can expose this as an indexing expression `S[i]`. To opt into this behavior, `Squares` simply needs to define `getindex`:

Methods to implement	Brief description
getindex(X, i)	X[i], indexed element access
setindex!(X, v, i)	X[i] = v, indexed assignment
firstindex(X)	The first index, used in X[begin]
lastindex(X)	The last index, used in X[end]

```
julia> function Base.getindex(S::Squares, i::Int)
           1 <= i <= S.count || throw(BoundsError(S, i))
           return i*i
       end

julia> Squares(100)[23]
529
```

另外，为了支持语法 S[begin] 和 S[end]，我们必须定义 lastindex 来指定最后一个有效索引。建议也定义 firstindex 来指定第一个有效索引：

```
julia> Base.firstindex(S::Squares) = 1

julia> Base.lastindex(S::Squares) = length(S)

julia> Squares(23)[end]
529
```

对多维的 begin/end 索引，例如，像是 a[3, begin, 7]，你应该定义 firstindex(a, dim) 和 lastindex(a, dim)（它们默认各自在 axes(a, dim) 上调用 first 和 last）

注意，上面只定义了一个整数索引的 getindex 方法，用除一个整数之外的其它东西索引会抛出MethodError，因为现在还没有匹配的方法。为了支持 Int 的范围或向量索引，必须另外写一个方法：

```
julia> Base.getindex(S::Squares, i::Number) = S[convert(Int, i)]

julia> Base.getindex(S::Squares, I) = [S[i] for i in I]

julia> Squares(10)[[3,4.,5]]
3-element Vector{Int64}:
  9
 16
 25
```

虽然这开始支持更多某些内置类型支持的索引操作，但仍然有很多行为不支持。因为我们为 Squares 序列所添加的行为，它开始看起来越来越像向量。我们可以正式定义其为 AbstractArray 的子类型，而不是自己定义所有这些行为。

15.3 抽象数组

如果一个类型被定义为 AbstractArray 的子类型，那它就继承了一大堆丰富的行为，包括构建在单元素访问之上的迭代和多维索引。有关更多支持的方法，请参阅文档 多维数组 及 Julia Base。

需要实现的方法		简短描述
size(A)		返回包含 A 各维度大小的元组
getindex(A, i::Int)		（若为 IndexLinear）线性标量索引
getindex(A, I::Vararg{Int, N})		（若为 IndexCartesian，其中 N = ndims(A)）N 维标量索引
可选方法	**默认定义**	**简短描述**
IndexStyle(::Type)	IndexCartesian()	返回 IndexLinear() 或 IndexCartesian()。请参阅下文描述。
setindex!(A, v, i::Int)		(if IndexLinear) Scalar indexed assignment
setindex!(A, v, I::Vararg{Int, N})		(if IndexCartesian, where N = ndims(A)) N-dimensional scalar indexed assignment
getindex(A, I...)	基于标量 getindex 定义	多维非标量索引
setindex!(A, X, I...)	基于标量 setindex! 定义	多维非标量索引元素赋值
iterate	基于标量 getindex 定义	Iteration
length(A)	prod(size(A))	元素数
similar(A)	similar(A, eltype(A), size(A))	返回具有相同形状和元素类型的可变数组
similar(A, ::Type{S})	similar(A, S, size(A))	返回具有相同形状和指定元素类型的可变数组
similar(A, dims::Dims)	similar(A, eltype(A), dims)	返回具有相同元素类型和大小为 *dims* 的可变数组
similar(A, ::Type{S}, dims::Dims)	Array{S}(undef, dims)	返回具有指定元素类型及大小的可变数组
不遵循惯例的索引	**默认定义**	**简短描述**
axes(A)	map(OneTo, size(A))	返回有效索引的 AbstractUnitRange{<:Integer}。The axes should be their own axes, that is axes.(axes(A),1) == axes(A) should be satisfied.
similar(A, ::Type{S}, inds)	similar(A, S, Base.to_shape(inds))	返回使用特殊索引 inds 的可变数组（详见下文）
similar(T::Union{Type,Function}, inds)	T(Base.to_shape(inds))	返回类似于 T 的使用特殊索引 inds 的数组（详见下文）

定义 AbstractArray 子类型的关键部分是 IndexStyle。由于索引是数组的重要部分且经常出现在 hot loops 中，使索引和索引赋值尽可能高效非常重要。数组数据结构通常以两种方式定义：要么仅使用一个索引（即线性索引）来最高效地访问其元素，要么实际上使用由各个维度确定的索引访问其元素。这两种方式被 Julia 标记为 IndexLinear() 和 IndexCartesian()。把线性索引转换为多重索引下标通常代价高昂，因此这提供了基于 traits 机制，以便能为所有矩阵类型提供高效的通用代码。

此区别决定了该类型必须定义的标量索引方法。IndexLinear() 很简单：只需定义 getindex(A::ArrayType, i::Int)。当数组后用多维索引集进行索引时，回退 getindex(A::AbstractArray, I...)() 高效地将

该索引转换为线性索引，然后调用上述方法。另一方面，IndexCartesian() 数组需要为每个支持的、使用 ndims(A) 个 Int 索引的维度定义方法。例如，SparseArrays 标准库里的 SparseMatrixCSC 只支持二维，所以它只定义了 getindex(A::SparseMatrixCSC, i::Int, j::Int)。setindex! 也是如此。

回到上面的平方数序列，我们可以将它定义为 AbstractArray{Int, 1} 的子类型：

```
julia> struct SquaresVector <: AbstractArray{Int, 1}
           count::Int
       end

julia> Base.size(S::SquaresVector) = (S.count,)

julia> Base.IndexStyle(::Type{<:SquaresVector}) = IndexLinear()

julia> Base.getindex(S::SquaresVector, i::Int) = i*i
```

请注意，指定 AbstractArray 的两个参数非常重要；第一个参数定义了 eltype，第二个则定义了 ndims。该超类型和这三个方法就足以使 SquaresVector 变成一个可迭代、可索引且功能齐全的数组：

```
julia> s = SquaresVector(4)
4-element SquaresVector:
  1
  4
  9
 16

julia> s[s .> 8]
2-element Vector{Int64}:
  9
 16

julia> s + s
4-element Vector{Int64}:
  2
  8
 18
 32

julia> sin.(s)
4-element Vector{Float64}:
  0.8414709848078965
 -0.7568024953079282
  0.4121184852417566
 -0.2879033166650653
```

作为一个更复杂的例子，让我们在 Dict 之上定义自己的玩具性质的 N 维稀疏数组类型。

```
julia> struct SparseArray{T,N} <: AbstractArray{T,N}
           data::Dict{NTuple{N,Int}, T}
           dims::NTuple{N,Int}
       end

julia> SparseArray(::Type{T}, dims::Int...) where {T} = SparseArray(T, dims);
```

```
julia> SparseArray(::Type{T}, dims::NTuple{N,Int}) where {T,N} =
↪  SparseArray{T,N}(Dict{NTuple{N,Int}, T}(), dims);

julia> Base.size(A::SparseArray) = A.dims

julia> Base.similar(A::SparseArray, ::Type{T}, dims::Dims) where {T} = SparseArray(T, dims)

julia> Base.getindex(A::SparseArray{T,N}, I::Vararg{Int,N}) where {T,N} = get(A.data, I, zero(T))

julia> Base.setindex!(A::SparseArray{T,N}, v, I::Vararg{Int,N}) where {T,N} = (A.data[I] = v)
```

请注意，这是个 IndexCartesian 数组，因此我们必须在数组的维度上手动定义 getindex 和 setindex!。与 SquaresVector 不同，我们可以定义 setindex!，这样便能更改数组：

```
julia> A = SparseArray(Float64, 3, 3)
3×3 SparseArray{Float64, 2}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> fill!(A, 2)
3×3 SparseArray{Float64, 2}:
 2.0  2.0  2.0
 2.0  2.0  2.0
 2.0  2.0  2.0

julia> A[:] = 1:length(A); A
3×3 SparseArray{Float64, 2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
```

索引 AbstractArray 的结果本身可以是数组（例如，在使用 AbstractRange 时）。AbstractArray 回退方法使用 similar 来分配具有适当大小和元素类型的 Array，该数组使用上述的基本索引方法填充。但是，在实现数组封装器时，你通常希望也封装结果：

```
julia> A[1:2,:]
2×3 SparseArray{Float64, 2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
```

在此例中，创建合适的封装数组通过定义 Base.similar(A::SparseArray, ::Type{T}, dims::Dims) where T 来实现。（请注意，虽然 similar 支持 1 参数和 2 参数形式，但在大多数情况下，你只需要专门定义 3 参数形式。）为此，SparseArray 是可变的（支持 setindex!）便很重要。为 SparseArray 定义 similar、getindex 和 setindex! 也使得该数组能够 copy 。

```
julia> copy(A)
3×3 SparseArray{Float64,2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
```

除了上面的所有可迭代和可索引方法之外，这些类型还能相互交互，并使用在 Julia Base 中为 AbstractArray 定义的大多数方法：

```
julia> A[SquaresVector(3)]
3-element SparseArray{Float64, 1}:
 1.0
 4.0
 9.0

julia> sum(A)
45.0
```

如果要定义允许非传统索引（索引以 1 之外的数字开始）的数组类型，你应该专门指定 axes。你也应该专门指定 similar，以便 dims 参数（通常是大小为 Dims 的元组）可以接收 AbstractUnitRange 对象，它也许是你自己设计的 range 类型 Ind。有关更多信息，请参阅使用自定义索引的数组。

15.4 等步长数组

实习方法		简要描述
strides(A)		返回每个维度中相邻元素之间的内存距离（以内存元素数量的形式）组成的元组。如果 A 是 AbstractArray{T,0}，这应该返回空元组。
Base.unsafe_convert(::Type{Ptr{T}}, A)		返回数组的本地内存地址
Base.elsize(::Type{<:A})		返回数组中连续元素的步长
可选方法	**默认定义**	**简要描述**
stride(A, i::Int)	strides(A)[i]	返回维度 i（译注：原文为 k）上相邻元素之间的内存距离（以内存元素数量的形式）。

等步长数组是 AbstractArray 的子类型，其条目以固定步长储存在内存中。如果数组的元素类型与 BLAS 兼容，则 strided 数组可以利用 BLAS 和 LAPACK 例程来实现更高效的线性代数例程。用户定义的 strided 数组的典型示例是把标准 Array 用附加结构进行封装的数组。

警告：如果底层存储实际上不是 strided，则不要实现这些方法，因为这可能导致错误的结果或段错误。

下面是一些示例，用来演示哪些数组类型是 strided 数组，哪些不是：

```
1:5   # not strided (there is no storage associated with this array.)
Vector(1:5)  # is strided with strides (1,)
A = [1 5; 2 6; 3 7; 4 8]  # is strided with strides (1,4)
V = view(A, 1:2, :)   # is strided with strides (1,4)
V = view(A, 1:2:3, 1:2)   # is strided with strides (2,4)
V = view(A, [1,2,4], :)   # is not strided, as the spacing between rows is not fixed.
```

15.5 自定义广播

广播可由 broadcast 或 broadcast! 的显式调用、或者像 A .+ b 或 f.(x, y) 这样的「点」操作隐式触发。任何具有 axes 且支持索引的对象都可作为参数参与广播，默认情况下，广播结果储存在 Array 中。这个基本框架可通过三个主要方式扩展：

需要实现的方法	简短描述
`Base.BroadcastStyle(::Type{SrcType}) = SrcStyle()`	`SrcType` 的广播行为
`Base.similar(bc::Broadcasted{DestStyle}, ::Type{ElType})`	输出容器的分配
可选方法	
`Base.BroadcastStyle(::Style1, ::Style2) = Style12()`	混合广播风格的优先级规则
`Base.axes(x)`	用于广播的 x 的索引的声明（默认为 `axes(x)`）
`Base.broadcastable(x)`	将 x 转换为一个具有 `axes` 且支持索引的对象
绕过默认机制	
`Base.copy(bc::Broadcasted{DestStyle})`	`broadcast` 的自定义实现
`Base.copyto!(dest, bc::Broadcasted{DestStyle})`	专门针对 `DestStyle` 的自定义 `broadcast!` 实现
`Base.copyto!(dest::DestType, bc::Broadcasted{Nothing})`	专门针对 `DestStyle` 的自定义 `broadcast!` 实现
`Base.Broadcast.broadcasted(f, args...)`	覆盖融合表达式中的默认惰性行为
`Base.Broadcast.instantiate(bc::Broadcasted{DestStyle`	覆盖惰性广播的 `axes` 的计算

- 确保所有参数都支持广播
- 为给定参数集选择合适的输出数组
- 为给定参数集选择高效的实现

不是所有类型都支持 `axes` 和索引，但许多类型便于支持广播。`Base.broadcastable` 函数会在每个广播参数上调用，它能返回与广播参数不同的支持 `axes` 和索引的对象。默认情况下，对于所有 `AbstractArray` 和 `Number` 来说这是 identity 函数——因为它们已经支持 `axes` 和索引了。

If a type is intended to act like a "0-dimensional scalar" (a single object) rather than as a container for broadcasting, then the following method should be defined:

```
Base.broadcastable(o::MyType) = Ref(o)
```

that returns the argument wrapped in a 0-dimensional `Ref` container. For example, such a wrapper method is defined for types themselves, functions, special singletons like `missing` and `nothing`, and dates.

自定义类型可以类似地指定 `Base.broadcastable` 来定义其形状，但是它们应当遵循 `collect(Base.broadcastable(x)) == collect(x)` 的约定。一个值得注意的例外是 `AbstractString`；字符串是个特例，为了能被广播其表现为标量，尽管它们是其字符的可迭代集合（详见 字符串）。

接下来的两个步骤（选择输出数组和实现）依赖于如何确定给定参数集的唯一解。广播必须接受其参数的所有不同类型，并把它们折叠到一个输出数组和实现。广播称此唯一解为“风格”。每个可广播对象都有自己的首选风格，并使用类似于类型提升的系统将这些风格组合成一个唯一解——“目标风格”。

广播风格

抽象类型 `Base.BroadcastStyle` 派生了所有的广播风格。其在用作函数时有两种可能的形式，分别为一元形式（单参数）和二元形式。使用一元形式表明你打算实现特定的广播行为和/或输出类型，并且不希望依赖于默认的回退 `Broadcast.DefaultArrayStyle`。

为了覆盖这些默认值，你可以为对象自定义 `BroadcastStyle`：

```
struct MyStyle <: Broadcast.BroadcastStyle end
Base.BroadcastStyle(::Type{<:MyType}) = MyStyle()
```

在某些情况下，无需定义 `MyStyle` 也许很方便，在这些情况下，你可以利用一个通用的广播封装器：

- `Base.BroadcastStyle(::Type{<:MyType}) = Broadcast.Style{MyType}()` 可用于任意类型。
- 如果 `MyType` 是一个 `AbstractArray`，首选是 `Base.BroadcastStyle(::Type{<:MyType}) = Broadcast.ArrayStyle{My`
- 对于只支持某个具体维度的 `AbstractArrays`，请创建 `Broadcast.AbstractArrayStyle{N}` 的子类型（请参阅下文）。

当你的广播操作涉及多个参数，各个广播风格将合并，来确定唯一一个 `DestStyle` 以控制输出容器的类型。有关更多详细信息，请参阅下文。

选择合适的输出数组

每个广播操作都会计算广播风格以便支持派发和专门化。结果数组的实际分配由 `similar` 处理，其使用 Broadcasted 对象作为其第一个参数。

```
Base.similar(bc::Broadcasted{DestStyle}, ::Type{ElType})
```

回退定义是

```
similar(bc::Broadcasted{DefaultArrayStyle{N}}, ::Type{ElType}) where {N,ElType} =
    similar(Array{ElType}, axes(bc))
```

但是，如果需要，你可以专门化任何或所有这些参数。最后的参数 `bc` 是（还可能是融合的）广播操作的惰性表示，即 `Broadcasted` 对象。出于这些目的，该封装器中最重要的字段是 `f` 和 `args`，分别描述函数和参数列表。请注意，参数列表可以——并且经常——包含其它嵌套的 `Broadcasted` 封装器。

举个完整的例子，假设你创建了类型 `ArrayAndChar`，该类型存储一个数组和单个字符：

```
struct ArrayAndChar{T,N} <: AbstractArray{T,N}
    data::Array{T,N}
    char::Char
end
Base.size(A::ArrayAndChar) = size(A.data)
Base.getindex(A::ArrayAndChar{T,N}, inds::Vararg{Int,N}) where {T,N} = A.data[inds...]
Base.setindex!(A::ArrayAndChar{T,N}, val, inds::Vararg{Int,N}) where {T,N} = A.data[inds...] = val
Base.showarg(io::IO, A::ArrayAndChar, toplevel) = print(io, typeof(A), " with char '", A.char, "'")
```

你可能想要广播保留“元数据”`char`。为此，我们首先定义

```
Base.BroadcastStyle(::Type{<:ArrayAndChar}) = Broadcast.ArrayStyle{ArrayAndChar}()
```

这意味着我们还必须定义相应的 `similar` 方法：

```julia
function Base.similar(bc::Broadcast.Broadcasted{Broadcast.ArrayStyle{ArrayAndChar}},
↪  ::Type{ElType}) where ElType
    # Scan the inputs for the ArrayAndChar:
    A = find_aac(bc)
    # Use the char field of A to create the output
    ArrayAndChar(similar(Array{ElType}, axes(bc)), A.char)
end

"`A = find_aac(As)` returns the first ArrayAndChar among the arguments."
find_aac(bc::Base.Broadcast.Broadcasted) = find_aac(bc.args)
find_aac(args::Tuple) = find_aac(find_aac(args[1]), Base.tail(args))
find_aac(x) = x
find_aac(::Tuple{}) = nothing
find_aac(a::ArrayAndChar, rest) = a
find_aac(::Any, rest) = find_aac(rest)
```

在这些定义中，可以得到以下行为：

```julia
julia> a = ArrayAndChar([1 2; 3 4], 'x')
2×2 ArrayAndChar{Int64, 2} with char 'x':
 1  2
 3  4

julia> a .+ 1
2×2 ArrayAndChar{Int64, 2} with char 'x':
 2  3
 4  5

julia> a .+ [5,10]
2×2 ArrayAndChar{Int64, 2} with char 'x':
  6   7
 13  14
```

使用自定义实现扩展广播

一般来说，广播操作由一个惰性 `Broadcasted` 容器表示，该容器保存要应用的函数及其参数。这些参数可能本身是嵌套得更深的 `Broadcasted` 容器，并一起形成了一个待求值的大型表达式树。嵌套的 `Broadcasted` 容器树可由隐式的点语法直接构造；例如，`5 .+ 2.*x` 由 `Broadcasted(+, 5, Broadcasted(*, 2, x))` 暂时表示。这对于用户是不可见的，因为它是通过调用 `copy` 立即实现的，但是此容器为自定义类型的作者提供了广播可扩展性的基础。然后，内置的广播机制将根据参数确定结果的类型和大小，为它分配内存，并最终通过默认的 `copyto!(::AbstractArray, ::Broadcasted)` 方法将 `Broadcasted` 对象复制到其中。内置的回退 `broadcast` 和 `broadcast!` 方法类似地构造操作的暂时 `Broadcasted` 表示，因此它们共享相同的代码路径。这便允许自定义的数组实现通过提供它们自己的专门化 `copyto!` 来定义和优化广播。这再次由计算后的广播风格确定。此广播风格在广播操作中非常重要，以至于它被存储为 `Broadcasted` 类型的第一个类型参数，且允许派发和专门化。

对于某些类型，跨越层层嵌套的广播的「融合」操作无法实现，或者无法更高效地逐步完成。在这种情况下，你可能需要或者想要求值 `x .* (x .+ 1)`，就好像该式已被编写成 `broadcast(*, x, broadcast(+, x, 1))`，其中内部广播操作会在处理外部广播操作前进行求值。这种直接的操作以有点间接的方式得到直接支持；Julia 不会直接构造 `Broadcasted` 对象，而会将待融合的表达式 `x .* (x .+ 1)` 降低为 `Broadcast.broadcasted(*, x, Broadcast.broadcasted(+, x, 1))`。现在，默认情况下，`broadcasted` 只会调用 `Broadcasted` 构造函数来创建待融合表达式树的惰性表示，但是你可以选择为函数和参数的特定组合覆盖它。

举个例子，内置的 AbstractRange 对象使用此机制优化广播表达式的片段，这些表达式片段可以只根据 start、step 和 length（或 stop）直接进行求值，而无需计算每个元素。与所有其它机制一样，broadcasted 也会计算并暴露其参数的组合广播风格，所以你可以为广播风格、函数和参数的任意组合专门化 broadcasted(::DestStyle, f, args...)，而不是专门化 broadcasted(f, args...)。

例如，以下定义支持 range 的负运算：

```
broadcasted(::DefaultArrayStyle{1}, ::typeof(-), r::OrdinalRange) = range(-first(r), step=-step(r),
↪  length=length(r))
```

扩展 in-place 广播

In-place 广播可通过定义合适的 copyto!(dest, bc::Broadcasted) 方法来支持。由于你可能想要专门化 dest 或 bc 的特定子类型，为了避免包之间的歧义，我们建议采用以下约定。

如果你想要专门化特定的广播风格 DestStyle，请为其定义一个方法

```
copyto!(dest, bc::Broadcasted{DestStyle})
```

你可选择使用此形式，如果使用，你还可以专门化 dest 的类型。

如果你想专门化目标类型 DestType 而不专门化 DestStyle，那么你应该定义一个带有以下签名的方法：

```
copyto!(dest::DestType, bc::Broadcasted{Nothing})
```

这利用了 copyto! 的回退实现，它将该封装器转换为一个 Broadcasted{Nothing} 对象。因此，专门化 DestType 的方法优先级低于专门化 DestStyle 的方法。

同样，你可以使用 copy(::Broadcasted) 方法完全覆盖 out-of-place 广播。

使用 Broadcasted 对象

当然，为了实现这样的 copy 或 copyto! 方法，你必须使用 Broadcasted 封装器来计算每个元素。这主要有两种方式：

- Broadcast.flatten 将可能的嵌套操作重新计算为单个函数并平铺参数列表。你自己负责实现广播形状规则，但这在有限的情况下可能会有所帮助。
- 迭代 axes(::Broadcasted) 的 CartesianIndices 并使用所生成的 CartesianIndex 对象的索引来计算结果。

编写二元广播规则

广播风格的优先级规则由二元 BroadcastStyle 调用定义：

```
Base.BroadcastStyle(::Style1, ::Style2) = Style12()
```

其中，Style12 是你要为输出所选择的 BroadcastStyle，所涉及的参数具有 Style1 及 Style2。例如，

```
Base.BroadcastStyle(::Broadcast.Style{Tuple}, ::Broadcast.AbstractArrayStyle{0}) =
↪  Broadcast.Style{Tuple}()
```

表示 Tuple「胜过」零维数组（输出容器将是元组）。值得注意的是，你不需要（也不应该）为此调用的两个参数顺序下定义；无论用户提供的以何种顺序提供参数，定义一个就够了。

对于 AbstractArray 类型，定义 BroadcastStyle 将取代回退选择 Broadcast.DefaultArrayStyle。DefaultArrayStyle 及其抽象超类型 AbstractArrayStyle 将维度存储为类型参数，以支持具有固定维度需求的特定数组类型。

由于以下方法，DefaultArrayStyle「输给」任何其它已定义的 AbstractArrayStyle：

```
BroadcastStyle(a::AbstractArrayStyle{Any}, ::DefaultArrayStyle) = a
BroadcastStyle(a::AbstractArrayStyle{N}, ::DefaultArrayStyle{N}) where N = a
BroadcastStyle(a::AbstractArrayStyle{M}, ::DefaultArrayStyle{N}) where {M,N} =
    typeof(a)(Val(max(M, N)))
```

除非你想要为两个或多个非 DefaultArrayStyle 的类型建立优先级，否则不需要编写二元 BroadcastStyle 规则。

如果你的数组类型确实有固定的维度需求，那么你应该定义一个 AbstractArrayStyle 的子类型。例如，稀疏数组的代码中有以下定义：

```
struct SparseVecStyle <: Broadcast.AbstractArrayStyle{1} end
struct SparseMatStyle <: Broadcast.AbstractArrayStyle{2} end
Base.BroadcastStyle(::Type{<:SparseVector}) = SparseVecStyle()
Base.BroadcastStyle(::Type{<:SparseMatrixCSC}) = SparseMatStyle()
```

每当你定义一个 AbstractArrayStyle 的子类型，你还需要定义用于组合维度的规则，这通过为你的广播风格创建带有一个 Val(N) 参数的构造函数。例如：

```
SparseVecStyle(::Val{0}) = SparseVecStyle()
SparseVecStyle(::Val{1}) = SparseVecStyle()
SparseVecStyle(::Val{2}) = SparseMatStyle()
SparseVecStyle(::Val{N}) where N = Broadcast.DefaultArrayStyle{N}()
```

这些规则表明 SparseVecStyle 与 0 维或 1 维数组的组合会产生另一个 SparseVecStyle，与 2 维数组的组合会产生 SparseMatStyle，而与维度更高的数组则回退到任意维密集矩阵的框架中。这些规则允许广播为产生一维或二维输出的操作保持其稀疏表示，但为任何其它维度生成 Array。

15.6 Instance Properties

Sometimes, it is desirable to change how the end-user interacts with the fields of an object. Instead of granting direct access to type fields, an extra layer of abstraction between the user and the code can be provided by overloading `object.field`. Properties are what the user *sees of* the object, fields what the object *actually is*.

By default, properties and fields are the same. However, this behavior can be changed. For example, take this representation of a point in a plane in polar coordinates:

Methods to implement	Default definition	Brief description
`propertynames(x::ObjType, private::Bool=false)`	`fieldnames(typeof(x))`	Return a tuple of the properties (`x.property`) of an object `x`. If `private=true`, also return property names intended to be kept as private
`getproperty(x::ObjType, s::Symbol)`	`getfield(x, s)`	Return property `s` of `x`. `x.s` calls `getproperty(x, :s)`.
`setproperty!(x::ObjType, s::Symbol, v)`	`setfield!(x, s, v)`	Set property `s` of `x` to `v`. `x.s = v` calls `setproperty!(x, :s, v)`. Should return `v`.

```
julia> mutable struct Point
           r::Float64
           ϕ::Float64
       end

julia> p = Point(7.0, pi/4)
Point(7.0, 0.7853981633974483)
```

As described in the table above dot access `p.r` is the same as `getproperty(p, :r)` which is by default the same as `getfield(p, :r)`:

```
julia> propertynames(p)
(:r, :ϕ)

julia> getproperty(p, :r), getproperty(p, :ϕ)
(7.0, 0.7853981633974483)

julia> p.r, p.ϕ
(7.0, 0.7853981633974483)

julia> getfield(p, :r), getproperty(p, :ϕ)
(7.0, 0.7853981633974483)
```

However, we may want users to be unaware that `Point` stores the coordinates as `r` and `ϕ` (fields), and instead interact with `x` and `y` (properties). The methods in the first column can be defined to add new functionality:

```
julia> Base.propertynames(::Point, private::Bool=false) = private ? (:x, :y, :r, :ϕ) : (:x, :y)

julia> function Base.getproperty(p::Point, s::Symbol)
           if s === :x
               return getfield(p, :r) * cos(getfield(p, :ϕ))
           elseif s === :y
               return getfield(p, :r) * sin(getfield(p, :ϕ))
           else
               # This allows accessing fields with p.r and p.ϕ
               return getfield(p, s)
           end
       end

julia> function Base.setproperty!(p::Point, s::Symbol, f)
```

```
        if s === :x
            y = p.y
            setfield!(p, :r, sqrt(f^2 + y^2))
            setfield!(p, :ϕ, atan(y, f))
            return f
        elseif s === :y
            x = p.x
            setfield!(p, :r, sqrt(x^2 + f^2))
            setfield!(p, :ϕ, atan(f, x))
            return f
        else
            # This allow modifying fields with p.r and p.ϕ
            return setfield!(p, s, f)
        end
    end
```

It is important that `getfield` and `setfield` are used inside `getproperty` and `setproperty!` instead of the dot syntax, since the dot syntax would make the functions recursive which can lead to type inference issues. We can now try out the new functionality:

```
julia> propertynames(p)
(:x, :y)

julia> p.x
4.949747468305833

julia> p.y = 4.0
4.0

julia> p.r
6.363961030678928
```

Finally, it is worth noting that adding instance properties like this is quite rarely done in Julia and should in general only be done if there is a good reason for doing so.

Chapter 16

模块

Julia 中的模块有助于将代码组织成连贯的部分。它们在语法上以 `module Name ... end` 界定，并具有以下特点：

1. 模块是独立的命名空间，每个都引入了一个新的全局作用域。这很有用，因为它允许对不同的函数或全局变量使用相同的名称而不会发生冲突，只要它们在不同的模块中即可。
2. 模块具有用于命名空间管理的工具：每个模块定义一组它 `export` 的名称，并且可以使用 `using` 和 `import` 从其他模块导入名称（我们将在下面解释这些）。
3. 模块可以预编译以加快加载速度，并可能包含用于运行时初始化的代码。

通常，在较大的 Julia 包中，你会看到模块的代码组织成文件，例如

```
module SomeModule

# export, using, import statements are usually here; we discuss these below

include("file1.jl")
include("file2.jl")

end
```

文件和文件名大多与模块无关；模块仅与模块表达式相关联。每个模块可以有多个文件，每个文件可以有多个模块。`include` 的行为就像在包含模块的全局作用域内执行源文件的内容一样。在本章中，我们使用简短和简化的示例，因此我们不会使用 `include`。

我们推荐不要缩进模块的主体，因为这通常会导致整个文件被缩进。此外，通常使用 `UpperCamelCase` 作为模块名称（就像类型一样），并在适用时使用复数形式，特别是如果模块包含类似命名的标识符，以避免名称冲突。例如，

```
module FastThings

struct FastThing
    ...
end

end
```

16.1 命名空间管理

命名空间管理是指语言提供的设施，用于使模块中的名称在其他模块中可用。我们在下面详细讨论相关的概念和功能。

合格的名称

全局作用域内的函数、变量和类型的名称，如 `sin`、`ARGS` 和 `UnitRange` 始终属于一个模块，称为母模块，例如，可以与`parentmodule`交互来找到该模块

```
julia> parentmodule(UnitRange)
Base
```

也可以通过在它们的模块前面加上前缀来引用它们的父模块之外的这些名称，例如 `Base.UnitRange`。这称为限定名称。父模块可以使用像 `Base.Math.sin` 这样的子模块链来访问，其中 `Base.Math` 被称为模块路径。由于句法歧义，限定只包含符号的名称，例如运算符，需要插入冒号，例如 `Base.:+`。少数运算符还需要括号，例如 `Base.:(==)`。

如果一个名称是限定的，那么它总是可访问的，在函数的情况下，它也可以通过使用限定的名称作为函数名称来添加方法。

在一个模块中，一个变量名可以通过将其声明 `global x` 不赋值而“保留”。这可以防止在加载时间后初始化的全局变量的名称冲突。语法 `M.x = y` 不适用于在另一个模块中分配一个全局变量；全局分配需要在模块本地进行操作。

导出列表

名称（指函数、类型、全局变量和常量）可以通过 `export` 添加到模块的 导出列表：这些是 `using` 模块时导入的符号。通常，它们位于或靠近模块定义的顶部，以便源代码的读者可以轻松找到它们，如：

```
julia> module NiceStuff
       export nice, DOG
       struct Dog end      # singleton type, not exported
       const DOG = Dog()   # named instance, exported
       nice(x) = "nice $x" # function, exported
       end;
```

但这只是一个风格建议——一个模块可以在任意位置有多个 `export` 语句。

导出构成 API（应用程序接口）一部分的名称是很常见的。在上面的代码中，导出列表建议用户应该使用 `nice` 和 `DOG`。然而，由于限定名称总是使标识符可访问，这只是组织 API 的一个选项：与其他语言不同，Julia 没有真正隐藏模块内部的功能。

此外，某些模块根本不导出名称。这通常是因为他们的 API 中使用常用词（例如 `derivative`），这很容易与其他模块的导出列表发生冲突。我们将在下面看到如何管理名称冲突。

单独使用 `using` 和 `import`

加载模块最常见的方式可能是 `using ModuleName`。这 加载 与 `ModuleName` 关联的代码，并引入

1. 模块名称

2. 和导出列表的元素到周围的全局命名空间中。

严格来说，声明 `using ModuleName` 意味着一个名为 `ModuleName` 的模块可用于根据需要解析名称。当遇到当前模块中没有定义的全局变量时，系统会在 `ModuleName` 导出的变量中查找，找到就使用。这意味着当前模块中该全局变量的所有使用都将解析为 `ModuleName` 中该变量的定义。

To load a module from a package, the statement `using ModuleName` can be used. To load a module from a locally defined module, a dot needs to be added before the module name like `using .ModuleName`.

继续我们的例子，

```
julia> using .NiceStuff
```

将加载上面的代码，使 `NiceStuff`（模块名称）、`DOG` 和 `nice` 可用。`Dog` 不在导出列表中，但如果名称被模块路径（这里只是模块名称）限定为 `NiceStuff.Dog`，则可以访问它。

重要的是，**导出列表只在 `using ModuleName` 的形式下起作用**。

相反，

```
julia> import .NiceStuff
```

仅将模块名称带入作用域。用户需要使用 `NiceStuff.DOG`、`NiceStuff.Dog` 和 `NiceStuff.nice` 来访问其内容。通常，当用户想要保持命名空间干净时，在上下文中使用 `import ModuleName`。正如我们将在下一节中看到的，`import .NiceStuff` 等同于 `using .NiceStuff: NiceStuff`。

你可以用逗号分隔符来组合相同类型的多个 `using` 和 `import` 语句，例如：

```
julia> using LinearAlgebra, Statistics
```

具有特定标识符的 `using` 和 `import`，并添加方法

当 `using ModuleName:` 或 `import ModuleName:` 后跟以逗号分隔的名称列表时，模块会被加载，但只有那些特定的名称才会被语句带入命名空间。例如，

```
julia> using .NiceStuff: nice, DOG
```

将导入名称 `nice` 和 `DOG`。

重要的是，模块名称 `NiceStuff` 不会出现在命名空间中。如果要使其可访问，则必须明确列出它，如：

```
julia> using .NiceStuff: nice, DOG, NiceStuff
```

Julia 有两种形式来表示似乎相同的内容，因为只有 `import ModuleName:f` 允许在没有模块路径的情况下向 `f` 添加方法。也就是说，以下示例将给出一个错误：

```
julia> using .NiceStuff: nice

julia> struct Cat end
```

```
julia> nice(::Cat) = "nice 😸"
ERROR: invalid method definition in Main: function NiceStuff.nice must be explicitly imported to be
↪  extended
Stacktrace:
 [1] top-level scope
   @ none:0
 [2] top-level scope
   @ none:1
```

此错误可防止意外将方法添加到你仅打算使用的其他模块中的函数。

有两种方法可以解决这个问题。你始终可以使用模块路径限定函数名称：

```
julia> using .NiceStuff

julia> struct Cat end

julia> NiceStuff.nice(::Cat) = "nice 😸"
```

或者，你可以 `import` 特定的函数名称：

```
julia> import .NiceStuff: nice

julia> struct Cat end

julia> nice(::Cat) = "nice 😸"
nice (generic function with 2 methods)
```

你选择哪一个取决于你的代码风格。第一种形式表明你正在向另一个模块中的函数添加一个方法（请记住，导入和方法定义可能在单独的文件中），而第二种形式较短，如果你定义了多个方法，这一点尤其方便。

一旦一个变量通过 `using` 或 `import` 引入，当前模块就不能创建同名的变量了。而且导入的变量是只读的，给全局变量赋值只能影响到由当前模块拥有的变量，否则会报错。

用 `as` 来重命名

由 `import` 或 `using` 引入作用域的标识符可以用关键字 `as` 重命名。这对于解决名称冲突以及缩短名称很有用。例如，`Base` 导出函数名 `read`，但 CSV.jl 包也提供了 `CSV.read`。如果我们要多次调用 CSV 读取，删除 `CSV.` 限定符会很方便。但是，我们指的是 `Base.read` 还是 `CSV.read` 是模棱两可的：

```
julia> read;

julia> import CSV: read
WARNING: ignoring conflicting import of CSV.read into Main
```

重命名提供了一个解决方案：

```
julia> import CSV: read as rd
```

导入的包本身也可以重命名:

```
import BenchmarkTools as BT
```

as 仅在将单个标识符引入作用域时才与 using 一起使用。例如，using CSV: read as rd 有效，但 using CSV as C 无效，因为它对 CSV 中的所有导出名称进行操作。

混合使用多个 using 和 import 语句

当使用上述任何形式的多个 using 或 import 语句时，它们的效果将按照它们出现的顺序组合。例如，

```
julia> using .NiceStuff         # exported names and the module name

julia> import .NiceStuff: nice  # allows adding methods to unqualified functions
```

会将 NiceStuff 的所有导出名称和模块名称本身带入作用域，并且还允许向 nice 添加方法而不用模块名称作为前缀。

处理名称冲突

考虑两个（或更多）包导出相同名称的情况，如

```
julia> module A
       export f
       f() = 1
       end
A
julia> module B
       export f
       f() = 2
       end
B
```

using .A, .B 语句有效，但是当你尝试调用 f 时，你会收到警告

```
julia> using .A, .B

julia> f
WARNING: both B and A export "f"; uses of it in module Main must be qualified
ERROR: UndefVarError: `f` not defined
```

在这里，Julia 无法确定您指的是哪个 f，因此你必须做出选择。常用的解决方法有以下几种:

1. 只需继续使用限定名称，如 A.f 和 B.f。这使代码的读者可以清楚地了解上下文，特别是如果 f 恰好重合但在不同的包中具有不同的含义。例如，degree 在数学、自然科学和日常生活中有多种用途，这些含义应该分开。

2. 使用上面的 as 关键字重命名一个或两个标识符，例如

```
julia> using .A: f as f

julia> using .B: f as g
```

会使 B.f 可用作 g。在这里，我们假设您之前没有使用 using A，这会把 f 代入命名空间。

3. 当问题中的多个名称确实有相同的含义时，通常一个模块会从另一个模块导入它，或者有一个轻量级的“基础”包，它的唯一功能是定义这样的接口，可以被其他包使用。按照惯例，这些包名以 ...Base 结尾（这与 Julia 的 Base 模块无关）

默认顶层定义以及裸模块

模块自动包含 using Core、using Base 以及 eval 和 include 函数的定义，这些函数在该模块的全局作用域内计算表达式/文件.

如果不需要这些默认定义，可以使用关键字 baremodule 来定义模块（注意：Core 仍然是导入的）。就 baremodule 而言，一个标准的 module 看起来像这样：

```
baremodule Mod

using Base

eval(x) = Core.eval(Mod, x)
include(p) = Base.include(Mod, p)

...

end
```

If even Core is not wanted, a module that imports nothing and defines no names at all can be defined with Module(:YourNameHere, false, false) and code can be evaluated into it with @eval or Core.eval:

```
julia> arithmetic = Module(:arithmetic, false, false)
Main.arithmetic

julia> @eval arithmetic add(x, y) = $(+)(x, y)
add (generic function with 1 method)

julia> arithmetic.add(12, 13)
25
```

标准模块

有三个重要的标准模块：

- Core 包含了语言“内置”的所有功能。
- Base 包含了绝大多数情况下都会用到的基本功能。
- Main 是顶层模块，当 julia 启动时，也是当前模块。

Standard library modules

默认情况下，Julia 附带了一些标准库模块。除了你不需要显式安装它们之外，它们的行为与常规 Julia 包类似。例如，如果您想执行一些单元测试，你可以按如下方式加载 `Test` 标准库：

```
using Test
```

16.2 子模块和相对路径

模块可以包含 子模块，嵌套相同的语法 `module ... end`。它们可用于引入单独的命名空间，这有助于组织复杂的代码库。请注意，每个 `module` 都引入了自己的 作用域，因此子模块不会自动从其父模块 “继承” 名称。

建议子模块在 `using` 和 `import` 语句中使用 相对模块限定符来引用封闭父模块中的其他模块（包括后者）。相对模块限定符以句点（`.`）开头，它对应于当前模块，每个连续的 `.` 都指向当前模块的父级。如有必要，这应该跟在模块之后，最后是要访问的实际名称，所有名称都以 `.` 分隔。

考虑以下示例，其中子模块 `SubA` 定义了一个函数，然后在其 “兄弟” 模块中进行扩展：

```
julia> module ParentModule
       module SubA
       export add_D  # exported interface
       const D = 3
       add_D(x) = x + D
       end
       using .SubA  # brings `add_D` into the namespace
       export add_D # export it from ParentModule too
       module SubB
       import ..SubA: add_D # relative path for a  “sibling”  module
       struct Infinity end
       add_D(x::Infinity) = x
       end
       end;
```

你可能会在包中看到代码，在类似的情况下，它使用

```
julia> import .ParentModule.SubA: add_D
```

然而，这是通过 代码加载 操作的，因此仅当 `ParentModule` 在包中时才有效。最好使用相对路径。

请注意，如果你正在评估值，定义的顺序也很重要。考虑

```
module TestPackage

export x, y

x = 0

module Sub
using ..TestPackage
z = y # ERROR: UndefVarError: `y` not defined
```

```
end

y = 1

end
```

其中 `Sub` 在定义之前尝试使用 `TestPackage.y`，因此它没有值。

出于类似的原因，你不能使用循环顺序：

```
module A

module B
using ..C # ERROR: UndefVarError: `C` not defined
end

module C
using ..B
end

end
```

16.3 模块初始化和预编译

因为执行模块中的所有语句通常需要编译大量代码，大型模块可能需要几秒钟才能加载。Julia 会创建模块的预编译缓存以减少这个时间。

Precompiled module files (sometimes called "cache files") are created and used automatically when `import` or `using` loads a module. If the cache file(s) do not yet exist, the module will be compiled and saved for future reuse. You can also manually call `Base.compilecache(Base.identify_package("modulename"))` to create these files without loading the module. The resulting cache files will be stored in the `compiled` subfolder of `DEPOT_PATH[1]`. If nothing about your system changes, such cache files will be used when you load the module with `import` or `using`.

Precompilation cache files store definitions of modules, types, methods, and constants. They may also store method specializations and the code generated for them, but this typically requires that the developer add explicit `precompile` directives or execute workloads that force compilation during the package build.

However, if you update the module's dependencies or change its source code, the module is automatically recompiled upon `using` or `import`. Dependencies are modules it imports, the Julia build, files it includes, or explicit dependencies declared by `include_dependency(path)` in the module file(s).

For file dependencies, a change is determined by examining whether the modification time (`mtime`) of each file loaded by `include` or added explicitly by `include_dependency` is unchanged, or equal to the modification time truncated to the nearest second (to accommodate systems that can't copy mtime with sub-second accuracy). It also takes into account whether the path to the file chosen by the search logic in `require` matches the path that had created the precompile file. It also takes into account the set of dependencies already loaded into the current process and won't recompile those modules, even if their files change or disappear, in order to avoid creating incompatibilities between the running system and the precompile cache. Finally, it takes account of changes in any compile-time preferences.

If you know that a module is *not* safe to precompile (for example, for one of the reasons described below), you should put `__precompile__(false)` in the module file (typically placed at the top). This will cause

Base.compilecache to throw an error, and will cause using / import to load it directly into the current process and skip the precompile and caching. This also thereby prevents the module from being imported by any other precompiled module.

在开发模块的时候，你可能需要了解一些与增量编译相关的固有行为。例如，外部状态不会被保留。为了解决这个问题，需要显式分离运行时与编译期的部分。Julia 允许你定义一个 __init__() 函数来执行任何需要在运行时发生的初始化。在编译期（--output-*），此函数将不会被调用。你可以假设在代码的生存周期中，此函数只会被运行一次。当然，如果有必要，你也可以手动调用它，但在默认的情况下，请假定此函数是为了处理与本机状态相关的信息，注意这些信息不需要，更不应该存入预编译镜像。此函数会在模块被导入到当前进程之后被调用，这包括在一个增量编译中导入该模块的时候（--output-incremental=yes），但在完整编译时该函数不会被调用。

特别的，如果你在模块里定义了一个名为 __init__() 的函数，那么 Julia 在加载这个模块之后会在第一次运行时（runtime）立刻调用这个函数（例如，通过 import，using，或者 require 加载时），也就是说 __init__ 只会在模块中所有其它命令都执行完以后被调用一次。因为这个函数将在模块完全载入后被调用，任何子模块或者已经载入的模块都将在当前模块调用 __init__ **之前**调用自己的 __init__ 函数。

__init__ 的典型用法有二，一是用于调用外部 C 库的运行时初始化函数，二是用于初始化涉及到外部库所返回的指针的全局常量。例如，假设我们正在调用一个 C 库 libfoo，它要求我们在运行时调用 foo_init() 这个初始化函数。假设我们还想定义一个全局常量 foo_data_ptr，它保存 libfoo 所定义的 void *foo_data() 函数的返回值——必须在运行时（而非编译时）初始化这个常量，因为指针地址不是固定的。可以通过在模块中定义 __init__ 函数来完成这个操作。

```
const foo_data_ptr = Ref{Ptr{Cvoid}}(0)
function __init__()
    ccall((:foo_init, :libfoo), Cvoid, ())
    foo_data_ptr[] = ccall((:foo_data, :libfoo), Ptr{Cvoid}, ())
    nothing
end
```

注意，在像 __init__ 这样的函数里定义一个全局变量是完全可以的，这是动态语言的优点之一。但是把全局作用域的值定义成常量，可以让编译器能确定该值的类型，并且能让编译器生成更好的优化过的代码。显然，你的模块（Module）中，任何其他依赖于 foo_data_ptr 的全局量也必须在 __init__ 中被初始化。

涉及大多数不是由 ccall 生成的 Julia 对象的常量不需要放在 __init__ 中：它们的定义可以从缓存的模块映像中预编译和加载。这包括复杂的堆分配对象，如数组。但是，任何返回原始指针值的例程都必须在运行时调用才能使预编译工作（Ptr 对象将变成空指针，除非它们隐藏在 isbits 目的）。这包括 Julia 函数 @cfunction 和 pointer 的返回值。

字典和集合类型，或者通常任何依赖于 hash(key) 方法的类型，都是比较棘手的情况。通常当键是数字、字符串、符号、范围、Expr 或这些类型的组合（通过数组、元组、集合、映射对等）时，可以安全地预编译它们。但是，对于一些其它的键类型，例如 Function 或 DataType、以及还没有定义散列方法的通用用户定义类型，回退（fallback）的散列（hash）方法依赖于对象的内存地址（通过 objectid），因此可能会在每次运行时发生变化。如果您有这些关键类型中的一种，或者您不确定，为了安全起见，您可以在您的 __init__ 函数中初始化这个字典。或者，您可以使用 IdDict 字典类型，它是由预编译专门处理的，因此在编译时初始化是安全的。

当使用预编译时，我们必须要清楚地区分代码的编译阶段和运行阶段。在此模式下，我们会更清楚发现 Julia 的编译器可以执行任何 Julia 代码，而不是一个用于生成编译后代码的独立的解释器。

其它已知的潜在失败场景包括：

1. 全局计数器，例如：为了试图唯一的标识对象。考虑以下代码片段：

```
mutable struct UniquedById
    myid::Int
    let counter = 0
        UniquedById() = new(counter += 1)
    end
end
```

尽管这段代码的目标是给每个实例赋一个唯一的 ID，但计数器的值会在代码编译结束时被记录。任何对此增量编译模块的后续使用，计数器都将从同一个值开始计数。

注意 `objectid` （工作原理是取内存指针的 hash）也有类似的问题，请查阅下面关于 `Dict` 的用法。

一种解决方案是用宏捕捉 `@__MODULE__`，并将它与目前的 `counter` 值一起保存。然而，更好的方案是对代码进行重新设计，不要依赖这种全局状态变量。

2. 像 `Dict` 和 `Set` 这种关联集合需要在 `__init__` 中 re-hash。Julia 在未来很可能会提供一个机制来注册初始化函数。

3. 依赖编译期的副作用会在加载时蔓延。例子包括：更改其它 Julia 模块里的数组或变量，操作文件或设备的句柄，保存指向其它系统资源（包括内存）的指针。

4. 无意中从其它模块中“拷贝”了全局状态：通过直接引用的方式而不是通过查找的方式。例如，在全局作用域下：

```
#mystdout = Base.stdout #= will not work correctly, since this will copy Base.stdout into this
↪  module =#
# instead use accessor functions:
getstdout() = Base.stdout #= best option =#
# or move the assignment into the runtime:
__init__() = global mystdout = Base.stdout #= also works =#
```

此处为预编译中的操作附加了若干限制，以帮助用户避免其他误操作：

1. 调用 `eval` 来在另一个模块中引发副作用。当增量预编译被标记时，该操作同时会导致抛出一个警告。

2. 当 `__init__()` 已经开始执行后，在局部作用域中声明 `global const`（见 issue #12010，计划为此情况添加一个错误提示）

3. 在增量预编译时替换模块是一个运行时错误。

一些其他需要注意的点：

1. 在源代码文件本身被修改之后，不会执行代码重载或缓存失效化处理（包括由 `Pkg.update` 执行的修改，此外在 `Pkg.rm` 执行后也没有清理操作）

2. 变形数组的内存共享特性会被预编译忽略（每个数组样貌都会获得一个拷贝）

3. 文件系统在编译期间和运行期间被假设为不变的，比如使用 `@__FILE__/source_path()` 在运行期间寻找资源、或使用 BinDeps 宏 `@checked_lib`。有时这是不可避免的。但是可能的话，在编译期将资源复制到模块里面是个好做法，这样在运行期间，就不需要去寻找它们了。

4. `WeakRef` 对象和完成器目前在序列化器中无法被恰当地处理（在接下来的发行版中将修复）。

5. 通常，最好避免去捕捉内部元数据对象的引用，如 `Method`、`MethodInstance`、`TypeMapLevel`、`TypeMapEntry` 及这些对象的字段，因为这会迷惑序列化器，且可能会引发你不想要的结果。此操作不足以成为一个错误，但你需做好准备：系统会尝试拷贝一部分，然后创建其余部分的单个独立实例。

在开发模块时，关闭增量预编译可能会有所帮助。命令行标记 `--compiled-modules={yes|no}` 可以让你切换预编译的开启和关闭。当 Julia 附加 `--compiled-modules=no` 启动，在载入模块和模块依赖时，编译缓存中的序列化模块会被忽略。More fine-grained control is available with `--pkgimages=no`, which suppresses only native-code storage during precompilation. `Base.compilecache` 仍可以被手动调用。此命令行标记的状态会被传递给 `Pkg.build`，禁止其在安装、更新、显式构建包时触发自动预编译。

You can also debug some precompilation failures with environment variables. Setting `JULIA_VERBOSE_LINKING=true` may help resolve failures in linking shared libraries of compiled native code. See the **Developer Documentation** part of the Julia manual, where you will find further details in the section documenting Julia's internals under ”Package Images”.

Chapter 17

文档

17.1 Accessing Documentation

Documentation can be accessed at the REPL or in IJulia by typing ? followed by the name of a function or macro, and pressing Enter. For example,

```
?cos
?@time
?r""
```

will show documentation for the relevant function, macro or string macro respectively. Most Julia environments provide a way to access documentation directly:

- VS Code shows documentation when you hover over a function name. You can also use the Julia panel in the sidebar to search for documentation.
- In Pluto, open the "Live Docs" panel on the bottom right.
- In Juno using Ctrl-J, Ctrl-D will show the documentation for the object under the cursor.

17.2 编写文档

Julia 允许开发者和用户，使用其内置的文档系统更加便捷地为函数、类型以及其他对象编写文档。

基础语法很简单：紧接在对象（函数，宏，类型和实例）之前的字符串都会被认为是对应对象的文档（称作 *docstrings*）。注意不要在 docstring 和文档对象之间有空行或者注释。这里有个基础的例子：

```
"Tell whether there are too foo items in the array."
foo(xs::Array) = ...
```

文档会被翻译成 Markdown，所以你可以使用缩进和代码块来分隔代码示例和文本。从技术上来说，任何对象都可以作为 metadata 与任何其他对象关联；Markdown 是默认的，但是可以创建其它字符串宏并传递给 @doc 宏来使用其他格式。

> **Note**
>
> Markdown 支持由 Markdown 标准库实现，有关支持语法的完整列表，请参阅其文档。

这里是一个更加复杂的例子，但仍然使用 Markdown：

```
"""
    bar(x[, y])

Compute the Bar index between `x` and `y`.

If `y` is unspecified, compute the Bar index between all pairs of columns of `x`.

# Examples
```julia-repl
julia> bar([1, 2], [1, 2])
1
```
"""
function bar(x, y) ...
```

如上例所示，我们推荐在写文档时遵守一些简单约定：

1. 始终在文档顶部显示函数的签名并带有四空格缩进，以便能够显示成 Julia 代码。

   这和在 Julia 代码中的签名是一样的（比如 `mean(x::AbstractArray)`），或是简化版。可选参数应该尽可能与默认值一同显示（例如 `f(x, y=1)`），这与实际的 Julia 语法一致。没有默认值的可选参数应该放在括号中（例如 `f(x[, y])` 和 `f(x[, y[, z]])`）。可选的解决方法是使用多行：一个没有可选参数，其他的拥有可选参数（或者多个可选参数）。这个解决方案也可以用作给某个函数的多个方法来写文档。当一个函数接收到多个关键字参数，只在签名中包含占位符 `<keyword arguments>`（例如 `f(x; <keyword arguments>)`），并在 `# Arguments` 章节给出完整列表（参照下列第 4 点）。

2. 在简化的签名块后请包含一个描述函数能做什么或者对象代表什么的单行句。如果需要的话，在一个空行之后，在第二段提供更详细的信息。

   撰写函数的文档时，单行语句应使用祈使结构（比如「Do this」、「Return that」）而非第三人称（不要写「Returns the length...」）。并且应以句号结尾。如果函数的意义不能简单地总结，更好的方法是分成分开的组合句（虽然这不应被看做是对于每种情况下的绝对要求）。

3. 不要自我重复。

   因为签名给出了函数名，所以没有必要用「The function bar...」开始文档：直接说要点。类似地，如果签名指定了参数的类型，在描述中提到这些是多余的。

4. 只在确实必要时提供参数列表。

   对于简单函数，直接在函数目的的描述中提到参数的作用常常更加清楚。参数列表只会重复再其他地方提供过的信息。但是，对于拥有多个参数的（特别是含有关键字参数的）复杂函数来说，提供一个参数列表是个好主意。在这种情况下，请在函数的一般描述之后、标题 `# Arguments` 之下插入参数列表，并在每个参数前加个着重号 `-`。参数列表应该提到参数的类型和默认值（如果有）：

   ```
   """
   ...
   # Arguments
   - `n::Integer`: the number of elements to compute.
   - `dim::Integer=1`: the dimensions along which to perform the computation.
   ...
   """
   ```

5. 给相关函数提供提示。

   有时会存在具有功能相联系的函数。为了更易于发现相关函数，请在段落 `See also` 中为其提供一个小列表。

```
See also [`bar!`](@ref), [`baz`](@ref), [`baaz`](@ref).
```

6. 请在 `# Examples` 中包含一些代码例子。

   例子应尽可能按照 *doctest* 来写。*doctest* 是一个栅栏分隔开的代码块（请参阅代码块），其以 ```` ```jldoctest ```` 开头并包含任意数量的提示符 `julia>` 以及用来模拟 Julia REPL 的输入和预期输出。

   Note

   Doctest 由 `Documenter.jl` 支持。有关更详细的文档，请参阅 Documenter 的手册。

   例如在下面的 docstring 中定义了变量 `a`，预期的输出，跟在 Julia REPL 中打印的一样，出现在后面。

````
"""
Some nice documentation here.

# Examples
```jldoctest
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```
"""
````

   Warning

   Calling `rand` and other RNG-related functions should be avoided in doctests since they will not produce consistent outputs during different Julia sessions. If you would like to show some random number generation related functionality, one option is to explicitly construct and seed your own RNG object (see `Random`) and pass it to the functions you are doctesting.

   Operating system word size (`Int32` or `Int64`) as well as path separator differences (/ or \) will also affect the reproducibility of some doctests.

   Note that whitespace in your doctest is significant! The doctest will fail if you misalign the output of pretty-printing an array, for example.

   你可以运行 `make -C doc doctest=true` 来运行在 Julia 手册和 API 文档中的 doctests，这样可以确保你的例子都能正常运行。

   为了表示输出结果被截断了，你应该在校验应该停止的一行写上 `[...]`。这个在当 doctest 显示有个异常被抛出时隐藏堆栈跟踪时很有用（堆栈跟踪包含对 julia 代码的行的非永久引用），例如：

```
```jldoctest
julia> div(1, 0)
ERROR: DivideError: integer division error
```
```

```
[...]
```
```

那些不能进行测试的例子应该写在以 ```julia 开头的栅栏分隔的代码块中，以便在生成的文档中正确地高亮显示。

**Tip**

例子应尽可能**独立**和**可运行**以便读者可以在不需要引入任何依赖的情况下对它们进行实验。

7. 使用倒引号来标识代码和方程。

   Julia 标识符和代码摘录应该出现在倒引号 ` 之间来使其能高亮显示。LaTeX 语法下的方程应该插入到双倒引号 `` 之间。请使用 Unicode 字符而非 LaTeX 转义序列，比如 ``α = 1`` 而非 ``\\alpha = 1``。

8. 请将起始和结束的""" 符号单独成行。

   也就是说，请写：

```
"""
...

...
"""
f(x, y) = ...
```

   而非：

```
"""...

..."""
f(x, y) = ...
```

   这将让 docstring 的起始和结束位置更加清楚。

9. 请在代码中遵守单行长度限制。

   Docstring 是使用与代码相同的工具编辑的。所以应运用同样的约定。建议一行 92 个字符后换行。

10. 请在 # Implementation 章节中提供自定义类型如何实现该函数的信息。这些实现细节是针对开发者而非用户的，解释了例如哪些函数应该被重写、哪些函数自动使用恰当的回退函数等信息，最好与描述函数的主体描述分开。

11. 对于长文档字符串，可以考虑使用 # Extended help 头拆分文档。典型的帮助模式将只显示标题上方的内容；你可以通过添加一个 ? 在表达的开头来查看完整的文档（即 ??foo 而不是 ?foo）。
```

17.3 函数与方法

在 Julia 中函数可能有多种实现，被称为方法。虽然通用函数一般只有一个目的，Julia 允许在必要时可以对方法独立写文档。通常，应该只有最通用的方法才有文档，或者甚至只是函数本身（也就是在 `function bar end` 之前没有任何方法的对象）。特定方法应该只因为其行为与其他通用方法有所区别才写文档。在任何情况下都不应重复其他地方有的信息。例如

```
"""
    *(x, y, z...)

Multiplication operator. `x * y * z *...` calls this function with multiple
arguments, i.e. `*(x, y, z...)`.
"""
function *(x, y, z...)
    # ... [implementation sold separately] ...
end

"""
    *(x::AbstractString, y::AbstractString, z::AbstractString...)

When applied to strings, concatenates them.
"""
function *(x::AbstractString, y::AbstractString, z::AbstractString...)
    # ... [insert secret sauce here] ...
end

help?> *
search: * .*

  *(x, y, z...)

  Multiplication operator. x * y * z *... calls this function with multiple
  arguments, i.e. *(x,y,z...).

  *(x::AbstractString, y::AbstractString, z::AbstractString...)

  When applied to strings, concatenates them.
```

当从通用函数里抽取文档时，每个方法的元数据会用函数 `catdoc` 拼接，其当然可以被自定义类型重写。

17.4 进阶用法

`@doc` 宏将它的第一个参数与它的第二个参数关联在各个模块的名为 `META` 的字典中。

为了让写文档更加简单，语法分析器对宏名 `@doc` 特殊对待：如果 `@doc` 的调用只有一个参数，但是在下一行出现了另外一个表达式，那么这个表达式就会追加为宏的参数。所以接下来的语法会被分析成 `@doc` 的 2 个参数的调用：

```
@doc raw"""
...
"""
f(x) = x
```

这就让使用任意对象（这里指的是原始字符串 raw""）作为 docstring 变得简单。

当 @doc 宏（或者 doc 函数）用作抽取文档时，他会在所有的 META 字典寻找与对象相关的元数据并且返回。返回的对象（例如一些 Markdown 内容）会默认智能地显示。这个设计也让以编程方法使用文档系统变得容易；例如，在一个函数的不同版本中重用文档：

```
@doc "..." foo!
@doc (@doc foo!) foo
```

或者与 Julia 的元编程功能一起使用：

```
for (f, op) in ((:add, :+), (:subtract, :-), (:multiply, :*), (:divide, :/))
    @eval begin
        $f(a,b) = $op(a,b)
    end
end
@doc "`add(a,b)` adds `a` and `b` together" add
@doc "`subtract(a,b)` subtracts `b` from `a`" subtract
```

写在非顶级块，比如 begin, if, for, 和 let，中的文档可以通过 @doc 宏加入文档系统中。例如：

```
if condition()
    @doc "..."
    f(x) = x
end
```

会被加到 f(x) 的文档中，当 condition() 是 true 的时候。注意即使 f(x) 在块的末尾离开了作用域，他的文档还会保留。

可以利用元编程来帮助创建文档。当在文档字符串中使用字符串插值时，需要使用额外的 $ 例如：$($name)

```
for func in (:day, :dayofmonth)
    name = string(func)
    @eval begin
        @doc """
            $($name)(dt::TimeType) -> Int64

        The day of month of a `Date` or `DateTime` as an `Int64`.
        """ $func(dt::Dates.TimeType)
    end
end
```

动态写文档

有些时候类型的实例的合适的文档并非只取决于类型本身，也取决于实例的值。在这些情况下，你可以添加一个方法给自定义类型的 Docs.getdoc 函数，返回基于每个实例的文档。例如，

```
struct MyType
    value::Int
end
```

```
Docs.getdoc(t::MyType) = "Documentation for MyType with value $(t.value)"

x = MyType(1)
y = MyType(2)
```

?x will display ”Documentation for MyType with value 1” while ?y will display ”Documentation for MyType with value 2”.

17.5 语法指南

本指南提供了如何将文档附加到所有可能的 Julia 语法构造的全面概述。

在下述例子中"..." 用来表示任意的 docstring。

$ 与 \ 字符

$ 和 \ 字符仍然被解析为字符串插值或转义序列的开始字符。raw"" 字符串宏和 @doc 宏可以用来避免对它们进行转义。当文档字符串包含 LaTeX 或 Julia 源代码，且示例中包含插值时，这是很方便的:

````
@doc raw"""
```math
\LaTeX
```
"""
function f end
````

函数与方法

```
"..."
function f end

"..."
f
```

把 docstring "..." 添加给了函数 f。首选的语法是第一种，虽然两者是等价的。

```
"..."
f(x) = x

"..."
function f(x)
    x
end

"..."
f(x)
```

把 docstring "..." 添加给了方法 f(::Any)。

```
"..."
f(x, y = 1) = x + y
```

把 docstring "..." 添加给了两个方法，分别为 f(::Any) 和 f(::Any, ::Any)。

宏

```
"..."
macro m(x) end
```

把 docstring "..." 添加给了宏 @m(::Any) 的定义。

```
"..."
:(@m)
```

把 docstring "..." 添加给了名为 @m 的宏。

类型

```
"..."
abstract type T1 end

"..."
mutable struct T2
    ...
end

"..."
struct T3
    ...
end
```

把 docstring "..." 添加给了类型 T1、T2 和 T3。

```
"..."
struct T
    "x"
    x
    "y"
    y
end
```

把 docstring "..." 添加给了类型 T，"x" 添加给字段 T.x，"y" 添加给字段 T.y。也可以运用于 mutable struct 类型。

模块

```
"..."
module M end

module M

"..."
M

end
```

把 docstring "..." 添加给了模块 M。首选的语法是在模块之前添加 docstring，虽然两者是等价的。

```
"..."
baremodule M
# ...
end

baremodule M

import Base: @doc

"..."
f(x) = x

end
```

通过在表达式上方放置一个 docstring 来记录 baremodule 会自动将 @doc 导入到模块中。当没有记录模块表达式时，必须手动完成这些导入。

全局变量

```
"..."
const a = 1

"..."
b = 2

"..."
global c = 3
```

把 docstring"..." 添加给了绑定 a，b 和 c。

绑定是用来在模块中存储对于特定符号的引用而非存储被引用的值本身。

Note

当一个 const 定义只是用作定义另外一个定义的别名时，比如函数 div 和其在 Base 中的别名 ÷，并不要为别名写文档，转而去为实际的函数写文档。

如果别名写了文档而实际定义没有，那么文档系统（? 模式）在寻找实际定义的文档时将不会返回别名的对应文档。

比如你应该写

```
"..."
f(x) = x + 1
const alias = f
```

而非

```
f(x) = x + 1
"..."
const alias = f
```

```
"..."
sym
```

把 docstring "..." 添加给值 sym。但是应首选在 sym 的定义处写文档。

多重对象

```
"..."
a, b
```

把 docstring "..." 添加给 a 和 b，两个都应该是可以写文档的表达式。这个语法等价于

```
"..."
a

"..."
b
```

这种方法可以给任意数量的表达式写文档。当两个函数相关，比如非变版本 f 和可变版本 f!，这个语法是有用的。

宏生成代码

```
"..."
@m expression
```

把 docstring `"..."` 添加给通过展开 `@m expression` 生成的表达式。这就允许由 `@inline`、`@noinline`、`@generated` 或者任意其他宏装饰的表达式，能和没有装饰的表达式以同样的方式写文档。

宏作者应该注意到只有只生成单个表达式的宏才会自动支持 docstring。如果宏返回的是含有多个子表达式的块，需要写文档的子表达式应该使用宏 `@__doc__` 标记。

`@enum` 宏使用了 `@__doc__` 来允许给 `Enum` 写文档。它的做法可以作为如何正确使用 `@__doc__` 的范例。

`Core.@__doc__` – Macro.

```
@__doc__(ex)
```

Low-level macro used to mark expressions returned by a macro that should be documented. If more than one expression is marked then the same docstring is applied to each expression.

```
macro example(f)
    quote
        $(f)() = 0
        @__doc__ $(f)(x) = 1
        $(f)(x, y) = 2
    end |> esc
end
```

`@__doc__` has no effect when a macro that uses it is not documented.

source

Chapter 18

元编程

Lisp 留给 Julia 最大的遗产就是它的元编程支持。和 Lisp 一样，Julia 把自己的代码表示为语言中的数据结构。既然代码被表示为了可以在语言中创建和操作的对象，程序就可以变换和生成自己的代码。这允许在没有额外构建步骤的情况下生成复杂的代码，并且还允许在 abstract syntax trees 级别上运行的真正的 Lisp 风格的宏。与之相对的是预处理器“宏”系统，比如 C 和 C++ 中的，它们在解析和解释代码之前进行文本操作和变换。由于 Julia 中的所有数据类型和代码都被表示为 Julia 的数据结构，强大的 reflection 功能可用于探索程序的内部及其类型，就像任何其他数据一样。

Warning

Metaprogramming is a powerful tool, but it introduces complexity that can make code more difficult to understand. For example, it can be surprisingly hard to get scope rules correct. Metaprogramming should typically be used only when other approaches such as higher order functions and closures cannot be applied.

`eval` and defining new macros should be typically used as a last resort. It is almost never a good idea to use `Meta.parse` or convert an arbitrary string into Julia code. For manipulating Julia code, use the `Expr` data structure directly to avoid the complexity of how Julia syntax is parsed.

The best uses of metaprogramming often implement most of their functionality in runtime helper functions, striving to minimize the amount of code they generate.

18.1 程序表示

每个 Julia 程序均以字符串开始：

```
julia> prog = "1 + 1"
"1 + 1"
```

接下来会发生什么？

下一步是 parse 每个字符串到一个称为表达式的对象，由 Julia 的类型 `Expr` 表示：

```
julia> ex1 = Meta.parse(prog)
:(1 + 1)

julia> typeof(ex1)
Expr
```

Expr 对象包含两个部分：

- 一个标识表达式类型的 Symbol。

Symbol 就是一个 interned string 标识符（下面会有更多讨论）

```
julia> ex1.head
:call
```

- 表达式的参数，可能是符号、其他表达式或字面量：

```
julia> ex1.args
3-element Vector{Any}:
  :+
 1
 1
```

表达式也可能直接用 prefix notation 构造：

```
julia> ex2 = Expr(:call, :+, 1, 1)
:(1 + 1)
```

上面构造的两个表达式–一个通过解析构造一个通过直接构造– 是等价的：

```
julia> ex1 == ex2
true
```

这里的关键点是 Julia 的代码在内部表示为可以从语言本身访问的数据结构

函数 dump 可以带有缩进和注释地显示 Expr 对象：

```
julia> dump(ex2)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 1
```

Expr 对象也可以嵌套：

```
julia> ex3 = Meta.parse("(4 + 4) / 2")
:((4 + 4) / 2)
```

另外一个查看表达式的方法是使用 Meta.show_sexpr，它能显示给定 Expr 的 S-expression，对 Lisp 用户来说，这看着很熟悉。下面是一个示例，阐释了如何显示嵌套的 Expr：

```
julia> Meta.show_sexpr(ex3)
(:call, :/, (:call, :+, 4, 4), 2)
```

符号

字符：在 Julia 中有两个作用 The first form creates a Symbol, an interned string used as one building-block of expressions, from valid identifiers:

```
julia> s = :foo
:foo

julia> typeof(s)
Symbol
```

构造函数 Symbol 接受任意数量的参数并通过把它们的字符串表示连在一起创建一个新的符号:

```
julia> :foo === Symbol("foo")
true

julia> Symbol("1foo") # `:1foo` would not work, as `1foo` is not a valid identifier
Symbol("1foo")

julia> Symbol("func",10)
:func10

julia> Symbol(:var,'_',"sym")
:var_sym
```

在表达式的上下文中，符号用来表示对变量的访问；当一个表达式被求值时，符号会被替换为这个符号在合适的 scope 中所绑定的值。

有时需要在：的参数两边加上额外的括号，以避免在解析时出现歧义:

```
julia> :(:)
:(:)

julia> :(::)
:(::)
```

18.2 表达式与求值

引用

：的第二个语义是不显式调用 Expr 构造器来创建表达式对象。这被称为引用。：后面跟着包围着单个 Julia 语句括号，可以基于被包围的代码生成一个 Expr 对象。下面是一个引用算数表达式的例子:

```
julia> ex = :(a+b*c+1)
:(a + b * c + 1)

julia> typeof(ex)
Expr
```

（为了查看这个表达式的结构，可以试一试 ex.head 和 ex.args，或者使用 dump 同时查看 ex.head 和 ex.args 或者 Meta.@dump）

注意等价的表达式也可以使用 Meta.parse 或者直接用 Expr 构造：

```
julia>      :(a + b*c + 1)       ==
       Meta.parse("a + b*c + 1") ==
       Expr(:call, :+, :a, Expr(:call, :*, :b, :c), 1)
true
```

解析器提供的表达式通常只有符号、其它表达式和字面量值作为其参数，而由 Julia 代码构造的表达式能以非字面量形式的任意运行期值作为其参数。在此特例中，+ 和 a 都是符号，*(b,c) 是子表达式，而 1 是 64 位带符号整数字面量。

引用多个表达式有第二种语法形式：在 quote ... end 中包含代码块。

```
julia> ex = quote
           x = 1
           y = 2
           x + y
       end
quote
    #= none:2 =#
    x = 1
    #= none:3 =#
    y = 2
    #= none:4 =#
    x + y
end

julia> typeof(ex)
Expr
```

插值

使用值参数直接构造 Expr 对象虽然很强大，但与「通常的」Julia 语法相比，Expr 构造函数可能让人觉得乏味。作为替代方法，Julia 允许将字面量或表达式插入到被引用的表达式中。表达式插值由前缀 $ 表示。

在此示例中，插入了变量 a 的值：

```
julia> a = 1;

julia> ex = :($a + b)
:(1 + b)
```

对未被引用的表达式进行插值是不支持的，这会导致编译期错误：

```
julia> $a + b
ERROR: syntax: "$" expression outside quote
```

在此示例中，元组 (1,2,3) 作为表达式插入到条件测试中：

```
julia> ex = :(a in $:((1,2,3)) )
:(a in (1, 2, 3))
```

在表达式插值中使用 $ 是有意让人联想到字符串插值和命令插值。表达式插值使得复杂 Julia 表达式的程序化构造变得方便和易读。

Splatting 插值

请注意，$ 插值语法只允许插入单个表达式到包含它的表达式中。有时，你手头有个由表达式组成的数组，需要它们都变成其所处表达式的参数，而这可通过 $(xs...) 语法做到。例如，下面的代码生成了一个函数调用，其参数数量通过编程确定：

```
julia> args = [:x, :y, :z];

julia> :(f(1, $(args...)))
:(f(1, x, y, z))
```

嵌套引用

自然地，引用表达式可以包含在其它引用表达式中。插值在这些情形中的工作方式可能会有点难以理解。考虑这个例子：

```
julia> x = :(1 + 2);

julia> e = quote quote $x end end
quote
    #= none:1 =#
    $(Expr(:quote, quote
    #= none:1 =#
    $(Expr(:$, :x))
end))
end
```

Notice that the result contains $x, which means that x has not been evaluated yet. In other words, the $ expression "belongs to" the inner quote expression, and so its argument is only evaluated when the inner quote expression is:

```
julia> eval(e)
quote
    #= none:1 =#
    1 + 2
end
```

但是，外部 quote 表达式可以把值插入到内部引用表达式的 $ 中去。这通过多个 $ 实现：

```
julia> e = quote quote $$x end end
quote
    #= none:1 =#
    $(Expr(:quote, quote
    #= none:1 =#
```

```
    $(Expr(:$, :(1 + 2)))
end))
end
```

Notice that (1 + 2) now appears in the result instead of the symbol x. Evaluating this expression yields an interpolated 3:

```
julia> eval(e)
quote
    #= none:1 =#
    3
end
```

这种行为背后的直觉是每个 $ 都将 x 求值一遍：一个 $ 工作方式类似于 eval(:x)，其返回 x 的值，而两个 $ 行为相当于 eval(eval(:x))。

QuoteNode

quote 形式在 AST 中通常表示为一个 head 为 :quote 的 Expr：

```
julia> dump(Meta.parse(":(1+2)"))
Expr
  head: Symbol quote
  args: Array{Any}((1,))
    1: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol +
        2: Int64 1
        3: Int64 2
```

正如我们所看到的，这些表达式支持插值符号 $。但是，在某些情况下，需要在不执行插值的情况下引用代码。这种引用还没有语法，但在内部表示为 QuoteNode 类型的对象：

```
julia> eval(Meta.quot(Expr(:$, :(1+2))))
3

julia> eval(QuoteNode(Expr(:$, :(1+2))))
:($(Expr(:$, :(1 + 2))))
```

解析器为简单的引用项（如符号）生成 QuoteNode：

```
julia> dump(Meta.parse(":x"))
QuoteNode
  value: Symbol x
```

QuoteNode 也可用于某些高级的元编程任务。

表达式求值

给定一个表达式对象，可以使用 eval 使 Julia 在全局作用域内评估（执行）它：

```
julia> ex1 = :(1 + 2)
:(1 + 2)

julia> eval(ex1)
3

julia> ex = :(a + b)
:(a + b)

julia> eval(ex)
ERROR: UndefVarError: `b` not defined
[...]

julia> a = 1; b = 2;

julia> eval(ex)
3
```

每个模块有自己的 eval 函数，该函数在其全局作用域内对表达式求值。传给 eval 的表达式不止可以返回值——它们还能具有改变封闭模块的环境状态的副作用：

```
julia> ex = :(x = 1)
:(x = 1)

julia> x
ERROR: UndefVarError: `x` not defined

julia> eval(ex)
1

julia> x
1
```

这里，表达式对象的求值导致一个值被赋值给全局变量 x。

由于表达式只是 Expr 对象，而其可以通过编程方式构造然后对它求值，因此可以动态地生成任意代码，然后使用 eval 运行所生成的代码。这是个简单的例子：

```
julia> a = 1;

julia> ex = Expr(:call, :+, a, :b)
:(1 + b)

julia> a = 0; b = 2;

julia> eval(ex)
3
```

a 的值被用于构造表达式 ex，该表达式将函数 + 作用于值 1 和变量 b。请注意 a 和 b 使用方式间的重要区别：

- 变量 a 在表达式构造时的值在表达式中用作立即值。因此，在对表达式求值时，a 的值就无关紧要了：表达式中的值已经是 1，与 a 的值无关。
- 另一方面，因为在表达式构造时用的是符号 :b，所以变量 b 的值无关紧要——:b 只是一个符号，变量 b 甚至无需被定义。然而，在表达式求值时，符号 :b 的值通过寻找变量 b 的值来解析。

关于表达式的函数

如上所述，Julia 能在其内部生成和操作 Julia 代码，这是个非常有用的功能。我们已经见过返回 Expr 对象的函数例子：Meta.parse 函数，它接受字符串形式的 Julia 代码并返回相应的 Expr。函数也可以接受一个或多个 Expr 对象作为参数，并返回另一个 Expr。这是个简单、提神的例子：

```
julia> function math_expr(op, op1, op2)
           expr = Expr(:call, op, op1, op2)
           return expr
       end
math_expr (generic function with 1 method)

julia>  ex = math_expr(:+, 1, Expr(:call, :*, 4, 5))
:(1 + 4 * 5)

julia> eval(ex)
21
```

作为另一个例子，这个函数将数值参数加倍，但不处理表达式：

```
julia> function make_expr2(op, opr1, opr2)
           opr1f, opr2f = map(x -> isa(x, Number) ? 2*x : x, (opr1, opr2))
           retexpr = Expr(:call, op, opr1f, opr2f)
           return retexpr
       end
make_expr2 (generic function with 1 method)

julia> make_expr2(:+, 1, 2)
:(2 + 4)

julia> ex = make_expr2(:+, 1, Expr(:call, :*, 5, 8))
:(2 + 5 * 8)

julia> eval(ex)
42
```

18.3 宏

宏提供了一种机制，可以将生成的代码包含在程序的最终主体中。宏将一组参数映射到返回的 表达式，并且生成的表达式被直接编译，而不需要运行时 eval 调用。宏参数可能包括表达式、字面量和符号。

基础

这是一个非常简单的宏：

```
julia> macro sayhello()
           return :( println("Hello, world!") )
       end
@sayhello (macro with 1 method)
```

宏在 Julia 的语法中有一个专门的字符 @ (at-sign)，紧接着是其使用 `macro NAME ... end` 形式来声明的唯一的宏名。在这个例子中，编译器会把所有的 `@sayhello` 替换成：

```
:( println("Hello, world!") )
```

当 `@sayhello` 在 REPL 中被输入时，解释器立即执行，因此我们只会看到计算后的结果：

```
julia> @sayhello()
Hello, world!
```

现在，考虑一个稍微复杂一点的宏：

```
julia> macro sayhello(name)
 return :( println("Hello, ", $name) )
 end
@sayhello (macro with 1 method)
```

这个宏接受一个参数 `name`。当遇到 `@sayhello` 时，quoted 表达式会被展开并将参数中的值插入到最终的表达式中：

```
julia> @sayhello("human")
Hello, human
```

我们可以使用函数 `macroexpand` 查看引用的返回表达式（**重要提示**：这是一个非常有用的调试宏的工具）：

```
julia> ex = macroexpand(Main, :(@sayhello("human")) )
:(Main.println("Hello, ", "human"))

julia> typeof(ex)
Expr
```

我们可以看到 `"human"` 字面量已被插入到表达式中了。

还有一个宏 `@macroexpand`，它可能比 `macroexpand` 函数更方便：

```
julia> @macroexpand @sayhello "human"
:(println("Hello, ", "human"))
```

停：为什么需要宏？

我们已经在上一节中看到了一个函数 `f(::Expr...) -> Expr`。其实`macroexpand`也是这样一个函数。那么，为什么会要设计宏呢？

宏是必需的，因为它们在解析代码时执行，因此，宏允许程序员在运行完整程序之前生成定制代码的片段。为了说明差异，请考虑以下示例：

```
julia> macro twostep(arg)
           println("I execute at parse time. The argument is: ", arg)
           return :(println("I execute at runtime. The argument is: ", $arg))
       end
@twostep (macro with 1 method)

julia> ex = macroexpand(Main, :(@twostep :(1, 2, 3)) );
I execute at parse time. The argument is: :((1, 2, 3))
```

第一个 `println` 调用在调用 `macroexpand` 时执行。生成的表达式只包含第二个 `println`：

```
julia> typeof(ex)
Expr

julia> ex
:(println("I execute at runtime. The argument is: ", $(Expr(:copyast, :($(QuoteNode(:((1, 2,
↪  3)))))))))

julia> eval(ex)
I execute at runtime. The argument is: (1, 2, 3)
```

宏的调用

宏的通常调用语法如下：

```
@name expr1 expr2 ...
@name(expr1, expr2, ...)
```

请注意，在宏名称前的标志 `@`，且在第一种形式中参数表达式间没有逗号，而在第二种形式中 `@name` 后没有空格。这两种风格不应混淆。例如，下列语法不同于上述例子；它把元组 `(expr1, expr2, ...)` 作为参数传给宏：

```
@name (expr1, expr2, ...)
```

在数组字面量（或推导式）上调用宏的另一种方法是不使用括号直接并列两者。在这种情况下，数组将是唯一的传给宏的表达式。以下语法等价（且与 `@name [a b] * v` 不同）：

```
@name[a b] * v
@name([a b]) * v
```

在这着重强调，宏把它们的参数作为表达式、字面量或符号接收。浏览宏参数的一种方法是在宏的内部调用 `show` 函数：

```
julia> macro showarg(x)
           show(x)
           # ... remainder of macro, returning an expression
       end
@showarg (macro with 1 method)

julia> @showarg(a)
:a

julia> @showarg(1+1)
:(1 + 1)

julia> @showarg(println("Yo!"))
:(println("Yo!"))
```

除了给定的参数列表，每个宏都会传递名为 `__source__` 和 `__module__` 的额外参数。

参数 `__source__` 提供 `@` 符号在宏调用处的解析器位置的相关信息（以 `LineNumberNode` 对象的形式）。这使得宏能包含更好的错误诊断信息，其通常用于日志记录、字符串解析器宏和文档，比如，用于实现 `@__LINE__`、`@__FILE__` 和 `@__DIR__` 宏。

引用 `__source__.line` 和 `__source__.file` 即可访问位置信息：

```
julia> macro __LOCATION__(); return QuoteNode(__source__); end
@__LOCATION__ (macro with 1 method)

julia> dump(
            @__LOCATION__(
       ))
LineNumberNode
  line: Int64 2
  file: Symbol none
```

参数 `__module__` 提供宏调用展开处的上下文相关信息（以 `Module` 对象的形式）。这允许宏查找上下文相关的信息，比如现有的绑定，或者将值作为附加参数插入到一个在当前模块中进行自我反射的运行时函数调用中。

构建高级的宏

这是 Julia 的 `@assert` 宏的简化定义：

```
julia> macro assert(ex)
           return :( $ex ? nothing : throw(AssertionError($(string(ex)))) )
       end
@assert (macro with 1 method)
```

这个宏可以像这样使用：

```
julia> @assert 1 == 1.0

julia> @assert 1 == 0
ERROR: AssertionError: 1 == 0
```

宏调用在解析时扩展为其返回结果，并替代已编写的语法。这相当于编写：

```
1 == 1.0 ? nothing : throw(AssertionError("1 == 1.0"))
1 == 0 ? nothing : throw(AssertionError("1 == 0"))
```

也就是说，在第一个调用中，表达式 `:(1 == 1.0)` 拼接到测试条件槽中，而 `string(:(1 == 1.0))` 拼接到断言信息槽中。如此构造的表达式会被放置在发生 `@assert` 宏调用处的语法树。然后在执行时，如果测试表达式的计算结果为真，则返回 `nothing`，但如果测试结果为假，则会引发错误，表明声明的表达式为假。请注意，将其编写为函数是不可能的，因为能获取的只有条件的值而无法在错误信息中显示计算出它的表达式。

在 Julia Base 中，`@assert` 的实际定义更复杂。它允许用户可选地制定自己的错误信息，而不仅仅是打印断言失败的表达式。与函数一样，具有可变数量的参数（变参函数）可在最后一个参数后面用省略号指定：

```
julia> macro assert(ex, msgs...)
           msg_body = isempty(msgs) ? ex : msgs[1]
           msg = string(msg_body)
           return :($ex ? nothing : throw(AssertionError($msg)))
       end
@assert (macro with 1 method)
```

现在 `@assert` 有两种操作模式，这取决于它接收到的参数数量！如果只有一个参数，`msgs` 捕获的表达式元组将为空，并且其行为与上面更简单的定义相同。但是现在如果用户指定了第二个参数，它会打印在消息正文中而不是不相等的表达式中。你可以使用恰当命名的 `@macroexpand` 宏检查宏展开的结果：

```
julia> @macroexpand @assert a == b
:(if Main.a == Main.b
        Main.nothing
    else
        Main.throw(Main.AssertionError("a == b"))
    end)

julia> @macroexpand @assert a==b "a should equal b!"
:(if Main.a == Main.b
        Main.nothing
    else
        Main.throw(Main.AssertionError("a should equal b!"))
    end)
```

实际的 `@assert` 宏还处理了另一种情形：我们如果除了打印「a should equal b」外还想打印它们的值？有人也许会天真地尝试在自定义消息中使用字符串插值，例如，`@assert a==b "a ($a) should equal b ($b)!"`，但这不会像上面的宏一样按预期工作。你能想到为什么吗？回想一下字符串插值，内插字符串会被重写为 `string` 的调用。比较：

```
julia> typeof(:("a should equal b"))
String

julia> typeof(:("a ($a) should equal b ($b)!"))
Expr
```

```
julia> dump(:("a ($a) should equal b ($b)!"))
Expr
  head: Symbol string
  args: Array{Any}((5,))
    1: String "a ("
    2: Symbol a
    3: String ") should equal b ("
    4: Symbol b
    5: String ")!"
```

所以，现在宏在 msg_body 中获得的不是单纯的字符串，其接收了一个完整的表达式，该表达式需进行求值才能按预期显示。这可作为 string 调用的参数直接拼接到返回的表达式中；有关完整实现，请参阅 error.jl。

@assert 宏充分利用拼接被引用的表达式，以便简化对宏内部表达式的操作。

卫生宏

在更复杂的宏中会出现关于卫生宏 的问题。简而言之，宏必须确保在其返回表达式中引入的变量不会意外地与其展开处周围代码中的现有变量相冲突。相反，作为参数传递给宏的表达式通常被认为在其周围代码的上下文中进行求值，与现有变量交互并修改之。另一个问题源于这样的事实：宏可以在不同于其定义所处模块的模块中调用。在这种情况下，我们需要确保所有全局变量都被解析到正确的模块中。Julia 比使用文本宏展开的语言（比如 C）具有更大的优势，因为它只需要考虑返回的表达式。所有其它变量（例如上面 @assert 中的 msg）遵循通常的作用域块规则。

为了演示这些问题，让我们来编写宏 @time，其以表达式为参数，记录当前时间，对表达式求值，再次记录当前时间，打印前后的时间差，然后以表达式的值作为其最终值。该宏可能看起来就像这样：

```
macro time(ex)
    return quote
        local t0 = time_ns()
        local val = $ex
        local t1 = time_ns()
        println("elapsed time: ", (t1-t0)/1e9, " seconds")
        val
    end
end
```

在这里，我们希望 t0、t1 和 val 成为私有临时变量，并且我们希望 time_ns 引用 Julia Base 中的 time_ns 函数，而不是任何用户可能拥有的 time_ns 变量（同样适用于 println）。想象一下，如果用户表达式 ex 还包含对名为 t0 的变量的赋值，或者定义了自己的 time_ns 变量，可能会出现什么问题。程序可能会报错，或者进行未知的行为。

Julia 的宏展开器通过以下方式解决了这些问题。首先，宏结果中的变量分为局部变量或全局变量。如果变量被赋值（并且不是声明为全局的）、声明为局部、或用作函数参数名称，则该变量被视为局部变量。否则，它被认为是全局的。然后将局部变量重命名为唯一的（使用 gensym 函数，该函数生成新符号），并在宏定义环境中解析全局变量。因此，上述两个问题都得到了处理；宏的局部变量不会与任何用户变量冲突，并且 time_ns 和 println 将引用 Julia Base 定义。

然而，仍有另外的问题。考虑此宏的以下用法：

```
module MyModule
import Base.@time

time_ns() = ... # compute something

@time time_ns()
end
```

这里的用户表达式 ex 是对 time_ns 的调用，但与宏使用的 time_ns 函数不同。它清楚地指向 MyModule.time_ns。因此我们必须安排在宏调用环境中解析 ex 中的代码。这是通过使用 esc "转义" 表达式来完成的：

```
macro time(ex)
    ...
    local val = $(esc(ex))
    ...
end
```

以这种方式封装的表达式会被宏展开器单独保留，并将其简单地逐字粘贴到输出中。因此，它将在宏调用所处环境中解析。

这种转义机制可以在必要时用于「违反」卫生，以便于引入或操作用户变量。例如，以下宏在其调用所处环境中将 x 设置为零：

```
julia> macro zerox()
           return esc(:(x = 0))
       end
@zerox (macro with 1 method)

julia> function foo()
           x = 1
           @zerox
           return x # is zero
       end
foo (generic function with 1 method)

julia> foo()
0
```

应当明智地使用这种变量操作，但它偶尔会很方便。

获得正确的规则也许是个艰巨的挑战。在使用宏之前，你可以去考虑是否函数闭包便已足够。另一个有用的策略是将尽可能多的工作推迟到运行时。例如，许多宏只是将其参数封装为 QuoteNode 或类似的 Expr。这方面的例子有 @task body，它只返回 schedule(Task(() -> $body))，和 @eval expr，它只返回 eval(QuoteNode(expr))。

为了演示，我们可以将上面的 @time 示例重新编写成：

```
macro time(expr)
    return :(timeit(() -> $(esc(expr))))
end
function timeit(f)
```

```
    t0 = time_ns()
    val = f()
    t1 = time_ns()
    println("elapsed time: ", (t1-t0)/1e9, " seconds")
    return val
end
```

但是，我们不这样做也是有充分理由的：将 `expr` 封装在新的作用域块（该匿名函数）中也会稍微改变该表达式的含义（其中任何变量的作用域），而我们想要 `@time` 使用时对其封装的代码影响最小。

宏与派发

与 Julia 函数一样，宏也是泛型的。由于多重派发，这意味着宏也能有多个方法定义：

```
julia> macro m end
@m (macro with 0 methods)

julia> macro m(args...)
           println("$(length(args)) arguments")
       end
@m (macro with 1 method)

julia> macro m(x,y)
           println("Two arguments")
       end
@m (macro with 2 methods)

julia> @m "asd"
1 arguments

julia> @m 1 2
Two arguments
```

但是应该记住，宏派发基于传递给宏的 AST 的类型，而不是 AST 在运行时进行求值的类型：

```
julia> macro m(::Int)
           println("An Integer")
       end
@m (macro with 3 methods)

julia> @m 2
An Integer

julia> x = 2
2

julia> @m x
1 arguments
```

18.4 代码生成

当需要大量重复的样板代码时，为了避免冗余，通常以编程方式生成它。在大多数语言中，这需要一个额外的构建步骤以及生成重复代码的独立程序。在 Julia 中，表达式插值和 eval 允许在通常的程序执行过程中生成这些代码。例如，考虑下列自定义类型

```
struct MyNumber
    x::Float64
end
# output
```

我们想为该类型添加一些方法。在下面的循环中，我们以编程的方式完成此工作：

```
for op = (:sin, :cos, :tan, :log, :exp)
    eval(quote
        Base.$op(a::MyNumber) = MyNumber($op(a.x))
    end)
end
# output
```

现在，我们对自定义类型调用这些函数：

```
julia> x = MyNumber(π)
MyNumber(3.141592653589793)

julia> sin(x)
MyNumber(1.2246467991473532e-16)

julia> cos(x)
MyNumber(-1.0)
```

在这种方法中，Julia 充当了自己的预处理器，并且允许从语言内部生成代码。使用 : 前缀的引用形式编写上述代码会使其更简洁：

```
for op = (:sin, :cos, :tan, :log, :exp)
    eval(:(Base.$op(a::MyNumber) = MyNumber($op(a.x))))
end
```

不管怎样，这种使用 eval(quote(...)) 模式生成语言内部的代码很常见，为此，Julia 自带了一个宏来缩写该模式：

```
for op = (:sin, :cos, :tan, :log, :exp)
    @eval Base.$op(a::MyNumber) = MyNumber($op(a.x))
end
```

@eval 重写此调用，使其与上面的较长版本完全等价。为了生成较长的代码块，可以把一个代码块作为表达式参数传给 @eval：

```
@eval begin
    # multiple lines
end
```

18.5 非标准字符串字面量

回想一下在字符串的文档中，以标识符为前缀的字符串字面量被称为非标准字符串字面量，它们可以具有与未加前缀的字符串字面量不同的语义。例如：

- `r"^\s*(?:#|$)"` produces a regular expression object rather than a string
- `b"DATA\xff\u2200"` is a byte array literal for `[68,65,84,65,255,226,136,128]`.

可能令人惊讶的是，这些行为并没有被硬编码到 Julia 的解释器或编译器中。相反，它们是由一个通用机制实现的自定义行为，且任何人都可以使用该机制：带前缀的字符串字面量被解析为特定名称的宏的调用。例如，正则表达式宏如下：

```
macro r_str(p)
    Regex(p)
end
```

这便是全部代码。这个宏说的是字符串字面量 `r"^\s*(?:#|$)"` 的字面内容应该传给宏 `@r_str`，并且展开后的结果应当放在该字符串字面量出现处的语法树中。换句话说，表达式 `r"^\s*(?:#|$)"` 等价于直接把下列对象放进语法树中：

```
Regex("^\\s*(?:#|\$)")
```

字符串字面量形式不仅更短、更方便，也更高效：因为正则表达式需要编译，Regex 对象实际上是在编译代码时创建的，所以编译只发生一次，而不是每次执行代码时都再编译一次。请考虑如果正则表达式出现在循环中：

```
for line = lines
    m = match(r"^\s*(?:#|$)", line)
    if m === nothing
        # non-comment
    else
        # comment
    end
end
```

因为正则表达式 `r"^\s*(?:#|$)"` 在这段代码解析时便已编译并被插入到语法树中，所以它只编译一次，而不是每次执行循环时都再编译一次。要在不使用宏的情况下实现此效果，必须像这样编写此循环：

```
re = Regex("^\\s*(?:#|\$)")
for line = lines
    m = match(re, line)
    if m === nothing
```

```
        # non-comment
    else
        # comment
    end
end
```

此外，如果编译器无法确定在所有循环中正则表达式对象都是常量，可能无法进行某些优化，使得此版本的效率依旧低于上面的更方便的字面量形式。当然，在某些情况下，非字面量形式更方便：如果需要向正则表达式中插入变量，就必须采用这种更冗长的方法；如果正则表达式模式本身是动态的，可能在每次循环迭代时发生变化，就必须在每次迭代中构造新的正则表达式对象。然而，在绝大多数用例中，正则表达式不是基于运行时的数据构造的。在大多数情况下，将正则表达式编写为编译期值的能力是无法估量的。

用户定义字符串文字的机制非常强大。不仅使用它实现了 Julia 的非标准字面量，而且还使用以下看起来无害的宏实现了命令行字面量语法（`` `echo "Hello, $person"` ``）：

```
macro cmd(str)
    :(cmd_gen($(shell_parse(str)[1])))
end
```

当然，这个宏的定义中使用的函数隐藏了许多复杂性，但它们只是函数且完全用 Julia 编写。你可以阅读它们的源代码并精确地看到它们的行为——它们所做的一切就是构造要插入到你的程序的语法树的表达式对象。

与字符串字面量一样，命令行字面量也可以以标识符为前缀，以形成所谓的非标准命令行字面量。这些命令行字面量被解析为对特殊命名的宏的调用。例如，语法 ``custom`literal` `` 被解析为 `@custom_cmd "literal"`。Julia 本身不包含任何非标准的命令行字面量，但包可以使用这种语法。除了不同的语法和 `_cmd` 后缀而不是 `_str` 后缀，非标准命令行字面量的行为与非标准字符串字面量完全一样。

如果两个模块提供了同名的非标准字符串或命令字面量，能使用模块名限定该字符串或命令字面量。例如，如果 `Foo` 和 `Bar` 提供了相同的字符串字面量 `@x_str`，那么可以编写 `Foo.x"literal"` 或 `Bar.x"literal"` 来消除两者的歧义。

以下是另一种定义宏的方式：

```
macro foo_str(str, flag)
    # do stuff
end
```

可以使用如下语法来调用这个宏

```
foo"str"flag
```

上述语法中 flag 的类型可以是一个 `String`，在字符串字面量之后包含的内容。

18.6 生成函数

有个非常特殊的宏叫 `@generated`，它允许你定义所谓的生成函数。它们能根据其参数类型生成专用代码，与用多重派发所能实现的代码相比，其代码更灵活和/或少。虽然宏在解析时使用表达式且无法访问其输入值的类型，但是生成函数在参数类型已知时会被展开，但该函数尚未编译。

生成函数的声明不会执行某些计算或操作，而会返回一个被引用的表达式，接着该表达式构成参数类型所对应方法的主体。在调用生成函数时，其返回的表达式会被编译然后执行。为了提高效率，通常会缓存结果。为了能推断是否缓存结果，只能使用语言的受限子集。因此，生成函数提供了一个灵活的方式来将工作重运行时移到编译时，代价则是其构造能力受到更大的限制。

定义生成函数与普通函数有五个主要区别：

1. 使用 `@generated` 标注函数声明。这会向 AST 附加一些信息，让编译器知道这个函数是生成函数。
2. 在生成函数的主体中，你只能访问参数的类型，而不能访问其值，以及在生成函数的定义之前便已定义的任何函数。
3. 不应计算某些东西或执行某些操作，应返回一个被引用的表达式，它会在被求值时执行你想要的操作。
4. 生成函数只允许调用在生成函数定义之前定义的函数。（如果不遵循这一点，引用来自未来世界的函数可能会导致 `MethodErrors` ）
5. 生成函数不能更改或观察任何非常量的全局状态。（例如，其包括 IO、锁、非局部的字典或者使用 `hasmethod`）即它们只能读取全局常量，且没有任何副作用。换句话说，它们必须是纯函数。由于实现限制，这也意味着它们目前无法定义闭包或生成器。

举例子来说明这个是最简单的。我们可以将生成函数 `foo` 声明为

```
julia> @generated function foo(x)
           Core.println(x)
           return :(x * x)
       end
foo (generic function with 1 method)
```

请注意，代码主体返回一个被引用的表达式，即 `:(x * x)`，而不仅仅是 `x * x` 的值。

从调用者的角度看，这与通常的函数等价；实际上，你无需知道你所调用的是通常的函数还是生成函数。让我们看看 `foo` 的行为：

```
julia> x = foo(2); # note: output is from println() statement in the body
Int64

julia> x           # now we print x
4

julia> y = foo("bar");
String

julia> y
"barbar"
```

因此，我们知道在生成函数的主体中，`x` 是所传递参数的类型，并且，生成函数的返回值是其定义所返回的被引用的表达式的求值结果，在该表达式求值时 `x` 表示其值。

如果我们使用我们已经使用过的类型再次对 `foo` 求值会发生什么？

```
julia> foo(4)
16
```

请注意，这里并没有打印 Int64。我们可以看到对于特定的参数类型集来说，生成函数的主体只执行一次，且结果会被缓存。此后，对于此示例，生成函数首次调用返回的表达式被重新用作方法主体。但是，实际的缓存行为是由实现定义的性能优化，过于依赖此行为并不实际。

生成函数可能只生成一次函数, 但也可能多次生成，或者看起来根本就没有生成过函数。因此，你应该从不编写有副作用的生成函数——因为副作用发生的时间和频率是不确定的。（对于宏来说也是如此——跟宏一样，在生成函数中使用 eval 也许意味着你正以错误的方式做某事。）但是，与宏不同，运行时系统无法正确处理对 eval 的调用，所以不允许这样做。

理解 @generated 函数与方法的重定义间如何相互作用也很重要。遵循正确的 @generated 函数不能观察任何可变状态或导致全局状态的任何更改的原则，我们看到以下行为。观察到，生成函数不能调用在生成函数本身的定义之前未定义的任何方法。

一开始 f(x) 有一个定义

```
julia> f(x) = "original definition";
```

定义使用 f(x) 的其它操作：

```
julia> g(x) = f(x);

julia> @generated gen1(x) = f(x);

julia> @generated gen2(x) = :(f(x));
```

我们现在为 f(x) 添加几个新定义：

```
julia> f(x::Int) = "definition for Int";

julia> f(x::Type{Int}) = "definition for Type{Int}";
```

并比较这些结果的差异：

```
julia> f(1)
"definition for Int"

julia> g(1)
"definition for Int"

julia> gen1(1)
"original definition"

julia> gen2(1)
"definition for Int"
```

生成函数的每个方法都有自己的已定义函数视图：

```
julia> @generated gen1(x::Real) = f(x);

julia> gen1(1)
"definition for Type{Int}"
```

上例中的生成函数 foo 能做的，通常的函数 foo(x) = x * x 也能做（除了在第一次调用时打印类型，并产生了更高的开销)。但是，生成函数的强大之处在于其能够根据传递给它的类型计算不同的被引用的表达式：

```
julia> @generated function bar(x)
           if x <: Integer
               return :(x ^ 2)
           else
               return :(x)
           end
       end
bar (generic function with 1 method)

julia> bar(4)
16

julia> bar("baz")
"baz"
```

（当然，这个刻意的例子可以更简单地通过多重派发实现……)

滥用它会破坏运行时系统并导致未定义行为：

```
julia> @generated function baz(x)
           if rand() < .9
               return :(x^2)
           else
               return :("boo!")
           end
       end
baz (generic function with 1 method)
```

由于生成函数的主体具有不确定性，其行为和所有后续代码的行为并未定义。

不要复制这些例子！

这些例子有助于说明生成函数定义和调用的工作方式；但是，不要复制它们，原因如下：

- foo 函数有副作用（对 Core.println 的调用），并且未确切定义这些副作用发生的时间、频率和次数。
- bar 函数解决的问题可通过多重派发被更好地解决——定义 bar(x) = x 和 bar(x::Integer) = x ^ 2 会做同样的事，但它更简单和快捷。
- baz 函数是病态的

请注意，不应在生成函数中尝试的操作并无严格限制，且运行时系统现在只能检测一部分无效操作。还有许多操作只会破坏运行时系统而没有通知，通常以微妙的方式而非显然地与错误的定义相关联。因为函数生成器是在类型推导期间运行的，所以它必须遵守该代码的所有限制。

一些不应该尝试的操作包括：

1. 缓存本地指针。
2. 以任何方式与 Core.Compiler 的内容或方法交互。
3. 观察任何可变状态。
   - 生成函数的类型推导可以在任何时候运行，包括你的代码正在尝试观察或更改此状态时。
4. 采用任何锁：你调用的 C 代码可以在内部使用锁（例如，调用 malloc 不会有问题，即使大多数实现在内部需要锁），但是不要试图在执行 Julia 代码时保持或请求任何锁。
5. 调用在生成函数的主体后定义的任何函数。对于增量加载的预编译模块，则放宽此条件，以允许调用模块中的任何函数。

那好，我们现在已经更好地理解了生成函数的工作方式，让我们使用它来构建一些更高级（和有效）的功能……

一个高级的例子

Julia 的 base 库有个内部函数 sub2ind，用于根据一组 n 重线性索引计算 n 维数组的线性索引——换句话说，用于计算索引 i，其可用于使用 A[i] 来索引数组 A，而不是用 A[x,y,z,...]。一种可能的实现如下：

```
julia> function sub2ind_loop(dims::NTuple{N}, I::Integer...) where N
           ind = I[N] - 1
           for i = N-1:-1:1
               ind = I[i]-1 + dims[i]*ind
           end
           return ind + 1
       end
sub2ind_loop (generic function with 1 method)

julia> sub2ind_loop((3, 5), 1, 2)
4
```

用递归可以完成同样的事情：

```
julia> sub2ind_rec(dims::Tuple{}) = 1;

julia> sub2ind_rec(dims::Tuple{}, i1::Integer, I::Integer...) =
           i1 == 1 ? sub2ind_rec(dims, I...) : throw(BoundsError());

julia> sub2ind_rec(dims::Tuple{Integer, Vararg{Integer}}, i1::Integer) = i1;

julia> sub2ind_rec(dims::Tuple{Integer, Vararg{Integer}}, i1::Integer, I::Integer...) =
           i1 + dims[1] * (sub2ind_rec(Base.tail(dims), I...) - 1);

julia> sub2ind_rec((3, 5), 1, 2)
4
```

这两种实现虽然不同，但本质上做同样的事情：在数组维度上的运行时循环，将每个维度上的偏移量收集到最后的索引中。

然而，循环所需的信息都已嵌入到参数的类型信息中。This allows the compiler to move the iteration to compile time and eliminate the runtime loops altogether. We can utilize generated functions to achieve a similar effect; 用编译器的说法，我们用生成函数手动展开循环。代码主体变得几乎相同，但我们不是计算线性索引，而是建立计算索引的表达式：

```
julia> @generated function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
           ex = :(I[$N] - 1)
           for i = (N - 1):-1:1
               ex = :(I[$i] - 1 + dims[$i] * $ex)
           end
           return :($ex + 1)
       end
sub2ind_gen (generic function with 1 method)

julia> sub2ind_gen((3, 5), 1, 2)
4
```

这会生成什么代码？

找出所生成代码的一个简单方法是将生成函数的主体提取到另一个（通常的）函数中：

```
julia> @generated function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
           return sub2ind_gen_impl(dims, I...)
       end
sub2ind_gen (generic function with 1 method)

julia> function sub2ind_gen_impl(dims::Type{T}, I...) where T <: NTuple{N,Any} where N
           length(I) == N || return :(error("partial indexing is unsupported"))
           ex = :(I[$N] - 1)
           for i = (N - 1):-1:1
               ex = :(I[$i] - 1 + dims[$i] * $ex)
           end
           return :($ex + 1)
       end
sub2ind_gen_impl (generic function with 1 method)
```

我们现在可以执行 `sub2ind_gen_impl` 并检查它所返回的表达式：

```
julia> sub2ind_gen_impl(Tuple{Int,Int}, Int, Int)
:(((I[1] - 1) + dims[1] * (I[2] - 1)) + 1)
```

因此，这里使用的方法主体根本不包含循环——只有两个元组的索引、乘法和加法/减法。所有循环都是在编译期执行的，我们完全避免了在执行期间的循环。因此，我们只需对每个类型循环一次，在本例中每个 `N` 循环一次（除了在该函数被多次生成的边缘情况——请参阅上面的免责声明）。

可选地生成函数

生成函数可以在运行时实现高效率，但需要编译时间成本：必须为具体的参数类型的每个组合生成新的函数体。通常，Julia 能够编译函数的「泛型」版本，其适用于任何参数，但对于生成函数，这是不可能的。这意味着大量使用生成函数的程序可能无法静态编译。

为了解决这个问题，语言提供用于编写生成函数的通常、非生成的替代实现的语法。应用于上面的 sub2ind 示例，它看起来像这样：

```
function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
    if N != length(I)
        throw(ArgumentError("Number of dimensions must match number of indices."))
    end
    if @generated
        ex = :(I[$N] - 1)
        for i = (N - 1):-1:1
            ex = :(I[$i] - 1 + dims[$i] * $ex)
        end
        return :($ex + 1)
    else
        ind = I[N] - 1
        for i = (N - 1):-1:1
            ind = I[i] - 1 + dims[i]*ind
        end
        return ind + 1
    end
end
```

在内部，这段代码创建了函数的两个实现：一个生成函数的实现，其使用 if @generated 中的第一个块，一个通常的函数的实现，其使用 else 块。在 if @generated 块的 then 部分中，代码与其它生成函数具有相同的语义：参数名称引用类型，且代码应返回表达式。可能会出现多个 if @generated 块，在这种情况下，生成函数的实现使用所有的 then 块，而替代实现使用所有的 else 块。

请注意，我们在函数顶部添加了错误检查。此代码对两个版本都是通用的，且是两个版本中的运行时代码（它将被引用并返回为生成函数版本中的表达式）。这意味着局部变量的值和类型在代码生成时不可用——用于代码生成的代码只能看到参数类型。

在这种定义方式中，代码生成功能本质上只是一种可选的优化。如果方便，编译器将使用它，否则可能选择使用通常的实现。这种方式是首选的，因为它允许编译器做出更多决策和以更多方式编译程序，还因为通常代码比由代码生成的代码更易读。但是，使用哪种实现取决于编译器实现细节，因此，两个实现的行为必须相同。

Chapter 19

一维和多维数组

与大多数科学计算语言一样，Julia 提供原生的数组实现。大多数科学计算语言非常重视其数组实现，而牺牲了其他容器。Julia 没有以任何特殊方式处理数组。就像和其它用 Julia 写的代码一样，Julia 的数组库几乎完全是用 Julia 自身实现的，并且由编译器保证其性能。因此，也可以通过继承 `AbstractArray` 来定义自定义数组类型。有关实现自定义数组类型的更多详细信息，请参阅 AbstractArray 接口的手册部分。

数组是存储在多维网格中对象的集合。允许使用零维数组，请参见 [常见问题]（@ref faq-array-0dim）。在最一般的情况下，数组中的对象可能是 `Any` 类型。对于大多数计算上的需求，数组中对象的类型应该更加具体，例如 `Float64` 或 `Int32`。

一般来说，与许多其他科学计算语言不同，Julia 不希望为了性能而以向量化的方式编写程序。Julia 的编译器使用类型推断，并为标量数组索引生成优化的代码，从而能够令用户方便地编写可读性良好的程序，而不牺牲性能，并且时常会减少内存使用。

在 Julia 中，所有函数的参数都是 非复制的方式进行传递的（比如说，通过指针传递）。一些科学计算语言用传值的方式传递数组，尽管这样做可以防止数组在被调函数中被意外地篡改，但这也会导致不必要的数组拷贝。作为 Julia 的一个惯例，以一个 ! 结尾的函数名它会对自己的一个或者多个参数的值进行修改或者销毁（例如，请比较 `sort` 和 `sort!`）。被调函数必须进行显式拷贝，以确保它们不会无意中修改输入参数。很多不以 ! 结尾的函数在实现的时候，都会先进行显式拷贝，然后调用一个以 ! 结尾的同名函数，最后返回之前拷贝的副本。

19.1 基本函数

函数	描述
`eltype(A)`	A 中元素的类型
`length(A)`	A 中元素的数量
`ndims(A)`	A 的维数
`size(A)`	一个包含 A 各个维度上元素数量的元组
`size(A,n)`	A 第 n 维中的元素数量
`axes(A)`	一个包含 A 有效索引的元组
`axes(A,n)`	第 n 维有效索引的范围
`eachindex(A)`	一个访问 A 中每一个位置的高效迭代器
`stride(A,k)`	在第 k 维上的间隔（stride）（相邻元素间的线性索引距离）
`strides(A)`	包含每一维上的间隔（stride）的元组

19.2 构造和初始化

Julia 提供了许多用于构造和初始化数组的函数。在下列函数中，参数 dims ... 可以是一个元组 tuple 来表示维数，也可以是一个可变长度的整数值作为维数。大部分函数的第一个参数都表示数组的元素类型 T 。如果类型 T 被省略，那么将默认为 Float64。

函数	描述
Array{T}(undef, dims...)	一个没有初始化的密集 Array
zeros(T, dims...)	一个全零 Array
ones(T, dims...)	一个元素均为 1 的 Array
trues(dims...)	一个每个元素都为 true 的 BitArray
falses(dims...)	一个每个元素都为 false 的 BitArray
reshape(A, dims...)	一个包含跟 A 相同数据但维数不同的数组
copy(A)	拷贝 A
deepcopy(A)	深拷贝，即拷贝 A，并递归地拷贝其元素
similar(A, T, dims...)	一个与 A 具有相同类型（这里指的是密集，稀疏等）的未初始化数组，但具有指定的元素类型和维数。第二个和第三个参数都是可选的，如果省略则默认为元素类型和 A 的维数。
reinterpret(T, A)	与 A 具有相同二进制数据的数组，但元素类型为 T
rand(T, dims...)	一个随机 Array，其元素值是 iid [1] 和均匀分布值的。对于浮点类型 T，数值位于半开放区间 $[0, 1)$ 内。
randn(T, dims...)	一个随机 Array，元素为标准正态分布，服从独立同分布
Matrix{T}(I, m, n)	m 行 n 列的单位矩阵（需要先执行 using LinearAlgebra 来才能使用 I）
range(start, stop, n)	从 start 到 stop 的带有 n 个线性间隔元素的范围
fill!(A, x)	用值 x 填充数组 A
fill(x, dims...)	一个由 x 填充的 Array。特别的，fill(x) 构造了一个包含 x 的零维 Array。

要查看各种方法，我们可以将不同维数传递给这些构造函数，请考虑以下示例：

```
julia> zeros(Int8, 2, 3)
2×3 Matrix{Int8}:
 0  0  0
 0  0  0

julia> zeros(Int8, (2, 3))
2×3 Matrix{Int8}:
 0  0  0
 0  0  0

julia> zeros((2, 3))
```

[1] *iid*，独立同分布

```
2×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
```

此处, (2, 3) 是一个元组 Tuple 并且第一个参数——元素类型是可选的, 默认值为 Float64.

19.3 数组常量

数组也可以直接用方括号来构造; 语法 [A, B, C, ...] 创建一个一维数组 (即一个向量), 该一维数组的元素用逗号分隔。所创建的数组中元素的类型 (eltype) 自动由括号内参数的类型确定。如果所有参数类型都相同, 则该类型称为数组的 eltype。如果所有元素都有相同的promotion type, 那么个元素都由convert转换成该类型并且该类型为数组的 eltype。否则, 生成一个可以包含任意类型的异构数组——Vector{Any} ; 该构造方法包含字符 [], 此时构造过程无参数给出。Array literal can be typed with the syntax T[A, B, C, ...] where T is a type.

```
julia> [1,2,3] # 元素类型为 Int 的向量
3-element Vector{Int64}:
 1
 2
 3

julia> promote(1, 2.3, 4//5) # Int, Float64 以及 Rational 类型放在一起则会提升到 Float64
(1.0, 2.3, 0.8)

julia> [1, 2.3, 4//5] # 从而它就是这个矩阵的元素类型
3-element Vector{Float64}:
 1.0
 2.3
 0.8

julia> Float32[1, 2.3, 4//5] # Specify element type manually
3-element Vector{Float32}:
 1.0
 2.3
 0.8

julia> []
Any[]
```

数组拼接

如果方括号里的参数不是由逗号分隔, 而是由单个分号 (;) 或者换行符分隔, 那么每一个参数就不再解析为一个单独的数组元素, 而是纵向拼接起来。

```
julia> [1:2, 4:5] # 这里有一个逗号, 因此并不会发生矩阵的拼接。这里居然的元素本身就是这些 range
2-element Vector{UnitRange{Int64}}:
 1:2
 4:5

julia> [1:2; 4:5]
4-element Vector{Int64}:
 1
```

```
 2
 4
 5

julia> [1:2
        4:5
        6]
5-element Vector{Int64}:
 1
 2
 4
 5
 6
```

类似的，如果这些参数是被制表符、空格符或者两个分号所分隔，那么它们的内容就横向拼接在一起。

```
julia> [1:2  4:5  7:8]
2×3 Matrix{Int64}:
 1  4  7
 2  5  8

julia> [[1,2]  [4,5]  [7,8]]
2×3 Matrix{Int64}:
 1  4  7
 2  5  8

julia> [1 2 3] # 数字可以被横向拼接
1×3 Matrix{Int64}:
 1  2  3

julia> [1;; 2;; 3;; 4]
1×4 Matrix{Int64}:
 1  2  3  4
```

单个分号（或换行符）和空格（或制表符）可以被结合起来使用进行横向或者纵向的拼接。

```
julia> [1 2
        3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> [zeros(Int, 2, 2) [1; 2]
        [3 4]            5]
3×3 Matrix{Int64}:
 0  0  1
 0  0  2
 3  4  5

julia> [[1 1]; 2 3; [4 4]]
3×2 Matrix{Int64}:
 1  1
```

```
 2  3
 4  4
```

Spaces (and tabs) have a higher precedence than semicolons, performing any horizontal concatenations first and then concatenating the result. Using double semicolons for the horizontal concatenation, on the other hand, performs any vertical concatenations before horizontally concatenating the result.

```
julia> [zeros(Int, 2, 2) ; [3 4] ;; [1; 2] ; 5]
3×3 Matrix{Int64}:
 0  0  1
 0  0  2
 3  4  5

julia> [1:2; 4;; 1; 3:4]
3×2 Matrix{Int64}:
 1  1
 2  3
 4  4
```

Just as `;` and `;;` concatenate in the first and second dimension, using more semicolons extends this same general scheme. The number of semicolons in the separator specifies the particular dimension, so `;;;` concatenates in the third dimension, `;;;;` in the 4th, and so on. Fewer semicolons take precedence, so the lower dimensions are generally concatenated first.

```
julia> [1; 2;; 3; 4;; 5; 6;;;
        7; 8;; 9; 10;; 11; 12]
2×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  3  5
 2  4  6

[:, :, 2] =
 7   9  11
 8  10  12
```

Like before, spaces (and tabs) for horizontal concatenation have a higher precedence than any number of semicolons. Thus, higher-dimensional arrays can also be written by specifying their rows first, with their elements textually arranged in a manner similar to their layout:

```
julia> [1 3 5
        2 4 6;;;
        7 9 11
        8 10 12]
2×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  3  5
 2  4  6

[:, :, 2] =
 7   9  11
 8  10  12
```

```
julia> [1 2;;; 3 4;;;; 5 6;;; 7 8]
1×2×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 1  2

[:, :, 2, 1] =
 3  4

[:, :, 1, 2] =
 5  6

[:, :, 2, 2] =
 7  8

julia> [[1 2;;; 3 4];;;; [5 6];;; [7 8]]
1×2×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 1  2

[:, :, 2, 1] =
 3  4

[:, :, 1, 2] =
 5  6

[:, :, 2, 2] =
 7  8
```

Although they both mean concatenation in the second dimension, spaces (or tabs) and `;;` cannot appear in the same array expression unless the double semicolon is simply serving as a ”line continuation” character. This allows a single horizontal concatenation to span multiple lines (without the line break being interpreted as a vertical concatenation).

```
julia> [1 2 ;;
       3 4]
1×4 Matrix{Int64}:
 1  2  3  4
```

Terminating semicolons may also be used to add trailing length 1 dimensions.

```
julia> [1;;]
1×1 Matrix{Int64}:
 1

julia> [2; 3;;;]
2×1×1 Array{Int64, 3}:
[:, :, 1] =
 2
 3
```

空格（和制表符）的优先级高于分号，首先执行任何纵向拼接，然后拼接结果。另一方面，使用双分号进行水平连接时，先纵向拼接再横向拼接。

Syntax	Function	Description
	cat	concatenate input arrays along dimension(s) k
[A; B; C; ...]	vcat	shorthand for `cat(A...; dims=1)`
[A B C ...]	hcat	shorthand for `cat(A...; dims=2)`
[A B; C D; ...]	hvcat	simultaneous vertical and horizontal concatenation
[A; C;; B; D;;; ...]	hvncat	simultaneous n-dimensional concatenation, where number of semicolons indicate the dimension to concatenate

Typed array literals

正如 ; 和 ;; 在第一维和第二维中拼接一样，使用更多的分号扩展了相同的通用方案。分隔符中的分号数指定了特定的维度，因此;;; 在第三个维度中拼接，;;;; 在第四个维度中，依此类推。较少的分号优先级高，因此较低的维度通常首先拼接。

```
julia> [1; 2;; 3; 4;; 5; 6;;;
        7; 8;; 9; 10;; 11; 12]
2×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  3  5
 2  4  6

[:, :, 2] =
 7   9  11
 8  10  12
```

像之前一样，用于水平拼接的空格（和制表符）的优先级高于任何数量的分号。因此，高维数组也可以通过首先指定它们的行来编写，它们的元素以类似于它们的布局的方式进行文本排列：

```
julia> [1 3 5
        2 4 6;;;
        7 9 11
        8 10 12]
2×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  3  5
 2  4  6

[:, :, 2] =
 7   9  11
 8  10  12

julia> [1 2;;; 3 4;;;; 5 6;;; 7 8]
1×2×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 1  2

[:, :, 2, 1] =
 3  4
```

```
[:, :, 1, 2] =
 5  6

[:, :, 2, 2] =
 7  8

julia> [[1 2;;; 3 4];;;; [5 6];;; [7 8]]
1×2×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 1  2

[:, :, 2, 1] =
 3  4

[:, :, 1, 2] =
 5  6

[:, :, 2, 2] =
 7  8
```

尽管它们都表示第二维中的连接，但空格（或制表符）和 ;; 不能出现在同一个数组表达式中，除非双分号只是作为“行继续”字符。这允许单个水平拼接跨越多行（不会将换行符解释为垂直拼接）。

```
julia> [1 2 ;;
       3 4]
1×4 Matrix{Int64}:
 1  2  3  4
```

终止分号也可用于在最后添加 1 个长度为 1 的维度。

```
julia> [1;;]
1×1 Matrix{Int64}:
 1

julia> [2; 3;;;]
2×1×1 Array{Int64, 3}:
[:, :, 1] =
 2
 3
```

更一般地，可以通过cat 函数来实现数组元素的拼接功能。以下这些的语法为这些函数的简写形式，它们本身也是非常方便使用的：

语法	函数	描述
	cat	沿着 s 的第 k 维拼接数组
[A; B; C; ...]	vcat	‘cat(A...; dims=1) 的简写
[A B C ...]	hcat	‘cat(A...; dims=2) 的简写
[A B; C D; ...]	hvcat	同时沿垂直和水平方向拼接
[A; C;; B; D;;; ...]	hvncat	同时进行 n 维拼接，其中分号的数量表示拼接所在的维度

指定类型的数组字面量

可以用 `T[A, B, C, ...]` 的方式声明一个元素为某种特定类型的数组。该方法定义一个元素类型为 `T` 的一维数组并且初始化元素为 A, B, C, ...。比如，`Any[x, y, z]` 会构建一个异构数组，该数组可以包含任意类型的元素。

类似的，拼接也可以用类型为前缀来指定结果的元素类型。

```
julia> [[1 2] [3 4]]
1×4 Matrix{Int64}:
 1  2  3  4

julia> Int8[[1 2] [3 4]]
1×4 Matrix{Int8}:
 1  2  3  4
```

19.4 数组推导

（数组）推导提供了构造数组的通用且强大的方法。其语法类似于数学中的集合构造的写法：

```
A = [ F(x, y, ...) for x=rx, y=ry, ... ]
```

这种形式的含义是 `F(x,y,...)` 取其给定列表中变量 `x`，`y` 等的每个值进行计算。值可以指定为任何可迭代对象，但通常是 `1:n` 或 `2:(n-1)` 之类的范围，或者像 `[1.2, 3.4, 5.7]` 这样的显式数组值。结果是一个 N 维密集数组，将变量范围 `rx`，`ry` 等的维数拼接起来得到其维数，并且每次 `F(x,y,...)` 计算返回一个标量。

下面的示例计算当前元素和沿一维网格其左，右相邻元素的加权平均值：

```
julia> x = rand(8)
8-element Array{Float64,1}:
 0.843025
 0.869052
 0.365105
 0.699456
 0.977653
 0.994953
 0.41084
 0.809411

julia> [ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
6-element Array{Float64,1}:
 0.736559
 0.57468
 0.685417
 0.912429
 0.8446
 0.656511
```

生成的数组的类型取决于参与计算元素的类型，就像数组字面量一样。为了显式地控制类型，可以在数组推导之前指定类型。例如，我们可以要求推导的结果为单精度类型：

```
Float32[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
```

19.5 生成器表达式

也可以在没有方括号的情况下编写（数组）推导，从而产生称为生成器的对象。可以迭代此对象以按需生成值，而不是预先分配数组并存储它们（请参阅 迭代）。例如，以下表达式在不分配内存的情况下对一个序列进行求和：

```
julia> sum(1/n^2 for n=1:1000)
1.6439345666815615
```

在参数列表中使用具有多个维度的生成器表达式时，需要使用括号将生成器与后续参数分开：

```
julia> map(tuple, 1/(i+j) for i=1:2, j=1:2, [1:4;])
ERROR: syntax: invalid iteration specification
```

`for` 后面所有逗号分隔的表达式都被解释为范围。添加括号让我们可以向 `map` 中添加第三个参数：

```
julia> map(tuple, (1/(i+j) for i=1:2, j=1:2), [1 3; 2 4])
2×2 Matrix{Tuple{Float64, Int64}}:
 (0.5, 1)       (0.333333, 3)
 (0.333333, 2)  (0.25, 4)
```

生成器是通过内部函数实现。与本语言中别处使用的内部函数一样，封闭作用域中的变量可以在内部函数中被「捕获」。例如，`sum(p[i] - q[i] for i=1:n)` 从封闭作用域中捕获三个变量 `p`、`q` 和 `n`。但是变量捕获可能会带来性能挑战；请参阅 性能提示。

通过编写多个 `for` 关键字，生成器和推导中的范围可以取决于之前的范围：

```
julia> [(i, j) for i=1:3 for j=1:i]
6-element Vector{Tuple{Int64, Int64}}:
 (1, 1)
 (2, 1)
 (2, 2)
 (3, 1)
 (3, 2)
 (3, 3)
```

在这些情况下，结果都是一维的。

可以使用 `if` 关键字过滤生成的值：

```
julia> [(i, j) for i=1:3 for j=1:i if i+j == 4]
2-element Vector{Tuple{Int64, Int64}}:
 (2, 2)
 (3, 1)
```

19.6 索引

索引 n 维数组 A 的一般语法是：

```
X = A[I_1, I_2, ..., I_n]
```

其中每个 I_k 可以是标量整数，整数数组或任何其他支持的索引类型。这包括 Colon (:) 来选择整个维度中的所有索引，形式为 a:c 或 a:b:c 的范围来选择连续或跨步的子区间，以及布尔数组以选择索引为 true 的元素。

如果所有索引都是标量，则结果 X 是数组 A 中的单个元素。否则，X 是一个数组，其维数与所有索引的维数之和相同。

如果所有索引 I_k 都是向量，则 X 的形状将是 (length(I_1), length(I_2), ..., length(I_n))，其中，X 中位于 i_1, i_2, ..., i_n 处的元素为 A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]。

例如：

```
julia> A = reshape(collect(1:16), (2, 2, 2, 2))
2×2×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
 6  8

[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> A[1, 2, 1, 1] # 全部为标量索引
3

julia> A[[1, 2], [1], [1, 2], [1]] # 全部为向量索引
2×1×2×1 Array{Int64, 4}:
[:, :, 1, 1] =
 1
 2

[:, :, 2, 1] =
 5
 6

julia> A[[1, 2], [1], [1, 2], 1] # 标量与向量索引的混合使用
2×1×2 Array{Int64, 3}:
[:, :, 1] =
 1
 2

[:, :, 2] =
```

```
5
6
```

请注意最后两种情况下得到的数组大小为何是不同的。

如果 `I_1` 是二维矩阵，则 `X` 是 `n+1` 维数组，其形状为 `(size(I_1, 1), size(I_1, 2), length(I_2), ..., length(I_n))`。矩阵会添加一个维度。

例如：

```
julia> A = reshape(collect(1:16), (2, 2, 2, 2));

julia> A[[1 2; 1 2]]
2×2 Matrix{Int64}:
 1  2
 1  2

julia> A[[1 2; 1 2], 1, 2, 1]
2×2 Matrix{Int64}:
 5  6
 5  6
```

位于 `i_1, i_2, i_3, ..., i_{n+1}` 处的元素值是 `A[I_1[i_1, i_2], I_2[i_3], ..., I_n[i_{n+1}]]`。所有使用标量索引的维度都将被丢弃，例如，假设 `J` 是索引数组，那么 `A[2, J, 3]` 的结果是一个大小为 `size(J)` 的数组、其第 j 个元素由 `A[2, J[j], 3]` 填充。

作为此语法的特殊部分，`end` 关键字可用于表示索引括号内每个维度的最后一个索引，由索引的最内层数组的大小决定。没有 `end` 关键字的索引语法相当于调用`getindex`：

```
X = getindex(A, I_1, I_2, ..., I_n)
```

例如：

```
julia> x = reshape(1:16, 4, 4)
4×4 reshape(::UnitRange{Int64}, 4, 4) with eltype Int64:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[2:3, 2:end-1]
2×2 Matrix{Int64}:
 6  10
 7  11

julia> x[1, [2 3; 4 1]]
2×2 Matrix{Int64}:
  5  9
 13  1
```

19.7 索引赋值

在 n 维数组 A 中赋值的一般语法是：

```
A[I_1, I_2, ..., I_n] = X
```

其中每个 I_k 可以是标量整数，整数数组或任何其他支持的索引类型。这包括 Colon (:) 来选择整个维度中的所有索引，形式为 a:c 或 a:b:c 的范围来选择连续或跨步的子区间，以及布尔数组以选择索引为 true 的元素。

如果所有 I_k 都为整数，则数组 A 中 I_1, I_2, ..., I_n 位置的值将被 X 的值覆盖，必要时将 convert 为数组 A 的 eltype。

如果索引 I_k 本身就是一个数组，那么右侧的 X 也必须是一个与索引 A[I_1, I_2, ..., I_n] 的结果具有相同形状的数组或是具有相同数量元素的向量。A 的位置 I_1[i_1], I_2[i_2], ..., I_n[i_n] 中的值被值 X[I_1, I_2, ..., I_n] 覆盖，如果必要也会进行类型转换。元素分配运算符 .= 可以用于沿着所选区域 广播 X：

```
A[I_1, I_2, ..., I_n] .= X
```

就像在索引中一样，end 关键字可用于表示索引括号中每个维度的最后一个索引，由被赋值的数组大小决定。没有 end 关键字的索引赋值语法相当于调用setindex!：

```
setindex!(A, X, I_1, I_2, ..., I_n)
```

例如：

```
julia> x = collect(reshape(1:9, 3, 3))
3×3 Matrix{Int64}:
 1  4  7
 2  5  8
 3  6  9

julia> x[3, 3] = -9;

julia> x[1:2, 1:2] = [-1 -4; -2 -5];

julia> x
3×3 Matrix{Int64}:
 -1  -4   7
 -2  -5   8
  3   6  -9
```

19.8 支持的索引类型

在表达式 A[I_1, I_2, ..., I_n] 中，每个 I_k 可以是标量索引，标量索引数组，或者用 to_indices 转换成的表示标量索引数组的对象：

1. 标量索引。默认情况下，这包括：

    - 非布尔的整数

  - CartesianIndex{N} 用来表达多个维度的信息（详见下文），其内部实际为 N 个整数组成的元组。

2. 标量索引数组。这包括：

  - 整数向量和多维整数数组
  - 像 [] 这样的空数组，它不选择任何元素。e.g. A[[]] (not to be confused with A[])
  - 如 a:c 或 a:b:c 的范围，从 a 到 c（包括）选择连续或间隔的部分元素
  - 任何自定义标量索引数组，它是 AbstractArray 的子类型
  - CartesianIndex{N} 数组（详见下文）

3. 一个表示标量索引数组的对象，可以通过to_indices转换为这样的对象。默认情况下，这包括：

  - Colon() (:)，表示整个维度内或整个数组中的所有索引
  - 布尔数组，选择其中值为 true 的索引对应的元素（更多细节见下文）

一些例子：

```
julia> A = reshape(collect(1:2:18), (3, 3))
3×3 Matrix{Int64}:
 1   7  13
 3   9  15
 5  11  17

julia> A[4]
7

julia> A[[2, 5, 8]]
3-element Vector{Int64}:
  3
  9
 15

julia> A[[1 4; 3 8]]
2×2 Matrix{Int64}:
 1   7
 5  15

julia> A[[]]
Int64[]

julia> A[1:2:5]
3-element Vector{Int64}:
 1
 5
 9

julia> A[2, :]
3-element Vector{Int64}:
  3
  9
 15
```

```
julia> A[:, 3]
3-element Vector{Int64}:
 13
 15
 17

julia> A[:, 3:3]
3×1 Matrix{Int64}:
 13
 15
 17
```

笛卡尔索引

特殊的 CartesianIndex{N} 对象表示一个标量索引，其行为类似于张成多个维度的 N 维整数元组。例如：

```
julia> A = reshape(1:32, 4, 4, 2);

julia> A[3, 2, 1]
7

julia> A[CartesianIndex(3, 2, 1)] == A[3, 2, 1] == 7
true
```

单独来看的话，这看起来很平凡：CartesianIndex 单纯只是将多个整数捆绑在一起作为一个对象来表示一个多维下标。当与其他取下标方式和生成 CartesianIndex 的迭代器进行工作的时候，它才真正能展现出它的简洁与高效。关于这个你可以参考 迭代器 这一部分，你也可以参考 关于多维算法和迭代器的介绍 这篇博客来了解更进阶的用法。

元素类型为 CartesianIndex{N} 的矩阵也是支持的。每一个元素都单独表示一个 N 维空间的索引下标，作为一个整体这样一个矩阵则表示一些 N 维空间的点的坐标，因此这种形式有时也称为逐点索引。例如：你可以通过它来访问上面所定义的三维矩阵 A 的第一页（第三维指标为 1）的对角线元素：

```
julia> page = A[:, :, 1]
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> page[[CartesianIndex(1, 1),
             CartesianIndex(2, 2),
             CartesianIndex(3, 3),
             CartesianIndex(4, 4)]]
4-element Vector{Int64}:
  1
  6
 11
 16
```

这可以通过 dot broadcasting 以及普通整数索引（而不是把从 A 中提取第一“页”作为单独的步骤）更加简单地表达。它甚至可以与 : 结合使用，同时从两个页面中提取两个对角线：

```
julia> A[CartesianIndex.(axes(A, 1), axes(A, 2)), 1]
4-element Vector{Int64}:
  1
  6
 11
 16

julia> A[CartesianIndex.(axes(A, 1), axes(A, 2)), :]
4×2 Matrix{Int64}:
  1  17
  6  22
 11  27
 16  32
```

Warning

CartesianIndex 和 CartesianIndex 数组与用来表示维度的最后一个索引的 end 关键字不兼容。不要在可能包含 CartesianIndex 或其数组的索引表达式中使用 end。

逻辑索引

通常被称为逻辑索引或带有逻辑掩码的索引，通过布尔数组进行索引选择其值为 true 的索引处的元素。通过布尔向量 B 进行索引实际上与通过findall(B)返回的整数向量进行索引相同。类似地，通过 N 维布尔数组进行索引与通过其值为 true 的 CartesianIndex{N} 的向量进行索引实际上是相同的。一个逻辑索引必须是一个与它所索引的维度长度相同的向量，或者它必须是唯一提供的索引并且匹配它所索引到的数组的大小和维度。通常直接使用布尔数组作为索引更有效，而不是调用 findall。

```
julia> x = reshape(1:16, 4, 4)
4×4 reshape(::UnitRange{Int64}, 4, 4) with eltype Int64:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[[false, true, true, false], :]
2×4 Matrix{Int64}:
 2  6  10  14
 3  7  11  15

julia> mask = map(ispow2, x)
4×4 Matrix{Bool}:
 1  0  0  0
 1  0  0  0
 0  0  0  0
 1  1  0  1

julia> x[mask]
5-element Vector{Int64}:
  1
  2
  4
```

```
  8
 16
```

索引数

笛卡尔索引

通常，为一个 N 维数组元素使用索引的方式是使用 N 个数字作为索引，每一个索引值确定一个具体的维度。例如，一个三维数组 `A = rand(4, 3, 2)`，`A[2, 3, 1]` 将选择的第二行第三列第一“页”中的元素。这种方式通常也被成为笛卡尔索引。

线性索引

当恰好提供了一个索引 `i` 时，该索引不再表示数组特定维度中的位置。相反，它使用线性遍历整个数组的列主迭代顺序选择第 `i` 个元素。这称为线性索引。它本质上将数组视为使用 `vec` 将其重新整形为一维向量。

```
julia> A = [2 6; 4 7; 3 1]
3×2 Matrix{Int64}:
 2  6
 4  7
 3  1

julia> A[5]
7

julia> vec(A)[5]
7
```

数组 `A` 中的线性索引可以转换为 `CartesianIndex` 以使用 `CartesianIndices(A)[i]` 进行笛卡尔索引（参见 `CartesianIndices`），一组 `N` 维笛卡尔索引可以通过 `LinearIndices(A)[i_1, i_2, ..., i_N]` 转换为线性索引（参见`LinearIndices`）。

```
julia> CartesianIndices(A)[5]
CartesianIndex(2, 2)

julia> LinearIndices(A)[2, 2]
5
```

需要注意的是，这些转换的性能存在很大的不对称性。将线性索引转换为一组笛卡尔索引需要做除法取余数，而相反的转换只是相乘和相加。在现代处理器中，整数除法比乘法慢 10-50 倍。虽然一些数组——比如 `Array` 本身——是使用线性内存块实现的，并在它们的实现中直接使用线性索引，但其他数组——比如 `Diagonal`——需要完整的笛卡尔索引集进行查找（请参阅 `IndexStyle` 以仔细推敲）。（译者注：OffsetArrays.jl 是 Julia 的一个包，支持矩阵的下标不从 1 开始）。

Warnings

When iterating over all the indices for an array, it is better to iterate over `eachindex(A)` instead of `1:length(A)`. Not only will this be faster in cases where `A` is `IndexCartesian`, but it will also support arrays with custom indexing, such as OffsetArrays. If only the values are needed, then is better to just iterate the array directly, i.e. `for a in A`.

省略和额外的索引

除了线性索引，在某些情况下，N 维数组的可能少于或多余 N 。

如果未索引的剩余维度的长度均为 1，则可以省略索引。换句话说，只有当那些省略的索引对于索引表达式只有一个可能的值时，才可以省略剩余索引。例如，一个大小为 (3, 4, 2, 1) 的四维数组可能只用三个索引进行索引，因为被跳过的维度（第四维）的长度为 1。请注意，线性索引优先级高于此规则。

```
julia> A = reshape(1:24, 3, 4, 2, 1)
3×4×2×1 reshape(::UnitRange{Int64}, 3, 4, 2, 1) with eltype Int64:
[:, :, 1, 1] =
 1  4  7  10
 2  5  8  11
 3  6  9  12

[:, :, 2, 1] =
 13  16  19  22
 14  17  20  23
 15  18  21  24

julia> A[1, 3, 2] # Omits the fourth dimension (length 1)
19

julia> A[1, 3] # Attempts to omit dimensions 3 & 4 (lengths 2 and 1)
ERROR: BoundsError: attempt to access 3×4×2×1 reshape(::UnitRange{Int64}, 3, 4, 2, 1) with eltype
↪  Int64 at index [1, 3]

julia> A[19] # Linear indexing
19
```

当用 A[] 省略全部索引时，这种语义提供了一种简单的习惯用法来检索数组中的唯一元素，同时确保只有一个元素。

类似地，如果超出数组维数的所有索引都是 1（或更一般地说是 axes(A, d) 的第一个也是唯一的元素，其中 d 是特定的维数），可以使用超过 N 维的索引。这允许向量像一列矩阵一样被索引，例如：

```
julia> A = [8,6,7]
3-element Vector{Int64}:
 8
 6
 7

julia> A[2,1]
6
```

19.9 迭代

迭代整个数组的推荐方法是

```
for a in A
    # Do something with the element a
end
```

```
for i in eachindex(A)
    # Do something with i and/or A[i]
end
```

当你需要每个元素的值而不是索引时，使用第一个构造。在第二个构造中，如果 A 是具有快速线性索引的数组类型，i 将是 Int; 否则，它将是一个 CartesianIndex：

```
julia> A = rand(4, 3);

julia> B = view(A, 1:3, 2:3);

julia> for i in eachindex(B)
           @show i
       end
i = CartesianIndex(1, 1)
i = CartesianIndex(2, 1)
i = CartesianIndex(3, 1)
i = CartesianIndex(1, 2)
i = CartesianIndex(2, 2)
i = CartesianIndex(3, 2)
```

Note

In contrast with `for i = 1:length(A)`, iterating with eachindex provides an efficient way to iterate over any array type. Besides, this also supports generic arrays with custom indexing such as OffsetArrays.

19.10 Array traits

如果你编写一个自定义的 AbstractArray 类型，你可以用以下代码指定它使用快速线性索引

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

此设置将导致 myArray 上的 eachindex 迭代使用整数。如果未指定此特征，则使用默认值 IndexCartesian()。

19.11 数组和向量化的算子与函数

以下运算符支持对数组操作

1. 一元运算符--, +
2. 二元运算符--, +, *, /, \, ^
3. 比较操作符--==, !=, ≈ (isapprox), ≉

另外，为了便于数学上和其他运算的向量化，Julia 提供了点语法（dot syntax） f.(args...)，例如，sin.(x) 或 min.(x,y)，用于数组或数组和标量的混合上的按元素运算（广播运算）；当与其他点调用（dot call）结合使用时，它们的额外优点是能「融合」到单个循环中，例如，sin.(cos.(x))。

此外，每个二元运算符支持相应的点操作版本，可以应用于此类融合 broadcasting 操作的数组（以及数组和标量的组合），例如 z .== sin.(x .* y)。

请注意，类似 == 的比较运算在作用于整个数组时，得到一个布尔结果。使用像 .== 这样的点运算符进行按元素的比较。（对于像 < 这样的比较操作，只有按元素运算的版本 .< 适用于数组。）

还要注意 max.(a,b) 和 maximum(a) 之间的区别，max.(a,b) 对 a 和 b 的每个元素 broadcasts max，maximum(a) 寻找在 a 中的最大值。min.(a,b) 和 minimum(a) 也有同样的关系。

19.12 广播

有时需要在不同尺寸的数组上执行元素对元素的操作，例如将矩阵的每一列加一个向量。一种低效的方法是将向量复制成矩阵的大小：

```
julia> a = rand(2, 1); A = rand(2, 3);

julia> repeat(a, 1, 3) + A
2×3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846
```

当维度较大的时候，这种方法将会十分浪费，所以 Julia 提供了广播 broadcast，它将会将参数中低维度的参数扩展，使得其与其他维度匹配，且不会使用额外的内存，并将所给的函数逐元素地应用。

```
julia> broadcast(+, a, A)
2×3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846

julia> b = rand(1,2)
1×2 Array{Float64,2}:
 0.867535  0.00457906

julia> broadcast(+, a, b)
2×2 Array{Float64,2}:
 1.71056  0.847604
 1.73659  0.873631
```

点运算符 如.+ 和.* 等价于 broadcast 调用（除了它们结合使用，如上所述）。还有一个 broadcast! 函数来指定一个明确的方式（也可以通过.= 赋值以融合方式访问）。事实上，f.(args...) 等价于 broadcast(f, args...)，提供了一种方便的语法来广播任何函数 (dot syntax)。嵌套的 “点运算符调用” f.(...)（包括对.+ 等的调用）自动融合 到单个 broadcast 调用中。

此外，broadcast 不限于数组（参见函数文档）；它还处理标量、元组和其它容器。默认情况下，只有一些参数类型被认为是标量，包括（但不限于）Numbers、Strings、Symbols、Types、Functions 和一些常见的单例，如 missing 和 nothing。所有其他参数都被迭代或逐个索引。

```
julia> convert.(Float32, [1, 2])
2-element Vector{Float32}:
 1.0
 2.0

julia> ceil.(UInt8, [1.2 3.4; 5.6 6.7])
```

```
2×2 Matrix{UInt8}:
 0x02  0x04
 0x06  0x07

julia> string.(1:3, ". ", ["First", "Second", "Third"])
3-element Vector{String}:
 "1. First"
 "2. Second"
 "3. Third"
```

有时，你希望一个通常参与广播的容器（如数组）受到“保护”，使其免受广播迭代其所有元素的行为的影响。通过将其放置在另一个容器中（如单个元素 Tuple），广播会将其视为单个值。

```
julia> ([1, 2, 3], [4, 5, 6]) .+ ([1, 2, 3],)
([2, 4, 6], [5, 7, 9])

julia> ([1, 2, 3], [4, 5, 6]) .+ tuple([1, 2, 3])
([2, 4, 6], [5, 7, 9])
```

19.13 实现

Julia 中的基本数组类型是抽象类型 AbstractArray{T,N}。它通过维数 N 和元素类型 T 进行参数化。AbstractVector 和 AbstractMatrix 是一维和二维情况下的别名。AbstractArray 对象的操作是使用更高级别的运算符和函数定义的，其方式独立于底层存储。这些操作可以正确地被用于任何特定数组实现的回退操作。

AbstractArray 类型包括任何类似数组的东西，它的实现可能与传统数组完全不同。例如，元素可能根据请求计算而不是存储。然而，任何具体的 AbstractArray{T,N} 类型通常应该至少实现 size(A)（返回一个 Int 元组），getindex(A,i) 和 getindex(A,i1,...,iN)；可变数组也应该实现 setindex!。建议这些操作具有常数时间复杂度，否则某些数组函数可能会出乎意料的慢。具体类型通常还应该提供一个 similar(A,T=eltype(A),dims=size(A)) 方法，用于为 copy 和其他不合适的操作。无论 AbstractArray{T,N} 在内部如何表示，T 都是由 整数索引（A[1, ..., 1]，当 A 非空）返回的对象类型并且 N 应该是 size 返回的元组的长度。有关自定义 AbstractArray 实现的更多详细信息，请参阅接口章节中的数组接口指南。

DenseArray 是 AbstractArray 的抽象子类型，旨在包括元素以列优先顺序连续存储的所有数组（请参阅 性能提示中的附加说明）。Array 类型是 DenseArray 的一个特定实例；Vector 和 Matrix 是一维和二维情况的别名。除了所有 AbstractArrays 所需的操作之外，很少有专门为 Array 实现的操作；大部分数组库都是以泛型方式实现的，允许所有自定义数组的行为类似。

SubArray 是 AbstractArray 的特例，它通过与原始数组共享内存而不是复制它来执行索引。使用view 函数创建 SubArray，它的调用方式与getindex 相同（作用于数组和一系列索引参数）。view 的结果看起来与 getindex 的结果相同，只是数据保持不变。view 将输入索引向量存储在 SubArray 对象中，该对象稍后可用于间接索引原始数组。通过将 @views 宏放在表达式或代码块之前，该表达式中的任何 array [...] 切片将被转换为创建一个 SubArray 视图。

BitArray 是节省空间“压缩”的布尔数组，每个比特（bit）存储一个布尔值。它们可以类似于 Array{Bool} 数组（每个字节（byte）存储一个布尔值），并且可以分别通过 Array(bitarray) 和 BitArray(array) 相互转换。

如果数组存储在内存中，其元素之间具有明确定义的间距（步长），则该数组是“等步长的”的。通过简单地传递其 pointer 和每个维度的步长，可以将有支持元素类型的等步长数组传递给外部（非 Julia）库，如 BLAS 或 LAPACK。stride(A, d) 是元素之间沿维度 d 的距离。例如，rand(5,7,2) 返回的

内置 Array 的元素按列优先顺序连续排列。这意味着第一个维度的步长——同一列中元素之间的间距——是 1：

```
julia> A = rand(5, 7, 2);

julia> stride(A, 1)
1
```

第二个维度的步长是同一行中元素之间的间距，跳过与单列（5）中的元素一样多的元素。类似地，在两个"页面"（在第三维中）之间跳转需要跳过 5*7 == 35 元素。这个数组的 strides 是这三个数字组成的元组：

```
julia> strides(A)
(1, 5, 35)
```

在这种特殊情况下，在内存中跳过的元素数与跳过的线性索引数相匹配。这仅适用于像 Array（和其他 DenseArray 子类型）这样的连续数组，通常情况下并非如此。具有范围索引的视图是 非连续等步长数组的一个很好的例子；考虑 V = @view A[1:3:4, 2:2:6, 2:-1:1]。这个视图 V 与 A 引用了相同的内存，但它跳过并重新排列了它的一些元素。V 的第一维的步幅是 3，因为我们只从原始数组中选择每第三行：

```
julia> V = @view A[1:3:4, 2:2:6, 2:-1:1];

julia> stride(V, 1)
3
```

这个视图类似于从我们原来的 A 中每隔一列选择一列——因此当在第二维的索引之间移动时，它需要跳过相当于两个五元素列的内容：

```
julia> stride(V, 2)
10
```

第三维很有趣因为它的顺序颠倒了！因此从第一" 页" 到第二页它必须在内存中到 *backwards*，所以它在这一维的 strides 是负的！

```
julia> stride(V, 3)
-35
```

这意味着 V 的 pointer 实际上指向 A 的内存块的中间，并且它在内存中指向元素是同时向后和向前的。有关定义你自己的跨距数组的更多详细信息，请参阅 等步长数组的接口指南。StridedVector 和 StridedMatrix 被认为是等步长数组的内置数组类型的方便别名，允许它们仅使用指针和步幅，来分派选择调用调整和优化后的 BLAS 和 LAPACK 函数。

需要强调的是 strides 是关于内存而不是索引中的偏移。如果你在找在线性（单索引）索引和笛卡尔（多索引）索引间切换的方法，见 LinearIndices 和 CartesianIndices。

Chapter 20

缺失值

Julia 支持表示统计意义上的缺失值，即某个变量在观察中没有可用值，但在理论上存在有效值的情况。缺失值由 `missing` 对象表示，该对象是 `Missing` 类型的唯一实例。`missing` 等价于 SQL 中的 `NULL` 以及 R 中的 `NA`，并在大多数情况下表现得与它们一样。

20.1 缺失值的传播

`missing` 值会自动在标准数学运算符和函数中传播。对于这类函数，其某个运算对象的值的不确定性会导致其结果的不确定性。在应用中，上述情形意味着若在数学操作中包括 `missing` 值，其结果也常常返回 `missing` 值：

```
julia> missing + 1
missing

julia> "a" * missing
missing

julia> abs(missing)
missing
```

由于 `missing` 是 Julia 中的正常对象，此传播规则仅在可实现该对象的函数中应用。This can be achieved by:

- adding a specific method defined for arguments of type `Missing`,
- accepting arguments of this type, and passing them to functions which propagate them (like standard math operators).

在包中定义新传播规则时，应考虑缺失值的传播是否具有实际意义，并在传播有意义时定义合适的方法。在某个不包含接受 `Missing` 类实参方法的函数中传递缺失值，则抛出 `MethodError`的报错，正如其它类型一样。

若希望函数不传播缺失值，可将其按照 Missings.jl 库中的 `passmissing` 函数封装起来。例如，将 `f(x)` 封装为 `passmissing(f)(x)`。

20.2 相等和比较运算符

标准相等和比较运算符遵循上面给出的传播规则。如果任何操作数是 `missing`，那么结果是 `missing`。这是一些例子：

```
julia> missing == 1
missing

julia> missing == missing
missing

julia> missing < 1
missing

julia> 2 >= missing
missing
```

特别要注意，missing == missing 返回 missing，所以 == 不能用于测试值是否为缺失值。要测试 x 是否为 missing，请用 ismissing(x)。

特殊的比较运算符 isequal 和 === 是传播规则的例外。它们总返回一个 Bool 值，即使存在 missing 值，并认为 missing 与 missing 相等且其与任何其它值不同。因此，它们可用于测试某个值是否为 missing：

```
julia> missing === 1
false

julia> isequal(missing, 1)
false

julia> missing === missing
true

julia> isequal(missing, missing)
true
```

isless 运算符是另一个例外：missing 被认为比任何其它值大。此运算符被用于 sort，因此 missing 值被放置在所有其它值之后：

```
julia> isless(1, missing)
true

julia> isless(missing, Inf)
false

julia> isless(missing, missing)
false
```

20.3 逻辑运算符

逻辑（或布尔）运算符 |、& 和 xor 是另一种特殊情况，因为它们只有在逻辑上是必需的时传递 missing 值。对于这些运算符来说，结果是否不确定取决于具体操作，其遵循三值逻辑的既定规则，这些规则由 SQL 中的 NULL 以及 R 中的 NA 实现。这个抽象的定义实际上对应于一系列相对自然的行为，这最好通过具体的例子来解释。

让我们用逻辑「或」运算符 | 来说明这个原理。按照布尔逻辑的规则，如果其中一个操作数是 true，则另一个操作数对结果没影响，结果总是 true：

```
julia> true | true
true

julia> true | false
true

julia> false | true
true
```

基于观察，我们可以得出结论，如果其中一个操作数是 true 而另一个是 missing，我们知道结果为 true，尽管另一个参数的实际值存在不确定性。如果我们能观察到第二个操作数的实际值，那么它只能是 true 或 false，在两种情况下结果都是 true。因此，在这种特殊情况下，值的缺失不会传播：

```
julia> true | missing
true

julia> missing | true
true
```

相反地，如果其中一个操作数是 false，结果可能是 true 或 false，这取决于另一个操作数的值。因此，如果一个操作数是 missing，那么结果也是 missing：

```
julia> false | true
true

julia> true | false
true

julia> false | false
false

julia> false | missing
missing

julia> missing | false
missing
```

逻辑「且」运算符 & 的行为与 | 运算符相似，区别在于当其中一个操作数为 false 时，值的缺失不会传播。例如，当第一个操作数是 false 时：

```
julia> false & false
false

julia> false & true
false

julia> false & missing
false
```

另一方面，当其中一个操作数为 true 时，值的缺失会传播。例如，当第一个操作数是 true 时：

```
julia> true & true
true

julia> true & false
false

julia> true & missing
missing
```

最后，逻辑「异或」运算符 xor 总传播 missing 值，因为两个操作数都总是对结果产生影响。还要注意，否定运算符 ! 在操作数是 missing 时返回 missing，这就像其它一元运算符。

20.4 流程控制和短路运算符

流程控制操作符，包括 if、while 和三元运算符 x ? y : z，不允许缺失值。这是因为如果我们能够观察实际值，它是 true 还是 false 是不确定的，这意味着我们不知道程序应该如何运行。一旦在以下上下文中遇到 missing 值，就会抛出 TypeError：

```
julia> if missing
           println("here")
       end
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

出于同样的原因，并与上面给出的逻辑运算符相反，短路布尔运算符 && 和 || 在当前操作数的值决定下一个操作数是否求值时不允许 missing 值。例如：

```
julia> missing || false
ERROR: TypeError: non-boolean (Missing) used in boolean context

julia> missing && false
ERROR: TypeError: non-boolean (Missing) used in boolean context

julia> true && missing && false
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

相反，如果无需 missing 值即可确定结果，则不会引发错误。代码在对 missing 操作数求值前短路，以及 missing 是最后一个操作数都是这种情况：

```
julia> true && missing
missing

julia> false && missing
false
```

20.5 包含缺失值的数组

包含缺失值的数组的创建就像其它数组：

```
julia> [1, missing]
2-element Vector{Union{Missing, Int64}}:
 1
  missing
```

如此示例所示，此类数组的元素类型为 Union{Missing, T}，其中 T 为非缺失值的类型。这反映了以下事实：数组条目可以具有类型 T（此处为 Int64）或类型 Missing。此类数组使用高效的内存存储，其等价于一个 Array{T} 和一个 Array{UInt8} 的组合，前者保存实际值，后者表示条目类型（即它是 Missing 还是 T）。

允许缺失值的数组可以使用标准语法构造。使用 Array{Union{Missing, T}}(missing, dims) 来创建填充缺失值的数组：

```
julia> Array{Union{Missing, String}}(missing, 2, 3)
2×3 Matrix{Union{Missing, String}}:
 missing  missing  missing
 missing  missing  missing
```

Note

使用 undef 或 similar 目前可能会给出一个填充有 missing 的数组，但这不是获得这样一个数组的正确方法。请使用如上所示的 missing 构造函数。

An array with element type allowing missing entries (e.g. Vector{Union{Missing, T}}) which does not contain any missing entries can be converted to an array type that does not allow for missing entries (e.g. Vector{T}) using convert. If the array contains missing values, a MethodError is thrown during conversion:

```
julia> x = Union{Missing, String}["a", "b"]
2-element Vector{Union{Missing, String}}:
 "a"
 "b"

julia> convert(Array{String}, x)
2-element Vector{String}:
 "a"
 "b"

julia> y = Union{Missing, String}[missing, "b"]
2-element Vector{Union{Missing, String}}:
 missing
 "b"

julia> convert(Array{String}, y)
ERROR: MethodError: Cannot `convert` an object of type Missing to an object of type String
```

20.6 跳过缺失值

由于 missing 会随着标准数学运算符传播，归约函数会在调用的数组包含缺失值时返回 missing：

```
julia> sum([1, missing])
missing
```

在这种情况下，使用 skipmissing 即可跳过缺失值：

```
julia> sum(skipmissing([1, missing]))
1
```

此函数方便地返回一个可高效滤除 missing 值的迭代器。因此，它可应用于所有支持迭代器的函数：

```
julia> x = skipmissing([3, missing, 2, 1])
skipmissing(Union{Missing, Int64}[3, missing, 2, 1])

julia> maximum(x)
3

julia> sum(x)
6

julia> mapreduce(sqrt, +, x)
4.146264369941973
```

通过在某数组中调用 skipmissing 生成的对象能以其在所属数组中的位置进行索引。对应缺失值的指标并不有效，若尝试使用之会丢出报错（它们在 keys 和 eachindex 中同样是被跳过的）。

```
julia> x[1]
3

julia> x[2]
ERROR: MissingException: the value at index (2,) is missing
[...]
```

这允许对索引进行操作的函数与 skipmissing 结合使用。搜索和查找函数尤其如此，它们返回对 skipmissing 函数返回的对象有效的索引，这些索引也是在父数组中匹配条目的索引。

```
julia> findall(==(1), x)
1-element Vector{Int64}:
 4

julia> findfirst(!iszero, x)
1

julia> argmax(x)
1
```

使用 collect 提取非 missing 值并将它们存储在一个数组里：

```
julia> collect(x)
3-element Vector{Int64}:
 3
 2
 1
```

20.7 数组上的逻辑运算

上面描述的逻辑运算符的三值逻辑也适用于针对数组的函数。因此，使用 == 运算符的数组相等性测试中，若在未知 missing 条目实际值时无法确定结果，就返回 missing。在实际应用中意味着，在待比较数组中所有非缺失值都相等，且某个或全部数组包含缺失值（也许在不同位置）时会返回 missing：

```
julia> [1, missing] == [2, missing]
false

julia> [1, missing] == [1, missing]
missing

julia> [1, 2, missing] == [1, missing, 2]
missing
```

对于单个值，isequal 会将 missing 值视为与其它 missing 值相等但与非缺失值不同：

```
julia> isequal([1, missing], [1, missing])
true

julia> isequal([1, 2, missing], [1, missing, 2])
false
```

函数 any 和 all 遵循三值逻辑的规则，会在结果无法被确定时返回 missing：

```
julia> all([true, missing])
missing

julia> all([false, missing])
false

julia> any([true, missing])
true

julia> any([false, missing])
missing
```

Chapter 21

网络和流

Julia 提供了一个功能丰富的接口来处理流式 I/O 对象，如终端、管道和 TCP 套接字。此接口虽然在系统级是异步的，但是其以同步的方式展现给程序员，通常也不需要考虑底层的异步操作。这是通过大量使用 Julia 协作线程（协程）功能实现的。

21.1 基础流 I/O

所有 Julia stream 都暴露了 `read` 和 `write` 方法，将 stream 作为它们的第一个参数，如：

```
julia> write(stdout, "Hello World");  # suppress return value 11 with ;
Hello World
julia> read(stdin, Char)

'\n': ASCII/Unicode U+000a (category Cc: Other, control)
```

注意，`write` 返回 11，字节数（`"Hello World"`）写入 `stdout`，但是返回值使用；抑制。

这里按了两次回车，以便 Julia 能够读取到换行符。正如你在这个例子中所看到的，`write` 以待写入的数据作为其第二个参数，而 `read` 以待读取的数据的类型作为其第二个参数。

例如，为了读取一个简单的字节数组，我们可以这样做：

```
julia> x = zeros(UInt8, 4)
4-element Array{UInt8,1}:
 0x00
 0x00
 0x00
 0x00

julia> read!(stdin, x)
abcd
4-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x64
```

但是，因为这有些繁琐，所以提供了几个方便的方法。例如，我们可以把上面的代码编写为：

```
julia> read(stdin, 4)
abcd
4-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x64
```

或者如果我们想要读取一整行：

```
julia> readline(stdin)
abcd
"abcd"
```

请注意，根据你的终端设置，你的 TTY 可能是行缓冲的，因此在数据发送给 Julia 前可能需要额外的回车。

若要读取 stdin 的每一行，可以使用 eachline：

```
for line in eachline(stdin)
    print("Found $line")
end
```

或者如果你想要按字符读取的话，使用 read ：

```
while !eof(stdin)
    x = read(stdin, Char)
    println("Found: $x")
end
```

21.2 文本 I/O

请注意，上面提到的 write 方法对二进制流进行操作。具体来说，值不会转换为任何规范的文本表示形式，而是按原样输出：

```
julia> write(stdout, 0x61);  # suppress return value 1 with ;
a
```

请注意，a 被 write 函数写入到 stdout 并且返回值为 1（因为 0x61 为一个字节）。

对于文本 I/O，请根据需要使用 print 或 show 方法（有关这两个方法之间的差异的详细讨论，请参阅它们的文档）：

```
julia> print(stdout, 0x61)
97
```

有关如何实现自定义类型的显示方法的更多信息，请参阅 自定义 pretty-printing。

21.3 IO 输出的上下文信息

有时，IO 输出可受益于将上下文信息传递到 show 方法的能力。`IOContext` 对象提供了将任意元数据与 IO 对象相关联的框架。例如，`:compact => true` 向 IO 对象添加一个参数来提示调用的 show 方法应该打印一个较短的输出（如果适用）。有关常用属性的列表，请参阅 `IOContext` 文档。

21.4 使用文件

You can write content to a file with the `write(filename::String, content)` method:

```
julia> write("hello.txt", "Hello, World!")
13
```

(13 is the number of bytes written.)

You can read the contents of a file with the `read(filename::String)` method, or `read(filename::String, String)` to the contents as a string:

```
julia> read("hello.txt", String)
"Hello, World!"
```

Advanced: streaming files

The `read` and `write` methods above allow you to read and write file contents. Like many other environments, Julia also has an open function, which takes a filename and returns an `IOStream` object that you can use to read and write things from the file. For example, if we have a file, `hello.txt`, whose contents are `Hello, World!`:

```
julia> f = open("hello.txt")
IOStream(<file hello.txt>)

julia> readlines(f)
1-element Array{String,1}:
 "Hello, World!"
```

若要写入文件，则可以带着 write（`"w"`）标志来打开它：

```
julia> f = open("hello.txt","w")
IOStream(<file hello.txt>)

julia> write(f,"Hello again.")
12
```

你如果在此刻检查 `hello.txt` 的内容，会注意到它是空的；改动实际上还没有写入到磁盘中。这是因为 `IOStream` 必须在写入实际刷新到磁盘前关闭：

```
julia> close(f)
```

再次检查 `hello.txt` 将显示其内容已被更改。

打开文件，对其内容执行一些操作，并再次关闭它是一种非常常见的模式。为了使这更容易，`open` 还有另一种调用方式，它以一个函数作为其第一个参数，以文件名作为其第二个参数，以该文件为参数调用该函数，然后再次关闭它。例如，给定函数：

```
function read_and_capitalize(f::IOStream)
    return uppercase(read(f, String))
end
```

可以调用：

```
julia> open(read_and_capitalize, "hello.txt")
"HELLO AGAIN."
```

来打开 `hello.txt`，对它调用 `read_and_capitalize`，关闭 `hello.txt` 并返回大写的内容。

为了避免被迫定义一个命名函数，你可以使用 do 语法，它可以动态地创建匿名函数：

```
julia> open("hello.txt") do f
           uppercase(read(f, String))
       end
"HELLO AGAIN."
```

21.5 一个简单的 TCP 示例

让我们直接进入一个 TCP 套接字相关的简单示例。此功能位于名为 Sockets 的标准库中。让我们先创建一个简单的服务器：

```
julia> using Sockets

julia> errormonitor(@async begin
           server = listen(2000)
           while true
               sock = accept(server)
               println("Hello World\n")
           end
       end)
Task (runnable) @0x00007fd31dc11ae0
```

对于那些熟悉 Unix 套接字 API 的人，这些方法名称会让人感觉很熟悉，可是它们的用法比原始的 Unix 套接字 API 要简单些。在本例中，首次调用 `listen` 会创建一个服务器，等待传入指定端口（2000）的连接。

```
julia> listen(2000) # 监听 (IPv4 下的) localhost:2000
Sockets.TCPServer(active)

julia> listen(ip"127.0.0.1",2000) # 等价于第一个
Sockets.TCPServer(active)

julia> listen(ip"::1",2000) # 监听 (IPv6 下的) localhost:2000
Sockets.TCPServer(active)
```

```
julia> listen(IPv4(0),2001) # 监听所有 IPv4 接口的端口 2001
Sockets.TCPServer(active)

julia> listen(IPv6(0),2001) # 监听所有 IPv6 接口的端口 2001
Sockets.TCPServer(active)

julia> listen("testsocket") # 监听 UNIX 域套接字
Sockets.PipeServer(active)

julia> listen("\\\\.\\pipe\\testsocket") # 监听 Windows 命名管道
Sockets.PipeServer(active)
```

请注意，最后一次调用返回的类型是不同的。这是因为此服务器不监听 TCP，而是监听命名管道（Windows）或 UNIX 域套接字。还请注意 Windows 命名管道格式必须具有特定的模式，即名称前缀（`\\.\pipe\`），以便唯一标识文件类型。TCP 和命名管道或 UNIX 域套接字之间的区别是微妙的，这与 `accept` 和 `connect` 方法有关。`accept` 方法检索到连接到我们刚创建的服务器的客户端的连接，而 `connect` 函数使用指定的方法连接到服务器。`connect` 函数接收与 `listen` 相同的参数，因此，假设环境（即 host、cwd 等）相同，你应该能够将相同的参数传递给 `connect`，就像你在监听建立连接时所做的那样。那么让我们尝试一下（在创建上面的服务器之后）：

```
julia> connect(2000)
TCPSocket(open, 0 bytes waiting)

julia> Hello World
```

不出所料，我们看到「Hello World」被打印出来。那么，让我们分析一下幕后发生的事情。在我们调用 `connect` 时，我们连接到刚刚创建的服务器。与此同时，accept 函数返回到新创建的套接字的服务器端连接，并打印「Hello World」来表明连接成功。

Julia 的强大优势在于，即使 I/O 实际上是异步发生的，API 也以同步方式暴露，我们不必担心回调，甚至不必确保服务器能够运行。在我们调用 `connect` 时，当前任务等待建立连接，并在这之后才继续执行。在此暂停中，服务器任务恢复执行（因为现在有一个连接请求是可用的），接受该连接，打印信息并等待下一个客户端。读取和写入以同样的方式运行。为了理解这一点，请考虑以下简单的 echo 服务器：

```
julia> errormonitor(@async begin
           server = listen(2001)
           while true
               sock = accept(server)
               @async while isopen(sock)
                   write(sock, readline(sock, keep=true))
               end
           end
       end)
Task (runnable) @0x00007fd31dc12e60

julia> clientside = connect(2001)
TCPSocket(RawFD(28) open, 0 bytes waiting)

julia> errormonitor(@async while isopen(clientside)
           write(stdout, readline(clientside, keep=true))
```

```
       end)
Task (runnable) @0x00007fd31dc11870

julia> println(clientside,"Hello World from the Echo Server")
Hello World from the Echo Server
```

与其他流一样，使用 `close` 即可断开该套接字：

```
julia> close(clientside)
```

21.6 解析 IP 地址

与 `listen` 方法不一致的 `connect` 方法之一是 `connect(host::String,port)`，它将尝试连接到由 `host` 参数给定的主机上的由 `port` 参数给定的端口。它允许你执行以下操作：

```
julia> connect("google.com", 80)
TCPSocket(RawFD(30) open, 0 bytes waiting)
```

此功能的基础是 `getaddrinfo`，它将执行适当的地址解析：

```
julia> getaddrinfo("google.com")
ip"74.125.226.225"
```

21.7 异步 I/O

`Base.read` 和 `Base.write` 的所有 I/O 操作都可以通过使用 coroutines 异步执行。你可以使用 `@async` 宏创建一个新的协程来读取或写入流：

```
julia> task = @async open("foo.txt", "w") do io
           write(io, "Hello, World!")
       end;

julia> wait(task)

julia> readlines("foo.txt")
1-element Array{String,1}:
 "Hello, World!"
```

通常会遇到您想要同时执行多个异步操作并等待它们全部完成的情况。你可以使用`@sync`宏，这会阻塞你的程序直到它所包裹的所有协程运行完毕。

```
julia> using Sockets

julia> @sync for hostname in ("google.com", "github.com", "julialang.org")
           @async begin
               conn = connect(hostname, 80)
               write(conn, "GET / HTTP/1.1\r\nHost:$(hostname)\r\n\r\n")
               readline(conn, keep=true)
```

```
            println("Finished connection to $(hostname)")
        end
    end
Finished connection to google.com
Finished connection to julialang.org
Finished connection to github.com
```

21.8 Multicast

Julia supports multicast over IPv4 and IPv6 using the User Datagram Protocol (UDP) as transport.

Unlike the Transmission Control Protocol (TCP), UDP makes almost no assumptions about the needs of the application. TCP provides flow control (it accelerates and decelerates to maximize throughput), reliability (lost or corrupt packets are automatically retransmitted), sequencing (packets are ordered by the operating system before they are given to the application), segment size, and session setup and teardown. UDP provides no such features.

A common use for UDP is in multicast applications. TCP is a stateful protocol for communication between exactly two devices. UDP can use special multicast addresses to allow simultaneous communication between many devices.

Receiving IP Multicast Packets

To transmit data over UDP multicast, simply `recv` on the socket, and the first packet received will be returned. Note that it may not be the first packet that you sent however!

```
using Sockets
group = ip"228.5.6.7"
socket = Sockets.UDPSocket()
bind(socket, ip"0.0.0.0", 6789)
join_multicast_group(socket, group)
println(String(recv(socket)))
leave_multicast_group(socket, group)
close(socket)
```

Sending IP Multicast Packets

To transmit data over UDP multicast, simply `send` to the socket. Notice that it is not necessary for a sender to join the multicast group.

```
using Sockets
group = ip"228.5.6.7"
socket = Sockets.UDPSocket()
send(socket, group, 6789, "Hello over IPv4")
close(socket)
```

IPv6 Example

This example gives the same functionality as the previous program, but uses IPv6 as the network-layer protocol.

Listener:

```
using Sockets
group = Sockets.IPv6("ff05::5:6:7")
socket = Sockets.UDPSocket()
bind(socket, Sockets.IPv6("::"), 6789)
join_multicast_group(socket, group)
println(String(recv(socket)))
leave_multicast_group(socket, group)
close(socket)
```

Sender:

```
using Sockets
group = Sockets.IPv6("ff05::5:6:7")
socket = Sockets.UDPSocket()
send(socket, group, 6789, "Hello over IPv6")
close(socket)
```

Chapter 22

并行计算

Julia 支持这四类并发和并行编程：

1. **异步“任务”或协程**：

   Julia Tasks 允许暂停和恢复 I/O、事件处理、生产者-消费者进程和类似模式的计算。Tasks 可以通过 `wait` 和 `fetch` 等操作进行同步，并通过 `Channel` 进行通信。虽然严格来说不是并行计算，但 Julia 允许在多个线程上调度 `Task`。

2. **多线程**：

   Julia 的多线程提供了在多个线程、CPU 内核或共享内存上同时调度任务的能力。这通常是在个人 PC 或单个大型多核服务器上获得并行性的最简单方法。Julia 的多线程是可组合的。当一个多线程函数调用另一个多线程函数时，Julia 将在可用资源上全局调度所有线程，而不会超额使用。

3. **分布式计算**:

   分布式计算运行多个具有独立内存空间的 Julia 进程。这些可以在同一台计算机或多台计算机上。`Distributed` 标准库提供了远程执行 Julia 函数的能力。使用这个基本构建块，可以构建许多不同类型的分布式计算抽象。像 `DistributedArrays.jl` 这样的包就是这种抽象的一个示例。另一方面，像 `MPI.jl` 和 `Elemental.jl` 这样的包提供对现有 MPI 生态库的访问。

4. **GPU 计算**:

   Julia GPU 编译器提供了在 GPU 上本地运行 Julia 代码的能力。有一个针对 GPU 的丰富的 Julia 软件包生态系统。JuliaGPU.org 网站提供了功能列表、支持的 GPU、相关包和文档。

Chapter 23

异步编程

当程序需要与外部世界交互时，例如通过互联网与另一台机器通信时，程序中的操作可能需要以无法预测的顺序发生。假设你的程序需要下载一个文件。我们想启动下载操作，在等待下载完成的同时执行其他操作，然后在空闲时继续执行下载文件的代码。这种场景属于异步编程，有时也称为并发编程（因为从概念上讲，同时发生多种事情）。

为了解决这些可能的情况，Julia 提供了任务 Task（也有其他几个名称，例如对称协程、轻量级线程、协作多任务处理或 one-shot continuations）。当一项计算工作（实际上，执行特定功能）被指定为 Task 时，可以通过切换到另一个 Task 来中断它。最初的 Task 稍后可以恢复，此时它将从上次中断的地方开始。初看这似乎类似于函数调用。但是，有两个关键区别。首先，切换任务不占用任何空间，因此可以在不消耗调用堆栈的情况下进行任意数量的任务切换。其次，任务之间的切换可以以任何顺序发生，这与函数调用不同，在函数调用中，被调用的函数必须在返回到调用函数之前完成执行。

23.1 基本 Task 操作

你可以将 Task 视为要执行的计算工作单元的句柄。它有一个创建-开始-运行-结束的生命周期。Task 是通过在要运行的 0 参数函数上调用 Task 构造函数来创建的，或者使用 @task 宏：

```
julia> t = @task begin; sleep(5); println("done"); end
Task (runnable) @0x00007f13a40c0eb0
```

@task x 等价于 Task(()->x)。

此任务将等待五秒钟，然后打印 done。但是，它还没有开始运行。我们可以随时通过调用 schedule 来运行它：

```
julia> schedule(t);
```

如果你在 REPL 中尝试这个，你会看到 schedule 立即有返回值。那是因为它只是将 t 添加到要运行的内部任务队列中。然后，REPL 将打印下一个提示并等待更多输入。等待键盘输入为其他任务提供了运行的机会，因此此时 t 将启动。t 调用 sleep，它设置一个计时器并停止执行。如果已经安排了其他任务，那么它们就可以运行了。五秒后，计时器触发并重新启动 t，你将看到打印的 done。然后 t 执行完毕了。

wait 函数会阻塞调用任务，直到其他任务完成。例如，如果输入：

```
julia> schedule(t); wait(t)
```

在下一个输入提示出现之前，你将看到五秒钟的停顿，而不是只调用 schedule。那是因为 REPL 等待 t 完成之后才继续。

一般来说，创建一个任务会想立即执行它，为此提供了宏 @async –- @async x 等价于 schedule(@task x)。

23.2 在 Channel 中进行通信

在某些问题中，所需的各种工作并不是通过函数调用自然关联的；在需要完成的工作中没有明显的"调用者"或"被调用者"。一个典型的例子是生产者-消费者问题，其中一个复杂的过程正在生成值，而另一个复杂的过程正在消耗它们。消费者不能简单地调用生产者函数来获取一个值，因为生产者可能有更多的值要生成，因此可能还没有准备好返回。对于任务，生产者和消费者都可以根据需要运行，根据需要来回传递值。

Julia 提供了 Channel 机制来解决这个问题。一个 Channel 是一个先进先出的队列，允许多个 Task 对它可以进行读和写。

让我们定义一个生产者任务，调用 put! 来生产数值。为了消费数值，我们需要对生产者开始新任务进行排班。可以使用一个特殊的 Channel 组件来运行一个与其绑定的 Task，它能接受单参数函数作为其参数，然后可以用 take! 从 Channel 对象里不断地提取值：

```
julia> function producer(c::Channel)
           put!(c, "start")
           for n=1:4
               put!(c, 2n)
           end
           put!(c, "stop")
       end;

julia> chnl = Channel(producer);

julia> take!(chnl)
"start"

julia> take!(chnl)
2

julia> take!(chnl)
4

julia> take!(chnl)
6

julia> take!(chnl)
8

julia> take!(chnl)
"stop"
```

一种思考这种行为的方式是，"生产者"能够多次返回。在两次调用 put! 之间，生产者的执行是挂起的，此时由消费者接管控制。

返回的 Channel 可以被用作一个 for 循环的迭代对象，此时循环变量会依次取到所有产生的值。当 Channel 关闭时，循环就会终止。

```
julia> for x in Channel(producer)
           println(x)
       end
start
2
4
6
8
stop
```

注意我们并不需要显式地在生产者中关闭 Channel。这是因为 Channel 对 Task 的绑定同时也意味着 Channel 的生命周期与绑定的 Task 一致。当 Task 结束时，Channel 对象会自动关闭。多个 Channel 可以绑定到一个 Task，反之亦然。

Task 构造函数需要一个不带参数的函数，而创建任务绑定的 channel 的 Channel 方法需要一个接受 Channel类型的单个参数的函数。一个常见的模式是对生产者进行参数化，在这种情况下，需要一个偏函数来创建一个 0 或 1 个参数 匿名函数。

对于 Task 对象，可以直接用，也可以为了方便用宏。

```
function mytask(myarg)
    ...
end

taskHdl = Task(() -> mytask(7))
# or, equivalently
taskHdl = @task mytask(7)
```

为了安排更高级的工作分配模式，bind 和 schedule 可以与 Task 和 Channel 构造函数配合使用，显式地连接一些 Channel 和生产者或消费者 Task。

更多关于 Channel 的知识

一个管道可以形象得看做是一个管子，一端可读，另一端可写：

- 不同的 task 可以通过 put! 往同一个 channel 并发地写入。
- 不同的 task 也可以通过 take! 从同一个 channel 并发地取数据
- 举个例子：

  ```
  # Given Channels c1 and c2,
  c1 = Channel(32)
  c2 = Channel(32)

  # and a function `foo` which reads items from c1, processes the item read
  # and writes a result to c2,
  function foo()
      while true
          data = take!(c1)
  ```

```
        [...]               # process data
        put!(c2, result)    # write out result
    end
end

# we can schedule `n` instances of `foo` to be active concurrently.
for _ in 1:n
    errormonitor(@async foo())
end
```

- Channel 可以通过 `Channel{T}(sz)` 构造，得到的 channel 只能存储类型 `T` 的数据。如果 `T` 没有指定，那么 channel 可以存任意类型。`sz` 表示该 channel 能够存储的最大元素个数。比如 `Channel(32)` 得到的 channel 最多可以存储 32 个元素。而 `Channel{MyType}(64)` 则可以最多存储 64 个 `MyType` 类型的数据。
- 如果一个 `Channel` 是空的，读取的 task(即执行 `take!` 的 task) 会被阻塞直到有新的数据准备好了。
- 如果一个 `Channel` 是满的，那么写入的 task(即执行 `put!` 的 task) 则会被阻塞，直到 Channel 有空余。
-
- 一个 `Channel` 一开始处于开启状态，也就是说可以被 `take!` 读取和 `put!` 写入。`close` 会关闭一个 `Channel`，对于一个已经关闭的 `Channel`，`put!` 会失败，例如：

```
julia> c = Channel(2);

julia> put!(c, 1) # `put!` on an open channel succeeds
1

julia> close(c);

julia> put!(c, 2) # `put!` on a closed channel throws an exception.
ERROR: InvalidStateException: Channel is closed.
Stacktrace:
[...]
```

- `take!` 和 `fetch` (只读取，不会将元素从 channel 中删掉) 仍然可以从一个已经关闭的 channel 中读数据，直到 channel 被取空了为止。继续上面的例子：

```
julia> fetch(c) # Any number of `fetch` calls succeed.
1

julia> fetch(c)
1

julia> take!(c) # The first `take!` removes the value.
1

julia> take!(c) # No more data available on a closed channel.
ERROR: InvalidStateException: Channel is closed.
Stacktrace:
[...]
```

考虑这样一个用 channel 做 task 之间通信的例子。首先，起 4 个 task 来处理一个 jobs channel 中的数据。jobs 中的每个任务通过 job_id 来表示，然后每个 task 模拟读取一个 job_id，然后随机等待一会儿，然后往一个 results channel 中写入一个元组，它分别包含 job_id 和执行的时间，最后将结果打印出来：

```
julia> const jobs = Channel{Int}(32);

julia> const results = Channel{Tuple}(32);

julia> function do_work()
           for job_id in jobs
               exec_time = rand()
               sleep(exec_time)                # simulates elapsed time doing actual work
                                               # typically performed externally.
               put!(results, (job_id, exec_time))
           end
       end;

julia> function make_jobs(n)
           for i in 1:n
               put!(jobs, i)
           end
       end;

julia> n = 12;

julia> errormonitor(@async make_jobs(n)); # feed the jobs channel with "n" jobs

julia> for i in 1:4 # start 4 tasks to process requests in parallel
           errormonitor(@async do_work())
       end

julia> @elapsed while n > 0 # print out results
           job_id, exec_time = take!(results)
           println("$job_id finished in $(round(exec_time; digits=2)) seconds")
           global n = n - 1
       end
4 finished in 0.22 seconds
3 finished in 0.45 seconds
1 finished in 0.5 seconds
7 finished in 0.14 seconds
2 finished in 0.78 seconds
5 finished in 0.9 seconds
9 finished in 0.36 seconds
6 finished in 0.87 seconds
8 finished in 0.79 seconds
10 finished in 0.64 seconds
12 finished in 0.5 seconds
11 finished in 0.97 seconds
0.029772311
```

不用 errormonitor(t)，一个更稳健的解决方案是使用 bind(results, t)，这不仅会记录任何意外故障，还会强制相关资源关闭并向上抛出错误。

23.3 更多任务操作

任务操作建立在称为 yieldto 的底层原始运算上。yieldto(task, value) 挂起当前 task，然后切换到指定的 task，并使该任务的最后一个 yieldto 调用返回指定的 value。请注意，yieldto 是使用任务式流程控制所需的唯一操作；我们总是切换到不同的任务，而不是调用和返回。这就是为什么这个特性也被称为"对称协程"；每个任务都使用相同的机制来回切换。

yieldto 功能强大，但大多数 Task 的使用都不会直接调用它。思考为什么会这样。如果你切换当前 Task，你很可能会在某个时候想切换回来。但知道什么时候切换回来和那个 Task 负责切换回来需要大量的协调。例如，put! 和 take! 是阻塞操作，当在渠道环境中使用时，维持状态以记住消费者是谁。不需要人为地记录消费 Task，正是使得 put! 比底层 yieldto 易用的原因。

除了 yieldto 之外，也需要一些其它的基本函数来更高效地使用 Task。

- current_task 获取当前运行 Task 的索引。
- istaskdone 查询一个 Task 是否退出.
- istaskstarted 查询一个 Task 是否已经开始运行。
- task_local_storage 操纵针对当前 Task 的键值存储。

23.4 Task 和事件

多数 Task 切换是在等待如 I/O 请求的事件，由 Julia Base 里的调度器执行。调度器维持一个可运行 Task 的队列，并执行一个事件循环，来根据例如收到消息等外部事件来重启 Task。

等待一个事件的基本函数是 wait。很多对象都实现了 wait 函数；例如，给定一个 Process 对象，wait 将等待它退出。wait 通常是隐式的，例如，wait 可能发生在调用 read 时等待数据可用。

在所有这些情况下，wait 最终会操作一个 Condition 对象，由它负责排队和重启 Task。当 Task 在一个 Condition 上调用 wait 时，该 Task 就被标记为不可执行，加到条件的队列中，并切回调度器。调度器将选择另一个 Task 来运行，或者阻止外部事件的等待。如果所有运行良好，最终一个事件处理器将在这个条件下调用 notify，使得等待该条件的 Task 又变成可运行。

通过调用 Task 显式创建的任务，一开始并不被调度器知道。这允许你根据需要使用 yieldto 手动管理任务。但是，当此类任务等待事件时，它仍会在事件发生时自动重新启动，正如你所期望。

Chapter 24

多线程

访问此 博客文章 以了解 Julia 多线程特性。

24.1 启用 Julia 多线程

Julia 默认启动一个线程执行代码，这点可以通过 `Threads.nthreads()` 来确认：

```
julia> Threads.nthreads()
1
```

执行线程的数量通过使用`-t`/`--threads` 命令行参数或使用`JULIA_NUM_THREADS` 环境变量。当两者都被指定时，`-t`/`--threads` 优先级更高。

The number of threads can either be specified as an integer (`--threads=4`) or as `auto` (`--threads=auto`), where `auto` tries to infer a useful default number of threads to use (see Command-line Options for more details).

Julia 1.5

`-t`/`--threads` 命令行参数至少需要 Julia 1.5。在旧版本中，你必须改用环境变量。

Julia 1.7

Using `auto` as value of the environment variable `JULIA_NUM_THREADS` requires at least Julia 1.7. In older versions, this value is ignored.

让我们以 4 个线程启动 Julia：

```
$ julia --threads 4
```

现在确认下确实有 4 个线程：

```
julia> Threads.nthreads()
4
```

不过我们现在是在 master 线程，用 `Threads.threadid` 确认下：

```
julia> Threads.threadid()
1
```

> **Note**
>
> 如果你更喜欢使用环境变量，可以按如下方式设置它 Bash (Linux/macOS):
>
> ```
> export JULIA_NUM_THREADS=4
> ```
>
> C shell on Linux/macOS, CMD on Windows:
>
> ```
> set JULIA_NUM_THREADS=4
> ```
>
> Powershell on Windows:
>
> ```
> $env:JULIA_NUM_THREADS=4
> ```
>
> Note that this must be done *before* starting Julia.

> **Note**
>
> 使用 `-t`/`--threads` 指定的线程数传播到使用 `-p`/`--procs` 或 `--machine-file` 命令行选项产生的工作进程。例如，`julia -p2 -t2` 产生 1 个主进程和 2 个工作进程，并且所有三个进程都启用了 2 个线程。要对工作线程进行更细粒度的控制，请使用 `addprocs` 并将 `-t`/`--threads` 作为 `exeflags` 传递。

Multiple GC Threads

The Garbage Collector (GC) can use multiple threads. The amount used is either half the number of compute worker threads or configured by either the `--gcthreads` command line argument or by using the `JULIA_NUM_GC_THREADS` environment variable.

> **Julia 1.10**
>
> The `--gcthreads` command line argument requires at least Julia 1.10.

24.2 Threadpools

When a program's threads are busy with many tasks to run, tasks may experience delays which may negatively affect the responsiveness and interactivity of the program. To address this, you can specify that a task is interactive when you `Threads.@spawn` it:

```
using Base.Threads
@spawn :interactive f()
```

Interactive tasks should avoid performing high latency operations, and if they are long duration tasks, should yield frequently.

Julia may be started with one or more threads reserved to run interactive tasks:

```
$ julia --threads 3,1
```

The environment variable `JULIA_NUM_THREADS` can also be used similarly:

```
export JULIA_NUM_THREADS=3,1
```

This starts Julia with 3 threads in the `:default` threadpool and 1 thread in the `:interactive` threadpool:

```
julia> using Base.Threads

julia> nthreadpools()
2

julia> threadpool() # the main thread is in the interactive thread pool
:interactive

julia> nthreads(:default)
3

julia> nthreads(:interactive)
1

julia> nthreads()
3
```

Note

The zero-argument version of `nthreads` returns the number of threads in the default pool.

Note

Depending on whether Julia has been started with interactive threads, the main thread is either in the default or interactive thread pool.

Either or both numbers can be replaced with the word `auto`, which causes Julia to choose a reasonable default.

24.3 Communication and synchronization

Although Julia's threads can communicate through shared memory, it is notoriously difficult to write correct and data-race free multi-threaded code. Julia's `Channel`s are thread-safe and may be used to communicate safely.

数据竞争自由

你有责任确保程序没有数据竞争，如果你不遵守该要求，则不能假设这里承诺的任何内容。观察到的结果可能是反直觉的。

为了确保这一点，最好的办法是获取多线程同时访问的数据的锁。例如，在大多数情况下，你应该使用以下代码模板：

```
julia> lock(lk) do
           use(a)
       end

julia> begin
           lock(lk)
           try
               use(a)
           finally
               unlock(lk)
           end
       end
```

其中 lk 是一个锁（例如 ReentrantLock()），a 是数据。

此外，Julia 在出现数据竞争时不是内存安全的。如果另一个线程可能会写入数据，则在读取任何数据时都要非常小心！相反，在更改其他线程访问的数据（例如分配给全局或闭包变量）时，请始终使用上述锁模式。

```
Thread 1:
global b = false
global a = rand()
global b = true

Thread 2:
while !b; end
bad_read1(a) # it is NOT safe to access `a` here!

Thread 3:
while !@isdefined(a); end
bad_read2(a) # it is NOT safe to access `a` here
```

24.4 @threads 宏

下面用一个简单的例子测试我们原生的线程，首先创建一个全零的数组：

```
julia> a = zeros(10)
10-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

现在用 4 个线程模拟操作这个数组，每个线程往对应的位置写入线程 ID。

Julia 用 `Threads.@threads` 宏实现并行循环，该宏加在 for 循环前面，提示 Julia 循环部分是一个多线程的区域：

```
julia> Threads.@threads for i = 1:10
           a[i] = Threads.threadid()
       end
```

根据线程调度，迭代在各线程中进行拆分，之后各线程将自己的线程 ID 写入对应区域。

```
julia> a
10-element Vector{Float64}:
 1.0
 1.0
 1.0
 2.0
 2.0
 2.0
 3.0
 3.0
 4.0
 4.0
```

注意 `Threads.@threads` 并没有一个像 `@distributed` 一样的可选的 reduction 参数。

Using `@threads` without data races

Taking the example of a naive sum

```
julia> function sum_single(a)
           s = 0
           for i in a
               s += i
           end
           s
       end
sum_single (generic function with 1 method)

julia> sum_single(1:1_000_000)
500000500000
```

Simply adding `@threads` exposes a data race with multiple threads reading and writing `s` at the same time.

```
julia> function sum_multi_bad(a)
           s = 0
           Threads.@threads for i in a
               s += i
           end
           s
       end
sum_multi_bad (generic function with 1 method)

julia> sum_multi_bad(1:1_000_000)
70140554652
```

Note that the result is not 500000500000 as it should be, and will most likely change each evaluation.

To fix this, buffers that are specific to the task may be used to segment the sum into chunks that are race-free. Here `sum_single` is reused, with its own internal buffer `s`, and vector `a` is split into `nthreads()` chunks for parallel work via `nthreads()` `@spawn`-ed tasks.

```
julia> function sum_multi_good(a)
           chunks = Iterators.partition(a, length(a) ÷ Threads.nthreads())
           tasks = map(chunks) do chunk
               Threads.@spawn sum_single(chunk)
           end
           chunk_sums = fetch.(tasks)
           return sum_single(chunk_sums)
       end
sum_multi_good (generic function with 1 method)

julia> sum_multi_good(1:1_000_000)
500000500000
```

> **Note**
>
> Buffers should not be managed based on `threadid()` i.e. `buffers = zeros(Threads.nthreads())` because concurrent tasks can yield, meaning multiple concurrent tasks may use the same buffer on a given thread, introducing risk of data races. Further, when more than one thread is available tasks may change thread at yield points, which is known as task migration.

Another option is the use of atomic operations on variables shared across tasks/threads, which may be more performant depending on the characteristics of the operations.

24.5 原子操作

Julia 支持访问和修改值的**原子**操作，即以一种线程安全的方式来避免竞态条件。一个值（必须是基本类型的，primitive type）可以通过 `Threads.Atomic` 来包装起来从而支持原子操作。下面看个例子：

```
julia> i = Threads.Atomic{Int}(0);

julia> ids = zeros(4);

julia> old_is = zeros(4);

julia> Threads.@threads for id in 1:4
           old_is[id] = Threads.atomic_add!(i, id)
           ids[id] = id
       end

julia> old_is
4-element Vector{Float64}:
 0.0
 1.0
 7.0
 3.0

julia> i[]
```

```
 10

julia> ids
4-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0
```

如果不加 `Atomic` 的话，那么会因为竞态条件而得到错误的结果，下面是一个没有避免竞态条件的例子：

```
julia> using Base.Threads

julia> Threads.nthreads()
4

julia> acc = Ref(0)
Base.RefValue{Int64}(0)

julia> @threads for i in 1:1000
          acc[] += 1
       end

julia> acc[]
926

julia> acc = Atomic{Int64}(0)
Atomic{Int64}(0)

julia> @threads for i in 1:1000
          atomic_add!(acc, 1)
       end

julia> acc[]
1000
```

24.6 field 粒度的原子操作

我们还可以使用`@atomic`、`@atomicswap`和 `@atomicreplace` 宏在更细粒度的级别上使用原子。

内存模型的具体细节和设计的其他细节写在 Julia Atomics Manifesto中，稍后将正式发布。

结构体声明中的任何字段都可以用 `@atomic` 修饰，然后任何写入也必须用 `@atomic` 标记，并且必须使用定义的原子顺序之一(`:monotonic`, `:acquire`, `:release`, `:acquire_release`, 或 `:sequentially_consistent`)。对原子字段的任何读取也可以使用原子排序约束进行注释，或者如果未指定，将使用单调（宽松）排序完成。

Julia 1.7

field 粒度的原子操作至少需要 Julia 1.7.

24.7 副作用和可变的函数参数

使用多线程时，我们必须小心使用非 纯 的函数，因为我们可能会得到错误的答案。例如，按照惯例具有 名称以！结尾 的函数会修改它们的参数，因此不是纯函数。

24.8 @threadcall

外部库，例如通过 `ccall` 调用的库，给 Julia 基于任务的 I/O 机制带来了问题。如果 C 库执行阻塞操作，这会阻止 Julia 调度程序执行任何其他任务，直到调用返回。（例外情况是调用回调到 Julia 的自定义 C 代码，然后它可能会 yield，或者调用 `jl_yield()` 的 C 代码，`jl_yield` 是 `yield` 的 C 等价物。）

`@threadcall` 宏提供了一种避免在这种情况下停止执行的方法。它调度一个 C 函数以在单独的线程中执行。为此使用默认大小为 4 的线程池。线程池的大小由环境变量 `UV_THREADPOOL_SIZE` 控制。在等待空闲线程时，以及一旦线程可用后的函数执行期间，请求任务（在主 Julia 事件循环上）让步给其他任务。注意，`@threadcall` 在执行完成之前不会返回。因此，从用户的角度来看，它与其他 Julia API 一样是一个阻塞调用。

非常关键的一点是，被调用的函数不会再调用回 Julia。

`@threadcall` 在 Julia 未来的版本中可能会被移除或改变。

24.9 注意！

此时，如果用户代码没有数据竞争，Julia 运行时和标准库中的大多数操作都可以以线程安全的方式使用。然而，在某些领域，稳定线程支持的工作正在进行中。多线程编程有许多内在的困难，如果使用线程的程序表现出异常或与预期不符的行为（例如崩溃或神秘的结果），通常应该首先怀疑线程交互。

在 Julia 中使用线程时需要注意以下这些特定的限制和警告：

- 如果多个线程同时使用基本容器类型，且至少有一个线程修改容器器时，需要手动加锁（常见示例包括 `push!` 数组，或将项插入 `Dict`）。
- `@spawn` 使用的时间表是不确定的，不应依赖。
- 计算绑定、非内存分配任务可以防止垃圾回收在其他正在分配内存的线程中运行。在这些情况下，可能需要手动调用 `GC.safepoint()` 以允许 GC 运行。

  该限制在未来会被移除。
- 避免并行运行顶层操作，例如，`include` 或 `eval` 评估类型、方法和模块定义。
- 请注意，如果启用线程，则库注册的终结器可能会中断。这可能需要在整个生态系统中进行一些过渡工作，然后才能放心地广泛采用线程。有关更多详细信息，请参阅下一节。

24.10 Task Migration

After a task starts running on a certain thread it may move to a different thread if the task yields.

Such tasks may have been started with `@spawn` or `@threads`, although the `:static` schedule option for `@threads` does freeze the threadid.

This means that in most cases `threadid()` should not be treated as constant within a task, and therefore should not be used to index into a vector of buffers or stateful objects.

Julia 1.7

Task migration was introduced in Julia 1.7. Before this tasks always remained on the same thread that they were started on.

24.11 终结器的安全使用

因为终结器可以中断任何代码，所以它们在如何与任何全局状态交互时必须非常小心。不幸的是，使用终结器的主要原因是更新全局状态（纯函数作为终结器通常毫无意义）。这让我们陷入了一个难题。有几种方法可以处理这个问题：

1. 当单线程时，代码可以调用内部 `jl_gc_enable_finalizers` C 函数以防止在关键区域内调度终结器。在内部，这在某些函数（例如我们的 C locks）中使用，以防止在执行某些操作（增量包加载、代码生成等）时发生递归。锁和此标志的组合可用于使终结器安全。

2. Base 在几个地方采用的第二种策略是显式延迟终结器，直到它可以非递归地获取其锁。以下示例演示了如何将此策略应用于 `Distributed.finalize_ref`：

```julia
function finalize_ref(r::AbstractRemoteRef)
    if r.where > 0 # Check if the finalizer is already run
        if islocked(client_refs) || !trylock(client_refs)
            # delay finalizer for later if we aren't free to acquire the lock
            finalizer(finalize_ref, r)
            return nothing
        end
        try # `lock` should always be followed by `try`
            if r.where > 0 # Must check again here
                # Do actual cleanup here
                r.where = 0
            end
        finally
            unlock(client_refs)
        end
    end
    nothing
end
```

3. 相关的第三种策略是使用不需要 yield 的队列。我们目前没有在 Base 中实现无锁队列，但 `Base.IntrusiveLinkedListSynchronized{T}` 是合适的。这通常是用于带有事件循环的代码的好策略。例如，这个策略被 `Gtk.jl` 用来管理生命周期引用计数。在这种方法中，我们不会在终结器内部做任何显式工作，而是将其添加到队列中以在更安全的时间运行。事实上，Julia 的任务调度器已经使用了这种方法，因此将终结器定义为 `x -> @spawn do_cleanup(x)` 就是这种方法的一个示例。但是请注意，这并不控制 `do_cleanup` 在哪个线程上运行，因此 `do_cleanup` 仍需要获取锁。如果你实现自己的队列，则不必如此，因为你只能明确地从线程中排出该队列。

Chapter 25

多进程和分布式计算

分布式内存并行计算的实现由模块 Distributed 作为 Julia 附带的标准库的一部分提供。

大多数现代计算机都拥有不止一个 CPU，而且多台计算机可以组织在一起形成一个集群。借助多个 CPU 的计算能力，许多计算过程能够更快地完成，这其中影响性能的两个主要因素分别是：CPU 自身的速度以及它们访问内存的速度。显然，在一个集群中，一个 CPU 访问同一个节点的 RAM 速度是最快的，不过令人吃惊的是，在一台典型的多核笔记本电脑上，由于访问主存和缓存的速度存在差别，类似的现象也会存在。因此，一个良好的多进程环境应该能够管理好某一片内存区域“所属”的 CPU。Julia 提供的多进程环境是基于消息传递来实现的，可以做到同时让程序在多个进程的不同内存区域中运行。

Julia 的消息传递实现不同于其他环境，例如 MPI[1]。Julia 中的通信通常是“单方面的”，这意味着程序员只需在双进程操作中显式管理一个进程。此外，这些操作通常看起来不像“消息发送”和“消息接收”，而是类似于更高级别的操作，例如调用用户函数。

Julia 中的分布式编程基于两个基本概念：**远程引用** (*remote references*) 和**远程调用** (*remote calls*)。远程引用是一个对象，任意一个进程可以通过它访问存储在某个特定进程上的对象。远程调用指是某个进程发起的执行函数的请求，该函数会在另一个（也可能是同一个）进程中执行。

远程引用有两种形式：Future 和 RemoteChannel。

远程调用返回 Future 作为其结果。远程调用立即返回；当远程调用发生在其他地方后，发出调用的进程继续执行下一个操作。你可以通过在返回的 Future 上调用 wait 来等待远程调用完成，并且可以使用 fetch。

对于 RemoteChannel 而言，它可以被反复写入。例如，多个进程可以通过引用同一个远程 Channel 来协调相互之间的操作。

每个进程都有一个关联的标识符。提供交互式 Julia 提示符的进程的 id 总是等于 1。默认情况下用于并行操作的进程被称为“workers”。当只有一个进程时，进程 1 被认为是一个 worker。否则，workers 被认为是进程 1 之外的所有进程。因此，需要添加 2 个或更多进程才能从 pmap 等并行处理方法中获益。如果你只想在主进程中做其他事情，同时在工作进程上运行长时间的计算，那么添加单个进程是有益的。

让我们开始尝试。以 julia -p n 开始，在本地机器上提供 n 个工作进程。通常，n 等于机器上的 CPU 线程（逻辑核心）的数量是有意义的。请注意，-p 参数隐式加载模块 Distributed。

```
$ julia -p 2

julia> r = remotecall(rand, 2, 2, 2)
Future(2, 1, 4, nothing)

julia> s = @spawnat 2 1 .+ fetch(r)
```

```
Future(2, 1, 5, nothing)

julia> fetch(s)
2×2 Array{Float64,2}:
 1.18526  1.50912
 1.16296  1.60607
```

remotecall 的第一个参数是想要调用的函数，第二个参数是执行函数的进程 id，其余的参数会喂给将要被调用的函数。在 Julia 中进行并行编程时，一般不需要显示地指明具体在哪个进程上执行，不过 remotecall 是一个相对底层的接口用来提供细粒度的管理。

可以看到，第一行代码请求进程 2 构建一个随机矩阵，第二行代码对该矩阵执行加一操作。每次执行的结果存在对应的 Future 中，即 r 和 s。这里 @spawnat 宏会在第一个参数所指定的进程中执行后面第二个参数中的表达式。

有时候，你可能会希望立即获取远程计算的结果，比如，在接下来的操作中就需要读取远程调用的结果，这时候你可以使用 remotecall_fetch 函数，其效果相当于 fetch(remotecall(...))，不过更高效些。

```
julia> remotecall_fetch(r-> fetch(r)[1, 1], 2, r)
0.18526337335308085
```

This fetches the array on worker 2 and returns the first value. Note, that fetch doesn't move any data in this case, since it's executed on the worker that owns the array. One can also write:

```
julia> remotecall_fetch(getindex, 2, r, 1, 1)
0.10824216411304866
```

Remember that getindex(r,1,1) is equivalent to r[1,1], so this call fetches the first element of the future r.

```
julia> remotecall_fetch(getindex, 2, r, 1, 1)
0.10824216411304866
```

回忆下，这里 getindex(r,1,1) 相当于 r[1,1]，因此，上面的调用相当于获取 r 的第一个元素。

为方便起见，可以将符号 :any 传递给 @spawnat，它会为你选择执行操作的位置：

```
julia> r = @spawnat :any rand(2,2)
Future(2, 1, 4, nothing)

julia> s = @spawnat :any 1 .+ fetch(r)
Future(3, 1, 5, nothing)

julia> fetch(s)
2×2 Array{Float64,2}:
 1.38854  1.9098
 1.20939  1.57158
```

请注意，我们使用了 1 .+ fetch(r) 而不是 1 .+ r。这是因为我们不知道代码将在哪里运行，因此通常可能需要一个 fetch 将 r 移动到执行添加的进程。在这种情况下，@spawnat 足够聪明，可以在拥有 r 的进程上执行计算，因此 fetch 将是一个空操作（没有工作被完成）。

(值得注意的是，@spawnat 不是内置的，而是在 Julia 中定义的 [宏](@ref man-macros)。你也可以自己定义此类构造。)

需要记住的重要一点是，一旦 fetch，Future 将在本地缓存其值。进一步的 fetch 调用不需要网络跃点。一旦所有引用 Future 都已获取，远程存储的值将被删除。

@async 类似于 @spawnat，但只在本地进程上运行任务。我们使用它为每个进程创建一个"feeder"任务。每个任务选择需要计算的下一个索引，然后等待其进程完成，然后重复直到我们用完索引。请注意，feeder 任务直到主任务到达 @sync 块的末尾才开始执行，此时它放弃控制并等待所有本地任务完成，然后从主任务返回功能。对于 v0.7 及更高版本，feeder 任务能够通过 nextidx 共享状态，因为它们都运行在同一个进程上。即使 Tasks 是协作调度的，在某些上下文中可能仍然需要锁定，例如在 asynchronous I/O 中。这意味着上下文切换只发生在明确定义的点：在这种情况下，当 remotecall_fetch 被调用时。这是当前的实现状态，它可能会在未来的 Julia 版本中发生变化，因为它旨在使在 M 个 Process 上运行最多 N 个 Tasks 成为可能，也就是 M:N Threading。然后，需要为 nextidx 提供锁获取/释放模型，因为让多个进程同时读写一个资源是不安全的。

25.1 访问代码以及加载库

对于想要并行执行的代码，需要所有对所有进程都可见。例如，在 Julia 命令行中输入以下命令：

```
julia> function rand2(dims...)
           return 2*rand(dims...)
       end

julia> rand2(2,2)
2×2 Array{Float64,2}:
 0.153756  0.368514
 1.15119   0.918912

julia> fetch(@spawnat :any rand2(2,2))
ERROR: RemoteException(2, CapturedException(UndefVarError(Symbol("#rand2"))
Stacktrace:
[...]
```

进程 1 知道函数 rand2 的存在，但进程 2 并不知道。

大多数情况下，你会从文件或者库中加载代码，在此过程中你可以灵活地控制哪个进程加载哪部分代码。假设有这样一个文件，DummyModule.jl，其代码如下：

```
module DummyModule

export MyType, f

mutable struct MyType
    a::Int
end

f(x) = x^2+1

println("loaded")

end
```

为了在所有进程中引用 MyType，DummyModule.jl 需要在每个进程中载入。单独执行 include("DummyModule.jl") 只会在一个进程中将其载入。为了让每个进程都载入它，可以用 @everywhere 宏来实现（启动 Julia 的时候，执行 julia -p 2）。

```
julia> @everywhere include("DummyModule.jl")
loaded
      From worker 3:    loaded
      From worker 2:    loaded
```

像往常一样，这不会将 DummyModule 引入任何进程的作用域，这需要 using 或 import。此外，当 DummyModule 被带入一个进程的作用域时，它不在任何其他进程中：

```
julia> using .DummyModule

julia> MyType(7)
MyType(7)

julia> fetch(@spawnat 2 MyType(7))
ERROR: On worker 2:
UndefVarError: `MyType` not defined


julia> fetch(@spawnat 2 DummyModule.MyType(7))
MyType(7)
```

不过，我们仍然可以在已经包含 (include) 过 DummyModule 的进程中，发送 MyType 类型的实例，尽管此时该进程的命名空间中并没有 MyType 变量:

```
julia> put!(RemoteChannel(2), MyType(7))
RemoteChannel{Channel{Any}}(2, 1, 13)
```

文件代码还可以在启动的时候，通过 -L 参数指定，从而提前在多个进程中载入，然后通过一个 driver.jl 文件控制执行逻辑:

```
julia -p <n> -L file1.jl -L file2.jl driver.jl
```

上面执行 driver.jl 的进程 id 为 1，就跟提供交互式命令行的 Julia 进程一样。

Finally, if DummyModule.jl is not a standalone file but a package, then using DummyModule will *load* DummyModule.jl on all processes, but only bring it into scope on the process where using was called.

最后，如果 DummyModule.jl 不是一个独立的文件，而是一个包，那么 using DummyModule 将在所有进程上加载 DummyModule.jl，但只在调用 [using]（@ref）的进程上将其纳入作用域。

25.2 启动和管理 worker 进程

Julia 自带两种集群管理模式：

Note

While Julia generally strives for backward compatibility, distribution of code to worker processes relies on `Serialization.serialize`. As pointed out in the corresponding documentation, this can not be guaranteed to work across different Julia versions, so it is advised that all workers on all machines use the same version.

Functions `addprocs`, `rmprocs`, `workers`, and others are available as a programmatic means of adding, removing and querying the processes in a cluster.

```
julia> using Distributed

julia> addprocs(2)
2-element Array{Int64,1}:
 2
 3
```

模块 `Distributed` 必须在调用 `addprocs` 之前显式加载到主进程上。它在工作进程上自动可用。

请注意，worker 不会运行 `~/.julia/config/startup.jl` 启动脚本，也不会将其全局状态（例如全局变量、新方法定义和加载的模块）与任何其他正在运行的进程同步。你可以使用 `addprocs(exeflags="--project")` 来初始化具有特定环境的 worker，然后使用 `@everywhere using <modulename>` 或 `@everywhere include("file.jl")`。

其它类型的集群可以通过自己写一个 `ClusterManager` 来实现，下面 集群管理器 部分会介绍。

25.3 数据转移

分布式程序的性能瓶颈主要是由发送消息和数据转移造成的，减少发送消息和转移数据的数量对于获取高性能和可扩展性至关重要，因此，深入了解 Julia 分布式程序是如何转移数据的非常有必要。

`fetch` 可以被认为是一个显式的数据转移操作，因为它直接要求将一个对象移动到本地机器。`@spawnat`（以及一些相关的结构体）也移动数据，但这并不明显，因此可以称为隐式数据转移操作。考虑这两种构造和平方一个随机矩阵的方法：

方法一：

```
julia> A = rand(1000,1000);

julia> Bref = @spawnat :any A^2;

[...]

julia> fetch(Bref);
```

方法二：

```
julia> Bref = @spawnat :any rand(1000,1000)^2;

[...]

julia> fetch(Bref);
```

这种差异看起来微不足道，但实际上由于 @spawnat 的行为而非常显着。在第一种方法中，在本地构造一个随机矩阵，然后将其发送到另一个进程进行平方。在第二种方法中，随机矩阵在另一个进程中被构造和平方。因此，第二种方法发送的数据比第一种方法少得多。

在这个简单示例中，这两种方法很容易区分和选择。然而，在一个真正的程序设计数据转移可能需要更多的思考和一些测量。例如，如果第一个进程需要矩阵 A，那么第一种方法可能更好。或者，如果计算 A 很昂贵并且只有当前进程拥有它，那么将它移到另一个进程可能是不可避免的。或者，如果当前进程在 @spawnat 和 fetch(Bref) 之间几乎没有什么关系，最好完全消除并行性。或者想象一下 rand(1000,1000) 被更昂贵的操作取代。那么为这一步添加另一个 @spawnat 语句可能是有意义的。

25.4 全局变量

通过 @spawnat 远程执行的表达式，或使用 remotecall 为远程执行指定的闭包可能会引用全局变量。与其他模块中的全局绑定相比，模块 Main 下的全局绑定的处理方式略有不同。考虑以下代码片段：

```
A = rand(10,10)
remotecall_fetch(()->sum(A), 2)
```

在这种情况下，sum 必须在远程进程中定义。请注意，A 是在本地工作区中定义的全局变量。worker 2 在 Main 下没有名为 A 的变量。将闭包 ()->sum(A) 传送到 worker 2 的行为导致 Main.A 被定义在 2 上。即使在调用 remotecall_fetch 返回之后，Main.A 仍然存在于 worker 2 上。带有嵌入式全局引用的远程调用（仅在 Main 模块下）以如下的方式管理全局变量：

- 在全局调用中引用的全局绑定会在将要执行该调用的 worker 中被创建。
- 全局常量仍然在远端结点定义为常量。
- 全局绑定会在下一次远程调用中引用到的时候，当其值发生改变时，再次发送给目标 worker。此外，集群并不会所有结点的全局绑定。例如：

  ```
  A = rand(10,10)
  remotecall_fetch(()->sum(A), 2) # worker 2
  A = rand(10,10)
  remotecall_fetch(()->sum(A), 3) # worker 3
  A = nothing
  ```

  可以看到，A 作为全局变量在 worker 2 中有定义，而 B 是一个局部变量，因而最后在 worker 2 中并没有 B 的绑定。执行以上代码之后，worker 2 和 worker 3 中的 Main.A 的值是不同的，同时，节点 1 上的值则为 nothing。

也许你也注意到了，在 master 主节点上被赋值为 nothing 之后，全局变量的内存会被回收，但在 worker 节点上的全局变量并没有被回收掉。执行 clear! 可以手动将远端结点上的特定全局变量置为 nothing，然后对应的内存会被周期性的垃圾回收机制回收。

因此，在远程调用中，需要非常小心地引用全局变量。事实上，应当尽量避免引用全局变量，如果必须引用，那么可以考虑用 let 代码块将全局变量局部化：

```
julia> A = rand(10,10);

julia> remotecall_fetch(()->A, 2);

julia> B = rand(10,10);
```

```
julia> let B = B
           remotecall_fetch(()->B, 2)
       end;

julia> @fetchfrom 2 InteractiveUtils.varinfo()
name           size summary
––––––––––––––––––––––––––––––––––––––––––

A         800 bytes 10×10 Array{Float64,2}
Base                Module
Core                Module
Main                Module
```

可以看到，A 作为全局变量在 worker 2 中有定义，而 B 是一个局部变量，因而最后在 worker 2 中并没有 B 的绑定。

25.5 并行的 Map 和 Loop

幸运的是，许多有用的并行计算不需要数据转移。一个常见的例子是蒙特卡罗模拟，其中多个进程可以同时处理独立的模拟试验。我们可以使用 @spawnat 在两个进程上抛硬币。首先，在 count_heads.jl 中编写以下函数：

```
function count_heads(n)
    c::Int = 0
    for i = 1:n
        c += rand(Bool)
    end
    c
end
```

函数 count_heads 只是简单地将 n 个随机 0-1 值累加，下面在两个机器上进行试验，并将结果叠加：

```
julia> @everywhere include_string(Main, $(read("count_heads.jl", String)), "count_heads.jl")

julia> a = @spawnat :any count_heads(100000000)
Future(2, 1, 6, nothing)

julia> b = @spawnat :any count_heads(100000000)
Future(3, 1, 7, nothing)

julia> fetch(a)+fetch(b)
100001564
```

上面的例子展示了一种非常常见而且有用的并行编程模式，在一些进程中执行多次独立的迭代，然后将它们的结果通过某个函数合并到一起，这个合并操作通常称作**聚合** (*reduction*)，也就是一般意义上的**张量降维** (tensor-rank-reducing)，比如将一个向量降维成一个数，或者是将一个 tensor 降维到某一行或者某一列等。在代码中，通常具有 x = f(x, v[i]) 这种形式，其中 x 是一个叠加器，f 是一个聚合函数，而 v[i] 则是将要被聚合的值。一般来说，f 要求满足结合律，这样不管执行的顺序如何，都不会影响计算结果。

请注意，我们可以将这种 count_heads 模式推广。我们使用了两个显式的 @spawnat 语句，将并行性限制为两个进程。要在任意数量的进程上运行，我们可以使用 并行 *for* 循环，在分布式内存中运行，可以在 Julia 中使用 @distributed 编写，如下所示：

```
nheads = @distributed (+) for i = 1:200000000
    Int(rand(Bool))
end
```

上面的写法将多次迭代分配到了不同的进程，然后通过一个聚合函数（这里是 (+)）合并计算结果，其中，每次迭代的结果作为 for 循环中的表达式的结果，最后整个循环的结果聚合后得到最终的结果。

注意，尽管这里 for 循环看起来跟串行的 for 循环差不多，实际表现完全不同。这里的迭代并没有特定的执行顺序，而且由于所有的迭代都在不同的进程中进行，其中变量的写入对全局来说不可见。所有并行的 for 循环中的变量都会复制并广播到每个进程。

比如，下面这段代码并不会像你想要的那样执行：

```
a = zeros(100000)
@distributed for i = 1:100000
    a[i] = i
end
```

这段代码并不会把 a 的所有元素初始化，因为每个进程都会有一份 a 的拷贝，因此类似的 for 循环一定要避免。幸运的是，共享数组 可以用来突破这种限制：

```
using SharedArrays

a = SharedArray{Float64}(10)
@distributed for i = 1:10
    a[i] = i
end
```

当然，对于 for 循环外面的变量来说，如果是只读的话，使用起来完全没问题：

```
a = randn(1000)
@distributed (+) for i = 1:100000
    f(a[rand(1:end)])
end
```

这里每次迭代都会从共享给每个进程的向量 a 中随机选一个样本，然后用来计算 f。

如你所见，如果不需要，可以省略归约运算符。在这种情况下，循环异步执行，即它在所有可用的 worker 上产生独立的任务，并立即返回一个 Future 数组，而无需等待完成。调用者可以稍后通过调用 fetch 来等待 Future 完成，或者通过添加前缀 @sync，比如 @sync @distributed for，来等待循环结束。

在一些不需要聚合函数的情况下，我们可能只是像对某个范围内的整数应用一个函数 (或者，更一般地，某个序列中的所有元素)，这种操作称作**并行的 map**，在 Julia 中有一个对应的函数 pmap。例如，可以像下面这样计算一些随机大矩阵的奇异值：

```
julia> M = Matrix{Float64}[rand(1000,1000) for i = 1:10];

julia> pmap(svdvals, M);
```

Julia 中的 pmap 是被设计用来处理一些计算量比较复杂的函数的并行化的。与之对比的是，@distributed for 是用来处理一些每次迭代计算都很轻量的计算，比如简单地对两个数求和。pmap 和 @distributed for 都只会用到 worker 的进程。对于 @distributed for 而言，最后的聚合计算由发起者的进程完成。

25.6 远程引用和 AbstractChannel

远程引用通常指某种 AbstractChannel 的实现。

AbstractChannel（如 Channel）的具体实现，需要实现 put!，take!，fetch，isready 和 wait。Future 所引用的远程对象存储在 Channel{Any}(1) 中，即大小为 1 的、能够容纳 Any 类型对象的 Channel 。

RemoteChannel 可以被反复写入，可以指向任意大小和类型的 channel（或者是任意 AbstractChannel 的实现）。

RemoteChannel(f::Function, pid)() 构造器可以构造一些引用，而这些引用指向的 channel 可以容纳多个某种具体类型的数据。其中 f 是将要在 pid 上执行的函数，其返回值必须是 AbstractChannel 类型。

例如，RemoteChannel(()->Channel{Int}(10), pid) 会创建一个 channel，其类型是 Int，容量是 10，这个 channel 存在于 pid 进程中。

针对 RemoteChannel 的 put!，take!，fetch，isready 和 wait 方法会被重定向到其底层存储着 channel 的进程。

因此，RemoteChannel 可以用来引用用户自定义的 AbstractChannel 对象。在 Examples repository 中的 dictchannel.jl 文件中有一个简单的例子，其中使用了一个字典用于远端存储。

25.7 Channel 和 RemoteChannel

- 一个 Channel 仅对局部的进程可见，worker 2 无法直接访问 worker 3 上的 Channel，反之亦如此。不过 RemoteChannel 可以跨 worker 获取和写入数据。
- RemoteChannel 可以看作是对 Channel 的封装。
- RemoteChannel 的 pid 就是其封装的 channel 所在的进程 id。
- 任意拥有 RemoteChannel 引用的进程都可以对其进行读写，数据会自动发送到 RemoteChannel 底层 channel 的进程（或从中获取数据）
- 序列化 Channel 会将其中的所有数据也都序列化，因此反序列化的时候也就可以得到一个原始数据的拷贝。
- 不过，对 RemoteChannel 的序列化则只会序列化其底层指向的 channel 的 id，因此反序列化之后得到的对象仍然会指向之前存储的对象。

如上的通道示例可以修改为进程间通信，如下所示

首先，起 4 个 worker 进程处理同一个 remote channel jobs，其中的每个 job 都有一个对应的 job_id，然后每个 task 读取一个 job_id，然后模拟随机等待一段时间，然后往存储结果的 RemoteChannel 中写入一个 Tuple 对象，其中包含 job_id 和等待的时间。最后将结果打印出来。

```
julia> addprocs(4); # add worker processes

julia> const jobs = RemoteChannel(()->Channel{Int}(32));

julia> const results = RemoteChannel(()->Channel{Tuple}(32));

julia> @everywhere function do_work(jobs, results) # define work function everywhere
           while true
               job_id = take!(jobs)
               exec_time = rand()
               sleep(exec_time) # simulates elapsed time doing actual work
               put!(results, (job_id, exec_time, myid()))
           end
       end

julia> function make_jobs(n)
           for i in 1:n
               put!(jobs, i)
           end
       end;

julia> n = 12;

julia> errormonitor(@async make_jobs(n)); # feed the jobs channel with "n" jobs

julia> for p in workers() # start tasks on the workers to process requests in parallel
           remote_do(do_work, p, jobs, results)
       end

julia> @elapsed while n > 0 # print out results
           job_id, exec_time, where = take!(results)
           println("$job_id finished in $(round(exec_time; digits=2)) seconds on worker $where")
           global n = n - 1
       end
1 finished in 0.18 seconds on worker 4
2 finished in 0.26 seconds on worker 5
6 finished in 0.12 seconds on worker 4
7 finished in 0.18 seconds on worker 4
5 finished in 0.35 seconds on worker 5
4 finished in 0.68 seconds on worker 2
3 finished in 0.73 seconds on worker 3
11 finished in 0.01 seconds on worker 3
12 finished in 0.02 seconds on worker 3
9 finished in 0.26 seconds on worker 5
8 finished in 0.57 seconds on worker 4
10 finished in 0.58 seconds on worker 2
0.055971741
```

远程调用和分布式垃圾回收

远程引用所指向的对象可以在其所有引用都被集群删除之后被释放掉。

存储值的节点会跟踪哪些 worker 引用了它。每次将 `RemoteChannel` 或（未获取的）`Future` 序列化为 worker 时，都会通知引用指向的节点。并且每次在本地对 `RemoteChannel` 或（未获取的）`Future` 进行

垃圾回收时，都会再次通知拥有该值的节点。这是在内部集群感知序列化程序中实现的。远程引用仅在正在运行的集群的上下文中有效。不支持对常规 IO 对象的引用进行序列化和反序列化。

上面说到的**通知**都是通过发送”跟踪”信息来实现的，当一个引用被序列化的时候，就会发送”添加引用”的信息，而一个引用被本地的垃圾回收器回收的时候，就会发送一个”删除引用”的信息。

由于 Future 是一次性写入并在本地缓存，因此 fetching 一个 Future 的行为也会更新拥有该值的节点上的引用跟踪信息。

一旦指向某个值的引用都被删除了，对应的节点会将其释放。

使用 [Future]（@ref Distributed.Future），将已获取的 [Future]（@ref Distributed.Future）序列化到其他节点也会发送该值，因为此时原始远程存储可能已收集该值了。

此外需要注意的是，本地的垃圾回收到底发生在什么时候取决于具体对象的大小以及当时系统的内存压力。

在远程引用的情况下，本地引用对象的大小非常小，而存储在远程节点上的值可能非常大。由于可能不会立即收集本地对象，因此在 RemoteChannel 的本地实例或未获取的 Future 上显式调用 finalize 是一个好习惯。由于在 Future 上调用 fetch 也会从远程存储中删除其引用，因此在获取的 Future 上不需要这样做。显式调用 finalize 会导致立即向远程节点发送消息以继续并删除其对该值的引用。

一旦执行了 finalize 之后，引用就不可用了。

25.8 本地调用

数据必须复制到远程节点以供执行。远程调用和数据存储到不同节点上的 RemoteChannel / Future 时都是这种情况。正如预期的那样，这会在远程节点上生成序列化对象的副本。但是，当目的节点是本地节点时，即调用进程 id 与远程节点 id 相同，则作为本地调用执行。它通常（并非总是）在不同的 Task 中执行 - 但没有数据的序列化/反序列化。因此，该调用引用了与传递相同的对象实例 - 没有创建副本。这种行为在下面突出显示：

```
julia> using Distributed;

julia> rc = RemoteChannel(()->Channel(3));   # RemoteChannel created on local node

julia> v = [0];

julia> for i in 1:3
           v[1] = i                          # Reusing `v`
           put!(rc, v)
       end;

julia> result = [take!(rc) for _ in 1:3];

julia> println(result);
Array{Int64,1}[[3], [3], [3]]

julia> println("Num Unique objects : ", length(unique(map(objectid, result))));
Num Unique objects : 1

julia> addprocs(1);

julia> rc = RemoteChannel(()->Channel(3), workers()[1]);   # RemoteChannel created on remote node

julia> v = [0];
```

```
julia> for i in 1:3
           v[1] = i
           put!(rc, v)
       end;

julia> result = [take!(rc) for _ in 1:3];

julia> println(result);
Array{Int64,1}[[1], [2], [3]]

julia> println("Num Unique objects : ", length(unique(map(objectid, result))));
Num Unique objects : 3
```

可以看出，本地拥有的 RemoteChannel 上的 put! 在调用之间修改了相同的对象 v 会导致存储相同的单个对象实例。与当拥有 rc 的节点是不同节点时创建的 v 副本相反。

需要注意的是，这通常不是问题。只有当对象既存储在本地又在调用后被修改时，才需要考虑这一点。在这种情况下，存储对象的 deepcopy 可能是合适的。

对于本地节点上的远程调用也是如此，如下例所示：

```
julia> using Distributed; addprocs(1);

julia> v = [0];

julia> v2 = remotecall_fetch(x->(x[1] = 1; x), myid(), v);     # Executed on local node

julia> println("v=$v, v2=$v2, ", v === v2);
v=[1], v2=[1], true

julia> v = [0];

julia> v2 = remotecall_fetch(x->(x[1] = 1; x), workers()[1], v); # Executed on remote node

julia> println("v=$v, v2=$v2, ", v === v2);
v=[0], v2=[1], false
```

再次可以看出，对本地节点的远程调用就像直接调用一样。调用修改作为参数传递的本地对象。在远程调用中，它对参数的副本进行操作。

重复一遍，一般来说这不是问题。如果本地节点也被用作计算节点，并且在调用后使用的参数，则需要考虑此行为，并且如果需要，必须将参数的深拷贝传递给在本地节点上唤起的调用。对远程节点的调用将始终对参数的副本进行操作。

25.9 共享数组

共享数组使用系统共享内存将数组映射到多个进程上，尽管和 DArray 有点像，但其实际表现有很大不同。在 DArray 中，每个进程可以访问数据中的一块，但任意两个进程都不能共享同一块数据，而对于 SharedArray，每个进程都可以访问整个数组。如果你想在一台机器上，让一大块数据能够被多个进程访问到，那么 SharedArray 是个不错的选择。

共享数组由 SharedArray 提供，必须在所有相关的 worker 中都显式地加载。

对 SharedArray 索引（访问和复制）操作就跟普通的数组一样，由于底层的内存对本地的进程是可见的，索引的效率很高，因此大多数单进程上的算法对 SharedArray 来说都是适用的，除非某些

算法必须使用 Array 类型（此时可以通过调用 sdata 来获取 SharedArray 数组）。对于其它类型的 AbstractArray 类型数组来说，sdata 仅仅会返回数组本身，因此，可以放心地使用 sdata 对任意类型的 Array 进行操作。

共享数组可以通过以下形式构造：

```
SharedArray{T,N}(dims::NTuple; init=false, pids=Int[])
```

它在由 pids 指定的进程中创建了一个位类型为 T 和形状为 dims 的 N 维共享数组。与分布式数组不同，共享数组只能从由 pids 命名参数指定的那些参与 worker 访问（如果创建过程在同一主机上，也是如此）。请注意，SharedArray 中仅支持 isbits 元素。

如果提供了一个类型为 initfn(S::SharedArray) 的 init 函数，那么所有相关的 worker 都会调用它。你可以让每个 worker 都在共享数组不同的地方执行 init 函数，从而实现并行初始化。

下面是个例子：

```
julia> using Distributed

julia> addprocs(3)
3-element Array{Int64,1}:
 2
 3
 4

julia> @everywhere using SharedArrays

julia> S = SharedArray{Int,2}((3,4), init = S -> S[localindices(S)] = repeat([myid()],
↪   length(localindices(S))))
3×4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  3  4  4

julia> S[3,2] = 7
7

julia> S
3×4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  7  4  4
```

SharedArrays.localindices 提供了一个以为的切片，可以很方便地用来将 task 分配到各个进程上。当然你可以按你想要的方式做区分：

```
julia> S = SharedArray{Int,2}((3,4), init = S -> S[indexpids(S):length(procs(S)):length(S)] =
↪   repeat([myid()], length( indexpids(S):length(procs(S)):length(S))))
3×4 SharedArray{Int64,2}:
 2  2  2  2
 3  3  3  3
 4  4  4  4
```

由于所有的进程都能够访问底层的数据，因此一定要小心避免出现冲突：

```
@sync begin
    for p in procs(S)
        @async begin
            remotecall_wait(fill!, p, S, p)
        end
    end
end
```

上面的代码会导致不确定的结果，因为每个进程都将**整个**数组赋值为其 `pid`，从而导致最后一个执行完成的进程会保留其 `pid`。

考虑更复杂的一种情况：

```
q[i,j,t+1] = q[i,j,t] + u[i,j,t]
```

这个例子中，如果首先将任务用按照一维的索引作区分，那么就会出问题：如果 `q[i,j,t]` 位于分配给某个 worker 的最后一个位置，而 `q[i,j,t+1]` 位于下一个 worker 的开始位置，那么后面这个 worker 开始计算的时候，可能 `q[i,j,t]` 还没有准备好，这时候，更好的做法是，手动分区，比如可以定义一个函数，按照 `(irange,jrange)` 给每个 worker 分配任务。

```
julia> @everywhere function myrange(q::SharedArray)
           idx = indexpids(q)
           if idx == 0 # This worker is not assigned a piece
               return 1:0, 1:0
           end
           nchunks = length(procs(q))
           splits = [round(Int, s) for s in range(0, stop=size(q,2), length=nchunks+1)]
           1:size(q,1), splits[idx]+1:splits[idx+1]
       end
```

然后定义计算内核：

```
julia> @everywhere function advection_chunk!(q, u, irange, jrange, trange)
           @show (irange, jrange, trange)  # display so we can see what's happening
           for t in trange, j in jrange, i in irange
               q[i,j,t+1] = q[i,j,t] + u[i,j,t]
           end
           q
       end
```

然后定义一个 wrapper：

```
julia> @everywhere advection_shared_chunk!(q, u) =
           advection_chunk!(q, u, myrange(q)..., 1:size(q,3)-1)
```

接下来，比较三个不同的版本，第一个是单进程版本：

```
julia> advection_serial!(q, u) = advection_chunk!(q, u, 1:size(q,1), 1:size(q,2), 1:size(q,3)-1);
```

然后是使用 @distributed:

```
julia> function advection_parallel!(q, u)
           for t = 1:size(q,3)-1
               @sync @distributed for j = 1:size(q,2)
                   for i = 1:size(q,1)
                       q[i,j,t+1]= q[i,j,t] + u[i,j,t]
                   end
               end
           end
           q
       end;
```

最后是使用分区:

```
julia> function advection_shared!(q, u)
           @sync begin
               for p in procs(q)
                   @async remotecall_wait(advection_shared_chunk!, p, q, u)
               end
           end
           q
       end;
```

如果创建好了 SharedArray 之后，计算这些函数的执行时间，那么可以得到以下结果（用 julia -p 4 启动）:

```
julia> q = SharedArray{Float64,3}((500,500,500));

julia> u = SharedArray{Float64,3}((500,500,500));
```

先执行一次以便 JIT 编译，然后用 @time 宏测试其第二次执行的时间:

```
julia> @time advection_serial!(q, u);
(irange,jrange,trange) = (1:500,1:500,1:499)
 830.220 milliseconds (216 allocations: 13820 bytes)

julia> @time advection_parallel!(q, u);
   2.495 seconds      (3999 k allocations: 289 MB, 2.09% gc time)

julia> @time advection_shared!(q,u);
        From worker 2:       (irange,jrange,trange) = (1:500,1:125,1:499)
        From worker 4:       (irange,jrange,trange) = (1:500,251:375,1:499)
        From worker 3:       (irange,jrange,trange) = (1:500,126:250,1:499)
        From worker 5:       (irange,jrange,trange) = (1:500,376:500,1:499)
 238.119 milliseconds (2264 allocations: 169 KB)
```

这里 advection_shared! 最大的优势在于，最小程度地降低了 woker 之间的通信，从而让每个 worker 能针对被分配的部分持续地计算一段时间。

共享数组与分布式垃圾回收

和远程引用一样，共享数组也依赖于创建节点上的垃圾回收来释放所有参与的 worker 上的引用。因此，创建大量生命周期比较短的数组，并尽可能快地显式 finilize 这些对象，代码会更高效，这样与之对用的内存和文件句柄都会更快地释放。

25.10 集群管理器

Julia 通过集群管理器实现对多个进程（所构成的逻辑上的集群）的启动，管理以及网络通信。一个 `ClusterManager` 负责：

- 在一个集群环境中启动 worker 进程
- 管理每个 worker 生命周期内的事件
- （可选），提供数据传输

一个 Julia 集群由以下特点：

- 初始进程，称为 `master`，其 `id` 为 `1`
- 只有 master 进程可以增加或删除 worker 进程
- 所有进程之间都可以直接通信

worker 之间的连接（用的是内置的 TCP/IP 传输）按照以下方式进行：

- master 进程对一个 `ClusterManager` 对象调用 `addprocs`
- `addprocs` 调用对应的 `launch` 方法，然后在对应的机器上启动相应数量的 worker 进程
- 每个 worker 监听一个端口，然后将其 host 和 port 信息传给 `stdout`
- 集群管理器捕获 `stdout` 中每个 worker 的信息，并提供给 master 进程
- master 进程解析信息并与相应的 worker 建立 TCP/IP 连接
- 每个 worker 都会被通知集群中的其它 worker
- 每个 worker 与 `id` 小于自己的 worker 连接
- 这样，一个网络就建立了，从而，每个 worker 都可以与其它 worker 建立连接

尽管默认的传输层使用的是 `TCPSocket`，对于一个自定义的集群管理器来说，完全可以使用其它传输方式。

Julia 提供了两种内置的集群管理器：

- `LocalManager`，调用 `addprocs()` 或 `addprocs(np::Integer)` 时会用到。
- `SSHManager`，调用 `addprocs(hostnames::Array)` 时，传递一个 hostnames 的列表。

`LocalManager` 用来在同一个 host 上启动多个 worker，从而利用多核/多处理器硬件。

因此，一个最小的集群管理器需要：

- 是一个 ClusterManager 抽象类的一个子类
- 实现 launch 接口，用来启动新的 worker
- 实现 manage，在一个 worker 的生命周期中多次被调用（例如，发送中断信号）

addprocs(manager::FooManager) 需要 FooManager 实现：

```
function launch(manager::FooManager, params::Dict, launched::Array, c::Condition)
    [...]
end

function manage(manager::FooManager, id::Integer, config::WorkerConfig, op::Symbol)
    [...]
end
```

作为一个例子，我们来看下 LocalManager 是怎么实现的：

```
struct LocalManager <: ClusterManager
    np::Integer
end

function launch(manager::LocalManager, params::Dict, launched::Array, c::Condition)
    [...]
end

function manage(manager::LocalManager, id::Integer, config::WorkerConfig, op::Symbol)
    [...]
end
```

launch 方法接收以下参数：

- manager::ClusterManager：调用 addprocs 时所用到的集群管理器
- params::Dict：所有的关键字参数都会传递到 addprocs 中
- launched::Array：用来存储一个或多个 WorkerConfig
- c::Condition：在 workers 启动后被通知的条件变量

launch 会在一个异步的 task 中调用，该 task 结束之后，意味着所有请求的 worker 都已经启动好了。因此，launch 函数**必须**在所有 worker 启动之后，尽快退出。

新启动的 worker 之间采用的是多对多的连接方式。在命令行中指定参数 --worker[=<cookie>] 会让所有启动的进程把自己当作 worker，然后通过 TCP/IP 构建连接。

集群中所有的 worker 默认使用同一个 master 的 cookie。如果 cookie 没有指定，（比如没有通过 --worker 指定），那么 worker 会尝试从它的标准输入中读取。LocalManager 和 SSHManager 都是通过标准输入来将 cookie 传递给新启动的 worker。

默认情况下，一个 worker 会监听从 getipaddr() 函数返回的地址上的一个开放端口。若要指定监听的地址，可以通过额外的参数 --bind-to bind_addr[:port] 指定，这对于多 host 的情况来说很方便。

对于非 TCP/IP 传输，可以选择 MPI 作为一种实现，此时一定**不要**指定 --worker 参数，另外，新启动的 worker 必须调用 init_worker(cookie) 之后再使用并行的结构体。

对于每个已经启动的 worker，launch 方法必须往 launched 中添加一个 WorkerConfig 对象（相应的值已经初始化）。

```
mutable struct WorkerConfig
    # Common fields relevant to all cluster managers
    io::Union{IO, Nothing}
    host::Union{AbstractString, Nothing}
    port::Union{Integer, Nothing}

    # Used when launching additional workers at a host
    count::Union{Int, Symbol, Nothing}
    exename::Union{AbstractString, Cmd, Nothing}
    exeflags::Union{Cmd, Nothing}

    # External cluster managers can use this to store information at a per-worker level
    # Can be a dict if multiple fields need to be stored.
    userdata::Any

    # SSHManager / SSH tunnel connections to workers
    tunnel::Union{Bool, Nothing}
    bind_addr::Union{AbstractString, Nothing}
    sshflags::Union{Cmd, Nothing}
    max_parallel::Union{Integer, Nothing}

    # Used by Local/SSH managers
    connect_at::Any

    [...]
end
```

WorkerConfig 中的大多数字段都是内置的集群管理器会用到，对于自定义的管理器，通常只需要指定 io 或 host/port:

- 如果指定了 io，那么就会用来读取 host/port 信息。每个 worker 会在启动时打印地址和端口，这样 worker 就可以自由监听可用的端口，而不必手动配置 worker 的端口。
- 如果 io 没有指定，那么 host 和 port 就会用来连接。
- count，exename 和 exeflags 用于从一个 worker 上启动额外的 worker。例如，一个集群管理器可能对每个节点都只启动一个 worker，然后再用它来启动额外的 worker。
  - count 可以是一个整数 n，用来指定启动 n 个 worker
  - count 还可以是 :auto，用来启动跟那台机器上 CPU 个数（逻辑上的核的个数）相同的 worker
  - exename 是 julia 可执行文件的全路径
  - exeflags 应该设置成传递给将要启动的 worker 命令行参数
- tunnel, bind_addr, sshflags 和 max_parallel 会在从 worker 与 master 进程建立 ssh 隧道时用到
- userdata 用来提供给自定义集群管理器存储自己的 worker 相关的信息

manage(manager::FooManager, id::Integer, config::WorkerConfig, op::Symbol) 会在一个 worker 生命周期中的不同时刻被调用，其中 op 的值可能是：

- `:register/:deregister`，从 Julia 的 worker 池子中添加/删除一个 worker
- `:interrupt`，当 `interrupt(workers)` 被调用是，此时，`ClusterManager` 应该给相应的 worker 发送终端信号
- `:finalize`，用于清理操作。

自定义集群管理器的传输方式

将默认的 TCP/IP 多对多 socket 连接替换成一个自定义的传输层需要做很多工作。每个 Julia 进程都有与其连接的 worker 数量相同的通信 task。例如，在一个有 32 个进程的多对多集群中：

- 每个进程都有 31 个通信 task
- 每个 task 在一个**消息处理循环**中从一个远端 worker 读取所有的输入信息
- 每个消息处理循环等待一个 `IO` 对象（比如，在默认实现中是一个 `TCPSocket`），然后读取整个信息，处理，等待下一个
- 发送消息则可以直接在任意 Julia task 中完成，而不只是通信 task，同样，也是通过相应的 `IO` 对象

要替换默认的传输方式，需要新的实现能够在远程 worker 之间建立连接，同时提供一个可以用来被消息处理循环等待的 `IO` 对象。集群管理器的回调函数需要实现如下函数：

```
connect(manager::FooManager, pid::Integer, config::WorkerConfig)
kill(manager::FooManager, pid::Int, config::WorkerConfig)
```

The default implementation (which uses TCP/IP sockets) is implemented as `connect(manager::ClusterManager, pid::Integer, config::WorkerConfig)`.

`connect` should return a pair of `IO` objects, one for reading data sent from worker `pid`, and the other to write data that needs to be sent to worker `pid`. Custom cluster managers can use an in-memory `BufferStream` as the plumbing to proxy data between the custom, possibly non-`IO` transport and Julia's in-built parallel infrastructure.

`BufferStream` 是一个内存中的 `IOBuffer`，其表现很像 `IO`，就是一个**流**（stream），可以异步地处理。

在 Examples repository 的 `clustermanager/0mq` 目录中，包含一个使用 ZeroMQ 连接 Julia worker 的例子，用的是星型拓补结构。需要注意的是：Julia 的进程仍然是**逻辑上**相互连接的，任意 worker 都可以与其它 worker 直接相连而无需感知到 0MQ 作为传输层的存在。

在使用自定义传输的时候：

- Julia 的 workers 必须**不能**通过 `--worker` 启动。如果启动的时候使用了 `--worker`，那么新启动的 worker 会默认使用基于 TCP/IP socket 的实现
- 对于每个 worker 逻辑上的输入连接，必须调用 `Base.process_messages(rd::IO, wr::IO)()`，这会创建一个新的 task 来处理 worker 消息的读写
- `init_worker(cookie, manager::FooManager)` 必须作为 worker 进程初始化的一部分呢被调用
- `WorkerConfig` 中的 `connect_at::Any` 字段可以被集群管理器在调用 `launch` 的时候设置，该字段的值会发送到所有的 `connect` 回调中。通常，其中包含的是**如何连接到**一个 worker 的信息。例如，在 TCP/IP socket 传输中，用这个字段存储 `(host, port)` 来声明如何连接到一个 worker。

`kill(manager, pid, config)` 用来从一个集群中删除一个 worker，在 master 进程中，对应的 `IO` 对象必须通过对应的实现来关闭，从而保证正确地释放资源。默认的实现简单地对指定的远端 worker 执行 `exit()` 即可。

在例子目录中，`clustermanager/simple` 展示了一个简单地实现，使用的是 UNIX 下的 socket。

LocalManager 和 SSHManager 的网络要求

Julia 集群设计的时候，默认是在一个安全的环境中执行，比如本地的笔记本，部门的集群，甚至是云端。这部分将介绍 `LocalManager` 和 `SSHManager` 的网络安全要点：

- master 进程不监听任何端口，它只负责向外连接 worker
- 每个 worker 都只绑定一个本地的接口，同时监听一个操作系统分配的临时端口。
- `addprocs(N)` 使用的 `LocalManager`，默认只会绑定到回环接口（loopback interface），这就意味着，之后在远程主机上（恶意）启动的 worker 无法连接到集群中，在执行 `addprocs(4)` 之后，又跟一个 `addprocs(["remote_host"])` 会失败。有些用户可能希望创建一个集群同时管理本地系统和几个远端系统，这可以通过在绑定 `LocalManager` 到外部网络接口的时候，指定一个 `restrict` 参数：`addprocs(4; restrict=false)`
- `addprocs(list_of_remote_hosts)` 使用的 `SSHManager` 会通过 SSH 启动远程机上的 worker。

默认 SSH 只会用来启动 Julia 的 worker。随后的 master-worker 和 worker-worker 连接使用的是普通的、未加密的 TCP/IP 通信。远程机必须开启免密登陆。额外的 SSH 标记或认证信息会通过关键字参数 `sshflags` 指定。

- `addprocs(list_of_remote_hosts; tunnel=true, sshflags=<ssh keys and other flags>)` 在我们希望给 master-worker 也使用 SSH 连接的时候很有用。一个典型的场景是本地的笔记本运行 Julia ERPL（做为 master）和云上的其他机器，比如 Amazon EC2，构成集群。这时候远程机器只要开启 22 端口就可以，然后要有 SSH 客户端通过公约基础设施（PKI）认证过。授权信息可以通过 `sshflags` 生效，比如 ``sshflags=`-i <keyfile>` ``。

  在一个所有节点联通的拓扑网中（默认情况下是这样的），所有的 worker 节点都通过普通 TCP socket 通信互相连接。这样集群的安全策略就必须允许 worker 节点间通过操作系统分配的临时端口范围自由连接。

  所有 worker-worker 间（都是 SSH）的安全和加密或者信息的加密都可以通过自定义 `ClusterManager` 完成。

- 如果将 `multiplex=true` 指定为 `addprocs` 的选项，则 SSH 多路复用用于在 master 和 worker 之间创建隧道。如果你自己配置了 SSH 多路复用并且已经建立了连接，则无论 `multiplex` 选项如何，都会使用 SSH 多路复用。如果启用了多路复用，则使用现有连接（ssh 中的 `-O forward` 选项）设置转发。如果你的服务器需要密码验证，那么这就很有用了；

  你可以通过在 `addprocs` 之前登录服务器来避免在 Julia 中进行身份验证。除非使用现有的多路复用连接，否则在会话期间控制套接字将位于 `~/.ssh/julia-%r@%h:%p`。请注意，如果你在一个节点上创建多个进程并启用多路复用，带宽可能会受到限制，因为在这种情况下，进程共享一个多路复用 TCP 连接。

集群 Cookie

集群上所有的进程都共享同一个 cookie，默认是 master 进程随机生成的字符串。

- `cluster_cookie()` 返回 cookie，而 `cluster_cookie(cookie)()` 设置并返回新的 cookie。

- 所有的连接都进行双向认证，从而保证只有 master 启动的 worker 才能相互连接。
- cookie 可以在 worker 启动的时候，通过参数 `--worker=<cookie>` 指定，如果参数 `--worker` 没有指定 cookie，那么 worker 会从它的标准输入中 (`stdin`) 读取，`stdin` 会在 cookie 获取之后立即关闭。
- `ClusterManager` 可以通过 `cluster_cookie()` 从 master 中过去 cookie，不适用默认 TCP/IP 传输的集群管理器（即没有指定 `--worker`）必须用于 master 相同的 cookie 调用 `init_worker(cookie, manager)`。

注意，在对安全性要求很高的环境中，可以通过自定义 `ClusterManager` 实现。例如，cookie 可以提前共享，然后不必再启动参数中指定。

25.11 指定网络拓补结构（实验性功能）

传递给 `addprocs` 的关键字参数 `topology` 用于指定 workers 必须如何相互连接：

- `:all_to_all`，默认的，所有 worker 之间相互都连接
- `:master_worker`，只有主进程，即 `pid` 为 1 的进程能够与 worker 建立连接
- `:custom`: 集群管理器的 `launch` 方法通过 `WorkerConfig` 中的 `ident` 和 `connect_idents` 指定连接的拓补结构。一个 worker 通过集群管理器提供的 `ident` 来连接到所有 `connect_idents` 指定的 worker。

关键字参数 `lazy=true|false` 只会影响 `topology` 选项中的 `:all_to_all`。如果是 `true`，那么集群启动的时候 master 会连接所有的 worker，然后 worker 之间的特定连接会在初次唤醒的是建立连接，这有利于降低集群初始化的时候对资源的分配。`lazy` 的默认值是 `true`。

目前，在没有建立连接的两个 worker 之间传递消息会出错，目前该行为是实验性的，未来的版本中可能会改变。

25.12 一些值得关注的外部库

除了 Julia 自带的并行机制之外，还有许多外部的库值得一提。例如 MPI.jl 提供了一个 `MPI` 协议的 Julia 的封装，或者是在 共享数组 提到的 DistributedArrays.jl，此外尤其值得一提的是 Julia 的 GPU 编程生态，其包括：

1. 底层（C 内核）的 OpenCL.jl 和 CUDAdrv.jl，分别提供了 OpenCL 和 CUDA 的封装。
2. 底层（Julia 内核）的接口，如 CUDAnative.jl，提供了 Julia 原生的 CUDA 实现。
3. 高层的特定抽象，如 CuArrays.jl 和 CLArrays.jl。
4. 高层的库，如 ArrayFire.jl 和 GPUArrays.jl。

下面的例子将介绍如何用 `DistributedArrays.jl` 和 `CuArrays.jl` 通过 `distribute()` 和 `CuArray()` 将数组分配到多个进程。

记住在载入 `DistributedArrays.jl` 时，需要用 `@everywhere` 将其载入到多个进程中。

```
$ ./julia -p 4

julia> addprocs()

julia> @everywhere using DistributedArrays

julia> using CUDA

julia> B = ones(10_000) ./ 2;

julia> A = ones(10_000) .* π;

julia> C = 2 .* A ./ B;

julia> all(C .≈ 4*π)
true

julia> typeof(C)
Array{Float64,1}

julia> dB = distribute(B);

julia> dA = distribute(A);

julia> dC = 2 .* dA ./ dB;

julia> all(dC .≈ 4*π)
true

julia> typeof(dC)
DistributedArrays.DArray{Float64,1,Array{Float64,1}}

julia> cuB = CuArray(B);

julia> cuA = CuArray(A);

julia> cuC = 2 .* cuA ./ cuB;

julia> all(cuC .≈ 4*π);
true

julia> typeof(cuC)
CuArray{Float64,1}
```

In the following example we will use both `DistributedArrays.jl` and `CUDA.jl` to distribute an array across multiple processes and call a generic function on it.

```
function power_method(M, v)
    for i in 1:100
        v = M*v
        v /= norm(v)
    end

    return v, norm(M*v) / norm(v)  # or  (M*v) ./ v
```

```
end
```

power_method 重复创建一个新的向量然后对其归一化，这里并没有在函数中指定类型信息，来看看是否对前面提到的类型适用：

```
julia> M = [2. 1; 1 1];

julia> v = rand(2)
2-element Array{Float64,1}:
0.40395
0.445877

julia> power_method(M,v)
([0.850651, 0.525731], 2.618033988749895)

julia> cuM = CuArray(M);

julia> cuv = CuArray(v);

julia> curesult = power_method(cuM, cuv);

julia> typeof(curesult)
CuArray{Float64,1}

julia> dM = distribute(M);

julia> dv = distribute(v);

julia> dC = power_method(dM, dv);

julia> typeof(dC)
Tuple{DistributedArrays.DArray{Float64,1,Array{Float64,1}},Float64}
```

最后，我们来看看 MPI.jl，这个库时 Julia 对 MPI 协议的封装。一一介绍其中的每个函数太累赘了，这里领会其实现协议的方法就够了。

考虑下面这个简单的脚本，它做的只是调用每个子进程，然后初始化其 rank，然后在 master 访问时，对 rank 求和。

```
import MPI

MPI.Init()

comm = MPI.COMM_WORLD
MPI.Barrier(comm)

root = 0
r = MPI.Comm_rank(comm)

sr = MPI.Reduce(r, MPI.SUM, root, comm)

if(MPI.Comm_rank(comm) == root)
   @printf("sum of ranks: %s\n", sr)
```

```
end

MPI.Finalize()
```

```
mpirun -np 4 ./julia example.jl
```

[1] In this context, MPI refers to the MPI-1 standard. Beginning with MPI-2, the MPI standards committee introduced a new set of communication mechanisms, collectively referred to as Remote Memory Access (RMA). The motivation for adding rma to the MPI standard was to facilitate one-sided communication patterns. For additional information on the latest MPI standard, see https://mpi-forum.org/docs.

Chapter 26

运行外部程序

Julia 中命令的反引号记法借鉴于 shell、Perl 和 Ruby。然而，在 Julia 中编写

```
julia> `echo hello`
`echo hello`
```

在多个方面上与 shell、Perl 和 Ruby 中的行为有所不同：

- 反引号创建一个 Cmd 对象来表示命令，而不是立即运行命令。你可以使用此对象将命令通过管道连接到其它命令、run 它以及对它进行 read 或 write。
- 在命令运行时，Julia 不会捕获命令的输出结果，除非你对它专门安排。相反，在默认情况下，命令的输出会被定向到 stdout，因为它将使用 libc 的 system 调用。
- 命令从不会在 shell 中运行。相反地，Julia 会直接解析命令语法，适当地插入变量并像 shell 那样拆分单词，同时遵从 shell 的引用语法。命令会作为 julia 的直接子进程运行，使用 fork 和 exec 调用。

Note

下面假设在 Linux 或 MacOS 上使用 Posix 环境。在 Windows 上，许多类似的命令，例如 echo 和 dir，不是外部程序，而是内置在 shell cmd.exe 本身中。运行这些命令的一种选择是调用 cmd.exe，例如 cmd /C echo hello。或者，Julia 可以在 Posix 环境中运行，例如 Cygwin。

这是运行外部程序的简单示例：

```
julia> mycommand = `echo hello`
`echo hello`

julia> typeof(mycommand)
Cmd

julia> run(mycommand);
hello
```

hello 是 echo 命令的输出，发送到 stdout。如果外部命令无法成功运行，则 run 方法会抛出 ProcessFailedException。

如果要读取外部命令的输出，可以使用 read 或 readchomp 代替：

```
julia> read(`echo hello`, String)
"hello\n"

julia> readchomp(`echo hello`)
"hello"
```

更一般地，你可以使用 open 来读取或写入外部命令。

```
julia> open(`less`, "w", stdout) do io
           for i = 1:3
               println(io, i)
           end
       end
1
2
3
```

命令中的程序名称和各个参数可以访问和迭代，这就好像命令也是一个字符串数组：

```
julia> collect(`echo "foo bar"`)
2-element Vector{String}:
 "echo"
 "foo bar"

julia> `echo "foo bar"`[2]
"foo bar"
```

26.1 插值

假设你想要做的事情更复杂，并使用以变量 file 表示的文件名作为命令的参数。那你可以像在字符串字面量中那样使用 $ 进行插值：

```
julia> file = "/etc/passwd"
"/etc/passwd"

julia> `sort $file`
`sort /etc/passwd`
```

通过 shell 运行外部程序的一个常见陷阱是，如果文件名中包含 shell 中的特殊字符，那么可能会导致不希望出现的行为。例如，假设我们想要对其内容进行排序的文件是 /Volumes/External HD/data.csv，而不是 /etc/passwd。让我们来试试：

```
julia> file = "/Volumes/External HD/data.csv"
"/Volumes/External HD/data.csv"

julia> `sort $file`
`sort '/Volumes/External HD/data.csv'`
```

文件名是如何被引用的？Julia 知道 `file` 是作为单个参数插入的，因此它替你引用了此单词。事实上，这不太准确：`file` 的值始终不会被 shell 解释，因此并不需要实际引用；插入引号只是为了展现给用户。就算你把值作为 shell 单词的一部分插入，这也可以工作：

```
julia> path = "/Volumes/External HD"
"/Volumes/External HD"

julia> name = "data"
"data"

julia> ext = "csv"
"csv"

julia> `sort $path/$name.$ext`
`sort '/Volumes/External HD/data.csv'`
```

如你所见，`path` 变量中的空格被恰当地转义了。但是，如果你想插入多个单词怎么办？在此情况下，只需使用数组（或其它可迭代容器）：

```
julia> files = ["/etc/passwd","/Volumes/External HD/data.csv"]
2-element Vector{String}:
 "/etc/passwd"
 "/Volumes/External HD/data.csv"

julia> `grep foo $files`
`grep foo /etc/passwd '/Volumes/External HD/data.csv'`
```

如果将数组作为 shell 单词的一部分插入，Julia 将模拟 shell 的 `{a,b,c}` 参数生成：

```
julia> names = ["foo","bar","baz"]
3-element Vector{String}:
 "foo"
 "bar"
 "baz"

julia> `grep xylophone $names.txt`
`grep xylophone foo.txt bar.txt baz.txt`
```

此外，若在同一单词中插入多个数组，则将模拟 shell 的笛卡尔积生成行为：

```
julia> names = ["foo","bar","baz"]
3-element Vector{String}:
 "foo"
 "bar"
 "baz"

julia> exts = ["aux","log"]
2-element Vector{String}:
 "aux"
 "log"

julia> `rm -f $names.$exts`
`rm -f foo.aux foo.log bar.aux bar.log baz.aux baz.log`
```

因为可以插入字面量数组，所以你可以使用此生成功能，而无需先创建临时数组对象：

```
julia> `rm -rf $["foo","bar","baz","qux"].$["aux","log","pdf"]`
`rm -rf foo.aux foo.log foo.pdf bar.aux bar.log bar.pdf baz.aux baz.log baz.pdf qux.aux qux.log
↪  qux.pdf`
```

26.2 引用

不可避免地，我们会想要编写不那么简单的命令，且有必要使用引号。下面是 shell 提示符下单行 Perl 程序的简单示例：

```
sh$ perl -le '$|=1; for (0..3) { print }'
0
1
2
3
```

该 Perl 表达式需要使用单引号有两个原因：一是为了避免空格将表达式分解为多个 shell 单词，二是为了在使用像 `$|`（是的，这在 Perl 中是变量名）这样的 Perl 变量时避免发生插值。在其它情况下，你可能想要使用双引号来真的进行插值：

```
sh$ first="A"
sh$ second="B"
sh$ perl -le '$|=1; print for @ARGV' "1: $first" "2: $second"
1: A
2: B
```

总之，Julia 反引号语法是经过精心设计的，因此你可以只是将 shell 命令剪切并粘贴到反引号中，接着它们将会工作：转义、引用和插值行为与 shell 相同。唯一的不同是，插值是集成的并且知道在 Julia 的概念中什么是单个字符串值、什么是多个值的容器。让我们在 Julia 中尝试上面的两个例子：

```
julia> A = `perl -le '$|=1; for (0..3) { print }'`
`perl -le '$|=1; for (0..3) { print }'`

julia> run(A);
0
1
2
3

julia> first = "A"; second = "B";

julia> B = `perl -le 'print for @ARGV' "1: $first" "2: $second"`
`perl -le 'print for @ARGV' '1: A' '2: B'`

julia> run(B);
1: A
2: B
```

结果是相同的，且 Julia 的插值行为模仿了 shell 的并对其做了一些改进，因为 Julia 支持头等的可迭代对象，但大多数 shell 通过使用空格分隔字符串来实现这一点，而这又引入了歧义。在尝试将 shell

命令移植到 Julia 中时，请先试着剪切并粘贴它。因为 Julia 会在运行命令前向你显示命令，所以你可以在不造成任何破坏的前提下轻松并安全地检查命令的解释。

26.3 管道

Shell 元字符，如 |、& 和 >，在 Julia 的反引号中需被引用（或转义）：

```
julia> run(`echo hello '|' sort`);
hello | sort

julia> run(`echo hello \| sort`);
hello | sort
```

此表达式调用 echo 命令并以三个单词作为其参数：hello、| 和 sort。结果是只打印了一行：hello | sort。那么，如何构造管道呢？为此，请使用 pipeline，而不是在反引号内使用 '|'：

```
julia> run(pipeline(`echo hello`, `sort`));
hello
```

这将 echo 命令的输出传输到 sort 命令中。当然，这不是很有趣，因为只有一行要排序，但是我们的当然可以做更多、更有趣的事：

```
julia> run(pipeline(`cut -d: -f3 /etc/passwd`, `sort -n`, `tail -n5`))
210
211
212
213
214
```

这将打印在 UNIX 系统上最高的五个用户 ID。cut、sort 和 tail 命令都是当前 julia 进程的直接子进程，这中间没有 shell 进程的干预。Julia 自己负责设置管道和连接文件描述符，而这通常由 shell 完成。因为 Julia 自己做了这些事，所以它能更好的控制并做 shell 做不到的一些事情。

Julia 可以并行地运行多个命令：

```
julia> run(`echo hello` & `echo world`);
world
hello
```

这里的输出顺序是不确定的，因为两个 echo 进程几乎同时启动，并且争着先写入 stdout 描述符和 julia 父进程。Julia 允许你将这两个进程的输出通过管道传输到另一个程序：

```
julia> run(pipeline(`echo world` & `echo hello`, `sort`));
hello
world
```

在 UNIX 管道方面，这里发生的是，一个 UNIX 管道对象由两个 echo 进程创建和写入，管道的另一端由 sort 命令读取。

IO 重定向可以通过向 pipeline 函数传递关键字参数 stdin、stdout 和 stderr 来实现：

```
pipeline(`do_work`, stdout=pipeline(`sort`, "out.txt"), stderr="errs.txt")
```

避免管道中的死锁

在单个进程中读取和写入管道的两端时，避免强制内核缓冲所有数据是很重要的。

例如，在读取命令的所有输出时，请调用 read(out, String)，而非 wait(process)，因为前者会积极地消耗由该进程写入的所有数据，而后者在等待读取者连接时会尝试将数据存储内核的缓冲区中。

另一个常见的解决方案是将读取者和写入者分离到单独的 Task 中：

```
writer = @async write(process, "data")
reader = @async do_compute(read(process, String))
wait(writer)
fetch(reader)
```

（通常，reader 不是一个单独的任务，因为无论如何我们都会立即 fetch 它）。

复杂示例

高级编程语言、头等的命令抽象以及进程间管道的自动设置，三者组合起来非常强大。为了更好地理解可被轻松创建的复杂管道，这里有一些更复杂的例子，以避免对单行 Perl 程序的滥用。

```
julia> prefixer(prefix, sleep) = `perl -nle '$|=1; print "'$prefix' ", $_; sleep '$sleep';'`;

julia> run(pipeline(`perl -le '$|=1; for(0..5){ print; sleep 1 }'`, prefixer("A",2) &
↪   prefixer("B",2)));
B 0
A 1
B 2
A 3
B 4
A 5
```

这是一个经典的例子，一个生产者为两个并发的消费者提供内容：一个 perl 进程生成从数字 0 到 5 的行，而两个并行进程则使用该输出，一个行首加字母「A」，另一个行首加字母「B」。哪个进程使用第一行是不确定的，但是一旦赢得了竞争，这些行会先后被其中一个进程及另一个进程交替使用。（在 Perl 中设置 $|=1 会导致每个 print 语句刷新 stdout 句柄，这是本例工作所必需的。此外，所有输出将被缓存并一次性打印到管道中，以便只由一个消费者进程读取。）

这是一个更加复杂的多阶段生产者——消费者示例：

```
julia> run(pipeline(`perl -le '$|=1; for(0..5){ print; sleep 1 }'`,
           prefixer("X",3) & prefixer("Y",3) & prefixer("Z",3),
           prefixer("A",2) & prefixer("B",2)));
A X 0
B Y 1
A Z 2
B X 3
A Y 4
B Z 5
```

此示例与前一个类似，不同之处在于本例中的消费者有两个阶段，并且阶段间有不同的延迟，因此它们使用不同数量的并行 worker 来维持饱和的吞吐量。

我们强烈建议你尝试所有这些例子，以便了解它们的工作原理。

26.4 Cmd 对象

反引号语法创建一个 Cmd 类型的对象。此类对象也可以直接从现有的 Cmd 或参数列表构造：

```
run(Cmd(`pwd`, dir=".."))
run(Cmd(["pwd"], detach=true, ignorestatus=true))
```

这允许你通过关键字参数指定 Cmd 的执行环境的几个方面。例如，dir 关键字提供对 Cmd 工作目录的控制：

```
julia> run(Cmd(`pwd`, dir="/"));
/
```

并且 env 关键字允许您设置执行环境变量：

```
julia> run(Cmd(`sh -c "echo foo \$HOWLONG"`, env=("HOWLONG" => "ever!",)));
foo ever!
```

有关其它关键字参数，请参阅 Cmd。setenv 和 addenv 命令分别提供了另一种替换或添加到 Cmd 执行环境变量的方法：

```
julia> run(setenv(`sh -c "echo foo \$HOWLONG"`, ("HOWLONG" => "ever!",)));
foo ever!

julia> run(addenv(`sh -c "echo foo \$HOWLONG"`, "HOWLONG" => "ever!"));
foo ever!
```

Chapter 27

调用 C 和 Fortran 代码

在数值计算领域，尽管有很多用 C 语言或 Fortran 写的高质量且成熟的库都可以用 Julia 重写，但为了便捷利用现有的 C 或 Fortran 代码，Julia 提供简洁且高效的调用方式。Julia 的哲学是 `no boilerplate`：Julia 可以直接调用 C/Fortran 的函数，不需要任何" 胶水" 代码，代码生成或其它编译过程——即使在交互式会话 (REPL/Jupyter notebook) 中使用也一样。This is accomplished just by making an appropriate call with the `@ccall` macro (or the less convenient `ccall` syntax, see the `ccall` syntax section).

被调用的代码必须是一个共享库（.so, .dylib, .dll）。大多数 C 和 Fortran 库都已经是以共享库的形式发布的，但在用 GCC 或 Clang 编译自己的代码时，需要添加 `-shared` 和 `-fPIC` 编译器选项。由于 Julia 的 JIT 生成的机器码跟原生 C 代码的调用是一样，所以在 Julia 里调用 C/Fortran 库的额外开销与直接从 C 里调用是一样的。[1]

默认情况下，Fortran 编译器会进行名称修饰（例如，将函数名转换为小写或大写，通常会添加下划线），要调用 Fortran 函数，传递的标识符必须与 Fortran 编译器名称修饰之后的一致。此外，在调用 Fortran 函数时，**所有**输入必须以指针形式传递，并已在堆或栈上分配内存。这不仅适用于通常是堆分配的数组及可变对象，而且适用于整数和浮点数等标量值，尽管这些值通常是栈分配的，且在使用 C 或 Julia 调用约定时通常是通过寄存器传递的。

The syntax for `@ccall` to generate a call to the library function is:

```
@ccall library.function_name(argvalue1::argtype1, ...)::returntype
@ccall function_name(argvalue1::argtype1, ...)::returntype
@ccall $function_pointer(argvalue1::argtype1, ...)::returntype
```

where `library` is a string constant or literal (but see Non-constant Function Specifications below). The library may be omitted, in which case the function name is resolved in the current process. This form can be used to call C library functions, functions in the Julia runtime, or functions in an application linked to Julia. The full path to the library may also be specified. Alternatively, `@ccall` may also be used to call a function pointer `$function_pointer`, such as one returned by `Libdl.dlsym`. The `argtypes` corresponds to the C-function signature and the `argvalues` are the actual argument values to be passed to the function.

Note

请参阅下文了解如何 将 C 类型映射到 Julia 类型。

作为一个完整但简单的例子，下面从大多数 Unix 派生系统上的标准 C 库中调用 `clock` 函数：

```
julia> t = @ccall clock()::Int32
2292761

julia> typeof(t)
Int32
```

clock 不接受任何参数并返回一个 Int32。要调用 getenv 函数来获取指向环境变量值的指针，可以这样调用：

```
julia> path = @ccall getenv("SHELL"::Cstring)::Cstring
Cstring(@0x00007fff5fbffc45)

julia> unsafe_string(path)
"/bin/bash"
```

在实践中，尤其是在提供可重用功能时，通常会在 Julia 函数中包装 @ccall 使用，这些函数设置参数，然后以 C 或 Fortran 函数指定的任何方式检查错误。如果发生错误，它会作为普通的 Julia 异常抛出。这一点尤其重要，因为 C 和 Fortran API 在它们指示错误条件的方式上是出了名的不一致。例如，getenv C 库函数被包裹在下面的 Julia 函数中，它是 env.jl 实际定义的简化版本：

```
function getenv(var::AbstractString)
    val = @ccall getenv(var::Cstring)::Cstring
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    return unsafe_string(val)
end
```

C 函数 getenv 通过返回 C_NULL 的方式进行报错，但是其他 C 标准库函数也会通过不同的方式来报错，这包括返回 -1，0，1 以及其它特殊值。此封装能够抛出异常信息，即是否调用者在尝试获取一个不存在的环境变量：

```
julia> getenv("SHELL")
"/bin/bash"

julia> getenv("FOOBAR")
ERROR: getenv: undefined variable: FOOBAR
```

这是一个稍微复杂的示例，用于发现本地计算机的主机名。

```
function gethostname()
    hostname = Vector{UInt8}(undef, 256) # MAXHOSTNAMELEN
    err = @ccall gethostname(hostname::Ptr{UInt8}, sizeof(hostname)::Csize_t)::Int32
    Base.systemerror("gethostname", err != 0)
    hostname[end] = 0 # ensure null-termination
    return GC.@preserve hostname unsafe_string(pointer(hostname))
end
```

此示例首先分配一个字节数组。然后它调用 C 库函数 gethostname 以使用主机名填充数组。最后，它接受一个指向主机名缓冲区的指针，并将该指针转换为一个 Julia 字符串，假设它是一个以 null 结尾的 C 字符串。

C 库通常使用这种模式，要求调用者分配要传递给被调用者并填充的内存。像这样从 Julia 分配内存通常是通过创建一个未初始化的数组并将指向其数据的指针传递给 C 函数来完成的。这就是我们在这里不使用 `Cstring` 类型的原因：由于数组未初始化，它可能包含 null 字节。作为 `@ccall` 的一部分，转换为 `Cstring` 会检查包含的 null 字节，因此可能会引发类型转换错误。

用 `unsafe_string` 取消引用 `pointer(hostname)` 是一种不安全的操作，因为它需要访问为 `hostname` 分配的内存，而这些内存可能在同时被垃圾收集。宏 `GC.@preserve` 防止这种情况发生，从而防止访问无效的内存位置。

Finally, here is an example of specifying a library via a path. We create a shared library with the following content

```
#include <stdio.h>

void say_y(int y)
{
    printf("Hello from C: got y = %d.\n", y);
}
```

and compile it with `gcc -fPIC -shared -o mylib.so mylib.c`. It can then be called by specifying the (absolute) path as the library name:

```
julia> @ccall "./mylib.so".say_y(5::Cint)::Cvoid
Hello from C: got y = 5.
```

27.1 创建和 C 兼容的 Julia 函数指针

可以将 Julia 函数传递给接受函数指针参数的原生 C 函数。例如，要匹配满足下面的 C 原型：

```
typedef returntype (*functiontype)(argumenttype, ...)
```

宏 `@cfunction` 为调用 Julia 函数生成 C 兼容函数指针。`@cfunction` 的参数是：

1. 一个 Julia 函数
2. 函数的返回值类型
3. 输入类型的元组，对应于函数签名

> **Note**
>
> 与 `ccall` 一样，返回类型和输入类型必须是字面量常量。

> **Note**
>
> 目前，仅支持平台默认的 C 调用约定。这意味着，`@cfunction` 生成的指针不能用于 WINAPI 要求在 32 位 Windows 上使用 `stdcall` 函数的调用中，但可以在 WIN64 上使用（其中 `stdcall` 与 C 调用约定统一）。

Note

Callback functions exposed via `@cfunction` should not throw errors, as that will return control to the Julia runtime unexpectedly and may leave the program in an undefined state.

A classic example is the standard C library `qsort` function, declared as:

```
void qsort(void *base, size_t nitems, size_t size,
           int (*compare)(const void*, const void*));
```

`base` 参数是一个指向长度为 `nitems` 的数组的指针，每个元素都有 `size` 字节。`compare` 是一个回调函数，它采用指向两个元素 `a` 和 `b` 的指针，如果 `a` 出现在 `b` 之前/之后，则返回小于/大于零的整数（如果允许任何顺序，则返回零）。

现在，假设我们在 Julia 中有一个 1 维数组 `A`，我们希望使用 `qsort` 函数（而不是 Julia 的内置 `sort` 函数）对其进行排序。在我们考虑调用 `qsort` 并传递参数之前，我们需要编写一个比较函数：

```
julia> function mycompare(a, b)::Cint
           return (a < b) ? -1 : ((a > b) ? +1 : 0)
       end;
```

`qsort` 需要一个返回 C `int` 的比较函数，因此我们将返回类型注释为 `Cint`。

为了将此函数传递给 C，我们使用宏 `@cfunction` 获取它的地址：

```
julia> mycompare_c = @cfunction(mycompare, Cint, (Ref{Cdouble}, Ref{Cdouble}));
```

`@cfunction` 需要三个参数: Julia 函数 (`mycompare`), 返回值类型 (`Cint`), 和一个输入参数类型的字面量元组, 此处是要排序的 `Cdouble`(`Float64`) 元素的数组.

`qsort` 的最终调用看起来是这样的：

```
julia> A = [1.3, -2.7, 4.4, 3.1];

julia> @ccall qsort(A::Ptr{Cdouble}, length(A)::Csize_t, sizeof(eltype(A))::Csize_t,
↪   mycompare_c::Ptr{Cvoid})::Cvoid

julia> A
4-element Vector{Float64}:
 -2.7
  1.3
  3.1
  4.4
```

如示例所示，原始 Julia 数组 `A` 现在已排序：`[-2.7, 1.3, 3.1, 4.4]`。请注意，Julia 负责将数组转换为 `Ptr{Cdouble}`)，计算元素类型的大小（以字节为单位），等等。

为了好玩，尝试在 `mycompare` 中插入一行 `println("mycompare($a, $b)")`，这将允许你查看 `qsort` 正在执行的比较（并验证它是否真的在调用你传递给它的 Julia 函数）。

27.2 将 C 类型映射到 Julia

将声明的 C 类型与其在 Julia 中的声明完全匹配至关重要。不一致会导致在一个系统上正常工作的代码在另一个系统上失败或产生不确定的结果。

请注意，在调用 C 函数的过程中没有任何地方使用 C 头文件：您有责任确保您的 Julia 类型和调用签名准确反映 C 头文件中的那些。[2]

自动类型转换

Julia 会自动插入对 Base.cconvert 函数的调用，以将每个参数转换为指定的类型。例如，以下调用：

```
@ccall "libfoo".foo(x::Int32, y::Float64)::Cvoid
```

将表现得好像它是这样写的：

```
@ccall "libfoo".foo(
    Base.unsafe_convert(Int32, Base.cconvert(Int32, x))::Int32,
    Base.unsafe_convert(Float64, Base.cconvert(Float64, y))::Float64
    )::Cvoid
```

Base.cconvert 通常只调用 convert，但可以定义为返回一个更适合传递给 C 的任意新对象。这应该用于执行 C 代码将访问的内存。例如，这用于将对象（例如字符串）的 Array 转换为指针数组。

Base.unsafe_convert 处理到 Ptr 类型转换。它被认为是不安全的，因为将对象转换为本地指针会隐藏垃圾收集器中的对象，导致它过早地被释放。

类型对应

首先，让我们回顾一些相关的 Julia 类型术语：

Bits Types

有几种特殊类型需要注意，因为没有其他类型可以定义为具有相同的行为：

- Float32

  和 C 语言中的 float 类型完全对应（以及 Fortran 中的 REAL*4 ）

- Float64

  和 C 语言中的 double 类型完全对应（以及 Fortran 中的 REAL*8 ）

- ComplexF32

  和 C 语言中的 complex float 类型完全对应（以及 Fortran 中的 COMPLEX*8 ）

- ComplexF64

  和 C 语言中的 complex double 类型完全对应（以及 Fortran 中的 COMPLEX*16 ）

- Signed

  和 C 语言中的 signed 类型标识完全对应（以及 Fortran 中的任意 INTEGER 类型）Julia 中任何不是Signed 的子类型的类型，都会被认为是 unsigned 类型。

语法 / 关键字	例子	描述
mutable struct	BitSet	Leaf Type：包含 type-tag 的一组相关数据，由 Julia GC 管理，通过 object-identity 来定义。为了保证实例可以被构造，Leaf Type 必须是完整定义的，即不允许使用 TypeVars。
abstract type	Any, AbstractArray{T, N}, Complex{T}	Super Type：用于描述一组类型，它不是 Leaf-Type，也无法被实例化。
T{A}	Vector{Int}	Type Parameter：某种类型的一种具体化，通常用于分派或存储优化。
		TypeVar：Type parameter 声明中的 T 是一个 TypeVar，它是类型变量的简称。
primitive type	Int, Float64	Primitive Type：一种没有成员变量的类型，但是它有大小。它是按值存储和定义的。
struct	Pair{Int, Int}	"Struct" :: 所有字段都定义为常量的类型。它是按值定义的，并且可以与类型标签一起存储。
	ComplexF64 (isbits)	"Is-Bits" :: 一个 primitive type，或者一个 struct 类型，其中所有字段都是其他 isbits 类型。它是按值定义的，并且在没有类型标签的情况下存储。
struct ...; end	nothing	Singleton：没有成员变量的 Leaf Type 或 Struct。
(...) or tuple(...)	(1, 2, 3)	"元组" :: 类似于匿名结构类型或常量数组的不可变数据结构。表示为数组或结构。

- Ref{T}

  和 Ptr{T} 行为相同，能通过 Julia 的 GC 管理其内存。

- Array{T,N}

  当数组作为 Ptr{T} 参数传递给 C 时，它不是重新解释转换：Julia 要求数组的元素类型与 T 匹配，并传递第一个元素的地址。

  因此，如果一个 Array 中的数据格式不正确，它必须被显式地转换，通过类似 trunc.(Int32, A) 的函数。

  若要将一个数组 A 以不同类型的指针传递，而不提前转换数据，（比如，将一个 Float64 数组传给一个处理原生字节的函数时），你可以将这一参数声明为 Ptr{Cvoid} 。

  如果一个元素类型为 Ptr{T} 的数组作为 Ptr{Ptr{T}} 类型的参数传递，Base.cconvert 将会首先尝试进行 null-terminated copy（即直到下一个元素为 null 才停止复制），并将每一个元素使用其通过 Base.cconvert 转换后的版本替换。这允许，比如，将一个 argv 的指针数组，其类型为 Vector{String} ，传递给一个类型为 Ptr{Ptr{Cchar}} 的参数。

在我们目前支持的所有系统上，基本的 C/C++ 值类型可以转换为 Julia 类型，如下所示。每个 C 类型还有一个对应的同名 Julia 类型，以 C 为前缀。这在编写可移植代码时很有帮助（记住 C 中的 int 与 Julia 中的 Int 不同）。

独立于系统的类型

Cstring 类型本质上是 Ptr{UInt8} 的同义词，但如果 Julia 字符串包含任何嵌入的 null 字符，则类型转换为 Cstring 会引发错误（如果 C 例程将 null 视为终止符，则会导致字符串被静默截断）。如果要将 char* 传递给不采用 null 终止的 C 例程（例如，因为传递的是显式字符串长度），或者如果确定

C 类型	Fortran 类型	标准 Julia 别名	Julia 基本类型
unsigned char	CHARACTER	Cuchar	UInt8
bool (_Bool in C99+)		Cuchar	UInt8
short	INTEGER*2, LOGICAL*2	Cshort	Int16
unsigned short		Cushort	UInt16
int, BOOL (C, typical)	INTEGER*4, LOGICAL*4	Cint	Int32
unsigned int		Cuint	UInt32
long long	INTEGER*8, LOGICAL*8	Clonglong	Int64
unsigned long long		Culonglong	UInt64
intmax_t		Cintmax_t	Int64
uintmax_t		Cuintmax_t	UInt64
float	REAL*4i	Cfloat	Float32
double	REAL*8	Cdouble	Float64
complex float	COMPLEX*8	ComplexF32	Complex{Float32}
complex double	COMPLEX*16	ComplexF64	Complex{Float64}
ptrdiff_t		Cptrdiff_t	Int
ssize_t		Cssize_t	Int
size_t		Csize_t	UInt
void			Cvoid
void and [[noreturn]] or _Noreturn			Union{}
void*			Ptr{Cvoid} (或类似的 Ref{Cvoid})
T* (where T represents an appropriately defined type)			Ref{T} （只有当 T 是 isbits 类型时，T 才可以安全地转变）
char* (or char[], e.g. a string)	CHARACTER*N		Cstring if null-terminated, or Ptr{UInt8} if not
char** (or *char[])			Ptr{Ptr{UInt8}}
jl_value_t* (any Julia Type)			Any
jl_value_t* const* (一个 Julia 值的引用)			Ref{Any}（常量，因为转变需要写屏障，不可能正确插入）
va_arg			Not supported
... (variadic function specification)			T...（其中 T 是上述类型之一，当使用 ccall 函数时）
... (variadic function specification)			; va_arg1::T、va_arg2::S 等（仅支持 @ccall 宏）

Julia 字符串不包含 null 并希望跳过检查，则可以使用 Ptr{UInt8} 作为参数类型。Cstring 也可以用作 ccall 返回类型，但在这种情况下，它显然不会引入任何额外的检查，只是为了提高调用的可读性。

系统独立类型

Note

调用 Fortran 时，所有输入都必须通过指向堆分配或堆栈分配值的指针传递，因此上述所有类型对应都应在其类型规范周围包含一个额外的 Ptr{..} 或 Ref{..} 包装器。

C 类型	标准 Julia 别名	Julia 基本类型
`char`	`Cchar`	`Int8` (x86, x86_64), `UInt8` (powerpc, arm)
`long`	`Clong`	`Int` (UNIX), `Int32` (Windows)
`unsigned long`	`Culong`	`UInt` (UNIX), `UInt32` (Windows)
`wchar_t`	`Cwchar_t`	`Int32` (UNIX), `UInt16` (Windows)

Warning

对于字符串参数 (`char*`)，Julia 类型应该是 `Cstring`（如果需要以 null 结尾的数据），否则为 `Ptr{Cchar}` 或 `Ptr{UInt8}`（这两种指针类型具有相同的效果），如上所述，而不是 `String`。类似地，对于数组参数（`T[]` 或 `T*`），Julia 类型应该还是 `Ptr{T}`，而不是 `Vector{T}`。

Warning

Julia 的 `Char` 类型是 32 位，这与所有平台上的宽字符类型（`wchar_t` 或 `wint_t`）不同。

Warning

`Union{}` 的返回类型意味着函数不会返回，即 C++11 `[[noreturn]]` 或 C11 `_Noreturn`（例如 `jl_throw` 或 `longjmp`）。不要将此用于不返回值（`void`）但返回的函数，对于这些函数，使用 `Cvoid`。

Note

对于 `wchar_t*` 参数，Julia 类型应为 `Cwstring`（如果 C 例程需要以 null 结尾的字符串），否则为 `Ptr{Cwchar_t}`。另请注意，Julia 中的 UTF-8 字符串数据在内部以 null 结尾，因此可以将其传递给需要以 null 结尾的数据的 C 函数，而无需进行复制（但使用 `Cwstring` 类型将导致抛出错误，如果字符串本身包含 null 字符）。

Note

可以在 Julia 中使用 `Ptr{Ptr{UInt8}}` 类型调用采用 `char**` 类型参数的 C 函数。例如，以下形式的 C 函数：

```
int main(int argc, char **argv);
```

可以通过以下 Julia 代码调用：

```
argv = [ "a.out", "arg1", "arg2" ]
@ccall main(length(argv)::Int32, argv::Ptr{Ptr{UInt8}})::Int32
```

Note

对于采用 character(len=*) 类型的可变长度字符串的 Fortran 函数，字符串长度作为隐藏参数提供。这些参数在列表中的类型和位置是特定于编译器的，编译器供应商通常默认使用 Csize_t 作为类型并将隐藏的参数附加到参数列表的末尾。虽然此行为对于某些编译器 (GNU) 是固定的，但其他编译器可选允许将隐藏参数直接放置在字符参数（Intel、PGI）之后。例如，如下的 Fortran 子程序

```
subroutine test(str1, str2)
character(len=*) :: str1,str2
```

can be called via the following Julia code, where the lengths are appended

```
str1 = "foo"
str2 = "bar"
ccall(:test, Cvoid, (Ptr{UInt8}, Ptr{UInt8}, Csize_t, Csize_t),
                    str1, str2, sizeof(str1), sizeof(str2))
```

Warning

Fortran 编译器还可以为指针、假定形状（:）和假定大小（*）数组添加其他隐藏参数。这种行为可以通过使用 ISO_C_BINDING 并在子例程的定义中包含 bind(c) 来避免，强烈推荐用于可互操作的代码。在这种情况下，将没有隐藏的参数，代价是一些语言特性（例如，只允许 character(len=1) 传递字符串）。

Note

声明为返回 Cvoid 的 C 函数将在 Julia 中返回值 nothing。

结构类型对应

复合类型，例如 C 中的 struct 或 Fortran90 中的 TYPE（或 F77 的某些变体中的 STRUCTURE/RECORD），可以通过创建具有相同字段布局的 struct 定义在 Julia 中进行镜像复制。

当递归使用时，isbits 类型被内联存储。所有其他类型都存储为指向数据的指针。在 C 中的另一个结构中镜像复制按值使用的结构时，不要尝试手动复制字段，因为这不会保留正确的字段对齐。相反，建议声明一个 isbits 结构类型并使用它。未命名的结构在翻译为 Julia 时是不可能的。

Julia 不支持压缩结构和联合声明。

如果你事先地知道将具有最大大小（可能包括填充）的字段，则可以获得 union 的近似。将你的字段转换为 Julia 时，将 Julia 字段声明为仅属于该类型。

参数数组可以用 NTuple 表示。例如，C 符号中的 struct 写成

```
struct B {
    int A[3];
};

b_a_2 = B.A[2];
```

可以用 Julia 写成

```
struct B
    A::NTuple{3, Cint}
end

b_a_2 = B.A[3]  # note the difference in indexing (1-based in Julia, 0-based in C)
```

不直接支持未知大小的数组（由 [] 或 [0] 指定的符合 C99 的可变长度结构）。通常处理这些的最好方法是直接处理字节偏移量。例如，如果一个 C 库声明了一个正确的字符串类型并返回一个指向它的指针：

```
struct String {
    int strlen;
    char data[];
};
```

在 Julia 中，我们可以独立访问这些部分以制作该字符串的副本：

```
str = from_c::Ptr{Cvoid}
len = unsafe_load(Ptr{Cint}(str))
unsafe_string(str + Core.sizeof(Cint), len)
```

类型参数

当定义了方法时，@ccall 和 @cfunction 的类型参数被静态地评估。因此，它们必须采用字面量元组的形式，而不是变量，并且不能引用局部变量。

这听起来像是一个奇怪的限制，但请记住，由于 C 不是像 Julia 那样的动态语言，它的函数只能接受具有静态已知的固定签名的参数类型。

然而，虽然必须静态地知道类型布局才能计算预期的 C ABI，但函数的静态参数被视为此静态环境的一部分。函数的静态参数可以用作调用签名中的类型参数，只要它们不影响类型的布局即可。例如，f(x::T) where {T} = @ccall valid(x::Ptr{T})::Ptr{T} 是有效的，因为 Ptr 始终是字大小的原始类型。但是，g(x::T) where {T} = @ccall notvalid(x::T)::T 是无效的，因为 T 的类型布局不是静态已知的。

SIMD 值

注意：此功能目前仅在 64 位 x86 和 AArch64 平台上实现。

如果 C/C++ 例程具有本机 SIMD 类型的参数或返回值，则相应的 Julia 类型是自然映射到 SIMD 类型的 VecElement 的同构元组。具体来说：

- 元组的大小必须与 SIMD 类型相同。例如，一个表示 __m128 的元组

在 x86 上必须有 16 字节的大小。

- 元组的元素类型必须是 VecElement{T} 的一个实例，其中 T 是一个原始类型

是 1、2、4 或 8 个字节。

例如，考虑这个使用 AVX 内在函数的 C 例程：

```
#include <immintrin.h>

__m256 dist( __m256 a, __m256 b ) {
    return _mm256_sqrt_ps(_mm256_add_ps(_mm256_mul_ps(a, a),
                                        _mm256_mul_ps(b, b)));
}
```

以下 Julia 代码使用 ccall 调用 dist：

```
const m256 = NTuple{8, VecElement{Float32}}

a = m256(ntuple(i -> VecElement(sin(Float32(i))), 8))
b = m256(ntuple(i -> VecElement(cos(Float32(i))), 8))

function call_dist(a::m256, b::m256)
    @ccall "libdist".dist(a::m256, b::m256)::m256
end

println(call_dist(a,b))
```

主机必须具有必要的 SIMD 寄存器。例如，上面的代码将无法在没有 AVX 支持的主机上运行。

内存所有权

malloc/free

此类对象的内存分配和释放必须通过调用正在使用的库中的适当清理例程来处理，就像在任何 C 程序中一样。不要尝试在 Julia 中使用 Libc.free 释放从 C 库接收的对象，因为这可能会导致通过错误的库调用 free 函数并导致进程中止。反过来（传递在 Julia 中分配的对象以供外部库释放）同样无效。

何时使用 T、Ptr{T} 以及 Ref{T}

在对外部 C 例程的 Julia 代码包装调用中，普通（非指针）数据应该在 @ccall 中声明为 T 类型，因为它们是通过值传递的。对于接受指针的 C 代码，Ref{T} 通常应用于输入参数的类型，允许通过对Base.cconvert 的隐式调用使用指向 Julia 或 C 管理的内存的指针。相反，被调用的 C 函数返回的指针应该声明为输出类型Ptr{T}，这反映了指向的内存仅由 C 管理。C 结构中包含的指针应在相应的 Julia 结构类型中表示为 Ptr{T} 类型的字段，这些结构类型旨在模拟相应 C 结构的内部结构。

在 Julia 代码包装对外部 Fortran 例程的调用中，所有输入参数都应声明为 Ref{T} 类型，因为 Fortran 通过指向内存位置的指针传递所有变量。Fortran 子程序的返回类型应该是 Cvoid，或者 Fortran 函数的返回类型应该是 T，返回类型是 T。

Chapter 28

将 C 函数映射到 Julia

@ccall / @cfunction 参数翻译指南

将 C 参数列表翻译为 Julia：

- T，其中 T 取值为：char、int、long、short、float、double、complex、enum 或其等价的 typedef 类型
  - T，其中 T 是等价的 Julia Bits 类型（参见上表）
  - 如果 T 是 enum，则参数类型应等价于 Cint 或 Cuint
  - 参数值将被复制（按值传递）
- struct T （包括 struct 的 typedef）
  - T，其中 T 是 Julia 叶类型
  - 参数值将被复制（按值传递）
- void*

```
* 取决于如何使用此参数，首先将其翻译为所需的指针类型，然后使用此列表中的其余规则确定 Julia 等
↪  价项
* 这个参数可以声明为 `Ptr{Cvoid}`，如果它真的只是一个未知的指针
```

- jl_value_t*
  - Any
  - 参数值必须是有效的 Julia 对象
- jl_value_t* const*
  - Ref{Any}
  - 参数列表必须是有效的 Julia 对象（或 C_NULL）
  - 不能用于输出参数，除非用户能够单独安排要 GC 保留的对象
- T*

  - Ref{T}，其中 T 是与 T 对应的 Julia 类型
  - 如果它是 inlinealloc 类型，则将复制参数值（包括 isbits，否则，值必须是有效的 Julia 对象）

- T (*)(...) （例如，指向函数的指针）

  - Ptr{Cvoid}（您可能需要显式使用 @cfunction 来创建此指针）

- ... （例如，可变参数）

对于 ccall ：T...，其中 T 是所有剩余参数的单个 Julia 类型

对于 @ccall ：; va_arg1::T, va_arg2::S, etc，其中 T 和 S 是 Julia 类型（即，使用 ; 将常规参数与可变参数分开）

  - 目前不支持 @cfunction

- va_arg

  - ccall 或 @cfunction 不支持

@ccall / @cfunction 返回类型翻译指南

将 C 返回类型翻译为 Julia：

- void

  - Cvoid（这将返回单例实例 nothing::Cvoid）

- T，其中 T 是原始类型之一：char，int，long，short，float，double，complex，enum 或任何等效的 typedef

  - T，其中 T 是等效的 Julia Bits 类型（请参阅上表）
  - 如果 T 是 enum，则参数类型应等效于 Cint 或 Cuint
  - 参数值将被复制（按值返回）

- struct T （包括 typedef 到结构体）

  - T，其中 T 是 Julia 叶类型
  - 参数值将被复制（按值返回）

- void*

```
* 取决于如何使用此参数，首先将其翻译为所需的指针类型，然后使用此列表中的其余规则确定 Julia
↪  等效项
* 如果它确实只是一个未知指针，则可以将此参数声明为 `Ptr{Cvoid}`
```

- jl_value_t*

  - Any
  - 参数值必须是有效的 Julia 对象

- jl_value_t**

    - Ptr{Any}（Ref{Any} 是无效的返回类型）

- T*

    - 如果内存已由 Julia 拥有，或者是 isbits 类型，并且已知为非空：

        * Ref{T}，其中 T 是对应于 T 的 Julia 类型
        * 返回类型 Ref{Any} 无效，它应该是 Any（对应于 jl_value_t*）或 Ptr{Any}（对应于 jl_value_t**）
        * C 不得修改通过 Ref{T} 返回的内存，如果 T 是 isbits 类型

    - 如果内存由 C 拥有：

        * Ptr{T}，其中 T 是对应于 T 的 Julia 类型

- T (*)(...)（例如，指向函数的指针）

    - Ptr{Cvoid}，以便从 Julia 直接调用此函数，你需要将此作为 ccall 的第一个参数传递。请参阅 间接调用。

传递修改输入的指针

因为 C 不支持多个返回值，所以 C 函数通常会使用指向函数将修改的数据的指针。要在 @ccall 中完成此操作，你需要首先将值封装在适当类型的 Ref{T} 中。当你将这个 Ref 对象作为参数传递时，Julia 会自动传递一个指向封装数据的 C 指针：

```
width = Ref{Cint}(0)
range = Ref{Cfloat}(0)
@ccall foo(width::Ref{Cint}, range::Ref{Cfloat})::Cvoid
```

返回时，可以通过 width[] 和 range[] 检索 width 和 range 的内容（如果它们被 foo 改变的话）；也就是说，它们就像零维数组。

28.1 C 包装器示例

让我们从一个返回 Ptr 类型的 C 包装器的简单示例开始：

```
mutable struct gsl_permutation
end

# The corresponding C signature is
#     gsl_permutation * gsl_permutation_alloc (size_t n);
function permutation_alloc(n::Integer)
    output_ptr = @ccall "libgsl".gsl_permutation_alloc(n::Csize_t)::Ptr{gsl_permutation}
    if output_ptr == C_NULL # Could not allocate memory
        throw(OutOfMemoryError())
    end
    return output_ptr
end
```

GNU 科学图书馆（这里假设可以通过 :libgsl 访问）定义了一个不透明的指针，gsl_permutation *，作为 C 函数 gsl_permutation_alloc 的返回类型。由于用户代码永远不必查看 gsl_permutation 结构内部，相应的 Julia 包装器只需要一个新的类型声明 gsl_permutation 它没有内部字段，其唯一目的

是放置在 Ptr 类型的类型参数中。ccall 的返回类型声明为 Ptr{gsl_permutation}，因为 output_ptr 分配和指向的内存由 C 控制。

输入 n 是按值传递的，因此函数的输入签名被简单地声明为 ::Csize_t，不需要任何 Ref 或 Ptr。（如果包装器改为调用 Fortran 函数，则相应的函数输入签名将改为 ::Ref{Csize_t}，因为 Fortran 变量是通过指针传递的。）此外，n 可以是任何可转换的类型到一个 Csize_t 整数；ccall 隐式调用 Base.cconvert(Csize_t, n)。

这是包装相应析构函数的第二个示例：

```
# The corresponding C signature is
#     void gsl_permutation_free (gsl_permutation * p);
function permutation_free(p::Ptr{gsl_permutation})
    @ccall "libgsl".gsl_permutation_free(p::Ptr{gsl_permutation})::Cvoid
end
```

这是传递 Julia 数组的第三个示例：

```
# The corresponding C signature is
#    int gsl_sf_bessel_Jn_array (int nmin, int nmax, double x,
#                                double result_array[])
function sf_bessel_Jn_array(nmin::Integer, nmax::Integer, x::Real)
    if nmax < nmin
        throw(DomainError())
    end
    result_array = Vector{Cdouble}(undef, nmax - nmin + 1)
    errorcode = @ccall "libgsl".gsl_sf_bessel_Jn_array(
                    nmin::Cint, nmax::Cint, x::Cdouble, result_array::Ref{Cdouble})::Cint
    if errorcode != 0
        error("GSL error code $errorcode")
    end
    return result_array
end
```

包装的 C 函数返回一个整数错误代码；Bessel J 函数的实际评估结果填充 Julia 数组 result_array。这个变量被声明为一个 Ref{Cdouble}，因为它的内存是由 Julia 分配和管理的。对 Base.cconvert(Ref{Cdouble}, result_array) 的隐式调用将指向 Julia 数组数据结构的 Julia 指针解包为 C 可以理解的形式。

28.2 Fortran 包装器示例

以下示例利用 ccall 调用通用 Fortran 库 (libBLAS) 中的函数来计算点积。请注意，这里的参数映射与上面的有点不同，因为我们需要从 Julia 映射到 Fortran。在每个参数类型上，我们指定 Ref 或 Ptr。此修改约定可能特定于你的 Fortran 编译器和操作系统，并且可能未记录在案。但是，将每个包装在一个 Ref（或 Ptr，等效地）中是 Fortran 编译器实现的一个常见要求：

```
function compute_dot(DX::Vector{Float64}, DY::Vector{Float64})
    @assert length(DX) == length(DY)
    n = length(DX)
    incx = incy = 1
    product = @ccall "libLAPACK".ddot(
        n::Ref{Int32}, DX::Ptr{Float64}, incx::Ref{Int32}, DY::Ptr{Float64},
        ↪  incy::Ref{Int32})::Float64
    return product
end
```

28.3 垃圾回收安全

将数据传递给 @ccall 时，最好避免使用 pointer 函数。而是定义一个 Base.cconvert 方法并将变量直接传递给 @ccall。@ccall 自动安排它的所有参数都将从垃圾收集中保留，直到调用返回。如果 C API 将存储对 Julia 分配的内存的引用，则在 @ccall 返回后，你必须确保该对象对垃圾收集器保持可见。建议的方法是创建一个类型为 Array{Ref,1} 的全局变量来保存这些值，直到 C 库通知你它已完成使用它们。

每当你创建了一个指向 Julia 数据的指针时，你必须确保原始数据存在，直到你完成使用该指针。Julia 中的许多方法，例如 unsafe_load 和 String 复制数据而不是获取缓冲区的所有权，因此可以安全地释放（或更改）原始数而不影响 Julia。一个值得注意的例外是 unsafe_wrap，出于性能原因，它共享（或可以被告知拥有）底层缓冲区。

垃圾收集器不保证任何终结顺序。也就是说，如果 a 包含对 b 的引用，并且 a 和 b 都需要进行垃圾回收，则不能保证 b 会在 a 之后完成。如果 a 的正确终结取决于 b 是否有效，则必须以其他方式处理。

28.4 非常数函数规范

在某些情况下，所需库的确切名称或路径是事先未知的，必须在运行时计算。为了处理这种情况，规范的库组件可以是一个函数调用，例如 find_blas().dgemm。调用表达式将在执行 ccall 本身时执行。但是，假设库位置一旦确定就不会改变，因此调用的结果可以被缓存和重用。因此，表达式执行的次数是未指定的，多次调用返回不同的值会导致未指定的行为。

如果需要更大的灵活性，可以通过 eval 分段使用计算值作为函数名称，如下所示：

```
@eval @ccall "lib".$(string("a", "b"))()::Cint
```

此表达式使用 string 构造一个名称，然后将此名称替换为一个新的 @ccall 表达式，然后对其进行评估。请记住，eval 仅在顶层运行，因此在此表达式中局部变量将不可用（除非它们的值被替换为 $）。出于这个原因，eval 通常仅用于形成顶级定义，例如在包装包含许多类似函数的库时。可以为 @cfunction 构造一个类似的示例。

但是，这样做也会很慢并且会泄漏内存，因此你通常应该避免这种情况，而是继续阅读。下一节讨论如何使用间接调用来有效地实现类似的效果。

28.5 间接调用

@ccall 的第一个参数也可以是在运行时计算的表达式。在这种情况下，表达式的计算结果必须为 Ptr，它将用作要调用的本地函数的地址。当第一个 @ccall 参数包含对非常量（例如局部变量、函数参数或非常量全局变量）的引用时，会发生此行为。

例如，你可以通过 dlsym 查找函数，然后将其缓存在该会话的共享引用中。例如：

```
macro dlsym(lib, func)
    z = Ref{Ptr{Cvoid}}(C_NULL)
    quote
        let zlocal = $z[]
            if zlocal == C_NULL
                zlocal = dlsym($(esc(lib))::Ptr{Cvoid}, $(esc(func)))::Ptr{Cvoid}
                $z[] = zlocal
```

```
            end
            zlocal
        end
    end
end

mylibvar = Libdl.dlopen("mylib")
@ccall $(@dlsym(mylibvar, "myfunc"))()::Cvoid
```

28.6 cfunction 闭包

`@cfunction` 的第一个参数可以用 $ 标记，在这种情况下，返回值将改为结束参数的 `struct CFunction`。你必须确保此返回对象保持活动状态，直到完成对它的所有使用。当这个引用被删除和 atexit 时，cfunction 指针处的内容和代码将通过 `finalizer` 删除。这通常不是必需的，因为此功能在 C 中不存在，但对于处理不提供单独的闭包环境参数的设计不良的 API 很有用。

```
function qsort(a::Vector{T}, cmp) where T
    isbits(T) || throw(ArgumentError("this method can only qsort isbits arrays"))
    callback = @cfunction $cmp Cint (Ref{T}, Ref{T})
    # Here, `callback` isa Base.CFunction, which will be converted to Ptr{Cvoid}
    # (and protected against finalization) by the ccall
    @ccall qsort(a::Ptr{T}, length(a)::Csize_t, Base.elsize(a)::Csize_t, callback::Ptr{Cvoid})
    # We could instead use:
    #    GC.@preserve callback begin
    #        use(Base.unsafe_convert(Ptr{Cvoid}, callback))
    #    end
    # if we needed to use it outside of a `ccall`
    return a
end
```

Note

闭包 `@cfunction` 依赖于 LLVM Trampolines，并非在所有平台（例如 ARM 和 PowerPC）上都可用。

28.7 关闭库

关闭（卸载）库以便重新加载有时很有用。例如，在开发与 Julia 一起使用的 C 代码时，可能需要编译、从 Julia 调用 C 代码，然后关闭库、进行编辑、重新编译并加载新的更改。可以重新启动 Julia 或使用 `Libdl` 函数来显式管理库，例如：

```
lib = Libdl.dlopen("./my_lib.so") # Open the library explicitly.
sym = Libdl.dlsym(lib, :my_fcn)   # Get a symbol for the function to call.
@ccall $sym(...) # Use the pointer `sym` instead of the library.symbol tuple.
Libdl.dlclose(lib) # Close the library explicitly.
```

Note that when using `@ccall` with the input (e.g., `@ccall "./my_lib.so".my_fcn(...)::Cvoid`), the library is opened implicitly and it may not be explicitly closed.

