Julia 文档

发布 ***0.3.0-prerelease***

作者 **JuliaLang** 译者 **JuliaCN**

4月 18, 2017

Contents

Part I

Julia 手册

CHAPTER 1

简介

科学计算对性能一直有着最高的需求， 但现在这个领域的专家开始大量使用比较慢的动态语言来完成日常工作。我们相信有很多使用动态语言的理由， 所以我们不会舍弃这样的特性。幸运的是， 现代语言设计和编译器技术使得为原型设计提供单一的高效开发环境，并且配置高性能的应用成为可能。Julia 语言在这其中扮演了这样一个角色：

作为灵活的动态语言，适合科学和数值计算，性能可与传统静态类型语言媲美。

由于 Julia 的编译器和其它语言比如 Python 或 R 有所不同，一开始您或许会觉得 Julia 中什么样的代码运行效率高，什么样的代码运行效率低似乎并不很直观。如果您发现 Julia 变慢了，我们非常建议您在尝试其它功能前读一下 代码性能优化 。只要您理解 Julia 的工作方式，就会很容易地写出运行效率甚至可以和 C 相媲美的代码。

通过使用类型推断和 即时(JIT)编译 ，以及 LLVM ， Julia 具有可选的类型声明，重载，高性能等特性。Julia 是多编程范式的，包含指令式、函数式和面向对象编程的特征。 它提供了简易和简洁的高等数值计算，它类似于 R 、 MATLAB 和 Python ， 支持一般用途的编程。 为了达到这个目的，Julia 在数学编程语言的基础上，参考了不少流行动态语言： Lisp 、 Perl 、 Python 、 Lua 和 Ruby 。

Julia 与传统动态语言最大的区别是：

- 核心语言很小；标准库是用 Julia 本身写的，如整数运算在内的基础运算
- 完善的类型，方便构造对象和做类型声明
- 基于参数类型进行函数 重载
- 参数类型不同，自动生成高效、专用的代码
- 高性能，接近静态编译语言，如 C 语言

动态语言是有类型的：每个对象，不管是基础的还是用户自定义的，都有类型。许多动态语言没有类型声明，意味着它不能告诉编译器值的类型，也就不能准确的判断出类型。静态语言必须告诉编译器值的类型，类型仅存在于编译时，在运行时则不能更改。在 Julia 中，类型本身就是运行时对象，同时它也可以把信息传递给编译器。

重载函数由参数（参数列表）的类型来区别，调用函数时传入的参数类型，决定了选取哪个函数来进行调用。对于数学领域的程序设计来说，这种方式比起传统面向对象程序设计中操作属于某个对象的方法的方式更显自然。在 Julia 中运算符仅仅是函数的别名。程序员可以为新数据类型定义 “+” 的新方法，原先的代码就可以无缝地重载到新数据类型上。

因为运行时类型推断（得益于可选的类型声明），以及从开始就看重性能，Julia 的计算性能超越了其他动态语言，甚至可与静态编译语言媲美。在大数据处理的问题上，性能一直是决定性的因素：在刚刚过去的十年中，数据量还在以摩尔定律增长着。

Julia 想要变成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 的优势还在于：

- 免费开源（ MIT 协议 ）
- 自定义类型与内置类型同样高效、紧凑
- 不需要把代码向量化；非向量化的代码跑得也很快
- 为并行和分布式计算而设计
- 轻量级“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
- 高效支持 Unicode, 包括且不只 UTF-8
- 直接调用 C 函数（不需封装或 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

CHAPTER 2

开始

Julia 的安装，不管是使用编译好的程序，还是自己从源代码编译，都很简单。按照 这儿 的说明下载并安装即可。

使用交互式会话（也记为 repl），是学习 Julia 最简单的方法：

```
$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.5.0-dev+2440 (2016-02-01 02:22 UTC)
 _/ |\__'_|_|_|\__'_|  |  Commit 2bb94d6 (11 days old master)
|__/                   |  x86_64-apple-darwin13.1.0

julia> 1 + 2
3

julia> ans
3
```

输入 `^D` — `ctrl` 键加 `d` 键，或者输入 `quit()` ，可以退出交互式会话。交互式模式下， `julia` 会显示一个横幅，并提示用户来输入。一旦用户输入了完整的表达式，例如 `1 + 2` ，然后按回车，交互式会话就对表达式求值并返回这个值。如果输入的表达式末尾有分号，就不会显示它的值了。变量 `ans` 的值就是上一次计算的表达式的值，无论上一次是否被显示。变量 `ans` 仅适用于交互式会话，不适用于以其它方式运行的 Julia 代码。

如果想运行写在源文件 `file.jl` 中的代码，可以输入命令 `include("file.jl")` 。

要在非交互式模式下运行代码，你可以把它当做 Julia 命令行的第一个参数：

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

如这个例子所示，julia 后面跟着的命令行参数，被认为是程序 `script.jl` 的命令行参数。这些参数使用全局变量 `ARGS` 来传递。使用 `-e` 选项，也可以在命令行设置 `ARGS` 参数。可如下操作，来打印传递的参数：

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

也可以把代码放在一个脚本中，然后运行：

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

定界符“–“ 可以用来将脚本文件和命令行的变量分割开来：

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

Julia 可以用 `-p` 或 `--machinefile` 选项来开启并行模式。 `-p n` 会发起额外的 `n` 个工作进程，而 `--machinefile file` 会为文件 `file` 的每一行发起一个工作进程。 `file` 定义的机器，必须要能经由无密码的 `ssh` 访问，且每个机器上的 Julia 安装的位置应完全相同，每个机器的定义为 `[user@]host[:port] [bind_addr]` 。 `user` defaults to current user, `port` to the standard ssh port. Optionally, in case of multi-homed hosts, `bind_addr` may be used to explicitly specify an interface.

如果你想让 Julia 在启动时运行一些代码，可以将代码放入 `~/.juliarc.jl` ：

```
$ echo 'println("Greetings! 好! 안녕하세요?")' > ~/.juliarc.jl
$ julia
Greetings! 好! 안녕하세요?

...
```

运行 Julia 有各种可选项：

```
julia [options] [program] [args...]
 -v, --version             Display version information
 -h, --help                Print this message
 -q, --quiet               Quiet startup without banner
 -H, --home <dir>          Set location of julia executable

 -e, --eval <expr>         Evaluate <expr>
 -E, --print <expr>        Evaluate and show <expr>
 -P, --post-boot <expr>    Evaluate <expr> right after boot
 -L, --load <file>         Load <file> right after boot on all processors
 -J, --sysimage <file>     Start up with the given system image file

 -p <n>                    Run n local processes
 --machinefile <file>      Run processes on hosts listed in <file>

 -i                        Force isinteractive() to be true
 --no-history-file         Don't load or save history
 -f, --no-startup          Don't load ~/.juliarc.jl
 -F                        Load ~/.juliarc.jl, then handle remaining inputs
 --color={yes|no}          Enable or disable color text

 --code-coverage           Count executions of source lines
 --check-bounds={yes|no}   Emit bounds checks always or never (ignoring declarations)
 --int-literals={32|64}    Select integer literal size independent of platform
```

资源

除了本手册，还有一些其它的资源:

- Julia 和 IJulia 使用说明
- 速学 Julia
- MIT 讲师 Homer Reid 数值分析课的教程
- 介绍 julia 的演讲
- 来自 MIT 的 Julia 视频教程
- Forio 的 Julia 教程

CHAPTER 3

变量

Julia 中，变量即是关联到某个值的名字。当你想存储一个值（比如数学计算中的某个中间变量）以备后用时，变量的作用就体现出来了。举个例子：

```
# 将 整数 10 赋值给变量 x
julia> x = 10
10

# 对 x 所存储的值做数值运算
julia> x + 1
11

# 重新定义 x 的值
julia> x = 1 + 1
2

# 你也可以给它赋予其它类型的值，比如字符串
julia> x = "Hello World!"
"Hello World!"
```

Julia 提供了极其灵活的变量命名系统。变量名区分大小写。

```
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，在尊严和权力上一律平等。"
"人人生而自由，在尊严和权力上一律平等。"
```

也可以使用 Unicode 字符（UTF-8 编码）来命名：

```
julia> δ = 0.00001
1.0e-5

julia> 안녕하세요 = "Hello"
"Hello"
```

在Julia REPL 和其一些Julia 的境中,支持Unicode 符的入 。只需要入的LaTeX 句,再按tab 可完成入 。比如,量名δ 可以通`\delta`-*tab* 入,又如α_2 ``可以由``\alpha`-*tab*-`\hat`-*tab*-`_2`-*tab* 完成 。Julia 甚至允许重新定义内置的常数和函数：

```
julia> pi
π = 3.1415926535897...

julia> pi = 3
Warning: imported binding for pi overwritten in module Main
3

julia> pi
3

julia> sqrt(100)
10.0

julia> sqrt = 4
    Warning: imported binding for sqrt overwritten in module Main
4
```

很显然, 不鼓励这样的做法。

可用的变量名

变量名必须的开头必须是如下字符:

- 字母
- **比 00A0大的unicode 子集 具体是指, Unicode character categories:**
  - Lu/Ll/Lt/Lm/Lo/Nl(字母))开头
  - Sc/So(货币和其它符号)
  - 以及其它一些类似于字母的符号(比如 Sm 数学符号)

在变量名中的字符还可以包含！和数字, 同时也可以是 Unicode 编码点: 变音符号 以及 其它 修饰符号, 一些标点连接符, 元素, 以及一些其它的字符.

类似于 + 的运算符也是允许的标识符, 但会以其它方式解析. 在上下文中, 运算符会被类似于变量一样使用; 比如 `(+)` 代表了加法函数, 而 `(+) = f` 会重新给它赋值. 大部分的 Unicode 运算符,比如 $\oplus$, 会被当做运算符解析, 并且可以由用户来定义. 比如, 您可以使用 `const ⊗ = kron` 来定义 $\otimes$ 为一个直乘运算符.

内置的关键字不能当变量名：

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

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

命名规范

尽管 Julia 对命名本身只有很少的限制, 但尽量遵循一定的命名规范吧:

- 变量名使用小写字母
- 单词间使用下划线 (`'_'`) 分隔，但不鼓励
- 类型名首字母大写, 单词间使用驼峰式分隔.
- 函数名和宏名使用小写字母, 不使用下划线分隔单词.
- 修改参数的函数结尾使用！. 这样的函数被称为 mutating functions 或 in-place functions

CHAPTER 4

整数和浮点数

整数和浮点数是算术和计算的基础。它们都是数字文本。例如 `1` 是整数文本， `1.0` 是浮点数文本。

Julia 提供了丰富的基础数值类型，全部的算数运算符和位运算符，以及标准数学函数。这些数据和操作直接对应于现代计算机支持的操作。因此, Julia 能充分利用硬件的计算资源。另外, Julia 还从软件层面支持 任意精度的算术 ，可以用于表示硬件不能原生支持的数值，当然，这牺牲了部分运算效率。

Julia 提供的基础数值类型有：

- **整数类型：**

`Char` 原生支持 Unicode 字符 ； 详见 字符串 。

- **浮点数类型：**

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

另外, 对 复数和分数 的支持建立在这些基础数据类型之上。所有的基础数据类型通过灵活用户可扩展的 类型提升系统 ，不需显式类型转换，就可以互相运算。

整数

使用标准方式来表示文本化的整数：

```
julia> 1
1

julia> 1234
1234
```

整数文本的默认类型，取决于目标系统是 32 位架构还是 64 位架构：

```
# 32-bit system:
julia> typeof(1)
Int32

# 64-bit system:
julia> typeof(1)
Int64
```

Julia 内部变量 `WORD_SIZE` 用以指示目标系统是 32 位还是 64 位.

```
# 32-bit system:
julia> WORD_SIZE
32

# 64-bit system:
julia> WORD_SIZE
64
```

另外，Julia定义了 `Int` 和 `Uint` 类型，它们分别是系统原生的有符号和无符号 整数类型的别名:

```
# 32-bit system:
julia> Int
Int32
julia> Uint
Uint32


# 64-bit system:
julia> Int
Int64
julia> Uint
Uint64
```

对于不能用 32 位而只能用 64 位来表示的大整数文本，不管系统类型是什么，始终被认为是 64 位整数:

```
# 32-bit or 64-bit system:
julia> typeof(3000000000)
Int64
```

无符号整数的输入和输出使用前缀 `0x` 和十六进制数字 `0-9a-f` （也可以使用 `A-F` ）。无符号数的位数大小，由十六进制数的位数决定:

```
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
```

二进制和八进制文本:

```
julia> 0b10
0x02

julia> typeof(ans)
Uint8

julia> 0o10
0x08

julia> typeof(ans)
Uint8
```

基础数值类型的最小值和最大值，可由 typemin 和 typemax 函数查询:

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

julia> for T = {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 的返回值，与所给的参数类型是同一类的。（上述例子用到了一些将要介绍到的特性，包括 *for* 循环，字符串，及 内插 。）

溢出

在 Julia 中，如果计算结果超出数据类型所能代表的最大值，将会发生溢出:

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

julia> x + 1
-9223372036854775808
```

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

可见, Julia 中的算数运算其实是一种 同余算术 。它反映了现代计算机底层整数算术运算特性。如果有可能发生溢出，一定要显式的检查是否溢出；或者使用 BigInt 类型（详见 任意精度的算术 ）。

为了减小溢出所带来的影响，整数加减法、乘法、指数运算都会把原先范围较小的整数类型提升到 Int 或 Uint 类型。（除法、求余、位运算则不提升类型）。

除法错误

整数除法（“div“函数）有两种可能的情况：

- 除以 0
- 将最小的负数 （typemin()） 除以 -1

这两种情况都会报 DivideError ，此外，取余函数和求模函数（rem 和 mod）也会在第二个参数为 0 时报错。

浮点数

使用标准格式来表示文本化的浮点数：

```
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
```

上述结果均为 Float64 值。文本化的 Float32 值也可以直接输入，这时使用 f 来替代 e：

```
julia> 0.5f0
0.5f0

julia> typeof(ans)
Float32

julia> 2.5f-4
0.00025f0
```

浮点数也可以很容易地转换为 Float32：

```
julia> float32(-1.5)
-1.5f0

julia> typeof(ans)
Float32
```

十六进制浮点数的类型，只能为 Float64：

```
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)，但只用来存储。计算时，它们被转换为 Float32：

```
julia> sizeof(float16(4.))
2

julia> 2*float16(4.)
8.0f0
```

浮点数类型的零

浮点数类型中存在 两个零，正数的 零和负数的零。它们相等，但有着不同的二进制表示，可以使用 bits 函数看出：

```
julia> 0.0 == -0.0
true

julia> bits(0.0)
"0000000000000000000000000000000000000000000000000000000000000000"

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

特殊的浮点数

有三个特殊的标准浮点数：

特殊值			名称	描述
Float16	Float32	Float64		
Inf16	Inf32	Inf	正无穷	比所有的有限的浮点数都大
-Inf16	-Inf32	-Inf	负无穷	比所有的有限的浮点数都小
NaN16	NaN32	NaN	不存在	不能和任意浮点数比较大小（包括它自己）

详见 数值比较 。按照 IEEE 754 标准 ， 这几个值可如下获得:

```
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 函数也适用于浮点数类型:

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

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

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

精度

大多数的实数并不能用浮点数精确表示，因此有必要知道两个相邻浮点数间的间距，也即 计算机的精度 。 Julia 提供了 eps 函数，可以用来检查 1.0 和下一个可表示的浮点数之间的间距:

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

julia> eps(Float64)
2.220446049250313e-16

julia> eps() # same as eps(Float64)
2.220446049250313e-16
```

eps 函数也可以取浮点数作为参数，给出这个值和下一个可表示的浮点数的绝对差，即， eps(x) 的结果与 x 同类型，且满足 x + eps(x) 是下一个比 x 稍大的、可表示的浮点数：

```
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
```

相邻的两个浮点数之间的距离并不是固定的，数值越小，间距越小；数值越大, 间距越大。换句话说，浮点数在 0 附近最稠密，随着数值越来越大，数值越来越稀疏，数值间的距离呈指数增长。根据定义，eps(1.0) 与 eps(Float64) 相同，因为 1.0 是 64 位浮点数。

函数 nextfloat 和 prevfloat 可以用来获取下一个或上一个浮点数：

```
julia> x = 1.25f0
1.25f0

julia> nextfloat(x)
1.2500001f0

julia> prevfloat(x)
1.2499999f0

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

julia> bits(x)
"00111111101000000000000000000000"

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

此例显示了邻接的浮点数和它们的二进制整数的表示。

舍入模型

如果一个数没有精确的浮点数表示，那就需要舍入了。可以根据 IEEE 754 标准 来更改舍入的模型：

```
julia> 1.1 + 0.1
1.2000000000000002
```

```
julia> with_rounding(Float64,RoundDown) do
       1.1 + 0.1
       end
1.2
```

