Julia 中文文档

Julia 中文社区

May 17, 2021

Contents

Part I

主页

Part II

Julia 1.5 中文文档

欢迎来到 Julia 1.5 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。

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。

Chapter 1

鸣谢

Chapter 2

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。

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。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。

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 III

手册

Chapter 3

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。

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 1.3.1 (2019-12-30)
 _/ |\__'_|_|_|\__'_|  |  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")`。

如果想非交互式地执行文件中的代码，可以把文件名作为 `julia` 命令的第一个参数：

```
$ julia script.jl arg1 arg2...
```

如这个例子所示，`julia` 后跟着的命令行参数会被作为程序 `script.jl` 的命令行参数。这些参数使用全局常量 `ARGS` 来传递，脚本自身的名字会以全局变量 `PROGRAM_FILE` 传入。注意当脚本以命令行里的 `-e` 选项输入时，`ARGS` 也会被设定（详见此页末尾列表）但是 `PROGRAM_FILE` 会是空的。例如，要把一个脚本的输入参数显示出来，你可以：

```
$ julia -e 'println(PROGRAM_FILE); for x in ARGS; println(x); end' foo bar
```

```
foo
bar
```

或者你可以把代码写到一个脚本文件中再执行它：

```
$ echo 'println(PROGRAM_FILE); for x in ARGS; println(x); end' > script.jl
$ julia script.jl foo bar
script.jl
foo
bar
```

可以使用 `--` 分隔符来将传给脚本文件的参数和 Julia 本身的命令行参数区分开：

```
$ julia --color=yes -O -- foo.jl arg1 arg2..
```

有关编写 Julia 脚本的更多信息，请参阅 脚本。

使用选项 `-p` 或者 `--machine-file` 可以在并行模式下启动 Julia。`-p n` 会启动额外的 `n` 个 worker，使用 `--machine-file file` 会为 `file` 文件中的每一行启动一个 worker。定义在 `file` 中的机器必须能够通过一个不需要密码的 `ssh` 登陆访问到，且 Julia 的安装位置需要和当前主机相同。定义机器的格式为 `[count*][user@]host[:port] [bind_addr[:port]]`。`user` 默认值是当前用户；`port` 默认值是标准 ssh 端口；`count` 是在这个节点上的 worker 的数量，默认是 `1`；可选的 `bind-to bind_addr[:port]` 指定了其它 worker 访问当前 worker 应当使用的 IP 地址与端口。

要让 Julia 每次启动都自动执行一些代码，你可以把它们放在 `~/.julia/config/startup.jl` 中：

```
$ echo 'println("Greetings! 你好! 🇫🇫🇫🇫🇫?")' > ~/.julia/config/startup.jl
$ julia
Greetings! 你好! 🇫🇫🇫🇫🇫?

...
```

在你第一次运行 Julia 后，你一你应该多了一个 `~/.julia` 文件夹。你还可以新建 `~/.julia/config` 文件夹和 `~/.julia/config/startup.jl` 文件来配置 Julia。

和 `perl` 和 `ruby` 程序类似，还有很多种运行 Julia 代码的方式，运行代码时也有很多选项：

```
julia [switches] -- [programfile] [args...]
```

> **Julia 1.1**
>
> 在 Julia 1.0 中，默认的 `--project=@.` 选项不会在 Git 仓库的根目录中寻找 `Project.toml` 文件。从 Julia 1.1 开始，此选项会在其中寻找该文件。

3.1 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。

选项	描述
`-v`, `--version`	显示版本信息
`-h`, `--help`	显示命令行参数
`--project[={<dir>|@.}]`	将 <dir> 设置为主项目/环境。默认的 @. 选项将搜索父目录，直至找到 Project.toml 或 JuliaProject.toml 文件。
`-J`, `--sysimage <file>`	用指定的系统镜像文件（system image file）启动
`-H`, `--home <dir>`	设置 `julia` 可执行文件的路径
`--startup-file={yes|no}`	是否载入 `~/.julia/config/startup.jl`
`--handle-signals={yes|no}`	开启或关闭 Julia 默认的 signal handlers
`--sysimage-native-code={yes|no}`	在可能的情况下，使用系统镜像里的原生代码
`--compiled-modules={yes|no}`	开启或关闭 module 的增量预编译功能
`-e`, `--eval <expr>`	执行 <expr>
`-E`, `--print <expr>`	执行 <expr> 并显示结果
`-L`, `--load <file>`	立即在所有进程中载入 <file>
`-t`, `--threads {N|auto}`	开启 N 个线程：`auto` 将 N 设置为当前 CPU 线程数，但这个行为可能在以后版本有所变动。
`-p`, `--procs {N|auto}`	这里的整数 N 表示启动 N 个额外的工作进程；`auto` 表示启动与 CPU 线程数目（logical cores）一样多的进程
`--machine-file <file>`	在 <file> 中列出的主机上运行进程
`-i`	交互式模式；REPL 运行且 `isinteractive()` 为 true
`-q`, `--quiet`	安静的启动；REPL 启动时无横幅，不显示警告
`--banner={yes|no|auto}`	开启或关闭 REPL 横幅
`--color={yes|no|auto}`	开启或关闭文字颜色
`--history-file={yes|no}`	载入或导出历史记录
`--depwarn={yes|no|error}`	开启或关闭语法弃用警告，`error` 表示将弃用警告转换为错误。
`--warn-overwrite={yes|no}`	开启或关闭“method overwrite”警告
`-C`, `--cpu-target <target>`	设置 <target> 来限制使用 CPU 的某些特性；设置为 `help` 可以查看可用的选项
`-O`, `--optimize={0,1,2,3}`	设置编译器优化级别 (若未配置此选项，则默认等级为 2；若配置了此选项却没指定具体级别，则默认级别为 3)。
`-g`, `-g <level>`	开启或设置 debug 信息的生成等级。若未配置此选项，则默认 debug 信息的级别为 1；若配置了此选项却没指定具体级别，则默认级别为 2。
`--inline={yes|no}`	控制是否允许函数内联，此选项会覆盖源文件中的 `@inline` 声明
`--check-bounds={yes|no|auto}`	设置边界检查状态：始终检查或永不检查。永不检查时会忽略源文件中的相应声明
`--math-mode={ieee,fast}`	开启或关闭非安全的浮点数代数计算优化，此选项会覆盖源文件中的 `@fastmath` 声明
`--code-coverage={none|user|all}`	对源文件中每行代码执行的次数计数
`--code-coverage`	等价于 `--code-coverage=user`
`--track-allocation={none|user|all}`	对源文件中每行代码的内存分配计数，单位 byte
`--track-allocation`	等价于 `--track-allocation=user`

Chapter 4

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> 🇫🇫🇫🇫🇫 = "Hello"
"Hello"
```

在 Julia REPL 和一些其它的 Julia 编辑环境中，很多 Unicode 数学符号可以使用反斜杠加 LaTeX 符号名再按 *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（比如别人的代码里）看到了一个不知道怎么输入的符号，你可以在 REPL 中输入 `?`，然后粘贴那个符号，帮助文档会告诉你输入方法。

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 variable MathConstants.pi from module Main

julia> sqrt(100)
10.0

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

4.1 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。在一些语境中，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 Sm 类别中的 Unicode 中缀运算符，像 `⊕`，则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 `+̂ₐ″` 被解析成一个与 `+` 具有相同优先级的中缀运算符。

唯一明确禁止的变量名称是内置关键字的名称：

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

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

一些 Unicode 字符在标识符中被认为是等效的。不同的输入 Unicode 组合字符的方法（例如：重音）被视为等价的（Julia 标识符是 NFC 标准化的）。Unicode 字符 ε (U+025B: Latin small letter open e) 和 µ (U+00B5: micro sign) 被视为等同于相应的希腊字母，因为前者很容易通过一些方法输入。

4.2 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。
- 用下划线（_）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，不仅仅会返回一些值还会更改输入参数的内容。

关于命名规范的更多信息，可查看代码风格指南。

Chapter 5

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`	half	16
`Float32`	single	32
`Float64`	double	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。

5.1 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> 0x1
0x01

julia> typeof(ans)
UInt8

julia> 0x123
0x0123

julia> typeof(ans)
UInt16

julia> 0x1234567
0x01234567

julia> typeof(ans)
UInt32

julia> 0x123456789abcdef
0x0123456789abcdef

julia> typeof(ans)
UInt64

julia> 0x11112222333344445555666677778888
0x11112222333344445555666677778888

julia> typeof(ans)
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。

还记得这个 `ans` 变量吗？它存着交互式会话中上一个表达式的运算结果，但以其他方式运行的 Julia 代码中没有这个变量。

二进制和八进制字面量也是支持的：

```
julia> 0b10
0x02

julia> typeof(ans)
UInt8

julia> 0o010
0x08

julia> typeof(ans)
UInt8

julia> 0x00000000000000001111222233334444
0x00000000000000001111222233334444

julia> typeof(ans)
UInt128
```

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。当开头都是 0 时，位数取决于一个字面量需要的最少位数，这里的字面量指的是一个有着同样长度但开头都为 1 的数。这样用户就可以控制位数了。那些无法使用 UInt128 类型存储下的值无法写成这样的字面量。

二进制、八进制和十六进制的字面量可以在前面紧接着加一个负号 -，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、字符串和插值，但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

Julia 中，超出一个类型可表示的最大值会导致循环行为：

```
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。

5.2 浮点数

浮点数字面量也使用标准格式表示，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` 值。使用 `f` 替代 `e` 可以得到 `Float32` 的字面量：

```
julia> 0.5f0
0.5f0

julia> typeof(ans)
Float32

julia> 2.5f-4
0.00025f0
```

数值容易就能转换成 `Float32`：

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

julia> typeof(ans)
Float32
```

也存在十六进制的浮点数字面量，但只适用于 Float64 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> 0x.4p-1
0.125

julia> typeof(ans)
Float64
```

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

```
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

浮点数有两个零，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
```

`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值直接的绝对差。也就是说，eps(x) 产生一个和 x 类型相同的值使得 x + eps(x) 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。

基础知识与参考文献

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。

5.3 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。

构造函数可以从基本的数值类型或 字符串字面量创建这些类型。@big_str 和 parse 可以用来从 AbstractString 构造它们。一旦被创建，它们就可以像所有其它数值类型一样参与算术（这也多亏了 Julia 的类型提升和转换机制）。

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

julia> big"123456789012345678901234567890" + 1
123456789012345678901234567891

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

julia> big"1.23456789012345678901"
1.2345678901234567890100000000000000000000000000000000000000000000000000000000000000000000000000000004

julia> parse(BigFloat, "1.23456789012345678901")
1.2345678901234567890100000000000000000000000000000000000000000000000000000000000000000000000000000004

julia> BigFloat(2.0^66) / 3
2.459565876494606882133333333333333333333333333333333333333333333333333333333333333333333333344e+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
1.100000000000000000000000000000000000000000000000000000000000000000000000000000000000000003

julia> setrounding(BigFloat, RoundDown) do
           BigFloat(1) + parse(BigFloat, "0.1")
       end
1.099999999999999999999999999999999999999999999999999999999999999999999999999999999999999986

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

5.4 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，都会因为左手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`。
- 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，都使用这样的解释方式来解决歧义：

- `0x` 开头的表达式总是十六进制字面量。

- 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。

5.5 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 6

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。

6.1 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)`

以及对 `Bool` 类型的否定：

表达式	名称	描述
`!x`	否定	将 `true` 和 `false` 互换

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。

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)。

6.2 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`	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> ~UInt32(123)
0xffffff84

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

6.3 复合赋值操作符

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
```

二元运算和位运算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
```

6.4 向量化 dot 运算符

Julia 中，**每个**二元运算符都有一个 dot 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 Array{Int64,1}:
  1
  8
 27
```

具体来说，`a .^ b` 被解析为 dot 调用 `(^).(a,b)`，这会执行 broadcast 操作：该操作能结合数组和标量、相同大小的数组（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）。更进一步，就像所有向量化的 dot 调用一样，这些 dot 运算符是**融合**的（fused）。例如，在计算表达式 `2 .* A.^2 .+ sin.(A)` 时，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)`。上述表达式也可以用`@.` 宏简写为 `@. 2A^2 + sin(A)`。特别的，类似 `f.(g.(x))` 的嵌套 dot 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 dot 运算符，我们还有 dot 复合赋值运算符，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。

6.5 数值比较

标准的比较操作对所有原始数值类型有定义：

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

下面是一些简单的例子：

```
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。

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

6.6 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：

要看**全部** Julia 运算符的优先级关系，可以看这个文件的最上面部分：`src/julia-parser.scm`

数字字面量系数，例如 `2x` 中的 `2`，它的优先级比二元运算符高，因此会当作乘法，并且它的优先级也比 `^` 高。

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`。

[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] The operators `+`, `++` and `*` are non-associative. `a + b + c` is parsed as `+(a, b, c)` not `+(+(a, b), c)`. However, the fallback methods for `+(a, b, c, d...)` and `*(a, b, c, d...)` both default to left-associative evaluation.

6.7 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。

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

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)	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)	以 2pi 为基取模；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

舍入函数

除法函数

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

所有标准的三角函数和双曲函数也都已经定义了：

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
```

函数	描述
sqrt(x), √x	x 的平方根
cbrt(x), ∛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）

```
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 7

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。

7.1 复数

在 Julia 中，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系数与标识符并置，这种绑定就足够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
0.20689655172413796 + 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 - 3.0518977991518im

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 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
```

7.2 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
```

对任意整数值 a 和 b（除了 a == 0 且 b == 0 时），从分数到浮点数的转换遵从以下的一致性：

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

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

Julia 接受构建无穷分数值：

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

julia> -3//0
-1//0

julia> typeof(ans)
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 8

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（和字符串文字）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。

8.1 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：

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

```
julia> typeof(ans)
Char
```

你可以轻松地将 Char 转换为其对应的整数值，即 Unicode 代码：

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

julia> typeof(ans)
Int64
```

在 32 位架构中，typeof(ans) 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)
```

8.2 字符串基础

字符串字面量由双引号或三重双引号分隔：

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

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

If you want to extract a character from a string, you index into it:

```
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。第一个元素的索引由 `firstindex(str)` 返回，最后一个由 `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` 总是返回 `1`。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（code units）组成。

你可以用 `end` 进行算术以及其它操作，就像普通值一样：

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

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

下标小于开头 `begin` (`1`) 或者大于结尾 `end` 都会导致错误：

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

julia> str[end+1]
ERROR: BoundsError: attempt to access 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"
```

注意到 `str[k]` 和 `str[k:k]` 输出的结果不一样：

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

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

前者是 `Char` 类型的单个字符值，后者是碰巧只有单个字符的字符串值。在 Julia 里面两者大不相同。

范围索引复制了原字符串的选定部分。此外，也可以用 `SubString` 类型创建字符串的 `view`，例如：

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

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

julia> typeof(substr)
SubString{String}
```

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

8.3 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("∀ x ∃ y", 2)
[...]

julia> s[3]
ERROR: StringIndexError("∀ x ∃ y", 3)
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("∀ x ∃ y", 9)
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 ∃ 的多字节表示。这里正确的方法是使用 `prevind(s, lastindex(s), 2)`，或者如果你使用那个值来索引在 s 中可以写 `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("∀ x ∃ y", 2)
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。因此总有 `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 Array{Int64,1}:
  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。

8.4 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 = a*b
"∀"

julia> collect.([a, b, c])
3-element Array{Array{Char,1},1}:
 ['\xe2\x88']
 ['\x80']
 ['∀']

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

8.5 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, $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 Array{Int64,1}:
 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.
```

8.6 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."
```

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"。

8.7 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(isequal('o'), "xylophone")
4

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

julia> findfirst(isequal('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(isequal('o'), "xylophone", 1)
4

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

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

julia> findnext(isequal('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。

8.8 非标准字符串字面量

有时当你想构造字符串或者使用字符串语义，标准的字符串构造却不能很好的满足需求。Julia 为这种情形提供了非标准字符串字面量。非标准字符串字面量看似常规双引号字符串字面量，但却直接加上了标识符前缀因而并不那么像普通的字符串字面量。下面将提到，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例子。其它例子见元编程章。

8.9 正则表达式

Julia 具有与 Perl 兼容的正则表达式 (regexes)，就像 PCRE 包所提供的那样，详细信息参见 PCRE 的语法说明。正则表达式以两种方式和字符串相关：一个显然的关联是，正则表达式被用于找到字符串中的正则模式；另一个关联是，正则表达式自身就是作为字符串输入，它们被解析到可用来高效搜索字符串中模式的状态机中。在 Julia 中正则表达式的输入使用了前缀各类以 `r` 开头的标识符的非标准字符串字面量。最基本的不打开任何选项的正则表达式只用到了 `r"..."`：

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

julia> typeof(ans)
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 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公共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 Array{Union{Nothing, SubString{String}},1}:
 "a"
 "c"
 "d"

julia> m.offset
1

julia> m.offsets
3-element Array{Int64,1}:
 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 Array{Union{Nothing, SubString{String}},1}:
 "a"
 nothing
 "d"

julia> m.offset
1

julia> m.offsets
3-element Array{Int64,1}:
 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：

```
julia> first, second, third = m.captures; 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，相应映射中代码点小于 255 的部分取自当前区域设置，更大代码点的部分取自 Unicode 规则。然而，跨越 Unicode 规则（ords 255/256）和 非 Unicode 规则边界的匹配将失败。

m 将字符串视为多行。也即更改 "^" 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。也即更改 "." 以匹配任何字符，即使是通常不能匹配的换行符。

像这样一起使用，r""ms，它们让 "." 匹配任何字符，同时也支持分别在字符串中换行符的后面和前面用 "^" 和 "$" 进行匹配。

x 令正则表达式解析器忽略多数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（略为）更易读的部分。和普通代码中一样，`#` 字符也被当作引入注释的元字符。

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("[\"( ]$name[\") ]")  # 插入 name 的值
r"[\"( ]Jon[\") ]"

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

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

8.10 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
```

如前所述，CodeUnits{UInt8,String} 类型的行为类似于只读 UInt8 数组。如果需要标准数组，你可以 '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: setindex! not defined for Base.CodeUnits{UInt8,String}
[...]

julia> Vector{UInt8}(x)
3-element Array{UInt8,1}:
 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。

8.11 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。

8.12 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 9

函数

在 Julia 里，函数是一个将参数值元组映射到返回值的对象。Julia 的函数不是纯粹的数学函数，在某种意义上，函数可以改变并受程序的全局状态的影响。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
```

9.1 参数传递行为

Julia 函数参数遵循有时称为“pass-by-sharing”的约定，这意味着变量在被传递给函数时其值并不会被复制。函数参数本身充当新的变量绑定（指向变量值的新地址），它们所指向的值与所传递变量的值完全相同。调用者可以看到对函数内可变值（如数组）的修改。这与 Scheme，大多数 Lisps，Python，Ruby 和 Perl 以及其他动态语言中的行为相同。

9.2 `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 nothing

For functions that do not need to return a value (functions used only for some side effects), the Julia convention is to return the value 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。

9.3 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。

9.4 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：

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

9.5 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 Array{Float64,1}:
 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 Array{Int64,1}:
  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)
```

The call to `time` is delayed by wrapping it in a 0-argument anonymous function that is called only when the requested key is absent from `dict`.

9.6 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）。

9.7 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
```

Named tuples are very similar to tuples, except that fields can additionally be accessed by name using dot syntax (`x.a`) in addition to the regular indexing syntax (`x[1]`).

9.8 多返回值

Julia 中，一个函数可以返回一个元组来实现返回多个值。不过，元组的创建和消除都不一定要用括号，这时候给人的感觉就是返回了多个值而非一个元组。比如下面这个例子，函数返回了两个值：

```
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)
```

这种值对的典型用法是把每个值抽取为一个变量。Julia 支持简洁的元组“解构”：

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

julia> x
5

julia> y
6
```

你也可以显式地使用 `return` 关键字来返回多个值：

```
function foo(a,b)
    return a+b, a*b
end
```

这与之前的定义的 `foo` 函数具有完全相同的效果。

9.9 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
```

Notice the extra set of parentheses in the definition of gap. Without those, gap would be a two-argument function, and this example would not work.

9.10 变参函数

It is often convenient to be able to write functions taking an arbitrary number of arguments. Such functions are traditionally known as "varargs" functions, which is short for "variable number of arguments". You can define a varargs function by following the last positional argument with an ellipsis:

```
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 Array{Int64,1}:
 3
 4

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

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

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

另外，参数可拆解的函数也不一定就是变参函数——尽管一般都是：

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

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

julia> baz(args...)
3

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

julia> baz(args...)
ERROR: MethodError: no method matching baz(::Int64, ::Int64, ::Int64)
Closest candidates are:
baz(::Any, ::Any) at none:1
```

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。

9.11 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：

```
function Date(y::Int64, m::Int64=1, d::Int64=1)
    err = validargs(Date, y, m, d)
    err === nothing || throw(err)
    return Date(UTD(totaldays(y, m, d)))
end
```

Observe, that this definition calls another method of the Date function that takes one argument of type UTInstant{Day}.

With this definition, the function can be called with either one, two or three arguments, and 1 is automatically passed when only one or two of the arguments are specified:

```
julia> using Dates

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。

9.12 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
```

Keyword arguments can also be used in varargs functions:

```
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` 内部，`kwargs` 会是一个具名元组。具名元组（以及键类型为 `Symbol` 的字典）可作为关键字参数传递，这通过在调用中使用分号，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。

When a bare identifier or dot expression occurs after a semicolon, the keyword argument name is implied by the identifier or field name. For example `plot(x, y; width)` is equivalent to `plot(x, y; width=width)` and `plot(x, y; options.width)` is equivalent to `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。

9.13 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`。

9.14 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 `map`。类似地，`do a, b` 会创建一个双参数匿名函数，而一个简单的 `do` 会声明一个满足形式 `() -> ...` 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。

A do block, like any other inner function, can "capture" variables from its enclosing scope. For example, the variable data in the above example of open...do is captured from the outer scope. Captured variables can create performance challenges as discussed in performance tips.

9.15 Function composition and piping

Functions in Julia can be combined by composing or piping (chaining) them together.

Function composition is when you combine functions together and apply the resulting composition to arguments. You use the function composition operator (∘) to compose the functions, so (f ∘ g)(args...) is the same as f(g(args...)).

You can type the composition operator at the REPL and suitably-configured editors using \circ<tab>.

For example, the sqrt and + functions can be composed like this:

```
julia> (sqrt ∘ +)(3, 6)
3.0
```

这个语句先把数字相加，再对结果求平方根。

The next example composes three functions and maps the result over an array of strings:

```
julia> map(first ∘ reverse ∘ uppercase, split("you can compose functions like this"))
6-element Array{Char,1}:
 '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)
```

Function chaining (sometimes called "piping" or "using a pipe" to send data to a subsequent function) is when you apply a function to the previous function's output:

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

Here, the total produced by sum is passed to the sqrt function. The equivalent composition would be:

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

The pipe operator can also be used with broadcasting, as .|>, to provide a useful combination of the chaining/piping and dot vectorization syntax (described next).

```
julia> ["a", "list", "of", "strings"] .|> [uppercase, reverse, titlecase, length]
4-element Array{Any,1}:
  "A"
  "tsil"
  "Of"
 7
```

9.16 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 Array{Float64,1}:
 1.0
 2.0
 3.0

julia> sin.(A)
3-element Array{Float64,1}:
 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 Array{Float64,1}:
 13.42477796076938
 17.42477796076938
 21.42477796076938

julia> f.(A, B)
3-element Array{Float64,1}:
 19.0
 26.0
 33.0
```

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 Array{Float64,1}:
  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 Array{Real,1}:
    1
    0.5
    6
   -4
 true
```

9.17 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 10

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：`&&`、`||` 和链式比较。
- 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。

10.1 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
```

10.2 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"
```

10.3 短路求值

短路求值非常类似条件求值。这种行为在多数有 && 和 || 布尔运算符地命令式编程语言里都可以找到：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。更明确的说，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
```

10.4 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 <= 5
           println(i)
           global i += 1
       end
1
2
3
4
5
```

`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 = 1:5
           println(i)
       end
1
2
3
4
5
```

这里的 `1:5` 是一个范围对象，代表数字 1, 2, 3, 4, 5 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。如果变量 `i` 没有在另一个作用域里引入，在 `for` 循环内，它就只在 `for` 循环内部可见，在外部和后面均不可见。你需要一个新的交互式会话实例或者一个新的变量名来测试这个特性：

```
julia> for j = 1:5
           println(j)
       end
1
2
```

```
3
4
5

julia> j
ERROR: UndefVarError: j not defined
```

参见变量作用域中对变量作用域的详细解释以及它在 Julia 中是如何工作的。

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 >= 5
               break
           end
           global i += 1
       end
1
2
3
4
5

julia> for j = 1:1000
           println(j)
           if j >= 5
               break
           end
       end
1
2
3
```

```
4
5
```

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`。

10.5 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。

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

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 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：

```
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 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` 组件的强大之处在于能够将高度嵌套的计算立刻解耦成更高层次地调用函数。有时没有错误产生，但需要能够解耦堆栈，并传值到上层。Julia 提供了 `rethrow`、`backtrace`、`catch_backtrace` 和 `Base.catch_stack` 函数进行更高级的错误处理。

`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
    # operate on file 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。

10.6 Task（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 11

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见的。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**，这是有资格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可能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	Allowed within
module, baremodule	全局	全局
struct	local (soft)	全局
for, while, try	local (soft)	全局或局部
macro	local (hard)	全局
let, functions, comprehensions, generators	local (hard)	全局或局部

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会从其调用者的作用域继承，而从函数定义处的作用域继承。举个例子，在下列的代码中 `foo` 中的 `x` 指向的是模块 `Bar` 的全局作用域中的 `x`。

```
julia> module Bar
           x = 1
           foo() = x
       end;
```

并且在 `foo` 被使用的地方 `x` 并不在作用域中：

```
julia> import .Bar

julia> x = -1;
```

```
julia> Bar.foo()
1
```

Thus *lexical scope* means that what a variable in a particular piece of code refers to can be deduced from the code in which it appears alone and does not depend on how the program executes. A scope nested inside another scope can "see" variables in all the outer scopes in which it is contained. Outer scopes, on the other hand, cannot see variables in inner scopes.

11.1 全局作用域

每个模块会引进一个新的全局作用域，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。值得注意的是变量绑定只能在它们的全局作用域中改变，在外部模块中不行。作为一个逃生窗口，你总是可以执行该模块内的代码来修改一个变量；这特别保证了不调用“eval”的外部代码绝不会修改模块绑定。

```
julia> module A
           a = 1 # a global in A's scope
       end;

julia> module B
           module C
               c = 2
           end
           b = C.c    # can access the namespace of a nested global scope
                      # through a qualified access
           import ..A # makes module A available
           d = A.a
       end;

julia> module D
           b = a # errors as D's global scope is separate from A's
       end;
ERROR: UndefVarError: a not defined

julia> module E
           import ..A # make module A available
           A.a = 2    # throws below error
       end;
ERROR: cannot assign variables in other modules
```

注意交互式提示行（即 REPL）是在模块 Main 的全局作用域中。

11.2 局部作用域

A new local scope is introduced by most code blocks (see above table for a complete list). Some programming languages require explicitly declaring new variables before using them. Explicit declaration works in Julia too: in any local scope, writing `local x` declares a new local variable in that scope, regardless of whether there is already a variable named x in an outer scope or not. Declaring each new local like this is somewhat verbose and tedious, however, so Julia, like many other languages, considers assignment to a new variable in a local scope to implicitly declare that variable as a new local. Mostly this is pretty intuitive, but as with many things that behave intuitively, the details are more subtle than one might naïvely imagine.

When `x = <value>` occurs in a local scope, Julia applies the following rules to decide what the expression means based on where the assignment expression occurs and what x already refers to at that location:

1. **Existing local:** If x is *already a local variable*, then the existing local x is assigned;
2. **Hard scope:** If x is *not already a local variable* and assignment occurs inside of any hard scope construct (i.e. within a let block, function or macro body, comprehension, or generator), a new local named x is created in the scope of the assignment;
3. **Soft scope:** If x is *not already a local variable* and all of the scope constructs containing the assignment are soft scopes (loops, `try`/`catch` blocks, or `struct` blocks), the behavior depends on whether the global variable x is defined:
   - if global x is *undefined*, a new local named x is created in the scope of the assignment;
   - if global x is *defined*, the assignment is considered ambiguous:
     * in *non-interactive* contexts (files, eval), an ambiguity warning is printed and a new local is created;
     * in *interactive* contexts (REPL, notebooks), the global variable x is assigned.

You may note that in non-interactive contexts the hard and soft scope behaviors are identical except that a warning is printed when an implicitly local variable (i.e. not declared with `local x`) shadows a global. In interactive contexts, the rules follow a more complex heuristic for the sake of convenience. This is covered in depth in examples that follow.

Now that you know the rules, let's look at some examples. Each example is assumed to be evaluated in a fresh REPL session so that the only globals in each snippet are the ones that are assigned in that block of code.

We'll begin with a nice and clear-cut situation—assignment inside of a hard scope, in this case a function body, when no local variable by that name already exists:

```
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
```

Inside of the `greet` function, the assignment `x = "hello"` causes x to be a new local variable in the function's scope. There are two relevant facts: the assignment occurs in local scope and there is no existing local x variable. Since x is local, it doesn't matter if there is a global named x or not. Here for example we define `x = 123` before defining and calling `greet`:

```
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
```

Since the x in `greet` is local, the value (or lack thereof) of the global x is unaffected by calling `greet`. The hard scope rule doesn't care whether a global named x exists or not: assignment to x in a hard scope is local (unless x is declared global).

The next clear cut situation we'll consider is when there is already a local variable named x, in which case `x = <value>` always assigns to this existing local x. The function `sum_to` computes the sum of the numbers from one up to n:

```
function sum_to(n)
    s = 0 # new local
    for i = 1:n
        s = s + i # assign existing local
    end
    return s # same local
end
```

As in the previous example, the first assignment to s at the top of `sum_to` causes s to be a new local variable in the body of the function. The `for` loop has its own inner local scope within the function scope. At the point where `s = s + i` occurs, s is already a local variable, so the assignment updates the existing s instead of creating a new local. We can test this out by calling `sum_to` in the REPL:

```
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
```

Since s is local to the function `sum_to`, calling the function has no effect on the global variable s. We can also see that the update `s = s + i` in the `for` loop must have updated the same s created by the initialization `s = 0` since we get the correct sum of 55 for the integers 1 through 10.

Let's dig into the fact that the `for` loop body has its own scope for a second by writing a slightly more verbose variation which we'll call `sum_to′`, in which we save the sum `s + i` in a variable t before updating s:

```
julia> function sum_to′(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′ (generic function with 1 method)

julia> sum_to′(10)
(55, false)
```

This version returns `s` as before but it also uses the `@isdefined` macro to return a boolean indicating whether there is a local variable named `t` defined in the function's outermost local scope. As you can see, there is no `t` defined outside of the `for` loop body. This is because of the hard scope rule again: since the assignment to `t` occurs inside of a function, which introduces a hard scope, the assignment causes `t` to become a new local variable in the local scope where it appears, i.e. inside of the loop body. Even if there were a global named `t`, it would make no difference—the hard scope rule isn't affected by anything in global scope.

Let's move onto some more ambiguous cases covered by the soft scope rule. We'll explore this by extracting the bodies of the `greet` and `sum_to′` functions into soft scope contexts. First, let's put the body of `greet` in a `for` loop—which is soft, rather than hard—and evaluate it in the REPL:

```
julia> for i = 1:3
            x = "hello" # new local
            println(x)
        end
hello
hello
hello

julia> x
ERROR: UndefVarError: x not defined
```

Since the global `x` is not defined when the `for` loop is evaluated, the first clause of the soft scope rule applies and `x` is created as local to the `for` loop and therefore global `x` remains undefined after the loop executes. Next, let's consider the body of `sum_to′` extracted into global scope, fixing its argument to `n = 10`

```
s = 0
for i = 1:10
    t = s + i
    s = t
end
s
@isdefined(t)
```

What does this code do? Hint: it's a trick question. The answer is "it depends." If this code is entered interactively, it behaves the same way it does in a function body. But if the code appears in a file, it prints an ambiguity warning and throws an undefined variable error. Let's see it working in the REPL first:

```
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
```

The REPL approximates being in the body of a function by deciding whether assignment inside the loop assigns to a global or creates new local based on whether a global variable by that name is defined or not. If a global by the name exists, then the assignment updates it. If no global exists, then the assignment creates a new local variable. In this example we see both cases in action:

- There is no global named `t`, so `t = s + i` creates a new `t` that is local to the `for` loop;
- There is a global named `s`, so `s = t` assigns to it.

The second fact is why execution of the loop changes the global value of `s` and the first fact is why `t` is still undefined after the loop executes. Now, let's try evaluating this same code as though it were in a file instead:

```
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
```

Here we use `include_string`, to evaluate `code` as though it were the contents of a file. We could also save `code` to a file and then call `include` on that file—the result would be the same. As you can see, this behaves quite different from evaluating the same code in the REPL. Let's break down what's happening here:

- global `s` is defined with the value `0` before the loop is evaluated
- the assignment `s = t` occurs in a soft scope—a `for` loop outside of any function body or other hard scope construct
- therefore the second clause of the soft scope rule applies, and the assignment is ambiguous so a warning is emitted
- execution continues, making `s` local to the `for` loop body
- since `s` is local to the `for` loop, it is undefined when `t = s + i` is evaluated, causing an error

- evaluation stops there, but if it got to `s` and `@isdefined(t)`, it would return `0` and `false`.

This demonstrates some important aspects of scope: in a scope, each variable can only have one meaning, and that meaning is determined regardless of the order of expressions. The presence of the expression `s = t` in the loop causes `s` to be local to the loop, which means that it is also local when it appears on the right hand side of `t = s + i`, even though that expression appears first and is evaluated first. One might imagine that the `s` on the first line of the loop could be global while the `s` on the second line of the loop is local, but that's not possible since the two lines are in the same scope block and each variable can only mean one thing in a given scope.

On Soft Scope

We have now covered all the local scope rules, but before wrapping up this section, perhaps a few words should be said about why the ambiguous soft scope case is handled differently in interactive and non-interactive contexts. There are two obvious questions one could ask:

1. Why doesn't it just work like the REPL everywhere?
2. Why doesn't it just work like in files everywhere? And maybe skip the warning?

In Julia ≤ 0.6, all global scopes did work like the current REPL: when `x = <value>` occurred in a loop (or `try/catch`, or `struct` body) but outside of a function body (or `let` block or comprehension), it was decided based on whether a global named `x` was defined or not whether `x` should be local to the loop. This behavior has the advantage of being intuitive and convenient since it approximates the behavior inside of a function body as closely as possible. In particular, it makes it easy to move code back and forth between a function body and the REPL when trying to debug the behavior of a function. However, it has some downsides. First, it's quite a complex behavior: many people over the years were confused about this behavior and complained that it was complicated and hard both to explain and understand. Fair point. Second, and arguably worse, is that it's bad for programming "at scale." When you see a small piece of code in one place like this, it's quite clear what's going on:

```
s = 0
for i = 1:10
    s += i
end
```

Obviously the intention is to modify the existing global variable `s`. What else could it mean? However, not all real world code is so short or so clear. We found that code like the following often occurs in the wild:

```
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
```

It's far less clear what should happen here. Since `x + "hello"` is a method error, it seems probable that the intention is for `x` to be local to the `for` loop. But runtime values and what methods happen to exist cannot be used to determine the scopes of variables. With the Julia ≤ 0.6 behavior, it's especially concerning that someone might have written the `for` loop first, had it working just fine, but later when someone else adds a new global far away—possibly in a different file—the code suddenly changes meaning and either breaks noisily or, worse still, silently does the wrong thing. This kind of "spooky action at a distance" is something that good programming language designs should prevent.

So in Julia 1.0, we simplified the rules for scope: in any local scope, assignment to a name that wasn't already a local variable created a new local variable. This eliminated the notion of soft scope entirely as well as removing the potential for spooky action. We uncovered and fixed a significant number of bugs due to the removal of soft scope, vindicating the choice to get rid of it. And there was much rejoicing! Well, no, not really. Because some people were angry that they now had to write:

```
s = 0
for i = 1:10
    global s += i
end
```

Do you see that `global` annotation in there? Hideous. Obviously this situation could not be tolerated. But seriously, there are two main issues with requiring `global` for this kind of top-level code:

1. It's no longer convenient to copy and paste the code from inside a function body into the REPL to debug it—you have to add `global` annotations and then remove them again to go back;
2. Beginners will write this kind of code without the `global` and have no idea why their code doesn't work—the error that they get is that `s` is undefined, which does not seem to enlighten anyone who happens to make this mistake.

As of Julia 1.5, this code works without the `global` annotation in interactive contexts like the REPL or Jupyter notebooks (just like Julia 0.6) and in files and other non-interactive contexts, it prints this very direct 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.

This addresses both issues while preserving the "programming at scale" benefits of the 1.0 behavior: global variables have no spooky effect on the meaning of code that may be far away; in the REPL copy-and-paste debugging works and beginners don't have any issues; any time someone either forgets a `global` annotation or accidentally shadows an existing global with a local in a soft scope, which would be confusing anyway, they get a nice clear warning.

An important property of this design is that any code that executes in a file without a warning will behave the same way in a fresh REPL. And on the flip side, if you take a REPL session and save it to file, if it behaves differently than it did in the REPL, then you will get a warning.

let 块

不像局部变量的赋值行为，`let` 语句每次运行都新建一个新的变量绑定。赋值改变的是已存在值的位置，`let` 会新建新的位置。这个区别通常都不重要，只会在通过闭包跳出作用域的变量的情况下能探测到。`let` 语法接受由逗号隔开的一系列的赋值和变量名：

```
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
```

这个赋值会按顺序评估，在左边的新变量被引入之前右边的每隔两都会在作用域中被评估。所以编写像 `let x = x` 这样的东西是有意义的，因为两个 `x` 变量是不一样的，拥有不同的存储位置。这里有个例子，在例子中 `let` 的行为是必须的：

```
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
```

这里我创建并存储了两个返回变量 `i` 的闭包。但是这两个始终是同一个变量 `i`。所以这两个闭包行为是相同的。我们可以使用 `let` 来为 `i` 创建新的绑定：

```
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
```

因为 `begin` 结构不会引入新的作用域，使用没有参数的 `let` 来只引进一个新的作用域块而不创建新的绑定可能是有用的：

```
julia> let
           local x = 1
           let
               local x = 2
           end
           x
       end
1
```

因为 `let` 引进了一个新的作用域块，内部的局部 `x` 与外部的局部 `x` 是不同的变量。

Loops and Comprehensions

In loops and comprehensions, new variables introduced in their body scopes are freshly allocated for each loop iteration, as if the loop body were surrounded by a `let` block, as demonstrated by this example:

```
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的变量：

```
julia> function f()
           i = 0
           for i = 1:3
               # empty
           end
           return i
       end;

julia> f()
0
```

但是，有时重复使用一个存在的局部变量作为迭代变量是有用的。这能够通过添加关键字 `outer` 来方便地做到：

```
julia> function f()
           i = 0
           for outer i = 1:3
               # empty
           end
           return i
       end;

julia> f()
3
```

11.3 常量

变量的经常的一个使用方式是给一个特定的不变的值一个名字。这样的变量只会被赋值一次。这个想法可以通过使用 `const` 关键字传递给编译器：

```
julia> const e  = 2.71828182845904523536;

julia> const pi = 3.14159265358979323846;
```

多个变量可以使用单个 `const` 语句进行声明：

```
julia> const a, b = 1, 2
(1, 2)
```

`const` 声明只应该在全局作用域中对全局变量使用。编译器很难为包含全局变量的代码优化，因为它们的值（甚至它们的类型）可以任何时候改变。如果一个全局变量不会改变，添加 `const` 声明会解决这个问题。

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。

特别的顶层赋值，比如使用 `function` 和 `structure` 关键字进行的，默认是不变的。

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下列情景是可能的：

- 如果一个新值的类型与常量类型不一样时会扔出一个错误：

```
julia> const x = 1.0
1.0

julia> x = 1
ERROR: invalid redefinition of constant x
```

- 如果一个新值的类型与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
```

- 如果赋值不会导致变量值的变化，不会给出任何信息：

```
julia> const z = 100
100

julia> z = 100
100
```

最后一条规则适用于不可变对象，即使变量绑定会改变，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
```

但是对于可变对象，警告会如预期出现：

```
julia> const a = [1]
1-element Array{Int64,1}:
 1

julia> a = [1]
WARNING: redefinition of constant a. This may fail, cause incorrect answers, or produce other
↪  errors.
1-element Array{Int64,1}:
 1
```

Note that although sometimes possible, changing the value of a `const` variable is strongly discouraged, and is intended only for convenience during interactive use. Changing constants can cause various problems or unexpected behaviors. For instance, if a method references a constant and is already compiled before the constant is changed, then it might keep using the old value:

```
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
```

Chapter 12

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，但通过允许指出某些变量具有特定类型，获得了静态类型系统的一些优点。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 的默认行为是允许变量为任何类型。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。

Julia 用类型系统的术语描述是动态（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类型，并且只有抽象类型可以作为其超类型。虽然起初看起来这可能过于严格，但它有许多有益的结果，但缺点却少得出奇。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。
- 值有类型，变量没有类型——变量仅仅是绑定给值的名字而已。
- 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、快速和健壮。

12.1 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，这可能帮助程序提升性能，在某些情况下

当被附加到一个计算值的表达式时，:: 操作符读作「是 · · · · · · 的实例」。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
```

可以在任何表达式的所在位置做类型断言。

当被附加到赋值左侧的变量或作为 `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> foo()
100

julia> typeof(ans)
Int8
```

这个特性用于避免性能「陷阱」，即给一个变量赋值时意外更改了类型。

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
```

并应用于整个当前作用域，甚至在该声明之前。目前，类型声明不能在全局作用域中使用，例如在 REPL 中就不可以，因为 Julia 还没有常量类型的全局变量。

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
```

此函数的返回值就像赋值给了一个类型已被声明的变量：返回值始终转换为 Float64。

12.2 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，从而描述由相关具体类型组成的集合：那些作为其后代的具体类型。我们从抽象类型开始，即使它们没有实例，因为它们是类型系统的主干：它们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，提供了具体类型可以适应的上下文。例如，这允许你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。

使用 abstract type 关键字来声明抽象类型。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——一个预定义的抽象类型，所有对象都是它的实例并且所有类型都是它的子类型。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。反过来，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」，并被用于像这样的声明右侧类型是新声明类型的直接超类型。它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 接下来会在内部定义并编译一个名为 `myplus` 的方法，专门用于基于上面给出的泛型函数的两个 `Int` 参数，即它定义并编译：

```
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，之后可以通过许多具体类型的组合将其用作默认方法。多亏了多重分派，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。）

12.3 原始类型

Warning

在新的复合类型中包装现有的基元类型，几乎总是比定义自己的基元类型更好。

这个功能允许 Julia 引导 LLVM 支持的标准基本类型。一旦它们被定义，就没有理由再定义更多了。

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。

12.4 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
```

当像函数一样使用类型时，它被称为构造函数。有两个构造函数会被自动生成（这些构造函数称为默认构造函数）。一个接受任何参数并通过调用 `convert` 函数将它们转换为字段的类型，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
```

使用 `struct` 声明的对象都是不可变的，它们在构造后无法修改。一开始看来这很奇怪，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。
- 不可能违反由类型的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

没有字段的不可变复合类型是单态类型；这种类型只能有一个实例：

```
julia> struct NoFields
       end

julia> NoFields() === NoFields()
true
```

=== 函数用来确认构造出来的「两个」NoFields 实例实际上是同一个。单态类型将在下面进一步详细描述。

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。

12.5 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
```

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。

12.6 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。

12.7 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。

12.8 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含地给出。

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> Point{Float64}(1.0, 2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(ans)
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。

在许多情况下，提供想要构造的 `Point` 对象的类型是多余的，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` 的隐含值是确定的：

```
julia> Point(1.0,2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(ans)
Point{Float64}

julia> Point(1,2)
Point{Int64}(1, 2)

julia> typeof(ans)
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 at none:2
```

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
```

在这里给出一个真实示例，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——不是函数本身的。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：`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类型 `Vararg`，它表示任意数量的尾随参数：

```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString,Vararg{Int64,N} where N}

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
```

请注意，`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）。

类型 `Vararg{T,N}` 对应于正好 `N` 个类型为 `T` 的元素。`NTuple{N,T}` 是 `Tuple{Vararg{T,N}}` 的别名，即包含正好 `N` 个类型为 `T` 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。

```
julia> typeof((a=1,b="hello"))
NamedTuple{(:a, :b),Tuple{Int64,String}}
```

@NamedTuple 宏提供了类结构体（struct）的具名元组（NamedTuple）声明，使用 key::Type 的语法，如果省略 ::Type 则默认为 ::Any。

```
julia> @NamedTuple{a::Int, b::String}
NamedTuple{(:a, :b),Tuple{Int64,String}}

julia> @NamedTuple begin
           a::Int
           b::String
       end
NamedTuple{(:a, :b),Tuple{Int64,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。

单态类型

这里必须提到一种特殊的抽象类型：单态类型。对于每个类型 T，「单态类型」Type{T} 是个抽象类型且唯一的实例就是对象 T。由于定义有点难以解释，让我们看一些例子：

```
julia> isa(Float64, Type{Float64})
true

julia> isa(Real, Type{Float64})
false

julia> isa(Real, Type{Real})
true

julia> isa(Float64, Type{Real})
false
```

换种说法，isa(A,Type{B}) 为真当且仅当 A 与 B 是同一对象且该对象是一个类型。不带参数时，Type 是个抽象类型，所有类型对象都是它的实例，当然也包括单态类型：

```
julia> isa(Type{Float64}, Type)
true

julia> isa(Float64, Type)
true

julia> isa(Real, Type)
true
```

只有对象是类型时，才是 Type 的实例：

```
julia> isa(1, Type)
false

julia> isa("foo", Type)
false
```

在我们讨论参数方法和类型转换之前，很难解释单态类型的作用，但简而言之，它允许针对特定类型值专门指定函数行为。这对于编写方法（尤其是参数方法）很有用，这些方法的行为取决于作为显式参数给出的类型，而不是隐含在它的某个参数的类型中。

一些流行的语言有单态类型，比如 Haskell、Scala 和 Ruby。在一般用法中，术语「单态类型」指的是唯一实例为单个值的类型。这定义适用于 Julia 的单态类型，但需要注意的是 Julia 里只有类型对象具有对应的单态类型。

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
```

12.9 UnionAll 类型

我们已经说过像 Ptr 这样的参数类型充当它所有实例（Ptr{Int64} 等）的超类型。这是如何工作的？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，并先把 B 替换为 A 中最外层的类型变量。结果应该是另一个 UnionAll 类型，然后把 C 替换为该类型的类型变量。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，而第二个元素是任意种类的数组 Array，且该数组的元素类型包含于第一个元组元素的类型。

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}
Array{Array{T,1} where T,1}

julia> const T2 = Array{Array{T,1}, 1} where T
Array{Array{T,1},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。

12.10 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
```

在 base/boot.jl 中，通过以下代码实现：

```
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。）

12.11 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
```

已经在手册各处的示例中使用的 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:
[...]
```

12.12 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)，我们将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 Array{Polar{Float64},1}:
 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，但如果你想，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），所以此输出不忠实地表示了表达式 `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的情况下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，它允许一起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)
               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，将使用这个新的紧凑表示。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 Array{Polar{Float64},2}:
 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。

12.13 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，在那里 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}()，它可以作为一种习惯的方式来利用这种技术需要更精细的层次结构。这可以作为利用这种技术的惯用方式，而且不需要更精细的层次结构。

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`；情况不太好时，你很容易使代码性能变得更糟糕。一般使用时，你可能从来不会想要写出上方示例那样的代码。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，目前设置为 4。

Chapter 13

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。

13.1 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)
```

[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.

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) at none:1

julia> f(Float32(2.0), 3.0)
ERROR: MethodError: no method matching f(::Float32, ::Float64)
Closest candidates are:
  f(!Matched::Float64, ::Float64) at none:1

julia> f(2.0, "3.0")
ERROR: MethodError: no method matching f(::Float64, ::String)
Closest candidates are:
  f(::Float64, !Matched::Float64) at none:1

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) at none:1

julia> f()
ERROR: MethodError: no method matching f()
Closest candidates are:
  f(!Matched::Float64, !Matched::Float64) at none:1
  f(!Matched::Number, !Matched::Number) at none:1
```

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)
```

这个输出告诉我们 f 是有两个方法的函数对象。为了找出那些方法的特征是什么，使用 methods函数：

```
julia> methods(f)
# 2 methods for generic function "f":
[1] f(x::Float64, y::Float64) in Main at none:1
[2] f(x::Number, y::Number) in Main at 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> f("foo", 1)
Whoa there, Nelly.
```

这个接受所有的方法比其他的对一堆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。

虽然这像是一个简单的概念，基于值的类型的多重分派可能是 Julia 语言的一个最强大和中心特性。核心运算符都典型地含有很多方法：

```
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
```

多重分派和灵活的参数类型系统让 Julia 有能力抽象地表达高层级算法，而与实现细节解耦，也能生成高效而专用的代码来在运行中处理每个情况。

13.2 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::Float64, y) in Main at none:1
  g(x, y::Float64) in Main at none:1
Possible fix, define
  g(::Float64, ::Float64)
```

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。

13.3 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 Array{Int64,1}:
 1
 2
 3
 4

julia> myappend([1,2,3],2.5)
ERROR: MethodError: no method matching myappend(::Array{Int64,1}, ::Float64)
Closest candidates are:
  myappend(::Array{T,1}, !Matched::T) where T at none:1

julia> myappend([1.0,2.0,3.0],4.0)
4-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0

julia> myappend([1.0,2.0,3.0],4)
ERROR: MethodError: no method matching myappend(::Array{Float64,1}, ::Int64)
Closest candidates are:
  myappend(::Array{T,1}, !Matched::T) where T at none:1
```

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 at none:1
  same_type_numeric(!Matched::Number, ::Number) at none:1

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。

13.4 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"
```

13.5 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

这里是一个正确地代码模板，它返回 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
```

使用了所谓的三角分派。注意如果 T 是一个 UnionAll 类型，比如 eltype(Array{T} where T <: Integer)，会返回 Any（如同 Base 中的 eltype 一样）。

另外一个方法，这是在 Julia v0.6 中的三角分派到来之前的唯一正确方法，是:

```
abstract type AbstractArray{T, N} end
eltype(::Type{AbstractArray}) = Any
eltype(::Type{AbstractArray{T}}) where {T} = T
eltype(::Type{AbstractArray{T, N}}) where {T, N} = T
eltype(::Type{A}) where {A<:AbstractArray} = eltype(supertype(A))
```

另外一个可能性如下例，这可以对适配那些参数 T 需要更严格匹配的情况有用:

```
eltype(::Type{AbstractArray{T, N} where {T<:S, N<:M}}) where {M, S} = Any
eltype(::Type{AbstractArray{T, N} where {T<:S}}) where {N, S} = Any
eltype(::Type{AbstractArray{T, N} where {N<:M}}) where {M, T} = T
eltype(::Type{AbstractArray{T, N}}) where {T, N} = T
eltype(::Type{A}) where {A <: AbstractArray} = eltype(supertype(A))
```

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!

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。(快速复习：你明白为什么吗 ?)

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，保证输入与输出是同一种类型:

```
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。

上一节的例子掩盖了map和promote的实现细节，这两个都是依据 trait 来进行运算的。当对一个矩阵进行迭代，比如 map 的实现中，一个重要的问题是按照什么顺序去遍历数据。当 AbstractArray 的子类型实现了Base.IndexStyletrait，其他函数，比如 map 就可以根据这个信息进行分派，以选择最好的算法（参见抽象数组接口）。这意味着每个子类型就没有必要去实现对应的 map 版本，因为通用的定义加 trait 类就能让系统选择最快的版本。这里一个玩具似的 map 实现说明了基于 trait 的分派：

```
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)...)
```

13.6 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) at none:1

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) at none:1
```

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。

13.7 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。

13.8 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。

13.9 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
```

13.10 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
```

这是经常使用的对的方案；但是有些环境下盲目地遵从这个建议会适得其反。特别地，范用函数有的方法越多，出现歧义的可能性越高。当你的方法层级比这些简单的例子更加复杂时，就值得你花时间去仔细想想其他的方案。

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。

[2] Arthur C. Clarke, *Profiles of the Future* (1961): Clarke's Third Law.

Chapter 14

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。

14.1 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。

14.2 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。

14.3 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。这些纯数据包括原始类型（比如 Int ）和由其它纯数据类型构成的不可变结构体。纯数据类型的初始值是未定义的：

```
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。

14.4 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);
```

此方法采用了 `convert` 函数，显式地将 `x` 转化成了 `Float64` 类型，之后再委托前面讲到的那个通用的外部构造函数来进行具体的构造工作，经过转化，两个参数的类型都是 `Float64`，所以可以正确构造出一个 `Point{Float64}` 对象，而不会像之前那样触发 `MethodError`。

```
julia> Point(1,2.5)
Point{Float64}(1.0, 2.5)

julia> typeof(ans)
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 at none:1
```

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)
```

所以，即使 Julia 提供的默认内部构造函数对于类型参数的要求非常严格，我们也有方法将其变得更加易用。正因为构造函数可以充分发挥类型系统、方法以及多重分派的作用，定义复杂的行为也会变得非常简单。

14.5 案例分析：分数的实现

上文主要讲了关于参数复合类型及其构造函数的一些零散内容，或许将这些内容结合起来的一个最佳方法是分析一个真实的案例。为此，我们来实现一个我们自己的分数类型 OurRational，它与 Julia 内置的分数类型 Rational 很相似，它的定义在 rational.jl 里：

```
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
               g = gcd(den, num)
               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。

现在事情开始变得有意思了，`OurRational` 只有一个内部构造函数，它的作用是检查 `num` 和 `den` 是否为 `0`，并确保构建的每个分数都是经过约分化简的形式，且分母为非负数。这可以令分子和分母同时除以它们的最大公约数来实现，最大公约数可以用 Julia 内置的 `gcd` 函数计算。由于 `gcd` 返回的最大公约数的符号是跟第一个参数 `den` 一致的，所以约分后一定会保证 `den` 的值为非负数。因为这是 `OurRational` 的唯一一个内部构造函数，所以我们可以确保构建出的 `OurRational` 对象一定是这种化简的形式。

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。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`，当然这要求 `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}
false
```

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，而且是自包涵的。

14.6 Outer-only constructors

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数组以及其累加和：

```
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(::Array{Int32,1}, ::Int32)
Closest candidates are:
  SummedArray(::Array{T,1}) where T at none:4
```

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 15

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。

15.1 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> convert(UInt8, x)
0x0c

julia> typeof(ans)
UInt8

julia> convert(AbstractFloat, x)
12.0

julia> typeof(ans)
Float64

julia> a = Any[1 2 3; 4 5 6]
2×3 Array{Any,2}:
 1  2  3
 4  5  6

julia> convert(Array{Float64}, a)
2×3 Array{Float64,2}:
 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)
```

此方法的第一个参数的类型是单态类型 Type{MyType}，其唯一实例是 MyType。因此，此方法仅在第一个参数是类型值 MyType 时才被调用。请注意第一个参数使用的语法：在 :: 符号之前省略了参数名，只是给出了参数类型。这是 Julia 中用于函数参数的语法，该参数的类型已经指定，但其值无需通过名称引用。在此例中，由于参数类型是单态类型，我们已经知道其值而无需引用参数名称。

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)
```

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。

15.2 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)
```

浮点值被提升为最大的浮点参数类型。整数值会被提升为本机机器字大小或最大的整数参数类型中较大的一个。整数和浮点值的混合会被提升为一个足以包含所有值的浮点类型。与有理数混合的整数会被提升有理数。与浮点数混合的有理数会被提升为浮点数。与实数值混合的复数值会被提升为合适类型的复数值。

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> Rational(Int8(15),Int32(-5))
-3//1

julia> typeof(ans)
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
```

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。但是，它本身可能是有用的。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。

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 16

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。

16.1 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**
`IteratorSize(IterType)`	`HasLength()`	`HasLength()`，`HasShape{N}()`，`IsInfinite()` 或者 `SizeUnknown()` 中合适的一个
`IteratorEltype(IterType)`	`HasEltype()`	`EltypeUnknown()` 或 `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
`HasLength()`	`length(iter)`
`HasShape{N}()`	`length(iter)` 和 `size(iter, [dim])`
`IsInfinite()`	（无）
`SizeUnknown()`	（无）

由 `IteratorEltype(IterType)` 返回的值	必需方法
`HasEltype()`	`eltype(IterType)`
`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 i in iter   # or  "for i = iter"
    # body
end
```

以上代码被解释为：

```
next = iterate(iter)
while next !== nothing
    (i, 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 i in Squares(7)
           println(i)
       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，比如标准库 `Statistics` 中的 `in`，`mean` 和 `std` 。

```
julia> 25 in Squares(10)
true

julia> using Statistics

julia> mean(Squares(100))
3383.5

julia> std(Squares(100))
3024.355854282583
```

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}`，而不是盲目地 `push!` 每一个元素到 `Vector{Any}`：

```
julia> collect(Squares(4))
4-element Array{Int64,1}:
  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 Array{Int64,1}:
 16
  9
  4
  1
```

16.2 Indexing

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]`

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`:

```
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
```

但请注意，上面只定义了带有一个整数索引的 `getindex`。使用除 `Int` 外的任何值进行索引会抛出 `MethodError`，表示没有匹配的方法。为了支持使用某个范围内的 `Int` 或 `Int` 向量进行索引，必须编写单独的方法：

```
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 Array{Int64,1}:
  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。

16.3 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。

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：

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
setindex!(A, v, i::Int)		（若为 IndexLinear）线性索引元素赋值
setindex!(A, v, I::Vararg{Int, N})		（若为 IndexCartesian，其中 N = ndims(A)）N 维标量索引元素赋值
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。
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
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）

```
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 Array{Int64,1}:
  9
 16

julia> s + s
4-element Array{Int64,1}:
  2
  8
 18
 32

julia> sin.(s)
4-element Array{Float64,1}:
  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{T}(A::SparseArray, ::Type{T}, dims::Dims) 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。

16.4 Strided 数组

Strided 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：

需要实现的方法		简短描述
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}, A)		返回数组的本地内存地址。
可选方法	**默认定义**	**简短描述**
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）。

```
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.
```

16.5 自定义广播

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

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：

- 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。少数其它类型（包括但不限于类型本身、函数、像 missing 和 nothing 这样的特殊单态类型以及日期）为了能被广播，Base.broadcastable 会返回封装在 Ref 的参数来充当 0 维「标量」。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：

```
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> 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)(_max(Val(M),Val(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。

Chapter 17

模块

Julia 中的模块（module）是一些互相隔离的可变工作空间，也就是说它们会引入新的全局作用域。它们在语法上以 `module Name ... end` 界定。模块允许你创建顶层定义（也称为全局变量），而无需担心命名冲突。在模块中，利用导入（importing），你可以控制其它模块中的哪些名称是可见的；利用导出（exporting），你可以控制你自己的模块中的哪些名称是公开的。

下面的示例演示了模块的主要功能。它不是为了运行，只是为了方便说明：

```
module MyModule
using Lib

using BigLib: thing1, thing2

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
```

注意，模块中的代码样式不需要缩进，否则的话，会导致整个文件缩进。

上面的模块定义了一个 `MyType` 类型，以及两个函数，其中，函数 `foo` 和类型 `MyType` 被导出了，因而可以被导入到其它模块，而函数 `bar` 是模块 `MyModule` 的私有函数。

`using Lib` 意味着一个名称为 `Lib` 的模块会在需要的时候用于解释变量名。当一个全局变量在当前模块中没有定义时，系统就会从 `Lib` 中导出的变量中搜索该变量，如果找到了的话，就导入进来。也就是说，当前模块中，所有使用该全局变量的地方都会解释为 `Lib` 中对应的变量。

代码 `using BigLib: thing1, thing2` 显式地将标识符 `thing1` 和 `thing2` 从模块 `BigLib` 中引入到当前作用域。如果这两个变量是函数的话，则**不允许**给它们增加新的方法，毕竟代码里写的是"using"（使用）它们，而不是扩展它们。

`import` 关键字所支持的语法与 `using` 一致。它并不会像 `using` 那样将模块添加到搜索空间中。与 `using` 不同，`import` 引入的函数 **可以**为其增加新的方法。

前面的 MyModule 模块中，我们希望给 show 函数增加一个方法，需要写成 import Base.show。如果用 using 的话，就不能扩展 show 函数。通过 using 导入才可见的名字是不能被扩展的。

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。

17.1 模块用法摘要

要导入一个模块，可以用 using 或 import 关键字。为了更好地理解它们的区别，请参考下面的例子：

```
module MyModule

export x, y

x() = "x"
y() = "y"
p() = "p"

end
```

这个模块用关键字 export 导出了 x 和 y 函数，此外还有一个没有被导出的函数 p。想要将该模块及其内部的函数导入当前模块有以下方法：

导入代码	当前作用域导入了哪些变量？	可增加新方法的名字
using MyModule	All exported names (x and y), MyModule.x, MyModule.y and MyModule.p	MyModule.x, MyModule.y and MyModule.p
using MyModule: x, p	x and p	
import MyModule	MyModule.x、MyModule.y 和 MyModule.p	MyModule.x、MyModule.y 和 MyModule.p
import MyModule.x, MyModule.p	x 和 p	x 和 p
import MyModule: x, p	x 和 p	x 和 p

模块和文件

模块与文件和文件名无关；模块只与模块表达式有关。一个模块可以有多个文件，一个文件也可以有多个模块。

```
module Foo

include("file1.jl")
include("file2.jl")

end
```

在不同的模块中引入同一段代码，可以提供一种类似 mixin 的行为。我们可以利用这个特性来观察，在不同的定义下，执行同一段代码会有什么结果。例如，在测试的时候，可以使用某些「安全」的运算符。

```
module Normal
include("mycode.jl")
end

module Testing
include("safe_operators.jl")
include("mycode.jl")
end
```

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。

默认顶层定义以及裸模块

除了默认包含 `using Base` 之外，所有模块都还包含 `eval` 和 `include` 函数。这两个函数用于在对应模块的全局环境中，执行表达式或文件。

如果连这些默认的定义都不需要，那么可以用 `baremodule` 定义裸模块（不过 `Core` 模块仍然会被引入，否则啥也干不了）。用裸模块表达的标准模块定义如下：

```
baremodule Mod

using Base

eval(x) = Core.eval(Mod, x)
include(p) = Base.include(Mod, p)

...

end
```

模块的绝对路径和相对路径

给定语句 `using Foo`，系统在顶层模块的内部表中查找名为 `Foo` 的包。如果模块不存在，系统会尝试 `require(:Foo)`，这通常会从已安装的包中加载代码。

但是，某些模块包含子模块，这意味着你有时需要访问非顶层模块。有两种方法可以做到这一点。第一种是使用绝对路径，例如 `using Base.Sort`。第二种是使用相对路径，这样可以更容易地导入当前模块或其任何封闭模块的子模块：

```
module Parent

module Utils
...
end

using .Utils

...
end
```

这里的模块 Parent 包含一个子模块 Utils，而 Parent 中的代码希望 Utils 的内容可见，这是可以使用 using 加点 . 这种相对路径来实现。添加更多的点会移动到模块层次结构中的更上级别。例如，using ..Utils 会在 Parent 的上级模块中查找 Utils 而不是在 Parent 中查找。

请注意，相对导入符号 . 仅在 using 和 import 语句中有效。

命名空间的相关话题

如果名称是限定的（例如 Base.sin），那么即使它没有被导出，我们也可以访问它。这通常在调试时很有用。若函数名也使用这种限定的方式，就可以为其添加方法。但是，对于函数名仅包含符号的情况，例如一个运算符，Base.+，由于会出现语法歧义，所以必须使用 Base.:+ 来引用它。如果运算符的字符不止一个，则必须用括号括起来，例如：Base.:(==)。

宏名称在导入和导出语句中用 @ 编写，例如：import Mod.@mac。其它模块中的宏可以用 Mod.@mac 或 @Mod.mac 触发。

不允许使用 M.x = y 这种写法给另一个模块中的全局变量赋值；必须在模块内部才能进行全局变量的赋值。

用 global x 声明变量可以仅“保留”名称而不赋值。有些全局变量需要在代码加载后才初始化，这样做可以防止命名冲突。

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。

当用 import 或 using 加载一个模块时，模块增量预编译文件会自动创建并使用。这会让模块在第一次加载时自动编译。另外，你也可以手工调用 Base.compilecache(modulename)，产生的缓存文件会放在 DEPOT_PATH[1]/compiled/ 目录下。之后，当该模块的任何一个依赖发生变更时，该模块会在 using 或 import 时自动重新编译；模块的依赖指的是：任何它导入的模块、Julia 自身、include 的文件或由 include_dependency(path) 显式声明的依赖。

对于文件依赖，判断是否有变动的方法是：在 include 或 include_dependency 的时候检查每个文件的变更时间（mtime）是否没变，或等于截断变更时间。截断变更时间是指将变更时间截断到最近的一秒，这是由于在某些操作系统中，用 mtime 无法获取亚秒级的精度。此外，也会考虑到 require 搜索到的文件路径与之前预编译文件中的是否匹配。对于已经加载到当前进程的依赖，即使它们的文件发成了变更，甚至是丢失，Julia 也不会重新编译这些模块，这是为了避免正在运行的系统与预编译缓存之间的不兼容性。

如果你认为预编译自己的模块是**不**安全的（基于下面所说的各种原因），那么你应该在模块文件中添加 __precompile__(false)，一般会将其写在文件的最上面。这就可以触发 Base.compilecache 报错，并且在直接使用 using / import 加载的时候跳过预编译和缓存。这样做同时也可以防止其它开启预编译的模块加载此模块。

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：它们的定义可以预编译并从缓存的模块映像中加载。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，以便进行预编译（除非将 `Ptr` 对象隐藏在 `isbits` 对象中，否则它们将转换为空指针）。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。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。

Chapter 18

文档

自 Julia 0.4 开始，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 missing, 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 `MersenneTwister` (or other pseudorandom number generator) 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`）。

18.1 访问文档

文档可以在 REPL 中访问，也可以在 IJulia 中通过键入 `?` 紧接函数或者宏的名字并按下 `Enter` 访问。例如，

```
?cos
?@time
?r""
```

会分别为相应的函数，宏或者字符显示文档。在 Juno 中，使用 `Ctrl-J, Ctrl-D` 会为光标处的对象显示文档。

18.2 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。

18.3 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`，中的文档会根据块的评估情况加入文档系统中，例如：

```
if condition()
    "..."
    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::String
end

Docs.getdoc(t::MyType) = "Documentation for MyType with value $(t.value)"

x = MyType("x")
y = MyType("y")
```

输入?x 会显示”Documentation for MyType with value x”，输入?y 则会显示”Documentation for MyType with value y”。

18.4 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
```

通过把 docstring 放在表达式之上来给一个 baremodule 写文档会在模块中自动引入 @doc。它在模块表达式并没有文档时必须手动引入。空的 baremodule 不能有文档。

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 19

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。

19.1 程序表示

每个 Julia 程序均以字符串开始：

```
julia> prog = "1 + 1"
"1 + 1"
```

What happens next?