28.8 Variadic function calls

To call variadic C functions a `semicolon` can be used in the argument list to separate required arguments from variadic arguments. An example with the `printf` function is given below:

```
julia> @ccall printf("%s = %d\n"::Cstring ; "foo"::Cstring, foo::Cint)::Cint
foo = 3
8
```

28.9 ccall interface

There is another alternative interface to `@ccall`. This interface is slightly less convenient but it does allow one to specify a calling convention.

The arguments to `ccall` are:

1. A `(:function, "library")` pair (most common),

   OR

   a `:function` name symbol or `"function"` name string (for symbols in the current process or libc),

   OR

   a function pointer (for example, from `dlsym`).

2. The function's return type

3. A tuple of input types, corresponding to the function signature. One common mistake is forgetting that a 1-tuple of argument types must be written with a trailing comma.

4. The actual argument values to be passed to the function, if any; each is a separate parameter.

> **Note**
>
> The `(:function, "library")` pair, return type, and input types must be literal constants (i.e., they can't be variables, but see Non-constant Function Specifications).
>
> The remaining parameters are evaluated at compile-time, when the containing method is defined.

A table of translations between the macro and function interfaces is given below.

28.10 Calling Convention

The second argument to `ccall` (immediately preceding return type) can optionally be a calling convention specifier (the `@ccall` macro currently does not support giving a calling convention). Without any specifier, the platform-default C calling convention is used. Other supported conventions are: `stdcall`, `cdecl`, `fastcall`, and `thiscall` (no-op on 64-bit Windows). For example (from `base/libc.jl`) we see the same `gethostnameccall` as above, but with the correct signature for Windows:

```
hn = Vector{UInt8}(undef, 256)
err = ccall(:gethostname, stdcall, Int32, (Ptr{UInt8}, UInt32), hn, length(hn))
```

@ccall	ccall
@ccall clock()::Int32	ccall(:clock, Int32, ())
@ccall f(a::Cint)::Cint	ccall(:a, Cint, (Cint,), a)
@ccall "mylib".f(a::Cint, b::Cdouble)::Cvoid	ccall((:f, "mylib"), Cvoid, (Cint, Cdouble), (a, b))
@ccall $fptr.f()::Cvoid	ccall(fptr, f, Cvoid, ())
@ccall printf("%s = %d\n"::Cstring ; "foo"::Cstring, foo::Cint)::Cint	<unavailable>
@ccall printf("%s = %d\n"::Cstring ; "2 + 2"::Cstring, "5"::Cstring)::Cint	ccall(:printf, Cint, (Cstring, Cstring...), "%s = %s\n", "2 + 2", "5")
<unavailable>	ccall(:gethostname, stdcall, Int32, (Ptr{UInt8}, UInt32), hn, length(hn))

请参阅 LLVM Language Reference 来获得更多信息。

还有一个额外的特殊调用约定 llvmcall，它允许直接插入对 LLVM 内部函数的调用。这在针对不常见的平台（例如 GPGPU）时特别有用。例如，对于CUDA，我们需要能够读取线程索引：

```
ccall("llvm.nvvm.read.ptx.sreg.tid.x", llvmcall, Int32, ())
```

与任何 ccall 一样，参数签名必须完全正确。另外，请注意，与 Core.Intrinsics 开放的等效 Julia 函数不同，没有兼容层级可以确保内在函数有意义并在当前目标上工作。

28.11 访问全局变量

可以使用 cglobal 函数按名称访问本地库导出的全局变量。cglobal 的参数与 ccall 使用相同的符号规范，以及描述存储在变量中的值的类型：

```
julia> cglobal((:errno, :libc), Int32)
Ptr{Int32} @0x00007f418d0816b8
```

结果是一个给出值地址的指针。可以使用 unsafe_load 和 unsafe_store! 通过这个指针来操作该值。

Note

在名为“libc”的库中可能找不到此 errno 符号，因为这是系统编译器的实现细节。通常标准库符号应该只通过名称访问，允许编译器填写正确的符号。然而，这个例子中显示的 errno 符号在大多数编译器中都是特殊的，所以这里看到的值可能不是你所期望或想要的。在任何支持多线程的系统上用 C 编译等效代码通常实际上会调用不同的函数（通过宏预处理器重载），并且可能给出与此处打印的遗留值不同的结果。

28.12 通过指针来访问数据

以下方法被描述为“不安全”，因为错误的指针或类型声明会导致 Julia 突然终止。

给定一个 Ptr{T}，通常可以使用 unsafe_load(ptr, [index]) 将 T 类型的内容从引用的内存复制到 Julia 对象中。index 参数是可选的（默认为 1），并遵循基于 1 的索引的 Julia 惯例。此函数类似于 getindex 和 setindex! 的行为（例如 [] 访问语法）。

返回值将是一个初始化为包含引用内存内容副本的新对象。引用的内存可以安全地释放或释放。

如果 T 是 Any，则假定内存包含对 Julia 对象的引用（jl_value_t*），结果将是对该对象的引用，并且不会复制该对象。在这种情况下，你必须小心确保对象始终对垃圾收集器可见（指针不计数，但新引用计数）以确保内存不会过早释放。请注意，如果对象最初不是由 Julia 分配的，则新对象将永远不会被 Julia 的垃圾收集器终结。如果 Ptr 本身实际上是一个 jl_value_t*，它可以通过 unsafe_pointer_to_objref(ptr) 转换回 Julia 对象引用。(Julia 值 v 可以通过调用 pointer_from_objref(v) 转换为 jl_value_t* 指针，如 Ptr{Cvoid}。)

可以使用 unsafe_store!(ptr, value, [index]) 执行反向操作（将数据写入 Ptr{T}）。目前，这仅支持原始类型或其他无指针（isbits）不可变结构类型。

任何引发错误的操作目前可能尚未实现，应作为错误发布，以便解决。

如果感兴趣的指针是纯数据数组（原始类型或不可变结构），则函数 unsafe_wrap(Array, ptr,dims, own = false) 可能更有用。如果 Julia 应该“获得”底层缓冲区的所有权并在返回的 Array 对象最终确定时调用 free(ptr)，则最后一个参数应该为 true。如果省略了 own 参数或为 false，则调用者必须确保缓冲区一直存在，直到所有访问完成。

Julia 中 Ptr 类型的算术（例如使用 +）与 C 的指针算术的行为不同。将整数添加到 Julia 中的 Ptr 总是将指针移动一定数量的 *bytes*，而不是元素。这样，通过指针运算获得的地址值不依赖于指针的元素类型。

28.13 线程安全

一些 C 库从不同的线程执行它们的回调，并且由于 Julia 不是线程安全的，因此你需要采取一些额外的预防措施。特别是，你需要设置一个两层系统：C 回调应该只 安排（通过 Julia 的事件循环）执行“真实”回调。为此，创建一个 AsyncCondition 对象并在其上创建 wait：

```
cond = Base.AsyncCondition()
wait(cond)
```

传递给 C 的回调应该只通过 ccall 将 cond.handle 作为参数传递给 :uv_async_send 并调用，注意避免任何内存分配操作或与 Julia 运行时的其他交互。

注意，事件可能会合并，因此对 uv_async_send 的多个调用可能会导致对该条件的单个唤醒通知。

28.14 关于 Callbacks 的更多内容

关于如何传递 callback 到 C 库的更多细节，请参考此博客。

28.15 C++

如需封装 C++ 库的工具，即用 C++ 写封装/胶水代码，请参考CxxWrap。

---

[1]Non-library function calls in both C and Julia can be inlined and thus may have even less overhead than calls to shared library functions. The point above is that the cost of actually doing foreign function call is about the same as doing a call in either native language.

[2]The Clang package can be used to auto-generate Julia code from a C header file.

Chapter 29

处理操作系统差异

当编写跨平台的应用或库时，通常需要考虑到操作系统之间的差异。变量 Sys.KERNEL 可以用于这些场合。在 Sys 模块中有一些函数将会使这些事情更加简单：isunix、islinux、isapple、isbsd、isfreebsd 以及 iswindows。这些函数可以按如下方式使用：

```
if Sys.iswindows()
    windows_specific_thing(a)
end
```

注意，islinux、isapple 和 isfreebsd 是 isunix 完全互斥的子集。另外，有一个宏 @static 可以使用这些函数有条件地隐藏无效代码，如以下示例所示。

简单例子：

```
ccall((@static Sys.iswindows() ? :_fopen : :fopen), ...)
```

复杂例子：

```
@static if Sys.islinux()
    linux_specific_thing(a)
elseif Sys.isapple()
    apple_specific_thing(a)
else
    generic_thing(a)
end
```

在链式嵌套的条件表达式中（包括 if/elseif/end），@static 必须在每一层都调用（括号是可选的，但是为了可读性，建议添加）。

```
@static Sys.iswindows() ? :a : (@static Sys.isapple() ? :b : :c)
```

Chapter 30

环境变量

Julia 可以配置许多环境变量，一种常见的方式是直接配置操作系统环境变量，另一种更便携的方式是在 Julia 中配置。假设你要将环境变量 JULIA_EDITOR 设置为 vim，可以直接在 REPL 中输入 ENV["JULIA_EDITOR"] = "vim"（请根据具体情况对此进行修改），也可以将其添加到用户主目录中的配置文件 ~/.julia/config/startup.jl，这样做会使其永久生效。环境变量的当前值是通过执行 ENV["JULIA_EDITOR"] 来确定的。

Julia 使用的环境变量通常以 JULIA 开头。如果调用 InteractiveUtils.versioninfo 时使用关键字参数 verbose = true，那么输出结果将列出与 Julia 相关的已定义环境变量，即包括那些名称中包含 JULIA 的环境变量。

Note

某些变量需要在 Julia 启动之前设置，比如 JULIA_NUM_THREADS 和 JULIA_PROJECT，因为在启动过程中将这些变量添加到 ~/.julia/config/startup.jl 中为时已晚。在 Bash 中，环境变量可以手动设置，这可通过在 Julia 启动前运行诸如 export JULIA_NUM_THREADS=4 的命令，亦可通过向 ~/.bashrc 或 ~/.bash_profile 添加相同命令来在 Bash 每次启动时设置该变量。

30.1 文件位置

JULIA_BINDIR

包含 Julia 可执行文件的目录的绝对路径，它会设置全局变量 Sys.BINDIR。$JULIA_BINDIR 如果没有设置，那么 Julia 会在运行时确定 Sys.BINDIR 的值。

在默认情况下，可执行文件是指：

```
$JULIA_BINDIR/julia
$JULIA_BINDIR/julia-debug
```

全局变量 Base.DATAROOTDIR 是一个从 Sys.BINDIR 到 Julia 数据目录的相对路径。

```
$JULIA_BINDIR/$DATAROOTDIR/julia/base
```

上述路径是 Julia 最初搜索源文件的路径（通过 Base.find_source_file()）。

同样，全局变量 Base.SYSCONFDIR 是一个到配置文件目录的相对路径。在默认情况下，Julia 会在下列文件中搜索 startup.jl 文件（通过 Base.load_julia_startup()）

```
$JULIA_BINDIR/$SYSCONFDIR/julia/startup.jl
$JULIA_BINDIR/../etc/julia/startup.jl
```

例如，在 Linux 下安装的 Julia 可执行文件位于 /bin/julia，DATAROOTDIR 为 ../share，SYSCONFDIR 为 ../etc，JULIA_BINDIR 会被设置为 /bin，会有一个源文件搜索路径：

```
/share/julia/base
```

和一个全局配置文件搜索路径：

```
/etc/julia/startup.jl
```

JULIA_PROJECT

指示哪个项目应该是初始活动项目的目录路径。设置这个环境变量和指定--project 启动选项效果一样，但是--project 优先级更高。如果变量设置为 @. （注意尾随的点），那么 Julia 会尝试从当前目录及其父目录中查找包含 Project.toml 或 JuliaProject.toml 文件的项目目录。另请参阅有关 代码加载 的章节。

> **Note**
>
> JULIA_PROJECT 必须在启动 julia 前定义；于 startup.jl 中定义它对于启动的过程为时已晚。

JULIA_LOAD_PATH

JULIA_LOAD_PATH 环境变量用于补充全局的 Julia 变量 LOAD_PATH ，该变量可用于确定通过 import 和 using 可以加载哪些包（请参阅 Code Loading）。

与 shell 使用的 PATH 变量不同，在 JULIA_LOAD_PATH 中的空条目将会在填充 LOAD_PATH 时被扩展为 LOAD_PATH 的默认值 ["@", "@v#.#", "@stdlib"] 。这样，无论 JULIA_LOAD_PATH 是否已被设置，均可以使用 shell 脚本轻松地在加载路径前面或后面添加值。例如要将 /foo/bar 添加到 LOAD_PATH 之前，只需要使用下列脚本：

```
export JULIA_LOAD_PATH="/foo/bar:$JULIA_LOAD_PATH"
```

如果已经设置了 JULIA_LOAD_PATH 环境变量，那么 /foo/bar 将被添加在原有值之前。另一方面，如果 JULIA_LOAD_PATH 尚未设置，那么它会被设置为 /foo/bar: ，而这将使用 LOAD_PATH 的值扩展为 ["/foo/bar", "@", "@v#.#", "@stdlib"] 。如果 JULIA_LOAD_PATH 被设置为空字符串，那么它将被扩展为一个空的 LOAD_PATH 数组。换句话说，这个空字符串数组将被认为是零元素的数组，而非是一个空字符串单元素的数组。使用这样的加载行为是为了可以通过环境变量设置空的加载路径。如果你需要使用默认的加载路径，请不要设置这一环境变量，如果它必须有值，那么可将其设置为字符串 : 。

> **Note**
>
> 在 Windows 上，路径元素由 ; 字符分隔，就像 Windows 上的大多数路径列表一样。将上一段中的 : 替换为 ;。

JULIA_DEPOT_PATH

JULIA_DEPOT_PATH 环境变量用于填充全局的 Julia 变量 DEPOT_PATH ，该变量用于控制包管理器以及 Juila 代码加载机制在何处查找包注册表、已安装的包、命名环境、克隆的存储库、缓存的预编译包映像、配置文件和 REPL 历史记录文件的默认位置。

与 shell 使用的 PATH 变量不同，但与 JULIA_LOAD_PATH 类似，在 JULIA_DEPOT_PATH 中的空条目将会被扩展为 DEPOT_PATH 的默认值。这样，无论 JULIA_DEPOT_PATH 是否已被设置，均可以使用 shell 脚本轻松地在仓库路径前面或后面添加值。例如要将 /foo/bar 添加到 DEPOT_PATH 之前，只需要使用下列脚本：

```
export JULIA_DEPOT_PATH="/foo/bar:$JULIA_DEPOT_PATH"
```

如果已经设置了 JULIA_DEPOT_PATH 环境变量，那么 /foo/bar 将被添加在原有值之前。另一方面，如果 JULIA_DEPOT_PATH 尚未设置，那么它会被设置为 /foo/bar: ，而这将使 /foo/bar 被添加到默认仓库路径之前。如果 JULIA_DEPOT_PATH 被设置为空字符串，那么它将扩展为一个空的 DEPOT_PATH 数组。换句话说，这个空字符串数组将被认为是零元素的数组，而非是一个空字符串单元素的数组。使用这样的加载行为是为了可以通过环境变量设置空的仓库路径。如果你需要使用默认的仓库路径，请不要设置这一环境变量，如果它必须有值，那么可将其设置为字符串 : 。

> **Note**
>
> 在 Windows 上，路径元素由 ; 字符分隔，就像 Windows 上的大多数路径列表一样。将上一段中的 : 替换为 ;。

> **Note**
>
> JULIA_DEPOT_PATH must be defined before starting julia; defining it in `startup.jl` is too late in the startup process; at that point you can instead directly modify the DEPOT_PATH array, which is populated from the environment variable.

JULIA_HISTORY

REPL 历史文件中 REPL.find_hist_file() 的绝对路径。如果没有设置 $JULIA_HISTORY，那么 REPL.find_hist_file() 默认为

```
$(DEPOT_PATH[1])/logs/repl_history.jl
```

JULIA_MAX_NUM_PRECOMPILE_FILES

Sets the maximum number of different instances of a single package that are to be stored in the precompile cache (default = 10).

JULIA_VERBOSE_LINKING

If set to true, linker commands will be displayed during precompilation.

30.2 Pkg.jl

JULIA_CI

If set to `true`, this indicates to the package server that any package operations are part of a continuous integration (CI) system for the purposes of gathering package usage statistics.

JULIA_NUM_PRECOMPILE_TASKS

The number of parallel tasks to use when precompiling packages. See `Pkg.precompile`.

JULIA_PKG_DEVDIR

The default directory used by `Pkg.develop` for downloading packages.

JULIA_PKG_IGNORE_HASHES

If set to 1, this will ignore incorrect hashes in artifacts. This should be used carefully, as it disables verification of downloads, but can resolve issues when moving files across different types of file systems. See Pkg.jl issue #2317 for more details.

> **Julia 1.6**
>
> This is only supported in Julia 1.6 and above.

JULIA_PKG_OFFLINE

If set to `true`, this will enable offline mode: see `Pkg.offline`.

> **Julia 1.5**
>
> Pkg's offline mode requires Julia 1.5 or later.

JULIA_PKG_PRECOMPILE_AUTO

If set to `0`, this will disable automatic precompilation by package actions which change the manifest. See `Pkg.precompile`.

JULIA_PKG_SERVER

由 `Pkg.jl` 使用，用于下载软件包和更新注册表。默认情况下，Pkg 使用 `https://pkg.julialang.org` 来获取 Julia 包。你可以使用此环境变量来选择不同的服务器。此外，你可以禁用 PkgServer 协议的使用，并通过设置直接从它们的主机（GitHub、GitLab 等）访问包：

```
export JULIA_PKG_SERVER=""
```

JULIA_PKG_SERVER_REGISTRY_PREFERENCE

Specifies the preferred registry flavor. Currently supported values are `conservative` (the default), which will only publish resources that have been processed by the storage server (and thereby have a higher probability

of being available from the PkgServers), whereas `eager` will publish registries whose resources have not necessarily been processed by the storage servers. Users behind restrictive firewalls that do not allow downloading from arbitrary servers should not use the `eager` flavor.

Julia 1.7

This only affects Julia 1.7 and above.

JULIA_PKG_UNPACK_REGISTRY

If set to `true`, this will unpack the registry instead of storing it as a compressed tarball.

Julia 1.7

This only affects Julia 1.7 and above. Earlier versions will always unpack the registry.

JULIA_PKG_USE_CLI_GIT

If set to `true`, Pkg operations which use the git protocol will use an external `git` executable instead of the default libgit2 library.

Julia 1.7

Use of the `git` executable is only supported on Julia 1.7 and above.

JULIA_PKGRESOLVE_ACCURACY

The accuracy of the package resolver. This should be a positive integer, the default is 1.

JULIA_PKG_PRESERVE_TIERED_INSTALLED

Change the default package installation strategy to `Pkg.PRESERVE_TIERED_INSTALLED` to let the package manager try to install versions of packages while keeping as many versions of packages already installed as possible.

Julia 1.9

This only affects Julia 1.9 and above.

30.3 Network transport

JULIA_NO_VERIFY_HOSTS / JULIA_SSL_NO_VERIFY_HOSTS / JULIA_SSH_NO_VERIFY_HOSTS / JULIA_ALWAYS_VERIFY_HOSTS

Specify hosts whose identity should or should not be verified for specific transport layers. See `NetworkOptions.verify_host`

JULIA_SSL_CA_ROOTS_PATH

Specify the file or directory containing the certificate authority roots. See `NetworkOptions.ca_roots`

30.4 External applications

JULIA_SHELL

Julia 用来执行外部命令的 shell 的绝对路径（通过 Base.repl_cmd()）。默认为环境变量 $SHELL，如果 $SHELL 未设置，则为 /bin/sh。

> **Note**
>
> 在 Windows 上，此环境变量将被忽略，并且外部命令会直接被执行。

JULIA_EDITOR

InteractiveUtils.editor() 的返回值–编辑器，例如，InteractiveUtils.edit，会启动偏好编辑器，比如 vim。

$JULIA_EDITOR 优先于 $VISUAL，而后者优先于 $EDITOR。如果这些环境变量都没有设置，那么在 Windows 和 OS X 上会设置为 open，或者 /etc/alternatives/editor（如果存在的话），否则为 emacs。

To use Visual Studio Code on Windows, set $JULIA_EDITOR to code.cmd.

30.5 并行

JULIA_CPU_THREADS

改写全局变量 Base.Sys.CPU_THREADS，逻辑 CPU 核心数。

JULIA_WORKER_TIMEOUT

一个 Float64 值，用来确定 Distributed.worker_timeout() 的值（默认：60.0）。此函数提供 worker 进程在死亡之前等待 master 进程建立连接的秒数。

JULIA_NUM_THREADS

一个无符号 64 位整数 (uint64_t)，用于设置 Julia 可用的最大线程数。如果 $JULIA_NUM_THREADS 不为正数或未设置，或者无法通过系统调用确定 CPU 线程数，则将线程数设置为 1。

如果 $JULIA_NUM_THREADS 设置为 auto，则线程数将设置为 CPU 线程数。

> **Note**
>
> JULIA_NUM_THREADS 必须在启动 julia 之前定义；启动过程中在 startup.jl 中定义它是不能奏效的。

> **Julia 1.5**
>
> 在 Julia 1.5 和更高版本中，也可在启动时使用 -t/--threads 命令行参数指定线程数。

> **Julia 1.7**
>
> $JULIA_NUM_THREADS 的 auto 值需要 Julia 1.7 或更高版本。

> **Julia 1.7**
>
> The `auto` value for `$JULIA_NUM_THREADS` requires Julia 1.7 or above.

JULIA_THREAD_SLEEP_THRESHOLD

如果被设置为字符串，并且以大小写敏感的子字符串 `"infinite"` 开头，那么自旋线程从不睡眠。否则，`$JULIA_THREAD_SLEEP_THRESHOLD` 被解释为一个无符号 64 位整数（`uint64_t`），并且提供以纳秒为单位的自旋线程睡眠的时间量。

JULIA_NUM_GC_THREADS

Sets the number of threads used by Garbage Collection. If unspecified is set to half of the number of worker threads.

> **Julia 1.10**
>
> The environment variable was added in 1.10

JULIA_IMAGE_THREADS

An unsigned 32-bit integer that sets the number of threads used by image compilation in this Julia process. The value of this variable may be ignored if the module is a small module. If left unspecified, the smaller of the value of `JULIA_CPU_THREADS` or half the number of logical CPU cores is used in its place.

JULIA_IMAGE_TIMINGS

A boolean value that determines if detailed timing information is printed during during image compilation. Defaults to 0.

JULIA_EXCLUSIVE

如果设置为 `0` 以外的任何值，那么 Julia 的线程策略与在专用计算机上一致：主线程在 proc 0 上且线程间是关联的。否则，Julia 让操作系统处理线程策略。

30.6 REPL 格式化输出

决定 REPL 应当如何格式化输出的环境变量。通常，这些变量应当被设置为 ANSI 终端转义序列。Julia 提供了具有相同功能的高级接口；请参阅 Julia REPL 章节。

JULIA_ERROR_COLOR

`Base.error_color()`（默认值：亮红，`"\033[91m"`），errors 在终端中的格式。

JULIA_WARN_COLOR

`Base.warn_color()`（默认值：黄，`"\033[93m"`），warnings 在终端中的格式。

JULIA_INFO_COLOR

`Base.info_color()`（默认值：青，`"\033[36m"`），info 在终端中的格式。

JULIA_INPUT_COLOR

`Base.input_color()`（默认值：标准，`"\033[0m"`），在终端中，输入应有的格式。

JULIA_ANSWER_COLOR

`Base.answer_color()`（默认值：标准，`"\033[0m"`），在终端中，输出应有的格式。

30.7 System and Package Image Building

JULIA_CPU_TARGET

Modify the target machine architecture for (pre)compiling system and package images. `JULIA_CPU_TARGET` only affects machine code image generation being output to a disk cache. Unlike the `--cpu-target`, or `-C`, command line option, it does not influence just-in-time (JIT) code generation within a Julia session where machine code is only stored in memory.

Valid values for `JULIA_CPU_TARGET` can be obtained by executing `julia -C help`.

Setting `JULIA_CPU_TARGET` is important for heterogeneous compute systems where processors of distinct types or features may be present. This is commonly encountered in high performance computing (HPC) clusters since the component nodes may be using distinct processors.

The CPU target string is a list of strings separated by `;` each string starts with a CPU or architecture name and followed by an optional list of features separated by `,`. A `generic` or empty CPU name means the basic required feature set of the target ISA which is at least the architecture the C/C++ runtime is compiled with. Each string is interpreted by LLVM.

A few special features are supported:

1. `clone_all`

   This forces the target to have all functions in sysimg cloned. When used in negative form (i.e. `-clone_all`), this disables full clone that's enabled by default for certain targets.

2. `base([0-9]*)`

   This specifies the (0-based) base target index. The base target is the target that the current target is based on, i.e. the functions that are not being cloned will use the version in the base target. This option causes the base target to be fully cloned (as if `clone_all` is specified for it) if it is not the default target (0). The index can only be smaller than the current index.

3. `opt_size`

   Optimize for size with minimum performance impact. Clang/GCC's `-Os`.

4. `min_size`

   Optimize only for size. Clang's `-Oz`.

30.8 Debugging and profiling

JULIA_DEBUG

Enable debug logging for a file or module, see `Logging` for more information.

JULIA_GC_ALLOC_POOL, JULIA_GC_ALLOC_OTHER, JULIA_GC_ALLOC_PRINT

这些环境变量取值为字符串，可以以字符 'r' 开头，后接一个由三个带符号 64 位整数（int64_t）组成的、以冒号分割的列表的插值字符串。这个整数的三元组 a:b:c 代表算术序列 a, a + b, a + 2*b, ... c。

- 如果是第 n 次调用 jl_gc_pool_alloc()，并且 n 属于 $JULIA_GC_ALLOC_POOL 代表的算术序列，那么垃圾回收是强制的。
- 如果是第 n 次调用 maybe_collect()，并且 n 属于 $JULIA_GC_ALLOC_OTHER 代表的算术序列，那么垃圾回收是强制的。
- 如果是第 n 次调用 jl_gc_alloc()，并且 n 属于 $JULIA_GC_ALLOC_PRINT 代表的算术序列，那么调用 jl_gc_pool_alloc() 和 maybe_collect() 的次数会被打印。

如果这些环境变量的值以字符 'r' 开头，那么垃圾回收事件间的间隔是随机的。

Note

这些环境变量生效要求 Julia 在编译时带有垃圾收集调试支持（也就是，在构建配置中将 WITH_GC_DEBUG_ENV 设置为 1）。

JULIA_GC_NO_GENERATIONAL

如果设置为 0 以外的任何值，那么 Julia 的垃圾收集器将从不执行「快速扫描」内存。

Note

此环境变量生效要求 Julia 在编译时带有垃圾收集调试支持（也就是，在构建配置中将 WITH_GC_DEBUG_ENV 设置为 1）。

JULIA_GC_WAIT_FOR_DEBUGGER

如果设置为 0 以外的任何值，Julia 的垃圾收集器每当出现严重错误时将等待调试器连接而不是中止。

Note

此环境变量生效要求 Julia 在编译时带有垃圾收集调试支持（也就是，在构建配置中将 WITH_GC_DEBUG_ENV 设置为 1）。

ENABLE_JITPROFILING

如果设置为 0 以外的任何值，那么编译器将为即时（JIT）性能分析创建并注册一个事件监听器。

Note

此环境变量仅在使用 JIT 性能分析支持编译 Julia 时有效，使用如下之一：

- Intel's VTune™ Amplifier(USE_INTEL_JITEVENTS 在配置中设置为 1), 或
- OProfile(USE_OPROFILE_JITEVENTS 在配置中设置为 1)。
- Perf (USE_PERF_JITEVENTS 在构建配置中设置为 1)。默认情况下启用此集成。

ENABLE_GDBLISTENER

如果设置为除 `0` 之外的任何内容，则在发布版本上启用 Julia 代码的 GDB 注册。在 Julia 的调试版本中，这始终处于启用状态。推荐与 `-g 2` 一起使用。

JULIA_LLVM_ARGS

要传递给 LLVM 后端的参数。

Chapter 31

嵌入 Julia

As we have seen in Calling C and Fortran Code, Julia has a simple and efficient way to call functions written in C. But there are situations where the opposite is needed: calling Julia functions from C code. This can be used to integrate Julia code into a larger C/C++ project, without the need to rewrite everything in C/C++. Julia has a C API to make this possible. As almost all programming languages have some way to call C functions, the Julia C API can also be used to build further language bridges (e.g. calling Julia from Python, Rust or C#). Even though Rust and C++ can use the C embedding API directly, both have packages helping with it, for C++ Jluna is useful.

31.1 高级别嵌入

Note: 本节包含可运行在类 Unix 系统上的、使用 C 编写的嵌入式 Julia 代码。For doing this on Windows, please see the section following this, High-Level Embedding on Windows with Visual Studio.

我们从一个简单的 C 程序开始初始化 Julia 并调用一些 Julia 代码：

```
#include <julia.h>
JULIA_DEFINE_FAST_TLS // only define this once, in an executable (not in a shared library) if you
↪  want fast code.

int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();

    /* run Julia commands */
    jl_eval_string("print(sqrt(2.0))");

    /* strongly recommended: notify Julia that the
         program is about to terminate. this allows
         Julia time to cleanup pending write requests
         and run all finalizers
    */
    jl_atexit_hook(0);
    return 0;
}
```

为构建这个程序，你必须将 Julia 头文件的路径放入 include 路径并链接 `libjulia` 。例如 Julia 被安装到 `$JULIA_DIR`，则可以用 `gcc` 来编译上面的测试程序 `test.c`：

```
gcc -o test -fPIC -I$JULIA_DIR/include/julia -L$JULIA_DIR/lib -Wl,-rpath,$JULIA_DIR/lib test.c
↪  -ljulia
```

或者，查看 `test/embedding/` 文件夹中 Julia 源代码树中的 `embedding.c` 程序。文件 `cli/loader_exe.c` 程序是另一个简单的例子，说明如何在链接 `libjulia` 时设置 `jl_options` 选项。

在调用任何其他 Julia C 函数之前第一件必须要做的事是初始化 Julia，通过调用 `jl_init` 尝试自动确定 Julia 的安装位置来实现。如果需要自定义位置或指定要加载的系统映像，请改用 `jl_init_with_image`。

测试程序中的第二个语句通过调用 `jl_eval_string` 来执行 Julia 语句。

在程序结束之前，强烈建议确保 `jl_atexit_hook` 已调用完成。上面的示例程序在 `main` 返回之前进行了调用。

Note

现在，动态链接 `libjulia` 的共享库需要传递选项 `RTLD_GLOBAL` 。比如在 Python 中像这样调用：

```
>>> julia=CDLL('./libjulia.dylib',RTLD_GLOBAL)
>>> julia.jl_init.argtypes = []
>>> julia.jl_init()
250593296
```

Note

如果 Julia 程序需要访问主可执行文件中的符号，那么除了下面描述的由 `julia-config.jl` 生成的标记之外，可能还需要在 Linux 上的编译时添加 `-Wl,--export-dynamic` 链接器标志。编译共享库时则不必要。

使用 julia-config 自动确定构建参数

`julia-config.jl` 创建脚本是为了帮助确定使用嵌入的 Julia 程序所需的构建参数。此脚本使用由其调用的特定 Julia 分发的构建参数和系统配置来导出嵌入程序的必要编译器标志以与该分发交互。此脚本位于 Julia 的 share 目录中。

例子

```
#include <julia.h>

int main(int argc, char *argv[])
{
    jl_init();
    (void)jl_eval_string("println(sqrt(2.0))");
    jl_atexit_hook(0);
    return 0;
}
```

在命令行中

命令行脚本简单用法：假设 `julia-config.jl` 位于 `/usr/local/julia/share/julia`，它可以直接在命令行上调用，并采用 3 个标志的任意组合：

```
/usr/local/julia/share/julia/julia-config.jl
Usage: julia-config [--cflags|--ldflags|--ldlibs]
```

如果上面的示例源代码保存为文件 `embed_example.c`，则以下命令将其编译为 Linux 和 Windows 上运行的程序（MSYS2 环境），或者如果在 macOS 上，则用 `clang` 替换 `gcc`。:

```
/usr/local/julia/share/julia/julia-config.jl --cflags --ldflags --ldlibs | xargs gcc
↪  embed_example.c
```

在 Makefiles 中使用

通常来说，嵌入的项目会比上面更复杂，因此一般会提供 makefile 支持。由于使用了 **shell** 宏扩展，我们就假设用 GNU make 。此外，尽管 `julia-config.jl` 通常位于 `/usr/local` 目录中，但如果不在，则可以使用 Julia 本身来查找 `julia-config.jl`，而 makefile 可以利用这一点。上面的示例已扩展到使用 makefile：

```
JL_SHARE = $(shell julia -e 'print(joinpath(Sys.BINDIR, Base.DATAROOTDIR, "julia"))')
CFLAGS += $(shell $(JL_SHARE)/julia-config.jl --cflags)
CXXFLAGS += $(shell $(JL_SHARE)/julia-config.jl --cflags)
LDFLAGS += $(shell $(JL_SHARE)/julia-config.jl --ldflags)
LDLIBS += $(shell $(JL_SHARE)/julia-config.jl --ldlibs)

all: embed_example
```

现在构建的命令就只需要简简单单的 `make` 了。

31.2 在 Windows 使用 Visual Studio 进行高级别嵌入

如果尚未设置 `JULIA_DIR` 环境变量，请在启动 Visual Studio 之前使用系统面板添加它。JULIA_DIR 下的 `bin` 文件夹应该在系统路径上。

我们首先打开 Visual Studio 并创建一个新的控制台应用程序项目。在 `stdafx.h` 头文件的末尾添加以下几行：

```
#include <julia.h>
```

然后，将项目中的 main() 函数替换为以下代码：

```
int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();

    /* run Julia commands */
    jl_eval_string("print(sqrt(2.0))");

    /* strongly recommended: notify Julia that the
         program is about to terminate. this allows
         Julia time to cleanup pending write requests
         and run all finalizers
```

```
    */
    jl_atexit_hook(0);
    return 0;
}
```

下一步是设置项目以查找 Julia 包含的文件和库。了解 Julia 安装的是 32 位还是 64 位非常重要。在继续之前删除与 Julia 安装不对应的任何平台配置。

使用项目属性对话框，转到 C/C++ | General 并将 $(JULIA_DIR)\include\julia\ 添加到 Additional Include Directories 属性。然后，转到 Linker | General 部分并将 $(JULIA_DIR)\lib 添加到 Additional Library Directories 属性。最后，在 Linker| Input 下，将 libjulia.dll.a;libopenlibm.dll.a; 添加到库列表中。

到这里，该项目应该成功构建和运行。

31.3 转换类型

真正的应用程序不仅仅要执行表达式，还要返回表达式的值给宿主程序。jl_eval_string 返回一个 jl_value_t*，它是指向堆分配的 Julia 对象的指针。存储像 Float64 这些简单数据类型叫做 装箱，然后提取存储的基础类型数据叫 拆箱。我们改进的示例程序在 Julia 中计算 2 的平方根，并在 C 中读取回结果，如下所示：

```
jl_value_t *ret = jl_eval_string("sqrt(2.0)");

if (jl_typeis(ret, jl_float64_type)) {
    double ret_unboxed = jl_unbox_float64(ret);
    printf("sqrt(2.0) in C: %e \n", ret_unboxed);
}
else {
    printf("ERROR: unexpected return type from sqrt(::Float64)\n");
}
```

为了检查 ret 是否为特定的 Julia 类型，我们可以使用 jl_isa，jl_typeis 或 jl_is_... 函数。通过输入 typeof(sqrt(2.0)) 到 Julia shell，我们可以看到返回类型是 Float64（在 C 中是 double 类型）。要将装箱的 Julia 值转换为 C 的 double，上面的代码片段使用了 jl_unbox_float64 函数。

相应的, 用 jl_box_... 函数是另一种转换的方式。

```
jl_value_t *a = jl_box_float64(3.0);
jl_value_t *b = jl_box_float32(3.0f);
jl_value_t *c = jl_box_int32(3);
```

正如我们将在下面看到的那样，装箱需要在调用 Julia 函数时使用特定参数。

31.4 调用 Julia 函数

虽然 jl_eval_string 允许 C 获取 Julia 表达式的结果，但它不允许将在 C 中计算的参数传递给 Julia。因此需要使用 jl_call 来直接调用 Julia 函数：

```
jl_function_t *func = jl_get_function(jl_base_module, "sqrt");
jl_value_t *argument = jl_box_float64(2.0);
jl_value_t *ret = jl_call1(func, argument);
```

在第一步中，通过调用 jl_get_function 检索出 Julia 函数 sqrt 的句柄(handle)。传递给 jl_get_function 的第一个参数是指向定义 sqrt 所在的 Base 模块的指针。然后，double 值通过 jl_box_float64 被装箱。最后，使用 jl_call1 调用该函数。也有 jl_call0，jl_call2 和 jl_call3 函数，方便地处理不同数量的参数。要传递更多参数，使用 jl_call：

```
jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t nargs)
```

它的第二个参数 args 是 jl_value_t* 类型的数组，nargs 是参数的个数

There is also an alternative, possibly simpler, way of calling Julia functions and that is via @cfunction. Using @cfunction allows you to do the type conversions on the Julia side which typically is easier than doing it on the C side. The sqrt example above would with @cfunction be written as:

```
double (*sqrt_jl)(double) = jl_unbox_voidpointer(jl_eval_string("@cfunction(sqrt, Float64,
↪  (Float64,))"));
double ret = sqrt_jl(2.0);
```

where we first define a C callable function in Julia, extract the function pointer from it and finally call it.

31.5 内存管理

正如我们所见，Julia 对象在 C 中表示为类型 jl_value_t* 的指针。这就出现了谁来负责释放这些对象的问题。

通常，Julia 对象由垃圾收集器（GC）释放，但 GC 不会自动就懂我们正 C 中保留对 Julia 值的引用。这意味着 GC 会在你的掌控之外释放对象，从而使指针无效。

GC 会在分配 Julia 对象时运行。像 jl_box_float64 这样的调用执行分配，分配可能发生在运行 Julia 代码的任何时候。

When writing code that embeds Julia, it is generally safe to use jl_value_t* values in between jl_... calls (as GC will only get triggered by those calls). 但是为了确保值可以在 jl_... 调用后留存下来，我们必须告诉 Julia 我们仍然持有对 Julia root 的引用，这个过程称为“GC rooting”。把一个值”扎根“将确保垃圾收集器不会意外地将此值识别为未使用并释放该值的内存。这可以使用 JL_GC_PUSH 宏来完成：

```
jl_value_t *ret = jl_eval_string("sqrt(2.0)");
JL_GC_PUSH1(&ret);
// Do something with ret
JL_GC_POP();
```

JL_GC_POP 调用会释放之前的 JL_GC_PUSH 建立的引用。请注意，JL_GC_PUSH 将引用存储在 C 堆栈上，因此在退出作用域之前，它必须与一个 JL_GC_POP 精确配对。也就是说，在函数返回之前，或者流程控制以其他方式离开调用了 JL_GC_PUSH 的块。

可以使用 JL_GC_PUSH2 到 JL_GC_PUSH6 宏一次推送多个 Julia 值：

```
JL_GC_PUSH2(&ret1, &ret2);
// ...
JL_GC_PUSH6(&ret1, &ret2, &ret3, &ret4, &ret5, &ret6);
```

要推送一个 Julia 数组，可以使用 JL_GC_PUSHARGS 宏，其用法如下：

```
jl_value_t **args;
JL_GC_PUSHARGS(args, 2); // args can now hold 2 `jl_value_t*` objects
args[0] = some_value;
args[1] = some_other_value;
// Do something with args (e.g. call jl_... functions)
JL_GC_POP();
```

每个作用域只能调用一次 JL_GC_PUSH*，并且只能与一次 JL_GC_POP 调用配对。如果一次调用 JL_GC_PUSH* 无法推送所有需要 root 的变量，或者需要推送的变量超过 6 个，并且不能使用参数数组，那么可以使用内部代码块：

```
jl_value_t *ret1 = jl_eval_string("sqrt(2.0)");
JL_GC_PUSH1(&ret1);
jl_value_t *ret2 = 0;
{
    jl_function_t *func = jl_get_function(jl_base_module, "exp");
    ret2 = jl_call1(func, ret1);
    JL_GC_PUSH1(&ret2);
    // Do something with ret2.
    JL_GC_POP();    // This pops ret2.
}
JL_GC_POP();    // This pops ret1.
```

Note that it is not necessary to have valid jl_value_t* values before calling JL_GC_PUSH*. It is fine to have a number of them initialized to NULL, pass those to JL_GC_PUSH* and then create the actual Julia values. For example:

```
jl_value_t *ret1 = NULL, *ret2 = NULL;
JL_GC_PUSH2(&ret1, &ret2);
ret1 = jl_eval_string("sqrt(2.0)");
ret2 = jl_eval_string("sqrt(3.0)");
// Use ret1 and ret2
JL_GC_POP();
```

If it is required to hold the pointer to a variable between functions (or block scopes), then it is not possible to use JL_GC_PUSH*. In this case, it is necessary to create and keep a reference to the variable in the Julia global scope. One simple way to accomplish this is to use a global IdDict that will hold the references, avoiding deallocation by the GC. However, this method will only work properly with mutable types.

```
// This functions shall be executed only once, during the initialization.
jl_value_t* refs = jl_eval_string("refs = IdDict()");
jl_function_t* setindex = jl_get_function(jl_base_module, "setindex!");

...
```

```
// `var` is the variable we want to protect between function calls.
jl_value_t* var = 0;

...

// `var` is a `Vector{Float64}`, which is mutable.
var = jl_eval_string("[sqrt(2.0); sqrt(4.0); sqrt(6.0)]");

// To protect `var`, add its reference to `refs`.
jl_call3(setindex, refs, var, var);
```

如果变量是不可变的，则需要将其包装在等效的可变容器中，或者最好在将其推送到 `IdDict` 之前包装在 `RefValue{Any}` 中。在这种方法中，容器必须通过 C 代码创建或填充，例如使用函数 `jl_new_struct`。如果容器是由 `jl_call*` 创建的，那么你将需要重新加载要在 C 代码中使用的指针。

```
// This functions shall be executed only once, during the initialization.
jl_value_t* refs = jl_eval_string("refs = IdDict()");
jl_function_t* setindex = jl_get_function(jl_base_module, "setindex!");
jl_datatype_t* reft = (jl_datatype_t*)jl_eval_string("Base.RefValue{Any}");

...

// `var` is the variable we want to protect between function calls.
jl_value_t* var = 0;

...

// `var` is a `Float64`, which is immutable.
var = jl_eval_string("sqrt(2.0)");

// Protect `var` until we add its reference to `refs`.
JL_GC_PUSH1(&var);

// Wrap `var` in `RefValue{Any}` and push to `refs` to protect it.
jl_value_t* rvar = jl_new_struct(reft, var);
JL_GC_POP();

jl_call3(setindex, refs, rvar, rvar);
```

GC 可以通过使用函数 `delete!` 从 `refs` 中删除对变量的引用来释放变量，前提是没有其它对该变量的引用保留在任何地方：

```
jl_function_t* delete = jl_get_function(jl_base_module, "delete!");
jl_call2(delete, refs, rvar);
```

作为非常简单情况的替代方案，可以只创建一个类型为 `Vector{Any}` 的全局容器，并在必要时从中获取元素，甚至可以使用以下方法为每个指针创建一个全局变量

```
jl_module_t *mod = jl_main_module;
jl_sym_t *var = jl_symbol("var");
```

```
jl_binding_t *bp = jl_get_binding_wr(mod, var);
jl_checked_assignment(bp, mod, var, val);
```

更新 GC 管理对象的字段

垃圾回收器的运行假设它知道每个年老代对象都指向一个年轻代对象。任何时候一个指针被更新打破了这个假设，它必须用 `jl_gc_wb`（写屏障）函数向回收器发出信号，如下所示：

```
jl_value_t *parent = some_old_value, *child = some_young_value;
((some_specific_type*)parent)->field = child;
jl_gc_wb(parent, child);
```

通常情况下不可能在运行时预测值是否是旧的，因此写屏障必须被插入在所有显式存储之后。一个需要注意的例外是如果 `parent` 对象刚分配，垃圾收集之后并不执行。请记住大多数 `jl_...` 函数有时候都会执行垃圾收集。

直接更新数据时，对于指针数组来说写屏障也是必需的例如：

```
jl_array_t *some_array = ...; // e.g. a Vector{Any}
void **data = (void**)jl_array_data(some_array);
jl_value_t *some_value = ...;
data[0] = some_value;
jl_gc_wb(some_array, some_value);
```

控制垃圾收集器

有一些函数能够控制 GC。在正常使用情况下这些不是必要的。

函数	描述
`jl_gc_collect()`	强制执行 GC
`jl_gc_enable(0)`	禁用 GC，返回前一个状态作为 int 类型
`jl_gc_enable(1)`	启用 GC，返回前一个状态作为 int 类型
`jl_gc_is_enabled()`	返回当前状态作为 int 类型

31.6 使用数组

Julia 和 C 可以不通过复制而共享数组数据。下面一个例子将展示它是如何工作的。

Julia 数组用数据类型 `jl_array_t *` 表示。基本上，`jl_array_t` 是一个包含以下内容的结构：

- 关于数据类型的信息
- 指向数据块的指针
- 关于数组长度的信息

为了让事情比较简单，我们从一维数组开始，创建一个存有 10 个 FLoat64 类型的数组如下所示：

```
jl_value_t* array_type = jl_apply_array_type((jl_value_t*)jl_float64_type, 1);
jl_array_t* x          = jl_alloc_array_1d(array_type, 10);
```

或者，如果您已经分配了数组，则可以生成一个简易的包装器来包裹其数据：

```
double *existingArray = (double*)malloc(sizeof(double)*10);
jl_array_t *x = jl_ptr_to_array_1d(array_type, existingArray, 10, 0);
```

最后一个参数是一个布尔值，表示 Julia 是否应该获取数据的所有权。如果这个参数不为零，当数组不再被引用时，GC 会在数据的指针上调用 `free` 。

为了访问 x 的数据，我们可以使用 `jl_array_data`：

```
double *xData = (double*)jl_array_data(x);
```

现在我们可以填充这个数组：

```
for(size_t i=0; i<jl_array_len(x); i++)
    xData[i] = i;
```

现在让我们调用一个对 x 就地操作的 Julia 函数：

```
jl_function_t *func = jl_get_function(jl_base_module, "reverse!");
jl_call1(func, (jl_value_t*)x);
```

通过打印数组，可以验证 x 的元素现在是否已被逆置 (reversed)。

获取返回的数组

如果 Julia 函数返回一个数组，`jl_eval_string` 和 `jl_call` 的返回值可以被强制转换为 `jl_array_t *`：

```
jl_function_t *func  = jl_get_function(jl_base_module, "reverse");
jl_array_t *y = (jl_array_t*)jl_call1(func, (jl_value_t*)x);
```

现在使用 `jl_array_data` 可以像前面一样访问 y 的内容。一如既往地，一定要在使用数组的时候确保持有使用数组的引用。

多维数组

Julia 的多维数组以列序优先存储在内存中。这是一些创建一个 2D 数组并访问其属性的代码：

```
// Create 2D array of float64 type
jl_value_t *array_type = jl_apply_array_type((jl_value_t*)jl_float64_type, 2);
jl_array_t *x  = jl_alloc_array_2d(array_type, 10, 5);

// Get array pointer
double *p = (double*)jl_array_data(x);
// Get number of dimensions
```

```
int ndims = jl_array_ndims(x);
// Get the size of the i-th dim
size_t size0 = jl_array_dim(x,0);
size_t size1 = jl_array_dim(x,1);

// Fill array with data
for(size_t i=0; i<size1; i++)
    for(size_t j=0; j<size0; j++)
        p[j + size0*i] = i + j;
```

请注意，虽然 Julia 的数组使用基于 1 的索引，但 C API 中使用基于 0 的索引（例如在调用 `jl_array_dim`）以便用 C 代码的习惯来阅读。

31.7 异常

Julia 代码可以抛出异常。比如：

```
jl_eval_string("this_function_does_not_exist()");
```

这个调用似乎什么都没做。但可以检查异常是否抛出：

```
if (jl_exception_occurred())
    printf("%s \n", jl_typeof_str(jl_exception_occurred()));
```

如果您使用支持异常的语言的 Julia C API（例如 Python，C ＃，C ++），使用检查是否有异常的函数将每个调用包装到 `libjulia` 中是有意义的，然后异常在宿主语言中重新抛出。

抛出 Julia 异常

在编写 Julia 可调用函数时，可能需要验证参数并抛出异常表示错误。典型的类型检查像这样：

```
if (!jl_typeis(val, jl_float64_type)) {
    jl_type_error(function_name, (jl_value_t*)jl_float64_type, val);
}
```

可以使用以下函数引发一般异常：

```
void jl_error(const char *str);
void jl_errorf(const char *fmt, ...);
```

`jl_error` 采用 C 字符串，而 `jl_errorf` 像 `printf` 一样调用:

```
jl_errorf("argument x = %d is too large", x);
```

where in this example x is assumed to be an integer.

Thread-safety

In general, the Julia C API is not fully thread-safe. When embedding Julia in a multi-threaded application care needs to be taken not to violate the following restrictions:

- `jl_init()` may only be called once in the application life-time. The same applies to `jl_atexit_hook()`, and it may only be called after `jl_init()`.
- `jl_...()` API functions may only be called from the thread in which `jl_init()` was called, *or from threads started by the Julia runtime*. Calling Julia API functions from user-started threads is not supported, and may lead to undefined behaviour and crashes.

The second condition above implies that you can not safely call `jl_...()` functions from threads that were not started by Julia (the thread calling `jl_init()` being the exception). For example, the following is not supported and will most likely segfault:

```
void *func(void*)
{
    // Wrong, jl_eval_string() called from thread that was not started by Julia
    jl_eval_string("println(Threads.threadid())");
    return NULL;
}

int main()
{
    pthread_t t;

    jl_init();

    // Start a new thread
    pthread_create(&t, NULL, func, NULL);
    pthread_join(t, NULL);

    jl_atexit_hook(0);
}
```

Instead, performing all Julia calls from the same user-created thread will work:

```
void *func(void*)
{
    // Okay, all jl_...() calls from the same thread,
    // even though it is not the main application thread
    jl_init();
    jl_eval_string("println(Threads.threadid())");
    jl_atexit_hook(0);
    return NULL;
}

int main()
{
    pthread_t t;
    // Create a new thread, which runs func()
    pthread_create(&t, NULL, func, NULL);
```

```
    pthread_join(t, NULL);
}
```

An example of calling the Julia C API from a thread started by Julia itself:

```
#include <julia/julia.h>
JULIA_DEFINE_FAST_TLS

double c_func(int i)
{
    printf("[C %08x] i = %d\n", pthread_self(), i);

    // Call the Julia sqrt() function to compute the square root of i, and return it
    jl_function_t *sqrt = jl_get_function(jl_base_module, "sqrt");
    jl_value_t* arg = jl_box_int32(i);
    double ret = jl_unbox_float64(jl_call1(sqrt, arg));

    return ret;
}

int main()
{
    jl_init();

    // Define a Julia function func() that calls our c_func() defined in C above
    jl_eval_string("func(i) = ccall(:c_func, Float64, (Int32,), i)");

    // Call func() multiple times, using multiple threads to do so
    jl_eval_string("println(Threads.threadpoolsize())");
    jl_eval_string("use(i) = println(\"[J $(Threads.threadid())] i = $(i) -> $(func(i))\")");
    jl_eval_string("Threads.@threads for i in 1:5 use(i) end");

    jl_atexit_hook(0);
}
```

If we run this code with 2 Julia threads we get the following output (note: the output will vary per run and system):

```
$ JULIA_NUM_THREADS=2 ./thread_example
2
[C 3bfd9c00] i = 1
[C 23938640] i = 4
[J 1] i = 1 -> 1.0
[C 3bfd9c00] i = 2
[J 1] i = 2 -> 1.4142135623730951
[C 3bfd9c00] i = 3
[J 2] i = 4 -> 2.0
[C 23938640] i = 5
[J 1] i = 3 -> 1.7320508075688772
[J 2] i = 5 -> 2.23606797749979
```

As can be seen, Julia thread 1 corresponds to pthread ID 3bfd9c00, and Julia thread 2 corresponds to ID

23938640, showing that indeed multiple threads are used at the C level, and that we can safely call Julia C API routines from those threads.

Chapter 32

代码加载

> **Note**
>
> 这一章包含了加载包的技术细节。如果要安装包，使用 Julia 的内置包管理器Pkg将包加入到你的活跃环境中。如果要使用已经在你的活跃环境中的包，使用 `import X` 或 `using X`，正如在模块中所描述的那样。

32.1 定义

Julia 加载代码有两种机制：

1. **代码包含**：例如 `include("source.jl")`。包含允许你把一个程序拆分为多个源文件。表达式 `include("source.jl")` 使得文件 `source.jl` 的内容在出现 `include` 调用的模块的全局作用域中执行。如果多次调用 `include("source.jl")`，`source.jl` 就被执行多次。`source.jl` 的包含路径解释为相对于出现 `include` 调用的文件路径。重定位源文件子树因此变得简单。在 REPL 中，包含路径为当前工作目录，即 `pwd()`。

2. **加载包**：例如 `import X` 或 `using X`。`import` 通过加载包（一个独立的，可重用的 Julia 代码集合，包含在一个模块中），并导入模块内部的名称 `X`，使得模块 `X` 可用。如果在同一个 Julia 会话中，多次导入包 `X`，那么后续导入模块为第一次导入模块的引用。但请注意，`import X` 可以在不同的上下文中加载不同的包：`X` 可以引用主工程中名为 `X` 的一个包，但它在各个依赖中可以引用不同的、名称同为 `X` 的包。更多机制说明如下。

代码包含是非常直接和简单的：其在调用者的上下文中解释运行给定的源文件。包加载是建立在代码包含之上的，它具有不同的用途。本章的其余部分将重点介绍程序包加载的行为和机制。

一个 包（*package*）就是一个源码树，其标准布局中提供了其他 Julia 项目可以复用的功能。包可以使用 `import X` 或 `using X` 语句加载，名为 `X` 的模块在加载包代码时生成，并在包含该 import 语句的模块中可用。`import X` 中 `X` 的含义与上下文有关：程序加载哪个 `X` 包取决于 import 语句出现的位置。因此，处理 `import X` 分为两步：首先，确定在此上下文中是**哪个**包被定义为 `X`；其次，确定到**哪里**找特定的 `X` 包。

这些问题可通过查询各项目文件（`Project.toml` 或 `JuliaProject.toml`）、清单文件（`Manifest.toml` 或 `JuliaManifest.toml`），或是源文件的文件夹列在`LOAD_PATH` 中的项目环境解决。

32.2 包的联合生态

大多数时候，一个包可以通过它的名字唯一确定。但有时在一个项目中，可能需要使用两个有着相同名字的不同的包。尽管你可以通过重命名其中一个包来解决这个问题，但在一个大型的、共享的代码库中被迫做这件事可能是有高度破坏性的。相反，Julia 的包加载机制允许相同的包名在一个应用的不同部分指向不同的包。

Julia 支持联合的包管理，这意味着多个独立的部分可以维护公有包、私有包以及包的注册表，并且项目可以依赖于一系列来自不同注册表的公有包和私有包。您也可以使用一组通用工具和工作流（workflow）来安装和管理来自各种注册表的包。Julia 附带的 Pkg 软件包管理器允许安装和管理项目的依赖项，它会帮助创建并操作项目文件（其描述了项目所依赖的其他项目）和清单文件（其为项目完整依赖库的确切版本的快照）。

联合管理的一个可能后果是没有包命名的中央权限。不同组织可以使用相同的名称来引用不相关的包。这并不是没有可能的，因为这些组织可能没有协作，甚至不知道彼此。由于缺乏中央命名权限，单个项目可能最终依赖着具有相同名称的不同包。Julia 的包加载机制不要求包名称是全局唯一的，即使在单个项目的依赖关系图中也是如此。相反，包由通用唯一标识符（UUID）进行标识，它在每个包创建时进行分配。通常，您不必直接使用这些有点麻烦的 128 位标识符，因为 Pkg 将负责生成和跟踪它们。但是，这些 UUID 为问题「*X* 所指的包是什么？」提供了确定的答案

由于去中心化的命名问题有些抽象，因此可以通过具体情境来理解问题。假设你正在开发一个名为 `App` 的应用程序，它使用两个包：`Pub` 和 `Priv`。`Priv` 是你创建的私有包，而 `Pub` 是你使用但不控制的公共包。当你创建 `Priv` 时，没有名为 `Priv` 的公共包。然而，随后一个名为 `Priv` 的不相关软件包发布并变得流行起来，而且 `Pub` 包已经开始使用它了。因此，当你下次升级 `Pub` 以获取最新的错误修复和特性时，`App` 将依赖于两个名为 `Priv` 的不同包——尽管你除了升级之外什么都没做。`App` 直接依赖于你的私有 `Priv` 包，以及通过 `Pub` 在新的公共 `Priv` 包上的间接依赖。由于这两个 `Priv` 包是不同的，但是 `App` 继续正常工作依赖于他们两者，因此表达式 `import Priv` 必须引用不同的 `Priv` 包，具体取决于它是出现在 `App` 的代码中还是出现在 `Pub` 的代码中。为了处理这种情况，Julia 的包加载机制通过 UUID 区分两个 `Priv` 包并根据它（调用 `import` 的模块）的上下文选择正确的包。这种区分的工作原理取决于环境，如以下各节所述。

32.3 环境（Environments）

环境决定了 `import X` 和 `using X` 语句在不同的代码上下文中的含义以及什么文件会被加载。Julia 有两类环境（environment）：

1. **项目环境（project environment）**是包含项目文件和清单文件（可选）的目录，并形成一个显式环境。项目文件确定项目的直接依赖项的名称和标识。清单文件（如果存在）提供完整的依赖关系图，包括所有直接和间接依赖关系，每个依赖的确切版本以及定位和加载正确版本的足够信息。
2. **包目录（package directory）**是包含一组包的源码树子目录的目录，并形成一个隐式环境。如果 `X` 是包目录的子目录并且存在 `X/src/X.jl`，那么程序包 `X` 在包目录环境中可用，而 `X/src/X.jl` 是加载它使用的源文件。

这些环境可以混合并用来创建**堆栈环境（stacked environment）**：是一组有序的项目环境和包目录，重叠为一个复合环境。然后，结合优先级规则和可见性规则，确定哪些包是可用的以及从哪里加载它们。例如，Julia 的负载路径是一个堆栈环境。

这些环境各有不同的用途：

- 项目环境提供**可迁移性**。通过将项目环境以及项目源代码的其余部分存放到版本控制（例如一个 git 存储库），您可以重现项目的确切状态和所有依赖项。特别是，清单文件会记录每个依赖

项的确切版本，而依赖项由其源码树的加密哈希值标识；这使得 Pkg 可以检索出正确的版本，并确保你正在运行准确的已记录的所有依赖项的代码。

- 当不需要完全仔细跟踪的项目环境时，包目录更**方便**。当你想要把一组包放在某处，并且希望能够直接使用它们而不必为之创建项目环境时，包目录是很实用的。
- 堆栈环境允许向基本环境**添加**工具。您可以将包含开发工具在内的环境堆到堆栈环境的末尾，使它们在 REPL 和脚本中可用，但在包内部不可用。

从更高层次上，每个环境在概念上定义了三个映射：roots、graph 和 paths。当解析 `import X` 的含义时，roots 和 graph 映射用于确定 X 的身份，同时 paths 映射用于定位 X 的源代码。这三个映射的具体作用是：

- **roots:** `name::Symbol` ⟶ `uuid::UUID`

  环境的 roots 映射将包名称分配给 UUID，以获取环境可用于主项目的所有顶级依赖项（即可以在 `Main` 中加载的那些依赖项）。当 Julia 在主项目中遇到 `import X` 时，它会将 X 的标识作为 `roots[:X]`。

- **graph:** `context::UUID` ⟶ `name::Symbol` ⟶ `uuid::UUID`

  环境的 graph 是一个多级映射，它为每个 `context` UUID 分配一个从名称到 UUID 的映射——类似于 roots 映射，但专一于那个 `context`。当 Julia 在 UUID 为 `context` 的包代码中运行到 `import X` 时，它会将 X 的标识看作为 `graph[context][:X]`。正是因为如此，`import X` 可以根据 `context` 引用不同的包。

- **paths:** `uuid::UUID` × `name::Symbol` ⟶ `path::String`

  paths 映射会为每个包分配 UUID-name 对，即该包的入口点源文件的位置。在 `import X` 中，X 的标识已经通过 roots 或 graph 解析为 UUID（取决于它是从主项目还是从依赖项加载），Julia 确定要加载哪个文件来获取 X 是通过在环境中查找 `paths[uuid,:X]`。要包含此文件应该定义一个名为 X 的模块。一旦加载了此包，任何解析为相同的 `uuid` 的后续导入只会创建一个到同一个已加载的包模块的绑定。

每种环境都以不同的方式定义这三种映射，详见以下各节。

> **Note**
>
> 为了清楚地说明，本章中的示例包括 roots、graph 和 paths 的完整数据结构。但是，为了提高效率，Julia 的包加载代码并没有显式地创建它们。相反，加载一个给定包只会简单地计算所需的结构。

项目环境（Project environments）

项目环境由包含名为 `Project.toml` 的项目文件的目录以及名为 `Manifest.toml` 的清单文件（可选）确定。这些文件也可以命名为 `JuliaProject.toml` 和 `JuliaManifest.toml`，此时 `Project.toml` 和 `Manifest.toml` 被忽略——这允许项目与可能需要名为 `Project.toml` 和 `Manifest.toml` 文件的其他重要工具共存。但是对于纯 Julia 项目，名称 `Project.toml` 和 `Manifest.toml` 是首选。

项目环境的 roots、graph 和 paths 映射定义如下：

roots 映射在环境中由其项目文件的内容决定，特别是它的顶级 `name` 和 `uuid` 条目及其 `[deps]` 部分（全部是可选的）。考虑以下一个假想的应用程序 App 的示例项目文件，如先前所述：

```
name = "App"
uuid = "8f986787-14fe-4607-ba5d-fbff2944afa9"

[deps]
Priv = "ba13f791-ae1d-465a-978b-69c3ad90f72b"
Pub  = "c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"
```

如果将它表示为 Julia 字典，那么这个项目文件意味着以下 roots 映射：

```
roots = Dict(
    :App  => UUID("8f986787-14fe-4607-ba5d-fbff2944afa9"),
    :Priv => UUID("ba13f791-ae1d-465a-978b-69c3ad90f72b"),
    :Pub  => UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"),
)
```

基于这个 root 映射，在 App 的代码中，语句 `import Priv` 将使 Julia 查找 `roots[:Priv]`，这将得到 `ba13f791-ae1d-465a-978b-69c3ad90f72b`，也就是要在这一部分加载的 `Priv` 包的 UUID。当主应用程序解释运行到 `import Priv` 时，此 UUID 标识了要加载和使用的 `Priv` 包。

依赖图（dependency graph）在项目环境中其清单文件的内容决定，如果其存在。如果没有清单文件，则 graph 为空。清单文件包含项目的直接或间接依赖项的节（stanza）。对于每个依赖项，该文件列出该包的 UUID 以及源码树的哈希值或源代码的显式路径。考虑以下 App 的示例清单文件：

```
[[Priv]] # 私有的那个
deps = ["Pub", "Zebra"]
uuid = "ba13f791-ae1d-465a-978b-69c3ad90f72b"
path = "deps/Priv"

[[Priv]] # 公共的那个
uuid = "2d15fe94-a1f7-436c-a4d8-07a9a496e01c"
git-tree-sha1 = "1bf63d3be994fe83456a03b874b409cfd59a6373"
version = "0.1.5"

[[Pub]]
uuid = "c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"
git-tree-sha1 = "9ebd50e2b0dd1e110e842df3b433cb5869b0dd38"
version = "2.1.4"

  [Pub.deps]
  Priv = "2d15fe94-a1f7-436c-a4d8-07a9a496e01c"
  Zebra = "f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"

[[Zebra]]
uuid = "f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"
git-tree-sha1 = "e808e36a5d7173974b90a15a353b564f3494092f"
version = "3.4.2"
```

这个清单文件描述了 `App` 项目可能的完整依赖关系图：

- 应用程序使用两个名为 `Priv` 的不同包，一个作为根依赖项的私有包，以及一个通过 `Pub` 作为间接依赖项的公共包。它们通过不同 UUID 来区分，并且有不同的依赖项：

- 私有的 Priv 依赖于 Pub 和 Zebra 包。
- 公有的 Priv 没有依赖关系。

• 该应用程序还依赖于 Pub 包，而后者依赖于公有的 Priv 以及私有的 Priv 包所依赖的那个 Zebra 包。

此依赖图以字典表示后如下所示：

```
graph = Dict(
    # Priv——私有的那个:
    UUID("ba13f791-ae1d-465a-978b-69c3ad90f72b") => Dict(
        :Pub   => UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"),
        :Zebra => UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"),
    ),
    # Priv——公共的那个:
    UUID("2d15fe94-a1f7-436c-a4d8-07a9a496e01c") => Dict(),
    # Pub:
    UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1") => Dict(
        :Priv  => UUID("2d15fe94-a1f7-436c-a4d8-07a9a496e01c"),
        :Zebra => UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"),
    ),
    # Zebra:
    UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62") => Dict(),
)
```

给定这个依赖图，当 Julia 看到 Pub 包中的 import Priv——它有 UUIDc07ecb7d-0dc9-4db7-8803-fadaaeaf08e1 时，它会查找：

```
graph[UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1")][:Priv]
```

会得到 2d15fe94-a1f7-436c-a4d8-07a9a496e01c，这意味着 Pub 包中的内容，import Priv 指代的是公有的 Priv 内容，而非应用程序直接依赖的私有包。这也是为何 Priv 在主项目中可指代不同的包，而不像其在某个依赖包中另有含义。在包生态中，该特性允许重名的出现。

如果在 App 主代码库中 import Zebra 会如何？因为 Zebra 不存在于项目文件，即使它 确实存在于清单文件中，其导入会是失败的。此外，import Zebra 这个行为若发生在公有的 Priv 包——UUID 为 2d15fe94-a1f7-436c-a4d8-07a9a496e01c 的包中，同样会失败。因为公有的 Priv 包未在清单文件中声明依赖，故而无法加载包。仅有在清单文件：Pub 包和一个 Priv 包中作为显式依赖的包可用于加载 Zebra。

项目环境的 **路径映射**从 manifest 文件中提取得到。而包的路径 uuid 和名称 X 则 (循序) 依据这些规则确定。

1. 如果目录中的项目文件与要求的 uuid 以及名称 X 匹配，那么可能出现以下情况的一种：
   - 若该文件具有顶层 路径入口，则 uuid 会被映射到该路径，文件的执行与包含项目文件的目录相关。
   - 此外，uuid 依照包含项目文件的目录，映射至与 src/X.jl。
2. 若非上述情况，且项目文件具有对应的清单文件，且该清单文件包含匹配 uuid 的节（stanza），那么：

- 若其具有一个 路径入口，则使用该路径（与包含清单文件的目录相关）。
- 若其具有一个 git-tree-sha1 入口，计算一个确定的 uuid 与 git-tree-sha1 函数——我们把这个函数称为 slug——并在每个 Julia DEPOT_PATH 的全局序列中的目录查询名为 packages/X/$slug 的目录。使用存在的第一个此类目录。

若某些结果成功，源码入口点的路径会是这些结果中的某个，结果的相对路径 +src/X.jl；否则，uuid 不存在路径映射。当加载 X 时，如果没找到源码路径，查找即告失败，用户可能会被提示安装适当的包版本或采取其他纠正措施（例如，将 X 声明为某种依赖性）。

在上述样例清单文件中，为找到首个 Priv 包的路径——该包 UUID 为 ba13f791-ae1d-465a-978b-69c3ad90f72b——Julia 寻找其在清单中的节（stanza）。发现其有路径入口，查看 App 项目目录中相关的 deps/Priv——不妨设 App 代码在/home/me/projects/App 中—则 Julia 发现/home/me/projects/App/deps/Priv 存在，并因此从中加载 Priv‘。

另一方面，如果 Julia 加载的是带有 *other* Priv 包——即 UUID 为 2d15fe94-a1f7-436c-a4d8-07a9a496e01c——它在清单中找到了它的节，请注意它没有 path 条目，但是它有一个 git-tree-sha1 条目。然后计算这个 slug 的 UUID/SHA-1 对，具体是 HDkrT（这个计算的确切细节并不重要，但它是始终一致的和确定的）。这意味着这个 Priv 包的路径 packages/Priv/HDkrT/src/Priv.jl 将在其中一个包仓库中。假设 DEPOT_PATH 的内容是 ["/home/me/.julia", "/usr/local/julia"]，Julia 将根据下面的路径来查看它们是否存在：

1. /home/me/.julia/packages/Priv/HDkrT
2. /usr/local/julia/packages/Priv/HDkrT

Julia 使用以上路径信息在仓库里依次查找 packages/Priv/HDKrT/src/Priv.jl 文件，并从第一个查找到的文件中加载公共的 Priv 包。

这是我们的示例 App 项目环境的可能路径映射的表示，如上面 Manifest 中所提供的依赖关系图，在搜索本地文件系统后：

```
paths = Dict(
    # Priv – the private one:
    (UUID("ba13f791-ae1d-465a-978b-69c3ad90f72b"), :Priv) =>
        # relative entry-point inside `App` repo:
        "/home/me/projects/App/deps/Priv/src/Priv.jl",
    # Priv – the public one:
    (UUID("2d15fe94-a1f7-436c-a4d8-07a9a496e01c"), :Priv) =>
        # package installed in the system depot:
        "/usr/local/julia/packages/Priv/HDkr/src/Priv.jl",
    # Pub:
    (UUID("c07ecb7d-0dc9-4db7-8803-fadaaeaf08e1"), :Pub) =>
        # package installed in the user depot:
        "/home/me/.julia/packages/Pub/oKpw/src/Pub.jl",
    # Zebra:
    (UUID("f7a24cb4-21fc-4002-ac70-f0e3a0dd3f62"), :Zebra) =>
        # package installed in the system depot:
        "/usr/local/julia/packages/Zebra/me9k/src/Zebra.jl",
)
```

这个例子包含三种不同类型的包位置信息（第一个和第三个是默认加载路径的一部分）

1. 私有 Priv 包”vendored” 包括在 App 仓库中。

2. 公共 Priv 与 Zebra 包位于系统仓库，系统管理员在此对相关包进行实时安装与管理。这些包允许系统上的所有用户使用。

3. Pub 包位于用户仓库，用户实时安装的包都储存在此。这些包仅限原安装用户使用。

包目录

包目录提供了一种更简单的环境，但不能处理名称冲突。在包目录中，顶层包集合是“类似”包的子目录集合。X 包存在于包目录中的条件，是目录包含下列“入口点”文件之一：

- X.jl
- X/src/X.jl
- X.jl/src/X.jl

包目录中的包可以导入哪些依赖项，取决于该包是否含有项目文件：

- 如果它有一个项目文件，那么它只能导入那些在项目文件的 [deps] 部分中已标识的包。
- 如果没有项目文件，它可以导入任何顶层包，即与在 Main 或者 REPL 中可加载的包相同。

根图是根据包目录的所有内容而形成的一个列表，包含所有已存在的包。此外，一个 UUID 将被赋予给每一个条目，例如对一个在文件夹 X 中找到的包

1. 如果 X/Project.toml 文件存在并且有一个 uuid 条目，那么这个 uuid 就是上述所要赋予的值。
2. 如果 X/Project.toml 文件存在，但没有包含一个顶层 UUID 条目，该 uuid 将是一个虚构的 UUID，是对 X/Project.toml 文件所在的规范（真实的）路径信息进行哈希处理而生成。
3. 否则（如果 Project.toml 文件不存在），uuid 将是一个全零值 nil UUID。

项目目录的**依赖关系图**是根据每个包的子目录中其项目文件的存在与否以及内容而形成。规则是：

- 如果包子目录没有项目文件，则在该图中忽略它，其代码中的 import 语句按顶层处理，与 main 项目和 REPL 相同。
- 如果包子目录有一个项目文件，那么图条目的 UUID 是项目文件的 [deps] 映射，如果该信息项不存在，则视为空。

作为一个例子，假设包目录具有以下结构和内容：

```
Aardvark/
    src/Aardvark.jl:
        import Bobcat
        import Cobra

Bobcat/
    Project.toml:
        [deps]
        Cobra = "4725e24d-f727-424b-bca0-c4307a3456fa"
        Dingo = "7a7925be-828c-4418-bbeb-bac8dfc843bc"
```

```
    src/Bobcat.jl:
        import Cobra
        import Dingo

Cobra/
    Project.toml:
        uuid = "4725e24d-f727-424b-bca0-c4307a3456fa"
        [deps]
        Dingo = "7a7925be-828c-4418-bbeb-bac8dfc843bc"

    src/Cobra.jl:
        import Dingo

Dingo/
    Project.toml:
        uuid = "7a7925be-828c-4418-bbeb-bac8dfc843bc"

    src/Dingo.jl:
        # no imports
```

下面是相应的根结构，表示为字典：

```
roots = Dict(
    :Aardvark => UUID("00000000-0000-0000-0000-000000000000"), # no project file, nil UUID
    :Bobcat   => UUID("85ad11c7-31f6-5d08-84db-0a4914d4cadf"), # dummy UUID based on path
    :Cobra    => UUID("4725e24d-f727-424b-bca0-c4307a3456fa"), # UUID from project file
    :Dingo    => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"), # UUID from project file
)
```

下面是对应的图结构，表示为字典：

```
graph = Dict(
    # Bobcat:
    UUID("85ad11c7-31f6-5d08-84db-0a4914d4cadf") => Dict(
        :Cobra => UUID("4725e24d-f727-424b-bca0-c4307a3456fa"),
        :Dingo => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"),
    ),
    # Cobra:
    UUID("4725e24d-f727-424b-bca0-c4307a3456fa") => Dict(
        :Dingo => UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"),
    ),
    # Dingo:
    UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc") => Dict(),
)
```

值得注意的一些通用规则：

1. 缺少项目文件的包能依赖于任何顶层依赖项，并且由于包目录中的每个包在顶层依赖中可用，因此它可以导入在环境中的所有包。
2. 含有项目文件的包不能依赖于缺少项目文件的包。因为有项目文件的包只能加载那些在 `graph` 中的包，而没有项目文件的包不会出现在 `graph`。

3. 具有项目文件但没有明确 UUID 的包只能被由没有项目文件的包所依赖，since dummy UUIDs assigned to these packages are strictly internal.

，因为赋予给这些包的虚构 UUID 全是项目内部的。

Observe the following specific instances of these rules in our example: 请注意以下我们例子中的规则具体实例：

- `Aardvark` 包可以导入 `Bobcat`、`Cobra` 或 `Dingo` 中的所有包；它确实导入 `Bobcat` and `Cobra` 包.
- `Bobcat` 包能导入 `Cobra` 与 `Dingo` 包。因为它们都有带有 UUID 的项目文件，并在 `Bobcat` 包的 `[deps]` 信息项声明为依赖项。
- `Bobcat` 包不能依赖于 `Aardvark` 包，因为 `Aardvark` 包缺少项目文件。
- `Cobra` 包能导入 `Dingo` 包。因为 `Dingo` 包有项目文件和 UUID，并在 `Cobra` 的 `[deps]` 信息项中声明为依赖项。
- `Cobra` 包不能依赖 `Aardvark` 或 `Bobcat` 包，因为两者都没有真实的 UUID。
- `Dingo` 包不能导入任何包，因为它的项目文件中缺少 `[deps]` 信息项。

包目录中的**路径映射**很简单: 它将子目录名映射到相应的入口点路径。换句话说，如果指向我们示例项目目录的路径是`/home/me/animals`，那么路径映射可以用此字典表示：

```
paths = Dict(
    (UUID("00000000-0000-0000-0000-000000000000"), :Aardvark) =>
        "/home/me/AnimalPackages/Aardvark/src/Aardvark.jl",
    (UUID("85ad11c7-31f6-5d08-84db-0a4914d4cadf"), :Bobcat) =>
        "/home/me/AnimalPackages/Bobcat/src/Bobcat.jl",
    (UUID("4725e24d-f727-424b-bca0-c4307a3456fa"), :Cobra) =>
        "/home/me/AnimalPackages/Cobra/src/Cobra.jl",
    (UUID("7a7925be-828c-4418-bbeb-bac8dfc843bc"), :Dingo) =>
        "/home/me/AnimalPackages/Dingo/src/Dingo.jl",
)
```

根据定义，包目录环境中的所有包都是具有预期入口点文件的子目录，因此它们的路径映射条目始终具有此格式。

环境堆栈

第三种也是最后一种环境是通过覆盖其中的几个环境来组合其他环境，使每个环境中的包在单个组合环境中可用。这些复合环境称为环境堆栈。Julia 的 `LOAD_PATH` 全局定义一个环境堆栈——Julia 进程在其中运行的环境。如果希望 Julia 进程只能访问一个项目或包目录中的包，请将其设置为 `LOAD_PATH` 中的唯一条目。然而，访问一些您喜爱的工具（标准库、探查器、调试器、个人实用程序等）通常是非常有用的，即使它们不是您正在处理的项目的依赖项。通过将包含这些工具的环境添加到加载路径，您可以立即在顶层代码中访问它们，而无需将它们添加到项目中。

组合环境堆栈组件中根、图和路径的数据结构的机制很简单：它们被作为字典进行合并，在发生键冲突时，优先使用前面的条目而不是后面的条目。换言之，如果我们有 `stack = [env₁, env₂, …]`，那么我们有：

```
roots = reduce(merge, reverse([roots₁, roots₂, ⋯]))
graph = reduce(merge, reverse([graph₁, graph₂, ⋯]))
paths = reduce(merge, reverse([paths₁, paths₂, ⋯]))
```

带下标的 $roots_i$, $graph_i$ and $paths_i$ 变量对应于在 `stack` 中包含的下标环境变量 env_i。使用 `reverse` 是因为当参数字典中的键之间发生冲突时，使 `merge` 倾向于使用最后一个参数，而不是第一个参数。这种设计有几个值得注意的特点：

1. 主环境——即堆栈中的第一个环境，被准确地嵌入到堆栈环境中。堆栈中第一个环境的完整依赖关系图是必然被完整包括在含有所有相同版本的依赖项的堆栈环境中。
2. 非主环境中的包能最终使用与其依赖项不兼容的版本，即使它们自己的环境是完全兼容。这种情况可能发生，当它们的一个依赖项被堆栈（通过图或路径，或两者）中某个早期环境中的版本所覆盖。

由于主环境通常是您正在处理的项目所在的环境，而堆栈中稍后的环境包含其他工具，因此这是正确的权衡：最好改进您的开发工具，但保持项目能工作。当这种不兼容发生时，你通常要将开发工具升级到与主项目兼容的版本。

Package Extensions

A package ”extension” is a module that is automatically loaded when a specified set of other packages (its ”extension dependencies”) are loaded in the current Julia session. Extensions are defined under the `[extensions]` section in the project file. The extension dependencies of an extension are a subset of those packages listed under the `[weakdeps]` section of the project file. Those packages can have compat entries like other packages.

```
name = "MyPackage"

[compat]
ExtDep = "1.0"
OtherExtDep = "1.0"

[weakdeps]
ExtDep = "c9a23..." # uuid
OtherExtDep = "862e..." # uuid

[extensions]
BarExt = ["ExtDep", "OtherExtDep"]
FooExt = "ExtDep"
...
```

The keys under `extensions` are the names of the extensions. They are loaded when all the packages on the right hand side (the extension dependencies) of that extension are loaded. If an extension only has one extension dependency the list of extension dependencies can be written as just a string for brevity. The location for the entry point of the extension is either in `ext/FooExt.jl` or `ext/FooExt/FooExt.jl` for extension `FooExt`. The content of an extension is often structured as:

```
module FooExt

# Load main package and extension dependencies
```

```
using MyPackage, ExtDep

# Extend functionality in main package with types from the extension dependencies
MyPackage.func(x::ExtDep.SomeStruct) = ...

end
```

When a package with extensions is added to an environment, the `weakdeps` and `extensions` sections are stored in the manifest file in the section for that package. The dependency lookup rules for a package are the same as for its "parent" except that the listed extension dependencies are also considered as dependencies.

Package/Environment Preferences

Preferences are dictionaries of metadata that influence package behavior within an environment. The preferences system supports reading preferences at compile-time, which means that at code-loading time, we must ensure that the precompilation files selected by Julia were built with the same preferences as the current environment before loading them. The public API for modifying Preferences is contained within the Preferences.jl package. Preferences are stored as TOML dictionaries within a `(Julia)LocalPreferences.toml` file next to the currently-active project. If a preference is "exported", it is instead stored within the `(Julia)Project.toml` instead. The intention is to allow shared projects to contain shared preferences, while allowing for users themselves to override those preferences with their own settings in the LocalPreferences.toml file, which should be .gitignored as the name implies.

Preferences that are accessed during compilation are automatically marked as compile-time preferences, and any change recorded to these preferences will cause the Julia compiler to recompile any cached precompilation file(s) (`.ji` and corresponding `.so`, `.dll`, or `.dylib` files) for that module. This is done by serializing the hash of all compile-time preferences during compilation, then checking that hash against the current environment when searching for the proper file(s) to load.

Preferences can be set with depot-wide defaults; if package Foo is installed within your global environment and it has preferences set, these preferences will apply as long as your global environment is part of your `LOAD_PATH`. Preferences in environments higher up in the environment stack get overridden by the more proximal entries in the load path, ending with the currently active project. This allows depot-wide preference defaults to exist, with active projects able to merge or even completely overwrite these inherited preferences. See the docstring for `Preferences.set_preferences!()` for the full details of how to set preferences to allow or disallow merging.

32.4 总结

在软件包系统中，联邦软件包管理和精确的软件可复制性是困难但有价值的目标。结合起来，这些目标导致了一个比大多数动态语言更加复杂的包加载机制，但它也产生了通常与静态语言相关的可伸缩性和可复制性。通常，Julia 用户应该能够使用内置的包管理器来管理他们的项目，而无需精确理解这些交互细节。通过调用 `Pkg.add("X")` 添加 `X` 包到对应的项目，并清晰显示相关文件，选择 `Pkg.activate("Y")` 后，可调用 `import X` 即可加载 `X` 包，而无需作过多考虑。

Chapter 33

性能分析

`Profile` 模块提供了一些工具来帮助开发者提高其代码的性能。在使用时，它运行代码并进行测量，并生成输出，该输出帮助你了解在每行（或几行）上花费了多少时间。最常见的用法是识别性能「瓶颈」并将其作为优化目标。

`Profile` 实现了所谓的「抽样」或统计分析器。它通过在执行任何任务期间定期进行回溯来工作。每次回溯捕获当前运行的函数和行号，以及导致该行执行的完整函数调用链，因此是当前执行状态的「快照」。

如果大部分运行时间都花在执行特定代码行上，则此行会在所有回溯的集合中频繁出现。换句话说，执行给定行的「成本」——或实际上，调用及包含此行的函数序列的成本——与它在所有回溯的集合中的出现频率成正比。

抽样分析器不提供完整的逐行覆盖功能，因为回溯是间隔发生的（默认情况下，该时间间隔在 Unix 上是 1 ms，而在 Windows 上是 10 ms，但实际调度受操作系统负载的影响）。此外，正如下文中进一步讨论的，因为样本是在所有执行点的稀疏子集处收集的，所以抽样分析器收集的数据会受到统计噪声的影响。

尽管有这些限制，但抽样分析器仍然有很大的优势：

- 你无需对代码进行任何修改即可进行时间测量。
- 它可以分析 Julia 的核心代码，甚至（可选）可以分析 C 和 Fortran 库。
- 通过「偶尔」运行，它只有很少的性能开销；代码在性能分析时能以接近本机的速度运行。

出于这些原因，建议你在考虑任何替代方案前尝试使用内置的抽样分析器。

33.1 基本用法

让我们使用一个简单的测试用例：

```
julia> function myfunc()
           A = rand(200, 200, 400)
           maximum(A)
       end
```

最好先至少运行一次你想要分析的代码（除非你想要分析 Julia 的 JIT 编译器）：

```
julia> myfunc() # run once to force compilation
```

现在我们准备分析这个函数：

```
julia> using Profile

julia> @profile myfunc()
```

有一些图形界面可以查看性能分析的结果。这其中有一类是基于 FlameGraphs.jl打造的，只不过提供了不同的用户接口：

- VS Code 是一个完整的 IDE，内置对性能分析可视化的支持
- ProfileView.jl 是一个基于 GTK 的独立可视化工具
- ProfileVega.jl 使用 VegaLight 并与 Jupyter notebooks 很好地集成
- StatProfilerHTML.jl 生成 HTML 并提供一些额外的摘要，并且还与 Jupyter 笔记本很好地集成
- ProfileSVG 渲染 SVG
- PProf.jl serves a local website for inspecting graphs, flamegraphs and more
- ProfileCanvas.jl is a HTML canvas based profile viewer UI, used by the Julia VS Code extension, but can also generate interactive HTML files.

一种完全独立的性能分析可视化方法是 PProf.jl，它使用外部 `pprof` 工具。

不过，在这里，我们将使用标准库附带的基于文本的显示：

```
julia> Profile.print()
80 ./event.jl:73; (::Base.REPL.##1#2{Base.REPL.REPLBackend})()
 80 ./REPL.jl:97; macro expansion
  80 ./REPL.jl:66; eval_user_input(::Any, ::Base.REPL.REPLBackend)
   80 ./boot.jl:235; eval(::Module, ::Any)
    80 ./<missing>:?; anonymous
     80 ./profile.jl:23; macro expansion
      52 ./REPL[1]:2; myfunc()
       38 ./random.jl:431; rand!(::MersenneTwister, ::Array{Float64,3}, ::Int64, ::Type{B...
        38 ./dSFMT.jl:84; dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_state, ::Ptr{F...
       14 ./random.jl:278; rand
        14 ./random.jl:277; rand
         14 ./random.jl:366; rand
          14 ./random.jl:369; rand
      28 ./REPL[1]:3; myfunc()
       28 ./reduce.jl:270; _mapreduce(::Base.#identity, ::Base.#scalarmax, ::IndexLinear,...
        3  ./reduce.jl:426; mapreduce_impl(::Base.#identity, ::Base.#scalarmax, ::Array{F...
        25 ./reduce.jl:428; mapreduce_impl(::Base.#identity, ::Base.#scalarmax, ::Array{F...
```

显示结果中的每行表示代码中的特定点（行数）。缩进用来标明嵌套的函数调用序列，其中缩进更多的行在调用序列中更深。在每一行中，第一个「字段」是在这一行或由这一行执行的任何函数中获

取的回溯（样本）数量。第二个字段是文件名和行数，第三个字段是函数名。请注意，具体的行号可能会随着 Julia 代码的改变而改变；如果你想跟上，最好自己运行这个示例。

在此例中，我们可以看到顶层的调用函数位于文件 `event.jl` 中。这是启动 Julia 时运行 REPL 的函数。如果你查看 `REPL.jl` 的第 97 行，你会看到这是调用函数 `eval_user_input()` 的地方。这是对你在 REPL 上的输入进行求值的函数，因为我们正以交互方式运行，所以当我们输入 `@profile myfunc()` 时会调用这些函数。下一行反映了 `@profile` 所采取的操作。

第一行显示在 `event.jl` 的第 73 行获取了 80 次回溯，但这并不是说此行本身「昂贵」：第三行表明所有这些 80 次回溯实际上它调用的 `eval_user_input` 中触发的，以此类推。为了找出实际占用时间的操作，我们需要深入了解调用链。

此输出中第一个「重要」的行是这行：

```
52 ./REPL[1]:2; myfunc()
```

REPL 指的是我们在 REPL 中定义了 `myfunc`，而不是把它放在文件中；如果我们使用文件，这将显示文件名。`[1]` 表示函数 `myfunc` 是在当前 REPL 会话中第一个进行求值的表达式。`myfunc()` 的第 2 行包含对 `rand` 的调用，（80 次中）有 52 次回溯发生在该行。在此之下，你可以看到在 `dSFMT.jl` 中对 `dsfmt_fill_array_close_open!` 的调用。

更进一步，你会看到：

```
28 ./REPL[1]:3; myfunc()
```

`myfunc` 的第 3 行包含对 `maximum` 的调用，（80 次中）有 28 次回溯发生在这里。在此之下，你可以看到对于这种类型的输入数据，`maximum` 函数中执行的耗时操作在 `base/reduce.jl` 中的具体位置。

总的来说，我们可以暂时得出结论，生成随机数的成本大概是找到最大元素的两倍。通过收集更多样本，我们可以增加对此结果的信心：

```
julia> @profile (for i = 1:100; myfunc(); end)

julia> Profile.print()
[....]
 3821 ./REPL[1]:2; myfunc()
  3511 ./random.jl:431; rand!(::MersenneTwister, ::Array{Float64,3}, ::Int64, ::Type...
   3511 ./dSFMT.jl:84; dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_state, ::Ptr...
  310  ./random.jl:278; rand
   [....]
 2893 ./REPL[1]:3; myfunc()
  2893 ./reduce.jl:270; _mapreduce(::Base.#identity, ::Base.#scalarmax, ::IndexLinea...
   [....]
```

一般来说，如果你在某行上收集到 `N` 个样本，那你可以预期其有 `sqrt(N)` 的不确定性（忽略其它噪音源，比如计算机在其它任务上的繁忙程度）。这个规则的主要例外是垃圾收集，它很少运行但往往成本高昂。（因为 Julia 的垃圾收集器是用 C 语言编写的，此类事件可使用下文描述的 `C=true` 输出模式来检测，或者使用 ProfileView.jl 来检测。）

这展示了默认的「树」形转储；另一种选择是「扁平」形转储，它会累积与其嵌套无关的计数：

```
julia> Profile.print(format=:flat)
 Count File          Line Function
```

```
6714 ./<missing>     -1 anonymous
6714 ./REPL.jl       66 eval_user_input(::Any, ::Base.REPL.REPLBackend)
6714 ./REPL.jl       97 macro expansion
3821 ./REPL[1]        2 myfunc()
2893 ./REPL[1]        3 myfunc()
6714 ./REPL[7]        1 macro expansion
6714 ./boot.jl      235 eval(::Module, ::Any)
3511 ./dSFMT.jl      84 dsfmt_fill_array_close_open!(::Base.dSFMT.DSFMT_s...
6714 ./event.jl      73 (::Base.REPL.##1#2{Base.REPL.REPLBackend})()
6714 ./profile.jl    23 macro expansion
3511 ./random.jl    431 rand!(::MersenneTwister, ::Array{Float64,3}, ::In...
 310 ./random.jl    277 rand
 310 ./random.jl    278 rand
 310 ./random.jl    366 rand
 310 ./random.jl    369 rand
2893 ./reduce.jl    270 _mapreduce(::Base.#identity, ::Base.#scalarmax, :...
   5 ./reduce.jl    420 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
 253 ./reduce.jl    426 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
2592 ./reduce.jl    428 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
  43 ./reduce.jl    429 mapreduce_impl(::Base.#identity, ::Base.#scalarma...
```

如果你的代码有递归，那么可能令人困惑的就是「子」函数中的行的累积计数可以多于总回溯次数。考虑以下函数定义：

```
dumbsum(n::Integer) = n == 1 ? 1 : 1 + dumbsum(n-1)
dumbsum3() = dumbsum(3)
```

如果你要分析 `dumbsum3`，并在执行 `dumbsum(1)` 时执行了回溯，那么该回溯将如下所示：

```
dumbsum3
    dumbsum(3)
        dumbsum(2)
            dumbsum(1)
```

因此，即使父函数只获得 1 个计数，这个子函数也会获得 3 个计数。「树」形表示使这更清晰，因此（以及其它原因）可能是查看结果的最实用方法。

33.2 结果累积和清空

`@profile` 的结果会累积在一个缓冲区中；如果你在 `@profile` 下运行多端代码，那么 `Profile.print()` 会显示合并的结果。这可能非常有用，但有时你会想重新开始，这可通过 `Profile.clear()`。

33.3 用于控制性能分析结果显示的选项

`Profile.print` 还有一些未曾描述的选项。让我们看看完整的声明：

```
function print(io::IO = stdout, data = fetch(); kwargs...)
```

我们先讨论两个位置参数，然后讨论关键字参数：

- io——允许你将结果保存到缓冲区，例如一个文件，但默认是打印到 stdout（控制台）。
- data——包含你要分析的数据；默认情况下，它是从 Profile.fetch() 中获取的，该函数从预先分配的缓冲区中拉出回溯。例如，如果你要分析性能分析器，可以说：

```
data = copy(Profile.fetch())
Profile.clear()
@profile Profile.print(stdout, data) # Prints the previous results
Profile.print()                      # Prints results from Profile.print()
```

关键字参数可以是以下参数的任意组合：

- format——上文已经介绍，确定是使用（默认值，:tree）还是不使用（:flat）缩进来表示其树形结构。
- C——如果为 true，则显示 C 和 Fortran 代码中的回溯（通常它们被排除在外）。请尝试用 Profile.print(C = true) 运行介绍性示例。这对于判断是 Julia 代码还是 C 代码导致了性能瓶颈非常有帮助；设置 C = true 也可提高嵌套的可解释性，代价是更长的性能分析转储。
- combine——某些代码行包含多个操作；例如，s += A[i] 包含一个数组引用（A[i]）和一个求和操作。这些操作在所生成的机器代码中对应不同的行，因此回溯期间可能会在此行中捕获两个或以上地址。combine = true 把它们混合在一起，可能你通常想要这样，但使用 combine = false，你可为每个唯一的指令指针单独生成输出。
- maxdepth——限制 :tree 格式中深度大于 maxdepth 的帧。
- * sortedby——控制 :flat 格式中的次序。为 :filefuncline（默认值）时按源代码行排序，而为 :count 时按收集的样本数排序。
- noisefloor——限制低于样本的启发式噪音下限的帧（只适用于格式 :tree）。尝试此选项的建议值是 2.0（默认值是 0）。此参数会隐藏 n <= noisefloor * √N 的样本，其中 n 是该行上的样本数，N 是被调用者的样本数。
- mincount——限制出现次数少于 mincount 的帧。

文件/函数名有时会被（用 ...）截断，缩进也有可能在开头用 +n 截断，其中 n 是在空间充足的情况下应该插入的额外空格数。如果你想要深层嵌套代码的完整性能分析，保存到文件并在 IOContext 中使用宽的 displaysize 通常是个好主意：

```
open("/tmp/prof.txt", "w") do s
    Profile.print(IOContext(s, :displaysize => (24, 500)))
end
```

33.4 配置

@profile 只是累积回溯，在你调用 Profile.print() 时才会进行性能分析。对于长时间运行的计算，完全有可能把用于存储回溯的预分配缓冲区填满。如果发生这种情况，回溯会停止，但你的计算会继续。因此，你也许会丢失一些重要的性能分析数据（当发生这种情况时，你会受到警告）。

你可通过以下方式获取和配置相关参数：

```
Profile.init() # returns the current settings
Profile.init(n = 10^7, delay = 0.01)
```

n 是能够存储的指令指针总数，默认值为 10^6。如果通常的回溯是 20 个指令指针，那么可以收集 50000 次回溯，这意味着统计不确定性少于 1%。这对于大多数应用来说可能已经足够了。

因此，你更可能需要修改 delay，它以秒为单位，设置在快照之间 Julia 用于执行所请求计算的时长。长时间运行的工作可能不需要经常回溯。默认设置为 delay = 0.001。当然，你可以减少和增加 delay；但是，一旦 delay 接近执行一次回溯所需的时间（在作者的笔记本上约为 30 微妙），性能分析的开销就会增加。

33.5 内存分配分析

减少内存分配是提高性能的最常用技术之一。Julia provides several tools measure this:

@time

The total amount of allocation can be measured with @time, @allocated and @allocations, and specific lines triggering allocation can often be inferred from profiling via the cost of garbage collection that these lines incur. However, sometimes it is more efficient to directly measure the amount of memory allocated by each line of code.

GC Logging

While @time logs high-level stats about memory usage and garbage collection over the course of evaluating an expression, it can be useful to log each garbage collection event, to get an intuitive sense of how often the garbage collector is running, how long it's running each time, and how much garbage it collects each time. This can be enabled with GC.enable_logging(true), which causes Julia to log to stderr every time a garbage collection happens.

Allocation Profiler

Julia 1.8

This functionality requires at least Julia 1.8.

The allocation profiler records the stack trace, type, and size of each allocation while it is running. It can be invoked with Profile.Allocs.@profile.

This information about the allocations is returned as an array of Alloc objects, wrapped in an AllocResults object. The best way to visualize these is currently with the PProf.jl and ProfileCanvas.jl packages, which can visualize the call stacks which are making the most allocations.

The allocation profiler does have significant overhead, so a sample_rate argument can be passed to speed it up by making it skip some allocations. Passing sample_rate=1.0 will make it record everything (which is slow); sample_rate=0.1 will record only 10% of the allocations (faster), etc.

> **Note**
>
> The current implementation of the Allocations Profiler *does not capture types for all allocations.* Allocations for which the profiler could not capture the type are represented as having type `Profile.Allocs.UnknownType`.
>
> You can read more about the missing types and the plan to improve this, here: issue #43688.

Line-by-Line Allocation Tracking

An alternative way to measure allocations is to start Julia with the `--track-allocation=<setting>` command-line option, for which you can choose none (the default, do not measure allocation), `user` (measure memory allocation everywhere except Julia's core code), or `all` (measure memory allocation at each line of Julia code). Allocation gets measured for each line of compiled code. When you quit Julia, the cumulative results are written to text files with `.mem` appended after the file name, residing in the same directory as the source file. Each line lists the total number of bytes allocated. The `Coverage package` contains some elementary analysis tools, for example to sort the lines in order of number of bytes allocated.

在解释结果时，有一些需要注意的细节。在 `user` 设定下，直接从 REPL 调用的任何函数的第一行都将会显示内存分配，这是由发生在 REPL 代码本身的事件造成的。更重要的是，JIT 编译也会添加内存分配计数，因为 Julia 的编译器大部分是用 Julia 编写的（并且编译通常需要内存分配）。建议的分析过程是先通过执行待分析的所有命令来强制编译，然后调用 `Profile.clear_malloc_data()` 来重置所有内存计数器。最后，执行所需的命令并退出 Julia 以触发 `.mem` 文件的生成。

> **Note**
>
> `--track-allocation` changes code generation to log the allocations, and so the allocations may be different than what happens without the option. We recommend using the allocation profiler instead.

33.6 外部性能分析

Julia 目前支持的外部性能分析工具有 `Intel VTune`、`OProfile` 和 `perf`。

根据你所选择的工具，编译时请在 `Make.user` 中将 `USE_INTEL_JITEVENTS`、`USE_OPROFILE_JITEVENTS` 和 `USE_PERF_JITEVENTS` 设置为 `1`。多个上述编译标志是支持的。

在运行 Julia 前，请将环境变量 `ENABLE_JITPROFILING` 设置为 `1`。

现在，你可以通过多种方式使用这些工具！例如，可以使用 `OProfile` 来尝试做个简单的记录：

```
>ENABLE_JITPROFILING=1 sudo operf -Vdebug ./julia test/fastmath.jl
>opreport -l `which ./julia`
```

或与 `perf` 类似：

```
$ ENABLE_JITPROFILING=1 perf record -o /tmp/perf.data --call-graph dwarf -k 1 ./julia
↪  /test/fastmath.jl
$ perf inject --jit --input /tmp/perf.data --output /tmp/perf-jit.data
$ perf report --call-graph -G -i /tmp/perf-jit.data
```

你可以测量关于程序的更多有趣数据，若要获得详尽的列表，请阅读 Linux perf 示例页面。

请记住，perf 会为每次执行保存一个 `perf.data` 文件，即使对于小程序，它也可能变得非常大。此外，perf LLVM 模块会将调试对象保存在 `~/.debug/jit` 中，记得经常清理该文件夹。

Chapter 34

栈跟踪

StackTraces 模块提供了简单的栈跟踪功能，这些栈跟踪信息既可读又易于编程使用。

34.1 查看栈跟踪

获取栈跟踪信息的主要函数是 stacktrace：

```
6-element Array{Base.StackTraces.StackFrame,1}:
 top-level scope
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 macro expansion at REPL.jl:116 [inlined]
 (::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
```

调用 stacktrace() 会返回一个 StackTraces.StackFrame 数组。为了使用方便，可以用 StackTraces.StackTrace 来代替 Vector{StackFrame}。下面例子中 [...] 的意思是这部分输出的内容可能会根据代码的实际执行情况而定。

```
julia> example() = stacktrace()
example (generic function with 1 method)

julia> example()
7-element Array{Base.StackTraces.StackFrame,1}:
 example() at REPL[1]:1
 top-level scope
 eval at boot.jl:317 [inlined]
[...]

julia> @noinline child() = stacktrace()
child (generic function with 1 method)

julia> @noinline parent() = child()
parent (generic function with 1 method)

julia> grandparent() = parent()
grandparent (generic function with 1 method)

julia> grandparent()
```

```
9-element Array{Base.StackTraces.StackFrame,1}:
 child() at REPL[3]:1
 parent() at REPL[4]:1
 grandparent() at REPL[5]:1
[...]
```

注意，在调用 stacktrace() 的时，通常会出现 eval at boot.jl 这帧。当从 REPL 里调用 stacktrace() 的时候，还会显示 REPL.jl 里的一些额外帧，就像下面一样：

```
julia> example() = stacktrace()
example (generic function with 1 method)

julia> example()
7-element Array{Base.StackTraces.StackFrame,1}:
 example() at REPL[1]:1
 top-level scope
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 macro expansion at REPL.jl:116 [inlined]
 (::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
```

34.2 抽取有用信息

每个 StackTraces.StackFrame 都会包含函数名，文件名，代码行数，lambda 信息，一个用于确认此帧是否被内联的标帜，一个用于确认函数是否为 C 函数的标帜（在默认的情况下 C 函数不会出现在栈跟踪信息中）以及一个用整数表示的指针，它是由 backtrace 返回的：

```
julia> frame = stacktrace()[3]
eval(::Module, ::Expr) at REPL.jl:5

julia> frame.func
:eval

julia> frame.file
Symbol("~/julia/usr/share/julia/stdlib/v0.7/REPL/src/REPL.jl")

julia> frame.line
5

julia> frame.linfo
MethodInstance for eval(::Module, ::Expr)

julia> frame.inlined
false

julia> frame.from_c
false

julia> frame.pointer
0x00007f92d6293171
```

这使得我们可以通过编程的方式将栈跟踪信息用于打印日志，处理错误以及其它更多用途。

34.3 错误处理

能够轻松地获取当前调用栈的状态信息在许多场景下都很有用，但最直接的应用是错误处理和调试。

```
julia> @noinline bad_function() = undeclared_variable
bad_function (generic function with 1 method)

julia> @noinline example() = try
           bad_function()
       catch
           stacktrace()
       end
example (generic function with 1 method)

julia> example()
7-element Array{Base.StackTraces.StackFrame,1}:
 example() at REPL[2]:4
 top-level scope
 eval at boot.jl:317 [inlined]
[...]
```

你可能已经注意到了，上述例子中第一个栈帧指向了stacktrace被调用的第 4 行，而不是 *bad_function* 被调用的第 2 行，且完全没有出现 bad_function 的栈帧。这是也是可以理解的，因为 stacktrace 是在 *catch* 的上下文中被调用的。虽然在这个例子中很容易查找到错误的真正源头，但在复杂的情况下查找错误源并不是一件容易的事。

为了补救，我们可以将 catch_backtrace 的输出传递给 stacktrace。catch_backtrace 会返回最近发生异常的上下文中的栈信息，而不是返回当前上下文中的调用栈信息。

```
julia> @noinline bad_function() = undeclared_variable
bad_function (generic function with 1 method)

julia> @noinline example() = try
           bad_function()
       catch
           stacktrace(catch_backtrace())
       end
example (generic function with 1 method)

julia> example()
8-element Array{Base.StackTraces.StackFrame,1}:
 bad_function() at REPL[1]:1
 example() at REPL[2]:2
[...]
```

可以看到，现在栈跟踪会显示正确的行号以及之前缺失的栈帧。

```
julia> @noinline child() = error("Whoops!")
child (generic function with 1 method)

julia> @noinline parent() = child()
```

```
parent (generic function with 1 method)

julia> @noinline function grandparent()
           try
               parent()
           catch err
               println("ERROR: ", err.msg)
               stacktrace(catch_backtrace())
           end
       end
grandparent (generic function with 1 method)

julia> grandparent()
ERROR: Whoops!
10-element Array{Base.StackTraces.StackFrame,1}:
 error at error.jl:33 [inlined]
 child() at REPL[1]:1
 parent() at REPL[2]:1
 grandparent() at REPL[3]:3
[...]
```

34.4 异常栈与 current_exceptions

Julia 1.1

异常栈需要 Julia 1.1 及以上版本。

在处理一个异常时，后续的异常同样可能被抛出。观察这些异常对定位问题的源头极有帮助。Julia runtime 支持将每个异常发生后推入一个内部的异常栈。当代码正常退出一个 catch 语句，可认为所有被推入栈中的异常在相应的 try 语句中被成功处理并已从栈中移除。

The stack of current exceptions can be accessed using the current_exceptions function. For example,

```
julia> try
           error("(A) The root cause")
       catch
           try
               error("(B) An exception while handling the exception")
           catch
               for (exc, bt) in current_exceptions()
                   showerror(stdout, exc, bt)
                   println(stdout)
               end
           end
       end
(A) The root cause
Stacktrace:
 [1] error(::String) at error.jl:33
 [2] top-level scope at REPL[7]:2
 [3] eval(::Module, ::Any) at boot.jl:319
 [4] eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 [5] macro expansion at REPL.jl:117 [inlined]
 [6] (::getfield(REPL, Symbol("##26#27")){REPL.REPLBackend})() at task.jl:259
```

```
(B) An exception while handling the exception
Stacktrace:
 [1] error(::String) at error.jl:33
 [2] top-level scope at REPL[7]:5
 [3] eval(::Module, ::Any) at boot.jl:319
 [4] eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 [5] macro expansion at REPL.jl:117 [inlined]
 [6] (::getfield(REPL, Symbol("##26#27")){REPL.REPLBackend})() at task.jl:259
```

在本例中，根源异常（A）排在栈头，其后放置着延伸异常（B）。在正常退出（例如，不抛出新异常）两个 catch 块后，所有异常都被移除出栈，无法访问。

The exception stack is stored on the Task where the exceptions occurred. When a task fails with uncaught exceptions, `current_exceptions(task)` may be used to inspect the exception stack for that task.

34.5 stacktrace 与 backtrace 的比较

调用 `backtrace` 会返回一个 `Union{Ptr{Nothing}, Base.InterpreterIP}` 的数组，可以将其传给 `stacktrace` 函数进行转化：

```
julia> trace = backtrace()
18-element Array{Union{Ptr{Nothing}, Base.InterpreterIP},1}:
 Ptr{Nothing} @0x00007fd8734c6209
 Ptr{Nothing} @0x00007fd87362b342
 Ptr{Nothing} @0x00007fd87362c136
 Ptr{Nothing} @0x00007fd87362c986
 Ptr{Nothing} @0x00007fd87362d089
 Base.InterpreterIP(CodeInfo(:(begin
      Core.SSAValue(0) = backtrace()
      trace = Core.SSAValue(0)
      return Core.SSAValue(0)
  end)), 0x0000000000000000)
 Ptr{Nothing} @0x00007fd87362e4cf
[...]

julia> stacktrace(trace)
6-element Array{Base.StackTraces.StackFrame,1}:
 top-level scope
 eval at boot.jl:317 [inlined]
 eval(::Module, ::Expr) at REPL.jl:5
 eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
 macro expansion at REPL.jl:116 [inlined]
 (::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
```

需要注意的是，`backtrace` 返回的向量有 18 个元素，而 `stacktrace` 返回的向量只包含 6 个元素。这是因为 `stacktrace` 在默认情况下会移除所有底层 C 函数的栈信息。如果你想显示 C 函数调用的栈帧，可以这样做：

```
julia> stacktrace(trace, true)
21-element Array{Base.StackTraces.StackFrame,1}:
 jl_apply_generic at gf.c:2167
 do_call at interpreter.c:324
```

```
eval_value at interpreter.c:416
eval_body at interpreter.c:559
jl_interpret_toplevel_thunk_callback at interpreter.c:798
top-level scope
jl_interpret_toplevel_thunk at interpreter.c:807
jl_toplevel_eval_flex at toplevel.c:856
jl_toplevel_eval_in at builtins.c:624
eval at boot.jl:317 [inlined]
eval(::Module, ::Expr) at REPL.jl:5
jl_apply_generic at gf.c:2167
eval_user_input(::Any, ::REPL.REPLBackend) at REPL.jl:85
jl_apply_generic at gf.c:2167
macro expansion at REPL.jl:116 [inlined]
(::getfield(REPL, Symbol("##28#29")){REPL.REPLBackend})() at event.jl:92
jl_fptr_trampoline at gf.c:1838
jl_apply_generic at gf.c:2167
jl_apply at julia.h:1540 [inlined]
start_task at task.c:268
ip:0xffffffffffffffff
```

`backtrace` 返回的单个指针可以通过 `StackTraces.lookup` 来转化成一组 `StackTraces.StackFrame`：

```
julia> pointer = backtrace()[1];

julia> frame = StackTraces.lookup(pointer)
1-element Array{Base.StackTraces.StackFrame,1}:
 jl_apply_generic at gf.c:2167

julia> println("The top frame is from $(frame[1].func)!")
The top frame is from jl_apply_generic!
```

Chapter 35

性能建议

下面几节简要地介绍了一些使 Julia 代码运行得尽可能快的技巧。

35.1 影响性能的关键代码应该在函数内部

任何对性能至关重要的代码都应该在函数内部。由于 Julia 编译器的工作方式，函数内部的代码往往比顶层代码运行得更快。

函数的使用不仅对性能很重要：函数更可重用和可测试，并阐明正在执行哪些步骤以及它们的输入和输出是什么，[编写函数，而不仅仅是脚本]（@ref）也是 Julia 的风格指南。

The functions should take arguments, instead of operating directly on global variables, see the next point.

35.2 Avoid untyped global variables

The value of an untyped global variable might change at any point, possibly leading to a change of its type. This makes it difficult for the compiler to optimize code using global variables. This also applies to type-valued variables, i.e. type aliases on the global level. Variables should be local, or passed as arguments to functions, whenever possible.

函数应该接收参数，而不是直接对全局变量进行操作，请参阅下一点。

35.3 避免全局变量

全局变量的值和类型随时都会发生变化，这使编译器难以优化使用全局变量的代码。变量应该是局部的，或者尽可能作为参数传递给函数。

我们发现全局变量经常是常量，将它们声明为常量可大幅提升性能。

```
const DEFAULT_VAL = 0
```

If a global is known to always be of the same type, the type should be annotated.

对于非常量的全局变量可以通过在使用的时候标注它们的类型来优化。

```
global x = rand(1000)

function loop_over_global()
    s = 0.0
```

```
    for i in x::Vector{Float64}
        s += i
    end
    return s
end
```

一个更好的编程风格是将变量作为参数传给函数。这样可以使得代码更易复用，以及清晰的展示函数的输入和输出。

> **Note**
>
> 所有的 REPL 中的代码都是在全局作用域中求值的，因此在顶层的变量的定义与赋值都会成为一个**全局**变量。在模块的顶层作用域定义的变量也是全局变量。

在下面的 REPL 会话中：

```
julia> x = 1.0
```

等价于：

```
julia> global x = 1.0
```

因此，所有上文关于性能问题的讨论都适用于它们。

35.4 使用 `@time` 评估性能以及注意内存分配

`@time` 宏是一个有用的性能评估工具。这里我们将重复上面全局变量的例子，但是这次移除类型声明：

```
julia> x = rand(1000);

julia> function sum_global()
           s = 0.0
           for i in x
               s += i
           end
           return s
       end;

julia> @time sum_global()
  0.011539 seconds (9.08 k allocations: 373.386 KiB, 98.69% compilation time)
523.0007221951678

julia> @time sum_global()
  0.000091 seconds (3.49 k allocations: 70.156 KiB)
523.0007221951678
```

在第一次调用（`@time sum_global()`）时，函数被编译。（如果不在此会话中使用 `@time`，它还会编译计时所需的函数）你不用把此次运行的结果放在心上。对于第二次运行，请注意，除了报告时间外，

它还表明分配了大量内存。我们在这里只是计算 64 位浮点数向量中所有元素的总和，因此不需要分配（堆）内存

We should clarify that what `@time` reports is specifically *heap* allocations, which are typically needed for either mutable objects or for creating/growing variable-sized containers (such as `Array` or `Dict`, strings, or "type-unstable" objects whose type is only known at runtime). Allocating (or deallocating) such blocks of memory may require an expensive system call (e.g. via `malloc` in C), and they must be tracked for garbage collection. In contrast, immutable values like numbers (except bignums), tuples, and immutable `structs` can be stored much more cheaply, e.g. in stack or CPU-register memory, so one doesn't typically worry about the performance cost of "allocating" them.

预料之外的内存分配几乎总是表示你的代码存在问题，通常是类型稳定性问题或创建了许多小的临时数组。因此，除了分配本身之外，你的函数的代码很可能远非最优。请认真对待此类迹象并遵循以下建议。

In this particular case, the memory allocation is due to the usage of a type-unstable global variable x, so if we instead pass x as an argument to the function it no longer allocates memory (the remaining allocation reported below is due to running the `@time` macro in global scope) and is significantly faster after the first call:

```
julia> x = rand(1000);

julia> function sum_arg(x)
           s = 0.0
           for i in x
               s += i
           end
           return s
       end;

julia> @time sum_arg(x)
  0.007551 seconds (3.98 k allocations: 200.548 KiB, 99.77% compilation time)
523.0007221951678

julia> @time sum_arg(x)
  0.000006 seconds (1 allocation: 16 bytes)
523.0007221951678
```

看到的 1 allocation 来自在全局范围内运行 `@time` 宏本身。如果我们改为在函数中运行计时，我们可以看到确实没有执行任何分配：

```
julia> time_sum(x) = @time sum_arg(x);

julia> time_sum(x)
  0.000002 seconds
523.0007221951678
```

在某些情况下，你的函数可能需要将分配内存作为其操作的一部分，这比上面的简单例子复杂的多。在这种情况下，请考虑使用下面的 工具 之一来诊断问题，或者编写一个将分配内存与算法方面分开的函数版本（请参阅 输出预分配）。

> **Note**
>
> 对于更严格的基准测试，请考虑 BenchmarkTools.jl 包，该包会多次评估函数以减少噪音。

35.5 工具

Julia 及其包生态系统包括可以帮助您诊断问题和提高代码性能的工具：

- Profiling 允许你测量正在运行的代码的性能并识别作为瓶颈的行。对于复杂的项目，ProfileView 包可以帮助你可视化分析结果。
- Traceur 包可以帮助你找到代码中常见的性能问题。
- 预期之外的大内存分配——正如 @time、@allocated 或 Profiler（通过调用垃圾收集例程）所报告的那样——暗示你的代码可能有问题。如果你没有看到分配的其他原因，请怀疑是类型问题。还可以使用 --track allocation=user 选项启动 Julia 并检查生成的 *.mem 文件以查看有关这些分配发生位置的信息。参见内存分配分析。
- @code_warntype 生成代码的表示形式，这有助于查找导致类型不确定性的表达式。请参阅下面的 @code_warntype。

35.6 避免使用抽象类型参数的容器

使用参数化类型（包括数组）时，最好尽可能避免使用抽象类型进行参数化。

考虑如下：

```
julia> a = Real[]
Real[]

julia> push!(a, 1); push!(a, 2.0); push!(a, π)
3-element Vector{Real}:
 1
 2.0
 π = 3.1415926535897...
```

因为 a 是一个抽象类型 Real 的数组，所以它能够保存任何 Real 值。由于 Real 对象允许具有任意大小和结构，因此 a 必须表示为指向单独分配的 Real 对象的指针数组。但是，如果我们只允许相同类型的数字，例如 Float64，a 可以更有效地存储：

```
julia> a = Float64[]
Float64[]

julia> push!(a, 1); push!(a, 2.0); push!(a,  π)
3-element Vector{Float64}:
 1.0
 2.0
 3.141592653589793
```

把数字赋值给 a 会即时将数字转换成 Float64 并且 a 会按照 64 位浮点数值的连续的块来储存，这就能高效地处理。

如果无法避免使用抽象值类型的容器，有时最好使用 Any 参数化以避免运行时类型检查。例如。IdDict{Any, Any} 的性能优于 IdDict{Type, Vector}。

也请参见在参数类型下的讨论。

35.7 添加类型声明

在有可选类型声明的语言中，添加声明是使代码运行更快的原则性方法。在 Julia 中并不是这种情况。在 Julia 中，编译器都知道所有的函数参数，局部变量和表达式的类型。但是，有一些特殊的情况下声明是有帮助的。

避免有抽象类型的字段

类型能在不指定其字段的类型的情况下被声明：

```
julia> struct MyAmbiguousType
           a
       end
```

这就允许 a 可以是任意类型。这经常很有用，但是有个缺点：对于类型 MyAmbiguousType 的对象，编译器不能够生成高性能的代码。原因是编译器使用对象的类型，而非值，来确定如何构建代码。不幸的是，几乎没有信息可以从类型 MyAmbiguousType 的对象中推导出来：

```
julia> b = MyAmbiguousType("Hello")
MyAmbiguousType("Hello")

julia> c = MyAmbiguousType(17)
MyAmbiguousType(17)

julia> typeof(b)
MyAmbiguousType

julia> typeof(c)
MyAmbiguousType
```

b 和 c 的值具有相同类型，但它们在内存中的数据的底层表示十分不同。即使你只在字段 a 中存储数值，UInt8 的内存表示与 Float64 也是不同的，这也意味着 CPU 需要使用两种不同的指令来处理它们。因为该类型中不提供所需的信息，所以必须在运行时进行这些判断。而这会降低性能。

通过声明 a 的类型，你能够做得更好。这里我们关注 a 可能是几种类型中任意一种的情况，在这种情况下，自然的一个解决方法是使用参数。例如：

```
julia> mutable struct MyType{T<:AbstractFloat}
           a::T
       end
```

比下面这种更好

```
julia> mutable struct MyStillAmbiguousType
           a::AbstractFloat
       end
```

因为第一种通过包装对象的类型指定了 a 的类型。例如：

```
julia> m = MyType(3.2)
MyType{Float64}(3.2)

julia> t = MyStillAmbiguousType(3.2)
MyStillAmbiguousType(3.2)

julia> typeof(m)
MyType{Float64}

julia> typeof(t)
MyStillAmbiguousType
```

字段 `a` 的类型可以很容易地通过 `m` 的类型确定，而不是通过 `t` 的类型确定。事实上，在 `t` 中是可以改变字段 `a` 的类型的：

```
julia> typeof(t.a)
Float64

julia> t.a = 4.5f0
4.5f0

julia> typeof(t.a)
Float32
```

反之，一旦 `m` 被构建出来，`m.a` 的类型就不能够更改了。

```
julia> m.a = 4.5f0
4.5f0

julia> typeof(m.a)
Float64
```

`m.a` 的类型是通过 `m` 的类型得知这一事实加上它的类型不能改变在函数中改变这一事实，这两者使得对于像 `m` 这样的对象编译器可以生成高度优化后的代码，但是对 `t` 这样的对象却不可以。当然，如果我们将 `m` 构造成一个具体类型，那么这两者都可以。我们可以通过明确地使用一个抽象类型去构建它来破坏这一点：

```
julia> m = MyType{AbstractFloat}(3.2)
MyType{AbstractFloat}(3.2)

julia> typeof(m.a)
Float64

julia> m.a = 4.5f0
4.5f0

julia> typeof(m.a)
Float32
```

对于一个实际的目的来说，这样的对象表现起来和那些 `MyStillAmbiguousType` 的对象一模一样。

比较为一个简单函数生成的代码的绝对数量是十分有指导意义的，

```
func(m::MyType) = m.a+1
```

使用

```
code_llvm(func, Tuple{MyType{Float64}})
code_llvm(func, Tuple{MyType{AbstractFloat}})
```

由于长度的原因，代码的结果没有在这里显示出来，但是你可能会希望自己去验证这一点。因为在第一种情况中，类型被完全指定了，在运行时，编译器不需要生成任何代码来决定类型。这就带来了更短和更快的代码。

One should also keep in mind that not-fully-parameterized types behave like abstract types. For example, even though a fully specified `Array{T,n}` is concrete, `Array` itself with no parameters given is not concrete:

```
julia> !isconcretetype(Array), !isabstracttype(Array), isstructtype(Array),
↪  !isconcretetype(Array{Int}), isconcretetype(Array{Int,1})
(true, true, true, true, true)
```

In this case, it would be better to avoid declaring `MyType` with a field `a::Array` and instead declare the field as `a::Array{T,N}` or as `a::A`, where `{T,N}` or `A` are parameters of `MyType`.

避免使用带抽象容器的字段

上面的做法同样也适用于容器的类型：

```
julia> struct MySimpleContainer{A<:AbstractVector}
           a::A
       end

julia> struct MyAmbiguousContainer{T}
           a::AbstractVector{T}
       end

julia> struct MyAlsoAmbiguousContainer
           a::Array
       end
```

例如:

```
julia> c = MySimpleContainer(1:3);

julia> typeof(c)
MySimpleContainer{UnitRange{Int64}}

julia> c = MySimpleContainer([1:3;]);

julia> typeof(c)
MySimpleContainer{Vector{Int64}}
```

```
julia> b = MyAmbiguousContainer(1:3);

julia> typeof(b)
MyAmbiguousContainer{Int64}

julia> b = MyAmbiguousContainer([1:3;]);

julia> typeof(b)
MyAmbiguousContainer{Int64}

julia> d = MyAlsoAmbiguousContainer(1:3);

julia> typeof(d), typeof(d.a)
(MyAlsoAmbiguousContainer, Vector{Int64})

julia> d = MyAlsoAmbiguousContainer(1:1.0:3);

julia> typeof(d), typeof(d.a)
(MyAlsoAmbiguousContainer, Vector{Float64})
```

对于 `MySimpleContainer` 来说，它被它的类型和参数完全确定了，因此编译器能够生成优化过的代码。在大多数实例中，这点能够实现。

尽管编译器现在可以将它的工作做得非常好，但是还是**有你**可能希望你的代码能够能够根据 a 的**元素类型**做不同的事情的时候。通常达成这个目的最好的方式是将你的具体操作 (here, foo) 打包到一个独立的函数中。

```
julia> function sumfoo(c::MySimpleContainer)
           s = 0
           for x in c.a
               s += foo(x)
           end
           s
       end
sumfoo (generic function with 1 method)

julia> foo(x::Integer) = x
foo (generic function with 1 method)

julia> foo(x::AbstractFloat) = round(x)
foo (generic function with 2 methods)
```

这使事情变得简单，同时也允许编译器在所有情况下生成经过优化的代码。

但是，在某些情况下，你可能需要声明外部函数的不同版本，这可能是为了不同的元素类型，也可能是为了 `MySimpleContainer` 中的字段 a 所具有的不同 `AbstractVector` 类型。你可以这样做：

```
julia> function myfunc(c::MySimpleContainer{<:AbstractArray{<:Integer}})
           return c.a[1]+1
       end
myfunc (generic function with 1 method)

julia> function myfunc(c::MySimpleContainer{<:AbstractArray{<:AbstractFloat}})
           return c.a[1]+2
```

```
       end
myfunc (generic function with 2 methods)

julia> function myfunc(c::MySimpleContainer{Vector{T}}) where T <: Integer
           return c.a[1]+3
       end
myfunc (generic function with 3 methods)
```

```
julia> myfunc(MySimpleContainer(1:3))
2

julia> myfunc(MySimpleContainer(1.0:3))
3.0

julia> myfunc(MySimpleContainer([1:3;]))
4
```

对从无类型位置获取的值进行类型注释

使用可能包含任何类型的值的数据结构（如类型为 Array{Any} 的数组）经常是很方便的。但是，如果你正在使用这些数据结构之一，并且恰巧知道某个元素的类型，那么让编译器也知道这一点会有所帮助：

```
function foo(a::Array{Any,1})
    x = a[1]::Int32
    b = x+1
    ...
end
```

在这里，我们恰巧知道 a 的第一个元素是个 Int32。留下这样的注释还有另外的好处，它将在该值不是预期类型时引发运行时错误，而这可能会更早地捕获某些错误。

在没有确切知道 a[1] 的类型的情况下，x 可以通过 x = convert(Int32, a[1])::Int32 来声明。使用 convert 函数则允许 a[1] 是可转换为 Int32 的任何对象（比如 UInt8），从而通过放松类型限制来提高代码的通用性。请注意，convert 本身在此上下文中需要类型注释才能实现类型稳定性。这是因为除非该函数所有参数的类型都已知，否则编译器无法推导出该函数返回值的类型，即使其为 convert。

如果类型是抽象的或在运行时构造的，则类型注释不会增强（实际上可能会阻碍）性能。这是因为编译器无法使用注解来特例化后续代码，并且类型检查本身需要时间。例如，在代码中：

```
function nr(a, prec)
    ctype = prec == 32 ? Float32 : Float64
    b = Complex{ctype}(a)
    c = (b + 1.0f0)::Complex{ctype}
    abs(c)
end
```

c 的注释会损害性能。要编写涉及在运行时构造类型的高性能代码，请使用下面讨论的 函数障碍技巧，并确保构造的类型出现在内核函数的参数类型中，以便内核操作由编译器合理地 特例化。例如，在上面的代码片段中，一旦构建了 b，它就可以传递给另一个函数 k，即内核。例如，如果函数 k 将

b 声明为类型为 Complex{T} 的参数，其中 T 是一个类型参数，那么出现在 k 的赋值语句中的类型注释的形式：

```
c = (b + 1.0f0)::Complex{T}
```

不会降低性能（但也不会有帮助），因为编译器可以在编译 k 时确定 c 的类型。

注意 Julia 何时避免特例化

作为一种启发式方法，Julia 避免在三种特定情况下自动特例化参数类型参数：Type、Function 和 Vararg。当在方法中使用参数时，Julia 将始终特例化，但如果参数只是传递给另一个函数，则不会。这通常在运行时没有性能影响并且提高编译器性能。如果你发现它在你的案例中在运行时确实有性能影响，您可以通过向方法声明添加类型参数来触发特例化。这里有些例子：

这不会特例化：

```
function f_type(t)  # or t::Type
    x = ones(t, 10)
    return sum(map(sin, x))
end
```

但是这会：

```
function g_type(t::Type{T}) where T
    x = ones(T, 10)
    return sum(map(sin, x))
end
```

这些不会特例化：

```
f_func(f, num) = ntuple(f, div(num, 2))
g_func(g::Function, num) = ntuple(g, div(num, 2))
```

但是这会：

```
h_func(h::H, num) where {H} = ntuple(h, div(num, 2))
```

这不会特例化：

```
f_vararg(x::Int...) = tuple(x...)
```

但是这会：

```
g_vararg(x::Vararg{Int, N}) where {N} = tuple(x...)
```

只需要引入一个类型参数就可以强制特例化，即使其他类型不受约束。比如下面这个例子，它会特例化，并且在参数不是全部相同类型时很有用：

```
h_vararg(x::Vararg{Any, N}) where {N} = tuple(x...)
```

请注意，`@code_typed` 和你的朋友给你的始终是特例化的代码，即使 Julia 通常不会特例化该方法调用。如果要查看更改参数类型时是否生成特例化，则需要检查 method internals，即是否 `Base.specializations(@which f(...))` 包含相关参数的特例化。

35.8 将函数拆分为多个定义

将一个函数写成许多小的定义能让编译器直接调用最适合的代码，甚至能够直接将它内联。

这是一个真的该被写成许多小的定义的**复合函数**的例子：

```
using LinearAlgebra

function mynorm(A)
    if isa(A, Vector)
        return sqrt(real(dot(A,A)))
    elseif isa(A, Matrix)
        return maximum(svdvals(A))
    else
        error("mynorm: invalid argument")
    end
end
```

这可以更简洁有效地写成：

```
mynorm(x::Vector) = sqrt(real(dot(x, x)))
mynorm(A::Matrix) = maximum(svdvals(A))
```

然而，应该注意的是，编译器会十分高效地优化掉编写得如同 `mynorm` 例子的代码中的死分支。

35.9 编写「类型稳定的」函数

如果可能，确保函数总是返回相同类型的值是有好处的。考虑以下定义：

```
pos(x) = x < 0 ? 0 : x
```

虽然这看起来挺合法的，但问题是 `0` 是一个（`Int` 类型的）整数而 `x` 可能是任何类型。于是，根据 `x` 的值，此函数可能返回两种类型中任何一种的值。这种行为是允许的，并且在某些情况下可能是合乎需要的。但它可以很容易地以如下方式修复：

```
pos(x) = x < 0 ? zero(x) : x
```

还有 `oneunit` 函数，以及更通用的 `oftype(x, y)` 函数，它返回被转换为 `x` 的类型的 `y`。

35.10 避免更改变量类型

类似的「类型稳定性」问题存在于在函数内重复使用的变量：

```
function foo()
    x = 1
    for i = 1:10
        x /= rand()
    end
    return x
end
```

局部变量 `x` 一开始是整数，在一次循环迭代后变为浮点数（`/` 运算符的结果）。这使得编译器更难优化循环体。有几种可能的解决方法：

- 使用 `x = 1.0` 初始化 `x`
- 显式声明 `x` 的类型：`x::Float64 = 1`
- 使用 `x = oneunit(Float64)` 进行显式的类型转换
- 使用第一个循环迭代初始化，即 `x = 1 / rand()`，接着循环 `for i = 2:10`

35.11 分离核心函数（又称为函数屏障）

许多函数遵循这一模式：先执行一些设置工作，再通过多次迭代来执行核心计算。如果可行，将这些核心计算放在单独的函数中是个好主意。例如，以下做作的函数返回一个数组，其类型是随机选择的。

```
julia> function strange_twos(n)
           a = Vector{rand(Bool) ? Int64 : Float64}(undef, n)
           for i = 1:n
               a[i] = 2
           end
           return a
       end;

julia> strange_twos(3)
3-element Vector{Int64}:
 2
 2
 2
```

它应该这么写：

```
julia> function fill_twos!(a)
           for i = eachindex(a)
               a[i] = 2
           end
       end;

julia> function strange_twos(n)
```

```
           a = Vector{rand(Bool) ? Int64 : Float64}(undef, n)
           fill_twos!(a)
           return a
       end;

julia> strange_twos(3)
3-element Vector{Int64}:
 2
 2
 2
```

Julia 的编译器会在函数边界处针对参数类型特化代码，因此在原始的实现中循环期间无法得知 a 的类型（因为它是随即选择的）。于是，第二个版本通常更快，因为对于不同类型的 a，内层循环都可被重新编译为 fill_twos! 的一部分。

第二种形式通常是更好的风格，并且可以带来更多的代码的重复利用。

这个模式在 Julia Base 的几个地方中有使用。相关的例子，请参阅 abstractarray.jl 中的 vcat 和 hcat，或者 fill! 函数，我们可使用该函数而不是编写自己的 fill_twos!。

诸如 strange_twos 的函数会在处理具有不确定类型的数据时出现，例如从可能包含整数、浮点数、字符串或其它内容的输入文件中加载的数据。

35.12 具有值作为参数的类型

比方说你想创建一个每个维度大小都是 3 的 N 维数组。这种数组可以这样创建：

```
julia> A = fill(5.0, (3, 3))
3×3 Matrix{Float64}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

这个方法工作得很好：编译器可以识别出来 A 是一个 Array{Float64,2} 因为它知道填充值 (5.0::Float64) 的类型和维度 ((3, 3)::NTuple{2,Int}).

但是现在打比方说你想写一个函数，在任何一个维度下，它都创建一个 3×3×... 的数组；你可能会心动地写下一个函数

```
julia> function array3(fillval, N)
           fill(fillval, ntuple(d->3, N))
       end
array3 (generic function with 1 method)

julia> array3(5.0, 2)
3×3 Matrix{Float64}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

这确实有用，但是（你可以自己使用 @code_warntype array3(5.0, 2) 来验证）问题是输出地类型不能被推断出来：参数 N 是一个 Int 类型的**值**，而且类型推断不会（也不能）提前预测它的值。这意

味着使用这个函数的结果的代码在每次获取 A 时都不得不保守地检查其类型；这样的代码将会是非常缓慢的。

现在，解决此类问题的一种很好的方法是使用 函数障碍技巧。但是，在某些情况下，你可能希望完全消除类型不稳定性。在这种情况下，一种方法是将维度作为参数传递，例如通过 Val{T}()（参见值类型）：

```
julia> function array3(fillval, ::Val{N}) where N
           fill(fillval, ntuple(d->3, Val(N)))
       end
array3 (generic function with 1 method)

julia> array3(5.0, Val(2))
3×3 Matrix{Float64}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

Julia 有一个特别版本的 ntuple，它接受一个 Val{::Int} 实例作为第二个参数；通过将 N 作为类型参数传递，你可以让编译器知道它的"值"。因此，这个版本的 array3 允许编译器预测返回类型。

然而，使用这些技术可能非常微妙。例如，如果你从这样的函数中调用 array3 将没有任何帮助：

```
function call_array3(fillval, n)
    A = array3(fillval, Val(n))
end
```

在这里，你又一次创造了同样的问题：编译器无法猜测 n 是什么，所以它不知道 Val(n) 的 类型。在许多情况下，尝试使用 Val 但不正确地使用很容易使性能变差。（只有在有效地将 Val 与函数障碍技巧结合起来的情况下，为了使内核函数更有效，才应使用上述代码。）

一个正确使用 Val 的例子是这样的：

```
function filter3(A::AbstractArray{T,N}) where {T,N}
    kernel = array3(1, Val(N))
    filter(A, kernel)
end
```

在此示例中，N 作为参数传递，因此编译器知道其"值"。本质上，Val(T) 仅在 T 是硬编码/i 字面量 (Val(3)) 或已在类型域中指定时才起作用。

35.13 滥用多重派发的危险（也就是更多关于以值作为参数的类型）

一旦一个人理解了多重派发，就会有一个倾向，即过度使用它并尝试将其用于所有事情。例如，您可能会想象使用它来存储信息，例如

```
struct Car{Make, Model}
    year::Int
    ...more fields...
end
```

然后派发到像 `Car{:Honda,:Accord}(year, args...)` 的对象上。

当存在以下任一情况，这可能是值得做的：

- 你需要对每个 `Car` 进行 CPU 密集型处理，如果你在编译时知道 `Make` 和 `Model`，并且将使用的不同 `Make` 或 `Model` 的总数不太大，则效率会大大提高。
- 你需要处理相同类型的 `Car` 的同类列表，因此可以将它们全部存储在一个数组 `{Car{:Honda,:Accord},N}` 中。

当后者成立时，处理此类同型数组的函数可以高效地特例化：Julia 预先知道每个元素的类型（容器中的所有对象都具有相同的具体类型），因此当函数被编译时，Julia 可以"查找"正确的方法调用（不需要在运行时检查），从而产生有效的代码来处理整个列表。

当这些都不成立时，你很可能不会获得任何好处；更糟糕的是，由此产生的"类型组合爆炸"将适得其反。如果 `items[i+1]` 与 `item[i]` 的类型不同，Julia 必须在运行时查找类型，在方法表中搜索适当的方法，决定（通过类型交集）哪一个匹配，确定它是否已经被 JIT 编译（如果没有，则执行），然后进行调用。本质上，你是在要求完整的类型系统和 JIT 编译机制在你自己的代码中基本上执行相当于 switch 语句或字典查找的操作。

在邮件列表中 可以找到一些运行时基准比较 (1) 类型派发、(2) 字典查找和 (3) "switch"语句

也许比运行时影响更糟糕的是编译期影响：Julia 将为每个不同的 `Car{Make, Model}` 编译专门的函数；如果你有成百上千个这样的类型，那么每个接受这样一个对象作为参数的函数（从你可能自己编写的自定义 get_year 函数，到 Julia Base 中的通用 `push!` 函数）都将成百上千个为它编译了的变体。这些都会增加编译代码缓存的大小、内部方法列表的长度等。对值作为参数的过度热情很容易浪费大量资源。

35.14 沿列按内存顺序访问数组

Julia 中的多维数组以列主序存储。这意味着数组一次堆叠一列。这可使用 `vec` 函数或语法 `[:]` 来验证，如下所示（请注意，数组的顺序是 `[1 3 2 4]`，而不是 `[1 2 3 4]`）：

```
julia> x = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> x[:]
4-element Vector{Int64}:
 1
 3
 2
 4
```

这种对数组进行排序的约定在许多语言中都很常见，例如 Fortran、Matlab 和 R（仅举几例）。列优先排序的替代方法是行优先排序，在其它语言中，这是 C 和 Python (numpy) 采用的约定。在遍历数组时，记住数组的顺序会对性能产生显着的影响。要记住的一个经验法则是，对于列优先数组，第一个索引变化最快。本质上，这意味着如果最内层的循环索引是第一个出现在切片表达式中的，则循环会更快。请记住，使用 `:` 索引数组是一个隐式循环，它迭代访问特定维度内的所有元素；例如，提取列比提取行更快。

考虑以下人为示例。假设我们想编写一个接收 `Vector` 并返回方阵 `Matrix` 的函数，所返回方阵的行或列都用输入向量的副本填充。并假设用这些副本填充的是行还是列并不重要（也许可以很容易地相应调整剩余代码）。我们至少可以想到四种方式（除了建议的调用内置函数 `repeat`）：

```
function copy_cols(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for i = inds
        out[:, i] = x
    end
    return out
end

function copy_rows(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for i = inds
        out[i, :] = x
    end
    return out
end

function copy_col_row(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for col = inds, row = inds
        out[row, col] = x[row]
    end
    return out
end

function copy_row_col(x::Vector{T}) where T
    inds = axes(x, 1)
    out = similar(Array{T}, inds, inds)
    for row = inds, col = inds
        out[row, col] = x[col]
    end
    return out
end
```

现在，我们使用相同的 10000 乘 1 的随机输入向量来对这些函数计时。

```
julia> x = randn(10000);

julia> fmt(f) = println(rpad(string(f)*": ", 14, ' '), @elapsed f(x))

julia> map(fmt, [copy_cols, copy_rows, copy_col_row, copy_row_col]);
copy_cols:    0.331706323
copy_rows:    1.799009911
copy_col_row: 0.415630047
copy_row_col: 1.721531501
```

请注意，copy_cols 比 copy_rows 快得多。这与预料的一致，因为 copy_cols 尊重 Matrix 基于列的内存布局。另外，copy_col_row 比 copy_row_col 快得多，因为它遵循我们的经验法则，即切片表达式中出现的第一个元素应该与最内层循环耦合。

35.15 输出预分配

如果函数返回 Array 或其它复杂类型，则可能需要分配内存。不幸的是，内存分配及其反面垃圾收集通常是很大的瓶颈。

有时，你可以通过预分配输出结果来避免在每个函数调用上分配内存的需要。作为一个简单的例子，比较

```
julia> function xinc(x)
           return [x, x+1, x+2]
       end;

julia> function loopinc()
           y = 0
           for i = 1:10^7
               ret = xinc(i)
               y += ret[2]
           end
           return y
       end;
```

和

```
julia> function xinc!(ret::AbstractVector{T}, x::T) where T
           ret[1] = x
           ret[2] = x+1
           ret[3] = x+2
           nothing
       end;

julia> function loopinc_prealloc()
           ret = Vector{Int}(undef, 3)
           y = 0
           for i = 1:10^7
               xinc!(ret, i)
               y += ret[2]
           end
           return y
       end;
```

计时结果：

```
julia> @time loopinc()
  0.529894 seconds (40.00 M allocations: 1.490 GiB, 12.14% gc time)
50000015000000

julia> @time loopinc_prealloc()
  0.030850 seconds (6 allocations: 288 bytes)
50000015000000
```

预分配还有其它优点，例如允许调用者在算法中控制「输出」类型。在上述例子中，我们如果需要，可以传递 SubArray 而不是 Array。

极端情况下，预分配可能会使你的代码更丑陋，所以可能需要做性能测试和一些判断。但是，对于「向量化」(逐元素）函数，方便的语法 x .= f.(y) 可用于具有融合循环的 in-place 操作且无需临时数组（请参阅向量化函数的点语法）。

35.16 点语法：融合向量化操作

Julia 有特殊的点语法，它可以将任何标量函数转换为「向量化」函数调用，将任何运算符转换为「向量化」运算符，其具有的特殊性质是嵌套「点调用」是融合的：它们在语法层级被组合为单个循环，无需分配临时数组。如果你使用 .= 和类似的赋值运算符，则结果也可以 in-place 存储在预分配的数组（参见上文）。

在线性代数的上下文中，这意味着即使诸如 vector + vector 和 vector * scalar 之类的运算，使用 vector .+ vector 和 vector .* scalar 来替代也可能是有利的，因为生成的循环可与周围的计算融合。例如，考虑两个函数：

```
julia> f(x) = 3x.^2 + 4x + 7x.^3;

julia> fdot(x) = @. 3x^2 + 4x + 7x^3; # equivalent to 3 .* x.^2 .+ 4 .* x .+ 7 .* x.^3
```

f 和 fdot 都做相同的计算。但是，fdot（在 @. 宏的帮助下定义）在作用于数组时明显更快：

```
julia> x = rand(10^6);

julia> @time f(x);
  0.019049 seconds (16 allocations: 45.777 MiB, 18.59% gc time)

julia> @time fdot(x);
  0.002790 seconds (6 allocations: 7.630 MiB)

julia> @time f.(x);
  0.002626 seconds (8 allocations: 7.630 MiB)
```

That is, fdot(x) is ten times faster and allocates 1/6 the memory of f(x), because each * and + operation in f(x) allocates a new temporary array and executes in a separate loop. In this example f.(x) is as fast as fdot(x) but in many contexts it is more convenient to sprinkle some dots in your expressions than to define a separate function for each vectorized operation.

35.17 考虑对切片使用视图

在 Julia 中，像 array[1:5, :] 这样的数组“切片”表达式会创建该数据的副本（赋值的左侧除外，其中 array[1:5, :] = ... 原地对 array 的那一部分进行赋值）。如果你在切片上执行许多操作，这对性能有好处，因为使用较小的连续副本比索引原始数组更有效。另一方面，如果你只是对切片进行一些简单的操作，那么分配和复制操作的成本可能会很高。

另一种方法是创建数组的“视图”，它是一个数组对象（一个 SubArray），它实际上就地引用了原始数组的数据，而不进行复制。（如果你写入视图，它也会修改原始数组的数据。）这可以通过调用 view 对单个切片完成，或者更简单地通过将整个表达式或代码块放入 @views 在该表达式前面。例如：

```
julia> fcopy(x) = sum(x[2:end-1]);

julia> @views fview(x) = sum(x[2:end-1]);
```

```
julia> x = rand(10^6);

julia> @time fcopy(x);
  0.003051 seconds (3 allocations: 7.629 MB)

julia> @time fview(x);
  0.001020 seconds (1 allocation: 16 bytes)
```

请注意，该函数的 `fview` 版本提速了 3 倍且减少了内存分配。

35.18 复制数据不总是坏的

数组被连续地存储在内存中，这使其可被 CPU 向量化，并且会由于缓存减少内存访问。这与建议以列序优先方式访问数组的原因相同（请参见上文）。由于不按顺序访问内存，无规律的访问方式和不连续的视图可能会大大减慢数组上的计算速度。

在对无规律访问的数据进行重复访问之前，将其复制到连续的数组中可能能带来巨大的加速，正如下例所示。其中，矩阵和向量在相乘前会访问已被随机混洗的索引处的值。即使增加了复制和分配的成本，复制到普通数组中也能加快乘法运算速度。

```
julia> using Random

julia> A = randn(3000, 3000);

julia> x = randn(2000);

julia> inds = shuffle(1:3000)[1:2000];

julia> function iterated_neural_network(A, x, depth)
           for _ in 1:depth
               x .= max.(0, A * x)
           end
           argmax(x)
       end

julia> @time iterated_neural_network(view(A, inds, inds), x, 10)
  0.324903 seconds (12 allocations: 157.562 KiB)
1569

julia> @time iterated_neural_network(A[inds, inds], x, 10)
  0.054576 seconds (13 allocations: 30.671 MiB, 13.33% gc time)
1569
```

只要有足够的内存，将视图复制到数组的成本，就会被在连续数组上进行重复矩阵乘法所带来的加速所抵消。

35.19 使用 StaticArrays.jl 进行小型固定大小的向量/矩阵运算

如果您的应用程序涉及许多固定大小的小（`< 100` 个元素）数组（即在执行之前已知大小），那么你可能需要考虑使用 [StaticArrays.jl 包](https://github.com/JuliaArrays/StaticArrays.jl)。这个包允许你以一

种避免不必要的堆分配的方式来表示这样的数组，并允许编译器为数组的大小特例化代码，例如，通过完全展开向量操作（消除循环）并将元素存储在 CPU 寄存器中。

例如，如果你正在使用 2d 几何图形进行计算，你可能会使用 2-分量向量进行许多计算。通过使用 StaticArrays.jl 中的 `SVector` 类型，你可以在向量 `v` 和 `w` 上使用方便的向量符号和操作，例如 `norm(3v - w)`，同时允许编译器将代码展开到最小，计算等效于 `@inbounds hypot(3v[1]-w[1], 3v[2]-w[2])`。

35.20 避免 I/O 中的字符串插值

将数据写入到文件（或其他 I/O 设备）中时，生成额外的中间字符串会带来开销。请不要写成这样：

```
println(file, "$a $b")
```

请写成这样：

```
println(file, a, " ", b)
```

第一个版本的代码生成一个字符串，然后将其写入到文件中，而第二个版本直接将值写入到文件中。另请注意，在某些情况下，字符串插值可能更难阅读。请考虑：

```
println(file, "$(f(a))$(f(b))")
```

与：

```
println(file, f(a), f(b))
```

35.21 并发执行时优化网络 I/O

当并发地执行一个远程函数时：

```
using Distributed

responses = Vector{Any}(undef, nworkers())
@sync begin
    for (idx, pid) in enumerate(workers())
        @async responses[idx] = remotecall_fetch(foo, pid, args...)
    end
end
```

会快于：

```
using Distributed

refs = Vector{Any}(undef, nworkers())
for (idx, pid) in enumerate(workers())
    refs[idx] = @spawnat pid foo(args...)
end
responses = [fetch(r) for r in refs]
```

第一种方式导致每个 worker 一次网络往返，而第二种方式是两次网络调用：一次 @spawnat 一次fetch（甚至是 wait）。fetch 和wait 都是同步执行，会导致较差的性能。

35.22 修复过期警告

过期的函数在内部会执行查找，以便仅打印一次相关警告。这种额外查找可能会显著影响性能，因此应根据警告建议修复掉过期函数的所有使用。

35.23 小技巧

有一些小的注意事项可能会帮助改善循环性能。

- 避免使用不必要的数组。比如，使用 x+y+z 而不是 sum([x,y,z])。
- 对于复数 z，使用 abs2(z) 而不是 abs(z)^2。一般的，对于复数参数，用 abs2 代替abs。
- 对于直接截断的整除，使用 div(x,y) 而不是 trunc(x/y)，使用fld(x,y) 而不是 floor(x/y)，使用 cld(x,y) 而不是 ceil(x/y)。

35.24 性能标注

有时，你可以通过承诺某些程序性质来启用更好的优化。

- 使用 @inbounds 来取消表达式中的数组边界检查。使用前请再三确定，如果下标越界，可能会发生崩溃或潜在的故障。
- 使用 @fastmath 来允许对于实数是正确的、但是对于 IEEE 数字会导致差异的浮点数优化。使用时请多多小心，因为这可能改变数值结果。这对应于 clang 的 -ffast-math 选项。
- 在 for 循环前编写 @simd 来承诺迭代是相互独立且可以重新排序的。请注意，在许多情况下，Julia 可以在没有 @simd 宏的情况下自动向量化代码；只有在这种转换原本是非法的情况下才有用，包括允许浮点数重新结合和忽略相互依赖的内存访问（@simd ivdep）等情况。此外，在断言 @simd 时要十分小心，因为错误地标注一个具有相互依赖的迭代的循环可能导致意外结果。尤其要注意的是，某些 AbstractArray 子类型的 setindex! 本质上依赖于迭代顺序。**此功能是实验性的**，在 Julia 未来的版本中可能会更改或消失。

如果 Array 使用非常规索引，那么使用 1:n 索引到 AbstractArray 的常见习惯用法是不安全的，如果关闭边界检查，可能会导致段错误。请改用 LinearIndices(x) 或 eachindex(x)（另请参阅具有自定义索引的数组）。

> **Note**
>
> 虽然 @simd 需要直接放在最内层 for 循环前面，但 @inbounds 和 @fastmath 都可作用于单个表达式或在嵌套代码块中出现的所有表达式，例如，可使用 @inbounds begin 或 @inbounds for ...。

下面是一个具有 @inbounds 和 @simd 标记的例子（我们这里使用 @noinline 来防止因优化器过于智能而破坏我们的基准测试）：

```
@noinline function inner(x, y)
    s = zero(eltype(x))
    for i=eachindex(x)
        @inbounds s += x[i]*y[i]
    end
    return s
end

@noinline function innersimd(x, y)
    s = zero(eltype(x))
    @simd for i = eachindex(x)
        @inbounds s += x[i] * y[i]
    end
    return s
end

function timeit(n, reps)
    x = rand(Float32, n)
    y = rand(Float32, n)
    s = zero(Float64)
    time = @elapsed for j in 1:reps
        s += inner(x, y)
    end
    println("GFlop/sec        = ", 2n*reps / time*1E-9)
    time = @elapsed for j in 1:reps
        s += innersimd(x, y)
    end
    println("GFlop/sec (SIMD) = ", 2n*reps / time*1E-9)
end

timeit(1000, 1000)
```

在配备 2.4GHz Intel Core i5 处理器的计算机上，其结果为：

```
GFlop/sec        = 1.9467069505224963
GFlop/sec (SIMD) = 17.578554163920018
```

(`GFlop/sec` 用来测试性能，数值越大越好。)

下面是一个具有三种标记的例子。此程序首先计算一个一维数组的有限差分，然后计算结果的 L2 范数：

```
function init!(u::Vector)
    n = length(u)
    dx = 1.0 / (n-1)
    @fastmath @inbounds @simd for i in 1:n # 通过断言 `u` 是一个 `Vector`，我们可以假定它具有
    ↪  1-based 索引
        u[i] = sin(2pi*dx*i)
    end
end

function deriv!(u::Vector, du)
    n = length(u)
```

```
    dx = 1.0 / (n-1)
    @fastmath @inbounds du[1] = (u[2] - u[1]) / dx
    @fastmath @inbounds @simd for i in 2:n-1
        du[i] = (u[i+1] - u[i-1]) / (2*dx)
    end
    @fastmath @inbounds du[n] = (u[n] - u[n-1]) / dx
end

function mynorm(u::Vector)
    n = length(u)
    T = eltype(u)
    s = zero(T)
    @fastmath @inbounds @simd for i in 1:n
        s += u[i]^2
    end
    @fastmath @inbounds return sqrt(s)
end

function main()
    n = 2000
    u = Vector{Float64}(undef, n)
    init!(u)
    du = similar(u)

    deriv!(u, du)
    nu = mynorm(du)

    @time for i in 1:10^6
        deriv!(u, du)
        nu = mynorm(du)
    end

    println(nu)
end

main()
```

在配备 2.7 GHz Intel Core i7 处理器的计算机上，其结果为：

```
$ julia wave.jl;
  1.207814709 seconds
4.443986180758249

$ julia --math-mode=ieee wave.jl;
  4.487083643 seconds
4.443986180758249
```

在这里，选项 `--math-mode=ieee` 禁用 `@fastmath` 宏，好让我们可以比较结果。

在这种情况下，`@fastmath` 加速了大约 3.7 倍。这非常大——通常来说，加速会更小。（在这个特定的例子中，基准测试的工作集足够小，可以放在该处理器的 L1 缓存中，因此内存访问延迟不起作用，计算时间主要由 CPU 使用率决定。在许多现实世界的程序中，情况并非如此。）此外，在这种情况下，此优化不会改变计算结果——通常来说，结果会略有不同。在某些情况下，尤其是数值不稳定的算法，计算结果可能会差很多。

标注 @fastmath 会重新排列浮点数表达式，例如更改求值顺序，或者假设某些特殊情况（如 inf、nan）不出现。在这种情况中（以及在这个特定的计算机上），主要区别是函数 deriv 中的表达式 1 / (2*dx) 会被提升出循环（即在循环外计算），就像编写了 idx = 1 / (2*dx)，然后，在循环中，表达式 ... / (2*dx) 变为 ... * idx，后者计算起来快得多。当然，编译器实际上采用的优化以及由此产生的加速都在很大程度上取决于硬件。你可以使用 Julia 的 code_native 函数来检查所生成代码的更改。

请注意，@fastmath 也假设了在计算中不会出现 NaN，这可能导致意想不到的行为：

```
julia> f(x) = isnan(x);

julia> f(NaN)
true

julia> f_fast(x) = @fastmath isnan(x);

julia> f_fast(NaN)
false
```

35.25 将次正规数视为零

次正规数，以前称为 非正规数，在许多情况下都很有用，但会在某些硬件上造成性能损失。调用 set_zero_subnormals(true) 授予浮点运算权限，将次正规输入或输出视为零，这可能会提高某些硬件的性能。调用 set_zero_subnormals(false) 对次正规数强制执行严格的 IEEE 行为。

下面是一个示例，其中次正规数显着影响某些硬件的性能：

```
function timestep(b::Vector{T}, a::Vector{T}, Δt::T) where T
    @assert length(a)==length(b)
    n = length(b)
    b[1] = 1                            # Boundary condition
    for i=2:n-1
        b[i] = a[i] + (a[i-1] - T(2)*a[i] + a[i+1]) * Δt
    end
    b[n] = 0                            # Boundary condition
end

function heatflow(a::Vector{T}, nstep::Integer) where T
    b = similar(a)
    for t=1:div(nstep,2)                # Assume nstep is even
        timestep(b,a,T(0.1))
        timestep(a,b,T(0.1))
    end
end

heatflow(zeros(Float32,10),2)           # Force compilation
for trial=1:6
    a = zeros(Float32,1000)
    set_zero_subnormals(iseven(trial))  # Odd trials use strict IEEE arithmetic
    @time heatflow(a,1000)
end
```

它的输出类似于

```
0.002202 seconds (1 allocation: 4.063 KiB)
0.001502 seconds (1 allocation: 4.063 KiB)
0.002139 seconds (1 allocation: 4.063 KiB)
0.001454 seconds (1 allocation: 4.063 KiB)
0.002115 seconds (1 allocation: 4.063 KiB)
0.001455 seconds (1 allocation: 4.063 KiB)
```

注意，每个偶数迭代的速度明显更快。

这个例子产生了许多次正规数，因为 `a` 中的值变成了一个指数递减的曲线，随着时间的推移慢慢渐进趋于 0。

应谨慎使用将次正规数视为零，因为这样做会破坏某些等式，例如 `x-y == 0` 意味着 `x == y`：

```
julia> x = 3f-38; y = 2f-38;

julia> set_zero_subnormals(true); (x - y, x == y)
(0.0f0, false)

julia> set_zero_subnormals(false); (x - y, x == y)
(1.0000001f-38, false)
```

在某些应用程序中，将次正规数归零的另一种方法是加入一点点噪音。例如，不是用零初始化 `a`，而是用以下方法初始化它：

```
a = rand(Float32,1000) * 1.f-9
```

35.26 @code_warntype

宏 `@code_warntype`（或其函数变体 `code_warntype`）有时可以帮助诊断类型相关的问题。这是一个例子：

```
julia> @noinline pos(x) = x < 0 ? 0 : x;

julia> function f(x)
           y = pos(x)
           return sin(y*x + 1)
       end;

julia> @code_warntype f(3.2)
MethodInstance for f(::Float64)
  from f(x) @ Main REPL[9]:1
Arguments
  #self#::Core.Const(f)
  x::Float64
Locals
  y::Union{Float64, Int64}
Body::Float64
1 ─      (y = Main.pos(x))
│   %2 = (y * x)::Float64
│   %3 = (%2 + 1)::Float64
│   %4 = Main.sin(%3)::Float64
└──      return %4
```

理解 @code_warntype 的输出，就像理解它的同类工具 @code_lowered, @code_typed, @code_llvm 和 @code_native 一样需要一些练习。你的代码以在生成编译机器代码的过程中经过大量摘要的形式呈现。大多数表达式都由类型注释，由 ::T 表示（例如，其中 T 可能是 Float64）。@code_warntype 最大的特点就是非具体类型用红色显示；由于本文档是用 Markdown 编写的，没有颜色，所以本文档中红色文字用大写表示。

在顶部，该函数类型推导后的返回类型显示为 Body::Float64。下一行以 Julia 的 SSA IR 形式表示了 f 的主体。被数字标记的方块表示代码中（通过 goto）跳转的目标。查看主体，你会看到首先调用了 pos，其返回值经类型推导为 Union 类型 Union{Float64, Int64} 并以大写字母显示，因为它是非具体类型。这意味着我们无法根据输入类型知道 pos 的确切返回类型。但是，无论 y 是 Float64 还是 Int64，y*x 的结果都是 Float64。最终的结果是 f(x::Float64) 在其输出中不会是类型不稳定的，即使有些中间计算是类型不稳定的。

如何使用这些信息取决于你。显然，最好将 pos 修改为类型稳定的：如果这样做，f 中的所有变量都是具体的，其性能将是最佳的。但是，在某些情况下，这种短暂的类型不稳定性可能无关紧要：例如，如果 pos 从不单独使用，那么 f 的输出（对于 Float64 输入）是类型稳定的这一事实将保护之后的代码免受类型不稳定性的传播影响。这与类型不稳定性难以或不可能修复的情况密切相关。在这些情况下，上面的建议（例如，添加类型注释并/或分解函数）是你控制类型不稳定性的「损害」的最佳工具。另请注意，即使是 Julia Base 也有类型不稳定的函数。例如，函数 findfirst 如果找到键则返回数组索引，如果没有找到键则返回 nothing，这是明显的类型不稳定性。为了更易于找到可能很重要的类型不稳定性，包含 missing 或 nothing 的 Union 会用黄色着重显示，而不是用红色。

以下示例可以帮助你解释被标记为包含非叶类型的表达式：

- 函数体以 Body::Union{T1,T2}) 开头
  - 解释：函数具有不稳定返回类型
  - 建议：使返回值类型稳定，即使你必须对其进行类型注释
- invoke Main.g(%%x::Int64)::Union{Float64, Int64}
  - 解释：调用类型不稳定的函数 g。
  - 建议：修改该函数，或在必要时对其返回值进行类型注释
- invoke Base.getindex(%%x::Array{Any,1}, 1::Int64)::Any
  - 解释：访问缺乏类型信息的数组的元素
  - 建议：使用具有更佳定义的类型的数组，或在必要时对访问的单个元素进行类型注释
- Base.getfield(%%x, :(:data))::Array{Float64,N} where N
  - 解释：获取一个非叶子类型的字段。在这种情况下，x 的类型，比如说 ArrayContainer，有一个字段 data::Array{T}。但是 Array 也需要维度 N 作为具体类型。
  - 建议：使用类似于 Array{T,3} 或 Array{T,N} 的具体类型，其中的 N 现在是 ArrayContainer 的参数

35.27 被捕获变量的性能

请考虑以下定义内部函数的示例：

```
function abmult(r::Int)
    if r < 0
        r = -r
    end
    f = x -> x * r
    return f
end
```

函数 abmult 返回一个函数 f，它将其参数乘以 r 的绝对值。赋值给 f 的函数称为「闭包」。内部函数还被语言用于 do 代码块和生成器表达式。

这种代码风格为语言带来了性能挑战。解析器在将其转换为较低级别的指令时，基本上通过将内部函数提取到单独的代码块来重新组织上述代码。「被捕获的」变量，比如 r，被内部函数共享，且包含它们的作用域会被提取到内部函数和外部函数皆可访问的堆分配「box」中，这是因为语言指定内部作用域中的 r 必须与外部作用域中的 r 相同，就算在外部作用域（或另一个内部函数）修改 r 后也需如此。

前一段的讨论中提到了「解析器」，也就是，包含 abmult 的模块被首次加载时发生的编译前期，而不是首次调用它的编译后期。解析器不「知道」Int 是固定类型，也不知道语句 r = -r 将一个 Int 转换为另一个 Int。类型推断的魔力在编译后期生效。

因此，解析器不知道 r 具有固定类型（Int）。一旦内部函数被创建，r 的值也不会改变（因此也不需要 box）。因此，解析器向包含具有抽象类型（比如 Any）的对象的 box 发出代码，这对于每次出现的 r 都需要运行时类型分派。这可以通过在上述函数中使用 @code_warntype 来验证。装箱和运行时的类型分派都有可能导致性能损失。

如果捕获的变量用于代码的性能关键部分，那么以下提示有助于确保它们的使用具有高效性。首先，如果已经知道被捕获的变量不会改变类型，则可以使用类型注释来显式声明类型（在变量上，而不是在右侧）：

```
function abmult2(r0::Int)
    r::Int = r0
    if r < 0
        r = -r
    end
    f = x -> x * r
    return f
end
```

类型注释部分恢复由于捕获而导致的丢失性能，因为解析器可以将具体类型与 box 中的对象相关联。更进一步，如果被捕获的变量不再需要 box（因为它不会在闭包创建后被重新分配），就可以用 let 代码块表示，如下所示。

```
function abmult3(r::Int)
    if r < 0
        r = -r
    end
    f = let r = r
            x -> x * r
    end
    return f
end
```

The `let` block creates a new variable `r` whose scope is only the inner function. The second technique recovers full language performance in the presence of captured variables. Note that this is a rapidly evolving aspect of the compiler, and it is likely that future releases will not require this degree of programmer annotation to attain performance. In the mean time, some user-contributed packages like FastClosures automate the insertion of `let` statements as in `abmult3`.

35.28 Multithreading and linear algebra

This section applies to multithreaded Julia code which, in each thread, performs linear algebra operations. Indeed, these linear algebra operations involve BLAS / LAPACK calls, which are themselves multithreaded. In this case, one must ensure that cores aren't oversubscribed due to the two different types of multithreading.

Julia compiles and uses its own copy of OpenBLAS for linear algebra, whose number of threads is controlled by the environment variable `OPENBLAS_NUM_THREADS`. It can either be set as a command line option when launching Julia, or modified during the Julia session with `BLAS.set_num_threads(N)` (the submodule `BLAS` is exported by `using LinearAlgebra`). Its current value can be accessed with `BLAS.get_num_threads()`.

When the user does not specify anything, Julia tries to choose a reasonable value for the number of OpenBLAS threads (e.g. based on the platform, the Julia version, etc.). However, it is generally recommended to check and set the value manually. The OpenBLAS behavior is as follows:

- If `OPENBLAS_NUM_THREADS=1`, OpenBLAS uses the calling Julia thread(s), i.e. it "lives in" the Julia thread that runs the computation.
- If `OPENBLAS_NUM_THREADS=N>1`, OpenBLAS creates and manages its own pool of threads (`N` in total). There is just one OpenBLAS thread pool shared among all Julia threads.

When you start Julia in multithreaded mode with `JULIA_NUM_THREADS=X`, it is generally recommended to set `OPENBLAS_NUM_THREADS=1`. Given the behavior described above, increasing the number of BLAS threads to `N>1` can very easily lead to worse performance, in particular when `N<<X`. However this is just a rule of thumb, and the best way to set each number of threads is to experiment on your specific application.

35.29 Alternative linear algebra backends

As an alternative to OpenBLAS, there exist several other backends that can help with linear algebra performance. Prominent examples include MKL.jl and AppleAccelerate.jl.

These are external packages, so we will not discuss them in detail here. Please refer to their respective documentations (especially because they have different behaviors than OpenBLAS with respect to multithreading).

Chapter 36

工作流程建议

这里是高效使用 Julia 的一些建议。

36.1 基于 REPL 的工作流程

正如在 Julia REPL 中演示的那样，Julia 的 REPL 为高效的交互式工作流程提供了丰富的功能。这里是一些可能进一步提升你在命令行下的体验的建议。The-Julia-REPL

一个基本的编辑器 / REPL 工作流程

最基本的 Julia 工作流程是将一个文本编辑器配合 `julia` 的命令行使用。一般会包含下面一些步骤：

- **把还在开发中的代码放到一个临时的模块中**。新建一个文件，例如 `Tmp.jl`，并放到模块中。

```
module Tmp
export say_hello

say_hello() = println("Hello!")

# your other definitions here

end
```

- **把测试代码放到另一个文件中**。新建另一个文件，例如 `tst.jl`，开头为

```
include("Tmp.jl")
import .Tmp
# using .Tmp # we can use `using` to bring the exported symbols in `Tmp` into our namespace

Tmp.say_hello()
# say_hello()

# your other test code here
```

  并把测试作为 `Tmp` 的内容。或者，你可以把测试文件的内容打包到一个模块中，例如

```julia
module Tst
    include("Tmp.jl")
    import .Tmp
    #using .Tmp

    Tmp.say_hello()
    # say_hello()

    # your other test code here
end
```

优点是你的测试代码现在包含在一个模块中，并且不会在 Main 的全局作用域中引入新定义，这样更加整洁。

- 使用 include("tst.jl") 来在 Julia REPL 中 include tst.jl 文件。
- **打肥皂，冲洗，重复**。(译者注：此为英语幽默，被称为“洗发算法”在 julia REPL 中摸索不同的想法，把好的想法存入 tst.jl。要在 tst.jl 被更改后执行它，只需再次 include 它。

36.2 基于浏览器的工作流程

There are a few ways to interact with Julia in a browser:

- Using Pluto notebooks through Pluto.jl
- Using Jupyter notebooks through IJulia.jl

36.3 基于 Revise 的工作流程

无论你是在 REPL 还是在 IJulia，你通常可以通过 Revise优化你的开发经历。通常情况，无论何时启动 Julia，就请按照Revise 文档中的说明配置好 Revise。一旦配置好，Revise 将跟踪任何加载模块中的文件变化。和任何用 includet 加载到 REPL 的文件 (但不包括普通的 include); 然后你就可以编辑这些文件，并且更改会在不重新启动 julia 会话的情况下生效。标准工作流与上面基于 REPL 的工作流类似，区别如下：

1. 把你的代码放到一个在你的加载路径里的模块中。要这样做有很多种方法，通常推荐以下两种选择：

   - 对于长期的项目，使用PkgTemplates:

     ```julia
     using PkgTemplates
     t = Template()
     t("MyPkg")
     ```

     这将在 .julia/dev 目录中创建一个空白包"MyPkg"。请注意，通过它的 Template 构造器，PkgTemplates 允许控制许多不同的选项。
     在下面的第 2 步中，编辑 MyPkg/src/MyPkg.jl 以更改源代码，并编辑 MyPkg/test/runtests.jl 以进行测试。
   - 对于“一次性”项目，您可以通过在临时目录（例如 /tmp）中进行工作来避免任何清理需求。
     切换到临时目录并启动 Julia，然后执行以下操作：

```
pkg> generate MyPkg            # type ] to enter pkg mode
julia> push!(LOAD_PATH, pwd())   # hit backspace to exit pkg mode
```

如果你重新启动 Julia 会话，则必须重新发出修改 `LOAD_PATH` 的命令。

在下面的第 2 步中， 编辑 `MyPkg/src/MyPkg.jl` 以更改源代码，并创建你选择的任何测试文件。

2. 构建你自己的包

在加载任何代码之前，确保 Revise 已经被启用：`using Revise` 或者按照教程设置自动加载。

然后切换到包含测试文件（假设文件为"runtests.jl"）的目录下，并：

```
julia> using MyPkg

julia> include("runtests.jl")
```

你可以修改在 MyPkg 文件夹中的代码然后用 `include("runtests.jl")` 重新跑一遍测试。通常，你可能需要重新启动 Julia 会话来使得这些变化生效（受一些 限制)。

Chapter 37

代码风格指南

接下来的部分将介绍如何写出具有 Julia 风格的代码。当然，这些规则并不是绝对的，它们只是一些建议，以便更好地帮助你熟悉这门语言，以及在不同的代码设计中做出选择。

37.1 缩进

每个缩进级别使用 4 个空格。

37.2 写函数，而不是仅仅写脚本

一开始解决问题的时候，直接从最外层一步步写代码的确很便捷，但你应该尽早地将代码组织成函数。函数有更强的复用性和可测试性，并且能更清楚地让人知道哪些步骤做完了，以及每一步骤的输入输出分别是什么。此外，由于 Julia 编译器特殊的工作方式，写在函数中的代码往往要比最外层的代码运行地快得多。

此外值得一提的是，函数应当接受参数，而不是直接使用全局变量进行操作（`pi` 等常数除外）。

37.3 类型不要写得过于具体

代码应该写得尽可能通用。例如，下面这段代码:

```
Complex{Float64}(x)
```

更好的写法是写成下面的通用函数：

```
complex(float(x))
```

第二个版本会把 `x` 转换成合适的类型，而不是某个写死的类型。

这种代码风格与函数的参数尤其相关。例如，当一个参数可以是任何整型时，不要将它的类型声明为 `Int` 或 `Int32`，而要使用抽象类型（abstract type）`Integer` 来表示。事实上，除非确实需要将其与其它的方法定义区分开，很多情况下你可以干脆完全省略掉参数的类型，因为如果你的操作中有不支持某种参数类型的操作的话，反正都会抛出 `MethodError` 的。这也称作 鸭子类型)。

例如，考虑这样的一个叫做 addone 的函数，其返回值为它的参数加 1：

```
addone(x::Int) = x + 1                 # works only for Int
addone(x::Integer) = x + oneunit(x)    # any integer type
addone(x::Number) = x + oneunit(x)     # any numeric type
addone(x) = x + oneunit(x)             # any type supporting + and oneunit
```

最后一种定义可以处理所有支持 `oneunit` （返回和 `x` 相同类型的 `1`，以避免不需要的类型提升（type promotion））以及 `+` 函数的类型。这里的关键点在于，**只定义通用的** `addone(x) = x + oneunit(x)` **并不会**带来性能上的损失，因为 Julia 会在需要的时候自动编译特定的版本。比如说，当第一次调用 `addone(12)` 时，Julia 会自动编译一个特定的 `addone` 函数，它接受一个 `x::Int` 的参数，并把调用的 `oneunit` 替换为内连的值 `1`。因此，上述的前三种 `addone` 的定义对于第四种来说是完全多余的。

37.4 让调用者处理多余的参数多样性

如下的代码：

```
function foo(x, y)
    x = Int(x); y = Int(y)
    ...
end
foo(x, y)
```

请写成这样：

```
function foo(x::Int, y::Int)
    ...
end
foo(Int(x), Int(y))
```

这种风格更好，因为 `foo` 函数其实不需要接受所有类型的数，而只需要接受 `Int`。

这里的关键在于，如果一个函数需要处理的是整数，强制让调用者来决定非整数如何被转换（比如说向下还是向上取整）会更好。同时，把类型声明得具体一些的话可以为以后的方法定义留有更多的空间。

37.5 在修改其参数的函数名称后加 !

如下的代码：

```
function double(a::AbstractArray{<:Number})
    for i = firstindex(a):lastindex(a)
        a[i] *= 2
    end
    return a
end
```

请写成这样：

```
function double!(a::AbstractArray{<:Number})
    for i = firstindex(a):lastindex(a)
        a[i] *= 2
    end
    return a
end
```

Julia 的 Base 模块中的函数都遵循了这种规范，且包含很多例子：有的函数同时有拷贝和修改的形式（比如 sort 和 sort!），还有一些只有修改（比如 push!，pop! 和 splice!）。为了方便起见，这类函数通常也会把修改后的数组作为返回值。

Functions related to IO or making use of random number generators (RNG) are notable exceptions: Since these functions almost invariably must mutate the IO or RNG, functions ending with ! are used to signify a mutation *other* than mutating the IO or advancing the RNG state. For example, `rand(x)` mutates the RNG, whereas `rand!(x)` mutates both the RNG and x; similarly, `read(io)` mutates io, whereas `read!(io, x)` mutates both arguments.

37.6 避免使用奇怪的 Union 类型

使用 `Union{Function,AbstractString}` 这样的类型的时候通常意味着设计还不够清晰。

37.7 避免复杂的容器类型

像下面这样构造数组通常没有什么好处：

```
a = Vector{Union{Int,AbstractString,Tuple,Array}}(undef, n)
```

这种情况下，`Vector{Any}(undef, n)` 更合适些。此外，相比将所有可能的类型都打包在一起，直接在使用时标注具体的数据类型（比如：`a[i]::Int`）对编译器来说更有用。

37.8 方法导出优先于直接字段访问

Idiomatic Julia code should generally treat a module's exported methods as the interface to its types. An object's fields are generally considered implementation details and user code should only access them directly if this is stated to be the API. This has several benefits:

- Package developers are freer to change the implementation without breaking user code.
- Methods can be passed to higher-order constructs like map (e.g. `map(imag, zs)`) rather than `[z.im for z in zs]`).
- Methods can be defined on abstract types.
- Methods can describe a conceptual operation that can be shared across disparate types (e.g. `real(z)` works on Complex numbers or Quaternions).

Julia's dispatch system encourages this style because `play(x::MyType)` only defines the `play` method on that particular type, leaving other types to have their own implementation.

Similarly, non-exported functions are typically internal and subject to change, unless the documentations states otherwise. Names sometimes are given a _ prefix (or suffix) to further suggest that something is "internal" or an implementation-detail, but it is not a rule.

Counter-examples to this rule include NamedTuple, RegexMatch, StatStruct.

37.9 Use naming conventions consistent with Julia base/

- modules and type names use capitalization and camel case: `module SparseArrays`, `struct UnitRange`.
- functions are lowercase (`maximum`, `convert`) and, when readable, with multiple words squashed together (`isequal`, `haskey`). When necessary, use underscores as word separators. Underscores are also used to indicate a combination of concepts (`remotecall_fetch` as a more efficient implementation of `fetch(remotecall(...))`) or as modifiers.
- functions mutating at least one of their arguments end in `!`.
- conciseness is valued, but avoid abbreviation (`indexin` rather than `indxin`) as it becomes difficult to remember whether and how particular words are abbreviated.

包开发人员可以更自由地更改实现而不会破坏用户代码。

-
- 方法可以传递给高阶结构，如 `map`（例如 `map(imag, zs)`）而不是 `[z.im for z in zs]`）。
- 方法可以定义在抽象类型上。
- 方法可以描述可以在不同类型之间共享的概念操作（例如 `real(z)` 适用于复数或四元数）。

Julia 的调度系统鼓励这种风格，因为 `play(x::MyType)` 只在该特定类型上定义了 `play` 方法，其它类型有自己的实现。

同样，除非文档另有说明，否则非导出函数通常是内部的并且可能会发生变化。名称有时会被赋予一个 _ 前缀（或后缀）以进一步暗示某些是“内部”或实现细节，但这不是规则。

该规则的反例包括 NamedTuple、RegexMatch、StatStruct。

37.10 使用和 Julia base/ 文件夹中的代码一致的命名习惯

- module 和 type 的名字使用大写开头的驼峰命名法：`module SparseArrays`，`struct UnitRange`。
- 函数名使用小写字母，且当可读时可以将多个单词拼在一起。必要的时候，可以使用下划线作为单词分隔符。下划线也被用于指明概念的组合（比如 `remotecall_fetch` 作为 `fetch(remotecall(...))` 的一个更高效的实现）或者变化。
- 至少改变其中一个参数的函数名称以 `!` 结尾。
- 虽然简洁性很重要，但避免使用缩写（用 `indexin` 而不是 `indxin`），因为这会让记住单词有没有被缩写或如何被缩写变得十分困难。

如果一个函数名需要多个单词，请考虑这个函数是否代表了超过一个概念，是不是分成几个更小的部分更好。

37.11 使用与 Julia Base 中的函数类似的参数顺序

一般来说，Base 库使用以下的函数参数顺序（如适用）:

1. **函数参数**. 函数的第一个参数可以接受 `Function` 类型，以便使用 `do` blocks 来传递多行匿名函数。
2. **I/O stream**. 函数的第一个参数可以接受 `IO` 对象，以便将函数传递给 `sprint` 之类的函数，例如 `sprint(show, x)`。
3. **在输入参数的内容会被更改的情况下**. 比如，在 `fill!(x, v)` 中，`x` 是要被修改的对象，所以放在要被插入 `x` 中的值前面。
4. **Type**. 把类型作为参数传入函数通常意味着返回值也会是同样的类型。在 `parse(Int, "1")` 中，类型在需要解析的字符串之前。还有很多类似的将类型作为函数第一个参数的例子，但是同时也需要注意到例如 `read(io, String)` 这样的函数中，会把 `IO` 参数放在类型的更前面，这样还是保持着这里描述的顺序。
5. **在输入参数的内容不会被更改的情况下**. 比如在 `fill!(x, v)` 中的**不**被修改的 `v`，会放在 `x` 之后传入。
6. **Key**. 对于关联集合来说，指的是键值对的键。对于其它有索引的集合来说，指的是索引。
7. **Value**. 对于关联集合来说，指的是键值对的值。像`fill!(x, v)`这样的情况，是 `v`。
8. **Everything else**. 任何的其它参数。
9. **Varargs**. 指的是在函数调用时可以被无限列在后面的参数。比如在 `Matrix{T}(uninitialized, dims)` 中，维数（dims）可以作为 `Tuple` 被传入（如 `Matrix{T}(uninitialized, (1,2))`），也可以作为可变参数（`Vararg`，如 `Matrix{T}(uninitialized, 1, 2)`。
10. **Keyword arguments**. 在 Julia 中，关键字参数本来就不得不定义在函数定义的最后，列在这里仅仅是为了完整性。

大多数函数并不会接受上述所有种类的参数，这些数字仅仅是表示当适用时的优先权。

当然，在一些情况下有例外。例如，`convert` 函数总是把类型作为第一个参数。`setindex!` 函数的值参数在索引参数之前，这样可以让索引作为可变参数传入。

设计 API 时，尽可能秉承着这种一般顺序会让函数的使用者有一种更一致的体验。

37.12 不要过度使用 try-catch

比起依赖于捕获错误，更好的是避免错误。

37.13 不要给条件语句加括号

Julia 不要求在 `if` 和 `while` 后的条件两边加括号。使用如下写法:

```
if a == b
```

而不是:

```
if (a == b)
```

37.14 不要过度使用 ...

拼接函数参数是会上瘾的。请用简单的 `[a; b]` 来代替 `[a..., b...]`，因为前者已经是被拼接的数组了。`collect(a)` 也比 `[a...]` 更好，但因为 `a` 已经是一个可迭代的变量了，通常不把它转换成数组就直接使用甚至更好。

37.15 不要使用不必要的静态参数

如下的函数签名：

```
foo(x::T) where {T<:Real} = ...
```

应当被写作：

```
foo(x::Real) = ...
```

尤其是当 `T` 没有被用在函数体中时格外有意义。即使 `T` 被用到了，通常也可以被替换为 `typeof(x)`，后者不会导致性能上的差别。注意这并不是针对静态参数的一般警告，而仅仅是针对那些不必要的情况。

同样需要注意的是，容器类型在函数调用中可能明确地需要类型参数。详情参见避免使用带抽象容器的字段。

37.16 避免判断变量是实例还是类型的混乱

如下的一组定义容易令人困惑：

```
foo(::Type{MyType}) = ...
foo(::MyType) = foo(MyType)
```

请决定问题里的概念应当是 `MyType` 还是 `MyType()`，然后坚持使用其一。

首选风格应是默认使用实例，只有在需要解决某些问题时才添加涉及 `Type{MyType}` 的方法。

如果一个类型实际上是个枚举，它应该被定义成一个单一的类型（理想的情况是不可变结构或原始类型），把枚举值作为它的实例。构造器和转换器可以检查那些值是否有效。这种设计比把枚举做成抽象类型，并把“值”做成子类型来得更受推崇。

37.17 不要过度使用宏

请注意有的宏实际上可以被写成一个函数。

在宏内部调用 `eval` 是一个特别危险的警告标志，它意味着这个宏仅在被最外层调用时起作用。如果这样的宏被写成函数，它会自然地访问得到它所需要的运行时值。

37.18 不要把不安全的操作暴露在接口层

如果你有一个使用本地指针的类型：

```
mutable struct NativeType
    p::Ptr{UInt8}
    ...
end
```

不要定义类似如下的函数：

```
getindex(x::NativeType, i) = unsafe_load(x.p, i)
```

这里的问题在于，这个类型的用户可能会在意识不到这个操作不安全的情况下写出 `x[i]`，然后容易遇到内存错误。

在这样的函数中，可以加上对操作的检查来确保安全，或者可以在名字的某处加上 `unsafe` 来警告调用者。

37.19 不要重载基础容器类型的方法

有时可能会想要写这样的定义：

```
show(io::IO, v::Vector{MyType}) = ...
```

这样可以提供对特定的某种新元素类型的向量的自定义显示。这种做法虽然很诱人，但应当被避免。这里的问题在于用户会想着一个像 `Vector()` 这样熟知的类型以某种方式表现，但过度自定义的行为会让使用变得更难。

37.20 避免类型盗版

"Type piracy" refers to the practice of extending or redefining methods in Base or other packages on types that you have not defined. In extreme cases, you can crash Julia (e.g. if your method extension or redefinition causes invalid input to be passed to a `ccall`). Type piracy can complicate reasoning about code, and may introduce incompatibilities that are hard to predict and diagnose.

例如，你也许想在一个模块中定义符号上的乘法：

```
module A
import Base.*
*(x::Symbol, y::Symbol) = Symbol(x,y)
end
```

这里的问题时现在其它用到 `Base.*` 的模块同样会看到这个定义。由于 `Symbol` 是定义在 `Base` 里再被其它模块所使用的，这可能不可预料地改变无关代码的行为。这里有几种替代的方式，包括使用一个不同的函数名称，或是把 `Symbol` 给包在另一个你自己定义的类型中。

有时候，耦合的包可能会使用类型盗版，以此来从定义分隔特性，尤其是当那些包是一些合作的作者设计的时候，且那些定义是可重用的时候。例如，一个包可能提供一些对处理色彩有用的类型，另一个包可能为那些类型定义色彩空间之间转换的方法。再举一个例子，一个包可能是一些 C 代码的简易包装，另一个包可能就"盗版"来实现一些更高级别的、对 Julia 友好的 API。

37.21 注意类型相等

通常会用 isa 和 <: 来对类型进行测试，而不会用到 ==。检测类型的相等通常只对和一个已知的具体类型比较有意义（例如 T == Float64），或者你**真的真的**知道自己在做什么。

37.22 不要写 x->f(x)

因为调用高阶函数时经常会用到匿名函数，很容易认为这是合理甚至必要的。但任何函数都可以被直接传递，并不需要被“包”在一个匿名函数中。比如 map(x->f(x), a) 应当被写成 map(f, a)。

37.23 尽可能避免使用浮点数作为通用代码的字面量

当写处理数字，且可以处理多种不同数字类型的参数的通用代码时，请使用对参数影响（通过类型提升）尽可能少的类型的字面量。

例如，

```
julia> f(x) = 2.0 * x
f (generic function with 1 method)

julia> f(1//2)
1.0

julia> f(1/2)
1.0

julia> f(1)
2.0
```

而应当被写作：

```
julia> g(x) = 2 * x
g (generic function with 1 method)

julia> g(1//2)
1//1

julia> g(1/2)
1.0

julia> g(1)
2
```

如你所见，使用了 Int 字面量的第二个版本保留了输入参数的类型，而第一个版本没有。这是因为例如 promote_type(Int, Float64) == Float64，且做乘法时会需要类型提升。类似地，Rational 字面量比 Float64 字面量对类型有着更小的破坏性，但比 Int 大。

```
julia> h(x) = 2//1 * x
h (generic function with 1 method)

julia> h(1//2)
1//1
```

```
julia> h(1/2)
1.0

julia> h(1)
2//1
```

所以，可能时尽量使用 `Int` 字面量，对非整数字面量使用 `Rational{Int}`，这样可以让代码变得更容易使用。

Chapter 38

常见问题

38.1 概述

Julia 的名字来源于某人或某事物吗？

不。

为什么不把 Matlab/Python/R 或者其他语言的代码编译为 Julia 呢？

由于大多数人对其他动态语言的语法很熟悉，而且已经在这些动态语言中编写了很多代码，人们也许会问：为什么我们不直接设计以 Julia 为后端的 Matlab 或是 Python 前端（也就是把其他代码"转译"到 Julia）？这样既能获得 Julia 的高性能，也能避免程序员花费精力来学一门新的语言。这是一个简单的解决方案，不是吗？

总的来说，我们这样做是因为 **Julia 编译器没有什么特别之处**：我们使用的是普通的编译器（LLVM），这里面没有什么其他语言开发者所不知道的"独家秘方"。诚然，Julia 编译器在许多地方比其他动态语言的编译器更简单（比如 PyPy 和 LuaJIT）。Julia 的性能优势几乎完全来自其前端：它的语义学使得高质量的 Julia 程序 能够给予编译器更多的机会来产生高效的代码和内存结构。如果你尝试将 Matlab 或 Python 代码编译为 Julia，我们的编译器会被其语义学限制而不能产生相对现有编译器更好的代码（甚至更差）。语义学的关键角色也正是一些现存的 Python 编译器（像 Numba 和 Pythran）仅仅尝试优化语言的一小部分（比如 Numpy 的矢量与标量运算）的原因，而这些部分已经至少在相同的语义学上与我们做的一样好。致力于这些项目的人员难以置信得聪明并且已经取得了令人惊叹的成就，但为被解释而设计的语言加装编译器是十分困难的。

Julia 的优势在于好的性能不止被限制在一小部分的内置类型与操作，用户能够写出使用任意自定义类型的高级泛型代码，同时也能保证很高的运行与内存效率。在如 Python 一般的语言中，类型没有给编译器提供太多的信息来达成这样的目的，当你试图像使用 Julia 前端一样使用这些语言时，你会遇到困难。

出于类似的原因，自动翻译为 Julia 的代码一般来说会是可读性差、缓慢且违反习惯的代码。这些代码不是从其他语言迁移到 Julia 的好的起点。

另一方面，语言**可迁移性**是极其有用的：我们会在一些时候想要将其他语言的高质量代码迁移到 Julia 中（也可能相反）。这一工作的最佳实践不是翻译器，而是使用简单的跨语言调用。我们对此有许多工作，从内置的 `ccall`（来调用 C 和 Fortran 模块）到JuliaInterop 包来链接 Julia 和 Python、Matlab、C++ 以及更多语言。

38.2 公共 API

How does Julia define its public API?

Julia Base and standard library functionality described in the the documentation that is not marked as unstable (e.g. experimental and internal) is covered by SemVer. Functions, types, and constants are not part of the public API if they are not included in the documentation, *even if they have docstrings*.

There is a useful undocumented function/type/constant. Can I use it?

Updating Julia may break your code if you use non-public API. If the code is self-contained, it may be a good idea to copy it into your project. If you want to rely on a complex non-public API, especially when using it from a stable package, it is a good idea to open an issue or pull request to start a discussion for turning it into a public API. However, we do not discourage the attempt to create packages that expose stable public interfaces while relying on non-public implementation details of Julia and buffering the differences across different Julia versions.

The documentation is not accurate enough. Can I rely on the existing behavior?

Please open an issue or pull request to start a discussion for turning the existing behavior into a public API.

38.3 Sessions and the REPL

Julia 如何定义其公共 API？

对于 `julia` 版本的 SemVer，唯一稳定的接口是 Julia 的 `Base` 和 文档 中的标准库接口中且未标记为不稳定（例如，实验性的和内部性的）的部分。如果函数、类型和常量未包含在文档中，则它们不是公共 API 的一部分，即使它们具有文档。

有一个有用的非官方的函数/类型/常量。我可以使用它吗？

如果您使用非公共 API，更新 Julia 可能会使你的代码失效。如果代码是自洽的，最好将其复制到你的项目中。如果你想依赖一个复杂的非公共 API，尤其是从稳定的包中使用它时，最好打开发起 issue 或 pull request 开始讨论将其转换为公共 API。尽管你可以在下游自己开发一个包来封装这个内部实现，并且屏蔽不同的 `Julia` 版本差异，但我们并不鼓励这样做。

文档不够准确。我可以依赖现有的行为吗？

请发起一个 issue 或 pull request 开始讨论将现有行为转换为公共 API。

38.4 会话和 REPL

如何从内存中删除某个对象？

Julia 没有类似于 MATLAB 的 `clear` 函数，某个名称一旦定义在 Julia 的会话中（准确地说，在 `Main` 模块中），它就会一直存在下去。

如果关心内存使用情况，你可以用消耗较少内存的对象替换原对象。例如，如果 `A` 是一个你不再需要的千兆字节大小的数组，你可以使用 `A = nothing` 来释放内存。下次垃圾收集器运行时会释放内存；您可以使用 `GC.gc()` 强制执行此操作。此外，尝试使用 `A` 可能会导致错误，因为大多数方法都没有在类型 `Nothing` 上定义。

如何在会话中修改某个类型的声明？

也许你定义了某个类型，后来发现需要向其中增加一个新的域。如果在 REPL 中尝试这样做，会得到一个错误：

```
ERROR: invalid redefinition of constant MyType
```

模块 Main 中的类型不能重新定义。

尽管这在开发新代码时会造成不便，但是这个问题仍然有一个不错的解决办法：可以用重新定义的模块替换原有的模块，把所有新代码封装在一个模块里，这样就能重新定义类型和常量了。虽说不能将类型名称导入到 Main 模块中再去重新定义，但是可以用模块名来改变作用范围。换言之，开发时的工作流可能类似这样：

```
include("mynewcode.jl") # this defines a module MyModule
obj1 = MyModule.ObjConstructor(a, b)
obj2 = MyModule.somefunction(obj1)
# Got an error. Change something in "mynewcode.jl"
include("mynewcode.jl") # reload the module
obj1 = MyModule.ObjConstructor(a, b) # old objects are no longer valid, must reconstruct
obj2 = MyModule.somefunction(obj1) # this time it worked!
obj3 = MyModule.someotherfunction(obj2, c)
...
```

38.5 脚本

该如何检查当前文件是否正在以主脚本运行？

当一个文件通过使用 julia file.jl 来当做主脚本运行时，有人也希望激活另外的功能例如命令行参数操作。确定文件是以这个方式运行的一个方法是检查 abspath(PROGRAM_FILE) == @__FILE__ 是不是 true。

However, it is recommended to not write files that double as a script and as an importable library. If one needs functionality both available as a library and a script, it is better to write is as a library, then import the functionality into a distinct script.

怎样在脚本中捕获 CTRL-C ？

通过 julia file.jl 方式运行的 Julia 脚本，在你尝试按 CTRL-C (SIGINT) 中止它时，并不会抛出 InterruptException。如果希望在脚本终止之后运行一些代码，请使用 atexit，注意：脚本的中止不一定是由 CTRL-C 导致的。另外你也可以通过 julia -e 'include(popfirst!(ARGS))' file.jl 命令运行脚本，然后可以通过 try 捕获 InterruptException。Note that with this strategy PROGRAM_FILE will not be set.

怎样通过 #!/usr/bin/env 传递参数给 julia？

Passing options to julia in a so-called shebang line, as in #!/usr/bin/env julia --startup-file=no, will not work on many platforms (BSD, macOS, Linux) where the kernel, unlike the shell, does not split arguments at space characters. The option env -S, which splits a single argument string into multiple arguments at spaces, similar to a shell, offers a simple workaround:

```
#!/usr/bin/env -S julia --color=yes --startup-file=no
@show ARGS  # put any Julia code here
```

Note

Option env -S appeared in FreeBSD 6.0 (2005), macOS Sierra (2016) and GNU/Linux coreutils 8.30 (2018).

Why doesn't `run` support * or pipes for scripting external programs?

Julia's `run` function launches external programs *directly*, without invoking an operating-system shell (unlike the `system("...")` function in other languages like Python, R, or C). That means that `run` does not perform wildcard expansion of * ("globbing"), nor does it interpret shell pipelines like | or >.

You can still do globbing and pipelines using Julia features, however. For example, the built-in `pipeline` function allows you to chain external programs and files, similar to shell pipes, and the Glob.jl package implements POSIX-compatible globbing.

You can, of course, run programs through the shell by explicitly passing a shell and a command string to `run`, e.g. `` run(`sh -c "ls > files.txt"`) `` to use the Unix Bourne shell, but you should generally prefer pure-Julia scripting like `` run(pipeline(`ls`, "files.txt")) ``. The reason why we avoid the shell by default is that shelling out sucks: launching processes via the shell is slow, fragile to quoting of special characters, has poor error handling, and is problematic for portability. (The Python developers came to a similar conclusion.)

38.6 Variables and Assignments

Why am I getting `UndefVarError` from a simple loop?

You might have something like:

```
x = 0
while x < 10
    x += 1
end
```

and notice that it works fine in an interactive environment (like the Julia REPL), but gives ``UndefVarError: `x` not defined`` when you try to run it in script or other file. What is going on is that Julia generally requires you to **be explicit about assigning to global variables in a local scope**.

Here, `x` is a global variable, `while` defines a local scope, and `x += 1` is an assignment to a global in that local scope.

As mentioned above, Julia (version 1.5 or later) allows you to omit the `global` keyword for code in the REPL (and many other interactive environments), to simplify exploration (e.g. copy-pasting code from a function to run interactively). However, once you move to code in files, Julia requires a more disciplined approach to global variables. You have least three options:

1. Put the code into a function (so that `x` is a *local* variable in a function). In general, it is good software engineering to use functions rather than global scripts (search online for "why global variables bad" to see many explanations). In Julia, global variables are also slow.

2. Wrap the code in a `let` block. (This makes x a local variable within the `let ... end` statement, again eliminating the need for `global`).

3. Explicitly mark x as `global` inside the local scope before assigning to it, e.g. write `global x += 1`.

More explanation can be found in the manual section on soft scope.

38.7 函数

向函数传递了参数 x，在函数中做了修改，但是在函数外变量 x 的值还是没有变。为什么？

假设函数被如此调用：

```
julia> x = 10
10

julia> function change_value!(y)
           y = 17
       end
change_value! (generic function with 1 method)

julia> change_value!(x)
17

julia> x # x is unchanged!
10
```

在 Julia 中，通过将 x 作为参数传递给函数，不能改变变量 x 的绑定。在上例中，调用 `change_value!(x)` 时，y 是一个新建变量，初始时与 x 的值绑定，即 `10`。然后 y 与常量 `17` 重新绑定，此时变量外作用域中的 x 并没有变动。

假设 x 被绑定至 `Array` 类型 (也有可能是其他 可变的类型)。在函数中，你无法将 x 与 Array 解绑，但是你可以改变其内容。

```
julia> x = [1,2,3]
3-element Vector{Int64}:
 1
 2
 3

julia> function change_array!(A)
           A[1] = 5
       end
change_array! (generic function with 1 method)

julia> change_array!(x)
5

julia> x
3-element Vector{Int64}:
 5
 2
 3
```

这里我们新建了一个函数 chang_array!，它把 5 赋值给传入的数组（在调用处与 x 绑定，在函数中与 A 绑定）的第一个元素。注意，在函数调用之后，x 依旧与同一个数组绑定，但是数组的内容变化了：变量 A 和 x 是不同的绑定，引用同一个可变的 Array 对象。

函数内部能否使用 using 或 import？

不可以，不能在函数内部使用 using 或 import 语句。如果你希望导入一个模块，但只在特定的一个或一组函数中使用它的符号，有以下两种方式：

1. 使用 import：

```
import Foo
function bar(...)
    # ... refer to Foo symbols via Foo.baz ...
end
```

   这会加载 Foo 模块，同时定义一个变量 Foo 引用该模块，但并不会将其他任何符号从该模块中导入当前的命名空间。Foo 等符号可以由限定的名称 Foo.bar 等引用。

2. 将函数封装到模块中：

```
module Bar
export bar
using Foo
function bar(...)
    # ... refer to Foo.baz as simply baz ....
end
end
using Bar
```

   这会从 Foo 中导入所有符号，但仅限于 Bar 模块内。

运算符 ... 有何作用？

... 运算符的两个用法：slurping 和 splatting

很多 Julia 的新手会对运算符 ... 的用法感到困惑。让 ... 用法如此困惑的部分原因是根据上下文它有两种不同的含义。

... 在函数定义中将多个参数组合成一个参数

在函数定义的上下文中，... 运算符用来将多个不同的参数组合成单个参数。... 运算符的这种将多个不同参数组合成单个参数的用法称为 slurping：

```
julia> function printargs(args...)
           println(typeof(args))
           for (i, arg) in enumerate(args)
               println("Arg #$i = $arg")
           end
       end
printargs (generic function with 1 method)

julia> printargs(1, 2, 3)
```

```
Tuple{Int64, Int64, Int64}
Arg #1 = 1
Arg #2 = 2
Arg #3 = 3
```

如果 Julia 是一个使用 ASCII 字符更加自由的语言的话，slurping 运算符可能会写作 <-... 而非...。

... 在函数调用中将一个参数分解成多个不同参数

与在定义函数时表示将多个不同参数组合成一个参数的... 运算符用法相对，当用在函数调用的上下文中... 运算符也用来将单个的函数参数分成多个不同的参数。... 函数的这个用法叫做 splatting：

```
julia> function threeargs(a, b, c)
           println("a = $a::$(typeof(a))")
           println("b = $b::$(typeof(b))")
           println("c = $c::$(typeof(c))")
       end
threeargs (generic function with 1 method)

julia> x = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> threeargs(x...)
a = 1::Int64
b = 2::Int64
c = 3::Int64
```

如果 Julia 是一个使用 ASCII 字符更加自由的语言的话，splatting 运算符可能会写作...-> 而非...。

赋值语句的返回值是什么？

= 运算符始终返回右侧的值，所以：

```
julia> function threeint()
           x::Int = 3.0
           x # returns variable x
       end
threeint (generic function with 1 method)

julia> function threefloat()
           x::Int = 3.0 # returns 3.0
       end
threefloat (generic function with 1 method)

julia> threeint()
3

julia> threefloat()
3.0
```

相似地：

```
julia> function twothreetup()
           x, y = [2, 3] # assigns 2 to x and 3 to y
           x, y # returns a tuple
       end
twothreetup (generic function with 1 method)

julia> function twothreearr()
           x, y = [2, 3] # returns an array
       end
twothreearr (generic function with 1 method)

julia> twothreetup()
(2, 3)

julia> twothreearr()
2-element Vector{Int64}:
 2
 3
```

38.8 类型，类型声明和构造函数

何谓“类型稳定”？

这意味着输出的类型可以由输入的类型预测出来。特别地，这意味着输出的类型不会因输入的值的不同而变化。以下代码不是类型稳定的：

```
julia> function unstable(flag::Bool)
           if flag
               return 1
           else
               return 1.0
           end
       end
unstable (generic function with 1 method)
```

根据参数值的不同，该函数可能返回 `Int` 或 `Float64`。由于 Julia 无法在编译期预测该函数的返回值类型，任何使用该函数的计算都需要考虑这两种可能的返回类型，这样难以生成高效的机器码。

为何 Julia 对某个看似合理的操作返回 `DomainError`？

某些运算在数学上有意义，但会产生错误：

```
julia> sqrt(-2.0)
ERROR: DomainError with -2.0:
sqrt was called with a negative real argument but will only return a complex result if called with
↪  a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

这一行为是为了保证类型稳定而带来的不便。对于 sqrt，许多用户会希望 sqrt(2.0) 产生一个实数，如果得到了复数 1.4142135623730951 + 0.0im 则会不高兴。也可以编写 sqrt 函数，只有当传递一个负数时才切换到复值输出，但结果将不是类型稳定的，而且 sqrt 函数的性能会很差。

在这样那样的情况下，若你想得到希望的结果，你可以选择一个输入类型，它可以使根据你的想法接受一个输出类型，从而结果可以这样表示：

```
julia> sqrt(-2.0+0im)
0.0 + 1.4142135623730951im
```

怎样限制或计算类型参数？

参数类型 的参数可以包含类型或比特值，并且类型本身选择如何使用这些参数。例如，Array{Float64, 2} 由类型 Float64 参数化以表示其元素类型，并通过整数值 2 来表示其维度数。在定义自己的参数类型时，可以使用子类型约束来声明某个参数必须是某个抽象类型的子类型 (<:) 或以前的类型参数。但是，没有专用的语法来声明参数必须是给定类型的值—也就是说，例如，你不能在 struct 定义中直接声明一个维度参数 isa Int。同样，你不能对类型参数进行计算（包括简单的加法或减法）。相反，这些类型的约束和关系可以通过在类型的 构造函数 中计算和强制执行的附加类型参数来表达。

例如，考虑

```
struct ConstrainedType{T,N,N+1} # NOTE: INVALID SYNTAX
    A::Array{T,N}
    B::Array{T,N+1}
end
```

其中，用户希望强制第三个类型参数始终是第二个参数加一。这可以使用显式类型参数来实现，该参数由 内部构造函数方法（可以与其他检查结合使用）进行检查：

```
struct ConstrainedType{T,N,M}
    A::Array{T,N}
    B::Array{T,M}
    function ConstrainedType(A::Array{T,N}, B::Array{T,M}) where {T,N,M}
        N + 1 == M || throw(ArgumentError("second argument should have one more axis" ))
        new{T,N,M}(A, B)
    end
end
```

这种检查通常是无成本的，因为编译器可以省略对有效具体类型的检查。如果还计算了第二个参数，则提供执行此计算的 外部构造函数方法 可能更好：

```
ConstrainedType(A) = ConstrainedType(A, compute_B(A))
```

为什么 Julia 使用机器算法进行整数运算？

Julia 使用机器算法进行整数计算。这意味着 Int 的范围是有界的，值在范围的两端循环，也就是说整数的加法，减法和乘法会出现上溢或者下溢，导致出现某些从开始就令人不安的结果：

```
julia> x = typemax(Int)
9223372036854775807
```

```
julia> y = x+1
-9223372036854775808

julia> z = -y
-9223372036854775808

julia> 2*z
0
```

无疑，这与数学上的整数的行为很不一样，并且你会想对于高阶编程语言来说把这个暴露给用户难称完美。然而，对于效率优先和透明度优先的数值计算来说，其他的备选方案可谓更糟。

一个备选方案是去检查每个整数运算是否溢出，如果溢出则将结果提升到更大的整数类型比如Int128或者BigInt。不幸的是，这会给所有的整数操作（比如让循环计数器自增）带来巨大的额外开销—这需要生成代码去在算法指令后进行运行溢出检测，并生成分支去处理潜在的溢出。更糟糕的是，这会让涉及整数的所有运算变得类型不稳定。如同上面提到的，对于高效生成高效的代码类型稳定很重要。如果不指望整数运算的结果是整数，就无法想 C 和 Fortran 编译器一样生成快速简单的代码。

这个方法有个变体可以避免类型不稳定的出现，这个变体是将类型 Int 和BigInt合并成单个混合整数类型，当结果不再满足机器整数的大小时会内部自动切换表示。虽然表面上在 Julia 代码层面解决了类型不稳定，但是这个只是通过将所有的困难硬塞给实现混合整数类型的 C 代码而掩盖了这个问题。这个方法可能有用，甚至在很多情况下速度很快，但是它有很多缺点。一个缺点是整数和整数数组的内存上的表示不再与 C、Fortran 和其他使用原生机器整数的怨言所使用的自然表示一样。所以，为了与那些语言协作，我们无论如何最终都需要引入原生整数类型。任何整数的无界表示都不会占用固定的比特数，所以无法使用固定大小的槽来内联地存储在数组中—大的整数值通常需要单独的堆分配的存储。并且无论使用的混合整数实现多么智能，总会存在性能陷阱—无法预期的性能下降的情况。复杂的表示，与 C 和 Fortran 协作能力的缺乏，无法在不使用另外的堆存储的情况下表示整数数组，和无法预测的性能特性让即使是最智能化的混合整数实现对于高性能数值计算来说也是个很差的选择。

除了使用混合整数和提升到 BigInt，另一个备选方案是使用饱和整数算法，此时最大整数值加一个数时值保持不变，最小整数值减一个数时也是同样的。这就是 Matlab™ 的做法：

```
>> int64(9223372036854775807)

ans =

  9223372036854775807

>> int64(9223372036854775807) + 1

ans =

  9223372036854775807

>> int64(-9223372036854775808)

ans =

 -9223372036854775808

>> int64(-9223372036854775808) - 1
```

```
ans =

 -9223372036854775808
```

乍一看，这个似乎足够合理，因为 9223372036854775807 比-9223372036854775808 更接近于 9223372036854775808 并且整数还是以固定大小的自然方式表示的，这与 C 和 Fortran 相兼容。但是饱和整数算法是很有问题的。首先最明显的问题是这并不是机器整数算法的工作方式，所以实现饱和整数算法需要生成指令，在每个机器整数运算后检查上溢或者下溢并正确地讲这些结果用`typemin(Int)`或者`typemax(Int)`取代。单单这个就将整数运算从单语句的快速的指令扩展成六个指令，还可能包括分支。哎呦喂 ~~ 但是还有更糟的—饱和整数算法并不满足结合律。考虑下列的 Matlab 计算：

```
>> n = int64(2)^62
4611686018427387904

>> n + (n - 1)
9223372036854775807

>> (n + n) - 1
9223372036854775806
```

这就让写很多基础整数算法变得困难因为很多常用技术都是基于有溢出的机器加法是满足结合律这一事实的。考虑一下在 Julia 中求整数值 `lo` 和 `hi` 之间的中点值，使用表达式 `(lo + hi) >>> 1`：

```
julia> n = 2^62
4611686018427387904

julia> (n + 2n) >>> 1
6917529027641081856
```

看到了吗？没有任何问题。那就是 `2^62` 和 `2^63` 之间的正确地中点值，虽然 `n + 2n` 的值是 -4611686018427387904。现在使用 Matlab 试一下：

```
>> (n + 2*n)/2

ans =

  4611686018427387904
```

哎呦喂。在 Matlab 中添加 `>>>` 运算符没有任何作用，因为在将 `n` 与 `2n` 相加时已经破坏了能计算出正确地中点值的必要信息，已经出现饱和。

没有结合性不但对于不能依靠像这样的技术的程序员是不幸的，并且让几乎所有的希望优化整数算法的编译器铩羽而归。例如，因为 Julia 中的整数使用平常的机器整数算法，LLVM 就可以自由地激进地优化像 `f(k) = 5k-1` 这样的简单地小函数。这个函数的机器码如下所示：

```
julia> code_native(f, Tuple{Int})
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
```

```
Source line: 1
  leaq  -1(%rdi,%rdi,4), %rax
  popq  %rbp
  retq
  nopl  (%rax,%rax)
```

这个函数的实际函数体只是一个简单地 `leap` 指令，可以立马计算整数乘法与加法。当 `f` 内联在其他函数中的时候这个更加有益：

```
julia> function g(k, n)
           for i = 1:n
               k = f(k)
           end
           return k
       end
g (generic function with 1 methods)

julia> code_native(g, Tuple{Int,Int})
  .text
Filename: none
  pushq %rbp
  movq  %rsp, %rbp
Source line: 2
  testq %rsi, %rsi
  jle L26
  nopl  (%rax)
Source line: 3
L16:
  leaq  -1(%rdi,%rdi,4), %rdi
Source line: 2
  decq  %rsi
  jne L16
Source line: 5
L26:
  movq  %rdi, %rax
  popq  %rbp
  retq
  nop
```

因为 `f` 的调用内联化，循环体就只是简单地 `leap` 指令。接着，考虑一下如果循环迭代的次数固定的时候会发生什么：

```
julia> function g(k)
           for i = 1:10
               k = f(k)
           end
           return k
       end
g (generic function with 2 methods)

julia> code_native(g,(Int,))
  .text
Filename: none
```

```
  pushq %rbp
  movq  %rsp, %rbp
Source line: 3
  imulq $9765625, %rdi, %rax    # imm = 0x9502F9
  addq  $-2441406, %rax         # imm = 0xFFDABF42
Source line: 5
  popq  %rbp
  retq
  nopw  %cs:(%rax,%rax)
```

因为编译器知道整数加法和乘法是满足结合律的并且乘法可以在加法上使用分配律—两者在饱和算法中都不成立—所以编译器就可以把整个循环优化到只有一个乘法和一个加法。饱和算法完全无法使用这种优化，因为在每个循环迭代中结合律和分配律都会失效导致不同的失效位置会得到不同的结果。编译器可以展开循环，但是不能代数上将多个操作简化到更少的等效操作。

让整数算术沉默地溢出的最合理替代方法是在任何地方进行检查算术，在加法、减法和乘法溢出时引发错误，产生不正确的值。在这篇博文中，Dan Luu 对此进行了分析，并发现这种方法在理论上应该具有的微不足道的成本，但由于编译器（LLVM 和 GCC）没有优雅地围绕添加的溢出检查进行优化，它最终会产生大量成本。如果这在未来有所改善，我们可以考虑在 Julia 中默认使用检查整数算法，但现在，我们必须忍受可能会溢出这一现状。

同时，可以通过使用SaferIntegers.jl等外部库来实现溢出安全的整数运算。请注意，如前所述，使用这些库会显着增加使用已检查整数类型的代码的执行时间。但是，对于有限的使用，这远比将其用于所有整数运算时的问题要小得多。你可以在 此处 中关注讨论的状态。

在远程执行中 `UndefVarError` 的可能原因有哪些？

如同这个错误表述的，远程结点上的 `UndefVarError` 的直接原因是变量名的绑定并不存在。让我们探索一下一些可能的原因。

```
julia> module Foo
           foo() = remotecall_fetch(x->x, 2, "Hello")
       end

julia> Foo.foo()
ERROR: On worker 2:
UndefVarError: `Foo` not defined
Stacktrace:
[...]
```

闭包 `x->x` 中有 `Foo` 的引用，因为 `Foo` 在节点 2 上不存在，所以 `UndefVarError` 被扔出。

在模块中而非 `Main` 中的全局变量不会在远程节点上按值序列化。只传递了一个引用。新建全局绑定的函数（除了 `Main` 中）可能会导致之后扔出 `UndefVarError`。

```
julia> @everywhere module Foo
           function foo()
               global gvar = "Hello"
               remotecall_fetch(()->gvar, 2)
           end
       end

julia> Foo.foo()
```

```
ERROR: On worker 2:
UndefVarError: `gvar` not defined
Stacktrace:
[...]
```

在上面的例子中，`@everywhere module Foo` 在所有节点上定义了 `Foo`。但是调用 `Foo.foo()` 在本地节点上新建了新的全局绑定 `gvar`，但是节点 2 中并没有找到这个绑定，这会导致 `UndefVarError` 错误。

注意着并不适用于在模块 `Main` 下新建的全局变量。模块 `Main` 下的全局变量会被序列化并且在远程节点的 `Main` 下新建新的绑定。

```
julia> gvar_self = "Node1"
"Node1"

julia> remotecall_fetch(()->gvar_self, 2)
"Node1"

julia> remotecall_fetch(varinfo, 2)
name          size summary
––––––––––––––––––––––––––
Base               Module
Core               Module
Main               Module
gvar_self 13 bytes String
```

这并不适用于函数或者结构体声明。但是绑定到全局变量的匿名函数被序列化，如下例所示。

```
julia> bar() = 1
bar (generic function with 1 method)

julia> remotecall_fetch(bar, 2)
ERROR: On worker 2:
UndefVarError: `#bar` not defined
[...]

julia> anon_bar  = ()->1
(::#21) (generic function with 1 method)

julia> remotecall_fetch(anon_bar, 2)
1
```

38.9 “method not matched”故障排除：参数类型不变性和 `MethodError`

Why doesn’t it work to declare `foo(bar::Vector{Real}) = 42` and then call `foo([1])`?

As you’ll see if you try this, the result is a `MethodError`:

```
julia> foo(x::Vector{Real}) = 42
foo (generic function with 1 method)

julia> foo([1])
ERROR: MethodError: no method matching foo(::Vector{Int64})
```

```
Closest candidates are:
  foo(!Matched::Vector{Real})
   @ Main none:1

Stacktrace:
[...]
```

This is because `Vector{Real}` is not a supertype of `Vector{Int}`! You can solve this problem with something like `foo(bar::Vector{T}) where {T<:Real}` (or the short form `foo(bar::Vector{<:Real})` if the static parameter T is not needed in the body of the function). The T is a wild card: you first specify that it must be a subtype of Real, then specify the function takes a Vector of with elements of that type.

This same issue goes for any composite type `Comp`, not just `Vector`. If `Comp` has a parameter declared of type Y, then another type `Comp2` with a parameter of type `X<:Y` is not a subtype of `Comp`. This is type-invariance (by contrast, Tuple is type-covariant in its parameters). See Parametric Composite Types for more explanation of these.

Why does Julia use * for string concatenation? Why not + or something else?

为什么声明 `foo(bar::Vector{Real}) = 42` 然后调用 `foo([1])` 不起作用？

如果你尝试了，结果就会看到 `MethodError`:

```
julia> foo(x::Vector{Real}) = 42
foo (generic function with 1 method)

julia> foo([1])
ERROR: MethodError: no method matching foo(::Vector{Int64})
Closest candidates are:
  foo(!Matched::Vector{Real}) at none:1
```

There are several differences between `using` and `import` (see the Modules section), but there is an important difference that may not seem intuitive at first glance, and on the surface (i.e. syntax-wise) it may seem very minor. When loading modules with `using`, you need to say `function Foo.bar(...` to extend module Foo's function bar with a new method, but with `import Foo.bar`, you only need to say `function bar(...` and it automatically extends module Foo's function bar.

同样的问题适用于任何复合类型 `Comp`，而不仅仅是 `Vector`。如果 `Comp` 有一个声明为 `Y` 类型的参数，那么另一个带有 `X<:Y` 类型参数的类型 `Comp2` 不是 `Comp` 的子类型。这是类型不变性（相比之下，元组在其参数中是类型协变的）。有关这些的更多解释，请参阅 参数复合类型。

为什么 Julia 使用 * 进行字符串拼接？而不是使用 + 或其他符号？

使用 + 的主要依据是：字符串拼接是不可交换的操作，而 + 通常是一个具有可交换性的操作符。Julia 社区也意识到其他语言使用了不同的操作符，一些用户也可能不熟悉 * 包含的特定代数性值。

注意：你也可以用 `string(...)` 来拼接字符串和其他能转换成字符串的值；类似的 `repeat` 函数可以用于替代用于重复字符串的 ^ 操作符。字符串插值语法在构造字符串时也很常用。

38.10 包和模块

"using" 和"import" 的区别是什么？

只有一个区别，并且在表面上（语法层面）这个区别看来很小。using 和 import 的区别是使用 using 时你需要写 function Foo.bar(.. 来用一个新方法来扩展模块 Foo 的函数 bar，但是使用 import Foo.bar 时，你只需要写 function bar(...，会自动扩展模块 Foo 的函数 bar。

这个区别足够重要以至于提供不同的语法的原因是你不希望意外地扩展一个你根本不知道其存在的函数，因为这很容易造成 bug。对于使用像字符串后者整数这样的常用类型的方法最有可能出现这个问题，因为你和其他模块都可能定义了方法来处理这样的常用类型。如果你使用 import，你会用你自己的新实现覆盖别的函数的 bar(s::AbstractString) 实现，这会导致做的事情天差地别（并且破坏模块 Foo 中其他的依赖于调用 bar 的函数的所有/大部分的将来的使用）。

38.11 空值与缺失值

在 Julia 中"null"，" 空" 或者" 缺失" 是怎么工作的?

不像其它很多语言（例如 C 和 Java），Julia 对象默认不能为"null"。当一个引用（变量，对象域，或者数组元素）没有被初始化，访问它会立即扔出一个错误。这种情况可以使用函数 isdefined 或者 isassigned 检测到。

一些函数只为了其副作用使用，并不需要返回一个值。在这些情况下，约定的是返回 nothing 这个值，这只是 Nothing 类型的一个单例对象。这是一个没有域的一般类型；除了这个约定之外没有任何特殊点，REPL 不会为它打印任何东西。有些语言结构不会有值，也产生 nothing，例如 if false; end。

对于类型 T 的值 x 只会有时存在的情况，Union{T,Nothing} 类型可以用作函数参数，对象域和数组元素的类型，与其他语言中的Nullable, Option or Maybe相等。如果值本身可以是 nothing(显然当 T 是 Any 时），Union{Some{T}, Nothing} 类型更加准确因为 x == nothing 表示值的缺失，x == Some(nothing) 表示与 nothing 相等的值的存在。something函数允许使用默认值的展开的 Some 对象，而非 nothing 参数。注意在使用 Union{T,Nothing} 参数或者域时编译器能够生成高效的代码。

在统计环境下表示缺失的数据（R 中的 NA 或者 SQL 中的 NULL）请使用 missing 对象。请参照缺失值章节来获取详细信息。

在某些语言中，空元组 (()) 被认为是" 没有 "的规范形式。但是，在 julia 中，最好将其视为恰好包含零个值的常规元组。

空（或者" 底层"）类型，写作 Union{}（空的 union 类型）是没有值和子类型（除了自己）的类型。通常你没有必要用这个类型。

38.12 内存

为什么当 x 和 y 都是数组时 x += y 还会申请内存？

在 Julia 中，x += y 在语法分析中会用 x = x + y 代替。对于数组，结果就是它会申请一个新数组来存储结果，而非把结果存在 x 同一位置的内存上。If you prefer to mutate x, use x .+= y to update each element individually.

这个行为可能会让一些人吃惊，但是这个结果是经过深思熟虑的。主要原因是 Julia 中的不可变对象，这些对象一旦新建就不能改变他们的值。实际上，数字是不可变对象，语句 x = 5; x += 1 不会改变 5 的意义，改变的是与 x 绑定的值。对于不可变对象，改变其值的唯一方法是重新赋值。

为了稍微详细一点，考虑下列的函数：

```julia
function power_by_squaring(x, n::Int)
    ispow2(n) || error(" 此实现只适用于 2 的幂")
    while n >= 2
        x *= x
        n >>= 1
    end
    x
end
```

在 `x = 5; y = power_by_squaring(x, 4)` 调用后，你可以得到期望的结果 `x == 5 && y == 625`。然而，现在假设当 `*=` 与矩阵一起使用时会改变左边的值，这会有两个问题：

- 对于普通的方阵，`A = A*B` 不能在没有临时存储的情况下实现：`A[1,1]` 会被计算并且在被右边使用完之前存储在左边。
- 假设你愿意申请一个计算的临时存储（这会消除 `*=` 就地计算的大部分要点）；如果你利用了 `x` 的可变性，这个函数会对于可变和不可变的输入有不同的行为。特别地，对于不可变的 `x`，在调用后（通常）你会得到 `y != x`，而对可变的 `x`，你会有 `y == x`。

因为支持范用计算被认为比能使用其他方法完成的潜在的性能优化（比如使用广播或显式循环）更加重要，所以像 `+=` 和 `*=` 运算符以绑定新值的方式工作。

38.13 异步 IO 与并发同步写入

为什么对于同一个流的并发写入会导致相互混合的输出？

虽然流式 I/O 的 API 是同步的，底层的实现是完全异步的。

思考一下下面的输出：

```julia
julia> @sync for i in 1:3
           @async write(stdout, string(i), " Foo ", " Bar ")
       end
123 Foo  Foo  Foo  Bar  Bar  Bar
```

这是因为，虽然 `write` 调用是同步的，每个参数的写入在等待那一部分 I/O 完成时会生成其他的 Tasks。

`print` 和 `println` 在调用中会”锁定”该流。因此把上例中的 `write` 改成 `println` 会导致：

```julia
julia> @sync for i in 1:3
           @async println(stdout, string(i), " Foo ", " Bar ")
       end
1 Foo  Bar
2 Foo  Bar
3 Foo  Bar
```

你可以使用 `ReentrantLock` 来锁定你的写入，就像这样：

```
julia> l = ReentrantLock();

julia> @sync for i in 1:3
           @async begin
               lock(l)
               try
                   write(stdout, string(i), " Foo ", " Bar ")
               finally
                   unlock(l)
               end
           end
       end
1 Foo  Bar 2 Foo  Bar 3 Foo  Bar
```

38.14 数组

零维数组和标量之间的有什么差别？

零维数组是 `Array{T,0}` 形式的数组，它与标量的行为相似，但是有很多重要的不同。这值得一提，因为这是使用数组的范用定义来解释也符合逻辑的特殊情况，虽然最开始看起来有些非直觉。下面一行定义了一个零维数组：

```
julia> A = zeros()
0-dimensional Array{Float64,0}:
0.0
```

在这个例子中，`A` 是一个含有一个元素的可变容器，这个元素可以通过 `A[] = 1.0` 来设置，通过 `A[]` 来读取。所有的零维数组都有同样的大小（`size(A) == ()`）和长度（`length(A) == 1`）。特别地，零维数组不是空数组。如果你觉得这个非直觉，这里有些想法可以帮助理解 Julia 的这个定义。

- 类比的话，零维数组是”点”，向量是”线”而矩阵是”面”。就像线没有面积一样（但是也能代表事物的一个集合），点没有长度和任意一个维度（但是也能表示一个事物）。
- 我们定义 `prod(())` 为 1，一个数组中的所有的元素个数是大小的乘积。零维数组的大小为 `()`，所以它的长度为 `1`。
- 零维数组没有任何你可以索引的维度——它们仅仅是 `A[]`。我们可以给它们应用同样的”尾一”规则就像对其它维度数组那样，比如 `A[1]`，`A[1,1]`，等；参见Omitted and extra indices.

理解它与普通的标量之间的区别也很重要。标量不是一个可变的容器（尽管它们是可迭代的，可以定义像 `length`，`getindex` 这样的东西，例如 `1[] == 1`）。特别地，如果 `x = 0.0` 是以一个标量来定义，尝试通过 `x[] = 1.0` 来改变它的值会报错。标量 `x` 能够通过 `fill(x)` 转化成包含它的零维数组，并且相对地，一个零维数组 `a` 可以通过 `a[]` 转化成其包含的标量。另外一个区别是标量可以参与到线性代数运算中，比如 `2 * rand(2,2)`，但是零维数组的相似操作 `fill(2) * rand(2,2)` 会报错。

为什么我的 Julia 的线性代数操作测试与其他的语言不同。

你可能找到一些简单的线性代数测试，比如，

```
using BenchmarkTools
A = randn(1000, 1000)
B = randn(1000, 1000)
@btime $A \ $B
@btime $A * $B
```

也许和其他语言不同比如 Matlab 或 R。

由于像这样的操作都非常直接地从相关的 BLAS 函数调用，这样做的原因是，

1. 在每种语言中使用的 BLAS 库
2. 并发线程的数量

Julia 编译并使用自己的 OpenBLAS 副本，当前线程数上限为 8（或内核数）。

修改 OpenBLAS 设置或使用不同的 BLAS 库编译 Julia，例如 Intel MKL，可能会提高性能。你可以使用 MKL.jl，这是一个使 Julia 的线性代数使用英特尔 MKL BLAS 和 LAPACK 而不是 OpenBLAS 的包，或搜索论坛以获取有关如何使用的建议。请注意，英特尔 MKL 不能与 Julia 捆绑在一起，因为它不是开源的。

38.15 计算集群

我该如何管理分布式文件系统的预编译缓存？

When using Julia in high-performance computing (HPC) facilities with shared filesystems, it is recommended to use a shared depot (via the JULIA_DEPOT_PATH environment variable). Since Julia v1.10, multiple Julia processes on functionally similar workers and using the same depot will coordinate via pidfile locks to only spend effort precompiling on one process while the others wait. The precompilation process will indicate when the process is precompiling or waiting for another that is precompiling. If non-interactive the messages are via @debug.

However, due to caching of binary code, the cache rejection since v1.9 is more strict and users may need to set the JULIA_CPU_TARGET environment variable appropriately to get a single cache that is usable throughout the HPC environment.

38.16 Julia 版本发布

你希望使用稳定的、长期支持的或是每日构建版本的 Julia？

Julia 的稳定版是最新发布的 Julia 版本，这是大多数人想要运行的版本。它具有最新的功能，包括改进的性能。Julia 的稳定版本根据 SemVer 版本化为 v1.x.y。在作为候选版本进行几周的测试后，大约每 4-5 个月就会发布一个与新稳定版本相对应的新 Julia 次要版本。与 LTS 版本不同，在 Julia 的另一个稳定版本发布后，稳定版本通常不会收到错误修正。但是，始终可以升级到下一个稳定版本，因为 Julia v1.x 的每个版本都将继续运行为早期版本编写的代码。

如果正在寻找非常稳定的代码库，你可能更喜欢 Julia 的 LTS（长期支持）版本。Julia 当前的 LTS 版本根据 SemVer 版本为 v1.0.x；此分支将继续接收错误修复，直到选择新的 LTS 分支，此时 v1.0.x 系列将不再收到常规错误修复，建议除最保守的用户之外的所有用户升级到新的 LTS 版本系列。作为软件包开发人员，你可能更喜欢针对 LTS 版本进行开发，以最大限度地增加可以使用你的软件包的用户数量。根据 SemVer，为 v1.0 编写的代码将继续适用于所有未来的 LTS 和稳定版本。一般来说，即使针对 LTS，也可以在最新的 Stable 版本中开发和运行代码，以利用改进的性能；只要避免使用新功能（例如添加的库函数或新方法）。

如果您想利用该语言的最新更新，您可能更喜欢 Julia 的每日构建版本，并且不介意今天可用的版本是否偶尔无法正常工作。顾名思义，每日构建版本的发布大约每晚发布一次（取决于构建基础设施的稳定性）。一般来说，每日构建的发布是相当安全的——你的代码不会着火。然而，它们可能出现偶尔的版本倒退和问题，直到更彻底的预发布测试才会发现。你可能希望针对每日构建版本进行测试，以确保在发布之前捕获影响你的用例的版本倒退。

最后，您也可以考虑为自己从源代码构建 Julia。此选项主要适用于那些熟悉命令行或对学习感兴趣的人。如果你是这样的人，你可能也有兴趣阅读我们的 贡献指南。

可以在https://julialang.org/downloads/的下载页面上找到每种下载类型的链接。请注意，并非所有版本的 Julia 都适用于所有平台。

更新我的 Julia 版本后，如何转移已安装软件包的列表？

julia 的每个次要版本都有自己的默认 环境。因此，在安装新的 Julia 次要版本时，默认情况下你使用先前次要版本添加的包将不可用。给定 julia 版本的环境由文件 `Project.toml` 和 `Manifest.toml` 定义，文件夹中的文件与`.julia/environments/`中的版本号匹配，例如`.julia/environments/v1.3`。

如果你安装了一个新的 Julia 次要版本，比如 `1.4`，并且想要在它的默认环境中使用与以前版本（例如 `1.3`）相同的包，你可以从 `1.3` 文件夹复制文件 `Project.toml` 的内容到 `1.4`。然后，在新的 Julia 版本的会话中，输入`]` 键进入 “包管理模式”，并运行命令 `instantiate`。

此操作将从复制的文件中解析一组与目标 Julia 版本兼容的可行包，并在合适时安装或更新它们。如果你不仅要重现软件包，还要重现在以前的 Julia 版本中使用的版本，您还应该在运行 Pkg 命令 `instantiate` 之前复制 `Manifest.toml` 文件。但是，请注意，包可能定义了兼容性约束，这些约束可能会受到更改 Julia 版本的影响，因此你在 `1.3` 中拥有的确切版本集可能不适用于 `1.4`。

Chapter 39

与其他语言的显著差异

39.1 与 MATLAB 的显著差异

虽然 MATLAB 用户可能会发现 Julia 的语法很熟悉，但 Julia 不是 MATLAB 的克隆。它们之间存在重大的语法和功能差异。以下是一些可能会使习惯于 MATLAB 的 Julia 用户感到困扰的显著差异：

- Julia 数组使用方括号 `A[i,j]` 进行索引。
- Julia 数组在分配给另一个变量时不会被复制。在 `A = B` 之后，改变 `B` 的元素也会改变 `A` 的元素。To avoid this, use `A = copy(B)`.
- Julia 的值在向函数传递时不发生复制。如果某个函数修改了数组，这一修改对调用者是可见的。
- Julia 不会在赋值语句中自动增长数组。而在 MATLAB 中 `a(4) = 3.2` 可以创建数组 `a = [0 0 0 3.2]`，而 `a(5) = 7` 可以将它增长为 `a = [0 0 0 3.2 7]`。如果 `a` 的长度小于 5 或者这个语句是第一次使用标识符 `a`，则相应的 Julia 语句 `a[5] = 7` 会抛出错误。Julia 使用 `push!` 和 `append!` 来增长 `Vector`，它们比 MATLAB 的 `a(end+1) = val` 更高效。
- 虚数单位 `sqrt(-1)` 在 Julia 中表示为 `im`，而不是在 MATLAB 中的 `i` 或 `j`。
- 在 Julia 中，不带小数点的字面数字（如 `42`）创建的是整数而不是浮点数。因此，如果某些操作的预期结果是浮点数，就会产生域错误；例如，`julia> a = -1; 2^a` 会产生域错误，因为结果不是整数（详情请参见常见问题中的域错误）。
- 在 Julia 中，能返回多个值并将其赋值为元组，例如 `(a, b) = (1, 2)` 或 `a, b = 1, 2`。在 Julia 中不存在 MATLAB 的 `nargout`，它通常在 MATLAB 中用于根据返回值的数量执行可选工作。取而代之的是，用户可以使用可选参数和关键字参数来实现类似的功能。
- Julia 拥有真正的一维数组。列向量的大小为 `N`，而不是 `Nx1`。例如，`rand(N)` 创建一个一维数组。
- 在 Julia 中，`[x,y,z]` 将始终构造一个包含 `x`、`y` 和 `z` 的 3 元数组。
  - 要在第一个维度（「垂直列」）中连接元素，请使用 `vcat(x,y,z)` 或用分号分隔（`[x; y; z]`）。
  - 要在第二个维度（「水平行」）中连接元素，请使用 `hcat(x,y,z)` 或用空格分隔（`[x y z]`）。
  - 要构造分块矩阵（在前两个维度中连接元素），请使用 `hvcat` 或组合空格和分号（`[a b; c d]`）。

- 在 Julia 中，a:b 和 a:b:c 构造 AbstractRange 对象。使用 collect(a:b) 构造一个类似 MATLAB 中完整的向量。通常，不需要调用 collect。在大多数情况下，AbstractRange 对象将像普通数组一样运行，但效率更高，因为它是懒惰求值。这种创建专用对象而不是完整数组的模式经常被使用，并且也可以在诸如 range 之类的函数中看到，或者在诸如 enumerate 和 zip 之类的迭代器中看到。特殊对象大多可以像正常数组一样使用。
- The : operator has a different precedence in R and Julia. In particular, in Julia arithmetic operators have higher precedence than the : operator, whereas the reverse is true in R. For example, 1:n-1 in Julia is equivalent to 1:(n-1) in R.
- Julia 中的函数返回其最后一个表达式或 return 关键字的值而无需在函数定义中列出要返回的变量的名称（有关详细信息，请参阅 return 关键字）。
- Julia 脚本可以包含任意数量的函数，并且在加载文件时，所有定义都将在外部可见。可以从当前工作目录之外的文件加载函数定义。
- 在 Julia 中，例如 sum、prod 和 max 的归约操作会作用到数组的每一个元素上，当调用时只有一个函数，例如 sum(A)，即使 A 并不只有一个维度。
- 在 Julia 中，调用无参数的函数时必须使用小括号，例如 rand()。
- Julia 不鼓励使用分号来结束语句。语句的结果不会自动打印（除了在 REPL 中），并且代码的一行不必使用分号结尾。println 或者 @printf 能用来打印特定输出。
- 在 Julia 中，如果 A 和 B 是数组，像 A == B 这样的逻辑比较运算符不会返回布尔值数组。相反地，请使用 A .== B。对于其他的像是 <、> 的布尔运算符同理。
- 在 Julia 中，运算符&、| 和 ⊻ （xor）进行按位操作，分别与 MATLAB 中的 and、or 和 xor 等价，并且优先级与 Python 的按位运算符相似（不像 C）。他们可以对标量运算或者数组中逐元素运算，可以用来合并逻辑数组，但是注意运算顺序的区别：括号可能是必要的（例如，选择 A 中等于 1 或 2 的元素可使用 (A .== 1) .| (A .== 2)）。
- 在 Julia 中，集合的元素可以使用 splat 运算符 ... 来作为参数传递给函数，如 xs=[1,2]; f(xs...)。
- Julia 的 svd 将奇异值作为向量而非密集对角矩阵返回。
- 在 Julia 中，... 不用于延续代码行。不同的是，Julia 中不完整的表达式会自动延续到下一行。
- 在 Julia 和 MATLAB 中，变量 ans 被设置为交互式会话中提交的最后一个表达式的值。在 Julia 中与 MATLAB 不同的是，当 Julia 代码以非交互式模式运行时并不会设置 ans。
- Julia 的 struct 不支持在运行时动态地添加字段，这与 MATLAB 的 class 不同。如需支持，请使用 Dict。Julia 中的字典不是有序的。
- 在 Julia 中，每个模块有自身的全局作用域/命名空间，而在 MATLAB 中只有一个全局作用域。
- 在 MATLAB 中，删除不需要的值的惯用方法是使用逻辑索引，如表达式 x(x>3) 或语句 x(x>3) = [] 来 in-place 修改 x。相比之下，Julia 提供了更高阶的函数 filter 和 filter!，允许用户编写 filter(z->z>3, x) 和 filter!(z->z>3, x) 来代替相应直译 x[x.>3] 和 x = x[x.>3]。使用 filter! 可以减少临时数组的使用。
- 类似于提取（或「解引用」）元胞数组的所有元素的操作，例如 MATLAB 中的 vertcat(A{:})，在 Julia 中是使用 splat 运算符编写的，例如 vcat(A...)。
- 在 Julia 中，adjoint 函数执行共轭转置；在 MATLAB 中，adjoint 提供了经典伴随，它是余子式的转置。
- 在 Julia 中，a^b^c 被认为是 a^(b^c) 而在 MATLAB 中它是 (a^b)^c。

39.2 与 R 的显著差异

Julia 的目标之一是为数据分析和统计编程提供高效的语言。对于从 R 转到 Julia 的用户来说，这是一些显著差异：

- Julia 的单引号封闭字符，而不是字符串。
- Julia 可以通过索引字符串来创建子字符串。在 R 中，在创建子字符串之前必须将字符串转换为字符向量。
- 在 Julia 中，与 Python 相同但与 R 不同的是，字符串可由三重引号 `""" ... """` 创建。此语法对于构造包含换行符的字符串很方便。
- 在 Julia 中，可变参数使用 splat 运算符 `...` 指定，该运算符总是跟在具体变量的名称后面，与 R 的不同，R 的 `...` 可以单独出现。
- 在 Julia 中，模数是 `mod(a, b)`，而不是 `a %% b`。Julia 中的 `%` 是余数运算符。
- Julia constructs vectors using brackets. Julia's `[1, 2, 3]` is the equivalent of R's `c(1, 2, 3)`.
- 在 Julia 中，并非所有数据结构都支持逻辑索引。此外，Julia 中的逻辑索引只支持长度等于被索引对象的向量。例如：
  - 在 R 中，`c(1, 2, 3, 4)[c(TRUE, FALSE)]` 等价于 `c(1, 3)`。
  - 在 R 中，`c(1, 2, 3, 4)[c(TRUE, FALSE, TRUE, FALSE)]` 等价于 `c(1, 3)`。
  - 在 Julia 中，`[1, 2, 3, 4][[true, false]]` 抛出 `BoundsError`。
  - 在 Julia 中，`[1, 2, 3, 4][[true, false, true, false]]` 产生 `[1, 3]`。
- 与许多语言一样，Julia 并不总是允许对不同长度的向量进行操作，与 R 不同，R 中的向量只需要共享一个公共的索引范围。例如，`c(1, 2, 3, 4) + c(1, 2)` 是有效的 R，但等价的 `[1, 2, 3, 4] + [1, 2]` 在 Julia 中会抛出一个错误。
- 在逗号不改变代码含义时，Julia 允许使用可选的尾随括号。在索引数组时，这可能在 R 用户间造成混淆。例如，R 中的 `x[1,]` 将返回矩阵的第一行；但是，在 Julia 中，引号被忽略，于是 `x[1,] == x[1]`，并且将返回第一个元素。要提取一行，请务必使用 `:`，如 `x[1,:]`。
- Julia 的 `map` 首先接受函数，然后是该函数的参数，这与 R 中的 `lapply(<structure>, function, ...)` 不同。类似地，R 中的 `apply(X, MARGIN, FUN, ...)` 等价于 Julia 的 `mapslices`，其中函数是第一个参数。
- R 中的多变量 apply，如 `mapply(choose, 11:13, 1:3)`，在 Julia 中可以编写成 `broadcast(binomial, 11:13, 1:3)`。等价地，Julia 提供了更短的点语法来向量化函数 `binomial.(11:13, 1:3)`。
- Julia 使用 `end` 来表示条件块（如 `if`）、循环块（如 `while`/`for`）和函数的结束。为了代替单行 `if ( cond ) statement`，Julia 允许形式为 `if cond; statement; end`、`cond && statement` 和 `!cond || statement` 的语句。后两种语法中的赋值语句必须显式地包含在括号中，例如 `cond && (x = value)`，这是因为运算符的优先级。
- 在 Julia 中，`<-`, `<<-` 和 `->` 不是赋值运算符。
- Julia 的 `->` 创建一个匿名函数。
- Julia 使用括号构造向量。Julia 的 `[1, 2, 3]` 等价于 R 的 `c(1, 2, 3)`。
- Julia 的 `*` 运算符可以执行矩阵乘法，这与 R 不同。如果 `A` 和 `B` 都是矩阵，那么 `A * B` 在 Julia 中表示矩阵乘法，等价于 R 的 `A %*% B`。在 R 中，相同的符号将执行逐元素（Hadamard）乘积。要在 Julia 中使用逐元素乘法运算，你需要编写 `A .* B`。

- Julia 使用 transpose 函数来执行矩阵转置，使用 ' 运算符或 adjoint 函数来执行共轭转置。因此，Julia 的 transpose(A) 等价于 R 的 t(A)。另外，Julia 中的非递归转置由 permutedims 函数提供。
- Julia 在编写 if 语句或 for/while 循环时不需要括号：请使用 for i in [1, 2, 3] 代替 for (int i=1; i <= 3; i++)，以及 if i == 1 代替 if (i == 1)
- Julia 不把数字 0 和 1 视为布尔值。在 Julia 中不能编写 if (1)，因为 if 语句只接受布尔值。相反，可以编写 if true、if Bool(1) 或 if 1==1。
- Julia 不提供 nrow 和 ncol。相反，请使用 size(M, 1) 代替 nrow(M) 以及 size(M, 2) 代替 ncol(M)
- Julia 仔细区分了标量、向量和矩阵。在 R 中，1 和 c(1) 是相同的。在 Julia 中，它们不能互换地使用。
- Julia 的 diag 和 diagm 与 R 的不同。
- Julia 赋值操作的左侧不能为函数调用的结果：你不能编写 diag(M) = fill(1, n)。
- Julia 不鼓励使用函数填充主命名空间。Julia 的大多数统计功能都可在 JuliaStats 组织的包中找到。例如：
  - 与概率分布相关的函数由 Distributions 包提供。
  - DataFrames 包提供数据帧。
  - 广义线性模型由 GLM 包提供。
- Julia 提供了元组和真正的哈希表，但不提供 R 风格的列表。在返回多个项时，通常应使用元组或具名元组：请使用 (1, 2) 或 (a=1, b=2) 代替 list(a = 1, b = 2)。
- Julia 鼓励用户编写自己的类型，它比 R 中的 S3 或 S4 对象更容易使用。Julia 的多重派发系统意味着 table(x::TypeA) 和 table(x::TypeB) 类似于 R 的 table.TypeA(x) 和 table.TypeB(x)。
- Julia 的值在向函数传递时不发生复制。如果某个函数修改了数组，这一修改对调用者是可见的。这与 R 非常不同，允许新函数更高效地操作大型数据结构。
- 在 Julia 中，向量和矩阵使用 hcat、vcat 和 hvcat 拼接，而不是像在 R 中那样使用 c、rbind 和 cbind。
- 在 Julia 中，像 a:b 这样的 range 不是 R 中的向量简写，而是一个专门的 AbstractRange 对象，该对象用于没有高内存开销地进行迭代。要将 range 转换为 vector，请使用 collect(a:b)。
- Julia 的 max 和 min 分别等价于 R 中的 pmax 和 pmin，但两者的参数都需要具有相同的维度。虽然 maximum 和 minimum 代替了 R 中的 max 和 min，但它们之间有重大区别。
- Julia 的 sum、prod、maximum 和 minimum 与它们在 R 中的对应物不同。它们都接受一个可选的关键字参数 dims，它表示执行操作的维度。例如，在 Julia 中令 A = [1 2; 3 4]，在 R 中令 B <- rbind(c(1,2),c(3,4)) 是与之相同的矩阵。然后 sum(A) 得到与 sum(B) 相同的结果，但 sum(A, dims=1) 是一个包含每一列总和的行向量，sum(A, dims=2) 是一个包含每一行总和的列向量。这与 R 的行为形成了对比，在 R 中，单独的 colSums(B) 和 rowSums(B) 提供了这些功能。如果 dims 关键字参数是向量，则它指定执行求和的所有维度，并同时保持待求和数组的维数，例如 sum(A, dims=(1,2)) == hcat(10)。应该注意的是，没有针对第二个参数的错误检查。
- Julia 具有一些可以改变其参数的函数。例如，它具有 sort 和 sort!。
- 在 R 中，高性能需要向量化。在 Julia 中，这几乎恰恰相反：性能最高的代码通常通过去向量化的循环来实现。

- Julia 是立即求值的，不支持 R 风格的惰性求值。对于大多数用户来说，这意味着很少有未引用的表达式或列名。
- Julia 不支持 NULL 类型。最接近的等价物是 nothing，但它的行为类似于标量值而不是列表。请使用 x === nothing 代替 is.null(x)。
- 在 Julia 中，缺失值由 missing 表示，而不是由 NA 表示。请使用 ismissing(x)（或者在向量上使用逐元素操作 ismissing.(x)）代替 isna(x)。通常使用 skipmissing 代替 na.rm=TRUE（尽管在某些特定情况下函数接受 skipmissing 参数）。
- Julia 缺少 R 中的 assign 或 get 的等价物。
- 在 Julia 中，return 不需要括号。
- 在 R 中，删除不需要的值的惯用方法是使用逻辑索引，如表达式 x[x>3] 或语句 x = x[x>3] 来 in-place 修改 x。相比之下，Julia 提供了更高阶的函数 filter 和 filter!，允许用户编写 filter(z->z>3, x) 和 filter!(z->z>3, x) 来代替相应直译 x[x.>3] 和 x = x[x.>3]。使用 filter! 可以减少临时数组的使用。

39.3 与 Python 的显著差异

- Julia 的 for, if, while 等语句块都以 end 关键字结束。代码的缩进不像在 Python 中那样重要。Julia 也没有 pass 关键字。
- Julia 中的字符串使用双引号构造，如 "text"，也可以使用三引号构造多行字符串。而在 Python 中可以使用单引号（'text'）或者双引号（"text"）。单引号在 Julia 中用来表示单个字符，例如 'c'。
- 在 Julia 中字符串的拼接使用 *，而不是像 Python 一样使用 +。类似的，字符串重复多次 Julia 使用 ^ 而不是 *。Julia 也不支持隐式的字符串拼接，例如 Python 中的 'ab' 'cd' == 'abcd'。
- Python 列表——灵活但缓慢——对应于 Julia 的 Vector{Any} 类型或更一般的 Vector{T}，其中 T 是一些非具体元素类型。“快”的数组，如 NumPy 数组，它们就地存储元素（即，dtype 是 np.float64、[('f1', np.uint64), ('f2', np.int32)]，等）可以用 Array{T} 表示，其中 T 是一个具体的、不可变的元素类型。这包括内置类型，如 Float64、Int32、Int64，也包括更复杂的类型，如 Tuple{UInt64,Float64} 和许多用户定义的类型。
- 在 Julia 中，数组、字符串等的索引从 1 开始，而不是从 0 开始。
- Julia 里的切片包含最后一个元素。Julia 里的 a[2:3] 等同于 Python 中的 a[1:3]。
- Unlike Python, Julia allows AbstractArrays with arbitrary indexes. Python's special interpretation of negative indexing, a[-1] and a[-2], should be written a[end] and a[end-1] in Julia.
- Julia 的索引必须写全。Python 中的 x[1:] 等价于 Julia 中的 x[2:end]。
- In Julia, : before any object creates a Symbol or *quotes* an expression; so, x[:5] is same as x[5]. If you want to get the first n elements of an array, then use range indexing.
- Julia 的范围语法为 x[start:step:stop]，而 Python 的格式为 x[start:(stop+1):step]。

因此 Python 中的 x[0:10:2] 等价于 Julia 里的 x[1:2:10]。类似的 Python 中的反转数组 x[::-1] 等价于 Julia 中的 x[end:-1:1]。

- In Julia, ranges can be constructed independently as start:step:stop, the same syntax it uses in array-indexing. The range function is also supported.

- 在 Julia 中队一个矩阵取索引 X[[1,2], [1,3]] 返回一个子矩阵，它包含了第一和第二行与第一和第三列的交集。在 Python 中 X[[1,2], [1,3]] 返回一个向量，它包含索引 [1,1] 和 [2,3] 的值。Julia 中的 X[[1,2], [1,3]] 等价于 Python 中的 X[np.ix_([0,1],[0,2])]。Python 中的 X[[1,2], [1,3]] 等价于 Julia 中的 X[[CartesianIndex(1,1), CartesianIndex(2,3)]]。
- Julia 没有用来续行的语法：如果在行的末尾，到目前为止的输入是一个完整的表达式，则认为其已经结束；否则，认为输入继续。强制表达式继续的一种方式是将其包含在括号中。
- 默认情况下，Julia 数组是列优先（Fortran 排序），而 NumPy 数组是行优先（C 排序）。为了在循环数组时获得最佳性能，Julia 中的循环顺序应相对于 NumPy 颠倒（请参阅性能提示的相关部分）。
- Julia 的更新运算符（例如 +=，-=，···）是非原位操作（not in-place），而 Numpy 的是。这意味着 A = [1, 1]; B = A; B += [3, 3] 不会改变 A 中的值，而将名称 B 重新绑定到右侧表达式 B = B + 3 的结果，这是一个新的数组。对于 in-place 操作，使用 B .+= 3（另请参阅 dot operators）、显式的循环或者 InplaceOps.jl。
- Julia 的函数在调用时，每次都对默认参数重新求值，不像 Python 只在函数定义时对默认参数求一次值。举例来说：Julia 的函数 f(x=rand()) = x 在无参数调用时（f()），每次都会返回不同的随机数。另一方面，函数 g(x=[1,2]) = push!(x,3) 无参数调用时 g()，永远返回 [1,2,3]。
- 在 Julia 中，必须使用关键字来传递关键字参数，这与 Python 中通常可以按位置传递它们不同。尝试按位置传递关键字参数会改变方法签名，从而导致 MethodError 或调用错误的方法。
- 在 Julia 中，% 是余数运算符，而在 Python 中是模运算符。

（译注：二者在参数有负数时有区别）

- 在 Julia 中，常用的整数类型 Int 对应机器的整数类型，Int32 或 Int64。不像 Python 中的整数 int 是任意精度的。这意味着 Julia 中默认的整数类型会溢出，因此 2^64 == 0。如果你要表示一个大数，请选择一个合适的类型。如：Int128、任意精度的 BigInt 或者浮点类型 Flost64。
- Julia 中虚数单位 sqrt(-1) 是 im，而不是 Python 中的 j。
- Julia 中指数是 ^，而不是 Python 中的 **。
- Julia 使用 Nothing 类型的实例 nothing 代表空值（null），而不是 Python 中 NoneType 类的 None。
- 在 Julia 中，标准的运算符作用在矩阵上就得到矩阵操作，不像 Python 标准运算符默认是逐元素操作。当 A 和 B 都是矩阵时，A * B 在 Julia 中代表着矩阵乘法，而不是 Python 中的逐元素相乘。即：Julia 中的 A * B 等同于 Python 的 A @ B；Python 中的 A * B 等同于 Julia 中的 A .* B。
- Julia 中的伴随操作符 ' 返回向量的转置（一种行向量的懒惰表示法）。Python 中对向量执行 .T 返回它本身（没有效果）。
- In Julia, a function may contain multiple concrete implementations (called *methods*), which are selected via multiple dispatch based on the types of all arguments to the call, as compared to functions in Python, which have a single implementation and no polymorphism (as opposed to Python method calls which use a different syntax and allows dispatch on the receiver of the method).
- Julia 没有类（class），取而代替的是结构体（structures），可以是可变的或不可变的，它们只包含数据而不包含方法。
- 在 Python 中调用类实例的方法 (x = MyClass(*args); x.f(y)) 对应于 Julia 中的函数调用，例如 x = MyType(args...); f(x, y)。总的来说，多重派发比 Python 类系统更灵活和强大。

- Julia 的结构体有且只能有一个抽象超类型（abstract supertype），而 Python 的类可以纪成一个或多个、抽象或具体的超类（superclasses）。
- 逻辑 Julia 程序结构（包和模块）独立于文件结构（`include` 用于附加文件），而 Python 代码结构由目录（包）和文件（模块）定义。
- Julia 中的三元运算符 `x > 0 ? 1 : -1` 对应于 Python 中的条件表达式 `1 if x > 0 else -1`。
- Julia 中以 `@` 开头的符号是宏（macro），而 Python 中是装饰器（decorator）。
- Julia 的异常处理使用 `try` —`catch` —`finally`，而不是 Python 的 `try` —except —`finally`。与 Python 不同的是，因为性能的原因，Julia 不推荐在正常流程中使用异常处理。(compared with Python, Julia is faster at ordinary control flow but slower at exception-catching).
- Julia 的循环很快，所以没必要手动向量化（vectorized）。
- 小心 Julia 中的非常量全局变量，尤其它出现在循环中时。因为你在 Julia 中可以写出贴近硬件的代码，这时使用全局变量的影响非常大（参见性能建议）
- In Julia, rounding and truncation are explicit. Python's `int(3.7)` should be `floor(Int, 3.7)` or `Int(floor(3.7))` and is distinguished from `round(Int, 3.7)`. `floor(x)` and `round(x)` on their own return an integer value of the same type as `x` rather than always returning `Int`.
- In Julia, parsing is explicit. Python's `float("3.7")` would be `parse(Float64, "3.7")` in Julia.
- Python 中大多数的值都能用在逻辑运算中。例如：`if "a"` 永真，`if ""` 恒假。在 Julia 中你只能使用布尔类型的值，或者显示的将其他值转为布尔类型，否则就会抛出异常。例如当你想测试字符串是否为空是，请使用 `if !isempty("")`。
- 在 Julia 中大多数代码块都会引入新的本地作用域（local scope）。例如：循环和异常处理的 try —catch —finally。注意：列表推断（comprehensions）与生成器在 Julia 和 Python 中都会引入新的作用域；而 if 分支则都不会引入。

39.4 与 C/C++ 的显著差异

- Julia 的数组由方括号索引，方括号中可以包含不止一个维度 `A[i,j]`。这样的语法不仅仅是像 C/C++ 中那样对指针或者地址引用的语法糖，参见关于数组构造的语法的 Julia 文档。
- 在 Julia 中，数组、字符串等的索引从 1 开始，而不是从 0 开始。
- Julia 的数组在赋值给另一个变量时不发生复制。执行 `A = B` 后，改变 `B` 中元素也会修改 `A`。像 += 这样的更新运算符不会以 in-place 的方式执行，而是相当于 `A = A + B`，将左侧绑定到右侧表达式的计算结果上。
- Julia 的数组是列优先的（Fortran 顺序），而 C/C++ 的数组默认是行优先的。要使数组上的循环性能最优，在 Julia 中循环的顺序应该与 C/C++ 相反（参见 性能建议）。
- Julia 的值在赋值或向函数传递时不发生复制。如果某个函数修改了数组，这一修改对调用者是可见的。
- 在 Julia 中，空格是有意义的，这与 C/C++ 不同，所以向 Julia 程序中添加或删除空格时必须谨慎。
- 在 Julia 中，没有小数点的数值字面量（如 42）生成有符号整数，类型为 `Int`，但如果字面量太长，超过了机器字长，则会被自动提升为容量更大的类型，例如 `Int64`（如果 `Int` 是 `Int32`）、`Int128`，或者任意精度的 `BigInt` 类型。不存在诸如 L, LL, U, UL, ULL 这样的数值字面量后缀指示无符号和/或有符号与无符号。十进制字面量始终是有符号的，十六进制字面量（像 C/C++

一样由 0x 开头）是无符号的。另外，十六进制字面量与 C/C++/Java 不同，也与 Julia 中的十进制字面量不同，它们的类型取决于字面量的**长度**，包括开头的 0。例如，0x0 和 0x00 的类型是 UInt8，0x000 和 0x0000 的类型是 UInt16。同理，字面量的长度在 5-8 之间，类型为 UInt32；在 9-16 之间，类型为 UInt64；在 17-32 之间，类型为 UInt128。当定义十六进制掩码时，就需要将这一问题考虑在内，比如 ~0xf == 0xf0 与 ~0x000f == 0xfff0 完全不同。64 位 Float64 和 32 位 Float32 的字面量分别表示为 1.0 和 1.0f0。浮点字面量在无法被精确表示时舍入（且不会提升为 BigFloat 类型）。浮点字面量在行为上与 C/C++ 更接近。八进制（前缀为 0o）和二进制（前缀为 0b）也被视为无符号的。

- 在 Julia 中，当两个操作数都是整数类型时，除法运算符 / 返回一个浮点数。要执行整数除法，请使用 div 或 ÷。
- 使用浮点类型索引数组在 Julia 中通常是错误的。C 表达式 a[i / 2] 的 Julia 等价写法是 a[i ÷ 2 + 1]，其中 i 是整数类型。
- 字符串字面量可用 " 或 """ 分隔，用 """ 分隔的字面量可以包含 " 字符而无需像 "\"" 这样来引用它。字符串字面量可以包含插入其中的其他变量或表达式，由 $variablename 或 $(expression) 表示，它在该函数所处的上下文中计算变量名或表达式。
- // 表示 Rational 数，而非单行注释（其在 Julia 中是 #）
- #= 表示多行注释的开头，=# 结束之。
- Julia 中的函数返回其最后一个表达式或 return 关键字的值。可以从函数中返回多个值并将其作为元组赋值，如 (a, b) = myfunction() 或 a, b = myfunction()，而不必像在 C/C++ 中那样必须传递指向值的指针（即 a = myfunction(&b))。
- Julia 不要求使用分号来结束语句。表达式的结果不会自动打印（除了在交互式提示符中，即 REPL），且代码行不需要以分号结尾。println 或 @printf 可用于打印特定输出。在 REPL 中，; 可用于抑制输出。; 在 [ ] 中也有不同的含义，需要注意。; 可用于在单行中分隔表达式，但在许多情况下不是绝对必要的，更经常是为了可读性。
- 在 Julia 中，运算符 ⊻（xor）执行按位 XOR 操作，即 C/C++ 中的 ^。此外，按位运算符不具有与 C/C++ 相同的优先级，所以可能需要括号。
- Julia 的 ^ 是取幂（pow），而非 C/C++ 中的按位 XOR（在 Julia 中请使用 ⊻ 或 xor）
- Julia 有两个右移运算符，>> 和 >>>。>> 执行算术移位，>>> 始终执行逻辑移位，这与 C/C++ 不同，其中 >> 的含义取决于被移位的值的类型。
- Julia 的 -> 创建一个匿名函数，它并不通过指针访问成员。
- Julia 在编写 if 语句或 for/while 循环时不需要括号：请使用 for i in [1, 2, 3] 代替 for (int i=1; i <= 3; i++)，以及 if i == 1 代替 if (i == 1)
- Julia 不把数字 0 和 1 视为布尔值。在 Julia 中不能编写 if (1)，因为 if 语句只接受布尔值。相反，可以编写 if true、if Bool(1) 或 if 1==1。
- Julia 使用 end 来表示条件块（如 if）、循环块（如 while/for）和函数的结束。为了代替单行 if (cond) statement，Julia 允许形式为 if cond; statement; end、cond && statement 和 !cond || statement 的语句。后两种语法中的赋值语句必须显式地包含在括号中，例如 cond && (x = value)，这是因为运算符的优先级。
- Julia 没有用来续行的语法：如果在行的末尾，到目前为止的输入是一个完整的表达式，则认为其已经结束；否则，认为输入继续。强制表达式继续的一种方式是将其包含在括号中。

- Julia 宏对已解析的表达式进行操作，而非程序的文本，这允许它们执行复杂的 Julia 代码转换。宏名称以 `@` 字符开头，具有类似函数的语法 `@mymacro(arg1, arg2, arg3)` 和类似语句的语法 `@mymacro arg1 arg2 arg3`。两种形式的语法可以相互转换；如果宏出现在另一个表达式中，则类似函数的形式尤其有用，并且它通常是最清晰的。类似语句的形式通常用于标注块，如在分布式 `for` 结构中：`@distributed for i in 1:n; #= body =#; end`。如果宏结构的结尾不那么清晰，请使用类似函数的形式。
- Julia 有一个枚举类型，使用宏 `@enum(name, value1, value2, ...)` 来表示，例如：`@enum(Fruit, banana=1, apple, pear)`。
- 按照惯例，修改其参数的函数在名称的末尾有个 `!`，例如 `push!`。
- 在 C++ 中，默认情况下，你具有静态分派，即为了支持动态派发，你需要将函数标注为 virtual 函数。另一方面，Julia 中的每个方法都是「virtual」（尽管它更通用，因为方法是在每个参数类型上派发的，而不仅仅是 `this`，并且使用的是最具体的声明规则）。

Julia ⇔ C/C++: Namespaces

- C/C++ `namespace`s correspond roughly to Julia `module`s.
- There are no private globals or fields in Julia. Everything is publicly accessible through fully qualified paths (or relative paths, if desired).
- `using MyNamespace::myfun` (C++) corresponds roughly to `import MyModule: myfun` (Julia).
- `using namespace MyNamespace` (C++) corresponds roughly to `using MyModule` (Julia)
  - In Julia, only `export`ed symbols are made available to the calling module.
  - In C++, only elements found in the included (public) header files are made available.
- Caveat: `import`/`using` keywords (Julia) also *load* modules (see below).
- Caveat: `import`/`using` (Julia) works only at the global scope level (`module`s)
  - In C++, `using namespace X` works within arbitrary scopes (ex: function scope).

Julia ⇔ C/C++: Module loading

- When you think of a C/C++ "**library**", you are likely looking for a Julia "**package**".
  - Caveat: C/C++ libraries often house multiple "software modules" whereas Julia "packages" typically house one.
  - Reminder: Julia `module`s are global scopes (not necessarily "software modules").
- **Instead of build/`make` scripts**, Julia uses "Project Environments" (sometimes called either "Project" or "Environment").
  - Build scripts are only needed for more complex applications (like those needing to compile or download C/C++ executables).
  - To develop application or project in Julia, you can initialize its root directory as a "Project Environment", and house application-specific code/packages there. This provides good control over project dependencies, and future reproducibility.
  - Available packages are added to a "Project Environment" with the `Pkg.add()` function or Pkg REPL mode. (This does not **load** said package, however).

  - The list of available packages (direct dependencies) for a "Project Environment" are saved in its `Project.toml` file.
  - The *full* dependency information for a "Project Environment" is auto-generated & saved in its `Manifest.toml` file by `Pkg.resolve()`.

- Packages ("software modules") available to the "Project Environment" are loaded with `import` or `using`.
  - In C/C++, you `#include <moduleheader>` to get object/function declarations, and link in libraries when you build the executable.
  - In Julia, calling using/import again just brings the existing module into scope, but does not load it again (similar to adding the non-standard `#pragma once` to C/C++).

- **Directory-based package repositories** (Julia) can be made available by adding repository paths to the `Base.LOAD_PATH` array.
  - Packages from directory-based repositories do not require the `Pkg.add()` tool prior to being loaded with `import` or `using`. They are simply available to the project.
  - Directory-based package repositories are the **quickest solution** to developping local libraries of "software modules".

Julia ⇔ C/C++: Assembling modules

- In C/C++, `.c`/`.cpp` files are compiled & added to a library with build/make scripts.
  - In Julia, `import [PkgName]`/`using [PkgName]` statements load `[PkgName].jl` located in a package's `[PkgName]/src/` subdirectory.
  - In turn, `[PkgName].jl` typically loads associated source files with calls to `include "[someotherfile].jl"`.

- `include "./path/to/somefile.jl"` (Julia) is very similar to `#include "./path/to/somefile.jl"` (C/C++).
  - However `include "..."` (Julia) is not used to include header files (not required).
  - **Do not use** `include "..."` (Julia) to load code from other "software modules" (use `import`/`using` instead).
  - `include "path/to/some/module.jl"` (Julia) would instantiate multiple versions of the same code in different modules (creating *distinct* types (etc.) with the *same* names).
  - `include "somefile.jl"` is typically used to assemble multiple files *within the same Julia package* ("software module"). It is therefore relatively straightforward to ensure file are `include`d only once (No `#ifdef` confusion).

Julia ⇔ C/C++: Module interface

- C++ exposes interfaces using "public" `.h`/`.hpp` files whereas Julia `module`s mark specific symbols that are intended for their users as `public`or `export`ed.
  - Often, Julia `module`s simply add functionality by generating new "methods" to existing functions (ex: `Base.push!`).
  - Developers of Julia packages therefore cannot rely on header files for interface documentation.
  - Interfaces for Julia packages are typically described using docstrings, README.md, static web pages, ...

- Some developers choose not to `export` all symbols required to use their package/module.
  - Users might be expected to access these components by qualifying functions/structs/... with the package/module name (ex: `MyModule.run_this_task(...)`).

Julia ⇔ C/C++: Quick reference

Software Concept	Julia	C/C++
unnamed scope	`begin` ... `end`	`{` ... `}`
function scope	`function x()` ... `end`	`int x() {` ... `}`
global scope	`module MyMod` ... `end`	`namespace MyNS {` ... `}`
software module	A Julia "package"	`.h`/`.hpp` files +compiled `somelib.a`

assem-bling
software modules
	`SomePkg.jl`: ...
	`import("subfile1.jl")`
	`import("subfile2.jl")`
	...
		`$(AR) *.o` ⇒ `somelib.a`

im-port software module	`import SomePkg`	`#include <somelib>` +link in `somelib.a`
module library	`LOAD_PATH[]`, *Git repository, **custom package registry	more `.h`/`.hpp` files +bigger compiled `somebiglib.a`

* The Julia package manager supports registering multiple packages from a single Git repository.
 * This allows users to house a library of related packages in a single repository.
 ** Julia registries are primarily designed to provide versioning \& distribution of packages.
 ** Custom package registries can be used to create a type of module library.

39.5 与 Common Lisp 的显著差异

- Julia 默认使用 1 开始的数组索引，它也能处理任意的索引顺序。
- 函数和变量共用一个命名空间（"Lisp-1"）。
- For performance, Julia prefers that operations have type stability. Where Common Lisp abstracts away from the underlying machine operations, Julia cleaves closer to them. For example:
  - Integer division using / always returns a floating-point result, even if the computation is exact.
    * // always returns a rational result
    * ÷ always returns a (truncated) integer result
  - Bignums are supported, but conversion is not automatic; ordinary integers overflow.
  - Complex numbers are supported, but to get complex results, you need complex inputs.
  - There are multiple Complex and Rational types, with different component types.
- 典型的使用 Julia 进行原型开发时，也会对镜像进行连续的修改，Revise.jl 包提供了这个功能。
- 对于性能，Julia 更喜欢操作具有 类型稳定性。Common Lisp 从底层机器操作中抽象出来，而 Julia 则更接近它们。例如：
  - 使用 / 的整数除法总是返回浮点结果，即使计算是精确的。
    * // 总是返回一个有理数结果
    * ÷ 总是返回一个（被截断的）整数结果
  - Julia 支持大整数，但不会自动转换。默认的整数类型会溢出。
  - 支持复数，但要获得复数结果，你需要复数输入。
  - 有多种 Complex 和 Rational 类型，具有不同的组成类型。

- 模块（名称空间）可以是分层的。`import` 和 `using` 有着双重角色：他们加载代码并让代码在命名空间中可用。`import` 用于仅有模块名是可用的情况，大致等价于 `ASDF:LOAD-OP`。槽名（Slot name）不需要单独导出。全局变量不能从模块的外部赋值，除了 `eval(mod, :(var = val))` 这个例外情况。
- 宏以 `@` 开头，并没有像 Common Lisp 那样无缝地集成到语言中；因此在 Julia 中，宏的使用不像在 Common Lisp 中那样广泛。Julia 支持宏的一种卫生（hygiene）形式。因为不同的表层语法，Julia 中没有 `COMMON-LISP:&BODY` 的等价形式。
- **所有的**函数都是通用的并且使用多重分派。函数的参数列表也无需遵循一样的模板，这让我们有了一个强大的范式：`do`。可选参数与关键字参数的处理方式不同。方法的歧义没有像在 Common Lisp 对象系统中那样得到解决，因此需要为交集定义更具体的方法。
- 符号不属于任何包，它**本身**也不包含任何值。`M.var` 会对 `M` 模块里的 `var` 符号求值。
- Julia 完全支持函数式编程风格，包括闭包等特性。但这并不是 Julia 的惯用风格。修改捕获变量时需要一些额外的变通以便提高性能。

Chapter 40

Unicode 输入表

在 Julia REPL 或其它编辑器中，可以像输入 LaTeX 符号一样，用 tab 补全下表列出的 Unicode 字符。在 REPL 中，可以先按 ? 进入帮助模式，然后将 Unicode 字符复制粘贴进去，一般在文档开头就会写输入方式。

Warning

此表第二列可能会缺失一些字符，对某些字符的显示效果也可能会与在 Julia REPL 中不一致。如果发生了这种状况，强烈建议用户检查一下浏览器或 REPL 的字体设置，目前已知很多字体都有显示问题。

Chapter 41

Command-line Interface

41.1 Using arguments inside scripts

When running a script using `julia`, you can pass additional arguments to your script:

```
$ julia script.jl arg1 arg2...
```

These additional command-line arguments are passed in the global constant `ARGS`. The name of the script itself is passed in as the global `PROGRAM_FILE`. Note that `ARGS` is also set when a Julia expression is given using the `-e` option on the command line (see the `julia` help output below) but `PROGRAM_FILE` will be empty. For example, to just print the arguments given to a script, you could do this:

```
$ julia -e 'println(PROGRAM_FILE); for x in ARGS; println(x); end' foo bar

foo
bar
```

Or you could put that code into a script and run it:

```
$ echo 'println(PROGRAM_FILE); for x in ARGS; println(x); end' > script.jl
$ julia script.jl foo bar
script.jl
foo
bar
```

The `--` delimiter can be used to separate command-line arguments intended for the script file from arguments intended for Julia:

```
$ julia --color=yes -O -- script.jl arg1 arg2..
```

See also Scripting for more information on writing Julia scripts.

41.2 Parallel mode

Julia can be started in parallel mode with either the `-p` or the `--machine-file` options. `-p n` will launch an additional `n` worker processes, while `--machine-file file` will launch a worker for each line in file `file`.

The machines defined in `file` must be accessible via a password-less `ssh` login, with Julia installed at the same location as the current host. Each machine definition takes the form `[count*][user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user, `port` to the standard ssh port. `count` is the number of workers to spawn on the node, and defaults to 1. The optional `bind-to bind_addr[:port]` specifies the IP address and port that other workers should use to connect to this worker.

41.3 Startup file

If you have code that you want executed whenever Julia is run, you can put it in `~/.julia/config/startup.jl`:

```
$ echo 'println("Greetings! 你好! 🄵🄵🄵🄵🄵?")' > ~/.julia/config/startup.jl
$ julia
Greetings! 你好! 🄵🄵🄵🄵🄵?

...
```

Note that although you should have a `~/.julia` directory once you've run Julia for the first time, you may need to create the `~/.julia/config` folder and the `~/.julia/config/startup.jl` file if you use it.

To have startup code run only in The Julia REPL (and not when `julia` is *e.g.* run on a script), use `atreplinit` in `startup.jl`:

```
atreplinit() do repl
    # ...
end
```

41.4 Command-line switches for Julia

There are various ways to run Julia code and provide options, similar to those available for the `perl` and `ruby` programs:

```
julia [switches] -- [programfile] [args...]
```

The following is a complete list of command-line switches available when launching julia (a '*' marks the default value, if applicable; settings marked '($)' may trigger package precompilation):

Julia 1.1

In Julia 1.0, the default `--project=@.` option did not search up from the root directory of a Git repository for the `Project.toml` file. From Julia 1.1 forward, it does.

Switch	Description
-v, --version	Display version information
-h, --help	Print command-line options (this message).
--help-hidden	Uncommon options not shown by -h
--project[={<dir>|@.}]	Set <dir> as the home project/environment. The default @. option will search through parent directories until a Project.toml or JuliaProject.toml file is found.
-J, --sysimage <file>	Start up with the given system image file
-H, --home <dir>	Set location of julia executable
--startup-file={yes*|no}	Load JULIA_DEPOT_PATH/config/startup.jl; if JULIA_DEPOT_PATH environment variable is unset, load ~/.julia/config/startup.jl
--handle-signals={yes*|no}	Enable or disable Julia's default signal handlers
--sysimage-native-code={yes*|no}	Use native code from system image if available
--compiled-modules={yes*|no}	Enable or disable incremental precompilation of modules
--pkgimages={yes*|no}	Enable or disable usage of native code caching in the form of pkgimages
-e, --eval <expr>	Evaluate <expr>
-E, --print <expr>	Evaluate <expr> and display the result
-L, --load <file>	Load <file> immediately on all processors
-t, --threads {N|auto}	Enable N threads; auto tries to infer a useful default number of threads to use but the exact behavior might change in the future. Currently, auto uses the number of CPUs assigned to this julia process based on the OS-specific affinity assignment interface, if supported (Linux and Windows). If this is not supported (macOS) or process affinity is not configured, it uses the number of CPU threads.
--gcthreads=N[,M]	Use N threads for the mark phase of GC and M (0 or 1) threads for the concurrent sweeping phase of GC. N is set to half of the number of compute threads and M is set to 0 if unspecified.
-p, --procs {N|auto}	Integer value N launches N additional local worker processes; auto launches as many workers as the number of local CPU threads (logical cores)
--machine-file <file>	Run processes on hosts listed in <file>
-i	Interactive mode; REPL runs and isinteractive() is true
-q, --quiet	Quiet startup: no banner, suppress REPL warnings
--banner={yes|no|auto*}	Enable or disable startup banner
--color={yes|no|auto*}	Enable or disable color text
--history-file={yes*|no}	Load or save history
--depwarn={yes|no*|error}	Enable or disable syntax and method deprecation warnings (error turns warnings into errors)
--warn-overwrite={yes|no*}	Enable or disable method overwrite warnings
--warn-scope={yes*|no}	Enable or disable warning for ambiguous top-level scope
-C, --cpu-target <target>	Limit usage of CPU features up to <target>; set to help to see the available options
-O, --optimize={0,1,2*,3}	Set the optimization level (level is 3 if -O is used without a level) ($)
--min-optlevel={0*,1,2,3}	Set the lower bound on per-module optimization
-g, --debug-info={0,1*,2}	Set the level of debug info generation (level is 2 if -g is used without a level) ($)
--inline={yes|no}	Control whether inlining is permitted, including overriding @inline declarations
--check-bounds={yes|no|auto*}	Emit bounds checks always, never, or respect @inbounds declarations ($)
--math-mode={ieee,fast}	Disallow or enable unsafe floating point optimizations (overrides @fastmath declaration)
--code-coverage[={none*|user|all}]	Count executions of source lines (omitting setting is equivalent to user)
--code-coverage=@<path>	Count executions but only in files that fall under the given file path/directory. The @ prefix is required to select this option. A @ with no path will track the current directory.
--code-coverage=tracefile.info	Append coverage information to the LCOV tracefile (filename supports format tokens)

Part III

Base

Chapter 42

基本功能

42.1 介绍

Julia Base 中包含一系列适用于科学及数值计算的函数和宏，但也可以用于通用编程。其它功能则由 Julia 生态圈中的各种库来提供。函数按主题划分如下：

一些通用的提示：

- 可以通过 `Import Module` 导入想要使用的模块，并利用 `Module.fn(x)` 语句来实现对模块内函数的调用。
- 此外，`using Module` 语句会将名为 `Module` 的模块中的所有可调函数引入当前的命名空间。
- 按照约定，名字以感叹号（`!`）结尾的函数会改变其输入参数的内容。一些函数同时拥有改变参数（例如 `sort!`）和不改变参数（`sort`）的版本

The behaviors of Base and standard libraries are stable as defined in SemVer only if they are documented; i.e., included in the Julia documentation and not marked as unstable. See API FAQ for more information.