默认舍入模型为 `RoundNearest`，它舍入到最近的可表示的值，这个被舍入的值使用尽量少的有效数字。

背景和参考资料

浮点数的算术运算同人们的预期存在着许多差异，特别是对不了解底层实现的人。许多科学计算的书籍都会详细的解释这些差异。下面是一些参考资料：

- 关于浮点数算数运算最权威的指南是 IEEE 754-2008 标准；然而，该指南没有免费的网络版
- 一个简短但是清晰地解释了浮点数是怎么表示的, 请参考 John D. Cook 的 文章 。它还 简述 了由于浮点数的表示方法不同于理想的实数会带来怎样的问题
- 推荐 Bruce Dawson 的 关于浮点数的博客
- David Goldberg 的 每个计算机科学家都需要了解的浮点数算术计算，是一篇非常精彩的文章， 深入讨论了浮点数和浮点数的精度问题
- 更深入的文档, 请参考 “浮点数之父” William Kahan 的 collected writings，其中详细记录了浮点数的历史、理论依据、问题，还有其它很多的数值计算方面的内容。更有兴趣的可以读 采访浮点数之父

任意精度的算术

为保证整数和浮点数计算的精度，Julia 打包了 GNU Multiple Precision Arithmetic Library, GMP 和 GNU MPFR Library 。Julia 相应提供了 `BigInt` 和 `BigFloat` 类型。

可以通过基础数值类型或 `String` 类型来构造：

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

julia> BigInt("123456789012345678901234567890") + 1
123456789012345678901234567891

julia> BigFloat("1.23456789012345678901")
1.2345678901234567890100000000000000000000000000000000000000000000000000000000000000000000000000000004e+00␣
↪with 256 bits of precision

julia> BigFloat(2.0^66) / 3
2.4595658764946068821333333333333333333333333333333333333333333333333333333333333333333333333333344e+19␣
↪with 256 bits of precision

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

然而，基础数据类型和 *BigInt*/*BigFloat* 不能自动进行类型转换，需要明确指定：

```
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 (constructor with 10 methods)
```

BigFloat 运算的默认精度（有效数字的位数）和舍入模型，是可以改的。然后，计算就都按照更改之后的设置来运行了：

```
julia> with_rounding(BigFloat,RoundUp) do
       BigFloat(1) + BigFloat("0.1")
       end
1.100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000003e+00␣
↪with 256 bits of precision

julia> with_rounding(BigFloat,RoundDown) do
       BigFloat(1) + BigFloat("0.1")
       end
1.099999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999986e+00␣
↪with 256 bits of precision

julia> with_bigfloat_precision(40) do
       BigFloat(1) + BigFloat("0.1")
       end
1.1000000000004e+00 with 40 bits of precision
```

代数系数

Julia 允许在变量前紧跟着数值文本，来表示乘法。这有助于写多项式表达式：

```
julia> x = 3
3

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

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

指数函数也更好看：

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

数值文本系数同单目运算符一样。因此 `2^3x` 被解析为 `2^(3x)`， `2x^3` 被解析为 `2*(x^3)` 。

数值文本也可以作为括号表达式的因子：

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

括号表达式可作为变量的因子:

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

不要接着写两个变量括号表达式，也不要把变量放在括号表达式之前。它们不能被用来指代乘法运算:

```
julia> (x-1)(x+1)
ERROR: type: apply: expected Function, got Int64

julia> x(x+1)
ERROR: type: apply: expected Function, got Int64
```

这两个表达式都被解析为函数调用：任何非数值文本的表达式，如果后面跟着括号，代表调用函数来处理括号内的数值（详见 函数 ）。因此，由于左面的值不是函数，这两个例子都出错了。

需要注意，代数因子和变量或括号表达式之间不能有空格。

语法冲突

文本因子与两个数值表达式语法冲突: 十六进制整数文本和浮点数文本的科学计数法:

- 十六进制整数文本表达式 `0xff` 可以被解析为数值文本 `0` 乘以变量 `xff`
- 浮点数文本表达式 `1e10` 可以被解析为数值文本 `1` 乘以变量 `e10` 。 `E` 格式也同样。

这两种情况下，我们都把表达式解析为数值文本:

- 以 `0x` 开头的表达式，都被解析为十六进制文本
- 以数字文本开头，后面跟着 `e` 或 `E` ，都被解析为浮点数文本

零和一

Julia 提供了一些函数, 用以得到特定数据类型的零和一文本。

函数	说明
`zero(x)`	类型 `x` 或变量 `x` 的类型下的文本零
`one(x)`	类型 `x` 或变量 `x` 的类型下的文本一

这俩函数在 数值比较 中可用来避免额外的 类型转换 。

例如:

```
julia> zero(Float32)
0.0f0

julia> zero(1.0)
0.0

julia> one(Int32)
1
```

```
julia> one(BigFloat)
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

CHAPTER 5

数学运算和基本函数

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。

算术运算符

下面的 算术运算符 适用于所有的基本数值类型：

表达式	名称	描述
`+x`	一元加法	`x` 本身
`-x`	一元减法	相反数
`x + y`	二元加法	做加法
`x - y`	二元减法	做减法
`x * y`	乘法	做乘法
`x / y`	除法	做除法
`x \ y`	反除	等价于 `y / x`
`x ^ y`	乘方	`x` 的 `y` 次幂
`x % y`	取余	等价于 `rem(x, y)`

以及 `Bool` 类型的非运算：

表达式	名称	描述
`!x`	非	`true` 和 `false` 互换

Julia 的类型提升系统使得参数类型混杂的算术运算也很简单自然。详见 类型转换和类型提升 。

算术运算的例子：

```
julia> 1 + 2 + 3
6

julia> 1 - 2
-1
```

```
julia> 3*2/12
0.5
```

（习惯上，优先级低的运算，前后多补些空格。这不是强制的。）

位运算符

下面的 位运算符 适用于所有整数类型：

Expression	Name
`~x`	按位取反
`x & y`	按位与
`x | y`	按位或
`x $ y`	按位异或
`x >>> y`	向右 逻辑移位 （高位补 0）
`x >> y`	向右 算术移位 （复制原高位）
`x << y`	向左逻辑/算术移位

位运算的例子：

```
julia> ~123
-124

julia> 123 & 234
106

julia> 123 | 234
251

julia> 123 $ 234
145

julia> ~uint32(123)
0xffffff84

julia> ~uint8(123)
0x84
```

复合赋值运算符

二元算术和位运算都有对应的复合赋值运算符，即运算的结果将会被赋值给左操作数。在操作符的后面直接加上 = 就组成了复合赋值运算符。例如, `x += 3` 相当于 `x = x + 3`：

```
julia> x = 1
1

julia> x += 3
4

julia> x
4
```

复合赋值运算符有：

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

数值比较

所有的基础数值类型都可以使用比较运算符：

运算符	名称
==	等于
!=	不等于
<	小于
<=	小于等于
>	大于
>=	大于等于

一些例子：

```
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
```

整数是按位比较的。浮点数是按 IEEE 754 标准 比较的：

- 有限数按照正常方式做比较.
- 正数的零等于但不大于负数的零.

- `Inf` 等于它本身，并且大于所有数, 除了 `NaN`.
- `-Inf` 等于它本身，并且小于所有数, 除了 `NaN`.
- `NaN` 不等于、不大于、不小于任何数，包括它本身.

上面最后一条是关于 `NaN` 的性质，值得留意：

```
julia> NaN == NaN
false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false
```

`NaN` 在 矩阵 中使用时会带来些麻烦：

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

Julia 提供了附加函数, 用以测试这些特殊值，它们使用哈希值来比较：

函数	测试
`isequal(x, y)`	`x` 是否等价于 `y`
`isfinite(x)`	`x` 是否为有限的数
`isinf(x)`	`x` 是否为无限的数
`isnan(x)`	`x` 是否不是数

`isequal` 函数，认为 `NaN` 等于它本身：

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

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

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

`isequal` 也可以用来区分有符号的零：

```
julia> -0.0 == 0.0
true

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

链式比较

与大多数语言不同，Julia 支持 Python链式比较 ：

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

对标量的比较，链式比较使用 `&&` 运算符；对逐元素的比较使用 `&` 运算符，此运算符也可用于数组。例如，`0 .< A .< 1` 的结果是一个对应的布尔数组，满足条件的元素返回 true 。

操作符 `.<` 是特别针对数组的; 只有当 `A` 和 `B` 有着相同的大小时, `A .< B` 才是合法的. 比较的结果是布尔型数组, 其大小同 `A` 和 `B` 相同. 这样的操作符被称为 按元素 操作符; Julia 提供了一整套的按元素操 作符: `.*`, `.+`, 等等. 有的按元素操作符也可以接受纯量, 例如上一段 的 `0 .< A .< B`. 这种表示法的意思是, 相应的纯量操作符会被施加到每一 个元素上去.

注意链式比较的比较顺序:

```
v(x) = (println(x); x)

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

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

中间的值只计算了一次，而不是像 `v(1) < v(2) && v(2) <= v(3)` 一样计算了两次。但是，链式比较的计算顺序是不确定的。不要在链式比较中使用带副作用（比如打印）的表达式。如果需要使用副作用表达式，推荐使用短路 `&&` 运算符（详见 短路求值 ）。

运算优先级

Julia 运算优先级从高至低依次为:

类型	运算符
语法	`.` followed by `::`
幂	`^` and its elementwise equivalent `.^`
分数	`//` and `.//`
乘除	`* / % & \` and `.* ./ .% .\`
位移	`<< >> >>>` and `.<< .>> .>>>`
加减	`+ - | $` and `.+ .-`
语法	`:` `..` followed by `|>`
比较	`> < >= <= == === != !== <:` and `.> .< .>= .<= .== .!=`
逻辑	`&&` followed by `||` followed by `?`
赋值	`= += -= *= /= //= \= ^= %= |= &= $= <<= >>= >>>=` 及 `.+= .-= .*= ./= .//= .\= .^= .%=`

基本函数

Julia 提供了一系列数学函数和运算符:

舍入函数

函数	描述	返回类型
`round(x)`	把 x 舍入到最近的整数	`FloatingPoint`
`iround(x)`	把 x 舍入到最近的整数	`Integer`
`floor(x)`	把 x 向 `-Inf` 取整	`FloatingPoint`
`ifloor(x)`	把 x 向 `-Inf` 取整	`Integer`
`ceil(x)`	把 x 向 `+Inf` 取整	`FloatingPoint`
`iceil(x)`	把 x 向 `+Inf` 取整	`Integer`
`trunc(x)`	把 x 向 0 取整	`FloatingPoint`
`itrunc(x)`	把 x 向 0 取整	`Integer`

除法函数

函数	描述
`div(x,y)`	截断取整除法；商向 0 舍入
`fld(x,y)`	向下取整除法；商向 `-Inf` 舍入
`cld(x,y)`	向上取整除法; 商向 `+Inf` 舍入
`rem(x,y)`	除法余数；满足 `x == div(x,y)*y + rem(x,y)` ，与 x 同号
`divrem(x,y)`	返回 `(div(x,y),rem(x,y))`
`mod(x,y)`	取模余数；满足 `x == fld(x,y)*y + mod(x,y)` ，与 y 同号
`mod2pi(x)`	对 2pi 取模余数； `0 <= mod2pi(x) < 2pi`
`gcd(x,y...)`	x, y, ... 的最大公约数，与 x 同号
`lcm(x,y...)`	x, y, ... 的最小公倍数，与 x 同号

符号函数和绝对值函数

函数	描述
`abs(x)`	x 的幅值
`abs2(x)`	x 的幅值的平方
`sign(x)`	x 的正负号，返回值为 -1, 0, 或 +1
`signbit(x)`	是否有符号位，有 (true) 或者 无 (false)
`copysign(x,y)`	返回一个数，它具有 x 的幅值， y 的符号位
`flipsign(x,y)`	返回一个数，它具有 x 的幅值， `x*y` 的符号位

乘方，对数和开方

函数	描述
sqrt(x) $\sqrt{x}$	x 的平方根
cbrt(x) $\sqrt[3]{x}$	x 的立方根
hypot(x,y)	误差较小的 sqrt(x^2 + y^2)
exp(x)	自然指数 e 的 x 次幂
expm1(x)	当 x 接近 0 时，精确计算 exp(x)-1
ldexp(x,n)	当 n 为整数时，高效计算“x*2^n“
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，精确计算 log(1+x)
exponent(x)	trunc(log2(x))
significand(x)	returns the binary significand (a.k.a. mantissa) of a floating-point number x

为什么要有 hypot, expm1, log1p 等函数，参见 John D. Cook 的博客： expm1, log1p, erfc 和 hypot 。

三角函数和双曲函数

Julia 内置了所有的标准三角函数和双曲函数

```
sin    cos    tan    cot    sec    csc
sinh   cosh   tanh   coth   sech   csch
asin   acos   atan   acot   asec   acsc
asinh  acosh  atanh  acoth  asech  acsch
sinc   cosc   atan2
```

除了 atan2 之外，都是单参数函数。 atan2 给出了 *x* 轴，与由 *x* 、 *y* 确定的点之间的 弧度 。

Additionally, sinpi(x) and cospi(x) are provided for more accurate computations of sin(pi*x) and cos(pi*x) respectively.

如果想要以度，而非弧度，为单位计算三角函数，应使用带 d 后缀的函数。例如， sind(x) 计算 x 的正弦值，这里 x 的单位是度。以下的列表是全部的以度为单位的三角函数：

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

特殊函数

函数	描述
erf(x)	x 处的 误差函数
erfc(x)	补误差函数。当 x 较大时，精确计算 1-erf(x)
erfinv(x)	erf 的反函数
erfcinv(x)	erfc 的反函数
erfi(x)	the imaginary error function defined as -im * erf(x * im), where im
erfcx(x)	the scaled complementary error function, i.e. accurate exp(x^2) * erf
dawson(x)	the scaled imaginary error function, a.k.a. Dawson function, i.e. accurate ex
gamma(x)	x 处的 gamma 函数

表 5.1 – 续上页

函数	描述
`lgamma(x)`	当 `x` 较大时，精确计算 `log(gamma(x))`
`lfact(x)`	accurate `log(factorial(x))` for large `x`; same as `lgamma(x+1)` for
`digamma(x)`	the digamma function (i.e. the derivative of `lgamma`) at `x`
`beta(x,y)`	the beta function at `x,y`
`lbeta(x,y)`	accurate `log(beta(x,y))` for large `x` or `y`
`eta(x)`	the Dirichlet eta function at `x`
`zeta(x)`	the Riemann zeta function at `x`
`airy(z),airyai(z),airy(0,z)`	the Airy Ai function at `z`
`airyprime(z),airyaiprime(z),airy(1,z)`	Airy Ai 函数在 `z` 处的导数
`airybi(z),airy(2,z)`	the Airy Bi function at `z`
`airybiprime(z),airy(3,z)`	Airy Bi 函数在 `z` 处的导数
`airyx(z),airyx(k,z)`	the scaled Airy AI function and `k` th derivatives at `z`
`besselj(nu,z)`	the Bessel function of the first kind of order `nu` at `z`
`besselj0(z)`	`besselj(0,z)`
`besselj1(z)`	`besselj(1,z)`
`besseljx(nu,z)`	the scaled Bessel function of the first kind of order `nu` at `z`
`bessely(nu,z)`	the Bessel function of the second kind of order `nu` at `z`
`bessely0(z)`	`bessely(0,z)`
`bessely1(z)`	`bessely(1,z)`
`besselyx(nu,z)`	the scaled Bessel function of the second kind of order `nu` at `z`
`besselh(nu,k,z)`	the Bessel function of the third kind (a.k.a. Hankel function) of order `nu` at
`hankelh1(nu,z)`	`besselh(nu, 1, z)`
`hankelh1x(nu,z)`	scaled `besselh(nu, 1, z)`
`hankelh2(nu,z)`	`besselh(nu, 2, z)`
`hankelh2x(nu,z)`	scaled `besselh(nu, 2, z)`
`besseli(nu,z)`	the modified Bessel function of the first kind of order `nu` at `z`
`besselix(nu,z)`	the scaled modified Bessel function of the first kind of order `nu` at `z`
`besselk(nu,z)`	the modified Bessel function of the second kind of order `nu` at `z`
`besselkx(nu,z)`	the scaled modified Bessel function of the second kind of order `nu` at `z`

CHAPTER 6

复数和分数

Julia 提供复数和分数类型，并对其支持所有的 标准数学运算 。对不同的数据类型进行混合运算时，无论是基础的还是复合的，都会自动使用 类型转换和类型提升 .