The next step is to parse each string into an object called an expression, represented by the Julia type Expr:

```
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、其他表达式或字面值：

```
julia> ex1.args
3-element Array{Any,1}:
  :+
 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。第一种形式构造一个 `Symbol`，这是作为表达式组成部分的一个 interned string：

```
julia> :foo
:foo

julia> typeof(ans)
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("func",10)
:func10

julia> Symbol(:var,'_',"sym")
:var_sym
```

注意，要使用：语法，符号的名称必须是有效的标识符。否则，必须使用 `Symbol(str)` 构造函数。

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> :(::)
:(::)
```

19.2 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
```

As we have seen, such expressions support interpolation with $. However, in some situations it is necessary to quote code *without* performing interpolation. This kind of quoting does not yet have syntax, but is represented internally as an object of type QuoteNode:

```
julia> eval(Meta.quot(Expr(:$, :(1+2))))
3

julia> eval(QuoteNode(Expr(:$, :(1+2))))
:($(Expr(:$, :(1 + 2))))
```

The parser yields QuoteNodes for simple quoted items like symbols:

```
julia> dump(Meta.parse(":x"))
QuoteNode
  value: Symbol x
```

QuoteNode can also be used for certain advanced metaprogramming tasks.

Evaluating expressions

Given an expression object, one can cause Julia to evaluate (execute) it at global scope using eval:

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

julia> eval(ans)
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：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
```

19.3 宏

宏提供了在程序的最终主体中包含所生成的代码的方法。宏将参数元组映射到所返回的表达式，且生成的表达式会被直接编译，并不需要运行时的 `eval` 调用。宏的参数可以包括表达式、字面量值和符号。

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
```

We can view the quoted return expression using the function `macroexpand` (**important note:** this is an extremely useful tool for debugging macros):

```
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"))
```

Hold up: why macros?

We have already seen a function `f(::Expr...) -> Expr` in a previous section. In fact, `macroexpand` is also such a function. So, why do macros exist?

Macros are necessary because they execute when code is parsed, therefore, macros allow the programmer to generate and include fragments of customized code *before* the full program is run. To illustrate the difference, consider the following example:

```
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)
```

Now @assert has two modes of operation, depending upon the number of arguments it receives! If there's only one argument, the tuple of expressions captured by msgs will be empty and it will behave the same as the simpler definition above. But now if the user specifies a second argument, it is printed in the message body instead of the failing expression. You can inspect the result of a macro expansion with the aptly named @macroexpand macro:

```
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 是私有的临时变量且 time 引用在 Julia Base 中的 time 函数，而不是用户也许具有的任何 time 变量（对于 println 也是一样）。想象一下，如果用户表达式 ex 中也包含对名为 t0 的变量的赋值、或者定义了自己的 time 变量，则可能会出现问题，我们可能会得到错误或者诡异且不正确的行为。

Julia 的宏展开器以下列方式解决这些问题。首先，宏返回结果中的变量被分为局部变量或全局变量。如果一个变量被赋值（且未声明为全局变量）、声明为局部变量或者用作函数参数名称，则将其视为局部变量。否则，则认为它是全局变量。接着，局部变量重命名为唯一名称（通过生成新符号的 gensym 函数），并在宏定义所处环境中解析全局变量。因此，上述两个问题都被解决了；宏的局部变量不会与任何用户变量相冲突，time 和 println 也将引用其在 Julia Base 中的定义。

然而，仍有另外的问题。考虑此宏的以下用法：

```
module MyModule
import Base.@time

time() = ... # compute something

@time time()
end
```

在这里，用户表达式 ex 是对 time 的调用，但不是宏所使用的 time 函数。它明确地引用 MyModule.time。因此，我们必须将 ex 中的代码安排在宏调用所处环境中解析。这通过用 esc「转义」表达式来完成：

```
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
```

19.4 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
```

19.5 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*(?:#|$)"` 生成一个正则表达式对象而不是一个字符串
- `b"DATA\xff\u2200"` 是字节数组 `[68,65,84,65,255,226,136,128]` 的字面量。

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。

与非标准字符串字面量一样，非标准命令字面量存在使用命令字面量语法的带前缀变种。命令字面量 `custom`literal`` 被解析为 `@custom_cmd "literal"`。Julia 本身不包含任何非标准命令字面量，但包可以使用此语法。除了语法不同以及使用 `_cmd` 而不是 `_str` 后缀，非标准命令字面量的行为与非标准字符串字面量完全相同。

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。

用户定义的字符串字面量的机制十分强大。不仅 Julia 的非标准字面量的实现使用它，而且命令字面量的语法（`` `echo "Hello, $person"` ``）用下面看起来人畜无害的宏实现：

```
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。

19.6 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。因此，我们可以利用生成函数将迭代移动到编译期；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 20

多维数组

与大多数技术计算语言一样，Julia 提供原生的数组实现。大多数技术计算语言非常重视其数组实现，但需要付出使用其它容器的代价。Julia 用同样的方式来处理数组。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，它的性能源自编译器。这样一来，用户就可以通过继承 `AbstractArray` 的方式来创建自定义数组类型。实现自定义数组类型的更多详细信息，请参阅manual section on the AbstractArray interface。

数组是存储在多维网格中对象的集合。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 passed by sharing。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。通常，以一个 ! 结尾的函数名表示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“non-mutating”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。

20.1 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

20.2 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 ...` 可以是一个包含维数大小的元组，也可以表示用任意个参数传递的一系列维数大小值。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`，元素值是 $[0,1)$ 半开区间中的均匀分布且服从一阶独立同分布 [1]
`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-by-n 单位阵。需要 `using LinearAlgebra` for `I`.
`range(start, stop=stop, length=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...)`	一个被值 `x` 填充的 `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 Array{Int8,2}:
 0  0  0
 0  0  0

julia> zeros(Int8, (2, 3))
2×3 Array{Int8,2}:
 0  0  0
 0  0  0

julia> zeros((2, 3))
2×3 Array{Float64,2}:
```

[1] *iid*，独立同分布

```
 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`.

20.3 Array literals

数组也可以直接用方括号来构造; 语法为 `[A, B, C, ...]` 创建一个一维数组 (即一个矢量), 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。

```
julia> [1,2,3] # An array of `Int`s
3-element Array{Int64,1}:
 1
 2
 3

julia> promote(1, 2.3, 4//5) # This combination of Int, Float64 and Rational promotes to Float64
(1.0, 2.3, 0.8)

julia> [1, 2.3, 4//5] # Thus that's the element type of this Array
3-element Array{Float64,1}:
 1.0
 2.3
 0.8

julia> []
Any[]
```

Concatenation

If the arguments inside the square brackets are separated by semicolons (`;`) or newlines instead of commas, then their contents are *vertically concatenated* together instead of the arguments being used as elements themselves.

```
julia> [1:2, 4:5] # Has a comma, so no concatenation occurs. The ranges are themselves the elements
2-element Array{UnitRange{Int64},1}:
 1:2
 4:5

julia> [1:2; 4:5]
4-element Array{Int64,1}:
 1
 2
 4
 5

julia> [1:2
        4:5
        6]
5-element Array{Int64,1}:
 1
```

```
 2
 4
 5
 6
```

Similarly, if the arguments are separated by tabs or spaces, then their contents are *horizontally concatenated* together.

```
julia> [1:2  4:5  7:8]
2×3 Array{Int64,2}:
 1  4  7
 2  5  8

julia> [[1,2]  [4,5]  [7,8]]
2×3 Array{Int64,2}:
 1  4  7
 2  5  8

julia> [1 2 3] # Numbers can also be horizontally concatenated
1×3 Array{Int64,2}:
 1  2  3
```

Using semicolons (or newlines) and spaces (or tabs) can be combined to concatenate both horizontally and vertically at the same time.

```
julia> [1 2
        3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> [zeros(Int, 2, 2) [1; 2]
        [3 4]            5]
3×3 Array{Int64,2}:
 0  0  1
 0  0  2
 3  4  5
```

More generally, concatenation can be accomplished through the `cat` function. These syntaxes are shorthands for function calls that themselves are convenience functions:

语法	函数	描述
	`cat`	沿着 `s` 的第 `k` 维拼接数组
`[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

Typed array literals

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 Array{Int64,2}:
 1  2  3  4

julia> Int8[[1 2] [3 4]]
1×4 Array{Int8,2}:
 1  2  3  4
```

20.4 Comprehensions

（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，其维数是变量范围 `rx`，`ry` 等的维数串联。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
```

The resulting array type depends on the types of the computed elements just like array literals do. In order to control the type explicitly, a type can be prepended to the comprehension. For example, we could have requested the result in single precision by writing:

```
Float32[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
```

20.5 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 Array{Tuple{Float64,Int64},2}:
 (0.5, 1)       (0.333333, 3)
 (0.333333, 2)  (0.25, 4)
```

Generators are implemented via inner functions. Just like inner functions used elsewhere in the language, variables from the enclosing scope can be "captured" in the inner function. For example, `sum(p[i] - q[i] for i=1:n)` captures the three variables `p`, `q` and `n` from the enclosing scope. Captured variables can present performance challenges; see performance tips.

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 Array{Tuple{Int64,Int64},1}:
 (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 Array{Tuple{Int64,Int64},1}:
 (2, 2)
 (3, 1)
```

20.6 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] # all scalar indices
3

julia> A[[1, 2], [1], [1, 2], [1]] # all vector indices
2×1×2×1 Array{Int64,4}:
[:, :, 1, 1] =
 1
 2

[:, :, 2, 1] =
 5
 6

julia> A[[1, 2], [1], [1, 2], 1] # a mix of index types
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 Array{Int64,2}:
 1  2
 1  2

julia> A[[1 2; 1 2], 1, 2, 1]
2×2 Array{Int64,2}:
 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 Array{Int64,2}:
 6  10
 7  11

julia> x[1, [2 3; 4 1]]
2×2 Array{Int64,2}:
  5  9
 13  1
```

20.7 Indexed Assignment

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`。

如果任一 `I_k` 选择了一个以上的位置，则等号右侧的 `X` 必须为一个与 `A[I_1, I_2, ..., I_n]` 形状一致的数组或一个具有相同元素数的向量。数组 `A` 中 `I_1[i_1], I_2[i_2], ..., I_n[i_n]` 位置的值将被 `X[I_1, I_2, ..., I_n]` 的值覆盖，必要时会转换类型。逐元素的赋值运算符 `.=` 可以用于将 `X` 沿选择的位置 broadcast：

```
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 Array{Int64,2}:
 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 Array{Int64,2}:
 -1  -4   7
 -2  -5   8
  3   6  -9
```

20.8 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}s，其行为类似于跨越多个维度的 N 维整数元组（详见下文）s, which behave like an N-tuple of integers spanning multiple dimensions (see below for more details)
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
   - 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 Array{Int64,2}:
 1   7  13
 3   9  15
 5  11  17

julia> A[4]
7

julia> A[[2, 5, 8]]
3-element Array{Int64,1}:
  3
  9
 15

julia> A[[1 4; 3 8]]
2×2 Array{Int64,2}:
 1   7
 5  15

julia> A[[]]
Int64[]

julia> A[1:2:5]
3-element Array{Int64,1}:
 1
 5
 9

julia> A[2, :]
3-element Array{Int64,1}:
  3
  9
 15

julia> A[:, 3]
3-element Array{Int64,1}:
 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
```

如果单独考虑，这可能看起来相对微不足道；`CartesianIndex` 只是将多个整数聚合成一个表示单个多维索引的对象。但是，当与其他索引形式和迭代器组合产生多个 `CartesianIndex` 时，这可以生成

非常优雅和高效的代码。请参阅下面的迭代，有关更高级的示例，请参阅关于多维算法和迭代博客文章。

也支持 `CartesianIndex {N}` 的数组。它们代表一组标量索引，每个索引都跨越 `N` 个维度，从而实现一种有时也称为逐点索引的索引形式。例如，它可以从上面的 `A` 的第一「页」访问对角元素：

```
julia> page = A[:,:,1]
4×4 Array{Int64,2}:
 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 Array{Int64,1}:
  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 Array{Int64,1}:
  1
  6
 11
 16

julia> A[CartesianIndex.(axes(A, 1), axes(A, 2)), :]
4×2 Array{Int64,2}:
  1  17
  6  22
 11  27
 16  32
```

> **Warning**
>
> `CartesianIndex` and arrays of `CartesianIndex` are not compatible with the end keyword to represent the last index of a dimension. Do not use end in indexing expressions that may contain either `CartesianIndex` or arrays thereof.

Logical indexing

Often referred to as logical indexing or indexing with a logical mask, indexing by a boolean array selects elements at the indices where its values are `true`. Indexing by a boolean vector `B` is effectively the same as indexing by the vector of integers that is returned by `findall(B)`. Similarly, indexing by a `N`-dimensional boolean array is effectively the same as indexing by the vector of `CartesianIndex{N}`s where its values are `true`. A logical index must be a vector of the same length as the dimension it indexes into, or it must be the only index provided and match the size and dimensionality of the array it indexes into. It is generally more efficient to use boolean arrays as indices directly instead of first calling `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 Array{Int64,2}:
 2  6  10  14
 3  7  11  15

julia> mask = map(ispow2, x)
4×4 Array{Bool,2}:
 1  0  0  0
 1  0  0  0
 0  0  0  0
 1  1  0  1

julia> x[mask]
5-element Array{Int64,1}:
  1
  2
  4
  8
 16
```

Number of indices

Cartesian indexing

The ordinary way to index into an `N`-dimensional array is to use exactly `N` indices; each index selects the position(s) in its particular dimension. For example, in the three-dimensional array `A = rand(4, 3, 2)`, `A[2, 3, 1]` will select the number in the second row of the third column in the first "page" of the array. This is often referred to as *cartesian indexing*.

Linear indexing

When exactly one index `i` is provided, that index no longer represents a location in a particular dimension of the array. Instead, it selects the `i`th element using the column-major iteration order that linearly spans the entire array. This is known as *linear indexing*. It essentially treats the array as though it had been reshaped into a one-dimensional vector with vec.

```
julia> A = [2 6; 4 7; 3 1]
3×2 Array{Int64,2}:
 2  6
 4  7
 3  1

julia> A[5]
7

julia> vec(A)[5]
7
```

A linear index into the array `A` can be converted to a `CartesianIndex` for cartesian indexing with `CartesianIndices(A)[i]` (see `CartesianIndices`), and a set of `N` cartesian indices can be converted to a linear index with `LinearIndices(A)[i_1, i_2, ..., i_N]` (see `LinearIndices`).

```
julia> CartesianIndices(A)[5]
CartesianIndex(2, 2)

julia> LinearIndices(A)[2, 2]
5
```

It's important to note that there's a very large assymmetry in the performance of these conversions. Converting a linear index to a set of cartesian indices requires dividing and taking the remainder, whereas going the other way is just multiplies and adds. In modern processors, integer division can be 10-50 times slower than multiplication. While some arrays —like `Array` itself —are implemented using a linear chunk of memory and directly use a linear index in their implementations, other arrays —like `Diagonal` —need the full set of cartesian indices to do their lookup (see `IndexStyle` to introspect which is which). As such, when iterating over an entire array, it's much better to iterate over `eachindex(A)` instead of `1:length(A)`. Not only will the former be much faster in cases where `A` is `IndexCartesian`, but it will also support OffsetArrays, too.

Omitted and extra indices

In addition to linear indexing, an `N`-dimensional array may be indexed with fewer or more than `N` indices in certain situations.

Indices may be omitted if the trailing dimensions that are not indexed into are all length one. In other words, trailing indices can be omitted only if there is only one possible value that those omitted indices could be for an in-bounds indexing expression. For example, a four-dimensional array with size `(3, 4, 2, 1)` may be indexed with only three indices as the dimension that gets skipped (the fourth dimension) has length one. Note that linear indexing takes precedence over this rule.

```
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
```

When omitting *all* indices with `A[]`, this semantic provides a simple idiom to retrieve the only element in an array and simultaneously ensure that there was only one element.

Similarly, more than N indices may be provided if all the indices beyond the dimensionality of the array are 1 (or more generally are the first and only element of axes(A, d) where d is that particular dimension number). This allows vectors to be indexed like one-column matrices, for example:

```
julia> A = [8,6,7]
3-element Array{Int64,1}:
 8
 6
 7

julia> A[2,1]
6
```

20.9 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)
```

与 for i = 1:length(A) 相比，eachindex 提供了一种迭代任何数组类型的有效方法。

20.10 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()。

20.11 Array and Vectorized Operators and Functions

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。

20.12 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
```

Dotted operators such as `.+` and `.*` are equivalent to `broadcast` calls (except that they fuse, as described above). There is also a `broadcast!` function to specify an explicit destination (which can also be accessed in a fusing fashion by `.=` assignment). In fact, `f.(args...)` is equivalent to `broadcast(f, args...)`, providing a convenient syntax to broadcast any function (dot syntax). Nested "dot calls" `f.(...)` (including calls to `.+` etcetera) automatically fuse into a single `broadcast` call.

Additionally, `broadcast` is not limited to arrays (see the function documentation); it also handles scalars, tuples and other collections. By default, only some argument types are considered scalars, including (but not limited to) `Numbers`, `Strings`, `Symbols`, `Types`, `Functions` and some common singletons like `missing` and `nothing`. All other arguments are iterated over or indexed into elementwise.

```
julia> convert.(Float32, [1, 2])
2-element Array{Float32,1}:
 1.0
 2.0

julia> ceil.(UInt8, [1.2 3.4; 5.6 6.7])
2×2 Array{UInt8,2}:
 0x02  0x04
 0x06  0x07

julia> string.(1:3, ". ", ["First", "Second", "Third"])
3-element Array{String,1}:
 "1. First"
 "2. Second"
 "3. Third"
```

Sometimes, you want a container (like an array) that would normally participate in broadcast to be "protected" from broadcast's behavior of iterating over all of its elements. By placing it inside another container (like a single element `Tuple`) broadcast will treat it as a single value.

```
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])
```

20.13 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` 类型包含任何模糊类似的东西，它的实现可能与传统数组完全不同。例如，可以根据请求而不是存储来计算元素。但是，任何具体的 `AbstractArray{T,N}` 类型通常应该至少实现 `size(A)`（返回 `Int` 元组），`getindex(A,i)` 和 `getindex(A,i1,...,iN)`；可变数组也应该实现 `setindex!`。建议这些操作具有几乎为常数的时间复杂性，或严格说来 Õ(1) 复杂性，否则某些数组函数可能出乎意料的慢。具体类型通常还应提供 `similar(A,T=eltype(A),dims=size(A))` 方法，用于为 `copy` 分配类似的数组和其他位于当前数组空间外的操作。无论在内部如何表示 `AbstractArray{T,N}`，`T` 是由 整数索引返回的对象类型（`A[1, ..., 1]`，当 `A` 不为空），`N` 应该是 `size` 返回的元组的长度。有关定义自定义 `AbstractArray` 实现的更多详细信息，请参阅接口章节中的数组接口导则。

DenseArray is an abstract subtype of AbstractArray intended to include all arrays where elements are stored contiguously in column-major order (see additional notes in Performance Tips). The Array type is a specific instance of DenseArray; Vector and Matrix are aliases for the 1-d and 2-d cases. Very few operations are implemented specifically for Array beyond those that are required for all AbstractArrays; much of the array library is implemented in a generic manner that allows all custom arrays to behave similarly.

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。

An array is ”strided” if it is stored in memory with well-defined spacings (strides) between its elements. A strided array with a supported element type may be passed to an external (non-Julia) library like BLAS or LAPACK by simply passing its pointer and the stride for each dimension. The stride(A, d) is the distance between elements along dimension d. For example, the builtin Array returned by rand(5,7,2) has its elements arranged contiguously in column major order. This means that the stride of the first dimension —the spacing between elements in the same column —is 1:

```
julia> A = rand(5,7,2);

julia> stride(A,1)
1
```

The stride of the second dimension is the spacing between elements in the same row, skipping as many elements as there are in a single column (5). Similarly, jumping between the two ”pages” (in the third dimension) requires skipping 5*7 == 35 elements. The strides of this array is the tuple of these three numbers together:

```
julia> strides(A)
(1, 5, 35)
```

In this particular case, the number of elements skipped *in memory* matches the number of *linear indices* skipped. This is only the case for contiguous arrays like Array (and other DenseArray subtypes) and is not true in general. Views with range indices are a good example of *non-contiguous* strided arrays; consider V = @view A[1:3:4, 2:2:6, 2:-1:1]. This view V refers to the same memory as A but is skipping and rearranging some of its elements. The stride of the first dimension of V is 3 because we’re only selecting every third row from our original array:

```
julia> V = @view A[1:3:4, 2:2:6, 2:-1:1];

julia> stride(V, 1)
3
```

This view is similarly selecting every other column from our original A —and thus it needs to skip the equivalent of two five-element columns when moving between indices in the second dimension:

```
julia> stride(V, 2)
10
```

The third dimension is interesting because its order is reversed! Thus to get from the first ”page” to the second one it must go *backwards* in memory, and so its stride in this dimension is negative!

```
julia> stride(V, 3)
-35
```

This means that the `pointer` for `V` is actually pointing into the middle of `A`’s memory block, and it refers to elements both backwards and forwards in memory. See the interface guide for strided arrays for more details on defining your own strided arrays. `StridedVector` and `StridedMatrix` are convenient aliases for many of the builtin array types that are considered strided arrays, allowing them to dispatch to select specialized implementations that call highly tuned and optimized BLAS and LAPACK functions using just the pointer and strides.

It is worth emphasizing that strides are about offsets in memory rather than indexing. If you are looking to convert between linear (single-index) indexing and cartesian (multi-index) indexing, see `LinearIndices` and `CartesianIndices`.

Chapter 21

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。

21.1 缺失值的传播

`missing` values *propagate* automatically when passed to standard mathematical operators and functions. For these functions, uncertainty about the value of one of the operands induces uncertainty about the result. In practice, this means a math operation involving a `missing` value generally returns `missing`

```
julia> missing + 1
missing

julia> "a" * missing
missing

julia> abs(missing)
missing
```

As `missing` is a normal Julia object, this propagation rule only works for functions which have opted in to implement this behavior. This can be achieved either via a specific method defined for arguments of type `Missing`, or simply by accepting arguments of this type, and passing them to functions which propagate them (like standard math operators). Packages should consider whether it makes sense to propagate missing values when defining new functions, and define methods appropriately if that is the case. Passing a `missing` value to a function for which no method accepting arguments of type `Missing` is defined throws a `MethodError`, just like for any other type.

Functions that do not propagate `missing` values can be made to do so by wrapping them in the `passmissing` function provided by the Missings.jl package. For example, `f(x)` becomes `passmissing(f)(x)`.

21.2 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
```

21.3 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，这些规则也由 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。

21.4 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
```

另一方面，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
```

21.5 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 Array{Union{Missing, Int64},1}:
 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。这简单地反映了以下事实：数组条目可以具有类型 T（在这是 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} 组合一个 Array{UInt8}，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 Array{Union{Missing, String},2}:
 missing  missing  missing
 missing  missing  missing
```

允许但不包含 missing 值的数组可使用 convert 转换回不允许缺失值的数组。如果该数组包含 missing 值，在类型转换时会抛出 MethodError

```
julia> x = Union{Missing, String}["a", "b"]
2-element Array{Union{Missing, String},1}:
 "a"
 "b"

julia> convert(Array{String}, x)
2-element Array{String,1}:
 "a"
 "b"

julia> y = Union{Missing, String}[missing, "b"]
2-element Array{Union{Missing, String},1}:
 missing
 "b"

julia> convert(Array{String}, y)
ERROR: MethodError: Cannot `convert` an object of type Missing to an object of type String
```

21.6 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
```

This convenience function returns an iterator which filters out missing values efficiently. It can therefore be used with any function which supports iterators

```
julia> x = skipmissing([3, missing, 2, 1])
skipmissing(Union{Missing, Int64}[3, missing, 2, 1])
```

```
julia> maximum(x)
3

julia> mean(x)
2.0

julia> mapreduce(sqrt, +, x)
4.146264369941973
```

Objects created by calling `skipmissing` on an array can be indexed using indices from the parent array. Indices corresponding to missing values are not valid for these objects and an error is thrown when trying to use them (they are also skipped by `keys` and `eachindex`)

```
julia> x[1]
3

julia> x[2]
ERROR: MissingException: the value at index (2,) is missing
[...]
```

This allows functions which operate on indices to work in combination with `skipmissing`. This is notably the case for search and find functions, which return indices valid for the object returned by `skipmissing` which are also the indices of the matching entries *in the parent array*

```
julia> findall(==(1), x)
1-element Array{Int64,1}:
 4

julia> findfirst(!iszero, x)
1

julia> argmax(x)
1
```

Use `collect` to extract non-`missing` values and store them in an array

```
julia> collect(x)
3-element Array{Int64,1}:
 3
 2
 1
```

21.7 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 22

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。

22.1 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
```

22.2 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。

22.3 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。

22.4 使用文件

Like many other environments, Julia 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."
```

22.5 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> @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)
```

Note that the return type of the last invocation is different. This is because this server does not listen on TCP, but rather on a named pipe (Windows) or UNIX domain socket. Also note that Windows named pipe format has to be a specific pattern such that the name prefix (`\\.\pipe\`) uniquely identifies the file type. The difference between TCP and named pipes or UNIX domain sockets is subtle and has to do with the `accept` and `connect` methods. The `accept` method retrieves a connection to the client that is connecting on the server we just created, while the `connect` function connects to a server using the specified method. The `connect` function takes the same arguments as `listen`, so, assuming the environment (i.e. host, cwd, etc.) is the same you should be able to pass the same arguments to `connect` as you did to listen to establish the connection. So let's try that out (after having created the server above):

```
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> @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> @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)
```

22.6 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"
```

Chapter 23

并行计算

Julia supports three main categories of features for concurrent and parallel programming:

1. Asynchronous "tasks", or coroutines
2. Multi-threading
3. Distributed computing

Julia Tasks allow suspending and resuming computations for I/O, event handling, producer-consumer processes, and similar patterns. Tasks can synchronize through operations like `wait` and `fetch`, and communicate via `Channel`s.

Multi-threading functionality builds on tasks by allowing them to run simultaneously on more than one thread or CPU core, sharing memory.

Finally, distributed computing runs multiple processes with separate memory spaces, potentially on different machines. This functionality is provided by the `Distributed` standard library as well as external packages like `MPI.jl` and `DistributedArrays.jl`.

Chapter 24

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。

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，会被发送到 `stdout` 中去。`run` 方法本身返回 `nothing`，如果外部命令未能成功运行，则抛出 `ErrorException`。

如果要读取外部命令的输出，可以使用 `read`：

```
julia> a = read(`echo hello`, String)
"hello\n"

julia> chomp(a) == "hello"
true
```

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 Array{String,1}:
 "echo"
 "foo bar"

julia> `echo "foo bar"`[2]
"foo bar"
```

24.1 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 Array{String,1}:
 "/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 Array{String,1}:
 "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 Array{String,1}:
 "foo"
 "bar"
 "baz"

julia> exts = ["aux","log"]
2-element Array{String,1}:
 "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`
```

24.2 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。

24.3 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)
```

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。

Chapter 25

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. 在 Julia 中，上述特性可以仅仅通过调用 `ccall` 实现，它的语法看起来就像是普通的函数调用。

The code to be called must be available as a shared library. Most C and Fortran libraries ship compiled as shared libraries already, but if you are compiling the code yourself using GCC (or Clang), you will need to use the `-shared` and `-fPIC` options. The machine instructions generated by Julia's JIT are the same as a native C call would be, so the resulting overhead is the same as calling a library function from C code. [1]

Shared libraries and functions are referenced by a tuple of the form `(:function, "library")` or `("function", "library")` where `function` is the C-exported function name, and `library` refers to the shared library name. Shared libraries available in the (platform-specific) load path will be resolved by name. The full path to the library may also be specified.

可以单独使用函数名来代替元组（只用 `:function` 或 `"function"`）。在这种情况下，函数名在当前进程中进行解析。这一调用形式可用于调用 C 库函数、Julia 运行时中的函数或链接到 Julia 的应用程序中的函数。

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），要通过 `ccall` 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。

Finally, you can use `ccall` to actually generate a call to the library function. The arguments to `ccall` are:

1. A `(:function, "library")` pair (most common),

   或

   a `:function` name symbol or `"function"` name string (for symbols in the current process or libc),

   或

   一个函数指针（例如，从 `dlsym` 获得的指针）。

2. The function's return type

3. A tuple of input types, corresponding to the function signature

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 below).

The remaining parameters are evaluated at compile time, when the containing method is defined.

Note

See below for how to map C types to Julia types.

As a complete but simple example, the following calls the `clock` function from the standard C library on most Unix-derived systems:

```
julia> t = ccall(:clock, Int32, ())
2292761

julia> t
2292761

julia> typeof(ans)
Int32
```

`clock` takes no arguments and returns an `Int32`. One common mistake is forgetting that a 1-tuple of argument types must be written with a trailing comma. For example, to call the `getenv` function to get a pointer to the value of an environment variable, one makes a call like this:

```
julia> path = ccall(:getenv, Cstring, (Cstring,), "SHELL")
Cstring(@0x00007fff5fbffc45)

julia> unsafe_string(path)
"/bin/bash"
```

Note that the argument type tuple must be written as `(Cstring,)`, not `(Cstring)`. This is because `(Cstring)` is just the expression `Cstring` surrounded by parentheses, rather than a 1-tuple containing `Cstring`:

```
julia> (Cstring)
Cstring

julia> (Cstring,)
(Cstring,)
```

In practice, especially when providing reusable functionality, one generally wraps `ccall` uses in Julia functions that set up arguments and then check for errors in whatever manner the C or Fortran function specifies. And if an error occurs it is thrown as a normal Julia exception. This is especially important since C and Fortran APIs are notoriously inconsistent about how they indicate error conditions. For example, the `getenv` C library function is wrapped in the following Julia function, which is a simplified version of the actual definition from `env.jl`:

```
function getenv(var::AbstractString)
    val = ccall(:getenv, Cstring, (Cstring,), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    return unsafe_string(val)
end
```

C 函数 getenv 通过返回 NULL 的方式进行报错，但是其他 C 标准库函数也会通过多种不同的方式来报错，这包括返回 -1，0，1 以及其它特殊值。此封装能够明确地抛出异常信息，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")
getenv: undefined variable: FOOBAR
```

Here is a slightly more complex example that discovers the local machine's hostname. In this example, the networking library code is assumed to be in a shared library named "libc". In practice, this function is usually part of the C standard library, and so the "libc" portion should be omitted, but we wish to show here the usage of this syntax.

```
function gethostname()
    hostname = Vector{UInt8}(undef, 256) # MAXHOSTNAMELEN
    err = ccall((:gethostname, "libc"), Int32,
                (Ptr{UInt8}, Csize_t),
                hostname, sizeof(hostname))
    Base.systemerror("gethostname", err != 0)
    hostname[end] = 0 # ensure null-termination
    return unsafe_string(pointer(hostname))
end
```

This example first allocates an array of bytes. It then calls the C library function gethostname to populate the array with the hostname. Finally, it takes a pointer to the hostname buffer, and converts the pointer to a Julia string, assuming that it is a NUL-terminated C string. It is common for C libraries to use this pattern of requiring the caller to allocate memory to be passed to the callee and populated. Allocation of memory from Julia like this is generally accomplished by creating an uninitialized array and passing a pointer to its data to the C function. This is why we don't use the Cstring type here: as the array is uninitialized, it could contain NUL bytes. Converting to a Cstring as part of the ccall checks for contained NUL bytes and could therefore throw a conversion error.

25.1 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, ...)
```

The macro @cfunction generates the C-compatible function pointer for a call to a Julia function. The arguments to @cfunction are:

1. A Julia function
2. The function's return type
3. A tuple of input types, corresponding to the function signature

Note

As with ccall, the return type and tuple of input types must be literal constants.

Note

Currently, only the platform-default C calling convention is supported. This means that `@cfunction`-generated pointers cannot be used in calls where WINAPI expects a `stdcall` function on 32-bit Windows, but can be used on WIN64 (where `stdcall` is unified with the C calling convention).

一个典型的例子就是标准 C 库函数 `qsort`，定义为：

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compare)(const void*, const void*));
```

The base argument is a pointer to an array of length `nmemb`, with elements of `size` bytes each. `compare` is a callback function which takes pointers to two elements a and b and returns an integer less/greater than zero if a should appear before/after b (or zero if any order is permitted).

Now, suppose that we have a 1-d array A of values in Julia that we want to sort using the `qsort` function (rather than Julia's built-in `sort` function). Before we consider calling `qsort` and passing arguments, we need to write a comparison function:

```
julia> function mycompare(a, b)::Cint
           return (a < b) ? -1 : ((a > b) ? +1 : 0)
       end
mycompare (generic function with 1 method)
```

`qsort` expects a comparison function that return a C `int`, so we annotate the return type to be `Cint`.

In order to pass this function to C, we obtain its address using the macro `@cfunction`:

```
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值元组, 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]
4-element Array{Float64,1}:
  1.3
 -2.7
  4.4
  3.1

julia> ccall(:qsort, Cvoid, (Ptr{Cdouble}, Csize_t, Csize_t, Ptr{Cvoid}),
             A, length(A), sizeof(eltype(A)), mycompare_c)

julia> A
4-element Array{Float64,1}:
 -2.7
  1.3
  3.1
  4.4
```

As the example shows, the original Julia array A has now been sorted: `[-2.7, 1.3, 3.1, 4.4]`. Note that Julia takes care of converting the array to a `Ptr{Cdouble}`), computing the size of the element type in bytes, and so on.

For fun, try inserting a `println("mycompare($a, $b)")` line into `mycompare`, which will allow you to see the comparisons that `qsort` is performing (and to verify that it is really calling the Julia function that you passed to it).

25.2 Mapping C Types to Julia

It is critical to exactly match the declared C type with its declaration in Julia. Inconsistencies can cause code that works correctly on one system to fail or produce indeterminate results on a different system.

Note that no C header files are used anywhere in the process of calling C functions: you are responsible for making sure that your Julia types and call signatures accurately reflect those in the C header file.[2]

Automatic Type Conversion

Julia automatically inserts calls to the `Base.cconvert` function to convert each argument to the specified type. For example, the following call:

```
ccall((:foo, "libfoo"), Cvoid, (Int32, Float64), x, y)
```

will behave as if it were written like this:

```
ccall((:foo, "libfoo"), Cvoid, (Int32, Float64),
      Base.unsafe_convert(Int32, Base.cconvert(Int32, x)),
      Base.unsafe_convert(Float64, Base.cconvert(Float64, y)))
```

`Base.cconvert` normally just calls `convert`, but can be defined to return an arbitrary new object more appropriate for passing to C. This should be used to perform all allocations of memory that will be accessed by the C code. For example, this is used to convert an `Array` of objects (e.g. strings) to an array of pointers.

`Base.unsafe_convert` handles conversion to `Ptr` types. It is considered unsafe because converting an object to a native pointer can hide the object from the garbage collector, causing it to be freed prematurely.

Type Correspondences

First, let's review some relevant Julia type terminology:

Bits Types

There are several special types to be aware of, as no other type can be defined to behave the same:

- `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` ）

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。It is stored and defined by-value.
`struct`	`Pair{Int, Int}`	"Struct" :: A type with all fields defined to be constant. It is defined by-value, and may be stored with a type-tag.
	`ComplexF64` (`isbits`)	"Is-Bits" :: A `primitive type`, or a `struct` type where all fields are other `isbits` types. It is defined by-value, and is stored without a type-tag.
`struct ...; end`	`nothing`	`Singleton`：没有成员变量的 `Leaf Type` 或 `Struct`。
`(...)` or `tuple(...)`	`(1, 2, 3)`	"Tuple" :: an immutable data-structure similar to an anonymous struct type, or a constant array. Represented as either an array or a struct.

- `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。

- `Ref{T}`

  和 `Ptr{T}` 行为相同，能通过 Julia 的 GC 管理其内存。

- `Array{T,N}`

  When an array is passed to C as a `Ptr{T}` argument, it is not reinterpret-cast: Julia requires that the element type of the array matches T, and the address of the first element is passed.

  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。

On all systems we currently support, basic C/C++ value types may be translated to Julia types as follows. Every C type also has a corresponding Julia type with the same name, prefixed by C. This can help when writing portable code (and remembering that an `int` in C is not the same as an `Int` in Julia).

System Independent Types

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}`
`T*` (where T represents an appropriately defined type)			`Ref{T}`
`char*` (or `char[]`, e.g. a string)	`CHARACTER*N`		`Cstring` if NUL-terminated, or `Ptr{UInt8}` if not
`char**` (or `*char[]`)			`Ptr{Ptr{UInt8}}`
`jl_value_t*` (any Julia Type)			`Any`
`jl_value_t**` (a reference to a Julia Type)			`Ref{Any}`
`va_arg`			Not supported
`...` (variadic function specification)			`T...` (where `T` is one of the above types, variadic functions of different argument types are not supported)

The `Cstring` type is essentially a synonym for `Ptr{UInt8}`, except the conversion to `Cstring` throws an error if the Julia string contains any embedded NUL characters (which would cause the string to be silently truncated if the C routine treats NUL as the terminator). If you are passing a `char*` to a C routine that does not assume NUL termination (e.g. because you pass an explicit string length), or if you know for certain that your Julia string does not contain NUL and want to skip the check, you can use `Ptr{UInt8}` as the argument type. `Cstring` can also be used as the `ccall` return type, but in that case it obviously does not introduce any extra checks and is only meant to improve readability of the call.

System Dependent Types

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)

Note

When calling Fortran, all inputs must be passed by pointers to heap- or stack-allocated values, so all type correspondences above should contain an additional `Ptr{..}` or `Ref{..}` wrapper around their type specification.

Warning

For string arguments (`char*`) the Julia type should be `Cstring` (if NUL- terminated data is expected), or either `Ptr{Cchar}` or `Ptr{UInt8}` otherwise (these two pointer types have the same effect), as described above, not `String`. Similarly, for array arguments (`T[]` or `T*`), the Julia type should again be `Ptr{T}`, not `Vector{T}`.

Warning

Julia's `Char` type is 32 bits, which is not the same as the wide character type (`wchar_t` or `wint_t`) on all platforms.

Warning

A return type of `Union{}` means the function will not return, i.e., C++11 `[[noreturn]]` or C11 `_Noreturn` (e.g. `jl_throw` or `longjmp`). Do not use this for functions that return no value (`void`) but do return, use `Cvoid` instead.

Note

For `wchar_t*` arguments, the Julia type should be `Cwstring` (if the C routine expects a NUL-terminated string), or `Ptr{Cwchar_t}` otherwise. Note also that UTF-8 string data in Julia is internally NUL-terminated, so it can be passed to C functions expecting NUL-terminated data without making a copy (but using the `Cwstring` type will cause an error to be thrown if the string itself contains NUL characters).

Note

C functions that take an argument of type `char**` can be called by using a `Ptr{Ptr{UInt8}}` type within Julia. For example, C functions of the form:

```
int main(int argc, char **argv);
```

can be called via the following Julia code:

```
argv = [ "a.out", "arg1", "arg2" ]
ccall(:main, Int32, (Int32, Ptr{Ptr{UInt8}}), length(argv), argv)
```

Note

For Fortran functions taking variable length strings of type `character(len=*)` the string lengths are provided as *hidden arguments*. Type and position of these arguments in the list are compiler specific, where compiler vendors usually default to using `Csize_t` as type and append the hidden arguments at the end of the argument list. While this behaviour is fixed for some compilers (GNU), others *optionally* permit placing hidden arguments directly after the character argument (Intel, PGI). For example, Fortran subroutines of the form

```
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 compilers *may* also add other hidden arguments for pointers, assumed-shape (`:`) and assumed-size (*) arrays. Such behaviour can be avoided by using `ISO_C_BINDING` and including `bind(c)` in the definition of the subroutine, which is strongly recommended for interoperable code. In this case there will be no hidden arguments, at the cost of some language features (e.g. only `character(len=1)` will be permitted to pass strings).

Note

A C function declared to return `Cvoid` will return the value `nothing` in Julia.

Struct Type Correspondences

Composite types such as `struct` in C or TYPE in Fortran90 (or STRUCTURE / RECORD in some variants of F77), can be mirrored in Julia by creating a `struct` definition with the same field layout.

When used recursively, `isbits` types are stored inline. All other types are stored as a pointer to the data. When mirroring a struct used by-value inside another struct in C, it is imperative that you do not attempt to manually copy the fields over, as this will not preserve the correct field alignment. Instead, declare an `isbits` struct type and use that instead. Unnamed structs are not possible in the translation to Julia.

Packed structs and union declarations are not supported by Julia.

You can get an approximation of a `union` if you know, a priori, the field that will have the greatest size (potentially including padding). When translating your fields to Julia, declare the Julia field to be only of that type.

Arrays of parameters can be expressed with `NTuple`. For example, the struct in C notation written as

```
struct B {
    int A[3];
};

b_a_2 = B.A[2];
```

can be written in Julia as

```
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)
```

Arrays of unknown size (C99-compliant variable length structs specified by `[]` or `[0]`) are not directly supported. Often the best way to deal with these is to deal with the byte offsets directly. For example, if a C library declared a proper string type and returned a pointer to it:

```
struct String {
    int strlen;
    char data[];
};
```

In Julia, we can access the parts independently to make a copy of that string:

```
str = from_c::Ptr{Cvoid}
len = unsafe_load(Ptr{Cint}(str))
unsafe_string(str + Core.sizeof(Cint), len)
```

Type Parameters

The type arguments to `ccall` and `@cfunction` are evaluated statically, when the method containing the usage is defined. They therefore must take the form of a literal tuple, not a variable, and cannot reference local variables.

This may sound like a strange restriction, but remember that since C is not a dynamic language like Julia, its functions can only accept argument types with a statically-known, fixed signature.

However, while the type layout must be known statically to compute the intended C ABI, the static parameters of the function are considered to be part of this static environment. The static parameters of the function may be used as type parameters in the call signature, as long as they don't affect the layout of the type. For example, `f(x::T) where {T} = ccall(:valid, Ptr{T}, (Ptr{T},), x)` is valid, since `Ptr` is always a word-size primitive type. But, `g(x::T) where {T} = ccall(:notvalid, T, (T,), x)` is not valid, since the type layout of T is not known statically.

SIMD 值

Note: This feature is currently implemented on 64-bit x86 and AArch64 platforms only.

If a C/C++ routine has an argument or return value that is a native SIMD type, the corresponding Julia type is a homogeneous tuple of `VecElement` that naturally maps to the SIMD type. Specifically:

- The tuple must be the same size as the SIMD type. For example, a tuple representing an `__m128` on x86 must have a size of 16 bytes.
- The element type of the tuple must be an instance of `VecElement{T}` where T is a primitive type that is 1, 2, 4 or 8 bytes.

For instance, consider this C routine that uses AVX intrinsics:

```
#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)));
}
```

The following Julia code calls `dist` using `ccall`:

```
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((:dist, "libdist"), m256, (m256, m256), a, b)
end

println(call_dist(a,b))
```

The host machine must have the requisite SIMD registers. For example, the code above will not work on hosts without AVX support.

内存所有权

malloc/free

Memory allocation and deallocation of such objects must be handled by calls to the appropriate cleanup routines in the libraries being used, just like in any C program. Do not try to free an object received from a C library with `Libc.free` in Julia, as this may result in the `free` function being called via the wrong library and cause the process to abort. The reverse (passing an object allocated in Julia to be freed by an external library) is equally invalid.

何时使用 T、Ptr{T} 以及 Ref{T}

In Julia code wrapping calls to external C routines, ordinary (non-pointer) data should be declared to be of type T inside the `ccall`, as they are passed by value. For C code accepting pointers, `Ref{T}` should generally be used for the types of input arguments, allowing the use of pointers to memory managed by either Julia or C through the implicit call to `Base.cconvert`. In contrast, pointers returned by the C function called should be declared to be of output type `Ptr{T}`, reflecting that the memory pointed to is managed by C only. Pointers contained in C structs should be represented as fields of type `Ptr{T}` within the corresponding Julia struct types designed to mimic the internal structure of corresponding C structs.

In Julia code wrapping calls to external Fortran routines, all input arguments should be declared as of type `Ref{T}`, as Fortran passes all variables by pointers to memory locations. The return type should either be `Cvoid` for Fortran subroutines, or a T for Fortran functions returning the type T.

25.3 Mapping C Functions to Julia

`ccall` / `@cfunction` argument translation guide

For translating a C argument list to Julia:

- T, where T is one of the primitive types: `char`, `int`, `long`, `short`, `float`, `double`, `complex`, `enum` or any of their `typedef` equivalents
  - T, where T is an equivalent Julia Bits Type (per the table above)

  - if T is an `enum`, the argument type should be equivalent to `Cint` or `Cuint`
  - argument value will be copied (passed by value)

- `struct T` (including typedef to a struct)
  - T, where T is a Julia leaf type
  - argument value will be copied (passed by value)

- `void*`
  - depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
  - this argument may be declared as `Ptr{Cvoid}`, if it really is just an unknown pointer

- `jl_value_t*`
  - `Any`
  - argument value must be a valid Julia object

- `jl_value_t**`
  - `Ref{Any}`
  - argument value must be a valid Julia object (or `C_NULL`)

- `T*`
  - `Ref{T}`, where T is the Julia type corresponding to T
  - argument value will be copied if it is an `isbits` type otherwise, the value must be a valid Julia object

- `T (*)(...)` (e.g. a pointer to a function)
  - `Ptr{Cvoid}` (you may need to use `@cfunction` explicitly to create this pointer)

- `...` (e.g. a vararg)
  - `T...`, where T is the Julia type
  - currently unsupported by `@cfunction`

- `va_arg`
  - not supported by `ccall` or `@cfunction`

`ccall` / `@cfunction` return type translation guide

For translating a C return type to Julia:

- `void`
  - `Cvoid` (this will return the singleton instance `nothing::Cvoid`)
- T, where T is one of the primitive types: `char`, `int`, `long`, `short`, `float`, `double`, `complex`, `enum` or any of their `typedef` equivalents
  - T, where T is an equivalent Julia Bits Type (per the table above)
  - if T is an `enum`, the argument type should be equivalent to `Cint` or `Cuint`
  - argument value will be copied (returned by-value)
- `struct T` (including typedef to a struct)
  - T, where T is a Julia Leaf Type
  - argument value will be copied (returned by-value)
- `void*`
  - depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
  - this argument may be declared as `Ptr{Cvoid}`, if it really is just an unknown pointer
- `jl_value_t*`
  - `Any`
  - argument value must be a valid Julia object
- `jl_value_t**`
  - `Ptr{Any}` (`Ref{Any}` is invalid as a return type)
  - argument value must be a valid Julia object (or `C_NULL`)
- `T*`
  - If the memory is already owned by Julia, or is an `isbits` type, and is known to be non-null:
    - `Ref{T}`, where T is the Julia type corresponding to T
    - a return type of `Ref{Any}` is invalid, it should either be `Any` (corresponding to `jl_value_t*`) or `Ptr{Any}` (corresponding to `jl_value_t**`)
    - C **MUST NOT** modify the memory returned via `Ref{T}` if T is an `isbits` type
  - If the memory is owned by C:
    - `Ptr{T}`, where T is the Julia type corresponding to T
- `T (*)(...)` (e.g. a pointer to a function)
  - `Ptr{Cvoid}` (you may need to use `@cfunction` explicitly to create this pointer)

Passing Pointers for Modifying Inputs

Because C doesn't support multiple return values, often C functions will take pointers to data that the function will modify. To accomplish this within a `ccall`, you need to first encapsulate the value inside a `Ref{T}` of the appropriate type. When you pass this `Ref` object as an argument, Julia will automatically pass a C pointer to the encapsulated data:

```
width = Ref{Cint}(0)
range = Ref{Cfloat}(0)
ccall(:foo, Cvoid, (Ref{Cint}, Ref{Cfloat}), width, range)
```

Upon return, the contents of `width` and `range` can be retrieved (if they were changed by `foo`) by `width[]` and `range[]`; that is, they act like zero-dimensional arrays.

25.4 C Wrapper Examples

Let's start with a simple example of a C wrapper that returns a `Ptr` type:

```
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(
        (:gsl_permutation_alloc, :libgsl), # name of C function and library
        Ptr{gsl_permutation},              # output type
        (Csize_t,),                        # tuple of input types
        n                                  # name of Julia variable to pass in
    )
    if output_ptr == C_NULL # Could not allocate memory
        throw(OutOfMemoryError())
    end
    return output_ptr
end
```

The GNU Scientific Library (here assumed to be accessible through `:libgsl`) defines an opaque pointer, `gsl_permutation *`, as the return type of the C function `gsl_permutation_alloc`. As user code never has to look inside the `gsl_permutation` struct, the corresponding Julia wrapper simply needs a new type declaration, `gsl_permutation`, that has no internal fields and whose sole purpose is to be placed in the type parameter of a `Ptr` type. The return type of the `ccall` is declared as `Ptr{gsl_permutation}`, since the memory allocated and pointed to by `output_ptr` is controlled by C.

The input `n` is passed by value, and so the function's input signature is simply declared as `(Csize_t,)` without any `Ref` or `Ptr` necessary. (If the wrapper was calling a Fortran function instead, the corresponding function input signature would instead be `(Ref{Csize_t},)`, since Fortran variables are passed by pointers.) Furthermore, `n` can be any type that is convertible to a `Csize_t` integer; the `ccall` implicitly calls `Base.cconvert(Csize_t, n)`.

Here is a second example wrapping the corresponding destructor:

```
# The corresponding C signature is
#     void gsl_permutation_free (gsl_permutation * p);
```

```
function permutation_free(p::Ref{gsl_permutation})
    ccall(
        (:gsl_permutation_free, :libgsl), # name of C function and library
        Cvoid,                            # output type
        (Ref{gsl_permutation},),          # tuple of input types
        p                                 # name of Julia variable to pass in
    )
end
```

Here, the input p is declared to be of type `Ref{gsl_permutation}`, meaning that the memory that p points to may be managed by Julia or by C. A pointer to memory allocated by C should be of type `Ptr{gsl_permutation}`, but it is convertible using `Base.cconvert` and therefore

Now if you look closely enough at this example, you may notice that it is incorrect, given our explanation above of preferred declaration types. Do you see it? The function we are calling is going to free the memory. This type of operation cannot be given a Julia object (it will crash or cause memory corruption). Therefore, it may be preferable to declare the p type as `Ptr{gsl_permutation }`, to make it harder for the user to mistakenly pass another sort of object there than one obtained via `gsl_permutation_alloc`.

If the C wrapper never expects the user to pass pointers to memory managed by Julia, then using `p::Ptr{gsl_permutation}` for the method signature of the wrapper and similarly in the `ccall` is also acceptable.

Here is a third example passing Julia arrays:

```
# 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(
        (:gsl_sf_bessel_Jn_array, :libgsl), # name of C function and library
        Cint,                               # output type
        (Cint, Cint, Cdouble, Ref{Cdouble}),# tuple of input types
        nmin, nmax, x, result_array         # names of Julia variables to pass in
    )
    if errorcode != 0
        error("GSL error code $errorcode")
    end
    return result_array
end
```

The C function wrapped returns an integer error code; the results of the actual evaluation of the Bessel J function populate the Julia array `result_array`. This variable is declared as a `Ref{Cdouble}`, since its memory is allocated and managed by Julia. The implicit call to `Base.cconvert(Ref{Cdouble}, result_array)` unpacks the Julia pointer to a Julia array data structure into a form understandable by C.

25.5 Fortran Wrapper Example

The following example utilizes ccall to call a function in a common Fortran library (libBLAS) to computes a dot product. Notice that the argument mapping is a bit different here than above, as we need to map from Julia to Fortran. On every argument type, we specify `Ref` or `Ptr`. This mangling convention may be specific to your

fortran compiler and operating system, and is likely undocumented. However, wrapping each in a `Ref` (or `Ptr`, where equivalent) is a frequent requirement of Fortran compiler implementations:

```
function compute_dot(DX::Vector{Float64}, DY::Vector{Float64})
    @assert length(DX) == length(DY)
    n = length(DX)
    incx = incy = 1
    product = ccall((:ddot_, "libLAPACK"),
                    Float64,
                    (Ref{Int32}, Ptr{Float64}, Ref{Int32}, Ptr{Float64}, Ref{Int32}),
                    n, DX, incx, DY, incy)
    return product
end
```

25.6 垃圾回收安全

When passing data to a `ccall`, it is best to avoid using the `pointer` function. Instead define a convert method and pass the variables directly to the `ccall`. `ccall` automatically arranges that all of its arguments will be preserved from garbage collection until the call returns. If a C API will store a reference to memory allocated by Julia, after the `ccall` returns, you must ensure that the object remains visible to the garbage collector. The suggested way to do this is to make a global variable of type `Array{Ref,1}` to hold these values, until the C library notifies you that it is finished with them.

Whenever you have created a pointer to Julia data, you must ensure the original data exists until you have finished using the pointer. Many methods in Julia such as `unsafe_load` and `String` make copies of data instead of taking ownership of the buffer, so that it is safe to free (or alter) the original data without affecting Julia. A notable exception is `unsafe_wrap` which, for performance reasons, shares (or can be told to take ownership of) the underlying buffer.