The behaviors of Base and standard libraries are stable as defined in SemVer only if they are documented; i.e., included in the Julia documentation and not marked as unstable. See API FAQ for more information.

42.2 Getting Around

`Base.exit` – Function.

```
exit(code=0)
```

Stop the program with an exit code. The default exit code is zero, indicating that the program completed successfully. In an interactive session, `exit()` can be called with the keyboard shortcut `^D`.

source

`Base.atexit` – Function.

```
atexit(f)
```

Register a zero- or one-argument function `f()` to be called at process exit. `atexit()` hooks are called in last in first out (LIFO) order and run before object finalizers.

If `f` has a method defined for one integer argument, it will be called as `f(n::Int32)`, where `n` is the current exit code, otherwise it will be called as `f()`.

> **Julia 1.9**
>
> The one-argument form requires Julia 1.9

Exit hooks are allowed to call `exit(n)`, in which case Julia will exit with exit code `n` (instead of the original exit code). If more than one exit hook calls `exit(n)`, then Julia will exit with the exit code corresponding to the last called exit hook that calls `exit(n)`. (Because exit hooks are called in LIFO order, "last called" is equivalent to "first registered".)

Note: Once all exit hooks have been called, no more exit hooks can be registered, and any call to `atexit(f)` after all hooks have completed will throw an exception. This situation may occur if you are registering exit hooks from background Tasks that may still be executing concurrently during shutdown.

source

`Base.isinteractive` – Function.

```
isinteractive() -> Bool
```

Determine whether Julia is running an interactive session.

source

`Base.summarysize` – Function.

```
Base.summarysize(obj; exclude=Union{...}, chargeall=Union{...}) -> Int
```

Compute the amount of memory, in bytes, used by all unique objects reachable from the argument.

Keyword Arguments

- `exclude`: specifies the types of objects to exclude from the traversal.
- `chargeall`: specifies the types of objects to always charge the size of all of their fields, even if those fields would normally be excluded.

See also `sizeof`.

Examples

```
julia> Base.summarysize(1.0)
8

julia> Base.summarysize(Ref(rand(100)))
848

julia> sizeof(Ref(rand(100)))
8
```

source

Base.__precompile__ – Function.

```
__precompile__(isprecompilable::Bool)
```

Specify whether the file calling this function is precompilable, defaulting to `true`. If a module or file is *not* safely precompilable, it should call `__precompile__(false)` in order to throw an error if Julia attempts to precompile it.

source

Base.include – Function.

```
Base.include([mapexpr::Function,] m::Module, path::AbstractString)
```

Evaluate the contents of the input source file in the global scope of module `m`. Every module (except those defined with `baremodule`) has its own definition of `include` omitting the `m` argument, which evaluates the file in that module. Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

The optional first argument `mapexpr` can be used to transform the included code before it is evaluated: for each parsed expression `expr` in `path`, the `include` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

Julia 1.5

Julia 1.5 is required for passing the `mapexpr` argument.

source

Base.MainInclude.include – Function.

```
include([mapexpr::Function,] path::AbstractString)
```

Evaluate the contents of the input source file in the global scope of the containing module. Every module (except those defined with `baremodule`) has its own definition of `include`, which evaluates the file in that module. Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files. The argument `path` is normalized using `normpath` which will resolve relative path tokens such as `..` and convert `/` to the appropriate path separator.

The optional first argument `mapexpr` can be used to transform the included code before it is evaluated: for each parsed expression `expr` in `path`, the `include` function actually evaluates `mapexpr(expr)`. If it is omitted, `mapexpr` defaults to `identity`.

Use `Base.include` to evaluate a file into another module.

> **Julia 1.5**
>
> Julia 1.5 is required for passing the mapexpr argument.

source

Base.include_string – Function.

```
include_string([mapexpr::Function,] m::Module, code::AbstractString,
↪   filename::AbstractString="string")
```

Like include, except reads code from the given string rather than from a file.

The optional first argument mapexpr can be used to transform the included code before it is evaluated: for each parsed expression expr in code, the include_string function actually evaluates mapexpr(expr). If it is omitted, mapexpr defaults to identity.

> **Julia 1.5**
>
> Julia 1.5 is required for passing the mapexpr argument.

source

Base.include_dependency – Function.

```
include_dependency(path::AbstractString)
```

In a module, declare that the file, directory, or symbolic link specified by path (relative or absolute) is a dependency for precompilation; that is, the module will need to be recompiled if the modification time of path changes.

This is only needed if your module depends on a path that is not used via include. It has no effect outside of compilation.

source

__init__ – Keyword.

```
__init__
```

The __init__() function in a module executes immediately *after* the module is loaded at runtime for the first time. It is called once, after all other statements in the module have been executed. Because it is called after fully importing the module, __init__ functions of submodules will be executed first. Two typical uses of __init__ are calling runtime initialization functions of external C libraries and initializing global constants that involve pointers returned by external libraries. See the manual section about modules for more details.

Examples

```
const foo_data_ptr = Ref{Ptr{Cvoid}}(0)
function __init__()
    ccall((:foo_init, :libfoo), Cvoid, ())
    foo_data_ptr[] = ccall((:foo_data, :libfoo), Ptr{Cvoid}, ())
    nothing
end
```

source

Base.which – Method.

```
which(f, types)
```

Returns the method of f (a Method object) that would be called for arguments of the given types.

If types is an abstract type, then the method that would be called by invoke is returned.

See also: parentmodule, and @which and @edit in InteractiveUtils.

source

Base.methods – Function.

```
methods(f, [types], [module])
```

Return the method table for f.

If types is specified, return an array of methods whose types match. If module is specified, return an array of methods defined in that module. A list of modules can also be specified as an array.

> **Julia 1.4**
>
> At least Julia 1.4 is required for specifying a module.

See also: which and @which.

source

Base.@show – Macro.

```
@show exs...
```

Prints one or more expressions, and their results, to stdout, and returns the last result.

See also: show, @info, println.

Examples

```
julia> x = @show 1+2
1 + 2 = 3
3

julia> @show x^2 x/2;
x ^ 2 = 9
x / 2 = 1.5
```

source

Base.MainInclude.ans – Constant.

```
ans
```

A variable referring to the last computed value, automatically imported to the interactive prompt.

source

Base.MainInclude.err – Constant.

```
err
```

A variable referring to the last thrown errors, automatically imported to the interactive prompt. The thrown errors are collected in a stack of exceptions.

source

Base.active_project – Function.

```
active_project()
```

Return the path of the active Project.toml file. See also Base.set_active_project.

source

Base.set_active_project – Function.

```
set_active_project(projfile::Union{AbstractString,Nothing})
```

Set the active Project.toml file to projfile. See also Base.active_project.

Julia 1.8

This function requires at least Julia 1.8.

source

42.3 Keywords

This is the list of reserved keywords in Julia: `baremodule`, `begin`, `break`, `catch`, `const`, `continue`, `do`, `else`, `elseif`, `end`, `export`, `false`, `finally`, `for`, `function`, `global`, `if`, `import`, `let`, `local`, `macro`, `module`, `quote`, `return`, `struct`, `true`, `try`, `using`, `while`. Those keywords are not allowed to be used as variable names.

The following two-word sequences are reserved: `abstract type`, `mutable struct`, `primitive type`. However, you can create variables with names: `abstract`, `mutable`, `primitive` and `type`.

Finally: `where` is parsed as an infix operator for writing parametric method and type definitions; `in` and `isa` are parsed as infix operators; `outer` is parsed as a keyword when used to modify the scope of a variable in an iteration specification of a `for` loop; and `as` is used as a keyword to rename an identifier brought into scope by `import` or `using`. Creation of variables named `where`, `in`, `isa`, `outer` and `as` is allowed, though.

`module` – Keyword.

```
module
```

`module` declares a `Module`, which is a separate global variable workspace. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting). Modules allow you to create top-level definitions without worrying about name conflicts when your code is used together with somebody else’s. See the manual section about modules for more details.

Examples

```
module Foo
import Base.show
export MyType, foo

struct MyType
    x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1
show(io::IO, a::MyType) = print(io, "MyType $(a.x)")
end
```

source

`export` – Keyword.

```
export
```

`export` is used within modules to tell Julia which functions should be made available to the user. For example: `export foo` makes the name `foo` available when `using` the module. See the manual section about modules for details.

source

`import` – Keyword.

```
import
```

`import Foo` will load the module or package `Foo`. Names from the imported `Foo` module can be accessed with dot syntax (e.g. `Foo.foo` to access the name `foo`). See the manual section about modules for details.

source

`using` – Keyword.

```
using
```

`using Foo` will load the module or package `Foo` and make its `export`ed names available for direct use. Names can also be used via dot syntax (e.g. `Foo.foo` to access the name `foo`), whether they are `export`ed or not. See the manual section about modules for details.

Note

When two or more packages/modules export a name and that name does not refer to the same thing in each of the packages, and the packages are loaded via `using` without an explicit list of names, it is an error to reference that name without qualification. It is thus recommended that code intended to be forward-compatible with future versions of its dependencies and of Julia, e.g., code in released packages, list the names it uses from each loaded package, e.g., `using Foo: Foo, f` rather than `using Foo`.

source

`as` – Keyword.

```
as
```

`as` is used as a keyword to rename an identifier brought into scope by `import` or `using`, for the purpose of working around name conflicts as well as for shortening names. (Outside of `import` or `using` statements, `as` is not a keyword and can be used as an ordinary identifier.)

`import LinearAlgebra as LA` brings the imported `LinearAlgebra` standard library into scope as `LA`.

`import LinearAlgebra: eigen as eig, cholesky as chol` brings the `eigen` and `cholesky` methods from `LinearAlgebra` into scope as `eig` and `chol` respectively.

`as` works with `using` only when individual identifiers are brought into scope. For example, `using LinearAlgebra: eigen as eig` or `using LinearAlgebra: eigen as eig, cholesky as chol` works, but `using LinearAlgebra as LA` is invalid syntax, since it is nonsensical to rename *all* exported names from `LinearAlgebra` to `LA`.

source

`baremodule` – Keyword.

```
baremodule
```

baremodule declares a module that does not contain using Base or local definitions of eval and include. It does still import Core. In other words,

```
module Mod

...

end
```

is equivalent to

```
baremodule Mod

using Base

eval(x) = Core.eval(Mod, x)
include(p) = Base.include(Mod, p)

...

end
```

source

function – Keyword.

```
function
```

Functions are defined with the function keyword:

```
function add(a, b)
    return a + b
end
```

Or the short form notation:

```
add(a, b) = a + b
```

The use of the return keyword is exactly the same as in other languages, but is often optional. A function without an explicit return statement will return the last expression in the function body.

source

macro – Keyword.

```
macro
```

macro defines a method for inserting generated code into a program. A macro maps a sequence of argument expressions to a returned expression, and the resulting expression is substituted directly into the program at the point where the macro is invoked. Macros are a way to run generated code without calling eval, since the generated code instead simply becomes part of the surrounding program. Macro arguments may include expressions, literal values, and symbols. Macros can be defined for variable number of arguments (varargs), but do not accept keyword arguments. Every macro also implicitly gets passed the arguments __source__, which contains the line number and file name the macro is called from, and __module__, which is the module the macro is expanded in.

See the manual section on Metaprogramming for more information about how to write a macro.

Examples

```
julia> macro sayhello(name)
           return :( println("Hello, ", $name, "!") )
       end
@sayhello (macro with 1 method)

julia> @sayhello "Charlie"
Hello, Charlie!

julia> macro saylots(x...)
           return :( println("Say: ", $(x...)) )
       end
@saylots (macro with 1 method)

julia> @saylots "hey " "there " "friend"
Say: hey there friend
```

source

return – Keyword.

```
return
```

return x causes the enclosing function to exit early, passing the given value x back to its caller. return by itself with no value is equivalent to return nothing (see nothing).

```
function compare(a, b)
    a == b && return "equal to"
    a < b ? "less than" : "greater than"
end
```

In general you can place a return statement anywhere within a function body, including within deeply nested loops or conditionals, but be careful with do blocks. For example:

```
function test1(xs)
    for x in xs
        iseven(x) && return 2x
    end
end
```

```
function test2(xs)
    map(xs) do x
        iseven(x) && return 2x
        x
    end
end
```

In the first example, the return breaks out of `test1` as soon as it hits an even number, so `test1([5,6,7])` returns 12.

You might expect the second example to behave the same way, but in fact the `return` there only breaks out of the *inner* function (inside the `do` block) and gives a value back to `map`. `test2([5,6,7])` then returns `[5,12,7]`.

When used in a top-level expression (i.e. outside any function), `return` causes the entire current top-level expression to terminate early.

source

do – Keyword.

```
do
```

Create an anonymous function and pass it as the first argument to a function call. For example:

```
map(1:10) do x
    2x
end
```

is equivalent to `map(x->2x, 1:10)`.

Use multiple arguments like so:

```
map(1:10, 11:20) do x, y
    x + y
end
```

source

begin – Keyword.

```
begin
```

`begin...end` denotes a block of code.

```
begin
    println("Hello, ")
    println("World!")
end
```

Usually begin will not be necessary, since keywords such as `function` and `let` implicitly begin blocks of code. See also `;`.

begin may also be used when indexing to represent the first index of a collection or the first index of a dimension of an array. For example, a[begin] is the first element of an array a.

Julia 1.4

Use of begin as an index requires Julia 1.4 or later.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> A[begin, :]
2-element Array{Int64,1}:
 1
 2
```

source

end – Keyword.

```
end
```

end marks the conclusion of a block of expressions, for example `module`, `struct`, `mutable struct`, `begin`, `let`, `for` etc.

end may also be used when indexing to represent the last index of a collection or the last index of a dimension of an array.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64, 2}:
 1  2
 3  4

julia> A[end, :]
2-element Array{Int64, 1}:
 3
 4
```

source

let – Keyword.

```
let
```

`let` blocks create a new hard scope and optionally introduce new local bindings.

Just like the other scope constructs, `let` blocks define the block of code where newly introduced local variables are accessible. Additionally, the syntax has a special meaning for comma-separated assignments and variable names that may optionally appear on the same line as the `let`:

```
let var1 = value1, var2, var3 = value3
    code
end
```

The variables introduced on this line are local to the `let` block and the assignments are evaluated in order, with each right-hand side evaluated in the scope without considering the name on the left-hand side. Therefore it makes sense to write something like `let x = x`, since the two `x` variables are distinct with the left-hand side locally shadowing the `x` from the outer scope. This can even be a useful idiom as new local variables are freshly created each time local scopes are entered, but this is only observable in the case of variables that outlive their scope via closures. A `let` variable without an assignment, such as `var2` in the example above, declares a new local variable that is not yet bound to a value.

By contrast, `begin` blocks also group multiple expressions together but do not introduce scope or have the special assignment syntax.

Examples

In the function below, there is a single `x` that is iteratively updated three times by the `map`. The closures returned all reference that one `x` at its final value:

```
julia> function test_outer_x()
           x = 0
           map(1:3) do _
               x += 1
               return ()->x
           end
       end
test_outer_x (generic function with 1 method)

julia> [f() for f in test_outer_x()]
3-element Vector{Int64}:
 3
 3
 3
```

If, however, we add a `let` block that introduces a *new* local variable we will end up with three distinct variables being captured (one at each iteration) even though we chose to use (shadow) the same name.

```
julia> function test_let_x()
           x = 0
           map(1:3) do _
               x += 1
               let x = x
                   return ()->x
```

```
            end
        end
    end
test_let_x (generic function with 1 method)

julia> [f() for f in test_let_x()]
3-element Vector{Int64}:
 1
 2
 3
```

All scope constructs that introduce new local variables behave this way when repeatedly run; the distinctive feature of `let` is its ability to succinctly declare new `locals` that may shadow outer variables of the same name. For example, directly using the argument of the do function similarly captures three distinct variables:

```
julia> function test_do_x()
           map(1:3) do x
               return ()->x
           end
       end
test_do_x (generic function with 1 method)

julia> [f() for f in test_do_x()]
3-element Vector{Int64}:
 1
 2
 3
```

source

`if` – Keyword.

```
if/elseif/else
```

`if/elseif/else` performs conditional evaluation, which allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the `if/elseif/else` conditional syntax:

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

If the condition expression `x < y` is true, then the corresponding block is evaluated; otherwise the condition expression `x > y` is evaluated, and if it is true, the corresponding block is evaluated; if neither expression

is true, the `else` block is evaluated. The `elseif` and `else` blocks are optional, and as many `elseif` blocks as desired can be used.

In contrast to some other languages conditions must be of type `Bool`. It does not suffice for conditions to be convertible to `Bool`.

```
julia> if 1 end
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

source

`for` – Keyword.

```
for
```

`for` loops repeatedly evaluate a block of statements while iterating over a sequence of values.

The iteration variable is always a new variable, even if a variable of the same name exists in the enclosing scope. Use `outer` to reuse an existing local variable for iteration.

Examples

```
julia> for i in [1, 4, 0]
           println(i)
       end
1
4
0
```

source

`while` – Keyword.

```
while
```

`while` loops repeatedly evaluate a conditional expression, and continue evaluating the body of the while loop as long as the expression remains true. If the condition expression is false when the while loop is first reached, the body is never evaluated.

Examples

```
julia> i = 1
1

julia> while i < 5
           println(i)
           global i += 1
       end
1
2
3
4
```

source

`break` – Keyword.

```
break
```

Break out of a loop immediately.

Examples

```
julia> i = 0
0

julia> while true
           global i += 1
           i > 5 && break
           println(i)
       end
1
2
3
4
5
```

source

`continue` – Keyword.

```
continue
```

Skip the rest of the current loop iteration.

Examples

```
julia> for i = 1:6
           iseven(i) && continue
           println(i)
       end
1
3
5
```

source

`try` – Keyword.

```
try/catch
```

A try/catch statement allows intercepting errors (exceptions) thrown by throw so that program execution can continue. For example, the following code attempts to write a file, but warns the user and proceeds instead of terminating execution if the file cannot be written:

```
try
    open("/danger", "w") do f
        println(f, "Hello")
    end
catch
    @warn "Could not write file."
end
```

or, when the file cannot be read into a variable:

```
lines = try
    open("/danger", "r") do f
        readlines(f)
    end
catch
    @warn "File not found."
end
```

The syntax catch e (where e is any variable) assigns the thrown exception object to the given variable within the catch block.

The power of the try/catch construct lies in the ability to unwind a deeply nested computation immediately to a much higher level in the stack of calling functions.

source

finally – Keyword.

```
finally
```

Run some code when a given block of code exits, regardless of how it exits. For example, here is how we can guarantee that an opened file is closed:

```
f = open("file")
try
    operate_on_file(f)
finally
    close(f)
end
```

When control leaves the try block (for example, due to a return, or just finishing normally), close(f) will be executed. If the try block exits due to an exception, the exception will continue propagating. A catch block may be combined with try and finally as well. In this case the finally block will run after catch has handled the error.

source

quote – Keyword.

```
quote
```

`quote` creates multiple expression objects in a block without using the explicit `Expr` constructor. For example:

```
ex = quote
    x = 1
    y = 2
    x + y
end
```

Unlike the other means of quoting, `:( ... )`, this form introduces `QuoteNode` elements to the expression tree, which must be considered when directly manipulating the tree. For other purposes, `:( ... )` and `quote .. end` blocks are treated identically.

source

`local` – Keyword.

```
local
```

`local` introduces a new local variable. See the manual section on variable scoping for more information.

Examples

```
julia> function foo(n)
           x = 0
           for i = 1:n
               local x # introduce a loop-local x
               x = i
           end
           x
       end
foo (generic function with 1 method)

julia> foo(10)
0
```

source

`global` – Keyword.

```
global
```

`global x` makes `x` in the current scope and its inner scopes refer to the global variable of that name. See the manual section on variable scoping for more information.

Examples

```
julia> z = 3
3

julia> function foo()
           global z = 6 # use the z variable defined outside foo
       end
foo (generic function with 1 method)

julia> foo()
6

julia> z
6
```

source

`outer` – Keyword.

```
for outer
```

Reuse an existing local variable for iteration in a `for` loop.

See the manual section on variable scoping for more information.

See also `for`.

Examples

```
julia> function f()
           i = 0
           for i = 1:3
               # empty
           end
           return i
       end;

julia> f()
0
```

```
julia> function f()
           i = 0
           for outer i = 1:3
               # empty
           end
           return i
       end;

julia> f()
3
```

```
julia> i = 0 # global variable
       for outer i = 1:3
       end
ERROR: syntax: no outer local variable declaration exists for "for outer"
[...]
```

source

`const` – Keyword.

```
const
```

`const` is used to declare global variables whose values will not change. In almost all code (and particularly performance sensitive code) global variables should be declared constant in this way.

```
const x = 5
```

Multiple variables can be declared within a single `const`:

```
const y, z = 7, 11
```

Note that `const` only applies to one `=` operation, therefore `const x = y = 1` declares `x` to be constant but not `y`. On the other hand, `const x = const y = 1` declares both `x` and `y` constant.

Note that "constant-ness" does not extend into mutable containers; only the association between a variable and its value is constant. If `x` is an array or dictionary (for example) you can still modify, add, or remove elements.

In some cases changing the value of a `const` variable gives a warning instead of an error. However, this can produce unpredictable behavior or corrupt the state of your program, and so should be avoided. This feature is intended only for convenience during interactive use.

source

`struct` – Keyword.

```
struct
```

The most commonly used kind of type in Julia is a struct, specified as a name and a set of fields.

```
struct Point
    x
    y
end
```

Fields can have type restrictions, which may be parameterized:

```
struct Point{X}
    x::X
    y::Float64
end
```

A struct can also declare an abstract super type via `<:` syntax:

```
struct Point <: AbstractPoint
    x
    y
end
```

`struct`s are immutable by default; an instance of one of these types cannot be modified after construction. Use `mutable struct` instead to declare a type whose instances can be modified.

See the manual section on Composite Types for more details, such as how to define constructors.

source

`mutable struct` – Keyword.

```
mutable struct
```

`mutable struct` is similar to `struct`, but additionally allows the fields of the type to be set after construction.

Individual fields of a mutable struct can be marked as `const` to make them immutable:

```
mutable struct Baz
    a::Int
    const b::Float64
end
```

Julia 1.8

The `const` keyword for fields of mutable structs requires at least Julia 1.8.

See the manual section on Composite Types for more information.

source

`Base.@kwdef` – Macro.

```
@kwdef typedef
```

This is a helper macro that automatically defines a keyword-based constructor for the type declared in the expression `typedef`, which must be a `struct` or `mutable struct` expression. The default argument is supplied by declaring fields of the form `field::T = default` or `field = default`. If no default is provided then the keyword argument becomes a required keyword argument in the resulting type constructor.

Inner constructors can still be defined, but at least one should accept arguments in the same form as the default inner constructor (i.e. one positional argument per field) in order to function correctly with the keyword outer constructor.

Julia 1.1

Base.@kwdef for parametric structs, and structs with supertypes requires at least Julia 1.1.

Julia 1.9

This macro is exported as of Julia 1.9.

Examples

```
julia> @kwdef struct Foo
           a::Int = 1         # specified default
           b::String          # required keyword
       end
Foo

julia> Foo(b="hi")
Foo(1, "hi")

julia> Foo()
ERROR: UndefKeywordError: keyword argument `b` not assigned
Stacktrace:
[...]
```

source

abstract type – Keyword.

```
abstract type
```

abstract type declares a type that cannot be instantiated, and serves only as a node in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. Abstract types form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations. For example:

```
abstract type Number end
abstract type Real <: Number end
```

Number has no supertype, whereas Real is an abstract subtype of Number.

source

primitive type – Keyword.

```
primitive type
```

`primitive type` declares a concrete type whose data consists only of a series of bits. Classic examples of primitive types are integers and floating-point values. Some example built-in primitive type declarations:

```
primitive type Char 32 end
primitive type Bool <: Integer 8 end
```

The number after the name indicates how many bits of storage the type requires. Currently, only sizes that are multiples of 8 bits are supported. The `Bool` declaration shows how a primitive type can be optionally declared to be a subtype of some supertype.

source

`where` – Keyword.

```
where
```

The `where` keyword creates a `UnionAll` type, which may be thought of as an iterated union of other types, over all values of some variable. For example `Vector{T} where T<:Real` includes all `Vector`s where the element type is some kind of `Real` number.

The variable bound defaults to `Any` if it is omitted:

```
Vector{T} where T    # short for `where T<:Any`
```

Variables can also have lower bounds:

```
Vector{T} where T>:Int
Vector{T} where Int<:T<:Real
```

There is also a concise syntax for nested `where` expressions. For example, this:

```
Pair{T, S} where S<:Array{T} where T<:Number
```

can be shortened to:

```
Pair{T, S} where {T<:Number, S<:Array{T}}
```

This form is often found on method signatures.

Note that in this form, the variables are listed outermost-first. This matches the order in which variables are substituted when a type is ”applied” to parameter values using the syntax `T{p1, p2, ...}`.

source

`...` – Keyword.

```
...
```

The ”splat” operator, `...`, represents a sequence of arguments. `...` can be used in function definitions, to indicate that the function accepts an arbitrary number of arguments. `...` can also be used to apply a function to a sequence of arguments.

Examples

```
julia> add(xs...) = reduce(+, xs)
add (generic function with 1 method)

julia> add(1, 2, 3, 4, 5)
15

julia> add([1, 2, 3]...)
6

julia> add(7, 1:100..., 1000:1100...)
111107
```

source

`;` – Keyword.

```
;
```

`;` has a similar role in Julia as in many C-like languages, and is used to delimit the end of the previous statement.

`;` is not necessary at the end of a line, but can be used to separate statements on a single line or to join statements into a single expression.

Adding `;` at the end of a line in the REPL will suppress printing the result of that expression.

In function declarations, and optionally in calls, `;` separates regular arguments from keywords.

In array literals, arguments separated by semicolons have their contents concatenated together. A separator made of a single `;` concatenates vertically (i.e. along the first dimension), `;;` concatenates horizontally (second dimension), `;;;` concatenates along the third dimension, etc. Such a separator can also be used in last position in the square brackets to add trailing dimensions of length 1.

A `;` in first position inside of parentheses can be used to construct a named tuple. The same `(; ...)` syntax on the left side of an assignment allows for property destructuring.

In the standard REPL, typing `;` on an empty line will switch to shell mode.

Examples

```
julia> function foo()
           x = "Hello, "; x *= "World!"
           return x
       end
foo (generic function with 1 method)
```

```
julia> bar() = (x = "Hello, Mars!"; return x)
bar (generic function with 1 method)

julia> foo();

julia> bar()
"Hello, Mars!"

julia> function plot(x, y; style="solid", width=1, color="black")
           ###
       end

julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> [1; 3;; 2; 4;;; 10*A]
2×2×2 Array{Int64, 3}:
[:, :, 1] =
 1  2
 3  4

[:, :, 2] =
 10  20
 30  40

julia> [2; 3;;;]
2×1×1 Array{Int64, 3}:
[:, :, 1] =
 2
 3

julia> nt = (; x=1) # without the ; or a trailing comma this would assign to x
(x = 1,)

julia> key = :a; c = 3;

julia> nt2 = (; key => 1, b=2, c, nt.x)
(a = 1, b = 2, c = 3, x = 1)

julia> (; b, x) = nt2; # set variables b and x using property destructuring

julia> b, x
(2, 1)

shell> echo hello
julia> ; # upon typing ;, the prompt changes (in place) to: shell>
hello
```

source

= – Keyword.

```
=
```

= is the assignment operator.

- For variable a and expression b, `a = b` makes a refer to the value of b.
- For functions `f(x)`, `f(x) = x` defines a new function constant `f`, or adds a new method to `f` if `f` is already defined; this usage is equivalent to `function f(x); x; end`.
- `a[i] = v` calls `setindex!(a,v,i)`.
- `a.b = c` calls `setproperty!(a,:b,c)`.
- Inside a function call, `f(a=b)` passes b as the value of keyword argument a.
- Inside parentheses with commas, `(a=1,)` constructs a `NamedTuple`.

Examples

Assigning a to b does not create a copy of b; instead use `copy` or `deepcopy`.

```
julia> b = [1]; a = b; b[1] = 2; a
1-element Array{Int64, 1}:
 2

julia> b = [1]; a = copy(b); b[1] = 2; a
1-element Array{Int64, 1}:
 1
```

Collections passed to functions are also not copied. Functions can modify (mutate) the contents of the objects their arguments refer to. (The names of functions which do this are conventionally suffixed with '!'.)

```
julia> function f!(x); x[:] .+= 1; end
f! (generic function with 1 method)

julia> a = [1]; f!(a); a
1-element Array{Int64, 1}:
 2
```

Assignment can operate on multiple variables in parallel, taking values from an iterable:

```
julia> a, b = 4, 5
(4, 5)

julia> a, b = 1:3
1:3

julia> a, b
(1, 2)
```

Assignment can operate on multiple variables in series, and will return the value of the right-hand-most expression:

```
julia> a = [1]; b = [2]; c = [3]; a = b = c
1-element Array{Int64, 1}:
 3

julia> b[1] = 2; a, b, c
([2], [2], [2])
```

Assignment at out-of-bounds indices does not grow a collection. If the collection is a `Vector` it can instead be grown with `push!` or `append!`.

```
julia> a = [1, 1]; a[3] = 2
ERROR: BoundsError: attempt to access 2-element Array{Int64, 1} at index [3]
[...]

julia> push!(a, 2, 3)
4-element Array{Int64, 1}:
 1
 1
 2
 3
```

Assigning `[]` does not eliminate elements from a collection; instead use `filter!`.

```
julia> a = collect(1:3); a[a .<= 1] = []
ERROR: DimensionMismatch: tried to assign 0 elements to 1 destinations
[...]

julia> filter!(x -> x > 1, a) # in-place & thus more efficient than a = a[a .> 1]
2-element Array{Int64, 1}:
 2
 3
```

source

`?:` – Keyword.

```
a ? b : c
```

Short form for conditionals; read "if a, evaluate b otherwise evaluate c". Also known as the ternary operator.

This syntax is equivalent to `if a; b else c end`, but is often used to emphasize the value b-or-c which is being used as part of a larger expression, rather than the side effects that evaluating b or c may have.

See the manual section on control flow for more details.

Examples

```
julia> x = 1; y = 2;

julia> x > y ? println("x is larger") : println("x is not larger")
```

```
x is not larger

julia> x > y ? "x is larger" : x == y ? "x and y are equal" : "y is larger"
"y is larger"
```

source

42.4 Standard Modules

Main – Module.

```
Main
```

Main is the top-level module, and Julia starts with Main set as the current module. Variables defined at the prompt go in Main, and varinfo lists variables in Main.

```
julia> @__MODULE__
Main
```

source

Core – Module.

```
Core
```

Core is the module that contains all identifiers considered "built in" to the language, i.e. part of the core language and not libraries. Every module implicitly specifies using Core, since you can't do anything without those definitions.

source

Base – Module.

```
Base
```

The base library of Julia. Base is a module that contains basic functionality (the contents of base/). All modules implicitly contain using Base, since this is needed in the vast majority of cases.

source

42.5 Base Submodules

Base.Broadcast – Module.

```
Base.Broadcast
```

Module containing the broadcasting implementation.

source

`Base.Docs` – Module.

```
Docs
```

The `Docs` module provides the `@doc` macro which can be used to set and retrieve documentation metadata for Julia objects.

Please see the manual section on documentation for more information.

source

`Base.Iterators` – Module.

Methods for working with Iterators.

source

`Base.Libc` – Module.

Interface to libc, the C standard library.

source

`Base.Meta` – Module.

Convenience functions for metaprogramming.

source

`Base.StackTraces` – Module.

Tools for collecting and manipulating stack traces. Mainly used for building errors.

source

`Base.Sys` – Module.

Provide methods for retrieving information about hardware and the operating system.

source

`Base.Threads` – Module.

Multithreading support.

source

`Base.GC` – Module.

```
Base.GC
```

Module with garbage collection utilities.

source

42.6 All Objects

`Core.:===` – Function.

```
===(x,y) -> Bool
≡(x,y) -> Bool
```

Determine whether x and y are identical, in the sense that no program could distinguish them. First the types of x and y are compared. If those are identical, mutable objects are compared by address in memory and immutable objects (such as numbers) are compared by contents at the bit level. This function is sometimes called "egal". It always returns a `Bool` value.

Examples

```
julia> a = [1, 2]; b = [1, 2];

julia> a == b
true

julia> a === b
false

julia> a === a
true
```

source

`Core.isa` – Function.

```
isa(x, type) -> Bool
```

Determine whether x is of the given `type`. Can also be used as an infix operator, e.g. `x isa type`.

Examples

```
julia> isa(1, Int)
true

julia> isa(1, Matrix)
false

julia> isa(1, Char)
false
```

```
julia> isa(1, Number)
true

julia> 1 isa Number
true
```

source

Base.isequal – Function.

```
isequal(x, y) -> Bool
```

Similar to ==, except for the treatment of floating point numbers and of missing values. isequal treats all floating-point NaN values as equal to each other, treats -0.0 as unequal to 0.0, and missing as equal to missing. Always returns a Bool value.

isequal is an equivalence relation - it is reflexive (=== implies isequal), symmetric (isequal(a, b) implies isequal(b, a)) and transitive (isequal(a, b) and isequal(b, c) implies isequal(a, c)).

Implementation

The default implementation of isequal calls ==, so a type that does not involve floating-point values generally only needs to define ==.

isequal is the comparison function used by hash tables (Dict). isequal(x,y) must imply that hash(x) == hash(y).

This typically means that types for which a custom == or isequal method exists must implement a corresponding hash method (and vice versa). Collections typically implement isequal by calling isequal recursively on all contents.

Furthermore, isequal is linked with isless, and they work together to define a fixed total ordering, where exactly one of isequal(x, y), isless(x, y), or isless(y, x) must be true (and the other two false).

Scalar types generally do not need to implement isequal separate from ==, unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on isnan, signbit, and ==).

Examples

```
julia> isequal([1., NaN], [1., NaN])
true

julia> [1., NaN] == [1., NaN]
false

julia> 0.0 == -0.0
true

julia> isequal(0.0, -0.0)
false

julia> missing == missing
missing
```

```
julia> isequal(missing, missing)
true
```

source

```
isequal(x)
```

Create a function that compares its argument to `x` using `isequal`, i.e. a function equivalent to `y -> isequal(y, x)`.

The returned function is of type `Base.Fix2{typeof(isequal)}`, which can be used to implement specialized methods.

source

`Base.isless` – Function.

```
isless(x, y)
```

Test whether `x` is less than `y`, according to a fixed total order (defined together with `isequal`). `isless` is not defined for pairs `(x, y)` of all types. However, if it is defined, it is expected to satisfy the following:

- If `isless(x, y)` is defined, then so is `isless(y, x)` and `isequal(x, y)`, and exactly one of those three yields `true`.
- The relation defined by `isless` is transitive, i.e., `isless(x, y) && isless(y, z)` implies `isless(x, z)`.

Values that are normally unordered, such as `NaN`, are ordered after regular values. `missing` values are ordered last.

This is the default comparison used by `sort!`.

Implementation

Non-numeric types with a total order should implement this function. Numeric types only need to implement it if they have special values such as `NaN`. Types with a partial order should implement `<`. See the documentation on Alternate Orderings for how to define alternate ordering methods that can be used in sorting and related functions.

Examples

```
julia> isless(1, 3)
true

julia> isless("Red", "Blue")
false
```

source

`Base.isunordered` – Function.

```
isunordered(x)
```

Return `true` if `x` is a value that is not orderable according to `<`, such as `NaN` or `missing`.

The values that evaluate to `true` with this predicate may be orderable with respect to other orderings such as `isless`.

> **Julia 1.7**
>
> This function requires Julia 1.7 or later.

source

`Base.ifelse` – Function.

```
ifelse(condition::Bool, x, y)
```

Return `x` if `condition` is `true`, otherwise return `y`. This differs from `?` or `if` in that it is an ordinary function, so all the arguments are evaluated first. In some cases, using `ifelse` instead of an `if` statement can eliminate the branch in generated code and provide higher performance in tight loops.

Examples

```
julia> ifelse(1 > 2, 1, 2)
2
```

source

`Core.typeassert` – Function.

```
typeassert(x, type)
```

Throw a `TypeError` unless `x isa type`. The syntax `x::type` calls this function.

Examples

```
julia> typeassert(2.5, Int)
ERROR: TypeError: in typeassert, expected Int64, got a value of type Float64
Stacktrace:
[...]
```

source

`Core.typeof` – Function.

```
typeof(x)
```

Get the concrete type of x.

See also `eltype`.

Examples

```
julia> a = 1//2;

julia> typeof(a)
Rational{Int64}

julia> M = [1 2; 3.5 4];

julia> typeof(M)
Matrix{Float64} (alias for Array{Float64, 2})
```

source

`Core.tuple` – Function.

```
tuple(xs...)
```

Construct a tuple of the given objects.

See also `Tuple`, `ntuple`, `NamedTuple`.

Examples

```
julia> tuple(1, 'b', pi)
(1, 'b', π)

julia> ans === (1, 'b', π)
true

julia> Tuple(Real[1, 2, pi])  # takes a collection
(1, 2, π)
```

source

`Base.ntuple` – Function.

```
ntuple(f::Function, n::Integer)
```

Create a tuple of length n, computing each element as `f(i)`, where `i` is the index of the element.

Examples

```
julia> ntuple(i -> 2*i, 4)
(2, 4, 6, 8)
```

source

```
ntuple(f, ::Val{N})
```

Create a tuple of length `N`, computing each element as `f(i)`, where `i` is the index of the element. By taking a `Val(N)` argument, it is possible that this version of ntuple may generate more efficient code than the version taking the length as an integer. But `ntuple(f, N)` is preferable to `ntuple(f, Val(N))` in cases where `N` cannot be determined at compile time.

Examples

```
julia> ntuple(i -> 2*i, Val(4))
(2, 4, 6, 8)
```

source

`Base.objectid` – Function.

```
objectid(x) -> UInt
```

Get a hash value for `x` based on object identity.

If `x === y` then `objectid(x) == objectid(y)`, and usually when `x !== y`, `objectid(x) != objectid(y)`.

See also `hash`, `IdDict`.

source

`Base.hash` – Function.

```
hash(x[, h::UInt]) -> UInt
```

Compute an integer hash code such that `isequal(x,y)` implies `hash(x)==hash(y)`. The optional second argument `h` is another hash code to be mixed with the result.

New types should implement the 2-argument form, typically by calling the 2-argument `hash` method recursively in order to mix hashes of the contents with each other (and with `h`). Typically, any type that implements `hash` should also implement its own `==` (hence `isequal`) to guarantee the property mentioned above.

The hash value may change when a new Julia process is started.

```
julia> a = hash(10)
0x95ea2955abd45275

julia> hash(10, a) # only use the output of another hash function as the second argument
0xd42bad54a8575b16
```

See also: `objectid`, `Dict`, `Set`.

source

`Base.finalizer` – Function.

```
finalizer(f, x)
```

Register a function `f(x)` to be called when there are no program-accessible references to x, and return x. The type of x must be a `mutable struct`, otherwise the function will throw.

`f` must not cause a task switch, which excludes most I/O operations such as `println`. Using the `@async` macro (to defer context switching to outside of the finalizer) or `ccall` to directly invoke IO functions in C may be helpful for debugging purposes.

Note that there is no guaranteed world age for the execution of `f`. It may be called in the world age in which the finalizer was registered or any later world age.

Examples

```
finalizer(my_mutable_struct) do x
    @async println("Finalizing $x.")
end

finalizer(my_mutable_struct) do x
    ccall(:jl_safe_printf, Cvoid, (Cstring, Cstring), "Finalizing %s.", repr(x))
end
```

A finalizer may be registered at object construction. In the following example note that we implicitly rely on the finalizer returning the newly created mutable struct x.

Example

```
mutable struct MyMutableStruct
    bar
    function MyMutableStruct(bar)
        x = new(bar)
        f(t) = @async println("Finalizing $t.")
        finalizer(f, x)
    end
end
```

source

`Base.finalize` – Function.

```
finalize(x)
```

Immediately run finalizers registered for object x.

source

`Base.copy` – Function.

```
copy(x)
```

Create a shallow copy of x: the outer structure is copied, but not all internal values. For example, copying an array produces a new array with identically-same elements as the original.

See also copy!, copyto!, deepcopy.

source

Base.deepcopy – Function.

```
deepcopy(x)
```

Create a deep copy of x: everything is copied recursively, resulting in a fully independent object. For example, deep-copying an array produces a new array whose elements are deep copies of the original elements. Calling deepcopy on an object should generally have the same effect as serializing and then deserializing it.

While it isn't normally necessary, user-defined types can override the default deepcopy behavior by defining a specialized version of the function deepcopy_internal(x::T, dict::IdDict) (which shouldn't otherwise be used), where T is the type to be specialized for, and dict keeps track of objects copied so far within the recursion. Within the definition, deepcopy_internal should be used in place of deepcopy, and the dict variable should be updated as appropriate before returning.

source

Base.getproperty – Function.

```
getproperty(value, name::Symbol)
getproperty(value, name::Symbol, order::Symbol)
```

The syntax a.b calls getproperty(a, :b). The syntax @atomic order a.b calls getproperty(a, :b, :order) and the syntax @atomic a.b calls getproperty(a, :b, :sequentially_consistent).

Examples

```
julia> struct MyType{T <: Number}
           x::T
       end

julia> function Base.getproperty(obj::MyType, sym::Symbol)
           if sym === :special
               return obj.x + 1
           else # fallback to getfield
               return getfield(obj, sym)
           end
       end

julia> obj = MyType(1);

julia> obj.special
2

julia> obj.x
1
```

One should overload `getproperty` only when necessary, as it can be confusing if the behavior of the syntax `obj.f` is unusual. Also note that using methods is often preferable. See also this style guide documentation for more information: Prefer exported methods over direct field access.

See also `getfield`, `propertynames` and `setproperty!`.

source

`Base.setproperty!` – Function.

```
setproperty!(value, name::Symbol, x)
setproperty!(value, name::Symbol, x, order::Symbol)
```

The syntax `a.b = c` calls `setproperty!(a, :b, c)`. The syntax `@atomic order a.b = c` calls `setproperty!(a, :b, c, :order)` and the syntax `@atomic a.b = c` calls `setproperty!(a, :b, c, :sequentially_consistent)`.

> **Julia 1.8**
>
> `setproperty!` on modules requires at least Julia 1.8.

See also `setfield!`, `propertynames` and `getproperty`.

source

`Base.replaceproperty!` – Function.

```
replaceproperty!(x, f::Symbol, expected, desired, success_order::Symbol=:not_atomic,
↪   fail_order::Symbol=success_order)
```

Perform a compare-and-swap operation on `x.f` from `expected` to `desired`, per egal. The syntax `@atomic_replace! x.f expected => desired` can be used instead of the function call form.

See also `replacefield!` and `setproperty!`.

source

`Base.swapproperty!` – Function.

```
swapproperty!(x, f::Symbol, v, order::Symbol=:not_atomic)
```

The syntax `@atomic a.b, _ = c, a.b` returns `(c, swapproperty!(a, :b, c, :sequentially_consistent))`, where there must be one `getproperty` expression common to both sides.

See also `swapfield!` and `setproperty!`.

source

`Base.modifyproperty!` – Function.

```
modifyproperty!(x, f::Symbol, op, v, order::Symbol=:not_atomic)
```

The syntax @atomic op(x.f, v) (and its equivalent @atomic x.f op v) returns modifyproperty!(x, :f, op, v, :sequentially_consistent), where the first argument must be a getproperty expression and is modified atomically.

Invocation of op(getproperty(x, f), v) must return a value that can be stored in the field f of the object x by default. In particular, unlike the default behavior of setproperty!, the convert function is not called automatically.

See also modifyfield! and setproperty!.

source

Base.propertynames – Function.

```
propertynames(x, private=false)
```

Get a tuple or a vector of the properties (x.property) of an object x. This is typically the same as fieldnames(typeof(x)), but types that overload getproperty should generally overload propertynames as well to get the properties of an instance of the type.

propertynames(x) may return only "public" property names that are part of the documented interface of x. If you want it to also return "private" property names intended for internal use, pass true for the optional second argument. REPL tab completion on x. shows only the private=false properties.

See also: hasproperty, hasfield.

source

Base.hasproperty – Function.

```
hasproperty(x, s::Symbol)
```

Return a boolean indicating whether the object x has s as one of its own properties.

Julia 1.2

This function requires at least Julia 1.2.

See also: propertynames, hasfield.

source

Core.getfield – Function.

```
getfield(value, name::Symbol, [order::Symbol])
getfield(value, i::Int, [order::Symbol])
```

Extract a field from a composite value by name or position. Optionally, an ordering can be defined for the operation. If the field was declared @atomic, the specification is strongly recommended to be compatible with the stores to that location. Otherwise, if not declared as @atomic, this parameter must be :not_atomic if specified. See also getproperty and fieldnames.

Examples

```
julia> a = 1//2
1//2

julia> getfield(a, :num)
1

julia> a.num
1

julia> getfield(a, 1)
1
```

source

Core.setfield! – Function.

```
setfield!(value, name::Symbol, x, [order::Symbol])
setfield!(value, i::Int, x, [order::Symbol])
```

Assign x to a named field in value of composite type. The value must be mutable and x must be a subtype of fieldtype(typeof(value), name). Additionally, an ordering can be specified for this operation. If the field was declared @atomic, this specification is mandatory. Otherwise, if not declared as @atomic, it must be :not_atomic if specified. See also setproperty!.

Examples

```
julia> mutable struct MyMutableStruct
           field::Int
       end

julia> a = MyMutableStruct(1);

julia> setfield!(a, :field, 2);

julia> getfield(a, :field)
2

julia> a = 1//2
1//2

julia> setfield!(a, :num, 3);
ERROR: setfield!: immutable struct of type Rational cannot be changed
```

source

Core.modifyfield! – Function.

```
modifyfield!(value, name::Symbol, op, x, [order::Symbol]) -> Pair
modifyfield!(value, i::Int, op, x, [order::Symbol]) -> Pair
```

These atomically perform the operations to get and set a field after applying the function op.

```
y = getfield(value, name)
z = op(y, x)
setfield!(value, name, z)
return y => z
```

If supported by the hardware (for example, atomic increment), this may be optimized to the appropriate hardware instruction, otherwise it'll use a loop.

source

`Core.replacefield!` – Function.

```
replacefield!(value, name::Symbol, expected, desired,
              [success_order::Symbol, [fail_order::Symbol=success_order]) -> (; old,
              ↪  success::Bool)
replacefield!(value, i::Int, expected, desired,
              [success_order::Symbol, [fail_order::Symbol=success_order]) -> (; old,
              ↪  success::Bool)
```

These atomically perform the operations to get and conditionally set a field to a given value.

```
y = getfield(value, name, fail_order)
ok = y === expected
if ok
    setfield!(value, name, desired, success_order)
end
return (; old = y, success = ok)
```

If supported by the hardware, this may be optimized to the appropriate hardware instruction, otherwise it'll use a loop.

source

`Core.swapfield!` – Function.

```
swapfield!(value, name::Symbol, x, [order::Symbol])
swapfield!(value, i::Int, x, [order::Symbol])
```

These atomically perform the operations to simultaneously get and set a field:

```
y = getfield(value, name)
setfield!(value, name, x)
return y
```

source

`Core.isdefined` – Function.

```
isdefined(m::Module, s::Symbol, [order::Symbol])
isdefined(object, s::Symbol, [order::Symbol])
isdefined(object, index::Int, [order::Symbol])
```

Tests whether a global variable or object field is defined. The arguments can be a module and a symbol or a composite object and field name (as a symbol) or index. Optionally, an ordering can be defined for the operation. If the field was declared `@atomic`, the specification is strongly recommended to be compatible with the stores to that location. Otherwise, if not declared as `@atomic`, this parameter must be `:not_atomic` if specified.

To test whether an array element is defined, use `isassigned` instead.

See also `@isdefined`.

Examples

```
julia> isdefined(Base, :sum)
true

julia> isdefined(Base, :NonExistentMethod)
false

julia> a = 1//2;

julia> isdefined(a, 2)
true

julia> isdefined(a, 3)
false

julia> isdefined(a, :num)
true

julia> isdefined(a, :numerator)
false
```

source

`Core.getglobal` – Function.

```
getglobal(module::Module, name::Symbol, [order::Symbol=:monotonic])
```

Retrieve the value of the binding `name` from the module `module`. Optionally, an atomic ordering can be defined for the operation, otherwise it defaults to monotonic.

While accessing module bindings using `getfield` is still supported to maintain compatibility, using `getglobal` should always be preferred since `getglobal` allows for control over atomic ordering (`getfield` is always monotonic) and better signifies the code's intent both to the user as well as the compiler.

Most users should not have to call this function directly —The `getproperty` function or corresponding syntax (i.e. `module.name`) should be preferred in all but few very specific use cases.

Julia 1.9

This function requires Julia 1.9 or later.

See also `getproperty` and `setglobal!`.

Examples

```
julia> a = 1
1

julia> module M
       a = 2
       end;

julia> getglobal(@__MODULE__, :a)
1

julia> getglobal(M, :a)
2
```

source

`Core.setglobal!` – Function.

```
setglobal!(module::Module, name::Symbol, x, [order::Symbol=:monotonic])
```

Set or change the value of the binding `name` in the module `module` to `x`. No type conversion is performed, so if a type has already been declared for the binding, `x` must be of appropriate type or an error is thrown.

Additionally, an atomic ordering can be specified for this operation, otherwise it defaults to monotonic.

Users will typically access this functionality through the `setproperty!` function or corresponding syntax (i.e. `module.name = x`) instead, so this is intended only for very specific use cases.

Julia 1.9

This function requires Julia 1.9 or later.

See also `setproperty!` and `getglobal`

Examples

```
julia> module M end;

julia> M.a  # same as `getglobal(M, :a)`
ERROR: UndefVarError: `a` not defined

julia> setglobal!(M, :a, 1)
1

julia> M.a
1
```

source

Base.@isdefined – Macro.

```
@isdefined s -> Bool
```

Tests whether variable s is defined in the current scope.

See also isdefined for field properties and isassigned for array indexes or haskey for other mappings.

Examples

```
julia> @isdefined newvar
false

julia> newvar = 1
1

julia> @isdefined newvar
true

julia> function f()
           println(@isdefined x)
           x = 3
           println(@isdefined x)
       end
f (generic function with 1 method)

julia> f()
false
true
```

source

Base.convert – Function.

```
convert(T, x)
```

Convert x to a value of type T.

If T is an Integer type, an InexactError will be raised if x is not representable by T, for example if x is not integer-valued, or is outside the range supported by T.

Examples

```
julia> convert(Int, 3.0)
3

julia> convert(Int, 3.5)
ERROR: InexactError: Int64(3.5)
Stacktrace:
[...]
```

If T is a AbstractFloat type, then it will return the closest value to x representable by T.

```
julia> x = 1/3
0.3333333333333333

julia> convert(Float32, x)
0.33333334f0

julia> convert(BigFloat, x)
0.333333333333333314829616256247390992939472198486328125
```

If T is a collection type and x a collection, the result of `convert(T, x)` may alias all or part of x.

```
julia> x = Int[1, 2, 3];

julia> y = convert(Vector{Int}, x);

julia> y === x
true
```

See also: round, trunc, oftype, reinterpret.

source

`Base.promote` – Function.

```
promote(xs...)
```

Convert all arguments to a common type, and return them all (as a tuple). If no arguments can be converted, an error is raised.

See also: promote_type, promote_rule.

Examples

```
julia> promote(Int8(1), Float16(4.5), Float32(4.1))
(1.0f0, 4.5f0, 4.1f0)

julia> promote_type(Int8, Float16, Float32)
Float32

julia> reduce(Base.promote_typejoin, (Int8, Float16, Float32))
Real

julia> promote(1, "x")
ERROR: promotion of types Int64 and String failed to change any arguments
[...]

julia> promote_type(Int, String)
Any
```

source

Base.oftype – Function.

```
oftype(x, y)
```

Convert y to the type of x i.e. `convert(typeof(x), y)`.

Examples

```
julia> x = 4;

julia> y = 3.;

julia> oftype(x, y)
3

julia> oftype(y, x)
4.0
```

source

Base.widen – Function.

```
widen(x)
```

If x is a type, return a "larger" type, defined so that arithmetic operations + and - are guaranteed not to overflow nor lose precision for any combination of values that type x can hold.

For fixed-size integer types less than 128 bits, `widen` will return a type with twice the number of bits.

If x is a value, it is converted to `widen(typeof(x))`.

Examples

```
julia> widen(Int32)
Int64

julia> widen(1.5f0)
1.5
```

source

Base.identity – Function.

```
identity(x)
```

The identity function. Returns its argument.

See also: `one`, `oneunit`, and `LinearAlgebra`'s `I`.

Examples

```
julia> identity("Well, what did you expect?")
"Well, what did you expect?"
```

source

Core.WeakRef – Type.

```
WeakRef(x)
```

w = WeakRef(x) constructs a weak reference to the Julia value x: although w contains a reference to x, it does not prevent x from being garbage collected. w.value is either x (if x has not been garbage-collected yet) or nothing (if x has been garbage-collected).

```
julia> x = "a string"
"a string"

julia> w = WeakRef(x)
WeakRef("a string")

julia> GC.gc()

julia> w           # a reference is maintained via `x`
WeakRef("a string")

julia> x = nothing # clear reference

julia> GC.gc()

julia> w
WeakRef(nothing)
```

source

42.7 Properties of Types

Type relations

Base.supertype – Function.

```
supertype(T::DataType)
```

Return the supertype of DataType T.

Examples

```
julia> supertype(Int32)
Signed
```

source

`Core.Type` – Type.

```
Core.Type{T}
```

`Core.Type` is an abstract type which has all type objects as its instances. The only instance of the singleton type `Core.Type{T}` is the object `T`.

Examples

```
julia> isa(Type{Float64}, Type)
true

julia> isa(Float64, Type)
true

julia> isa(Real, Type{Float64})
false

julia> isa(Real, Type{Real})
true
```

source

`Core.DataType` – Type.

```
DataType <: Type{T}
```

`DataType` represents explicitly declared types that have names, explicitly declared supertypes, and, optionally, parameters. Every concrete value in the system is an instance of some `DataType`.

Examples

```
julia> typeof(Real)
DataType

julia> typeof(Int)
DataType

julia> struct Point
           x::Int
           y
       end

julia> typeof(Point)
DataType
```

source

`Core.:<:` – Function.

```
<:(T1, T2)
```

Subtype operator: returns `true` if and only if all values of type T1 are also of type T2.

Examples

```
julia> Float64 <: AbstractFloat
true

julia> Vector{Int} <: AbstractArray
true

julia> Matrix{Float64} <: Matrix{AbstractFloat}
false
```

source

`Base.:>:` – Function.

```
>:(T1, T2)
```

Supertype operator, equivalent to `T2 <: T1`.

source

`Base.typejoin` – Function.

```
typejoin(T, S, ...)
```

Return the closest common ancestor of types T and S, i.e. the narrowest type from which they both inherit. Recurses on additional varargs.

Examples

```
julia> typejoin(Int, Float64)
Real

julia> typejoin(Int, Float64, ComplexF32)
Number
```

source

`Base.typeintersect` – Function.

```
typeintersect(T::Type, S::Type)
```

Compute a type that contains the intersection of T and S. Usually this will be the smallest such type or one close to it.

source

`Base.promote_type` – Function.

```
promote_type(type1, type2, ...)
```

Promotion refers to converting values of mixed types to a single common type. `promote_type` represents the default promotion behavior in Julia when operators (usually mathematical) are given arguments of differing types. `promote_type` generally tries to return a type which can at least approximate most values of either input type without excessively widening. Some loss is tolerated; for example, `promote_type(Int64, Float64)` returns `Float64` even though strictly, not all `Int64` values can be represented exactly as `Float64` values.

See also: `promote`, `promote_typejoin`, `promote_rule`.

Examples

```
julia> promote_type(Int64, Float64)
Float64

julia> promote_type(Int32, Int64)
Int64

julia> promote_type(Float32, BigInt)
BigFloat

julia> promote_type(Int16, Float16)
Float16

julia> promote_type(Int64, Float16)
Float16

julia> promote_type(Int8, UInt16)
UInt16
```

Don't overload this directly

To overload promotion for your own types you should overload `promote_rule`. `promote_type` calls `promote_rule` internally to determine the type. Overloading `promote_type` directly can cause ambiguity errors.

source

`Base.promote_rule` – Function.

```
promote_rule(type1, type2)
```

Specifies what type should be used by `promote` when given values of types `type1` and `type2`. This function should not be called directly, but should have definitions added to it for new types as appropriate.

source

`Base.promote_typejoin` – Function.

```
promote_typejoin(T, S)
```

Compute a type that contains both T and S, which could be either a parent of both types, or a `Union` if appropriate. Falls back to `typejoin`.

See instead `promote`, `promote_type`.

Examples

```
julia> Base.promote_typejoin(Int, Float64)
Real

julia> Base.promote_type(Int, Float64)
Float64
```

source

`Base.isdispatchtuple` – Function.

```
isdispatchtuple(T)
```

Determine whether type T is a tuple "leaf type", meaning it could appear as a type signature in dispatch and has no subtypes (or supertypes) which could appear in a call.

source

Declared structure

`Base.ismutable` – Function.

```
ismutable(v) -> Bool
```

Return `true` if and only if value v is mutable. See Mutable Composite Types for a discussion of immutability. Note that this function works on values, so if you give it a `DataType`, it will tell you that a value of the type is mutable.

> **Note**
>
> For technical reasons, `ismutable` returns `true` for values of certain special types (for example `String` and `Symbol`) even though they cannot be mutated in a permissible way.

See also `isbits`, `isstructtype`.

Examples

```
julia> ismutable(1)
false

julia> ismutable([1,2])
true
```

Julia 1.5

This function requires at least Julia 1.5.

source

Base.isimmutable – Function.

```
isimmutable(v) -> Bool
```

Warning

Consider using !ismutable(v) instead, as isimmutable(v) will be replaced by !ismutable(v) in a future release. (Since Julia 1.5)

Return true iff value v is immutable. See Mutable Composite Types for a discussion of immutability. Note that this function works on values, so if you give it a type, it will tell you that a value of DataType is mutable.

Examples

```
julia> isimmutable(1)
true

julia> isimmutable([1,2])
false
```

source

Base.ismutabletype – Function.

```
ismutabletype(T) -> Bool
```

Determine whether type T was declared as a mutable type (i.e. using mutable struct keyword).

Julia 1.7

This function requires at least Julia 1.7.

source

Base.isabstracttype – Function.

```
isabstracttype(T)
```

Determine whether type T was declared as an abstract type (i.e. using the abstract type syntax).

Examples

```
julia> isabstracttype(AbstractArray)
true

julia> isabstracttype(Vector)
false
```

source

Base.isprimitivetype – Function.

```
isprimitivetype(T) -> Bool
```

Determine whether type T was declared as a primitive type (i.e. using the `primitive type` syntax).

source

Base.issingletontype – Function.

```
Base.issingletontype(T)
```

Determine whether type T has exactly one possible instance; for example, a struct type with no fields.

source

Base.isstructtype – Function.

```
isstructtype(T) -> Bool
```

Determine whether type T was declared as a struct type (i.e. using the `struct` or `mutable struct` keyword).

source

Base.nameof – Method.

```
nameof(t::DataType) -> Symbol
```

Get the name of a (potentially `UnionAll`-wrapped) `DataType` (without its parent module) as a symbol.

Examples

```
julia> module Foo
           struct S{T}
           end
       end
Foo

julia> nameof(Foo.S{T} where T)
:S
```

source

Base.fieldnames – Function.

```
fieldnames(x::DataType)
```

Get a tuple with the names of the fields of a DataType.

See also propertynames, hasfield.

Examples

```
julia> fieldnames(Rational)
(:num, :den)

julia> fieldnames(typeof(1+im))
(:re, :im)
```

source

Base.fieldname – Function.

```
fieldname(x::DataType, i::Integer)
```

Get the name of field i of a DataType.

Examples

```
julia> fieldname(Rational, 1)
:num

julia> fieldname(Rational, 2)
:den
```

source

Core.fieldtype – Function.

```
fieldtype(T, name::Symbol | index::Int)
```

Determine the declared type of a field (specified by name or index) in a composite DataType T.

Examples

```
julia> struct Foo
           x::Int64
           y::String
       end
```

```
julia> fieldtype(Foo, :x)
Int64

julia> fieldtype(Foo, 2)
String
```

source

Base.fieldtypes – Function.

```
fieldtypes(T::Type)
```

The declared types of all fields in a composite DataType T as a tuple.

Julia 1.1

This function requires at least Julia 1.1.

Examples

```
julia> struct Foo
           x::Int64
           y::String
       end

julia> fieldtypes(Foo)
(Int64, String)
```

source

Base.fieldcount – Function.

```
fieldcount(t::Type)
```

Get the number of fields that an instance of the given type would have. An error is thrown if the type is too abstract to determine this.

source

Base.hasfield – Function.

```
hasfield(T::Type, name::Symbol)
```

Return a boolean indicating whether T has name as one of its own fields.

See also fieldnames, fieldcount, hasproperty.

Julia 1.2

This function requires at least Julia 1.2.

Examples

```
julia> struct Foo
           bar::Int
       end

julia> hasfield(Foo, :bar)
true

julia> hasfield(Foo, :x)
false
```

source

Core.nfields – Function.

```
nfields(x) -> Int
```

Get the number of fields in the given object.

Examples

```
julia> a = 1//2;

julia> nfields(a)
2

julia> b = 1
1

julia> nfields(b)
0

julia> ex = ErrorException("I've done a bad thing");

julia> nfields(ex)
1
```

In these examples, a is a Rational, which has two fields. b is an Int, which is a primitive bitstype with no fields at all. ex is an ErrorException, which has one field.

source

Base.isconst – Function.

```
isconst(m::Module, s::Symbol) -> Bool
```

Determine whether a global is declared `const` in a given module `m`.

source

```
isconst(t::DataType, s::Union{Int,Symbol}) -> Bool
```

Determine whether a field `s` is declared `const` in a given type `t`.

source

`Base.isfieldatomic` – Function.

```
isfieldatomic(t::DataType, s::Union{Int,Symbol}) -> Bool
```

Determine whether a field `s` is declared `@atomic` in a given type `t`.

source

Memory layout

`Base.sizeof` – Method.

```
sizeof(T::DataType)
sizeof(obj)
```

Size, in bytes, of the canonical binary representation of the given `DataType` `T`, if any. Or the size, in bytes, of object `obj` if it is not a `DataType`.

See also `Base.summarysize`.

Examples

```
julia> sizeof(Float32)
4

julia> sizeof(ComplexF64)
16

julia> sizeof(1.0)
8

julia> sizeof(collect(1.0:10.0))
80

julia> struct StructWithPadding
           x::Int64
           flag::Bool
       end

julia> sizeof(StructWithPadding) # not the sum of `sizeof` of fields due to padding
16

julia> sizeof(Int64) + sizeof(Bool) # different from above
9
```

If `DataType` T does not have a specific size, an error is thrown.

```
julia> sizeof(AbstractArray)
ERROR: Abstract type AbstractArray does not have a definite size.
Stacktrace:
[...]
```

source

`Base.isconcretetype` – Function.

```
isconcretetype(T)
```

Determine whether type T is a concrete type, meaning it could have direct instances (values x such that `typeof(x) === T`).

See also: `isbits`, `isabstracttype`, `issingletontype`.

Examples

```
julia> isconcretetype(Complex)
false

julia> isconcretetype(Complex{Float32})
true

julia> isconcretetype(Vector{Complex})
true

julia> isconcretetype(Vector{Complex{Float32}})
true

julia> isconcretetype(Union{})
false

julia> isconcretetype(Union{Int,String})
false
```

source

`Base.isbits` – Function.

```
isbits(x)
```

Return `true` if x is an instance of an `isbitstype` type.

source

`Base.isbitstype` – Function.

```
isbitstype(T)
```

Return `true` if type T is a "plain data" type, meaning it is immutable and contains no references to other values, only `primitive` types and other `isbitstype` types. Typical examples are numeric types such as `UInt8`, `Float64`, and `Complex{Float64}`. This category of types is significant since they are valid as type parameters, may not track `isdefined` / `isassigned` status, and have a defined layout that is compatible with C.

See also `isbits`, `isprimitivetype`, `ismutable`.

Examples

```
julia> isbitstype(Complex{Float64})
true

julia> isbitstype(Complex)
false
```

source

`Base.fieldoffset` – Function.

```
fieldoffset(type, i)
```

The byte offset of field `i` of a type relative to the data start. For example, we could use it in the following manner to summarize information about a struct:

```
julia> structinfo(T) = [(fieldoffset(T,i), fieldname(T,i), fieldtype(T,i)) for i =
↪  1:fieldcount(T)];

julia> structinfo(Base.Filesystem.StatStruct)
13-element Vector{Tuple{UInt64, Symbol, Type}}:
 (0x0000000000000000, :desc, Union{RawFD, String})
 (0x0000000000000008, :device, UInt64)
 (0x0000000000000010, :inode, UInt64)
 (0x0000000000000018, :mode, UInt64)
 (0x0000000000000020, :nlink, Int64)
 (0x0000000000000028, :uid, UInt64)
 (0x0000000000000030, :gid, UInt64)
 (0x0000000000000038, :rdev, UInt64)
 (0x0000000000000040, :size, Int64)
 (0x0000000000000048, :blksize, Int64)
 (0x0000000000000050, :blocks, Int64)
 (0x0000000000000058, :mtime, Float64)
 (0x0000000000000060, :ctime, Float64)
```

source

`Base.datatype_alignment` – Function.

```
Base.datatype_alignment(dt::DataType) -> Int
```

Memory allocation minimum alignment for instances of this type. Can be called on any `isconcretetype`.

source

`Base.datatype_haspadding` – Function.

```
Base.datatype_haspadding(dt::DataType) -> Bool
```

Return whether the fields of instances of this type are packed in memory, with no intervening padding bytes. Can be called on any `isconcretetype`.

source

`Base.datatype_pointerfree` – Function.

```
Base.datatype_pointerfree(dt::DataType) -> Bool
```

Return whether instances of this type can contain references to gc-managed memory. Can be called on any `isconcretetype`.

source

Special values

`Base.typemin` – Function.

```
typemin(T)
```

The lowest value representable by the given (real) numeric DataType T.

See also: `floatmin`, `typemax`, `eps`.

Examples

```
julia> typemin(Int8)
-128

julia> typemin(UInt32)
0x00000000

julia> typemin(Float16)
-Inf16

julia> typemin(Float32)
-Inf32

julia> nextfloat(-Inf32)  # smallest finite Float32 floating point number
-3.4028235f38
```

source

Base.typemax – Function.

```
typemax(T)
```

The highest value representable by the given (real) numeric DataType.

See also: floatmax, typemin, eps.

Examples

```
julia> typemax(Int8)
127

julia> typemax(UInt32)
0xffffffff

julia> typemax(Float64)
Inf

julia> typemax(Float32)
Inf32

julia> floatmax(Float32)  # largest finite Float32 floating point number
3.4028235f38
```

source

Base.floatmin – Function.

```
floatmin(T = Float64)
```

Return the smallest positive normal number representable by the floating-point type T.

Examples

```
julia> floatmin(Float16)
Float16(6.104e-5)

julia> floatmin(Float32)
1.1754944f-38

julia> floatmin()
2.2250738585072014e-308
```

source

Base.floatmax – Function.

```
floatmax(T = Float64)
```

Return the largest finite number representable by the floating-point type T.

See also: `typemax`, `floatmin`, `eps`.

Examples

```
julia> floatmax(Float16)
Float16(6.55e4)

julia> floatmax(Float32)
3.4028235f38

julia> floatmax()
1.7976931348623157e308

julia> typemax(Float64)
Inf
```

source

`Base.maxintfloat` – Function.

```
maxintfloat(T=Float64)
```

The largest consecutive integer-valued floating-point number that is exactly represented in the given floating-point type T (which defaults to `Float64`).

That is, `maxintfloat` returns the smallest positive integer-valued floating-point number n such that n+1 is *not* exactly representable in the type T.

When an `Integer`-type value is needed, use `Integer(maxintfloat(T))`.

source

```
maxintfloat(T, S)
```

The largest consecutive integer representable in the given floating-point type T that also does not exceed the maximum integer representable by the integer type S. Equivalently, it is the minimum of `maxintfloat(T)` and `typemax(S)`.

source

`Base.eps` – Method.

```
eps(::Type{T}) where T<:AbstractFloat
eps()
```

Return the *machine epsilon* of the floating point type T (T = Float64 by default). This is defined as the gap between 1 and the next largest value representable by typeof(one(T)), and is equivalent to eps(one(T)). (Since eps(T) is a bound on the *relative error* of T, it is a "dimensionless" quantity like one.)

Examples

```
julia> eps()
2.220446049250313e-16

julia> eps(Float32)
1.1920929f-7

julia> 1.0 + eps()
1.0000000000000002

julia> 1.0 + eps()/2
1.0
```

source

Base.eps – Method.

```
eps(x::AbstractFloat)
```

Return the *unit in last place* (ulp) of x. This is the distance between consecutive representable floating point values at x. In most cases, if the distance on either side of x is different, then the larger of the two is taken, that is

```
eps(x) == max(x-prevfloat(x), nextfloat(x)-x)
```

The exceptions to this rule are the smallest and largest finite values (e.g. nextfloat(-Inf) and prevfloat(Inf) for Float64), which round to the smaller of the values.

The rationale for this behavior is that eps bounds the floating point rounding error. Under the default RoundNearest rounding mode, if y is a real number and x is the nearest floating point number to y, then

$$|y - x| \leq \text{eps}(x)/2.$$

See also: nextfloat, issubnormal, floatmax.

Examples

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(prevfloat(2.0))
2.220446049250313e-16

julia> eps(2.0)
4.440892098500626e-16
```

```
julia> x = prevfloat(Inf)      # largest finite Float64
1.7976931348623157e308

julia> x + eps(x)/2            # rounds up
Inf

julia> x + prevfloat(eps(x)/2) # rounds down
1.7976931348623157e308
```

source

Base.instances – Function.

```
instances(T::Type)
```

Return a collection of all instances of the given type, if applicable. Mostly used for enumerated types (see @enum).

Example

```
julia> @enum Color red blue green

julia> instances(Color)
(red, blue, green)
```

source

42.8 Special Types

Core.Any – Type.

```
Any::DataType
```

Any is the union of all types. It has the defining property isa(x, Any) == true for any x. Any therefore describes the entire universe of possible values. For example Integer is a subset of Any that includes Int, Int8, and other integer types.

source

Core.Union – Type.

```
Union{Types...}
```

A type union is an abstract type which includes all instances of any of its argument types. The empty union Union{} is the bottom type of Julia.

Examples

```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}

julia> 1 isa IntOrString
true

julia> "Hello!" isa IntOrString
true

julia> 1.0 isa IntOrString
false
```

source

Union{} – Keyword.

```
Union{}
```

Union{}, the empty Union of types, is the type that has no values. That is, it has the defining property isa(x, Union{}) == false for any x. Base.Bottom is defined as its alias and the type of Union{} is Core.TypeofBottom.

Examples

```
julia> isa(nothing, Union{})
false
```

source

Core.UnionAll – Type.

```
UnionAll
```

A union of types over all values of a type parameter. UnionAll is used to describe parametric types where the values of some parameters are not known.

Examples

```
julia> typeof(Vector)
UnionAll

julia> typeof(Vector{Int})
DataType
```

source

Core.Tuple – Type.

```
Tuple{Types...}
```

A tuple is a fixed-length container that can hold any values of different types, but cannot be modified (it is immutable). The values can be accessed via indexing. Tuple literals are written with commas and parentheses:

```
julia> (1, 1+1)
(1, 2)

julia> (1,)
(1,)

julia> x = (0.0, "hello", 6*7)
(0.0, "hello", 42)

julia> x[2]
"hello"

julia> typeof(x)
Tuple{Float64, String, Int64}
```

A length-1 tuple must be written with a comma, `(1,)`, since `(1)` would just be a parenthesized value. `()` represents the empty (length-0) tuple.

A tuple can be constructed from an iterator by using a `Tuple` type as constructor:

```
julia> Tuple(["a", 1])
("a", 1)

julia> Tuple{String, Float64}(["a", 1])
("a", 1.0)
```

Tuple types are covariant in their parameters: `Tuple{Int}` is a subtype of `Tuple{Any}`. Therefore `Tuple{Any}` is considered an abstract type, and tuple types are only concrete if their parameters are. Tuples do not have field names; fields are only accessed by index. Tuple types may have any number of parameters.

See the manual section on Tuple Types.

See also `Vararg`, `NTuple`, `ntuple`, `tuple`, `NamedTuple`.

source

`Core.NTuple` – Type.

```
NTuple{N, T}
```

A compact way of representing the type for a tuple of length `N` where all elements are of type `T`.

Examples

```
julia> isa((1, 2, 3, 4, 5, 6), NTuple{6, Int})
true
```

See also `ntuple`.

source

`Core.NamedTuple` – Type.

```
NamedTuple
```

`NamedTuple`s are, as their name suggests, named `Tuple`s. That is, they're a tuple-like collection of values, where each entry has a unique name, represented as a `Symbol`. Like `Tuple`s, `NamedTuple`s are immutable; neither the names nor the values can be modified in place after construction.

A named tuple can be created as a tuple literal with keys, e.g. `(a=1, b=2)`, or as a tuple literal with semicolon after the opening parenthesis, e.g. `(; a=1, b=2)` (this form also accepts programmatically generated names as described below), or using a `NamedTuple` type as constructor, e.g. `NamedTuple{(:a, :b)}((1,2))`.

Accessing the value associated with a name in a named tuple can be done using field access syntax, e.g. `x.a`, or using `getindex`, e.g. `x[:a]` or `x[(:a, :b)]`. A tuple of the names can be obtained using `keys`, and a tuple of the values can be obtained using `values`.

Note

Iteration over `NamedTuple`s produces the *values* without the names. (See example below.) To iterate over the name-value pairs, use the `pairs` function.

The `@NamedTuple` macro can be used for conveniently declaring `NamedTuple` types.

Examples

```
julia> x = (a=1, b=2)
(a = 1, b = 2)

julia> x.a
1

julia> x[:a]
1

julia> x[(:a,)]
(a = 1,)

julia> keys(x)
(:a, :b)

julia> values(x)
(1, 2)

julia> collect(x)
2-element Vector{Int64}:
```

```
 1
 2

julia> collect(pairs(x))
2-element Vector{Pair{Symbol, Int64}}:
 :a => 1
 :b => 2
```

In a similar fashion as to how one can define keyword arguments programmatically, a named tuple can be created by giving pairs `name::Symbol => value` after a semicolon inside a tuple literal. This and the `name=value` syntax can be mixed:

```
julia> (; :a => 1, :b => 2, c=3)
(a = 1, b = 2, c = 3)
```

The name-value pairs can also be provided by splatting a named tuple or any iterator that yields two-value collections holding each a symbol as first value:

```
julia> keys = (:a, :b, :c); values = (1, 2, 3);

julia> NamedTuple{keys}(values)
(a = 1, b = 2, c = 3)

julia> (; (keys .=> values)...)
(a = 1, b = 2, c = 3)

julia> nt1 = (a=1, b=2);

julia> nt2 = (c=3, d=4);

julia> (; nt1..., nt2..., b=20) # the final b overwrites the value from nt1
(a = 1, b = 20, c = 3, d = 4)

julia> (; zip(keys, values)...) # zip yields tuples such as (:a, 1)
(a = 1, b = 2, c = 3)
```

As in keyword arguments, identifiers and dot expressions imply names:

```
julia> x = 0
0

julia> t = (; x)
(x = 0,)

julia> (; t.x)
(x = 0,)
```

Julia 1.5

Implicit names from identifiers and dot expressions are available as of Julia 1.5.

Julia 1.7

Use of getindex methods with multiple Symbols is available as of Julia 1.7.

source

Base.@NamedTuple – Macro.

```
@NamedTuple{key1::Type1, key2::Type2, ...}
@NamedTuple begin key1::Type1; key2::Type2; ...; end
```

This macro gives a more convenient syntax for declaring NamedTuple types. It returns a NamedTuple type with the given keys and types, equivalent to NamedTuple{(:key1, :key2, ...), Tuple{Type1,Type2,...}}. If the ::Type declaration is omitted, it is taken to be Any. The begin ... end form allows the declarations to be split across multiple lines (similar to a struct declaration), but is otherwise equivalent. The NamedTuple macro is used when printing NamedTuple types to e.g. the REPL.

For example, the tuple (a=3.1, b="hello") has a type NamedTuple{(:a, :b), Tuple{Float64, String}}, which can also be declared via @NamedTuple as:

```
julia> @NamedTuple{a::Float64, b::String}
@NamedTuple{a::Float64, b::String}

julia> @NamedTuple begin
           a::Float64
           b::String
       end
@NamedTuple{a::Float64, b::String}
```

Julia 1.5

This macro is available as of Julia 1.5.

source

Base.@Kwargs – Macro.

```
@Kwargs{key1::Type1, key2::Type2, ...}
```

This macro gives a convenient way to construct the type representation of keyword arguments from the same syntax as @NamedTuple. For example, when we have a function call like func([positional arguments]; kw1=1.0, kw2="2"), we can use this macro to construct the internal type representation of the keyword arguments as @Kwargs{kw1::Float64, kw2::String}. The macro syntax is specifically designed to simplify the signature type of a keyword method when it is printed in the stack trace view.

```
julia> @Kwargs{init::Int} # the internal representation of keyword arguments
Base.Pairs{Symbol, Int64, Tuple{Symbol}, @NamedTuple{init::Int64}}
```

```
julia> sum("julia"; init=1)
ERROR: MethodError: no method matching +(::Char, ::Char)

Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...)
   @ Base operators.jl:585
  +(::Integer, ::AbstractChar)
   @ Base char.jl:247
  +(::T, ::Integer) where T<:AbstractChar
   @ Base char.jl:237

Stacktrace:
  [1] add_sum(x::Char, y::Char)
    @ Base ./reduce.jl:24
  [2] BottomRF
    @ Base ./reduce.jl:86 [inlined]
  [3] _foldl_impl(op::Base.BottomRF{typeof(Base.add_sum)}, init::Int64, itr::String)
    @ Base ./reduce.jl:62
  [4] foldl_impl(op::Base.BottomRF{typeof(Base.add_sum)}, nt::Int64, itr::String)
    @ Base ./reduce.jl:48 [inlined]
  [5] mapfoldl_impl(f::typeof(identity), op::typeof(Base.add_sum), nt::Int64, itr::String)
    @ Base ./reduce.jl:44 [inlined]
  [6] mapfoldl(f::typeof(identity), op::typeof(Base.add_sum), itr::String; init::Int64)
    @ Base ./reduce.jl:175 [inlined]
  [7] mapreduce(f::typeof(identity), op::typeof(Base.add_sum), itr::String;
  ↪   kw::@Kwargs{init::Int64})
    @ Base ./reduce.jl:307 [inlined]
  [8] sum(f::typeof(identity), a::String; kw::@Kwargs{init::Int64})
    @ Base ./reduce.jl:535 [inlined]
  [9] sum(a::String; kw::@Kwargs{init::Int64})
    @ Base ./reduce.jl:564 [inlined]
 [10] top-level scope
    @ REPL[12]:1
```

Julia 1.10

This macro is available as of Julia 1.10.

source

`Base.Val` – Type.

```
Val(c)
```

Return `Val{c}()`, which contains no run-time data. Types like this can be used to pass the information between functions through the value `c`, which must be an `isbits` value or a `Symbol`. The intent of this construct is to be able to dispatch on constants directly (at compile time) without having to test the value of the constant at run time.

Examples

```
julia> f(::Val{true}) = "Good"
f (generic function with 1 method)

julia> f(::Val{false}) = "Bad"
f (generic function with 2 methods)

julia> f(Val(true))
"Good"
```

source

Core.Vararg – Constant.

```
Vararg{T,N}
```

The last parameter of a tuple type Tuple can be the special value Vararg, which denotes any number of trailing elements. Vararg{T,N} corresponds to exactly N elements of type T. Finally Vararg{T} corresponds to zero or more elements of type T. Vararg tuple types are used to represent the arguments accepted by varargs methods (see the section on Varargs Functions in the manual.)

See also NTuple.

Examples

```
julia> mytupletype = Tuple{AbstractString, Vararg{Int}}
Tuple{AbstractString, Vararg{Int64}}

julia> isa(("1",), mytupletype)
true

julia> isa(("1",1), mytupletype)
true

julia> isa(("1",1,2), mytupletype)
true

julia> isa(("1",1,2,3.0), mytupletype)
false
```

source

Core.Nothing – Type.

```
Nothing
```

A type with no fields that is the type of nothing.

See also: isnothing, Some, Missing.

source

Base.isnothing – Function.

```
isnothing(x)
```

Return true if x === nothing, and return false if not.

Julia 1.1

This function requires at least Julia 1.1.

See also something, Base.notnothing, ismissing.

source

Base.notnothing – Function.

```
notnothing(x)
```

Throw an error if x === nothing, and return x if not.

source

Base.Some – Type.

```
Some{T}
```

A wrapper type used in Union{Some{T}, Nothing} to distinguish between the absence of a value (nothing) and the presence of a nothing value (i.e. Some(nothing)).

Use something to access the value wrapped by a Some object.

source

Base.something – Function.

```
something(x...)
```

Return the first value in the arguments which is not equal to nothing, if any. Otherwise throw an error. Arguments of type Some are unwrapped.

See also coalesce, skipmissing, @something.

Examples

```
julia> something(nothing, 1)
1

julia> something(Some(1), nothing)
1
```

```
julia> something(Some(nothing), 2) === nothing
true

julia> something(missing, nothing)
missing

julia> something(nothing, nothing)
ERROR: ArgumentError: No value arguments present
```

source

Base.@something – Macro.

```
@something(x...)
```

Short-circuiting version of something.

Examples

```
julia> f(x) = (println("f($x)"); nothing);

julia> a = 1;

julia> a = @something a f(2) f(3) error("Unable to find default for `a`")
1

julia> b = nothing;

julia> b = @something b f(2) f(3) error("Unable to find default for `b`")
f(2)
f(3)
ERROR: Unable to find default for `b`
[...]

julia> b = @something b f(2) f(3) Some(nothing)
f(2)
f(3)

julia> b === nothing
true
```

Julia 1.7

This macro is available as of Julia 1.7.

source

Base.Enums.Enum – Type.

```
Enum{T<:Integer}
```

The abstract supertype of all enumerated types defined with `@enum`.

source

`Base.Enums.@enum` – Macro.

```
@enum EnumName[::BaseType] value1[=x] value2[=y]
```

Create an `Enum{BaseType}` subtype with name `EnumName` and enum member values of `value1` and `value2` with optional assigned values of `x` and `y`, respectively. `EnumName` can be used just like other types and enum member values as regular values, such as

Examples

```
julia> @enum Fruit apple=1 orange=2 kiwi=3

julia> f(x::Fruit) = "I'm a Fruit with value: $(Int(x))"
f (generic function with 1 method)

julia> f(apple)
"I'm a Fruit with value: 1"

julia> Fruit(1)
apple::Fruit = 1
```

Values can also be specified inside a `begin` block, e.g.

```
@enum EnumName begin
    value1
    value2
end
```

`BaseType`, which defaults to `Int32`, must be a primitive subtype of `Integer`. Member values can be converted between the enum type and `BaseType`. `read` and `write` perform these conversions automatically. In case the enum is created with a non-default `BaseType`, `Integer(value1)` will return the integer `value1` with the type `BaseType`.

To list all the instances of an enum use `instances`, e.g.

```
julia> instances(Fruit)
(apple, orange, kiwi)
```

It is possible to construct a symbol from an enum instance:

```
julia> Symbol(apple)
:apple
```

source

Core.Expr – Type.

```
Expr(head::Symbol, args...)
```

A type representing compound expressions in parsed julia code (ASTs). Each expression consists of a head Symbol identifying which kind of expression it is (e.g. a call, for loop, conditional statement, etc.), and subexpressions (e.g. the arguments of a call). The subexpressions are stored in a Vector{Any} field called args.

See the manual chapter on Metaprogramming and the developer documentation Julia ASTs.

Examples

```
julia> Expr(:call, :+, 1, 2)
:(1 + 2)

julia> dump(:(a ? b : c))
Expr
  head: Symbol if
  args: Array{Any}((3,))
    1: Symbol a
    2: Symbol b
    3: Symbol c
```

source

Core.Symbol – Type.

```
Symbol
```

The type of object used to represent identifiers in parsed julia code (ASTs). Also often used as a name or label to identify an entity (e.g. as a dictionary key). Symbols can be entered using the : quote operator:

```
julia> :name
:name

julia> typeof(:name)
Symbol

julia> x = 42
42

julia> eval(:x)
42
```

Symbols can also be constructed from strings or other values by calling the constructor Symbol(x...).

Symbols are immutable and their implementation re-uses the same object for all Symbols with the same name.

Unlike strings, Symbols are "atomic" or "scalar" entities that do not support iteration over characters.

source

Core.Symbol – Method.

```
Symbol(x...) -> Symbol
```

Create a Symbol by concatenating the string representations of the arguments together.

Examples

```
julia> Symbol("my", "name")
:myname

julia> Symbol("day", 4)
:day4
```

source

Core.Module – Type.

```
Module
```

A Module is a separate global variable workspace. See module and the manual section about modules for details.

```
Module(name::Symbol=:anonymous, std_imports=true, default_names=true)
```

Return a module with the specified name. A baremodule corresponds to Module(:ModuleName, false)

An empty module containing no names at all can be created with Module(:ModuleName, false, false). This module will not import Base or Core and does not contain a reference to itself.

source

42.9 Generic Functions

Core.Function – Type.

```
Function
```

Abstract type of all functions.

Examples

```
julia> isa(+, Function)
true

julia> typeof(sin)
typeof(sin) (singleton type of function sin, subtype of Function)

julia> ans <: Function
true
```

source

Base.hasmethod – Function.

```
hasmethod(f, t::Type{<:Tuple}[, kwnames]; world=get_world_counter()) -> Bool
```

Determine whether the given generic function has a method matching the given Tuple of argument types with the upper bound of world age given by world.

If a tuple of keyword argument names kwnames is provided, this also checks whether the method of f matching t has the given keyword argument names. If the matching method accepts a variable number of keyword arguments, e.g. with kwargs..., any names given in kwnames are considered valid. Otherwise the provided names must be a subset of the method's keyword arguments.

See also applicable.

Julia 1.2

Providing keyword argument names requires Julia 1.2 or later.

Examples

```
julia> hasmethod(length, Tuple{Array})
true

julia> f(; oranges=0) = oranges;

julia> hasmethod(f, Tuple{}, (:oranges,))
true

julia> hasmethod(f, Tuple{}, (:apples, :bananas))
false

julia> g(; xs...) = 4;

julia> hasmethod(g, Tuple{}, (:a, :b, :c, :d))  # g accepts arbitrary kwargs
true
```

source

Core.applicable – Function.

```
applicable(f, args...) -> Bool
```

Determine whether the given generic function has a method applicable to the given arguments.

See also hasmethod.

Examples

```
julia> function f(x, y)
           x + y
       end;

julia> applicable(f, 1)
false

julia> applicable(f, 1, 2)
true
```

source

Base.isambiguous – Function.

```
Base.isambiguous(m1, m2; ambiguous_bottom=false) -> Bool
```

Determine whether two methods m1 and m2 may be ambiguous for some call signature. This test is performed in the context of other methods of the same function; in isolation, m1 and m2 might be ambiguous, but if a third method resolving the ambiguity has been defined, this returns false. Alternatively, in isolation m1 and m2 might be ordered, but if a third method cannot be sorted with them, they may cause an ambiguity together.

For parametric types, the ambiguous_bottom keyword argument controls whether Union{} counts as an ambiguous intersection of type parameters –when true, it is considered ambiguous, when false it is not.

Examples

```
julia> foo(x::Complex{<:Integer}) = 1
foo (generic function with 1 method)

julia> foo(x::Complex{<:Rational}) = 2
foo (generic function with 2 methods)

julia> m1, m2 = collect(methods(foo));

julia> typeintersect(m1.sig, m2.sig)
Tuple{typeof(foo), Complex{Union{}}}

julia> Base.isambiguous(m1, m2, ambiguous_bottom=true)
true

julia> Base.isambiguous(m1, m2, ambiguous_bottom=false)
false
```

source

Core.invoke – Function.

```
invoke(f, argtypes::Type, args...; kwargs...)
```

Invoke a method for the given generic function `f` matching the specified types `argtypes` on the specified arguments `args` and passing the keyword arguments `kwargs`. The arguments `args` must conform with the specified types in `argtypes`, i.e. conversion is not automatically performed. This method allows invoking a method other than the most specific matching method, which is useful when the behavior of a more general definition is explicitly needed (often as part of the implementation of a more specific method of the same function).

Be careful when using `invoke` for functions that you don't write. What definition is used for given `argtypes` is an implementation detail unless the function is explicitly states that calling with certain `argtypes` is a part of public API. For example, the change between `f1` and `f2` in the example below is usually considered compatible because the change is invisible by the caller with a normal (non-`invoke`) call. However, the change is visible if you use `invoke`.

Examples

```
julia> f(x::Real) = x^2;

julia> f(x::Integer) = 1 + invoke(f, Tuple{Real}, x);

julia> f(2)
5

julia> f1(::Integer) = Integer
       f1(::Real) = Real;

julia> f2(x::Real) = _f2(x)
       _f2(::Integer) = Integer
       _f2(_) = Real;

julia> f1(1)
Integer

julia> f2(1)
Integer

julia> invoke(f1, Tuple{Real}, 1)
Real

julia> invoke(f2, Tuple{Real}, 1)
Integer
```

source

Base.@invoke – Macro.

```
@invoke f(arg::T, ...; kwargs...)
```

Provides a convenient way to call invoke by expanding `@invoke f(arg1::T1, arg2::T2; kwargs...)` to `invoke(f, Tuple{T1,T2}, arg1, arg2; kwargs...)`. When an argument's type annotation is omitted, it's replaced with `Core.Typeof` that argument. To invoke a method where an argument is untyped or explicitly typed as `Any`, annotate the argument with `::Any`.

It also supports the following syntax:

- `@invoke (x::X).f` expands to `invoke(getproperty, Tuple{X,Symbol}, x, :f)`
- `@invoke (x::X).f = v::V` expands to `invoke(setproperty!, Tuple{X,Symbol,V}, x, :f, v)`
- `@invoke (xs::Xs)[i::I]` expands to `invoke(getindex, Tuple{Xs,I}, xs, i)`
- `@invoke (xs::Xs)[i::I] = v::V` expands to `invoke(setindex!, Tuple{Xs,V,I}, xs, v, i)`

Examples

```
julia> @macroexpand @invoke f(x::T, y)
:(Core.invoke(f, Tuple{T, Core.Typeof(y)}, x, y))

julia> @invoke 420::Integer % Unsigned
0x00000000000001a4

julia> @macroexpand @invoke (x::X).f
:(Core.invoke(Base.getproperty, Tuple{X, Core.Typeof(:f)}, x, :f))

julia> @macroexpand @invoke (x::X).f = v::V
:(Core.invoke(Base.setproperty!, Tuple{X, Core.Typeof(:f), V}, x, :f, v))

julia> @macroexpand @invoke (xs::Xs)[i::I]
:(Core.invoke(Base.getindex, Tuple{Xs, I}, xs, i))

julia> @macroexpand @invoke (xs::Xs)[i::I] = v::V
:(Core.invoke(Base.setindex!, Tuple{Xs, V, I}, xs, v, i))
```

Julia 1.7

This macro requires Julia 1.7 or later.

Julia 1.9

This macro is exported as of Julia 1.9.

Julia 1.10

The additional syntax is supported as of Julia 1.10.

source

`Base.invokelatest` – Function.

```
invokelatest(f, args...; kwargs...)
```

Calls `f(args...; kwargs...)`, but guarantees that the most recent method of `f` will be executed. This is useful in specialized circumstances, e.g. long-running event loops or callback functions that may call obsolete versions of a function `f`. (The drawback is that `invokelatest` is somewhat slower than calling `f` directly, and the type of the result cannot be inferred by the compiler.)

> **Julia 1.9**
>
> Prior to Julia 1.9, this function was not exported, and was called as `Base.invokelatest`.

source

`Base.@invokelatest` – Macro.

```
@invokelatest f(args...; kwargs...)
```

Provides a convenient way to call invokelatest. `@invokelatest f(args...; kwargs...)` will simply be expanded into `Base.invokelatest(f, args...; kwargs...)`.

It also supports the following syntax:

- `@invokelatest x.f` expands to `Base.invokelatest(getproperty, x, :f)`
- `@invokelatest x.f = v` expands to `Base.invokelatest(setproperty!, x, :f, v)`
- `@invokelatest xs[i]` expands to `Base.invokelatest(getindex, xs, i)`
- `@invokelatest xs[i] = v` expands to `Base.invokelatest(setindex!, xs, v, i)`

```
julia> @macroexpand @invokelatest f(x; kw=kwv)
:(Base.invokelatest(f, x; kw = kwv))

julia> @macroexpand @invokelatest x.f
:(Base.invokelatest(Base.getproperty, x, :f))

julia> @macroexpand @invokelatest x.f = v
:(Base.invokelatest(Base.setproperty!, x, :f, v))

julia> @macroexpand @invokelatest xs[i]
:(Base.invokelatest(Base.getindex, xs, i))

julia> @macroexpand @invokelatest xs[i] = v
:(Base.invokelatest(Base.setindex!, xs, v, i))
```

> **Julia 1.7**
>
> This macro requires Julia 1.7 or later.

> **Julia 1.9**
>
> Prior to Julia 1.9, this macro was not exported, and was called as `Base.@invokelatest`.

> **Julia 1.10**
>
> The additional `x.f` and `xs[i]` syntax requires Julia 1.10.

source

new – Keyword.

```
new, or new{A,B,...}
```

Special function available to inner constructors which creates a new object of the type. The form new{A,B,...} explicitly specifies values of parameters for parametric types. See the manual section on Inner Constructor Methods for more information.

source

`Base.:|>` – Function.

```
|>(x, f)
```

Infix operator which applies function `f` to the argument `x`. This allows `f(g(x))` to be written `x |> g |> f`. When used with anonymous functions, parentheses are typically required around the definition to get the intended chain.

Examples

```
julia> 4 |> inv
0.25

julia> [2, 3, 5] |> sum |> inv
0.1

julia> [0 1; 2 3] .|> (x -> x^2) |> sum
14
```

source

`Base.:∘` – Function.

```
f ∘ g
```

Compose functions: i.e. `(f ∘ g)(args...; kwargs...)` means `f(g(args...; kwargs...))`. The ∘ symbol can be entered in the Julia REPL (and most editors, appropriately configured) by typing `\circ<tab>`.

Function composition also works in prefix form: `∘(f, g)` is the same as `f ∘ g`. The prefix form supports composition of multiple functions: `∘(f, g, h) = f ∘ g ∘ h` and splatting `∘(fs...)` for composing an iterable collection of functions. The last argument to `∘` execute first.

Julia 1.4

Multiple function composition requires at least Julia 1.4.

Julia 1.5

Composition of one function ∘(f) requires at least Julia 1.5.

Julia 1.7

Using keyword arguments requires at least Julia 1.7.

Examples

```
julia> map(uppercase∘first, ["apple", "banana", "carrot"])
3-element Vector{Char}:
 'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
 'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
 'C': ASCII/Unicode U+0043 (category Lu: Letter, uppercase)

julia> (==(6)∘length).(["apple", "banana", "carrot"])
3-element BitVector:
 0
 1
 1

julia> fs = [
           x -> 2x
           x -> x-1
           x -> x/2
           x -> x+1
       ];

julia> ∘(fs...)(3)
2.0
```

See also `ComposedFunction`, `!f::Function`.

source

`Base.ComposedFunction` – Type.

```
ComposedFunction{Outer,Inner} <: Function
```

Represents the composition of two callable objects `outer::Outer` and `inner::Inner`. That is

```
ComposedFunction(outer, inner)(args...; kw...) === outer(inner(args...; kw...))
```

The preferred way to construct an instance of `ComposedFunction` is to use the composition operator `∘`:

```
julia> sin ∘ cos === ComposedFunction(sin, cos)
true

julia> typeof(sin∘cos)
ComposedFunction{typeof(sin), typeof(cos)}
```

The composed pieces are stored in the fields of `ComposedFunction` and can be retrieved as follows:

```
julia> composition = sin ∘ cos
sin ∘ cos

julia> composition.outer === sin
true

julia> composition.inner === cos
true
```

Julia 1.6

ComposedFunction requires at least Julia 1.6. In earlier versions ∘ returns an anonymous function instead.

See also `∘`.

source

`Base.splat` – Function.

```
splat(f)
```

Equivalent to

```
    my_splat(f) = args->f(args...)
```

i.e. given a function returns a new function that takes one argument and splats it into the original function. This is useful as an adaptor to pass a multi-argument function in a context that expects a single argument, but passes a tuple as that single argument.

Example usage:

```
julia> map(splat(+), zip(1:3,4:6))
3-element Vector{Int64}:
 5
 7
```

```
9

julia> my_add = splat(+)
splat(+)

julia> my_add((1,2,3))
6
```

source

Base.Fix1 – Type.

```
Fix1(f, x)
```

A type representing a partially-applied version of the two-argument function f, with the first argument fixed to the value "x". In other words, Fix1(f, x) behaves similarly to y->f(x, y).

See also Fix2.

source

Base.Fix2 – Type.

```
Fix2(f, x)
```

A type representing a partially-applied version of the two-argument function f, with the second argument fixed to the value "x". In other words, Fix2(f, x) behaves similarly to y->f(y, x).

source

42.10 Syntax

Core.eval – Function.

```
Core.eval(m::Module, expr)
```

Evaluate an expression in the given module and return the result.

source

Base.MainInclude.eval – Function.

```
eval(expr)
```

Evaluate an expression in the global scope of the containing module. Every Module (except those defined with baremodule) has its own 1-argument definition of eval, which evaluates expressions in that module.

source

Base.@eval – Macro.

```
@eval [mod,] ex
```

Evaluate an expression with values interpolated into it using eval. If two arguments are provided, the first is the module to evaluate in.

source

Base.evalfile – Function.

```
evalfile(path::AbstractString, args::Vector{String}=String[])
```

Load the file into an anonymous module using include, evaluate all expressions, and return the value of the last expression. The optional args argument can be used to set the input arguments of the script (i.e. the global ARGS variable). Note that definitions (e.g. methods, globals) are evaluated in the anonymous module and do not affect the current module.

Example

```
julia> write("testfile.jl", """
           @show ARGS
           1 + 1
       """);

julia> x = evalfile("testfile.jl", ["ARG1", "ARG2"]);
ARGS = ["ARG1", "ARG2"]

julia> x
2

julia> rm("testfile.jl")
```

source

Base.esc – Function.

```
esc(e)
```

Only valid in the context of an Expr returned from a macro. Prevents the macro hygiene pass from turning embedded variables into gensym variables. See the Macros section of the Metaprogramming chapter of the manual for more details and examples.

source

Base.@inbounds – Macro.

```
@inbounds(blk)
```

Eliminates array bounds checking within expressions.

In the example below the in-range check for referencing element `i` of array `A` is skipped to improve performance.

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i in eachindex(A)
        @inbounds r += A[i]
    end
    return r
end
```

Warning

Using `@inbounds` may return incorrect results/crashes/corruption for out-of-bounds indices. The user is responsible for checking it manually. Only use `@inbounds` when it is certain from the information locally available that all accesses are in bounds. In particular, using `1:length(A)` instead of `eachindex(A)` in a function like the one above is *not* safely inbounds because the first index of `A` may not be 1 for all user defined types that subtype `AbstractArray`.

source

`Base.@boundscheck` – Macro.

```
@boundscheck(blk)
```

Annotates the expression `blk` as a bounds checking block, allowing it to be elided by `@inbounds`.

Note

The function in which `@boundscheck` is written must be inlined into its caller in order for `@inbounds` to have effect.

Examples

```
julia> @inline function g(A, i)
           @boundscheck checkbounds(A, i)
           return "accessing ($A)[$i]"
       end;

julia> f1() = return g(1:2, -1);

julia> f2() = @inbounds return g(1:2, -1);

julia> f1()
ERROR: BoundsError: attempt to access 2-element UnitRange{Int64} at index [-1]
Stacktrace:
 [1] throw_boundserror(::UnitRange{Int64}, ::Tuple{Int64}) at ./abstractarray.jl:455
 [2] checkbounds at ./abstractarray.jl:420 [inlined]
 [3] g at ./none:2 [inlined]
```

```
 [4] f1() at ./none:1
 [5] top-level scope

julia> f2()
"accessing (1:2)[-1]"
```

Warning

The `@boundscheck` annotation allows you, as a library writer, to opt-in to allowing *other code* to remove your bounds checks with `@inbounds`. As noted there, the caller must verify—using information they can access—that their accesses are valid before using `@inbounds`. For indexing into your `AbstractArray` subclasses, for example, this involves checking the indices against its `axes`. Therefore, `@boundscheck` annotations should only be added to a `getindex` or `setindex!` implementation after you are certain its behavior is correct.

source

`Base.@propagate_inbounds` – Macro.

```
@propagate_inbounds
```

Tells the compiler to inline a function while retaining the caller's inbounds context.

source

`Base.@inline` – Macro.

```
@inline
```

Give a hint to the compiler that this function is worth inlining.

Small functions typically do not need the `@inline` annotation, as the compiler does it automatically. By using `@inline` on bigger functions, an extra nudge can be given to the compiler to inline it.

`@inline` can be applied immediately before a function definition or within a function body.

```
# annotate long-form definition
@inline function longdef(x)
    ...
end

# annotate short-form definition
@inline shortdef(x) = ...

# annotate anonymous function that a `do` block creates
f() do
    @inline
    ...
end
```

> **Julia 1.8**
>
> The usage within a function body requires at least Julia 1.8.

```
@inline block
```

Give a hint to the compiler that calls within `block` are worth inlining.

```
# The compiler will try to inline `f`
@inline f(...)

# The compiler will try to inline `f`, `g` and `+`
@inline f(...) + g(...)
```

> **Note**
>
> A callsite annotation always has the precedence over the annotation applied to the definition of the called function:
>
> ```
> @noinline function explicit_noinline(args...)
>     # body
> end
>
> let
>     @inline explicit_noinline(args...) # will be inlined
> end
> ```

> **Note**
>
> When there are nested callsite annotations, the innermost annotation has the precedence:
>
> ```
> @noinline let a0, b0 = ...
>     a = @inline f(a0)  # the compiler will try to inline this call
>     b = f(b0)          # the compiler will NOT try to inline this call
>     return a, b
> end
> ```

> **Warning**
>
> Although a callsite annotation will try to force inlining in regardless of the cost model, there are still chances it can't succeed in it. Especially, recursive calls can not be inlined even if they are annotated as `@inlined`.

Julia 1.8

The callsite annotation requires at least Julia 1.8.

source

Base.@noinline – Macro.

```
@noinline
```

Give a hint to the compiler that it should not inline a function.

Small functions are typically inlined automatically. By using `@noinline` on small functions, auto-inlining can be prevented.

`@noinline` can be applied immediately before a function definition or within a function body.

```
# annotate long-form definition
@noinline function longdef(x)
    ...
end

# annotate short-form definition
@noinline shortdef(x) = ...

# annotate anonymous function that a `do` block creates
f() do
    @noinline
    ...
end
```

Julia 1.8

The usage within a function body requires at least Julia 1.8.

```
@noinline block
```

Give a hint to the compiler that it should not inline the calls within `block`.

```
# The compiler will try to not inline `f`
@noinline f(...)

# The compiler will try to not inline `f`, `g` and `+`
@noinline f(...) + g(...)
```

> **Note**
>
> A callsite annotation always has the precedence over the annotation applied to the definition of the called function:
>
> ```
> @inline function explicit_inline(args...)
>     # body
> end
>
> let
>     @noinline explicit_inline(args...) # will not be inlined
> end
> ```

> **Note**
>
> When there are nested callsite annotations, the innermost annotation has the precedence:
>
> ```
> @inline let a0, b0 = ...
>     a = @noinline f(a0)  # the compiler will NOT try to inline this call
>     b = f(b0)            # the compiler will try to inline this call
>     return a, b
> end
> ```

> **Julia 1.8**
>
> The callsite annotation requires at least Julia 1.8.

> **Note**
>
> If the function is trivial (for example returning a constant) it might get inlined anyway.

source

`Base.@nospecialize` – Macro.

```
@nospecialize
```

Applied to a function argument name, hints to the compiler that the method implementation should not be specialized for different types of that argument, but instead use the declared type for that argument. It can be applied to an argument within a formal argument list, or in the function body. When applied to an argument, the macro must wrap the entire argument expression, e.g., `@nospecialize(x::Real)` or `@nospecialize(i::Integer...)` rather than wrapping just the argument name. When used in a function body, the macro must occur in statement position and before any code.

When used without arguments, it applies to all arguments of the parent scope. In local scope, this means all arguments of the containing function. In global (top-level) scope, this means all methods subsequently defined in the current module.

Specialization can reset back to the default by using @specialize.

```
function example_function(@nospecialize x)
    ...
end

function example_function(x, @nospecialize(y = 1))
    ...
end

function example_function(x, y, z)
    @nospecialize x y
    ...
end

@nospecialize
f(y) = [x for x in y]
@specialize
```

Note

@nospecialize affects code generation but not inference: it limits the diversity of the resulting native code, but it does not impose any limitations (beyond the standard ones) on type-inference. Use Base.@nospecializeinfer together with @nospecialize to additionally suppress inference.

Example

```
julia> f(A::AbstractArray) = g(A)
f (generic function with 1 method)

julia> @noinline g(@nospecialize(A::AbstractArray)) = A[1]
g (generic function with 1 method)

julia> @code_typed f([1.0])
CodeInfo(
1 ─ %1 = invoke Main.g(_2::AbstractArray)::Float64
└──      return %1
) => Float64
```

Here, the @nospecialize annotation results in the equivalent of

```
f(A::AbstractArray) = invoke(g, Tuple{AbstractArray}, A)
```

ensuring that only one version of native code will be generated for g, one that is generic for any AbstractArray. However, the specific return type is still inferred for both g and f, and this is still used in optimizing the callers of f and g.

source

Base.@specialize – Macro.

```
@specialize
```

Reset the specialization hint for an argument back to the default. For details, see @nospecialize.

source

Base.@nospecializeinfer – Macro.

```
Base.@nospecializeinfer function f(args...)
    @nospecialize ...
    ...
end
Base.@nospecializeinfer f(@nospecialize args...) = ...
```

Tells the compiler to infer f using the declared types of @nospecialized arguments. This can be used to limit the number of compiler-generated specializations during inference.

Example

```
julia> f(A::AbstractArray) = g(A)
f (generic function with 1 method)

julia> @noinline Base.@nospecializeinfer g(@nospecialize(A::AbstractArray)) = A[1]
g (generic function with 1 method)

julia> @code_typed f([1.0])
CodeInfo(
1 ─ %1 = invoke Main.g(_2::AbstractArray)::Any
└──      return %1
) => Any
```

In this example, f will be inferred for each specific type of A, but g will only be inferred once with the declared argument type A::AbstractArray, meaning that the compiler will not likely see the excessive inference time on it while it can not infer the concrete return type of it. Without the @nospecializeinfer, f([1.0]) would infer the return type of g as Float64, indicating that inference ran for g(::Vector{Float64}) despite the prohibition on specialized code generation.

Julia 1.10

Using Base.@nospecializeinfer requires Julia version 1.10.

source

Base.@constprop – Macro.

```
@constprop setting [ex]
```

Control the mode of interprocedural constant propagation for the annotated function.

Two settings are supported:

- `@constprop :aggressive [ex]`: apply constant propagation aggressively. For a method where the return type depends on the value of the arguments, this can yield improved inference results at the cost of additional compile time.
- `@constprop :none [ex]`: disable constant propagation. This can reduce compile times for functions that Julia might otherwise deem worthy of constant-propagation. Common cases are for functions with `Bool`- or `Symbol`-valued arguments or keyword arguments.

`@constprop` can be applied immediately before a function definition or within a function body.

```
# annotate long-form definition
@constprop :aggressive function longdef(x)
  ...
end

# annotate short-form definition
@constprop :aggressive shortdef(x) = ...

# annotate anonymous function that a `do` block creates
f() do
    @constprop :aggressive
    ...
end
```

Julia 1.10

The usage within a function body requires at least Julia 1.10.

source

`Base.gensym` – Function.

```
gensym([tag])
```

Generates a symbol which will not conflict with other variable names (in the same module).

source

`Base.@gensym` – Macro.

```
@gensym
```

Generates a gensym symbol for a variable. For example, `@gensym x y` is transformed into `x = gensym("x"); y = gensym("y")`.

source

`var"name"` – Keyword.

```
var
```

The syntax `var"#example#"` refers to a variable named `Symbol("#example#")`, even though `#example#` is not a valid Julia identifier name.

This can be useful for interoperability with programming languages which have different rules for the construction of valid identifiers. For example, to refer to the R variable `draw.segments`, you can use `var"draw.segments"` in your Julia code.

It is also used to `show` julia source code which has gone through macro hygiene or otherwise contains variable names which can't be parsed normally.

Note that this syntax requires parser support so it is expanded directly by the parser rather than being implemented as a normal string macro `@var_str`.

Julia 1.3

This syntax requires at least Julia 1.3.

source

`Base.@goto` – Macro.

```
@goto name
```

`@goto name` unconditionally jumps to the statement at the location `@label name`.

`@label` and `@goto` cannot create jumps to different top-level statements. Attempts cause an error. To still use `@goto`, enclose the `@label` and `@goto` in a block.

source

`Base.@label` – Macro.

```
@label name
```

Labels a statement with the symbolic label `name`. The label marks the end-point of an unconditional jump with `@goto name`.

source

`Base.SimdLoop.@simd` – Macro.

```
@simd
```

Annotate a `for` loop to allow the compiler to take extra liberties to allow loop re-ordering

Warning

This feature is experimental and could change or disappear in future versions of Julia. Incorrect use of the `@simd` macro may cause unexpected results.

The object iterated over in a `@simd for` loop should be a one-dimensional range. By using `@simd`, you are asserting several properties of the loop:

- It is safe to execute iterations in arbitrary or overlapping order, with special consideration for reduction variables.
- Floating-point operations on reduction variables can be reordered or contracted, possibly causing different results than without `@simd`.

In many cases, Julia is able to automatically vectorize inner for loops without the use of `@simd`. Using `@simd` gives the compiler a little extra leeway to make it possible in more situations. In either case, your inner loop should have the following properties to allow vectorization:

- The loop must be an innermost loop
- The loop body must be straight-line code. Therefore, `@inbounds` is currently needed for all array accesses. The compiler can sometimes turn short &&, `||`, and `?:` expressions into straight-line code if it is safe to evaluate all operands unconditionally. Consider using the `ifelse` function instead of `?:` in the loop if it is safe to do so.
- Accesses must have a stride pattern and cannot be "gathers" (random-index reads) or "scatters" (random-index writes).
- The stride should be unit stride.

Note

The `@simd` does not assert by default that the loop is completely free of loop-carried memory dependencies, which is an assumption that can easily be violated in generic code. If you are writing non-generic code, you can use `@simd ivdep for ... end` to also assert that:

- There exists no loop-carried memory dependencies
- No iteration ever waits on a previous iteration to make forward progress.

source

`Base.@polly` – Macro.

```
@polly
```

Tells the compiler to apply the polyhedral optimizer Polly to a function.

source

`Base.@generated` – Macro.

```
@generated f
```

`@generated` is used to annotate a function which will be generated. In the body of the generated function, only types of arguments can be read (not the values). The function returns a quoted expression evaluated when the function is called. The `@generated` macro should not be used on functions mutating the global scope or depending on mutable elements.

See Metaprogramming for further details.

Examples

```
julia> @generated function bar(x)
           if x <: Integer
               return :(x ^ 2)
           else
               return :(x)
           end
       end
bar (generic function with 1 method)

julia> bar(4)
16

julia> bar("baz")
"baz"
```

source

Base.@assume_effects – Macro.

```
@assume_effects setting... [ex]
```

Override the compiler's effect modeling for the given method or foreign call. `@assume_effects` can be applied immediately before a function definition or within a function body. It can also be applied immediately before a `@ccall` expression.

Julia 1.8

Using `Base.@assume_effects` requires Julia version 1.8.

Examples

```
julia> Base.@assume_effects :terminates_locally function pow(x)
           # this :terminates_locally allows `pow` to be constant-folded
           res = 1
           1 < x < 20 || error("bad pow")
           while x > 1
               res *= x
               x -= 1
           end
           return res
       end
pow (generic function with 1 method)

julia> code_typed() do
           pow(12)
       end
1-element Vector{Any}:
 CodeInfo(
```

```
1 ─     return 479001600
) => Int64

julia> code_typed() do
           map((2,3,4)) do x
               # this :terminates_locally allows this anonymous function to be constant-folded
               Base.@assume_effects :terminates_locally
               res = 1
               1 < x < 20 || error("bad pow")
               while x > 1
                   res *= x
                   x -= 1
               end
               return res
           end
       end
1-element Vector{Any}:
 CodeInfo(
1 ─     return (2, 6, 24)
) => Tuple{Int64, Int64, Int64}

julia> Base.@assume_effects :total !:nothrow @ccall jl_type_intersection(Vector{Int}::Any,
↪  Vector{<:Integer}::Any)::Any
Vector{Int64} (alias for Array{Int64, 1})
```

Julia 1.10

The usage within a function body requires at least Julia 1.10.

Warning

Improper use of this macro causes undefined behavior (including crashes, incorrect answers, or other hard to track bugs). Use with care and only as a last resort if absolutely required. Even in such a case, you SHOULD take all possible steps to minimize the strength of the effect assertion (e.g., do not use `:total` if `:nothrow` would have been sufficient).

In general, each `setting` value makes an assertion about the behavior of the function, without requiring the compiler to prove that this behavior is indeed true. These assertions are made for all world ages. It is thus advisable to limit the use of generic functions that may later be extended to invalidate the assumption (which would cause undefined behavior).

The following `settings` are supported.

- `:consistent`
- `:effect_free`
- `:nothrow`
- `:terminates_globally`
- `:terminates_locally`
- `:notaskstate`

- `:inaccessiblememonly`
- `:foldable`
- `:removable`
- `:total`

Extended help

`:consistent`

The `:consistent` setting asserts that for egal (===) inputs:

- The manner of termination (return value, exception, non-termination) will always be the same.
- If the method returns, the results will always be egal.

> **Note**
>
> This in particular implies that the method must not return a freshly allocated mutable object. Multiple allocations of mutable objects (even with identical contents) are not egal.

> **Note**
>
> The `:consistent`-cy assertion is made world-age wise. More formally, write f for the evaluation of f in world-age i, then we require:
>
> $$i, x, y : x \; y \rightarrow f(x) \; f(y)$$
>
> However, for two world ages i, j s.t. $i \; j$, we may have $f(x) \; f(y)$.
>
> A further implication is that `:consistent` functions may not make their return value dependent on the state of the heap or any other global state that is not constant for a given world age.

> **Note**
>
> The `:consistent`-cy includes all legal rewrites performed by the optimizer. For example, floating-point fastmath operations are not considered `:consistent`, because the optimizer may rewrite them causing the output to not be `:consistent`, even for the same world age (e.g. because one ran in the interpreter, while the other was optimized).

> **Note**
>
> The `:consistent`-cy assertion currrently includes the assertion that the function will not execute any undefined behavior (for any input). Note that undefined behavior may technically cause the function to violate other effect assertions (such as `:nothrow` or `:effect_free`) as well, but we do not model this, and all effects except `:consistent` assume the absence of undefined behavior.

> **Note**
>
> If `:consistent` functions terminate by throwing an exception, that exception itself is not required to meet the egality requirement specified above.

`:effect_free`

The `:effect_free` setting asserts that the method is free of externally semantically visible side effects. The following is an incomplete list of externally semantically visible side effects:

- Changing the value of a global variable.
- Mutating the heap (e.g. an array or mutable value), except as noted below
- Changing the method table (e.g. through calls to eval)
- File/Network/etc. I/O
- Task switching

However, the following are explicitly not semantically visible, even if they may be observable:

- Memory allocations (both mutable and immutable)
- Elapsed time
- Garbage collection
- Heap mutations of objects whose lifetime does not exceed the method (i.e. were allocated in the method and do not escape).
- The returned value (which is externally visible, but not a side effect)

The rule of thumb here is that an externally visible side effect is anything that would affect the execution of the remainder of the program if the function were not executed.

> **Note**
>
> The `:effect_free` assertion is made both for the method itself and any code that is executed by the method. Keep in mind that the assertion must be valid for all world ages and limit use of this assertion accordingly.

`:nothrow`

The `:nothrow` settings asserts that this method does not terminate abnormally (i.e. will either always return a value or never return).

> **Note**
>
> It is permissible for `:nothrow` annotated methods to make use of exception handling internally as long as the exception is not rethrown out of the method itself.

> **Note**
>
> `MethodErrors` and similar exceptions count as abnormal termination.

`:terminates_globally`

The `:terminates_globally` settings asserts that this method will eventually terminate (either normally or abnormally), i.e. does not loop indefinitely.

> **Note**
>
> This `:terminates_globally` assertion covers any other methods called by the annotated method.

> **Note**
>
> The compiler will consider this a strong indication that the method will terminate relatively *quickly* and may (if otherwise legal), call this method at compile time. I.e. it is a bad idea to annotate this setting on a method that *technically*, but not *practically*, terminates.

`:terminates_locally`

The `:terminates_locally` setting is like `:terminates_globally`, except that it only applies to syntactic control flow *within* the annotated method. It is thus a much weaker (and thus safer) assertion that allows for the possibility of non-termination if the method calls some other method that does not terminate.

> **Note**
>
> `:terminates_globally` implies `:terminates_locally`.

`:notaskstate`

The `:notaskstate` setting asserts that the method does not use or modify the local task state (task local storage, RNG state, etc.) and may thus be safely moved between tasks without observable results.

> **Note**
>
> The implementation of exception handling makes use of state stored in the task object. However, this state is currently not considered to be within the scope of `:notaskstate` and is tracked separately using the `:nothrow` effect.

> **Note**
>
> The `:notaskstate` assertion concerns the state of the *currently running task*. If a reference to a Task object is obtained by some other means that does not consider which task is *currently* running, the `:notaskstate` effect need not be tainted. This is true, even if said task object happens to be === to the currently running task.

> **Note**
>
> Access to task state usually also results in the tainting of other effects, such as `:effect_free` (if task state is modified) or `:consistent` (if task state is used in the computation of the result). In particular, code that is not `:notaskstate`, but is `:effect_free` and `:consistent` may still be dead-code-eliminated and thus promoted to `:total`.

`:inaccessiblememonly`

The `:inaccessiblememonly` setting asserts that the method does not access or modify externally accessible mutable memory. This means the method can access or modify mutable memory for newly allocated objects that is not accessible by other methods or top-level execution before return from the method, but it can not access or modify any mutable global state or mutable memory pointed to by its arguments.

> **Note**
>
> Below is an incomplete list of examples that invalidate this assumption:
>
> - a global reference or `getglobal` call to access a mutable global variable
> - a global assignment or `setglobal!` call to perform assignment to a non-constant global variable
> - `setfield!` call that changes a field of a global mutable variable

> **Note**
>
> This `:inaccessiblememonly` assertion covers any other methods called by the annotated method.

`:foldable`

This setting is a convenient shortcut for the set of effects that the compiler requires to be guaranteed to constant fold a call at compile time. It is currently equivalent to the following `settings`:

- `:consistent`
- `:effect_free`
- `:terminates_globally`

> **Note**
>
> This list in particular does not include `:nothrow`. The compiler will still attempt constant propagation and note any thrown error at compile time. Note however, that by the `:consistent`-cy requirements, any such annotated call must consistently throw given the same argument values.

> **Note**
>
> An explicit `@inbounds` annotation inside the function will also disable constant folding and not be overriden by `:foldable`.

`:removable`

This setting is a convenient shortcut for the set of effects that the compiler requires to be guaranteed to delete a call whose result is unused at compile time. It is currently equivalent to the following `settings`:

- `:effect_free`

- `:nothrow`
- `:terminates_globally`

`:total`

This `setting` is the maximum possible set of effects. It currently implies the following other `settings`:

- `:consistent`
- `:effect_free`
- `:nothrow`
- `:terminates_globally`
- `:notaskstate`
- `:inaccessiblememonly`

> **Warning**
>
> `:total` is a very strong assertion and will likely gain additional semantics in future versions of Julia (e.g. if additional effects are added and included in the definition of `:total`). As a result, it should be used with care. Whenever possible, prefer to use the minimum possible set of specific effect assertions required for a particular application. In cases where a large number of effect overrides apply to a set of functions, a custom macro is recommended over the use of `:total`.

Negated effects

Effect names may be prefixed by `!` to indicate that the effect should be removed from an earlier meta effect. For example, `:total !:nothrow` indicates that while the call is generally total, it may however throw.

source

`Base.@deprecate` – Macro.

```
@deprecate old new [export_old=true]
```

Deprecate method `old` and specify the replacement call new, defining a new method `old` with the specified signature in the process.

To prevent `old` from being exported, set `export_old` to `false`.

> **Julia 1.5**
>
> As of Julia 1.5, functions defined by `@deprecate` do not print warning when `julia` is run without the `--depwarn=yes` flag set, as the default value of `--depwarn` option is no. The warnings are printed from tests run by `Pkg.test()`.

Examples

```
julia> @deprecate old(x) new(x)
old (generic function with 1 method)

julia> @deprecate old(x) new(x) false
old (generic function with 1 method)
```

Calls to `@deprecate` without explicit type-annotations will define deprecated methods accepting any number of positional and keyword arguments of type `Any`.

> **Julia 1.9**
>
> Keyword arguments are forwarded when there is no explicit type annotation as of Julia 1.9. For older versions, you can manually forward positional and keyword arguments by doing `@deprecate old(args...; kwargs...) new(args...; kwargs...)`.

To restrict deprecation to a specific signature, annotate the arguments of `old`. For example,

```
julia> new(x::Int) = x;

julia> new(x::Float64) = 2x;

julia> @deprecate old(x::Int) new(x);

julia> methods(old)
# 1 method for generic function "old" from Main:
 [1] old(x::Int64)
     @ deprecated.jl:94
```

will define and deprecate a method `old(x::Int)` that mirrors `new(x::Int)` but will not define nor deprecate the method `old(x::Float64)`.

source

42.11 Missing Values

`Base.Missing` – Type.

```
Missing
```

A type with no fields whose singleton instance `missing` is used to represent missing values.

See also: `skipmissing`, `nonmissingtype`, `Nothing`.

source

`Base.missing` – Constant.

```
missing
```

The singleton instance of type `Missing` representing a missing value.

See also: `NaN`, `skipmissing`, `nonmissingtype`.

source

Base.coalesce – Function.

```
coalesce(x...)
```

Return the first value in the arguments which is not equal to `missing`, if any. Otherwise return `missing`.

See also `skipmissing`, `something`, `@coalesce`.

Examples

```
julia> coalesce(missing, 1)
1

julia> coalesce(1, missing)
1

julia> coalesce(nothing, 1)  # returns `nothing`

julia> coalesce(missing, missing)
missing
```

source

Base.@coalesce – Macro.

```
@coalesce(x...)
```

Short-circuiting version of `coalesce`.

Examples

```
julia> f(x) = (println("f($x)"); missing);

julia> a = 1;

julia> a = @coalesce a f(2) f(3) error("`a` is still missing")
1

julia> b = missing;

julia> b = @coalesce b f(2) f(3) error("`b` is still missing")
f(2)
f(3)
ERROR: `b` is still missing
[...]
```

Julia 1.7

This macro is available as of Julia 1.7.

source

Base.ismissing – Function.

```
ismissing(x)
```

Indicate whether x is `missing`.

See also: `skipmissing`, `isnothing`, `isnan`.

source

Base.skipmissing – Function.

```
skipmissing(itr)
```

Return an iterator over the elements in `itr` skipping `missing` values. The returned object can be indexed using indices of `itr` if the latter is indexable. Indices corresponding to missing values are not valid: they are skipped by `keys` and `eachindex`, and a `MissingException` is thrown when trying to use them.

Use `collect` to obtain an `Array` containing the non-`missing` values in `itr`. Note that even if `itr` is a multidimensional array, the result will always be a `Vector` since it is not possible to remove missings while preserving dimensions of the input.

See also `coalesce`, `ismissing`, `something`.

Examples

```
julia> x = skipmissing([1, missing, 2])
skipmissing(Union{Missing, Int64}[1, missing, 2])

julia> sum(x)
3

julia> x[1]
1

julia> x[2]
ERROR: MissingException: the value at index (2,) is missing
[...]

julia> argmax(x)
3

julia> collect(keys(x))
2-element Vector{Int64}:
 1
 3
```

```
julia> collect(skipmissing([1, missing, 2]))
2-element Vector{Int64}:
 1
 2

julia> collect(skipmissing([1 missing; 2 missing]))
2-element Vector{Int64}:
 1
 2
```

source

Base.nonmissingtype – Function.

```
nonmissingtype(T::Type)
```

If T is a union of types containing Missing, return a new type with Missing removed.

Examples

```
julia> nonmissingtype(Union{Int64,Missing})
Int64

julia> nonmissingtype(Any)
Any
```

Julia 1.3

This function is exported as of Julia 1.3.

source

42.12 System

Base.run – Function.

```
run(command, args...; wait::Bool = true)
```

Run a command object, constructed with backticks (see the Running External Programs section in the manual). Throws an error if anything goes wrong, including the process exiting with a non-zero status (when wait is true).

The args... allow you to pass through file descriptors to the command, and are ordered like regular unix file descriptors (eg stdin, stdout, stderr, FD(3), FD(4)...).

If wait is false, the process runs asynchronously. You can later wait for it and check its exit status by calling success on the returned process object.

When wait is false, the process' I/O streams are directed to devnull. When wait is true, I/O streams are shared with the parent process. Use pipeline to control I/O redirection.

source

Base.devnull – Constant.

```
devnull
```

Used in a stream redirect to discard all data written to it. Essentially equivalent to /dev/null on Unix or NUL on Windows. Usage:

```
run(pipeline(`cat test.txt`, devnull))
```

source

Base.success – Function.

```
success(command)
```

Run a command object, constructed with backticks (see the Running External Programs section in the manual), and tell whether it was successful (exited with a code of 0). An exception is raised if the process cannot be started.

source

Base.process_running – Function.

```
process_running(p::Process)
```

Determine whether a process is currently running.

source

Base.process_exited – Function.

```
process_exited(p::Process)
```

Determine whether a process has exited.

source

Base.kill – Method.

```
kill(p::Process, signum=Base.SIGTERM)
```

Send a signal to a process. The default is to terminate the process. Returns successfully if the process has already exited, but throws an error if killing the process failed for other reasons (e.g. insufficient permissions).

source

`Base.Sys.set_process_title` – Function.

```
Sys.set_process_title(title::AbstractString)
```

Set the process title. No-op on some operating systems.

source

`Base.Sys.get_process_title` – Function.

```
Sys.get_process_title()
```

Get the process title. On some systems, will always return an empty string.

source

`Base.ignorestatus` – Function.

```
ignorestatus(command)
```

Mark a command object so that running it will not throw an error if the result code is non-zero.

source

`Base.detach` – Function.

```
detach(command)
```

Mark a command object so that it will be run in a new process group, allowing it to outlive the julia process, and not have Ctrl-C interrupts passed to it.

source

`Base.Cmd` – Type.

```
Cmd(cmd::Cmd; ignorestatus, detach, windows_verbatim, windows_hide, env, dir)
Cmd(exec::Vector{String})
```

Construct a new `Cmd` object, representing an external program and arguments, from `cmd`, while changing the settings of the optional keyword arguments:

- `ignorestatus::Bool`: If `true` (defaults to `false`), then the `Cmd` will not throw an error if the return code is nonzero.
- `detach::Bool`: If `true` (defaults to `false`), then the `Cmd` will be run in a new process group, allowing it to outlive the `julia` process and not have Ctrl-C passed to it.

- `windows_verbatim::Bool`: If `true` (defaults to `false`), then on Windows the `Cmd` will send a command-line string to the process with no quoting or escaping of arguments, even arguments containing spaces. (On Windows, arguments are sent to a program as a single "command-line" string, and programs are responsible for parsing it into arguments. By default, empty arguments and arguments with spaces or tabs are quoted with double quotes `"` in the command line, and `\` or `"` are preceded by backslashes. `windows_verbatim=true` is useful for launching programs that parse their command line in nonstandard ways.) Has no effect on non-Windows systems.
- `windows_hide::Bool`: If `true` (defaults to `false`), then on Windows no new console window is displayed when the `Cmd` is executed. This has no effect if a console is already open or on non-Windows systems.
- `env`: Set environment variables to use when running the `Cmd`. `env` is either a dictionary mapping strings to strings, an array of strings of the form `"var=val"`, an array or tuple of `"var"=>val` pairs. In order to modify (rather than replace) the existing environment, initialize `env` with `copy(ENV)` and then set `env["var"]=val` as desired. To add to an environment block within a `Cmd` object without replacing all elements, use `addenv()` which will return a `Cmd` object with the updated environment.
- `dir::AbstractString`: Specify a working directory for the command (instead of the current directory).

For any keywords that are not specified, the current settings from `cmd` are used.

Note that the `Cmd(exec)` constructor does not create a copy of `exec`. Any subsequent changes to `exec` will be reflected in the `Cmd` object.

The most common way to construct a `Cmd` object is with command literals (backticks), e.g.

```
`ls -l`
```

This can then be passed to the `Cmd` constructor to modify its settings, e.g.

```
Cmd(`echo "Hello world"`, ignorestatus=true, detach=false)
```

source

`Base.setenv` – Function.

```
setenv(command::Cmd, env; dir)
```

Set environment variables to use when running the given `command`. `env` is either a dictionary mapping strings to strings, an array of strings of the form `"var=val"`, or zero or more `"var"=>val` pair arguments. In order to modify (rather than replace) the existing environment, create `env` through `copy(ENV)` and then setting `env["var"]=val` as desired, or use `addenv`.

The `dir` keyword argument can be used to specify a working directory for the command. `dir` defaults to the currently set `dir` for `command` (which is the current working directory if not specified already).

See also `Cmd`, `addenv`, `ENV`, `pwd`.

source

`Base.addenv` – Function.

```
addenv(command::Cmd, env...; inherit::Bool = true)
```

Merge new environment mappings into the given `Cmd` object, returning a new `Cmd` object. Duplicate keys are replaced. If `command` does not contain any environment values set already, it inherits the current environment at time of `addenv()` call if `inherit` is `true`. Keys with value `nothing` are deleted from the env.

See also `Cmd`, `setenv`, `ENV`.

> **Julia 1.6**
>
> This function requires Julia 1.6 or later.

source

`Base.withenv` – Function.

```
withenv(f, kv::Pair...)
```

Execute `f` in an environment that is temporarily modified (not replaced as in `setenv`) by zero or more `"var"=>val` arguments `kv`. `withenv` is generally used via the `withenv(kv...) do ... end` syntax. A value of `nothing` can be used to temporarily unset an environment variable (if it is set). When `withenv` returns, the original environment has been restored.

> **Warning**
>
> Changing the environment is not thread-safe. For running external commands with a different environment from the parent process, prefer using `addenv` over `withenv`.

source

`Base.setcpuaffinity` – Function.

```
setcpuaffinity(original_command::Cmd, cpus) -> command::Cmd
```

Set the CPU affinity of the `command` by a list of CPU IDs (1-based) `cpus`. Passing `cpus = nothing` means to unset the CPU affinity if the `original_command` has any.

This function is supported only in Linux and Windows. It is not supported in macOS because libuv does not support affinity setting.

> **Julia 1.8**
>
> This function requires at least Julia 1.8.

Examples

In Linux, the `taskset` command line program can be used to see how `setcpuaffinity` works.

```
julia> run(setcpuaffinity(`sh -c 'taskset -p $$'`, [1, 2, 5]));
pid 2273's current affinity mask: 13
```

Note that the mask value 13 reflects that the first, second, and the fifth bits (counting from the least significant position) are turned on:

```
julia> 0b010011
0x13
```

source

Base.pipeline – Method.

```
pipeline(from, to, ...)
```

Create a pipeline from a data source to a destination. The source and destination can be commands, I/O streams, strings, or results of other pipeline calls. At least one argument must be a command. Strings refer to filenames. When called with more than two arguments, they are chained together from left to right. For example, pipeline(a,b,c) is equivalent to pipeline(pipeline(a,b),c). This provides a more concise way to specify multi-stage pipelines.

Examples:

```
run(pipeline(`ls`, `grep xyz`))
run(pipeline(`ls`, "out.txt"))
run(pipeline("out.txt", `grep xyz`))
```

source

Base.pipeline – Method.

```
pipeline(command; stdin, stdout, stderr, append=false)
```

Redirect I/O to or from the given command. Keyword arguments specify which of the command's streams should be redirected. append controls whether file output appends to the file. This is a more general version of the 2-argument pipeline function. pipeline(from, to) is equivalent to pipeline(from, stdout=to) when from is a command, and to pipeline(to, stdin=from) when from is another kind of data source.

Examples:

```
run(pipeline(`dothings`, stdout="out.txt", stderr="errs.txt"))
run(pipeline(`update`, stdout="log.txt", append=true))
```

source

Base.Libc.gethostname – Function.

```
gethostname() -> String
```

Get the local machine's host name.

source

Base.Libc.getpid – Function.

```
getpid() -> Int32
```

Get Julia's process ID.

source

```
getpid(process) -> Int32
```

Get the child process ID, if it still exists.

Julia 1.1

This function requires at least Julia 1.1.

source

Base.Libc.time – Method.

```
time() -> Float64
```

Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution.

source

Base.time_ns – Function.

```
time_ns() -> UInt64
```

Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

source

Base.@time – Macro.

```
@time expr
@time "description" expr
```

A macro to execute an expression, printing the time it took to execute, the number of allocations, and the total number of bytes its execution caused to be allocated, before returning the value of the expression. Any time spent garbage collecting (gc), compiling new code, or recompiling invalidated code is shown as a percentage.

Optionally provide a description string to print before the time report.

In some cases the system will look inside the `@time` expression and compile some of the called code before execution of the top-level expression begins. When that happens, some compilation time will not be counted. To include this time you can run `@time @eval ...`.

See also `@showtime`, `@timev`, `@timed`, `@elapsed`, `@allocated`, and `@allocations`.

Note

For more serious benchmarking, consider the `@btime` macro from the BenchmarkTools.jl package which among other things evaluates the function multiple times in order to reduce noise.

Julia 1.8

The option to add a description was introduced in Julia 1.8.

Recompilation time being shown separately from compilation time was introduced in Julia 1.8

```
julia> x = rand(10,10);

julia> @time x * x;
  0.606588 seconds (2.19 M allocations: 116.555 MiB, 3.75% gc time, 99.94% compilation time)

julia> @time x * x;
  0.000009 seconds (1 allocation: 896 bytes)

julia> @time begin
           sleep(0.3)
           1+1
       end
  0.301395 seconds (8 allocations: 336 bytes)
2

julia> @time "A one second sleep" sleep(1)
A one second sleep: 1.005750 seconds (5 allocations: 144 bytes)

julia> for loop in 1:3
            @time loop sleep(1)
        end
1: 1.006760 seconds (5 allocations: 144 bytes)
2: 1.001263 seconds (5 allocations: 144 bytes)
3: 1.003676 seconds (5 allocations: 144 bytes)
```

source

`Base.@showtime` – Macro.

```
@showtime expr
```

Like `@time` but also prints the expression being evaluated for reference.

Julia 1.8

This macro was added in Julia 1.8.

See also `@time`.

```
julia> @showtime sleep(1)
sleep(1): 1.002164 seconds (4 allocations: 128 bytes)
```

source

`Base.@timev` – Macro.

```
@timev expr
@timev "description" expr
```

This is a verbose version of the `@time` macro. It first prints the same information as `@time`, then any non-zero memory allocation counters, and then returns the value of the expression.

Optionally provide a description string to print before the time report.

Julia 1.8

The option to add a description was introduced in Julia 1.8.

See also `@time`, `@timed`, `@elapsed`, `@allocated`, and `@allocations`.

```
julia> x = rand(10,10);

julia> @timev x * x;
  0.546770 seconds (2.20 M allocations: 116.632 MiB, 4.23% gc time, 99.94% compilation time)
elapsed time (ns): 546769547
gc time (ns):      23115606
bytes allocated:   122297811
pool allocs:       2197930
non-pool GC allocs:1327
malloc() calls:    36
realloc() calls:   5
GC pauses:         3

julia> @timev x * x;
  0.000010 seconds (1 allocation: 896 bytes)
elapsed time (ns): 9848
bytes allocated:   896
pool allocs:       1
```

source

Base.@timed – Macro.

```
@timed
```

A macro to execute an expression, and return the value of the expression, elapsed time, total bytes allocated, garbage collection time, and an object with various memory allocation counters.

In some cases the system will look inside the `@timed` expression and compile some of the called code before execution of the top-level expression begins. When that happens, some compilation time will not be counted. To include this time you can run `@timed @eval ...`.

See also `@time`, `@timev`, `@elapsed`, `@allocated`, and `@allocations`.

```
julia> stats = @timed rand(10^6);

julia> stats.time
0.006634834

julia> stats.bytes
8000256

julia> stats.gctime
0.0055765

julia> propertynames(stats.gcstats)
(:allocd, :malloc, :realloc, :poolalloc, :bigalloc, :freecall, :total_time, :pause, :full_sweep)

julia> stats.gcstats.total_time
5576500
```

Julia 1.5

The return type of this macro was changed from `Tuple` to `NamedTuple` in Julia 1.5.

source

Base.@elapsed – Macro.

```
@elapsed
```

A macro to evaluate an expression, discarding the resulting value, instead returning the number of seconds it took to execute as a floating-point number.

In some cases the system will look inside the `@elapsed` expression and compile some of the called code before execution of the top-level expression begins. When that happens, some compilation time will not be counted. To include this time you can run `@elapsed @eval ...`.

See also `@time`, `@timev`, `@timed`, `@allocated`, and `@allocations`.

```
julia> @elapsed sleep(0.3)
0.301391426
```

source

Base.@allocated – Macro.

```
@allocated
```

A macro to evaluate an expression, discarding the resulting value, instead returning the total number of bytes allocated during evaluation of the expression.

See also `@allocations`, `@time`, `@timev`, `@timed`, and `@elapsed`.

```
julia> @allocated rand(10^6)
8000080
```

source

Base.@allocations – Macro.

```
@allocations
```

A macro to evaluate an expression, discard the resulting value, and instead return the total number of allocations during evaluation of the expression.

See also `@allocated`, `@time`, `@timev`, `@timed`, and `@elapsed`.

```
julia> @allocations rand(10^6)
2
```

Julia 1.9

This macro was added in Julia 1.9.

source

Base.EnvDict – Type.

```
EnvDict() -> EnvDict
```

A singleton of this type provides a hash table interface to environment variables.

source

Base.ENV – Constant.

```
ENV
```

Reference to the singleton `EnvDict`, providing a dictionary interface to system environment variables.

(On Windows, system environment variables are case-insensitive, and ENV correspondingly converts all keys to uppercase for display, iteration, and copying. Portable code should not rely on the ability to distinguish variables by case, and should beware that setting an ostensibly lowercase variable may result in an uppercase ENV key.)

> **Warning**
>
> Mutating the environment is not thread-safe.

Examples

```
julia> ENV
Base.EnvDict with "50" entries:
  "SECURITYSESSIONID"            => "123"
  "USER"                         => "username"
  "MallocNanoZone"               => "0"
  ⋮                              => ⋮

julia> ENV["JULIA_EDITOR"] = "vim"
"vim"

julia> ENV["JULIA_EDITOR"]
"vim"
```

See also: `withenv`, `addenv`.

source

`Base.Sys.STDLIB` – Constant.

```
Sys.STDLIB::String
```

A string containing the full path to the directory containing the `stdlib` packages.

source

`Base.Sys.isunix` – Function.

```
Sys.isunix([os])
```

Predicate for testing if the OS provides a Unix-like interface. See documentation in Handling Operating System Variation.

source

`Base.Sys.isapple` – Function.

```
Sys.isapple([os])
```

Predicate for testing if the OS is a derivative of Apple Macintosh OS X or Darwin. See documentation in Handling Operating System Variation.

source

Base.Sys.islinux – Function.

```
Sys.islinux([os])
```

Predicate for testing if the OS is a derivative of Linux. See documentation in Handling Operating System Variation.

source

Base.Sys.isbsd – Function.

```
Sys.isbsd([os])
```

Predicate for testing if the OS is a derivative of BSD. See documentation in Handling Operating System Variation.

Note

The Darwin kernel descends from BSD, which means that `Sys.isbsd()` is `true` on macOS systems. To exclude macOS from a predicate, use `Sys.isbsd() && !Sys.isapple()`.

source

Base.Sys.isfreebsd – Function.

```
Sys.isfreebsd([os])
```

Predicate for testing if the OS is a derivative of FreeBSD. See documentation in Handling Operating System Variation.

Note

Not to be confused with `Sys.isbsd()`, which is `true` on FreeBSD but also on other BSD-based systems. `Sys.isfreebsd()` refers only to FreeBSD.

Julia 1.1

This function requires at least Julia 1.1.

source

Base.Sys.isopenbsd – Function.

```
Sys.isopenbsd([os])
```

Predicate for testing if the OS is a derivative of OpenBSD. See documentation in Handling Operating System Variation.

Note

Not to be confused with `Sys.isbsd()`, which is `true` on OpenBSD but also on other BSD-based systems. `Sys.isopenbsd()` refers only to OpenBSD.

Julia 1.1

This function requires at least Julia 1.1.

source

Base.Sys.isnetbsd – Function.

```
Sys.isnetbsd([os])
```

Predicate for testing if the OS is a derivative of NetBSD. See documentation in Handling Operating System Variation.

Note

Not to be confused with `Sys.isbsd()`, which is `true` on NetBSD but also on other BSD-based systems. `Sys.isnetbsd()` refers only to NetBSD.

Julia 1.1

This function requires at least Julia 1.1.

source

Base.Sys.isdragonfly – Function.

```
Sys.isdragonfly([os])
```

Predicate for testing if the OS is a derivative of DragonFly BSD. See documentation in Handling Operating System Variation.

Note

Not to be confused with `Sys.isbsd()`, which is `true` on DragonFly but also on other BSD-based systems. `Sys.isdragonfly()` refers only to DragonFly.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

source

Base.Sys.iswindows – Function.

```
Sys.iswindows([os])
```

Predicate for testing if the OS is a derivative of Microsoft Windows NT. See documentation in Handling Operating System Variation.

source

Base.Sys.windows_version – Function.

```
Sys.windows_version()
```

Return the version number for the Windows NT Kernel as a `VersionNumber`, i.e. `v"major.minor.build"`, or `v"0.0.0"` if this is not running on Windows.

source

Base.Sys.free_memory – Function.

```
Sys.free_memory()
```

Get the total free memory in RAM in bytes.

source

Base.Sys.total_memory – Function.

```
Sys.total_memory()
```

Get the total memory in RAM (including that which is currently used) in bytes. This amount may be constrained, e.g., by Linux control groups. For the unconstrained amount, see `Sys.total_physical_memory()`.

source

Base.Sys.free_physical_memory – Function.

```
Sys.free_physical_memory()
```

Get the free memory of the system in bytes. The entire amount may not be available to the current process; use `Sys.free_memory()` for the actually available amount.

source

`Base.Sys.total_physical_memory` – Function.

```
Sys.total_physical_memory()
```

Get the total memory in RAM (including that which is currently used) in bytes. The entire amount may not be available to the current process; see `Sys.total_memory()`.

source

`Base.Sys.uptime` – Function.

```
Sys.uptime()
```

Gets the current system uptime in seconds.

source

`Base.Sys.isjsvm` – Function.

```
Sys.isjsvm([os])
```

Predicate for testing if Julia is running in a JavaScript VM (JSVM), including e.g. a WebAssembly JavaScript embedding in a web browser.

Julia 1.2

This function requires at least Julia 1.2.

source

`Base.Sys.loadavg` – Function.

```
Sys.loadavg()
```

Get the load average. See: https://en.wikipedia.org/wiki/Load_(computing).

source

`Base.Sys.isexecutable` – Function.

```
Sys.isexecutable(path::String)
```

Return `true` if the given `path` has executable permissions.

Note

Prior to Julia 1.6, this did not correctly interrogate filesystem ACLs on Windows, therefore it would return `true` for any file. From Julia 1.6 on, it correctly determines whether the file is marked as executable or not.

source

Base.@static – Macro.

```
@static
```

Partially evaluate an expression at parse time.

For example, `@static Sys.iswindows() ? foo : bar` will evaluate `Sys.iswindows()` and insert either `foo` or `bar` into the expression. This is useful in cases where a construct would be invalid on other platforms, such as a `ccall` to a non-existent function. `@static if Sys.isapple() foo end` and `@static foo <&&,||> bar` are also valid syntax.

source

42.13 Versioning

Base.VersionNumber – Type.

```
VersionNumber
```

Version number type which follows the specifications of semantic versioning (semver), composed of major, minor and patch numeric values, followed by pre-release and build alpha-numeric annotations.

`VersionNumber` objects can be compared with all of the standard comparison operators (==, <, <=, etc.), with the result following semver rules.

See also `@v_str` to efficiently construct `VersionNumber` objects from semver-format literal strings, `VERSION` for the `VersionNumber` of Julia itself, and Version Number Literals in the manual.

Examples

```
julia> a = VersionNumber(1, 2, 3)
v"1.2.3"

julia> a >= v"1.2"
true

julia> b = VersionNumber("2.0.1-rc1")
v"2.0.1-rc1"

julia> b >= v"2.0.1"
false
```

source

Base.@v_str – Macro.

```
@v_str
```

String macro used to parse a string to a `VersionNumber`.

Examples

```
julia> v"1.2.3"
v"1.2.3"

julia> v"2.0.1-rc1"
v"2.0.1-rc1"
```

source

42.14 Errors

`Base.error` – Function.

```
error(message::AbstractString)
```

Raise an `ErrorException` with the given message.

source

```
error(msg...)
```

Raise an `ErrorException` with the given message.

source

`Core.throw` – Function.

```
throw(e)
```

Throw an object as an exception.

See also: `rethrow`, `error`.

source

`Base.rethrow` – Function.

```
rethrow()
```

Rethrow the current exception from within a `catch` block. The rethrown exception will continue propagation as if it had not been caught.

> **Note**
>
> The alternative form `rethrow(e)` allows you to associate an alternative exception object `e` with the current backtrace. However this misrepresents the program state at the time of the error so you're encouraged to instead throw a new exception using `throw(e)`. In Julia 1.1 and above, using `throw(e)` will preserve the root cause exception on the stack, as described in `current_exceptions`.

source

`Base.backtrace` – Function.

```
backtrace()
```

Get a backtrace object for the current program point.

source

`Base.catch_backtrace` – Function.

```
catch_backtrace()
```

Get the backtrace of the current exception, for use within `catch` blocks.

source

`Base.current_exceptions` – Function.

```
current_exceptions(task::Task=current_task(); [backtrace::Bool=true])
```

Get the stack of exceptions currently being handled. For nested catch blocks there may be more than one current exception in which case the most recently thrown exception is last in the stack. The stack is returned as an `ExceptionStack` which is an AbstractVector of named tuples `(exception,backtrace)`. If `backtrace` is false, the backtrace in each pair will be set to `nothing`.

Explicitly passing `task` will return the current exception stack on an arbitrary task. This is useful for inspecting tasks which have failed due to uncaught exceptions.

> **Julia 1.7**
>
> This function went by the experimental name `catch_stack()` in Julia 1.1–1.6, and had a plain Vector-of-tuples as a return type.

source

`Base.@assert` – Macro.

```
@assert cond [text]
```

Throw an `AssertionError` if `cond` is `false`. Preferred syntax for writing assertions. Message `text` is optionally displayed upon assertion failure.

> **Warning**
>
> An assert might be disabled at various optimization levels. Assert should therefore only be used as a debugging tool and not used for authentication verification (e.g., verifying passwords), nor should side effects needed for the function to work correctly be used inside of asserts.

Examples

```
julia> @assert iseven(3) "3 is an odd number!"
ERROR: AssertionError: 3 is an odd number!

julia> @assert isodd(3) "What even are numbers?"
```

source

`Base.Experimental.register_error_hint` – Function.

```
Experimental.register_error_hint(handler, exceptiontype)
```

Register a "hinting" function `handler(io, exception)` that can suggest potential ways for users to circumvent errors. `handler` should examine `exception` to see whether the conditions appropriate for a hint are met, and if so generate output to `io`. Packages should call `register_error_hint` from within their `__init__` function.

For specific exception types, `handler` is required to accept additional arguments:

- `MethodError`: provide `handler(io, exc::MethodError, argtypes, kwargs)`, which splits the combined arguments into positional and keyword arguments.

When issuing a hint, the output should typically start with `\n`.

If you define custom exception types, your `showerror` method can support hints by calling `Experimental.show_error_hints`.

Example

```
julia> module Hinter

       only_int(x::Int)      = 1
       any_number(x::Number) = 2

       function __init__()
           Base.Experimental.register_error_hint(MethodError) do io, exc, argtypes, kwargs
               if exc.f == only_int
                    # Color is not necessary, this is just to show it's possible.
                    print(io, "\nDid you mean to call ")
```

```
                  printstyled(io, "`any_number`?", color=:cyan)
              end
          end
      end

      end
```

Then if you call `Hinter.only_int` on something that isn't an `Int` (thereby triggering a `MethodError`), it issues the hint:

```
julia> Hinter.only_int(1.0)
ERROR: MethodError: no method matching only_int(::Float64)
Did you mean to call `any_number`?
Closest candidates are:
    ...
```

Julia 1.5

Custom error hints are available as of Julia 1.5.

Warning

This interface is experimental and subject to change or removal without notice. To insulate yourself against changes, consider putting any registrations inside an `if isdefined(Base.Experimental, :register_error_hint) ... end` block.

source

`Base.Experimental.show_error_hints` – Function.

```
Experimental.show_error_hints(io, ex, args...)
```

Invoke all handlers from `Experimental.register_error_hint` for the particular exception type `typeof(ex)`. `args` must contain any other arguments expected by the handler for that type.

Julia 1.5

Custom error hints are available as of Julia 1.5.

Warning

This interface is experimental and subject to change or removal without notice.

source

`Core.ArgumentError` – Type.

```
ArgumentError(msg)
```

The arguments passed to a function are invalid. `msg` is a descriptive error message.

source

`Core.AssertionError` – Type.

```
AssertionError([msg])
```

The asserted condition did not evaluate to `true`. Optional argument `msg` is a descriptive error string.

Examples

```
julia> @assert false "this is not true"
ERROR: AssertionError: this is not true
```

`AssertionError` is usually thrown from `@assert`.

source

`Core.BoundsError` – Type.

```
BoundsError([a],[i])
```

An indexing operation into an array, `a`, tried to access an out-of-bounds element at index `i`.

Examples

```
julia> A = fill(1.0, 7);

julia> A[8]
ERROR: BoundsError: attempt to access 7-element Vector{Float64} at index [8]


julia> B = fill(1.0, (2,3));

julia> B[2, 4]
ERROR: BoundsError: attempt to access 2×3 Matrix{Float64} at index [2, 4]


julia> B[9]
ERROR: BoundsError: attempt to access 2×3 Matrix{Float64} at index [9]
```

source

`Base.CompositeException` – Type.

```
CompositeException
```

Wrap a `Vector` of exceptions thrown by a `Task` (e.g. generated from a remote worker over a channel or an asynchronously executing local I/O write or a remote worker under `pmap`) with information about the series of exceptions. For example, if a group of workers are executing several tasks, and multiple workers fail, the resulting `CompositeException` will contain a ”bundle” of information from each worker indicating where and why the exception(s) occurred.

source

`Base.DimensionMismatch` – Type.

```
DimensionMismatch([msg])
```

The objects called do not have matching dimensionality. Optional argument `msg` is a descriptive error string.

source

`Core.DivideError` – Type.

```
DivideError()
```

Integer division was attempted with a denominator value of 0.

Examples

```
julia> 2/0
Inf

julia> div(2, 0)
ERROR: DivideError: integer division error
Stacktrace:
[...]
```

source

`Core.DomainError` – Type.

```
DomainError(val)
DomainError(val, msg)
```

The argument `val` to a function or constructor is outside the valid domain.

Examples

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

source

Base.EOFError – Type.

```
EOFError()
```

No more data was available to read from a file or stream.

source

Core.ErrorException – Type.

```
ErrorException(msg)
```

Generic error type. The error message, in the .msg field, may provide more specific details.

Examples

```
julia> ex = ErrorException("I've done a bad thing");

julia> ex.msg
"I've done a bad thing"
```

source

Core.InexactError – Type.

```
InexactError(name::Symbol, T, val)
```

Cannot exactly convert val to type T in a method of function name.

Examples

```
julia> convert(Float64, 1+2im)
ERROR: InexactError: Float64(1 + 2im)
Stacktrace:
[...]
```

source

Core.InterruptException – Type.

```
InterruptException()
```

The process was stopped by a terminal interrupt (CTRL+C).

Note that, in Julia script started without `-i` (interactive) option, `InterruptException` is not thrown by default. Calling `Base.exit_on_sigint(false)` in the script can recover the behavior of the REPL. Alternatively, a Julia script can be started with

```
julia -e "include(popfirst!(ARGS))" script.jl
```

to let `InterruptException` be thrown by CTRL+C during the execution.

source

`Base.KeyError` – Type.

```
KeyError(key)
```

An indexing operation into an `AbstractDict` (`Dict`) or `Set` like object tried to access or delete a non-existent element.

source

`Core.LoadError` – Type.

```
LoadError(file::AbstractString, line::Int, error)
```

An error occurred while `include`ing, `require`ing, or `using` a file. The error specifics should be available in the `.error` field.

> **Julia 1.7**
>
> LoadErrors are no longer emitted by `@macroexpand`, `@macroexpand1`, and `macroexpand` as of Julia 1.7.

source

`Core.MethodError` – Type.

```
MethodError(f, args)
```

A method with the required type signature does not exist in the given generic function. Alternatively, there is no unique most-specific method.

source

`Base.MissingException` – Type.

```
MissingException(msg)
```

Exception thrown when a `missing` value is encountered in a situation where it is not supported. The error message, in the `msg` field may provide more specific details.

source

`Core.OutOfMemoryError` – Type.

```
OutOfMemoryError()
```

An operation allocated too much memory for either the system or the garbage collector to handle properly.

source

`Core.ReadOnlyMemoryError` – Type.

```
ReadOnlyMemoryError()
```

An operation tried to write to memory that is read-only.

source

`Core.OverflowError` – Type.

```
OverflowError(msg)
```

The result of an expression is too large for the specified type and will cause a wraparound.

source

`Base.ProcessFailedException` – Type.

```
ProcessFailedException
```

Indicates problematic exit status of a process. When running commands or pipelines, this is thrown to indicate a nonzero exit code was returned (i.e. that the invoked process failed).

source

`Base.TaskFailedException` – Type.

```
TaskFailedException
```

This exception is thrown by a `wait(t)` call when task `t` fails. `TaskFailedException` wraps the failed task `t`.

source

`Core.StackOverflowError` – Type.

```
StackOverflowError()
```

The function call grew beyond the size of the call stack. This usually happens when a call recurses infinitely.

source

`Base.SystemError` – Type.

```
SystemError(prefix::AbstractString, [errno::Int32])
```

A system call failed with an error code (in the `errno` global variable).

source

`Core.TypeError` – Type.

```
TypeError(func::Symbol, context::AbstractString, expected::Type, got)
```

A type assertion failure, or calling an intrinsic function with an incorrect argument type.

source

`Core.UndefKeywordError` – Type.

```
UndefKeywordError(var::Symbol)
```

The required keyword argument `var` was not assigned in a function call.

Examples

```
julia> function my_func(;my_arg)
           return my_arg + 1
       end
my_func (generic function with 1 method)

julia> my_func()
ERROR: UndefKeywordError: keyword argument `my_arg` not assigned
Stacktrace:
 [1] my_func() at ./REPL[1]:2
 [2] top-level scope at REPL[2]:1
```

source

`Core.UndefRefError` – Type.

```
UndefRefError()
```

The item or field is not defined for the given object.

Examples

```
julia> struct MyType
           a::Vector{Int}
           MyType() = new()
       end

julia> A = MyType()
MyType(#undef)

julia> A.a
ERROR: UndefRefError: access to undefined reference
Stacktrace:
[...]
```

source

Core.UndefVarError – Type.

```
UndefVarError(var::Symbol)
```

A symbol in the current scope is not defined.

Examples

```
julia> a
ERROR: UndefVarError: `a` not defined

julia> a = 1;

julia> a
1
```

source

Base.StringIndexError – Type.

```
StringIndexError(str, i)
```

An error occurred when trying to access `str` at index `i` that is not valid.

source

Core.InitError – Type.

```
InitError(mod::Symbol, error)
```

An error occurred when running a module's `__init__` function. The actual error thrown is available in the `.error` field.

source

`Base.retry` – Function.

```
retry(f;  delays=ExponentialBackOff(), check=nothing) -> Function
```

Return an anonymous function that calls function `f`. If an exception arises, `f` is repeatedly called again, each time check returns `true`, after waiting the number of seconds specified in `delays`. `check` should input `delays`'s current state and the `Exception`.

Julia 1.2

Before Julia 1.2 this signature was restricted to `f::Function`.

Examples

```
retry(f, delays=fill(5.0, 3))
retry(f, delays=rand(5:10, 2))
retry(f, delays=Base.ExponentialBackOff(n=3, first_delay=5, max_delay=1000))
retry(http_get, check=(s,e)->e.status == "503")(url)
retry(read, check=(s,e)->isa(e, IOError))(io, 128; all=false)
```

source

`Base.ExponentialBackOff` – Type.

```
ExponentialBackOff(; n=1, first_delay=0.05, max_delay=10.0, factor=5.0, jitter=0.1)
```

A Float64 iterator of length n whose elements exponentially increase at a rate in the interval `factor` * (1 ± `jitter`). The first element is `first_delay` and all elements are clamped to `max_delay`.

source

42.15 Events

`Base.Timer` – Method.

```
Timer(callback::Function, delay; interval = 0)
```

Create a timer that runs the function `callback` at each timer expiration.

Waiting tasks are woken and the function `callback` is called after an initial delay of `delay` seconds, and then repeating with the given `interval` in seconds. If `interval` is equal to `0`, the callback is only run once. The function `callback` is called with a single argument, the timer itself. Stop a timer by calling `close`. The `callback` may still be run one final time, if the timer has already expired.

Examples

Here the first number is printed after a delay of two seconds, then the following numbers are printed quickly.

```
julia> begin
           i = 0
           cb(timer) = (global i += 1; println(i))
           t = Timer(cb, 2, interval=0.2)
           wait(t)
           sleep(0.5)
           close(t)
       end
1
2
3
```

source

`Base.Timer` – Type.

```
Timer(delay; interval = 0)
```

Create a timer that wakes up tasks waiting for it (by calling `wait` on the timer object).

Waiting tasks are woken after an initial delay of at least `delay` seconds, and then repeating after at least `interval` seconds again elapse. If `interval` is equal to `0`, the timer is only triggered once. When the timer is closed (by `close`) waiting tasks are woken with an error. Use `isopen` to check whether a timer is still active.

Note

`interval` is subject to accumulating time skew. If you need precise events at a particular absolute time, create a new timer at each expiration with the difference to the next time computed.

Note

A `Timer` requires yield points to update its state. For instance, `isopen(t::Timer)` cannot be used to timeout a non-yielding while loop.

source

`Base.AsyncCondition` – Type.

```
AsyncCondition()
```

Create a async condition that wakes up tasks waiting for it (by calling wait on the object) when notified from C by a call to uv_async_send. Waiting tasks are woken with an error when the object is closed (by close). Use isopen to check whether it is still active.

This provides an implicit acquire & release memory ordering between the sending and waiting threads.

source

Base.AsyncCondition – Method.

```
AsyncCondition(callback::Function)
```

Create a async condition that calls the given callback function. The callback is passed one argument, the async condition object itself.

source

42.16 Reflection

Base.nameof – Method.

```
nameof(m::Module) -> Symbol
```

Get the name of a Module as a Symbol.

Examples

```
julia> nameof(Base.Broadcast)
:Broadcast
```

source

Base.parentmodule – Function.

```
parentmodule(m::Module) -> Module
```

Get a module's enclosing Module. Main is its own parent.

See also: names, nameof, fullname, @__MODULE__.

Examples

```
julia> parentmodule(Main)
Main

julia> parentmodule(Base.Broadcast)
Base
```

source

```
parentmodule(t::DataType) -> Module
```

Determine the module containing the definition of a (potentially `UnionAll`-wrapped) `DataType`.

Examples

```
julia> module Foo
           struct Int end
       end
Foo

julia> parentmodule(Int)
Core

julia> parentmodule(Foo.Int)
Foo
```

source

```
parentmodule(f::Function) -> Module
```

Determine the module containing the (first) definition of a generic function.

source

```
parentmodule(f::Function, types) -> Module
```

Determine the module containing the first method of a generic function `f` matching the specified `types`.

source

```
parentmodule(m::Method) -> Module
```

Return the module in which the given method `m` is defined.

> **Julia 1.9**
>
> Passing a `Method` as an argument requires Julia 1.9 or later.

source

`Base.pathof` – Method.

```
pathof(m::Module)
```

Return the path of the `m.jl` file that was used to `import` module `m`, or `nothing` if `m` was not imported from a package.

Use `dirname` to get the directory part and `basename` to get the file name part of the path.

source

`Base.pkgdir` – Method.

```
pkgdir(m::Module[, paths::String...])
```

Return the root directory of the package that declared module `m`, or `nothing` if `m` was not declared in a package. Optionally further path component strings can be provided to construct a path within the package root.

To get the root directory of the package that implements the current module the form `pkgdir(@__MODULE__)` can be used.

If an extension module is given, the root of the parent package is returned.

```
julia> pkgdir(Foo)
"/path/to/Foo.jl"

julia> pkgdir(Foo, "src", "file.jl")
"/path/to/Foo.jl/src/file.jl"
```

Julia 1.7

The optional argument `paths` requires at least Julia 1.7.

source

`Base.pkgversion` – Method.

```
pkgversion(m::Module)
```

Return the version of the package that imported module `m`, or `nothing` if `m` was not imported from a package, or imported from a package without a version field set.

The version is read from the package's Project.toml during package load.

To get the version of the package that imported the current module the form `pkgversion(@__MODULE__)` can be used.

Julia 1.9

This function was introduced in Julia 1.9.

source

`Base.moduleroot` – Function.

```
moduleroot(m::Module) -> Module
```

Find the root module of a given module. This is the first module in the chain of parent modules of `m` which is either a registered root module or which is its own parent module.

source

`__module__` – Keyword.

```
__module__
```

The argument `__module__` is only visible inside the macro, and it provides information (in the form of a `Module` object) about the expansion context of the macro invocation. See the manual section on Macro invocation for more information.

source

`__source__` – Keyword.

```
__source__
```

The argument `__source__` is only visible inside the macro, and it provides information (in the form of a `LineNumberNode` object) about the parser location of the `@` sign from the macro invocation. See the manual section on Macro invocation for more information.

source

`Base.@__MODULE__` – Macro.

```
@__MODULE__ -> Module
```

Get the `Module` of the toplevel eval, which is the `Module` code is currently being read from.

source

`Base.@__FILE__` – Macro.

```
@__FILE__ -> String
```

Expand to a string with the path to the file containing the macrocall, or an empty string if evaluated by `julia -e <expr>`. Return `nothing` if the macro was missing parser source information. Alternatively see `PROGRAM_FILE`.

source

`Base.@__DIR__` – Macro.

```
@__DIR__ -> String
```

Expand to a string with the absolute path to the directory of the file containing the macrocall. Return the current working directory if run from a REPL or if evaluated by `julia -e <expr>`.

source

`Base.@__LINE__` – Macro.

```
@__LINE__ -> Int
```

Expand to the line number of the location of the macrocall. Return `0` if the line number could not be determined.

source

`Base.fullname` – Function.

```
fullname(m::Module)
```

Get the fully-qualified name of a module as a tuple of symbols. For example,

Examples

```
julia> fullname(Base.Iterators)
(:Base, :Iterators)

julia> fullname(Main)
(:Main,)
```

source

`Base.names` – Function.

```
names(x::Module; all::Bool = false, imported::Bool = false)
```

Get an array of the names exported by a `Module`, excluding deprecated names. If `all` is true, then the list also includes non-exported names defined in the module, deprecated names, and compiler-generated names. If `imported` is true, then names explicitly imported from other modules are also included.

As a special case, all names defined in `Main` are considered "exported", since it is not idiomatic to explicitly export names from `Main`.

See also: `@locals`, `@__MODULE__`.

source

`Base.nameof` – Method.

```
nameof(f::Function) -> Symbol
```

Get the name of a generic `Function` as a symbol. For anonymous functions, this is a compiler-generated name. For explicitly-declared subtypes of `Function`, it is the name of the function's type.

source

`Base.functionloc` – Method.

```
functionloc(f::Function, types)
```

Return a tuple (filename,line) giving the location of a generic Function definition.

source

Base.functionloc – Method.

```
functionloc(m::Method)
```

Return a tuple (filename,line) giving the location of a Method definition.

source

Base.@locals – Macro.

```
@locals()
```

Construct a dictionary of the names (as symbols) and values of all local variables defined as of the call site.

Julia 1.1

This macro requires at least Julia 1.1.

Examples

```
julia> let x = 1, y = 2
           Base.@locals
       end
Dict{Symbol, Any} with 2 entries:
  :y => 2
  :x => 1

julia> function f(x)
           local y
           show(Base.@locals); println()
           for i = 1:1
               show(Base.@locals); println()
           end
           y = 2
           show(Base.@locals); println()
           nothing
       end;

julia> f(42)
Dict{Symbol, Any}(:x => 42)
Dict{Symbol, Any}(:i => 1, :x => 42)
Dict{Symbol, Any}(:y => 2, :x => 42)
```

source

42.17 Code loading

`Base.identify_package` – Function.

```
Base.identify_package(name::String)::Union{PkgId, Nothing}
Base.identify_package(where::Union{Module,PkgId}, name::String)::Union{PkgId, Nothing}
```

Identify the package by its name from the current environment stack, returning its `PkgId`, or `nothing` if it cannot be found.

If only the `name` argument is provided, it searches each environment in the stack and its named direct dependencies.

There `where` argument provides the context from where to search for the package: in this case it first checks if the name matches the context itself, otherwise it searches all recursive dependencies (from the resolved manifest of each environment) until it locates the context `where`, and from there identifies the dependency with the corresponding name.

```
julia> Base.identify_package("Pkg") # Pkg is a dependency of the default environment
Pkg [44cfe95a-1eb2-52ea-b672-e2afdf69b78f]

julia> using LinearAlgebra

julia> Base.identify_package(LinearAlgebra, "Pkg") # Pkg is not a dependency of LinearAlgebra
```

source

`Base.locate_package` – Function.

```
Base.locate_package(pkg::PkgId)::Union{String, Nothing}
```

The path to the entry-point file for the package corresponding to the identifier pkg, or `nothing` if not found. See also `identify_package`.

```
julia> pkg = Base.identify_package("Pkg")
Pkg [44cfe95a-1eb2-52ea-b672-e2afdf69b78f]

julia> Base.locate_package(pkg)
"/path/to/julia/stdlib/v1.10/Pkg/src/Pkg.jl"
```

source

`Base.require` – Function.

```
require(into::Module, module::Symbol)
```

This function is part of the implementation of `using` / `import`, if a module is not already defined in `Main`. It can also be called directly to force reloading a module, regardless of whether it has been loaded before (for example, when interactively developing libraries).

Loads a source file, in the context of the Main module, on every active node, searching standard locations for files. require is considered a top-level operation, so it sets the current include path but does not use it to search for files (see help for include). This function is typically used to load library code, and is implicitly called by using to load packages.

When searching for files, require first looks for package code in the global array LOAD_PATH. require is case-sensitive on all platforms, including those with case-insensitive filesystems like macOS and Windows.

For more details regarding code loading, see the manual sections on modules and parallel computing.

source

Base.compilecache – Function.

```
Base.compilecache(module::PkgId)
```

Creates a precompiled cache file for a module and all of its dependencies. This can be used to reduce package load times. Cache files are stored in DEPOT_PATH[1]/compiled. See Module initialization and precompilation for important notes.

source

Base.isprecompiled – Function.

```
Base.isprecompiled(pkg::PkgId; ignore_loaded::Bool=false)
```

Returns whether a given PkgId within the active project is precompiled.

By default this check observes the same approach that code loading takes with respect to when different versions of dependencies are currently loaded to that which is expected. To ignore loaded modules and answer as if in a fresh julia session specify ignore_loaded=true.

Julia 1.10

This function requires at least Julia 1.10.

source

Base.get_extension – Function.

```
get_extension(parent::Module, extension::Symbol)
```

Return the module for extension of parent or return nothing if the extension is not loaded.

source

42.18 Internals

Base.GC.gc – Function.

```
GC.gc([full=true])
```

Perform garbage collection. The argument `full` determines the kind of collection: A full collection (default) sweeps all objects, which makes the next GC scan much slower, while an incremental collection may only sweep so-called young objects.

Warning

Excessive use will likely lead to poor performance.

source

`Base.GC.enable` – Function.

```
GC.enable(on::Bool)
```

Control whether garbage collection is enabled using a boolean argument (`true` for enabled, `false` for disabled). Return previous GC state.

Warning

Disabling garbage collection should be used only with caution, as it can cause memory use to grow without bound.

source

`Base.GC.@preserve` – Macro.

```
GC.@preserve x1 x2 ... xn expr
```

Mark the objects `x1, x2, ...` as being *in use* during the evaluation of the expression `expr`. This is only required in unsafe code where `expr` *implicitly uses* memory or other resources owned by one of the xs.

Implicit use of x covers any indirect use of resources logically owned by x which the compiler cannot see. Some examples:

- Accessing memory of an object directly via a `Ptr`
- Passing a pointer to x to `ccall`
- Using resources of x which would be cleaned up in the finalizer.

`@preserve` should generally not have any performance impact in typical use cases where it briefly extends object lifetime. In implementation, `@preserve` has effects such as protecting dynamically allocated objects from garbage collection.

Examples

When loading from a pointer with `unsafe_load`, the underlying object is implicitly used, for example x is implicitly used by `unsafe_load(p)` in the following:

```
julia> let
           x = Ref{Int}(101)
           p = Base.unsafe_convert(Ptr{Int}, x)
           GC.@preserve x unsafe_load(p)
       end
101
```

When passing pointers to `ccall`, the pointed-to object is implicitly used and should be preserved. (Note however that you should normally just pass x directly to `ccall` which counts as an explicit use.)

```
julia> let
           x = "Hello"
           p = pointer(x)
           Int(GC.@preserve x @ccall strlen(p::Cstring)::Csize_t)
           # Preferred alternative
           Int(@ccall strlen(x::Cstring)::Csize_t)
       end
5
```

source

`Base.GC.safepoint` – Function.

```
GC.safepoint()
```

Inserts a point in the program where garbage collection may run. This can be useful in rare cases in multi-threaded programs where some threads are allocating memory (and hence may need to run GC) but other threads are doing only simple operations (no allocation, task switches, or I/O). Calling this function periodically in non-allocating threads allows garbage collection to run.

Julia 1.4

This function is available as of Julia 1.4.

source

`Base.GC.enable_logging` – Function.

```
GC.enable_logging(on::Bool)
```

When turned on, print statistics about each GC to stderr.

source

`Base.Meta.lower` – Function.

```
lower(m, x)
```

Takes the expression x and returns an equivalent expression in lowered form for executing in module m. See also code_lowered.

source

Base.Meta.@lower – Macro.

```
@lower [m] x
```

Return lowered form of the expression x in module m. By default m is the module in which the macro is called. See also lower.

source

Base.Meta.parse – Method.

```
parse(str, start; greedy=true, raise=true, depwarn=true, filename="none")
```

Parse the expression string and return an expression (which could later be passed to eval for execution). start is the code unit index into str of the first character to start parsing at (as with all string indexing, these are not character indices). If greedy is true (default), parse will try to consume as much input as it can; otherwise, it will stop as soon as it has parsed a valid expression. Incomplete but otherwise syntactically valid expressions will return Expr(:incomplete, "(error message)"). If raise is true (default), syntax errors other than incomplete expressions will raise an error. If raise is false, parse will return an expression that will raise an error upon evaluation. If depwarn is false, deprecation warnings will be suppressed. The filename argument is used to display diagnostics when an error is raised.

```
julia> Meta.parse("(α, β) = 3, 5", 1) # start of string
(:((α, β) = (3, 5)), 16)

julia> Meta.parse("(α, β) = 3, 5", 1, greedy=false)
(:((α, β)), 9)

julia> Meta.parse("(α, β) = 3, 5", 16) # end of string
(nothing, 16)

julia> Meta.parse("(α, β) = 3, 5", 11) # index of 3
(:((3, 5)), 16)

julia> Meta.parse("(α, β) = 3, 5", 11, greedy=false)
(3, 13)
```

source

Base.Meta.parse – Method.

```
parse(str; raise=true, depwarn=true, filename="none")
```

Parse the expression string greedily, returning a single expression. An error is thrown if there are additional characters after the first expression. If raise is true (default), syntax errors will raise an error; otherwise,

parse will return an expression that will raise an error upon evaluation. If depwarn is false, deprecation warnings will be suppressed. The filename argument is used to display diagnostics when an error is raised.

```
julia> Meta.parse("x = 3")
:(x = 3)

julia> Meta.parse("1.0.2")
ERROR: ParseError:
# Error @ none:1:1
1.0.2
└──┘ ── invalid numeric constant
[...]

julia> Meta.parse("1.0.2"; raise = false)
:($(Expr(:error, "invalid numeric constant "1.0."")))

julia> Meta.parse("x = ")
:($(Expr(:incomplete, "incomplete: premature end of input")))
```

source

Base.Meta.ParseError – Type.

```
ParseError(msg)
```

The expression passed to the parse function could not be interpreted as a valid Julia expression.

source

Core.QuoteNode – Type.

```
QuoteNode
```

A quoted piece of code, that does not support interpolation. See the manual section about QuoteNodes for details.

source

Base.macroexpand – Function.

```
macroexpand(m::Module, x; recursive=true)
```

Take the expression x and return an equivalent expression with all macros removed (expanded) for executing in module m. The recursive keyword controls whether deeper levels of nested macros are also expanded. This is demonstrated in the example below:

```
julia> module M
           macro m1()
               42
```

```
           end
           macro m2()
               :(@m1())
           end
       end
M

julia> macroexpand(M, :(@m2()), recursive=true)
42

julia> macroexpand(M, :(@m2()), recursive=false)
:(#= REPL[16]:6 =# M.@m1)
```

source

Base.@macroexpand – Macro.

```
@macroexpand
```

Return equivalent expression with all macros removed (expanded).

There are differences between @macroexpand and macroexpand.

- While macroexpand takes a keyword argument recursive, @macroexpand is always recursive. For a non recursive macro version, see @macroexpand1.
- While macroexpand has an explicit module argument, @macroexpand always expands with respect to the module in which it is called.

This is best seen in the following example:

```
julia> module M
           macro m()
               1
           end
           function f()
               (@macroexpand(@m),
                macroexpand(M, :(@m)),
                macroexpand(Main, :(@m))
               )
           end
       end
M

julia> macro m()
           2
       end
@m (macro with 1 method)

julia> M.f()
(1, 1, 2)
```

With `@macroexpand` the expression expands where `@macroexpand` appears in the code (module `M` in the example). With `macroexpand` the expression expands in the module given as the first argument.

source

`Base.@macroexpand1` – Macro.

```
@macroexpand1
```

Non recursive version of `@macroexpand`.

source

`Base.code_lowered` – Function.

```
code_lowered(f, types; generated=true, debuginfo=:default)
```

Return an array of the lowered forms (IR) for the methods matching the given generic function and type signature.

If `generated` is `false`, the returned `CodeInfo` instances will correspond to fallback implementations. An error is thrown if no fallback implementation exists. If `generated` is `true`, these `CodeInfo` instances will correspond to the method bodies yielded by expanding the generators.

The keyword `debuginfo` controls the amount of code metadata present in the output.

Note that an error will be thrown if `types` are not leaf types when `generated` is `true` and any of the corresponding methods are an `@generated` method.

source

`Base.code_typed` – Function.

```
code_typed(f, types; kw...)
```

Returns an array of type-inferred lowered form (IR) for the methods matching the given generic function and type signature.

Keyword Arguments

- `optimize::Bool = true`: optional, controls whether additional optimizations, such as inlining, are also applied.
- `debuginfo::Symbol = :default`: optional, controls the amount of code metadata present in the output, possible options are `:source` or `:none`.

Internal Keyword Arguments

This section should be considered internal, and is only for who understands Julia compiler internals.

- `world::UInt = Base.get_world_counter()`: optional, controls the world age to use when looking up methods, use current world age if not specified.

- `interp::Core.Compiler.AbstractInterpreter = Core.Compiler.NativeInterpreter(world)`: optional, controls the abstract interpreter to use, use the native interpreter if not specified.

Example

One can put the argument types in a tuple to get the corresponding code_typed.

```
julia> code_typed(+, (Float64, Float64))
1-element Vector{Any}:
 CodeInfo(
1 ─ %1 = Base.add_float(x, y)::Float64
└──      return %1
) => Float64
```

source

`Base.precompile` – Function.

```
precompile(f, argtypes::Tuple{Vararg{Any}})
```

Compile the given function `f` for the argument tuple (of types) `argtypes`, but do not execute it.

source

```
precompile(f, argtypes::Tuple{Vararg{Any}}, m::Method)
```

Precompile a specific method for the given argument types. This may be used to precompile a different method than the one that would ordinarily be chosen by dispatch, thus mimicking `invoke`.

source

`Base.jit_total_bytes` – Function.

```
Base.jit_total_bytes()
```

Return the total amount (in bytes) allocated by the just-in-time compiler for e.g. native code and data.

source

42.19 Meta

`Base.Meta.quot` – Function.

```
Meta.quot(ex)::Expr
```

Quote expression ex to produce an expression with head `quote`. This can for instance be used to represent objects of type `Expr` in the AST. See also the manual section about QuoteNode.

Examples

```
julia> eval(Meta.quot(:x))
:x

julia> dump(Meta.quot(:x))
Expr
  head: Symbol quote
  args: Array{Any}((1,))
    1: Symbol x

julia> eval(Meta.quot(:(1+2)))
:(1 + 2)
```

source

Base.isexpr – Function.

```
Meta.isexpr(ex, head[, n])::Bool
```

Return `true` if ex is an `Expr` with the given type head and optionally that the argument list is of length n. head may be a `Symbol` or collection of `Symbols`. For example, to check that a macro was passed a function call expression, you might use `isexpr(ex, :call)`.

Examples

```
julia> ex = :(f(x))
:(f(x))

julia> Meta.isexpr(ex, :block)
false

julia> Meta.isexpr(ex, :call)
true

julia> Meta.isexpr(ex, [:block, :call]) # multiple possible heads
true

julia> Meta.isexpr(ex, :call, 1)
false

julia> Meta.isexpr(ex, :call, 2)
true
```

source

Base.isidentifier – Function.

```
isidentifier(s) -> Bool
```

Return whether the symbol or string `s` contains characters that are parsed as a valid ordinary identifier (not a binary/unary operator) in Julia code; see also `Base.isoperator`.

Internally Julia allows any sequence of characters in a Symbol (except \0s), and macros automatically use variable names containing # in order to avoid naming collision with the surrounding code. In order for the parser to recognize a variable, it uses a limited set of characters (greatly extended by Unicode). isidentifier() makes it possible to query the parser directly whether a symbol contains valid characters.

Examples

```
julia> Meta.isidentifier(:x), Meta.isidentifier("1x")
(true, false)
```

source

Base.isoperator – Function.

```
isoperator(s::Symbol)
```

Return true if the symbol can be used as an operator, false otherwise.

Examples

```
julia> Meta.isoperator(:+), Meta.isoperator(:f)
(true, false)
```

source

Base.isunaryoperator – Function.

```
isunaryoperator(s::Symbol)
```

Return true if the symbol can be used as a unary (prefix) operator, false otherwise.

Examples

```
julia> Meta.isunaryoperator(:-), Meta.isunaryoperator(:√), Meta.isunaryoperator(:f)
(true, true, false)
```

source

Base.isbinaryoperator – Function.

```
isbinaryoperator(s::Symbol)
```

Return true if the symbol can be used as a binary (infix) operator, false otherwise.

Examples

```
julia> Meta.isbinaryoperator(:-), Meta.isbinaryoperator(:√), Meta.isbinaryoperator(:f)
(true, false, false)
```

source

Base.Meta.show_sexpr – Function.

```
Meta.show_sexpr([io::IO,], ex)
```

Show expression ex as a lisp style S-expression.

Examples

```
julia> Meta.show_sexpr(:(f(x, g(y,z))))
(:call, :f, :x, (:call, :g, :y, :z))
```

source

Chapter 43

集合和数据结构

43.1 迭代

序列迭代由 `iterate` 实现广义的 `for` 循环

```
for i in iter   # or  "for i = iter"
    # body
end
```

被转换成

```
next = iterate(iter)
while next !== nothing
    (i, state) = next
    # body
    next = iterate(iter, state)
end
```

`state` 对象可以是任何对象，并且对于每个可迭代类型应该选择合适的 `state` 对象。请参照 帮助文档接口的迭代小节 来获取关于定义一个常见迭代类型的更多细节。

`Base.iterate` – Function.

```
iterate(iter [, state]) -> Union{Nothing, Tuple{Any, Any}}
```

Advance the iterator to obtain the next element. If no elements remain, `nothing` should be returned. Otherwise, a 2-tuple of the next element and the new iteration state should be returned.

source

`Base.IteratorSize` – Type.

```
IteratorSize(itertype::Type) -> IteratorSize
```

Given the type of an iterator, return one of the following values:

- `SizeUnknown()` if the length (number of elements) cannot be determined in advance.
- `HasLength()` if there is a fixed, finite length.
- `HasShape{N}()` if there is a known length plus a notion of multidimensional shape (as for an array). In this case `N` should give the number of dimensions, and the `axes` function is valid for the iterator.
- `IsInfinite()` if the iterator yields values forever.

The default value (for iterators that do not define this function) is `HasLength()`. This means that most iterators are assumed to implement `length`.

This trait is generally used to select between algorithms that pre-allocate space for their result, and algorithms that resize their result incrementally.

```
julia> Base.IteratorSize(1:5)
Base.HasShape{1}()

julia> Base.IteratorSize((2,3))
Base.HasLength()
```

source

`Base.IteratorEltype` – Type.

```
IteratorEltype(itertype::Type) -> IteratorEltype
```

Given the type of an iterator, return one of the following values:

- `EltypeUnknown()` if the type of elements yielded by the iterator is not known in advance.
- `HasEltype()` if the element type is known, and `eltype` would return a meaningful value.

`HasEltype()` is the default, since iterators are assumed to implement `eltype`.

This trait is generally used to select between algorithms that pre-allocate a specific type of result, and algorithms that pick a result type based on the types of yielded values.

```
julia> Base.IteratorEltype(1:5)
Base.HasEltype()
```

source

以下类型均完全实现了上述函数：

- `AbstractRange`
- `UnitRange`
- `Tuple`
- `Number`
- `AbstractArray`

- BitSet
- IdDict
- Dict
- WeakKeyDict
- EachLine
- AbstractString
- Set
- Pair
- NamedTuple

43.2 构造函数和类型

`Base.AbstractRange` – Type.

```
AbstractRange{T}
```

Supertype for ranges with elements of type T. `UnitRange` and other types are subtypes of this.

source

`Base.OrdinalRange` – Type.

```
OrdinalRange{T, S} <: AbstractRange{T}
```

Supertype for ordinal ranges with elements of type T with spacing(s) of type S. The steps should be always-exact multiples of `oneunit`, and T should be a "discrete" type, which cannot have values smaller than `oneunit`. For example, `Integer` or `Date` types would qualify, whereas `Float64` would not (since this type can represent values smaller than `oneunit(Float64)`. `UnitRange`, `StepRange`, and other types are subtypes of this.

source

`Base.AbstractUnitRange` – Type.

```
AbstractUnitRange{T} <: OrdinalRange{T, T}
```

Supertype for ranges with a step size of `oneunit(T)` with elements of type T. `UnitRange` and other types are subtypes of this.

source

`Base.StepRange` – Type.

```
StepRange{T, S} <: OrdinalRange{T, S}
```

Ranges with elements of type T with spacing of type S. The step between each element is constant, and the range is defined in terms of a `start` and `stop` of type T and a `step` of type S. Neither T nor S should be floating point types. The syntax `a:b:c` with `b != 0` and a, b, and c all integers creates a `StepRange`.

Examples

```
julia> collect(StepRange(1, Int8(2), 10))
5-element Vector{Int64}:
 1
 3
 5
 7
 9

julia> typeof(StepRange(1, Int8(2), 10))
StepRange{Int64, Int8}

julia> typeof(1:3:6)
StepRange{Int64, Int64}
```

source

`Base.UnitRange` – Type.

```
UnitRange{T<:Real}
```

A range parameterized by a `start` and `stop` of type T, filled with elements spaced by 1 from `start` until `stop` is exceeded. The syntax `a:b` with a and b both `Integer`s creates a `UnitRange`.

Examples

```
julia> collect(UnitRange(2.3, 5.2))
3-element Vector{Float64}:
 2.3
 3.3
 4.3

julia> typeof(1:10)
UnitRange{Int64}
```

source

`Base.LinRange` – Type.

```
LinRange{T,L}
```

A range with `len` linearly spaced elements between its `start` and `stop`. The size of the spacing is controlled by `len`, which must be an `Integer`.

Examples

```
julia> LinRange(1.5, 5.5, 9)
9-element LinRange{Float64, Int64}:
 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5
```

Compared to using `range`, directly constructing a `LinRange` should have less overhead but won't try to correct for floating point errors:

```
julia> collect(range(-0.1, 0.3, length=5))
5-element Vector{Float64}:
 -0.1
  0.0
  0.1
  0.2
  0.3

julia> collect(LinRange(-0.1, 0.3, 5))
5-element Vector{Float64}:
 -0.1
 -1.3877787807814457e-17
  0.09999999999999999
  0.19999999999999998
  0.3
```

source

43.3 通用集合

`Base.isempty` – Function.

```
isempty(collection) -> Bool
```

Determine whether a collection is empty (has no elements).

> **Warning**
>
> `isempty(itr)` may consume the next element of a stateful iterator `itr` unless an appropriate `Base.isdone(itr)` or `isempty` method is defined. Use of `isempty` should therefore be avoided when writing generic code which should support any iterator type.

Examples

```
julia> isempty([])
true

julia> isempty([1 2 3])
false
```

source

```
isempty(condition)
```

Return `true` if no tasks are waiting on the condition, `false` otherwise.

source

`Base.empty!` – Function.

```
empty!(collection) -> collection
```

Remove all elements from a `collection`.

Examples

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String, Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> empty!(A);

julia> A
Dict{String, Int64}()
```

source

`Base.length` – Function.

```
length(collection) -> Integer
```

Return the number of elements in the collection.

Use `lastindex` to get the last valid index of an indexable collection.

See also: `size`, `ndims`, `eachindex`.

Examples

```
julia> length(1:5)
5

julia> length([1, 2, 3, 4])
4

julia> length([1 2; 3 4])
4
```

source

`Base.checked_length` – Function.

```
Base.checked_length(r)
```

Calculates `length(r)`, but may check for overflow errors where applicable when the result doesn't fit into `Union{Integer(eltype(r)),Int}`.

source

以下类型均完全实现了上述函数:

- `AbstractRange`
- `UnitRange`
- `Tuple`
- `Number`
- `AbstractArray`
- `BitSet`
- `IdDict`
- `Dict`
- `WeakKeyDict`
- `AbstractString`
- `Set`
- `NamedTuple`

43.4 可迭代集合

`Base.in` – Function.

```
in(item, collection) -> Bool
∈(item, collection) -> Bool
```

Determine whether an item is in the given collection, in the sense that it is `==` to one of the values generated by iterating over the collection. Return a `Bool` value, except if `item` is `missing` or `collection` contains `missing` but not `item`, in which case `missing` is returned (three-valued logic, matching the behavior of `any` and `==`).

Some collections follow a slightly different definition. For example, `Set`s check whether the item `isequal` to one of the elements; `Dict`s look for `key=>value` pairs, and the key is compared using `isequal`.

To test for the presence of a key in a dictionary, use `haskey` or `k in keys(dict)`. For the collections mentioned above, the result is always a `Bool`.

When broadcasting with `in.(items, collection)` or `items .∈ collection`, both `item` and `collection` are broadcasted over, which is often not what is intended. For example, if both arguments are vectors

(and the dimensions match), the result is a vector indicating whether each value in collection `items` is `in` the value at the corresponding position in `collection`. To get a vector indicating whether each value in `items` is in `collection`, wrap `collection` in a tuple or a `Ref` like this: `in.(items, Ref(collection))` or `items .∈ Ref(collection)`.

See also: `∉`, `insorted`, `contains`, `occursin`, `issubset`.

Examples

```
julia> a = 1:3:20
1:3:19

julia> 4 in a
true

julia> 5 in a
false

julia> missing in [1, 2]
missing

julia> 1 in [2, missing]
missing

julia> 1 in [1, missing]
true

julia> missing in Set([1, 2])
false

julia> (1=>missing) in Dict(1=>10, 2=>20)
missing

julia> [1, 2] .∈ [2, 3]
2-element BitVector:
 0
 0

julia> [1, 2] .∈ ([2, 3],)
2-element BitVector:
 0
 1
```

source

Base.:∉ – Function.

```
∉(item, collection) -> Bool
∌(collection, item) -> Bool
```

Negation of ∈ and ∋, i.e. checks that `item` is not in `collection`.

When broadcasting with `items .∉ collection`, both `item` and `collection` are broadcasted over, which is often not what is intended. For example, if both arguments are vectors (and the dimensions match), the

result is a vector indicating whether each value in collection `items` is not in the value at the corresponding position in `collection`. To get a vector indicating whether each value in `items` is not in `collection`, wrap `collection` in a tuple or a `Ref` like this: `items .∉ Ref(collection)`.

Examples

```
julia> 1 ∉ 2:4
true

julia> 1 ∉ 1:3
false

julia> [1, 2] .∉ [2, 3]
2-element BitVector:
 1
 1

julia> [1, 2] .∉ ([2, 3],)
2-element BitVector:
 1
 0
```

source

`Base.eltype` – Function.

```
eltype(type)
```

Determine the type of the elements generated by iterating a collection of the given `type`. For dictionary types, this will be a `Pair{KeyType,ValType}`. The definition `eltype(x) = eltype(typeof(x))` is provided for convenience so that instances can be passed instead of types. However the form that accepts a type argument should be defined for new types.

See also: `keytype`, `typeof`.

Examples

```
julia> eltype(fill(1f0, (2,2)))
Float32

julia> eltype(fill(0x1, (2,2)))
UInt8
```

source

`Base.indexin` – Function.

```
indexin(a, b)
```

Return an array containing the first index in `b` for each value in `a` that is a member of `b`. The output array contains `nothing` wherever `a` is not a member of `b`.

See also: sortperm, findfirst.

Examples

```
julia> a = ['a', 'b', 'c', 'b', 'd', 'a'];

julia> b = ['a', 'b', 'c'];

julia> indexin(a, b)
6-element Vector{Union{Nothing, Int64}}:
 1
 2
 3
 2
  nothing
 1

julia> indexin(b, a)
3-element Vector{Union{Nothing, Int64}}:
 1
 2
 3
```

source

Base.unique – Function.

```
unique(itr)
```

Return an array containing only the unique elements of collection itr, as determined by isequal, in the order that the first of each set of equivalent elements originally appears. The element type of the input is preserved.

See also: unique!, allunique, allequal.

Examples

```
julia> unique([1, 2, 6, 2])
3-element Vector{Int64}:
 1
 2
 6

julia> unique(Real[1, 1.0, 2])
2-element Vector{Real}:
 1
 2
```

source

```
unique(f, itr)
```

Return an array containing one value from `itr` for each unique value produced by `f` applied to elements of `itr`.

Examples

```
julia> unique(x -> x^2, [1, -1, 3, -3, 4])
3-element Vector{Int64}:
 1
 3
 4
```

This functionality can also be used to extract the *indices* of the first occurrences of unique elements in an array:

```
julia> a = [3.1, 4.2, 5.3, 3.1, 3.1, 3.1, 4.2, 1.7];

julia> i = unique(i -> a[i], eachindex(a))
4-element Vector{Int64}:
 1
 2
 3
 8

julia> a[i]
4-element Vector{Float64}:
 3.1
 4.2
 5.3
 1.7

julia> a[i] == unique(a)
true
```

source

```
unique(A::AbstractArray; dims::Int)
```

Return unique regions of `A` along dimension `dims`.

Examples

```
julia> A = map(isodd, reshape(Vector(1:8), (2,2,2)))
2×2×2 Array{Bool, 3}:
[:, :, 1] =
 1  1
 0  0

[:, :, 2] =
 1  1
 0  0

julia> unique(A)
2-element Vector{Bool}:
```

```
 1
 0

julia> unique(A, dims=2)
2×1×2 Array{Bool, 3}:
[:, :, 1] =
 1
 0

[:, :, 2] =
 1
 0

julia> unique(A, dims=3)
2×2×1 Array{Bool, 3}:
[:, :, 1] =
 1  1
 0  0
```

source

Base.unique! – Function.

```
unique!(f, A::AbstractVector)
```

Selects one value from A for each unique value produced by f applied to elements of A, then return the modified A.

Julia 1.1

This method is available as of Julia 1.1.

Examples

```
julia> unique!(x -> x^2, [1, -1, 3, -3, 4])
3-element Vector{Int64}:
 1
 3
 4

julia> unique!(n -> n%3, [5, 1, 8, 9, 3, 4, 10, 7, 2, 6])
3-element Vector{Int64}:
 5
 1
 9

julia> unique!(iseven, [2, 3, 5, 7, 9])
2-element Vector{Int64}:
 2
 3
```

source

```
unique!(A::AbstractVector)
```

Remove duplicate items as determined by `isequal`, then return the modified `A`. `unique!` will return the elements of `A` in the order that they occur. If you do not care about the order of the returned data, then calling `(sort!(A); unique!(A))` will be much more efficient as long as the elements of `A` can be sorted.

Examples

```
julia> unique!([1, 1, 1])
1-element Vector{Int64}:
 1

julia> A = [7, 3, 2, 3, 7, 5];

julia> unique!(A)
4-element Vector{Int64}:
 7
 3
 2
 5

julia> B = [7, 6, 42, 6, 7, 42];

julia> sort!(B);  # unique! is able to process sorted data much more efficiently.

julia> unique!(B)
3-element Vector{Int64}:
  6
  7
 42
```

source

`Base.allunique` – Function.

```
allunique(itr) -> Bool
```

Return `true` if all values from `itr` are distinct when compared with `isequal`.

See also: `unique`, `issorted`, `allequal`.

Examples

```
julia> allunique([1, 2, 3])
true

julia> allunique([1, 2, 1, 2])
false

julia> allunique(Real[1, 1.0, 2])
false
```

```
julia> allunique([NaN, 2.0, NaN, 4.0])
false
```

source

Base.allequal – Function.

```
allequal(itr) -> Bool
```

Return true if all values from itr are equal when compared with isequal.

See also: unique, allunique.

> **Julia 1.8**
>
> The allequal function requires at least Julia 1.8.

Examples

```
julia> allequal([])
true

julia> allequal([1])
true

julia> allequal([1, 1])
true

julia> allequal([1, 2])
false

julia> allequal(Dict(:a => 1, :b => 1))
false
```

source

Base.reduce – Method.

```
reduce(op, itr; [init])
```

Reduce the given collection itr with the given binary operator op. If provided, the initial value init must be a neutral element for op that will be returned for empty collections. It is unspecified whether init is used for non-empty collections.

For empty collections, providing init will be necessary, except for some special cases (e.g. when op is one of +, *, max, min, &, |) when Julia can determine the neutral element of op.

Reductions for certain commonly-used operators may have special implementations, and should be used instead: maximum(itr), minimum(itr), sum(itr), prod(itr), any(itr), all(itr). There are efficient methods for concatenating certain arrays of arrays by calling reduce(vcat, arr) or reduce(hcat, arr).

The associativity of the reduction is implementation dependent. This means that you can't use non-associative operations like - because it is undefined whether reduce(-,[1,2,3]) should be evaluated as (1-2)-3 or 1-(2-3). Use foldl or foldr instead for guaranteed left or right associativity.

Some operations accumulate error. Parallelism will be easier if the reduction can be executed in groups. Future versions of Julia might change the algorithm. Note that the elements are not reordered if you use an ordered collection.

Examples

```
julia> reduce(*, [2; 3; 4])
24

julia> reduce(*, [2; 3; 4]; init=-1)
-24
```

source

Base.reduce – Method.

```
reduce(f, A::AbstractArray; dims=:, [init])
```

Reduce 2-argument function f along dimensions of A. dims is a vector specifying the dimensions to reduce, and the keyword argument init is the initial value to use in the reductions. For +, *, max and min the init argument is optional.

The associativity of the reduction is implementation-dependent; if you need a particular associativity, e.g. left-to-right, you should write your own loop or consider using foldl or foldr. See documentation for reduce.

Examples

```
julia> a = reshape(Vector(1:16), (4,4))
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> reduce(max, a, dims=2)
4×1 Matrix{Int64}:
 13
 14
 15
 16

julia> reduce(max, a, dims=1)
1×4 Matrix{Int64}:
 4  8  12  16
```

source

Base.foldl – Method.

```
foldl(op, itr; [init])
```

Like `reduce`, but with guaranteed left associativity. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

See also `mapfoldl`, `foldr`, `accumulate`.

Examples

```
julia> foldl(=>, 1:4)
((1 => 2) => 3) => 4

julia> foldl(=>, 1:4; init=0)
(((0 => 1) => 2) => 3) => 4

julia> accumulate(=>, (1,2,3,4))
(1, 1 => 2, (1 => 2) => 3, ((1 => 2) => 3) => 4)
```

source

`Base.foldr` – Method.

```
foldr(op, itr; [init])
```

Like `reduce`, but with guaranteed right associativity. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

Examples

```
julia> foldr(=>, 1:4)
1 => (2 => (3 => 4))

julia> foldr(=>, 1:4; init=0)
1 => (2 => (3 => (4 => 0)))
```

source

`Base.maximum` – Function.

```
maximum(f, itr; [init])
```

Return the largest result of calling function `f` on each element of `itr`.

The value returned for empty `itr` can be specified by `init`. It must be a neutral element for `max` (i.e. which is less than or equal to any other element) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> maximum(length, ["Julion", "Julia", "Jule"])
6

julia> maximum(length, []; init=-1)
-1

julia> maximum(sin, Real[]; init=-1.0)  # good, since output of sin is >= -1
-1.0
```

source

```
maximum(itr; [init])
```

Return the largest element in a collection.

The value returned for empty `itr` can be specified by `init`. It must be a neutral element for `max` (i.e. which is less than or equal to any other element) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> maximum(-20.5:10)
9.5

julia> maximum([1,2,3])
3

julia> maximum(())
ERROR: MethodError: reducing over an empty collection is not allowed; consider supplying `init`
↪  to the reducer
Stacktrace:
[...]

julia> maximum((); init=-Inf)
-Inf
```

source

```
maximum(A::AbstractArray; dims)
```

Compute the maximum value of an array over the given dimensions. See also the `max(a,b)` function to take the maximum of two or more arguments, which can be applied elementwise to arrays via `max.(a,b)`.

See also: `maximum!`, `extrema`, `findmax`, `argmax`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> maximum(A, dims=1)
1×2 Matrix{Int64}:
 3  4

julia> maximum(A, dims=2)
2×1 Matrix{Int64}:
 2
 4
```

source

```
maximum(f, A::AbstractArray; dims)
```

Compute the maximum value by calling the function `f` on each element of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> maximum(abs2, A, dims=1)
1×2 Matrix{Int64}:
 9  16

julia> maximum(abs2, A, dims=2)
2×1 Matrix{Int64}:
  4
 16
```

source

`Base.maximum!` – Function.

```
maximum!(r, A)
```

Compute the maximum value of `A` over the singleton dimensions of `r`, and write results to `r`.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> maximum!([1; 1], A)
2-element Vector{Int64}:
 2
 4

julia> maximum!([1 1], A)
1×2 Matrix{Int64}:
 3  4
```

source

Base.minimum – Function.

```
minimum(f, itr; [init])
```

Return the smallest result of calling function f on each element of itr.

The value returned for empty itr can be specified by init. It must be a neutral element for min (i.e. which is greater than or equal to any other element) as it is unspecified whether init is used for non-empty collections.

Julia 1.6

Keyword argument init requires Julia 1.6 or later.

Examples

```
julia> minimum(length, ["Julion", "Julia", "Jule"])
4

julia> minimum(length, []; init=typemax(Int64))
9223372036854775807

julia> minimum(sin, Real[]; init=1.0)  # good, since output of sin is <= 1
1.0
```

source

```
minimum(itr; [init])
```

Return the smallest element in a collection.

The value returned for empty itr can be specified by init. It must be a neutral element for min (i.e. which is greater than or equal to any other element) as it is unspecified whether init is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> minimum(-20.5:10)
-20.5

julia> minimum([1,2,3])
1

julia> minimum([])
ERROR: MethodError: reducing over an empty collection is not allowed; consider supplying `init`
↪  to the reducer
Stacktrace:
[...]

julia> minimum([]; init=Inf)
Inf
```

source

```
minimum(A::AbstractArray; dims)
```

Compute the minimum value of an array over the given dimensions. See also the `min(a,b)` function to take the minimum of two or more arguments, which can be applied elementwise to arrays via `min.(a,b)`.

See also: `minimum!`, `extrema`, `findmin`, `argmin`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> minimum(A, dims=1)
1×2 Matrix{Int64}:
 1  2

julia> minimum(A, dims=2)
2×1 Matrix{Int64}:
 1
 3
```

source

```
minimum(f, A::AbstractArray; dims)
```

Compute the minimum value by calling the function `f` on each element of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> minimum(abs2, A, dims=1)
1×2 Matrix{Int64}:
 1  4

julia> minimum(abs2, A, dims=2)
2×1 Matrix{Int64}:
 1
 9
```

source

Base.minimum! – Function.

```
minimum!(r, A)
```

Compute the minimum value of A over the singleton dimensions of r, and write results to r.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> minimum!([1; 1], A)
2-element Vector{Int64}:
 1
 3

julia> minimum!([1 1], A)
1×2 Matrix{Int64}:
 1  2
```

source

Base.extrema – Function.

```
extrema(itr; [init]) -> (mn, mx)
```

Compute both the minimum `mn` and maximum `mx` element in a single pass, and return them as a 2-tuple.

The value returned for empty `itr` can be specified by `init`. It must be a 2-tuple whose first and second elements are neutral elements for `min` and `max` respectively (i.e. which are greater/less than or equal to any other element). As a consequence, when `itr` is empty the returned `(mn, mx)` tuple will satisfy `mn ≥ mx`. When `init` is specified it may be used even for non-empty `itr`.

Julia 1.8

Keyword argument `init` requires Julia 1.8 or later.

Examples

```
julia> extrema(2:10)
(2, 10)

julia> extrema([9,pi,4.5])
(3.141592653589793, 9.0)

julia> extrema([]; init = (Inf, -Inf))
(Inf, -Inf)
```

source

```
extrema(f, itr; [init]) -> (mn, mx)
```

Compute both the minimum `mn` and maximum `mx` of `f` applied to each element in `itr` and return them as a 2-tuple. Only one pass is made over `itr`.

The value returned for empty `itr` can be specified by `init`. It must be a 2-tuple whose first and second elements are neutral elements for `min` and `max` respectively (i.e. which are greater/less than or equal to any other element). It is used for non-empty collections. Note: it implies that, for empty `itr`, the returned value `(mn, mx)` satisfies `mn ≥ mx` even though for non-empty `itr` it satisfies `mn ≤ mx`. This is a "paradoxical" but yet expected result.

Julia 1.2

This method requires Julia 1.2 or later.

Julia 1.8

Keyword argument `init` requires Julia 1.8 or later.

Examples

```
julia> extrema(sin, 0:π)
(0.0, 0.9092974268256817)

julia> extrema(sin, Real[]; init = (1.0, -1.0))  # good, since -1 ≤ sin(::Real) ≤ 1
(1.0, -1.0)
```

source

```
extrema(A::AbstractArray; dims) -> Array{Tuple}
```

Compute the minimum and maximum elements of an array over the given dimensions.

See also: `minimum`, `maximum`, `extrema!`.

Examples

```
julia> A = reshape(Vector(1:2:16), (2,2,2))
2×2×2 Array{Int64, 3}:
[:, :, 1] =
 1  5
 3  7

[:, :, 2] =
  9  13
 11  15

julia> extrema(A, dims = (1,2))
1×1×2 Array{Tuple{Int64, Int64}, 3}:
[:, :, 1] =
 (1, 7)

[:, :, 2] =
 (9, 15)
```

source

```
extrema(f, A::AbstractArray; dims) -> Array{Tuple}
```

Compute the minimum and maximum of `f` applied to each element in the given dimensions of `A`.

> **Julia 1.2**
>
> This method requires Julia 1.2 or later.

source

`Base.extrema!` – Function.

```
extrema!(r, A)
```

Compute the minimum and maximum value of A over the singleton dimensions of `r`, and write results to `r`.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

> **Julia 1.8**
>
> This method requires Julia 1.8 or later.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> extrema!([(1, 1); (1, 1)], A)
2-element Vector{Tuple{Int64, Int64}}:
 (1, 2)
 (3, 4)

julia> extrema!([(1, 1);; (1, 1)], A)
1×2 Matrix{Tuple{Int64, Int64}}:
 (1, 3)  (2, 4)
```

source

`Base.argmax` – Function.

```
argmax(r::AbstractRange)
```

Ranges can have multiple maximal elements. In that case `argmax` will return a maximal index, but not necessarily the first one.

source

```
argmax(f, domain)
```

Return a value `x` from `domain` for which `f(x)` is maximised. If there are multiple maximal values for `f(x)` then the first one will be found.

`domain` must be a non-empty iterable.

Values are compared with `isless`.

> **Julia 1.7**
>
> This method requires Julia 1.7 or later.

See also argmin, findmax.

Examples

```
julia> argmax(abs, -10:5)
-10

julia> argmax(cos, 0:π/2:2π)
0.0
```

source

```
argmax(itr)
```

Return the index or key of the maximal element in a collection. If there are multiple maximal elements, then the first one will be returned.

The collection must not be empty.

Values are compared with `isless`.

See also: argmin, findmax.

Examples

```
julia> argmax([8, 0.1, -9, pi])
1

julia> argmax([1, 7, 7, 6])
2

julia> argmax([1, 7, 7, NaN])
4
```

source

```
argmax(A; dims) -> indices
```

For an array input, return the indices of the maximum elements over the given dimensions. NaN is treated as greater than all other values except `missing`.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> argmax(A, dims=1)
1×2 Matrix{CartesianIndex{2}}:
 CartesianIndex(2, 1)  CartesianIndex(2, 2)

julia> argmax(A, dims=2)
```

```
2×1 Matrix{CartesianIndex{2}}:
 CartesianIndex(1, 2)
 CartesianIndex(2, 2)
```

source

`Base.argmin` – Function.

```
argmin(r::AbstractRange)
```

Ranges can have multiple minimal elements. In that case `argmin` will return a minimal index, but not necessarily the first one.

source

```
argmin(f, domain)
```

Return a value `x` from `domain` for which `f(x)` is minimised. If there are multiple minimal values for `f(x)` then the first one will be found.

`domain` must be a non-empty iterable.

`NaN` is treated as less than all other values except `missing`.

Julia 1.7

This method requires Julia 1.7 or later.

See also `argmax`, `findmin`.

Examples

```
julia> argmin(sign, -10:5)
-10

julia> argmin(x -> -x^3 + x^2 - 10, -5:5)
5

julia> argmin(acos, 0:0.1:1)
1.0
```

source

```
argmin(itr)
```

Return the index or key of the minimal element in a collection. If there are multiple minimal elements, then the first one will be returned.

The collection must not be empty.

NaN is treated as less than all other values except `missing`.

See also: `argmax`, `findmin`.

Examples

```
julia> argmin([8, 0.1, -9, pi])
3

julia> argmin([7, 1, 1, 6])
2

julia> argmin([7, 1, 1, NaN])
4
```

source

```
argmin(A; dims) -> indices
```

For an array input, return the indices of the minimum elements over the given dimensions. NaN is treated as less than all other values except `missing`.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> argmin(A, dims=1)
1×2 Matrix{CartesianIndex{2}}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)

julia> argmin(A, dims=2)
2×1 Matrix{CartesianIndex{2}}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)
```

source

`Base.findmax` – Function.

```
findmax(f, domain) -> (f(x), index)
```

Return a pair of a value in the codomain (outputs of `f`) and the index of the corresponding value in the domain (inputs to `f`) such that `f(x)` is maximised. If there are multiple maximal points, then the first one will be returned.

`domain` must be a non-empty iterable.

Values are compared with `isless`.

Julia 1.7

This method requires Julia 1.7 or later.

Examples

```
julia> findmax(identity, 5:9)
(9, 5)

julia> findmax(-, 1:10)
(-1, 1)

julia> findmax(first, [(1, :a), (3, :b), (3, :c)])
(3, 2)

julia> findmax(cos, 0:π/2:2π)
(1.0, 1)
```

source

```
findmax(itr) -> (x, index)
```

Return the maximal element of the collection `itr` and its index or key. If there are multiple maximal elements, then the first one will be returned. Values are compared with `isless`.

See also: `findmin`, `argmax`, `maximum`.

Examples

```
julia> findmax([8, 0.1, -9, pi])
(8.0, 1)

julia> findmax([1, 7, 7, 6])
(7, 2)

julia> findmax([1, 7, 7, NaN])
(NaN, 4)
```

source

```
findmax(A; dims) -> (maxval, index)
```

For an array input, returns the value and index of the maximum over the given dimensions. `NaN` is treated as greater than all other values except `missing`.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0
```

```
julia> findmax(A, dims=1)
([3.0 4.0], CartesianIndex{2}[CartesianIndex(2, 1) CartesianIndex(2, 2)])

julia> findmax(A, dims=2)
([2.0; 4.0;;], CartesianIndex{2}[CartesianIndex(1, 2); CartesianIndex(2, 2);;])
```

source

```
findmax(f, A; dims) -> (f(x), index)
```

For an array input, returns the value in the codomain and index of the corresponding value which maximize `f` over the given dimensions.

Examples

```
julia> A = [-1.0 1; -0.5 2]
2×2 Matrix{Float64}:
 -1.0  1.0
 -0.5  2.0

julia> findmax(abs2, A, dims=1)
([1.0 4.0], CartesianIndex{2}[CartesianIndex(1, 1) CartesianIndex(2, 2)])

julia> findmax(abs2, A, dims=2)
([1.0; 4.0;;], CartesianIndex{2}[CartesianIndex(1, 1); CartesianIndex(2, 2);;])
```

source

`Base.findmin` – Function.

```
findmin(f, domain) -> (f(x), index)
```

Return a pair of a value in the codomain (outputs of `f`) and the index of the corresponding value in the `domain` (inputs to `f`) such that `f(x)` is minimised. If there are multiple minimal points, then the first one will be returned.

`domain` must be a non-empty iterable.

`NaN` is treated as less than all other values except `missing`.

Julia 1.7

This method requires Julia 1.7 or later.

Examples

```
julia> findmin(identity, 5:9)
(5, 1)
```

```
julia> findmin(-, 1:10)
(-10, 10)

julia> findmin(first, [(2, :a), (2, :b), (3, :c)])
(2, 1)

julia> findmin(cos, 0:π/2:2π)
(-1.0, 3)
```

source

```
findmin(itr) -> (x, index)
```

Return the minimal element of the collection `itr` and its index or key. If there are multiple minimal elements, then the first one will be returned. `NaN` is treated as less than all other values except `missing`.

See also: `findmax`, `argmin`, `minimum`.

Examples

```
julia> findmin([8, 0.1, -9, pi])
(-9.0, 3)

julia> findmin([1, 7, 7, 6])
(1, 1)

julia> findmin([1, 7, 7, NaN])
(NaN, 4)
```

source

```
findmin(A; dims) -> (minval, index)
```

For an array input, returns the value and index of the minimum over the given dimensions. `NaN` is treated as less than all other values except `missing`.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> findmin(A, dims=1)
([1.0 2.0], CartesianIndex{2}[CartesianIndex(1, 1) CartesianIndex(1, 2)])

julia> findmin(A, dims=2)
([1.0; 3.0;;], CartesianIndex{2}[CartesianIndex(1, 1); CartesianIndex(2, 1);;])
```

source

```
findmin(f, A; dims) -> (f(x), index)
```

For an array input, returns the value in the codomain and index of the corresponding value which minimize `f` over the given dimensions.

Examples

```
julia> A = [-1.0 1; -0.5 2]
2×2 Matrix{Float64}:
 -1.0  1.0
 -0.5  2.0

julia> findmin(abs2, A, dims=1)
([0.25 1.0], CartesianIndex{2}[CartesianIndex(2, 1) CartesianIndex(1, 2)])

julia> findmin(abs2, A, dims=2)
([1.0; 0.25;;], CartesianIndex{2}[CartesianIndex(1, 1); CartesianIndex(2, 1);;])
```

source

`Base.findmax!` – Function.

```
findmax!(rval, rind, A) -> (maxval, index)
```

Find the maximum of `A` and the corresponding linear index along singleton dimensions of `rval` and `rind`, and store the results in `rval` and `rind`. `NaN` is treated as greater than all other values except `missing`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

`Base.findmin!` – Function.

```
findmin!(rval, rind, A) -> (minval, index)
```

Find the minimum of `A` and the corresponding linear index along singleton dimensions of `rval` and `rind`, and store the results in `rval` and `rind`. `NaN` is treated as less than all other values except `missing`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

`Base.sum` – Function.

```
sum(f, itr; [init])
```

Sum the results of calling function `f` on each element of `itr`.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

The value returned for empty `itr` can be specified by `init`. It must be the additive identity (i.e. zero) as it is unspecified whether `init` is used for non-empty collections.

> **Julia 1.6**
>
> Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> sum(abs2, [2; 3; 4])
29
```

Note the important difference between `sum(A)` and `reduce(+, A)` for arrays with small integer eltype:

```
julia> sum(Int8[100, 28])
128

julia> reduce(+, Int8[100, 28])
-128
```

In the former case, the integers are widened to system word size and therefore the result is 128. In the latter case, no such widening happens and integer overflow results in -128.

source

```
sum(itr; [init])
```

Return the sum of all elements in a collection.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

The value returned for empty `itr` can be specified by `init`. It must be the additive identity (i.e. zero) as it is unspecified whether `init` is used for non-empty collections.

> **Julia 1.6**
>
> Keyword argument `init` requires Julia 1.6 or later.

See also: `reduce`, `mapreduce`, `count`, `union`.

Examples

```
julia> sum(1:20)
210

julia> sum(1:20; init = 0.0)
210.0
```

source

```
sum(A::AbstractArray; dims)
```

Sum elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> sum(A, dims=1)
1×2 Matrix{Int64}:
 4  6

julia> sum(A, dims=2)
2×1 Matrix{Int64}:
 3
 7
```

source

```
sum(f, A::AbstractArray; dims)
```

Sum the results of calling function `f` on each element of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> sum(abs2, A, dims=1)
1×2 Matrix{Int64}:
 10  20

julia> sum(abs2, A, dims=2)
2×1 Matrix{Int64}:
  5
 25
```

source

`Base.sum!` – Function.

```
sum!(r, A)
```

Sum elements of A over the singleton dimensions of `r`, and write results to `r`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> sum!([1; 1], A)
2-element Vector{Int64}:
 3
 7

julia> sum!([1 1], A)
1×2 Matrix{Int64}:
 4  6
```

source

`Base.prod` – Function.

```
prod(f, itr; [init])
```

Return the product of `f` applied to each element of `itr`.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

The value returned for empty `itr` can be specified by `init`. It must be the multiplicative identity (i.e. one) as it is unspecified whether `init` is used for non-empty collections.

Julia 1.6

Keyword argument `init` requires Julia 1.6 or later.

Examples

```
julia> prod(abs2, [2; 3; 4])
576
```

source

```
prod(itr; [init])
```

Return the product of all elements of a collection.

The return type is `Int` for signed integers of less than system word size, and `UInt` for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

The value returned for empty `itr` can be specified by `init`. It must be the multiplicative identity (i.e. one) as it is unspecified whether `init` is used for non-empty collections.

> **Julia 1.6**
>
> Keyword argument `init` requires Julia 1.6 or later.

See also: `reduce`, `cumprod`, `any`.

Examples

```
julia> prod(1:5)
120

julia> prod(1:5; init = 1.0)
120.0
```

source

```
prod(A::AbstractArray; dims)
```

Multiply elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> prod(A, dims=1)
1×2 Matrix{Int64}:
 3  8

julia> prod(A, dims=2)
2×1 Matrix{Int64}:
  2
 12
```

source

```
prod(f, A::AbstractArray; dims)
```

Multiply the results of calling the function `f` on each element of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> prod(abs2, A, dims=1)
1×2 Matrix{Int64}:
 9  64

julia> prod(abs2, A, dims=2)
2×1 Matrix{Int64}:
   4
 144
```

source

`Base.prod!` – Function.

```
prod!(r, A)
```

Multiply elements of `A` over the singleton dimensions of `r`, and write results to `r`.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> prod!([1; 1], A)
2-element Vector{Int64}:
  2
 12

julia> prod!([1 1], A)
1×2 Matrix{Int64}:
 3  8
```

source

Base.any – Method.

```
any(itr) -> Bool
```

Test whether any elements of a boolean collection are `true`, returning `true` as soon as the first `true` value in `itr` is encountered (short-circuiting). To short-circuit on `false`, use `all`.

If the input contains `missing` values, return `missing` if all non-missing values are `false` (or equivalently, if the input contains no `true` value), following three-valued logic.

See also: `all`, `count`, `sum`, `|`, , `||`.

Examples

```
julia> a = [true,false,false,true]
4-element Vector{Bool}:
 1
 0
 0
 1

julia> any(a)
true

julia> any((println(i); v) for (i, v) in enumerate(a))
1
true

julia> any([missing, true])
true

julia> any([false, missing])
missing
```

source

Base.any – Method.

```
any(p, itr) -> Bool
```

Determine whether predicate `p` returns `true` for any elements of `itr`, returning `true` as soon as the first item in `itr` for which `p` returns `true` is encountered (short-circuiting). To short-circuit on `false`, use `all`.

If the input contains `missing` values, return `missing` if all non-missing values are `false` (or equivalently, if the input contains no `true` value), following three-valued logic.

Examples

```
julia> any(i->(4<=i<=6), [3,5,7])
true

julia> any(i -> (println(i); i > 3), 1:10)
```

```
1
2
3
4
true

julia> any(i -> i > 0, [1, missing])
true

julia> any(i -> i > 0, [-1, missing])
missing

julia> any(i -> i > 0, [-1, 0])
false
```

source

Base.any! – Function.

```
any!(r, A)
```

Test whether any values in A along the singleton dimensions of r are true, and write results to r.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [true false; true false]
2×2 Matrix{Bool}:
 1  0
 1  0

julia> any!([1; 1], A)
2-element Vector{Int64}:
 1
 1

julia> any!([1 1], A)
1×2 Matrix{Int64}:
 1  0
```

source

Base.all – Method.

```
all(itr) -> Bool
```

Test whether all elements of a boolean collection are `true`, returning `false` as soon as the first `false` value in `itr` is encountered (short-circuiting). To short-circuit on `true`, use `any`.

If the input contains `missing` values, return `missing` if all non-missing values are `true` (or equivalently, if the input contains no `false` value), following three-valued logic.

See also: `all!`, `any`, `count`, `&`, , `&&`, `allunique`.

Examples

```
julia> a = [true,false,false,true]
4-element Vector{Bool}:
 1
 0
 0
 1

julia> all(a)
false

julia> all((println(i); v) for (i, v) in enumerate(a))
1
2
false

julia> all([missing, false])
false

julia> all([true, missing])
missing
```

source

`Base.all` – Method.

```
all(p, itr) -> Bool
```

Determine whether predicate `p` returns `true` for all elements of `itr`, returning `false` as soon as the first item in `itr` for which `p` returns `false` is encountered (short-circuiting). To short-circuit on `true`, use `any`.

If the input contains `missing` values, return `missing` if all non-missing values are `true` (or equivalently, if the input contains no `false` value), following three-valued logic.

Examples

```
julia> all(i->(4<=i<=6), [4,5,6])
true

julia> all(i -> (println(i); i < 3), 1:10)
1
2
3
false
```

```
julia> all(i -> i > 0, [1, missing])
missing

julia> all(i -> i > 0, [-1, missing])
false

julia> all(i -> i > 0, [1, 2])
true
```

source

Base.all! – Function.

```
all!(r, A)
```

Test whether all values in A along the singleton dimensions of r are true, and write results to r.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [true false; true false]
2×2 Matrix{Bool}:
 1  0
 1  0

julia> all!([1; 1], A)
2-element Vector{Int64}:
 0
 0

julia> all!([1 1], A)
1×2 Matrix{Int64}:
 1  0
```

source

Base.count – Function.

```
count([f=identity,] itr; init=0) -> Integer
```

Count the number of elements in itr for which the function f returns true. If f is omitted, count the number of true elements in itr (which should be a collection of boolean values). init optionally specifies the value to start counting from and therefore also determines the output type.

Julia 1.6

init keyword was added in Julia 1.6.

See also: any, sum.

Examples

```
julia> count(i->(4<=i<=6), [2,3,4,5,6])
3

julia> count([true, false, true, true])
3

julia> count(>(3), 1:7, init=0x03)
0x07
```

source

```
count(
    pattern::Union{AbstractChar,AbstractString,AbstractPattern},
    string::AbstractString;
    overlap::Bool = false,
)
```

Return the number of matches for `pattern` in `string`. This is equivalent to calling `length(findall(pattern, string))` but more efficient.

If `overlap=true`, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from disjoint character ranges.

Julia 1.3

This method requires at least Julia 1.3.

Julia 1.7

Using a character as the pattern requires at least Julia 1.7.

Examples

```
julia> count('a', "JuliaLang")
2

julia> count(r"a(.)a", "cabacabac", overlap=true)
3

julia> count(r"a(.)a", "cabacabac")
2
```

source

```
count([f=identity,] A::AbstractArray; dims=:)
```

Count the number of elements in A for which f returns true over the given dimensions.

Julia 1.5

dims keyword was added in Julia 1.5.

Julia 1.6

init keyword was added in Julia 1.6.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> count(<=(2), A, dims=1)
1×2 Matrix{Int64}:
 1  1

julia> count(<=(2), A, dims=2)
2×1 Matrix{Int64}:
 2
 0
```

source

Base.foreach – Function.

```
foreach(f, c...) -> Nothing
```

Call function f on each element of iterable c. For multiple iterable arguments, f is called elementwise, and iteration stops when any iterator is finished.

foreach should be used instead of map when the results of f are not needed, for example in foreach(println, array).

Examples

```
julia> tri = 1:3:7; res = Int[];

julia> foreach(x -> push!(res, x^2), tri)

julia> res
3-element Vector{Int64}:
  1
 16
```

```
 49

julia> foreach((x, y) -> println(x, " with ", y), tri, 'a':'z')
1 with a
4 with b
7 with c
```

source

Base.map – Function.

```
map(f, c...) -> collection
```

Transform collection c by applying f to each element. For multiple collection arguments, apply f elementwise, and stop when any of them is exhausted.

See also map!, foreach, mapreduce, mapslices, zip, Iterators.map.

Examples

```
julia> map(x -> x * 2, [1, 2, 3])
3-element Vector{Int64}:
 2
 4
 6

julia> map(+, [1, 2, 3], [10, 20, 30, 400, 5000])
3-element Vector{Int64}:
 11
 22
 33
```

source

```
map(f, A::AbstractArray...) -> N-array
```

When acting on multi-dimensional arrays of the same ndims, they must all have the same axes, and the answer will too.

See also broadcast, which allows mismatched sizes.

Examples

```
julia> map(//, [1 2; 3 4], [4 3; 2 1])
2×2 Matrix{Rational{Int64}}:
 1//4  2//3
 3//2  4//1

julia> map(+, [1 2; 3 4], zeros(2,1))
ERROR: DimensionMismatch
```

```
julia> map(+, [1 2; 3 4], [1,10,100,1000], zeros(3,1))  # iterates until 3rd is exhausted
3-element Vector{Float64}:
   2.0
  13.0
 102.0
```

source

`Base.map!` – Function.

```
map!(function, destination, collection...)
```

Like `map`, but stores the result in `destination` rather than a new collection. `destination` must be at least as large as the smallest collection.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also: `map`, `foreach`, `zip`, `copyto!`.

Examples

```
julia> a = zeros(3);

julia> map!(x -> x * 2, a, [1, 2, 3]);

julia> a
3-element Vector{Float64}:
 2.0
 4.0
 6.0

julia> map!(+, zeros(Int, 5), 100:999, 1:3)
5-element Vector{Int64}:
 101
 103
 105
   0
   0
```

source

```
map!(f, values(dict::AbstractDict))
```

Modifies `dict` by transforming each value from `val` to `f(val)`. Note that the type of `dict` cannot be changed: if `f(val)` is not an instance of the value type of `dict` then it will be converted to the value type if possible and otherwise raise an error.

Julia 1.2

`map!(f, values(dict::AbstractDict))` requires Julia 1.2 or later.

Examples

```
julia> d = Dict(:a => 1, :b => 2)
Dict{Symbol, Int64} with 2 entries:
  :a => 1
  :b => 2

julia> map!(v -> v-1, values(d))
ValueIterator for a Dict{Symbol, Int64} with 2 entries. Values:
  0
  1
```

source

`Base.mapreduce` – Method.

```
mapreduce(f, op, itrs...; [init])
```

Apply function `f` to each element(s) in `itrs`, and then reduce the result using the binary function `op`. If provided, `init` must be a neutral element for `op` that will be returned for empty collections. It is unspecified whether `init` is used for non-empty collections. In general, it will be necessary to provide `init` to work with empty collections.

`mapreduce` is functionally equivalent to calling `reduce(op, map(f, itr); init=init)`, but will in general execute faster since no intermediate collection needs to be created. See documentation for `reduce` and `map`.

Julia 1.2

`mapreduce` with multiple iterators requires Julia 1.2 or later.

Examples

```
julia> mapreduce(x->x^2, +, [1:3;]) # == 1 + 4 + 9
14
```

The associativity of the reduction is implementation-dependent. Additionally, some implementations may reuse the return value of `f` for elements that appear multiple times in `itr`. Use `mapfoldl` or `mapfoldr` instead for guaranteed left or right associativity and invocation of `f` for every value.

source

`Base.mapfoldl` – Method.

```
mapfoldl(f, op, itr; [init])
```

Like `mapreduce`, but with guaranteed left associativity, as in `foldl`. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

source

`Base.mapfoldr` – Method.

```
mapfoldr(f, op, itr; [init])
```

Like `mapreduce`, but with guaranteed right associativity, as in `foldr`. If provided, the keyword argument `init` will be used exactly once. In general, it will be necessary to provide `init` to work with empty collections.

source

`Base.first` – Function.

```
first(coll)
```

Get the first element of an iterable collection. Return the start point of an `AbstractRange` even if it is empty.

See also: `only`, `firstindex`, `last`.

Examples

```
julia> first(2:2:10)
2

julia> first([1; 2; 3; 4])
1
```

source

```
first(itr, n::Integer)
```

Get the first n elements of the iterable collection `itr`, or fewer elements if `itr` is not long enough.

See also: `startswith`, `Iterators.take`.

> **Julia 1.6**
>
> This method requires at least Julia 1.6.

Examples

```
julia> first(["foo", "bar", "qux"], 2)
2-element Vector{String}:
 "foo"
 "bar"

julia> first(1:6, 10)
1:6

julia> first(Bool[], 1)
Bool[]
```

source

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first n characters of s.

Examples

```
julia> first("∀ϵ≠0: ϵ²>0", 0)
""

julia> first("∀ϵ≠0: ϵ²>0", 1)
"∀"

julia> first("∀ϵ≠0: ϵ²>0", 3)
"∀ϵ≠"
```

source

Base.last – Function.

```
last(coll)
```

Get the last element of an ordered collection, if it can be computed in O(1) time. This is accomplished by calling lastindex to get the last index. Return the end point of an AbstractRange even if it is empty.

See also first, endswith.

Examples

```
julia> last(1:2:10)
9

julia> last([1; 2; 3; 4])
4
```

source

```
last(itr, n::Integer)
```

Get the last n elements of the iterable collection `itr`, or fewer elements if `itr` is not long enough.

Julia 1.6

This method requires at least Julia 1.6.

Examples

```
julia> last(["foo", "bar", "qux"], 2)
2-element Vector{String}:
 "bar"
 "qux"

julia> last(1:6, 10)
1:6

julia> last(Float64[], 1)
Float64[]
```

source

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last n characters of s.

Examples

```
julia> last("∀ϵ≠0: ϵ²>0", 0)
""

julia> last("∀ϵ≠0: ϵ²>0", 1)
"0"

julia> last("∀ϵ≠0: ϵ²>0", 3)
"²>0"
```

source

`Base.front` – Function.

```
front(x::Tuple)::Tuple
```

Return a `Tuple` consisting of all but the last component of `x`.

See also: `first`, `tail`.

Examples

```
julia> Base.front((1,2,3))
(1, 2)

julia> Base.front(())
ERROR: ArgumentError: Cannot call front on an empty tuple.
```

source

Base.tail – Function.

```
tail(x::Tuple)::Tuple
```

Return a Tuple consisting of all but the first component of x.

See also: front, rest, first, Iterators.peel.

Examples

```
julia> Base.tail((1,2,3))
(2, 3)

julia> Base.tail(())
ERROR: ArgumentError: Cannot call tail on an empty tuple.
```

source

Base.step – Function.

```
step(r)
```

Get the step size of an AbstractRange object.

Examples

```
julia> step(1:10)
1

julia> step(1:2:10)
2

julia> step(2.5:0.3:10.9)
0.3

julia> step(range(2.5, stop=10.9, length=85))
0.1
```

source

Base.collect – Method.

```
collect(collection)
```

Return an `Array` of all items in a collection or iterator. For dictionaries, returns `Vector{Pair{KeyType, ValType}}`. If the argument is array-like or is an iterator with the `HasShape` trait, the result will have the same shape and number of dimensions as the argument.

Used by comprehensions to turn a generator into an `Array`.

Examples

```
julia> collect(1:2:13)
7-element Vector{Int64}:
  1
  3
  5
  7
  9
 11
 13

julia> [x^2 for x in 1:8 if isodd(x)]
4-element Vector{Int64}:
  1
  9
 25
 49
```

source

`Base.collect` – Method.

```
collect(element_type, collection)
```

Return an `Array` with the given element type of all items in a collection or iterable. The result has the same shape and number of dimensions as `collection`.

Examples

```
julia> collect(Float64, 1:2:5)
3-element Vector{Float64}:
 1.0
 3.0
 5.0
```

source

`Base.filter` – Function.

```
filter(f, a)
```

Return a copy of collection a, removing elements for which f is false. The function f is passed one argument.

Julia 1.4

Support for a as a tuple requires at least Julia 1.4.

See also: filter!, Iterators.filter.

Examples

```
julia> a = 1:10
1:10

julia> filter(isodd, a)
5-element Vector{Int64}:
 1
 3
 5
 7
 9
```

source

```
filter(f)
```

Create a function that filters its arguments with function f using filter, i.e. a function equivalent to x -> filter(f, x).

The returned function is of type Base.Fix1{typeof(filter)}, which can be used to implement specialized methods.

Examples

```
julia> (1, 2, Inf, 4, NaN, 6) |> filter(isfinite)
(1, 2, 4, 6)

julia> map(filter(iseven), [1:3, 2:4, 3:5])
3-element Vector{Vector{Int64}}:
 [2]
 [2, 4]
 [4]
```

Julia 1.9

This method requires at least Julia 1.9.

source

```
filter(f, d::AbstractDict)
```

Return a copy of d, removing elements for which f is false. The function f is passed key=>value pairs.

Examples

```
julia> d = Dict(1=>"a", 2=>"b")
Dict{Int64, String} with 2 entries:
  2 => "b"
  1 => "a"

julia> filter(p->isodd(p.first), d)
Dict{Int64, String} with 1 entry:
  1 => "a"
```

source

```
filter(f, itr::SkipMissing{<:AbstractArray})
```

Return a vector similar to the array wrapped by the given SkipMissing iterator but with all missing elements and those for which f returns false removed.

Julia 1.2

This method requires Julia 1.2 or later.

Examples

```
julia> x = [1 2; missing 4]
2×2 Matrix{Union{Missing, Int64}}:
 1         2
  missing  4

julia> filter(isodd, skipmissing(x))
1-element Vector{Int64}:
 1
```

source

Base.filter! – Function.

```
filter!(f, a)
```

Update collection a, removing elements for which f is false. The function f is passed one argument.

Examples

```
julia> filter!(isodd, Vector(1:10))
5-element Vector{Int64}:
 1
 3
 5
 7
 9
```

source

```
filter!(f, d::AbstractDict)
```

Update d, removing elements for which f is false. The function f is passed key=>value pairs.

Example

```
julia> d = Dict(1=>"a", 2=>"b", 3=>"c")
Dict{Int64, String} with 3 entries:
  2 => "b"
  3 => "c"
  1 => "a"

julia> filter!(p->isodd(p.first), d)
Dict{Int64, String} with 2 entries:
  3 => "c"
  1 => "a"
```

source

Base.replace – Method.

```
replace(A, old_new::Pair...; [count::Integer])
```

Return a copy of collection A where, for each pair old=>new in old_new, all occurrences of old are replaced by new. Equality is determined using isequal. If count is specified, then replace at most count occurrences in total.

The element type of the result is chosen using promotion (see promote_type) based on the element type of A and on the types of the new values in pairs. If count is omitted and the element type of A is a Union, the element type of the result will not include singleton types which are replaced with values of a different type: for example, Union{T,Missing} will become T if missing is replaced.

See also replace!, splice!, delete!, insert!.

> **Julia 1.7**
>
> Version 1.7 is required to replace elements of a Tuple.

Examples

```
julia> replace([1, 2, 1, 3], 1=>0, 2=>4, count=2)
4-element Vector{Int64}:
 0
 4
 1
 3

julia> replace([1, missing], missing=>0)
2-element Vector{Int64}:
 1
 0
```

source

Base.replace – Method.

```
replace(new::Union{Function, Type}, A; [count::Integer])
```

Return a copy of A where each value x in A is replaced by new(x). If count is specified, then replace at most count values in total (replacements being defined as new(x) !== x).

Julia 1.7

Version 1.7 is required to replace elements of a Tuple.

Examples

```
julia> replace(x -> isodd(x) ? 2x : x, [1, 2, 3, 4])
4-element Vector{Int64}:
 2
 2
 6
 4

julia> replace(Dict(1=>2, 3=>4)) do kv
           first(kv) < 3 ? first(kv)=>3 : kv
       end
Dict{Int64, Int64} with 2 entries:
  3 => 4
  1 => 3
```

source

Base.replace! – Function.

```
replace!(A, old_new::Pair...; [count::Integer])
```

For each pair old=>new in old_new, replace all occurrences of old in collection A by new. Equality is determined using isequal. If count is specified, then replace at most count occurrences in total. See also replace.

Examples

```
julia> replace!([1, 2, 1, 3], 1=>0, 2=>4, count=2)
4-element Vector{Int64}:
 0
 4
 1
 3

julia> replace!(Set([1, 2, 3]), 1=>0)
Set{Int64} with 3 elements:
```

```
 0
 2
 3
```

source

```
replace!(new::Union{Function, Type}, A; [count::Integer])
```

Replace each element x in collection A by `new(x)`. If `count` is specified, then replace at most `count` values in total (replacements being defined as `new(x) !== x`).

Examples

```
julia> replace!(x -> isodd(x) ? 2x : x, [1, 2, 3, 4])
4-element Vector{Int64}:
 2
 2
 6
 4

julia> replace!(Dict(1=>2, 3=>4)) do kv
           first(kv) < 3 ? first(kv)=>3 : kv
       end
Dict{Int64, Int64} with 2 entries:
  3 => 4
  1 => 3

julia> replace!(x->2x, Set([3, 6]))
Set{Int64} with 2 elements:
  6
  12
```

source

`Base.rest` – Function.

```
Base.rest(collection[, itr_state])
```

Generic function for taking the tail of `collection`, starting from a specific iteration state `itr_state`. Return a `Tuple`, if `collection` itself is a `Tuple`, a subtype of `AbstractVector`, if `collection` is an `AbstractArray`, a subtype of `AbstractString` if `collection` is an `AbstractString`, and an arbitrary iterator, falling back to `Iterators.rest(collection[, itr_state])`, otherwise.

Can be overloaded for user-defined collection types to customize the behavior of slurping in assignments in final position, like `a, b... = collection`.

Julia 1.6

`Base.rest` requires at least Julia 1.6.

See also: `first`, `Iterators.rest`, `Base.split_rest`.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> first, state = iterate(a)
(1, 2)

julia> first, Base.rest(a, state)
(1, [3, 2, 4])
```

source

`Base.split_rest` – Function.

```
Base.split_rest(collection, n::Int[, itr_state]) -> (rest_but_n, last_n)
```

Generic function for splitting the tail of `collection`, starting from a specific iteration state `itr_state`. Returns a tuple of two new collections. The first one contains all elements of the tail but the `n` last ones, which make up the second collection.

The type of the first collection generally follows that of `Base.rest`, except that the fallback case is not lazy, but is collected eagerly into a vector.

Can be overloaded for user-defined collection types to customize the behavior of slurping in assignments in non-final position, like `a, b..., c = collection`.

Julia 1.9

`Base.split_rest` requires at least Julia 1.9.

See also: `Base.rest`.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> first, state = iterate(a)
(1, 2)

julia> first, Base.split_rest(a, 1, state)
(1, ([3, 2], [4]))
```

source

43.5 可索引集合

`Base.getindex` – Function.

```
getindex(collection, key...)
```

Retrieve the value(s) stored at the given key or index within a collection. The syntax `a[i,j,...]` is converted by the compiler to `getindex(a, i, j, ...)`.

See also `get`, `keys`, `eachindex`.

Examples

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String, Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> getindex(A, "a")
1
```

source

`Base.setindex!` – Function.

```
setindex!(collection, value, key...)
```

Store the given value at the given key or index within a collection. The syntax `a[i,j,...] = x` is converted by the compiler to `(setindex!(a, x, i, j, ...); x)`.

Examples

```
julia> a = Dict("a"=>1)
Dict{String, Int64} with 1 entry:
  "a" => 1

julia> setindex!(a, 2, "b")
Dict{String, Int64} with 2 entries:
  "b" => 2
  "a" => 1
```

source

`Base.firstindex` – Function.

```
firstindex(collection) -> Integer
firstindex(collection, d) -> Integer
```

Return the first index of `collection`. If `d` is given, return the first index of `collection` along dimension `d`.

The syntaxes `A[begin]` and `A[1, begin]` lower to `A[firstindex(A)]` and `A[1, firstindex(A, 2)]`, respectively.

See also: `first`, `axes`, `lastindex`, `nextind`.

Examples

```
julia> firstindex([1,2,4])
1

julia> firstindex(rand(3,4,5), 2)
1
```

source

`Base.lastindex` – Function.

```
lastindex(collection) -> Integer
lastindex(collection, d) -> Integer
```

Return the last index of `collection`. If `d` is given, return the last index of `collection` along dimension `d`.

The syntaxes `A[end]` and `A[end, end]` lower to `A[lastindex(A)]` and `A[lastindex(A, 1), lastindex(A, 2)]`, respectively.

See also: `axes`, `firstindex`, `eachindex`, `prevind`.

Examples

```
julia> lastindex([1,2,4])
3

julia> lastindex(rand(3,4,5), 2)
4
```

source

以下类型均完全实现了上述函数：

- `Array`
- `BitArray`
- `AbstractArray`
- `SubArray`

以下类型仅实现了部分上述函数：

- `AbstractRange`
- `UnitRange`

- Tuple
- AbstractString
- Dict
- IdDict
- WeakKeyDict
- NamedTuple

43.6 字典

Dict 是一个标准字典。其实现利用了 hash 作为键的哈希函数和 isequal 来决定是否相等。对于自定义类型，可以定义这两个函数来重载它们在哈希表内的存储方式。

IdDict 是一种特殊的哈希表，在里面键始终是对象标识符。

WeakKeyDict 是一个哈希表的实现，里面键是对象的弱引用，所以即使键在哈希表中被引用也有可能被垃圾回收。它像 Dict 一样使用 hash 来做哈希和 isequal 来做相等判断，但是它不会在插入时转换键，这点不像 Dict。

Dicts 可以由传递含有 => 的成对对象给 Dict 的构造函数来被创建：Dict("A"=>1, "B"=>2)。这个调用会尝试从键值对中推到类型信息（比如这个例子创造了一个 Dict{String, Int64}）。为了显式指定类型，请使用语法 Dict{KeyType,ValueType}(...)。例如：Dict{String,Int32}("A"=>1, "B"=>2)。

字典也可以用生成器创建。例如：Dict(i => f(i) for i = 1:10)。

对于字典 D，若键 x 的值存在，则语法 D[x] 返回 x 的值；否则抛出一个错误。D[x] = y 存储键值对 x => y 到 D 中，会覆盖键 x 的已有的值。多个参数传入 D[...] 会被转化成元组；例如：语法 D[x,y] 等于 D[(x,y)]，也就是说，它指向键为元组 (x,y) 的值。

Base.AbstractDict – Type.

```
AbstractDict{K, V}
```

Supertype for dictionary-like types with keys of type K and values of type V. Dict, IdDict and other types are subtypes of this. An AbstractDict{K, V} should be an iterator of Pair{K, V}.

source

Base.Dict – Type.

```
Dict([itr])
```

Dict{K,V}() constructs a hash table with keys of type K and values of type V. Keys are compared with isequal and hashed with hash.

Given a single iterable argument, constructs a Dict whose key-value pairs are taken from 2-tuples (key,value) generated by the argument.

Examples

```
julia> Dict([("A", 1), ("B", 2)])
Dict{String, Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

Alternatively, a sequence of pair arguments may be passed.

```
julia> Dict("A"=>1, "B"=>2)
Dict{String, Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

source

Base.IdDict – Type.

```
IdDict([itr])
```

`IdDict{K,V}()` constructs a hash table using `objectid` as hash and `===` as equality with keys of type `K` and values of type `V`.

See `Dict` for further help. In the example below, The `Dict` keys are all `isequal` and therefore get hashed the same, so they get overwritten. The `IdDict` hashes by object-id, and thus preserves the 3 different keys.

Examples

```
julia> Dict(true => "yes", 1 => "no", 1.0 => "maybe")
Dict{Real, String} with 1 entry:
  1.0 => "maybe"

julia> IdDict(true => "yes", 1 => "no", 1.0 => "maybe")
IdDict{Any, String} with 3 entries:
  true => "yes"
  1.0  => "maybe"
  1    => "no"
```

source

Base.WeakKeyDict – Type.

```
WeakKeyDict([itr])
```

`WeakKeyDict()` constructs a hash table where the keys are weak references to objects which may be garbage collected even when referenced in a hash table.

See `Dict` for further help. Note, unlike `Dict`, `WeakKeyDict` does not convert keys on insertion, as this would imply the key object was unreferenced anywhere before insertion.

See also `WeakRef`.

source

`Base.ImmutableDict` – Type.

```
ImmutableDict
```

`ImmutableDict` is a dictionary implemented as an immutable linked list, which is optimal for small dictionaries that are constructed over many individual insertions. Note that it is not possible to remove a value, although it can be partially overridden and hidden by inserting a new value with the same key.

```
ImmutableDict(KV::Pair)
```

Create a new entry in the `ImmutableDict` for a `key => value` pair

- use `(key => value) in dict` to see if this particular combination is in the properties set
- use `get(dict, key, default)` to retrieve the most recent value for a particular key

source

`Base.haskey` – Function.

```
haskey(collection, key) -> Bool
```

Determine whether a collection has a mapping for a given key.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char, Int64} with 2 entries:
  'a' => 2
  'b' => 3

julia> haskey(D, 'a')
true

julia> haskey(D, 'c')
false
```

source

`Base.get` – Function.

```
get(collection, key, default)
```

Return the value stored for the given key, or the given default value if no mapping for the key is present.

Julia 1.7

For tuples and numbers, this function requires at least Julia 1.7.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> get(d, "a", 3)
1

julia> get(d, "c", 3)
3
```

source

```
get(f::Union{Function, Type}, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, return `f()`. Use `get!` to also store the default value in the dictionary.

This is intended to be called using do block syntax

```
get(dict, key) do
    # default value calculated here
    time()
end
```

source

`Base.get!` – Function.

```
get!(collection, key, default)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => default`, and return `default`.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> get!(d, "a", 5)
1

julia> get!(d, "d", 4)
4

julia> d
Dict{String, Int64} with 4 entries:
  "c" => 3
  "b" => 2
  "a" => 1
  "d" => 4
```

source

```
get!(f::Union{Function, Type}, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => f()`, and return `f()`.

This is intended to be called using do block syntax.

Examples

```
julia> squares = Dict{Int, Int}();

julia> function get_square!(d, i)
           get!(d, i) do
               i^2
           end
       end
get_square! (generic function with 1 method)

julia> get_square!(squares, 2)
4

julia> squares
Dict{Int64, Int64} with 1 entry:
  2 => 4
```

source

`Base.getkey` – Function.

```
getkey(collection, key, default)
```

Return the key matching argument key if one exists in `collection`, otherwise return `default`.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char, Int64} with 2 entries:
  'a' => 2
  'b' => 3

julia> getkey(D, 'a', 1)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> getkey(D, 'd', 'a')
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

source

`Base.delete!` – Function.

```
delete!(collection, key)
```

Delete the mapping for the given key in a collection, if any, and return the collection.

Examples

```
julia> d = Dict("a"=>1, "b"=>2)
Dict{String, Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> delete!(d, "b")
Dict{String, Int64} with 1 entry:
  "a" => 1

julia> delete!(d, "b") # d is left unchanged
Dict{String, Int64} with 1 entry:
  "a" => 1
```

source

Base.pop! – Method.

```
pop!(collection, key[, default])
```

Delete and return the mapping for key if it exists in `collection`, otherwise return `default`, or throw an error if `default` is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> pop!(d, "a")
1

julia> pop!(d, "d")
ERROR: KeyError: key "d" not found
Stacktrace:
[...]

julia> pop!(d, "e", 4)
4
```

source

Base.keys – Function.

```
keys(iterator)
```

For an iterator or collection that has keys and values (e.g. arrays and dictionaries), return an iterator over the keys.

source

Base.values – Function.

```
values(iterator)
```

For an iterator or collection that has keys and values, return an iterator over the values. This function simply returns its argument by default, since the elements of a general iterator are normally considered its "values".

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> values(d)
ValueIterator for a Dict{String, Int64} with 2 entries. Values:
  2
  1

julia> values([2])
1-element Vector{Int64}:
 2
```

source

```
values(a::AbstractDict)
```

Return an iterator over all values in a collection. `collect(values(a))` returns an array of values. When the values are stored internally in a hash table, as is the case for `Dict`, the order in which they are returned may vary. But `keys(a)`, `values(a)` and `pairs(a)` all iterate a and return the elements in the same order.

Examples

```
julia> D = Dict('a'=>2, 'b'=>3)
Dict{Char, Int64} with 2 entries:
  'a' => 2
  'b' => 3

julia> collect(values(D))
2-element Vector{Int64}:
 2
 3
```

source

Base.pairs – Function.

```
pairs(IndexLinear(), A)
pairs(IndexCartesian(), A)
pairs(IndexStyle(A), A)
```

An iterator that accesses each element of the array `A`, returning `i => x`, where `i` is the index for the element and `x = A[i]`. Identical to `pairs(A)`, except that the style of index can be selected. Also similar to `enumerate(A)`, except `i` will be a valid index for `A`, while `enumerate` always counts from 1 regardless of the indices of `A`.

Specifying `IndexLinear()` ensures that `i` will be an integer; specifying `IndexCartesian()` ensures that `i` will be a `Base.CartesianIndex`; specifying `IndexStyle(A)` chooses whichever has been defined as the native indexing style for array `A`.

Mutation of the bounds of the underlying array will invalidate this iterator.

Examples

```
julia> A = ["a" "d"; "b" "e"; "c" "f"];

julia> for (index, value) in pairs(IndexStyle(A), A)
           println("$index $value")
       end
1 a
2 b
3 c
4 d
5 e
6 f

julia> S = view(A, 1:2, :);

julia> for (index, value) in pairs(IndexStyle(S), S)
           println("$index $value")
       end
CartesianIndex(1, 1) a
CartesianIndex(2, 1) b
CartesianIndex(1, 2) d
CartesianIndex(2, 2) e
```

See also `IndexStyle`, `axes`.

source

```
pairs(collection)
```

Return an iterator over `key => value` pairs for any collection that maps a set of keys to a set of values. This includes arrays, where the keys are the array indices. When the entries are stored internally in a hash table, as is the case for `Dict`, the order in which they are returned may vary. But `keys(a)`, `values(a)` and `pairs(a)` all iterate `a` and return the elements in the same order.

Examples

```
julia> a = Dict(zip(["a", "b", "c"], [1, 2, 3]))
Dict{String, Int64} with 3 entries:
  "c" => 3
  "b" => 2
  "a" => 1

julia> pairs(a)
Dict{String, Int64} with 3 entries:
  "c" => 3
  "b" => 2
  "a" => 1

julia> foreach(println, pairs(["a", "b", "c"]))
1 => "a"
2 => "b"
3 => "c"

julia> (;a=1, b=2, c=3) |> pairs |> collect
3-element Vector{Pair{Symbol, Int64}}:
 :a => 1
 :b => 2
 :c => 3

julia> (;a=1, b=2, c=3) |> collect
3-element Vector{Int64}:
 1
 2
 3
```

source

Base.merge – Function.

```
merge(d::AbstractDict, others::AbstractDict...)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. If the same key is present in another collection, the value for that key will be the value it has in the last collection listed. See also mergewith for custom handling of values with the same key.

Examples

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)
Dict{String, Float64} with 2 entries:
  "bar" => 42.0
  "foo" => 0.0

julia> b = Dict("baz" => 17, "bar" => 4711)
Dict{String, Int64} with 2 entries:
  "bar" => 4711
  "baz" => 17

julia> merge(a, b)
```

```
Dict{String, Float64} with 3 entries:
  "bar" => 4711.0
  "baz" => 17.0
  "foo" => 0.0

julia> merge(b, a)
Dict{String, Float64} with 3 entries:
  "bar" => 42.0
  "baz" => 17.0
  "foo" => 0.0
```

source

```
merge(a::NamedTuple, bs::NamedTuple...)
```

Construct a new named tuple by merging two or more existing ones, in a left-associative manner. Merging proceeds left-to-right, between pairs of named tuples, and so the order of fields present in both the leftmost and rightmost named tuples take the same position as they are found in the leftmost named tuple. However, values are taken from matching fields in the rightmost named tuple that contains that field. Fields present in only the rightmost named tuple of a pair are appended at the end. A fallback is implemented for when only a single named tuple is supplied, with signature `merge(a::NamedTuple)`.

Julia 1.1

Merging 3 or more `NamedTuple` requires at least Julia 1.1.

Examples

```
julia> merge((a=1, b=2, c=3), (b=4, d=5))
(a = 1, b = 4, c = 3, d = 5)
```

```
julia> merge((a=1, b=2), (b=3, c=(d=1,)), (c=(d=2,),))
(a = 1, b = 3, c = (d = 2,))
```

source

```
merge(a::NamedTuple, iterable)
```

Interpret an iterable of key-value pairs as a named tuple, and perform a merge.

```
julia> merge((a=1, b=2, c=3), [:b=>4, :d=>5])
(a = 1, b = 4, c = 3, d = 5)
```

source

`Base.mergewith` – Function.

```
mergewith(combine, d::AbstractDict, others::AbstractDict...)
mergewith(combine)
merge(combine, d::AbstractDict, others::AbstractDict...)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. Values with the same key will be combined using the combiner function. The curried form `mergewith(combine)` returns the function `(args...) -> mergewith(combine, args...)`.

Method `merge(combine::Union{Function,Type}, args...)` as an alias of `mergewith(combine, args...)` is still available for backward compatibility.

> **Julia 1.5**
>
> `mergewith` requires Julia 1.5 or later.

Examples

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)
Dict{String, Float64} with 2 entries:
  "bar" => 42.0
  "foo" => 0.0

julia> b = Dict("baz" => 17, "bar" => 4711)
Dict{String, Int64} with 2 entries:
  "bar" => 4711
  "baz" => 17

julia> mergewith(+, a, b)
Dict{String, Float64} with 3 entries:
  "bar" => 4753.0
  "baz" => 17.0
  "foo" => 0.0

julia> ans == mergewith(+)(a, b)
true
```

source

`Base.merge!` – Function.

```
merge!(d::AbstractDict, others::AbstractDict...)
```

Update collection with pairs from the other collections. See also `merge`.

Examples

```
julia> d1 = Dict(1 => 2, 3 => 4);

julia> d2 = Dict(1 => 4, 4 => 5);
```

```
julia> merge!(d1, d2);

julia> d1
Dict{Int64, Int64} with 3 entries:
  4 => 5
  3 => 4
  1 => 4
```

source

Base.mergewith! – Function.

```
mergewith!(combine, d::AbstractDict, others::AbstractDict...) -> d
mergewith!(combine)
merge!(combine, d::AbstractDict, others::AbstractDict...) -> d
```

Update collection with pairs from the other collections. Values with the same key will be combined using the combiner function. The curried form `mergewith!(combine)` returns the function `(args...) -> mergewith!(combine, args...)`.

Method `merge!(combine::Union{Function,Type}, args...)` as an alias of `mergewith!(combine, args...)` is still available for backward compatibility.

Julia 1.5

`mergewith!` requires Julia 1.5 or later.

Examples

```
julia> d1 = Dict(1 => 2, 3 => 4);

julia> d2 = Dict(1 => 4, 4 => 5);

julia> mergewith!(+, d1, d2);

julia> d1
Dict{Int64, Int64} with 3 entries:
  4 => 5
  3 => 4
  1 => 6

julia> mergewith!(-, d1, d1);

julia> d1
Dict{Int64, Int64} with 3 entries:
  4 => 0
  3 => 0
  1 => 0

julia> foldl(mergewith!(+), [d1, d2]; init=Dict{Int64, Int64}())
Dict{Int64, Int64} with 3 entries:
```

```
4 => 5
3 => 0
1 => 4
```

source

Base.sizehint! – Function.

```
sizehint!(s, n) -> s
```

Suggest that collection s reserve capacity for at least n elements. That is, if you expect that you're going to have to push a lot of values onto s, you can avoid the cost of incremental reallocation by doing it once up front; this can improve performance.

See also resize!.

Notes on the performance model

For types that support sizehint!,

1. push! and append! methods generally may (but are not required to) preallocate extra storage. For types implemented in Base, they typically do, using a heuristic optimized for a general use case.
2. sizehint! may control this preallocation. Again, it typically does this for types in Base.
3. empty! is nearly costless (and O(1)) for types that support this kind of preallocation.

source

Base.keytype – Function.

```
keytype(T::Type{<:AbstractArray})
keytype(A::AbstractArray)
```

Return the key type of an array. This is equal to the eltype of the result of keys(...), and is provided mainly for compatibility with the dictionary interface.

Examples

```
julia> keytype([1, 2, 3]) == Int
true

julia> keytype([1 2; 3 4])
CartesianIndex{2}
```

Julia 1.2

For arrays, this function requires at least Julia 1.2.

source

```
keytype(type)
```

Get the key type of a dictionary type. Behaves similarly to `eltype`.

Examples

```
julia> keytype(Dict(Int32(1) => "foo"))
Int32
```

source

`Base.valtype` – Function.

```
valtype(T::Type{<:AbstractArray})
valtype(A::AbstractArray)
```

Return the value type of an array. This is identical to `eltype` and is provided mainly for compatibility with the dictionary interface.

Examples

```
julia> valtype(["one", "two", "three"])
String
```

> **Julia 1.2**
>
> For arrays, this function requires at least Julia 1.2.

source

```
valtype(type)
```

Get the value type of a dictionary type. Behaves similarly to `eltype`.

Examples

```
julia> valtype(Dict(Int32(1) => "foo"))
String
```

source

以下类型均完全实现了上述函数:

- `IdDict`
- `Dict`
- `WeakKeyDict`

以下类型仅实现了部分上述函数：

- BitSet
- Set
- EnvDict
- Array
- BitArray
- ImmutableDict
- Iterators.Pairs

43.7 类似 Set 的集合

Base.AbstractSet – Type.

```
AbstractSet{T}
```

Supertype for set-like types whose elements are of type T. Set, BitSet and other types are subtypes of this.

source

Base.Set – Type.

```
Set{T} <: AbstractSet{T}
```

Sets are mutable containers that provide fast membership testing.

Sets have efficient implementations of set operations such as in, union and intersect. Elements in a Set are unique, as determined by the elements' definition of isequal. The order of elements in a Set is an implementation detail and cannot be relied on.

See also: AbstractSet, BitSet, Dict, push!, empty!, union!, in, isequal

Examples

```
julia> s = Set("aaBca")
Set{Char} with 3 elements:
  'a'
  'c'
  'B'

julia> push!(s, 'b')
Set{Char} with 4 elements:
  'a'
  'b'
  'B'
  'c'
```

```
julia> s = Set([NaN, 0.0, 1.0, 2.0]);

julia> -0.0 in s # isequal(0.0, -0.0) is false
false

julia> NaN in s # isequal(NaN, NaN) is true
true
```

source

Base.BitSet – Type.

```
BitSet([itr])
```

Construct a sorted set of Ints generated by the given iterable object, or an empty set. Implemented as a bit string, and therefore designed for dense integer sets. If the set will be sparse (for example, holding a few very large integers), use Set instead.

source

Base.union – Function.

```
union(s, itrs...)
∪(s, itrs...)
```

Construct an object containing all distinct elements from all of the arguments.

The first argument controls what kind of container is returned. If this is an array, it maintains the order in which elements first appear.

Unicode ∪ can be typed by writing \cup then pressing tab in the Julia REPL, and in many editors. This is an infix operator, allowing s ∪ itr.

See also unique, intersect, isdisjoint, vcat, Iterators.flatten.

Examples

```
julia> union([1, 2], [3])
3-element Vector{Int64}:
 1
 2
 3

julia> union([4 2 3 4 4], 1:3, 3.0)
4-element Vector{Float64}:
 4.0
 2.0
 3.0
 1.0

julia> (0, 0.0) ∪ (-0.0, NaN)
```

```
3-element Vector{Real}:
   0
  -0.0
 NaN

julia> union(Set([1, 2]), 2:3)
Set{Int64} with 3 elements:
  2
  3
  1
```

source

Base.union! – Function.

```
union!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Construct the union of passed in sets and overwrite s with the result. Maintain order with arrays.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> a = Set([3, 4, 5]);

julia> union!(a, 1:2:7);

julia> a
Set{Int64} with 5 elements:
  5
  4
  7
  3
  1
```

source

Base.intersect – Function.

```
intersect(s, itrs...)
∩(s, itrs...)
```

Construct the set containing those elements which appear in all of the arguments.

The first argument controls what kind of container is returned. If this is an array, it maintains the order in which elements first appear.

Unicode ∩ can be typed by writing \cap then pressing tab in the Julia REPL, and in many editors. This is an infix operator, allowing `s ∩ itr`.

See also `setdiff`, `isdisjoint`, `issubset`, `issetequal`.

> **Julia 1.8**
>
> As of Julia 1.8 intersect returns a result with the eltype of the type-promoted eltypes of the two inputs

Examples

```
julia> intersect([1, 2, 3], [3, 4, 5])
1-element Vector{Int64}:
 3

julia> intersect([1, 4, 4, 5, 6], [6, 4, 6, 7, 8])
2-element Vector{Int64}:
 4
 6

julia> intersect(1:16, 7:99)
7:16

julia> (0, 0.0) ∩ (-0.0, 0)
1-element Vector{Real}:
 0

julia> intersect(Set([1, 2]), BitSet([2, 3]), 1.0:10.0)
Set{Float64} with 1 element:
  2.0
```

source

Base.setdiff – Function.

```
setdiff(s, itrs...)
```

Construct the set of elements in `s` but not in any of the iterables in `itrs`. Maintain order with arrays.

See also `setdiff!`, `union` and `intersect`.

Examples

```
julia> setdiff([1,2,3], [3,4,5])
2-element Vector{Int64}:
 1
 2
```

source

Base.setdiff! – Function.

```
setdiff!(s, itrs...)
```

Remove from set s (in-place) each element of each iterable from `itrs`. Maintain order with arrays.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> a = Set([1, 3, 4, 5]);

julia> setdiff!(a, 1:2:6);

julia> a
Set{Int64} with 1 element:
  4
```

source

`Base.symdiff` – Function.

```
symdiff(s, itrs...)
```

Construct the symmetric difference of elements in the passed in sets. When s is not an `AbstractSet`, the order is maintained.

See also `symdiff!`, `setdiff`, `union` and `intersect`.

Examples

```
julia> symdiff([1,2,3], [3,4,5], [4,5,6])
3-element Vector{Int64}:
 1
 2
 6

julia> symdiff([1,2,1], [2, 1, 2])
Int64[]
```

source

`Base.symdiff!` – Function.

```
symdiff!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Construct the symmetric difference of the passed in sets, and overwrite s with the result. When s is an array, the order is maintained. Note that in this case the multiplicity of elements matters.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

Base.intersect! – Function.

```
intersect!(s::Union{AbstractSet,AbstractVector}, itrs...)
```

Intersect all passed in sets and overwrite s with the result. Maintain order with arrays.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

Base.issubset – Function.

```
issubset(a, b) -> Bool
⊆(a, b) -> Bool
⊇(b, a) -> Bool
```

Determine whether every element of a is also in b, using in.

See also ⊊, ⊈, ∩, ∪, contains.

Examples

```
julia> issubset([1, 2], [1, 2, 3])
true

julia> [1, 2, 3] ⊆ [1, 2]
false

julia> [1, 2, 3] ⊇ [1, 2]
true
```

source

Base.:⊈ – Function.

```
⊈(a, b) -> Bool
⊉(b, a) -> Bool
```

Negation of ⊆ and ⊇, i.e. checks that a is not a subset of b.

See also `issubset` (⊆), ⊊.

Examples

```
julia> (1, 2) ⊈ (2, 3)
true

julia> (1, 2) ⊈ (1, 2, 3)
false
```

source

Base.:⊊ – Function.

```
⊊(a, b) -> Bool
⊋(b, a) -> Bool
```

Determines if a is a subset of, but not equal to, b.

See also `issubset` (⊆), ⊈.

Examples

```
julia> (1, 2) ⊊ (1, 2, 3)
true

julia> (1, 2) ⊊ (1, 2)
false
```

source

Base.issetequal – Function.

```
issetequal(a, b) -> Bool
```

Determine whether a and b have the same elements. Equivalent to `a ⊆ b && b ⊆ a` but more efficient when possible.

See also: `isdisjoint`, `union`.

Examples

```
julia> issetequal([1, 2], [1, 2, 3])
false

julia> issetequal([1, 2], [2, 1])
true
```

source

Base.isdisjoint – Function.

```
isdisjoint(a, b) -> Bool
```

Determine whether the collections a and b are disjoint. Equivalent to `isempty(a ∩ b)` but more efficient when possible.

See also: `intersect`, `isempty`, `issetequal`.

> **Julia 1.5**
>
> This function requires at least Julia 1.5.

Examples

```
julia> isdisjoint([1, 2], [2, 3, 4])
false

julia> isdisjoint([3, 1], [2, 4])
true
```

source

以下类型均完全实现了上述函数：

- `BitSet`
- `Set`

以下类型仅实现了部分上述函数：

- `Array`

43.8 双端队列

Base.push! – Function.

```
push!(collection, items...) -> collection
```

Insert one or more `items` in `collection`. If `collection` is an ordered container, the items are inserted at the end (in the given order).

Examples

```
julia> push!([1, 2, 3], 4, 5, 6)
6-element Vector{Int64}:
 1
 2
```

```
 3
 4
 5
 6
```

If `collection` is ordered, use `append!` to add all the elements of another collection to it. The result of the preceding example is equivalent to `append!([1, 2, 3], [4, 5, 6])`. For `AbstractSet` objects, `union!` can be used instead.

See `sizehint!` for notes about the performance model.

See also `pushfirst!`.

source

`Base.pop!` – Function.

```
pop!(collection) -> item
```

Remove an item in `collection` and return it. If `collection` is an ordered container, the last item is returned; for unordered containers, an arbitrary element is returned.

See also: `popfirst!`, `popat!`, `delete!`, `deleteat!`, `splice!`, and `push!`.

Examples

```
julia> A=[1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> pop!(A)
3

julia> A
2-element Vector{Int64}:
 1
 2

julia> S = Set([1, 2])
Set{Int64} with 2 elements:
  2
  1

julia> pop!(S)
2

julia> S
Set{Int64} with 1 element:
  1

julia> pop!(Dict(1=>2))
1 => 2
```

source

```
pop!(collection, key[, default])
```

Delete and return the mapping for key if it exists in `collection`, otherwise return `default`, or throw an error if `default` is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);

julia> pop!(d, "a")
1

julia> pop!(d, "d")
ERROR: KeyError: key "d" not found
Stacktrace:
[...]

julia> pop!(d, "e", 4)
4
```

source

`Base.popat!` – Function.

```
popat!(a::Vector, i::Integer, [default])
```

Remove the item at the given `i` and return it. Subsequent items are shifted to fill the resulting gap. When `i` is not a valid index for `a`, return `default`, or throw an error if `default` is not specified.

See also: `pop!`, `popfirst!`, `deleteat!`, `splice!`.

Julia 1.5

This function is available as of Julia 1.5.

Examples

```
julia> a = [4, 3, 2, 1]; popat!(a, 2)
3

julia> a
3-element Vector{Int64}:
 4
 2
 1

julia> popat!(a, 4, missing)
missing
```

```
julia> popat!(a, 4)
ERROR: BoundsError: attempt to access 3-element Vector{Int64} at index [4]
[...]
```

source

Base.pushfirst! – Function.

```
pushfirst!(collection, items...) -> collection
```

Insert one or more items at the beginning of collection.

This function is called unshift in many other programming languages.

Examples

```
julia> pushfirst!([1, 2, 3, 4], 5, 6)
6-element Vector{Int64}:
 5
 6
 1
 2
 3
 4
```

source

Base.popfirst! – Function.

```
popfirst!(collection) -> item
```

Remove the first item from collection.

This function is called shift in many other programming languages.

See also: pop!, popat!, delete!.

Examples

```
julia> A = [1, 2, 3, 4, 5, 6]
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6

julia> popfirst!(A)
1
```

```
julia> A
5-element Vector{Int64}:
 2
 3
 4
 5
 6
```

source

Base.insert! – Function.

```
insert!(a::Vector, index::Integer, item)
```

Insert an item into a at the given index. index is the index of item in the resulting a.

See also: push!, replace, popat!, splice!.

Examples

```
julia> insert!(Any[1:6;], 3, "here")
7-element Vector{Any}:
 1
 2
  "here"
 3
 4
 5
 6
```

source

Base.deleteat! – Function.

```
deleteat!(a::Vector, i::Integer)
```

Remove the item at the given i and return the modified a. Subsequent items are shifted to fill the resulting gap.

See also: keepat!, delete!, popat!, splice!.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 2)
5-element Vector{Int64}:
 6
 4
 3
 2
 1
```

source

```
deleteat!(a::Vector, inds)
```

Remove the items at the indices given by `inds`, and return the modified `a`. Subsequent items are shifted to fill the resulting gap.

`inds` can be either an iterator or a collection of sorted and unique integer indices, or a boolean vector of the same length as `a` with `true` indicating entries to delete.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 1:2:5)
3-element Vector{Int64}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], [true, false, true, false, true, false])
3-element Vector{Int64}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], (2, 2))
ERROR: ArgumentError: indices must be unique and sorted
Stacktrace:
[...]
```

source

`Base.keepat!` – Function.

```
keepat!(a::Vector, inds)
keepat!(a::BitVector, inds)
```

Remove the items at all the indices which are not given by `inds`, and return the modified `a`. Items which are kept are shifted to fill the resulting gaps.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

`inds` must be an iterator of sorted and unique integer indices. See also `deleteat!`.

Julia 1.7

This function is available as of Julia 1.7.

Examples

```
julia> keepat!([6, 5, 4, 3, 2, 1], 1:2:5)
3-element Vector{Int64}:
 6
 4
 2
```

source

```
keepat!(a::Vector, m::AbstractVector{Bool})
keepat!(a::BitVector, m::AbstractVector{Bool})
```

The in-place version of logical indexing `a = a[m]`. That is, `keepat!(a, m)` on vectors of equal length a and `m` will remove all elements from a for which `m` at the corresponding index is `false`.

Examples

```
julia> a = [:a, :b, :c];

julia> keepat!(a, [true, false, true])
2-element Vector{Symbol}:
 :a
 :c

julia> a
2-element Vector{Symbol}:
 :a
 :c
```

source

`Base.splice!` – Function.

```
splice!(a::Vector, index::Integer, [replacement]) -> item
```

Remove the item at the given index, and return the removed item. Subsequent items are shifted left to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed item.

See also: `replace`, `delete!`, `deleteat!`, `pop!`, `popat!`.

Examples

```
julia> A = [6, 5, 4, 3, 2, 1]; splice!(A, 5)
2

julia> A
5-element Vector{Int64}:
 6
 5
 4
 3
```

```
 1

julia> splice!(A, 5, -1)
1

julia> A
5-element Vector{Int64}:
  6
  5
  4
  3
 -1

julia> splice!(A, 1, [-1, -2, -3])
6

julia> A
7-element Vector{Int64}:
 -1
 -2
 -3
  5
  4
  3
 -1
```

To insert `replacement` before an index `n` without removing any items, use `splice!(collection, n:n-1, replacement)`.

source

```
splice!(a::Vector, indices, [replacement]) -> items
```

Remove items at specified indices, and return a collection containing the removed items. Subsequent items are shifted left to fill the resulting gaps. If specified, replacement values from an ordered collection will be spliced in place of the removed items; in this case, `indices` must be a `AbstractUnitRange`.

To insert `replacement` before an index `n` without removing any items, use `splice!(collection, n:n-1, replacement)`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Julia 1.5

Prior to Julia 1.5, `indices` must always be a `UnitRange`.

Julia 1.8

Prior to Julia 1.8, `indices` must be a `UnitRange` if splicing in replacement values.

Examples

```
julia> A = [-1, -2, -3, 5, 4, 3, -1]; splice!(A, 4:3, 2)
Int64[]

julia> A
8-element Vector{Int64}:
 -1
 -2
 -3
  2
  5
  4
  3
 -1
```

source

Base.resize! – Function.

```
resize!(a::Vector, n::Integer) -> Vector
```

Resize a to contain n elements. If n is smaller than the current collection length, the first n elements will be retained. If n is larger, the new elements are not guaranteed to be initialized.

Examples

```
julia> resize!([6, 5, 4, 3, 2, 1], 3)
3-element Vector{Int64}:
 6
 5
 4

julia> a = resize!([6, 5, 4, 3, 2, 1], 8);

julia> length(a)
8

julia> a[1:6]
6-element Vector{Int64}:
 6
 5
 4
 3
 2
 1
```

source

Base.append! – Function.

```
append!(collection, collections...) -> collection.
```

For an ordered container `collection`, add the elements of each `collections` to the end of it.

> **Julia 1.6**
>
> Specifying multiple collections to be appended requires at least Julia 1.6.

Examples

```
julia> append!([1], [2, 3])
3-element Vector{Int64}:
 1
 2
 3

julia> append!([1, 2, 3], [4, 5], [6])
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
```

Use `push!` to add individual items to `collection` which are not already themselves in another collection. The result of the preceding example is equivalent to `push!([1, 2, 3], 4, 5, 6)`.

See `sizehint!` for notes about the performance model.

See also `vcat` for vectors, `union!` for sets, and `prepend!` and `pushfirst!` for the opposite order.

source

`Base.prepend!` – Function.

```
prepend!(a::Vector, collections...) -> collection
```

Insert the elements of each `collections` to the beginning of `a`.

When `collections` specifies multiple collections, order is maintained: elements of `collections[1]` will appear leftmost in `a`, and so on.

> **Julia 1.6**
>
> Specifying multiple collections to be prepended requires at least Julia 1.6.

Examples

```
julia> prepend!([3], [1, 2])
3-element Vector{Int64}:
 1
 2
 3

julia> prepend!([6], [1, 2], [3, 4, 5])
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
```

source

以下类型均完全实现了上述函数：

- Vector (a.k.a. 1-dimensional Array)
- BitVector (a.k.a. 1-dimensional BitArray)

43.9 集合相关的实用工具

Core.Pair – Type.

```
Pair(x, y)
x => y
```

Construct a Pair object with type Pair{typeof(x), typeof(y)}. The elements are stored in the fields first and second. They can also be accessed via iteration (but a Pair is treated as a single "scalar" for broadcasting operations).

See also Dict.

Examples

```
julia> p = "foo" => 7
"foo" => 7

julia> typeof(p)
Pair{String, Int64}

julia> p.first
"foo"

julia> for x in p
           println(x)
       end
foo
7
```

```
julia> replace.(["xops", "oxps"], "x" => "o")
2-element Vector{String}:
 "oops"
 "oops"
```

source

Base.Pairs – Type.

```
Iterators.Pairs(values, keys) <: AbstractDict{eltype(keys), eltype(values)}
```

Transforms an indexable container into a Dictionary-view of the same data. Modifying the key-space of the underlying data may invalidate this object.

source

Chapter 44

数学相关

44.1 数学运算符

`Base.:-` – Method.

```
-(x)
```

Unary minus operator.

See also: `abs`, `flipsign`.

Examples

```
julia> -1
-1

julia> -(2)
-2

julia> -[1 2; 3 4]
2×2 Matrix{Int64}:
 -1  -2
 -3  -4
```

source

`Base.:+` – Function.

```
dt::Date + t::Time -> DateTime
```

The addition of a `Date` with a `Time` produces a `DateTime`. The hour, minute, second, and millisecond parts of the `Time` are used along with the year, month, and day of the `Date` to create the new `DateTime`. Non-zero microseconds or nanoseconds in the `Time` type will result in an `InexactError` being thrown.

```
+(x, y...)
```

Addition operator. `x+y+z+...` calls this function with all arguments, i.e. `+(x, y, z, ...)`.

Examples

```
julia> 1 + 20 + 4
25

julia> +(1, 20, 4)
25
```

source

Base.:- – Method.

```
-(x, y)
```

Subtraction operator.

Examples

```
julia> 2 - 3
-1

julia> -(2, 4.5)
-2.5
```

source

Base.:* – Method.

```
*(x, y...)
```

Multiplication operator. `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`.

Examples

```
julia> 2 * 7 * 8
112

julia> *(2, 7, 8)
112
```

source

Base.:/ – Function.

```
/(x, y)
```

Right division operator: multiplication of x by the inverse of y on the right. Gives floating-point results for integer arguments.

Examples

```
julia> 1/2
0.5

julia> 4/2
2.0

julia> 4.5/2
2.25
```

source

```
A / B
```

Matrix right-division: `A / B` is equivalent to `(B' \ A')'` where `\` is the left-division operator. For square matrices, the result X is such that `A == X*B`.

See also: `rdiv!`.

Examples

```
julia> A = Float64[1 4 5; 3 9 2]; B = Float64[1 4 2; 3 4 2; 8 7 1];

julia> X = A / B
2×3 Matrix{Float64}:
 -0.65   3.75  -1.2
  3.25  -2.75   1.0

julia> isapprox(A, X*B)
true

julia> isapprox(X, A*pinv(B))
true
```

`Base.:\` – Method.

```
\(x, y)
```

Left division operator: multiplication of y by the inverse of x on the left. Gives floating-point results for integer arguments.

Examples

```
julia> 3 \ 6
2.0

julia> inv(3) * 6
```

```
2.0

julia> A = [4 3; 2 1]; x = [5, 6];

julia> A \ x
2-element Vector{Float64}:
  6.5
 -7.0

julia> inv(A) * x
2-element Vector{Float64}:
  6.5
 -7.0
```

source

Base.:^ – Method.

```
^(x, y)
```

Exponentiation operator. If x is a matrix, computes matrix exponentiation.

If y is an Int literal (e.g. 2 in x^2 or -3 in x^-3), the Julia code x^y is transformed by the compiler to Base.literal_pow(^, x, Val(y)), to enable compile-time specialization on the value of the exponent. (As a default fallback we have Base.literal_pow(^, x, Val(y)) = ^(x,y), where usually ^ == Base.^ unless ^ has been defined in the calling namespace.) If y is a negative integer literal, then Base.literal_pow transforms the operation to inv(x)^-y by default, where -y is positive.

Examples

```
julia> 3^5
243

julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> A^3
2×2 Matrix{Int64}:
 37   54
 81  118
```

source

Base.fma – Function.

```
fma(x, y, z)
```

Computes x*y+z without rounding the intermediate result x*y. On some systems this is significantly more expensive than x*y+z. fma is used to improve accuracy in certain algorithms. See muladd.

source

Base.muladd – Function.

```
muladd(x, y, z)
```

Combined multiply-add: computes x*y+z, but allowing the add and multiply to be merged with each other or with surrounding operations for performance. For example, this may be implemented as an fma if the hardware supports it efficiently. The result can be different on different machines and can also be different on the same machine due to constant propagation or other optimizations. See fma.

Examples

```
julia> muladd(3, 2, 1)
7

julia> 3 * 2 + 1
7
```

source

```
muladd(A, y, z)
```

Combined multiply-add, A*y .+ z, for matrix-matrix or matrix-vector multiplication. The result is always the same size as A*y, but z may be smaller, or a scalar.

Julia 1.6

These methods require Julia 1.6 or later.

Examples

```
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; z=[0, 100];

julia> muladd(A, B, z)
2×2 Matrix{Float64}:
   3.0    3.0
 107.0  107.0
```

Base.inv – Method.

```
inv(x)
```

Return the multiplicative inverse of x, such that x*inv(x) or inv(x)*x yields one(x) (the multiplicative identity) up to roundoff errors.

If x is a number, this is essentially the same as one(x)/x, but for some types inv(x) may be slightly more efficient.

Examples

```
julia> inv(2)
0.5

julia> inv(1 + 2im)
0.2 - 0.4im

julia> inv(1 + 2im) * (1 + 2im)
1.0 + 0.0im

julia> inv(2//3)
3//2
```

Julia 1.2

`inv(::Missing)` requires at least Julia 1.2.

source

Base.div – Function.

```
div(x, y)
÷(x, y)
```

The quotient from Euclidean (integer) division. Generally equivalent to a mathematical operation x/y without a fractional part.

See also: `cld`, `fld`, `rem`, `divrem`.

Examples

```
julia> 9 ÷ 4
2

julia> -5 ÷ 3
-1

julia> 5.0 ÷ 2
2.0

julia> div.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 -1  -1  -1  0  0  0  0  0  1  1  1
```

source

Base.fld – Function.

```
fld(x, y)
```

Largest integer less than or equal to `x / y`. Equivalent to `div(x, y, RoundDown)`.

See also div, cld, fld1.

Examples

```
julia> fld(7.3, 5.5)
1.0

julia> fld.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 -2  -2  -1  -1  -1  0  0  0  1  1  1
```

Because `fld(x, y)` implements strictly correct floored rounding based on the true value of floating-point numbers, unintuitive situations can arise. For example:

```
julia> fld(6.0, 0.1)
59.0
julia> 6.0 / 0.1
60.0
julia> 6.0 / big(0.1)
59.99999999999999666933092612453056361837965690217069245739573412231113406246995
```

What is happening here is that the true value of the floating-point number written as `0.1` is slightly larger than the numerical value 1/10 while `6.0` represents the number 6 precisely. Therefore the true value of `6.0 / 0.1` is slightly less than 60. When doing division, this is rounded to precisely `60.0`, but `fld(6.0, 0.1)` always takes the floor of the true value, so the result is `59.0`.

source

`Base.cld` – Function.

```
cld(x, y)
```

Smallest integer larger than or equal to `x / y`. Equivalent to `div(x, y, RoundUp)`.

See also div, fld.

Examples

```
julia> cld(5.5, 2.2)
3.0

julia> cld.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 -1  -1  -1  0  0  0  1  1  1  2  2
```

source

`Base.mod` – Function.

```
mod(x::Integer, r::AbstractUnitRange)
```

Find y in the range r such that $x\ y(mod n)$, where n = length(r), i.e. y = mod(x - first(r), n) + first(r).

See also mod1.

Examples

```
julia> mod(0, Base.OneTo(3))  # mod1(0, 3)
3

julia> mod(3, 0:2)  # mod(3, 3)
0
```

Julia 1.3

This method requires at least Julia 1.3.

source

```
mod(x, y)
rem(x, y, RoundDown)
```

The reduction of x modulo y, or equivalently, the remainder of x after floored division by y, i.e. x - y*fld(x,y) if computed without intermediate rounding.

The result will have the same sign as y, and magnitude less than abs(y) (with some exceptions, see note below).

Note

When used with floating point values, the exact result may not be representable by the type, and so rounding error may occur. In particular, if the exact result is very close to y, then it may be rounded to y.

See also: rem, div, fld, mod1, invmod.

```
julia> mod(8, 3)
2

julia> mod(9, 3)
0

julia> mod(8.9, 3)
2.9000000000000004

julia> mod(eps(), 3)
2.220446049250313e-16

julia> mod(-eps(), 3)
3.0

julia> mod.(-5:5, 3)'
```

```
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 1  2  0  1  2  0  1  2  0  1  2
```

source

```
rem(x::Integer, T::Type{<:Integer}) -> T
mod(x::Integer, T::Type{<:Integer}) -> T
%(x::Integer, T::Type{<:Integer}) -> T
```

Find `y::T` such that x ≡ y (mod n), where n is the number of integers representable in T, and y is an integer in `[typemin(T),typemax(T)]`. If T can represent any integer (e.g. `T == BigInt`), then this operation corresponds to a conversion to T.

Examples

```
julia> x = 129 % Int8
-127

julia> typeof(x)
Int8

julia> x = 129 % BigInt
129

julia> typeof(x)
BigInt
```

source

`Base.rem` – Function.

```
rem(x, y)
%(x, y)
```

Remainder from Euclidean division, returning a value of the same sign as x, and smaller in magnitude than y. This value is always exact.

See also: `div`, `mod`, `mod1`, `divrem`.

Examples

```
julia> x = 15; y = 4;

julia> x % y
3

julia> x == div(x, y) * y + rem(x, y)
true

julia> rem.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 -2  -1  0  -2  -1  0  1  2  0  1  2
```

source

`Base.Math.rem2pi` – Function.

```
rem2pi(x, r::RoundingMode)
```

Compute the remainder of x after integer division by 2π, with the quotient rounded according to the rounding mode `r`. In other words, the quantity

```
x - 2π*round(x/(2π),r)
```

without any intermediate rounding. This internally uses a high precision approximation of 2π, and so will give a more accurate result than `rem(x,2π,r)`

- if `r == RoundNearest`, then the result is in the interval $[-,]$. This will generally be the most accurate result. See also `RoundNearest`.
- if `r == RoundToZero`, then the result is in the interval $[0,2]$ if x is positive,. or $[-2,0]$ otherwise. See also `RoundToZero`.
- if `r == RoundDown`, then the result is in the interval $[0,2]$. See also `RoundDown`.
- if `r == RoundUp`, then the result is in the interval $[-2,0]$. See also `RoundUp`.

Examples

```
julia> rem2pi(7pi/4, RoundNearest)
-0.7853981633974485

julia> rem2pi(7pi/4, RoundDown)
5.497787143782138
```

source

`Base.Math.mod2pi` – Function.

```
mod2pi(x)
```

Modulus after division by 2π, returning in the range $[0,2)$.

This function computes a floating point representation of the modulus after division by numerically exact 2π, and is therefore not exactly the same as `mod(x,2π)`, which would compute the modulus of x relative to division by the floating-point number 2π.

Note

Depending on the format of the input value, the closest representable value to 2π may be less than 2π. For example, the expression `mod2pi(2π)` will not return `0`, because the intermediate value of `2*π` is a `Float64` and `2*Float64(π) < 2*big(π)`. See `rem2pi` for more refined control of this behavior.

Examples

```
julia> mod2pi(9*pi/4)
0.7853981633974481
```

source

Base.divrem – Function.

```
divrem(x, y, r::RoundingMode=RoundToZero)
```

The quotient and remainder from Euclidean division. Equivalent to (div(x, y, r), rem(x, y, r)). Equivalently, with the default value of r, this call is equivalent to (x ÷ y, x % y).

See also: fldmod, cld.

Examples

```
julia> divrem(3, 7)
(0, 3)

julia> divrem(7, 3)
(2, 1)
```

source

Base.fldmod – Function.

```
fldmod(x, y)
```

The floored quotient and modulus after division. A convenience wrapper for divrem(x, y, RoundDown). Equivalent to (fld(x, y), mod(x, y)).

See also: fld, cld, fldmod1.

source

Base.fld1 – Function.

```
fld1(x, y)
```

Flooring division, returning a value consistent with mod1(x,y)

See also mod1, fldmod1.

Examples

```
julia> x = 15; y = 4;

julia> fld1(x, y)
4
```

```
julia> x == fld(x, y) * y + mod(x, y)
true

julia> x == (fld1(x, y) - 1) * y + mod1(x, y)
true
```

source

Base.mod1 – Function.

```
mod1(x, y)
```

Modulus after flooring division, returning a value r such that mod(r, y) == mod(x, y) in the range $(0, y]$ for positive y and in the range $[y, 0)$ for negative y.

With integer arguments and positive y, this is equal to mod(x, 1:y), and hence natural for 1-based indexing. By comparison, mod(x, y) == mod(x, 0:y-1) is natural for computations with offsets or strides.

See also mod, fld1, fldmod1.

Examples

```
julia> mod1(4, 2)
2

julia> mod1.(-5:5, 3)'
1×11 adjoint(::Vector{Int64}) with eltype Int64:
 1  2  3  1  2  3  1  2  3  1  2

julia> mod1.([-0.1, 0, 0.1, 1, 2, 2.9, 3, 3.1]', 3)
1×8 Matrix{Float64}:
 2.9  3.0  0.1  1.0  2.0  2.9  3.0  0.1
```

source

Base.fldmod1 – Function.

```
fldmod1(x, y)
```

Return (fld1(x,y), mod1(x,y)).

See also fld1, mod1.

source

Base.:// – Function.

```
//(num, den)
```

Divide two integers or rational numbers, giving a `Rational` result.

Examples

```
julia> 3 // 5
3//5

julia> (3 // 5) // (2 // 1)
3//10
```

source

`Base.rationalize` – Function.

```
rationalize([T<:Integer=Int,] x; tol::Real=eps(x))
```

Approximate floating point number `x` as a `Rational` number with components of the given integer type. The result will differ from `x` by no more than `tol`.