复数

全局变量 `im` 即复数 *i* ，表示 -1 的正平方根。因为 `i` 经常作为索引变量，所以不使用它来代表复数了。Julia 允许数值文本作为 代数系数 ，也适用于复数：

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

可以对复数做标准算术运算：

```
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
2.7296244647840084 - 6.960664459571898im

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
```

类型提升机制保证了不同类型的运算对象能够在一起运算：

```
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)`，因为文本系数比除法优先。

处理复数的标准函数：

```
julia> real(1 + 2im)
1

julia> imag(1 + 2im)
2

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

julia> abs(1 + 2im)
2.23606797749979

julia> abs2(1 + 2im)
5
```

```
julia> angle(1 + 2im)
1.1071487177940904
```

通常， 复数的绝对值(abs)是它到零的距离。 函数 abs2 返回绝对值的平方， 特别地用在复数上来避免开根。angle 函数返回弧度制的相位(即 *argument* 或 *arg*)。 所有的 基本函数 也可以应用在复数上：

```
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
```

作用在实数上的数学函数，返回值一般为实数；作用在复数上的，返回值为复数。例如， sqrt 对 -1 和 -1 + 0im 的结果不同，即使 -1 == -1 + 0im：

```
julia> sqrt(-1)
ERROR: DomainError
sqrt will only return a complex result if called with a complex argument.
try sqrt(complex(x))
 in sqrt at math.jl:131

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

代数系数 不能用于使用变量构造复数。乘法必须显式的写出来：

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

但是， 不 推荐使用上面的方法。推荐使用 complex 函数构造复数：

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

这种构造方式避免了乘法和加法操作。

Inf 和 NaN 也可以参与构造复数 (参考 特殊的浮点数 部分)：

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

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

分数

Julia 有分数类型。使用 // 运算符构造分数：

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

如果分子、分母有公约数，将自动约简至最简分数，且分母为非负数：

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

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

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

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

约简后的分数都是唯一的，可以通过分别比较分子、分母来确定两个分数是否相等。使用 num 和 den 函数来取得约简后的分子和分母：

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

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

其实并不需要比较分数和分母，我们已经为分数定义了标准算术和比较运算：

```
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
```

分数可以简单地转换为浮点数：

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

分数到浮点数的转换遵循，对任意整数 a 和 b，除 a == 0 及 b == 0 之外，有：

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

可以构造结果为 Inf 的分数：

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

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

julia> typeof(ans)
Rational{Int64} (constructor with 1 method)
```

但不能构造结果为 NaN 的分数：

```
julia> 0//0
ERROR: invalid rational: 0//0
 in Rational at rational.jl:6
 in // at rational.jl:15
```

类型提升系统使得分数类型与其它数值类型交互非常简单：

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

julia> 3//5 - 0.5
0.09999999999999998

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

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

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

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

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

julia> 0.5 == 1//2
true

julia> 0.33 == 1//3
false

julia> 0.33 < 1//3
true
```

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

CHAPTER 7

字符串

Julia 中处理 ASCII 文本简洁高效，也可以处理 Unicode 。使用 C 风格的字符串代码来处理 ASCII 字符串，性能和语义都没问题。如果这种代码遇到非 ASCII 文本，会提示错误，而不是显示乱码。这时，修改代码以兼容非 ASCII 数据也很简单。

关于 Julia 字符串，有一些值得注意的高级特性：

- `AbstractString` 是个抽象类型，不是具体类型——很多不同的表述都可以实现 `AbstractString` 的接口， 但他们很容易清晰地展示出相互关系并很容易的被一起使用。任何字符串类型的变量都可以传入一个在函数定义中声明了‘‘AbstractString‘‘类型的自变量。
- 和 C语言 以及Java 一样（但和大部分动态语言不同）， Julia 的 `Char` 类型代表单字符，是由 32 位整数表示的 Unicode 码位
- 与 Java 中一样，字符串不可更改： `String` 对象的值不能改变。要得到不同的字符串，需要构造新的字符串
- 概念上，字符串是从索引值映射到字符的 部分函数 ，对某些索引值，如果不是字符，会抛出异常
- Julia 支持全部 Unicode 字符: 文本字符通常都是 ASCII 或 UTF-8 的，但也支持其它编码

字符

`Char` 表示单个字符：它是 32 位整数，值参见 Unicode 码位 。 `Char` 必须使用单引号：

```
julia> 'x'
'x'

julia> typeof(ans)
Char
```

可以把 `Char` 转换为对应整数值：

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

```
julia> typeof(ans)
Int64
```

在 32 位架构上， `typeof(ans)` 的类型为 `Int32` 。也可以把整数值转换为 `Char` ：

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

并非所有的整数值都是有效的 Unicode 码位，但为了性能， `Char` 一般不检查其是否有效。如果你想要确保其有效，使用 `isvalid` 函数：

```
julia> Char(0x110000)
'\U110000'

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

目前，有效的 Unicode 码位为，从 `U+00` 至 `U+d7ff` ，以及从 `U+e000` 至 `U+10ffff` 。

可以用单引号包住 `\u` 及跟着的最多四位十六进制数，或者 `\U` 及跟着的最多八位（有效的字符，最多需要六位）十六进制数，来输入 Unicode 字符：

```
julia> '\u0'
'\0'

julia> '\u78'
'x'

julia> '\u2200'
'∀'

julia> '\U10ffff'
'\U10ffff'
```

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，所有 C 语言转义的输入格式 都能使：

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

julia> Int('\t')
9

julia> Int('\n')
10

julia> Int('\e')
27

julia> Int('\x7f')
127

julia> Int('\177')
127

julia> Int('\xff')
255
```

可以对 Char 值比较大小，也可以做少量算术运算：

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

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

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

julia> 'x' - 'a'
23

julia> 'A' + 1
'B'
```

字符串基础

字符串文本应放在双引号 "..." 或三个双引号 """...""" 中间：

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

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

使用索引从字符串提取字符：

```
julia> str[1]
'H'

julia> str[6]
','

julia> str[end]
'\n'
```

Julia 中的索引都是从 1 开始的，最后一个元素的索引与字符串长度相同，都是 n 。

在任何索引表达式中，关键词 end 都是最后一个索引值（由 endof(str) 计算得到）的缩写。可以对字符串做 end 算术或其它运算：

```
julia> str[end-1]
'.'

julia> str[end/2]
' '

julia> str[end/3]
ERROR: InexactError()
 in getindex at string.jl:59

julia> str[end/4]
ERROR: InexactError()
 in getindex at string.jl:59
```

索引小于 1 或者大于 end，会提示错误:

```
julia> str[0]
ERROR: BoundsError()

julia> str[end+1]
ERROR: BoundsError()
```

使用范围索引来提取子字符串:

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

str[k] 和 str[k:k] 的结果不同:

```
julia> str[6]
','

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

前者是类型为 Char 的单个字符，后者为仅有一个字符的字符串。在 Julia 中这两者完全不同。

Unicode 和 UTF-8

Julia 完整支持 Unicode 字符和字符串。正如 上文所讨论的，在字符文本中， Unicode 码位可以由 \u 和 \U 来转义，也可以使用标准 C 的转义序列。它们都可以用来写字符串文本:

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

非 ASCII 字符串文本使用 UTF-8 编码。 UTF-8 是一种变长编码，意味着并非所有的字符的编码长度都是相同的。在 UTF-8 中，码位低于 0x80 (128) 的字符即 ASCII 字符，编码如在 ASCII 中一样，使用单字节；其余码位的字符使用多字节，每字符最多四字节。这意味着 UTF-8 字符串中，并非所有的字节索引值都是有效的字符索引值。如果索引到无效的字节索引值，会抛出错误:

```
julia> s[1]
'∀'

julia> s[2]
ERROR: invalid UTF-8 character index
 in next at ./utf8.jl:68
 in getindex at string.jl:57

julia> s[3]
ERROR: invalid UTF-8 character index
 in next at ./utf8.jl:68
 in getindex at string.jl:57

julia> s[4]
' '
```

上例中，字符 ∀ 为 3 字节字符，所以索引值 2 和 3 是无效的，而下一个字符的索引值为 4。

由于变长编码，字符串的字符数（由 length(s) 确定）不一定等于字符串的最后索引值。对字符串 s 进行索引，并从 1 遍历至 endof(s)，如果没有抛出异常，返回的字符序列将包括 s 的序列。因而 length(s)

<= endof(s) 。下面是个低效率的遍历 s 字符的例子:

```
julia> for i = 1:endof(s)
         try
           println(s[i])
         catch
           # ignore the index error
         end
       end
∀

x

∃

y
```

所幸我们可以把字符串作为遍历对象，而不需处理异常:

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

x

∃

y
```

Julia 不只支持 UTF-8 ，增加其它编码的支持也很简单。In particular, Julia also provides `UTF16String` and `UTF32String` types, constructed by the `utf16(s)` and `utf32(s)` functions respectively, for UTF-16 and UTF-32 encodings. It also provides aliases `WString` and `wstring(s)` for either UTF-16 or UTF-32 strings, depending on the size of `Cwchar_t`. 有关 UTF-8 的讨论，详见下面的 字节数组文本 。

内插

字符串连接是最常用的操作:

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

julia> whom = "world"
"world"

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

像 Perl 一样， Julia 允许使用 $ 来内插字符串文本:

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

系统会将其重写为字符串文本连接。

$ 将其后的最短的完整表达式内插进字符串。可以使用小括号将任意表达式内插:

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

字符串连接和内插都调用 `string` 函数来把对象转换为 `String` 。与在交互式会话中一样，大多数非 `String` 对象被转换为字符串：

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

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

`Char` 值也可以被内插到字符串中：

```
julia> c = 'x'
'x'

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

要在字符串文本中包含 `$` 文本，应使用反斜杠将其转义：

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

Triple-Quoted String Literals

When strings are created using triple-quotes (`"""...."""`) they have some special behavior that can be useful for creating longer blocks of text. First, if the opening `"""` is followed by a newline, the newline is stripped from the resulting string.

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

is equivalent to

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

but

```
"""

hello"""
```

will contain a literal newline at the beginning. Trailing whitespace is left unaltered. They can contain `"` symbols without escaping. Triple-quoted strings are also dedented to the level of the least-indented line. This is useful for defining strings within code that is indented. For example:

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

```
        """
"  Hello,\n  world.\n"
```

In this case the final (empty) line before the closing """ sets the indentation level.

Note that line breaks in literal strings, whether single- or triple-quoted, result in a newline (LF) character \n in the string, even if your editor uses a carriage return \r (CR) or CRLF combination to end lines. To include a CR in a string, use an explicit escape \r; for example, you can enter the literal string "a CRLF line ending\r\n".

一般操作

使用标准比较运算符，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
```

使用 search 函数查找某个字符的索引值：

```
julia> search("xylophone", 'x')
1

julia> search("xylophone", 'p')
5

julia> search("xylophone", 'z')
0
```

可以通过提供第三个参数，从此偏移值开始查找：

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

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

julia> search("xylophone", 'o', 8)
0
```

另一个好用的处理字符串的函数 repeat：

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

其它一些有用的函数：

- endof(str) 给出 str 的最大（字节）索引值
- length(str) 给出 str 的字符数

- `i = start(str)` 给出第一个可在 `str` 中被找到的字符的有效索引值（一般为 1）
- `c, j = next(str,i)` 返回索引值 `i` 处或之后的下一个字符，以及之后的下一个有效字符的索引值。通过 `start` 和 `endof`，可以用来遍历 `str` 中的字符
- `ind2chr(str,i)` 给出字符串中第 i 个索引值所在的字符，对应的是第几个字符
- `chr2ind(str,j)` 给出字符串中索引为 i 的字符，对应的（第一个）字节的索引值

非标准字符串文本

Julia 提供了 非标准字符串文本 。它在正常的双引号括起来的字符串文本上，添加了前缀标识符。下面将要介绍的正则表达式、字节数组文本和版本号文本，就是非标准字符串文本的例子。 元编程 章节有另外的一些例子。

正则表达式

Julia 的正则表达式 (regexp) 与 Perl 兼容，由 PCRE 库提供。它是一种非标准字符串文本，前缀为 `r`，最后面可再跟一些标识符。最基础的正则表达式仅为 `r"..."` 的形式：

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

julia> typeof(ans)
Regex (constructor with 3 methods)
```

检查正则表达式是否匹配字符串，使用 `ismatch` 函数：

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

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

`ismatch` 根据正则表达式是否匹配字符串，返回真或假。 `match` 函数可以返回匹配的具体情况：

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

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

如果没有匹配， `match` 返回 `nothing`，这个值不会在交互式会话中打印。除了不被打印，这个值完全可以在编程中正常使用：

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

如果匹配成功， `match` 的返回值是一个 `RegexMatch` 对象。这个对象记录正则表达式是如何匹配的，包括类型匹配的子字符串，和其他捕获的子字符串。本例中仅捕获了匹配字符串的一部分，假如我们想要注释字符后的非空白开头的文本，可以这么写：

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

When calling `match`, you have the option to specify an index at which to start the search. For example:

```
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")
```

可以在 `RegexMatch` 对象中提取下列信息:

- 完整匹配的子字符串: `m.match`
- 捕获的子字符串组成的字符串多元组: `m.captures`
- 完整匹配的起始偏移值: `m.offset`
- 捕获的子字符串的偏移值向量: `m.offsets`

对于没匹配的捕获, `m.captures` 的内容不是子字符串, 而是 `nothing` , `m.offsets` 为 0 偏移（Julia 中的索引值都是从 1 开始的, 因此 0 偏移值表示无效）:

```
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(SubString{UTF8String},Nothing),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(SubString{UTF8String},Nothing),1}:
 "a"
 nothing
 "d"
```

```
julia> m.offset
1

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

可以把结果多元组绑定给本地变量：

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

可以在右引号之后，使用标识符 i, m, s, 及 x 的组合，来修改正则表达式的行为。这几个标识符的用法与 Perl 中的一样，详见 perlre manpage：

```
i    不区分大小写

m    多行匹配。 "^" 和 "$" 匹配多行的起始和结尾

s    单行匹配。 "." 匹配所有字符，包括换行符

     一起使用时，例如 r""ms 中， "." 匹配任意字符，而 "^" 与 "$" 匹配字符串中新行之前和之后的字符

x    忽略大多数空白，除非是反斜杠。可以使用这个标识符，把正则表达式分为可读的小段。 '#' 字符被认为是引入注释的元字符
```

例如，下面的正则表达式使用了所有选项：

```
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")
```

Julia 支持三个双引号所引起来的正则表达式字符串，即 r"""..."""。这种形式在正则表达式包含引号或换行符时比较有用。

... Triple-quoted regex strings, of the form r"""..."""", are also ... supported (and may be convenient for regular expressions containing ... quotation marks or newlines).

字节数组文本

另一类非标准字符串文本为 b"..."，可以表示文本化的字节数组，如 Uint8 数组。习惯上，非标准文本的前缀为大写，会生成实际的字符串对象；而前缀为小写的，会生成非字符串对象，如字节数组或编译后的正则表达式。字节表达式的规则如下：

- ASCII 字符与 ASCII 转义符生成一个单字节
- \x 和 八进制转义序列生成对应转义值的 字节
- Unicode 转义序列生成 UTF-8 码位的字节序列

三种情况都有的例子：

```
julia> b"DATA\xff\u2200"
8-element Array{Uint8,1}:
 0x44
 0x41
 0x54
 0x41
 0xff
 0xe2
 0x88
 0x80
```

ASCII 字符串 "DATA" 对应于字节 68, 65, 84, 65 。 `\xff` 生成的单字节为 255 。Unicode 转义 `\u2200` 按 UTF-8 编码为三字节 226, 136, 128 。注意，字节数组的结果并不对应于一个有效的 UTF-8 字符串，如果把它当作普通的字符串文本，会得到语法错误：

```
julia> "DATA\xff\u2200"
ERROR: syntax: invalid UTF-8 sequence
```

`\xff` 和 `\uff` 也不同：前者是 字节 *255* 的转义序列；后者是 码位 *255* 的转义序列，将被 UTF-8 编码为两个字节：

```
julia> b"\xff"
1-element Array{Uint8,1}:
 0xff

julia> b"\uff"
2-element Array{Uint8,1}:
 0xc3
 0xbf
```

在字符文本中，这两个是相同的。 `\xff` 也可以代表码位 255，因为字符 永远 代表码位。然而在字符串中， `\x` 转义永远表示字节而不是码位，而 `\u` 和 `\U` 转义永远表示码位，编码后为 1 或多个字节。

版本号常量

版本号可以很容易的用非标准的字符串常量表达 `v"..."` 。 版本号常量会按照‘语义版本控制 <http://semver.org>‘_ 的规格，根据预览版本和 build alpha-numeric 注释，创建一个 `VersionNumber` 对象。例如，`v"0.2.1-rc1+win64"``会被拆成主版本 ``0`，次要版本 `2`，修补版本 `1`，预览版本 `rc1` 和构建版本 `win64`。 在版本号常量中，除了主版本号以外的都是可选的，比如 `v"0.2"` 等价于 `v"0.2.0"``（没有预览版和编译注释），``v"2"` 等价于 `v"2.0.0"`, 以此类推。

`VersionNumber` 对象对正确地对比两个或多个版本非常有用，常数 `VERSION` 将Julia的版本皓以 `VersionNumber` 对象的形式存储下来，于是这使得我们可以用如下简单的命令来规范版本：

```
if v"0.2" <= VERSION < v"0.3-"
    # do something specific to 0.2 release series
end
```