The garbage collector does not guarantee any order of finalization. That is, if a contained a reference to b and both a and b are due for garbage collection, there is no guarantee that b would be finalized after a. If proper finalization of a depends on b being valid, it must be handled in other ways.

25.7 Non-constant Function Specifications

A `(name, library)` function specification must be a constant expression. However, it is possible to use computed values as function names by staging through `eval` as follows:

```
@eval ccall(($(string("a", "b")), "lib"), ...
```

This expression constructs a name using `string`, then substitutes this name into a new `ccall` expression, which is then evaluated. Keep in mind that `eval` only operates at the top level, so within this expression local variables will not be available (unless their values are substituted with `$`). For this reason, `eval` is typically only used to form top-level definitions, for example when wrapping libraries that contain many similar functions. A similar example can be constructed for `@cfunction`.

However, doing this will also be very slow and leak memory, so you should usually avoid this and instead keep reading. The next section discusses how to use indirect calls to efficiently achieve a similar effect.

25.8 非直接调用

The first argument to `ccall` can also be an expression evaluated at run time. In this case, the expression must evaluate to a `Ptr`, which will be used as the address of the native function to call. This behavior occurs when

the first ccall argument contains references to non-constants, such as local variables, function arguments, or non-constant globals.

For example, you might look up the function via dlsym, then cache it in a shared reference for that session. For example:

```
macro dlsym(func, lib)
    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("myfunc", mylibvar), Cvoid, ())
```

25.9 Closure cfunctions

The first argument to @cfunction can be marked with a $, in which case the return value will instead be a `struct CFunction` which closes over the argument. You must ensure that this return object is kept alive until all uses of it are done. The contents and code at the cfunction pointer will be erased via a finalizer when this reference is dropped and atexit. This is not usually needed, since this functionality is not present in C, but can be useful for dealing with ill-designed APIs which don't provide a separate closure environment parameter.

```
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, Cvoid, (Ptr{T}, Csize_t, Csize_t, Ptr{Cvoid}),
        a, length(a), Base.elsize(a), callback)
    # 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

Closure @cfunction rely on LLVM trampolines, which are not available on all platforms (for example ARM and PowerPC).

25.10 关闭库

It is sometimes useful to close (unload) a library so that it can be reloaded. For instance, when developing C code for use with Julia, one may need to compile, call the C code from Julia, then close the library, make

an edit, recompile, and load in the new changes. One can either restart Julia or use the `Libdl` functions to manage the library explicitly, such as:

```
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 (symbol, library) tuple (remaining arguments
↪  are the
same).  Libdl.dlclose(lib) # Close the library explicitly.
```

Note that when using `ccall` with the tuple input (e.g., `ccall((:my_fcn, "./my_lib.so"), ...)`), the library is opened implicitly and it may not be explicitly closed.

25.11 调用规约

The second argument to `ccall` can optionally be a calling convention specifier (immediately preceding return type). 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 `gethostname`ccall 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))
```

请参阅 LLVM Language Reference 来获得更多信息。

There is one additional special calling convention `llvmcall`, which allows inserting calls to LLVM intrinsics directly. This can be especially useful when targeting unusual platforms such as GPGPUs. For example, for CUDA, we need to be able to read the thread index:

```
ccall("llvm.nvvm.read.ptx.sreg.tid.x", llvmcall, Int32, ())
```

As with any `ccall`, it is essential to get the argument signature exactly correct. Also, note that there is no compatibility layer that ensures the intrinsic makes sense and works on the current target, unlike the equivalent Julia functions exposed by `Core.Intrinsics`.

25.12 访问全局变量

Global variables exported by native libraries can be accessed by name using the `cglobal` function. The arguments to `cglobal` are a symbol specification identical to that used by `ccall`, and a type describing the value stored in the variable:

```
julia> cglobal((:errno, :libc), Int32)
Ptr{Int32} @0x00007f418d0816b8
```

The result is a pointer giving the address of the value. The value can be manipulated through this pointer using `unsafe_load` and `unsafe_store!`.

> **Note**
>
> This `errno` symbol may not be found in a library named "libc", as this is an implementation detail of your system compiler. Typically standard library symbols should be accessed just by name,

allowing the compiler to fill in the correct one. Also, however, the `errno` symbol shown in this example is special in most compilers, and so the value seen here is probably not what you expect or want. Compiling the equivalent code in C on any multi-threaded-capable system would typically actually call a different function (via macro preprocessor overloading), and may give a different result than the legacy value printed here.

25.13 Accessing Data through a Pointer

The following methods are described as "unsafe" because a bad pointer or type declaration can cause Julia to terminate abruptly.

Given a `Ptr{T}`, the contents of type T can generally be copied from the referenced memory into a Julia object using `unsafe_load(ptr, [index])`. The index argument is optional (default is 1), and follows the Julia-convention of 1-based indexing. This function is intentionally similar to the behavior of `getindex` and `setindex!` (e.g. `[]` access syntax).

The return value will be a new object initialized to contain a copy of the contents of the referenced memory. The referenced memory can safely be freed or released.

If T is `Any`, then the memory is assumed to contain a reference to a Julia object (a `jl_value_t*`), the result will be a reference to this object, and the object will not be copied. You must be careful in this case to ensure that the object was always visible to the garbage collector (pointers do not count, but the new reference does) to ensure the memory is not prematurely freed. Note that if the object was not originally allocated by Julia, the new object will never be finalized by Julia's garbage collector. If the `Ptr` itself is actually a `jl_value_t*`, it can be converted back to a Julia object reference by `unsafe_pointer_to_objref(ptr)`. (Julia values v can be converted to `jl_value_t*` pointers, as `Ptr{Cvoid}`, by calling `pointer_from_objref(v)`.)

The reverse operation (writing data to a `Ptr{T}`), can be performed using `unsafe_store!(ptr, value, [index])`. Currently, this is only supported for primitive types or other pointer-free (`isbits`) immutable struct types.

Any operation that throws an error is probably currently unimplemented and should be posted as a bug so that it can be resolved.

If the pointer of interest is a plain-data array (primitive type or immutable struct), the function `unsafe_wrap(Array, ptr,dims, own = false)` may be more useful. The final parameter should be true if Julia should "take ownership" of the underlying buffer and call `free(ptr)` when the returned `Array` object is finalized. If the `own` parameter is omitted or false, the caller must ensure the buffer remains in existence until all access is complete.

Arithmetic on the `Ptr` type in Julia (e.g. using +) does not behave the same as C's pointer arithmetic. Adding an integer to a `Ptr` in Julia always moves the pointer by some number of *bytes*, not elements. This way, the address values obtained from pointer arithmetic do not depend on the element types of pointers.

25.14 线程安全

Some C libraries execute their callbacks from a different thread, and since Julia isn't thread-safe you'll need to take some extra precautions. In particular, you'll need to set up a two-layered system: the C callback should only *schedule* (via Julia's event loop) the execution of your "real" callback. To do this, create an `AsyncCondition` object and `wait` on it:

```
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。

25.15 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。

25.16 C++

如需要直接易用的 C++ 接口，即直接用 Julia 写封装代码，请参考 Cxx。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 26

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)
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 27

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。

The environment variables that Julia uses generally start with JULIA. If InteractiveUtils.versioninfo is called with the keyword verbose=true, then the output will list any defined environment variables relevant for Julia, including those which include JULIA in their names.

> **Note**
>
> Some variables, such as JULIA_NUM_THREADS and JULIA_PROJECT, need to be set before Julia starts, therefore adding these to ~/.julia/config/startup.jl is too late in the startup process. In Bash, environment variables can either be set manually by running, e.g., export JULIA_NUM_THREADS=4 before starting Julia, or by adding the same command to ~/.bashrc or ~/.bash_profile to set the variable each time Bash is started.

27.1 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，一个 Linux 安装包的 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

A directory path that indicates which project should be the initial active project. Setting this environment variable has the same effect as specifying the `--project` start-up option, but `--project` has higher precedence. If the variable is set to `@.` then Julia tries to find a project directory that contains `Project.toml` or `JuliaProject.toml` file from the current directory and its parents. See also the chapter on Code Loading.

> **Note**
>
> `JULIA_PROJECT` must be defined before starting julia; defining it in `startup.jl` is too late in the startup process.

JULIA_LOAD_PATH

The `JULIA_LOAD_PATH` environment variable is used to populate the global Julia `LOAD_PATH` variable, which determines which packages can be loaded via `import` and `using` (see Code Loading).

Unlike the shell `PATH` variable, empty entries in `JULIA_LOAD_PATH` are expanded to the default value of `LOAD_PATH`, `["@", "@v#.#", "@stdlib"]` when populating `LOAD_PATH`. This allows easy appending, prepending, etc. of the load path value in shell scripts regardless of whether `JULIA_LOAD_PATH` is already set or not. For example, to prepend the directory `/foo/bar` to `LOAD_PATH` just do

```
export JULIA_LOAD_PATH="/foo/bar:$JULIA_LOAD_PATH"
```

If the `JULIA_LOAD_PATH` environment variable is already set, its old value will be prepended with `/foo/bar`. On the other hand, if `JULIA_LOAD_PATH` is not set, then it will be set to `/foo/bar:` which will expand to a `LOAD_PATH` value of `["/foo/bar", "@", "@v#.#", "@stdlib"]`. If `JULIA_LOAD_PATH` is set to the empty string, it expands to an empty `LOAD_PATH` array. In other words, the empty string is interpreted as a zero-element array, not a one-element array of the empty string. This behavior was chosen so that it would be possible to set an empty load path via the environment variable. If you want the default load path, either unset the environment variable or if it must have a value, set it to the string `:`.

JULIA_DEPOT_PATH

The `JULIA_DEPOT_PATH` environment variable is used to populate the global Julia `DEPOT_PATH` variable, which controls 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, configuration files, and the default location of the REPL's history file.

Unlike the shell `PATH` variable but similar to `JULIA_LOAD_PATH`, empty entries in `JULIA_DEPOT_PATH` are expanded to the default value of `DEPOT_PATH`. This allows easy appending, prepending, etc. of the depot path value in shell scripts regardless of whether `JULIA_DEPOT_PATH` is already set or not. For example, to prepend the directory `/foo/bar` to `DEPOT_PATH` just do

```
export JULIA_DEPOT_PATH="/foo/bar:$JULIA_DEPOT_PATH"
```

If the JULIA_DEPOT_PATH environment variable is already set, its old value will be prepended with /foo/bar. On the other hand, if JULIA_DEPOT_PATH is not set, then it will be set to /foo/bar: which will have the effect of prepending /foo/bar to the default depot path. If JULIA_DEPOT_PATH is set to the empty string, it expands to an empty DEPOT_PATH array. In other words, the empty string is interpreted as a zero-element array, not a one-element array of the empty string. This behavior was chosen so that it would be possible to set an empty depot path via the environment variable. If you want the default depot path, either unset the environment variable or if it must have a value, set it to the string :.

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
```

27.2 外部应用

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。

27.3 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），用来设置 Julia 可用线程的最大数。如果 $JULIA_NUM_THREADS 超过可用的物理 CPU 核心数，那么线程数设置为核心数。如果 $JULIA_NUM_THREADS 不是正数或没有设置，或者无法通过系统调用确定 CPU 核心数，那么线程数就会被设置为 1。

> **Note**
>
> JULIA_NUM_THREADS 必须在启动 julia 前定义；在启动过程中于 startup.jl 中定义它为时已晚。

Julia 1.5

In Julia 1.5 and above the number of threads can also be specified on startup using the `-t`/`--threads` command line argument.

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 z 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_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。

27.4 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。

JULIA_STACKFRAME_LINEINFO_COLOR

`Base.stackframe_lineinfo_color()`（默认值：粗体，`"\033[1m"`），栈跟踪时行信息在终端中的格式。

JULIA_STACKFRAME_FUNCTION_COLOR

`Base.stackframe_function_color()`（默认值：粗体，`"\033[1m"`），栈跟踪期间函数调用在终端中的形式。

27.5 调试和性能分析

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**
>
> 此变量生效要求 Julia 编译时带有 JIT 性能分析支持，请使用
>
> - 英特尔的 VTune™ Amplifier（在构建配置中将 USE_INTEL_JITEVENTS 设置为 1），或
> - OProfile（在构建配置中将 USE_OPROFILE_JITEVENTS 设置为 1）。

JULIA_LLVM_ARGS

传递给 LLVM 后端的参数。

Chapter 28

嵌入 Julia

正如我们在 调用 C 和 Fortran 代码 中看到的, Julia 有着简单高效的方法来调用 C 编写的函数。但有时恰恰相反，我们需要在 C 中调用 Julia 的函数。这可以将 Julia 代码集成到一个更大的 C/C++ 项目而无需在 C/C++ 中重写所有内容。Julia 有一个 C API 来实现这一目标。几乎所有编程语言都能以某种方式来调用 C 语言的函数，因此 Julia 的 C API 也就能够进行更多语言的桥接。(例如在 Python 或是 C# 中调用 Julia).

28.1 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。Windows 平台请参阅下一节。

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 Julia 源代码目录 `test/embedding/` 文件夹下的 `embedding.c` 文件。文件 `ui/repl.c` 则是另一个简单示例，用于设置链接 `libjulia` 时 `jl_options` 的选项。

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，强烈建议调用 `jl_atexit_hook`。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 OS/X 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

但通常来说，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 。另外，尽管很多时候 `julia-config.jl` 会在目录 `/usr/local` 中出现多次，不过也未必如此，但 Julia 也定位 `julia-config.jl`，并且可以使用 makefile 来利用它。上面的示例程序使用 Makefile 来扩展。:

```
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。

28.2 在 Windows 使用 Visual Studio 进行高级别嵌入

If the `JULIA_DIR` environment variable hasn't been setup, add it using the System panel before starting Visual Studio. The `bin` folder under JULIA_DIR should be on the system PATH.

We start by opening Visual Studio and creating a new Console Application project. To the 'stdafx.h' header file, add the following lines at the end:

```
#include <julia.h>
```

Then, replace the main() function in the project with this 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;
}
```

The next step is to set up the project to find the Julia include files and the libraries. It's important to know whether the Julia installation is 32- or 64-bits. Remove any platform configuration that doesn't correspond to the Julia installation before proceeding.

Using the project Properties dialog, go to `C/C++` | `General` and add `$(JULIA_DIR)\include\julia\` to the Additional Include Directories property. Then, go to the `Linker` | `General` section and add `$(JULIA_DIR)\lib` to the Additional Library Directories property. Finally, under `Linker` | `Input`, add `libjulia.dll.a;libopenlibm.dll.a;` to the list of libraries.

At this point, the project should build and run.

28.3 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。

28.4 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

28.5 内存管理

正如我们所见，Julia 对象在 C 中表示为指针。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。

The GC can only run when Julia objects are allocated. Calls like `jl_box_float64` perform allocation, and allocation might also happen at any point in running Julia code. However, it is generally safe to use pointers

in between `jl_...` calls. But in order to make sure that values can survive `jl_...` calls, we have to tell Julia that we still hold a reference to Julia root values, a process called "GC rooting". Rooting a value will ensure that the garbage collector does not accidentally identify this value as unused and free the memory backing that value. This can be done using the `JL_GC_PUSH` macros:

```
jl_value_t *ret = jl_eval_string("sqrt(2.0)");
JL_GC_PUSH1(&ret);
// Do something with ret
JL_GC_POP();
```

The `JL_GC_POP` call releases the references established by the previous `JL_GC_PUSH`. Note that `JL_GC_PUSH` stores references on the C stack, so it must be exactly paired with a `JL_GC_POP` before the scope is exited. That is, before the function returns, or control flow otherwise leaves the block in which the `JL_GC_PUSH` was invoked.

Several Julia values can be pushed at once using the `JL_GC_PUSH2` , `JL_GC_PUSH3` , `JL_GC_PUSH4` , `JL_GC_PUSH5` , and `JL_GC_PUSH6` macros. To push an array of Julia values one can use the `JL_GC_PUSHARGS` macro, which can be used as follows:

```
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();
```

Each scope must have only one call to `JL_GC_PUSH*`. Hence, if all variables cannot be pushed once by a single call to `JL_GC_PUSH*`, or if there are more than 6 variables to be pushed and using an array of arguments is not an option, then one can use inner blocks:

```
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.
```

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);
```

If the variable is immutable, then it needs to be wrapped in an equivalent mutable container or, preferably, in a `RefValue{Any}` before it is pushed to `IdDict`. In this approach, the container has to be created or filled in via C code using, for example, the function `jl_new_struct`. If the container is created by `jl_call*`, then you will need to reload the pointer to be used in C code.

```
// 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);
```

The GC can be allowed to deallocate a variable by removing the reference to it from `refs` using the function `delete!`, provided that no other reference to the variable is kept anywhere:

```
jl_function_t* delete = jl_get_function(jl_base_module, "delete!");
jl_call2(delete, refs, rvar);
```

As an alternative for very simple cases, it is possible to just create a global container of type `Vector{Any}` and fetch the elements from that when necessary, or even to create one global variable per pointer using

```
jl_set_global(jl_main_module, jl_symbol("var"), var);
```

Updating fields of GC-managed objects

The garbage collector operates under the assumption that it is aware of every old-generation object pointing to a young-generation one. Any time a pointer is updated breaking that assumption, it must be signaled to the collector with the `jl_gc_wb` (write barrier) function like so:

```
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

28.6 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_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。

28.7 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);
```

在这个例子中假定 x 是一个 int 值。

Chapter 29

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。

29.1 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。

29.2 包的联合

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。

29.3 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. **A project environment** is a directory with a project file and an optional manifest file, and forms an *explicit environment*. The project file determines what the names and identities of the direct dependencies of a project are. The manifest file, if present, gives a complete dependency graph, including all direct and indirect dependencies, exact versions of each dependency, and sufficient information to locate and load the correct version.

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(),
)
```

需要注意的一些概括性规则：

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 = [env_1, env_2, …]，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。

29.4 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 30

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：

- You do not have to make any modifications to your code to take timing measurements.
- 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。

30.1 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()
```

To see the profiling results, there are several graphical browsers. One "family" of visualizers is based on FlameGraphs.jl, with each family member providing a different user interface:

- Juno is a full IDE with built-in support for profile visualization
- ProfileView.jl is a stand-alone visualizer based on GTK
- ProfileVega.jl uses VegaLight and integrates well with Jupyter notebooks
- StatProfilerHTML produces HTML and presents some additional summaries, and also integrates well with Jupyter notebooks
- ProfileSVG renders SVG

An entirely independent approach to profile visualization is PProf.jl, which uses the external `pprof` tool.

Here, though, we'll use the text-based display that comes with the standard library:

```
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。

30.2 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()`。

30.3 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
```

30.4 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。

Chapter 31

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。内存分配总量可以用 `@time` 和 `@allocated`，触发内存分配的特定行通常可以通过这些行产生的垃圾分配成本从性能分析中推断出来。但是，直接测量每行代码的内存分配总量有时会更高效。

为了逐行测量内存分配，启动 Julia 时请使用命令行选项 `--track-allocation=<setting>`，该选项的可选值有 `none`（默认值，不测量内存分配）、`user`（测量除 Julia core 代码之外的所有代码的内存分配）或 `all`（测量 Julia 代码中每一行的内存分配）。这会为每行已编译的代码测量内存。在退出 Julia 时，累积的结果将写入到文本文件中，此文本文件名称为该文件名称后加 `.mem`，并与源文件位于同一目录下。该文件的每行列出内存分配的总字节数。`Coverage` 包括了一些基本分析工具，例如，按照内存分配的字节数对行进行排序的工具。

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。

Chapter 32

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`
```

Or similary with `perf`:

```
$ ENABLE_JITPROFILING=1 perf record -o /tmp/perf.data --call-graph dwarf ./julia /test/fastmath.jl
$ perf report --call-graph -G
```

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 33

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。

33.1 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
```

33.2 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> top_frame.linfo
MethodInstance for eval(::Module, ::Expr)

julia> top_frame.inlined
false

julia> top_frame.from_c
false
```

```
julia> top_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。

33.3 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
[...]
```

33.4 异常栈与 `catch_stack`

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。

存放当前异常的栈可通过测试函数 `Base.catch_stack` 获取，例如

```
julia> try
            error("(A) The root cause")
        catch
            try
                error("(B) An exception while handling the exception")
            catch
                for (exc, bt) in Base.catch_stack()
                    showerror(stdout, exc, bt)
                    println()
                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。

异常栈被存放于发生异常的 Task 处。当某个任务失败，出现意料外的异常时，`catch_stack(task)` 可能会被用于观察该任务的异常栈。

33.5 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 34

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。

34.1 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。

任何注重性能或者需要测试性能的代码都应该被放置在函数之中。

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
```

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**
>
> All code in the REPL is evaluated in global scope, so a variable defined and assigned at top level will be a **global** variable. Variables defined at top level scope inside modules are also global.

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。

34.2 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()
  0.017705 seconds (15.28 k allocations: 694.484 KiB)
496.84883432553846

julia> @time sum_global()
  0.000140 seconds (3.49 k allocations: 70.313 KiB)
496.84883432553846
```

在第一次调用函数(@time sum_global())的时候，它会被编译。如果你这次会话中还没有使用过@time，这时也会编译计时需要的相关函数。你不必认真对待这次运行的结果。接下来看第二次运行，除了运行的耗时以外，它还表明了分配了大量的内存。我们这里仅仅是计算了一个 64 位浮点向量元素和，因此这里应该没有申请内存的必要（至少不用在 @time 报告的堆上申请内存）。

未被预料的内存分配往往说明你的代码中存在一些问题，这些问题常常是由于类型的稳定性或者创建了太多临时的小数组。因此，除了分配内存本身，这也很可能说明你所写的函数远没有生成性能良好的代码。认真对待这些现象，遵循接下来的建议。

如果你换成将 x 作为参数传给函数，就可以避免内存的分配（这里报告的内存分配是由于在全局作用域中运行 @time 导致的），而且在第一次运行之后运行速度也会得到显著的提高。

```
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)
  0.007701 seconds (821 allocations: 43.059 KiB)
```

```
496.84883432553846

julia> @time sum_arg(x)
  0.000006 seconds (5 allocations: 176 bytes)
496.84883432553846
```

这里出现的 5 个内存分配是由于在全局作用域中运行 `@time` 宏导致的。如果我们在函数内运行时间测试，我们将发现事实上并没有发生任何内存分配。

```
julia> time_sum(x) = @time sum_arg(x);

julia> time_sum(x)
  0.000001 seconds
496.84883432553846
```

In some situations, your function may need to allocate memory as part of its operation, and this can complicate the simple picture above. In such cases, consider using one of the tools below to diagnose problems, or write a version of your function that separates allocation from its algorithmic aspects (see Pre-allocating outputs).

> **Note**
>
> For more serious benchmarking, consider the BenchmarkTools.jl package which among other things evaluates the function multiple times in order to reduce noise.

34.3 Tools

Julia and its package ecosystem includes tools that may help you diagnose problems and improve the performance of your code:

- Profiling allows you to measure the performance of your running code and identify lines that serve as bottlenecks. For complex projects, the ProfileView package can help you visualize your profiling results.
- The Traceur package can help you find common performance problems in your code.
- Unexpectedly-large memory allocations–as reported by `@time`, `@allocated`, or the profiler (through calls to the garbage-collection routines)–hint that there might be issues with your code. If you don't see another reason for the allocations, suspect a type problem. You can also start Julia with the `--track-allocation=user` option and examine the resulting `*.mem` files to see information about where those allocations occur. See Memory allocation analysis.
- `@code_warntype` generates a representation of your code that can be helpful in finding expressions that result in type uncertainty. See `@code_warntype` below.

34.4 Avoid containers with abstract type parameters

When working with parameterized types, including arrays, it is best to avoid parameterizing with abstract types where possible.

Consider the following:

```
julia> a = Real[]
Real[]

julia> push!(a, 1); push!(a, 2.0); push!(a, π)
3-element Array{Real,1}:
 1
 2.0
 π = 3.1415926535897...
```

因为 a 是一个抽象类型Real的数组，它必须能容纳任何一个 Real 值。因为 Real 对象可以有任意的大小和结构，a 必须用指针的数组来表示，以便能独立地为 Real 对象进行内存分配。但是如果我们只允许同样类型的数，比如Float64，才能存在 a 中，它们就能被更有效率地存储：

```
julia> a = Float64[]
Float64[]

julia> push!(a, 1); push!(a, 2.0); push!(a,  π)
3-element Array{Float64,1}:
 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。

也请参见在参数类型下的讨论。

34.5 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。

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
```

例如:

```
julia> c = MySimpleContainer(1:3);

julia> typeof(c)
MySimpleContainer{UnitRange{Int64}}

julia> c = MySimpleContainer([1:3;]);

julia> typeof(c)
MySimpleContainer{Array{Int64,1}}

julia> b = MyAmbiguousContainer(1:3);

julia> typeof(b)
MyAmbiguousContainer{Int64}

julia> b = MyAmbiguousContainer([1:3;]);

julia> typeof(b)
MyAmbiguousContainer{Int64}
```

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`。

如果类型注释中的类型是在运行时构造的，那么类型注释不会增强（并且实际上可能会降低）性能。这是因为编译器无法使用该类型注释来专门化代码，而类型检查本身又需要时间。例如，在以下代码中：

```
function nr(a, prec)
    ctype = prec == 32 ? Float32 : Float64
    b = Complex{ctype}(a)
    c = (b + 1.0f0)::Complex{ctype}
    abs(c)
end
```

the annotation of c harms performance. To write performant code involving types constructed at run-time, use the function-barrier technique discussed below, and ensure that the constructed type appears among the argument types of the kernel function so that the kernel operations are properly specialized by the compiler. For example, in the above snippet, as soon as b is constructed, it can be passed to another function k, the kernel. If, for example, function k declares b as an argument of type `Complex{T}`, where T is a type parameter, then a type annotation appearing in an assignment statement within k of the form:

```
c = (b + 1.0f0)::Complex{T}
```

不会降低性能（但也不会提高），因为编译器可以在编译 k 时确定 c 的类型。

Be aware of when Julia avoids specializing

As a heuristic, Julia avoids automatically specializing on argument type parameters in three specific cases: `Type`, `Function`, and `Vararg`. Julia will always specialize when the argument is used within the method, but not if the argument is just passed through to another function. This usually has no performance impact at runtime and improves compiler performance. If you find it does have a performance impact at runtime in your case, you can trigger specialization by adding a type parameter to the method declaration. Here are some examples:

This will not specialize:

```
function f_type(t)  # or t::Type
    x = ones(t, 10)
    return sum(map(sin, x))
end
```

but this will:

```
function g_type(t::Type{T}) where T
    x = ones(T, 10)
    return sum(map(sin, x))
end
```

These will not specialize:

```
f_func(f, num) = ntuple(f, div(num, 2))
g_func(g::Function, num) = ntuple(g, div(num, 2))
```

but this will:

```
h_func(h::H, num) where {H} = ntuple(h, div(num, 2))
```

This will not specialize:

```
f_vararg(x::Int...) = tuple(x...)
```

but this will:

```
g_vararg(x::Vararg{Int, N}) where {N} = tuple(x...)
```

One only needs to introduce a single type parameter to force specialization, even if the other types are unconstrained. For example, this will also specialize, and is useful when the arguments are not all of the same type:

```
h_vararg(x::Vararg{Any, N}) where {N} = tuple(x...)
```

Note that `@code_typed` and friends will always show you specialized code, even if Julia would not normally specialize that method call. You need to check the method internals if you want to see whether specializations are generated when argument types are changed, i.e., if `(@which f(...)).specializations` contains specializations for the argument in question.

34.6 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：

```
norm(x::Vector) = sqrt(real(dot(x, x)))
norm(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。

34.7 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`。

34.8 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
- Declare the type of x explicitly as x::Float64 = 1
- Use an explicit conversion by x = oneunit(Float64)
- 使用第一个循环迭代初始化，即 x = 1 / rand()，接着循环 for i = 2:10

34.9 Separate kernel functions (aka, function barriers)

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 Array{Float64,1}:
 2.0
 2.0
 2.0
```

这应该写作：

```
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 Array{Float64,1}:
 2.0
 2.0
 2.0
```

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。

34.10 Types with values-as-parameters

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 Array{Float64,2}:
 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 Array{Float64,2}:
 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。

Now, one very good way to solve such problems is by using the function-barrier technique. However, in some cases you might want to eliminate the type-instability altogether. In such cases, one approach is to pass the dimensionality as a parameter, for example through Val{T}() (see "Value types"):

```
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 Array{Float64,2}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

Julia has a specialized version of `ntuple` that accepts a `Val{::Int}` instance as the second parameter; by passing `N` as a type-parameter, you make its "value" known to the compiler. Consequently, this version of `array3` allows the compiler to predict the return type.

However, making use of such techniques can be surprisingly subtle. For example, it would be of no help if you called `array3` from a function like this:

```
function call_array3(fillval, n)
    A = array3(fillval, Val(n))
end
```

Here, you've created the same problem all over again: the compiler can't guess what `n` is, so it doesn't know the *type* of `Val(n)`. Attempting to use `Val`, but doing so incorrectly, can easily make performance *worse* in many situations. (Only in situations where you're effectively combining `Val` with the function-barrier trick, to make the kernel function more efficient, should code like the above be used.)

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
```

In this example, `N` is passed as a parameter, so its "value" is known to the compiler. Essentially, `Val(T)` works only when `T` is either hard-coded/literal (`Val(3)`) or already specified in the type-domain.

34.11 The dangers of abusing multiple dispatch (aka, more on types with values-as-parameters)

Once one learns to appreciate multiple dispatch, there's an understandable tendency to go overboard and try to use it for everything. For example, you might imagine using it to store information, e.g.

```
struct Car{Make, Model}
    year::Int
    ...more fields...
end
```

and then dispatch on objects like `Car{:Honda,:Accord}(year, args...)`.

This might be worthwhile when either of the following are true:

- You require CPU-intensive processing on each `Car`, and it becomes vastly more efficient if you know the `Make` and `Model` at compile time and the total number of different `Make` or `Model` that will be used is not too large.
- You have homogenous lists of the same type of `Car` to process, so that you can store them all in an `Array{Car{:Honda,:Accord},N}`.

When the latter holds, a function processing such a homogenous array can be productively specialized: Julia knows the type of each element in advance (all objects in the container have the same concrete type), so Julia can "look up" the correct method calls when the function is being compiled (obviating the need to check at run-time) and thereby emit efficient code for processing the whole list.

When these do not hold, then it's likely that you'll get no benefit; worse, the resulting "combinatorial explosion of types" will be counterproductive. If `items[i+1]` has a different type than `item[i]`, Julia has to look up the type at run-time, search for the appropriate method in method tables, decide (via type intersection) which one matches, determine whether it has been JIT-compiled yet (and do so if not), and then make the call. In essence, you're asking the full type- system and JIT-compilation machinery to basically execute the equivalent of a switch statement or dictionary lookup in your own code.

Some run-time benchmarks comparing (1) type dispatch, (2) dictionary lookup, and (3) a "switch" statement can be found on the mailing list.

Perhaps even worse than the run-time impact is the compile-time impact: Julia will compile specialized functions for each different `Car{Make, Model}`; if you have hundreds or thousands of such types, then every function that accepts such an object as a parameter (from a custom `get_year` function you might write yourself, to the generic `push!` function in Julia Base) will have hundreds or thousands of variants compiled for it. Each of these increases the size of the cache of compiled code, the length of internal lists of methods, etc. Excess enthusiasm for values-as-parameters can easily waste enormous resources.

34.12 Access arrays in memory order, along columns

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 Array{Int64,2}:
 1  2
 3  4

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

This convention for ordering arrays is common in many languages like Fortran, Matlab, and R (to name a few). The alternative to column-major ordering is row-major ordering, which is the convention adopted by C and Python (`numpy`) among other languages. Remembering the ordering of arrays can have significant performance effects when looping over arrays. A rule of thumb to keep in mind is that with column-major arrays, the first index changes most rapidly. Essentially this means that looping will be faster if the inner-most loop index is the first to appear in a slice expression. Keep in mind that indexing an array with `:` is an implicit loop that iteratively accesses all elements within a particular dimension; it can be faster to extract columns than rows, for example.

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。

34.13 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）。

34.14 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. (Of course, if you just do `f.(x)` then it is as fast as `fdot(x)` in this example, but in many contexts it is more convenient to just sprinkle some dots in your expressions rather than defining a separate function for each vectorized operation.)

34.15 Consider using views for slices

In Julia, an array "slice" expression like `array[1:5, :]` creates a copy of that data (except on the left-hand side of an assignment, where `array[1:5, :] = ...` assigns in-place to that portion of `array`). If you are doing many operations on the slice, this can be good for performance because it is more efficient to work with a smaller contiguous copy than it would be to index into the original array. On the other hand, if you are just doing a few simple operations on the slice, the cost of the allocation and copy operations can be substantial.

An alternative is to create a "view" of the array, which is an array object (a `SubArray`) that actually references the data of the original array in-place, without making a copy. (If you write to a view, it modifies the original array's data as well.) This can be done for individual slices by calling `view`, or more simply for a whole expression or block of code by putting `@views` in front of that expression. For example:

```
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 (7 allocations: 7.630 MB)

julia> @time fview(x);
  0.001020 seconds (6 allocations: 224 bytes)
```

请注意，该函数的 `fview` 版本提速了 3 倍且减少了内存分配。

34.16 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操作前，将其复制到连续的数组中可能带来巨大的加速，正如下例所示。其中，矩阵和向量在相乘前会访问其 800,000 个已被随机混洗的索引处的值。将视图复制到普通数组会加速乘法，即使考虑了复制操作的成本。

```
julia> using Random

julia> x = randn(1_000_000);

julia> inds = shuffle(1:1_000_000)[1:800000];

julia> A = randn(50, 1_000_000);

julia> xtmp = zeros(800_000);

julia> Atmp = zeros(50, 800_000);

julia> @time sum(view(A, :, inds) * view(x, inds))
  0.412156 seconds (14 allocations: 960 bytes)
-4256.759568345458

julia> @time begin
           copyto!(xtmp, view(x, inds))
           copyto!(Atmp, view(A, :, inds))
           sum(Atmp * xtmp)
       end
  0.285923 seconds (14 allocations: 960 bytes)
-4256.759568345134
```

倘若副本本身的内存足够大，那么将视图复制到数组的成本可能远远超过在连续数组上执行矩阵乘法所带来的加速。

34.17 避免 I/0 中的字符串插值

将数据写入到文件（或其他 I/0 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))
```

34.18 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。

34.19 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。

34.20 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)`。

34.21 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。

The common idiom of using 1:n to index into an AbstractArray is not safe if the Array uses unconventional indexing, and may cause a segmentation fault if bounds checking is turned off. Use `LinearIndices(x)` or `eachindex(x)` instead (see also Arrays with custom indices).

> **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
```

34.22 Treat Subnormal Numbers as Zeros

Subnormal numbers, formerly called denormal numbers, are useful in many contexts, but incur a performance penalty on some hardware. A call `set_zero_subnormals(true)` grants permission for floating-point operations to treat subnormal inputs or outputs as zeros, which may improve performance on some hardware. A call `set_zero_subnormals(false)` enforces strict IEEE behavior for subnormal numbers.

Below is an example where subnormals noticeably impact performance on some hardware:

```
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
```

This gives an output similar to

```
  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)
```

Note how each even iteration is significantly faster.

This example generates many subnormal numbers because the values in a become an exponentially decreasing curve, which slowly flattens out over time.

Treating subnormals as zeros should be used with caution, because doing so breaks some identities, such as `x-y == 0` implies `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)
```

In some applications, an alternative to zeroing subnormal numbers is to inject a tiny bit of noise. For example, instead of initializing a with zeros, initialize it with:

```
a = rand(Float32,1000) * 1.f-9
```

34.23 @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)
Variables
  #self#::Core.Compiler.Const(f, false)
  x::Float64
  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
```

Interpreting the output of `@code_warntype`, like that of its cousins `@code_lowered`, `@code_typed`, `@code_llvm`, and `@code_native`, takes a little practice. Your code is being presented in form that has been heavily digested on its way to generating compiled machine code. Most of the expressions are annotated by a type, indicated by the `::T` (where T might be `Float64`, for example). The most important characteristic of `@code_warntype` is that non-concrete types are displayed in red; since this document is written in Markdown, which has no color, in this document, red text is denoted by uppercase.

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：获取值为非叶类型的字段。在这种情况下，ArrayContainer 具有字段 data::Array{T}。但是 Array 也需要维度 N 来成为具体类型。
  - 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

34.24 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
```

let 代码块创建了一个新的变量 r，它的作用域只是内部函数。第二种技术在捕获变量存在时完全恢复了语言性能。请注意，这是编译器的一个快速发展的方面，未来的版本可能不需要依靠这种程度的程序员注释来获得性能。与此同时，一些用户提供的包（如 FastClosures）会自动插入像在 abmult3 中那样的 let 语句。

Chapter 35

Checking for equality with a singleton

When checking if a value is equal to some singleton it can be better for performance to check for identicality (===) instead of equality (==). The same advice applies to using `!==` over `!=`. These type of checks frequently occur e.g. when implementing the iteration protocol and checking if `nothing` is returned from `iterate`.

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。

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

```
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

也可以通过 IJulia 在浏览器中与 Julia REPL 进行交互，请到该库的主页查看详细用法。

36.3 Revise-based workflows

Whether you're at the REPL or in IJulia, you can typically improve your development experience with Revise. It is common to configure Revise to start whenever julia is started, as per the instructions in the Revise documentation. Once configured, Revise will track changes to files in any loaded modules, and to any files loaded in to the REPL with `includet` (but not with plain `include`); you can then edit the files and the changes take effect without restarting your julia session. A standard workflow is similar to the REPL-based workflow above, with the following modifications:

1. Put your code in a module somewhere on your load path. There are several options for achieving this, of which two recommended choices are:

   a. For long-term projects, use PkgTemplates:

   `julia using PkgTemplates t = Template() generate("MyPkg", t)` This will create a blank package, `"MyPkg"`, in your `.julia/dev` directory. Note that PkgTemplates allows you to control many different options through its `Template` constructor.

   In step 2 below, edit `MyPkg/src/MyPkg.jl` to change the source code, and `MyPkg/test/runtests.jl` for the tests.

   b. For "throw-away" projects, you can avoid any need for cleanup by doing your work in your temporary directory (e.g., `/tmp`).

   Navigate to your temporary directory and launch Julia, then do the following:

   `julia pkg> generate MyPkg # type ] to enter pkg mode julia> push!(LOAD_PATH, pwd()) # hit backspace to exit pkg mode` If you restart your Julia session you'll have to re-issue that command modifying `LOAD_PATH`.

   In step 2 below, edit `MyPkg/src/MyPkg.jl` to change the source code, and create any test file of your choosing.

2. Develop your package

   Before loading any code, make sure you're running Revise: say `using Revise` or follow its documentation on configuring it to run automatically.

   Then navigate to the directory containing your test file (here assumed to be `"runtests.jl"`) and do the following:

```
julia> using MyPkg

julia> include("runtests.jl")
```

You can iteratively modify the code in MyPkg in your editor and re-run the tests with `include("runtests.jl")`. You generally should not need to restart your Julia session to see the changes take effect (subject to a few limitations, see https://timholy.github.io/Revise.jl/stable/limitations/).

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 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）。

37.2 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。

37.3 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。

37.4 Append ! to names of functions that modify their arguments

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。

37.5 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。

37.6 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。

37.7 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。
- 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。

37.8 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。In cases like fill!(x, v), this is 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。

37.9 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。

37.10 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)
```

37.11 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。

37.12 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。

37.13 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。

默认使用实例是比较受推崇的风格，然后只在为了解决一些问题必要时添加涉及到 `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。

37.14 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。

37.15 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。

37.16 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。

37.17 避免类型盗版

“类型盗版”（type piracy）指的是扩展或是重定义 Base 或其它包中的并不是你所定义的类型的方法。在某些情况下，你可以几乎毫无副作用地逃避类型盗版。但在极端情况下，你甚至会让 Julia 崩溃（比如说你的方法扩展或重定义造成了对 `ccall` 传入了无效的输入）。类型盗版也让代码推导变得更复杂，且可能会引入难以预料和诊断的不兼容性。

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。

37.18 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。

37.19 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)`。

37.20 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 General

Is Julia named after someone or something?

No.

Why don't you compile Matlab/Python/R/⋯code to Julia?

Since many people are familiar with the syntax of other dynamic languages, and lots of code has already been written in those languages, it is natural to wonder why we didn't just plug a Matlab or Python front-end into a Julia back-end (or "transpile" code to Julia) in order to get all the performance benefits of Julia without requiring programmers to learn a new language. Simple, right?

The basic issue is that there is *nothing special about Julia's compiler*: we use a commonplace compiler (LLVM) with no "secret sauce" that other language developers don't know about. Indeed, Julia's compiler is in many ways much simpler than those of other dynamic languages (e.g. PyPy or LuaJIT). Julia's performance advantage derives almost entirely from its front-end: its language semantics allow a well-written Julia program to *give more opportunities to the compiler* to generate efficient code and memory layouts. If you tried to compile Matlab or Python code to Julia, our compiler would be limited by the semantics of Matlab or Python to producing code no better than that of existing compilers for those languages (and probably worse). The key role of semantics is also why several existing Python compilers (like Numba and Pythran) only attempt to optimize a small subset of the language (e.g. operations on Numpy arrays and scalars), and for this subset they are already doing at least as well as we could for the same semantics. The people working on those projects are incredibly smart and have accomplished amazing things, but retrofitting a compiler onto a language that was designed to be interpreted is a very difficult problem.

Julia's advantage is that good performance is not limited to a small subset of "built-in" types and operations, and one can write high-level type-generic code that works on arbitrary user-defined types while remaining fast and memory-efficient. Types in languages like Python simply don't provide enough information to the compiler for similar capabilities, so as soon as you used those languages as a Julia front-end you would be stuck.

For similar reasons, automated translation to Julia would also typically generate unreadable, slow, non-idiomatic code that would not be a good starting point for a native Julia port from another language.

On the other hand, language *interoperability* is extremely useful: we want to exploit existing high-quality code in other languages from Julia (and vice versa)! The best way to enable this is not a transpiler, but rather via easy inter-language calling facilities. We have worked hard on this, from the built-in `ccall` intrinsic (to call C and Fortran libraries) to JuliaInterop packages that connect Julia to Python, Matlab, C++, and more.

38.2 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。

如果关心内存用量，一个对象总能被一个占用更少内存的对象替换掉。例如，如果 `A` 是一个不再需要的 GB 量级的数组，可以使用 `A = nothing` 来释放内存。该内存将在下一次垃圾回收器运行时被释放，也可以使用 `gc()` 强制进行垃圾回收。另外，试图使用 `A` 很可能导致错误，因为大部分方法（method）在 `Nothing` 类型上没有定义。

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)
...
```

38.3 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`。

How do I catch CTRL-C in a script?

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`。

怎样通过 `#!/usr/bin/env` 传递参数给 `julia`？

通过类似 `#!/usr/bin/env julia --startup-file=no` 的方式，使用 shebang 传递选项给 Julia 的方法，可能在像 Linux 这样的平台上无法正常工作。这是因为各平台上 shebang 的参数解析是平台相关的，

并且尚未标准化。在类 Unix 的环境中，可以通过以 bash 脚本作为可执行脚本的开头，并使用 exec 代替给 julia 传递选项的过程，来可靠的为 julia 传递选项。

```
#!/bin/bash
#=
exec julia --color=yes --startup-file=no "${BASH_SOURCE[0]}" "$@"
=#

@show ARGS  # put any Julia code here
```

在以上例子中，位于 #= 和 =# 之间的代码可以当作一个 bash 脚本。因为这些代码放在 Julia 的多行注释中，所以 Julia 会忽略它们。在 =# 之后的 Julia 代码会被 bash 忽略，J 因为当文件解析到 exec 语句时会停止解析，开始执行命令。

Note

In order to catch CTRL-C in the script you can use

```
#!/bin/bash
#=
exec julia --color=yes --startup-file=no -e 'include(popfirst!(ARGS))' \
    "${BASH_SOURCE[0]}" "$@"
=#

@show ARGS  # put any Julia code here
```

instead. Note that with this strategy PROGRAM_FILE will not be set.

38.4 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。

However, if x is bound to an object of type Array (or any other *mutable* type). From within the function, you cannot "unbind" x from this Array, but you *can* change its content. For example:

```
julia> x = [1,2,3]
3-element Array{Int64,1}:
 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 Array{Int64,1}:
 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 Array{Int64,1}:
 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 threetup()
           x, y = [3, 3]
           x, y # returns a tuple
       end
threetup (generic function with 1 method)

julia> function threearr()
           x, y = [3, 3] # returns an array
       end
threearr (generic function with 1 method)

julia> threetup()
(3, 3)

julia> threearr()
2-element Array{Int64,1}:
 3
 3
```

38.5 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)
```

It returns either an `Int` or a `Float64` depending on the value of its argument. Since Julia can't predict the return type of this function at compile-time, any computation that uses it must be able to cope with values of both types, which makes it hard to produce fast machine code.

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 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
```

How can I constrain or compute type parameters?

The parameters of a parametric type can hold either types or bits values, and the type itself chooses how it makes use of these parameters. For example, `Array{Float64, 2}` is parameterized by the type `Float64` to express its element type and the integer value 2 to express its number of dimensions. When defining your own parametric type, you can use subtype constraints to declare that a certain parameter must be a subtype (`<:`) of some abstract type or a previous type parameter. There is not, however, a dedicated syntax to declare that a parameter must be a *value* of a given type —that is, you cannot directly declare that a dimensionality-like parameter `isa Int` within the `struct` definition, for example. Similarly, you cannot do computations (including simple things like addition or subtraction) on type parameters. Instead, these sorts of constraints and relationships may be expressed through additional type parameters that are computed and enforced within the type's constructors.

As an example, consider

```
struct ConstrainedType{T,N,N+1} # NOTE: INVALID SYNTAX
    A::Array{T,N}
    B::Array{T,N+1}
end
```

where the user would like to enforce that the third type parameter is always the second plus one. This can be implemented with an explicit type parameter that is checked by an inner constructor method (where it can be combined with other checks):

```
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
```

This check is usually *costless*, as the compiler can elide the check for valid concrete types. If the second argument is also computed, it may be advantageous to provide an outer constructor method that performs this calculation:

```
ConstrainedType(A) = ConstrainedType(A, compute_B(A))
```

Why does Julia use native machine integer arithmetic?

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> typemax(Int)
9223372036854775807

julia> ans+1
-9223372036854775808

julia> -ans
-9223372036854775808

julia> 2*ans
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。

让整数算法静默溢出的最合理的备用方案是所有地方都使用检查算法，当加法、减法和乘法溢出，产生不正确的值时引发错误。在blog post中，Dan Luu 分析了这个方案，发现这个方案理论上的性能微不足道，但是最终仍然会消耗大量的性能因为编译器（LLVM 和 GCC）无法在加法溢出检测处优雅地进行优化。如果未来有所进步我们会考虑在 Julia 中默认设置为检查整数算法，但是现在，我们需要和溢出可能共同相处。

In the meantime, overflow-safe integer operations can be achieved through the use of external libraries such as SaferIntegers.jl. Note that, as stated previously, the use of these libraries significantly increases the execution time of code using the checked integer types. However, for limited usage, this is far less of an issue than if it were used for all integer operations. You can follow the status of the discussion here.

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
```

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。

38.6 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）。

38.7 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。

In some languages, the empty tuple (()) is considered the canonical form of nothingness. However, in julia it is best thought of as just a regular tuple that happens to contain zero values.

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。

38.8 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。

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：

```
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。

38.9 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> @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> @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
```

38.10 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。
- 零维数组原生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"trailing one" 规则，as for all other array dimensionalities, so you can indeed index them as A[1], A[1,1], etc; see 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。

Why are my Julia benchmarks for linear algebra operations different from other languages?

You may find that simple benchmarks of linear algebra building blocks like

```
using BenchmarkTools
A = randn(1000, 1000)
B = randn(1000, 1000)
@btime $A \ $B
@btime $A * $B
```

can be different when compared to other languages like Matlab or R.

Since operations like this are very thin wrappers over the relevant BLAS functions, the reason for the discrepancy is very likely to be

1. the BLAS library each language is using,
2. the number of concurrent threads.

Julia compiles and uses its own copy of OpenBLAS, with threads currently capped at 8 (or the number of your cores).

Modifying OpenBLAS settings or compiling Julia with a different BLAS library, eg Intel MKL, may provide performance improvements. You can use MKL.jl, a package that makes Julia's linear algebra use Intel MKL BLAS and LAPACK instead of OpenBLAS, or search the discussion forum for suggestions on how to set this up manually. Note that Intel MKL cannot be bundled with Julia, as it is not open source.

38.11 Julia 版本发布

Do I want to use the Stable, LTS, or nightly version of Julia?

The Stable version of Julia is the latest released version of Julia, this is the version most people will want to run. It has the latest features, including improved performance. The Stable version of Julia is versioned according to SemVer as v1.x.y. A new minor release of Julia corresponding to a new Stable version is made approximately every 4-5 months after a few weeks of testing as a release candidate. Unlike the LTS version the a Stable version will not normally recieve bugfixes after another Stable version of Julia has been released. However, upgrading to the next Stable release will always be possible as each release of Julia v1.x will continue to run code written for earlier versions.

You may prefer the LTS (Long Term Support) version of Julia if you are looking for a very stable code base. The current LTS version of Julia is versioned according to SemVer as v1.0.x; this branch will continue to recieve bugfixes until a new LTS branch is chosen, at which point the v1.0.x series will no longer recieved regular bug fixes and all but the most conservative users will be advised to upgrade to the new LTS version series. As a package developer, you may prefer to develop for the LTS version, to maximize the number of users who can use your package. As per SemVer, code written for v1.0 will continue to work for all future LTS and Stable versions. In general, even if targetting the LTS, one can develop and run code in the latest Stable version, to take advantage of the improved performance; so long as one avoids using new features (such as added library functions or new methods).

You may prefer the nightly version of Julia if you want to take advantage of the latest updates to the language, and don't mind if the version available today occasionally doesn't actually work. As the name implies, releases to the nightly version are made roughly every night (depending on build infrastructure stability). In general nightly released are fairly safe to use—your code will not catch on fire. However, they may be occasional regressions and or issues that will not be found until more thorough pre-release testing. You may wish to test against the nightly version to ensure that such regressions that affect your use case are caught before a release is made.

Finally, you may also consider building Julia from source for yourself. This option is mainly for those individuals who are comfortable at the command line, or interested in learning. If this describes you, you may also be interested in reading our guidelines for contributing.

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。

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 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`。
- 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，没有小数点的数字字面量（例如 `42`）会创建整数而不是浮点数。也支持任意大整数字面量。因此，某些操作（如 `2^-1`）将抛出 domain error，因为结果不是整数（有关的详细信息，请参阅常见问题中有关 domain errors 的条目）。point numbers. As a result, some operations can throw a domain error if they expect a float; for example, `julia> a = -1; 2^a` throws a domain error, as the result is not an integer (see the FAQ entry on domain errors for details).
- 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。

- 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。

- 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...)。

- In Julia, the adjoint function performs conjugate transposition; in MATLAB, adjoint provides the "adjugate" or classical adjoint, which is the transpose of the matrix of cofactors.

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。
- 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 等代码块由 end 关键字终止。缩进级别并不像在 Python 中那么重要。is not significant as it is in Python. Unlike Python, Julia has no pass keyword.
- Strings are denoted by double quotation marks ("text") in Julia (with three double quotation marks for multi-line strings), whereas in Python they can be denoted either by single ('text') or double quotation marks ("text"). Single quotation marks are used for characters in Julia ('c').
- String concatenation is done with * in Julia, not + like in Python. Analogously, string repetition is done with ^, not *. Implicit string concatenation of string literals like in Python (e.g. 'ab' 'cd' == 'abcd') is not done in Julia.
- Python Lists—flexible but slow—correspond to the Julia Vector{Any} type or more generally Vector{T} where T is some non-concrete element type. "Fast" arrays like Numpy arrays that store elements in-place (i.e., dtype is np.float64, [('f1', np.uint64), ('f2', np.int32)], etc.) can be represented by Array{T} where T is a concrete, immutable element type. This includes built-in types like Float64, Int32, Int64 but also more complex types like Tuple{UInt64,Float64} and many user-defined types as well.
- 在 Julia 中，数组、字符串等的索引从 1 开始，而不是从 0 开始。
- Julia 的切片索引包含最后一个元素，这与 Python 不同。Julia 中的 a[2:3] 就是 Python 中的 a[1:3]。
- Julia 不支持负数索引。特别地，列表或数组的最后一个元素在 Julia 中使用 end 索引，而不像在 Python 中使用 -1。
- Julia requires end for indexing until the last element. x[1:] in Python is equivalent to x[2:end] in Julia.
- Julia's range indexing has the format of x[start:step:stop], whereas Python's format is x[start:(stop+1):step]. Hence, x[0:10:2] in Python is equivalent to x[1:2:10] in Julia. Similarly, x[::-1] in Python, which refers to the reversed array, is equivalent to x[end:-1:1] in Julia.
- In Julia, indexing a matrix with arrays like X[[1,2], [1,3]] refers to a sub-matrix that contains the intersections of the first and second rows with the first and third columns. In Python, X[[1,2], [1,3]] refers to a vector that contains the values of cell [1,1] and [2,3] in the matrix. X[[1,2], [1,3]] in Julia is equivalent with X[np.ix_([0,1],[0,2])] in Python. X[[0,1], [0,2]] in Python is equivalent with X[[CartesianIndex(1,1), CartesianIndex(2,3)]] in Julia.

- 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 数组是列优先的（Fortran 顺序），而 NumPy 数组是行优先（C 顺序）。为了在循环数组时获得最佳性能，循环顺序应该在 Julia 中相对于 NumPy 反转（请参阅 Performance Tips 中的对应章节）。

  be reversed in Julia relative to NumPy (see relevant section of Performance Tips).
- Julia 的更新运算符（例如 `+=`，`-=`，`···`）是 *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`。
- 每次调用方法时，Julia 都会计算函数参数的默认值，不像在 Python 中，默认值只会在函数定义时被计算一次。例如，每次无输入参数调用时，函数 `f(x=rand()) = x` 都返回一个新的随机数在另一方面，函数 `g(x=[1,2]) = push!(x,3)` 在每次以 `g()` 调用时返回 `[1,2,3]`。
- 在 Julia 中，`%` 是余数运算符，而在 Python 中是模运算符。
- In Julia, the commonly used `Int` type corresponds to the machine integer type (`Int32` or `Int64`), unlike in Python, where `int` is an arbitrary length integer. This means in Julia the `Int` type will overflow, such that `2^64 == 0`. If you need larger values use another appropriate type, such as `Int128`, `BigInt` or a floating point type like `Float64`.
- The imaginary unit `sqrt(-1)` is represented in Julia as `im`, not `j` as in Python.
- In Julia, the exponentiation operator is ^, not ** as in Python.
- Julia uses `nothing` of type `Nothing` to represent a null value, whereas Python uses `None` of type `NoneType`.
- In Julia, the standard operators over a matrix type are matrix operations, whereas, in Python, the standard operators are element-wise operations. When both `A` and `B` are matrices, `A * B` in Julia performs matrix multiplication, not element-wise multiplication as in Python. `A * B` in Julia is equivalent with `A @ B` in Python, whereas `A * B` in Python is equivalent with `A .* B` in Julia.
- The adjoint operator `'` in Julia returns an adjoint of a vector (a lazy representation of row vector), whereas the transpose operator `.T` over a vector in Python returns the original vector (non-op).
- In Julia, a function may contain multiple concrete implementations (called *Methods*), selected via multiple dispatch, whereas functions in Python have a single implementation (no polymorphism).
- There are no classes in Julia. Instead they are structures (mutable or immutable), containing data but no methods.
- Calling a method of a class in Python (`a = MyClass(x), x.func(y)`) corresponds to a function call in Julia, e.g. `a = MyStruct(x), func(x::MyStruct, y)`. In general, multiple dispatch is more flexible and powerful than the Python class system.
- Julia structures may have exactly one abstract supertype, whereas Python classes can inherit from one or more (abstract or concrete) superclasses.
- The logical Julia program structure (Packages and Modules) is independent of the file strucutre (`include` for additional files), whereas the Python code structure is defined by directories (Packages) and files (Modules).
- The ternary operator `x > 0 ? 1 : -1` in Julia corresponds to conditional expression in Python `1 if x > 0 else -1`.

- In Julia the `@` symbol refers to a macro, whereas in Python it refers to a decorator.
- Exception handling in Julia is done using `try`—`catch`—`finally`, instead of `try`—`except`—`finally`. In contrast to Python, it is not recommended to use exception handling as part of the normal workflow in Julia due to performance reasons.
- In Julia loops are fast, there is no need to write "vectorized" code for performance reasons.
- Be careful with non-constant global variables in Julia, especially in tight loops. Since you can write close-to-metal code in Julia (unlike Python), the effect of globals can be drastic (see Performance Tips).
- In Python, the majority of values can be used in logical contexts (e.g. `if "a":` means the following block is executed, and `if "":` means it is not). In Julia, you need explicit conversion to `Bool` (e.g. `if "a"` throws an exception). If you want to test for a non-empty string in Julia, you would explicitly write `if !isempty("")`.
- In Julia, a new local scope is introduced by most code blocks, including loops and `try`—`catch`—`finally`. Note that comprehensions (list, generator, etc.) introduce a new local scope both in Python and Julia, whereas `if` blocks do not introduce a new local scope in both languages.

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（依版本不同有所变动）。
- 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）。

  reversed in Julia relative to C/C++ (see relevant section of Performance Tips).
- 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。

- 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），in Julia)
- Julia 中有两个右移运算符，>> 和 >>>。>>> 执行逻辑移位，>> 总是执行算术移位（译注：此处原文为「>>> performs an arithmetic shift, >> always performs a logical shift」，疑误），与 C/C++ 不同，C/C++ 中的 >> 的含义依赖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）。

39.5 Noteworthy differences from Common Lisp

- Julia uses 1-based indexing for arrays by default, and it can also handle arbitrary index offsets.
- Functions and variables share the same namespace ("Lisp-1").
- There is a `Pair` type, but it is not meant to be used as a `COMMON-LISP:CONS`. Various iterable collections can be used interchangeably in most parts of the language (eg splatting, tuples, etc). `Tuples` are the closest to Common Lisp lists for *short* collections of heterogeneous elements. Use `NamedTuples` in place of alists. For larger collections of homogeneous types, `Arrays` and `Dicts` should be used.
- The typical Julia workflow for prototyping also uses continuous manipulation of the image, implemented with the Revise.jl package.
- Bignums are supported, but conversion is not automatic; ordinary integers overflow.
- Modules (namespaces) can be hierarchical. `import` and `using` have a dual role: they load the code and make it available in the namespace. `import` for only the module name is possible (roughly equivalent to `ASDF:LOAD-OP`). Slot names don't need to be exported separately. Global variables can't be assigned to from outside the module (except with `eval(mod, :(var = val))` as an escape hatch).
- Macros start with `@`, and are not as seamlessly integrated into the language as Common Lisp; consequently, macro usage is not as widespread as in the latter. A form of hygiene for macros is supported by the language. Because of the different surface syntax, there is no equivalent to `COMMON-LISP:&BODY`.
- *All* functions are generic and use multiple dispatch. Argument lists don't have to follow the same template, which leads to a powerful idiom (see `do`). Optional and keyword arguments are handled differently. Method ambiguities are not resolved like in the Common Lisp Object System, necessitating the definition of a more specific method for the intersection.
- Symbols do not belong to any package, and do not contain any values *per se*. `M.var` evaluates the symbol `var` in the module `M`.
- A functional programming style is fully supported by the language, including closures, but isn't always the idiomatic solution for Julia. Some workarounds may be necessary for performance when modifying captured variables.

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。

Part IV

Base

Chapter 41

基本功能

41.1 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

41.2 概览

`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-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.

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.

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.__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([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 1-argument 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.

source

`Base.MainInclude.include` – Function.

```
include(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 1-argument 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.

Use `Base.include` to evaluate a file into another module.

source

`Base.include_string` – Function.