Examples

```
julia> rationalize(5.6)
28//5

julia> a = rationalize(BigInt, 10.3)
103//10

julia> typeof(numerator(a))
BigInt
```

source

`Base.numerator` – Function.

```
numerator(x)
```

Numerator of the rational representation of `x`.

Examples

```
julia> numerator(2//3)
2

julia> numerator(4)
4
```

source

`Base.denominator` – Function.

```
denominator(x)
```

Denominator of the rational representation of x.

Examples

```
julia> denominator(2//3)
3

julia> denominator(4)
1
```

source

Base.:<< – Function.

```
<<(x, n)
```

Left bit shift operator, x << n. For n >= 0, the result is x shifted left by n bits, filling with 0s. This is equivalent to x * 2^n. For n < 0, this is equivalent to x >> -n.

Examples

```
julia> Int8(3) << 2
12

julia> bitstring(Int8(3))
"00000011"

julia> bitstring(Int8(12))
"00001100"
```

See also >>, >>>, exp2, ldexp.

source

```
<<(B::BitVector, n) -> BitVector
```

Left bit shift operator, B << n. For n >= 0, the result is B with elements shifted n positions backwards, filling with false values. If n < 0, elements are shifted forwards. Equivalent to B >> -n.

Examples

```
julia> B = BitVector([true, false, true, false, false])
5-element BitVector:
 1
 0
 1
 0
```

```
 0

julia> B << 1
5-element BitVector:
 0
 1
 0
 0
 0

julia> B << -1
5-element BitVector:
 0
 1
 0
 1
 0
```

source

Base.:>> – Function.

```
>>(x, n)
```

Right bit shift operator, `x >> n`. For `n >= 0`, the result is `x` shifted right by `n` bits, filling with `0`s if `x >= 0`, `1`s if `x < 0`, preserving the sign of `x`. This is equivalent to `fld(x, 2^n)`. For `n < 0`, this is equivalent to `x << -n`.

Examples

```
julia> Int8(13) >> 2
3

julia> bitstring(Int8(13))
"00001101"

julia> bitstring(Int8(3))
"00000011"

julia> Int8(-14) >> 2
-4

julia> bitstring(Int8(-14))
"11110010"

julia> bitstring(Int8(-4))
"11111100"
```

See also >>>, <<.

source

```
>>(B::BitVector, n) -> BitVector
```

Right bit shift operator, `B >> n`. For `n >= 0`, the result is `B` with elements shifted `n` positions forward, filling with `false` values. If `n < 0`, elements are shifted backwards. Equivalent to `B << -n`.

Examples

```
julia> B = BitVector([true, false, true, false, false])
5-element BitVector:
 1
 0
 1
 0
 0

julia> B >> 1
5-element BitVector:
 0
 1
 0
 1
 0

julia> B >> -1
5-element BitVector:
 0
 1
 0
 0
 0
```

source

`Base.:>>>` – Function.

```
>>>(x, n)
```

Unsigned right bit shift operator, `x >>> n`. For `n >= 0`, the result is `x` shifted right by `n` bits, filling with `0`s. For `n < 0`, this is equivalent to `x << -n`.

For `Unsigned` integer types, this is equivalent to `>>`. For `Signed` integer types, this is equivalent to `signed(unsigned(x) >> n)`.

Examples

```
julia> Int8(-14) >>> 2
60

julia> bitstring(Int8(-14))
"11110010"

julia> bitstring(Int8(60))
"00111100"
```

BigInts are treated as if having infinite size, so no filling is required and this is equivalent to >>.

See also >>, <<.

source

```
>>>(B::BitVector, n) -> BitVector
```

Unsigned right bitshift operator, `B >>> n`. Equivalent to `B >> n`. See >> for details and examples.

source

`Base.bitrotate` – Function.

```
bitrotate(x::Base.BitInteger, k::Integer)
```

`bitrotate(x, k)` implements bitwise rotation. It returns the value of x with its bits rotated left k times. A negative value of k will rotate to the right instead.

Julia 1.5

This function requires Julia 1.5 or later.

See also: <<, `circshift`, `BitArray`.

```
julia> bitrotate(UInt8(114), 2)
0xc9

julia> bitstring(bitrotate(0b01110010, 2))
"11001001"

julia> bitstring(bitrotate(0b01110010, -2))
"10011100"

julia> bitstring(bitrotate(0b01110010, 8))
"01110010"
```

source

`Base.::` – Function.

```
:expr
```

Quote an expression `expr`, returning the abstract syntax tree (AST) of `expr`. The AST may be of type `Expr`, `Symbol`, or a literal value. The syntax `:identifier` evaluates to a `Symbol`.

See also: `Expr`, `Symbol`, `Meta.parse`

Examples

```
julia> expr = :(a = b + 2*x)
:(a = b + 2x)

julia> sym = :some_identifier
:some_identifier

julia> value = :0xff
0xff

julia> typeof((expr, sym, value))
Tuple{Expr, Symbol, UInt8}
```

source

Base.range – Function.

```
range(start, stop, length)
range(start, stop; length, step)
range(start; length, stop, step)
range(;start, length, stop, step)
```

Construct a specialized array with evenly spaced elements and optimized storage (an `AbstractRange`) from the arguments. Mathematically a range is uniquely determined by any three of `start`, `step`, `stop` and `length`. Valid invocations of range are:

- Call range with any three of `start`, `step`, `stop`, `length`.
- Call range with two of `start`, `stop`, `length`. In this case `step` will be assumed to be one. If both arguments are Integers, a `UnitRange` will be returned.
- Call range with one of `stop` or `length`. `start` and `step` will be assumed to be one.

See Extended Help for additional details on the returned type.

Examples

```
julia> range(1, length=100)
1:100

julia> range(1, stop=100)
1:100

julia> range(1, step=5, length=100)
1:5:496

julia> range(1, step=5, stop=100)
1:5:96

julia> range(1, 10, length=101)
1.0:0.09:10.0

julia> range(1, 100, step=5)
1:5:96
```

```
julia> range(stop=10, length=5)
6:10

julia> range(stop=10, step=1, length=5)
6:1:10

julia> range(start=1, step=1, stop=10)
1:1:10

julia> range(; length = 10)
Base.OneTo(10)

julia> range(; stop = 6)
Base.OneTo(6)

julia> range(; stop = 6.5)
1.0:1.0:6.0
```

If `length` is not specified and `stop - start` is not an integer multiple of `step`, a range that ends before `stop` will be produced.

```
julia> range(1, 3.5, step=2)
1.0:2.0:3.0
```

Special care is taken to ensure intermediate values are computed rationally. To avoid this induced overhead, see the `LinRange` constructor.

> **Julia 1.1**
>
> `stop` as a positional argument requires at least Julia 1.1.

> **Julia 1.7**
>
> The versions without keyword arguments and `start` as a keyword argument require at least Julia 1.7.

> **Julia 1.8**
>
> The versions with `stop` as a sole keyword argument, or `length` as a sole keyword argument require at least Julia 1.8.

Extended Help

range will produce a `Base.OneTo` when the arguments are Integers and

- Only `length` is provided
- Only `stop` is provided

range will produce a `UnitRange` when the arguments are Integers and

- Only `start` and `stop` are provided
- Only `length` and `stop` are provided

A `UnitRange` is not produced if `step` is provided even if specified as one.

source

`Base.OneTo` – Type.

```
Base.OneTo(n)
```

Define an `AbstractUnitRange` that behaves like `1:n`, with the added distinction that the lower limit is guaranteed (by the type system) to be 1.

source

`Base.StepRangeLen` – Type.

```
StepRangeLen(         ref::R, step::S, len, [offset=1]) where {  R,S}
StepRangeLen{T,R,S}(  ref::R, step::S, len, [offset=1]) where {T,R,S}
StepRangeLen{T,R,S,L}(ref::R, step::S, len, [offset=1]) where {T,R,S,L}
```

A range `r` where `r[i]` produces values of type `T` (in the first form, `T` is deduced automatically), parameterized by a `ref`erence value, a `step`, and the `len`gth. By default `ref` is the starting value `r[1]`, but alternatively you can supply it as the value of `r[offset]` for some other index `1 <= offset <= len`. The syntax `a:b` or `a:b:c`, where any of `a`, `b`, or `c` are floating-point numbers, creates a `StepRangeLen`.

Julia 1.7

The 4th type parameter `L` requires at least Julia 1.7.

source

`Base.:==` – Function.

```
==(x, y)
```

Generic equality operator. Falls back to `===`. Should be implemented for all types with a notion of equality, based on the abstract value that an instance represents. For example, all numeric types are compared by numeric value, ignoring type. Strings are compared as sequences of characters, ignoring encoding. For collections, `==` is generally called recursively on all contents, though other properties (like the shape for arrays) may also be taken into account.

This operator follows IEEE semantics for floating-point numbers: `0.0 == -0.0` and `NaN != NaN`.

The result is of type `Bool`, except when one of the operands is `missing`, in which case `missing` is returned (three-valued logic). For collections, `missing` is returned if at least one of the operands contains a `missing` value and all non-missing values are equal. Use `isequal` or `===` to always get a `Bool` result.

Implementation

New numeric types should implement this function for two arguments of the new type, and handle comparison to other types via promotion rules where possible.

`isequal` falls back to `==`, so new methods of `==` will be used by the `Dict` type to compare keys. If your type will be used as a dictionary key, it should therefore also implement `hash`.

If some type defines `==`, `isequal`, and `isless` then it should also implement `<` to ensure consistency of comparisons.

source

`Base.:!=` – Function.

```
!=(x, y)
≠(x,y)
```

Not-equals comparison operator. Always gives the opposite answer as `==`.

Implementation

New types should generally not implement this, and rely on the fallback definition `!=(x,y) = !(x==y)` instead.

Examples

```
julia> 3 != 2
true

julia> "foo" ≠ "foo"
false
```

source

```
!=(x)
```

Create a function that compares its argument to `x` using `!=`, i.e. a function equivalent to `y -> y != x`. The returned function is of type `Base.Fix2{typeof(!=)}`, which can be used to implement specialized methods.

> **Julia 1.2**
>
> This functionality requires at least Julia 1.2.

source

`Base.:!==` – Function.

```
!==(x, y)
≢(x,y)
```

Always gives the opposite answer as `===`.

Examples

```
julia> a = [1, 2]; b = [1, 2];

julia> a ≢ b
true

julia> a ≢ a
false
```

source

Base.:< – Function.

```
<(x, y)
```

Less-than comparison operator. Falls back to `isless`. Because of the behavior of floating-point NaN values, this operator implements a partial order.

Implementation

New types with a canonical partial order should implement this function for two arguments of the new type. Types with a canonical total order should implement `isless` instead.

See also `isunordered`.

Examples

```
julia> 'a' < 'b'
true

julia> "abc" < "abd"
true

julia> 5 < 3
false
```

source

```
<(x)
```

Create a function that compares its argument to `x` using `<`, i.e. a function equivalent to `y -> y < x`. The returned function is of type `Base.Fix2{typeof(<)}`, which can be used to implement specialized methods.

Julia 1.2

This functionality requires at least Julia 1.2.

source

Base.:<= – Function.

```
<=(x, y)
≤(x,y)
```

Less-than-or-equals comparison operator. Falls back to `(x < y) | (x == y)`.

Examples

```
julia> 'a' <= 'b'
true

julia> 7 ≤ 7 ≤ 9
true

julia> "abc" ≤ "abc"
true

julia> 5 <= 3
false
```

source

```
<=(x)
```

Create a function that compares its argument to `x` using `<=`, i.e. a function equivalent to `y -> y <= x`. The returned function is of type `Base.Fix2{typeof(<=)}`, which can be used to implement specialized methods.

Julia 1.2

This functionality requires at least Julia 1.2.

source

`Base.:>` – Function.

```
>(x, y)
```

Greater-than comparison operator. Falls back to `y < x`.

Implementation

Generally, new types should implement `<` instead of this function, and rely on the fallback definition `>(x, y) = y < x`.

Examples

```
julia> 'a' > 'b'
false

julia> 7 > 3 > 1
```

```
true

julia> "abc" > "abd"
false

julia> 5 > 3
true
```

source

```
>(x)
```

Create a function that compares its argument to x using >, i.e. a function equivalent to y -> y > x. The returned function is of type Base.Fix2{typeof(>)}, which can be used to implement specialized methods.

> **Julia 1.2**
>
> This functionality requires at least Julia 1.2.

source

Base.:>= – Function.

```
>=(x, y)
≥(x,y)
```

Greater-than-or-equals comparison operator. Falls back to y <= x.

Examples

```
julia> 'a' >= 'b'
false

julia> 7 ≥ 7 ≥ 3
true

julia> "abc" ≥ "abc"
true

julia> 5 >= 3
true
```

source

```
>=(x)
```

Create a function that compares its argument to x using >=, i.e. a function equivalent to y -> y >= x. The returned function is of type Base.Fix2{typeof(>=)}, which can be used to implement specialized methods.

Julia 1.2

This functionality requires at least Julia 1.2.

source

Base.cmp – Function.

```
cmp(x,y)
```

Return -1, 0, or 1 depending on whether x is less than, equal to, or greater than y, respectively. Uses the total order implemented by isless.

Examples

```
julia> cmp(1, 2)
-1

julia> cmp(2, 1)
1

julia> cmp(2+im, 3-im)
ERROR: MethodError: no method matching isless(::Complex{Int64}, ::Complex{Int64})
[...]
```

source

```
cmp(<, x, y)
```

Return -1, 0, or 1 depending on whether x is less than, equal to, or greater than y, respectively. The first argument specifies a less-than comparison function to use.

source

```
cmp(a::AbstractString, b::AbstractString) -> Int
```

Compare two strings. Return 0 if both strings have the same length and the character at each index is the same in both strings. Return -1 if a is a prefix of b, or if a comes before b in alphabetical order. Return 1 if b is a prefix of a, or if b comes before a in alphabetical order (technically, lexicographical order by Unicode code points).

Examples

```
julia> cmp("abc", "abc")
0

julia> cmp("ab", "abc")
-1

julia> cmp("abc", "ab")
```

```
1

julia> cmp("ab", "ac")
-1

julia> cmp("ac", "ab")
1

julia> cmp("α", "a")
1

julia> cmp("b", "β")
-1
```

source

Base.:~ – Function.

```
~(x)
```

Bitwise not.

See also: !, &, |.

Examples

```
julia> ~4
-5

julia> ~10
-11

julia> ~true
false
```

source

Base.:& – Function.

```
x & y
```

Bitwise and. Implements three-valued logic, returning missing if one operand is missing and the other is true. Add parentheses for function application form: (&)(x, y).

See also: |, xor, &&.

Examples

```
julia> 4 & 10
0
```

```
julia> 4 & 12
4

julia> true & missing
missing

julia> false & missing
false
```

source

Base.:| – Function.

```
x | y
```

Bitwise or. Implements three-valued logic, returning missing if one operand is missing and the other is false.

See also: &, xor, ||.

Examples

```
julia> 4 | 10
14

julia> 4 | 1
5

julia> true | missing
true

julia> false | missing
missing
```

source

Base.xor – Function.

```
xor(x, y)
⊻(x, y)
```

Bitwise exclusive or of x and y. Implements three-valued logic, returning missing if one of the arguments is missing.

The infix operation a ⊻ b is a synonym for xor(a,b), and ⊻ can be typed by tab-completing \xor or \veebar in the Julia REPL.

Examples

```
julia> xor(true, false)
true

julia> xor(true, true)
false

julia> xor(true, missing)
missing

julia> false ⊻ false
false

julia> [true; true; false] .⊻ [true; false; false]
3-element BitVector:
 0
 1
 0
```

source

Base.nand – Function.

```
nand(x, y)
⊼(x, y)
```

Bitwise nand (not and) of x and y. Implements three-valued logic, returning `missing` if one of the arguments is `missing`.

The infix operation a ⊼ b is a synonym for `nand(a,b)`, and ⊼ can be typed by tab-completing `\nand` or `\barwedge` in the Julia REPL.

Examples

```
julia> nand(true, false)
true

julia> nand(true, true)
false

julia> nand(true, missing)
missing

julia> false ⊼ false
true

julia> [true; true; false] .⊼ [true; false; false]
3-element BitVector:
 0
 1
 1
```

source

Base.nor – Function.

```
nor(x, y)
⊽(x, y)
```

Bitwise nor (not or) of x and y. Implements three-valued logic, returning `missing` if one of the arguments is `missing` and the other is not `true`.

The infix operation a ⊽ b is a synonym for `nor(a,b)`, and ⊽ can be typed by tab-completing `\nor` or `\barvee` in the Julia REPL.

Examples

```
julia> nor(true, false)
false

julia> nor(true, true)
false

julia> nor(true, missing)
false

julia> false ⊽ false
true

julia> false ⊽ missing
missing

julia> [true; true; false] .⊽ [true; false; false]
3-element BitVector:
 0
 0
 1
```

source

Base.:! – Function.

```
!(x)
```

Boolean not. Implements three-valued logic, returning `missing` if x is `missing`.

See also ~ for bitwise not.

Examples

```
julia> !true
false

julia> !false
true
```

```
julia> !missing
missing

julia> .![true false true]
1×3 BitMatrix:
 0  1  0
```

source

```
!f::Function
```

Predicate function negation: when the argument of ! is a function, it returns a composed function which computes the boolean negation of f.

See also ∘.

Examples

```
julia> str = "∀ ε > 0, ∃ δ > 0: |x-y| < δ ⇒ |f(x)-f(y)| < ε"
"∀ ε > 0, ∃ δ > 0: |x-y| < δ ⇒ |f(x)-f(y)| < ε"

julia> filter(isletter, str)
"εδxyδfxfyε"

julia> filter(!isletter, str)
"∀  > 0, ∃  > 0: |-| <  ⇒ |()-()| < "
```

Julia 1.9

Starting with Julia 1.9, !f returns a ComposedFunction instead of an anonymous function.

source

&& – Keyword.

```
x && y
```

Short-circuiting boolean AND.

See also &, the ternary operator ? :, and the manual section on control flow.

Examples

```
julia> x = 3;

julia> x > 1 && x < 10 && x isa Int
true

julia> x < 0 && error("expected positive x")
false
```

source

`||` – Keyword.

```
x || y
```

Short-circuiting boolean OR.

See also: `|`, `xor`, `&&`.

Examples

```
julia> pi < 3 || ℯ < 3
true

julia> false || true || println("neither is true!")
true
```

source

44.2 数学函数

`Base.isapprox` – Function.

```
isapprox(x, y; atol::Real=0, rtol::Real=atol>0 ? 0 : √eps, nans::Bool=false[, norm::Function])
```

Inexact equality comparison. Two numbers compare equal if their relative distance *or* their absolute distance is within tolerance bounds: `isapprox` returns `true` if `norm(x-y) <= max(atol, rtol*max(norm(x), norm(y)))`. The default `atol` (absolute tolerance) is zero and the default `rtol` (relative tolerance) depends on the types of `x` and `y`. The keyword argument `nans` determines whether or not NaN values are considered equal (defaults to false).

For real or complex floating-point values, if an `atol > 0` is not specified, `rtol` defaults to the square root of `eps` of the type of `x` or `y`, whichever is bigger (least precise). This corresponds to requiring equality of about half of the significant digits. Otherwise, e.g. for integer arguments or if an `atol > 0` is supplied, `rtol` defaults to zero.

The `norm` keyword defaults to `abs` for numeric `(x,y)` and to `LinearAlgebra.norm` for arrays (where an alternative `norm` choice is sometimes useful). When `x` and `y` are arrays, if `norm(x-y)` is not finite (i.e. `±Inf` or `NaN`), the comparison falls back to checking whether all elements of `x` and `y` are approximately equal component-wise.

The binary operator `≈` is equivalent to `isapprox` with the default arguments, and `x ≉ y` is equivalent to `!isapprox(x,y)`.

Note that `x ≈ 0` (i.e., comparing to zero with the default tolerances) is equivalent to `x == 0` since the default `atol` is `0`. In such cases, you should either supply an appropriate `atol` (or use `norm(x) ≤ atol`) or rearrange your code (e.g. use `x ≈ y` rather than `x - y ≈ 0`). It is not possible to pick a nonzero `atol` automatically because it depends on the overall scaling (the "units") of your problem: for example, in `x - y ≈ 0`, `atol=1e-9` is an absurdly small tolerance if `x` is the radius of the Earth in meters, but an absurdly large tolerance if `x` is the radius of a Hydrogen atom in meters.

> **Julia 1.6**
>
> Passing the `norm` keyword argument when comparing numeric (non-array) arguments requires Julia 1.6 or later.

Examples

```
julia> isapprox(0.1, 0.15; atol=0.05)
true

julia> isapprox(0.1, 0.15; rtol=0.34)
true

julia> isapprox(0.1, 0.15; rtol=0.33)
false

julia> 0.1 + 1e-10 ≈ 0.1
true

julia> 1e-10 ≈ 0
false

julia> isapprox(1e-10, 0, atol=1e-8)
true

julia> isapprox([10.0^9, 1.0], [10.0^9, 2.0]) # using `norm`
true
```

source

```
isapprox(x; kwargs...) / ≈(x; kwargs...)
```

Create a function that compares its argument to x using ≈, i.e. a function equivalent to `y -> y ≈ x`.

The keyword arguments supported here are the same as those in the 2-argument `isapprox`.

> **Julia 1.5**
>
> This method requires Julia 1.5 or later.

source

`Base.sin` – Method.

```
sin(x)
```

Compute sine of x, where x is in radians.

See also `sind`, `sinpi`, `sincos`, `cis`, `asin`.

Examples

```
julia> round.(sin.(range(0, 2pi, length=9)'), digits=3)
1×9 Matrix{Float64}:
 0.0  0.707  1.0  0.707  0.0  -0.707  -1.0  -0.707  -0.0

julia> sind(45)
0.7071067811865476

julia> sinpi(1/4)
0.7071067811865475

julia> round.(sincos(pi/6), digits=3)
(0.5, 0.866)

julia> round(cis(pi/6), digits=3)
0.866 + 0.5im

julia> round(exp(im*pi/6), digits=3)
0.866 + 0.5im
```

source

`Base.cos` – Method.

```
cos(x)
```

Compute cosine of x, where x is in radians.

See also `cosd`, `cospi`, `sincos`, `cis`.

source

`Base.Math.sincos` – Method.

```
sincos(x)
```

Simultaneously compute the sine and cosine of x, where x is in radians, returning a tuple `(sine, cosine)`.

See also `cis`, `sincospi`, `sincosd`.

source

`Base.tan` – Method.

```
tan(x)
```

Compute tangent of x, where x is in radians.

source

`Base.Math.sind` – Function.

```
sind(x)
```

Compute sine of x, where x is in degrees. If x is a matrix, x needs to be a square matrix.

> **Julia 1.7**
>
> Matrix arguments require Julia 1.7 or later.

source

Base.Math.cosd – Function.

```
cosd(x)
```

Compute cosine of x, where x is in degrees. If x is a matrix, x needs to be a square matrix.

> **Julia 1.7**
>
> Matrix arguments require Julia 1.7 or later.

source

Base.Math.tand – Function.

```
tand(x)
```

Compute tangent of x, where x is in degrees. If x is a matrix, x needs to be a square matrix.

> **Julia 1.7**
>
> Matrix arguments require Julia 1.7 or later.

source

Base.Math.sincosd – Function.

```
sincosd(x)
```

Simultaneously compute the sine and cosine of x, where x is in degrees.

> **Julia 1.3**
>
> This function requires at least Julia 1.3.

source

Base.Math.sinpi – Function.

```
sinpi(x)
```

Compute $\sin(\pi x)$ more accurately than sin(pi*x), especially for large x.

See also sind, cospi, sincospi.

source

Base.Math.cospi – Function.

```
cospi(x)
```

Compute $\cos(\pi x)$ more accurately than cos(pi*x), especially for large x.

source

Base.Math.tanpi – Function.

```
tanpi(x)
```

Compute $\tan(\pi x)$ more accurately than tan(pi*x), especially for large x.

Julia 1.10

This function requires at least Julia 1.10.

See also tand, sinpi, cospi, sincospi.

source

Base.Math.sincospi – Function.

```
sincospi(x)
```

Simultaneously compute sinpi(x) and cospi(x) (the sine and cosine of π*x, where x is in radians), returning a tuple (sine, cosine).

Julia 1.6

This function requires Julia 1.6 or later.

See also: cispi, sincosd, sinpi.

source

Base.sinh – Method.

```
sinh(x)
```

Compute hyperbolic sine of x.

source

Base.cosh – Method.

```
cosh(x)
```

Compute hyperbolic cosine of x.

source

Base.tanh – Method.

```
tanh(x)
```

Compute hyperbolic tangent of x.

See also tan, atanh.

Examples

```
julia> tanh.(-3:3f0)  # Here 3f0 isa Float32
7-element Vector{Float32}:
 -0.9950548
 -0.9640276
 -0.7615942
  0.0
  0.7615942
  0.9640276
  0.9950548

julia> tan.(im .* (1:3))
3-element Vector{ComplexF64}:
 0.0 + 0.7615941559557649im
 0.0 + 0.9640275800758169im
 0.0 + 0.9950547536867306im
```

source

Base.asin – Method.

```
asin(x)
```

Compute the inverse sine of x, where the output is in radians.

See also asind for output in degrees.

Examples

```
julia> asin.((0, 1/2, 1))
(0.0, 0.5235987755982989, 1.5707963267948966)

julia> asind.((0, 1/2, 1))
(0.0, 30.000000000000004, 90.0)
```

source

Base.acos – Method.

```
acos(x)
```

Compute the inverse cosine of x, where the output is in radians

source

Base.atan – Method.

```
atan(y)
atan(y, x)
```

Compute the inverse tangent of y or y/x, respectively.

For one argument, this is the angle in radians between the positive *x*-axis and the point (1, *y*), returning a value in the interval $[-\pi/2, \pi/2]$.

For two arguments, this is the angle in radians between the positive *x*-axis and the point (*x*, *y*), returning a value in the interval $[-\pi, \pi]$. This corresponds to a standard atan2 function. Note that by convention atan(0.0,x) is defined as π and atan(-0.0,x) is defined as $-\pi$ when x < 0.

See also atand for degrees.

Examples

```
julia> rad2deg(atan(-1/√3))
-30.000000000000004

julia> rad2deg(atan(-1, √3))
-30.000000000000004

julia> rad2deg(atan(1, -√3))
150.0
```

source

Base.Math.asind – Function.

```
asind(x)
```

Compute the inverse sine of x, where the output is in degrees. If x is a matrix, x needs to be a square matrix.

> **Julia 1.7**
>
> Matrix arguments require Julia 1.7 or later.

source

Base.Math.acosd – Function.

```
acosd(x)
```

Compute the inverse cosine of x, where the output is in degrees. If x is a matrix, x needs to be a square matrix.

> **Julia 1.7**
>
> Matrix arguments require Julia 1.7 or later.

source

Base.Math.atand – Function.

```
atand(y)
atand(y,x)
```

Compute the inverse tangent of y or y/x, respectively, where the output is in degrees.

> **Julia 1.7**
>
> The one-argument method supports square matrix arguments as of Julia 1.7.

source

Base.Math.sec – Method.

```
sec(x)
```

Compute the secant of x, where x is in radians.

source

Base.Math.csc – Method.

```
csc(x)
```

Compute the cosecant of x, where x is in radians.

source

`Base.Math.cot` – Method.

```
cot(x)
```

Compute the cotangent of x, where x is in radians.

source

`Base.Math.secd` – Function.

```
secd(x)
```

Compute the secant of x, where x is in degrees.

source

`Base.Math.cscd` – Function.

```
cscd(x)
```

Compute the cosecant of x, where x is in degrees.

source

`Base.Math.cotd` – Function.

```
cotd(x)
```

Compute the cotangent of x, where x is in degrees.

source

`Base.Math.asec` – Method.

```
asec(x)
```

Compute the inverse secant of x, where the output is in radians.

source

`Base.Math.acsc` – Method.

```
acsc(x)
```

Compute the inverse cosecant of x, where the output is in radians.

source

`Base.Math.acot` – Method.

```
acot(x)
```

Compute the inverse cotangent of x, where the output is in radians.

source

Base.Math.asecd – Function.

```
asecd(x)
```

Compute the inverse secant of x, where the output is in degrees. If x is a matrix, x needs to be a square matrix.

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.acscd – Function.

```
acscd(x)
```

Compute the inverse cosecant of x, where the output is in degrees. If x is a matrix, x needs to be a square matrix.

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.acotd – Function.

```
acotd(x)
```

Compute the inverse cotangent of x, where the output is in degrees. If x is a matrix, x needs to be a square matrix.

Julia 1.7

Matrix arguments require Julia 1.7 or later.

source

Base.Math.sech – Method.

```
sech(x)
```

Compute the hyperbolic secant of x.

source

`Base.Math.csch` – Method.

```
csch(x)
```

Compute the hyperbolic cosecant of x.

source

`Base.Math.coth` – Method.

```
coth(x)
```

Compute the hyperbolic cotangent of x.

source

`Base.asinh` – Method.

```
asinh(x)
```

Compute the inverse hyperbolic sine of x.

source

`Base.acosh` – Method.

```
acosh(x)
```

Compute the inverse hyperbolic cosine of x.

source

`Base.atanh` – Method.

```
atanh(x)
```

Compute the inverse hyperbolic tangent of x.

source

`Base.Math.asech` – Method.

```
asech(x)
```

Compute the inverse hyperbolic secant of x.

source

`Base.Math.acsch` – Method.

```
acsch(x)
```

Compute the inverse hyperbolic cosecant of x.

source

`Base.Math.acoth` – Method.

```
acoth(x)
```

Compute the inverse hyperbolic cotangent of x.

source

`Base.Math.sinc` – Function.

```
sinc(x)
```

Compute $\sin(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

See also `cosc`, its derivative.

source

`Base.Math.cosc` – Function.

```
cosc(x)
```

Compute $\cos(\pi x)/x - \sin(\pi x)/(\pi x^2)$ if $x \neq 0$, and 0 if $x = 0$. This is the derivative of `sinc(x)`.

source

`Base.Math.deg2rad` – Function.

```
deg2rad(x)
```

Convert x from degrees to radians.

See also `rad2deg`, `sind`, `pi`.

Examples

```
julia> deg2rad(90)
1.5707963267948966
```

source

`Base.Math.rad2deg` – Function.

```
rad2deg(x)
```

Convert x from radians to degrees.

See also `deg2rad`.

Examples

```
julia> rad2deg(pi)
180.0
```

source

`Base.Math.hypot` – Function.

```
hypot(x, y)
```

Compute the hypotenuse $\sqrt{|x|^2 + |y|^2}$ avoiding overflow and underflow.

This code is an implementation of the algorithm described in: An Improved Algorithm for `hypot(a,b)` by Carlos F. Borges The article is available online at arXiv at the link https://arxiv.org/abs/1904.09481

```
hypot(x...)
```

Compute the hypotenuse $\sqrt{\sum |x_i|^2}$ avoiding overflow and underflow.

See also `norm` in the `LinearAlgebra` standard library.

Examples

```
julia> a = Int64(10)^10;

julia> hypot(a, a)
1.4142135623730951e10

julia> √(a^2 + a^2) # a^2 overflows
ERROR: DomainError with -2.914184810805068e18:
sqrt was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]

julia> hypot(3, 4im)
```

```
5.0

julia> hypot(-5.7)
5.7

julia> hypot(3, 4im, 12.0)
13.0

julia> using LinearAlgebra

julia> norm([a, a, a, a]) == hypot(a, a, a, a)
true
```

source

Base.log – Method.

```
log(x)
```

Compute the natural logarithm of x. Throws DomainError for negative Real arguments. Use complex negative arguments to obtain complex results.

See also , log1p, log2, log10.

Examples

```
julia> log(2)
0.6931471805599453

julia> log(-3)
ERROR: DomainError with -3.0:
log was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try log(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]

julia> log.(exp.(-1:1))
3-element Vector{Float64}:
 -1.0
  0.0
  1.0
```

source

Base.log – Method.

```
log(b,x)
```

Compute the base b logarithm of x. Throws DomainError for negative Real arguments.

Examples

```
julia> log(4,8)
1.5

julia> log(4,2)
0.5

julia> log(-2, 3)
ERROR: DomainError with -2.0:
log was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try log(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]

julia> log(2, -3)
ERROR: DomainError with -3.0:
log was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try log(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

Note

If b is a power of 2 or 10, `log2` or `log10` should be used, as these will typically be faster and more accurate. For example,

```
julia> log(100,1000000)
2.9999999999999996

julia> log10(1000000)/2
3.0
```

source

Base.log2 – Function.

```
log2(x)
```

Compute the logarithm of x to base 2. Throws `DomainError` for negative `Real` arguments.

See also: `exp2`, `ldexp`, `ispow2`.

Examples

```
julia> log2(4)
2.0

julia> log2(10)
3.321928094887362
```

```
julia> log2(-2)
ERROR: DomainError with -2.0:
log2 was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try log2(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64) at ./math.jl:31
[...]

julia> log2.(2.0 .^ (-1:1))
3-element Vector{Float64}:
 -1.0
  0.0
  1.0
```

source

Base.log10 – Function.

```
log10(x)
```

Compute the logarithm of x to base 10. Throws DomainError for negative Real arguments.

Examples

```
julia> log10(100)
2.0

julia> log10(2)
0.3010299956639812

julia> log10(-2)
ERROR: DomainError with -2.0:
log10 was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try log10(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64) at ./math.jl:31
[...]
```

source

Base.log1p – Function.

```
log1p(x)
```

Accurate natural logarithm of 1+x. Throws DomainError for Real arguments less than -1.

Examples

```
julia> log1p(-0.5)
-0.6931471805599453

julia> log1p(0)
0.0

julia> log1p(-2)
ERROR: DomainError with -2.0:
log1p was called with a real argument < -1 but will only return a complex result if called with
↪  a complex argument. Try log1p(Complex(x)).
Stacktrace:
 [1] throw_complex_domainerror(::Symbol, ::Float64) at ./math.jl:31
[...]
```

source

`Base.Math.frexp` – Function.

```
frexp(val)
```

Return `(x,exp)` such that x has a magnitude in the interval $[1/2, 1)$ or 0, and `val` is equal to $x \times 2^{exp}$.

Examples

```
julia> frexp(12.8)
(0.8, 4)
```

source

`Base.exp` – Method.

```
exp(x)
```

Compute the natural base exponential of x, in other words ^{x}.

See also `exp2`, `exp10` and `cis`.

Examples

```
julia> exp(1.0)
2.718281828459045

julia> exp(im * pi) ≈ cis(pi)
true
```

source

`Base.exp2` – Function.

```
exp2(x)
```

Compute the base 2 exponential of x, in other words 2^x.

See also ldexp, <<.

Examples

```
julia> exp2(5)
32.0

julia> 2^5
32

julia> exp2(63) > typemax(Int)
true
```

source

Base.exp10 – Function.

```
exp10(x)
```

Compute the base 10 exponential of x, in other words 10^x.

Examples

```
julia> exp10(2)
100.0

julia> 10^2
100
```

source

Base.Math.ldexp – Function.

```
ldexp(x, n)
```

Compute $x \times 2^n$.

Examples

```
julia> ldexp(5., 2)
20.0
```

source

Base.Math.modf – Function.

```
modf(x)
```

Return a tuple (`fpart, ipart`) of the fractional and integral parts of a number. Both parts have the same sign as the argument.

Examples

```
julia> modf(3.5)
(0.5, 3.0)

julia> modf(-3.5)
(-0.5, -3.0)
```

source

`Base.expm1` – Function.

```
expm1(x)
```

Accurately compute $e^x - 1$. It avoids the loss of precision involved in the direct evaluation of exp(x)-1 for small values of x.

Examples

```
julia> expm1(1e-16)
1.0e-16

julia> exp(1e-16) - 1
0.0
```

source

`Base.round` – Method.

```
round([T,] x, [r::RoundingMode])
round(x, [r::RoundingMode]; digits::Integer=0, base = 10)
round(x, [r::RoundingMode]; sigdigits::Integer, base = 10)
```

Rounds the number x.

Without keyword arguments, x is rounded to an integer value, returning a value of type T, or of the same type of x if no T is provided. An `InexactError` will be thrown if the value is not representable by T, similar to `convert`.

If the `digits` keyword argument is provided, it rounds to the specified number of digits after the decimal place (or before if negative), in base base.

If the `sigdigits` keyword argument is provided, it rounds to the specified number of significant digits, in base base.

The RoundingMode r controls the direction of the rounding; the default is RoundNearest, which rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer. Note that round may give incorrect results if the global rounding mode is changed (see rounding).

Examples

```
julia> round(1.7)
2.0

julia> round(Int, 1.7)
2

julia> round(1.5)
2.0

julia> round(2.5)
2.0

julia> round(pi; digits=2)
3.14

julia> round(pi; digits=3, base=2)
3.125

julia> round(123.456; sigdigits=2)
120.0

julia> round(357.913; sigdigits=4, base=2)
352.0
```

Note

Rounding to specified digits in bases other than 2 can be inexact when operating on binary floating point numbers. For example, the Float64 value represented by 1.15 is actually *less* than 1.15, yet will be rounded to 1.2. For example:

```
julia> x = 1.15
1.15

julia> big(1.15)
1.149999999999999911182158029987476766109466552734375

julia> x < 115//100
true

julia> round(x, digits=1)
1.2
```

Extensions

To extend round to new numeric types, it is typically sufficient to define `Base.round(x::NewType, r::RoundingMode)`.

source

`Base.Rounding.RoundingMode` – Type.

```
RoundingMode
```

A type used for controlling the rounding mode of floating point operations (via `rounding`/`setrounding` functions), or as optional arguments for rounding to the nearest integer (via the `round` function).

Currently supported rounding modes are:

- `RoundNearest` (default)
- `RoundNearestTiesAway`
- `RoundNearestTiesUp`
- `RoundToZero`
- `RoundFromZero`
- `RoundUp`
- `RoundDown`

Julia 1.9

`RoundFromZero` requires at least Julia 1.9. Prior versions support `RoundFromZero` for `BigFloats` only.

source

`Base.Rounding.RoundNearest` – Constant.

```
RoundNearest
```

The default rounding mode. Rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer.

source

`Base.Rounding.RoundNearestTiesAway` – Constant.

```
RoundNearestTiesAway
```

Rounds to nearest integer, with ties rounded away from zero (C/C++ `round` behaviour).

source

`Base.Rounding.RoundNearestTiesUp` – Constant.

```
RoundNearestTiesUp
```

Rounds to nearest integer, with ties rounded toward positive infinity (Java/JavaScript `round` behaviour).

source

`Base.Rounding.RoundToZero` – Constant.

```
RoundToZero
```

`round` using this rounding mode is an alias for `trunc`.

source

`Base.Rounding.RoundFromZero` – Constant.

```
RoundFromZero
```

Rounds away from zero.

Julia 1.9

`RoundFromZero` requires at least Julia 1.9. Prior versions support `RoundFromZero` for `BigFloats` only.

Examples

```
julia> BigFloat("1.0000000000000001", 5, RoundFromZero)
1.06
```

source

`Base.Rounding.RoundUp` – Constant.

```
RoundUp
```

`round` using this rounding mode is an alias for `ceil`.

source

`Base.Rounding.RoundDown` – Constant.

```
RoundDown
```

`round` using this rounding mode is an alias for `floor`.

source

`Base.round` – Method.

```
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]])
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]]; digits=0, base=10)
round(z::Complex[, RoundingModeReal, [RoundingModeImaginary]]; sigdigits, base=10)
```

Return the nearest integral value of the same type as the complex-valued z to z, breaking ties using the specified RoundingModes. The first RoundingMode is used for rounding the real components while the second is used for rounding the imaginary components.

RoundingModeReal and RoundingModeImaginary default to RoundNearest, which rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer.

Example

```
julia> round(3.14 + 4.5im)
3.0 + 4.0im

julia> round(3.14 + 4.5im, RoundUp, RoundNearestTiesUp)
4.0 + 5.0im

julia> round(3.14159 + 4.512im; digits = 1)
3.1 + 4.5im

julia> round(3.14159 + 4.512im; sigdigits = 3)
3.14 + 4.51im
```

source

Base.ceil – Function.

```
ceil([T,] x)
ceil(x; digits::Integer= [, base = 10])
ceil(x; sigdigits::Integer= [, base = 10])
```

ceil(x) returns the nearest integral value of the same type as x that is greater than or equal to x.

ceil(T, x) converts the result to type T, throwing an InexactError if the value is not representable.

Keywords digits, sigdigits and base work as for round.

source

Base.floor – Function.

```
floor([T,] x)
floor(x; digits::Integer= [, base = 10])
floor(x; sigdigits::Integer= [, base = 10])
```

floor(x) returns the nearest integral value of the same type as x that is less than or equal to x.

floor(T, x) converts the result to type T, throwing an InexactError if the value is not representable.

Keywords digits, sigdigits and base work as for round.

source

Base.trunc – Function.

```
trunc([T,] x)
trunc(x; digits::Integer= [, base = 10])
trunc(x; sigdigits::Integer= [, base = 10])
```

`trunc(x)` returns the nearest integral value of the same type as `x` whose absolute value is less than or equal to the absolute value of `x`.

`trunc(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

Keywords `digits`, `sigdigits` and `base` work as for `round`.

See also: `%`, `floor`, `unsigned`, `unsafe_trunc`.

Examples

```
julia> trunc(2.22)
2.0

julia> trunc(-2.22, digits=1)
-2.2

julia> trunc(Int, -2.22)
-2
```

source

`Base.unsafe_trunc` – Function.

```
unsafe_trunc(T, x)
```

Return the nearest integral value of type `T` whose absolute value is less than or equal to the absolute value of `x`. If the value is not representable by `T`, an arbitrary value will be returned. See also `trunc`.

Examples

```
julia> unsafe_trunc(Int, -2.2)
-2

julia> unsafe_trunc(Int, NaN)
-9223372036854775808
```

source

`Base.min` – Function.

```
min(x, y, ...)
```

Return the minimum of the arguments (with respect to `isless`). See also the `minimum` function to take the minimum element from a collection.

Examples

```
julia> min(2, 5, 1)
1
```

source

Base.max – Function.

```
max(x, y, ...)
```

Return the maximum of the arguments (with respect to `isless`). See also the `maximum` function to take the maximum element from a collection.

Examples

```
julia> max(2, 5, 1)
5
```

source

Base.minmax – Function.

```
minmax(x, y)
```

Return `(min(x,y), max(x,y))`.

See also `extrema` that returns `(minimum(x), maximum(x))`.

Examples

```
julia> minmax('c','b')
('b', 'c')
```

source

Base.Math.clamp – Function.

```
clamp(x, lo, hi)
```

Return `x` if `lo <= x <= hi`. If `x > hi`, return `hi`. If `x < lo`, return `lo`. Arguments are promoted to a common type.

See also `clamp!`, `min`, `max`.

Julia 1.3

`missing` as the first argument requires at least Julia 1.3.

Examples

```
julia> clamp.([pi, 1.0, big(10)], 2.0, 9.0)
3-element Vector{BigFloat}:
 3.141592653589793238462643383279502884197169399375105820974944592307816406286198
 2.0
 9.0

julia> clamp.([11, 8, 5], 10, 6)  # an example where lo > hi
3-element Vector{Int64}:
  6
  6
 10
```

source

```
clamp(x, T)::T
```

Clamp x between `typemin(T)` and `typemax(T)` and convert the result to type T.

See also `trunc`.

Examples

```
julia> clamp(200, Int8)
127

julia> clamp(-200, Int8)
-128

julia> trunc(Int, 4pi^2)
39
```

source

```
clamp(x::Integer, r::AbstractUnitRange)
```

Clamp x to lie within range `r`.

> **Julia 1.6**
>
> This method requires at least Julia 1.6.

source

`Base.Math.clamp!` – Function.

```
clamp!(array::AbstractArray, lo, hi)
```

Restrict values in `array` to the specified range, in-place. See also `clamp`.

> **Julia 1.3**
>
> `missing` entries in `array` require at least Julia 1.3.

Examples

```
julia> row = collect(-4:4)';

julia> clamp!(row, 0, Inf)
1×9 adjoint(::Vector{Int64}) with eltype Int64:
 0  0  0  0  0  1  2  3  4

julia> clamp.((-4:4)', 0, Inf)
1×9 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0  1.0  2.0  3.0  4.0
```

source

`Base.abs` – Function.

```
abs(x)
```

The absolute value of `x`.

When `abs` is applied to signed integers, overflow may occur, resulting in the return of a negative value. This overflow occurs only when `abs` is applied to the minimum representable value of a signed integer. That is, when `x == typemin(typeof(x))`, `abs(x) == x < 0`, not `-x` as might be expected.

See also: `abs2`, `unsigned`, `sign`.

Examples

```
julia> abs(-3)
3

julia> abs(1 + im)
1.4142135623730951

julia> abs.(Int8[-128 -127 -126 0 126 127])  # overflow at typemin(Int8)
1×6 Matrix{Int8}:
 -128  127  126  0  126  127

julia> maximum(abs, [1, -2, 3, -4])
4
```

source

`Base.Checked.checked_abs` – Function.

```
Base.checked_abs(x)
```

Calculates abs(x), checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. Int) cannot represent abs(typemin(Int)), thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_neg – Function.

```
Base.checked_neg(x)
```

Calculates -x, checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. Int) cannot represent -typemin(Int), thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_add – Function.

```
Base.checked_add(x, y)
```

Calculates x+y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_sub – Function.

```
Base.checked_sub(x, y)
```

Calculates x-y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_mul – Function.

```
Base.checked_mul(x, y)
```

Calculates x*y, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

Base.Checked.checked_div – Function.

```
Base.checked_div(x, y)
```

Calculates `div(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_rem` – Function.

```
Base.checked_rem(x, y)
```

Calculates `x%y`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_fld` – Function.

```
Base.checked_fld(x, y)
```

Calculates `fld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_mod` – Function.

```
Base.checked_mod(x, y)
```

Calculates `mod(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.checked_cld` – Function.

```
Base.checked_cld(x, y)
```

Calculates `cld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

source

`Base.Checked.add_with_overflow` – Function.

```
Base.add_with_overflow(x, y) -> (r, f)
```

Calculates `r = x+y`, with the flag `f` indicating whether overflow has occurred.

source

`Base.Checked.sub_with_overflow` – Function.

```
Base.sub_with_overflow(x, y) -> (r, f)
```

Calculates `r = x-y`, with the flag `f` indicating whether overflow has occurred.

source

`Base.Checked.mul_with_overflow` – Function.

```
Base.mul_with_overflow(x, y) -> (r, f)
```

Calculates `r = x*y`, with the flag `f` indicating whether overflow has occurred.

source

`Base.abs2` – Function.

```
abs2(x)
```

Squared absolute value of `x`.

This can be faster than `abs(x)^2`, especially for complex numbers where `abs(x)` requires a square root via `hypot`.

See also `abs`, `conj`, `real`.

Examples

```
julia> abs2(-3)
9

julia> abs2(3.0 + 4.0im)
25.0

julia> sum(abs2, [1+2im, 3+4im])  # LinearAlgebra.norm(x)^2
30
```

source

`Base.copysign` – Function.

```
copysign(x, y) -> z
```

Return z which has the magnitude of x and the same sign as y.

Examples

```
julia> copysign(1, -2)
-1

julia> copysign(-1, 2)
1
```

source

Base.sign – Function.

```
sign(x)
```

Return zero if x==0 and $x/|x|$ otherwise (i.e., ±1 for real x).

See also signbit, zero, copysign, flipsign.

Examples

```
julia> sign(-4.0)
-1.0

julia> sign(99)
1

julia> sign(-0.0)
-0.0

julia> sign(0 + im)
0.0 + 1.0im
```

source

Base.signbit – Function.

```
signbit(x)
```

Return `true` if the value of the sign of x is negative, otherwise `false`.

See also sign and copysign.

Examples

```
julia> signbit(-4)
true

julia> signbit(5)
false

julia> signbit(5.5)
false

julia> signbit(-4.1)
true
```

source

Base.flipsign – Function.

```
flipsign(x, y)
```

Return x with its sign flipped if y is negative. For example `abs(x) = flipsign(x,x)`.

Examples

```
julia> flipsign(5, 3)
5

julia> flipsign(5, -3)
-5
```

source

Base.sqrt – Method.

```
sqrt(x)
```

Return $\sqrt{x}$. Throws `DomainError` for negative `Real` arguments. Use complex negative arguments instead. The prefix operator √ is equivalent to `sqrt`.

See also: `hypot`.

Examples

```
julia> sqrt(big(81))
9.0

julia> sqrt(big(-81))
ERROR: DomainError with -81.0:
NaN result for non-NaN input.
Stacktrace:
 [1] sqrt(::BigFloat) at ./mpfr.jl:501
[...]
```

```
julia> sqrt(big(complex(-81)))
0.0 + 9.0im

julia> .√(1:4)
4-element Vector{Float64}:
 1.0
 1.4142135623730951
 1.7320508075688772
 2.0
```

source

`Base.isqrt` – Function.

```
isqrt(n::Integer)
```

Integer square root: the largest integer `m` such that `m*m <= n`.

```
julia> isqrt(5)
2
```

source

`Base.Math.cbrt` – Function.

```
cbrt(x::Real)
```

Return the cube root of `x`, i.e. $x^{1/3}$. Negative values are accepted (returning the negative real root when $x < 0$).

The prefix operator `∛` is equivalent to `cbrt`.

Examples

```
julia> cbrt(big(27))
3.0

julia> cbrt(big(-27))
-3.0
```

source

`Base.real` – Function.

```
real(z)
```

Return the real part of the complex number z.

See also: imag, reim, complex, isreal, Real.

Examples

```
julia> real(1 + 3im)
1
```

source

```
real(T::Type)
```

Return the type that represents the real part of a value of type T. e.g: for T == Complex{R}, returns R. Equivalent to typeof(real(zero(T))).

Examples

```
julia> real(Complex{Int})
Int64

julia> real(Float64)
Float64
```

source

```
real(A::AbstractArray)
```

Return an array containing the real part of each entry in array A.

Equivalent to real.(A), except that when eltype(A) <: Real A is returned without copying, and that when A has zero dimensions, a 0-dimensional array is returned (rather than a scalar).

Examples

```
julia> real([1, 2im, 3 + 4im])
3-element Vector{Int64}:
 1
 0
 3

julia> real(fill(2 - im))
0-dimensional Array{Int64, 0}:
2
```

source

Base.imag – Function.

```
imag(z)
```

Return the imaginary part of the complex number z.

See also: conj, reim, adjoint, angle.

Examples

```
julia> imag(1 + 3im)
3
```

source

```
imag(A::AbstractArray)
```

Return an array containing the imaginary part of each entry in array A.

Equivalent to `imag.(A)`, except that when A has zero dimensions, a 0-dimensional array is returned (rather than a scalar).

Examples

```
julia> imag([1, 2im, 3 + 4im])
3-element Vector{Int64}:
 0
 2
 4

julia> imag(fill(2 - im))
0-dimensional Array{Int64, 0}:
-1
```

source

`Base.reim` – Function.

```
reim(z)
```

Return a tuple of the real and imaginary parts of the complex number z.

Examples

```
julia> reim(1 + 3im)
(1, 3)
```

source

```
reim(A::AbstractArray)
```

Return a tuple of two arrays containing respectively the real and the imaginary part of each entry in A.

Equivalent to `(real.(A), imag.(A))`, except that when `eltype(A) <: Real` A is returned without copying to represent the real part, and that when A has zero dimensions, a 0-dimensional array is returned (rather than a scalar).

Examples

```
julia> reim([1, 2im, 3 + 4im])
([1, 0, 3], [0, 2, 4])

julia> reim(fill(2 - im))
(fill(2), fill(-1))
```

source

`Base.conj` – Function.

```
conj(z)
```

Compute the complex conjugate of a complex number z.

See also: `angle`, `adjoint`.

Examples

```
julia> conj(1 + 3im)
1 - 3im
```

source

```
conj(A::AbstractArray)
```

Return an array containing the complex conjugate of each entry in array A.

Equivalent to `conj.(A)`, except that when `eltype(A) <: Real` A is returned without copying, and that when A has zero dimensions, a 0-dimensional array is returned (rather than a scalar).

Examples

```
julia> conj([1, 2im, 3 + 4im])
3-element Vector{Complex{Int64}}:
 1 + 0im
 0 - 2im
 3 - 4im

julia> conj(fill(2 - im))
0-dimensional Array{Complex{Int64}, 0}:
2 + 1im
```

source

`Base.angle` – Function.

```
angle(z)
```

Compute the phase angle in radians of a complex number z.

See also: atan, cis.

Examples

```
julia> rad2deg(angle(1 + im))
45.0

julia> rad2deg(angle(1 - im))
-45.0

julia> rad2deg(angle(-1 - im))
-135.0
```

source

Base.cis – Function.

```
cis(x)
```

More efficient method for exp(im*x) by using Euler's formula: $cos(x) + isin(x) = \exp(ix)$.

See also cispi, sincos, exp, angle.

Examples

```
julia> cis(π) ≈ -1
true
```

source

Base.cispi – Function.

```
cispi(x)
```

More accurate method for cis(pi*x) (especially for large x).

See also cis, sincospi, exp, angle.

Examples

```
julia> cispi(10000)
1.0 + 0.0im

julia> cispi(0.25 + 1im)
0.030556854645954562 + 0.030556854645954562im
```

Julia 1.6

This function requires Julia 1.6 or later.

source

`Base.binomial` – Function.

```
binomial(n::Integer, k::Integer)
```

The *binomial coefficient* $\binom{n}{k}$, being the coefficient of the kth term in the polynomial expansion of $(1+x)^n$.

If n is non-negative, then it is the number of ways to choose k out of n items:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

where $n!$ is the `factorial` function.

If n is negative, then it is defined in terms of the identity

$$\binom{n}{k} = (-1)^k \binom{k-n-1}{k}$$

See also `factorial`.

Examples

```
julia> binomial(5, 3)
10

julia> factorial(5) ÷ (factorial(5-3) * factorial(3))
10

julia> binomial(-5, 3)
-35
```

External links

- Binomial coefficient on Wikipedia.

source

```
binomial(x::Number, k::Integer)
```

The generalized binomial coefficient, defined for `k ≥ 0` by the polynomial

$$\frac{1}{k!} \prod_{j=0}^{k-1} (x-j)$$

When k < 0 it returns zero.

For the case of integer x, this is equivalent to the ordinary integer binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Further generalizations to non-integer k are mathematically possible, but involve the Gamma function and/or the beta function, which are not provided by the Julia standard library but are available in external packages such as SpecialFunctions.jl.

External links

- Binomial coefficient on Wikipedia.

source

Base.factorial – Function.

```
factorial(n::Integer)
```

Factorial of n. If n is an Integer, the factorial is computed as an integer (promoted to at least 64 bits). Note that this may overflow if n is not small, but you can use factorial(big(n)) to compute the result exactly in arbitrary precision.

See also binomial.

Examples

```
julia> factorial(6)
720

julia> factorial(21)
ERROR: OverflowError: 21 is too large to look up in the table; consider using
↪  `factorial(big(21))` instead
Stacktrace:
[...]

julia> factorial(big(21))
51090942171709440000
```

External links

- Factorial on Wikipedia.

source

Base.gcd – Function.

```
gcd(x, y...)
```

Greatest common (positive) divisor (or zero if all arguments are zero). The arguments may be integer and rational numbers.

Julia 1.4

Rational arguments require Julia 1.4 or later.

Examples

```
julia> gcd(6, 9)
3

julia> gcd(6, -9)
3

julia> gcd(6, 0)
6

julia> gcd(0, 0)
0

julia> gcd(1//3, 2//3)
1//3

julia> gcd(1//3, -2//3)
1//3

julia> gcd(1//3, 2)
1//3

julia> gcd(0, 0, 10, 15)
5
```

source

`Base.lcm` – Function.

```
lcm(x, y...)
```

Least common (positive) multiple (or zero if any argument is zero). The arguments may be integer and rational numbers.

Julia 1.4

Rational arguments require Julia 1.4 or later.

Examples

```
julia> lcm(2, 3)
6
```

```
julia> lcm(-2, 3)
6

julia> lcm(0, 3)
0

julia> lcm(0, 0)
0

julia> lcm(1//3, 2//3)
2//3

julia> lcm(1//3, -2//3)
2//3

julia> lcm(1//3, 2)
2//1

julia> lcm(1, 3, 5, 7)
105
```

source

Base.gcdx – Function.

```
gcdx(a, b)
```

Computes the greatest common (positive) divisor of a and b and their Bézout coefficients, i.e. the integer coefficients u and v that satisfy $ua + vb = d = gcd(a, b)$. $gcdx(a, b)$ returns (d, u, v).

The arguments may be integer and rational numbers.

Julia 1.4

Rational arguments require Julia 1.4 or later.

Examples

```
julia> gcdx(12, 42)
(6, -3, 1)

julia> gcdx(240, 46)
(2, -9, 47)
```

> **Note**
>
> Bézout coefficients are *not* uniquely defined. gcdx returns the minimal Bézout coefficients that are computed by the extended Euclidean algorithm. (Ref: D. Knuth, TAoCP, 2/e, p. 325, Algorithm X.) For signed integers, these coefficients u and v are minimal in the sense that $|u| < |b/d|$ and $|v| < |a/d|$. Furthermore, the signs of u and v are chosen so that d is positive. For unsigned integers, the coefficients u and v might be near their typemax, and the identity then holds only via the unsigned integers' modulo arithmetic.

source

Base.ispow2 – Function.

```
ispow2(n::Number) -> Bool
```

Test whether n is an integer power of two.

See also count_ones, prevpow, nextpow.

Examples

```
julia> ispow2(4)
true

julia> ispow2(5)
false

julia> ispow2(4.5)
false

julia> ispow2(0.25)
true

julia> ispow2(1//8)
true
```

> **Julia 1.6**
>
> Support for non-Integer arguments was added in Julia 1.6.

source

Base.nextpow – Function.

```
nextpow(a, x)
```

The smallest a^n not less than x, where n is a non-negative integer. a must be greater than 1, and x must be greater than 0.

See also prevpow.

Examples

```
julia> nextpow(2, 7)
8

julia> nextpow(2, 9)
16

julia> nextpow(5, 20)
25

julia> nextpow(4, 16)
16
```

source

Base.prevpow – Function.

```
prevpow(a, x)
```

The largest a^n not greater than x, where n is a non-negative integer. a must be greater than 1, and x must not be less than 1.

See also nextpow, isqrt.

Examples

```
julia> prevpow(2, 7)
4

julia> prevpow(2, 9)
8

julia> prevpow(5, 20)
5

julia> prevpow(4, 16)
16
```

source

Base.nextprod – Function.

```
nextprod(factors::Union{Tuple,AbstractVector}, n)
```

Next integer greater than or equal to n that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2, etcetera, for factors k_i in factors.

Examples

```
julia> nextprod((2, 3), 105)
108

julia> 2^2 * 3^3
108
```

Julia 1.6

The method that accepts a tuple requires Julia 1.6 or later.

source

`Base.invmod` – Function.

```
invmod(n, m)
```

Take the inverse of `n` modulo `m`: `y` such that $ny = 1 \pmod m$, and $div(y, m) = 0$. This will throw an error if $m = 0$, or if $gcd(n, m) \neq 1$.

Examples

```
julia> invmod(2, 5)
3

julia> invmod(2, 3)
2

julia> invmod(5, 6)
5
```

source

`Base.powermod` – Function.

```
powermod(x::Integer, p::Integer, m)
```

Compute $x^p \pmod m$.

Examples

```
julia> powermod(2, 6, 5)
4

julia> mod(2^6, 5)
4

julia> powermod(5, 2, 20)
5
```

```
julia> powermod(5, 2, 19)
6

julia> powermod(5, 3, 19)
11
```

source

Base.ndigits – Function.

```
ndigits(n::Integer; base::Integer=10, pad::Integer=1)
```

Compute the number of digits in integer n written in base base (base must not be in [-1, 0, 1]), optionally padded with zeros to a specified size (the result will never be less than pad).

See also digits, count_ones.

Examples

```
julia> ndigits(0)
1

julia> ndigits(12345)
5

julia> ndigits(1022, base=16)
3

julia> string(1022, base=16)
"3fe"

julia> ndigits(123, pad=5)
5

julia> ndigits(-123)
3
```

source

Base.add_sum – Function.

```
Base.add_sum(x, y)
```

The reduction operator used in sum. The main difference from + is that small integers are promoted to Int/UInt.

source

Base.widemul – Function.

```
widemul(x, y)
```

Multiply x and y, giving the result as a larger type.

See also promote, Base.add_sum.

Examples

```
julia> widemul(Float32(3.0), 4.0) isa BigFloat
true

julia> typemax(Int8) * typemax(Int8)
1

julia> widemul(typemax(Int8), typemax(Int8))  # == 127^2
16129
```

source

Base.Math.evalpoly – Function.

```
evalpoly(x, p)
```

Evaluate the polynomial $\sum_k x^{k-1} p[k]$ for the coefficients p[1], p[2], ...; that is, the coefficients are given in ascending order by power of x. Loops are unrolled at compile time if the number of coefficients is statically known, i.e. when p is a Tuple. This function generates efficient code using Horner's method if x is real, or using a Goertzel-like [1] algorithm if x is complex.

> **Julia 1.4**
>
> This function requires Julia 1.4 or later.

Example

```
julia> evalpoly(2, (1, 2, 3))
17
```

source

Base.Math.@evalpoly – Macro.

```
@evalpoly(z, c...)
```

Evaluate the polynomial $\sum_k z^{k-1} c[k]$ for the coefficients c[1], c[2], ...; that is, the coefficients are given in ascending order by power of z. This macro expands to efficient inline code that uses either Horner's method or, for complex z, a more efficient Goertzel-like algorithm.

[1] Donald Knuth, Art of Computer Programming, Volume 2: Seminumerical Algorithms, Sec. 4.6.4.

See also `evalpoly`.

Examples

```
julia> @evalpoly(3, 1, 0, 1)
10

julia> @evalpoly(2, 1, 0, 1)
5

julia> @evalpoly(2, 1, 1, 1)
7
```

source

`Base.FastMath.@fastmath` – Macro.

```
@fastmath expr
```

Execute a transformed version of the expression, which calls functions that may violate strict IEEE semantics. This allows the fastest possible operation, but results are undefined –be careful when doing this, as it may change numerical results.

This sets the LLVM Fast-Math flags, and corresponds to the `-ffast-math` option in clang. See the notes on performance annotations for more details.

Examples

```
julia> @fastmath 1+2
3

julia> @fastmath(sin(3))
0.1411200080598672
```

source

44.3 自定义二元运算符

某些 unicode 字符可用于定义新的支持中缀表示法的二元运算符。例如，`⊗(x,y) = kron(x,y)` 定义 ⊗ (otimes) 为 Kronecker 积，并且可以通过中缀语法将它作为一个二元运算符调用：`C = A ⊗ B` 也可以使用常用的前缀语法 `C = ⊗(A,B)`。

其他支持这种扩展的字符包括 \odot ⊙ 和 \oplus ⊕

The complete list is in the parser code: https://github.com/JuliaLang/julia/blob/master/src/julia-parser.scm

像 * 一样解析的包括（按优先级排列）* / ÷ % & ⋅ ∘ × |\\| ∩ ∧ ⊗ ⊘ ⊙ ⊚ ⊛ ⊠ ⊡ ⊓ ∗ ∙ |⁄ □ □ □ ⋄ ⋆ □ □ □ □ □ ⅄ □ ⊍ ▷ □ □ □ □ 像 + 一样解析的包括 + - |\|| ⊕ ⊖ ⊞ ⊟ |++| ∪ ∨ ⊔ ± ∓ □ ∔ ≏ ⊎ ⊻ □ ⋎ □ □ ⧺ ⧻ □ 还有许多其他的与箭头、比较和幂相关的符号。

Chapter 45

标准数值类型

A type tree for all subtypes of `Number` in `Base` is shown below. Abstract types have been marked, the rest are concrete types.

```
Number  (Abstract Type)
├─ Complex
└─ Real  (Abstract Type)
   ├─ AbstractFloat  (Abstract Type)
   │  ├─ Float16
   │  ├─ Float32
   │  ├─ Float64
   │  └─ BigFloat
   ├─ Integer  (Abstract Type)
   │  ├─ Bool
   │  ├─ Signed  (Abstract Type)
   │  │  ├─ Int8
   │  │  ├─ Int16
   │  │  ├─ Int32
   │  │  ├─ Int64
   │  │  ├─ Int128
   │  │  └─ BigInt
   │  └─ Unsigned  (Abstract Type)
   │     ├─ UInt8
   │     ├─ UInt16
   │     ├─ UInt32
   │     ├─ UInt64
   │     └─ UInt128
   ├─ Rational
   └─ AbstractIrrational  (Abstract Type)
      └─ Irrational
```

抽象数值类型

`Core.Number` – Type.

```
Number
```

Abstract supertype for all number types.

source

Core.Real – Type.

```
Real <: Number
```

Abstract supertype for all real numbers.

source

Core.AbstractFloat – Type.

```
AbstractFloat <: Real
```

Abstract supertype for all floating point numbers.

source

Core.Integer – Type.

```
Integer <: Real
```

Abstract supertype for all integers.

source

Core.Signed – Type.

```
Signed <: Integer
```

Abstract supertype for all signed integers.

source

Core.Unsigned – Type.

```
Unsigned <: Integer
```

Abstract supertype for all unsigned integers.

source

Base.AbstractIrrational – Type.

```
AbstractIrrational <: Real
```

Number type representing an exact irrational value, which is automatically rounded to the correct precision in arithmetic operations with other numeric quantities.

Subtypes `MyIrrational <: AbstractIrrational` should implement at least `==(::MyIrrational, ::MyIrrational)`, `hash(x::MyIrrational, h::UInt)`, and `convert(::Type{F}, x::MyIrrational) where {F <: Union{BigFloat,Float3`

If a subtype is used to represent values that may occasionally be rational (e.g. a square-root type that represents √n for integers `n` will give a rational result when `n` is a perfect square), then it should also implement `isinteger`, `iszero`, `isone`, and `==` with `Real` values (since all of these default to `false` for `AbstractIrrational` types), as well as defining hash to equal that of the corresponding `Rational`.

source

具象数值类型

`Core.Float16` – Type.

```
Float16 <: AbstractFloat
```

16-bit floating point number type (IEEE 754 standard).

Binary format: 1 sign, 5 exponent, 10 fraction bits.

source

`Core.Float32` – Type.

```
Float32 <: AbstractFloat
```

32-bit floating point number type (IEEE 754 standard).

Binary format: 1 sign, 8 exponent, 23 fraction bits.

source

`Core.Float64` – Type.

```
Float64 <: AbstractFloat
```

64-bit floating point number type (IEEE 754 standard).

Binary format: 1 sign, 11 exponent, 52 fraction bits.

source

`Base.MPFR.BigFloat` – Type.

```
BigFloat <: AbstractFloat
```

Arbitrary precision floating point number type.

source

`Core.Bool` – Type.

```
Bool <: Integer
```

Boolean type, containing the values `true` and `false`.

`Bool` is a kind of number: `false` is numerically equal to `0` and `true` is numerically equal to `1`. Moreover, `false` acts as a multiplicative "strong zero":

```
julia> false == 0
true

julia> true == 1
true

julia> 0 * NaN
NaN

julia> false * NaN
0.0
```

See also: `digits`, `iszero`, `NaN`.

source

`Core.Int8` – Type.

```
Int8 <: Signed
```

8-bit signed integer type.

source

`Core.UInt8` – Type.

```
UInt8 <: Unsigned
```

8-bit unsigned integer type.

source

`Core.Int16` – Type.

```
Int16 <: Signed
```

16-bit signed integer type.

source

`Core.UInt16` – Type.

```
UInt16 <: Unsigned
```

16-bit unsigned integer type.

source

`Core.Int32` – Type.

```
Int32 <: Signed
```

32-bit signed integer type.

source

`Core.UInt32` – Type.

```
UInt32 <: Unsigned
```

32-bit unsigned integer type.

source

`Core.Int64` – Type.

```
Int64 <: Signed
```

64-bit signed integer type.

source

`Core.UInt64` – Type.

```
UInt64 <: Unsigned
```

64-bit unsigned integer type.

source

`Core.Int128` – Type.

```
Int128 <: Signed
```

128-bit signed integer type.

source

`Core.UInt128` – Type.

```
UInt128 <: Unsigned
```

128-bit unsigned integer type.

source

Base.GMP.BigInt – Type.

```
BigInt <: Signed
```

Arbitrary precision integer type.

source

Base.Complex – Type.

```
Complex{T<:Real} <: Number
```

Complex number type with real and imaginary part of type T.

ComplexF16, ComplexF32 and ComplexF64 are aliases for Complex{Float16}, Complex{Float32} and Complex{Float64} respectively.

See also: Real, complex, real.

source

Base.Rational – Type.

```
Rational{T<:Integer} <: Real
```

Rational number type, with numerator and denominator of type T. Rationals are checked for overflow.

source

Base.Irrational – Type.

```
Irrational{sym} <: AbstractIrrational
```

Number type representing an exact irrational value denoted by the symbol sym, such as π, □ and γ.

See also AbstractIrrational.

source

45.1 数据格式

Base.digits – Function.

```
digits([T<:Integer], n::Integer; base::T = 10, pad::Integer = 1)
```

Return an array with element type T (default `Int`) of the digits of n in the given base, optionally padded with zeros to a specified size. More significant digits are at higher indices, such that `n == sum(digits[k]*base^(k-1) for k=1:length(digits))`.

See also `ndigits`, `digits!`, and for base 2 also `bitstring`, `count_ones`.

Examples

```
julia> digits(10)
2-element Vector{Int64}:
 0
 1

julia> digits(10, base = 2)
4-element Vector{Int64}:
 0
 1
 0
 1

julia> digits(-256, base = 10, pad = 5)
5-element Vector{Int64}:
 -6
 -5
 -2
  0
  0

julia> n = rand(-999:999);

julia> n == evalpoly(13, digits(n, base = 13))
true
```

source

`Base.digits!` – Function.

```
digits!(array, n::Integer; base::Integer = 10)
```

Fills an array of the digits of n in the given base. More significant digits are at higher indices. If the array length is insufficient, the least significant digits are filled up to the array length. If the array length is excessive, the excess portion is filled with zeros.

Examples

```
julia> digits!([2, 2, 2, 2], 10, base = 2)
4-element Vector{Int64}:
 0
 1
```

```
 0
 1

julia> digits!([2, 2, 2, 2, 2, 2], 10, base = 2)
6-element Vector{Int64}:
 0
 1
 0
 1
 0
 0
```

source

Base.bitstring – Function.

```
bitstring(n)
```

A string giving the literal bit representation of a primitive type.

See also count_ones, count_zeros, digits.

Examples

```
julia> bitstring(Int32(4))
"00000000000000000000000000000100"

julia> bitstring(2.2)
"0100000000000001100110011001100110011001100110011001100110011010"
```

source

Base.parse – Function.

```
parse(::Type{Platform}, triplet::AbstractString)
```

Parses a string platform triplet back into a `Platform` object.

source

```
parse(type, str; base)
```

Parse a string as a number. For `Integer` types, a base can be specified (the default is 10). For floating-point types, the string is parsed as a decimal floating-point number. `Complex` types are parsed from decimal strings of the form `"R±Iim"` as a `Complex(R,I)` of the requested type; `"i"` or `"j"` can also be used instead of `"im"`, and `"R"` or `"Iim"` are also permitted. If the string does not contain a valid number, an error is raised.

Julia 1.1

`parse(Bool, str)` requires at least Julia 1.1.

Examples

```
julia> parse(Int, "1234")
1234

julia> parse(Int, "1234", base = 5)
194

julia> parse(Int, "afc", base = 16)
2812

julia> parse(Float64, "1.2e-3")
0.0012

julia> parse(Complex{Float64}, "3.2e-1 + 4.5im")
0.32 + 4.5im
```

source

`Base.tryparse` – Function.

```
tryparse(type, str; base)
```

Like `parse`, but returns either a value of the requested type, or `nothing` if the string does not contain a valid number.

source

`Base.big` – Function.

```
big(x)
```

Convert a number to a maximum precision representation (typically `BigInt` or `BigFloat`). See `BigFloat` for information about some pitfalls with floating-point numbers.

source

`Base.signed` – Function.

```
signed(T::Integer)
```

Convert an integer bitstype to the signed type of the same size.

Examples

```
julia> signed(UInt16)
Int16
julia> signed(UInt64)
Int64
```

source

```
signed(x)
```

Convert a number to a signed integer. If the argument is unsigned, it is reinterpreted as signed without checking for overflow.

See also: `unsigned`, `sign`, `signbit`.

source

`Base.unsigned` – Function.

```
unsigned(T::Integer)
```

Convert an integer bitstype to the unsigned type of the same size.

Examples

```
julia> unsigned(Int16)
UInt16
julia> unsigned(UInt64)
UInt64
```

source

`Base.float` – Method.

```
float(x)
```

Convert a number or array to a floating point data type.

See also: `complex`, `oftype`, `convert`.

Examples

```
julia> float(1:1000)
1.0:1.0:1000.0

julia> float(typemax(Int32))
2.147483647e9
```

source

`Base.Math.significand` – Function.

```
significand(x)
```

Extract the significand (a.k.a. mantissa) of a floating-point number. If x is a non-zero finite number, then the result will be a number of the same type and sign as x, and whose absolute value is on the interval $[1, 2)$. Otherwise x is returned.

Examples

```
julia> significand(15.2)
1.9

julia> significand(-15.2)
-1.9

julia> significand(-15.2) * 2^3
-15.2

julia> significand(-Inf), significand(Inf), significand(NaN)
(-Inf, Inf, NaN)
```

source

`Base.Math.exponent` – Function.

```
exponent(x) -> Int
```

Returns the largest integer y such that `2^y ≤ abs(x)`. For a normalized floating-point number x, this corresponds to the exponent of x.

Examples

```
julia> exponent(8)
3

julia> exponent(64//1)
6

julia> exponent(6.5)
2

julia> exponent(16.0)
4

julia> exponent(3.142e-4)
-12
```

source

`Base.complex` – Method.

```
complex(r, [i])
```

Convert real numbers or arrays to complex. `i` defaults to zero.

Examples

```
julia> complex(7)
7 + 0im

julia> complex([1, 2, 3])
3-element Vector{Complex{Int64}}:
 1 + 0im
 2 + 0im
 3 + 0im
```

source

`Base.bswap` – Function.

```
bswap(n)
```

Reverse the byte order of `n`.

(See also `ntoh` and `hton` to convert between the current native byte order and big-endian order.)

Examples

```
julia> a = bswap(0x10203040)
0x40302010

julia> bswap(a)
0x10203040

julia> string(1, base = 2)
"1"

julia> string(bswap(1), base = 2)
"100000000000000000000000000000000000000000000000000000000"
```

source

`Base.hex2bytes` – Function.

```
hex2bytes(itr)
```

Given an iterable `itr` of ASCII codes for a sequence of hexadecimal digits, returns a `Vector{UInt8}` of bytes corresponding to the binary representation: each successive pair of hexadecimal digits in `itr` gives the value of one byte in the return vector.

The length of `itr` must be even, and the returned array has half of the length of `itr`. See also `hex2bytes!` for an in-place version, and `bytes2hex` for the inverse.

> **Julia 1.7**
>
> Calling `hex2bytes` with iterators producing `UInt8` values requires Julia 1.7 or later. In earlier versions, you can `collect` the iterator before calling `hex2bytes`.

Examples

```
julia> s = string(12345, base = 16)
"3039"

julia> hex2bytes(s)
2-element Vector{UInt8}:
 0x30
 0x39

julia> a = b"01abEF"
6-element Base.CodeUnits{UInt8, String}:
 0x30
 0x31
 0x61
 0x62
 0x45
 0x46

julia> hex2bytes(a)
3-element Vector{UInt8}:
 0x01
 0xab
 0xef
```

source

`Base.hex2bytes!` – Function.

```
hex2bytes!(dest::AbstractVector{UInt8}, itr)
```

Convert an iterable `itr` of bytes representing a hexadecimal string to its binary representation, similar to `hex2bytes` except that the output is written in-place to `dest`. The length of `dest` must be half the length of `itr`.

> **Julia 1.7**
>
> Calling hex2bytes! with iterators producing UInt8 requires version 1.7. In earlier versions, you can collect the iterable before calling instead.

source

`Base.bytes2hex` – Function.

```
bytes2hex(itr) -> String
bytes2hex(io::IO, itr)
```

Convert an iterator `itr` of bytes to its hexadecimal string representation, either returning a `String` via `bytes2hex(itr)` or writing the string to an `io` stream via `bytes2hex(io, itr)`. The hexadecimal characters are all lowercase.

Julia 1.7

Calling `bytes2hex` with arbitrary iterators producing `UInt8` values requires Julia 1.7 or later. In earlier versions, you can `collect` the iterator before calling `bytes2hex`.

Examples

```
julia> a = string(12345, base = 16)
"3039"

julia> b = hex2bytes(a)
2-element Vector{UInt8}:
 0x30
 0x39

julia> bytes2hex(b)
"3039"
```

source

45.2 常用数值函数和常量

`Base.one` – Function.

```
one(x)
one(T::type)
```

Return a multiplicative identity for `x`: a value such that `one(x)*x == x*one(x) == x`. Alternatively `one(T)` can take a type `T`, in which case `one` returns a multiplicative identity for any `x` of type `T`.

If possible, `one(x)` returns a value of the same type as `x`, and `one(T)` returns a value of type `T`. However, this may not be the case for types representing dimensionful quantities (e.g. time in days), since the multiplicative identity must be dimensionless. In that case, `one(x)` should return an identity value of the same precision (and shape, for matrices) as `x`.

If you want a quantity that is of the same type as `x`, or of type `T`, even if `x` is dimensionful, use `oneunit` instead.

See also the `identity` function, and `I` in `LinearAlgebra` for the identity matrix.

Examples

```
julia> one(3.7)
1.0

julia> one(Int)
1

julia> import Dates; one(Dates.Day(1))
1
```

source

Base.oneunit – Function.

```
oneunit(x::T)
oneunit(T::Type)
```

Return T(one(x)), where T is either the type of the argument or (if a type is passed) the argument. This differs from one for dimensionful quantities: one is dimensionless (a multiplicative identity) while oneunit is dimensionful (of the same type as x, or of type T).

Examples

```
julia> oneunit(3.7)
1.0

julia> import Dates; oneunit(Dates.Day)
1 day
```

source

Base.zero – Function.

```
zero(x)
zero(::Type)
```

Get the additive identity element for the type of x (x can also specify the type itself).

See also iszero, one, oneunit, oftype.

Examples

```
julia> zero(1)
0

julia> zero(big"2.0")
0.0

julia> zero(rand(2,2))
2×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0
```

source

Base.im – Constant.

```
im
```

The imaginary unit.

See also: imag, angle, complex.

Examples

```
julia> im * im
-1 + 0im

julia> (2.0 + 3im)^2
-5.0 + 12.0im
```

source

Base.MathConstants.pi – Constant.

```
π
pi
```

The constant pi.

Unicode π can be typed by writing \pi then pressing tab in the Julia REPL, and in many editors.

See also: sinpi, sincospi, deg2rad.

Examples

```
julia> pi
π = 3.1415926535897...

julia> 1/2pi
0.15915494309189535
```

source

Base.MathConstants.ℯ – Constant.

```
ℯ
e
```

The constant e.

Unicode ℯ can be typed by writing \euler and pressing tab in the Julia REPL, and in many editors.

See also: exp, cis, cispi.

Examples

```
julia> ℯ
ℯ = 2.7182818284590...

julia> log(ℯ)
1

julia> ℯ^(im)π ≈ -1
true
```

source

Base.MathConstants.catalan – Constant.

```
catalan
```

Catalan's constant.

Examples

```
julia> Base.MathConstants.catalan
catalan = 0.9159655941772...

julia> sum(log(x)/(1+x^2) for x in 1:0.01:10^6) * 0.01
0.9159466120554123
```

source

Base.MathConstants.eulergamma – Constant.

```
γ
eulergamma
```

Euler's constant.

Examples

```
julia> Base.MathConstants.eulergamma
γ = 0.5772156649015...

julia> dx = 10^-6;

julia> sum(-exp(-x) * log(x) for x in dx:dx:100) * dx
0.5772078382499133
```

source

Base.MathConstants.golden – Constant.

```
φ
golden
```

The golden ratio.

Examples

```
julia> Base.MathConstants.golden
φ = 1.6180339887498...

julia> (2ans - 1)^2 ≈ 5
true
```

source

Base.Inf – Constant.

```
Inf, Inf64
```

Positive infinity of type Float64.

See also: isfinite, typemax, NaN, Inf32.

Examples

```
julia> π/0
Inf

julia> +1.0 / -0.0
-Inf

julia> ℯ^-Inf
0.0
```

source

Base.Inf64 – Constant.

```
Inf, Inf64
```

Positive infinity of type Float64.

See also: isfinite, typemax, NaN, Inf32.

Examples

```
julia> π/0
Inf
```

```
julia> +1.0 / -0.0
-Inf

julia> □^-Inf
0.0
```

source

Base.Inf32 – Constant.

```
Inf32
```

Positive infinity of type Float32.

source

Base.Inf16 – Constant.

```
Inf16
```

Positive infinity of type Float16.

source

Base.NaN – Constant.

```
NaN, NaN64
```

A not-a-number value of type Float64.

See also: isnan, missing, NaN32, Inf.

Examples

```
julia> 0/0
NaN

julia> Inf - Inf
NaN

julia> NaN == NaN, isequal(NaN, NaN), NaN === NaN
(false, true, true)
```

source

Base.NaN64 – Constant.

```
NaN, NaN64
```

A not-a-number value of type `Float64`.

See also: `isnan`, `missing`, `NaN32`, `Inf`.

Examples

```
julia> 0/0
NaN

julia> Inf - Inf
NaN

julia> NaN == NaN, isequal(NaN, NaN), NaN === NaN
(false, true, true)
```

source

`Base.NaN32` – Constant.

```
NaN32
```

A not-a-number value of type `Float32`.

source

`Base.NaN16` – Constant.

```
NaN16
```

A not-a-number value of type `Float16`.

source

`Base.issubnormal` – Function.

```
issubnormal(f) -> Bool
```

Test whether a floating point number is subnormal.

An IEEE floating point number is subnormal when its exponent bits are zero and its significand is not zero.

Examples

```
julia> floatmin(Float32)
1.1754944f-38

julia> issubnormal(1.0f-37)
```

```
false

julia> issubnormal(1.0f-38)
true
```

source

Base.isfinite – Function.

```
isfinite(f) -> Bool
```

Test whether a number is finite.

Examples

```
julia> isfinite(5)
true

julia> isfinite(NaN32)
false
```

source

Base.isinf – Function.

```
isinf(f) -> Bool
```

Test whether a number is infinite.

See also: `Inf`, `iszero`, `isfinite`, `isnan`.

source

Base.isnan – Function.

```
isnan(f) -> Bool
```

Test whether a number value is a NaN, an indeterminate value which is neither an infinity nor a finite number (”not a number”).

See also: `iszero`, `isone`, `isinf`, `ismissing`.

source

Base.iszero – Function.

```
iszero(x)
```

Return `true` if `x == zero(x)`; if `x` is an array, this checks whether all of the elements of `x` are zero.

See also: `isone`, `isinteger`, `isfinite`, `isnan`.

Examples

```
julia> iszero(0.0)
true

julia> iszero([1, 9, 0])
false

julia> iszero([false, 0, 0])
true
```

source

`Base.isone` – Function.

```
isone(x)
```

Return `true` if `x == one(x)`; if `x` is an array, this checks whether `x` is an identity matrix.

Examples

```
julia> isone(1.0)
true

julia> isone([1 0; 0 2])
false

julia> isone([1 0; 0 true])
true
```

source

`Base.nextfloat` – Function.

```
nextfloat(x::AbstractFloat, n::Integer)
```

The result of `n` iterative applications of `nextfloat` to `x` if `n >= 0`, or `-n` applications of `prevfloat` if `n < 0`.

source

```
nextfloat(x::AbstractFloat)
```

Return the smallest floating point number `y` of the same type as `x` such `x < y`. If no such `y` exists (e.g. if `x` is `Inf` or `NaN`), then return `x`.

See also: `prevfloat`, `eps`, `issubnormal`.

source

`Base.prevfloat` – Function.

```
prevfloat(x::AbstractFloat, n::Integer)
```

The result of `n` iterative applications of `prevfloat` to `x` if `n >= 0`, or `-n` applications of `nextfloat` if `n < 0`.

source

```
prevfloat(x::AbstractFloat)
```

Return the largest floating point number `y` of the same type as `x` such `y < x`. If no such `y` exists (e.g. if `x` is `-Inf` or `NaN`), then return `x`.

source

`Base.isinteger` – Function.

```
isinteger(x) -> Bool
```

Test whether `x` is numerically equal to some integer.

Examples

```
julia> isinteger(4.0)
true
```

source

`Base.isreal` – Function.

```
isreal(x) -> Bool
```

Test whether `x` or all its elements are numerically equal to some real number including infinities and NaNs. `isreal(x)` is true if `isequal(x, real(x))` is true.

Examples

```
julia> isreal(5.)
true

julia> isreal(1 - 3im)
false

julia> isreal(Inf + 0im)
true

julia> isreal([4.; complex(0,1)])
false
```

source

`Core.Float32` – Method.

```
Float32(x [, mode::RoundingMode])
```

Create a `Float32` from `x`. If `x` is not exactly representable then `mode` determines how `x` is rounded.

Examples

```
julia> Float32(1/3, RoundDown)
0.3333333f0

julia> Float32(1/3, RoundUp)
0.33333334f0
```

See `RoundingMode` for available rounding modes.

source

`Core.Float64` – Method.

```
Float64(x [, mode::RoundingMode])
```

Create a `Float64` from `x`. If `x` is not exactly representable then `mode` determines how `x` is rounded.

Examples

```
julia> Float64(pi, RoundDown)
3.141592653589793

julia> Float64(pi, RoundUp)
3.1415926535897936
```

See `RoundingMode` for available rounding modes.

source

`Base.Rounding.rounding` – Function.

```
rounding(T)
```

Get the current floating point rounding mode for type `T`, controlling the rounding of basic arithmetic functions (`+`, `-`, `*`, `/` and `sqrt`) and type conversion.

See `RoundingMode` for available modes.

source

`Base.Rounding.setrounding` – Method.

```
setrounding(T, mode)
```

Set the rounding mode of floating point type T, controlling the rounding of basic arithmetic functions (+, -, *, / and sqrt) and type conversion. Other numerical functions may give incorrect or invalid values when using rounding modes other than the default RoundNearest.

Note that this is currently only supported for T == BigFloat.

Warning

This function is not thread-safe. It will affect code running on all threads, but its behavior is undefined if called concurrently with computations that use the setting.

source

Base.Rounding.setrounding – Method.

```
setrounding(f::Function, T, mode)
```

Change the rounding mode of floating point type T for the duration of f. It is logically equivalent to:

```
old = rounding(T)
setrounding(T, mode)
f()
setrounding(T, old)
```

See RoundingMode for available rounding modes.

source

Base.Rounding.get_zero_subnormals – Function.

```
get_zero_subnormals() -> Bool
```

Return false if operations on subnormal floating-point values ("denormals") obey rules for IEEE arithmetic, and true if they might be converted to zeros.

Warning

This function only affects the current thread.

source

Base.Rounding.set_zero_subnormals – Function.

```
set_zero_subnormals(yes::Bool) -> Bool
```

If yes is `false`, subsequent floating-point operations follow rules for IEEE arithmetic on subnormal values ("denormals"). Otherwise, floating-point operations are permitted (but not required) to convert subnormal inputs or outputs to zero. Returns `true` unless `yes==true` but the hardware does not support zeroing of subnormal numbers.

`set_zero_subnormals(true)` can speed up some computations on some hardware. However, it can break identities such as `(x-y==0) == (x==y)`.

> **Warning**
>
> This function only affects the current thread.

source

整型

`Base.count_ones` – Function.

```
count_ones(x::Integer) -> Integer
```

Number of ones in the binary representation of x.

Examples

```
julia> count_ones(7)
3

julia> count_ones(Int32(-1))
32
```

source

`Base.count_zeros` – Function.

```
count_zeros(x::Integer) -> Integer
```

Number of zeros in the binary representation of x.

Examples

```
julia> count_zeros(Int32(2 ^ 16 - 1))
16

julia> count_zeros(-1)
0
```

source

`Base.leading_zeros` – Function.

```
leading_zeros(x::Integer) -> Integer
```

Number of zeros leading the binary representation of x.

Examples

```
julia> leading_zeros(Int32(1))
31
```

source

Base.leading_ones – Function.

```
leading_ones(x::Integer) -> Integer
```

Number of ones leading the binary representation of x.

Examples

```
julia> leading_ones(UInt32(2 ^ 32 - 2))
31
```

source

Base.trailing_zeros – Function.

```
trailing_zeros(x::Integer) -> Integer
```

Number of zeros trailing the binary representation of x.

Examples

```
julia> trailing_zeros(2)
1
```

source

Base.trailing_ones – Function.

```
trailing_ones(x::Integer) -> Integer
```

Number of ones trailing the binary representation of x.

Examples

```
julia> trailing_ones(3)
2
```

source

Base.isodd – Function.

```
isodd(x::Number) -> Bool
```

Return `true` if `x` is an odd integer (that is, an integer not divisible by 2), and `false` otherwise.

Julia 1.7

Non-`Integer` arguments require Julia 1.7 or later.

Examples

```
julia> isodd(9)
true

julia> isodd(10)
false
```

source

Base.iseven – Function.

```
iseven(x::Number) -> Bool
```

Return `true` if `x` is an even integer (that is, an integer divisible by 2), and `false` otherwise.

Julia 1.7

Non-`Integer` arguments require Julia 1.7 or later.

Examples

```
julia> iseven(9)
false

julia> iseven(10)
true
```

source

Core.@int128_str – Macro.

```
@int128_str str
```

Parse `str` as an `Int128`. Throw an `ArgumentError` if the string is not a valid integer.

Examples

```
julia> int128"123456789123"
123456789123

julia> int128"123456789123.4"
ERROR: LoadError: ArgumentError: invalid base 10 digit '.' in "123456789123.4"
[...]
```

source

`Core.@uint128_str` – Macro.

```
@uint128_str str
```

Parse `str` as an `UInt128`. Throw an `ArgumentError` if the string is not a valid integer.

Examples

```
julia> uint128"123456789123"
0x00000000000000000000001cbe991a83

julia> uint128"-123456789123"
ERROR: LoadError: ArgumentError: invalid base 10 digit '-' in "-123456789123"
[...]
```

source

45.3 BigFloats and BigInts

The `BigFloat` and `BigInt` types implements arbitrary-precision floating point and integer arithmetic, respectively. For `BigFloat` the GNU MPFR library is used, and for `BigInt` the GNU Multiple Precision Arithmetic Library (GMP) is used.

`Base.MPFR.BigFloat` – Method.

```
BigFloat(x::Union{Real, AbstractString} [, rounding::RoundingMode=rounding(BigFloat)];
↪  [precision::Integer=precision(BigFloat)])
```

Create an arbitrary precision floating point number from `x`, with precision `precision`. The `rounding` argument specifies the direction in which the result should be rounded if the conversion cannot be done exactly. If not provided, these are set by the current global values.

BigFloat(x::Real) is the same as convert(BigFloat,x), except if x itself is already BigFloat, in which case it will return a value with the precision set to the current global precision; convert will always return x.

BigFloat(x::AbstractString) is identical to parse. This is provided for convenience since decimal literals are converted to Float64 when parsed, so BigFloat(2.1) may not yield what you expect.

See also:

- @big_str
- rounding and setrounding
- precision and setprecision

Julia 1.1

precision as a keyword argument requires at least Julia 1.1. In Julia 1.0 precision is the second positional argument (BigFloat(x, precision)).

Examples

```
julia> BigFloat(2.1) # 2.1 here is a Float64
2.100000000000000088817841970012523233890533447265625

julia> BigFloat("2.1") # the closest BigFloat to 2.1
2.099999999999999999999999999999999999999999999999999999999999999999999999999986

julia> BigFloat("2.1", RoundUp)
2.100000000000000000000000000000000000000000000000000000000000000000000000000021

julia> BigFloat("2.1", RoundUp, precision=128)
2.100000000000000000000000000000000000007
```

source

Base.precision – Function.

```
precision(num::AbstractFloat; base::Integer=2)
precision(T::Type; base::Integer=2)
```

Get the precision of a floating point number, as defined by the effective number of bits in the significand, or the precision of a floating-point type T (its current default, if T is a variable-precision type like BigFloat).

If base is specified, then it returns the maximum corresponding number of significand digits in that base.

Julia 1.8

The base keyword requires at least Julia 1.8.

source

Base.MPFR.setprecision – Function.

```
setprecision([T=BigFloat,] precision::Int; base=2)
```

Set the precision (in bits, by default) to be used for T arithmetic. If `base` is specified, then the precision is the minimum required to give at least `precision` digits in the given `base`.

> **Warning**
>
> This function is not thread-safe. It will affect code running on all threads, but its behavior is undefined if called concurrently with computations that use the setting.

> **Julia 1.8**
>
> The `base` keyword requires at least Julia 1.8.

source

```
setprecision(f::Function, [T=BigFloat,] precision::Integer; base=2)
```

Change the T arithmetic precision (in the given `base`) for the duration of `f`. It is logically equivalent to:

```
old = precision(BigFloat)
setprecision(BigFloat, precision)
f()
setprecision(BigFloat, old)
```

Often used as `setprecision(T, precision) do ... end`

Note: `nextfloat()`, `prevfloat()` do not use the precision mentioned by `setprecision`.

> **Julia 1.8**
>
> The `base` keyword requires at least Julia 1.8.

source

`Base.GMP.BigInt` – Method.

```
BigInt(x)
```

Create an arbitrary precision integer. `x` may be an `Int` (or anything that can be converted to an `Int`). The usual mathematical operators are defined for this type, and results are promoted to a `BigInt`.

Instances can be constructed from strings via `parse`, or using the `big` string literal.

Examples

```
julia> parse(BigInt, "42")
42

julia> big"313"
313

julia> BigInt(10)^19
10000000000000000000
```

source

Core.@big_str – Macro.

```
@big_str str
```

Parse a string into a BigInt or BigFloat, and throw an ArgumentError if the string is not a valid number. For integers _ is allowed in the string as a separator.

Examples

```
julia> big"123_456"
123456

julia> big"7891.5"
7891.5

julia> big"_"
ERROR: ArgumentError: invalid number format _ for BigInt or BigFloat
[...]
```

source

Chapter 46

字符串

`Core.AbstractString` – Type.

The `AbstractString` type is the supertype of all string implementations in Julia. Strings are encodings of sequences of Unicode code points as represented by the `AbstractChar` type. Julia makes a few assumptions about strings:

- Strings are encoded in terms of fixed-size ”code units”
  - Code units can be extracted with `codeunit(s, i)`
  - The first code unit has index 1
  - The last code unit has index `ncodeunits(s)`
  - Any index `i` such that `1 ≤ i ≤ ncodeunits(s)` is in bounds
- String indexing is done in terms of these code units:
  - Characters are extracted by `s[i]` with a valid string index `i`
  - Each `AbstractChar` in a string is encoded by one or more code units
  - Only the index of the first code unit of an `AbstractChar` is a valid index
  - The encoding of an `AbstractChar` is independent of what precedes or follows it
  - String encodings are [self-synchronizing] –i.e. `isvalid(s, i)` is O(1)

[self-synchronizing]: https://en.wikipedia.org/wiki/Self-synchronizing_code

Some string functions that extract code units, characters or substrings from strings error if you pass them out-of-bounds or invalid string indices. This includes `codeunit(s, i)` and `s[i]`. Functions that do string index arithmetic take a more relaxed approach to indexing and give you the closest valid string index when in-bounds, or when out-of-bounds, behave as if there were an infinite number of characters padding each side of the string. Usually these imaginary padding characters have code unit length 1 but string types may choose different ”imaginary” character sizes as makes sense for their implementations (e.g. substrings may pass index arithmetic through to the underlying string they provide a view into). Relaxed indexing functions include those intended for index arithmetic: `thisind`, `nextind` and `prevind`. This model allows index arithmetic to work with out-of- bounds indices as intermediate values so long as one never uses them to retrieve a character, which often helps avoid needing to code around edge cases.

See also `codeunit`, `ncodeunits`, `thisind`, `nextind`, `prevind`.

source

`Core.AbstractChar` – Type.

The AbstractChar type is the supertype of all character implementations in Julia. A character represents a Unicode code point, and can be converted to an integer via the codepoint function in order to obtain the numerical value of the code point, or constructed from the same integer. These numerical values determine how characters are compared with < and ==, for example. New T <: AbstractChar types should define a codepoint(::T) method and a T(::UInt32) constructor, at minimum.

A given AbstractChar subtype may be capable of representing only a subset of Unicode, in which case conversion from an unsupported UInt32 value may throw an error. Conversely, the built-in Char type represents a *superset* of Unicode (in order to losslessly encode invalid byte streams), in which case conversion of a non-Unicode value *to* UInt32 throws an error. The isvalid function can be used to check which codepoints are representable in a given AbstractChar type.

Internally, an AbstractChar type may use a variety of encodings. Conversion via codepoint(char) will not reveal this encoding because it always returns the Unicode value of the character. print(io, c) of any c::AbstractChar produces an encoding determined by io (UTF-8 for all built-in IO types), via conversion to Char if necessary.

write(io, c), in contrast, may emit an encoding depending on typeof(c), and read(io, typeof(c)) should read the same encoding as write. New AbstractChar types must provide their own implementations of write and read.

source

Core.Char – Type.

```
Char(c::Union{Number,AbstractChar})
```

Char is a 32-bit AbstractChar type that is the default representation of characters in Julia. Char is the type used for character literals like 'x' and it is also the element type of String.

In order to losslessly represent arbitrary byte streams stored in a String, a Char value may store information that cannot be converted to a Unicode codepoint —converting such a Char to UInt32 will throw an error. The isvalid(c::Char) function can be used to query whether c represents a valid Unicode character.

source

Base.codepoint – Function.

```
codepoint(c::AbstractChar) -> Integer
```

Return the Unicode codepoint (an unsigned integer) corresponding to the character c (or throw an exception if c does not represent a valid character). For Char, this is a UInt32 value, but AbstractChar types that represent only a subset of Unicode may return a different-sized integer (e.g. UInt8).

source

Base.length – Method.

```
length(s::AbstractString) -> Int
length(s::AbstractString, i::Integer, j::Integer) -> Int
```

Return the number of characters in string s from indices i through j.

This is computed as the number of code unit indices from i to j which are valid character indices. With only a single string argument, this computes the number of characters in the entire string. With i and j arguments it computes the number of indices between i and j inclusive that are valid indices in the string s. In addition to in-bounds values, i may take the out-of-bounds value ncodeunits(s) + 1 and j may take the out-of-bounds value 0.

Note

The time complexity of this operation is linear in general. That is, it will take the time proportional to the number of bytes or characters in the string because it counts the value on the fly. This is in contrast to the method for arrays, which is a constant-time operation.

See also isvalid, ncodeunits, lastindex, thisind, nextind, prevind.

Examples

```
julia> length("jμΛIα")
5
```

source

Base.sizeof – Method.

```
sizeof(str::AbstractString)
```

Size, in bytes, of the string str. Equal to the number of code units in str multiplied by the size, in bytes, of one code unit in str.

Examples

```
julia> sizeof("")
0

julia> sizeof("∀")
3
```

source

Base.:* – Method.

```
*(s::Union{AbstractString, AbstractChar}, t::Union{AbstractString, AbstractChar}...) ->
↪  AbstractString
```

Concatenate strings and/or characters, producing a String. This is equivalent to calling the string function on the arguments. Concatenation of built-in string types always produces a value of type String but other string types may choose to return a string of a different type as appropriate.

Examples

```
julia> "Hello " * "world"
"Hello world"

julia> 'j' * "ulia"
"julia"
```

source

Base.:^ – Method.

```
^(s::Union{AbstractString,AbstractChar}, n::Integer) -> AbstractString
```

Repeat a string or character n times. This can also be written as `repeat(s, n)`.

See also `repeat`.

Examples

```
julia> "Test "^3
"Test Test Test "
```

source

`Base.string` – Function.

```
string(n::Integer; base::Integer = 10, pad::Integer = 1)
```

Convert an integer n to a string in the given base, optionally specifying a number of digits to pad to.

See also `digits`, `bitstring`, `count_zeros`.

Examples

```
julia> string(5, base = 13, pad = 4)
"0005"

julia> string(-13, base = 5, pad = 4)
"-0023"
```

source

```
string(xs...)
```

Create a string from any values using the `print` function.

`string` should usually not be defined directly. Instead, define a method `print(io::IO, x::MyType)`. If `string(x)` for a certain type needs to be highly efficient, then it may make sense to add a method to `string` and define `print(io::IO, x::MyType) = print(io, string(x))` to ensure the functions are consistent.

See also: `String`, `repr`, `sprint`, `show`.

Examples

```
julia> string("a", 1, true)
"a1true"
```

source

`Base.repeat` – Method.

```
repeat(s::AbstractString, r::Integer)
```

Repeat a string `r` times. This can be written as `s^r`.

See also `^`.

Examples

```
julia> repeat("ha", 3)
"hahaha"
```

source

`Base.repeat` – Method.

```
repeat(c::AbstractChar, r::Integer) -> String
```

Repeat a character `r` times. This can equivalently be accomplished by calling `c^r`.

Examples

```
julia> repeat('A', 3)
"AAA"
```

source

`Base.repr` – Method.

```
repr(x; context=nothing)
```

Create a string from any value using the `show` function. You should not add methods to `repr`; define a `show` method instead.

The optional keyword argument `context` can be set to a `:key=>value` pair, a tuple of `:key=>value` pairs, or an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `show`.

Note that `repr(x)` is usually similar to how the value of `x` would be entered in Julia. See also `repr(MIME("text/plain"), x)` to instead return a "pretty-printed" version of `x` designed more for human consumption, equivalent to the REPL display of `x`.

Julia 1.7

Passing a tuple to keyword `context` requires Julia 1.7 or later.

Examples

```
julia> repr(1)
"1"

julia> repr(zeros(3))
"[0.0, 0.0, 0.0]"

julia> repr(big(1/3))
"0.333333333333333314829616256247390992939472198486328125"

julia> repr(big(1/3), context=:compact => true)
"0.333333"
```

source

`Core.String` – Method.

```
String(s::AbstractString)
```

Create a new `String` from an existing `AbstractString`.

source

`Base.SubString` – Type.

```
SubString(s::AbstractString, i::Integer, j::Integer=lastindex(s))
SubString(s::AbstractString, r::UnitRange{<:Integer})
```

Like `getindex`, but returns a view into the parent string `s` within range `i:j` or `r` respectively instead of making a copy.

The `@views` macro converts any string slices `s[i:j]` into substrings `SubString(s, i, j)` in a block of code.

Examples

```
julia> SubString("abc", 1, 2)
"ab"

julia> SubString("abc", 1:2)
"ab"

julia> SubString("abc", 2)
"bc"
```

source

`Base.LazyString` – Type.

```
LazyString <: AbstractString
```

A lazy representation of string interpolation. This is useful when a string needs to be constructed in a context where performing the actual interpolation and string construction is unnecessary or undesirable (e.g. in error paths of functions).

This type is designed to be cheap to construct at runtime, trying to offload as much work as possible to either the macro or later printing operations.

Examples

```
julia> n = 5; str = LazyString("n is ", n)
"n is 5"
```

See also `@lazy_str`.

> **Julia 1.8**
>
> `LazyString` requires Julia 1.8 or later.

Extended help

Safety properties for concurrent programs

A lazy string itself does not introduce any concurrency problems even if it is printed in multiple Julia tasks. However, if `print` methods on a captured value can have a concurrency issue when invoked without synchronizations, printing the lazy string may cause an issue. Furthermore, the `print` methods on the captured values may be invoked multiple times, though only exactly one result will be returned.

> **Julia 1.9**
>
> `LazyString` is safe in the above sense in Julia 1.9 and later.

source

`Base.@lazy_str` – Macro.

```
lazy"str"
```

Create a `LazyString` using regular string interpolation syntax. Note that interpolations are *evaluated* at LazyString construction time, but *printing* is delayed until the first access to the string.

See `LazyString` documentation for the safety properties for concurrent programs.

Examples

```
julia> n = 5; str = lazy"n is $n"
"n is 5"

julia> typeof(str)
LazyString
```

> **Julia 1.8**
>
> `lazy"str"` requires Julia 1.8 or later.

source

`Base.transcode` – Function.

```
transcode(T, src)
```

Convert string data between Unicode encodings. `src` is either a `String` or a `Vector{UIntXX}` of UTF-XX code units, where XX is 8, 16, or 32. T indicates the encoding of the return value: `String` to return a (UTF-8 encoded) `String` or `UIntXX` to return a `Vector{UIntXX}` of UTF-XX data. (The alias `Cwchar_t` can also be used as the integer type, for converting `wchar_t*` strings used by external C libraries.)

The `transcode` function succeeds as long as the input data can be reasonably represented in the target encoding; it always succeeds for conversions between UTF-XX encodings, even for invalid Unicode data.

Only conversion to/from UTF-8 is currently supported.

Examples

```
julia> str = "αβγ"
"αβγ"

julia> transcode(UInt16, str)
3-element Vector{UInt16}:
 0x03b1
 0x03b2
 0x03b3

julia> transcode(String, transcode(UInt16, str))
"αβγ"
```

source

`Base.unsafe_string` – Function.

```
unsafe_string(p::Ptr{UInt8}, [length::Integer])
```

Copy a string from the address of a C-style (NUL-terminated) string encoded as UTF-8. (The pointer can be safely freed afterwards.) If `length` is specified (the length of the data in bytes), the string does not have to be NUL-terminated.

This function is labeled ”unsafe” because it will crash if `p` is not a valid memory address to data of the requested length.

source

`Base.ncodeunits` – Method.

```
ncodeunits(s::AbstractString) -> Int
```

Return the number of code units in a string. Indices that are in bounds to access this string must satisfy 1 ≤ i ≤ ncodeunits(s). Not all such indices are valid —they may not be the start of a character, but they will return a code unit value when calling codeunit(s,i).

Examples

```
julia> ncodeunits("The Julia Language")
18

julia> ncodeunits("∫eˣ")
6

julia> ncodeunits('∫'), ncodeunits('e'), ncodeunits('ˣ')
(3, 1, 2)
```

See also codeunit, checkbounds, sizeof, length, lastindex.

source

Base.codeunit – Function.

```
codeunit(s::AbstractString) -> Type{<:Union{UInt8, UInt16, UInt32}}
```

Return the code unit type of the given string object. For ASCII, Latin-1, or UTF-8 encoded strings, this would be UInt8; for UCS-2 and UTF-16 it would be UInt16; for UTF-32 it would be UInt32. The code unit type need not be limited to these three types, but it's hard to think of widely used string encodings that don't use one of these units. codeunit(s) is the same as typeof(codeunit(s,1)) when s is a non-empty string.

See also ncodeunits.

source

```
codeunit(s::AbstractString, i::Integer) -> Union{UInt8, UInt16, UInt32}
```

Return the code unit value in the string s at index i. Note that

```
codeunit(s, i) :: codeunit(s)
```

I.e. the value returned by codeunit(s, i) is of the type returned by codeunit(s).

Examples

```
julia> a = codeunit("Hello", 2)
0x65

julia> typeof(a)
UInt8
```

See also ncodeunits, checkbounds.

source

Base.codeunits – Function.

```
codeunits(s::AbstractString)
```

Obtain a vector-like object containing the code units of a string. Returns a CodeUnits wrapper by default, but codeunits may optionally be defined for new string types if necessary.

Examples

```
julia> codeunits("Juλia")
6-element Base.CodeUnits{UInt8, String}:
 0x4a
 0x75
 0xce
 0xbb
 0x69
 0x61
```

source

Base.ascii – Function.

```
ascii(s::AbstractString)
```

Convert a string to String type and check that it contains only ASCII data, otherwise throwing an ArgumentError indicating the position of the first non-ASCII byte.

See also the isascii predicate to filter or replace non-ASCII characters.

Examples

```
julia> ascii("abcdeγfgh")
ERROR: ArgumentError: invalid ASCII at index 6 in "abcdeγfgh"
Stacktrace:
[...]

julia> ascii("abcdefgh")
"abcdefgh"
```

source

Base.Regex – Type.

```
Regex(pattern[, flags]) <: AbstractPattern
```

A type representing a regular expression. Regex objects can be used to match strings with `match`.

Regex objects can be created using the `@r_str` string macro. The `Regex(pattern[, flags])` constructor is usually used if the `pattern` string needs to be interpolated. See the documentation of the string macro for details on flags.

> **Note**
>
> To escape interpolated variables use `\Q` and `\E` (e.g. `Regex("\\Q$x\\E")`)

source

`Base.@r_str` – Macro.

```
@r_str -> Regex
```

Construct a regex, such as `r"^[a-z]*$"`, without interpolation and unescaping (except for quotation mark `"` which still has to be escaped). The regex also accepts one or more flags, listed after the ending quote, to change its behaviour:

- `i` enables case-insensitive matching
- `m` treats the `^` and `$` tokens as matching the start and end of individual lines, as opposed to the whole string.
- `s` allows the `.` modifier to match newlines.
- `x` enables "comment mode": whitespace is enabled except when escaped with `\`, and `#` is treated as starting a comment.
- `a` enables ASCII mode (disables `UTF` and `UCP` modes). By default `\B`, `\b`, `\D`, `\d`, `\S`, `\s`, `\W`, `\w`, etc. match based on Unicode character properties. With this option, these sequences only match ASCII characters. This includes `\u` also, which will emit the specified character value directly as a single byte, and not attempt to encode it into UTF-8. Importantly, this option allows matching against invalid UTF-8 strings, by treating both matcher and target as simple bytes (as if they were ISO/IEC 8859-1 / Latin-1 bytes) instead of as character encodings. In this case, this option is often combined with `s`. This option can be further refined by starting the pattern with (*UCP) or (*UTF).

See `Regex` if interpolation is needed.

Examples

```
julia> match(r"a+.*b+.*?d$"ism, "Goodbye,\nOh, angry,\nBad world\n")
RegexMatch("angry,\nBad world")
```

This regex has the first three flags enabled.

source

`Base.SubstitutionString` – Type.

```
SubstitutionString(substr) <: AbstractString
```

Stores the given string `substr` as a `SubstitutionString`, for use in regular expression substitutions. Most commonly constructed using the `@s_str` macro.

Examples

```
julia> SubstitutionString("Hello \\g<name>, it's \\1")
s"Hello \g<name>, it's \1"

julia> subst = s"Hello \g<name>, it's \1"
s"Hello \g<name>, it's \1"

julia> typeof(subst)
SubstitutionString{String}
```

source

`Base.@s_str` – Macro.

```
@s_str -> SubstitutionString
```

Construct a substitution string, used for regular expression substitutions. Within the string, sequences of the form `\N` refer to the Nth capture group in the regex, and `\g<groupname>` refers to a named capture group with name `groupname`.

Examples

```
julia> msg = "#Hello# from Julia";

julia> replace(msg, r"#(.+)# from (?<from>\w+)" => s"FROM: \g<from>; MESSAGE: \1")
"FROM: Julia; MESSAGE: Hello"
```

source

`Base.@raw_str` – Macro.

```
@raw_str -> String
```

Create a raw string without interpolation and unescaping. The exception is that quotation marks still must be escaped. Backslashes escape both quotation marks and other backslashes, but only when a sequence of backslashes precedes a quote character. Thus, 2n backslashes followed by a quote encodes n backslashes and the end of the literal while 2n+1 backslashes followed by a quote encodes n backslashes followed by a quote character.

Examples

```
julia> println(raw"\ $x")
\ $x

julia> println(raw"\"")
"
```

```
julia> println(raw"\\\"")
\"

julia> println(raw"\\x \\\"")
\\x \"
```

source

Base.@b_str – Macro.

```
@b_str
```

Create an immutable byte (UInt8) vector using string syntax.

Examples

```
julia> v = b"12\x01\x02"
4-element Base.CodeUnits{UInt8, String}:
 0x31
 0x32
 0x01
 0x02

julia> v[2]
0x32
```

source

Base.Docs.@html_str – Macro.

```
@html_str -> Docs.HTML
```

Create an HTML object from a literal string.

Examples

```
julia> html"Julia"
HTML{String}("Julia")
```

source

Base.Docs.@text_str – Macro.

```
@text_str -> Docs.Text
```

Create a Text object from a literal string.

Examples

```
julia> text"Julia"
Julia
```

source

Base.isvalid – Method.

```
isvalid(value) -> Bool
```

Return true if the given value is valid for its type, which currently can be either AbstractChar or String or SubString{String}.

Examples

```
julia> isvalid(Char(0xd800))
false

julia> isvalid(SubString(String(UInt8[0xfe,0x80,0x80,0x80,0x80,0x80]),1,2))
false

julia> isvalid(Char(0xd799))
true
```

source

Base.isvalid – Method.

```
isvalid(T, value) -> Bool
```

Return true if the given value is valid for that type. Types currently can be either AbstractChar or String. Values for AbstractChar can be of type AbstractChar or UInt32. Values for String can be of that type, SubString{String}, Vector{UInt8}, or a contiguous subarray thereof.

Examples

```
julia> isvalid(Char, 0xd800)
false

julia> isvalid(String, SubString("thisisvalid",1,5))
true

julia> isvalid(Char, 0xd799)
true
```

Julia 1.6

Support for subarray values was added in Julia 1.6.

source

Base.isvalid – Method.

```
isvalid(s::AbstractString, i::Integer) -> Bool
```

Predicate indicating whether the given index is the start of the encoding of a character in s or not. If isvalid(s, i) is true then s[i] will return the character whose encoding starts at that index, if it's false, then s[i] will raise an invalid index error or a bounds error depending on if i is in bounds. In order for isvalid(s, i) to be an O(1) function, the encoding of s must be self-synchronizing. This is a basic assumption of Julia's generic string support.

See also getindex, iterate, thisind, nextind, prevind, length.

Examples

```
julia> str = "αβγdef";

julia> isvalid(str, 1)
true

julia> str[1]
'α': Unicode U+03B1 (category Ll: Letter, lowercase)

julia> isvalid(str, 2)
false

julia> str[2]
ERROR: StringIndexError: invalid index [2], valid nearby indices [1]=>'α', [3]=>'β'
Stacktrace:
[...]
```

source

Base.match – Function.

```
match(r::Regex, s::AbstractString[, idx::Integer[, addopts]])
```

Search for the first match of the regular expression r in s and return a RegexMatch object containing the match, or nothing if the match failed. The matching substring can be retrieved by accessing m.match and the captured sequences can be retrieved by accessing m.captures The optional idx argument specifies an index at which to start the search.

Examples

```
julia> rx = r"a(.)a"
r"a(.)a"

julia> m = match(rx, "cabac")
RegexMatch("aba", 1="b")

julia> m.captures
1-element Vector{Union{Nothing, SubString{String}}}:
```

```
 "b"

julia> m.match
"aba"

julia> match(rx, "cabac", 3) === nothing
true
```

source

Base.eachmatch – Function.

```
eachmatch(r::Regex, s::AbstractString; overlap::Bool=false)
```

Search for all matches of the regular expression `r` in `s` and return an iterator over the matches. If `overlap` is `true`, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from distinct character ranges.

Examples

```
julia> rx = r"a.a"
r"a.a"

julia> m = eachmatch(rx, "a1a2a3a")
Base.RegexMatchIterator(r"a.a", "a1a2a3a", false)

julia> collect(m)
2-element Vector{RegexMatch}:
 RegexMatch("a1a")
 RegexMatch("a3a")

julia> collect(eachmatch(rx, "a1a2a3a", overlap = true))
3-element Vector{RegexMatch}:
 RegexMatch("a1a")
 RegexMatch("a2a")
 RegexMatch("a3a")
```

source

Base.RegexMatch – Type.

```
RegexMatch <: AbstractMatch
```

A type representing a single match to a Regex found in a string. Typically created from the `match` function.

The `match` field stores the substring of the entire matched string. The `captures` field stores the substrings for each capture group, indexed by number. To index by capture group name, the entire match object should be indexed instead, as shown in the examples. The location of the start of the match is stored in the `offset` field. The `offsets` field stores the locations of the start of each capture group, with 0 denoting a group that was not captured.

This type can be used as an iterator over the capture groups of the Regex, yielding the substrings captured in each group. Because of this, the captures of a match can be destructured. If a group was not captured, `nothing` will be yielded instead of a substring.

Methods that accept a `RegexMatch` object are defined for `iterate`, `length`, `eltype`, `keys`, `haskey`, and `getindex`, where keys are the the names or numbers of a capture group. See `keys` for more information.

Examples

```
julia> m = match(r"(?<hour>\d+):(?<minute>\d+)(am|pm)?", "11:30 in the morning")
RegexMatch("11:30", hour="11", minute="30", 3=nothing)

julia> m.match
"11:30"

julia> m.captures
3-element Vector{Union{Nothing, SubString{String}}}:
 "11"
 "30"
 nothing


julia> m["minute"]
"30"

julia> hr, min, ampm = m; # destructure capture groups by iteration

julia> hr
"11"
```

source

`Base.keys` – Method.

```
keys(m::RegexMatch) -> Vector
```

Return a vector of keys for all capture groups of the underlying regex. A key is included even if the capture group fails to match. That is, `idx` will be in the return value even if `m[idx] == nothing`.

Unnamed capture groups will have integer keys corresponding to their index. Named capture groups will have string keys.

Julia 1.7

This method was added in Julia 1.7

Examples

```
julia> keys(match(r"(?<hour>\d+):(?<minute>\d+)(am|pm)?", "11:30"))
3-element Vector{Any}:
  "hour"
  "minute"
 3
```

source

Base.isless – Method.

```
isless(a::AbstractString, b::AbstractString) -> Bool
```

Test whether string a comes before string b in alphabetical order (technically, in lexicographical order by Unicode code points).

Examples

```
julia> isless("a", "b")
true

julia> isless("β", "α")
false

julia> isless("a", "a")
false
```

source

Base.:== – Method.

```
==(a::AbstractString, b::AbstractString) -> Bool
```

Test whether two strings are equal character by character (technically, Unicode code point by code point).

Examples

```
julia> "abc" == "abc"
true

julia> "abc" == "αβγ"
false
```

source

Base.cmp – Method.

```
cmp(a::AbstractString, b::AbstractString) -> Int
```

Compare two strings. Return `0` if both strings have the same length and the character at each index is the same in both strings. Return `-1` if a is a prefix of b, or if a comes before b in alphabetical order. Return `1` if b is a prefix of a, or if b comes before a in alphabetical order (technically, lexicographical order by Unicode code points).

Examples

```
julia> cmp("abc", "abc")
0

julia> cmp("ab", "abc")
-1

julia> cmp("abc", "ab")
1

julia> cmp("ab", "ac")
-1

julia> cmp("ac", "ab")
1

julia> cmp("α", "a")
1

julia> cmp("b", "β")
-1
```

source

Base.lpad – Function.

```
lpad(s, n::Integer, p::Union{AbstractChar,AbstractString}=' ') -> String
```

Stringify s and pad the resulting string on the left with p to make it n characters (in textwidth) long. If s is already n characters long, an equal string is returned. Pad with spaces by default.

Examples

```
julia> lpad("March", 10)
"     March"
```

Julia 1.7

In Julia 1.7, this function was changed to use textwidth rather than a raw character (codepoint) count.

source

Base.rpad – Function.

```
rpad(s, n::Integer, p::Union{AbstractChar,AbstractString}=' ') -> String
```

Stringify s and pad the resulting string on the right with p to make it n characters (in textwidth) long. If s is already n characters long, an equal string is returned. Pad with spaces by default.

Examples

```
julia> rpad("March", 20)
"March               "
```

Julia 1.7

In Julia 1.7, this function was changed to use `textwidth` rather than a raw character (codepoint) count.

source

`Base.findfirst` – Method.

```
findfirst(pattern::AbstractString, string::AbstractString)
findfirst(pattern::AbstractPattern, string::String)
```

Find the first occurrence of `pattern` in `string`. Equivalent to `findnext(pattern, string, firstindex(s))`.

Examples

```
julia> findfirst("z", "Hello to the world") # returns nothing, but not printed in the REPL

julia> findfirst("Julia", "JuliaLang")
1:5
```

source

`Base.findnext` – Method.

```
findnext(pattern::AbstractString, string::AbstractString, start::Integer)
findnext(pattern::AbstractPattern, string::String, start::Integer)
```

Find the next occurrence of `pattern` in `string` starting at position `start`. `pattern` can be either a string, or a regular expression, in which case `string` must be of type `String`.

The return value is a range of indices where the matching sequence is found, such that `s[findnext(x, s, i)] == x`:

`findnext("substring", string, i) == start:stop` such that `string[start:stop] == "substring"` and `i <= start`, or `nothing` if unmatched.

Examples

```
julia> findnext("z", "Hello to the world", 1) === nothing
true

julia> findnext("o", "Hello to the world", 6)
8:8

julia> findnext("Lang", "JuliaLang", 2)
6:9
```

source

Base.findnext – Method.

```
findnext(ch::AbstractChar, string::AbstractString, start::Integer)
```

Find the next occurrence of character `ch` in `string` starting at position `start`.

Julia 1.3

This method requires at least Julia 1.3.

Examples

```
julia> findnext('z', "Hello to the world", 1) === nothing
true

julia> findnext('o', "Hello to the world", 6)
8
```

source

Base.findlast – Method.

```
findlast(pattern::AbstractString, string::AbstractString)
```

Find the last occurrence of `pattern` in `string`. Equivalent to `findprev(pattern, string, lastindex(string))`.

Examples

```
julia> findlast("o", "Hello to the world")
15:15

julia> findfirst("Julia", "JuliaLang")
1:5
```

source

Base.findlast – Method.

```
findlast(ch::AbstractChar, string::AbstractString)
```

Find the last occurrence of character `ch` in `string`.

Julia 1.3

This method requires at least Julia 1.3.

Examples

```
julia> findlast('p', "happy")
4

julia> findlast('z', "happy") === nothing
true
```

source

Base.findprev – Method.

```
findprev(pattern::AbstractString, string::AbstractString, start::Integer)
```

Find the previous occurrence of `pattern` in `string` starting at position `start`.

The return value is a range of indices where the matching sequence is found, such that `s[findprev(x, s, i)] == x`:

`findprev("substring", string, i) == start:stop` such that `string[start:stop] == "substring"` and `stop <= i`, or `nothing` if unmatched.

Examples

```
julia> findprev("z", "Hello to the world", 18) === nothing
true

julia> findprev("o", "Hello to the world", 18)
15:15

julia> findprev("Julia", "JuliaLang", 6)
1:5
```

source

Base.occursin – Function.

```
occursin(needle::Union{AbstractString,AbstractPattern,AbstractChar}, haystack::AbstractString)
```

Determine whether the first argument is a substring of the second. If `needle` is a regular expression, checks whether `haystack` contains a match.

Examples

```
julia> occursin("Julia", "JuliaLang is pretty cool!")
true

julia> occursin('a', "JuliaLang is pretty cool!")
true

julia> occursin(r"a.a", "aba")
true
```

```
julia> occursin(r"a.a", "abba")
false
```

See also `contains`.

source

```
occursin(haystack)
```

Create a function that checks whether its argument occurs in `haystack`, i.e. a function equivalent to `needle -> occursin(needle, haystack)`.

The returned function is of type `Base.Fix2{typeof(occursin)}`.

> **Julia 1.6**
>
> This method requires Julia 1.6 or later.

Examples

```
julia> search_f = occursin("JuliaLang is a programming language");

julia> search_f("JuliaLang")
true

julia> search_f("Python")
false
```

source

`Base.reverse` – Method.

```
reverse(s::AbstractString) -> AbstractString
```

Reverses a string. Technically, this function reverses the codepoints in a string and its main utility is for reversed-order string processing, especially for reversed regular-expression searches. See also `reverseind` to convert indices in `s` to indices in `reverse(s)` and vice-versa, and `graphemes` from module `Unicode` to operate on user-visible "characters" (graphemes) rather than codepoints. See also `Iterators.reverse` for reverse-order iteration without making a copy. Custom string types must implement the `reverse` function themselves and should typically return a string with the same type and encoding. If they return a string with a different encoding, they must also override `reverseind` for that string type to satisfy `s[reverseind(s,i)] == reverse(s)[i]`.

Examples

```
julia> reverse("JuliaLang")
"gnaLailuJ"
```

> **Note**
>
> The examples below may be rendered differently on different systems. The comments indicate how they're supposed to be rendered

Combining characters can lead to surprising results:

```
julia> reverse("axe") # hat is above x in the input, above e in the output
"êxa"

julia> using Unicode

julia> join(reverse(collect(graphemes("axe")))) # reverses graphemes; hat is above x in both in-
↪  and output
"exa"
```

source

Base.replace – Method.

```
replace([io::IO], s::AbstractString, pat=>r, [pat2=>r2, ...]; [count::Integer])
```

Search for the given pattern `pat` in `s`, and replace each occurrence with `r`. If `count` is provided, replace at most `count` occurrences. `pat` may be a single character, a vector or a set of characters, a string, or a regular expression. If `r` is a function, each occurrence is replaced with `r(s)` where `s` is the matched substring (when `pat` is a `AbstractPattern` or `AbstractString`) or character (when `pat` is an `AbstractChar` or a collection of `AbstractChar`). If `pat` is a regular expression and `r` is a `SubstitutionString`, then capture group references in `r` are replaced with the corresponding matched text. To remove instances of `pat` from `string`, set `r` to the empty `String` (`""`).

The return value is a new string after the replacements. If the `io::IO` argument is supplied, the transformed string is instead written to `io` (returning `io`). (For example, this can be used in conjunction with an `IOBuffer` to re-use a pre-allocated buffer array in-place.)

Multiple patterns can be specified, and they will be applied left-to-right simultaneously, so only one pattern will be applied to any character, and the patterns will only be applied to the input text, not the replacements.

> **Julia 1.7**
>
> Support for multiple patterns requires version 1.7.

> **Julia 1.10**
>
> The `io::IO` argument requires version 1.10.

Examples

```
julia> replace("Python is a programming language.", "Python" => "Julia")
"Julia is a programming language."
```

```
julia> replace("The quick foxes run quickly.", "quick" => "slow", count=1)
"The slow foxes run quickly."

julia> replace("The quick foxes run quickly.", "quick" => "", count=1)
"The  foxes run quickly."

julia> replace("The quick foxes run quickly.", r"fox(es)?" => s"bus\1")
"The quick buses run quickly."

julia> replace("abcabc", "a" => "b", "b" => "c", r".+" => "a")
"bca"
```

source

Base.eachsplit – Function.

```
eachsplit(str::AbstractString, dlm; limit::Integer=0, keepempty::Bool=true)
eachsplit(str::AbstractString; limit::Integer=0, keepempty::Bool=false)
```

Split `str` on occurrences of the delimiter(s) `dlm` and return an iterator over the substrings. `dlm` can be any of the formats allowed by `findnext`'s first argument (i.e. as a string, regular expression or a function), or as a single character or collection of characters.

If `dlm` is omitted, it defaults to `isspace`.

The optional keyword arguments are:

- `limit`: the maximum size of the result. `limit=0` implies no maximum (default)
- `keepempty`: whether empty fields should be kept in the result. Default is `false` without a `dlm` argument, `true` with a `dlm` argument.

See also `split`.

Julia 1.8

The `eachsplit` function requires at least Julia 1.8.

Examples

```
julia> a = "Ma.rch"
"Ma.rch"

julia> b = eachsplit(a, ".")
Base.SplitIterator{String, String}("Ma.rch", ".", 0, true)

julia> collect(b)
2-element Vector{SubString{String}}:
 "Ma"
 "rch"
```

source

Base.split – Function.

```
split(str::AbstractString, dlm; limit::Integer=0, keepempty::Bool=true)
split(str::AbstractString; limit::Integer=0, keepempty::Bool=false)
```

Split `str` into an array of substrings on occurrences of the delimiter(s) `dlm`. `dlm` can be any of the formats allowed by `findnext`'s first argument (i.e. as a string, regular expression or a function), or as a single character or collection of characters.

If `dlm` is omitted, it defaults to `isspace`.

The optional keyword arguments are:

- `limit`: the maximum size of the result. `limit=0` implies no maximum (default)
- `keepempty`: whether empty fields should be kept in the result. Default is `false` without a `dlm` argument, `true` with a `dlm` argument.

See also `rsplit`, `eachsplit`.

Examples

```
julia> a = "Ma.rch"
"Ma.rch"

julia> split(a, ".")
2-element Vector{SubString{String}}:
 "Ma"
 "rch"
```

source

Base.rsplit – Function.

```
rsplit(s::AbstractString; limit::Integer=0, keepempty::Bool=false)
rsplit(s::AbstractString, chars; limit::Integer=0, keepempty::Bool=true)
```

Similar to `split`, but starting from the end of the string.

Examples

```
julia> a = "M.a.r.c.h"
"M.a.r.c.h"

julia> rsplit(a, ".")
5-element Vector{SubString{String}}:
 "M"
 "a"
 "r"
 "c"
 "h"
```

```
julia> rsplit(a, "."; limit=1)
1-element Vector{SubString{String}}:
 "M.a.r.c.h"

julia> rsplit(a, "."; limit=2)
2-element Vector{SubString{String}}:
 "M.a.r.c"
 "h"
```

source

Base.strip – Function.

```
strip([pred=isspace,] str::AbstractString) -> SubString
strip(str::AbstractString, chars) -> SubString
```

Remove leading and trailing characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove leading and trailing whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, vector or set of characters.

See also `lstrip` and `rstrip`.

Julia 1.2

The method which accepts a predicate function requires Julia 1.2 or later.

Examples

```
julia> strip("{3, 5}\n", ['{', '}', '\n'])
"3, 5"
```

source

Base.lstrip – Function.

```
lstrip([pred=isspace,] str::AbstractString) -> SubString
lstrip(str::AbstractString, chars) -> SubString
```

Remove leading characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove leading whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, or a vector or set of characters.

See also `strip` and `rstrip`.

Examples

```
julia> a = lpad("March", 20)
"               March"

julia> lstrip(a)
"March"
```

source

Base.`rstrip` – Function.

```
rstrip([pred=isspace,] str::AbstractString) -> SubString
rstrip(str::AbstractString, chars) -> SubString
```

Remove trailing characters from `str`, either those specified by `chars` or those for which the function `pred` returns `true`.

The default behaviour is to remove trailing whitespace and delimiters: see `isspace` for precise details.

The optional `chars` argument specifies which characters to remove: it can be a single character, or a vector or set of characters.

See also `strip` and `lstrip`.

Examples

```
julia> a = rpad("March", 20)
"March               "

julia> rstrip(a)
"March"
```

source

Base.`startswith` – Function.

```
startswith(s::AbstractString, prefix::AbstractString)
```

Return `true` if `s` starts with `prefix`. If `prefix` is a vector or set of characters, test whether the first character of `s` belongs to that set.

See also `endswith`, `contains`.

Examples

```
julia> startswith("JuliaLang", "Julia")
true
```

source

```
startswith(io::IO, prefix::Union{AbstractString,Base.Chars})
```

Check if an IO object starts with a prefix. See also peek.

source

```
startswith(prefix)
```

Create a function that checks whether its argument starts with prefix, i.e. a function equivalent to y -> startswith(y, prefix).

The returned function is of type Base.Fix2{typeof(startswith)}, which can be used to implement specialized methods.

Julia 1.5

The single argument startswith(prefix) requires at least Julia 1.5.

Examples

```
julia> startswith("Julia")("JuliaLang")
true

julia> startswith("Julia")("Ends with Julia")
false
```

source

```
startswith(s::AbstractString, prefix::Regex)
```

Return true if s starts with the regex pattern, prefix.

Note

startswith does not compile the anchoring into the regular expression, but instead passes the anchoring as match_option to PCRE. If compile time is amortized, occursin(r"^...", s) is faster than startswith(s, r"...").

See also occursin and endswith.

Julia 1.2

This method requires at least Julia 1.2.

Examples

```
julia> startswith("JuliaLang", r"Julia|Romeo")
true
```

source

Base.endswith – Function.

```
endswith(s::AbstractString, suffix::AbstractString)
```

Return `true` if `s` ends with `suffix`. If `suffix` is a vector or set of characters, test whether the last character of `s` belongs to that set.

See also `startswith`, `contains`.

Examples

```
julia> endswith("Sunday", "day")
true
```

source

```
endswith(suffix)
```

Create a function that checks whether its argument ends with `suffix`, i.e. a function equivalent to `y -> endswith(y, suffix)`.

The returned function is of type `Base.Fix2{typeof(endswith)}`, which can be used to implement specialized methods.

Julia 1.5

The single argument `endswith(suffix)` requires at least Julia 1.5.

Examples

```
julia> endswith("Julia")("Ends with Julia")
true

julia> endswith("Julia")("JuliaLang")
false
```

source

```
endswith(s::AbstractString, suffix::Regex)
```

Return `true` if `s` ends with the regex pattern, `suffix`.

Note

`endswith` does not compile the anchoring into the regular expression, but instead passes the anchoring as `match_option` to PCRE. If compile time is amortized, `occursin(r"...$", s)` is faster than `endswith(s, r"...")`.

See also `occursin` and `startswith`.

Julia 1.2

This method requires at least Julia 1.2.

Examples

```
julia> endswith("JuliaLang", r"Lang|Roberts")
true
```

source

`Base.contains` – Function.

```
contains(haystack::AbstractString, needle)
```

Return `true` if `haystack` contains `needle`. This is the same as `occursin(needle, haystack)`, but is provided for consistency with `startswith(haystack, needle)` and `endswith(haystack, needle)`.

See also `occursin`, `in`, `issubset`.

Examples

```
julia> contains("JuliaLang is pretty cool!", "Julia")
true

julia> contains("JuliaLang is pretty cool!", 'a')
true

julia> contains("aba", r"a.a")
true

julia> contains("abba", r"a.a")
false
```

Julia 1.5

The `contains` function requires at least Julia 1.5.

source

```
contains(needle)
```

Create a function that checks whether its argument contains `needle`, i.e. a function equivalent to `haystack -> contains(haystack, needle)`.

The returned function is of type `Base.Fix2{typeof(contains)}`, which can be used to implement specialized methods.

source

Base.first – Method.

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first n characters of s.

Examples

```
julia> first("∀ϵ≠0: ϵ²>0", 0)
""

julia> first("∀ϵ≠0: ϵ²>0", 1)
"∀"

julia> first("∀ϵ≠0: ϵ²>0", 3)
"∀ϵ≠"
```

source

Base.last – Method.

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last n characters of s.

Examples

```
julia> last("∀ϵ≠0: ϵ²>0", 0)
""

julia> last("∀ϵ≠0: ϵ²>0", 1)
"0"

julia> last("∀ϵ≠0: ϵ²>0", 3)
"²>0"
```

source

Base.Unicode.uppercase – Function.

```
uppercase(c::AbstractChar)
```

Convert c to uppercase.

See also lowercase, titlecase.

Examples

```
julia> uppercase('a')
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

julia> uppercase('ê')
'Ê': Unicode U+00CA (category Lu: Letter, uppercase)
```

source

```
uppercase(s::AbstractString)
```

Return s with all characters converted to uppercase.

See also `lowercase`, `titlecase`, `uppercasefirst`.

Examples

```
julia> uppercase("Julia")
"JULIA"
```

source

`Base.Unicode.lowercase` – Function.

```
lowercase(c::AbstractChar)
```

Convert c to lowercase.

See also `uppercase`, `titlecase`.

Examples

```
julia> lowercase('A')
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> lowercase('Ö')
'ö': Unicode U+00F6 (category Ll: Letter, lowercase)
```

source

```
lowercase(s::AbstractString)
```

Return s with all characters converted to lowercase.

See also `uppercase`, `titlecase`, `lowercasefirst`.

Examples

```
julia> lowercase("STRINGS AND THINGS")
"strings and things"
```

source

Base.Unicode.titlecase – Function.

```
titlecase(c::AbstractChar)
```

Convert c to titlecase. This may differ from uppercase for digraphs, compare the example below.

See also uppercase, lowercase.

Examples

```
julia> titlecase('a')
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

julia> titlecase('□')
'□': Unicode U+01C5 (category Lt: Letter, titlecase)

julia> uppercase('□')
'□': Unicode U+01C4 (category Lu: Letter, uppercase)
```

source

```
titlecase(s::AbstractString; [wordsep::Function], strict::Bool=true) -> String
```

Capitalize the first character of each word in s; if `strict` is true, every other character is converted to lowercase, otherwise they are left unchanged. By default, all non-letters beginning a new grapheme are considered as word separators; a predicate can be passed as the `wordsep` keyword to determine which characters should be considered as word separators. See also `uppercasefirst` to capitalize only the first character in s.

See also `uppercase`, `lowercase`, `uppercasefirst`.

Examples

```
julia> titlecase("the JULIA programming language")
"The Julia Programming Language"

julia> titlecase("ISS - international space station", strict=false)
"ISS - International Space Station"

julia> titlecase("a-a b-b", wordsep = c->c==' ')
"A-a B-b"
```

source

Base.Unicode.uppercasefirst – Function.

```
uppercasefirst(s::AbstractString) -> String
```

Return s with the first character converted to uppercase (technically ”title case” for Unicode). See also `titlecase` to capitalize the first character of every word in s.

See also `lowercasefirst`, `uppercase`, `lowercase`, `titlecase`.

Examples

```
julia> uppercasefirst("python")
"Python"
```

source

`Base.Unicode.lowercasefirst` – Function.

```
lowercasefirst(s::AbstractString)
```

Return `s` with the first character converted to lowercase.

See also `uppercasefirst`, `uppercase`, `lowercase`, `titlecase`.

Examples

```
julia> lowercasefirst("Julia")
"julia"
```

source

`Base.join` – Function.

```
join([io::IO,] iterator [, delim [, last]])
```

Join any `iterator` into a single string, inserting the given delimiter (if any) between adjacent items. If `last` is given, it will be used instead of `delim` between the last two items. Each item of `iterator` is converted to a string via `print(io::IOBuffer, x)`. If `io` is given, the result is written to `io` rather than returned as a `String`.

Examples

```
julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
"apples, bananas and pineapples"

julia> join([1,2,3,4,5])
"12345"
```

source

`Base.chop` – Function.

```
chop(s::AbstractString; head::Integer = 0, tail::Integer = 1)
```

Remove the first head and the last `tail` characters from `s`. The call `chop(s)` removes the last character from `s`. If it is requested to remove more characters than `length(s)` then an empty string is returned.

See also `chomp`, `startswith`, `first`.

Examples

```
julia> a = "March"
"March"

julia> chop(a)
"Marc"

julia> chop(a, head = 1, tail = 2)
"ar"

julia> chop(a, head = 5, tail = 5)
""
```

source

`Base.chopprefix` – Function.

```
chopprefix(s::AbstractString, prefix::Union{AbstractString,Regex}) -> SubString
```

Remove the prefix `prefix` from `s`. If `s` does not start with `prefix`, a string equal to `s` is returned.

See also `chopsuffix`.

Julia 1.8

This function is available as of Julia 1.8.

Examples

```
julia> chopprefix("Hamburger", "Ham")
"burger"

julia> chopprefix("Hamburger", "hotdog")
"Hamburger"
```

source

`Base.chopsuffix` – Function.

```
chopsuffix(s::AbstractString, suffix::Union{AbstractString,Regex}) -> SubString
```

Remove the suffix `suffix` from `s`. If `s` does not end with `suffix`, a string equal to `s` is returned.

See also `chopprefix`.

Julia 1.8

This function is available as of Julia 1.8.

Examples

```
julia> chopsuffix("Hamburger", "er")
"Hamburg"

julia> chopsuffix("Hamburger", "hotdog")
"Hamburger"
```

source

`Base.chomp` – Function.

```
chomp(s::AbstractString) -> SubString
```

Remove a single trailing newline from a string.

See also chop.

Examples

```
julia> chomp("Hello\n")
"Hello"
```

source

`Base.thisind` – Function.

```
thisind(s::AbstractString, i::Integer) -> Int
```

If `i` is in bounds in `s` return the index of the start of the character whose encoding code unit `i` is part of. In other words, if `i` is the start of a character, return `i`; if `i` is not the start of a character, rewind until the start of a character and return that index. If `i` is equal to 0 or `ncodeunits(s)+1` return `i`. In all other cases throw `BoundsError`.

Examples

```
julia> thisind("α", 0)
0

julia> thisind("α", 1)
1

julia> thisind("α", 2)
1
```

```
julia> thisind("α", 3)
3

julia> thisind("α", 4)
ERROR: BoundsError: attempt to access 2-codeunit String at index [4]
[...]

julia> thisind("α", -1)
ERROR: BoundsError: attempt to access 2-codeunit String at index [-1]
[...]
```

source

`Base.nextind` – Function.

```
nextind(str::AbstractString, i::Integer, n::Integer=1) -> Int
```

- Case `n == 1`

  If `i` is in bounds in `s` return the index of the start of the character whose encoding starts after index `i`. In other words, if `i` is the start of a character, return the start of the next character; if `i` is not the start of a character, move forward until the start of a character and return that index. If `i` is equal to `0` return `1`. If `i` is in bounds but greater or equal to `lastindex(str)` return `ncodeunits(str)+1`. Otherwise throw `BoundsError`.

- Case `n > 1`

  Behaves like applying `n` times `nextind` for `n==1`. The only difference is that if `n` is so large that applying `nextind` would reach `ncodeunits(str)+1` then each remaining iteration increases the returned value by `1`. This means that in this case `nextind` can return a value greater than `ncodeunits(str)+1`.

- Case `n == 0`

  Return `i` only if `i` is a valid index in `s` or is equal to `0`. Otherwise `StringIndexError` or `BoundsError` is thrown.

Examples

```
julia> nextind("α", 0)
1

julia> nextind("α", 1)
3

julia> nextind("α", 3)
ERROR: BoundsError: attempt to access 2-codeunit String at index [3]
[...]

julia> nextind("α", 0, 2)
3

julia> nextind("α", 1, 2)
4
```

source

`Base.prevind` – Function.

```
prevind(str::AbstractString, i::Integer, n::Integer=1) -> Int
```

- Case `n == 1`

  If `i` is in bounds in `s` return the index of the start of the character whose encoding starts before index `i`. In other words, if `i` is the start of a character, return the start of the previous character; if `i` is not the start of a character, rewind until the start of a character and return that index. If `i` is equal to `1` return `0`. If `i` is equal to `ncodeunits(str)+1` return `lastindex(str)`. Otherwise throw `BoundsError`.
- Case `n > 1`

  Behaves like applying `n` times `prevind` for `n==1`. The only difference is that if `n` is so large that applying `prevind` would reach `0` then each remaining iteration decreases the returned value by `1`. This means that in this case `prevind` can return a negative value.
- Case `n == 0`

  Return `i` only if `i` is a valid index in `str` or is equal to `ncodeunits(str)+1`. Otherwise `StringIndexError` or `BoundsError` is thrown.

Examples

```
julia> prevind("α", 3)
1

julia> prevind("α", 1)
0

julia> prevind("α", 0)
ERROR: BoundsError: attempt to access 2-codeunit String at index [0]
[...]

julia> prevind("α", 2, 2)
0

julia> prevind("α", 2, 3)
-1
```

source

`Base.Unicode.textwidth` – Function.

```
textwidth(c)
```

Give the number of columns needed to print a character.

Examples

```
julia> textwidth('α')
1

julia> textwidth('⛵')
2
```

source

```
textwidth(s::AbstractString)
```

Give the number of columns needed to print a string.

Examples

```
julia> textwidth("March")
5
```

source

Base.isascii – Function.

```
isascii(c::Union{AbstractChar,AbstractString}) -> Bool
```

Test whether a character belongs to the ASCII character set, or whether this is true for all elements of a string.

Examples

```
julia> isascii('a')
true

julia> isascii('α')
false

julia> isascii("abc")
true

julia> isascii("αβγ")
false
```

For example, `isascii` can be used as a predicate function for `filter` or `replace` to remove or replace non-ASCII characters, respectively:

```
julia> filter(isascii, "abcdeγfgh") # discard non-ASCII chars
"abcdefgh"

julia> replace("abcdeγfgh", !isascii=>' ') # replace non-ASCII chars with spaces
"abcde fgh"
```

source

```
isascii(cu::AbstractVector{CU}) where {CU <: Integer} -> Bool
```

Test whether all values in the vector belong to the ASCII character set (0x00 to 0x7f). This function is intended to be used by other string implementations that need a fast ASCII check.

source

`Base.Unicode.iscntrl` – Function.

```
iscntrl(c::AbstractChar) -> Bool
```

Tests whether a character is a control character. Control characters are the non-printing characters of the Latin-1 subset of Unicode.

Examples

```
julia> iscntrl('\x01')
true

julia> iscntrl('a')
false
```

source

`Base.Unicode.isdigit` – Function.

```
isdigit(c::AbstractChar) -> Bool
```

Tests whether a character is a decimal digit (0-9).

See also: `isletter`.

Examples

```
julia> isdigit('❤')
false

julia> isdigit('9')
true

julia> isdigit('α')
false
```

source

`Base.Unicode.isletter` – Function.

```
isletter(c::AbstractChar) -> Bool
```

Test whether a character is a letter. A character is classified as a letter if it belongs to the Unicode general category Letter, i.e. a character whose category code begins with 'L'.

See also: `isdigit`.

Examples

```
julia> isletter('❤')
false

julia> isletter('α')
true

julia> isletter('9')
false
```

source

`Base.Unicode.islowercase` – Function.

```
islowercase(c::AbstractChar) -> Bool
```

Tests whether a character is a lowercase letter (according to the Unicode standard's `Lowercase` derived property).

See also `isuppercase`.

Examples

```
julia> islowercase('α')
true

julia> islowercase('Γ')
false

julia> islowercase('❤')
false
```

source

`Base.Unicode.isnumeric` – Function.

```
isnumeric(c::AbstractChar) -> Bool
```

Tests whether a character is numeric. A character is classified as numeric if it belongs to the Unicode general category Number, i.e. a character whose category code begins with 'N'.

Note that this broad category includes characters such as ¾ and ௰. Use `isdigit` to check whether a character is a decimal digit between 0 and 9.

Examples

```
julia> isnumeric('௰')
true

julia> isnumeric('9')
true
```

```
julia> isnumeric('α')
false

julia> isnumeric('❤')
false
```

source

Base.Unicode.isprint – Function.

```
isprint(c::AbstractChar) -> Bool
```

Tests whether a character is printable, including spaces, but not a control character.

Examples

```
julia> isprint('\x01')
false

julia> isprint('A')
true
```

source

Base.Unicode.ispunct – Function.

```
ispunct(c::AbstractChar) -> Bool
```

Tests whether a character belongs to the Unicode general category Punctuation, i.e. a character whose category code begins with 'P'.

Examples

```
julia> ispunct('α')
false

julia> ispunct('/')
true

julia> ispunct(';')
true
```

source

Base.Unicode.isspace – Function.

```
isspace(c::AbstractChar) -> Bool
```

Tests whether a character is any whitespace character. Includes ASCII characters '\t', '\n', '\v', '\f', '\r', and ' ', Latin-1 character U+0085, and characters in Unicode category Zs.

Examples

```
julia> isspace('\n')
true

julia> isspace('\r')
true

julia> isspace(' ')
true

julia> isspace('\x20')
true
```

source

Base.Unicode.isuppercase – Function.

```
isuppercase(c::AbstractChar) -> Bool
```

Tests whether a character is an uppercase letter (according to the Unicode standard's `Uppercase` derived property).

See also `islowercase`.

Examples

```
julia> isuppercase('γ')
false

julia> isuppercase('Γ')
true

julia> isuppercase('❤')
false
```

source

Base.Unicode.isxdigit – Function.

```
isxdigit(c::AbstractChar) -> Bool
```

Test whether a character is a valid hexadecimal digit. Note that this does not include `x` (as in the standard `0x` prefix).

Examples

```
julia> isxdigit('a')
true

julia> isxdigit('x')
false
```

source

Base.escape_string – Function.

```
escape_string(str::AbstractString[, esc]; keep = ())::AbstractString
escape_string(io, str::AbstractString[, esc]; keep = ())::Nothing
```

General escaping of traditional C and Unicode escape sequences. The first form returns the escaped string, the second prints the result to `io`.

Backslashes (`\`) are escaped with a double-backslash (`"\\"`). Non-printable characters are escaped either with their standard C escape codes, `"\0"` for NUL (if unambiguous), unicode code point (`"\u"` prefix) or hex (`"\x"` prefix).

The optional `esc` argument specifies any additional characters that should also be escaped by a prepending backslash (`"` is also escaped by default in the first form).

The argument keep specifies a collection of characters which are to be kept as they are. Notice that `esc` has precedence here.

See also `unescape_string` for the reverse operation.

Julia 1.7

The keep argument is available as of Julia 1.7.

Examples

```
julia> escape_string("aaa\nbbb")
"aaa\\nbbb"

julia> escape_string("aaa\nbbb"; keep = '\n')
"aaa\nbbb"

julia> escape_string("\xfe\xff") # invalid utf-8
"\\xfe\\xff"

julia> escape_string(string('\u2135','\0')) # unambiguous
"ℵ\\0"

julia> escape_string(string('\u2135','\0','0')) # \0 would be ambiguous
"ℵ\\x000"
```

source

Base.unescape_string – Function.

```
unescape_string(str::AbstractString, keep = ())::AbstractString
unescape_string(io, s::AbstractString, keep = ())::Nothing
```

General unescaping of traditional C and Unicode escape sequences. The first form returns the escaped string, the second prints the result to io. The argument keep specifies a collection of characters which (along with backlashes) are to be kept as they are.

The following escape sequences are recognised:

- Escaped backslash (\\)
- Escaped double-quote (\")
- Standard C escape sequences (\a, \b, \t, \n, \v, \f, \r, \e)
- Unicode BMP code points (\u with 1-4 trailing hex digits)
- All Unicode code points (\U with 1-8 trailing hex digits; max value = 0010ffff)
- Hex bytes (\x with 1-2 trailing hex digits)
- Octal bytes (\ with 1-3 trailing octal digits)

See also escape_string.

Examples

```
julia> unescape_string("aaa\\nbbb") # C escape sequence
"aaa\nbbb"

julia> unescape_string("\\u03c0") # unicode
"π"

julia> unescape_string("\\101") # octal
"A"

julia> unescape_string("aaa \\g \\n", ['g']) # using `keep` argument
"aaa \\g \n"
```

source

Chapter 47

数组

47.1 构造函数与类型

`Core.AbstractArray` – Type.

```
AbstractArray{T,N}
```

Supertype for N-dimensional arrays (or array-like types) with elements of type T. `Array` and other types are subtypes of this. See the manual section on the `AbstractArray` interface.

See also: `AbstractVector`, `AbstractMatrix`, `eltype`, `ndims`.

source

`Base.AbstractVector` – Type.

```
AbstractVector{T}
```

Supertype for one-dimensional arrays (or array-like types) with elements of type T. Alias for `AbstractArray{T,1}`.

source

`Base.AbstractMatrix` – Type.

```
AbstractMatrix{T}
```

Supertype for two-dimensional arrays (or array-like types) with elements of type T. Alias for `AbstractArray{T,2}`.

source

`Base.AbstractVecOrMat` – Type.

```
AbstractVecOrMat{T}
```

Union type of `AbstractVector{T}` and `AbstractMatrix{T}`.

source

`Core.Array` – Type.

```
Array{T,N} <: AbstractArray{T,N}
```

N-dimensional dense array with elements of type T.

source

`Core.Array` – Method.

```
Array{T}(undef, dims)
Array{T,N}(undef, dims)
```

Construct an uninitialized N-dimensional `Array` containing elements of type T. N can either be supplied explicitly, as in `Array{T,N}(undef, dims)`, or be determined by the length or number of `dims`. `dims` may be a tuple or a series of integer arguments corresponding to the lengths in each dimension. If the rank N is supplied explicitly, then it must match the length or number of `dims`. Here `undef` is the `UndefInitializer`.

Examples

```
julia> A = Array{Float64, 2}(undef, 2, 3) # N given explicitly
2×3 Matrix{Float64}:
 6.90198e-310  6.90198e-310  6.90198e-310
 6.90198e-310  6.90198e-310  0.0

julia> B = Array{Float64}(undef, 4) # N determined by the input
4-element Vector{Float64}:
   2.360075077e-314
 NaN
   2.2671131793e-314
   2.299821756e-314

julia> similar(B, 2, 4, 1) # use typeof(B), and the given size
2×4×1 Array{Float64, 3}:
[:, :, 1] =
 2.26703e-314  2.26708e-314  0.0           2.80997e-314
 0.0           2.26703e-314  2.26708e-314  0.0
```

source

`Core.Array` – Method.

```
Array{T}(nothing, dims)
Array{T,N}(nothing, dims)
```

Construct an N-dimensional `Array` containing elements of type T, initialized with `nothing` entries. Element type T must be able to hold these values, i.e. `Nothing <: T`.

Examples

```
julia> Array{Union{Nothing, String}}(nothing, 2)
2-element Vector{Union{Nothing, String}}:
 nothing
 nothing

julia> Array{Union{Nothing, Int}}(nothing, 2, 3)
2×3 Matrix{Union{Nothing, Int64}}:
 nothing  nothing  nothing
 nothing  nothing  nothing
```

source

Core.Array – Method.

```
Array{T}(missing, dims)
Array{T,N}(missing, dims)
```

Construct an N-dimensional `Array` containing elements of type T, initialized with `missing` entries. Element type T must be able to hold these values, i.e. `Missing <: T`.

Examples

```
julia> Array{Union{Missing, String}}(missing, 2)
2-element Vector{Union{Missing, String}}:
 missing
 missing

julia> Array{Union{Missing, Int}}(missing, 2, 3)
2×3 Matrix{Union{Missing, Int64}}:
 missing  missing  missing
 missing  missing  missing
```

source

Core.UndefInitializer – Type.

```
UndefInitializer
```

Singleton type used in array initialization, indicating the array-constructor-caller would like an uninitialized array. See also `undef`, an alias for `UndefInitializer()`.

Examples

```
julia> Array{Float64, 1}(UndefInitializer(), 3)
3-element Array{Float64, 1}:
 2.2752528595e-314
 2.202942107e-314
 2.275252907e-314
```

source

`Core.undef` – Constant.

```
undef
```

Alias for `UndefInitializer()`, which constructs an instance of the singleton type `UndefInitializer`, used in array initialization to indicate the array-constructor-caller would like an uninitialized array.

See also: `missing`, `similar`.

Examples

```
julia> Array{Float64, 1}(undef, 3)
3-element Vector{Float64}:
 2.2752528595e-314
 2.202942107e-314
 2.275252907e-314
```

source

`Base.Vector` – Type.

```
Vector{T} <: AbstractVector{T}
```

One-dimensional dense array with elements of type T, often used to represent a mathematical vector. Alias for `Array{T,1}`.

See also `empty`, `similar` and `zero` for creating vectors.

source

`Base.Vector` – Method.

```
Vector{T}(undef, n)
```

Construct an uninitialized `Vector{T}` of length n.

Examples

```
julia> Vector{Float64}(undef, 3)
3-element Array{Float64, 1}:
 6.90966e-310
 6.90966e-310
 6.90966e-310
```

source

`Base.Vector` – Method.

```
Vector{T}(nothing, m)
```

Construct a `Vector{T}` of length `m`, initialized with `nothing` entries. Element type `T` must be able to hold these values, i.e. `Nothing <: T`.

Examples

```
julia> Vector{Union{Nothing, String}}(nothing, 2)
2-element Vector{Union{Nothing, String}}:
 nothing
 nothing
```

source

`Base.Vector` – Method.

```
Vector{T}(missing, m)
```

Construct a `Vector{T}` of length `m`, initialized with `missing` entries. Element type `T` must be able to hold these values, i.e. `Missing <: T`.

Examples

```
julia> Vector{Union{Missing, String}}(missing, 2)
2-element Vector{Union{Missing, String}}:
 missing
 missing
```

source

`Base.Matrix` – Type.

```
Matrix{T} <: AbstractMatrix{T}
```

Two-dimensional dense array with elements of type `T`, often used to represent a mathematical matrix. Alias for `Array{T,2}`.

See also `fill`, `zeros`, `undef` and `similar` for creating matrices.

source

`Base.Matrix` – Method.

```
Matrix{T}(undef, m, n)
```

Construct an uninitialized `Matrix{T}` of size `m×n`.

Examples

```
julia> Matrix{Float64}(undef, 2, 3)
2×3 Array{Float64, 2}:
 2.36365e-314  2.28473e-314    5.0e-324
 2.26704e-314  2.26711e-314  NaN

julia> similar(ans, Int32, 2, 2)
2×2 Matrix{Int32}:
 490537216  1277177453
         1  1936748399
```

source

Base.Matrix – Method.

```
Matrix{T}(nothing, m, n)
```

Construct a Matrix{T} of size m×n, initialized with nothing entries. Element type T must be able to hold these values, i.e. Nothing <: T.

Examples

```
julia> Matrix{Union{Nothing, String}}(nothing, 2, 3)
2×3 Matrix{Union{Nothing, String}}:
 nothing  nothing  nothing
 nothing  nothing  nothing
```

source

Base.Matrix – Method.

```
Matrix{T}(missing, m, n)
```

Construct a Matrix{T} of size m×n, initialized with missing entries. Element type T must be able to hold these values, i.e. Missing <: T.

Examples

```
julia> Matrix{Union{Missing, String}}(missing, 2, 3)
2×3 Matrix{Union{Missing, String}}:
 missing  missing  missing
 missing  missing  missing
```

source

Base.VecOrMat – Type.

```
VecOrMat{T}
```

Union type of `Vector{T}` and `Matrix{T}` which allows functions to accept either a Matrix or a Vector.

Examples

```
julia> Vector{Float64} <: VecOrMat{Float64}
true

julia> Matrix{Float64} <: VecOrMat{Float64}
true

julia> Array{Float64, 3} <: VecOrMat{Float64}
false
```

source

`Core.DenseArray` – Type.

```
DenseArray{T, N} <: AbstractArray{T,N}
```

N-dimensional dense array with elements of type T. The elements of a dense array are stored contiguously in memory.

source

`Base.DenseVector` – Type.

```
DenseVector{T}
```

One-dimensional `DenseArray` with elements of type T. Alias for `DenseArray{T,1}`.

source

`Base.DenseMatrix` – Type.

```
DenseMatrix{T}
```

Two-dimensional `DenseArray` with elements of type T. Alias for `DenseArray{T,2}`.

source

`Base.DenseVecOrMat` – Type.

```
DenseVecOrMat{T}
```

Union type of `DenseVector{T}` and `DenseMatrix{T}`.

source

`Base.StridedArray` – Type.

```
StridedArray{T, N}
```

A hard-coded `Union` of common array types that follow the strided array interface, with elements of type `T` and `N` dimensions.

If `A` is a `StridedArray`, then its elements are stored in memory with offsets, which may vary between dimensions but are constant within a dimension. For example, `A` could have stride 2 in dimension 1, and stride 3 in dimension 2. Incrementing `A` along dimension `d` jumps in memory by [`stride(A, d)`] slots. Strided arrays are particularly important and useful because they can sometimes be passed directly as pointers to foreign language libraries like BLAS.

source

`Base.StridedVector` – Type.

```
StridedVector{T}
```

One dimensional `StridedArray` with elements of type `T`.

source

`Base.StridedMatrix` – Type.

```
StridedMatrix{T}
```

Two dimensional `StridedArray` with elements of type `T`.

source

`Base.StridedVecOrMat` – Type.

```
StridedVecOrMat{T}
```

Union type of `StridedVector` and `StridedMatrix` with elements of type `T`.

source

`Base.Slices` – Type.

```
Slices{P,SM,AX,S,N} <: AbstractSlices{S,N}
```

An `AbstractArray` of slices into a parent array over specified dimension(s), returning views that select all the data from the other dimension(s).

These should typically be constructed by `eachslice`, `eachcol` or `eachrow`.

`parent(s::Slices)` will return the parent array.

source

Base.RowSlices – Type.

```
RowSlices{M,AX,S}
```

A special case of Slices that is a vector of row slices of a matrix, as constructed by eachrow.

parent can be used to get the underlying matrix.

source

Base.ColumnSlices – Type.

```
ColumnSlices{M,AX,S}
```

A special case of Slices that is a vector of column slices of a matrix, as constructed by eachcol.

parent can be used to get the underlying matrix.

source

Base.getindex – Method.

```
getindex(type[, elements...])
```

Construct a 1-d array of the specified type. This is usually called with the syntax Type[]. Element values can be specified using Type[a,b,c,...].

Examples

```
julia> Int8[1, 2, 3]
3-element Vector{Int8}:
 1
 2
 3

julia> getindex(Int8, 1, 2, 3)
3-element Vector{Int8}:
 1
 2
 3
```

source

Base.zeros – Function.

```
zeros([T=Float64,] dims::Tuple)
zeros([T=Float64,] dims...)
```

Create an Array, with element type T, of all zeros with size specified by dims. See also fill, ones, zero.

Examples

```
julia> zeros(1)
1-element Vector{Float64}:
 0.0

julia> zeros(Int8, 2, 3)
2×3 Matrix{Int8}:
 0  0  0
 0  0  0
```

source

Base.ones – Function.

```
ones([T=Float64,] dims::Tuple)
ones([T=Float64,] dims...)
```

Create an Array, with element type T, of all ones with size specified by dims. See also fill, zeros.

Examples

```
julia> ones(1,2)
1×2 Matrix{Float64}:
 1.0  1.0

julia> ones(ComplexF64, 2, 3)
2×3 Matrix{ComplexF64}:
 1.0+0.0im  1.0+0.0im  1.0+0.0im
 1.0+0.0im  1.0+0.0im  1.0+0.0im
```

source

Base.BitArray – Type.

```
BitArray{N} <: AbstractArray{Bool, N}
```

Space-efficient N-dimensional boolean array, using just one bit for each boolean value.

BitArrays pack up to 64 values into every 8 bytes, resulting in an 8x space efficiency over Array{Bool, N} and allowing some operations to work on 64 values at once.

By default, Julia returns BitArrays from broadcasting operations that generate boolean elements (including dotted-comparisons like .==) as well as from the functions trues and falses.

Note

Due to its packed storage format, concurrent access to the elements of a BitArray where at least one of them is a write is not thread-safe.

source

Base.BitArray – Method.

```
BitArray(undef, dims::Integer...)
BitArray{N}(undef, dims::NTuple{N,Int})
```

Construct an undef `BitArray` with the given dimensions. Behaves identically to the `Array` constructor. See `undef`.

Examples

```
julia> BitArray(undef, 2, 2)
2×2 BitMatrix:
 0  0
 0  0

julia> BitArray(undef, (3, 1))
3×1 BitMatrix:
 0
 0
 0
```

source

Base.BitArray – Method.

```
BitArray(itr)
```

Construct a `BitArray` generated by the given iterable object. The shape is inferred from the `itr` object.

Examples

```
julia> BitArray([1 0; 0 1])
2×2 BitMatrix:
 1  0
 0  1

julia> BitArray(x+y == 3 for x = 1:2, y = 1:3)
2×3 BitMatrix:
 0  1  0
 1  0  0

julia> BitArray(x+y == 3 for x = 1:2 for y = 1:3)
6-element BitVector:
 0
 1
 0
 1
 0
 0
```

source

Base.trues – Function.

```
trues(dims)
```

Create a `BitArray` with all values set to `true`.

Examples

```
julia> trues(2,3)
2×3 BitMatrix:
 1  1  1
 1  1  1
```

source

`Base.falses` – Function.

```
falses(dims)
```

Create a `BitArray` with all values set to `false`.

Examples

```
julia> falses(2,3)
2×3 BitMatrix:
 0  0  0
 0  0  0
```

source

`Base.fill` – Function.

```
fill(value, dims::Tuple)
fill(value, dims...)
```

Create an array of size `dims` with every location set to `value`.

For example, `fill(1.0, (5,5))` returns a 5×5 array of floats, with `1.0` in every location of the array.

The dimension lengths `dims` may be specified as either a tuple or a sequence of arguments. An `N`-length tuple or `N` arguments following the `value` specify an `N`-dimensional array. Thus, a common idiom for creating a zero-dimensional array with its only location set to `x` is `fill(x)`.

Every location of the returned array is set to (and is thus `===` to) the `value` that was passed; this means that if the `value` is itself modified, all elements of the `fill`ed array will reflect that modification because they're *still* that very `value`. This is of no concern with `fill(1.0, (5,5))` as the `value` `1.0` is immutable and cannot itself be modified, but can be unexpected with mutable values like —most commonly —arrays. For example, `fill([], 3)` places *the very same* empty array in all three locations of the returned vector:

```
julia> v = fill([], 3)
3-element Vector{Vector{Any}}:
 []
 []
 []

julia> v[1] === v[2] === v[3]
true

julia> value = v[1]
Any[]

julia> push!(value, 867_5309)
1-element Vector{Any}:
 8675309

julia> v
3-element Vector{Vector{Any}}:
 [8675309]
 [8675309]
 [8675309]
```

To create an array of many independent inner arrays, use a comprehension instead. This creates a new and distinct array on each iteration of the loop:

```
julia> v2 = [[] for _ in 1:3]
3-element Vector{Vector{Any}}:
 []
 []
 []

julia> v2[1] === v2[2] === v2[3]
false

julia> push!(v2[1], 8675309)
1-element Vector{Any}:
 8675309

julia> v2
3-element Vector{Vector{Any}}:
 [8675309]
 []
 []
```

See also: `fill!`, `zeros`, `ones`, `similar`.

Examples

```
julia> fill(1.0, (2,3))
2×3 Matrix{Float64}:
 1.0  1.0  1.0
 1.0  1.0  1.0
```

```
julia> fill(42)
0-dimensional Array{Int64, 0}:
42

julia> A = fill(zeros(2), 2) # sets both elements to the same [0.0, 0.0] vector
2-element Vector{Vector{Float64}}:
 [0.0, 0.0]
 [0.0, 0.0]

julia> A[1][1] = 42; # modifies the filled value to be [42.0, 0.0]

julia> A # both A[1] and A[2] are the very same vector
2-element Vector{Vector{Float64}}:
 [42.0, 0.0]
 [42.0, 0.0]
```

source

Base.fill! – Function.

```
fill!(A, x)
```

Fill array A with the value x. If x is an object reference, all elements will refer to the same object. `fill!(A, Foo())` will return A filled with the result of evaluating `Foo()` once.

Examples

```
julia> A = zeros(2,3)
2×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> fill!(A, 2.)
2×3 Matrix{Float64}:
 2.0  2.0  2.0
 2.0  2.0  2.0

julia> a = [1, 1, 1]; A = fill!(Vector{Vector{Int}}(undef, 3), a); a[1] = 2; A
3-element Vector{Vector{Int64}}:
 [2, 1, 1]
 [2, 1, 1]
 [2, 1, 1]

julia> x = 0; f() = (global x += 1; x); fill!(Vector{Int}(undef, 3), f())
3-element Vector{Int64}:
 1
 1
 1
```

source

Base.empty – Function.

```
empty(x::Tuple)
```

Return an empty tuple, `()`.

source

```
empty(v::AbstractVector, [eltype])
```

Create an empty vector similar to `v`, optionally changing the `eltype`.

See also: `empty!`, `isempty`, `isassigned`.

Examples

```
julia> empty([1.0, 2.0, 3.0])
Float64[]

julia> empty([1.0, 2.0, 3.0], String)
String[]
```

source

```
empty(a::AbstractDict, [index_type=keytype(a)], [value_type=valtype(a)])
```

Create an empty `AbstractDict` container which can accept indices of type `index_type` and values of type `value_type`. The second and third arguments are optional and default to the input's `keytype` and `valtype`, respectively. (If only one of the two types is specified, it is assumed to be the `value_type`, and the `index_type` we default to `keytype(a)`).

Custom `AbstractDict` subtypes may choose which specific dictionary type is best suited to return for the given index and value types, by specializing on the three-argument signature. The default is to return an empty `Dict`.

source

`Base.similar` – Function.

```
similar(A::AbstractSparseMatrixCSC{Tv,Ti}, [::Type{TvNew}, ::Type{TiNew}, m::Integer,
↪  n::Integer]) where {Tv,Ti}
```

Create an uninitialized mutable array with the given element type, index type, and size, based upon the given source `SparseMatrixCSC`. The new sparse matrix maintains the structure of the original sparse matrix, except in the case where dimensions of the output matrix are different from the output.

The output matrix has zeros in the same locations as the input, but uninitialized values for the nonzero locations.

source

```
similar(array, [element_type=eltype(array)], [dims=size(array)])
```

Create an uninitialized mutable array with the given element type and size, based upon the given source array. The second and third arguments are both optional, defaulting to the given array's `eltype` and `size`. The dimensions may be specified either as a single tuple argument or as a series of integer arguments.

Custom AbstractArray subtypes may choose which specific array type is best-suited to return for the given element type and dimensionality. If they do not specialize this method, the default is an `Array{element_type}(undef, dims...)`.

For example, `similar(1:10, 1, 4)` returns an uninitialized `Array{Int,2}` since ranges are neither mutable nor support 2 dimensions:

```
julia> similar(1:10, 1, 4)
1×4 Matrix{Int64}:
 4419743872  4374413872  4419743888  0
```

Conversely, `similar(trues(10,10), 2)` returns an uninitialized `BitVector` with two elements since `BitArray`s are both mutable and can support 1-dimensional arrays:

```
julia> similar(trues(10,10), 2)
2-element BitVector:
 0
 0
```

Since `BitArray`s can only store elements of type `Bool`, however, if you request a different element type it will create a regular `Array` instead:

```
julia> similar(falses(10), Float64, 2, 4)
2×4 Matrix{Float64}:
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
```

See also: `undef`, `isassigned`.

source

```
similar(storagetype, axes)
```

Create an uninitialized mutable array analogous to that specified by `storagetype`, but with axes specified by the last argument.

Examples:

```
similar(Array{Int}, axes(A))
```

creates an array that "acts like" an `Array{Int}` (and might indeed be backed by one), but which is indexed identically to A. If A has conventional indexing, this will be identical to `Array{Int}(undef, size(A))`, but if A has unconventional indexing then the indices of the result will match A.

```
similar(BitArray, (axes(A, 2),))
```

would create a 1-dimensional logical array whose indices match those of the columns of A.

source

47.2 基础函数

Base.ndims – Function.

```
ndims(A::AbstractArray) -> Integer
```

Return the number of dimensions of A.

See also: size, axes.

Examples

```
julia> A = fill(1, (3,4,5));

julia> ndims(A)
3
```

source

Base.size – Function.

```
size(A::AbstractArray, [dim])
```

Return a tuple containing the dimensions of A. Optionally you can specify a dimension to just get the length of that dimension.

Note that size may not be defined for arrays with non-standard indices, in which case axes may be useful. See the manual chapter on arrays with custom indices.

See also: length, ndims, eachindex, sizeof.

Examples

```
julia> A = fill(1, (2,3,4));

julia> size(A)
(2, 3, 4)

julia> size(A, 2)
3
```

source

Base.axes – Method.

```
axes(A)
```

Return the tuple of valid indices for array A.

See also: size, keys, eachindex.

Examples

```
julia> A = fill(1, (5,6,7));

julia> axes(A)
(Base.OneTo(5), Base.OneTo(6), Base.OneTo(7))
```

source

Base.axes – Method.

```
axes(A, d)
```

Return the valid range of indices for array A along dimension d.

See also size, and the manual chapter on arrays with custom indices.

Examples

```
julia> A = fill(1, (5,6,7));

julia> axes(A, 2)
Base.OneTo(6)

julia> axes(A, 4) == 1:1  # all dimensions d > ndims(A) have size 1
true
```

Usage note

Each of the indices has to be an `AbstractUnitRange{<:Integer}`, but at the same time can be a type that uses custom indices. So, for example, if you need a subset, use generalized indexing constructs like begin/end or firstindex/lastindex:

```
ix = axes(v, 1)
ix[2:end]          # will work for eg Vector, but may fail in general
ix[(begin+1):end]  # works for generalized indexes
```

source

Base.length – Method.

```
length(A::AbstractArray)
```

Return the number of elements in the array, defaults to `prod(size(A))`.

Examples

```
julia> length([1, 2, 3, 4])
4

julia> length([1 2; 3 4])
4
```

source

Base.keys – Method.

```
keys(a::AbstractArray)
```

Return an efficient array describing all valid indices for a arranged in the shape of a itself.

The keys of 1-dimensional arrays (vectors) are integers, whereas all other N-dimensional arrays use `CartesianIndex` to describe their locations. Often the special array types `LinearIndices` and `CartesianIndices` are used to efficiently represent these arrays of integers and `CartesianIndex`es, respectively.

Note that the keys of an array might not be the most efficient index type; for maximum performance use `eachindex` instead.

Examples

```
julia> keys([4, 5, 6])
3-element LinearIndices{1, Tuple{Base.OneTo{Int64}}}:
 1
 2
 3

julia> keys([4 5; 6 7])
CartesianIndices((2, 2))
```

source

Base.eachindex – Function.

```
eachindex(A...)
eachindex(::IndexStyle, A::AbstractArray...)
```

Create an iterable object for visiting each index of an `AbstractArray` `A` in an efficient manner. For array types that have opted into fast linear indexing (like `Array`), this is simply the range `1:length(A)` if they use 1-based indexing. For array types that have not opted into fast linear indexing, a specialized Cartesian range is typically returned to efficiently index into the array with indices specified for every dimension.

In general `eachindex` accepts arbitrary iterables, including strings and dictionaries, and returns an iterator object supporting arbitrary index types (e.g. unevenly spaced or non-integer indices).

If A is AbstractArray it is possible to explicitly specify the style of the indices that should be returned by eachindex by passing a value having IndexStyle type as its first argument (typically IndexLinear() if linear indices are required or IndexCartesian() if Cartesian range is wanted).

If you supply more than one AbstractArray argument, eachindex will create an iterable object that is fast for all arguments (typically a UnitRange if all inputs have fast linear indexing, a CartesianIndices otherwise). If the arrays have different sizes and/or dimensionalities, a DimensionMismatch exception will be thrown.

See also pairs(A) to iterate over indices and values together, and axes(A, 2) for valid indices along one dimension.

Examples

```
julia> A = [10 20; 30 40];

julia> for i in eachindex(A) # linear indexing
           println("A[", i, "] == ", A[i])
       end
A[1] == 10
A[2] == 30
A[3] == 20
A[4] == 40

julia> for i in eachindex(view(A, 1:2, 1:1)) # Cartesian indexing
           println(i)
       end
CartesianIndex(1, 1)
CartesianIndex(2, 1)
```

source

Base.IndexStyle – Type.

```
IndexStyle(A)
IndexStyle(typeof(A))
```

IndexStyle specifies the "native indexing style" for array A. When you define a new AbstractArray type, you can choose to implement either linear indexing (with IndexLinear) or cartesian indexing. If you decide to only implement linear indexing, then you must set this trait for your array type:

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

The default is IndexCartesian().

Julia's internal indexing machinery will automatically (and invisibly) recompute all indexing operations into the preferred style. This allows users to access elements of your array using any indexing style, even when explicit methods have not been provided.

If you define both styles of indexing for your AbstractArray, this trait can be used to select the most performant indexing style. Some methods check this trait on their inputs, and dispatch to different algorithms depending on the most efficient access pattern. In particular, eachindex creates an iterator whose type depends on the setting of this trait.

source

`Base.IndexLinear` – Type.

```
IndexLinear()
```

Subtype of `IndexStyle` used to describe arrays which are optimally indexed by one linear index.

A linear indexing style uses one integer index to describe the position in the array (even if it's a multidimensional array) and column-major ordering is used to efficiently access the elements. This means that requesting `eachindex` from an array that is `IndexLinear` will return a simple one-dimensional range, even if it is multidimensional.

A custom array that reports its `IndexStyle` as `IndexLinear` only needs to implement indexing (and indexed assignment) with a single `Int` index; all other indexing expressions —including multidimensional accesses —will be recomputed to the linear index. For example, if `A` were a 2×3 custom matrix with linear indexing, and we referenced `A[1, 3]`, this would be recomputed to the equivalent linear index and call `A[5]` since `1 + 2*(3 - 1) = 5`.

See also `IndexCartesian`.

source

`Base.IndexCartesian` – Type.

```
IndexCartesian()
```

Subtype of `IndexStyle` used to describe arrays which are optimally indexed by a Cartesian index. This is the default for new custom `AbstractArray` subtypes.

A Cartesian indexing style uses multiple integer indices to describe the position in a multidimensional array, with exactly one index per dimension. This means that requesting `eachindex` from an array that is `IndexCartesian` will return a range of `CartesianIndices`.

A `N`-dimensional custom array that reports its `IndexStyle` as `IndexCartesian` needs to implement indexing (and indexed assignment) with exactly `N` `Int` indices; all other indexing expressions —including linear indexing —will be recomputed to the equivalent Cartesian location. For example, if `A` were a 2×3 custom matrix with cartesian indexing, and we referenced `A[5]`, this would be recomputed to the equivalent Cartesian index and call `A[1, 3]` since `5 = 1 + 2*(3 - 1)`.

It is significantly more expensive to compute Cartesian indices from a linear index than it is to go the other way. The former operation requires division —a very costly operation —whereas the latter only uses multiplication and addition and is essentially free. This asymmetry means it is far more costly to use linear indexing with an `IndexCartesian` array than it is to use Cartesian indexing with an `IndexLinear` array.

See also `IndexLinear`.

source

`Base.conj!` – Function.

```
conj!(A)
```

Transform an array to its complex conjugate in-place.

See also conj.

Examples

```
julia> A = [1+im 2-im; 2+2im 3+im]
2×2 Matrix{Complex{Int64}}:
 1+1im  2-1im
 2+2im  3+1im

julia> conj!(A);

julia> A
2×2 Matrix{Complex{Int64}}:
 1-1im  2+1im
 2-2im  3-1im
```

source

Base.stride – Function.

```
stride(A, k::Integer)
```

Return the distance in memory (in number of elements) between adjacent elements in dimension k.

See also: strides.

Examples

```
julia> A = fill(1, (3,4,5));

julia> stride(A,2)
3

julia> stride(A,3)
12
```

source

Base.strides – Function.

```
strides(A)
```

Return a tuple of the memory strides in each dimension.

See also: stride.

Examples

```
julia> A = fill(1, (3,4,5));

julia> strides(A)
(1, 3, 12)
```

source

47.3 广播与矢量化

也可参照 dot syntax for vectorizing functions；例如，`f.(args...)` 隐式调用 `broadcast(f, args...)`。与其依赖如 `sin` 函数的“已矢量化”方法，你应该使用 `sin.(a)` 来使用 `broadcast` 来矢量化。

`Base.Broadcast.broadcast` – Function.

```
broadcast(f, As...)
```

Broadcast the function `f` over the arrays, tuples, collections, `Ref`s and/or scalars `As`.

Broadcasting applies the function `f` over the elements of the container arguments and the scalars themselves in `As`. Singleton and missing dimensions are expanded to match the extents of the other arguments by virtually repeating the value. By default, only a limited number of types are considered scalars, including `Number`s, `String`s, `Symbol`s, `Type`s, `Function`s and some common singletons like `missing` and `nothing`. All other arguments are iterated over or indexed into elementwise.

The resulting container type is established by the following rules:

- If all the arguments are scalars or zero-dimensional arrays, it returns an unwrapped scalar.
- If at least one argument is a tuple and all others are scalars or zero-dimensional arrays, it returns a tuple.
- All other combinations of arguments default to returning an `Array`, but custom container types can define their own implementation and promotion-like rules to customize the result when they appear as arguments.

A special syntax exists for broadcasting: `f.(args...)` is equivalent to `broadcast(f, args...)`, and nested `f.(g.(args...))` calls are fused into a single broadcast loop.

Examples

```
julia> A = [1, 2, 3, 4, 5]
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> B = [1 2; 3 4; 5 6; 7 8; 9 10]
5×2 Matrix{Int64}:
 1  2
 3  4
 5  6
```

```
 7   8
 9  10

julia> broadcast(+, A, B)
5×2 Matrix{Int64}:
  2   3
  5   6
  8   9
 11  12
 14  15

julia> parse.(Int, ["1", "2"])
2-element Vector{Int64}:
 1
 2

julia> abs.((1, -2))
(1, 2)

julia> broadcast(+, 1.0, (0, -2.0))
(1.0, -1.0)

julia> (+).([[0,2], [1,3]], Ref{Vector{Int}}([1,-1]))
2-element Vector{Vector{Int64}}:
 [1, 1]
 [2, 2]

julia> string.(("one","two","three","four"), ": ", 1:4)
4-element Vector{String}:
 "one: 1"
 "two: 2"
 "three: 3"
 "four: 4"
```

source

`Base.Broadcast.broadcast!` – Function.

```
broadcast!(f, dest, As...)
```

Like broadcast, but store the result of `broadcast(f, As...)` in the `dest` array. Note that `dest` is only used to store the result, and does not supply arguments to `f` unless it is also listed in the `As`, as in `broadcast!(f, A, A, B)` to perform `A[:] = broadcast(f, A, B)`.

Examples

```
julia> A = [1.0; 0.0]; B = [0.0; 0.0];

julia> broadcast!(+, B, A, (0, -2.0));

julia> B
2-element Vector{Float64}:
  1.0
```

```
 -2.0

julia> A
2-element Vector{Float64}:
 1.0
 0.0

julia> broadcast!(+, A, A, (0, -2.0));

julia> A
2-element Vector{Float64}:
  1.0
 -2.0
```

source

Base.Broadcast.@__dot__ – Macro.

```
@. expr
```

Convert every function call or operator in expr into a "dot call" (e.g. convert f(x) to f.(x)), and convert every assignment in expr to a "dot assignment" (e.g. convert += to .+=).

If you want to *avoid* adding dots for selected function calls in expr, splice those function calls in with $. For example, @. sqrt(abs($sort(x))) is equivalent to sqrt.(abs.(sort(x))) (no dot for sort).

(@. is equivalent to a call to @__dot__.)

Examples

```
julia> x = 1.0:3.0; y = similar(x);

julia> @. y = x + 3 * sin(x)
3-element Vector{Float64}:
 3.5244129544236893
 4.727892280477045
 3.4233600241796016
```

source

自定义类型的广播，请参照

Base.Broadcast.BroadcastStyle – Type.

BroadcastStyle is an abstract type and trait-function used to determine behavior of objects under broadcasting. BroadcastStyle(typeof(x)) returns the style associated with x. To customize the broadcasting behavior of a type, one can declare a style by defining a type/method pair

```
struct MyContainerStyle <: BroadcastStyle end
Base.BroadcastStyle(::Type{<:MyContainer}) = MyContainerStyle()
```

One then writes method(s) (at least similar) operating on Broadcasted{MyContainerStyle}. There are also several pre-defined subtypes of BroadcastStyle that you may be able to leverage; see the Interfaces chapter for more information.

source

Base.Broadcast.AbstractArrayStyle – Type.

Broadcast.AbstractArrayStyle{N} <: BroadcastStyle is the abstract supertype for any style associated with an AbstractArray type. The N parameter is the dimensionality, which can be handy for AbstractArray types that only support specific dimensionalities:

```
struct SparseMatrixStyle <: Broadcast.AbstractArrayStyle{2} end
Base.BroadcastStyle(::Type{<:SparseMatrixCSC}) = SparseMatrixStyle()
```

For AbstractArray types that support arbitrary dimensionality, N can be set to Any:

```
struct MyArrayStyle <: Broadcast.AbstractArrayStyle{Any} end
Base.BroadcastStyle(::Type{<:MyArray}) = MyArrayStyle()
```

In cases where you want to be able to mix multiple AbstractArrayStyles and keep track of dimensionality, your style needs to support a Val constructor:

```
struct MyArrayStyleDim{N} <: Broadcast.AbstractArrayStyle{N} end
(::Type{<:MyArrayStyleDim})(::Val{N}) where N = MyArrayStyleDim{N}()
```

Note that if two or more AbstractArrayStyle subtypes conflict, broadcasting machinery will fall back to producing Arrays. If this is undesirable, you may need to define binary BroadcastStyle rules to control the output type.

See also Broadcast.DefaultArrayStyle.

source

Base.Broadcast.ArrayStyle – Type.

Broadcast.ArrayStyle{MyArrayType}() is a BroadcastStyle indicating that an object behaves as an array for broadcasting. It presents a simple way to construct Broadcast.AbstractArrayStyles for specific AbstractArray container types. Broadcast styles created this way lose track of dimensionality; if keeping track is important for your type, you should create your own custom Broadcast.AbstractArrayStyle.

source

Base.Broadcast.DefaultArrayStyle – Type.

Broadcast.DefaultArrayStyle{N}() is a BroadcastStyle indicating that an object behaves as an N-dimensional array for broadcasting. Specifically, DefaultArrayStyle is used for any AbstractArray type that hasn't defined a specialized style, and in the absence of overrides from other broadcast arguments the resulting output type is Array. When there are multiple inputs to broadcast, DefaultArrayStyle "loses" to any other Broadcast.ArrayStyle.

source

Base.Broadcast.broadcastable – Function.

```
Broadcast.broadcastable(x)
```

Return either x or an object like x such that it supports `axes`, indexing, and its type supports `ndims`.

If x supports iteration, the returned value should have the same axes and indexing behaviors as `collect(x)`.

If x is not an `AbstractArray` but it supports axes, indexing, and its type supports `ndims`, then `broadcastable(::typeof(x))` may be implemented to just return itself. Further, if x defines its own `BroadcastStyle`, then it must define its `broadcastable` method to return itself for the custom style to have any effect.

Examples

```
julia> Broadcast.broadcastable([1,2,3]) # like `identity` since arrays already support axes and
↪  indexing
3-element Vector{Int64}:
 1
 2
 3

julia> Broadcast.broadcastable(Int) # Types don't support axes, indexing, or iteration but are
↪  commonly used as scalars
Base.RefValue{Type{Int64}}(Int64)

julia> Broadcast.broadcastable("hello") # Strings break convention of matching iteration and act
↪  like a scalar instead
Base.RefValue{String}("hello")
```

source

`Base.Broadcast.combine_axes` – Function.

```
combine_axes(As...) -> Tuple
```

Determine the result axes for broadcasting across all values in As.

```
julia> Broadcast.combine_axes([1], [1 2; 3 4; 5 6])
(Base.OneTo(3), Base.OneTo(2))

julia> Broadcast.combine_axes(1, 1, 1)
()
```

source

`Base.Broadcast.combine_styles` – Function.

```
combine_styles(cs...) -> BroadcastStyle
```

Decides which `BroadcastStyle` to use for any number of value arguments. Uses `BroadcastStyle` to get the style for each argument, and uses `result_style` to combine styles.

Examples

```
julia> Broadcast.combine_styles([1], [1 2; 3 4])
Base.Broadcast.DefaultArrayStyle{2}()
```

source

`Base.Broadcast.result_style` – Function.

```
result_style(s1::BroadcastStyle[, s2::BroadcastStyle]) -> BroadcastStyle
```

Takes one or two `BroadcastStyle`s and combines them using `BroadcastStyle` to determine a common `BroadcastStyle`.

Examples

```
julia> Broadcast.result_style(Broadcast.DefaultArrayStyle{0}(),
↪  Broadcast.DefaultArrayStyle{3}())
Base.Broadcast.DefaultArrayStyle{3}()

julia> Broadcast.result_style(Broadcast.Unknown(), Broadcast.DefaultArrayStyle{1}())
Base.Broadcast.DefaultArrayStyle{1}()
```

source

47.4 索引与赋值

`Base.getindex` – Method.

```
getindex(A, inds...)
```

Return a subset of array A as specified by `inds`, where each `ind` may be, for example, an `Int`, an `AbstractRange`, or a `Vector`. See the manual section on array indexing for details.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> getindex(A, 1)
1

julia> getindex(A, [2, 1])
2-element Vector{Int64}:
 3
 1
```

```
julia> getindex(A, 2:4)
3-element Vector{Int64}:
 3
 2
 4
```

source

Base.setindex! – Method.

```
setindex!(A, X, inds...)
A[inds...] = X
```

Store values from array X within some subset of A as specified by inds. The syntax A[inds...] = X is equivalent to (setindex!(A, X, inds...); X).

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = zeros(2,2);

julia> setindex!(A, [10, 20], [1, 2]);

julia> A[[3, 4]] = [30, 40];

julia> A
2×2 Matrix{Float64}:
 10.0  30.0
 20.0  40.0
```

source

Base.copyto! – Method.

```
copyto!(dest, Rdest::CartesianIndices, src, Rsrc::CartesianIndices) -> dest
```

Copy the block of src in the range of Rsrc to the block of dest in the range of Rdest. The sizes of the two regions must match.

Examples

```
julia> A = zeros(5, 5);

julia> B = [1 2; 3 4];
```

```
julia> Ainds = CartesianIndices((2:3, 2:3));

julia> Binds = CartesianIndices(B);

julia> copyto!(A, Ainds, B, Binds)
5×5 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0
 0.0  1.0  2.0  0.0  0.0
 0.0  3.0  4.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```

source

Base.copy! – Function.

```
copy!(dst, src) -> dst
```

In-place copy of src into dst, discarding any pre-existing elements in dst. If dst and src are of the same type, dst == src should hold after the call. If dst and src are multidimensional arrays, they must have equal axes.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also copyto!.

Julia 1.1

This method requires at least Julia 1.1. In Julia 1.0 this method is available from the Future standard library as Future.copy!.

source

Base.isassigned – Function.

```
isassigned(array, i) -> Bool
```

Test whether the given array has a value associated with index i. Return false if the index is out of bounds, or has an undefined reference.

Examples

```
julia> isassigned(rand(3, 3), 5)
true

julia> isassigned(rand(3, 3), 3 * 3 + 1)
```

```
false

julia> mutable struct Foo end

julia> v = similar(rand(3), Foo)
3-element Vector{Foo}:
 #undef
 #undef
 #undef

julia> isassigned(v, 1)
false
```

source

Base.Colon – Type.

```
Colon()
```

Colons (:) are used to signify indexing entire objects or dimensions at once.

Very few operations are defined on Colons directly; instead they are converted by `to_indices` to an internal vector type (`Base.Slice`) to represent the collection of indices they span before being used.

The singleton instance of `Colon` is also a function used to construct ranges; see `:`.

source

Base.IteratorsMD.CartesianIndex – Type.

```
CartesianIndex(i, j, k...)   -> I
CartesianIndex((i, j, k...)) -> I
```

Create a multidimensional index `I`, which can be used for indexing a multidimensional array `A`. In particular, `A[I]` is equivalent to `A[i,j,k...]`. One can freely mix integer and `CartesianIndex` indices; for example, `A[Ipre, i, Ipost]` (where `Ipre` and `Ipost` are `CartesianIndex` indices and `i` is an `Int`) can be a useful expression when writing algorithms that work along a single dimension of an array of arbitrary dimensionality.

A `CartesianIndex` is sometimes produced by `eachindex`, and always when iterating with an explicit `CartesianIndices`.

An `I::CartesianIndex` is treated as a "scalar" (not a container) for `broadcast`. In order to iterate over the components of a `CartesianIndex`, convert it to a tuple with `Tuple(I)`.

Examples

```
julia> A = reshape(Vector(1:16), (2, 2, 2, 2))
2×2×2×2 Array{Int64, 4}:
[:, :, 1, 1] =
 1  3
 2  4
```

```
[:, :, 2, 1] =
 5  7
 6  8

[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> A[CartesianIndex((1, 1, 1, 1))]
1

julia> A[CartesianIndex((1, 1, 1, 2))]
9

julia> A[CartesianIndex((1, 1, 2, 1))]
5
```

Julia 1.10

Using a `CartesianIndex` as a "scalar" for `broadcast` requires Julia 1.10; in previous releases, use `Ref(I)`.

source

`Base.IteratorsMD.CartesianIndices` – Type.

```
CartesianIndices(sz::Dims) -> R
CartesianIndices((istart:[istep:]istop, jstart:[jstep:]jstop, ...)) -> R
```

Define a region `R` spanning a multidimensional rectangular range of integer indices. These are most commonly encountered in the context of iteration, where `for I in R ... end` will return `CartesianIndex` indices `I` equivalent to the nested loops

```
for j = jstart:jstep:jstop
    for i = istart:istep:istop
        ...
    end
end
```

Consequently these can be useful for writing algorithms that work in arbitrary dimensions.

```
CartesianIndices(A::AbstractArray) -> R
```

As a convenience, constructing a `CartesianIndices` from an array makes a range of its indices.

> **Julia 1.6**
>
> The step range method `CartesianIndices((istart:istep:istop, jstart:[jstep:]jstop, ...))` requires at least Julia 1.6.

Examples

```
julia> foreach(println, CartesianIndices((2, 2, 2)))
CartesianIndex(1, 1, 1)
CartesianIndex(2, 1, 1)
CartesianIndex(1, 2, 1)
CartesianIndex(2, 2, 1)
CartesianIndex(1, 1, 2)
CartesianIndex(2, 1, 2)
CartesianIndex(1, 2, 2)
CartesianIndex(2, 2, 2)

julia> CartesianIndices(fill(1, (2,3)))
CartesianIndices((2, 3))
```

Conversion between linear and cartesian indices

Linear index to cartesian index conversion exploits the fact that a `CartesianIndices` is an `AbstractArray` and can be indexed linearly:

```
julia> cartesian = CartesianIndices((1:3, 1:2))
CartesianIndices((1:3, 1:2))

julia> cartesian[4]
CartesianIndex(1, 2)

julia> cartesian = CartesianIndices((1:2:5, 1:2))
CartesianIndices((1:2:5, 1:2))

julia> cartesian[2, 2]
CartesianIndex(3, 2)
```

Broadcasting

`CartesianIndices` support broadcasting arithmetic (+ and -) with a `CartesianIndex`.

> **Julia 1.1**
>
> Broadcasting of CartesianIndices requires at least Julia 1.1.

```
julia> CIs = CartesianIndices((2:3, 5:6))
CartesianIndices((2:3, 5:6))

julia> CI = CartesianIndex(3, 4)
CartesianIndex(3, 4)

julia> CIs .+ CI
CartesianIndices((5:6, 9:10))
```

For cartesian to linear index conversion, see LinearIndices.

source

Base.Dims – Type.

```
Dims{N}
```

An NTuple of N Ints used to represent the dimensions of an AbstractArray.

source

Base.LinearIndices – Type.

```
LinearIndices(A::AbstractArray)
```

Return a LinearIndices array with the same shape and axes as A, holding the linear index of each entry in A. Indexing this array with cartesian indices allows mapping them to linear indices.

For arrays with conventional indexing (indices start at 1), or any multidimensional array, linear indices range from 1 to length(A). However, for AbstractVectors linear indices are axes(A, 1), and therefore do not start at 1 for vectors with unconventional indexing.

Calling this function is the ”safe” way to write algorithms that exploit linear indexing.

Examples

```
julia> A = fill(1, (5,6,7));

julia> b = LinearIndices(A);

julia> extrema(b)
(1, 210)
```

```
LinearIndices(inds::CartesianIndices) -> R
LinearIndices(sz::Dims) -> R
LinearIndices((istart:istop, jstart:jstop, ...)) -> R
```

Return a LinearIndices array with the specified shape or axes.

Example

The main purpose of this constructor is intuitive conversion from cartesian to linear indexing:

```
julia> linear = LinearIndices((1:3, 1:2))
3×2 LinearIndices{2, Tuple{UnitRange{Int64}, UnitRange{Int64}}}:
 1  4
 2  5
 3  6
```

```
julia> linear[1,2]
4
```

source

Base.to_indices – Function.

```
to_indices(A, I::Tuple)
```

Convert the tuple I to a tuple of indices for use in indexing into array A.

The returned tuple must only contain either Ints or AbstractArrays of scalar indices that are supported by array A. It will error upon encountering a novel index type that it does not know how to process.

For simple index types, it defers to the unexported Base.to_index(A, i) to process each index i. While this internal function is not intended to be called directly, Base.to_index may be extended by custom array or index types to provide custom indexing behaviors.

More complicated index types may require more context about the dimension into which they index. To support those cases, to_indices(A, I) calls to_indices(A, axes(A), I), which then recursively walks through both the given tuple of indices and the dimensional indices of A in tandem. As such, not all index types are guaranteed to propagate to Base.to_index.

Examples

```
julia> A = zeros(1,2,3,4);

julia> to_indices(A, (1,1,2,2))
(1, 1, 2, 2)

julia> to_indices(A, (1,1,2,20)) # no bounds checking
(1, 1, 2, 20)

julia> to_indices(A, (CartesianIndex((1,)), 2, CartesianIndex((3,4)))) # exotic index
(1, 2, 3, 4)

julia> to_indices(A, ([1,1], 1:2, 3, 4))
([1, 1], 1:2, 3, 4)

julia> to_indices(A, (1,2)) # no shape checking
(1, 2)
```

source

Base.checkbounds – Function.

```
checkbounds(Bool, A, I...)
```

Return true if the specified indices I are in bounds for the given array A. Subtypes of AbstractArray should specialize this method if they need to provide custom bounds checking behaviors; however, in many cases one can rely on A's indices and checkindex.

See also checkindex.

Examples

```
julia> A = rand(3, 3);

julia> checkbounds(Bool, A, 2)
true

julia> checkbounds(Bool, A, 3, 4)
false

julia> checkbounds(Bool, A, 1:3)
true

julia> checkbounds(Bool, A, 1:3, 2:4)
false
```

source

```
checkbounds(A, I...)
```

Throw an error if the specified indices `I` are not in bounds for the given array `A`.

source

`Base.checkindex` – Function.

```
checkindex(Bool, inds::AbstractUnitRange, index)
```

Return `true` if the given `index` is within the bounds of `inds`. Custom types that would like to behave as indices for all arrays can extend this method in order to provide a specialized bounds checking implementation.

See also checkbounds.

Examples

```
julia> checkindex(Bool, 1:20, 8)
true

julia> checkindex(Bool, 1:20, 21)
false
```

source

`Base.elsize` – Function.

```
elsize(type)
```

Compute the memory stride in bytes between consecutive elements of `eltype` stored inside the given `type`, if the array elements are stored densely with a uniform linear stride.

Examples

```
julia> Base.elsize(rand(Float32, 10))
4
```

source

47.5 Views (SubArrays 以及其它 view 类型)

“视图”是一种表现和数组相似的数据结构（它是 `AbstractArray` 的子类型），但是它的底层数据实际上是另一个数组的一部分。

例如，`x` 是一个数组，`v = @view x[1:10]`，则 `v` 表现得就像一个含有 10 个元素的数组，但是它的数据实际上是访问 `x` 的前 10 个元素。对视图的写入，如 `v[3] = 2`，直接写入了底层的数组 `x`（这里是修改 `x[3]`）。

在 Julia 中，像 `x[1:10]` 这样的切片操作会创建一个副本。`@view x[1:10]` 将它变成创建一个视图。`@views` 宏可以用于整个代码块（如 `@views function foo() .... end` 或 `@views begin ... end`）来将整个代码块中的切片操作变为使用视图。如性能建议所描述的，有时候使用数据的副本更快，而有时候使用视图会更快。

`Base.view` – Function.

```
view(A, inds...)
```

Like `getindex`, but returns a lightweight array that lazily references (or is effectively a *view* into) the parent array `A` at the given index or indices `inds` instead of eagerly extracting elements or constructing a copied subset. Calling `getindex` or `setindex!` on the returned value (often a `SubArray`) computes the indices to access or modify the parent array on the fly. The behavior is undefined if the shape of the parent array is changed after `view` is called because there is no bound check for the parent array; e.g., it may cause a segmentation fault.

Some immutable parent arrays (like ranges) may choose to simply recompute a new array in some circumstances instead of returning a `SubArray` if doing so is efficient and provides compatible semantics.

Julia 1.6

In Julia 1.6 or later, `view` can be called on an `AbstractString`, returning a `SubString`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> b = view(A, :, 1)
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 1
```

```
 3

julia> fill!(b, 0)
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 0
 0

julia> A # Note A has changed even though we modified b
2×2 Matrix{Int64}:
 0  2
 0  4

julia> view(2:5, 2:3) # returns a range as type is immutable
3:4
```

source

Base.@view – Macro.

```
@view A[inds...]
```

Transform the indexing expression `A[inds...]` into the equivalent `view` call.

This can only be applied directly to a single indexing expression and is particularly helpful for expressions that include the special begin or end indexing syntaxes like `A[begin, 2:end-1]` (as those are not supported by the normal `view` function).

Note that `@view` cannot be used as the target of a regular assignment (e.g., `@view(A[1, 2:end]) = ...`), nor would the un-decorated indexed assignment (`A[1, 2:end] = ...`) or broadcasted indexed assignment (`A[1, 2:end] .= ...`) make a copy. It can be useful, however, for *updating* broadcasted assignments like `@view(A[1, 2:end]) .+= 1` because this is a simple syntax for `@view(A[1, 2:end]) .= @view(A[1, 2:end]) + 1`, and the indexing expression on the right-hand side would otherwise make a copy without the `@view`.

See also `@views` to switch an entire block of code to use views for non-scalar indexing.

Julia 1.5

Using begin in an indexing expression to refer to the first index requires at least Julia 1.5.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> b = @view A[:, 1]
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 1
 3
```

```
julia> fill!(b, 0)
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 0
 0

julia> A
2×2 Matrix{Int64}:
 0  2
 0  4
```

source

Base.@views – Macro.

```
@views expression
```

Convert every array-slicing operation in the given expression (which may be a begin/end block, loop, function, etc.) to return a view. Scalar indices, non-array types, and explicit getindex calls (as opposed to array[...]) are unaffected.

Similarly, @views converts string slices into SubString views.

Note

The @views macro only affects array[...] expressions that appear explicitly in the given expression, not array slicing that occurs in functions called by that code.

Julia 1.5

Using begin in an indexing expression to refer to the first index requires at least Julia 1.5.

Examples

```
julia> A = zeros(3, 3);

julia> @views for row in 1:3
           b = A[row, :]
           b[:] .= row
       end

julia> A
3×3 Matrix{Float64}:
 1.0  1.0  1.0
 2.0  2.0  2.0
 3.0  3.0  3.0
```

source

Base.parent – Function.

```
parent(A)
```

Return the underlying parent object of the view. This parent of objects of types `SubArray`, `SubString`, `ReshapedArray` or `LinearAlgebra.Transpose` is what was passed as an argument to `view`, `reshape`, `transpose`, etc. during object creation. If the input is not a wrapped object, return the input itself. If the input is wrapped multiple times, only the outermost wrapper will be removed.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> V = view(A, 1:2, :)
2×2 view(::Matrix{Int64}, 1:2, :) with eltype Int64:
 1  2
 3  4

julia> parent(V)
2×2 Matrix{Int64}:
 1  2
 3  4
```

source

`Base.parentindices` – Function.

```
parentindices(A)
```

Return the indices in the `parent` which correspond to the view `A`.

Examples

```
julia> A = [1 2; 3 4];

julia> V = view(A, 1, :)
2-element view(::Matrix{Int64}, 1, :) with eltype Int64:
 1
 2

julia> parentindices(V)
(1, Base.Slice(Base.OneTo(2)))
```

source

`Base.selectdim` – Function.

```
selectdim(A, d::Integer, i)
```

Return a view of all the data of `A` where the index for dimension `d` equals `i`.

Equivalent to `view(A,:,:,...,i,:,:,...)` where `i` is in position `d`.

See also: `eachslice`.

Examples

```
julia> A = [1 2 3 4; 5 6 7 8]
2×4 Matrix{Int64}:
 1  2  3  4
 5  6  7  8

julia> selectdim(A, 2, 3)
2-element view(::Matrix{Int64}, :, 3) with eltype Int64:
 3
 7

julia> selectdim(A, 2, 3:4)
2×2 view(::Matrix{Int64}, :, 3:4) with eltype Int64:
 3  4
 7  8
```

source

`Base.reinterpret` – Function.

```
reinterpret(::Type{Out}, x::In)
```

Change the type-interpretation of the binary data in the isbits value `x` to that of the isbits type `Out`. The size (ignoring padding) of `Out` has to be the same as that of the type of `x`. For example, `reinterpret(Float32, UInt32(7))` interprets the 4 bytes corresponding to `UInt32(7)` as a `Float32`.

```
julia> reinterpret(Float32, UInt32(7))
1.0f-44

julia> reinterpret(NTuple{2, UInt8}, 0x1234)
(0x34, 0x12)

julia> reinterpret(UInt16, (0x34, 0x12))
0x1234

julia> reinterpret(Tuple{UInt16, UInt8}, (0x01, 0x0203))
(0x0301, 0x02)
```

> **Warning**
>
> Use caution if some combinations of bits in `Out` are not considered valid and would otherwise be prevented by the type's constructors and methods. Unexpected behavior may result without additional validation.

source

```
reinterpret(T::DataType, A::AbstractArray)
```

Construct a view of the array with the same binary data as the given array, but with T as element type.

This function also works on "lazy" array whose elements are not computed until they are explicitly retrieved. For instance, `reinterpret` on the range `1:6` works similarly as on the dense vector `collect(1:6)`:

```
julia> reinterpret(Float32, UInt32[1 2 3 4 5])
1×5 reinterpret(Float32, ::Matrix{UInt32}):
 1.0f-45  3.0f-45  4.0f-45  6.0f-45  7.0f-45

julia> reinterpret(Complex{Int}, 1:6)
3-element reinterpret(Complex{Int64}, ::UnitRange{Int64}):
 1 + 2im
 3 + 4im
 5 + 6im
```

source

```
reinterpret(reshape, T, A::AbstractArray{S}) -> B
```

Change the type-interpretation of A while consuming or adding a "channel dimension."

If `sizeof(T) = n*sizeof(S)` for n>1, A's first dimension must be of size `n` and B lacks A's first dimension. Conversely, if `sizeof(S) = n*sizeof(T)` for n>1, B gets a new first dimension of size `n`. The dimensionality is unchanged if `sizeof(T) == sizeof(S)`.

> **Julia 1.6**
>
> This method requires at least Julia 1.6.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> reinterpret(reshape, Complex{Int}, A)    # the result is a vector
2-element reinterpret(reshape, Complex{Int64}, ::Matrix{Int64}) with eltype Complex{Int64}:
 1 + 3im
 2 + 4im
```

```
julia> a = [(1,2,3), (4,5,6)]
2-element Vector{Tuple{Int64, Int64, Int64}}:
 (1, 2, 3)
 (4, 5, 6)

julia> reinterpret(reshape, Int, a)             # the result is a matrix
3×2 reinterpret(reshape, Int64, ::Vector{Tuple{Int64, Int64, Int64}}) with eltype Int64:
 1  4
 2  5
 3  6
```

source

Base.reshape – Function.

```
reshape(A, dims...) -> AbstractArray
reshape(A, dims) -> AbstractArray
```

Return an array with the same data as A, but with different dimension sizes or number of dimensions. The two arrays share the same underlying data, so that the result is mutable if and only if A is mutable, and setting elements of one alters the values of the other.

The new dimensions may be specified either as a list of arguments or as a shape tuple. At most one dimension may be specified with a `:`, in which case its length is computed such that its product with all the specified dimensions is equal to the length of the original array A. The total number of elements must not change.

Examples

```
julia> A = Vector(1:16)
16-element Vector{Int64}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16

julia> reshape(A, (4, 4))
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
```

```
 4  8  12  16

julia> reshape(A, 2, :)
2×8 Matrix{Int64}:
 1  3  5  7   9  11  13  15
 2  4  6  8  10  12  14  16

julia> reshape(1:6, 2, 3)
2×3 reshape(::UnitRange{Int64}, 2, 3) with eltype Int64:
 1  3  5
 2  4  6
```

source

Base.dropdims – Function.

```
dropdims(A; dims)
```

Return an array with the same data as A, but with the dimensions specified by dims removed. size(A,d) must equal 1 for every d in dims, and repeated dimensions or numbers outside 1:ndims(A) are forbidden.

The result shares the same underlying data as A, such that the result is mutable if and only if A is mutable, and setting elements of one alters the values of the other.

See also: reshape, vec.

Examples

```
julia> a = reshape(Vector(1:4),(2,2,1,1))
2×2×1×1 Array{Int64, 4}:
[:, :, 1, 1] =
 1  3
 2  4

julia> b = dropdims(a; dims=3)
2×2×1 Array{Int64, 3}:
[:, :, 1] =
 1  3
 2  4

julia> b[1,1,1] = 5; a
2×2×1×1 Array{Int64, 4}:
[:, :, 1, 1] =
 5  3
 2  4
```

source

Base.vec – Function.

```
vec(a::AbstractArray) -> AbstractVector
```

Reshape the array a as a one-dimensional column vector. Return a if it is already an `AbstractVector`. The resulting array shares the same underlying data as a, so it will only be mutable if a is mutable, in which case modifying one will also modify the other.

Examples

```
julia> a = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> vec(a)
6-element Vector{Int64}:
 1
 4
 2
 5
 3
 6

julia> vec(1:3)
1:3
```

See also `reshape`, `dropdims`.

source

`Base.SubArray` – Type.

```
SubArray{T,N,P,I,L} <: AbstractArray{T,N}
```

N-dimensional view into a parent array (of type P) with an element type T, restricted by a tuple of indices (of type I). L is true for types that support fast linear indexing, and `false` otherwise.

Construct `SubArray`s using the `view` function.

source

47.6 拼接与排列

`Base.cat` – Function.

```
cat(A...; dims)
```

Concatenate the input arrays along the dimensions specified in `dims`.

Along a dimension `d in dims`, the size of the output array is `sum(size(a,d) for a in A)`. Along other dimensions, all input arrays should have the same size, which will also be the size of the output array along those dimensions.

If `dims` is a single number, the different arrays are tightly packed along that dimension. If `dims` is an iterable containing several dimensions, the positions along these dimensions are increased simultaneously

for each input array, filling with zero elsewhere. This allows one to construct block-diagonal matrices as `cat(matrices...; dims=(1,2))`, and their higher-dimensional analogues.

The special case `dims=1` is `vcat`, and `dims=2` is `hcat`. See also `hvcat`, `hvncat`, `stack`, `repeat`.

The keyword also accepts `Val(dims)`.

> **Julia 1.8**
>
> For multiple dimensions `dims = Val(::Tuple)` was added in Julia 1.8.

Examples

```
julia> cat([1 2; 3 4], [pi, pi], fill(10, 2,3,1); dims=2)  # same as hcat
2×6×1 Array{Float64, 3}:
[:, :, 1] =
 1.0  2.0  3.14159  10.0  10.0  10.0
 3.0  4.0  3.14159  10.0  10.0  10.0

julia> cat(true, trues(2,2), trues(4)', dims=(1,2))  # block-diagonal
4×7 Matrix{Bool}:
 1  0  0  0  0  0  0
 0  1  1  0  0  0  0
 0  1  1  0  0  0  0
 0  0  0  1  1  1  1

julia> cat(1, [2], [3;;]; dims=Val(2))
1×3 Matrix{Int64}:
 1  2  3
```

source

`Base.vcat` – Function.

```
vcat(A...)
```

Concatenate arrays or numbers vertically. Equivalent to `cat(A...; dims=1)`, and to the syntax `[a; b; c]`.

To concatenate a large vector of arrays, `reduce(vcat, A)` calls an efficient method when `A isa AbstractVector{<:Abstrac` rather than working pairwise.

See also `hcat`, `Iterators.flatten`, `stack`.

Examples

```
julia> v = vcat([1,2], [3,4])
4-element Vector{Int64}:
 1
 2
 3
 4

julia> v == vcat(1, 2, [3,4])  # accepts numbers
```

```
true

julia> v == [1; 2; [3,4]]  # syntax for the same operation
true

julia> summary(ComplexF64[1; 2; [3,4]])  # syntax for supplying the element type
"4-element Vector{ComplexF64}"

julia> vcat(range(1, 2, length=3))  # collects lazy ranges
3-element Vector{Float64}:
 1.0
 1.5
 2.0

julia> two = ([10, 20, 30]', Float64[4 5 6; 7 8 9])  # row vector and a matrix
([10 20 30], [4.0 5.0 6.0; 7.0 8.0 9.0])

julia> vcat(two...)
3×3 Matrix{Float64}:
 10.0  20.0  30.0
  4.0   5.0   6.0
  7.0   8.0   9.0

julia> vs = [[1, 2], [3, 4], [5, 6]];

julia> reduce(vcat, vs)  # more efficient than vcat(vs...)
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6

julia> ans == collect(Iterators.flatten(vs))
true
```

source

`Base.hcat` – Function.

```
hcat(A...)
```

Concatenate arrays or numbers horizontally. Equivalent to `cat(A...; dims=2)`, and to the syntax `[a b c]` or `[a;; b;; c]`.

For a large vector of arrays, `reduce(hcat, A)` calls an efficient method when `A isa AbstractVector{<:AbstractVecOrMat}`. For a vector of vectors, this can also be written `stack(A)`.

See also `vcat`, `hvcat`.

Examples

```
julia> hcat([1,2], [3,4], [5,6])
2×3 Matrix{Int64}:
 1  3  5
 2  4  6

julia> hcat(1, 2, [30 40], [5, 6, 7]')  # accepts numbers
1×7 Matrix{Int64}:
 1  2  30  40  5  6  7

julia> ans == [1 2 [30 40] [5, 6, 7]']  # syntax for the same operation
true

julia> Float32[1 2 [30 40] [5, 6, 7]']  # syntax for supplying the eltype
1×7 Matrix{Float32}:
 1.0  2.0  30.0  40.0  5.0  6.0  7.0

julia> ms = [zeros(2,2), [1 2; 3 4], [50 60; 70 80]];

julia> reduce(hcat, ms)  # more efficient than hcat(ms...)
2×6 Matrix{Float64}:
 0.0  0.0  1.0  2.0  50.0  60.0
 0.0  0.0  3.0  4.0  70.0  80.0

julia> stack(ms) |> summary  # disagrees on a vector of matrices
"2×2×3 Array{Float64, 3}"

julia> hcat(Int[], Int[], Int[])  # empty vectors, each of size (0,)
0×3 Matrix{Int64}

julia> hcat([1.1, 9.9], Matrix(undef, 2, 0))  # hcat with empty 2×0 Matrix
2×1 Matrix{Any}:
 1.1
 9.9
```

source

Base.hvcat – Function.

```
hvcat(blocks_per_row::Union{Tuple{Vararg{Int}}, Int}, values...)
```

Horizontal and vertical concatenation in one call. This function is called for block matrix syntax. The first argument specifies the number of arguments to concatenate in each block row. If the first argument is a single integer n, then all block rows are assumed to have n block columns.

Examples

```
julia> a, b, c, d, e, f = 1, 2, 3, 4, 5, 6
(1, 2, 3, 4, 5, 6)

julia> [a b c; d e f]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

```
julia> hvcat((3,3), a,b,c,d,e,f)
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> [a b; c d; e f]
3×2 Matrix{Int64}:
 1  2
 3  4
 5  6

julia> hvcat((2,2,2), a,b,c,d,e,f)
3×2 Matrix{Int64}:
 1  2
 3  4
 5  6
julia> hvcat((2,2,2), a,b,c,d,e,f) == hvcat(2, a,b,c,d,e,f)
true
```

source

Base.hvncat – Function.

```
hvncat(dim::Int, row_first, values...)
hvncat(dims::Tuple{Vararg{Int}}, row_first, values...)
hvncat(shape::Tuple{Vararg{Tuple}}, row_first, values...)
```

Horizontal, vertical, and n-dimensional concatenation of many `values` in one call.

This function is called for block matrix syntax. The first argument either specifies the shape of the concatenation, similar to `hvcat`, as a tuple of tuples, or the dimensions that specify the key number of elements along each axis, and is used to determine the output dimensions. The `dims` form is more performant, and is used by default when the concatenation operation has the same number of elements along each axis (e.g., [a b; c d;;; e f ; g h]). The `shape` form is used when the number of elements along each axis is unbalanced (e.g., [a b ; c]). Unbalanced syntax needs additional validation overhead. The `dim` form is an optimization for concatenation along just one dimension. `row_first` indicates how `values` are ordered. The meaning of the first and second elements of `shape` are also swapped based on `row_first`.

Examples

```
julia> a, b, c, d, e, f = 1, 2, 3, 4, 5, 6
(1, 2, 3, 4, 5, 6)

julia> [a b c;;; d e f]
1×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  2  3

[:, :, 2] =
 4  5  6

julia> hvncat((2,1,3), false, a,b,c,d,e,f)
```

```
2×1×3 Array{Int64, 3}:
[:, :, 1] =
 1
 2

[:, :, 2] =
 3
 4

[:, :, 3] =
 5
 6

julia> [a b;;; c d;;; e f]
1×2×3 Array{Int64, 3}:
[:, :, 1] =
 1  2

[:, :, 2] =
 3  4

[:, :, 3] =
 5  6

julia> hvncat(((3, 3), (3, 3), (6,)), true, a, b, c, d, e, f)
1×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  2  3

[:, :, 2] =
 4  5  6
```

Examples for construction of the arguments

```
[a b c ; d e f ;;;
 g h i ; j k l ;;;
 m n o ; p q r ;;;
 s t u ; v w x]
⇒ dims = (2, 3, 4)

[a b ; c ;;; d ;;;;]
 ___   _     _
 2     1     1 = elements in each row (2, 1, 1)
 _______     _
 3           1 = elements in each column (3, 1)
 _____________
 4             = elements in each 3d slice (4,)
 _____________
 4             = elements in each 4d slice (4,)
⇒ shape = ((2, 1, 1), (3, 1), (4,), (4,)) with `row_first` = true
```

source

Base.stack – Function.

```
stack(iter; [dims])
```

Combine a collection of arrays (or other iterable objects) of equal size into one larger array, by arranging them along one or more new dimensions.

By default the axes of the elements are placed first, giving `size(result) = (size(first(iter))..., size(iter)...)`. This has the same order of elements as `Iterators.flatten(iter)`.

With keyword `dims::Integer`, instead the `i`th element of `iter` becomes the slice `selectdim(result, dims, i)`, so that `size(result, dims) == length(iter)`. In this case `stack` reverses the action of `eachslice` with the same `dims`.

The various `cat` functions also combine arrays. However, these all extend the arrays' existing (possibly trivial) dimensions, rather than placing the arrays along new dimensions. They also accept arrays as separate arguments, rather than a single collection.

> **Julia 1.9**
>
> This function requires at least Julia 1.9.

Examples

```
julia> vecs = (1:2, [30, 40], Float32[500, 600]);

julia> mat = stack(vecs)
2×3 Matrix{Float32}:
 1.0  30.0  500.0
 2.0  40.0  600.0

julia> mat == hcat(vecs...) == reduce(hcat, collect(vecs))
true

julia> vec(mat) == vcat(vecs...) == reduce(vcat, collect(vecs))
true

julia> stack(zip(1:4, 10:99))  # accepts any iterators of iterators
2×4 Matrix{Int64}:
  1   2   3   4
 10  11  12  13

julia> vec(ans) == collect(Iterators.flatten(zip(1:4, 10:99)))
true

julia> stack(vecs; dims=1)  # unlike any cat function, 1st axis of vecs[1] is 2nd axis of result
3×2 Matrix{Float32}:
   1.0    2.0
  30.0   40.0
 500.0  600.0

julia> x = rand(3,4);

julia> x == stack(eachcol(x)) == stack(eachrow(x), dims=1)  # inverse of eachslice
true
```

Higher-dimensional examples:

```
julia> A = rand(5, 7, 11);

julia> E = eachslice(A, dims=2);  # a vector of matrices

julia> (element = size(first(E)), container = size(E))
(element = (5, 11), container = (7,))

julia> stack(E) |> size
(5, 11, 7)

julia> stack(E) == stack(E; dims=3) == cat(E...; dims=3)
true

julia> A == stack(E; dims=2)
true

julia> M = (fill(10i+j, 2, 3) for i in 1:5, j in 1:7);

julia> (element = size(first(M)), container = size(M))
(element = (2, 3), container = (5, 7))

julia> stack(M) |> size  # keeps all dimensions
(2, 3, 5, 7)

julia> stack(M; dims=1) |> size  # vec(container) along dims=1
(35, 2, 3)

julia> hvcat(5, M...) |> size  # hvcat puts matrices next to each other
(14, 15)
```

source

```
stack(f, args...; [dims])
```

Apply a function to each element of a collection, and `stack` the result. Or to several collections, `zip`ped together.

The function should return arrays (or tuples, or other iterators) all of the same size. These become slices of the result, each separated along `dims` (if given) or by default along the last dimensions.

See also `mapslices`, `eachcol`.

Examples

```
julia> stack(c -> (c, c-32), "julia")
2×5 Matrix{Char}:
 'j'  'u'  'l'  'i'  'a'
 'J'  'U'  'L'  'I'  'A'

julia> stack(eachrow([1 2 3; 4 5 6]), (10, 100); dims=1) do row, n
         vcat(row, row .* n, row ./ n)
       end
```

```
2×9 Matrix{Float64}:
 1.0  2.0  3.0   10.0   20.0   30.0  0.1   0.2   0.3
 4.0  5.0  6.0  400.0  500.0  600.0  0.04  0.05  0.06
```

source

Base.vect – Function.

```
vect(X...)
```

Create a Vector with element type computed from the promote_typeof of the argument, containing the argument list.

Examples

```
julia> a = Base.vect(UInt8(1), 2.5, 1//2)
3-element Vector{Float64}:
 1.0
 2.5
 0.5
```

source

Base.circshift – Function.

```
circshift(A, shifts)
```

Circularly shift, i.e. rotate, the data in an array. The second argument is a tuple or vector giving the amount to shift in each dimension, or an integer to shift only in the first dimension.

See also: circshift!, circcopy!, bitrotate, <<.

Examples

```
julia> b = reshape(Vector(1:16), (4,4))
4×4 Matrix{Int64}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> circshift(b, (0,2))
4×4 Matrix{Int64}:
  9  13  1  5
 10  14  2  6
 11  15  3  7
 12  16  4  8

julia> circshift(b, (-1,0))
4×4 Matrix{Int64}:
```

```
 2  6  10  14
 3  7  11  15
 4  8  12  16
 1  5   9  13

julia> a = BitArray([true, true, false, false, true])
5-element BitVector:
 1
 1
 0
 0
 1

julia> circshift(a, 1)
5-element BitVector:
 1
 1
 1
 0
 0

julia> circshift(a, -1)
5-element BitVector:
 1
 0
 0
 1
 1
```

source

Base.circshift! – Function.

```
circshift!(dest, src, shifts)
```

Circularly shift, i.e. rotate, the data in src, storing the result in dest. shifts specifies the amount to shift in each dimension.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also circshift.

source

Base.circcopy! – Function.

```
circcopy!(dest, src)
```

Copy `src` to `dest`, indexing each dimension modulo its length. `src` and `dest` must have the same size, but can be offset in their indices; any offset results in a (circular) wraparound. If the arrays have overlapping indices, then on the domain of the overlap `dest` agrees with `src`.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also: `circshift`.

Examples

```
julia> src = reshape(Vector(1:16), (4,4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> dest = OffsetArray{Int}(undef, (0:3,2:5))

julia> circcopy!(dest, src)
OffsetArrays.OffsetArray{Int64,2,Array{Int64,2}} with indices 0:3×2:5:
 8  12  16  4
 5   9  13  1
 6  10  14  2
 7  11  15  3

julia> dest[1:3,2:4] == src[1:3,2:4]
true
```

source

`Base.findall` – Method.

```
findall(A)
```

Return a vector `I` of the `true` indices or keys of `A`. If there are no such elements of `A`, return an empty array. To search for other kinds of values, pass a predicate as the first argument.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

See also: `findfirst`, `searchsorted`.

Examples

```
julia> A = [true, false, false, true]
4-element Vector{Bool}:
 1
 0
 0
 1
```

```
julia> findall(A)
2-element Vector{Int64}:
 1
 4

julia> A = [true false; false true]
2×2 Matrix{Bool}:
 1  0
 0  1

julia> findall(A)
2-element Vector{CartesianIndex{2}}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 2)

julia> findall(falses(3))
Int64[]
```

source

Base.findall – Method.

```
findall(f::Function, A)
```

Return a vector I of the indices or keys of A where f(A[I]) returns true. If there are no such elements of A, return an empty array.

Indices or keys are of the same type as those returned by keys(A) and pairs(A).

Examples

```
julia> x = [1, 3, 4]
3-element Vector{Int64}:
 1
 3
 4

julia> findall(isodd, x)
2-element Vector{Int64}:
 1
 2

julia> A = [1 2 0; 3 4 0]
2×3 Matrix{Int64}:
 1  2  0
 3  4  0
julia> findall(isodd, A)
2-element Vector{CartesianIndex{2}}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)

julia> findall(!iszero, A)
```

```
4-element Vector{CartesianIndex{2}}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)
 CartesianIndex(1, 2)
 CartesianIndex(2, 2)

julia> d = Dict(:A => 10, :B => -1, :C => 0)
Dict{Symbol, Int64} with 3 entries:
  :A => 10
  :B => -1
  :C => 0

julia> findall(x -> x >= 0, d)
2-element Vector{Symbol}:
 :A
 :C
```

source

Base.findfirst – Method.

```
findfirst(A)
```

Return the index or key of the first `true` value in `A`. Return `nothing` if no such value is found. To search for other kinds of values, pass a predicate as the first argument.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

See also: `findall`, `findnext`, `findlast`, `searchsortedfirst`.

Examples

```
julia> A = [false, false, true, false]
4-element Vector{Bool}:
 0
 0
 1
 0

julia> findfirst(A)
3

julia> findfirst(falses(3)) # returns nothing, but not printed in the REPL

julia> A = [false false; true false]
2×2 Matrix{Bool}:
 0  0
 1  0

julia> findfirst(A)
CartesianIndex(2, 1)
```

source

Base.findfirst – Method.

```
findfirst(predicate::Function, A)
```

Return the index or key of the first element of A for which `predicate` returns `true`. Return `nothing` if there is no such element.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [1, 4, 2, 2]
4-element Vector{Int64}:
 1
 4
 2
 2

julia> findfirst(iseven, A)
2

julia> findfirst(x -> x>10, A) # returns nothing, but not printed in the REPL

julia> findfirst(isequal(4), A)
2

julia> A = [1 4; 2 2]
2×2 Matrix{Int64}:
 1  4
 2  2

julia> findfirst(iseven, A)
CartesianIndex(2, 1)
```

source

Base.findlast – Method.

```
findlast(A)
```

Return the index or key of the last `true` value in A. Return `nothing` if there is no `true` value in A.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

See also: `findfirst`, `findprev`, `findall`.

Examples

```
julia> A = [true, false, true, false]
4-element Vector{Bool}:
 1
 0
 1
```

```
 0

julia> findlast(A)
3

julia> A = falses(2,2);

julia> findlast(A) # returns nothing, but not printed in the REPL

julia> A = [true false; true false]
2×2 Matrix{Bool}:
 1  0
 1  0

julia> findlast(A)
CartesianIndex(2, 1)
```

source

Base.findlast – Method.

```
findlast(predicate::Function, A)
```

Return the index or key of the last element of A for which `predicate` returns `true`. Return `nothing` if there is no such element.

Indices or keys are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [1, 2, 3, 4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> findlast(isodd, A)
3

julia> findlast(x -> x > 5, A) # returns nothing, but not printed in the REPL

julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> findlast(isodd, A)
CartesianIndex(2, 1)
```

source

Base.findnext – Method.

```
findnext(A, i)
```

Find the next index after or including `i` of a `true` element of `A`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [false, false, true, false]
4-element Vector{Bool}:
 0
 0
 1
 0

julia> findnext(A, 1)
3

julia> findnext(A, 4) # returns nothing, but not printed in the REPL

julia> A = [false false; true false]
2×2 Matrix{Bool}:
 0  0
 1  0

julia> findnext(A, CartesianIndex(1, 1))
CartesianIndex(2, 1)
```

source

`Base.findnext` – Method.

```
findnext(predicate::Function, A, i)
```

Find the next index after or including `i` of an element of `A` for which `predicate` returns `true`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [1, 4, 2, 2];

julia> findnext(isodd, A, 1)
1

julia> findnext(isodd, A, 2) # returns nothing, but not printed in the REPL

julia> A = [1 4; 2 2];

julia> findnext(isodd, A, CartesianIndex(1, 1))
CartesianIndex(1, 1)
```

source

Base.findprev – Method.

```
findprev(A, i)
```

Find the previous index before or including `i` of a `true` element of `A`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

See also: `findnext`, `findfirst`, `findall`.

Examples

```
julia> A = [false, false, true, true]
4-element Vector{Bool}:
 0
 0
 1
 1

julia> findprev(A, 3)
3

julia> findprev(A, 1) # returns nothing, but not printed in the REPL

julia> A = [false false; true true]
2×2 Matrix{Bool}:
 0  0
 1  1

julia> findprev(A, CartesianIndex(2, 1))
CartesianIndex(2, 1)
```

source

Base.findprev – Method.

```
findprev(predicate::Function, A, i)
```

Find the previous index before or including `i` of an element of `A` for which `predicate` returns `true`, or `nothing` if not found.

Indices are of the same type as those returned by `keys(A)` and `pairs(A)`.

Examples

```
julia> A = [4, 6, 1, 2]
4-element Vector{Int64}:
 4
 6
 1
 2
```

```
julia> findprev(isodd, A, 1) # returns nothing, but not printed in the REPL

julia> findprev(isodd, A, 3)
3

julia> A = [4 6; 1 2]
2×2 Matrix{Int64}:
 4  6
 1  2

julia> findprev(isodd, A, CartesianIndex(1, 2))
CartesianIndex(2, 1)
```

source

Base.permutedims – Function.

```
permutedims(A::AbstractArray, perm)
```

Permute the dimensions of array A. perm is a vector or a tuple of length ndims(A) specifying the permutation.

See also permutedims!, PermutedDimsArray, transpose, invperm.

Examples

```
julia> A = reshape(Vector(1:8), (2,2,2))
2×2×2 Array{Int64, 3}:
[:, :, 1] =
 1  3
 2  4

[:, :, 2] =
 5  7
 6  8

julia> perm = (3, 1, 2); # put the last dimension first

julia> B = permutedims(A, perm)
2×2×2 Array{Int64, 3}:
[:, :, 1] =
 1  2
 5  6

[:, :, 2] =
 3  4
 7  8

julia> A == permutedims(B, invperm(perm)) # the inverse permutation
true
```

For each dimension i of B = permutedims(A, perm), its corresponding dimension of A will be perm[i]. This means the equality size(B, i) == size(A, perm[i]) holds.

```
julia> A = randn(5, 7, 11, 13);

julia> perm = [4, 1, 3, 2];

julia> B = permutedims(A, perm);

julia> size(B)
(13, 5, 11, 7)

julia> size(A)[perm] == ans
true
```

source

```
permutedims(m::AbstractMatrix)
```

Permute the dimensions of the matrix `m`, by flipping the elements across the diagonal of the matrix. Differs from `LinearAlgebra`'s `transpose` in that the operation is not recursive.

Examples

```
julia> a = [1 2; 3 4];

julia> b = [5 6; 7 8];

julia> c = [9 10; 11 12];

julia> d = [13 14; 15 16];

julia> X = [[a] [b]; [c] [d]]
2×2 Matrix{Matrix{Int64}}:
 [1 2; 3 4]     [5 6; 7 8]
 [9 10; 11 12]  [13 14; 15 16]

julia> permutedims(X)
2×2 Matrix{Matrix{Int64}}:
 [1 2; 3 4]  [9 10; 11 12]
 [5 6; 7 8]  [13 14; 15 16]

julia> transpose(X)
2×2 transpose(::Matrix{Matrix{Int64}}) with eltype Transpose{Int64, Matrix{Int64}}:
 [1 3; 2 4]  [9 11; 10 12]
 [5 7; 6 8]  [13 15; 14 16]
```

source

```
permutedims(v::AbstractVector)
```

Reshape vector v into a `1 × length(v)` row matrix. Differs from `LinearAlgebra`'s `transpose` in that the operation is not recursive.

Examples

```
julia> permutedims([1, 2, 3, 4])
1×4 Matrix{Int64}:
 1  2  3  4

julia> V = [[[1 2; 3 4]]; [[5 6; 7 8]]]
2-element Vector{Matrix{Int64}}:
 [1 2; 3 4]
 [5 6; 7 8]

julia> permutedims(V)
1×2 Matrix{Matrix{Int64}}:
 [1 2; 3 4]  [5 6; 7 8]

julia> transpose(V)
1×2 transpose(::Vector{Matrix{Int64}}) with eltype Transpose{Int64, Matrix{Int64}}:
 [1 3; 2 4]  [5 7; 6 8]
```

source

`Base.permutedims!` – Function.

```
permutedims!(dest, src, perm)
```

Permute the dimensions of array `src` and store the result in the array `dest`. `perm` is a vector specifying a permutation of length `ndims(src)`. The preallocated array `dest` should have `size(dest) == size(src)[perm]` and is completely overwritten. No in-place permutation is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

See also `permutedims`.

source

`Base.PermutedDimsArrays.PermutedDimsArray` – Type.

```
PermutedDimsArray(A, perm) -> B
```

Given an AbstractArray A, create a view B such that the dimensions appear to be permuted. Similar to `permutedims`, except that no copying occurs (B shares storage with A).

See also `permutedims`, `invperm`.

Examples

```
julia> A = rand(3,5,4);

julia> B = PermutedDimsArray(A, (3,1,2));

julia> size(B)
(4, 3, 5)

julia> B[3,1,2] == A[1,2,3]
true
```

source

Base.promote_shape – Function.

```
promote_shape(s1, s2)
```

Check two array shapes for compatibility, allowing trailing singleton dimensions, and return whichever shape has more dimensions.

Examples

```
julia> a = fill(1, (3,4,1,1,1));

julia> b = fill(1, (3,4));

julia> promote_shape(a,b)
(Base.OneTo(3), Base.OneTo(4), Base.OneTo(1), Base.OneTo(1), Base.OneTo(1))

julia> promote_shape((2,3,1,4), (2, 3, 1, 4, 1))
(2, 3, 1, 4, 1)
```

source

47.7 数组函数

Base.accumulate – Function.

```
accumulate(op, A; dims::Integer, [init])
```

Cumulative operation op along the dimension dims of A (providing dims is optional for vectors). An initial value init may optionally be provided by a keyword argument. See also accumulate! to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

For common operations there are specialized variants of accumulate, see cumsum, cumprod. For a lazy version, see Iterators.accumulate.

Julia 1.5

accumulate on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> accumulate(+, [1,2,3])
3-element Vector{Int64}:
 1
 3
 6

julia> accumulate(min, (1, -2, 3, -4, 5), init=0)
(0, -2, -2, -4, -4)
```

```
julia> accumulate(/, (2, 4, Inf), init=100)
(50.0, 12.5, 0.0)

julia> accumulate(=>, i^2 for i in 1:3)
3-element Vector{Any}:
          1
        1 => 4
 (1 => 4) => 9

julia> accumulate(+, fill(1, 3, 4))
3×4 Matrix{Int64}:
 1  4  7  10
 2  5  8  11
 3  6  9  12

julia> accumulate(+, fill(1, 2, 5), dims=2, init=100.0)
2×5 Matrix{Float64}:
 101.0  102.0  103.0  104.0  105.0
 101.0  102.0  103.0  104.0  105.0
```

source

Base.accumulate! – Function.

```
accumulate!(op, B, A; [dims], [init])
```

Cumulative operation op on A along the dimension `dims`, storing the result in B. Providing `dims` is optional for vectors. If the keyword argument `init` is given, its value is used to instantiate the accumulation.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also `accumulate`, `cumsum!`, `cumprod!`.

Examples

```
julia> x = [1, 0, 2, 0, 3];

julia> y = rand(5);

julia> accumulate!(+, y, x);

julia> y
5-element Vector{Float64}:
 1.0
 1.0
 3.0
 3.0
 6.0
```

```
julia> A = [1 2 3; 4 5 6];

julia> B = similar(A);

julia> accumulate!(-, B, A, dims=1)
2×3 Matrix{Int64}:
  1   2   3
 -3  -3  -3

julia> accumulate!(*, B, A, dims=2, init=10)
2×3 Matrix{Int64}:
 10   20    60
 40  200  1200
```

source

Base.cumprod – Function.

```
cumprod(A; dims::Integer)
```

Cumulative product along the dimension `dim`. See also `cumprod!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

Examples

```
julia> a = Int8[1 2 3; 4 5 6];

julia> cumprod(a, dims=1)
2×3 Matrix{Int64}:
 1   2   3
 4  10  18

julia> cumprod(a, dims=2)
2×3 Matrix{Int64}:
 1   2    6
 4  20  120
```

source

```
cumprod(itr)
```

Cumulative product of an iterator.

See also `cumprod!`, `accumulate`, `cumsum`.

Julia 1.5

`cumprod` on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> cumprod(fill(1//2, 3))
3-element Vector{Rational{Int64}}:
 1//2
 1//4
 1//8

julia> cumprod((1, 2, 1, 3, 1))
(1, 2, 2, 6, 6)

julia> cumprod("julia")
5-element Vector{String}:
 "j"
 "ju"
 "jul"
 "juli"
 "julia"
```

source

Base.cumprod! – Function.

```
cumprod!(B, A; dims::Integer)
```

Cumulative product of A along the dimension dims, storing the result in B. See also cumprod.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

```
cumprod!(y::AbstractVector, x::AbstractVector)
```

Cumulative product of a vector x, storing the result in y. See also cumprod.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

Base.cumsum – Function.

```
cumsum(A; dims::Integer)
```

Cumulative sum along the dimension dims. See also cumsum! to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

Examples

```
julia> a = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> cumsum(a, dims=1)
2×3 Matrix{Int64}:
 1  2  3
 5  7  9

julia> cumsum(a, dims=2)
2×3 Matrix{Int64}:
 1  3   6
 4  9  15
```

Note

The return array's `eltype` is `Int` for signed integers of less than system word size and `UInt` for unsigned integers of less than system word size. To preserve `eltype` of arrays with small signed or unsigned integer `accumulate(+, A)` should be used.

```
julia> cumsum(Int8[100, 28])
2-element Vector{Int64}:
 100
 128

julia> accumulate(+,Int8[100, 28])
2-element Vector{Int8}:
  100
 -128
```

In the former case, the integers are widened to system word size and therefore the result is `Int64[100, 128]`. In the latter case, no such widening happens and integer overflow results in `Int8[100, -128]`.

source

```
cumsum(itr)
```

Cumulative sum of an iterator.

See also `accumulate` to apply functions other than `+`.

Julia 1.5

`cumsum` on a non-array iterator requires at least Julia 1.5.

Examples

```
julia> cumsum(1:3)
3-element Vector{Int64}:
 1
 3
 6

julia> cumsum((true, false, true, false, true))
(1, 1, 2, 2, 3)

julia> cumsum(fill(1, 2) for i in 1:3)
3-element Vector{Vector{Int64}}:
 [1, 1]
 [2, 2]
 [3, 3]
```

source

Base.cumsum! – Function.

```
cumsum!(B, A; dims::Integer)
```

Cumulative sum of A along the dimension dims, storing the result in B. See also cumsum.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

Base.diff – Function.

```
diff(A::AbstractVector)
diff(A::AbstractArray; dims::Integer)
```

Finite difference operator on a vector or a multidimensional array A. In the latter case the dimension to operate on needs to be specified with the dims keyword argument.

Julia 1.1

diff for arrays with dimension higher than 2 requires at least Julia 1.1.

Examples

```
julia> a = [2 4; 6 16]
2×2 Matrix{Int64}:
 2   4
 6  16
```

```
julia> diff(a, dims=2)
2×1 Matrix{Int64}:
  2
 10

julia> diff(vec(a))
3-element Vector{Int64}:
  4
 -2
 12
```

source

Base.repeat – Function.

```
repeat(A::AbstractArray, counts::Integer...)
```

Construct an array by repeating array A a given number of times in each dimension, specified by `counts`.

See also: `fill`, `Iterators.repeated`, `Iterators.cycle`.

Examples

```
julia> repeat([1, 2, 3], 2)
6-element Vector{Int64}:
 1
 2
 3
 1
 2
 3

julia> repeat([1, 2, 3], 2, 3)
6×3 Matrix{Int64}:
 1  1  1
 2  2  2
 3  3  3
 1  1  1
 2  2  2
 3  3  3
```

source

```
repeat(A::AbstractArray; inner=ntuple(Returns(1), ndims(A)), outer=ntuple(Returns(1), ndims(A)))
```

Construct an array by repeating the entries of A. The i-th element of `inner` specifies the number of times that the individual entries of the i-th dimension of A should be repeated. The i-th element of `outer` specifies the number of times that a slice along the i-th dimension of A should be repeated. If `inner` or `outer` are omitted, no repetition is performed.

Examples

```
julia> repeat(1:2, inner=2)
4-element Vector{Int64}:
 1
 1
 2
 2

julia> repeat(1:2, outer=2)
4-element Vector{Int64}:
 1
 2
 1
 2

julia> repeat([1 2; 3 4], inner=(2, 1), outer=(1, 3))
4×6 Matrix{Int64}:
 1  2  1  2  1  2
 1  2  1  2  1  2
 3  4  3  4  3  4
 3  4  3  4  3  4
```

source

```
repeat(s::AbstractString, r::Integer)
```

Repeat a string r times. This can be written as s^r.

See also ^.

Examples

```
julia> repeat("ha", 3)
"hahaha"
```

source

```
repeat(c::AbstractChar, r::Integer) -> String
```

Repeat a character r times. This can equivalently be accomplished by calling c^r.

Examples

```
julia> repeat('A', 3)
"AAA"
```

source

Base.rot180 – Function.

```
rot180(A)
```

Rotate matrix A 180 degrees.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> rot180(a)
2×2 Matrix{Int64}:
 4  3
 2  1
```

source

```
rot180(A, k)
```

Rotate matrix A 180 degrees an integer k number of times. If k is even, this is equivalent to a copy.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> rot180(a,1)
2×2 Matrix{Int64}:
 4  3
 2  1

julia> rot180(a,2)
2×2 Matrix{Int64}:
 1  2
 3  4
```

source

`Base.rotl90` – Function.

```
rotl90(A)
```

Rotate matrix A left 90 degrees.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> rotl90(a)
2×2 Matrix{Int64}:
 2  4
 1  3
```

source

```
rotl90(A, k)
```

Left-rotate matrix A 90 degrees counterclockwise an integer k number of times. If k is a multiple of four (including zero), this is equivalent to a copy.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> rotl90(a,1)
2×2 Matrix{Int64}:
 2  4
 1  3

julia> rotl90(a,2)
2×2 Matrix{Int64}:
 4  3
 2  1

julia> rotl90(a,3)
2×2 Matrix{Int64}:
 3  1
 4  2

julia> rotl90(a,4)
2×2 Matrix{Int64}:
 1  2
 3  4
```

source

Base.rotr90 – Function.

```
rotr90(A)
```

Rotate matrix A right 90 degrees.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> rotr90(a)
2×2 Matrix{Int64}:
 3  1
 4  2
```

source

```
rotr90(A, k)
```

Right-rotate matrix A 90 degrees clockwise an integer k number of times. If k is a multiple of four (including zero), this is equivalent to a copy.

Examples

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> rotr90(a,1)
2×2 Matrix{Int64}:
 3  1
 4  2

julia> rotr90(a,2)
2×2 Matrix{Int64}:
 4  3
 2  1

julia> rotr90(a,3)
2×2 Matrix{Int64}:
 2  4
 1  3

julia> rotr90(a,4)
2×2 Matrix{Int64}:
 1  2
 3  4
```

source

Base.mapslices – Function.

```
mapslices(f, A; dims)
```

Transform the given dimensions of array A by applying a function f on each slice of the form A[..., :, ..., :, ...], with a colon at each d in dims. The results are concatenated along the remaining dimensions.

For example, if dims = [1,2] and A is 4-dimensional, then f is called on x = A[:,:,i,j] for all i and j, and f(x) becomes R[:,:,i,j] in the result R.

See also eachcol or eachslice, used with map or stack.

Examples

```
julia> A = reshape(1:30,(2,5,3))
2×5×3 reshape(::UnitRange{Int64}, 2, 5, 3) with eltype Int64:
[:, :, 1] =
 1  3  5  7   9
 2  4  6  8  10

[:, :, 2] =
 11  13  15  17  19
 12  14  16  18  20

[:, :, 3] =
 21  23  25  27  29
 22  24  26  28  30

julia> f(x::Matrix) = fill(x[1,1], 1,4);  # returns a 1×4 matrix

julia> B = mapslices(f, A, dims=(1,2))
1×4×3 Array{Int64, 3}:
[:, :, 1] =
 1  1  1  1

[:, :, 2] =
 11  11  11  11

[:, :, 3] =
 21  21  21  21

julia> f2(x::AbstractMatrix) = fill(x[1,1], 1,4);

julia> B == stack(f2, eachslice(A, dims=3))
true

julia> g(x) = x[begin] // x[end-1];  # returns a number

julia> mapslices(g, A, dims=[1,3])
1×5×1 Array{Rational{Int64}, 3}:
[:, :, 1] =
 1//21  3//23  1//5  7//27  9//29

julia> map(g, eachslice(A, dims=2))
5-element Vector{Rational{Int64}}:
 1//21
 3//23
 1//5
 7//27
 9//29
```

```
julia> mapslices(sum, A; dims=(1,3)) == sum(A; dims=(1,3))
true
```

Notice that in `eachslice(A; dims=2)`, the specified dimension is the one *without* a colon in the slice. This is `view(A,:,i,:)`, whereas `mapslices(f, A; dims=(1,3))` uses `A[:,i,:]`. The function `f` may mutate values in the slice without affecting `A`.

source

`Base.eachrow` – Function.

```
eachrow(A::AbstractVecOrMat) <: AbstractVector
```

Create a `RowSlices` object that is a vector of rows of matrix or vector `A`. Row slices are returned as `AbstractVector` views of `A`.

For the inverse, see `stack(rows; dims=1)`.

See also `eachcol`, `eachslice` and `mapslices`.

Julia 1.1

This function requires at least Julia 1.1.

Julia 1.9

Prior to Julia 1.9, this returned an iterator.

Example

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> s = eachrow(a)
2-element RowSlices{Matrix{Int64}, Tuple{Base.OneTo{Int64}}, SubArray{Int64, 1, Matrix{Int64},
↪  Tuple{Int64, Base.Slice{Base.OneTo{Int64}}}, true}}:
 [1, 2]
 [3, 4]

julia> s[1]
2-element view(::Matrix{Int64}, 1, :) with eltype Int64:
 1
 2
```

source

`Base.eachcol` – Function.

```
eachcol(A::AbstractVecOrMat) <: AbstractVector
```

Create a `ColumnSlices` object that is a vector of columns of matrix or vector `A`. Column slices are returned as `AbstractVector` views of `A`.

For the inverse, see `stack(cols)` or `reduce(hcat, cols)`.

See also `eachrow`, `eachslice` and `mapslices`.

> **Julia 1.1**
>
> This function requires at least Julia 1.1.

> **Julia 1.9**
>
> Prior to Julia 1.9, this returned an iterator.

Example

```
julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> s = eachcol(a)
2-element ColumnSlices{Matrix{Int64}, Tuple{Base.OneTo{Int64}}, SubArray{Int64, 1,
↪  Matrix{Int64}, Tuple{Base.Slice{Base.OneTo{Int64}}, Int64}, true}}:
 [1, 3]
 [2, 4]

julia> s[1]
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 1
 3
```

source

`Base.eachslice` – Function.

```
eachslice(A::AbstractArray; dims, drop=true)
```

Create a `Slices` object that is an array of slices over dimensions `dims` of `A`, returning views that select all the data from the other dimensions in `A`. `dims` can either by an integer or a tuple of integers.

If `drop = true` (the default), the outer `Slices` will drop the inner dimensions, and the ordering of the dimensions will match those in `dims`. If `drop = false`, then the `Slices` will have the same dimensionality as the underlying array, with inner dimensions having size 1.

See `stack(slices; dims)` for the inverse of `eachslice(A; dims::Integer)`.

See also `eachrow`, `eachcol`, `mapslices` and `selectdim`.

Julia 1.1

This function requires at least Julia 1.1.

Julia 1.9

Prior to Julia 1.9, this returned an iterator, and only a single dimension `dims` was supported.

Example

```
julia> m = [1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> s = eachslice(m, dims=1)
3-element RowSlices{Matrix{Int64}, Tuple{Base.OneTo{Int64}}, SubArray{Int64, 1, Matrix{Int64},
↪  Tuple{Int64, Base.Slice{Base.OneTo{Int64}}}, true}}:
 [1, 2, 3]
 [4, 5, 6]
 [7, 8, 9]

julia> s[1]
3-element view(::Matrix{Int64}, 1, :) with eltype Int64:
 1
 2
 3

julia> eachslice(m, dims=1, drop=false)
3×1 Slices{Matrix{Int64}, Tuple{Int64, Colon}, Tuple{Base.OneTo{Int64}, Base.OneTo{Int64}},
↪  SubArray{Int64, 1, Matrix{Int64}, Tuple{Int64, Base.Slice{Base.OneTo{Int64}}}, true}, 2}:
 [1, 2, 3]
 [4, 5, 6]
 [7, 8, 9]
```

source

47.8 组合学

`Base.invperm` – Function.

```
invperm(v)
```

Return the inverse permutation of `v`. If `B = A[v]`, then `A == B[invperm(v)]`.

See also `sortperm`, `invpermute!`, `isperm`, `permutedims`.

Examples

```
julia> p = (2, 3, 1);

julia> invperm(p)
(3, 1, 2)

julia> v = [2; 4; 3; 1];

julia> invperm(v)
4-element Vector{Int64}:
 4
 1
 3
 2

julia> A = ['a','b','c','d'];

julia> B = A[v]
4-element Vector{Char}:
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> B[invperm(v)]
4-element Vector{Char}:
 'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
```

source

`Base.isperm` – Function.

```
isperm(v) -> Bool
```

Return `true` if v is a valid permutation.

Examples

```
julia> isperm([1; 2])
true

julia> isperm([1; 3])
false
```

source

`Base.permute!` – Method.

```
permute!(v, p)
```

Permute vector v in-place, according to permutation p. No checking is done to verify that p is a permutation.

To return a new permutation, use `v[p]`. This is generally faster than `permute!(v, p)`; it is even faster to write into a pre-allocated output array with `u .= @view v[p]`. (Even though `permute!` overwrites v in-place, it internally requires some allocation to keep track of which elements have been moved.)

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

See also `invpermute!`.

Examples

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> permute!(A, perm);

julia> A
4-element Vector{Int64}:
 1
 4
 3
 1
```

source

`Base.invpermute!` – Function.

```
invpermute!(v, p)
```

Like `permute!`, but the inverse of the given permutation is applied.

Note that if you have a pre-allocated output array (e.g. `u = similar(v)`), it is quicker to instead employ `u[p] = v`. (`invpermute!` internally allocates a copy of the data.)

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> invpermute!(A, perm);

julia> A
4-element Vector{Int64}:
 4
 1
 3
 1
```

source

Base.reverse – Method.

```
reverse(A; dims=:)
```

Reverse A along dimension dims, which can be an integer (a single dimension), a tuple of integers (a tuple of dimensions) or : (reverse along all the dimensions, the default). See also reverse! for in-place reversal.

Examples

```
julia> b = Int64[1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> reverse(b, dims=2)
2×2 Matrix{Int64}:
 2  1
 4  3

julia> reverse(b)
2×2 Matrix{Int64}:
 4  3
 2  1
```

Julia 1.6

Prior to Julia 1.6, only single-integer dims are supported in reverse.

source

Base.reverseind – Function.

```
reverseind(v, i)
```

Given an index i in reverse(v), return the corresponding index in v so that v[reverseind(v,i)] == reverse(v)[i]. (This can be nontrivial in cases where v contains non-ASCII characters.)

Examples

```
julia> s = "Julia□"
"Julia□"

julia> r = reverse(s)
"□ailuJ"

julia> for i in eachindex(s)
           print(r[reverseind(r, i)])
       end
Julia□
```

source

Base.reverse! – Function.

```
reverse!(v [, start=firstindex(v) [, stop=lastindex(v) ]]) -> v
```

In-place version of reverse.

Examples

```
julia> A = Vector(1:5)
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> reverse!(A);

julia> A
5-element Vector{Int64}:
 5
 4
 3
 2
 1
```

source

```
reverse!(A; dims=:)
```

Like reverse, but operates in-place in A.

Julia 1.6

Multidimensional `reverse!` requires Julia 1.6.

source

Chapter 48

Tasks

`Core.Task` – Type.

```
Task(func)
```

Create a `Task` (i.e. coroutine) to execute the given function `func` (which must be callable with no arguments). The task exits when this function returns. The task will run in the "world age" from the parent at construction when `schedule`d.

> **Warning**
>
> By default tasks will have the sticky bit set to true `t.sticky`. This models the historic default for `@async`. Sticky tasks can only be run on the worker thread they are first scheduled on, and when scheduled will make the task that they were scheduled from sticky. To obtain the behavior of `Threads.@spawn` set the sticky bit manually to `false`.

Examples

```
julia> a() = sum(i for i in 1:1000);

julia> b = Task(a);
```

In this example, `b` is a runnable `Task` that hasn't started yet.

source

`Base.@task` – Macro.

```
@task
```

Wrap an expression in a `Task` without executing it, and return the `Task`. This only creates a task, and does not run it.

> **Warning**
>
> By default tasks will have the sticky bit set to true `t.sticky`. This models the historic default for `@async`. Sticky tasks can only be run on the worker thread they are first scheduled on, and when scheduled will make the task that they were scheduled from sticky. To obtain the behavior of `Threads.@spawn` set the sticky bit manually to `false`.

Examples

```
julia> a1() = sum(i for i in 1:1000);

julia> b = @task a1();

julia> istaskstarted(b)
false

julia> schedule(b);

julia> yield();

julia> istaskdone(b)
true
```

source

`Base.@async` – Macro.

```
@async
```

Wrap an expression in a `Task` and add it to the local machine's scheduler queue.

Values can be interpolated into `@async` via `$`, which copies the value directly into the constructed underlying closure. This allows you to insert the *value* of a variable, isolating the asynchronous code from changes to the variable's value in the current task.

> **Warning**
>
> It is strongly encouraged to favor `Threads.@spawn` over `@async` always **even when no parallelism is required** especially in publicly distributed libraries. This is because a use of `@async` disables the migration of the *parent* task across worker threads in the current implementation of Julia. Thus, seemingly innocent use of `@async` in a library function can have a large impact on the performance of very different parts of user applications.

> **Julia 1.4**
>
> Interpolating values via `$` is available as of Julia 1.4.

source

`Base.asyncmap` – Function.

```
asyncmap(f, c...; ntasks=0, batch_size=nothing)
```

Uses multiple concurrent tasks to map `f` over a collection (or multiple equal length collections). For multiple collection arguments, `f` is applied elementwise.

`ntasks` specifies the number of tasks to run concurrently. Depending on the length of the collections, if `ntasks` is unspecified, up to 100 tasks will be used for concurrent mapping.

`ntasks` can also be specified as a zero-arg function. In this case, the number of tasks to run in parallel is checked before processing every element and a new task started if the value of `ntasks_func` is greater than the current number of tasks.

If `batch_size` is specified, the collection is processed in batch mode. `f` must then be a function that must accept a `Vector` of argument tuples and must return a vector of results. The input vector will have a length of `batch_size` or less.

The following examples highlight execution in different tasks by returning the `objectid` of the tasks in which the mapping function is executed.

First, with `ntasks` undefined, each element is processed in a different task.

```
julia> tskoid() = objectid(current_task());

julia> asyncmap(x->tskoid(), 1:5)
5-element Array{UInt64,1}:
 0x6e15e66c75c75853
 0x440f8819a1baa682
 0x9fb3eeadd0c83985
 0xebd3e35fe90d4050
 0x29efc93edce2b961

julia> length(unique(asyncmap(x->tskoid(), 1:5)))
5
```

With `ntasks=2` all elements are processed in 2 tasks.

```
julia> asyncmap(x->tskoid(), 1:5; ntasks=2)
5-element Array{UInt64,1}:
 0x027ab1680df7ae94
 0xa23d2f80cd7cf157
 0x027ab1680df7ae94
 0xa23d2f80cd7cf157
 0x027ab1680df7ae94

julia> length(unique(asyncmap(x->tskoid(), 1:5; ntasks=2)))
2
```

With `batch_size` defined, the mapping function needs to be changed to accept an array of argument tuples and return an array of results. `map` is used in the modified mapping function to achieve this.

```
julia> batch_func(input) = map(x->string("args_tuple: ", x, ", element_val: ", x[1], ", task: ",
↪  tskoid()), input)
batch_func (generic function with 1 method)
```

```
julia> asyncmap(batch_func, 1:5; ntasks=2, batch_size=2)
5-element Array{String,1}:
 "args_tuple: (1,), element_val: 1, task: 9118321258196414413"
 "args_tuple: (2,), element_val: 2, task: 4904288162898683522"
 "args_tuple: (3,), element_val: 3, task: 9118321258196414413"
 "args_tuple: (4,), element_val: 4, task: 4904288162898683522"
 "args_tuple: (5,), element_val: 5, task: 9118321258196414413"
```

source

`Base.asyncmap!` – Function.

```
asyncmap!(f, results, c...; ntasks=0, batch_size=nothing)
```

Like `asyncmap`, but stores output in `results` rather than returning a collection.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

`Base.current_task` – Function.

```
current_task()
```

Get the currently running `Task`.

source

`Base.istaskdone` – Function.

```
istaskdone(t::Task) -> Bool
```

Determine whether a task has exited.

Examples

```
julia> a2() = sum(i for i in 1:1000);

julia> b = Task(a2);

julia> istaskdone(b)
false

julia> schedule(b);
```

```
julia> yield();

julia> istaskdone(b)
true
```

source

Base.istaskstarted – Function.

```
istaskstarted(t::Task) -> Bool
```

Determine whether a task has started executing.

Examples

```
julia> a3() = sum(i for i in 1:1000);

julia> b = Task(a3);

julia> istaskstarted(b)
false
```

source

Base.istaskfailed – Function.

```
istaskfailed(t::Task) -> Bool
```

Determine whether a task has exited because an exception was thrown.

Examples

```
julia> a4() = error("task failed");

julia> b = Task(a4);

julia> istaskfailed(b)
false

julia> schedule(b);

julia> yield();

julia> istaskfailed(b)
true
```

Julia 1.3

This function requires at least Julia 1.3.

source

Base.task_local_storage – Method.

```
task_local_storage(key)
```

Look up the value of a key in the current task's task-local storage.

source

Base.task_local_storage – Method.

```
task_local_storage(key, value)
```

Assign a value to a key in the current task's task-local storage.

source

Base.task_local_storage – Method.

```
task_local_storage(body, key, value)
```

Call the function body with a modified task-local storage, in which value is assigned to key; the previous value of key, or lack thereof, is restored afterwards. Useful for emulating dynamic scoping.

source

48.1 Scheduling

Base.yield – Function.

```
yield()
```

Switch to the scheduler to allow another scheduled task to run. A task that calls this function is still runnable, and will be restarted immediately if there are no other runnable tasks.

source

```
yield(t::Task, arg = nothing)
```

A fast, unfair-scheduling version of schedule(t, arg); yield() which immediately yields to t before calling the scheduler.

source

Base.yieldto – Function.

```
yieldto(t::Task, arg = nothing)
```

Switch to the given task. The first time a task is switched to, the task's function is called with no arguments. On subsequent switches, `arg` is returned from the task's last call to `yieldto`. This is a low-level call that only switches tasks, not considering states or scheduling in any way. Its use is discouraged.

source

`Base.sleep` – Function.

```
sleep(seconds)
```

Block the current task for a specified number of seconds. The minimum sleep time is 1 millisecond or input of `0.001`.

source

`Base.schedule` – Function.

```
schedule(t::Task, [val]; error=false)
```

Add a Task to the scheduler's queue. This causes the task to run constantly when the system is otherwise idle, unless the task performs a blocking operation such as `wait`.

If a second argument `val` is provided, it will be passed to the task (via the return value of `yieldto`) when it runs again. If `error` is `true`, the value is raised as an exception in the woken task.

Warning

It is incorrect to use `schedule` on an arbitrary `Task` that has already been started. See the API reference for more information.

Warning

By default tasks will have the sticky bit set to true `t.sticky`. This models the historic default for `@async`. Sticky tasks can only be run on the worker thread they are first scheduled on, and when scheduled will make the task that they were scheduled from sticky. To obtain the behavior of `Threads.@spawn` set the sticky bit manually to `false`.

Examples

```
julia> a5() = sum(i for i in 1:1000);

julia> b = Task(a5);

julia> istaskstarted(b)
false

julia> schedule(b);

julia> yield();

julia> istaskstarted(b)
```

```
true

julia> istaskdone(b)
true
```

source

48.2 Synchronization

`Base.errormonitor` – Function.

```
errormonitor(t::Task)
```

Print an error log to `stderr` if task `t` fails.

Examples

```
julia> Base._wait(errormonitor(Threads.@spawn error("task failed")))
Unhandled Task ERROR: task failed
Stacktrace:
[...]
```

source

`Base.@sync` – Macro.

```
@sync
```

Wait until all lexically-enclosed uses of `@async`, `@spawn`, `@spawnat` and `@distributed` are complete. All exceptions thrown by enclosed async operations are collected and thrown as a `CompositeException`.

Examples

```
julia> Threads.nthreads()
4

julia> @sync begin
           Threads.@spawn println("Thread-id $(Threads.threadid()), task 1")
           Threads.@spawn println("Thread-id $(Threads.threadid()), task 2")
       end;
Thread-id 3, task 1
Thread-id 1, task 2
```

source

`Base.wait` – Function.

Special note for `Threads.Condition`:

The caller must be holding the `lock` that owns a `Threads.Condition` before calling this method. The calling task will be blocked until some other task wakes it, usually by calling `notify` on the same `Threads.Condition` object. The lock will be atomically released when blocking (even if it was locked recursively), and will be reacquired before returning.

source

```
wait(r::Future)
```

Wait for a value to become available for the specified `Future`.

source

```
wait(r::RemoteChannel, args...)
```

Wait for a value to become available on the specified `RemoteChannel`.

source

```
wait([x])
```

Block the current task until some event occurs, depending on the type of the argument:

- `Channel`: Wait for a value to be appended to the channel.
- `Condition`: Wait for `notify` on a condition and return the `val` parameter passed to `notify`. Waiting on a condition additionally allows passing `first=true` which results in the waiter being put *first* in line to wake up on `notify` instead of the usual first-in-first-out behavior.
- `Process`: Wait for a process or process chain to exit. The `exitcode` field of a process can be used to determine success or failure.
- `Task`: Wait for a `Task` to finish. If the task fails with an exception, a `TaskFailedException` (which wraps the failed task) is thrown.
- `RawFD`: Wait for changes on a file descriptor (see the `FileWatching` package).

If no argument is passed, the task blocks for an undefined period. A task can only be restarted by an explicit call to `schedule` or `yieldto`.

Often `wait` is called within a `while` loop to ensure a waited-for condition is met before proceeding.

source

```
wait(c::Channel)
```

Blocks until the Channel `isready`.

```
julia> c = Channel(1);

julia> isready(c)
false
```

```
julia> task = Task(() -> wait(c));

julia> schedule(task);

julia> istaskdone(task)  # task is blocked because channel is not ready
false

julia> put!(c, 1);

julia> istaskdone(task)  # task is now unblocked
true
```

source

Base.fetch – Method.

```
fetch(t::Task)
```

Wait for a Task to finish, then return its result value. If the task fails with an exception, a TaskFailedException (which wraps the failed task) is thrown.

source

Base.fetch – Method.

```
fetch(x::Any)
```

Return x.

source

Base.timedwait – Function.

```
timedwait(testcb, timeout::Real; pollint::Real=0.1)
```

Waits until `testcb()` returns `true` or `timeout` seconds have passed, whichever is earlier. The test function is polled every `pollint` seconds. The minimum value for `pollint` is 0.001 seconds, that is, 1 millisecond.

Return `:ok` or `:timed_out`.

source

Base.Condition – Type.

```
Condition()
```

Create an edge-triggered event source that tasks can wait for. Tasks that call `wait` on a `Condition` are suspended and queued. Tasks are woken up when `notify` is later called on the `Condition`. Edge triggering

means that only tasks waiting at the time `notify` is called can be woken up. For level-triggered notifications, you must keep extra state to keep track of whether a notification has happened. The `Channel` and `Threads.Event` types do this, and can be used for level-triggered events.

This object is NOT thread-safe. See `Threads.Condition` for a thread-safe version.

source

`Base.Threads.Condition` – Type.

```
Threads.Condition([lock])
```

A thread-safe version of `Base.Condition`.

To call `wait` or `notify` on a `Threads.Condition`, you must first call `lock` on it. When `wait` is called, the lock is atomically released during blocking, and will be reacquired before `wait` returns. Therefore idiomatic use of a `Threads.Condition` `c` looks like the following:

```
lock(c)
try
    while !thing_we_are_waiting_for
        wait(c)
    end
finally
    unlock(c)
end
```

Julia 1.2

This functionality requires at least Julia 1.2.

source

`Base.Event` – Type.

```
Event([autoreset=false])
```

Create a level-triggered event source. Tasks that call `wait` on an `Event` are suspended and queued until `notify` is called on the `Event`. After `notify` is called, the `Event` remains in a signaled state and tasks will no longer block when waiting for it, until `reset` is called.

If `autoreset` is true, at most one task will be released from `wait` for each call to `notify`.

This provides an acquire & release memory ordering on notify/wait.

Julia 1.1

This functionality requires at least Julia 1.1.

Julia 1.8

The `autoreset` functionality and memory ordering guarantee requires at least Julia 1.8.

source

Base.notify – Function.

```
notify(condition, val=nothing; all=true, error=false)
```

Wake up tasks waiting for a condition, passing them `val`. If `all` is `true` (the default), all waiting tasks are woken, otherwise only one is. If `error` is `true`, the passed value is raised as an exception in the woken tasks.

Return the count of tasks woken up. Return 0 if no tasks are waiting on `condition`.

source

Base.reset – Method.

```
reset(::Event)
```

Reset an `Event` back into an un-set state. Then any future calls to `wait` will block until `notify` is called again.

source

Base.Semaphore – Type.

```
Semaphore(sem_size)
```

Create a counting semaphore that allows at most `sem_size` acquires to be in use at any time. Each acquire must be matched with a release.

This provides a acquire & release memory ordering on acquire/release calls.

source

Base.acquire – Function.

```
acquire(s::Semaphore)
```

Wait for one of the `sem_size` permits to be available, blocking until one can be acquired.

source

```
acquire(f, s::Semaphore)
```

Execute `f` after acquiring from Semaphore `s`, and `release` on completion or error.

For example, a do-block form that ensures only 2 calls of `foo` will be active at the same time:

```
s = Base.Semaphore(2)
@sync for _ in 1:100
    Threads.@spawn begin
        Base.acquire(s) do
            foo()
        end
    end
end
```

Julia 1.8

This method requires at least Julia 1.8.

source

`Base.release` – Function.

```
release(s::Semaphore)
```

Return one permit to the pool, possibly allowing another task to acquire it and resume execution.

source

`Base.AbstractLock` – Type.

```
AbstractLock
```

Abstract supertype describing types that implement the synchronization primitives: `lock`, `trylock`, `unlock`, and `islocked`.

source

`Base.lock` – Function.

```
lock(lock)
```

Acquire the `lock` when it becomes available. If the lock is already locked by a different task/thread, wait for it to become available.

Each `lock` must be matched by an `unlock`.

source

```
lock(f::Function, lock)
```

Acquire the `lock`, execute `f` with the `lock` held, and release the `lock` when `f` returns. If the lock is already locked by a different task/thread, wait for it to become available.

When this function returns, the `lock` has been released, so the caller should not attempt to `unlock` it.

> **Julia 1.7**
>
> Using a `Channel` as the second argument requires Julia 1.7 or later.

source

`Base.unlock` – Function.

```
unlock(lock)
```

Releases ownership of the `lock`.

If this is a recursive lock which has been acquired before, decrement an internal counter and return immediately.

source

`Base.trylock` – Function.

```
trylock(lock) -> Success (Boolean)
```

Acquire the lock if it is available, and return `true` if successful. If the lock is already locked by a different task/thread, return `false`.

Each successful `trylock` must be matched by an `unlock`.

Function `trylock` combined with `islocked` can be used for writing the test-and-test-and-set or exponential backoff algorithms *if it is supported by the `typeof(lock)`* (read its documentation).

source

`Base.islocked` – Function.

```
islocked(lock) -> Status (Boolean)
```

Check whether the `lock` is held by any task/thread. This function alone should not be used for synchronization. However, `islocked` combined with `trylock` can be used for writing the test-and-test-and-set or exponential backoff algorithms *if it is supported by the `typeof(lock)`* (read its documentation).

Extended help

For example, an exponential backoff can be implemented as follows if the `lock` implementation satisfied the properties documented below.

```
nspins = 0
while true
    while islocked(lock)
        GC.safepoint()
        nspins += 1
        nspins > LIMIT && error("timeout")
    end
    trylock(lock) && break
    backoff()
end
```

Implementation

A lock implementation is advised to define `islocked` with the following properties and note it in its docstring.

- `islocked(lock)` is data-race-free.
- If `islocked(lock)` returns `false`, an immediate invocation of `trylock(lock)` must succeed (returns `true`) if there is no interference from other tasks.

source

`Base.ReentrantLock` – Type.

```
ReentrantLock()
```

Creates a re-entrant lock for synchronizing `Task`s. The same task can acquire the lock as many times as required (this is what the "Reentrant" part of the name means). Each `lock` must be matched with an `unlock`.

Calling 'lock' will also inhibit running of finalizers on that thread until the corresponding 'unlock'. Use of the standard lock pattern illustrated below should naturally be supported, but beware of inverting the try/lock order or missing the try block entirely (e.g. attempting to return with the lock still held):

This provides a acquire/release memory ordering on lock/unlock calls.

```
lock(l)
try
    <atomic work>
finally
    unlock(l)
end
```

If `!islocked(lck::ReentrantLock)` holds, `trylock(lck)` succeeds unless there are other tasks attempting to hold the lock "at the same time."

source

48.3 Channels

`Base.AbstractChannel` – Type.

```
AbstractChannel{T}
```

Representation of a channel passing objects of type T.

source

Base.Channel – Type.

```
Channel{T=Any}(size::Int=0)
```

Constructs a Channel with an internal buffer that can hold a maximum of size objects of type T. put! calls on a full channel block until an object is removed with take!.

Channel(0) constructs an unbuffered channel. put! blocks until a matching take! is called. And vice-versa.

Other constructors:

- Channel(): default constructor, equivalent to Channel{Any}(0)
- Channel(Inf): equivalent to Channel{Any}(typemax(Int))
- Channel(sz): equivalent to Channel{Any}(sz)

Julia 1.3

The default constructor Channel() and default size=0 were added in Julia 1.3.

source

Base.Channel – Method.

```
Channel{T=Any}(func::Function, size=0; taskref=nothing, spawn=false, threadpool=nothing)
```

Create a new task from func, bind it to a new channel of type T and size size, and schedule the task, all in a single call. The channel is automatically closed when the task terminates.

func must accept the bound channel as its only argument.

If you need a reference to the created task, pass a Ref{Task} object via the keyword argument taskref.

If spawn=true, the Task created for func may be scheduled on another thread in parallel, equivalent to creating a task via Threads.@spawn.

If spawn=true and the threadpool argument is not set, it defaults to :default.

If the threadpool argument is set (to :default or :interactive), this implies that spawn=true and the new Task is spawned to the specified threadpool.

Return a Channel.

Examples

```
julia> chnl = Channel() do ch
           foreach(i -> put!(ch, i), 1:4)
       end;

julia> typeof(chnl)
Channel{Any}

julia> for i in chnl
           @show i
       end;
i = 1
i = 2
i = 3
i = 4
```

Referencing the created task:

```
julia> taskref = Ref{Task}();

julia> chnl = Channel(taskref=taskref) do ch
           println(take!(ch))
       end;

julia> istaskdone(taskref[])
false

julia> put!(chnl, "Hello");
Hello

julia> istaskdone(taskref[])
true
```

Julia 1.3

The `spawn=` parameter was added in Julia 1.3. This constructor was added in Julia 1.3. In earlier versions of Julia, Channel used keyword arguments to set `size` and `T`, but those constructors are deprecated.

Julia 1.9

The `threadpool=` argument was added in Julia 1.9.

```
julia> chnl = Channel{Char}(1, spawn=true) do ch
           for c in "hello world"
               put!(ch, c)
           end
       end
Channel{Char}(1) (2 items available)

julia> String(collect(chnl))
"hello world"
```

source

Base.put! – Method.

```
put!(c::Channel, v)
```

Append an item v to the channel c. Blocks if the channel is full.

For unbuffered channels, blocks until a take! is performed by a different task.

> **Julia 1.1**
>
> v now gets converted to the channel's type with convert as put! is called.

source

Base.take! – Method.

```
take!(c::Channel)
```

Removes and returns a value from a Channel in order. Blocks until data is available. For unbuffered channels, blocks until a put! is performed by a different task.

Examples

Buffered channel:

```
julia> c = Channel(1);

julia> put!(c, 1);

julia> take!(c)
1
```

Unbuffered channel:

```
julia> c = Channel(0);

julia> task = Task(() -> put!(c, 1));

julia> schedule(task);

julia> take!(c)
1
```

source

Base.isready – Method.

```
isready(c::Channel)
```

Determines whether a `Channel` has a value stored in it. Returns immediately, does not block.

For unbuffered channels returns `true` if there are tasks waiting on a `put!`.

Examples

Buffered channel:

```
julia> c = Channel(1);

julia> isready(c)
false

julia> put!(c, 1);

julia> isready(c)
true
```

Unbuffered channel:

```
julia> c = Channel();

julia> isready(c)  # no tasks waiting to put!
false

julia> task = Task(() -> put!(c, 1));

julia> schedule(task);  # schedule a put! task

julia> isready(c)
true
```

source

`Base.fetch` – Method.

```
fetch(c::Channel)
```

Waits for and returns (without removing) the first available item from the `Channel`. Note: `fetch` is unsupported on an unbuffered (0-size) `Channel`.

Examples

Buffered channel:

```
julia> c = Channel(3) do ch
           foreach(i -> put!(ch, i), 1:3)
       end;
```

```
julia> fetch(c)
1

julia> collect(c)  # item is not removed
3-element Vector{Any}:
 1
 2
 3
```

source

Base.close – Method.

```
close(c::Channel[, excp::Exception])
```

Close a channel. An exception (optionally given by excp), is thrown by:

- put! on a closed channel.
- take! and fetch on an empty, closed channel.

source

Base.bind – Method.

```
bind(chnl::Channel, task::Task)
```

Associate the lifetime of chnl with a task. Channel chnl is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on chnl.

The chnl object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed Channel objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

Examples

```
julia> c = Channel(0);

julia> task = @async foreach(i->put!(c, i), 1:4);

julia> bind(c,task);

julia> for i in c
           @show i
       end;
i = 1
i = 2
i = 3
i = 4

julia> isopen(c)
false
```

```
julia> c = Channel(0);

julia> task = @async (put!(c, 1); error("foo"));

julia> bind(c, task);

julia> take!(c)
1

julia> put!(c, 1);
ERROR: TaskFailedException
Stacktrace:
[...]
    nested task error: foo
[...]
```

source

48.4 Low-level synchronization using `schedule` and `wait`

The easiest correct use of `schedule` is on a Task that is not started (scheduled) yet. However, it is possible to use `schedule` and `wait` as a very low-level building block for constructing synchronization interfaces. A crucial pre-condition of calling `schedule(task)` is that the caller must "own" the `task`; i.e., it must know that the call to `wait` in the given `task` is happening at the locations known to the code calling `schedule(task)`. One strategy for ensuring such pre-condition is to use atomics, as demonstrated in the following example:

```
@enum OWEState begin
    OWE_EMPTY
    OWE_WAITING
    OWE_NOTIFYING
end

mutable struct OneWayEvent
    @atomic state::OWEState
    task::Task
    OneWayEvent() = new(OWE_EMPTY)
end

function Base.notify(ev::OneWayEvent)
    state = @atomic ev.state
    while state !== OWE_NOTIFYING
        # Spin until we successfully update the state to OWE_NOTIFYING:
        state, ok = @atomicreplace(ev.state, state => OWE_NOTIFYING)
        if ok
            if state == OWE_WAITING
                # OWE_WAITING -> OWE_NOTIFYING transition means that the waiter task is
                # already waiting or about to call `wait`. The notifier task must wake up
                # the waiter task.
                schedule(ev.task)
            else
                @assert state == OWE_EMPTY
                # Since we are assuming that there is only one notifier task (for
                # simplicity), we know that the other possible case here is OWE_EMPTY.
```

```
                # We do not need to do anything because we know that the waiter task has
                # not called `wait(ev::OneWayEvent)` yet.
            end
            break
        end
    end
    return
end

function Base.wait(ev::OneWayEvent)
    ev.task = current_task()
    state, ok = @atomicreplace(ev.state, OWE_EMPTY => OWE_WAITING)
    if ok
        # OWE_EMPTY -> OWE_WAITING transition means that the notifier task is guaranteed to
        # invoke OWE_WAITING -> OWE_NOTIFYING transition.  The waiter task must call
        # `wait()` immediately.  In particular, it MUST NOT invoke any function that may
        # yield to the scheduler at this point in code.
        wait()
    else
        @assert state == OWE_NOTIFYING
        # Otherwise, the `state` must have already been moved to OWE_NOTIFYING by the
        # notifier task.
    end
    return
end

ev = OneWayEvent()
@sync begin
    @async begin
        wait(ev)
        println("done")
    end
    println("notifying...")
    notify(ev)
end

# output
notifying...
done
```

`OneWayEvent` lets one task to `wait` for another task's `notify`. It is a limited communication interface since `wait` can only be used once from a single task (note the non-atomic assignment of `ev.task`)

In this example, `notify(ev::OneWayEvent)` is allowed to call `schedule(ev.task)` if and only if *it* modifies the state from `OWE_WAITING` to `OWE_NOTIFYING`. This lets us know that the task executing `wait(ev::OneWayEvent)` is now in the `ok` branch and that there cannot be other tasks that tries to `schedule(ev.task)` since their `@atomicreplace(ev.state, state => OWE_NOTIFYING)` will fail.

Chapter 49

Multi-Threading

`Base.Threads.@threads` – Macro.

```
Threads.@threads [schedule] for ... end
```

A macro to execute a `for` loop in parallel. The iteration space is distributed to coarse-grained tasks. This policy can be specified by the `schedule` argument. The execution of the loop waits for the evaluation of all iterations.

See also: `@spawn` and `pmap` in `Distributed`.

Extended help

Semantics

Unless stronger guarantees are specified by the scheduling option, the loop executed by `@threads` macro have the following semantics.

The `@threads` macro executes the loop body in an unspecified order and potentially concurrently. It does not specify the exact assignments of the tasks and the worker threads. The assignments can be different for each execution. The loop body code (including any code transitively called from it) must not make any assumptions about the distribution of iterations to tasks or the worker thread in which they are executed. The loop body for each iteration must be able to make forward progress independent of other iterations and be free from data races. As such, invalid synchronizations across iterations may deadlock while unsynchronized memory accesses may result in undefined behavior.

For example, the above conditions imply that:

- A lock taken in an iteration *must* be released within the same iteration.
- Communicating between iterations using blocking primitives like `Channel`s is incorrect.
- Write only to locations not shared across iterations (unless a lock or atomic operation is used).
- Unless the `:static` schedule is used, the value of `threadid()` may change even within a single iteration. See `Task Migration`.

Schedulers

Without the scheduler argument, the exact scheduling is unspecified and varies across Julia releases. Currently, `:dynamic` is used when the scheduler is not specified.

Julia 1.5

The `schedule` argument is available as of Julia 1.5.

`:dynamic` (default)

`:dynamic` scheduler executes iterations dynamically to available worker threads. Current implementation assumes that the workload for each iteration is uniform. However, this assumption may be removed in the future.

This scheduling option is merely a hint to the underlying execution mechanism. However, a few properties can be expected. The number of `Task`s used by `:dynamic` scheduler is bounded by a small constant multiple of the number of available worker threads (`Threads.threadpoolsize()`). Each task processes contiguous regions of the iteration space. Thus, `@threads :dynamic for x in xs; f(x); end` is typically more efficient than `@sync for x in xs; @spawn f(x); end` if `length(xs)` is significantly larger than the number of the worker threads and the run-time of `f(x)` is relatively smaller than the cost of spawning and synchronizing a task (typically less than 10 microseconds).

Julia 1.8

The `:dynamic` option for the `schedule` argument is available and the default as of Julia 1.8.

`:static`

`:static` scheduler creates one task per thread and divides the iterations equally among them, assigning each task specifically to each thread. In particular, the value of `threadid()` is guaranteed to be constant within one iteration. Specifying `:static` is an error if used from inside another `@threads` loop or from a thread other than 1.

Note

`:static` scheduling exists for supporting transition of code written before Julia 1.3. In newly written library functions, `:static` scheduling is discouraged because the functions using this option cannot be called from arbitrary worker threads.

Example

To illustrate of the different scheduling strategies, consider the following function `busywait` containing a non-yielding timed loop that runs for a given number of seconds.

```
julia> function busywait(seconds)
            tstart = time_ns()
            while (time_ns() - tstart) / 1e9 < seconds
            end
        end

julia> @time begin
            Threads.@spawn busywait(5)
            Threads.@threads :static for i in 1:Threads.threadpoolsize()
                busywait(1)
            end
        end
6.003001 seconds (16.33 k allocations: 899.255 KiB, 0.25% compilation time)
```

```
julia> @time begin
           Threads.@spawn busywait(5)
           Threads.@threads :dynamic for i in 1:Threads.threadpoolsize()
               busywait(1)
           end
       end
2.012056 seconds (16.05 k allocations: 883.919 KiB, 0.66% compilation time)
```

The `:dynamic` example takes 2 seconds since one of the non-occupied threads is able to run two of the 1-second iterations to complete the for loop.

source

`Base.Threads.foreach` – Function.

```
Threads.foreach(f, channel::Channel;
                schedule::Threads.AbstractSchedule=Threads.FairSchedule(),
                ntasks=Threads.threadpoolsize())
```

Similar to `foreach(f, channel)`, but iteration over `channel` and calls to `f` are split across `ntasks` tasks spawned by `Threads.@spawn`. This function will wait for all internally spawned tasks to complete before returning.

If `schedule isa FairSchedule`, `Threads.foreach` will attempt to spawn tasks in a manner that enables Julia's scheduler to more freely load-balance work items across threads. This approach generally has higher per-item overhead, but may perform better than `StaticSchedule` in concurrence with other multithreaded workloads.

If `schedule isa StaticSchedule`, `Threads.foreach` will spawn tasks in a manner that incurs lower per-item overhead than `FairSchedule`, but is less amenable to load-balancing. This approach thus may be more suitable for fine-grained, uniform workloads, but may perform worse than `FairSchedule` in concurrence with other multithreaded workloads.

Examples

```
julia> n = 20

julia> c = Channel{Int}(ch -> foreach(i -> put!(ch, i), 1:n), 1)

julia> d = Channel{Int}(n) do ch
           f = i -> put!(ch, i^2)
           Threads.foreach(f, c)
       end

julia> collect(d)
collect(d) = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361,
↪  400]
```

Julia 1.6

This function requires Julia 1.6 or later.

source

`Base.Threads.@spawn` – Macro.

```
Threads.@spawn [:default|:interactive] expr
```

Create a `Task` and `schedule` it to run on any available thread in the specified threadpool (`:default` if unspecified). The task is allocated to a thread once one becomes available. To wait for the task to finish, call `wait` on the result of this macro, or call `fetch` to wait and then obtain its return value.

Values can be interpolated into `@spawn` via `$`, which copies the value directly into the constructed underlying closure. This allows you to insert the *value* of a variable, isolating the asynchronous code from changes to the variable's value in the current task.

Note

The thread that the task runs on may change if the task yields, therefore `threadid()` should not be treated as constant for a task. See `Task Migration`, and the broader multi-threading manual for further important caveats. See also the chapter on threadpools.

Julia 1.3

This macro is available as of Julia 1.3.

Julia 1.4

Interpolating values via `$` is available as of Julia 1.4.

Julia 1.9

A threadpool may be specified as of Julia 1.9.

Examples

```
julia> t() = println("Hello from ", Threads.threadid());

julia> tasks = fetch.([Threads.@spawn t() for i in 1:4]);
Hello from 1
Hello from 1
Hello from 3
Hello from 4
```

source

`Base.Threads.threadid` – Function.

```
Threads.threadid() -> Int
```

Get the ID number of the current thread of execution. The master thread has ID 1.

Examples

```
julia> Threads.threadid()
1

julia> Threads.@threads for i in 1:4
          println(Threads.threadid())
       end
4
2
5
4
```

> **Note**
>
> The thread that a task runs on may change if the task yields, which is known as `Task Migration`. For this reason in most cases it is not safe to use `threadid()` to index into, say, a vector of buffer or stateful objects.

source

`Base.Threads.maxthreadid` – Function.

```
Threads.maxthreadid() -> Int
```

Get a lower bound on the number of threads (across all thread pools) available to the Julia process, with atomic-acquire semantics. The result will always be greater than or equal to `threadid()` as well as `threadid(task)` for any task you were able to observe before calling `maxthreadid`.

source

`Base.Threads.nthreads` – Function.

```
Threads.nthreads(:default | :interactive) -> Int
```

Get the current number of threads within the specified thread pool. The threads in `:interactive` have id numbers `1:nthreads(:interactive)`, and the threads in `:default` have id numbers in `nthreads(:interactive) .+ (1:nthreads(:default))`.

See also `BLAS.get_num_threads` and `BLAS.set_num_threads` in the `LinearAlgebra` standard library, and `nprocs()` in the `Distributed` standard library and `Threads.maxthreadid()`.

source

`Base.Threads.threadpool` – Function.

```
Threads.threadpool(tid = threadid()) -> Symbol
```

Returns the specified thread's threadpool; either `:default`, `:interactive`, or `:foreign`.

source

`Base.Threads.nthreadpools` – Function.

```
Threads.nthreadpools() -> Int
```

Returns the number of threadpools currently configured.

source

`Base.Threads.threadpoolsize` – Function.

```
Threads.threadpoolsize(pool::Symbol = :default) -> Int
```

Get the number of threads available to the default thread pool (or to the specified thread pool).

See also: `BLAS.get_num_threads` and `BLAS.set_num_threads` in the `LinearAlgebra` standard library, and `nprocs()` in the `Distributed` standard library.

source

`Base.Threads.ngcthreads` – Function.

```
Threads.ngcthreads() -> Int
```

Returns the number of GC threads currently configured. This includes both mark threads and concurrent sweep threads.

source

See also Multi-Threading.

49.1 原子操作

`atomic` – Keyword.

Unsafe pointer operations are compatible with loading and storing pointers declared with `_Atomic` and `std::atomic` type in C11 and C++23 respectively. An error may be thrown if there is not support for atomically loading the Julia type T.

See also: `unsafe_load`, `unsafe_modify!`, `unsafe_replace!`, `unsafe_store!`, `unsafe_swap!`

source

`Base.@atomic` – Macro.

```
@atomic var
@atomic order ex
```

Mark `var` or ex as being performed atomically, if ex is a supported expression. If no `order` is specified it defaults to :sequentially_consistent.

```
@atomic a.b.x = new
@atomic a.b.x += addend
@atomic :release a.b.x = new
@atomic :acquire_release a.b.x += addend
```

Perform the store operation expressed on the right atomically and return the new value.