注意上面的例子使用了非标准的版本号 `v"0.3-"`，加上了一个后缀 `-`：这代表比 `0.3` 版本要老的旧版本，也就是说这个代码只能在稳定的 `0.2` 版本上运行，并会排除类似于“v”0.3.0-rc1”“ 这样的版本。为了使得不稳定的 `0.2` 版本也能使用，最低版本检查应该这样写：`v"0.2-" <= VERSION`。

另外一种非标准的版本规范扩展允许使用一个 `+` 作为后缀来表达更高的版本，例如 `VERSION > "v"0.2-rc1+"` 可以用来代表所有 `0.2-rc1` 以后的版本和它的编译版本：对于 `v"0.2-rc1+win64"``会返回``false``对于 ``v"0.2-rc2"` 则会返回 `true`。

一般来讲‘‘-‘‘ 总应当作为后缀出现在上界的限定中，但不能用来做为真实的版本号， 因为它在语义版本控制的标准中不存在。

此外常数‘‘VERSION‘‘ 和对象 `VersionNumber` 常常用来在 `Pkg <Base.Pkg>` 模块中指定依赖关系。

CHAPTER 8

函数

Julia 中的函数是将一系列参数组成的元组映设到一个返回值的对象，Julia 的函数不是纯的数学式函数，有些函数可以改变或者影响程序的全局状态。Julia 中定义函数的基本语法为：

```
function f(x,y)
  x + y
end
```

Julia 中可以精炼地定义函数。上述传统的声明语法，等价于下列紧凑的“赋值形式”：

```
f(x,y) = x + y
```

对于赋值形式，函数体通常是单表达式，但也可以为复合表达式（详见 复合表达式 ）。Julia 中常见这种短小简单的函数定义。短函数语法相对而言更方便输入和阅读。

使用圆括号来调用函数：

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

没有圆括号时， `f` 表达式指向的是函数对象，这个函数对象可以像值一样被传递：

```
julia> g = f;

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

调用函数有两种方法：使用特定函数名的特殊运算符语法（详见后面 函数运算符 ），或者使用 `apply` 函数：

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

`apply` 函数把第一个参数当做函数对象，应用在后面的参数上。

和变量名称一样, 函数名称也可以使用 Unicode 字符:

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

参数传递行为

Julia 函数的参数遵循 “pass-by-sharing” 的惯例，即不传递值，而是传递引用。函数参数本身，有点儿像新变量 绑定 （引用值的新位置），但它们引用的值与传递的值完全相同。对可变值（如数组）的修改，会影响其它函数。

return 关键字

函数返回值通常是函数体中最后一个表达式的值。上一节中 f 是表达式 x + y 的值。在 C 和大部分命令式语言或函数式语言中， return 关键字使得函数在计算完该表达式的值后立即返回：

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

对比下列两个函数：

```
f(x,y) = x + y

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

julia> f(2,3)
5

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

在纯线性函数体，比如 g 中，不需要使用 return ，它不会计算表达式 x + y 。可以把 x * y 作为函数的最后一个表达式，并省略 return 。只有涉及其它控制流时， return 才有用。下例计算直角三角形的斜边长度，其中直角边为 x 和 y ，为避免溢出：

```
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
```

最后一行的 return 可以省略。

函数运算符

Julia 中，大多数运算符都是支持特定语法的函数。 `&&` 、 `||` 等短路运算是例外，它们不是函数，因为 短路求值 先算前面的值，再算后面的值。 对于函数运算符，可以像其它函数一样，把参数列表用圆括号括起来，作为函数运算符的参数:

```
julia> 1 + 2 + 3
6

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

中缀形式与函数形式完全等价，事实上，前者被内部解析为函数调用的形式。可以像对其它函数一样，对 + 、 * 等运算符进行赋值、传递:

```
julia> f = +;

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

但是，这时 `f` 函数不支持中缀表达式。

特殊名字的运算符

有一些表达式调用特殊名字的运算符:

表达式	调用
`[A B C ...]`	`hcat`
`[A, B, C, ...]`	`vcat`
`[A B; C D; ...]`	`hvcat`
`A'`	`ctranspose`
`A.'`	`transpose`
`1:n`	`colon`
`A[i]`	`getindex`
`A[i]=x`	`setindex!`

这些函数都存在于 `Base.Operators` 模块中。

匿名函数

Julia 中函数是 第一类对象 ，可以被赋值给变量，可以通过赋值后的变量来调用函数, 还可以当做参数和返回值，甚至可以被匿名构造:

```
julia> x -> x^2 + 2x - 1
(anonymous function)
```

上例构造了一个匿名函数，输入一个参数 *x* ，返回多项式 *x*^2 + 2*x* - 1 的值。匿名函数的主要作用是把它传递给接受其它函数作为参数的函数。最经典的例子是 `map` 函数，它将函数应用在数组的每个值上，返回结果数组:

```
julia> map(round, [1.2,3.5,1.7])
3-element Array{Float64,1}:
 1.0
 4.0
 2.0
```

map 的第一个参数可以是非匿名函数。但是大多数情况，不存在这样的函数时，匿名函数就可以简单地构造单用途的函数对象，而不需要名字：

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

匿名函数可以通过类似 (x,y,z)->2x+y-z 的语法接收多个参数。无参匿名函数则类似于 ()->3 。无参匿名函数可以“延迟”计算，做这个用处时，代码被封装进无参函数，以后可以通过把它命名为 f() 来引入。

多返回值

Julia 中可以通过返回多元组来模拟返回多值。但是，多元组并不需要圆括号来构造和析构，因此造成了可以返回多值的假象。下例返回一对儿值：

```
julia> function foo(a,b)
         a+b, a*b
       end;
```

如果在交互式会话中调用这个函数，但不将返回值赋值出去，会看到返回的是多元组：

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

Julia 支持简单的多元组“析构”来给变量赋值：

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

julia> x
5

julia> y
6
```

也可以通过 return 来返回：

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

这与之前定义的 foo 结果相同。

变参函数

函数的参数列表如果可以为任意个数，有时会非常方便。这种函数被称为“变参”函数，是“参数个数可变”的

简称。可以在最后一个参数后紧跟省略号 ... 来定义变参函数:

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

变量 a 和 b 是前两个普通的参数，变量 x 是尾随的可迭代的参数集合，其参数个数为 0 或多个:

```
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))
```

上述例子中， x 是传递给 bar 的尾随的值多元组。

函数调用时，也可以使用 ... :

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

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

上例中，多元组的值完全按照变参函数的定义进行内插，也可以不完全遵守其函数定义来调用:

```
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))
```

被内插的对象也可以不是多元组:

```
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))
```

原函数也可以不是变参函数（大多数情况下，应该写成变参函数）：

```
baz(a,b) = a + b

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

julia> baz(args...)
3

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

julia> baz(args...)
no method baz(Int64,Int64,Int64)
```

但如果输入的参数个数不对，函数调用会失败。

可选参数

很多时候，函数参数都有默认值。例如，库函数 `parseint(num,base)` 把字符串解析为某个进制的数。`base` 参数默认为 `10` 。这种情形可以写为：

```
function parseint(num, base=10)
    ###
end
```

这时，调用函数时，参数可以是一个或两个。当第二个参数未指明时，自动传递 `10` ：

```
julia> parseint("12",10)
12

julia> parseint("12",3)
5

julia> parseint("12")
12
```

可选参数很方便参数个数不同的多方法定义（详见 方法 ）。

关键字参数

有些函数的参数个数很多，或者有很多行为。很难记住如何调用这种函数。关键字参数，允许通过参数名来区分参数，便于使用、扩展这些复杂接口。

例如，函数 plot 用于画出一条线。此函数有许多可选项，控制线的类型、宽度、颜色等。如果它接收关键字参数，当我们要指明线的宽度时，可以调用 plot(x, y, width=2) 之类的形式。这样的调用方法给参数添加了标签，便于阅读；也可以按任何顺序传递部分参数。

使用关键字参数的函数，在函数签名中使用分号来定义：

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

额外的关键字参数，可以像变参函数中一样，使用 ... 来匹配：

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

在函数 f 内部， args 可以是 (key,value) 多元组的集合，其中 key 是符号。可以在函数调用时使用分号来传递这个集合, 如 f(x, z=1; args...). 这种情况下也可以使用字典。

关键字参数的默认值仅在必要的时候从左至右地被求值(当对应的关键字参数没有被传递)，所以默认的(关键字参数的)表达式可以调用在它之前的关键字参数。

默认值的求值作用域

可选参数和关键字参数的区别在于它们的默认值是怎样被求值的。当可选的参数被求值时，只有在它 之前的 的参数在作用域之内; 与之相对的, 当关键字参数的默认值被计算时, 所有的 参数都是在作用域之内。比如，定义函数：

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

在 a=b 中的 b 指的是该函数的作用域之外的 b ， 而不是接下来 的参数 b。然而，如果 a 和 b 都是关键字参数，那么它们都将在 生成在同一个作用域上， a=b 中的 b 指向的是接下来的参数 b (遮蔽 了任何外层空间的 b), 并且 a=b 会得到未定义变量的错误 (因为默认 参数的表达式是自左而右的求值的， b 并没有被赋值)。

函数参数的块语法

将函数作为参数传递给其它函数，当行数较多时，有时不太方便。下例在多行函数中调用 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 提供了保留字 do 来重写这种代码，使之更清晰：

```
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 语法会建立一个以 x 为参数的匿名函数，并将其作为第一个参数传递给 map . 类似地， do a,b 会创造一个含双参数的匿名函数，而一个普通的

do 将声明其后是一个形式为 () -> ... 的匿名函数。

这些参数的初始化方式取决于"outer"函数；这里 map 将依次将 x 设为 A , B, C, 各自调用匿名函数，效果就像使用语法 map(func, [A, B, C]) 一样。

这一语法使得函数使用更为容易，函数调用就像普通的代码块，从而有效拓展了这一语言。 也有许多不同于 map 的使用方法存在，例如管理系统状态。例如，有一个版本的 open 语法可 确保所打开的文件最终被关闭:

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

这一功能由如下的定义所实现:

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

相较 map 的例子而言，这里 io 由 open("outfile", "w") 返回的的结果初始化。 该串流之后被传递至负责写入的匿名函数；最终，open 函数将保证你的函数退出之后该串流被正确关闭。 try/finally 结构将在 控制流 部分介绍。

do 块结构帮助检查文档以确定用户函数的参数如何被初始化。

延伸阅读

我们有必要在这里说明：以上对于函数定义的解释还远远不够。Julia 有一个复杂的类型系统可以运行使用多重派发。上面并没有给出相关的例子。

- 类型系统的介绍 类型
- 以方法的形式定义函数，并在运行时被多重派发的类型在这里有描述 方法

CHAPTER 9

控制流

Julia 提供一系列控制流：

- 复合表达式： `begin` 和 `(;)`
- 条件求值： `if-elseif-else` 和 `?:` (ternary operator)
- 短路求值： `&&`, `||` 和 chained comparisons
- 重复求值: 循环： `while` 和 `for`
- 异常处理： `try-catch`， `error` 和 `throw`
- 任务（也称为协程）： `yieldto`

前五个控制流机制是高级编程语言的标准。但任务不是：它提供了非本地的控制流，便于在临时暂停的计算中进行切换。在 Julia 中，异常处理和协同多任务都是使用的这个机制。

复合表达式

用一个表达式按照顺序对一系列子表达式求值，并返回最后一个子表达式的值，有两种方法： `begin` 块和 `(;)` 链。 `begin` 块的例子：

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

这个块很短也很简单，可以用 `(;)` 链语法将其放在一行上：

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

这个语法在 函数 中的单行函数定义非常有用。 `begin` 块也可以写成单行， `(;)` 链也可以写成多行：

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

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

条件求值

一个 `if-elseif-else` 条件表达式的例子:

```
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
```

如果条件表达式 `x < y` 为真，相应的语句块将会被执行；否则就执行条件表达式 `x > y`，如果结果为真，相应的语句块将被执行；如果两个表达式都是假，`else` 语句块将被执行。这是它用在实际中的例子:

```
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` 及 `else` 块是可选的。

注意，很短 (单行) 的条件语句在 Julia 中经常被写为短路求值的形式 (见下节).

如果条件表达式的值是除 `true` 和 `false` 之外的值，会出错:

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

“问号表达式”语法 `?:` 与 `if-elseif-else` 语法相关，但是适用于单个表达式:

```
a ? b : c
```

? 之前的 a 是条件表达式，如果为 true ，就执行 : 之前的 b 表达式，如果为 false ，就执行 : 的 c 表达式。

用问号表达式来重写，可以使前面的例子更加紧凑。先看一个二选一的例子：

```
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
```

三选一的例子需要链式调用问号表达式：

```
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
```

链式问号表达式的结合规则是从右到左。

与 if-elseif-else 类似， : 前后的表达式，只有在对应条件表达式为 true 或 false 时才执行：

```
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"
```

短路求值

&& 和 || 布尔运算符被称为短路求值，它们连接一系列布尔表达式，仅计算最少的表达式来确定整个链的布尔值。这意味着：

- 在表达式 a && b 中，只有 a 为 true 时才计算子表达式 b
- 在表达式 a || b 中，只有 a 为 false 时才计算子表达式 b

`&&` 和 `||` 都与右侧结合，但 `&&` 比 `||` 优先级高:

```
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
```

这种方式在 Julia 里经常作为短 `if` 语句的一个简洁的替代。可以把 `if <cond> <statement> end` 写成 `<cond> && <statement>` (读作 **<cond>** 从而 **<statement>**)。类似地，可以把 `if ! <cond> <statement> end` 写成 `<cond> || <statement>` (读作 **<cond>** 要不就 **<statement>**)。

例如, 递归阶乘可以这样写:

```
julia> function factorial(n::Int)
           n >= 0 || error("n must be non-negative")
           n == 0 && return 1
           n * factorial(n-1)
       end
factorial (generic function with 1 method)

julia> factorial(5)
120
```

```
julia> factorial(0)
1

julia> factorial(-1)
ERROR: n must be non-negative
 in factorial at none:2
```

非 短路求值运算符，可以使用 数学运算和基本函数 中介绍的位布尔运算符 `&` 和 `|`：

```
julia> f(1) & t(2)
1
2
false

julia> t(1) | t(2)
1
2
true
```

`&&` 和 `||` 的运算对象也必须是布尔值（`true` 或 `false`）。除了最后一项外，在短路求值中使用非布尔值是一个错误：

```
julia> 1 && true
ERROR: type: non-boolean (Int64) used in boolean context
```

另一方面，短路求值的最后一项可以是任何类型的表达式。取决于之前的条件，它可以被求值并返回。

```
julia> true && (x = rand(2,2))
2x2 Array{Float64,2}:
 0.768448  0.673959
 0.940515  0.395453

julia> false && (x = rand(2,2))
false
```

重复求值: 循环

有两种循环表达式： `while` 循环和 `for` 循环。下面是 `while` 的例子：

```
julia> i = 1;

julia> while i <= 5
         println(i)
         i += 1
       end
1
2
3
4
5
```

上例也可以重写为 `for` 循环：

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

此处的 `1:5` 是一个 `Range` 对象，表示的是 1, 2, 3, 4, 5 序列。 `for` 循环遍历这些数，将其逐一赋给变量 `i` 。 `while` 循环和 `for` 循环的另一区别是变量的作用域。如果在其它作用域中没有引入变量 `i` ，那么它仅存在于 `for` 循环中。不难验证：

```
julia> for j = 1:5
         println(j)
       end
1
2
3
4
5

julia> j
ERROR: j not defined
```

有关变量作用域，详见 变量的作用域 。

通常， `for` 循环可以遍历任意容器。这时，应使用另一个（但是完全等价的）关键词 `in` ，而不是 `=` ，它使得代码更易阅读：

```
julia> for i in [1,4,0]
         println(i)
       end
1
4
0

julia> for s in ["foo","bar","baz"]
         println(s)
       end
foo
bar
baz
```

手册中将介绍各种可迭代容器（详见 多维数组 ）。

有时要提前终止 `while` 或 `for` 循环。可以通过关键词 `break` 来实现：

```
julia> i = 1;

julia> while true
         println(i)
         if i >= 5
           break
         end
         i += 1
       end
1
```

```
2
3
4
5

julia> for i = 1:1000
         println(i)
         if i >= 5
           break
         end
       end
1
2
3
4
5
```

有时需要中断本次循环，进行下一次循环，这时可以用关键字 `continue`：

```
julia> for i = 1:10
         if i % 3 != 0
           continue
         end
         println(i)
       end
3
6
9
```

多层 `for` 循环可以被重写为一个外层循环，迭代类似于笛卡尔乘积的形式：