```
include_string(m::Module, code::AbstractString, filename::AbstractString="string")
```

Like `include`, except reads code from the given string rather than from a file.

source

`Base.include_dependency` – Function.

```
include_dependency(path::AbstractString)
```

In a module, declare that the file specified by `path` (relative or absolute) is a dependency for precompilation; that is, the module will need to be recompiled if this file changes.

This is only needed if your module depends on a file that is not used via `include`. It has no effect outside of compilation.

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.

source

`Base.methods` – Function.

```
methods(f, [types])
```

Returns the method table for `f`.

If `types` is specified, returns an array of methods whose types match.

source

`Base.@show` – Macro.

```
@show
```

Show an expression and result, returning the result. See also `show`.

source

`ans` – Keyword.

```
ans
```

A variable referring to the last computed value, automatically set at the interactive prompt.

source

41.3 关键字

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. Also `in` and `isa` are parsed as infix operators. Creation of a variable named `where`, `in` or `isa` 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 exported 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 exported or not. See the manual section about modules for details.

source

baremodule – Keyword.

```
baremodule
```

baremodule declares a module that does not contain using Base or a definition of eval. It does still import Core.

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.

Examples

```
julia> macro sayhello(name)
           return :( println("Hello, ", $name, "!") )
       end
@sayhello (macro with 1 method)

julia> @sayhello "Charlie"
Hello, Charlie!
```

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 `;`.

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 into an array to represent the last index of a dimension.

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 statements allocate new variable bindings each time they run. Whereas an assignment modifies an existing value location, let creates new locations. This difference is only detectable in the case of variables that outlive their scope via closures. The let syntax accepts a comma-separated series of assignments and variable names:

```
let var1 = value1, var2, var3 = value3
    code
end
```

The assignments are evaluated in order, with each right-hand side evaluated in the scope before the new variable on the left-hand side has been introduced. Therefore it makes sense to write something like let x = x, since the two x variables are distinct and have separate storage.

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.

source

for – Keyword.

```
for
```

for loops repeatedly evaluate a block of statements while iterating over a sequence of values.

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

`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
```

`structs` 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. See the manual section on Composite Types for more information.

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 type that is 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 after new lines, but can be used to separate statements on a single line or to join statements into a single expression. ; is also used to suppress output printing in the REPL and similar interfaces.

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!"
```

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

41.4 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

41.5 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.

Experimental multithreading support.

source

Base.GC – Module.

```
Base.GC
```

Module with garbage collection utilities.

source

41.6 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)
```

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.

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.

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
```

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. `isless` is not defined on all pairs of values `(x, y)`. 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 in an arbitrary but consistent fashion. `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 `<`.

source

`Core.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 Float64
Stacktrace:
[...]
```

source

Core.typeof – Function.

```
typeof(x)
```

Get the concrete type of x.

Examples

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

julia> typeof(a)
Rational{Int64}

julia> M = [1 2; 3.5 4];

julia> typeof(M)
Array{Float64,2}
```

source

Core.tuple – Function.

```
tuple(xs...)
```

Construct a tuple of the given objects.

Examples

```
julia> tuple(1, 'a', pi)
(1, 'a', 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

Base.objectid – Function.

```
objectid(x)
```

Get a hash value for x based on object identity. objectid(x)==objectid(y) if x === y.

source

Base.hash – Function.

```
hash(x[, h::UInt])
```

Compute an integer hash code such that isequal(x,y) implies hash(x)==hash(y). The optional second argument h is a 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. Types supporting subtraction (operator -) should also implement widen, which is required to hash values inside heterogeneous arrays.

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 behavior of this function is unpredictable.

f must not cause a task switch, which excludes most I/O operations such as println. @schedule println("message") or ccall(:jl_, Void, (Any,), "message") may be helpful for debugging purposes.

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.

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.

As a special case, functions can only be actually deep-copied if they are anonymous, otherwise they are just copied. The difference is only relevant in the case of closures, i.e. functions which may contain hidden internal references.

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)
```

The syntax `a.b` calls `getproperty(a, :b)`.

Examples

```
julia> struct MyType
           x
       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
```

See also `propertynames` and `setproperty!`.

source

`Base.setproperty!` – Function.

```
setproperty!(value, name::Symbol, x)
```

The syntax `a.b = c` calls `setproperty!(a, :b, c)`.

See also `propertynames` and `getproperty`.

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" fieldnames intended for internal use, pass `true` for the optional second argument. REPL tab completion on `x.` shows only the `private=false` properties.

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.

source

Core.getfield – Function.

```
getfield(value, name::Symbol)
```

Extract a named field from a `value` of composite type. See also `getproperty`.

Examples

```
julia> a = 1//2
1//2

julia> getfield(a, :num)
1

julia> a.num
1
```

source

Core.setfield! – Function.

```
setfield!(value, name::Symbol, x)
```

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)`. 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.isdefined – Function.

```
isdefined(m::Module, s::Symbol)
isdefined(object, s::Symbol)
isdefined(object, index::Int)
```

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.

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

Base.@isdefined – Macro.

```
@isdefined s -> Bool
```

Tests whether variable s is defined in the current scope.

See also isdefined.

Examples

```
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 or Rational 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(Rational{Int32}, x)
1//3

julia> convert(Rational{Int64}, x)
6004799503160661//18014398509481984
```

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
```

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.

Examples

```
julia> promote(Int8(1), Float16(4.5), Float32(4.1))
(1.0f0, 4.5f0, 4.1f0)
```

source

Base.oftype – Function.

```
oftype(x, y)
```

Convert y to the type of x (`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.

Examples

```
julia> identity("Well, what did you expect?")
"Well, what did you expect?"
```

source

41.7 Properties of Types

Type relations

Base.supertype – Function.

```
supertype(T::DataType)
```

Return the supertype of DataType T.

Examples

```
julia> supertype(Int32)
Signed
```

source

Missing docstring.

Missing docstring for Core.Type. Check Documenter's build log for details.

Missing docstring.

Missing docstring for Core.DataType. Check Documenter's build log for details.

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 T and S, i.e. the narrowest type from which they both inherit.

source

Base.typeintersect – Function.

```
typeintersect(T, S)
```

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.

```
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
```

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.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

Missing docstring.

Missing docstring for `Base.ismutable`. Check Documenter's build log for details.

`Base.isimmutable` – Function.

```
isimmutable(v) -> Bool
```

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.isabstracttype` – Function.

```
isabstracttype(T)
```

Determine whether type T was declared as an abstract type (i.e. using the `abstract` keyword).

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` keyword).

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`.

Examples

```
julia> fieldnames(Rational)
(:num, :den)
```

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

Base.hasfield – Function.

```
hasfield(T::Type, name::Symbol)
```

Return a boolean indicating whether T has name as one of its own fields.

> **Julia 1.2**
> This function requires at least Julia 1.2.

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. Size, in bytes, of object obj if it is not DataType.

Examples

```
julia> sizeof(Float32)
4

julia> sizeof(ComplexF64)
16

julia> sizeof(1.0)
8

julia> sizeof([1.0:10.0;])
80
```

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`).

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.

Examples

```
julia> isbitstype(Complex{Float64})
true

julia> isbitstype(Complex)
false
```

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.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)
12-element Array{Tuple{UInt64,Symbol,DataType},1}:
 (0x0000000000000000, :device, UInt64)
 (0x0000000000000008, :inode, UInt64)
 (0x0000000000000010, :mode, UInt64)
 (0x0000000000000018, :nlink, Int64)
 (0x0000000000000020, :uid, UInt64)
 (0x0000000000000028, :gid, UInt64)
 (0x0000000000000030, :rdev, UInt64)
 (0x0000000000000038, :size, Int64)
 (0x0000000000000040, :blksize, Int64)
 (0x0000000000000048, :blocks, Int64)
 (0x0000000000000050, :mtime, Float64)
 (0x0000000000000058, :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.

Examples

```
julia> typemin(Float16)
-Inf16

julia> typemin(Float32)
-Inf32
```

source

Base.typemax – Function.

```
typemax(T)
```

The highest value representable by the given (real) numeric DataType.

Examples

```
julia> typemax(Int8)
127

julia> typemax(UInt32)
0xffffffff
```

source

Base.floatmin – Function.

```
floatmin(T)
```

The smallest in absolute value non-subnormal value representable by the given floating-point DataType T.

source

Base.floatmax – Function.

```
floatmax(T)
```

The highest finite value representable by the given floating-point DataType T.

Examples

```
julia> floatmax(Float16)
Float16(6.55e4)

julia> floatmax(Float32)
3.4028235f38
```

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

$$|y - x| \leq \operatorname{eps}(x)/2.$$

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

41.8 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 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"

julia> 1.0 :: IntOrString
ERROR: TypeError: in typeassert, expected Union{Int64, AbstractString}, got Float64
```

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...}
```

Tuples are an abstraction of the arguments of a function – without the function itself. The salient aspects of a function's arguments are their order and their types. Therefore a tuple type is similar to a parameterized immutable type where each parameter is the type of one field. Tuple types may have any number of parameters.

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.

See the manual section on Tuple Types.

source

Core.NamedTuple – Type.

```
NamedTuple
```

NamedTuples are, as their name suggests, named Tuples. That is, they're a tuple-like collection of values, where each entry has a unique name, represented as a Symbol. Like Tuples, NamedTuples are immutable; neither the names nor the values can be modified in place after construction.

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]. A tuple of the names can be obtained using keys, and a tuple of the values can be obtained using values.

> **Note**
>
> Iteration over NamedTuples produces the *values* without the names. (See example below.) To iterate over the name-value pairs, use the pairs function.

Examples

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

julia> x.a
1

julia> x[:a]
1

julia> keys(x)
(:a, :b)

julia> values(x)
(1, 2)

julia> collect(x)
2-element Array{Int64,1}:
 1
 2

julia> collect(pairs(x))
2-element Array{Pair{Symbol,Int64},1}:
 :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 a pair name::Symbol => value or splatting an iterator yielding such pairs after a semicolon inside a tuple literal:

```
julia> (; :a => 1)
(a = 1,)

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

```
julia> (; zip(keys, values)...)
(a = 1, b = 2, c = 3)
```

source

Missing docstring.

Missing docstring for `Base.@NamedTuple`. Check Documenter's build log for details.

`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. 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` – Type.

```
Vararg{T,N}
```

The last parameter of a tuple type `Tuple` can be the special type `Vararg`, which denotes any number of trailing elements. The type `Vararg{T,N}` corresponds to exactly `N` elements of type `T`. `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.)

Examples

```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString,Vararg{Int64,N} where N}

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.

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.

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, y...)
```

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.

Examples

```
julia> something(nothing, 1)
1

julia> something(Some(1), nothing)
1

julia> something(missing, nothing)
missing

julia> something(nothing, nothing)
ERROR: ArgumentError: No value arguments present
```

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.

To list all the instances of an enum use `instances`, e.g.

```
julia> instances(Fruit)
(apple, orange, kiwi)
```

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 should be compared using ===. The implementation re-uses the same object for all Symbols with the same name, so comparison tends to be efficient (it can just compare pointers).

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.

source

41.9 Generic Functions

`Core.Function` – Type.

```
Function
```

Abstract type of all functions.

Examples

```
julia> isa(+, Function)
true

julia> typeof(sin)
typeof(sin)

julia> ans <: Function
true
```

source

`Base.hasmethod` – Function.

```
hasmethod(f, t::Type{<:Tuple}[, kwnames]; world=typemax(UInt)) -> 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> hasmethod(sum, Tuple{Function, Array}, (:dims,))
true

julia> hasmethod(sum, Tuple{Function, Array}, (: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

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).

Examples

```
julia> f(x::Real) = x^2;

julia> f(x::Integer) = 1 + invoke(f, Tuple{Real}, x);

julia> f(2)
5
```

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.)

source

`new` – Keyword.

```
new
```

Special function available to inner constructors which created a new object of the type. See the manual section on Inner Constructor Methods for more information.

source

`Base.:|>` – Function.

```
|>(x, f)
```

Applies a function to the preceding argument. This allows for easy function chaining.

Examples

```
julia> [1:5;] |> x->x.^2 |> sum |> inv
0.01818181818181818
```

source

`Base.:∘` – Function.

```
f ∘ g
```

Compose functions: i.e. `(f ∘ g)(args...)` means `f(g(args...))`. The `∘` symbol can be entered in the Julia REPL (and most editors, appropriately configured) by typing `\circ<tab>`.

Examples

```
julia> map(uppercase∘first, ["apple", "banana", "carrot"])
3-element Array{Char,1}:
 'A'
 'B'
 'C'
```

source

41.10 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 using `include`, evaluate all expressions, and return the value of the last one.

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 = 1:length(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.

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 `size`. 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. This is shown in the following example:

```
@inline function bigfunction(x)
    #=
        Function Definition
    =#
end
```

source

Base.@noinline – Macro.

```
@noinline
```

Prevents the compiler from inlining a function.

Small functions are typically inlined automatically. By using `@noinline` on small functions, auto-inlining can be prevented. This is shown in the following example:

```
@noinline function smallfunction(x)
    #=
        Function Definition
    =#
end
```

source

Base.@nospecialize – Macro.

```
@nospecialize
```

Applied to a function argument name, hints to the compiler that the method should not be specialized for different types of that argument, but instead to use precisely the declared type for each argument. This is only a hint for avoiding excess code generation. 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. 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(@nospecialize(x = 1), y)
    ...
end

function example_function(x, y, z)
```

```
    @nospecialize x y
    ...
end

@nospecialize
f(y) = [x for x in y]
@specialize
```

source

Base.@specialize – Macro.

```
@specialize
```

Reset the specialization hint for an argument back to the default. For details, see @nospecialize.

source

Base.gensym – Function.

```
gensym([tag])
```

Generates a symbol which will not conflict with other variable names.

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, 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(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.

Example:

```
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.@pure – Macro.

```
@pure ex
@pure(ex)
```

`@pure` gives the compiler a hint for the definition of a pure function, helping for type inference.

A pure function can only depend on immutable information. This also means a `@pure` function cannot use any global mutable state, including generic functions. Calls to generic functions depend on method tables which are mutable global state. Use with caution, incorrect `@pure` annotation of a function may introduce hard to identify bugs. Double check for calls to generic functions.

source

Base.@deprecate – Macro.

```
@deprecate old new [ex=true]
```

The first argument `old` is the signature of the deprecated method, the second one `new` is the call which replaces it. `@deprecate` exports `old` unless the optional third argument is `false`.

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)
```

source

41.11 Missing Values

Base.Missing – Type.

```
Missing
```

A type with no fields whose singleton instance `missing` is used to represent missing values.

source

Base.missing – Constant.

```
missing
```

The singleton instance of type `Missing` representing a missing value.

source

Base.coalesce – Function.

```
coalesce(x, y...)
```

Return the first value in the arguments which is not equal to `missing`, if any. Otherwise return `missing`.

Examples

```
julia> coalesce(missing, 1)
1

julia> coalesce(1, missing)
1

julia> coalesce(nothing, 1)  # returns `nothing`

julia> coalesce(missing, missing)
missing
```

source

Base.ismissing – Function.

```
ismissing(x)
```

Indicate whether x is `missing`.

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.

Examples

```
julia> x = skipmissing([1, missing, 2])
Base.SkipMissing{Array{Union{Missing, Int64},1}}(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 Array{Int64,1}:
 1
 3

julia> collect(skipmissing([1, missing, 2]))
2-element Array{Int64,1}:
 1
 2

julia> collect(skipmissing([1 missing; 2 missing]))
2-element Array{Int64,1}:
 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

41.12 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).

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=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)
```

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, or `nothing`. In order to modify (rather than replace) the existing environment, create `env` by `copy(ENV)` and then set `env["var"]=val` as desired.
- `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. Normally, to create a `Cmd` object in the first place, one uses backticks, 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` by `copy(ENV)` and then setting `env["var"]=val` as desired, or use `withenv`.

The `dir` keyword argument can be used to specify a working directory for the command.

source

Base.withenv – Function.

```
withenv(f::Function, 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.

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() -> AbstractString
```

Get the local machine's host name.

source

`Base.Libc.getpid` – Function.

```
getpid(process) -> Int32
```

Get the child process ID, if it still exists.

> **Julia 1.1**
> This function requires at least Julia 1.1.

source

```
getpid() -> Int32
```

Get Julia's process ID.

source

`Base.Libc.time` – Method.

```
time()
```

Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution.

source

Base.time_ns – Function.

```
time_ns()
```

Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

source

Base.@time – Macro.

```
@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.

```
julia> @time rand(10^6);
  0.001525 seconds (7 allocations: 7.630 MiB)

julia> @time begin
           sleep(0.3)
           1+1
       end
  0.301395 seconds (8 allocations: 336 bytes)
2
```

source

Base.@timev – Macro.

```
@timev
```

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.

See also @time, @timed, @elapsed, and @allocated.

```
julia> @timev rand(10^6);
  0.001006 seconds (7 allocations: 7.630 MiB)
elapsed time (ns): 1005567
bytes allocated:   8000256
pool allocs:       6
malloc() calls:    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.

See also @time, @timev, @elapsed, and @allocated.

```
julia> val, t, bytes, gctime, memallocs = @timed rand(10^6);

julia> t
0.006634834

julia> bytes
8000256

julia> gctime
0.0055765

julia> fieldnames(typeof(memallocs))
(:allocd, :malloc, :realloc, :poolalloc, :bigalloc, :freecall, :total_time, :pause, :full_sweep)

julia> memallocs.total_time
5576500
```

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.

See also @time, @timev, @timed, and @allocated.

```
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. Note: the expression is evaluated inside a local function, instead of the current context, in order to eliminate the effects of compilation, however, there still may be some allocations due to JIT compilation. This also makes the results inconsistent with the `@time` macros, which do not try to adjust for the effects of compilation.

See also @time, @timev, @timed, and @elapsed.

```
julia> @allocated rand(10^6)
8000080
```

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.)

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 kilobytes.

source

Base.Sys.total_memory – Function.

```
Sys.total_memory()
```

Get the total memory in RAM (including that which is currently used) in kilobytes.

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

41.13 Versioning

Base.VersionNumber – Type.

```
VersionNumber
```

Version number type which follow the specifications of semantic versioning, composed of major, minor and patch numeric values, followed by pre-release and build alpha-numeric annotations. See also `@v_str`.

Examples

```
julia> VersionNumber("1.2.3")
v"1.2.3"

julia> VersionNumber("2.0.1-rc1")
v"2.0.1-rc1"
```

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

41.14 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.

source

`Base.rethrow` – Function.

```
rethrow([e])
```

Throw an object without changing the current exception backtrace. The default argument is the current exception (if called within a `catch` block).

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.catch_stack` – Function.

```
catch_stack(task=current_task(); [inclue_bt=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 a Vector of `(exception,backtrace)` pairs, or a Vector of exceptions if `include_bt` is false.

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 1.1**
>
> This function is experimental in Julia 1.1 and will likely be renamed in a future release (see https://github.com/JuliaLang/julia/pull/29901).

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

Missing docstring.

Missing docstring for `Base.Experimental.register_error_hint`. Check Documenter's build log for details.

Missing docstring.

Missing docstring for `Base.Experimental.show_error_hints`. Check Documenter's build log for details.

`Core.ArgumentError` – Type.

```
ArgumentError(msg)
```

The parameters to a function call do not match a valid signature. Argument `msg` is a descriptive error string.

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 Array{Float64,1} at index [8]
Stacktrace:
 [1] getindex(::Array{Float64,1}, ::Int64) at ./array.jl:660
 [2] top-level scope

julia> B = fill(1.0, (2,3));

julia> B[2, 4]
ERROR: BoundsError: attempt to access 2×3 Array{Float64,2} at index [2, 4]
Stacktrace:
```

```
 [1] getindex(::Array{Float64,2}, ::Int64, ::Int64) at ./array.jl:661
 [2] top-level scope

julia> B[9]
ERROR: BoundsError: attempt to access 2×3 Array{Float64,2} at index [9]
Stacktrace:
 [1] getindex(::Array{Float64,2}, ::Int64) at ./array.jl:660
 [2] top-level scope
```

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 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).

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.

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

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

41.15 Events

Base.Timer – Method.

```
Timer(callback::Function, delay; interval = 0)
```

Create a timer that wakes up tasks waiting for it (by calling `wait` on the timer object) and calls the function `callback`.

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 timer is only triggered once. The function `callback` is called with a single argument, the timer itself. When the timer is closed (by `close` waiting tasks are woken with an error. Use `isopen` to check whether a timer is still active.

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 `delay` seconds, and then repeating with the given `interval` in seconds. 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.

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.

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

41.16 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.

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 a given definition of a generic function.

source

`Base.pathof` – Method.

```
pathof(m::Module)
```

Return the path of `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.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

`Base.@__MODULE__` – Macro.

```
@__MODULE__ -> Module
```

Get the `Module` of the toplevel eval, which is the `Module` code is currently being read from.

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.

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`.

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)
```

Returns a tuple `(filename,line)` giving the location of a generic `Function` definition.

source

Base.functionloc – Method.

```
functionloc(m::Method)
```

Returns a tuple `(filename,line)` giving the location of a `Method` definition.

source

41.17 Internals

Base.GC.gc – Function.

```
GC.gc()
```

Perform garbage collection.

> **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
```

Temporarily protect the given objects from being garbage collected, even if they would otherwise be unreferenced.

The last argument is the expression during which the object(s) will be preserved. The previous arguments are the objects to preserve.

source

Missing docstring.

Missing docstring for `Base.GC.safepoint`. Check Documenter's build log for details.

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)
```

Parse the expression string and return an expression (which could later be passed to eval for execution). `start` is the index of the first character to start parsing. 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.

```
julia> Meta.parse("x = 3, y = 5", 7)
(:(y = 5), 13)

julia> Meta.parse("x = 3, y = 5", 5)
(:((3, y) = 5), 13)
```

source

Base.Meta.parse – Method.

```
parse(str; raise=true, depwarn=true)
```

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.

```
julia> Meta.parse("x = 3")
:(x = 3)

julia> Meta.parse("x = ")
:($(Expr(:incomplete, "incomplete: premature end of input")))

julia> Meta.parse("1.0.2")
ERROR: Base.Meta.ParseError("invalid numeric constant \"1.0.\"")
Stacktrace:
[...]

julia> Meta.parse("1.0.2"; raise = false)
:($(Expr(:error, "invalid numeric constant \"1.0.\"")))
```

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; optimize=true, debuginfo=:default)
```

Returns an array of type-inferred lowered form (IR) for the methods matching the given generic function and type signature. The keyword argument `optimize` controls whether additional optimizations, such as inlining, are also applied. The keyword `debuginfo` controls the amount of code metadata present in the output, possible options are `:source` or `:none`.

source

Base.precompile – Function.

```
precompile(f, args::Tuple{Vararg{Any}})
```

Compile the given function `f` for the argument tuple (of types) `args`, but do not execute it.

source

41.18 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.Meta.isexpr – Function.

```
Meta.isexpr(ex, head[, n])::Bool
```

Check if ex is an expression with head head and n arguments.

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.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 42

集合和数据结构

42.1 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

42.2 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 > 1 and a, b, and c all integers creates a StepRange.

Examples

```
julia> collect(StepRange(1, Int8(2), 10))
5-element Array{Int64,1}:
 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 `Integers` creates a `UnitRange`.

Examples

```
julia> collect(UnitRange(2.3, 5.2))
3-element Array{Float64,1}:
 2.3
 3.3
 4.3

julia> typeof(1:10)
UnitRange{Int64}
```

source

Base.LinRange – Type.

```
LinRange{T}
```

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 `Int`.

Examples

```
julia> LinRange(1.5, 5.5, 9)
9-element LinRange{Float64}:
 1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5
```

source

42.3 通用集合

Base.isempty – Function.

```
isempty(collection) -> Bool
```

Determine whether a collection is empty (has no elements).

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} with 0 entries
```

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.

Examples

```
julia> length(1:5)
5

julia> length([1, 2, 3, 4])
4

julia> length([1 2; 3 4])
4
```

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

42.4 可迭代集合

Base.in – Function.

```
in(item, collection) -> Bool
∈(item, collection) -> Bool
∋(collection, item) -> 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. Returns 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, Sets check whether the item isequal to one of the elements. Dicts 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 these collections, the result is always a Bool and never missing.

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
```

source

Base.:∉ – Function.

```
∉(item, collection) -> Bool
∌(collection, item) -> Bool
```

Negation of ∈ and ∋, i.e. checks that `item` is not in `collection`.

Examples

```
julia> 1 ∉ 2:4
true

julia> 1 ∉ 1:3
false
```

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.

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`.

Examples

```
julia> a = ['a', 'b', 'c', 'b', 'd', 'a'];

julia> b = ['a', 'b', 'c'];

julia> indexin(a, b)
6-element Array{Union{Nothing, Int64},1}:
 1
 2
 3
 2
```

```
 nothing
 1

julia> indexin(b, a)
3-element Array{Union{Nothing, Int64},1}:
 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.

Examples

```
julia> unique([1, 2, 6, 2])
3-element Array{Int64,1}:
 1
 2
 6

julia> unique(Real[1, 1.0, 2])
2-element Array{Real,1}:
 1
 2
```

source

```
unique(f, itr)
```

Returns 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 Array{Int64,1}:
 1
 3
 4
```

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 Array{Bool,1}:
 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 Array{Int64,1}:
 1
 3
 4

julia> unique!(n -> n%3, [5, 1, 8, 9, 3, 4, 10, 7, 2, 6])
3-element Array{Int64,1}:
 5
 1
 9
```

```
julia> unique!(iseven, [2, 3, 5, 7, 9])
2-element Array{Int64,1}:
 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 Array{Int64,1}:
 1

julia> A = [7, 3, 2, 3, 7, 5];

julia> unique!(A)
4-element Array{Int64,1}:
 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 Array{Int64,1}:
  6
  7
 42
```

source

Base.allunique – Function.

```
allunique(itr) -> Bool
```

Return true if all values from itr are distinct when compared with isequal.

Examples

```
julia> a = [1; 2; 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> allunique([a, a])
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)`.

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.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.

Examples

```
julia> foldl(=>, 1:4)
((1 => 2) => 3) => 4

julia> foldl(=>, 1:4; init=0)
(((0 => 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)
```

Returns the largest result of calling function `f` on each element of `itr`.

Examples

```
julia> maximum(length, ["Julion", "Julia", "Jule"])
6
```

source

```
maximum(itr)
```

Returns the largest element in a collection.

Examples

```
julia> maximum(-20.5:10)
9.5

julia> maximum([1,2,3])
3
```

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)`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> maximum(A, dims=1)
1×2 Array{Int64,2}:
 3  4

julia> maximum(A, dims=2)
2×1 Array{Int64,2}:
 2
 4
```

source

`Base.maximum!` – Function.

```
maximum!(r, A)
```

Compute the maximum value of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> maximum!([1; 1], A)
2-element Array{Int64,1}:
 2
 4

julia> maximum!([1 1], A)
1×2 Array{Int64,2}:
 3  4
```

source

`Base.minimum` – Function.

```
minimum(f, itr)
```

Returns the smallest result of calling function `f` on each element of `itr`.

Examples

```
julia> minimum(length, ["Julion", "Julia", "Jule"])
4
```

source

```
minimum(itr)
```

Returns the smallest element in a collection.

Examples

```
julia> minimum(-20.5:10)
-20.5

julia> minimum([1,2,3])
1
```

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)`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> minimum(A, dims=1)
1×2 Array{Int64,2}:
 1  2

julia> minimum(A, dims=2)
2×1 Array{Int64,2}:
 1
 3
```

source

Base.minimum! – Function.

```
minimum!(r, A)
```

Compute the minimum value of A over the singleton dimensions of r, and write results to r.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> minimum!([1; 1], A)
2-element Array{Int64,1}:
 1
 3

julia> minimum!([1 1], A)
1×2 Array{Int64,2}:
 1  2
```

source

Base.extrema – Function.

```
extrema(itr) -> Tuple
```

Compute both the minimum and maximum element in a single pass, and return them as a 2-tuple.

Examples

```
julia> extrema(2:10)
(2, 10)

julia> extrema([9,pi,4.5])
(3.141592653589793, 9.0)
```

source

```
extrema(f, itr) -> Tuple
```

Compute both the minimum and maximum of `f` applied to each element in `itr` and return them as a 2-tuple. Only one pass is made over `itr`.

> **Julia 1.2**
> This method requires Julia 1.2 or later.

Examples

```
julia> extrema(sin, 0:π)
(0.0, 0.9092974268256817)
```

source

```
extrema(A::AbstractArray; dims) -> Array{Tuple}
```

Compute the minimum and maximum elements of an array over the given dimensions.

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.argmax` – Function.

```
argmax(itr) -> Integer
```

Return the index of the maximum element in a collection. If there are multiple maximal elements, then the first one will be returned.

The collection must not be empty.

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.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> argmax(A, dims=1)
1×2 Array{CartesianIndex{2},2}:
 CartesianIndex(2, 1)  CartesianIndex(2, 2)

julia> argmax(A, dims=2)
2×1 Array{CartesianIndex{2},2}:
 CartesianIndex(1, 2)
 CartesianIndex(2, 2)
```

source

Base.argmin – Function.

```
argmin(itr) -> Integer
```

Return the index of the minimum element in a collection. If there are multiple minimal elements, then the first one will be returned.

The collection must not be empty.

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.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> argmin(A, dims=1)
1×2 Array{CartesianIndex{2},2}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)

julia> argmin(A, dims=2)
2×1 Array{CartesianIndex{2},2}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)
```

source

Base.findmax – Function.

```
findmax(itr) -> (x, index)
```

Return the maximum element of the collection `itr` and its index. If there are multiple maximal elements, then the first one will be returned. If any data element is NaN, this element is returned. The result is in line with max.

The collection must not be empty.

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.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 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

Base.findmin – Function.

```
findmin(itr) -> (x, index)
```

Return the minimum element of the collection `itr` and its index. If there are multiple minimal elements, then the first one will be returned. If any data element is NaN, this element is returned. The result is in line with `min`.

The collection must not be empty.

Examples

```
julia> findmin([8,0.1,-9,pi])
(-9.0, 3)

julia> findmin([7,1,1,6])
(1, 2)

julia> findmin([7,1,1,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.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 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

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.

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.

source

Base.sum – Function.

```
sum(f, itr)
```

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.

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)
```

Returns 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.

Examples

```
julia> sum(1:20)
210
```

source

```
sum(A::AbstractArray; dims)
```

Sum elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> sum(A, dims=1)
1×2 Array{Int64,2}:
 4  6

julia> sum(A, dims=2)
2×1 Array{Int64,2}:
 3
 7
```

source

Base.sum! – Function.

```
sum!(r, A)
```

Sum elements of A over the singleton dimensions of r, and write results to r.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> sum!([1; 1], A)
2-element Array{Int64,1}:
 3
 7

julia> sum!([1 1], A)
1×2 Array{Int64,2}:
 4  6
```

source

Base.prod – Function.

```
prod(f, itr)
```

Returns 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.

Examples

```
julia> prod(abs2, [2; 3; 4])
576
```

source

```
prod(itr)
```

Returns 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.

Examples

```
julia> prod(1:20)
2432902008176640000
```

source

```
prod(A::AbstractArray; dims)
```

Multiply elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> prod(A, dims=1)
1×2 Array{Int64,2}:
 3  8

julia> prod(A, dims=2)
2×1 Array{Int64,2}:
  2
 12
```

source

`Base.prod!` – Function.

```
prod!(r, A)
```

Multiply elements of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> prod!([1; 1], A)
2-element Array{Int64,1}:
  2
 12

julia> prod!([1 1], A)
1×2 Array{Int64,2}:
 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).

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> a = [true,false,false,true]
4-element Array{Bool,1}:
 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).

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.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
 1  0
 1  0

julia> any!([1; 1], A)
2-element Array{Int64,1}:
 1
 1

julia> any!([1 1], A)
1×2 Array{Int64,2}:
 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).

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> a = [true,false,false,true]
4-element Array{Bool,1}:
 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).

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.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
 1  0
 1  0
```

```
julia> all!([1; 1], A)
2-element Array{Int64,1}:
 0
 0

julia> all!([1 1], A)
1×2 Array{Int64,2}:
 1  0
```

source

Base.count – Function.

```
count(p, itr) -> Integer
count(itr) -> Integer
```

Count the number of elements in `itr` for which predicate `p` returns `true`. If `p` is omitted, counts the number of `true` elements in `itr` (which should be a collection of boolean values).

Examples

```
julia> count(i->(4<=i<=6), [2,3,4,5,6])
3

julia> count([true, false, true, true])
3
```

source

```
count(
    pattern::Union{AbstractString,Regex},
    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.

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).

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.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).

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.foreach – Function.

```
foreach(f, c...) -> Nothing
```

Call function f on each element of iterable c. For multiple iterable arguments, f is called element-wise. foreach should be used instead of map when the results of f are not needed, for example in foreach(println, array).

Examples

```
julia> a = 1:3:7;

julia> foreach(x -> println(x^2), a)
1
16
49
```

source

Base.map – Function.

```
map(f, c...) -> collection
```

Transform collection c by applying f to each element. For multiple collection arguments, apply f elementwise.

See also: mapslices

Examples

```
julia> map(x -> x * 2, [1, 2, 3])
3-element Array{Int64,1}:
 2
 4
 6

julia> map(+, [1, 2, 3], [10, 20, 30])
3-element Array{Int64,1}:
 11
 22
 33
```

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 first collection.

Examples

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

julia> map!(x -> x * 2, a, [1, 2, 3]);

julia> a
3-element Array{Float64,1}:
 2.0
 4.0
 6.0
```

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 key type of `dict` then it will be converted to the key type if possible and otherwise raise an error.

Examples

“‘jldoctest julia> d = Dict(:a => 1, :b => 2) Dict{Symbol,Int64} with 2 entries: :a => 1 :b => 2

julia> map!(v -> v-1, values(d)) Dict{Symbol,Int64} with 2 entries: :a => 0 :b => 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.

Examples

```
julia> first(2:2:10)
2

julia> first([1; 2; 3; 4])
1
```

source

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first n characters of s.

```
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.

Examples

```
julia> last(1:2:10)
9

julia> last([1; 2; 3; 4])
4
```

source

```
last(s::AbstractString, n::Integer)
```

Get a string consisting of the last n characters of s.

```
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.

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.

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 `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.

Examples

```
julia> collect(1:2:13)
7-element Array{Int64,1}:
  1
  3
  5
  7
  9
 11
 13
```

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 Array{Float64,1}:
 1.0
 3.0
 5.0
```

source

`Base.filter` – Function.

```
filter(f, a::AbstractArray)
```

Return a copy of `a`, removing elements for which `f` is `false`. The function `f` is passed one argument.

Examples

```
julia> a = 1:10
1:10

julia> filter(isodd, a)
5-element Array{Int64,1}:
 1
 3
 5
 7
 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 Array{Union{Missing, Int64},2}:
 1         2
  missing  4

julia> filter(isodd, skipmissing(x))
1-element Array{Int64,1}:
 1
```

source

Base.filter! – Function.

```
filter!(f, a::AbstractVector)
```

Update a, removing elements for which f is false. The function f is passed one argument.

Examples

```
julia> filter!(isodd, Vector(1:10))
5-element Array{Int64,1}:
 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!.

Examples

```
julia> replace([1, 2, 1, 3], 1=>0, 2=>4, count=2)
4-element Array{Int64,1}:
 0
 4
 1
 3

julia> replace([1, missing], missing=>0)
2-element Array{Int64,1}:
 1
 0
```

source

Base.replace – Method.

```
replace(new::Function, 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).

Examples

```
julia> replace(x -> isodd(x) ? 2x : x, [1, 2, 3, 4])
4-element Array{Int64,1}:
 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 Array{Int64,1}:
 0
 4
 1
 3

julia> replace!(Set([1, 2, 3]), 1=>0)
Set([0, 2, 3])
```

source

```
replace!(new::Function, 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 Array{Int64,1}:
 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([6, 12])
```

source

42.5 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, ...)`.

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)`.

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`.

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.

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

42.6 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.

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 object-id as hash and === as equality with keys of type K and values of type V.

See Dict for further help.

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.

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 Immutable Dictionary for the 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` – Method.

```
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.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> get(d, "a", 3)
1

julia> get(d, "c", 3)
3
```

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.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> get(d, "a", 3)
1

julia> get(d, "c", 3)
3
```

source

```
get(f::Function, 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! – Method.

```
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

Base.get! – Method.

```
get!(f::Function, 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:

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

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, 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
```

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)
Base.ValueIterator for a Dict{String,Int64} with 2 entries. Values:
  2
  1

julia> values([2])
1-element Array{Int64,1}:
 2
```

source

```
values(a::AbstractDict)
```

Return an iterator over all values in a collection. `collect(values(a))` returns an array of values. Since the values are stored internally in a hash table, the order in which they are returned may vary. But `keys(a)` and `values(a)` both 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 Array{Int64,1}:
 2
 3
```

source

`Base.pairs` – Function.

```
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.

source

```
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 `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

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.

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(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.

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" => 4753.0
  "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

Missing docstring.

Missing docstring for `Base.mergewith`. Check Documenter's build log for details.

`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

```
merge!(combine, d::AbstractDict, others::AbstractDict...)
```

Update collection with pairs from the other collections. Values with the same key will be combined using the combiner function.

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 => 6

julia> merge!(-, d1, d1);

julia> d1
Dict{Int64,Int64} with 3 entries:
  4 => 0
  3 => 0
  1 => 0
```

source

Missing docstring.

Missing docstring for `Base.mergewith!`. Check Documenter's build log for details.

Base.sizehint! – Function.

```
sizehint!(s, n)
```

Suggest that collection s reserve capacity for at least n elements. This can improve performance.

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 an 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 an 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

42.7 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([itr])
```

Construct a Set of the values generated by the given iterable object, or an empty set. Should be used instead of BitSet for sparse integer sets, or for sets of arbitrary objects.

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 the union of sets. Maintain order with arrays.

Examples

```
julia> union([1, 2], [3, 4])
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> union([1, 2], [2, 4])
3-element Array{Int64,1}:
 1
 2
 4

julia> union([4, 2], 1:2)
3-element Array{Int64,1}:
 4
 2
 1

julia> union(Set([1, 2]), 2:3)
Set([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.

Examples

```
julia> a = Set([1, 3, 4, 5]);

julia> union!(a, 1:2:8);

julia> a
Set([7, 4, 3, 5, 1])
```

source

Base.intersect – Function.

```
intersect(s, itrs...)
∩(s, itrs...)
```

Construct the intersection of sets. Maintain order with arrays.

Examples

```
julia> intersect([1, 2, 3], [3, 4, 5])
1-element Array{Int64,1}:
 3

julia> intersect([1, 4, 4, 5, 6], [4, 6, 6, 7, 8])
2-element Array{Int64,1}:
 4
 6

julia> intersect(Set([1, 2]), BitSet([2, 3]))
Set([2])
```

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.

Examples

```
julia> setdiff([1,2,3], [3,4,5])
2-element Array{Int64,1}:
 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.

Examples

```
julia> a = Set([1, 3, 4, 5]);

julia> setdiff!(a, 1:2:6);

julia> a
Set([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. Note that in this case the multiplicity of elements matters.

Examples

```
julia> symdiff([1,2,3], [3,4,5], [4,5,6])
3-element Array{Int64,1}:
 1
 2
 6

julia> symdiff([1,2,1], [2, 1, 2])
2-element Array{Int64,1}:
 1
 2

julia> symdiff(unique([1,2,1]), unique([2, 1, 2]))
0-element Array{Int64,1}
```

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.

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.

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.

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.

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.

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.

Examples

```
julia> issetequal([1, 2], [1, 2, 3])
false

julia> issetequal([1, 2], [2, 1])
true
```

source

Missing docstring.

Missing docstring for Base.isdisjoint. Check Documenter's build log for details.

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

42.8 双端队列

Base.push! – Function.

```
push!(collection, items...) -> collection
```

Insert one or more items at the end of collection.

Examples

```
julia> push!([1, 2, 3], 4, 5, 6)
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

Use append! to add all the elements of another collection to collection. The result of the preceding example is equivalent to append!([1, 2, 3], [4, 5, 6]).

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.

Examples

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

julia> pop!(A)
3

julia> A
2-element Array{Int64,1}:
 1
 2

julia> S = Set([1, 2])
Set([2, 1])

julia> pop!(S)
2

julia> S
Set([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

Missing docstring.

Missing docstring for `Base.popat!`. Check Documenter's build log for details.

`Base.pushfirst!` – Function.

```
pushfirst!(collection, items...) -> collection
```

Insert one or more `items` at the beginning of `collection`.

Examples

```
julia> pushfirst!([1, 2, 3, 4], 5, 6)
6-element Array{Int64,1}:
 5
 6
 1
 2
 3
 4
```

source

`Base.popfirst!` – Function.

```
popfirst!(collection) -> item
```

Remove the first `item` from `collection`.

Examples

```
julia> A = [1, 2, 3, 4, 5, 6]
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6

julia> popfirst!(A)
1

julia> A
5-element Array{Int64,1}:
 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`.

Examples

```
julia> insert!([6, 5, 4, 2, 1], 4, 3)
6-element Array{Int64,1}:
 6
 5
 4
 3
 2
 1
```

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.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 2)
5-element Array{Int64,1}:
 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 Array{Int64,1}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], [true, false, true, false, true, false])
3-element Array{Int64,1}:
 5
 3
 1

julia> deleteat!([6, 5, 4, 3, 2, 1], (2, 2))
ERROR: ArgumentError: indices must be unique and sorted
Stacktrace:
[...]
```

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.

Examples

```
julia> A = [6, 5, 4, 3, 2, 1]; splice!(A, 5)
2

julia> A
5-element Array{Int64,1}:
 6
 5
 4
 3
 1

julia> splice!(A, 5, -1)
1

julia> A
5-element Array{Int64,1}:
```

```
 6
 5
 4
 3
-1

julia> splice!(A, 1, [-1, -2, -3])
6

julia> A
7-element Array{Int64,1}:
 -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, range, [replacement]) -> items
```

Remove items in the specified index range, and return a collection containing the removed items. 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 items.

To insert `replacement` before an index `n` without removing any items, use `splice!(collection, n:n-1, replacement)`.

Examples

```
julia> A = [-1, -2, -3, 5, 4, 3, -1]; splice!(A, 4:3, 2)
0-element Array{Int64,1}

julia> A
8-element Array{Int64,1}:
 -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 Array{Int64,1}:
 6
 5
 4

julia> a = resize!([6, 5, 4, 3, 2, 1], 8);

julia> length(a)
8

julia> a[1:6]
6-element Array{Int64,1}:
 6
 5
 4
 3
 2
 1
```

source

Base.append! – Function.

```
append!(collection, collection2) -> collection.
```

Add the elements of collection2 to the end of collection.

Examples

```
julia> append!([1],[2,3])
3-element Array{Int64,1}:
 1
 2
 3

julia> append!([1, 2, 3], [4, 5, 6])
6-element Array{Int64,1}:
 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).

source

Base.prepend! – Function.

```
prepend!(a::Vector, items) -> collection
```

Insert the elements of items to the beginning of a.

Examples

```
julia> prepend!([3],[1,2])
3-element Array{Int64,1}:
 1
 2
 3
```

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`)

42.9 集合相关的实用工具

`Base.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
```

source

`Base.Iterators.Pairs` – Type.

```
Iterators.Pairs(values, keys) <: AbstractDict{eltype(keys), eltype(values)}
```

Transforms an indexable container into an Dictionary-view of the same data. Modifying the key-space of the underlying data may invalidate this object.

source

Chapter 43

数学相关

43.1 数学运算符

Base.:- – Method.

```
-(x)
```

Unary minus operator.

Examples

```
julia> -1
-1

julia> -(2)
-2

julia> -[1 2; 3 4]
2×2 Array{Int64,2}:
 -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

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 Array{Float64,1}:
  6.5
 -7.0

julia> inv(A) * x
2-element Array{Float64,1}:
  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.)

```
julia> 3^5
243

julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> A^3
2×2 Array{Int64,2}:
 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

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 division. Computes x/y, truncated to an integer.

Examples

```
julia> 9 ÷ 4
2

julia> -5 ÷ 3
-1
```

source

Base.fld – Function.

```
fld(x, y)
```

Largest integer less than or equal to x/y.

Examples

```
julia> fld(7.3,5.5)
1.0
```

source

Base.cld – Function.

```
cld(x, y)
```

Smallest integer larger than or equal to x/y.

Examples

```
julia> cld(5.5,2.2)
3.0
```

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))
3

julia> mod(3, 0:2)
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.

```
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
```

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> 129 % Int8
-127
```

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.

Examples

```
julia> x = 15; y = 4;

julia> x % y
```

```
3

julia> x == div(x, y) * y + rem(x, y)
true
```

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.

Examples

```
julia> mod2pi(9*pi/4)
0.7853981633974481
```

source

Base.divrem – Function.

```
divrem(x, y)
```

The quotient and remainder from Euclidean division. Equivalent to `(div(x,y), rem(x,y))` or `(x÷y, x%y)`.

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. Equivalent to `(fld(x,y), mod(x,y))`.

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`.

See also: fld1, fldmod1.

Examples

```
julia> mod1(4, 2)
2

julia> mod1(4, 3)
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 `0`s. 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 >>, >>>.

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 BitArray{1}:
 1
 0
 1
 0
 0

julia> B << 1
5-element BitArray{1}:
 0
```

```
 1
 0
 0
 0

julia> B << -1
5-element BitArray{1}:
 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, where `n >= 0`, 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 BitArray{1}:
 1
```

```
 0
 1
 0
 0

julia> B >> 1
5-element BitArray{1}:
 0
 1
 0
 1
 0

julia> B >> -1
5-element BitArray{1}:
 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, where `n >= 0`, 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"
```

`BigInt`s 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

Missing docstring.

Missing docstring for `Base.bitrotate`. Check Documenter's build log for details.

Base.:: – Function.

```
(:)(I::CartesianIndex, J::CartesianIndex)
```

Construct CartesianIndices from two CartesianIndex.

> **Julia 1.1**
> This method requires at least Julia 1.1.

Examples

```
julia> I = CartesianIndex(2,1);

julia> J = CartesianIndex(3,3);

julia> I:J
2×3 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(2, 1)  CartesianIndex(2, 2)  CartesianIndex(2, 3)
 CartesianIndex(3, 1)  CartesianIndex(3, 2)  CartesianIndex(3, 3)
```

source

```
(:)(start, [step], stop)
```

Range operator. a:b constructs a range from a to b with a step size of 1 (a UnitRange) , and a:s:b is similar but uses a step size of s (a StepRange).

: is also used in indexing to select whole dimensions and for Symbol literals, as in e.g. :hello.

source

Base.range – Function.

```
range(start[, stop]; length, stop, step=1)
```

Given a starting value, construct a range either by length or from start to stop, optionally with a given step (defaults to 1, a UnitRange). One of length or stop is required. If length, stop, and step are all specified, they must agree.

If length and stop are provided and step is not, the step size will be computed automatically such that there are length linearly spaced elements in the range (a LinRange).

If step and stop are provided and length is not, the overall range length will be computed automatically such that the elements are step spaced (a StepRange).

stop may be specified as either a positional or keyword argument.

> **Julia 1.1**
> stop as a positional argument requires at least Julia 1.1.

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
```

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{T,R,S}(ref::R, step::S, len, [offset=1]) where {T,R,S}
StepRangeLen(       ref::R, step::S, len, [offset=1]) where {  R,S}
```

A range r where r[i] produces values of type T (in the second 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. In conjunction with TwicePrecision this can be used to implement ranges that are free of roundoff error.

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.

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.

source

```
==(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.:!= – 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 numeric 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. (x < y) | (x == y)

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.

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`.

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`.

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 BitArray{1}:
 0
 1
 0
```

source

`Base.:!` – Function.

```
!(x)
```

Boolean not. Implements three-valued logic, returning `missing` if x is `missing`.

Examples

```
julia> !true
false

julia> !false
true

julia> !missing
missing

julia> .![true false true]
1×3 BitArray{2}:
 0  1  0
```

source

```
!f::Function
```

Predicate function negation: when the argument of `!` is a function, it returns a function which computes the boolean negation of `f`.

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: |-| <  ⇒ |()-()| < "
```

source

&& – Keyword.

```
x && y
```

Short-circuiting boolean AND.

source

|| – Keyword.

```
x || y
```

Short-circuiting boolean OR.

source

43.2 数学函数

Base.isapprox – Function.

```
isapprox(x, y; rtol::Real=atol>0 ? 0 : √eps, atol::Real=0, nans::Bool=false, norm::Function)
```

Inexact equality comparison: `true` if `norm(x-y) <= max(atol, rtol*max(norm(x), norm(y)))`. The default `atol` is zero and the default `rtol` 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 significand digits. Otherwise, e.g. for integer arguments or if an `atol > 0` is supplied, `rtol` defaults to zero.

`x` and `y` may also be arrays of numbers, in which case `norm` defaults to the usual `norm` function in LinearAlgebra, but may be changed by passing a `norm::Function` keyword argument. (For numbers, `norm` is the same thing as `abs`.) 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.

Examples

```
julia> 0.1 ≈ (0.1 - 1e-10)
true

julia> isapprox(10, 11; atol = 2)
true

julia> isapprox([10.0^9, 1.0], [10.0^9, 2.0])
true

julia> 1e-10 ≈ 0
false

julia> isapprox(1e-10, 0, atol=1e-8)
true
```

source

Base.sin – Method.

```
sin(x)
```

Compute sine of x, where x is in radians.

source

Base.cos – Method.

```
cos(x)
```

Compute cosine of x, where x is in radians.

source

Base.Math.sincos – Method.

```
sincos(x)
```

Simultaneously compute the sine and cosine of x, where the x is in radians.

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.

source

Base.Math.cosd – Function.

```
cosd(x)
```

Compute cosine of x, where x is in degrees.

source

Base.Math.tand – Function.

```
tand(x)
```

Compute tangent of x, where x is in degrees.

source

Base.Math.sinpi – Function.

```
sinpi(x)
```

Compute $\sin(\pi x)$ more accurately than `sin(pi*x)`, especially for large x.

source

Base.Math.cospi – Function.

```
cospi(x)
```

Compute $\cos(\pi x)$ more accurately than `cos(pi*x)`, especially for large x.

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.

source

Base.asin – Method.

```
asin(x)
```

Compute the inverse sine of x, where the output is in radians.

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.

source

Base.Math.asind – Function.

```
asind(x)
```

Compute the inverse sine of x, where the output is in degrees.

source

Base.Math.acosd – Function.

```
acosd(x)
```

Compute the inverse cosine of x, where the output is in degrees.

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.

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.

source

Base.Math.acscd – Function.

```
acscd(x)
```

Compute the inverse cosecant of x, where the output is in degrees.

source

Base.Math.acotd – Function.

```
acotd(x)
```

Compute the inverse cotangent of x, where the output is in degrees.

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 $x = 0$.

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.

Examples

```
julia> deg2rad(90)
1.5707963267948966
```

source

Base.Math.rad2deg – Function.

```
rad2deg(x)
```

Convert x from radians to degrees.

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

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 will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]

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

source

```
hypot(x...)
```

Compute the hypotenuse $\sqrt{\sum |x_i|^2}$ avoiding overflow and underflow.

Examples

```
julia> hypot(-5.7)
5.7

julia> hypot(3, 4im, 12.0)
13.0
```

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.

Examples

```
julia> log(2)
0.6931471805599453

julia> log(-3)
ERROR: DomainError with -3.0:
log 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
[...]
```

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 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 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.

Examples

```
julia> log2(4)
2.0

julia> log2(10)
3.321928094887362

julia> log2(-2)
ERROR: DomainError with -2.0:
NaN result for non-NaN input.
Stacktrace:
 [1] nan_dom_err at ./math.jl:325 [inlined]
[...]
```

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:
NaN result for non-NaN input.
Stacktrace:
 [1] nan_dom_err at ./math.jl:325 [inlined]
[...]
```

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 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 $x \times 2^{exp}$.

source

Base.exp – Method.

```
exp(x)
```

Compute the natural base exponential of x, in other words e^x .

Examples

```
julia> exp(1.0)
2.718281828459045
```

source

Base.exp2 – Function.

```
exp2(x)
```

Compute the base 2 exponential of x, in other words 2^x .

Examples

```
julia> exp2(5)
32.0
```

source

Base.exp10 – Function.

```
exp10(x)
```

Compute the base 10 exponential of x, in other words 10^x .

Examples

```
julia> exp10(2)
100.0
```

source

```
exp10(x)
```

Compute 10^x .

Examples

```
julia> exp10(2)
100.0

julia> exp10(0.2)
1.5848931924611136
```

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 $e^x - 1$.

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.

Chapter 44

Examples

```
julia> x = 1.15
1.15

julia> @sprintf "%.20f" x
"1.14999999999999991118"

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` (`BigFloat` only)
- `RoundUp`
- `RoundDown`

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. This rounding mode may only be used with T == BigFloat inputs to round.

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=, 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 `RoundingMode`s. The first `RoundingMode` is used for rounding the real components while the second is used for rounding the imaginary components.

Example

```
julia> round(3.14 + 4.5im)
3.0 + 4.0im
```

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.

`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.

`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 `x`.

`trunc(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits`, `sigdigits` and `base` work as for `round`.

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 x. If the value is not representable by T, an arbitrary value will be returned.

source

Base.min – Function.

```
min(x, y, ...)
```

Return the minimum of the arguments. 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. 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.

Examples

```
julia> clamp.([pi, 1.0, big(10.)], 2., 9.)
3-element Array{BigFloat,1}:
 3.14159265358979323846264338327950288419716939937510582097494459230781640628619
 2.0
 9.0

julia> clamp.([11,8,5],10,6) # an example where lo > hi
3-element Array{Int64,1}:
  6
  6
 10
```

source

Base.Math.clamp! – Function.

```
clamp!(array::AbstractArray, lo, hi)
```

Restrict values in `array` to the specified range, in-place. See also `clamp`.

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.

Examples

```
julia> abs(-3)
3

julia> abs(1 + im)
1.4142135623730951

julia> abs(typemin(Int64))
-9223372036854775808
```

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`.

Examples

```
julia> abs2(-3)
9
```

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).

source

Base.signbit – Function.

```
signbit(x)
```

Returns `true` if the value of the sign of x is negative, otherwise `false`.

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.

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
```

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 – Method.

```
real(z)
```

Return the real part of the complex number z.

Examples

```
julia> real(1 + 3im)
1
```

source

Base.imag – Function.

```
imag(z)
```

Return the imaginary part of the complex number z.

Examples

```
julia> imag(1 + 3im)
3
```

source

Base.reim – Function.

```
reim(z)
```

Return both the real and imaginary parts of the complex number z.

Examples

```
julia> reim(1 + 3im)
(1, 3)
```

source

Base.conj – Function.

```
conj(z)
```

Compute the complex conjugate of a complex number z.

Examples

```
julia> conj(1 + 3im)
1 - 3im
```

source

Base.angle – Function.

```
angle(z)
```

Compute the phase angle in radians of a complex number z.

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(z)
```

Return $\exp(iz)$.

Examples

```
julia> cis(π) ≈ -1
true
```

source

Base.binomial – Function.

```
binomial(n::Integer, k::Integer)
```

The *binomial coefficient* $\binom{n}{k}$, being the coefficient of the k th 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

$$\binom{n}{k} = (-1)^k \binom{k-n-1}{k}$$

Examples

```
julia> binomial(5, 3)
10

julia> factorial(5) ÷ (factorial(5-3) * factorial(3))
10

julia> binomial(-5, 3)
-35
```

See also

- factorial

External links

- Binomial coeffient 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.

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
```

See also

- binomial

External links

- Factorial on Wikipedia.

source

Base.gcd – Function.

```
gcd(x,y)
```

Greatest common (positive) divisor (or zero if x and y are both zero).

Examples

```
julia> gcd(6,9)
3

julia> gcd(6,-9)
3
```

source

`Base.lcm` – Function.

```
lcm(x,y)
```

Least common (non-negative) multiple.

Examples

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

julia> lcm(-2,3)
6
```

source

`Base.gcdx` – Function.

```
gcdx(x,y)
```

Computes the greatest common (positive) divisor of x and y and their Bézout coefficients, i.e. the integer coefficients u and v that satisfy $ux + vy = d = gcd(x, y)$. $gcdx(x, y)$ returns (d, u, v) .

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 $|u| < |y/d|$ and $|v| < |x/d|$. 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::Integer) -> Bool
```

Test whether n is a power of two.

Examples

```
julia> ispow2(4)
true

julia> ispow2(5)
false
```

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.

Examples

```
julia> nextpow(2, 7)
8

julia> nextpow(2, 9)
16

julia> nextpow(5, 20)
25

julia> nextpow(4, 16)
16
```

See also prevpow.

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.

Examples

```
julia> prevpow(2, 7)
4

julia> prevpow(2, 9)
8

julia> prevpow(5, 20)
5

julia> prevpow(4, 16)
16
```

See also nextpow.

source

Base.nextprod – Function.

```
nextprod([k_1, k_2,...], 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 , etc.

Examples

```
julia> nextprod([2, 3], 105)
108

julia> 2^2 * 3^3
108
```

source

Base.invmod – Function.

```
invmod(x,m)
```

Take the inverse of x modulo m: y such that $xy = 1 \pmod m$, with $div(x,y) = 0$. This is undefined for $m = 0$, or if $gcd(x,m) \neq 1$.

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).

Examples

```
julia> ndigits(12345)
5

julia> ndigits(1022, base=16)
3

julia> string(1022, base=16)
"3fe"

julia> ndigits(123, pad=5)
5
```

source

`Base.widemul` – Function.

```
widemul(x, y)
```

Multiply x and y, giving the result as a larger type.

Examples

```
julia> widemul(Float32(3.), 4.)
12.0
```

source

Missing docstring.

Missing docstring for `Base.Math.evalpoly`. Check Documenter's build log for details.

`Base.Math.@evalpoly` – Macro.

```
@evalpoly(z, c...)
```

Evaluate the polynomial $\sum_k c[k] z^{k-1}$ 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.

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

44.1 Customizable binary operators

Some unicode characters can be used to define new binary operators that support infix notation. For example `⊗(x,y) = kron(x,y)` defines the ⊗ (otimes) function to be the Kronecker product, and one can call it as binary operator using infix syntax: `C = A ⊗ B` as well as with the usual prefix syntax `C = ⊗(A,B)`.

Other characters that support such extensions include \odot ⊙ and \oplus ⊕

The complete list is in the parser code: https://github.com/JuliaLang/julia/blob/master/src/julia-parser.scm

Those that are parsed like * (in terms of precedence) include * / ÷ % & ⋅ ∘ × |\\| ∩ ∧ ⊗ ⊘ ⊙ ⊚ ⊛ ⊠ ⊡ ⊓ ∗ ∙ |∕ ⧸ ⦸ ⨸ ⋄ ⋆ ⨰ ⨱ ⨲ ⨳ ⨴ ⨵ ⨶ ⨷ ⨻ ⨼ ⨽ ⩀ ⩃ ⩄ ⩋ ⩍ ⩎ ⩑ ⩓ ⩕ ⩘ ⩚ ⩜ ⩞ ⩟ ⩠ ⫛ ⫝̸ ⫝ ⫞ ⫟ ⫠ ⫡ ⊲ ⊳ ⋉ ⋊ ⋋ ⋌ ⋏ ⋒ ⟑ ⦸ ⦼ ⦾ ⦿ ⧶ ⧷ ⨇ ⨰ ⨱ ⨲ ⨳ ⨴ ⨵ ⨶ ⨷ ⨸ ⨻ ⨼ ⨽ ⩀ ⩃ ⩄ ⩋ ⩍ ⩎ ⩑ ⩓ ⩕ ⩘ ⩚ ⩜ ⩞ ⩟ ⩠ ⫛ ⊍ ▷ ⨝ ⟕ ⟖ ⟗ and those that are parsed like + include + - |\|| ⊕ ⊖ ⊞ ⊟ |++| ∪ ∨ ⊔ ± ∓ ∔ ∸ ≏ ⊎ ⊻ ⊽ ⋎ ⋓ ⩁ ⩂ ⩅ ⩊ ⩌ ⩏ ⩐ ⩒ ⩔ ⩖ ⩗ ⩛ ⩝ ⩡ ⩢ ⩣ ⧺ ⧻ ⨈ ⨢ ⨣ ⨤ ⨥ ⨦ ⨧ ⨨ ⨩ ⨪ ⨫ ⨬ ⨭ ⨮ ⨹ ⨺ ⩁ ⩂ ⩅ ⩊ ⩌ ⩏ ⩐ ⩒ ⩔ ⩖ ⩗ ⩛ ⩝ ⩡ ⩢ ⩣ There are many others that are related to arrows, comparisons, and powers.