With =, this operation translates to a `setproperty!(a.b, :x, new)` call. With any operator also, this operation translates to a `modifyproperty!(a.b, :x, +, addend)[2]` call.

```
@atomic a.b.x max arg2
@atomic a.b.x + arg2
@atomic max(a.b.x, arg2)
@atomic :acquire_release max(a.b.x, arg2)
@atomic :acquire_release a.b.x + arg2
@atomic :acquire_release a.b.x max arg2
```

Perform the binary operation expressed on the right atomically. Store the result into the field in the first argument and return the values `(old, new)`.

This operation translates to a `modifyproperty!(a.b, :x, func, arg2)` call.

See Per-field atomics section in the manual for more details.

Examples

```
julia> mutable struct Atomic{T}; @atomic x::T; end

julia> a = Atomic(1)
Atomic{Int64}(1)

julia> @atomic a.x # fetch field x of a, with sequential consistency
1

julia> @atomic :sequentially_consistent a.x = 2 # set field x of a, with sequential consistency
2

julia> @atomic a.x += 1 # increment field x of a, with sequential consistency
3

julia> @atomic a.x + 1 # increment field x of a, with sequential consistency
3 => 4

julia> @atomic a.x # fetch field x of a, with sequential consistency
4

julia> @atomic max(a.x, 10) # change field x of a to the max value, with sequential consistency
4 => 10

julia> @atomic a.x max 5 # again change field x of a to the max value, with sequential
↪  consistency
10 => 10
```

Julia 1.7

This functionality requires at least Julia 1.7.

source

Base.@atomicswap – Macro.

```
@atomicswap a.b.x = new
@atomicswap :sequentially_consistent a.b.x = new
```

Stores new into `a.b.x` and returns the old value of `a.b.x`.

This operation translates to a `swapproperty!(a.b, :x, new)` call.

See Per-field atomics section in the manual for more details.

Examples

```
julia> mutable struct Atomic{T}; @atomic x::T; end

julia> a = Atomic(1)
Atomic{Int64}(1)

julia> @atomicswap a.x = 2+2 # replace field x of a with 4, with sequential consistency
1

julia> @atomic a.x # fetch field x of a, with sequential consistency
4
```

Julia 1.7

This functionality requires at least Julia 1.7.

source

Base.@atomicreplace – Macro.

```
@atomicreplace a.b.x expected => desired
@atomicreplace :sequentially_consistent a.b.x expected => desired
@atomicreplace :sequentially_consistent :monotonic a.b.x expected => desired
```

Perform the conditional replacement expressed by the pair atomically, returning the values `(old, success::Bool)`. Where `success` indicates whether the replacement was completed.

This operation translates to a `replaceproperty!(a.b, :x, expected, desired)` call.

See Per-field atomics section in the manual for more details.

Examples

```
julia> mutable struct Atomic{T}; @atomic x::T; end

julia> a = Atomic(1)
Atomic{Int64}(1)

julia> @atomicreplace a.x 1 => 2 # replace field x of a with 2 if it was 1, with sequential
↪  consistency
(old = 1, success = true)

julia> @atomic a.x # fetch field x of a, with sequential consistency
2

julia> @atomicreplace a.x 1 => 2 # replace field x of a with 2 if it was 1, with sequential
↪  consistency
(old = 2, success = false)

julia> xchg = 2 => 0; # replace field x of a with 0 if it was 2, with sequential consistency

julia> @atomicreplace a.x xchg
(old = 2, success = true)

julia> @atomic a.x # fetch field x of a, with sequential consistency
0
```

Julia 1.7

This functionality requires at least Julia 1.7.

source

Note

The following APIs are fairly primitive, and will likely be exposed through an `unsafe_*`-like wrapper.

```
Core.Intrinsics.atomic_pointerref(pointer::Ptr{T}, order::Symbol) --> T
Core.Intrinsics.atomic_pointerset(pointer::Ptr{T}, new::T, order::Symbol) --> pointer
Core.Intrinsics.atomic_pointerswap(pointer::Ptr{T}, new::T, order::Symbol) --> old
Core.Intrinsics.atomic_pointermodify(pointer::Ptr{T}, function::(old::T,arg::S)->T, arg::S,
↪  order::Symbol) --> old
Core.Intrinsics.atomic_pointerreplace(pointer::Ptr{T}, expected::Any, new::T,
↪  success_order::Symbol, failure_order::Symbol) --> (old, cmp)
```

Warning

The following APIs are deprecated, though support for them is likely to remain for several releases.

`Base.Threads.Atomic` – Type.

```
Threads.Atomic{T}
```

Holds a reference to an object of type T, ensuring that it is only accessed atomically, i.e. in a thread-safe manner.

Only certain "simple" types can be used atomically, namely the primitive boolean, integer, and float-point types. These are `Bool`, `Int8`...`Int128`, `UInt8`...`UInt128`, and `Float16`...`Float64`.

New atomic objects can be created from a non-atomic values; if none is specified, the atomic object is initialized with zero.

Atomic objects can be accessed using the `[]` notation:

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> x[] = 1
1

julia> x[]
1
```

Atomic operations use an `atomic_` prefix, such as `atomic_add!`, `atomic_xchg!`, etc.

source

`Base.Threads.atomic_cas!` – Function.

```
Threads.atomic_cas!(x::Atomic{T}, cmp::T, newval::T) where T
```

Atomically compare-and-set x

Atomically compares the value in x with `cmp`. If equal, write `newval` to x. Otherwise, leaves x unmodified. Returns the old value in x. By comparing the returned value to `cmp` (via ===) one knows whether x was modified and now holds the new value `newval`.

For further details, see LLVM's `cmpxchg` instruction.

This function can be used to implement transactional semantics. Before the transaction, one records the value in x. After the transaction, the new value is stored only if x has not been modified in the mean time.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 4, 2);

julia> x
Base.Threads.Atomic{Int64}(3)
```

```
julia> Threads.atomic_cas!(x, 3, 2);

julia> x
Base.Threads.Atomic{Int64}(2)
```

source

`Base.Threads.atomic_xchg!` – Function.

```
Threads.atomic_xchg!(x::Atomic{T}, newval::T) where T
```

Atomically exchange the value in `x`

Atomically exchanges the value in `x` with `newval`. Returns the **old** value.

For further details, see LLVM's `atomicrmw xchg` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_xchg!(x, 2)
3

julia> x[]
2
```

source

`Base.Threads.atomic_add!` – Function.

```
Threads.atomic_add!(x::Atomic{T}, val::T) where T <: ArithmeticTypes
```

Atomically add `val` to `x`

Performs `x[] += val` atomically. Returns the **old** value. Not defined for `Atomic{Bool}`.

For further details, see LLVM's `atomicrmw add` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_add!(x, 2)
3

julia> x[]
5
```

source

`Base.Threads.atomic_sub!` – Function.

```
Threads.atomic_sub!(x::Atomic{T}, val::T) where T <: ArithmeticTypes
```

Atomically subtract `val` from `x`

Performs `x[] -= val` atomically. Returns the **old** value. Not defined for `Atomic{Bool}`.

For further details, see LLVM's `atomicrmw sub` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_sub!(x, 2)
3

julia> x[]
1
```

source

`Base.Threads.atomic_and!` – Function.

```
Threads.atomic_and!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-and `x` with `val`

Performs `x[] &= val` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw and` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_and!(x, 2)
3

julia> x[]
2
```

source

`Base.Threads.atomic_nand!` – Function.

```
Threads.atomic_nand!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-nand (not-and) x with `val`

Performs `x[] = ~(x[] & val)` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw nand` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_nand!(x, 2)
3

julia> x[]
-3
```

source

`Base.Threads.atomic_or!` – Function.

```
Threads.atomic_or!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-or x with `val`

Performs `x[] |= val` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw or` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_or!(x, 7)
5

julia> x[]
7
```

source

`Base.Threads.atomic_xor!` – Function.

```
Threads.atomic_xor!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-xor (exclusive-or) x with `val`

Performs `x[] $= val` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw xor` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_xor!(x, 7)
5

julia> x[]
2
```

source

Base.Threads.atomic_max! – Function.

```
Threads.atomic_max!(x::Atomic{T}, val::T) where T
```

Atomically store the maximum of x and `val` in x

Performs `x[] = max(x[], val)` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw max` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_max!(x, 7)
5

julia> x[]
7
```

source

Base.Threads.atomic_min! – Function.

```
Threads.atomic_min!(x::Atomic{T}, val::T) where T
```

Atomically store the minimum of x and `val` in x

Performs `x[] = min(x[], val)` atomically. Returns the **old** value.

For further details, see LLVM's `atomicrmw min` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(7)
Base.Threads.Atomic{Int64}(7)

julia> Threads.atomic_min!(x, 5)
7

julia> x[]
5
```

source

`Base.Threads.atomic_fence` – Function.

```
Threads.atomic_fence()
```

Insert a sequential-consistency memory fence

Inserts a memory fence with sequentially-consistent ordering semantics. There are algorithms where this is needed, i.e. where an acquire/release ordering is insufficient.

This is likely a very expensive operation. Given that all other atomic operations in Julia already have acquire/release semantics, explicit fences should not be necessary in most cases.

For further details, see LLVM's `fence` instruction.

source

49.2 ccall using a libuv threadpool (Experimental)

`Base.@threadcall` – Macro.

```
@threadcall((cfunc, clib), rettype, (argtypes...), argvals...)
```

The `@threadcall` macro is called in the same way as `ccall` but does the work in a different thread. This is useful when you want to call a blocking C function without causing the current `julia` thread to become blocked. Concurrency is limited by size of the libuv thread pool, which defaults to 4 threads but can be increased by setting the `UV_THREADPOOL_SIZE` environment variable and restarting the `julia` process.

Note that the called function should never call back into Julia.

source

49.3 Low-level synchronization primitives

These building blocks are used to create the regular synchronization objects.

`Base.Threads.SpinLock` – Type.

```
SpinLock()
```

Create a non-reentrant, test-and-test-and-set spin lock. Recursive use will result in a deadlock. This kind of lock should only be used around code that takes little time to execute and does not block (e.g. perform I/O). In general, `ReentrantLock` should be used instead.

Each `lock` must be matched with an `unlock`. If `!islocked(lck::SpinLock)` holds, `trylock(lck)` succeeds unless there are other tasks attempting to hold the lock "at the same time."

Test-and-test-and-set spin locks are quickest up to about 30ish contending threads. If you have more contention than that, different synchronization approaches should be considered.

source

Chapter 50

常量

`Core.nothing` – Constant.

```
nothing
```

The singleton instance of type `Nothing`, used by convention when there is no value to return (as in a C `void` function) or when a variable or field holds no value.

See also: `isnothing`, `something`, `missing`.

source

`Base.PROGRAM_FILE` – Constant.

```
PROGRAM_FILE
```

A string containing the script name passed to Julia from the command line. Note that the script name remains unchanged from within included files. Alternatively see `@__FILE__`.

source

`Base.ARGS` – Constant.

```
ARGS
```

An array of the command line arguments passed to Julia, as strings.

source

`Base.C_NULL` – Constant.

```
C_NULL
```

The C null pointer constant, sometimes used when calling external code.

source

Base.VERSION – Constant.

```
VERSION
```

A VersionNumber object describing which version of Julia is in use. See also Version Number Literals.

source

Base.DEPOT_PATH – Constant.

```
DEPOT_PATH
```

A stack of "depot" locations where the package manager, as well as Julia's code loading mechanisms, look for package registries, installed packages, named environments, repo clones, cached compiled package images, and configuration files. By default it includes:

1. ~/.julia where ~ is the user home as appropriate on the system;
2. an architecture-specific shared system directory, e.g. /usr/local/share/julia;
3. an architecture-independent shared system directory, e.g. /usr/share/julia.

So DEPOT_PATH might be:

```
[joinpath(homedir(), ".julia"), "/usr/local/share/julia", "/usr/share/julia"]
```

The first entry is the "user depot" and should be writable by and owned by the current user. The user depot is where: registries are cloned, new package versions are installed, named environments are created and updated, package repos are cloned, newly compiled package image files are saved, log files are written, development packages are checked out by default, and global configuration data is saved. Later entries in the depot path are treated as read-only and are appropriate for registries, packages, etc. installed and managed by system administrators.

DEPOT_PATH is populated based on the JULIA_DEPOT_PATH environment variable if set.

DEPOT_PATH contents

Each entry in DEPOT_PATH is a path to a directory which contains subdirectories used by Julia for various purposes. Here is an overview of some of the subdirectories that may exist in a depot:

- artifacts: Contains content that packages use for which Pkg manages the installation of.
- clones: Contains full clones of package repos. Maintained by Pkg.jl and used as a cache.
- config: Contains julia-level configuration such as a startup.jl
- compiled: Contains precompiled *.ji files for packages. Maintained by Julia.
- dev: Default directory for Pkg.develop. Maintained by Pkg.jl and the user.
- environments: Default package environments. For instance the global environment for a specific julia version. Maintained by Pkg.jl.
- logs: Contains logs of Pkg and REPL operations. Maintained by Pkg.jl and Julia.
- packages: Contains packages, some of which were explicitly installed and some which are implicit dependencies. Maintained by Pkg.jl.

- `registries`: Contains package registries. By default only `General`. Maintained by `Pkg.jl`.
- `scratchspaces`: Contains content that a package itself installs via the `Scratch.jl` package. `Pkg.gc()` will delete content that is known to be unused.

> **Note**
>
> Packages that want to store content should use the `scratchspaces` subdirectory via `Scratch.jl` instead of creating new subdirectories in the depot root.

See also `JULIA_DEPOT_PATH`, and Code Loading.

source

`Base.LOAD_PATH` – Constant.

```
LOAD_PATH
```

An array of paths for `using` and `import` statements to consider as project environments or package directories when loading code. It is populated based on the `JULIA_LOAD_PATH` environment variable if set; otherwise it defaults to `["@", "@v#.#", "@stdlib"]`. Entries starting with `@` have special meanings:

- `@` refers to the "current active environment", the initial value of which is initially determined by the `JULIA_PROJECT` environment variable or the `--project` command-line option.
- `@stdlib` expands to the absolute path of the current Julia installation's standard library directory.
- `@name` refers to a named environment, which are stored in depots (see `JULIA_DEPOT_PATH`) under the `environments` subdirectory. The user's named environments are stored in `~/.julia/environments` so `@name` would refer to the environment in `~/.julia/environments/name` if it exists and contains a `Project.toml` file. If `name` contains `#` characters, then they are replaced with the major, minor and patch components of the Julia version number. For example, if you are running Julia 1.2 then `@v#.#` expands to `@v1.2` and will look for an environment by that name, typically at `~/.julia/environments/v1.2`.

The fully expanded value of `LOAD_PATH` that is searched for projects and packages can be seen by calling the `Base.load_path()` function.

See also `JULIA_LOAD_PATH`, `JULIA_PROJECT`, `JULIA_DEPOT_PATH`, and Code Loading.

source

`Base.Sys.BINDIR` – Constant.

```
Sys.BINDIR::String
```

A string containing the full path to the directory containing the `julia` executable.

source

`Base.Sys.CPU_THREADS` – Constant.

```
Sys.CPU_THREADS::Int
```

The number of logical CPU cores available in the system, i.e. the number of threads that the CPU can run concurrently. Note that this is not necessarily the number of CPU cores, for example, in the presence of hyper-threading.

See Hwloc.jl or CpuId.jl for extended information, including number of physical cores.

source

`Base.Sys.WORD_SIZE` – Constant.

```
Sys.WORD_SIZE::Int
```

Standard word size on the current machine, in bits.

source

`Base.Sys.KERNEL` – Constant.

```
Sys.KERNEL::Symbol
```

A symbol representing the name of the operating system, as returned by `uname` of the build configuration.

source

`Base.Sys.ARCH` – Constant.

```
Sys.ARCH::Symbol
```

A symbol representing the architecture of the build configuration.

source

`Base.Sys.MACHINE` – Constant.

```
Sys.MACHINE::String
```

A string containing the build triple.

source

参见：

- `stdin`
- `stdout`
- `stderr`
- `ENV`
- `ENDIAN_BOM`

Chapter 51

文件系统

`Base.Filesystem.pwd` – Function.

```
pwd() -> String
```

Get the current working directory.

See also: `cd`, `tempdir`.

Examples

```
julia> pwd()
"/home/JuliaUser"

julia> cd("/home/JuliaUser/Projects/julia")

julia> pwd()
"/home/JuliaUser/Projects/julia"
```

source

`Base.Filesystem.cd` – Method.

```
cd(dir::AbstractString=homedir())
```

Set the current working directory.

See also: `pwd`, `mkdir`, `mkpath`, `mktempdir`.

Examples

```
julia> cd("/home/JuliaUser/Projects/julia")

julia> pwd()
"/home/JuliaUser/Projects/julia"

julia> cd()
```

```
julia> pwd()
"/home/JuliaUser"
```

source

Base.Filesystem.cd – Method.

```
cd(f::Function, dir::AbstractString=homedir())
```

Temporarily change the current working directory to `dir`, apply function `f` and finally return to the original directory.

Examples

```
julia> pwd()
"/home/JuliaUser"

julia> cd(readdir, "/home/JuliaUser/Projects/julia")
34-element Array{String,1}:
 ".circleci"
 ".freebsdci.sh"
 ".git"
 ".gitattributes"
 ".github"
 ⋮
 "test"
 "ui"
 "usr"
 "usr-staging"

julia> pwd()
"/home/JuliaUser"
```

source

Base.Filesystem.readdir – Function.

```
readdir(dir::AbstractString=pwd();
    join::Bool = false,
    sort::Bool = true,
) -> Vector{String}
```

Return the names in the directory `dir` or the current working directory if not given. When `join` is false, `readdir` returns just the names in the directory as is; when `join` is true, it returns `joinpath(dir, name)` for each `name` so that the returned strings are full paths. If you want to get absolute paths back, call `readdir` with an absolute directory path and `join` set to true.

By default, `readdir` sorts the list of names it returns. If you want to skip sorting the names and get them in the order that the file system lists them, you can use `readdir(dir, sort=false)` to opt out of sorting.

See also: walkdir.

Julia 1.4

The join and sort keyword arguments require at least Julia 1.4.

Examples

```
julia> cd("/home/JuliaUser/dev/julia")

julia> readdir()
30-element Array{String,1}:
 ".appveyor.yml"
 ".git"
 ".gitattributes"
 ⋮
 "ui"
 "usr"
 "usr-staging"

julia> readdir(join=true)
30-element Array{String,1}:
 "/home/JuliaUser/dev/julia/.appveyor.yml"
 "/home/JuliaUser/dev/julia/.git"
 "/home/JuliaUser/dev/julia/.gitattributes"
 ⋮
 "/home/JuliaUser/dev/julia/ui"
 "/home/JuliaUser/dev/julia/usr"
 "/home/JuliaUser/dev/julia/usr-staging"

julia> readdir("base")
145-element Array{String,1}:
 ".gitignore"
 "Base.jl"
 "Enums.jl"
 ⋮
 "version_git.sh"
 "views.jl"
 "weakkeydict.jl"

julia> readdir("base", join=true)
145-element Array{String,1}:
 "base/.gitignore"
 "base/Base.jl"
 "base/Enums.jl"
 ⋮
 "base/version_git.sh"
 "base/views.jl"
 "base/weakkeydict.jl"

julia> readdir(abspath("base"), join=true)
145-element Array{String,1}:
 "/home/JuliaUser/dev/julia/base/.gitignore"
 "/home/JuliaUser/dev/julia/base/Base.jl"
 "/home/JuliaUser/dev/julia/base/Enums.jl"
```

```
⋮
"/home/JuliaUser/dev/julia/base/version_git.sh"
"/home/JuliaUser/dev/julia/base/views.jl"
"/home/JuliaUser/dev/julia/base/weakkeydict.jl"
```

source

Base.Filesystem.walkdir – Function.

```
walkdir(dir; topdown=true, follow_symlinks=false, onerror=throw)
```

Return an iterator that walks the directory tree of a directory. The iterator returns a tuple containing (rootpath, dirs, files). The directory tree can be traversed top-down or bottom-up. If walkdir or stat encounters a IOError it will rethrow the error by default. A custom error handling function can be provided through onerror keyword argument. onerror is called with a IOError as argument.

See also: readdir.

Examples

```
for (root, dirs, files) in walkdir(".")
    println("Directories in $root")
    for dir in dirs
        println(joinpath(root, dir)) # path to directories
    end
    println("Files in $root")
    for file in files
        println(joinpath(root, file)) # path to files
    end
end
```

```
julia> mkpath("my/test/dir");

julia> itr = walkdir("my");

julia> (root, dirs, files) = first(itr)
("my", ["test"], String[])

julia> (root, dirs, files) = first(itr)
("my/test", ["dir"], String[])

julia> (root, dirs, files) = first(itr)
("my/test/dir", String[], String[])
```

source

Base.Filesystem.mkdir – Function.

```
mkdir(path::AbstractString; mode::Unsigned = 0o777)
```

Make a new directory with name `path` and permissions `mode`. `mode` defaults to `0o777`, modified by the current file creation mask. This function never creates more than one directory. If the directory already exists, or some intermediate directories do not exist, this function throws an error. See `mkpath` for a function which creates all required intermediate directories. Return `path`.

Examples

```
julia> mkdir("testingdir")
"testingdir"

julia> cd("testingdir")

julia> pwd()
"/home/JuliaUser/testingdir"
```

source

`Base.Filesystem.mkpath` – Function.

```
mkpath(path::AbstractString; mode::Unsigned = 0o777)
```

Create all intermediate directories in the `path` as required. Directories are created with the permissions `mode` which defaults to `0o777` and is modified by the current file creation mask. Unlike `mkdir`, `mkpath` does not error if `path` (or parts of it) already exists. However, an error will be thrown if `path` (or parts of it) points to an existing file. Return `path`.

If `path` includes a filename you will probably want to use `mkpath(dirname(path))` to avoid creating a directory using the filename.

Examples

```
julia> cd(mktempdir())

julia> mkpath("my/test/dir") # creates three directories
"my/test/dir"

julia> readdir()
1-element Array{String,1}:
 "my"

julia> cd("my")

julia> readdir()
1-element Array{String,1}:
 "test"

julia> readdir("test")
1-element Array{String,1}:
 "dir"
```

```
julia> mkpath("intermediate_dir/actually_a_directory.txt") # creates two directories
"intermediate_dir/actually_a_directory.txt"

julia> isdir("intermediate_dir/actually_a_directory.txt")
true
```

source

Base.Filesystem.hardlink – Function.

```
hardlink(src::AbstractString, dst::AbstractString)
```

Creates a hard link to an existing source file `src` with the name `dst`. The destination, `dst`, must not exist.

See also: `symlink`.

Julia 1.8

This method was added in Julia 1.8.

source

Base.Filesystem.symlink – Function.

```
symlink(target::AbstractString, link::AbstractString; dir_target = false)
```

Creates a symbolic link to `target` with the name `link`.

On Windows, symlinks must be explicitly declared as referring to a directory or not. If `target` already exists, by default the type of `link` will be auto- detected, however if `target` does not exist, this function defaults to creating a file symlink unless `dir_target` is set to `true`. Note that if the user sets `dir_target` but `target` exists and is a file, a directory symlink will still be created, but dereferencing the symlink will fail, just as if the user creates a file symlink (by calling `symlink()` with `dir_target` set to `false` before the directory is created) and tries to dereference it to a directory.

Additionally, there are two methods of making a link on Windows; symbolic links and junction points. Junction points are slightly more efficient, but do not support relative paths, so if a relative directory symlink is requested (as denoted by `isabspath(target)` returning `false`) a symlink will be used, else a junction point will be used. Best practice for creating symlinks on Windows is to create them only after the files/directories they reference are already created.

See also: `hardlink`.

Note

This function raises an error under operating systems that do not support soft symbolic links, such as Windows XP.

> **Julia 1.6**
>
> The `dir_target` keyword argument was added in Julia 1.6. Prior to this, symlinks to nonexistent paths on windows would always be file symlinks, and relative symlinks to directories were not supported.

source

`Base.Filesystem.readlink` – Function.

```
readlink(path::AbstractString) -> String
```

Return the target location a symbolic link `path` points to.

source

`Base.Filesystem.chmod` – Function.

```
chmod(path::AbstractString, mode::Integer; recursive::Bool=false)
```

Change the permissions mode of `path` to `mode`. Only integer `mode`s (e.g. `0o777`) are currently supported. If `recursive=true` and the path is a directory all permissions in that directory will be recursively changed. Return `path`.

> **Note**
>
> Prior to Julia 1.6, this did not correctly manipulate filesystem ACLs on Windows, therefore it would only set read-only bits on files. It now is able to manipulate ACLs.

source

`Base.Filesystem.chown` – Function.

```
chown(path::AbstractString, owner::Integer, group::Integer=-1)
```

Change the owner and/or group of `path` to `owner` and/or `group`. If the value entered for `owner` or `group` is `-1` the corresponding ID will not change. Only integer `owner`s and `group`s are currently supported. Return `path`.

source

`Base.Libc.RawFD` – Type.

```
RawFD
```

Primitive type which wraps the native OS file descriptor. `RawFD`s can be passed to methods like `stat` to discover information about the underlying file, and can also be used to open streams, with the `RawFD` describing the OS file backing the stream.

source

`Base.stat` – Function.

```
stat(file)
```

Return a structure whose fields contain information about the file. The fields of the structure are:

Name	Description
desc	The path or OS file descriptor
size	The size (in bytes) of the file
device	ID of the device that contains the file
inode	The inode number of the file
mode	The protection mode of the file
nlink	The number of hard links to the file
uid	The user id of the owner of the file
gid	The group id of the file owner
rdev	If this file refers to a device, the ID of the device it refers to
blksize	The file-system preferred block size for the file
blocks	The number of such blocks allocated
mtime	Unix timestamp of when the file was last modified
ctime	Unix timestamp of when the file's metadata was changed

source

`Base.Filesystem.diskstat` – Function.

```
diskstat(path=pwd())
```

Returns statistics in bytes about the disk that contains the file or directory pointed at by `path`. If no argument is passed, statistics about the disk that contains the current working directory are returned.

Julia 1.8

This method was added in Julia 1.8.

source

`Base.Filesystem.lstat` – Function.

```
lstat(file)
```

Like `stat`, but for symbolic links gets the info for the link itself rather than the file it refers to. This function must be called on a file path rather than a file object or a file descriptor.

source

`Base.Filesystem.ctime` – Function.

```
ctime(file)
```

Equivalent to `stat(file).ctime`.

source

`Base.Filesystem.mtime` – Function.

```
mtime(file)
```

Equivalent to `stat(file).mtime`.

source

`Base.Filesystem.filemode` – Function.

```
filemode(file)
```

Equivalent to `stat(file).mode`.

source

`Base.filesize` – Function.

```
filesize(path...)
```

Equivalent to `stat(file).size`.

source

`Base.Filesystem.uperm` – Function.

```
uperm(file)
```

Get the permissions of the owner of the file as a bitfield of

Value	Description
01	Execute Permission
02	Write Permission
04	Read Permission

For allowed arguments, see `stat`.

source

`Base.Filesystem.gperm` – Function.

```
gperm(file)
```

Like `uperm` but gets the permissions of the group owning the file.

source

`Base.Filesystem.operm` – Function.

```
operm(file)
```

Like `uperm` but gets the permissions for people who neither own the file nor are a member of the group owning the file

source

`Base.Filesystem.cp` – Function.

```
cp(src::AbstractString, dst::AbstractString; force::Bool=false, follow_symlinks::Bool=false)
```

Copy the file, link, or directory from `src` to `dst`. `force=true` will first remove an existing `dst`.

If `follow_symlinks=false`, and `src` is a symbolic link, `dst` will be created as a symbolic link. If `follow_symlinks=true` and `src` is a symbolic link, `dst` will be a copy of the file or directory `src` refers to. Return `dst`.

> **Note**
>
> The `cp` function is different from the `cp` command. The `cp` function always operates on the assumption that `dst` is a file, while the command does different things depending on whether `dst` is a directory or a file. Using `force=true` when `dst` is a directory will result in loss of all the contents present in the `dst` directory, and `dst` will become a file that has the contents of `src` instead.

source

`Base.download` – Function.

```
download(url::AbstractString, [path::AbstractString = tempname()]) -> path
```

Download a file from the given url, saving it to the location `path`, or if not specified, a temporary path. Returns the path of the downloaded file.

> **Note**
>
> Since Julia 1.6, this function is deprecated and is just a thin wrapper around `Downloads.download`. In new code, you should use that function directly instead of calling this.

source

`Base.Filesystem.mv` – Function.

```
mv(src::AbstractString, dst::AbstractString; force::Bool=false)
```

Move the file, link, or directory from `src` to `dst`. `force=true` will first remove an existing `dst`. Return `dst`.

Examples

```
julia> write("hello.txt", "world");

julia> mv("hello.txt", "goodbye.txt")
"goodbye.txt"

julia> "hello.txt" in readdir()
false

julia> readline("goodbye.txt")
"world"

julia> write("hello.txt", "world2");

julia> mv("hello.txt", "goodbye.txt")
ERROR: ArgumentError: 'goodbye.txt' exists. `force=true` is required to remove 'goodbye.txt'
↪  before moving.
Stacktrace:
 [1] #checkfor_mv_cp_cptree#10(::Bool, ::Function, ::String, ::String, ::String) at
↪  ./file.jl:293
[...]

julia> mv("hello.txt", "goodbye.txt", force=true)
"goodbye.txt"

julia> rm("goodbye.txt");
```

source

`Base.Filesystem.rm` – Function.

```
rm(path::AbstractString; force::Bool=false, recursive::Bool=false)
```

Delete the file, link, or empty directory at the given path. If `force=true` is passed, a non-existing path is not treated as error. If `recursive=true` is passed and the path is a directory, then all contents are removed recursively.

Examples

```
julia> mkpath("my/test/dir");

julia> rm("my", recursive=true)

julia> rm("this_file_does_not_exist", force=true)

julia> rm("this_file_does_not_exist")
```

```
ERROR: IOError: unlink("this_file_does_not_exist"): no such file or directory (ENOENT)
Stacktrace:
[...]
```

source

Base.Filesystem.touch – Function.

```
Base.touch(::Pidfile.LockMonitor)
```

Update the mtime on the lock, to indicate it is still fresh.

See also the refresh keyword in the mkpidlock constructor.

```
touch(path::AbstractString)
touch(fd::File)
```

Update the last-modified timestamp on a file to the current time.

If the file does not exist a new file is created.

Return path.

Examples

```
julia> write("my_little_file", 2);

julia> mtime("my_little_file")
1.5273815391135583e9

julia> touch("my_little_file");

julia> mtime("my_little_file")
1.527381559163435e9
```

We can see the mtime has been modified by touch.

source

Base.Filesystem.tempname – Function.

```
tempname(parent=tempdir(); cleanup=true) -> String
```

Generate a temporary file path. This function only returns a path; no file is created. The path is likely to be unique, but this cannot be guaranteed due to the very remote possibility of two simultaneous calls to tempname generating the same file name. The name is guaranteed to differ from all files already existing at the time of the call to tempname.

When called with no arguments, the temporary name will be an absolute path to a temporary name in the system temporary directory as given by tempdir(). If a parent directory argument is given, the temporary path will be in that directory instead.

The `cleanup` option controls whether the process attempts to delete the returned path automatically when the process exits. Note that the `tempname` function does not create any file or directory at the returned location, so there is nothing to cleanup unless you create a file or directory there. If you do and `clean` is `true` it will be deleted upon process termination.

Julia 1.4

The `parent` and `cleanup` arguments were added in 1.4. Prior to Julia 1.4 the path `tempname` would never be cleaned up at process termination.

Warning

This can lead to security holes if another process obtains the same file name and creates the file before you are able to. Open the file with `JL_O_EXCL` if this is a concern. Using `mktemp()` is also recommended instead.

source

`Base.Filesystem.tempdir` – Function.

```
tempdir()
```

Gets the path of the temporary directory. On Windows, `tempdir()` uses the first environment variable found in the ordered list `TMP`, `TEMP`, `USERPROFILE`. On all other operating systems, `tempdir()` uses the first environment variable found in the ordered list `TMPDIR`, `TMP`, `TEMP`, and `TEMPDIR`. If none of these are found, the path `"/tmp"` is used.

source

`Base.Filesystem.mktemp` – Method.

```
mktemp(parent=tempdir(); cleanup=true) -> (path, io)
```

Return `(path, io)`, where `path` is the path of a new temporary file in `parent` and `io` is an open file object for this path. The `cleanup` option controls whether the temporary file is automatically deleted when the process exits.

Julia 1.3

The `cleanup` keyword argument was added in Julia 1.3. Relatedly, starting from 1.3, Julia will remove the temporary paths created by `mktemp` when the Julia process exits, unless `cleanup` is explicitly set to `false`.

source

`Base.Filesystem.mktemp` – Method.

```
mktemp(f::Function, parent=tempdir())
```

Apply the function `f` to the result of `mktemp(parent)` and remove the temporary file upon completion.

See also: `mktempdir`.

source

`Base.Filesystem.mktempdir` – Method.

```
mktempdir(parent=tempdir(); prefix="jl_", cleanup=true) -> path
```

Create a temporary directory in the `parent` directory with a name constructed from the given `prefix` and a random suffix, and return its path. Additionally, on some platforms, any trailing `'X'` characters in `prefix` may be replaced with random characters. If `parent` does not exist, throw an error. The `cleanup` option controls whether the temporary directory is automatically deleted when the process exits.

> **Julia 1.2**
>
> The `prefix` keyword argument was added in Julia 1.2.

> **Julia 1.3**
>
> The `cleanup` keyword argument was added in Julia 1.3. Relatedly, starting from 1.3, Julia will remove the temporary paths created by `mktempdir` when the Julia process exits, unless `cleanup` is explicitly set to `false`.

See also: `mktemp`, `mkdir`.

source

`Base.Filesystem.mktempdir` – Method.

```
mktempdir(f::Function, parent=tempdir(); prefix="jl_")
```

Apply the function `f` to the result of `mktempdir(parent; prefix)` and remove the temporary directory all of its contents upon completion.

See also: `mktemp`, `mkdir`.

> **Julia 1.2**
>
> The `prefix` keyword argument was added in Julia 1.2.

source

`Base.Filesystem.isblockdev` – Function.

```
isblockdev(path) -> Bool
```

Return `true` if `path` is a block device, `false` otherwise.

source

`Base.Filesystem.ischardev` – Function.

```
ischardev(path) -> Bool
```

Return `true` if `path` is a character device, `false` otherwise.

source

`Base.Filesystem.isdir` – Function.

```
isdir(path) -> Bool
```

Return `true` if `path` is a directory, `false` otherwise.

Examples

```
julia> isdir(homedir())
true

julia> isdir("not/a/directory")
false
```

See also `isfile` and `ispath`.

source

`Base.Filesystem.isfifo` – Function.

```
isfifo(path) -> Bool
```

Return `true` if `path` is a FIFO, `false` otherwise.

source

`Base.Filesystem.isfile` – Function.

```
isfile(path) -> Bool
```

Return `true` if `path` is a regular file, `false` otherwise.

Examples

```
julia> isfile(homedir())
false

julia> filename = "test_file.txt";

julia> write(filename, "Hello world!");

julia> isfile(filename)
true

julia> rm(filename);

julia> isfile(filename)
false
```

See also `isdir` and `ispath`.

source

`Base.Filesystem.islink` – Function.

```
islink(path) -> Bool
```

Return `true` if `path` is a symbolic link, `false` otherwise.

source

`Base.Filesystem.ismount` – Function.

```
ismount(path) -> Bool
```

Return `true` if `path` is a mount point, `false` otherwise.

source

`Base.Filesystem.ispath` – Function.

```
ispath(path) -> Bool
```

Return `true` if a valid filesystem entity exists at `path`, otherwise returns `false`. This is the generalization of `isfile`, `isdir` etc.

source

`Base.Filesystem.issetgid` – Function.

```
issetgid(path) -> Bool
```

Return `true` if `path` has the setgid flag set, `false` otherwise.

source

`Base.Filesystem.issetuid` – Function.

```
issetuid(path) -> Bool
```

Return `true` if `path` has the setuid flag set, `false` otherwise.

source

`Base.Filesystem.issocket` – Function.

```
issocket(path) -> Bool
```

Return `true` if `path` is a socket, `false` otherwise.

source

`Base.Filesystem.issticky` – Function.

```
issticky(path) -> Bool
```

Return `true` if `path` has the sticky bit set, `false` otherwise.

source

`Base.Filesystem.homedir` – Function.

```
homedir() -> String
```

Return the current user's home directory.

Note

`homedir` determines the home directory via `libuv`'s `uv_os_homedir`. For details (for example on how to specify the home directory via environment variables), see the `uv_os_homedir` documentation.

source

`Base.Filesystem.dirname` – Function.

```
dirname(path::AbstractString) -> String
```

Get the directory part of a path. Trailing characters ('/' or '\') in the path are counted as part of the path.

Examples

```
julia> dirname("/home/myuser")
"/home"

julia> dirname("/home/myuser/")
"/home/myuser"
```

See also basename.

source

Base.Filesystem.basename – Function.

```
basename(path::AbstractString) -> String
```

Get the file name part of a path.

Note

This function differs slightly from the Unix basename program, where trailing slashes are ignored, i.e. `$ basename /foo/bar/` returns `bar`, whereas basename in Julia returns an empty string `""`.

Examples

```
julia> basename("/home/myuser/example.jl")
"example.jl"

julia> basename("/home/myuser/")
""
```

See also dirname.

source

Base.Filesystem.isabspath – Function.

```
isabspath(path::AbstractString) -> Bool
```

Determine whether a path is absolute (begins at the root directory).

Examples

```
julia> isabspath("/home")
true

julia> isabspath("home")
false
```

source

`Base.Filesystem.isdirpath` – Function.

```
isdirpath(path::AbstractString) -> Bool
```

Determine whether a path refers to a directory (for example, ends with a path separator).

Examples

```
julia> isdirpath("/home")
false

julia> isdirpath("/home/")
true
```

source

`Base.Filesystem.joinpath` – Function.

```
joinpath(parts::AbstractString...) -> String
joinpath(parts::Vector{AbstractString}) -> String
joinpath(parts::Tuple{AbstractString}) -> String
```

Join path components into a full path. If some argument is an absolute path or (on Windows) has a drive specification that doesn't match the drive computed for the join of the preceding paths, then prior components are dropped.

Note on Windows since there is a current directory for each drive, `joinpath("c:", "foo")` represents a path relative to the current directory on drive "c:" so this is equal to "c:foo", not "c:\foo". Furthermore, `joinpath` treats this as a non-absolute path and ignores the drive letter casing, hence `joinpath("C:\A","c:b") = "C:\A\b"`.

Examples

```
julia> joinpath("/home/myuser", "example.jl")
"/home/myuser/example.jl"
```

```
julia> joinpath(["/home/myuser", "example.jl"])
"/home/myuser/example.jl"
```

source

`Base.Filesystem.abspath` – Function.

```
abspath(path::AbstractString) -> String
```

Convert a path to an absolute path by adding the current directory if necessary. Also normalizes the path as in `normpath`.

Example

If you are in a directory called `JuliaExample` and the data you are using is two levels up relative to the `JuliaExample` directory, you could write:

abspath(”../../data”)

Which gives a path like `"/home/JuliaUser/data/"`.

See also `joinpath`, `pwd`, `expanduser`.

source

```
abspath(path::AbstractString, paths::AbstractString...) -> String
```

Convert a set of paths to an absolute path by joining them together and adding the current directory if necessary. Equivalent to `abspath(joinpath(path, paths...))`.

source

`Base.Filesystem.normpath` – Function.

```
normpath(path::AbstractString) -> String
```

Normalize a path, removing ”.” and ”..” entries and changing ”/” to the canonical path separator for the system.

Examples

```
julia> normpath("/home/myuser/../example.jl")
"/home/example.jl"

julia> normpath("Documents/Julia") == joinpath("Documents", "Julia")
true
```

source

```
normpath(path::AbstractString, paths::AbstractString...) -> String
```

Convert a set of paths to a normalized path by joining them together and removing ”.” and ”..” entries. Equivalent to `normpath(joinpath(path, paths...))`.

source

`Base.Filesystem.realpath` – Function.

```
realpath(path::AbstractString) -> String
```

Canonicalize a path by expanding symbolic links and removing ”.” and ”..” entries. On case-insensitive case-preserving filesystems (typically Mac and Windows), the filesystem's stored case for the path is returned.

(This function throws an exception if `path` does not exist in the filesystem.)

source

`Base.Filesystem.relpath` – Function.

```
relpath(path::AbstractString, startpath::AbstractString = ".") -> String
```

Return a relative filepath to `path` either from the current directory or from an optional start directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of `path` or `startpath`.

On Windows, case sensitivity is applied to every part of the path except drive letters. If `path` and `startpath` refer to different drives, the absolute path of `path` is returned.

source

`Base.Filesystem.expanduser` – Function.

```
expanduser(path::AbstractString) -> AbstractString
```

On Unix systems, replace a tilde character at the start of a path with the current user's home directory.

See also: `contractuser`.

source

`Base.Filesystem.contractuser` – Function.

```
contractuser(path::AbstractString) -> AbstractString
```

On Unix systems, if the path starts with `homedir()`, replace it with a tilde character.

See also: `expanduser`.

source

`Base.Filesystem.samefile` – Function.

```
samefile(path_a::AbstractString, path_b::AbstractString)
```

Check if the paths `path_a` and `path_b` refer to the same existing file or directory.

source

`Base.Filesystem.splitdir` – Function.

```
splitdir(path::AbstractString) -> (AbstractString, AbstractString)
```

Split a path into a tuple of the directory name and file name.

Examples

```
julia> splitdir("/home/myuser")
("/home", "myuser")
```

source

Base.Filesystem.splitdrive – Function.

```
splitdrive(path::AbstractString) -> (AbstractString, AbstractString)
```

On Windows, split a path into the drive letter part and the path part. On Unix systems, the first component is always the empty string.

source

Base.Filesystem.splitext – Function.

```
splitext(path::AbstractString) -> (String, String)
```

If the last component of a path contains one or more dots, split the path into everything before the last dot and everything including and after the dot. Otherwise, return a tuple of the argument unmodified and the empty string. ”splitext” is short for ”split extension”.

Examples

```
julia> splitext("/home/myuser/example.jl")
("/home/myuser/example", ".jl")

julia> splitext("/home/myuser/example.tar.gz")
("/home/myuser/example.tar", ".gz")

julia> splitext("/home/my.user/example")
("/home/my.user/example", "")
```

source

Base.Filesystem.splitpath – Function.

```
splitpath(path::AbstractString) -> Vector{String}
```

Split a file path into all its path components. This is the opposite of `joinpath`. Returns an array of substrings, one for each directory or file in the path, including the root directory if present.

Julia 1.1

This function requires at least Julia 1.1.

Examples

```
julia> splitpath("/home/myuser/example.jl")
4-element Vector{String}:
 "/"
 "home"
 "myuser"
 "example.jl"
```

source

Chapter 52

I/O 与网络

52.1 通用 I/O

`Base.stdout` – Constant.

```
stdout::IO
```

Global variable referring to the standard out stream.

source

`Base.stderr` – Constant.

```
stderr::IO
```

Global variable referring to the standard error stream.

source

`Base.stdin` – Constant.

```
stdin::IO
```

Global variable referring to the standard input stream.

source

`Base.open` – Function.

```
open(f::Function, args...; kwargs...)
```

Apply the function `f` to the result of `open(args...; kwargs...)` and close the resulting file descriptor upon completion.

Examples

```
julia> write("myfile.txt", "Hello world!");

julia> open(io->read(io, String), "myfile.txt")
"Hello world!"

julia> rm("myfile.txt")
```

source

```
open(filename::AbstractString; lock = true, keywords...) -> IOStream
```

Open a file in a mode specified by five boolean keyword arguments:

Keyword	Description	Default
`read`	open for reading	`!write`
`write`	open for writing	`truncate | append`
`create`	create if non-existent	`!read & write | truncate | append`
`truncate`	truncate to zero size	`!read & write`
`append`	seek to end	`false`

The default when no keywords are passed is to open files for reading only. Returns a stream for accessing the opened file.

The `lock` keyword argument controls whether operations will be locked for safe multi-threaded access.

Julia 1.5

The `lock` argument is available as of Julia 1.5.

source

```
open(filename::AbstractString, [mode::AbstractString]; lock = true) -> IOStream
```

Alternate syntax for open, where a string-based mode specifier is used instead of the five booleans. The values of `mode` correspond to those from `fopen(3)` or Perl `open`, and are equivalent to setting the following boolean groups:

Mode	Description	Keywords
`r`	read	none
`w`	write, create, truncate	`write = true`
`a`	write, create, append	`append = true`
`r+`	read, write	`read = true, write = true`
`w+`	read, write, create, truncate	`truncate = true, read = true`
`a+`	read, write, create, append	`append = true, read = true`

The `lock` keyword argument controls whether operations will be locked for safe multi-threaded access.

Examples

```
julia> io = open("myfile.txt", "w");

julia> write(io, "Hello world!");

julia> close(io);

julia> io = open("myfile.txt", "r");

julia> read(io, String)
"Hello world!"

julia> write(io, "This file is read only")
ERROR: ArgumentError: write failed, IOStream is not writeable
[...]

julia> close(io)

julia> io = open("myfile.txt", "a");

julia> write(io, "This stream is not read only")
28

julia> close(io)

julia> rm("myfile.txt")
```

Julia 1.5

The `lock` argument is available as of Julia 1.5.

source

```
open(fd::OS_HANDLE) -> IO
```

Take a raw file descriptor wrap it in a Julia-aware IO type, and take ownership of the fd handle. Call `open(Libc.dup(fd))` to avoid the ownership capture of the original handle.

Warning

Do not call this on a handle that's already owned by some other part of the system.

source

```
open(command, mode::AbstractString, stdio=devnull)
```

Run command asynchronously. Like `open(command, stdio; read, write)` except specifying the read and write flags via a mode string instead of keyword arguments. Possible mode strings are:

source

Mode	Description	Keywords
r	read	none
w	write	write = true
r+	read, write	read = true, write = true
w+	read, write	read = true, write = true

```
open(command, stdio=devnull; write::Bool = false, read::Bool = !write)
```

Start running command asynchronously, and return a process::IO object. If read is true, then reads from the process come from the process's standard output and stdio optionally specifies the process's standard input stream. If write is true, then writes go to the process's standard input and stdio optionally specifies the process's standard output stream. The process's standard error stream is connected to the current global stderr.

source

```
open(f::Function, command, args...; kwargs...)
```

Similar to open(command, args...; kwargs...), but calls f(stream) on the resulting process stream, then closes the input stream and waits for the process to complete. Return the value returned by f on success. Throw an error if the process failed, or if the process attempts to print anything to stdout.

source

Base.IOStream – Type.

```
IOStream
```

A buffered IO stream wrapping an OS file descriptor. Mostly used to represent files returned by open.

source

Base.IOBuffer – Type.

```
IOBuffer([data::AbstractVector{UInt8}]; keywords...) -> IOBuffer
```

Create an in-memory I/O stream, which may optionally operate on a pre-existing array.

It may take optional keyword arguments:

- read, write, append: restricts operations to the buffer; see open for details.
- truncate: truncates the buffer size to zero length.
- maxsize: specifies a size beyond which the buffer may not be grown.
- sizehint: suggests a capacity of the buffer (data must implement sizehint!(data, size)).

When data is not given, the buffer will be both readable and writable by default.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members.")
56

julia> String(take!(io))
"JuliaLang is a GitHub organization. It has many members."

julia> io = IOBuffer(b"JuliaLang is a GitHub organization.")
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false, size=35,
↪  maxsize=Inf, ptr=1, mark=-1)

julia> read(io, String)
"JuliaLang is a GitHub organization."

julia> write(io, "This isn't writable.")
ERROR: ArgumentError: ensureroom failed, IOBuffer is not writeable

julia> io = IOBuffer(UInt8[], read=true, write=true, maxsize=34)
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false, size=0,
↪  maxsize=34, ptr=1, mark=-1)

julia> write(io, "JuliaLang is a GitHub organization.")
34

julia> String(take!(io))
"JuliaLang is a GitHub organization"

julia> length(read(IOBuffer(b"data", read=true, truncate=false)))
4

julia> length(read(IOBuffer(b"data", read=true, truncate=true)))
0
```

source

```
IOBuffer(string::String)
```

Create a read-only `IOBuffer` on the data underlying the given string.

Examples

```
julia> io = IOBuffer("Haho");

julia> String(take!(io))
"Haho"

julia> String(take!(io))
"Haho"
```

source

`Base.take!` – Method.

```
take!(b::IOBuffer)
```

Obtain the contents of an `IOBuffer` as an array. Afterwards, the `IOBuffer` is reset to its initial state.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members.")
56

julia> String(take!(io))
"JuliaLang is a GitHub organization. It has many members."
```

source

`Base.fdio` – Function.

```
fdio([name::AbstractString, ]fd::Integer[, own::Bool=false]) -> IOStream
```

Create an `IOStream` object from an integer file descriptor. If own is `true`, closing this object will close the underlying descriptor. By default, an `IOStream` is closed when it is garbage collected. name allows you to associate the descriptor with a named file.

source

`Base.flush` – Function.

```
flush(stream)
```

Commit all currently buffered writes to the given stream.

source

`Base.close` – Function.

```
close(stream)
```

Close an I/O stream. Performs a `flush` first.

source

`Base.closewrite` – Function.

```
closewrite(stream)
```

Shutdown the write half of a full-duplex I/O stream. Performs a `flush` first. Notify the other end that no more data will be written to the underlying file. This is not supported by all IO types.

Examples

```
julia> io = Base.BufferStream(); # this never blocks, so we can read and write on the same Task

julia> write(io, "request");

julia> # calling `read(io)` here would block forever

julia> closewrite(io);

julia> read(io, String)
"request"
```

source

Base.write – Function.

```
write(io::IO, x)
```

Write the canonical binary representation of a value to the given I/O stream or file. Return the number of bytes written into the stream. See also `print` to write a text representation (with an encoding that may depend upon `io`).

The endianness of the written value depends on the endianness of the host system. Convert to/from a fixed endianness when writing/reading (e.g. using `htol` and `ltoh`) to get results that are consistent across platforms.

You can write multiple values with the same `write` call. i.e. the following are equivalent:

```
write(io, x, y...)
write(io, x) + write(io, y...)
```

Examples

Consistent serialization:

```
julia> fname = tempname(); # random temporary filename

julia> open(fname,"w") do f
           # Make sure we write 64bit integer in little-endian byte order
           write(f,htol(Int64(42)))
       end
8

julia> open(fname,"r") do f
           # Convert back to host byte order and host integer type
           Int(ltoh(read(f,Int64)))
       end
42
```

Merging write calls:

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.", " It has many members.")
56

julia> String(take!(io))
"JuliaLang is a GitHub organization. It has many members."

julia> write(io, "Sometimes those members") + write(io, " write documentation.")
44

julia> String(take!(io))
"Sometimes those members write documentation."
```

User-defined plain-data types without `write` methods can be written when wrapped in a `Ref`:

```
julia> struct MyStruct; x::Float64; end

julia> io = IOBuffer()
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false, size=0,
↪  maxsize=Inf, ptr=1, mark=-1)

julia> write(io, Ref(MyStruct(42.0)))
8

julia> seekstart(io); read!(io, Ref(MyStruct(NaN)))
Base.RefValue{MyStruct}(MyStruct(42.0))
```

source

`Base.read` – Function.

```
read(io::IO, T)
```

Read a single value of type T from `io`, in canonical binary representation.

Note that Julia does not convert the endianness for you. Use `ntoh` or `ltoh` for this purpose.

```
read(io::IO, String)
```

Read the entirety of `io`, as a `String` (see also `readchomp`).

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> read(io, Char)
'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)
```

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> read(io, String)
"JuliaLang is a GitHub organization"
```

source

```
read(filename::AbstractString)
```

Read the entire contents of a file as a `Vector{UInt8}`.

```
read(filename::AbstractString, String)
```

Read the entire contents of a file as a string.

```
read(filename::AbstractString, args...)
```

Open a file and read its contents. `args` is passed to `read`: this is equivalent to `open(io->read(io, args...), filename)`.

source

```
read(s::IO, nb=typemax(Int))
```

Read at most nb bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

source

```
read(s::IOStream, nb::Integer; all=true)
```

Read at most nb bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

If `all` is `true` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `all` is `false`, at most one `read` call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `all` option.

source

```
read(command::Cmd)
```

Run `command` and return the resulting output as an array of bytes.

source

```
read(command::Cmd, String)
```

Run `command` and return the resulting output as a `String`.

source

Base.read! – Function.

```
read!(stream::IO, array::AbstractArray)
read!(filename::AbstractString, array::AbstractArray)
```

Read binary data from an I/O stream or file, filling in `array`.

source

Base.readbytes! – Function.

```
readbytes!(stream::IO, b::AbstractVector{UInt8}, nb=length(b))
```

Read at most `nb` bytes from `stream` into `b`, returning the number of bytes read. The size of `b` will be increased if needed (i.e. if `nb` is greater than `length(b)` and enough bytes could be read), but it will never be decreased.

source

```
readbytes!(stream::IOStream, b::AbstractVector{UInt8}, nb=length(b); all::Bool=true)
```

Read at most `nb` bytes from `stream` into `b`, returning the number of bytes read. The size of `b` will be increased if needed (i.e. if `nb` is greater than `length(b)` and enough bytes could be read), but it will never be decreased.

If `all` is `true` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `all` is `false`, at most one `read` call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `all` option.

source

Base.unsafe_read – Function.

```
unsafe_read(io::IO, ref, nbytes::UInt)
```

Copy `nbytes` from the `IO` stream object into `ref` (converted to a pointer).

It is recommended that subtypes `T<:IO` override the following method signature to provide more efficient implementations: `unsafe_read(s::T, p::Ptr{UInt8}, n::UInt)`

source

Base.unsafe_write – Function.

```
unsafe_write(io::IO, ref, nbytes::UInt)
```

Copy `nbytes` from `ref` (converted to a pointer) into the `IO` object.

It is recommended that subtypes `T<:IO` override the following method signature to provide more efficient implementations: `unsafe_write(s::T, p::Ptr{UInt8}, n::UInt)`

source

Base.readeach – Function.

```
readeach(io::IO, T)
```

Return an iterable object yielding `read(io, T)`.

See also `skipchars`, `eachline`, `readuntil`.

> **Julia 1.6**
>
> readeach requires Julia 1.6 or later.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.\n It has many members.\n");

julia> for c in readeach(io, Char)
           c == '\n' && break
           print(c)
       end
JuliaLang is a GitHub organization.
```

source

Base.peek – Function.

```
peek(stream[, T=UInt8])
```

Read and return a value of type T from a stream without advancing the current position in the stream. See also `startswith(stream, char_or_string)`.

Examples

```
julia> b = IOBuffer("julia");

julia> peek(b)
0x6a

julia> position(b)
0

julia> peek(b, Char)
'j': ASCII/Unicode U+006A (category Ll: Letter, lowercase)
```

> **Julia 1.5**
>
> The method which accepts a type requires Julia 1.5 or later.

source

`Base.position` – Function.

```
position(l::Lexer)
```

Returns the current position.

source

```
position(s)
```

Get the current position of a stream.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> position(io)
5

julia> skip(io, 10);

julia> position(io)
15

julia> seekend(io);

julia> position(io)
35
```

source

`Base.seek` – Function.

```
seek(s, pos)
```

Seek a stream to the given position.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> read(io, Char)
'L': ASCII/Unicode U+004C (category Lu: Letter, uppercase)
```

source

`Base.seekstart` – Function.

```
seekstart(s)
```

Seek a stream to its beginning.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> read(io, Char)
'L': ASCII/Unicode U+004C (category Lu: Letter, uppercase)

julia> seekstart(io);

julia> read(io, Char)
'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)
```

source

`Base.seekend` – Function.

```
seekend(s)
```

Seek a stream to its end.

source

`Base.skip` – Function.

```
skip(s, offset)
```

Seek a stream relative to the current position.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> seek(io, 5);

julia> skip(io, 10);

julia> read(io, Char)
'G': ASCII/Unicode U+0047 (category Lu: Letter, uppercase)
```

source

`Base.mark` – Function.

```
mark(s::IO)
```

Add a mark at the current position of stream `s`. Return the marked position.

See also `unmark`, `reset`, `ismarked`.

source

`Base.unmark` – Function.

```
unmark(s::IO)
```

Remove a mark from stream `s`. Return `true` if the stream was marked, `false` otherwise.

See also `mark`, `reset`, `ismarked`.

source

`Base.reset` – Method.

```
reset(s::IO)
```

Reset a stream `s` to a previously marked position, and remove the mark. Return the previously marked position. Throw an error if the stream is not marked.

See also `mark`, `unmark`, `ismarked`.

source

`Base.ismarked` – Function.

```
ismarked(s::IO)
```

Return `true` if stream `s` is marked.

See also `mark`, `unmark`, `reset`.

source

`Base.eof` – Function.

```
eof(stream) -> Bool
```

Test whether an I/O stream is at end-of-file. If the stream is not yet exhausted, this function will block to wait for more data if necessary, and then return `false`. Therefore it is always safe to read one byte after seeing `eof` return `false`. `eof` will return `false` as long as buffered data is still available, even if the remote end of a connection is closed.

Examples

```
julia> b = IOBuffer("my buffer");

julia> eof(b)
false

julia> seekend(b);

julia> eof(b)
true
```

source

Base.isreadonly – Function.

```
isreadonly(io) -> Bool
```

Determine whether a stream is read-only.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> isreadonly(io)
true

julia> io = IOBuffer();

julia> isreadonly(io)
false
```

source

Base.iswritable – Function.

```
iswritable(io) -> Bool
```

Return `false` if the specified IO object is not writable.

Examples

```
julia> open("myfile.txt", "w") do io
           print(io, "Hello world!");
           iswritable(io)
       end
true

julia> open("myfile.txt", "r") do io
           iswritable(io)
       end
false

julia> rm("myfile.txt")
```

source

Base.isreadable – Function.

```
isreadable(io) -> Bool
```

Return `false` if the specified IO object is not readable.

Examples

```
julia> open("myfile.txt", "w") do io
           print(io, "Hello world!");
           isreadable(io)
       end
false

julia> open("myfile.txt", "r") do io
           isreadable(io)
       end
true

julia> rm("myfile.txt")
```

source

Base.isopen – Function.

```
isopen(object) -> Bool
```

Determine whether an object - such as a stream or timer –is not yet closed. Once an object is closed, it will never produce a new event. However, since a closed stream may still have data to read in its buffer, use `eof` to check for the ability to read data. Use the `FileWatching` package to be notified when a stream might be writable or readable.

Examples

```
julia> io = open("my_file.txt", "w+");

julia> isopen(io)
true

julia> close(io)

julia> isopen(io)
false
```

source

Base.fd – Function.

```
fd(stream)
```

Return the file descriptor backing the stream or file. Note that this function only applies to synchronous `File`'s and `IOStream`'s not to any of the asynchronous streams.

source

`Base.redirect_stdio` – Function.

```
redirect_stdio(;stdin=stdin, stderr=stderr, stdout=stdout)
```

Redirect a subset of the streams `stdin`, `stderr`, `stdout`. Each argument must be an `IOStream`, TTY, `Pipe`, socket, or `devnull`.

Julia 1.7

`redirect_stdio` requires Julia 1.7 or later.

source

```
redirect_stdio(f; stdin=nothing, stderr=nothing, stdout=nothing)
```

Redirect a subset of the streams `stdin`, `stderr`, `stdout`, call `f()` and restore each stream.

Possible values for each stream are:

- `nothing` indicating the stream should not be redirected.
- `path::AbstractString` redirecting the stream to the file at `path`.
- `io` an `IOStream`, TTY, `Pipe`, socket, or `devnull`.

Examples

```
julia> redirect_stdio(stdout="stdout.txt", stderr="stderr.txt") do
           print("hello stdout")
           print(stderr, "hello stderr")
       end

julia> read("stdout.txt", String)
"hello stdout"

julia> read("stderr.txt", String)
"hello stderr"
```

Edge cases

It is possible to pass the same argument to `stdout` and `stderr`:

```
julia> redirect_stdio(stdout="log.txt", stderr="log.txt", stdin=devnull) do
    ...
end
```

However it is not supported to pass two distinct descriptors of the same file.

```
julia> io1 = open("same/path", "w")

julia> io2 = open("same/path", "w")

julia> redirect_stdio(f, stdout=io1, stderr=io2) # not supported
```

Also the `stdin` argument may not be the same descriptor as `stdout` or `stderr`.

```
julia> io = open(...)

julia> redirect_stdio(f, stdout=io, stdin=io) # not supported
```

Julia 1.7

`redirect_stdio` requires Julia 1.7 or later.

source

`Base.redirect_stdout` – Function.

```
redirect_stdout([stream]) -> stream
```

Create a pipe to which all C and Julia level `stdout` output will be redirected. Return a stream representing the pipe ends. Data written to `stdout` may now be read from the `rd` end of the pipe.

Note

`stream` must be a compatible objects, such as an `IOStream`, `TTY`, `Pipe`, socket, or `devnull`.

See also `redirect_stdio`.

source

`Base.redirect_stdout` – Method.

```
redirect_stdout(f::Function, stream)
```

Run the function `f` while redirecting `stdout` to `stream`. Upon completion, `stdout` is restored to its prior setting.

source

Base.redirect_stderr – Function.

```
redirect_stderr([stream]) -> stream
```

Like redirect_stdout, but for stderr.

Note

stream must be a compatible objects, such as an IOStream, TTY, Pipe, socket, or devnull.

See also redirect_stdio.

source

Base.redirect_stderr – Method.

```
redirect_stderr(f::Function, stream)
```

Run the function f while redirecting stderr to stream. Upon completion, stderr is restored to its prior setting.

source

Base.redirect_stdin – Function.

```
redirect_stdin([stream]) -> stream
```

Like redirect_stdout, but for stdin. Note that the direction of the stream is reversed.

Note

stream must be a compatible objects, such as an IOStream, TTY, Pipe, socket, or devnull.

See also redirect_stdio.

source

Base.redirect_stdin – Method.

```
redirect_stdin(f::Function, stream)
```

Run the function f while redirecting stdin to stream. Upon completion, stdin is restored to its prior setting.

source

Base.readchomp – Function.

```
readchomp(x)
```

Read the entirety of x as a string and remove a single trailing newline if there is one. Equivalent to chomp(read(x, String)).

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> readchomp("my_file.txt")
"JuliaLang is a GitHub organization.\nIt has many members."

julia> rm("my_file.txt");
```

source

Base.truncate – Function.

```
truncate(file, n)
```

Resize the file or buffer given by the first argument to exactly n bytes, filling previously unallocated space with '\0' if the file or buffer is grown.

Examples

```
julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.")
35

julia> truncate(io, 15)
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false, size=15,
↪  maxsize=Inf, ptr=16, mark=-1)

julia> String(take!(io))
"JuliaLang is a "

julia> io = IOBuffer();

julia> write(io, "JuliaLang is a GitHub organization.");

julia> truncate(io, 40);

julia> String(take!(io))
"JuliaLang is a GitHub organization.\0\0\0\0\0"
```

source

Base.skipchars – Function.

```
skipchars(predicate, io::IO; linecomment=nothing)
```

Advance the stream `io` such that the next-read character will be the first remaining for which `predicate` returns `false`. If the keyword argument `linecomment` is specified, all characters from that character until the start of the next line are ignored.

Examples

```
julia> buf = IOBuffer("    text")
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false, size=8,
↪  maxsize=Inf, ptr=1, mark=-1)

julia> skipchars(isspace, buf)
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false, size=8,
↪  maxsize=Inf, ptr=5, mark=-1)

julia> String(readavailable(buf))
"text"
```

source

`Base.countlines` – Function.

```
countlines(io::IO; eol::AbstractChar = '\n')
countlines(filename::AbstractString; eol::AbstractChar = '\n')
```

Read `io` until the end of the stream/file and count the number of lines. To specify a file pass the filename as the first argument. EOL markers other than `'\n'` are supported by passing them as the second argument. The last non-empty line of `io` is counted even if it does not end with the EOL, matching the length returned by `eachline` and `readlines`.

To count lines of a `String`, `countlines(IOBuffer(str))` can be used.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization.\n");

julia> countlines(io)
1

julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> countlines(io)
1

julia> eof(io) # counting lines moves the file pointer
true

julia> io = IOBuffer("JuliaLang is a GitHub organization.");

julia> countlines(io, eol = '.')
1
```

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\n")
36

julia> countlines("my_file.txt")
1

julia> countlines("my_file.txt", eol = 'n')
4

julia> rm("my_file.txt")
```

source

Base.PipeBuffer – Function.

```
PipeBuffer(data::Vector{UInt8}=UInt8[]; maxsize::Integer = typemax(Int))
```

An `IOBuffer` that allows reading and performs writes by appending. Seeking and truncating are not supported. See `IOBuffer` for the available constructors. If `data` is given, creates a `PipeBuffer` to operate on a data vector, optionally specifying a size beyond which the underlying `Array` may not be grown.

source

Base.readavailable – Function.

```
readavailable(stream)
```

Read available buffered data from a stream. Actual I/O is performed only if no data has already been buffered. The result is a `Vector{UInt8}`.

> **Warning**
>
> The amount of data returned is implementation-dependent; for example it can depend on the internal choice of buffer size. Other functions such as `read` should generally be used instead.

source

Base.IOContext – Type.

```
IOContext
```

`IOContext` provides a mechanism for passing output configuration settings among `show` methods.

In short, it is an immutable dictionary that is a subclass of `IO`. It supports standard dictionary operations such as `getindex`, and can also be used as an I/O stream.

source

Base.IOContext – Method.

```
IOContext(io::IO, KV::Pair...)
```

Create an `IOContext` that wraps a given stream, adding the specified `key=>value` pairs to the properties of that stream (note that `io` can itself be an `IOContext`).

- use `(key => value) in io` to see if this particular combination is in the properties set
- use `get(io, key, default)` to retrieve the most recent value for a particular key

The following properties are in common use:

- `:compact`: Boolean specifying that values should be printed more compactly, e.g. that numbers should be printed with fewer digits. This is set when printing array elements. `:compact` output should not contain line breaks.
- `:limit`: Boolean specifying that containers should be truncated, e.g. showing `⋯` in place of most elements.
- `:displaysize`: A `Tuple{Int,Int}` giving the size in rows and columns to use for text output. This can be used to override the display size for called functions, but to get the size of the screen use the `displaysize` function.
- `:typeinfo`: a Type characterizing the information already printed concerning the type of the object about to be displayed. This is mainly useful when displaying a collection of objects of the same type, so that redundant type information can be avoided (e.g. `[Float16(0)]` can be shown as "Float16[0.0]" instead of "Float16[Float16(0.0)]" : while displaying the elements of the array, the `:typeinfo` property will be set to `Float16`).
- `:color`: Boolean specifying whether ANSI color/escape codes are supported/expected. By default, this is determined by whether `io` is a compatible terminal and by any `--color` command-line flag when `julia` was launched.

Examples

```
julia> io = IOBuffer();

julia> printstyled(IOContext(io, :color => true), "string", color=:red)

julia> String(take!(io))
"\e[31mstring\e[39m"

julia> printstyled(io, "string", color=:red)

julia> String(take!(io))
"string"
```

```
julia> print(IOContext(stdout, :compact => false), 1.12341234)
1.12341234
julia> print(IOContext(stdout, :compact => true), 1.12341234)
1.12341
```

```
julia> function f(io::IO)
           if get(io, :short, false)
               print(io, "short")
           else
               print(io, "loooooong")
           end
       end
f (generic function with 1 method)

julia> f(stdout)
loooooong
julia> f(IOContext(stdout, :short => true))
short
```

source

Base.IOContext – Method.

```
IOContext(io::IO, context::IOContext)
```

Create an IOContext that wraps an alternate IO but inherits the properties of context.

source

52.2 文本 I/O

Base.show – Method.

```
show([io::IO = stdout], x)
```

Write a text representation of a value x to the output stream io. New types T should overload show(io::IO, x::T). The representation used by show generally includes Julia-specific formatting and type information, and should be parseable Julia code when possible.

repr returns the output of show as a string.

For a more verbose human-readable text output for objects of type T, define show(io::IO, ::MIME"text/plain", ::T) in addition. Checking the :compact IOContext key (often checked as get(io, :compact, false)::Bool) of io in such methods is recommended, since some containers show their elements by calling this method with :compact => true.

See also print, which writes un-decorated representations.

Examples

```
julia> show("Hello World!")
"Hello World!"
julia> print("Hello World!")
Hello World!
```

source

Base.summary – Function.

```
summary(io::IO, x)
str = summary(x)
```

Print to a stream io, or return a string str, giving a brief description of a value. By default returns string(typeof(x)), e.g. Int64.

For arrays, returns a string of size and type info, e.g. 10-element Array{Int64,1}.

Examples

```
julia> summary(1)
"Int64"

julia> summary(zeros(2))
"2-element Vector{Float64}"
```

source

Base.print – Function.

```
print([io::IO], xs...)
```

Write to io (or to the default output stream stdout if io is not given) a canonical (un-decorated) text representation. The representation used by print includes minimal formatting and tries to avoid Julia-specific details.

print falls back to calling show, so most types should just define show. Define print if your type has a separate "plain" representation. For example, show displays strings with quotes, and print displays strings without quotes.

See also println, string, printstyled.

Examples

```
julia> print("Hello World!")
Hello World!
julia> io = IOBuffer();

julia> print(io, "Hello", ' ', :World!)

julia> String(take!(io))
"Hello World!"
```

source

Base.println – Function.

```
println([io::IO], xs...)
```

Print (using `print`) xs to io followed by a newline. If io is not supplied, prints to the default output stream `stdout`.

See also `printstyled` to add colors etc.

Examples

```
julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello", ',', " world.")

julia> String(take!(io))
"Hello, world.\n"
```

source

Base.printstyled – Function.

```
printstyled([io], xs...; bold::Bool=false, italic::Bool=false, underline::Bool=false,
↪  blink::Bool=false, reverse::Bool=false, hidden::Bool=false,
↪  color::Union{Symbol,Int}=:normal)
```

Print xs in a color specified as a symbol or integer, optionally in bold.

Keyword `color` may take any of the values `:normal`, `:italic`, `:default`, `:bold`, `:black`, `:blink`, `:blue`, `:cyan`, `:green`, `:hidden`, `:light_black`, `:light_blue`, `:light_cyan`, `:light_green`, `:light_magenta`, `:light_red`, `:light_white`, `:light_yellow`, `:magenta`, `:nothing`, `:red`, `:reverse`, `:underline`, `:white`, or `:yellow` or an integer between 0 and 255 inclusive. Note that not all terminals support 256 colors.

Keywords `bold=true`, `italic=true`, `underline=true`, `blink=true` are self-explanatory. Keyword `reverse=true` prints with foreground and background colors exchanged, and `hidden=true` should be invisible in the terminal but can still be copied. These properties can be used in any combination.

See also `print`, `println`, `show`.

Note

Not all terminals support italic output. Some terminals interpret italic as reverse or blink.

Julia 1.7

Keywords except `color` and `bold` were added in Julia 1.7.

Julia 1.10

Support for italic output was added in Julia 1.10.

source

Base.sprint – Function.

```
sprint(f::Function, args...; context=nothing, sizehint=0)
```

Call the given function with an I/O stream and the supplied extra arguments. Everything written to this I/O stream is returned as a string. `context` can be an `IOContext` whose properties will be used, a `Pair` specifying a property and its value, or a tuple of `Pair` specifying multiple properties and their values. `sizehint` suggests the capacity of the buffer (in bytes).

The optional keyword argument `context` can be set to a `:key=>value` pair, a tuple of `:key=>value` pairs, or an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `f`. The optional `sizehint` is a suggested size (in bytes) to allocate for the buffer used to write the string.

> **Julia 1.7**
>
> Passing a tuple to keyword `context` requires Julia 1.7 or later.

Examples

```
julia> sprint(show, 66.66666; context=:compact => true)
"66.6667"

julia> sprint(showerror, BoundsError([1], 100))
"BoundsError: attempt to access 1-element Vector{Int64} at index [100]"
```

source

Base.showerror – Function.

```
showerror(io, e)
```

Show a descriptive representation of an exception object `e`. This method is used to display the exception after a call to `throw`.

Examples

```
julia> struct MyException <: Exception
           msg::String
       end

julia> function Base.showerror(io::IO, err::MyException)
           print(io, "MyException: ")
           print(io, err.msg)
       end

julia> err = MyException("test exception")
MyException("test exception")
```

```
julia> sprint(showerror, err)
"MyException: test exception"

julia> throw(MyException("test exception"))
ERROR: MyException: test exception
```

source

`Base.dump` – Function.

```
dump(x; maxdepth=8)
```

Show every part of the representation of a value. The depth of the output is truncated at `maxdepth`.

Examples

```
julia> struct MyStruct
           x
           y
       end

julia> x = MyStruct(1, (2,3));

julia> dump(x)
MyStruct
  x: Int64 1
  y: Tuple{Int64, Int64}
    1: Int64 2
    2: Int64 3

julia> dump(x; maxdepth = 1)
MyStruct
  x: Int64 1
  y: Tuple{Int64, Int64}
```

source

`Base.Meta.@dump` – Macro.

```
@dump expr
```

Show every part of the representation of the given expression. Equivalent to `dump(:(expr))`.

source

`Base.readline` – Function.

```
readline(io::IO=stdin; keep::Bool=false)
readline(filename::AbstractString; keep::Bool=false)
```

Read a single line of text from the given I/O stream or file (defaults to stdin). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with '\n' or "\r\n" or the end of an input stream. When keep is false (as it is by default), these trailing newline characters are removed from the line before it is returned. When keep is true, they are returned as part of the line.

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> readline("my_file.txt")
"JuliaLang is a GitHub organization."

julia> readline("my_file.txt", keep=true)
"JuliaLang is a GitHub organization.\n"

julia> rm("my_file.txt")
```

```
julia> print("Enter your name: ")
Enter your name:

julia> your_name = readline()
Logan
"Logan"
```

source

Base.readuntil – Function.

```
readuntil(stream::IO, delim; keep::Bool = false)
readuntil(filename::AbstractString, delim; keep::Bool = false)
```

Read a string from an I/O stream or a file, up to the given delimiter. The delimiter can be a UInt8, AbstractChar, string, or vector. Keyword argument keep controls whether the delimiter is included in the result. The text is assumed to be encoded in UTF-8.

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> readuntil("my_file.txt", 'L')
"Julia"

julia> readuntil("my_file.txt", '.', keep = true)
"JuliaLang is a GitHub organization."

julia> rm("my_file.txt")
```

source

Base.readlines – Function.

```
readlines(io::IO=stdin; keep::Bool=false)
readlines(filename::AbstractString; keep::Bool=false)
```

Read all lines of an I/O stream or a file as a vector of strings. Behavior is equivalent to saving the result of reading `readline` repeatedly with the same arguments and saving the resulting lines as a vector of strings. See also `eachline` to iterate over the lines without reading them all at once.

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt has many members.\n");

julia> readlines("my_file.txt")
2-element Vector{String}:
 "JuliaLang is a GitHub organization."
 "It has many members."

julia> readlines("my_file.txt", keep=true)
2-element Vector{String}:
 "JuliaLang is a GitHub organization.\n"
 "It has many members.\n"

julia> rm("my_file.txt")
```

source

Base.eachline – Function.

```
eachline(io::IO=stdin; keep::Bool=false)
eachline(filename::AbstractString; keep::Bool=false)
```

Create an iterable EachLine object that will yield each line from an I/O stream or a file. Iteration calls `readline` on the stream argument repeatedly with keep passed through, determining whether trailing end-of-line characters are retained. When called with a file name, the file is opened once at the beginning of iteration and closed at the end. If iteration is interrupted, the file will be closed when the EachLine object is garbage collected.

To iterate over each line of a `String`, `eachline(IOBuffer(str))` can be used.

`Iterators.reverse` can be used on an EachLine object to read the lines in reverse order (for files, buffers, and other I/O streams supporting `seek`), and `first` or `last` can be used to extract the initial or final lines, respectively.

Examples

```
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\n It has many members.\n");

julia> for line in eachline("my_file.txt")
           print(line)
       end
JuliaLang is a GitHub organization. It has many members.

julia> rm("my_file.txt");
```

> **Julia 1.8**
>
> Julia 1.8 is required to use `Iterators.reverse` or `last` with `eachline` iterators.

source

`Base.displaysize` – Function.

```
displaysize([io::IO]) -> (lines, columns)
```

Return the nominal size of the screen that may be used for rendering output to this `IO` object. If no input is provided, the environment variables `LINES` and `COLUMNS` are read. If those are not set, a default size of `(24, 80)` is returned.

Examples

```
julia> withenv("LINES" => 30, "COLUMNS" => 100) do
           displaysize()
       end
(30, 100)
```

To get your TTY size,

```
julia> displaysize(stdout)
(34, 147)
```

source

52.3 多媒体 I/O

就像文本输出用 `print` 实现，用户自定义类型可以通过重载 `show` 来指定其文本化表示，Julia 提供了一个应用于富多媒体输出的标准化机制（例如图片、格式化文本、甚至音频和视频），由以下三部分组成：

- 函数 `display(x)` 来请求一个 Julia 对象 x 最丰富的多媒体展示，并以纯文本作为后备模式。
- 重载 `show` 允许指定用户自定义类型的任意多媒体表现形式（以标准 MIME 类型为键值）。
- Multimedia-capable display backends may be registered by subclassing a generic `AbstractDisplay` type 并通过 `pushdisplay` 将其压进显示后端的栈中。

基础 Julia 运行环境只提供纯文本显示，但是更富的显示可以通过加载外部模块或者使用图形化 Julia 环境（比如基于 IPython 的 IJulia notebook）来实现。

`Base.Multimedia.AbstractDisplay` – Type.

```
AbstractDisplay
```

Abstract supertype for rich display output devices. `TextDisplay` is a subtype of this.

source

`Base.Multimedia.display` – Function.

```
display(x)
display(d::AbstractDisplay, x)
display(mime, x)
display(d::AbstractDisplay, mime, x)
```

Display `x` using the topmost applicable display in the display stack, typically using the richest supported multimedia output for `x`, with plain-text `stdout` output as a fallback. The `display(d, x)` variant attempts to display `x` on the given display `d` only, throwing a `MethodError` if `d` cannot display objects of this type.

In general, you cannot assume that `display` output goes to `stdout` (unlike `print(x)` or `show(x)`). For example, `display(x)` may open up a separate window with an image. `display(x)` means "show x in the best way you can for the current output device(s)." If you want REPL-like text output that is guaranteed to go to `stdout`, use `show(stdout, "text/plain", x)` instead.

There are also two variants with a `mime` argument (a MIME type string, such as `"image/png"`), which attempt to display `x` using the requested MIME type *only*, throwing a `MethodError` if this type is not supported by either the display(s) or by `x`. With these variants, one can also supply the "raw" data in the requested MIME type by passing `x::AbstractString` (for MIME types with text-based storage, such as text/html or application/postscript) or `x::Vector{UInt8}` (for binary MIME types).

To customize how instances of a type are displayed, overload `show` rather than `display`, as explained in the manual section on custom pretty-printing.

source

`Base.Multimedia.redisplay` – Function.

```
redisplay(x)
redisplay(d::AbstractDisplay, x)
redisplay(mime, x)
redisplay(d::AbstractDisplay, mime, x)
```

By default, the `redisplay` functions simply call `display`. However, some display backends may override `redisplay` to modify an existing display of `x` (if any). Using `redisplay` is also a hint to the backend that `x` may be redisplayed several times, and the backend may choose to defer the display until (for example) the next interactive prompt.

source

`Base.Multimedia.displayable` – Function.

```
displayable(mime) -> Bool
displayable(d::AbstractDisplay, mime) -> Bool
```

Return a boolean value indicating whether the given `mime` type (string) is displayable by any of the displays in the current display stack, or specifically by the display `d` in the second variant.

source

Base.show – Method.

```
show(io::IO, mime, x)
```

The `display` functions ultimately call show in order to write an object x as a given mime type to a given I/O stream io (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-defined type T, it is only necessary to define a new show method for T, via: `show(io, ::MIME"mime", x::T) = ...`, where mime is a MIME-type string and the function body calls `write` (or similar) to write that representation of x to io. (Note that the `MIME""` notation only supports literal strings; to construct MIME types in a more flexible manner use `MIME{Symbol("")}`.)

For example, if you define a MyImage type and know how to write it to a PNG file, you could define a function `show(io, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable AbstractDisplay (such as IJulia). As usual, be sure to `import Base.show` in order to add new methods to the built-in Julia function show.

Technically, the `MIME"mime"` macro defines a singleton type for the given mime string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The default MIME type is `MIME"text/plain"`. There is a fallback definition for text/plain output that calls show with 2 arguments, so it is not always necessary to add a method for that case. If a type benefits from custom human-readable output though, `show(::IO, ::MIME"text/plain", ::T)` should be defined. For example, the Day type uses `1 day` as the output for the text/plain MIME type, and `Day(1)` as the output of 2-argument show.

Examples

```
julia> struct Day
           n::Int
       end

julia> Base.show(io::IO, ::MIME"text/plain", d::Day) = print(io, d.n, " day")

julia> Day(1)
1 day
```

Container types generally implement 3-argument show by calling `show(io, MIME"text/plain"(), x)` for elements x, with `:compact => true` set in an `IOContext` passed as the first argument.

source

Base.Multimedia.showable – Function.

```
showable(mime, x)
```

Return a boolean value indicating whether or not the object x can be written as the given mime type.

(By default, this is determined automatically by the existence of the corresponding `show` method for `typeof(x)`. Some types provide custom showable methods; for example, if the available MIME formats depend on the *value* of x.)

Examples

```
julia> showable(MIME("text/plain"), rand(5))
true

julia> showable("image/png", rand(5))
false
```

source

Base.repr – Method.

```
repr(mime, x; context=nothing)
```

Return an AbstractString or Vector{UInt8} containing the representation of x in the requested mime type, as written by show(io, mime, x) (throwing a MethodError if no appropriate show is available). An AbstractString is returned for MIME types with textual representations (such as "text/html" or "application/postscript"), whereas binary data is returned as Vector{UInt8}. (The function istextmime(mime) returns whether or not Julia treats a given mime type as text.)

The optional keyword argument context can be set to :key=>value pair or an IO or IOContext object whose attributes are used for the I/O stream passed to show.

As a special case, if x is an AbstractString (for textual MIME types) or a Vector{UInt8} (for binary MIME types), the repr function assumes that x is already in the requested mime format and simply returns x. This special case does not apply to the "text/plain" MIME type. This is useful so that raw data can be passed to display(m::MIME, x).

In particular, repr("text/plain", x) is typically a "pretty-printed" version of x designed for human consumption. See also repr(x) to instead return a string corresponding to show(x) that may be closer to how the value of x would be entered in Julia.

Examples

```
julia> A = [1 2; 3 4];

julia> repr("text/plain", A)
"2×2 Matrix{Int64}:\n 1  2\n 3  4"
```

source

Base.Multimedia.MIME – Type.

```
MIME
```

A type representing a standard internet data format. "MIME" stands for "Multipurpose Internet Mail Extensions", since the standard was originally used to describe multimedia attachments to email messages.

A MIME object can be passed as the second argument to show to request output in that format.

Examples

```
julia> show(stdout, MIME("text/plain"), "hi")
"hi"
```

source

Base.Multimedia.@MIME_str – Macro.

```
@MIME_str
```

A convenience macro for writing MIME types, typically used when adding methods to show. For example the syntax show(io::IO, ::MIME"text/html", x::MyType) = ... could be used to define how to write an HTML representation of MyType.

source

如上面提到的，用户可以定义新的显示后端。例如，可以在窗口显示 PNG 图片的模块可以在 Julia 中注册这个能力，以便为有 PNG 表示的类型调用 display(x) 时可以在模块窗口中自动显示图片。

In order to define a new display backend, one should first create a subtype D of the abstract class AbstractDisplay. Then, for each MIME type (mime string) that can be displayed on D, one should define a function display(d::D, ::MIME"mime", x) = ... that displays x as that MIME type, usually by calling show(io, mime, x) or repr(io, mime, x). A MethodError should be thrown if x cannot be displayed as that MIME type; this is automatic if one calls show or repr. Finally, one should define a function display(d::D, x) that queries showable(mime, x) for the mime types supported by D and displays the "best" one; a MethodError should be thrown if no supported MIME types are found for x. Similarly, some subtypes may wish to override redisplay(d::D, ...). (Again, one should import Base.display to add new methods to display.) The return values of these functions are up to the implementation (since in some cases it may be useful to return a display "handle" of some type). The display functions for D can then be called directly, but they can also be invoked automatically from display(x) simply by pushing a new display onto the display-backend stack with:

Base.Multimedia.pushdisplay – Function.

```
pushdisplay(d::AbstractDisplay)
```

Pushes a new display d on top of the global display-backend stack. Calling display(x) or display(mime, x) will display x on the topmost compatible backend in the stack (i.e., the topmost backend that does not throw a MethodError).

source

Base.Multimedia.popdisplay – Function.

```
popdisplay()
popdisplay(d::AbstractDisplay)
```

Pop the topmost backend off of the display-backend stack, or the topmost copy of d in the second variant.

source

Base.Multimedia.TextDisplay – Type.

```
TextDisplay(io::IO)
```

Return a `TextDisplay <: AbstractDisplay`, which displays any object as the text/plain MIME type (by default), writing the text representation to the given I/O stream. (This is how objects are printed in the Julia REPL.)

source

`Base.Multimedia.istextmime` – Function.

```
istextmime(m::MIME)
```

Determine whether a MIME type is text data. MIME types are assumed to be binary data except for a set of types known to be text data (possibly Unicode).

Examples

```
julia> istextmime(MIME("text/plain"))
true

julia> istextmime(MIME("image/png"))
false
```

source

52.4 网络 I/O

`Base.bytesavailable` – Function.

```
bytesavailable(io)
```

Return the number of bytes available for reading before a read from this stream or buffer will block.

Examples

```
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> bytesavailable(io)
34
```

source

`Base.ntoh` – Function.

```
ntoh(x)
```

Convert the endianness of a value from Network byte order (big-endian) to that used by the Host.

source

Base.hton – Function.

```
hton(x)
```

Convert the endianness of a value from that used by the Host to Network byte order (big-endian).

source

Base.ltoh – Function.

```
ltoh(x)
```

Convert the endianness of a value from Little-endian to that used by the Host.

source

Base.htol – Function.

```
htol(x)
```

Convert the endianness of a value from that used by the Host to Little-endian.

source

Base.ENDIAN_BOM – Constant.

```
ENDIAN_BOM
```

The 32-bit byte-order-mark indicates the native byte order of the host machine. Little-endian machines will contain the value `0x04030201`. Big-endian machines will contain the value `0x01020304`.

source

Chapter 53

运算符与记号

数学符号与函数的扩展文档在 这里.

符号	含义
@	the at-sign marks a macro invocation; optionally followed by an argument list
!	感叹号是一个表示逻辑否定的前缀算符
a!	function names that end with an exclamation mark modify one or more of their arguments by convention
#	the number sign (or hash or pound) character begins single line comments
#=	when followed by an equals sign, it begins a multi-line comment (these are nestable)
=#	end a multi-line comment by immediately preceding the number sign with an equals sign
$	the dollar sign is used for string and expression interpolation
%	the percent symbol is the remainder operator
^	the caret is the exponentiation operator
&	single ampersand is bitwise and
&&	double ampersands is short-circuiting boolean and
|	single pipe character is bitwise or
||	double pipe characters is short-circuiting boolean or
⊻	the unicode xor character is bitwise exclusive or
~	the tilde is an operator for bitwise not
'	a trailing apostrophe is the adjoint (that is, the complex transpose) operator A^H
*	the asterisk is used for multiplication, including matrix multiplication and string concatenation
/	forward slash divides the argument on its left by the one on its right
\	backslash operator divides the argument on its right by the one on its left, commonly used to solve matrix equations
()	parentheses with no arguments constructs an empty Tuple
(a,...)	parentheses with comma-separated arguments constructs a tuple containing its arguments
(a=1,...)	parentheses with comma-separated assignments constructs a NamedTuple
(x;y)	parentheses can also be used to group one or more semicolon separated expressions
a[]	array indexing (calling getindex or setindex!)
[,]	vector literal constructor (calling vect)
[;]	vertical concatenation (calling vcat or hvcat)
[ ]	with space-separated expressions, horizontal concatenation (calling hcat or hvcat)
T{ }	curly braces following a type list that type's parameters
{}	curly braces can also be used to group multiple where expressions in function declarations
;	semicolons separate statements, begin a list of keyword arguments in function declarations or calls, or are used to separate array literals for vertical concatenation
,	commas separate function arguments or tuple or array components
?	the question mark delimits the ternary conditional operator (used like: `conditional ? if_true : if_false`)
" "	the single double-quote character delimits String literals
""" """	three double-quote characters delimits string literals that may contain " and ignore leading indentation
' '	the single-quote character delimits Char (that is, character) literals
` `	the backtick character delimits external process (Cmd) literals
A...	triple periods are a postfix operator that "splat" their arguments' contents into many arguments of a function call or declare a varargs function that "slurps" up many arguments into a single tuple
a.b	single periods access named fields in objects/modules (calling getproperty or setproperty!)
f.()	periods may also prefix parentheses (like f.(...)) or infix operators (like .+) to perform the function element-wise (calling broadcast)
a:b	colons (:) used as a binary infix operator construct a range from a to b (inclusive) with fixed step size 1
a:s:b	colons (:) used as a ternary infix operator construct a range from a to b (inclusive) with step size s
:	when used by themselves, Colons represent all indices within a dimension, frequently combined with indexing
::	double-colons represent a type annotation or typeassert, depending on context, frequently used when declaring function arguments
:( )	quoted expression

Chapter 54

排序及相关函数

Julia 拥有广泛而灵活的应用程序接口，可用于对已排序的数组值进行排序和交互。默认情况下，Julia 会选择合理的算法并按升序排序：

```
julia> sort([2,3,1])
3-element Vector{Int64}:
 1
 2
 3
```

You can sort in reverse order as well:

```
julia> sort([2,3,1], rev=true)
3-element Vector{Int64}:
 3
 2
 1
```

`sort` constructs a sorted copy leaving its input unchanged. Use the ”bang” version of the sort function to mutate an existing array:

```
julia> a = [2,3,1];

julia> sort!(a);

julia> a
3-element Vector{Int64}:
 1
 2
 3
```

Instead of directly sorting an array, you can compute a permutation of the array’s indices that puts the array into sorted order:

```
julia> v = randn(5)
5-element Array{Float64,1}:
  0.297288
```

```
  0.382396
 -0.597634
 -0.0104452
 -0.839027

julia> p = sortperm(v)
5-element Array{Int64,1}:
 5
 3
 4
 1
 2

julia> v[p]
5-element Array{Float64,1}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

Arrays can be sorted according to an arbitrary transformation of their values:

```
julia> sort(v, by=abs)
5-element Array{Float64,1}:
 -0.0104452
  0.297288
  0.382396
 -0.597634
 -0.839027
```

或者通过转换来进行逆序排序

```
julia> sort(v, by=abs, rev=true)
5-element Array{Float64,1}:
 -0.839027
 -0.597634
  0.382396
  0.297288
 -0.0104452
```

如有必要，可以选择排序算法：

```
julia> sort(v, alg=InsertionSort)
5-element Array{Float64,1}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

All the sorting and order related functions rely on a "less than" relation defining a strict weak order on the values to be manipulated. The `isless` function is invoked by default, but the relation can be specified via the `lt` keyword, a function that takes two array elements and returns `true` if and only if the first argument is "less than" the second. See `sort!` and Alternate Orderings for more information.

54.1 排序函数

`Base.sort!` – Function.

```
sort!(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward)
```

Sort the vector v in place. A stable algorithm is used by default: the ordering of elements that compare equal is preserved. A specific algorithm can be selected via the `alg` keyword (see Sorting Algorithms for available algorithms).

Elements are first transformed with the function by and then compared according to either the function `lt` or the ordering `order`. Finally, the resulting order is reversed if `rev=true` (this preserves forward stability: elements that compare equal are not reversed). The current implemention applies the by transformation before each comparison rather than once per element.

Passing an `lt` other than `isless` along with an `order` other than `Base.Order.Forward` or `Base.Order.Reverse` is not permitted, otherwise all options are independent and can be used together in all possible combinations. Note that `order` can also include a "by" transformation, in which case it is applied after that defined with the by keyword. For more information on `order` values see the documentation on Alternate Orderings.

Relations between two elements are defined as follows (with "less" and "greater" exchanged when `rev=true`):

- x is less than y if `lt(by(x), by(y))` (or `Base.Order.lt(order, by(x), by(y))`) yields true.
- x is greater than y if y is less than x.
- x and y are equivalent if neither is less than the other ("incomparable" is sometimes used as a synonym for "equivalent").

The result of `sort!` is sorted in the sense that every element is greater than or equivalent to the previous one.

The `lt` function must define a strict weak order, that is, it must be

- irreflexive: `lt(x, x)` always yields `false`,
- asymmetric: if `lt(x, y)` yields `true` then `lt(y, x)` yields `false`,
- transitive: `lt(x, y) && lt(y, z)` implies `lt(x, z)`,
- transitive in equivalence: `!lt(x, y) && !lt(y, x)` and `!lt(y, z) && !lt(z, y)` together imply `!lt(x, z) && !lt(z, x)`. In words: if x and y are equivalent and y and z are equivalent then x and z must be equivalent.

For example < is a valid `lt` function for `Int` values but ≤ is not: it violates irreflexivity. For `Float64` values even < is invalid as it violates the fourth condition: `1.0` and `NaN` are equivalent and so are `NaN` and `2.0` but `1.0` and `2.0` are not equivalent.

See also `sort`, `sortperm`, `sortslices`, `partialsort!`, `partialsortperm`, `issorted`, `searchsorted`, `insorted`, `Base.Order.ord`.

Examples

```
julia> v = [3, 1, 2]; sort!(v); v
3-element Vector{Int64}:
 1
 2
 3

julia> v = [3, 1, 2]; sort!(v, rev = true); v
3-element Vector{Int64}:
 3
 2
 1

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[1]); v
3-element Vector{Tuple{Int64, String}}:
 (1, "c")
 (2, "b")
 (3, "a")

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[2]); v
3-element Vector{Tuple{Int64, String}}:
 (3, "a")
 (2, "b")
 (1, "c")

julia> sort(0:3, by=x->x-2, order=Base.Order.By(abs)) # same as sort(0:3, by=abs(x->x-2))
4-element Vector{Int64}:
 2
 1
 3
 0

julia> sort([2, NaN, 1, NaN, 3]) # correct sort with default lt=isless
5-element Vector{Float64}:
   1.0
   2.0
   3.0
 NaN
 NaN

julia> sort([2, NaN, 1, NaN, 3], lt=<) # wrong sort due to invalid lt. This behavior is
↪  undefined.
5-element Vector{Float64}:
   2.0
 NaN
   1.0
 NaN
   3.0
```

source

```
sort!(A; dims::Integer, alg::Algorithm=defalg(A), lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward)
```

Sort the multidimensional array A along dimension dims. See the one-dimensional version of sort! for a

description of possible keyword arguments.

To sort slices of an array, refer to `sortslices`.

Julia 1.1

This function requires at least Julia 1.1.

Examples

```
julia> A = [4 3; 1 2]
2×2 Matrix{Int64}:
 4  3
 1  2

julia> sort!(A, dims = 1); A
2×2 Matrix{Int64}:
 1  2
 4  3

julia> sort!(A, dims = 2); A
2×2 Matrix{Int64}:
 1  2
 3  4
```

source

Base.sort – Function.

```
sort(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward)
```

Variant of `sort!` that returns a sorted copy of v leaving v itself unmodified.

Examples

```
julia> v = [3, 1, 2];

julia> sort(v)
3-element Vector{Int64}:
 1
 2
 3

julia> v
3-element Vector{Int64}:
 3
 1
 2
```

source

```
sort(A; dims::Integer, alg::Algorithm=defalg(A), lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward)
```

Sort a multidimensional array A along the given dimension. See `sort!` for a description of possible keyword arguments.

To sort slices of an array, refer to `sortslices`.

Examples

```
julia> A = [4 3; 1 2]
2×2 Matrix{Int64}:
 4  3
 1  2

julia> sort(A, dims = 1)
2×2 Matrix{Int64}:
 1  2
 4  3

julia> sort(A, dims = 2)
2×2 Matrix{Int64}:
 3  4
 1  2
```

source

`Base.sortperm` – Function.

```
sortperm(A; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward, [dims::Integer])
```

Return a permutation vector or array `I` that puts `A[I]` in sorted order along the given dimension. If `A` has more than one dimension, then the `dims` keyword argument must be specified. The order is specified using the same keywords as `sort!`. The permutation is guaranteed to be stable even if the sorting algorithm is unstable: the indices of equal elements will appear in ascending order.

See also `sortperm!`, `partialsortperm`, `invperm`, `indexin`. To sort slices of an array, refer to `sortslices`.

Julia 1.9

The method accepting `dims` requires at least Julia 1.9.

Examples

```
julia> v = [3, 1, 2];

julia> p = sortperm(v)
3-element Vector{Int64}:
 2
 3
```

```
 1

julia> v[p]
3-element Vector{Int64}:
 1
 2
 3

julia> A = [8 7; 5 6]
2×2 Matrix{Int64}:
 8  7
 5  6

julia> sortperm(A, dims = 1)
2×2 Matrix{Int64}:
 2  4
 1  3

julia> sortperm(A, dims = 2)
2×2 Matrix{Int64}:
 3  1
 2  4
```

source

`Base.Sort.InsertionSort` – Constant.

```
InsertionSort
```

Use the insertion sort algorithm.

Insertion sort traverses the collection one element at a time, inserting each element into its correct, sorted position in the output vector.

Characteristics:

- *stable*: preserves the ordering of elements that compare equal

(e.g. ”a” and ”A” in a sort of letters that ignores case).

- *in-place* in memory.
- *quadratic performance* in the number of elements to be sorted:

it is well-suited to small collections but should not be used for large ones.

source

`Base.Sort.MergeSort` – Constant.

```
MergeSort
```

Indicate that a sorting function should use the merge sort algorithm. Merge sort divides the collection into subcollections and repeatedly merges them, sorting each subcollection at each step, until the entire collection has been recombined in sorted form.

Characteristics:

- *stable*: preserves the ordering of elements that compare equal (e.g. "a" and "A" in a sort of letters that ignores case).
- *not in-place* in memory.
- *divide-and-conquer* sort strategy.
- *good performance* for large collections but typically not quite as fast as `QuickSort`.

source

`Base.Sort.QuickSort` – Constant.

```
QuickSort
```

Indicate that a sorting function should use the quick sort algorithm, which is *not* stable.

Characteristics:

- *not stable*: does not preserve the ordering of elements that compare equal (e.g. "a" and "A" in a sort of letters that ignores case).
- *in-place* in memory.
- *divide-and-conquer*: sort strategy similar to `MergeSort`.
- *good performance* for large collections.

source

`Base.Sort.PartialQuickSort` – Type.

```
PartialQuickSort{T <: Union{Integer,OrdinalRange}}
```

Indicate that a sorting function should use the partial quick sort algorithm. `PartialQuickSort(k)` is like `QuickSort`, but is only required to find and sort the elements that would end up in `v[k]` were v fully sorted.

Characteristics:

- *not stable*: does not preserve the ordering of elements that compare equal (e.g. "a" and "A" in a sort of letters that ignores case).
- *in-place* in memory.
- *divide-and-conquer*: sort strategy similar to `MergeSort`.

Note that `PartialQuickSort(k)` does not necessarily sort the whole array. For example,

```
julia> x = rand(100);

julia> k = 50:100;

julia> s1 = sort(x; alg=QuickSort);

julia> s2 = sort(x; alg=PartialQuickSort(k));

julia> map(issorted, (s1, s2))
(true, false)

julia> map(x->issorted(x[k]), (s1, s2))
(true, true)

julia> s1[k] == s2[k]
true
```

source

Base.Sort.sortperm! – Function.

```
sortperm!(ix, A; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward, [dims::Integer])
```

Like `sortperm`, but accepts a preallocated index vector or array `ix` with the same axes as `A`. `ix` is initialized to contain the values `LinearIndices(A)`.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

Julia 1.9

The method accepting `dims` requires at least Julia 1.9.

Examples

```
julia> v = [3, 1, 2]; p = zeros(Int, 3);

julia> sortperm!(p, v); p
3-element Vector{Int64}:
 2
 3
 1

julia> v[p]
3-element Vector{Int64}:
 1
 2
 3
```

```
julia> A = [8 7; 5 6]; p = zeros(Int,2, 2);

julia> sortperm!(p, A; dims=1); p
2×2 Matrix{Int64}:
 2  4
 1  3

julia> sortperm!(p, A; dims=2); p
2×2 Matrix{Int64}:
 3  1
 2  4
```

source

Base.sortslices – Function.

```
sortslices(A; dims, alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward)
```

Sort slices of an array A. The required keyword argument dims must be either an integer or a tuple of integers. It specifies the dimension(s) over which the slices are sorted.

E.g., if A is a matrix, dims=1 will sort rows, dims=2 will sort columns. Note that the default comparison function on one dimensional slices sorts lexicographically.

For the remaining keyword arguments, see the documentation of sort!.

Examples

```
julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1) # Sort rows
3×3 Matrix{Int64}:
 -1   6  4
  7   3  5
  9  -2  8

julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1, lt=(x,y)->isless(x[2],y[2]))
3×3 Matrix{Int64}:
  9  -2  8
  7   3  5
 -1   6  4

julia> sortslices([7 3 5; -1 6 4; 9 -2 8], dims=1, rev=true)
3×3 Matrix{Int64}:
  9  -2  8
  7   3  5
 -1   6  4

julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2) # Sort columns
3×3 Matrix{Int64}:
  3   5  7
 -1  -4  6
 -2   8  9
```

```
julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2, alg=InsertionSort,
↪  lt=(x,y)->isless(x[2],y[2]))
3×3 Matrix{Int64}:
  5   3  7
 -4  -1  6
  8  -2  9

julia> sortslices([7 3 5; 6 -1 -4; 9 -2 8], dims=2, rev=true)
3×3 Matrix{Int64}:
 7   5   3
 6  -4  -1
 9   8  -2
```

Higher dimensions

`sortslices` extends naturally to higher dimensions. E.g., if A is a a 2x2x2 array, `sortslices(A, dims=3)` will sort slices within the 3rd dimension, passing the 2x2 slices `A[:, :, 1]` and `A[:, :, 2]` to the comparison function. Note that while there is no default order on higher-dimensional slices, you may use the `by` or `lt` keyword argument to specify such an order.

If `dims` is a tuple, the order of the dimensions in `dims` is relevant and specifies the linear order of the slices. E.g., if A is three dimensional and `dims` is `(1, 2)`, the orderings of the first two dimensions are re-arranged such that the slices (of the remaining third dimension) are sorted. If `dims` is `(2, 1)` instead, the same slices will be taken, but the result order will be row-major instead.

Higher dimensional examples

```
julia> A = permutedims(reshape([4 3; 2 1; 'A' 'B'; 'C' 'D'], (2, 2, 2)), (1, 3, 2))
2×2×2 Array{Any, 3}:
[:, :, 1] =
 4  3
 2  1

[:, :, 2] =
 'A'  'B'
 'C'  'D'

julia> sortslices(A, dims=(1,2))
2×2×2 Array{Any, 3}:
[:, :, 1] =
 1  3
 2  4

[:, :, 2] =
 'D'  'B'
 'C'  'A'

julia> sortslices(A, dims=(2,1))
2×2×2 Array{Any, 3}:
[:, :, 1] =
 1  2
 3  4

[:, :, 2] =
```

```
 'D'  'C'
 'B'  'A'

julia> sortslices(reshape([5; 4; 3; 2; 1], (1,1,5)), dims=3, by=x->x[1,1])
1×1×5 Array{Int64, 3}:
[:, :, 1] =
 1

[:, :, 2] =
 2

[:, :, 3] =
 3

[:, :, 4] =
 4

[:, :, 5] =
 5
```

source

54.2 排列顺序相关的函数

Base.issorted – Function.

```
issorted(v, lt=isless, by=identity, rev::Bool=false, order::Ordering=Forward)
```

Test whether a collection is in sorted order. The keywords modify what order is considered sorted, as described in the sort! documentation.

Examples

```
julia> issorted([1, 2, 3])
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[1])
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[2])
false

julia> issorted([(1, "b"), (2, "a")], by = x -> x[2], rev=true)
true

julia> issorted([1, 2, -2, 3], by=abs)
true
```

source

Base.Sort.searchsorted – Function.

```
searchsorted(v, x; by=identity, lt=isless, rev=false)
```

Return the range of indices in v where values are equivalent to x, or an empty range located at the insertion point if v does not contain values equivalent to x. The vector v must be sorted according to the order defined by the keywords. Refer to `sort!` for the meaning of the keywords and the definition of equivalence. Note that the by function is applied to the searched value x as well as the values in v.

The range is generally found using binary search, but there are optimized implementations for some inputs.

See also: `searchsortedfirst`, `sort!`, `insorted`, `findall`.

Examples

```
julia> searchsorted([1, 2, 4, 5, 5, 7], 4) # single match
3:3

julia> searchsorted([1, 2, 4, 5, 5, 7], 5) # multiple matches
4:5

julia> searchsorted([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
3:2

julia> searchsorted([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
7:6

julia> searchsorted([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
1:0

julia> searchsorted([1=>"one", 2=>"two", 2=>"two", 4=>"four"], 2=>"two", by=first) # compare the
↪  keys of the pairs
2:3
```

source

`Base.Sort.searchsortedfirst` – Function.

```
searchsortedfirst(v, x; by=identity, lt=isless, rev=false)
```

Return the index of the first value in v greater than or equivalent to x. If x is greater than all values in v, return `lastindex(v) + 1`.

The vector v must be sorted according to the order defined by the keywords. `insert!`ing x at the returned index will maintain the sorted order. Refer to `sort!` for the meaning of the keywords and the definition of ”greater than” and equivalence. Note that the by function is applied to the searched value x as well as the values in v.

The index is generally found using binary search, but there are optimized implementations for some inputs.

See also: `searchsortedlast`, `searchsorted`, `findfirst`.

Examples

```
julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 4) # single match
3

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 5) # multiple matches
4

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
3

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
7

julia> searchsortedfirst([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
1

julia> searchsortedfirst([1=>"one", 2=>"two", 4=>"four"], 3=>"three", by=first) # compare the
↪  keys of the pairs
3
```

source

Base.Sort.searchsortedlast – Function.

```
searchsortedlast(v, x; by=identity, lt=isless, rev=false)
```

Return the index of the last value in v less than or equivalent to x. If x is less than all values in v the function returns `firstindex(v) - 1`.

The vector v must be sorted according to the order defined by the keywords. Refer to `sort!` for the meaning of the keywords and the definition of "less than" and equivalence. Note that the by function is applied to the searched value x as well as the values in v.

The index is generally found using binary search, but there are optimized implementations for some inputs

Examples

```
julia> searchsortedlast([1, 2, 4, 5, 5, 7], 4) # single match
3

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 5) # multiple matches
5

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 3) # no match, insert in the middle
2

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 9) # no match, insert at end
6

julia> searchsortedlast([1, 2, 4, 5, 5, 7], 0) # no match, insert at start
0

julia> searchsortedlast([1=>"one", 2=>"two", 4=>"four"], 3=>"three", by=first) # compare the
↪  keys of the pairs
2
```

source

Base.Sort.insorted – Function.

```
insorted(x, v; by=identity, lt=isless, rev=false) -> Bool
```

Determine whether a vector v contains any value equivalent to x. The vector v must be sorted according to the order defined by the keywords. Refer to sort! for the meaning of the keywords and the definition of equivalence. Note that the by function is applied to the searched value x as well as the values in v.

The check is generally done using binary search, but there are optimized implementations for some inputs.

See also in.

Examples

```
julia> insorted(4, [1, 2, 4, 5, 5, 7]) # single match
true

julia> insorted(5, [1, 2, 4, 5, 5, 7]) # multiple matches
true

julia> insorted(3, [1, 2, 4, 5, 5, 7]) # no match
false

julia> insorted(9, [1, 2, 4, 5, 5, 7]) # no match
false

julia> insorted(0, [1, 2, 4, 5, 5, 7]) # no match
false

julia> insorted(2=>"TWO", [1=>"one", 2=>"two", 4=>"four"], by=first) # compare the keys of the
↪  pairs
true
```

Julia 1.6

insorted was added in Julia 1.6.

source

Base.Sort.partialsort! – Function.

```
partialsort!(v, k; by=identity, lt=isless, rev=false)
```

Partially sort the vector v in place so that the value at index k (or range of adjacent values if k is a range) occurs at the position where it would appear if the array were fully sorted. If k is a single index, that value is returned; if k is a range, an array of values at those indices is returned. Note that partialsort! may not fully sort the input array.

For the keyword arguments, see the documentation of sort!.

Examples

```
julia> a = [1, 2, 4, 3, 4]
5-element Vector{Int64}:
 1
 2
 4
 3
 4

julia> partialsort!(a, 4)
4

julia> a
5-element Vector{Int64}:
 1
 2
 3
 4
 4

julia> a = [1, 2, 4, 3, 4]
5-element Vector{Int64}:
 1
 2
 4
 3
 4

julia> partialsort!(a, 4, rev=true)
2

julia> a
5-element Vector{Int64}:
 4
 4
 3
 2
 1
```

source

`Base.Sort.partialsort` – Function.

```
partialsort(v, k, by=identity, lt=isless, rev=false)
```

Variant of `partialsort!` that copies v before partially sorting it, thereby returning the same thing as `partialsort!` but leaving v unmodified.

source

`Base.Sort.partialsortperm` – Function.

```
partialsortperm(v, k; by=ientity, lt=isless, rev=false)
```

Return a partial permutation I of the vector v, so that v[I] returns values of a fully sorted version of v at index k. If k is a range, a vector of indices is returned; if k is an integer, a single index is returned. The order is specified using the same keywords as `sort!`. The permutation is stable: the indices of equal elements will appear in ascending order.

This function is equivalent to, but more efficient than, calling `sortperm(...)[k]`.

Examples

```
julia> v = [3, 1, 2, 1];

julia> v[partialsortperm(v, 1)]
1

julia> p = partialsortperm(v, 1:3)
3-element view(::Vector{Int64}, 1:3) with eltype Int64:
 2
 4
 3

julia> v[p]
3-element Vector{Int64}:
 1
 1
 2
```

source

`Base.Sort.partialsortperm!` – Function.

```
partialsortperm!(ix, v, k; by=identity, lt=isless, rev=false)
```

Like `partialsortperm`, but accepts a preallocated index vector ix the same size as v, which is used to store (a permutation of) the indices of v.

`ix` is initialized to contain the indices of v.

(Typically, the indices of v will be `1:length(v)`, although if v has an alternative array type with non-one-based indices, such as an `OffsetArray`, `ix` must share those same indices)

Upon return, `ix` is guaranteed to have the indices k in their sorted positions, such that

```
partialsortperm!(ix, v, k);
v[ix[k]] == partialsort(v, k)
```

The return value is the kth element of `ix` if k is an integer, or view into `ix` if k is a range.

> **Warning**
>
> Behavior can be unexpected when any mutated argument shares memory with any other argument.

Examples

```
julia> v = [3, 1, 2, 1];

julia> ix = Vector{Int}(undef, 4);

julia> partialsortperm!(ix, v, 1)
2

julia> ix = [1:4;];

julia> partialsortperm!(ix, v, 2:3)
2-element view(::Vector{Int64}, 2:3) with eltype Int64:
 4
 3
```

```

```

source

54.3 排序算法

目前，Julia Base 中有四种可用的排序算法：

- `InsertionSort`
- `QuickSort`
- `PartialQuickSort(k)`
- `MergeSort`

By default, the `sort` family of functions uses stable sorting algorithms that are fast on most inputs. The exact algorithm choice is an implementation detail to allow for future performance improvements. Currently, a hybrid of `RadixSort`, `ScratchQuickSort`, `InsertionSort`, and `CountingSort` is used based on input type, size, and composition. Implementation details are subject to change but currently available in the extended help of `??Base.DEFAULT_STABLE` and the docstrings of internal sorting algorithms listed there.

You can explicitly specify your preferred algorithm with the `alg` keyword (e.g. `sort!(v, alg=PartialQuickSort(10:20))`) or reconfigure the default sorting algorithm for custom types by adding a specialized method to the `Base.Sort.defalg` function. For example, InlineStrings.jl defines the following method:

```
Base.Sort.defalg(::AbstractArray{<:Union{SmallInlineStrings, Missing}}) = InlineStringSort
```

Julia 1.9

The default sorting algorithm (returned by `Base.Sort.defalg`) is guaranteed to be stable since Julia 1.9. Previous versions had unstable edge cases when sorting numeric arrays.

54.4 Alternate Orderings

By default, sort, searchsorted, and related functions use isless to compare two elements in order to determine which should come first. The Base.Order.Ordering abstract type provides a mechanism for defining alternate orderings on the same set of elements: when calling a sorting function like sort!, an instance of Ordering can be provided with the keyword argument order.

Instances of Ordering define an order through the Base.Order.lt function, which works as a generalization of isless. This function's behavior on custom Orderings must satisfy all the conditions of a strict weak order. See sort! for details and examples of valid and invalid lt functions.

Base.Order.Ordering – Type.

```
Base.Order.Ordering
```

Abstract type which represents a total order on some set of elements.

Use Base.Order.lt to compare two elements according to the ordering.

source

Base.Order.lt – Function.

```
lt(o::Ordering, a, b)
```

Test whether a is less than b according to the ordering o.

source

Base.Order.ord – Function.

```
ord(lt, by, rev::Union{Bool, Nothing}, order::Ordering=Forward)
```

Construct an Ordering object from the same arguments used by sort!. Elements are first transformed by the function by (which may be identity) and are then compared according to either the function lt or an existing ordering order. lt should be isless or a function that obeys the same rules as the lt parameter of sort!. Finally, the resulting order is reversed if rev=true.

Passing an lt other than isless along with an order other than Base.Order.Forward or Base.Order.Reverse is not permitted, otherwise all options are independent and can be used together in all possible combinations.

source

Base.Order.Forward – Constant.

```
Base.Order.Forward
```

Default ordering according to isless.

source

`Base.Order.ReverseOrdering` – Type.

```
ReverseOrdering(fwd::Ordering=Forward)
```

A wrapper which reverses an ordering.

For a given `Ordering` o, the following holds for all a, b:

```
lt(ReverseOrdering(o), a, b) == lt(o, b, a)
```

source

`Base.Order.Reverse` – Constant.

```
Base.Order.Reverse
```

Reverse ordering according to `isless`.

source

`Base.Order.By` – Type.

```
By(by, order::Ordering=Forward)
```

`Ordering` which applies `order` to elements after they have been transformed by the function by.

source

`Base.Order.Lt` – Type.

```
Lt(lt)
```

`Ordering` that calls `lt(a, b)` to compare elements. `lt` must obey the same rules as the `lt` parameter of `sort!`.

source

`Base.Order.Perm` – Type.

```
Perm(order::Ordering, data::AbstractVector)
```

`Ordering` on the indices of `data` where `i` is less than `j` if `data[i]` is less than `data[j]` according to `order`. In the case that `data[i]` and `data[j]` are equal, `i` and `j` are compared by numeric value.

source

Chapter 55

迭代相关

`Base.Iterators.Stateful` – Type.

```
Stateful(itr)
```

There are several different ways to think about this iterator wrapper:

1. It provides a mutable wrapper around an iterator and its iteration state.
2. It turns an iterator-like abstraction into a `Channel`-like abstraction.
3. It's an iterator that mutates to become its own rest iterator whenever an item is produced.

`Stateful` provides the regular iterator interface. Like other mutable iterators (e.g. `Base.Channel`), if iteration is stopped early (e.g. by a `break` in a `for` loop), iteration can be resumed from the same spot by continuing to iterate over the same iterator object (in contrast, an immutable iterator would restart from the beginning).

Examples

```
julia> a = Iterators.Stateful("abcdef");

julia> isempty(a)
false

julia> popfirst!(a)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> collect(Iterators.take(a, 3))
3-element Vector{Char}:
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)

julia> collect(a)
2-element Vector{Char}:
 'e': ASCII/Unicode U+0065 (category Ll: Letter, lowercase)
 'f': ASCII/Unicode U+0066 (category Ll: Letter, lowercase)

julia> Iterators.reset!(a); popfirst!(a)
```

```
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> Iterators.reset!(a, "hello"); popfirst!(a)
'h': ASCII/Unicode U+0068 (category Ll: Letter, lowercase)
```

```
julia> a = Iterators.Stateful([1,1,1,2,3,4]);

julia> for x in a; x == 1 || break; end

julia> peek(a)
3

julia> sum(a) # Sum the remaining elements
7
```

source

`Base.Iterators.zip` – Function.

```
zip(iters...)
```

Run multiple iterators at the same time, until any of them is exhausted. The value type of the `zip` iterator is a tuple of values of its subiterators.

Note

`zip` orders the calls to its subiterators in such a way that stateful iterators will not advance when another iterator finishes in the current iteration.

Note

`zip()` with no arguments yields an infinite iterator of empty tuples.

See also: `enumerate`, `Base.splat`.

Examples

```
julia> a = 1:5
1:5

julia> b = ["e","d","b","c","a"]
5-element Vector{String}:
 "e"
 "d"
 "b"
 "c"
 "a"

julia> c = zip(a,b)
zip(1:5, ["e", "d", "b", "c", "a"])
```

```
julia> length(c)
5

julia> first(c)
(1, "e")
```

source

Base.Iterators.enumerate – Function.

```
enumerate(iter)
```

An iterator that yields `(i, x)` where `i` is a counter starting at 1, and `x` is the `i`th value from the given iterator. It's useful when you need not only the values `x` over which you are iterating, but also the number of iterations so far.

Note that `i` may not be valid for indexing `iter`, or may index a different element. This will happen if `iter` has indices that do not start at 1, and may happen for strings, dictionaries, etc. See the `pairs(IndexLinear(), iter)` method if you want to ensure that `i` is an index.

Examples

```
julia> a = ["a", "b", "c"];

julia> for (index, value) in enumerate(a)
           println("$index $value")
       end
1 a
2 b
3 c

julia> str = "naïve";

julia> for (i, val) in enumerate(str)
           print("i = ", i, ", val = ", val, ", ")
           try @show(str[i]) catch e println(e) end
       end
i = 1, val = n, str[i] = 'n'
i = 2, val = a, str[i] = 'a'
i = 3, val = ï, str[i] = 'ï'
i = 4, val = v, StringIndexError("naïve", 4)
i = 5, val = e, str[i] = 'v'
```

source

Base.Iterators.rest – Function.

```
rest(iter, state)
```

An iterator that yields the same elements as `iter`, but starting at the given `state`.

See also: `Iterators.drop`, `Iterators.peel`, `Base.rest`.

Examples

```
julia> collect(Iterators.rest([1,2,3,4], 2))
3-element Vector{Int64}:
 2
 3
 4
```

source

`Base.Iterators.countfrom` – Function.

```
countfrom(start=1, step=1)
```

An iterator that counts forever, starting at `start` and incrementing by `step`.

Examples

```
julia> for v in Iterators.countfrom(5, 2)
           v > 10 && break
           println(v)
       end
5
7
9
```

source

`Base.Iterators.take` – Function.

```
take(iter, n)
```

An iterator that generates at most the first n elements of `iter`.

See also: `drop`, `peel`, `first`, `Base.take!`.

Examples

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
6-element Vector{Int64}:
  1
  3
  5
  7
```

```
  9
 11

julia> collect(Iterators.take(a,3))
3-element Vector{Int64}:
 1
 3
 5
```

source

Base.Iterators.takewhile – Function.

```
takewhile(pred, iter)
```

An iterator that generates element from iter as long as predicate pred is true, afterwards, drops every element.

Julia 1.4

This function requires at least Julia 1.4.

Examples

```
julia> s = collect(1:5)
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> collect(Iterators.takewhile(<(3),s))
2-element Vector{Int64}:
 1
 2
```

source

Base.Iterators.drop – Function.

```
drop(iter, n)
```

An iterator that generates all but the first n elements of iter.

Examples

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
6-element Vector{Int64}:
  1
  3
  5
  7
  9
 11

julia> collect(Iterators.drop(a,4))
2-element Vector{Int64}:
  9
 11
```

source

`Base.Iterators.dropwhile` – Function.

```
dropwhile(pred, iter)
```

An iterator that drops element from `iter` as long as predicate `pred` is true, afterwards, returns every element.

Julia 1.4

This function requires at least Julia 1.4.

Examples

```
julia> s = collect(1:5)
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> collect(Iterators.dropwhile(<(3),s))
3-element Vector{Int64}:
 3
 4
 5
```

source

`Base.Iterators.cycle` – Function.

```
cycle(iter)
```

An iterator that cycles through `iter` forever. If `iter` is empty, so is `cycle(iter)`.

See also: `Iterators.repeated`, `Base.repeat`.

Examples

```
julia> for (i, v) in enumerate(Iterators.cycle("hello"))
           print(v)
           i > 10 && break
       end
hellohelloh
```

source

`Base.Iterators.repeated` – Function.

```
repeated(x[, n::Int])
```

An iterator that generates the value `x` forever. If `n` is specified, generates `x` that many times (equivalent to `take(repeated(x), n)`).

See also: `Iterators.cycle`, `Base.repeat`.

Examples

```
julia> a = Iterators.repeated([1 2], 4);

julia> collect(a)
4-element Vector{Matrix{Int64}}:
 [1 2]
 [1 2]
 [1 2]
 [1 2]
```

source

`Base.Iterators.product` – Function.

```
product(iters...)
```

Return an iterator over the product of several iterators. Each generated element is a tuple whose `i`th element comes from the `i`th argument iterator. The first iterator changes the fastest.

See also: `zip`, `Iterators.flatten`.

Examples

```
julia> collect(Iterators.product(1:2, 3:5))
2×3 Matrix{Tuple{Int64, Int64}}:
 (1, 3)  (1, 4)  (1, 5)
 (2, 3)  (2, 4)  (2, 5)

julia> ans == [(x,y) for x in 1:2, y in 3:5]  # collects a generator involving Iterators.product
true
```

source

Base.Iterators.flatten – Function.

```
flatten(iter)
```

Given an iterator that yields iterators, return an iterator that yields the elements of those iterators. Put differently, the elements of the argument iterator are concatenated.

Examples

```
julia> collect(Iterators.flatten((1:2, 8:9)))
4-element Vector{Int64}:
 1
 2
 8
 9

julia> [(x,y) for x in 0:1 for y in 'a':'c']  # collects generators involving Iterators.flatten
6-element Vector{Tuple{Int64, Char}}:
 (0, 'a')
 (0, 'b')
 (0, 'c')
 (1, 'a')
 (1, 'b')
 (1, 'c')
```

source

Base.Iterators.flatmap – Function.

```
Iterators.flatmap(f, iterators...)
```

Equivalent to `flatten(map(f, iterators...))`.

See also `Iterators.flatten`, `Iterators.map`.

Julia 1.9

This function was added in Julia 1.9.

Examples

```
julia> Iterators.flatmap(n -> -n:2:n, 1:3) |> collect
9-element Vector{Int64}:
 -1
  1
 -2
  0
  2
 -3
 -1
  1
  3

julia> stack(n -> -n:2:n, 1:3)
ERROR: DimensionMismatch: stack expects uniform slices, got axes(x) == (1:3,) while first had
↪  (1:2,)
[...]

julia> Iterators.flatmap(n -> (-n, 10n), 1:2) |> collect
4-element Vector{Int64}:
 -1
 10
 -2
 20

julia> ans == vec(stack(n -> (-n, 10n), 1:2))
true
```

source

Base.Iterators.partition – Function.

```
partition(collection, n)
```

Iterate over a collection n elements at a time.

Examples

```
julia> collect(Iterators.partition([1,2,3,4,5], 2))
3-element Vector{SubArray{Int64, 1, Vector{Int64}, Tuple{UnitRange{Int64}}, true}}:
 [1, 2]
 [3, 4]
 [5]
```

source

Base.Iterators.map – Function.

```
Iterators.map(f, iterators...)
```

Create a lazy mapping. This is another syntax for writing `(f(args...) for args in zip(iterators...))`.

> **Julia 1.6**
>
> This function requires at least Julia 1.6.

Examples

```
julia> collect(Iterators.map(x -> x^2, 1:3))
3-element Vector{Int64}:
 1
 4
 9
```

source

`Base.Iterators.filter` – Function.

```
Iterators.filter(flt, itr)
```

Given a predicate function `flt` and an iterable object `itr`, return an iterable object which upon iteration yields the elements `x` of `itr` that satisfy `flt(x)`. The order of the original iterator is preserved.

This function is *lazy*; that is, it is guaranteed to return in (1) time and use (1) additional space, and `flt` will not be called by an invocation of `filter`. Calls to `flt` will be made when iterating over the returned iterable object. These calls are not cached and repeated calls will be made when reiterating.

See `Base.filter` for an eager implementation of filtering for arrays.

Examples

```
julia> f = Iterators.filter(isodd, [1, 2, 3, 4, 5])
Base.Iterators.Filter{typeof(isodd), Vector{Int64}}(isodd, [1, 2, 3, 4, 5])

julia> foreach(println, f)
1
3
5

julia> [x for x in [1, 2, 3, 4, 5] if isodd(x)]  # collects a generator over Iterators.filter
3-element Vector{Int64}:
 1
 3
 5
```

source

`Base.Iterators.accumulate` – Function.

```
Iterators.accumulate(f, itr; [init])
```

Given a 2-argument function `f` and an iterator `itr`, return a new iterator that successively applies `f` to the previous value and the next element of `itr`.

This is effectively a lazy version of `Base.accumulate`.

> **Julia 1.5**
>
> Keyword argument `init` is added in Julia 1.5.

Examples

```
julia> a = Iterators.accumulate(+, [1,2,3,4]);

julia> foreach(println, a)
1
3
6
10

julia> b = Iterators.accumulate(/, (2, 5, 2, 5); init = 100);

julia> collect(b)
4-element Vector{Float64}:
 50.0
 10.0
  5.0
  1.0
```

source

`Base.Iterators.reverse` – Function.

```
Iterators.reverse(itr)
```

Given an iterator `itr`, then `reverse(itr)` is an iterator over the same collection but in the reverse order. This iterator is "lazy" in that it does not make a copy of the collection in order to reverse it; see `Base.reverse` for an eager implementation.

(By default, this returns an `Iterators.Reverse` object wrapping `itr`, which is iterable if the corresponding `iterate` methods are defined, but some `itr` types may implement more specialized `Iterators.reverse` behaviors.)

Not all iterator types T support reverse-order iteration. If T doesn't, then iterating over `Iterators.reverse(itr::T)` will throw a `MethodError` because of the missing `iterate` methods for `Iterators.Reverse{T}`. (To implement these methods, the original iterator `itr::T` can be obtained from an `r::Iterators.Reverse{T}` object by `r.itr`; more generally, one can use `Iterators.reverse(r)`.)

Examples

```
julia> foreach(println, Iterators.reverse(1:5))
5
4
3
2
1
```

source

Base.Iterators.only – Function.

```
only(x)
```

Return the one and only element of collection x, or throw an ArgumentError if the collection has zero or multiple elements.

See also first, last.

Julia 1.4

This method requires at least Julia 1.4.

Examples

```
julia> only(["a"])
"a"

julia> only("a")
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> only(())
ERROR: ArgumentError: Tuple contains 0 elements, must contain exactly 1 element
Stacktrace:
[...]

julia> only(('a', 'b'))
ERROR: ArgumentError: Tuple contains 2 elements, must contain exactly 1 element
Stacktrace:
[...]
```

source

Base.Iterators.peel – Function.

```
peel(iter)
```

Returns the first element and an iterator over the remaining elements.

If the iterator is empty return nothing (like iterate).

Julia 1.7

Prior versions throw a BoundsError if the iterator is empty.

See also: Iterators.drop, Iterators.take.

Examples

```
julia> (a, rest) = Iterators.peel("abc");

julia> a
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> collect(rest)
2-element Vector{Char}:
 'b': ASCII/Unicode U+0062 (category Ll: Letter, lowercase)
 'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
```

source

Chapter 56

反射与自我检查

Julia 提供了多种运行时的反射功能。

56.1 模块绑定

由 Module 导出的名称可用 names(m::Module) 获得，它会返回一个元素为 Symbol 的数组来表示模块导出的绑定。不管导出状态如何，names(m::Module, all = true) 返回 m 中所有绑定的符号。

56.2 DateType 字段

DataType 的所有字段名称可以使用 fieldnames 来获取。例如，对于下面给定的类型，fieldnames(Point) 会返回一个表示字段名称的 Symbol 元组：

```
julia> struct Point
           x::Int
           y
       end

julia> fieldnames(Point)
(:x, :y)
```

Point 对象中每个字段的类型存储在 Point 本身的 types 变量中：

```
julia> Point.types
svec(Int64, Any)
```

虽然 x 被注释为 Int，但 y 在类型定义里没有注释，因此 y 默认为 Any 类型。

类型本身表示为一个叫做 DataType 的结构：

```
julia> typeof(Point)
DataType
```

Note that fieldnames(DataType) gives the names for each field of DataType itself, and one of these fields is the types field observed in the example above.

56.3 Subtypes

The *direct* subtypes of any DataType may be listed using subtypes. For example, the abstract DataType AbstractFloat has four (concrete) subtypes:

```
julia> subtypes(AbstractFloat)
4-element Vector{Any}:
 BigFloat
 Float16
 Float32
 Float64
```

任何抽象子类型也包括此列表中，但子类型的子类型不在其中。递归使用 subtypes 可以遍历出整个类型树。

56.4 DataType 布局

The internal representation of a DataType is critically important when interfacing with C code and several functions are available to inspect these details. isbitstype(T::DataType) returns true if T is stored with C-compatible alignment. fieldoffset(T::DataType, i::Integer) returns the (byte) offset for field *i* relative to the start of the type.

isbits(T::DataType) 如果 T 类型是以 C 兼容的对齐方式存储，则为 true。fieldoffset(T::DataType, i::Integer) 返回字段 *i* 相对于类型开始的 (字节) 偏移量。

56.5 函数方法

任何泛型函数的方法都可以使用 methods 来列出。用 methodswith 搜索方法调度表来查找接收给定类型的方法。

56.6 扩展和更底层

As discussed in the Metaprogramming section, the macroexpand function gives the unquoted and interpolated expression (Expr) form for a given macro. To use macroexpand, quote the expression block itself (otherwise, the macro will be evaluated and the result will be passed instead!). For example:

```
julia> macroexpand(@__MODULE__, :(@edit println("")) )
:(InteractiveUtils.edit(println, (Base.typesof)("")))
```

The functions Base.Meta.show_sexpr and dump are used to display S-expr style views and depth-nested detail views for any expression.

Finally, the Meta.lower function gives the lowered form of any expression and is of particular interest for understanding how language constructs map to primitive operations such as assignments, branches, and calls:

```
julia> Meta.lower(@__MODULE__, :( [1+2, sin(0.5)] ))
:($(Expr(:thunk, CodeInfo(
    @ none within `top-level scope`
1 ─ %1 = 1 + 2
│   %2 = sin(0.5)
│   %3 = Base.vect(%1, %2)
```

```
└──      return %3
))))
```

56.7 中间表示和编译后表示

检查函数的底层形式需要选择所要显示的特定方法，因为泛型函数可能会有许多具有不同类型签名的方法。为此，用 `code_lowered` 可以指定代码底层中的方法。并且可以用 `code_typed` 来进行类型推断。`code_warntype` 增加 `code_typed` 输出的高亮。

更加接近于机器，一个函数的 LLVM-IR 可以通过使用 `code_llvm` 打印出。最终编译的机器码使用 `code_native` 查看（这将触发之前未调用过的任何函数的 JIT 编译/代码生成）。

为方便起见，上述函数有宏的版本，它们接受标准函数调用并自动展开参数类型：

```
julia> @code_llvm +(1,1)
;  @ int.jl:87 within `+`
; Function Attrs: sspstrong uwtable
define i64 @"julia_+_476"(i64 signext %0, i64 signext %1) #0 {
top:
  %2 = add i64 %1, %0
  ret i64 %2
}
```

For more information see `@code_lowered`, `@code_typed`, `@code_warntype`, `@code_llvm`, and `@code_native`.

Printing of debug information

The aforementioned functions and macros take the keyword argument `debuginfo` that controls the level debug information printed.

```
julia> @code_typed debuginfo=:source +(1,1)
CodeInfo(
    @ int.jl:53 within `+'
1 ─ %1 = Base.add_int(x, y)::Int64
└──      return %1
) => Int64
```

Possible values for `debuginfo` are: `:none`, `:source`, and `:default`. Per default debug information is not printed, but that can be changed by setting `Base.IRShow.default_debuginfo[] = :source`.

Chapter 57

C 接口

Base.@ccall – Macro.

```
@ccall library.function_name(argvalue1::argtype1, ...)::returntype
@ccall function_name(argvalue1::argtype1, ...)::returntype
@ccall $function_pointer(argvalue1::argtype1, ...)::returntype
```

Call a function in a C-exported shared library, specified by `library.function_name`, where `library` is a string constant or literal. The library may be omitted, in which case the `function_name` is resolved in the current process. Alternatively, `@ccall` may also be used to call a function pointer `$function_pointer`, such as one returned by `dlsym`.

Each `argvalue` to `@ccall` is converted to the corresponding `argtype`, by automatic insertion of calls to `unsafe_convert(argtype, cconvert(argtype, argvalue))`. (See also the documentation for `unsafe_convert` and `cconvert` for further details.) In most cases, this simply results in a call to `convert(argtype, argvalue)`.

Examples

```
@ccall strlen(s::Cstring)::Csize_t
```

This calls the C standard library function:

```
size_t strlen(char *)
```

with a Julia variable named `s`. See also `ccall`.

Varargs are supported with the following convention:

```
@ccall printf("%s = %d"::Cstring ; "foo"::Cstring, foo::Cint)::Cint
```

The semicolon is used to separate required arguments (of which there must be at least one) from variadic arguments.

Example using an external library:

```
# C signature of g_uri_escape_string:
# char *g_uri_escape_string(const char *unescaped, const char *reserved_chars_allowed, gboolean
↪  allow_utf8);

const glib = "libglib-2.0"
@ccall glib.g_uri_escape_string(my_uri::Cstring, ":/"::Cstring, true::Cint)::Cstring
```

The string literal could also be used directly before the function name, if desired `"libglib-2.0".g_uri_escape_string(...`

source

`ccall` – Keyword.

```
ccall((function_name, library), returntype, (argtype1, ...), argvalue1, ...)
ccall(function_name, returntype, (argtype1, ...), argvalue1, ...)
ccall(function_pointer, returntype, (argtype1, ...), argvalue1, ...)
```

Call a function in a C-exported shared library, specified by the tuple `(function_name, library)`, where each component is either a string or symbol. Instead of specifying a library, one can also use a `function_name` symbol or string, which is resolved in the current process. Alternatively, `ccall` may also be used to call a function pointer `function_pointer`, such as one returned by `dlsym`.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each `argvalue` to the `ccall` will be converted to the corresponding `argtype`, by automatic insertion of calls to `unsafe_convert(argtype, cconvert(argtype, argvalue))`. (See also the documentation for `unsafe_convert` and `cconvert` for further details.) In most cases, this simply results in a call to `convert(argtype, argvalue)`.

source

`Core.Intrinsics.cglobal` – Function.

```
cglobal((symbol, library) [, type=Cvoid])
```

Obtain a pointer to a global variable in a C-exported shared library, specified exactly as in `ccall`. Returns a `Ptr{Type}`, defaulting to `Ptr{Cvoid}` if no Type argument is supplied. The values can be read or written by `unsafe_load` or `unsafe_store!`, respectively.

source

`Base.@cfunction` – Macro.

```
@cfunction(callable, ReturnType, (ArgumentTypes...,)) -> Ptr{Cvoid}
@cfunction($callable, ReturnType, (ArgumentTypes...,)) -> CFunction
```

Generate a C-callable function pointer from the Julia function `callable` for the given type signature. To pass the return value to a `ccall`, use the argument type `Ptr{Cvoid}` in the signature.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression (although it can include a splat expression). And that these arguments will be evaluated in global scope

during compile-time (not deferred until runtime). Adding a '$' in front of the function argument changes this to instead create a runtime closure over the local variable `callable` (this is not supported on all architectures).

See manual section on ccall and cfunction usage.

Examples

```
julia> function foo(x::Int, y::Int)
           return x + y
       end

julia> @cfunction(foo, Int, (Int, Int))
Ptr{Cvoid} @0x000000001b82fcd0
```

source

`Base.CFunction` – Type.

```
CFunction struct
```

Garbage-collection handle for the return value from `@cfunction` when the first argument is annotated with '$'. Like all `cfunction` handles, it should be passed to `ccall` as a `Ptr{Cvoid}`, and will be converted automatically at the call site to the appropriate type.

See `@cfunction`.

source

`Base.unsafe_convert` – Function.

```
unsafe_convert(T, x)
```

Convert x to a C argument of type T where the input x must be the return value of `cconvert(T, ...)`.

In cases where `convert` would need to take a Julia object and turn it into a `Ptr`, this function should be used to define and perform that conversion.

Be careful to ensure that a Julia reference to x exists as long as the result of this function will be used. Accordingly, the argument x to this function should never be an expression, only a variable name or field reference. For example, x=`a.b.c` is acceptable, but x=`[a,b,c]` is not.

The `unsafe` prefix on this function indicates that using the result of this function after the x argument to this function is no longer accessible to the program may cause undefined behavior, including program corruption or segfaults, at any later time.

See also `cconvert`

source

`Base.cconvert` – Function.

```
cconvert(T,x)
```

Convert x to a value to be passed to C code as type T, typically by calling `convert(T, x)`.

In cases where x cannot be safely converted to T, unlike `convert`, `cconvert` may return an object of a type different from T, which however is suitable for `unsafe_convert` to handle. The result of this function should be kept valid (for the GC) until the result of `unsafe_convert` is not needed anymore. This can be used to allocate memory that will be accessed by the `ccall`. If multiple objects need to be allocated, a tuple of the objects can be used as return value.

Neither `convert` nor `cconvert` should take a Julia object and turn it into a `Ptr`.

source

`Base.unsafe_load` – Function.

```
unsafe_load(p::Ptr{T}, i::Integer=1)
unsafe_load(p::Ptr{T}, order::Symbol)
unsafe_load(p::Ptr{T}, i::Integer, order::Symbol)
```

Load a value of type T from the address of the `i`th element (1-indexed) starting at p. This is equivalent to the C expression `p[i-1]`. Optionally, an atomic memory ordering can be provided.

The `unsafe` prefix on this function indicates that no validation is performed on the pointer p to ensure that it is valid. Like C, the programmer is responsible for ensuring that referenced memory is not freed or garbage collected while invoking this function. Incorrect usage may segfault your program or return garbage answers. Unlike C, dereferencing memory region allocated as different type may be valid provided that the types are compatible.

Julia 1.10

The `order` argument is available as of Julia 1.10.

See also: `atomic`

source

`Base.unsafe_store!` – Function.

```
unsafe_store!(p::Ptr{T}, x, i::Integer=1)
unsafe_store!(p::Ptr{T}, x, order::Symbol)
unsafe_store!(p::Ptr{T}, x, i::Integer, order::Symbol)
```

Store a value of type T to the address of the `i`th element (1-indexed) starting at p. This is equivalent to the C expression `p[i-1] = x`. Optionally, an atomic memory ordering can be provided.

The `unsafe` prefix on this function indicates that no validation is performed on the pointer p to ensure that it is valid. Like C, the programmer is responsible for ensuring that referenced memory is not freed or garbage collected while invoking this function. Incorrect usage may segfault your program. Unlike C, storing memory region allocated as different type may be valid provided that that the types are compatible.

Julia 1.10

The `order` argument is available as of Julia 1.10.

See also: `atomic`

`source`

`Base.unsafe_modify!` – Function.

```
unsafe_modify!(p::Ptr{T}, op, x, [order::Symbol]) -> Pair
```

These atomically perform the operations to get and set a memory address after applying the function op. If supported by the hardware (for example, atomic increment), this may be optimized to the appropriate hardware instruction, otherwise its execution will be similar to:

```
y = unsafe_load(p)
z = op(y, x)
unsafe_store!(p, z)
return y => z
```

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Like C, the programmer is responsible for ensuring that referenced memory is not freed or garbage collected while invoking this function. Incorrect usage may segfault your program.

Julia 1.10

This function requires at least Julia 1.10.

See also: `modifyproperty!`, `atomic`

`source`

`Base.unsafe_replace!` – Function.

```
unsafe_replace!(p::Ptr{T}, expected, desired,
               [success_order::Symbol[, fail_order::Symbol=success_order]]) -> (; old,
               ↪  success::Bool)
```

These atomically perform the operations to get and conditionally set a memory address to a given value. If supported by the hardware, this may be optimized to the appropriate hardware instruction, otherwise its execution will be similar to:

```
y = unsafe_load(p, fail_order)
ok = y === expected
if ok
    unsafe_store!(p, desired, success_order)
end
return (; old = y, success = ok)
```

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Like C, the programmer is responsible for ensuring that referenced memory is not freed or garbage collected while invoking this function. Incorrect usage may segfault your program.

Julia 1.10

This function requires at least Julia 1.10.

See also: `replaceproperty!`, `atomic`

source

`Base.unsafe_swap!` – Function.

```
unsafe_swap!(p::Ptr{T}, x, [order::Symbol])
```

These atomically perform the operations to simultaneously get and set a memory address. If supported by the hardware, this may be optimized to the appropriate hardware instruction, otherwise its execution will be similar to:

```
y = unsafe_load(p)
unsafe_store!(p, x)
return y
```

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Like C, the programmer is responsible for ensuring that referenced memory is not freed or garbage collected while invoking this function. Incorrect usage may segfault your program.

Julia 1.10

This function requires at least Julia 1.10.

See also: `swapproperty!`, `atomic`

source

`Base.unsafe_copyto!` – Method.

```
unsafe_copyto!(dest::Ptr{T}, src::Ptr{T}, N)
```

Copy `N` elements from a source pointer to a destination, with no checking. The size of an element is determined by the type of the pointers.

The `unsafe` prefix on this function indicates that no validation is performed on the pointers `dest` and `src` to ensure that they are valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

`Base.unsafe_copyto!` – Method.

```
unsafe_copyto!(dest::Array, do, src::Array, so, N)
```

Copy N elements from a source array to a destination, starting at the linear index so in the source and do in the destination (1-indexed).

The unsafe prefix on this function indicates that no validation is performed to ensure that N is inbounds on either array. Incorrect usage may corrupt or segfault your program, in the same manner as C.

Warning

Behavior can be unexpected when any mutated argument shares memory with any other argument.

source

Base.copyto! – Function.

```
copyto!(dest::AbstractMatrix, src::UniformScaling)
```

Copies a UniformScaling onto a matrix.

Julia 1.1

In Julia 1.0 this method only supported a square destination matrix. Julia 1.1. added support for a rectangular matrix.

```
copyto!(dest, do, src, so, N)
```

Copy N elements from collection src starting at the linear index so, to array dest starting at the index do. Return dest.

source

```
copyto!(dest::AbstractArray, src) -> dest
```

Copy all elements from collection src to array dest, whose length must be greater than or equal to the length n of src. The first n elements of dest are overwritten, the other elements are left untouched.

See also copy!, copy.

Examples

```
julia> x = [1., 0., 3., 0., 5.];

julia> y = zeros(7);

julia> copyto!(y, x);
```

```
julia> y
7-element Vector{Float64}:
 1.0
 0.0
 3.0
 0.0
 5.0
 0.0
 0.0
```

source

```
copyto!(dest, Rdest::CartesianIndices, src, Rsrc::CartesianIndices) -> dest
```

Copy the block of `src` in the range of `Rsrc` to the block of `dest` in the range of `Rdest`. The sizes of the two regions must match.

Examples

```
julia> A = zeros(5, 5);

julia> B = [1 2; 3 4];

julia> Ainds = CartesianIndices((2:3, 2:3));

julia> Binds = CartesianIndices(B);

julia> copyto!(A, Ainds, B, Binds)
5×5 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0
 0.0  1.0  2.0  0.0  0.0
 0.0  3.0  4.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```

source

`Base.pointer` – Function.

```
pointer(array [, index])
```

Get the native address of an array or string, optionally at a given location `index`.

This function is "unsafe". Be careful to ensure that a Julia reference to `array` exists as long as this pointer will be used. The `GC.@preserve` macro should be used to protect the `array` argument from garbage collection within a given block of code.

Calling `Ref(array[, index])` is generally preferable to this function as it guarantees validity.

source

`Base.unsafe_wrap` – Method.

```
unsafe_wrap(Array, pointer::Ptr{T}, dims; own = false)
```

Wrap a Julia `Array` object around the data at the address given by `pointer`, without making a copy. The pointer element type T determines the array element type. `dims` is either an integer (for a 1d array) or a tuple of the array dimensions. own optionally specifies whether Julia should take ownership of the memory, calling `free` on the pointer when the array is no longer referenced.

This function is labeled "unsafe" because it will crash if `pointer` is not a valid memory address to data of the requested length. Unlike `unsafe_load` and `unsafe_store!`, the programmer is responsible also for ensuring that the underlying data is not accessed through two arrays of different element type, similar to the strict aliasing rule in C.

source

`Base.pointer_from_objref` – Function.

```
pointer_from_objref(x)
```

Get the memory address of a Julia object as a `Ptr`. The existence of the resulting `Ptr` will not protect the object from garbage collection, so you must ensure that the object remains referenced for the whole time that the `Ptr` will be used.

This function may not be called on immutable objects, since they do not have stable memory addresses.

See also `unsafe_pointer_to_objref`.

source

`Base.unsafe_pointer_to_objref` – Function.

```
unsafe_pointer_to_objref(p::Ptr)
```

Convert a `Ptr` to an object reference. Assumes the pointer refers to a valid heap-allocated Julia object. If this is not the case, undefined behavior results, hence this function is considered "unsafe" and should be used with care.

See also `pointer_from_objref`.

source

`Base.disable_sigint` – Function.

```
disable_sigint(f::Function)
```

Disable Ctrl-C handler during execution of a function on the current task, for calling external code that may call julia code that is not interrupt safe. Intended to be called using do block syntax as follows:

```
disable_sigint() do
    # interrupt-unsafe code
    ...
end
```

This is not needed on worker threads (`Threads.threadid() != 1`) since the `InterruptException` will only be delivered to the master thread. External functions that do not call julia code or julia runtime automatically disable sigint during their execution.

source

`Base.reenable_sigint` – Function.

```
reenable_sigint(f::Function)
```

Re-enable Ctrl-C handler during execution of a function. Temporarily reverses the effect of `disable_sigint`.

source

`Base.exit_on_sigint` – Function.

```
exit_on_sigint(on::Bool)
```

Set `exit_on_sigint` flag of the julia runtime. If `false`, Ctrl-C (SIGINT) is capturable as `InterruptException` in `try` block. This is the default behavior in REPL, any code run via `-e` and `-E` and in Julia script run with `-i` option.

If `true`, `InterruptException` is not thrown by Ctrl-C. Running code upon such event requires `atexit`. This is the default behavior in Julia script run without `-i` option.

> **Julia 1.5**
>
> Function `exit_on_sigint` requires at least Julia 1.5.

source

`Base.systemerror` – Function.

```
systemerror(sysfunc[, errno::Cint=Libc.errno()])
systemerror(sysfunc, iftrue::Bool)
```

Raises a `SystemError` for `errno` with the descriptive string `sysfunc` if `iftrue` is `true`

source

`Base.windowserror` – Function.

```
windowserror(sysfunc[, code::UInt32=Libc.GetLastError()])
windowserror(sysfunc, iftrue::Bool)
```

Like `systemerror`, but for Windows API functions that use `GetLastError` to return an error code instead of setting `errno`.

source

`Core.Ptr` – Type.

```
Ptr{T}
```

A memory address referring to data of type T. However, there is no guarantee that the memory is actually valid, or that it actually represents data of the specified type.

source

`Core.Ref` – Type.

```
Ref{T}
```

An object that safely references data of type T. This type is guaranteed to point to valid, Julia-allocated memory of the correct type. The underlying data is protected from freeing by the garbage collector as long as the `Ref` itself is referenced.

In Julia, `Ref` objects are dereferenced (loaded or stored) with `[]`.

Creation of a `Ref` to a value x of type T is usually written `Ref(x)`. Additionally, for creating interior pointers to containers (such as Array or Ptr), it can be written `Ref(a, i)` for creating a reference to the `i`-th element of a.

`Ref{T}()` creates a reference to a value of type T without initialization. For a bitstype T, the value will be whatever currently resides in the memory allocated. For a non-bitstype T, the reference will be undefined and attempting to dereference it will result in an error, "UndefRefError: access to undefined reference".

To check if a `Ref` is an undefined reference, use `isassigned(ref::RefValue)`. For example, `isassigned(Ref{T}())` is `false` if T is not a bitstype. If T is a bitstype, `isassigned(Ref{T}())` will always be true.

When passed as a `ccall` argument (either as a `Ptr` or `Ref` type), a `Ref` object will be converted to a native pointer to the data it references. For most T, or when converted to a `Ptr{Cvoid}`, this is a pointer to the object data. When T is an `isbits` type, this value may be safely mutated, otherwise mutation is strictly undefined behavior.

As a special case, setting `T = Any` will instead cause the creation of a pointer to the reference itself when converted to a `Ptr{Any}` (a `jl_value_t const* const*` if T is immutable, else a `jl_value_t *const *`). When converted to a `Ptr{Cvoid}`, it will still return a pointer to the data region as for any other T.

A `C_NULL` instance of `Ptr` can be passed to a `ccall` `Ref` argument to initialize it.

Use in broadcasting

`Ref` is sometimes used in broadcasting in order to treat the referenced values as a scalar.

Examples

```
julia> Ref(5)
Base.RefValue{Int64}(5)

julia> isa.(Ref([1,2,3]), [Array, Dict, Int]) # Treat reference values as scalar during
↪  broadcasting
3-element BitVector:
 1
 0
```

```
 0

julia> Ref{Function}()  # Undefined reference to a non-bitstype, Function
Base.RefValue{Function}(#undef)

julia> try
           Ref{Function}()[] # Dereferencing an undefined reference will result in an error
       catch e
           println(e)
       end
UndefRefError()

julia> Ref{Int64}()[]; # A reference to a bitstype refers to an undetermined value if not given

julia> isassigned(Ref{Int64}()) # A reference to a bitstype is always assigned
true

julia> Ref{Int64}(0)[] == 0 # Explicitly give a value for a bitstype reference
true
```

source

`Base.isassigned` – Method.

```
isassigned(ref::RefValue) -> Bool
```

Test whether the given `Ref` is associated with a value. This is always true for a `Ref` of a bitstype object. Return `false` if the reference is undefined.

Examples

```
julia> ref = Ref{Function}()
Base.RefValue{Function}(#undef)

julia> isassigned(ref)
false

julia> ref[] = (foobar(x) = x)
foobar (generic function with 1 method)

julia> isassigned(ref)
true

julia> isassigned(Ref{Int}())
true
```

source

`Base.Cchar` – Type.

```
Cchar
```

Equivalent to the native `char` c-type.

source

`Base.Cuchar` – Type.

```
Cuchar
```

Equivalent to the native `unsigned char` c-type (`UInt8`).

source

`Base.Cshort` – Type.

```
Cshort
```

Equivalent to the native `signed short` c-type (`Int16`).

source

`Base.Cstring` – Type.

```
Cstring
```

A C-style string composed of the native character type `Cchar`s. `Cstring`s are NUL-terminated. For C-style strings composed of the native wide character type, see `Cwstring`. For more information about string interoperability with C, see the manual.

source

`Base.Cushort` – Type.

```
Cushort
```

Equivalent to the native `unsigned short` c-type (`UInt16`).

source

`Base.Cint` – Type.

```
Cint
```

Equivalent to the native `signed int` c-type (`Int32`).

source

`Base.Cuint` – Type.

```
Cuint
```

Equivalent to the native `unsigned int` c-type (`UInt32`).

source

`Base.Clong` – Type.

```
Clong
```

Equivalent to the native `signed long` c-type.

source

`Base.Culong` – Type.

```
Culong
```

Equivalent to the native `unsigned long` c-type.

source

`Base.Clonglong` – Type.

```
Clonglong
```

Equivalent to the native `signed long long` c-type (`Int64`).

source

`Base.Culonglong` – Type.

```
Culonglong
```

Equivalent to the native `unsigned long long` c-type (`UInt64`).

source

`Base.Cintmax_t` – Type.

```
Cintmax_t
```

Equivalent to the native `intmax_t` c-type (`Int64`).

source

`Base.Cuintmax_t` – Type.

```
Cuintmax_t
```

Equivalent to the native uintmax_t c-type (UInt64).

source

Base.Csize_t – Type.

```
Csize_t
```

Equivalent to the native size_t c-type (UInt).

source

Base.Cssize_t – Type.

```
Cssize_t
```

Equivalent to the native ssize_t c-type.

source

Base.Cptrdiff_t – Type.

```
Cptrdiff_t
```

Equivalent to the native ptrdiff_t c-type (Int).

source

Base.Cwchar_t – Type.

```
Cwchar_t
```

Equivalent to the native wchar_t c-type (Int32).

source

Base.Cwstring – Type.

```
Cwstring
```

A C-style string composed of the native wide character type Cwchar_ts. Cwstrings are NUL-terminated. For C-style strings composed of the native character type, see Cstring. For more information about string interoperability with C, see the manual.

source

Base.Cfloat – Type.

```
Cfloat
```

Equivalent to the native `float` c-type (`Float32`).

`source`

`Base.Cdouble` – Type.

```
Cdouble
```

Equivalent to the native `double` c-type (`Float64`).

`source`

Chapter 58

LLVM 接口

`Core.Intrinsics.llvmcall` – Function.

```
llvmcall(fun_ir::String, returntype, Tuple{argtype1, ...}, argvalue1, ...)
llvmcall((mod_ir::String, entry_fn::String), returntype, Tuple{argtype1, ...}, argvalue1, ...)
llvmcall((mod_bc::Vector{UInt8}, entry_fn::String), returntype, Tuple{argtype1, ...}, argvalue1,
↪  ...)
```

Call the LLVM code provided in the first argument. There are several ways to specify this first argument:

- as a literal string, representing function-level IR (similar to an LLVM `define` block), with arguments are available as consecutive unnamed SSA variables (%0, %1, etc.);
- as a 2-element tuple, containing a string of module IR and a string representing the name of the entry-point function to call;
- as a 2-element tuple, but with the module provided as an `Vector{UInt8}` with bitcode.

Note that contrary to `ccall`, the argument types must be specified as a tuple type, and not a tuple of types. All types, as well as the LLVM code, should be specified as literals, and not as variables or expressions (it may be necessary to use `@eval` to generate these literals).

See `test/llvmcall.jl` for usage examples.

source

Chapter 59

C 标准库

`Base.Libc.malloc` – Function.

```
malloc(size::Integer) -> Ptr{Cvoid}
```

Call `malloc` from the C standard library.

source

`Base.Libc.calloc` – Function.

```
calloc(num::Integer, size::Integer) -> Ptr{Cvoid}
```

Call `calloc` from the C standard library.

source

`Base.Libc.realloc` – Function.

```
realloc(addr::Ptr, size::Integer) -> Ptr{Cvoid}
```

Call `realloc` from the C standard library.

See warning in the documentation for `free` regarding only using this on memory originally obtained from `malloc`.

source

`Base.memcpy` – Function.

```
memcpy(dst::Ptr, src::Ptr, n::Integer) -> Ptr{Cvoid}
```

Call `memcpy` from the C standard library.

> **Julia 1.10**
>
> Support for `memcpy` requires at least Julia 1.10.

source

`Base.memmove` – Function.

```
memmove(dst::Ptr, src::Ptr, n::Integer) -> Ptr{Cvoid}
```

Call `memmove` from the C standard library.

> **Julia 1.10**
>
> Support for `memmove` requires at least Julia 1.10.

source

`Base.memset` – Function.

```
memset(dst::Ptr, val, n::Integer) -> Ptr{Cvoid}
```

Call `memset` from the C standard library.

> **Julia 1.10**
>
> Support for `memset` requires at least Julia 1.10.

source

`Base.memcmp` – Function.

```
memcmp(a::Ptr, b::Ptr, n::Integer) -> Int
```

Call `memcmp` from the C standard library.

> **Julia 1.10**
>
> Support for `memcmp` requires at least Julia 1.9.

source

`Base.Libc.free` – Function.

```
free(addr::Ptr)
```

Call free from the C standard library. Only use this on memory obtained from malloc, not on pointers retrieved from other C libraries. Ptr objects obtained from C libraries should be freed by the free functions defined in that library, to avoid assertion failures if multiple libc libraries exist on the system.

source

Base.Libc.errno – Function.

```
errno([code])
```

Get the value of the C library's errno. If an argument is specified, it is used to set the value of errno.

The value of errno is only valid immediately after a ccall to a C library routine that sets it. Specifically, you cannot call errno at the next prompt in a REPL, because lots of code is executed between prompts.

source

Base.Libc.strerror – Function.

```
strerror(n=errno())
```

Convert a system call error code to a descriptive string

source

Base.Libc.GetLastError – Function.

```
GetLastError()
```

Call the Win32 GetLastError function [only available on Windows].

source

Base.Libc.FormatMessage – Function.

```
FormatMessage(n=GetLastError())
```

Convert a Win32 system call error code to a descriptive string [only available on Windows].

source

Base.Libc.time – Method.

```
time(t::TmStruct) -> Float64
```

Converts a TmStruct struct to a number of seconds since the epoch.

source

Base.Libc.strftime – Function.

```
strftime([format], time)
```

Convert time, given as a number of seconds since the epoch or a `TmStruct`, to a formatted string using the given format. Supported formats are the same as those in the standard C library.

source

`Base.Libc.strptime` – Function.

```
strptime([format], timestr)
```

Parse a formatted time string into a `TmStruct` giving the seconds, minute, hour, date, etc. Supported formats are the same as those in the standard C library. On some platforms, timezones will not be parsed correctly. If the result of this function will be passed to `time` to convert it to seconds since the epoch, the `isdst` field should be filled in manually. Setting it to `-1` will tell the C library to use the current system settings to determine the timezone.

source

`Base.Libc.TmStruct` – Type.

```
TmStruct([seconds])
```

Convert a number of seconds since the epoch to broken-down format, with fields `sec`, `min`, `hour`, `mday`, `month`, `year`, `wday`, `yday`, and `isdst`.

source

`Base.Libc.flush_cstdio` – Function.

```
flush_cstdio()
```

Flushes the C `stdout` and `stderr` streams (which may have been written to by external C code).

source

`Base.Libc.systemsleep` – Function.

```
systemsleep(s::Real)
```

Suspends execution for `s` seconds. This function does not yield to Julia's scheduler and therefore blocks the Julia thread that it is running on for the duration of the sleep time.

See also `sleep`.

source

Chapter 60

堆栈跟踪

Base.StackTraces.StackFrame – Type.

```
StackFrame
```

Stack information representing execution context, with the following fields:

- `func::Symbol`
  The name of the function containing the execution context.
- `linfo::Union{Core.MethodInstance, Method, Module, Core.CodeInfo, Nothing}`
  The MethodInstance or CodeInfo containing the execution context (if it could be found), or Module (for macro expansions)"
- `file::Symbol`
  The path to the file containing the execution context.
- `line::Int`
  The line number in the file containing the execution context.
- `from_c::Bool`
  True if the code is from C.
- `inlined::Bool`
  True if the code is from an inlined frame.
- `pointer::UInt64`
  Representation of the pointer to the execution context as returned by `backtrace`.

source

Base.StackTraces.StackTrace – Type.

```
StackTrace
```

An alias for `Vector{StackFrame}` provided for convenience; returned by calls to `stacktrace`.

source

Base.StackTraces.stacktrace – Function.

```
stacktrace([trace::Vector{Ptr{Cvoid}},] [c_funcs::Bool=false]) -> StackTrace
```

Return a stack trace in the form of a vector of `StackFrames`. (By default stacktrace doesn't return C functions, but this can be enabled.) When called without specifying a trace, `stacktrace` first calls `backtrace`.

source

`Base.StackTraces` 中以下方法和类型不会被导出，需要显式调用，例如 `StackTraces.lookup(ptr)`。

`Base.StackTraces.lookup` – Function.

```
lookup(pointer::Ptr{Cvoid}) -> Vector{StackFrame}
```

Given a pointer to an execution context (usually generated by a call to `backtrace`), looks up stack frame context information. Returns an array of frame information for all functions inlined at that point, innermost function first.

source

`Base.StackTraces.remove_frames!` – Function.

```
remove_frames!(stack::StackTrace, name::Symbol)
```

Takes a `StackTrace` (a vector of `StackFrames`) and a function name (a `Symbol`) and removes the `StackFrame` specified by the function name from the `StackTrace` (also removing all frames above the specified function). Primarily used to remove `StackTraces` functions from the `StackTrace` prior to returning it.

source

```
remove_frames!(stack::StackTrace, m::Module)
```

Return the `StackTrace` with all `StackFrames` from the provided `Module` removed.

source

Chapter 61

SIMD 支持

`VecElement{T}` 类型是为了构建 SIMD 运算符的库。实际使用中要求使用 `llvmcall`。类型按下文定义：

```
struct VecElement{T}
    value::T
end
```

It has a special compilation rule: a homogeneous tuple of `VecElement{T}` maps to an LLVM `vector` type when T is a primitive bits type.

使用 `-O3` 参数时，编译器 可能自动为这样的元组向量化运算符。例如接下来的程序，使用 `julia -O3` 编译，在 x86 系统中会生成两个 SIMD 附加指令（`addps`）：

```
const m128 = NTuple{4,VecElement{Float32}}

function add(a::m128, b::m128)
    (VecElement(a[1].value+b[1].value),
     VecElement(a[2].value+b[2].value),
     VecElement(a[3].value+b[3].value),
     VecElement(a[4].value+b[4].value))
end

triple(c::m128) = add(add(c,c),c)

code_native(triple,(m128,))
```

然而，因为无法依靠自动向量化，以后将主要通过使用基于 `llvmcall` 的库来提供 SIMD 支持。

Part IV

标准库

Chapter 62

ArgTools

62.1 Argument Handling

ArgTools.ArgRead – Type.

```
ArgRead = Union{AbstractString, AbstractCmd, IO}
```

The ArgRead types is a union of the types that the arg_read function knows how to convert into readable IO handles. See arg_read for details.

ArgTools.ArgWrite – Type.

```
ArgWrite = Union{AbstractString, AbstractCmd, IO}
```

The ArgWrite types is a union of the types that the arg_write function knows how to convert into writeable IO handles, except for Nothing which arg_write handles by generating a temporary file. See arg_write for details.

ArgTools.arg_read – Function.

```
arg_read(f::Function, arg::ArgRead) -> f(arg_io)
```

The arg_read function accepts an argument arg that can be any of these:

- AbstractString: a file path to be opened for reading
- AbstractCmd: a command to be run, reading from its standard output
- IO: an open IO handle to be read from

Whether the body returns normally or throws an error, a path which is opened will be closed before returning from arg_read and an IO handle will be flushed but not closed before returning from arg_read.

Note: when opening a file, ArgTools will pass lock = false to the file open(...) call. Therefore, the object returned by this function should not be used from multiple threads. This restriction may be relaxed in the future, which would not break any working code.

`ArgTools.arg_write` – Function.

```
arg_write(f::Function, arg::ArgWrite) -> arg
arg_write(f::Function, arg::Nothing) -> tempname()
```

The `arg_read` function accepts an argument `arg` that can be any of these:

- `AbstractString`: a file path to be opened for writing
- `AbstractCmd`: a command to be run, writing to its standard input
- `IO`: an open IO handle to be written to
- `Nothing`: a temporary path should be written to

If the body returns normally, a path that is opened will be closed upon completion; an IO handle argument is left open but flushed before return. If the argument is `nothing` then a temporary path is opened for writing and closed open completion and the path is returned from `arg_write`. In all other cases, `arg` itself is returned. This is a useful pattern since you can consistently return whatever was written, whether an argument was passed or not.

If there is an error during the evaluation of the body, a path that is opened by `arg_write` for writing will be deleted, whether it's passed in as a string or a temporary path generated when `arg` is `nothing`.

Note: when opening a file, ArgTools will pass `lock = false` to the file `open(...)` call. Therefore, the object returned by this function should not be used from multiple threads. This restriction may be relaxed in the future, which would not break any working code.

`ArgTools.arg_isdir` – Function.

```
arg_isdir(f::Function, arg::AbstractString) -> f(arg)
```

The `arg_isdir` function takes `arg` which must be the path to an existing directory (an error is raised otherwise) and passes that path to `f` finally returning the result of `f(arg)`. This is definitely the least useful tool offered by `ArgTools` and mostly exists for symmetry with `arg_mkdir` and to give consistent error messages.

`ArgTools.arg_mkdir` – Function.

```
arg_mkdir(f::Function, arg::AbstractString) -> arg
arg_mkdir(f::Function, arg::Nothing) -> mktempdir()
```

The `arg_mkdir` function takes `arg` which must either be one of:

- a path to an already existing empty directory,
- a non-existent path which can be created as a directory, or
- `nothing` in which case a temporary directory is created.

In all cases the path to the directory is returned. If an error occurs during `f(arg)`, the directory is returned to its original state: if it already existed but was empty, it will be emptied; if it did not exist it will be deleted.

62.2 Function Testing

ArgTools.arg_readers – Function.

```
arg_readers(arg :: AbstractString, [ type = ArgRead ]) do arg::Function
    ## pre-test setup ##
    @arg_test arg begin
        arg :: ArgRead
        ## test using `arg` ##
    end
    ## post-test cleanup ##
end
```

The arg_readers function takes a path to be read and a single-argument do block, which is invoked once for each test reader type that arg_read can handle. If the optional type argument is given then the do block is only invoked for readers that produce arguments of that type.

The arg passed to the do block is not the argument value itself, because some of test argument types need to be initialized and finalized for each test case. Consider an open file handle argument: once you've used it for one test, you can't use it again; you need to close it and open the file again for the next test. This function arg can be converted into an ArgRead instance using @arg_test arg begin ... end.

ArgTools.arg_writers – Function.

```
arg_writers([ type = ArgWrite ]) do path::String, arg::Function
    ## pre-test setup ##
    @arg_test arg begin
        arg :: ArgWrite
        ## test using `arg` ##
    end
    ## post-test cleanup ##
end
```

The arg_writers function takes a do block, which is invoked once for each test writer type that arg_write can handle with a temporary (non-existent) path and arg which can be converted into various writable argument types which write to path. If the optional type argument is given then the do block is only invoked for writers that produce arguments of that type.

The arg passed to the do block is not the argument value itself, because some of test argument types need to be initialized and finalized for each test case. Consider an open file handle argument: once you've used it for one test, you can't use it again; you need to close it and open the file again for the next test. This function arg can be converted into an ArgWrite instance using @arg_test arg begin ... end.

There is also an arg_writers method that takes a path name like arg_readers:

```
arg_writers(path::AbstractString, [ type = ArgWrite ]) do arg::Function
    ## pre-test setup ##
    @arg_test arg begin
        # here `arg :: ArgWrite`
        ## test using `arg` ##
    end
    ## post-test cleanup ##
end
```

This method is useful if you need to specify `path` instead of using path name generated by `tempname()`. Since `path` is passed from outside of `arg_writers`, the path is not an argument to the do block in this form.

`ArgTools.@arg_test` – Macro.

```
@arg_test arg1 arg2 ... body
```

The `@arg_test` macro is used to convert `arg` functions provided by `arg_readers` and `arg_writers` into actual argument values. When you write `@arg_test arg body` it is equivalent to `arg(arg -> body)`.

Chapter 63

Artifacts

Starting with Julia 1.6, the artifacts support has moved from `Pkg.jl` to Julia itself. Until proper documentation can be added here, you can learn more about artifacts in the `Pkg.jl` manual at https://julialang.github.io/Pkg.jl/v1/artifacts/.

> **Julia 1.6**
>
> Julia's artifacts API requires at least Julia 1.6. In Julia versions 1.3 to 1.5, you can use `Pkg.Artifacts` instead.

`Artifacts.artifact_meta` – Function.

```
artifact_meta(name::String, artifacts_toml::String;
              platform::AbstractPlatform = HostPlatform(),
              pkg_uuid::Union{Base.UUID,Nothing}=nothing)
```

Get metadata about a given artifact (identified by name) stored within the given `(Julia)Artifacts.toml` file. If the artifact is platform-specific, use `platform` to choose the most appropriate mapping. If none is found, return `nothing`.

> **Julia 1.3**
>
> This function requires at least Julia 1.3.

`Artifacts.artifact_hash` – Function.

```
artifact_hash(name::String, artifacts_toml::String;
              platform::AbstractPlatform = HostPlatform())
```

Thin wrapper around `artifact_meta()` to return the hash of the specified, platform- collapsed artifact. Returns `nothing` if no mapping can be found.

> **Julia 1.3**
>
> This function requires at least Julia 1.3.

`Artifacts.find_artifacts_toml` – Function.

```
find_artifacts_toml(path::String)
```

Given the path to a `.jl` file, (such as the one returned by `__source__.file` in a macro context), find the `(Julia)Artifacts.toml` that is contained within the containing project (if it exists), otherwise return `nothing`.

Julia 1.3

This function requires at least Julia 1.3.

`Artifacts.@artifact_str` – Macro.

```
macro artifact_str(name)
```

Return the on-disk path to an artifact. Automatically looks the artifact up by name in the project's `(Julia)Artifacts.toml` file. Throws an error on if the requested artifact is not present. If run in the REPL, searches for the toml file starting in the current directory, see `find_artifacts_toml()` for more.

If the artifact is marked "lazy" and the package has `using LazyArtifacts` defined, the artifact will be downloaded on-demand with Pkg the first time this macro tries to compute the path. The files will then be left installed locally for later.

If `name` contains a forward or backward slash, all elements after the first slash will be taken to be path names indexing into the artifact, allowing for an easy one-liner to access a single file/directory within an artifact. Example:

```
ffmpeg_path = @artifact"FFMPEG/bin/ffmpeg"
```

Julia 1.3

This macro requires at least Julia 1.3.

Julia 1.6

Slash-indexing requires at least Julia 1.6.

Chapter 64

Base64

Base64.Base64 – Module.

```
Base64
```

Functionality for base64 encoding and decoding, a method to represent binary data using text, common on the web.

Base64.Base64EncodePipe – Type.

```
Base64EncodePipe(ostream)
```

Return a new write-only I/O stream, which converts any bytes written to it into base64-encoded ASCII bytes written to ostream. Calling close on the Base64EncodePipe stream is necessary to complete the encoding (but does not close ostream).

Examples

```
julia> io = IOBuffer();

julia> iob64_encode = Base64EncodePipe(io);

julia> write(iob64_encode, "Hello!")
6

julia> close(iob64_encode);

julia> str = String(take!(io))
"SGVsbG8h"

julia> String(base64decode(str))
"Hello!"
```

Base64.base64encode – Function.

```
base64encode(writefunc, args...; context=nothing)
base64encode(args...; context=nothing)
```

Given a `write`-like function `writefunc`, which takes an I/O stream as its first argument, `base64encode(writefunc, args...)` calls `writefunc` to write `args...` to a base64-encoded string, and returns the string. `base64encode(args...)` is equivalent to `base64encode(write, args...)`: it converts its arguments into bytes using the standard `write` functions and returns the base64-encoded string.

The optional keyword argument `context` can be set to `:key=>value` pair or an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `writefunc` or `write`.

See also `base64decode`.

`Base64.Base64DecodePipe` – Type.

```
Base64DecodePipe(istream)
```

Return a new read-only I/O stream, which decodes base64-encoded data read from `istream`.

Examples

```
julia> io = IOBuffer();

julia> iob64_decode = Base64DecodePipe(io);

julia> write(io, "SGVsbG8h")
8

julia> seekstart(io);

julia> String(read(iob64_decode))
"Hello!"
```

`Base64.base64decode` – Function.

```
base64decode(string)
```

Decode the base64-encoded `string` and returns a `Vector{UInt8}` of the decoded bytes.

See also `base64encode`.

Examples

```
julia> b = base64decode("SGVsbG8h")
6-element Vector{UInt8}:
 0x48
 0x65
 0x6c
 0x6c
 0x6f
```

```
 0x21

julia> String(b)
"Hello!"
```

`Base64.stringmime` – Function.

```
stringmime(mime, x; context=nothing)
```

Return an `AbstractString` containing the representation of `x` in the requested `mime` type. This is similar to `repr(mime, x)` except that binary data is base64-encoded as an ASCII string.

The optional keyword argument `context` can be set to `:key=>value` pair or an `IO` or `IOContext` object whose attributes are used for the I/O stream passed to `show`.

Chapter 65

CRC32c

Standard library module for computing the CRC-32c checksum.

`CRC32c.crc32c` – Function.

```
crc32c(data, crc::UInt32=0x00000000)
```

Compute the CRC-32c checksum of the given `data`, which can be an `Array{UInt8}`, a contiguous subarray thereof, or a `String`. Optionally, you can pass a starting `crc` integer to be mixed in with the checksum. The `crc` parameter can be used to compute a checksum on data divided into chunks: performing `crc32c(data2, crc32c(data1))` is equivalent to the checksum of `[data1; data2]`. (Technically, a little-endian checksum is computed.)

There is also a method `crc32c(io, nb, crc)` to checksum `nb` bytes from a stream `io`, or `crc32c(io, crc)` to checksum all the remaining bytes. Hence you can do `open(crc32c, filename)` to checksum an entire file, or `crc32c(seekstart(buf))` to checksum an `IOBuffer` without calling `take!`.

For a `String`, note that the result is specific to the UTF-8 encoding (a different checksum would be obtained from a different Unicode encoding). To checksum an `a::Array` of some other bitstype, you can do `crc32c(reinterpret(UInt8,a))`, but note that the result may be endian-dependent.

`CRC32c.crc32c` – Method.

```
crc32c(io::IO, [nb::Integer,] crc::UInt32=0x00000000)
```

Read up to `nb` bytes from `io` and return the CRC-32c checksum, optionally mixed with a starting `crc` integer. If `nb` is not supplied, then `io` will be read until the end of the stream.

Chapter 66

日期

Dates 模块提供了两种类型来处理日期：Date 和 DateTime，分别精确到日和毫秒；两者都是抽象类型 TimeType 的子类型。区分类型的动机很简单：不必处理更高精度所带来的复杂性时，一些操作在代码和思维推理上都更加简单。例如，由于 Date 类型仅精确到日（即没有时、分或秒），因此避免了时区、夏令时和闰秒等不必要的通常考虑。

Date 和 DateTime 类型都是基本不可变类型 Int64 的包装类。这两种类型的单个 instant 字段实际上属于 UTInstant{P} 类型。这种类型表示的是一种基于世界时间（UT）持续增长的机器时间 [1]。DateTime 类型并不考虑时区（用 Python 的话讲，它是 *naive* 的），与 Java 8 中的 *LocalDateTime* 类似。如果需要附加时区功能，可以通过 TimeZones.jl 包 实现，其汇编了来自 IANA 时区数据库 的数据。Date 和 DateTime 都基于 ISO 8601 标准，遵循公历（格里高利历）。值得注意的是，ISO 8601 标准对公元前的日期需要特别处理。通常来说，公元前的最后一天是公元前 1 年的 12 月 31 日，接下来的一天是公元 1 年的 1 月 1 日，公元 0 年是不存在的。但是，在 ISO 8601 标准中，公元前 1 年被表示为 0 年，即 0001-01-01 的前一天是 0000-12-31，而 -0001（没错，年数为-1）的那一年则实际上是公元前 2 年，-0002 则表示公元前 3 年，以此类推。

66.1 构造函数

Date 和 DateTime 类型可以通过整数或 Period 类型，解析，或调整器来构造（稍后会详细介绍）：

```
julia> DateTime(2013)
2013-01-01T00:00:00

julia> DateTime(2013,7)
2013-07-01T00:00:00

julia> DateTime(2013,7,1)
2013-07-01T00:00:00

julia> DateTime(2013,7,1,12)
2013-07-01T12:00:00
```

[1] The notion of the UT second is actually quite fundamental. There are basically two different notions of time generally accepted, one based on the physical rotation of the earth (one full rotation = 1 day), the other based on the SI second (a fixed, constant value). These are radically different! Think about it, a "UT second", as defined relative to the rotation of the earth, may have a different absolute length depending on the day! Anyway, the fact that Date and DateTime are based on UT seconds is a simplifying, yet honest assumption so that things like leap seconds and all their complexity can be avoided. This basis of time is formally called UT or UT1. Basing types on the UT second basically means that every minute has 60 seconds and every day has 24 hours and leads to more natural calculations when working with calendar dates.

```
julia> DateTime(2013,7,1,12,30)
2013-07-01T12:30:00

julia> DateTime(2013,7,1,12,30,59)
2013-07-01T12:30:59

julia> DateTime(2013,7,1,12,30,59,1)
2013-07-01T12:30:59.001

julia> Date(2013)
2013-01-01

julia> Date(2013,7)
2013-07-01

julia> Date(2013,7,1)
2013-07-01

julia> Date(Dates.Year(2013),Dates.Month(7),Dates.Day(1))
2013-07-01

julia> Date(Dates.Month(7),Dates.Year(2013))
2013-07-01
```

Date or DateTime parsing is accomplished by the use of format strings. Format strings work by the notion of defining *delimited* or *fixed-width* "slots" that contain a period to parse and passing the text to parse and format string to a Date or DateTime constructor, of the form `Date("2015-01-01",dateformat"y-m-d")` or `DateTime("20150101",dateformat"yyyymmdd")`.

有分隔的插入点是通过指定解析器在两个时段之间的分隔符来进行标记的。例如，`"y-m-d"` 会告诉解析器，一个诸如 `"2014-07-16"` 的时间字符串，应该在第一个和第二个插入点之间查找 `-` 字符。`y`，`m` 和 `d` 字符则告诉解析器每一个插入点对应的时段名称。

As in the case of constructors above such as `Date(2013)`, delimited `DateFormats` allow for missing parts of dates and times so long as the preceding parts are given. The other parts are given the usual default values. For example, `Date("1981-03", dateformat"y-m-d")` returns `1981-03-01`, whilst `Date("31/12", dateformat"d/m/y")` gives `0001-12-31`. (Note that the default year is 1 AD/CE.) Consequently, an empty string will always return `0001-01-01` for `Date`s, and `0001-01-01T00:00:00.000` for `DateTime`s.

As in the case of constructors above such as `Date(2013)`, delimited `DateFormats` allow for missing parts of dates and times so long as the preceding parts are given. The other parts are given the usual default values. For example, `Date("1981-03", dateformat"y-m-d")` returns `1981-03-01`, whilst `Date("31/12", dateformat"d/m/y")` gives `0001-12-31`. (Note that the default year is 1 AD/CE.) An empty string, however, always throws an `ArgumentError`.

Fixed-width slots are specified by repeating the period character the number of times corresponding to the width with no delimiter between characters. So `dateformat"yyyymmdd"` would correspond to a date string like `"20140716"`. The parser distinguishes a fixed-width slot by the absence of a delimiter, noting the transition `"yyyymm"` from one period character to the next.

Support for text-form month parsing is also supported through the `u` and `U` characters, for abbreviated and full-length month names, respectively. By default, only English month names are supported, so `u` corresponds to "Jan", "Feb", "Mar", etc. And `U` corresponds to "January", "February", "March", etc. Similar to other name=>value mapping functions dayname and monthname, custom locales can be loaded by passing in

the `locale=>Dict{String,Int}` mapping to the `MONTHTOVALUEABBR` and `MONTHTOVALUE` dicts for abbreviated and full-name month names, respectively.

The above examples used the `dateformat""` string macro. This macro creates a `DateFormat` object once when the macro is expanded and uses the same `DateFormat` object even if a code snippet is run multiple times.

```
julia> for i = 1:10^5
           Date("2015-01-01", dateformat"y-m-d")
       end
```

Or you can create the DateFormat object explicitly:

```
julia> df = DateFormat("y-m-d");

julia> dt = Date("2015-01-01",df)
2015-01-01

julia> dt2 = Date("2015-01-02",df)
2015-01-02
```

Alternatively, use broadcasting:

```
julia> years = ["2015", "2016"];

julia> Date.(years, DateFormat("yyyy"))
2-element Vector{Date}:
 2015-01-01
 2016-01-01
```

For convenience, you may pass the format string directly (e.g., `Date("2015-01-01","y-m-d")`), although this form incurs performance costs if you are parsing the same format repeatedly, as it internally creates a new `DateFormat` object each time.

As well as via the constructors, a `Date` or `DateTime` can be constructed from strings using the `parse` and `tryparse` functions, but with an optional third argument of type `DateFormat` specifying the format; for example, `parse(Date, "06.23.2013", dateformat"m.d.y")`, or `tryparse(DateTime, "1999-12-31T23:59:59")` which uses the default format. The notable difference between the functions is that with `tryparse`, an error is not thrown if the string is empty or in an invalid format; instead `nothing` is returned.

Julia 1.9

Before Julia 1.9, empty strings could be passed to constructors and `parse` without error, returning as appropriate `DateTime(1)`, `Date(1)` or `Time(0)`. Likewise, `tryparse` did not return `nothing`.

A full suite of parsing and formatting tests and examples is available in `stdlib/Dates/test/io.jl`.

66.2 Durations/Comparisons

Finding the length of time between two `Date` or `DateTime` is straightforward given their underlying representation as `UTInstant{Day}` and `UTInstant{Millisecond}`, respectively. The difference between `Date` is returned

in the number of Day, and DateTime in the number of Millisecond. Similarly, comparing TimeType is a simple matter of comparing the underlying machine instants (which in turn compares the internal Int64 values).

```
julia> dt = Date(2012,2,29)
2012-02-29

julia> dt2 = Date(2000,2,1)
2000-02-01

julia> dump(dt)
Date
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 734562

julia> dump(dt2)
Date
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 730151

julia> dt > dt2
true

julia> dt != dt2
true

julia> dt + dt2
ERROR: MethodError: no method matching +(::Date, ::Date)
[...]

julia> dt * dt2
ERROR: MethodError: no method matching *(::Date, ::Date)
[...]

julia> dt / dt2
ERROR: MethodError: no method matching /(::Date, ::Date)

julia> dt - dt2
4411 days

julia> dt2 - dt
-4411 days

julia> dt = DateTime(2012,2,29)
2012-02-29T00:00:00

julia> dt2 = DateTime(2000,2,1)
2000-02-01T00:00:00

julia> dt - dt2
381110400000 milliseconds
```

66.3 Accessor Functions

Because the `Date` and `DateTime` types are stored as single `Int64` values, date parts or fields can be retrieved through accessor functions. The lowercase accessors return the field as an integer:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.year(t)
2014

julia> Dates.month(t)
1

julia> Dates.week(t)
5

julia> Dates.day(t)
31
```

While propercase return the same value in the corresponding `Period` type:

```
julia> Dates.Year(t)
2014 years

julia> Dates.Day(t)
31 days
```

Compound methods are provided because it is more efficient to access multiple fields at the same time than individually:

```
julia> Dates.yearmonth(t)
(2014, 1)

julia> Dates.monthday(t)
(1, 31)

julia> Dates.yearmonthday(t)
(2014, 1, 31)
```

One may also access the underlying `UTInstant` or integer value:

```
julia> dump(t)
Date
  instant: Dates.UTInstant{Day}
    periods: Day
      value: Int64 735264

julia> t.instant
Dates.UTInstant{Day}(Day(735264))

julia> Dates.value(t)
735264
```

66.4 Query Functions

Query functions provide calendrical information about a TimeType. They include information about the day of the week:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.dayofweek(t)
5

julia> Dates.dayname(t)
"Friday"

julia> Dates.dayofweekofmonth(t) # 5th Friday of January
5
```

Month of the year:

```
julia> Dates.monthname(t)
"January"

julia> Dates.daysinmonth(t)
31
```

As well as information about the TimeType's year and quarter:

```
julia> Dates.isleapyear(t)
false

julia> Dates.dayofyear(t)
31

julia> Dates.quarterofyear(t)
1

julia> Dates.dayofquarter(t)
31
```

The dayname and monthname methods can also take an optional `locale` keyword that can be used to return the name of the day or month of the year for other languages/locales. There are also versions of these functions returning the abbreviated names, namely dayabbr and monthabbr. First the mapping is loaded into the LOCALES variable:

```
julia> french_months = ["janvier", "février", "mars", "avril", "mai", "juin",
                        "juillet", "août", "septembre", "octobre", "novembre", "décembre"];

julia> french_monts_abbrev = ["janv","févr","mars","avril","mai","juin",
                              "juil","août","sept","oct","nov","déc"];

julia> french_days = ["lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanche"];

julia> Dates.LOCALES["french"] = Dates.DateLocale(french_months, french_monts_abbrev, french_days,
↪  [""]);
```

The above mentioned functions can then be used to perform the queries:

```
julia> Dates.dayname(t;locale="french")
"vendredi"

julia> Dates.monthname(t;locale="french")
"janvier"

julia> Dates.monthabbr(t;locale="french")
"janv"
```

Since the abbreviated versions of the days are not loaded, trying to use the function `dayabbr` will throw an error.

```
julia> Dates.dayabbr(t;locale="french")
ERROR: BoundsError: attempt to access 1-element Vector{String} at index [5]
Stacktrace:
[...]
```

66.5 TimeType-Period Arithmetic

It's good practice when using any language/date framework to be familiar with how date-period arithmetic is handled as there are some tricky issues to deal with (though much less so for day-precision types).

The `Dates` module approach tries to follow the simple principle of trying to change as little as possible when doing `Period` arithmetic. This approach is also often known as *calendrical* arithmetic or what you would probably guess if someone were to ask you the same calculation in a conversation. Why all the fuss about this? Let's take a classic example: add 1 month to January 31st, 2014. What's the answer? Javascript will say March 3 (assumes 31 days). PHP says March 2 (assumes 30 days). The fact is, there is no right answer. In the `Dates` module, it gives the result of February 28th. How does it figure that out? Consider the classic 7-7-7 gambling game in casinos.

Now just imagine that instead of 7-7-7, the slots are Year-Month-Day, or in our example, 2014-01-31. When you ask to add 1 month to this date, the month slot is incremented, so now we have 2014-02-31. Then the day number is checked if it is greater than the last valid day of the new month; if it is (as in the case above), the day number is adjusted down to the last valid day (28). What are the ramifications with this approach? Go ahead and add another month to our date, `2014-02-28 + Month(1) == 2014-03-28`. What? Were you expecting the last day of March? Nope, sorry, remember the 7-7-7 slots. As few slots as possible are going to change, so we first increment the month slot by 1, 2014-03-28, and boom, we're done because that's a valid date. On the other hand, if we were to add 2 months to our original date, 2014-01-31, then we end up with 2014-03-31, as expected. The other ramification of this approach is a loss in associativity when a specific ordering is forced (i.e. adding things in different orders results in different outcomes). For example:

```
julia> (Date(2014,1,29)+Dates.Day(1)) + Dates.Month(1)
2014-02-28

julia> (Date(2014,1,29)+Dates.Month(1)) + Dates.Day(1)
2014-03-01
```

What's going on there? In the first line, we're adding 1 day to January 29th, which results in 2014-01-30; then we add 1 month, so we get 2014-02-30, which then adjusts down to 2014-02-28. In the second example, we add 1 month *first*, where we get 2014-02-29, which adjusts down to 2014-02-28, and *then* add 1 day, which results in 2014-03-01. One design principle that helps in this case is that, in the presence of multiple Periods, the operations will be ordered by the Periods' *types*, not their value or positional order; this means Year will always be added first, then Month, then Week, etc. Hence the following *does* result in associativity and Just Works:

```
julia> Date(2014,1,29) + Dates.Day(1) + Dates.Month(1)
2014-03-01

julia> Date(2014,1,29) + Dates.Month(1) + Dates.Day(1)
2014-03-01
```

Tricky? Perhaps. What is an innocent Dates user to do? The bottom line is to be aware that explicitly forcing a certain associativity, when dealing with months, may lead to some unexpected results, but otherwise, everything should work as expected. Thankfully, that's pretty much the extent of the odd cases in date-period arithmetic when dealing with time in UT (avoiding the "joys" of dealing with daylight savings, leap seconds, etc.).

As a bonus, all period arithmetic objects work directly with ranges:

```
julia> dr = Date(2014,1,29):Day(1):Date(2014,2,3)
Date("2014-01-29"):Day(1):Date("2014-02-03")

julia> collect(dr)
6-element Vector{Date}:
 2014-01-29
 2014-01-30
 2014-01-31
 2014-02-01
 2014-02-02
 2014-02-03

julia> dr = Date(2014,1,29):Dates.Month(1):Date(2014,07,29)
Date("2014-01-29"):Month(1):Date("2014-07-29")

julia> collect(dr)
7-element Vector{Date}:
 2014-01-29
 2014-02-28
 2014-03-29
 2014-04-29
 2014-05-29
 2014-06-29
 2014-07-29
```

66.6 Adjuster Functions

As convenient as date-period arithmetic is, often the kinds of calculations needed on dates take on a *calendrical* or *temporal* nature rather than a fixed number of periods. Holidays are a perfect example; most follow rules such as "Memorial Day = Last Monday of May", or "Thanksgiving = 4th Thursday of November". These kinds

of temporal expressions deal with rules relative to the calendar, like first or last of the month, next Tuesday, or the first and third Wednesdays, etc.

The Dates module provides the *adjuster* API through several convenient methods that aid in simply and succinctly expressing temporal rules. The first group of adjuster methods deal with the first and last of weeks, months, quarters, and years. They each take a single TimeType as input and return or *adjust to* the first or last of the desired period relative to the input.

```
julia> Dates.firstdayofweek(Date(2014,7,16)) # Adjusts the input to the Monday of the input's week
2014-07-14

julia> Dates.lastdayofmonth(Date(2014,7,16)) # Adjusts to the last day of the input's month
2014-07-31

julia> Dates.lastdayofquarter(Date(2014,7,16)) # Adjusts to the last day of the input's quarter
2014-09-30
```

The next two higher-order methods, tonext, and toprev, generalize working with temporal expressions by taking a DateFunction as first argument, along with a starting TimeType. A DateFunction is just a function, usually anonymous, that takes a single TimeType as input and returns a Bool, true indicating a satisfied adjustment criterion. For example:

```
julia> istuesday = x->Dates.dayofweek(x) == Dates.Tuesday; # Returns true if the day of the week of
↪  x is Tuesday

julia> Dates.tonext(istuesday, Date(2014,7,13)) # 2014-07-13 is a Sunday
2014-07-15

julia> Dates.tonext(Date(2014,7,13), Dates.Tuesday) # Convenience method provided for day of the
↪  week adjustments
2014-07-15
```

This is useful with the do-block syntax for more complex temporal expressions:

```
julia> Dates.tonext(Date(2014,7,13)) do x
           # Return true on the 4th Thursday of November (Thanksgiving)
           Dates.dayofweek(x) == Dates.Thursday &&
           Dates.dayofweekofmonth(x) == 4 &&
           Dates.month(x) == Dates.November
       end
2014-11-27
```

The Base.filter method can be used to obtain all valid dates/moments in a specified range:

```
# Pittsburgh street cleaning; Every 2nd Tuesday from April to November
# Date range from January 1st, 2014 to January 1st, 2015
julia> dr = Dates.Date(2014):Day(1):Dates.Date(2015);

julia> filter(dr) do x
           Dates.dayofweek(x) == Dates.Tue &&
           Dates.April <= Dates.month(x) <= Dates.Nov &&
           Dates.dayofweekofmonth(x) == 2
```

```
      end
8-element Vector{Date}:
 2014-04-08
 2014-05-13
 2014-06-10
 2014-07-08
 2014-08-12
 2014-09-09
 2014-10-14
 2014-11-11
```

Additional examples and tests are available in `stdlib/Dates/test/adjusters.jl`.

66.7 Period Types

Periods are a human view of discrete, sometimes irregular durations of time. Consider 1 month; it could represent, in days, a value of 28, 29, 30, or 31 depending on the year and month context. Or a year could represent 365 or 366 days in the case of a leap year. `Period` types are simple `Int64` wrappers and are constructed by wrapping any `Int64` convertible type, i.e. `Year(1)` or `Month(3.0)`. Arithmetic between `Period` of the same type behave like integers, and limited `Period-Real` arithmetic is available. You can extract the underlying integer with `Dates.value`.

```
julia> y1 = Dates.Year(1)
1 year

julia> y2 = Dates.Year(2)
2 years

julia> y3 = Dates.Year(10)
10 years

julia> y1 + y2
3 years

julia> div(y3,y2)
5

julia> y3 - y2
8 years

julia> y3 % y2
0 years

julia> div(y3,3) # mirrors integer division
3 years

julia> Dates.value(Dates.Millisecond(10))
10
```

Representing periods or durations that are not integer multiples of the basic types can be achieved with the `Dates.CompoundPeriod` type. Compound periods may be constructed manually from simple `Period` types. Additionally, the `canonicalize` function can be used to break down a period into a `Dates.CompoundPeriod`.

This is particularly useful to convert a duration, e.g., a difference of two DateTime, into a more convenient representation.

```
julia> cp = Dates.CompoundPeriod(Day(1),Minute(1))
1 day, 1 minute

julia> t1 = DateTime(2018,8,8,16,58,00)
2018-08-08T16:58:00

julia> t2 = DateTime(2021,6,23,10,00,00)
2021-06-23T10:00:00

julia> canonicalize(t2-t1) # creates a CompoundPeriod
149 weeks, 6 days, 17 hours, 2 minutes
```

66.8 Rounding

Date and DateTime values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with floor, ceil, or round:

```
julia> floor(Date(1985, 8, 16), Dates.Month)
1985-08-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:45:00

julia> round(DateTime(2016, 8, 6, 20, 15), Dates.Day)
2016-08-07T00:00:00
```

Unlike the numeric round method, which breaks ties toward the even number by default, the TimeTyperound method uses the RoundNearestTiesUp rounding mode. (It's difficult to guess what breaking ties to nearest "even" TimeType would entail.) Further details on the available RoundingMode s can be found in the API reference.

Rounding should generally behave as expected, but there are a few cases in which the expected behaviour is not obvious.

Rounding Epoch

In many cases, the resolution specified for rounding (e.g., Dates.Second(30)) divides evenly into the next largest period (in this case, Dates.Minute(1)). But rounding behaviour in cases in which this is not true may lead to confusion. What is the expected result of rounding a DateTime to the nearest 10 hours?

```
julia> round(DateTime(2016, 7, 17, 11, 55), Dates.Hour(10))
2016-07-17T12:00:00
```

That may seem confusing, given that the hour (12) is not divisible by 10. The reason that 2016-07-17T12:00:00 was chosen is that it is 17,676,660 hours after 0000-01-01T00:00:00, and 17,676,660 is divisible by 10.

As Julia Date and DateTime values are represented according to the ISO 8601 standard, 0000-01-01T00:00:00 was chosen as base (or "rounding epoch") from which to begin the count of days (and milliseconds) used in rounding calculations. (Note that this differs slightly from Julia's internal representation of Date s using Rata Die

notation; but since the ISO 8601 standard is most visible to the end user, `0000-01-01T00:00:00` was chosen as the rounding epoch instead of the `0000-12-31T00:00:00` used internally to minimize confusion.)

The only exception to the use of `0000-01-01T00:00:00` as the rounding epoch is when rounding to weeks. Rounding to the nearest week will always return a Monday (the first day of the week as specified by ISO 8601). For this reason, we use `0000-01-03T00:00:00` (the first day of the first week of year 0000, as defined by ISO 8601) as the base when rounding to a number of weeks.

Here is a related case in which the expected behaviour is not necessarily obvious: What happens when we round to the nearest `P(2)`, where `P` is a `Period` type? In some cases (specifically, when `P <: Dates.TimePeriod`) the answer is clear:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Hour(2))
2016-07-17T08:00:00

julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Minute(2))
2016-07-17T08:56:00
```

This seems obvious, because two of each of these periods still divides evenly into the next larger order period. But in the case of two months (which still divides evenly into one year), the answer may be surprising:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Month(2))
2016-07-01T00:00:00
```

Why round to the first day in July, even though it is month 7 (an odd number)? The key is that months are 1-indexed (the first month is assigned 1), unlike hours, minutes, seconds, and milliseconds (the first of which are assigned 0).

This means that rounding a `DateTime` to an even multiple of seconds, minutes, hours, or years (because the ISO 8601 specification includes a year zero) will result in a `DateTime` with an even value in that field, while rounding a `DateTime` to an even multiple of months will result in the months field having an odd value. Because both months and years may contain an irregular number of days, whether rounding to an even number of days will result in an even value in the days field is uncertain.

See the API reference for additional information on methods exported from the `Dates` module.

Chapter 67

API reference

67.1 Dates and Time Types

`Dates.Period` – Type.

```
Period
Year
Quarter
Month
Week
Day
Hour
Minute
Second
Millisecond
Microsecond
Nanosecond
```

`Period` types represent discrete, human representations of time.

`Dates.CompoundPeriod` – Type.

```
CompoundPeriod
```

A `CompoundPeriod` is useful for expressing time periods that are not a fixed multiple of smaller periods. For example, ”a year and a day” is not a fixed number of days, but can be expressed using a `CompoundPeriod`. In fact, a `CompoundPeriod` is automatically generated by addition of different period types, e.g. `Year(1) + Day(1)` produces a `CompoundPeriod` result.

`Dates.Instant` – Type.

```
Instant
```

`Instant` types represent integer-based, machine representations of time as continuous timelines starting from an epoch.

`Dates.UTInstant` – Type.

```
UTInstant{T}
```

The `UTInstant` represents a machine timeline based on UT time (1 day = one revolution of the earth). The `T` is a `Period` parameter that indicates the resolution or precision of the instant.

`Dates.TimeType` – Type.

```
TimeType
```

`TimeType` types wrap `Instant` machine instances to provide human representations of the machine instant. `Time`, `DateTime` and `Date` are subtypes of `TimeType`.

`Dates.DateTime` – Type.

```
DateTime
```

`DateTime` wraps a `UTInstant{Millisecond}` and interprets it according to the proleptic Gregorian calendar.

`Dates.Date` – Type.

```
Date
```

`Date` wraps a `UTInstant{Day}` and interprets it according to the proleptic Gregorian calendar.

`Dates.Time` – Type.

```
Time
```

`Time` wraps a `Nanosecond` and represents a specific moment in a 24-hour day.

`Dates.TimeZone` – Type.

```
TimeZone
```

Geographic zone generally based on longitude determining what the time is at a certain location. Some time zones observe daylight savings (eg EST -> EDT). For implementations and more support, see the `TimeZones.jl` package

`Dates.UTC` – Type.

```
UTC
```

UTC, or Coordinated Universal Time, is the TimeZone from which all others are measured. It is associated with the time at 0° longitude. It is not adjusted for daylight savings.

67.2 Dates Functions

Dates.DateTime – Method.

```
DateTime(y, [m, d, h, mi, s, ms]) -> DateTime
```

Construct a DateTime type by parts. Arguments must be convertible to Int64.

Dates.DateTime – Method.

```
DateTime(periods::Period...) -> DateTime
```

Construct a DateTime type by Period type parts. Arguments may be in any order. DateTime parts not provided will default to the value of Dates.default(period).

Dates.DateTime – Method.

```
DateTime(f::Function, y[, m, d, h, mi, s]; step=Day(1), limit=10000) -> DateTime
```

Create a DateTime through the adjuster API. The starting point will be constructed from the provided y, m, d... arguments, and will be adjusted until f::Function returns true. The step size in adjusting can be provided manually through the step keyword. limit provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that f::Function is never satisfied).

Examples

```
julia> DateTime(dt -> second(dt) == 40, 2010, 10, 20, 10; step = Second(1))
2010-10-20T10:00:40

julia> DateTime(dt -> hour(dt) == 20, 2010, 10, 20, 10; step = Hour(1), limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

Dates.DateTime – Method.

```
DateTime(dt::Date) -> DateTime
```

Convert a Date to a DateTime. The hour, minute, second, and millisecond parts of the new DateTime are assumed to be zero.

`Dates.DateTime` – Method.

```
DateTime(dt::AbstractString, format::AbstractString; locale="english") -> DateTime
```

Construct a `DateTime` by parsing the `dt` date time string following the pattern given in the `format` string (see `DateFormat` for syntax).

Note

This method creates a `DateFormat` object each time it is called. It is recommended that you create a `DateFormat` object instead and use that as the second argument to avoid performance loss when using the same format repeatedly.

Example

```
julia> DateTime("2020-01-01", "yyyy-mm-dd")
2020-01-01T00:00:00

julia> a = ("2020-01-01", "2020-01-02");

julia> [DateTime(d, dateformat"yyyy-mm-dd") for d ∈ a] # preferred
2-element Vector{DateTime}:
 2020-01-01T00:00:00
 2020-01-02T00:00:00
```

`Dates.format` – Method.

```
format(dt::TimeType, format::AbstractString; locale="english") -> AbstractString
```

Construct a string by using a `TimeType` object and applying the provided `format`. The following character codes can be used to construct the `format` string:

Code	Examples	Comment
`y`	6	Numeric year with a fixed width
`Y`	1996	Numeric year with a minimum width
`m`	1, 12	Numeric month with a minimum width
`u`	Jan	Month name shortened to 3-chars according to the `locale`
`U`	January	Full month name according to the `locale` keyword
`d`	1, 31	Day of the month with a minimum width
`H`	0, 23	Hour (24-hour clock) with a minimum width
`M`	0, 59	Minute with a minimum width
`S`	0, 59	Second with a minimum width
`s`	000, 500	Millisecond with a minimum width of 3
`e`	Mon, Tue	Abbreviated days of the week
`E`	Monday	Full day of week name

The number of sequential code characters indicate the width of the code. A format of `yyyy-mm` specifies that the code `y` should have a width of four while `m` a width of two. Codes that yield numeric digits have

an associated mode: fixed-width or minimum-width. The fixed-width mode left-pads the value with zeros when it is shorter than the specified width and truncates the value when longer. Minimum-width mode works the same as fixed-width except that it does not truncate values longer than the width.

When creating a `format` you can use any non-code characters as a separator. For example to generate the string "1996-01-15T00:00:00" you could use `format`: "yyyy-mm-ddTHH:MM:SS". Note that if you need to use a code character as a literal you can use the escape character backslash. The string "1996y01m" can be produced with the format "yyyy\ymm\m".

`Dates.DateFormat` – Type.

```
DateFormat(format::AbstractString, locale="english") -> DateFormat
```

Construct a date formatting object that can be used for parsing date strings or formatting a date object as a string. The following character codes can be used to construct the `format` string:

Code	Matches	Comment
`Y`	1996, 96	Returns year of 1996, 0096
`y`	1996, 96	Same as `Y` on `parse` but discards excess digits on `format`
`m`	1, 01	Matches 1 or 2-digit months
`u`	Jan	Matches abbreviated months according to the `locale` keyword
`U`	January	Matches full month names according to the `locale` keyword
`d`	1, 01	Matches 1 or 2-digit days
`H`	00	Matches hours (24-hour clock)
`I`	00	For outputting hours with 12-hour clock
`M`	00	Matches minutes
`S`	00	Matches seconds
`s`	.500	Matches milliseconds
`e`	Mon, Tues	Matches abbreviated days of the week
`E`	Monday	Matches full name days of the week
`p`	AM	Matches AM/PM (case-insensitive)
`yyyymmdd`	19960101	Matches fixed-width year, month, and day

Characters not listed above are normally treated as delimiters between date and time slots. For example a `dt` string of "1996-01-15T00:00:00.0" would have a `format` string like "y-m-dTH:M:S.s". If you need to use a code character as a delimiter you can escape it using backslash. The date "1995y01m" would have the format "y\ym\m".

Note that 12:00AM corresponds 00:00 (midnight), and 12:00PM corresponds to 12:00 (noon). When parsing a time with a `p` specifier, any hour (either `H` or `I`) is interpreted as as a 12-hour clock, so the `I` code is mainly useful for output.

Creating a DateFormat object is expensive. Whenever possible, create it once and use it many times or try the `dateformat""` string macro. Using this macro creates the DateFormat object once at macro expansion time and reuses it later. There are also several pre-defined formatters, listed later.

See `DateTime` and `format` for how to use a DateFormat object to parse and write Date strings respectively.

`Dates.@dateformat_str` – Macro.

```
dateformat"Y-m-d H:M:S"
```

Create a DateFormat object. Similar to DateFormat("Y-m-d H:M:S") but creates the DateFormat object once during macro expansion.

See DateFormat for details about format specifiers.

Dates.DateTime – Method.

```
DateTime(dt::AbstractString, df::DateFormat=ISODateTimeFormat) -> DateTime
```

Construct a DateTime by parsing the dt date time string following the pattern given in the DateFormat object, or dateformat"yyyy-mm-dd\THH:MM:SS.s" if omitted.

Similar to DateTime(::AbstractString, ::AbstractString) but more efficient when repeatedly parsing similarly formatted date time strings with a pre-created DateFormat object.

Dates.Date – Method.

```
Date(y, [m, d]) -> Date
```

Construct a Date type by parts. Arguments must be convertible to Int64.

Dates.Date – Method.

```
Date(period::Period...) -> Date
```

Construct a Date type by Period type parts. Arguments may be in any order. Date parts not provided will default to the value of Dates.default(period).

Dates.Date – Method.

```
Date(f::Function, y[, m, d]; step=Day(1), limit=10000) -> Date
```

Create a Date through the adjuster API. The starting point will be constructed from the provided y, m, d arguments, and will be adjusted until f::Function returns true. The step size in adjusting can be provided manually through the step keyword. limit provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (given that f::Function is never satisfied).

Examples

```
julia> Date(date -> week(date) == 20, 2010, 01, 01)
2010-05-17

julia> Date(date -> year(date) == 2010, 2000, 01, 01)
2010-01-01
```

```
julia> Date(date -> month(date) == 10, 2000, 01, 01; limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

`Dates.Date` – Method.

```
Date(dt::DateTime) -> Date
```

Convert a `DateTime` to a `Date`. The hour, minute, second, and millisecond parts of the `DateTime` are truncated, so only the year, month and day parts are used in construction.

`Dates.Date` – Method.

```
Date(d::AbstractString, format::AbstractString; locale="english") -> Date
```

Construct a `Date` by parsing the `d` date string following the pattern given in the `format` string (see `DateFormat` for syntax).

> **Note**
>
> This method creates a `DateFormat` object each time it is called. It is recommended that you create a `DateFormat` object instead and use that as the second argument to avoid performance loss when using the same format repeatedly.

Example

```
julia> Date("2020-01-01", "yyyy-mm-dd")
2020-01-01

julia> a = ("2020-01-01", "2020-01-02");

julia> [Date(d, dateformat"yyyy-mm-dd") for d ∈ a] # preferred
2-element Vector{Date}:
 2020-01-01
 2020-01-02
```

`Dates.Date` – Method.

```
Date(d::AbstractString, df::DateFormat=ISODateFormat) -> Date
```

Construct a `Date` by parsing the `d` date string following the pattern given in the `DateFormat` object, or dateformat"yyyy-mm-dd" if omitted.

Similar to `Date(::AbstractString, ::AbstractString)` but more efficient when repeatedly parsing similarly formatted date strings with a pre-created `DateFormat` object.

`Dates.Time` – Method.

```
Time(h, [mi, s, ms, us, ns]) -> Time
```

Construct a `Time` type by parts. Arguments must be convertible to `Int64`.

`Dates.Time` – Method.

```
Time(period::TimePeriod...) -> Time
```

Construct a `Time` type by `Period` type parts. Arguments may be in any order. `Time` parts not provided will default to the value of `Dates.default(period)`.

`Dates.Time` – Method.

```
Time(f::Function, h, mi=0; step::Period=Second(1), limit::Int=10000)
Time(f::Function, h, mi, s; step::Period=Millisecond(1), limit::Int=10000)
Time(f::Function, h, mi, s, ms; step::Period=Microsecond(1), limit::Int=10000)
Time(f::Function, h, mi, s, ms, us; step::Period=Nanosecond(1), limit::Int=10000)
```

Create a `Time` through the adjuster API. The starting point will be constructed from the provided `h, mi, s, ms, us` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `f::Function` is never satisfied). Note that the default step will adjust to allow for greater precision for the given arguments; i.e. if hour, minute, and second arguments are provided, the default step will be `Millisecond(1)` instead of `Second(1)`.

Examples

```
julia> Time(t -> minute(t) == 30, 20)
20:30:00

julia> Time(t -> minute(t) == 0, 20)
20:00:00

julia> Time(t -> hour(t) == 10, 3; limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

`Dates.Time` – Method.

```
Time(dt::DateTime) -> Time
```

Convert a `DateTime` to a `Time`. The hour, minute, second, and millisecond parts of the `DateTime` are used to create the new `Time`. Microsecond and nanoseconds are zero by default.

`Dates.Time` – Method.

```
Time(t::AbstractString, format::AbstractString; locale="english") -> Time
```

Construct a `Time` by parsing the `t` time string following the pattern given in the `format` string (see `DateFormat` for syntax).

Note

This method creates a `DateFormat` object each time it is called. It is recommended that you create a `DateFormat` object instead and use that as the second argument to avoid performance loss when using the same format repeatedly.

Example

```
julia> Time("12:34pm", "HH:MMp")
12:34:00

julia> a = ("12:34pm", "2:34am");

julia> [Time(d, dateformat"HH:MMp") for d ∈ a] # preferred
2-element Vector{Time}:
 12:34:00
 02:34:00
```

`Dates.Time` – Method.

```
Time(t::AbstractString, df::DateFormat=ISOTimeFormat) -> Time
```

Construct a `Time` by parsing the `t` date time string following the pattern given in the `DateFormat` object, or dateformat"HH:MM:SS.s" if omitted.

Similar to `Time(::AbstractString, ::AbstractString)` but more efficient when repeatedly parsing similarly formatted time strings with a pre-created `DateFormat` object.

`Dates.now` – Method.

```
now() -> DateTime
```

Return a `DateTime` corresponding to the user's system time including the system timezone locale.

`Dates.now` – Method.

```
now(::Type{UTC}) -> DateTime
```

Return a `DateTime` corresponding to the user's system time as UTC/GMT. For other time zones, see the TimeZones.jl package.

Example

```
julia> now(UTC)
2023-01-04T10:52:24.864
```

`Base.eps` – Method.

```
eps(::Type{DateTime}) -> Millisecond
eps(::Type{Date}) -> Day
eps(::Type{Time}) -> Nanosecond
eps(::TimeType) -> Period
```

Return the smallest unit value supported by the `TimeType`.

Examples

```
julia> eps(DateTime)
1 millisecond

julia> eps(Date)
1 day

julia> eps(Time)
1 nanosecond
```

Accessor Functions

`Dates.year` – Function.

```
year(dt::TimeType) -> Int64
```

The year of a `Date` or `DateTime` as an `Int64`.

`Dates.month` – Function.

```
month(dt::TimeType) -> Int64
```

The month of a `Date` or `DateTime` as an `Int64`.

`Dates.week` – Function.

```
week(dt::TimeType) -> Int64
```

Return the ISO week date of a `Date` or `DateTime` as an `Int64`. Note that the first week of a year is the week that contains the first Thursday of the year, which can result in dates prior to January 4th being in the last week of the previous year. For example, `week(Date(2005, 1, 1))` is the 53rd week of 2004.

Examples

```
julia> week(Date(1989, 6, 22))
25

julia> week(Date(2005, 1, 1))
53

julia> week(Date(2004, 12, 31))
53
```

Dates.day – Function.

```
day(dt::TimeType) -> Int64
```

The day of month of a Date or DateTime as an Int64.

Dates.hour – Function.

```
hour(dt::DateTime) -> Int64
```

The hour of day of a DateTime as an Int64.

```
hour(t::Time) -> Int64
```

The hour of a Time as an Int64.

Dates.minute – Function.

```
minute(dt::DateTime) -> Int64
```

The minute of a DateTime as an Int64.

```
minute(t::Time) -> Int64
```

The minute of a Time as an Int64.

Dates.second – Function.

```
second(dt::DateTime) -> Int64
```

The second of a DateTime as an Int64.

```
second(t::Time) -> Int64
```

The second of a Time as an Int64.

`Dates.millisecond` – Function.

```
millisecond(dt::DateTime) -> Int64
```

The millisecond of a `DateTime` as an `Int64`.

```
millisecond(t::Time) -> Int64
```

The millisecond of a `Time` as an `Int64`.

`Dates.microsecond` – Function.

```
microsecond(t::Time) -> Int64
```

The microsecond of a `Time` as an `Int64`.

`Dates.nanosecond` – Function.

```
nanosecond(t::Time) -> Int64
```

The nanosecond of a `Time` as an `Int64`.

`Dates.Year` – Method.

```
Year(v)
```

Construct a `Year` object with the given v value. Input must be losslessly convertible to an `Int64`.

`Dates.Month` – Method.

```
Month(v)
```

Construct a `Month` object with the given v value. Input must be losslessly convertible to an `Int64`.

`Dates.Week` – Method.

```
Week(v)
```

Construct a `Week` object with the given v value. Input must be losslessly convertible to an `Int64`.

`Dates.Day` – Method.

```
Day(v)
```

Construct a Day object with the given v value. Input must be losslessly convertible to an `Int64`.

`Dates.Hour` – Method.

```
Hour(dt::DateTime) -> Hour
```

The hour part of a DateTime as a `Hour`.

`Dates.Minute` – Method.

```
Minute(dt::DateTime) -> Minute
```

The minute part of a DateTime as a `Minute`.

`Dates.Second` – Method.

```
Second(dt::DateTime) -> Second
```

The second part of a DateTime as a Second.

`Dates.Millisecond` – Method.

```
Millisecond(dt::DateTime) -> Millisecond
```

The millisecond part of a DateTime as a `Millisecond`.

`Dates.Microsecond` – Method.

```
Microsecond(dt::Time) -> Microsecond
```

The microsecond part of a Time as a `Microsecond`.

`Dates.Nanosecond` – Method.

```
Nanosecond(dt::Time) -> Nanosecond
```

The nanosecond part of a Time as a Nanosecond.

`Dates.yearmonth` – Function.

```
yearmonth(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the year and month parts of a `Date` or `DateTime`.

`Dates.monthday` – Function.

```
monthday(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the month and day parts of a `Date` or `DateTime`.

`Dates.yearmonthday` – Function.

```
yearmonthday(dt::TimeType) -> (Int64, Int64, Int64)
```

Simultaneously return the year, month and day parts of a `Date` or `DateTime`.

Query Functions

`Dates.dayname` – Function.

```
dayname(dt::TimeType; locale="english") -> String
dayname(day::Integer; locale="english") -> String
```

Return the full day name corresponding to the day of the week of the `Date` or `DateTime` in the given `locale`. Also accepts `Integer`.

Examples

```
julia> dayname(Date("2000-01-01"))
"Saturday"

julia> dayname(4)
"Thursday"
```

`Dates.dayabbr` – Function.

```
dayabbr(dt::TimeType; locale="english") -> String
dayabbr(day::Integer; locale="english") -> String
```

Return the abbreviated name corresponding to the day of the week of the `Date` or `DateTime` in the given `locale`. Also accepts `Integer`.

Examples

```
julia> dayabbr(Date("2000-01-01"))
"Sat"

julia> dayabbr(3)
"Wed"
```

`Dates.dayofweek` – Function.

```
dayofweek(dt::TimeType) -> Int64
```

Return the day of the week as an `Int64` with `1 = Monday, 2 = Tuesday, etc.`.

Examples

```
julia> dayofweek(Date("2000-01-01"))
6
```

`Dates.dayofmonth` – Function.

```
dayofmonth(dt::TimeType) -> Int64
```

The day of month of a `Date` or `DateTime` as an `Int64`.

`Dates.dayofweekofmonth` – Function.

```
dayofweekofmonth(dt::TimeType) -> Int
```

For the day of week of `dt`, return which number it is in `dt`'s month. So if the day of the week of `dt` is Monday, then `1 = First Monday of the month, 2 = Second Monday of the month, etc.` In the range 1:5.

Examples

```
julia> dayofweekofmonth(Date("2000-02-01"))
1

julia> dayofweekofmonth(Date("2000-02-08"))
2

julia> dayofweekofmonth(Date("2000-02-15"))
3
```

`Dates.daysofweekinmonth` – Function.

```
daysofweekinmonth(dt::TimeType) -> Int
```

For the day of week of `dt`, return the total number of that day of the week in `dt`'s month. Returns 4 or 5. Useful in temporal expressions for specifying the last day of a week in a month by including `dayofweekofmonth(dt) == daysofweekinmonth(dt)` in the adjuster function.

Examples

```
julia> daysofweekinmonth(Date("2005-01-01"))
5

julia> daysofweekinmonth(Date("2005-01-04"))
4
```

`Dates.monthname` – Function.

```
monthname(dt::TimeType; locale="english") -> String
monthname(month::Integer, locale="english") -> String
```

Return the full name of the month of the `Date` or `DateTime` or `Integer` in the given `locale`.

Examples

```
julia> monthname(Date("2005-01-04"))
"January"

julia> monthname(2)
"February"
```

`Dates.monthabbr` – Function.

```
monthabbr(dt::TimeType; locale="english") -> String
monthabbr(month::Integer, locale="english") -> String
```

Return the abbreviated month name of the `Date` or `DateTime` or `Integer` in the given `locale`.

Examples

```
julia> monthabbr(Date("2005-01-04"))
"Jan"

julia> monthabbr(2)
"Feb"
```

`Dates.daysinmonth` – Function.

```
daysinmonth(dt::TimeType) -> Int
```

Return the number of days in the month of `dt`. Value will be 28, 29, 30, or 31.

Examples

```
julia> daysinmonth(Date("2000-01"))
31

julia> daysinmonth(Date("2001-02"))
28

julia> daysinmonth(Date("2000-02"))
29
```

`Dates.isleapyear` – Function.

```
isleapyear(dt::TimeType) -> Bool
```

Return `true` if the year of `dt` is a leap year.

Examples

```
julia> isleapyear(Date("2004"))
true

julia> isleapyear(Date("2005"))
false
```

`Dates.dayofyear` – Function.

```
dayofyear(dt::TimeType) -> Int
```

Return the day of the year for `dt` with January 1st being day 1.

`Dates.daysinyear` – Function.

```
daysinyear(dt::TimeType) -> Int
```

Return 366 if the year of `dt` is a leap year, otherwise return 365.

Examples

```
julia> daysinyear(1999)
365

julia> daysinyear(2000)
366
```

`Dates.quarterofyear` – Function.

```
quarterofyear(dt::TimeType) -> Int
```

Return the quarter that `dt` resides in. Range of value is 1:4.

`Dates.dayofquarter` – Function.

```
dayofquarter(dt::TimeType) -> Int
```

Return the day of the current quarter of `dt`. Range of value is 1:92.

Adjuster Functions

`Base.trunc` – Method.

```
trunc(dt::TimeType, ::Type{Period}) -> TimeType
```

Truncates the value of `dt` according to the provided `Period` type.

Examples

```
julia> trunc(DateTime("1996-01-01T12:30:00"), Day)
1996-01-01T00:00:00
```

`Dates.firstdayofweek` – Function.

```
firstdayofweek(dt::TimeType) -> TimeType
```

Adjusts `dt` to the Monday of its week.

Examples

```
julia> firstdayofweek(DateTime("1996-01-05T12:30:00"))
1996-01-01T00:00:00
```

`Dates.lastdayofweek` – Function.

```
lastdayofweek(dt::TimeType) -> TimeType
```

Adjusts `dt` to the Sunday of its week.

Examples

```
julia> lastdayofweek(DateTime("1996-01-05T12:30:00"))
1996-01-07T00:00:00
```

`Dates.firstdayofmonth` – Function.

```
firstdayofmonth(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its month.

Examples

```
julia> firstdayofmonth(DateTime("1996-05-20"))
1996-05-01T00:00:00
```

`Dates.lastdayofmonth` – Function.

```
lastdayofmonth(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its month.

Examples

```
julia> lastdayofmonth(DateTime("1996-05-20"))
1996-05-31T00:00:00
```

`Dates.firstdayofyear` – Function.

```
firstdayofyear(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its year.

Examples

```
julia> firstdayofyear(DateTime("1996-05-20"))
1996-01-01T00:00:00
```

`Dates.lastdayofyear` – Function.

```
lastdayofyear(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its year.

Examples

```
julia> lastdayofyear(DateTime("1996-05-20"))
1996-12-31T00:00:00
```

`Dates.firstdayofquarter` – Function.

```
firstdayofquarter(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its quarter.

Examples

```
julia> firstdayofquarter(DateTime("1996-05-20"))
1996-04-01T00:00:00

julia> firstdayofquarter(DateTime("1996-08-20"))
1996-07-01T00:00:00
```

`Dates.lastdayofquarter` – Function.

```
lastdayofquarter(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its quarter.

Examples

```
julia> lastdayofquarter(DateTime("1996-05-20"))
1996-06-30T00:00:00

julia> lastdayofquarter(DateTime("1996-08-20"))
1996-09-30T00:00:00
```

`Dates.tonext` – Method.

```
tonext(dt::TimeType, dow::Int; same::Bool=false) -> TimeType
```

Adjusts `dt` to the next day of week corresponding to `dow` with `1 = Monday, 2 = Tuesday, etc`. Setting `same=true` allows the current `dt` to be considered as the next `dow`, allowing for no adjustment to occur.

`Dates.toprev` – Method.

```
toprev(dt::TimeType, dow::Int; same::Bool=false) -> TimeType
```

Adjusts `dt` to the previous day of week corresponding to `dow` with `1 = Monday, 2 = Tuesday, etc`. Setting `same=true` allows the current `dt` to be considered as the previous `dow`, allowing for no adjustment to occur.

`Dates.tofirst` – Function.

```
tofirst(dt::TimeType, dow::Int; of=Month) -> TimeType
```

Adjusts `dt` to the first `dow` of its month. Alternatively, `of=Year` will adjust to the first `dow` of the year.

`Dates.tolast` – Function.

```
tolast(dt::TimeType, dow::Int; of=Month) -> TimeType
```

Adjusts `dt` to the last `dow` of its month. Alternatively, `of=Year` will adjust to the last `dow` of the year.

`Dates.tonext` – Method.

```
tonext(func::Function, dt::TimeType; step=Day(1), limit=10000, same=false) -> TimeType
```

Adjusts `dt` by iterating at most `limit` iterations by `step` increments until `func` returns `true`. `func` must take a single `TimeType` argument and return a `Bool`. `same` allows `dt` to be considered in satisfying `func`.

`Dates.toprev` – Method.

```
toprev(func::Function, dt::TimeType; step=Day(-1), limit=10000, same=false) -> TimeType
```

Adjusts `dt` by iterating at most `limit` iterations by `step` increments until `func` returns `true`. `func` must take a single `TimeType` argument and return a `Bool`. `same` allows `dt` to be considered in satisfying `func`.

Periods

`Dates.Period` – Method.

```
Year(v)
Quarter(v)
Month(v)
Week(v)
Day(v)
Hour(v)
Minute(v)
Second(v)
Millisecond(v)
Microsecond(v)
Nanosecond(v)
```

Construct a `Period` type with the given `v` value. Input must be losslessly convertible to an `Int64`.

`Dates.CompoundPeriod` – Method.

```
CompoundPeriod(periods) -> CompoundPeriod
```

Construct a `CompoundPeriod` from a `Vector` of `Period`s. All `Period`s of the same type will be added together.

Examples

```
julia> Dates.CompoundPeriod(Dates.Hour(12), Dates.Hour(13))
25 hours

julia> Dates.CompoundPeriod(Dates.Hour(-1), Dates.Minute(1))
-1 hour, 1 minute

julia> Dates.CompoundPeriod(Dates.Month(1), Dates.Week(-2))
1 month, -2 weeks

julia> Dates.CompoundPeriod(Dates.Minute(50000))
50000 minutes
```

`Dates.canonicalize` – Function.

```
canonicalize(::CompoundPeriod) -> CompoundPeriod
```

Reduces the `CompoundPeriod` into its canonical form by applying the following rules:

- Any `Period` large enough be partially representable by a coarser `Period` will be broken into multiple `Period`s (eg. `Hour(30)` becomes `Day(1) + Hour(6)`)
- `Period`s with opposite signs will be combined when possible (eg. `Hour(1) - Day(1)` becomes `-Hour(23)`)

Examples

```
julia> canonicalize(Dates.CompoundPeriod(Dates.Hour(12), Dates.Hour(13)))
1 day, 1 hour

julia> canonicalize(Dates.CompoundPeriod(Dates.Hour(-1), Dates.Minute(1)))
-59 minutes

julia> canonicalize(Dates.CompoundPeriod(Dates.Month(1), Dates.Week(-2)))
1 month, -2 weeks

julia> canonicalize(Dates.CompoundPeriod(Dates.Minute(50000)))
4 weeks, 6 days, 17 hours, 20 minutes
```

`Dates.value` – Function.

```
Dates.value(x::Period) -> Int64
```

For a given period, return the value associated with that period. For example, `value(Millisecond(10))` returns 10 as an integer.

`Dates.default` – Function.

```
default(p::Period) -> Period
```

Return a sensible "default" value for the input Period by returning `T(1)` for Year, Month, and Day, and `T(0)` for Hour, Minute, Second, and Millisecond.

`Dates.periods` – Function.

```
Dates.periods(::CompoundPeriod) -> Vector{Period}
```

Return the `Vector` of `Period`s that comprise the given `CompoundPeriod`.

Julia 1.7

This function requires Julia 1.7 or later.

Rounding Functions

`Date` and `DateTime` values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with `floor`, `ceil`, or `round`.

`Base.floor` – Method.

```
floor(dt::TimeType, p::Period) -> TimeType
```

Return the nearest `Date` or `DateTime` less than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `floor(dt, Dates.Hour)` is a shortcut for `floor(dt, Dates.Hour(1))`.

```
julia> floor(Date(1985, 8, 16), Month)
1985-08-01

julia> floor(DateTime(2013, 2, 13, 0, 31, 20), Minute(15))
2013-02-13T00:30:00

julia> floor(DateTime(2016, 8, 6, 12, 0, 0), Day)
2016-08-06T00:00:00
```

`Base.ceil` – Method.

```
ceil(dt::TimeType, p::Period) -> TimeType
```

Return the nearest `Date` or `DateTime` greater than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `ceil(dt, Dates.Hour)` is a shortcut for `ceil(dt, Dates.Hour(1))`.

```
julia> ceil(Date(1985, 8, 16), Month)
1985-09-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Minute(15))
2013-02-13T00:45:00

julia> ceil(DateTime(2016, 8, 6, 12, 0, 0), Day)
2016-08-07T00:00:00
```

Base.round – Method.

```
round(dt::TimeType, p::Period, [r::RoundingMode]) -> TimeType
```

Return the Date or DateTime nearest to dt at resolution p. By default (RoundNearestTiesUp), ties (e.g., rounding 9:30 to the nearest hour) will be rounded up.

For convenience, p may be a type instead of a value: round(dt, Dates.Hour) is a shortcut for round(dt, Dates.Hour(1)).

```
julia> round(Date(1985, 8, 16), Month)
1985-08-01

julia> round(DateTime(2013, 2, 13, 0, 31, 20), Minute(15))
2013-02-13T00:30:00

julia> round(DateTime(2016, 8, 6, 12, 0, 0), Day)
2016-08-07T00:00:00
```

Valid rounding modes for round(::TimeType, ::Period, ::RoundingMode) are RoundNearestTiesUp (default), RoundDown (floor), and RoundUp (ceil).

Most Period values can also be rounded to a specified resolution:

Base.floor – Method.

```
floor(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> T
```

Round x down to the nearest multiple of precision. If x and precision are different subtypes of Period, the return value will have the same type as precision.

For convenience, precision may be a type instead of a value: floor(x, Dates.Hour) is a shortcut for floor(x, Dates.Hour(1)).

```
julia> floor(Day(16), Week)
2 weeks

julia> floor(Minute(44), Minute(15))
30 minutes

julia> floor(Hour(36), Day)
1 day
```

Rounding to a `precision` of `Month`s or `Year`s is not supported, as these `Period`s are of inconsistent length.

`Base.ceil` – Method.

```
ceil(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> T
```

Round `x` up to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`.

For convenience, `precision` may be a type instead of a value: `ceil(x, Dates.Hour)` is a shortcut for `ceil(x, Dates.Hour(1))`.

```
julia> ceil(Day(16), Week)
3 weeks

julia> ceil(Minute(44), Minute(15))
45 minutes

julia> ceil(Hour(36), Day)
2 days
```

Rounding to a `precision` of `Month`s or `Year`s is not supported, as these `Period`s are of inconsistent length.

`Base.round` – Method.

```
round(x::Period, precision::T, [r::RoundingMode]) where T <: Union{TimePeriod, Week, Day} -> T
```

Round `x` to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`. By default (`RoundNearestTiesUp`), ties (e.g., rounding 90 minutes to the nearest hour) will be rounded up.

For convenience, `precision` may be a type instead of a value: `round(x, Dates.Hour)` is a shortcut for `round(x, Dates.Hour(1))`.

```
julia> round(Day(16), Week)
2 weeks

julia> round(Minute(44), Minute(15))
45 minutes

julia> round(Hour(36), Day)
2 days
```

Valid rounding modes for `round(::Period, ::T, ::RoundingMode)` are `RoundNearestTiesUp` (default), `RoundDown` (`floor`), and `RoundUp` (`ceil`).

Rounding to a `precision` of `Month`s or `Year`s is not supported, as these `Period`s are of inconsistent length.

The following functions are not exported:

`Dates.floorceil` – Function.

```
floorceil(dt::TimeType, p::Period) -> (TimeType, TimeType)
```

Simultaneously return the `floor` and `ceil` of a `Date` or `DateTime` at resolution `p`. More efficient than calling both `floor` and `ceil` individually.

```
floorceil(x::Period, precision::T) where T <: Union{TimePeriod, Week, Day} -> (T, T)
```

Simultaneously return the `floor` and `ceil` of `Period` at resolution `p`. More efficient than calling both `floor` and `ceil` individually.

`Dates.epochdays2date` – Function.

```
epochdays2date(days) -> Date
```

Take the number of days since the rounding epoch (`0000-01-01T00:00:00`) and return the corresponding `Date`.

`Dates.epochms2datetime` – Function.

```
epochms2datetime(milliseconds) -> DateTime
```

Take the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`) and return the corresponding `DateTime`.

`Dates.date2epochdays` – Function.

```
date2epochdays(dt::Date) -> Int64
```

Take the given `Date` and return the number of days since the rounding epoch (`0000-01-01T00:00:00`) as an `Int64`.

`Dates.datetime2epochms` – Function.

```
datetime2epochms(dt::DateTime) -> Int64
```

Take the given `DateTime` and return the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`) as an `Int64`.

Conversion Functions

`Dates.today` – Function.

```
today() -> Date
```

Return the date portion of `now()`.

`Dates.unix2datetime` – Function.

```
unix2datetime(x) -> DateTime
```

Take the number of seconds since unix epoch `1970-01-01T00:00:00` and convert to the corresponding `DateTime`.

`Dates.datetime2unix` – Function.

```
datetime2unix(dt::DateTime) -> Float64
```

Take the given `DateTime` and return the number of seconds since the unix epoch `1970-01-01T00:00:00` as a `Float64`.

`Dates.julian2datetime` – Function.

```
julian2datetime(julian_days) -> DateTime
```

Take the number of Julian calendar days since epoch `-4713-11-24T12:00:00` and return the corresponding `DateTime`.

`Dates.datetime2julian` – Function.

```
datetime2julian(dt::DateTime) -> Float64
```

Take the given `DateTime` and return the number of Julian calendar days since the julian epoch `-4713-11-24T12:00:00` as a `Float64`.

`Dates.rata2datetime` – Function.

```
rata2datetime(days) -> DateTime
```

Take the number of Rata Die days since epoch `0000-12-31T00:00:00` and return the corresponding `DateTime`.

`Dates.datetime2rata` – Function.

```
datetime2rata(dt::TimeType) -> Int64
```

Return the number of Rata Die days since epoch from the given `Date` or `DateTime`.

Variable	Abbr.	Value (Int)
`Monday`	`Mon`	1
`Tuesday`	`Tue`	2
`Wednesday`	`Wed`	3
`Thursday`	`Thu`	4
`Friday`	`Fri`	5
`Saturday`	`Sat`	6
`Sunday`	`Sun`	7

Constants

Days of the Week:

Months of the Year:

Variable	Abbr.	Value (Int)
`January`	`Jan`	1
`February`	`Feb`	2
`March`	`Mar`	3
`April`	`Apr`	4
`May`	`May`	5
`June`	`Jun`	6
`July`	`Jul`	7
`August`	`Aug`	8
`September`	`Sep`	9
`October`	`Oct`	10
`November`	`Nov`	11
`December`	`Dec`	12

Common Date Formatters

`Dates.ISODateTimeFormat` – Constant.

```
Dates.ISODateTimeFormat
```

Describes the ISO8601 formatting for a date and time. This is the default value for `Dates.format` of a `DateTime`.

Example

```
julia> Dates.format(DateTime(2018, 8, 8, 12, 0, 43, 1), ISODateTimeFormat)
"2018-08-08T12:00:43.001"
```

`Dates.ISODateFormat` – Constant.

```
Dates.ISODateFormat
```

Describes the ISO8601 formatting for a date. This is the default value for `Dates.format` of a `Date`.

Example

```
julia> Dates.format(Date(2018, 8, 8), ISODateFormat)
"2018-08-08"
```

`Dates.ISOTimeFormat` – Constant.

```
Dates.ISOTimeFormat
```

Describes the ISO8601 formatting for a time. This is the default value for `Dates.format` of a `Time`.

Example

```
julia> Dates.format(Time(12, 0, 43, 1), ISOTimeFormat)
"12:00:43.001"
```

`Dates.RFC1123Format` – Constant.

```
Dates.RFC1123Format
```

Describes the RFC1123 formatting for a date and time.

Example

```
julia> Dates.format(DateTime(2018, 8, 8, 12, 0, 43, 1), RFC1123Format)
"Wed, 08 Aug 2018 12:00:43"
```

Chapter 68

分隔符文件

`DelimitedFiles.readdlm` – Method.

```
readdlm(source, delim::AbstractChar, T::Type, eol::AbstractChar; header=false, skipstart=0,
↪  skipblanks=true, use_mmap, quotes=true, dims, comments=false, comment_char='#')
```

Read a matrix from the source where each line (separated by `eol`) gives one row, with elements separated by the given delimiter. The source can be a text file, stream or byte array. Memory mapped files can be used by passing the byte array representation of the mapped segment as source.

If T is a numeric type, the result is an array of that type, with any non-numeric elements as `NaN` for floating-point types, or zero. Other useful values of T include `String`, `AbstractString`, and `Any`.

If `header` is `true`, the first row of data will be read as header and the tuple `(data_cells, header_cells)` is returned instead of only `data_cells`.

Specifying `skipstart` will ignore the corresponding number of initial lines from the input.

If `skipblanks` is `true`, blank lines in the input will be ignored.

If `use_mmap` is `true`, the file specified by `source` is memory mapped for potential speedups if the file is large. Default is `false`. On a Windows filesystem, `use_mmap` should not be set to `true` unless the file is only read once and is also not written to. Some edge cases exist where an OS is Unix-like but the filesystem is Windows-like.

If `quotes` is `true`, columns enclosed within double-quote (") characters are allowed to contain new lines and column delimiters. Double-quote characters within a quoted field must be escaped with another double-quote. Specifying `dims` as a tuple of the expected rows and columns (including header, if any) may speed up reading of large files. If `comments` is `true`, lines beginning with `comment_char` and text following `comment_char` in any line are ignored.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [5; 6; 7; 8];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
```

```
       end

julia> readdlm("delim_file.txt", '\t', Int, '\n')
4×2 Matrix{Int64}:
 1  5
 2  6
 3  7
 4  8

julia> rm("delim_file.txt")
```

source

DelimitedFiles.readdlm – Method.

```
readdlm(source, delim::AbstractChar, eol::AbstractChar; options...)
```

If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

source

DelimitedFiles.readdlm – Method.

```
readdlm(source, delim::AbstractChar, T::Type; options...)
```

The end of line delimiter is taken as `\n`.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [1.1; 2.2; 3.3; 4.4];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y], ',')
       end;

julia> readdlm("delim_file.txt", ',', Float64)
4×2 Matrix{Float64}:
 1.0  1.1
 2.0  2.2
 3.0  3.3
 4.0  4.4

julia> rm("delim_file.txt")
```

source

`DelimitedFiles.readdlm` – Method.

```
readdlm(source, delim::AbstractChar; options...)
```

The end of line delimiter is taken as `\n`. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [1.1; 2.2; 3.3; 4.4];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y], ',')
       end;

julia> readdlm("delim_file.txt", ',')
4×2 Matrix{Float64}:
 1.0  1.1
 2.0  2.2
 3.0  3.3
 4.0  4.4

julia> z = ["a"; "b"; "c"; "d"];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x z], ',')
       end;

julia> readdlm("delim_file.txt", ',')
4×2 Matrix{Any}:
 1  "a"
 2  "b"
 3  "c"
 4  "d"

julia> rm("delim_file.txt")
```

source

`DelimitedFiles.readdlm` – Method.

```
readdlm(source, T::Type; options...)
```

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as `\n`.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [5; 6; 7; 8];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
       end;

julia> readdlm("delim_file.txt", Int64)
4×2 Matrix{Int64}:
 1  5
 2  6
 3  7
 4  8

julia> readdlm("delim_file.txt", Float64)
4×2 Matrix{Float64}:
 1.0  5.0
 2.0  6.0
 3.0  7.0
 4.0  8.0

julia> rm("delim_file.txt")
```

source

DelimitedFiles.readdlm – Method.

```
readdlm(source; options...)
```

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as \n. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = ["a"; "b"; "c"; "d"];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
       end;

julia> readdlm("delim_file.txt")
4×2 Matrix{Any}:
 1  "a"
 2  "b"
 3  "c"
```

```
 4  "d"

julia> rm("delim_file.txt")
```

source

DelimitedFiles.writedlm – Function.

```
writedlm(f, A, delim='\t'; opts)
```

Write A (a vector, matrix, or an iterable collection of iterable rows) as text to f (either a filename string or an IO stream) using the given delimiter delim (which defaults to tab, but can be any printable Julia object, typically a Char or AbstractString).

For example, two vectors x and y of the same length can be written as two columns of tab-delimited text to f by either writedlm(f, [x y]) or by writedlm(f, zip(x, y)).

Examples

```
julia> using DelimitedFiles

julia> x = [1; 2; 3; 4];

julia> y = [5; 6; 7; 8];

julia> open("delim_file.txt", "w") do io
           writedlm(io, [x y])
       end

julia> readdlm("delim_file.txt", '\t', Int, '\n')
4×2 Matrix{Int64}:
 1  5
 2  6
 3  7
 4  8

julia> rm("delim_file.txt")
```

source

Chapter 69

Distributed Computing

Tools for distributed parallel processing.

`Distributed.addprocs` – Function.

```
addprocs(manager::ClusterManager; kwargs...) -> List of process identifiers
```

Launches worker processes via the specified cluster manager.

For example, Beowulf clusters are supported via a custom cluster manager implemented in the package `ClusterManagers.jl`.

The number of seconds a newly launched worker waits for connection establishment from the master can be specified via variable `JULIA_WORKER_TIMEOUT` in the worker process's environment. Relevant only when using TCP/IP as transport.

To launch workers without blocking the REPL, or the containing function if launching workers programmatically, execute `addprocs` in its own task.

Examples

```
# On busy clusters, call `addprocs` asynchronously
t = @async addprocs(...)
```

```
# Utilize workers as and when they come online
if nprocs() > 1   # Ensure at least one new worker is available
   ....   # perform distributed execution
end
```

```
# Retrieve newly launched worker IDs, or any error messages
if istaskdone(t)   # Check if `addprocs` has completed to ensure `fetch` doesn't block
    if nworkers() == N
        new_pids = fetch(t)
    else
        fetch(t)
    end
end
```

source

```
addprocs(machines; tunnel=false, sshflags=``, max_parallel=10, kwargs...) -> List of process
↪  identifiers
```

Add worker processes on remote machines via SSH. Configuration is done with keyword arguments (see below). In particular, the `exename` keyword can be used to specify the path to the `julia` binary on the remote machine(s).

`machines` is a vector of "machine specifications" which are given as strings of the form `[user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user and `port` to the standard SSH port. If `[bind_addr[:port]]` is specified, other workers will connect to this worker at the specified `bind_addr` and `port`.

It is possible to launch multiple processes on a remote host by using a tuple in the `machines` vector or the form `(machine_spec, count)`, where `count` is the number of workers to be launched on the specified host. Passing `:auto` as the worker count will launch as many workers as the number of CPU threads on the remote host.

Examples:

```
addprocs([
    "remote1",               # one worker on 'remote1' logging in with the current username
    "user@remote2",          # one worker on 'remote2' logging in with the 'user' username
    "user@remote3:2222",     # specifying SSH port to '2222' for 'remote3'
    ("user@remote4", 4),     # launch 4 workers on 'remote4'
    ("user@remote5", :auto), # launch as many workers as CPU threads on 'remote5'
])
```

Keyword arguments:

- `tunnel`: if `true` then SSH tunneling will be used to connect to the worker from the master process. Default is `false`.
- `multiplex`: if `true` then SSH multiplexing is used for SSH tunneling. Default is `false`.
- `ssh`: the name or path of the SSH client executable used to start the workers. Default is `"ssh"`.
- `sshflags`: specifies additional ssh options, e.g. ``sshflags=`-i /home/foo/bar.pem` ``
- `max_parallel`: specifies the maximum number of workers connected to in parallel at a host. Defaults to 10.
- `shell`: specifies the type of shell to which ssh connects on the workers.
  - `shell=:posix`: a POSIX-compatible Unix/Linux shell (sh, ksh, bash, dash, zsh, etc.). The default.
  - `shell=:csh`: a Unix C shell (csh, tcsh).
  - `shell=:wincmd`: Microsoft Windows `cmd.exe`.
- `dir`: specifies the working directory on the workers. Defaults to the host's current directory (as found by `pwd()`)
- `enable_threaded_blas`: if `true` then BLAS will run on multiple threads in added processes. Default is `false`.
- `exename`: name of the `julia` executable. Defaults to `"$(Sys.BINDIR)/julia"` or `"$(Sys.BINDIR)/julia-debug"` as the case may be. It is recommended that a common Julia version is used on all remote machines because serialization and code distribution might fail otherwise.
- `exeflags`: additional flags passed to the worker processes.

- `topology`: Specifies how the workers connect to each other. Sending a message between unconnected workers results in an error.
  - `topology=:all_to_all`: All processes are connected to each other. The default.
  - `topology=:master_worker`: Only the driver process, i.e. `pid` 1 connects to the workers. The workers do not connect to each other.
  - `topology=:custom`: The `launch` method of the cluster manager specifies the connection topology via fields `ident` and `connect_idents` in `WorkerConfig`. A worker with a cluster manager identity `ident` will connect to all workers specified in `connect_idents`.
- `lazy`: Applicable only with `topology=:all_to_all`. If `true`, worker-worker connections are setup lazily, i.e. they are setup at the first instance of a remote call between workers. Default is true.
- `env`: provide an array of string pairs such as `env=["JULIA_DEPOT_PATH"=>"/depot"]` to request that environment variables are set on the remote machine. By default only the environment variable `JULIA_WORKER_TIMEOUT` is passed automatically from the local to the remote environment.
- `cmdline_cookie`: pass the authentication cookie via the `--worker` commandline option. The (more secure) default behaviour of passing the cookie via ssh stdio may hang with Windows workers that use older (pre-ConPTY) Julia or Windows versions, in which case `cmdline_cookie=true` offers a work-around.

Julia 1.6

The keyword arguments `ssh`, `shell`, `env` and `cmdline_cookie` were added in Julia 1.6.

Environment variables:

If the master process fails to establish a connection with a newly launched worker within 60.0 seconds, the worker treats it as a fatal situation and terminates. This timeout can be controlled via environment variable `JULIA_WORKER_TIMEOUT`. The value of `JULIA_WORKER_TIMEOUT` on the master process specifies the number of seconds a newly launched worker waits for connection establishment.

source

```
addprocs(np::Integer=Sys.CPU_THREADS; restrict=true, kwargs...) -> List of process identifiers
```

Launch np workers on the local host using the in-built `LocalManager`.

Local workers inherit the current package environment (i.e., active project, `LOAD_PATH`, and `DEPOT_PATH`) from the main process.

Keyword arguments:

- `restrict::Bool`: if `true` (default) binding is restricted to `127.0.0.1`.
- `dir`, `exename`, `exeflags`, `env`, `topology`, `lazy`, `enable_threaded_blas`: same effect as for `SSHManager`, see documentation for `addprocs(machines::AbstractVector)`.

Julia 1.9

The inheriting of the package environment and the env keyword argument were added in Julia 1.9.

source

`Distributed.nprocs` – Function.

```
nprocs()
```

Get the number of available processes.

Examples

```
julia> nprocs()
3

julia> workers()
2-element Array{Int64,1}:
 2
 3
```

source

`Distributed.nworkers` – Function.

```
nworkers()
```

Get the number of available worker processes. This is one less than `nprocs()`. Equal to `nprocs()` if `nprocs() == 1`.

Examples

```
$ julia -p 2

julia> nprocs()
3

julia> nworkers()
2
```

source

`Distributed.procs` – Method.

```
procs()
```

Return a list of all process identifiers, including pid 1 (which is not included by `workers()`).

Examples

```
$ julia -p 2

julia> procs()
3-element Array{Int64,1}:
```

```
 1
 2
 3
```

source

`Distributed.procs` – Method.

```
procs(pid::Integer)
```

Return a list of all process identifiers on the same physical node. Specifically all workers bound to the same ip-address as `pid` are returned.

source

`Distributed.workers` – Function.

```
workers()
```

Return a list of all worker process identifiers.

Examples

```
$ julia -p 2

julia> workers()
2-element Array{Int64,1}:
 2
 3
```

source

`Distributed.rmprocs` – Function.

```
rmprocs(pids...; waitfor=typemax(Int))
```

Remove the specified workers. Note that only process 1 can add or remove workers.

Argument `waitfor` specifies how long to wait for the workers to shut down:

- If unspecified, `rmprocs` will wait until all requested `pids` are removed.
- An `ErrorException` is raised if all workers cannot be terminated before the requested `waitfor` seconds.
- With a `waitfor` value of 0, the call returns immediately with the workers scheduled for removal in a different task. The scheduled `Task` object is returned. The user should call `wait` on the task before invoking any other parallel calls.

Examples

```
$ julia -p 5

julia> t = rmprocs(2, 3, waitfor=0)
Task (runnable) @0x0000000107c718d0

julia> wait(t)

julia> workers()
3-element Array{Int64,1}:
 4
 5
 6
```

source

`Distributed.interrupt` – Function.

```
interrupt(pids::Integer...)
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

source

```
interrupt(pids::AbstractVector=workers())
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

source

`Distributed.myid` – Function.

```
myid()
```

Get the id of the current process.

Examples

```
julia> myid()
1

julia> remotecall_fetch(() -> myid(), 4)
4
```

source

`Distributed.pmap` – Function.

```
pmap(f, [::AbstractWorkerPool], c...; distributed=true, batch_size=1, on_error=nothing,
↪  retry_delays=[], retry_check=nothing) -> collection
```

Transform collection c by applying f to each element using available workers and tasks.

For multiple collection arguments, apply f elementwise.

Note that f must be made available to all worker processes; see Code Availability and Loading Packages for details.

If a worker pool is not specified, all available workers, i.e., the default worker pool is used.

By default, pmap distributes the computation over all specified workers. To use only the local process and distribute over tasks, specify `distributed=false`. This is equivalent to using `asyncmap`. For example, `pmap(f, c; distributed=false)` is equivalent to `asyncmap(f,c; ntasks=()->nworkers())`

pmap can also use a mix of processes and tasks via the `batch_size` argument. For batch sizes greater than 1, the collection is processed in multiple batches, each of length `batch_size` or less. A batch is sent as a single request to a free worker, where a local `asyncmap` processes elements from the batch using multiple concurrent tasks.

Any error stops pmap from processing the remainder of the collection. To override this behavior you can specify an error handling function via argument `on_error` which takes in a single argument, i.e., the exception. The function can stop the processing by rethrowing the error, or, to continue, return any value which is then returned inline with the results to the caller.

Consider the following two examples. The first one returns the exception object inline, the second a 0 in place of any exception:

```
julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=identity)
4-element Array{Any,1}:
 1
  ErrorException("foo")
 3
  ErrorException("foo")

julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=ex->0)
4-element Array{Int64,1}:
 1
 0
 3
 0
```

Errors can also be handled by retrying failed computations. Keyword arguments `retry_delays` and `retry_check` are passed through to `retry` as keyword arguments `delays` and `check` respectively. If batching is specified, and an entire batch fails, all items in the batch are retried.

Note that if both `on_error` and `retry_delays` are specified, the `on_error` hook is called before retrying. If `on_error` does not throw (or rethrow) an exception, the element will not be retried.

Example: On errors, retry f on an element a maximum of 3 times without any delay between retries.

```
pmap(f, c; retry_delays = zeros(3))
```

Example: Retry f only if the exception is not of type InexactError, with exponentially increasing delays up to 3 times. Return a NaN in place for all InexactError occurrences.

```
pmap(f, c; on_error = e->(isa(e, InexactError) ? NaN : rethrow()), retry_delays =
↪  ExponentialBackOff(n = 3))
```

source

Distributed.RemoteException – Type.

```
RemoteException(captured)
```

Exceptions on remote computations are captured and rethrown locally. A RemoteException wraps the pid of the worker and a captured exception. A CapturedException captures the remote exception and a serializable form of the call stack when the exception was raised.

source

Distributed.ProcessExitedException – Type.

```
ProcessExitedException(worker_id::Int)
```

After a client Julia process has exited, further attempts to reference the dead child will throw this exception.

source

Distributed.Future – Type.

```
Future(w::Int, rrid::RRID, v::Union{Some, Nothing}=nothing)
```

A Future is a placeholder for a single computation of unknown termination status and time. For multiple potential computations, see RemoteChannel. See remoteref_id for identifying an AbstractRemoteRef.

source

Distributed.RemoteChannel – Type.

```
RemoteChannel(pid::Integer=myid())
```

Make a reference to a Channel{Any}(1) on process pid. The default pid is the current process.

```
RemoteChannel(f::Function, pid::Integer=myid())
```

Create references to remote channels of a specific size and type. f is a function that when executed on pid must return an implementation of an AbstractChannel.

For example, RemoteChannel(()->Channel{Int}(10), pid), will return a reference to a channel of type Int and size 10 on pid.

The default pid is the current process.

source

`Base.fetch` – Method.

```
fetch(x::Future)
```

Wait for and get the value of a `Future`. The fetched value is cached locally. Further calls to `fetch` on the same reference return the cached value. If the remote value is an exception, throws a `RemoteException` which captures the remote exception and backtrace.

source

`Base.fetch` – Method.

```
fetch(c::RemoteChannel)
```

Wait for and get a value from a `RemoteChannel`. Exceptions raised are the same as for a `Future`. Does not remove the item fetched.

source

```
fetch(x::Any)
```

Return x.

source

`Distributed.remotecall` – Method.

```
remotecall(f, id::Integer, args...; kwargs...) -> Future
```

Call a function `f` asynchronously on the given arguments on the specified process. Return a `Future`. Keyword arguments, if any, are passed through to `f`.

source

`Distributed.remotecall_wait` – Method.

```
remotecall_wait(f, id::Integer, args...; kwargs...)
```

Perform a faster `wait(remotecall(...))` in one message on the `Worker` specified by worker id `id`. Keyword arguments, if any, are passed through to `f`.

See also `wait` and `remotecall`.

source

`Distributed.remotecall_fetch` – Method.

```
remotecall_fetch(f, id::Integer, args...; kwargs...)
```

Perform `fetch(remotecall(...))` in one message. Keyword arguments, if any, are passed through to `f`. Any remote exceptions are captured in a `RemoteException` and thrown.

See also `fetch` and `remotecall`.

Examples

```
$ julia -p 2

julia> remotecall_fetch(sqrt, 2, 4)
2.0

julia> remotecall_fetch(sqrt, 2, -4)
ERROR: On worker 2:
DomainError with -4.0:
sqrt was called with a negative real argument but will only return a complex result if called
↪  with a complex argument. Try sqrt(Complex(x)).
...
```

source

`Distributed.remote_do` – Method.

```
remote_do(f, id::Integer, args...; kwargs...) -> nothing
```

Executes `f` on worker `id` asynchronously. Unlike `remotecall`, it does not store the result of computation, nor is there a way to wait for its completion.

A successful invocation indicates that the request has been accepted for execution on the remote node.

While consecutive `remotecalls` to the same worker are serialized in the order they are invoked, the order of executions on the remote worker is undetermined. For example, `remote_do(f1, 2); remotecall(f2, 2); remote_do(f3, 2)` will serialize the call to `f1`, followed by `f2` and `f3` in that order. However, it is not guaranteed that `f1` is executed before `f3` on worker 2.

Any exceptions thrown by `f` are printed to `stderr` on the remote worker.

Keyword arguments, if any, are passed through to `f`.

source

`Base.put!` – Method.

```
put!(rr::RemoteChannel, args...)
```

Store a set of values to the `RemoteChannel`. If the channel is full, blocks until space is available. Return the first argument.

source

`Base.put!` – Method.

```
put!(rr::Future, v)
```

Store a value to a `Future` `rr`. `Future`s are write-once remote references. A `put!` on an already set `Future` throws an `Exception`. All asynchronous remote calls return `Future`s and set the value to the return value of the call upon completion.

source

`Base.take!` – Method.

```
take!(rr::RemoteChannel, args...)
```

Fetch value(s) from a `RemoteChannel` `rr`, removing the value(s) in the process.

source

`Base.isready` – Method.

```
isready(rr::RemoteChannel, args...)
```

Determine whether a `RemoteChannel` has a value stored to it. Note that this function can cause race conditions, since by the time you receive its result it may no longer be true. However, it can be safely used on a `Future` since they are assigned only once.

source

`Base.isready` – Method.

```
isready(rr::Future)
```

Determine whether a `Future` has a value stored to it.

If the argument `Future` is owned by a different node, this call will block to wait for the answer. It is recommended to wait for `rr` in a separate task instead or to use a local `Channel` as a proxy:

```
p = 1
f = Future(p)
errormonitor(@async put!(f, remotecall_fetch(long_computation, p)))
isready(f)  # will not block
```

source

`Distributed.AbstractWorkerPool` – Type.

```
AbstractWorkerPool
```

Supertype for worker pools such as `WorkerPool` and `CachingPool`. An `AbstractWorkerPool` should implement:

- `push!` - add a new worker to the overall pool (available + busy)
- `put!` - put back a worker to the available pool
- `take!` - take a worker from the available pool (to be used for remote function execution)
- `length` - number of workers available in the overall pool
- `isready` - return false if a `take!` on the pool would block, else true

The default implementations of the above (on a `AbstractWorkerPool`) require fields

- `channel::Channel{Int}`
- `workers::Set{Int}`

where `channel` contains free worker pids and `workers` is the set of all workers associated with this pool.

source

`Distributed.WorkerPool` – Type.

```
WorkerPool(workers::Union{Vector{Int},AbstractRange{Int}})
```

Create a `WorkerPool` from a vector or range of worker ids.

Examples

```
$ julia -p 3

julia> WorkerPool([2, 3])
WorkerPool(Channel{Int64}(sz_max:9223372036854775807,sz_curr:2), Set([2, 3]),
↪  RemoteChannel{Channel{Any}}(1, 1, 6))

julia> WorkerPool(2:4)
WorkerPool(Channel{Int64}(sz_max:9223372036854775807,sz_curr:2), Set([4, 2, 3]),
↪  RemoteChannel{Channel{Any}}(1, 1, 7))
```

source

`Distributed.CachingPool` – Type.

```
CachingPool(workers::Vector{Int})
```

An implementation of an `AbstractWorkerPool`. `remote`, `remotecall_fetch`, `pmap` (and other remote calls which execute functions remotely) benefit from caching the serialized/deserialized functions on the worker nodes, especially closures (which may capture large amounts of data).

The remote cache is maintained for the lifetime of the returned CachingPool object. To clear the cache earlier, use clear!(pool).

For global variables, only the bindings are captured in a closure, not the data. let blocks can be used to capture global data.

Examples

```
const foo = rand(10^8);
wp = CachingPool(workers())
let foo = foo
    pmap(i -> sum(foo) + i, wp, 1:100);
end
```

The above would transfer foo only once to each worker.

source

Distributed.default_worker_pool – Function.

```
default_worker_pool()
```

AbstractWorkerPool containing idle workers - used by remote(f) and pmap (by default). Unless one is explicitly set via default_worker_pool!(pool), the default worker pool is initialized to a WorkerPool.

Examples

```
$ julia -p 3

julia> default_worker_pool()
WorkerPool(Channel{Int64}(sz_max:9223372036854775807,sz_curr:3), Set([4, 2, 3]),
↪  RemoteChannel{Channel{Any}}(1, 1, 4))
```

source

Distributed.clear! – Method.

```
clear!(pool::CachingPool) -> pool
```

Removes all cached functions from all participating workers.

source

Distributed.remote – Function.

```
remote([p::AbstractWorkerPool], f) -> Function
```

Return an anonymous function that executes function f on an available worker (drawn from WorkerPool p if provided) using remotecall_fetch.

source

`Distributed.remotecall` – Method.

```
remotecall(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remotecall` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> f = remotecall(maximum, wp, A)
Future(2, 1, 6, nothing)
```

In this example, the task ran on pid 2, called from pid 1.

source

`Distributed.remotecall_wait` – Method.

```
remotecall_wait(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall_wait(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remotecall_wait` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> f = remotecall_wait(maximum, wp, A)
Future(3, 1, 9, nothing)

julia> fetch(f)
0.9995177101692958
```

source

`Distributed.remotecall_fetch` – Method.

```
remotecall_fetch(f, pool::AbstractWorkerPool, args...; kwargs...) -> result
```

WorkerPool variant of `remotecall_fetch(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remotecall_fetch` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> remotecall_fetch(maximum, wp, A)
0.9995177101692958
```

source

`Distributed.remote_do` – Method.

```
remote_do(f, pool::AbstractWorkerPool, args...; kwargs...) -> nothing
```

WorkerPool variant of `remote_do(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remote_do` on it.

source

`Distributed.@spawnat` – Macro.

```
@spawnat p expr
```

Create a closure around an expression and run the closure asynchronously on process `p`. Return a Future to the result. If `p` is the quoted literal symbol `:any`, then the system will pick a processor to use automatically.

Examples

```
julia> addprocs(3);

julia> f = @spawnat 2 myid()
Future(2, 1, 3, nothing)

julia> fetch(f)
2

julia> f = @spawnat :any myid()
Future(3, 1, 7, nothing)

julia> fetch(f)
3
```

Julia 1.3

The `:any` argument is available as of Julia 1.3.

source

`Distributed.@fetch` – Macro.

```
@fetch expr
```

Equivalent to `fetch(@spawnat :any expr)`. See `fetch` and `@spawnat`.

Examples

```
julia> addprocs(3);

julia> @fetch myid()
2

julia> @fetch myid()
3

julia> @fetch myid()
4

julia> @fetch myid()
2
```

source

`Distributed.@fetchfrom` – Macro.

```
@fetchfrom
```

Equivalent to `fetch(@spawnat p expr)`. See `fetch` and `@spawnat`.

Examples

```
julia> addprocs(3);

julia> @fetchfrom 2 myid()
2

julia> @fetchfrom 4 myid()
4
```

source

`Distributed.@distributed` – Macro.

```
@distributed
```

A distributed memory, parallel for loop of the form :

```
@distributed [reducer] for var = range
    body
end
```

The specified range is partitioned and locally executed across all workers. In case an optional reducer function is specified, `@distributed` performs local reductions on each worker with a final reduction on the calling process.

Note that without a reducer function, `@distributed` executes asynchronously, i.e. it spawns independent tasks on all available workers and returns immediately without waiting for completion. To wait for completion, prefix the call with `@sync`, like :

```
@sync @distributed for var = range
    body
end
```

source

`Distributed.@everywhere` – Macro.

```
@everywhere [procs()] expr
```

Execute an expression under `Main` on all `procs`. Errors on any of the processes are collected into a `CompositeException` and thrown. For example:

```
@everywhere bar = 1
```

will define `Main.bar` on all current processes. Any processes added later (say with `addprocs()`) will not have the expression defined.

Unlike `@spawnat`, `@everywhere` does not capture any local variables. Instead, local variables can be broadcast using interpolation:

```
foo = 1
@everywhere bar = $foo
```

The optional argument `procs` allows specifying a subset of all processes to have execute the expression.

Similar to calling `remotecall_eval(Main, procs, expr)`, but with two extra features:

```
- `using` and `import` statements run on the calling process first, to ensure
  packages are precompiled.
- The current source file path used by `include` is propagated to other processes.
```

source

`Distributed.clear!` – Method.

```
clear!(syms, pids=workers(); mod=Main)
```

Clears global bindings in modules by initializing them to `nothing`. `syms` should be of type `Symbol` or a collection of `Symbols` . `pids` and `mod` identify the processes and the module in which global variables are to be reinitialized. Only those names found to be defined under `mod` are cleared.

An exception is raised if a global constant is requested to be cleared.

source

`Distributed.remoteref_id` – Function.

```
remoteref_id(r::AbstractRemoteRef) -> RRID
```

`Futures` and `RemoteChannels` are identified by fields:

- `where` - refers to the node where the underlying object/storage referred to by the reference actually exists.
- `whence` - refers to the node the remote reference was created from. Note that this is different from the node where the underlying object referred to actually exists. For example calling `RemoteChannel(2)` from the master process would result in a `where` value of 2 and a `whence` value of 1.
- `id` is unique across all references created from the worker specified by `whence`.

Taken together, `whence` and `id` uniquely identify a reference across all workers.

`remoteref_id` is a low-level API which returns a `RRID` object that wraps `whence` and `id` values of a remote reference.

source

`Distributed.channel_from_id` – Function.

```
channel_from_id(id) -> c
```

A low-level API which returns the backing `AbstractChannel` for an `id` returned by `remoteref_id`. The call is valid only on the node where the backing channel exists.

source

`Distributed.worker_id_from_socket` – Function.

```
worker_id_from_socket(s) -> pid
```

A low-level API which, given a `IO` connection or a `Worker`, returns the `pid` of the worker it is connected to. This is useful when writing custom `serialize` methods for a type, which optimizes the data written out depending on the receiving process id.

source

`Distributed.cluster_cookie` – Method.

```
cluster_cookie() -> cookie
```

Return the cluster cookie.

source

`Distributed.cluster_cookie` – Method.

```
cluster_cookie(cookie) -> cookie
```

Set the passed cookie as the cluster cookie, then returns it.

source

69.1 Cluster Manager Interface

This interface provides a mechanism to launch and manage Julia workers on different cluster environments. There are two types of managers present in Base: `LocalManager`, for launching additional workers on the same host, and `SSHManager`, for launching on remote hosts via `ssh`. TCP/IP sockets are used to connect and transport messages between processes. It is possible for Cluster Managers to provide a different transport.

`Distributed.ClusterManager` – Type.

```
ClusterManager
```

Supertype for cluster managers, which control workers processes as a cluster. Cluster managers implement how workers can be added, removed and communicated with. `SSHManager` and `LocalManager` are subtypes of this.

source

`Distributed.WorkerConfig` – Type.

```
WorkerConfig
```

Type used by `ClusterManager`s to control workers added to their clusters. Some fields are used by all cluster managers to access a host:

- `io` – the connection used to access the worker (a subtype of `IO` or `Nothing`)
- `host` – the host address (either a `String` or `Nothing`)
- `port` – the port on the host used to connect to the worker (either an `Int` or `Nothing`)

Some are used by the cluster manager to add workers to an already-initialized host:

- `count` – the number of workers to be launched on the host
- `exename` – the path to the Julia executable on the host, defaults to `"$(Sys.BINDIR)/julia"` or `"$(Sys.BINDIR)/julia-d`

- `exeflags` –flags to use when launching Julia remotely

The `userdata` field is used to store information for each worker by external managers.

Some fields are used by `SSHManager` and similar managers:

- `tunnel` –`true` (use tunneling), `false` (do not use tunneling), or `nothing` (use default for the manager)
- `multiplex` –`true` (use SSH multiplexing for tunneling) or `false`
- `forward` –the forwarding option used for `-L` option of ssh
- `bind_addr` –the address on the remote host to bind to
- `sshflags` –flags to use in establishing the SSH connection
- `max_parallel` –the maximum number of workers to connect to in parallel on the host

Some fields are used by both `LocalManagers` and `SSHManagers`:

- `connect_at` –determines whether this is a worker-to-worker or driver-to-worker setup call
- `process` –the process which will be connected (usually the manager will assign this during `addprocs`)
- `ospid` –the process ID according to the host OS, used to interrupt worker processes
- `environ` –private dictionary used to store temporary information by Local/SSH managers
- `ident` –worker as identified by the `ClusterManager`
- `connect_idents` –list of worker ids the worker must connect to if using a custom topology
- `enable_threaded_blas` –`true`, `false`, or `nothing`, whether to use threaded BLAS or not on the workers

source

`Distributed.launch` – Function.

```
launch(manager::ClusterManager, params::Dict, launched::Array, launch_ntfy::Condition)
```

Implemented by cluster managers. For every Julia worker launched by this function, it should append a `WorkerConfig` entry to `launched` and notify `launch_ntfy`. The function MUST exit once all workers, requested by `manager` have been launched. `params` is a dictionary of all keyword arguments `addprocs` was called with.

source

`Distributed.manage` – Function.

```
manage(manager::ClusterManager, id::Integer, config::WorkerConfig. op::Symbol)
```

Implemented by cluster managers. It is called on the master process, during a worker's lifetime, with appropriate `op` values:

- with `:register`/`:deregister` when a worker is added / removed from the Julia worker pool.
- with `:interrupt` when `interrupt(workers)` is called. The `ClusterManager` should signal the appropriate worker with an interrupt signal.

- with `:finalize` for cleanup purposes.

source

`Base.kill` – Method.

```
kill(manager::ClusterManager, pid::Int, config::WorkerConfig)
```

Implemented by cluster managers. It is called on the master process, by `rmprocs`. It should cause the remote worker specified by `pid` to exit. `kill(manager::ClusterManager.....)` executes a remote `exit()` on `pid`.

source

`Sockets.connect` – Method.

```
connect(manager::ClusterManager, pid::Int, config::WorkerConfig) -> (instrm::IO, outstrm::IO)
```

Implemented by cluster managers using custom transports. It should establish a logical connection to worker with id `pid`, specified by `config` and return a pair of `IO` objects. Messages from `pid` to current process will be read off `instrm`, while messages to be sent to `pid` will be written to `outstrm`. The custom transport implementation must ensure that messages are delivered and received completely and in order. `connect(manager::ClusterManager.....)` sets up TCP/IP socket connections in-between workers.

source

`Distributed.init_worker` – Function.

```
init_worker(cookie::AbstractString, manager::ClusterManager=DefaultClusterManager())
```

Called by cluster managers implementing custom transports. It initializes a newly launched process as a worker. Command line argument `--worker[=<cookie>]` has the effect of initializing a process as a worker using TCP/IP sockets for transport. `cookie` is a `cluster_cookie`.

source

`Distributed.start_worker` – Function.

```
start_worker([out::IO=stdout], cookie::AbstractString=readline(stdin); close_stdin::Bool=true,
↪  stderr_to_stdout::Bool=true)
```

`start_worker` is an internal function which is the default entry point for worker processes connecting via TCP/IP. It sets up the process as a Julia cluster worker.

host:port information is written to stream `out` (defaults to stdout).

The function reads the cookie from stdin if required, and listens on a free port (or if specified, the port in the `--bind-to` command line option) and schedules tasks to process incoming TCP connections and requests. It also (optionally) closes stdin and redirects stderr to stdout.

It does not return.

source

`Distributed.process_messages` – Function.

```
process_messages(r_stream::IO, w_stream::IO, incoming::Bool=true)
```

Called by cluster managers using custom transports. It should be called when the custom transport implementation receives the first message from a remote worker. The custom transport must manage a logical connection to the remote worker and provide two `IO` objects, one for incoming messages and the other for messages addressed to the remote worker. If `incoming` is `true`, the remote peer initiated the connection. Whichever of the pair initiates the connection sends the cluster cookie and its Julia version number to perform the authentication handshake.

See also `cluster_cookie`.

source

`Distributed.default_addprocs_params` – Function.

```
default_addprocs_params(mgr::ClusterManager) -> Dict{Symbol, Any}
```

Implemented by cluster managers. The default keyword parameters passed when calling `addprocs(mgr)`. The minimal set of options is available by calling `default_addprocs_params()`

source

Chapter 70

Downloads

Downloads.download – Function.

```
download(url, [ output = tempname() ];
    [ method = "GET", ]
    [ headers = <none>, ]
    [ timeout = <none>, ]
    [ progress = <none>, ]
    [ verbose = false, ]
    [ debug = <none>, ]
    [ downloader = <default>, ]
) -> output

    url        :: AbstractString
    output     :: Union{AbstractString, AbstractCmd, IO}
    method     :: AbstractString
    headers    :: Union{AbstractVector, AbstractDict}
    timeout    :: Real
    progress   :: (total::Integer, now::Integer) --> Any
    verbose    :: Bool
    debug      :: (type, message) --> Any
    downloader :: Downloader
```

Download a file from the given url, saving it to `output` or if not specified, a temporary path. The `output` can also be an `IO` handle, in which case the body of the response is streamed to that handle and the handle is returned. If `output` is a command, the command is run and output is sent to it on stdin.

If the `downloader` keyword argument is provided, it must be a `Downloader` object. Resources and connections will be shared between downloads performed by the same `Downloader` and cleaned up automatically when the object is garbage collected or there have been no downloads performed with it for a grace period. See `Downloader` for more info about configuration and usage.

If the `headers` keyword argument is provided, it must be a vector or dictionary whose elements are all pairs of strings. These pairs are passed as headers when downloading URLs with protocols that supports them, such as HTTP/S.

The `timeout` keyword argument specifies a timeout for the download to complete in seconds, with a resolution of milliseconds. By default no timeout is set, but this can also be explicitly requested by passing a timeout value of `Inf`. Separately, if 20 seconds elapse without receiving any data, the download will timeout. See extended help for how to disable this timeout.

If the `progress` keyword argument is provided, it must be a callback function which will be called whenever there are updates about the size and status of the ongoing download. The callback must take two integer arguments: `total` and `now` which are the total size of the download in bytes, and the number of bytes which have been downloaded so far. Note that `total` starts out as zero and remains zero until the server gives an indication of the total size of the download (e.g. with a `Content-Length` header), which may never happen. So a well-behaved progress callback should handle a total size of zero gracefully.

If the `verbose` option is set to true, `libcurl`, which is used to implement the download functionality will print debugging information to `stderr`. If the `debug` option is set to a function accepting two `String` arguments, then the verbose option is ignored and instead the data that would have been printed to `stderr` is passed to the `debug` callback with `type` and `message` arguments. The `type` argument indicates what kind of event has occurred, and is one of: `TEXT`, `HEADER IN`, `HEADER OUT`, `DATA IN`, `DATA OUT`, `SSL DATA IN` or `SSL DATA OUT`. The `message` argument is the description of the debug event.

Extended Help

For further customization, use a `Downloader` and `easy_hooks`. For example, to disable the 20 second timeout when no data is received, you may use the following:

```
downloader = Downloads.Downloader()
downloader.easy_hook = (easy, info) -> Downloads.Curl.setopt(easy,
↪  Downloads.Curl.CURLOPT_LOW_SPEED_TIME, 0)

Downloads.download("https://httpbingo.julialang.org/delay/30"; downloader)
```

`Downloads.request` – Function.

```
request(url;
    [ input = <none>, ]
    [ output = <none>, ]
    [ method = input ? "PUT" : output ? "GET" : "HEAD", ]
    [ headers = <none>, ]
    [ timeout = <none>, ]
    [ progress = <none>, ]
    [ verbose = false, ]
    [ debug = <none>, ]
    [ throw = true, ]
    [ downloader = <default>, ]
    [ interrupt = <none>, ]
) -> Union{Response, RequestError}

    url        :: AbstractString
    input      :: Union{AbstractString, AbstractCmd, IO}
    output     :: Union{AbstractString, AbstractCmd, IO}
    method     :: AbstractString
    headers    :: Union{AbstractVector, AbstractDict}
    timeout    :: Real
    progress   :: (dl_total, dl_now, ul_total, ul_now) --> Any
    verbose    :: Bool
    debug      :: (type, message) --> Any
    throw      :: Bool
    downloader :: Downloader
    interrupt  :: Base.Event
```

Make a request to the given url, returning a Response object capturing the status, headers and other information about the response. The body of the response is written to output if specified and discarded otherwise. For HTTP/S requests, if an input stream is given, a PUT request is made; otherwise if an output stream is given, a GET request is made; if neither is given a HEAD request is made. For other protocols, appropriate default methods are used based on what combination of input and output are requested. The following options differ from the download function:

- input allows providing a request body; if provided default to PUT request
- progress is a callback taking four integers for upload and download progress
- throw controls whether to throw or return a RequestError on request error

Note that unlike download which throws an error if the requested URL could not be downloaded (indicated by non-2xx status code), request returns a Response object no matter what the status code of the response is. If there is an error with getting a response at all, then a RequestError is thrown or returned.

If the interrupt keyword argument is provided, it must be a Base.Event object. If the event is triggered while the request is in progress, the request will be cancelled and an error will be thrown. This can be used to interrupt a long running request, for example if the user wants to cancel a download.

Downloads.Response – Type.

```
struct Response
    proto   :: String
    url     :: String
    status  :: Int
    message :: String
    headers :: Vector{Pair{String,String}}
end
```

Response is a type capturing the properties of a successful response to a request as an object. It has the following fields:

- proto: the protocol that was used to get the response
- url: the URL that was ultimately requested after following redirects
- status: the status code of the response, indicating success, failure, etc.
- message: a textual message describing the nature of the response
- headers: any headers that were returned with the response

The meaning and availability of some of these responses depends on the protocol used for the request. For many protocols, including HTTP/S and S/FTP, a 2xx status code indicates a successful response. For responses in protocols that do not support headers, the headers vector will be empty. HTTP/2 does not include a status message, only a status code, so the message will be empty.

Downloads.RequestError – Type.

```
struct RequestError <: ErrorException
    url      :: String
    code     :: Int
    message  :: String
    response :: Response
end
```

RequestError is a type capturing the properties of a failed response to a request as an exception object:

- url: the original URL that was requested without any redirects
- code: the libcurl error code; 0 if a protocol-only error occurred
- message: the libcurl error message indicating what went wrong
- response: response object capturing what response info is available

The same RequestError type is thrown by download if the request was successful but there was a protocol-level error indicated by a status code that is not in the 2xx range, in which case code will be zero and the message field will be the empty string. The request API only throws a RequestError if the libcurl error code is non-zero, in which case the included response object is likely to have a status of zero and an empty message. There are, however, situations where a curl-level error is thrown due to a protocol error, in which case both the inner and outer code and message may be of interest.

Downloads.Downloader – Type.

```
Downloader(; [ grace::Real = 30 ])
```

Downloader objects are used to perform individual download operations. Connections, name lookups and other resources are shared within a Downloader. These connections and resources are cleaned up after a configurable grace period (default: 30 seconds) since anything was downloaded with it, or when it is garbage collected, whichever comes first. If the grace period is set to zero, all resources will be cleaned up immediately as soon as there are no more ongoing downloads in progress. If the grace period is set to Inf then resources are not cleaned up until Downloader is garbage collected.

Chapter 71

文件相关事件

FileWatching.poll_fd – Function.

```
poll_fd(fd, timeout_s::Real=-1; readable=false, writable=false)
```

Monitor a file descriptor `fd` for changes in the read or write availability, and with a timeout given by `timeout_s` seconds.

The keyword arguments determine which of read and/or write status should be monitored; at least one of them must be set to `true`.

The returned value is an object with boolean fields `readable`, `writable`, and `timedout`, giving the result of the polling.

FileWatching.poll_file – Function.

```
poll_file(path::AbstractString, interval_s::Real=5.007, timeout_s::Real=-1) ->
↪  (previous::StatStruct, current)
```

Monitor a file for changes by polling every `interval_s` seconds until a change occurs or `timeout_s` seconds have elapsed. The `interval_s` should be a long period; the default is 5.007 seconds.

Returns a pair of status objects `(previous, current)` when a change is detected. The `previous` status is always a `StatStruct`, but it may have all of the fields zeroed (indicating the file didn't previously exist, or wasn't previously accessible).

The `current` status object may be a `StatStruct`, an `EOFError` (indicating the timeout elapsed), or some other `Exception` subtype (if the `stat` operation failed - for example, if the path does not exist).

To determine when a file was modified, compare `current isa StatStruct && mtime(prev) != mtime(current)` to detect notification of changes. However, using `watch_file` for this operation is preferred, since it is more reliable and efficient, although in some situations it may not be available.

FileWatching.watch_file – Function.

```
watch_file(path::AbstractString, timeout_s::Real=-1)
```

Watch file or directory `path` for changes until a change occurs or `timeout_s` seconds have elapsed. This function does not poll the file system and instead uses platform-specific functionality to receive notifications from the operating system (e.g. via inotify on Linux). See the NodeJS documentation linked below for details.

The returned value is an object with boolean fields `renamed`, `changed`, and `timedout`, giving the result of watching the file.

This behavior of this function varies slightly across platforms. See https://nodejs.org/api/fs.html#fs_caveats for more detailed information.

`FileWatching.watch_folder` – Function.

```
watch_folder(path::AbstractString, timeout_s::Real=-1)
```

Watches a file or directory `path` for changes until a change has occurred or `timeout_s` seconds have elapsed. This function does not poll the file system and instead uses platform-specific functionality to receive notifications from the operating system (e.g. via inotify on Linux). See the NodeJS documentation linked below for details.

This will continuing tracking changes for `path` in the background until `unwatch_folder` is called on the same `path`.

The returned value is an pair where the first field is the name of the changed file (if available) and the second field is an object with boolean fields `renamed`, `changed`, and `timedout`, giving the event.

This behavior of this function varies slightly across platforms. See https://nodejs.org/api/fs.html#fs_caveats for more detailed information.

`FileWatching.unwatch_folder` – Function.

```
unwatch_folder(path::AbstractString)
```

Stop background tracking of changes for `path`. It is not recommended to do this while another task is waiting for `watch_folder` to return on the same path, as the result may be unpredictable.

Chapter 72

Pidfile

A simple utility tool for creating advisory pidfiles (lock files).

72.1 Primary Functions

`FileWatching.Pidfile.mkpidlock` – Function.

```
mkpidlock([f::Function], at::String, [pid::Cint, proc::Process]; kwopts...)
```

Create a pidfile lock for the path "at" for the current process or the process identified by pid or proc. Can take a function to execute once locked, for usage in do blocks, after which the lock will be automatically closed. If the lock fails and `wait` is false, then an error is thrown.

The lock will be released by either `close`, a `finalizer`, or shortly after `proc` exits. Make sure the return value is live through the end of the critical section of your program, so the `finalizer` does not reclaim it early.

Optional keyword arguments:

- `mode`: file access mode (modified by the process umask). Defaults to world-readable.
- `poll_interval`: Specify the maximum time to between attempts (if `watch_file` doesn't work)
- `stale_age`: Delete an existing pidfile (ignoring the lock) if it is older than this many seconds, based on its mtime. The file won't be deleted until 5x longer than this if the pid in the file appears that it may be valid. Or 25x longer if `refresh` is overridden to 0 to disable lock refreshing. By default this is disabled (`stale_age` = 0), but a typical recommended value would be about 3-5x an estimated normal completion time.
- `refresh`: Keeps a lock from becoming stale by updating the mtime every interval of time that passes. By default, this is set to `stale_age/2`, which is the recommended value.
- `wait`: If true, block until we get the lock, if false, raise error if lock fails.

`FileWatching.Pidfile.trymkpidlock` – Function.

```
trymkpidlock([f::Function], at::String, [pid::Cint, proc::Process]; kwopts...)
```

Like `mkpidlock` except returns `false` instead of waiting if the file is already locked.

> **Julia 1.10**
>
> This function requires at least Julia 1.10.

`Base.close` – Method.

```
close(lock::LockMonitor)
```

Release a pidfile lock.

72.2 Helper Functions

`FileWatching.Pidfile.open_exclusive` – Function.

```
open_exclusive(path::String; mode, poll_interval, wait, stale_age, refresh) :: File
```

Create a new a file for read-write advisory-exclusive access. If `wait` is `false` then error out if the lock files exist otherwise block until we get the lock.

For a description of the keyword arguments, see `mkpidlock`.

`FileWatching.Pidfile.tryopen_exclusive` – Function.

```
tryopen_exclusive(path::String, mode::Integer = 0o444) :: Union{Void, File}
```

Try to create a new file for read-write advisory-exclusive access, return nothing if it already exists.

`FileWatching.Pidfile.write_pidfile` – Function.

```
write_pidfile(io, pid)
```

Write our pidfile format to an open IO descriptor.

`FileWatching.Pidfile.parse_pidfile` – Function.

```
parse_pidfile(file::Union{IO, String}) => (pid, hostname, age)
```

Attempt to parse our pidfile format, replaced an element with (0, "", 0.0), respectively, for any read that failed.

`FileWatching.Pidfile.stale_pidfile` – Function.

```
stale_pidfile(path::String, stale_age::Real, refresh::Real) :: Bool
```

Helper function for open_exclusive for deciding if a pidfile is stale.

`FileWatching.Pidfile.isvalidpid` – Function.

```
isvalidpid(hostname::String, pid::Cuint) :: Bool
```

Attempt to conservatively estimate whether pid is a valid process id.

`Base.Filesystem.touch` – Method.

```
Base.touch(::Pidfile.LockMonitor)
```

Update the `mtime` on the lock, to indicate it is still fresh.

See also the `refresh` keyword in the `mkpidlock` constructor.

Chapter 73

Future

The `Future` module implements future behavior of already existing functions, which will replace the current version in a future release of Julia.

`Future.copy!` – Function.

```
Future.copy!(dst, src) -> dst
```

Copy `src` into `dst`.

> **Julia 1.1**
>
> This function has moved to `Base` with Julia 1.1, consider using `copy!(dst, src)` instead. `Future.copy!` will be deprecated in the future.

`Future.randjump` – Function.

```
randjump(r::MersenneTwister, steps::Integer) -> MersenneTwister
```

Create an initialized `MersenneTwister` object, whose state is moved forward (without generating numbers) from `r` by `steps` steps. One such step corresponds to the generation of two `Float64` numbers. For each different value of `steps`, a large polynomial has to be generated internally. One is already pre-computed for `steps=big(10)^20`.

Chapter 74

Interactive Utilities

This module is intended for interactive work. It is loaded automatically in interactive mode.

`Base.Docs.apropos` – Function.

```
apropos([io::IO=stdout], pattern::Union{AbstractString,Regex})
```

Search available docstrings for entries containing `pattern`.

When `pattern` is a string, case is ignored. Results are printed to `io`.

`apropos` can be called from the help mode in the REPL by wrapping the query in double quotes:

```
help?> "pattern"
```

`InteractiveUtils.varinfo` – Function.

```
varinfo(m::Module=Main, pattern::Regex=r""; all=false, imported=false, recursive=false,
↪  sortby::Symbol=:name, minsize::Int=0)
```

Return a markdown table giving information about exported global variables in a module, optionally restricted to those matching `pattern`.

The memory consumption estimate is an approximate lower bound on the size of the internal structure of the object.

- `all` : also list non-exported objects defined in the module, deprecated objects, and compiler-generated objects.
- `imported` : also list objects explicitly imported from other modules.
- `recursive` : recursively include objects in sub-modules, observing the same settings in each.
- `sortby` : the column to sort results by. Options are `:name` (default), `:size`, and `:summary`.
- `minsize` : only includes objects with size at least `minsize` bytes. Defaults to `0`.

The output of `varinfo` is intended for display purposes only. See also `names` to get an array of symbols defined in a module, which is suitable for more general manipulations.

`InteractiveUtils.versioninfo` – Function.

```
versioninfo(io::IO=stdout; verbose::Bool=false)
```

Print information about the version of Julia in use. The output is controlled with boolean keyword arguments:

- `verbose`: print all additional information

> **Warning**
>
> The output of this function may contain sensitive information. Before sharing the output, please review the output and remove any data that should not be shared publicly.

See also: `VERSION`.

`InteractiveUtils.methodswith` – Function.

```
methodswith(typ[, module or function]; supertypes::Bool=false])
```

Return an array of methods with an argument of type `typ`.

The optional second argument restricts the search to a particular module or function (the default is all top-level modules).

If keyword `supertypes` is `true`, also return arguments with a parent type of `typ`, excluding type `Any`.

`InteractiveUtils.subtypes` – Function.

```
subtypes(T::DataType)
```

Return a list of immediate subtypes of DataType T. Note that all currently loaded subtypes are included, including those not visible in the current module.

See also `supertype`, `supertypes`, `methodswith`.

Examples

```
julia> subtypes(Integer)
3-element Vector{Any}:
 Bool
 Signed
 Unsigned
```

`InteractiveUtils.supertypes` – Function.

```
supertypes(T::Type)
```

Return a tuple (`T, ..., Any`) of T and all its supertypes, as determined by successive calls to the `supertype` function, listed in order of `<:` and terminated by `Any`.

See also `subtypes`.

Examples

```
julia> supertypes(Int)
(Int64, Signed, Integer, Real, Number, Any)
```

`InteractiveUtils.edit` – Method.

```
edit(path::AbstractString, line::Integer=0, column::Integer=0)
```

Edit a file or directory optionally providing a line number to edit the file at. Return to the `julia` prompt when you quit the editor. The editor can be changed by setting `JULIA_EDITOR`, `VISUAL` or `EDITOR` as an environment variable.

> **Julia 1.9**
>
> The `column` argument requires at least Julia 1.9.

See also `InteractiveUtils.define_editor`.

`InteractiveUtils.edit` – Method.

```
edit(function, [types])
edit(module)
```

Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit. For modules, open the main source file. The module needs to be loaded with `using` or `import` first.

> **Julia 1.1**
>
> `edit` on modules requires at least Julia 1.1.

To ensure that the file can be opened at the given line, you may need to call `define_editor` first.

`InteractiveUtils.@edit` – Macro.

```
@edit
```

Evaluates the arguments to the function or macro call, determines their types, and calls the `edit` function on the resulting expression.

See also: `@less`, `@which`.

`InteractiveUtils.define_editor` – Function.

```
define_editor(fn, pattern; wait=false)
```

Define a new editor matching `pattern` that can be used to open a file (possibly at a given line number) using `fn`.

The `fn` argument is a function that determines how to open a file with the given editor. It should take four arguments, as follows:

- `cmd` - a base command object for the editor
- `path` - the path to the source file to open
- `line` - the line number to open the editor at
- `column` - the column number to open the editor at

Editors which cannot open to a specific line with a command or a specific column may ignore the `line` and/or `column` argument. The `fn` callback must return either an appropriate `Cmd` object to open a file or `nothing` to indicate that they cannot edit this file. Use `nothing` to indicate that this editor is not appropriate for the current environment and another editor should be attempted. It is possible to add more general editing hooks that need not spawn external commands by pushing a callback directly to the vector `EDITOR_CALLBACKS`.