```
julia> for i = 1:2, j = 3:4
         println((i, j))
       end
(1,3)
(1,4)
(2,3)
(2,4)
```

这种情况下用 `break` 会直接跳出所有循环，而不仅仅是最内层的循环。

异常处理

当遇到意外条件时，函数可能无法给调用者返回一个合理值。这时，要么终止程序，打印诊断错误信息；要么程序员编写异常处理。

内置异常 `Exception`

如果程序遇到意外条件，异常将会被抛出。表中列出内置异常。

Exception
ArgumentError
BoundsError
DivideError
DomainError
EOFError
ErrorException
InexactError
InterruptException
KeyError
LoadError
MemoryError
MethodError
OverflowError
ParseError
SystemError
TypeError
UndefRefError
UndefVarError

例如，当对负实数使用内置的 sqrt 函数时，将抛出 DomainError()：

```
julia> sqrt(-1)
ERROR: DomainError
sqrt will only return a complex result if called with a complex argument.
try sqrt(complex(x))
 in sqrt at math.jl:131
```

你可以用以下方式定义自己的异常：

```
julia> type MyCustomException <: Exception end
```

throw 函数

可以使用 throw 函数显式创建异常。例如，某个函数只对非负数做了定义，如果参数为负数，可以抛出 DomaineError 异常：

```
julia> f(x) = x>=0 ? exp(-x) : throw(DomainError())
f (generic function with 1 method)

julia> f(1)
0.36787944117144233

julia> f(-1)
ERROR: DomainError
 in f at none:1
```

注意， DomainError 使用时需要使用带括号的形式，否则返回的并不是异常，而是异常的类型。必须带括号才能返回 Exception 对象：

```
julia> typeof(DomainError()) <: Exception
true

julia> typeof(DomainError) <: Exception
false
```

另外，某些异常接受一个或多个参数：

```
julia> throw(UndefVarError(:x))
ERROR: x not defined
```

仿照 UndefVarError 定义的方式，这个机制也可以在自定义异常类型中使用：

```
julia> type MyUndefVarError <: Exception
           var::Symbol
       end
julia> Base.showerror(io::IO, e::MyUndefVarError) = print(io, e.var, " not defined");
```

error 函数

error 函数用来产生 ErrorException，阻断程序的正常执行。

如下改写 sqrt 函数，当参数为负数时，提示错误，立即停止执行：

```
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
 in fussy_sqrt at none:1
```

当对负数调用 fussy_sqrt 时，它会立即返回，显示错误信息：

```
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
 in verbose_fussy_sqrt at none:3
```

warn 和 info 函数

Julia 还提供一些函数，用来向标准错误 I/O 输出一些消息，但不抛出异常，因而并不会打断程序的执行：

```
julia> info("Hi"); 1+1
INFO: Hi
2

julia> warn("Hi"); 1+1
WARNING: Hi
2

julia> error("Hi"); 1+1
ERROR: Hi
 in error at error.jl:21
```

try/catch 语句

try/catch 语句可以用于处理一部分预料中的异常 Exception 。例如，下面求平方根函数可以正确处理实数或者复数：

```
julia> f(x) = try
         sqrt(x)
       catch
         sqrt(complex(x, 0))
       end
f (generic function with 1 method)

julia> f(1)
1.0

julia> f(-1)
0.0 + 1.0im
```

但是处理异常比正常采用分支来处理，会慢得多。

try/catch 语句使用时也可以把异常赋值给某个变量。例如：

```
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
```

```
 in sqrt_second at none:7
```

注意，紧跟 catch 的符号会作为异常的名字，所以在将 try/catch 写在单行内的时候需要特别注意。下面的代码 不会 在发生错误的时候返回 x 的值:

```
try bad() catch x end
```

相对的，使用分号或在 catch 后另起一行:

```
try bad() catch; x end

try bad()
catch
  x
end
```

Julia 还提供了更高级的异常处理函数 rethrow， backtrace 和 catch_backtrace 。

finally 语句

在改变状态或者使用文件等资源时，通常需要在操作执行完成时做清理工作（比如关闭文件）。异常的存在使得这样的任务变得复杂，因为异常会导致程序提前退出。关键字 finally 可以解决这样的问题，无论程序是怎样退出的， finally 语句总是会被执行。

例如, 下面的程序说明了怎样保证打开的文件总是会被关闭:

```
f = open("file")
try
    # operate on file f
finally
    close(f)
end
```

当程序执行完 try 语句块（例如因为执行到 return 语句，或者只是正常完成）， close 语句将会被执行。如果 try 语句块因为异常提前退出，异常将会继续传播。 catch 语句可以和 try， finally 一起使用。这时。 finally 语句将会在 catch 处理完异常之后执行。

任务（也称为协程）

任务是一种允许计算灵活地挂起和恢复的控制流，有时也被称为对称协程、轻量级线程、协同多任务等。

如果一个计算（比如运行一个函数）被设计为 Task ，有可能因为切换到其它 Task 而被中断。原先的 Task 在以后恢复时，会从原先中断的地方继续工作。切换任务不需要任何空间，同时可以有任意数量的任务切换，而不需要考虑堆栈问题。任务切换与函数调用不同，可以按照任何顺序来进行。

任务比较适合生产者-消费者模式，一个过程用来生产值，另一个用来消费值。消费者不能简单的调用生产者来得到值，因为两者的执行时间不一定协同。在任务中，两者则可以 正常运行。

Julia 提供了 produce 和 consume 函数来解决这个问题。生产者调用 produce 函数来生产值:

```
julia> function producer()
         produce("start")
         for n=1:4
           produce(2n)
         end
```

```
        produce("stop")
    end;
```

要消费生产的值，先对生产者调用 Task 函数，然后对返回的对象重复调用 consume：

```
julia> p = Task(producer);

julia> consume(p)
"start"

julia> consume(p)
2

julia> consume(p)
4

julia> consume(p)
6

julia> consume(p)
8

julia> consume(p)
"stop"
```

可以在 for 循环中迭代任务，生产的值被赋值给循环变量：

```
julia> for x in Task(producer)
         println(x)
       end
start
2
4
6
8
stop
```

注意 Task() 函数的参数，应为零参函数。生产者常常是参数化的，因此需要为其构造零参 匿名函数 。可以直接写，也可以调用宏：

```
function mytask(myarg)
    ...
end

taskHdl = Task(() -> mytask(7))
# 也可以写成
taskHdl = @task mytask(7)
```

produce 和 consume 但它并不在不同的 CPU 发起线程。我们将在 并行计算 中，讨论真正的内核线程。

核心任务操作

尽管 produce 和 consume 已经阐释了任务的本质，他们实际上是由库函数调用更原始的函数 yieldto 实现的。 yieldto(task,value) 挂起当前任务，切换到指定的 task ， 并使这个 task 的上一次 yeidlto 返回指定的 value 。注意 yieldto 是任务风格的控制流唯一需要的操作；取代调用和返回，我

们只用在不同的任务之间切换即可。这就是为什么 这个特性被称做 “对称式协程”：每一个任务的切换都是用相同的机制。

尽管 yeildto 很强大，但是大多数任务并不直接调用它。这当中的原因可以理解。当你从当前的任务 切换走，你一般还会切换回来。然而正确的处理切换的时机和任务需要相当的协调。例如， procude 需要保持某个状态来记录消费者。无需手动地记录正在消费的任务让 produce 比 yieldto 更容易使用。

除 yieldto 之外，我们也需要一些其他的基本函数以高效地使用任务。 current_task() 获得当前运行任务的引用。 istaskdone(t) 查询任务是否终止。 istaskstarted(t) 查询任务是否启动。 task_local_storage 处理当前任务的键值储存。

任务与事件

大多数任务的切换发生在等待 I/O 请求这样的事件的时候，并由标准库的调度器完成。调度器记录正在运行的任务的队列，并执行一个循环来根据外部事件（比如消息到达）重启任务。

处理等待事件的基本函数是 wait 。有几种对象实现了 wait，比如对于 Process 对象， wait 会等待它终止。更多的时候 wait 是隐式的， 比如 wait 可以发生在调用 read 的时候，以等待可用数据。

在所有的情况中, wait 最终会操作在一个负责将任务排队和重启的 Condition 对象上。 当任务在 Condition 上调用 wait ，任务会被标记为不可运行，并被加入到 Condition 的队列中，再切换至调度器。调度器会选取另一个任务来运行，或者等待外部事件。如果一切正常，最终一个 事件句柄会在 Condition 上调用 notify ，使正在等待的任务再次变得可以运行。

调用 Task 可以生成一个未被调度器管理的任务，这允许你用 yieldto 手动管理任务。不管怎样，当这样的任务正在等待事件时，事件一旦发生，它仍然会自动重启。而且任何时候你都可以调用 schedule(task) 或者用宏 @schedule 或 @async 来让调度器来运行一个任务， 而不用去等待任何事件。(参见 并行计算)

任务状态

任务包含一个 state 域，它用来描述任务的执行状态。任务状态取如下的几种符号中的一种:

符号	意义
:runnable	任务正在运行，或可被切换到该任务
:waiting	正在阻塞等待事件
:queued	在调度器的队列中将要重新开始运行
:done	成功执行完毕
:failed	由于未处理的异常而终止

CHAPTER 10

变量的作用域

变量的 作用域 是变量可见的区域。变量作用域能帮助避免变量命名冲突。

作用域块 是作为变量作用域的代码区域。变量的作用域被限制在这些块内部。作用域块有：

- `function` 函数体（或 语法）
- `while` 循环体
- `for` 循环体
- `try` 块
- `catch` 块
- `let` 块
- `type` 块

注意 *begin* 块 不能 引入新作用域块。

当变量被引入到一个作用域中时，所有的内部作用域都继承了这个变量，除非某个内部作用域显式复写了它。将新变量引入当前作用域的方法：

- 声明 `local x` 或 `const x`，可以引入新本地变量
- 声明 `global x` 使得 `x` 引入当前作用域和更内层的作用域
- 函数的参数，作为新变量被引入函数体的作用域
- 无论是在当前代码之前或 之后，`x = y` 赋值都将引入新变量 `x`，除非 `x` 已经在任何外层作用域内被声明为全局变量或被引入为本地变量

下面例子中，循环内部和外部，仅有一个 `x` 被赋值：

```
function foo(n)
  x = 0
  for i = 1:n
    x = x + 1
  end
  x
```

```
end

julia> foo(10)
10
```

下例中，循环体有一个独立的 x ，函数始终返回 0 ：

```
function foo(n)
  x = 0
  for i = 1:n
    local x
    x = i
  end
  x
end

julia> foo(10)
0
```

下例中， x 仅存在于循环体内部，因此函数在最后一行会遇到变量未定义的错误（除非 x 已经声明为全局变量）：

```
function foo(n)
  for i = 1:n
    x = i
  end
  x
end

julia> foo(10)
in foo: x not defined
```

在非顶层作用域给全局变量赋值的唯一方法，是在某个作用域中显式声明变量是全局的。否则，赋值会引入新的局部变量，而不是给全局变量赋值。

不必在内部使用前，就在外部赋值引入 x ：

```
function foo(n)
  f = y -> n + x + y
  x = 1
  f(2)
end

julia> foo(10)
13
```

上例看起来有点儿奇怪，但是并没有问题。因为在这儿是将一个函数绑定给变量。这使得我们可以按照任意顺序定义函数，不需要像 C 一样自底向上或者提前声明。这儿有个低效率的例子，互递归地验证一个正数的奇偶:

```
even(n) = n == 0 ? true  :  odd(n-1)
odd(n)  = n == 0 ? false : even(n-1)

julia> even(3)
false

julia> odd(3)
true
```

Julia 内置了高效的函数 `iseven` 和 `isodd` 来验证奇偶性。

由于函数可以先被调用再定义，因此不需要提前声明，定义的顺序也可以是任意的。

在交互式模式下，可以假想有一层作用域块包在任何输入之外，类似于全局作用域：

```
julia> for i = 1:1; y = 10; end

julia> y
ERROR: y not defined

julia> y = 0
0

julia> for i = 1:1; y = 10; end

julia> y
10
```

前一个例子中， `y` 仅存在于 `for` 循环中。后一个例子中，外部声明的 `y` 被引入到循环中。由于会话的作用域与全局作用域差不多，因此在循环中不必声明 `global y` 。但是，不在交互式模式下运行的代码，必须声明全局变量。

使用以下的语法形式，可以将多个变量声明为全局变量：

```
function foo()
    global x=1, y="bar", z=3
end

julia> foo()
3

julia> x
1

julia> y
"bar"

julia> z
3
```

`let` 语句提供了另一种引入变量的方法。 `let` 语句每次运行都会声明新变量。 `let` 语法接受由逗号隔开的赋值语句或者变量名：

```
let var1 = value1, var2, var3 = value3
    code
end
```

`let x = x` 是合乎语法的，因为这两个 `x` 变量不同。它先对右边的求值，然后再引入左边的新变量并赋值。下面是个需要使用 `let` 的例子：

```
Fs = cell(2)
i = 1
while i <= 2
  Fs[i] = ()->i
  i += 1
end

julia> Fs[1]()
```

```
3

julia> Fs[2]()
3
```

两个闭包的返回值相同。如果用 `let` 来绑定变量 `i`：

```
Fs = cell(2)
i = 1
while i <= 2
  let i = i
    Fs[i] = ()->i
  end
  i += 1
end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

由于 `begin` 块并不引入新作用域块，使用 `let` 来引入新作用域块是很有用的：

```
julia> begin
         local x = 1
         begin
           local x = 2
         end
         x
       end
ERROR: syntax: local "x" declared twice

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

第一个例子，不能在同一个作用域中声明同名本地变量。第二个例子， `let` 引入了新作用域块，内层的本地变量 `x` 与外层的本地变量 `x` 不同。

For 循环及 Comprehensions

For 循环及 *Comprehensions* 有特殊的行为：在其中声明的新变量，都会在每次循环中重新声明。因此，它有点儿类似于带有内部 `let` 块的 `while` 循环：

```
Fs = cell(2)
for i = 1:2
    Fs[i] = ()->i
end
```

```
julia> Fs[1]()
1

julia> Fs[2]()
2
```

for 循环会复用已存在的变量来迭代：

```
i = 0
for i = 1:3
end
i  # here equal to 3
```

但是, comprehensions 与之不同，它总是声明新变量：

```
x = 0
[ x for x=1:3 ]
x  # here still equal to 0
```

常量

const 关键字告诉编译器要声明常量：

```
const e  = 2.71828182845904523536
const pi = 3.14159265358979323846
```

const 可以声明全局常量和局部常量，最好用它来声明全局常量。全局变量的值（甚至类型）可能随时会改变，编译器很难对其进行优化。如果全局变量不改变的话，可以添加一个 const 声明来解决性能问题。

本地变量则不同。编译器能自动推断本地变量是否为常量，所以本地常量的声明不是必要的。

特殊的顶层赋值默认为常量，如使用 function 和 type 关键字的赋值。

注意 const 仅对变量的绑定有影响；变量有可能被绑定到可变对象（如数组），这个对象仍能被修改。

CHAPTER 11

类型

Julia 中，如果类型被省略，则值可以是任意类型。添加类型会显著提高性能和系统稳定性。

Julia 类型系统 的特性是，具体类型不能作为具体类型的子类型，所有的具体类型都是最终的，它们可以拥有抽象类型作为父类型。其它高级特性有：

- **不区分对象和非对象值：Julia 中的所有值都是一个有类型的对象，这个类型** 属于一个单一、全连通类型图，图中的每个节点都是类型.
- **没有"编译时类型"：程序运行时仅有其实际类型，这在面向对象编程语言中被** 称为"运行时类型".
- 值有类型，变量没有类型——变量仅仅是绑定了值的名字而已.
- 抽象类型和具体类型都可以被其它类型和值参数化.具体来讲, 参数化可以是 符号, 可以是 *isbits* 返回值为 true 的类型任意值 (本质想是讲, 这些数 像整数或者布尔值一样, 储存形式类似于 C 中的数据类型或者 struct, 并且 没有指向其他数据的指针), 也可以是元组. 如果类型参数不需要被使用或者 限制, 可以省略不写.

Julia的类型系统的设计旨在有效及具表现力，既清楚直观又不夸张。许多Julia程序员可能永远不会觉得有必要去明确地指出类型。然而某些程序会因声明类型变得更清晰，更简单，更迅速及健壮。

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。这样做有两个理由：

1. 作为断言，帮助确认程序是否正常运行
2. 给编译器提供额外类型信息，帮助提升性能

`::` 运算符放在表示值的表达式之后时读作"前者是后者的实例"，它用来断言左侧表达式是否为右侧表达式的实例。如果右侧是具体类型，此类型应该是左侧的实例。如果右侧是抽象类型，左侧应是一个具体类型的实例的值，该具体类型是这个抽象类型的子类型。如果类型断言为假，将抛出异常，否则，返回左值：

```
julia> (1+2)::FloatingPoint
ERROR: type: typeassert: expected FloatingPoint, got Int64
```

```
julia> (1+2)::Int
3
```

可以在任何表达式的所在位置做类型断言。`::` 运算符最常见的用法就是用在函数定义中，来声明操作对象的类型，例如 `f(x::Int8) = ...` (详情请看 方法).

`::` 运算符跟在表达式上下文中的 变量名 后时，它声明变量应该是某个类型，有点儿类似于 C 等静态语言中的类型声明。赋给这个变量的值会被 `convert` 函数转换为所声明的类型：

```
julia> function foo()
         x::Int8 = 1000
         x
       end
foo (generic function with 1 method)

julia> foo()
-24

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。

“声明”仅发生在特定的上下文中：

```
x::Int8         # a variable by itself
local x::Int8   # in a local declaration
x::Int8 = 10    # as the left-hand side of an assignment
```

and applies to the whole current scope, even before the declaration. Currently, type declarations cannot be used in global scope, e.g. in the REPL, since Julia does not yet have constant-type globals. Note that in a function return statement, the first two of the above expressions compute a value and then `::` is a type assertion and not a declaration.