Chapter 45

Numbers

45.1 标准数值类型

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.

source

具象数值类型

Core.Float16 – Type.

```
Float16 <: AbstractFloat
```

16-bit floating point number type.

source

Core.Float32 – Type.

```
Float32 <: AbstractFloat
```

32-bit floating point number type.

source

Core.Float64 – Type.

```
Float64 <: AbstractFloat
```

64-bit floating point number type.

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`.

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.

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.

source

45.2 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)]).

Examples

```
julia> digits(10, base = 10)
2-element Array{Int64,1}:
 0
 1

julia> digits(10, base = 2)
4-element Array{Int64,1}:
 0
 1
 0
 1

julia> digits(10, base = 2, pad = 6)
6-element Array{Int64,1}:
 0
 1
 0
 1
 0
 0
```

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 Array{Int64,1}:
 0
 1
 0
 1

julia> digits!([2,2,2,2,2,2], 10, base = 2)
6-element Array{Int64,1}:
 0
 1
 0
 1
 0
 0
```

source

Base.bitstring – Function.

```
bitstring(n)
```

A string giving the literal bit representation of a number.

Examples

```
julia> bitstring(4)
"00000000000000000000000000000000000000000000000000000000000000100"

julia> bitstring(2.2)
"0100000000000001100110011001100110011001100110011001100110011010"
```

source

Base.parse – Function.

```
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(x)
```

Convert a number to a signed integer. If the argument is unsigned, it is reinterpreted as signed without checking for overflow.

source

Base.unsigned – Function.

```
unsigned(x) -> Unsigned
```

Convert a number to an unsigned integer. If the argument is signed, it is reinterpreted as unsigned without checking for negative values.

Examples

```
julia> unsigned(-2)
0xfffffffffffffffe

julia> unsigned(2)
0x0000000000000002

julia> signed(unsigned(-2))
-2
```

source

Base.float – Method.

```
float(x)
```

Convert a number or array to a floating point data type.

source

Base.Math.significand – Function.

```
significand(x)
```

Extract the `significand(s)` (a.k.a. mantissa), in binary representation, of a floating-point number. If x is a non-zero finite number, then the result will be a number of the same type on the interval $[1, 2)$. Otherwise x is returned.

Examples

```
julia> significand(15.2)/15.2
0.125

julia> significand(15.2)*8
15.2
```

source

Base.Math.exponent – Function.

```
exponent(x) -> Int
```

Get the exponent of a normalized floating-point number.

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 Array{Complex{Int64},1}:
 1 + 0im
 2 + 0im
 3 + 0im
```

source

Base.bswap – Function.

```
bswap(n)
```

Reverse the byte order of `n`.

Examples

```
julia> a = bswap(0x10203040)
0x40302010

julia> bswap(a)
0x10203040

julia> string(1, base = 2)
"1"

julia> string(bswap(1), base = 2)
"10000000000000000000000000000000000000000000000000000000000000000"
```

source

Base.hex2bytes – Function.

```
hex2bytes(s::Union{AbstractString,AbstractVector{UInt8}})
```

Given a string or array `s` 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 `s` gives the value of one byte in the return vector.

The length of `s` must be even, and the returned array has half of the length of `s`. See also `hex2bytes!` for an in-place version, and `bytes2hex` for the inverse.

Examples

```
julia> s = string(12345, base = 16)
"3039"

julia> hex2bytes(s)
2-element Array{UInt8,1}:
 0x30
 0x39

julia> a = b"01abEF"
6-element Base.CodeUnits{UInt8,String}:
 0x30
 0x31
 0x61
 0x62
 0x45
 0x46

julia> hex2bytes(a)
3-element Array{UInt8,1}:
 0x01
 0xab
 0xef
```

source

Base.hex2bytes! – Function.

```
hex2bytes!(d::AbstractVector{UInt8}, s::Union{String,AbstractVector{UInt8}})
```

Convert an array s of bytes representing a hexadecimal string to its binary representation, similar to hex2bytes except that the output is written in-place in d. The length of s must be exactly twice the length of d.

source

Base.bytes2hex – Function.

```
bytes2hex(a::AbstractArray{UInt8}) -> String
bytes2hex(io::IO, a::AbstractArray{UInt8})
```

Convert an array a of bytes to its hexadecimal string representation, either returning a `String` via `bytes2hex(a)` or writing the string to an `io` stream via `bytes2hex(io, a)`. The hexadecimal characters are all lowercase.

Examples

```
julia> a = string(12345, base = 16)
"3039"

julia> b = hex2bytes(a)
2-element Array{UInt8,1}:
 0x30
 0x39

julia> bytes2hex(b)
"3039"
```

source

45.3 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.

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)
```

Returns 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)
```

Get the additive identity element for the type of x (x can also specify the type itself).

Examples

```
julia> zero(1)
0

julia> zero(big"2.0")
0.0

julia> zero(rand(2,2))
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
```

source

Base.im – Constant.

```
im
```

The imaginary unit.

Examples

```
julia> im * im
-1 + 0im
```

source

Base.MathConstants.pi – Constant.

```
π
pi
```

The constant pi.

Examples

```
julia> pi
π = 3.1415926535897...
```

source

Base.MathConstants.□ – Constant.

```
□
e
```

The constant *e*.

Examples

```
julia> □
□ = 2.7182818284590...
```

source

Base.MathConstants.catalan – Constant.

```
catalan
```

Catalan's constant.

Examples

```
julia> Base.MathConstants.catalan
catalan = 0.9159655941772...
```

source

`Base.MathConstants.eulergamma` – Constant.

```
γ
eulergamma
```

Euler's constant.

Examples

```
julia> Base.MathConstants.eulergamma
γ = 0.5772156649015...
```

source

`Base.MathConstants.golden` – Constant.

```
φ
golden
```

The golden ratio.

Examples

```
julia> Base.MathConstants.golden
φ = 1.6180339887498...
```

source

`Base.Inf` – Constant.

```
Inf, Inf64
```

Positive infinity of type `Float64`.

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.

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.

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.

source

Base.isnan – Function.

```
isnan(f) -> Bool
```

Test whether a floating point number is not a number (NaN).

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.

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`.

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(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
```

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
```

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::Integer) -> Bool
```

Return `true` if `x` is odd (that is, not divisible by 2), and `false` otherwise.

Examples

```
julia> isodd(9)
true

julia> isodd(10)
false
```

source

`Base.iseven` – Function.

```
iseven(x::Integer) -> Bool
```

Return `true` is `x` is even (that is, divisible by 2), and `false` otherwise.

Examples

```
julia> iseven(9)
false

julia> iseven(10)
true
```

source

`Core.@int128_str` – Macro.

```
@int128_str str
@int128_str(str)
```

`@int128_str` parses a string into a Int128 Throws an `ArgumentError` if the string is not a valid integer

source

`Core.@uint128_str` – Macro.

```
@uint128_str str
@uint128_str(str)
```

`@uint128_str` parses a string into a UInt128 Throws an `ArgumentError` if the string is not a valid integer

source

45.4 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.

> **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
2.099999999999999999999999999999999999999999999999999999999999999999999999999999986

julia> BigFloat("2.1", RoundUp)
2.100000000000000000000000000000000000000000000000000000000000000000000000000000021

julia> BigFloat("2.1", RoundUp, precision=128)
2.100000000000000000000000000000000007
```

See also

- `@big_str`
- `rounding` and `setrounding`
- `precision` and `setprecision`

source

`Base.precision` – Function.

```
precision(num::AbstractFloat)
```

Get the precision of a floating point number, as defined by the effective number of bits in the mantissa.

source

`Base.precision` – Method.

```
precision(BigFloat)
```

Get the precision (in bits) currently used for `BigFloat` arithmetic.

source

`Base.MPFR.setprecision` – Function.

```
setprecision([T=BigFloat,] precision::Int)
```

Set the precision (in bits) to be used for T arithmetic.

> **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

```
setprecision(f::Function, [T=BigFloat,] precision::Integer)
```

Change the T arithmetic precision (in bits) 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`

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
```

source

`Core.@big_str` – Macro.

```
@big_str str
@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
```

source

Chapter 46

字符串

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
```

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`.

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)
```

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.

```
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, except nothing, 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.

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 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.

Examples

```
julia> repr(1)
"1"

julia> repr(zeros(3))
"[0.0, 0.0, 0.0]"

julia> repr(big(1/3))
"0.33333333333333331482961625624739099293947219848632812 5"

julia> repr(big(1/3), context=:compact => true)
"0.333333"
```

source

Core.String – Method.

```
String(s::AbstractString)
```

Convert a string to a contiguous byte array representation encoded as UTF-8 bytes. This representation is often appropriate for passing strings to C.

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.

Examples

```
julia> SubString("abc", 1, 2)
"ab"

julia> SubString("abc", 1:2)
"ab"

julia> SubString("abc", 2)
"bc"
```

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.

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)`.

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 unit code 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)`.

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.

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.

Examples

```
julia> ascii("abcdeγfgh")
ERROR: ArgumentError: invalid ASCII at index 6 in "abcdeγfgh"
Stacktrace:
[...]

julia> ascii("abcdefgh")
"abcdefgh"
```

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` disables UCP mode (enables ASCII mode). 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.

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)
```

Stores the given string `substr` as a `SubstitutionString`, for use in regular expression substitutions. Most commonly constructed using the `@s_str` macro.

```
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`.

```
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.

source

Base.Docs.@text_str – Macro.

```
@text_str -> Docs.Text
```

Create a Text object from a literal string.

source

Base.isvalid – Method.

```
isvalid(value) -> Bool
```

Returns `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
```

Returns `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, or `Vector{UInt8}` or `SubString{String}`.

Examples

```
julia> isvalid(Char, 0xd800)
false

julia> isvalid(String, SubString("thisisvalid",1,5))
true

julia> isvalid(Char, 0xd799)
true
```

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("αβγdef", 2)
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 Array{Union{Nothing, SubString{String}},1}:
 "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 a the regular expression `r` in `s` and return a 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 Array{RegexMatch,1}:
 RegexMatch("a1a")
 RegexMatch("a3a")

julia> collect(eachmatch(rx, "a1a2a3a", overlap = true))
3-element Array{RegexMatch,1}:
 RegexMatch("a1a")
 RegexMatch("a2a")
 RegexMatch("a3a")
```

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 (code points) long. If `s` is already `n` characters long, an equal string is returned. Pad with spaces by default.

Examples

```
julia> lpad("March", 10)
"     March"
```

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 (code points) long. If `s` is already `n` characters long, an equal string is returned. Pad with spaces by default.

Examples

```
julia> rpad("March", 20)
"March               "
```

source

Base.findfirst – Method.

```
findfirst(pattern::AbstractString, string::AbstractString)
findfirst(pattern::Regex, 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::Regex, 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,Regex,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
```

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"

julia> reverse("a̋e") # combining characters can lead to surprising results
"ê̋a"

julia> using Unicode
```

```
julia> join(reverse(collect(graphemes("a̲x̂e"))) # reverses graphemes
"ex̂a"
```

source

Base.replace – Method.

```
replace(s::AbstractString, pat=>r; [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 Regex 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` (`""`).

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."
```

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`.

Examples

```
julia> a = "Ma.rch"
"Ma.rch"

julia> split(a,".")
2-element Array{SubString{String},1}:
 "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 Array{SubString{String},1}:
 "M"
 "a"
 "r"
 "c"
 "h"

julia> rsplit(a,".";limit=1)
1-element Array{SubString{String},1}:
 "M.a.r.c.h"

julia> rsplit(a,".";limit=2)
2-element Array{SubString{String},1}:
 "M.a.r.c"
 "h"
```

source

Base.strip – Function.

```
strip([pred=isspace,] str::AbstractString)
strip(str::AbstractString, chars)
```

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 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.

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)
lstrip(str::AbstractString, chars)
```

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.

Examples

```
julia> a = lpad("March", 20)
"               March"

julia> lstrip(a)
"March"
```

source

Base.rstrip – Function.

```
rstrip([pred=isspace,] str::AbstractString)
rstrip(str::AbstractString, chars)
```

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.

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`.

Examples

```
julia> startswith("JuliaLang", "Julia")
true
```

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`.

Examples

```
julia> endswith("Sunday", "day")
true
```

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

Missing docstring.

Missing docstring for Base.contains. Check Documenter's build log for details.

Base.first – Method.

```
first(s::AbstractString, n::Integer)
```

Get a string consisting of the first n characters of s.

```
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.

```
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(s::AbstractString)
```

Return s with all characters converted to uppercase.

Examples

```
julia> uppercase("Julia")
"JULIA"
```

source

Base.Unicode.lowercase – Function.

```
lowercase(s::AbstractString)
```

Return s with all characters converted to lowercase.

Examples

```
julia> lowercase("STRINGS AND THINGS")
"strings and things"
```

source

Base.Unicode.titlecase – Function.

```
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 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.

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,] strings [, delim [, last]])
```

Join an array of strings into a single string, inserting the given delimiter (if any) between adjacent strings. If last is given, it will be used instead of delim between the last two strings. If io is given, the result is written to io rather than returned as as a String.

strings can be any iterable over elements x which are convertible to strings via print(io::IOBuffer, x). strings will be printed to io.

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.

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.chomp – Function.

```
chomp(s::AbstractString)
```

Remove a single trailing newline from a string.

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 String
  at index [4]
[...]

julia> thisind("α", -1)
ERROR: BoundsError: attempt to access 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 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 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
```

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).

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'.

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. A character is classified as lowercase if it belongs to Unicode category Ll, Letter: Lowercase.

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 ৫. Use `isdigit` to check whether a character a decimal digit between 0 and 9.

Examples

```
julia> isnumeric('৫')
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. A character is classified as uppercase if it belongs to Unicode category Lu, Letter: Uppercase, or Lt, Letter: Titlecase.

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])::AbstractString
escape_string(io, str::AbstractString[, esc::])::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).

Examples

```
julia> escape_string("aaa\nbbb")
"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"
```

See also

unescape_string for the reverse operation.

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 code points (\u or \U prefixes with 1-4 trailing hex digits)
- Hex bytes (\x with 1-2 trailing hex digits)
- Octal bytes (\ with 1-3 trailing octal digits)

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"
```

See also

escape_string.

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.

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`. See `undef`.

Examples

```
julia> A = Array{Float64,2}(undef, 2, 3) # N given explicitly
2×3 Array{Float64,2}:
 6.90198e-310  6.90198e-310  6.90198e-310
 6.90198e-310  6.90198e-310  0.0

julia> B = Array{Float64}(undef, 2) # N determined by the input
2-element Array{Float64,1}:
 1.87103e-320
 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 Array{Union{Nothing, String},1}:
 nothing
 nothing

julia> Array{Union{Nothing, Int}}(nothing, 2, 3)
2×3 Array{Union{Nothing, Int64},2}:
 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 Array{Union{Missing, String},1}:
 missing
```

```
 missing

julia> Array{Union{Missing, Int}}(missing, 2, 3)
2×3 Array{Union{Missing, Int64},2}:
 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.

Examples

```
julia> Array{Float64,1}(undef, 3)
3-element Array{Float64,1}:
 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}.

source

Base.Vector – Method.

```
Vector{T}(undef, n)
```

Construct an uninitialized Vector{T} of length n. See undef.

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 Array{Union{Nothing, String},1}:
 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 Array{Union{Missing, String},1}:
 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}.

source

Base.Matrix – Method.

```
Matrix{T}(undef, m, n)
```

Construct an uninitialized Matrix{T} of size m×n. See undef.

Examples

```
julia> Matrix{Float64}(undef, 2, 3)
2×3 Array{Float64,2}:
 6.93517e-310  6.93517e-310  6.93517e-310
 6.93517e-310  6.93517e-310  1.29396e-320
```

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 Array{Union{Nothing, String},2}:
 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 Array{Union{Missing, String},2}:
 missing  missing  missing
 missing  missing  missing
```

source

Base.VecOrMat – Type.

```
VecOrMat{T}
```

Union type of Vector{T} and Matrix{T}.

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}
```

An N dimensional *strided* array with elements of type T. These arrays follow the strided array interface. 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 [strides(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.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 Array{Int8,1}:
 1
 2
 3

julia> getindex(Int8, 1, 2, 3)
3-element Array{Int8,1}:
 1
 2
 3
```

source

Base.zeros – Function.

```
zeros([T=Float64,] dims...)
```

Create an Array, with element type T, of all zeros with size specified by dims. See also fill, ones.

Examples

```
julia> zeros(1)
1-element Array{Float64,1}:
 0.0

julia> zeros(Int8, 2, 3)
2×3 Array{Int8,2}:
 0  0  0
 0  0  0
```

source

Base.ones – Function.

```
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 Array{Float64,2}:
 1.0  1.0

julia> ones(ComplexF64, 2, 3)
2×3 Array{Complex{Float64},2}:
 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.

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 BitArray{2}:
 false  false
 false  true

julia> BitArray(undef, (3, 1))
3×1 BitArray{2}:
 false
 true
 false
```

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 BitArray{2}:
 1  0
 0  1

julia> BitArray(x+y == 3 for x = 1:2, y = 1:3)
2×3 BitArray{2}:
 0  1  0
 1  0  0

julia> BitArray(x+y == 3 for x = 1:2 for y = 1:3)
```

```
6-element BitArray{1}:
 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 BitArray{2}:
 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 BitArray{2}:
 0  0  0
 0  0  0
```

source

Base.fill – Function.

```
fill(x, dims)
```

Create an array filled with the value x. For example, fill(1.0, (5,5)) returns a 5×5 array of floats, with each element initialized to 1.0.

Examples

```
julia> fill(1.0, (5,5))
5×5 Array{Float64,2}:
 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  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0
```

If x is an object reference, all elements will refer to the same object. `fill(Foo(), dims)` will return an array filled with the result of evaluating `Foo()` once.

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 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> fill!(A, 2.)
2×3 Array{Float64,2}:
 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 Array{Array{Int64,1},1}:
 [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 Array{Int64,1}:
 1
 1
 1
```

source

`Base.similar` – Function.

```
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 Array{Int64,2}:
 4419743872  4374413872  4419743888  0
```

Conversely, `similar(trues(10,10), 2)` returns an uninitialized `BitVector` with two elements since `BitArrays` are both mutable and can support 1-dimensional arrays:

```
julia> similar(trues(10,10), 2)
2-element BitArray{1}:
 0
 0
```

Since `BitArrays` 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 Array{Float64,2}:
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
 2.18425e-314  2.18425e-314  2.18425e-314  2.18425e-314
```

source

```
similar(storagetype, axes)
```

Create an uninitialized mutable array analogous to that specified by `storagetype`, but with `axes` specified by the last argument. `storagetype` might be a type or a function.

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`.

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.

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.

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)
```

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.eachindex – Function.

```
eachindex(A...)
```

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). For other array types, return a specialized Cartesian range to efficiently index into the array with indices specified for every dimension. For other iterables, including strings and dictionaries, return an iterator object supporting arbitrary index types (e.g. unevenly spaced or non-integer indices).

If you supply more than one AbstractArray argument, eachindex will create an iterable object that is fast for all arguments (a UnitRange if all inputs have fast linear indexing, a CartesianIndices otherwise). If the arrays have different sizes and/or dimensionalities, eachindex will return an iterable that spans the largest range along each dimension.

Examples

```
julia> A = [1 2; 3 4];

julia> for i in eachindex(A) # linear indexing
           println(i)
       end
1
2
3
4

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 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) convert 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 to describe the position in the array (even if it's a multidimensional array) and column-major ordering is used to access the elements. For example, if `A` were a `(2, 3)` custom matrix type with linear indexing, and we referenced `A[5]` (using linear style), this would be equivalent to referencing `A[1, 3]` (since `2*1 + 3 = 5`). See also `IndexCartesian`.

source

`Base.IndexCartesian` – Type.

```
IndexCartesian()
```

Subtype of `IndexStyle` used to describe arrays which are optimally indexed by a Cartesian index.

A cartesian indexing style uses multiple integers/indices to describe the position in the array. For example, if `A` were a `(2, 3, 4)` custom matrix type with cartesian indexing, we could reference `A[2, 1, 3]` and Julia would automatically convert this into the correct location in the underlying memory. 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 Array{Complex{Int64},2}:
 1+1im  2-1im
 2+2im  3+1im

julia> conj!(A);

julia> A
2×2 Array{Complex{Int64},2}:
 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.

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.

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, Refs 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 `Numbers`, `Strings`, `Symbols`, `Types`, `Functions` 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 Array{Int64,1}:
 1
 2
 3
 4
 5

julia> B = [1 2; 3 4; 5 6; 7 8; 9 10]
5×2 Array{Int64,2}:
 1   2
 3   4
 5   6
 7   8
 9  10

julia> broadcast(+, A, B)
5×2 Array{Int64,2}:
  2   3
  5   6
  8   9
 11  12
 14  15

julia> parse.(Int, ["1", "2"])
2-element Array{Int64,1}:
 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 Array{Array{Int64,1},1}:
 [1, 1]
 [2, 2]

julia> string.(("one","two","three","four"), ": ", 1:4)
4-element Array{String,1}:
 "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 Array{Float64,1}:
  1.0
 -2.0

julia> A
2-element Array{Float64,1}:
 1.0
 0.0

julia> broadcast!(+, A, A, (0, -2.0));

julia> A
2-element Array{Float64,1}:
  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 Array{Float64,1}:
 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 Array{Int64,1}:
 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 BroadcastStyles 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 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 Array{Int64,2}:
 1  2
 3  4

julia> getindex(A, 1)
1

julia> getindex(A, [2, 1])
2-element Array{Int64,1}:
 3
 1

julia> getindex(A, 2:4)
3-element Array{Int64,1}:
 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...)`.

Examples

```
julia> A = zeros(2,2);

julia> setindex!(A, [10, 20], [1, 2]);

julia> A[[3, 4]] = [30, 40];

julia> A
2×2 Array{Float64,2}:
 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.

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 Array{Foo,1}:
 #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.

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
```

source

Base.IteratorsMD.CartesianIndices – Type.

```
CartesianIndices(sz::Dims) -> R
CartesianIndices((istart:istop, jstart: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:jstop
    for i = istart: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.

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)))
2×3 CartesianIndices{2,Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)  CartesianIndex(1, 3)
 CartesianIndex(2, 1)  CartesianIndex(2, 2)  CartesianIndex(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))
3×2 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)
 CartesianIndex(2, 1)  CartesianIndex(2, 2)
 CartesianIndex(3, 1)  CartesianIndex(3, 2)

julia> cartesian[4]
CartesianIndex(1, 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))
2×2 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(2, 5)  CartesianIndex(2, 6)
 CartesianIndex(3, 5)  CartesianIndex(3, 6)

julia> CI = CartesianIndex(3, 4)
CartesianIndex(3, 4)

julia> CIs .+ CI
2×2 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(5, 9)  CartesianIndex(5, 10)
 CartesianIndex(6, 9)  CartesianIndex(6, 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 `Int`s or `AbstractArray`s 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`.

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.

Examples

```
julia> checkindex(Bool, 1:20, 8)
true

julia> checkindex(Bool, 1:20, 21)
false
```

source

47.5 Views (SubArrays 以及其它 view 类型)

A "view" is a data structure that acts like an array (it is a subtype of `AbstractArray`), but the underlying data is actually part of another array.

For example, if `x` is an array and `v = @view x[1:10]`, then `v` acts like a 10-element array, but its data is actually accessing the first 10 elements of `x`. Writing to a view, e.g. `v[3] = 2`, writes directly to the underlying array `x` (in this case modifying `x[3]`).

Slicing operations like `x[1:10]` create a copy by default in Julia. `@view x[1:10]` changes it to make a view. The `@views` macro can be used on a whole block of code (e.g. `@views function foo() .... end` or `@views begin ... end`) to change all the slicing operations in that block to use views. Sometimes making a copy of the data is faster and sometimes using a view is faster, as described in the performance tips.

`Base.view` – Function.

```
view(A, inds...)
```

Like `getindex`, but returns a view into the parent array `A` with the given indices instead of making a copy. Calling `getindex` or `setindex!` on the returned `SubArray` computes the indices to the parent array on the fly without checking bounds.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> b = view(A, :, 1)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 1
 3
```

```
julia> fill!(b, 0)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 0
 0

julia> A # Note A has changed even though we modified b
2×2 Array{Int64,2}:
 0  2
 0  4
```

source

Base.@view – Macro.

```
@view A[inds...]
```

Creates a SubArray from an indexing expression. This can only be applied directly to a reference expression (e.g. `@view A[1,2:end]`), and should *not* be used as the target of an assignment (e.g. `@view(A[1,2:end]) = ...`). See also @views to switch an entire block of code to use views for slicing.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> b = @view A[:, 1]
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 1
 3

julia> fill!(b, 0)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 0
 0

julia> A
2×2 Array{Int64,2}:
 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.

> **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.

Examples

```
julia> A = zeros(3, 3);

julia> @views for row in 1:3
           b = A[row, :]
           b[:] .= row
       end

julia> A
3×3 Array{Float64,2}:
 1.0  1.0  1.0
 2.0  2.0  2.0
 3.0  3.0  3.0
```

source

Base.parent – Function.

```
parent(A)
```

Returns the ”parent array” of an array view type (e.g., SubArray), or the array itself if it is not a view.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> V = view(A, 1:2, :)
2×2 view(::Array{Int64,2}, 1:2, :) with eltype Int64:
 1  2
 3  4

julia> parent(V)
2×2 Array{Int64,2}:
 1  2
 3  4
```

source

Base.parentindices – Function.

```
parentindices(A)
```

Return the indices in the parent which correspond to the array view A.

Examples

```
julia> A = [1 2; 3 4];

julia> V = view(A, 1, :)
2-element view(::Array{Int64,2}, 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.

Examples

```
julia> A = [1 2 3 4; 5 6 7 8]
2×4 Array{Int64,2}:
 1  2  3  4
 5  6  7  8

julia> selectdim(A, 2, 3)
2-element view(::Array{Int64,2}, :, 3) with eltype Int64:
 3
 7
```

source

Base.reinterpret – Function.

```
reinterpret(type, A)
```

Change the type-interpretation of a block of memory. For arrays, this constructs a view of the array with the same binary data as the given array, but with the specified element type. For example, reinterpret(Float32, UInt32(7)) interprets the 4 bytes corresponding to UInt32(7) as a Float32.

Examples

```
julia> reinterpret(Float32, UInt32(7))
1.0f-44

julia> reinterpret(Float32, UInt32[1 2 3 4 5])
1×5 reinterpret(Float32, ::Array{UInt32,2}):
 1.4013e-45  2.8026e-45  4.2039e-45  5.60519e-45  7.00649e-45
```

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 Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16

julia> reshape(A, (4, 4))
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> reshape(A, 2, :)
2×8 Array{Int64,2}:
 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)
```

Remove the dimensions specified by `dims` from array `A`. Elements of `dims` must be unique and within the range `1:ndims(A)`. `size(A,i)` must equal 1 for all `i` in `dims`.

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> dropdims(a; dims=3)
2×2×1 Array{Int64,3}:
[:, :, 1] =
```

```
1  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 Array{Int64,2}:
 1  2  3
 4  5  6

julia> vec(a)
6-element Array{Int64,1}:
 1
 4
 2
 5
 3
 6

julia> vec(1:3)
1:3
```

See also reshape.

source

47.6 Concatenation and permutation

Base.cat – Function.

```
cat(A...; dims=dims)
```

Concatenate the input arrays along the specified dimensions in the iterable `dims`. For dimensions not in `dims`, all input arrays should have the same size, which will also be the size of the output array along that dimension. For dimensions in `dims`, the size of the output array is the sum of the sizes of the input arrays along that dimension. If `dims` is a single number, the different arrays are tightly stacked along that dimension. If `dims` is an iterable containing several dimensions, this allows one to construct block diagonal matrices and their higher-dimensional analogues by simultaneously increasing several dimensions for every new input array and putting zero blocks elsewhere. For example, `cat(matrices...; dims=(1,2))` builds a block diagonal matrix, i.e. a block matrix with `matrices[1]`, `matrices[2]`, ... as diagonal blocks and matching zero blocks away from the diagonal.

source

Base.vcat – Function.

```
vcat(A...)
```

Concatenate along dimension 1.

Examples

```
julia> a = [1 2 3 4 5]
1×5 Array{Int64,2}:
 1  2  3  4  5

julia> b = [6 7 8 9 10; 11 12 13 14 15]
2×5 Array{Int64,2}:
  6   7   8   9  10
 11  12  13  14  15

julia> vcat(a,b)
3×5 Array{Int64,2}:
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15

julia> c = ([1 2 3], [4 5 6])
([1 2 3], [4 5 6])

julia> vcat(c...)
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

source

Base.hcat – Function.

```
hcat(A...)
```

Concatenate along dimension 2.

Examples

```
julia> a = [1; 2; 3; 4; 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> b = [6 7; 8 9; 10 11; 12 13; 14 15]
5×2 Array{Int64,2}:
  6   7
  8   9
 10  11
 12  13
 14  15

julia> hcat(a,b)
5×3 Array{Int64,2}:
```

```
 1   6   7
 2   8   9
 3  10  11
 4  12  13
 5  14  15

julia> c = ([1; 2; 3], [4; 5; 6])
([1, 2, 3], [4, 5, 6])

julia> hcat(c...)
3×2 Array{Int64,2}:
 1  4
 2  5
 3  6
```

source

Base.hvcat – Function.

```
hvcat(rows::Tuple{Vararg{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.

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 Array{Int64,2}:
 1  2  3
 4  5  6

julia> hvcat((3,3), a,b,c,d,e,f)
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> [a b;c d; e f]
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6

julia> hvcat((2,2,2), a,b,c,d,e,f)
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6
```

If the first argument is a single integer n, then all block rows are assumed to have n block columns.

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 Array{Float64,1}:
 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.

Examples

```
julia> b = 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> circshift(b, (0,2))
4×4 Array{Int64,2}:
  9  13  1  5
 10  14  2  6
 11  15  3  7
 12  16  4  8

julia> circshift(b, (-1,0))
4×4 Array{Int64,2}:
 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 BitArray{1}:
 1
 1
 0
 0
 1

julia> circshift(a, 1)
5-element BitArray{1}:
 1
 1
 1
 0
```

```
 0

julia> circshift(a, -1)
5-element BitArray{1}:
 1
 0
 0
 1
 1
```

See also `circshift!`.

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.

The `dest` array must be distinct from the `src` array (they cannot alias each other).

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`.

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)`.

Examples

```
julia> A = [true, false, false, true]
4-element Array{Bool,1}:
 1
 0
 0
 1

julia> findall(A)
2-element Array{Int64,1}:
 1
 4

julia> A = [true false; false true]
2×2 Array{Bool,2}:
 1  0
 0  1

julia> findall(A)
2-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 2)

julia> findall(falses(3))
0-element Array{Int64,1}
```

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 Array{Int64,1}:
 1
 3
 4

julia> findall(isodd, x)
2-element Array{Int64,1}:
 1
 2
```

```
julia> A = [1 2 0; 3 4 0]
2×3 Array{Int64,2}:
 1  2  0
 3  4  0
julia> findall(isodd, A)
2-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)

julia> findall(!iszero, A)
4-element Array{CartesianIndex{2},1}:
 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 Array{Symbol,1}:
 :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)`.

Examples

```
julia> A = [false, false, true, false]
4-element Array{Bool,1}:
 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 Array{Bool,2}:
 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 Array{Int64,1}:
 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 Array{Int64,2}:
 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).

Examples

```
julia> A = [true, false, true, false]
4-element Array{Bool,1}:
 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 Array{Bool,2}:
 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 Array{Int64,1}:
 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 Array{Int64,2}:
 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 Array{Bool,1}:
 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 Array{Bool,2}:
 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)`.

Examples

```
julia> A = [false, false, true, true]
4-element Array{Bool,1}:
 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 Array{Bool,2}:
 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 Array{Int64,1}:
 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 Array{Int64,2}:
 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 specifying a permutation of length `ndims(A)`.

See also: `PermutedDimsArray`.

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> permutedims(A, [3, 2, 1])
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  3
 5  7

[:, :, 2] =
 2  4
 6  8
```

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 Array{Array{Int64,2},2}:
 [1 2; 3 4]     [5 6; 7 8]
 [9 10; 11 12]  [13 14; 15 16]

julia> permutedims(X)
2×2 Array{Array{Int64,2},2}:
 [1 2; 3 4]  [9 10; 11 12]
 [5 6; 7 8]  [13 14; 15 16]

julia> transpose(X)
2×2 Transpose{Transpose{Int64,Array{Int64,2}},Array{Array{Int64,2},2}}:
 [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 Array{Int64,2}:
 1  2  3  4

julia> V = [[[1 2; 3 4]]; [[5 6; 7 8]]]
2-element Array{Array{Int64,2},1}:
 [1 2; 3 4]
 [5 6; 7 8]

julia> permutedims(V)
1×2 Array{Array{Int64,2},2}:
 [1 2; 3 4]  [5 6; 7 8]

julia> transpose(V)
1×2 Transpose{Transpose{Int64,Array{Int64,2}},Array{Array{Int64,2},1}}:
 [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`.

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 Array functions

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`

Examples

```
julia> accumulate(+, [1,2,3])
3-element Array{Int64,1}:
 1
 3
 6

julia> accumulate(*, [1,2,3])
3-element Array{Int64,1}:
 1
 2
 6

julia> accumulate(+, [1,2,3]; init=100)
3-element Array{Int64,1}:
 101
 103
```

```
 106

julia> accumulate(min, [1,2,-1]; init=0)
3-element Array{Int64,1}:
  0
  0
 -1

julia> accumulate(+, fill(1, 3, 3), dims=1)
3×3 Array{Int64,2}:
 1  1  1
 2  2  2
 3  3  3

julia> accumulate(+, fill(1, 3, 3), dims=2)
3×3 Array{Int64,2}:
 1  2  3
 1  2  3
 1  2  3
```

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. See also accumulate.

Examples

```
julia> x = [1, 0, 2, 0, 3];

julia> y = [0, 0, 0, 0, 0];

julia> accumulate!(+, y, x);

julia> y
5-element Array{Int64,1}:
 1
 1
 3
 3
 6

julia> A = [1 2; 3 4];

julia> B = [0 0; 0 0];

julia> accumulate!(-, B, A, dims=1);

julia> B
2×2 Array{Int64,2}:
  1   2
 -2  -2
```

```
julia> accumulate!(-, B, A, dims=2);

julia> B
2×2 Array{Int64,2}:
 1  -1
 3  -1
```

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 = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> cumprod(a, dims=1)
2×3 Array{Int64,2}:
 1   2   3
 4  10  18

julia> cumprod(a, dims=2)
2×3 Array{Int64,2}:
 1   2    6
 4  20  120
```

source

```
cumprod(x::AbstractVector)
```

Cumulative product of a vector. 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> cumprod(fill(1//2, 3))
3-element Array{Rational{Int64},1}:
 1//2
 1//4
 1//8

julia> cumprod([fill(1//3, 2, 2) for i in 1:3])
3-element Array{Array{Rational{Int64},2},1}:
 [1//3 1//3; 1//3 1//3]
 [2//9 2//9; 2//9 2//9]
 [4//27 4//27; 4//27 4//27]
```

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.

source

```
cumprod!(y::AbstractVector, x::AbstractVector)
```

Cumulative product of a vector x, storing the result in y. See also cumprod.

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 Array{Int64,2}:
 1  2  3
 4  5  6

julia> cumsum(a, dims=1)
2×3 Array{Int64,2}:
 1  2  3
 5  7  9

julia> cumsum(a, dims=2)
2×3 Array{Int64,2}:
 1  3   6
 4  9  15
```

source

```
cumsum(x::AbstractVector)
```

Cumulative sum a vector. 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> cumsum([1, 1, 1])
3-element Array{Int64,1}:
 1
 2
 3

julia> cumsum([fill(1, 2) for i in 1:3])
3-element Array{Array{Int64,1},1}:
 [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.

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 Array{Int64,2}:
 2   4
 6  16

julia> diff(a, dims=2)
2×1 Array{Int64,2}:
  2
 10

julia> diff(vec(a))
3-element Array{Int64,1}:
  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.

Examples

```
julia> repeat([1, 2, 3], 2)
6-element Array{Int64,1}:
 1
 2
 3
 1
 2
 3

julia> repeat([1, 2, 3], 2, 3)
```

```
6×3 Array{Int64,2}:
 1  1  1
 2  2  2
 3  3  3
 1  1  1
 2  2  2
 3  3  3
```

source

```
repeat(A::AbstractArray; inner=ntuple(x->1, ndims(A)), outer=ntuple(x->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 Array{Int64,1}:
 1
 1
 2
 2

julia> repeat(1:2, outer=2)
4-element Array{Int64,1}:
 1
 2
 1
 2

julia> repeat([1 2; 3 4], inner=(2, 1), outer=(1, 3))
4×6 Array{Int64,2}:
 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 Array{Int64,2}:
 1  2
 3  4

julia> rot180(a)
2×2 Array{Int64,2}:
 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 Array{Int64,2}:
 1  2
 3  4

julia> rot180(a,1)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rot180(a,2)
2×2 Array{Int64,2}:
 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 Array{Int64,2}:
 1  2
 3  4

julia> rotl90(a)
2×2 Array{Int64,2}:
 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 Array{Int64,2}:
 1  2
 3  4

julia> rotl90(a,1)
2×2 Array{Int64,2}:
 2  4
 1  3

julia> rotl90(a,2)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rotl90(a,3)
2×2 Array{Int64,2}:
 3  1
 4  2

julia> rotl90(a,4)
2×2 Array{Int64,2}:
 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 Array{Int64,2}:
 1  2
```

```
 3  4

julia> rotr90(a)
2×2 Array{Int64,2}:
 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 Array{Int64,2}:
 1  2
 3  4

julia> rotr90(a,1)
2×2 Array{Int64,2}:
 3  1
 4  2

julia> rotr90(a,2)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rotr90(a,3)
2×2 Array{Int64,2}:
 2  4
 1  3

julia> rotr90(a,4)
2×2 Array{Int64,2}:
 1  2
 3  4
```

source

`Base.mapslices` – Function.

```
mapslices(f, A; dims)
```

Transform the given dimensions of array `A` using function `f`. `f` is called on each slice of `A` of the form `A[...,:,...,:,...]`. `dims` is an integer vector specifying where the colons go in this expression. The results are concatenated along the remaining dimensions. For example, if `dims` is `[1,2]` and `A` is 4-dimensional, `f` is called on `A[:,:,i,j]` for all `i` and `j`.

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> mapslices(sum, a, dims = [1,2])
1×1×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 10

[:, :, 2, 1] =
 26

[:, :, 1, 2] =
 42

[:, :, 2, 2] =
 58
```

source

Base.eachrow – Function.

```
eachrow(A::AbstractVecOrMat)
```

Create a generator that iterates over the first dimension of vector or matrix A, returning the rows as views.

See also eachcol and eachslice.

Julia 1.1

This function requires at least Julia 1.1.

source

Base.eachcol – Function.

```
eachcol(A::AbstractVecOrMat)
```

Create a generator that iterates over the second dimension of matrix A, returning the columns as views.

See also eachrow and eachslice.

Julia 1.1

This function requires at least Julia 1.1.

source

Base.eachslice – Function.

```
eachslice(A::AbstractArray; dims)
```

Create a generator that iterates over dimensions `dims` of `A`, returning views that select all the data from the other dimensions in `A`.

Only a single dimension in `dims` is currently supported. Equivalent to `(view(A,:,:,...,i,:,: ...)) for i in axes(A, dims))`, where `i` is in position `dims`.

See also `eachrow`, `eachcol`, and `selectdim`.

> **Julia 1.1**
> This function requires at least Julia 1.1.

source

47.8 Combinatorics

Base.invperm – Function.

```
invperm(v)
```

Return the inverse permutation of `v`. If `B = A[v]`, then `A == B[invperm(v)]`.

Examples

```
julia> v = [2; 4; 3; 1];

julia> invperm(v)
4-element Array{Int64,1}:
 4
 1
 3
 2

julia> A = ['a','b','c','d'];

julia> B = A[v]
4-element Array{Char,1}:
 'b'
 'd'
 'c'
 'a'

julia> B[invperm(v)]
4-element Array{Char,1}:
 'a'
 'b'
 'c'
 'd'
```

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]`. Note that this is generally faster than `permute!(v,p)` for large vectors.

See also `invpermute!`.

Examples

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> permute!(A, perm);

julia> A
4-element Array{Int64,1}:
 1
 4
 3
 1
```

source

`Base.invpermute!` – Function.

```
invpermute!(v, p)
```

Like `permute!`, but the inverse of the given permutation is applied.

Examples

```
julia> A = [1, 1, 3, 4];

julia> perm = [2, 4, 3, 1];

julia> invpermute!(A, perm);

julia> A
4-element Array{Int64,1}:
 4
 1
 3
 1
```

source

Base.reverse – Method.

```
reverse(v [, start=1 [, stop=length(v) ]] )
```

Return a copy of v reversed from start to stop. See also Iterators.reverse for reverse-order iteration without making a copy.

Examples

```
julia> A = Vector(1:5)
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> reverse(A)
5-element Array{Int64,1}:
 5
 4
 3
 2
 1

julia> reverse(A, 1, 4)
5-element Array{Int64,1}:
 4
 3
 2
 1
 5

julia> reverse(A, 3, 5)
5-element Array{Int64,1}:
 1
 2
 5
 4
 3
```

source

```
reverse(A; dims::Integer)
```

Reverse A in dimension dims.

Examples

```
julia> b = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> reverse(b, dims=2)
```

```
2×2 Array{Int64,2}:
 2  1
 4  3
```

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> r = reverse("Julia")
"ailuJ"

julia> for i in 1:length(r)
           print(r[reverseind("Julia", i)])
       end
Julia
```

source

Base.reverse! – Function.

```
reverse!(v [, start=1 [, stop=length(v) ]]) -> v
```

In-place version of `reverse`.

Examples

```
julia> A = Vector(1:5)
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> reverse!(A);

julia> A
5-element Array{Int64,1}:
 5
 4
 3
 2
 1
```

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.

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.

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.

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 less 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"
```

Note

Currently, all tasks in Julia are executed in a single OS thread co-operatively. Consequently, `asyncmap` is beneficial only when the mapping function involves any I/O - disk, network, remote worker invocation, etc.

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.

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
```

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.

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.@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`.

source

Base.wait – Function.

```
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.
- `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

Special note for `Threads.Condition`:

The caller must be holding the `lock` that owns `c` before calling this method. The calling task will be blocked until some other task wakes it, usually by calling `notify`‘ on the same 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.

```
wait(r::RemoteChannel, args...)
```

Wait for a value to become available on the specified RemoteChannel.

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.timedwait – Function.

```
timedwait(testcb::Function, secs::Float64; pollint::Float64=0.1)
```

Waits until testcb returns true or for secs seconds, whichever is earlier. testcb is polled every pollint seconds.

Returns :ok, :timed_out, or :error

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.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.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.

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

`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

`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`.

source

`Base.islocked` – Function.

```
islocked(lock) -> Status (Boolean)
```

Check whether the `lock` is held by any task/thread. This should not be used for synchronization (see instead `trylock`).

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. Each `lock` must be matched with an `unlock`.

source

48.3 Channels

`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)
```

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.

`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`.

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> chnl = Channel{Char}(1, spawn=true) do ch
           for c in "hello world"
               put!(ch, c)
           end
       end
Channel{Char}(sz_max:1,sz_curr:1)

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)
```

Remove and return a value from a Channel. Blocks until data is available.

For unbuffered channels, blocks until a put! is performed by a different task.

source

Base.isready – Method.

```
isready(c::Channel)
```

Determine whether a Channel has a value stored to it. Returns immediately, does not block.

For unbuffered channels returns true if there are tasks waiting on a put!.

source

Base.fetch – Method.

```
fetch(c::Channel)
```

Wait for and get the first available item from the channel. Does not remove the item. fetch is unsupported on an unbuffered (0-size) channel.

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: foo
Stacktrace:
[...]
```

source

Chapter 49

Multi-Threading

`Base.Threads.@threads` – Macro.

```
Threads.@threads
```

A macro to parallelize a for-loop to run with multiple threads. This spawns `nthreads()` number of threads, splits the iteration space amongst them, and iterates in parallel. A barrier is placed at the end of the loop which waits for all the threads to finish execution, and the loop returns.

source

`Base.Threads.@spawn` – Macro.

```
Threads.@spawn expr
```

Create and run a `Task` on any available thread. 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.

Note

This feature is currently considered experimental.

Julia 1.3

This macro is available as of Julia 1.3.

source

`Base.Threads.threadid` – Function.

```
Threads.threadid()
```

Get the ID number of the current thread of execution. The master thread has ID 1.

source

`Base.Threads.nthreads` – Function.

```
Threads.nthreads()
```

Get the number of threads available to the Julia process. This is the inclusive upper bound on `threadid()`.

source

49.1 Synchronization

`Base.Threads.Condition` – Type.

```
Threads.Condition([lock])
```

A thread-safe version of `Base.Condition`.

> **Julia 1.2**
> This functionality requires at least Julia 1.2.

source

`Base.Event` – Type.

```
Event()
```

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.

> **Julia 1.1**
> This functionality requires at least Julia 1.1.

source

See also Synchronization.

49.2 Atomic operations

> **Warning**
> The API for atomic operations has not yet been finalized and is likely to change.

`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

49.3 ccall using a 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 main `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

49.4 Low-level synchronization primitives

These building blocks are used to create the regular synchronization objects.

Base.Threads.SpinLock – Type.

```
SpinLock()
```

Create a non-reentrant lock. Recursive use will result in a deadlock. Each `lock` must be matched with an `unlock`.

Test-and-test-and-set spin locks are quickest up to about 30ish contending threads. If you have more contention than that, perhaps a lock is the wrong way to synchronize.

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.

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. For details see 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.

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
```

A string containing the full path to the directory containing the `julia` executable.

source

Base.Sys.CPU_THREADS – Constant.

```
Sys.CPU_THREADS
```

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
```

Standard word size on the current machine, in bits.

source

Base.Sys.KERNEL – Constant.

```
Sys.KERNEL
```

A symbol representing the name of the operating system, as returned by uname of the build configuration.

source

Base.Sys.ARCH – Constant.

```
Sys.ARCH
```

A symbol representing the architecture of the build configuration.

source

Base.Sys.MACHINE – Constant.

```
Sys.MACHINE
```

A string containing the build triple.

source

参见：

- stdin
- stdout
- stderr
- ENV
- ENDIAN_BOM
- Libc.MS_ASYNC
- Libc.MS_INVALIDATE
- Libc.MS_SYNC

Chapter 51

文件系统

Base.Filesystem.pwd – Function.

```
pwd() -> AbstractString
```

Get the current working directory.

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.

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=".") -> Vector{String}
```

Return the files and directories in the directory `dir` (or the current working directory if not given).

Examples

```
julia> readdir("/home/JuliaUser/Projects/julia")
34-element Array{String,1}:
 ".circleci"
 ".freebsdci.sh"
 ".git"
 ".gitattributes"
 ".github"
 
 "test"
 "ui"
 "usr"
 "usr-staging"
```

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` encounters a `SystemError` it will rethrow the error by default. A custom error handling function can be provided through `onerror` keyword argument. `onerror` is called with a `SystemError` as argument.

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 directories in the given `path`, with permissions `mode`. `mode` defaults to `0o777`, modified by the current file creation mask. Return `path`.

Examples

```
julia> mkdir("testingdir")
"testingdir"

julia> cd("testingdir")

julia> pwd()
"/home/JuliaUser/testingdir"

julia> mkpath("my/test/dir")
"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"
```

source

Base.Filesystem.symlink – Function.

```
symlink(target::AbstractString, link::AbstractString)
```

Creates a symbolic link to `target` with the name `link`.

> **Note**
> This function raises an error under operating systems that do not support soft symbolic links, such as Windows XP.

source

Base.Filesystem.readlink – Function.

```
readlink(path::AbstractString) -> AbstractString
```

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`.

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. RawFDs 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)
```

Returns a structure whose fields contain information about the file. The fields of the structure are:

Name	Description
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 was created

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.Filesystem.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`.

source

`Base.download` – Function.

```
download(url::AbstractString, [localfile::AbstractString])
```

Download a file from the given url, optionally renaming it to the given local file name. If no filename is given this will download into a randomly-named file in your temp directory. Note that this function relies on the availability of external tools such as `curl`, `wget` or `fetch` to download the file and is provided for convenience. For production use or situations in which more options are needed, please use a package that provides the desired functionality instead.

Returns the filename of the downloaded file.

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: no such file or directory (ENOENT)
Stacktrace:
[...]
```

source

Base.Filesystem.touch – Function.

```
touch(path::AbstractString)
```

Update the last-modified timestamp on a file to the current time. 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()
```

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.

> **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.

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.

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, any trailing `X` characters 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.

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.

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> f = open("test_file.txt", "w");

julia> isfile(f)
true

julia> close(f); rm("test_file.txt")
```

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() -> AbstractString
```

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) -> AbstractString
```

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) -> AbstractString
```

Get the file name part of a path.

Examples

```
julia> basename("/home/myuser/example.jl")
"example.jl"
```

See also: dirname

source

Base.@__FILE__ – Macro.

```
@__FILE__ -> AbstractString
```

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__ -> AbstractString
```

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.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
```

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.

Examples

```
julia> joinpath("/home/myuser", "example.jl")
"/home/myuser/example.jl"
```

source

`Base.Filesystem.abspath` – Function.

```
abspath(path::AbstractString) -> AbstractString
```

Convert a path to an absolute path by adding the current directory if necessary. Also normalizes the path as in `normpath`.

source

```
abspath(path::AbstractString, paths::AbstractString...) -> AbstractString
```

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) -> AbstractString
```

Normalize a path, removing ”.” and ”..” entries.

Examples

```
julia> normpath("/home/myuser/../example.jl")
"/home/example.jl"
```

source

Base.Filesystem.realpath – Function.

```
realpath(path::AbstractString) -> AbstractString
```

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 = ".") -> AbstractString
```

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`.

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.

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) -> (AbstractString, AbstractString)
```

If the last component of a path contains a dot, split the path into everything before the dot and everything including and after the dot. Otherwise, return a tuple of the argument unmodified and the empty string.

Examples

```
julia> splitext("/home/myuser/example.jl")
("/home/myuser/example", ".jl")

julia> splitext("/home/myuser/example")
("/home/myuser/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 Array{String,1}:
 "/"
 "home"
 "myuser"
 "example.jl"
```

source

Chapter 52

I/O 与网络

52.1 通用 I/O

`Base.stdout` – Constant.

```
stdout
```

Global variable referring to the standard out stream.

source

`Base.stderr` – Constant.

```
stderr
```

Global variable referring to the standard error stream.

source

`Base.stdin` – Constant.

```
stdin
```

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> open("myfile.txt", "w") do io
           write(io, "Hello world!")
       end;

julia> open(f->read(f, String), "myfile.txt")
"Hello world!"

julia> rm("myfile.txt")
```

source

```
open(filename::AbstractString; 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.

source

```
open(filename::AbstractString, [mode::AbstractString]) -> 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

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")
```

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.

Warn

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:

Mode	Description	Keywords
r	read	none
w	write	`write = true`
r+	read, write	`read = true, write = true`
w+	read, write	`read = true, write = true`

source

```
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. Returns the value returned by `f`.

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, without copying. 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.write – Function.

```
write(io::IO, x)
write(filename::AbstractString, 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`).

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

```
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.

```
read(io::IO, String)
```

Read the entirety of `io`, as a `String`.

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, args...)
```

Open a file and read its contents. `args` is passed to `read`: this is equivalent to `open(io->read(io, args...), filename)`.

```
read(filename::AbstractString, String)
```

Read the entire contents of a file as a string.

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::Union{Array, BitArray})
read!(filename::AbstractString, array::Union{Array, BitArray})
```

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

Missing docstring.

Missing docstring for Base.peek. Check Documenter's build log for details.

Base.position – Function.

```
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)
```

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)
```

Remove a mark from stream `s`. Return `true` if the stream was marked, `false` otherwise.

See also mark, reset, ismarked.

source

Base.reset – Function.

```
reset(s)
```

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)
```

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.

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 `true` if the specified IO object is writable (if that can be determined).

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 `true` if the specified IO object is readable (if that can be determined).

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_stdout – Function.

```
redirect_stdout([stream]) -> (rd, wr)
```

Create a pipe to which all C and Julia level stdout output will be redirected. Returns a tuple (rd, wr) representing the pipe ends. Data written to stdout may now be read from the rd end of the pipe. The wr end is given for convenience in case the old stdout object was cached by the user and needs to be replaced elsewhere.

If called with the optional stream argument, then returns stream itself.

Note

`stream` must be a TTY, a `Pipe`, or a socket.

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.

Note

`stream` must be a TTY, a `Pipe`, or a socket.

source

`Base.redirect_stderr` – Function.

```
redirect_stderr([stream]) -> (rd, wr)
```

Like `redirect_stdout`, but for `stderr`.

Note

`stream` must be a TTY, a `Pipe`, or a socket.

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.

Note

`stream` must be a TTY, a `Pipe`, or a socket.

source

`Base.redirect_stdin` – Function.

```
redirect_stdin([stream]) -> (rd, wr)
```

Like `redirect_stdout`, but for `stdin`. Note that the order of the return tuple is still `(rd, wr)`, i.e. data to be read from `stdin` may be written to `wr`.

Note

`stream` must be a TTY, a `Pipe`, or a socket.

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.

> **Note**
>
> `stream` must be a TTY, a `Pipe`, or a socket.

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> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.\n");
       end;

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')
```

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`.

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> countlines(io, eol = '.')
0
```

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 all available data on the stream, blocking the task only if no data is available. The result is a Vector{UInt8,1}.

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 small values should be printed more compactly, e.g. that numbers should be printed with fewer digits. This is set when printing array elements.
- :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, "looooong")
           end
       end
f (generic function with 1 method)

julia> f(stdout)
looooong
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

Missing docstring.

Missing docstring for Base.show(::IO, ::Any). Check Documenter's build log for details.

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 Array{Float64,1}"
```

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.

string returns the output of print as a string.

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 followed by a newline. If io is not supplied, prints to stdout.

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, color::Union{Symbol,Int}=:normal)
```

Print xs in a color specified as a symbol or integer, optionally in bold.

color may take any of the values :normal, :default, :bold, :black, :blink, :blue, :cyan, :green, :hidden, :light_black, :light_blue, :light_cyan, :light_green, :light_magenta, :light_red, :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. If the keyword bold is given as true, the result will be printed in bold.

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 either an IOContext whose properties will be used, or a Pair specifying a property and its value. sizehint suggests the capacity of the buffer (in bytes).

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 f. The optional sizehint is a suggested size (in bytes) to allocate for the buffer used to write the string.

Examples

```
julia> sprint(show, 66.66666; context=:compact => true)
"66.6667"

julia> sprint(showerror, BoundsError([1], 100))
"BoundsError: attempt to access 1-element Array{Int64,1} 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::AbstractString
       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> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.\n");
       end
57

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")
```

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> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.\n");
       end
57

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.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\nIt has many members.\n");
       end
57

julia> readlines("my_file.txt")
2-element Array{String,1}:
 "JuliaLang is a GitHub organization."
 "It has many members."

julia> readlines("my_file.txt", keep=true)
2-element Array{String,1}:
```

```
 "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.

Examples

```
julia> open("my_file.txt", "w") do io
           write(io, "JuliaLang is a GitHub organization.\n It has many members.\n");
       end;

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");
```

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)
```

AbstractDisplay `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).

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
```

Returns 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, 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`.

The default MIME type is `MIME"text/plain"`. There is a fallback definition for `text/plain` output that calls `show` with 2 arguments. Therefore, this case should be handled by defining a 2-argument `show(io::IO, x::MyType)` method.

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 first argument to `show` can be an `IOContext` specifying output format properties. See `IOContext` for details.

source

`Base.Multimedia.showable` – Function.

```
showable(mime, x)
```

Returns 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("img/png", rand(5))
false
```

source

Base.repr – Method.

```
repr(mime, x; context=nothing)
```

Returns 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 Array{Int64,2}:\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)
```

Returns 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("img/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
`@m`	the at-symbol invokes macro `m`; followed by space-separated expressions or a function-call-like argument list
`!`	an exclamation mark is a prefix operator for logical negation (”not”)
`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, `Colon`s represent all indices within a dimension, frequently combined with indexing
`::`	double-colons represent a type annotation or `typeassert`, depending on context, frequently

Chapter 54

排序及相关函数

Julia 拥有为数众多的灵活的 API，用于对已经排序的值数组进行排序和交互。默认情况下，Julia 会选择合理的算法并按标准升序进行排序：

```
julia> sort([2,3,1])
3-element Array{Int64,1}:
 1
 2
 3
```

你同样可以轻松实现逆序排序：

```
julia> sort([2,3,1], rev=true)
3-element Array{Int64,1}:
 3
 2
 1
```

对数组进行 in-place 排序时，要使用！版的排序函数：

```
julia> a = [2,3,1];

julia> sort!(a);

julia> a
3-element Array{Int64,1}:
 1
 2
 3
```

你可以计算用于排列的索引，而不是直接对数组进行排序：

```
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
```

数组可以根据对其值任意的转换结果来进行排序：

```
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
```

所有与排序和顺序相关的函数依赖于“小于”关系，该关系定义了要操纵的值的总顺序。默认情况下会调用 `isless` 函数，但可以通过 `lt` 关键字指定关系。

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. `QuickSort` is used by default for numeric arrays while `MergeSort` is used for other arrays. You can specify an algorithm to use via the `alg` keyword (see Sorting Algorithms for available algorithms). The by keyword lets you provide a function that will be applied to each element before comparison; the `lt` keyword allows providing a custom "less than" function; use `rev=true` to reverse the sorting order. These options are independent and can be used together in all possible combinations: if both by and `lt` are specified, the `lt` function is applied to the result of the by function; `rev=true` reverses whatever ordering specified via the by and `lt` keywords.

Examples

```
julia> v = [3, 1, 2]; sort!(v); v
3-element Array{Int64,1}:
 1
 2
 3

julia> v = [3, 1, 2]; sort!(v, rev = true); v
3-element Array{Int64,1}:
 3
 2
 1

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[1]); v
3-element Array{Tuple{Int64,String},1}:
 (1, "c")
 (2, "b")
 (3, "a")

julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x -> x[2]); v
3-element Array{Tuple{Int64,String},1}:
 (3, "a")
 (2, "b")
 (1, "c")
```

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 `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 Array{Int64,2}:
 4  3
 1  2

julia> sort!(A, dims = 1); A
2×2 Array{Int64,2}:
 1  2
 4  3

julia> sort!(A, dims = 2); A
2×2 Array{Int64,2}:
 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 Array{Int64,1}:
 1
 2
 3

julia> v
3-element Array{Int64,1}:
 3
 1
 2
```

source

```
sort(A; dims::Integer, alg::Algorithm=DEFAULT_UNSTABLE, 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 Array{Int64,2}:
 4  3
 1  2

julia> sort(A, dims = 1)
```

```
2×2 Array{Int64,2}:
 1  2
 4  3

julia> sort(A, dims = 2)
2×2 Array{Int64,2}:
 3  4
 1  2
```

source

Base.sortperm – Function.

```
sortperm(v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward)
```

Return a permutation vector `I` that puts `v[I]` in sorted order. The order is specified using the same keywords as `sort!`. The permutation is guaranteed to be stable even if the sorting algorithm is unstable, meaning that indices of equal elements appear in ascending order.

See also `sortperm!`.

Examples

```
julia> v = [3, 1, 2];

julia> p = sortperm(v)
3-element Array{Int64,1}:
 2
 3
 1

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

source

Base.Sort.InsertionSort – Constant.

```
InsertionSort
```

Indicate that a sorting function should 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 list.