The `pattern` argument is a string, regular expression, or an array of strings and regular expressions. For the `fn` to be called, one of the patterns must match the value of `EDITOR`, `VISUAL` or `JULIA_EDITOR`. For strings, the string must equal the `basename` of the first word of the editor command, with its extension, if any, removed. E.g. ”vi” doesn’t match ”vim -g” but matches ”/usr/bin/vi -m”; it also matches `vi.exe`. If `pattern` is a regex it is matched against all of the editor command as a shell-escaped string. An array pattern matches if any of its items match. If multiple editors match, the one added most recently is used.

By default julia does not wait for the editor to close, running it in the background. However, if the editor is terminal based, you will probably want to set `wait=true` and julia will wait for the editor to close before resuming.

If one of the editor environment variables is set, but no editor entry matches it, the default editor entry is invoked:

```
(cmd, path, line, column) -> `$cmd $path`
```

Note that many editors are already defined. All of the following commands should already work:

- emacs
- emacsclient
- vim
- nvim
- nano
- micro
- kak
- helix
- textmate

- mate
- kate
- subl
- atom
- notepad++
- Visual Studio Code
- open
- pycharm
- bbedit

Example:

The following defines the usage of terminal-based `emacs`:

```
define_editor(
    r"\bemacs\b.*\s(-nw|--no-window-system)\b", wait=true) do cmd, path, line
    `$cmd +$line $path`
end
```

> **Julia 1.4**
>
> `define_editor` was introduced in Julia 1.4.

`InteractiveUtils.less` – Method.

```
less(file::AbstractString, [line::Integer])
```

Show a file using the default pager, optionally providing a starting line number. Returns to the `julia` prompt when you quit the pager.

`InteractiveUtils.less` – Method.

```
less(function, [types])
```

Show the definition of a function using the default pager, optionally specifying a tuple of types to indicate which method to see.

`InteractiveUtils.@less` – Macro.

```
@less
```

Evaluates the arguments to the function or macro call, determines their types, and calls the `less` function on the resulting expression.

See also: `@edit`, `@which`, `@code_lowered`.

`InteractiveUtils.@which` – Macro.

```
@which
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns the `Method` object for the method that would be called for those arguments. Applied to a variable, it returns the module in which the variable was bound. It calls out to the `which` function.

See also: `@less`, `@edit`.

`InteractiveUtils.@functionloc` – Macro.

```
@functionloc
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns a tuple `(filename,line)` giving the location for the method that would be called for those arguments. It calls out to the `functionloc` function.

`InteractiveUtils.@code_lowered` – Macro.

```
@code_lowered
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_lowered` on the resulting expression.

`InteractiveUtils.@code_typed` – Macro.

```
@code_typed
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_typed` on the resulting expression. Use the optional argument `optimize` with

```
@code_typed optimize=true foo(x)
```

to control whether additional optimizations, such as inlining, are also applied.

`InteractiveUtils.code_warntype` – Function.

```
code_warntype([io::IO], f, types; debuginfo=:default)
```

Prints lowered and type-inferred ASTs for the methods matching the given generic function and type signature to `io` which defaults to `stdout`. The ASTs are annotated in such a way as to cause "non-leaf" types which may be problematic for performance to be emphasized (if color is available, displayed in red). This serves as a warning of potential type instability.

Not all non-leaf types are particularly problematic for performance, and the performance characteristics of a particular type is an implementation detail of the compiler. `code_warntype` will err on the side of coloring types red if they might be a performance concern, so some types may be colored red even if they do not impact performance. Small unions of concrete types are usually not a concern, so these are highlighted in yellow.

Keyword argument `debuginfo` may be one of `:source` or `:none` (default), to specify the verbosity of code comments.

See the `@code_warntype` section in the Performance Tips page of the manual for more information.

`InteractiveUtils.@code_warntype` – Macro.

```
@code_warntype
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_warntype` on the resulting expression.

`InteractiveUtils.code_llvm` – Function.

```
code_llvm([io=stdout,], f, types; raw=false, dump_module=false, optimize=true,
↪  debuginfo=:default)
```

Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to `io`.

If the `optimize` keyword is unset, the code will be shown before LLVM optimizations. All metadata and dbg.* calls are removed from the printed bitcode. For the full IR, set the `raw` keyword to true. To dump the entire module that encapsulates the function (with declarations), set the `dump_module` keyword to true. Keyword argument `debuginfo` may be one of source (default) or none, to specify the verbosity of code comments.

`InteractiveUtils.@code_llvm` – Macro.

```
@code_llvm
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_llvm` on the resulting expression. Set the optional keyword arguments `raw`, `dump_module`, `debuginfo`, `optimize` by putting them and their value before the function call, like this:

```
@code_llvm raw=true dump_module=true debuginfo=:default f(x)
@code_llvm optimize=false f(x)
```

`optimize` controls whether additional optimizations, such as inlining, are also applied. `raw` makes all metadata and dbg.* calls visible. `debuginfo` may be one of `:source` (default) or `:none`, to specify the verbosity of code comments. `dump_module` prints the entire module that encapsulates the function.

`InteractiveUtils.code_native` – Function.

```
code_native([io=stdout,], f, types; syntax=:intel, debuginfo=:default, binary=false,
↪  dump_module=true)
```

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to `io`.

- Set assembly syntax by setting `syntax` to `:intel` (default) for intel syntax or `:att` for AT&T syntax.
- Specify verbosity of code comments by setting `debuginfo` to `:source` (default) or `:none`.
- If `binary` is `true`, also print the binary machine code for each instruction precedented by an abbreviated address.
- If `dump_module` is `false`, do not print metadata such as rodata or directives.
- If `raw` is `false`, uninteresting instructions (like the safepoint function prologue) are elided.

See also: `@code_native`, `code_llvm`, `code_typed` and `code_lowered`

`InteractiveUtils.@code_native` – Macro.

```
@code_native
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_native` on the resulting expression.

Set any of the optional keyword arguments `syntax`, `debuginfo`, `binary` or `dump_module` by putting it before the function call, like this:

```
@code_native syntax=:intel debuginfo=:default binary=true dump_module=false f(x)
```

- Set assembly syntax by setting `syntax` to `:intel` (default) for Intel syntax or `:att` for AT&T syntax.
- Specify verbosity of code comments by setting `debuginfo` to `:source` (default) or `:none`.
- If `binary` is `true`, also print the binary machine code for each instruction precedented by an abbreviated address.
- If `dump_module` is `false`, do not print metadata such as rodata or directives.

See also: `code_native`, `@code_llvm`, `@code_typed` and `@code_lowered`

`Base.@time_imports` – Macro.

```
@time_imports
```

A macro to execute an expression and produce a report of any time spent importing packages and their dependencies. Any compilation time will be reported as a percentage, and how much of which was recompilation, if any.

One line is printed per package or package extension. The duration shown is the time to import that package itself, not including the time to load any of its dependencies.

On Julia 1.9+ package extensions will show as Parent → Extension.

Note

During the load process a package sequentially imports all of its dependencies, not just its direct dependencies.

```
julia> @time_imports using CSV
     50.7 ms  Parsers 17.52% compilation time
      0.2 ms  DataValueInterfaces
      1.6 ms  DataAPI
      0.1 ms  IteratorInterfaceExtensions
      0.1 ms  TableTraits
     17.5 ms  Tables
     26.8 ms  PooledArrays
    193.7 ms  SentinelArrays 75.12% compilation time
      8.6 ms  InlineStrings
     20.3 ms  WeakRefStrings
      2.0 ms  TranscodingStreams
      1.4 ms  Zlib_jll
      1.8 ms  CodecZlib
      0.8 ms  Compat
     13.1 ms  FilePathsBase 28.39% compilation time
   1681.2 ms  CSV 92.40% compilation time
```

Julia 1.8

This macro requires at least Julia 1.8

`InteractiveUtils.clipboard` – Function.

```
clipboard(x)
```

Send a printed form of x to the operating system clipboard (”copy”).

```
clipboard() -> String
```

Return a string with the contents of the operating system clipboard (”paste”).

Chapter 75

Lazy Artifacts

In order for a package to download artifacts lazily, `LazyArtifacts` must be explicitly listed as a dependency of that package.

For further information on artifacts, see Artifacts.

Chapter 76

LibCURL

This is a simple Julia wrapper around http://curl.haxx.se/libcurl/ generated using Clang.jl. Please see the libcurl API documentation for help on how to use this package.

Chapter 77

LibGit2

The LibGit2 module provides bindings to libgit2, a portable C library that implements core functionality for the Git version control system. These bindings are currently used to power Julia's package manager. It is expected that this module will eventually be moved into a separate package.

Functionality

Some of this documentation assumes some prior knowledge of the libgit2 API. For more information on some of the objects and methods referenced here, consult the upstream libgit2 API reference.

`LibGit2.Buffer` – Type.

```
LibGit2.Buffer
```

A data buffer for exporting data from libgit2. Matches the `git_buf` struct.

When fetching data from LibGit2, a typical usage would look like:

```
buf_ref = Ref(Buffer())
@check ccall(..., (Ptr{Buffer},), buf_ref)
# operation on buf_ref
free(buf_ref)
```

In particular, note that `LibGit2.free` should be called afterward on the `Ref` object.

`LibGit2.CheckoutOptions` – Type.

```
LibGit2.CheckoutOptions
```

Matches the `git_checkout_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `checkout_strategy`: determine how to handle conflicts and whether to force the checkout/recreate missing files.

- `disable_filters`: if nonzero, do not apply filters like CLRF (to convert file newlines between UNIX and DOS).
- `dir_mode`: read/write/access mode for any directories involved in the checkout. Default is `0755`.
- `file_mode`: read/write/access mode for any files involved in the checkout. Default is `0755` or `0644`, depending on the blob.
- `file_open_flags`: bitflags used to open any files during the checkout.
- `notify_flags`: Flags for what sort of conflicts the user should be notified about.
- `notify_cb`: An optional callback function to notify the user if a checkout conflict occurs. If this function returns a non-zero value, the checkout will be cancelled.
- `notify_payload`: Payload for the notify callback function.
- `progress_cb`: An optional callback function to display checkout progress.
- `progress_payload`: Payload for the progress callback.
- `paths`: If not empty, describes which paths to search during the checkout. If empty, the checkout will occur over all files in the repository.
- `baseline`: Expected content of the `workdir`, captured in a (pointer to a) `GitTree`. Defaults to the state of the tree at HEAD.
- `baseline_index`: Expected content of the `workdir`, captured in a (pointer to a) `GitIndex`. Defaults to the state of the index at HEAD.
- `target_directory`: If not empty, checkout to this directory instead of the `workdir`.
- `ancestor_label`: In case of conflicts, the name of the common ancestor side.
- `our_label`: In case of conflicts, the name of "our" side.
- `their_label`: In case of conflicts, the name of "their" side.
- `perfdata_cb`: An optional callback function to display performance data.
- `perfdata_payload`: Payload for the performance callback.

`LibGit2.CloneOptions` – Type.

```
LibGit2.CloneOptions
```

Matches the `git_clone_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `checkout_opts`: The options for performing the checkout of the remote as part of the clone.
- `fetch_opts`: The options for performing the pre-checkout fetch of the remote as part of the clone.
- `bare`: If `0`, clone the full remote repository. If non-zero, perform a bare clone, in which there is no local copy of the source files in the repository and the `gitdir` and `workdir` are the same.
- `localclone`: Flag whether to clone a local object database or do a fetch. The default is to let git decide. It will not use the git-aware transport for a local clone, but will use it for URLs which begin with `file://`.
- `checkout_branch`: The name of the branch to checkout. If an empty string, the default branch of the remote will be checked out.

- `repository_cb`: An optional callback which will be used to create the *new* repository into which the clone is made.
- `repository_cb_payload`: The payload for the repository callback.
- `remote_cb`: An optional callback used to create the `GitRemote` before making the clone from it.
- `remote_cb_payload`: The payload for the remote callback.

`LibGit2.DescribeOptions` – Type.

```
LibGit2.DescribeOptions
```

Matches the `git_describe_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `max_candidates_tags`: consider this many most recent tags in `refs/tags` to describe a commit. Defaults to 10 (so that the 10 most recent tags would be examined to see if they describe a commit).
- `describe_strategy`: whether to consider all entries in `refs/tags` (equivalent to `git-describe --tags`) or all entries in `refs/` (equivalent to `git-describe --all`). The default is to only show annotated tags. If `Consts.DESCRIBE_TAGS` is passed, all tags, annotated or not, will be considered. If `Consts.DESCRIBE_ALL` is passed, any ref in `refs/` will be considered.
- `pattern`: only consider tags which match `pattern`. Supports glob expansion.
- `only_follow_first_parent`: when finding the distance from a matching reference to the described object, only consider the distance from the first parent.
- `show_commit_oid_as_fallback`: if no matching reference can be found which describes a commit, show the commit's `GitHash` instead of throwing an error (the default behavior).

`LibGit2.DescribeFormatOptions` – Type.

```
LibGit2.DescribeFormatOptions
```

Matches the `git_describe_format_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `abbreviated_size`: lower bound on the size of the abbreviated `GitHash` to use, defaulting to 7.
- `always_use_long_format`: set to 1 to use the long format for strings even if a short format can be used.
- `dirty_suffix`: if set, this will be appended to the end of the description string if the `workdir` is dirty.

`LibGit2.DiffDelta` – Type.

```
LibGit2.DiffDelta
```

Description of changes to one entry. Matches the `git_diff_delta` struct.

The fields represent:

- `status`: One of `Consts.DELTA_STATUS`, indicating whether the file has been added/modified/deleted.
- `flags`: Flags for the delta and the objects on each side. Determines whether to treat the file(s) as binary/text, whether they exist on each side of the diff, and whether the object ids are known to be correct.
- `similarity`: Used to indicate if a file has been renamed or copied.
- `nfiles`: The number of files in the delta (for instance, if the delta was run on a submodule commit id, it may contain more than one file).
- `old_file`: A `DiffFile` containing information about the file(s) before the changes.
- `new_file`: A `DiffFile` containing information about the file(s) after the changes.

`LibGit2.DiffFile` – Type.

```
LibGit2.DiffFile
```

Description of one side of a delta. Matches the `git_diff_file` struct.

The fields represent:

- `id`: the `GitHash` of the item in the diff. If the item is empty on this side of the diff (for instance, if the diff is of the removal of a file), this will be `GitHash(0)`.
- `path`: a `NULL` terminated path to the item relative to the working directory of the repository.
- `size`: the size of the item in bytes.
- `flags`: a combination of the `git_diff_flag_t` flags. The `i`th bit of this integer sets the `i`th flag.
- `mode`: the `stat` mode for the item.
- `id_abbrev`: only present in LibGit2 versions newer than or equal to `0.25.0`. The length of the `id` field when converted using `string`. Usually equal to `OID_HEXSZ` (40).

`LibGit2.DiffOptionsStruct` – Type.

```
LibGit2.DiffOptionsStruct
```

Matches the `git_diff_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always `1`.
- `flags`: flags controlling which files will appear in the diff. Defaults to `DIFF_NORMAL`.
- `ignore_submodules`: whether to look at files in submodules or not. Defaults to `SUBMODULE_IGNORE_UNSPECIFIED`, which means the submodule's configuration will control whether it appears in the diff or not.
- `pathspec`: path to files to include in the diff. Default is to use all files in the repository.
- `notify_cb`: optional callback which will notify the user of changes to the diff as file deltas are added to it.
- `progress_cb`: optional callback which will display diff progress. Only relevant on libgit2 versions at least as new as 0.24.0.
- `payload`: the payload to pass to `notify_cb` and `progress_cb`.

- `context_lines`: the number of *unchanged* lines used to define the edges of a hunk. This is also the number of lines which will be shown before/after a hunk to provide context. Default is 3.
- `interhunk_lines`: the maximum number of *unchanged* lines *between* two separate hunks allowed before the hunks will be combined. Default is 0.
- `id_abbrev`: sets the length of the abbreviated `GitHash` to print. Default is 7.
- `max_size`: the maximum file size of a blob. Above this size, it will be treated as a binary blob. The default is 512 MB.
- `old_prefix`: the virtual file directory in which to place old files on one side of the diff. Default is `"a"`.
- `new_prefix`: the virtual file directory in which to place new files on one side of the diff. Default is `"b"`.

`LibGit2.FetchHead` – Type.

```
LibGit2.FetchHead
```

Contains the information about HEAD during a fetch, including the name and URL of the branch fetched from, the oid of the HEAD, and whether the fetched HEAD has been merged locally.

The fields represent:

- `name`: The name in the local reference database of the fetch head, for example, `"refs/heads/master"`.
- `url`: The URL of the fetch head.
- `oid`: The `GitHash` of the tip of the fetch head.
- `ismerge`: Boolean flag indicating whether the changes at the remote have been merged into the local copy yet or not. If `true`, the local copy is up to date with the remote fetch head.

`LibGit2.FetchOptions` – Type.

```
LibGit2.FetchOptions
```

Matches the `git_fetch_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `callbacks`: remote callbacks to use during the fetch.
- `prune`: whether to perform a prune after the fetch or not. The default is to use the setting from the `GitConfig`.
- `update_fetchhead`: whether to update the `FetchHead` after the fetch. The default is to perform the update, which is the normal git behavior.
- `download_tags`: whether to download tags present at the remote or not. The default is to request the tags for objects which are being downloaded anyway from the server.
- `proxy_opts`: options for connecting to the remote through a proxy. See `ProxyOptions`. Only present on libgit2 versions newer than or equal to 0.25.0.
- `custom_headers`: any extra headers needed for the fetch. Only present on libgit2 versions newer than or equal to 0.24.0.

`LibGit2.GitAnnotated` – Type.

```
GitAnnotated(repo::GitRepo, commit_id::GitHash)
GitAnnotated(repo::GitRepo, ref::GitReference)
GitAnnotated(repo::GitRepo, fh::FetchHead)
GitAnnotated(repo::GitRepo, committish::AbstractString)
```

An annotated git commit carries with it information about how it was looked up and why, so that rebase or merge operations have more information about the context of the commit. Conflict files contain information about the source/target branches in the merge which are conflicting, for instance. An annotated commit can refer to the tip of a remote branch, for instance when a `FetchHead` is passed, or to a branch head described using `GitReference`.

`LibGit2.GitBlame` – Type.

```
GitBlame(repo::GitRepo, path::AbstractString; options::BlameOptions=BlameOptions())
```

Construct a `GitBlame` object for the file at `path`, using change information gleaned from the history of `repo`. The `GitBlame` object records who changed which chunks of the file when, and how. `options` controls how to separate the contents of the file and which commits to probe - see `BlameOptions` for more information.

`LibGit2.GitBlob` – Type.

```
GitBlob(repo::GitRepo, hash::AbstractGitHash)
GitBlob(repo::GitRepo, spec::AbstractString)
```

Return a `GitBlob` object from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.GitCommit` – Type.

```
GitCommit(repo::GitRepo, hash::AbstractGitHash)
GitCommit(repo::GitRepo, spec::AbstractString)
```

Return a `GitCommit` object from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.GitConfig` – Type.

```
GitConfig(path::AbstractString, level::Consts.GIT_CONFIG=Consts.CONFIG_LEVEL_APP,
↪  force::Bool=false)
```

Create a new `GitConfig` by loading configuration information from the file at `path`. See `addfile` for more information about the `level`, `repo` and `force` options.

```
GitConfig(repo::GitRepo)
```

Get the stored configuration for the git repository `repo`. If `repo` does not have a specific configuration file set, the default git configuration will be used.

```
GitConfig(level::Consts.GIT_CONFIG=Consts.CONFIG_LEVEL_DEFAULT)
```

Get the default git configuration by loading the global and system configuration files into a prioritized configuration. This can be used to access default configuration options outside a specific git repository.

`LibGit2.GitHash` – Type.

```
GitHash
```

A git object identifier, based on the sha-1 hash. It is a 20 byte string (40 hex digits) used to identify a `GitObject` in a repository.

`LibGit2.GitObject` – Type.

```
GitObject(repo::GitRepo, hash::AbstractGitHash)
GitObject(repo::GitRepo, spec::AbstractString)
```

Return the specified object (`GitCommit`, `GitBlob`, `GitTree` or `GitTag`) from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.GitRemote` – Type.

```
GitRemote(repo::GitRepo, rmt_name::AbstractString, rmt_url::AbstractString) -> GitRemote
```

Look up a remote git repository using its name and URL. Uses the default fetch refspec.

Examples

```
repo = LibGit2.init(repo_path)
remote = LibGit2.GitRemote(repo, "upstream", repo_url)
```

```
GitRemote(repo::GitRepo, rmt_name::AbstractString, rmt_url::AbstractString,
↪  fetch_spec::AbstractString) -> GitRemote
```

Look up a remote git repository using the repository's name and URL, as well as specifications for how to fetch from the remote (e.g. which remote branch to fetch from).

Examples

```
repo = LibGit2.init(repo_path)
refspec = "+refs/heads/mybranch:refs/remotes/origin/mybranch"
remote = LibGit2.GitRemote(repo, "upstream", repo_url, refspec)
```

`LibGit2.GitRemoteAnon` – Function.

```
GitRemoteAnon(repo::GitRepo, url::AbstractString) -> GitRemote
```

Look up a remote git repository using only its URL, not its name.

Examples

```
repo = LibGit2.init(repo_path)
remote = LibGit2.GitRemoteAnon(repo, repo_url)
```

`LibGit2.GitRepo` – Type.

```
LibGit2.GitRepo(path::AbstractString)
```

Open a git repository at `path`.

`LibGit2.GitRepoExt` – Function.

```
LibGit2.GitRepoExt(path::AbstractString, flags::Cuint = Cuint(Consts.REPOSITORY_OPEN_DEFAULT))
```

Open a git repository at `path` with extended controls (for instance, if the current user must be a member of a special access group to read `path`).

`LibGit2.GitRevWalker` – Type.

```
GitRevWalker(repo::GitRepo)
```

A `GitRevWalker` *walks* through the *revisions* (i.e. commits) of a git repository `repo`. It is a collection of the commits in the repository, and supports iteration and calls to `LibGit2.map` and `LibGit2.count` (for instance, `LibGit2.count` could be used to determine what percentage of commits in a repository were made by a certain author).

```
cnt = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    LibGit2.count((oid,repo)->(oid == commit_oid1), walker, oid=commit_oid1,
    ↪  by=LibGit2.Consts.SORT_TIME)
end
```

Here, `LibGit2.count` finds the number of commits along the walk with a certain `GitHash`. Since the `GitHash` is unique to a commit, `cnt` will be 1.

`LibGit2.GitShortHash` – Type.

```
GitShortHash(hash::GitHash, len::Integer)
```

A shortened git object identifier, which can be used to identify a git object when it is unique, consisting of the initial `len` hexadecimal digits of `hash` (the remaining digits are ignored).

`LibGit2.GitSignature` – Type.

```
LibGit2.GitSignature
```

This is a Julia wrapper around a pointer to a `git_signature` object.

`LibGit2.GitStatus` – Type.

```
LibGit2.GitStatus(repo::GitRepo; status_opts=StatusOptions())
```

Collect information about the status of each file in the git repository `repo` (e.g. is the file modified, staged, etc.). `status_opts` can be used to set various options, for instance whether or not to look at untracked files or whether to include submodules or not. See `StatusOptions` for more information.

`LibGit2.GitTag` – Type.

```
GitTag(repo::GitRepo, hash::AbstractGitHash)
GitTag(repo::GitRepo, spec::AbstractString)
```

Return a `GitTag` object from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.GitTree` – Type.

```
GitTree(repo::GitRepo, hash::AbstractGitHash)
GitTree(repo::GitRepo, spec::AbstractString)
```

Return a `GitTree` object from `repo` specified by `hash`/`spec`.

- `hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.
- `spec` is a textual specification: see the git docs for a full list.

`LibGit2.IndexEntry` – Type.

```
LibGit2.IndexEntry
```

In-memory representation of a file entry in the index. Matches the `git_index_entry` struct.

`LibGit2.IndexTime` – Type.

```
LibGit2.IndexTime
```

Matches the `git_index_time` struct.

`LibGit2.BlameOptions` – Type.

```
LibGit2.BlameOptions
```

Matches the `git_blame_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `flags`: one of `Consts.BLAME_NORMAL` or `Consts.BLAME_FIRST_PARENT` (the other blame flags are not yet implemented by libgit2).
- `min_match_characters`: the minimum number of *alphanumeric* characters which much change in a commit in order for the change to be associated with that commit. The default is 20. Only takes effect if one of the `Consts.BLAME_*_COPIES` flags are used, which libgit2 does not implement yet.
- `newest_commit`: the `GitHash` of the newest commit from which to look at changes.
- `oldest_commit`: the `GitHash` of the oldest commit from which to look at changes.
- `min_line`: the first line of the file from which to starting blaming. The default is 1.
- `max_line`: the last line of the file to which to blame. The default is `0`, meaning the last line of the file.

`LibGit2.MergeOptions` – Type.

```
LibGit2.MergeOptions
```

Matches the `git_merge_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `flags`: an `enum` for flags describing merge behavior. Defined in `git_merge_flag_t`. The corresponding Julia enum is `GIT_MERGE` and has values:
  - `MERGE_FIND_RENAMES`: detect if a file has been renamed between the common ancestor and the "ours" or "theirs" side of the merge. Allows merges where a file has been renamed.
  - `MERGE_FAIL_ON_CONFLICT`: exit immediately if a conflict is found rather than trying to resolve it.
  - `MERGE_SKIP_REUC`: do not write the REUC extension on the index resulting from the merge.

  - `MERGE_NO_RECURSIVE`: if the commits being merged have multiple merge bases, use the first one, rather than trying to recursively merge the bases.

- `rename_threshold`: how similar two files must to consider one a rename of the other. This is an integer that sets the percentage similarity. The default is 50.
- `target_limit`: the maximum number of files to compare with to look for renames. The default is 200.
- `metric`: optional custom function to use to determine the similarity between two files for rename detection.
- `recursion_limit`: the upper limit on the number of merges of common ancestors to perform to try to build a new virtual merge base for the merge. The default is no limit. This field is only present on libgit2 versions newer than 0.24.0.
- `default_driver`: the merge driver to use if both sides have changed. This field is only present on libgit2 versions newer than 0.25.0.
- `file_favor`: how to handle conflicting file contents for the `text` driver.
  - `MERGE_FILE_FAVOR_NORMAL`: if both sides of the merge have changes to a section, make a note of the conflict in the index which `git checkout` will use to create a merge file, which the user can then reference to resolve the conflicts. This is the default.
  - `MERGE_FILE_FAVOR_OURS`: if both sides of the merge have changes to a section, use the version in the "ours" side of the merge in the index.
  - `MERGE_FILE_FAVOR_THEIRS`: if both sides of the merge have changes to a section, use the version in the "theirs" side of the merge in the index.
  - `MERGE_FILE_FAVOR_UNION`: if both sides of the merge have changes to a section, include each unique line from both sides in the file which is put into the index.
- `file_flags`: guidelines for merging files.

`LibGit2.ProxyOptions` – Type.

```
LibGit2.ProxyOptions
```

Options for connecting through a proxy.

Matches the `git_proxy_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `proxytype`: an `enum` for the type of proxy to use. Defined in `git_proxy_t`. The corresponding Julia enum is `GIT_PROXY` and has values:
  - `PROXY_NONE`: do not attempt the connection through a proxy.
  - `PROXY_AUTO`: attempt to figure out the proxy configuration from the git configuration.
  - `PROXY_SPECIFIED`: connect using the URL given in the `url` field of this struct.

  Default is to auto-detect the proxy type.
- `url`: the URL of the proxy.
- `credential_cb`: a pointer to a callback function which will be called if the remote requires authentication to connect.

- certificate_cb: a pointer to a callback function which will be called if certificate verification fails. This lets the user decide whether or not to keep connecting. If the function returns 1, connecting will be allowed. If it returns 0, the connection will not be allowed. A negative value can be used to return errors.
- payload: the payload to be provided to the two callback functions.

Examples

```
julia> fo = LibGit2.FetchOptions(
           proxy_opts = LibGit2.ProxyOptions(url = Cstring("https://my_proxy_url.com")))

julia> fetch(remote, "master", options=fo)
```

LibGit2.PushOptions – Type.

```
LibGit2.PushOptions
```

Matches the git_push_options struct.

The fields represent:

- version: version of the struct in use, in case this changes later. For now, always 1.
- parallelism: if a pack file must be created, this variable sets the number of worker threads which will be spawned by the packbuilder. If 0, the packbuilder will auto-set the number of threads to use. The default is 1.
- callbacks: the callbacks (e.g. for authentication with the remote) to use for the push.
- proxy_opts: only relevant if the LibGit2 version is greater than or equal to 0.25.0. Sets options for using a proxy to communicate with a remote. See ProxyOptions for more information.
- custom_headers: only relevant if the LibGit2 version is greater than or equal to 0.24.0. Extra headers needed for the push operation.

LibGit2.RebaseOperation – Type.

```
LibGit2.RebaseOperation
```

Describes a single instruction/operation to be performed during the rebase. Matches the git_rebase_operation struct.

The fields represent:

- optype: the type of rebase operation currently being performed. The options are:
  - REBASE_OPERATION_PICK: cherry-pick the commit in question.
  - REBASE_OPERATION_REWORD: cherry-pick the commit in question, but rewrite its message using the prompt.
  - REBASE_OPERATION_EDIT: cherry-pick the commit in question, but allow the user to edit the commit's contents and its message.

  - REBASE_OPERATION_SQUASH: squash the commit in question into the previous commit. The commit messages of the two commits will be merged.
  - REBASE_OPERATION_FIXUP: squash the commit in question into the previous commit. Only the commit message of the previous commit will be used.
  - REBASE_OPERATION_EXEC: do not cherry-pick a commit. Run a command and continue if the command exits successfully.
- `id`: the `GitHash` of the commit being worked on during this rebase step.
- `exec`: in case REBASE_OPERATION_EXEC is used, the command to run during this step (for instance, running the test suite after each commit).

`LibGit2.RebaseOptions` – Type.

```
LibGit2.RebaseOptions
```

Matches the `git_rebase_options` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `quiet`: inform other git clients helping with/working on the rebase that the rebase should be done "quietly". Used for interoperability. The default is 1.
- `inmemory`: start an in-memory rebase. Callers working on the rebase can go through its steps and commit any changes, but cannot rewind HEAD or update the repository. The `workdir` will not be modified. Only present on libgit2 versions newer than or equal to 0.24.0.
- `rewrite_notes_ref`: name of the reference to notes to use to rewrite the commit notes as the rebase is finished.
- `merge_opts`: merge options controlling how the trees will be merged at each rebase step. Only present on libgit2 versions newer than or equal to 0.24.0.
- `checkout_opts`: checkout options for writing files when initializing the rebase, stepping through it, and aborting it. See `CheckoutOptions` for more information.

`LibGit2.RemoteCallbacks` – Type.

```
LibGit2.RemoteCallbacks
```

Callback settings. Matches the `git_remote_callbacks` struct.

`LibGit2.SignatureStruct` – Type.

```
LibGit2.SignatureStruct
```

An action signature (e.g. for committers, taggers, etc). Matches the `git_signature` struct.

The fields represent:

- `name`: The full name of the committer or author of the commit.

- `email`: The email at which the committer/author can be contacted.
- `when`: a `TimeStruct` indicating when the commit was authored/committed into the repository.

`LibGit2.StatusEntry` – Type.

```
LibGit2.StatusEntry
```

Providing the differences between the file as it exists in HEAD and the index, and providing the differences between the index and the working directory. Matches the `git_status_entry` struct.

The fields represent:

- `status`: contains the status flags for the file, indicating if it is current, or has been changed in some way in the index or work tree.
- `head_to_index`: a pointer to a `DiffDelta` which encapsulates the difference(s) between the file as it exists in HEAD and in the index.
- `index_to_workdir`: a pointer to a `DiffDelta` which encapsulates the difference(s) between the file as it exists in the index and in the `workdir`.

`LibGit2.StatusOptions` – Type.

```
LibGit2.StatusOptions
```

Options to control how `git_status_foreach_ext()` will issue callbacks. Matches the `git_status_opt_t` struct.

The fields represent:

- `version`: version of the struct in use, in case this changes later. For now, always 1.
- `show`: a flag for which files to examine and in which order. The default is `Consts.STATUS_SHOW_INDEX_AND_WORKDIR`.
- `flags`: flags for controlling any callbacks used in a status call.
- `pathspec`: an array of paths to use for path-matching. The behavior of the path-matching will vary depending on the values of `show` and `flags`.
- The `baseline` is the tree to be used for comparison to the working directory and index; defaults to HEAD.

`LibGit2.StrArrayStruct` – Type.

```
LibGit2.StrArrayStruct
```

A LibGit2 representation of an array of strings. Matches the `git_strarray` struct.

When fetching data from LibGit2, a typical usage would look like:

```
sa_ref = Ref(StrArrayStruct())
@check ccall(..., (Ptr{StrArrayStruct},), sa_ref)
res = convert(Vector{String}, sa_ref[])
free(sa_ref)
```

In particular, note that `LibGit2.free` should be called afterward on the `Ref` object.

Conversely, when passing a vector of strings to LibGit2, it is generally simplest to rely on implicit conversion:

```
strs = String[...]
@check ccall(..., (Ptr{StrArrayStruct},), strs)
```

Note that no call to `free` is required as the data is allocated by Julia.

`LibGit2.TimeStruct` – Type.

```
LibGit2.TimeStruct
```

Time in a signature. Matches the `git_time` struct.

`LibGit2.addfile` – Function.

```
addfile(cfg::GitConfig, path::AbstractString,
        level::Consts.GIT_CONFIG=Consts.CONFIG_LEVEL_APP,
        repo::Union{GitRepo, Nothing} = nothing,
        force::Bool=false)
```

Add an existing git configuration file located at `path` to the current `GitConfig` `cfg`. If the file does not exist, it will be created.

- `level` sets the git configuration priority level and is determined by

`Consts.GIT_CONFIG`.

- `repo` is an optional repository to allow parsing of conditional includes.
- If `force` is `false` and a configuration for the given priority level already exists,

`addfile` will error. If `force` is `true`, the existing configuration will be replaced by the one in the file at `path`.

`LibGit2.add!` – Function.

```
add!(repo::GitRepo, files::AbstractString...; flags::Cuint = Consts.INDEX_ADD_DEFAULT)
add!(idx::GitIndex, files::AbstractString...; flags::Cuint = Consts.INDEX_ADD_DEFAULT)
```

Add all the files with paths specified by `files` to the index `idx` (or the index of the `repo`). If the file already exists, the index entry will be updated. If the file does not exist already, it will be newly added into the index. `files` may contain glob patterns which will be expanded and any matching files will be added (unless `INDEX_ADD_DISABLE_PATHSPEC_MATCH` is set, see below). If a file has been ignored (in `.gitignore` or in the config), it *will not* be added, *unless* it is already being tracked in the index, in which case it *will* be updated. The keyword argument `flags` is a set of bit-flags which control the behavior with respect to ignored files:

- `Consts.INDEX_ADD_DEFAULT` - default, described above.
- `Consts.INDEX_ADD_FORCE` - disregard the existing ignore rules and force addition of the file to the index even if it is already ignored.
- `Consts.INDEX_ADD_CHECK_PATHSPEC` - cannot be used at the same time as `INDEX_ADD_FORCE`. Check that each file in `files` which exists on disk is not in the ignore list. If one of the files *is* ignored, the function will return `EINVALIDSPEC`.
- `Consts.INDEX_ADD_DISABLE_PATHSPEC_MATCH` - turn off glob matching, and only add files to the index which exactly match the paths specified in `files`.

`LibGit2.add_fetch!` – Function.

```
add_fetch!(repo::GitRepo, rmt::GitRemote, fetch_spec::String)
```

Add a *fetch* refspec for the specified `rmt`. This refspec will contain information about which branch(es) to fetch from.

Examples

```
julia> LibGit2.add_fetch!(repo, remote, "upstream");

julia> LibGit2.fetch_refspecs(remote)
String["+refs/heads/*:refs/remotes/upstream/*"]
```

`LibGit2.add_push!` – Function.

```
add_push!(repo::GitRepo, rmt::GitRemote, push_spec::String)
```

Add a *push* refspec for the specified `rmt`. This refspec will contain information about which branch(es) to push to.

Examples

```
julia> LibGit2.add_push!(repo, remote, "refs/heads/master");

julia> remote = LibGit2.get(LibGit2.GitRemote, repo, branch);

julia> LibGit2.push_refspecs(remote)
String["refs/heads/master"]
```

Note

You may need to `close` and reopen the `GitRemote` in question after updating its push refspecs in order for the change to take effect and for calls to `push` to work.

`LibGit2.addblob!` – Function.

```
LibGit2.addblob!(repo::GitRepo, path::AbstractString)
```

Read the file at `path` and adds it to the object database of `repo` as a loose blob. Return the `GitHash` of the resulting blob.

Examples

```
hash_str = string(commit_oid)
blob_file = joinpath(repo_path, ".git", "objects", hash_str[1:2], hash_str[3:end])
id = LibGit2.addblob!(repo, blob_file)
```

`LibGit2.author` – Function.

```
author(c::GitCommit)
```

Return the `Signature` of the author of the commit `c`. The author is the person who made changes to the relevant file(s). See also `committer`.

`LibGit2.authors` – Function.

```
authors(repo::GitRepo) -> Vector{Signature}
```

Return all authors of commits to the `repo` repository.

Examples

```
repo = LibGit2.GitRepo(repo_path)
repo_file = open(joinpath(repo_path, test_file), "a")

println(repo_file, commit_msg)
flush(repo_file)
LibGit2.add!(repo, test_file)
sig = LibGit2.Signature("TEST", "TEST@TEST.COM", round(time(), 0), 0)
commit_oid1 = LibGit2.commit(repo, "commit1"; author=sig, committer=sig)
println(repo_file, randstring(10))
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid2 = LibGit2.commit(repo, "commit2"; author=sig, committer=sig)

# will be a Vector of [sig, sig]
auths = LibGit2.authors(repo)
```

`LibGit2.branch` – Function.

```
branch(repo::GitRepo)
```

Equivalent to `git branch`. Create a new branch from the current HEAD.

`LibGit2.branch!` – Function.

```
branch!(repo::GitRepo, branch_name::AbstractString, commit::AbstractString=""; kwargs...)
```

Checkout a new git branch in the `repo` repository. `commit` is the `GitHash`, in string form, which will be the start of the new branch. If `commit` is an empty string, the current HEAD will be used.

The keyword arguments are:

- `track::AbstractString=""`: the name of the remote branch this new branch should track, if any. If empty (the default), no remote branch will be tracked.
- `force::Bool=false`: if `true`, branch creation will be forced.
- `set_head::Bool=true`: if `true`, after the branch creation finishes the branch head will be set as the HEAD of `repo`.

Equivalent to `git checkout [-b|-B] <branch_name> [<commit>] [--track <track>]`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.branch!(repo, "new_branch", set_head=false)
```

`LibGit2.checkout!` – Function.

```
checkout!(repo::GitRepo, commit::AbstractString=""; force::Bool=true)
```

Equivalent to `git checkout [-f] --detach <commit>`. Checkout the git commit `commit` (a `GitHash` in string form) in `repo`. If `force` is `true`, force the checkout and discard any current changes. Note that this detaches the current HEAD.

Examples

```
repo = LibGit2.GitRepo(repo_path)
open(joinpath(LibGit2.path(repo), "file1"), "w") do f
    write(f, "111
")
end
LibGit2.add!(repo, "file1")
commit_oid = LibGit2.commit(repo, "add file1")
open(joinpath(LibGit2.path(repo), "file1"), "w") do f
    write(f, "112
")
end
# would fail without the force=true
# since there are modifications to the file
LibGit2.checkout!(repo, string(commit_oid), force=true)
```

`LibGit2.clone` – Function.

```
clone(repo_url::AbstractString, repo_path::AbstractString, clone_opts::CloneOptions)
```

Clone the remote repository at `repo_url` (which can be a remote URL or a path on the local filesystem) to `repo_path` (which must be a path on the local filesystem). Options for the clone, such as whether to perform a bare clone or not, are set by `CloneOptions`.

Examples

```
repo_url = "https://github.com/JuliaLang/Example.jl"
repo = LibGit2.clone(repo_url, "/home/me/projects/Example")
```

```
clone(repo_url::AbstractString, repo_path::AbstractString; kwargs...)
```

Clone a remote repository located at `repo_url` to the local filesystem location `repo_path`.

The keyword arguments are:

- `branch::AbstractString=""`: which branch of the remote to clone, if not the default repository branch (usually `master`).
- `isbare::Bool=false`: if `true`, clone the remote as a bare repository, which will make `repo_path` itself the git directory instead of `repo_path/.git`. This means that a working tree cannot be checked out. Plays the role of the git CLI argument `--bare`.
- `remote_cb::Ptr{Cvoid}=C_NULL`: a callback which will be used to create the remote before it is cloned. If `C_NULL` (the default), no attempt will be made to create the remote - it will be assumed to already exist.
- `credentials::Creds=nothing`: provides credentials and/or settings when authenticating against a private repository.
- `callbacks::Callbacks=Callbacks()`: user provided callbacks and payloads.

Equivalent to `git clone [-b <branch>] [--bare] <repo_url> <repo_path>`.

Examples

```
repo_url = "https://github.com/JuliaLang/Example.jl"
repo1 = LibGit2.clone(repo_url, "test_path")
repo2 = LibGit2.clone(repo_url, "test_path", isbare=true)
julia_url = "https://github.com/JuliaLang/julia"
julia_repo = LibGit2.clone(julia_url, "julia_path", branch="release-0.6")
```

`LibGit2.commit` – Function.

```
commit(repo::GitRepo, msg::AbstractString; kwargs...) -> GitHash
```

Wrapper around `git_commit_create`. Create a commit in the repository `repo`. `msg` is the commit message. Return the OID of the new commit.

The keyword arguments are:

- refname::AbstractString=Consts.HEAD_FILE: if not NULL, the name of the reference to update to point to the new commit. For example, "HEAD" will update the HEAD of the current branch. If the reference does not yet exist, it will be created.
- author::Signature = Signature(repo) is a Signature containing information about the person who authored the commit.
- committer::Signature = Signature(repo) is a Signature containing information about the person who committed the commit to the repository. Not necessarily the same as author, for instance if author emailed a patch to committer who committed it.
- tree_id::GitHash = GitHash() is a git tree to use to create the commit, showing its ancestry and relationship with any other history. tree must belong to repo.
- parent_ids::Vector{GitHash}=GitHash[] is a list of commits by GitHash to use as parent commits for the new one, and may be empty. A commit might have multiple parents if it is a merge commit, for example.

```
LibGit2.commit(rb::GitRebase, sig::GitSignature)
```

Commit the current patch to the rebase rb, using sig as the committer. Is silent if the commit has already been applied.

LibGit2.committer – Function.

```
committer(c::GitCommit)
```

Return the Signature of the committer of the commit c. The committer is the person who committed the changes originally authored by the author, but need not be the same as the author, for example, if the author emailed a patch to a committer who committed it.

LibGit2.count – Function.

```
LibGit2.count(f::Function, walker::GitRevWalker; oid::GitHash=GitHash(),
↪   by::Cint=Consts.SORT_NONE, rev::Bool=false)
```

Using the GitRevWalker walker to "walk" over every commit in the repository's history, find the number of commits which return true when f is applied to them. The keyword arguments are: * oid: The GitHash of the commit to begin the walk from. The default is to use push_head! and therefore the HEAD commit and all its ancestors. * by: The sorting method. The default is not to sort. Other options are to sort by topology (LibGit2.Consts.SORT_TOPOLOGICAL), to sort forwards in time (LibGit2.Consts.SORT_TIME, most ancient first) or to sort backwards in time (LibGit2.Consts.SORT_REVERSE, most recent first). * rev: Whether to reverse the sorted order (for instance, if topological sorting is used).

Examples

```
cnt = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    LibGit2.count((oid, repo)->(oid == commit_oid1), walker, oid=commit_oid1,
    ↪   by=LibGit2.Consts.SORT_TIME)
end
```

`LibGit2.count` finds the number of commits along the walk with a certain `GitHash` `commit_oid1`, starting the walk from that commit and moving forwards in time from it. Since the `GitHash` is unique to a commit, `cnt` will be 1.

`LibGit2.counthunks` – Function.

```
counthunks(blame::GitBlame)
```

Return the number of distinct "hunks" with a file. A hunk may contain multiple lines. A hunk is usually a piece of a file that was added/changed/removed together, for example, a function added to a source file or an inner loop that was optimized out of that function later.

`LibGit2.create_branch` – Function.

```
LibGit2.create_branch(repo::GitRepo, bname::AbstractString, commit_obj::GitCommit;
↪  force::Bool=false)
```

Create a new branch in the repository `repo` with name `bname`, which points to commit `commit_obj` (which has to be part of `repo`). If `force` is `true`, overwrite an existing branch named `bname` if it exists. If `force` is `false` and a branch already exists named `bname`, this function will throw an error.

`LibGit2.credentials_callback` – Function.

```
credential_callback(...) -> Cint
```

A LibGit2 credential callback function which provides different credential acquisition functionality w.r.t. a connection protocol. The `payload_ptr` is required to contain a `LibGit2.CredentialPayload` object which will keep track of state and settings.

The `allowed_types` contains a bitmask of `LibGit2.Consts.GIT_CREDTYPE` values specifying which authentication methods should be attempted.

Credential authentication is done in the following order (if supported):

- SSH agent
- SSH private/public key pair
- Username/password plain text

If a user is presented with a credential prompt they can abort the prompt by typing `^D` (pressing the control key together with the `d` key).

Note: Due to the specifics of the `libgit2` authentication procedure, when authentication fails, this function is called again without any indication whether authentication was successful or not. To avoid an infinite loop from repeatedly using the same faulty credentials, we will keep track of state using the payload.

For addition details see the LibGit2 guide on authenticating against a server.

`LibGit2.credentials_cb` – Function.

C function pointer for `credentials_callback`

`LibGit2.default_signature` – Function.

Return signature object. Free it after use.

`LibGit2.delete_branch` – Function.

```
LibGit2.delete_branch(branch::GitReference)
```

Delete the branch pointed to by `branch`.

`LibGit2.diff_files` – Function.

```
diff_files(repo::GitRepo, branch1::AbstractString, branch2::AbstractString; kwarg...) ->
↪  Vector{AbstractString}
```

Show which files have changed in the git repository `repo` between branches `branch1` and `branch2`.

The keyword argument is:

- `filter::Set{Consts.DELTA_STATUS}=Set([Consts.DELTA_ADDED, Consts.DELTA_MODIFIED, Consts.DELTA_DELET`
  and it sets options for the diff. The default is to show files added, modified, or deleted.

Return only the *names* of the files which have changed, *not* their contents.

Examples

```
LibGit2.branch!(repo, "branch/a")
LibGit2.branch!(repo, "branch/b")
# add a file to repo
open(joinpath(LibGit2.path(repo),"file"),"w") do f
    write(f, "hello repo
")
end
LibGit2.add!(repo, "file")
LibGit2.commit(repo, "add file")
# returns ["file"]
filt = Set([LibGit2.Consts.DELTA_ADDED])
files = LibGit2.diff_files(repo, "branch/a", "branch/b", filter=filt)
# returns [] because existing files weren't modified
filt = Set([LibGit2.Consts.DELTA_MODIFIED])
files = LibGit2.diff_files(repo, "branch/a", "branch/b", filter=filt)
```

Equivalent to `git diff --name-only --diff-filter=<filter> <branch1> <branch2>`.

`LibGit2.entryid` – Function.

```
entryid(te::GitTreeEntry)
```

Return the `GitHash` of the object to which `te` refers.

`LibGit2.entrytype` – Function.

```
entrytype(te::GitTreeEntry)
```

Return the type of the object to which `te` refers. The result will be one of the types which `objtype` returns, e.g. a `GitTree` or `GitBlob`.

`LibGit2.fetch` – Function.

```
fetch(rmt::GitRemote, refspecs; options::FetchOptions=FetchOptions(), msg="")
```

Fetch from the specified `rmt` remote git repository, using `refspecs` to determine which remote branch(es) to fetch. The keyword arguments are:

- `options`: determines the options for the fetch, e.g. whether to prune afterwards. See `FetchOptions` for more information.
- `msg`: a message to insert into the reflogs.

```
fetch(repo::GitRepo; kwargs...)
```

Fetches updates from an upstream of the repository `repo`.

The keyword arguments are:

- `remote::AbstractString="origin"`: which remote, specified by name, of `repo` to fetch from. If this is empty, the URL will be used to construct an anonymous remote.
- `remoteurl::AbstractString=""`: the URL of `remote`. If not specified, will be assumed based on the given name of `remote`.
- `refspecs=AbstractString[]`: determines properties of the fetch.
- `credentials=nothing`: provides credentials and/or settings when authenticating against a private `remote`.
- `callbacks=Callbacks()`: user provided callbacks and payloads.

Equivalent to `git fetch [<remoteurl>|<repo>] [<refspecs>]`.

`LibGit2.fetchheads` – Function.

```
fetchheads(repo::GitRepo) -> Vector{FetchHead}
```

Return the list of all the fetch heads for `repo`, each represented as a `FetchHead`, including their names, URLs, and merge statuses.

Examples

```
julia> fetch_heads = LibGit2.fetchheads(repo);

julia> fetch_heads[1].name
"refs/heads/master"

julia> fetch_heads[1].ismerge
true

julia> fetch_heads[2].name
"refs/heads/test_branch"

julia> fetch_heads[2].ismerge
false
```

`LibGit2.fetch_refspecs` – Function.

```
fetch_refspecs(rmt::GitRemote) -> Vector{String}
```

Get the *fetch* refspecs for the specified `rmt`. These refspecs contain information about which branch(es) to fetch from.

Examples

```
julia> remote = LibGit2.get(LibGit2.GitRemote, repo, "upstream");

julia> LibGit2.add_fetch!(repo, remote, "upstream");

julia> LibGit2.fetch_refspecs(remote)
String["+refs/heads/*:refs/remotes/upstream/*"]
```

`LibGit2.fetchhead_foreach_cb` – Function.

C function pointer for `fetchhead_foreach_callback`

`LibGit2.merge_base` – Function.

```
merge_base(repo::GitRepo, one::AbstractString, two::AbstractString) -> GitHash
```

Find a merge base (a common ancestor) between the commits one and `two`. one and `two` may both be in string form. Return the `GitHash` of the merge base.

`LibGit2.merge!` – Method.

```
merge!(repo::GitRepo; kwargs...) -> Bool
```

Perform a git merge on the repository `repo`, merging commits with diverging history into the current branch. Return `true` if the merge succeeded, `false` if not.

The keyword arguments are:

- `committish::AbstractString=""`: Merge the named commit(s) in `committish`.
- `branch::AbstractString=""`: Merge the branch `branch` and all its commits since it diverged from the current branch.
- `fastforward::Bool=false`: If `fastforward` is `true`, only merge if the merge is a fast-forward (the current branch head is an ancestor of the commits to be merged), otherwise refuse to merge and return `false`. This is equivalent to the git CLI option `--ff-only`.
- `merge_opts::MergeOptions=MergeOptions()`: `merge_opts` specifies options for the merge, such as merge strategy in case of conflicts.
- `checkout_opts::CheckoutOptions=CheckoutOptions()`: `checkout_opts` specifies options for the checkout step.

Equivalent to `git merge [--ff-only] [<committish> | <branch>]`.

Note

If you specify a branch, this must be done in reference format, since the string will be turned into a `GitReference`. For example, if you wanted to merge branch `branch_a`, you would call `merge!(repo, branch="refs/heads/branch_a")`.

`LibGit2.merge!` – Method.

```
merge!(repo::GitRepo, anns::Vector{GitAnnotated}; kwargs...) -> Bool
```

Merge changes from the annotated commits (captured as `GitAnnotated` objects) `anns` into the HEAD of the repository `repo`. The keyword arguments are:

- `merge_opts::MergeOptions = MergeOptions()`: options for how to perform the merge, including whether fastforwarding is allowed. See `MergeOptions` for more information.
- `checkout_opts::CheckoutOptions = CheckoutOptions()`: options for how to perform the checkout. See `CheckoutOptions` for more information.

`anns` may refer to remote or local branch heads. Return `true` if the merge is successful, otherwise return `false` (for instance, if no merge is possible because the branches have no common ancestor).

Examples

```
upst_ann = LibGit2.GitAnnotated(repo, "branch/a")

# merge the branch in
LibGit2.merge!(repo, [upst_ann])
```

`LibGit2.merge!` – Method.

```
merge!(repo::GitRepo, anns::Vector{GitAnnotated}, fastforward::Bool; kwargs...) -> Bool
```

Merge changes from the annotated commits (captured as `GitAnnotated` objects) `anns` into the HEAD of the repository `repo`. If `fastforward` is `true`, *only* a fastforward merge is allowed. In this case, if conflicts

occur, the merge will fail. Otherwise, if `fastforward` is `false`, the merge may produce a conflict file which the user will need to resolve.

The keyword arguments are:

- `merge_opts::MergeOptions = MergeOptions()`: options for how to perform the merge, including whether fastforwarding is allowed. See `MergeOptions` for more information.
- `checkout_opts::CheckoutOptions = CheckoutOptions()`: options for how to perform the checkout. See `CheckoutOptions` for more information.

anns may refer to remote or local branch heads. Return `true` if the merge is successful, otherwise return `false` (for instance, if no merge is possible because the branches have no common ancestor).

Examples

```
upst_ann_1 = LibGit2.GitAnnotated(repo, "branch/a")

# merge the branch in, fastforward
LibGit2.merge!(repo, [upst_ann_1], true)

# merge conflicts!
upst_ann_2 = LibGit2.GitAnnotated(repo, "branch/b")
# merge the branch in, try to fastforward
LibGit2.merge!(repo, [upst_ann_2], true) # will return false
LibGit2.merge!(repo, [upst_ann_2], false) # will return true
```

`LibGit2.ffmerge!` – Function.

```
ffmerge!(repo::GitRepo, ann::GitAnnotated)
```

Fastforward merge changes into current HEAD. This is only possible if the commit referred to by ann is descended from the current HEAD (e.g. if pulling changes from a remote branch which is simply ahead of the local branch tip).

`LibGit2.fullname` – Function.

```
LibGit2.fullname(ref::GitReference)
```

Return the name of the reference pointed to by the symbolic reference `ref`. If `ref` is not a symbolic reference, return an empty string.

`LibGit2.features` – Function.

```
features()
```

Return a list of git features the current version of libgit2 supports, such as threading or using HTTPS or SSH.

`LibGit2.filename` – Function.

```
filename(te::GitTreeEntry)
```

Return the filename of the object on disk to which `te` refers.

`LibGit2.filemode` – Function.

```
filemode(te::GitTreeEntry) -> Cint
```

Return the UNIX filemode of the object on disk to which `te` refers as an integer.

`LibGit2.gitdir` – Function.

```
LibGit2.gitdir(repo::GitRepo)
```

Return the location of the "git" files of `repo`:

- for normal repositories, this is the location of the `.git` folder.
- for bare repositories, this is the location of the repository itself.

See also `workdir`, `path`.

`LibGit2.git_url` – Function.

```
LibGit2.git_url(; kwargs...) -> String
```

Create a string based upon the URL components provided. When the `scheme` keyword is not provided the URL produced will use the alternative scp-like syntax.

Keywords

- `scheme::AbstractString=""`: the URL scheme which identifies the protocol to be used. For HTTP use "http", SSH use "ssh", etc. When `scheme` is not provided the output format will be "ssh" but using the scp-like syntax.
- `username::AbstractString=""`: the username to use in the output if provided.
- `password::AbstractString=""`: the password to use in the output if provided.
- `host::AbstractString=""`: the hostname to use in the output. A hostname is required to be specified.
- `port::Union{AbstractString,Integer}=""`: the port number to use in the output if provided. Cannot be specified when using the scp-like syntax.
- `path::AbstractString=""`: the path to use in the output if provided.

Warning

Avoid using passwords in URLs. Unlike the credential objects, Julia is not able to securely zero or destroy the sensitive data after use and the password may remain in memory; possibly to be exposed by an uninitialized memory.

Examples

```
julia> LibGit2.git_url(username="git", host="github.com", path="JuliaLang/julia.git")
"git@github.com:JuliaLang/julia.git"

julia> LibGit2.git_url(scheme="https", host="github.com", path="/JuliaLang/julia.git")
"https://github.com/JuliaLang/julia.git"

julia> LibGit2.git_url(scheme="ssh", username="git", host="github.com", port=2222,
↪  path="JuliaLang/julia.git")
"ssh://git@github.com:2222/JuliaLang/julia.git"
```

`LibGit2.@githash_str` – Macro.

```
@githash_str -> AbstractGitHash
```

Construct a git hash object from the given string, returning a `GitShortHash` if the string is shorter than 40 hexadecimal digits, otherwise a `GitHash`.

Examples

```
julia> LibGit2.githash"d114feb74ce633"
GitShortHash("d114feb74ce633")

julia> LibGit2.githash"d114feb74ce63307afe878a5228ad014e0289a85"
GitHash("d114feb74ce63307afe878a5228ad014e0289a85")
```

`LibGit2.head` – Function.

```
LibGit2.head(repo::GitRepo) -> GitReference
```

Return a `GitReference` to the current HEAD of `repo`.

```
head(pkg::AbstractString) -> String
```

Return current HEAD `GitHash` of the pkg repo as a string.

`LibGit2.head!` – Function.

```
LibGit2.head!(repo::GitRepo, ref::GitReference) -> GitReference
```

Set the HEAD of `repo` to the object pointed to by `ref`.

`LibGit2.head_oid` – Function.

```
LibGit2.head_oid(repo::GitRepo) -> GitHash
```

Lookup the object id of the current HEAD of git repository `repo`.

`LibGit2.headname` – Function.

```
LibGit2.headname(repo::GitRepo)
```

Lookup the name of the current HEAD of git repository `repo`. If `repo` is currently detached, return the name of the HEAD it's detached from.

`LibGit2.init` – Function.

```
LibGit2.init(path::AbstractString, bare::Bool=false) -> GitRepo
```

Open a new git repository at `path`. If `bare` is `false`, the working tree will be created in `path/.git`. If `bare` is `true`, no working directory will be created.

`LibGit2.is_ancestor_of` – Function.

```
is_ancestor_of(a::AbstractString, b::AbstractString, repo::GitRepo) -> Bool
```

Return `true` if `a`, a `GitHash` in string form, is an ancestor of `b`, a `GitHash` in string form.

Examples

```
julia> repo = GitRepo(repo_path);

julia> LibGit2.add!(repo, test_file1);

julia> commit_oid1 = LibGit2.commit(repo, "commit1");

julia> LibGit2.add!(repo, test_file2);

julia> commit_oid2 = LibGit2.commit(repo, "commit2");

julia> LibGit2.is_ancestor_of(string(commit_oid1), string(commit_oid2), repo)
true
```

`LibGit2.isbinary` – Function.

```
isbinary(blob::GitBlob) -> Bool
```

Use a heuristic to guess if a file is binary: searching for NULL bytes and looking for a reasonable ratio of printable to non-printable characters among the first 8000 bytes.

`LibGit2.iscommit` – Function.

```
iscommit(id::AbstractString, repo::GitRepo) -> Bool
```

Check if commit `id` (which is a `GitHash` in string form) is in the repository.

Examples

```
julia> repo = GitRepo(repo_path);

julia> LibGit2.add!(repo, test_file);

julia> commit_oid = LibGit2.commit(repo, "add test_file");

julia> LibGit2.iscommit(string(commit_oid), repo)
true
```

`LibGit2.isdiff` – Function.

```
LibGit2.isdiff(repo::GitRepo, treeish::AbstractString, pathspecs::AbstractString="";
↪  cached::Bool=false)
```

Checks if there are any differences between the tree specified by `treeish` and the tracked files in the working tree (if `cached=false`) or the index (if `cached=true`). `pathspecs` are the specifications for options for the diff.

Examples

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.isdiff(repo, "HEAD") # should be false
open(joinpath(repo_path, new_file), "a") do f
    println(f, "here's my cool new file")
end
LibGit2.isdiff(repo, "HEAD") # now true
```

Equivalent to `git diff-index <treeish> [-- <pathspecs>]`.

`LibGit2.isdirty` – Function.

```
LibGit2.isdirty(repo::GitRepo, pathspecs::AbstractString=""; cached::Bool=false) -> Bool
```

Check if there have been any changes to tracked files in the working tree (if `cached=false`) or the index (if `cached=true`). `pathspecs` are the specifications for options for the diff.

Examples

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.isdirty(repo) # should be false
open(joinpath(repo_path, new_file), "a") do f
    println(f, "here's my cool new file")
end
LibGit2.isdirty(repo) # now true
LibGit2.isdirty(repo, new_file) # now true
```

Equivalent to `git diff-index HEAD [-- <pathspecs>]`.

`LibGit2.isorphan` – Function.

```
LibGit2.isorphan(repo::GitRepo)
```

Check if the current branch is an "orphan" branch, i.e. has no commits. The first commit to this branch will have no parents.

`LibGit2.isset` – Function.

```
isset(val::Integer, flag::Integer)
```

Test whether the bits of `val` indexed by `flag` are set (`1`) or unset (`0`).

`LibGit2.iszero` – Function.

```
iszero(id::GitHash) -> Bool
```

Determine whether all hexadecimal digits of the given `GitHash` are zero.

`LibGit2.lookup_branch` – Function.

```
lookup_branch(repo::GitRepo, branch_name::AbstractString, remote::Bool=false) ->
↪  Union{GitReference, Nothing}
```

Determine if the branch specified by `branch_name` exists in the repository `repo`. If `remote` is `true`, `repo` is assumed to be a remote git repository. Otherwise, it is part of the local filesystem.

Return either a `GitReference` to the requested branch if it exists, or `nothing` if not.

`LibGit2.map` – Function.

```
LibGit2.map(f::Function, walker::GitRevWalker; oid::GitHash=GitHash(), range::AbstractString="",
↪  by::Cint=Consts.SORT_NONE, rev::Bool=false)
```

Using the GitRevWalker walker to "walk" over every commit in the repository's history, apply f to each commit in the walk. The keyword arguments are: * oid: The GitHash of the commit to begin the walk from. The default is to use push_head! and therefore the HEAD commit and all its ancestors. * range: A range of GitHashs in the format oid1..oid2. f will be applied to all commits between the two. * by: The sorting method. The default is not to sort. Other options are to sort by topology (LibGit2.Consts.SORT_TOPOLOGICAL), to sort forwards in time (LibGit2.Consts.SORT_TIME, most ancient first) or to sort backwards in time (LibGit2.Consts.SORT_REVERSE, most recent first). * rev: Whether to reverse the sorted order (for instance, if topological sorting is used).

Examples

```
oids = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    LibGit2.map((oid, repo)->string(oid), walker, by=LibGit2.Consts.SORT_TIME)
end
```

Here, LibGit2.map visits each commit using the GitRevWalker and finds its GitHash.

LibGit2.mirror_callback – Function.

Mirror callback function

Function sets +refs/*:refs/* refspecs and mirror flag for remote reference.

LibGit2.mirror_cb – Function.

C function pointer for mirror_callback

LibGit2.message – Function.

```
message(c::GitCommit, raw::Bool=false)
```

Return the commit message describing the changes made in commit c. If raw is false, return a slightly "cleaned up" message (which has any leading newlines removed). If raw is true, the message is not stripped of any such newlines.

LibGit2.merge_analysis – Function.

```
merge_analysis(repo::GitRepo, anns::Vector{GitAnnotated}) -> analysis, preference
```

Run analysis on the branches pointed to by the annotated branch tips anns and determine under what circumstances they can be merged. For instance, if anns[1] is simply an ancestor of ann[2], then merge_analysis will report that a fast-forward merge is possible.

Return two outputs, analysis and preference. analysis has several possible values: * MERGE_ANALYSIS_NONE: it is not possible to merge the elements of anns. * MERGE_ANALYSIS_NORMAL: a regular merge, when HEAD and the commits that the user wishes to merge have all diverged from a common ancestor. In this case the changes have to be resolved and conflicts may occur. * MERGE_ANALYSIS_UP_TO_DATE: all the input commits the user wishes to merge can be reached from HEAD, so no merge needs to be performed. * MERGE_ANALYSIS_FASTFORWARD: the input commit is a descendant of HEAD and so no merge needs to be performed - instead, the user can simply checkout the input commit(s). * MERGE_ANALYSIS_UNBORN: the HEAD of the repository refers to a commit which does not exist. It is not possible to merge, but it may be

possible to checkout the input commits. `preference` also has several possible values: * `MERGE_PREFERENCE_NONE`: the user has no preference. * `MERGE_PREFERENCE_NO_FASTFORWARD`: do not allow any fast-forward merges. * `MERGE_PREFERENCE_FASTFORWARD_ONLY`: allow only fast-forward merges and no other type (which may introduce conflicts). `preference` can be controlled through the repository or global git configuration.

`LibGit2.name` – Function.

```
LibGit2.name(ref::GitReference)
```

Return the full name of `ref`.

```
name(rmt::GitRemote)
```

Get the name of a remote repository, for instance `"origin"`. If the remote is anonymous (see `GitRemoteAnon`) the name will be an empty string `""`.

Examples

```
julia> repo_url = "https://github.com/JuliaLang/Example.jl";

julia> repo = LibGit2.clone(cache_repo, "test_directory");

julia> remote = LibGit2.GitRemote(repo, "origin", repo_url);

julia> name(remote)
"origin"
```

```
LibGit2.name(tag::GitTag)
```

The name of `tag` (e.g. `"v0.5"`).

`LibGit2.need_update` – Function.

```
need_update(repo::GitRepo)
```

Equivalent to `git update-index`. Return `true` if `repo` needs updating.

`LibGit2.objtype` – Function.

```
objtype(obj_type::Consts.OBJECT)
```

Return the type corresponding to the enum value.

`LibGit2.path` – Function.

```
LibGit2.path(repo::GitRepo)
```

Return the base file path of the repository `repo`.

- for normal repositories, this will typically be the parent directory of the ".git" directory (note: this may be different than the working directory, see `workdir` for more details).
- for bare repositories, this is the location of the "git" files.

See also `gitdir`, `workdir`.

`LibGit2.peel` – Function.

```
peel([T,] ref::GitReference)
```

Recursively peel `ref` until an object of type T is obtained. If no T is provided, then `ref` will be peeled until an object other than a `GitTag` is obtained.

- A `GitTag` will be peeled to the object it references.
- A `GitCommit` will be peeled to a `GitTree`.

Note

Only annotated tags can be peeled to `GitTag` objects. Lightweight tags (the default) are references under `refs/tags/` which point directly to `GitCommit` objects.

```
peel([T,] obj::GitObject)
```

Recursively peel `obj` until an object of type T is obtained. If no T is provided, then `obj` will be peeled until the type changes.

- A `GitTag` will be peeled to the object it references.
- A `GitCommit` will be peeled to a `GitTree`.

`LibGit2.posixpath` – Function.

```
LibGit2.posixpath(path)
```

Standardise the path string `path` to use POSIX separators.

`LibGit2.push` – Function.

```
push(rmt::GitRemote, refspecs; force::Bool=false, options::PushOptions=PushOptions())
```

Push to the specified `rmt` remote git repository, using `refspecs` to determine which remote branch(es) to push to. The keyword arguments are:

- `force`: if `true`, a force-push will occur, disregarding conflicts.
- `options`: determines the options for the push, e.g. which proxy headers to use. See `PushOptions` for more information.

> **Note**
>
> You can add information about the push refspecs in two other ways: by setting an option in the repository's `GitConfig` (with `push.default` as the key) or by calling `add_push!`. Otherwise you will need to explicitly specify a push refspec in the call to push for it to have any effect, like so: `LibGit2.push(repo, refspecs=["refs/heads/master"])`.

```
push(repo::GitRepo; kwargs...)
```

Pushes updates to an upstream of `repo`.

The keyword arguments are:

- `remote::AbstractString="origin"`: the name of the upstream remote to push to.
- `remoteurl::AbstractString=""`: the URL of `remote`.
- `refspecs=AbstractString[]`: determines properties of the push.
- `force::Bool=false`: determines if the push will be a force push, overwriting the remote branch.
- `credentials=nothing`: provides credentials and/or settings when authenticating against a private `remote`.
- `callbacks=Callbacks()`: user provided callbacks and payloads.

Equivalent to `git push [<remoteurl>|<repo>] [<refspecs>]`.

`LibGit2.push!` – Method.

```
LibGit2.push!(w::GitRevWalker, cid::GitHash)
```

Start the `GitRevWalker` `walker` at commit `cid`. This function can be used to apply a function to all commits since a certain year, by passing the first commit of that year as `cid` and then passing the resulting `w` to `LibGit2.map`.

`LibGit2.push_head!` – Function.

```
LibGit2.push_head!(w::GitRevWalker)
```

Push the HEAD commit and its ancestors onto the `GitRevWalker` `w`. This ensures that HEAD and all its ancestor commits will be encountered during the walk.

`LibGit2.push_refspecs` – Function.

```
push_refspecs(rmt::GitRemote) -> Vector{String}
```

Get the *push* refspecs for the specified `rmt`. These refspecs contain information about which branch(es) to push to.

Examples

```
julia> remote = LibGit2.get(LibGit2.GitRemote, repo, "upstream");

julia> LibGit2.add_push!(repo, remote, "refs/heads/master");

julia> close(remote);

julia> remote = LibGit2.get(LibGit2.GitRemote, repo, "upstream");

julia> LibGit2.push_refspecs(remote)
String["refs/heads/master"]
```

`LibGit2.raw` – Function.

```
raw(id::GitHash) -> Vector{UInt8}
```

Obtain the raw bytes of the `GitHash` as a vector of length 20.

`LibGit2.read_tree!` – Function.

```
LibGit2.read_tree!(idx::GitIndex, tree::GitTree)
LibGit2.read_tree!(idx::GitIndex, treehash::AbstractGitHash)
```

Read the tree `tree` (or the tree pointed to by `treehash` in the repository owned by `idx`) into the index `idx`. The current index contents will be replaced.

`LibGit2.rebase!` – Function.

```
LibGit2.rebase!(repo::GitRepo, upstream::AbstractString="", newbase::AbstractString="")
```

Attempt an automatic merge rebase of the current branch, from `upstream` if provided, or otherwise from the upstream tracking branch. `newbase` is the branch to rebase onto. By default this is `upstream`.

If any conflicts arise which cannot be automatically resolved, the rebase will abort, leaving the repository and working tree in its original state, and the function will throw a `GitError`. This is roughly equivalent to the following command line statement:

```
git rebase --merge [<upstream>]
if [ -d ".git/rebase-merge" ]; then
    git rebase --abort
fi
```

`LibGit2.ref_list` – Function.

```
LibGit2.ref_list(repo::GitRepo) -> Vector{String}
```

Get a list of all reference names in the `repo` repository.

`LibGit2.reftype` – Function.

```
LibGit2.reftype(ref::GitReference) -> Cint
```

Return a `Cint` corresponding to the type of `ref`:

- `0` if the reference is invalid
- `1` if the reference is an object id
- `2` if the reference is symbolic

`LibGit2.remotes` – Function.

```
LibGit2.remotes(repo::GitRepo)
```

Return a vector of the names of the remotes of `repo`.

`LibGit2.remove!` – Function.

```
remove!(repo::GitRepo, files::AbstractString...)
remove!(idx::GitIndex, files::AbstractString...)
```

Remove all the files with paths specified by `files` in the index `idx` (or the index of the `repo`).

`LibGit2.reset` – Function.

```
reset(val::Integer, flag::Integer)
```

Unset the bits of `val` indexed by `flag`, returning them to `0`.

`LibGit2.reset!` – Function.

```
reset!(payload, [config]) -> CredentialPayload
```

Reset the `payload` state back to the initial values so that it can be used again within the credential callback. If a `config` is provided the configuration will also be updated.

Updates some entries, determined by the `pathspecs`, in the index from the target commit tree.

Sets the current head to the specified commit oid and optionally resets the index and working tree to match.

git reset [<committish>] [–] <pathspecs>...

```
reset!(repo::GitRepo, id::GitHash, mode::Cint=Consts.RESET_MIXED)
```

Reset the repository `repo` to its state at `id`, using one of three modes set by `mode`:

1. `Consts.RESET_SOFT` - move HEAD to `id`.
2. `Consts.RESET_MIXED` - default, move HEAD to `id` and reset the index to `id`.
3. `Consts.RESET_HARD` - move HEAD to `id`, reset the index to `id`, and discard all working changes.

Examples

```
# fetch changes
LibGit2.fetch(repo)
isfile(joinpath(repo_path, our_file)) # will be false

# fastforward merge the changes
LibGit2.merge!(repo, fastforward=true)

# because there was not any file locally, but there is
# a file remotely, we need to reset the branch
head_oid = LibGit2.head_oid(repo)
new_head = LibGit2.reset!(repo, head_oid, LibGit2.Consts.RESET_HARD)
```

In this example, the remote which is being fetched from *does* have a file called `our_file` in its index, which is why we must reset.

Equivalent to `git reset [--soft | --mixed | --hard] <id>`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
head_oid = LibGit2.head_oid(repo)
open(joinpath(repo_path, "file1"), "w") do f
    write(f, "111
")
end
LibGit2.add!(repo, "file1")
mode = LibGit2.Consts.RESET_HARD
# will discard the changes to file1
# and unstage it
new_head = LibGit2.reset!(repo, head_oid, mode)
```

`LibGit2.restore` – Function.

```
restore(s::State, repo::GitRepo)
```

Return a repository `repo` to a previous `State` `s`, for example the HEAD of a branch before a merge attempt. `s` can be generated using the `snapshot` function.

`LibGit2.revcount` – Function.

```
LibGit2.revcount(repo::GitRepo, commit1::AbstractString, commit2::AbstractString)
```

List the number of revisions between `commit1` and `commit2` (committish OIDs in string form). Since `commit1` and `commit2` may be on different branches, `revcount` performs a "left-right" revision list (and count), returning a tuple of `Int`s - the number of left and right commits, respectively. A left (or right) commit refers to which side of a symmetric difference in a tree the commit is reachable from.

Equivalent to `git rev-list --left-right --count <commit1> <commit2>`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
repo_file = open(joinpath(repo_path, test_file), "a")
println(repo_file, "hello world")
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid1 = LibGit2.commit(repo, "commit 1")
println(repo_file, "hello world again")
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid2 = LibGit2.commit(repo, "commit 2")
LibGit2.revcount(repo, string(commit_oid1), string(commit_oid2))
```

This will return `(-1, 0)`.

`LibGit2.set_remote_url` – Function.

```
set_remote_url(repo::GitRepo, remote_name, url)
set_remote_url(repo::String, remote_name, url)
```

Set both the fetch and push `url` for `remote_name` for the `GitRepo` or the git repository located at `path`. Typically git repos use "`origin`" as the remote name.

Examples

```
repo_path = joinpath(tempdir(), "Example")
repo = LibGit2.init(repo_path)
LibGit2.set_remote_url(repo, "upstream", "https://github.com/JuliaLang/Example.jl")
LibGit2.set_remote_url(repo_path, "upstream2", "https://github.com/JuliaLang/Example2.jl")
```

`LibGit2.shortname` – Function.

```
LibGit2.shortname(ref::GitReference)
```

Return a shortened version of the name of `ref` that's "human-readable".

```
julia> repo = GitRepo(path_to_repo);

julia> branch_ref = LibGit2.head(repo);

julia> LibGit2.name(branch_ref)
"refs/heads/master"

julia> LibGit2.shortname(branch_ref)
"master"
```

`LibGit2.snapshot` – Function.

```
snapshot(repo::GitRepo) -> State
```

Take a snapshot of the current state of the repository `repo`, storing the current HEAD, index, and any uncommitted work. The output `State` can be used later during a call to `restore` to return the repository to the snapshotted state.

`LibGit2.split_cfg_entry` – Function.

```
LibGit2.split_cfg_entry(ce::LibGit2.ConfigEntry) -> Tuple{String,String,String,String}
```

Break the `ConfigEntry` up to the following pieces: section, subsection, name, and value.

Examples

Given the git configuration file containing:

```
[credential "https://example.com"]
    username = me
```

The `ConfigEntry` would look like the following:

```
julia> entry
ConfigEntry("credential.https://example.com.username", "me")

julia> LibGit2.split_cfg_entry(entry)
("credential", "https://example.com", "username", "me")
```

Refer to the git config syntax documentation for more details.

`LibGit2.status` – Function.

```
LibGit2.status(repo::GitRepo, path::String) -> Union{Cuint, Cvoid}
```

Lookup the status of the file at `path` in the git repository `repo`. For instance, this can be used to check if the file at `path` has been modified and needs to be staged and committed.

`LibGit2.stage` – Function.

```
stage(ie::IndexEntry) -> Cint
```

Get the stage number of `ie`. The stage number `0` represents the current state of the working tree, but other numbers can be used in the case of a merge conflict. In such a case, the various stage numbers on an `IndexEntry` describe which side(s) of the conflict the current state of the file belongs to. Stage `0` is the state before the attempted merge, stage 1 is the changes which have been made locally, stages 2 and larger are for changes from other branches (for instance, in the case of a multi-branch "octopus" merge, stages 2, 3, and 4 might be used).

`LibGit2.tag_create` – Function.

```
LibGit2.tag_create(repo::GitRepo, tag::AbstractString, commit; kwargs...)
```

Create a new git tag `tag` (e.g. `"v0.5"`) in the repository `repo`, at the commit `commit`.

The keyword arguments are:

- `msg::AbstractString=""`: the message for the tag.
- `force::Bool=false`: if `true`, existing references will be overwritten.
- `sig::Signature=Signature(repo)`: the tagger's signature.

`LibGit2.tag_delete` – Function.

```
LibGit2.tag_delete(repo::GitRepo, tag::AbstractString)
```

Remove the git tag `tag` from the repository `repo`.

`LibGit2.tag_list` – Function.

```
LibGit2.tag_list(repo::GitRepo) -> Vector{String}
```

Get a list of all tags in the git repository `repo`.

`LibGit2.target` – Function.

```
LibGit2.target(tag::GitTag)
```

The `GitHash` of the target object of `tag`.

`LibGit2.toggle` – Function.

```
toggle(val::Integer, flag::Integer)
```

Flip the bits of `val` indexed by `flag`, so that if a bit is `0` it will be `1` after the toggle, and vice-versa.

`LibGit2.transact` – Function.

```
transact(f::Function, repo::GitRepo)
```

Apply function `f` to the git repository `repo`, taking a `snapshot` before applying `f`. If an error occurs within `f`, `repo` will be returned to its snapshot state using `restore`. The error which occurred will be rethrown, but the state of `repo` will not be corrupted.

`LibGit2.treewalk` – Function.

```
treewalk(f, tree::GitTree, post::Bool=false)
```

Traverse the entries in `tree` and its subtrees in post or pre order. Preorder means beginning at the root and then traversing the leftmost subtree (and recursively on down through that subtree's leftmost subtrees) and moving right through the subtrees. Postorder means beginning at the bottom of the leftmost subtree, traversing upwards through it, then traversing the next right subtree (again beginning at the bottom) and finally visiting the tree root last of all.

The function parameter `f` should have following signature:

```
(String, GitTreeEntry) -> Cint
```

A negative value returned from `f` stops the tree walk. A positive value means that the entry will be skipped if `post` is `false`.

`LibGit2.upstream` – Function.

```
upstream(ref::GitReference) -> Union{GitReference, Nothing}
```

Determine if the branch containing `ref` has a specified upstream branch.

Return either a `GitReference` to the upstream branch if it exists, or `nothing` if the requested branch does not have an upstream counterpart.

`LibGit2.update!` – Function.

```
update!(repo::GitRepo, files::AbstractString...)
update!(idx::GitIndex, files::AbstractString...)
```

Update all the files with paths specified by `files` in the index `idx` (or the index of the `repo`). Match the state of each file in the index with the current state on disk, removing it if it has been removed on disk, or updating its entry in the object database.

`LibGit2.url` – Function.

```
url(rmt::GitRemote)
```

Get the fetch URL of a remote git repository.

Examples

```
julia> repo_url = "https://github.com/JuliaLang/Example.jl";

julia> repo = LibGit2.init(mktempdir());

julia> remote = LibGit2.GitRemote(repo, "origin", repo_url);

julia> LibGit2.url(remote)
"https://github.com/JuliaLang/Example.jl"
```

`LibGit2.version` – Function.

```
version() -> VersionNumber
```

Return the version of libgit2 in use, as a `VersionNumber`.

`LibGit2.with` – Function.

```
with(f::Function, obj)
```

Resource management helper function. Applies `f` to `obj`, making sure to call `close` on `obj` after `f` successfully returns or throws an error. Ensures that allocated git resources are finalized as soon as they are no longer needed.

`LibGit2.with_warn` – Function.

```
with_warn(f::Function, ::Type{T}, args...)
```

Resource management helper function. Apply `f` to `args`, first constructing an instance of type T from `args`. Makes sure to call `close` on the resulting object after `f` successfully returns or throws an error. Ensures that allocated git resources are finalized as soon as they are no longer needed. If an error is thrown by `f`, a warning is shown containing the error.

`LibGit2.workdir` – Function.

```
LibGit2.workdir(repo::GitRepo)
```

Return the location of the working directory of `repo`. This will throw an error for bare repositories.

> **Note**
>
> This will typically be the parent directory of `gitdir(repo)`, but can be different in some cases: e.g. if either the `core.worktree` configuration variable or the `GIT_WORK_TREE` environment variable is set.

See also `gitdir`, `path`.

`LibGit2.GitObject` – Method.

```
(::Type{T})(te::GitTreeEntry) where T<:GitObject
```

Get the git object to which `te` refers and return it as its actual type (the type `entrytype` would show), for instance a `GitBlob` or `GitTag`.

Examples

```
tree = LibGit2.GitTree(repo, "HEAD^{tree}")
tree_entry = tree[1]
blob = LibGit2.GitBlob(tree_entry)
```

`LibGit2.UserPasswordCredential` – Type.

Credential that support only `user` and `password` parameters

`LibGit2.SSHCredential` – Type.

SSH credential type

`LibGit2.isfilled` – Function.

```
isfilled(cred::AbstractCredential) -> Bool
```

Verifies that a credential is ready for use in authentication.

`LibGit2.CachedCredentials` – Type.

Caches credential information for re-use

`LibGit2.CredentialPayload` – Type.

```
LibGit2.CredentialPayload
```

Retains the state between multiple calls to the credential callback for the same URL. A `CredentialPayload` instance is expected to be `reset!` whenever it will be used with a different URL.

`LibGit2.approve` – Function.

```
approve(payload::CredentialPayload; shred::Bool=true) -> Nothing
```

Store the `payload` credential for re-use in a future authentication. Should only be called when authentication was successful.

The `shred` keyword controls whether sensitive information in the payload credential field should be destroyed. Should only be set to `false` during testing.

`LibGit2.reject` – Function.

```
reject(payload::CredentialPayload; shred::Bool=true) -> Nothing
```

Discard the `payload` credential from begin re-used in future authentication. Should only be called when authentication was unsuccessful.

The `shred` keyword controls whether sensitive information in the payload credential field should be destroyed. Should only be set to `false` during testing.

`LibGit2.Consts.GIT_CONFIG` – Type.

Priority level of a config file.

These priority levels correspond to the natural escalation logic (from higher to lower) when searching for config entries in git.

- `CONFIG_LEVEL_DEFAULT` - Open the global, XDG and system configuration files if any available.
- `CONFIG_LEVEL_PROGRAMDATA` - System-wide on Windows, for compatibility with portable git
- `CONFIG_LEVEL_SYSTEM` - System-wide configuration file; `/etc/gitconfig` on Linux systems
- `CONFIG_LEVEL_XDG` - XDG compatible configuration file; typically `~/.config/git/config`
- `CONFIG_LEVEL_GLOBAL` - User-specific configuration file (also called Global configuration file); typically `~/.gitconfig`
- `CONFIG_LEVEL_LOCAL` - Repository specific configuration file; `$WORK_DIR/.git/config` on non-bare repos
- `CONFIG_LEVEL_APP` - Application specific configuration file; freely defined by applications
- `CONFIG_HIGHEST_LEVEL` - Represents the highest level available config file (i.e. the most specific config file available that actually is loaded)

Chapter 78

动态链接器

`Base.Libc.Libdl.dlopen` – Function.

```
dlopen(libfile::AbstractString [, flags::Integer]; throw_error:Bool = true)
```

Load a shared library, returning an opaque handle.

The extension given by the constant `dlext` (`.so`, `.dll`, or `.dylib`) can be omitted from the `libfile` string, as it is automatically appended if needed. If `libfile` is not an absolute path name, then the paths in the array `DL_LOAD_PATH` are searched for `libfile`, followed by the system load path.

The optional flags argument is a bitwise-or of zero or more of `RTLD_LOCAL`, `RTLD_GLOBAL`, `RTLD_LAZY`, `RTLD_NOW`, `RTLD_NODELETE`, `RTLD_NOLOAD`, `RTLD_DEEPBIND`, and `RTLD_FIRST`. These are converted to the corresponding flags of the POSIX (and/or GNU libc and/or MacOS) dlopen command, if possible, or are ignored if the specified functionality is not available on the current platform. The default flags are platform specific. On MacOS the default `dlopen` flags are `RTLD_LAZY|RTLD_DEEPBIND|RTLD_GLOBAL` while on other platforms the defaults are `RTLD_LAZY|RTLD_DEEPBIND|RTLD_LOCAL`. An important usage of these flags is to specify non default behavior for when the dynamic library loader binds library references to exported symbols and if the bound references are put into process local or global scope. For instance `RTLD_LAZY|RTLD_DEEPBIND|RTLD_GLOBAL` allows the library's symbols to be available for usage in other shared libraries, addressing situations where there are dependencies between shared libraries.

If the library cannot be found, this method throws an error, unless the keyword argument `throw_error` is set to `false`, in which case this method returns `nothing`.

> **Note**
>
> From Julia 1.6 on, this method replaces paths starting with `@executable_path/` with the path to the Julia executable, allowing for relocatable relative-path loads. In Julia 1.5 and earlier, this only worked on macOS.

source

`Base.Libc.Libdl.dlopen_e` – Function.

```
dlopen_e(libfile::AbstractString [, flags::Integer])
```

Similar to dlopen, except returns C_NULL instead of raising errors. This method is now deprecated in favor of dlopen(libfile::AbstractString [, flags::Integer]; throw_error=false).

source

Base.Libc.Libdl.RTLD_NOW – Constant.

```
RTLD_DEEPBIND
RTLD_FIRST
RTLD_GLOBAL
RTLD_LAZY
RTLD_LOCAL
RTLD_NODELETE
RTLD_NOLOAD
RTLD_NOW
```

Enum constant for dlopen. See your platform man page for details, if applicable.

source

Base.Libc.Libdl.dlsym – Function.

```
dlsym(handle, sym; throw_error::Bool = true)
```

Look up a symbol from a shared library handle, return callable function pointer on success.

If the symbol cannot be found, this method throws an error, unless the keyword argument throw_error is set to false, in which case this method returns nothing.

source

Base.Libc.Libdl.dlsym_e – Function.

```
dlsym_e(handle, sym)
```

Look up a symbol from a shared library handle, silently return C_NULL on lookup failure. This method is now deprecated in favor of dlsym(handle, sym; throw_error=false).

source

Base.Libc.Libdl.dlclose – Function.

```
dlclose(handle)
```

Close shared library referenced by handle.

source

```
dlclose(::Nothing)
```

For the very common pattern usage pattern of

```
try
    hdl = dlopen(library_name)
    ... do something
finally
    dlclose(hdl)
end
```

We define a `dlclose()` method that accepts a parameter of type `Nothing`, so that user code does not have to change its behavior for the case that `library_name` was not found.

source

`Base.Libc.Libdl.dlext` – Constant.

```
dlext
```

File extension for dynamic libraries (e.g. dll, dylib, so) on the current platform.

source

`Base.Libc.Libdl.dllist` – Function.

```
dllist()
```

Return the paths of dynamic libraries currently loaded in a `Vector{String}`.

source

`Base.Libc.Libdl.dlpath` – Function.

```
dlpath(handle::Ptr{Cvoid})
```

Given a library `handle` from `dlopen`, return the full path.

source

```
dlpath(libname::Union{AbstractString, Symbol})
```

Get the full path of the library `libname`.

Example

```
julia> dlpath("libjulia")
```

source

`Base.Libc.Libdl.find_library` – Function.

```
find_library(names [, locations])
```

Searches for the first library in names in the paths in the `locations` list, `DL_LOAD_PATH`, or system library paths (in that order) which can successfully be dlopen'd. On success, the return value will be one of the names (potentially prefixed by one of the paths in locations). This string can be assigned to a `global const` and used as the library name in future `ccall`'s. On failure, it returns the empty string.

source

`Base.DL_LOAD_PATH` – Constant.

```
DL_LOAD_PATH
```

When calling `dlopen`, the paths in this list will be searched first, in order, before searching the system locations for a valid library handle.

source

Chapter 79

Linear Algebra

除了（且作为一部分）对多维数组的支持，Julia 还提供了许多常见和实用的线性代数运算的本地实现，可通过 using LinearAlgebra 加载。基本的运算，比如 tr，det 和 inv 都是支持的：

```
julia> A = [1 2 3; 4 1 6; 7 8 1]
3×3 Matrix{Int64}:
 1  2  3
 4  1  6
 7  8  1

julia> tr(A)
3

julia> det(A)
104.0

julia> inv(A)
3×3 Matrix{Float64}:
 -0.451923   0.211538    0.0865385
  0.365385  -0.192308    0.0576923
  0.240385   0.0576923  -0.0673077
```

还有其它实用的运算，比如寻找特征值或特征向量：

```
julia> A = [-4. -17.; 2. 2.]
2×2 Matrix{Float64}:
 -4.0  -17.0
  2.0    2.0

julia> eigvals(A)
2-element Vector{ComplexF64}:
 -1.0 - 5.0im
 -1.0 + 5.0im

julia> eigvecs(A)
2×2 Matrix{ComplexF64}:
  0.945905-0.0im        0.945905+0.0im
 -0.166924+0.278207im  -0.166924-0.278207im
```

此外，Julia 提供了多种矩阵分解，通过将矩阵预先分解成更适合问题的形式（出于性能或内存上的原因），它们可用于加快问题的求解，如线性求解或矩阵求幂。更多有关信息请参阅文档 factorize。举个例子：

```
julia> A = [1.5 2 -4; 3 -1 -6; -10 2.3 4]
3×3 Matrix{Float64}:
   1.5   2.0  -4.0
   3.0  -1.0  -6.0
 -10.0   2.3   4.0

julia> factorize(A)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3×3 Matrix{Float64}:
  1.0    0.0       0.0
 -0.15   1.0       0.0
 -0.3   -0.132196  1.0
U factor:
3×3 Matrix{Float64}:
 -10.0  2.3     4.0
   0.0  2.345  -3.4
   0.0  0.0    -5.24947
```

因为 A 不是埃尔米特、对称、三角、三对角或双对角矩阵，LU 分解也许是我们能做的最好分解。与之相比：

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Matrix{Float64}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> factorize(B)
BunchKaufman{Float64, Matrix{Float64}, Vector{Int64}}
D factor:
3×3 Tridiagonal{Float64, Vector{Float64}}:
 -1.64286   0.0   ⋅
  0.0      -2.8  0.0
   ⋅        0.0  5.0
U factor:
3×3 UnitUpperTriangular{Float64, Matrix{Float64}}:
 1.0  0.142857  -0.8
  ⋅   1.0       -0.6
  ⋅    ⋅         1.0
permutation:
3-element Vector{Int64}:
 1
 2
 3
```

在这里，Julia 能够发现 B 确实是对称矩阵，并且使用一种更适当的分解。针对一个具有某些属性的矩阵，比如一个对称或三对角矩阵，往往有可能写出更高效的代码。Julia 提供了一些特殊的类型好让你可以根据矩阵所具有的属性「标记」它们。例如：

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Matrix{Float64}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> sB = Symmetric(B)
3×3 Symmetric{Float64, Matrix{Float64}}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0
```

sB 已经被标记成（实）对称矩阵，所以对于之后可能在它上面执行的操作，例如特征因子化或矩阵-向量乘积，只引用矩阵的一半可以提高效率。举个例子：

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Matrix{Float64}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> sB = Symmetric(B)
3×3 Symmetric{Float64, Matrix{Float64}}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> x = [1; 2; 3]
3-element Vector{Int64}:
 1
 2
 3

julia> sB\x
3-element Vector{Float64}:
 -1.7391304347826084
 -1.1086956521739126
 -1.4565217391304346
```

\ 运算在这里执行线性求解。左除运算符相当强大，很容易写出紧凑、可读的代码，它足够灵活，可以求解各种线性方程组。

79.1 特殊矩阵

具有特殊对称性和结构的矩阵经常在线性代数中出现并且与各种矩阵分解相关。Julia 具有丰富的特殊矩阵类型，可以快速计算专门为特定矩阵类型开发的专用例程。

下表总结了在 Julia 中已经实现的特殊矩阵类型，以及为它们提供各种优化方法的钩子在 LAPACK 中是否可用。

基本运算

Legend:

类型	描述
Symmetric	Symmetric matrix
Hermitian	Hermitian matrix
UpperTriangular	上三角矩阵
UnitUpperTriangular	单位上三角矩阵 with unit diagonal
LowerTriangular	下三角矩阵
UnitLowerTriangular	单位下三角矩阵
UpperHessenberg	Upper Hessenberg matrix
Tridiagonal	Tridiagonal matrix
SymTridiagonal	对称三对角矩阵
Bidiagonal	上/下双对角矩阵
Diagonal	Diagonal matrix
UniformScaling	Uniform scaling operator

矩阵类型	+	-	*	\	具有优化方法的其它函数
Symmetric				MV	inv, sqrt, exp
Hermitian				MV	inv, sqrt, exp
UpperTriangular			MV	MV	inv, det, logdet
UnitUpperTriangular			MV	MV	inv, det, logdet
LowerTriangular			MV	MV	inv, det, logdet
UnitLowerTriangular			MV	MV	inv, det, logdet
UpperHessenberg				MM	inv, det
SymTridiagonal	M	M	MS	MV	eigmax, eigmin
Tridiagonal	M	M	MS	MV	
Bidiagonal	M	M	MS	MV	
Diagonal	M	M	MV	MV	inv, det, logdet, /
UniformScaling	M	M	MVS	MVS	/

Key	说明
M（矩阵）	针对矩阵与矩阵运算的优化方法可用
V（向量）	针对矩阵与向量运算的优化方法可用
S（标量）	针对矩阵与标量运算的优化方法可用

矩阵分解

图例：

均匀缩放运算符

UniformScaling 运算符代表一个标量乘以单位运算符，λ*I。单位运算符 I 被定义为常量，是 UniformScaling 的实例。这些运算符的大小是通用的，并且会在二元运算符 +，-，* 和 \ 中与另一个矩阵相匹配。对于 A+I 和 A-I ，这意味着 A 必须是个方阵。与单位运算符 I 相乘是一个空操作（除了检查比例因子是一），因此几乎没有开销。

来查看 UniformScaling 运算符的运行结果：

```
julia> U = UniformScaling(2);

julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
```

矩阵类型	LAPACK	eigen	eigvals	eigvecs	svd	svdvals
Symmetric	SY		ARI			
Hermitian	HE		ARI			
UpperTriangular	TR	A	A	A		
UnitUpperTriangular	TR	A	A	A		
LowerTriangular	TR	A	A	A		
UnitLowerTriangular	TR	A	A	A		
SymTridiagonal	ST	A	ARI	AV		
Tridiagonal	GT					
Bidiagonal	BD				A	A
Diagonal	DI		A			

键名	说明	例子
A (all)	找到所有特征值和/或特征向量的优化方法可用	e.g. `eigvals(M)`
R (range)	通过第 `ih` 个特征值寻找第 `il` 个特征值的优化方法可用	`eigvals(M, il, ih)`
I (interval)	寻找在区间 `[vl, vh]` 内的特征值的优化方法可用	`eigvals(M, vl, vh)`
V (vectors)	寻找对应于特征值 `x=[x1, x2,...]` 的特征向量的优化方法可用	`eigvecs(M, x)`

```
 1  2
 3  4

julia> a + U
2×2 Matrix{Int64}:
 3  2
 3  6

julia> a * U
2×2 Matrix{Int64}:
 2  4
 6  8

julia> [a U]
2×4 Matrix{Int64}:
 1  2  2  0
 3  4  0  2

julia> b = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> b - U
ERROR: DimensionMismatch: matrix is not square: dimensions are (2, 3)
Stacktrace:
[...]
```

If you need to solve many systems of the form `(A+μI)x = b` for the same A and different μ, it might be beneficial

to first compute the Hessenberg factorization F of A via the hessenberg function. Given F, Julia employs an efficient algorithm for `(F+μ*I) \ b` (equivalent to `(A+μ*I)x \ b`) and related operations like determinants.

79.2 Matrix factorizations

Matrix factorizations (a.k.a. matrix decompositions) compute the factorization of a matrix into a product of matrices, and are one of the central concepts in (numerical) linear algebra.

The following table summarizes the types of matrix factorizations that have been implemented in Julia. Details of their associated methods can be found in the Standard functions section of the Linear Algebra documentation.

Type	Description
`BunchKaufman`	Bunch-Kaufman factorization
`Cholesky`	Cholesky factorization
`CholeskyPivoted`	Pivoted Cholesky factorization
`LDLt`	LDL(T) factorization
`LU`	LU factorization
`QR`	QR factorization
`QRCompactWY`	Compact WY form of the QR factorization
`QRPivoted`	Pivoted QR factorization
`LQ`	QR factorization of `transpose(A)`
`Hessenberg`	Hessenberg decomposition
`Eigen`	Spectral decomposition
`GeneralizedEigen`	Generalized spectral decomposition
`SVD`	Singular value decomposition
`GeneralizedSVD`	Generalized SVD
`Schur`	Schur decomposition
`GeneralizedSchur`	Generalized Schur decomposition

Adjoints and transposes of `Factorization` objects are lazily wrapped in `AdjointFactorization` and `TransposeFactorization` objects, respectively. Generically, transpose of real `Factorization`s are wrapped as `AdjointFactorization`.

79.3 Orthogonal matrices (`AbstractQ`)

Some matrix factorizations generate orthogonal/unitary "matrix" factors. These factorizations include QR-related factorizations obtained from calls to `qr`, i.e., `QR`, `QRCompactWY` and `QRPivoted`, the Hessenberg factorization obtained from calls to `hessenberg`, and the LQ factorization obtained from `lq`. While these orthogonal/unitary factors admit a matrix representation, their internal representation is, for performance and memory reasons, different. Hence, they should be rather viewed as matrix-backed, function-based linear operators. In particular, reading, for instance, a column of its matrix representation requires running "matrix"-vector multiplication code, rather than simply reading out data from memory (possibly filling parts of the vector with structural zeros). Another clear distinction from other, non-triangular matrix types is that the underlying multiplication code allows for in-place modification during multiplication. Furthermore, objects of specific `AbstractQ` subtypes as those created via `qr`, `hessenberg` and `lq` can behave like a square or a rectangular matrix depending on context:

```
julia> using LinearAlgebra

julia> Q = qr(rand(3,2)).Q
3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
```

```
julia> Matrix(Q)
3×2 Matrix{Float64}:
 -0.320597   0.865734
 -0.765834  -0.475694
 -0.557419   0.155628

julia> Q*I
3×3 Matrix{Float64}:
 -0.320597   0.865734  -0.384346
 -0.765834  -0.475694  -0.432683
 -0.557419   0.155628   0.815514

julia> Q*ones(2)
3-element Vector{Float64}:
  0.5451367118802273
 -1.241527373086654
 -0.40179067589600226

julia> Q*ones(3)
3-element Vector{Float64}:
  0.16079054743832022
 -1.674209978965636
  0.41372375588835797

julia> ones(1,2) * Q'
1×3 Matrix{Float64}:
 0.545137  -1.24153  -0.401791

julia> ones(1,3) * Q'
1×3 Matrix{Float64}:
 0.160791  -1.67421  0.413724
```

Due to this distinction from dense or structured matrices, the abstract `AbstractQ` type does not subtype `AbstractMatrix`, but instead has its own type hierarchy. Custom types that subtype `AbstractQ` can rely on generic fallbacks if the following interface is satisfied. For example, for

```
struct MyQ{T} <: LinearAlgebra.AbstractQ{T}
    # required fields
end
```

provide overloads for

```
Base.size(Q::MyQ) # size of corresponding square matrix representation
Base.convert(::Type{AbstractQ{T}}, Q::MyQ) # eltype promotion [optional]
LinearAlgebra.lmul!(Q::MyQ, x::AbstractVecOrMat) # left-multiplication
LinearAlgebra.rmul!(A::AbstractMatrix, Q::MyQ) # right-multiplication
```

If `eltype` promotion is not of interest, the `convert` method is unnecessary, since by default `convert(::Type{AbstractQ{T}}, Q::AbstractQ{T})` returns `Q` itself. Adjoints of `AbstractQ`-typed objects are lazily wrapped in an `AdjointQ` wrapper type, which requires its own `LinearAlgebra.lmul!` and `LinearAlgebra.rmul!` methods. Given this set of methods, any `Q::MyQ` can be used like a matrix, preferably in a multiplicative context: multiplication via

* with scalars, vectors and matrices from left and right, obtaining a matrix representation of `Q` via `Matrix(Q)` (or `Q*I`) and indexing into the matrix representation all work. In contrast, addition and subtraction as well as more generally broadcasting over elements in the matrix representation fail because that would be highly inefficient. For such use cases, consider computing the matrix representation up front and cache it for future reuse.

79.4 Standard functions

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK. Sparse matrix factorizations call functions from SuiteSparse. Other sparse solvers are available as Julia packages.

`Base.:*` – Method.

```
*(A::AbstractMatrix, B::AbstractMatrix)
```

Matrix multiplication.

Examples

```
julia> [1 1; 0 1] * [1 0; 1 1]
2×2 Matrix{Int64}:
 2  1
 1  1
```

`Base.:\` – Method.

```
\(A, B)
```

Matrix division using a polyalgorithm. For input matrices A and B, the result X is such that `A*X == B` when A is square. The solver that is used depends upon the structure of A. If A is upper or lower triangular (or diagonal), no factorization of A is required and the system is solved with either forward or backward substitution. For non-triangular square matrices, an LU factorization is used.

For rectangular A the result is the minimum-norm least squares solution computed by a pivoted QR factorization of A and a rank estimate of A based on the R factor.

When A is sparse, a similar polyalgorithm is used. For indefinite matrices, the `LDLt` factorization does not use pivoting during the numerical factorization and therefore the procedure can fail even for invertible matrices.

See also: `factorize`, `pinv`.

Examples

```
julia> A = [1 0; 1 -2]; B = [32; -4];

julia> X = A \ B
2-element Vector{Float64}:
 32.0
 18.0

julia> A * X == B
true
```

Base.:/ – Method.

```
A / B
```

Matrix right-division: A / B is equivalent to (B' \ A')' where \ is the left-division operator. For square matrices, the result X is such that A == X*B.

See also: rdiv!.

Examples

```
julia> A = Float64[1 4 5; 3 9 2]; B = Float64[1 4 2; 3 4 2; 8 7 1];

julia> X = A / B
2×3 Matrix{Float64}:
 -0.65   3.75  -1.2
  3.25  -2.75   1.0

julia> isapprox(A, X*B)
true

julia> isapprox(X, A*pinv(B))
true
```

LinearAlgebra.SingularException – Type.

```
SingularException
```

Exception thrown when the input matrix has one or more zero-valued eigenvalues, and is not invertible. A linear solve involving such a matrix cannot be computed. The info field indicates the location of (one of) the singular value(s).

LinearAlgebra.PosDefException – Type.

```
PosDefException
```

Exception thrown when the input matrix was not positive definite. Some linear algebra functions and factorizations are only applicable to positive definite matrices. The info field indicates the location of (one of) the eigenvalue(s) which is (are) less than/equal to 0.

LinearAlgebra.ZeroPivotException – Type.

```
ZeroPivotException <: Exception
```

Exception thrown when a matrix factorization/solve encounters a zero in a pivot (diagonal) position and cannot proceed. This may *not* mean that the matrix is singular: it may be fruitful to switch to a different factorization such as pivoted LU that can re-order variables to eliminate spurious zero pivots. The info field indicates the location of (one of) the zero pivot(s).

`LinearAlgebra.dot` – Function.

```
dot(x, y)
x ⋅ y
```

Compute the dot product between two vectors. For complex vectors, the first vector is conjugated.

`dot` also works on arbitrary iterable objects, including arrays of any dimension, as long as `dot` is defined on the elements.

`dot` is semantically equivalent to `sum(dot(vx,vy) for (vx,vy) in zip(x, y))`, with the added restriction that the arguments must have equal lengths.

`x ⋅ y` (where `⋅` can be typed by tab-completing `\cdot` in the REPL) is a synonym for `dot(x, y)`.

Examples

```
julia> dot([1; 1], [2; 3])
5

julia> dot([im; im], [1; 1])
0 - 2im

julia> dot(1:5, 2:6)
70

julia> x = fill(2., (5,5));

julia> y = fill(3., (5,5));

julia> dot(x, y)
150.0
```

`LinearAlgebra.dot` – Method.

```
dot(x, A, y)
```

Compute the generalized dot product `dot(x, A*y)` between two vectors `x` and `y`, without storing the intermediate result of `A*y`. As for the two-argument `dot(_,_)`, this acts recursively. Moreover, for complex vectors, the first vector is conjugated.

> **Julia 1.4**
>
> Three-argument `dot` requires at least Julia 1.4.

Examples

```
julia> dot([1; 1], [1 2; 3 4], [2; 3])
26

julia> dot(1:5, reshape(1:25, 5, 5), 2:6)
```

```
4850

julia> ⋅(1:5, reshape(1:25, 5, 5), 2:6) == dot(1:5, reshape(1:25, 5, 5), 2:6)
true
```

`LinearAlgebra.cross` – Function.

```
cross(x, y)
×(x,y)
```

Compute the cross product of two 3-vectors.

Examples

```
julia> a = [0;1;0]
3-element Vector{Int64}:
 0
 1
 0

julia> b = [0;0;1]
3-element Vector{Int64}:
 0
 0
 1

julia> cross(a,b)
3-element Vector{Int64}:
 1
 0
 0
```

`LinearAlgebra.axpy!` – Function.

```
axpy!(α, x::AbstractArray, y::AbstractArray)
```

Overwrite y with `x * α + y` and return y. If x and y have the same axes, it's equivalent with `y .+= x .* a`.

Examples

```
julia> x = [1; 2; 3];

julia> y = [4; 5; 6];

julia> axpy!(2, x, y)
3-element Vector{Int64}:
  6
  9
 12
```

`LinearAlgebra.axpby!` – Function.

```
axpby!(α, x::AbstractArray, β, y::AbstractArray)
```

Overwrite y with `x * α + y * β` and return y. If x and y have the same axes, it's equivalent with `y .= x .* a .+ y .* β`.

Examples

```
julia> x = [1; 2; 3];

julia> y = [4; 5; 6];

julia> axpby!(2, x, 2, y)
3-element Vector{Int64}:
 10
 14
 18
```

`LinearAlgebra.rotate!` – Function.

```
rotate!(x, y, c, s)
```

Overwrite x with `c*x + s*y` and y with `-conj(s)*x + c*y`. Returns x and y.

> **Julia 1.5**
>
> `rotate!` requires at least Julia 1.5.

`LinearAlgebra.reflect!` – Function.

```
reflect!(x, y, c, s)
```

Overwrite x with `c*x + s*y` and y with `conj(s)*x - c*y`. Returns x and y.

> **Julia 1.5**
>
> `reflect!` requires at least Julia 1.5.

`LinearAlgebra.factorize` – Function.

```
factorize(A)
```

Compute a convenient factorization of A, based upon the type of the input matrix. `factorize` checks A to see if it is symmetric/triangular/etc. if A is passed as a generic matrix. `factorize` checks every element of A

Properties of A	type of factorization
Positive-definite	Cholesky (see cholesky)
Dense Symmetric/Hermitian	Bunch-Kaufman (see bunchkaufman)
Sparse Symmetric/Hermitian	LDLt (see ldlt)
Triangular	Triangular
Diagonal	Diagonal
Bidiagonal	Bidiagonal
Tridiagonal	LU (see lu)
Symmetric real tridiagonal	LDLt (see ldlt)
General square	LU (see lu)
General non-square	QR (see qr)

to verify/rule out each property. It will short-circuit as soon as it can rule out symmetry/triangular structure. The return value can be reused for efficient solving of multiple systems. For example: `A=factorize(A); x=A\b; y=A\C`.

If `factorize` is called on a Hermitian positive-definite matrix, for instance, then `factorize` will return a Cholesky factorization.

Examples

```
julia> A = Array(Bidiagonal(fill(1.0, (5, 5)), :U))
5×5 Matrix{Float64}:
 1.0  1.0  0.0  0.0  0.0
 0.0  1.0  1.0  0.0  0.0
 0.0  0.0  1.0  1.0  0.0
 0.0  0.0  0.0  1.0  1.0
 0.0  0.0  0.0  0.0  1.0

julia> factorize(A) # factorize will check to see that A is already factorized
5×5 Bidiagonal{Float64, Vector{Float64}}:
 1.0  1.0   ⋅    ⋅    ⋅
  ⋅   1.0  1.0   ⋅    ⋅
  ⋅    ⋅   1.0  1.0   ⋅
  ⋅    ⋅    ⋅   1.0  1.0
  ⋅    ⋅    ⋅    ⋅   1.0
```

This returns a `5×5 Bidiagonal{Float64}`, which can now be passed to other linear algebra functions (e.g. eigensolvers) which will use specialized methods for `Bidiagonal` types.

`LinearAlgebra.Diagonal` – Type.

```
Diagonal(V::AbstractVector)
```

Construct a lazy matrix with `V` as its diagonal.

See also `UniformScaling` for the lazy identity matrix `I`, `diagm` to make a dense matrix, and `diag` to extract diagonal elements.

Examples

```
julia> d = Diagonal([1, 10, 100])
3×3 Diagonal{Int64, Vector{Int64}}:
 1   ⋅    ⋅
 ⋅  10    ⋅
 ⋅   ⋅  100

julia> diagm([7, 13])
2×2 Matrix{Int64}:
 7   0
 0  13

julia> ans + I
2×2 Matrix{Int64}:
 8   0
 0  14

julia> I(2)
2×2 Diagonal{Bool, Vector{Bool}}:
 1  ⋅
 ⋅  1
```

Note that a one-column matrix is not treated like a vector, but instead calls the method `Diagonal(A::AbstractMatrix)` which extracts 1-element `diag(A)`:

```
julia> A = transpose([7.0 13.0])
2×1 transpose(::Matrix{Float64}) with eltype Float64:
  7.0
 13.0

julia> Diagonal(A)
1×1 Diagonal{Float64, Vector{Float64}}:
 7.0
```

```
Diagonal(A::AbstractMatrix)
```

Construct a matrix from the diagonal of `A`.

Examples

```
julia> A = permutedims(reshape(1:15, 5, 3))
3×5 Matrix{Int64}:
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15

julia> Diagonal(A)
3×3 Diagonal{Int64, Vector{Int64}}:
 1  ⋅   ⋅
 ⋅  7   ⋅
 ⋅  ⋅  13

julia> diag(A, 2)
```

```
3-element Vector{Int64}:
  3
  9
 15
```

```
Diagonal{T}(undef, n)
```

Construct an uninitialized `Diagonal{T}` of length `n`. See `undef`.

`LinearAlgebra.Bidiagonal` – Type.

```
Bidiagonal(dv::V, ev::V, uplo::Symbol) where V <: AbstractVector
```

Constructs an upper (`uplo=:U`) or lower (`uplo=:L`) bidiagonal matrix using the given diagonal (`dv`) and off-diagonal (`ev`) vectors. The result is of type `Bidiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). The length of `ev` must be one less than the length of `dv`.

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Vector{Int64}:
 7
 8
 9

julia> Bu = Bidiagonal(dv, ev, :U) # ev is on the first superdiagonal
4×4 Bidiagonal{Int64, Vector{Int64}}:
 1  7  ⋅  ⋅
 ⋅  2  8  ⋅
 ⋅  ⋅  3  9
 ⋅  ⋅  ⋅  4

julia> Bl = Bidiagonal(dv, ev, :L) # ev is on the first subdiagonal
4×4 Bidiagonal{Int64, Vector{Int64}}:
 1  ⋅  ⋅  ⋅
 7  2  ⋅  ⋅
 ⋅  8  3  ⋅
 ⋅  ⋅  9  4
```

```
Bidiagonal(A, uplo::Symbol)
```

Construct a Bidiagonal matrix from the main diagonal of A and its first super- (if uplo=:U) or sub-diagonal (if uplo=:L).

Examples

```
julia> A = [1 1 1 1; 2 2 2 2; 3 3 3 3; 4 4 4 4]
4×4 Matrix{Int64}:
 1  1  1  1
 2  2  2  2
 3  3  3  3
 4  4  4  4

julia> Bidiagonal(A, :U) # contains the main diagonal and first superdiagonal of A
4×4 Bidiagonal{Int64, Vector{Int64}}:
 1  1  ⋅  ⋅
 ⋅  2  2  ⋅
 ⋅  ⋅  3  3
 ⋅  ⋅  ⋅  4

julia> Bidiagonal(A, :L) # contains the main diagonal and first subdiagonal of A
4×4 Bidiagonal{Int64, Vector{Int64}}:
 1  ⋅  ⋅  ⋅
 2  2  ⋅  ⋅
 ⋅  3  3  ⋅
 ⋅  ⋅  4  4
```

LinearAlgebra.SymTridiagonal – Type.

```
SymTridiagonal(dv::V, ev::V) where V <: AbstractVector
```

Construct a symmetric tridiagonal matrix from the diagonal (dv) and first sub/super-diagonal (ev), respectively. The result is of type SymTridiagonal and provides efficient specialized eigensolvers, but may be converted into a regular matrix with convert(Array, _) (or Array(_) for short).

For SymTridiagonal block matrices, the elements of dv are symmetrized. The argument ev is interpreted as the superdiagonal. Blocks from the subdiagonal are (materialized) transpose of the corresponding superdiagonal blocks.

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Vector{Int64}:
 7
 8
 9
```

```
julia> SymTridiagonal(dv, ev)
4×4 SymTridiagonal{Int64, Vector{Int64}}:
 1  7  ⋅  ⋅
 7  2  8  ⋅
 ⋅  8  3  9
 ⋅  ⋅  9  4

julia> A = SymTridiagonal(fill([1 2; 3 4], 3), fill([1 2; 3 4], 2));

julia> A[1,1]
2×2 Symmetric{Int64, Matrix{Int64}}:
 1  2
 2  4

julia> A[1,2]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> A[2,1]
2×2 Matrix{Int64}:
 1  3
 2  4
```

```
SymTridiagonal(A::AbstractMatrix)
```

Construct a symmetric tridiagonal matrix from the diagonal and first superdiagonal of the symmetric matrix A.

Examples

```
julia> A = [1 2 3; 2 4 5; 3 5 6]
3×3 Matrix{Int64}:
 1  2  3
 2  4  5
 3  5  6

julia> SymTridiagonal(A)
3×3 SymTridiagonal{Int64, Vector{Int64}}:
 1  2  ⋅
 2  4  5
 ⋅  5  6

julia> B = reshape([[1 2; 2 3], [1 2; 3 4], [1 3; 2 4], [1 2; 2 3]], 2, 2);

julia> SymTridiagonal(B)
2×2 SymTridiagonal{Matrix{Int64}, Vector{Matrix{Int64}}}:
 [1 2; 2 3]  [1 3; 2 4]
 [1 2; 3 4]  [1 2; 2 3]
```

`LinearAlgebra.Tridiagonal` – Type.

```
Tridiagonal(dl::V, d::V, du::V) where V <: AbstractVector
```

Construct a tridiagonal matrix from the first subdiagonal, diagonal, and first superdiagonal, respectively. The result is of type `Tridiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). The lengths of `dl` and `du` must be one less than the length of `d`.

Examples

```
julia> dl = [1, 2, 3];

julia> du = [4, 5, 6];

julia> d = [7, 8, 9, 0];

julia> Tridiagonal(dl, d, du)
4×4 Tridiagonal{Int64, Vector{Int64}}:
 7  4  ⋅  ⋅
 1  8  5  ⋅
 ⋅  2  9  6
 ⋅  ⋅  3  0
```

```
Tridiagonal(A)
```

Construct a tridiagonal matrix from the first sub-diagonal, diagonal and first super-diagonal of the matrix `A`.

Examples

```
julia> A = [1 2 3 4; 1 2 3 4; 1 2 3 4; 1 2 3 4]
4×4 Matrix{Int64}:
 1  2  3  4
 1  2  3  4
 1  2  3  4
 1  2  3  4

julia> Tridiagonal(A)
4×4 Tridiagonal{Int64, Vector{Int64}}:
 1  2  ⋅  ⋅
 1  2  3  ⋅
 ⋅  2  3  4
 ⋅  ⋅  3  4
```

`LinearAlgebra.Symmetric` – Type.

```
Symmetric(A::AbstractMatrix, uplo::Symbol=:U)
```

Construct a `Symmetric` view of the upper (if `uplo = :U`) or lower (if `uplo = :L`) triangle of the matrix `A`.

Symmetric views are mainly useful for real-symmetric matrices, for which specialized algorithms (e.g. for eigenproblems) are enabled for Symmetric types. More generally, see also Hermitian(A) for Hermitian matrices A == A', which is effectively equivalent to Symmetric for real matrices but is also useful for complex matrices. (Whereas complex Symmetric matrices are supported but have few if any specialized algorithms.)

To compute the symmetric part of a real matrix, or more generally the Hermitian part (A + A') / 2 of a real or complex matrix A, use hermitianpart.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> Supper = Symmetric(A)
3×3 Symmetric{Int64, Matrix{Int64}}:
 1  2  3
 2  5  6
 3  6  9

julia> Slower = Symmetric(A, :L)
3×3 Symmetric{Int64, Matrix{Int64}}:
 1  4  7
 4  5  8
 7  8  9

julia> hermitianpart(A)
3×3 Hermitian{Float64, Matrix{Float64}}:
 1.0  3.0  5.0
 3.0  5.0  7.0
 5.0  7.0  9.0
```

Note that Supper will not be equal to Slower unless A is itself symmetric (e.g. if A == transpose(A)).

LinearAlgebra.Hermitian – Type.

```
Hermitian(A::AbstractMatrix, uplo::Symbol=:U)
```

Construct a Hermitian view of the upper (if uplo = :U) or lower (if uplo = :L) triangle of the matrix A.

To compute the Hermitian part of A, use hermitianpart.

Examples

```
julia> A = [1 2+2im 3-3im; 4 5 6-6im; 7 8+8im 9]
3×3 Matrix{Complex{Int64}}:
 1+0im  2+2im  3-3im
 4+0im  5+0im  6-6im
 7+0im  8+8im  9+0im

julia> Hupper = Hermitian(A)
```

```
3×3 Hermitian{Complex{Int64}, Matrix{Complex{Int64}}}:
 1+0im  2+2im  3-3im
 2-2im  5+0im  6-6im
 3+3im  6+6im  9+0im

julia> Hlower = Hermitian(A, :L)
3×3 Hermitian{Complex{Int64}, Matrix{Complex{Int64}}}:
 1+0im  4+0im  7+0im
 4+0im  5+0im  8-8im
 7+0im  8+8im  9+0im

julia> hermitianpart(A)
3×3 Hermitian{ComplexF64, Matrix{ComplexF64}}:
 1.0+0.0im  3.0+1.0im  5.0-1.5im
 3.0-1.0im  5.0+0.0im  7.0-7.0im
 5.0+1.5im  7.0+7.0im  9.0+0.0im
```

Note that `Hupper` will not be equal to `Hlower` unless `A` is itself Hermitian (e.g. if `A == adjoint(A)`).

All non-real parts of the diagonal will be ignored.

```
Hermitian(fill(complex(1,1), 1, 1)) == fill(1, 1, 1)
```

`LinearAlgebra.LowerTriangular` – Type.

```
LowerTriangular(A::AbstractMatrix)
```

Construct a `LowerTriangular` view of the matrix `A`.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> LowerTriangular(A)
3×3 LowerTriangular{Float64, Matrix{Float64}}:
 1.0   ⋅    ⋅
 4.0  5.0   ⋅
 7.0  8.0  9.0
```

`LinearAlgebra.UpperTriangular` – Type.

```
UpperTriangular(A::AbstractMatrix)
```

Construct an `UpperTriangular` view of the matrix `A`.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UpperTriangular(A)
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 1.0  2.0  3.0
  ⋅   5.0  6.0
  ⋅    ⋅   9.0
```

`LinearAlgebra.UnitLowerTriangular` – Type.

```
UnitLowerTriangular(A::AbstractMatrix)
```

Construct a `UnitLowerTriangular` view of the matrix `A`. Such a view has the `oneunit` of the `eltype` of `A` on its diagonal.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UnitLowerTriangular(A)
3×3 UnitLowerTriangular{Float64, Matrix{Float64}}:
 1.0   ⋅    ⋅
 4.0  1.0   ⋅
 7.0  8.0  1.0
```

`LinearAlgebra.UnitUpperTriangular` – Type.

```
UnitUpperTriangular(A::AbstractMatrix)
```

Construct an `UnitUpperTriangular` view of the matrix `A`. Such a view has the `oneunit` of the `eltype` of `A` on its diagonal.

Examples

```
julia> A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]
3×3 Matrix{Float64}:
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0

julia> UnitUpperTriangular(A)
3×3 UnitUpperTriangular{Float64, Matrix{Float64}}:
```

```
 1.0  2.0  3.0
  ⋅   1.0  6.0
  ⋅    ⋅   1.0
```

`LinearAlgebra.UpperHessenberg` – Type.

```
UpperHessenberg(A::AbstractMatrix)
```

Construct an `UpperHessenberg` view of the matrix `A`. Entries of `A` below the first subdiagonal are ignored.

> **Julia 1.3**
>
> This type was added in Julia 1.3.

Efficient algorithms are implemented for `H \ b`, `det(H)`, and similar.

See also the `hessenberg` function to factor any matrix into a similar upper-Hessenberg matrix.

If `F::Hessenberg` is the factorization object, the unitary matrix can be accessed with `F.Q` and the Hessenberg matrix with `F.H`. When `Q` is extracted, the resulting type is the `HessenbergQ` object, and may be converted to a regular matrix with `convert(Array, _)` (or `Array(_)` for short).

Iterating the decomposition produces the factors `F.Q` and `F.H`.

Examples

```
julia> A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
4×4 Matrix{Int64}:
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16

julia> UpperHessenberg(A)
4×4 UpperHessenberg{Int64, Matrix{Int64}}:
 1   2   3   4
 5   6   7   8
 ⋅  10  11  12
 ⋅   ⋅  15  16
```

`LinearAlgebra.UniformScaling` – Type.

```
UniformScaling{T<:Number}
```

Generically sized uniform scaling operator defined as a scalar times the identity operator, `λ*I`. Although without an explicit `size`, it acts similarly to a matrix in many cases and includes support for some indexing. See also `I`.

> **Julia 1.6**
>
> Indexing using ranges is available as of Julia 1.6.

Examples

```
julia> J = UniformScaling(2.)
UniformScaling{Float64}
2.0*I

julia> A = [1. 2.; 3. 4.]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> J*A
2×2 Matrix{Float64}:
 2.0  4.0
 6.0  8.0

julia> J[1:2, 1:2]
2×2 Matrix{Float64}:
 2.0  0.0
 0.0  2.0
```

`LinearAlgebra.I` – Constant.

```
I
```

An object of type `UniformScaling`, representing an identity matrix of any size.

Examples

```
julia> fill(1, (5,6)) * I == fill(1, (5,6))
true

julia> [1 2im 3; 1im 2 3] * I
2×3 Matrix{Complex{Int64}}:
 1+0im  0+2im  3+0im
 0+1im  2+0im  3+0im
```

`LinearAlgebra.UniformScaling` – Method.

```
(I::UniformScaling)(n::Integer)
```

Construct a `Diagonal` matrix from a `UniformScaling`.

Julia 1.2

This method is available as of Julia 1.2.

Examples

```
julia> I(3)
3×3 Diagonal{Bool, Vector{Bool}}:
 1  ⋅  ⋅
 ⋅  1  ⋅
 ⋅  ⋅  1

julia> (0.7*I)(3)
3×3 Diagonal{Float64, Vector{Float64}}:
 0.7   ⋅    ⋅
  ⋅   0.7   ⋅
  ⋅    ⋅   0.7
```

`LinearAlgebra.Factorization` – Type.

```
LinearAlgebra.Factorization
```

Abstract type for matrix factorizations a.k.a. matrix decompositions. See online documentation for a list of available matrix factorizations.

`LinearAlgebra.LU` – Type.

```
LU <: Factorization
```

Matrix factorization type of the `LU` factorization of a square matrix `A`. This is the return type of `lu`, the corresponding matrix factorization function.

The individual components of the factorization `F::LU` can be accessed via `getproperty`:

Component	Description
`F.L`	`L` (unit lower triangular) part of `LU`
`F.U`	`U` (upper triangular) part of `LU`
`F.p`	(right) permutation `Vector`
`F.P`	(right) permutation `Matrix`

Iterating the factorization produces the components `F.L`, `F.U`, and `F.p`.

Examples

```
julia> A = [4 3; 6 3]
2×2 Matrix{Int64}:
 4  3
 6  3

julia> F = lu(A)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
2×2 Matrix{Float64}:
 1.0       0.0
 0.666667  1.0
```

```
U factor:
2×2 Matrix{Float64}:
 6.0  3.0
 0.0  1.0

julia> F.L * F.U == A[F.p, :]
true

julia> l, u, p = lu(A); # destructuring via iteration

julia> l == F.L && u == F.U && p == F.p
true
```

`LinearAlgebra.lu` – Function.

```
lu(A::AbstractSparseMatrixCSC; check = true, q = nothing, control = get_umfpack_control()) ->
↪  F::UmfpackLU
```

Compute the LU factorization of a sparse matrix `A`.

For sparse `A` with real or complex element type, the return type of `F` is `UmfpackLU{Tv, Ti}`, with `Tv` = `Float64` or `ComplexF64` respectively and `Ti` is an integer type (`Int32` or `Int64`).

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

The permutation `q` can either be a permutation vector or `nothing`. If no permutation vector is provided or `q` is `nothing`, UMFPACK's default is used. If the permutation is not zero-based, a zero-based copy is made.

The `control` vector defaults to the package's default configuration for UMFPACK, but can be changed by passing a vector of length `UMFPACK_CONTROL`. See the UMFPACK manual for possible configurations. The corresponding variables are named `JL_UMFPACK_` since Julia uses one-based indexing.

The individual components of the factorization `F` can be accessed by indexing:

Component	Description
L	L (lower triangular) part of LU
U	U (upper triangular) part of LU
p	right permutation `Vector`
q	left permutation `Vector`
Rs	`Vector` of scaling factors
:	`(L,U,p,q,Rs)` components

The relation between `F` and `A` is

`F.L*F.U == (F.Rs .* A)[F.p, F.q]`

`F` further supports the following functions:

- `\`
- `det`

See also `lu!`

Note

`lu(A::AbstractSparseMatrixCSC)` uses the UMFPACK[1] library that is part of SuiteSparse. As this library only supports sparse matrices with `Float64` or `ComplexF64` elements, `lu` converts `A` into a copy that is of type `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{ComplexF64}` as appropriate.

source

```
lu(A, pivot = RowMaximum(); check = true) -> F::LU
```

Compute the LU factorization of `A`.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

In most cases, if `A` is a subtype `S` of `AbstractMatrix{T}` with an element type `T` supporting `+`, `-`, `*` and `/`, the return type is `LU{T,S{T}}`.

In general, LU factorization involves a permutation of the rows of the matrix (corresponding to the `F.p` output described below), known as "pivoting" (because it corresponds to choosing which row contains the "pivot", the diagonal entry of `F.U`). One of the following pivoting strategies can be selected via the optional `pivot` argument:

- `RowMaximum()` (default): the standard pivoting strategy; the pivot corresponds to the element of maximum absolute value among the remaining, to be factorized rows. This pivoting strategy requires the element type to also support `abs` and `<`. (This is generally the only numerically stable option for floating-point matrices.)
- `RowNonZero()`: the pivot corresponds to the first non-zero element among the remaining, to be factorized rows. (This corresponds to the typical choice in hand calculations, and is also useful for more general algebraic number types that support `iszero` but not `abs` or `<`.)
- `NoPivot()`: pivoting turned off (may fail if a zero entry is encountered).

The individual components of the factorization `F` can be accessed via `getproperty`:

Component	Description
`F.L`	`L` (lower triangular) part of `LU`
`F.U`	`U` (upper triangular) part of `LU`
`F.p`	(right) permutation `Vector`
`F.P`	(right) permutation `Matrix`

Iterating the factorization produces the components `F.L`, `F.U`, and `F.p`.

The relationship between `F` and `A` is

`F.L*F.U == A[F.p, :]`

`F` further supports the following functions:

[1] Davis, Timothy A. (2004b). Algorithm 832: UMFPACK V4.3—an Unsymmetric-Pattern Multifrontal Method. ACM Trans. Math. Softw., 30(2), 196–199. doi:10.1145/992200.992206

Supported function	LU	LU{T,Tridiagonal{T}}
/	✓	
\	✓	✓
inv	✓	✓
det	✓	✓
logdet	✓	✓
logabsdet	✓	✓
size	✓	✓

Examples

```
julia> A = [4 3; 6 3]
2×2 Matrix{Int64}:
 4  3
 6  3

julia> F = lu(A)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
2×2 Matrix{Float64}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Matrix{Float64}:
 6.0  3.0
 0.0  1.0

julia> F.L * F.U == A[F.p, :]
true

julia> l, u, p = lu(A); # destructuring via iteration

julia> l == F.L && u == F.U && p == F.p
true
```

`LinearAlgebra.lu!` – Function.

```
lu!(F::UmfpackLU, A::AbstractSparseMatrixCSC; check=true, reuse_symbolic=true, q=nothing) ->
↪  F::UmfpackLU
```

Compute the LU factorization of a sparse matrix A, reusing the symbolic factorization of an already existing LU factorization stored in F. Unless `reuse_symbolic` is set to false, the sparse matrix A must have an identical nonzero pattern as the matrix used to create the LU factorization F, otherwise an error is thrown. If the size of A and F differ, all vectors will be resized accordingly.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

The permutation q can either be a permutation vector or `nothing`. If no permutation vector is provided or q is `nothing`, UMFPACK's default is used. If the permutation is not zero based, a zero based copy is made.

See also `lu`

Note

`lu!(F::UmfpackLU, A::AbstractSparseMatrixCSC)` uses the UMFPACK library that is part of SuiteSparse. As this library only supports sparse matrices with `Float64` or `ComplexF64` elements, `lu!` will automatically convert the types to those set by the LU factorization or `SparseMatrixCSC{ComplexF64}` as appropriate.

Julia 1.5

`lu!` for `UmfpackLU` requires at least Julia 1.5.

Examples

```
julia> A = sparse(Float64[1.0 2.0; 0.0 3.0]);

julia> F = lu(A);

julia> B = sparse(Float64[1.0 1.0; 0.0 1.0]);

julia> lu!(F, B);

julia> F \ ones(2)
2-element Vector{Float64}:
 0.0
 1.0
```

source

```
lu!(A, pivot = RowMaximum(); check = true) -> LU
```

`lu!` is the same as `lu`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Examples

```
julia> A = [4. 3.; 6. 3.]
2×2 Matrix{Float64}:
 4.0  3.0
 6.0  3.0

julia> F = lu!(A)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
2×2 Matrix{Float64}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Matrix{Float64}:
 6.0  3.0
 0.0  1.0
```

```
julia> iA = [4 3; 6 3]
2×2 Matrix{Int64}:
 4  3
 6  3

julia> lu!(iA)
ERROR: InexactError: Int64(0.6666666666666666)
Stacktrace:
[...]
```

`LinearAlgebra.Cholesky` – Type.

```
Cholesky <: Factorization
```

Matrix factorization type of the Cholesky factorization of a dense symmetric/Hermitian positive definite matrix `A`. This is the return type of `cholesky`, the corresponding matrix factorization function.

The triangular Cholesky factor can be obtained from the factorization `F::Cholesky` via `F.L` and `F.U`, where `A ≈ F.U' * F.U ≈ F.L * F.L'`.

The following functions are available for `Cholesky` objects: `size`, `\`, `inv`, `det`, `logdet` and `isposdef`.

Iterating the decomposition produces the components `L` and `U`.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Matrix{Float64}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A)
Cholesky{Float64, Matrix{Float64}}
U factor:
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 2.0  6.0  -8.0
  ⋅   1.0   5.0
  ⋅    ⋅    3.0

julia> C.U
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 2.0  6.0  -8.0
  ⋅   1.0   5.0
  ⋅    ⋅    3.0

julia> C.L
3×3 LowerTriangular{Float64, Matrix{Float64}}:
  2.0   ⋅    ⋅
  6.0  1.0   ⋅
 -8.0  5.0  3.0

julia> C.L * C.U == A
```

```
true

julia> l, u = C; # destructuring via iteration

julia> l == C.L && u == C.U
true
```

`LinearAlgebra.CholeskyPivoted` – Type.

```
CholeskyPivoted
```

Matrix factorization type of the pivoted Cholesky factorization of a dense symmetric/Hermitian positive semi-definite matrix `A`. This is the return type of `cholesky(_, ::RowMaximum)`, the corresponding matrix factorization function.

The triangular Cholesky factor can be obtained from the factorization `F::CholeskyPivoted` via `F.L` and `F.U`, and the permutation via `F.p`, where `A[F.p, F.p] ≈ Ur' * Ur ≈ Lr * Lr'` with `Ur = F.U[1:F.rank, :]` and `Lr = F.L[:, 1:F.rank]`, or alternatively `A ≈ Up' * Up ≈ Lp * Lp'` with `Up = F.U[1:F.rank, invperm(F.p)]` and `Lp = F.L[invperm(F.p), 1:F.rank]`.

The following functions are available for `CholeskyPivoted` objects: `size`, `\`, `inv`, `det`, and `rank`.

Iterating the decomposition produces the components `L` and `U`.

Examples

```
julia> X = [1.0, 2.0, 3.0, 4.0];

julia> A = X * X';

julia> C = cholesky(A, RowMaximum(), check = false)
CholeskyPivoted{Float64, Matrix{Float64}, Vector{Int64}}
U factor with rank 1:
4×4 UpperTriangular{Float64, Matrix{Float64}}:
 4.0  2.0  3.0  1.0
  ⋅   0.0  6.0  2.0
  ⋅    ⋅   9.0  3.0
  ⋅    ⋅    ⋅   1.0
permutation:
4-element Vector{Int64}:
 4
 2
 3
 1

julia> C.U[1:C.rank, :]' * C.U[1:C.rank, :] ≈ A[C.p, C.p]
true

julia> l, u = C; # destructuring via iteration

julia> l == C.L && u == C.U
true
```

`LinearAlgebra.cholesky` – Function.

```
cholesky(A, NoPivot(); check = true) -> Cholesky
```

Compute the Cholesky factorization of a dense symmetric positive definite matrix `A` and return a `Cholesky` factorization. The matrix `A` can either be a `Symmetric` or `Hermitian AbstractMatrix` or a *perfectly* symmetric or Hermitian `AbstractMatrix`.

The triangular Cholesky factor can be obtained from the factorization `F` via `F.L` and `F.U`, where `A ≈ F.U' * F.U ≈ F.L * F.L'`.

The following functions are available for `Cholesky` objects: `size`, `\`, `inv`, `det`, `logdet` and `isposdef`.

If you have a matrix `A` that is slightly non-Hermitian due to roundoff errors in its construction, wrap it in `Hermitian(A)` before passing it to `cholesky` in order to treat it as perfectly Hermitian.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Matrix{Float64}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A)
Cholesky{Float64, Matrix{Float64}}
U factor:
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 2.0  6.0  -8.0
  ⋅   1.0   5.0
  ⋅    ⋅    3.0

julia> C.U
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 2.0  6.0  -8.0
  ⋅   1.0   5.0
  ⋅    ⋅    3.0

julia> C.L
3×3 LowerTriangular{Float64, Matrix{Float64}}:
  2.0   ⋅    ⋅
  6.0  1.0   ⋅
 -8.0  5.0  3.0

julia> C.L * C.U == A
true
```

```
cholesky(A, RowMaximum(); tol = 0.0, check = true) -> CholeskyPivoted
```

Compute the pivoted Cholesky factorization of a dense symmetric positive semi-definite matrix A and return a `CholeskyPivoted` factorization. The matrix A can either be a `Symmetric` or `Hermitian` `AbstractMatrix` or a *perfectly* symmetric or Hermitian `AbstractMatrix`.

The triangular Cholesky factor can be obtained from the factorization F via `F.L` and `F.U`, and the permutation via `F.p`, where `A[F.p, F.p] ≈ Ur' * Ur ≈ Lr * Lr'` with `Ur = F.U[1:F.rank, :]` and `Lr = F.L[:, 1:F.rank]`, or alternatively `A ≈ Up' * Up ≈ Lp * Lp'` with `Up = F.U[1:F.rank, invperm(F.p)]` and `Lp = F.L[invperm(F.p), 1:F.rank]`.

The following functions are available for `CholeskyPivoted` objects: `size`, `\`, `inv`, `det`, and `rank`.

The argument `tol` determines the tolerance for determining the rank. For negative values, the tolerance is the machine precision.

If you have a matrix A that is slightly non-Hermitian due to roundoff errors in its construction, wrap it in `Hermitian(A)` before passing it to `cholesky` in order to treat it as perfectly Hermitian.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

Examples

```
julia> X = [1.0, 2.0, 3.0, 4.0];

julia> A = X * X';

julia> C = cholesky(A, RowMaximum(), check = false)
CholeskyPivoted{Float64, Matrix{Float64}, Vector{Int64}}
U factor with rank 1:
4×4 UpperTriangular{Float64, Matrix{Float64}}:
 4.0  2.0  3.0  1.0
  ⋅   0.0  6.0  2.0
  ⋅    ⋅   9.0  3.0
  ⋅    ⋅    ⋅   1.0
permutation:
4-element Vector{Int64}:
 4
 2
 3
 1

julia> C.U[1:C.rank, :]' * C.U[1:C.rank, :] ≈ A[C.p, C.p]
true

julia> l, u = C; # destructuring via iteration

julia> l == C.L && u == C.U
true
```

```
cholesky(A::SparseMatrixCSC; shift = 0.0, check = true, perm = nothing) -> CHOLMOD.Factor
```

Compute the Cholesky factorization of a sparse positive definite matrix A. A must be a `SparseMatrixCSC` or a `Symmetric`/`Hermitian` view of a `SparseMatrixCSC`. Note that even if A doesn't have the type tag, it must still be symmetric or Hermitian. If `perm` is not given, a fill-reducing permutation is used. `F = cholesky(A)` is most frequently used to solve systems of equations with `F\b`, but also the methods `diag`, `det`, and

logdet are defined for F. You can also extract individual factors from F, using F.L. However, since pivoting is on by default, the factorization is internally represented as A == P'*L*L'*P with a permutation matrix P; using just L without accounting for P will give incorrect answers. To include the effects of permutation, it's typically preferable to extract "combined" factors like PtL = F.PtL (the equivalent of P'*L) and LtP = F.UP (the equivalent of L'*P).

When check = true, an error is thrown if the decomposition fails. When check = false, responsibility for checking the decomposition's validity (via issuccess) lies with the user.

Setting the optional shift keyword argument computes the factorization of A+shift*I instead of A. If the perm argument is provided, it should be a permutation of 1:size(A,1) giving the ordering to use (instead of CHOLMOD's default AMD ordering).

Examples

In the following example, the fill-reducing permutation used is [3, 2, 1]. If perm is set to 1:3 to enforce no permutation, the number of nonzero elements in the factor is 6.

```
julia> A = [2 1 1; 1 2 0; 1 0 2]
3×3 Matrix{Int64}:
 2  1  1
 1  2  0
 1  0  2

julia> C = cholesky(sparse(A))
SparseArrays.CHOLMOD.Factor{Float64, Int64}
type:    LLt
method:  simplicial
maxnnz:  5
nnz:     5
success: true

julia> C.p
3-element Vector{Int64}:
 3
 2
 1

julia> L = sparse(C.L);

julia> Matrix(L)
3×3 Matrix{Float64}:
 1.41421   0.0       0.0
 0.0       1.41421   0.0
 0.707107  0.707107  1.0

julia> L * L' ≈ A[C.p, C.p]
true

julia> P = sparse(1:3, C.p, ones(3))
3×3 SparseMatrixCSC{Float64, Int64} with 3 stored entries:
  ⋅    ⋅   1.0
  ⋅   1.0   ⋅
 1.0   ⋅    ⋅

julia> P' * L * L' * P ≈ A
true
```

```
julia> C = cholesky(sparse(A), perm=1:3)
SparseArrays.CHOLMOD.Factor{Float64, Int64}
type:    LLt
method:  simplicial
maxnnz:  6
nnz:     6
success: true

julia> L = sparse(C.L);

julia> Matrix(L)
3×3 Matrix{Float64}:
 1.41421    0.0       0.0
 0.707107   1.22474   0.0
 0.707107  -0.408248  1.1547

julia> L * L' ≈ A
true
```

Note

This method uses the CHOLMOD[2,3] library from SuiteSparse. CHOLMOD only supports double or complex double element types. Input matrices not of those element types will be converted to `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{ComplexF64}` as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the `Base.SparseArrays.CHOLMOD` module.

source

`LinearAlgebra.cholesky!` – Function.

```
cholesky!(A::AbstractMatrix, NoPivot(); check = true) -> Cholesky
```

The same as `cholesky`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Examples

```
julia> A = [1 2; 2 50]
2×2 Matrix{Int64}:
 1   2
 2  50
```

[2] Chen, Y., Davis, T. A., Hager, W. W., & Rajamanickam, S. (2008). Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. ACM Trans. Math. Softw., 35(3). doi:10.1145/1391989.1391995

[3] Davis, Timothy A., & Hager, W. W. (2009). Dynamic Supernodes in Sparse Cholesky Update/Downdate and Triangular Solves. ACM Trans. Math. Softw., 35(4). doi:10.1145/1462173.1462176

```
julia> cholesky!(A)
ERROR: InexactError: Int64(6.782329983125268)
Stacktrace:
[...]
```

```
cholesky!(A::AbstractMatrix, RowMaximum(); tol = 0.0, check = true) -> CholeskyPivoted
```

The same as `cholesky`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

```
cholesky!(F::CHOLMOD.Factor, A::SparseMatrixCSC; shift = 0.0, check = true) -> CHOLMOD.Factor
```

Compute the Cholesky (LL') factorization of `A`, reusing the symbolic factorization `F`. `A` must be a `SparseMatrixCSC` or a `Symmetric`/ `Hermitian` view of a `SparseMatrixCSC`. Note that even if `A` doesn't have the type tag, it must still be symmetric or Hermitian.

See also `cholesky`.

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{ComplexF64}` as appropriate.

source

`LinearAlgebra.lowrankupdate` – Function.

```
lowrankupdate(C::Cholesky, v::AbstractVector) -> CC::Cholesky
```

Update a Cholesky factorization `C` with the vector `v`. If `A = C.U'C.U` then `CC = cholesky(C.U'C.U + v*v')` but the computation of `CC` only uses `O(n^2)` operations.

```
lowrankupdate(F::CHOLMOD.Factor, C::AbstractArray) -> FF::CHOLMOD.Factor
```

Get an `LDLt` Factorization of `A + C*C'` given an `LDLt` or `LLt` factorization `F` of `A`.

The returned factor is always an `LDLt` factorization.

See also `lowrankupdate!`, `lowrankdowndate`, `lowrankdowndate!`.

source

`LinearAlgebra.lowrankdowndate` – Function.

```
lowrankdowndate(C::Cholesky, v::AbstractVector) -> CC::Cholesky
```

Downdate a Cholesky factorization C with the vector v. If A = C.U'C.U then CC = cholesky(C.U'C.U - v*v') but the computation of CC only uses O(n^2) operations.

```
lowrankdowndate(F::CHOLMOD.Factor, C::AbstractArray) -> FF::CHOLMOD.Factor
```

Get an LDLt Factorization of A + C*C' given an LDLt or LLt factorization F of A.

The returned factor is always an LDLt factorization.

See also lowrankdowndate!, lowrankupdate, lowrankupdate!.

source

LinearAlgebra.lowrankupdate! – Function.

```
lowrankupdate!(C::Cholesky, v::AbstractVector) -> CC::Cholesky
```

Update a Cholesky factorization C with the vector v. If A = C.U'C.U then CC = cholesky(C.U'C.U + v*v') but the computation of CC only uses O(n^2) operations. The input factorization C is updated in place such that on exit C == CC. The vector v is destroyed during the computation.

```
lowrankupdate!(F::CHOLMOD.Factor, C::AbstractArray)
```

Update an LDLt or LLt Factorization F of A to a factorization of A + C*C'.

LLt factorizations are converted to LDLt.

See also lowrankupdate, lowrankdowndate, lowrankdowndate!.

source

LinearAlgebra.lowrankdowndate! – Function.

```
lowrankdowndate!(C::Cholesky, v::AbstractVector) -> CC::Cholesky
```

Downdate a Cholesky factorization C with the vector v. If A = C.U'C.U then CC = cholesky(C.U'C.U - v*v') but the computation of CC only uses O(n^2) operations. The input factorization C is updated in place such that on exit C == CC. The vector v is destroyed during the computation.

```
lowrankdowndate!(F::CHOLMOD.Factor, C::AbstractArray)
```

Update an LDLt or LLt Factorization F of A to a factorization of A - C*C'.

LLt factorizations are converted to LDLt.

See also lowrankdowndate, lowrankupdate, lowrankupdate!.

source

LinearAlgebra.LDLt – Type.

```
LDLt <: Factorization
```

Matrix factorization type of the LDLt factorization of a real SymTridiagonal matrix S such that S = L*Diagonal(d)*L', where L is a UnitLowerTriangular matrix and d is a vector. The main use of an LDLt factorization F = ldlt(S) is to solve the linear system of equations Sx = b with F\b. This is the return type of ldlt, the corresponding matrix factorization function.

The individual components of the factorization F::LDLt can be accessed via getproperty:

Component	Description
F.L	L (unit lower triangular) part of LDLt
F.D	D (diagonal) part of LDLt
F.Lt	Lt (unit upper triangular) part of LDLt
F.d	diagonal values of D as a Vector

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 3.0  1.0   ⋅
 1.0  4.0  2.0
  ⋅   2.0  5.0

julia> F = ldlt(S)
LDLt{Float64, SymTridiagonal{Float64, Vector{Float64}}}
L factor:
3×3 UnitLowerTriangular{Float64, SymTridiagonal{Float64, Vector{Float64}}}:
 1.0        ⋅         ⋅
 0.333333  1.0        ⋅
 0.0       0.545455  1.0
D factor:
3×3 Diagonal{Float64, Vector{Float64}}:
 3.0   ⋅        ⋅
  ⋅   3.66667   ⋅
  ⋅    ⋅       3.90909
```

LinearAlgebra.ldlt – Function.

```
ldlt(S::SymTridiagonal) -> LDLt
```

Compute an LDLt (i.e., LDL^T) factorization of the real symmetric tridiagonal matrix S such that S = L*Diagonal(d)*L' where L is a unit lower triangular matrix and d is a vector. The main use of an LDLt factorization F = ldlt(S) is to solve the linear system of equations Sx = b with F\b.

See also bunchkaufman for a similar, but pivoted, factorization of arbitrary symmetric or Hermitian matrices.

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 3.0  1.0   ⋅
 1.0  4.0  2.0
  ⋅   2.0  5.0

julia> ldltS = ldlt(S);

julia> b = [6., 7., 8.];

julia> ldltS \ b
3-element Vector{Float64}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255

julia> S \ b
3-element Vector{Float64}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255
```

```
ldlt(A::SparseMatrixCSC; shift = 0.0, check = true, perm=nothing) -> CHOLMOD.Factor
```

Compute the LDL' factorization of a sparse matrix A. A must be a `SparseMatrixCSC` or a `Symmetric`/`Hermitian` view of a `SparseMatrixCSC`. Note that even if A doesn't have the type tag, it must still be symmetric or Hermitian. A fill-reducing permutation is used. `F = ldlt(A)` is most frequently used to solve systems of equations `A*x = b` with `F\b`. The returned factorization object F also supports the methods `diag`, `det`, `logdet`, and `inv`. You can extract individual factors from F using `F.L`. However, since pivoting is on by default, the factorization is internally represented as `A == P'*L*D*L'*P` with a permutation matrix P; using just L without accounting for P will give incorrect answers. To include the effects of permutation, it is typically preferable to extract "combined" factors like `PtL = F.PtL` (the equivalent of `P'*L`) and `LtP = F.UP` (the equivalent of `L'*P`). The complete list of supported factors is `:L, :PtL, :D, :UP, :U, :LD, :DU, :PtLD, :DUP`.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

Setting the optional `shift` keyword argument computes the factorization of `A+shift*I` instead of A. If the `perm` argument is provided, it should be a permutation of `1:size(A,1)` giving the ordering to use (instead of CHOLMOD's default AMD ordering).

Note

This method uses the CHOLMOD[2,3] library from SuiteSparse. CHOLMOD only supports double or complex double element types. Input matrices not of those element types will be converted to `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{ComplexF64}` as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the `Base.SparseArrays.CHOLMOD` module.

source

`LinearAlgebra.ldlt!` – Function.

```
ldlt!(S::SymTridiagonal) -> LDLt
```

Same as `ldlt`, but saves space by overwriting the input S, instead of creating a copy.

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 3.0  1.0   ⋅
 1.0  4.0  2.0
  ⋅   2.0  5.0

julia> ldltS = ldlt!(S);

julia> ldltS === S
false

julia> S
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 3.0       0.333333   ⋅
 0.333333  3.66667   0.545455
  ⋅        0.545455  3.90909
```

```
ldlt!(F::CHOLMOD.Factor, A::SparseMatrixCSC; shift = 0.0, check = true) -> CHOLMOD.Factor
```

Compute the LDL' factorization of A, reusing the symbolic factorization F. A must be a `SparseMatrixCSC` or a `Symmetric`/`Hermitian` view of a `SparseMatrixCSC`. Note that even if A doesn't have the type tag, it must still be symmetric or Hermitian.

See also `ldlt`.

> **Note**
>
> This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{ComplexF64}` as appropriate.

source

`LinearAlgebra.QR` – Type.

```
QR <: Factorization
```

A QR matrix factorization stored in a packed format, typically obtained from `qr`. If A is an `m×n` matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors v_i and coefficients τ_i where:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

Iterating the decomposition produces the components `Q` and `R`.

The object has two fields:

- `factors` is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix `V = I + tril(F.factors, -1)`.
- `τ` is a vector of length `min(m,n)` containing the coefficients au_i.

`LinearAlgebra.QRCompactWY` – Type.

```
QRCompactWY <: Factorization
```

A QR matrix factorization stored in a compact blocked format, typically obtained from `qr`. If A is an `m×n` matrix, then

$$A = QR$$

where Q is an orthogonal/unitary matrix and R is upper triangular. It is similar to the `QR` format except that the orthogonal/unitary matrix Q is stored in *Compact WY* format [4]. For the block size n_b, it is stored as a `m×n` lower trapezoidal matrix V and a matrix $T = (T_1 \; T_2 \; ... \; T_{b-1} \; T_b')$ composed of $b = \lceil \min(m,n)/n_b \rceil$ upper triangular matrices T_j of size n_b×n_b ($j = 1, ..., b-1$) and an upper trapezoidal n_b×$\min(m,n) - (b-1)n_b$ matrix T_b' ($j = b$) whose upper square part denoted with T_b satisfying

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T) = \prod_{j=1}^{b} (I - V_j T_j V_j^T)$$

such that v_i is the ith column of V, τ_i is the ith element of `[diag(T_1); diag(T_2); ⋯; diag(T_b)]`, and $(V_1 \; V_2 \; ... \; V_b)$ is the left `m×min(m, n)` block of V. When constructed using `qr`, the block size is given by $n_b = \min(m, n, 36)$.

Iterating the decomposition produces the components `Q` and `R`.

The object has two fields:

- `factors`, as in the `QR` type, is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.

  - The subdiagonal part contains the reflectors v_i stored in a packed format such that `V = I + tril(F.factors, -1)`.
- T is a n_b-by-$\min(m,n)$ matrix as described above. The subdiagonal elements for each triangular matrix T_j are ignored.

Note

This format should not to be confused with the older *WY* representation [5].

`LinearAlgebra.QRPivoted` – Type.

```
QRPivoted <: Factorization
```

A QR matrix factorization with column pivoting in a packed format, typically obtained from `qr`. If A is an `m×n` matrix, then

$$AP = QR$$

where P is a permutation matrix, Q is an orthogonal/unitary matrix and R is upper triangular. The matrix Q is stored as a sequence of Householder reflectors:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

Iterating the decomposition produces the components `Q`, `R`, and `p`.

The object has three fields:

- `factors` is an `m×n` matrix.
  - The upper triangular part contains the elements of R, that is `R = triu(F.factors)` for a `QR` object `F`.
  - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the ith column of the matrix `V = I + tril(F.factors, -1)`.
- `τ` is a vector of length `min(m,n)` containing the coefficients au_i.
- `jpvt` is an integer vector of length `n` corresponding to the permutation P.

`LinearAlgebra.qr` – Function.

[5]C Bischof and C Van Loan, "The WY representation for products of Householder matrices", SIAM J Sci Stat Comput 8 (1987), s2-s13. doi:10.1137/0908009

[4]R Schreiber and C Van Loan, "A storage-efficient WY representation for products of Householder transformations", SIAM J Sci Stat Comput 10 (1989), 53-57. doi:10.1137/0910005

```
qr(A::SparseMatrixCSC; tol=_default_tol(A), ordering=ORDERING_DEFAULT) -> QRSparse
```

Compute the QR factorization of a sparse matrix A. Fill-reducing row and column permutations are used such that `F.R = F.Q'*A[F.prow,F.pcol]`. The main application of this type is to solve least squares or underdetermined problems with `\`. The function calls the C library SPQR[6].

> **Note**
>
> `qr(A::SparseMatrixCSC)` uses the SPQR library that is part of SuiteSparse. As this library only supports sparse matrices with `Float64` or `ComplexF64` elements, as of Julia v1.4 `qr` converts `A` into a copy that is of type `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{ComplexF64}` as appropriate.

Examples

```
julia> A = sparse([1,2,3,4], [1,1,2,2], [1.0,1.0,1.0,1.0])
4×2 SparseMatrixCSC{Float64, Int64} with 4 stored entries:
 1.0   ⋅
 1.0   ⋅
  ⋅   1.0
  ⋅   1.0

julia> qr(A)
SparseArrays.SPQR.QRSparse{Float64, Int64}
Q factor:
4×4 SparseArrays.SPQR.QRSparseQ{Float64, Int64}
R factor:
2×2 SparseMatrixCSC{Float64, Int64} with 2 stored entries:
 -1.41421    ⋅
   ⋅       -1.41421
Row permutation:
4-element Vector{Int64}:
 1
 3
 4
 2
Column permutation:
2-element Vector{Int64}:
 1
 2
```

source

```
qr(A, pivot = NoPivot(); blocksize) -> F
```

Compute the QR factorization of the matrix `A`: an orthogonal (or unitary if `A` is complex-valued) matrix `Q`, and an upper triangular matrix `R` such that

[6] Foster, L. V., & Davis, T. A. (2013). Algorithm 933: Reliable Calculation of Numerical Rank, Null Space Bases, Pseudoinverse Solutions, and Basic Solutions Using SuitesparseQR. ACM Trans. Math. Softw., 40(1). doi:10.1145/2513109.2513116

$$A = QR$$

The returned object F stores the factorization in a packed format:

- if `pivot == ColumnNorm()` then F is a `QRPivoted` object,
- otherwise if the element type of A is a BLAS type (`Float32`, `Float64`, `ComplexF32` or `ComplexF64`), then F is a `QRCompactWY` object,
- otherwise F is a `QR` object.

The individual components of the decomposition F can be retrieved via property accessors:

- `F.Q`: the orthogonal/unitary matrix `Q`
- `F.R`: the upper triangular matrix `R`
- `F.p`: the permutation vector of the pivot (`QRPivoted` only)
- `F.P`: the permutation matrix of the pivot (`QRPivoted` only)

Iterating the decomposition produces the components `Q`, `R`, and if extant `p`.

The following functions are available for the QR objects: `inv`, `size`, and `\`. When A is rectangular, `\` will return a least squares solution and if the solution is not unique, the one with smallest norm is returned. When A is not full rank, factorization with (column) pivoting is required to obtain a minimum norm solution.

Multiplication with respect to either full/square or non-full/square `Q` is allowed, i.e. both `F.Q*F.R` and `F.Q*A` are supported. A `Q` matrix can be converted into a regular matrix with `Matrix`. This operation returns the "thin" Q factor, i.e., if `A` is `m×n` with `m>=n`, then `Matrix(F.Q)` yields an `m×n` matrix with orthonormal columns. To retrieve the "full" Q factor, an `m×m` orthogonal matrix, use `F.Q*I` or `collect(F.Q)`. If `m<=n`, then `Matrix(F.Q)` yields an `m×m` orthogonal matrix.

The block size for QR decomposition can be specified by keyword argument `blocksize :: Integer` when `pivot == NoPivot()` and `A isa StridedMatrix{<:BlasFloat}`. It is ignored when `blocksize > minimum(size(A))`. See `QRCompactWY`.

Julia 1.4

The `blocksize` keyword argument requires Julia 1.4 or later.

Examples

```
julia> A = [3.0 -6.0; 4.0 -8.0; 0.0 1.0]
3×2 Matrix{Float64}:
 3.0  -6.0
 4.0  -8.0
 0.0   1.0

julia> F = qr(A)
LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
Q factor: 3×3 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
R factor:
2×2 Matrix{Float64}:
 -5.0  10.0
```

```
 0.0  -1.0

julia> F.Q * F.R == A
true
```

Note

qr returns multiple types because LAPACK uses several representations that minimize the memory storage requirements of products of Householder elementary reflectors, so that the Q and R matrices can be stored compactly rather as two separate dense matrices.

`LinearAlgebra.qr!` – Function.

```
qr!(A, pivot = NoPivot(); blocksize)
```

`qr!` is the same as `qr` when A is a subtype of `AbstractMatrix`, but saves space by overwriting the input A, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of A, e.g. for integer types.

Julia 1.4

The `blocksize` keyword argument requires Julia 1.4 or later.

Examples

```
julia> a = [1. 2.; 3. 4.]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> qr!(a)
LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
Q factor: 2×2 LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
R factor:
2×2 Matrix{Float64}:
 -3.16228  -4.42719
  0.0      -0.632456

julia> a = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> qr!(a)
ERROR: InexactError: Int64(3.1622776601683795)
Stacktrace:
[...]
```

`LinearAlgebra.LQ` – Type.

```
LQ <: Factorization
```

Matrix factorization type of the `LQ` factorization of a matrix `A`. The `LQ` decomposition is the `QR` decomposition of `transpose(A)`. This is the return type of `lq`, the corresponding matrix factorization function.

If `S::LQ` is the factorization object, the lower triangular component can be obtained via `S.L`, and the orthogonal/unitary component via `S.Q`, such that `A ≈ S.L*S.Q`.

Iterating the decomposition produces the components `S.L` and `S.Q`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> S = lq(A)
LQ{Float64, Matrix{Float64}, Vector{Float64}}
L factor:
2×2 Matrix{Float64}:
 -8.60233   0.0
  4.41741  -0.697486
Q factor: 2×2 LinearAlgebra.LQPackedQ{Float64, Matrix{Float64}, Vector{Float64}}

julia> S.L * S.Q
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> l, q = S; # destructuring via iteration

julia> l == S.L &&  q == S.Q
true
```

`LinearAlgebra.lq` – Function.

```
lq(A) -> S::LQ
```

Compute the LQ decomposition of `A`. The decomposition's lower triangular component can be obtained from the `LQ` object `S` via `S.L`, and the orthogonal/unitary component via `S.Q`, such that `A ≈ S.L*S.Q`.

Iterating the decomposition produces the components `S.L` and `S.Q`.

The LQ decomposition is the QR decomposition of `transpose(A)`, and it is useful in order to compute the minimum-norm solution `lq(A) \ b` to an underdetermined system of equations (`A` has more columns than rows, but has full row rank).

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Matrix{Float64}:
```

```
  5.0   7.0
 -2.0  -4.0

julia> S = lq(A)
LQ{Float64, Matrix{Float64}, Vector{Float64}}
L factor:
2×2 Matrix{Float64}:
 -8.60233   0.0
  4.41741  -0.697486
Q factor: 2×2 LinearAlgebra.LQPackedQ{Float64, Matrix{Float64}, Vector{Float64}}

julia> S.L * S.Q
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> l, q = S; # destructuring via iteration

julia> l == S.L &&  q == S.Q
true
```

`LinearAlgebra.lq!` – Function.

```
lq!(A) -> LQ
```

Compute the `LQ` factorization of `A`, using the input matrix as a workspace. See also `lq`.

`LinearAlgebra.BunchKaufman` – Type.

```
BunchKaufman <: Factorization
```

Matrix factorization type of the Bunch-Kaufman factorization of a symmetric or Hermitian matrix `A` as `P'UDU'P` or `P'LDL'P`, depending on whether the upper (the default) or the lower triangle is stored in `A`. If `A` is complex symmetric then `U'` and `L'` denote the unconjugated transposes, i.e. `transpose(U)` and `transpose(L)`, respectively. This is the return type of `bunchkaufman`, the corresponding matrix factorization function.

If `S::BunchKaufman` is the factorization object, the components can be obtained via `S.D`, `S.U` or `S.L` as appropriate given `S.uplo`, and `S.p`.

Iterating the decomposition produces the components `S.D`, `S.U` or `S.L` as appropriate given `S.uplo`, and `S.p`.

Examples

```
julia> A = [1 2; 2 3]
2×2 Matrix{Int64}:
 1  2
 2  3

julia> S = bunchkaufman(A) # A gets wrapped internally by Symmetric(A)
```

```
BunchKaufman{Float64, Matrix{Float64}, Vector{Int64}}
D factor:
2×2 Tridiagonal{Float64, Vector{Float64}}:
 -0.333333  0.0
  0.0       3.0
U factor:
2×2 UnitUpperTriangular{Float64, Matrix{Float64}}:
 1.0  0.666667
  ⋅   1.0
permutation:
2-element Vector{Int64}:
 1
 2

julia> d, u, p = S; # destructuring via iteration

julia> d == S.D && u == S.U && p == S.p
true

julia> S = bunchkaufman(Symmetric(A, :L))
BunchKaufman{Float64, Matrix{Float64}, Vector{Int64}}
D factor:
2×2 Tridiagonal{Float64, Vector{Float64}}:
 3.0   0.0
 0.0  -0.333333
L factor:
2×2 UnitLowerTriangular{Float64, Matrix{Float64}}:
 1.0        ⋅
 0.666667  1.0
permutation:
2-element Vector{Int64}:
 2
 1
```

`LinearAlgebra.bunchkaufman` – Function.

```
bunchkaufman(A, rook::Bool=false; check = true) -> S::BunchKaufman
```

Compute the Bunch-Kaufman [7] factorization of a symmetric or Hermitian matrix `A` as `P'*U*D*U'*P` or `P'*L*D*L'*P`, depending on which triangle is stored in `A`, and return a `BunchKaufman` object. Note that if `A` is complex symmetric then `U'` and `L'` denote the unconjugated transposes, i.e. `transpose(U)` and `transpose(L)`.

Iterating the decomposition produces the components `S.D`, `S.U` or `S.L` as appropriate given `S.uplo`, and `S.p`.

If `rook` is `true`, rook pivoting is used. If `rook` is false, rook pivoting is not used.

When `check = true`, an error is thrown if the decomposition fails. When `check = false`, responsibility for checking the decomposition's validity (via `issuccess`) lies with the user.

The following functions are available for BunchKaufman objects: `size`, \, `inv`, `issymmetric`, `ishermitian`, `getindex`.

Examples

```
julia> A = [1 2; 2 3]
2×2 Matrix{Int64}:
 1  2
 2  3

julia> S = bunchkaufman(A) # A gets wrapped internally by Symmetric(A)
BunchKaufman{Float64, Matrix{Float64}, Vector{Int64}}
D factor:
2×2 Tridiagonal{Float64, Vector{Float64}}:
 -0.333333  0.0
  0.0       3.0
U factor:
2×2 UnitUpperTriangular{Float64, Matrix{Float64}}:
 1.0  0.666667
  ⋅   1.0
permutation:
2-element Vector{Int64}:
 1
 2

julia> d, u, p = S; # destructuring via iteration

julia> d == S.D && u == S.U && p == S.p
true

julia> S.U*S.D*S.U' - S.P*A*S.P'
2×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0

julia> S = bunchkaufman(Symmetric(A, :L))
BunchKaufman{Float64, Matrix{Float64}, Vector{Int64}}
D factor:
2×2 Tridiagonal{Float64, Vector{Float64}}:
 3.0   0.0
 0.0  -0.333333
L factor:
2×2 UnitLowerTriangular{Float64, Matrix{Float64}}:
 1.0        ⋅
 0.666667  1.0
permutation:
2-element Vector{Int64}:
 2
 1

julia> S.L*S.D*S.L' - A[S.p, S.p]
2×2 Matrix{Float64}:
```

[7] J R Bunch and L Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Mathematics of Computation 31:137 (1977), 163-179. url.

```
 0.0  0.0
 0.0  0.0
```

`LinearAlgebra.bunchkaufman!` – Function.

```
bunchkaufman!(A, rook::Bool=false; check = true) -> BunchKaufman
```

`bunchkaufman!` is the same as `bunchkaufman`, but saves space by overwriting the input `A`, instead of creating a copy.

`LinearAlgebra.Eigen` – Type.

```
Eigen <: Factorization
```

Matrix factorization type of the eigenvalue/spectral decomposition of a square matrix `A`. This is the return type of `eigen`, the corresponding matrix factorization function.

If `F::Eigen` is the factorization object, the eigenvalues can be obtained via `F.values` and the eigenvectors as the columns of the matrix `F.vectors`. (The kth eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Examples

```
julia> F = eigen([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
  1.0
  3.0
 18.0
vectors:
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> F.values
3-element Vector{Float64}:
  1.0
  3.0
 18.0

julia> F.vectors
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> vals, vecs = F; # destructuring via iteration
```

```
julia> vals == F.values && vecs == F.vectors
true
```

`LinearAlgebra.GeneralizedEigen` – Type.

```
GeneralizedEigen <: Factorization
```

Matrix factorization type of the generalized eigenvalue/spectral decomposition of A and B. This is the return type of `eigen`, the corresponding matrix factorization function, when called with two matrix arguments.

If `F::GeneralizedEigen` is the factorization object, the eigenvalues can be obtained via `F.values` and the eigenvectors as the columns of the matrix `F.vectors`. (The kth eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Matrix{Int64}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Matrix{Int64}:
 0  1
 1  0

julia> F = eigen(A, B)
GeneralizedEigen{ComplexF64, ComplexF64, Matrix{ComplexF64}, Vector{ComplexF64}}
values:
2-element Vector{ComplexF64}:
 0.0 - 1.0im
 0.0 + 1.0im
vectors:
2×2 Matrix{ComplexF64}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> F.values
2-element Vector{ComplexF64}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> F.vectors
2×2 Matrix{ComplexF64}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

`LinearAlgebra.eigvals` – Function.

```
eigvals(A; permute::Bool=true, scale::Bool=true, sortby) -> values
```

Return the eigenvalues of `A`.

For general non-symmetric matrices it is possible to specify how the matrix is balanced before the eigenvalue calculation. The `permute`, `scale`, and `sortby` keywords are the same as for `eigen`.

Examples

```
julia> diag_matrix = [1 0; 0 4]
2×2 Matrix{Int64}:
 1  0
 0  4

julia> eigvals(diag_matrix)
2-element Vector{Float64}:
 1.0
 4.0
```

For a scalar input, `eigvals` will return a scalar.

Example

```
julia> eigvals(-2)
-2
```

```
eigvals(A, B) -> values
```

Compute the generalized eigenvalues of `A` and `B`.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Matrix{Int64}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Matrix{Int64}:
 0  1
 1  0

julia> eigvals(A,B)
2-element Vector{ComplexF64}:
 0.0 - 1.0im
 0.0 + 1.0im
```

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> values
```

Return the eigenvalues of `A`. It is possible to calculate only a subset of the eigenvalues by specifying a `UnitRange` `irange` covering indices of the sorted eigenvalues, e.g. the 2nd to 8th eigenvalues.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 1.0  2.0   ⋅
 2.0  2.0  3.0
  ⋅   3.0  1.0

julia> eigvals(A, 2:2)
1-element Vector{Float64}:
 0.9999999999999996

julia> eigvals(A)
3-element Vector{Float64}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259
```

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> values
```

Return the eigenvalues of `A`. It is possible to calculate only a subset of the eigenvalues by specifying a pair `vl` and `vu` for the lower and upper boundaries of the eigenvalues.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 1.0  2.0   ⋅
 2.0  2.0  3.0
  ⋅   3.0  1.0

julia> eigvals(A, -1, 2)
1-element Vector{Float64}:
 1.0000000000000009

julia> eigvals(A)
3-element Vector{Float64}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259
```

`LinearAlgebra.eigvals!` – Function.

```
eigvals!(A; permute::Bool=true, scale::Bool=true, sortby) -> values
```

Same as eigvals, but saves space by overwriting the input A, instead of creating a copy. The permute, scale, and sortby keywords are the same as for eigen.

Note

The input matrix A will not contain its eigenvalues after eigvals! is called on it - A is used as a workspace.

Examples

```
julia> A = [1. 2.; 3. 4.]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> eigvals!(A)
2-element Vector{Float64}:
 -0.3722813232690143
  5.372281323269014

julia> A
2×2 Matrix{Float64}:
 -0.372281  -1.0
  0.0        5.37228
```

```
eigvals!(A, B; sortby) -> values
```

Same as eigvals, but saves space by overwriting the input A (and B), instead of creating copies.

Note

The input matrices A and B will not contain their eigenvalues after eigvals! is called. They are used as workspaces.

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Matrix{Float64}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Matrix{Float64}:
 0.0  1.0
 1.0  0.0

julia> eigvals!(A, B)
2-element Vector{ComplexF64}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> A
```

```
2×2 Matrix{Float64}:
 -0.0  -1.0
  1.0  -0.0

julia> B
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

```
eigvals!(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. `irange` is a range of eigenvalue *indices* to search for - for instance, the 2nd to 8th eigenvalues.

```
eigvals!(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. `vl` is the lower bound of the interval to search for eigenvalues, and `vu` is the upper bound.

`LinearAlgebra.eigmax` – Function.

```
eigmax(A; permute::Bool=true, scale::Bool=true)
```

Return the largest eigenvalue of `A`. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of `A` are complex, this method will fail, since complex numbers cannot be sorted.

Examples

```
julia> A = [0 im; -im 0]
2×2 Matrix{Complex{Int64}}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmax(A)
1.0

julia> A = [0 im; -1 0]
2×2 Matrix{Complex{Int64}}:
  0+0im  0+1im
 -1+0im  0+0im

julia> eigmax(A)
ERROR: DomainError with Complex{Int64}[0+0im 0+1im; -1+0im 0+0im]:
`A` cannot have complex eigenvalues.
Stacktrace:
[...]
```

`LinearAlgebra.eigmin` – Function.

```
eigmin(A; permute::Bool=true, scale::Bool=true)
```

Return the smallest eigenvalue of `A`. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of `A` are complex, this method will fail, since complex numbers cannot be sorted.

Examples

```
julia> A = [0 im; -im 0]
2×2 Matrix{Complex{Int64}}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmin(A)
-1.0

julia> A = [0 im; -1 0]
2×2 Matrix{Complex{Int64}}:
  0+0im  0+1im
 -1+0im  0+0im

julia> eigmin(A)
ERROR: DomainError with Complex{Int64}[0+0im 0+1im; -1+0im 0+0im]:
`A` cannot have complex eigenvalues.
Stacktrace:
[...]
```

`LinearAlgebra.eigvecs` – Function.

```
eigvecs(A::SymTridiagonal[, eigvals]) -> Matrix
```

Return a matrix `M` whose columns are the eigenvectors of `A`. (The kth eigenvector can be obtained from the slice `M[:, k]`.)

If the optional vector of eigenvalues `eigvals` is specified, `eigvecs` returns the specific corresponding eigenvectors.

Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64, Vector{Float64}}:
 1.0  2.0   ⋅
 2.0  2.0  3.0
  ⋅   3.0  1.0

julia> eigvals(A)
3-element Vector{Float64}:
 -2.1400549446402604
```

```
  1.0000000000000002
  5.140054944640259

julia> eigvecs(A)
3×3 Matrix{Float64}:
  0.418304  -0.83205      0.364299
 -0.656749  -7.39009e-16  0.754109
  0.627457   0.5547       0.546448

julia> eigvecs(A, [1.])
3×1 Matrix{Float64}:
  0.8320502943378438
  4.263514128092366e-17
 -0.5547001962252291
```

```
eigvecs(A; permute::Bool=true, scale::Bool=true, `sortby`) -> Matrix
```

Return a matrix `M` whose columns are the eigenvectors of `A`. (The `k`th eigenvector can be obtained from the slice `M[:, k]`.) The `permute`, `scale`, and `sortby` keywords are the same as for `eigen`.

Examples

```
julia> eigvecs([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

```
eigvecs(A, B) -> Matrix
```

Return a matrix `M` whose columns are the generalized eigenvectors of `A` and `B`. (The `k`th eigenvector can be obtained from the slice `M[:, k]`.)

Examples

```
julia> A = [1 0; 0 -1]
2×2 Matrix{Int64}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Matrix{Int64}:
 0  1
 1  0

julia> eigvecs(A, B)
2×2 Matrix{ComplexF64}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im
```

`LinearAlgebra.eigen` – Function.

```
eigen(A; permute::Bool=true, scale::Bool=true, sortby) -> Eigen
```

Compute the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F.values` and the eigenvectors in the columns of the matrix `F.vectors`. This corresponds to solving an eigenvalue problem of the form `Ax = λx`, where `A` is a matrix, `x` is an eigenvector, and `λ` is an eigenvalue. (The kth eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

For general nonsymmetric matrices it is possible to specify how the matrix is balanced before the eigenvector calculation. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is `true` for both options.

By default, the eigenvalues and vectors are sorted lexicographically by `(real(λ),imag(λ))`. A different comparison function `by(λ)` can be passed to `sortby`, or you can pass `sortby=nothing` to leave the eigenvalues in an arbitrary order. Some special matrix types (e.g. `Diagonal` or `SymTridiagonal`) may implement their own sorting convention and not accept a `sortby` keyword.

Examples

```
julia> F = eigen([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
  1.0
  3.0
 18.0
vectors:
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> F.values
3-element Vector{Float64}:
  1.0
  3.0
 18.0

julia> F.vectors
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

```
eigen(A, B; sortby) -> GeneralizedEigen
```

Compute the generalized eigenvalue decomposition of A and B, returning a `GeneralizedEigen` factorization object F which contains the generalized eigenvalues in `F.values` and the generalized eigenvectors in the columns of the matrix `F.vectors`. This corresponds to solving a generalized eigenvalue problem of the form Ax = λBx, where `A, B` are matrices, x is an eigenvector, and λ is an eigenvalue. (The kth generalized eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

By default, the eigenvalues and vectors are sorted lexicographically by `(real(λ),imag(λ))`. A different comparison function `by(λ)` can be passed to `sortby`, or you can pass `sortby=nothing` to leave the eigenvalues in an arbitrary order.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Matrix{Int64}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Matrix{Int64}:
 0  1
 1  0

julia> F = eigen(A, B);

julia> F.values
2-element Vector{ComplexF64}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> F.vectors
2×2 Matrix{ComplexF64}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> vals, vecs = F; # destructuring via iteration

julia> vals == F.values && vecs == F.vectors
true
```

```
eigen(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> Eigen
```

Compute the eigenvalue decomposition of A, returning an `Eigen` factorization object F which contains the eigenvalues in `F.values` and the eigenvectors in the columns of the matrix `F.vectors`. (The kth eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

The `UnitRange` `irange` specifies indices of the sorted eigenvalues to search for.

> **Note**
>
> If `irange` is not `1:n`, where `n` is the dimension of `A`, then the returned factorization will be a *truncated* factorization.

```
eigen(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> Eigen
```

Compute the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F.values` and the eigenvectors in the columns of the matrix `F.vectors`. (The kth eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

`vl` is the lower bound of the window of eigenvalues to search for, and `vu` is the upper bound.

> **Note**
>
> If [`vl`, `vu`] does not contain all eigenvalues of `A`, then the returned factorization will be a *truncated* factorization.

`LinearAlgebra.eigen!` – Function.

```
eigen!(A; permute, scale, sortby)
eigen!(A, B; sortby)
```

Same as `eigen`, but saves space by overwriting the input `A` (and `B`), instead of creating a copy.

`LinearAlgebra.Hessenberg` – Type.

```
Hessenberg <: Factorization
```

A `Hessenberg` object represents the Hessenberg factorization `QHQ'` of a square matrix, or a shift `Q(H+μI)Q'` thereof, which is produced by the `hessenberg` function.

`LinearAlgebra.hessenberg` – Function.

```
hessenberg(A) -> Hessenberg
```

Compute the Hessenberg decomposition of `A` and return a `Hessenberg` object. If `F` is the factorization object, the unitary matrix can be accessed with `F.Q` (of type `LinearAlgebra.HessenbergQ`) and the Hessenberg matrix with `F.H` (of type `UpperHessenberg`), either of which may be converted to a regular matrix with `Matrix(F.H)` or `Matrix(F.Q)`.

If `A` is `Hermitian` or real-`Symmetric`, then the Hessenberg decomposition produces a real-symmetric tridiagonal matrix and `F.H` is of type `SymTridiagonal`.

Note that the shifted factorization `A+μI = Q (H+μI) Q'` can be constructed efficiently by `F + μ*I` using the `UniformScaling` object `I`, which creates a new `Hessenberg` object with shared storage and a modified shift. The shift of a given `F` is obtained by `F.μ`. This is useful because multiple shifted solves `(F + μ*I) \ b` (for different `μ` and/or `b`) can be performed efficiently once `F` is created.

Iterating the decomposition produces the factors `F.Q, F.H, F.μ`.

Examples

```
julia> A = [4. 9. 7.; 4. 4. 1.; 4. 3. 2.]
3×3 Matrix{Float64}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0

julia> F = hessenberg(A)
Hessenberg{Float64, UpperHessenberg{Float64, Matrix{Float64}}, Matrix{Float64}, Vector{Float64},
↪  Bool}
Q factor: 3×3 LinearAlgebra.HessenbergQ{Float64, Matrix{Float64}, Vector{Float64}, false}
H factor:
3×3 UpperHessenberg{Float64, Matrix{Float64}}:
  4.0      -11.3137       -1.41421
 -5.65685    5.0           2.0
   ⋅        -8.88178e-16   1.0

julia> F.Q * F.H * F.Q'
3×3 Matrix{Float64}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0

julia> q, h = F; # destructuring via iteration

julia> q == F.Q && h == F.H
true
```

`LinearAlgebra.hessenberg!` – Function.

```
hessenberg!(A) -> Hessenberg
```

`hessenberg!` is the same as `hessenberg`, but saves space by overwriting the input `A`, instead of creating a copy.

`LinearAlgebra.Schur` – Type.

```
Schur <: Factorization
```

Matrix factorization type of the Schur factorization of a matrix `A`. This is the return type of `schur(_)`, the corresponding matrix factorization function.

If `F::Schur` is the factorization object, the (quasi) triangular Schur factor can be obtained via either `F.Schur` or `F.T` and the orthogonal/unitary Schur vectors via `F.vectors` or `F.Z` such that `A = F.vectors * F.Schur * F.vectors'`. The eigenvalues of `A` can be obtained with `F.values`.

Iterating the decomposition produces the components F.T, F.Z, and F.values.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur(A)
Schur{Float64, Matrix{Float64}, Vector{Float64}}
T factor:
2×2 Matrix{Float64}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Matrix{Float64}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Vector{Float64}:
  3.0
 -2.0

julia> F.vectors * F.Schur * F.vectors'
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> t, z, vals = F; # destructuring via iteration

julia> t == F.T && z == F.Z && vals == F.values
true
```

LinearAlgebra.GeneralizedSchur – Type.

```
GeneralizedSchur <: Factorization
```

Matrix factorization type of the generalized Schur factorization of two matrices A and B. This is the return type of schur(_, _), the corresponding matrix factorization function.

If F::GeneralizedSchur is the factorization object, the (quasi) triangular Schur factors can be obtained via F.S and F.T, the left unitary/orthogonal Schur vectors via F.left or F.Q, and the right unitary/orthogonal Schur vectors can be obtained with F.right or F.Z such that A=F.left*F.S*F.right' and B=F.left*F.T*F.right'. The generalized eigenvalues of A and B can be obtained with F.α./F.β.

Iterating the decomposition produces the components F.S, F.T, F.Q, F.Z, F.α, and F.β.

LinearAlgebra.schur – Function.

```
schur(A) -> F::Schur
```

Computes the Schur factorization of the matrix `A`. The (quasi) triangular Schur factor can be obtained from the `Schur` object `F` with either `F.Schur` or `F.T` and the orthogonal/unitary Schur vectors can be obtained with `F.vectors` or `F.Z` such that `A = F.vectors * F.Schur * F.vectors'`. The eigenvalues of `A` can be obtained with `F.values`.

For real `A`, the Schur factorization is "quasitriangular", which means that it is upper-triangular except with 2×2 diagonal blocks for any conjugate pair of complex eigenvalues; this allows the factorization to be purely real even when there are complex eigenvalues. To obtain the (complex) purely upper-triangular Schur factorization from a real quasitriangular factorization, you can use `Schur{Complex}(schur(A))`.

Iterating the decomposition produces the components `F.T`, `F.Z`, and `F.values`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur(A)
Schur{Float64, Matrix{Float64}, Vector{Float64}}
T factor:
2×2 Matrix{Float64}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Matrix{Float64}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Vector{Float64}:
  3.0
 -2.0

julia> F.vectors * F.Schur * F.vectors'
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> t, z, vals = F; # destructuring via iteration

julia> t == F.T && z == F.Z && vals == F.values
true
```

```
schur(A, B) -> F::GeneralizedSchur
```

Computes the Generalized Schur (or QZ) factorization of the matrices `A` and `B`. The (quasi) triangular Schur factors can be obtained from the `Schur` object `F` with `F.S` and `F.T`, the left unitary/orthogonal Schur vectors can be obtained with `F.left` or `F.Q` and the right unitary/orthogonal Schur vectors can be obtained with `F.right` or `F.Z` such that `A=F.left*F.S*F.right'` and `B=F.left*F.T*F.right'`. The generalized eigenvalues of `A` and `B` can be obtained with `F.α./F.β`.

Iterating the decomposition produces the components `F.S`, `F.T`, `F.Q`, `F.Z`, `F.α`, and `F.β`.

`LinearAlgebra.schur!` – Function.

```
schur!(A) -> F::Schur
```

Same as `schur` but uses the input argument A as workspace.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Matrix{Float64}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur!(A)
Schur{Float64, Matrix{Float64}, Vector{Float64}}
T factor:
2×2 Matrix{Float64}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Matrix{Float64}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Vector{Float64}:
  3.0
 -2.0

julia> A
2×2 Matrix{Float64}:
 3.0   9.0
 0.0  -2.0
```

```
schur!(A::StridedMatrix, B::StridedMatrix) -> F::GeneralizedSchur
```

Same as `schur` but uses the input matrices A and B as workspace.

`LinearAlgebra.ordschur` – Function.

```
ordschur(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Reorders the Schur factorization F of a matrix `A = Z*T*Z'` according to the logical array `select` returning the reordered factorization F object. The selected eigenvalues appear in the leading diagonal of `F.Schur` and the corresponding leading columns of `F.vectors` form an orthogonal/unitary basis of the corresponding right invariant subspace. In the real case, a complex conjugate pair of eigenvalues must be either both included or both excluded via `select`.

```
ordschur(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::GeneralizedSchur
```

Reorders the Generalized Schur factorization F of a matrix pair `(A, B) = (Q*S*Z', Q*T*Z')` according to the logical array `select` and returns a GeneralizedSchur object F. The selected eigenvalues appear in

the leading diagonal of both `F.S` and `F.T`, and the left and right orthogonal/unitary Schur vectors are also reordered such that `(A, B) = F.Q*(F.S, F.T)*F.Z'` still holds and the generalized eigenvalues of `A` and `B` can still be obtained with `F.α./F.β`.

`LinearAlgebra.ordschur!` – Function.

```
ordschur!(F::Schur, select::Union{Vector{Bool},BitVector}) -> F::Schur
```

Same as `ordschur` but overwrites the factorization `F`.

```
ordschur!(F::GeneralizedSchur, select::Union{Vector{Bool},BitVector}) -> F::GeneralizedSchur
```

Same as `ordschur` but overwrites the factorization `F`.

`LinearAlgebra.SVD` – Type.

```
SVD <: Factorization
```

Matrix factorization type of the singular value decomposition (SVD) of a matrix `A`. This is the return type of `svd(_)`, the corresponding matrix factorization function.

If `F::SVD` is the factorization object, `U`, `S`, `V` and `Vt` can be obtained via `F.U`, `F.S`, `F.V` and `F.Vt`, such that `A = U * Diagonal(S) * Vt`. The singular values in `S` are sorted in descending order.

Iterating the decomposition produces the components `U`, `S`, and `V`.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Matrix{Float64}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> F = svd(A)
SVD{Float64, Float64, Matrix{Float64}, Vector{Float64}}
U factor:
4×4 Matrix{Float64}:
 0.0  1.0   0.0  0.0
 1.0  0.0   0.0  0.0
 0.0  0.0   0.0  1.0
 0.0  0.0  -1.0  0.0
singular values:
4-element Vector{Float64}:
 3.0
 2.23606797749979
 2.0
 0.0
Vt factor:
4×5 Matrix{Float64}:
 -0.0        0.0  1.0  -0.0  0.0
```

```
  0.447214   0.0  0.0   0.0  0.894427
  0.0       -1.0  0.0   0.0  0.0
  0.0        0.0  0.0   1.0  0.0

julia> F.U * Diagonal(F.S) * F.Vt
4×5 Matrix{Float64}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> u, s, v = F; # destructuring via iteration

julia> u == F.U && s == F.S && v == F.V
true
```

`LinearAlgebra.GeneralizedSVD` – Type.

```
GeneralizedSVD <: Factorization
```

Matrix factorization type of the generalized singular value decomposition (SVD) of two matrices `A` and `B`, such that `A = F.U*F.D1*F.R0*F.Q'` and `B = F.V*F.D2*F.R0*F.Q'`. This is the return type of `svd(_, _)`, the corresponding matrix factorization function.

For an M-by-N matrix `A` and P-by-N matrix `B`,

- `U` is a M-by-M orthogonal matrix,
- `V` is a P-by-P orthogonal matrix,
- `Q` is a N-by-N orthogonal matrix,
- `D1` is a M-by-(K+L) diagonal matrix with 1s in the first K entries,
- `D2` is a P-by-(K+L) matrix whose top right L-by-L block is diagonal,
- `R0` is a (K+L)-by-N matrix whose rightmost (K+L)-by-(K+L) block is nonsingular upper block triangular,

K+L is the effective numerical rank of the matrix `[A; B]`.

Iterating the decomposition produces the components `U`, `V`, `Q`, `D1`, `D2`, and `R0`.

The entries of `F.D1` and `F.D2` are related, as explained in the LAPACK documentation for the generalized SVD and the xGGSVD3 routine which is called underneath (in LAPACK 3.6.0 and newer).

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Matrix{Float64}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Matrix{Float64}:
 0.0  1.0
 1.0  0.0
```

```
julia> F = svd(A, B)
GeneralizedSVD{Float64, Matrix{Float64}, Float64, Vector{Float64}}
U factor:
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
V factor:
2×2 Matrix{Float64}:
 -0.0  -1.0
  1.0   0.0
Q factor:
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
D1 factor:
2×2 Matrix{Float64}:
 0.707107  0.0
 0.0       0.707107
D2 factor:
2×2 Matrix{Float64}:
 0.707107  0.0
 0.0       0.707107
R0 factor:
2×2 Matrix{Float64}:
 1.41421   0.0
 0.0      -1.41421

julia> F.U*F.D1*F.R0*F.Q'
2×2 Matrix{Float64}:
 1.0   0.0
 0.0  -1.0

julia> F.V*F.D2*F.R0*F.Q'
2×2 Matrix{Float64}:
 -0.0  1.0
  1.0  0.0
```

`LinearAlgebra.svd` – Function.

```
svd(A; full::Bool = false, alg::Algorithm = default_svd_alg(A)) -> SVD
```

Compute the singular value decomposition (SVD) of `A` and return an `SVD` object.

`U`, `S`, `V` and `Vt` can be obtained from the factorization `F` with `F.U`, `F.S`, `F.V` and `F.Vt`, such that `A = U * Diagonal(S) * Vt`. The algorithm produces `Vt` and hence `Vt` is more efficient to extract than `V`. The singular values in `S` are sorted in descending order.

Iterating the decomposition produces the components `U`, `S`, and `V`.

If `full = false` (default), a ”thin” SVD is returned. For an $M \times N$ matrix `A`, in the full factorization `U` is $M \times M$ and `V` is $N \times N$, while in the thin factorization `U` is $M \times K$ and `V` is $N \times K$, where $K = \min(M, N)$ is the number of singular values.

If `alg = DivideAndConquer()` a divide-and-conquer algorithm is used to calculate the SVD. Another (typically slower but more accurate) option is `alg = QRIteration()`.

Julia 1.3

The `alg` keyword argument requires Julia 1.3 or later.

Examples

```
julia> A = rand(4,3);

julia> F = svd(A); # Store the Factorization Object

julia> A ≈ F.U * Diagonal(F.S) * F.Vt
true

julia> U, S, V = F; # destructuring via iteration

julia> A ≈ U * Diagonal(S) * V'
true

julia> Uonly, = svd(A); # Store U only

julia> Uonly == U
true
```

```
svd(A, B) -> GeneralizedSVD
```

Compute the generalized SVD of `A` and `B`, returning a `GeneralizedSVD` factorization object `F` such that `[A;B] = [F.U * F.D1; F.V * F.D2] * F.R0 * F.Q'`

- `U` is a M-by-M orthogonal matrix,
- `V` is a P-by-P orthogonal matrix,
- `Q` is a N-by-N orthogonal matrix,
- `D1` is a M-by-(K+L) diagonal matrix with 1s in the first K entries,
- `D2` is a P-by-(K+L) matrix whose top right L-by-L block is diagonal,
- `R0` is a (K+L)-by-N matrix whose rightmost (K+L)-by-(K+L) block is nonsingular upper block triangular,

K+L is the effective numerical rank of the matrix `[A; B]`.

Iterating the decomposition produces the components `U`, `V`, `Q`, `D1`, `D2`, and `R0`.

The generalized SVD is used in applications such as when one wants to compare how much belongs to `A` vs. how much belongs to `B`, as in human vs yeast genome, or signal vs noise, or between clusters vs within clusters. (See Edelman and Wang for discussion: https://arxiv.org/abs/1901.00485)

It decomposes `[A; B]` into `[UC; VS]H`, where `[UC; VS]` is a natural orthogonal basis for the column space of `[A; B]`, and `H = RQ'` is a natural non-orthogonal basis for the rowspace of `[A;B]`, where the top rows are most closely attributed to the `A` matrix, and the bottom to the `B` matrix. The multi-cosine/sine matrices `C` and `S` provide a multi-measure of how much `A` vs how much `B`, and `U` and `V` provide directions in which these are measured.

Examples

```
julia> A = randn(3,2); B=randn(4,2);

julia> F = svd(A, B);

julia> U,V,Q,C,S,R = F;

julia> H = R*Q';

julia> [A; B] ≈ [U*C; V*S]*H
true

julia> [A; B] ≈ [F.U*F.D1; F.V*F.D2]*F.R0*F.Q'
true

julia> Uonly, = svd(A,B);

julia> U == Uonly
true
```

`LinearAlgebra.svd!` – Function.

```
svd!(A; full::Bool = false, alg::Algorithm = default_svd_alg(A)) -> SVD
```

`svd!` is the same as `svd`, but saves space by overwriting the input `A`, instead of creating a copy. See documentation of `svd` for details.

```
svd!(A, B) -> GeneralizedSVD
```

`svd!` is the same as `svd`, but modifies the arguments `A` and `B` in-place, instead of making copies. See documentation of `svd` for details.

`LinearAlgebra.svdvals` – Function.

```
svdvals(A)
```

Return the singular values of `A` in descending order.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Matrix{Float64}:
 1.0  0.0  0.0  0.0  2.0
 0.0  0.0  3.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  2.0  0.0  0.0  0.0

julia> svdvals(A)
4-element Vector{Float64}:
```

```
3.0
2.23606797749979
2.0
0.0
```

```
svdvals(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of A and B. See also svd.

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Matrix{Float64}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Matrix{Float64}:
 0.0  1.0
 1.0  0.0

julia> svdvals(A, B)
2-element Vector{Float64}:
 1.0
 1.0
```

LinearAlgebra.svdvals! – Function.

```
svdvals!(A)
```

Return the singular values of A, saving space by overwriting the input. See also svdvals and svd.

```
svdvals!(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of A and B, saving space by overwriting A and B. See also svd and svdvals.

LinearAlgebra.Givens – Type.

```
LinearAlgebra.Givens(i1,i2,c,s) -> G
```

A Givens rotation linear operator. The fields c and s represent the cosine and sine of the rotation angle, respectively. The Givens type supports left multiplication G*A and conjugated transpose right multiplication A*G'. The type doesn't have a size and can therefore be multiplied with matrices of arbitrary size as long as i2<=size(A,2) for G*A or i2<=size(A,1) for A*G'.

See also givens.

`LinearAlgebra.givens` – Function.

```
givens(f::T, g::T, i1::Integer, i2::Integer) where {T} -> (G::Givens, r::T)
```

Computes the Givens rotation `G` and scalar `r` such that for any vector `x` where

```
x[i1] = f
x[i2] = g
```

the result of the multiplication

```
y = G*x
```

has the property that

```
y[i1] = r
y[i2] = 0
```

See also `LinearAlgebra.Givens`.

```
givens(A::AbstractArray, i1::Integer, i2::Integer, j::Integer) -> (G::Givens, r)
```

Computes the Givens rotation `G` and scalar `r` such that the result of the multiplication

```
B = G*A
```

has the property that

```
B[i1,j] = r
B[i2,j] = 0
```

See also `LinearAlgebra.Givens`.

```
givens(x::AbstractVector, i1::Integer, i2::Integer) -> (G::Givens, r)
```

Computes the Givens rotation `G` and scalar `r` such that the result of the multiplication

```
B = G*x
```

has the property that

```
B[i1] = r
B[i2] = 0
```

See also `LinearAlgebra.Givens`.

`LinearAlgebra.triu` – Function.

```
triu(M)
```

Upper triangle of a matrix.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> triu(a)
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 0.0  1.0  1.0  1.0
 0.0  0.0  1.0  1.0
 0.0  0.0  0.0  1.0
```

```
triu(M, k::Integer)
```

Return the upper triangle of `M` starting from the kth superdiagonal.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> triu(a,3)
4×4 Matrix{Float64}:
 0.0  0.0  0.0  1.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0

julia> triu(a,-3)
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

`LinearAlgebra.triu!` – Function.

```
triu!(M)
```

Upper triangle of a matrix, overwriting `M` in the process. See also `triu`.

```
triu!(M, k::Integer)
```

Return the upper triangle of `M` starting from the kth superdiagonal, overwriting `M` in the process.

Examples

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5]
5×5 Matrix{Int64}:
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5

julia> triu!(M, 1)
5×5 Matrix{Int64}:
 0  2  3  4  5
 0  0  3  4  5
 0  0  0  4  5
 0  0  0  0  5
 0  0  0  0  0
```

`LinearAlgebra.tril` – Function.

```
tril(M)
```

Lower triangle of a matrix.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a)
4×4 Matrix{Float64}:
 1.0  0.0  0.0  0.0
 1.0  1.0  0.0  0.0
 1.0  1.0  1.0  0.0
 1.0  1.0  1.0  1.0
```

```
tril(M, k::Integer)
```

Return the lower triangle of `M` starting from the kth superdiagonal.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a,3)
4×4 Matrix{Float64}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0

julia> tril(a,-3)
4×4 Matrix{Float64}:
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0
```

`LinearAlgebra.tril!` – Function.

```
tril!(M)
```

Lower triangle of a matrix, overwriting `M` in the process. See also `tril`.

```
tril!(M, k::Integer)
```

Return the lower triangle of `M` starting from the kth superdiagonal, overwriting `M` in the process.

Examples

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5]
5×5 Matrix{Int64}:
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5

julia> tril!(M, 2)
5×5 Matrix{Int64}:
 1  2  3  0  0
```

```
 1  2  3  4  0
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
```

`LinearAlgebra.diagind` – Function.

```
diagind(M, k::Integer=0)
```

An `AbstractRange` giving the indices of the kth diagonal of the matrix `M`.

See also: `diag`, `diagm`, `Diagonal`.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> diagind(A,-1)
2:4:6
```

`LinearAlgebra.diag` – Function.

```
diag(M, k::Integer=0)
```

The kth diagonal of a matrix, as a vector.

See also `diagm`, `diagind`, `Diagonal`, `isdiag`.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> diag(A,1)
2-element Vector{Int64}:
 2
 6
```

`LinearAlgebra.diagm` – Function.

```
diagm(kv::Pair{<:Integer,<:AbstractVector}...)
diagm(m::Integer, n::Integer, kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a matrix from `Pairs` of diagonals and vectors. Vector `kv.second` will be placed on the `kv.first` diagonal. By default the matrix is square and its size is inferred from `kv`, but a non-square size `m×n` (padded with zeros as needed) can be specified by passing `m,n` as the first arguments. For repeated diagonal indices `kv.first` the values in the corresponding vectors `kv.second` will be added.

`diagm` constructs a full matrix; if you want storage-efficient versions with fast arithmetic, see `Diagonal`, `Bidiagonal` `Tridiagonal` and `SymTridiagonal`.

Examples

```
julia> diagm(1 => [1,2,3])
4×4 Matrix{Int64}:
 0  1  0  0
 0  0  2  0
 0  0  0  3
 0  0  0  0

julia> diagm(1 => [1,2,3], -1 => [4,5])
4×4 Matrix{Int64}:
 0  1  0  0
 4  0  2  0
 0  5  0  3
 0  0  0  0

julia> diagm(1 => [1,2,3], 1 => [1,2,3])
4×4 Matrix{Int64}:
 0  2  0  0
 0  0  4  0
 0  0  0  6
 0  0  0  0
```

```
diagm(v::AbstractVector)
diagm(m::Integer, n::Integer, v::AbstractVector)
```

Construct a matrix with elements of the vector as diagonal elements. By default, the matrix is square and its size is given by `length(v)`, but a non-square size `m×n` can be specified by passing `m,n` as the first arguments.

Examples

```
julia> diagm([1,2,3])
3×3 Matrix{Int64}:
 1  0  0
 0  2  0
 0  0  3
```

`LinearAlgebra.rank` – Function.

```
rank(::QRSparse{Tv,Ti}) -> Ti
```

Return the rank of the QR factorization

source

```
rank(S::SparseMatrixCSC{Tv,Ti}; [tol::Real]) -> Ti
```

Calculate rank of S by calculating its QR factorization. Values smaller than `tol` are considered as zero. See SPQR's manual.

source

```
rank(A::AbstractMatrix; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
rank(A::AbstractMatrix, rtol::Real)
```

Compute the numerical rank of a matrix by counting how many outputs of `svdvals(A)` are greater than `max(atol, rtol*σ₁)` where σ_1 is A's largest calculated singular value. `atol` and `rtol` are the absolute and relative tolerances, respectively. The default relative tolerance is `n*ϵ`, where `n` is the size of the smallest dimension of `A`, and `ϵ` is the eps of the element type of `A`.

> **Note**
>
> Numerical rank can be a sensitive and imprecise characterization of ill-conditioned matrices with singular values that are close to the threshold tolerance `max(atol, rtol*σ₁)`. In such cases, slight perturbations to the singular-value computation or to the matrix can change the result of rank by pushing one or more singular values across the threshold. These variations can even occur due to changes in floating-point errors between different Julia versions, architectures, compilers, or operating systems.

> **Julia 1.1**
>
> The `atol` and `rtol` keyword arguments requires at least Julia 1.1. In Julia 1.0 `rtol` is available as a positional argument, but this will be deprecated in Julia 2.0.

Examples

```
julia> rank(Matrix(I, 3, 3))
3

julia> rank(diagm(0 => [1, 0, 2]))
2

julia> rank(diagm(0 => [1, 0.001, 2]), rtol=0.1)
2

julia> rank(diagm(0 => [1, 0.001, 2]), rtol=0.00001)
3
```

```
julia> rank(diagm(0 => [1, 0.001, 2]), atol=1.5)
1
```

`LinearAlgebra.norm` – Function.

```
norm(A, p::Real=2)
```

For any iterable container `A` (including arrays of any dimension) of numbers (or any element type for which `norm` is defined), compute the p-norm (defaulting to p=2) as if `A` were a vector of the corresponding length.

The p-norm is defined as

$$\|A\|_p = \left(\sum_{i=1}^n |a_i|^p \right)^{1/p}$$

with a_i the entries of A, $|a_i|$ the `norm` of a_i, and n the length of A. Since the p-norm is computed using the `norm`s of the entries of `A`, the p-norm of a vector of vectors is not compatible with the interpretation of it as a block vector in general if `p != 2`.

`p` can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, `norm(A, Inf)` returns the largest value in `abs.(A)`, whereas `norm(A, -Inf)` returns the smallest. If `A` is a matrix and p=2, then this is equivalent to the Frobenius norm.

The second argument `p` is not necessarily a part of the interface for `norm`, i.e. a custom type may only implement `norm(A)` without second argument.

Use `opnorm` to compute the operator norm of a matrix.

Examples

```
julia> v = [3, -2, 6]
3-element Vector{Int64}:
  3
 -2
  6

julia> norm(v)
7.0

julia> norm(v, 1)
11.0

julia> norm(v, Inf)
6.0

julia> norm([1 2 3; 4 5 6; 7 8 9])
16.881943016134134

julia> norm([1 2 3 4 5 6 7 8 9])
16.881943016134134
```

```
julia> norm(1:9)
16.881943016134134

julia> norm(hcat(v,v), 1) == norm(vcat(v,v), 1) != norm([v,v], 1)
true

julia> norm(hcat(v,v), 2) == norm(vcat(v,v), 2) == norm([v,v], 2)
true

julia> norm(hcat(v,v), Inf) == norm(vcat(v,v), Inf) != norm([v,v], Inf)
true
```

```
norm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p\right)^{1/p}$.

Examples

```
julia> norm(2, 1)
2.0

julia> norm(-2, 1)
2.0

julia> norm(2, 2)
2.0

julia> norm(-2, 2)
2.0

julia> norm(2, Inf)
2.0

julia> norm(-2, Inf)
2.0
```

`LinearAlgebra.opnorm` – Function.

```
opnorm(A::AbstractMatrix, p::Real=2)
```

Compute the operator norm (or matrix norm) induced by the vector p-norm, where valid values of p are 1, 2, or `Inf`. (Note that for sparse matrices, p=2 is currently not implemented.) Use `norm` to compute the Frobenius norm.

When p=1, the operator norm is the maximum absolute column sum of A:

$$\|A\|_1 = \max_{1\ j\ n} \sum_{i=1}^{m} |a_{ij}|$$

with a_{ij} the entries of A, and m and n its dimensions.

When p=2, the operator norm is the spectral norm, equal to the largest singular value of A.

When p=`Inf`, the operator norm is the maximum absolute row sum of A:

$$\|A\|_\infty = \max_{1\ i\ m} \sum_{j=1}^{n} |a_{ij}|$$

Examples

```
julia> A = [1 -2 -3; 2 3 -1]
2×3 Matrix{Int64}:
 1  -2  -3
 2   3  -1

julia> opnorm(A, Inf)
6.0

julia> opnorm(A, 1)
5.0
```

```
opnorm(x::Number, p::Real=2)
```

For numbers, return $\left(|x|^p\right)^{1/p}$. This is equivalent to `norm`.

```
opnorm(A::Adjoint{<:Any,<:AbstracVector}, q::Real=2)
opnorm(A::Transpose{<:Any,<:AbstracVector}, q::Real=2)
```

For Adjoint/Transpose-wrapped vectors, return the operator q-norm of A, which is equivalent to the p-norm with value `p = q/(q-1)`. They coincide at `p = q = 2`. Use `norm` to compute the p norm of A as a vector.

The difference in norm between a vector space and its dual arises to preserve the relationship between duality and the dot product, and the result is consistent with the operator p-norm of a `1 × n` matrix.

Examples

```
julia> v = [1; im];

julia> vc = v';

julia> opnorm(vc, 1)
1.0

julia> norm(vc, 1)
2.0

julia> norm(v, 1)
2.0

julia> opnorm(vc, 2)
```

```
1.4142135623730951

julia> norm(vc, 2)
1.4142135623730951

julia> norm(v, 2)
1.4142135623730951

julia> opnorm(vc, Inf)
2.0

julia> norm(vc, Inf)
1.0

julia> norm(v, Inf)
1.0
```

`LinearAlgebra.normalize!` – Function.

```
normalize!(a::AbstractArray, p::Real=2)
```

Normalize the array a in-place so that its p-norm equals unity, i.e. `norm(a, p) == 1`. See also `normalize` and `norm`.

`LinearAlgebra.normalize` – Function.

```
normalize(a, p::Real=2)
```

Normalize a so that its p-norm equals unity, i.e. `norm(a, p) == 1`. For scalars, this is similar to sign(a), except normalize(0) = NaN. See also `normalize!`, `norm`, and `sign`.

Examples

```
julia> a = [1,2,4];

julia> b = normalize(a)
3-element Vector{Float64}:
 0.2182178902359924
 0.4364357804719848
 0.8728715609439696

julia> norm(b)
1.0

julia> c = normalize(a, 1)
3-element Vector{Float64}:
 0.14285714285714285
 0.2857142857142857
 0.5714285714285714

julia> norm(c, 1)
```

```
1.0

julia> a = [1 2 4 ; 1 2 4]
2×3 Matrix{Int64}:
 1  2  4
 1  2  4

julia> norm(a)
6.48074069840786

julia> normalize(a)
2×3 Matrix{Float64}:
 0.154303  0.308607  0.617213
 0.154303  0.308607  0.617213

julia> normalize(3, 1)
1.0

julia> normalize(-8, 1)
-1.0

julia> normalize(0, 1)
NaN
```

`LinearAlgebra.cond` – Function.

```
cond(M, p::Real=2)
```

Condition number of the matrix `M`, computed using the operator `p`-norm. Valid values for `p` are `1`, `2` (default), or `Inf`.

`LinearAlgebra.condskeel` – Function.

```
condskeel(M, [x, p::Real=Inf])
```

$$\kappa_S(M,p) = \left\Vert \left\vert M \right\vert \left\vert M^{-1} \right\vert \right\Vert_p$$
$$\kappa_S(M,x,p) = \frac{\left\Vert \left\vert M \right\vert \left\vert M^{-1} \right\vert \left\vert x \right\vert \right\Vert_p}{\left\Vert x \right\Vert_p}$$

Skeel condition number κ_S of the matrix `M`, optionally with respect to the vector `x`, as computed using the operator `p`-norm. $\left\vert M \right\vert$ denotes the matrix of (entry wise) absolute values of M; $\left\vert M \right\vert_{ij} = \left\vert M_{ij} \right\vert$. Valid values for `p` are `1`, `2` and `Inf` (default).

This quantity is also known in the literature as the Bauer condition number, relative condition number, or componentwise relative condition number.

`LinearAlgebra.tr` – Function.

```
tr(M)
```

Matrix trace. Sums the diagonal elements of `M`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> tr(A)
5
```

`LinearAlgebra.det` – Function.

```
det(M)
```

Matrix determinant.

See also: `logdet` and `logabsdet`.

Examples

```
julia> M = [1 0; 2 2]
2×2 Matrix{Int64}:
 1  0
 2  2

julia> det(M)
2.0
```

`LinearAlgebra.logdet` – Function.

```
logdet(M)
```

Log of matrix determinant. Equivalent to `log(det(M))`, but may provide increased accuracy and/or speed.

Examples

```
julia> M = [1 0; 2 2]
2×2 Matrix{Int64}:
 1  0
 2  2

julia> logdet(M)
0.6931471805599453

julia> logdet(Matrix(I, 3, 3))
0.0
```

`LinearAlgebra.logabsdet` – Function.

```
logabsdet(M)
```

Log of absolute value of matrix determinant. Equivalent to `(log(abs(det(M))), sign(det(M)))`, but may provide increased accuracy and/or speed.

Examples

```
julia> A = [-1. 0.; 0. 1.]
2×2 Matrix{Float64}:
 -1.0  0.0
  0.0  1.0

julia> det(A)
-1.0

julia> logabsdet(A)
(0.0, -1.0)

julia> B = [2. 0.; 0. 1.]
2×2 Matrix{Float64}:
 2.0  0.0
 0.0  1.0

julia> det(B)
2.0

julia> logabsdet(B)
(0.6931471805599453, 1.0)
```

`Base.inv` – Method.

```
inv(M)
```

Matrix inverse. Computes matrix `N` such that `M * N = I`, where `I` is the identity matrix. Computed by solving the left-division `N = M \ I`.

Examples

```
julia> M = [2 5; 1 3]
2×2 Matrix{Int64}:
 2  5
 1  3

julia> N = inv(M)
2×2 Matrix{Float64}:
  3.0  -5.0
 -1.0   2.0

julia> M*N == N*M == Matrix(I, 2, 2)
true
```

`LinearAlgebra.pinv` – Function.

```
pinv(M; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
pinv(M, rtol::Real) = pinv(M; rtol=rtol) # to be deprecated in Julia 2.0
```

Computes the Moore-Penrose pseudoinverse.

For matrices `M` with floating point elements, it is convenient to compute the pseudoinverse by inverting only singular values greater than `max(atol, rtol*σ₁)` where `σ₁` is the largest singular value of `M`.

The optimal choice of absolute (`atol`) and relative tolerance (`rtol`) varies both with the value of `M` and the intended application of the pseudoinverse. The default relative tolerance is `n*ϵ`, where `n` is the size of the smallest dimension of `M`, and `ϵ` is the eps of the element type of `M`.

For inverting dense ill-conditioned matrices in a least-squares sense, `rtol = sqrt(eps(real(float(oneunit(eltype(M))))))` is recommended.

For more information, see [8], [9], [10], [11].

Examples

```
julia> M = [1.5 1.3; 1.2 1.9]
2×2 Matrix{Float64}:
 1.5  1.3
 1.2  1.9

julia> N = pinv(M)
2×2 Matrix{Float64}:
  1.47287   -1.00775
 -0.930233   1.16279

julia> M * N
2×2 Matrix{Float64}:
 1.0          -2.22045e-16
 4.44089e-16   1.0
```

`LinearAlgebra.nullspace` – Function.

```
nullspace(M; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
nullspace(M, rtol::Real) = nullspace(M; rtol=rtol) # to be deprecated in Julia 2.0
```

Computes a basis for the nullspace of `M` by including the singular vectors of `M` whose singular values have magnitudes smaller than `max(atol, rtol*σ₁)`, where `σ₁` is `M`'s largest singular value.

[8] Issue 8859, "Fix least squares", https://github.com/JuliaLang/julia/pull/8859

[9] Åke Björck, "Numerical Methods for Least Squares Problems", SIAM Press, Philadelphia, 1996, "Other Titles in Applied Mathematics", Vol. 51. doi:10.1137/1.9781611971484

[10] G. W. Stewart, "Rank Degeneracy", SIAM Journal on Scientific and Statistical Computing, 5(2), 1984, 403-413. doi:10.1137/0905030

[11] Konstantinos Konstantinides and Kung Yao, "Statistical analysis of effective singular values in matrix rank determination", IEEE Transactions on Acoustics, Speech and Signal Processing, 36(5), 1988, 757-763. doi:10.1109/29.1585

By default, the relative tolerance `rtol` is `n*ϵ`, where `n` is the size of the smallest dimension of `M`, and `ϵ` is the `eps` of the element type of `M`.

Examples

```
julia> M = [1 0 0; 0 1 0; 0 0 0]
3×3 Matrix{Int64}:
 1  0  0
 0  1  0
 0  0  0

julia> nullspace(M)
3×1 Matrix{Float64}:
 0.0
 0.0
 1.0

julia> nullspace(M, rtol=3)
3×3 Matrix{Float64}:
 0.0  1.0  0.0
 1.0  0.0  0.0
 0.0  0.0  1.0

julia> nullspace(M, atol=0.95)
3×1 Matrix{Float64}:
 0.0
 0.0
 1.0
```

`Base.kron` – Function.

```
kron(A, B)
```

Computes the Kronecker product of two vectors, matrices or numbers.

For real vectors `v` and `w`, the Kronecker product is related to the outer product by `kron(v,w) == vec(w * transpose(v))` or `w * transpose(v) == reshape(kron(v,w), (length(w), length(v)))`. Note how the ordering of `v` and `w` differs on the left and right of these expressions (due to column-major storage). For complex vectors, the outer product `w * v'` also differs by conjugation of `v`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> B = [im 1; 1 -im]
2×2 Matrix{Complex{Int64}}:
 0+1im  1+0im
 1+0im  0-1im

julia> kron(A, B)
4×4 Matrix{Complex{Int64}}:
```

```
 0+1im  1+0im  0+2im  2+0im
 1+0im  0-1im  2+0im  0-2im
 0+3im  3+0im  0+4im  4+0im
 3+0im  0-3im  4+0im  0-4im

julia> v = [1, 2]; w = [3, 4, 5];

julia> w*transpose(v)
3×2 Matrix{Int64}:
 3   6
 4   8
 5  10

julia> reshape(kron(v,w), (length(w), length(v)))
3×2 Matrix{Int64}:
 3   6
 4   8
 5  10
```

`Base.kron!` – Function.

```
kron!(C, A, B)
```

Computes the Kronecker product of `A` and `B` and stores the result in `C`, overwriting the existing content of `C`. This is the in-place version of `kron`.

> **Julia 1.6**
>
> This function requires Julia 1.6 or later.

`Base.exp` – Method.

```
exp(A::AbstractMatrix)
```

Compute the matrix exponential of `A`, defined by

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

For symmetric or Hermitian `A`, an eigendecomposition (`eigen`) is used, otherwise the scaling and squaring algorithm (see [12]) is chosen.

Examples

[12] Nicholas J. Higham, "The squaring and scaling method for the matrix exponential revisited", SIAM Journal on Matrix Analysis and Applications, 26(4), 2005, 1179-1193. doi:10.1137/090768539

```
julia> A = Matrix(1.0I, 2, 2)
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0

julia> exp(A)
2×2 Matrix{Float64}:
 2.71828  0.0
 0.0      2.71828
```

`Base.cis` – Method.

```
cis(A::AbstractMatrix)
```

More efficient method for `exp(im*A)` of square matrix `A` (especially if `A` is `Hermitian` or real-`Symmetric`).

See also `cispi`, `sincos`, `exp`.

> **Julia 1.7**
>
> Support for using `cis` with matrices was added in Julia 1.7.

Examples

```
julia> cis([π 0; 0 π]) ≈ -I
true
```

`Base.:^` – Method.

```
^(A::AbstractMatrix, p::Number)
```

Matrix power, equivalent to $\exp(p \log(A))$

Examples

```
julia> [1 2; 0 3]^3
2×2 Matrix{Int64}:
 1  26
 0  27
```

`Base.:^` – Method.

```
^(b::Number, A::AbstractMatrix)
```

Matrix exponential, equivalent to $\exp(\log(b)A)$.

Julia 1.1

Support for raising `Irrational` numbers (like ℯ) to a matrix was added in Julia 1.1.

Examples

```
julia> 2^[1 2; 0 3]
2×2 Matrix{Float64}:
 2.0  6.0
 0.0  8.0

julia> ℯ^[1 2; 0 3]
2×2 Matrix{Float64}:
 2.71828  17.3673
 0.0      20.0855
```

`Base.log` – Method.

```
log(A::AbstractMatrix)
```

If `A` has no negative real eigenvalue, compute the principal matrix logarithm of `A`, i.e. the unique matrix X such that $e^X = A$ and $-\pi < Im(\lambda) < \pi$ for all the eigenvalues λ of X. If `A` has nonpositive eigenvalues, a nonprincipal matrix function is returned whenever possible.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used, if `A` is triangular an improved version of the inverse scaling and squaring method is employed (see [13] and [14]). If `A` is real with no negative eigenvalues, then the real Schur form is computed. Otherwise, the complex Schur form is computed. Then the upper (quasi-)triangular algorithm in [14] is used on the upper (quasi-)triangular factor.

Examples

```
julia> A = Matrix(2.7182818*I, 2, 2)
2×2 Matrix{Float64}:
 2.71828  0.0
 0.0      2.71828

julia> log(A)
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0
```

`Base.sqrt` – Method.

[13] Awad H. Al-Mohy and Nicholas J. Higham, "Improved inverse scaling and squaring algorithms for the matrix logarithm", SIAM Journal on Scientific Computing, 34(4), 2012, C153-C169. doi:10.1137/110852553

[14] Awad H. Al-Mohy, Nicholas J. Higham and Samuel D. Relton, "Computing the Fréchet derivative of the matrix logarithm and estimating the condition number", SIAM Journal on Scientific Computing, 35(4), 2013, C394-C410. doi:10.1137/120885991

```
sqrt(x)
```

Return $\sqrt{x}$. Throws DomainError for negative Real arguments. Use complex negative arguments instead. The prefix operator √ is equivalent to sqrt.

See also: hypot.

Examples

```
julia> sqrt(big(81))
9.0

julia> sqrt(big(-81))
ERROR: DomainError with -81.0:
NaN result for non-NaN input.
Stacktrace:
 [1] sqrt(::BigFloat) at ./mpfr.jl:501
[...]

julia> sqrt(big(complex(-81)))
0.0 + 9.0im

julia> .√(1:4)
4-element Vector{Float64}:
 1.0
 1.4142135623730951
 1.7320508075688772
 2.0
```

source

```
sqrt(A::AbstractMatrix)
```

If A has no negative real eigenvalues, compute the principal matrix square root of A, that is the unique matrix X with eigenvalues having positive real part such that $X^2 = A$. Otherwise, a nonprincipal square root is returned.

If A is real-symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the square root. For such matrices, eigenvalues λ that appear to be slightly negative due to roundoff errors are treated as if they were zero. More precisely, matrices with all eigenvalues ≥ -rtol*(max |λ|) are treated as semidefinite (yielding a Hermitian square root), with negative eigenvalues taken to be zero. rtol is a keyword argument to sqrt (in the Hermitian/real-symmetric case only) that defaults to machine precision scaled by size(A,1).

Otherwise, the square root is determined by means of the Björck-Hammarling method [15], which computes the complex Schur form (schur) and then the complex square root of the triangular factor. If a real square root exists, then an extension of this method [16] that computes the real Schur form and then the real square root of the quasi-triangular factor is instead used.

[15] Åke Björck and Sven Hammarling, "A Schur method for the square root of a matrix", Linear Algebra and its Applications, 52-53, 1983, 127-140. doi:10.1016/0024-3795(83)80010-X

[16] Nicholas J. Higham, "Computing real square roots of a real matrix", Linear Algebra and its Applications, 88-89, 1987, 405-430. doi:10.1016/0024-3795(87)90118-2

Examples

```
julia> A = [4 0; 0 4]
2×2 Matrix{Int64}:
 4  0
 0  4

julia> sqrt(A)
2×2 Matrix{Float64}:
 2.0  0.0
 0.0  2.0
```

`Base.cos` – Method.

```
cos(A::AbstractMatrix)
```

Compute the matrix cosine of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the cosine. Otherwise, the cosine is determined by calling `exp`.

Examples

```
julia> cos(fill(1.0, (2,2)))
2×2 Matrix{Float64}:
  0.291927  -0.708073
 -0.708073   0.291927
```

`Base.sin` – Method.

```
sin(A::AbstractMatrix)
```

Compute the matrix sine of a square matrix `A`.

If `A` is symmetric or Hermitian, its eigendecomposition (`eigen`) is used to compute the sine. Otherwise, the sine is determined by calling `exp`.

Examples

```
julia> sin(fill(1.0, (2,2)))
2×2 Matrix{Float64}:
 0.454649  0.454649
 0.454649  0.454649
```

`Base.Math.sincos` – Method.

```
sincos(A::AbstractMatrix)
```

Compute the matrix sine and cosine of a square matrix A.

Examples

```
julia> S, C = sincos(fill(1.0, (2,2)));

julia> S
2×2 Matrix{Float64}:
 0.454649  0.454649
 0.454649  0.454649

julia> C
2×2 Matrix{Float64}:
  0.291927  -0.708073
 -0.708073   0.291927
```

Base.tan – Method.

```
tan(A::AbstractMatrix)
```

Compute the matrix tangent of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the tangent. Otherwise, the tangent is determined by calling exp.

Examples

```
julia> tan(fill(1.0, (2,2)))
2×2 Matrix{Float64}:
 -1.09252  -1.09252
 -1.09252  -1.09252
```

Base.Math.sec – Method.

```
sec(A::AbstractMatrix)
```

Compute the matrix secant of a square matrix A.

Base.Math.csc – Method.

```
csc(A::AbstractMatrix)
```

Compute the matrix cosecant of a square matrix A.

Base.Math.cot – Method.

```
cot(A::AbstractMatrix)
```

Compute the matrix cotangent of a square matrix A.

`Base.cosh` – Method.

```
cosh(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosine of a square matrix `A`.

`Base.sinh` – Method.

```
sinh(A::AbstractMatrix)
```

Compute the matrix hyperbolic sine of a square matrix `A`.

`Base.tanh` – Method.

```
tanh(A::AbstractMatrix)
```

Compute the matrix hyperbolic tangent of a square matrix `A`.

`Base.Math.sech` – Method.

```
sech(A::AbstractMatrix)
```

Compute the matrix hyperbolic secant of square matrix `A`.

`Base.Math.csch` – Method.

```
csch(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosecant of square matrix `A`.

`Base.Math.coth` – Method.

```
coth(A::AbstractMatrix)
```

Compute the matrix hyperbolic cotangent of square matrix `A`.

`Base.acos` – Method.

```
acos(A::AbstractMatrix)
```

Compute the inverse matrix cosine of a square matrix `A`.

If A is symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the inverse cosine. Otherwise, the inverse cosine is determined by using log and sqrt. For the theory and logarithmic formulas used to compute this function, see [17].

Examples

```
julia> acos(cos([0.5 0.1; -0.2 0.3]))
2×2 Matrix{ComplexF64}:
  0.5-8.32667e-17im  0.1+0.0im
 -0.2+2.63678e-16im  0.3-3.46945e-16im
```

Base.asin – Method.

```
asin(A::AbstractMatrix)
```

Compute the inverse matrix sine of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the inverse sine. Otherwise, the inverse sine is determined by using log and sqrt. For the theory and logarithmic formulas used to compute this function, see [18].

Examples

```
julia> asin(sin([0.5 0.1; -0.2 0.3]))
2×2 Matrix{ComplexF64}:
  0.5-4.16334e-17im  0.1-5.55112e-17im
 -0.2+9.71445e-17im  0.3-1.249e-16im
```

Base.atan – Method.

```
atan(A::AbstractMatrix)
```

Compute the inverse matrix tangent of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the inverse tangent. Otherwise, the inverse tangent is determined by using log. For the theory and logarithmic formulas used to compute this function, see [19].

Examples

[17] Mary Aprahamian and Nicholas J. Higham, ”Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms”, MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

[18] Mary Aprahamian and Nicholas J. Higham, ”Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms”, MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

[19] Mary Aprahamian and Nicholas J. Higham, ”Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms”, MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

```
julia> atan(tan([0.5 0.1; -0.2 0.3]))
2×2 Matrix{ComplexF64}:
  0.5+1.38778e-17im  0.1-2.77556e-17im
 -0.2+6.93889e-17im  0.3-4.16334e-17im
```

`Base.Math.asec` – Method.

```
asec(A::AbstractMatrix)
```

Compute the inverse matrix secant of A.

`Base.Math.acsc` – Method.

```
acsc(A::AbstractMatrix)
```

Compute the inverse matrix cosecant of A.

`Base.Math.acot` – Method.

```
acot(A::AbstractMatrix)
```

Compute the inverse matrix cotangent of A.

`Base.acosh` – Method.

```
acosh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix cosine of a square matrix A. For the theory and logarithmic formulas used to compute this function, see [20].

`Base.asinh` – Method.

```
asinh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix sine of a square matrix A. For the theory and logarithmic formulas used to compute this function, see [21].

`Base.atanh` – Method.

[20] Mary Aprahamian and Nicholas J. Higham, ”Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms”, MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

[21] Mary Aprahamian and Nicholas J. Higham, ”Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms”, MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

```
atanh(A::AbstractMatrix)
```

Compute the inverse hyperbolic matrix tangent of a square matrix A. For the theory and logarithmic formulas used to compute this function, see [22].

`Base.Math.asech` – Method.

```
asech(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic secant of A.

`Base.Math.acsch` – Method.

```
acsch(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cosecant of A.

`Base.Math.acoth` – Method.

```
acoth(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cotangent of A.

`LinearAlgebra.lyap` – Function.

```
lyap(A, C)
```

Computes the solution X to the continuous Lyapunov equation `AX + XA' + C = 0`, where no eigenvalue of A has a zero real part and no two eigenvalues are negative complex conjugates of each other.

Examples

```
julia> A = [3. 4.; 5. 6]
2×2 Matrix{Float64}:
 3.0  4.0
 5.0  6.0

julia> B = [1. 1.; 1. 2.]
2×2 Matrix{Float64}:
 1.0  1.0
 1.0  2.0

julia> X = lyap(A, B)
```

[22] Mary Aprahamian and Nicholas J. Higham, ”Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms”, MIMS EPrint: 2016.4. https://doi.org/10.1137/16M1057577

```
2×2 Matrix{Float64}:
  0.5  -0.5
 -0.5   0.25

julia> A*X + X*A' ≈ -B
true
```

`LinearAlgebra.sylvester` – Function.

```
sylvester(A, B, C)
```

Computes the solution `X` to the Sylvester equation `AX + XB + C = 0`, where `A`, `B` and `C` have compatible dimensions and `A` and `-B` have no eigenvalues with equal real part.

Examples

```
julia> A = [3. 4.; 5. 6]
2×2 Matrix{Float64}:
 3.0  4.0
 5.0  6.0

julia> B = [1. 1.; 1. 2.]
2×2 Matrix{Float64}:
 1.0  1.0
 1.0  2.0

julia> C = [1. 2.; -2. 1]
2×2 Matrix{Float64}:
  1.0  2.0
 -2.0  1.0

julia> X = sylvester(A, B, C)
2×2 Matrix{Float64}:
 -4.46667   1.93333
  3.73333  -1.8

julia> A*X + X*B ≈ -C
true
```

`LinearAlgebra.issuccess` – Function.

```
issuccess(F::Factorization)
```

Test that a factorization of a matrix succeeded.

Julia 1.6

`issuccess(::CholeskyPivoted)` requires Julia 1.6 or later.

```
julia> F = cholesky([1 0; 0 1]);

julia> issuccess(F)
true

julia> F = lu([1 0; 0 0]; check = false);

julia> issuccess(F)
false
```

`LinearAlgebra.issymmetric` – Function.

```
issymmetric(A) -> Bool
```

Test whether a matrix is symmetric.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Matrix{Int64}:
 1   2
 2  -1

julia> issymmetric(a)
true

julia> b = [1 im; -im 1]
2×2 Matrix{Complex{Int64}}:
 1+0im  0+1im
 0-1im  1+0im

julia> issymmetric(b)
false
```

`LinearAlgebra.isposdef` – Function.

```
isposdef(A) -> Bool
```

Test whether a matrix is positive definite (and Hermitian) by trying to perform a Cholesky factorization of A.

See also `isposdef!`, `cholesky`.

Examples

```
julia> A = [1 2; 2 50]
2×2 Matrix{Int64}:
 1   2
 2  50
```

```
julia> isposdef(A)
true
```

`LinearAlgebra.isposdef!` – Function.

```
isposdef!(A) -> Bool
```

Test whether a matrix is positive definite (and Hermitian) by trying to perform a Cholesky factorization of `A`, overwriting `A` in the process. See also `isposdef`.

Examples

```
julia> A = [1. 2.; 2. 50.];

julia> isposdef!(A)
true

julia> A
2×2 Matrix{Float64}:
 1.0  2.0
 2.0  6.78233
```

`LinearAlgebra.istril` – Function.

```
istril(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether `A` is lower triangular starting from the `k`th superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Matrix{Int64}:
 1   2
 2  -1

julia> istril(a)
false

julia> istril(a, 1)
true

julia> b = [1 0; -im -1]
2×2 Matrix{Complex{Int64}}:
 1+0im   0+0im
 0-1im  -1+0im

julia> istril(b)
true

julia> istril(b, -1)
false
```

`LinearAlgebra.istriu` – Function.

```
istriu(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether A is upper triangular starting from the kth superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Matrix{Int64}:
 1   2
 2  -1

julia> istriu(a)
false

julia> istriu(a, -1)
true

julia> b = [1 im; 0 -1]
2×2 Matrix{Complex{Int64}}:
 1+0im   0+1im
 0+0im  -1+0im

julia> istriu(b)
true

julia> istriu(b, 1)
false
```

`LinearAlgebra.isdiag` – Function.

```
isdiag(A) -> Bool
```

Test whether a matrix is diagonal in the sense that `iszero(A[i,j])` is true unless `i == j`. Note that it is not necessary for A to be square; if you would also like to check that, you need to check that `size(A, 1) == size(A, 2)`.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Matrix{Int64}:
 1   2
 2  -1

julia> isdiag(a)
false

julia> b = [im 0; 0 -im]
2×2 Matrix{Complex{Int64}}:
 0+1im  0+0im
```

```
 0+0im  0-1im

julia> isdiag(b)
true

julia> c = [1 0 0; 0 2 0]
2×3 Matrix{Int64}:
 1  0  0
 0  2  0

julia> isdiag(c)
true

julia> d = [1 0 0; 0 2 3]
2×3 Matrix{Int64}:
 1  0  0
 0  2  3

julia> isdiag(d)
false
```

`LinearAlgebra.ishermitian` – Function.

```
ishermitian(A) -> Bool
```

Test whether a matrix is Hermitian.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Matrix{Int64}:
 1   2
 2  -1

julia> ishermitian(a)
true

julia> b = [1 im; -im 1]
2×2 Matrix{Complex{Int64}}:
 1+0im  0+1im
 0-1im  1+0im

julia> ishermitian(b)
true
```

`Base.transpose` – Function.

```
transpose(A)
```

Lazy transpose. Mutating the returned object should appropriately mutate `A`. Often, but not always, yields `Transpose(A)`, where `Transpose` is a lazy transpose wrapper. Note that this operation is recursive.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`, which is non-recursive.

Examples

```
julia> A = [3 2; 0 0]
2×2 Matrix{Int64}:
 3  2
 0  0

julia> B = transpose(A)
2×2 transpose(::Matrix{Int64}) with eltype Int64:
 3  0
 2  0

julia> B isa Transpose
true

julia> transpose(B) === A # the transpose of a transpose unwraps the parent
true

julia> Transpose(B) # however, the constructor always wraps its argument
2×2 transpose(transpose(::Matrix{Int64})) with eltype Int64:
 3  2
 0  0

julia> B[1,2] = 4; # modifying B will modify A automatically

julia> A
2×2 Matrix{Int64}:
 3  2
 4  0
```

For complex matrices, the `adjoint` operation is equivalent to a conjugate-transpose.

```
julia> A = reshape([Complex(x, x) for x in 1:4], 2, 2)
2×2 Matrix{Complex{Int64}}:
 1+1im  3+3im
 2+2im  4+4im

julia> adjoint(A) == conj(transpose(A))
true
```

The `transpose` of an `AbstractVector` is a row-vector:

```
julia> v = [1,2,3]
3-element Vector{Int64}:
 1
 2
 3

julia> transpose(v) # returns a row-vector
1×3 transpose(::Vector{Int64}) with eltype Int64:
 1  2  3
```

```
julia> transpose(v) * v # compute the dot product
14
```

For a matrix of matrices, the individual blocks are recursively operated on:

```
julia> C = [1 3; 2 4]
2×2 Matrix{Int64}:
 1  3
 2  4

julia> D = reshape([C, 2C, 3C, 4C], 2, 2) # construct a block matrix
2×2 Matrix{Matrix{Int64}}:
 [1 3; 2 4]  [3 9; 6 12]
 [2 6; 4 8]  [4 12; 8 16]

julia> transpose(D) # blocks are recursively transposed
2×2 transpose(::Matrix{Matrix{Int64}}) with eltype Transpose{Int64, Matrix{Int64}}:
 [1 2; 3 4]   [2 4; 6 8]
 [3 6; 9 12]  [4 8; 12 16]
```

```
transpose(F::Factorization)
```

Lazy transpose of the factorization F. By default, returns a `TransposeFactorization`, except for `Factorization`s with real `eltype`, in which case returns an `AdjointFactorization`.

`LinearAlgebra.transpose!` – Function.

```
transpose!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{Tv,Ti}) where {Tv,Ti}
```

Transpose the matrix A and stores it in the matrix X. `size(X)` must be equal to `size(transpose(A))`. No additional memory is allocated other than resizing the rowval and nzval of X, if needed.

See `halfperm!`

source

```
transpose!(dest,src)
```

Transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2),size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 8+7im  4+6im
```

```
julia> B = zeros(Complex{Int64}, 2, 2)
2×2 Matrix{Complex{Int64}}:
 0+0im  0+0im
 0+0im  0+0im

julia> transpose!(B, A);

julia> B
2×2 Matrix{Complex{Int64}}:
 3+2im  8+7im
 9+2im  4+6im

julia> A
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 8+7im  4+6im
```

`LinearAlgebra.Transpose` – Type.

```
Transpose
```

Lazy wrapper type for a transpose view of the underlying linear algebra object, usually an `AbstractVector`/`AbstractMatrix`. Usually, the `Transpose` constructor should not be called directly, use `transpose` instead. To materialize the view use `copy`.

This type is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [2 3; 0 0]
2×2 Matrix{Int64}:
 2  3
 0  0

julia> Transpose(A)
2×2 transpose(::Matrix{Int64}) with eltype Int64:
 2  0
 3  0
```

`LinearAlgebra.TransposeFactorization` – Type.

```
TransposeFactorization
```

Lazy wrapper type for the transpose of the underlying `Factorization` object. Usually, the `TransposeFactorization` constructor should not be called directly, use `transpose(:: Factorization)` instead.

`Base.adjoint` – Function.

```
A'
adjoint(A)
```

Lazy adjoint (conjugate transposition). Note that `adjoint` is applied recursively to elements.

For number types, `adjoint` returns the complex conjugate, and therefore it is equivalent to the identity function for real numbers.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 0  0]
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 0+0im  0+0im

julia> B = A' # equivalently adjoint(A)
2×2 adjoint(::Matrix{Complex{Int64}}) with eltype Complex{Int64}:
 3-2im  0+0im
 9-2im  0+0im

julia> B isa Adjoint
true

julia> adjoint(B) === A # the adjoint of an adjoint unwraps the parent
true

julia> Adjoint(B) # however, the constructor always wraps its argument
2×2 adjoint(adjoint(::Matrix{Complex{Int64}})) with eltype Complex{Int64}:
 3+2im  9+2im
 0+0im  0+0im

julia> B[1,2] = 4 + 5im; # modifying B will modify A automatically

julia> A
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 4-5im  0+0im
```

For real matrices, the `adjoint` operation is equivalent to a `transpose`.

```
julia> A = reshape([x for x in 1:4], 2, 2)
2×2 Matrix{Int64}:
 1  3
 2  4

julia> A'
2×2 adjoint(::Matrix{Int64}) with eltype Int64:
 1  2
 3  4

julia> adjoint(A) == transpose(A)
true
```

The adjoint of an `AbstractVector` is a row-vector:

```
julia> x = [3, 4im]
2-element Vector{Complex{Int64}}:
 3 + 0im
 0 + 4im

julia> x'
1×2 adjoint(::Vector{Complex{Int64}}) with eltype Complex{Int64}:
 3+0im  0-4im

julia> x'x # compute the dot product, equivalently x' * x
25 + 0im
```

For a matrix of matrices, the individual blocks are recursively operated on:

```
julia> A = reshape([x + im*x for x in 1:4], 2, 2)
2×2 Matrix{Complex{Int64}}:
 1+1im  3+3im
 2+2im  4+4im

julia> C = reshape([A, 2A, 3A, 4A], 2, 2)
2×2 Matrix{Matrix{Complex{Int64}}}:
 [1+1im 3+3im; 2+2im 4+4im]  [3+3im 9+9im; 6+6im 12+12im]
 [2+2im 6+6im; 4+4im 8+8im]  [4+4im 12+12im; 8+8im 16+16im]

julia> C'
2×2 adjoint(::Matrix{Matrix{Complex{Int64}}}) with eltype Adjoint{Complex{Int64},
↪  Matrix{Complex{Int64}}}:
 [1-1im 2-2im; 3-3im 4-4im]    [2-2im 4-4im; 6-6im 8-8im]
 [3-3im 6-6im; 9-9im 12-12im]  [4-4im 8-8im; 12-12im 16-16im]
```

```
adjoint(F::Factorization)
```

Lazy adjoint of the factorization F. By default, returns an `AdjointFactorization` wrapper.

`LinearAlgebra.adjoint!` – Function.

```
adjoint!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{Tv,Ti}) where {Tv,Ti}
```

Transpose the matrix A and stores the adjoint of the elements in the matrix X. `size(X)` must be equal to `size(transpose(A))`. No additional memory is allocated other than resizing the rowval and nzval of X, if needed.

See `halfperm!`

source

```
adjoint!(dest,src)
```

Conjugate transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2),size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 8+7im  4+6im

julia> B = zeros(Complex{Int64}, 2, 2)
2×2 Matrix{Complex{Int64}}:
 0+0im  0+0im
 0+0im  0+0im

julia> adjoint!(B, A);

julia> B
2×2 Matrix{Complex{Int64}}:
 3-2im  8-7im
 9-2im  4-6im

julia> A
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 8+7im  4+6im
```

`LinearAlgebra.Adjoint` – Type.

```
Adjoint
```

Lazy wrapper type for an adjoint view of the underlying linear algebra object, usually an `AbstractVector`/`AbstractMatrix`. Usually, the `Adjoint` constructor should not be called directly, use `adjoint` instead. To materialize the view use `copy`.

This type is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 0 0]
2×2 Matrix{Complex{Int64}}:
 3+2im  9+2im
 0+0im  0+0im

julia> Adjoint(A)
2×2 adjoint(::Matrix{Complex{Int64}}) with eltype Complex{Int64}:
 3-2im  0+0im
 9-2im  0+0im
```

`LinearAlgebra.AdjointFactorization` – Type.

```
AdjointFactorization
```

Lazy wrapper type for the adjoint of the underlying `Factorization` object. Usually, the `AdjointFactorization` constructor should not be called directly, use `adjoint(:: Factorization)` instead.

`Base.copy` – Method.

```
copy(A::Transpose)
copy(A::Adjoint)
```

Eagerly evaluate the lazy matrix transpose/adjoint. Note that the transposition is applied recursively to elements.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`, which is non-recursive.

Examples

```
julia> A = [1 2im; -3im 4]
2×2 Matrix{Complex{Int64}}:
 1+0im  0+2im
 0-3im  4+0im

julia> T = transpose(A)
2×2 transpose(::Matrix{Complex{Int64}}) with eltype Complex{Int64}:
 1+0im  0-3im
 0+2im  4+0im

julia> copy(T)
2×2 Matrix{Complex{Int64}}:
 1+0im  0-3im
 0+2im  4+0im
```

`LinearAlgebra.stride1` – Function.

```
stride1(A) -> Int
```

Return the distance between successive array elements in dimension 1 in units of element size.

Examples

```
julia> A = [1,2,3,4]
4-element Vector{Int64}:
 1
 2
 3
 4

julia> LinearAlgebra.stride1(A)
```

```
1

julia> B = view(A, 2:2:4)
2-element view(::Vector{Int64}, 2:2:4) with eltype Int64:
 2
 4

julia> LinearAlgebra.stride1(B)
2
```

`LinearAlgebra.checksquare` – Function.

```
LinearAlgebra.checksquare(A)
```

Check that a matrix is square, then return its common dimension. For multiple arguments, return a vector.

Examples

```
julia> A = fill(1, (4,4)); B = fill(1, (5,5));

julia> LinearAlgebra.checksquare(A, B)
2-element Vector{Int64}:
 4
 5
```

`LinearAlgebra.peakflops` – Function.

```
LinearAlgebra.peakflops(n::Integer=4096; eltype::DataType=Float64, ntrials::Integer=3,
↪  parallel::Bool=false)
```

`peakflops` computes the peak flop rate of the computer by using double precision `gemm!`. By default, if no arguments are specified, it multiplies two `Float64` matrices of size `n x n`, where `n = 4096`. If the underlying BLAS is using multiple threads, higher flop rates are realized. The number of BLAS threads can be set with `BLAS.set_num_threads(n)`.

If the keyword argument `eltype` is provided, `peakflops` will construct matrices with elements of type `eltype` for calculating the peak flop rate.

By default, `peakflops` will use the best timing from 3 trials. If the `ntrials` keyword argument is provided, `peakflops` will use those many trials for picking the best timing.

If the keyword argument `parallel` is set to `true`, `peakflops` is run in parallel on all the worker processors. The flop rate of the entire parallel computer is returned. When running in parallel, only 1 BLAS thread is used. The argument `n` still refers to the size of the problem that is solved on each processor.

Julia 1.1

This function requires at least Julia 1.1. In Julia 1.0 it is available from the standard library `InteractiveUtils`.

`LinearAlgebra.hermitianpart` – Function.

```
hermitianpart(A::AbstractMatrix, uplo::Symbol=:U) -> Hermitian
```

Return the Hermitian part of the square matrix `A`, defined as `(A + A') / 2`, as a `Hermitian` matrix. For real matrices `A`, this is also known as the symmetric part of `A`; it is also sometimes called the "operator real part". The optional argument `uplo` controls the corresponding argument of the `Hermitian` view. For real matrices, the latter is equivalent to a `Symmetric` view.

See also `hermitianpart!` for the corresponding in-place operation.

Julia 1.10

This function requires Julia 1.10 or later.

`LinearAlgebra.hermitianpart!` – Function.

```
hermitianpart!(A::AbstractMatrix, uplo::Symbol=:U) -> Hermitian
```

Overwrite the square matrix `A` in-place with its Hermitian part `(A + A') / 2`, and return `Hermitian(A, uplo)`. For real matrices `A`, this is also known as the symmetric part of `A`.

See also `hermitianpart` for the corresponding out-of-place operation.

Julia 1.10

This function requires Julia 1.10 or later.

79.5 Low-level matrix operations

In many cases there are in-place versions of matrix operations that allow you to supply a pre-allocated output vector or matrix. This is useful when optimizing critical code in order to avoid the overhead of repeated allocations. These in-place operations are suffixed with `!` below (e.g. `mul!`) according to the usual Julia convention.

`LinearAlgebra.mul!` – Function.

```
mul!(Y, A, B) -> Y
```

Calculates the matrix-matrix or matrix-vector product AB and stores the result in `Y`, overwriting the existing value of `Y`. Note that `Y` must not be aliased with either `A` or `B`.

Examples

```
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; Y = similar(B); mul!(Y, A, B);

julia> Y
2×2 Matrix{Float64}:
 3.0  3.0
 7.0  7.0
```

Implementation

For custom matrix and vector types, it is recommended to implement 5-argument `mul!` rather than implementing 3-argument `mul!` directly if possible.

```
mul!(C, A, B, α, β) -> C
```

Combined inplace matrix-matrix or matrix-vector multiply-add $AB + C$. The result is stored in `C` by overwriting it. Note that `C` must not be aliased with either `A` or `B`.

Julia 1.3

Five-argument `mul!` requires at least Julia 1.3.

Examples

```
julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0]; C=[1.0 2.0; 3.0 4.0];

julia> mul!(C, A, B, 100.0, 10.0) === C
true

julia> C
2×2 Matrix{Float64}:
 310.0  320.0
 730.0  740.0
```

`LinearAlgebra.lmul!` – Function.

```
lmul!(a::Number, B::AbstractArray)
```

Scale an array `B` by a scalar `a` overwriting `B` in-place. Use `rmul!` to multiply scalar from right. The scaling operation respects the semantics of the multiplication `*` between `a` and an element of `B`. In particular, this also applies to multiplication involving non-finite numbers such as `NaN` and `±Inf`.

Julia 1.1

Prior to Julia 1.1, `NaN` and `±Inf` entries in `B` were treated inconsistently.

Examples

```
julia> B = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> lmul!(2, B)
2×2 Matrix{Int64}:
 2  4
 6  8
```

```
julia> lmul!(0.0, [Inf])
1-element Vector{Float64}:
 NaN
```

```
lmul!(A, B)
```

Calculate the matrix-matrix product AB, overwriting B, and return the result. Here, A must be of special matrix type, like, e.g., `Diagonal`, `UpperTriangular` or `LowerTriangular`, or of some orthogonal type, see `QR`.

Examples

```
julia> B = [0 1; 1 0];

julia> A = UpperTriangular([1 2; 0 3]);

julia> lmul!(A, B);

julia> B
2×2 Matrix{Int64}:
 2  1
 3  0

julia> B = [1.0 2.0; 3.0 4.0];

julia> F = qr([0 1; -1 0]);

julia> lmul!(F.Q, B)
2×2 Matrix{Float64}:
 3.0  4.0
 1.0  2.0
```

`LinearAlgebra.rmul!` – Function.

```
rmul!(A::AbstractArray, b::Number)
```

Scale an array A by a scalar b overwriting A in-place. Use `lmul!` to multiply scalar from left. The scaling operation respects the semantics of the multiplication `*` between an element of A and b. In particular, this also applies to multiplication involving non-finite numbers such as NaN and `±Inf`.

> **Julia 1.1**
>
> Prior to Julia 1.1, NaN and `±Inf` entries in A were treated inconsistently.

Examples

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
```

```
 3  4

julia> rmul!(A, 2)
2×2 Matrix{Int64}:
 2  4
 6  8

julia> rmul!([NaN], 0.0)
1-element Vector{Float64}:
 NaN
```

```
rmul!(A, B)
```

Calculate the matrix-matrix product AB, overwriting A, and return the result. Here, B must be of special matrix type, like, e.g., Diagonal, UpperTriangular or LowerTriangular, or of some orthogonal type, see QR.

Examples

```
julia> A = [0 1; 1 0];

julia> B = UpperTriangular([1 2; 0 3]);

julia> rmul!(A, B);

julia> A
2×2 Matrix{Int64}:
 0  3
 1  2

julia> A = [1.0 2.0; 3.0 4.0];

julia> F = qr([0 1; -1 0]);

julia> rmul!(A, F.Q)
2×2 Matrix{Float64}:
 2.0  1.0
 4.0  3.0
```

`LinearAlgebra.ldiv!` – Function.

```
ldiv!(Y, A, B) -> Y
```

Compute `A \ B` in-place and store the result in Y, returning the result.

The argument A should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholesky`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lu!`), and performance-critical situations requiring `ldiv!` usually also require fine-grained control over the factorization of A.

Note

Certain structured matrix types, such as `Diagonal` and `UpperTriangular`, are permitted, as these are already in a factorized form

Examples

```
julia> A = [1 2.2 4; 3.1 0.2 3; 4 1 2];

julia> X = [1; 2.5; 3];

julia> Y = zero(X);

julia> ldiv!(Y, qr(A), X);

julia> Y ≈ A\X
true
```

```
ldiv!(A, B)
```

Compute `A \ B` in-place and overwriting `B` to store the result.

The argument `A` should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholesky`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lu!`), and performance-critical situations requiring `ldiv!` usually also require fine-grained control over the factorization of `A`.

Note

Certain structured matrix types, such as `Diagonal` and `UpperTriangular`, are permitted, as these are already in a factorized form

Examples

```
julia> A = [1 2.2 4; 3.1 0.2 3; 4 1 2];

julia> X = [1; 2.5; 3];

julia> Y = copy(X);

julia> ldiv!(qr(A), X);

julia> X ≈ A\Y
true
```

```
ldiv!(a::Number, B::AbstractArray)
```

Divide each entry in an array `B` by a scalar `a` overwriting `B` in-place. Use `rdiv!` to divide scalar from right.

Examples

```
julia> B = [1.0 2.0; 3.0 4.0]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> ldiv!(2.0, B)
2×2 Matrix{Float64}:
 0.5  1.0
 1.5  2.0
```

`LinearAlgebra.rdiv!` – Function.

```
rdiv!(A, B)
```

Compute `A / B` in-place and overwriting `A` to store the result.

The argument `B` should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `cholesky`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lu!`), and performance-critical situations requiring `rdiv!` usually also require fine-grained control over the factorization of `B`.

Note

Certain structured matrix types, such as `Diagonal` and `UpperTriangular`, are permitted, as these are already in a factorized form

```
rdiv!(A::AbstractArray, b::Number)
```

Divide each entry in an array `A` by a scalar `b` overwriting `A` in-place. Use `ldiv!` to divide scalar from left.

Examples

```
julia> A = [1.0 2.0; 3.0 4.0]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0

julia> rdiv!(A, 2.0)
2×2 Matrix{Float64}:
 0.5  1.0
 1.5  2.0
```

79.6 BLAS functions

In Julia (as in much of scientific computation), dense linear-algebra operations are based on the LAPACK library, which in turn is built on top of basic linear-algebra building-blocks known as the BLAS. There are highly optimized implementations of BLAS available for every computer architecture, and sometimes in high-performance linear algebra routines it is useful to call the BLAS functions directly.

LinearAlgebra.BLAS provides wrappers for some of the BLAS functions. Those BLAS functions that overwrite one of the input arrays have names ending in '!'. Usually, a BLAS function has four methods defined, for Float32, Float64, ComplexF32, and ComplexF64 arrays.

BLAS character arguments

Many BLAS functions accept arguments that determine whether to transpose an argument (trans), which triangle of a matrix to reference (uplo or ul), whether the diagonal of a triangular matrix can be assumed to be all ones (dA) or which side of a matrix multiplication the input argument belongs on (side). The possibilities are:

Multiplication order

side	Meaning
'L'	The argument goes on the *left* side of a matrix-matrix operation.
'R'	The argument goes on the *right* side of a matrix-matrix operation.

Triangle referencing

uplo/ul	Meaning
'U'	Only the *upper* triangle of the matrix will be used.
'L'	Only the *lower* triangle of the matrix will be used.

Transposition operation

trans/tX	Meaning
'N'	The input matrix X is not transposed or conjugated.
'T'	The input matrix X will be transposed.
'C'	The input matrix X will be conjugated and transposed.

Unit diagonal

diag/dX	Meaning
'N'	The diagonal values of the matrix X will be read.
'U'	The diagonal of the matrix X is assumed to be all ones.

LinearAlgebra.BLAS – Module.

Interface to BLAS subroutines.

LinearAlgebra.BLAS.set_num_threads – Function.

```
set_num_threads(n::Integer)
set_num_threads(::Nothing)
```

Set the number of threads the BLAS library should use equal to `n::Integer`.

Also accepts `nothing`, in which case julia tries to guess the default number of threads. Passing `nothing` is discouraged and mainly exists for historical reasons.

`LinearAlgebra.BLAS.get_num_threads` – Function.

```
get_num_threads()
```

Get the number of threads the BLAS library is using.

Julia 1.6

`get_num_threads` requires at least Julia 1.6.

BLAS functions can be divided into three groups, also called three levels, depending on when they were first proposed, the type of input parameters, and the complexity of the operation.

Level 1 BLAS functions

The level 1 BLAS functions were first proposed in [(Lawson, 1979)][Lawson-1979] and define operations between scalars and vectors.

[Lawson-1979]: https://dl.acm.org/doi/10.1145/355841.355847

`LinearAlgebra.BLAS.rot!` – Function.

```
rot!(n, X, incx, Y, incy, c, s)
```

Overwrite X with `c*X + s*Y` and Y with `-conj(s)*X + c*Y` for the first n elements of array X with stride `incx` and first n elements of array Y with stride `incy`. Returns X and Y.

Julia 1.5

`rot!` requires at least Julia 1.5.

`LinearAlgebra.BLAS.scal!` – Function.

```
scal!(n, a, X, incx)
scal!(a, X)
```

Overwrite X with a*X for the first n elements of array X with stride `incx`. Returns X.

If n and `incx` are not provided, `length(X)` and `stride(X,1)` are used.

`LinearAlgebra.BLAS.scal` – Function.

```
scal(n, a, X, incx)
scal(a, X)
```

Return X scaled by a for the first n elements of array X with stride `incx`.

If n and `incx` are not provided, `length(X)` and `stride(X,1)` are used.

`LinearAlgebra.BLAS.blascopy!` – Function.

```
blascopy!(n, X, incx, Y, incy)
```

Copy n elements of array X with stride `incx` to array Y with stride `incy`. Returns Y.

`LinearAlgebra.BLAS.dot` – Function.

```
dot(n, X, incx, Y, incy)
```

Dot product of two vectors consisting of n elements of array X with stride `incx` and n elements of array Y with stride `incy`.

Examples

```
julia> BLAS.dot(10, fill(1.0, 10), 1, fill(1.0, 20), 2)
10.0
```

`LinearAlgebra.BLAS.dotu` – Function.

```
dotu(n, X, incx, Y, incy)
```

Dot function for two complex vectors consisting of n elements of array X with stride `incx` and n elements of array Y with stride `incy`.

Examples

```
julia> BLAS.dotu(10, fill(1.0im, 10), 1, fill(1.0+im, 20), 2)
-10.0 + 10.0im
```

`LinearAlgebra.BLAS.dotc` – Function.

```
dotc(n, X, incx, U, incy)
```

Dot function for two complex vectors, consisting of n elements of array X with stride `incx` and n elements of array U with stride `incy`, conjugating the first vector.

Examples

```
julia> BLAS.dotc(10, fill(1.0im, 10), 1, fill(1.0+im, 20), 2)
10.0 - 10.0im
```

`LinearAlgebra.BLAS.nrm2` – Function.

```
nrm2(n, X, incx)
```

2-norm of a vector consisting of `n` elements of array `X` with stride `incx`.

Examples

```
julia> BLAS.nrm2(4, fill(1.0, 8), 2)
2.0

julia> BLAS.nrm2(1, fill(1.0, 8), 2)
1.0
```

`LinearAlgebra.BLAS.asum` – Function.

```
asum(n, X, incx)
```

Sum of the magnitudes of the first `n` elements of array `X` with stride `incx`.

For a real array, the magnitude is the absolute value. For a complex array, the magnitude is the sum of the absolute value of the real part and the absolute value of the imaginary part.

Examples

```
julia> BLAS.asum(5, fill(1.0im, 10), 2)
5.0

julia> BLAS.asum(2, fill(1.0im, 10), 5)
2.0
```

`LinearAlgebra.BLAS.iamax` – Function.

```
iamax(n, dx, incx)
iamax(dx)
```

Find the index of the element of `dx` with the maximum absolute value. `n` is the length of `dx`, and `incx` is the stride. If `n` and `incx` are not provided, they assume default values of `n=length(dx)` and `incx=stride1(dx)`.

Level 2 BLAS functions

The level 2 BLAS functions were published in [(Dongarra, 1988)][Dongarra-1988], and define matrix-vector operations.

[Dongarra-1988]: https://dl.acm.org/doi/10.1145/42288.42291

return a vector

`LinearAlgebra.BLAS.gemv!` – Function.

```
gemv!(tA, alpha, A, x, beta, y)
```

Update the vector y as `alpha*A*x + beta*y` or `alpha*A'x + beta*y` according to `tA`. `alpha` and `beta` are scalars. Return the updated y.

`LinearAlgebra.BLAS.gemv` – Method.

```
gemv(tA, alpha, A, x)
```

Return `alpha*A*x` or `alpha*A'x` according to `tA`. `alpha` is a scalar.

`LinearAlgebra.BLAS.gemv` – Method.

```
gemv(tA, A, x)
```

Return `A*x` or `A'x` according to `tA`.

`LinearAlgebra.BLAS.gbmv!` – Function.

```
gbmv!(trans, m, kl, ku, alpha, A, x, beta, y)
```

Update vector y as `alpha*A*x + beta*y` or `alpha*A'*x + beta*y` according to `trans`. The matrix `A` is a general band matrix of dimension `m` by `size(A,2)` with `kl` sub-diagonals and `ku` super-diagonals. `alpha` and `beta` are scalars. Return the updated `y`.

`LinearAlgebra.BLAS.gbmv` – Function.

```
gbmv(trans, m, kl, ku, alpha, A, x)
```

Return `alpha*A*x` or `alpha*A'*x` according to `trans`. The matrix `A` is a general band matrix of dimension `m` by `size(A,2)` with `kl` sub-diagonals and `ku` super-diagonals, and `alpha` is a scalar.

`LinearAlgebra.BLAS.hemv!` – Function.

```
hemv!(ul, alpha, A, x, beta, y)
```

Update the vector y as alpha*A*x + beta*y. A is assumed to be Hermitian. Only the ul triangle of A is used. alpha and beta are scalars. Return the updated y.

LinearAlgebra.BLAS.hemv – Method.

```
hemv(ul, alpha, A, x)
```

Return alpha*A*x. A is assumed to be Hermitian. Only the ul triangle of A is used. alpha is a scalar.

LinearAlgebra.BLAS.hemv – Method.

```
hemv(ul, A, x)
```

Return A*x. A is assumed to be Hermitian. Only the ul triangle of A is used.

LinearAlgebra.BLAS.hpmv! – Function.

```
hpmv!(uplo, α, AP, x, β, y)
```

Update vector y as α*A*x + β*y, where A is a Hermitian matrix provided in packed format AP.

With uplo = 'U', the array AP must contain the upper triangular part of the Hermitian matrix packed sequentially, column by column, so that AP[1] contains A[1, 1], AP[2] and AP[3] contain A[1, 2] and A[2, 2] respectively, and so on.

With uplo = 'L', the array AP must contain the lower triangular part of the Hermitian matrix packed sequentially, column by column, so that AP[1] contains A[1, 1], AP[2] and AP[3] contain A[2, 1] and A[3, 1] respectively, and so on.

The scalar inputs α and β must be complex or real numbers.

The array inputs x, y and AP must all be of ComplexF32 or ComplexF64 type.

Return the updated y.

Julia 1.5

hpmv! requires at least Julia 1.5.

LinearAlgebra.BLAS.symv! – Function.

```
symv!(ul, alpha, A, x, beta, y)
```

Update the vector y as alpha*A*x + beta*y. A is assumed to be symmetric. Only the ul triangle of A is used. alpha and beta are scalars. Return the updated y.

LinearAlgebra.BLAS.symv – Method.

```
symv(ul, alpha, A, x)
```

Return `alpha*A*x`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. `alpha` is a scalar.

`LinearAlgebra.BLAS.symv` – Method.

```
symv(ul, A, x)
```

Return `A*x`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.sbmv!` – Function.

```
sbmv!(uplo, k, alpha, A, x, beta, y)
```

Update vector `y` as `alpha*A*x + beta*y` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`. The storage layout for `A` is described the reference BLAS module, level-2 BLAS at http://www.netlib.org/lapack/explore-html/. Only the `uplo` triangle of `A` is used.

Return the updated `y`.

`LinearAlgebra.BLAS.sbmv` – Method.

```
sbmv(uplo, k, alpha, A, x)
```

Return `alpha*A*x` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`. Only the `uplo` triangle of `A` is used.

`LinearAlgebra.BLAS.sbmv` – Method.

```
sbmv(uplo, k, A, x)
```

Return `A*x` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`. Only the `uplo` triangle of `A` is used.

`LinearAlgebra.BLAS.spmv!` – Function.

```
spmv!(uplo, α, AP, x, β, y)
```

Update vector `y` as `α*A*x + β*y`, where `A` is a symmetric matrix provided in packed format `AP`.

With `uplo = 'U'`, the array AP must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that `AP[1]` contains `A[1, 1]`, `AP[2]` and `AP[3]` contain `A[1, 2]` and `A[2, 2]` respectively, and so on.

With uplo = 'L', the array AP must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that AP[1] contains A[1, 1], AP[2] and AP[3] contain A[2, 1] and A[3, 1] respectively, and so on.

The scalar inputs α and β must be real.

The array inputs x, y and AP must all be of Float32 or Float64 type.

Return the updated y.

Julia 1.5

spmv! requires at least Julia 1.5.

LinearAlgebra.BLAS.trmv! – Function.

```
trmv!(ul, tA, dA, A, b)
```

Return op(A)*b, where op is determined by tA. Only the ul triangle of A is used. dA determines if the diagonal values are read or are assumed to be all ones. The multiplication occurs in-place on b.

LinearAlgebra.BLAS.trmv – Function.

```
trmv(ul, tA, dA, A, b)
```

Return op(A)*b, where op is determined by tA. Only the ul triangle of A is used. dA determines if the diagonal values are read or are assumed to be all ones.

LinearAlgebra.BLAS.trsv! – Function.

```
trsv!(ul, tA, dA, A, b)
```

Overwrite b with the solution to A*x = b or one of the other two variants determined by tA and ul. dA determines if the diagonal values are read or are assumed to be all ones. Return the updated b.

LinearAlgebra.BLAS.trsv – Function.

```
trsv(ul, tA, dA, A, b)
```

Return the solution to A*x = b or one of the other two variants determined by tA and ul. dA determines if the diagonal values are read or are assumed to be all ones.

return a matrix

LinearAlgebra.BLAS.ger! – Function.

```
ger!(alpha, x, y, A)
```

Rank-1 update of the matrix A with vectors x and y as alpha*x*y' + A.

`LinearAlgebra.BLAS.her!` – Function.

```
her!(uplo, alpha, x, A)
```

Methods for complex arrays only. Rank-1 update of the Hermitian matrix `A` with vector `x` as `alpha*x*x' + A`. `uplo` controls which triangle of `A` is updated. Returns `A`.

`LinearAlgebra.BLAS.syr!` – Function.

```
syr!(uplo, alpha, x, A)
```

Rank-1 update of the symmetric matrix `A` with vector `x` as `alpha*x*transpose(x) + A`. `uplo` controls which triangle of `A` is updated. Returns `A`.

`LinearAlgebra.BLAS.spr!` – Function.

```
spr!(uplo, α, x, AP)
```

Update matrix `A` as `A+α*x*x'`, where `A` is a symmetric matrix provided in packed format `AP` and `x` is a vector.

With `uplo = 'U'`, the array AP must contain the upper triangular part of the symmetric matrix packed sequentially, column by column, so that `AP[1]` contains `A[1, 1]`, `AP[2]` and `AP[3]` contain `A[1, 2]` and `A[2, 2]` respectively, and so on.

With `uplo = 'L'`, the array AP must contain the lower triangular part of the symmetric matrix packed sequentially, column by column, so that `AP[1]` contains `A[1, 1]`, `AP[2]` and `AP[3]` contain `A[2, 1]` and `A[3, 1]` respectively, and so on.

The scalar input `α` must be real.

The array inputs `x` and `AP` must all be of `Float32` or `Float64` type. Return the updated `AP`.

Julia 1.8

`spr!` requires at least Julia 1.8.

Level 3 BLAS functions

The level 3 BLAS functions were published in [(Dongarra, 1990)][Dongarra-1990], and define matrix-matrix operations.

[Dongarra-1990]: https://dl.acm.org/doi/10.1145/77626.79170

`LinearAlgebra.BLAS.gemm!` – Function.

```
gemm!(tA, tB, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or the other three variants according to `tA` and `tB`. Return the updated `C`.

`LinearAlgebra.BLAS.gemm` – Method.

```
gemm(tA, tB, alpha, A, B)
```

Return `alpha*A*B` or the other three variants according to `tA` and `tB`.

`LinearAlgebra.BLAS.gemm` – Method.

```
gemm(tA, tB, A, B)
```

Return `A*B` or the other three variants according to `tA` and `tB`.

`LinearAlgebra.BLAS.symm!` – Function.

```
symm!(side, ul, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or `alpha*B*A + beta*C` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. Return the updated `C`.

`LinearAlgebra.BLAS.symm` – Method.

```
symm(side, ul, alpha, A, B)
```

Return `alpha*A*B` or `alpha*B*A` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.symm` – Method.

```
symm(side, ul, A, B)
```

Return `A*B` or `B*A` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.hemm!` – Function.

```
hemm!(side, ul, alpha, A, B, beta, C)
```

Update `C` as `alpha*A*B + beta*C` or `alpha*B*A + beta*C` according to `side`. `A` is assumed to be Hermitian. Only the `ul` triangle of `A` is used. Return the updated `C`.

`LinearAlgebra.BLAS.hemm` – Method.

```
hemm(side, ul, alpha, A, B)
```

Return `alpha*A*B` or `alpha*B*A` according to `side`. `A` is assumed to be Hermitian. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.hemm` – Method.

```
hemm(side, ul, A, B)
```

Return `A*B` or `B*A` according to `side`. `A` is assumed to be Hermitian. Only the `ul` triangle of `A` is used.

`LinearAlgebra.BLAS.syrk!` – Function.

```
syrk!(uplo, trans, alpha, A, beta, C)
```

Rank-k update of the symmetric matrix `C` as `alpha*A*transpose(A) + beta*C` or `alpha*transpose(A)*A + beta*C` according to `trans`. Only the `uplo` triangle of `C` is used. Return `C`.

`LinearAlgebra.BLAS.syrk` – Function.

```
syrk(uplo, trans, alpha, A)
```

Return either the upper triangle or the lower triangle of `A`, according to `uplo`, of `alpha*A*transpose(A)` or `alpha*transpose(A)*A`, according to `trans`.

`LinearAlgebra.BLAS.herk!` – Function.

```
herk!(uplo, trans, alpha, A, beta, C)
```

Methods for complex arrays only. Rank-k update of the Hermitian matrix `C` as `alpha*A*A' + beta*C` or `alpha*A'*A + beta*C` according to `trans`. Only the `uplo` triangle of `C` is updated. Returns `C`.

`LinearAlgebra.BLAS.herk` – Function.

```
herk(uplo, trans, alpha, A)
```

Methods for complex arrays only. Returns the `uplo` triangle of `alpha*A*A'` or `alpha*A'*A`, according to `trans`.

`LinearAlgebra.BLAS.syr2k!` – Function.

```
syr2k!(uplo, trans, alpha, A, B, beta, C)
```

Rank-2k update of the symmetric matrix `C` as `alpha*A*transpose(B) + alpha*B*transpose(A) + beta*C` or `alpha*transpose(A)*B + alpha*transpose(B)*A + beta*C` according to `trans`. Only the `uplo` triangle of `C` is used. Returns `C`.

`LinearAlgebra.BLAS.syr2k` – Function.

```
syr2k(uplo, trans, alpha, A, B)
```

Returns the uplo triangle of alpha*A*transpose(B) + alpha*B*transpose(A) or alpha*transpose(A)*B + alpha*transpose(B)*A, according to trans.

```
syr2k(uplo, trans, A, B)
```

Return the uplo triangle of A*transpose(B) + B*transpose(A) or transpose(A)*B + transpose(B)*A, according to trans.

LinearAlgebra.BLAS.her2k! – Function.

```
her2k!(uplo, trans, alpha, A, B, beta, C)
```

Rank-2k update of the Hermitian matrix C as alpha*A*B' + alpha*B*A' + beta*C or alpha*A'*B + alpha*B'*A + beta*C according to trans. The scalar beta has to be real. Only the uplo triangle of C is used. Return C.

LinearAlgebra.BLAS.her2k – Function.

```
her2k(uplo, trans, alpha, A, B)
```

Return the uplo triangle of alpha*A*B' + alpha*B*A' or alpha*A'*B + alpha*B'*A, according to trans.

```
her2k(uplo, trans, A, B)
```

Return the uplo triangle of A*B' + B*A' or A'*B + B'*A, according to trans.

LinearAlgebra.BLAS.trmm! – Function.

```
trmm!(side, ul, tA, dA, alpha, A, B)
```

Update B as alpha*A*B or one of the other three variants determined by side and tA. Only the ul triangle of A is used. dA determines if the diagonal values are read or are assumed to be all ones. Return the updated B.

LinearAlgebra.BLAS.trmm – Function.

```
trmm(side, ul, tA, dA, alpha, A, B)
```

Return alpha*A*B or one of the other three variants determined by side and tA. Only the ul triangle of A is used. dA determines if the diagonal values are read or are assumed to be all ones.

LinearAlgebra.BLAS.trsm! – Function.

```
trsm!(side, ul, tA, dA, alpha, A, B)
```

Overwrite B with the solution to `A*X = alpha*B` or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of A is used. `dA` determines if the diagonal values are read or are assumed to be all ones. Returns the updated B.

`LinearAlgebra.BLAS.trsm` – Function.

```
trsm(side, ul, tA, dA, alpha, A, B)
```

Return the solution to `A*X = alpha*B` or one of the other three variants determined by determined by `side` and `tA`. Only the `ul` triangle of A is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

79.7 LAPACK functions

`LinearAlgebra.LAPACK` provides wrappers for some of the LAPACK functions for linear algebra. Those functions that overwrite one of the input arrays have names ending in `'!'`.

Usually a function has 4 methods defined, one each for `Float64`, `Float32`, `ComplexF64` and `ComplexF32` arrays.

Note that the LAPACK API provided by Julia can and will change in the future. Since this API is not user-facing, there is no commitment to support/deprecate this specific set of functions in future releases.

`LinearAlgebra.LAPACK` – Module.

Interfaces to LAPACK subroutines.

`LinearAlgebra.LAPACK.gbtrf!` – Function.

```
gbtrf!(kl, ku, m, AB) -> (AB, ipiv)
```

Compute the LU factorization of a banded matrix AB. `kl` is the first subdiagonal containing a nonzero band, `ku` is the last superdiagonal containing one, and `m` is the first dimension of the matrix AB. Returns the LU factorization in-place and `ipiv`, the vector of pivots used.

`LinearAlgebra.LAPACK.gbtrs!` – Function.

```
gbtrs!(trans, kl, ku, m, AB, ipiv, B)
```

Solve the equation `AB * X = B`. `trans` determines the orientation of AB. It may be `N` (no transpose), `T` (transpose), or `C` (conjugate transpose). `kl` is the first subdiagonal containing a nonzero band, `ku` is the last superdiagonal containing one, and `m` is the first dimension of the matrix AB. `ipiv` is the vector of pivots returned from `gbtrf!`. Returns the vector or matrix X, overwriting B in-place.

`LinearAlgebra.LAPACK.gebal!` – Function.

```
gebal!(job, A) -> (ilo, ihi, scale)
```

Balance the matrix `A` before computing its eigensystem or Schur factorization. `job` can be one of `N` (`A` will not be permuted or scaled), `P` (`A` will only be permuted), `S` (`A` will only be scaled), or `B` (`A` will be both permuted and scaled). Modifies `A` in-place and returns `ilo`, `ihi`, and `scale`. If permuting was turned on, `A[i,j] = 0` if `j > i` and `1 < j < ilo` or `j > ihi`. `scale` contains information about the scaling/permutations performed.

`LinearAlgebra.LAPACK.gebak!` – Function.

```
gebak!(job, side, ilo, ihi, scale, V)
```

Transform the eigenvectors `V` of a matrix balanced using `gebal!` to the unscaled/unpermuted eigenvectors of the original matrix. Modifies `V` in-place. `side` can be `L` (left eigenvectors are transformed) or `R` (right eigenvectors are transformed).

`LinearAlgebra.LAPACK.gebrd!` – Function.

```
gebrd!(A) -> (A, d, e, tauq, taup)
```

Reduce `A` in-place to bidiagonal form `A = QBP'`. Returns `A`, containing the bidiagonal matrix `B`; `d`, containing the diagonal elements of `B`; `e`, containing the off-diagonal elements of `B`; `tauq`, containing the elementary reflectors representing `Q`; and `taup`, containing the elementary reflectors representing `P`.

`LinearAlgebra.LAPACK.gelqf!` – Function.

```
gelqf!(A, tau)
```

Compute the `LQ` factorization of `A`, `A = LQ`. `tau` contains scalars which parameterize the elementary reflectors of the factorization. `tau` must have length greater than or equal to the smallest dimension of `A`.

Returns `A` and `tau` modified in-place.

```
gelqf!(A) -> (A, tau)
```

Compute the `LQ` factorization of `A`, `A = LQ`.

Returns `A`, modified in-place, and `tau`, which contains scalars which parameterize the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.geqlf!` – Function.

```
geqlf!(A, tau)
```

Compute the QL factorization of A, A = QL. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

```
geqlf!(A) -> (A, tau)
```

Compute the QL factorization of A, A = QL.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqrf! – Function.

```
geqrf!(A, tau)
```

Compute the QR factorization of A, A = QR. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

```
geqrf!(A) -> (A, tau)
```

Compute the QR factorization of A, A = QR.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqp3! – Function.

```
geqp3!(A, [jpvt, tau]) -> (A, tau, jpvt)
```

Compute the pivoted QR factorization of A, AP = QR using BLAS level 3. P is a pivoting matrix, represented by jpvt. tau stores the elementary reflectors. The arguments jpvt and tau are optional and allow for passing preallocated arrays. When passed, jpvt must have length greater than or equal to n if A is an (m x n) matrix and tau must have length greater than or equal to the smallest dimension of A.

A, jpvt, and tau are modified in-place.

LinearAlgebra.LAPACK.gerqf! – Function.

```
gerqf!(A, tau)
```

Compute the RQ factorization of A, A = RQ. tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

```
gerqf!(A) -> (A, tau)
```

Compute the RQ factorization of A, A = RQ.

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqrt! – Function.

```
geqrt!(A, T)
```

Compute the blocked QR factorization of A, A = QR. T contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of T sets the block size and it must be between 1 and n. The second dimension of T must equal the smallest dimension of A.

Returns A and T modified in-place.

```
geqrt!(A, nb) -> (A, T)
```

Compute the blocked QR factorization of A, A = QR. nb sets the block size and it must be between 1 and n, the second dimension of A.

Returns A, modified in-place, and T, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.geqrt3! – Function.

```
geqrt3!(A, T)
```

Recursively computes the blocked QR factorization of A, A = QR. T contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of T sets the block size and it must be between 1 and n. The second dimension of T must equal the smallest dimension of A.

Returns A and T modified in-place.

```
geqrt3!(A) -> (A, T)
```

Recursively computes the blocked QR factorization of A, A = QR.

Returns A, modified in-place, and T, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

LinearAlgebra.LAPACK.getrf! – Function.

```
getrf!(A) -> (A, ipiv, info)
```

Compute the pivoted LU factorization of A, A = LU.

Returns A, modified in-place, ipiv, the pivoting information, and an info code which indicates success (info = 0), a singular value in U (info = i, in which case U[i,i] is singular), or an error code (info < 0).

`LinearAlgebra.LAPACK.tzrzf!` – Function.

```
tzrzf!(A) -> (A, tau)
```

Transforms the upper trapezoidal matrix `A` to upper triangular form in-place. Returns `A` and `tau`, the scalar parameters for the elementary reflectors of the transformation.

`LinearAlgebra.LAPACK.ormrz!` – Function.

```
ormrz!(side, trans, A, tau, C)
```

Multiplies the matrix `C` by `Q` from the transformation supplied by `tzrzf!`. Depending on `side` or `trans` the multiplication can be left-sided (`side = L, Q*C`) or right-sided (`side = R, C*Q`) and `Q` can be unmodified (`trans = N`), transposed (`trans = T`), or conjugate transposed (`trans = C`). Returns matrix `C` which is modified in-place with the result of the multiplication.

`LinearAlgebra.LAPACK.gels!` – Function.

```
gels!(trans, A, B) -> (F, B, ssr)
```

Solves the linear equation `A * X = B`, `transpose(A) * X = B`, or `adjoint(A) * X = B` using a QR or LQ factorization. Modifies the matrix/vector `B` in place with the solution. `A` is overwritten with its `QR` or `LQ` factorization. `trans` may be one of `N` (no modification), `T` (transpose), or `C` (conjugate transpose). `gels!` searches for the minimum norm/least squares solution. `A` may be under or over determined. The solution is returned in `B`.

`LinearAlgebra.LAPACK.gesv!` – Function.

```
gesv!(A, B) -> (B, A, ipiv)
```

Solves the linear equation `A * X = B` where `A` is a square matrix using the `LU` factorization of `A`. `A` is overwritten with its `LU` factorization and `B` is overwritten with the solution `X`. `ipiv` contains the pivoting information for the `LU` factorization of `A`.

`LinearAlgebra.LAPACK.getrs!` – Function.

```
getrs!(trans, A, ipiv, B)
```

Solves the linear equation `A * X = B`, `transpose(A) * X = B`, or `adjoint(A) * X = B` for square `A`. Modifies the matrix/vector `B` in place with the solution. `A` is the `LU` factorization from `getrf!`, with `ipiv` the pivoting information. `trans` may be one of `N` (no modification), `T` (transpose), or `C` (conjugate transpose).

`LinearAlgebra.LAPACK.getri!` – Function.

```
getri!(A, ipiv)
```

Computes the inverse of A, using its LU factorization found by getrf!. ipiv is the pivot information output and A contains the LU factorization of getrf!. A is overwritten with its inverse.

LinearAlgebra.LAPACK.gesvx! – Function.

```
gesvx!(fact, trans, A, AF, ipiv, equed, R, C, B) -> (X, equed, R, C, B, rcond, ferr, berr, work)
```

Solves the linear equation A * X = B (trans = N), transpose(A) * X = B (trans = T), or adjoint(A) * X = B (trans = C) using the LU factorization of A. fact may be E, in which case A will be equilibrated and copied to AF; F, in which case AF and ipiv from a previous LU factorization are inputs; or N, in which case A will be copied to AF and then factored. If fact = F, equed may be N, meaning A has not been equilibrated; R, meaning A was multiplied by Diagonal(R) from the left; C, meaning A was multiplied by Diagonal(C) from the right; or B, meaning A was multiplied by Diagonal(R) from the left and Diagonal(C) from the right. If fact = F and equed = R or B the elements of R must all be positive. If fact = F and equed = C or B the elements of C must all be positive.

Returns the solution X; equed, which is an output if fact is not N, and describes the equilibration that was performed; R, the row equilibration diagonal; C, the column equilibration diagonal; B, which may be overwritten with its equilibrated form Diagonal(R)*B (if trans = N and equed = R,B) or Diagonal(C)*B (if trans = T,C and equed = C,B); rcond, the reciprocal condition number of A after equilbrating; ferr, the forward error bound for each solution vector in X; berr, the forward error bound for each solution vector in X; and work, the reciprocal pivot growth factor.

```
gesvx!(A, B)
```

The no-equilibration, no-transpose simplification of gesvx!.

LinearAlgebra.LAPACK.gelsd! – Function.

```
gelsd!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of A * X = B by finding the SVD factorization of A, then dividing-and-conquering the problem. B is overwritten with the solution X. Singular values below rcond will be treated as zero. Returns the solution in B and the effective rank of A in rnk.

LinearAlgebra.LAPACK.gelsy! – Function.

```
gelsy!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of A * X = B by finding the full QR factorization of A, then dividing-and-conquering the problem. B is overwritten with the solution X. Singular values below rcond will be treated as zero. Returns the solution in B and the effective rank of A in rnk.

LinearAlgebra.LAPACK.gglse! – Function.

```
gglse!(A, c, B, d) -> (X,res)
```

Solves the equation `A * x = c` where `x` is subject to the equality constraint `B * x = d`. Uses the formula `||c - A*x||^2 = 0` to solve. Returns `X` and the residual sum-of-squares.

`LinearAlgebra.LAPACK.geev!` – Function.

```
geev!(jobvl, jobvr, A) -> (W, VL, VR)
```

Finds the eigensystem of `A`. If `jobvl = N`, the left eigenvectors of `A` aren't computed. If `jobvr = N`, the right eigenvectors of `A` aren't computed. If `jobvl = V` or `jobvr = V`, the corresponding eigenvectors are computed. Returns the eigenvalues in `W`, the right eigenvectors in `VR`, and the left eigenvectors in `VL`.

`LinearAlgebra.LAPACK.gesdd!` – Function.

```
gesdd!(job, A) -> (U, S, VT)
```

Finds the singular value decomposition of `A`, `A = U * S * V'`, using a divide and conquer approach. If `job = A`, all the columns of `U` and the rows of `V'` are computed. If `job = N`, no columns of `U` or rows of `V'` are computed. If `job = O`, `A` is overwritten with the columns of (thin) `U` and the rows of (thin) `V'`. If `job = S`, the columns of (thin) `U` and the rows of (thin) `V'` are computed and returned separately.

`LinearAlgebra.LAPACK.gesvd!` – Function.

```
gesvd!(jobu, jobvt, A) -> (U, S, VT)
```

Finds the singular value decomposition of `A`, `A = U * S * V'`. If `jobu = A`, all the columns of `U` are computed. If `jobvt = A` all the rows of `V'` are computed. If `jobu = N`, no columns of `U` are computed. If `jobvt = N` no rows of `V'` are computed. If `jobu = O`, `A` is overwritten with the columns of (thin) `U`. If `jobvt = O`, `A` is overwritten with the rows of (thin) `V'`. If `jobu = S`, the columns of (thin) `U` are computed and returned separately. If `jobvt = S` the rows of (thin) `V'` are computed and returned separately. `jobu` and `jobvt` can't both be `O`.

Returns `U`, `S`, and `Vt`, where `S` are the singular values of `A`.

`LinearAlgebra.LAPACK.ggsvd!` – Function.

```
ggsvd!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l, R)
```

Finds the generalized singular value decomposition of `A` and `B`, `U'*A*Q = D1*R` and `V'*B*Q = D2*R`. `D1` has `alpha` on its diagonal and `D2` has `beta` on its diagonal. If `jobu = U`, the orthogonal/unitary matrix `U` is computed. If `jobv = V` the orthogonal/unitary matrix `V` is computed. If `jobq = Q`, the orthogonal/unitary matrix `Q` is computed. If `jobu`, `jobv` or `jobq` is `N`, that matrix is not computed. This function is only available in LAPACK versions prior to 3.6.0.

`LinearAlgebra.LAPACK.ggsvd3!` – Function.

```
ggsvd3!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l, R)
```

Finds the generalized singular value decomposition of A and B, U'*A*Q = D1*R and V'*B*Q = D2*R. D1 has alpha on its diagonal and D2 has beta on its diagonal. If jobu = U, the orthogonal/unitary matrix U is computed. If jobv = V the orthogonal/unitary matrix V is computed. If jobq = Q, the orthogonal/unitary matrix Q is computed. If jobu, jobv, or jobq is N, that matrix is not computed. This function requires LAPACK 3.6.0.

LinearAlgebra.LAPACK.geevx! – Function.

```
geevx!(balanc, jobvl, jobvr, sense, A) -> (A, w, VL, VR, ilo, ihi, scale, abnrm, rconde, rcondv)
```

Finds the eigensystem of A with matrix balancing. If jobvl = N, the left eigenvectors of A aren't computed. If jobvr = N, the right eigenvectors of A aren't computed. If jobvl = V or jobvr = V, the corresponding eigenvectors are computed. If balanc = N, no balancing is performed. If balanc = P, A is permuted but not scaled. If balanc = S, A is scaled but not permuted. If balanc = B, A is permuted and scaled. If sense = N, no reciprocal condition numbers are computed. If sense = E, reciprocal condition numbers are computed for the eigenvalues only. If sense = V, reciprocal condition numbers are computed for the right eigenvectors only. If sense = B, reciprocal condition numbers are computed for the right eigenvectors and the eigenvectors. If sense = E,B, the right and left eigenvectors must be computed.

LinearAlgebra.LAPACK.ggev! – Function.

```
ggev!(jobvl, jobvr, A, B) -> (alpha, beta, vl, vr)
```

Finds the generalized eigendecomposition of A and B. If jobvl = N, the left eigenvectors aren't computed. If jobvr = N, the right eigenvectors aren't computed. If jobvl = V or jobvr = V, the corresponding eigenvectors are computed.

LinearAlgebra.LAPACK.ggev3! – Function.

```
ggev3!(jobvl, jobvr, A, B) -> (alpha, beta, vl, vr)
```

Finds the generalized eigendecomposition of A and B using a blocked algorithm. If jobvl = N, the left eigenvectors aren't computed. If jobvr = N, the right eigenvectors aren't computed. If jobvl = V or jobvr = V, the corresponding eigenvectors are computed. This function requires LAPACK 3.6.0.

LinearAlgebra.LAPACK.gtsv! – Function.

```
gtsv!(dl, d, du, B)
```

Solves the equation A * X = B where A is a tridiagonal matrix with dl on the subdiagonal, d on the diagonal, and du on the superdiagonal.

Overwrites B with the solution X and returns it.

`LinearAlgebra.LAPACK.gttrf!` – Function.

```
gttrf!(dl, d, du) -> (dl, d, du, du2, ipiv)
```

Finds the LU factorization of a tridiagonal matrix with `dl` on the subdiagonal, `d` on the diagonal, and `du` on the superdiagonal.

Modifies `dl`, `d`, and `du` in-place and returns them and the second superdiagonal `du2` and the pivoting vector `ipiv`.

`LinearAlgebra.LAPACK.gttrs!` – Function.

```
gttrs!(trans, dl, d, du, du2, ipiv, B)
```

Solves the equation `A * X = B` (`trans = N`), `transpose(A) * X = B` (`trans = T`), or `adjoint(A) * X = B` (`trans = C`) using the LU factorization computed by `gttrf!`. `B` is overwritten with the solution `X`.

`LinearAlgebra.LAPACK.orglq!` – Function.

```
orglq!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `LQ` factorization after calling `gelqf!` on `A`. Uses the output of `gelqf!`. `A` is overwritten by `Q`.

`LinearAlgebra.LAPACK.orgqr!` – Function.

```
orgqr!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `QR` factorization after calling `geqrf!` on `A`. Uses the output of `geqrf!`. `A` is overwritten by `Q`.

`LinearAlgebra.LAPACK.orgql!` – Function.

```
orgql!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `QL` factorization after calling `geqlf!` on `A`. Uses the output of `geqlf!`. `A` is overwritten by `Q`.

`LinearAlgebra.LAPACK.orgrq!` – Function.

```
orgrq!(A, tau, k = length(tau))
```

Explicitly finds the matrix `Q` of a `RQ` factorization after calling `gerqf!` on `A`. Uses the output of `gerqf!`. `A` is overwritten by `Q`.

`LinearAlgebra.LAPACK.ormlq!` – Function.

```
ormlq!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `LQ` factorization of `A` computed using `gelqf!`. `C` is overwritten.

`LinearAlgebra.LAPACK.ormqr!` – Function.

```
ormqr!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QR` factorization of `A` computed using `geqrf!`. `C` is overwritten.

`LinearAlgebra.LAPACK.ormql!` – Function.

```
ormql!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QL` factorization of `A` computed using `geqlf!`. `C` is overwritten.

`LinearAlgebra.LAPACK.ormrq!` – Function.

```
ormrq!(side, trans, A, tau, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `RQ` factorization of `A` computed using `gerqf!`. `C` is overwritten.

`LinearAlgebra.LAPACK.gemqrt!` – Function.