抽象类型

抽象类型不能被实例化，它组织了类型等级关系，方便程序员编程。如，编程时可针对任意整数类型，而不需指明是哪种具体的整数类型。

使用 `abstract` 关键字声明抽象类型：

```
abstract «name»
abstract «name» <: «supertype»
```

`abstract` 关键字引入了新的抽象类型，类型名为 `«name»` 。类型名后可跟 `<:` 及已存在的类型，表明新声明的抽象类型是这个“父”类型的子类型。

如果没有指明父类型，则父类型默认为 `Any` ——所有对象和类型都是这个抽象类型的子类型。在类型理论中， `Any` 位于类型图的顶峰，被称为“顶”。Julia 也有预定义的抽象“底”类型，它位于类型图的最底处，被称为 `None` 。 `None``与 ``Any` 对立：任何对象都不是 `None` 的实例，所有的类型都是 `None` 的父类型。

下面是构造 Julia 数值体系的抽象类型子集的具体例子：

```
abstract Number
abstract Real      <: Number
abstract FloatingPoint <: Real
abstract Integer   <: Real
```

```
abstract Signed   <: Integer
abstract Unsigned <: Integer
```

<: 运算符意思为“前者是后者的子类型”，它声明右侧是左侧新声明类型的直接父类型。也可以用来判断左侧是不是右侧的子类型:

```
julia> Integer <: Number
true

julia> Integer <: FloatingPoint
false
```

抽象类型的一个重要用途是为具体的类型提供默认实现. 举个简单的例子

```
function myplus(x, y)
    x + y
endof
```

第一点需要注意的是, 上面的参数声明等效于 x::Any 和 y::Any. 当 这个函数被调用时, 例如 myplus(2, 5), Julia 会首先查找参数类型匹配 的 myplus 函数. (关于多重分派的详细信息请参考下文.) 如果没有找到比上面的函数更相关的函数, Julia 根据上面的通用函数定义并编译一个 myplus 具体函数, 其参数为两个 Int 型变量, 也就是说, Julia 会定义并 编译

```
function myplus(x::Int, y::Int)
    x + y
end
```

最后, 调用这个具体的函数.

因此, 程序员可以利用抽象类型编写通用的函数, 然后这个通用函数可以被许多 具体的类型组合调用. 也正是由于多重分派, 程序员可以精确的控制是调用更具 体的还是通用的函数.

需要注意的一点是, 编写面向抽象类型的函数并不会带来性能上的损失, 因为每 次调用函数时, 根据不同的参数组合, 函数总是要重新编译的. (然而, 如果参 数类型为包含抽象类型的容器是, 会有性能方面的问题; 参见下面的关于性能的 提示.)

位类型

位类型是具体类型，它的数据是由位构成的。整数和浮点数都是位类型。标准的位类型是用 Julia 语言本身定义的:

```
bitstype 16 Float16 <: FloatingPoint
bitstype 32 Float32 <: FloatingPoint
bitstype 64 Float64 <: FloatingPoint

bitstype 8  Bool <: Integer
bitstype 32 Char <: Integer

bitstype 8  Int8     <: Signed
bitstype 8  Uint8    <: Unsigned
bitstype 16 Int16    <: Signed
bitstype 16 Uint16   <: Unsigned
bitstype 32 Int32    <: Signed
bitstype 32 Uint32   <: Unsigned
bitstype 64 Int64    <: Signed
```

```
bitstype 64 Uint64   <: Unsigned
bitstype 128 Int128  <: Signed
bitstype 128 Uint128 <: Unsigned
```

声明位类型的通用语法是:

```
bitstype «bits» «name»
bitstype «bits» «name» <: «supertype»
```

«bits» 表明类型需要多少空间来存储，«name» 为新类型的名字。目前，位类型的声明的位数只支持 8 的倍数，因此布尔类型也是 8 位的。

Bool, Int8 及 Uint8 类型的声明是完全相同的，都占用了 8 位内存，但它们是互相独立的。

复合类型

复合类型 也被称为记录、结构、或者对象。复合类型是变量名域的集合。它是 Julia 中最常用的自定义类型。在 Julia 中，所有的值都是对象，但函数并不与它们所操作的对象绑定。Julia 重载时，根据函数 所有 参数的类型，而不仅仅是第一个参数的类型，来选取调用哪个方法（详见 方法 ）。

使用 type 关键字来定义复合类型:

```
julia> type Foo
         bar
         baz::Int
         qux::Float64
       end
```

构建复合类型 Foo 的对象:

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.",23,1.5)

julia> typeof(foo)
Foo (constructor with 2 methods)
```

当一个类型像函数一样被调用时，它可以被叫做类型构造函数（*constructor*)。 每个类型有两种构造函数是自动被生成的（它们被叫做*默认构造函数*)。 第一种是当传给构造函数的参数和这个类型的字段类型不一一匹配时，构造函数会把它的参数传给 convert 函数，并且转换到这个类型相应的字段类型。 第二种是当传给构造函数的每个参数和这个类型的字段类型都一一相同时，构造函数直接生成类型。 要自动生成两种默认构造函数的原因是：为了防止用户在声明别的新变量的时候不小心把构造函数给覆盖掉。

由于没有约束 bar 的类型，它可以被赋任意值；但是 baz 必须能被转换为 Int :

```
julia> Foo((), 23.5, 1)
ERROR: InexactError()
 in Foo at no file
```

你可以用 names 这个函数来获取类型的所有字段。

```
julia> names(foo)
3-element Array{Symbol,1}:
 :bar
 :baz
 :qux
```

获取复合对象域的值:

```
julia> foo.bar
"Hello, world."

julia> foo.baz
23

julia> foo.qux
1.5
```

修改复合对象域的值:

```
julia> foo.qux = 2
2.0

julia> foo.bar = 1//2
1//2
```

没有域的复合类型是单态类型，这种类型只能有一个实例:

```
type NoFields
end

julia> is(NoFields(), NoFields())
true
```

`is` 函数验证 `NoFields` 的"两个"实例是否为同一个。有关单态类型， 后面 会详细讲。

有关复合类型如何实例化，需要 参数化类型 和 方法 这两个背景知识。将在 构造函数 中详细介绍构造实例。

不可变复合类型

可以使用关键词 `immutable` 替代 `type` 来定义 不可变 复合类型：：

```
immutable Complex
  real::Float64
  imag::Float64
end
```

这种类型和其他复合类型类似，除了它们的实例不能被更改。不可变复合类型具有以下几种优势:

- 它们在一些情况下更高效。像上面‘‘Complex‘‘例子里的类型就被有效地封装到数组里，而且有些时候编译器能够避免完整地分配不可变对象。
- 不会与类型的构造函数提供的不变量冲突。
- 用不可变对象的代码不容易被侵入。

一个不可变对象可以包含可变对象，比如数组，域。那些被包含的可变对象仍然保持可变;只有不可变对象自己的域不能变得指向别的对象。

理解不可变复合变量的一个有用的办法是每个实例都是和特定域的值相关联的 — 这些域的值就能告诉你关于这个对象的一切。相反地，一个可变的对象就如同一个小的容器可能包含了各种各样的值，所以它不能从它的域的值确定出这个对象。在决定是否把一个类型定义为不变的时候，先问问是否两个实例包含相同的域的值就被认为是相同，或者它们会独立地改变。如果它们被认为是相同的，那么这个类型就该被定义成不可变的。

再次强调下, Julia 中不可变类型有两个重要的特性:

- 不可变复合类型的数据在传递时会被拷贝 (在赋值时是这样, 在调用函数时也 是这样), 相对的, 可变类型的数据是以引用的方式互相传递.
- 不可变复合类型内的域不可改变.

对于有着 C/C++ 背景的读者, 需要仔细想下为什么这两个特性是息息相关的. 设想下, 如果这两个特性是分开的, 也就是说, 如果数据在传递时是拷贝的, 然 而数据内部的变量可以被改变, 那么将很难界定某段代码的实际作用. 举个例子, 假设 `x` 是某个函数的参数, 同时假设函数改变了参数中的一个域: `x.isprocessed = true`. 根据 `x` 是值传递或者引用传递, 在调用完函 数是, 原来 `x` 的值有可能没有改变, 也有可能改变. 为了防止出现这种不 确定效应, Julia 限定如果参数是值传递, 其内部域的值不可改变.

被声明类型

以上的三种类型是紧密相关的。它们有相同的特性:

- 明确地被声明
- 有名字
- 有明确的父类
- 可以有参数

正因有共有的特性，这些类型内在地表达为同一种概念的实例，`DataType`,是以下类型之一:

```
julia> typeof(Real)
DataType

julia> typeof(Int)
DataType
```

`DataType` 既可以抽象也可以具体。如果是具体的，它会拥有既定的大小，存储安排和（可选的）名域。所以一个位类型是一个大小非零的 `DataType`，但没有名域。一个复合类型是一个可能拥有名域也可以为空集(大小为零)的 `DataType` 。

在这个系统里的每一个具体的值都是某个 `DataType` 的实例，或者一个多元组。

多元组类型

多元组的类型是类型的多元组:

```
julia> typeof((1,"foo",2.5))
(Int64,ASCIIString,Float64)
```

类型多元组可以在任何需要类型的地方使用:

```
julia> (1,"foo",2.5) :: (Int64,String,Any)
(1,"foo",2.5)

julia> (1,"foo",2.5) :: (Int64,String,Float32)
ERROR: type: typeassert: expected (Int64,String,Float32), got (Int64,ASCIIString,
↪Float64)
```

如果类型多元组中有非类型出现，会报错:

```
julia> (1,"foo",2.5) :: (Int64,String,3)
ERROR: type: typeassert: expected Type{T<:Top}, got (DataType,DataType,Int64)
```

注意，空多元组 () 的类型是其本身：

```
julia> typeof(())
()
```

多元组类型是关于它的组成类型是 协变 的，一个多元组是另一个多元组的子类型 意味着对应的第一个多元组的各元素的类型是第二个多元组对应元素类型的子类型。比如:

```
julia> (Int,String) <: (Real,Any)
true

julia> (Int,String) <: (Real,Real)
false

julia> (Int,String) <: (Real,)
false
```

直观地看，这就像一个函数的各个参数的类型必须是函数签名的子类型（当签名匹配的时候）。

类型共用体

类型共用体是特殊的抽象类型，使用 Union 函数来声明：

```
julia> IntOrString = Union(Int,String)
Union(String,Int64)

julia> 1 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"

julia> 1.0 :: IntOrString
ERROR: type: typeassert: expected Union(String,Int64), got Float64
```

不含任何类型的类型共用体，是“底”类型 None：

```
julia> Union()
None
```

抽象类型 None 是所有其它类型的子类型，且没有实例。零参的 Union 调用，将返回无实例的类型 None。

参数化类型

Julia 的类型系统支持参数化：类型可以引入参数，这样类型声明为每种可能的参数组合声明一个新类型。

所有被声明的类型（ DataType 的变体）都可以使用同样的语法来参数化。我们将按照如下顺序来讨论：参数化符合类型、参数化抽象类型、参数化位类型。

参数化复合类型

类型参数跟在类型名后，用花括号括起来：

```
type Point{T}
  x::T
  y::T
end
```

这个声明定义了新参数化类型 `Point{T}`，它有两个 `T` 类型的"坐标轴"。参数化类型可以是任何类型（也可以是整数，此例中我们用的是类型）。具体类型 `Point{Float64}` 等价于将 `Point` 中的 `T` 替换为 `Float64` 后的类型。上例实际上声明了许多种类型：`Point{Float64}`, `Point{String}`, `Point{Int64}` 等等，因此，现在每个都是可以使用的具体类型：

```
julia> Point{Float64}
Point{Float64} (constructor with 1 method)

julia> Point{String}
Point{String} (constructor with 1 method)
```

`Point` 本身也是个有效的类型对象：

```
julia> Point
Point{T} (constructor with 1 method)
```

`Point` 在这儿是一个抽象类型，它包含所有如 `Point{Float64}`, `Point{String}` 之类的具体实例：

```
julia> Point{Float64} <: Point
true

julia> Point{String} <: Point
true
```

其它类型则不是其子类型：

```
julia> Float64 <: Point
false

julia> String <: 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
```

这一点非常重要：

> 虽然 `Float64 <: Real`，但 `Point{Float64} <: Point{Real}` 不成立！

换句话说，Julia 的类型参数是 不相关 的。尽管 `Point{Float64}` 的实例按照概念来说，应该是 `Point{Real}` 的实例，但两者在内存中的表示上有区别：

- `Point{Float64}` 的实例可以简便、有效地表示 64 位数对儿

- `Point{Real}` 的实例可以表示任意 `Real` 实例的数对儿。由于 `Real` 的实例可以为任意大小、任意结构，因此 `Point{Real}` 实际上表示指向 `Real` 对象的指针对儿

上述区别在数组中更明显： `Array{Float64}` 可以在一块连续内存中存储 64 位浮点数，而 `Array{Real}` 则保存指向每个 `Real` 对象的指针数组。而每个 `Real` 对象的大小，可能比 64 位浮点数的大。

构造函数 中将介绍如何给复合类型自定义构造方法，但如果没有特殊构造声明时，默认有两种构造新复合对象的方法：一种是明确指明构造方法的类型参数；另一种是由对象构造方法的参数来隐含类型参数。

指明构造方法的类型参数：

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

julia> typeof(ans)
Point{Float64} (constructor with 1 method)
```

参数个数应与构造函数相匹配：

```
julia> Point{Float64}(1.0)
ERROR: no method Point{Float64}(Float64)

julia> Point{Float64}(1.0,2.0,3.0)
ERROR: no method Point{Float64}(Float64, Float64, Float64)
```

对于带有类型参数的类型，因为重载构造函数是不可能的，所以只有一种默认构造函数被自动生成——这个构造函数接受任何参数并且把们转换成对应的字段类型并赋值

大多数情况下不需要提供 `Point` 对象的类型，它可由参数类型来提供信息。因此，可以不提供 `T` 的值：

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

julia> typeof(ans)
Point{Float64} (constructor with 1 method)

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

julia> typeof(ans)
Point{Int64} (constructor with 1 method)
```

上例中， `Point` 的两个参数类型相同，因此 `T` 可以省略。但当参数类型不同时，会报错：

```
julia> Point(1,2.5)
ERROR: `Point{T}` has no method matching Point{T}(::Int64, ::Float64)
```

这种情况其实也可以处理，详见 构造函数 。

参数化抽象类型

类似地，参数化抽象类型声明一个抽象类型的集合：

```
abstract Pointy{T}
```

对每个类型或整数值 `T` ， `Pointy{T}` 都是一个不同的抽象类型。 `Pointy` 的每个实例都是它的子类型：

```
julia> Pointy{Int64} <: Pointy
true

julia> Pointy{1} <: Pointy
true
```

参数化抽象类型也是不相关的:

```
julia> Pointy{Float64} <: Pointy{Real}
false

julia> Pointy{Real} <: Pointy{Float64}
false
```

可以如下声明 `Point{T}` 是 `Pointy{T}` 的子类型:

```
type Point{T} <: Pointy{T}
  x::T
  y::T
end
```

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{String} <: Pointy{String}
true
```

它们仍然是不相关的:

```
julia> Point{Float64} <: Pointy{Real}
false
```

参数化抽象类型 `Pointy` 有什么用呢？假设我们要构造一个坐标点的实现，点都在对角线 $x = y$ 上，因此我们只需要一个坐标轴:

```
type DiagPoint{T} <: Pointy{T}
  x::T
end
```

`Point{Float64}` 和 `DiagPoint{Float64}` 都是 `Pointy{Float64}` 抽象类型的实现，这对其它可选类型 `T` 也一样。`Pointy` 可以作为它的子类型的公共接口。有关方法和重载，详见下一节 方法 。

有时需要对 `T` 的范围做限制:

```
abstract Pointy{T<:Real}
```

此时， `T` 只能是 `Real` 的子类型:

```
julia> Pointy{Float64}
Pointy{Float64}

julia> Pointy{Real}
Pointy{Real}
```

```
julia> Pointy{String}
ERROR: type: Pointy: in T, expected T<:Real, got Type{String}

julia> Pointy{1}
ERROR: type: Pointy: in T, expected T<:Real, got Int64
```