Characteristics:

- *stable*: preserves the ordering of elements which compare equal (e.g. "a" and "A" in a sort of letters which 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 which compare equal (e.g. "a" and "A" in a sort of letters which ignores case).
- *not in-place* in memory.
- *divide-and-conquer* sort strategy.

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 which compare equal (e.g. "a" and "A" in a sort of letters which 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{Int,OrdinalRange}}
```

Indicate that a sorting function should use the partial quick sort algorithm. Partial quick sort returns the smallest k elements sorted from smallest to largest, finding them and sorting them using QuickSort.

Characteristics:

- *not stable*: does not preserve the ordering of elements which compare equal (e.g. "a" and "A" in a sort of letters which ignores case).
- *in-place* in memory.
- *divide-and-conquer*: sort strategy similar to MergeSort.

source

Base.Sort.sortperm! – Function.

```
sortperm!(ix, v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false,
↪  order::Ordering=Forward, initialized::Bool=false)
```

Like sortperm, but accepts a preallocated index vector ix. If initialized is false (the default), ix is initialized to contain the values 1:length(v).

Examples

```
julia> v = [3, 1, 2]; p = zeros(Int, 3);

julia> sortperm!(p, v); p
3-element Array{Int64,1}:
 2
 3
 1

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

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 Array{Int64,2}:
 -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 Array{Int64,2}:
  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 Array{Int64,2}:
  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 Array{Int64,2}:
  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 Array{Int64,2}:
  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 Array{Int64,2}:
 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 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 vector is in sorted order. The `lt`, `by` and `rev` keywords modify what order is considered to be sorted just as they do for `sort`.

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
```

source

Base.Sort.searchsorted – Function.

```
searchsorted(a, x; by=<transform>, lt=<comparison>, rev=false)
```

Return the range of indices of a which compare as equal to x (using binary search) according to the order specified by the by, `lt` and `rev` keywords, assuming that a is already sorted in that order. Return an empty range located at the insertion point if a does not contain values equal to x.

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
```

source

Base.Sort.searchsortedfirst – Function.

```
searchsortedfirst(a, x; by=<transform>, lt=<comparison>, rev=false)
```

Return the index of the first value in a greater than or equal to x, according to the specified order. Return `length(a) + 1` if x is greater than all values in a. a is assumed to be sorted.

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
```

source

Base.Sort.searchsortedlast – Function.

```
searchsortedlast(a, x; by=<transform>, lt=<comparison>, rev=false)
```

Return the index of the last value in a less than or equal to x, according to the specified order. Return `0` if x is less than all values in a. a is assumed to be sorted.

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
```

source

Base.Sort.partialsort! – Function.

```
partialsort!(v, k; by=<transform>, lt=<comparison>, rev=false)
```

Partially sort the vector v in place, according to the order specified by by, lt and rev 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 via a non-stable algorithm. 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! does not fully sort the input array.

Examples

```
julia> a = [1, 2, 4, 3, 4]
5-element Array{Int64,1}:
 1
 2
 4
 3
 4

julia> partialsort!(a, 4)
4

julia> a
5-element Array{Int64,1}:
 1
 2
 3
 4
 4

julia> a = [1, 2, 4, 3, 4]
5-element Array{Int64,1}:
 1
 2
 4
 3
 4

julia> partialsort!(a, 4, rev=true)
2

julia> a
5-element Array{Int64,1}:
 4
 4
```

```
3
2
1
```

source

Base.Sort.partialsort – Function.

```
partialsort(v, k, by=<transform>, lt=<comparison>, rev=false)
```

Variant of partialsort! which 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=<transform>, lt=<comparison>, 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, meaning that indices of equal elements appear in ascending order.

Note that 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(::Array{Int64,1}, 1:3) with eltype Int64:
 2
 4
 3

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

source

Base.Sort.partialsortperm! – Function.

```
partialsortperm!(ix, v, k; by=<transform>, lt=<comparison>, rev=false, initialized=false)
```

Like partialsortperm, but accepts a preallocated index vector ix. If initialized is false (the default), ix is initialized to contain the values 1:length(ix).

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

InsertionSort 是一个在 QuickSort 中使用的时间复杂度为 O(n^2) 的稳定的排序算法，它通常在 n 比较小的时候才具有较高的效率。

QuickSort 是一个内置并且非常快，但是不稳定的时间复杂度为 O(n log n) 的排序算法，例如即使数组两个元素相等的，它们排序之后的顺序也可能和在原数组中顺序不一致。QuickSort 是内置的包括整数和浮点数在内的数字值的默认排序算法。

PartialQuickSort(k) 类似于 QuickSort，但是如果 k 是一个整数，输出数组只排序到索引 k，如果 k 是 OrdinalRange，则输出数组排在 k 范围内。例如：

```
x = rand(1:500, 100)
k = 50
k2 = 50:100
s = sort(x; alg=QuickSort)
ps = sort(x; alg=PartialQuickSort(k))
qs = sort(x; alg=PartialQuickSort(k2))
map(issorted, (s, ps, qs))             # => (true, false, false)
map(x->issorted(x[1:k]), (s, ps, qs))  # => (true, true, false)
map(x->issorted(x[k2]), (s, ps, qs))   # => (true, false, true)
s[1:k] == ps[1:k]                      # => true
s[k2] == qs[k2]                        # => true
```

MergeSort 是一个时间复杂度为 O(n log n) 的稳定但是非 in-place 的算法，它需要一个大小为输入数组一般的临时数组——同时也不像 QuickSort 一样快。MergeSort 是非数值型数据的默认排序算法。

默认排序算法的选择是基于它们的快速稳定，或者 *appear* 之类的。对于数值类型，实际上选择了 QuickSort，因为在这种情况下，它更快，与稳定排序没有区别 (除非数组以某种方式记录了突变)

Julia 选择默认排序算法的机制是通过 Base.Sort.defalg 来实现的，其允许将特定算法注册为特定数组的所有排序函数中的默认值。例如，这有两个默认算法 sort.jl:

```
defalg(v::AbstractArray) = MergeSort
defalg(v::AbstractArray{<:Number}) = QuickSort
```

对于数值型数组，选择非稳定的默认排序算法的原则是稳定的排序算法没有必要的（例如：但两个值相比较时相等且不可区分时）。

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. `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 Array{Char,1}:
 'b'
 'c'
 'd'

julia> collect(a)
2-element Array{Char,1}:
 'e'
 'f'
```

```
julia> a = Iterators.Stateful([1,1,1,2,3,4]);

julia> for x in a; x == 1 || break; end
```

```
julia> Base.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.

Examples

```
julia> a = 1:5
1:5

julia> b = ["e","d","b","c","a"]
5-element Array{String,1}:
 "e"
 "d"
 "b"
 "c"
 "a"

julia> c = zip(a,b)
Base.Iterators.Zip{Tuple{UnitRange{Int64},Array{String,1}}}((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`; it's also possible that `x != iter[i]`, if `iter` has indices that do not start at 1. 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
```

source

Base.Iterators.rest – Function.

```
rest(iter, state)
```

An iterator that yields the same elements as `iter`, but starting at the given `state`.

Examples

```
julia> collect(Iterators.rest([1,2,3,4], 2))
3-element Array{Int64,1}:
 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`.

Examples

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
6-element Array{Int64,1}:
  1
  3
```

```
 5
 7
 9
11

julia> collect(Iterators.take(a,3))
3-element Array{Int64,1}:
 1
 3
 5
```

source

Missing docstring.

Missing docstring for `Base.Iterators.takewhile`. Check Documenter's build log for details.

`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 Array{Int64,1}:
  1
  3
  5
  7
  9
 11

julia> collect(Iterators.drop(a,4))
2-element Array{Int64,1}:
  9
 11
```

source

Missing docstring.

Missing docstring for `Base.Iterators.dropwhile`. Check Documenter's build log for details.

`Base.Iterators.cycle` – Function.

```
cycle(iter)
```

An iterator that cycles through `iter` forever. If `iter` is empty, so is `cycle(iter)`.

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)`).

Examples

```
julia> a = Iterators.repeated([1 2], 4);

julia> collect(a)
4-element Array{Array{Int64,2},1}:
 [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.

Examples

```
julia> collect(Iterators.product(1:2, 3:5))
2×3 Array{Tuple{Int64,Int64},2}:
 (1, 3)  (1, 4)  (1, 5)
 (2, 3)  (2, 4)  (2, 5)
```

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 Array{Int64,1}:
 1
 2
 8
 9
```

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 Array{Array{Int64,1},1}:
 [1, 2]
 [3, 4]
 [5]
```

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),Array{Int64,1}}(isodd, [1, 2, 3, 4, 5])

julia> foreach(println, f)
1
3
5
```

source

Missing docstring.

Missing docstring for `Base.Iterators.accumulate`. Check Documenter's build log for details.

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.

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 `r = Iterators.reverse(itr)` by `r.itr`.)

Examples

```
julia> foreach(println, Iterators.reverse(1:5))
5
4
3
2
1
```

source

Missing docstring.

Missing docstring for `Base.Iterators.only`. Check Documenter's build log for details.

`Base.Iterators.peel` – Function.

```
peel(iter)
```

Returns the first element and an iterator over the remaining elements.

Examples

```
julia> (a, rest) = Iterators.peel("abc");

julia> a
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> collect(rest)
2-element Array{Char,1}:
 'b'
 'c'
```

source

Chapter 56

C 接口

`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`.

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)
```

Load a value of type T from the address of the ith element (1-indexed) starting at p. This is equivalent to the C expression p[i-1].

The unsafe prefix on this function indicates that no validation is performed on the pointer p to ensure that it is valid. Incorrect usage may segfault your program or return garbage answers, in the same manner as C.

source

Base.unsafe_store! – Function.

```
unsafe_store!(p::Ptr{T}, x, i::Integer=1)
```

Store a value of type T to the address of the ith element (1-indexed) starting at p. This is equivalent to the C expression p[i-1] = x.

The unsafe prefix on this function indicates that no validation is performed on the pointer p to ensure that it is valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

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 offset 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.

source

Base.copyto! – Function.

```
copyto!(dest, do, src, so, N)
```

Copy N elements from collection src starting at offset so, to array dest starting at offset 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.

Examples

```
julia> x = [1., 0., 3., 0., 5.];

julia> y = zeros(7);

julia> copyto!(y, x);

julia> y
7-element Array{Float64,1}:
 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.

source

```
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.

`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.

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

Missing docstring.

Missing docstring for `Base.exit_on_sigint`. Check Documenter's build log for details.

`Base.systemerror` – Function.

```
systemerror(sysfunc, iftrue)
```

Raises a `SystemError` for `errno` with the descriptive string `sysfunc` if `iftrue` is `true`

source

`Base.windowserror` – Function.

```
windowserror(sysfunc, iftrue)
```

Like `systemerror`, but for Windows API functions that use `GetLastError` 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`.

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.

There is no invalid (NULL) `Ref` in Julia, but a `C_NULL` instance of `Ptr` can be passed to a `ccall` Ref argument.

Use in broadcasting

Broadcasting with `Ref(x)` treats `x` as a scalar:

```
julia> isa.(Ref([1,2,3]), [Array, Dict, Int])
3-element BitArray{1}:
 1
 0
 0
```

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 Cchars. `Cstrings` are NUL-terminated. For C-style strings composed of the native wide character type, see Cwstring. For more information about string interopability 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_t`s. `Cwstring`s are NUL-terminated. For C-style strings composed of the native character type, see `Cstring`. For more information about string interopability 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 57

LLVM 接口

Core.Intrinsics.llvmcall – Function.

```
llvmcall(IR::String, ReturnType, (ArgumentType1, ...), ArgumentValue1, ...)
llvmcall((declarations::String, IR::String), ReturnType, (ArgumentType1, ...), ArgumentValue1,
↪  ...)
```

Call LLVM IR string in the first argument. Similar to an LLVM function `define` block, arguments are available as consecutive unnamed SSA variables (%0, %1, etc.).

The optional declarations string contains external functions declarations that are necessary for llvm to compile the IR string. Multiple declarations can be passed in by separating them with line breaks.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each `ArgumentValue` to `llvmcall` will be converted to the corresponding `ArgumentType`, by automatic insertion of calls to `unsafe_convert(ArgumentType, cconvert(ArgumentType, ArgumentValue))`. (See also the documentation for `unsafe_convert` and `cconvert` for further details.) In most cases, this simply results in a call to `convert(ArgumentType, ArgumentValue)`.

See `test/llvmcall.jl` for usage examples.

source

Chapter 58

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.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)
```

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 59

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, CodeInfo, Nothing}`
  The MethodInstance containing the execution context (if it could be found).
- `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
```

Returns 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::Union{Ptr{Cvoid}, UInt}) -> 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)
```

Returns the StackTrace with all StackFrames from the provided Module removed.

source

Chapter 60

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 V

Standard Library

Chapter 61

Base64

Base64.Base64 – Module.

```
Base64
```

Functionality for base-64 encoded strings and IO.

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 Array{UInt8,1}:
 0x48
 0x65
 0x6c
 0x6c
 0x6f
 0x21

julia> String(b)
"Hello!"
```

`Base64.stringmime` – Function.

```
stringmime(mime, x; context=nothing)
```

Returns 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 62

CRC32c

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 63

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。

Both Date and DateTime are basically immutable Int64 wrappers. The single `instant` field of either type is actually a `UTInstant{P}` type, which represents a continuously increasing machine timeline based on the UT second [1]. The DateTime type is not aware of time zones (*naive*, in Python parlance), analogous to a *LocalDateTime* in Java 8. Additional time zone functionality can be added through the TimeZones.jl package, which compiles the IANA time zone database. Both Date and DateTime are based on the ISO 8601 standard, which follows the proleptic Gregorian calendar. One note is that the ISO 8601 standard is particular about BC/BCE dates. In general, the last day of the BC/BCE era, 1-12-31 BC/BCE, was followed by 1-1-1 AD/CE, thus no year zero exists. The ISO standard, however, states that 1 BC/BCE is year zero, so `0000-12-31` is the day before `0001-01-01`, and year `-0001` (yes, negative one for the year) is 2 BC/BCE, year `-0002` is 3 BC/BCE, etc.

63.1 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","y-m-d")` or `DateTime("20150101","yyyymmdd")`.

Delimited slots are marked by specifying the delimiter the parser should expect between two subsequent periods; so `"y-m-d"` lets the parser know that between the first and second slots in a date string like `"2014-07-16"`, it should find the `-` character. The `y`, `m`, and `d` characters let the parser know which periods to parse in each slot.

Fixed-width slots are specified by repeating the period character the number of times corresponding to the width with no delimiter between characters. So `"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.

One note on parsing performance: using the `Date(date_string,format_string)` function is fine if only called a few times. If there are many similarly formatted date strings to parse however, it is much more efficient to first create a `Dates.DateFormat`, and pass it instead of a raw format string.

```
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
```

You can also use the `dateformat""` string macro. This macro creates the `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
```

A full suite of parsing and formatting tests and examples is available in `stdlib/Dates/test/io.jl`.

63.2 持续时间/比较

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
```

63.3 访问函数

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
```

当首字母大写时会返回对应 `Period` 类型的相同值：

```
julia> Dates.Year(t)
2014 years

julia> Dates.Day(t)
31 days
```

Compound methods are provided, as they provide a measure of efficiency if multiple fields are needed at the same time:

```
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
```

63.4 查询函数

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
```

一年中的月份：

```
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"
```

自从缩写版本的 days 函数不加载之后，试图访问函数 dayabbr 将导致一个错误。

```
julia> Dates.dayabbr(t;locale="french")
ERROR: BoundsError: attempt to access 1-element Array{String,1} at index [5]
Stacktrace:
[...]
```

63.5 TimeType 时间运算

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? I like to think of 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.).

另外，所有时间运算都可以与范围一起使用：

```
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 Array{Date,1}:
 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 Array{Date,1}:
 2014-01-29
 2014-02-28
 2014-03-29
```

```
2014-04-29
2014-05-29
2014-06-29
2014-07-29
```

63.6 调整器函数

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; # 当 x 是周二时返回 true

julia> Dates.tonext(istuesday, Date(2014,7,13)) # 2014-07-13 是周日
2014-07-15

julia> Dates.tonext(Date(2014,7,13), Dates.Tuesday) # 星期调整的便捷方法
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
           # 在十一月的第四个星期四——感恩节那天返回 true
           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 Array{Date,1}:
 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`.

63.7 时间段类型

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
```

63.8 取整

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 64

API reference

64.1 日期和时间类型

`Dates.Period` – Type.

```
Period
Year
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.

64.2 日期函数

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 -> Dates.second(dt) == 40, 2010, 10, 20, 10; step = Dates.Second(1))
2010-10-20T10:00:40

julia> DateTime(dt -> Dates.hour(dt) == 20, 2010, 10, 20, 10; step = Dates.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.

This method creates a DateFormat object each time it is called. If you are parsing many date time strings of the same format, consider creating a DateFormat object once and using that as the second argument instead.

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	Returns year of 1996, 0096. Equivalent to y
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. see @dateformat_str.

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) -> DateTime
```

Construct a DateTime by parsing the dt date time string following the pattern given in the DateFormat object. 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 -> Dates.week(date) == 20, 2010, 01, 01)
2010-05-17

julia> Date(date -> Dates.year(date) == 2010, 2000, 01, 01)
2010-01-01

julia> Date(date -> Dates.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.

This method creates a DateFormat object each time it is called. If you are parsing many date strings of the same format, consider creating a DateFormat object once and using that as the second argument instead.

Dates.Date – Method.

```
Date(d::AbstractString, df::DateFormat) -> Date
```

Parse a date from a date string d using a DateFormat object df.

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> Dates.Time(t -> Dates.minute(t) == 30, 20)
20:30:00

julia> Dates.Time(t -> Dates.minute(t) == 0, 20)
20:00:00

julia> Dates.Time(t -> Dates.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.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.

`Base.eps` – Function.

```
eps(::DateTime) -> Millisecond
eps(::Date) -> Day
eps(::Time) -> Nanosecond
```

Returns `Millisecond(1)` for `DateTime` values, `Day(1)` for `Date` values, and `Nanosecond(1)` for `Time` values.

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> Dates.week(Date(1989, 6, 22))
25

julia> Dates.week(Date(2005, 1, 1))
53

julia> Dates.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> Dates.dayname(Date("2000-01-01"))
"Saturday"

julia> Dates.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> Dates.dayabbr(Date("2000-01-01"))
"Sat"

julia> Dates.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> Dates.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> Dates.dayofweekofmonth(Date("2000-02-01"))
1

julia> Dates.dayofweekofmonth(Date("2000-02-08"))
2

julia> Dates.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> Dates.daysofweekinmonth(Date("2005-01-01"))
5

julia> Dates.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> Dates.monthname(Date("2005-01-04"))
"January"

julia> Dates.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> Dates.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> Dates.daysinmonth(Date("2000-01"))
31

julia> Dates.daysinmonth(Date("2001-02"))
28

julia> Dates.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> Dates.isleapyear(Date("2004"))
true

julia> Dates.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> Dates.daysinyear(1999)
365

julia> Dates.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(Dates.DateTime("1996-01-01T12:30:00"), Dates.Day)
1996-01-01T00:00:00
```

Dates.firstdayofweek – Function.

```
firstdayofweek(dt::TimeType) -> TimeType
```

Adjusts dt to the Monday of its week.

Examples

```
julia> Dates.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> Dates.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> Dates.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> Dates.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> Dates.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> Dates.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> Dates.firstdayofquarter(DateTime("1996-05-20"))
1996-04-01T00:00:00

julia> Dates.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> Dates.lastdayofquarter(DateTime("1996-05-20"))
1996-06-30T00:00:00

julia> Dates.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)
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 Periods. All Periods 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.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
```

Returns 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.

取整函数

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), Dates.Month)
1985-08-01

julia> floor(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:30:00

julia> floor(DateTime(2016, 8, 6, 12, 0, 0), Dates.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), Dates.Month)
1985-09-01

julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:45:00

julia> ceil(DateTime(2016, 8, 6, 12, 0, 0), Dates.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), Dates.Month)
1985-08-01

julia> round(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))
2013-02-13T00:30:00

julia> round(DateTime(2016, 8, 6, 12, 0, 0), Dates.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(Dates.Day(16), Dates.Week)
2 weeks

julia> floor(Dates.Minute(44), Dates.Minute(15))
30 minutes

julia> floor(Dates.Hour(36), Dates.Day)
1 day
```

Rounding to a precision of Months or Years is not supported, as these Periods 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(Dates.Day(16), Dates.Week)
3 weeks

julia> ceil(Dates.Minute(44), Dates.Minute(15))
45 minutes

julia> ceil(Dates.Hour(36), Dates.Day)
2 days
```

Rounding to a precision of Months or Years is not supported, as these Periods 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(Dates.Day(16), Dates.Week)
2 weeks

julia> round(Dates.Minute(44), Dates.Minute(15))
45 minutes

julia> round(Dates.Hour(36), Dates.Day)
2 days
```

Valid rounding modes for round(::Period, ::T, ::RoundingMode) are RoundNearestTiesUp (default), RoundDown (floor), and RoundUp (ceil).

Rounding to a precision of Months or Years is not supported, as these Periods 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`.

转换函数

`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`.

常量

Days of the Week:

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

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

Chapter 65

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. Default is `true` except on Windows. On Windows, you may want to specify `true` if the file is large, and is only read once and not written to.

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 Array{Int64,2}:
 1  5
 2  6
 3  7
 4  8
```

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.

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 Array{Float64,2}:
 1.0  1.1
 2.0  2.2
 3.0  3.3
 4.0  4.4

julia> rm("delim_file.txt")
```

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 Array{Float64,2}:
 1.0  1.1
 2.0  2.2
 3.0  3.3
 4.0  4.4

julia> rm("delim_file.txt")

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 Array{Any,2}:
 1  "a"
 2  "b"
 3  "c"
 4  "d"

julia> rm("delim_file.txt")
```

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 Array{Int64,2}:
 1  5
 2  6
 3  7
 4  8

julia> readdlm("delim_file.txt", Float64)
4×2 Array{Float64,2}:
 1.0  5.0
 2.0  6.0
 3.0  7.0
```

```
 4.0  8.0

julia> rm("delim_file.txt")
```

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 Array{Any,2}:
 1  "a"
 2  "b"
 3  "c"
 4  "d"

julia> rm("delim_file.txt")
```

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 Array{Int64,2}:
 1  5
 2  6
 3  7
 4  8

julia> rm("delim_file.txt")
```

Chapter 66

分布式计算

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
```

```
addprocs(machines; tunnel=false, sshflags=``, max_parallel=10, kwargs...) -> List of process
    identifiers
```

Add processes on remote machines via SSH. Requires `julia` to be installed in the same location on each node, or to be available via a shared file system.

`machines` is a vector of machine specifications. Workers are started for each specification.

A machine specification is either a string `machine_spec` or a tuple - `(machine_spec, count)`.

`machine_spec` is a string of the form `[user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user, `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`.

`count` is the number of workers to be launched on the specified host. If specified as `:auto` it will launch as many workers as the number of CPU threads on the specific host.

Keyword arguments:

- `tunnel`: if `true` then SSH tunneling will be used to connect to the worker from the master process. Default is `false`.
- `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.
- `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.
- `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.

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.

```
addprocs(; kwargs...) -> List of process identifiers
```

Equivalent to `addprocs(Sys.CPU_THREADS; kwargs...)`

Note that workers do not run a `.julia/config/startup.jl` startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

```
addprocs(np::Integer; restrict=true, kwargs...) -> List of process identifiers
```

Launches workers using the in-built `LocalManager` which only launches workers on the local host. This can be used to take advantage of multiple cores. `addprocs(4)` will add 4 processes on the local machine. If `restrict` is `true`, binding is restricted to `127.0.0.1`. Keyword args `dir`, `exename`, `exeflags`, `topology`, `lazy` and `enable_threaded_blas` have the same effect as documented for `addprocs(machines)`.

`Distributed.nprocs` – Function.

```
nprocs()
```

Get the number of available processes.

Examples

```
julia> nprocs()
3

julia> workers()
5-element Array{Int64,1}:
 2
 3
```

`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 5

julia> nprocs()
6

julia> nworkers()
5
```

`Distributed.procs` – Method.

```
procs()
```

Return a list of all process identifiers, including pid 1 (which is not included by `workers()`).

Examples

```
$ julia -p 5

julia> procs()
3-element Array{Int64,1}:
 1
 2
 3
```

`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.

`Distributed.workers` – Function.

```
workers()
```

Return a list of all worker process identifiers.

Examples

```
$ julia -p 5

julia> workers()
2-element Array{Int64,1}:
 2
 3
```

`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
```

`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.

```
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.

`Distributed.myid` – Function.

```
myid()
```

Get the id of the current process.

Examples

```
julia> myid()
1

julia> remotecall_fetch(() -> myid(), 4)
4
```

`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))
```

`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.

`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`.

`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.

`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.

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.

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.

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.

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 will only return a complex result if called with a complex argument. Try sqrt(Complex(x)).
...
```

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.

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.

Base.put! – Method.

```
put!(rr::Future, v)
```

Store a value to a Future rr. Futures are write-once remote references. A put! on an already set Future throws an Exception. All asynchronous remote calls return Futures and set the value to the return value of the call upon completion.

Base.take! – Method.

```
take!(rr::RemoteChannel, args...)
```

Fetch value(s) from a RemoteChannel rr, removing the value(s) in the process.

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.

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)
@async put!(f, remotecall_fetch(long_computation, p))
isready(f)  # will not block
```

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.

`Distributed.WorkerPool` – Type.

```
WorkerPool(workers::Vector{Int})
```

Create a `WorkerPool` from a vector 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))
```

`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(wp, i -> sum(foo) + i, 1:100);
end
```

The above would transfer `foo` only once to each worker.

`Distributed.default_worker_pool` – Function.

```
default_worker_pool()
```

`WorkerPool` containing idle `workers` - used by `remote(f)` and `pmap` (by default).

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))
```

`Distributed.clear!` – Method.

```
clear!(pool::CachingPool) -> pool
```

Removes all cached functions from all participating workers.

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`.

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.

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
```

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
```

`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.

`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.

`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
```

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
```

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
```

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 processes.

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.

Equivalent to calling `remotecall_eval(Main, procs, expr)`.

`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.

`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.

`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.

`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.

Distributed.cluster_cookie – Method.

```
cluster_cookie() -> cookie
```

Return the cluster cookie.

Distributed.cluster_cookie – Method.

```
cluster_cookie(cookie) -> cookie
```

Set the passed cookie as the cluster cookie, then returns it.

66.1 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.

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 an `AbstractString` 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-debug"`
- `exeflags` –flags to use when lauching 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)
- `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 `LocalManager`s and `SSHManager`s:

- `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

`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.

`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.

`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`.

`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.

`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`.

`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.

`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`.

Chapter 67

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.

The returned value is an object with boolean fields `changed`, `renamed`, 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 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 changed, renamed, 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 68

交互式组件

Base.Docs.apropos – Function.

```
apropos(string)
```

Search through all documentation for a string, ignoring case.

InteractiveUtils.varinfo – Function.

```
varinfo(m::Module=Main, pattern::Regex=r"")
```

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.

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

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.

Examples

```
julia> subtypes(Integer)
3-element Array{Any,1}:
 Bool
 Signed
 Unsigned
```

Missing docstring.

Missing docstring for `InteractiveUtils.supertypes`. Check Documenter's build log for details.

`InteractiveUtils.edit` – Method.

```
edit(path::AbstractString, line::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.

`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.

The editor can be changed by setting `JULIA_EDITOR`, `VISUAL` or `EDITOR` as an environment variable.

`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.

Missing docstring.

Missing docstring for `InteractiveUtils.define_editor`. Check Documenter's build log for details.

`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.

`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.

`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 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, so the results need to be used judiciously. In particular, unions containing either `missing` or `nothing` are displayed in yellow, since these are often intentional.

Keyword argument `debuginfo` may be one of `:source` or `:none` (default), to specify the verbosity of code comments.

See `@code_warntype` 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=:att, debuginfo=:default)
```

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to io. Switch assembly syntax using syntax symbol parameter set to :att for AT&T syntax or :intel for Intel syntax. Keyword argument debuginfo may be one of source (default) or none, to specify the verbosity of code comments.

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 the optional keyword argument debuginfo by putting it before the function call, like this:

```
@code_native debuginfo=:default f(x)
```

debuginfo may be one of :source (default) or :none, to specify the verbosity of code comments.

InteractiveUtils.clipboard – Function.

```
clipboard(x)
```

Send a printed form of x to the operating system clipboard (”copy”).

```
clipboard() -> AbstractString
```

Return a string with the contents of the operating system clipboard (”paste”).

Chapter 69

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.

`source`

`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).

source

`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.

source

`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"`.

source

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, comittish::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.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 `map` and `count` (for instance, `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
    count((oid,repo)->(oid == commit_oid1), walker, oid=commit_oid1,
    ↪  by=LibGit2.Consts.SORT_TIME)
end
```

Here, `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.

source

`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.

source

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)
```

source

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.

source

`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.

source

`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.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.init(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
    count((oid, repo)->(oid == commit_oid1), walker, oid=commit_oid1,
    ↪  by=LibGit2.Consts.SORT_TIME)
end
```

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 = LibGit2.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 = LibGit2.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 `GitHash`s 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, `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 `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 = LibGit2.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.

Chapter 70

动态链接器

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`.

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)`.

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.

Libdl.dlsym – Function.

```
dlsym(handle, sym)
```

Look up a symbol from a shared library handle, return callable function pointer on success.

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).

Libdl.dlclose – Function.

```
dlclose(handle)
```

Close shared library referenced by handle.

```
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.

Libdl.dlext – Constant.

```
dlext
```

File extension for dynamic libraries (e.g. dll, dylib, so) on the current platform.

Libdl.dllist – Function.

```
dllist()
```

Return the paths of dynamic libraries currently loaded in a Vector{String}.

Libdl.dlpath – Function.

```
dlpath(handle::Ptr{Cvoid})
```

Given a library handle from dlopen, return the full path.

```
dlpath(libname::Union{AbstractString, Symbol})
```

Get the full path of the library libname.

Example

```
julia> dlpath("libjulia")
```

`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.

`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.

Chapter 71

线性代数

In addition to (and as part of) its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations which can be loaded with `using LinearAlgebra`. Basic operations, such as `tr`, `det`, and `inv` are all supported:

```
julia> A = [1 2 3; 4 1 6; 7 8 1]
3×3 Array{Int64,2}:
 1  2  3
 4  1  6
 7  8  1

julia> tr(A)
3

julia> det(A)
104.0

julia> inv(A)
3×3 Array{Float64,2}:
 -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 Array{Float64,2}:
 -4.0  -17.0
  2.0    2.0

julia> eigvals(A)
2-element Array{Complex{Float64},1}:
 -1.0 - 5.0im
 -1.0 + 5.0im

julia> eigvecs(A)
2×2 Array{Complex{Float64},2}:
  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，它们可用于加快问题的求解，比如线性求解或矩阵或矩阵求幂，这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）。有关的更多信息，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 Array{Float64,2}:
   1.5   2.0  -4.0
   3.0  -1.0  -6.0
 -10.0   2.3   4.0

julia> factorize(A)
LU{Float64,Array{Float64,2}}
L factor:
3×3 Array{Float64,2}:
  1.0    0.0       0.0
 -0.15   1.0       0.0
 -0.3   -0.132196  1.0
U factor:
3×3 Array{Float64,2}:
 -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 Array{Float64,2}:
  1.5   2.0  -4.0
  2.0  -1.0  -3.0
 -4.0  -3.0   5.0

julia> factorize(B)
BunchKaufman{Float64,Array{Float64,2}}
D factor:
3×3 Tridiagonal{Float64,Array{Float64,1}}:
 -1.64286   0.0   ⋅
  0.0      -2.8  0.0
   ⋅        0.0  5.0
U factor:
3×3 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  0.142857  -0.8
  ⋅   1.0       -0.6
  ⋅    ⋅         1.0
permutation:
3-element Array{Int64,1}:
 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 Array{Float64,2}:
  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,Array{Float64,2}}:
  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 Array{Float64,2}:
  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,Array{Float64,2}}:
  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 Array{Int64,1}:
 1
 2
 3

julia> sB\x
3-element Array{Float64,1}:
 -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。

71.1 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。

Elementary operations

Legend:

类型	描述
Symmetric	对称矩阵
Hermitian	埃尔米特矩阵
UpperTriangular	上三角矩阵
UnitUpperTriangular	Upper triangular matrix with unit diagonal
LowerTriangular	Lower triangular matrix
UnitLowerTriangular	Lower triangular matrix with unit diagonal
UpperHessenberg	Upper Hessenberg matrix
Tridiagonal	Tridiagonal matrix
SymTridiagonal	Symmetric tridiagonal matrix
Bidiagonal	Upper/lower bidiagonal matrix
Diagonal	Diagonal matrix
UniformScaling	Uniform scaling operator

Matrix type	+	-	*	\	Other functions with optimized methods
Symmetric				MV	inv, sqrt, exp
Hermitian				MV	inv, sqrt, exp
UpperTriangular			MV	MV	inv, det
UnitUpperTriangular			MV	MV	inv, det
LowerTriangular			MV	MV	inv, det
UnitLowerTriangular			MV	MV	inv, det
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	Description
M (matrix)	An optimized method for matrix-matrix operations is available
V (vector)	An optimized method for matrix-vector operations is available
S (scalar)	An optimized method for matrix-scalar operations is available

Matrix factorizations

Legend:

The uniform scaling operator

A `UniformScaling` operator represents a scalar times the identity operator, `λ*I`. The identity operator `I` is defined as a constant and is an instance of `UniformScaling`. The size of these operators are generic and match the other matrix in the binary operations `+`, `-`, `*` and `\`. For `A+I` and `A-I` this means that `A` must be square. Multiplication with the identity operator `I` is a noop (except for checking that the scaling factor is one) and therefore almost without overhead.

To see the `UniformScaling` operator in action:

```
julia> U = UniformScaling(2);

julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
```

Matrix type	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			

Key	Description	Example
A (all)	An optimized method to find all the characteristic values and/or vectors is available	e.g. `eigvals(M)`
R (range)	An optimized method to find the `il`th through the `ih`th characteristic values are available	`eigvals(M, il, ih)`
I (interval)	An optimized method to find the characteristic values in the interval `[vl, vh]` is available	`eigvals(M, vl, vh)`
V (vectors)	An optimized method to find the characteristic vectors corresponding to the characteristic values `x=[x1, x2,...]` is available	`eigvecs(M, x)`

```
 1  2
 3  4

julia> a + U
2×2 Array{Int64,2}:
 3  2
 3  6

julia> a * U
2×2 Array{Int64,2}:
 2  4
 6  8

julia> [a U]
2×4 Array{Int64,2}:
 1  2  2  0
 3  4  0  2

julia> b = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 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.

71.2 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 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

71.3 Standard functions

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK. Sparse factorizations call functions from SuiteSparse.

Base.:* – Method.

```
*(A::AbstractMatrix, B::AbstractMatrix)
```

Matrix multiplication.

Examples

```
julia> [1 1; 0 1] * [1 0; 1 1]
2×2 Array{Int64,2}:
 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.

Examples

```
julia> A = [1 0; 1 -2]; B = [32; -4];

julia> X = A \ B
2-element Array{Float64,1}:
 32.0
 18.0

julia> A * X == 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.

Missing docstring.

Missing docstring for LinearAlgebra.ZeroPivotException. Check Documenter's build log for details.

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.cross` – Function.

```
cross(x, y)
×(x,y)
```

Compute the cross product of two 3-vectors.

Examples

```
julia> a = [0;1;0]
3-element Array{Int64,1}:
 0
 1
 0

julia> b = [0;0;1]
3-element Array{Int64,1}:
 0
 0
 1

julia> cross(a,b)
3-element Array{Int64,1}:
 1
 0
 0
```

`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` 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`.

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)

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 Array{Float64,2}:
 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,Array{Float64,1}}:
 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(A::AbstractMatrix)
```

Construct a matrix from the diagonal of `A`.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> Diagonal(A)
3×3 Diagonal{Int64,Array{Int64,1}}:
 1  ·  ·
 ·  5  ·
 ·  ·  9
```

```
Diagonal(V::AbstractVector)
```

Construct a matrix with `V` as its diagonal.

Examples

```
julia> V = [1, 2]
2-element Array{Int64,1}:
 1
 2

julia> Diagonal(V)
2×2 Diagonal{Int64,Array{Int64,1}}:
 1  ·
 ·  2
```

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 Array{Int64,1}:
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> Bu = Bidiagonal(dv, ev, :U) # ev is on the first superdiagonal
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1  7  ·  ·
 ·  2  8  ·
 ·  ·  3  9
 ·  ·  ·  4

julia> Bl = Bidiagonal(dv, ev, :L) # ev is on the first subdiagonal
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 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 Array{Int64,2}:
 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,Array{Int64,1}}:
 1  1  ·  ·
 ·  2  2  ·
 ·  ·  3  3
 ·  ·  ·  4

julia> Bidiagonal(A, :L) # contains the main diagonal and first subdiagonal of A
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 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).

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> SymTridiagonal(dv, ev)
4×4 SymTridiagonal{Int64,Array{Int64,1}}:
 1  7  ·  ·
 7  2  8  ·
 ·  8  3  9
 ·  ·  9  4
```

```
SymTridiagonal(A::AbstractMatrix)
```

Construct a symmetric tridiagonal matrix from the diagonal and first sub/super-diagonal, of the symmetric matrix A.

Examples

```
julia> A = [1 2 3; 2 4 5; 3 5 6]
3×3 Array{Int64,2}:
 1  2  3
 2  4  5
 3  5  6

julia> SymTridiagonal(A)
3×3 SymTridiagonal{Int64,Array{Int64,1}}:
 1  2  ·
 2  4  5
 ·  5  6
```

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,Array{Int64,1}}:
 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 Array{Int64,2}:
 1  2  3  4
 1  2  3  4
 1  2  3  4
 1  2  3  4

julia> Tridiagonal(A)
```

```
4×4 Tridiagonal{Int64,Array{Int64,1}}:
 1  2  ·  ·
 1  2  3  ·
 ·  2  3  4
 ·  ·  3  4
```

LinearAlgebra.Symmetric – Type.

```
Symmetric(A, uplo=:U)
```

Construct a Symmetric view of the upper (if uplo = :U) or lower (if uplo = :L) triangle 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 1 0; 2 0 3 0 4]
5×5 Array{Int64,2}:
 1  0  2  0  3
 0  4  0  5  0
 6  0  7  0  8
 0  9  0  1  0
 2  0  3  0  4

julia> Supper = Symmetric(A)
5×5 Symmetric{Int64,Array{Int64,2}}:
 1  0  2  0  3
 0  4  0  5  0
 2  0  7  0  8
 0  5  0  1  0
 3  0  8  0  4

julia> Slower = Symmetric(A, :L)
5×5 Symmetric{Int64,Array{Int64,2}}:
 1  0  6  0  2
 0  4  0  9  0
 6  0  7  0  3
 0  9  0  1  0
 2  0  3  0  4
```

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, uplo=:U)
```

Construct a Hermitian view of the upper (if uplo = :U) or lower (if uplo = :L) triangle of the matrix A.

Examples

```
julia> A = [1 0 2+2im 0 3-3im; 0 4 0 5 0; 6-6im 0 7 0 8+8im; 0 9 0 1 0; 2+2im 0 3-3im 0 4];

julia> Hupper = Hermitian(A)
5×5 Hermitian{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0+0im  2+2im  0+0im  3-3im
 0+0im  4+0im  0+0im  5+0im  0+0im
 2-2im  0+0im  7+0im  0+0im  8+8im
 0+0im  5+0im  0+0im  1+0im  0+0im
 3+3im  0+0im  8-8im  0+0im  4+0im
```

```
julia> Hlower = Hermitian(A, :L)
5×5 Hermitian{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0+0im  6+6im  0+0im  2-2im
 0+0im  4+0im  0+0im  9+0im  0+0im
 6-6im  0+0im  7+0im  0+0im  3+3im
 0+0im  9+0im  0+0im  1+0im  0+0im
 2+2im  0+0im  3-3im  0+0im  4+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 Array{Float64,2}:
 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,Array{Float64,2}}:
 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 Array{Float64,2}:
 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,Array{Float64,2}}:
 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 Array{Float64,2}:
 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,Array{Float64,2}}:
 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 Array{Float64,2}:
 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,Array{Float64,2}}:
 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.

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 Array{Int64,2}:
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16

julia> UpperHessenberg(A)
4×4 UpperHessenberg{Int64,Array{Int64,2}}:
 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. See also I.

Examples

```
julia> J = UniformScaling(2.)
UniformScaling{Float64}
2.0*I

julia> A = [1. 2.; 3. 4.]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> J*A
2×2 Array{Float64,2}:
 2.0  4.0
 6.0  8.0
```

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 Array{Complex{Int64},2}:
 1+0im  0+2im  3+0im
 0+1im  2+0im  3+0im
```

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 Array{Int64,2}:
 4  3
 6  3

julia> F = lu(A)
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Array{Float64,2}:
 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, pivot=Val(true); 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}}. If pivoting is chosen (default) the element type should also support abs and <.

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:

Supported function	LU	LU{T,Tridiagonal{T}}
/	✓	
\	✓	✓
inv	✓	✓
det	✓	✓
logdet	✓	✓
logabsdet	✓	✓
size	✓	✓

Examples

```
julia> A = [4 3; 6 3]
2×2 Array{Int64,2}:
 4  3
 6  3

julia> F = lu(A)
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Array{Float64,2}:
 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, pivot=Val(true); 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 Array{Float64,2}:
 4.0  3.0
 6.0  3.0

julia> F = lu!(A)
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0       0.0
 0.666667  1.0
U factor:
2×2 Array{Float64,2}:
 6.0  3.0
 0.0  1.0

julia> iA = [4 3; 6 3]
2×2 Array{Int64,2}:
 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`.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A)
Cholesky{Float64,Array{Float64,2}}
U factor:
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.U
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
```

```
  ·    ·    3.0

julia> C.L
3×3 LowerTriangular{Float64,Array{Float64,2}}:
  2.0   ·    ·
  6.0  1.0   ·
 -8.0  5.0  3.0

julia> C.L * C.U == A
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(_, Val(true)), the corresponding matrix factorization function.

The triangular Cholesky factor can be obtained from the factorization F::CholeskyPivoted via F.L and F.U.

Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]
3×3 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A, Val(true))
CholeskyPivoted{Float64,Array{Float64,2}}
U factor with rank 3:
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 9.89949  -4.34366  -1.61624
  ·        4.25825   1.1694
  ·         ·        0.142334
permutation:
3-element Array{Int64,1}:
 3
 2
 1
```

LinearAlgebra.cholesky – Function.

```
cholesky(A, Val(false); 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 StridedMatrix or a *perfectly* symmetric or Hermitian StridedMatrix. The triangular Cholesky factor can be obtained from the factorization F with: F.L and F.U. The following functions are available for Cholesky objects: size, \, inv, det, logdet and isposdef.

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 Array{Float64,2}:
   4.0   12.0  -16.0
  12.0   37.0  -43.0
 -16.0  -43.0   98.0

julia> C = cholesky(A)
Cholesky{Float64,Array{Float64,2}}
U factor:
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.U
3×3 UpperTriangular{Float64,Array{Float64,2}}:
 2.0  6.0  -8.0
  ·   1.0   5.0
  ·    ·    3.0

julia> C.L
3×3 LowerTriangular{Float64,Array{Float64,2}}:
  2.0   ·    ·
  6.0  1.0   ·
 -8.0  5.0  3.0

julia> C.L * C.U == A
true
```

```
cholesky(A, Val(true); 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` `StridedMatrix` or a *perfectly* symmetric or Hermitian `StridedMatrix`. The triangular Cholesky factor can be obtained from the factorization `F` with: `F.L` and `F.U`. 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.

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.

`LinearAlgebra.cholesky!` – Function.

```
cholesky!(A, Val(false); 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 Array{Int64,2}:
 1   2
 2  50
```

```
julia> cholesky!(A)
ERROR: InexactError: Int64(6.782329983125268)
Stacktrace:
[...]
```

```
cholesky!(A, Val(true); 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.

LinearAlgebra.lowrankupdate – Function.

```
lowrankupdate(C::Cholesky, v::StridedVector) -> 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.

LinearAlgebra.lowrankdowndate – Function.

```
lowrankdowndate(C::Cholesky, v::StridedVector) -> 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.

LinearAlgebra.lowrankupdate! – Function.

```
lowrankupdate!(C::Cholesky, v::StridedVector) -> 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.

LinearAlgebra.lowrankdowndate! – Function.

```
lowrankdowndate!(C::Cholesky, v::StridedVector) -> 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.

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.

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 3.0  1.0   ·
 1.0  4.0  2.0
  ·   2.0  5.0
```

```
julia> F = ldlt(S)
LDLt{Float64,SymTridiagonal{Float64,Array{Float64,1}}}([3.0 0.3333333333333333 0.0;
↪  0.3333333333333333 3.6666666666666665 0.5454545454545455; 0.0 0.5454545454545455
↪  3.909090909090909])
```

LinearAlgebra.ldlt – Function.

```
ldlt(S::SymTridiagonal) -> LDLt
```

Compute an LDLt 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`.

Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 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 Array{Float64,1}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255

julia> S \ b
3-element Array{Float64,1}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255
```

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,Array{Float64,1}}:
 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,Array{Float64,1}}:
 3.0       0.333333   ·
 0.333333  3.66667    0.545455
  ·        0.545455   3.90909
```

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 i th 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 [1], as a lower trapezoidal matrix V and an upper triangular matrix T where

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T) = I - VTV^T$$

such that v_i is the i th column of V , and au_i is the i th diagonal element of T .

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 square matrix with `min(m,n)` columns, whose upper triangular part gives the matrix T above (the subdiagonal elements are ignored).

  Note

  This format should not to be confused with the older *WY* representation [2].

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 i th 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.

```
qr(A, pivot=Val(false)) -> F
```

[2] 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

[1] 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

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

$$A = QR$$

The returned object F stores the factorization in a packed format:

- if `pivot == Val(true)` 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*Matrix(I,m,m)`. If `m<=n`, then `Matrix(F.Q)` yields an `m×m` orthogonal matrix.

Examples

```
julia> A = [3.0 -6.0; 4.0 -8.0; 0.0 1.0]
3×2 Array{Float64,2}:
 3.0  -6.0
 4.0  -8.0
 0.0   1.0

julia> F = qr(A)
LinearAlgebra.QRCompactWY{Float64,Array{Float64,2}}
Q factor:
3×3 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
 -0.6   0.0   0.8
 -0.8   0.0  -0.6
  0.0  -1.0   0.0
R factor:
2×2 Array{Float64,2}:
 -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=Val(false))
```

qr! is the same as qr when A is a subtype of StridedMatrix, 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.; 3. 4.]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> qr!(a)
LinearAlgebra.QRCompactWY{Float64,Array{Float64,2}}
Q factor:
2×2 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
 -0.316228  -0.948683
 -0.948683   0.316228
R factor:
2×2 Array{Float64,2}:
 -3.16228  -4.42719
  0.0      -0.632456

julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 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

“‘jldoctest julia> A = [5. 7.; -2. -4.] 2×2 Array{Float64,2}: 5.0 7.0 -2.0 -4.0

julia> S = lq(A) LQ{Float64,Array{Float64,2}} with factors L and Q: [-8.60233 0.0; 4.41741 -0.697486] [-0.581238 -0.813733; -0.813733 0.581238]

julia> S.L * S.Q 2×2 Array{Float64,2}: 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).

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> S = lq(A)
LQ{Float64,Array{Float64,2}} with factors L and Q:
[-8.60233 0.0; 4.41741 -0.697486]
[-0.581238 -0.813733; -0.813733 0.581238]

julia> S.L * S.Q
2×2 Array{Float64,2}:
  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 Array{Int64,2}:
 1  2
 2  3

julia> S = bunchkaufman(A) # A gets wrapped internally by Symmetric(A)
BunchKaufman{Float64,Array{Float64,2}}
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 -0.333333  0.0
  0.0       3.0
U factor:
2×2 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  0.666667
  ·   1.0
permutation:
2-element Array{Int64,1}:
 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,Array{Float64,2}}
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 3.0   0.0
 0.0  -0.333333
L factor:
2×2 UnitLowerTriangular{Float64,Array{Float64,2}}:
 1.0        ·
 0.666667  1.0
permutation:
2-element Array{Int64,1}:
 2
 1
```

`LinearAlgebra.bunchkaufman` – Function.

```
bunchkaufman(A, rook::Bool=false; check = true) -> S::BunchKaufman
```

Compute the Bunch-Kaufman [3] 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 Array{Int64,2}:
 1  2
 2  3

julia> S = bunchkaufman(A) # A gets wrapped internally by Symmetric(A)
BunchKaufman{Float64,Array{Float64,2}}
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 -0.333333  0.0
  0.0       3.0
U factor:
2×2 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  0.666667
  ·   1.0
permutation:
2-element Array{Int64,1}:
 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,Array{Float64,2}}
D factor:
2×2 Tridiagonal{Float64,Array{Float64,1}}:
 3.0   0.0
 0.0  -0.333333
L factor:
2×2 UnitLowerTriangular{Float64,Array{Float64,2}}:
 1.0        ·
 0.666667  1.0
permutation:
2-element Array{Int64,1}:
 2
 1
```

`LinearAlgebra.bunchkaufman!` – Function.

[3] 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.

```
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,Array{Float64,2},Array{Float64,1}}
eigenvalues:
3-element Array{Float64,1}:
  1.0
  3.0
 18.0
eigenvectors:
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> F.values
3-element Array{Float64,1}:
  1.0
  3.0
 18.0

julia> F.vectors
3×3 Array{Float64,2}:
 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 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> F = eigen(A, B)
GeneralizedEigen{Complex{Float64},Complex{Float64},Array{Complex{Float64},2},Array{Complex{Float64},1}}
eigenvalues:
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im
eigenvectors:
2×2 Array{Complex{Float64},2}:
  0.0+1.0im   0.0-1.0im
 -1.0+0.0im  -1.0-0.0im

julia> F.values
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> F.vectors
2×2 Array{Complex{Float64},2}:
  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 Array{Int64,2}:
```

```
 1  0
 0  4

julia> eigvals(diag_matrix)
2-element Array{Float64,1}:
 1.0
 4.0
```

For a scalar input, `eigvals` will return a scalar.

Example

```
julia> eigvals(-2)
-2
```

```
eigvals(A, B) -> values
```

Computes the generalized eigenvalues of `A` and `B`.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> eigvals(A,B)
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im
```

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> values
```

Returns 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.

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 1.0  2.0   ·
 2.0  2.0  3.0
  ·   3.0  1.0

julia> eigvals(A, 2:2)
1-element Array{Float64,1}:
 0.9999999999999996

julia> eigvals(A)
3-element Array{Float64,1}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259
```

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::Real, vu::Real) -> values
```

Returns 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.

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 1.0  2.0   ·
 2.0  2.0  3.0
  ·   3.0  1.0

julia> eigvals(A, -1, 2)
1-element Array{Float64,1}:
 1.0000000000000009

julia> eigvals(A)
3-element Array{Float64,1}:
 -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 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> eigvals!(A)
2-element Array{Float64,1}:
 -0.3722813232690143
  5.372281323269014

julia> A
2×2 Array{Float64,2}:
 -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 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> eigvals!(A, B)
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> A
2×2 Array{Float64,2}:
 -0.0  -1.0
  1.0  -0.0

julia> B
2×2 Array{Float64,2}:
 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 Array{Complex{Int64},2}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmax(A)
1.0
```

```
julia> A = [0 im; -1 0]
2×2 Array{Complex{Int64},2}:
  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 Array{Complex{Int64},2}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmin(A)
-1.0

julia> A = [0 im; -1 0]
2×2 Array{Complex{Int64},2}:
  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,Array{Float64,1}}:
 1.0  2.0   ·
 2.0  2.0  3.0
```

```
  ·   3.0  1.0

julia> eigvals(A)
3-element Array{Float64,1}:
 -2.1400549446402604
  1.0000000000000002
  5.140054944640259

julia> eigvecs(A)
3×3 Array{Float64,2}:
  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 Array{Float64,2}:
  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 kth 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 Array{Float64,2}:
 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 kth eigenvector can be obtained from the slice `M[:, k]`.)

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> eigvecs(A, B)
2×2 Array{Complex{Float64},2}:
  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
```

Computes 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.

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,Array{Float64,2},Array{Float64,1}}
eigenvalues:
3-element Array{Float64,1}:
  1.0
  3.0
 18.0
eigenvectors:
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> F.values
3-element Array{Float64,1}:
  1.0
  3.0
 18.0

julia> F.vectors
3×3 Array{Float64,2}:
 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) -> GeneralizedEigen
```

Computes 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`. (The kth generalized eigenvector can be obtained from the slice `F.vectors[:, k]`.)

Iterating the decomposition produces the components `F.values` and `F.vectors`.

Any keyword arguments passed to `eigen` are passed through to the lower-level `eigen!` function.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0  1
 1  0

julia> F = eigen(A, B);

julia> F.values
2-element Array{Complex{Float64},1}:
 0.0 - 1.0im
 0.0 + 1.0im

julia> F.vectors
2×2 Array{Complex{Float64},2}:
  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
```

Computes 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
```

Computes 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, [B]; permute, scale, 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 Array{Float64,2}:
 4.0  9.0  7.0
 4.0  4.0  1.0
 4.0  3.0  2.0

julia> F = hessenberg(A);

julia> F.Q * F.H * F.Q'
3×3 Array{Float64,2}:
 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 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur(A)
Schur{Float64,Array{Float64,2}}
T factor:
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Array{Float64,2}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Array{Float64,1}:
  3.0
 -2.0

julia> F.vectors * F.Schur * F.vectors'
2×2 Array{Float64,2}:
  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::StridedMatrix) -> 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`.

Iterating the decomposition produces the components `F.T`, `F.Z`, and `F.values`.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur(A)
Schur{Float64,Array{Float64,2}}
T factor:
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Array{Float64,2}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Array{Float64,1}:
  3.0
 -2.0

julia> F.vectors * F.Schur * F.vectors'
2×2 Array{Float64,2}:
  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::StridedMatrix, B::StridedMatrix) -> 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::StridedMatrix) -> F::Schur
```

Same as schur but uses the input argument A as workspace.

Examples

```
julia> A = [5. 7.; -2. -4.]
2×2 Array{Float64,2}:
  5.0   7.0
 -2.0  -4.0

julia> F = schur!(A)
Schur{Float64,Array{Float64,2}}
T factor:
2×2 Array{Float64,2}:
 3.0   9.0
 0.0  -2.0
Z factor:
2×2 Array{Float64,2}:
  0.961524  0.274721
 -0.274721  0.961524
eigenvalues:
2-element Array{Float64,1}:
  3.0
 -2.0

julia> A
2×2 Array{Float64,2}:
 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 Array{Float64,2}:
 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,Array{Float64,2}}
U factor:
4×4 Array{Float64,2}:
 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 Array{Float64,1}:
 3.0
 2.23606797749979
 2.0
 0.0
Vt factor:
4×5 Array{Float64,2}:
 -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 Array{Float64,2}:
 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 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> F = svd(A, B);

julia> F.U*F.D1*F.R0*F.Q'
2×2 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0
```

```
julia> F.V*F.D2*F.R0*F.Q'
2×2 Array{Float64,2}:
 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 a $M \times N$ 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 = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0. 2. 0. 0. 0.]
4×5 Array{Float64,2}:
 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);

julia> F.U * Diagonal(F.S) * F.Vt
4×5 Array{Float64,2}:
 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
```

```
svd(A, B) -> GeneralizedSVD
```

Compute the generalized SVD of A and B, returning a GeneralizedSVD factorization object F, such that A = F.U*F.D1*F.R0*F.Q' and B = F.V*F.D2*F.R0*F.Q'.

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 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> F = svd(A, B);

julia> F.U*F.D1*F.R0*F.Q'
2×2 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> F.V*F.D2*F.R0*F.Q'
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0
```

`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.

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 Array{Float64,2}:
 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);
```

```
julia> F.U * Diagonal(F.S) * F.Vt
4×5 Array{Float64,2}:
 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> A
4×5 Array{Float64,2}:
 -2.23607   0.0   0.0  0.0  0.618034
  0.0      -3.0   1.0  0.0  0.0
  0.0       0.0   0.0  0.0  0.0
  0.0       0.0  -2.0  0.0  0.0
```

```
svd!(A, B) -> GeneralizedSVD
```

`svd!` is the same as `svd`, but modifies the arguments `A` and `B` in-place, instead of making copies.

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> F = svd!(A, B);

julia> F.U*F.D1*F.R0*F.Q'
2×2 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> F.V*F.D2*F.R0*F.Q'
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> A
2×2 Array{Float64,2}:
 1.41421   0.0
 0.0      -1.41421

julia> B
2×2 Array{Float64,2}:
 1.0  -0.0
 0.0  -1.0
```

`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 Array{Float64,2}:
 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 Array{Float64,1}:
 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 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> svdvals(A, B)
2-element Array{Float64,1}:
 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.

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 Array{Float64,2}:
 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 Array{Float64,1}:
 3.0
```

```
 2.23606797749979
 2.0
 0.0

julia> A
4×5 Array{Float64,2}:
 -2.23607   0.0   0.0  0.0  0.618034
  0.0      -3.0   1.0  0.0  0.0
  0.0       0.0   0.0  0.0  0.0
  0.0       0.0  -2.0  0.0  0.0
```

```
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.

Examples

```
julia> A = [1. 0.; 0. -1.]
2×2 Array{Float64,2}:
 1.0   0.0
 0.0  -1.0

julia> B = [0. 1.; 1. 0.]
2×2 Array{Float64,2}:
 0.0  1.0
 1.0  0.0

julia> svdvals!(A, B)
2-element Array{Float64,1}:
 1.0
 1.0

julia> A
2×2 Array{Float64,2}:
 1.41421   0.0
 0.0      -1.41421

julia> B
2×2 Array{Float64,2}:
 1.0  -0.0
 0.0  -1.0
```

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 Array{Float64,2}:
 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 Array{Float64,2}:
 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)
```

Returns the upper triangle of `M` starting from the kth superdiagonal.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Array{Float64,2}:
 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 Array{Float64,2}:
 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 Array{Float64,2}:
 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 Array{Int64,2}:
 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 Array{Int64,2}:
 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 Array{Float64,2}:
 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 Array{Float64,2}:
 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)
```

Returns the lower triangle of `M` starting from the kth superdiagonal.

Examples

```
julia> a = fill(1.0, (4,4))
4×4 Array{Float64,2}:
 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 Array{Float64,2}:
 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 Array{Float64,2}:
 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 Array{Int64,2}:
 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 Array{Int64,2}:
 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.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 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

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9

julia> diag(A,1)
2-element Array{Int64,1}:
 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.

`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 Array{Int64,2}:
 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 Array{Int64,2}:
 0  1  0  0
 4  0  2  0
 0  5  0  3
 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 (if `size=nothing`), 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 Array{Int64,2}:
 1  0  0
 0  2  0
 0  0  3
```

`LinearAlgebra.rank` – Function.

```
rank(A::AbstractMatrix; atol::Real=0, rtol::Real=atol>0 ? 0 : n*ϵ)
rank(A::AbstractMatrix, rtol::Real)
```

Compute the rank of a matrix by counting how many singular values of `A` have magnitude greater than `max(atol, rtol*σ₁)` where `σ₁` is `A`'s largest 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`.

> **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 Array{Int64,1}:
  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:

$$\|A\|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}|$$

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 Array{Int64,2}:
 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!(v::AbstractVector, p::Real=2)
```

Normalize the vector v in-place so that its p-norm equals unity, i.e. `norm(v, p) == 1`. See also normalize and norm.

LinearAlgebra.normalize – Function.

```
normalize(v::AbstractVector, p::Real=2)
```

Normalize the vector v so that its p-norm equals unity, i.e. `norm(v, p) == 1`. See also normalize! and norm.