```
gemqrt!(side, trans, V, T, C)
```

Computes `Q * C` (`trans = N`), `transpose(Q) * C` (`trans = T`), `adjoint(Q) * C` (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using `Q` from a `QR` factorization of `A` computed using `geqrt!`. `C` is overwritten.

`LinearAlgebra.LAPACK.posv!` – Function.

```
posv!(uplo, A, B) -> (A, B)
```

Finds the solution to `A * X = B` where `A` is a symmetric or Hermitian positive definite matrix. If `uplo = U` the upper Cholesky decomposition of `A` is computed. If `uplo = L` the lower Cholesky decomposition of `A` is computed. `A` is overwritten by its Cholesky decomposition. `B` is overwritten with the solution `X`.

`LinearAlgebra.LAPACK.potrf!` – Function.

```
potrf!(uplo, A)
```

Computes the Cholesky (upper if `uplo = U`, lower if `uplo = L`) decomposition of positive-definite matrix `A`. `A` is overwritten and returned with an info code.

`LinearAlgebra.LAPACK.potri!` – Function.

```
potri!(uplo, A)
```

Computes the inverse of positive-definite matrix `A` after calling `potrf!` to find its (upper if `uplo = U`, lower if `uplo = L`) Cholesky decomposition.

`A` is overwritten by its inverse and returned.

`LinearAlgebra.LAPACK.potrs!` – Function.

```
potrs!(uplo, A, B)
```

Finds the solution to `A * X = B` where `A` is a symmetric or Hermitian positive definite matrix whose Cholesky decomposition was computed by `potrf!`. If `uplo = U` the upper Cholesky decomposition of `A` was computed. If `uplo = L` the lower Cholesky decomposition of `A` was computed. `B` is overwritten with the solution `X`.

`LinearAlgebra.LAPACK.pstrf!` – Function.

```
pstrf!(uplo, A, tol) -> (A, piv, rank, info)
```

Computes the (upper if `uplo = U`, lower if `uplo = L`) pivoted Cholesky decomposition of positive-definite matrix `A` with a user-set tolerance `tol`. `A` is overwritten by its Cholesky decomposition.

Returns `A`, the pivots `piv`, the rank of `A`, and an `info` code. If `info = 0`, the factorization succeeded. If `info = i > 0`, then `A` is indefinite or rank-deficient.

`LinearAlgebra.LAPACK.ptsv!` – Function.

```
ptsv!(D, E, B)
```

Solves `A * X = B` for positive-definite tridiagonal `A`. `D` is the diagonal of `A` and `E` is the off-diagonal. `B` is overwritten with the solution `X` and returned.

`LinearAlgebra.LAPACK.pttrf!` – Function.

```
pttrf!(D, E)
```

Computes the LDLt factorization of a positive-definite tridiagonal matrix with D as diagonal and E as off-diagonal. D and E are overwritten and returned.

LinearAlgebra.LAPACK.pttrs! – Function.

```
pttrs!(D, E, B)
```

Solves A * X = B for positive-definite tridiagonal A with diagonal D and off-diagonal E after computing A's LDLt factorization using pttrf!. B is overwritten with the solution X.

LinearAlgebra.LAPACK.trtri! – Function.

```
trtri!(uplo, diag, A)
```

Finds the inverse of (upper if uplo = U, lower if uplo = L) triangular matrix A. If diag = N, A has non-unit diagonal elements. If diag = U, all diagonal elements of A are one. A is overwritten with its inverse.

LinearAlgebra.LAPACK.trtrs! – Function.

```
trtrs!(uplo, trans, diag, A, B)
```

Solves A * X = B (trans = N), transpose(A) * X = B (trans = T), or adjoint(A) * X = B (trans = C) for (upper if uplo = U, lower if uplo = L) triangular matrix A. If diag = N, A has non-unit diagonal elements. If diag = U, all diagonal elements of A are one. B is overwritten with the solution X.

LinearAlgebra.LAPACK.trcon! – Function.

```
trcon!(norm, uplo, diag, A)
```

Finds the reciprocal condition number of (upper if uplo = U, lower if uplo = L) triangular matrix A. If diag = N, A has non-unit diagonal elements. If diag = U, all diagonal elements of A are one. If norm = I, the condition number is found in the infinity norm. If norm = O or 1, the condition number is found in the one norm.

LinearAlgebra.LAPACK.trevc! – Function.

```
trevc!(side, howmny, select, T, VL = similar(T), VR = similar(T))
```

Finds the eigensystem of an upper triangular matrix T. If side = R, the right eigenvectors are computed. If side = L, the left eigenvectors are computed. If side = B, both sets are computed. If howmny = A, all eigenvectors are found. If howmny = B, all eigenvectors are found and backtransformed using VL and VR. If howmny = S, only the eigenvectors corresponding to the values in select are computed.

LinearAlgebra.LAPACK.trrfs! – Function.

```
trrfs!(uplo, trans, diag, A, B, X, Ferr, Berr) -> (Ferr, Berr)
```

Estimates the error in the solution to `A * X = B` (`trans = N`), `transpose(A) * X = B` (`trans = T`), `adjoint(A) * X = B` (`trans = C`) for `side = L`, or the equivalent equations a right-handed `side = R` `X * A` after computing `X` using `trtrs!`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, `A` is lower triangular. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `Ferr` and `Berr` are optional inputs. `Ferr` is the forward error and `Berr` is the backward error, each component-wise.

`LinearAlgebra.LAPACK.stev!` – Function.

```
stev!(job, dv, ev) -> (dv, Zmat)
```

Computes the eigensystem for a symmetric tridiagonal matrix with dv as diagonal and ev as off-diagonal. If `job = N` only the eigenvalues are found and returned in `dv`. If `job = V` then the eigenvectors are also found and returned in `Zmat`.

`LinearAlgebra.LAPACK.stebz!` – Function.

```
stebz!(range, order, vl, vu, il, iu, abstol, dv, ev) -> (dv, iblock, isplit)
```

Computes the eigenvalues for a symmetric tridiagonal matrix with dv as diagonal and ev as off-diagonal. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. If `order = B`, eigvalues are ordered within a block. If `order = E`, they are ordered across all the blocks. `abstol` can be set as a tolerance for convergence.

`LinearAlgebra.LAPACK.stegr!` – Function.

```
stegr!(jobz, range, dv, ev, vl, vu, il, iu) -> (w, Z)
```

Computes the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) for a symmetric tridiagonal matrix with dv as diagonal and ev as off-diagonal. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. The eigenvalues are returned in `w` and the eigenvectors in `Z`.

`LinearAlgebra.LAPACK.stein!` – Function.

```
stein!(dv, ev_in, w_in, iblock_in, isplit_in)
```

Computes the eigenvectors for a symmetric tridiagonal matrix with `dv` as diagonal and `ev_in` as off-diagonal. `w_in` specifies the input eigenvalues for which to find corresponding eigenvectors. `iblock_in` specifies the submatrices corresponding to the eigenvalues in `w_in`. `isplit_in` specifies the splitting points between the submatrix blocks.

`LinearAlgebra.LAPACK.syconv!` – Function.

```
syconv!(uplo, A, ipiv) -> (A, work)
```

Converts a symmetric matrix `A` (which has been factorized into a triangular matrix) into two matrices `L` and `D`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, it is lower triangular. `ipiv` is the pivot vector from the triangular factorization. `A` is overwritten by `L` and `D`.

`LinearAlgebra.LAPACK.sysv!` – Function.

```
sysv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

`LinearAlgebra.LAPACK.sytrf!` – Function.

```
sytrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

`LinearAlgebra.LAPACK.sytri!` – Function.

```
sytri!(uplo, A, ipiv)
```

Computes the inverse of a symmetric matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `A` is overwritten by its inverse.

`LinearAlgebra.LAPACK.sytrs!` – Function.

```
sytrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a symmetric matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`.

`LinearAlgebra.LAPACK.hesv!` – Function.

```
hesv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

`LinearAlgebra.LAPACK.hetrf!` – Function.

```
hetrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a Hermitian matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

`LinearAlgebra.LAPACK.hetri!` – Function.

```
hetri!(uplo, A, ipiv)
```

Computes the inverse of a Hermitian matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `A` is overwritten by its inverse.

`LinearAlgebra.LAPACK.hetrs!` – Function.

```
hetrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a Hermitian matrix `A` using the results of `sytrf!`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`.

`LinearAlgebra.LAPACK.syev!` – Function.

```
syev!(jobz, uplo, A)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used.

`LinearAlgebra.LAPACK.syevr!` – Function.

```
syevr!(jobz, range, uplo, A, vl, vu, il, iu, abstol) -> (W, Z)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval `(vl, vu]` are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. `abstol` can be set as a tolerance for convergence.

The eigenvalues are returned in `W` and the eigenvectors in `Z`.

`LinearAlgebra.LAPACK.syevd!` – Function.

```
syevd!(jobz, uplo, A)
```

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A`. If `uplo = U`, the upper triangle of `A` is used. If `uplo = L`, the lower triangle of `A` is used.

Use the divide-and-conquer method, instead of the QR iteration used by `syev!` or multiple relatively robust representations used by `syevr!`. See James W. Demmel et al, SIAM J. Sci. Comput. 30, 3, 1508 (2008) for a comparison of the accuracy and performatce of different methods.

`LinearAlgebra.LAPACK.sygvd!` – Function.

```
sygvd!(itype, jobz, uplo, A, B) -> (w, A, B)
```

Finds the generalized eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix `A` and symmetric positive-definite matrix `B`. If `uplo = U`, the upper triangles of `A` and `B` are used. If `uplo = L`, the lower triangles of `A` and `B` are used. If `itype = 1`, the problem to solve is `A * x = lambda * B * x`. If `itype = 2`, the problem to solve is `A * B * x = lambda * x`. If `itype = 3`, the problem to solve is `B * A * x = lambda * x`.

`LinearAlgebra.LAPACK.bdsqr!` – Function.

```
bdsqr!(uplo, d, e_, Vt, U, C) -> (d, Vt, U, C)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal. If `uplo = U`, `e_` is the superdiagonal. If `uplo = L`, `e_` is the subdiagonal. Can optionally also compute the product `Q' * C`.

Returns the singular values in `d`, and the matrix `C` overwritten with `Q' * C`.

`LinearAlgebra.LAPACK.bdsdc!` – Function.

```
bdsdc!(uplo, compq, d, e_) -> (d, e, u, vt, q, iq)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal using a divide and conqueq method. If `uplo = U`, `e_` is the superdiagonal. If `uplo = L`, `e_` is the subdiagonal. If `compq = N`, only the singular values are found. If `compq = I`, the singular values and vectors are found. If `compq = P`, the singular values and vectors are found in compact form. Only works for real types.

Returns the singular values in `d`, and if `compq = P`, the compact singular vectors in `iq`.

`LinearAlgebra.LAPACK.gecon!` – Function.

```
gecon!(normtype, A, anorm)
```

Finds the reciprocal condition number of matrix `A`. If `normtype = I`, the condition number is found in the infinity norm. If `normtype = O` or `1`, the condition number is found in the one norm. `A` must be the result of `getrf!` and `anorm` is the norm of `A` in the relevant norm.

`LinearAlgebra.LAPACK.gehrd!` – Function.

```
gehrd!(ilo, ihi, A) -> (A, tau)
```

Converts a matrix `A` to Hessenberg form. If `A` is balanced with `gebal!` then `ilo` and `ihi` are the outputs of `gebal!`. Otherwise they should be `ilo = 1` and `ihi = size(A,2)`. `tau` contains the elementary reflectors of the factorization.

`LinearAlgebra.LAPACK.orghr!` – Function.

```
orghr!(ilo, ihi, A, tau)
```

Explicitly finds `Q`, the orthogonal/unitary matrix from `gehrd!`. `ilo`, `ihi`, `A`, and `tau` must correspond to the input/output to `gehrd!`.

`LinearAlgebra.LAPACK.gees!` – Function.

```
gees!(jobvs, A) -> (A, vs, w)
```

Computes the eigenvalues (`jobvs = N`) or the eigenvalues and Schur vectors (`jobvs = V`) of matrix `A`. `A` is overwritten by its Schur form.

Returns `A`, `vs` containing the Schur vectors, and `w`, containing the eigenvalues.

`LinearAlgebra.LAPACK.gges!` – Function.

```
gges!(jobvsl, jobvsr, A, B) -> (A, B, alpha, beta, vsl, vsr)
```

Computes the generalized eigenvalues, generalized Schur form, left Schur vectors (`jobsvl = V`), or right Schur vectors (`jobvsr = V`) of `A` and `B`.

The generalized eigenvalues are returned in `alpha` and `beta`. The left Schur vectors are returned in `vsl` and the right Schur vectors are returned in `vsr`.

`LinearAlgebra.LAPACK.gges3!` – Function.

```
gges3!(jobvsl, jobvsr, A, B) -> (A, B, alpha, beta, vsl, vsr)
```

Computes the generalized eigenvalues, generalized Schur form, left Schur vectors (`jobsvl = V`), or right Schur vectors (`jobvsr = V`) of `A` and `B` using a blocked algorithm. This function requires LAPACK 3.6.0.

The generalized eigenvalues are returned in `alpha` and `beta`. The left Schur vectors are returned in `vsl` and the right Schur vectors are returned in `vsr`.

`LinearAlgebra.LAPACK.trexc!` – Function.

```
trexc!(compq, ifst, ilst, T, Q) -> (T, Q)
trexc!(ifst, ilst, T, Q) -> (T, Q)
```

Reorder the Schur factorization T of a matrix, such that the diagonal block of T with row index ifst is moved to row index ilst. If compq = V, the Schur vectors Q are reordered. If compq = N they are not modified. The 4-arg method calls the 5-arg method with compq = V.

LinearAlgebra.LAPACK.trsen! – Function.

```
trsen!(job, compq, select, T, Q) -> (T, Q, w, s, sep)
trsen!(select, T, Q) -> (T, Q, w, s, sep)
```

Reorder the Schur factorization of a matrix and optionally finds reciprocal condition numbers. If job = N, no condition numbers are found. If job = E, only the condition number for this cluster of eigenvalues is found. If job = V, only the condition number for the invariant subspace is found. If job = B then the condition numbers for the cluster and subspace are found. If compq = V the Schur vectors Q are updated. If compq = N the Schur vectors are not modified. select determines which eigenvalues are in the cluster. The 3-arg method calls the 5-arg method with job = N and compq = V.

Returns T, Q, reordered eigenvalues in w, the condition number of the cluster of eigenvalues s, and the condition number of the invariant subspace sep.

LinearAlgebra.LAPACK.tgsen! – Function.

```
tgsen!(select, S, T, Q, Z) -> (S, T, alpha, beta, Q, Z)
```

Reorders the vectors of a generalized Schur decomposition. select specifies the eigenvalues in each cluster.

LinearAlgebra.LAPACK.trsyl! – Function.

```
trsyl!(transa, transb, A, B, C, isgn=1) -> (C, scale)
```

Solves the Sylvester matrix equation A * X +/- X * B = scale*C where A and B are both quasi-upper triangular. If transa = N, A is not modified. If transa = T, A is transposed. If transa = C, A is conjugate transposed. Similarly for transb and B. If isgn = 1, the equation A * X + X * B = scale * C is solved. If isgn = -1, the equation A * X - X * B = scale * C is solved.

Returns X (overwriting C) and scale.

LinearAlgebra.LAPACK.hseqr! – Function.

```
hseqr!(job, compz, ilo, ihi, H, Z) -> (H, Z, w)
```

Computes all eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form. If H is balanced with gebal! then ilo and ihi are the outputs of gebal!. Otherwise they should be ilo = 1 and ihi = size(H,2). tau contains the elementary reflectors of the factorization.

Chapter 80

日志记录

Logging 模块提供了一个将历史和计算进度记录为事件的日志。事件通过在源代码里插入日志语句产生，例如：

```
@warn "Abandon printf debugging, all ye who enter here!"
┌ Warning: Abandon printf debugging, all ye who enter here!
└ @ Main REPL[1]:1
```

The system provides several advantages over peppering your source code with calls to `println()`. First, it allows you to control the visibility and presentation of messages without editing the source code. For example, in contrast to the `@warn` above

```
@debug "The sum of some values $(sum(rand(100)))"
```

will produce no output by default. Furthermore, it's very cheap to leave debug statements like this in the source code because the system avoids evaluating the message if it would later be ignored. In this case `sum(rand(100))` and the associated string processing will never be executed unless debug logging is enabled.

Second, the logging tools allow you to attach arbitrary data to each event as a set of key–value pairs. This allows you to capture local variables and other program state for later analysis. For example, to attach the local array variable `A` and the sum of a vector `v` as the key `s` you can use

```
A = ones(Int, 4, 4)
v = ones(100)
@info "Some variables"  A  s=sum(v)

# output
┌ Info: Some variables
│   A =
│    4×4 Matrix{Int64}:
│     1  1  1  1
│     1  1  1  1
│     1  1  1  1
│     1  1  1  1
└   s = 100.0
```

所有的日志宏如 `@debug`, `@info`, `@warn` 和 `@error` 有着共同的特征，这些共同特征在更通用的宏 `@logmsg` 的文档里有细致说明。

80.1 日志事件结构

Each event generates several pieces of data, some provided by the user and some automatically extracted. Let's examine the user-defined data first:

- The *log level* is a broad category for the message that is used for early filtering. There are several standard levels of type `LogLevel`; user-defined levels are also possible. Each is distinct in purpose:
  - `Logging.Debug` (log level -1000) is information intended for the developer of the program. These events are disabled by default.
  - `Logging.Info` (log level 0) is for general information to the user. Think of it as an alternative to using `println` directly.
  - `Logging.Warn` (log level 1000) means something is wrong and action is likely required but that for now the program is still working.
  - `Logging.Error` (log level 2000) means something is wrong and it is unlikely to be recovered, at least by this part of the code. Often this log-level is unneeded as throwing an exception can convey all the required information.
- The *message* is an object describing the event. By convention `AbstractStrings` passed as messages are assumed to be in markdown format. Other types will be displayed using `print(io, obj)` or `string(obj)` for text-based output and possibly `show(io,mime,obj)` for other multimedia displays used in the installed logger.
- Optional *key–value pairs* allow arbitrary data to be attached to each event. Some keys have conventional meaning that can affect the way an event is interpreted (see `@logmsg`).

The system also generates some standard information for each event:

- The `module` in which the logging macro was expanded.
- The `file` and `line` where the logging macro occurs in the source code.
- A message `id` that is a unique, fixed identifier for the *source code statement* where the logging macro appears. This identifier is designed to be fairly stable even if the source code of the file changes, as long as the logging statement itself remains the same.
- A `group` for the event, which is set to the base name of the file by default, without extension. This can be used to group messages into categories more finely than the log level (for example, all deprecation warnings have group `:depwarn`), or into logical groupings across or within modules.

Notice that some useful information such as the event time is not included by default. This is because such information can be expensive to extract and is also *dynamically* available to the current logger. It's simple to define a custom logger to augment event data with the time, backtrace, values of global variables and other useful information as required.

80.2 Processing log events

As you can see in the examples, logging statements make no mention of where log events go or how they are processed. This is a key design feature that makes the system composable and natural for concurrent use. It does this by separating two different concerns:

- *Creating* log events is the concern of the module author who needs to decide where events are triggered and which information to include.
- *Processing* of log events —that is, display, filtering, aggregation and recording —is the concern of the application author who needs to bring multiple modules together into a cooperating application.

Loggers

Processing of events is performed by a *logger*, which is the first piece of user configurable code to see the event. All loggers must be subtypes of `AbstractLogger`.

When an event is triggered, the appropriate logger is found by looking for a task-local logger with the global logger as fallback. The idea here is that the application code knows how log events should be processed and exists somewhere at the top of the call stack. So we should look up through the call stack to discover the logger —that is, the logger should be *dynamically scoped*. (This is a point of contrast with logging frameworks where the logger is *lexically scoped*; provided explicitly by the module author or as a simple global variable. In such a system it's awkward to control logging while composing functionality from multiple modules.)

The global logger may be set with `global_logger`, and task-local loggers controlled using `with_logger`. Newly spawned tasks inherit the logger of the parent task.

There are three logger types provided by the library. `ConsoleLogger` is the default logger you see when starting the REPL. It displays events in a readable text format and tries to give simple but user friendly control over formatting and filtering. `NullLogger` is a convenient way to drop all messages where necessary; it is the logging equivalent of the `devnull` stream. `SimpleLogger` is a very simplistic text formatting logger, mainly useful for debugging the logging system itself.

Custom loggers should come with overloads for the functions described in the reference section.

Early filtering and message handling

When an event occurs, a few steps of early filtering occur to avoid generating messages that will be discarded:

1. The message log level is checked against a global minimum level (set via `disable_logging`). This is a crude but extremely cheap global setting.
2. The current logger state is looked up and the message level checked against the logger's cached minimum level, as found by calling `Logging.min_enabled_level`. This behavior can be overridden via environment variables (more on this later).
3. The `Logging.shouldlog` function is called with the current logger, taking some minimal information (level, module, group, id) which can be computed statically. Most usefully, `shouldlog` is passed an event `id` which can be used to discard events early based on a cached predicate.

If all these checks pass, the message and key–value pairs are evaluated in full and passed to the current logger via the `Logging.handle_message` function. `handle_message()` may perform additional filtering as required and display the event to the screen, save it to a file, etc.

Exceptions that occur while generating the log event are captured and logged by default. This prevents individual broken events from crashing the application, which is helpful when enabling little-used debug events in a production system. This behavior can be customized per logger type by extending `Logging.catch_exceptions`.

80.3 Testing log events

Log events are a side effect of running normal code, but you might find yourself wanting to test particular informational messages and warnings. The `Test` module provides a `@test_logs` macro that can be used to pattern match against the log event stream.

80.4 Environment variables

Message filtering can be influenced through the `JULIA_DEBUG` environment variable, and serves as an easy way to enable debug logging for a file or module. Loading julia with `JULIA_DEBUG=loading` will activate @debug log messages in `loading.jl`. For example, in Linux shells:

```
$ JULIA_DEBUG=loading julia -e 'using OhMyREPL'
┌ Debug: Rejecting cache file /home/user/.julia/compiled/v0.7/OhMyREPL.ji due to it containing an
↪  invalid cache header
└ @ Base loading.jl:1328
[ Info: Recompiling stale cache file /home/user/.julia/compiled/v0.7/OhMyREPL.ji for module
↪  OhMyREPL
┌ Debug: Rejecting cache file /home/user/.julia/compiled/v0.7/Tokenize.ji due to it containing an
↪  invalid cache header
└ @ Base loading.jl:1328
...
```

On windows, the same can be achieved in `CMD` via first running `set JULIA_DEBUG="loading"` and in `Powershell` via `$env:JULIA_DEBUG="loading"`.

Similarly, the environment variable can be used to enable debug logging of modules, such as Pkg, or module roots (see `Base.moduleroot`). To enable all debug logging, use the special value `all`.

To turn debug logging on from the REPL, set `ENV["JULIA_DEBUG"]` to the name of the module of interest. Functions defined in the REPL belong to module `Main`; logging for them can be enabled like this:

```
julia> foo() = @debug "foo"
foo (generic function with 1 method)

julia> foo()

julia> ENV["JULIA_DEBUG"] = Main
Main

julia> foo()
┌ Debug: foo
└ @ Main REPL[1]:1
```

Use a comma separator to enable debug for multiple modules: `JULIA_DEBUG=loading,Main`.

80.5 Examples

Example: Writing log events to a file

Sometimes it can be useful to write log events to a file. Here is an example of how to use a task-local and global logger to write information to a text file:

```
# Load the logging module
julia> using Logging

# Open a textfile for writing
julia> io = open("log.txt", "w+")
IOStream(<file log.txt>)

# Create a simple logger
julia> logger = SimpleLogger(io)
SimpleLogger(IOStream(<file log.txt>), Info, Dict{Any,Int64}())

# Log a task-specific message
julia> with_logger(logger) do
           @info("a context specific log message")
       end

# Write all buffered messages to the file
julia> flush(io)

# Set the global logger to logger
julia> global_logger(logger)
SimpleLogger(IOStream(<file log.txt>), Info, Dict{Any,Int64}())

# This message will now also be written to the file
julia> @info("a global log message")

# Close the file
julia> close(io)
```

Example: Enable debug-level messages

Here is an example of creating a `ConsoleLogger` that lets through any messages with log level higher than, or equal, to `Logging.Debug`.

```
julia> using Logging

# Create a ConsoleLogger that prints any log messages with level >= Debug to stderr
julia> debuglogger = ConsoleLogger(stderr, Logging.Debug)

# Enable debuglogger for a task
julia> with_logger(debuglogger) do
           @debug "a context specific log message"
       end

# Set the global logger
julia> global_logger(debuglogger)
```

80.6 Reference

Logging module

`Logging.Logging` – Module.

Utilities for capturing, filtering and presenting streams of log events. Normally you don't need to import Logging to create log events; for this the standard logging macros such as @info are already exported by Base and available by default.

Creating events

Logging.@logmsg – Macro.

```
@debug message  [key=value | value ...]
@info  message  [key=value | value ...]
@warn  message  [key=value | value ...]
@error message  [key=value | value ...]

@logmsg level message [key=value | value ...]
```

Create a log record with an informational message. For convenience, four logging macros @debug, @info, @warn and @error are defined which log at the standard severity levels Debug, Info, Warn and Error. @logmsg allows level to be set programmatically to any LogLevel or custom log level types.

message should be an expression which evaluates to a string which is a human readable description of the log event. By convention, this string will be formatted as markdown when presented.

The optional list of key=value pairs supports arbitrary user defined metadata which will be passed through to the logging backend as part of the log record. If only a value expression is supplied, a key representing the expression will be generated using Symbol. For example, x becomes x=x, and foo(10) becomes Symbol("foo(10)")=foo(10). For splatting a list of key value pairs, use the normal splatting syntax, @info "blah" kws....

There are some keys which allow automatically generated log data to be overridden:

- _module=mod can be used to specify a different originating module from the source location of the message.
- _group=symbol can be used to override the message group (this is normally derived from the base name of the source file).
- _id=symbol can be used to override the automatically generated unique message identifier. This is useful if you need to very closely associate messages generated on different source lines.
- _file=string and _line=integer can be used to override the apparent source location of a log message.

There's also some key value pairs which have conventional meaning:

- maxlog=integer should be used as a hint to the backend that the message should be displayed no more than maxlog times.
- exception=ex should be used to transport an exception with a log message, often used with @error. An associated backtrace bt may be attached using the tuple exception=(ex,bt).

Examples

```
@debug "Verbose debugging information.  Invisible by default"
@info  "An informational message"
@warn  "Something was odd.  You should pay attention"
```

```
@error "A non fatal error occurred"

x = 10
@info "Some variables attached to the message" x a=42.0

@debug begin
    sA = sum(A)
    "sum(A) = $sA is an expensive operation, evaluated only when `shouldlog` returns true"
end

for i=1:10000
    @info "With the default backend, you will only see (i = $i) ten times"  maxlog=10
    @debug "Algorithm1" i progress=i/10000
end
```

source

`Logging.LogLevel` – Type.

```
LogLevel(level)
```

Severity/verbosity of a log record.

The log level provides a key against which potential log records may be filtered, before any other work is done to construct the log record data structure itself.

Examples

```
julia> Logging.LogLevel(0) == Logging.Info
true
```

source

`Logging.Debug` – Constant.

```
Debug
```

Alias for `LogLevel(-1000)`.

`Logging.Info` – Constant.

```
Info
```

Alias for `LogLevel(0)`.

`Logging.Warn` – Constant.

```
Warn
```

Alias for LogLevel(1000).

Logging.Error – Constant.

```
Error
```

Alias for LogLevel(2000).

Processing events with AbstractLogger

Event processing is controlled by overriding functions associated with AbstractLogger:

Methods to implement		Brief description
Logging.handle_message		Handle a log event
Logging.shouldlog		Early filtering of events
Logging.min_enabled_level		Lower bound for log level of accepted events
Optional methods	**Default definition**	**Brief description**
Logging.catch_exceptions	true	Catch exceptions during event evaluation

Logging.AbstractLogger – Type.

A logger controls how log records are filtered and dispatched. When a log record is generated, the logger is the first piece of user configurable code which gets to inspect the record and decide what to do with it.

source

Logging.handle_message – Function.

```
handle_message(logger, level, message, _module, group, id, file, line; key1=val1, ...)
```

Log a message to logger at level. The logical location at which the message was generated is given by module _module and group; the source location by file and line. id is an arbitrary unique value (typically a Symbol) to be used as a key to identify the log statement when filtering.

source

Logging.shouldlog – Function.

```
shouldlog(logger, level, _module, group, id)
```

Return true when logger accepts a message at level, generated for _module, group and with unique log identifier id.

source

Logging.min_enabled_level – Function.

```
min_enabled_level(logger)
```

Return the minimum enabled level for `logger` for early filtering. That is, the log level below or equal to which all messages are filtered.

source

`Logging.catch_exceptions` – Function.

```
catch_exceptions(logger)
```

Return `true` if the logger should catch exceptions which happen during log record construction. By default, messages are caught

By default all exceptions are caught to prevent log message generation from crashing the program. This lets users confidently toggle little-used functionality - such as debug logging - in a production system.

If you want to use logging as an audit trail you should disable this for your logger type.

source

`Logging.disable_logging` – Function.

```
disable_logging(level)
```

Disable all log messages at log levels equal to or less than `level`. This is a *global* setting, intended to make debug logging extremely cheap when disabled.

Examples

```
Logging.disable_logging(Logging.Info) # Disable debug and info
```

source

Using Loggers

Logger installation and inspection:

`Logging.global_logger` – Function.

```
global_logger()
```

Return the global logger, used to receive messages when no specific logger exists for the current task.

```
global_logger(logger)
```

Set the global logger to `logger`, and return the previous global logger.

source

`Logging.with_logger` – Function.

```
with_logger(function, logger)
```

Execute `function`, directing all log messages to `logger`.

Example

```
function test(x)
    @info "x = $x"
end

with_logger(logger) do
    test(1)
    test([1,2])
end
```

source

`Logging.current_logger` – Function.

```
current_logger()
```

Return the logger for the current task, or the global logger if none is attached to the task.

source

Loggers that are supplied with the system:

`Logging.NullLogger` – Type.

```
NullLogger()
```

Logger which disables all messages and produces no output - the logger equivalent of /dev/null.

source

`Logging.ConsoleLogger` – Type.

```
ConsoleLogger([stream,] min_level=Info; meta_formatter=default_metafmt,
              show_limited=true, right_justify=0)
```

Logger with formatting optimized for readability in a text console, for example interactive work with the Julia REPL.

Log levels less than `min_level` are filtered out.

Message formatting can be controlled by setting keyword arguments:

- `meta_formatter` is a function which takes the log event metadata (`level, _module, group, id, file, line`) and returns a color (as would be passed to printstyled), prefix and suffix for the log message. The default is to prefix with the log level and a suffix containing the module, file and line location.
- `show_limited` limits the printing of large data structures to something which can fit on the screen by setting the `:limit` `IOContext` key during formatting.
- `right_justify` is the integer column which log metadata is right justified at. The default is zero (metadata goes on its own line).

`Logging.SimpleLogger` – Type.

```
SimpleLogger([stream,] min_level=Info)
```

Simplistic logger for logging all messages with level greater than or equal to `min_level` to `stream`. If stream is closed then messages with log level greater or equal to `Warn` will be logged to `stderr` and below to `stdout`.

source

Chapter 81

Markdown

本节描述 Julia 的 markdown 语法，它是由 Markdown 标准库启用的。它支持以下的 Markdown 元素:

81.1 内联元素

此处的“内联”指可以在段落中找到的元素。包括下面的元素。

粗体

用两个 ** 包围来将其内部的文本显示为粗体。

```
A paragraph containing a **bold** word.
```

斜体

用单个 * 包围来将其内部的文本显示为斜体。

```
A paragraph containing an *italicized* word.
```

文字

用一个重音符号 ` 包围的文本将会原封不动地显示出来。

```
A paragraph containing a `literal` word.
```

当文本指代变量名、函数名或者 Julia 程序的其他部分时，应当使用字面量。

Tip

为了在字面量中包含一个重音符，需要使用三个重音符而不是一个来包围文本。

```
A paragraph containing ``` `backtick` characters ```.
```

通过扩展，可以使用任何奇数个反引号来包围较少数量的反引号。

LaTeX

使用两个重音符的 LaTeX 语法来包围那些是数学表达式的文本，``.

```
A paragraph containing some ``\LaTeX`` markup.
```

> **Tip**
>
> As with literals in the previous section, if literal backticks need to be written within double backticks use an even number greater than two. Note that if a single literal backtick needs to be included within LaTeXmarkup then two enclosing backticks is sufficient.

> **Note**
>
> The \ character should be escaped appropriately if the text is embedded in a Julia source code, for example, `"``\\LaTeX`` syntax in a docstring."`, since it is interpreted as a string literal. Alternatively, in order to avoid escaping, it is possible to use the raw string macro together with the @doc macro:
>
> ```
> @doc raw"``\LaTeX`` syntax in a docstring." functionname
> ```

Links

Links to either external or internal targets can be written using the following syntax, where the text enclosed in square brackets, [ ], is the name of the link and the text enclosed in parentheses, ( ), is the URL.

```
A paragraph containing a link to [Julia](http://www.julialang.org).
```

It's also possible to add cross-references to other documented functions/methods/variables within the Julia documentation itself. For example:

```
"""
    tryparse(type, str; base)

Like [`parse`](@ref), but returns either a value of the requested type,
or [`nothing`](@ref) if the string does not contain a valid number.
"""
```

This will create a link in the generated docs to the `parse` documentation (which has more information about what this function actually does), and to the `nothing` documentation. It's good to include cross references to mutating/non-mutating versions of a function, or to highlight a difference between two similar-seeming functions.

> **Note**
>
> The above cross referencing is *not* a Markdown feature, and relies on Documenter.jl, which is used to build base Julia's documentation.

Footnote references

Named and numbered footnote references can be written using the following syntax. A footnote name must be a single alphanumeric word containing no punctuation.

```
A paragraph containing a numbered footnote [^1] and a named one [^named].
```

Note

The text associated with a footnote can be written anywhere within the same page as the footnote reference. The syntax used to define the footnote text is discussed in the Footnotes section below.

81.2 Toplevel elements

The following elements can be written either at the ”toplevel” of a document or within another ”toplevel” element.

Paragraphs

A paragraph is a block of plain text, possibly containing any number of inline elements defined in the Inline elements section above, with one or more blank lines above and below it.

```
This is a paragraph.

And this is *another* paragraph containing some emphasized text.
A new line, but still part of the same paragraph.
```

Headers

A document can be split up into different sections using headers. Headers use the following syntax:

```
# Level One
## Level Two
### Level Three
#### Level Four
##### Level Five
###### Level Six
```

A header line can contain any inline syntax in the same way as a paragraph can.

Tip

Try to avoid using too many levels of header within a single document. A heavily nested document may be indicative of a need to restructure it or split it into several pages covering separate topics.

Code blocks

Source code can be displayed as a literal block using an indent of four spaces as shown in the following example.

```
This is a paragraph.

    function func(x)
        # ...
    end

Another paragraph.
```

Additionally, code blocks can be enclosed using triple backticks with an optional ”language” to specify how a block of code should be highlighted.

````
A code block without a "language":

```
function func(x)
    # ...
end
```

and another one with the "language" specified as `julia`:

```julia
function func(x)
    # ...
end
```
````

Note

”Fenced” code blocks, as shown in the last example, should be preferred over indented code blocks since there is no way to specify what language an indented code block is written in.

Block quotes

Text from external sources, such as quotations from books or websites, can be quoted using > characters prepended to each line of the quote as follows.

```
Here's a quote:

> Julia is a high-level, high-performance dynamic programming language for
> technical computing, with syntax that is familiar to users of other
> technical computing environments.
```

Note that a single space must appear after the > character on each line. Quoted blocks may themselves contain other toplevel or inline elements.

Images

The syntax for images is similar to the link syntax mentioned above. Prepending a ! character to a link will display an image from the specified URL rather than a link to it.

```
![alternative text](link/to/image.png)
```

Lists

Unordered lists can be written by prepending each item in a list with either *, +, or -.

```
A list of items:

  * item one
  * item two
  * item three
```

Note the two spaces before each * and the single space after each one.

Lists can contain other nested toplevel elements such as lists, code blocks, or quoteblocks. A blank line should be left between each list item when including any toplevel elements within a list.

````
Another list:

  * item one

  * item two

    ```
    f(x) = x
    ```

  * And a sublist:

      + sub-item one
      + sub-item two
````

Note

The contents of each item in the list must line up with the first line of the item. In the above example the fenced code block must be indented by four spaces to align with the `i` in `item two`.

Ordered lists are written by replacing the ”bullet” character, either *, +, or -, with a positive integer followed by either `.` or `)`.

```
Two ordered lists:

 1. item one
 2. item two
 3. item three

 5) item five
 6) item six
 7) item seven
```

An ordered list may start from a number other than one, as in the second list of the above example, where it is numbered from five. As with unordered lists, ordered lists can contain nested toplevel elements.

Display equations

Large LaTeX equations that do not fit inline within a paragraph may be written as display equations using a fenced code block with the "language" `math` as in the example below.

````
```math
f(a) = \frac{1}{2\pi}\int_{0}^{2\pi} (\alpha+R\cos(\theta))d\theta
```
````

Footnotes

This syntax is paired with the inline syntax for Footnote references. Make sure to read that section as well.

Footnote text is defined using the following syntax, which is similar to footnote reference syntax, aside from the `:` character that is appended to the footnote label.

````
[^1]: Numbered footnote text.

[^note]:

    Named footnote text containing several toplevel elements.

      * item one
      * item two
      * item three

    ```julia
    function func(x)
        # ...
    end
    ```
````

Note

No checks are done during parsing to make sure that all footnote references have matching footnotes.

Horizontal rules

The equivalent of an `<hr>` HTML tag can be achieved using three hyphens (`---`). For example:

```
Text above the line.

---

And text below the line.
```

Tables

Basic tables can be written using the syntax described below. Note that markdown tables have limited features and cannot contain nested toplevel elements unlike other elements discussed above –only inline elements are allowed. Tables must always contain a header row with column names. Cells cannot span multiple rows or columns of the table.

```
| Column One | Column Two | Column Three |
|:---------- | ---------- |:------------:|
| Row `1`    | Column `2` |              |
| *Row* 2    | **Row** 2  | Column ``3`` |
```

Note

As illustrated in the above example each column of `|` characters must be aligned vertically.

A `:` character on either end of a column's header separator (the row containing `-` characters) specifies whether the row is left-aligned, right-aligned, or (when `:` appears on both ends) center-aligned. Providing no `:` characters will default to right-aligning the column.

Admonitions

Specially formatted blocks, known as admonitions, can be used to highlight particular remarks. They can be defined using the following `!!!` syntax:

```
!!! note

    This is the content of the note.

!!! warning "Beware!"

    And this is another one.

    This warning admonition has a custom title: `"Beware!"`.
```

The first word after `!!!` declares the type of the admonition. There are standard admonition types that should produce special styling. Namely (in order of decreasing severity): `danger`, `warning`, `info`/`note`, and `tip`.

You can also use your own admonition types, as long as the type name only contains lowercase Latin characters (a-z). For example, you could have a `terminology` block like this:

```
!!! terminology "julia vs Julia"

    Strictly speaking, "Julia" refers to the language,
    and "julia" to the standard implementation.
```

However, unless the code rendering the Markdown special-cases that particular admonition type, it will get the default styling.

A custom title for the box can be provided as a string (in double quotes) after the admonition type. If no title text is specified after the admonition type, then the type name will be used as the title (e.g. `"Note"` for the `note` admonition).

Admonitions, like most other toplevel elements, can contain other toplevel elements (e.g. lists, images).

81.3 Markdown Syntax Extensions

Julia's markdown supports interpolation in a very similar way to basic string literals, with the difference that it will store the object itself in the Markdown tree (as opposed to converting it to a string). When the Markdown

content is rendered the usual `show` methods will be called, and these can be overridden as usual. This design allows the Markdown to be extended with arbitrarily complex features (such as references) without cluttering the basic syntax.

In principle, the Markdown parser itself can also be arbitrarily extended by packages, or an entirely custom flavour of Markdown can be used, but this should generally be unnecessary.

Chapter 82

内存映射 I/O

Low level module for mmap (memory mapping of files).

Mmap.Anonymous – Type.

```
Mmap.Anonymous(name::AbstractString="", readonly::Bool=false, create::Bool=true)
```

Create an IO-like object for creating zeroed-out mmapped-memory that is not tied to a file for use in mmap. Used by SharedArray for creating shared memory arrays.

Examples

```
julia> using Mmap

julia> anon = Mmap.Anonymous();

julia> isreadable(anon)
true

julia> iswritable(anon)
true

julia> isopen(anon)
true
```

Mmap.mmap – Function.

```
mmap(io::Union{IOStream,AbstractString,Mmap.AnonymousMmap}[, type::Type{Array{T,N}}, dims,
↪  offset]; grow::Bool=true, shared::Bool=true)
mmap(type::Type{Array{T,N}}, dims)
```

Create an Array whose values are linked to a file, using memory-mapping. This provides a convenient way of working with data too large to fit in the computer's memory.

The type is an Array{T,N} with a bits-type element of T and dimension N that determines how the bytes of the array are interpreted. Note that the file must be stored in binary format, and no format conversions are possible (this is a limitation of operating systems, not Julia).

`dims` is a tuple or single `Integer` specifying the size or length of the array.

The file is passed via the stream argument, either as an open `IOStream` or filename string. When you initialize the stream, use `"r"` for a "read-only" array, and `"w+"` to create a new array used to write values to disk.

If no `type` argument is specified, the default is `Vector{UInt8}`.

Optionally, you can specify an offset (in bytes) if, for example, you want to skip over a header in the file. The default value for the offset is the current stream position for an `IOStream`.

The `grow` keyword argument specifies whether the disk file should be grown to accommodate the requested size of array (if the total file size is < requested array size). Write privileges are required to grow the file.

The `shared` keyword argument specifies whether the resulting `Array` and changes made to it will be visible to other processes mapping the same file.

For example, the following code

```
# Create a file for mmapping
# (you could alternatively use mmap to do this step, too)
using Mmap
A = rand(1:20, 5, 30)
s = open("/tmp/mmap.bin", "w+")
# We'll write the dimensions of the array as the first two Ints in the file
write(s, size(A,1))
write(s, size(A,2))
# Now write the data
write(s, A)
close(s)

# Test by reading it back in
s = open("/tmp/mmap.bin")   # default is read-only
m = read(s, Int)
n = read(s, Int)
A2 = mmap(s, Matrix{Int}, (m,n))
```

creates a `m`-by-`n` `Matrix{Int}`, linked to the file associated with stream `s`.

A more portable file would need to encode the word size –32 bit or 64 bit –and endianness information in the header. In practice, consider encoding binary data using standard formats like HDF5 (which can be used with memory-mapping).

```
mmap(io, BitArray, [dims, offset])
```

Create a `BitArray` whose values are linked to a file, using memory-mapping; it has the same purpose, works in the same way, and has the same arguments, as `mmap`, but the byte representation is different.

Examples

```
julia> using Mmap

julia> io = open("mmap.bin", "w+");

julia> B = mmap(io, BitArray, (25,30000));
```

```
julia> B[3, 4000] = true;

julia> Mmap.sync!(B);

julia> close(io);

julia> io = open("mmap.bin", "r+");

julia> C = mmap(io, BitArray, (25,30000));

julia> C[3, 4000]
true

julia> C[2, 4000]
false

julia> close(io)

julia> rm("mmap.bin")
```

This creates a 25-by-30000 `BitArray`, linked to the file associated with stream `io`.

`Mmap.sync!` – Function.

```
Mmap.sync!(array)
```

Forces synchronization between the in-memory version of a memory-mapped `Array` or `BitArray` and the on-disk version.

Chapter 83

Network Options

`NetworkOptions.ca_roots` – Function.

```
ca_roots() :: Union{Nothing, String}
```

The `ca_roots()` function tells the caller where, if anywhere, to find a file or directory of PEM-encoded certificate authority roots. By default, on systems like Windows and macOS where the built-in TLS engines know how to verify hosts using the system's built-in certificate verification mechanism, this function will return `nothing`. On classic UNIX systems (excluding macOS), root certificates are typically stored in a file in `/etc`: the common places for the current UNIX system will be searched and if one of these paths exists, it will be returned; if none of these typical root certificate paths exist, then the path to the set of root certificates that are bundled with Julia is returned.

The default value returned by `ca_roots()` may be overridden by setting the `JULIA_SSL_CA_ROOTS_PATH`, `SSL_CERT_DIR`, or `SSL_CERT_FILE` environment variables, in which case this function will always return the value of the first of these variables that is set (whether the path exists or not). If `JULIA_SSL_CA_ROOTS_PATH` is set to the empty string, then the other variables are ignored (as if unset); if the other variables are set to the empty string, they behave is if they are not set.

`NetworkOptions.ca_roots_path` – Function.

```
ca_roots_path() :: String
```

The `ca_roots_path()` function is similar to the `ca_roots()` function except that it always returns a path to a file or directory of PEM-encoded certificate authority roots. When called on a system like Windows or macOS, where system root certificates are not stored in the file system, it will currently return the path to the set of root certificates that are bundled with Julia. (In the future, this function may instead extract the root certificates from the system and save them to a file whose path would be returned.)

If it is possible to configure a library that uses TLS to use the system certificates that is generally preferable: i.e. it is better to use `ca_roots()` which returns `nothing` to indicate that the system certs should be used. The `ca_roots_path()` function should only be used when configuring libraries which *require* a path to a file or directory for root certificates.

The default value returned by `ca_roots_path()` may be overridden by setting the `JULIA_SSL_CA_ROOTS_PATH`, `SSL_CERT_DIR`, or `SSL_CERT_FILE` environment variables, in which case this function will always return the value of the first of these variables that is set (whether the path exists or not). If `JULIA_SSL_CA_ROOTS_PATH`

is set to the empty string, then the other variables are ignored (as if unset); if the other variables are set to the empty string, they behave is if they are not set.

`NetworkOptions.ssh_dir` – Function.

```
ssh_dir() :: String
```

The `ssh_dir()` function returns the location of the directory where the `ssh` program keeps/looks for configuration files. By default this is `~/.ssh` but this can be overridden by setting the environment variable `SSH_DIR`.

`NetworkOptions.ssh_key_pass` – Function.

```
ssh_key_pass() :: String
```

The `ssh_key_pass()` function returns the value of the environment variable `SSH_KEY_PASS` if it is set or `nothing` if it is not set. In the future, this may be able to find a password by other means, such as secure system storage, so packages that need a password to decrypt an SSH private key should use this API instead of directly checking the environment variable so that they gain such capabilities automatically when they are added.

`NetworkOptions.ssh_key_name` – Function.

```
ssh_key_name() :: String
```

The `ssh_key_name()` function returns the base name of key files that SSH should use for when establishing a connection. There is usually no reason that this function should be called directly and libraries should generally use the `ssh_key_path` and `ssh_pub_key_path` functions to get full paths. If the environment variable `SSH_KEY_NAME` is set then this function returns that; otherwise it returns `id_rsa` by default.

`NetworkOptions.ssh_key_path` – Function.

```
ssh_key_path() :: String
```

The `ssh_key_path()` function returns the path of the SSH private key file that should be used for SSH connections. If the `SSH_KEY_PATH` environment variable is set then it will return that value. Otherwise it defaults to returning

```
joinpath(ssh_dir(), ssh_key_name())
```

This default value in turn depends on the `SSH_DIR` and `SSH_KEY_NAME` environment variables.

`NetworkOptions.ssh_pub_key_path` – Function.

```
ssh_pub_key_path() :: String
```

The `ssh_pub_key_path()` function returns the path of the SSH public key file that should be used for SSH connections. If the `SSH_PUB_KEY_PATH` environment variable is set then it will return that value. If that isn't set but `SSH_KEY_PATH` is set, it will return that path with the `.pub` suffix appended. If neither is set, it defaults to returning

```
joinpath(ssh_dir(), ssh_key_name() * ".pub")
```

This default value in turn depends on the `SSH_DIR` and `SSH_KEY_NAME` environment variables.

`NetworkOptions.ssh_known_hosts_files` – Function.

```
ssh_known_hosts_files() :: Vector{String}
```

The `ssh_known_hosts_files()` function returns a vector of paths of SSH known hosts files that should be used when establishing the identities of remote servers for SSH connections. By default this function returns

```
[joinpath(ssh_dir(), "known_hosts"), bundled_known_hosts]
```

where `bundled_known_hosts` is the path of a copy of a known hosts file that is bundled with this package (containing known hosts keys for `github.com` and `gitlab.com`). If the environment variable `SSH_KNOWN_HOSTS_FILES` is set, however, then its value is split into paths on the `:` character (or on `;` on Windows) and this vector of paths is returned instead. If any component of this vector is empty, it is expanded to the default known hosts paths.

Packages that use `ssh_known_hosts_files()` should ideally look for matching entries by comparing the host name and key types, considering the first entry in any of the files which matches to be the definitive identity of the host. If the caller cannot compare the key type (e.g. because it has been hashes) then it must approximate the above algorithm by looking for all matching entries for a host in each file: if a file has any entries for a host then one of them must match; the caller should only continue to search further known hosts files if there are no entries for the host in question in an earlier file.

`NetworkOptions.ssh_known_hosts_file` – Function.

```
ssh_known_hosts_file() :: String
```

The `ssh_known_hosts_file()` function returns a single path of an SSH known hosts file that should be used when establishing the identities of remote servers for SSH connections. It returns the first path returned by `ssh_known_hosts_files` that actually exists. Callers who can look in more than one known hosts file should use `ssh_known_hosts_files` instead and look for host matches in all the files returned as described in that function's docs.

`NetworkOptions.verify_host` – Function.

```
verify_host(url::AbstractString, [transport::AbstractString]) :: Bool
```

The `verify_host` function tells the caller whether the identity of a host should be verified when communicating over secure transports like TLS or SSH. The `url` argument may be:

1. a proper URL staring with `proto://`
2. an `ssh`-style bare host name or host name prefixed with `user@`
3. an `scp`-style host as above, followed by `:` and a path location

In each case the host name part is parsed out and the decision about whether to verify or not is made based solely on the host name, not anything else about the input URL. In particular, the protocol of the URL does not matter (more below).

The `transport` argument indicates the kind of transport that the query is about. The currently known values are SSL/`ssl` (alias TLS/`tls`) and SSH/`ssh`. If the transport is omitted, the query will return `true` only if the host name should not be verified regardless of transport.

The host name is matched against the host patterns in the relevant environment variables depending on whether `transport` is supplied and what its value is:

- `JULIA_NO_VERIFY_HOSTS` — hosts that should not be verified for any transport
- `JULIA_SSL_NO_VERIFY_HOSTS` — hosts that should not be verified for SSL/TLS
- `JULIA_SSH_NO_VERIFY_HOSTS` — hosts that should not be verified for SSH
- `JULIA_ALWAYS_VERIFY_HOSTS` —hosts that should always be verified

The values of each of these variables is a comma-separated list of host name patterns with the following syntax —each pattern is split on `.` into parts and each part must one of:

1. A literal domain name component consisting of one or more ASCII letter, digit, hyphen or underscore (technically not part of a legal host name, but sometimes used). A literal domain name component matches only itself.
2. A **, which matches zero or more domain name components.
3. A *, which match any one domain name component.

When matching a host name against a pattern list in one of these variables, the host name is split on `.` into components and that sequence of words is matched against the pattern: a literal pattern matches exactly one host name component with that value; a * pattern matches exactly one host name component with any value; a ** pattern matches any number of host name components. For example:

- ** matches any host name
- `**.org` matches any host name in the `.org` top-level domain
- `example.com` matches only the exact host name `example.com`
- `*.example.com` matches `api.example.com` but not `example.com` or `v1.api.example.com`
- `**.example.com` matches any domain under `example.com`, including `example.com` itself, `api.example.com` and `v1.api.example.com`

Chapter 84

Pkg

Pkg is Julia's builtin package manager, and handles operations such as installing, updating and removing packages.

> **Note**
>
> What follows is a very brief introduction to Pkg. For more information on `Project.toml` files, `Manifest.toml` files, package version compatibility (`[compat]`), environments, registries, etc., it is highly recommended to read the full manual, which is available here: https://pkgdocs.julialang.org.

What follows is a quick overview of the basic features of Pkg. It should help new users become familiar with basic Pkg features such as adding and removing packages and working with environments.

> **Note**
>
> Some Pkg output is omitted in this section in order to keep this basic guide focused. This will help maintain a good pace and not get bogged down in details. If you require more details, refer to subsequent sections of the Pkg manual.

> **Note**
>
> This guide uses the Pkg REPL to execute Pkg commands. For non-interactive use, we recommend the Pkg API. The Pkg API is fully documented in the API Reference section of the Pkg documentation.

Pkg comes with a REPL. Enter the Pkg REPL by pressing `]` from the Julia REPL. To get back to the Julia REPL, press `Ctrl+C` or backspace (when the REPL cursor is at the beginning of the input).

Upon entering the Pkg REPL, you should see the following prompt:

```
(@v1.8) pkg>
```

To add a package, use add:

```
(@v1.8) pkg> add Example
   Resolving package versions...
   Installed Example ─ v0.5.3
    Updating `~/.julia/environments/v1.8/Project.toml`
```

```
  [7876af07] + Example v0.5.3
    Updating `~/.julia/environments/v1.8/Manifest.toml`
  [7876af07] + Example v0.5.3
```

After the package is installed, it can be loaded into the Julia session:

```
julia> import Example

julia> Example.hello("friend")
"Hello, friend"
```

We can also specify multiple packages at once to install:

```
(@v1.8) pkg> add JSON StaticArrays
```

The `status` command (or the shorter `st` command) can be used to see installed packages.

```
(@v1.8) pkg> st
Status `~/.julia/environments/v1.6/Project.toml`
  [7876af07] Example v0.5.3
  [682c06a0] JSON v0.21.3
  [90137ffa] StaticArrays v1.5.9
```

Note

Some Pkg REPL commands have a short and a long version of the command, for example `status` and `st`.

To remove packages, use `rm` (or `remove`):

```
(@v1.8) pkg> rm JSON StaticArrays
```

Use up (or `update`) to update the installed packages

```
(@v1.8) pkg> up
```

If you have been following this guide it is likely that the packages installed are at the latest version so up will not do anything. Below we show the status output in the case where we deliberately have installed an old version of the Example package and then upgrade it:

```
(@v1.8) pkg> st
Status `~/.julia/environments/v1.8/Project.toml`
^ [7876af07] Example v0.5.1
Info Packages marked with ^ have new versions available and may be upgradable.

(@v1.8) pkg> up
    Updating `~/.julia/environments/v1.8/Project.toml`
  [7876af07] ↑ Example v0.5.1 ⇒ v0.5.3
```

We can see that the status output tells us that there is a newer version available and that up upgrades the package.

For more information about managing packages, see the Managing Packages section of the documentation.

Up to this point, we have covered basic package management: adding, updating, and removing packages.

You may have noticed the `(@v1.8)` in the REPL prompt. This lets us know that `v1.8` is the **active environment**. Different environments can have different totally different packages and versions installed from another environment. The active environment is the environment that will be modified by Pkg commands such as `add`, `rm` and `update`.

Let's set up a new environment so we may experiment. To set the active environment, use `activate`:

```
(@v1.8) pkg> activate tutorial
[ Info: activating new environment at `~/tutorial/Project.toml`.
```

Pkg lets us know we are creating a new environment and that this environment will be stored in the `~/tutorial` directory. The path to the environment is created relative to the current working directory of the REPL.

Pkg has also updated the REPL prompt in order to reflect the new active environment:

```
(tutorial) pkg>
```

We can ask for information about the active environment by using `status`:

```
(tutorial) pkg> status
    Status `~/tutorial/Project.toml`
   (empty environment)
```

`~/tutorial/Project.toml` is the location of the active environment's **project file**. A project file is a TOML file where Pkg stores the packages that have been explicitly installed. Notice this new environment is empty. Let us add some packages and observe:

```
(tutorial) pkg> add Example JSON
...

(tutorial) pkg> status
    Status `~/tutorial/Project.toml`
  [7876af07] Example v0.5.3
  [682c06a0] JSON v0.21.3
```

We can see that the `tutorial` environment now contains `Example` and `JSON`.

Note

If you have the same package (at the same version) installed in multiple environments, the package will only be downloaded and stored on the hard drive once. This makes environments very lightweight and effectively free to create. Using only the default environment with a huge number of packages in it is a common beginners mistake in Julia. Learning how to use environments effectively will improve your experience with Julia packages.

For more information about environments, see the Working with Environments section of the documentation.

If you are ever stuck, you can ask Pkg for help:

```
(@v1.8) pkg> ?
```

You should see a list of available commands along with short descriptions. You can ask for more detailed help by specifying a command:

```
(@v1.8) pkg> ?develop
```

This guide should help you get started with Pkg. Pkg has much more to offer in terms of powerful package management, read the full manual to learn more!

Chapter 85

Printf

`Printf.@printf` – Macro.

```
@printf([io::IO], "%Fmt", args...)
```

Print `args` using C `printf` style format specification string. Optionally, an `IO` may be passed as the first argument to redirect output.

Examples

```
julia> @printf "Hello %s" "world"
Hello world

julia> @printf "Scientific notation %e" 1.234
Scientific notation 1.234000e+00

julia> @printf "Scientific notation three digits %.3e" 1.23456
Scientific notation three digits 1.235e+00

julia> @printf "Decimal two digits %.2f" 1.23456
Decimal two digits 1.23

julia> @printf "Padded to length 5 %5i" 123
Padded to length 5   123

julia> @printf "Padded with zeros to length 6 %06i" 123
Padded with zeros to length 6 000123

julia> @printf "Use shorter of decimal or scientific %g %g" 1.23 12300000.0
Use shorter of decimal or scientific 1.23 1.23e+07

julia> @printf "Use dynamic width and precision  %*.*f" 10 2 0.12345
Use dynamic width and precision        0.12
```

For a systematic specification of the format, see here. See also `@sprintf` to get the result as a `String` instead of it being printed.

Caveats

Inf and NaN are printed consistently as Inf and NaN for flags %a, %A, %e, %E, %f, %F, %g, and %G. Furthermore, if a floating point number is equally close to the numeric values of two possible output strings, the output string further away from zero is chosen.

Examples

```
julia> @printf("%f %F %f %F", Inf, Inf, NaN, NaN)
Inf Inf NaN NaN

julia> @printf "%.0f %.1f %f" 0.5 0.025 -0.0078125
0 0.0 -0.007812
```

Julia 1.8

Starting in Julia 1.8, %s (string) and %c (character) widths are computed using textwidth, which e.g. ignores zero-width characters (such as combining characters for diacritical marks) and treats certain "wide" characters (e.g. emoji) as width 2.

Julia 1.10

Dynamic width specifiers like %*s and %0*.*f require Julia 1.10.

Printf.@sprintf – Macro.

```
@sprintf("%Fmt", args...)
```

Return @printf formatted output as string.

Examples

```
julia> @sprintf "this is a %s %15.1f" "test" 34.567
"this is a test            34.6"
```

Chapter 86

性能分析

86.1 CPU Profiling

There are two main approaches to CPU profiling julia code:

86.2 Via `@profile`

Where profiling is enabled for a given call via the `@profile` macro.

```
julia> using Profile

julia> @profile foo()

julia> Profile.print()
Overhead ╎ [+additional indent] Count File:Line; Function
=========================================================
    ╎147  @Base/client.jl:506; _start()
        ╎ 147  @Base/client.jl:318; exec_options(opts::Base.JLOptions)
...
```

86.3 Triggered During Execution

Tasks that are already running can also be profiled for a fixed time period at any user-triggered time.

To trigger the profiling:

- MacOS & FreeBSD (BSD-based platforms): Use `ctrl-t` or pass a `SIGINFO` signal to the julia process i.e. `% kill -INFO $julia_pid`
- Linux: Pass a `SIGUSR1` signal to the julia process i.e. `% kill -USR1 $julia_pid`
- Windows: Not currently supported.

First, a single stack trace at the instant that the signal was thrown is shown, then a 1 second profile is collected, followed by the profile report at the next yield point, which may be at task completion for code without yield points e.g. tight loops.

Optionally set environment variable `JULIA_PROFILE_PEEK_HEAP_SNAPSHOT` to `1` to also automatically collect a heap snapshot.

```
julia> foo()
##== the user sends a trigger while foo is running ==##
load: 2.53  cmd: julia 88903 running 6.16u 0.97s

======================================================================================
Information request received. A stacktrace will print followed by a 1.0 second profile
======================================================================================

signal (29): Information request: 29
__psynch_cvwait at /usr/lib/system/libsystem_kernel.dylib (unknown line)
_pthread_cond_wait at /usr/lib/system/libsystem_pthread.dylib (unknown line)
...

======================================================================
Profile collected. A report will print if the Profile module is loaded
======================================================================

Overhead ╎ [+additional indent] Count File:Line; Function
=========================================================
Thread 1 Task 0x000000011687c010 Total snapshots: 572. Utilization: 100%
   ╎147 @Base/client.jl:506; _start()
       ╎ 147 @Base/client.jl:318; exec_options(opts::Base.JLOptions)
...

Thread 2 Task 0x0000000116960010 Total snapshots: 572. Utilization: 0%
   ╎572 @Base/task.jl:587; task_done_hook(t::Task)
      ╎ 572 @Base/task.jl:879; wait()
...
```

Customization

The duration of the profiling can be adjusted via `Profile.set_peek_duration`

The profile report is broken down by thread and task. Pass a no-arg function to `Profile.peek_report[]` to override this. i.e. `Profile.peek_report[] = () -> Profile.print()` to remove any grouping. This could also be overridden by an external profile data consumer.

86.4 Reference

`Profile.@profile` – Macro.

```
@profile
```

`@profile <expression>` runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

source

`Profile` 里的方法均未导出，需要通过 `Profile.print()` 的方式调用。

`Profile.clear` – Function.

```
clear()
```

Clear any existing backtraces from the internal buffer.

source

`Profile.print` – Function.

```
print([io::IO = stdout,] [data::Vector = fetch()], [lidict::Union{LineInfoDict,
↪  LineInfoFlatDict} = getdict(data)]; kwargs...)
```

Prints profiling results to `io` (by default, `stdout`). If you do not supply a `data` vector, the internal buffer of accumulated backtraces will be used.

The keyword arguments can be any combination of:

- `format` —Determines whether backtraces are printed with (default, `:tree`) or without (`:flat`) indentation indicating tree structure.
- `C` —If `true`, backtraces from C and Fortran code are shown (normally they are excluded).
- `combine` —If `true` (default), instruction pointers are merged that correspond to the same line of code.
- `maxdepth` —Limits the depth higher than `maxdepth` in the `:tree` format.
- `sortedby` —Controls the order in `:flat` format. `:filefuncline` (default) sorts by the source line, `:count` sorts in order of number of collected samples, and `:overhead` sorts by the number of samples incurred by each function by itself.
- `groupby` —Controls grouping over tasks and threads, or no grouping. Options are `:none` (default), `:thread`, `:task`, `[:thread, :task]`, or `[:task, :thread]` where the last two provide nested grouping.
- `noisefloor` —Limits frames that exceed the heuristic noise floor of the sample (only applies to format `:tree`). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which `n <= noisefloor * √N`, where `n` is the number of samples on this line, and `N` is the number of samples for the callee.
- `mincount` —Limits the printout to only those lines with at least `mincount` occurrences.
- `recur` —Controls the recursion handling in `:tree` format. `:off` (default) prints the tree as normal. `:flat` instead compresses any recursion (by ip), showing the approximate effect of converting any self-recursion into an iterator. `:flatc` does the same but also includes collapsing of C frames (may do odd things around `jl_apply`).
- `threads::Union{Int,AbstractVector{Int}}` —Specify which threads to include snapshots from in the report. Note that this does not control which threads samples are collected on (which may also have been collected on another machine).
- `tasks::Union{Int,AbstractVector{Int}}` —Specify which tasks to include snapshots from in the report. Note that this does not control which tasks samples are collected within.

Julia 1.8

The `groupby`, `threads`, and `tasks` keyword arguments were introduced in Julia 1.8.

Note

Profiling on windows is limited to the main thread. Other threads have not been sampled and will not show in the report.

source

```
print([io::IO = stdout,] data::Vector, lidict::LineInfoDict; kwargs...)
```

Prints profiling results to `io`. This variant is used to examine results exported by a previous call to `retrieve`. Supply the vector `data` of backtraces and a dictionary `lidict` of line information.

See `Profile.print([io], data)` for an explanation of the valid keyword arguments.

source

`Profile.init` – Function.

```
init(; n::Integer, delay::Real)
```

Configure the `delay` between backtraces (measured in seconds), and the number `n` of instruction pointers that may be stored per thread. Each instruction pointer corresponds to a single line of code; backtraces generally consist of a long list of instruction pointers. Note that 6 spaces for instruction pointers per backtrace are used to store metadata and two NULL end markers. Current settings can be obtained by calling this function with no arguments, and each can be set independently using keywords or in the order `(n, delay)`.

source

`Profile.fetch` – Function.

```
fetch(;include_meta = true) -> data
```

Return a copy of the buffer of profile backtraces. Note that the values in `data` have meaning only on this machine in the current session, because it depends on the exact memory addresses used in JIT-compiling. This function is primarily for internal use; `retrieve` may be a better choice for most users. By default metadata such as threadid and taskid is included. Set `include_meta` to `false` to strip metadata.

source

`Profile.retrieve` – Function.

```
retrieve(; kwargs...) -> data, lidict
```

"Exports" profiling results in a portable format, returning the set of all backtraces (`data`) and a dictionary that maps the (session-specific) instruction pointers in `data` to `LineInfo` values that store the file name, function name, and line number. This function allows you to save profiling results for future analysis.

source

`Profile.callers` – Function.

```
callers(funcname, [data, lidict], [filename=<filename>], [linerange=<start:stop>]) ->
↪  Vector{Tuple{count, lineinfo}}
```

Given a previous profiling run, determine who called a particular function. Supplying the filename (and optionally, range of line numbers over which the function is defined) allows you to disambiguate an overloaded method. The returned value is a vector containing a count of the number of calls and line information about the caller. One can optionally supply backtrace `data` obtained from `retrieve`; otherwise, the current internal profile buffer is used.

source

`Profile.clear_malloc_data` – Function.

```
clear_malloc_data()
```

Clears any stored memory allocation data when running julia with `--track-allocation`. Execute the command(s) you want to test (to force JIT-compilation), then call `clear_malloc_data`. Then execute your command(s) again, quit Julia, and examine the resulting `*.mem` files.

source

`Profile.get_peek_duration` – Function.

```
get_peek_duration()
```

Get the duration in seconds of the profile "peek" that is triggered via `SIGINFO` or `SIGUSR1`, depending on platform.

source

`Profile.set_peek_duration` – Function.

```
set_peek_duration(t::Float64)
```

Set the duration in seconds of the profile "peek" that is triggered via `SIGINFO` or `SIGUSR1`, depending on platform.

source

86.5 Memory profiling

`Profile.Allocs.@profile` – Macro.

```
Profile.Allocs.@profile [sample_rate=0.1] expr
```

Profile allocations that happen during `expr`, returning both the result and and AllocResults struct.

A sample rate of 1.0 will record everything; 0.0 will record nothing.

```
julia> Profile.Allocs.@profile sample_rate=0.01 peakflops()
1.03733270279065e11

julia> results = Profile.Allocs.fetch()

julia> last(sort(results.allocs, by=x->x.size))
Profile.Allocs.Alloc(Vector{Any}, Base.StackTraces.StackFrame[_new_array_ at array.c:127, ...],
↪  5576)
```

The best way to visualize these is currently with the PProf.jl package, by invoking `PProf.Allocs.pprof`.

Note

The current implementation of the Allocations Profiler does not capture types for all allocations. Allocations for which the profiler could not capture the type are represented as having type `Profile.Allocs.UnknownType`.

You can read more about the missing types and the plan to improve this, here: https://github.com/JuliaLang/julia/issues/43688.

Julia 1.8

The allocation profiler was added in Julia 1.8.

source

The methods in `Profile.Allocs` are not exported and need to be called e.g. as `Profile.Allocs.fetch()`.

`Profile.Allocs.clear` – Function.

```
Profile.Allocs.clear()
```

Clear all previously profiled allocation information from memory.

source

`Profile.Allocs.fetch` – Function.

```
Profile.Allocs.fetch()
```

Retrieve the recorded allocations, and decode them into Julia objects which can be analyzed.

source

`Profile.Allocs.start` – Function.

```
Profile.Allocs.start(sample_rate::Real)
```

Begin recording allocations with the given sample rate A sample rate of 1.0 will record everything; 0.0 will record nothing.

source

`Profile.Allocs.stop` – Function.

```
Profile.Allocs.stop()
```

Stop recording allocations.

source

86.6 Heap Snapshots

`Profile.take_heap_snapshot` – Function.

```
Profile.take_heap_snapshot(io::IOStream, all_one::Bool=false)
Profile.take_heap_snapshot(filepath::String, all_one::Bool=false)
Profile.take_heap_snapshot(all_one::Bool=false; dir::String)
```

Write a snapshot of the heap, in the JSON format expected by the Chrome Devtools Heap Snapshot viewer (.heapsnapshot extension) to a file (`$pid_$timestamp.heapsnapshot`) in the current directory by default (or tempdir if the current directory is unwritable), or in `dir` if given, or the given full file path, or IO stream.

If `all_one` is true, then report the size of every object as one so they can be easily counted. Otherwise, report the actual size.

source

The methods in `Profile` are not exported and need to be called e.g. as `Profile.take_heap_snapshot()`.

```
julia> using Profile

julia> Profile.take_heap_snapshot("snapshot.heapsnapshot")
```

Traces and records julia objects on the heap. This only records objects known to the Julia garbage collector. Memory allocated by external libraries not managed by the garbage collector will not show up in the snapshot.

The resulting heap snapshot file can be uploaded to chrome devtools to be viewed. For more information, see the chrome devtools docs.

Chapter 87

The Julia REPL

Julia 附带了一个全功能的交互式命令行 REPL（read-eval-print loop），其内置于 `julia` 可执行文件中。它除了允许快速简便地执行 Julia 语句外，还具有可搜索的历史记录，tab 补全，许多有用的按键绑定以及专用的 help 和 shell 模式。只需不附带任何参数地调用 `julia` 或双击可执行文件即可启动 REPL：

```
$ julia

               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.10.9 (2025-03-10)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |


julia>
```

要退出交互式会话，在空白行上键入 `^D`——control 键和 d 键，或者先键入 `quit()`，然后键入 return 或 enter 键。REPL 用横幅和 `julia>` 提示符欢迎你。

87.1 不同的提示符模式

Julian 模式

The REPL has five main modes of operation. The first and most common is the Julian prompt. It is the default mode of operation; each new line initially starts with `julia>`. It is here that you can enter Julia expressions. Hitting return or enter after a complete expression has been entered will evaluate the entry and show the result of the last expression.

```
julia> string(1 + 2)
"3"
```

There are a number of useful features unique to interactive work. In addition to showing the result, the REPL also binds the result to the variable `ans`. A trailing semicolon on the line can be used as a flag to suppress showing the result.

```
julia> string(3 * 4);

julia> ans
"12"
```

In Julia mode, the REPL supports something called *prompt pasting*. This activates when pasting text that starts with `julia>` into the REPL. In that case, only expressions starting with `julia>` (as well as the other REPL mode prompts: `shell>`, `help?>`, `pkg>`) are parsed, but others are removed. This makes it possible to paste a chunk of text that has been copied from a REPL session without having to scrub away prompts and outputs. This feature is enabled by default but can be disabled or enabled at will with `REPL.enable_promptpaste(::Bool)`. If it is enabled, you can try it out by pasting the code block above this paragraph straight into the REPL. This feature does not work on the standard Windows command prompt due to its limitation at detecting when a paste occurs.

Objects are printed at the REPL using the `show` function with a specific `IOContext`. In particular, the `:limit` attribute is set to `true`. Other attributes can receive in certain `show` methods a default value if it's not already set, like `:compact`. It's possible, as an experimental feature, to specify the attributes used by the REPL via the `Base.active_repl.options.iocontext` dictionary (associating values to attributes). For example:

```
julia> rand(2, 2)
2×2 Array{Float64,2}:
 0.8833    0.329197
 0.719708  0.59114

julia> show(IOContext(stdout, :compact => false), "text/plain", rand(2, 2))
 0.43540323669187075  0.15759787870609387
 0.2540832269192739   0.4597637838786053
julia> Base.active_repl.options.iocontext[:compact] = false;

julia> rand(2, 2)
2×2 Array{Float64,2}:
 0.2083967319174056  0.13330606013126012
 0.6244375177790158  0.9777957560761545
```

In order to define automatically the values of this dictionary at startup time, one can use the `atreplinit` function in the `~/.julia/config/startup.jl` file, for example:

```
atreplinit() do repl
    repl.options.iocontext[:compact] = false
end
```

帮助模式

When the cursor is at the beginning of the line, the prompt can be changed to a help mode by typing `?`. Julia will attempt to print help or documentation for anything entered in help mode:

```
julia> ? # upon typing ?, the prompt changes (in place) to: help?>

help?> string
search: string String Cstring Cwstring RevString randstring bytestring SubString
```

```
  string(xs...)

  Create a string from any values using the print function.
```

Macros, types and variables can also be queried:

```
help?> @time
  @time

  A macro to execute an expression, printing the time it took to execute, the number of
  ↪  allocations,
  and the total number of bytes its execution caused to be allocated, before returning the value of
  ↪  the
  expression.

  See also @timev, @timed, @elapsed, and @allocated.

help?> Int32
search: Int32 UInt32

  Int32 <: Signed

  32-bit signed integer type.
```

A string or regex literal searches all docstrings using `apropos`:

```
help?> "aprop"
REPL.stripmd
Base.Docs.apropos

help?> r"ap..p"
Base.:∘
Base.shell_escape_posixly
Distributed.CachingPool
REPL.stripmd
Base.Docs.apropos
```

Another feature of help mode is the ability to access extended docstrings. You can do this by typing something like `??Print` rather than `?Print` which will display the `# Extended help` section from the source codes documentation.

Help mode can be exited by pressing backspace at the beginning of the line.

Shell mode

Just as help mode is useful for quick access to documentation, another common task is to use the system shell to execute system commands. Just as `?` entered help mode when at the beginning of the line, a semicolon (`;`) will enter the shell mode. And it can be exited by pressing backspace at the beginning of the line.

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>
```

```
shell> echo hello
hello
```

Note

For Windows users, Julia's shell mode does not expose windows shell commands. Hence, this will fail:

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>

shell> dir
ERROR: IOError: could not spawn `dir`: no such file or directory (ENOENT)
Stacktrace!
.......
```

However, you can get access to `PowerShell` like this:

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>

shell> powershell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
PS C:\Users\elm>
```

... and to `cmd.exe` like that (see the `dir` command):

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>

shell> cmd
Microsoft Windows [version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.
C:\Users\elm>dir
 Volume in drive C has no label
 Volume Serial Number is 1643-0CD7
  Directory of C:\Users\elm

29/01/2020  22:15    <DIR>          .
29/01/2020  22:15    <DIR>          ..
02/02/2020  08:06    <DIR>          .atom
```

Pkg mode

The Package manager mode accepts specialized commands for loading and updating packages. It is entered by pressing the] key at the Julian REPL prompt and exited by pressing CTRL-C or pressing the backspace key at the beginning of the line. The prompt for this mode is pkg>. It supports its own help-mode, which is entered by pressing ? at the beginning of the line of the pkg> prompt. The Package manager mode is documented in the Pkg manual, available at https://julialang.github.io/Pkg.jl/v1/.

Search modes

In all of the above modes, the executed lines get saved to a history file, which can be searched. To initiate an incremental search through the previous history, type ^R –the control key together with the r key. The prompt will change to (reverse-i-search)`':, and as you type the search query will appear in the quotes. The most recent result that matches the query will dynamically update to the right of the colon as more is typed. To find an older result using the same query, simply type ^R again.

Just as ^R is a reverse search, ^S is a forward search, with the prompt (i-search)`':. The two may be used in conjunction with each other to move through the previous or next matching results, respectively.

All executed commands in the Julia REPL are logged into ~/.julia/logs/repl_history.jl along with a timestamp of when it was executed and the current REPL mode you were in. Search mode queries this log file in order to find the commands which you previously ran. This can be disabled at startup by passing the --history-file=no flag to Julia.

87.2 Key bindings

The Julia REPL makes great use of key bindings. Several control-key bindings were already introduced above (^D to exit, ^R and ^S for searching), but there are many more. In addition to the control-key, there are also meta-key bindings. These vary more by platform, but most terminals default to using alt- or option- held down with a key to send the meta-key (or can be configured to do so), or pressing Esc and then the key.

Customizing keybindings

Julia's REPL keybindings may be fully customized to a user's preferences by passing a dictionary to REPL.setup_interface. The keys of this dictionary may be characters or strings. The key '*' refers to the default action. Control plus character x bindings are indicated with "^x". Meta plus x can be written "\\M-x" or "\ex", and Control plus x can be written "\\C-x" or "^x". The values of the custom keymap must be nothing (indicating that the input should be ignored) or functions that accept the signature (PromptState, AbstractREPL, Char). The REPL.setup_interface function must be called before the REPL is initialized, by registering the operation with atreplinit . For example, to bind the up and down arrow keys to move through history without prefix search, one could put the following code in ~/.julia/config/startup.jl:

```
import REPL
import REPL.LineEdit

const mykeys = Dict{Any,Any}(
    # Up Arrow
    "\e[A" => (s,o...)->(LineEdit.edit_move_up(s) || LineEdit.history_prev(s,
    ↪  LineEdit.mode(s).hist)),
    # Down Arrow
    "\e[B" => (s,o...)->(LineEdit.edit_move_down(s) || LineEdit.history_next(s,
    ↪  LineEdit.mode(s).hist))
)

function customize_keys(repl)
    repl.interface = REPL.setup_interface(repl; extra_repl_keymap = mykeys)
end

atreplinit(customize_keys)
```

Users should refer to LineEdit.jl to discover the available actions on key input.

87.3 Tab completion

In both the Julian and help modes of the REPL, one can enter the first few characters of a function or type and then press the tab key to get a list all matches:

```
julia> x[TAB]
julia> xor
```

In some cases it only completes part of the name, up to the next ambiguity:

```
julia> mapf[TAB]
julia> mapfold
```

If you hit tab again, then you get the list of things that might complete this:

```
julia> mapfold[TAB]
mapfoldl mapfoldr
```

Like other components of the REPL, the search is case-sensitive:

```
julia> stri[TAB]
stride     strides     string      strip

julia> Stri[TAB]
StridedArray    StridedMatrix    StridedVecOrMat  StridedVector    String
```

The tab key can also be used to substitute LaTeX math symbols with their Unicode equivalents, and get a list of LaTeX matches as well:

```
julia> \pi[TAB]
julia> π
π = 3.1415926535897...

julia> e\_1[TAB] = [1,0]
julia> e₁ = [1,0]
2-element Array{Int64,1}:
 1
 0

julia> e\^1[TAB] = [1 0]
julia> e¹ = [1 0]
1×2 Array{Int64,2}:
 1  0

julia> \sqrt[TAB]2     # √ is equivalent to the sqrt function
julia> √2
1.4142135623730951

julia> \hbar[TAB](h) = h / 2\pi[TAB]
julia> ħ(h) = h / 2π
ħ (generic function with 1 method)
```

```
julia> \h[TAB]
\hat              \hermitconjmatrix  \hkswarow          \hrectangle
\hatapprox        \hexagon           \hookleftarrow     \hrectangleblack
\hbar             \hexagonblack      \hookrightarrow    \hslash
\heartsuit        \hksearow          \house             \hspace

julia> α="\alpha[TAB]"   # LaTeX completion also works in strings
julia> α="α"
```

A full list of tab-completions can be found in the Unicode Input section of the manual.

Completion of paths works for strings and julia's shell mode:

```
julia> path="/[TAB]"
.dockerenv  .juliabox/   boot/         etc/          lib/          media/        opt/          root/
↪  sbin/         sys/          usr/
.dockerinit bin/         dev/          home/         lib64/        mnt/          proc/         run/
↪  srv/          tmp/          var/
shell> /[TAB]
.dockerenv  .juliabox/   boot/         etc/          lib/          media/        opt/          root/
↪  sbin/         sys/          usr/
.dockerinit bin/         dev/          home/         lib64/        mnt/          proc/         run/
↪  srv/          tmp/          var/
```

Dictionary keys can also be tab completed:

```
julia> foo = Dict("qwer1"=>1, "qwer2"=>2, "asdf"=>3)
Dict{String,Int64} with 3 entries:
  "qwer2" => 2
  "asdf"  => 3
  "qwer1" => 1

julia> foo["q[TAB]

"qwer1" "qwer2"
julia> foo["qwer
```

Tab completion can also help completing fields:

```
julia> x = 3 + 4im;

julia> x.[TAB][TAB]
im re

julia> import UUIDs

julia> UUIDs.uuid[TAB][TAB]
uuid1        uuid4         uuid5        uuid_version
```

Fields for output from functions can also be completed:

```
julia> split("","")[1].[TAB]
lastindex  offset  string
```

The completion of fields for output from functions uses type inference, and it can only suggest fields if the function is type stable.

Tab completion can help with investigation of the available methods matching the input arguments:

```
julia> max([TAB] # All methods are displayed, not shown here due to size of the list

julia> max([1, 2], [TAB] # All methods where `Vector{Int}` matches as first argument
max(x, y) in Base at operators.jl:215
max(a, b, c, xs...) in Base at operators.jl:281

julia> max([1, 2], max(1, 2), [TAB] # All methods matching the arguments.
max(x, y) in Base at operators.jl:215
max(a, b, c, xs...) in Base at operators.jl:281
```

Keywords are also displayed in the suggested methods after `;`, see below line where `limit` and `keepempty` are keyword arguments:

```
julia> split("1 1 1", [TAB]
split(str::AbstractString; limit, keepempty) in Base at strings/util.jl:302
split(str::T, splitter; limit, keepempty) where T<:AbstractString in Base at strings/util.jl:277
```

The completion of the methods uses type inference and can therefore see if the arguments match even if the arguments are output from functions. The function needs to be type stable for the completion to be able to remove non-matching methods.

If you wonder which methods can be used with particular argument types, use ? as the function name. This shows an example of looking for functions in InteractiveUtils that accept a single string:

```
julia> InteractiveUtils.?("somefile")[TAB]
edit(path::AbstractString) in InteractiveUtils at InteractiveUtils/src/editless.jl:197
less(file::AbstractString) in InteractiveUtils at InteractiveUtils/src/editless.jl:266
```

This listed methods in the `InteractiveUtils` module that can be called on a string. By default, this excludes methods where all arguments are typed as Any, but you can see those too by holding down SHIFT-TAB instead of TAB:

```
julia> InteractiveUtils.?("somefile")[SHIFT-TAB]
apropos(string) in REPL at REPL/src/docview.jl:796
clipboard(x) in InteractiveUtils at InteractiveUtils/src/clipboard.jl:64
code_llvm(f) in InteractiveUtils at InteractiveUtils/src/codeview.jl:221
code_native(f) in InteractiveUtils at InteractiveUtils/src/codeview.jl:243
edit(path::AbstractString) in InteractiveUtils at InteractiveUtils/src/editless.jl:197
edit(f) in InteractiveUtils at InteractiveUtils/src/editless.jl:225
eval(x) in InteractiveUtils at InteractiveUtils/src/InteractiveUtils.jl:3
include(x) in InteractiveUtils at InteractiveUtils/src/InteractiveUtils.jl:3
less(file::AbstractString) in InteractiveUtils at InteractiveUtils/src/editless.jl:266
less(f) in InteractiveUtils at InteractiveUtils/src/editless.jl:274
```

```
report_bug(kind) in InteractiveUtils at InteractiveUtils/src/InteractiveUtils.jl:391
separate_kwargs(args...; kwargs...) in InteractiveUtils at InteractiveUtils/src/macros.jl:7
```

You can also use `?("somefile")[TAB]` and look across all modules, but the method lists can be long.

By omitting the closing parenthesis, you can include functions that might require additional arguments:

```
julia> using Mmap

help?> Mmap.?("file",[TAB]
Mmap.Anonymous(name::String, readonly::Bool, create::Bool) in Mmap at Mmap/src/Mmap.jl:16
mmap(file::AbstractString) in Mmap at Mmap/src/Mmap.jl:245
mmap(file::AbstractString, ::Type{T}) where T<:Array in Mmap at Mmap/src/Mmap.jl:245
mmap(file::AbstractString, ::Type{T}, dims::Tuple{Vararg{Integer, N}}) where {T<:Array, N} in Mmap
↪  at Mmap/src/Mmap.jl:245
mmap(file::AbstractString, ::Type{T}, dims::Tuple{Vararg{Integer, N}}, offset::Integer; grow,
↪  shared) where {T<:Array, N} in Mmap at Mmap/src/Mmap.jl:245
mmap(file::AbstractString, ::Type{T}, len::Integer) where T<:Array in Mmap at Mmap/src/Mmap.jl:251
mmap(file::AbstractString, ::Type{T}, len::Integer, offset::Integer; grow, shared) where T<:Array
↪  in Mmap at Mmap/src/Mmap.jl:251
mmap(file::AbstractString, ::Type{T}, dims::Tuple{Vararg{Integer, N}}) where {T<:BitArray, N} in
↪  Mmap at Mmap/src/Mmap.jl:316
mmap(file::AbstractString, ::Type{T}, dims::Tuple{Vararg{Integer, N}}, offset::Integer; grow,
↪  shared) where {T<:BitArray, N} in Mmap at Mmap/src/Mmap.jl:316
mmap(file::AbstractString, ::Type{T}, len::Integer) where T<:BitArray in Mmap at
↪  Mmap/src/Mmap.jl:322
mmap(file::AbstractString, ::Type{T}, len::Integer, offset::Integer; grow, shared) where
↪  T<:BitArray in Mmap at Mmap/src/Mmap.jl:322
```

87.4 Customizing Colors

The colors used by Julia and the REPL can be customized, as well. To change the color of the Julia prompt you can add something like the following to your `~/.julia/config/startup.jl` file, which is to be placed inside your home directory:

```
function customize_colors(repl)
    repl.prompt_color = Base.text_colors[:cyan]
end

atreplinit(customize_colors)
```

The available color keys can be seen by typing `Base.text_colors` in the help mode of the REPL. In addition, the integers 0 to 255 can be used as color keys for terminals with 256 color support.

You can also change the colors for the help and shell prompts and input and answer text by setting the appropriate field of `repl` in the `customize_colors` function above (respectively, `help_color`, `shell_color`, `input_color`, and `answer_color`). For the latter two, be sure that the `envcolors` field is also set to false.

It is also possible to apply boldface formatting by using `Base.text_colors[:bold]` as a color. For instance, to print answers in boldface font, one can use the following as a `~/.julia/config/startup.jl`:

```
function customize_colors(repl)
    repl.envcolors = false
    repl.answer_color = Base.text_colors[:bold]
end

atreplinit(customize_colors)
```

You can also customize the color used to render warning and informational messages by setting the appropriate environment variables. For instance, to render error, warning, and informational messages respectively in magenta, yellow, and cyan you can add the following to your `~/.julia/config/startup.jl` file:

```
ENV["JULIA_ERROR_COLOR"] = :magenta
ENV["JULIA_WARN_COLOR"] = :yellow
ENV["JULIA_INFO_COLOR"] = :cyan
```

87.5 Changing the contextual module which is active at the REPL

When entering expressions at the REPL, they are by default evaluated in the `Main` module;

```
julia> @__MODULE__
Main
```

It is possible to change this contextual module via the function `REPL.activate(m)` where `m` is a `Module` or by typing the module in the REPL and pressing the keybinding Alt-m (the cursor must be on the module name). The active module is shown in the prompt:

```
julia> using REPL

julia> REPL.activate(Base)

(Base) julia> @__MODULE__
Base

(Base) julia> using REPL # Need to load REPL into Base module to use it

(Base) julia> REPL.activate(Main)

julia>

julia> Core<Alt-m> # using the keybinding to change module

(Core) julia>

(Core) julia> Main<Alt-m> # going back to Main via keybinding

julia>
```

Functions that take an optional module argument often defaults to the REPL context module. As an example, calling `varinfo()` will show the variables of the current active module:

```
julia> module CustomMod
           export var, f
           var = 1
           f(x) = x^2
       end;

julia> REPL.activate(CustomMod)

(Main.CustomMod) julia> varinfo()
  name         size summary
  ─────────────────────────────────────────────────────
  CustomMod         Module
  f         0 bytes f (generic function with 1 method)
  var       8 bytes Int64
```

87.6 Numbered prompt

It is possible to get an interface which is similar to the IPython REPL and the Mathematica notebook with numbered input prompts and output prefixes. This is done by calling `REPL.numbered_prompt!()`. If you want to have this enabled on startup, add

```
atreplinit() do repl
    @eval import REPL
    if !isdefined(repl, :interface)
        repl.interface = REPL.setup_interface(repl)
    end
    REPL.numbered_prompt!(repl)
end
```

to your `startup.jl` file. In numbered prompt the variable `Out[n]` (where `n` is an integer) can be used to refer to earlier results:

```
In [1]: 5 + 3
Out[1]: 8

In [2]: Out[1] + 5
Out[2]: 13

In [3]: Out
Out[3]: Dict{Int64, Any} with 2 entries:
  2 => 13
  1 => 8
```

Note

Since all outputs from previous REPL evaluations are saved in the `Out` variable, one should be careful if they are returning many large in-memory objects like arrays, since they will be protected from garbage collection so long as a reference to them remains in `Out`. If you need to remove references to objects in `Out`, you can clear the entire history it stores with `empty!(Out)`, or clear an individual entry with `Out[n] = nothing`.

87.7 TerminalMenus

TerminalMenus is a submodule of the Julia REPL and enables small, low-profile interactive menus in the terminal.

Examples

```
import REPL
using REPL.TerminalMenus

options = ["apple", "orange", "grape", "strawberry",
            "blueberry", "peach", "lemon", "lime"]
```

RadioMenu

The RadioMenu allows the user to select one option from the list. The `request` function displays the interactive menu and returns the index of the selected choice. If a user presses 'q' or `ctrl-c`, `request` will return a `-1`.

```
# `pagesize` is the number of items to be displayed at a time.
#  The UI will scroll if the number of options is greater
#   than the `pagesize`
menu = RadioMenu(options, pagesize=4)

# `request` displays the menu and returns the index after the
#   user has selected a choice
choice = request("Choose your favorite fruit:", menu)

if choice != -1
    println("Your favorite fruit is ", options[choice], "!")
else
    println("Menu canceled.")
end
```

Output:

```
Choose your favorite fruit:
^  grape
   strawberry
 > blueberry
v  peach
Your favorite fruit is blueberry!
```

MultiSelectMenu

The MultiSelectMenu allows users to select many choices from a list.

```
# here we use the default `pagesize` 10
menu = MultiSelectMenu(options)

# `request` returns a `Set` of selected indices
# if the menu us canceled (ctrl-c or q), return an empty set
choices = request("Select the fruits you like:", menu)
```

```
if length(choices) > 0
    println("You like the following fruits:")
    for i in choices
        println("  - ", options[i])
    end
else
    println("Menu canceled.")
end
```

Output:

```
Select the fruits you like:
[press: Enter=toggle, a=all, n=none, d=done, q=abort]
   [ ] apple
 > [X] orange
   [X] grape
   [ ] strawberry
   [ ] blueberry
   [X] peach
   [ ] lemon
   [ ] lime
You like the following fruits:
  - orange
  - grape
  - peach
```

Customization / Configuration

ConfiguredMenu subtypes

Starting with Julia 1.6, the recommended way to configure menus is via the constructor. For instance, the default multiple-selection menu

```
julia> menu = MultiSelectMenu(options, pagesize=5);

julia> request(menu) # ASCII is used by default
[press: Enter=toggle, a=all, n=none, d=done, q=abort]
   [ ] apple
   [X] orange
   [ ] grape
 > [X] strawberry
v  [ ] blueberry
```

can instead be rendered with Unicode selection and navigation characters with

```
julia> menu = MultiSelectMenu(options, pagesize=5, charset=:unicode);

julia> request(menu)
[press: Enter=toggle, a=all, n=none, d=done, q=abort]
   ⬚ apple
   ✓ orange
```

```
   ⬚ grape
 → ✓ strawberry
↓  ⬚ blueberry
```

More fine-grained configuration is also possible:

```
julia> menu = MultiSelectMenu(options, pagesize=5, charset=:unicode, checked="YEP!",
↪  unchecked="NOPE", cursor='□');

julia> request(menu)
julia> request(menu)
[press: Enter=toggle, a=all, n=none, d=done, q=abort]
   NOPE apple
   YEP! orange
   NOPE grape
 □ YEP! strawberry
↓  NOPE blueberry
```

Aside from the overall `charset` option, for RadioMenu the configurable options are:

- `cursor::Char='>'|'→'`: character to use for cursor
- `up_arrow::Char='^'|'↑'`: character to use for up arrow
- `down_arrow::Char='v'|'↓'`: character to use for down arrow
- `updown_arrow::Char='I'|'↕'`: character to use for up/down arrow in one-line page
- `scroll_wrap::Bool=false`: optionally wrap-around at the beginning/end of a menu
- `ctrl_c_interrupt::Bool=true`: If `false`, return empty on ^C, if `true` throw InterruptException() on ^C

`MultiSelectMenu` adds:

- `checked::String="[X]"|"✓"`: string to use for checked
- `unchecked::String="[ ]"|"⬚")`: string to use for unchecked

You can create new menu types of your own. Types that are derived from `TerminalMenus.ConfiguredMenu` configure the menu options at construction time.

Legacy interface

Prior to Julia 1.6, and still supported throughout Julia 1.x, one can also configure menus by calling `TerminalMenus.config()`.

87.8 References

REPL

`Base.atreplinit` – Function.

```
atreplinit(f)
```

Register a one-argument function to be called before the REPL interface is initialized in interactive sessions; this is useful to customize the interface. The argument of `f` is the REPL object. This function should be called from within the `.julia/config/startup.jl` initialization file.

source

TerminalMenus

Menus

`REPL.TerminalMenus.RadioMenu` – Type.

```
RadioMenu
```

A menu that allows a user to select a single option from a list.

Sample Output

```
julia> request(RadioMenu(options, pagesize=4))
Choose your favorite fruit:
^  grape
   strawberry
 > blueberry
v  peach
Your favorite fruit is blueberry!
```

`REPL.TerminalMenus.MultiSelectMenu` – Type.

```
MultiSelectMenu
```

A menu that allows a user to select a multiple options from a list.

Sample Output

```
julia> request(MultiSelectMenu(options))
Select the fruits you like:
[press: Enter=toggle, a=all, n=none, d=done, q=abort]
   [ ] apple
 > [X] orange
   [X] grape
   [ ] strawberry
   [ ] blueberry
   [X] peach
   [ ] lemon
   [ ] lime
You like the following fruits:
  - orange
  - grape
  - peach
```

Configuration

REPL.TerminalMenus.Config – Type.

```
Config(; scroll_wrap=false, ctrl_c_interrupt=true, charset=:ascii, cursor::Char, up_arrow::Char,
↪  down_arrow::Char)
```

Configure behavior for selection menus via keyword arguments:

- scroll_wrap, if true, causes the menu to wrap around when scrolling above the first or below the last entry
- ctrl_c_interrupt, if true, throws an InterruptException if the user hits Ctrl-C during menu selection. If false, TerminalMenus.request will return the default result from TerminalMenus.selected.
- charset affects the default values for cursor, up_arrow, and down_arrow, and can be :ascii or :unicode
- cursor is the character printed to indicate the option that will be chosen by hitting "Enter." Defaults are '>' or '→', depending on charset.
- up_arrow is the character printed when the display does not include the first entry. Defaults are '^' or '↑', depending on charset.
- down_arrow is the character printed when the display does not include the last entry. Defaults are 'v' or '↓', depending on charset.

Subtypes of ConfiguredMenu will print cursor, up_arrow, and down_arrow automatically as needed, your writeline method should not print them.

> **Julia 1.6**
>
> Config is available as of Julia 1.6. On older releases use the global CONFIG.

REPL.TerminalMenus.MultiSelectConfig – Type.

```
MultiSelectConfig(; charset=:ascii, checked::String, unchecked::String, kwargs...)
```

Configure behavior for a multiple-selection menu via keyword arguments:

- checked is the string to print when an option has been selected. Defaults are "[X]" or "✓", depending on charset.
- unchecked is the string to print when an option has not been selected. Defaults are "[ ]" or "⬚", depending on charset.

All other keyword arguments are as described for TerminalMenus.Config. checked and unchecked are not printed automatically, and should be printed by your writeline method.

> **Julia 1.6**
>
> MultiSelectConfig is available as of Julia 1.6. On older releases use the global CONFIG.

`REPL.TerminalMenus.config` – Function.

```
config( <see arguments> )
```

Keyword-only function to configure global menu parameters

Arguments

- `charset::Symbol=:na`: ui characters to use (`:ascii` or `:unicode`); overridden by other arguments
- `cursor::Char='>'|'→'`: character to use for cursor
- `up_arrow::Char='^'|'↑'`: character to use for up arrow
- `down_arrow::Char='v'|'↓'`: character to use for down arrow
- `checked::String="[X]"|"✓"`: string to use for checked
- `unchecked::String="[ ]"|"⬚")`: string to use for unchecked
- `scroll::Symbol=:nowrap`: If `:wrap` wrap cursor around top and bottom, if `:nowrap` do not wrap cursor
- `supress_output::Bool=false`: Ignored legacy argument, pass `suppress_output` as a keyword argument to `request` instead.
- `ctrl_c_interrupt::Bool=true`: If `false`, return empty on ^C, if `true` throw InterruptException() on ^C

> **Julia 1.6**
>
> As of Julia 1.6, `config` is deprecated. Use `Config` or `MultiSelectConfig` instead.

User interaction

`REPL.TerminalMenus.request` – Function.

```
request(m::AbstractMenu; cursor=1)
```

Display the menu and enter interactive mode. `cursor` indicates the item number used for the initial cursor position. `cursor` can be either an `Int` or a `RefValue{Int}`. The latter is useful for observation and control of the cursor position from the outside.

Returns `selected(m)`.

> **Julia 1.6**
>
> The `cursor` argument requires Julia 1.6 or later.

```
request([term,] msg::AbstractString, m::AbstractMenu)
```

Shorthand for `println(msg); request(m)`.

AbstractMenu extension interface

Any subtype of `AbstractMenu` must be mutable, and must contain the fields `pagesize::Int` and `pageoffset::Int`. Any subtype must also implement the following functions:

`REPL.TerminalMenus.pick` – Function.

```
pick(m::AbstractMenu, cursor::Int)
```

Defines what happens when a user presses the Enter key while the menu is open. If `true` is returned, `request()` will exit. `cursor` indexes the position of the selection.

`REPL.TerminalMenus.cancel` – Function.

```
cancel(m::AbstractMenu)
```

Define what happens when a user cancels ('q' or ctrl-c) a menu. `request()` will always exit after calling this function.

`REPL.TerminalMenus.writeline` – Function.

```
writeline(buf::IO, m::AbstractMenu, idx::Int, iscursor::Bool)
```

Write the option at index `idx` to `buf`. `iscursor`, if `true`, indicates that this item is at the current cursor position (the one that will be selected by hitting "Enter").

If `m` is a `ConfiguredMenu`, `TerminalMenus` will print the cursor indicator. Otherwise the callee is expected to handle such printing.

> **Julia 1.6**
>
> `writeline` requires Julia 1.6 or higher.
>
> On older versions of Julia, this was `writeLine(buf::IO, m::AbstractMenu, idx, iscursor::Bool)` and `m` is assumed to be unconfigured. The selection and cursor indicators can be obtained from `TerminalMenus.CONFIG`.
>
> This older function is supported on all Julia 1.x versions but will be dropped in Julia 2.0.

It must also implement either `options` or `numoptions`:

`REPL.TerminalMenus.options` – Function.

```
options(m::AbstractMenu)
```

Return a list of strings to be displayed as options in the current page.

Alternatively, implement `numoptions`, in which case `options` is not needed.

`REPL.TerminalMenus.numoptions` – Function.

```
numoptions(m::AbstractMenu) -> Int
```

Return the number of options in menu `m`. Defaults to `length(options(m))`.

Julia 1.6

This function requires Julia 1.6 or later.

If the subtype does not have a field named `selected`, it must also implement

`REPL.TerminalMenus.selected` – Function.

```
selected(m::AbstractMenu)
```

Return information about the user-selected option. By default it returns `m.selected`.

The following are optional but can allow additional customization:

`REPL.TerminalMenus.header` – Function.

```
header(m::AbstractMenu) -> String
```

Return a header string to be printed above the menu. Defaults to "".

`REPL.TerminalMenus.keypress` – Function.

```
keypress(m::AbstractMenu, i::UInt32) -> Bool
```

Handle any non-standard keypress event. If `true` is returned, `TerminalMenus.request` will exit. Defaults to `false`.

Keybinding	Description
Program control	
`^D`	Exit (when buffer is empty)
`^C`	Interrupt or cancel
`^L`	Clear console screen
Return/Enter, `^J`	New line, executing if it is complete
meta-Return/Enter	Insert new line without executing it
`?` or `;`	Enter help or shell mode (when at start of a line)
`^R`, `^S`	Incremental history search, described above
Cursor movement	
Right arrow, `^F`	Move right one character
Left arrow, `^B`	Move left one character
ctrl-Right, `meta-F`	Move right one word
ctrl-Left, `meta-B`	Move left one word
Home, `^A`	Move to beginning of line
End, `^E`	Move to end of line
Up arrow, `^P`	Move up one line (or change to the previous history entry that matches the text before the cursor)
Down arrow, `^N`	Move down one line (or change to the next history entry that matches the text before the cursor)
Shift-Arrow Key	Move cursor according to the direction of the Arrow key, while activating the region ("shift selection")
Page-up, `meta-P`	Change to the previous history entry
Page-down, `meta-N`	Change to the next history entry
`meta-<`	Change to the first history entry (of the current session if it is before the current position in history)
`meta->`	Change to the last history entry
`^-Space`	Set the "mark" in the editing region (and de-activate the region if it's active)
`^-Space ^-Space`	Set the "mark" in the editing region and make the region "active", i.e. highlighted
`^G`	De-activate the region (i.e. make it not highlighted)
`^X^X`	Exchange the current position with the mark
Editing	
Backspace, `^H`	Delete the previous character, or the whole region when it's active
Delete, `^D`	Forward delete one character (when buffer has text)
meta-Backspace	Delete the previous word
`meta-d`	Forward delete the next word
`^W`	Delete previous text up to the nearest whitespace
`meta-w`	Copy the current region in the kill ring
`meta-W`	"Kill" the current region, placing the text in the kill ring
`^U`	"Kill" to beginning of line, placing the text in the kill ring
`^K`	"Kill" to end of line, placing the text in the kill ring
`^Y`	"Yank" insert the text from the kill ring
`meta-y`	Replace a previously yanked text with an older entry from the kill ring
`^T`	Transpose the characters about the cursor
`meta-Up arrow`	Transpose current line with line above
`meta-Down arrow`	Transpose current line with line below
`meta-u`	Change the next word to uppercase
`meta-c`	Change the next word to titlecase
`meta-l`	Change the next word to lowercase
`^/, ^`	Undo previous editing action

Chapter 88

随机数

Random number generation in Julia uses the Xoshiro256++ algorithm by default, with per-Task state. Other RNG types can be plugged in by inheriting the AbstractRNG type; they can then be used to obtain multiple streams of random numbers.

The PRNGs (pseudorandom number generators) exported by the Random package are:

- TaskLocalRNG: a token that represents use of the currently active Task-local stream, deterministically seeded from the parent task, or by RandomDevice (with system randomness) at program start
- Xoshiro: generates a high-quality stream of random numbers with a small state vector and high performance using the Xoshiro256++ algorithm
- RandomDevice: for OS-provided entropy. This may be used for cryptographically secure random numbers (CS(P)RNG).
- MersenneTwister: an alternate high-quality PRNG which was the default in older versions of Julia, and is also quite fast, but requires much more space to store the state vector and generate a random sequence.

Most functions related to random generation accept an optional AbstractRNG object as first argument. Some also accept dimension specifications dims... (which can also be given as a tuple) to generate arrays of random values. In a multi-threaded program, you should generally use different RNG objects from different threads or tasks in order to be thread-safe. However, the default RNG is thread-safe as of Julia 1.3 (using a per-thread RNG up to version 1.6, and per-task thereafter).

The provided RNGs can generate uniform random numbers of the following types: Float16, Float32, Float64, BigFloat, Bool, Int8, UInt8, Int16, UInt16, Int32, UInt32, Int64, UInt64, Int128, UInt128, BigInt (or complex numbers of those types). Random floating point numbers are generated uniformly in $[0, 1)$. As BigInt represents unbounded integers, the interval must be specified (e.g. rand(big.(1:6))).

另外，正态和指数分布是针对某些 AbstractFloat 和 Complex 类型，详细内容见 randn 和 randexp。

To generate random numbers from other distributions, see the Distributions.jl package.

Warning

Because the precise way in which random numbers are generated is considered an implementation detail, bug fixes and speed improvements may change the stream of numbers that are generated after a version change. Relying on a specific seed or generated stream of numbers during unit testing is thus discouraged - consider testing properties of the methods in question instead.

88.1 Random numbers module

Random.Random – Module.

```
Random
```

Support for generating random numbers. Provides rand, randn, AbstractRNG, MersenneTwister, and RandomDevice.

88.2 Random generation functions

Base.rand – Function.

```
rand([rng=default_rng()], [S], [dims...])
```

Pick a random element or array of random elements from the set of values specified by S; S can be

- an indexable collection (for example 1:9 or ('x', "y", :z)),
- an AbstractDict or AbstractSet object,
- a string (considered as a collection of characters), or
- a type: the set of values to pick from is then equivalent to typemin(S):typemax(S) for integers (this is not applicable to BigInt), to $[0,1)$ for floating point numbers and to $[0,1)+i[0,1)$ for complex floating point numbers;

S defaults to Float64. When only one argument is passed besides the optional rng and is a Tuple, it is interpreted as a collection of values (S) and not as dims.

See also randn for normally distributed numbers, and rand! and randn! for the in-place equivalents.

Julia 1.1

Support for S as a tuple requires at least Julia 1.1.

Examples

```
julia> rand(Int, 2)
2-element Array{Int64,1}:
 1339893410598768192
 1575814717733606317

julia> using Random

julia> rand(MersenneTwister(0), Dict(1=>2, 3=>4))
1=>2

julia> rand((2, 3))
3

julia> rand(Float64, (2, 3))
```

```
2×3 Array{Float64,2}:
 0.999717  0.0143835  0.540787
 0.696556  0.783855   0.938235
```

Note

The complexity of `rand(rng, s::Union{AbstractDict,AbstractSet})` is linear in the length of `s`, unless an optimized method with constant complexity is available, which is the case for `Dict`, `Set` and dense `BitSet`s. For more than a few calls, use `rand(rng, collect(s))` instead, or either `rand(rng, Dict(s))` or `rand(rng, Set(s))` as appropriate.

`Random.rand!` – Function.

```
rand!([rng=default_rng()], A, [S=eltype(A)])
```

Populate the array A with random values. If S is specified (S can be a type or a collection, cf. rand for details), the values are picked randomly from S. This is equivalent to `copyto!(A, rand(rng, S, size(A)))` but without allocating a new array.

Examples

```
julia> rng = MersenneTwister(1234);

julia> rand!(rng, zeros(5))
5-element Vector{Float64}:
 0.5908446386657102
 0.7667970365022592
 0.5662374165061859
 0.4600853424625171
 0.7940257103317943
```

`Random.bitrand` – Function.

```
bitrand([rng=default_rng()], [dims...])
```

Generate a `BitArray` of random boolean values.

Examples

```
julia> rng = MersenneTwister(1234);

julia> bitrand(rng, 10)
10-element BitVector:
 0
 0
 0
 0
 1
```

```
0
0
0
1
1
```

Base.randn – Function.

```
randn([rng=default_rng()], [T=Float64], [dims...])
```

Generate a normally-distributed random number of type T with mean 0 and standard deviation 1. Optionally generate an array of normally-distributed random numbers. The Base module currently provides an implementation for the types `Float16`, `Float32`, and `Float64` (the default), and their `Complex` counterparts. When the type argument is complex, the values are drawn from the circularly symmetric complex normal distribution of variance 1 (corresponding to real and imaginary part having independent normal distribution with mean zero and variance 1/2).

See also `randn!` to act in-place.

Examples

```
julia> using Random

julia> rng = MersenneTwister(1234);

julia> randn(rng, ComplexF64)
0.6133070881429037 - 0.6376291670853887im

julia> randn(rng, ComplexF32, (2, 3))
2×3 Matrix{ComplexF32}:
 -0.349649-0.638457im  0.376756-0.192146im  -0.396334-0.0136413im
  0.611224+1.56403im   0.355204-0.365563im  0.0905552+1.31012im
```

Random.randn! – Function.

```
randn!([rng=default_rng()], A::AbstractArray) -> A
```

Fill the array A with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the `rand` function.

Examples

```
julia> rng = MersenneTwister(1234);

julia> randn!(rng, zeros(5))
5-element Vector{Float64}:
  0.8673472019512456
 -0.9017438158568171
 -0.4944787535042339
 -0.9029142938652416
  0.8644013132535154
```

Random.randexp – Function.

```
randexp([rng=default_rng()], [T=Float64], [dims...])
```

Generate a random number of type T according to the exponential distribution with scale 1. Optionally generate an array of such random numbers. The Base module currently provides an implementation for the types Float16, Float32, and Float64 (the default).

Examples

```
julia> rng = MersenneTwister(1234);

julia> randexp(rng, Float32)
2.4835055f0

julia> randexp(rng, 3, 3)
3×3 Matrix{Float64}:
 1.5167    1.30652   0.344435
 0.604436  2.78029   0.418516
 0.695867  0.693292  0.643644
```

Random.randexp! – Function.

```
randexp!([rng=default_rng()], A::AbstractArray) -> A
```

Fill the array A with random numbers following the exponential distribution (with scale 1).

Examples

```
julia> rng = MersenneTwister(1234);

julia> randexp!(rng, zeros(5))
5-element Vector{Float64}:
 2.4835053723904896
 1.516703605376473
 0.6044364871025417
 0.6958665886385867
 1.3065196315496677
```

Random.randstring – Function.

```
randstring([rng=default_rng()], [chars], [len=8])
```

Create a random string of length len, consisting of characters from chars, which defaults to the set of upper- and lower-case letters and the digits 0-9. The optional rng argument specifies a random number generator, see Random Numbers.

Examples

```
julia> Random.seed!(3); randstring()
"Lxz5hUwn"

julia> randstring(MersenneTwister(3), 'a':'z', 6)
"ocucay"

julia> randstring("ACGT")
"TGCTCCTC"
```

Note

chars can be any collection of characters, of type Char or UInt8 (more efficient), provided rand can randomly pick characters from it.

88.3 Subsequences, permutations and shuffling

Random.randsubseq – Function.

```
randsubseq([rng=default_rng(),] A, p) -> Vector
```

Return a vector consisting of a random subsequence of the given array A, where each element of A is included (in order) with independent probability p. (Complexity is linear in p*length(A), so this function is efficient even if p is small and A is large.) Technically, this process is known as "Bernoulli sampling" of A.

Examples

```
julia> rng = MersenneTwister(1234);

julia> randsubseq(rng, 1:8, 0.3)
2-element Vector{Int64}:
 7
 8
```

Random.randsubseq! – Function.

```
randsubseq!([rng=default_rng(),] S, A, p)
```

Like randsubseq, but the results are stored in S (which is resized as needed).

Examples

```
julia> rng = MersenneTwister(1234);

julia> S = Int64[];

julia> randsubseq!(rng, S, 1:8, 0.3)
2-element Vector{Int64}:
```

```
 7
 8

julia> S
2-element Vector{Int64}:
 7
 8
```

Random.randperm – Function.

```
randperm([rng=default_rng(),] n::Integer)
```

Construct a random permutation of length n. The optional rng argument specifies a random number generator (see Random Numbers). The element type of the result is the same as the type of n.

To randomly permute an arbitrary vector, see shuffle or shuffle!.

Julia 1.1

In Julia 1.1 randperm returns a vector v with eltype(v) == typeof(n) while in Julia 1.0 eltype(v) == Int.

Examples

```
julia> randperm(MersenneTwister(1234), 4)
4-element Vector{Int64}:
 2
 1
 4
 3
```

Random.randperm! – Function.

```
randperm!([rng=default_rng(),] A::Array{<:Integer})
```

Construct in A a random permutation of length length(A). The optional rng argument specifies a random number generator (see Random Numbers). To randomly permute an arbitrary vector, see shuffle or shuffle!.

Examples

```
julia> randperm!(MersenneTwister(1234), Vector{Int}(undef, 4))
4-element Vector{Int64}:
 2
 1
 4
 3
```

Random.randcycle – Function.

```
randcycle([rng=default_rng(),] n::Integer)
```

Construct a random cyclic permutation of length n. The optional rng argument specifies a random number generator, see Random Numbers. The element type of the result is the same as the type of n.

> **Julia 1.1**
>
> In Julia 1.1 randcycle returns a vector v with eltype(v) == typeof(n) while in Julia 1.0 eltype(v) == Int.

Examples

```
julia> randcycle(MersenneTwister(1234), 6)
6-element Vector{Int64}:
 3
 5
 4
 6
 1
 2
```

Random.randcycle! – Function.

```
randcycle!([rng=default_rng(),] A::Array{<:Integer})
```

Construct in A a random cyclic permutation of length length(A). The optional rng argument specifies a random number generator, see Random Numbers.

Examples

```
julia> randcycle!(MersenneTwister(1234), Vector{Int}(undef, 6))
6-element Vector{Int64}:
 3
 5
 4
 6
 1
 2
```

Random.shuffle – Function.

```
shuffle([rng=default_rng(),] v::AbstractArray)
```

Return a randomly permuted copy of v. The optional rng argument specifies a random number generator (see Random Numbers). To permute v in-place, see shuffle!. To obtain randomly permuted indices, see randperm.

Examples

```
julia> rng = MersenneTwister(1234);

julia> shuffle(rng, Vector(1:10))
10-element Vector{Int64}:
  6
  1
 10
  2
  3
  9
  5
  7
  4
  8
```

Random.shuffle! – Function.

```
shuffle!([rng=default_rng(),] v::AbstractArray)
```

In-place version of shuffle: randomly permute v in-place, optionally supplying the random-number generator rng.

Examples

```
julia> rng = MersenneTwister(1234);

julia> shuffle!(rng, Vector(1:16))
16-element Vector{Int64}:
  2
 15
  5
 14
  1
  9
 10
  6
 11
  3
 16
  7
  4
 12
  8
 13
```

88.4 Generators (creation and seeding)

Random.default_rng – Function.

```
default_rng() -> rng
```

Return the default global random number generator (RNG).

> **Note**
>
> What the default RNG is is an implementation detail. Across different versions of Julia, you should not expect the default RNG to be always the same, nor that it will return the same stream of random numbers for a given seed.

> **Julia 1.3**
>
> This function was introduced in Julia 1.3.

Random.seed! – Function.

```
seed!([rng=default_rng()], seed) -> rng
seed!([rng=default_rng()]) -> rng
```

Reseed the random number generator: rng will give a reproducible sequence of numbers if and only if a seed is provided. Some RNGs don't accept a seed, like RandomDevice. After the call to seed!, rng is equivalent to a newly created object initialized with the same seed.

If rng is not specified, it defaults to seeding the state of the shared task-local generator.

Examples

```
julia> Random.seed!(1234);

julia> x1 = rand(2)
2-element Vector{Float64}:
 0.32597672886359486
 0.5490511363155669

julia> Random.seed!(1234);

julia> x2 = rand(2)
2-element Vector{Float64}:
 0.32597672886359486
 0.5490511363155669

julia> x1 == x2
true

julia> rng = Xoshiro(1234); rand(rng, 2) == x1
true

julia> Xoshiro(1) == Random.seed!(rng, 1)
true
```

```
julia> rand(Random.seed!(rng), Bool) # not reproducible
true

julia> rand(Random.seed!(rng), Bool) # not reproducible either
false

julia> rand(Xoshiro(), Bool) # not reproducible either
true
```

`Random.AbstractRNG` – Type.

```
AbstractRNG
```

Supertype for random number generators such as `MersenneTwister` and `RandomDevice`.

`Random.TaskLocalRNG` – Type.

```
TaskLocalRNG
```

The `TaskLocalRNG` has state that is local to its task, not its thread. It is seeded upon task creation, from the state of its parent task, but without advancing the state of the parent's RNG.

As an upside, the `TaskLocalRNG` is pretty fast, and permits reproducible multithreaded simulations (barring race conditions), independent of scheduler decisions. As long as the number of threads is not used to make decisions on task creation, simulation results are also independent of the number of available threads / CPUs. The random stream should not depend on hardware specifics, up to endianness and possibly word size.

Using or seeding the RNG of any other task than the one returned by `current_task()` is undefined behavior: it will work most of the time, and may sometimes fail silently.

> **Julia 1.10**
>
> Task creation no longer advances the parent task's RNG state as of Julia 1.10.

`Random.Xoshiro` – Type.

```
Xoshiro(seed)
Xoshiro()
```

Xoshiro256++ is a fast pseudorandom number generator described by David Blackman and Sebastiano Vigna in "Scrambled Linear Pseudorandom Number Generators", ACM Trans. Math. Softw., 2021. Reference implementation is available at http://prng.di.unimi.it

Apart from the high speed, Xoshiro has a small memory footprint, making it suitable for applications where many different random states need to be held for long time.

Julia's Xoshiro implementation has a bulk-generation mode; this seeds new virtual PRNGs from the parent, and uses SIMD to generate in parallel (i.e. the bulk stream consists of multiple interleaved xoshiro

instances). The virtual PRNGs are discarded once the bulk request has been serviced (and should cause no heap allocations).

Examples

```
julia> using Random

julia> rng = Xoshiro(1234);

julia> x1 = rand(rng, 2)
2-element Vector{Float64}:
 0.32597672886359486
 0.5490511363155669

julia> rng = Xoshiro(1234);

julia> x2 = rand(rng, 2)
2-element Vector{Float64}:
 0.32597672886359486
 0.5490511363155669

julia> x1 == x2
true
```

`Random.MersenneTwister` – Type.

```
MersenneTwister(seed)
MersenneTwister()
```

Create a `MersenneTwister` RNG object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers. The `seed` may be a non-negative integer or a vector of `UInt32` integers. If no seed is provided, a randomly generated one is created (using entropy from the system). See the `seed!` function for reseeding an already existing `MersenneTwister` object.

Examples

```
julia> rng = MersenneTwister(1234);

julia> x1 = rand(rng, 2)
2-element Vector{Float64}:
 0.5908446386657102
 0.7667970365022592

julia> rng = MersenneTwister(1234);

julia> x2 = rand(rng, 2)
2-element Vector{Float64}:
 0.5908446386657102
 0.7667970365022592

julia> x1 == x2
true
```

`Random.RandomDevice` – Type.

```
RandomDevice()
```

Create a `RandomDevice` RNG object. Two such objects will always generate different streams of random numbers. The entropy is obtained from the operating system.

88.5 Hooking into the `Random` API

There are two mostly orthogonal ways to extend `Random` functionalities:

1. generating random values of custom types
2. creating new generators

The API for 1) is quite functional, but is relatively recent so it may still have to evolve in subsequent releases of the `Random` module. For example, it's typically sufficient to implement one `rand` method in order to have all other usual methods work automatically.

The API for 2) is still rudimentary, and may require more work than strictly necessary from the implementor, in order to support usual types of generated values.

Generating random values of custom types

Generating random values for some distributions may involve various trade-offs. *Pre-computed* values, such as an alias table for discrete distributions, or "squeezing" functions for univariate distributions, can speed up sampling considerably. How much information should be pre-computed can depend on the number of values we plan to draw from a distribution. Also, some random number generators can have certain properties that various algorithms may want to exploit.

The `Random` module defines a customizable framework for obtaining random values that can address these issues. Each invocation of `rand` generates a *sampler* which can be customized with the above trade-offs in mind, by adding methods to `Sampler`, which in turn can dispatch on the random number generator, the object that characterizes the distribution, and a suggestion for the number of repetitions. Currently, for the latter, `Val{1}` (for a single sample) and `Val{Inf}` (for an arbitrary number) are used, with `Random.Repetition` an alias for both.

The object returned by `Sampler` is then used to generate the random values. When implementing the random generation interface for a value `X` that can be sampled from, the implementor should define the method

```
rand(rng, sampler)
```

for the particular `sampler` returned by `Sampler(rng, X, repetition)`.

Samplers can be arbitrary values that implement `rand(rng, sampler)`, but for most applications the following predefined samplers may be sufficient:

1. `SamplerType{T}()` can be used for implementing samplers that draw from type `T` (e.g. `rand(Int)`). This is the default returned by `Sampler` for *types*.

2. `SamplerTrivial(self)` is a simple wrapper for `self`, which can be accessed with `[]`. This is the recommended sampler when no pre-computed information is needed (e.g. `rand(1:3)`), and is the default returned by `Sampler` for *values*.

3. `SamplerSimple(self, data)` also contains the additional `data` field, which can be used to store arbitrary pre-computed values, which should be computed in a *custom method* of `Sampler`.

We provide examples for each of these. We assume here that the choice of algorithm is independent of the RNG, so we use `AbstractRNG` in our signatures.

`Random.Sampler` – Type.

```
Sampler(rng, x, repetition = Val(Inf))
```

Return a sampler object that can be used to generate random values from `rng` for `x`.

When `sp = Sampler(rng, x, repetition)`, `rand(rng, sp)` will be used to draw random values, and should be defined accordingly.

`repetition` can be `Val(1)` or `Val(Inf)`, and should be used as a suggestion for deciding the amount of precomputation, if applicable.

`Random.SamplerType` and `Random.SamplerTrivial` are default fallbacks for *types* and *values*, respectively. `Random.SamplerSimple` can be used to store pre-computed values without defining extra types for only this purpose.

`Random.SamplerType` – Type.

```
SamplerType{T}()
```

A sampler for types, containing no other information. The default fallback for `Sampler` when called with types.

`Random.SamplerTrivial` – Type.

```
SamplerTrivial(x)
```

Create a sampler that just wraps the given value `x`. This is the default fall-back for values. The `eltype` of this sampler is equal to `eltype(x)`.

The recommended use case is sampling from values without precomputed data.

`Random.SamplerSimple` – Type.

```
SamplerSimple(x, data)
```

Create a sampler that wraps the given value `x` and the `data`. The `eltype` of this sampler is equal to `eltype(x)`.

The recommended use case is sampling from values with precomputed data.

Decoupling pre-computation from actually generating the values is part of the API, and is also available to the user. As an example, assume that `rand(rng, 1:20)` has to be called repeatedly in a loop: the way to take advantage of this decoupling is as follows:

```
rng = MersenneTwister()
sp = Random.Sampler(rng, 1:20) # or Random.Sampler(MersenneTwister, 1:20)
for x in X
    n = rand(rng, sp) # similar to n = rand(rng, 1:20)
    # use n
end
```

This is the mechanism that is also used in the standard library, e.g. by the default implementation of random array generation (like in `rand(1:20, 10)`).

Generating values from a type

Given a type T, it's currently assumed that if `rand(T)` is defined, an object of type T will be produced. `SamplerType` is the *default sampler for types*. In order to define random generation of values of type T, the `rand(rng::AbstractRNG, ::Random.SamplerType{T})` method should be defined, and should return values what `rand(rng, T)` is expected to return.

Let's take the following example: we implement a `Die` type, with a variable number n of sides, numbered from 1 to n. We want `rand(Die)` to produce a `Die` with a random number of up to 20 sides (and at least 4):

```
struct Die
    nsides::Int # number of sides
end

Random.rand(rng::AbstractRNG, ::Random.SamplerType{Die}) = Die(rand(rng, 4:20))

# output
```

Scalar and array methods for `Die` now work as expected:

```
julia> rand(Die)
Die(5)

julia> rand(MersenneTwister(0), Die)
Die(11)

julia> rand(Die, 3)
3-element Vector{Die}:
 Die(9)
 Die(15)
 Die(14)

julia> a = Vector{Die}(undef, 3); rand!(a)
3-element Vector{Die}:
 Die(19)
 Die(7)
 Die(17)
```

A simple sampler without pre-computed data

Here we define a sampler for a collection. If no pre-computed data is required, it can be implemented with a `SamplerTrivial` sampler, which is in fact the *default fallback for values*.

In order to define random generation out of objects of type S, the following method should be defined: `rand(rng::AbstractRNG, sp::Random.SamplerTrivial{S})`. Here, `sp` simply wraps an object of type S, which can be accessed via `sp[]`. Continuing the `Die` example, we want now to define `rand(d::Die)` to produce an `Int` corresponding to one of d's sides:

```
julia> Random.rand(rng::AbstractRNG, d::Random.SamplerTrivial{Die}) = rand(rng, 1:d[].nsides);

julia> rand(Die(4))
1

julia> rand(Die(4), 3)
3-element Vector{Any}:
 2
 3
 3
```

Given a collection type S, it's currently assumed that if `rand(::S)` is defined, an object of type `eltype(S)` will be produced. In the last example, a `Vector{Any}` is produced; the reason is that `eltype(Die) == Any`. The remedy is to define `Base.eltype(::Type{Die}) = Int`.

Generating values for an `AbstractFloat` type

`AbstractFloat` types are special-cased, because by default random values are not produced in the whole type domain, but rather in `[0,1)`. The following method should be implemented for `T <: AbstractFloat`: `Random.rand(::AbstractRNG, ::Random.SamplerTrivial{Random.CloseOpen01{T}})`

An optimized sampler with pre-computed data

Consider a discrete distribution, where numbers `1:n` are drawn with given probabilities that sum to one. When many values are needed from this distribution, the fastest method is using an alias table. We don't provide the algorithm for building such a table here, but suppose it is available in `make_alias_table(probabilities)` instead, and `draw_number(rng, alias_table)` can be used to draw a random number from it.

Suppose that the distribution is described by

```
struct DiscreteDistribution{V <: AbstractVector}
    probabilities::V
end
```

and that we *always* want to build an alias table, regardless of the number of values needed (we learn how to customize this below). The methods

```
Random.eltype(::Type{<:DiscreteDistribution}) = Int

function Random.Sampler(::Type{<:AbstractRNG}, distribution::DiscreteDistribution, ::Repetition)
    SamplerSimple(disribution, make_alias_table(distribution.probabilities))
end
```

should be defined to return a sampler with pre-computed data, then

```
function rand(rng::AbstractRNG, sp::SamplerSimple{<:DiscreteDistribution})
    draw_number(rng, sp.data)
end
```

will be used to draw the values.

Custom sampler types

The `SamplerSimple` type is sufficient for most use cases with precomputed data. However, in order to demonstrate how to use custom sampler types, here we implement something similar to `SamplerSimple`.

Going back to our `Die` example: `rand(::Die)` uses random generation from a range, so there is an opportunity for this optimization. We call our custom sampler `SamplerDie`.

```
import Random: Sampler, rand

struct SamplerDie <: Sampler{Int} # generates values of type Int
    die::Die
    sp::Sampler{Int} # this is an abstract type, so this could be improved
end

Sampler(RNG::Type{<:AbstractRNG}, die::Die, r::Random.Repetition) =
    SamplerDie(die, Sampler(RNG, 1:die.nsides, r))
# the `r` parameter will be explained later on

rand(rng::AbstractRNG, sp::SamplerDie) = rand(rng, sp.sp)
```

It's now possible to get a sampler with `sp = Sampler(rng, die)`, and use `sp` instead of `die` in any `rand` call involving `rng`. In the simplistic example above, `die` doesn't need to be stored in `SamplerDie` but this is often the case in practice.

Of course, this pattern is so frequent that the helper type used above, namely `Random.SamplerSimple`, is available, saving us the definition of `SamplerDie`: we could have implemented our decoupling with:

```
Sampler(RNG::Type{<:AbstractRNG}, die::Die, r::Random.Repetition) =
    SamplerSimple(die, Sampler(RNG, 1:die.nsides, r))

rand(rng::AbstractRNG, sp::SamplerSimple{Die}) = rand(rng, sp.data)
```

Here, `sp.data` refers to the second parameter in the call to the `SamplerSimple` constructor (in this case equal to `Sampler(rng, 1:die.nsides, r)`), while the `Die` object can be accessed via `sp[]`.

Like `SamplerDie`, any custom sampler must be a subtype of `Sampler{T}` where T is the type of the generated values. Note that `SamplerSimple(x, data) isa Sampler{eltype(x)}`, so this constrains what the first argument to `SamplerSimple` can be (it's recommended to use `SamplerSimple` like in the `Die` example, where x is simply forwarded while defining a `Sampler` method). Similarly, `SamplerTrivial(x) isa Sampler{eltype(x)}`.

Another helper type is currently available for other cases, `Random.SamplerTag`, but is considered as internal API, and can break at any time without proper deprecations.

Using distinct algorithms for scalar or array generation

In some cases, whether one wants to generate only a handful of values or a large number of values will have an impact on the choice of algorithm. This is handled with the third parameter of the `Sampler` constructor. Let's assume we defined two helper types for `Die`, say `SamplerDie1` which should be used to generate only few random values, and `SamplerDieMany` for many values. We can use those types as follows:

```
Sampler(RNG::Type{<:AbstractRNG}, die::Die, ::Val{1}) = SamplerDie1(...)
Sampler(RNG::Type{<:AbstractRNG}, die::Die, ::Val{Inf}) = SamplerDieMany(...)
```

Of course, rand must also be defined on those types (i.e. `rand(::AbstractRNG, ::SamplerDie1)` and `rand(::AbstractRNG, ::SamplerDieMany)`). Note that, as usual, `SamplerTrivial` and `SamplerSimple` can be used if custom types are not necessary.

Note: `Sampler(rng, x)` is simply a shorthand for `Sampler(rng, x, Val(Inf))`, and `Random.Repetition` is an alias for `Union{Val{1}, Val{Inf}}`.

Creating new generators

The API is not clearly defined yet, but as a rule of thumb:

1. any `rand` method producing "basic" types (`isbitstype` integer and floating types in `Base`) should be defined for this specific RNG, if they are needed;
2. other documented `rand` methods accepting an `AbstractRNG` should work out of the box, (provided the methods from 1) what are relied on are implemented), but can of course be specialized for this RNG if there is room for optimization;
3. `copy` for pseudo-RNGs should return an independent copy that generates the exact same random sequence as the original from that point when called in the same way. When this is not feasible (e.g. hardware-based RNGs), `copy` must not be implemented.

Concerning 1), a `rand` method may happen to work automatically, but it's not officially supported and may break without warnings in a subsequent release.

To define a new `rand` method for an hypothetical `MyRNG` generator, and a value specification `s` (e.g. `s == Int`, or `s == 1:10`) of type `S==typeof(s)` or `S==Type{s}` if `s` is a type, the same two methods as we saw before must be defined:

1. `Sampler(::Type{MyRNG}, ::S, ::Repetition)`, which returns an object of type say `SamplerS`
2. `rand(rng::MyRNG, sp::SamplerS)`

It can happen that `Sampler(rng::AbstractRNG, ::S, ::Repetition)` is already defined in the `Random` module. It would then be possible to skip step 1) in practice (if one wants to specialize generation for this particular RNG type), but the corresponding `SamplerS` type is considered as internal detail, and may be changed without warning.

Specializing array generation

In some cases, for a given RNG type, generating an array of random values can be more efficient with a specialized method than by merely using the decoupling technique explained before. This is for example the case for `MersenneTwister`, which natively writes random values in an array.

To implement this specialization for `MyRNG` and for a specification `s`, producing elements of type `S`, the following method can be defined: `rand!(rng::MyRNG, a::AbstractArray{S}, ::SamplerS)`, where `SamplerS` is the type of the sampler returned by `Sampler(MyRNG, s, Val(Inf))`. Instead of `AbstractArray`, it's possible to implement the functionality only for a subtype, e.g. `Array{S}`. The non-mutating array method of `rand` will automatically call this specialization internally.

Chapter 89

Reproducibility

By using an RNG parameter initialized with a given seed, you can reproduce the same pseudorandom number sequence when running your program multiple times. However, a minor release of Julia (e.g. 1.3 to 1.4) *may change* the sequence of pseudorandom numbers generated from a specific seed, in particular if `MersenneTwister` is used. (Even if the sequence produced by a low-level function like `rand` does not change, the output of higher-level functions like `randsubseq` may change due to algorithm updates.) Rationale: guaranteeing that pseudorandom streams never change prohibits many algorithmic improvements.

If you need to guarantee exact reproducibility of random data, it is advisable to simply *save the data* (e.g. as a supplementary attachment in a scientific publication). (You can also, of course, specify a particular Julia version and package manifest, especially if you require bit reproducibility.)

Software tests that rely on *specific* "random" data should also generally either save the data, embed it into the test code, or use third-party packages like StableRNGs.jl. On the other hand, tests that should pass for *most* random data (e.g. testing `A \ (A*x) ≈ x` for a random matrix `A = randn(n,n)`) can use an RNG with a fixed seed to ensure that simply running the test many times does not encounter a failure due to very improbable data (e.g. an extremely ill-conditioned matrix).

The statistical *distribution* from which random samples are drawn *is* guaranteed to be the same across any minor Julia releases.

Chapter 90

SHA

90.1 SHA functions

用法非常直接：

```
julia> using SHA

julia> bytes2hex(sha256("test"))
"9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08"
```

Each exported function (at the time of this writing, SHA-1, SHA-2 224, 256, 384 and 512, and SHA-3 224, 256, 384 and 512 functions are implemented) takes in either an `AbstractVector{UInt8}`, an `AbstractString` or an `IO` object. This makes it trivial to checksum a file:

```
shell> cat /tmp/test.txt
test
julia> using SHA

julia> open("/tmp/test.txt") do f
           sha2_256(f)
       end
32-element Array{UInt8,1}:
 0x9f
 0x86
 0xd0
 0x81
 0x88
 0x4c
 0x7d
 0x65
    ⋮
 0x5d
 0x6c
 0x15
 0xb0
 0xf0
 0x0a
 0x08
```

All SHA functions

Due to the colloquial usage of sha256 to refer to sha2_256, convenience functions are provided, mapping shaxxx() function calls to sha2_xxx(). For SHA-3, no such colloquialisms exist and the user must use the full sha3_xxx() names.

shaxxx() takes AbstractString and array-like objects (NTuple and Array) with elements of type UInt8.

SHA-1

SHA.sha1 – Function.

```
sha1(data)
```

Hash data using the sha1 algorithm and return the resulting digest. See also SHA1_CTX.

```
sha1(io::IO)
```

Hash data from io using sha1 algorithm.

SHA-2

SHA.sha224 – Function.

```
sha224(data)
```

Hash data using the sha224 algorithm and return the resulting digest. See also SHA2_224_CTX.

```
sha224(io::IO)
```

Hash data from io using sha224 algorithm.

SHA.sha256 – Function.

```
sha256(data)
```

Hash data using the sha256 algorithm and return the resulting digest. See also SHA2_256_CTX.

```
sha256(io::IO)
```

Hash data from io using sha256 algorithm.

SHA.sha384 – Function.

```
sha384(data)
```

Hash data using the sha384 algorithm and return the resulting digest. See also SHA2_384_CTX.

```
sha384(io::IO)
```

Hash data from io using sha384 algorithm.

SHA.sha512 – Function.

```
sha512(data)
```

Hash data using the sha512 algorithm and return the resulting digest. See also SHA2_512_CTX.

```
sha512(io::IO)
```

Hash data from io using sha512 algorithm.

SHA.sha2_224 – Function.

```
sha2_224(data)
```

Hash data using the sha2_224 algorithm and return the resulting digest. See also SHA2_224_CTX.

```
sha2_224(io::IO)
```

Hash data from io using sha2_224 algorithm.

SHA.sha2_256 – Function.

```
sha2_256(data)
```

Hash data using the sha2_256 algorithm and return the resulting digest. See also SHA2_256_CTX.

```
sha2_256(io::IO)
```

Hash data from io using sha2_256 algorithm.

SHA.sha2_384 – Function.

```
sha2_384(data)
```

Hash data using the sha2_384 algorithm and return the resulting digest. See also SHA2_384_CTX.

```
sha2_384(io::IO)
```

Hash data from io using sha2_384 algorithm.

SHA.sha2_512 – Function.

```
sha2_512(data)
```

Hash data using the sha2_512 algorithm and return the resulting digest. See also SHA2_512_CTX.

```
sha2_512(io::IO)
```

Hash data from io using sha2_512 algorithm.

SHA-3

SHA.sha3_224 – Function.

```
sha3_224(data)
```

Hash data using the sha3_224 algorithm and return the resulting digest. See also SHA3_224_CTX.

```
sha3_224(io::IO)
```

Hash data from io using sha3_224 algorithm.

SHA.sha3_256 – Function.

```
sha3_256(data)
```

Hash data using the sha3_256 algorithm and return the resulting digest. See also SHA3_256_CTX.

```
sha3_256(io::IO)
```

Hash data from io using sha3_256 algorithm.

SHA.sha3_384 – Function.

```
sha3_384(data)
```

Hash data using the sha3_384 algorithm and return the resulting digest. See also SHA3_384_CTX.

```
sha3_384(io::IO)
```

Hash data from io using sha3_384 algorithm.

SHA.sha3_512 – Function.

```
sha3_512(data)
```

Hash data using the sha3_512 algorithm and return the resulting digest. See also `SHA3_512_CTX`.

```
sha3_512(io::IO)
```

Hash data from io using sha3_512 algorithm.

90.2 Working with context

To create a hash from multiple items the `SHAX_XXX_CTX()` types can be used to create a stateful hash object that is updated with `update!` and finalized with `digest!`

```
julia> using SHA

julia> ctx = SHA2_256_CTX()
SHA2 256-bit hash state

julia> update!(ctx, b"some data")
0x0000000000000009

julia> update!(ctx, b"some more data")
0x0000000000000017

julia> digest!(ctx)
32-element Vector{UInt8}:
 0xbe
 0xcf
 0x23
 0xda
 0xaf
 0x02
 0xf7
 0xa3
 0x57
 0x92
    ⋮
 0x89
 0x4f
 0x59
 0xd8
 0xb3
 0xb4
 0x81
 0x8b
 0xc5
```

Note that, at the time of this writing, the SHA3 code is not optimized, and as such is roughly an order of magnitude slower than SHA2.

`SHA.update!` – Function.

```
update!(context, data[, datalen])
```

Update the SHA context with the bytes in data. See also `digest!` for finalizing the hash.

Examples

```
julia> ctx = SHA1_CTX()
SHA1 hash state

julia> update!(ctx, b"data to to be hashed")
```

SHA.`digest!` – Function.

```
digest!(context)
```

Finalize the SHA context and return the hash as array of bytes (Array{Uint8, 1}). Updating the context after calling `digest!` on it will error.

Examples

```
julia> ctx = SHA1_CTX()
SHA1 hash state

julia> update!(ctx, b"data to to be hashed")

julia> digest!(ctx)
20-element Array{UInt8,1}:
 0x83
 0xe4
 ⋮
 0x89
 0xf5

julia> update!(ctx, b"more data")
ERROR: Cannot update CTX after `digest!` has been called on it
[...]
```

All SHA context types

SHA-1

SHA.`SHA1_CTX` – Type.

```
SHA1_CTX()
```

Construct an empty SHA1 context.

SHA-2

Convenience types are also provided, where `SHAXXX_CTX` is a type alias for `SHA2_XXX_CTX`.

`SHA.SHA224_CTX` – Type.

```
SHA2_224_CTX()
```

Construct an empty SHA2_224 context.

`SHA.SHA256_CTX` – Type.

```
SHA2_256_CTX()
```

Construct an empty SHA2_256 context.

`SHA.SHA384_CTX` – Type.

```
SHA2_384()
```

Construct an empty SHA2_384 context.

`SHA.SHA512_CTX` – Type.

```
SHA2_512_CTX()
```

Construct an empty SHA2_512 context.

`SHA.SHA2_224_CTX` – Type.

```
SHA2_224_CTX()
```

Construct an empty SHA2_224 context.

`SHA.SHA2_256_CTX` – Type.

```
SHA2_256_CTX()
```

Construct an empty SHA2_256 context.

`SHA.SHA2_384_CTX` – Type.

```
SHA2_384()
```

Construct an empty SHA2_384 context.

`SHA.SHA2_512_CTX` – Type.

```
SHA2_512_CTX()
```

Construct an empty SHA2_512 context.

SHA-3

`SHA.SHA3_224_CTX` – Type.

```
SHA3_224_CTX()
```

Construct an empty SHA3_224 context.

`SHA.SHA3_256_CTX` – Type.

```
SHA3_256_CTX()
```

Construct an empty SHA3_256 context.

`SHA.SHA3_384_CTX` – Type.

```
SHA3_384_CTX()
```

Construct an empty SHA3_384 context.

`SHA.SHA3_512_CTX` – Type.

```
SHA3_512_CTX()
```

Construct an empty SHA3_512 context.

90.3 HMAC functions

```
julia> using SHA

julia> key = collect(codeunits("key_string"))
10-element Vector{UInt8}:
 0x6b
 0x65
 0x79
 0x5f
 0x73
 0x74
 0x72
```

```
 0x69
 0x6e
 0x67

julia> bytes2hex(hmac_sha3_256(key, "test-message"))
"bc49a6f2aa29b27ee5ed1e944edd7f3d153e8a01535d98b5e24dac9a589a6248"
```

To create a hash from multiple items, the `HMAC_CTX()` types can be used to create a stateful hash object that is updated with `update!` and finalized with `digest!`.

```
julia> using SHA

julia> key = collect(codeunits("key_string"))
10-element Vector{UInt8}:
 0x6b
 0x65
 0x79
 0x5f
 0x73
 0x74
 0x72
 0x69
 0x6e
 0x67

julia> ctx = HMAC_CTX(SHA3_256_CTX(), key);

julia> update!(ctx, b"test-")
0x000000000000000000000000000000008d

julia> update!(ctx, b"message")
0x00000000000000000000000000000000094

julia> bytes2hex(digest!(ctx))
"bc49a6f2aa29b27ee5ed1e944edd7f3d153e8a01535d98b5e24dac9a589a6248"
```

All HMAC functions

HMAC context type

`SHA.HMAC_CTX` – Type.

```
HMAC_CTX(ctx::CTX, key::Vector{UInt8}) where {CTX<:SHA_CTX}
```

Construct an empty HMAC_CTX context.

SHA-1

`SHA.hmac_sha1` – Function.

```
hmac_sha1(key, data)
```

Hash data using the sha1 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha1(key, io::IO)
```

Hash data from io with the passed key using sha1 algorithm.

SHA-2

SHA.hmac_sha224 – Function.

```
hmac_sha224(key, data)
```

Hash data using the sha224 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha224(key, io::IO)
```

Hash data from io with the passed key using sha224 algorithm.

SHA.hmac_sha256 – Function.

```
hmac_sha256(key, data)
```

Hash data using the sha256 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha256(key, io::IO)
```

Hash data from io with the passed key using sha256 algorithm.

SHA.hmac_sha384 – Function.

```
hmac_sha384(key, data)
```

Hash data using the sha384 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha384(key, io::IO)
```

Hash data from io with the passed key using sha384 algorithm.

SHA.hmac_sha512 – Function.

```
hmac_sha512(key, data)
```

Hash data using the sha512 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha512(key, io::IO)
```

Hash data from io with the passed key using sha512 algorithm.

SHA.hmac_sha2_224 – Function.

```
hmac_sha2_224(key, data)
```

Hash data using the sha2_224 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha2_224(key, io::IO)
```

Hash data from io with the passed key using sha2_224 algorithm.

SHA.hmac_sha2_256 – Function.

```
hmac_sha2_256(key, data)
```

Hash data using the sha2_256 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha2_256(key, io::IO)
```

Hash data from io with the passed key using sha2_256 algorithm.

SHA.hmac_sha2_384 – Function.

```
hmac_sha2_384(key, data)
```

Hash data using the sha2_384 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha2_384(key, io::IO)
```

Hash data from io with the passed key using sha2_384 algorithm.

SHA.hmac_sha2_512 – Function.

```
hmac_sha2_512(key, data)
```

Hash data using the sha2_512 algorithm using the passed key. See also HMAC_CTX.

```
hmac_sha2_512(key, io::IO)
```

Hash data from `io` with the passed key using `sha2_512` algorithm.

SHA-3

`SHA.hmac_sha3_224` – Function.

```
hmac_sha3_224(key, data)
```

Hash data using the `sha3_224` algorithm using the passed key. See also `HMAC_CTX`.

```
hmac_sha3_224(key, io::IO)
```

Hash data from `io` with the passed key using `sha3_224` algorithm.

`SHA.hmac_sha3_256` – Function.

```
hmac_sha3_256(key, data)
```

Hash data using the `sha3_256` algorithm using the passed key. See also `HMAC_CTX`.

```
hmac_sha3_256(key, io::IO)
```

Hash data from `io` with the passed key using `sha3_256` algorithm.

`SHA.hmac_sha3_384` – Function.

```
hmac_sha3_384(key, data)
```

Hash data using the `sha3_384` algorithm using the passed key. See also `HMAC_CTX`.

```
hmac_sha3_384(key, io::IO)
```

Hash data from `io` with the passed key using `sha3_384` algorithm.

`SHA.hmac_sha3_512` – Function.

```
hmac_sha3_512(key, data)
```

Hash data using the `sha3_512` algorithm using the passed key. See also `HMAC_CTX`.

```
hmac_sha3_512(key, io::IO)
```

Hash data from `io` with the passed key using `sha3_512` algorithm.

Chapter 91

序列化

Provides serialization of Julia objects.

`Serialization.serialize` – Function.

```
serialize(stream::IO, value)
```

Write an arbitrary value to a stream in an opaque format, such that it can be read back by `deserialize`. The read-back value will be as identical as possible to the original, but note that `Ptr` values are serialized as all-zero bit patterns (`NULL`).

An 8-byte identifying header is written to the stream first. To avoid writing the header, construct a `Serializer` and use it as the first argument to `serialize` instead. See also `Serialization.writeheader`.

The data format can change in minor (1.x) Julia releases, but files written by prior 1.x versions will remain readable. The main exception to this is when the definition of a type in an external package changes. If that occurs, it may be necessary to specify an explicit compatible version of the affected package in your environment. Renaming functions, even private functions, inside packages can also put existing files out of sync. Anonymous functions require special care: because their names are automatically generated, minor code changes can cause them to be renamed. Serializing anonymous functions should be avoided in files intended for long-term storage.

In some cases, the word size (32- or 64-bit) of the reading and writing machines must match. In rarer cases the OS or architecture must also match, for example when using packages that contain platform-dependent code.

```
serialize(filename::AbstractString, value)
```

Open a file and serialize the given value to it.

Julia 1.1

This method is available as of Julia 1.1.

`Serialization.deserialize` – Function.

```
deserialize(stream)
```

Read a value written by `serialize`. `deserialize` assumes the binary data read from `stream` is correct and has been serialized by a compatible implementation of `serialize`. `deserialize` is designed for simplicity and performance, and so does not validate the data read. Malformed data can result in process termination. The caller must ensure the integrity and correctness of data read from `stream`.

```
deserialize(filename::AbstractString)
```

Open a file and deserialize its contents.

> **Julia 1.1**
>
> This method is available as of Julia 1.1.

`Serialization.writeheader` – Function.

```
Serialization.writeheader(s::AbstractSerializer)
```

Write an identifying header to the specified serializer. The header consists of 8 bytes as follows:

Offset	Description
0	tag byte (0x37)
1-2	signature bytes "JL"
3	protocol version
4	bits 0-1: endianness: 0 = little, 1 = big
4	bits 2-3: platform: 0 = 32-bit, 1 = 64-bit
5-7	reserved

Chapter 92

共享数组

SharedArray represents an array, which is shared across multiple processes, on a single machine.

SharedArrays.SharedArray – Type.

```
SharedArray{T}(dims::NTuple; init=false, pids=Int[])
SharedArray{T,N}(...)
```

Construct a SharedArray of a bits type T and size dims across the processes specified by pids - all of which have to be on the same host. If N is specified by calling SharedArray{T,N}(dims), then N must match the length of dims.

If pids is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, localindices and indexpids will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

If an init function of the type initfn(S::SharedArray) is specified, it is called on all the participating workers.

The shared array is valid as long as a reference to the SharedArray object exists on the node which created the mapping.

```
SharedArray{T}(filename::AbstractString, dims::NTuple, [offset=0]; mode=nothing, init=false,
↪  pids=Int[])
SharedArray{T,N}(...)
```

Construct a SharedArray backed by the file filename, with element type T (must be a bits type) and size dims, across the processes specified by pids - all of which have to be on the same host. This file is mmapped into the host memory, with the following consequences:

- The array data must be represented in binary format (e.g., an ASCII format like CSV cannot be supported)
- Any changes you make to the array values (e.g., A[3] = 0) will also change the values on disk

If pids is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, localindices and indexpids will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

mode must be one of "r", "r+", "w+", or "a+", and defaults to "r+" if the file specified by filename already exists, or "w+" if not. If an init function of the type initfn(S::SharedArray) is specified, it is called on all the participating workers. You cannot specify an init function if the file is not writable.

offset allows you to skip the specified number of bytes at the beginning of the file.

source

SharedArrays.SharedVector – Type.

```
SharedVector
```

A one-dimensional SharedArray.

source

SharedArrays.SharedMatrix – Type.

```
SharedMatrix
```

A two-dimensional SharedArray.

source

Distributed.procs – Method.

```
procs(S::SharedArray)
```

Get the vector of processes mapping the shared array.

source

SharedArrays.sdata – Function.

```
sdata(S::SharedArray)
```

Return the actual Array object backing S.

source

SharedArrays.indexpids – Function.

```
indexpids(S::SharedArray)
```

Return the current worker's index in the list of workers mapping the SharedArray (i.e. in the same list returned by procs(S)), or 0 if the SharedArray is not mapped locally.

source

SharedArrays.localindices – Function.

```
localindices(S::SharedArray)
```

Return a range describing the ”default” indices to be handled by the current process. This range should be interpreted in the sense of linear indexing, i.e., as a sub-range of `1:length(S)`. In multi-process contexts, returns an empty range in the parent process (or any process for which `indexpids` returns 0).

It’s worth emphasizing that `localindices` exists purely as a convenience, and you can partition work on the array among workers any way you wish. For a `SharedArray`, all indices should be equally fast for each worker process.

source

Chapter 93

套接字

`Sockets.Sockets` – Module.

Support for sockets. Provides `IPAddr` and subtypes, `TCPSocket`, and `UDPSocket`.

`Sockets.connect` – Method.

```
connect([host], port::Integer) -> TCPSocket
```

Connect to the host `host` on port `port`.

`Sockets.connect` – Method.

```
connect(path::AbstractString) -> PipeEndpoint
```

Connect to the named pipe / UNIX domain socket at `path`.

> **Note**
>
> Path length on Unix is limited to somewhere between 92 and 108 bytes (cf. `man unix`).

`Sockets.listen` – Method.

```
listen([addr, ]port::Integer; backlog::Integer=BACKLOG_DEFAULT) -> TCPServer
```

Listen on port on the address specified by `addr`. By default this listens on `localhost` only. To listen on all interfaces pass `IPv4(0)` or `IPv6(0)` as appropriate. `backlog` determines how many connections can be pending (not having called `accept`) before the server will begin to reject them. The default value of `backlog` is 511.

`Sockets.listen` – Method.

```
listen(path::AbstractString) -> PipeServer
```

Create and listen on a named pipe / UNIX domain socket.

Note

Path length on Unix is limited to somewhere between 92 and 108 bytes (cf. `man unix`).

`Sockets.getaddrinfo` – Function.

```
getaddrinfo(host::AbstractString, IPAddr=IPv4) -> IPAddr
```

Gets the first IP address of the `host` of the specified `IPAddr` type. Uses the operating system's underlying getaddrinfo implementation, which may do a DNS lookup.

`Sockets.getipaddr` – Function.

```
getipaddr() -> IPAddr
```

Get an IP address of the local machine, preferring IPv4 over IPv6. Throws if no addresses are available.

```
getipaddr(addr_type::Type{T}) where T<:IPAddr -> T
```

Get an IP address of the local machine of the specified type. Throws if no addresses of the specified type are available.

This function is a backwards-compatibility wrapper around `getipaddrs`. New applications should use `getipaddrs` instead.

Examples

```
julia> getipaddr()
ip"192.168.1.28"

julia> getipaddr(IPv6)
ip"fe80::9731:35af:e1c5:6e49"
```

See also `getipaddrs`.

`Sockets.getipaddrs` – Function.

```
getipaddrs(addr_type::Type{T}=IPAddr; loopback::Bool=false) where T<:IPAddr -> Vector{T}
```

Get the IP addresses of the local machine.

Setting the optional `addr_type` parameter to `IPv4` or `IPv6` causes only addresses of that type to be returned.

The `loopback` keyword argument dictates whether loopback addresses (e.g. `ip"127.0.0.1"`, `ip"::1"`) are included.

Julia 1.2

This function is available as of Julia 1.2.

Examples

```
julia> getipaddrs()
5-element Array{IPAddr,1}:
 ip"198.51.100.17"
 ip"203.0.113.2"
 ip"2001:db8:8:4:445e:5fff:fe5d:5500"
 ip"2001:db8:8:4:c164:402e:7e3c:3668"
 ip"fe80::445e:5fff:fe5d:5500"

julia> getipaddrs(IPv6)
3-element Array{IPv6,1}:
 ip"2001:db8:8:4:445e:5fff:fe5d:5500"
 ip"2001:db8:8:4:c164:402e:7e3c:3668"
 ip"fe80::445e:5fff:fe5d:5500"
```

See also `islinklocaladdr`.

`Sockets.islinklocaladdr` – Function.

```
islinklocaladdr(addr::IPAddr)
```

Tests if an IP address is a link-local address. Link-local addresses are not guaranteed to be unique beyond their network segment, therefore routers do not forward them. Link-local addresses are from the address blocks `169.254.0.0/16` or `fe80::/10`.

Example

```
filter(!islinklocaladdr, getipaddrs())
```

`Sockets.getalladdrinfo` – Function.

```
getalladdrinfo(host::AbstractString) -> Vector{IPAddr}
```

Gets all of the IP addresses of the `host`. Uses the operating system's underlying `getaddrinfo` implementation, which may do a DNS lookup.

Example

```
julia> getalladdrinfo("google.com")
2-element Array{IPAddr,1}:
 ip"172.217.6.174"
 ip"2607:f8b0:4000:804::200e"
```

`Sockets.DNSError` – Type.

```
DNSError
```

The type of exception thrown when an error occurs in DNS lookup. The `host` field indicates the host URL string. The code field indicates the error code based on libuv.

`Sockets.getnameinfo` – Function.

```
getnameinfo(host::IPAddr) -> String
```

Performs a reverse-lookup for IP address to return a hostname and service using the operating system's underlying `getnameinfo` implementation.

Examples

```
julia> getnameinfo(IPv4("8.8.8.8"))
"google-public-dns-a.google.com"
```

`Sockets.getsockname` – Function.

```
getsockname(sock::Union{TCPServer, TCPSocket}) -> (IPAddr, UInt16)
```

Get the IP address and port that the given socket is bound to.

`Sockets.getpeername` – Function.

```
getpeername(sock::TCPSocket) -> (IPAddr, UInt16)
```

Get the IP address and port of the remote endpoint that the given socket is connected to. Valid only for connected TCP sockets.

`Sockets.IPAddr` – Type.

```
IPAddr
```

Abstract supertype for IP addresses. `IPv4` and `IPv6` are subtypes of this.

`Sockets.IPv4` – Type.

```
IPv4(host::Integer) -> IPv4
```

Return an IPv4 object from ip address `host` formatted as an `Integer`.

Examples

```
julia> IPv4(3223256218)
ip"192.30.252.154"
```

`Sockets.IPv6` – Type.

```
IPv6(host::Integer) -> IPv6
```

Return an IPv6 object from ip address `host` formatted as an `Integer`.

Examples

```
julia> IPv6(3223256218)
ip"::c01e:fc9a"
```

`Sockets.@ip_str` – Macro.

```
@ip_str str -> IPAddr
```

Parse `str` as an IP address.

Examples

```
julia> ip"127.0.0.1"
ip"127.0.0.1"

julia> @ip_str "2001:db8:0:0:0:0:2:1"
ip"2001:db8::2:1"
```

`Sockets.TCPSocket` – Type.

```
TCPSocket(; delay=true)
```

Open a TCP socket using libuv. If `delay` is true, libuv delays creation of the socket's file descriptor till the first `bind` call. `TCPSocket` has various fields to denote the state of the socket as well as its send/receive buffers.

`Sockets.UDPSocket` – Type.

```
UDPSocket()
```

Open a UDP socket using libuv. `UDPSocket` has various fields to denote the state of the socket.

`Sockets.accept` – Function.

```
accept(server[, client])
```

Accepts a connection on the given server and returns a connection to the client. An uninitialized client stream may be provided, in which case it will be used instead of creating a new stream.

`Sockets.listenany` – Function.

```
listenany([host::IPAddr,] port_hint; backlog::Integer=BACKLOG_DEFAULT) -> (UInt16, TCPServer)
```

Create a `TCPServer` on any port, using hint as a starting point. Returns a tuple of the actual port that the server was created on and the server itself. The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow.

`Base.bind` – Function.

```
bind(socket::Union{TCPServer, UDPSocket, TCPSocket}, host::IPAddr, port::Integer;
↪  ipv6only=false, reuseaddr=false, kws...)
```

Bind `socket` to the given `host:port`. Note that `0.0.0.0` will listen on all devices.

- The `ipv6only` parameter disables dual stack mode. If `ipv6only=true`, only an IPv6 stack is created.
- If `reuseaddr=true`, multiple threads or processes can bind to the same address without error if they all set `reuseaddr=true`, but only the last to bind will receive any traffic.

```
bind(chnl::Channel, task::Task)
```

Associate the lifetime of `chnl` with a task. `Channel` `chnl` is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on `chnl`.

The `chnl` object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed `Channel` objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

Examples

```
julia> c = Channel(0);

julia> task = @async foreach(i->put!(c, i), 1:4);

julia> bind(c,task);

julia> for i in c
           @show i
       end;
i = 1
i = 2
```

```
i = 3
i = 4

julia> isopen(c)
false
```

```
julia> c = Channel(0);

julia> task = @async (put!(c, 1); error("foo"));

julia> bind(c, task);

julia> take!(c)
1

julia> put!(c, 1);
ERROR: TaskFailedException
Stacktrace:
[...]
    nested task error: foo
[...]
```

source

`Sockets.send` – Function.

```
send(socket::UDPSocket, host::IPAddr, port::Integer, msg)
```

Send `msg` over `socket` to `host:port`.

`Sockets.recv` – Function.

```
recv(socket::UDPSocket)
```

Read a UDP packet from the specified socket, and return the bytes received. This call blocks.

`Sockets.recvfrom` – Function.

```
recvfrom(socket::UDPSocket) -> (host_port, data)
```

Read a UDP packet from the specified socket, returning a tuple of `(host_port, data)`, where `host_port` will be an InetAddr{IPv4} or InetAddr{IPv6}, as appropriate.

Julia 1.3

Prior to Julia version 1.3, the first returned value was an address (`IPAddr`). In version 1.3 it was changed to an `InetAddr`.

`Sockets.setopt` – Function.

```
setopt(sock::UDPSocket; multicast_loop=nothing, multicast_ttl=nothing, enable_broadcast=nothing,
↪  ttl=nothing)
```

Set UDP socket options.

- `multicast_loop`: loopback for multicast packets (default: `true`).
- `multicast_ttl`: TTL for multicast packets (default: `nothing`).
- `enable_broadcast`: flag must be set to `true` if socket will be used for broadcast messages, or else the UDP system will return an access error (default: `false`).
- `ttl`: Time-to-live of packets sent on the socket (default: `nothing`).

`Sockets.nagle` – Function.

```
nagle(socket::Union{TCPServer, TCPSocket}, enable::Bool)
```

Enables or disables Nagle's algorithm on a given TCP server or socket.

Julia 1.3

This function requires Julia 1.3 or later.

`Sockets.quickack` – Function.

```
quickack(socket::Union{TCPServer, TCPSocket}, enable::Bool)
```

On Linux systems, the TCP_QUICKACK is disabled or enabled on `socket`.

Chapter 94

稀疏数组

Julia 在 SparseArrays 标准库模块中提供了对稀疏向量和稀疏矩阵的支持。与稠密数组相比，包含足够多零值的稀疏数组在以特殊的数据结构存储时可以节省大量的空间和运算时间。

External packages which implement different sparse storage types, multidimensional sparse arrays, and more can be found in Noteworthy external packages

94.1 压缩稀疏列（CSC）稀疏矩阵存储

在 Julia 中，稀疏矩阵是按照压缩稀疏列（CSC）格式存储的。Julia 稀疏矩阵具有 SparseMatrixCSC{Tv,Ti} 类型，其中 Tv 是存储值的类型，Ti 是存储列指针和行索引的整型类型。SparseMatrixCSC 的内部表示如下所示：

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrixCSC{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column j is in colptr[j]:(colptr[j+1]-1)
    rowval::Vector{Ti}      # Row indices of stored values
    nzval::Vector{Tv}       # Stored values, typically nonzeros
end
```

压缩稀疏列存储格式使得访问稀疏矩阵的列元素非常简单快速，而访问稀疏矩阵的行会非常缓慢。在 CSC 稀疏矩阵中执行类似插入新元素的操作也会非常慢。这是由于在稀疏矩阵中插入新元素时，在插入点之后的所有元素都要向后移动一位。

All operations on sparse matrices are carefully implemented to exploit the CSC data structure for performance, and to avoid expensive operations.

If you have data in CSC format from a different application or library, and wish to import it in Julia, make sure that you use 1-based indexing. The row indices in every column need to be sorted, and if they are not, the matrix will display incorrectly. If your SparseMatrixCSC object contains unsorted row indices, one quick way to sort them is by doing a double transpose. Since the transpose operation is lazy, make a copy to materialize each transpose.

In some applications, it is convenient to store explicit zero values in a SparseMatrixCSC. These *are* accepted by functions in Base (but there is no guarantee that they will be preserved in mutating operations). Such explicitly stored zeros are treated as structural nonzeros by many routines. The nnz function returns the number of elements explicitly stored in the sparse data structure, including non-structural zeros. In order to count the exact number of numerical nonzeros, use count(!iszero, x), which inspects every stored element

of a sparse matrix. dropzeros, and the in-place dropzeros!, can be used to remove stored zeros from the sparse matrix.

```
julia> A = sparse([1, 1, 2, 3], [1, 3, 2, 3], [0, 1, 2, 0])
3×3 SparseMatrixCSC{Int64, Int64} with 4 stored entries:
 0  ⋅  1
 ⋅  2  ⋅
 ⋅  ⋅  0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Int64, Int64} with 2 stored entries:
 ⋅  ⋅  1
 ⋅  2  ⋅
 ⋅  ⋅  ⋅
```

94.2 稀疏向量储存

Sparse vectors are stored in a close analog to compressed sparse column format for sparse matrices. In Julia, sparse vectors have the type `SparseVector{Tv,Ti}` where Tv is the type of the stored values and Ti the integer type for the indices. The internal representation is as follows:

```
struct SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
    n::Int              # Length of the sparse vector
    nzind::Vector{Ti}   # Indices of stored values
    nzval::Vector{Tv}   # Stored values, typically nonzeros
end
```

对于 `SparseMatrixCSC`，`SparseVector` 类型也能包含显示存储的，零值。(见 稀疏矩阵存储。)

94.3 稀疏向量与矩阵构造函数

创建一个稀疏矩阵的最简单的方法是使用一个与 Julia 提供的用来处理稠密矩阵的`zeros` 等价的函数。要产生一个稀疏矩阵，你可以用同样的名字加上 `sp` 前缀：

```
julia> spzeros(3)
3-element SparseVector{Float64, Int64} with 0 stored entries
```

`sparse` 函数通常是一个构建稀疏矩阵的便捷方法。例如，要构建一个稀疏矩阵，我们可以输入一个列索引向量 `I`，一个行索引向量 `J`，一个储存值的向量 `V`（这也叫作 COO（坐标）格式）。然后 `sparse(I,J,V)` 创建一个满足 `S[I[k], J[k]] = V[k]` 的稀疏矩阵。等价的稀疏向量构建函数是 `sparsevec`，它接受（行）索引向量 `I` 和储存值的向量 `V` 并创建一个满足 `R[I[k]] = V[k]` 的向量 `R`。

```
julia> I = [1, 4, 3, 5]; J = [4, 7, 18, 9]; V = [1, 2, -5, 3];

julia> S = sparse(I,J,V)
5×18 SparseMatrixCSC{Int64, Int64} with 4 stored entries:
⎡□□□□□□□□□⎤
⎣□□□□□□□□□⎦

julia> R = sparsevec(I,V)
```

```
5-element SparseVector{Int64, Int64} with 4 stored entries:
  [1]  =  1
  [3]  =  -5
  [4]  =  2
  [5]  =  3
```

The inverse of the `sparse` and `sparsevec` functions is `findnz`, which retrieves the inputs used to create the sparse array. `findall(!iszero, x)` returns the Cartesian indices of non-zero entries in x (including stored entries equal to zero).

```
julia> findnz(S)
([1, 4, 5, 3], [4, 7, 9, 18], [1, 2, 3, -5])

julia> findall(!iszero, S)
4-element Vector{CartesianIndex{2}}:
 CartesianIndex(1, 4)
 CartesianIndex(4, 7)
 CartesianIndex(5, 9)
 CartesianIndex(3, 18)

julia> findnz(R)
([1, 3, 4, 5], [1, -5, 2, 3])

julia> findall(!iszero, R)
4-element Vector{Int64}:
 1
 3
 4
 5
```

另一个创建稀疏数组的方法是使用 `sparse` 函数将一个稠密数组转化为稀疏数组：

```
julia> sparse(Matrix(1.0I, 5, 5))
5×5 SparseMatrixCSC{Float64, Int64} with 5 stored entries:
 1.0   ⋅    ⋅    ⋅    ⋅
  ⋅   1.0   ⋅    ⋅    ⋅
  ⋅    ⋅   1.0   ⋅    ⋅
  ⋅    ⋅    ⋅   1.0   ⋅
  ⋅    ⋅    ⋅    ⋅   1.0

julia> sparse([1.0, 0.0, 1.0])
3-element SparseVector{Float64, Int64} with 2 stored entries:
  [1]  =  1.0
  [3]  =  1.0
```

You can go in the other direction using the `Array` constructor. The `issparse` function can be used to query if a matrix is sparse.

```
julia> issparse(spzeros(5))
true
```

94.4 稀疏矩阵的操作

Arithmetic operations on sparse matrices also work as they do on dense matrices. Indexing of, assignment into, and concatenation of sparse matrices work in the same way as dense matrices. Indexing operations, especially assignment, are expensive, when carried out one element at a time. In many cases it may be better to convert the sparse matrix into `(I,J,V)` format using `findnz`, manipulate the values or the structure in the dense vectors `(I,J,V)`, and then reconstruct the sparse matrix.

94.5 Correspondence of dense and sparse methods

The following table gives a correspondence between built-in methods on sparse matrices and their corresponding methods on dense matrix types. In general, methods that generate sparse matrices differ from their dense counterparts in that the resulting matrix follows the same sparsity pattern as a given sparse matrix S, or that the resulting sparse matrix has density d, i.e. each matrix element has a probability d of being non-zero.

Details can be found in the Sparse Vectors and Matrices section of the standard library reference.

构造函数	密度	说明
`spzeros(m,n)`	`zeros(m,n)`	Creates a *m*-by-*n* matrix of zeros. (`spzeros(m,n)` is empty.)
`sparse(I,n,n)`	`Matrix(I,n,n)`	Creates a *n*-by-*n* identity matrix.
`sparse(A)`	`Array(S)`	Interconverts between dense and sparse formats.
`sprand(m,n,d)`	`rand(m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed uniformly on the half-open interval $[0, 1)$.
`sprandn(m,n,d)`	`randn(m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements distributed according to the standard normal (Gaussian) distribution.
`sprandn(rng,m,n,d)`	`randn(rng,m,n)`	Creates a *m*-by-*n* random matrix (of density *d*) with iid non-zero elements generated with the `rng` random number generator

Chapter 95

SparseArrays API

SparseArrays.AbstractSparseArray – Type.

```
AbstractSparseArray{Tv,Ti,N}
```

Supertype for N-dimensional sparse arrays (or array-like types) with elements of type Tv and index type Ti. SparseMatrixCSC, SparseVector and SuiteSparse.CHOLMOD.Sparse are subtypes of this.

source

SparseArrays.AbstractSparseVector – Type.

```
AbstractSparseVector{Tv,Ti}
```

Supertype for one-dimensional sparse arrays (or array-like types) with elements of type Tv and index type Ti. Alias for AbstractSparseArray{Tv,Ti,1}.

source

SparseArrays.AbstractSparseMatrix – Type.

```
AbstractSparseMatrix{Tv,Ti}
```

Supertype for two-dimensional sparse arrays (or array-like types) with elements of type Tv and index type Ti. Alias for AbstractSparseArray{Tv,Ti,2}.

source

SparseArrays.SparseVector – Type.

```
SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
```

Vector type for storing sparse vectors. Can be created by passing the length of the vector, a *sorted* vector of non-zero indices, and a vector of non-zero values.

For instance, the vector [5, 6, 0, 7] can be represented as

```
SparseVector(4, [1, 2, 4], [5, 6, 7])
```

This indicates that the element at index 1 is 5, at index 2 is 6, at index 3 is `zero(Int)`, and at index 4 is 7.

It may be more convenient to create sparse vectors directly from dense vectors using `sparse` as

```
sparse([5, 6, 0, 7])
```

yields the same sparse vector.

source

`SparseArrays.SparseMatrixCSC` – Type.

```
SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrixCSC{Tv,Ti}
```

Matrix type for storing sparse matrices in the Compressed Sparse Column format. The standard way of constructing SparseMatrixCSC is through the `sparse` function. See also `spzeros`, `spdiagm` and `sprand`.

source

`SparseArrays.sparse` – Function.

```
sparse(A)
```

Convert an AbstractMatrix A into a sparse matrix.

Examples

```
julia> A = Matrix(1.0I, 3, 3)
3×3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> sparse(A)
3×3 SparseMatrixCSC{Float64, Int64} with 3 stored entries:
 1.0   ⋅    ⋅
  ⋅   1.0   ⋅
  ⋅    ⋅   1.0
```

source

```
sparse(I, J, V,[ m, n, combine])
```

Create a sparse matrix S of dimensions `m x n` such that `S[I[k], J[k]] = V[k]`. The `combine` function is used to combine duplicates. If `m` and `n` are not specified, they are set to `maximum(I)` and `maximum(J)` respectively. If the `combine` function is not supplied, `combine` defaults to `+` unless the elements of `V` are

Booleans in which case combine defaults to |. All elements of I must satisfy 1 <= I[k] <= m, and all elements of J must satisfy 1 <= J[k] <= n. Numerical zeros in (I, J, V) are retained as structural nonzeros; to drop numerical zeros, use dropzeros!.

For additional documentation and an expert driver, see SparseArrays.sparse!.

Examples

```
julia> Is = [1; 2; 3];

julia> Js = [1; 2; 3];

julia> Vs = [1; 2; 3];

julia> sparse(Is, Js, Vs)
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 1  ⋅  ⋅
 ⋅  2  ⋅
 ⋅  ⋅  3
```

source

SparseArrays.sparse! – Function.

```
sparse!(I::AbstractVector{Ti}, J::AbstractVector{Ti}, V::AbstractVector{Tv},
        m::Integer, n::Integer, combine, klasttouch::Vector{Ti},
        csrrowptr::Vector{Ti}, csrcolval::Vector{Ti}, csrnzval::Vector{Tv},
        [csccolptr::Vector{Ti}], [cscrowval::Vector{Ti}, cscnzval::Vector{Tv}] ) where
        ↪  {Tv,Ti<:Integer}
```

Parent of and expert driver for sparse; see sparse for basic usage. This method allows the user to provide preallocated storage for sparse's intermediate objects and result as described below. This capability enables more efficient successive construction of SparseMatrixCSCs from coordinate representations, and also enables extraction of an unsorted-column representation of the result's transpose at no additional cost.

This method consists of three major steps: (1) Counting-sort the provided coordinate representation into an unsorted-row CSR form including repeated entries. (2) Sweep through the CSR form, simultaneously calculating the desired CSC form's column-pointer array, detecting repeated entries, and repacking the CSR form with repeated entries combined; this stage yields an unsorted-row CSR form with no repeated entries. (3) Counting-sort the preceding CSR form into a fully-sorted CSC form with no repeated entries.

Input arrays csrrowptr, csrcolval, and csrnzval constitute storage for the intermediate CSR forms and require length(csrrowptr) >= m + 1, length(csrcolval) >= length(I), and length(csrnzval >= length(I)). Input array klasttouch, workspace for the second stage, requires length(klasttouch) >= n. Optional input arrays csccolptr, cscrowval, and cscnzval constitute storage for the returned CSC form S. If necessary, these are resized automatically to satisfy length(csccolptr) = n + 1, length(cscrowval) = nnz(S) and length(cscnzval) = nnz(S); hence, if nnz(S) is unknown at the outset, passing in empty vectors of the appropriate type (Vector{Ti}() and Vector{Tv}() respectively) suffices, or calling the sparse! method neglecting cscrowval and cscnzval.

On return, csrrowptr, csrcolval, and csrnzval contain an unsorted-column representation of the result's transpose.

You may reuse the input arrays' storage (I, J, V) for the output arrays (csccolptr, cscrowval, cscnzval). For example, you may call sparse!(I, J, V, csrrowptr, csrcolval, csrnzval, I, J, V). Note that they will be resized to satisfy the conditions above.

For the sake of efficiency, this method performs no argument checking beyond 1 <= I[k] <= m and 1 <= J[k] <= n. Use with care. Testing with --check-bounds=yes is wise.

This method runs in O(m, n, length(I)) time. The HALFPERM algorithm described in F. Gustavson, "Two fast algorithms for sparse matrices: multiplication and permuted transposition," ACM TOMS 4(3), 250-269 (1978) inspired this method's use of a pair of counting sorts.

source

```
SparseArrays.sparse!(I, J, V, [m, n, combine]) -> SparseMatrixCSC
```

Variant of sparse! that re-uses the input vectors (I, J, V) for the final matrix storage. After construction the input vectors will alias the matrix buffers; S.colptr === I, S.rowval === J, and S.nzval === V holds, and they will be resize!d as necessary.

Note that some work buffers will still be allocated. Specifically, this method is a convenience wrapper around sparse!(I, J, V, m, n, combine, klasttouch, csrrowptr, csrcolval, csrnzval, csccolptr, cscrowval, cscnzval) where this method allocates klasttouch, csrrowptr, csrcolval, and csrnzval of appropriate size, but reuses I, J, and V for csccolptr, cscrowval, and cscnzval.

Arguments m, n, and combine defaults to maximum(I), maximum(J), and +, respectively.

Julia 1.10

This method requires Julia version 1.10 or later.

source

SparseArrays.sparsevec – Function.

```
sparsevec(I, V, [m, combine])
```

Create a sparse vector S of length m such that S[I[k]] = V[k]. Duplicates are combined using the combine function, which defaults to + if no combine argument is provided, unless the elements of V are Booleans in which case combine defaults to |.

Examples

```
julia> II = [1, 3, 3, 5]; V = [0.1, 0.2, 0.3, 0.2];

julia> sparsevec(II, V)
5-element SparseVector{Float64, Int64} with 3 stored entries:
  [1]  =  0.1
  [3]  =  0.5
  [5]  =  0.2

julia> sparsevec(II, V, 8, -)
8-element SparseVector{Float64, Int64} with 3 stored entries:
  [1]  =  0.1
```

```
  [3]  =  -0.1
  [5]  =  0.2

julia> sparsevec([1, 3, 1, 2, 2], [true, true, false, false, false])
3-element SparseVector{Bool, Int64} with 3 stored entries:
  [1]  =  1
  [2]  =  0
  [3]  =  1
```

source

```
sparsevec(d::Dict, [m])
```

Create a sparse vector of length `m` where the nonzero indices are keys from the dictionary, and the nonzero values are the values from the dictionary.

Examples

```
julia> sparsevec(Dict(1 => 3, 2 => 2))
2-element SparseVector{Int64, Int64} with 2 stored entries:
  [1]  =  3
  [2]  =  2
```

source

```
sparsevec(A)
```

Convert a vector `A` into a sparse vector of length `m`.

Examples

```
julia> sparsevec([1.0, 2.0, 0.0, 0.0, 3.0, 0.0])
6-element SparseVector{Float64, Int64} with 3 stored entries:
  [1]  =  1.0
  [2]  =  2.0
  [5]  =  3.0
```

source

`Base.similar` – Method.

```
similar(A::AbstractSparseMatrixCSC{Tv,Ti}, [::Type{TvNew}, ::Type{TiNew}, m::Integer,
↪  n::Integer]) where {Tv,Ti}
```

Create an uninitialized mutable array with the given element type, index type, and size, based upon the given source `SparseMatrixCSC`. The new sparse matrix maintains the structure of the original sparse matrix, except in the case where dimensions of the output matrix are different from the output.

The output matrix has zeros in the same locations as the input, but uninitialized values for the nonzero locations.

source

`SparseArrays.issparse` – Function.

```
issparse(S)
```

Returns `true` if `S` is sparse, and `false` otherwise.

Examples

```
julia> sv = sparsevec([1, 4], [2.3, 2.2], 10)
10-element SparseVector{Float64, Int64} with 2 stored entries:
  [1]  =  2.3
  [4]  =  2.2

julia> issparse(sv)
true

julia> issparse(Array(sv))
false
```

source

`SparseArrays.nnz` – Function.

```
nnz(A)
```

Returns the number of stored (filled) elements in a sparse array.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 2  ⋅  ⋅
 ⋅  2  ⋅
 ⋅  ⋅  2

julia> nnz(A)
3
```

source

`SparseArrays.findnz` – Function.

```
findnz(A::SparseMatrixCSC)
```

Return a tuple (I, J, V) where I and J are the row and column indices of the stored ("structurally non-zero") values in sparse matrix A, and V is a vector of the values.

Examples

```
julia> A = sparse([1 2 0; 0 0 3; 0 4 0])
3×3 SparseMatrixCSC{Int64, Int64} with 4 stored entries:
 1  2  ⋅
 ⋅  ⋅  3
 ⋅  4  ⋅

julia> findnz(A)
([1, 1, 3, 2], [1, 2, 2, 3], [1, 2, 4, 3])
```

source

SparseArrays.spzeros – Function.

```
spzeros([type,]m[,n])
```

Create a sparse vector of length m or sparse matrix of size m x n. This sparse array will not contain any nonzero values. No storage will be allocated for nonzero values during construction. The type defaults to Float64 if not specified.

Examples

```
julia> spzeros(3, 3)
3×3 SparseMatrixCSC{Float64, Int64} with 0 stored entries:
  ⋅    ⋅    ⋅
  ⋅    ⋅    ⋅
  ⋅    ⋅    ⋅

julia> spzeros(Float32, 4)
4-element SparseVector{Float32, Int64} with 0 stored entries
```

source

```
spzeros([type], I::AbstractVector, J::AbstractVector, [m, n])
```

Create a sparse matrix S of dimensions m x n with structural zeros at S[I[k], J[k]].

This method can be used to construct the sparsity pattern of the matrix, and is more efficient than using e.g. sparse(I, J, zeros(length(I))).

For additional documentation and an expert driver, see SparseArrays.spzeros!.

> **Julia 1.10**
>
> This methods requires Julia version 1.10 or later.

source

SparseArrays.spzeros! – Function.

```
spzeros!(::Type{Tv}, I::AbstractVector{Ti}, J::AbstractVector{Ti}, m::Integer, n::Integer,
         klasttouch::Vector{Ti}, csrrowptr::Vector{Ti}, csrcolval::Vector{Ti},
         [csccolptr::Vector{Ti}], [cscrowval::Vector{Ti}, cscnzval::Vector{Tv}]) where
         ↪  {Tv,Ti<:Integer}
```

Parent of and expert driver for `spzeros(I, J)` allowing user to provide preallocated storage for intermediate objects. This method is to `spzeros` what `SparseArrays.sparse!` is to `sparse`. See documentation for `SparseArrays.sparse!` for details and required buffer lengths.

Julia 1.10

This methods requires Julia version 1.10 or later.

source

```
SparseArrays.spzeros!(::Type{Tv}, I, J, [m, n]) -> SparseMatrixCSC{Tv}
```

Variant of `spzeros!` that re-uses the input vectors `I` and `J` for the final matrix storage. After construction the input vectors will alias the matrix buffers; `S.colptr === I` and `S.rowval === J` holds, and they will be `resize!`d as necessary.

Note that some work buffers will still be allocated. Specifically, this method is a convenience wrapper around `spzeros!(Tv, I, J, m, n, klasttouch, csrrowptr, csrcolval, csccolptr, cscrowval)` where this method allocates `klasttouch`, `csrrowptr`, and `csrcolval` of appropriate size, but reuses `I` and `J` for `csccolptr` and `cscrowval`.

Arguments `m` and `n` defaults to `maximum(I)` and `maximum(J)`.

Julia 1.10

This method requires Julia version 1.10 or later.

source

SparseArrays.spdiagm – Function.

```
spdiagm(kv::Pair{<:Integer,<:AbstractVector}...)
spdiagm(m::Integer, n::Integer, kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a sparse diagonal matrix from `Pair`s of vectors and diagonals. Each vector `kv.second` will be placed on the `kv.first` diagonal. By default, the matrix is square and its size is inferred from `kv`, but a non-square size `m×n` (padded with zeros as needed) can be specified by passing `m,n` as the first arguments.

Examples

```
julia> spdiagm(-1 => [1,2,3,4], 1 => [4,3,2,1])
5×5 SparseMatrixCSC{Int64, Int64} with 8 stored entries:
```

```
 ⋅  4  ⋅  ⋅  ⋅
 1  ⋅  3  ⋅  ⋅
 ⋅  2  ⋅  2  ⋅
 ⋅  ⋅  3  ⋅  1
 ⋅  ⋅  ⋅  4  ⋅
```

source

```
spdiagm(v::AbstractVector)
spdiagm(m::Integer, n::Integer, v::AbstractVector)
```

Construct a sparse matrix with elements of the vector as diagonal elements. By default (no given `m` and `n`), the matrix is square and its size is given by `length(v)`, but a non-square size `m×n` can be specified by passing `m` and `n` as the first arguments.

Julia 1.6

These functions require at least Julia 1.6.

Examples

```
julia> spdiagm([1,2,3])
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 1  ⋅  ⋅
 ⋅  2  ⋅
 ⋅  ⋅  3

julia> spdiagm(sparse([1,0,3]))
3×3 SparseMatrixCSC{Int64, Int64} with 2 stored entries:
 1  ⋅  ⋅
 ⋅  ⋅  ⋅
 ⋅  ⋅  3
```

source

`SparseArrays.sparse_hcat` – Function.

```
sparse_hcat(A...)
```

Concatenate along dimension 2. Return a SparseMatrixCSC object.

Julia 1.8

This method was added in Julia 1.8. It mimics previous concatenation behavior, where the concatenation with specialized "sparse" matrix types from LinearAlgebra.jl automatically yielded sparse output even in the absence of any SparseArray argument.

source

`SparseArrays.sparse_vcat` – Function.

```
sparse_vcat(A...)
```

Concatenate along dimension 1. Return a SparseMatrixCSC object.

> **Julia 1.8**
>
> This method was added in Julia 1.8. It mimics previous concatenation behavior, where the concatenation with specialized "sparse" matrix types from LinearAlgebra.jl automatically yielded sparse output even in the absence of any SparseArray argument.

source

`SparseArrays.sparse_hvcat` – Function.

```
sparse_hvcat(rows::Tuple{Vararg{Int}}, values...)
```

Sparse horizontal and vertical concatenation in one call. This function is called for block matrix syntax. The first argument specifies the number of arguments to concatenate in each block row.

> **Julia 1.8**
>
> This method was added in Julia 1.8. It mimics previous concatenation behavior, where the concatenation with specialized "sparse" matrix types from LinearAlgebra.jl automatically yielded sparse output even in the absence of any SparseArray argument.

source

`SparseArrays.blockdiag` – Function.

```
blockdiag(A...)
```

Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

Examples

```
julia> blockdiag(sparse(2I, 3, 3), sparse(4I, 2, 2))
5×5 SparseMatrixCSC{Int64, Int64} with 5 stored entries:
 2  ⋅  ⋅  ⋅  ⋅
 ⋅  2  ⋅  ⋅  ⋅
 ⋅  ⋅  2  ⋅  ⋅
 ⋅  ⋅  ⋅  4  ⋅
 ⋅  ⋅  ⋅  ⋅  4
```

source

`SparseArrays.sprand` – Function.

```
sprand([rng],[T::Type],m,[n],p::AbstractFloat)
sprand([rng],m,[n],p::AbstractFloat,[rfn=rand])
```

Create a random length `m` sparse vector or `m` by `n` sparse matrix, in which the probability of any element being nonzero is independently given by `p` (and hence the mean density of nonzeros is also exactly `p`). The optional `rng` argument specifies a random number generator, see Random Numbers. The optional `T` argument specifies the element type, which defaults to `Float64`.

By default, nonzero values are sampled from a uniform distribution using the rand function, i.e. by `rand(T)`, or `rand(rng, T)` if `rng` is supplied; for the default `T=Float64`, this corresponds to nonzero values sampled uniformly in `[0,1)`.

You can sample nonzero values from a different distribution by passing a custom `rfn` function instead of `rand`. This should be a function `rfn(k)` that returns an array of `k` random numbers sampled from the desired distribution; alternatively, if `rng` is supplied, it should instead be a function `rfn(rng, k)`.

Examples

```
julia> sprand(Bool, 2, 2, 0.5)
2×2 SparseMatrixCSC{Bool, Int64} with 2 stored entries:
 1  1
 ⋅  ⋅

julia> sprand(Float64, 3, 0.75)
3-element SparseVector{Float64, Int64} with 2 stored entries:
  [1]  =  0.795547
  [2]  =  0.49425
```

source

`SparseArrays.sprandn` – Function.

```
sprandn([rng][,Type],m[,n],p::AbstractFloat)
```

Create a random sparse vector of length `m` or sparse matrix of size `m` by `n` with the specified (independent) probability `p` of any entry being nonzero, where nonzero values are sampled from the normal distribution. The optional `rng` argument specifies a random number generator, see Random Numbers.

Julia 1.1

Specifying the output element type Type requires at least Julia 1.1.

Examples

```
julia> sprandn(2, 2, 0.75)
2×2 SparseMatrixCSC{Float64, Int64} with 3 stored entries:
 -1.20577     ⋅
  0.311817  -0.234641
```

source

SparseArrays.nonzeros – Function.

```
nonzeros(A)
```

Return a vector of the structural nonzero values in sparse array A. This includes zeros that are explicitly stored in the sparse array. The returned vector points directly to the internal nonzero storage of A, and any modifications to the returned vector will mutate A as well. See rowvals and nzrange.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 2  ⋅  ⋅
 ⋅  2  ⋅
 ⋅  ⋅  2

julia> nonzeros(A)
3-element Vector{Int64}:
 2
 2
 2
```

source

SparseArrays.rowvals – Function.

```
rowvals(A::AbstractSparseMatrixCSC)
```

Return a vector of the row indices of A. Any modifications to the returned vector will mutate A as well. Providing access to how the row indices are stored internally can be useful in conjunction with iterating over structural nonzero values. See also nonzeros and nzrange.

Examples

```
julia> A = sparse(2I, 3, 3)
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 2  ⋅  ⋅
 ⋅  2  ⋅
 ⋅  ⋅  2

julia> rowvals(A)
3-element Vector{Int64}:
 1
 2
 3
```

source

SparseArrays.nzrange – Function.

```
nzrange(A::AbstractSparseMatrixCSC, col::Integer)
```

Return the range of indices to the structural nonzero values of a sparse matrix column. In conjunction with `nonzeros` and `rowvals`, this allows for convenient iterating over a sparse matrix :

```
A = sparse(I,J,V)
rows = rowvals(A)
vals = nonzeros(A)
m, n = size(A)
for j = 1:n
   for i in nzrange(A, j)
      row = rows[i]
      val = vals[i]
      # perform sparse wizardry...
   end
end
```

Warning

Adding or removing nonzero elements to the matrix may invalidate the `nzrange`, one should not mutate the matrix while iterating.

source

```
nzrange(x::SparseVectorUnion, col)
```

Give the range of indices to the structural nonzero values of a sparse vector. The column index `col` is ignored (assumed to be 1).

source

`SparseArrays.droptol!` – Function.

```
droptol!(A::AbstractSparseMatrixCSC, tol)
```

Removes stored values from `A` whose absolute value is less than or equal to `tol`.

source

```
droptol!(x::AbstractCompressedVector, tol)
```

Removes stored values from `x` whose absolute value is less than or equal to `tol`.

source

`SparseArrays.dropzeros!` – Function.

```
dropzeros!(x::AbstractCompressedVector)
```

Removes stored numerical zeros from x.

For an out-of-place version, see dropzeros. For algorithmic information, see fkeep!.

source

SparseArrays.dropzeros – Function.

```
dropzeros(A::AbstractSparseMatrixCSC;)
```

Generates a copy of A and removes stored numerical zeros from that copy.

For an in-place version and algorithmic information, see dropzeros!.

Examples

```
julia> A = sparse([1, 2, 3], [1, 2, 3], [1.0, 0.0, 1.0])
3×3 SparseMatrixCSC{Float64, Int64} with 3 stored entries:
 1.0   ⋅    ⋅
  ⋅   0.0   ⋅
  ⋅    ⋅   1.0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Float64, Int64} with 2 stored entries:
 1.0   ⋅    ⋅
  ⋅    ⋅    ⋅
  ⋅    ⋅   1.0
```

source

```
dropzeros(x::AbstractCompressedVector)
```

Generates a copy of x and removes numerical zeros from that copy.

For an in-place version and algorithmic information, see dropzeros!.

Examples

```
julia> A = sparsevec([1, 2, 3], [1.0, 0.0, 1.0])
3-element SparseVector{Float64, Int64} with 3 stored entries:
  [1]  =  1.0
  [2]  =  0.0
  [3]  =  1.0

julia> dropzeros(A)
3-element SparseVector{Float64, Int64} with 2 stored entries:
  [1]  =  1.0
  [3]  =  1.0
```

source

`SparseArrays.permute` – Function.

```
permute(A::AbstractSparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
        q::AbstractVector{<:Integer}) where {Tv,Ti}
```

Bilaterally permute `A`, returning `PAQ` (`A[p,q]`). Column-permutation `q`'s length must match `A`'s column count (`length(q) == size(A, 2)`). Row-permutation `p`'s length must match `A`'s row count (`length(p) == size(A, 1)`).

For expert drivers and additional information, see `permute!`.

Examples

```
julia> A = spdiagm(0 => [1, 2, 3, 4], 1 => [5, 6, 7])
4×4 SparseMatrixCSC{Int64, Int64} with 7 stored entries:
 1  5  ⋅  ⋅
 ⋅  2  6  ⋅
 ⋅  ⋅  3  7
 ⋅  ⋅  ⋅  4

julia> permute(A, [4, 3, 2, 1], [1, 2, 3, 4])
4×4 SparseMatrixCSC{Int64, Int64} with 7 stored entries:
 ⋅  ⋅  ⋅  4
 ⋅  ⋅  3  7
 ⋅  2  6  ⋅
 1  5  ⋅  ⋅

julia> permute(A, [1, 2, 3, 4], [4, 3, 2, 1])
4×4 SparseMatrixCSC{Int64, Int64} with 7 stored entries:
 ⋅  ⋅  5  1
 ⋅  6  2  ⋅
 7  3  ⋅  ⋅
 4  ⋅  ⋅  ⋅
```

source

`Base.permute!` – Method.

```
permute!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{Tv,Ti},
         p::AbstractVector{<:Integer}, q::AbstractVector{<:Integer},
         [C::AbstractSparseMatrixCSC{Tv,Ti}]) where {Tv,Ti}
```

Bilaterally permute `A`, storing result `PAQ` (`A[p,q]`) in `X`. Stores intermediate result `(AQ)^T` (`transpose(A[:,q])`) in optional argument `C` if present. Requires that none of `X`, `A`, and, if present, `C` alias each other; to store result `PAQ` back into `A`, use the following method lacking `X`:

```
permute!(A::AbstractSparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
         q::AbstractVector{<:Integer}[, C::AbstractSparseMatrixCSC{Tv,Ti},
         [workcolptr::Vector{Ti}]]) where {Tv,Ti}
```

X's dimensions must match those of A (`size(X, 1) == size(A, 1)` and `size(X, 2) == size(A, 2)`), and X must have enough storage to accommodate all allocated entries in A (`length(rowvals(X)) >= nnz(A)` and `length(nonzeros(X)) >= nnz(A)`). Column-permutation q's length must match A's column count (`length(q) == size(A, 2)`). Row-permutation p's length must match A's row count (`length(p) == size(A, 1)`).

C's dimensions must match those of `transpose(A)` (`size(C, 1) == size(A, 2)` and `size(C, 2) == size(A, 1)`), and C must have enough storage to accommodate all allocated entries in A (`length(rowvals(C)) >= nnz(A)` and `length(nonzeros(C)) >= nnz(A)`).

For additional (algorithmic) information, and for versions of these methods that forgo argument checking, see (unexported) parent methods `unchecked_noalias_permute!` and `unchecked_aliasing_permute!`.

See also `permute`.

source

`SparseArrays.halfperm!` – Function.

```
halfperm!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{TvA,Ti},
          q::AbstractVector{<:Integer}, f::Function = identity) where {Tv,TvA,Ti}
```

Column-permute and transpose A, simultaneously applying f to each entry of A, storing the result `(f(A)Q)^T` (`map(f, transpose(A[:,q]))`) in X.

Element type Tv of X must match `f(::TvA)`, where TvA is the element type of A. X's dimensions must match those of `transpose(A)` (`size(X, 1) == size(A, 2)` and `size(X, 2) == size(A, 1)`), and X must have enough storage to accommodate all allocated entries in A (`length(rowvals(X)) >= nnz(A)` and `length(nonzeros(X)) >= nnz(A)`). Column-permutation q's length must match A's column count (`length(q) == size(A, 2)`).

This method is the parent of several methods performing transposition and permutation operations on `SparseMatrixCSC`s. As this method performs no argument checking, prefer the safer child methods (`[c]transpose[!]`, `permute[!]`) to direct use.

This method implements the HALFPERM algorithm described in F. Gustavson, "Two fast algorithms for sparse matrices: multiplication and permuted transposition," ACM TOMS 4(3), 250-269 (1978). The algorithm runs in `O(size(A, 1), size(A, 2), nnz(A))` time and requires no space beyond that passed in.

source

`SparseArrays.ftranspose!` – Function.

```
ftranspose!(X::AbstractSparseMatrixCSC{Tv,Ti}, A::AbstractSparseMatrixCSC{Tv,Ti}, f::Function)
↪  where {Tv,Ti}
```

Transpose A and store it in X while applying the function f to the non-zero elements. Does not remove the zeros created by f. `size(X)` must be equal to `size(transpose(A))`. No additional memory is allocated other than resizing the rowval and nzval of X, if needed.

See `halfperm!`

source

Chapter 96

Noteworthy external packages

Several other Julia packages provide sparse matrix implementations that should be mentioned:

1. SuiteSparseGraphBLAS.jl is a wrapper over the fast, multithreaded SuiteSparse:GraphBLAS C library. On CPU this is typically the fastest option, often significantly outperforming MKLSparse.

2. CUDA.jl exposes the CUSPARSE library for GPU sparse matrix operations.

3. SparseMatricesCSR.jl provides a Julia native implementation of the Compressed Sparse Rows (CSR) format.

4. MKLSparse.jl accelerates SparseArrays sparse-dense matrix operations using Intel's MKL library.

5. SparseArrayKit.jl available for multidimensional sparse arrays.

6. LuxurySparse.jl provides static sparse array formats, as well as a coordinate format.

7. ExtendableSparse.jl enables fast insertion into sparse matrices using a lazy approach to new stored indices.

Chapter 97

统计

统计模块包含了基本的统计函数。

`Statistics.std` – Function.

```
std(itr; corrected::Bool=true, mean=nothing[, dims])
```

Compute the sample standard deviation of collection `itr`.

The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `itr` is a sample drawn from the same unknown distribution, with the samples uncorrelated. For arrays, this computation is equivalent to calculating `sqrt(sum((itr .- mean(itr)).^2) / (length(itr) - 1))`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` with `n` the number of elements in `itr`.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the standard deviation over dimensions.

A pre-computed `mean` may be provided. When `dims` is specified, `mean` must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

> **Note**
>
> If array contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the standard deviation of non-missing values.

source

`Statistics.stdm` – Function.

```
stdm(itr, mean; corrected::Bool=true[, dims])
```

Compute the sample standard deviation of collection `itr`, with known mean(s) `mean`.

The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `itr` is a sample drawn from the same unknown distribution, with the samples uncorrelated. For arrays, this computation is equivalent to calculating `sqrt(sum((itr .- mean(itr)).^2)`

/ (length(itr) - 1)). If corrected is true, then the sum is scaled with n-1, whereas the sum is scaled with n if corrected is false with n the number of elements in itr.

If itr is an AbstractArray, dims can be provided to compute the standard deviation over dimensions. In that case, mean must be an array with the same shape as mean(itr, dims=dims) (additional trailing singleton dimensions are allowed).

Note

If array contains NaN or missing values, the result is also NaN or missing (missing takes precedence if array contains both). Use the skipmissing function to omit missing entries and compute the standard deviation of non-missing values.

source

Statistics.var – Function.

```
var(itr; corrected::Bool=true, mean=nothing[, dims])
```

Compute the sample variance of collection itr.

The algorithm returns an estimator of the generative distribution's variance under the assumption that each entry of itr is a sample drawn from the same unknown distribution, with the samples uncorrelated. For arrays, this computation is equivalent to calculating sum((itr .- mean(itr)).^2) / (length(itr) - 1). If corrected is true, then the sum is scaled with n-1, whereas the sum is scaled with n if corrected is false where n is the number of elements in itr.

If itr is an AbstractArray, dims can be provided to compute the variance over dimensions.

A pre-computed mean may be provided. When dims is specified, mean must be an array with the same shape as mean(itr, dims=dims) (additional trailing singleton dimensions are allowed).

Note

If array contains NaN or missing values, the result is also NaN or missing (missing takes precedence if array contains both). Use the skipmissing function to omit missing entries and compute the variance of non-missing values.

source

Statistics.varm – Function.

```
varm(itr, mean; dims, corrected::Bool=true)
```

Compute the sample variance of collection itr, with known mean(s) mean.

The algorithm returns an estimator of the generative distribution's variance under the assumption that each entry of itr is a sample drawn from the same unknown distribution, with the samples uncorrelated. For arrays, this computation is equivalent to calculating sum((itr .- mean(itr)).^2) / (length(itr) - 1). If corrected is true, then the sum is scaled with n-1, whereas the sum is scaled with n if corrected is false with n the number of elements in itr.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the variance over dimensions. In that case, mean must be an array with the same shape as `mean(itr, dims=dims)` (additional trailing singleton dimensions are allowed).

> **Note**
>
> If array contains `NaN` or `missing` values, the result is also `NaN` or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the variance of non-missing values.

source

`Statistics.cor` – Function.

```
cor(x::AbstractVector)
```

Return the number one.

source

```
cor(X::AbstractMatrix; dims::Int=1)
```

Compute the Pearson correlation matrix of the matrix X along the dimension `dims`.

source

```
cor(x::AbstractVector, y::AbstractVector)
```

Compute the Pearson correlation between the vectors `x` and `y`.

source

```
cor(X::AbstractVecOrMat, Y::AbstractVecOrMat; dims=1)
```

Compute the Pearson correlation between the vectors or matrices X and Y along the dimension `dims`.

source

`Statistics.cov` – Function.

```
cov(x::AbstractVector; corrected::Bool=true)
```

Compute the variance of the vector `x`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

source

```
cov(X::AbstractMatrix; dims::Int=1, corrected::Bool=true)
```

Compute the covariance matrix of the matrix X along the dimension `dims`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, dims)`.

source

```
cov(x::AbstractVector, y::AbstractVector; corrected::Bool=true)
```

Compute the covariance between the vectors `x` and `y`. If `corrected` is `true` (the default), computes $\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})^*$ where $*$ denotes the complex conjugate and `n = length(x) = length(y)`. If `corrected` is `false`, computes $\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})^*$.

source

```
cov(X::AbstractVecOrMat, Y::AbstractVecOrMat; dims::Int=1, corrected::Bool=true)
```

Compute the covariance between the vectors or matrices X and Y along the dimension `dims`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, dims) = size(Y, dims)`.

source

`Statistics.mean!` – Function.

```
mean!(r, v)
```

Compute the mean of `v` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> using Statistics

julia> v = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> mean!([1., 1.], v)
2-element Vector{Float64}:
 1.5
 3.5

julia> mean!([1. 1.], v)
1×2 Matrix{Float64}:
 2.0  3.0
```

source

`Statistics.mean` – Function.

```
mean(itr)
```

Compute the mean of all elements in a collection.

Note

If `itr` contains NaN or `missing` values, the result is also NaN or `missing` (`missing` takes precedence if array contains both). Use the `skipmissing` function to omit `missing` entries and compute the mean of non-missing values.

Examples

```
julia> using Statistics

julia> mean(1:20)
10.5

julia> mean([1, missing, 3])
missing

julia> mean(skipmissing([1, missing, 3]))
2.0
```

source

```
mean(f, itr)
```

Apply the function `f` to each element of collection `itr` and take the mean.

```
julia> using Statistics

julia> mean(√, [1, 2, 3])
1.3820881233139908

julia> mean([√1, √2, √3])
1.3820881233139908
```

source

```
mean(f, A::AbstractArray; dims)
```

Apply the function `f` to each element of array `A` and take the mean over dimensions `dims`.

Julia 1.3

This method requires at least Julia 1.3.

```
julia> using Statistics

julia> mean(√, [1, 2, 3])
1.3820881233139908

julia> mean([√1, √2, √3])
1.3820881233139908

julia> mean(√, [1 2 3; 4 5 6], dims=2)
2×1 Matrix{Float64}:
 1.3820881233139908
 2.2285192400943226
```

source

```
mean(A::AbstractArray; dims)
```

Compute the mean of an array over the given dimensions.

Julia 1.1

mean for empty arrays requires at least Julia 1.1.

Examples

```
julia> using Statistics

julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> mean(A, dims=1)
1×2 Matrix{Float64}:
 2.0  3.0

julia> mean(A, dims=2)
2×1 Matrix{Float64}:
 1.5
 3.5
```

source

`Statistics.median!` – Function.

```
median!(v)
```

Like `median`, but may overwrite the input vector.

source

`Statistics.median` – Function.

```
median(itr)
```

Compute the median of all elements in a collection. For an even number of elements no exact median element exists, so the result is equivalent to calculating mean of two median elements.

Note

If `itr` contains NaN or `missing` values, the result is also NaN or `missing` (`missing` takes precedence if `itr` contains both). Use the `skipmissing` function to omit `missing` entries and compute the median of non-missing values.

Examples

```
julia> using Statistics

julia> median([1, 2, 3])
2.0

julia> median([1, 2, 3, 4])
2.5

julia> median([1, 2, missing, 4])
missing

julia> median(skipmissing([1, 2, missing, 4]))
2.0
```

source

```
median(A::AbstractArray; dims)
```

Compute the median of an array along the given dimensions.

Examples

```
julia> using Statistics

julia> median([1 2; 3 4], dims=1)
1×2 Matrix{Float64}:
 2.0  3.0
```

source

`Statistics.middle` – Function.

```
middle(x)
```

Compute the middle of a scalar value, which is equivalent to x itself, but of the type of `middle(x, x)` for consistency.

source

```
middle(x, y)
```

Compute the middle of two numbers x and y, which is equivalent in both value and type to computing their mean (`(x + y) / 2`).

source

```
middle(a::AbstractArray)
```

Compute the middle of an array a, which consists of finding its extrema and then computing their mean.

```
julia> using Statistics

julia> middle(1:10)
5.5

julia> a = [1,2,3.6,10.9]
4-element Vector{Float64}:
  1.0
  2.0
  3.6
 10.9

julia> middle(a)
5.95
```

source

`Statistics.quantile!` – Function.

```
quantile!([q::AbstractArray, ] v::AbstractVector, p; sorted=false, alpha::Real=1.0,
↪  beta::Real=alpha)
```

Compute the quantile(s) of a vector v at a specified probability or vector or tuple of probabilities p on the interval [0,1]. If p is a vector, an optional output array q may also be specified. (If not provided, a new output array is created.) The keyword argument `sorted` indicates whether v can be assumed to be sorted; if `false` (the default), then the elements of v will be partially sorted in-place.

Samples quantile are defined by `Q(p) = (1-γ)*x[j] + γ*x[j+1]`, where `x[j]` is the j-th order statistic of v, `j = floor(n*p + m)`, `m = alpha + p*(1 - alpha - beta)` and `γ = n*p + m - j`.

By default (`alpha = beta = 1`), quantiles are computed via linear interpolation between the points `((k-1)/(n-1), x[k])`, for `k = 1:n` where `n = length(v)`. This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R and NumPy default.

The keyword arguments `alpha` and `beta` correspond to the same parameters in Hyndman and Fan, setting them to different values allows to calculate quantiles with any of the methods 4-9 defined in this paper:

- Def. 4: alpha=0, beta=1
- Def. 5: alpha=0.5, beta=0.5
- Def. 6: alpha=0, beta=0 (Excel PERCENTILE.EXC, Python default, Stata altdef)
- Def. 7: alpha=1, beta=1 (Julia, R and NumPy default, Excel PERCENTILE and PERCENTILE.INC, Python 'inclusive')
- Def. 8: alpha=1/3, beta=1/3
- Def. 9: alpha=3/8, beta=3/8

Note

An ArgumentError is thrown if v contains NaN or missing values.

References

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365
- Quantile on Wikipedia details the different quantile definitions

Examples

```
julia> using Statistics

julia> x = [3, 2, 1];

julia> quantile!(x, 0.5)
2.0

julia> x
3-element Vector{Int64}:
 1
 2
 3

julia> y = zeros(3);

julia> quantile!(y, x, [0.1, 0.5, 0.9]) === y
true

julia> y
3-element Vector{Float64}:
 1.2000000000000002
 2.0
 2.8000000000000003
```

source

Statistics.quantile – Function.

```
quantile(itr, p; sorted=false, alpha::Real=1.0, beta::Real=alpha)
```

Compute the quantile(s) of a collection `itr` at a specified probability or vector or tuple of probabilities `p` on the interval [0,1]. The keyword argument `sorted` indicates whether `itr` can be assumed to be sorted.

Samples quantile are defined by `Q(p) = (1-γ)*x[j] + γ*x[j+1]`, where `x[j]` is the j-th order statistic of `itr`, `j = floor(n*p + m)`, `m = alpha + p*(1 - alpha - beta)` and `γ = n*p + m - j`.

By default (`alpha = beta = 1`), quantiles are computed via linear interpolation between the points `((k-1)/(n-1), x[k])`, for `k = 1:n` where `n = length(itr)`. This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R and NumPy default.

The keyword arguments `alpha` and `beta` correspond to the same parameters in Hyndman and Fan, setting them to different values allows to calculate quantiles with any of the methods 4-9 defined in this paper:

- Def. 4: `alpha=0`, `beta=1`
- Def. 5: `alpha=0.5`, `beta=0.5`
- Def. 6: `alpha=0`, `beta=0` (Excel `PERCENTILE.EXC`, Python default, Stata `altdef`)
- Def. 7: `alpha=1`, `beta=1` (Julia, R and NumPy default, Excel `PERCENTILE` and `PERCENTILE.INC`, Python `'inclusive'`)
- Def. 8: `alpha=1/3`, `beta=1/3`
- Def. 9: `alpha=3/8`, `beta=3/8`

> **Note**
>
> An `ArgumentError` is thrown if `v` contains `NaN` or `missing` values. Use the `skipmissing` function to omit `missing` entries and compute the quantiles of non-missing values.

References

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365
- Quantile on Wikipedia details the different quantile definitions

Examples

```
julia> using Statistics

julia> quantile(0:20, 0.5)
10.0

julia> quantile(0:20, [0.1, 0.5, 0.9])
3-element Vector{Float64}:
  2.0
 10.0
 18.000000000000004

julia> quantile(skipmissing([1, 10, missing]), 0.5)
5.5
```

source

Chapter 98

TOML

TOML.jl is a Julia standard library for parsing and writing TOML v1.0 files.

98.1 Parsing TOML data

```
julia> using TOML

julia> data = """
           [database]
           server = "192.168.1.1"
           ports = [ 8001, 8001, 8002 ]
       """;

julia> TOML.parse(data)
Dict{String, Any} with 1 entry:
  "database" => Dict{String, Any}("server"=>"192.168.1.1", "ports"=>[8001, 8001⋯
```

To parse a file, use `TOML.parsefile`. If the file has a syntax error, an exception is thrown:

```
julia> using TOML

julia> TOML.parse("""
           value = 0.0.0
       """)
ERROR: TOML Parser error:
none:1:16 error: failed to parse value
      value = 0.0.0
                 ^
[...]
```

There are other versions of the parse functions (`TOML.tryparse` and [`TOML.tryparsefile`]) that instead of throwing exceptions on parser error returns a `TOML.ParserError` with information:

```
julia> using TOML

julia> err = TOML.tryparse("""
           value = 0.0.0
```

```
       """);

julia> err.type
ErrGenericValueError::ErrorType = 14

julia> err.line
1

julia> err.column
16
```

98.2 Exporting data to TOML file

The `TOML.print` function is used to print (or serialize) data into TOML format.

```
julia> using TOML

julia> data = Dict(
          "names" => ["Julia", "Julio"],
          "age" => [10, 20],
       );

julia> TOML.print(data)
names = ["Julia", "Julio"]
age = [10, 20]

julia> fname = tempname();

julia> open(fname, "w") do io
           TOML.print(io, data)
       end

julia> TOML.parsefile(fname)
Dict{String, Any} with 2 entries:
  "names" => ["Julia", "Julio"]
  "age"   => [10, 20]
```

Keys can be sorted according to some value

```
julia> using TOML

julia> TOML.print(Dict(
       "abc"  => 1,
       "ab"   => 2,
       "abcd" => 3,
       ); sorted=true, by=length)
ab = 2
abc = 1
abcd = 3
```

For custom structs, pass a function that converts the struct to a supported type

```
julia> using TOML

julia> struct MyStruct
           a::Int
           b::String
       end

julia> TOML.print(Dict("foo" => MyStruct(5, "bar"))) do x
           x isa MyStruct && return [x.a, x.b]
           error("unhandled type $(typeof(x))")
       end
foo = [5, "bar"]
```

98.3 References

`TOML.parse` – Function.

```
parse(x::Union{AbstractString, IO})
parse(p::Parser, x::Union{AbstractString, IO})
```

Parse the string or stream x, and return the resulting table (dictionary). Throw a `ParserError` upon failure.

See also `TOML.tryparse`.

`TOML.parsefile` – Function.

```
parsefile(f::AbstractString)
parsefile(p::Parser, f::AbstractString)
```

Parse file `f` and return the resulting table (dictionary). Throw a `ParserError` upon failure.

See also `TOML.tryparsefile`.

`TOML.tryparse` – Function.

```
tryparse(x::Union{AbstractString, IO})
tryparse(p::Parser, x::Union{AbstractString, IO})
```

Parse the string or stream x, and return the resulting table (dictionary). Return a `ParserError` upon failure.

See also `TOML.parse`.

`TOML.tryparsefile` – Function.

```
tryparsefile(f::AbstractString)
tryparsefile(p::Parser, f::AbstractString)
```

Parse file `f` and return the resulting table (dictionary). Return a `ParserError` upon failure.

See also `TOML.parsefile`.

`TOML.print` – Function.

```
print([to_toml::Function], io::IO [=stdout], data::AbstractDict; sorted=false, by=identity)
```

Write `data` as TOML syntax to the stream `io`. If the keyword argument `sorted` is set to `true`, sort tables according to the function given by the keyword argument `by`.

The following data types are supported: `AbstractDict`, `AbstractVector`, `AbstractString`, `Integer`, `AbstractFloat`, `Bool`, `Dates.DateTime`, `Dates.Time`, `Dates.Date`. Note that the integers and floats need to be convertible to `Float64` and `Int64` respectively. For other data types, pass the function `to_toml` that takes the data types and returns a value of a supported type.

`TOML.Parser` – Type.

```
Parser()
```

Constructor for a TOML `Parser`. Note that in most cases one does not need to explicitly create a `Parser` but instead one directly use use `TOML.parsefile` or `TOML.parse`. Using an explicit parser will however reuse some internal data structures which can be beneficial for performance if a larger number of small files are parsed.

`TOML.ParserError` – Type.

```
ParserError
```

Type that is returned from `tryparse` and `tryparsefile` when parsing fails. It contains (among others) the following fields:

- `pos`, the position in the string when the error happened
- `table`, the result that so far was successfully parsed
- `type`, an error type, different for different types of errors

Chapter 99

Tar

Tar.create – Function.

```
create(
    [ predicate, ] dir, [ tarball ];
    [ skeleton, ] [ portable = false ]
) -> tarball

    predicate :: String --> Bool
    dir       :: AbstractString
    tarball   :: Union{AbstractString, AbstractCmd, IO}
    skeleton  :: Union{AbstractString, AbstractCmd, IO}
    portable  :: Bool
```

Create a tar archive (”tarball”) of the directory `dir`. The resulting archive is written to the path `tarball` or if no path is specified, a temporary path is created and returned by the function call. If `tarball` is an IO object then the tarball content is written to that handle instead (the handle is left open).

If a `predicate` function is passed, it is called on each system path that is encountered while recursively searching `dir` and `path` is only included in the tarball if `predicate(path)` is true. If `predicate(path)` returns false for a directory, then the directory is excluded entirely: nothing under that directory will be included in the archive.

If the `skeleton` keyword is passed then the file or IO handle given is used as a ”skeleton” to generate the tarball. You create a skeleton file by passing the `skeleton` keyword to the `extract` command. If `create` is called with that skeleton file and the extracted files haven't changed, an identical tarball is recreated. The `skeleton` and `predicate` arguments cannot be used together.

If the `portable` flag is true then path names are checked for validity on Windows, which ensures that they don't contain illegal characters or have names that are reserved. See https://stackoverflow.com/a/31976060/659248 for details.

Tar.extract – Function.

```
extract(
    [ predicate, ] tarball, [ dir ];
    [ skeleton = <none>, ]
    [ copy_symlinks = <auto>, ]
    [ set_permissions = true, ]
```

```
) -> dir

    predicate       :: Header --> Bool
    tarball         :: Union{AbstractString, AbstractCmd, IO}
    dir             :: AbstractString
    skeleton        :: Union{AbstractString, AbstractCmd, IO}
    copy_symlinks   :: Bool
    set_permissions :: Bool
```

Extract a tar archive ("tarball") located at the path `tarball` into the directory `dir`. If `tarball` is an IO object instead of a path, then the archive contents will be read from that IO stream. The archive is extracted to `dir` which must either be an existing empty directory or a non-existent path which can be created as a new directory. If `dir` is not specified, the archive is extracted into a temporary directory which is returned by `extract`.

If a `predicate` function is passed, it is called on each `Header` object that is encountered while extracting `tarball` and the entry is only extracted if the `predicate(hdr)` is true. This can be used to selectively extract only parts of an archive, to skip entries that cause `extract` to throw an error, or to record what is extracted during the extraction process.

Before it is passed to the predicate function, the `Header` object is somewhat modified from the raw header in the tarball: the `path` field is normalized to remove `.` entries and replace multiple consecutive slashes with a single slash. If the entry has type `:hardlink`, the link target path is normalized the same way so that it will match the path of the target entry; the size field is set to the size of the target path (which must be an already-seen file).

If the `skeleton` keyword is passed then a "skeleton" of the extracted tarball is written to the file or IO handle given. This skeleton file can be used to recreate an identical tarball by passing the `skeleton` keyword to the `create` function. The `skeleton` and `predicate` arguments cannot be used together.

If `copy_symlinks` is `true` then instead of extracting symbolic links as such, they will be extracted as copies of what they link to if they are internal to the tarball and if it is possible to do so. Non-internal symlinks, such as a link to `/etc/passwd` will not be copied. Symlinks which are in any way cyclic will also not be copied and will instead be skipped. By default, `extract` will detect whether symlinks can be created in `dir` or not and will automatically copy symlinks if they cannot be created.

If `set_permissions` is `false`, no permissions are set on the extracted files.

`Tar.list` – Function.

```
list(tarball; [ strict = true ]) -> Vector{Header}
list(callback, tarball; [ strict = true ])

    callback  :: Header, [ <data> ] --> Any
    tarball   :: Union{AbstractString, AbstractCmd, IO}
    strict    :: Bool
```

List the contents of a tar archive ("tarball") located at the path `tarball`. If `tarball` is an IO handle, read the tar contents from that stream. Returns a vector of `Header` structs. See `Header` for details.

If a `callback` is provided then instead of returning a vector of headers, the callback is called on each `Header`. This can be useful if the number of items in the tarball is large or if you want examine items prior to an error in the tarball. If the `callback` function can accept a second argument of either type

Vector{UInt8} or Vector{Pair{Symbol, String}} then it will be called with a representation of the raw header data either as a single byte vector or as a vector of pairs mapping field names to the raw data for that field (if these fields are concatenated together, the result is the raw data of the header).

By default list will error if it encounters any tarball contents which the extract function would refuse to extract. With strict=false it will skip these checks and list all the the contents of the tar file whether extract would extract them or not. Beware that malicious tarballs can do all sorts of crafty and unexpected things to try to trick you into doing something bad.

If the tarball argument is a skeleton file (see extract and create) then list will detect that from the file header and appropriately list or iterate the headers of the skeleton file.

Tar.rewrite – Function.

```
rewrite(
    [ predicate, ] old_tarball, [ new_tarball ];
    [ portable = false, ]
) -> new_tarball

    predicate   :: Header --> Bool
    old_tarball :: Union{AbstractString, AbtractCmd, IO}
    new_tarball :: Union{AbstractString, AbtractCmd, IO}
    portable    :: Bool
```

Rewrite old_tarball to the standard format that create generates, while also checking that it doesn't contain anything that would cause extract to raise an error. This is functionally equivalent to doing

```
Tar.create(Tar.extract(predicate, old_tarball), new_tarball)
```

However, it never extracts anything to disk and instead uses the seek function to navigate the old tarball's data. If no new_tarball argument is passed, the new tarball is written to a temporary file whose path is returned.

If a predicate function is passed, it is called on each Header object that is encountered while extracting old_tarball and the entry is skipped unless predicate(hdr) is true. This can be used to selectively rewrite only parts of an archive, to skip entries that would cause extract to throw an error, or to record what content is encountered during the rewrite process.

Before it is passed to the predicate function, the Header object is somewhat modified from the raw header in the tarball: the path field is normalized to remove . entries and replace multiple consecutive slashes with a single slash. If the entry has type :hardlink, the link target path is normalized the same way so that it will match the path of the target entry; the size field is set to the size of the target path (which must be an already-seen file).

If the portable flag is true then path names are checked for validity on Windows, which ensures that they don't contain illegal characters or have names that are reserved. See https://stackoverflow.com/a/31976060/659248 for details.

Tar.tree_hash – Function.

```
tree_hash([ predicate, ] tarball;
          [ algorithm = "git-sha1", ]
          [ skip_empty = false ]) -> hash::String
```

```
    predicate  :: Header --> Bool
    tarball    :: Union{AbstractString, AbstractCmd, IO}
    algorithm  :: AbstractString
    skip_empty :: Bool
```

Compute a tree hash value for the file tree that the tarball contains. By default, this uses git's tree hashing algorithm with the SHA1 secure hash function (like current versions of git). This means that for any tarball whose file tree git can represent—i.e. one with only files, symlinks and non-empty directories—the hash value computed by this function will be the same as the hash value git would compute for that file tree. Note that tarballs can represent file trees with empty directories, which git cannot store, and this function can generate hashes for those, which will, by default (see `skip_empty` below for how to change this behavior), differ from the hash of a tarball which omits those empty directories. In short, the hash function agrees with git on all trees which git can represent, but extends (in a consistent way) the domain of hashable trees to other trees which git cannot represent.

If a `predicate` function is passed, it is called on each `Header` object that is encountered while processing `tarball` and an entry is only hashed if `predicate(hdr)` is true. This can be used to selectively hash only parts of an archive, to skip entries that cause `extract` to throw an error, or to record what is extracted during the hashing process.

Before it is passed to the predicate function, the `Header` object is somewhat modified from the raw header in the tarball: the `path` field is normalized to remove `.` entries and replace multiple consecutive slashes with a single slash. If the entry has type `:hardlink`, the link target path is normalized the same way so that it will match the path of the target entry; the size field is set to the size of the target path (which must be an already-seen file).

Currently supported values for `algorithm` are `git-sha1` (the default) and `git-sha256`, which uses the same basic algorithm as `git-sha1` but replaces the SHA1 hash function with SHA2-256, the hash function that git will transition to using in the future (due to known attacks on SHA1). Support for other file tree hashing algorithms may be added in the future.

The `skip_empty` option controls whether directories in the tarball which recursively contain no files or symlinks are included in the hash or ignored. In general, if you are hashing the content of a tarball or a file tree, you care about all directories, not just non-empty ones, so including these in the computed hash is the default. So why does this function even provide the option to skip empty directories? Because git refuses to store empty directories and will ignore them if you try to add them to a repo. So if you compute a reference tree hash by by adding files to a git repo and then asking git for the tree hash, the hash value that you get will match the hash value computed by `tree_hash` with `skip_empty=true`. In other words, this option allows `tree_hash` to emulate how git would hash a tree with empty directories. If you are hashing trees that may contain empty directories (i.e. do not come from a git repo), however, it is recommended that you hash them using a tool (such as this one) that does not ignore empty directories.

`Tar.Header` – Type.

The `Header` type is a struct representing the essential metadata for a single record in a tar file with this definition:

```
struct Header
    path :: String # path relative to the root
    type :: Symbol # type indicator (see below)
    mode :: UInt16 # mode/permissions (best viewed in octal)
    size :: Int64  # size of record data in bytes
```

```
    link :: String # target path of a symlink
end
```

Types are represented with the following symbols: `file`, `hardlink`, `symlink`, `chardev`, `blockdev`, `directory`, `fifo`, or for unknown types, the typeflag character as a symbol. Note that `extract` refuses to extract records types other than `file`, `symlink` and `directory`; `list` will only list other kinds of records if called with `strict=false`.

The tar format includes various other metadata about records, including user and group IDs, user and group names, and timestamps. The `Tar` package, by design, completely ignores these. When creating tar files, these fields are always set to zero/empty. When reading tar files, these fields are ignored aside from verifying header checksums for each header record for all fields.

Chapter 100

单元测试

100.1 测试 Julia Base 库

Julia 处于快速开发中，有着可以扩展的测试套件，用来跨平台测试功能。如果你是通过源代码构建的 Julia ，你可以通过 `make test` 来运行这个测试套件。如果是通过二进制包安装的，你可以通过 `Base.runtests()` 来运行这个测试套件。

`Base.runtests` – Function.

```
Base.runtests(tests=["all"]; ncores=ceil(Int, Sys.CPU_THREADS / 2),
              exit_on_error=false, revise=false, [seed])
```

Run the Julia unit tests listed in `tests`, which can be either a string or an array of strings, using `ncores` processors. If `exit_on_error` is `false`, when one test fails, all remaining tests in other files will still be run; they are otherwise discarded, when `exit_on_error == true`. If `revise` is `true`, the `Revise` package is used to load any modifications to Base or to the standard libraries before running the tests. If a seed is provided via the keyword argument, it is used to seed the global RNG in the context where the tests are run; otherwise the seed is chosen randomly.

source

100.2 基本的单元测试

The `Test` module provides simple *unit testing* functionality. Unit testing is a way to see if your code is correct by checking that the results are what you expect. It can be helpful to ensure your code still works after you make changes, and can be used when developing as a way of specifying the behaviors your code should have when complete. You may also want to look at the documentation for adding tests to your Julia Package.

简单的单元测试可以通过 `@test` 和 `@test_throws` 宏来完成：

`Test.@test` – Macro.

```
@test ex
@test f(args...) key=val ...
@test ex broken=true
@test ex skip=true
```

Test that the expression ex evaluates to `true`. If executed inside a `@testset`, return a `Pass` `Result` if it does, a `Fail` `Result` if it is `false`, and an `Error` `Result` if it could not be evaluated. If executed outside a `@testset`, throw an exception instead of returning `Fail` or `Error`.

Examples

```
julia> @test true
Test Passed

julia> @test [1, 2] + [2, 1] == [3, 3]
Test Passed
```

The `@test f(args...) key=val...` form is equivalent to writing `@test f(args..., key=val...)` which can be useful when the expression is a call using infix syntax such as approximate comparisons:

```
julia> @test π ≈ 3.14 atol=0.01
Test Passed
```

This is equivalent to the uglier test `@test ≈(π, 3.14, atol=0.01)`. It is an error to supply more than one expression unless the first is a call expression and the rest are assignments (k=v).

You can use any key for the `key=val` arguments, except for `broken` and `skip`, which have special meanings in the context of `@test`:

- `broken=cond` indicates a test that should pass but currently consistently fails when `cond==true`. Tests that the expression ex evaluates to `false` or causes an exception. Returns a `Broken` `Result` if it does, or an `Error` `Result` if the expression evaluates to `true`. Regular `@test ex` is evaluated when `cond==false`.
- `skip=cond` marks a test that should not be executed but should be included in test summary reporting as `Broken`, when `cond==true`. This can be useful for tests that intermittently fail, or tests of not-yet-implemented functionality. Regular `@test ex` is evaluated when `cond==false`.

Examples

```
julia> @test 2 + 2 ≈ 6 atol=1 broken=true
Test Broken
  Expression: ≈(2 + 2, 6, atol = 1)

julia> @test 2 + 2 ≈ 5 atol=1 broken=false
Test Passed

julia> @test 2 + 2 == 5 skip=true
Test Broken
  Skipped: 2 + 2 == 5

julia> @test 2 + 2 == 4 skip=false
Test Passed
```

Julia 1.7

The `broken` and `skip` keyword arguments require at least Julia 1.7.

source

Test.@test_throws – Macro.

```
@test_throws exception expr
```

Tests that the expression expr throws exception. The exception may specify either a type, a string, regular expression, or list of strings occurring in the displayed error message, a matching function, or a value (which will be tested for equality by comparing fields). Note that @test_throws does not support a trailing keyword form.

Julia 1.8

The ability to specify anything other than a type or a value as exception requires Julia v1.8 or later.

Examples

```
julia> @test_throws BoundsError [1, 2, 3][4]
Test Passed
      Thrown: BoundsError

julia> @test_throws DimensionMismatch [1, 2, 3] + [1, 2]
Test Passed
      Thrown: DimensionMismatch

julia> @test_throws "Try sqrt(Complex" sqrt(-1)
Test Passed
     Message: "DomainError with -1.0:\nsqrt was called with a negative real argument but will
↪  only return a complex result if called with a complex argument. Try sqrt(Complex(x))."
```

In the final example, instead of matching a single string it could alternatively have been performed with:

- ["Try", "Complex"] (a list of strings)
- r"Try sqrt\([Cc]omplex" (a regular expression)
- str -> occursin("complex", str) (a matching function)

source

例如，假设我们想要测试新的函数 foo(x) 是否按照期望的方式工作：

```
julia> using Test

julia> foo(x) = length(x)^2
foo (generic function with 1 method)
```

If the condition is true, a Pass is returned:

```
julia> @test foo("bar") == 9
Test Passed
  Expression: foo("bar") == 9
   Evaluated: 9 == 9

julia> @test foo("fizz") >= 10
Test Passed
  Expression: foo("fizz") >= 10
   Evaluated: 16 >= 10
```

如果条件为假，则返回 `Fail` 并抛出异常。

```
julia> @test foo("f") == 20
Test Failed at none:1
  Expression: foo("f") == 20
   Evaluated: 1 == 20

ERROR: There was an error during testing
```

If the condition could not be evaluated because an exception was thrown, which occurs in this case because `length` is not defined for symbols, an `Error` object is returned and an exception is thrown:

```
julia> @test foo(:cat) == 1
Error During Test
  Test threw an exception of type MethodError
  Expression: foo(:cat) == 1
  MethodError: no method matching length(::Symbol)
  Closest candidates are:
    length(::SimpleVector) at essentials.jl:256
    length(::Base.MethodList) at reflection.jl:521
    length(::MethodTable) at reflection.jl:597
    ...
  Stacktrace:
  [...]
ERROR: There was an error during testing
```

If we expect that evaluating an expression *should* throw an exception, then we can use `@test_throws` to check that this occurs:

```
julia> @test_throws MethodError foo(:cat)
Test Passed
  Expression: foo(:cat)
      Thrown: MethodError
```

100.3 Working with Test Sets

Typically a large number of tests are used to make sure functions work correctly over a range of inputs. In the event a test fails, the default behavior is to throw an exception immediately. However, it is normally preferable to run the rest of the tests first to get a better picture of how many errors there are in the code being tested.

Note

The `@testset` will create a local scope of its own when running the tests in it.

The `@testset` macro can be used to group tests into *sets*. All the tests in a test set will be run, and at the end of the test set a summary will be printed. If any of the tests failed, or could not be evaluated due to an error, the test set will then throw a `TestSetException`.

`Test.@testset` – Macro.

```
@testset [CustomTestSet] [options...] ["description"] begin test_ex end
@testset [CustomTestSet] [options...] ["description $v"] for v in itr test_ex end
@testset [CustomTestSet] [options...] ["description $v, $w"] for v in itrv, w in itrw test_ex
↪  end
@testset [CustomTestSet] [options...] ["description"] test_func()
@testset let v = v, w = w; test_ex; end
```

With begin/end or function call

When @testset is used, with begin/end or a single function call, the macro starts a new test set in which to evaluate the given expression.

If no custom testset type is given it defaults to creating a `DefaultTestSet`. `DefaultTestSet` records all the results and, if there are any `Fail`s or `Error`s, throws an exception at the end of the top-level (non-nested) test set, along with a summary of the test results.

Any custom testset type (subtype of `AbstractTestSet`) can be given and it will also be used for any nested `@testset` invocations. The given options are only applied to the test set where they are given. The default test set type accepts three boolean options:

- `verbose`: if `true`, the result summary of the nested testsets is shown even when they all pass (the default is `false`).
- `showtiming`: if `true`, the duration of each displayed testset is shown (the default is `true`).
- `failfast`: if `true`, any test failure or error will cause the testset and any child testsets to return immediately (the default is `false`). This can also be set globally via the env var `JULIA_TEST_FAILFAST`.

Julia 1.8

`@testset test_func()` requires at least Julia 1.8.

Julia 1.9

`failfast` requires at least Julia 1.9.

The description string accepts interpolation from the loop indices. If no description is provided, one is constructed based on the variables. If a function call is provided, its name will be used. Explicit description strings override this behavior.

By default the `@testset` macro will return the testset object itself, though this behavior can be customized in other testset types. If a `for` loop is used then the macro collects and returns a list of the return values of the `finish` method, which by default will return a list of the testset objects used in each iteration.

Before the execution of the body of a `@testset`, there is an implicit call to `Random.seed!(seed)` where `seed` is the current seed of the global RNG. Moreover, after the execution of the body, the state of the global RNG is restored to what it was before the `@testset`. This is meant to ease reproducibility in case of failure, and to allow seamless re-arrangements of `@testset`s regardless of their side-effect on the global RNG state.

Examples

```
julia> @testset "trigonometric identities" begin
           θ = 2/3*π
           @test sin(-θ) ≈ -sin(θ)
           @test cos(-θ) ≈ cos(θ)
           @test sin(2θ) ≈ 2*sin(θ)*cos(θ)
           @test cos(2θ) ≈ cos(θ)^2 - sin(θ)^2
       end;
Test Summary:            | Pass  Total  Time
trigonometric identities |    4      4  0.2s
```

`@testset for`

When `@testset for` is used, the macro starts a new test for each iteration of the provided loop. The semantics of each test set are otherwise identical to that of that `begin/end` case (as if used for each loop iteration).

`@testset let`

When `@testset let` is used, the macro starts a *transparent* test set with the given object added as a context object to any failing test contained therein. This is useful when performing a set of related tests on one larger object and it is desirable to print this larger object when any of the individual tests fail. Transparent test sets do not introduce additional levels of nesting in the test set hierarchy and are passed through directly to the parent test set (with the context object appended to any failing tests.)

> **Julia 1.9**
>
> `@testset let` requires at least Julia 1.9.

> **Julia 1.10**
>
> Multiple `let` assignements are supported since Julia 1.10.

Examples

```
julia> @testset let logi = log(im)
           @test imag(logi) == π/2
           @test !iszero(real(logi))
       end
Test Failed at none:3
  Expression: !(iszero(real(logi)))
     Context: logi = 0.0 + 1.5707963267948966im

ERROR: There was an error during testing

julia> @testset let logi = log(im), op = !iszero
           @test imag(logi) == π/2
```

```
            @test op(real(logi))
        end
Test Failed at none:3
  Expression: op(real(logi))
     Context: logi = 0.0 + 1.5707963267948966im
                op = !iszero

ERROR: There was an error during testing
```

source

Test.TestSetException – Type.

```
TestSetException
```

Thrown when a test set finishes and not all tests passed.

source

We can put our tests for the `foo(x)` function in a test set:

```
julia> @testset "Foo Tests" begin
           @test foo("a")   == 1
           @test foo("ab")  == 4
           @test foo("abc") == 9
       end;
Test Summary: | Pass  Total  Time
Foo Tests     |    3      3  0.0s
```

测试集可以嵌套：

```
julia> @testset "Foo Tests" begin
           @testset "Animals" begin
               @test foo("cat") == 9
               @test foo("dog") == foo("cat")
           end
           @testset "Arrays $i" for i in 1:3
               @test foo(zeros(i)) == i^2
               @test foo(fill(1.0, i)) == i^2
           end
       end;
Test Summary: | Pass  Total  Time
Foo Tests     |    8      8  0.0s
```

As well as call functions:

```
julia> f(x) = @test isone(x)
f (generic function with 1 method)

julia> @testset f(1);
Test Summary: | Pass  Total  Time
f             |    1      1  0.0s
```

This can be used to allow for factorization of test sets, making it easier to run individual test sets by running the associated functions instead. Note that in the case of functions, the test set will be given the name of the called function. In the event that a nested test set has no failures, as happened here, it will be hidden in the summary, unless the `verbose=true` option is passed:

```
julia> @testset verbose = true "Foo Tests" begin
           @testset "Animals" begin
               @test foo("cat") == 9
               @test foo("dog") == foo("cat")
           end
           @testset "Arrays $i" for i in 1:3
               @test foo(zeros(i)) == i^2
               @test foo(fill(1.0, i)) == i^2
           end
       end;
Test Summary: | Pass  Total  Time
Foo Tests     |    8      8  0.0s
  Animals     |    2      2  0.0s
  Arrays 1    |    2      2  0.0s
  Arrays 2    |    2      2  0.0s
  Arrays 3    |    2      2  0.0s
```

If we do have a test failure, only the details for the failed test sets will be shown:

```
julia> @testset "Foo Tests" begin
           @testset "Animals" begin
               @testset "Felines" begin
                   @test foo("cat") == 9
               end
               @testset "Canines" begin
                   @test foo("dog") == 9
               end
           end
           @testset "Arrays" begin
               @test foo(zeros(2)) == 4
               @test foo(fill(1.0, 4)) == 15
           end
       end

Arrays: Test Failed
  Expression: foo(fill(1.0, 4)) == 15
   Evaluated: 16 == 15
[...]
Test Summary: | Pass  Fail  Total  Time
Foo Tests     |    3     1      4  0.0s
  Animals     |    2            2  0.0s
  Arrays      |    1     1      2  0.0s
ERROR: Some tests did not pass: 3 passed, 1 failed, 0 errored, 0 broken.
```

100.4 Testing Log Statements

One can use the `@test_logs` macro to test log statements, or use a `TestLogger`.

`Test.@test_logs` – Macro.

```
@test_logs [log_patterns...] [keywords] expression
```

Collect a list of log records generated by `expression` using `collect_test_logs`, check that they match the sequence `log_patterns`, and return the value of `expression`. The `keywords` provide some simple filtering of log records: the `min_level` keyword controls the minimum log level which will be collected for the test, the `match_mode` keyword defines how matching will be performed (the default `:all` checks that all logs and patterns match pairwise; use `:any` to check that the pattern matches at least once somewhere in the sequence.)

The most useful log pattern is a simple tuple of the form `(level,message)`. A different number of tuple elements may be used to match other log metadata, corresponding to the arguments to passed to `AbstractLogger` via the `handle_message` function: `(level,message,module,group,id,file,line)`. Elements which are present will be matched pairwise with the log record fields using `==` by default, with the special cases that `Symbol`s may be used for the standard log levels, and `Regex`s in the pattern will match string or Symbol fields using `occursin`.

Examples

Consider a function which logs a warning, and several debug messages:

```
function foo(n)
    @info "Doing foo with n=$n"
    for i=1:n
        @debug "Iteration $i"
    end
    42
end
```

We can test the info message using

```
@test_logs (:info,"Doing foo with n=2") foo(2)
```

If we also wanted to test the debug messages, these need to be enabled with the `min_level` keyword:

```
using Logging
@test_logs (:info,"Doing foo with n=2") (:debug,"Iteration 1") (:debug,"Iteration 2")
↪  min_level=Logging.Debug foo(2)
```

If you want to test that some particular messages are generated while ignoring the rest, you can set the keyword `match_mode=:any`:

```
using Logging
@test_logs (:info,) (:debug,"Iteration 42") min_level=Logging.Debug match_mode=:any foo(100)
```

The macro may be chained with `@test` to also test the returned value:

```
@test (@test_logs (:info,"Doing foo with n=2") foo(2)) == 42
```

If you want to test for the absence of warnings, you can omit specifying log patterns and set the min_level accordingly:

```
# test that the expression logs no messages when the logger level is warn:
@test_logs min_level=Logging.Warn @info("Some information") # passes
@test_logs min_level=Logging.Warn @warn("Some information") # fails
```

If you want to test the absence of warnings (or error messages) in stderr which are not generated by @warn, see @test_nowarn.

source

Test.TestLogger – Type.

```
TestLogger(; min_level=Info, catch_exceptions=false)
```

Create a TestLogger which captures logged messages in its logs::Vector{LogRecord} field.

Set min_level to control the LogLevel, catch_exceptions for whether or not exceptions thrown as part of log event generation should be caught, and respect_maxlog for whether or not to follow the convention of logging messages with maxlog=n for some integer n at most n times.

See also: LogRecord.

Example

```
julia> using Test, Logging

julia> f() = @info "Hi" number=5;

julia> test_logger = TestLogger();

julia> with_logger(test_logger) do
           f()
           @info "Bye!"
       end

julia> @test test_logger.logs[1].message == "Hi"
Test Passed

julia> @test test_logger.logs[1].kwargs[:number] == 5
Test Passed

julia> @test test_logger.logs[2].message == "Bye!"
Test Passed
```

source

Test.LogRecord – Type.

```
LogRecord
```

Stores the results of a single log event. Fields:

- `level`: the LogLevel of the log message
- `message`: the textual content of the log message
- `_module`: the module of the log event
- `group`: the logging group (by default, the name of the file containing the log event)
- `id`: the ID of the log event
- `file`: the file containing the log event
- `line`: the line within the file of the log event
- `kwargs`: any keyword arguments passed to the log event

source

100.5 Other Test Macros

As calculations on floating-point values can be imprecise, you can perform approximate equality checks using either `@test a ≈ b` (where ≈, typed via tab completion of `\approx`, is the `isapprox` function) or use `isapprox` directly.

```
julia> @test 1 ≈ 0.999999999
Test Passed
  Expression: 1 ≈ 0.999999999
   Evaluated: 1 ≈ 0.999999999

julia> @test 1 ≈ 0.999999
Test Failed at none:1
  Expression: 1 ≈ 0.999999
   Evaluated: 1 ≈ 0.999999

ERROR: There was an error during testing
```

You can specify relative and absolute tolerances by setting the `rtol` and `atol` keyword arguments of `isapprox`, respectively, after the ≈ comparison:

```
julia> @test 1 ≈ 0.999999  rtol=1e-5
Test Passed
```

Note that this is not a specific feature of the ≈ but rather a general feature of the `@test` macro: `@test a <op> b key=val` is transformed by the macro into `@test op(a, b, key=val)`. It is, however, particularly useful for ≈ tests.

`Test.@inferred` – Macro.

```
@inferred [AllowedType] f(x)
```

Tests that the call expression `f(x)` returns a value of the same type inferred by the compiler. It is useful to check for type stability.

`f(x)` can be any call expression. Returns the result of `f(x)` if the types match, and an `Error Result` if it finds different types.

Optionally, `AllowedType` relaxes the test, by making it pass when either the type of `f(x)` matches the inferred type modulo `AllowedType`, or when the return type is a subtype of `AllowedType`. This is useful when testing type stability of functions returning a small union such as `Union{Nothing, T}` or `Union{Missing, T}`.

```
julia> f(a) = a > 1 ? 1 : 1.0
f (generic function with 1 method)

julia> typeof(f(2))
Int64

julia> @code_warntype f(2)
MethodInstance for f(::Int64)
  from f(a) @ Main none:1
Arguments
  #self#::Core.Const(f)
  a::Int64
Body::UNION{FLOAT64, INT64}
1 ─ %1 = (a > 1)::Bool
└──      goto #3 if not %1
2 ─      return 1
3 ─      return 1.0

julia> @inferred f(2)
ERROR: return type Int64 does not match inferred return type Union{Float64, Int64}
[...]

julia> @inferred max(1, 2)
2

julia> g(a) = a < 10 ? missing : 1.0
g (generic function with 1 method)

julia> @inferred g(20)
ERROR: return type Float64 does not match inferred return type Union{Missing, Float64}
[...]

julia> @inferred Missing g(20)
1.0

julia> h(a) = a < 10 ? missing : f(a)
h (generic function with 1 method)

julia> @inferred Missing h(20)
ERROR: return type Int64 does not match inferred return type Union{Missing, Float64, Int64}
[...]
```

source

`Test.@test_deprecated` – Macro.

```
@test_deprecated [pattern] expression
```

When `--depwarn=yes`, test that `expression` emits a deprecation warning and return the value of `expression`. The log message string will be matched against `pattern` which defaults to `r"deprecated"i`.

When `--depwarn=no`, simply return the result of executing `expression`. When `--depwarn=error`, check that an ErrorException is thrown.

Examples

```
# Deprecated in julia 0.7
@test_deprecated num2hex(1)

# The returned value can be tested by chaining with @test:
@test (@test_deprecated num2hex(1)) == "0000000000000001"
```

source

`Test.@test_warn` – Macro.

```
@test_warn msg expr
```

Test whether evaluating `expr` results in `stderr` output that contains the `msg` string or matches the `msg` regular expression. If `msg` is a boolean function, tests whether `msg(output)` returns `true`. If `msg` is a tuple or array, checks that the error output contains/matches each item in `msg`. Returns the result of evaluating `expr`.

See also `@test_nowarn` to check for the absence of error output.

Note: Warnings generated by `@warn` cannot be tested with this macro. Use `@test_logs` instead.

source

`Test.@test_nowarn` – Macro.

```
@test_nowarn expr
```

Test whether evaluating `expr` results in empty `stderr` output (no warnings or other messages). Returns the result of evaluating `expr`.

Note: The absence of warnings generated by `@warn` cannot be tested with this macro. Use `@test_logs` instead.

source

100.6 Broken Tests

If a test fails consistently it can be changed to use the `@test_broken` macro. This will denote the test as `Broken` if the test continues to fail and alerts the user via an `Error` if the test succeeds.

`Test.@test_broken` – Macro.

```
@test_broken ex
@test_broken f(args...) key=val ...
```

Indicates a test that should pass but currently consistently fails. Tests that the expression ex evaluates to `false` or causes an exception. Returns a `Broken` `Result` if it does, or an `Error` `Result` if the expression evaluates to `true`. This is equivalent to `@test ex broken=true`.

The `@test_broken f(args...) key=val...` form works as for the `@test` macro.

Examples

```
julia> @test_broken 1 == 2
Test Broken
  Expression: 1 == 2

julia> @test_broken 1 == 2 atol=0.1
Test Broken
  Expression: ==(1, 2, atol = 0.1)
```

source

`@test_skip` is also available to skip a test without evaluation, but counting the skipped test in the test set reporting. The test will not run but gives a `Broken` `Result`.

`Test.@test_skip` – Macro.

```
@test_skip ex
@test_skip f(args...) key=val ...
```

Marks a test that should not be executed but should be included in test summary reporting as `Broken`. This can be useful for tests that intermittently fail, or tests of not-yet-implemented functionality. This is equivalent to `@test ex skip=true`.

The `@test_skip f(args...) key=val...` form works as for the `@test` macro.

Examples

```
julia> @test_skip 1 == 2
Test Broken
  Skipped: 1 == 2

julia> @test_skip 1 == 2 atol=0.1
Test Broken
  Skipped: ==(1, 2, atol = 0.1)
```

source

100.7 Test result types

`Test.Result` – Type.

```
Test.Result
```

All tests produce a result object. This object may or may not be stored, depending on whether the test is part of a test set.

source

`Test.Pass` – Type.

```
Test.Pass <: Test.Result
```

The test condition was true, i.e. the expression evaluated to true or the correct exception was thrown.

source

`Test.Fail` – Type.

```
Test.Fail <: Test.Result
```

The test condition was false, i.e. the expression evaluated to false or the correct exception was not thrown.

source

`Test.Error` – Type.

```
Test.Error <: Test.Result
```

The test condition couldn't be evaluated due to an exception, or it evaluated to something other than a `Bool`. In the case of `@test_broken` it is used to indicate that an unexpected `Pass` `Result` occurred.

source

`Test.Broken` – Type.

```
Test.Broken <: Test.Result
```

The test condition is the expected (failed) result of a broken test, or was explicitly skipped with `@test_skip`.

source

100.8 Creating Custom `AbstractTestSet` Types

Packages can create their own `AbstractTestSet` subtypes by implementing the `record` and `finish` methods. The subtype should have a one-argument constructor taking a description string, with any options passed in as keyword arguments.

`Test.record` – Function.

```
record(ts::AbstractTestSet, res::Result)
```

Record a result to a testset. This function is called by the `@testset` infrastructure each time a contained `@test` macro completes, and is given the test result (which could be an `Error`). This will also be called with an `Error` if an exception is thrown inside the test block but outside of a `@test` context.

source

`Test.finish` – Function.

```
finish(ts::AbstractTestSet)
```

Do any final processing necessary for the given testset. This is called by the `@testset` infrastructure after a test block executes.

Custom `AbstractTestSet` subtypes should call `record` on their parent (if there is one) to add themselves to the tree of test results. This might be implemented as:

```
if get_testset_depth() != 0
    # Attach this test set to the parent test set
    parent_ts = get_testset()
    record(parent_ts, self)
    return self
end
```

source

`Test` takes responsibility for maintaining a stack of nested testsets as they are executed, but any result accumulation is the responsibility of the `AbstractTestSet` subtype. You can access this stack with the `get_testset` and `get_testset_depth` methods. Note that these functions are not exported.

`Test.get_testset` – Function.

```
get_testset()
```

Retrieve the active test set from the task's local storage. If no test set is active, use the fallback default test set.

source

`Test.get_testset_depth` – Function.

```
get_testset_depth()
```

Return the number of active test sets, not including the default test set

source

`Test` also makes sure that nested `@testset` invocations use the same `AbstractTestSet` subtype as their parent unless it is set explicitly. It does not propagate any properties of the testset. Option inheritance behavior can be implemented by packages using the stack infrastructure that `Test` provides.

Defining a basic `AbstractTestSet` subtype might look like:

```
import Test: Test, record, finish
using Test: AbstractTestSet, Result, Pass, Fail, Error
using Test: get_testset_depth, get_testset
struct CustomTestSet <: Test.AbstractTestSet
    description::AbstractString
    foo::Int
    results::Vector
    # constructor takes a description string and options keyword arguments
    CustomTestSet(desc; foo=1) = new(desc, foo, [])
end

record(ts::CustomTestSet, child::AbstractTestSet) = push!(ts.results, child)
record(ts::CustomTestSet, res::Result) = push!(ts.results, res)
function finish(ts::CustomTestSet)
    # just record if we're not the top-level parent
    if get_testset_depth() > 0
        record(get_testset(), ts)
    end
    ts
end
```

And using that testset looks like:

```
@testset CustomTestSet foo=4 "custom testset inner 2" begin
    # this testset should inherit the type, but not the argument.
    @testset "custom testset inner" begin
        @test true
    end
end
```

100.9 Test utilities

`Test.GenericArray` – Type.

The `GenericArray` can be used to test generic array APIs that program to the `AbstractArray` interface, in order to ensure that functions can work with array types besides the standard `Array` type.

source

`Test.GenericDict` – Type.

The `GenericDict` can be used to test generic dict APIs that program to the `AbstractDict` interface, in order to ensure that functions can work with associative types besides the standard `Dict` type.

source

`Test.GenericOrder` – Type.

The `GenericOrder` can be used to test APIs for their support of generic ordered types.

source

`Test.GenericSet` – Type.

The `GenericSet` can be used to test generic set APIs that program to the `AbstractSet` interface, in order to ensure that functions can work with set types besides the standard `Set` and `BitSet` types.

source

`Test.GenericString` – Type.

The `GenericString` can be used to test generic string APIs that program to the `AbstractString` interface, in order to ensure that functions can work with string types besides the standard `String` type.

source

`Test.detect_ambiguities` – Function.

```
detect_ambiguities(mod1, mod2...; recursive=false,
                                  ambiguous_bottom=false,
                                  allowed_undefineds=nothing)
```

Return a vector of `(Method,Method)` pairs of ambiguous methods defined in the specified modules. Use `recursive=true` to test in all submodules.

`ambiguous_bottom` controls whether ambiguities triggered only by `Union{}` type parameters are included; in most cases you probably want to set this to `false`. See `Base.isambiguous`.

See `Test.detect_unbound_args` for an explanation of `allowed_undefineds`.

> **Julia 1.8**
>
> `allowed_undefineds` requires at least Julia 1.8.

source

`Test.detect_unbound_args` – Function.

```
detect_unbound_args(mod1, mod2...; recursive=false, allowed_undefineds=nothing)
```

Return a vector of `Method`s which may have unbound type parameters. Use `recursive=true` to test in all submodules.

By default, any undefined symbols trigger a warning. This warning can be suppressed by supplying a collection of `GlobalRef`s for which the warning can be skipped. For example, setting

```
allowed_undefineds = Set([GlobalRef(Base, :active_repl),
                          GlobalRef(Base, :active_repl_backend)])
```

would suppress warnings about `Base.active_repl` and `Base.active_repl_backend`.

> **Julia 1.8**
>
> `allowed_undefineds` requires at least Julia 1.8.

source

100.10 Workflow for Testing Packages

Using the tools available to us in the previous sections, here is a potential workflow of creating a package and adding tests to it.

Generating an Example Package

For this workflow, we will create a package called `Example`:

```
pkg> generate Example
shell> cd Example
shell> mkdir test
pkg> activate .
```

Creating Sample Functions

The number one requirement for testing a package is to have functionality to test. For that, we will add some simple functions to `Example` that we can test. Add the following to `src/Example.jl`:

```
module Example

function greet()
    "Hello world!"
end

function simple_add(a, b)
    a + b
end

function type_multiply(a::Float64, b::Float64)
    a * b
end

end
```

Creating a Test Environment

From within the root of the `Example` package, navigate to the `test` directory, activate a new environment there, and add the `Test` package to the environment:

```
shell> cd test
pkg> activate .
(test) pkg> add Test
```

Testing Our Package

Now, we are ready to add tests to `Example`. It is standard practice to create a file within the `test` directory called `runtests.jl` which contains the test sets we want to run. Go ahead and create that file within the `test` directory and add the following code to it:

```
using Example
using Test

@testset "Example tests" begin

    @testset "Math tests" begin
        include("math_tests.jl")
    end

    @testset "Greeting tests" begin
        include("greeting_tests.jl")
    end
end
```

We will need to create those two included files, `math_tests.jl` and `greeting_tests.jl`, and add some tests to them.

> **Note:** Notice how we did not have to specify add `Example` into the `test` environment's `Project.toml`. This is a benefit of Julia's testing system that you could read about more here.

Writing Tests for `math_tests.jl`

Using our knowledge of `Test.jl`, here are some example tests we could add to `math_tests.jl`:

```
@testset "Testset 1" begin
    @test 2 == simple_add(1, 1)
    @test 3.5 == simple_add(1, 2.5)
        @test_throws MethodError simple_add(1, "A")
        @test_throws MethodError simple_add(1, 2, 3)
end

@testset "Testset 2" begin
    @test 1.0 == type_multiply(1.0, 1.0)
        @test isa(type_multiply(2.0, 2.0), Float64)
    @test_throws MethodError type_multiply(1, 2.5)
end
```

Writing Tests for `greeting_tests.jl`

Using our knowledge of `Test.jl`, here are some example tests we could add to `math_tests.jl`:

```
@testset "Testset 3" begin
    @test "Hello world!" == greet()
    @test_throws MethodError greet("Antonia")
end
```

Testing Our Package

Now that we have added our tests and our `runtests.jl` script in `test`, we can test our `Example` package by going back to the root of the `Example` package environment and reactivating the `Example` environment:

```
shell> cd ..
pkg> activate .
```

From there, we can finally run our test suite as follows:

```
(Example) pkg> test
     Testing Example
      Status `/tmp/jl_Yngpvy/Project.toml`
  [fa318bd2] Example v0.1.0 `/home/src/Projects/tmp/errata/Example`
  [8dfed614] Test `@stdlib/Test`
      Status `/tmp/jl_Yngpvy/Manifest.toml`
  [fa318bd2] Example v0.1.0 `/home/src/Projects/tmp/errata/Example`
  [2a0f44e3] Base64 `@stdlib/Base64`
  [b77e0a4c] InteractiveUtils `@stdlib/InteractiveUtils`
  [56ddb016] Logging `@stdlib/Logging`
  [d6f4376e] Markdown `@stdlib/Markdown`
  [9a3f8284] Random `@stdlib/Random`
  [ea8e919c] SHA `@stdlib/SHA`
  [9e88b42a] Serialization `@stdlib/Serialization`
  [8dfed614] Test `@stdlib/Test`
     Testing Running tests...
Test Summary: | Pass  Total
Example tests |    9      9
     Testing Example tests passed
```

And if all went correctly, you should see a similar output as above. Using `Test.jl`, more complicated tests can be added for packages but this should ideally point developers in the direction of how to get started with testing their own created packages.

Chapter 101

UUIDs

UUIDs.uuid1 – Function.

```
uuid1([rng::AbstractRNG]) -> UUID
```

Generates a version 1 (time-based) universally unique identifier (UUID), as specified by RFC 4122. Note that the Node ID is randomly generated (does not identify the host) according to section 4.5 of the RFC.

The default rng used by `uuid1` is not `GLOBAL_RNG` and every invocation of `uuid1()` without an argument should be expected to return a unique identifier. Importantly, the outputs of `uuid1` do not repeat even when `Random.seed!(seed)` is called. Currently (as of Julia 1.6), `uuid1` uses `Random.RandomDevice` as the default rng. However, this is an implementation detail that may change in the future.

Julia 1.6

The output of `uuid1` does not depend on `GLOBAL_RNG` as of Julia 1.6.

Examples

```
julia> rng = MersenneTwister(1234);

julia> uuid1(rng)
UUID("cfc395e8-590f-11e8-1f13-43a2532b2fa8")
```

UUIDs.uuid4 – Function.

```
uuid4([rng::AbstractRNG]) -> UUID
```

Generates a version 4 (random or pseudo-random) universally unique identifier (UUID), as specified by RFC 4122.

The default rng used by `uuid4` is not `GLOBAL_RNG` and every invocation of `uuid4()` without an argument should be expected to return a unique identifier. Importantly, the outputs of `uuid4` do not repeat even when `Random.seed!(seed)` is called. Currently (as of Julia 1.6), `uuid4` uses `Random.RandomDevice` as the default rng. However, this is an implementation detail that may change in the future.

Julia 1.6

The output of `uuid4` does not depend on `GLOBAL_RNG` as of Julia 1.6.

Examples

```
julia> rng = MersenneTwister(1234);

julia> uuid4(rng)
UUID("7a052949-c101-4ca3-9a7e-43a2532b2fa8")
```

`UUIDs.uuid5` – Function.

```
uuid5(ns::UUID, name::String) -> UUID
```

Generates a version 5 (namespace and domain-based) universally unique identifier (UUID), as specified by RFC 4122.

Julia 1.1

This function requires at least Julia 1.1.

Examples

```
julia> rng = MersenneTwister(1234);

julia> u4 = uuid4(rng)
UUID("7a052949-c101-4ca3-9a7e-43a2532b2fa8")

julia> u5 = uuid5(u4, "julia")
UUID("086cc5bb-2461-57d8-8068-0aed7f5b5cd1")
```

`UUIDs.uuid_version` – Function.

```
uuid_version(u::UUID) -> Int
```

Inspects the given UUID and returns its version (see RFC 4122).

Examples

```
julia> uuid_version(uuid4())
4
```

Chapter 102

Unicode

`Unicode.julia_chartransform` – Function.

```
Unicode.julia_chartransform(c::Union{Char,Integer})
```

Map the Unicode character (`Char`) or codepoint (`Integer`) `c` to the corresponding "equivalent" character or codepoint, respectively, according to the custom equivalence used within the Julia parser (in addition to NFC normalization).

For example, `'µ'` (U+00B5 micro) is treated as equivalent to `'μ'` (U+03BC mu) by Julia's parser, so `julia_chartransform` performs this transformation while leaving other characters unchanged:

```
julia> Unicode.julia_chartransform('µ')
'μ': Unicode U+03BC (category Ll: Letter, lowercase)

julia> Unicode.julia_chartransform('x')
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

`julia_chartransform` is mainly useful for passing to the `Unicode.normalize` function in order to mimic the normalization used by the Julia parser:

```
julia> s = "µö"
"µö"

julia> s2 = Unicode.normalize(s, compose=true, stable=true,
↪  chartransform=Unicode.julia_chartransform)
"μö"

julia> collect(s2)
2-element Vector{Char}:
 'μ': Unicode U+03BC (category Ll: Letter, lowercase)
 'ö': Unicode U+00F6 (category Ll: Letter, lowercase)

julia> s2 == string(Meta.parse(s))
true
```

> **Julia 1.8**
>
> This function was introduced in Julia 1.8.

`Unicode.isassigned` – Function.

```
Unicode.isassigned(c) -> Bool
```

Return `true` if the given char or integer is an assigned Unicode code point.

Examples

```
julia> Unicode.isassigned(101)
true

julia> Unicode.isassigned('\x01')
true
```

`Unicode.isequal_normalized` – Function.

```
isequal_normalized(s1::AbstractString, s2::AbstractString; casefold=false, stripmark=false,
↪  chartransform=identity)
```

Return whether `s1` and `s2` are canonically equivalent Unicode strings. If `casefold=true`, ignores case (performs Unicode case-folding); if `stripmark=true`, strips diacritical marks and other combining characters.

As with `Unicode.normalize`, you can also pass an arbitrary function via the `chartransform` keyword (mapping `Integer` codepoints to codepoints) to perform custom normalizations, such as `Unicode.julia_chartransform`.

> **Julia 1.8**
>
> The `isequal_normalized` function was added in Julia 1.8.

Examples

For example, the string `"noël"` can be constructed in two canonically equivalent ways in Unicode, depending on whether `"ë"` is formed from a single codepoint U+00EB or from the ASCII character `'e'` followed by the U+0308 combining-diaeresis character.

```
julia> s1 = "noël"
"noël"

julia> s2 = "noël"
"noël"

julia> s1 == s2
false

julia> isequal_normalized(s1, s2)
```

```
true

julia> isequal_normalized(s1, "noel", stripmark=true)
true

julia> isequal_normalized(s1, "NOËL", casefold=true)
true
```

`Unicode.normalize` – Function.

```
Unicode.normalize(s::AbstractString; keywords...)
Unicode.normalize(s::AbstractString, normalform::Symbol)
```

Normalize the string `s`. By default, canonical composition (`compose=true`) is performed without ensuring Unicode versioning stability (`compat=false`), which produces the shortest possible equivalent string but may introduce composition characters not present in earlier Unicode versions.

Alternatively, one of the four "normal forms" of the Unicode standard can be specified: `normalform` can be `:NFC`, `:NFD`, `:NFKC`, or `:NFKD`. Normal forms C (canonical composition) and D (canonical decomposition) convert different visually identical representations of the same abstract string into a single canonical form, with form C being more compact. Normal forms KC and KD additionally canonicalize "compatibility equivalents": they convert characters that are abstractly similar but visually distinct into a single canonical choice (e.g. they expand ligatures into the individual characters), with form KC being more compact.

Alternatively, finer control and additional transformations may be obtained by calling `Unicode.normalize(s; keywords...)`, where any number of the following boolean keywords options (which all default to `false` except for `compose`) are specified:

- `compose=false`: do not perform canonical composition
- `decompose=true`: do canonical decomposition instead of canonical composition (`compose=true` is ignored if present)
- `compat=true`: compatibility equivalents are canonicalized
- `casefold=true`: perform Unicode case folding, e.g. for case-insensitive string comparison
- `newline2lf=true`, `newline2ls=true`, or `newline2ps=true`: convert various newline sequences (LF, CRLF, CR, NEL) into a linefeed (LF), line-separation (LS), or paragraph-separation (PS) character, respectively
- `stripmark=true`: strip diacritical marks (e.g. accents)
- `stripignore=true`: strip Unicode's "default ignorable" characters (e.g. the soft hyphen or the left-to-right marker)
- `stripcc=true`: strip control characters; horizontal tabs and form feeds are converted to spaces; newlines are also converted to spaces unless a newline-conversion flag was specified
- `rejectna=true`: throw an error if unassigned code points are found
- `stable=true`: enforce Unicode versioning stability (never introduce characters missing from earlier Unicode versions)

You can also use the `chartransform` keyword (which defaults to `identity`) to pass an arbitrary *function* mapping `Integer` codepoints to codepoints, which is is called on each character in `s` as it is processed, in order to perform arbitrary additional normalizations. For example, by passing `chartransform=Unicode.julia_chartransform`,

you can apply a few Julia-specific character normalizations that are performed by Julia when parsing identifiers (in addition to NFC normalization: `compose=true, stable=true`).

For example, NFKC corresponds to the options `compose=true, compat=true, stable=true`.

Examples

```
julia> "é" == Unicode.normalize("é") #LHS: Unicode U+00e9, RHS: U+0065 & U+0301
true

julia> "μ" == Unicode.normalize("µ", compat=true) #LHS: Unicode U+03bc, RHS: Unicode U+00b5
true

julia> Unicode.normalize("JuLiA", casefold=true)
"julia"

julia> Unicode.normalize("JúLiA", stripmark=true)
"JuLiA"
```

Julia 1.8

The `chartransform` keyword argument requires Julia 1.8.

`Unicode.graphemes` – Function.

```
graphemes(s::AbstractString) -> GraphemeIterator
```

Return an iterator over substrings of `s` that correspond to the extended graphemes in the string, as defined by Unicode UAX #29. (Roughly, these are what users would perceive as single characters, even though they may contain more than one codepoint; for example a letter combined with an accent mark is a single grapheme.)

```
graphemes(s::AbstractString, m:n) -> SubString
```

Returns a `SubString` of `s` consisting of the `m`-th through `n`-th graphemes of the string `s`, where the second argument `m:n` is an integer-valued `AbstractUnitRange`.

Loosely speaking, this corresponds to the `m:n`-th user-perceived "characters" in the string. For example:

```
julia> s = graphemes("exposé", 3:6)
"posé"

julia> collect(s)
5-element Vector{Char}:
 'p': ASCII/Unicode U+0070 (category Ll: Letter, lowercase)
 'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
 's': ASCII/Unicode U+0073 (category Ll: Letter, lowercase)
 'e': ASCII/Unicode U+0065 (category Ll: Letter, lowercase)
 '́': Unicode U+0301 (category Mn: Mark, nonspacing)
```

This consists of the 3rd to *7th* codepoints (`Char`s) in `"exposé"`, because the grapheme `"é"` is actually *two* Unicode codepoints (an `'e'` followed by an acute-accent combining character U+0301).

Because finding grapheme boundaries requires iteration over the string contents, the `graphemes(s, m:n)` function requires time proportional to the length of the string (number of codepoints) before the end of the substring.

Julia 1.9

The `m:n` argument of `graphemes` requires Julia 1.9.

Part V

开发者文档

Chapter 103

Documentation of Julia's Internals

103.1 Julia 运行时的初始化

How does the Julia runtime execute `julia -e 'println("Hello World!")'` ?

main()

Execution starts at `main()` in `cli/loader_exe.c`, which calls `jl_load_repl()` in `cli/loader_lib.c` which loads a few libraries, eventually calling `jl_repl_entrypoint()` in `src/jlapi.c`.

`jl_repl_entrypoint()` calls `libsupport_init()` to set the C library locale and to initialize the "ios" library (see `ios_init_stdstreams()` and Legacy `ios.c` library).

Next `jl_parse_opts()` is called to process command line options. Note that `jl_parse_opts()` only deals with options that affect code generation or early initialization. Other options are handled later by `exec_options()` in `base/client.jl`.

`jl_parse_opts()` stores command line options in the global `jl_options` struct.

julia_init()

`julia_init()` in `init.c` is called by `main()` and calls `_julia_init()` in `init.c`.

`_julia_init()` begins by calling `libsupport_init()` again (it does nothing the second time).

`restore_signals()` is called to zero the signal handler mask.

`jl_resolve_sysimg_location()` searches configured paths for the base system image. See Building the Julia system image.

`jl_gc_init()` sets up allocation pools and lists for weak refs, preserved values and finalization.

`jl_init_frontend()` loads and initializes a pre-compiled femtolisp image containing the scanner/parser.

`jl_init_types()` creates `jl_datatype_t` type description objects for the built-in types defined in `julia.h`. e.g.