参数化复合类型的类型参数，也可以同样被限制：

```
type Point{T<:Real} <: Pointy{T}
  x::T
  y::T
end
```

下面是 Julia 的 `Rational` 的 immutable 类型是如何定义的，这个类型表示分数：

```
immutable Rational{T<:Integer} <: Real
  num::T
  den::T
end
```

单态类型

单态类型是一种特殊的抽象参数化类型。对每个类型 `T`，抽象类型“单态” `Type{T}` 的实例为对象 `T`。来看些例子：

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

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

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

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

换句话说，仅当 `A` 和 `B` 是同一个对象，且此对象是类型时， `isa(A,Type{B})` 才返回真。没有参数时，`Type` 仅是抽象类型，所有的类型都是它的实例，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
```

Julia 中只有类型对象才有单态类型。

参数化位类型

可以参数化地声明位类型。例如，Julia 中指针被定义为位类型：

```
# 32-bit system:
bitstype 32 Ptr{T}

# 64-bit system:
bitstype 64 Ptr{T}
```

这儿的参数类型 `T` 不是用来做类型定义，而是个抽象标签，它定义了一组结构相同的类型，这些类型仅能由类型参数来区分。尽管 `Ptr{Float64}` 和 `Ptr{Int64}` 的表示是一样的，它们是不同的类型。所有的特定指针类型，都是 `Ptr` 类型的子类型：

```
julia> Ptr{Float64} <: Ptr
true

julia> Ptr{Int64} <: Ptr
true
```

类型别名

Julia 提供 `typealias` 机制来实现类型别名。如， `Uint` 是 `Uint32` 或 `Uint64` 的类型别名，这取决于系统的指针大小：

```
# 32-bit system:
julia> Uint
Uint32

# 64-bit system:
julia> Uint
Uint64
```

它是通过 `base/boot.jl` 中的代码实现的：

```
if is(Int,Int64)
    typealias Uint Uint64
else
    typealias Uint Uint32
end
```

对参数化类型， `typealias` 提供了简单的参数化类型名。Julia 的数组类型为 `Array{T,n}` ，其中 `T` 是元素类型， `n` 是数组维度的数值。为简单起见， `Array{Float64}` 可以只指明元素类型而不需指明维度：

```
julia> Array{Float64,1} <: Array{Float64} <: Array
true
```

`Vector` 和 `Matrix` 对象是如下定义的：

```
typealias Vector{T} Array{T,1}
typealias Matrix{T} Array{T,2}
```

类型运算

Julia 中，类型本身也是对象，可以对其使用普通的函数。如 `<:` 运算符，可以判断左侧是否是右侧的子类型。

`isa` 函数检测对象是否属于某个指定的类型：

```
julia> isa(1,Int)
true

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

`typeof` 函数返回参数的类型。类型也是对象，因此它也有类型：

```
julia> typeof(Rational)
DataType

julia> typeof(Union(Real,Float64,Rational))
DataType

julia> typeof((Rational,None))
(DataType,UnionType)
```

类型的类型是什么？它们的类型是 `DataType`：

```
julia> typeof(DataType)
DataType

julia> typeof(UnionType)
DataType
```

读者也许会注意到，`DataType` 类似于空多元组（详见 上文）。因此，递归使用 `()` 和 `DataType` 所组成的多元组的类型，是该类型本身：

```
julia> typeof(())
()

julia> typeof(DataType)
DataType

julia> typeof(((),))
((),)

julia> typeof((DataType,))
(DataType,)

julia> typeof(((),DataType))
((),DataType)
```

`super` 可以指明一些类型的父类型。只有声明的类型(`DataType`)才有父类型：

```
julia> super(Float64)
FloatingPoint

julia> super(Number)
Any

julia> super(String)
Any

julia> super(Any)
Any
```

对其它类型对象（或非类型对象）使用 `super`，会引发 “no method” 错误:

```
julia> super(Union(Float64,Int64))
ERROR: `super` has no method matching super(::Type{Union(Float64,Int64)})

julia> super(None)
ERROR: `super` has no method matching super(::Type{None})

julia> super((Float64,Int64))
ERROR: `super` has no method matching super(::Type{(Float64,Int64)})
```

CHAPTER 12

方法

函数 中说到，函数是从参数多元组映射到返回值的对象，若没有合适返回值则抛出异常。实际中常需要对不同类型的参数做同样的运算，例如对整数做加法、对浮点数做加法、对整数与浮点数做加法，它们都是加法。在 Julia 中，它们都属于同一对象： + 函数。

对同一概念做一系列实现时，可以逐个定义特定参数类型、个数所对应的特定函数行为。 方法 是对函数中某一特定的行为定义。函数中可以定义多个方法。对一个特定的参数多元组调用函数时，最匹配此参数多元组的方法被调用。

函数调用时，选取调用哪个方法，被称为 重载 。 Julia 依据参数个数、类型来进行重载。

定义方法

Julia 的所有标准函数和运算符，如前面提到的 + 函数，都有许多针对各种参数类型组合和不同参数个数而定义的方法。

定义函数时，可以像 复合类型 中介绍的那样，使用 **::** 类型断言运算符，选择性地对参数类型进行限制：

```
julia> f(x::Float64, y::Float64) = 2x + y;
```

此函数中参数 x 和 y 只能是 Float64 类型：

```
julia> f(2.0, 3.0)
7.0
```

如果参数是其它类型，会引发 “no method” 错误：

```
julia> f(2.0, 3)
ERROR: `f` has no method matching f(::Float64, ::Int64)

julia> f(float32(2.0), 3.0)
ERROR: `f` has no method matching f(::Float32, ::Float64)

julia> f(2.0, "3.0")
```

```
ERROR: `f` has no method matching f(::Float64, ::ASCIIString)

julia> f("2.0", "3.0")
ERROR: `f` has no method matching f(::ASCIIString, ::ASCIIString)
```

有时需要写一些通用方法，这时应声明参数为抽象类型：

```
julia> f(x::Number, y::Number) = 2x - y;

julia> f(2.0, 3)
1.0
```

要想给一个函数定义多个方法，只需要多次定义这个函数，每次定义的参数个数和类型需不同。函数调用时，最匹配的方法被重载：

```
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
```

对非数值的值，或参数个数少于 2， `f` 是未定义的，调用它会返回 “no method” 错误：

```
julia> f("foo", 3)
ERROR: `f` has no method matching f(::ASCIIString, ::Int64)

julia> f()
ERROR: `f` has no method matching f()
```

在交互式会话中输入函数对象本身，可以看到函数所存在的方法：

```
julia> f
f (generic function with 2 methods)
```

This output tells us that `f` is a function object with two methods. To find out what the signatures of those methods are, use the `methods` function:

```
julia> methods(f)
# 2 methods for generic function "f":
f(x::Float64,y::Float64) at none:1
f(x::Number,y::Number) at none:1
```

which shows that f has two methods, one taking two `Float64` arguments and one taking arguments of type `Number`. It also indicates the file and line number where the methods were defined: because these methods were defined at the REPL, we get the apparent line number `none:1`.

定义类型时如果没使用 `::` ，则方法参数的类型默认为 `Any` 。对 `f` 定义一个总括匹配的方法：

```
julia> f(x,y) = println("Whoa there, Nelly.");

julia> f("foo", 1)
Whoa there, Nelly.
```

总括匹配的方法，是重载时的最后选择。

重载是 Julia 最强大最核心的特性。核心运算一般都有好几十种方法:

```
julia> methods(+)
# 125 methods for generic function "+":
+(x::Bool) at bool.jl:36
+(x::Bool,y::Bool) at bool.jl:39
+(y::FloatingPoint,x::Bool) at bool.jl:49
+(A::BitArray{N},B::BitArray{N}) at bitarray.jl:848
+(A::Union(DenseArray{Bool,N},SubArray{Bool,N,A<:DenseArray{T,N},I<:(Union(Range
↪{Int64},Int64)...,)}),B::Union(DenseArray{Bool,N},SubArray{Bool,N,A<:DenseArray{T,N}
↪,I<:(Union(Range{Int64},Int64)...,)})) at array.jl:797
+{S,T}(A::Union(SubArray{S,N,A<:DenseArray{T,N},I<:(Union(Range{Int64},Int64)...,)},
↪DenseArray{S,N}),B::Union(SubArray{T,N,A<:DenseArray{T,N},I<:(Union(Range{Int64},
↪Int64)...,)},DenseArray{T,N})) at array.jl:719
+{T<:Union(Int16,Int32,Int8)}(x::T<:Union(Int16,Int32,Int8),y::T<:Union(Int16,Int32,
↪Int8)) at int.jl:16
+{T<:Union(Uint32,Uint16,Uint8)}(x::T<:Union(Uint32,Uint16,Uint8),y::T<:Union(Uint32,
↪Uint16,Uint8)) at int.jl:20
+(x::Int64,y::Int64) at int.jl:33
+(x::Uint64,y::Uint64) at int.jl:34
+(x::Int128,y::Int128) at int.jl:35
+(x::Uint128,y::Uint128) at int.jl:36
+(x::Float32,y::Float32) at float.jl:119
+(x::Float64,y::Float64) at float.jl:120
+(z::Complex{T<:Real},w::Complex{T<:Real}) at complex.jl:110
+(x::Real,z::Complex{T<:Real}) at complex.jl:120
+(z::Complex{T<:Real},x::Real) at complex.jl:121
+(x::Rational{T<:Integer},y::Rational{T<:Integer}) at rational.jl:113
+(x::Char,y::Char) at char.jl:23
+(x::Char,y::Integer) at char.jl:26
+(x::Integer,y::Char) at char.jl:27
+(a::Float16,b::Float16) at float16.jl:132
+(x::BigInt,y::BigInt) at gmp.jl:194
+(a::BigInt,b::BigInt,c::BigInt) at gmp.jl:217
+(a::BigInt,b::BigInt,c::BigInt,d::BigInt) at gmp.jl:223
+(a::BigInt,b::BigInt,c::BigInt,d::BigInt,e::BigInt) at gmp.jl:230
+(x::BigInt,c::Uint64) at gmp.jl:242
+(c::Uint64,x::BigInt) at gmp.jl:246
+(c::Union(Uint32,Uint16,Uint8,Uint64),x::BigInt) at gmp.jl:247
+(x::BigInt,c::Union(Uint32,Uint16,Uint8,Uint64)) at gmp.jl:248
+(x::BigInt,c::Union(Int64,Int16,Int32,Int8)) at gmp.jl:249
+(c::Union(Int64,Int16,Int32,Int8),x::BigInt) at gmp.jl:250
+(x::BigFloat,c::Uint64) at mpfr.jl:147
+(c::Uint64,x::BigFloat) at mpfr.jl:151
+(c::Union(Uint32,Uint16,Uint8,Uint64),x::BigFloat) at mpfr.jl:152
+(x::BigFloat,c::Union(Uint32,Uint16,Uint8,Uint64)) at mpfr.jl:153
+(x::BigFloat,c::Int64) at mpfr.jl:157
+(c::Int64,x::BigFloat) at mpfr.jl:161
+(x::BigFloat,c::Union(Int64,Int16,Int32,Int8)) at mpfr.jl:162
+(c::Union(Int64,Int16,Int32,Int8),x::BigFloat) at mpfr.jl:163
+(x::BigFloat,c::Float64) at mpfr.jl:167
+(c::Float64,x::BigFloat) at mpfr.jl:171
+(c::Float32,x::BigFloat) at mpfr.jl:172
+(x::BigFloat,c::Float32) at mpfr.jl:173
+(x::BigFloat,c::BigInt) at mpfr.jl:177
+(c::BigInt,x::BigFloat) at mpfr.jl:181
+(x::BigFloat,y::BigFloat) at mpfr.jl:328
```

```
+(a::BigFloat,b::BigFloat,c::BigFloat) at mpfr.jl:339
+(a::BigFloat,b::BigFloat,c::BigFloat,d::BigFloat) at mpfr.jl:345
+(a::BigFloat,b::BigFloat,c::BigFloat,d::BigFloat,e::BigFloat) at mpfr.jl:352
+(x::MathConst{sym},y::MathConst{sym}) at constants.jl:23
+{T<:Number}(x::T<:Number,y::T<:Number) at promotion.jl:188
+{T<:FloatingPoint}(x::Bool,y::T<:FloatingPoint) at bool.jl:46
+(x::Number,y::Number) at promotion.jl:158
+(x::Integer,y::Ptr{T}) at pointer.jl:68
+(x::Bool,A::AbstractArray{Bool,N}) at array.jl:767
+(x::Number) at operators.jl:71
+(r1::OrdinalRange{T,S},r2::OrdinalRange{T,S}) at operators.jl:325
+{T<:FloatingPoint}(r1::FloatRange{T<:FloatingPoint},r2::FloatRange{T<:FloatingPoint}
↪) at operators.jl:331
+(r1::FloatRange{T<:FloatingPoint},r2::FloatRange{T<:FloatingPoint}) at operators.
↪jl:348
+(r1::FloatRange{T<:FloatingPoint},r2::OrdinalRange{T,S}) at operators.jl:349
+(r1::OrdinalRange{T,S},r2::FloatRange{T<:FloatingPoint}) at operators.jl:350
+(x::Ptr{T},y::Integer) at pointer.jl:66
+{S,T<:Real}(A::Union(SubArray{S,N,A<:DenseArray{T,N},I<:(Union(Range{Int64},Int64)...
↪,)},DenseArray{S,N}),B::Range{T<:Real}) at array.jl:727
+{S<:Real,T}(A::Range{S<:Real},B::Union(SubArray{T,N,A<:DenseArray{T,N},I
↪<:(Union(Range{Int64},Int64)...,)},DenseArray{T,N})) at array.jl:736
+(A::AbstractArray{Bool,N},x::Bool) at array.jl:766
+{Tv,Ti}(A::SparseMatrixCSC{Tv,Ti},B::SparseMatrixCSC{Tv,Ti}) at sparse/sparsematrix.
↪jl:530
+{TvA,TiA,TvB,TiB}(A::SparseMatrixCSC{TvA,TiA},B::SparseMatrixCSC{TvB,TiB}) at sparse/
↪sparsematrix.jl:522
+(A::SparseMatrixCSC{Tv,Ti<:Integer},B::Array{T,N}) at sparse/sparsematrix.jl:621
+(A::Array{T,N},B::SparseMatrixCSC{Tv,Ti<:Integer}) at sparse/sparsematrix.jl:623
+(A::SymTridiagonal{T},B::SymTridiagonal{T}) at linalg/tridiag.jl:45
+(A::Tridiagonal{T},B::Tridiagonal{T}) at linalg/tridiag.jl:207
+(A::Tridiagonal{T},B::SymTridiagonal{T}) at linalg/special.jl:99
+(A::SymTridiagonal{T},B::Tridiagonal{T}) at linalg/special.jl:98
+{T,MT,uplo}(A::Triangular{T,MT,uplo,IsUnit},B::Triangular{T,MT,uplo,IsUnit}) at␣
↪linalg/triangular.jl:10
+{T,MT,uplo1,uplo2}(A::Triangular{T,MT,uplo1,IsUnit},B::Triangular{T,MT,uplo2,IsUnit}
↪) at linalg/triangular.jl:11
+(Da::Diagonal{T},Db::Diagonal{T}) at linalg/diagonal.jl:44
+(A::Bidiagonal{T},B::Bidiagonal{T}) at linalg/bidiag.jl:92
+{T}(B::BitArray{2},J::UniformScaling{T}) at linalg/uniformscaling.jl:26
+(A::Diagonal{T},B::Bidiagonal{T}) at linalg/special.jl:89
+(A::Bidiagonal{T},B::Diagonal{T}) at linalg/special.jl:90
+(A::Diagonal{T},B::Tridiagonal{T}) at linalg/special.jl:89
+(A::Tridiagonal{T},B::Diagonal{T}) at linalg/special.jl:90
+(A::Diagonal{T},B::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit}) at linalg/
↪special.jl:89
+(A::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit},B::Diagonal{T}) at linalg/
↪special.jl:90
+(A::Diagonal{T},B::Array{T,2}) at linalg/special.jl:89
+(A::Array{T,2},B::Diagonal{T}) at linalg/special.jl:90
+(A::Bidiagonal{T},B::Tridiagonal{T}) at linalg/special.jl:89
+(A::Tridiagonal{T},B::Bidiagonal{T}) at linalg/special.jl:90
+(A::Bidiagonal{T},B::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit}) at linalg/
↪special.jl:89
+(A::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit},B::Bidiagonal{T}) at linalg/
↪special.jl:90
+(A::Bidiagonal{T},B::Array{T,2}) at linalg/special.jl:89
+(A::Array{T,2},B::Bidiagonal{T}) at linalg/special.jl:90
```

```
+(A::Tridiagonal{T},B::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit}) at linalg/
↪special.jl:89
+(A::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit},B::Tridiagonal{T}) at linalg/
↪special.jl:90
+(A::Tridiagonal{T},B::Array{T,2}) at linalg/special.jl:89
+(A::Array{T,2},B::Tridiagonal{T}) at linalg/special.jl:90
+(A::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit},B::Array{T,2}) at linalg/special.
↪jl:89
+(A::Array{T,2},B::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit}) at linalg/special.
↪jl:90
+(A::SymTridiagonal{T},B::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit}) at linalg/
↪special.jl:98
+(A::Triangular{T,S<:AbstractArray{T,2},UpLo,IsUnit},B::SymTridiagonal{T}) at linalg/
↪special.jl:99
+(A::SymTridiagonal{T},B::Array{T,2}) at linalg/special.jl:98
+(A::Array{T,2},B::SymTridiagonal{T}) at linalg/special.jl:99
+(A::Diagonal{T},B::SymTridiagonal{T}) at linalg/special.jl:107
+(A::SymTridiagonal{T},B::Diagonal{T}) at linalg/special.jl:108
+(A::Bidiagonal{T},B::SymTridiagonal{T}) at linalg/special.jl:107
+(A::SymTridiagonal{T},B::Bidiagonal{T}) at linalg/special.jl:108
+{T<:Number}(x::AbstractArray{T<:Number,N}) at abstractarray.jl:358
+(A::AbstractArray{T,N},x::Number) at array.jl:770
+(x::Number,A::AbstractArray{T,N}) at array.jl:771
+(J1::UniformScaling{T<:Number},J2::UniformScaling{T<:Number}) at linalg/
↪uniformscaling.jl:25
+(J::UniformScaling{T<:Number},B::BitArray{2}) at linalg/uniformscaling.jl:27
+(J::UniformScaling{T<:Number},A::AbstractArray{T,2}) at linalg/uniformscaling.jl:28
+(J::UniformScaling{T<:Number},x::Number) at linalg/uniformscaling.jl:29
+(x::Number,J::UniformScaling{T<:Number}) at linalg/uniformscaling.jl:30
+{TA,TJ}(A::AbstractArray{TA,2},J::UniformScaling{TJ}) at linalg/uniformscaling.jl:33
+{T}(a::HierarchicalValue{T},b::HierarchicalValue{T}) at pkg/resolve/versionweight.
↪jl:19
+(a::VWPreBuildItem,b::VWPreBuildItem) at pkg/resolve/versionweight.jl:82
+(a::VWPreBuild,b::VWPreBuild) at pkg/resolve/versionweight.jl:120
+(a::VersionWeight,b::VersionWeight) at pkg/resolve/versionweight.jl:164
+(a::FieldValue,b::FieldValue) at pkg/resolve/fieldvalue.jl:41
+(a::Vec2,b::Vec2) at graphics.jl:60
+(bb1::BoundingBox,bb2::BoundingBox) at graphics.jl:123
+(a,b,c) at operators.jl:82
+(a,b,c,xs...) at operators.jl:83
```