Examples

```
julia> a = [1,2,4];

julia> b = normalize(a)
3-element Array{Float64,1}:
 0.2182178902359924
 0.4364357804719848
 0.8728715609439696

julia> norm(b)
1.0

julia> c = normalize(a, 1)
3-element Array{Float64,1}:
 0.14285714285714285
 0.2857142857142857
 0.5714285714285714

julia> norm(c, 1)
1.0
```

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])
```

$$\kappa_S(M,p) = \left\| |M| \left|M^{-1}\right| \right\|_p$$
$$\kappa_S(M,x,p) = \left\| |M| \left|M^{-1}\right| |x| \right\|_p$$

Skeel condition number κ_S of the matrix `M`, optionally with respect to the vector `x`, as computed using the operator `p`-norm. $|M|$ denotes the matrix of (entry wise) absolute values of M ; $|M|_{ij} = |M_{ij}|$. 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 Array{Int64,2}:
 1  2
 3  4

julia> tr(A)
5
```

`LinearAlgebra.det` – Function.

```
det(M)
```

Matrix determinant.

Examples

```
julia> M = [1 0; 2 2]
2×2 Array{Int64,2}:
 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 Array{Int64,2}:
 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 Array{Float64,2}:
 -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 Array{Float64,2}:
 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 Array{Int64,2}:
 2  5
 1  3

julia> N = inv(M)
```

```
2×2 Array{Float64,2}:
  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(one(eltype(M))))))` is recommended.

For more information, see [4], [5], [6], [7].

Examples

```
julia> M = [1.5 1.3; 1.2 1.9]
2×2 Array{Float64,2}:
 1.5  1.3
 1.2  1.9

julia> N = pinv(M)
2×2 Array{Float64,2}:
  1.47287   -1.00775
 -0.930233   1.16279

julia> M * N
2×2 Array{Float64,2}:
 1.0          -2.22045e-16
 4.44089e-16   1.0
```

LinearAlgebra.nullspace – Function.

```
nullspace(M; atol::Real=0, rtol::Rea=atol>0 ? 0 : n*ϵ)
nullspace(M, rtol::Real) = nullspace(M; rtol=rtol) # to be deprecated in Julia 2.0
```

[4] Issue 8859, "Fix least squares", https://github.com/JuliaLang/julia/pull/8859

[5] Åke Björck, "Numerical Methods for Least Squares Problems", SIAM Press, Philadelphia, 1996, "Other Titles in Applied Mathematics", Vol. 51. doi:10.1137/1.9781611971484

[6] G. W. Stewart, "Rank Degeneracy", SIAM Journal on Scientific and Statistical Computing, 5(2), 1984, 403-413. doi:10.1137/0905030

[7] 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

Computes a basis for the nullspace of `M` by including the singular vectors of `A` whose singular have magnitude are greater than `max(atol, rtol*σ₁)`, where `σ₁` is `M`'s largest singularvalue.

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 Array{Int64,2}:
 1  0  0
 0  1  0
 0  0  0

julia> nullspace(M)
3×1 Array{Float64,2}:
 0.0
 0.0
 1.0

julia> nullspace(M, rtol=3)
3×3 Array{Float64,2}:
 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 Array{Float64,2}:
 0.0
 0.0
 1.0
```

`Base.kron` – Function.

```
kron(A, B)
```

Kronecker tensor product of two vectors or two matrices.

For 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).

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> B = [im 1; 1 -im]
2×2 Array{Complex{Int64},2}:
 0+1im  1+0im
 1+0im  0-1im

julia> kron(A, B)
4×4 Array{Complex{Int64},2}:
 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 Array{Int64,2}:
 3   6
 4   8
 5  10

julia> reshape(kron(v,w), (length(w), length(v)))
3×2 Array{Int64,2}:
 3   6
 4   8
 5  10
```

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 [8]) is chosen.

Examples

```
julia> A = Matrix(1.0I, 2, 2)
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0

julia> exp(A)
2×2 Array{Float64,2}:
 2.71828  0.0
 0.0      2.71828
```

Base.:^ – Method.

```
^(A::AbstractMatrix, p::Number)
```

Matrix power, equivalent to $\exp(p\log(A))$

Examples

[8] 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> [1 2; 0 3]^3
2×2 Array{Int64,2}:
 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 □) to a matrix was added in Julia 1.1.

Examples

```
julia> 2^[1 2; 0 3]
2×2 Array{Float64,2}:
 2.0  6.0
 0.0  8.0

julia> □^[1 2; 0 3]
2×2 Array{Float64,2}:
 2.71828  17.3673
 0.0      20.0855
```

Base.log – Method.

```
log(A{T}::StridedMatrix{T})
```

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 [9] and [10]). For general matrices, the complex Schur form (schur) is computed and the triangular algorithm is used on the triangular factor.

Examples

```
julia> A = Matrix(2.7182818*I, 2, 2)
2×2 Array{Float64,2}:
 2.71828  0.0
 0.0      2.71828

julia> log(A)
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

[9] 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

[10] 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

Base.sqrt – Method.

```
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 symmetric or Hermitian, its eigendecomposition (eigen) is used to compute the square root. Otherwise, the square root is determined by means of the Björck-Hammarling method [11], which computes the complex Schur form (schur) and then the complex square root of the triangular factor.

Examples

```
julia> A = [4 0; 0 4]
2×2 Array{Int64,2}:
 4  0
 0  4

julia> sqrt(A)
2×2 Array{Float64,2}:
 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 Array{Float64,2}:
  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 Array{Float64,2}:
 0.454649  0.454649
 0.454649  0.454649
```

[11] 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

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 Array{Float64,2}:
 0.454649  0.454649
 0.454649  0.454649

julia> C
2×2 Array{Float64,2}:
  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 Array{Float64,2}:
 -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 [12].

Examples

```
julia> acos(cos([0.5 0.1; -0.2 0.3]))
2×2 Array{Complex{Float64},2}:
  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)
```

[12] 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

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 [13].

Examples

```
julia> asin(sin([0.5 0.1; -0.2 0.3]))
2×2 Array{Complex{Float64},2}:
  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 [14].

Examples

```
julia> atan(tan([0.5 0.1; -0.2 0.3]))
2×2 Array{Complex{Float64},2}:
  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)
```

[13] 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

[14] 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

Compute the inverse hyperbolic matrix cosine of a square matrix A. For the theory and logarithmic formulas used to compute this function, see [15].

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 [16].

Base.atanh – Method.

```
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 [17].

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 Array{Float64,2}:
 3.0  4.0
 5.0  6.0
```

[15] 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

[16] 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

[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

```
julia> B = [1. 1.; 1. 2.]
2×2 Array{Float64,2}:
 1.0  1.0
 1.0  2.0

julia> X = lyap(A, B)
2×2 Array{Float64,2}:
  0.5  -0.5
 -0.5   0.25

julia> A*X + X*A' + B
2×2 Array{Float64,2}:
 0.0          6.66134e-16
 6.66134e-16  8.88178e-16
```

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 Array{Float64,2}:
 3.0  4.0
 5.0  6.0

julia> B = [1. 1.; 1. 2.]
2×2 Array{Float64,2}:
 1.0  1.0
 1.0  2.0

julia> C = [1. 2.; -2. 1]
2×2 Array{Float64,2}:
  1.0  2.0
 -2.0  1.0

julia> X = sylvester(A, B, C)
2×2 Array{Float64,2}:
 -4.46667   1.93333
  3.73333  -1.8

julia> A*X + X*B + C
2×2 Array{Float64,2}:
  2.66454e-15  1.77636e-15
 -3.77476e-15  4.44089e-16
```

LinearAlgebra.issuccess – Function.

```
issuccess(F::Factorization)
```

Test that a factorization of a matrix succeeded.

```
julia> F = cholesky([1 0; 0 1]);
```

```
julia> LinearAlgebra.issuccess(F)
true

julia> F = lu([1 0; 0 0]; check = false);

julia> LinearAlgebra.issuccess(F)
false
```

LinearAlgebra.issymmetric – Function.

```
issymmetric(A) -> Bool
```

Test whether a matrix is symmetric.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> issymmetric(a)
true

julia> b = [1 im; -im 1]
2×2 Array{Complex{Int64},2}:
 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!

Examples

```
julia> A = [1 2; 2 50]
2×2 Array{Int64,2}:
 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 Array{Float64,2}:
 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 kth superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> istril(a)
false

julia> istril(a, 1)
true

julia> b = [1 0; -im -1]
2×2 Array{Complex{Int64},2}:
 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 Array{Int64,2}:
 1   2
 2  -1

julia> istriu(a)
false

julia> istriu(a, -1)
```

```
true

julia> b = [1 im; 0 -1]
2×2 Array{Complex{Int64},2}:
 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.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> isdiag(a)
false

julia> b = [im 0; 0 -im]
2×2 Array{Complex{Int64},2}:
 0+1im  0+0im
 0+0im  0-1im

julia> isdiag(b)
true
```

LinearAlgebra.ishermitian – Function.

```
ishermitian(A) -> Bool
```

Test whether a matrix is Hermitian.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> ishermitian(a)
true

julia> b = [1 im; -im 1]
2×2 Array{Complex{Int64},2}:
 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+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> transpose(A)
2×2 Transpose{Complex{Int64},Array{Complex{Int64},2}}:
 3+2im  8+7im
 9+2im  4+6im
```

LinearAlgebra.transpose! – Function.

```
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 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> B = zeros(Complex{Int64}, 2, 2)
2×2 Array{Complex{Int64},2}:
 0+0im  0+0im
 0+0im  0+0im

julia> transpose!(B, A);

julia> B
2×2 Array{Complex{Int64},2}:
 3+2im  8+7im
 9+2im  4+6im

julia> A
2×2 Array{Complex{Int64},2}:
 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`, but also some `Factorization`, for instance. 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 = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> transpose(A)
2×2 Transpose{Complex{Int64},Array{Complex{Int64},2}}:
 3+2im  8+7im
 9+2im  4+6im
```

Base.adjoint – Function.

```
adjoint(A)
```

Lazy adjoint (conjugate transposition) (also postfix `'`). Note that `adjoint` is applied recursively to elements.

This operation is intended for linear algebra usage - for general data manipulation see `permutedims`.

Examples

```
julia> A = [3+2im 9+2im; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> adjoint(A)
2×2 Adjoint{Complex{Int64},Array{Complex{Int64},2}}:
 3-2im  8-7im
 9-2im  4-6im
```

LinearAlgebra.adjoint! – Function.

```
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 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im
```

```
julia> B = zeros(Complex{Int64}, 2, 2)
2×2 Array{Complex{Int64},2}:
 0+0im  0+0im
 0+0im  0+0im

julia> adjoint!(B, A);

julia> B
2×2 Array{Complex{Int64},2}:
 3-2im  8-7im
 9-2im  4-6im

julia> A
2×2 Array{Complex{Int64},2}:
 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`, but also some `Factorization`, for instance. 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; 8+7im  4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> adjoint(A)
2×2 Adjoint{Complex{Int64},Array{Complex{Int64},2}}:
 3-2im  8-7im
 9-2im  4-6im
```

`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 Array{Complex{Int64},2}:
 1+0im  0+2im
 0-3im  4+0im
```

```
julia> T = transpose(A)
2×2 Transpose{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0-3im
 0+2im  4+0im

julia> copy(T)
2×2 Array{Complex{Int64},2}:
 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 Array{Int64,1}:
 1
 2
 3
 4

julia> LinearAlgebra.stride1(A)
1

julia> B = view(A, 2:2:4)
2-element view(::Array{Int64,1}, 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 Array{Int64,1}:
 4
 5
```

LinearAlgebra.peakflops – Function.

```
LinearAlgebra.peakflops(n::Integer=2000; 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 a matrix of size n x n, where n = 2000. 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 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.

71.4 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 Array{Float64,2}:
 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 Array{Float64,2}:
 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 Array{Int64,2}:
 1  2
 3  4

julia> lmul!(2, B)
2×2 Array{Int64,2}:
 2  4
 6  8

julia> lmul!(0.0, [Inf])
1-element Array{Float64,1}:
 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 = LinearAlgebra.UpperTriangular([1 2; 0 3]);

julia> LinearAlgebra.lmul!(A, B);

julia> B
2×2 Array{Int64,2}:
 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 Array{Float64,2}:
 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 Array{Int64,2}:
 1  2
 3  4

julia> rmul!(A, 2)
2×2 Array{Int64,2}:
 2  4
 6  8

julia> rmul!([NaN], 0.0)
1-element Array{Float64,1}:
 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 = LinearAlgebra.UpperTriangular([1 2; 0 3]);

julia> LinearAlgebra.rmul!(A, B);

julia> A
2×2 Array{Int64,2}:
 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 Array{Float64,2}:
 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.

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
3-element Array{Float64,1}:
  0.7128099173553719
 -0.051652892561983674
  0.10020661157024757

julia> A\X
3-element Array{Float64,1}:
  0.7128099173553719
 -0.05165289256198333
  0.10020661157024785
```

```
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.

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
3-element Array{Float64,1}:
  0.7128099173553719
 -0.051652892561983674
  0.10020661157024757

julia> A\Y
3-element Array{Float64,1}:
  0.7128099173553719
 -0.05165289256198333
  0.10020661157024785
```

```
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 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> ldiv!(2.0, B)
2×2 Array{Float64,2}:
 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`.

```
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 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> rdiv!(A, 2.0)
2×2 Array{Float64,2}:
 0.5  1.0
 1.5  2.0
```

71.5 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 `Float64`, `Float32`, `ComplexF64`, and `ComplexF32` 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.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.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.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 absolute values of the first n elements of array X with stride `incx`.

Examples

```
julia> BLAS.asum(5, fill(1.0im, 10), 2)
5.0

julia> BLAS.asum(2, fill(1.0im, 10), 5)
2.0
```

`LinearAlgebra.axpy!` – Function.

```
axpy!(a, X, Y)
```

Overwrite Y with a*X + Y, where a is a scalar. Return Y.

Examples

```
julia> x = [1; 2; 3];

julia> y = [4; 5; 6];

julia> BLAS.axpy!(2, x, y)
3-element Array{Int64,1}:
  6
  9
 12
```

`LinearAlgebra.axpby!` – Function.

```
axpby!(a, X, b, Y)
```

Overwrite Y with X*a + Y*b, where a and b are scalars. Return Y.

Examples

```
julia> x = [1., 2, 3];

julia> y = [4., 5, 6];

julia> BLAS.axpby!(2., x, 3., y)
3-element Array{Float64,1}:
 14.0
 19.0
 24.0
```

`LinearAlgebra.BLAS.scal!` – Function.

```
scal!(n, a, X, incx)
```

Overwrite X with a*X for the first n elements of array X with stride `incx`. Returns X.

`LinearAlgebra.BLAS.scal` – Function.

```
scal(n, a, X, incx)
```

Return X scaled by a for the first n elements of array X with stride `incx`.

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).

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.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.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. Returns C.

LinearAlgebra.BLAS.syrk – Function.

```
syrk(uplo, trans, alpha, A)
```

Returns 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.

Missing docstring.

Missing docstring for LinearAlgebra.BLAS.syr2k!. Check Documenter's build log for details.

Missing docstring.

Missing docstring for LinearAlgebra.BLAS.syr2k. Check Documenter's build log for details.

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.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.

Missing docstring.

Missing docstring for LinearAlgebra.BLAS.her2k!. Check Documenter's build log for details.

Missing docstring.

Missing docstring for LinearAlgebra.BLAS.her2k. Check Documenter's build log for details.

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.sbmv! – Function.

```
sbmv!(uplo, k, alpha, A, x, beta, y)
```

Update vector y as alpha*A*x + beta*y where A is a 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.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.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.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.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.

Missing docstring.

Missing docstring for LinearAlgebra.BLAS.hemm!. Check Documenter's build log for details.

Missing docstring.

Missing docstring for LinearAlgebra.BLAS.hemm(::Any, ::Any, ::Any, ::Any, ::Any). Check Documenter's build log for details.

Missing docstring.

Missing docstring for LinearAlgebra.BLAS.hemm(::Any, ::Any, ::Any, ::Any). Check Documenter's build log for details.

Missing docstring.

Missing docstring for LinearAlgebra.BLAS.hemv!. Check Documenter's build log for details.

Missing docstring.

Missing docstring for LinearAlgebra.BLAS.hemv(::Any, ::Any, ::Any, ::Any). Check Documenter's build log for details.

Missing docstring.

Missing docstring for LinearAlgebra.BLAS.hemv(::Any, ::Any, ::Any). Check Documenter's build log for details.

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. Returns the updated B.

LinearAlgebra.BLAS.trmm – Function.

```
trmm(side, ul, tA, dA, alpha, A, B)
```

Returns 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.

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.

LinearAlgebra.BLAS.set_num_threads – Function.

```
set_num_threads(n)
```

Set the number of threads the BLAS library should use.

71.6 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)
```

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. `jpvt` must have length length greater than or equal to `n` if `A` is an `(m x n)` matrix. `tau` must have length greater than or equal to the smallest dimension of `A`.

`A`, `jpvt`, and `tau` are modified in-place.

```
geqp3!(A, jpvt) -> (A, jpvt, tau)
```

Compute the pivoted QR factorization of A, AP = QR using BLAS level 3. P is a pivoting matrix, represented by jpvt. jpvt must have length greater than or equal to n if A is an (m x n) matrix.

Returns A and jpvt, modified in-place, and tau, which stores the elementary reflectors.

```
geqp3!(A) -> (A, jpvt, tau)
```

Compute the pivoted QR factorization of A, AP = QR using BLAS level 3.

Returns A, modified in-place, jpvt, which represents the pivoting matrix P, and tau, which stores the elementary reflectors.

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.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 = 0` 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.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.trexc!` – Function.

```
trexc!(compq, ifst, ilst, T, Q) -> (T, Q)
```

Reorder the Schur factorization of a matrix. If `compq = V`, the Schur vectors `Q` are reordered. If `compq = N` they are not modified. `ifst` and `ilst` specify the reordering of the vectors.

`LinearAlgebra.LAPACK.trsen!` – Function.

```
trsen!(compq, job, 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.

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`.

Chapter 72

日志记录

The Logging module provides a way to record the history and progress of a computation as a log of events. Events are created by inserting a logging statement into the source code, for example:

```
@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)

# 输出:
┌ Info: Some variables
│   A =
│    4×4 Array{Int64,2}:
│     1  1  1  1
│     1  1  1  1
│     1  1  1  1
│     1  1  1  1
└   s = 100.0
```

All of the logging macros `@debug`, `@info`, `@warn` and `@error` share common features that are described in detail in the documentation for the more general macro `@logmsg`.

72.1 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:

  - `Debug` is information intended for the developer of the program.

  These events are disabled by default.

  - `Info` is for general information to the user.

  Think of it as an alternative to using `println` directly.

  - `Warn` means something is wrong and action is likely required

  but that for now the program is still working.

  - `Error` 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 `AbstractString`s 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.

72.2 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`.

72.3 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.

72.4 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. For example, loading julia with `JULIA_DEBUG=loading` will activate `@debug` log messages in `loading.jl`:

```
$ 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
...
```

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
```

72.5 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)
```

72.6 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.

source

Processing events with AbstractLogger

Event processing is controlled by overriding functions associated with `AbstractLogger`:

`Logging.AbstractLogger` – Type.

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

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 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.

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=stderr, 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=stderr, min_level=Info)
```

Simplistic logger for logging all messages with level greater than or equal to `min_level` to `stream`.

source

Chapter 73

Markdown

This section describes Julia's markdown syntax, which is enabled by the Markdown standard library. The following Markdown elements are supported:

73.1 Inline elements

Here "inline" refers to elements that can be found within blocks of text, i.e. paragraphs. These include the following elements.

Bold

Surround words with two asterisks, **, to display the enclosed text in boldface.

```
A paragraph containing a **bold** word.
```

Italics

Surround words with one asterisk, *, to display the enclosed text in italics.

```
A paragraph containing an *italicized* word.
```

Literals

Surround text that should be displayed exactly as written with single backticks, ` .

```
A paragraph containing a `literal` word.
```

Literals should be used when writing text that refers to names of variables, functions, or other parts of a Julia program.

> **Tip**
>
> To include a backtick character within literal text use three backticks rather than one to enclose the text.
>
> ```
> A paragraph containing ``` `backtick` characters ```.
> ```
>
> By extension any odd number of backticks may be used to enclose a lesser number of backticks.

LaTeX

Surround text that should be displayed as mathematics using LaTeX syntax with double backticks, `` .

```
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 LaTeX markup 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.

73.2 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 type of the admonition can be any word made up of only lowercase Latin characters (a-z), but some types produce special styling, namely (in order of decreasing severity): `danger`, `warning`, `info`, `note`, and `tip`.

A custom title for the box can be provided as a string (in double quotes) after the admonition type. For that standard types (`danger`, `warning`... etc_, if no title text is specified after the admonition type, then the type title used will be the type of the block. E.g. `"Note"` in the case of the `note` admonition.

If you would like to define your own block, for example a `terminology` block used like so:

```
!!! terminology "julia vs Julia"
    Strictly speaking, Julia refers to the language,
    and julia the standard implementation.
```

Admonitions, like most other toplevel elements, can contain other toplevel elements.

73.3 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 74

内存映射 I/O

`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.mmap`. Used by `SharedArray` for creating shared memory arrays.

Examples

```
julia> anon = Mmap.Anonymous();

julia> isreadable(anon)
true

julia> iswritable(anon)
true

julia> isopen(anon)
true
```

`Mmap.mmap` – Function.

```
Mmap.mmap(io::Union{IOStream,AbstractString,Mmap.AnonymousMmap}[, type::Type{Array{T,N}}, dims,
↪  offset]; grow::Bool=true, shared::Bool=true)
Mmap.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)
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.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.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> io = open("mmap.bin", "w+");

julia> B = Mmap.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.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 75

Pkg

75.1 介绍

Pkg 是 Julia 1.0 及后续新版本的标准包管理器。与那些安装和管理单个全局软件包集的传统包管理器不同，Pkg 是围绕「环境」设计的。每个项目都有一套独立与其他项目的软件包集合。同一个软件包也可以在多个项目中通过名字共享。项目环境的软件包信息是保存在 清单文件里的。清单文件确切的描述了每一个依赖软件包和它的版本。清单文件可以检入项目存储库并在版本控制中进行跟踪，从而显着提高项目的可重复性。如果你曾经试图运行一段时间未曾使用过的代码，但发现其完全无法工作，而这只是因为你更新或卸载了项目使用的一些软件包，那么你会理解这种方法的意图。在 Pkg 中，由于每个项目都维护着各自独立的软件包集，你再也不会遇到这个问题了。此外，如果你签出项目到新系统中，搭建出其清单文件所描述的环境将会非常地简单，并且你可以立即启动和并运行该项目，因为我们知道项目依赖项是好的。

由于项目包环境是彼此独立地进行管理和更新的，Pkg 显著地缓解了「依赖地狱」问题。你如果想在新项目中使用最新、最棒的包，但在另一个项目中却卡在了使用旧版本的包，那也没问题——因为它们的环境是彼此分离的，不同项目可以使用装在系统的不同位置的不同版本的包。每个版本的包的位置都是规范的，所以当多个环境使用的包版本相同时，它们可以共享同一安装包，这就避免不必要的重复安装。不被任何环境使用的老旧版本的包，会被包管理器定期「垃圾收集」掉。

Pkg 对本地环境的处理方法可能让曾经使用过 Python 的 `virtualenv` 或 Ruby 的 `bundler` 的人感到熟悉。在 Julia 中，我们不仅没有通过破解语言的代码加载机制来支持环境，而且还有 Julia 本身就理解它们的好处。此外，Julia 环境是「可堆叠的」：你可以将一个环境叠加在另一个环境上，从而可以访问主环境之外的其它包。这使得更容易在提供主环境的项目上工作，同时依然访问所有你常用的开发工具，如分析器、调试器等，这只需在加载路径中更后地包含具有这些开发环境的路径。

Last but not least, Pkg is designed to support federated package registries. This means that it allows multiple registries managed by different parties to interact seamlessly. In particular, this includes private registries which can live behind corporate firewalls. You can install and update your own packages from a private registry with exactly the same tools and workflows that you use to install and manage official Julia packages. If you urgently need to apply a hotfix for a public package that's critical to your company's product, you can tag a private version of it in your company's internal registry and get a fix to your developers and ops teams quickly and easily without having to wait for an upstream patch to be accepted and published. Once an official fix is published, however, you can just upgrade your dependencies and you'll be back on an official release again.

75.2 词汇表

项目（Project）：一个具有标准布局的源代码树，包括了用来放置主要的 Julia 代码的 `src` 目录、用来放置测试的 `test` 目录、用来放置文档的 `docs` 目录和可选的用来放置构建脚本及其输出的 `deps` 目录。项目通常有一个项目文件和一个可选的清单文件：

- **项目文件(Project file)**:一个在项目根目录下的文件,叫做 `Project.toml`(或 `JuliaProject.toml`),用来描述项目的元数据,包括项目的名称、UUID(针对包)、作者、许可证和它所依赖的包和库的名称及 UUID。
- **清单文件(Manifest file)**:一个在项目根目录下的文件,叫做 `Manifest.toml`(或 `JuliaManifest.toml`),用来描述完整的依赖关系图、每个包的确切版本以及项目使用的库。

包(Package):一个提供可重用功能的项目,其它 Julia 项目可以同 `import X` 或 `using X` 使用它。一个包应该包含一个具有 `uuid` 条目(此条目给出该包 UUID)的项目文件。此 UUID 用于在依赖它的项目中标识该包。

> **Note**
>
> 由于历史原因,可以在 REPL 或脚本的顶级中加载没有项目文件或 UUID 的包。但是,无法在具有项目文件或 UUID 的项目中加载没有它们的包。一旦你曾从项目文件加载包,所有包就都需要项目文件和 UUID。

应用(application):一个提供独立功能的项目,不打算被其它 Julia 项目重用。例如,Web 应用、命令行工具或者科学论文附带的模拟或分析代码。应用可以有 UUID 但也可以没有。应用还可以为其所依赖的包提供全局配置选项。另一方面,包不可能提供全局配置,因为这可能与主应用的配置相冲突。

> **Note**
>
> **项目** ***vs.*** **包** ***vs.*** **应用**:
>
> 1. **项目**是一个总称:包和应用都是一种项目。
> 2. **包**应该有 UUID,而应用可以有也可以没有。
> 3. **应用**可以提供全局的配置,而包不行。

Library (future work): a compiled binary dependency (not written in Julia) packaged to be used by a Julia project. These are currently typically built in- place by a `deps/build.jl` script in a project's source tree, but in the future we plan to make libraries first-class entities directly installed and upgraded by the package manager.

环境(Environment):项目文件和清单文件的组合,项目文件与依赖关系图相结合后提供了顶级名称映射,而清单文件提供了包到它们入口点的映射。有关的详细信息,请参阅手册中代码加载的相关章节。

- **显式环境(Explicit environment)**:在同一目录下具有显式的项目文件和可选的与其对应的清单文件。如果清单文件不存在,那么隐含的依赖关系图和位置映射为空。
- **隐式环境(Implicit environment)**:作为目录提供的环境(没有项目文件或清单文件),此目录包含包且包含的包具有形式为 `X.jl`、`X.jl/src/X.jl` 或 `X/src/X.jl` 的入口点,这些包的入口点隐含了顶级名称映射。依赖关系图隐含在这些包所在目录的项目文件里,例如 `X.jl/Project.toml` 或 `X/Project.toml`。如果 `X` 存在对应的项目文件,则其依赖关系就是其项目文件的依赖关系。入口点本身就隐含了位置映射。

Registry: a source tree with a standard layout recording metadata about a registered set of packages, the tagged versions of them which are available, and which versions of packages are compatible or incompatible with each other. A registry is indexed by package name and UUID, and has a directory for each registered package providing the following metadata about it:

- name——例如 `DataFrames`
- UUID——例如 `a93c6f00-e57d-5684-b7b6-d8193f3e46c0`
- authors——例如 `Jane Q. Developer <jane@example.com>`
- license——例如 MIT，BSD3 或 GPLv2
- repository——例如 `https://github.com/JuliaData/DataFrames.jl.git`
- description——一个总结包功能的文本块
- keywords——例如 `data`，`tabular`，`analysis`，`statistics`
- versions——所有已注册版本的标签列表

每个包的已注册版本都会提供以下信息：

- its semantic version number —e.g. `v1.2.3`
- its git tree SHA-1 hash — e.g. `7ffb18ea3245ef98e368b02b81e8a86543a11103`
- a map from names to UUIDs of dependencies
- which versions of other packages it is compatible/incompatible with

Dependencies and compatibility are stored in a compressed but human-readable format using ranges of package versions.

Depot: a directory on a system where various package-related resources live, including:

- `environments`: shared named environments (e.g. `v0.7`, `devtools`)
- `clones`: bare clones of package repositories
- `compiled`: cached compiled package images (`.ji` files)
- `config`: global configuration files (e.g. `startup.jl`)
- `dev`: default directory for package development
- `logs`: log files (e.g. `manifest_usage.toml`, `repl_history.jl`)
- `packages`: installed package versions
- `registries`: clones of registries (e.g. `General`)

Load path: a stack of environments where package identities, their dependencies, and entry-points are searched for. The load path is controlled in Julia by the `LOAD_PATH` global variable which is populated at startup based on the value of the `JULIA_LOAD_PATH` environment variable. The first entry is your primary environment, often the current project, while later entries provide additional packages one may want to use from the REPL or top-level scripts.

Depot path: a stack of depot locations where the package manager, as well as Julia's code loading mechanisms, look for registries, installed packages, named environments, repo clones, cached compiled package images, and configuration files. The depot path is controlled by the Julia `DEPOT_PATH` global variable which is populated at startup based on the value of the `JULIA_DEPOT_PATH` environment variable. 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.

75.3 入门

在 Julia REPL 中使用] 键即可进入 Pkg 模式。

```
(v0.7) pkg>
```

提示符括号内的部分显示当前项目的名称。由于我们尚未创建自己的项目，我们正处于默认项目中，其位于 ~/.julia/environments/v0.7（或任何你恰巧在运行的 Julia 版本）。

要返回 julia> 提示符，请在输入行为空时按退格键或直接按 Ctrl+C。可通过调用 pkg>help 获得帮助。如果你所处的环境无法访问 PEPL，你仍可以通过字符串宏 pkg（其在 using Pkg 后可用）使用 REPL 模式的命令。命令 pkg"cms" 将等价于在 RPEL 模式中执行 cmd。

此处的文档介绍了如何使用 REPL 的 Pkg 模式。使用 Pkg API（通过调用 Pkg. 函数）的文档正在编写中。

添加包

有两种方法可以添加包，分别是使用 add 命令和 dev 命令。最常用的是 add，我们首先介绍它的用法。

添加已注册的包

在 REPL 的 Pkg 模式中，添加包可以使用 add 命令，其后接包的名称，例如：

```
(v0.7) pkg> add Example
   Cloning default registries into /Users/kristoffer/.julia/registries
   Cloning registry General from "https://github.com/JuliaRegistries/General.git"
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] + Example v0.5.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] + Example v0.5.1
  [8dfed614] + Test
```

在这里，我们将包 Example 添加到当前项目中。此例中，我们使用的是全新的 Julia 安装，并且这是我们第一次使用 Pkg 添加包。默认情况下，Pkg 会克隆 Julia 的 General 注册表，并使用此注册表来查找需要包含在当前环境中的包。状态更新在左侧显示了简短形式的包 UUID，接着是包名称和版本号。因为标准库（例如 Test）随 Julia 一起提供，所以它们没有版本号。项目状态包含你自己添加的包，在此例中为 Example：

```
(v0.7) pkg> st
    Status `Project.toml`
  [7876af07] Example v0.5.1
```

此外，清单状态包含了显式添加的包的依赖项。

```
(v0.7) pkg> st --manifest
    Status `Manifest.toml`
  [7876af07] Example v0.5.1
  [8dfed614] Test
```

可以在一次命令中添加多个包，例如 `pkg> add A B C`。

在包已添加进项目中后，可在 Julia 中加载它：

```
julia> using Example

julia> Example.hello("User")
"Hello, User"
```

可以通过在 `@` 符号后附加版本号来安装特定版本，例如在包名称后附加 `@v0.4`：

```
(v0.7) pkg> add Example@0.4
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] + Example v0.4.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] + Example v0.4.1
```

如果 `Example` 的主分支（或某个提交 SHA）有尚未包含在已注册版本中的修补程序，我们可以通过在包名称后附加 `#branch`（或 `#commit`）来显式跟踪该分支（或提交）：

```
(v0.7) pkg> add Example#master
  Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] ~ Example v0.5.1 ⇒ v0.5.1+ #master (https://github.com/JuliaLang/Example.jl.git)
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] ~ Example v0.5.1 ⇒ v0.5.1+ #master (https://github.com/JuliaLang/Example.jl.git)
```

状态输出现在显示我们正在跟踪 `Example` 的 `master` 分支。在更新包时，我们将从该分支中拉取更新。

要返回到跟踪 `Example` 的注册表版本，请使用 `free` 命令：

```
(v0.7) pkg> free Example
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] ~ Example v0.5.1+ #master (https://github.com/JuliaLang/Example.jl.git) ⇒ v0.5.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] ~ Example v0.5.1+ #master )https://github.com/JuliaLang/Example.jl.git) ⇒ v0.5.1
```

添加未注册包

如果某个包不在注册表中，通过将其存储库的 URL 传给 `add` 而不是包名称，仍然可以添加它。

```
(v0.7) pkg> add https://github.com/fredrikekre/ImportMacros.jl
  Updating git-repo `https://github.com/fredrikekre/ImportMacros.jl`
 Resolving package versions...
Downloaded MacroTools ─ v0.4.1
  Updating `~/.julia/environments/v0.7/Project.toml`
  [e6797606] + ImportMacros v0.0.0 # (https://github.com/fredrikekre/ImportMacros.jl)
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [e6797606] + ImportMacros v0.0.0 # (https://github.com/fredrikekre/ImportMacros.jl)
  [1914dd2f] + MacroTools v0.4.1
```

可以看到，未注册包的依赖项（此处为 `MacroTools`）已被添加。对于未注册包，我们可以使用 `#` 来给定一个分支（或 commit SHA）来进行跟踪，就像已注册包一样。

添加本地包

我们可以将一个 git 存储库的本地路径传给 add 而不是其 URL，其效果类似于传 URL。该本地存储库（的某个分支）会被跟踪，并在包更新时从已拉取的本地存储库中获取更新。请注意，本地包存储库中的文件更改不会在包加载时立即反映出来。为了拉取更改，必须提交该更改并更新包。

开发包

仅使用 add 会让你的清单始终为「可再现状态」，换句话说，只要所使用的存储库和注册表仍然可以访问，就可以检索出项目中所有依赖项的确切状态。这样做的好处是你可以将你的项目（Project.toml 和 Manifest.toml）发送该其他人，然后他们可以该项目「实例化」到与你本地项目相同的状态。但是，当你在开发包时，在某个路径上以当前状态加载包会更方便。因此，命令 dev 有存在必要。

让我们来尝试 dev 一个已注册的包：

```
(v0.7) pkg> dev Example
  Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] + Example v0.5.1+ [`~/.julia/dev/Example`]
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] + Example v0.5.1+ [`~/.julia/dev/Example`]
```

dev 命令会获取包的完整克隆到 ~/.julia/dev/ 目录下（可通过设置环境变量 JULIA_PKG_DEVDIR 来更改此路径）。在导入 Example 时，julia 现在将从 ~/.julia/dev/Example 导入它，并且该路径下文件的所有本地更改都将反映在加载的代码中。在使用 add 时，我们说我们跟踪了包存储库，在这里则说我们跟踪了路径本身。请注意，包管理器永远不会触碰已跟踪路径上的任何文件。因此，需要你自己拉取更新、更改分支等。如果我们尝试 dev 包的某个已经存在于 ~/.julia/dev/ 里的分支，则包管理器只会使用已存在的路径。例如：

```
(v0.7) pkg> dev Example
  Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
[ Info: Path `/Users/kristoffer/.julia/dev/Example` exists and looks like the correct package, using
      existing path instead of cloning
```

请注意，info 信息表明它正在使用现有路径。一般来说，包管理器不会触碰正在跟踪的路径文件。

如果在本地路径上使用 dev，则该包的路径会被记录并在该包加载时使用之。除非该路径以绝对路径的形式给出，否则它会以相对于项目文件的形式记录下来。

要停止跟踪路径并再次使用已注册版本，请使用 free

```
(v0.7) pkg> free Example
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] ↓ Example v0.5.1+ [`~/.julia/dev/Example`] ⇒ v0.5.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] ↓ Example v0.5.1+ [`~/.julia/dev/Example`] ⇒ v0.5.1
```

值得提及的是，通过使用 dev，你的项目现在具有其内在状态。其状态取决于该路径中文件的当前内容，并且在不知道所跟踪路径中所有包的确切内容的情况下，其他人无法「实例化」清单。

Note that if you add a dependency to a package that tracks a local path, the Manifest (which contains the whole dependency graph) will become out of sync with the actual dependency graph. This means that the package will not be able to load that dependency since it is not recorded in the Manifest. To update sync the Manifest, use the REPL command resolve.

删除包

通过使用 `pkg> rm Package`，可从当前项目中删除包。这只会删除已存在于项目中的包，要删除仅作为依赖项的包，请使用 `pkg> rm --manifest DepPackage`。请注意，这会删除所有依赖于 `DepPackage` 的包。

更新包

当项目正在使用的包发布新版本时，最好进行更新。简单地调用 `up` 会尝试将项目的所有依赖项更新到最新的兼容版本。有时这并不是你想要的。通过将依赖项子集作为参数传给 `up`，你可以指定要升级的依赖项，例如

```
(v0.7) pkg> up Example
```

所有其他包直接依赖项的版本会保持不变。如果你为了降低项目中断的风险，只想要更新包的次版本号，你可以加上 `--minor` 标志，例如：

```
(v0.7) pkg> up --minor Example
```

跟踪存储库的包在进行次要更新时不会被更新，而跟踪路径的包永远不会被包管理器所触及。

Pinning a package

A pinned package will never be updated. A package can be pinned using `pin` as for example

```
(v0.7) pkg> pin Example
 Resolving package versions...
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] ~ Example v0.5.1 ⇒ v0.5.1 
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] ~ Example v0.5.1 ⇒ v0.5.1 
```

Note the pin symbol showing that the package is pinned. Removing the pin is done using `free`

```
(v0.7) pkg> free Example
  Updating `~/.julia/environments/v0.7/Project.toml`
  [7876af07] ~ Example v0.5.1  ⇒ v0.5.1
  Updating `~/.julia/environments/v0.7/Manifest.toml`
  [7876af07] ~ Example v0.5.1  ⇒ v0.5.1
```

测试包

包的测试可通过 `test` 命令来运行：

```
(v0.7) pkg> test Example
   Testing Example
   Testing Example tests passed
```

构建包

第一次安装某个包时，会自动执行该包的构建步骤。构建过程的输出会被重定向到文件中。要显式执行包的构建步骤，请使用 `build` 命令：

```
(v0.7) pkg> build MbedTLS
  Building MbedTLS → `~/.julia/packages/MbedTLS/h1Vu/deps/build.log`

shell> cat ~/.julia/packages/MbedTLS/h1Vu/deps/build.log
```

```
┌ Warning: `wait(t::Task)` is deprecated, use `fetch(t)` instead.
│   caller = macro expansion at OutputCollector.jl:63 [inlined]
└ @ Core OutputCollector.jl:63
...
[ Info: using prebuilt binaries
```

75.4 Creating your own projects

So far we have added packages to the default project at `~/.julia/environments/v0.7`, it is, however, easy to create other, independent, projects. It should be pointed out if two projects uses the same package at the same version, the content of this package is not duplicated. In order to create a new project, create a directory for it and then activate that directory to make it the "active project" which package operations manipulate:

```
shell> mkdir MyProject

shell> cd MyProject
/Users/kristoffer/MyProject

(v0.7) pkg> activate .

(MyProject) pkg> st
    Status `Project.toml`
```

Note that the REPL prompt changed when the new project is activated. Since this is a newly created project, the status command show it contains no packages, and in fact, it has no project or manifest file until we add a package to it:

```
shell> ls -l
total 0

(MyProject) pkg> add Example
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `Project.toml`
  [7876af07] + Example v0.5.1
  Updating `Manifest.toml`
  [7876af07] + Example v0.5.1
  [8dfed614] + Test

shell> ls -l
total 8
-rw-r--r-- 1 stefan staff 207 Jul  3 16:35 Manifest.toml
-rw-r--r-- 1 stefan staff  56 Jul  3 16:35 Project.toml

shell> cat Project.toml
[deps]
Example = "7876af07-990d-54b4-ab0e-23690620f79a"

shell> cat Manifest.toml
[[Example]]
deps = ["Test"]
git-tree-sha1 = "8eb7b4d4ca487caade9ba3e85932e28ce6d6e1f8"
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
version = "0.5.1"
```

```
[[Test]]
uuid = "8dfed614-e22c-5e08-85e1-65c5234f0b40"
```

This new environment is completely separate from the one we used earlier.

75.5 垃圾收集旧的、不再使用的包

随着包的更新和项目被删除，曾经使用的已安装的包将不可避免地变旧，并且不被用于任何现有项目。Pkg 会记录所有已使用项目的日志，这样便可通过遍历日志，明确知道哪些项目仍然存在以及这些项目使用了哪些包，剩下的包则会被删除。命令 `gc` 可执行此操作：

```
(v0.7) pkg> gc
    Active manifests at:
        `/Users/kristoffer/BinaryProvider/Manifest.toml`
        ...
        `/Users/kristoffer/Compat.jl/Manifest.toml`
   Deleted /Users/kristoffer/.julia/packages/BenchmarkTools/1cAj: 146.302 KiB
   Deleted /Users/kristoffer/.julia/packages/Cassette/BXVB: 795.557 KiB
   ...
   Deleted /Users/kristoffer/.julia/packages/WeakRefStrings/YrK6: 27.328 KiB
   Deleted 36 package installations: 113.205 MiB
```

请注意，只有在 `~/.julia/packages` 中的包才会被删除。

75.6 Creating your own packages

A package is a project with a `name`, `uuid` and `version` entry in the `Project.toml` file `src/PackageName.jl` file that defines the module `PackageName`. This file is executed when the package is loaded.

Generating files for a package

To generate files for a new package, use `pkg> generate`.

```
(v0.7) pkg> generate HelloWorld
```

This creates a new project `HelloWorld` with the following files (visualized with the external `tree` command):

```
shell> cd HelloWorld

shell> tree .
.
├── Project.toml
└── src
    └── HelloWorld.jl

1 directory, 2 files
```

The `Project.toml` file contains the name of the package, its unique UUID, its version, the author and eventual dependencies:

```
name = "HelloWorld"
uuid = "b4cd1eb8-1e24-11e8-3319-93036a3eb9f3"
version = "0.1.0"
author = ["Some One <someone@email.com>"]

[deps]
```

The content of `src/HelloWorld.jl` is:

```
module HelloWorld

greet() = print("Hello World!")

end # module
```

We can now activate the project and load the package:

```
pkg> activate .

julia> import HelloWorld

julia> HelloWorld.greet()
Hello World!
```

Adding dependencies to the project

Let's say we want to use the standard library package `Random` and the registered package `JSON` in our project. We simply add these packages (note how the prompt now shows the name of the newly generated project, since we are inside the `HelloWorld` project directory):

```
(HelloWorld) pkg> add Random JSON
 Resolving package versions...
  Updating "~/Documents/HelloWorld/Project.toml"
 [682c06a0] + JSON v0.17.1
 [9a3f8284] + Random
  Updating "~/Documents/HelloWorld/Manifest.toml"
 [34da2185] + Compat v0.57.0
 [682c06a0] + JSON v0.17.1
 [4d1e1d77] + Nullables v0.0.4
 ...
```

Both `Random` and `JSON` got added to the project's `Project.toml` file, and the resulting dependencies got added to the `Manifest.toml` file. The resolver has installed each package with the highest possible version, while still respecting the compatibility that each package enforce on its dependencies.

We can now use both `Random` and `JSON` in our project. Changing `src/HelloWorld.jl` to

```
module HelloWorld

import Random
import JSON

greet() = print("Hello World!")
greet_alien() = print("Hello ", Random.randstring(8))

end # module
```

and reloading the package, the new `greet_alien` function that uses `Random` can be used:

```
julia> HelloWorld.greet_alien()
Hello aT157rHV
```

Adding a build step to the package.

The build step is executed the first time a package is installed or when explicitly invoked with `build`. A package is built by executing the file `deps/build.jl`.

```
shell> cat deps/build.log
I am being built...

(HelloWorld) pkg> build
  Building HelloWorld → `deps/build.log`
 Resolving package versions...

shell> cat deps/build.log
I am being built...
```

If the build step fails, the output of the build step is printed to the console

```
shell> cat deps/build.jl
error("Ooops")

(HelloWorld) pkg> build
  Building HelloWorld → `deps/build.log`
 Resolving package versions...
┌ Error: Error building `HelloWorld`:
│ ERROR: LoadError: Ooops
│ Stacktrace:
│  [1] error(::String) at ./error.jl:33
│  [2] top-level scope at none:0
│  [3] include at ./boot.jl:317 [inlined]
│  [4] include_relative(::Module, ::String) at ./loading.jl:1071
│  [5] include(::Module, ::String) at ./sysimg.jl:29
│  [6] include(::String) at ./client.jl:393
│  [7] top-level scope at none:0
│ in expression starting at /Users/kristoffer/.julia/dev/Pkg/HelloWorld/deps/build.jl:1
└ @ Pkg.Operations Operations.jl:938
```

Adding tests to the package

When a package is tested the file `test/runtests.jl` is executed.

```
shell> cat test/runtests.jl
println("Testing...")
(HelloWorld) pkg> test
   Testing HelloWorld
 Resolving package versions...
Testing...
   Testing HelloWorld tests passed
```

Test-specific dependencies

Sometimes one might want to use some packages only at testing time but not enforce a dependency on them when the package is used. This is possible by adding dependencies to `[extras]` and a `test` target in `[targets]` to the Project file. Here we add the `Test` standard library as a test-only dependency by adding the following to the Project file:

```
[extras]
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"
```

```
[targets]
test = ["Test"]
```

We can now use `Test` in the test script and we can see that it gets installed on testing:

```
shell> cat test/runtests.jl
using Test
@test 1 == 1

(HelloWorld) pkg> test
   Testing HelloWorld
 Resolving package versions...
  Updating `/var/folders/64/76tk_g152sg6c6t0b4nkn1vw0000gn/T/tmpPzUPPw/Project.toml`
  [d8327f2a] + HelloWorld v0.1.0 [`~/.julia/dev/Pkg/HelloWorld`]
  [8dfed614] + Test
  Updating `/var/folders/64/76tk_g152sg6c6t0b4nkn1vw0000gn/T/tmpPzUPPw/Manifest.toml`
  [d8327f2a] + HelloWorld v0.1.0 [`~/.julia/dev/Pkg/HelloWorld`]
   Testing HelloWorld tests passed```
```

Compatibility

Compatibility refers to the ability to restrict what version of the dependencies that your project is compatible with. If the compatibility for a dependency is not given, the project is assumed to be compatible with all versions of that dependency.

Compatibility for a dependency is entered in the `Project.toml` file as for example:

```
[compat]
Example = "0.4.3"
```

After a compatibility entry is put into the project file, up can be used to apply it.

The format of the version specifier is described in detail below.

> **Info**
>
> There is currently no way to give compatibility from the Pkg REPL mode so for now, one has to manually edit the project file.

Version specifier format

Similar to other package managers, the Julia package manager respects semantic versioning (semver). As an example, a version specifier is given as e.g. `1.2.3` is therefore assumed to be compatible with the versions `[1.2.3 - 2.0.0)` where) is a non-inclusive upper bound. More specifically, a version specifier is either given as a **caret specifier**, e.g. `^1.2.3` or a **tilde specifier** `~1.2.3`. Caret specifiers are the default and hence `1.2.3 == ^1.2.3`. The difference between a caret and tilde is described in the next section. The intersection of multiple version specifiers can be formed by comma separating indiviual version specifiers.

Caret specifiers A caret specifier allows upgrade that would be compatible according to semver. An updated dependency is considered compatible if the new version does not modify the left-most non zero digit in the version specifier.

Some examples are shown below.

```
^1.2.3 = [1.2.3, 2.0.0)
^1.2 = [1.2.0, 2.0.0)
^1 =  [1.0.0, 2.0.0)
^0.2.3 = [0.2.3, 0.3.0)
^0.0.3 = [0.0.3, 0.0.4)
^0.0 = [0.0.0, 0.1.0)
^0 = [0.0.0, 1.0.0)
```

While the semver specification says that all versions with a major version of 0 are incompatible with each other, we have made that choice that a version given as `0.a.b` is considered compatible with `0.a.c` if `a != 0` and `c >= b`.

Tilde specifiers A tilde specifier provides more limited upgrade possibilities. With a tilde, only the last specified digit is allowed to increment by one. This gives the following example.

```
~1.2.3 = [1.2.3, 1.2.4)
~1.2 = [1.2.0, 1.3.0)
~1 = [1.0.0, 2.0.0)
```

Inequality specifiers

Inequalities can also be used to specify version ranges:

```
>= 1.2.3 = [1.2.3,  ∞)
≥ 1.2.3 = [1.2.3,  ∞)
= 1.2.3 = [1.2.3, 1.2.3]
< 1.2.3 = [0.0.0, 1.2.2]
```

75.7 预编译项目

REPL 命令 `precompile` 可用于预编译项目中的所有依赖。例如，这样做可以

```
(HelloWorld) pkg> update; precompile
```

更新依赖项，然后预编译它们。

75.8 预览模式

如果你只想查看某个命令运行的效果，但不想更改包的状态，则可以 `preview` 该命令。例如：

```
(HelloWorld) pkg> preview add Plots
```

或

```
(HelloWorld) pkg> preview up
```

将向你展示添加 `Plots`、或者进行完全升级分别会对你的项目产生的影响。但是，这没有安装任何东西，也不会触及你的 `Project.toml` 和 `Manifest.toml`。

75.9 使用别人的项目

只需使用诸如 `git clone` 来克隆项目，接着 `cd` 到项目目录并调用

```
(v0.7) pkg> activate .

(SomeProject) pkg> instantiate
```

如果该项目包含了清单，则会以与该清单给定的相同状态安装包。否则，它将解析为与项目兼容的最新版本的依赖项。

75.10 References

This section describes the ”API mode” of interacting with Pkg.jl which is recommended for non-interactive usage, in i.e. scripts. In the REPL mode packages (with associated version, UUID, URL etc) are parsed from strings, for example, `"Package#master"`,`"Package@v0.1"`, `"www.mypkg.com/MyPkg#my/feature"`. It is possible to use strings as arguments for simple commands in the API mode (like `Pkg.add(["PackageA", "PackageB"])`, more complicated commands, that e.g. specify URLs or version range, uses a more structured format over strings. This is done by creating an instance of a `PackageSpec` which are passed in to functions.

`Pkg.PackageSpec` – Function.

```
PackageSpec(name::String, [uuid::UUID, version::VersionNumber])
PackageSpec(; name, url, path, rev, version, mode, level)
```

A PackageSpec is a representation of a package with various metadata. This includes:

- The `name` of the package.
- The package’s unique `uuid`.
- A `version` (for example when adding a package). When upgrading, can also be an instance of

the enum `UpgradeLevel`.

- A `url` and an optional git `rev`ision. `rev` can be a branch name or a git commit SHA1.
- A local `path`. This is equivalent to using the `url` argument but can be more descriptive.
- A `mode`, which is an instance of the enum `PackageMode`, with possible values `PKGMODE_PROJECT`

(the default) or `PKGMODE_MANIFEST`. Used in e.g. `Pkg.rm`.

Most functions in Pkg take a `Vector` of PackageSpec and do the operation on all the packages in the vector.

Below is a comparison between the REPL version and the API version:

REPL	API
`Package`	`PackageSpec("Package")`
`Package@0.2`	`PackageSpec(name="Package", version="0.2")`
`Package=a67d...`	`PackageSpec(name="Package", uuid="a67d...")`
`Package#master`	`PackageSpec(name="Package", rev="master")`
`local/path#feature`	`PackageSpec(path="local/path"; rev="feature")`
`www.mypkg.com`	`PackageSpec(url="www.mypkg.com")`
`--manifest Package`	`PackageSpec(name="Package", mode=PKGSPEC_MANIFEST)`
`--major Package`	`PackageSpec(name="Package", version=PKGLEVEL_MAJOR)`

`Pkg.PackageMode` – Type.

```
PackageMode
```

An enum with the instances

- `PKGMODE_MANIFEST`

- PKGMODE_PROJECT

Determines if operations should be made on a project or manifest level. Used as an argument to PackageSpec or as an argument to Pkg.rm.

Pkg.UpgradeLevel – Type.

```
UpgradeLevel
```

An enum with the instances

- UPLEVEL_FIXED
- UPLEVEL_PATCH
- UPLEVEL_MINOR
- UPLEVEL_MAJOR

Determines how much a package is allowed to be updated. Used as an argument to PackageSpec or as an argument to Pkg.update.

Pkg.add – Function.

```
Pkg.add(pkg::Union{String, Vector{String}})
Pkg.add(pkg::Union{PackageSpec, Vector{PackageSpec}})
```

Add a package to the current project. This package will be available by using the `import` and `using` keywords in the Julia REPL, and if the current project is a package, also inside that package.

Examples

```
Pkg.add("Example") # Add a package from registry
Pkg.add(PackageSpec(name="Example", version="0.3")) # Specify version; latest release in the 0.3
↪  series
Pkg.add(PackageSpec(name="Example", version="0.3.1")) # Specify version; exact release
Pkg.add(PackageSpec(url="https://github.com/JuliaLang/Example.jl", rev="master")) # From url to
↪  remote gitrepo
Pkg.add(PackageSpec(url="/remote/mycompany/juliapackages/OurPackage"))` # From path to local
↪  gitrepo
```

See also PackageSpec.

Pkg.develop – Function.

```
Pkg.develop(pkg::Union{String, Vector{String}})
Pkg.develop(pkgs::Union{Packagespec, Vector{Packagespec}})
```

Make a package available for development by tracking it by path. If pkg is given with only a name or by a URL, the package will be downloaded to the location specified by the environment variable `JULIA_PKG_DEVDIR`, with `.julia/dev` as the default.

If pkg is given as a local path, the package at that path will be tracked.

Examples

```
# By name
Pkg.develop("Example")

# By url
Pkg.develop(PackageSpec(url="https://github.com/JuliaLang/Compat.jl"))

# By path
Pkg.develop(PackageSpec(path="MyJuliaPackages/Package.jl"))
```

See also PackageSpec

Pkg.activate – Function.

```
Pkg.activate([s::String]; shared::Bool=false)
```

Activate the environment at s. The active environment is the environment that is modified by executing package commands. The logic for what path is activated is as follows:

- If shared is true, the first existing environment named s from the depots in the depot stack will be activated. If no such environment exists, create and activate that environment in the first depot.
- If s is an existing path, then activate the environment at that path.
- If s is a package in the current project and s is tracking a path, then activate the environment at the tracked path.
- Otherwise, s is interpreted as a non-existing path, which is then activated.

If no argument is given to activate, then activate the home project. The home project is specified by either the --project command line option to the julia executable, or the JULIA_PROJECT environment variable.

Examples

```
Pkg.activate()
Pkg.activate("local/path")
Pkg.activate("MyDependency")
```

Pkg.rm – Function.

```
Pkg.rm(pkg::Union{String, Vector{String}})
Pkg.rm(pkg::Union{PackageSpec, Vector{PackageSpec}})
```

Remove a package from the current project. If the mode of pkg is PKGMODE_MANIFEST also remove it from the manifest including all recursive dependencies of pkg.

See also PackageSpec, PackageMode.

Pkg.update – Function.

```
Pkg.update(; level::UpgradeLevel=UPLEVEL_MAJOR, mode::PackageMode = PKGMODE_PROJECT)
Pkg.update(pkg::Union{String, Vector{String}})
Pkg.update(pkg::Union{PackageSpec, Vector{PackageSpec}})
```

Update a package pkg. If no posistional argument is given, update all packages in the manifest if mode is PKGMODE_MANIFEST and packages in both manifest and project if mode is PKGMODE_PROJECT. If no positional argument is given, level can be used to control by how much packages are allowed to be upgraded (major, minor, patch, fixed).

See also PackageSpec, PackageMode, UpgradeLevel.

`Pkg.test` – Function.

```
Pkg.test(; kwargs...)
Pkg.test(pkg::Union{String, Vector{String}; kwargs...)
Pkg.test(pkgs::Union{PackageSpec, Vector{PackageSpec}}; kwargs...)
```

Keyword arguments:

- `coverage::Bool=false`: enable or disable generation of coverage statistics.
- `julia_args::Union{Cmd, Vector{String}}`: options to be passed the test process.
- `test_args::Union{Cmd, Vector{String}}`: test arguments (`ARGS`) available in the test process.

  Julia 1.3

  `julia_args` and `test_args` requires at least Julia 1.3.

Run the tests for package `pkg`, or for the current project (which thus needs to be a package) if no positional argument is given to `Pkg.test`. A package is tested by running its `test/runtests.jl` file.

The tests are run by generating a temporary environment with only `pkg` and its (recursive) dependencies in it. If a manifest exists, the versions in that manifest are used, otherwise a feasible set of packages is resolved and installed.

During the tests, test-specific dependencies are active, which are given in the project file as e.g.

```
[extras]
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"

[targets]
test = ["Test"]
```

The tests are executed in a new process with `check-bounds=yes` and by default `startup-file=no`. If using the startup file (`~/.julia/config/startup.jl`) is desired, start julia with `--startup-file=yes`. Inlining of functions during testing can be disabled (for better coverage accuracy) by starting julia with `--inline=no`.

`Pkg.build` – Function.

```
Pkg.build(; verbose = false)
Pkg.build(pkg::Union{String, Vector{String}}; verbose = false)
Pkg.build(pkgs::Union{PackageSpec, Vector{PackageSpec}}; verbose = false)
```

Run the build script in `deps/build.jl` for `pkg` and all of its dependencies in depth-first recursive order. If no argument is given to `build`, the current project is built, which thus needs to be a package. This function is called automatically on any package that gets installed for the first time. `verbose = true` prints the build output to `stdout`/`stderr` instead of redirecting to the `build.log` file.

`Pkg.pin` – Function.

```
Pkg.pin(pkg::Union{String, Vector{String}})
Pkg.pin(pkgs::Union{PackageSpec, Vector{PackageSpec}})
```

Pin a package to the current version (or the one given in the PackageSpec) or to a certain git revision. A pinned package is never updated.

Examples

```
Pkg.pin("Example")
Pkg.pin(PackageSpec(name="Example", version="0.3.1"))
```

`Pkg.free` – Function.

```
Pkg.free(pkg::Union{String, Vector{String}})
Pkg.free(pkgs::Union{PackageSpec, Vector{PackageSpec}})
```

If pkg is pinned, remove the pin. If pkg is tracking a path, e.g. after `Pkg.develop`, go back to tracking registered versions.

Examples

```
Pkg.free("Package")
Pkg.free(PackageSpec("Package"))
```

`Pkg.instantiate` – Function.

```
Pkg.instantiate(; verbose = false)
```

If a `Manifest.toml` file exists in the current project, download all the packages declared in that manifest. Otherwise, resolve a set of feasible packages from the `Project.toml` files and install them. `verbose = true` prints the build output to `stdout`/`stderr` instead of redirecting to the `build.log` file.

`Pkg.resolve` – Function.

```
Pkg.resolve()
```

Update the current manifest with potential changes to the dependency graph from packages that are tracking a path.

`Pkg.setprotocol!` – Function.

```
setprotocol!(;
    domain::AbstractString = "github.com",
    protocol::Union{Nothing, AbstractString}=nothing
)
```

Set the protocol used to access hosted packages when adding a url or developing a package. Defaults to delegating the choice to the package developer (`protocol === nothing`). Other choices for `protocol` are `"https"` or `"git"`.

Examples

```
julia> Pkg.setprotocol!(domain = "github.com", protocol = "ssh")

julia> Pkg.setprotocol!(domain = "gitlab.mycompany.com")
```

Chapter 76

Printf

`Printf.@printf` – Macro.