```
jl_any_type = jl_new_abstracttype(jl_symbol("Any"), core, NULL, jl_emptysvec);
jl_any_type->super = jl_any_type;

jl_type_type = jl_new_abstracttype(jl_symbol("Type"), core, jl_any_type, jl_emptysvec);

jl_int32_type = jl_new_primitivetype(jl_symbol("Int32"), core,
                                     jl_any_type, jl_emptysvec, 32);
```

`jl_init_tasks()` creates the `jl_datatype_t* jl_task_type` object; initializes the global `jl_root_task` struct; and sets `jl_current_task` to the root task.

`jl_init_codegen()` initializes the LLVM library.

`jl_init_serializer()` initializes 8-bit serialization tags for builtin `jl_value_t` values.

If there is no sysimg file (`!jl_options.image_file`) then the `Core` and `Main` modules are created and `boot.jl` is evaluated:

`jl_core_module = jl_new_module(jl_symbol("Core"))` creates the Julia `Core` module.

`jl_init_intrinsic_functions()` creates a new Julia module `Intrinsics` containing constant `jl_intrinsic_type` symbols. These define an integer code for each intrinsic function. `emit_intrinsic()` translates these symbols into LLVM instructions during code generation.

`jl_init_primitives()` hooks C functions up to Julia function symbols. e.g. the symbol `Core.:(===)()` is bound to C function pointer `jl_f_is()` by calling `add_builtin_func("===", jl_f_is)`.

`jl_new_main_module()` creates the global "Main" module and sets `jl_current_task->current_module = jl_main_module`.

Note: `_julia_init()` then sets `jl_root_task->current_module = jl_core_module`. `jl_root_task` is an alias of `jl_current_task` at this point, so the `current_module` set by `jl_new_main_module()` above is overwritten.

`jl_load("boot.jl", sizeof("boot.jl"))` calls `jl_parse_eval_all` which repeatedly calls `jl_toplevel_eval_flex()` to execute `boot.jl`. <!–TODO –drill down into eval? –>

`jl_get_builtin_hooks()` initializes global C pointers to Julia globals defined in `boot.jl`.

`jl_init_box_caches()` pre-allocates global boxed integer value objects for values up to 1024. This speeds up allocation of boxed ints later on. e.g.:

```
jl_value_t *jl_box_uint8(uint32_t x)
{
    return boxed_uint8_cache[(uint8_t)x];
}
```

`_julia_init()` iterates over the `jl_core_module->bindings.table` looking for `jl_datatype_t` values and sets the type name's module prefix to `jl_core_module`.

`jl_add_standard_imports(jl_main_module)` does "using Base" in the "Main" module.

Note: `_julia_init()` now reverts to `jl_root_task->current_module = jl_main_module` as it was before being set to `jl_core_module` above.

Platform specific signal handlers are initialized for `SIGSEGV` (OSX, Linux), and `SIGFPE` (Windows).

Other signals (`SIGINFO, SIGBUS, SIGILL, SIGTERM, SIGABRT, SIGQUIT, SIGSYS` and `SIGPIPE`) are hooked up to `sigdie_handler()` which prints a backtrace.

`jl_init_restored_module()` calls `jl_module_run_initializer()` for each deserialized module to run the `__init__()` function.

Finally `sigint_handler()` is hooked up to `SIGINT` and calls `jl_throw(jl_interrupt_exception)`.

`_julia_init()` then returns back to `main()` in `cli/loader_exe.c` and `main()` calls `repl_entrypoint(argc, (char**)argv)`.

sysimg

If there is a sysimg file, it contains a pre-cooked image of the `Core` and `Main` modules (and whatever else is created by `boot.jl`). See Building the Julia system image.

`jl_restore_system_image()` deserializes the saved sysimg into the current Julia runtime environment and initialization continues after `jl_init_box_caches()` below...

Note: `jl_restore_system_image()` (and `staticdata.c` in general) uses the Legacy `ios.c` library.

`repl_entrypoint()`

`repl_entrypoint()` loads the contents of `argv[]` into `Base.ARGS`.

If a `.jl` "program" file was supplied on the command line, then `exec_program()` calls `jl_load(program,len)` which calls `jl_parse_eval_all` which repeatedly calls `jl_toplevel_eval_flex()` to execute the program.

However, in our example (`julia -e 'println("Hello World!")'`), `jl_get_global(jl_base_module, jl_symbol("_start"))` looks up `Base._start` and `jl_apply()` executes it.

`Base._start`

`Base._start` calls `Base.exec_options` which calls `jl_parse_input_line("println("Hello World!")")` to create an expression object and `Core.eval(Main, ex)` to execute the parsed expression ex in the module context of `Main`.

`Core.eval`

`Core.eval(Main, ex)` calls `jl_toplevel_eval_in(m, ex)`, which calls `jl_toplevel_eval_flex`. `jl_toplevel_eval_flex` implements a simple heuristic to decide whether to compile a given code thunk or run it by interpreter. When given `println("Hello World!")`, it would usually decide to run the code by interpreter, in which case it calls `jl_interpret_toplevel_thunk`, which then calls `eval_body`.

The stack dump below shows how the interpreter works its way through various methods of `Base.println()` and `Base.print()` before arriving at `write(s::IO, a::Array{T}) where T` which does `ccall(jl_uv_write())`.

`jl_uv_write()` calls `uv_write()` to write "Hello World!" to `JL_STDOUT`. See Libuv wrappers for stdio.:

```
Hello World!
```

Since our example has just one function call, which has done its job of printing "Hello World!", the stack now rapidly unwinds back to `main()`.

`jl_atexit_hook()`

`main()` calls `jl_atexit_hook()`. This calls `Base._atexit`, then calls `jl_gc_run_all_finalizers()` and cleans up libuv handles.

`julia_save()`

Finally, `main()` calls `julia_save()`, which if requested on the command line, saves the runtime state to a new system image. See `jl_compile_all()` and `jl_save_system_image()`.

Stack frame	Source code	Notes
jl_uv_write()	jl_uv.c	called though ccall
julia_write_282942	stream.jl	function write!(s::IO, a::Array{T}) where T
julia_print_284639	ascii.jl	print(io::IO, s::String) = (write(io, s); nothing)
jlcall_print_284639		
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.print(Base.TTY, String)
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.print(Base.TTY, String, Char, Char...)
jl_apply()	julia.h	
jl_f_apply()	builtins.c	
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.println(Base.TTY, String, String...)
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_apply_generic()	gf.c	Base.println(String,)
jl_apply()	julia.h	
do_call()	interpreter.c	
eval_body()	interpreter.c	
jl_interpret_toplevel_thunk	interpreter.c	
jl_toplevel_eval_flex	toplevel.c	
jl_toplevel_eval_in	toplevel.c	
Core.eval	boot.jl	

103.2 Julia 的 AST

Julia 有两种代码的表现形式。第一种是解析器返回的表面语法 AST （例如 `Meta.parse` 函数），由宏来操控。是代码编写时的结构化表示，由 `julia-parser.scm` 用字符流构造而成。另一种则是底层形式，或者 IR（中间表示），这种形式在进行类型推导和代码生成的时候被使用。在这种底层形式中结点的类型相对更少，所有的宏都会被展开，所有的控制流会被转化成显式的分支和语句的序列。底层的形式由 `julia-syntax.scm` 构建。

首先，我们将关注 AST，因为需要它来编写宏。

表面语法 AST

前端 AST 几乎由 `Expr` 和原子（例如符号、数字）。对于视觉上不同的语法形式，通常有不同的表达式头。示例将在 s-expression 语法中给出。每个圆括号括着的列表都对应着一个 Expr，其中第一个元素是它的头部。例如 `(call f x)` 对应于 Julia 中的 `Expr(:call, :f, :x)` 。

调用

do syntax:

输入	AST
`f(x)`	`(call f x)`
`f(x, y=1, z=2)`	`(call f x (kw y 1) (kw z 2))`
`f(x; y=1)`	`(call f (parameters (kw y 1)) x)`
`f(x...)`	`(call f (... x))`

```
f(x) do a,b
    body
end
```

parses as `(do (call f x) (-> (tuple a b) (block body)))`.

运算符

Most uses of operators are just function calls, so they are parsed with the head `call`. However some operators are special forms (not necessarily function calls), and in those cases the operator itself is the expression head. In julia-parser.scm these are referred to as "syntactic operators". Some operators (+ and *) use N-ary parsing; chained calls are parsed as a single N-argument call. Finally, chains of comparisons have their own special expression structure.

Input	AST
`x+y`	`(call + x y)`
`a+b+c+d`	`(call + a b c d)`
`2x`	`(call * 2 x)`
`a&&b`	`(&& a b)`
`x += 1`	`(+= x 1)`
`a ? 1 : 2`	`(if a 1 2)`
`a,b`	`(tuple a b)`
`a==b`	`(call == a b)`
`1<i<=n`	`(comparison 1 < i <= n)`
`a.b`	`(. a (quote b))`
`a.(b)`	`(. a (tuple b))`

Bracketed forms

Macros

Strings

Doc string syntax:

```
"some docs"
f(x) = x
```

parses as `(macrocall (|.| Core '@doc) (line) "some docs" (= (call f x) (block x)))`.

Imports and such

`using` has the same representation as `import`, but with expression head `:using` instead of `:import`.

Input	AST
`a[i]`	`(ref a i)`
`t[i;j]`	`(typed_vcat t i j)`
`t[i j]`	`(typed_hcat t i j)`
`t[a b; c d]`	`(typed_vcat t (row a b) (row c d))`
`t[a b;;; c d]`	`(typed_ncat t 3 (row a b) (row c d))`
`a{b}`	`(curly a b)`
`a{b;c}`	`(curly a (parameters c) b)`
`[x]`	`(vect x)`
`[x,y]`	`(vect x y)`
`[x;y]`	`(vcat x y)`
`[x y]`	`(hcat x y)`
`[x y; z t]`	`(vcat (row x y) (row z t))`
`[x;y;; z;t;;;]`	`(ncat 3 (nrow 2 (nrow 1 x y) (nrow 1 z t)))`
`[x for y in z, a in b]`	`(comprehension (generator x (= y z) (= a b)))`
`T[x for y in z]`	`(typed_comprehension T (generator x (= y z)))`
`(a, b, c)`	`(tuple a b c)`
`(a; b; c)`	`(block a b c)`

Input	AST
`@m x y`	`(macrocall @m (line) x y)`
`Base.@m x y`	`(macrocall (. Base (quote @m)) (line) x y)`
`@Base.m x y`	`(macrocall (. Base (quote @m)) (line) x y)`

Input	AST
`"a"`	`"a"`
`x"y"`	`(macrocall @x_str (line) "y")`
`x"y"z`	`(macrocall @x_str (line) "y" "z")`
`"x = $x"`	`(string "x = " x)`
`` `a b c` ``	`(macrocall @cmd (line) "a b c")`

Numbers

Julia supports more number types than many scheme implementations, so not all numbers are represented directly as scheme numbers in the AST.

Block forms

A block of statements is parsed as `(block stmt1 stmt2 ...)`.

If statement:

```
if a
    b
elseif c
    d
else
    e
end
```

Input	AST
`import a`	`(import (. a))`
`import a.b.c`	`(import (. a b c))`
`import ...a`	`(import (. . . . a))`
`import a.b, c.d`	`(import (. a b) (. c d))`
`import Base: x`	`(import (: (. Base) (. x)))`
`import Base: x, y`	`(import (: (. Base) (. x) (. y)))`
`export a, b`	`(export a b)`

Input	AST
`11111111111111111111`	`(macrocall @int128_str nothing "11111111111111111111")`
`0xfffffffffffffffff`	`(macrocall @uint128_str nothing "0xfffffffffffffffff")`
`1111...many digits...`	`(macrocall @big_str nothing "1111....")`

parses as:

```
(if a (block (line 2) b)
    (elseif (block (line 3) c) (block (line 4) d)
            (block (line 6 e))))
```

A `while` loop parses as `(while condition body)`.

A `for` loop parses as `(for (= var iter) body)`. If there is more than one iteration specification, they are parsed as a block: `(for (block (= v1 iter1) (= v2 iter2)) body)`.

`break` and `continue` are parsed as 0-argument expressions `(break)` and `(continue)`.

`let` is parsed as `(let (= var val) body)` or `(let (block (= var1 val1) (= var2 val2) ...) body)`, like `for` loops.

A basic function definition is parsed as `(function (call f x) body)`. A more complex example:

```
function f(x::T; k = 1) where T
    return x+1
end
```

parses as:

```
(function (where (call f (parameters (kw k 1))
                       (:: x T))
                 T)
          (block (line 2) (return (call + x 1))))
```

Type definition:

```
mutable struct Foo{T<:S}
    x::T
end
```

parses as:

```
(struct true (curly Foo (<: T S))
        (block (line 2) (:: x T)))
```

The first argument is a boolean telling whether the type is mutable.

`try` blocks parse as `(try try_block var catch_block finally_block)`. If no variable is present after `catch`, `var` is `#f`. If there is no `finally` clause, then the last argument is not present.

Quote expressions

Julia source syntax forms for code quoting (`quote` and `:( )`) support interpolation with `$`. In Lisp terminology, this means they are actually "backquote" or "quasiquote" forms. Internally, there is also a need for code quoting without interpolation. In Julia's scheme code, non-interpolating quote is represented with the expression head `inert`.

`inert` expressions are converted to Julia `QuoteNode` objects. These objects wrap a single value of any type, and when evaluated simply return that value.

A `quote` expression whose argument is an atom also gets converted to a `QuoteNode`.

Line numbers

Source location information is represented as `(line line_num file_name)` where the third component is optional (and omitted when the current line number, but not file name, changes).

These expressions are represented as `LineNumberNode`s in Julia.

Macros

Macro hygiene is represented through the expression head pair `escape` and `hygienic-scope`. The result of a macro expansion is automatically wrapped in `(hygienic-scope block module)`, to represent the result of the new scope. The user can insert `(escape block)` inside to interpolate code from the caller.

Lowered form

Lowered form (IR) is more important to the compiler, since it is used for type inference, optimizations like inlining, and code generation. It is also less obvious to the human, since it results from a significant rearrangement of the input syntax.

In addition to `Symbol`s and some number types, the following data types exist in lowered form:

- `Expr`

  Has a node type indicated by the head field, and an `args` field which is a `Vector{Any}` of subexpressions. While almost every part of a surface AST is represented by an `Expr`, the IR uses only a limited number of `Expr`s, mostly for calls and some top-level-only forms.

- `SlotNumber`

  Identifies arguments and local variables by consecutive numbering. It has an integer-valued `id` field giving the slot index. The types of these slots can be found in the `slottypes` field of their `CodeInfo` object. When a slot has different types at different uses and thus requires per-use type annotations, they are converted to temporary `Core.Compiler.TypedSlot` object. This object has an additional `typ` field as well as the `id` field. Note that `Core.Compiler.TypedSlot` only appears in an unoptimized lowered form that is scheduled for optimization, and it never appears elsewhere.

- `Argument`

  The same as `SlotNumber`, but appears only post-optimization. Indicates that the referenced slot is an argument of the enclosing function.

- `CodeInfo`

  Wraps the IR of a group of statements. Its `code` field is an array of expressions to execute.

- `GotoNode`

  Unconditional branch. The argument is the branch target, represented as an index in the code array to jump to.

- `GotoIfNot`

  Conditional branch. If the `cond` field evaluates to false, goes to the index identified by the `dest` field.

- `ReturnNode`

  Returns its argument (the `val` field) as the value of the enclosing function. If the `val` field is undefined, then this represents an unreachable statement.

- `QuoteNode`

  Wraps an arbitrary value to reference as data. For example, the function `f() = :a` contains a `QuoteNode` whose `value` field is the symbol `a`, in order to return the symbol itself instead of evaluating it.

- `GlobalRef`

  Refers to global variable `name` in module `mod`.

- `SSAValue`

  Refers to a consecutively-numbered (starting at 1) static single assignment (SSA) variable inserted by the compiler. The number (`id`) of an `SSAValue` is the code array index of the expression whose value it represents.

- `NewvarNode`

  Marks a point where a variable (slot) is created. This has the effect of resetting a variable to undefined.

Expr types

These symbols appear in the `head` field of `Expr`s in lowered form.

- `call`

  Function call (dynamic dispatch). `args[1]` is the function to call, `args[2:end]` are the arguments.

- `invoke`

  Function call (static dispatch). `args[1]` is the MethodInstance to call, `args[2:end]` are the arguments (including the function that is being called, at `args[2]`).

- `static_parameter`

  Reference a static parameter by index.

- `=`

  Assignment. In the IR, the first argument is always a `SlotNumber` or a `GlobalRef`.

- `method`

  Adds a method to a generic function and assigns the result if necessary.

  Has a 1-argument form and a 3-argument form. The 1-argument form arises from the syntax `function foo end`. In the 1-argument form, the argument is a symbol. If this symbol already names a function in the current scope, nothing happens. If the symbol is undefined, a new function is created and assigned to the identifier specified by the symbol. If the symbol is defined but names a non-function, an error is raised. The definition of "names a function" is that the binding is constant, and refers to an object of singleton type. The rationale for this is that an instance of a singleton type uniquely identifies the type to add the method to. When the type has fields, it wouldn't be clear whether the method was being added to the instance or its type.

  The 3-argument form has the following arguments:

  - `args[1]`

    A function name, or `nothing` if unknown or unneeded. If a symbol, then the expression first behaves like the 1-argument form above. This argument is ignored from then on. It can be `nothing` when methods are added strictly by type, `(::T)(x) = x`, or when a method is being added to an existing function, `MyModule.f(x) = x`.

  - `args[2]`

    A `SimpleVector` of argument type data. `args[2][1]` is a `SimpleVector` of the argument types, and `args[2][2]` is a `SimpleVector` of type variables corresponding to the method's static parameters.

  - `args[3]`

    A `CodeInfo` of the method itself. For "out of scope" method definitions (adding a method to a function that also has methods defined in different scopes) this is an expression that evaluates to a `:lambda` expression.

- `struct_type`

  A 7-argument expression that defines a new `struct`:

  - `args[1]`

    The name of the `struct`

  - `args[2]`

    A `call` expression that creates a `SimpleVector` specifying its parameters

  - `args[3]`

    A `call` expression that creates a `SimpleVector` specifying its fieldnames

  - `args[4]`

    A `Symbol`, `GlobalRef`, or `Expr` specifying the supertype (e.g., `:Integer`, `GlobalRef(Core, :Any)`, or `:(Core.apply_type(AbstractArray, T, N))`)

  - `args[5]`

    A `call` expression that creates a `SimpleVector` specifying its fieldtypes

  - `args[6]`

    A Bool, true if `mutable`

  - `args[7]`

    The number of arguments to initialize. This will be the number of fields, or the minimum number of fields called by an inner constructor's new statement.

- `abstract_type`

  A 3-argument expression that defines a new abstract type. The arguments are the same as arguments 1, 2, and 4 of `struct_type` expressions.

- `primitive_type`

  A 4-argument expression that defines a new primitive type. Arguments 1, 2, and 4 are the same as `struct_type`. Argument 3 is the number of bits.

  > **Julia 1.5**
  >
  > `struct_type`, `abstract_type`, and `primitive_type` were removed in Julia 1.5 and replaced by calls to new builtins.

- `global`

  Declares a global binding.

- `const`

  Declares a (global) variable as constant.

- `new`

  Allocates a new struct-like object. First argument is the type. The new pseudo-function is lowered to this, and the type is always inserted by the compiler. This is very much an internal-only feature, and does no checking. Evaluating arbitrary new expressions can easily segfault.

- `splatnew`

  Similar to new, except field values are passed as a single tuple. Works similarly to `splat(new)` if new were a first-class function, hence the name.

- `isdefined`

  `Expr(:isdefined, :x)` returns a Bool indicating whether x has already been defined in the current scope.

- `the_exception`

  Yields the caught exception inside a `catch` block, as returned by `jl_current_exception()`.

- `enter`

  Enters an exception handler (`setjmp`). `args[1]` is the label of the catch block to jump to on error. Yields a token which is consumed by `pop_exception`.

- `leave`

  Pop exception handlers. `args[1]` is the number of handlers to pop.

- `pop_exception`

  Pop the stack of current exceptions back to the state at the associated `enter` when leaving a catch block. `args[1]` contains the token from the associated `enter`.

  > **Julia 1.1**
  >
  > `pop_exception` is new in Julia 1.1.

- `inbounds`

  Controls turning bounds checks on or off. A stack is maintained; if the first argument of this expression is true or false (`true` means bounds checks are disabled), it is pushed onto the stack. If the first argument is `:pop`, the stack is popped.

- `boundscheck`

  Has the value `false` if inlined into a section of code marked with `@inbounds`, otherwise has the value `true`.

- `loopinfo`

  Marks the end of the a loop. Contains metadata that is passed to `LowerSimdLoop` to either mark the inner loop of `@simd` expression, or to propagate information to LLVM loop passes.

- `copyast`

  Part of the implementation of quasi-quote. The argument is a surface syntax AST that is simply copied recursively and returned at run time.

- `meta`

  Metadata. `args[1]` is typically a symbol specifying the kind of metadata, and the rest of the arguments are free-form. The following kinds of metadata are commonly used:

  - `:inline` and `:noinline`: Inlining hints.

- `foreigncall`

  Statically-computed container for `ccall` information. The fields are:

  - `args[1]` : name

    The expression that'll be parsed for the foreign function.

  - `args[2]::Type` : RT

    The (literal) return type, computed statically when the containing method was defined.

  - `args[3]::SimpleVector` (of Types) : AT

    The (literal) vector of argument types, computed statically when the containing method was defined.

  - `args[4]::Int` : nreq

    The number of required arguments for a varargs function definition.

  - `args[5]::QuoteNode{Symbol}` : calling convention

    The calling convention for the call.

  - `args[6:5+length(args[3])]` : arguments

    The values for all the arguments (with types of each given in args[3]).

  - `args[6+length(args[3])+1:end]` : gc-roots

    The additional objects that may need to be gc-rooted for the duration of the call. See Working with LLVM for where these are derived from and how they get handled.

- `new_opaque_closure`

  Constructs a new opaque closure. The fields are:

  - `args[1]` : signature

    The function signature of the opaque closure. Opaque closures don't participate in dispatch, but the input types can be restricted.

  - `args[2]` : isva

    Indicates whether the closure accepts varargs.

  - `args[3]` : lb

    Lower bound on the output type. (Defaults to `Union{}`)

- `args[4]` : ub

  Upper bound on the output type. (Defaults to Any)

- `args[5]` : method

  The actual method as an `opaque_closure_method` expression.

- `args[6:end]` : captures

  The values captured by the opaque closure.

> **Julia 1.7**
>
> Opaque closures were added in Julia 1.7

Method

A unique'd container describing the shared metadata for a single method.

- `name`, `module`, `file`, `line`, `sig`

  Metadata to uniquely identify the method for the computer and the human.

- `ambig`

  Cache of other methods that may be ambiguous with this one.

- `specializations`

  Cache of all MethodInstance ever created for this Method, used to ensure uniqueness. Uniqueness is required for efficiency, especially for incremental precompile and tracking of method invalidation.

- `source`

  The original source code (if available, usually compressed).

- `generator`

  A callable object which can be executed to get specialized source for a specific method signature.

- `roots`

  Pointers to non-AST things that have been interpolated into the AST, required by compression of the AST, type-inference, or the generation of native code.

- `nargs`, `isva`, `called`, `is_for_opaque_closure`,

  Descriptive bit-fields for the source code of this Method.

- `primary_world`

  The world age that "owns" this Method.

MethodInstance

A unique'd container describing a single callable signature for a Method. See especially Proper maintenance and care of multi-threading locks for important details on how to modify these fields safely.

- `specTypes`

  The primary key for this MethodInstance. Uniqueness is guaranteed through a `def.specializations` lookup.

- `def`

  The `Method` that this function describes a specialization of. Or a `Module`, if this is a top-level Lambda expanded in Module, and which is not part of a Method.

- `sparam_vals`

  The values of the static parameters in `specTypes` indexed by `def.sparam_syms`. For the `MethodInstance` at `Method.unspecialized`, this is the empty `SimpleVector`. But for a runtime `MethodInstance` from the `MethodTable` cache, this will always be defined and indexable.

- `uninferred`

  The uncompressed source code for a toplevel thunk. Additionally, for a generated function, this is one of many places that the source code might be found.

- `backedges`

  We store the reverse-list of cache dependencies for efficient tracking of incremental reanalysis/recompilation work that may be needed after a new method definitions. This works by keeping a list of the other `MethodInstance` that have been inferred or optimized to contain a possible call to this `MethodInstance`. Those optimization results might be stored somewhere in the cache, or it might have been the result of something we didn't want to cache, such as constant propagation. Thus we merge all of those backedges to various cache entries here (there's almost always only the one applicable cache entry with a sentinel value for max_world anyways).

- `cache`

  Cache of `CodeInstance` objects that share this template instantiation.

CodeInstance

- `def`

  The `MethodInstance` that this cache entry is derived from.

- `rettype/rettype_const`

  The inferred return type for the `specFunctionObject` field, which (in most cases) is also the computed return type for the function in general.

- `inferred`

  May contain a cache of the inferred source for this function, or it could be set to `nothing` to just indicate `rettype` is inferred.

- `ftpr`

  The generic jlcall entry point.

- `jlcall_api`

  The ABI to use when calling `fptr`. Some significant ones include:

  - 0 - Not compiled yet
  - 1 - JL_CALLABLE `jl_value_t *(*)(jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`
  - 2 - Constant (value stored in `rettype_const`)
  - 3 - With Static-parameters forwarded `jl_value_t *(*)(jl_svec_t *sparams, jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`

  - 4 - Run in interpreter `jl_value_t *(*)(jl_method_instance_t *meth, jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`

- `min_world` / `max_world`

  The range of world ages for which this method instance is valid to be called. If max_world is the special token value `-1`, the value is not yet known. It may continue to be used until we encounter a backedge that requires us to reconsider.

CodeInfo

A (usually temporary) container for holding lowered source code.

- `code`

  An Any array of statements

- `slotnames`

  An array of symbols giving names for each slot (argument or local variable).

- `slotflags`

  A `UInt8` array of slot properties, represented as bit flags:

  - 0x02 - assigned (only false if there are *no* assignment statements with this var on the left)
  - 0x08 - used (if there is any read or write of the slot)
  - 0x10 - statically assigned once
  - 0x20 - might be used before assigned. This flag is only valid after type inference.

- `ssavaluetypes`

  Either an array or an `Int`.

  If an `Int`, it gives the number of compiler-inserted temporary locations in the function (the length of code array). If an array, specifies a type for each location.

- `ssaflags`

  Statement-level flags for each expression in the function. Many of these are reserved, but not yet implemented:

  - 0x01 « 0 = statement is marked as `@inbounds`
  - 0x01 « 1 = statement is marked as `@inline`
  - 0x01 « 2 = statement is marked as `@noinline`
  - 0x01 « 3 = statement is within a block that leads to `throw` call
  - 0x01 « 4 = statement may be removed if its result is unused, in particular it is thus be both pure and effect free
  - 0x01 « 5-6 = <unused>
  - 0x01 « 7 = <reserved> has out-of-band info

- `linetable`

  An array of source location objects

- `codelocs`

  An array of integer indices into the `linetable`, giving the location associated with each statement.

Optional Fields:

- `slottypes`

  An array of types for the slots.

- `rettype`

  The inferred return type of the lowered form (IR). Default value is `Any`.

- `method_for_inference_limit_heuristics`

  The `method_for_inference_heuristics` will expand the given method's generator if necessary during inference.

- `parent`

  The `MethodInstance` that "owns" this object (if applicable).

- `edges`

  Forward edges to method instances that must be invalidated.

- `min_world`/`max_world`

  The range of world ages for which this code was valid at the time when it had been inferred.

Boolean properties:

- `inferred`

  Whether this has been produced by type inference.

- `inlineable`

  Whether this should be eligible for inlining.

- `propagate_inbounds`

  Whether this should propagate `@inbounds` when inlined for the purpose of eliding `@boundscheck` blocks.

`UInt8` settings:

- `constprop`

  - 0 = use heuristic
  - 1 = aggressive
  - 2 = none

- `purity` Constructed from 5 bit flags:

  - 0x01 « 0 = this method is guaranteed to return or terminate consistently (`:consistent`)
  - 0x01 « 1 = this method is free from externally semantically visible side effects (`:effect_free`)
  - 0x01 « 2 = this method is guaranteed to not throw an exception (`:nothrow`)
  - 0x01 « 3 = this method is guaranteed to terminate (`:terminates_globally`)
  - 0x01 « 4 = the syntactic control flow within this method is guaranteed to terminate (`:terminates_locally`)

  See the documentation of `Base.@assume_effects` for more details.

103.3 More about types

使用 Julia 一段时间之后，你就会体会到类型在其中的基础性作用。本部分我们将深入到类型体系的内部，并着重关注 Parametric Types。

Types and sets (and Any and Union{}/Bottom)

Julia 的类型系统很容易会被看作是一种集合（set）。程序处理个体值，类型处理值的集合。但是集合与类型是两个不同的概念。一组值组成的集合 Set 本身也是一个值的集合。而类型描述的是一个组可能值组成的集合，即类型的值是不确定的。

函数 typeof 可以返回具体类型 T 包含的值的直接标签 T。而抽象类型描述的集合则可能会更大。

类型 Any 包含所有可能值。类型 Integer 是 Any 的一个子类型，而 Integer 的子类型有包括 Int，Int8 等其他具体类型。在内部表征上，Julia 类型系统还非常依赖类型 Bottom，也记做 Union{}。这对应于集合中的空集。

Julia 类型系统支持集合理论的标准操作：你可以用 T1 <: T2 来判断类型 T1 是否是 T2 的子集（子类型）。typeintersect 和 typejoin 可用来计算两个类型的交集和合集；用 Union 用于集合所有列出的类型。

```
julia> typeintersect(Int, Float64)
Union{}

julia> Union{Int, Float64}
Union{Float64, Int64}

julia> typejoin(Int, Float64)
Real

julia> typeintersect(Signed, Union{UInt8, Int8})
Int8

julia> Union{Signed, Union{UInt8, Int8}}
Union{UInt8, Signed}

julia> typejoin(Signed, Union{UInt8, Int8})
Integer

julia> typeintersect(Tuple{Integer, Float64}, Tuple{Int, Real})
Tuple{Int64, Float64}

julia> Union{Tuple{Integer, Float64}, Tuple{Int, Real}}
Union{Tuple{Int64, Real}, Tuple{Integer, Float64}}

julia> typejoin(Tuple{Integer, Float64}, Tuple{Int, Real})
Tuple{Integer, Real}
```

这些操作看起来很抽象，但是他们处于 Julia 语言的核心位置。例如，方法的派发过程就是对方法列表中的项目进行逐步搜索，直到找到一个其类型是方法标签子类型的参数元组为止。该算法有效的前提是方法必须按照特异性（specificity）进行排序，搜索过程必须从最具特异性的开始。所以，Julia 也需要对类型进行偏序排序 (partial order)，该函数与 <: 类似但又不完全相同。

UnionAll 类型

Julia 的类型系统也可以表征类型的迭代合集 (iterated union)，即某个变量的所有值的集合。当参数化类型的某些参数值是未知数时，迭代合集是非常有用的。

例如，下面这个数组 `Array{Int,2}` 有两个参数。如果其成分类型未知，该数组可以写成 `Array{T,2} where T`。该数组是取所有不同 T 值后的所有数组 `Array{T,2}` 的合集：`Union{Array{Int8,2}, Array{Int16,2}, ...}`。

类型的迭代集合由类型为 `UnionAll` 的对象表征。`UnionAll` 对象有一个类型为 `TypeVar` 的变量，此处为 `T`，和一个包裹化的类型，此处为 `Array{T,2}`。

考虑下面的例子：

```
f1(A::Array) = 1
f2(A::Array{Int}) = 2
f3(A::Array{T}) where {T<:Any} = 3
f4(A::Array{Any}) = 4
```

The signature - as described in Function calls - of f3 is a `UnionAll` type wrapping a tuple type: `Tuple{typeof(f3), Array{T}} where T`. All but f4 can be called with `a = [1,2]`; all but f2 can be called with `b = Any[1,2]`.

`dump()` 函数可用于进一步查看这些类型：

```
julia> dump(Array)
UnionAll
  var: TypeVar
    name: Symbol T
    lb: Union{}
    ub: Any
  body: UnionAll
    var: TypeVar
      name: Symbol N
      lb: Union{}
      ub: Any
    body: Array{T, N} <: DenseArray{T, N}
```

这说明数组 `Array` 实际上命名了一个 `UnionAll` 类型。嵌套其中的参数也都是 `UnionAll` 类型。句法 `Array{Int,2}` 等价于 `Array{Int}{2}`。`UnionAll` 在内部会被实例化为变量的特定值，每次一个，从最外侧开始。这使得省略尾端参数是自然而有意义的：类型 `Array{Int}` 与类型 `Array{Int, N} where N`。

`TypeVar` 本身不是类型，而是 `UnionAll` 类型的一个内部结构成分。类型变量的值会有上界和下界，分别由字段 `lb` 和 `ub` 表示。符号 `name` 是纯装饰性的（cosmetic）。在内部 `TypeVar` 是通过地址比较的。所以为了区分“不同”变量类型，`TypeVar` 是一个可更改类型，但通常不应修改它们。

你也可以手动创建 `TypeVar`:

```
julia> TypeVar(:V, Signed, Real)
Signed<:V<:Real
```

你可以用更简便的方式省掉除 `name` 之外的任意参数。

句法 `Array{T} where T<:Integer` 会被降级为如下格式。

```
let T = TypeVar(:T,Integer)
    UnionAll(T, Array{T})
end
```

所以极少需要手动建构 `TypeVar`（实际上这也应该是尽量避免的）。

自由变量

自由类型变量在类型系统中是至关重要的。如果类型 `T` 不包含一个引入变量 `V` 的 `UnionAll` 类型，那么类型 `T` 中的变量 `V` 就是自由的（free）。例如，类型 `Array{Array{V} where V<:Integer}` 没有自由变量，但该类型的子成分 `Array{V}` 则包含了一个自由变量 `V`。

从某种程度上讲，一个包含自由变量的类型根本就不是一个类型。例如，类型 `Array{Array{T}} where T` 是一个成分为数组的数组，且所有子数组元素的类型都相同。其内部类型 `Array{T}` 乍一看似乎包含所有数组类型。但外部数组的成分必须有相同类型，所以 `Array{T}` 不能指向所有数组。我们可以说 `Array{T}` 出现了很多次，但每次 `T` 值都必须相同。

故此，C 应用程序接口函数 `jl_has_free_typevars` 是非常重要的。若该函数的返回值为真，则说明类型中存在自由变量。此时子类型判断和其他类型函数中的返回结果没有太多意义。

类型名称 TypeNames

下面两个数组 Array 类型在功能上是相同的，但有不同的打印方式：

```
julia> TV, NV = TypeVar(:T), TypeVar(:N)
(T, N)

julia> Array
Array

julia> Array{TV, NV}
Array{T, N}
```

二者之间的差别可以通过类型的名称 `name` 字段来区分。名称 `name` 字段是一个类型为 `TypeName` 的对象。

```
julia> dump(Array{Int,1}.name)
TypeName
  name: Symbol Array
  module: Module Core
  names: empty SimpleVector
  wrapper: UnionAll
    var: TypeVar
      name: Symbol T
      lb: Union{}
      ub: Any
    body: UnionAll
      var: TypeVar
        name: Symbol N
        lb: Union{}
        ub: Any
      body: Array{T, N} <: DenseArray{T, N}
  cache: SimpleVector
```

```
  ...

linearcache: SimpleVector
  ...

hash: Int64 -7900426068641098781
mt: MethodTable
  name: Symbol Array
  defs: Nothing nothing
  cache: Nothing nothing
  max_args: Int64 0
  module: Module Core
  : Int64 0
  : Int64 0
```

与此相关的字段是包装器 wrapper，该字段存储了一个指向顶层类型（tip-level）的引用，用以产生新的数组 Array 类型。

```
julia> pointer_from_objref(Array)
Ptr{Cvoid} @0x00007fcc7de64850

julia> pointer_from_objref(Array.body.body.name.wrapper)
Ptr{Cvoid} @0x00007fcc7de64850

julia> pointer_from_objref(Array{TV,NV})
Ptr{Cvoid} @0x00007fcc80c4d930

julia> pointer_from_objref(Array{TV,NV}.name.wrapper)
Ptr{Cvoid} @0x00007fcc7de64850
```

数组 Array 的包装器 wrapper 字段指向它自己，而在数组 Array{TV,NV} 中，该字段则指回该类型的原始定义。

那其他字段都是什么作用呢？字段 hash 会给每个类型指派一个整数。要查看 cache 字段的内容，最好选不像数组那么常用的类型。我们可以自己创造一个类型：

```
julia> struct MyType{T,N} end

julia> MyType{Int,2}
MyType{Int64, 2}

julia> MyType{Float32, 5}
MyType{Float32, 5}
```

当参数类型被实例化时，每个具体类型都会被存储到类型缓存中（MyType.body.body.name.cache）。不过含有自由变量的实例是不会被缓存的。

元组类型

元组类型是一个有趣的特例。为了使派发在诸如 x::Tuple 之类的声明中正常工作，该类型必须能包含所有元组。我们可以查看一下元组的参数：

```
julia> Tuple
Tuple

julia> Tuple.parameters
svec(Vararg{Any})
```

与其他类型不同，元组类型的参数是共变的（covariant），所以类型 `Tuple` 能与任何类型的元组相匹配。

```
julia> typeintersect(Tuple, Tuple{Int,Float64})
Tuple{Int64, Float64}

julia> typeintersect(Tuple{Vararg{Any}}, Tuple{Int,Float64})
Tuple{Int64, Float64}
```

但是，如果一个可变元组 (Vararg) 类型含有自由变量，那么他描述的元组类型则可能就是不同的：

```
julia> typeintersect(Tuple{Vararg{T} where T}, Tuple{Int,Float64})
Tuple{Int64, Float64}

julia> typeintersect(Tuple{Vararg{T}} where T, Tuple{Int,Float64})
Union{}
```

当 `T` 绑定的 `UnionAll` 类型位于元组 `Tuple` 之外，即 `T` 是元组 `Tuple` 的自由变量时，一个唯一的 `T` 值必须作用与整个类型。此时异质的元组是不匹配的。

最后，意识到元组 `Tuple{}` 的独特性是有意义的：

```
julia> Tuple{}
Tuple{}

julia> Tuple{}.parameters
svec()

julia> typeintersect(Tuple{}, Tuple{Int})
Union{}
```

那么什么是元组类型的基本（primary）类型呢？

```
julia> pointer_from_objref(Tuple)
Ptr{Cvoid} @0x00007f5998a04370

julia> pointer_from_objref(Tuple{})
Ptr{Cvoid} @0x00007f5998a570d0

julia> pointer_from_objref(Tuple.name.wrapper)
Ptr{Cvoid} @0x00007f5998a04370

julia> pointer_from_objref(Tuple{}.name.wrapper)
Ptr{Cvoid} @0x00007f5998a04370
```

所以 `Tuple == Tuple{Vararg{Any}}` 事实上就是其基本类型。

对角变量

考虑类型 `Tuple{T,T} where T`。使用该标签的方法看起来会是下面的样子：

```
f(x::T, y::T) where {T} = ...
```

根据对类型 `UnionAll` 的通常理解，`T` 将覆盖所有的类型，包括 `T`，所以此类型应该等价于 `Tuple{Any,Any}`。但是这种理解会面临很多实际的问题。

首先，`T` 值需要在方法定之内可及。对诸如 `f(1, 1.0)` 的调用来说，`T` 的值是不明确的。`T` 可以是 `Union{Int,Float64}` 或 `Real`。直觉上，声明 `x::T` 意味着 `T` 应该为 `x` 的类型，即 `T === typeof(x)`。为了保有这种不变性，该方法应满足如下关系：`typeof(x) === typeof(y) === T`。这说明该方法只能被拥有相同类型的参数元组调用。

能依据两个值的类型是否相同进行派发是非常有用的（例如类型提升系统就用到了该机制）。所以我们有很多理由给 `Tuple{T,T} where T` 一个不同的含义。为此，我们在子类型系统中增加了如下规则：\textbf{如果一个变量在共变位置上出现了不止一次，那么其作用范围将仅局限于具体类型}。共变位置指在一个变量和引入该变量的 `UnionAll` 类型之间，只出现了元组 `Tuple` 类型和联合 `Union` 类型。这些变量被称为对角变量（diagonal variables）或具体变量（concrete variables）。

例如，元组 `Tuple{T,T} where T` 可以被看作是如下元组的集合 `Union{Tuple{Int8,Int8}, Tuple{Int16,Int16}, ...}`，即 `T` 包括所有的具体类型。该规则会产生一些有趣的子类型结果。例如 `Tuple{Real,Real}` 不是 `Tuple{T,T} where T` 的子类型，因为前者包含了诸如 `Tuple{Int8,Int16}` 的子类型，该子类型的元素类型是不同的。`Tuple{Real,Real}` 和 `Tuple{T,T} where T` 又一个不容小视的交集 `Tuple{T,T} where T<:Real`。但是 `Tuple{Real}` 是 `Tuple{T} where T` 的子集，因为此时 `T` 只出现了一次，所以是不对角的。

现在考虑如下签名：

```
f(a::Array{T}, x::T, y::T) where {T} = ...
```

此例中，`T` 出现在了非共变位置，即 `Array{T}` 之内。这意味着无论数组传递的类型是什么，该类型都会无歧义的决定 `T`，即 `T` 又一个等价性限制（equality constraint）。此时是不需要对角规则的，因为数组决定了 `T`，然后 `x` 和 `y` 则可以是 `T` 的任何子类型。所以非共变位置的变量均不受对角线规则的限制。上述定义的行为选择有一点矛盾，或许应该写成：

```
f(a::Array{T}, x::S, y::S) where {T, S<:T} = ...
```

为明确 `x` 和 `y` 应该有相同的类型。如果 `x` 和 `y` 可以有不同类型，这个版本的签名可以引入针对类型 `y` 的第三个变量。

下面一个难题是合集 `Union{}` 和对角线变量的交互作用，例如

```
f(x::Union{Nothing,T}, y::T) where {T} = ...
```

考虑一下这个生命的含义。`y` 的类型是 `T`。而 `x` 的类型或者与 `y` 相同 `T` 或类型为空 `Nothing`。下面调用都是匹配的。

```
f(1, 1)
f("", "")
f(2.0, 2.0)
```

```
f(nothing, 1)
f(nothing, "")
f(nothing, 2.0)
```

上述例子告诉我们：当 x 的类型是空集 nothing::Nothing 时，y 的类型就没有任何限制了。此时方法标签中字段 y 可以是任何类型，即 y::Any。确实，下面两个类型是等价的：

```
(Tuple{Union{Nothing,T},T} where T) == Union{Tuple{Nothing,Any}, Tuple{T,T} where T}
```

一般的规则是：如果出现在共变位置的一个具体变量只被子类型算法（subtyping algorithm）使用一次，则该具体变量的行为就会像一个抽象变量。上例中，当 x 类型为空时 Nothing，集合 Union{Nothing,T} 中的类型 T 不起作用，即 T 只在第二个槽中被用到。此时，无论把 Tuple{T} where T 中的 T 限制为具体类型还是不限制，该类型都等价于 Tuple{Any}。

当变量出现在非共变位置时，无论是否被使用，该变量都不再是具体变量。否则类型的行为就会因为比较对象类型的不同而不同，从而违反传递律（transitive）。例如，

```
Tuple{Int,Int8,Vector{Integer}} <: Tuple{T,T,Vector{Union{Integer,T}}} where T
```

如果忽略掉合集 Union 内部的 T，则 T 就是具体的，且上述判断结果是"否"，因为左侧元组前两个成分的类型是不同的。这两个类型不相同。再看下例：

```
Tuple{Int,Int8,Vector{Any}} <: Tuple{T,T,Vector{Union{Integer,T}}} where T
```

此处合集 Union 内的 T 是不能被忽略的，因为 T 必须等于 Any，即 T == Any。所以 T 不能是具体的，且判断结果为"真"。所以，类型 T 的具体还是抽象是受制于其他类型特征的。这是不被接受的，因为类型的含义必须是清晰和自洽的。所以，向量 Vector 之内的 T 需要同时考虑这两种情况。

对角变量的子类型

对角变量的子系统算法（substyping algorithm）有两个成分：（1）确定变量出现次数；（2）确保对角变量只包含具体类型。

第一个任务由两个计数器完成。环境中的每个自由变量都有两个计数器 occurs_inv 和 occurs_cov（在文件 src/subtype.c 中），分别用于追踪不变和共变的出现次数。如果 occurs_inv == 0 && occurs_cov > 1 则这个变量就是对角的。

Most operations for dealing with types are found in the files jltypes.c and subtype.c. A good way to start is to watch subtyping in action. Build Julia with make debug and fire up Julia within a debugger. gdb debugging tips has some tips which may be useful.

```
julia> function mysubtype(a,b)
           ccall(:jl_breakpoint, Cvoid, (Any,), nothing)
           a <: b
       end
```

一旦该断点被触发，你就可以在其他函数中设定断点了。

作为热身，试一下下面代码

```
mysubtype(Tuple{Int, Float64}, Tuple{Integer, Real})
```

你也可以用一个更复杂的例子让其变得更有趣：

```
mysubtype(Tuple{Array{Int,2}, Int8}, Tuple{Array{T}, T} where T)
```

子类型和方法排序

函数 `type_morespecific` 可用来对函数的方法列表从最具特异行到最不具特异性进行部分排序(partial order)。特异性是严格的：如果 a 比 b 更特异，则 b 就不比 a 更特异。

如果 a 是 b 的严格子类型(strict subtype)，那么 a 就自动的比 b 更特异。接下来，函数 `type_morespecific` 还引进了一些不太形式化的规则。例如，子类型 `subtype` 对参数的数量敏感，而函数 `type_morespecific` 则不敏感。特别的，`Tuple{Int,AbstractFloat}` 比 `Tuple{Integer}` 更特异，虽然前者不是后者的子类型。此外，`Tuple{Int,AbstractFloat}` 和 `Tuple{Integer,Float64}` 不存在特异性关系。类似的，`Tuple{Int,Vararg{Int}}` 不是 `Tuple{Integer}` 的子类型，但前者比后者更特异。但是，长度确实也影响特异性关系 `morespecific`，如 `Tuple{Int,Int}` 比 `Tuple{Int,Vararg{Int}}` 更特异。

要调试方法的排序，定义下面的函数很方便：

```
type_morespecific(a, b) = ccall(:jl_type_morespecific, Cint, (Any,Any), a, b)
```

它可以被用来测试元组 a 是否比元组 b 更特异。

103.4 Memory layout of Julia Objects

Object layout (`jl_value_t`)

The `jl_value_t` struct is the name for a block of memory owned by the Julia Garbage Collector, representing the data associated with a Julia object in memory. Absent any type information, it is simply an opaque pointer:

```
typedef struct jl_value_t* jl_pvalue_t;
```

Each `jl_value_t` struct is contained in a `jl_typetag_t` struct that contains metadata information about the Julia object, such as its type and garbage collector (gc) reachability:

```
typedef struct {
    opaque metadata;
    jl_value_t value;
} jl_typetag_t;
```

The type of any Julia object is an instance of a leaf `jl_datatype_t` object. The `jl_typeof()` function can be used to query for it:

```
jl_value_t *jl_typeof(jl_value_t *v);
```

The layout of the object depends on its type. Reflection methods can be used to inspect that layout. A field can be accessed by calling one of the get-field methods:

```
jl_value_t *jl_get_nth_field_checked(jl_value_t *v, size_t i);
jl_value_t *jl_get_field(jl_value_t *o, char *fld);
```

If the field types are known, a priori, to be all pointers, the values can also be extracted directly as an array access:

```
jl_value_t *v = value->fieldptr[n];
```

As an example, a "boxed" `uint16_t` is stored as follows:

```
struct {
    opaque metadata;
    struct {
        uint16_t data;        // -- 2 bytes
    } jl_value_t;
};
```

This object is created by `jl_box_uint16()`. Note that the `jl_value_t` pointer references the data portion, not the metadata at the top of the struct.

A value may be stored "unboxed" in many circumstances (just the data, without the metadata, and possibly not even stored but just kept in registers), so it is unsafe to assume that the address of a box is a unique identifier. The "egal" test (corresponding to the === function in Julia), should instead be used to compare two unknown objects for equivalence:

```
int jl_egal(jl_value_t *a, jl_value_t *b);
```

This optimization should be relatively transparent to the API, since the object will be "boxed" on-demand, whenever a `jl_value_t` pointer is needed.

Note that modification of a `jl_value_t` pointer in memory is permitted only if the object is mutable. Otherwise, modification of the value may corrupt the program and the result will be undefined. The mutability property of a value can be queried for with:

```
int jl_is_mutable(jl_value_t *v);
```

If the object being stored is a `jl_value_t`, the Julia garbage collector must be notified also:

```
void jl_gc_wb(jl_value_t *parent, jl_value_t *ptr);
```

However, the Embedding Julia section of the manual is also required reading at this point, for covering other details of boxing and unboxing various types, and understanding the gc interactions.

Mirror structs for some of the built-in types are defined in `julia.h`. The corresponding global `jl_datatype_t` objects are created by `jl_init_types` in `jltypes.c`.

Garbage collector mark bits

The garbage collector uses several bits from the metadata portion of the `jl_typetag_t` to track each object in the system. Further details about this algorithm can be found in the comments of the garbage collector implementation in `gc.c`.

Object allocation

Most new objects are allocated by `jl_new_structv()`:

```
jl_value_t *jl_new_struct(jl_datatype_t *type, ...);
jl_value_t *jl_new_structv(jl_datatype_t *type, jl_value_t **args, uint32_t na);
```

Although, `isbits` objects can be also constructed directly from memory:

```
jl_value_t *jl_new_bits(jl_value_t *bt, void *data)
```

And some objects have special constructors that must be used instead of the above functions:

Types:

```
jl_datatype_t *jl_apply_type(jl_datatype_t *tc, jl_tuple_t *params);
jl_datatype_t *jl_apply_array_type(jl_datatype_t *type, size_t dim);
```

While these are the most commonly used options, there are more low-level constructors too, which you can find declared in `julia.h`. These are used in `jl_init_types()` to create the initial types needed to bootstrap the creation of the Julia system image.

Tuples:

```
jl_tuple_t *jl_tuple(size_t n, ...);
jl_tuple_t *jl_tuplev(size_t n, jl_value_t **v);
jl_tuple_t *jl_alloc_tuple(size_t n);
```

The representation of tuples is highly unique in the Julia object representation ecosystem. In some cases, a `Base.tuple()` object may be an array of pointers to the objects contained by the tuple equivalent to:

```
typedef struct {
    size_t length;
    jl_value_t *data[length];
} jl_tuple_t;
```

However, in other cases, the tuple may be converted to an anonymous `isbits` type and stored unboxed, or it may not stored at all (if it is not being used in a generic context as a `jl_value_t*`).

Symbols:

```
jl_sym_t *jl_symbol(const char *str);
```

Functions and MethodInstance:

```
jl_function_t *jl_new_generic_function(jl_sym_t *name);
jl_method_instance_t *jl_new_method_instance(jl_value_t *ast, jl_tuple_t *sparams);
```

Arrays:

```
jl_array_t *jl_new_array(jl_value_t *atype, jl_tuple_t *dims);
jl_array_t *jl_new_arrayv(jl_value_t *atype, ...);
jl_array_t *jl_alloc_array_1d(jl_value_t *atype, size_t nr);
jl_array_t *jl_alloc_array_2d(jl_value_t *atype, size_t nr, size_t nc);
jl_array_t *jl_alloc_array_3d(jl_value_t *atype, size_t nr, size_t nc, size_t z);
jl_array_t *jl_alloc_vec_any(size_t n);
```

Note that many of these have alternative allocation functions for various special-purposes. The list here reflects the more common usages, but a more complete list can be found by reading the `julia.h` header file.

Internal to Julia, storage is typically allocated by `newstruct()` (or `newobj()` for the special types):

```
jl_value_t *newstruct(jl_value_t *type);
jl_value_t *newobj(jl_value_t *type, size_t nfields);
```

And at the lowest level, memory is getting allocated by a call to the garbage collector (in `gc.c`), then tagged with its type:

```
jl_value_t *jl_gc_allocobj(size_t nbytes);
void jl_set_typeof(jl_value_t *v, jl_datatype_t *type);
```

Out of date Warning

The documentation and usage for the function `jl_gc_allocobj` may be out of date

Note that all objects are allocated in multiples of 4 bytes and aligned to the platform pointer size. Memory is allocated from a pool for smaller objects, or directly with `malloc()` for large objects.

Singleton Types

Singleton types have only one instance and no data fields. Singleton instances have a size of 0 bytes, and consist only of their metadata. e.g. `nothing::Nothing`.

See Singleton Types and Nothingness and missing values

103.5 Julia 代码的 eval

学习 Julia 语言如何运行代码的最难的一部分是学习如何让所有的小部分工作协同工作来执行一段代码。

每个代码块通常会通过许多步骤来执行，在转变为期望的结果之前（但愿如此）。并且你可能不熟悉它们的名称，例如（非特定顺序）：flisp，AST，C++，LLVM，`eval`，`typeinf`，`macroexpand`，sysimg（或 system image），启动，变异，解析，执行，即时编译器，解释器解释，装箱，拆箱，内部函数，原始函数

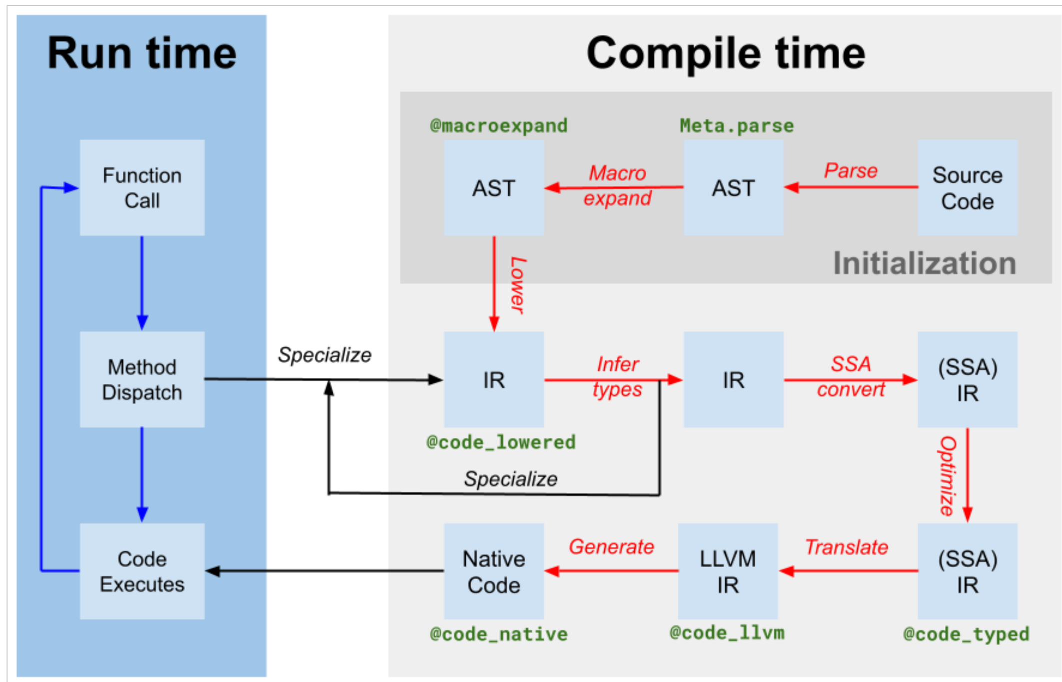

Figure 103.1: Diagram of the compiler flow

Definitions

- REPL

  REPL 表示读取-求值-输出-循环（Read-Eval-Print Loop）。我们管这个命令行环境的简称就叫 REPL。

- AST

  抽象语法树（Abstract Syntax Tree）是代码结构的数据表现。在这种表现形式下代码被符号化，因此更加方便操作和执行。

Julia Execution

整个进程的千里之行如下：

1. 用户打开了 `julia`。
2. The C function `main()` from `cli/loader_exe.c` gets called. This function processes the command line arguments, filling in the `jl_options` struct and setting the variable ARGS. It then initializes Julia (by calling `julia_init` in `init.c`, which may load a previously compiled sysimg). Finally, it passes off control to Julia by calling `Base._start()`.

3. When _start() takes over control, the subsequent sequence of commands depends on the command line arguments given. For example, if a filename was supplied, it will proceed to execute that file. Otherwise, it will start an interactive REPL.

4. Skipping the details about how the REPL interacts with the user, let's just say the program ends up with a block of code that it wants to run.

5. If the block of code to run is in a file, jl_load(char *filename) gets invoked to load the file and parse it. Each fragment of code is then passed to eval to execute.

6. Each fragment of code (or AST), is handed off to eval() to turn into results.

7. eval() takes each code fragment and tries to run it in jl_toplevel_eval_flex().

8. jl_toplevel_eval_flex() decides whether the code is a "toplevel" action (such as using or module), which would be invalid inside a function. If so, it passes off the code to the toplevel interpreter.

9. jl_toplevel_eval_flex() then expands the code to eliminate any macros and to "lower" the AST to make it simpler to execute.

10. jl_toplevel_eval_flex() then uses some simple heuristics to decide whether to JIT compile the AST or to interpret it directly.

11. The bulk of the work to interpret code is handled by eval in interpreter.c.

12. If instead, the code is compiled, the bulk of the work is handled by codegen.cpp. Whenever a Julia function is called for the first time with a given set of argument types, type inference will be run on that function. This information is used by the codegen step to generate faster code.

13. Eventually, the user quits the REPL, or the end of the program is reached, and the _start() method returns.

14. Just before exiting, main() calls jl_atexit_hook(exit_code). This calls Base._atexit() (which calls any functions registered to atexit() inside Julia). Then it calls jl_gc_run_all_finalizers(). Finally, it gracefully cleans up all libuv handles and waits for them to flush and close.

Parsing

The Julia parser is a small lisp program written in femtolisp, the source-code for which is distributed inside Julia in src/flisp.

The interface functions for this are primarily defined in jlfrontend.scm. The code in ast.c handles this handoff on the Julia side.

The other relevant files at this stage are julia-parser.scm, which handles tokenizing Julia code and turning it into an AST, and julia-syntax.scm, which handles transforming complex AST representations into simpler, "lowered" AST representations which are more suitable for analysis and execution.

If you want to test the parser without re-building Julia in its entirety, you can run the frontend on its own as follows:

```
$ cd src
$ flisp/flisp
> (load "jlfrontend.scm")
> (jl-parse-file "<filename>")
```

Macro Expansion

When `eval()` encounters a macro, it expands that AST node before attempting to evaluate the expression. Macro expansion involves a handoff from `eval()` (in Julia), to the parser function `jl_macroexpand()` (written in `flisp`) to the Julia macro itself (written in - what else - Julia) via `fl_invoke_julia_macro()`, and back.

Typically, macro expansion is invoked as a first step during a call to `Meta.lower()`/`jl_expand()`, although it can also be invoked directly by a call to `macroexpand()`/`jl_macroexpand()`.

Type Inference

Type inference is implemented in Julia by `typeinf()` in `compiler/typeinfer.jl`. Type inference is the process of examining a Julia function and determining bounds for the types of each of its variables, as well as bounds on the type of the return value from the function. This enables many future optimizations, such as unboxing of known immutable values, and compile-time hoisting of various run-time operations such as computing field offsets and function pointers. Type inference may also include other steps such as constant propagation and inlining.

More Definitions

- JIT

  Just-In-Time Compilation The process of generating native-machine code into memory right when it is needed.

- LLVM

  Low-Level Virtual Machine (a compiler) The Julia JIT compiler is a program/library called libLLVM. Codegen in Julia refers both to the process of taking a Julia AST and turning it into LLVM instructions, and the process of LLVM optimizing that and turning it into native assembly instructions.

- C++

  The programming language that LLVM is implemented in, which means that codegen is also implemented in this language. The rest of Julia's library is implemented in C, in part because its smaller feature set makes it more usable as a cross-language interface layer.

- box

  This term is used to describe the process of taking a value and allocating a wrapper around the data that is tracked by the garbage collector (gc) and is tagged with the object's type.

- unbox

  The reverse of boxing a value. This operation enables more efficient manipulation of data when the type of that data is fully known at compile-time (through type inference).

- generic function

  A Julia function composed of multiple "methods" that are selected for dynamic dispatch based on the argument type-signature

- anonymous function or "method"

  A Julia function without a name and without type-dispatch capabilities

- primitive function

  A function implemented in C but exposed in Julia as a named function "method" (albeit without generic function dispatch capabilities, similar to a anonymous function)

- intrinsic function

  A low-level operation exposed as a function in Julia. These pseudo-functions implement operations on raw bits such as add and sign extend that cannot be expressed directly in any other way. Since they operate on bits directly, they must be compiled into a function and surrounded by a call to `Core.Intrinsics.box(T, ...)` to reassign type information to the value.

JIT Code Generation

Codegen is the process of turning a Julia AST into native machine code.

The JIT environment is initialized by an early call to `jl_init_codegen in codegen.cpp`.

On demand, a Julia method is converted into a native function by the function `emit_function(jl_method_instance_t*)`. (note, when using the MCJIT (in LLVM v3.4+), each function must be JIT into a new module.) This function recursively calls `emit_expr()` until the entire function has been emitted.

Much of the remaining bulk of this file is devoted to various manual optimizations of specific code patterns. For

example, `emit_known_call()` knows how to inline many of the primitive functions (defined in `builtins.c`) for various combinations of argument types.

Other parts of codegen are handled by various helper files:

- `debuginfo.cpp`

  Handles backtraces for JIT functions

- `ccall.cpp`

  Handles the ccall and llvmcall FFI, along with various `abi_*.cpp` files

- `intrinsics.cpp`

  Handles the emission of various low-level intrinsic functions

Bootstrapping

The process of creating a new system image is called "bootstrapping".

The etymology of this word comes from the phrase "pulling oneself up by the bootstraps", and refers to the idea of starting from a very limited set of available functions and definitions and ending with the creation of a full-featured environment.

System Image

The system image is a precompiled archive of a set of Julia files. The `sys.ji` file distributed with Julia is one such system image, generated by executing the file `sysimg.jl`, and serializing the resulting environment (including Types, Functions, Modules, and all other defined values) into a file. Therefore, it contains a frozen version of the `Main`, `Core`, and `Base` modules (and whatever else was in the environment at the end of bootstrapping). This serializer/deserializer is implemented by `jl_save_system_image/jl_restore_system_image in staticdata.c`.

If there is no sysimg file (`jl_options.image_file == NULL`), this also implies that `--build` was given on the command line, so the final result should be a new sysimg file. During Julia initialization, minimal `Core` and `Main` modules are created. Then a file named `boot.jl` is evaluated from the current directory. Julia then evaluates any file given as a command line argument until it reaches the end. Finally, it saves the resulting environment to a "sysimg" file for use as a starting point for a future Julia run.

103.6 Calling Conventions

Julia uses three calling conventions for four distinct purposes:

Name	Prefix	Purpose
Native	`julia_`	Speed via specialized signatures
JL Call	`jlcall_`	Wrapper for generic calls
JL Call	`jl_`	Builtins
C ABI	`jlcapi_`	Wrapper callable from C

Julia Native Calling Convention

The native calling convention is designed for fast non-generic calls. It usually uses a specialized signature.

- LLVM ghosts (zero-length types) are omitted.
- LLVM scalars and vectors are passed by value.
- LLVM aggregates (arrays and structs) are passed by reference.

A small return values is returned as LLVM return values. A large return values is returned via the "structure return" (`sret`) convention, where the caller provides a pointer to a return slot.

An argument or return values that is a homogeneous tuple is sometimes represented as an LLVM vector instead of an LLVM array.

JL Call Convention

The JL Call convention is for builtins and generic dispatch. Hand-written functions using this convention are declared via the macro `JL_CALLABLE`. The convention uses exactly 3 parameters:

- F - Julia representation of function that is being applied
- `args` - pointer to array of pointers to boxes
- `nargs` - length of the array

The return value is a pointer to a box.

C ABI

C ABI wrappers enable calling Julia from C. The wrapper calls a function using the native calling convention.

Tuples are always represented as C arrays.

103.7 本机代码生成过程的高级概述

指针的表示

When emitting code to an object file, pointers will be emitted as relocations. The deserialization code will ensure any object that pointed to one of these constants gets recreated and contains the right runtime pointer.

Otherwise, they will be emitted as literal constants.

To emit one of these objects, call `literal_pointer_val`. It'll handle tracking the Julia value and the LLVM global, ensuring they are valid both for the current runtime and after deserialization.

When emitted into the object file, these globals are stored as references in a large `gvals` table. This allows the deserializer to reference them by index, and implement a custom manual mechanism similar to a Global Offset Table (GOT) to restore them.

Function pointers are handled similarly. They are stored as values in a large `fvals` table. Like globals, this allows the deserializer to reference them by index.

Note that `extern` functions are handled separately, with names, via the usual symbol resolution mechanism in the linker.

Note too that `ccall` functions are also handled separately, via a manual GOT and Procedure Linkage Table (PLT).

Representation of Intermediate Values

Values are passed around in a `jl_cgval_t` struct. This represents an R-value, and includes enough information to determine how to assign or pass it somewhere.

They are created via one of the helper constructors, usually: `mark_julia_type` (for immediate values) and `mark_julia_slot` (for pointers to values).

The function `convert_julia_type` can transform between any two types. It returns an R-value with `cgval.typ` set to `typ`. It'll cast the object to the requested representation, making heap boxes, allocating stack copies, and computing tagged unions as needed to change the representation.

By contrast `update_julia_type` will change `cgval.typ` to `typ`, only if it can be done at zero-cost (i.e. without emitting any code).

Union representation

Inferred union types may be stack allocated via a tagged type representation.

The primitive routines that need to be able to handle tagged unions are:

- mark-type
- load-local
- store-local
- isa
- is
- emit_typeof
- emit_sizeof
- boxed
- unbox
- specialized cc-ret

Everything else should be possible to handle in inference by using these primitives to implement union-splitting.

The representation of the tagged-union is as a pair of `< void* union, byte selector >`. The selector is fixed-size as `byte & 0x7f`, and will union-tag the first 126 isbits. It records the one-based depth-first count into the type-union of the isbits objects inside. An index of zero indicates that the `union*` is actually a tagged heap-allocated `jl_value_t*`, and needs to be treated as normal for a boxed object rather than as a tagged union.

The high bit of the selector (`byte & 0x80`) can be tested to determine if the `void*` is actually a heap-allocated (`jl_value_t*`) box, thus avoiding the cost of re-allocating a box, while maintaining the ability to efficiently handle union-splitting based on the low bits.

It is guaranteed that `byte & 0x7f` is an exact test for the type, if the value can be represented by a tag –it will never be marked `byte = 0x80`. It is not necessary to also test the type-tag when testing `isa`.

The `union*` memory region may be allocated at *any* size. The only constraint is that it is big enough to contain the data currently specified by `selector`. It might not be big enough to contain the union of all types that could be stored there according to the associated Union type field. Use appropriate care when copying.

Specialized Calling Convention Signature Representation

A `jl_returninfo_t` object describes the calling convention details of any callable.

If any of the arguments or return type of a method can be represented unboxed, and the method is not varargs, it'll be given an optimized calling convention signature based on its `specTypes` and `rettype` fields.

The general principles are that:

- Primitive types get passed in int/float registers.
- Tuples of VecElement types get passed in vector registers.
- Structs get passed on the stack.
- Return values are handle similarly to arguments, with a size-cutoff at which they will instead be returned via a hidden sret argument.

The total logic for this is implemented by `get_specsig_function` and `deserves_sret`.

Additionally, if the return type is a union, it may be returned as a pair of values (a pointer and a tag). If the union values can be stack-allocated, then sufficient space to store them will also be passed as a hidden first argument. It is up to the callee whether the returned pointer will point to this space, a boxed object, or even other constant memory.

103.8 Julia 函数

本文档将解释函数、方法定义以及方法表是如何工作的。

方法表

Julia 中的每个函数都是泛型函数。泛型函数在概念上是单个函数，但由许多定义或方法组成。泛型函数的方法储存在方法表中。方法表（类型 `MethodTable`）与 `TypeName` 相关。`TypeName` 描述了一系列参数化类型。例如，`Complex{Float32}` 和 `Complex{Float64}` 共享相同的 type name 对象 `Complex`。

Julia 中的所有对象都可能是可调用的，因为每个对象都有类型，而类型又有 `TypeName`。

函数调用

给定调用 `f(x,y)`，会执行以下步骤：首先，用 `typeof(f).name.mt` 访问要使用的方法表。其次，生成一个参数元组类型 `Tuple{typeof(f), typeof(x), typeof(y)}`。请注意，函数本身的类型是第一个元素。这因为该类型可能有参数，所以需要参与派发。这个元组类型会在方法表中查找。

这个派发过程由 `jl_apply_generic` 执行，它有两个参数：一个指向由值 `f`、`x` 和 `y` 组成的数组的指针，以及值的数量（此例中是 `3`）。

在整个系统中，处理函数和参数列表的 API 有两种：一种单独接收函数和参数，一种接收一个单独的参数结构。在第一种 API 中，「参数」部分不包含函数的相关信息，因为它是单独传递的。在第二种 API 中，函数是参数结构的第一个元素。

例如，以下用于执行调用的函数只接收 `args` 指针，因此 args 数组的第一个元素将会是要调用的函数：

```
jl_value_t *jl_apply(jl_value_t **args, uint32_t nargs)
```

这个用于相同功能的入口点单独接收该函数，因此 args 数组中不包含该函数：

```
jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t nargs);
```

添加方法

Given the above dispatch process, conceptually all that is needed to add a new method is (1) a tuple type, and (2) code for the body of the method. jl_method_def implements this operation. jl_method_table_for is called to extract the relevant method table from what would be the type of the first argument. This is much more complicated than the corresponding procedure during dispatch, since the argument tuple type might be abstract. For example, we can define:

```
(::Union{Foo{Int},Foo{Int8}})(x) = 0
```

这是可行的，因为所有可能的匹配方法都属于同一方法表。

创建泛型函数

因为每个对象都是可调用的，所以创建泛型函数不需要特殊的东西。因此，jl_new_generic_function 只是创建一个新的 Function 的单态类型（大小为 0）并返回它的实例。函数可有一个帮助记忆的「显示名称」，用于调试信息和打印对象。例如，Base.sin 的名称为 sin。按照约定，所创建类型的名称与函数名称相同，带前缀 #。所以 typeof(sin) 即 Base.#sin。

闭包

闭包只是一个可调用对象，其字段名称对应于被捕获的变量。例如，以下代码：

```
function adder(x)
    return y->x+y
end
```

（大致）降低为：

```
struct ##1{T}
    x::T
end

(_::##1)(y) = _.x + y

function adder(x)
    return ##1(x)
end
```

构造函数

构造函数调用只是对类型的调用。Type 的方法表包含所有的构造函数定义。Type 的所有子类型（Type、UnionAll、Union 和 DataType）目前通过特殊的安排方式共享一个方法表。

内置函数

「内置」函数定义在 `Core` 模块中，有：

```
<: === _abstracttype _apply_iterate _apply_pure _call_in_world
_call_in_world_total _call_latest _compute_sparams _equiv_typedef _expr
_primitivetype _setsuper! _structtype _svec_ref _typebody! _typevar applicable
apply_type arrayref arrayset arraysize compilerbarrier const_arrayref donotdelete
fieldtype finalizer get_binding_type getfield getglobal ifelse invoke isa
isdefined modifyfield! nfields replacefield! set_binding_type! setfield!
setglobal! sizeof svec swapfield! throw tuple typeassert typeof
```

这些都是单态对象，其类型为 `Builtin` 的子类型，而或后者为 `Function` 的子类型。它们的用处是在运行时暴露遵循「jlcall」调用约定的入口点。

```
jl_value_t *(jl_value_t*, jl_value_t**, uint32_t)
```

内建函数的方法表是空的。相反地，它们具有单独的 catch-all 方法缓存条目（`Tuple{Vararg{Any}}`），其 jlcall fptr 指向正确的函数。这是一种 hack，但效果相当不错。

关键字参数

Keyword arguments work by adding methods to the kwcall function. This function is usually the "keyword argument sorter" or "keyword sorter", which then calls the inner body of the function (defined anonymously). Every definition in the kwsorter function has the same arguments as some definition in the normal method table, except with a single `NamedTuple` argument prepended, which gives the names and values of passed keyword arguments. The kwsorter's job is to move keyword arguments into their canonical positions based on name, plus evaluate and substitute any needed default value expressions. The result is a normal positional argument list, which is then passed to yet another compiler-generated function.

理解该过程的最简单方法是查看关键字参数方法的定义的降低方式。代码：

```julia
function circle(center, radius; color = black, fill::Bool = true, options...)
    # draw
end
```

实际上生成三个方法定义。第一个方法是一个接收所有参数（包括关键字参数）作为其位置参数的函数，其代码包含该方法体。它有一个自动生成的名称：

```julia
function #circle#1(color, fill::Bool, options, circle, center, radius)
    # draw
end
```

第二个方法是原始 `circle` 函数的普通定义，负责处理没有传递关键字参数的情况：

```julia
function circle(center, radius)
    #circle#1(black, true, pairs(NamedTuple()), circle, center, radius)
end
```

这只是派发到第一个方法，传递默认值。pairs 应用于其余的参数组成的具名元组，以提供键值对迭代。请注意，如果方法不接受其余的关键字参数，那么此参数不存在。

最后，kwsorter 定义为：

```
function (::Core.kwftype(typeof(circle)))(kws, circle, center, radius)
    if haskey(kws, :color)
        color = kws.color
    else
        color = black
    end
    # etc.

    # put remaining kwargs in `options`
    options = structdiff(kws, NamedTuple{(:color, :fill)})

    # if the method doesn't accept rest keywords, throw an error
    # unless `options` is empty

    #circle#1(color, fill, pairs(options), circle, center, radius)
end
```

函数 Core.kwftype(t) 创建字段 t.name.mt.kwsorter（如果它未被创建），并返回该函数的类型。

此设计的特点是不使用关键字参数的调用点不需要特殊处理；这一切的工作方式好像它们根本不是语言的一部分。不使用关键字参数的调用点直接派发到被调用函数的 kwsorter。例如，调用：

```
circle((0,0), 1.0, color = red; other...)
```

降低为：

```
kwcall(merge((color = red,), other), circle, (0,0), 1.0)
```

kwcall (also inCore) denotes a kwcall signature and dispatch. The keyword splatting operation (written as other...) calls the named tuple merge function. This function further unpacks each *element* of other, expecting each one to contain two values (a symbol and a value). Naturally, a more efficient implementation is available if all splatted arguments are named tuples. Notice that the original circle function is passed through, to handle closures.

Compiler efficiency issues

为每个函数生成新类型在与 Julia 的「默认专门化所有参数」这一设计理念结合使用时，可能对编译器资源的使用产生严重后果。实际上，此设计的初始实现经历了更长的测试和构造时间、高内存占用以及比基线大近乎 2 倍的系统镜像。在一个幼稚的实现中，该问题非常严重，以至于系统几乎无法使用。需要进行几项重要的优化才能使设计变得可行。

第一个问题是函数值参数的不同值导致函数的过度专门化。许多函数只是将参数「传递」到其它地方，例如，到另一个函数或存储位置。这种函数不需要为每个可能传入的闭包专门化。幸运的是，这种情况很容易区分，只需考虑函数是否调用它的某个参数（即，参数出现在某处的「头部位置」）。性能关键的高阶函数，如 map，肯定会直接调用它们的参数函数，因此仍然会按预期进行专门化。此优化通过在前端记录 analyze-variables 传递期间所调用的参数来实现。当 cache_method 看到某个在 Function 类型层次结构的参数传递到声明为 Any 或 Function 的槽时，它的行为就好像应用了 @nospecialize 注释一样。这种启发式方法在实践中似乎非常有效。

下一个问题涉及方法缓存哈希表的结构。经验研究表明，绝大多数动态分派调用只涉及一个或两个元素。反过来看，只考虑第一个元素便可解决许多这些情况。(旁白：单派发的支持者根本不会对此感到惊讶。但是，这个观点意味着「多重派发在实践中很容易优化」，因此我们应该使用它，而不是「我们应该使用单派发」!）因此，方法缓存使用第一个参数作为其主键。但请注意，这对应于函数调用的元组类型的第二个元素（第一个元素是函数本身的类型）。通常，头部位置的类型非常少变化——实际上，大多数函数属于没有参数的单态类型。但是，构造函数不是这种情况，一个方法表便保存了所有类型的构造函数。因此，`Type` 方法表是特殊的，使用元组类型的第一个元素而不是第二个。

前端为所有闭包生成类型声明。起初，这通过生成通常的类型声明来实现。但是，这产生了大量的构造函数，这些构造函数全都很简单（只是将所有参数传递给 `new`)。因为方法是部分排序的，所以插入所有这些方法是 O(n^2)，此外要保留的方法实在太多了。这可通过直接生成 `struct_type` 表达式（绕过默认的构造函数生成）并直接使用 `new` 来创建闭包的实例来优化。这事并不漂亮，但你需要做你该做的。

下个问题是 `@test` 宏，它为每个测试用例生成一个 0 参数闭包。这不是必需的，因为每个用例只需运行一次。因此，`@test` 被改写以展开到一个 try-catch 块中，该块记录测试结果（true、false 或所引发的异常）并对它调用测试套件处理程序。

103.9 笛卡尔

The (non-exported) Cartesian module provides macros that facilitate writing multidimensional algorithms. Most often you can write such algorithms with straightforward techniques; however, there are a few cases where `Base.Cartesian` is still useful or necessary.

Principles of usage

A simple example of usage is:

```
@nloops 3 i A begin
    s += @nref 3 A i
end
```

which generates the following code:

```
for i_3 = axes(A, 3)
    for i_2 = axes(A, 2)
        for i_1 = axes(A, 1)
            s += A[i_1, i_2, i_3]
        end
    end
end
```

In general, Cartesian allows you to write generic code that contains repetitive elements, like the nested loops in this example. Other applications include repeated expressions (e.g., loop unwinding) or creating function calls with variable numbers of arguments without using the "splat" construct (`i...`).

基本语法

The (basic) syntax of `@nloops` is as follows:

- The first argument must be an integer (*not* a variable) specifying the number of loops.

- The second argument is the symbol-prefix used for the iterator variable. Here we used `i`, and variables `i_1, i_2, i_3` were generated.
- The third argument specifies the range for each iterator variable. If you use a variable (symbol) here, it's taken as `axes(A, dim)`. More flexibly, you can use the anonymous-function expression syntax described below.
- The last argument is the body of the loop. Here, that's what appears between the `begin...end`.

There are some additional features of `@nloops` described in the reference section.

`@nref` follows a similar pattern, generating `A[i_1,i_2,i_3]` from `@nref 3 A i`. The general practice is to read from left to right, which is why `@nloops` is `@nloops 3 i A expr` (as in `for i_2 = axes(A, 2)`, where `i_2` is to the left and the range is to the right) whereas `@nref` is `@nref 3 A i` (as in `A[i_1,i_2,i_3]`, where the array comes first).

If you're developing code with Cartesian, you may find that debugging is easier when you examine the generated code, using `@macroexpand`:

```
julia> @macroexpand @nref 2 A i
:(A[i_1, i_2])
```

Supplying the number of expressions

The first argument to both of these macros is the number of expressions, which must be an integer. When you're writing a function that you intend to work in multiple dimensions, this may not be something you want to hard-code. The recommended approach is to use a `@generated function`. Here's an example:

```
@generated function mysum(A::Array{T,N}) where {T,N}
    quote
        s = zero(T)
        @nloops $N i A begin
            s += @nref $N A i
        end
        s
    end
end
```

Naturally, you can also prepare expressions or perform calculations before the `quote` block.

Anonymous-function expressions as macro arguments

Perhaps the single most powerful feature in `Cartesian` is the ability to supply anonymous-function expressions that get evaluated at parsing time. Let's consider a simple example:

```
@nexprs 2 j->(i_j = 1)
```

`@nexprs` generates n expressions that follow a pattern. This code would generate the following statements:

```
i_1 = 1
i_2 = 1
```

In each generated statement, an "isolated" j (the variable of the anonymous function) gets replaced by values in the range 1:2. Generally speaking, Cartesian employs a LaTeX-like syntax. This allows you to do math on the index j. Here's an example computing the strides of an array:

```
s_1 = 1
@nexprs 3 j->(s_{j+1} = s_j * size(A, j))
```

would generate expressions

```
s_1 = 1
s_2 = s_1 * size(A, 1)
s_3 = s_2 * size(A, 2)
s_4 = s_3 * size(A, 3)
```

Anonymous-function expressions have many uses in practice.

Macro reference

Base.Cartesian.@nloops – Macro.

```
@nloops N itersym rangeexpr bodyexpr
@nloops N itersym rangeexpr preexpr bodyexpr
@nloops N itersym rangeexpr preexpr postexpr bodyexpr
```

Generate N nested loops, using itersym as the prefix for the iteration variables. rangeexpr may be an anonymous-function expression, or a simple symbol var in which case the range is axes(var, d) for dimension d.

Optionally, you can provide "pre" and "post" expressions. These get executed first and last, respectively, in the body of each loop. For example:

```
@nloops 2 i A d -> j_d = min(i_d, 5) begin
    s += @nref 2 A j
end
```

would generate:

```
for i_2 = axes(A, 2)
    j_2 = min(i_2, 5)
    for i_1 = axes(A, 1)
        j_1 = min(i_1, 5)
        s += A[j_1, j_2]
    end
end
```

If you want just a post-expression, supply nothing for the pre-expression. Using parentheses and semi-colons, you can supply multi-statement expressions.

source

`Base.Cartesian.@nref` – Macro.

```
@nref N A indexexpr
```

Generate expressions like `A[i_1, i_2, ...]`. `indexexpr` can either be an iteration-symbol prefix, or an anonymous-function expression.

Examples

```
julia> @macroexpand Base.Cartesian.@nref 3 A i
:(A[i_1, i_2, i_3])
```

source

`Base.Cartesian.@nextract` – Macro.

```
@nextract N esym isym
```

Generate `N` variables `esym_1`, `esym_2`, ..., `esym_N` to extract values from `isym`. `isym` can be either a `Symbol` or anonymous-function expression.

`@nextract 2 x y` would generate

```
x_1 = y[1]
x_2 = y[2]
```

while `@nextract 3 x d->y[2d-1]` yields

```
x_1 = y[1]
x_2 = y[3]
x_3 = y[5]
```

source

`Base.Cartesian.@nexprs` – Macro.

```
@nexprs N expr
```

Generate `N` expressions. `expr` should be an anonymous-function expression.

Examples

```
julia> @macroexpand Base.Cartesian.@nexprs 4 i -> y[i] = A[i+j]
quote
    y[1] = A[1 + j]
    y[2] = A[2 + j]
    y[3] = A[3 + j]
    y[4] = A[4 + j]
end
```

source

Base.Cartesian.@ncall – Macro.

```
@ncall N f sym...
```

Generate a function call expression. sym represents any number of function arguments, the last of which may be an anonymous-function expression and is expanded into N arguments.

For example, @ncall 3 func a generates

```
func(a_1, a_2, a_3)
```

while @ncall 2 func a b i->c[i] yields

```
func(a, b, c[1], c[2])
```

source

Base.Cartesian.@ntuple – Macro.

```
@ntuple N expr
```

Generates an N-tuple. @ntuple 2 i would generate (i_1, i_2), and @ntuple 2 k->k+1 would generate (2,3).

source

Base.Cartesian.@nall – Macro.

```
@nall N expr
```

Check whether all of the expressions generated by the anonymous-function expression expr evaluate to true.

@nall 3 d->(i_d > 1) would generate the expression (i_1 > 1 && i_2 > 1 && i_3 > 1). This can be convenient for bounds-checking.

source

Base.Cartesian.@nany – Macro.

```
@nany N expr
```

Check whether any of the expressions generated by the anonymous-function expression expr evaluate to true.

@nany 3 d->(i_d > 1) would generate the expression (i_1 > 1 || i_2 > 1 || i_3 > 1).

source

`Base.Cartesian.@nif` – Macro.

```
@nif N conditionexpr expr
@nif N conditionexpr expr elseexpr
```

Generates a sequence of `if ... elseif ... else ... end` statements. For example:

```
@nif 3 d->(i_d >= size(A,d)) d->(error("Dimension ", d, " too big")) d->println("All OK")
```

would generate:

```
if i_1 > size(A, 1)
    error("Dimension ", 1, " too big")
elseif i_2 > size(A, 2)
    error("Dimension ", 2, " too big")
else
    println("All OK")
end
```

source

103.10 Talking to the compiler (the `:meta` mechanism)

In some circumstances, one might wish to provide hints or instructions that a given block of code has special properties: you might always want to inline it, or you might want to turn on special compiler optimization passes. Starting with version 0.4, Julia has a convention that these instructions can be placed inside a `:meta` expression, which is typically (but not necessarily) the first expression in the body of a function.

`:meta` expressions are created with macros. As an example, consider the implementation of the `@inline` macro:

```
macro inline(ex)
    esc(isa(ex, Expr) ? pushmeta!(ex, :inline) : ex)
end
```

Here, ex is expected to be an expression defining a function. A statement like this:

```
@inline function myfunction(x)
    x*(x+3)
end
```

gets turned into an expression like this:

```
quote
    function myfunction(x)
        Expr(:meta, :inline)
        x*(x+3)
    end
end
```

`Base.pushmeta!(ex, :symbol, args...)` appends `:symbol` to the end of the `:meta` expression, creating a new `:meta` expression if necessary. If `args` is specified, a nested expression containing `:symbol` and these arguments is appended instead, which can be used to specify additional information.

To use the metadata, you have to parse these `:meta` expressions. If your implementation can be performed within Julia, `Base.popmeta!` is very handy: `Base.popmeta!(body, :symbol)` will scan a function *body* expression (one without the function signature) for the first `:meta` expression containing `:symbol`, extract any arguments, and return a tuple `(found::Bool, args::Array{Any})`. If the metadata did not have any arguments, or `:symbol` was not found, the `args` array will be empty.

Not yet provided is a convenient infrastructure for parsing `:meta` expressions from C++.

103.11 子数组

Julia 的 `SubArray` 类型是编码父类型 `AbstractArray` 的"视图"的一个容器。本页介绍了 `SubArray` 的一些设计原则和实现。

One of the major design goals is to ensure high performance for views of both `IndexLinear` and `IndexCartesian` arrays. Furthermore, views of `IndexLinear` arrays should themselves be `IndexLinear` to the extent that it is possible.

Index replacement

Consider making 2d slices of a 3d array:

```
julia> A = rand(2,3,4);

julia> S1 = view(A, :, 1, 2:3)
2×2 view(::Array{Float64, 3}, :, 1, 2:3) with eltype Float64:
 0.839622  0.711389
 0.967143  0.103929

julia> S2 = view(A, 1, :, 2:3)
3×2 view(::Array{Float64, 3}, 1, :, 2:3) with eltype Float64:
 0.839622  0.711389
 0.789764  0.806704
 0.566704  0.962715
```

`view` drops "singleton" dimensions (ones that are specified by an `Int`), so both S1 and S2 are two-dimensional `SubArray`s. Consequently, the natural way to index these is with `S1[i,j]`. To extract the value from the parent array A, the natural approach is to replace `S1[i,j]` with `A[i,1,(2:3)[j]]` and `S2[i,j]` with `A[1,i,(2:3)[j]]`.

The key feature of the design of SubArrays is that this index replacement can be performed without any runtime overhead.

SubArray design

Type parameters and fields

The strategy adopted is first and foremost expressed in the definition of the type:

```
struct SubArray{T,N,P,I,L} <: AbstractArray{T,N}
    parent::P
    indices::I
```

```
    offset1::Int       # for linear indexing and pointer, only valid when L==true
    stride1::Int       # used only for linear indexing
    ...
end
```

SubArray has 5 type parameters. The first two are the standard element type and dimensionality. The next is the type of the parent AbstractArray. The most heavily-used is the fourth parameter, a Tuple of the types of the indices for each dimension. The final one, L, is only provided as a convenience for dispatch; it's a boolean that represents whether the index types support fast linear indexing. More on that later.

If in our example above A is a Array{Float64, 3}, our S1 case above would be a SubArray{Float64,2,Array{Float64,3},Tuple
Note in particular the tuple parameter, which stores the types of the indices used to create S1. Likewise,

```
julia> S1.indices
(Base.Slice(Base.OneTo(2)), 1, 2:3)
```

Storing these values allows index replacement, and having the types encoded as parameters allows one to dispatch to efficient algorithms.

Index translation

Performing index translation requires that you do different things for different concrete SubArray types. For example, for S1, one needs to apply the i,j indices to the first and third dimensions of the parent array, whereas for S2 one needs to apply them to the second and third. The simplest approach to indexing would be to do the type-analysis at runtime:

```
parentindices = Vector{Any}()
for thisindex in S.indices
    ...
    if isa(thisindex, Int)
        # Don't consume one of the input indices
        push!(parentindices, thisindex)
    elseif isa(thisindex, AbstractVector)
        # Consume an input index
        push!(parentindices, thisindex[inputindex[j]])
        j += 1
    elseif isa(thisindex, AbstractMatrix)
        # Consume two input indices
        push!(parentindices, thisindex[inputindex[j], inputindex[j+1]])
        j += 2
    elseif ...
end
S.parent[parentindices...]
```

Unfortunately, this would be disastrous in terms of performance: each element access would allocate memory, and involves the running of a lot of poorly-typed code.

The better approach is to dispatch to specific methods to handle each type of stored index. That's what reindex does: it dispatches on the type of the first stored index and consumes the appropriate number of input indices, and then it recurses on the remaining indices. In the case of S1, this expands to

```
Base.reindex(S1, S1.indices, (i, j)) == (i, S1.indices[2], S1.indices[3][j])
```

for any pair of indices `(i,j)` (except `CartesianIndex`s and arrays thereof, see below).

This is the core of a `SubArray`; indexing methods depend upon `reindex` to do this index translation. Sometimes, though, we can avoid the indirection and make it even faster.

Linear indexing

Linear indexing can be implemented efficiently when the entire array has a single stride that separates successive elements, starting from some offset. This means that we can pre-compute these values and represent linear indexing simply as an addition and multiplication, avoiding the indirection of `reindex` and (more importantly) the slow computation of the cartesian coordinates entirely.

For `SubArray` types, the availability of efficient linear indexing is based purely on the types of the indices, and does not depend on values like the size of the parent array. You can ask whether a given set of indices supports fast linear indexing with the internal `Base.viewindexing` function:

```
julia> Base.viewindexing(S1.indices)
IndexCartesian()

julia> Base.viewindexing(S2.indices)
IndexLinear()
```

This is computed during construction of the `SubArray` and stored in the `L` type parameter as a boolean that encodes fast linear indexing support. While not strictly necessary, it means that we can define dispatch directly on `SubArray{T,N,A,I,true}` without any intermediaries.

Since this computation doesn't depend on runtime values, it can miss some cases in which the stride happens to be uniform:

```
julia> A = reshape(1:4*2, 4, 2)
4×2 reshape(::UnitRange{Int64}, 4, 2) with eltype Int64:
 1  5
 2  6
 3  7
 4  8

julia> diff(A[2:2:4,:][:])
3-element Vector{Int64}:
 2
 2
 2
```

A view constructed as `view(A, 2:2:4, :)` happens to have uniform stride, and therefore linear indexing indeed could be performed efficiently. However, success in this case depends on the size of the array: if the first dimension instead were odd,

```
julia> A = reshape(1:5*2, 5, 2)
5×2 reshape(::UnitRange{Int64}, 5, 2) with eltype Int64:
 1   6
 2   7
```

```
 3   8
 4   9
 5  10

julia> diff(A[2:2:4,:][:])
3-element Vector{Int64}:
 2
 3
 2
```

then `A[2:2:4,:]` does not have uniform stride, so we cannot guarantee efficient linear indexing. Since we have to base this decision based purely on types encoded in the parameters of the `SubArray`, `S = view(A, 2:2:4, :)` cannot implement efficient linear indexing.

A few details

- Note that the `Base.reindex` function is agnostic to the types of the input indices; it simply determines how and where the stored indices should be reindexed. It not only supports integer indices, but it supports non-scalar indexing, too. This means that views of views don't need two levels of indirection; they can simply re-compute the indices into the original parent array!
- Hopefully by now it's fairly clear that supporting slices means that the dimensionality, given by the parameter `N`, is not necessarily equal to the dimensionality of the parent array or the length of the `indices` tuple. Neither do user-supplied indices necessarily line up with entries in the `indices` tuple (e.g., the second user-supplied index might correspond to the third dimension of the parent array, and the third element in the `indices` tuple).

  What might be less obvious is that the dimensionality of the stored parent array must be equal to the number of effective indices in the `indices` tuple. Some examples:

  ```
  A = reshape(1:35, 5, 7) # A 2d parent Array
  S = view(A, 2:7)         # A 1d view created by linear indexing
  S = view(A, :, :, 1:1)   # Appending extra indices is supported
  ```

  Naively, you'd think you could just set `S.parent = A` and `S.indices = (:,:,1:1)`, but supporting this dramatically complicates the reindexing process, especially for views of views. Not only do you need to dispatch on the types of the stored indices, but you need to examine whether a given index is the final one and "merge" any remaining stored indices together. This is not an easy task, and even worse: it's slow since it implicitly depends upon linear indexing.

  Fortunately, this is precisely the computation that `ReshapedArray` performs, and it does so linearly if possible. Consequently, `view` ensures that the parent array is the appropriate dimensionality for the given indices by reshaping it if needed. The inner `SubArray` constructor ensures that this invariant is satisfied.
- `CartesianIndex` and arrays thereof throw a nasty wrench into the `reindex` scheme. Recall that `reindex` simply dispatches on the type of the stored indices in order to determine how many passed indices should be used and where they should go. But with `CartesianIndex`, there's no longer a one-to-one correspondence between the number of passed arguments and the number of dimensions that they index into. If we return to the above example of `Base.reindex(S1, S1.indices, (i, j))`, you can see that the expansion is incorrect for `i, j = CartesianIndex(), CartesianIndex(2,1)`. It should *skip* the `CartesianIndex()` entirely and return:

```
(CartesianIndex(2,1)[1], S1.indices[2], S1.indices[3][CartesianIndex(2,1)[2]])
```

Instead, though, we get:

```
(CartesianIndex(), S1.indices[2], S1.indices[3][CartesianIndex(2,1)])
```

Doing this correctly would require *combined* dispatch on both the stored and passed indices across all combinations of dimensionalities in an intractable manner. As such, `reindex` must never be called with `CartesianIndex` indices. Fortunately, the scalar case is easily handled by first flattening the `CartesianIndex` arguments to plain integers. Arrays of `CartesianIndex`, however, cannot be split apart into orthogonal pieces so easily. Before attempting to use `reindex`, `view` must ensure that there are no arrays of `CartesianIndex` in the argument list. If there are, it can simply "punt" by avoiding the `reindex` calculation entirely, constructing a nested `SubArray` with two levels of indirection instead.

103.12 isbits Union Optimizations

In Julia, the `Array` type holds both "bits" values as well as heap-allocated "boxed" values. The distinction is whether the value itself is stored inline (in the direct allocated memory of the array), or if the memory of the array is simply a collection of pointers to objects allocated elsewhere. In terms of performance, accessing values inline is clearly an advantage over having to follow a pointer to the actual value. The definition of "isbits" generally means any Julia type with a fixed, determinate size, meaning no "pointer" fields, see `?isbitstype`.

Julia also supports Union types, quite literally the union of a set of types. Custom Union type definitions can be extremely handy for applications wishing to "cut across" the nominal type system (i.e. explicit subtype relationships) and define methods or functionality on these, otherwise unrelated, set of types. A compiler challenge, however, is in determining how to treat these Union types. The naive approach (and indeed, what Julia itself did pre-0.7), is to simply make a "box" and then a pointer in the box to the actual value, similar to the previously mentioned "boxed" values. This is unfortunate, however, because of the number of small, primitive "bits" types (think `UInt8`, `Int32`, `Float64`, etc.) that would easily fit themselves inline in this "box" without needing any indirection for value access. There are two main ways Julia can take advantage of this optimization as of 0.7: isbits Union fields in types, and isbits Union Arrays.

isbits Union Structs

Julia now includes an optimization wherein "isbits Union" fields in types (`mutable struct`, `struct`, etc.) will be stored inline. This is accomplished by determining the "inline size" of the Union type (e.g. `Union{UInt8, Int16}` will have a size of two bytes, which represents the size needed of the largest Union type `Int16`), and in addition, allocating an extra "type tag byte" (`UInt8`), whose value signals the type of the actual value stored inline of the "Union bytes". The type tag byte value is the index of the actual value's type in the Union type's order of types. For example, a type tag value of `0x02` for a field with type `Union{Nothing, UInt8, Int16}` would indicate that an `Int16` value is stored in the 16 bits of the field in the structure's memory; a `0x01` value would indicate that a `UInt8` value was stored in the first 8 bits of the 16 bits of the field's memory. Lastly, a value of `0x00` signals that the `nothing` value will be returned for this field, even though, as a singleton type with a single type instance, it technically has a size of 0. The type tag byte for a type's Union field is stored directly after the field's computed Union memory.

isbits Union Arrays

Julia can now also store "isbits Union" values inline in an Array, as opposed to requiring an indirection box. The optimization is accomplished by storing an extra "type tag array" of bytes, one byte per array element, alongside the bytes of the actual array data. This type tag array serves the same function as the type field

case: its value signals the type of the actual stored Union value in the array. In terms of layout, a Julia Array can include extra "buffer" space before and after its actual data values, which are tracked in the `a->offset` and `a->maxsize` fields of the `jl_array_t*` type. The "type tag array" is treated exactly as another `jl_array_t*`, but which shares the same `a->offset`, `a->maxsize`, and `a->len` fields. So the formula to access an isbits Union Array's type tag bytes is `a->data + (a->maxsize - a->offset) * a->elsize + a->offset`; i.e. the Array's `a->data` pointer is already shifted by `a->offset`, so correcting for that, we follow the data all the way to the max of what it can hold `a->maxsize`, then adjust by `a->offset` more bytes to account for any present "front buffering" the array might be doing. This layout in particular allows for very efficient resizing operations as the type tag data only ever has to move when the actual array's data has to move.

103.13 System Image Building

Building the Julia system image

Julia ships with a preparsed system image containing the contents of the `Base` module, named `sys.ji`. This file is also precompiled into a shared library called `sys.{so,dll,dylib}` on as many platforms as possible, so as to give vastly improved startup times. On systems that do not ship with a precompiled system image file, one can be generated from the source files shipped in Julia's `DATAROOTDIR/julia/base` folder.

Julia will by default generate its system image on half of the available system threads. This may be controlled by the `JULIA_IMAGE_THREADS` environment variable.

This operation is useful for multiple reasons. A user may:

- Build a precompiled shared library system image on a platform that did not ship with one, thereby improving startup times.
- Modify `Base`, rebuild the system image and use the new `Base` next time Julia is started.
- Include a `userimg.jl` file that includes packages into the system image, thereby creating a system image that has packages embedded into the startup environment.

The `PackageCompiler.jl` package contains convenient wrapper functions to automate this process.

System image optimized for multiple microarchitectures

The system image can be compiled simultaneously for multiple CPU microarchitectures under the same instruction set architecture (ISA). Multiple versions of the same function may be created with minimum dispatch point inserted into shared functions in order to take advantage of different ISA extensions or other microarchitecture features. The version that offers the best performance will be selected automatically at runtime based on available CPU features.

Specifying multiple system image targets

A multi-microarchitecture system image can be enabled by passing multiple targets during system image compilation. This can be done either with the `JULIA_CPU_TARGET` make option or with the `-C` command line option when running the compilation command manually. Multiple targets are separated by `;` in the option string. The syntax for each target is a CPU name followed by multiple features separated by `,`. All features supported by LLVM are supported and a feature can be disabled with a `-` prefix. (`+` prefix is also allowed and ignored to be consistent with LLVM syntax). Additionally, a few special features are supported to control the function cloning behavior.

> **Note**
>
> It is good practice to specify either `clone_all` or `base(<n>)` for every target apart from the first one. This makes it explicit which targets have all functions cloned, and which targets are based on other targets. If this is not done, the default behavior is to not clone every function, and to use the first target's function definition as the fallback when not cloning a function.

1. `clone_all`

   By default, only functions that are the most likely to benefit from the microarchitecture features will be cloned. When `clone_all` is specified for a target, however, **all** functions in the system image will be cloned for the target. The negative form `-clone_all` can be used to prevent the built-in heuristic from cloning all functions.

2. `base(<n>)`

   Where <n> is a placeholder for a non-negative number (e.g. `base(0)`, `base(1)`). By default, a partially cloned (i.e. not `clone_all`) target will use functions from the default target (first one specified) if a function is not cloned. This behavior can be changed by specifying a different base with the `base(<n>)` option. The nth target (0-based) will be used as the base target instead of the default (`0`th) one. The base target has to be either `0` or another `clone_all` target. Specifying a non-`clone_all` target as the base target will cause an error.

3. `opt_size`

   This causes the function for the target to be optimized for size when there isn't a significant runtime performance impact. This corresponds to `-Os` GCC and Clang option.

4. `min_size`

   This causes the function for the target to be optimized for size that might have a significant runtime performance impact. This corresponds to `-Oz` Clang option.

As an example, at the time of this writing, the following string is used in the creation of the official x86_64 Julia binaries downloadable from julialang.org:

```
generic;sandybridge,-xsaveopt,clone_all;haswell,-rdrnd,base(1)
```

This creates a system image with three separate targets; one for a generic x86_64 processor, one with a `sandybridge` ISA (explicitly excluding `xsaveopt`) that explicitly clones all functions, and one targeting the `haswell` ISA, based off of the `sandybridge` sysimg version, and also excluding `rdrnd`. When a Julia implementation loads the generated sysimg, it will check the host processor for matching CPU capability flags, enabling the highest ISA level possible. Note that the base level (`generic`) requires the `cx16` instruction, which is disabled in some virtualization software and must be enabled for the `generic` target to be loaded. Alternatively, a sysimg could be generated with the target `generic,-cx16` for greater compatibility, however note that this may cause performance and stability problems in some code.

Implementation overview

This is a brief overview of different part involved in the implementation. See code comments for each components for more implementation details.

1. System image compilation

   The parsing and cloning decision are done in `src/processor*`. We currently support cloning of function based on the present of loops, simd instructions, or other math operations (e.g. fastmath, fma, muladd). This information is passed on to `src/llvm-multiversioning.cpp` which does the actual cloning. In addition to doing the cloning and insert dispatch slots (see comments in `MultiVersioning::runOnModule` for how this is done), the pass also generates metadata so that the runtime can load and initialize the system image correctly. A detailed description of the metadata is available in `src/processor.h`.

2. System image loading

   The loading and initialization of the system image is done in `src/processor*` by parsing the metadata saved during system image generation. Host feature detection and selection decision are done in `src/processor_*.cpp` depending on the ISA. The target selection will prefer exact CPU name match, larger vector register size, and larger number of features. An overview of this process is in `src/processor.cpp`.

103.14 Package Images

Julia package images provide object (native code) caches for Julia packages. They are similar to Julia's system image and support many of the same features. In fact the underlying serialization format is the same, and the system image is the base image that the package images are build against.

High-level overview

Package images are shared libraries that contain both code and data. Like `.ji` cache files, they are generated per package. The data section contains both global data (global variables in the package) as well as the necessary metadata about what methods and types are defined by the package. The code section contains native objects that cache the final output of Julia's LLVM-based compiler.

The command line option `--pkgimages=no` can be used to turn off object caching for this session. Note that this means that cache files have to likely be regenerated. See `JULIA_MAX_NUM_PRECOMPILE_FILES` for the upper limit of variants Julia caches per default.

> **Note**
>
> While the package images present themselves as native shared libraries, they are only an approximation thereof. You will not be able to link against them from a native program and they must be loaded from Julia.

Linking

Since the package images contain native code, we must run a linker over them before we can use them. You can set the environment variable `JULIA_VERBOSE_LINKING` to `true` to make the package image linking process verbose.

Furthermore, we cannot assume that the user has a working system linker installed. Therefore, Julia ships with LLD, the LLVM linker, to provide a working out of the box experience. In `base/linking.jl`, we implement a limited interface to be able to link package images on all supported platforms.

Quirks

Despite LLD being a multi-platform linker, it does not provide a consistent interface across platforms. Furthermore, it is meant to be used from `clang` or another compiler driver, we therefore reimplement some of the logic from `llvm-project/clang/lib/Driver/ToolChains`. Thankfully one can use `lld -flavor` to set lld to the right platform

Windows

To avoid having to deal with `link.exe` we use `-flavor gnu`, effectively turning `lld` into a cross-linker from a mingw32 environment. Windows DLLs are required to contain a `_DllMainCRTStartup` function and to minimize our dependence on mingw32 libraries, we inject a stub definition ourselves.

MacOS

Dynamic libraries on macOS need to link against `-lSystem`. On recent macOS versions, `-lSystem` is only available for linking when Xcode is available. To that effect we link with `-undefined dynamic_lookup`.

Package images optimized for multiple microarchitectures

Similar to multi-versioning for system images, package images support multi-versioning. If you are in a heterogenous environment, with a unified cache, you can set the environment variable `JULIA_CPU_TARGET=generic` to multi-version the object caches.

Flags that impact package image creation and selection

These are the Julia command line flags that impact cache selection. Package images that were created with different flags will be rejected.

- `-g`, `--debug-info`: Exact match required since it changes code generation.
- `--check-bounds`: Exact match required since it changes code generation.
- `--inline`: Exact match required since it changes code generation.
- `--pkgimages`: To allow running without object caching enabled.
- `-O`, `--optimize`: Reject package images generated for a lower optimization level, but allow for higher optimization levels to be loaded.

103.15 Working with LLVM

This is not a replacement for the LLVM documentation, but a collection of tips for working on LLVM for Julia.

Overview of Julia to LLVM Interface

Julia dynamically links against LLVM by default. Build with `USE_LLVM_SHLIB=0` to link statically.

The code for lowering Julia AST to LLVM IR or interpreting it directly is in directory `src/`.

Some of the `.cpp` files form a group that compile to a single object.

The difference between an intrinsic and a builtin is that a builtin is a first class function that can be used like any other Julia function. An intrinsic can operate only on unboxed data, and therefore its arguments must be statically typed.

Alias Analysis

Julia currently uses LLVM's Type Based Alias Analysis. To find the comments that document the inclusion relationships, look for `static MDNode*` in `src/codegen.cpp`.

The `-O` option enables LLVM's Basic Alias Analysis.

File	Description
aotcompile.cpp	Legacy pass manager pipeline, compiler C-interface entry
builtins.c	Builtin functions
ccall.cpp	Lowering ccall
cgutils.cpp	Lowering utilities, notably for array and tuple accesses
codegen.cpp	Top-level of code generation, pass list, lowering builtins
debuginfo.cpp	Tracks debug information for JIT code
disasm.cpp	Handles native object file and JIT code diassembly
gf.c	Generic functions
intrinsics.cpp	Lowering intrinsics
jitlayers.cpp	JIT-specific code, ORC compilation layers/utilities
llvm-alloc-helpers.cpp	Julia-specific escape analysis
llvm-alloc-opt.cpp	Custom LLVM pass to demote heap allocations to the stack
llvm-cpufeatures.cpp	Custom LLVM pass to lower CPU-based functions (e.g. haveFMA)
llvm-demote-float16.cpp	Custom LLVM pass to lower 16b float ops to 32b float ops
llvm-final-gc-lowering.cpp	Custom LLVM pass to lower GC calls to their final form
llvm-gc-invariant-verifier.cpp	Custom LLVM pass to verify Julia GC invariants
llvm-julia-licm.cpp	Custom LLVM pass to hoist/sink Julia-specific intrinsics
llvm-late-gc-lowering.cpp	Custom LLVM pass to root GC-tracked values
llvm-lower-handlers.cpp	Custom LLVM pass to lower try-catch blocks
llvm-muladd.cpp	Custom LLVM pass for fast-match FMA
llvm-multiversioning.cpp	Custom LLVM pass to generate sysimg code on multiple architectures
llvm-propagate-addrspaces.cpp	Custom LLVM pass to canonicalize addrspaces
llvm-ptls.cpp	Custom LLVM pass to lower TLS operations
llvm-remove-addrspaces.cpp	Custom LLVM pass to remove Julia addrspaces
llvm-remove-ni.cpp	Custom LLVM pass to remove Julia non-integral addrspaces
llvm-simdloop.cpp	Custom LLVM pass for @simd
pipeline.cpp	New pass manager pipeline, pass pipeline parsing
sys.c	I/O and operating system utility functions

Building Julia with a different version of LLVM

The default version of LLVM is specified in `deps/llvm.version`. You can override it by creating a file called `Make.user` in the top-level directory and adding a line to it such as:

```
LLVM_VER = 13.0.0
```

Besides the LLVM release numerals, you can also use `DEPS_GIT = llvm` in combination with `USE_BINARYBUILDER_LLVM = 0` to build against the latest development version of LLVM.

You can also specify to build a debug version of LLVM, by setting either `LLVM_DEBUG = 1` or `LLVM_DEBUG = Release` in your `Make.user` file. The former will be a fully unoptimized build of LLVM and the latter will produce an optimized build of LLVM. Depending on your needs the latter will suffice and it quite a bit faster. If you use `LLVM_DEBUG = Release` you will also want to set `LLVM_ASSERTIONS = 1` to enable diagnostics for different passes. Only `LLVM_DEBUG = 1` implies that option by default.

Passing options to LLVM

You can pass options to LLVM via the environment variable `JULIA_LLVM_ARGS`. Here are example settings using bash syntax:

- `export JULIA_LLVM_ARGS=-print-after-all` dumps IR after each pass.
- `export JULIA_LLVM_ARGS=-debug-only=loop-vectorize` dumps LLVM `DEBUG(...)` diagnostics for loop vectorizer. If you get warnings about "Unknown command line argument", rebuild LLVM with `LLVM_ASSERTIONS = 1`.
- `export JULIA_LLVM_ARGS=-help` shows a list of available options. `export JULIA_LLVM_ARGS=-help-hidden` shows even more.
- `export JULIA_LLVM_ARGS="-fatal-warnings -print-options"` is an example how to use multiple options.

Useful `JULIA_LLVM_ARGS` parameters

- `-print-after`=PASS: prints the IR after any execution of PASS, useful for checking changes done by a pass.
- `-print-before`=PASS: prints the IR before any execution of PASS, useful for checking the input to a pass.
- `-print-changed`: prints the IR whenever a pass changes the IR, useful for narrowing down which passes are causing problems.
- `-print-(before|after)`=MARKER-PASS: the Julia pipeline ships with a number of marker passes in the pipeline, which can be used to identify where problems or optimizations are occurring. A marker pass is defined as a pass which appears once in the pipeline and performs no transformations on the IR, and is only useful for targeting print-before/print-after. Currently, the following marker passes exist in the pipeline:
  - BeforeOptimization
  - BeforeEarlySimplification
  - AfterEarlySimplification
  - BeforeEarlyOptimization
  - AfterEarlyOptimization
  - BeforeLoopOptimization
  - BeforeLICM
  - AfterLICM
  - BeforeLoopSimplification
  - AfterLoopSimplification
  - AfterLoopOptimization
  - BeforeScalarOptimization
  - AfterScalarOptimization
  - BeforeVectorization
  - AfterVectorization
  - BeforeIntrinsicLowering
  - AfterIntrinsicLowering
  - BeforeCleanup
  - AfterCleanup
  - AfterOptimization

- `-time-passes`: prints the time spent in each pass, useful for identifying which passes are taking a long time.
- `-print-module-scope`: used in conjunction with `-print-(before|after)`, gets the entire module rather than the IR unit received by the pass
- `-debug`: prints out a lot of debugging information throughout LLVM
- `-debug-only=NAME`, prints out debugging statements from files with `DEBUG_TYPE` defined to `NAME`, useful for getting additional context about a problem

Debugging LLVM transformations in isolation

On occasion, it can be useful to debug LLVM's transformations in isolation from the rest of the Julia system, e.g. because reproducing the issue inside `julia` would take too long, or because one wants to take advantage of LLVM's tooling (e.g. bugpoint). To get unoptimized IR for the entire system image, pass the `--output-unopt-bc unopt.bc` option to the system image build process, which will output the unoptimized IR to an `unopt.bc` file. This file can then be passed to LLVM tools as usual. `libjulia` can function as an LLVM pass plugin and can be loaded into LLVM tools, to make julia-specific passes available in this environment. In addition, it exposes the `-julia` meta-pass, which runs the entire Julia pass-pipeline over the IR. As an example, to generate a system image with the old pass manager, one could do:

```
opt -enable-new-pm=0 -load libjulia-codegen.so -julia -o opt.bc unopt.bc
llc -o sys.o opt.bc
cc -shared -o sys.so sys.o
```

To generate a system image with the new pass manager, one could do:

```
opt -load-pass-plugin=libjulia-codegen.so --passes='julia' -o opt.bc unopt.bc
llc -o sys.o opt.bc
cc -shared -o sys.so sys.o
```

This system image can then be loaded by `julia` as usual.

It is also possible to dump an LLVM IR module for just one Julia function, using:

```
fun, T = +, Tuple{Int,Int} # Substitute your function of interest here
optimize = false
open("plus.ll", "w") do file
    println(file, InteractiveUtils._dump_function(fun, T, false, false, false, true, :att,
    ↪  optimize, :default))
end
```

These files can be processed the same way as the unoptimized sysimg IR shown above.

Improving LLVM optimizations for Julia

Improving LLVM code generation usually involves either changing Julia lowering to be more friendly to LLVM's passes, or improving a pass.

If you are planning to improve a pass, be sure to read the LLVM developer policy. The best strategy is to create a code example in a form where you can use LLVM's `opt` tool to study it and the pass of interest in isolation.

1. Create an example Julia code of interest.
2. Use `JULIA_LLVM_ARGS=-print-after-all` to dump the IR.
3. Pick out the IR at the point just before the pass of interest runs.
4. Strip the debug metadata and fix up the TBAA metadata by hand.

The last step is labor intensive. Suggestions on a better way would be appreciated.

The jlcall calling convention

Julia has a generic calling convention for unoptimized code, which looks somewhat as follows:

```
jl_value_t *any_unoptimized_call(jl_value_t *, jl_value_t **, int);
```

where the first argument is the boxed function object, the second argument is an on-stack array of arguments and the third is the number of arguments. Now, we could perform a straightforward lowering and emit an alloca for the argument array. However, this would betray the SSA nature of the uses at the call site, making optimizations (including GC root placement), significantly harder. Instead, we emit it as follows:

```
call %jl_value_t *@julia.call(jl_value_t *(*)(...) @any_unoptimized_call, %jl_value_t *%arg1,
↪  %jl_value_t *%arg2)
```

This allows us to retain the SSA-ness of the uses throughout the optimizer. GC root placement will later lower this call to the original C ABI.

GC root placement

GC root placement is done by an LLVM pass late in the pass pipeline. Doing GC root placement this late enables LLVM to make more aggressive optimizations around code that requires GC roots, as well as allowing us to reduce the number of required GC roots and GC root store operations (since LLVM doesn't understand our GC, it wouldn't otherwise know what it is and is not allowed to do with values stored to the GC frame, so it'll conservatively do very little). As an example, consider an error path

```
if some_condition()
    #= Use some variables maybe =#
    error("An error occurred")
end
```

During constant folding, LLVM may discover that the condition is always false, and can remove the basic block. However, if GC root lowering is done early, the GC root slots used in the deleted block, as well as any values kept alive in those slots only because they were used in the error path, would be kept alive by LLVM. By doing GC root lowering late, we give LLVM the license to do any of its usual optimizations (constant folding, dead code elimination, etc.), without having to worry (too much) about which values may or may not be GC tracked.

However, in order to be able to do late GC root placement, we need to be able to identify a) which pointers are GC tracked and b) all uses of such pointers. The goal of the GC placement pass is thus simple:

Minimize the number of needed GC roots/stores to them subject to the constraint that at every safepoint, any live GC-tracked pointer (i.e. for which there is a path after this point that contains a use of this pointer) is in some GC slot.

Representation

The primary difficulty is thus choosing an IR representation that allows us to identify GC-tracked pointers and their uses, even after the program has been run through the optimizer. Our design makes use of three LLVM features to achieve this:

- Custom address spaces
- Operand Bundles
- Non-integral pointers

Custom address spaces allow us to tag every point with an integer that needs to be preserved through optimizations. The compiler may not insert casts between address spaces that did not exist in the original program and it must never change the address space of a pointer on a load/store/etc operation. This allows us to annotate which pointers are GC-tracked in an optimizer-resistant way. Note that metadata would not be able to achieve the same purpose. Metadata is supposed to always be discardable without altering the semantics of the program. However, failing to identify a GC-tracked pointer alters the resulting program behavior dramatically - it'll probably crash or return wrong results. We currently use three different address spaces (their numbers are defined in `src/codegen_shared.cpp`):

- GC Tracked Pointers (currently 10): These are pointers to boxed values that may be put into a GC frame. It is loosely equivalent to a `jl_value_t*` pointer on the C side. N.B. It is illegal to ever have a pointer in this address space that may not be stored to a GC slot.
- Derived Pointers (currently 11): These are pointers that are derived from some GC tracked pointer. Uses of these pointers generate uses of the original pointer. However, they need not themselves be known to the GC. The GC root placement pass MUST always find the GC tracked pointer from which this pointer is derived and use that as the pointer to root.
- Callee Rooted Pointers (currently 12): This is a utility address space to express the notion of a callee rooted value. All values of this address space MUST be storable to a GC root (though it is possible to relax this condition in the future), but unlike the other pointers need not be rooted if passed to a call (they do still need to be rooted if they are live across another safepoint between the definition and the call).
- Pointers loaded from tracked object (currently 13): This is used by arrays, which themselves contain a pointer to the managed data. This data area is owned by the array, but is not a GC-tracked object by itself. The compiler guarantees that as long as this pointer is live, the object that this pointer was loaded from will keep being live.

Invariants

The GC root placement pass makes use of several invariants, which need to be observed by the frontend and are preserved by the optimizer.

First, only the following address space casts are allowed:

- 0->{Tracked,Derived,CalleeRooted}: It is allowable to decay an untracked pointer to any of the others. However, do note that the optimizer has broad license to not root such a value. It is never safe to have a value in address space 0 in any part of the program if it is (or is derived from) a value that requires a GC root.

- Tracked->Derived: This is the standard decay route for interior values. The placement pass will look for these to identify the base pointer for any use.
- Tracked->CalleeRooted: Addrspace CalleeRooted serves merely as a hint that a GC root is not required. However, do note that the Derived->CalleeRooted decay is prohibited, since pointers should generally be storable to a GC slot, even in this address space.

Now let us consider what constitutes a use:

- Loads whose loaded values is in one of the address spaces
- Stores of a value in one of the address spaces to a location
- Stores to a pointer in one of the address spaces
- Calls for which a value in one of the address spaces is an operand
- Calls in jlcall ABI, for which the argument array contains a value
- Return instructions.

We explicitly allow load/stores and simple calls in address spaces Tracked/Derived. Elements of jlcall argument arrays must always be in address space Tracked (it is required by the ABI that they are valid `jl_value_t*` pointers). The same is true for return instructions (though note that struct return arguments are allowed to have any of the address spaces). The only allowable use of an address space CalleeRooted pointer is to pass it to a call (which must have an appropriately typed operand).

Further, we disallow `getelementptr` in addrspace Tracked. This is because unless the operation is a noop, the resulting pointer will not be validly storable to a GC slot and may thus not be in this address space. If such a pointer is required, it should be decayed to addrspace Derived first.

Lastly, we disallow `inttoptr`/`ptrtoint` instructions in these address spaces. Having these instructions would mean that some `i64` values are really GC tracked. This is problematic, because it breaks that stated requirement that we're able to identify GC-relevant pointers. This invariant is accomplished using the LLVM "non-integral pointers" feature, which is new in LLVM 5.0. It prohibits the optimizer from making optimizations that would introduce these operations. Note we can still insert static constants at JIT time by using `inttoptr` in address space 0 and then decaying to the appropriate address space afterwards.

Supporting `ccall`

One important aspect missing from the discussion so far is the handling of `ccall`. `ccall` has the peculiar feature that the location and scope of a use do not coincide. As an example consider:

```
A = randn(1024)
ccall(:foo, Cvoid, (Ptr{Float64},), A)
```

In lowering, the compiler will insert a conversion from the array to the pointer which drops the reference to the array value. However, we of course need to make sure that the array does stay alive while we're doing the `ccall`. To understand how this is done, first recall the lowering of the above code:

```
return $(Expr(:foreigncall, :(:foo), Cvoid, svec(Ptr{Float64}), 0, :(:ccall), Expr(:foreigncall,
↪  :(:jl_array_ptr), Ptr{Float64}, svec(Any), 0, :(:ccall), :(A)), :(A)))
```

The last `:(A)`, is an extra argument list inserted during lowering that informs the code generator which Julia level values need to be kept alive for the duration of this `ccall`. We then take this information and represent it in an "operand bundle" at the IR level. An operand bundle is essentially a fake use that is attached to the call site. At the IR level, this looks like so:

```
call void inttoptr (i64 ... to void (double*)*)(double* %5) [ "jl_roots"(%jl_value_t addrspace(10)*
↪  %A) ]
```

The GC root placement pass will treat the `jl_roots` operand bundle as if it were a regular operand. However, as a final step, after the GC roots are inserted, it will drop the operand bundle to avoid confusing instruction selection.

Supporting `pointer_from_objref`

`pointer_from_objref` is special because it requires the user to take explicit control of GC rooting. By our above invariants, this function is illegal, because it performs an address space cast from 10 to 0. However, it can be useful, in certain situations, so we provide a special intrinsic:

```
declared %jl_value_t *julia.pointer_from_objref(%jl_value_t addrspace(10)*)
```

which is lowered to the corresponding address space cast after GC root lowering. Do note however that by using this intrinsic, the caller assumes all responsibility for making sure that the value in question is rooted. Further this intrinsic is not considered a use, so the GC root placement pass will not provide a GC root for the function. As a result, the external rooting must be arranged while the value is still tracked by the system. I.e. it is not valid to attempt to use the result of this operation to establish a global root - the optimizer may have already dropped the value.

Keeping values alive in the absence of uses

In certain cases it is necessary to keep an object alive, even though there is no compiler-visible use of said object. This may be case for low level code that operates on the memory-representation of an object directly or code that needs to interface with C code. In order to allow this, we provide the following intrinsics at the LLVM level:

```
token @llvm.julia.gc_preserve_begin(...)
void @llvm.julia.gc_preserve_end(token)
```

(The `llvm.` in the name is required in order to be able to use the `token` type). The semantics of these intrinsics are as follows: At any safepoint that is dominated by a `gc_preserve_begin` call, but that is not not dominated by a corresponding `gc_preserve_end` call (i.e. a call whose argument is the token returned by a `gc_preserve_begin` call), the values passed as arguments to that `gc_preserve_begin` will be kept live. Note that the `gc_preserve_begin` still counts as a regular use of those values, so the standard lifetime semantics will ensure that the values will be kept alive before entering the preserve region.

103.16 printf() and stdio in the Julia runtime

Libuv wrappers for stdio

`julia.h` defines libuv wrappers for the `stdio.h` streams:

```
uv_stream_t *JL_STDIN;
uv_stream_t *JL_STDOUT;
uv_stream_t *JL_STDERR;
```

... and corresponding output functions:

```
int jl_printf(uv_stream_t *s, const char *format, ...);
int jl_vprintf(uv_stream_t *s, const char *format, va_list args);
```

These `printf` functions are used by the `.c` files in the `src/` and `cli/` directories wherever stdio is needed to ensure that output buffering is handled in a unified way.

In special cases, like signal handlers, where the full libuv infrastructure is too heavy, `jl_safe_printf()` can be used to `write(2)` directly to STDERR_FILENO:

```
void jl_safe_printf(const char *str, ...);
```

Interface between JL_STD* and Julia code

`Base.stdin`, `Base.stdout` and `Base.stderr` are bound to the JL_STD* libuv streams defined in the runtime.

Julia's `__init__()` function (in `base/sysimg.jl`) calls `reinit_stdio()` (in `base/stream.jl`) to create Julia objects for `Base.stdin`, `Base.stdout` and `Base.stderr`.

`reinit_stdio()` uses `ccall` to retrieve pointers to JL_STD* and calls `jl_uv_handle_type()` to inspect the type of each stream. It then creates a Julia `Base.IOStream`, `Base.TTY` or `Base.PipeEndpoint` object to represent each stream, e.g.:

```
$ julia -e 'println(typeof((stdin, stdout, stderr)))'
Tuple{Base.TTY,Base.TTY,Base.TTY}

$ julia -e 'println(typeof((stdin, stdout, stderr)))' < /dev/null 2>/dev/null
Tuple{IOStream,Base.TTY,IOStream}

$ echo hello | julia -e 'println(typeof((stdin, stdout, stderr)))' | cat
Tuple{Base.PipeEndpoint,Base.PipeEndpoint,Base.TTY}
```

The `Base.read` and `Base.write` methods for these streams use `ccall` to call libuv wrappers in `src/jl_uv.c`, e.g.:

```
stream.jl: function write(s::IO, p::Ptr, nb::Integer)
               -> ccall(:jl_uv_write, ...)
  jl_uv.c:          -> int jl_uv_write(uv_stream_t *stream, ...)
                        -> uv_write(uvw, stream, buf, ...)
```

printf() during initialization

The libuv streams relied upon by `jl_printf()` etc., are not available until midway through initialization of the runtime (see `init.c`, `init_stdio()`). Error messages or warnings that need to be printed before this are routed to the standard C library `fwrite()` function by the following mechanism:

In sys.c, the JL_STD* stream pointers are statically initialized to integer constants: STD*_FILENO (0, 1 and 2). In jl_uv.c the jl_uv_puts() function checks its uv_stream_t* stream argument and calls fwrite() if stream is set to STDOUT_FILENO or STDERR_FILENO.

This allows for uniform use of jl_printf() throughout the runtime regardless of whether or not any particular piece of code is reachable before initialization is complete.

Legacy ios.c library

The src/support/ios.c library is inherited from femtolisp. It provides cross-platform buffered file IO and in-memory temporary buffers.

ios.c is still used by:

- src/flisp/*.c
- src/dump.c –for serialization file IO and for memory buffers.
- src/staticdata.c –for serialization file IO and for memory buffers.
- base/iostream.jl –for file IO (see base/fs.jl for libuv equivalent).

Use of ios.c in these modules is mostly self-contained and separated from the libuv I/O system. However, there is one place where femtolisp calls through to jl_printf() with a legacy ios_t stream.

There is a hack in ios.h that makes the ios_t.bm field line up with the uv_stream_t.type and ensures that the values used for ios_t.bm to not overlap with valid UV_HANDLE_TYPE values. This allows uv_stream_t pointers to point to ios_t streams.

This is needed because jl_printf() caller jl_static_show() is passed an ios_t stream by femtolisp's fl_print() function. Julia's jl_uv_puts() function has special handling for this:

```
if (stream->type > UV_HANDLE_TYPE_MAX) {
    return ios_write((ios_t*)stream, str, n);
}
```

103.17 边界检查

和许多其他现代编程语言一样，Julia 在访问数组元素的时候也要通过边界检查来确保程序安全。当循环次数很多，或者在其他性能敏感的场景下，你可能希望不进行边界检查以提高运行时性能。比如要使用矢量 (SIMD) 指令，循环体就不能有分支语句，因此无法进行边界检查。Julia 提供了一个宏 @inbounds(...) 来告诉编译器在指定语句块不进行边界检查。用户自定义的数组类型可以通过宏 @boundscheck(...) 来达到上下文敏感的代码选择目的。

移除边界检查

宏 @boundscheck(...) 把代码块标记为要执行边界检查。但当这些代码块被被宏 @inbounds(...) 标记的代码包裹时，它们可能会被编译器移除。仅当 @boundscheck(...) 代码块被调用函数包裹时，编译器会移除它们。比如你可能这样写的 sum 方法：

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i in eachindex(A)
```

```
        @inbounds r += A[i]
    end
    return r
end
```

使用自定义的类数组类型 `MyArray`，我们有：

```
@inline getindex(A::MyArray, i::Real) = (@boundscheck checkbounds(A, i); A.data[to_index(i)])
```

当 `getindex` 内联到 `sum` 时，对 `checkbounds(A, i)` 的调用将被忽略。如果函数包含多层内联，那么只有最深一层内联的 `@boundscheck` 块才会被忽略。该规则可防止堆栈上层的代码对程序行为造成意外改变。

Caution!

It is easy to accidentally expose unsafe operations with `@inbounds`. You might be tempted to write the above example as

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i in 1:length(A)
        @inbounds r += A[i]
    end
    return r
end
```

Which quietly assumes 1-based indexing and therefore exposes unsafe memory access when used with `OffsetArrays`:

```
julia> using OffsetArrays

julia> sum(OffsetArray([1, 2, 3], -10))
9164911648 # inconsistent results or segfault
```

While the original source of the error here is `1:length(A)`, the use of `@inbounds` increases the consequences from a bounds error to a less easily caught and debugged unsafe memory access. It is often difficult or impossible to prove that a method which uses `@inbounds` is safe, so one must weigh the benefits of performance improvements against the risk of segfaults and silent misbehavior, especially in public facing APIs.

Propagating inbounds

There may be certain scenarios where for code-organization reasons you want more than one layer between the `@inbounds` and `@boundscheck` declarations. For instance, the default `getindex` methods have the chain `getindex(A::AbstractArray, i::Real)` calls `getindex(IndexStyle(A), A, i)` calls `_getindex(::IndexLinear, A, i)`.

To override the "one layer of inlining" rule, a function may be marked with `Base.@propagate_inbounds` to propagate an inbounds context (or out of bounds context) through one additional layer of inlining.

The bounds checking call hierarchy

The overall hierarchy is:

- `checkbounds(A, I...)` which calls
  - `checkbounds(Bool, A, I...)` which calls
    - `checkbounds_indices(Bool, axes(A), I)` which recursively calls
      - `checkindex` for each dimension

Here `A` is the array, and `I` contains the "requested" indices. `axes(A)` returns a tuple of "permitted" indices of `A`.

`checkbounds(A, I...)` throws an error if the indices are invalid, whereas `checkbounds(Bool, A, I...)` returns `false` in that circumstance. `checkbounds_indices` discards any information about the array other than its `axes` tuple, and performs a pure indices-vs-indices comparison: this allows relatively few compiled methods to serve a huge variety of array types. Indices are specified as tuples, and are usually compared in a 1-1 fashion with individual dimensions handled by calling another important function, `checkindex`: typically,

```
checkbounds_indices(Bool, (IA1, IA...), (I1, I...)) = checkindex(Bool, IA1, I1) &
                                                      checkbounds_indices(Bool, IA, I)
```

so `checkindex` checks a single dimension. All of these functions, including the unexported `checkbounds_indices` have docstrings accessible with `?` .

If you have to customize bounds checking for a specific array type, you should specialize `checkbounds(Bool, A, I...)`. However, in most cases you should be able to rely on `checkbounds_indices` as long as you supply useful `axes` for your array type.

If you have novel index types, first consider specializing `checkindex`, which handles a single index for a particular dimension of an array. If you have a custom multidimensional index type (similar to `CartesianIndex`), then you may have to consider specializing `checkbounds_indices`.

Note this hierarchy has been designed to reduce the likelihood of method ambiguities. We try to make `checkbounds` the place to specialize on array type, and try to avoid specializations on index types; conversely, `checkindex` is intended to be specialized only on index type (especially, the last argument).

Emit bounds checks

Julia can be launched with `--check-bounds={yes|no|auto}` to emit bounds checks always, never, or respect `@inbounds` declarations.

103.18 Proper maintenance and care of multi-threading locks

The following strategies are used to ensure that the code is dead-lock free (generally by addressing the 4th Coffman condition: circular wait).

1. structure code such that only one lock will need to be acquired at a time
2. always acquire shared locks in the same order, as given by the table below
3. avoid constructs that expect to need unrestricted recursion

Locks

Below are all of the locks that exist in the system and the mechanisms for using them that avoid the potential for deadlocks (no Ostrich algorithm allowed here):

The following are definitely leaf locks (level 1), and must not try to acquire any other lock:

- safepoint

  > Note that this lock is acquired implicitly by `JL_LOCK` and `JL_UNLOCK`. use the `_NOGC` variants to avoid that for level 1 locks.
  > While holding this lock, the code must not do any allocation or hit any safepoints. Note that there are safepoints when doing allocation, enabling / disabling GC, entering / restoring exception frames, and taking / releasing locks.

- shared_map
- finalizers
- pagealloc
- gc*perm*lock
- flisp
- jl*in*stackwalk (Win32)
- ResourcePool<?>::mutex
- RLST_mutex
- jl*locked*stream::mutex
- debuginfo_asyncsafe
- inference*timing*mutex
- ExecutionEngine::SessionLock

  > flisp itself is already threadsafe, this lock only protects the `jl_ast_context_list_t` pool likewise, the ResourcePool<?>::mutexes just protect the associated resource pool

The following is a leaf lock (level 2), and only acquires level 1 locks (safepoint) internally:

- typecache
- Module->lock
- JLDebuginfoPlugin::PluginMutex
- newly*inferred*mutex

The following is a level 3 lock, which can only acquire level 1 or level 2 locks internally:

- Method->writelock

The following is a level 4 lock, which can only recurse to acquire level 1, 2, or 3 locks:

- MethodTable->writelock

No Julia code may be called while holding a lock above this point.

orc::ThreadSafeContext (TSCtx) locks occupy a special spot in the locking hierarchy. They are used to protect LLVM's global non-threadsafe state, but there may be an arbitrary number of them. By default, all of these locks may be treated as level 5 locks for the purposes of comparing with the rest of the hierarchy. Acquiring a TSCtx should only be done from the JIT's pool of TSCtx's, and all locks on that TSCtx should be released prior to returning it to the pool. If multiple TSCtx locks must be acquired at the same time (due to recursive compilation), then locks should be acquired in the order that the TSCtxs were borrowed from the pool.

The following is a level 5 lock

- JuliaOJIT::EmissionMutex

The following are a level 6 lock, which can only recurse to acquire locks at lower levels:

- codegen
- jl*modules*mutex

The following is an almost root lock (level end-1), meaning only the root look may be held when trying to acquire it:

- typeinf

  this one is perhaps one of the most tricky ones, since type-inference can be invoked from many points
  currently the lock is merged with the codegen lock, since they call each other recursively

The following lock synchronizes IO operation. Be aware that doing any I/O (for example, printing warning messages or debug information) while holding any other lock listed above may result in pernicious and hard-to-find deadlocks. BE VERY CAREFUL!

- iolock
- Individual ThreadSynchronizers locks

  this may continue to be held after releasing the iolock, or acquired without it, but be very careful to never attempt to acquire the iolock while holding it

The following is the root lock, meaning no other lock shall be held when trying to acquire it:

- toplevel

  this should be held while attempting a top-level action (such as making a new type or defining a new method): trying to obtain this lock inside a staged function will cause a deadlock condition!
  additionally, it's unclear if *any* code can safely run in parallel with an arbitrary toplevel expression, so it may require all threads to get to a safepoint first

Broken Locks

The following locks are broken:

- toplevel

  doesn't exist right now

  fix: create it

- Module->lock

  This is vulnerable to deadlocks since it can't be certain it is acquired in sequence. Some operations (such as `import_module`) are missing a lock.

  fix: replace with `jl_modules_mutex`?

- loading.jl: `require` and `register_root_module`

  This file potentially has numerous problems.

  fix: needs locks

Shared Global Data Structures

These data structures each need locks due to being shared mutable global state. It is the inverse list for the above lock priority list. This list does not include level 1 leaf resources due to their simplicity.

MethodTable modifications (def, cache) : MethodTable->writelock

Type declarations : toplevel lock

Type application : typecache lock

Global variable tables : Module->lock

Module serializer : toplevel lock

JIT & type-inference : codegen lock

MethodInstance/CodeInstance updates : Method->writelock, codegen lock

- These are set at construction and immutable:
  - specTypes
  - sparam_vals
  - def

- These are set by `jl_type_infer` (while holding codegen lock):
  - cache
  - rettype
  - inferred

```
    * valid ages
```

- `inInference` flag:

  - optimization to quickly avoid recurring into `jl_type_infer` while it is already running
  - actual state (of setting `inferred`, then `fptr`) is protected by codegen lock

- Function pointers:
  - these transition once, from `NULL` to a value, while the codegen lock is held
- Code-generator cache (the contents of `functionObjectsDecls`):
  - these can transition multiple times, but only while the codegen lock is held
  - it is valid to use old version of this, or block for new versions of this, so races are benign, as long as the code is careful not to reference other data in the method instance (such as `rettype`) and assume it is coordinated, unless also holding the codegen lock

LLVMContext : codegen lock

Method : Method->writelock

- roots array (serializer and codegen)
- invoke / specializations / tfunc modifications

103.19 Arrays with custom indices

Conventionally, Julia's arrays are indexed starting at 1, whereas some other languages start numbering at 0, and yet others (e.g., Fortran) allow you to specify arbitrary starting indices. While there is much merit in picking a standard (i.e., 1 for Julia), there are some algorithms which simplify considerably if you can index outside the range `1:size(A,d)` (and not just `0:size(A,d)-1`, either). To facilitate such computations, Julia supports arrays with arbitrary indices.

The purpose of this page is to address the question, "what do I have to do to support such arrays in my own code?" First, let's address the simplest case: if you know that your code will never need to handle arrays with unconventional indexing, hopefully the answer is "nothing." Old code, on conventional arrays, should function essentially without alteration as long as it was using the exported interfaces of Julia. If you find it more convenient to just force your users to supply traditional arrays where indexing starts at one, you can add

```
Base.require_one_based_indexing(arrays...)
```

where `arrays...` is a list of the array objects that you wish to check for anything that violates 1-based indexing.

Generalizing existing code

As an overview, the steps are:

- replace many uses of `size` with `axes`
- replace `1:length(A)` with `eachindex(A)`, or in some cases `LinearIndices(A)`
- replace explicit allocations like `Array{Int}(undef, size(B))` with `similar(Array{Int}, axes(B))`

These are described in more detail below.

Things to watch out for

Because unconventional indexing breaks many people's assumptions that all arrays start indexing with 1, there is always the chance that using such arrays will trigger errors. The most frustrating bugs would be incorrect results or segfaults (total crashes of Julia). For example, consider the following function:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    length(dest) == length(src) || throw(DimensionMismatch("vectors must match"))
    # OK, now we're safe to use @inbounds, right? (not anymore!)
    for i = 1:length(src)
        @inbounds dest[i] = src[i]
    end
    dest
end
```

This code implicitly assumes that vectors are indexed from 1; if `dest` starts at a different index than `src`, there is a chance that this code would trigger a segfault. (If you do get segfaults, to help locate the cause try running julia with the option `--check-bounds=yes`.)

Using axes for bounds checks and loop iteration

`axes(A)` (reminiscent of `size(A)`) returns a tuple of `AbstractUnitRange{<:Integer}` objects, specifying the range of valid indices along each dimension of `A`. When `A` has unconventional indexing, the ranges may not start at 1. If you just want the range for a particular dimension `d`, there is `axes(A, d)`.

Base implements a custom range type, `OneTo`, where `OneTo(n)` means the same thing as `1:n` but in a form that guarantees (via the type system) that the lower index is 1. For any new `AbstractArray` type, this is the default returned by `axes`, and it indicates that this array type uses "conventional" 1-based indexing.

For bounds checking, note that there are dedicated functions `checkbounds` and `checkindex` which can sometimes simplify such tests.

Linear indexing (`LinearIndices`)

Some algorithms are most conveniently (or efficiently) written in terms of a single linear index, `A[i]` even if `A` is multi-dimensional. Regardless of the array's native indices, linear indices always range from `1:length(A)`. However, this raises an ambiguity for one-dimensional arrays (a.k.a., `AbstractVector`): does `v[i]` mean linear indexing , or Cartesian indexing with the array's native indices?

For this reason, your best option may be to iterate over the array with `eachindex(A)`, or, if you require the indices to be sequential integers, to get the index range by calling `LinearIndices(A)`. This will return `axes(A, 1)` if A is an AbstractVector, and the equivalent of `1:length(A)` otherwise.

By this definition, 1-dimensional arrays always use Cartesian indexing with the array's native indices. To help enforce this, it's worth noting that the index conversion functions will throw an error if shape indicates a 1-dimensional array with unconventional indexing (i.e., is a `Tuple{UnitRange}` rather than a tuple of `OneTo`). For arrays with conventional indexing, these functions continue to work the same as always.

Using `axes` and `LinearIndices`, here is one way you could rewrite `mycopy!`:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    axes(dest) == axes(src) || throw(DimensionMismatch("vectors must match"))
    for i in LinearIndices(src)
        @inbounds dest[i] = src[i]
    end
```

```
    dest
end
```

Allocating storage using generalizations of similar

Storage is often allocated with `Array{Int}(undef, dims)` or `similar(A, args...)`. When the result needs to match the indices of some other array, this may not always suffice. The generic replacement for such patterns is to use `similar(storagetype, shape)`. `storagetype` indicates the kind of underlying "conventional" behavior you'd like, e.g., `Array{Int}` or `BitArray` or even `dims->zeros(Float32, dims)` (which would allocate an all-zeros array). `shape` is a tuple of `Integer` or `AbstractUnitRange` values, specifying the indices that you want the result to use. Note that a convenient way of producing an all-zeros array that matches the indices of A is simply `zeros(A)`.

Let's walk through a couple of explicit examples. First, if `A` has conventional indices, then `similar(Array{Int}, axes(A))` would end up calling `Array{Int}(undef, size(A))`, and thus return an array. If `A` is an `AbstractArray` type with unconventional indexing, then `similar(Array{Int}, axes(A))` should return something that "behaves like" an `Array{Int}` but with a shape (including indices) that matches `A`. (The most obvious implementation is to allocate an `Array{Int}(undef, size(A))` and then "wrap" it in a type that shifts the indices.)

Note also that `similar(Array{Int}, (axes(A, 2),))` would allocate an `AbstractVector{Int}` (i.e., 1-dimensional array) that matches the indices of the columns of `A`.

Writing custom array types with non-1 indexing

Most of the methods you'll need to define are standard for any `AbstractArray` type, see Abstract Arrays. This page focuses on the steps needed to define unconventional indexing.

Custom AbstractUnitRange types

If you're writing a non-1 indexed array type, you will want to specialize `axes` so it returns a `UnitRange`, or (perhaps better) a custom `AbstractUnitRange`. The advantage of a custom type is that it "signals" the allocation type for functions like `similar`. If we're writing an array type for which indexing will start at 0, we likely want to begin by creating a new `AbstractUnitRange`, `ZeroRange`, where `ZeroRange(n)` is equivalent to `0:n-1`.

In general, you should probably *not* export `ZeroRange` from your package: there may be other packages that implement their own `ZeroRange`, and having multiple distinct `ZeroRange` types is (perhaps counterintuitively) an advantage: `ModuleA.ZeroRange` indicates that `similar` should create a `ModuleA.ZeroArray`, whereas `ModuleB.ZeroRange` indicates a `ModuleB.ZeroArray` type. This design allows peaceful coexistence among many different custom array types.

Note that the Julia package CustomUnitRanges.jl can sometimes be used to avoid the need to write your own `ZeroRange` type.

Specializing axes

Once you have your `AbstractUnitRange` type, then specialize `axes`:

```
Base.axes(A::ZeroArray) = map(n->ZeroRange(n), A.size)
```

where here we imagine that `ZeroArray` has a field called `size` (there would be other ways to implement this).

In some cases, the fallback definition for `axes(A, d)`:

```
axes(A::AbstractArray{T,N}, d) where {T,N} = d <= N ? axes(A)[d] : OneTo(1)
```

may not be what you want: you may need to specialize it to return something other than `OneTo(1)` when `d > ndims(A)`. Likewise, in `Base` there is a dedicated function `axes1` which is equivalent to `axes(A, 1)` but which avoids checking (at runtime) whether `ndims(A) > 0`. (This is purely a performance optimization.) It is defined as:

```
axes1(A::AbstractArray{T,0}) where {T} = OneTo(1)
axes1(A::AbstractArray) = axes(A)[1]
```

If the first of these (the zero-dimensional case) is problematic for your custom array type, be sure to specialize it appropriately.

Specializing `similar`

Given your custom `ZeroRange` type, then you should also add the following two specializations for `similar`:

```
function Base.similar(A::AbstractArray, T::Type, shape::Tuple{ZeroRange,Vararg{ZeroRange}})
    # body
end

function Base.similar(f::Union{Function,DataType}, shape::Tuple{ZeroRange,Vararg{ZeroRange}})
    # body
end
```

Both of these should allocate your custom array type.

Specializing `reshape`

Optionally, define a method

```
Base.reshape(A::AbstractArray, shape::Tuple{ZeroRange,Vararg{ZeroRange}}) = ...
```

and you can `reshape` an array so that the result has custom indices.

For objects that mimic AbstractArray but are not subtypes

`has_offset_axes` depends on having axes defined for the objects you call it on. If there is some reason you don't have an `axes` method defined for your object, consider defining a method

```
Base.has_offset_axes(obj::MyNon1IndexedArraylikeObject) = true
```

This will allow code that assumes 1-based indexing to detect a problem and throw a helpful error, rather than returning incorrect results or segfaulting julia.

Catching errors

If your new array type triggers errors in other code, one helpful debugging step can be to comment out `@boundscheck` in your `getindex` and `setindex!` implementation. This will ensure that every element access checks bounds. Or, restart julia with `--check-bounds=yes`.

In some cases it may also be helpful to temporarily disable `size` and `length` for your new array type, since code that makes incorrect assumptions frequently uses these functions.

103.20 Module loading

`Base.require` is responsible for loading modules and it also manages the precompilation cache. It is the implementation of the `import` statement.

Experimental features

The features below are experimental and not part of the stable Julia API. Before building upon them inform yourself about the current thinking and whether they might change soon.

Module loading callbacks

It is possible to listen to the modules loaded by `Base.require`, by registering a callback.

```
loaded_packages = Channel{Symbol}()
callback = (mod::Symbol) -> put!(loaded_packages, mod)
push!(Base.package_callbacks, callback)
```

Please note that the symbol given to the callback is a non-unique identifier and it is the responsibility of the callback provider to walk the module chain to determine the fully qualified name of the loaded binding.

The callback below is an example of how to do that:

```
# Get the fully-qualified name of a module.
function module_fqn(name::Symbol)
    fqn = fullname(Base.root_module(name))
    return join(fqn, '.')
end
```

103.21 类型推导

类型推导是如何工作的

In Julia compiler, "type inference" refers to the process of deducing the types of later values from the types of input values. Julia's approach to inference has been described in the blog posts below:

1. Shows a simplified implementation of the data-flow analysis algorithm, that Julia's type inference routine is based on.
2. Gives a high level view of inference with a focus on its inter-procedural convergence guarantee.
3. Explains a refinement on the algorithm introduced in 2.

调试 compiler.jl

You can start a Julia session, edit `compiler/*.jl` (for example to insert `print` statements), and then replace `Core.Compiler` in your running session by navigating to base and executing `include("compiler/compiler.jl")`. This trick typically leads to much faster development than if you rebuild Julia for each change.

Alternatively, you can use the Revise.jl package to track the compiler changes by using the command `Revise.track(Core.Compil` at the beginning of your Julia session. As explained in the Revise documentation, the modifications to the compiler will be reflected when the modified files are saved.

A convenient entry point into inference is `typeinf_code`. Here's a demo running inference on `convert(Int, UInt(1))`:

```
# Get the method
atypes = Tuple{Type{Int}, UInt}  # argument types
mths = methods(convert, atypes)  # worth checking that there is only one
m = first(mths)

# Create variables needed to call `typeinf_code`
interp = Core.Compiler.NativeInterpreter()
sparams = Core.svec()      # this particular method doesn't have type-parameters
optimize = true            # run all inference optimizations
types = Tuple{typeof(convert), atypes.parameters...} # Tuple{typeof(convert), Type{Int}, UInt}
Core.Compiler.typeinf_code(interp, m, types, sparams, optimize)
```

If your debugging adventures require a `MethodInstance`, you can look it up by calling `Core.Compiler.specialize_method` using many of the variables above. A `CodeInfo` object may be obtained with

```
# Returns the CodeInfo object for `convert(Int, ::UInt)`:
ci = (@code_typed convert(Int, UInt(1)))[1]
```

The inlining algorithm (`inline_worthy`)

Much of the hardest work for inlining runs in `ssa_inlining_pass!`. However, if your question is "why didn't my function inline?" then you will most likely be interested in `inline_worthy`, which makes a decision to inline the function call or not.

`inline_worthy` implements a cost-model, where "cheap" functions get inlined; more specifically, we inline functions if their anticipated run-time is not large compared to the time it would take to issue a call to them if they were not inlined. The cost-model is extremely simple and ignores many important details: for example, all `for` loops are analyzed as if they will be executed once, and the cost of an `if...else...end` includes the summed cost of all branches. It's also worth acknowledging that we currently lack a suite of functions suitable for testing how well the cost model predicts the actual run-time cost, although BaseBenchmarks provides a great deal of indirect information about the successes and failures of any modification to the inlining algorithm.

The foundation of the cost-model is a lookup table, implemented in `add_tfunc` and its callers, that assigns an estimated cost (measured in CPU cycles) to each of Julia's intrinsic functions. These costs are based on standard ranges for common architectures (see Agner Fog's analysis for more detail).

We supplement this low-level lookup table with a number of special cases. For example, an `:invoke` expression (a call for which all input and output types were inferred in advance) is assigned a fixed cost (currently 20 cycles). In contrast, a `:call` expression, for functions other than intrinsics/builtins, indicates that the call will require dynamic dispatch, in which case we assign a cost set by `Params.inline_nonleaf_penalty` (currently set at `1000`). Note that this is not a "first-principles" estimate of the raw cost of dynamic dispatch, but a mere heuristic indicating that dynamic dispatch is extremely expensive.

Each statement gets analyzed for its total cost in a function called `statement_cost`. You can display the cost associated with each statement as follows:

```
julia> Base.print_statement_costs(stdout, map, (typeof(sqrt), Tuple{Int},)) # map(sqrt, (2,))
map(f, t::Tuple{Any}) @ Base tuple.jl:273
  0 1 ─ %1  = Base.getfield(_3, 1, true)::Int64
  1 │   %2  = Base.sitofp(Float64, %1)::Float64
  2 │   %3  = Base.lt_float(%2, 0.0)::Bool
  0 └──       goto #3 if not %3
  0 2 ─       invoke Base.Math.throw_complex_domainerror(:sqrt::Symbol, %2::Float64)::Union{}
  0 └──       unreachable
 20 3 ─ %7  = Base.Math.sqrt_llvm(%2)::Float64
  0 └──       goto #4
  0 4 ─       goto #5
  0 5 ─ %10 = Core.tuple(%7)::Tuple{Float64}
  0 └──       return %10
```

The line costs are in the left column. This includes the consequences of inlining and other forms of optimization.

103.22 Julia SSA-form IR

Background

Beginning in Julia 0.7, parts of the compiler use a new SSA-form intermediate representation (IR). Historically, the compiler would directly generate LLVM IR from a lowered form of the Julia AST. This form had most syntactic abstractions removed, but still looked a lot like an abstract syntax tree. Over time, in order to facilitate optimizations, SSA values were introduced to this IR and the IR was linearized (i.e. turned into a form where function arguments could only be SSA values or constants). However, non-SSA values (slots) remained in the IR due to the lack of Phi nodes in the IR (necessary for back-edges and re-merging of conditional control flow). This negated much of the usefulness of SSA form representation when performing middle end optimizations. Some heroic effort was put into making these optimizations work without a complete SSA form representation, but the lack of such a representation ultimately proved prohibitive.

New IR nodes

With the new IR representation, the compiler learned to handle four new IR nodes, Phi nodes, Pi nodes as well as PhiC nodes and Upsilon nodes (the latter two are only used for exception handling).

Phi nodes and Pi nodes

Phi nodes are part of generic SSA abstraction (see the link above if you're not familiar with the concept). In the Julia IR, these nodes are represented as:

```
struct PhiNode
    edges::Vector{Int32}
    values::Vector{Any}
end
```

where we ensure that both vectors always have the same length. In the canonical representation (the one handled by codegen and the interpreter), the edge values indicate come-from statement numbers (i.e. if edge has an entry of 15, there must be a `goto`, `gotoifnot` or implicit fall through from statement 15 that targets this phi node). Values are either SSA values or constants. It is also possible for a value to be unassigned if the

variable was not defined on this path. However, undefinedness checks get explicitly inserted and represented as booleans after middle end optimizations, so code generators may assume that any use of a Phi node will have an assigned value in the corresponding slot. It is also legal for the mapping to be incomplete, i.e. for a Phi node to have missing incoming edges. In that case, it must be dynamically guaranteed that the corresponding value will not be used.

PiNodes encode statically proven information that may be implicitly assumed in basic blocks dominated by a given pi node. They are conceptually equivalent to the technique introduced in the paper ABCD: Eliminating Array Bounds Checks on Demand or the predicate info nodes in LLVM. To see how they work, consider, e.g.

```
%x::Union{Int, Float64} # %x is some Union{Int, Float64} typed ssa value
if isa(x, Int)
    # use x
else
    # use x
end
```

We can perform predicate insertion and turn this into:

```
%x::Union{Int, Float64} # %x is some Union{Int, Float64} typed ssa value
if isa(x, Int)
    %x_int = PiNode(x, Int)
    # use %x_int
else
    %x_float = PiNode(x, Float64)
    # use %x_float
end
```

Pi nodes are generally ignored in the interpreter, since they don't have any effect on the values, but they may sometimes lead to code generation in the compiler (e.g. to change from an implicitly union split representation to a plain unboxed representation). The main usefulness of PiNodes stems from the fact that path conditions of the values can be accumulated simply by def-use chain walking that is generally done for most optimizations that care about these conditions anyway.

PhiC nodes and Upsilon nodes

Exception handling complicates the SSA story moderately, because exception handling introduces additional control flow edges into the IR across which values must be tracked. One approach to do so, which is followed by LLVM, is to make calls which may throw exceptions into basic block terminators and add an explicit control flow edge to the catch handler:

```
invoke @function_that_may_throw() to label %regular unwind to %catch

regular:
# Control flow continues here

catch:
# Exceptions go here
```

However, this is problematic in a language like Julia, where at the start of the optimization pipeline, we do not know which calls throw. We would have to conservatively assume that every call (which in Julia is every statement) throws. This would have several negative effects. On the one hand, it would essentially reduce

the scope of every basic block to a single call, defeating the purpose of having operations be performed at the basic block level. On the other hand, every catch basic block would have `n*m` phi node arguments (`n`, the number of statements in the critical region, `m` the number of live values through the catch block).

To work around this, we use a combination of `Upsilon` and `PhiC` nodes (the C standing for `catch`, written φ^c in the IR pretty printer, because unicode subscript c is not available). There are several ways to think of these nodes, but perhaps the easiest is to think of each `PhiC` as a load from a unique store-many, read-once slot, with `Upsilon` being the corresponding store operation. The `PhiC` has an operand list of all the upsilon nodes that store to its implicit slot. The `Upsilon` nodes however, do not record which `PhiC` node they store to. This is done for more natural integration with the rest of the SSA IR. E.g. if there are no more uses of a `PhiC` node, it is safe to delete it, and the same is true of an `Upsilon` node. In most IR passes, `PhiC` nodes can be treated like `Phi` nodes. One can follow use-def chains through them, and they can be lifted to new `PhiC` nodes and new `Upsilon` nodes (in the same places as the original `Upsilon` nodes). The result of this scheme is that the number of `Upsilon` nodes (and `PhiC` arguments) is proportional to the number of assigned values to a particular variable (before SSA conversion), rather than the number of statements in the critical region.

To see this scheme in action, consider the function

```
@noinline opaque() = invokelatest(identity, nothing) # Something opaque
function foo()
    local y
    x = 1
    try
        y = 2
        opaque()
        y = 3
        error()
    catch
    end
    (x, y)
end
```

The corresponding IR (with irrelevant types stripped) is:

```
1 ─       nothing::Nothing
2 ─ %2  = $(Expr(:enter, #4))
3 ─ %3  = ϒ (false)
│   %4  = ϒ (#undef)
│   %5  = ϒ (1)
│   %6  = ϒ (true)
│   %7  = ϒ (2)
│         invoke Main.opaque()::Any
│   %9  = ϒ (true)
│   %10 = ϒ (3)
│         invoke Main.error()::Union{}
└──       $(Expr(:unreachable))::Union{}
4 ┄ %13 = φᶜ (%3, %6, %9)::Bool
│   %14 = φᶜ (%4, %7, %10)::Core.Compiler.MaybeUndef(Int64)
│   %15 = φᶜ (%5)::Core.Const(1)
└──       $(Expr(:leave, 1))
5 ─       $(Expr(:pop_exception, :(%2)))::Any
│         $(Expr(:throw_undef_if_not, :y, :(%13)))::Any
│   %19 = Core.tuple(%15, %14)
└──       return %19
```

Note in particular that every value live into the critical region gets an upsilon node at the top of the critical region. This is because catch blocks are considered to have an invisible control flow edge from outside the function. As a result, no SSA value dominates the catch blocks, and all incoming values have to come through a φ^c node.

Main SSA data structure

The main `SSAIR` data structure is worthy of discussion. It draws inspiration from LLVM and Webkit's B3 IR. The core of the data structure is a flat vector of statements. Each statement is implicitly assigned an SSA value based on its position in the vector (i.e. the result of the statement at idx 1 can be accessed using `SSAValue(1)` etc). For each SSA value, we additionally maintain its type. Since, SSA values are definitionally assigned only once, this type is also the result type of the expression at the corresponding index. However, while this representation is rather efficient (since the assignments don't need to be explicitly encoded), it of course carries the drawback that order is semantically significant, so reorderings and insertions change statement numbers. Additionally, we do not keep use lists (i.e. it is impossible to walk from a def to all its uses without explicitly computing this map–def lists however are trivial since you can look up the corresponding statement from the index), so the LLVM-style RAUW (replace-all-uses-with) operation is unavailable.

Instead, we do the following:

- We keep a separate buffer of nodes to insert (including the position to insert them at, the type of the corresponding value and the node itself). These nodes are numbered by their occurrence in the insertion buffer, allowing their values to be immediately used elsewhere in the IR (i.e. if there are 12 statements in the original statement list, the first new statement will be accessible as `SSAValue(13)`).
- RAUW style operations are performed by setting the corresponding statement index to the replacement value.
- Statements are erased by setting the corresponding statement to `nothing` (this is essentially just a special-case convention of the above.
- If there are any uses of the statement being erased, they will be set to `nothing`.

There is a `compact!` function that compacts the above data structure by performing the insertion of nodes in the appropriate place, trivial copy propagation, and renaming of uses to any changed SSA values. However, the clever part of this scheme is that this compaction can be done lazily as part of the subsequent pass. Most optimization passes need to walk over the entire list of statements, performing analysis or modifications along the way. We provide an `IncrementalCompact` iterator that can be used to iterate over the statement list. It will perform any necessary compaction and return the new index of the node, as well as the node itself. It is legal at this point to walk def-use chains, as well as make any modifications or deletions to the IR (insertions are disallowed however).

The idea behind this arrangement is that, since the optimization passes need to touch the corresponding memory anyway and incur the corresponding memory access penalty, performing the extra housekeeping should have comparatively little overhead (and save the overhead of maintaining these data structures during IR modification).

103.23 `EscapeAnalysis`

`Core.Compiler.EscapeAnalysis` is a compiler utility module that aims to analyze escape information of Julia's SSA-form IR a.k.a. `IRCode`.

This escape analysis aims to:

- leverage Julia's high-level semantics, especially reason about escapes and aliasing via inter-procedural calls
- be versatile enough to be used for various optimizations including alias-aware SROA, early finalize insertion, copy-free ImmutableArray construction, stack allocation of mutable objects, and so on.
- achieve a simple implementation based on a fully backward data-flow analysis implementation as well as a new lattice design that combines orthogonal lattice properties

Try it out!

You can give a try to the escape analysis by loading the `EAUtils.jl` utility script that define the convenience entries code_escapes and @code_escapes for testing and debugging purposes:

```
julia> include(normpath(Sys.BINDIR, "..", "share", "julia", "test", "compiler", "EscapeAnalysis",
↪  "EAUtils.jl")); using .EAUtils

julia> mutable struct SafeRef{T}
           x::T
       end

julia> Base.getindex(x::SafeRef) = x.x;

julia> Base.setindex!(x::SafeRef, v) = x.x = v;

julia> Base.isassigned(x::SafeRef) = true;

julia> get′(x) = isassigned(x) ? x[] : throw(x);

julia> result = code_escapes((String,String,String,String)) do s1, s2, s3, s4
           r1 = Ref(s1)
           r2 = Ref(s2)
           r3 = SafeRef(s3)
           try
               s1 = get′(r1)
               ret = sizeof(s1)
           catch err
               global GV = err # will definitely escape `r1`
           end
           s2 = get′(r2)       # still `r2` doesn't escape fully
           s3 = get′(r3)       # still `r3` doesn't escape fully
           s4 = sizeof(s4)     # the argument `s4` doesn't escape here
           return s2, s3, s4
       end
#1(X _2::String, ↑ _3::String, ↑ _4::String, ✓ _5::String) in Main at REPL[7]:2
X  1 ── %1  = %new(Base.RefValue{String}, _2)::Base.RefValue{String}
*′ │    %2  = %new(Base.RefValue{String}, _3)::Base.RefValue{String}
✓′ └─── %3  = %new(Main.SafeRef{String}, _4)::Main.SafeRef{String}
✓′ 2 ── %4  = ϒ (%3)::Main.SafeRef{String}
*′ │    %5  = ϒ (%2)::Base.RefValue{String}
✓  │    %6  = ϒ (_5)::String
◌  └─── %7  = $(Expr(:enter, #8))
◌  3 ── %8  = Base.isdefined(%1, :x)::Bool
◌  └───       goto #5 if not %8
X  4 ──       Base.getfield(%1, :x)::String
◌  └───       goto #6
```

```
◌  5 ──       Main.throw(%1)::Union{}
◌  └───       unreachable
◌  6 ──       $(Expr(:leave, 1))
◌  7 ──       goto #11
✓′ 8 ┄─ %16 = φᶜ (%4)::Main.SafeRef{String}
*′ │    %17 = φᶜ (%5)::Base.RefValue{String}
✓  │    %18 = φᶜ (%6)::String
◌  └───       $(Expr(:leave, 1))
X  9 ── %20 = $(Expr(:the_exception))::Any
◌  10 ─       (Main.GV = %20)::Any
◌  └───       $(Expr(:pop_exception, :(%7)))::Any
✓′ 11 ┄ %23 = φ (#7 => %3, #10 => %16)::Main.SafeRef{String}
*′ │    %24 = φ (#7 => %2, #10 => %17)::Base.RefValue{String}
✓  │    %25 = φ (#7 => _5, #10 => %18)::String
◌  │    %26 = Base.isdefined(%24, :x)::Bool
◌  └───       goto #13 if not %26
↑  12 ─ %28 = Base.getfield(%24, :x)::String
◌  └───       goto #14
◌  13 ─       Main.throw(%24)::Union{}
◌  └───       unreachable
↑  14 ─ %32 = Base.getfield(%23, :x)::String
◌  │    %33 = Core.sizeof(%25)::Int64
↑′ │    %34 = Core.tuple(%28, %32, %33)::Tuple{String, String, Int64}
◌  └───       return %34
```

The symbols in the side of each call argument and SSA statements represents the following meaning:

- ◌ (plain): this value is not analyzed because escape information of it won't be used anyway (when the object is `isbitstype` for example)
- ✓ (green or cyan): this value never escapes (`has_no_escape(result.state[x])` holds), colored blue if it has arg escape also (`has_arg_escape(result.state[x])` holds)
- ↑ (blue or yellow): this value can escape to the caller via return (`has_return_escape(result.state[x])` holds), colored yellow if it has unhandled thrown escape also (`has_thrown_escape(result.state[x])` holds)
- X (red): this value can escape to somewhere the escape analysis can't reason about like escapes to a global memory (`has_all_escape(result.state[x])` holds)
- * (bold): this value's escape state is between the `ReturnEscape` and `AllEscape` in the partial order of `EscapeInfo`, colored yellow if it has unhandled thrown escape also (`has_thrown_escape(result.state[x])` holds)
- ′: this value has additional object field / array element information in its `AliasInfo` property

Escape information of each call argument and SSA value can be inspected programmatically as like:

```
julia> result.state[Core.Argument(3)] # get EscapeInfo of `s2`
ReturnEscape

julia> result.state[Core.SSAValue(3)] # get EscapeInfo of `r3`
NoEscape′
```

Analysis Design

Lattice Design

`EscapeAnalysis` is implemented as a data-flow analysis that works on a lattice of `x::EscapeInfo`, which is composed of the following properties:

- `x.Analyzed::Bool`: not formally part of the lattice, only indicates x has not been analyzed or not
- `x.ReturnEscape::BitSet`: records SSA statements where x can escape to the caller via return
- `x.ThrownEscape::BitSet`: records SSA statements where x can be thrown as exception (used for the exception handling described below)
- `x.AliasInfo`: maintains all possible values that can be aliased to fields or array elements of x (used for the alias analysis described below)
- `x.ArgEscape::Int` (not implemented yet): indicates it will escape to the caller through `setfield!` on argument(s)

These attributes can be combined to create a partial lattice that has a finite height, given the invariant that an input program has a finite number of statements, which is assured by Julia's semantics. The clever part of this lattice design is that it enables a simpler implementation of lattice operations by allowing them to handle each lattice property separately[1].

Backward Escape Propagation

This escape analysis implementation is based on the data-flow algorithm described in the paper[2]. The analysis works on the lattice of `EscapeInfo` and transitions lattice elements from the bottom to the top until every lattice element gets converged to a fixed point by maintaining a (conceptual) working set that contains program counters corresponding to remaining SSA statements to be analyzed. The analysis manages a single global state that tracks `EscapeInfo` of each argument and SSA statement, but also note that some flow-sensitivity is encoded as program counters recorded in `EscapeInfo`'s `ReturnEscape` property, which can be combined with domination analysis later to reason about flow-sensitivity if necessary.

One distinctive design of this escape analysis is that it is fully *backward*, i.e. escape information flows *from usages to definitions*. For example, in the code snippet below, EA first analyzes the statement `return %1` and imposes `ReturnEscape` on `%1` (corresponding to `obj`), and then it analyzes `%1 = %new(Base.RefValue{String}, _2}))` and propagates the `ReturnEscape` imposed on `%1` to the call argument `_2` (corresponding to `s`):

```
julia> code_escapes((String,)) do s
           obj = Ref(s)
           return obj
       end
#3(↑ _2::String) in Main at REPL[1]:2
↑′ 1 ─ %1 = %new(Base.RefValue{String}, _2)::Base.RefValue{String}
◌  └──      return %1
```

The key observation here is that this backward analysis allows escape information to flow naturally along the use-def chain rather than control-flow[3]. As a result this scheme enables a simple implementation of escape analysis, e.g. `PhiNode` for example can be handled simply by propagating escape information imposed on a `PhiNode` to its predecessor values:

```
julia> code_escapes((Bool, String, String)) do cnd, s, t
           if cnd
               obj = Ref(s)
           else
               obj = Ref(t)
           end
           return obj
       end
#5(✓ _2::Bool, ↑ _3::String, ↑ _4::String) in Main at REPL[1]:2
◌  1 ─      goto #3 if not _2
↑′ 2 ─ %2 = %new(Base.RefValue{String}, _3)::Base.RefValue{String}
◌  └──      goto #4
↑′ 3 ─ %4 = %new(Base.RefValue{String}, _4)::Base.RefValue{String}
↑′ 4 ┄ %5 = φ (#2 => %2, #3 => %4)::Base.RefValue{String}
◌  └──      return %5
```

Alias Analysis

EscapeAnalysis implements a backward field analysis in order to reason about escapes imposed on object fields with certain accuracy, and x::EscapeInfo's x.AliasInfo property exists for this purpose. It records all possible values that can be aliased to fields of x at "usage" sites, and then the escape information of that recorded values are propagated to the actual field values later at "definition" sites. More specifically, the analysis records a value that may be aliased to a field of object by analyzing getfield call, and then it propagates its escape information to the field when analyzing %new(...) expression or setfield! call[4].

```
julia> code_escapes((String,)) do s
           obj = SafeRef("init")
           obj[] = s
           v = obj[]
           return v
       end
#7(↑ _2::String) in Main at REPL[1]:2
✓′ 1 ─ %1 = %new(Main.SafeRef{String}, "init")::Main.SafeRef{String}
◌  │        Base.setfield!(%1, :x, _2)::String
↑  │   %3 = Base.getfield(%1, :x)::String
◌  └──      return %3
```

In the example above, ReturnEscape imposed on %3 (corresponding to v) is *not* directly propagated to %1 (corresponding to obj) but rather that ReturnEscape is only propagated to _2 (corresponding to s). Here %3 is recorded in %1's AliasInfo property as it can be aliased to the first field of %1, and then when analyzing Base.setfield!(%1, :x, _2)::String, that escape information is propagated to _2 but not to %1.

So EscapeAnalysis tracks which IR elements can be aliased across a getfield-%new/setfield! chain in order to analyze escapes of object fields, but actually this alias analysis needs to be generalized to handle other IR elements as well. This is because in Julia IR the same object is sometimes represented by different IR elements and so we should make sure that those different IR elements that actually can represent the same object share the same escape information. IR elements that return the same object as their operand(s), such as PiNode and typeassert, can cause that IR-level aliasing and thus requires escape information imposed on any of such aliased values to be shared between them. More interestingly, it is also needed for correctly reasoning about mutations on PhiNode. Let's consider the following example:

```
julia> code_escapes((Bool, String,)) do cond, x
           if cond
               ϕ2 = ϕ1 = SafeRef("foo")
           else
               ϕ2 = ϕ1 = SafeRef("bar")
           end
           ϕ2[] = x
           y = ϕ1[]
           return y
       end
#9(✓ _2::Bool, ↑ _3::String) in Main at REPL[1]:2
◌  1 ─      goto #3 if not _2
✓′ 2 ─ %2 = %new(Main.SafeRef{String}, "foo")::Main.SafeRef{String}
◌  └──      goto #4
✓′ 3 ─ %4 = %new(Main.SafeRef{String}, "bar")::Main.SafeRef{String}
✓′ 4 ┄ %5 = φ (#2 => %2, #3 => %4)::Main.SafeRef{String}
✓′ │   %6 = φ (#2 => %2, #3 => %4)::Main.SafeRef{String}
◌  │        Base.setfield!(%5, :x, _3)::String
↑  │   %8 = Base.getfield(%6, :x)::String
◌  └──      return %8
```

`ϕ1 = %5` and `ϕ2 = %6` are aliased and thus `ReturnEscape` imposed on `%8 = Base.getfield(%6, :x)::String` (corresponding to `y = ϕ1[]`) needs to be propagated to `Base.setfield!(%5, :x, _3)::String` (corresponding to `ϕ2[] = x`). In order for such escape information to be propagated correctly, the analysis should recognize that the *predecessors* of `ϕ1` and `ϕ2` can be aliased as well and equalize their escape information.

One interesting property of such aliasing information is that it is not known at "usage" site but can only be derived at "definition" site (as aliasing is conceptually equivalent to assignment), and thus it doesn't naturally fit in a backward analysis. In order to efficiently propagate escape information between related values, EscapeAnalysis.jl uses an approach inspired by the escape analysis algorithm explained in an old JVM paper[5]. That is, in addition to managing escape lattice elements, the analysis also maintains an "equi"-alias set, a disjoint set of aliased arguments and SSA statements. The alias set manages values that can be aliased to each other and allows escape information imposed on any of such aliased values to be equalized between them.

Array Analysis

The alias analysis for object fields described above can also be generalized to analyze array operations. `EscapeAnalysis` implements handlings for various primitive array operations so that it can propagate escapes via `arrayref`-`arrayset` use-def chain and does not escape allocated arrays too conservatively:

```
julia> code_escapes((String,)) do s
           ary = Any[]
           push!(ary, SafeRef(s))
           return ary[1], length(ary)
       end
#11(↑ _2::String) in Main at REPL[1]:2
*′ 1 ─ %1 = $(Expr(:foreigncall, :(:jl_alloc_array_1d), Vector{Any}, svec(Any, Int64), 0,
↪  :(:ccall), Vector{Any}, 0, 0))::Vector{Any}
↑′ │   %2 = %new(Main.SafeRef{String}, _2)::Main.SafeRef{String}
◌  │        $(Expr(:foreigncall, :(:jl_array_grow_end), Nothing, svec(Any, UInt64), 0, :(:ccall),
↪  :(%1), 0x0000000000000001, 0x0000000000000001))::Nothing
◌  │   %4 = Base.arraylen(%1)::Int64
◌  │        Base.arrayset(false, %1, %2, %4)::Vector{Any}
```

```
↑′ │   %6 = Base.arrayref(true, %1, 1)::Any
◌  │   %7 = Base.arraylen(%1)::Int64
↑′ │   %8 = Core.tuple(%6, %7)::Tuple{Any, Int64}
◌  └──      return %8
```

In the above example `EscapeAnalysis` understands that `%20` and `%2` (corresponding to the allocated object `SafeRef(s)`) are aliased via the `arrayset`-`arrayref` chain and imposes `ReturnEscape` on them, but not impose it on the allocated array `%1` (corresponding to `ary`). `EscapeAnalysis` still imposes `ThrownEscape` on `ary` since it also needs to account for potential escapes via `BoundsError`, but also note that such unhandled `ThrownEscape` can often be ignored when optimizing the `ary` allocation.

Furthermore, in cases when array index information as well as array dimensions can be known *precisely*, `EscapeAnalysis` is able to even reason about "per-element" aliasing via `arrayref`-`arrayset` chain, as `EscapeAnalysis` does "per-field" alias analysis for objects:

```
julia> code_escapes((String,String)) do s, t
           ary = Vector{Any}(undef, 2)
           ary[1] = SafeRef(s)
           ary[2] = SafeRef(t)
           return ary[1], length(ary)
       end
#13(↑ _2::String, * _3::String) in Main at REPL[1]:2
*′ 1 ─ %1 = $(Expr(:foreigncall, :(:jl_alloc_array_1d), Vector{Any}, svec(Any, Int64), 0,
↪  :(:ccall), Vector{Any}, 2, 2))::Vector{Any}
↑′ │   %2 = %new(Main.SafeRef{String}, _2)::Main.SafeRef{String}
◌  │        Base.arrayset(true, %1, %2, 1)::Vector{Any}
*′ │   %4 = %new(Main.SafeRef{String}, _3)::Main.SafeRef{String}
◌  │        Base.arrayset(true, %1, %4, 2)::Vector{Any}
↑′ │   %6 = Base.arrayref(true, %1, 1)::Any
◌  │   %7 = Base.arraylen(%1)::Int64
↑′ │   %8 = Core.tuple(%6, %7)::Tuple{Any, Int64}
◌  └──      return %8
```

Note that `ReturnEscape` is only imposed on `%2` (corresponding to `SafeRef(s)`) but not on `%4` (corresponding to `SafeRef(t)`). This is because the allocated array's dimension and indices involved with all `arrayref`/`arrayset` operations are available as constant information and `EscapeAnalysis` can understand that `%6` is aliased to `%2` but never be aliased to `%4`. In this kind of case, the succeeding optimization passes will be able to replace `Base.arrayref(true, %1, 1)::Any` with `%2` (a.k.a. "load-forwarding") and eventually eliminate the allocation of array `%1` entirely (a.k.a. "scalar-replacement").

When compared to object field analysis, where an access to object field can be analyzed trivially using type information derived by inference, array dimension isn't encoded as type information and so we need an additional analysis to derive that information. `EscapeAnalysis` at this moment first does an additional simple linear scan to analyze dimensions of allocated arrays before firing up the main analysis routine so that the succeeding escape analysis can precisely analyze operations on those arrays.

However, such precise "per-element" alias analysis is often hard. Essentially, the main difficulty inherit to array is that array dimension and index are often non-constant:

- loop often produces loop-variant, non-constant array indices
- (specific to vectors) array resizing changes array dimension and invalidates its constant-ness

Let's discuss those difficulties with concrete examples.

In the following example, `EscapeAnalysis` fails the precise alias analysis since the index at the `Base.arrayset(false, %4, %8, %6)::Vector{Any}` is not (trivially) constant. Especially `Any[nothing, nothing]` forms a loop and calls that `arrayset` operation in a loop, where `%6` is represented as a ϕ-node value (whose value is control-flow dependent). As a result, `ReturnEscape` ends up imposed on both `%23` (corresponding to `SafeRef(s)`) and `%25` (corresponding to `SafeRef(t)`), although ideally we want it to be imposed only on `%23` but not on `%25`:

```
julia> code_escapes((String,String)) do s, t
           ary = Any[nothing, nothing]
           ary[1] = SafeRef(s)
           ary[2] = SafeRef(t)
           return ary[1], length(ary)
       end
#15(↑ _2::String, ↑ _3::String) in Main at REPL[1]:2
*′ 1 ─ %1  = $(Expr(:foreigncall, :(:jl_alloc_array_1d), Vector{Any}, svec(Any, Int64), 0,
↪  :(:ccall), Vector{Any}, 2, 2))::Vector{Any}
◌  └──       goto #7 if not true
◌  2 ┄ %3  = φ (#1 => 1, #6 => %11)::Int64
◌  │   %4  = φ (#1 => 1, #6 => %12)::Int64
◌  │         Base.arrayset(false, %1, nothing, %3)::Vector{Any}
◌  │   %6  = (%4 === 2)::Bool
◌  └──       goto #4 if not %6
◌  3 ─       goto #5
◌  4 ─ %9  = Base.add_int(%4, 1)::Int64
◌  └──       goto #5
◌  5 ┄ %11 = φ (#4 => %9)::Int64
◌  │   %12 = φ (#4 => %9)::Int64
◌  │   %13 = φ (#3 => true, #4 => false)::Bool
◌  │   %14 = Base.not_int(%13)::Bool
◌  └──       goto #7 if not %14
◌  6 ─       goto #2
◌  7 ┄       goto #8
↑′ 8 ─ %18 = %new(Main.SafeRef{String}, _2)::Main.SafeRef{String}
◌  │         Base.arrayset(true, %1, %18, 1)::Vector{Any}
↑′ │   %20 = %new(Main.SafeRef{String}, _3)::Main.SafeRef{String}
◌  │         Base.arrayset(true, %1, %20, 2)::Vector{Any}
↑′ │   %22 = Base.arrayref(true, %1, 1)::Any
◌  │   %23 = Base.arraylen(%1)::Int64
↑′ │   %24 = Core.tuple(%22, %23)::Tuple{Any, Int64}
◌  └──       return %24
```

The next example illustrates how vector resizing makes precise alias analysis hard. The essential difficulty is that the dimension of allocated array `%1` is first initialized as `0`, but it changes by the two `:jl_array_grow_end` calls afterwards. `EscapeAnalysis` currently simply gives up precise alias analysis whenever it encounters any array resizing operations and so `ReturnEscape` is imposed on both `%2` (corresponding to `SafeRef(s)`) and `%20` (corresponding to `SafeRef(t)`):

```
julia> code_escapes((String,String)) do s, t
           ary = Any[]
           push!(ary, SafeRef(s))
           push!(ary, SafeRef(t))
           ary[1], length(ary)
       end
#17(↑ _2::String, ↑ _3::String) in Main at REPL[1]:2
```

```
*′ 1 ─ %1  = $(Expr(:foreigncall, :(:jl_alloc_array_1d), Vector{Any}, svec(Any, Int64), 0,
↪  :(:ccall), Vector{Any}, 0, 0))::Vector{Any}
↑′ │   %2  = %new(Main.SafeRef{String}, _2)::Main.SafeRef{String}
◌  │         $(Expr(:foreigncall, :(:jl_array_grow_end), Nothing, svec(Any, UInt64), 0, :(:ccall),
↪  :(%1), 0x0000000000000001, 0x0000000000000001))::Nothing
◌  │   %4  = Base.arraylen(%1)::Int64
◌  │         Base.arrayset(false, %1, %2, %4)::Vector{Any}
↑′ │   %6  = %new(Main.SafeRef{String}, _3)::Main.SafeRef{String}
◌  │         $(Expr(:foreigncall, :(:jl_array_grow_end), Nothing, svec(Any, UInt64), 0, :(:ccall),
↪  :(%1), 0x0000000000000001, 0x0000000000000001))::Nothing
◌  │   %8  = Base.arraylen(%1)::Int64
◌  │         Base.arrayset(false, %1, %6, %8)::Vector{Any}
↑′ │   %10 = Base.arrayref(true, %1, 1)::Any
◌  │   %11 = Base.arraylen(%1)::Int64
↑′ │   %12 = Core.tuple(%10, %11)::Tuple{Any, Int64}
◌  └──       return %12
```

In order to address these difficulties, we need inference to be aware of array dimensions and propagate array dimensions in a flow-sensitive way[6], as well as come up with nice representation of loop-variant values.

EscapeAnalysis at this moment quickly switches to the more imprecise analysis that doesn't track precise index information in cases when array dimensions or indices are trivially non constant. The switch can naturally be implemented as a lattice join operation of EscapeInfo.AliasInfo property in the data-flow analysis framework.

Exception Handling

It would be also worth noting how EscapeAnalysis handles possible escapes via exceptions. Naively it seems enough to propagate escape information imposed on :the_exception object to all values that may be thrown in a corresponding try block. But there are actually several other ways to access to the exception object in Julia, such as Base.current_exceptions and rethrow. For example, escape analysis needs to account for potential escape of r in the example below:

```
julia> const GR = Ref{Any}();

julia> @noinline function rethrow_escape!()
           try
               rethrow()
           catch err
               GR[] = err
           end
       end;

julia> get′(x) = isassigned(x) ? x[] : throw(x);

julia> code_escapes() do
           r = Ref{String}()
           local t
           try
               t = get′(r)
           catch err
               t = typeof(err)   # `err` (which `r` aliases to) doesn't escape here
               rethrow_escape!() # but `r` escapes here
           end
```

```
           return t
       end
#19() in Main at REPL[4]:2
X  1 ── %1  = %new(Base.RefValue{String})::Base.RefValue{String}
◌  2 ── %2  = $(Expr(:enter, #8))
◌  3 ── %3  = Base.isdefined(%1, :x)::Bool
◌  └───       goto #5 if not %3
X  4 ── %5  = Base.getfield(%1, :x)::String
◌  └───       goto #6
◌  5 ──       Main.throw(%1)::Union{}
◌  └───       unreachable
◌  6 ──       $(Expr(:leave, 1))
◌  7 ──       goto #10
◌  8 ┈─       $(Expr(:leave, 1))
✓  9 ── %12 = $(Expr(:the_exception))::Any
X  │    %13 = Main.typeof(%12)::DataType
X  │          invoke Main.rethrow_escape!()::Any
◌  └───       $(Expr(:pop_exception, :(%2)))::Any
X  10 ┈ %16 = φ (#7 => %5, #9 => %13)::Union{DataType, String}
◌  └───       return %16
```

It requires a global analysis in order to correctly reason about all possible escapes via existing exception interfaces. For now we always propagate the topmost escape information to all potentially thrown objects conservatively, since such an additional analysis might not be worthwhile to do given that exception handling and error path usually don't need to be very performance sensitive, and also optimizations of error paths might be very ineffective anyway since they are often even "unoptimized" intentionally for latency reasons.

`x::EscapeInfo`'s `x.ThrownEscape` property records SSA statements where `x` can be thrown as an exception. Using this information `EscapeAnalysis` can propagate possible escapes via exceptions limitedly to only those may be thrown in each `try` region:

```
julia> result = code_escapes((String,String)) do s1, s2
           r1 = Ref(s1)
           r2 = Ref(s2)
           local ret
           try
               s1 = get′(r1)
               ret = sizeof(s1)
           catch err
               global GV = err # will definitely escape `r1`
           end
           s2 = get′(r2)       # still `r2` doesn't escape fully
           return s2
       end
#21(X _2::String, ↑ _3::String) in Main at REPL[1]:2
X  1 ── %1  = %new(Base.RefValue{String}, _2)::Base.RefValue{String}
*′ └─── %2  = %new(Base.RefValue{String}, _3)::Base.RefValue{String}
*′ 2 ── %3  = ϒ (%2)::Base.RefValue{String}
◌  └─── %4  = $(Expr(:enter, #8))
◌  3 ── %5  = Base.isdefined(%1, :x)::Bool
◌  └───       goto #5 if not %5
X  4 ──       Base.getfield(%1, :x)::String
◌  └───       goto #6
◌  5 ──       Main.throw(%1)::Union{}
```

```
◌  └───       unreachable
◌  6 ──       $(Expr(:leave, 1))
◌  7 ──       goto #11
*′ 8 ┄─ %13 = φᶜ (%3)::Base.RefValue{String}
◌  └───       $(Expr(:leave, 1))
X  9 ── %15 = $(Expr(:the_exception))::Any
◌  10 ─       (Main.GV = %15)::Any
◌  └───       $(Expr(:pop_exception, :(%4)))::Any
*′ 11 ┄ %18 = φ (#7 => %2, #10 => %13)::Base.RefValue{String}
◌  │    %19 = Base.isdefined(%18, :x)::Bool
◌  └───       goto #13 if not %19
↑  12 ─ %21 = Base.getfield(%18, :x)::String
◌  └───       goto #14
◌  13 ─       Main.throw(%18)::Union{}
◌  └───       unreachable
◌  14 ─       return %21
```

Analysis Usage

`analyze_escapes` is the entry point to analyze escape information of SSA-IR elements.

Most optimizations like SROA (`sroa_pass!`) are more effective when applied to an optimized source that the inlining pass (`ssa_inlining_pass!`) has simplified by resolving inter-procedural calls and expanding callee sources. Accordingly, `analyze_escapes` is also able to analyze post-inlining IR and collect escape information that is useful for certain memory-related optimizations.

However, since certain optimization passes like inlining can change control flows and eliminate dead code, they can break the inter-procedural validity of escape information. In particularity, in order to collect inter-procedurally valid escape information, we need to analyze a pre-inlining IR.

Because of this reason, `analyze_escapes` can analyze `IRCode` at any Julia-level optimization stage, and especially, it is supposed to be used at the following two stages:

- `IPO EA`: analyze pre-inlining IR to generate IPO-valid escape information cache
- `Local EA`: analyze post-inlining IR to collect locally-valid escape information

Escape information derived by `IPO EA` is transformed to the `ArgEscapeCache` data structure and cached globally. By passing an appropriate `get_escape_cache` callback to `analyze_escapes`, the escape analysis can improve analysis accuracy by utilizing cached inter-procedural information of non-inlined callees that has been derived by previous `IPO EA`. More interestingly, it is also valid to use `IPO EA` escape information for type inference, e.g., inference accuracy can be improved by forming `Const`/`PartialStruct`/`MustAlias` of mutable object.

Since the computational cost of `analyze_escapes` is not that cheap, both `IPO EA` and `Local EA` are better to run only when there is any profitability. Currently `EscapeAnalysis` provides the `is_ipo_profitable` heuristic to check a profitability of `IPO EA`.

`Core.Compiler.EscapeAnalysis.analyze_escapes` – Function.

```julia
analyze_escapes(ir::IRCode, nargs::Int, call_resolved::Bool, get_escape_cache::Callable)
    -> estate::EscapeState
```

Analyzes escape information in `ir`:

- `nargs`: the number of actual arguments of the analyzed call
- `call_resolved`: if interprocedural calls are already resolved by `ssa_inlining_pass!`
- `get_escape_cache(::Union{InferenceResult,MethodInstance}) -> Union{Nothing,ArgEscapeCache}`: retrieves cached argument escape information

source

`Core.Compiler.EscapeAnalysis.EscapeState` – Type.

```
estate::EscapeState
```

Extended lattice that maps arguments and SSA values to escape information represented as `EscapeInfo`. Escape information imposed on SSA IR element x can be retrieved by `estate[x]`.

source

`Core.Compiler.EscapeAnalysis.EscapeInfo` – Type.

```
x::EscapeInfo
```

A lattice for escape information, which holds the following properties:

- `x.Analyzed::Bool`: not formally part of the lattice, only indicates x has not been analyzed or not
- `x.ReturnEscape::Bool`: indicates x can escape to the caller via return
- `x.ThrownEscape::BitSet`: records SSA statement numbers where x can be thrown as exception:
  - `isempty(x.ThrownEscape)`: x will never be thrown in this call frame (the bottom)
  - `pc ∈ x.ThrownEscape`: x may be thrown at the SSA statement at pc
  - `-1 ∈ x.ThrownEscape`: x may be thrown at arbitrary points of this call frame (the top)

  This information will be used by `escape_exception!` to propagate potential escapes via exception.
- `x.AliasInfo::Union{Bool,IndexableFields,IndexableElements,Unindexable}`: maintains all possible values that can be aliased to fields or array elements of x:
  - `x.AliasInfo === false` indicates the fields/elements of x aren't analyzed yet
  - `x.AliasInfo === true` indicates the fields/elements of x can't be analyzed, e.g. the type of x is not known or is not concrete and thus its fields/elements can't be known precisely
  - `x.AliasInfo::IndexableFields` records all the possible values that can be aliased to fields of object x with precise index information
  - `x.AliasInfo::IndexableElements` records all the possible values that can be aliased to elements of array x with precise index information
  - `x.AliasInfo::Unindexable` records all the possible values that can be aliased to fields/elements of x without precise index information
- `x.Liveness::BitSet`: records SSA statement numbers where x should be live, e.g. to be used as a call argument, to be returned to a caller, or preserved for `:foreigncall`:
  - `isempty(x.Liveness)`: x is never be used in this call frame (the bottom)
  - `0 ∈ x.Liveness` also has the special meaning that it's a call argument of the currently analyzed call frame (and thus it's visible from the caller immediately).

  - `pc ∈ x.Liveness`: `x` may be used at the SSA statement at `pc`
  - `-1 ∈ x.Liveness`: `x` may be used at arbitrary points of this call frame (the top)

There are utility constructors to create common `EscapeInfo`s, e.g.,

- `NoEscape()`: the bottom(-like) element of this lattice, meaning it won't escape to anywhere
- `AllEscape()`: the topmost element of this lattice, meaning it will escape to everywhere

`analyze_escapes` will transition these elements from the bottom to the top, in the same direction as Julia's native type inference routine. An abstract state will be initialized with the bottom(-like) elements:

- the call arguments are initialized as `ArgEscape()`, whose `Liveness` property includes `0` to indicate that it is passed as a call argument and visible from a caller immediately
- the other states are initialized as `NotAnalyzed()`, which is a special lattice element that is slightly lower than `NoEscape`, but at the same time doesn't represent any meaning other than it's not analyzed yet (thus it's not formally part of the lattice)

source

`Core.Compiler.EscapeAnalysis.is_ipo_profitable` – Function.

```
is_ipo_profitable(ir::IRCode, nargs::Int) -> Bool
```

Heuristically checks if there is any profitability to run the escape analysis on `ir` and generate IPO escape information cache. Specifically, this function examines if any call argument is "interesting" in terms of their escapability.

source

---

[1] Our type inference implementation takes the alternative approach, where each lattice property is represented by a special lattice element type object. It turns out that it started to complicate implementations of the lattice operations mainly because it often requires conversion rules between each lattice element type object. And we are working on overhauling our type inference lattice implementation with `EscapeInfo`-like lattice design.

[2] *A Graph-Free approach to Data-Flow Analysis*. Markas Mohnen, 2002, April. https://api.semanticscholar.org/CorpusID:28519618.

[3] Our type inference algorithm in contrast is implemented as a forward analysis, because type information usually flows from "definition" to "usage" and it is more natural and effective to propagate such information in a forward way.

[4] In some cases, however, object fields can't be analyzed precisely. For example, object may escape to somewhere `EscapeAnalysis` can't account for possible memory effects on it, or fields of the objects simply can't be known because of the lack of type information. In such cases `AliasInfo` property is raised to the topmost element within its own lattice order, and it causes succeeding field analysis to be conservative and escape information imposed on fields of an unanalyzable object to be propagated to the object itself.

[5] *Escape Analysis in the Context of Dynamic Compilation and Deoptimization*. Thomas Kotzmann and Hanspeter Mössenböck, 2005, June. https://dl.acm.org/doi/10.1145/1064979.1064996.

[6] Otherwise we will need yet another forward data-flow analysis on top of the escape analysis.

103.24 Static analyzer annotations for GC correctness in C code

Running the analysis

The analyzer plugin that drives the analysis ships with julia. Its source code can be found in `src/clangsa`. Running it requires the clang dependency to be build. Set the `BUILD_LLVM_CLANG` variable in your Make.user in order to build an appropriate version of clang. You may also want to use the prebuilt binaries using the `USE_BINARYBUILDER_LLVM` options.

Alternatively (or if these do not suffice), try

```
make -C src install-analysis-deps
```

from Julia's toplevel directory.

Afterwards, running the analysis over the source tree is as simple as running `make -C src analyzegc`.

General Overview

Since Julia's GC is precise, it needs to maintain correct rooting information for any value that may be referenced at any time GC may occur. These places are known as `safepoints` and in the function local context, we extend this designation to any function call that may recursively end up at a safepoint.

In generated code, this is taken care of automatically by the GC root placement pass (see the chapter on GC rooting in the LLVM codegen devdocs). However, in C code, we need to inform the runtime of any GC roots manually. This is done using the following macros:

```
// The value assigned to any slot passed as an argument to these
// is rooted for the duration of this GC frame.
JL_GC_PUSH{1,...,6}(args...)
// The values assigned into the size `n` array `rts` are rooted
// for the duration of this GC frame.
JL_GC_PUSHARGS(rts, n)
// Pop a GC frame
JL_GC_POP
```

If these macros are not used where they need to be, or they are used incorrectly, the result is silent memory corruption. As such it is very important that they are placed correctly in all applicable code.

As such, we employ static analysis (and in particular the clang static analyzer) to help ensure that these macros are used correctly. The remainder of this document gives an overview of this static analysis and describes the support needed in the julia code base to make things work.

GC Invariants

There is two simple invariants correctness:

- All `GC_PUSH` calls need to be followed by an appropriate `GC_POP` (in practice we enforce this at the function level)
- If a value was previously not rooted at any safepoint, it may no longer be referenced afterwards

Of course the devil is in the details here. In particular to satisfy the second of the above conditions, we need to know:

- Which calls are safepoints and which are not
- Which values are rooted at any given safepoint and which are not
- When is a value referenced

For the second point in particular, we need to know which memory locations will be considered rooting at runtime (i.e. values assigned to such locations are rooted). This includes locations explicitly designated as such by passing them to one of the `GC_PUSH` macros, globally rooted locations and values, as well as any location recursively reachable from one of those locations.

Static Analysis Algorithm

The idea itself is very simple, although the implementation is quite a bit more complicated (mainly due to a large number of special cases and intricacies of C and C++). In essence, we keep track of all locations that are rooting, all values that are rootable and any expression (assignments, allocations, etc) affect the rootedness of any rootable values. Then, at any safepoint, we perform a "symbolic GC" and poison any values that are not rooted at said location. If these values are later referenced, we emit an error.

The clang static analyzer works by constructing a graph of states and exploring this graph for sources of errors. Several nodes in this graph are generated by the analyzer itself (e.g. for control flow), but the definitions above augment this graph with our own state.

The static analyzer is interprocedural and can analyze control flow across function boundaries. However, the static analyzer is not fully recursive and makes heuristic decisions about which calls to explore (additionally some calls are cross-translation unit and invisible to the analyzer). In our case, our definition of correctness requires total information. As such, we need to annotate the prototypes of all function calls with whatever information the analysis required, even if that information would otherwise be available by interprocedural static analysis.

Luckily however, we can still use this interprocedural analysis to ensure that the annotations we place on a given function are indeed correct given the implementation of said function.

The analyzer annotations

These annotations are found in src/support/analyzer_annotations.h. The are only active when the analyzer is being used and expand either to nothing (for prototype annotations) or to no-ops (for function like annotations).

JL_NOTSAFEPOINT

This is perhaps the most common annotation, and should be placed on any function that is known not to possibly lead to reaching a GC safepoint. In general, it is only safe for such a function to perform arithmetic, memory accesses and calls to functions either annotated `JL_NOTSAFEPOINT` or otherwise known not to be safepoints (e.g. function in the C standard library, which are hardcoded as such in the analyzer)

It is valid to keep values unrooted across calls to any function annotated with this attribute:

Usage Example:

```
void jl_get_one() JL_NOTSAFEPOINT {
  return 1;
}

jl_value_t *example() {
  jl_value_t *val = jl_alloc_whatever();
  // This is valid, even though `val` is unrooted, because
```

```
  // jl_get_one is not a safepoint
  jl_get_one();
  return val;
}
```

JL_MAYBE_UNROOTED/JL_ROOTS_TEMPORARILY

When JL_MAYBE_UNROOTED is annotated as an argument on a function, indicates that said argument may be passed, even if it is not rooted. In the ordinary course of events, the julia ABI guarantees that callers root values before passing them to callees. However, some functions do not follow this ABI and allow values to be passed to them even though they are not rooted. Note however, that this does not automatically imply that said argument will be preserved. The ROOTS_TEMPORARILY annotation provides the stronger guarantee that, not only may the value be unrooted when passed, it will also be preserved across any internal safepoints by the callee.

Note that JL_NOTSAFEPOINT essentially implies JL_MAYBE_UNROOTED/JL_ROOTS_TEMPORARILY, because the rootedness of an argument is irrelevant if the function contains no safepoints.

One additional point to note is that these annotations apply on both the caller and the callee side. On the caller side, they lift rootedness restrictions that are normally required for julia ABI functions. On the callee side, they have the reverse effect of preventing these arguments from being considered implicitly rooted.

If either of these annotations is applied to the function as a whole, it applies to all arguments of the function. This should generally only be necessary for varargs functions.

Usage example:

```
JL_DLLEXPORT void JL_NORETURN jl_throw(jl_value_t *e JL_MAYBE_UNROOTED);
jl_value_t *jl_alloc_error();

void example() {
  // The return value of the allocation is unrooted. This would normally
  // be an error, but is allowed because of the above annotation.
  jl_throw(jl_alloc_error());
}
```

JL_PROPAGATES_ROOT

This annotation is commonly found on accessor functions that return one rootable object stored within another. When annotated on a function argument, it tells the analyzer that the root for that argument also applies to the value returned by the function.

Usage Example:

```
jl_value_t *jl_svecref(jl_svec_t *t JL_PROPAGATES_ROOT, size_t i) JL_NOTSAFEPOINT;

size_t example(jl_svec_t *svec) {
  jl_value_t *val = jl_svecref(svec, 1)
  // This is valid, because, as annotated by the PROPAGATES_ROOT annotation,
  // jl_svecref propagates the rooted-ness from `svec` to `val`
  jl_gc_safepoint();
  return jl_unbox_long(val);
}
```

JL_ROOTING_ARGUMENT/JL_ROOTED_ARGUMENT

This is essentially the assignment counterpart to `JL_PROPAGATES_ROOT`. When assigning a value to a field of another value that is already rooted, the assigned value will inherit the root of the value it is assigned into.

Usage Example:

```
void jl_svecset(void *t JL_ROOTING_ARGUMENT, size_t i, void *x JL_ROOTED_ARGUMENT) JL_NOTSAFEPOINT


size_t example(jl_svec_t *svec) {
  jl_value_t *val = jl_box_long(10000);
  jl_svecset(svec, val);
  // This is valid, because the annotations imply that the
  // jl_svecset propagates the rooted-ness from `svec` to `val`
  jl_gc_safepoint();
  return jl_unbox_long(val);
}
```

JL_GC_DISABLED

This annotation implies that this function is only called with the GC runtime-disabled. Functions of this kind are most often encountered during startup and in the GC code itself. Note that this annotation is checked against the runtime enable/disable calls, so clang will know if you lie. This is not a good way to disable processing of a given function if the GC is not actually disabled (use `ifdef __clang_analyzer__` for that if you must).

Usage example:

```
void jl_do_magic() JL_GC_DISABLED {
  // Wildly allocate here with no regard for roots
}

void example() {
  int en = jl_gc_enable(0);
  jl_do_magic();
  jl_gc_enable(en);
}
```

JL_REQUIRE_ROOTED_SLOT

This annotation requires the caller to pass in a slot that is rooted (i.e. values assigned to this slot will be rooted).

Usage example:

```
void jl_do_processing(jl_value_t **slot JL_REQUIRE_ROOTED_SLOT) {
  *slot = jl_box_long(1);
  // Ok, only, because the slot was annotated as rooting
  jl_gc_safepoint();
}

void example() {
  jl_value_t *slot = NULL;
  JL_GC_PUSH1(&slot);
  jl_do_processing(&slot);
```

```
  JL_GC_POP();
}
```

JL_GLOBALLY_ROOTED

This annotation implies that a given value is always globally rooted. It can be applied to global variable declarations, in which case it will apply to the value of those variables (or values if the declaration if for an array), or to functions, in which case it will apply to the return value of such functions (e.g. for functions that always return some private, globally rooted value).

Usage example:

```
extern JL_DLLEXPORT jl_datatype_t *jl_any_type JL_GLOBALLY_ROOTED;
jl_ast_context_t *jl_ast_ctx(fl_context_t *fl) JL_GLOBALLY_ROOTED;
```

JL_ALWAYS_LEAFTYPE

This annotations is essentially equivalent to `JL_GLOBALLY_ROOTED`, except that is should only be used if those values are globally rooted by virtue of being a leaftype. The rooting of leaftypes is a bit complicated. They are generally rooted through cache field of the corresponding `TypeName`, which itself is rooted by the containing module (so they're rooted as long as the containing module is ok) and we can generally assume that leaftypes are rooted where they are used, but we may refine this property in the future, so the separate annotation helps split out the reason for being globally rooted.

The analyzer also automatically detects checks for leaftype-ness and will not complain about missing GC roots on these paths.

```
JL_DLLEXPORT jl_value_t *jl_apply_array_type(jl_value_t *type, size_t dim) JL_ALWAYS_LEAFTYPE;
```

JL_GC_PROMISE_ROOTED

This is a function-like annotation. Any value passed to this annotation will be considered rooted for the scope of the current function. It is designed as an escape hatch for analyzer inadequacy or complicated situations. However, it should be used sparingly, in favor of improving the analyzer itself.

```
void example() {
  jl_value_t *val = jl_alloc_something();
  if (some_condition) {
    // We happen to know for complicated external reasons
    // that val is rooted under these conditions
    JL_GC_PROMISE_ROOTED(val);
  }
}
```

Completeness of analysis

The analyzer only looks at local information. In particular, e.g. in the `PROPAGATES_ROOT` case above, it assumes that such memory is only modified in ways it can see, not in any called functions (unless it happens to decide to consider them in its analysis) and not in any concurrently running threads. As such, it may miss a few problematic cases, though in practice such concurrent modification is fairly rare. Improving the analyzer to handle more such cases may be an interesting topic for future work.

103.25 Garbage Collection in Julia

Introduction

Julia has a non-moving, partially concurrent, parallel, generational and mostly precise mark-sweep collector (an interface for conservative stack scanning is provided as an option for users who wish to call Julia from C).

Allocation

Julia uses two types of allocators, the size of the allocation request determining which one is used. Objects up to 2k bytes are allocated on a per-thread free-list pool allocator, while objects larger than 2k bytes are allocated through libc malloc.

Julia's pool allocator partitions objects on different size classes, so that a memory page managed by the pool allocator (which spans 4 operating system pages on 64bit platforms) only contains objects of the same size class. Each memory page from the pool allocator is paired with some page metadata stored on per-thread lock-free lists. The page metadata contains information such as whether the page has live objects at all, number of free slots, and offsets to the first and last objects in the free-list contained in that page. These metadata are used to optimize the collection phase: a page which has no live objects at all may be returned to the operating system without any need of scanning it, for example.

While a page that has no objects may be returned to the operating system, its associated metadata is permanently allocated and may outlive the given page. As mentioned above, metadata for allocated pages are stored on per-thread lock-free lists. Metadata for free pages, however, may be stored into three separate lock-free lists depending on whether the page has been mapped but never accessed (`page_pool_clean`), or whether the page has been lazily sweeped and it's waiting to be madvised by a background GC thread (`page_pool_lazily_freed`), or whether the page has been madvised (`page_pool_freed`).

Julia's pool allocator follows a "tiered" allocation discipline. When requesting a memory page for the pool allocator, Julia will:

- Try to claim a page from `page_pool_lazily_freed`, which contains pages which were empty on the last stop-the-world phase, but not yet madivsed by a concurrent sweeper GC thread.
- If it failed claiming a page from `page_pool_lazily_freed`, it will try to claim a page from `the page_pool_clean`, which contains pages which were mmaped on a previous page allocation request but never accessed.
- If it failed claiming a page from `pool_page_clean` and from `page_pool_lazily_freed`, it will try to claim a page from `page_pool_freed`, which contains pages which have already been madvised by a concurrent sweeper GC thread and whose underlying virtual address can be recycled.
- If it failed in all of the attempts mentioned above, it will mmap a batch of pages, claim one page for itself, and insert the remaining pages into `page_pool_clean`.

Marking and Generational Collection

Julia's mark phase is implemented through a parallel iterative depth-first-search over the object graph. Julia's collector is non-moving, so object age information can't be determined through the memory region in which the object resides alone, but has to be somehow encoded in the object header or on a side table. The lowest two bits of an object's header are used to store, respectively, a mark bit that is set when an object is scanned during the mark phase and an age bit for the generational collection.

Generational collection is implemented through sticky bits: objects are only pushed to the mark-stack, and therefore traced, if their mark-bits are not set. When objects reach the oldest generation, their mark-bits are not reset during the so-called "quick-sweep", which leads to these objects not being traced in a subsequent

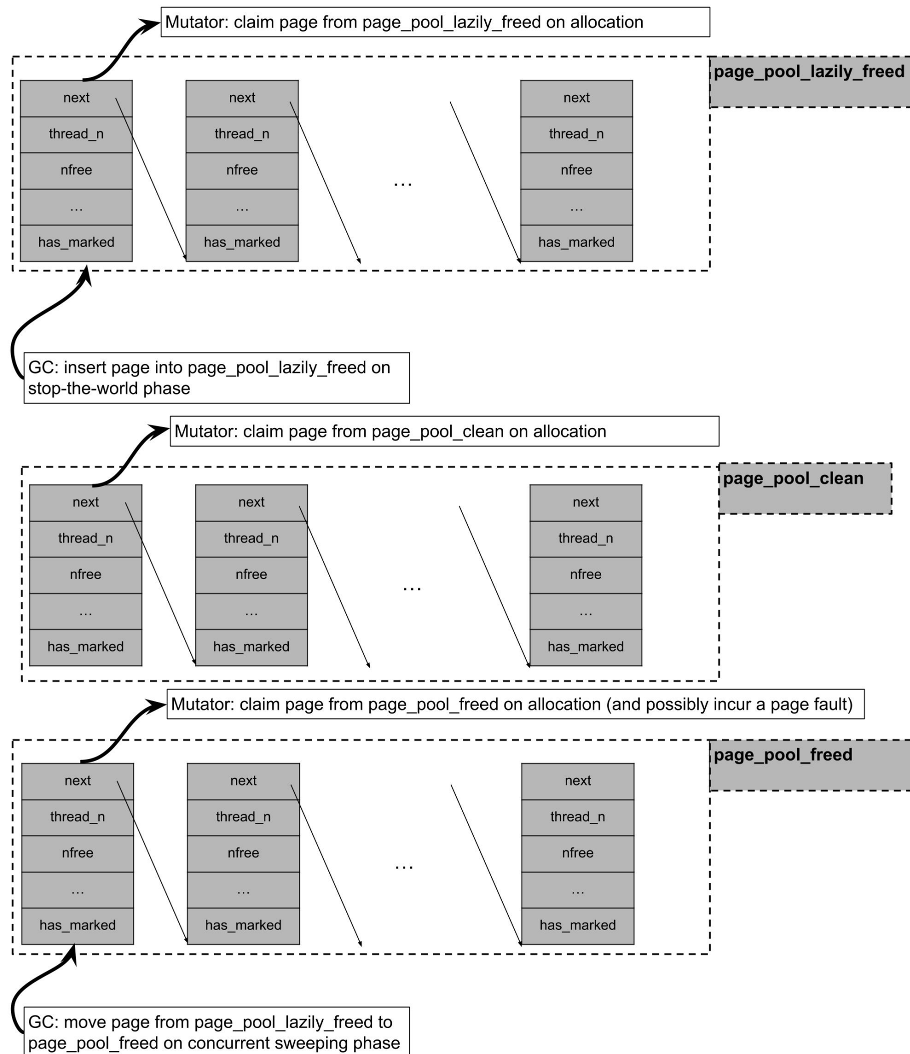

Figure 103.2: Diagram of tiered pool allocation

mark phase. A "full-sweep", however, causes the mark-bits of all objects to be reset, leading to all objects being traced in a subsequent mark phase. Objects are promoted to the next generation during every sweep phase they survive. On the mutator side, field writes are intercepted through a write barrier that pushes an object's address into a per-thread remembered set if the object is in the last generation, and if the object at the field being written is not. Objects in this remembered set are then traced during the mark phase.

Sweeping

Sweeping of object pools for Julia may fall into two categories: if a given page managed by the pool allocator contains at least one live object, then a free-list must be threaded through its dead objects; if a given page contains no live objects at all, then its underlying physical memory may be returned to the operating system through, for instance, the use of madvise system calls on Linux.

The first category of sweeping is currently serial and performed in the stop-the-world phase. For the second category of sweeping, if concurrent page sweeping is enabled through the flag `--gcthreads=X,1` we

```
julia> using Test2
Precompiling Test2
  Progress [>                                        ]  0/2
  ◒ Test1 Waiting for background task / IO / timer. Interrupt to inspect
```

Figure 103.3: Screenshot of precompilation hang

perform the madvise system calls in a background sweeper thread, concurrently with the mutator threads. During the stop-the-world phase of the collector, pool allocated pages which contain no live objects are initially pushed into the `pool_page_lazily_freed`. The background sweeping thread is then woken up and is responsible for removing pages from `pool_page_lazily_freed`, calling madvise on them, and inserting them into `pool_page_freed`. As described above, `pool_page_lazily_freed` is also shared with mutator threads. This implies that on allocation-heavy multithreaded workloads, mutator threads would often avoid a page fault on allocation (coming from accessing a fresh mmaped page or accessing a madvised page) by directly allocating from a page in `pool_page_lazily_freed`, while the background sweeper thread needs to madvise a reduce number of pages given some of them were already claimed by the mutators.

Heuristics

GC heuristics tune the GC by changing the size of the allocation interval between garbage collections. If a GC was unproductive, then we increase the size of the allocation interval to allow objects more time to die. If a GC returns a lot of space we can shrink the interval. The goal is to find a steady state where we are allocating just about the same amount as we are collecting.

103.26 Fixing precompilation hangs due to open tasks or IO

On Julia 1.10 or higher, you might see the following message:

This may repeat. If it continues to repeat with no hints that it will resolve itself, you may have a "precompilation hang" that requires fixing. Even if it's transient, you might prefer to resolve it so that users will not be bothered by this warning. This page walks you through how to analyze and fix such issues.

If you follow the advice and hit `Ctrl-C`, you might see

```
^C Interrupted: Exiting precompilation...

  1 dependency had warnings during precompilation:
┌ Test1 [ac89d554-e2ba-40bc-bc5c-de68b658c982]
│  [pid 2745] waiting for IO to finish:
│   Handle type        uv_handle_t->data
│   timer              0x55580decd1e0->0x7f94c3a4c340
```

This message conveys two key pieces of information:

- the hang is occurring during precompilation of `Test1`, a dependency of `Test2` (the package we were trying to load with `using Test2`)
- during precompilation of `Test1`, Julia created a `Timer` object (use `?Timer` if you're unfamiliar with Timers) which is still open; until that closes, the process is hung

If this is enough of a hint for you to figure out how `timer = Timer(args...)` is being created, one good solution is to add `wait(timer)` if `timer` eventually finishes on its own, or `close(timer)` if you need to force-close it, before the final end of the module.

However, there are cases that may not be that straightforward. Usually the best option is to start by determining whether the hang is due to code in Test1 or whether it is due to one of Test1's dependencies:

- Option 1: `Pkg.add("Aqua")` and use `Aqua.test_persistent_tasks`. This should help you identify which package is causing the problem, after which the instructions below should be followed. If needed, you can create a `PkgId` as `Base.PkgId(UUID("..."), "Test1")`, where `...` comes from the `uuid` entry in `Test1/Project.toml`.
- Option 2: manually diagnose the source of the hang.

To manually diagnose:

1. `Pkg.develop("Test1")`
2. Comment out all the code `include`d or defined in `Test1`, *except* the `using/import` statements.
3. Try `using Test2` (or even `using Test1` assuming that hangs too) again

Now we arrive at a fork in the road: either

- the hang persists, indicating it is due to one of your dependencies
- the hang disappears, indicating that it is due to something in your code.

Diagnosing and fixing hangs due to a package dependency

Use a binary search to identify the problematic dependency: start by commenting out half your dependencies, then when you isolate which half is responsible comment out half of that half, etc. (You don't have to remove them from the project, just comment out the `using/import` statements.)

Once you've identified a suspect (here we'll call it `ThePackageYouThinkIsCausingTheProblem`), first try precompiling that package. If it also hangs during precompilation, continue chasing the problem backwards.

However, most likely `ThePackageYouThinkIsCausingTheProblem` will precompile fine. This suggests it's in the function `ThePackageYouThinkIsCausingTheProblem.__init__`, which does not run during precompilation of `ThePackageYouThinkIsCausingTheProblem` but *does* in any package that loads `ThePackageYouThinkIsCausingTheProblem`. To test this theory, set up a minimal working example (MWE), something like

```
(@v1.10) pkg> generate MWE
  Generating  project MWE:
    MWE\Project.toml
    MWE\src\MWE.jl
```

where the source code of `MWE.jl` is

```
module MWE
using ThePackageYouThinkIsCausingTheProblem
end
```

and you've added `ThePackageYouThinkIsCausingTheProblem` to MWE's dependencies.

If that MWE reproduces the hang, you've found your culprit: `ThePackageYouThinkIsCausingTheProblem.__init__` must be creating the `Timer` object. If the timer object can be safely `close`d, that's a good option. Otherwise, the most common solution is to avoid creating the timer while *any* package is being precompiled: add

```
ccall(:jl_generating_output, Cint, ()) == 1 && return nothing
```

as the first line of `ThePackageYouThinkIsCausingTheProblem.__init__`, and it will avoid doing any initialization in any Julia process whose purpose is to precompile packages.

Fixing package code to avoid hangs

Search your package for suggestive words (here like "Timer") and see if you can identify where the problem is being created. Note that a method *definition* like

```
maketimer() = Timer(timer -> println("hi"), 0; interval=1)
```

is not problematic in and of itself: it can cause this problem only if `maketimer` gets called while the module is being defined. This might be happening from a top-level statement such as

```
const GLOBAL_TIMER = maketimer()
```

or it might conceivably occur in a precompile workload.

If you struggle to identify the causative lines, then consider doing a binary search: comment out sections of your package (or `include` lines to omit entire files) until you've reduced the problem in scope.

Chapter 104

Developing/debugging Julia's C code

104.1 报告和分析崩溃（段错误）

So you managed to break Julia. Congratulations! Collected here are some general procedures you can undergo for common symptoms encountered when something goes awry. Including the information from these debugging steps can greatly help the maintainers when tracking down a segfault or trying to figure out why your script is running slower than expected.

If you've been directed to this page, find the symptom that best matches what you're experiencing and follow the instructions to generate the debugging information requested. Table of symptoms:

- Segfaults during bootstrap (`sysimg.jl`)
- Segfaults when running a script
- Errors during Julia startup
- Other generic segfaults or unreachables reached

Version/Environment info

No matter the error, we will always need to know what version of Julia you are running. When Julia first starts up, a header is printed out with a version number and date. Please also include the output of `versioninfo()` (exported from the `InteractiveUtils` standard library) in any report you create:

```
julia> using InteractiveUtils

julia> versioninfo()
Julia Version 1.10.9
Commit 5595d20a287 (2025-03-10 12:51 UTC)
Build Info:
  Official https://julialang.org/ release
Platform Info:
  OS: Linux (x86_64-linux-gnu)
  CPU: 4 × AMD EPYC 7763 64-Core Processor
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-15.0.7 (ORCJIT, znver3)
Threads: 1 default, 0 interactive, 1 GC (on 4 virtual cores)
```

Segfaults during bootstrap (`sysimg.jl`)

Segfaults toward the end of the `make` process of building Julia are a common symptom of something going wrong while Julia is preparsing the corpus of code in the `base/` folder. Many factors can contribute toward this process dying unexpectedly, however it is as often as not due to an error in the C-code portion of Julia, and as such must typically be debugged with a debug build inside of `gdb`. Explicitly:

Create a debug build of Julia:

```
$ cd <julia_root>
$ make debug
```

Note that this process will likely fail with the same error as a normal `make` incantation, however this will create a debug executable that will offer `gdb` the debugging symbols needed to get accurate backtraces. Next, manually run the bootstrap process inside of `gdb`:

```
$ cd base/
$ gdb -x ../contrib/debug_bootstrap.gdb
```

This will start `gdb`, attempt to run the bootstrap process using the debug build of Julia, and print out a backtrace if (when) it segfaults. You may need to hit `<enter>` a few times to get the full backtrace. Create a gist with the backtrace, the version info, and any other pertinent information you can think of and open a new issue on Github with a link to the gist.

Segfaults when running a script

The procedure is very similar to Segfaults during bootstrap (`sysimg.jl`). Create a debug build of Julia, and run your script inside of a debugged Julia process:

```
$ cd <julia_root>
$ make debug
$ gdb --args usr/bin/julia-debug <path_to_your_script>
```

Note that `gdb` will sit there, waiting for instructions. Type `r` to run the process, and `bt` to generate a backtrace once it segfaults:

```
(gdb) r
Starting program: /home/sabae/src/julia/usr/bin/julia-debug ./test.jl
...
(gdb) bt
```

Create a gist with the backtrace, the version info, and any other pertinent information you can think of and open a new issue on Github with a link to the gist.

Errors during Julia startup

Occasionally errors occur during Julia's startup process (especially when using binary distributions, as opposed to compiling from source) such as the following:

```
$ julia
exec: error -5
```

These errors typically indicate something is not getting loaded properly very early on in the bootup phase, and our best bet in determining what's going wrong is to use external tools to audit the disk activity of the `julia` process:

- On Linux, use `strace`:

  ```
  $ strace julia
  ```

- On OSX, use `dtruss`:

  ```
  $ dtruss -f julia
  ```

Create a gist with the `strace`/ `dtruss` output, the version info, and any other pertinent information and open a new issue on Github with a link to the gist.

Other generic segfaults or unreachables reached

As mentioned elsewhere, `julia` has good integration with `rr` for generating traces; this includes, on Linux, the ability to automatically run `julia` under `rr` and share the trace after a crash. This can be immensely helpful when debugging such crashes and is strongly encouraged when reporting crash issues to the JuliaLang/julia repo. To run `julia` under `rr` automatically, do:

```
julia --bug-report=rr
```

To generate the `rr` trace locally, but not share, you can do:

```
julia --bug-report=rr-local
```

Note that this is only works on Linux. The blog post on Time Travelling Bug Reporting has many more details.

Glossary

A few terms have been used as shorthand in this guide:

- `<julia_root>` refers to the root directory of the Julia source tree; e.g. it should contain folders such as base, deps, src, test, etc.....

104.2 gdb 调试提示

显示 Julia 变量

在 `gdb` 中, 任何 `jl_value_t*` 类型的变量 `obj` 的展示可以通过使用:

```
(gdb) call jl_(obj)
```

这个对象会在 julia 会话中展示，而不是在 gdb 会话中。这是一种行之有效的方式来发现由 Julia 的 C 代码操控的对象的类型和值。

同样，如果你在调试一些 Julia 内部的东西（比如 compiler.jl），你可以通过使用这些来打印 obj：

```
ccall(:jl_, Cvoid, (Any,), obj)
```

这是一种很好的方法，可以避免 Julia 的输出流初始化顺序引起的问题。

Julia 的 flisp 解释器使用 value_t 对象；能够通过 call fl_print(fl_ctx, ios_stdout, obj) 来展示。

有用的用于检查的 Julia 变量

While the addresses of many variables, like singletons, can be useful to print for many failures, there are a number of additional variables (see julia.h for a complete list) that are even more useful.

- (when in jl_apply_generic) mfunc and jl_uncompress_ast(mfunc->def, mfunc->code) :: for figuring out a bit about the call-stack
- jl_lineno and jl_filename :: for figuring out what line in a test to go start debugging from (or figure out how far into a file has been parsed)
- $1 :: not really a variable, but still a useful shorthand for referring to the result of the last gdb command (such as print)
- jl_options :: sometimes useful, since it lists all of the command line options that were successfully parsed
- jl_uv_stderr :: because who doesn’t like to be able to interact with stdio

Useful Julia functions for Inspecting those variables

- jl_gdblookup($rip) :: For looking up the current function and line. (use $eip on i686 platforms)
- jlbacktrace() :: For dumping the current Julia backtrace stack to stderr. Only usable after record_backtrace() has been called.
- jl_dump_llvm_value(Value*) :: For invoking Value->dump() in gdb, where it doesn’t work natively. For example, f->linfo->functionObject, f->linfo->specFunctionObject, and to_function(f->linfo).
- Type->dump() :: only works in lldb. Note: add something like ;1 to prevent lldb from printing its prompt over the output
- jl_eval_string("expr") :: for invoking side-effects to modify the current state or to lookup symbols
- jl_typeof(jl_value_t*) :: for extracting the type tag of a Julia value (in gdb, call macro define jl_typeof jl_typeof first, or pick something short like ty for the first arg to define a shorthand)

Inserting breakpoints for inspection from gdb

In your gdb session, set a breakpoint in jl_breakpoint like so:

```
(gdb) break jl_breakpoint
```

Then within your Julia code, insert a call to `jl_breakpoint` by adding

```
ccall(:jl_breakpoint, Cvoid, (Any,), obj)
```

where `obj` can be any variable or tuple you want to be accessible in the breakpoint.

It's particularly helpful to back up to the `jl_apply` frame, from which you can display the arguments to a function using, e.g.,

```
(gdb) call jl_(args[0])
```

Another useful frame is `to_function(jl_method_instance_t *li, bool cstyle)`. The `jl_method_instance_t*` argument is a struct with a reference to the final AST sent into the compiler. However, the AST at this point will usually be compressed; to view the AST, call `jl_uncompress_ast` and then pass the result to `jl_`:

```
#2  0x00007ffff7928bf7 in to_function (li=0x2812060, cstyle=false) at codegen.cpp:584
584          abort();
(gdb) p jl_(jl_uncompress_ast(li, li->ast))
```

Inserting breakpoints upon certain conditions

Loading a particular file

Let's say the file is `sysimg.jl`:

```
(gdb) break jl_load if strcmp(fname, "sysimg.jl")==0
```

Calling a particular method

```
(gdb) break jl_apply_generic if strcmp((char*)(jl_symbol_name)(jl_gf_mtable(F)->name),
↪  "method_to_break")==0
```

Since this function is used for every call, you will make everything 1000x slower if you do this.

Dealing with signals

Julia requires a few signals to function properly. The profiler uses `SIGUSR2` for sampling and the garbage collector uses `SIGSEGV` for threads synchronization. If you are debugging some code that uses the profiler or multiple threads, you may want to let the debugger ignore these signals since they can be triggered very often during normal operations. The command to do this in GDB is (replace `SIGSEGV` with `SIGUSR2` or other signals you want to ignore):

```
(gdb) handle SIGSEGV noprint nostop pass
```

The corresponding LLDB command is (after the process is started):

```
(lldb) pro hand -p true -s false -n false SIGSEGV
```

If you are debugging a segfault with threaded code, you can set a breakpoint on `jl_critical_error` (`sigdie_handler` should also work on Linux and BSD) in order to only catch the actual segfault rather than the GC synchronization points.

Debugging during Julia's build process (bootstrap)

Errors that occur during `make` need special handling. Julia is built in two stages, constructing `sys0` and `sys.ji`. To see what commands are running at the time of failure, use `make VERBOSE=1`.

At the time of this writing, you can debug build errors during the `sys0` phase from the `base` directory using:

```
julia/base$ gdb --args ../usr/bin/julia-debug -C native --build ../usr/lib/julia/sys0 sysimg.jl
```

You might need to delete all the files in `usr/lib/julia/` to get this to work.

You can debug the `sys.ji` phase using:

```
julia/base$ gdb --args ../usr/bin/julia-debug -C native --build ../usr/lib/julia/sys -J
↪  ../usr/lib/julia/sys0.ji sysimg.jl
```

By default, any errors will cause Julia to exit, even under gdb. To catch an error "in the act", set a breakpoint in `jl_error` (there are several other useful spots, for specific kinds of failures, including: `jl_too_few_args`, `jl_too_many_args`, and `jl_throw`).

Once an error is caught, a useful technique is to walk up the stack and examine the function by inspecting the related call to `jl_apply`. To take a real-world example:

```
Breakpoint 1, jl_throw (e=0x7ffdf42de400) at task.c:802
802 {
(gdb) p jl_(e)
ErrorException("auto_unbox: unable to determine argument type")
$2 = void
(gdb) bt 10
#0  jl_throw (e=0x7ffdf42de400) at task.c:802
#1  0x00007ffff65412fe in jl_error (str=0x7ffde56be000 <_j_str267> "auto_unbox:
   unable to determine argument type")
   at builtins.c:39
#2  0x00007ffde56bd01a in julia_convert_16886 ()
#3  0x00007ffff6541154 in jl_apply (f=0x7ffdf367f630, args=0x7fffffffc2b0, nargs=2) at julia.h:1281
...
```

The most recent `jl_apply` is at frame #3, so we can go back there and look at the AST for the function `julia_convert_16886`. This is the uniqued name for some method of `convert`. `f` in this frame is a `jl_function_t*`, so we can look at the type signature, if any, from the `specTypes` field:

```
(gdb) f 3
#3  0x00007ffff6541154 in jl_apply (f=0x7ffdf367f630, args=0x7fffffffc2b0, nargs=2) at julia.h:1281
1281            return f->fptr((jl_value_t*)f, args, nargs);
(gdb) p f->linfo->specTypes
$4 = (jl_tupletype_t *) 0x7ffdf39b1030
(gdb) p jl_( f->linfo->specTypes )
Tuple{Type{Float32}, Float64}           # <-- type signature for julia_convert_16886
```

Then, we can look at the AST for this function:

```
(gdb) p jl_( jl_uncompress_ast(f->linfo, f->linfo->ast) )
Expr(:lambda, Array{Any, 1}[:#s29, :x], Array{Any, 1}[Array{Any, 1}[], Array{Any, 1}[Array{Any,
↪  1}[:#s29, :Any, 0], Array{Any, 1}[:x, :Any, 0]], Array{Any, 1}[], 0], Expr(:body,
Expr(:line, 90, :float.jl)::Any,
Expr(:return, Expr(:call, :box, :Float32, Expr(:call, :fptrunc, :Float32,
↪  :x)::Any)::Any)::Any)::Any)::Any
```

Finally, and perhaps most usefully, we can force the function to be recompiled in order to step through the codegen process. To do this, clear the cached `functionObject` from the `jl_lamdbda_info_t*`:

```
(gdb) p f->linfo->functionObject
$8 = (void *) 0x1289d070
(gdb) set f->linfo->functionObject = NULL
```

Then, set a breakpoint somewhere useful (e.g. `emit_function`, `emit_expr`, `emit_call`, etc.), and run codegen:

```
(gdb) p jl_compile(f)
... # your breakpoint here
```

Debugging precompilation errors

Module precompilation spawns a separate Julia process to precompile each module. Setting a breakpoint or catching failures in a precompile worker requires attaching a debugger to the worker. The easiest approach is to set the debugger watch for new process launches matching a given name. For example:

```
(gdb) attach -w -n julia-debug
```

or:

```
(lldb) process attach -w -n julia-debug
```

Then run a script/command to start precompilation. As described earlier, use conditional breakpoints in the parent process to catch specific file-loading events and narrow the debugging window. (some operating systems may require alternative approaches, such as following each `fork` from the parent process)

Mozilla's Record and Replay Framework (rr)

Julia now works out of the box with rr, the lightweight recording and deterministic debugging framework from Mozilla. This allows you to replay the trace of an execution deterministically. The replayed execution's address spaces, register contents, syscall data etc are exactly the same in every run.

A recent version of rr (3.1.0 or higher) is required.

Reproducing concurrency bugs with rr

rr simulates a single-threaded machine by default. In order to debug concurrent code you can use `rr record --chaos` which will cause rr to simulate between one to eight cores, chosen randomly. You might therefore want to set `JULIA_NUM_THREADS=8` and rerun your code under rr until you have caught your bug.

104.3 在 Julia 中使用 Valgrind

Valgrind is a tool for memory debugging, memory leak detection, and profiling. This section describes things to keep in mind when using Valgrind to debug memory issues with Julia.

General considerations

By default, Valgrind assumes that there is no self modifying code in the programs it runs. This assumption works fine in most instances but fails miserably for a just-in-time compiler like `julia`. For this reason it is crucial to pass `--smc-check=all-non-file` to `valgrind`, else code may crash or behave unexpectedly (often in subtle ways).

In some cases, to better detect memory errors using Valgrind it can help to compile `julia` with memory pools disabled. The compile-time flag `MEMDEBUG` disables memory pools in Julia, and `MEMDEBUG2` disables memory pools in FemtoLisp. To build `julia` with both flags, add the following line to `Make.user`:

```
CFLAGS = -DMEMDEBUG -DMEMDEBUG2
```

Another thing to note: if your program uses multiple workers processes, it is likely that you want all such worker processes to run under Valgrind, not just the parent process. To do this, pass `--trace-children=yes` to `valgrind`.

Yet another thing to note: if using `valgrind` errors with `Unable to find compatible target in system image`, try rebuilding the sysimage with target `generic` or julia with `JULIA_CPU_TARGET=generic`.

Suppressions

Valgrind will typically display spurious warnings as it runs. To reduce the number of such warnings, it helps to provide a suppressions file to Valgrind. A sample suppressions file is included in the Julia source distribution at `contrib/valgrind-julia.supp`.

The suppressions file can be used from the `julia/` source directory as follows:

```
$ valgrind --smc-check=all-non-file --suppressions=contrib/valgrind-julia.supp ./julia progname.jl
```

Any memory errors that are displayed should either be reported as bugs or contributed as additional suppressions. Note that some versions of Valgrind are shipped with insufficient default suppressions, so that may be one thing to consider before submitting any bugs.

Running the Julia test suite under Valgrind

It is possible to run the entire Julia test suite under Valgrind, but it does take quite some time (typically several hours). To do so, run the following command from the `julia/test/` directory:

```
valgrind --smc-check=all-non-file --trace-children=yes
↪  --suppressions=$PWD/../contrib/valgrind-julia.supp ../julia runtests.jl all
```

If you would like to see a report of "definite" memory leaks, pass the flags `--leak-check=full --show-leak-kinds=definite` to `valgrind` as well.

Additional spurious warnings

This section covers Valgrind warnings which cannot be added to the suppressions file yet are nonetheless safe to ignore.

Unhandled rr system calls

Valgrind will emit a warning if it encounters any of the system calls that are specific to rr, the Record and Replay Framework. In particular, a warning about an unhandled `1008` syscall will be shown when julia tries to detect whether it is running under rr:

```
--xxxxxx-- WARNING: unhandled amd64-linux syscall: 1008
--xxxxxx-- You may be able to write your own handler.
--xxxxxx-- Read the file README_MISSING_SYSCALL_OR_IOCTL.
--xxxxxx-- Nevertheless we consider this a bug.  Please report
--xxxxxx-- it at http://valgrind.org/support/bug_reports.html.
```

This issue has been reported to the Valgrind developers as they have requested.

Caveats

Valgrind currently does not support multiple rounding modes, so code that adjusts the rounding mode will behave differently when run under Valgrind.

In general, if after setting `--smc-check=all-non-file` you find that your program behaves differently when run under Valgrind, it may help to pass `--tool=none` to `valgrind` as you investigate further. This will enable the minimal Valgrind machinery but will also run much faster than when the full memory checker is enabled.

104.4 External Profiler Support

Julia provides explicit support for some external tracing profilers, enabling you to obtain a high-level overview of the runtime's execution behavior.

The currently supported profilers are:

- Tracy
- Intel VTune (ITTAPI)

Adding New Zones

To add new zones, use the `JL_TIMING` macro. You can find numerous examples throughout the codebase by searching for `JL_TIMING`. To add a new type of zone you add it to `JL_TIMING_OWNERS` (and possibly `JL_TIMING_EVENTS`).

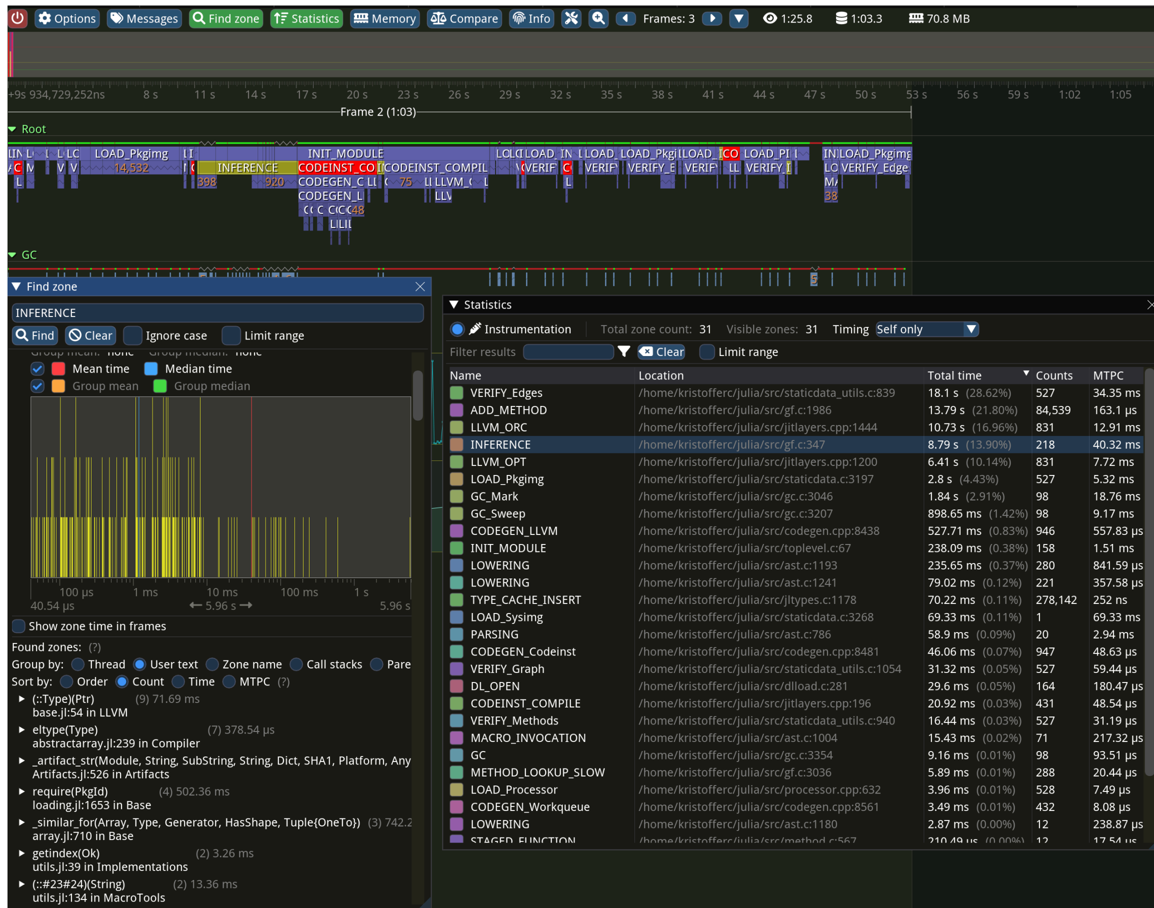

Figure 104.1: Typical Tracy usage

Dynamically Enabling and Disabling Zones

The JULIA_TIMING_SUBSYSTEMS environment variable allows you to enable or disable zones for a specific Julia run. For instance, setting the variable to +GC,-INFERENCE will enable the GC zones and disable the INFERENCE zones.

Tracy Profiler

Tracy is a flexible profiler that can be optionally integrated with Julia.

A typical Tracy session might look like this:

Building Julia with Tracy

To enable Tracy integration, build Julia with the extra option WITH_TRACY=1 in the Make.user file.

Installing the Tracy Profile Viewer

The easiest way to obtain the profile viewer is by adding the TracyProfiler_jll package and launching the profiler with:

```
run(TracyProfiler_jll.tracy())
```

Note

On macOS, you may want to set the `TRACY_DPI_SCALE` environment variable to `1.0` if the UI elements in the profiler appear excessively large.

To run a "headless" instance that saves the trace to disk, use `TracyProfiler_jll.capture() -o mytracefile.tracy` instead.

For information on using the Tracy UI, refer to the Tracy manual.

Profiling Julia with Tracy

A typical workflow for profiling Julia with Tracy involves starting Julia using:

```
JULIA_WAIT_FOR_TRACY=1 ./julia -e '...'
```

The environment variable ensures that Julia waits until it has successfully connected to the Tracy profiler before continuing execution. Afterward, use the Tracy profiler UI, click `Connect`, and Julia execution should resume and profiling should start.

Profiling package precompilation with Tracy

To profile a package precompilation process it is easiest to explicitly call into `Base.compilecache` with the package you want to precompile:

```
pkg = Base.identify_package("SparseArrays")
withenv("JULIA_WAIT_FOR_TRACY" => 1, "TRACY_PORT" => 9001) do
    Base.compilecache(pkg)
end
```

Here, we use a custom port for tracy which makes it easier to find the correct client in the Tracy UI to connect to.

Adding metadata to zones

The various `jl_timing_show_*` and `jl_timing_printf` functions can be used to attach a string (or strings) to a zone. For example, the trace zone for inference shows the method instance that is being inferred.

The `TracyCZoneColor` function can be used to set the color of a certain zone. Search through the codebase to see how it is used.

Viewing Tracy files in your browser

Visit https://topolarity.github.io/trace-viewer/ for an (experimental) web viewer for Tracy traces.

You can open a local `.tracy` file or provide a URL from the web (e.g. a file in a Github repo). If you load a trace file from the web, you can also share the page URL directly with others, enabling them to view the same trace.

Enabling stack trace samples

To enable call stack sampling in Tracy, build Julia with these options in your `Make.user` file:

```
WITH_TRACY := 1
WITH_TRACY_CALLSTACKS := 1
USE_BINARYBUILDER_LIBTRACYCLIENT := 0
```

You may also need to run `make -C deps clean-libtracyclient` to force a re-build of Tracy.

This feature has a significant impact on trace size and profiling overhead, so it is recommended to leave call stack sampling off when possible, especially if you intend to share your trace files online.

Note that the Julia JIT runtime does not yet have integration for Tracy's symbolification, so Julia functions will typically be unknown in these stack traces.

Intel VTune (ITTAPI) Profiler

This section is yet to be written.

104.5 Sanitizer support

Sanitizers can be used in custom Julia builds to make it easier to detect certain kinds of errors in Julia's internal C/C++ code.

Address Sanitizer: easy build

From a source-checkout of Julia, you should be able to build a version supporting address sanitization in Julia and LLVM as follows:

```
$ mkdir /tmp/julia
$ contrib/asan/build.sh /tmp/julia/
```

Here we've chosen `/tmp/julia` as a build directory, but you can choose whatever you wish. Once built, run the workload you wish to test with `/tmp/julia/julia`. Memory bugs will result in errors.

If you require customization or further detail, see the documentation below.

General considerations

Using Clang's sanitizers obviously requires you to use Clang (`USECLANG=1`), but there's another catch: most sanitizers require a run-time library, provided by the host compiler, while the instrumented code generated by Julia's JIT relies on functionality from that library. This implies that the LLVM version of your host compiler must match that of the LLVM library used within Julia.

An easy solution is to have a dedicated build folder for providing a matching toolchain, by building with `BUILD_LLVM_CLANG=1`. You can then refer to this toolchain from another build folder by specifying `USECLANG=1` while overriding the `CC` and `CXX` variables.

The sanitizers error out when they detect a shared library being opened using `RTLD_DEEPBIND` (ref: google/sanitizers#611). Since libblastrampoline by default uses `RTLD_DEEPBIND`, we need to set the environment variable `LBT_USE_RTLD_DEEPBIND=0` when using a sanitizer.

To use one of of the sanitizers set `SANITIZE=1` and then the appropriate flag for the sanitizer you want to use.

On macOS, this might need some extra flags also to work. Altogether, it might look like this, plus one or more of the SANITIZE_* flags listed below:

```
make -C deps USE_BINARYBUILDER_LLVM=0 LLVM_VER=svn stage-llvm

make -C src SANITIZE=1 USECLANG=1 \
    CC=~+/deps/scratch/llvm-svn/build_Release/bin/clang \
    CXX=~+/deps/scratch/llvm-svn/build_Release/bin/clang++ \
    CPPFLAGS="-isysroot $(xcode-select -p)/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk" \
    CXXFLAGS="-isystem $(xcode-select -p)/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1"
```

(or put these into your `Make.user`, so you don't need to remember them every time).

Address Sanitizer (ASAN)

For detecting or debugging memory bugs, you can use Clang's address sanitizer (ASAN). By compiling with SANITIZE_ADDRESS=1 you enable ASAN for the Julia compiler and its generated code. In addition, you can specify LLVM_SANITIZE=1 to sanitize the LLVM library as well. Note that these options incur a high performance and memory cost. For example, using ASAN for Julia and LLVM makes `testall1` take 8-10 times as long while using 20 times as much memory (this can be reduced to respectively a factor of 3 and 4 by using the options described below).

By default, Julia sets the `allow_user_segv_handler=1` ASAN flag, which is required for signal delivery to work properly. You can define other options using the `ASAN_OPTIONS` environment flag, in which case you'll need to repeat the default option mentioned before. For example, memory usage can be reduced by specifying `fast_unwind_on_malloc=0` and `malloc_context_size=2`, at the cost of backtrace accuracy. For now, Julia also sets `detect_leaks=0`, but this should be removed in the future.

Example setup

Step 1: Install toolchain

Checkout a Git worktree (or create out-of-tree build directory) at `$TOOLCHAIN_WORKTREE` and create a config file `$TOOLCHAIN_WORKTREE/Make.user` with

```
USE_BINARYBUILDER_LLVM=1
BUILD_LLVM_CLANG=1
```

Run:

```
cd $TOOLCHAIN_WORKTREE
make -C deps install-llvm install-clang install-llvm-tools
```

to install toolchain binaries in `$TOOLCHAIN_WORKTREE/usr/tools`

Step 2: Build Julia with ASAN

Checkout a Git worktree (or create out-of-tree build directory) at `$BUILD_WORKTREE` and create a config file `$BUILD_WORKTREE/Make.user` with

```
TOOLCHAIN=$(TOOLCHAIN_WORKTREE)/usr/tools

# use our new toolchain
USECLANG=1
override CC=$(TOOLCHAIN)/clang
override CXX=$(TOOLCHAIN)/clang++
export ASAN_SYMBOLIZER_PATH=$(TOOLCHAIN)/llvm-symbolizer

USE_BINARYBUILDER_LLVM=1

override SANITIZE=1
override SANITIZE_ADDRESS=1

# make the GC use regular malloc/frees, which are hooked by ASAN
override WITH_GC_DEBUG_ENV=1

# default to a debug build for better line number reporting
override JULIA_BUILD_MODE=debug

# make ASAN consume less memory
export
↪  ASAN_OPTIONS=detect_leaks=0:fast_unwind_on_malloc=0:allow_user_segv_handler=1:malloc_context_size=2

JULIA_PRECOMPILE=1

# tell libblastrampoline to not use RTLD_DEEPBIND
export LBT_USE_RTLD_DEEPBIND=0
```

Run:

```
cd $BUILD_WORKTREE
make debug
```

to build `julia-debug` with ASAN.

Memory Sanitizer (MSAN)

For detecting use of uninitialized memory, you can use Clang's memory sanitizer (MSAN) by compiling with `SANITIZE_MEMORY=1`.

Thread Sanitizer (TSAN)

For debugging data-races and other threading related issues you can use Clang's thread sanitizer (TSAN) by compiling with `SANITIZE_THREAD=1`.

104.6 Instrumenting Julia with DTrace, and bpftrace

DTrace and bpftrace are tools that enable lightweight instrumentation of processes. You can turn the instrumentation on and off while the process is running, and with instrumentation off the overhead is minimal.

Julia 1.8

Support for probes was added in Julia 1.8

Note

This documentation has been written from a Linux perspective, most of this should hold on Mac OS/Darwin and FreeBSD.

Enabling support

On Linux install the `systemtap` package that has a version of `dtrace` and create a `Make.user` file containing

```
WITH_DTRACE=1
```

to enable USDT probes.

Verifying

```
> readelf -n usr/lib/libjulia-internal.so.1

Displaying notes found in: .note.gnu.build-id
  Owner                Data size  Description
  GNU                  0x00000014 NT_GNU_BUILD_ID (unique build ID bitstring)
    Build ID: 57161002f35548772a87418d2385c284ceb3ead8

Displaying notes found in: .note.stapsdt
  Owner                Data size  Description
  stapsdt              0x00000029 NT_STAPSDT (SystemTap probe descriptors)
    Provider: julia
    Name: gc__begin
    Location: 0x000000000013213e, Base: 0x00000000002bb4da, Semaphore: 0x0000000000346cac
    Arguments:
  stapsdt              0x00000032 NT_STAPSDT (SystemTap probe descriptors)
    Provider: julia
    Name: gc__stop_the_world
    Location: 0x0000000000132144, Base: 0x00000000002bb4da, Semaphore: 0x0000000000346cae
    Arguments:
  stapsdt              0x00000027 NT_STAPSDT (SystemTap probe descriptors)
    Provider: julia
    Name: gc__end
    Location: 0x000000000013214a, Base: 0x00000000002bb4da, Semaphore: 0x0000000000346cb0
    Arguments:
  stapsdt              0x0000002d NT_STAPSDT (SystemTap probe descriptors)
    Provider: julia
    Name: gc__finalizer
    Location: 0x0000000000132150, Base: 0x00000000002bb4da, Semaphore: 0x0000000000346cb2
    Arguments:
```

Adding probes in libjulia

Probes are declared in dtraces format in the file `src/uprobes.d`. The generated header file is included in `src/julia_internal.h` and if you add probes you should provide a noop implementation there.

The header will contain a semaphore `*_ENABLED` and the actual call to the probe. If the probe arguments are expensive to compute you should first check if the probe is enabled and then compute the arguments and call the probe.

```
if (JL_PROBE_{PROBE}_ENABLED())
  auto expensive_arg = ...;
  JL_PROBE_{PROBE}(expensive_arg);
```

If your probe has no arguments it is preferred to not include the semaphore check. With USDT probes enabled the cost of a semaphore is a memory load, irrespective of the fact that the probe is enabled or not.

```
#define JL_PROBE_GC_BEGIN_ENABLED() __builtin_expect (julia_gc__begin_semaphore, 0)
__extension__ extern unsigned short julia_gc__begin_semaphore __attribute__ ((unused))
↪  __attribute__ ((section (".probes")));
```

Whereas the probe itself is a noop sled that will be patched to a trampoline to the probe handler.

Available probes

GC probes

1. `julia:gc__begin`: GC begins running on one thread and triggers stop-the-world.

2. `julia:gc__stop_the_world`: All threads have reached a safepoint and GC runs.

3. `julia:gc__mark__begin`: Beginning the mark phase

4. `julia:gc__mark_end(scanned_bytes, perm_scanned)`: Mark phase ended

5. `julia:gc__sweep_begin(full)`: Starting sweep

6. `julia:gc__sweep_end`: Sweep phase finished

7. `julia:gc__end`: GC is finished, other threads continue work

8. `julia:gc__finalizer`: Initial GC thread has finished running finalizers

Task runtime probes

1. `julia:rt__run__task(task)`: Switching to task `task` on current thread.

2. `julia:rt__pause__task(task)`: Switching from task `task` on current thread.

3. `julia:rt__new__task(parent, child)`: Task `parent` created task `child` on current thread.

4. `julia:rt__start__task(task)`: Task `task` started for the first time with a new stack.

5. `julia:rt__finish__task(task)`: Task `task` finished and will no longer execute.

6. `julia:rt__start__process__events(task)`: Task `task` started processing libuv events.

7. `julia:rt__finish__process__events(task)`: Task `task` finished processing libuv events.

Task queue probes

1. `julia:rt__taskq__insert(ptls, task)`: Thread `ptls` attempted to insert `task` into a PARTR multiq.
2. `julia:rt__taskq__get(ptls, task)`: Thread `ptls` popped `task` from a PARTR multiq.

Thread sleep/wake probes

1. `julia:rt__sleep__check__wake(ptls, old_state)`: Thread (PTLS `ptls`) waking up, previously in state `old_state`.
2. `julia:rt__sleep__check__wakeup(ptls)`: Thread (PTLS `ptls`) woke itself up.
3. `julia:rt__sleep__check__sleep(ptls)`: Thread (PTLS `ptls`) is attempting to sleep.
4. `julia:rt__sleep__check__taskq__wake(ptls)`: Thread (PTLS `ptls`) fails to sleep due to tasks in PARTR multiq.
5. `julia:rt__sleep__check__task__wake(ptls)`: Thread (PTLS `ptls`) fails to sleep due to tasks in Base workqueue.
6. `julia:rt__sleep__check__uv__wake(ptls)`: Thread (PTLS `ptls`) fails to sleep due to libuv wakeup.

Probe usage examples

GC stop-the-world latency

An example `bpftrace` script is given in `contrib/gc_stop_the_world_latency.bt` and it creates a histogram of the latency for all threads to reach a safepoint.

Running this Julia code, with `julia -t 2`

```
using Base.Threads

fib(x) = x <= 1 ? 1 : fib(x-1) + fib(x-2)

beaver = @spawn begin
    while true
        fib(30)
        # This safepoint is necessary until #41616, since otherwise this
        # loop will never yield to GC.
        GC.safepoint()
    end
end

allocator = @spawn begin
    while true
        zeros(1024)
    end
end

wait(allocator)
```

and in a second terminal

```
> sudo contrib/bpftrace/gc_stop_the_world_latency.bt
Attaching 4 probes...
Tracing Julia GC Stop-The-World Latency... Hit Ctrl-C to end.
^C


@usecs[1743412]:
[4, 8)               971 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[8, 16)              837 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@        |
[16, 32)             129 |@@@@@@                                              |
[32, 64)              10 |                                                    |
[64, 128)              1 |                                                    |
```

We can see the latency distribution of the stop-the-world phase in the executed Julia process.

Task spawn monitor

It’s sometimes useful to know when a task is spawning other tasks. This is very easy to see with `rt__new__task`. The first argument to the probe, `parent`, is the existing task which is creating a new task. This means that if you know the address of the task you want to monitor, you can easily just look at the tasks that that specific task spawned. Let’s see how to do this; first let’s start a Julia session and get the PID and REPL’s task address:

```
> julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.6.2 (2021-07-14)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

1> getpid()
997825

2> current_task()
Task (runnable) @0x00007f524d088010
```

Now we can start `bpftrace` and have it monitor `rt__new__task` for *only* this parent:

```
sudo bpftrace -p 997825 -e 'usdt:usr/lib/libjulia-internal.so:julia:rt__new__task /arg0==0x00007f524d088010
printf("Task: %x\n", arg0); }'
```

(Note that in the above, `arg0` is the first argument, `parent`).

And if we spawn a single task:

```
@async 1+1
```

we see this task being created:

```
Task: 4d088010
```

However, if we spawn a bunch of tasks from that newly-spawned task:

```
@async for i in 1:10
   @async 1+1
end
```

we still only see one task from `bpftrace`:

`Task: 4d088010`

and it's still the same task we were monitoring! Of course, we can remove this filter to see *all* newly-created tasks just as easily:

`sudo bpftrace -p 997825 -e 'usdt:usr/lib/libjulia-internal.so:julia:rt__new__task { printf("Task: %x\n", arg0); }'`

```
Task: 4d088010
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
Task: 4dc4e290
```

We can see our root task, and the newly-spawned task as the parent of the ten even newer tasks.

Thundering herd detection

Task runtimes can often suffer from the "thundering herd" problem: when some work is added to a quiet task runtime, all threads may be woken up from their slumber, even if there isn't enough work for each thread to process. This can cause extra latency and CPU cycles while all threads awaken (and simultaneously go back to sleep, not finding any work to execute).

We can see this problem illustrated with `bpftrace` quite easily. First, in one terminal we start Julia with multiple threads (6 in this example), and get the PID of that process:

```
> julia -t 6
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.6.2 (2021-07-14)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

1> getpid()
997825
```

And in another terminal we start `bpftrace` monitoring our process, specifically probing the `rt__sleep__check__wake` hook:

`sudo bpftrace -p 997825 -e 'usdt:usr/lib/libjulia-internal.so:julia:rt__sleep__check__wake { printf("Thread wake up! %x\n", arg0); }'`

Now, we create and execute a single task in Julia:

`Threads.@spawn 1+1`

And in `bpftrace` we see printed out something like:

```
Thread wake up! 3f926100
Thread wake up! 3ebd5140
Thread wake up! 3f876130
Thread wake up! 3e2711a0
Thread wake up! 3e312190
```

Even though we only spawned a single task (which only one thread could process at a time), we woke up all of our other threads! In the future, a smarter task runtime might only wake up a single thread (or none at all; the spawning thread could execute this task!), and we should see this behavior go away.

Task Monitor with BPFnative.jl

BPFnative.jl is able to attach to USDT probe points just like `bpftrace`. There is a demo available for monitoring the task runtime, GC, and thread sleep/wake transitions here.

Notes on using `bpftrace`

An example probe in the bpftrace format looks like:

```
usdt:usr/lib/libjulia-internal.so:julia:gc__begin
{
  @start[pid] = nsecs;
}
```

The probe declaration takes the kind `usdt`, then either the path to the library or the PID, the provider name `julia` and the probe name `gc__begin`. Note that I am using a relative path to the `libjulia-internal.so`, but this might need to be an absolute path on a production system.

Useful references:

- Julia Evans blog on Linux tracing systems
- LWN article on USDT and BPF
- GDB support for probes
- Brendan Gregg – Linux Performance

Chapter 105

Building Julia

105.1 Building Julia (Detailed)

Downloading the Julia source code

If you are behind a firewall, you may need to use the `https` protocol instead of the `git` protocol:

```
git config --global url."https://".insteadOf git://
```

Be sure to also configure your system to use the appropriate proxy settings, e.g. by setting the `https_proxy` and `http_proxy` variables.

Building Julia

When compiled the first time, the build will automatically download pre-built external dependencies. If you prefer to build all the dependencies on your own, or are building on a system that cannot access the network during the build process, add the following in `Make.user`:

```
USE_BINARYBUILDER=0
```

Building Julia requires 5GiB if building all dependencies and approximately 4GiB of virtual memory.

To perform a parallel build, use `make -j N` and supply the maximum number of concurrent processes. If the defaults in the build do not work for you, and you need to set specific make parameters, you can save them in `Make.user`, and place the file in the root of your Julia source. The build will automatically check for the existence of `Make.user` and use it if it exists.

You can create out-of-tree builds of Julia by specifying `make O=<build-directory> configure` on the command line. This will create a directory mirror, with all of the necessary Makefiles to build Julia, in the specified directory. These builds will share the source files in Julia and `deps/srccache`. Each out-of-tree build directory can have its own `Make.user` file to override the global `Make.user` file in the top-level folder.

If everything works correctly, you will see a Julia banner and an interactive prompt into which you can enter expressions for evaluation. (Errors related to libraries might be caused by old, incompatible libraries sitting around in your PATH. In this case, try moving the `julia` directory earlier in the PATH). Note that most of the instructions above apply to unix systems.

To run julia from anywhere you can:

- add an alias (in bash: `echo "alias julia='/path/to/install/folder/bin/julia'" >> ~/.bashrc && source ~/.bashrc`), or
- add a soft link to the `julia` executable in the `julia` directory to `/usr/local/bin` (or any suitable directory already in your path), or
- add the `julia` directory to your executable path for this shell session (in `bash`: `export PATH="$(pwd):$PATH"` ; in `csh` or `tcsh`:

`set path= ( $path $cwd )`), or

- add the `julia` directory to your executable path permanently (e.g. in `.bash_profile`), or
- write `prefix=/path/to/install/folder` into `Make.user` and then run `make install`. If there is a version of Julia already installed in this folder, you should delete it before running `make install`.

Some of the options you can set to control the build of Julia are listed and documented at the beginning of the file `Make.inc`, but you should never edit it for this purpose, use `Make.user` instead.

Julia's Makefiles define convenient automatic rules called `print-<VARNAME>` for printing the value of variables, replacing `<VARNAME>` with the name of the variable to print the value of. For example

```
$ make print-JULIA_PRECOMPILE
JULIA_PRECOMPILE=1
```

These rules are useful for debugging purposes.

Now you should be able to run Julia like this:

```
julia
```

If you are building a Julia package for distribution on Linux, macOS, or Windows, take a look at the detailed notes in distributing.md.

Updating an existing source tree

If you have previously downloaded `julia` using `git clone`, you can update the existing source tree using `git pull` rather than starting anew:

```
cd julia
git pull && make
```

Assuming that you had made no changes to the source tree that will conflict with upstream updates, these commands will trigger a build to update to the latest version.

General troubleshooting

1. Over time, the base library may accumulate enough changes such that the bootstrapping process in building the system image will fail. If this happens, the build may fail with an error like

```
*** This error is usually fixed by running 'make clean'. If the error persists, try 'make
↪  cleanall' ***
```

As described, running `make clean && make` is usually sufficient. Occasionally, the stronger cleanup done by `make cleanall` is needed.

2. New versions of external dependencies may be introduced which may occasionally cause conflicts with existing builds of older versions.

   a. Special make targets exist to help wipe the existing build of a dependency. For example, `make -C deps clean-llvm` will clean out the existing build of `llvm` so that `llvm` will be rebuilt from the downloaded source distribution the next time make is called. `make -C deps distclean-llvm` is a stronger wipe which will also delete the downloaded source distribution, ensuring that a fresh copy of the source distribution will be downloaded and that any new patches will be applied the next time make is called.

   b. To delete existing binaries of `julia` and all its dependencies, delete the `./usr` directory *in the source tree*.

3. If you've updated macOS recently, be sure to run `xcode-select --install` to update the command line tools. Otherwise, you could run into errors for missing headers and libraries, such as `ld: library not found for -lcrt1.10.6.o`.

4. If you've moved the source directory, you might get errors such as `CMake Error: The current CMakeCache.txt directory ... is different than the directory ... where CMakeCache.txt was created.`, in which case you may delete the offending dependency under deps

5. In extreme cases, you may wish to reset the source tree to a pristine state. The following git commands may be helpful:

```
git reset --hard #Forcibly remove any changes to any files under version control
git clean -x -f -d #Forcibly remove any file or directory not under version control
```

To avoid losing work, make sure you know what these commands do before you run them. `git` will not be able to undo these changes!

Platform-Specific Notes

Notes for various operating systems:

- Linux
- macOS
- Windows
- FreeBSD

Notes for various architectures:

- ARM

Required Build Tools and External Libraries

Building Julia requires that the following software be installed:

- **[GNU make]** —building dependencies.
- **[gcc & g++][gcc]** (>= 7.1) or **[Clang][clang]** (>= 5.0, >= 9.3 for Apple Clang) —compiling and linking C, C++.
- **[libatomic][gcc]** —provided by **[gcc]** and needed to support atomic operations.
- **[python]** (>=2.7) —needed to build LLVM.
- **[gfortran]** —compiling and linking Fortran libraries.
- **[perl]** —preprocessing of header files of libraries.
- **[wget]**, **[curl]**, or **[fetch]** (FreeBSD) —to automatically download external libraries.
- **[m4]** —needed to build GMP.
- **[awk]** —helper tool for Makefiles.
- **[patch]** —for modifying source code.
- **[cmake]** (>= 3.4.3) —needed to build `libgit2`.
- **[pkg-config]** —needed to build `libgit2` correctly, especially for proxy support.
- **[powershell]** (>= 3.0) —necessary only on Windows.
- **[which]** —needed for checking build dependencies.

On Debian-based distributions (e.g. Ubuntu), you can easily install them with `apt-get`:

```
sudo apt-get install build-essential libatomic1 python gfortran perl wget m4 cmake pkg-config curl
```

Julia uses the following external libraries, which are automatically downloaded (or in a few cases, included in the Julia source repository) and then compiled from source the first time you run `make`. The specific version numbers of these libraries that Julia uses are listed in `deps/$(libname).version`:

- **[LLVM]** (15.0 + patches) —compiler infrastructure (see note below).
- **[FemtoLisp]** —packaged with Julia source, and used to implement the compiler front-end.
- **[libuv]** (custom fork) —portable, high-performance event-based I/O library.
- **[OpenLibm]** —portable libm library containing elementary math functions.
- **[DSFMT]** —fast Mersenne Twister pseudorandom number generator library.
- **[OpenBLAS]** —fast, open, and maintained [basic linear algebra subprograms (BLAS)]
- **[LAPACK]** —library of linear algebra routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems.
- **[MKL]** (optional) –OpenBLAS and LAPACK may be replaced by Intel's MKL library.

- **[SuiteSparse]** —library of linear algebra routines for sparse matrices.
- **[PCRE]** —Perl-compatible regular expressions library.
- **[GMP]** —GNU multiple precision arithmetic library, needed for `BigInt` support.
- **[MPFR]** —GNU multiple precision floating point library, needed for arbitrary precision floating point (`BigFloat`) support.
- **[libgit2]** —Git linkable library, used by Julia's package manager.
- **[curl]** —libcurl provides download and proxy support.
- **[libssh2]** —library for SSH transport, used by libgit2 for packages with SSH remotes.
- **[mbedtls]** —library used for cryptography and transport layer security, used by libssh2
- **[utf8proc]** —a library for processing UTF-8 encoded Unicode strings.
- **[LLVM libunwind]** —LLVM's fork of [libunwind], a library that determines the call-chain of a program.
- **[ITTAPI]** —Intel's Instrumentation and Tracing Technology and Just-In-Time API.

[GNU make]: https://www.gnu.org/software/make [patch]: https://www.gnu.org/software/patch [wget]: https://www.gnu.org/softw [m4]: https://www.gnu.org/software/m4 [awk]: https://www.gnu.org/software/gawk [gcc]: https://gcc.gnu.org [clang]: https://clang.llvm.org [python]: https://www.python.org/ [gfortran]: https://gcc.gnu.org/fortran/ [curl]: https://curl.haxx.se [fetch]: https://www.freebsd.org/cgi/man.cgi?fetch(1) [perl]: https://www.perl.org [cmake]: https://www.cmake.org [OpenLibm]: https://github.com/JuliaLang/openlibm [DSFMT]: https://github.com/MersenneTwister-Lab/dSFMT [OpenBLAS]: https://github.com/xianyi/OpenBLAS [LAPACK]: https://www.netlib.org/lapack [MKL]: https://software.intel.com/en-us/articles/intel-mkl [SuiteSparse]: https://people.engr.tamu.edu/davis/suitesparse.html [PCRE]: https://www.pcre.org [LLVM]: https://www.llvm.org [LLVM libunwind]: https://github.com/llvm/llvm-project/tree/main/libunw [FemtoLisp]: https://github.com/JeffBezanson/femtolisp [GMP]: https://gmplib.org [MPFR]: https://www.mpfr.org [libuv]: https://github.com/JuliaLang/libuv [libgit2]: https://libgit2.org/ [utf8proc]: https://julialang.org/utf8proc/ [libunwind]: https://www.nongnu.org/libunwind [libssh2]: https://www.libssh2.org [mbedtls]: https://tls.mbed.org/ [pkg-config]: https://www.freedesktop.org/wiki/Software/pkg-config/ [powershell]: https://docs.microsoft.com/en-us/powershell/scripting/wmf/overview [which]: https://carlowood.github.io/which/ [ITTAPI]: https://github.com/intel/ittapi

Build dependencies

If you already have one or more of these packages installed on your system, you can prevent Julia from compiling duplicates of these libraries by passing `USE_SYSTEM_...=1` to `make` or adding the line to `Make.user`. The complete list of possible flags can be found in `Make.inc`.

Please be aware that this procedure is not officially supported, as it introduces additional variability into the installation and versioning of the dependencies, and is recommended only for system package maintainers. Unexpected compile errors may result, as the build system will do no further checking to ensure the proper packages are installed.

LLVM

The most complicated dependency is LLVM, for which we require additional patches from upstream (LLVM is not backward compatible).

For packaging Julia with LLVM, we recommend either:

- bundling a Julia-only LLVM library inside the Julia package, or

- adding the patches to the LLVM package of the distribution.
  - A complete list of patches is available in on Github see the `julia-release/15.x` branch.
  - The only Julia-specific patch is the lib renaming (`llvm7-symver-jlprefix.patch`), which should *not* be applied to a system LLVM.
  - The remaining patches are all upstream bug fixes, and have been contributed into upstream LLVM.

Using an unpatched or different version of LLVM will result in errors and/or poor performance. You can build a different version of LLVM from a remote Git repository with the following options in the `Make.user` file:

```
# Force source build of LLVM
USE_BINARYBUILDER_LLVM = 0
# Use Git for fetching LLVM source code
# this is either `1` to get all of them
DEPS_GIT = 1
# or a space-separated list of specific dependencies to download with git
DEPS_GIT = llvm

# Other useful options:
#URL of the Git repository you want to obtain LLVM from:
#  LLVM_GIT_URL = ...
#Name of the alternate branch to clone from git
#  LLVM_BRANCH = julia-16.0.6-0
#SHA hash of the alterate commit to check out automatically
#  LLVM_SHA1 = $(LLVM_BRANCH)
#List of LLVM targets to build.  It is strongly recommended to keep at least all the
#default targets listed in `deps/llvm.mk`, even if you don't necessarily need all of them.
#  LLVM_TARGETS = ...
#Use ccache for faster recompilation in case you need to restart a build.
#  USECCACHE = 1
#  CMAKE_GENERATOR=Ninja
#  LLVM_ASSERTIONS=1
#  LLVM_DEBUG=Symbols
```

The various build phases are controlled by specific files:

- `deps/llvm.version` : touch or change to checkout a new version, `make get-llvm check-llvm`
- `deps/srccache/llvm/source-extracted` : result of `make extract-llvm`
- `deps/llvm/build_Release*/build-configured` : result of `make configure-llvm`
- `deps/llvm/build_Release*/build-configured` : result of `make compile-llvm`
- `usr-staging/llvm/build_Release*.tgz` : result of `make stage-llvm` (regenerate with `make reinstall-llvm`)
- `usr/manifest/llvm` : result of `make install-llvm` (regenerate with `make uninstall-llvm`)
- `make version-check-llvm` : runs every time to warn the user if there are local modifications

Though Julia can be built with newer LLVM versions, support for this should be regarded as experimental and not suitable for packaging.

libuv

Julia uses a custom fork of libuv. It is a small dependency, and can be safely bundled in the same package as Julia, and will not conflict with the system library. Julia builds should *not* try to use the system libuv.

BLAS and LAPACK

As a high-performance numerical language, Julia should be linked to a multi-threaded BLAS and LAPACK, such as OpenBLAS or ATLAS, which will provide much better performance than the reference `libblas` implementations which may be default on some systems.

Source distributions of releases

Each pre-release and release of Julia has a "full" source distribution and a "light" source distribution.

The full source distribution contains the source code for Julia and all dependencies so that it can be built from source without an internet connection. The light source distribution does not include the source code of dependencies.

For example, `julia-1.0.0.tar.gz` is the light source distribution for the `v1.0.0` release of Julia, while `julia-1.0.0-full.tar.gz` is the full source distribution.

Building Julia from source with a Git checkout of a stdlib

If you need to build Julia from source with a Git checkout of a stdlib, then use `make DEPS_GIT=NAME_OF_STDLIB` when building Julia.

For example, if you need to build Julia from source with a Git checkout of Pkg, then use `make DEPS_GIT=Pkg` when building Julia. The `Pkg` repo is in `stdlib/Pkg`, and created initially with a detached `HEAD`. If you're doing this from a pre-existing Julia repository, you may need to `make clean` beforehand.

If you need to build Julia from source with Git checkouts of more than one stdlib, then `DEPS_GIT` should be a space-separated list of the stdlib names. For example, if you need to build Julia from source with a Git checkout of Pkg, Tar, and Downloads, then use `make DEPS_GIT='Pkg Tar Downloads'` when building Julia.

Building an "assert build" of Julia

An "assert build" of Julia is a build that was built with both `FORCE_ASSERTIONS=1` and `LLVM_ASSERTIONS=1`. To build an assert build, define both of the following variables in your `Make.user` file:

```
FORCE_ASSERTIONS=1
LLVM_ASSERTIONS=1
```

Please note that assert builds of Julia will be slower than regular (non-assert) builds.

Building 32-bit Julia on a 64-bit machine

Occasionally, bugs specific to 32-bit architectures may arise, and when this happens it is useful to be able to debug the problem on your local machine. Since most modern 64-bit systems support running programs built for 32-bit ones, if you don't have to recompile Julia from source (e.g. you mainly need to inspect the behavior of a 32-bit Julia without having to touch the C code), you can likely use a 32-bit build of Julia for your system that you can obtain from the official downloads page. However, if you do need to recompile Julia from source one option is to use a Docker container of a 32-bit system. At least for now, building a 32-bit version of Julia is relatively straightforward using ubuntu 32-bit docker images. In brief, after setting up `docker` here are the required steps:

```
$ docker pull i386/ubuntu
$ docker run --platform i386 -i -t i386/ubuntu /bin/bash
```

At this point you should be in a 32-bit machine console (note that `uname` reports the host architecture, so will still say 64-bit, but this will not affect the Julia build). You can add packages and compile code; when you `exit`, all the changes will be lost, so be sure to finish your analysis in a single session or set up a copy/pastable script you can use to set up your environment.

From this point, you should

```
# apt update
```

(Note that `sudo` isn't installed, but neither is it necessary since you are running as `root`, so you can omit `sudo` from all commands.)

Then add all the build dependencies, a console-based editor of your choice, `git`, and anything else you'll need (e.g., `gdb`, `rr`, etc). Pick a directory to work in and `git clone` Julia, check out the branch you wish to debug, and build Julia as usual.

Update the version number of a dependency

There are two types of builds

1. Build everything (`deps/` and `src/`) from source code. (Add `USE_BINARYBUILDER=0` to `Make.user`, see Building Julia)
2. Build from source (`src/`) with pre-compiled dependencies (default)

When you want to update the version number of a dependency in `deps/`, you may want to use the following checklist:

```
### Check list

Version numbers:
- [ ] `deps/$(libname).version`: `LIBNAME_VER`, `LIBNAME_BRANCH`, `LIBNAME_SHA1` and
↪  `LIBNAME_JLL_VER`
- [ ] `stdlib/$(LIBNAME_JLL_NAME)_jll/Project.toml`: `version`

Checksum:
- [ ] `deps/checksums/$(libname)`
- [ ] `deps/checksums/$(LIBNAME_JLL_NAME)-*/`: `md5` and `sha512`

Patches:
- [ ] `deps/$(libname).mk`
- [ ] `deps/patches/$(libname)-*.patch`
```

Note:

- For specific dependencies, some items in the checklist may not exist.
- For checksum file, it may be **a single file** without a suffix, or **a folder** containing two files.

Example: `OpenLibm`

1. Update Version numbers in `deps/openlibm.version`

   - `OPENLIBM_VER := 0.X.Y`
   - `OPENLIBM_BRANCH = v0.X.Y`
   - `OPENLIBM_SHA1 = new-sha1-hash`

2. Update Version number in `stdlib/OpenLibm_jll/Project.toml`

   - `version = "0.X.Y+0"`

3. Update checksums in `deps/checksums/openlibm`

   - `make -f contrib/refresh_checksums.mk openlibm`

4. Check if the patch files `deps/patches/openlibm-*.patch` exist

   - if patches don't exist, skip.
   - if patches exist, check if they have been merged into the new version and need to be removed. When deleting a patch, remember to modify the corresponding Makefile file (`deps/openlibm.mk`).

105.2 Linux

- GCC version 4.7 or later is required to build Julia.
- To use external shared libraries not in the system library search path, set `USE_SYSTEM_XXX=1` and `LDFLAGS=-Wl,-rpath,/pat` in `Make.user`.
- Instead of setting `LDFLAGS`, putting the library directory into the environment variable `LD_LIBRARY_PATH` (at both compile and run time) also works.
- The `USE_SYSTEM_*` flags should be used with caution. These are meant only for troubleshooting, porting, and packaging, where package maintainers work closely with the Julia developers to make sure that Julia is built correctly. Production use cases should use the officially provided binaries. Issues arising from the use of these flags will generally not be accepted.
- See also the external dependencies.

Architecture Customization

Julia can be built for a non-generic architecture by configuring the `ARCH` Makefile variable in a `Make.user` file. See the appropriate section of `Make.inc` for additional customization options, such as `MARCH` and `JULIA_CPU_TARGET`.

For example, to build for Pentium 4, set `MARCH=pentium4` and install the necessary system libraries for linking. On Ubuntu, these may include lib32gfortran-6-dev, lib32gcc1, and lib32stdc++6, among others.

You can also set `MARCH=native` in `Make.user` for a maximum-performance build customized for the current machine CPU.

Problem	Possible Solution
OpenBLAS build failure	Set one of the following build options in Make.user and build again: - OPENBLAS_TARGET_ARCH=BARCELONA (AMD CPUs) or OPENBLAS_TARGET_ARCH=NEHALEM (Intel CPUs) Set OPENBLAS_DYNAMIC_ARCH = 0 to disable compiling multiple architectures in a single binary. - OPENBLAS_NO_AVX2 = 1 disables AVX2 instructions, allowing OpenBLAS to compile with OPENBLAS_DYNAMIC_ARCH = 1 using old versions of binutils - USE_SYSTEM_BLAS=1 uses the system provided libblas - Set LIBBLAS=-lopenblas and LIBBLASNAME=libopenblas to force the use of the system provided OpenBLAS when multiple BLAS versions are installed. If you get an error that looks like ../kernel/x86_64/dgemm_kernel_4x4_haswell.S:1709: Error: no such instruction: `vpermpd $ 0xb1,%ymm0,%ymm0', then you need to set OPENBLAS_DYNAMIC_ARCH = 0 or OPENBLAS_NO_AVX2 = 1, or you need a newer version of binutils (2.18 or newer). (Issue #7653) If the linker cannot find gfortran and you get an error like julia /usr/bin/x86_64-linux-gnu-ld: cannot find -lgfortran, check the path with gfortran -print-file-name=libgfortran.so and use the output to export something similar to this: export LDFLAGS=-L/usr/lib/gcc/x86_64-linux-gnu/8/. See Issue #6150.
Illegal Instruction error	Check if your CPU supports AVX while your OS does not (e.g. through virtualization, as described in this issue).

Linux Build Troubleshooting

105.3 macOS

You need to have the current Xcode command line utilities installed: run `xcode-select --install` in the terminal. You will need to rerun this terminal command after each macOS update, otherwise you may run into errors involving missing libraries or headers.

The dependent libraries are now built with BinaryBuilder and will be automatically downloaded. This is the preferred way to build Julia source. In case you want to build them all on your own, you will need a 64-bit gfortran to compile Julia dependencies.

```
brew install gcc
```

If you have set `LD_LIBRARY_PATH` or `DYLD_LIBRARY_PATH` in your `.bashrc` or equivalent, Julia may be unable to find various libraries that come bundled with it. These environment variables need to be unset for Julia to work.

105.4 Windows

This file describes how to install, or build, and use Julia on Windows.

For more general information about Julia, please see the main README or the documentation.

General Information for Windows

We highly recommend running Julia using a modern terminal application, in particular Windows Terminal, which can be installed from the Microsoft Store.

Line endings

Julia uses binary-mode files exclusively. Unlike many other Windows programs, if you write `\n` to a file, you get a `\n` in the file, not some other bit pattern. This matches the behavior exhibited by other operating systems. If you have installed Git for Windows, it is suggested, but not required, that you configure your system Git to use the same convention:

```
git config --global core.eol lf
git config --global core.autocrlf input
```

or edit `%USERPROFILE%\.gitconfig` and add/edit the lines:

```
[core]
    eol = lf
    autocrlf = input
```

Binary distribution

For the binary distribution installation notes on Windows please see the instructions at https://julialang.org/downloads/platform/#w

Source distribution

Cygwin-to-MinGW cross-compiling

The recommended way of compiling Julia from source on Windows is by cross compiling from Cygwin, using versions of the MinGW-w64 compilers available through Cygwin's package manager.

1. Download and run Cygwin setup for 32 bit or 64 bit. Note, that you can compile either 32 or 64 bit Julia from either 32 or 64 bit Cygwin. 64 bit Cygwin has a slightly smaller but often more up-to-date selection of packages.

   Advanced: you may skip steps 2-4 by running:

   ```
   setup-x86_64.exe -s <url> -q -P
   ↪  cmake,gcc-g++,git,make,patch,curl,m4,python3,p7zip,mingw64-i686-gcc-g++,mingw64-i686-gcc-fortran,mingw64-x86
   :: replace <url> with a site from https://cygwin.com/mirrors.html
   :: or run setup manually first and select a mirror
   ```

2. Select installation location and download mirror.
3. At the '*Select Packages*' step, select the following:
   1. From the *Devel* category: `cmake`, `gcc-g++`, `git`, `make`, `patch`
   2. From the *Net* category: `curl`
   3. From *Interpreters* (or *Python*) category: `m4`, `python3`
   4. From the *Archive* category: `p7zip`
   5. For 32 bit Julia, and also from the *Devel* category: `mingw64-i686-gcc-g++` and `mingw64-i686-gcc-fortran`
   6. For 64 bit Julia, and also from the *Devel* category: `mingw64-x86_64-gcc-g++` and `mingw64-x86_64-gcc-fortran`
4. Allow Cygwin installation to finish, then start from the installed shortcut a *'Cygwin Terminal'*, or *'Cygwin64 Terminal'*, respectively.

5. Build Julia and its dependencies from source:

   1. Get the Julia sources

      ```
      git clone https://github.com/JuliaLang/julia.git
      cd julia
      ```

      Tip: If you get an `error: cannot fork() for fetch-pack: Resource temporarily unavailable` from git, add `alias git="env PATH=/usr/bin git"` to `~/.bashrc` and restart Cygwin.

   2. Set the XC_HOST variable in `Make.user` to indicate MinGW-w64 cross compilation

      ```
      echo 'XC_HOST = i686-w64-mingw32' > Make.user     # for 32 bit Julia
      # or
      echo 'XC_HOST = x86_64-w64-mingw32' > Make.user   # for 64 bit Julia
      ```

   3. Start the build

      ```
      make -j 4       # Adjust the number of threads (4) to match your build environment.
      make -j 4 debug # This builds julia-debug.exe
      ```

```
> Protip: build both!
> ```sh
> make O=julia-win32 configure
> make O=julia-win64 configure
> echo 'XC_HOST = i686-w64-mingw32' > julia-win32/Make.user
> echo 'XC_HOST = x86_64-w64-mingw32' > julia-win64/Make.user
> echo 'ifeq ($(BUILDROOT),$(JULIAHOME))
>         $(error "in-tree build disabled")
>       endif' >> Make.user
> make -C julia-win32  # build for Windows x86 in julia-win32 folder
> make -C julia-win64  # build for Windows x86-64 in julia-win64 folder
> ```
```

1. Run Julia using the Julia executables directly

   ```
   usr/bin/julia.exe
   usr/bin/julia-debug.exe
   ```

Compiling with MinGW/MSYS2

MSYS2 provides a robust MSYS experience.

Note: MSYS2 requires **64 bit** Windows 7 or newer.

1. Install and configure MSYS2, Software Distribution and Building Platform for Windows.

   1. Download and run the latest installer for the 64-bit distribution. The installer will have a name like `msys2-x86_64-yyyymmdd.exe`.
   2. Open MSYS2. Update package database and base packages: `sh pacman -Syu`
   3. Exit and restart MSYS2, Update the rest of the base packages: `sh pacman -Syu`
   4. Then install tools required to build julia: "'sh

105.5 tools

pacman -S cmake diffutils git m4 make patch tar p7zip curl python

105.6 For 64 bit Julia, install x86_64

pacman -S mingw-w64-x86_64-gcc

105.7 For 32 bit Julia, install i686

pacman -S mingw-w64-i686-gcc “‘

5. Configuration of MSYS2 is complete. Now `exit` the MSYS2 shell.

1. Build Julia and its dependencies with pre-build dependencies.

   1. Open a new **MINGW64/MINGW32 shell**. Currently we can’t use both mingw32 and mingw64, so if you want to build the x86_64 and i686 versions, you’ll need to build them in each environment separately.
   2. and clone the Julia sources `sh git clone https://github.com/JuliaLang/julia.git cd julia`
   3. Start the build `sh make -j$(nproc)`

      Protip: build in dir

```
make O=julia-mingw-w64 configure
echo 'ifeq ($(BUILDROOT),$(JULIAHOME))
        $(error "in-tree build disabled")
      endif' >> Make.user
make -C julia-mingw-w64
```

Cross-compiling from Unix (Linux/Mac/WSL)

You can also use MinGW-w64 cross compilers to build a Windows version of Julia from Linux, Mac, or the Windows Subsystem for Linux (WSL).

First, you will need to ensure your system has the required dependencies. We need wine (>=1.7.5), a system compiler, and some downloaders. Note: a cygwin install might interfere with this method if using WSL.

On Ubuntu (on other Linux systems the dependency names are likely to be similar):

```
apt-get install wine-stable gcc wget p7zip-full winbind mingw-w64 gfortran-mingw-w64
dpkg --add-architecture i386 && apt-get update && apt-get install wine32 # add sudo to each if
↪  needed
# switch all of the following to their "-posix" variants (interactively):
for pkg in i686-w64-mingw32-g++ i686-w64-mingw32-gcc i686-w64-mingw32-gfortran
↪  x86_64-w64-mingw32-g++ x86_64-w64-mingw32-gcc x86_64-w64-mingw32-gfortran; do sudo
↪  update-alternatives --config $pkg; done
```

On Mac: Install XCode, XCode command line tools, X11 (now XQuartz), and MacPorts or Homebrew. Then run `port install wine wget mingw-w64`, or `brew install wine wget mingw-w64`, as appropriate.

Then run the build:

1. `git clone https://github.com/JuliaLang/julia.git julia-win32`
2. `cd julia-win32`
3. `echo override XC_HOST = i686-w64-mingw32 >> Make.user`
4. `make`
5. `make win-extras` (Necessary before running `make binary-dist`)
6. `make binary-dist` then `make exe` to create the Windows installer.
7. move the `julia-*.exe` installer to the target machine

If you are building for 64-bit windows, the steps are essentially the same. Just replace `i686` in `XC_HOST` with `x86_64`. (note: on Mac, wine only runs in 32-bit mode).

Debugging a cross-compiled build under wine

The most effective way to debug a cross-compiled version of Julia on the cross-compilation host is to install a windows version of gdb and run it under wine as usual. The pre-built packages available as part of the MSYS2 project are known to work. Apart from the GDB package you may also need the python and termcap packages. Finally, GDB's prompt may not work when launch from the command line. This can be worked around by prepending `wineconsole` to the regular GDB invocation.

After compiling

Compiling using one of the options above creates a basic Julia build, but not some extra components that are included if you run the full Julia binary installer. If you need these components, the easiest way to get them is to build the installer yourself using `make win-extras` followed by `make binary-dist` and `make exe`. Then running the resulting installer.

Windows Build Debugging

GDB hangs with cygwin mintty

- Run gdb under the windows console (cmd) instead. gdb may not function properly under mintty with non-cygwin applications. You can use `cmd /c start` to start the windows console from mintty if necessary.

GDB not attaching to the right process

- Use the PID from the windows task manager or `WINPID` from the `ps` command instead of the PID from unix style command line tools (e.g. `pgrep`). You may need to add the PID column if it is not shown by default in the windows task manager.

GDB not showing the right backtrace

- When attaching to the julia process, GDB may not be attaching to the right thread. Use `info threads` command to show all the threads and `thread <threadno>` to switch threads.
- Be sure to use a 32 bit version of GDB to debug a 32 bit build of Julia, or a 64 bit version of GDB to debug a 64 bit build of Julia.

Build process is slow/eats memory/hangs my computer

- Disable the Windows Superfetch and Program Compatibility Assistant services, as they are known to have spurious interactions with MinGW/Cygwin.

  As mentioned in the link above: excessive memory use by `svchost` specifically may be investigated in the Task Manager by clicking on the high-memory `svchost.exe` process and selecting `Go to Services`. Disable child services one-by-one until a culprit is found.

- Beware of BLODA. The vmmap tool is indispensable for identifying such software conflicts. Use vmmap to inspect the list of loaded DLLs for bash, mintty, or another persistent process used to drive the build. Essentially *any* DLL outside of the Windows System directory is potential BLODA.

105.8 FreeBSD

Clang is the default compiler on FreeBSD 11.0-RELEASE and above. The remaining build tools are available from the Ports Collection, and can be installed using `pkg install git gcc gmake cmake pkgconf`. To build Julia, simply run `gmake`. (Note that `gmake` must be used rather than `make`, since `make` on FreeBSD corresponds to the incompatible BSD Make rather than GNU Make.)

As mentioned above, it is important to note that the `USE_SYSTEM_*` flags should be used with caution on FreeBSD. This is because many system libraries, and even libraries from the Ports Collection, link to the system's `libgcc_s.so.1`, or to another library which links to the system `libgcc_s`. This library declares its GCC version to be 4.6, which is too old to build Julia, and conflicts with other libraries when linking. Thus it is highly recommended to simply allow Julia to build all of its dependencies. If you do choose to use the `USE_SYSTEM_*` flags, note that `/usr/local` is not on the compiler path by default, so you may need to add `LDFLAGS=-L/usr/local/lib` and `CPPFLAGS=-I/usr/local/include` to your `Make.user`, though doing so may interfere with other dependencies.

Note that the x86 architecture does not support threading due to lack of compiler runtime library support, so you may need to set `JULIA_THREADS=0` in your `Make.user` if you're on a 32-bit system.

105.9 ARM (Linux)

Julia fully supports ARMv8 (AArch64) processors, and supports ARMv7 and ARMv6 (AArch32) with some caveats. This file provides general guidelines for compilation, in addition to instructions for specific devices.

A list of known issues for ARM is available. If you encounter difficulties, please create an issue including the output from `cat /proc/cpuinfo`.

32-bit (ARMv6, ARMv7)

Julia has been successfully compiled on several variants of the following ARMv6 & ARMv7 devices:

- ARMv7 / Cortex A15 Samsung Chromebooks running Ubuntu Linux under Crouton;
- Raspberry Pi.
- Odroid.

Julia requires at least the `armv6` and `vfpv2` instruction sets. It's recommended to use `armv7-a`. `armv5` or soft float are not supported.

Raspberry Pi 1 / Raspberry Pi Zero

If the type of ARM CPU used in the Raspberry Pi is not detected by LLVM, then explicitly set the CPU target by adding the following to `Make.user`:

```
JULIA_CPU_TARGET=arm1176jzf-s
```

To complete the build, you may need to increase the swap file size. To do so, edit `/etc/dphys-swapfile`, changing the line:

```
CONF_SWAPSIZE=100
```

to:

```
CONF_SWAPSIZE=512
```

before restarting the swapfile service:

```
sudo /etc/init.d/dphys-swapfile stop
sudo /etc/init.d/dphys-swapfile start
```

Raspberry Pi 2

The type of ARM CPU used in the Raspberry Pi 2 is not detected by LLVM. Explicitly set the CPU target by adding the following to `Make.user`:

`JULIA_CPU_TARGET=cortex-a7`

Depending on the exact compiler and distribution, there might be a build failure due to unsupported inline assembly. In that case, add `MCPU=armv7-a` to `Make.user`.

AArch64 (ARMv8)

Julia is expected to work and build on ARMv8 cpus. One should follow the general build instructions. Julia expects to have around 8GB of ram or swap enabled to build itself.

Known issues

Starting from Julia v1.10, JITLink is automatically enabled on this architecture for all operating systems when linking to LLVM 15 or later versions. Due to a bug in LLVM memory manager, non-trivial workloads may generate too many memory mappings that on Linux can exceed the limit of memory mappings (`mmap`) set in the file `/proc/sys/vm/max_map_count`, resulting in an error like

```
JIT session error: Cannot allocate memory
```

Should this happen, ask your system administrator to increase the limit of memory mappings for example with the command

```
sysctl -w vm.max_map_count=262144
```

105.10 Binary distributions

These notes are for those wishing to compile a binary distribution of Julia for distribution on various platforms. We love users spreading Julia as far and wide as they can, trying it out on as wide an array of operating systems and hardware configurations as possible. As each platform has specific gotchas and processes that must be followed in order to create a portable, working Julia distribution, we have separated most of the notes by OS.

Note that while the code for Julia is MIT-licensed, with a few exceptions, the distribution created by the techniques described herein will be GPL licensed, as various dependent libraries such as `SuiteSparse` are GPL licensed. We do hope to have a non-GPL distribution of Julia in the future.

Versioning and Git

The Makefile uses both the `VERSION` file and commit hashes and tags from the git repository to generate the `base/version_git.jl` with information we use to fill the splash screen and the `versioninfo()` output. If you for some reason don't want to have the git repository available when building you should pregenerate the `base/version_git.jl` file with:

```
make -C base version_git.jl.phony
```

Julia has lots of build dependencies where we use patched versions that has not yet been included by the popular package managers. These dependencies will usually be automatically downloaded when you build, but if you want to be able to build Julia on a computer without internet access you should create a full-source-dist archive with the special make target

```
make full-source-dist
```

that creates a julia-version-commit.tar.gz archive with all required dependencies.

When compiling a tagged release in the git repository, we don't display the branch/commit hash info in the splash screen. You can use this line to show a release description of up to 45 characters. To set this line you have to create a Make.user file containing:

```
override TAGGED_RELEASE_BANNER = "my-package-repository build"
```

Target Architectures

By default, Julia optimizes its system image to the native architecture of the build machine. This is usually not what you want when building packages, as it will make Julia fail at startup on any machine with incompatible CPUs (in particular older ones with more restricted instruction sets).

We therefore recommend that you pass the `MARCH` variable when calling `make`, setting it to the baseline target you intend to support. This will determine the target CPU for both the Julia executable and libraries, and the system image (the latter can also be set using `JULIA_CPU_TARGET`). Typically useful values for x86 CPUs are `x86-64` and `core2` (for 64-bit builds) and `pentium4` (for 32-bit builds). Unfortunately, CPUs older than Pentium 4 are currently not supported (see this issue).

The full list of CPU targets supported by LLVM can be obtained by running `llc -mattr=help`.

Linux

On Linux, `make binary-dist` creates a tarball that contains a fully functional Julia installation. If you wish to create a distribution package such as a `.deb`, or `.rpm`, some extra effort is needed. See the julia-debian repository for an example of what metadata is needed for creating `.deb` packages for Debian and Ubuntu-based systems. See the Fedora package for RPM-based distributions. Although we have not yet experimented with it, Alien could be used to generate Julia packages for various Linux distributions.

Julia supports overriding standard installation directories via `prefix` and other environment variables you can pass when calling `make` and `make install`. See Make.inc for their list. `DESTDIR` can also be used to force the installation into a temporary directory.

By default, Julia loads `$prefix/etc/julia/startup.jl` as an installation-wide initialization file. This file can be used by distribution managers to set up custom paths or initialization code. For Linux distribution packages, if `$prefix` is set to `/usr`, there is no `/usr/etc` to look into. This requires the path to Julia's private `etc` directory to be changed. This can be done via the `sysconfdir` make variable when building. Simply pass `sysconfdir=/etc` to `make` when building and Julia will first check `/etc/julia/startup.jl` before trying `$prefix/etc/julia/startup.jl`.

OS X

To create a binary distribution on OSX, build Julia first, then cd to `contrib/mac/app`, and run `make` with the same makevars that were used with `make` when building Julia proper. This will then create a `.dmg` file in the `contrib/mac/app` directory holding a completely self-contained Julia.app.

Alternatively, Julia may be built as a framework by invoking `make` with the `darwinframework` target and `DARWIN_FRAMEWORK=1` set. For example, `make DARWIN_FRAMEWORK=1 darwinframework`.

Windows

Instructions for reating a Julia distribution on Windows are described in the build devdocs for Windows.

Notes on BLAS and LAPACK

Julia builds OpenBLAS by default, which includes the BLAS and LAPACK libraries. On 32-bit architectures, Julia builds OpenBLAS to use 32-bit integers, while on 64-bit architectures, Julia builds OpenBLAS to use 64-bit integers (ILP64). It is essential that all Julia functions that call BLAS and LAPACK API routines use integers of the correct width.

Most BLAS and LAPACK distributions provided on linux distributions, and even commercial implementations ship libraries that use 32-bit APIs. In many cases, a 64-bit API is provided as a separate library.

When using vendor provided or OS provided libraries, a `make` option called `USE_BLAS64` is available as part of the Julia build. When doing `make USE_BLAS64=0`, Julia will call BLAS and LAPACK assuming a 32-bit API, where all integers are 32-bit wide, even on a 64-bit architecture.

Other libraries that Julia uses, such as SuiteSparse also use BLAS and LAPACK internally. The APIs need to be consistent across all libraries that depend on BLAS and LAPACK. The Julia build process will build all these libraries correctly, but when overriding defaults and using system provided libraries, this consistency must be ensured.

Also note that Linux distributions sometimes ship several versions of OpenBLAS, some of which enable multi-threading, and others only working in a serial fashion. For example, in Fedora, `libopenblasp.so` is threaded, but `libopenblas.so` is not. We recommend using the former for optimal performance. To choose an OpenBLAS library whose name is different from the default `libopenblas.so`, pass `LIBBLAS=-l$(YOURBLAS)` and `LIBBLASNAME=lib$(YOURBLAS)` to `make`, replacing `$(YOURBLAS)` with the name of your library. You can also

add `.so.0` to the name of the library if you want your package to work without requiring the unversioned `.so` symlink.

Finally, OpenBLAS includes its own optimized version of LAPACK. If you set `USE_SYSTEM_BLAS=1` and `USE_SYSTEM_LAPACK=1`, you should also set `LIBLAPACK=-l$(YOURBLAS)` and `LIBLAPACKNAME=lib$(YOURBLAS)`. Else, the reference LAPACK will be used and performance will typically be much lower.

Starting with Julia 1.7, Julia uses libblastrampoline to pick a different BLAS at runtime.

105.11 Point releasing 101

Creating a point/patch release consists of several distinct steps.

Backporting commits

Some pull requests are labeled "backport pending x.y", e.g. "backport pending 0.6". This designates that the next subsequent release tagged from the release-x.y branch should include the commit(s) in that pull request. Once the pull request is merged into master, each of the commits should be cherry picked to a dedicated branch that will ultimately be merged into release-x.y.

Creating a backports branch

First, create a new branch based on release-x.y. The typical convention for Julia branches is to prefix the branch name with your initials if it's intended to be a personal branch. For the sake of example, we'll say that the author of the branch is Jane Smith.

```
git fetch origin
git checkout release-x.y
git rebase origin/release-x.y
git checkout -b js/backport-x.y
```

This ensures that your local copy of release-x.y is up to date with origin before you create a new branch from it.

Cherry picking commits

Now we do the actual backporting. Find all merged pull requests labeled "backport pending x.y" in the GitHub web UI. For each of these, scroll to the bottom where it says "someperson merged commit `123abc` into `master` XX minutes ago". Note that the commit name is a link; if you click it, you'll be shown the contents of the commit. If this page shows that `123abc` is a merge commit, go back to the PR page—-we don't want merge commits, we want the actual commits. However, if this does not show a merge commit, it means that the PR was squash-merged. In that case, use the git SHA of the commit, listed next to commit on this page.

Once you have the SHA of the commit, cherry-pick it onto the backporting branch:

```
git cherry-pick -x -e <sha>
```

There may be conflicts which need to be resolved manually. Once conflicts are resolved (if applicable), add a reference to the GitHub pull request that introduced the commit in the body of the commit message.

After all of the relevant commits are on the backports branch, push the branch to GitHub.

Checking for performance regressions

Point releases should never introduce performance regressions. Luckily the Julia benchmarking bot, Nanosoldier, can run benchmarks against any branch, not just master. In this case we want to check the benchmark results of js/backport-x.y against release-x.y. To do this, awaken the Nanosoldier from his robotic slumber using a comment on your backporting pull request:

```
@nanosoldier `runbenchmarks(ALL, vs=":release-x.y")`
```

This will run all registered benchmarks on release-x.y and js/backport-x.y and produce a summary of results, marking all improvements and regressions.

If Nanosoldier finds any regressions, try verifying locally and rerun Nanosoldier if necessary. If the regressions are deemed to be real rather than just noise, you'll have to find a commit on master to backport that fixes it if one exists, otherwise you should determine what caused the regression and submit a patch (or get someone who knows the code to submit a patch) to master, then backport the commit once that's merged. (Or submit a patch directly to the backport branch if appropriate.)

Building test binaries

After the backport PR has been merged into the `release-x.y` branch, update your local clone of Julia, then get the SHA of the branch using

```
git rev-parse origin/release-x.y
```

Keep that handy, as it's what you'll enter in the "Revision" field in the buildbot UI.

For now, all you need are binaries for Linux x86-64, since this is what's used for running PackageEvaluator. Go to https://buildog.julialang.org, submit a job for `nuke_linux64`, then queue up a job for `package_linux64`, providing the SHA as the revision. When the packaging job completes, it will upload the binary to the `julialang2` bucket on AWS. Retrieve the URL, as it will be used for PackageEvaluator.

Checking for package breakages

Point releases should never break packages, with the possible exception of packages that are doing some seriously questionable hacks using Base internals that are not intended to be user-facing. (In those cases, maybe have a word with the package author.)

Checking whether changes made in the forthcoming new version will break packages can be accomplished using PackageEvaluator, often called "PkgEval" for short. PkgEval is what populates the status badges on GitHub repos and on pkg.julialang.org. It typically runs on one of the non-benchmarking nodes of Nanosoldier and uses Vagrant to perform its duties in separate, parallel VirtualBox virtual machines.

Setting up PackageEvaluator

Clone PackageEvaluator and create a branch called `backport-x.y.z`, and check it out. Note that the required changes are a little hacky and confusing, and hopefully that will be addressed in a future version of PackageEvaluator. The changes to make will be modeled off of this commit.

The setup script takes its first argument as the version of Julia to run and the second as the range of package names (AK for packages named A-K, LZ for L-Z). The basic idea is that we're going to tweak that a bit to run only two versions of Julia, the current x.y release and our backport version, each with three ranges of packages.

In the linked diff, we're saying that if the second argument is LZ, use the binaries built from our backport branch, otherwise (AK) use the release binaries. Then we're using the first argument to run a section of the package list: A-F for input 0.4, G-N for 0.5, and O-Z for 0.6.

Running PackageEvaluator

To run PkgEval, find a hefty enough machine (such as Nanosoldier node 1), then run

```
git clone https://github.com/JuliaCI/PackageEvaluator.jl.git
cd PackageEvaluator.jl/scripts
git checkout backport-x.y.z
./runvagrant.sh
```

This produces some folders in the scripts/ directory. The folder names and their contents are decoded below:

Folder name	Julia version	Package range
0.4AK	Release	A-F
0.4LZ	Backport	A-F
0.5AK	Release	G-N
0.5LZ	Backport	G-N
0.6AK	Release	O-Z
0.6LZ	Backport	O-Z

Investigating results

Once that's done, you can use `./summary.sh` from that same directory to produce a summary report of the findings. We'll do so for each of the folders to aggregate overall results by version.

```
./summary.sh 0.4AK/*.json > summary_release.txt
./summary.sh 0.5AK/*.json >> summary_release.txt
./summary.sh 0.6AK/*.json >> summary_release.txt
./summary.sh 0.4LZ/*.json > summary_backport.txt
./summary.sh 0.5LZ/*.json >> summary_backport.txt
./summary.sh 0.6LZ/*.json >> summary_backport.txt
```

Now we have two files, `summary_release.txt` and `summary_backport.txt`, containing the PackageEvaluator test results (pass/fail) for each package for the two versions.

To make these easier to ingest into a Julia, we'll convert them into CSV files then use the DataFrames package to process the results. To convert to CSV, copy each .txt file to a corresponding .csv file, then enter Vim and execute `ggVGI"<esc>` then `:%s/\.json /",/g`. (You don't have to use Vim; this just is one way to do it.) Now process the results with Julia code similar to the following.

```
using DataFrames

release = readtable("summary_release.csv", header=false, names=[:package, :release])
backport = readtable("summary_backport.csv", header=false, names=[:package, :backport])

results = join(release, backport, on=:package, kind=:outer)

for result in eachrow(results)
```

```
    a = result[:release]
    b = result[:backport]
    if (isna(a) && !isna(b)) || (isna(b) && !isna(a))
        color = :yellow
    elseif a != b && occursin("pass", b)
        color = :green
    elseif a != b
        color = :red
    else
        continue
    end
    printstyled(result[:package], ": Release ", a, " -> Backport ", b, "\n", color=color)
end
```

This will write color-coded lines to `stdout`. All lines in red must be investigated as they signify potential breakages caused by the backport version. Lines in yellow should be looked into since it means a package ran on one version but not on the other for some reason. If you find that your backported branch is causing breakages, use `git bisect` to identify the problematic commits, `git revert` those commits, and repeat the process.

Merging backports into the release branch

After you have ensured that

- the backported commits pass all of Julia's unit tests,
- there are no performance regressions introduced by the backported commits as compared to the release branch, and
- the backported commits do not break any registered packages,

then the backport branch is ready to be merged into release-x.y. Once it's merged, go through and remove the "backport pending x.y" label from all pull requests containing the commits that have been backported. Do not remove the label from PRs that have not been backported.

The release-x.y branch should now contain all of the new commits. The last thing we want to do to the branch is to adjust the version number. To do this, submit a PR against release-x.y that edits the VERSION file to remove `-pre` from the version number. Once that's merged, we're ready to tag.

Tagging the release

It's time! Check out the release-x.y branch and make sure that your local copy of the branch is up to date with the remote branch. At the command line, run

```
git tag v$(cat VERSION)
git push --tags
```

This creates the tag locally and pushes it to GitHub.

After tagging the release, submit another PR to release-x.y to bump the patch number and add `-pre` back to the end. This denotes that the branch state reflects a prerelease version of the next point release in the x.y series.

Follow the remaining directions in the Makefile.

Signing binaries

Some of these steps will require secure passwords. To obtain the appropriate passwords, contact Elliot Saba (staticfloat) or Alex Arslan (ararslan). Note that code signing for each platform must be performed on that platform (e.g. Windows signing must be done on Windows, etc.).

Linux

Code signing must be done manually on Linux, but it's quite simple. First obtain the file `julia.key` from the CodeSigning folder in the `juliasecure` AWS bucket. Add this to your GnuPG keyring using

```
gpg --import julia.key
```

This will require entering a password that you must obtain from Elliot or Alex. Next, set the trust level for the key to maximum. Start by entering a gpg session:

```
gpg --edit-key julia
```

At the prompt, type `trust`, then when asked for a trust level, provide the maximum available (likely 5). Exit GnuPG.

Now, for each of the Linux tarballs that were built on the buildbots, enter

```
gpg -u julia --armor --detach-sig julia-x.y.z-linux-<arch>.tar.gz
```

This will produce a corresponding .asc file for each tarball. And that's it!

macOS

Code signing should happen automatically on the macOS buildbots. However, it's important to verify that it was successful. On a system or virtual machine running macOS, download the .dmg file that was built on the buildbots. For the sake of example, say that the .dmg file is called `julia-x.y.z-osx.dmg`. Run

```
mkdir ./jlmnt
hdiutil mount -readonly -mountpoint ./jlmnt julia-x.y.z-osx.dmg
codesign -v jlmnt/Julia-x.y.app
```

Be sure to note the name of the mounted disk listed when mounting! For the sake of example, we'll assume this is `disk3`. If the code signing verification exited successfully, there will be no output from the `codesign` step. If it was indeed successful, you can detach the .dmg now:

```
hdiutil eject /dev/disk3
rm -rf ./jlmnt
```

If you get a message like

> Julia-x.y.app: code object is not signed at all

then you'll need to sign manually.

To sign manually, first retrieve the OS X certificates from the CodeSigning folder in the `juliasecure` bucket on AWS. Add the .p12 file to your keychain using Keychain.app. Ask Elliot Saba (staticfloat) or Alex Arslan (ararslan) for the password for the key. Now run

```
hdiutil convert julia-x.y.z-osx.dmg -format UDRW -o julia-x.y.z-osx_writable.dmg
mkdir ./jlmnt
hdiutil mount -mountpoint julia-x.y.z-osx_writable.dmg
codesign -s "AFB379C0B4CBD9DB9A762797FC2AB5460A2B0DBE" --deep jlmnt/Julia-x.y.app
```

This may fail with a message like

> Julia-x.y.app: resource fork, Finder information, or similar detritus not allowed

If that's the case, you'll need to remove extraneous attributes:

```
xattr -cr jlmnt/Julia-x.y.app
```

Then retry code signing. If that produces no errors, retry verification. If all is now well, unmount the writable .dmg and convert it back to read-only:

```
hdiutil eject /dev/disk3
rm -rf ./jlmnt
hdiutil convert julia-x.y.z-osx_writable.dmg -format UDZO -o julia-x.y.z-osx_fixed.dmg
```

Verify that the resulting .dmg is in fact fixed by double clicking it. If everything looks good, eject it then drop the `_fixed` suffix from the name. And that's it!

Windows

Signing must be performed manually on Windows. First obtain the Windows 10 SDK, which contains the necessary signing utilities, from the Microsoft website. We need the `SignTool` utility which should have been installed somewhere like `C:\Program Files (x86)\Windows Kits\10\App Certification Kit`. Grab the Windows certificate files from CodeSigning on `juliasecure` and put them in the same directory as the executables. Open a Windows CMD window, cd to where all the files are, and run

```
set PATH=%PATH%;C:\Program Files (x86)\Windows Kits\10\App Certification Kit;
signtool sign /f julia-windows-code-sign_2017.p12 /p "PASSWORD" ^
   /t http://timestamp.verisign.com/scripts/timstamp.dll ^
   /v julia-x.y.z-win32.exe
```

Note that ^ is a line continuation character in Windows CMD and `PASSWORD` is a placeholder for the password for this certificate. As usual, contact Elliot or Alex for passwords. If there are no errors, we're all good!

Uploading binaries

Now that everything is signed, we need to upload the binaries to AWS. You can use a program like Cyberduck or the aws command line utility. The binaries should go in the `julialang2` bucket in the appropriate folders. For example, Linux x86-64 goes in `julialang2/bin/linux/x.y`. Be sure to delete the current `julia-x.y-latest-linux-<arch>.tar.gz` file and replace it with a duplicate of `julia-x.y.z-linux-<arch>.tar.gz`.

We also need to upload the checksums for everything we've built, including the source tarballs and all release binaries. This is simple:

```
shasum -a 256 julia-x.y.z* | grep -v -e sha256 -e md5 -e asc > julia-x.y.z.sha256
md5sum julia-x.y.z* | grep -v -e sha256 -e md5 -e asc > julia-x.y.z.md5
```

Note that if you're running those commands on macOS, you'll get very slightly different output, which can be reformatted by looking at an existing file. Mac users will also need to use `md5 -r` instead of `md5sum`. Upload the .md5 and .sha256 files to `julialang2/bin/checksums` on AWS.

Ensure that the permissions on AWS for all uploaded files are set to "Everyone: READ."

For each file we've uploaded, we need to purge the Fastly cache so that the links on the website point to the updated files. As an example:

```
curl -X PURGE https://julialang-s3.julialang.org/bin/checksums/julia-x.y.z.sha256
```

Sometimes this isn't necessary but it's good to do anyway.