```
@printf([io::IOStream], "%Fmt", args...)
```

Print `args` using C `printf` style format specification string, with some 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.

Optionally, an `IOStream` may be passed as the first argument to redirect output.

See also: `@sprintf`

Examples

```
julia> @printf("%f %F %f %F\n", Inf, Inf, NaN, NaN)
Inf Inf NaN NaN


julia> @printf "%.0f %.1f %f\n" 0.5 0.025 -0.0078125
1 0.0 -0.007813
```

`Printf.@sprintf` – Macro.

```
@sprintf("%Fmt", args...)
```

Return `@printf` formatted output as string.

Examples

```
julia> s = @sprintf "this is a %s %15.1f" "test" 34.567;

julia> println(s)
this is a test            34.6
```

Chapter 77

性能分析

`Profile.@profile` – Macro.

```
@profile
```

`@profile <expression>` runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

`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.

`Profile.print` – Function.

```
print([io::IO = stdout,] [data::Vector]; 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, whereas `:count` sorts in order of number of collected samples.
- `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.

```
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.

`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. Each instruction pointer corresponds to a single line of code; backtraces generally consist of a long list of instruction pointers. Default 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)`.

`Profile.fetch` – Function.

```
fetch() -> data
```

Returns a reference to the internal buffer of backtraces. Note that subsequent operations, like `clear`, can affect `data` unless you first make a copy. 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.

`Profile.retrieve` – Function.

```
retrieve() -> 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.

`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.

`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.

Chapter 78

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 1.3.1 (2019-12-30)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |


julia>
```

To exit the interactive session, type ^D —the control key together with the d key on a blank line —or type `exit()` followed by the return or enter key. The REPL greets you with a banner and a `julia>` prompt.

78.1 不同的提示符模式

Julian 模式

REPL 有四种主要的操作模式。第一个也是最常见的是 Julian 提示符。这是默认的操作模式；每个新行最初都以 `julia>` 开头。就在这里，你可以输入 Julia 表达式。在输入完整表达式后按下 return 或 enter 将执行该表达式，并显示最后一个表达式的结果。

```
julia> string(1 + 2)
"3"
```

交互式运行有许多独特的实用功能。除了显示结果外，REPL 还将结果绑定到变量 `ans` 上。一行的尾随分号可用作禁止显示结果的标志。

```
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>` are parsed, others are removed. This makes it is possible to paste a chunk of code 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
```

Help mode

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.
```

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
```

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.

78.2 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.

78.3 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> 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/
```

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.

Tab completion can also help completing fields:

```
julia> import UUIDs

julia> UUIDs.uuid[TAB]
uuid1        uuid4         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.

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
```

78.4 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
```

78.5 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: d=done, a=all, n=none]
   [ ] apple
 > [X] orange
   [X] grape
   [ ] strawberry
   [ ] blueberry
   [X] peach
   [ ] lemon
   [ ] lime
You like the following fruits:
  - orange
  - grape
  - peach
```

Customization / Configuration

All interface customization is done through the keyword only `TerminalMenus.config()` function.

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=:na`: If `:wrap` then wrap the cursor around top and bottom, if :nowrap do not wrap cursor
- `supress_output::Bool=false`: For testing. If true, menu will not be printed to console.
- `ctrl_c_interrupt::Bool=true`: If `false`, return empty on ^C, if `true` throw InterruptException() on ^C

Examples

```
julia> menu = MultiSelectMenu(options, pagesize=5);

julia> request(menu) # ASCII is used by default
[press: d=done, a=all, n=none]
   [ ] apple
   [X] orange
   [ ] grape
 > [X] strawberry
v  [ ] blueberry
Set([4, 2])

julia> TerminalMenus.config(charset=:unicode)
```

```
julia> request(menu)
[press: d=done, a=all, n=none]
   ⬚ apple
   ✓ orange
   ⬚ grape
 → ✓ strawberry
↓  ⬚ blueberry
Set([4, 2])

julia> TerminalMenus.config(checked="YEP!", unchecked="NOPE", cursor='')

julia> request(menu)
[press: d=done, a=all, n=none]
   NOPE apple
   YEP! orange
   NOPE grape
  YEP! strawberry
↓  NOPE blueberry
Set([4, 2])
```

78.6 References

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

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
^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 79

随机数

Random number generation in Julia uses the Mersenne Twister library via `MersenneTwister` objects. Julia has a global RNG, which is used by default. Other RNG types can be plugged in by inheriting the `AbstractRNG` type; they can then be used to have multiple streams of random numbers. Besides `MersenneTwister`, Julia also provides the `RandomDevice` RNG type, which is a wrapper over the OS provided entropy.

大部分与随机数生成相关的函数都接受一个可选的 `AbstractRNG` 对象作为第一个参数，如果不指定则使用全局默认的。此外，某些函数还接受一个可选的维度参数 `dims...`（可以是元组）来生成随机数组。

一个 `MersenneTwister` 或 `RandomDevice` RNG 能够生成如下类型的随机数：`Float16`, `Float32`, `Float64`, `BigFloat`, `Bool`, `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int128`, `UInt128`, `BigInt`（或者这些类型的复数）。随机浮点数在 $[0, 1)$ 区间均匀生成。由于 `BigInt` 代表无界的整数，必须要指定区间（如 `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`。

> **Warn**
>
> 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.

79.1 Random numbers module

`Random.Random` – Module.

```
Random
```

Support for generating random numbers. Provides `rand`, `randn`, `AbstractRNG`, `MersenneTwister`, and `RandomDevice`.

79.2 Random generation functions

`Base.rand` – Function.

```
rand([rng=GLOBAL_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`.

> **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
```

> **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 `BitSet`. 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=GLOBAL_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 Array{Float64,1}:
 0.5908446386657102
 0.7667970365022592
 0.5662374165061859
 0.4600853424625171
 0.7940257103317943
```

`Random.bitrand` – Function.

```
bitrand([rng=GLOBAL_RNG], [dims...])
```

Generate a BitArray of random boolean values.

Examples

```
julia> rng = MersenneTwister(1234);

julia> bitrand(rng, 10)
10-element BitArray{1}:
 0
 1
 1
 1
 1
 0
 1
 0
 0
 1
```

Base.randn – Function.

```
randn([rng=GLOBAL_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).

Examples

```
julia> using Random

julia> rng = MersenneTwister(1234);

julia> randn(rng, ComplexF64)
0.6133070881429037 - 0.6376291670853887im

julia> randn(rng, ComplexF32, (2, 3))
2×3 Array{Complex{Float32},2}:
 -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=GLOBAL_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 Array{Float64,1}:
  0.8673472019512456
 -0.9017438158568171
 -0.4944787535042339
 -0.9029142938652416
  0.8644013132535154
```

Random.randexp – Function.

```
randexp([rng=GLOBAL_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 Array{Float64,2}:
 1.5167    1.30652   0.344435
 0.604436  2.78029   0.418516
 0.695867  0.693292  0.643644
```

Random.randexp! – Function.

```
randexp!([rng=GLOBAL_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 Array{Float64,1}:
 2.4835053723904896
 1.516703605376473
 0.6044364871025417
 0.6958665886385867
 1.3065196315496677
```

Random.randstring – Function.

```
randstring([rng=GLOBAL_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!(0); randstring()
"0IPrGg0J"

julia> randstring(MersenneTwister(0), 'a':'z', 6)
"aszvqk"

julia> randstring("ACGT")
"TATCGGTC"
```

Note

chars can be any collection of characters, of type Char or UInt8 (more efficient), provided rand can randomly pick characters from it.

79.3 Subsequences, permutations and shuffling

Random.randsubseq – Function.

```
randsubseq([rng=GLOBAL_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, collect(1:8), 0.3)
2-element Array{Int64,1}:
 7
 8
```

Random.randsubseq! – Function.

```
randsubseq!([rng=GLOBAL_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, collect(1:8), 0.3);

julia> S
2-element Array{Int64,1}:
 7
 8
```

Random.randperm – Function.

```
randperm([rng=GLOBAL_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 Array{Int64,1}:
 2
 1
 4
 3
```

Random.randperm! – Function.

```
randperm!([rng=GLOBAL_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 Array{Int64,1}:
 2
 1
 4
 3
```

Random.randcycle – Function.

```
randcycle([rng=GLOBAL_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 Array{Int64,1}:
 3
 5
 4
 6
 1
 2
```

Random.randcycle! – Function.

```
randcycle!([rng=GLOBAL_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 Array{Int64,1}:
 3
 5
 4
 6
 1
 2
```

Random.shuffle – Function.

```
shuffle([rng=GLOBAL_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 Array{Int64,1}:
  6
  1
 10
  2
  3
  9
  5
  7
  4
  8
```

Random.shuffle! – Function.

```
shuffle!([rng=GLOBAL_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 Array{Int64,1}:
  2
```

```
15
 5
14
 1
 9
10
 6
11
 3
16
 7
 4
12
 8
13
```

79.4 Generators (creation and seeding)

`Random.seed!` – Function.

```
seed!([rng=GLOBAL_RNG], seed) -> rng
seed!([rng=GLOBAL_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 thread-local generator.

Examples

```
julia> Random.seed!(1234);

julia> x1 = rand(2)
2-element Array{Float64,1}:
 0.590845
 0.766797

julia> Random.seed!(1234);

julia> x2 = rand(2)
2-element Array{Float64,1}:
 0.590845
 0.766797

julia> x1 == x2
true

julia> rng = MersenneTwister(1234); rand(rng, 2) == x1
true

julia> MersenneTwister(1) == Random.seed!(rng, 1)
true

julia> rand(Random.seed!(rng), Bool) # not reproducible
true
```

```
julia> rand(Random.seed!(rng), Bool)
false

julia> rand(MersenneTwister(), Bool) # not reproducible either
true
```

Random.AbstractRNG – Type.

```
AbstractRNG
```

Supertype for random number generators such as MersenneTwister and RandomDevice.

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 Array{Float64,1}:
 0.5908446386657102
 0.7667970365022592

julia> rng = MersenneTwister(1234);

julia> x2 = rand(rng, 2)
2-element Array{Float64,1}:
 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.

79.5 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(10)

julia> rand(MersenneTwister(0), Die)
Die(16)

julia> rand(Die, 3)
3-element Array{Die,1}:
 Die(5)
 Die(20)
 Die(9)

julia> a = Vector{Die}(undef, 3); rand!(a)
3-element Array{Die,1}:
 Die(11)
 Die(20)
 Die(10)
```

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))
2

julia> rand(Die(4), 3)
3-element Array{Any,1}:
 1
 4
 2
```

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 80

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. (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 save the data or embed it into the test code. 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 81

SHA

用法非常直接：

```
julia> using SHA

julia> bytes2hex(sha256("test"))
"9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08"
```

每个导出函数（SHA-1, SHA-2 224, 256, 384, 512, 以及 SHA-3 224, 256, 384, 512 函数在本文撰写时都已实现）都可以接受 `Array{UInt8}`, `ByteString` 或 `IO` 对象。这使计算文件校验和变得轻而易举：

```
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
```

注意 `/tmp/text.txt` 文件结尾缺少换行符。Julia 会自动在 `julia>` 提示符前插入换行符。

由于 `sha256` 通常指的是 `sha2_256`，因此提供了函数名简写，将 `shaxxx()` 函数调用映射到 `sha2_xxx()`。SHA-3 不存在这样的俗称，用户必须使用完整的函数名 `sha3_xxx()`。

shaxxx() 接受 UInt8 类型的 AbstractString 和类数组对象（NTuple 和 Array）。

请注意，在本文撰写时，SHA-3 代码还未进行优化，因此会比 SHA-2 慢大约一个数量级。

Chapter 82

序列化

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. In general, this process will not work if the reading and writing are done by different versions of Julia, or an instance of Julia with a different system image. 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.

```
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. It has been designed with simplicity and performance as a goal and does not validate the data read. Malformed data can result in process termination. The caller has to 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 83

共享数组

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.

SharedArrays.SharedVector – Type.

```
SharedVector
```

A one-dimensional SharedArray.

SharedArrays.SharedMatrix – Type.

```
SharedMatrix
```

A two-dimensional SharedArray.

Distributed.procs – Method.

```
procs(S::SharedArray)
```

Get the vector of processes mapping the shared array.

SharedArrays.sdata – Function.

```
sdata(S::SharedArray)
```

Returns the actual Array object backing S.

SharedArrays.indexpids – Function.

```
indexpids(S::SharedArray)
```

Returns 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.

SharedArrays.localindices – Function.

```
localindices(S::SharedArray)
```

Returns 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.

Chapter 84

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.

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.

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.

Examples

```
julia> getipaddr()
ip"192.168.1.28"

julia> getipaddr(IPv6)
ip"fe80::9731:35af:e1c5:6e49"
```

Sockets.getipaddrs – Function.

```
getipaddrs(; loopback::Bool=false) -> Vector{IPAddr}
```

Get the IPv4 addresses of the local machine.

```
getipaddrs(addr_type::Type{T}; loopback::Bool=false) where T<:IPAddr -> Vector{T}
```

Get the IP addresses of the local machine of the specified type.

The `loopback` keyword argument dictates whether loopback addresses are included.

> **Julia 1.2**
>
> This function is available as of Julia 1.2.

Examples

```
julia> getipaddrs()
2-element Array{IPv4,1}:
 ip"10.255.0.183"
 ip"172.17.0.1"

julia> getipaddrs(IPv6)
2-element Array{IPv6,1}:
 ip"fe80::9731:35af:e1c5:6e49"
 ip"fe80::445e:5fff:fe5d:5500"
```

> **Missing docstring.**
>
> Missing docstring for `Sockets.islinklocaladdr`. Check Documenter's build log for details.

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"
```

Missing docstring.

Missing docstring for `Sockets.DNSError`. Check Documenter's build log for details.

`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(Sockets.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
```

Returns 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
```

Returns 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) -> (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.

Base.bind – Function.

```
bind(socket::Union{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: foo
Stacktrace:
[...]
```

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.

`Sockets.quickack` – Function.

```
quickack(socket::Union{TCPServer, TCPSocket}, enable::Bool)
```

On Linux systems, the TCP_QUICKACK is disabled or enabled on `socket`.

Chapter 85

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。

85.1 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} <: AbstractSparseMatrix{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. If your SparseMatrixCSC object contains unsorted row indices, one quick way to sort them is by doing a double 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 structural nonzeros. 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:
```

```
  [1, 1]  =  0
  [2, 2]  =  2
  [1, 3]  =  1
  [3, 3]  =  0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Int64,Int64} with 2 stored entries:
  [2, 2]  =  2
  [1, 3]  =  1
```

85.2 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
```

As for `SparseMatrixCSC`, the `SparseVector` type can also contain explicitly stored zeros. (See Sparse Matrix Storage.).

85.3 稀疏向量与矩阵构造函数

The simplest way to create a sparse array is to use a function equivalent to the `zeros` function that Julia provides for working with dense arrays. To produce a sparse array instead, you can use the same name with an `sp` prefix:

```
julia> spzeros(3)
3-element SparseVector{Float64,Int64} with 0 stored entries
```

The `sparse` function is often a handy way to construct sparse arrays. For example, to construct a sparse matrix we can input a vector `I` of row indices, a vector `J` of column indices, and a vector `V` of stored values (this is also known as the COO (coordinate) format). `sparse(I,J,V)` then constructs a sparse matrix such that `S[I[k], J[k]] = V[k]`. The equivalent sparse vector constructor is `sparsevec`, which takes the (row) index vector `I` and the vector `V` with the stored values and constructs a sparse vector `R` such that `R[I[k]] = V[k]`.

```
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:
  [1,  4]  =  1
  [4,  7]  =  2
  [5,  9]  =  3
  [3, 18]  =  -5

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 Array{CartesianIndex{2},1}:
 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 Array{Int64,1}:
 1
 3
 4
 5
```

Another way to create a sparse array is to convert a dense array into a sparse array using the `sparse` function:

```
julia> sparse(Matrix(1.0I, 5, 5))
5×5 SparseMatrixCSC{Float64,Int64} with 5 stored entries:
  [1, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
  [4, 4]  =  1.0
  [5, 5]  =  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
```

85.4 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.

85.5 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.
`Array(S)`	`sparse(A)`	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 86

Sparse Arrays

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.

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}‘.

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}.

SparseArrays.SparseVector – Type.

```
SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
```

Vector type for storing sparse vectors.

SparseArrays.SparseMatrixCSC – Type.

```
SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{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.

SparseArrays.sparse – Function.

```
sparse(A)
```

Convert an AbstractMatrix A into a sparse matrix.

Examples

```
julia> A = Matrix(1.0I, 3, 3)
3×3 Array{Float64,2}:
 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, 1]  =  1.0
  [2, 2]  =  1.0
  [3, 3]  =  1.0
```

```
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, 1]  =  1
  [2, 2]  =  2
  [3, 3]  =  3
```

`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
```

```
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
```

```
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
```

`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
```

`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:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2

julia> nnz(A)
3
```

SparseArrays.findnz – Function.

```
findnz(A)
```

Return a tuple `(I, J, V)` where `I` and `J` are the row and column indices of the stored ("structurally nonzero") 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, 1]  =  1
  [1, 2]  =  2
  [3, 2]  =  4
  [2, 3]  =  3

julia> findnz(A)
([1, 1, 3, 2], [1, 2, 2, 3], [1, 2, 4, 3])
```

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
```

SparseArrays.spdiagm – Function.

```
spdiagm(kv::Pair{<:Integer,<:AbstractVector}...)
spdiagm(m::Integer, n::Ingeger, 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 (if `size=nothing`), 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:
  [2, 1]  =  1
  [1, 2]  =  4
  [3, 2]  =  2
  [2, 3]  =  3
  [4, 3]  =  3
  [3, 4]  =  2
  [5, 4]  =  4
  [4, 5]  =  1
```

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:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2
  [4, 4]  =  4
  [5, 5]  =  4
```

SparseArrays.sprand – Function.

```
sprand([rng],[type],m,[n],p::AbstractFloat,[rfn])
```

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`). Nonzero values are sampled from the distribution specified by `rfn` and have the type `type`. The uniform distribution is used in case `rfn` is not specified. The optional `rng` argument specifies a random number generator, see Random Numbers.

Examples

```
julia> sprand(Bool, 2, 2, 0.5)
2×2 SparseMatrixCSC{Bool,Int64} with 1 stored entry:
  [2, 2]  =  1

julia> sprand(Float64, 3, 0.75)
3-element SparseVector{Float64,Int64} with 1 stored entry:
  [3]  =  0.298614
```

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 2 stored entries:
  [1, 2]  =  0.586617
  [2, 2]  =  0.297336
```

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:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2

julia> nonzeros(A)
3-element Array{Int64,1}:
 2
 2
 2
```

SparseArrays.rowvals – Function.

```
rowvals(A::SparseMatrixCSC)
```

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:
  [1, 1]  =  2
  [2, 2]  =  2
  [3, 3]  =  2

julia> rowvals(A)
3-element Array{Int64,1}:
 1
 2
 3
```

SparseArrays.nzrange – Function.

```
nzrange(A::SparseMatrixCSC, 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
```

SparseArrays.droptol! – Function.

```
droptol!(A::SparseMatrixCSC, tol; trim::Bool = true)
```

Removes stored values from `A` whose absolute value is less than or equal to `tol`, optionally trimming resulting excess space from `A.rowval` and `A.nzval` when `trim` is `true`.

```
droptol!(x::SparseVector, tol; trim::Bool = true)
```

Removes stored values from `x` whose absolute value is less than or equal to `tol`, optionally trimming resulting excess space from `A.rowval` and `A.nzval` when `trim` is `true`.

SparseArrays.dropzeros! – Function.

```
dropzeros!(A::SparseMatrixCSC; trim::Bool = true)
```

Removes stored numerical zeros from `A`, optionally trimming resulting excess space from `A.rowval` and `A.nzval` when `trim` is `true`.

For an out-of-place version, see dropzeros. For algorithmic information, see `fkeep!`.

```
dropzeros!(x::SparseVector; trim::Bool = true)
```

Removes stored numerical zeros from `x`, optionally trimming resulting excess space from `x.nzind` and `x.nzval` when `trim` is `true`.

For an out-of-place version, see dropzeros. For algorithmic information, see `fkeep!`.

SparseArrays.dropzeros – Function.

```
dropzeros(A::SparseMatrixCSC; trim::Bool = true)
```

Generates a copy of `A` and removes stored numerical zeros from that copy, optionally trimming excess space from the result's `rowval` and `nzval` arrays when `trim` is `true`.

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, 1]  =  1.0
  [2, 2]  =  0.0
  [3, 3]  =  1.0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 1]  =  1.0
  [3, 3]  =  1.0
```

```
dropzeros(x::SparseVector; trim::Bool = true)
```

Generates a copy of x and removes numerical zeros from that copy, optionally trimming excess space from the result's `nzind` and `nzval` arrays when `trim` is `true`.

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
```

`SparseArrays.permute` – Function.

```
permute(A::SparseMatrixCSC{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) == A.n`). Row-permutation `p`'s length must match `A`'s row count (`length(p) == A.m`).

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, 1]  =  1
  [1, 2]  =  5
  [2, 2]  =  2
  [2, 3]  =  6
  [3, 3]  =  3
  [3, 4]  =  7
  [4, 4]  =  4

julia> permute(A, [4, 3, 2, 1], [1, 2, 3, 4])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
  [4, 1]  =  1
  [3, 2]  =  2
  [4, 2]  =  5
  [2, 3]  =  3
  [3, 3]  =  6
  [1, 4]  =  4
  [2, 4]  =  7

julia> permute(A, [1, 2, 3, 4], [4, 3, 2, 1])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
  [3, 1]  =  7
  [4, 1]  =  4
  [2, 2]  =  6
```

```
  [3, 2]  =  3
  [1, 3]  =  5
  [2, 3]  =  2
  [1, 4]  =  1
```

`Base.permute!` – Method.

```
permute!(X::SparseMatrixCSC{Tv,Ti}, A::SparseMatrixCSC{Tv,Ti},
         p::AbstractVector{<:Integer}, q::AbstractVector{<:Integer},
         [C::SparseMatrixCSC{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::SparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer},
         q::AbstractVector{<:Integer}[, C::SparseMatrixCSC{Tv,Ti},
         [workcolptr::Vector{Ti}]]) where {Tv,Ti}
```

`X`'s dimensions must match those of `A` (`X.m == A.m` and `X.n == A.n`), and `X` must have enough storage to accommodate all allocated entries in `A` (`length(X.rowval) >= nnz(A)` and `length(X.nzval) >= nnz(A)`). Column-permutation `q`'s length must match `A`'s column count (`length(q) == A.n`). Row-permutation `p`'s length must match `A`'s row count (`length(p) == A.m`).

`C`'s dimensions must match those of `transpose(A)` (`C.m == A.n` and `C.n == A.m`), and `C` must have enough storage to accommodate all allocated entries in `A` (`length(C.rowval) >= nnz(A)` and `length(C.nzval) >= 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`.

Chapter 87

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 an IID drawn from that generative distribution. 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`.

A pre-computed `mean` may be provided.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the standard deviation over dimensions, and `means` may contain means for each dimension of `itr`.

> **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.

Statistics.stdm – Function.

```
stdm(itr, m; corrected::Bool=true)
```

Compute the sample standard deviation of collection `itr`, with known mean(s) `m`.

The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `itr` is an IID drawn from that generative distribution. 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`.

A pre-computed `mean` may be provided.

If `itr` is an `AbstractArray`, `dims` can be provided to compute the standard deviation over dimensions, and `m` may contain means for each dimension of `itr`.

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.

Statistics.var – Function.

```
var(itr; dims, corrected::Bool=true, mean=nothing)
```

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 an IID drawn from that generative distribution. For arrays, this computation is equivalent to calculating sum((itr .- mean(itr)).^2) / (length(itr) - 1)). Ifcorrectedistrue, then the sum is scaled withn-1, whereas the sum is scaled withnifcorrectedisfalsewithnthe number of elements initr‘.

A pre-computed mean may be provided.

If itr is an AbstractArray, dims can be provided to compute the variance over dimensions, and mean may contain means for each dimension of itr.

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.

Statistics.varm – Function.

```
varm(itr, m; dims, corrected::Bool=true)
```

Compute the sample variance of collection itr, with known mean(s) m.

The algorithm returns an estimator of the generative distribution's variance under the assumption that each entry of itr is an IID drawn from that generative distribution. 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, and m may contain means for each dimension of itr.

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.

Statistics.cor – Function.

```
cor(x::AbstractVector)
```

Return the number one.

```
cor(X::AbstractMatrix; dims::Int=1)
```

Compute the Pearson correlation matrix of the matrix X along the dimension dims.

```
cor(x::AbstractVector, y::AbstractVector)
```

Compute the Pearson correlation between the vectors x and y.

```
cor(X::AbstractVecOrMat, Y::AbstractVecOrMat; dims=1)
```

Compute the Pearson correlation between the vectors or matrices X and Y along the dimension dims.

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).

```
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).

```
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 $\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})^*$.

```
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).

Statistics.mean! – Function.

```
mean!(r, v)
```

Compute the mean of v over the singleton dimensions of r, and write results to r.

Examples

```
julia> v = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> mean!([1., 1.], v)
2-element Array{Float64,1}:
 1.5
 3.5

julia> mean!([1. 1.], v)
1×2 Array{Float64,2}:
 2.0  3.0
```

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> mean(1:20)
10.5

julia> mean([1, missing, 3])
missing

julia> mean(skipmissing([1, missing, 3]))
2.0
```

```
mean(f::Function, itr)
```

Apply the function `f` to each element of collection `itr` and take the mean.

```
julia> mean(√, [1, 2, 3])
1.3820881233139908

julia> mean([√1, √2, √3])
1.3820881233139908
```

```
mean(f::Function, 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> 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 Array{Float64,2}:
 1.3820881233139908
 2.2285192400943226
```

```
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> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> mean(A, dims=1)
1×2 Array{Float64,2}:
 2.0  3.0

julia> mean(A, dims=2)
2×1 Array{Float64,2}:
 1.5
 3.5
```

`Statistics.median!` – Function.

```
median!(v)
```

Like `median`, but may overwrite the input vector.

`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> 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
```

```
median(A::AbstractArray; dims)
```

Compute the median of an array along the given dimensions.

Examples

```
julia> median([1 2; 3 4], dims=1)
1×2 Array{Float64,2}:
 2.0  3.0
```

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.

```
middle(x, y)
```

Compute the middle of two reals x and y, which is equivalent in both value and type to computing their mean (`(x + y) / 2`).

```
middle(range)
```

Compute the middle of a range, which consists of computing the mean of its extrema. Since a range is sorted, the mean is performed with the first and last element.

```
julia> middle(1:10)
5.5
```

```
middle(a)
```

Compute the middle of an array a, which consists of finding its extrema and then computing their mean.

```
julia> a = [1,2,3.6,10.9]
4-element Array{Float64,1}:
  1.0
  2.0
  3.6
 10.9

julia> middle(a)
5.95
```

Statistics.quantile! – Function.

```
quantile!([q::AbstractArray, ] v::AbstractVector, p; sorted=false)
```

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.

Quantiles are computed via linear interpolation between the points `((k-1)/(n-1), v[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 default.

> **Note**
>
> An `ArgumentError` is thrown if v contains `NaN` or `missing` values.

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365

Examples

```
julia> x = [3, 2, 1];

julia> quantile!(x, 0.5)
2.0

julia> x
3-element Array{Int64,1}:
 1
 2
 3

julia> y = zeros(3);

julia> quantile!(y, x, [0.1, 0.5, 0.9]) === y
true

julia> y
3-element Array{Float64,1}:
 1.2
 2.0
 2.8
```

`Statistics.quantile` – Function.

```
quantile(itr, p; sorted=false)
```

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.

Quantiles are computed via linear interpolation between the points `((k-1)/(n-1), v[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 default.

> **Note**
>
> An `ArgumentError` is thrown if `itr` contains `NaN` or `missing` values. Use the `skipmissing` function to omit `missing` entries and compute the quantiles of non-missing values.

- Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361-365

Examples

```
julia> quantile(0:20, 0.5)
10.0

julia> quantile(0:20, [0.1, 0.5, 0.9])
3-element Array{Float64,1}:
  2.0
 10.0
 18.0

julia> quantile(skipmissing([1, 10, missing]), 0.5)
5.5
```

Chapter 88

单元测试

88.1 测试 Julia Base 库

Julia is under rapid development and has an extensive test suite to verify functionality across multiple platforms. If you build Julia from source, you can run this test suite with `make test`. In a binary install, you can run the test suite using `Base.runtests()`.

`Base.runtests` – Function.

```
Base.runtests(tests=["all"]; ncores=ceil(Int, Sys.CPU_THREADS / 2),
              exit_on_error=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 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

88.2 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.

Simple unit testing can be performed with the `@test` and `@test_throws` macros:

`Test.@test` – Macro.

```
@test ex
@test f(args...) key=val ...
```

Tests that the expression `ex` evaluates to `true`. Returns a `Pass Result` if it does, a `Fail Result` if it is `false`, and an `Error Result` if it could not be evaluated.

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`).

`Test.@test_throws` – Macro.

```
@test_throws exception expr
```

Tests that the expression `expr` throws `exception`. The exception may specify either a type, or a value (which will be tested for equality by comparing fields). Note that `@test_throws` does not support a trailing keyword form.

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
```

For example, suppose we want to check our new function `foo(x)` works as expected:

```
julia> using Test

julia> foo(x) = length(x)^2
foo (generic function with 1 method)
```

如果条件为真，则返回 `Pass`：

```
julia> @test foo("bar") == 9
Test Passed

julia> @test foo("fizz") >= 10
Test Passed
```

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
      Thrown: MethodError
```

88.3 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.

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] [option=val  ...] ["description"] begin ... end
@testset [CustomTestSet] [option=val  ...] ["description $v"] for v in (...) ... end
@testset [CustomTestSet] [option=val  ...] ["description $v, $w"] for v in (...), w in (...) ...
↪  end
```

Starts a new test set, or multiple test sets if a `for` loop is provided.

If no custom testset type is given it defaults to creating a `DefaultTestSet`. `DefaultTestSet` records all the results and, if there are any `Fails` or `Errors`, 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 does not take any options.

The description string accepts interpolation from the loop indices. If no description is provided, one is constructed based on the variables.

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 `@testsets` 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
trigonometric identities |    4      4
```

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
Foo Tests     |    3      3
```

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
Foo Tests     |    8      8
```

In the event that a nested test set has no failures, as happened here, it will be hidden in the summary. 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
Foo Tests     |    3     1      4
  Animals     |    2            2
  Arrays      |    1     1      2
ERROR: Some tests did not pass: 3 passed, 1 failed, 0 errored, 0 broken.
```

88.4 其他测试宏

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

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
```

`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)
Variables
```

```
  #self#::Core.Compiler.Const(f, false)
  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}
[...]
```

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 Regexs 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:

```
@test_logs (:info,"Doing foo with n=2") (:debug,"Iteration 1") (:debug,"Iteration 2") min_level=
     Debug foo(2)
```

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
```

`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"
```

`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.

`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`.

88.5 损坏的测试

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`.

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)
```

`@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.

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)
```

88.6 自定义 `AbstractTestSet` 类型

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.

`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. One common use for this function is to record the testset to the parent's results list, using `get_testset`.

`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.

`Test.get_testset_depth` – Function.

```
get_testset_depth()
```

Returns the number of active test sets, not including the default test set

`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.

定义一个基本的 `AbstractTestSet` 子类：

```
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
```

使用测试集：

```
@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
```

Chapter 89

UUIDs

UUIDs.uuid1 – Function.

```
uuid1([rng::AbstractRNG=GLOBAL_RNG]) -> 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.

Examples

```
julia> rng = MersenneTwister(1234);

julia> uuid1(rng)
UUID("cfc395e8-590f-11e8-1f13-43a2532b2fa8")
```

UUIDs.uuid4 – Function.

```
uuid4([rng::AbstractRNG=GLOBAL_RNG]) -> UUID
```

Generates a version 4 (random or pseudo-random) universally unique identifier (UUID), as specified by RFC 4122.

Examples

```
julia> rng = MersenneTwister(1234);

julia> uuid4(rng)
UUID("196f2941-2d58-45ba-9f13-43a2532b2fa8")
```

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("196f2941-2d58-45ba-9f13-43a2532b2fa8")

julia> u5 = uuid5(u4, "julia")
UUID("b37756f8-b0c0-54cd-a466-19b3d25683bc")
```

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 90

Unicode

`Unicode.isassigned` – Function.

```
Unicode.isassigned(c) -> Bool
```

Returns `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.normalize` – Function.

```
Unicode.normalize(s::AbstractString, normalform::Symbol)
```

Normalize the string `s` according to one of the four "normal forms" of the Unicode standard: `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 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

For example, NFKC corresponds to the options `compose=true, compat=true, stable=true`.

Examples

```
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"
```

`Unicode.graphemes` – Function.

```
graphemes(s::AbstractString) -> GraphemeIterator
```

Returns 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.)

Part VI

Developer Documentation

Chapter 91

反射与自我检查

Julia 提供了多种运行时的反射功能。

91.1 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。

91.2 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
```

注意 `fieldnames(DataType)` 给出了 `DataType` 本身的每个字段的名称，其中的一个字段是上面示例中提到的 `types` 字段。

91.3 子类型

任何 DataType 的直接子类型都可以通过使用 subtypes 来列出。例如抽象 DataType AbstractFloat 有四个（具体的）子类型：

```
julia> subtypes(AbstractFloat)
4-element Array{Any,1}:
 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。

91.4 DataType 布局

用 C 代码接口时，DataType 的内部表现非常重要。有几个函数可以检查这些细节。

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。

91.5 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。

91.6 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
))))
```

91.7 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)

define i64 @"julia_+_130862"(i64, i64) {
top:
    %2 = add i64 %1, %0
    ret i64 %2
}
```

For more informations 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 92

Documentation of Julia's Internals

92.1 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 `ui/repl.c`.

`main()` 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 `process_options()` in `base/client.jl`.

`jl_parse_opts()` stores command line options in the global `jl_options` struct.

`julia_init()`

`julia_init()` in `task.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_modules()` 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 `ui/repl.c` and `main()` calls `true_main(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.

true_main()

true_main() 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.process_options which calls jl_parse_input_line("println("Hello World!")") to create an expression object and Base.eval() to execute it.

Base.eval

Base.eval() was mapped to jl_f_top_eval by jl_init_primitives().

jl_f_top_eval() calls jl_toplevel_eval_in(jl_main_module, ex), where ex is the parsed expression println("Hello World!").

jl_toplevel_eval_in() calls jl_toplevel_eval_flex() which calls eval() in interpreter.c.

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().

92.2 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。

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()	interpreter.c	
jl_interpret_toplevel_expr()	interpreter.c	
jl_toplevel_eval_flex()	toplevel.c	
jl_toplevel_eval()	toplevel.c	
jl_toplevel_eval_in()	builtins.c	
jl_f_top_eval()	builtins.c	

First we will focus on the AST, since it is needed to write macros.

Surface syntax AST

Front end ASTs consist almost entirely of Exprs and atoms (e.g. symbols, numbers). There is generally a different expression head for each visually distinct syntactic form. Examples will be given in s-expression syntax. Each parenthesized list corresponds to an Expr, where the first element is the head. For example (call f x) corresponds to Expr(:call, :f, :x) in Julia.

Calls

Input	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))

do syntax:

```
f(x) do a,b
    body
end
```

parses as `(do (call f x) (-> (tuple a b) (block body)))`.

Operators

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	(: a b)
a:b:c	(: a b c)
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

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))
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 for y in z, a in b]	(comprehension x (= y z) (= a b))
T[x for y in z]	(typed_comprehension T x (= y z))
(a, b, c)	(tuple a b c)
(a; b; c)	(block a (block b c))

Macros

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)`

Strings

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")`

Doc string syntax:

```
"some docs"
f(x) = x
```

parses as `(macrocall (|.| Core '@doc) (line) "some docs" (= (call f x) (block x)))`.

Imports and such

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)`

`using` has the same representation as `import`, but with expression head `:using` instead of `:import`.

Numbers

Julia supports more number types than many scheme implementations, so not all numbers are represented directly as scheme numbers in the AST.

Input	AST
`11111111111111111111`	`(macrocall @int128_str (null) "11111111111111111111")`
`0xffffffffffffffff`	`(macrocall @uint128_str (null) "0xffffffffffffffff")`
`1111...many digits...`	`(macrocall @big_str (null) "1111....")`

Block forms

A block of statements is parsed as `(block stmt1 stmt2 ...)`.

If statement:

```
if a
    b
elseif c
    d
else
    e
end
```

parses as:

```
(if a (block (line 2) b)
    (elseif (block (line 3) c) (block (line 4) d)
            (block (line 5 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, conditional branches (`gotoifnot`), and returns.

- `Slot`

  Identifies arguments and local variables by consecutive numbering. `Slot` is an abstract type with subtypes `SlotNumber` and `TypedSlot`. Both types have an integer-valued `id` field giving the slot index. Most slots have the same type at all uses, and so are represented with `SlotNumber`. The types of these slots are found in the `slottypes` field of their `MethodInstance` object. Slots that require per-use type annotations are represented with `TypedSlot`, which has a `typ` field.

- `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.

- `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.

- `gotoifnot`

  Conditional branch. If `args[1]` is false, goes to the index identified in `args[2]`.

- `=`

  Assignment. In the IR, the first argument is always a Slot 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 `false` if unknown. If a symbol, then the expression first behaves like the 1-argument form above. This argument is ignored from then on. When this is `false`, it means a method is being added strictly by type, `(::T)(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 `Base.splat(new)` if `new` were a first-class function, hence the name.

- `return`

  Returns its argument as the value of the enclosing function.

- `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:length(args[3])]` : arguments

    The values for all the arguments (with types of each given in args[3]).

  - `args[(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.

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`, `isstaged`, `pure`

  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 sentinal 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:

  - 2 - assigned (only false if there are *no* assignment statements with this var on the left)
  - 8 - const (currently unused for local variables)
  - 16 - statically assigned once
  - 32 - 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:

  - 0 = inbounds
  - 1,2 = <reserved> inlinehint,always-inline,noinline
  - 3 = <reserved> strict-ieee (strictfp)
  - 4-6 = <unused>
  - 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).

- `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.

- `pure`

  Whether this is known to be a pure function of its arguments, without respect to the state of the method caches or other mutable global state.

92.3 More about types

If you've used Julia for a while, you understand the fundamental role that types play. Here we try to get under the hood, focusing particularly on Parametric Types.

Types and sets (and `Any` and `Union{}`/`Bottom`)

It's perhaps easiest to conceive of Julia's type system in terms of sets. While programs manipulate individual values, a type refers to a set of values. This is not the same thing as a collection; for example a `Set` of values is itself a single `Set` value. Rather, a type describes a set of *possible* values, expressing uncertainty about which value we have.

A *concrete* type T describes the set of values whose direct tag, as returned by the `typeof` function, is T. An *abstract* type describes some possibly-larger set of values.

`Any` describes the entire universe of possible values. `Integer` is a subset of `Any` that includes `Int`, `Int8`, and other concrete types. Internally, Julia also makes heavy use of another type known as `Bottom`, which can also be written as `Union{}`. This corresponds to the empty set.

Julia's types support the standard operations of set theory: you can ask whether T1 is a "subset" (subtype) of T2 with `T1 <: T2`. Likewise, you intersect two types using `typeintersect`, take their union with `Union`, and compute a type that contains their union with `typejoin`:

```
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}
```

While these operations may seem abstract, they lie at the heart of Julia. For example, method dispatch is implemented by stepping through the items in a method list until reaching one for which the type of the argument tuple is a subtype of the method signature. For this algorithm to work, it's important that methods be sorted by their specificity, and that the search begins with the most specific methods. Consequently, Julia also implements a partial order on types; this is achieved by functionality that is similar to `<:`, but with differences that will be discussed below.

UnionAll types

Julia's type system can also express an *iterated union* of types: a union of types over all values of some variable. This is needed to describe parametric types where the values of some parameters are not known.

For example, Array has two parameters as in Array{Int,2}. If we did not know the element type, we could write Array{T,2} where T, which is the union of Array{T,2} for all values of T: Union{Array{Int8,2}, Array{Int16,2}, ...}.

Such a type is represented by a UnionAll object, which contains a variable (T in this example, of type TypeVar), and a wrapped type (Array{T,2} in this example).

Consider the following methods:

```
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].

Let's look at these types a little more closely:

```
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}
```

This indicates that Array actually names a UnionAll type. There is one UnionAll type for each parameter, nested. The syntax Array{Int,2} is equivalent to Array{Int}{2}; internally each UnionAll is instantiated with a particular variable value, one at a time, outermost-first. This gives a natural meaning to the omission of trailing type parameters; Array{Int} gives a type equivalent to Array{Int,N} where N.

A TypeVar is not itself a type, but rather should be considered part of the structure of a UnionAll type. Type variables have lower and upper bounds on their values (in the fields lb and ub). The symbol name is purely cosmetic. Internally, TypeVars are compared by address, so they are defined as mutable types to ensure that "different" type variables can be distinguished. However, by convention they should not be mutated.

One can construct TypeVars manually:

```
julia> TypeVar(:V, Signed, Real)
Signed<:V<:Real
```

There are convenience versions that allow you to omit any of these arguments except the name symbol.

The syntax Array{T} where T<:Integer is lowered to

```
let T = TypeVar(:T,Integer)
    UnionAll(T, Array{T})
end
```

so it is seldom necessary to construct a TypeVar manually (indeed, this is to be avoided).

Free variables

The concept of a *free* type variable is extremely important in the type system. We say that a variable `V` is free in type `T` if `T` does not contain the `UnionAll` that introduces variable `V`. For example, the type `Array{Array{V} where V<:Integer}` has no free variables, but the `Array{V}` part inside of it does have a free variable, `V`.

A type with free variables is, in some sense, not really a type at all. Consider the type `Array{Array{T}} where T`, which refers to all homogeneous arrays of arrays. The inner type `Array{T}`, seen by itself, might seem to refer to any kind of array. However, every element of the outer array must have the *same* array type, so `Array{T}` cannot refer to just any old array. One could say that `Array{T}` effectively "occurs" multiple times, and `T` must have the same value each "time".

For this reason, the function `jl_has_free_typevars` in the C API is very important. Types for which it returns true will not give meaningful answers in subtyping and other type functions.

TypeNames

The following two `Array` types are functionally equivalent, yet print differently:

```
julia> TV, NV = TypeVar(:T), TypeVar(:N)
(T, N)

julia> Array
Array

julia> Array{TV,NV}
Array{T,N}
```

These can be distinguished by examining the `name` field of the type, which is an object of type `TypeName`:

```
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
    kwsorter: #undef
    module: Module Core
    : Int64 0
    : Int64 0
```

In this case, the relevant field is `wrapper`, which holds a reference to the top-level type used to make new `Array` types.

```
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
```

The `wrapper` field of `Array` points to itself, but for `Array{TV,NV}` it points back to the original definition of the type.

What about the other fields? `hash` assigns an integer to each type. To examine the `cache` field, it's helpful to pick a type that is less heavily used than Array. Let's first create our own type:

```
julia> struct MyType{T,N} end

julia> MyType{Int,2}
MyType{Int64,2}

julia> MyType{Float32, 5}
MyType{Float32,5}
```

When you instantiate a parametric type, each concrete type gets saved in a type cache (`MyType.body.body.name.cache`). However, instances containing free type variables are not cached.

Tuple types

Tuple types constitute an interesting special case. For dispatch to work on declarations like `x::Tuple`, the type has to be able to accommodate any tuple. Let's check the parameters:

```
julia> Tuple
Tuple

julia> Tuple.parameters
svec(Vararg{Any,N} where N)
```

Unlike other types, tuple types are covariant in their parameters, so this definition permits `Tuple` to match any type of tuple:

```
julia> typeintersect(Tuple, Tuple{Int,Float64})
Tuple{Int64,Float64}

julia> typeintersect(Tuple{Vararg{Any}}, Tuple{Int,Float64})
Tuple{Int64,Float64}
```

However, if a variadic (`Vararg`) tuple type has free variables it can describe different kinds of tuples:

```
julia> typeintersect(Tuple{Vararg{T} where T}, Tuple{Int,Float64})
Tuple{Int64,Float64}

julia> typeintersect(Tuple{Vararg{T}} where T, Tuple{Int,Float64})
Union{}
```

Notice that when T is free with respect to the `Tuple` type (i.e. its binding `UnionAll` type is outside the `Tuple` type), only one T value must work over the whole type. Therefore a heterogeneous tuple does not match.

Finally, it's worth noting that `Tuple{}` is distinct:

```
julia> Tuple{}
Tuple{}

julia> Tuple{}.parameters
svec()

julia> typeintersect(Tuple{}, Tuple{Int})
Union{}
```

What is the "primary" tuple-type?

```
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
```

so `Tuple == Tuple{Vararg{Any}}` is indeed the primary type.

Diagonal types

Consider the type `Tuple{T,T} where T`. A method with this signature would look like:

```
f(x::T, y::T) where {T} = ...
```

According to the usual interpretation of a `UnionAll` type, this T ranges over all types, including `Any`, so this type should be equivalent to `Tuple{Any,Any}`. However, this interpretation causes some practical problems.

First, a value of T needs to be available inside the method definition. For a call like `f(1, 1.0)`, it's not clear what T should be. It could be `Union{Int,Float64}`, or perhaps `Real`. Intuitively, we expect the declaration `x::T` to mean `T === typeof(x)`. To make sure that invariant holds, we need `typeof(x) === typeof(y) === T` in this method. That implies the method should only be called for arguments of the exact same type.

It turns out that being able to dispatch on whether two values have the same type is very useful (this is used by the promotion system for example), so we have multiple reasons to want a different interpretation of `Tuple{T,T} where T`. To make this work we add the following rule to subtyping: if a variable occurs more than once in covariant position, it is restricted to ranging over only concrete types. ("Covariant position" means that only `Tuple` and `Union` types occur between an occurrence of a variable and the `UnionAll` type that introduces it.) Such variables are called "diagonal variables" or "concrete variables".

So for example, `Tuple{T,T} where T` can be seen as `Union{Tuple{Int8,Int8}, Tuple{Int16,Int16}, ...}`, where T ranges over all concrete types. This gives rise to some interesting subtyping results. For example `Tuple{Real,Real}` is not a subtype of `Tuple{T,T} where T`, because it includes some types like `Tuple{Int8,Int16}` where the two elements have different types. `Tuple{Real,Real}` and `Tuple{T,T} where T` have the non-trivial intersection `Tuple{T,T} where T<:Real`. However, `Tuple{Real}` *is* a subtype of `Tuple{T} where T`, because in that case T occurs only once and so is not diagonal.

Next consider a signature like the following:

```
f(a::Array{T}, x::T, y::T) where {T} = ...
```

In this case, T occurs in invariant position inside `Array{T}`. That means whatever type of array is passed unambiguously determines the value of T —we say T has an *equality constraint* on it. Therefore in this case the diagonal rule is not really necessary, since the array determines T and we can then allow x and y to be of any subtypes of T. So variables that occur in invariant position are never considered diagonal. This choice of behavior is slightly controversial —- some feel this definition should be written as

```
f(a::Array{T}, x::S, y::S) where {T, S<:T} = ...
```

to clarify whether x and y need to have the same type. In this version of the signature they would, or we could introduce a third variable for the type of y if x and y can have different types.

The next complication is the interaction of unions and diagonal variables, e.g.

```
f(x::Union{Nothing,T}, y::T) where {T} = ...
```

Consider what this declaration means. y has type T. x then can have either the same type T, or else be of type `Nothing`. So all of the following calls should match:

```
f(1, 1)
f("", "")
f(2.0, 2.0)
f(nothing, 1)
f(nothing, "")
f(nothing, 2.0)
```

These examples are telling us something: when x is `nothing::Nothing`, there are no extra constraints on y. It is as if the method signature had `y::Any`. Indeed, we have the following type equivalence:

```
(Tuple{Union{Nothing,T},T} where T) == Union{Tuple{Nothing,Any}, Tuple{T,T} where T}
```

The general rule is: a concrete variable in covariant position acts like it's not concrete if the subtyping algorithm only *uses* it once. When x has type `Nothing`, we don't need to use the T in `Union{Nothing,T}`; we only use it in the second slot. This arises naturally from the observation that in `Tuple{T} where T` restricting T to concrete types makes no difference; the type is equal to `Tuple{Any}` either way.

However, appearing in *invariant* position disqualifies a variable from being concrete whether that appearance of the variable is used or not. Otherwise types can behave differently depending on which other types they are compared to, making subtyping not transitive. For example, consider

Tuple{Int,Int8,Vector{Integer}} <: Tuple{T,T,Vector{Union{Integer,T}}} where T

If the T inside the Union is ignored, then T is concrete and the answer is "false" since the first two types aren't the same. But consider instead

Tuple{Int,Int8,Vector{Any}} <: Tuple{T,T,Vector{Union{Integer,T}}} where T

Now we cannot ignore the T in the Union (we must have T == Any), so T is not concrete and the answer is "true". That would make the concreteness of T depend on the other type, which is not acceptable since a type must have a clear meaning on its own. Therefore the appearance of T inside `Vector` is considered in both cases.

Subtyping diagonal variables

The subtyping algorithm for diagonal variables has two components: (1) identifying variable occurrences, and (2) ensuring that diagonal variables range over concrete types only.

The first task is accomplished by keeping counters `occurs_inv` and `occurs_cov` (in `src/subtype.c`) for each variable in the environment, tracking the number of invariant and covariant occurrences, respectively. A variable is diagonal when `occurs_inv == 0 && occurs_cov > 1`.

The second task is accomplished by imposing a condition on a variable's lower bound. As the subtyping algorithm runs, it narrows the bounds of each variable (raising lower bounds and lowering upper bounds) to keep track of the range of variable values for which the subtype relation would hold. When we are done evaluating the body of a `UnionAll` type whose variable is diagonal, we look at the final values of the bounds. Since the variable must be concrete, a contradiction occurs if its lower bound could not be a subtype of a concrete type. For example, an abstract type like `AbstractArray` cannot be a subtype of a concrete type, but a concrete type like `Int` can be, and the empty type `Bottom` can be as well. If a lower bound fails this test the algorithm stops with the answer `false`.

For example, in the problem `Tuple{Int,String} <: Tuple{T,T} where T`, we derive that this would be true if T were a supertype of `Union{Int,String}`. However, `Union{Int,String}` is an abstract type, so the relation does not hold.

This concreteness test is done by the function `is_leaf_bound`. Note that this test is slightly different from `jl_is_leaf_type`, since it also returns `true` for `Bottom`. Currently this function is heuristic, and does not catch all possible concrete types. The difficulty is that whether a lower bound is concrete might depend on the values of other type variable bounds. For example, `Vector{T}` is equivalent to the concrete type `Vector{Int}` only if both the upper and lower bounds of T equal `Int`. We have not yet worked out a complete algorithm for this.

Introduction to the internal machinery

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.

Because the subtyping code is used heavily in the REPL itself–and hence breakpoints in this code get triggered often–it will be easiest if you make the following definition:

```
julia> function mysubtype(a,b)
           ccall(:jl_breakpoint, Cvoid, (Any,), nothing)
           a <: b
       end
```

and then set a breakpoint in `jl_breakpoint`. Once this breakpoint gets triggered, you can set breakpoints in other functions.

As a warm-up, try the following:

```
mysubtype(Tuple{Int,Float64}, Tuple{Integer,Real})
```

We can make it more interesting by trying a more complex case:

```
mysubtype(Tuple{Array{Int,2}, Int8}, Tuple{Array{T}, T} where T)
```

Subtyping and method sorting

The `type_morespecific` functions are used for imposing a partial order on functions in method tables (from most-to-least specific). Specificity is strict; if a is more specific than b, then a does not equal b and b is not more specific than a.

If a is a strict subtype of b, then it is automatically considered more specific. From there, `type_morespecific` employs some less formal rules. For example, `subtype` is sensitive to the number of arguments, but `type_morespecific` may not be. In particular, `Tuple{Int,AbstractFloat}` is more specific than `Tuple{Integer}`, even though it is not a subtype. (Of `Tuple{Int,AbstractFloat}` and `Tuple{Integer,Float64}`, neither is more specific than the other.) Likewise, `Tuple{Int,Vararg{Int}}` is not a subtype of `Tuple{Integer}`, but it is considered more specific. However, `morespecific` does get a bonus for length: in particular, `Tuple{Int,Int}` is more specific than `Tuple{Int,Vararg{Int}}`.

If you're debugging how methods get sorted, it can be convenient to define the function:

```
type_morespecific(a, b) = ccall(:jl_type_morespecific, Cint, (Any,Any), a, b)
```

which allows you to test whether tuple type a is more specific than tuple type b.

92.4 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);
```

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

92.5 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

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 ui/repl.c gets called. This function processes the command line arguments, filling in the jl_options struct and setting the variable ARGS. It then initializes 在 ui/repl.c 中的 C 语言的函数 main() 被调用。这个函数处理命令行参数，填充到 jl_options 结构图并且设置变了 ARGS 。接下来初始化 Julia (通过调用 julia_init in task.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 compiler 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.

92.6 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.

92.7 High-level Overview of the Native-Code Generation Process

Representation of Pointers

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.

92.8 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

在上述派发过程中，添加一个新方法在概念上所需的只是（1）一个元组类型，以及（2）方法体的代码。jl_method_def 实现了此操作。jl_first_argument_datatype 会被调用，用来从第一个参数的类型中提取相关的方法表。这比派发期间的相应过程复杂得多，因为参数元组类型可能是抽象类型。例如，我们可以定义：

```
(::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：

```
=== typeof sizeof <: isa typeassert throw tuple getfield setfield! fieldtype
nfields isdefined arrayref arrayset arraysize applicable invoke apply_type _apply
_expr svec
```

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

关键字参数的工作方式是将每个具有关键字参数的方法表与一个特殊的隐藏函数对象相关联。该函数称为「keyword argument sorter」、「keyword sorter」或「kwsorter」，存储在 MethodTable 对象的 kwsorter 字段中。在 kwsorter 函数的每个定义与通常的方法表中的某个函数具有相同的参数，除了前面还有一个 NamedTuple 参数，该参数给出所传递关键字参数的名称和值。kwsorter 的作用是根据名称将关键字参数移到预先要求的位置，并对任何所需的默认值表达式进行求值和替换。其返回结果是一个通常的位置参数列表，接着会被传递给另一个由编译器生成的函数。

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：

```
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：

```
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：

```
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：

```
kwfunc(circle)(merge((color = red,), other), circle, (0,0), 1.0)
```

kwfunc（也在 Core 中）可获取被调用函数的 kwsorter。关键字 splatting 函数（编写为 other...）调用具名元组 merge 函数。此函数进一步解包了 other 的每个元素，预期中每个元素包含两个值（一个符号和一个值）。当然，如果所有 splatted 参数都是具名元组，则可使用更高效的实现。请注意，原来的 circle 被传递，以处理闭包。

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。

92.9 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 semicolons, 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

92.10 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++.

92.11 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:
 0.200586  0.066423
 0.298614  0.956753

julia> S2 = view(A, 1, :, 2:3)
3×2 view(::Array{Float64,3}, 1, :, 2:3) with eltype Float64:
```

```
 0.200586  0.066423
 0.246837  0.646691
 0.648882  0.276021
```

`view` drops "singleton" dimensions (ones that are specified by an `Int`), so both `S1` and `S2` are two-dimensional `SubArrays`. 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{Base.S1`
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 Array{Int64,1}:
 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 Array{Int64,1}:
 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.

92.12 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->ofset` 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.

92.13 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.

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.

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 `n`th 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 detail 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 larget number of features. An overview of this process is in `src/processor.cpp`.

92.14 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/`.

File	Description
`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
`llvm-simdloop.cpp`	Custom LLVM pass for `@simd`
`sys.c`	I/O and operating system utility functions

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.

Building Julia with a different version of LLVM

The default version of LLVM is specified in `deps/Versions.make`. 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 = 6.0.1
```

Besides the LLVM release numerals, you can also use `LLVM_VER = svn` 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 diagonstics 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`.

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, one could do:

```
opt -load libjulia.so -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.

Alternatively, you can use `--output-jit-bc jit.bc` to obtain a trace of all IR passed to the JIT. This is useful for code that cannot be run as part of the sysimg generation process (e.g. because it creates unserializable state). However, the resulting `jit.bc` does not include sysimage data, and can thus not be used as such.

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:

```
%bitcast = bitcast @any_unoptimized_call to %jl_value_t *(*)(%jl_value_t *, %jl_value_t *)
call cc 37 %jl_value_t *%bitcast(%jl_value_t *%arg1, %jl_value_t *%arg2)
```

The special `cc 37` annotation marks the fact that this call site is really using the jlcall calling convention. 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. In the code the calling convention number is represented by the `JLCALL_F_CC` constant. In addition, there is the `JLCALL_CC` calling convention which functions similarly, but omits the first argument.

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.

92.15 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 `ui/` 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);
}
```

92.16 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 = 1:length(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 被 sum 包裹时，对 checkbounds(A,i) 的调用会被忽略。如果存在多层包裹，最多只有一个 @boundscheck 被忽略。这个规则用来防止将来代码被改变时潜在的多余忽略。

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).

92.17 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

  flisp itself is already threadsafe, this lock only protects the `jl_ast_context_list_t` pool

The following is a leaf lock (level 2), and only acquires level 1 locks (safepoint) internally:

- typecache
- Module->lock

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.

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, kwsorter type) : 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

92.18 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` 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.

92.19 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
```

92.20 类型推导

类型推导是如何工作的

类型推导指的是由输入值的类型推导其他值得类型得过程。

这两篇博客 (1, 2) 描述了 Julia 的类型推导实现。

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.Compiler)` 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`
params = Core.Compiler.Params(typemax(UInt))  # parameter is the world age,
                                              # typemax(UInt) -> most recent
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(m, types, sparams, optimize, params)
```

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 `inlining_pass`. However, if your question is "why didn't my function inline?" then you will most likely be interested in `isinlineable` and its primary callee, `inline_worthy`. `isinlineable` handles a number of special cases (e.g., critical functions like `next` and `done`, incorporating a bonus for functions that return tuples, etc.). The main decision-making happens in `inline_worthy`, which returns `true` if the function should be inlined.

`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 run this yourself by following the sketch below, where `f` is your function and `tt` is the Tuple-type of the arguments:

```
# A demo on `fill(3.5, (2, 3))`
f = fill
tt = Tuple{Float64, Tuple{Int,Int}}
# Create the objects we need to interact with the compiler
params = Core.Compiler.Params(typemax(UInt))
mi = Base.method_instances(f, tt)[1]
ci = code_typed(f, tt)[1][1]
opt = Core.Compiler.OptimizationState(mi, params)
# Calculate cost of each statement
cost(stmt::Expr) = Core.Compiler.statement_cost(stmt, -1, ci, opt.sptypes, opt.slottypes,
↪  opt.params)
cost(stmt) = 0
cst = map(cost, ci.code)

# output
```

```
31-element Array{Int64,1}:
  0
  0
 20
  4
  1
  1
  1
  0
  0
  0
  □
  0
  0
  0
  0
  0
  0
  0
  0
  0
```

The output is a `Vector{Int}` holding the estimated cost of each statement in `ci.code`. Note that `ci` includes the consequences of inlining callees, and consequently the costs do too.

Chapter 93

Developing/debugging Julia's C code

93.1 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:

- 自举启动阶段的段错误 (`sysimg.jl`)
- 运行脚本时的段错误
- 启动 Julia 时发生的段错误

版本/环境信息

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 1.3.1
Commit 2d5741174c (2019-12-30 21:36 UTC)
Platform Info:
  OS: Linux (x86_64-pc-linux-gnu)
  CPU: Intel(R) Xeon(R) CPU
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  TRAVIS_JULIA_VERSION = 1.3
  JULIA_PROJECT = @.
```

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.

术语表

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.....

93.2 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 signal to function property. 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 SIGUSRS 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
```

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.

93.3 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`.

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.

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.

93.4 Sanitizer support

General considerations

Using Clang's sanitizers obviously require 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 matches that of the LLVM library used within Julia.

An easy solution is to have an 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.

Address Sanitizer (ASAN)

For detecting or debugging memory bugs, you can use Clang's address sanitizer (ASAN). By compiling with `SANITIZE=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` takes 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.

Memory Sanitizer (MSAN)

For detecting use of uninitialized memory, you can use Clang's memory sanitizer (MSAN) by compiling with `SANITIZE_MEMORY=1`.