重载和灵活的参数化类型系统一起，使得 Julia 可以抽象表达高级算法，不需关注实现的具体细节，生成有效率、运行时专用的代码。

方法歧义

函数方法的适用范围可能会重叠:

```
julia> g(x::Float64, y) = 2x + y;

julia> g(x, y::Float64) = x + 2y;
Warning: New definition
    g(Any,Float64) at none:1
is ambiguous with:
    g(Float64,Any) at none:1.
To fix, define
```

```
    g(Float64,Float64)
before the new definition.

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
7.0
```

此处 `g(2.0, 3.0)` 既可以调用 `g(Float64, Any)`，也可以调用 `g(Any, Float64)`，两种方法没有优先级。遇到这种情况，Julia会警告定义含糊，但仍会任选一个方法来继续执行。应避免含糊的方法：

```
julia> g(x::Float64, y::Float64) = 2x + 2y;

julia> g(x::Float64, y) = 2x + y;

julia> g(x, y::Float64) = x + 2y;

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
10.0
```

要消除 Julia 的警告，应先定义清晰的方法。

参数化方法

构造参数化方法，应在方法名与参数多元组之间，添加类型参数：

```
julia> same_type{T}(x::T, y::T) = true;

julia> same_type(x,y) = false;
```

这两个方法定义了一个布尔函数，它检查两个参数是否为同一类型：

```
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
```

类型参数可用于函数定义或函数体的任何地方：

```
julia> myappend{T}(v::Vector{T}, x::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: `myappend` has no method matching myappend(::Array{Int64,1}, ::Float64)

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: `myappend` has no method matching myappend(::Array{Float64,1}, ::Int64)
```

下例中，方法类型参数 `T` 被用作返回值：

```
julia> mytypeof{T}(x::T) = T
mytypeof (generic function with 1 method)

julia> mytypeof(1)
Int64

julia> mytypeof(1.0)
Float64
```

方法的类型参数也可以被限制范围：

```
same_type_numeric{T<:Number}(x::T, y::T) = true
same_type_numeric(x::Number, y::Number) = false

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)
no method same_type_numeric(ASCIIString,Float64)
```

```
julia> same_type_numeric("foo", "bar")
no method same_type_numeric(ASCIIString,ASCIIString)

julia> same_type_numeric(int32(1), int64(2))
false
```

same_type_numeric 函数与 same_type 大致相同，但只应用于数对儿。

关于可选参数和关键字参数

函数 中曾简略提到，可选参数是可由多方法定义语法的实现。例如：

```
f(a=1,b=2) = a+2b
```

可以翻译为下面三个方法：

```
f(a,b) = a+2b
f(a) = f(a,2)
f() = f(1,2)
```

关键字参数则与普通的与位置有关的参数不同。它们不用于方法重载。方法重载仅基于位置参数，选取了匹配的方法后，才处理关键字参数。

CHAPTER 13

构造函数

构造函数[1] 是构造新对象，即新 复合类型 实例的函数。在 Julia 中， 类型本身同时也可以作为构造函数使用：将他们作为函数调用可以构造相应类型的实例。这在引入复合类型 的时候已经有过简单的介绍。例如：

```
type Foo
  bar
  baz
end

julia> foo = Foo(1,2)
Foo(1,2)

julia> foo.bar
1

julia> foo.baz
2
```

对于很多类型来说，构造新对象唯一需要的是对各个域赋值。不过，构造新对象也可能会需要更复杂的操作。 为了保证特定的不变性，我们可能需要对参数进行检查或变换。 递归数据结构 ，尤其是自引用的数据结构，常需要先构造为非完整状态，再按步骤将其完善。我们有时也可能希望用更少或不同类型的参数更方便的构造对象。Julia 的构造函数可以让包括这些在内的各种需求得到满足。

外部构造方法

构造函数与 Julia 中的其它函数一样，它的行为取决于它全部方法的行为的组合。因此，你可以通过定义新方法来给构造函数增加新性能。下例给 `Foo` 添加了新构造方法，仅输入一个参数，将该参数值赋给 `bar` 和 `baz` 域：

[1] 关于命名：尽管“构造函数”通常被用来描述创建新对象的函数，它也经常被滥用于特定的构造方法。通常情况下，可以很容易地从上下文推断出到底是“构造函数”还是“构造方法”。

```
Foo(x) = Foo(x,x)

julia> Foo(1)
Foo(1,1)
```

添加 Foo 的零参构造方法，给 bar 和 baz 域赋默认值：

```
Foo() = Foo(0)

julia> Foo()
Foo(0,0)
```

这种追加的构造方法被称为 外部 构造方法。它仅能通过提供默认值的方式，调用其它构造方法来构造实例。

内部构造方法

内部 构造方法与外部构造方法类似，但有两个区别：

1. 它在类型声明块内部被声明，而不是像普通方法一样在外部被声明
2. 它调用本地已存在的 new 函数，来构造声明块的类型的对象

例如，要声明一个保存实数对的类型，且第一个数不大于第二个数：

```
type OrderedPair
  x::Real
  y::Real

  OrderedPair(x,y) = x > y ? error("out of order") : new(x,y)
end
```

仅当 x <= y 时，才会构造 OrderedPair 对象：

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

julia> OrderedPair(2,1)
ERROR: out of order
 in OrderedPair at none:5
```

所有的外部构造方法，最终都会调用内部构造方法。

当然，如果类型被声明为 immutable ，它的构造函数的结构就不能变了。这对判断一个类型是否应该是 immutable 时很重要。

如果定义了内部构造方法，Julia 将不再提供默认的构造方法。默认的构造方法等价于一个自定义内部构造方法，它将对象的所有域作为参数（如果对应域有类型，应为具体类型），传递给 new ，最后返回结果对象：

```
type Foo
  bar
  baz

  Foo(bar,baz) = new(bar,baz)
end
```

这个声明与前面未指明内部构造方法的 Foo 是等价的。下面两者也是等价的，一个使用默认构造方法，一个写明了构造方法：

```
type T1
  x::Int64
end

type 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)
```

内部构造方法能不写就不写。提供默认值之类的事儿，应该写成外部构造方法，由它们调用内部构造方法。

部分初始化

考虑如下递归类型声明：

```
type SelfReferential
  obj::SelfReferential
end
```

如果 a 是 SelfReferential 的实例，则可以如下构造第二个实例：

```
b = SelfReferential(a)
```

但是，当没有任何实例来为 obj 域提供有效值时，如何构造第一个实例呢？唯一的解决方法是构造 obj 域未赋值的 SelfReferential 部分初始化实例，使用这个实例作为另一个实例（如它本身）中 obj 域的有效值。

构造部分初始化对象时，Julia 允许调用 new 函数来处理比该类型域个数少的参数，返回部分域未初始化的对象。这时，内部构造函数可以使用这个不完整的对象，并在返回之前完成它的初始化。下例中，我们定义 SelfReferential 类型时，使用零参内部构造方法，返回一个 obj 域指向它本身的实例：

```
type SelfReferential
  obj::SelfReferential

  SelfReferential() = (x = new(); x.obj = x)
end
```

此构造方法可以运行并构造自引对象：

```
julia> x = SelfReferential();

julia> is(x, x)
true

julia> is(x, x.obj)
true

julia> is(x, x.obj.obj)
true
```

内部构造方法最好返回完全初始化的对象，但也可以返回部分初始化对象：

```
julia> type Incomplete
         xx
         Incomplete() = new()
       end

julia> z = Incomplete();
```

尽管可以构造未初始化域的对象，但读取未初始化的引用会报错：

```
julia> z.xx
ERROR: access to undefined reference
```

这避免了持续检查 `null` 值。但是，所有对象的域都是引用。Julia 认为一些类型是“普通数据”，即他们的数据都是独立的，都不引用其他的对象。普通数据类型是由位类型或者其他普通数据类型的不可变数据结构所构成的（例如 `Int` ）。普通数据类型的初始内容是未定义的：

```
julia> type HasPlain
         n::Int
         HasPlain() = new()
       end

julia> HasPlain()
HasPlain(438103441441)
```

普通数据类型所构成的数组具有相同的行为。

可以在内部构造方法中，将不完整的对象传递给其它函数，来委托完成全部初始化：

```
type Lazy
  xx

  Lazy(v) = complete_me(new(), v)
end
```

如果 `complete_me` 或其它被调用的函数试图在初始化 `Lazy` 对象的 `xx` 域之前读取它，将会立即报错。

参数化构造方法

参数化构造方法的例子：

```
julia> type Point{T<:Real}
         x::T
         y::T
```

```
        end

## implicit T ##

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

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

julia> Point(1,2.5)
ERROR: `Point{T<:Real}` has no method matching Point{T<:Real}(::Int64, ::Float64)

## explicit T ##

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

julia> Point{Int64}(1.0,2.5)
ERROR: InexactError()

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

julia> Point{Float64}(1,2)
Point{Float64}(1.0,2.0)
```

上面的参数化构造方法等价于下面的声明：

```
type Point{T<:Real}
  x::T
  y::T

  Point(x,y) = new(x,y)
end

Point{T<:Real}(x::T, y::T) = Point{T}(x,y)
```

内部构造方法只定义 `Point{T}` 的方法，而非 `Point` 的构造函数的方法。 `Point` 不是具体类型，不能有内部构造方法。外部构造方法定义了 `Point` 的构造方法。

可以将整数值 `1` “提升”为浮点数 `1.0` ，来完成构造：

```
julia> Point(x::Int64, y::Float64) = Point(convert(Float64,x),y);
```

这样下例就可以正常运行：

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

julia> typeof(ans)
Point{Float64} (constructor with 1 method)
```

但下例仍会报错：

```
julia> Point(1.5,2)
ERROR: `Point{T<:Real}` has no method matching Point{T<:Real}(::Float64, ::Int64)
```

其实只需定义下列外部构造方法:

```
julia> Point(x::Real, y::Real) = Point(promote(x,y)...);
```

promote 函数将它的所有参数转换为相同类型。现在，所有的实数参数都可以正常运行:

```
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)
```

案例：分数

下面是 rational.jl 文件的开头部分，它实现了 Julia 的 分数 :

```
immutable Rational{T<:Integer} <: Real
    num::T
    den::T

    function Rational(num::T, den::T)
        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
Rational{T<:Integer}(n::T, d::T) = Rational{T}(n,d)
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)
Rational(n::Integer) = Rational(n,one(n))

//(n::Integer, d::Integer) = Rational(n,d)
//(x::Rational, y::Integer) = x.num // (x.den*y)
//(x::Integer, y::Rational) = (x*y.den) // y.num
//(x::Complex, y::Real) = complex(real(x)//y, imag(x)//y)
//(x::Real, y::Complex) = x*y'//real(y*y')

function //(x::Complex, y::Complex)
    xy = x*y'
    yy = real(y*y')
    complex(real(xy)//yy, imag(xy)//yy)
end
```

复数分数的例子:

```
julia> (1 + 2im)//(1 - 2im)
-3//5 + 4//5*im

julia> typeof(ans)
Complex{Rational{Int64}} (constructor with 1 method)
```

```
julia> ans <: Complex{Rational}
false
```

CHAPTER 14

类型转换和类型提升

Julia 可以将数学运算符的参数提升为同一个类型，这些参数的类型曾经在 整数和浮点数 ， 数学运算和基本函数 ， 类型 ， 及 方法 中提到过。

在某种意义上，Julia 是“非自动类型提升”的：数学运算符只是有特殊语法的函数，函数的参数不会被自动转换。但通过重载，仍能做到“自动”类型提升。

类型转换

`convert` 函数用于将值转换为各种类型。它有两个参数：第一个是类型对象，第二个是要转换的值；返回值是转换为指定类型的值：

```
julia> x = 12
12

julia> typeof(x)
Int64

julia> convert(Uint8, x)
0x0c

julia> typeof(ans)
Uint8

julia> convert(FloatingPoint, x)
12.0

julia> typeof(ans)
Float64
```

遇到不能转换时， `convert` 会引发 “no method” 错误：

```
julia> convert(FloatingPoint, "foo")
ERROR: `convert` has no method matching convert(::Type{FloatingPoint}, ::ASCIIString)
 in convert at base.jl:13
```

Julia 不做字符串和数字之间的类型转换。

定义新类型转换

要定义新类型转换，只需给 `convert` 提供新方法即可。下例将数值转换为布尔值：

```
convert(::Type{Bool}, x::Number) = (x!=0)
```

此方法第一个参数的类型是 单态类型 ， `Bool` 是 `Type{Bool}` 的唯一实例。此方法仅在第一个参数是 `Bool` 才调用。 Notice the syntax used for the first argument: the argument name is omitted prior to the `::` symbol, and only the type is given. This is the syntax in Julia for a function argument whose type is specified but whose value is never used in the function body. In this example, since the type is a singleton, there would never be any reason to use its value within the body. 转换时检查数值是否为 0 ：

```
julia> convert(Bool, 1)
true

julia> convert(Bool, 0)
false

julia> convert(Bool, 1im)
ERROR: InexactError()
 in convert at complex.jl:18

julia> convert(Bool, 0im)
false
```

实际使用的类型转换都比较复杂，下例是 Julia 中的一个实现：

```
convert{T<:Real}(::Type{T}, z::Complex) = (imag(z)==0 ? convert(T,real(z)) :
                                           throw(InexactError()))

julia> convert(Bool, 1im)
InexactError()
 in convert at complex.jl:40
```

案例：分数类型转换

继续 Julia 的 `Rational` 类型的案例研究， rational.jl 中类型转换的声明紧跟在类型声明和构造函数之后：

```
convert{T<:Integer}(::Type{Rational{T}}, x::Rational) = Rational(convert(T,x.num),
↪convert(T,x.den))
convert{T<:Integer}(::Type{Rational{T}}, x::Integer) = Rational(convert(T,x),␣
↪convert(T,1))

function convert{T<:Integer}(::Type{Rational{T}}, x::FloatingPoint, tol::Real)
    if isnan(x); return zero(T)//zero(T); end
    if isinf(x); return sign(x)//zero(T); end
    y = x
    a = d = one(T)
```

```
    b = c = zero(T)
    while true
        f = convert(T,round(y)); y -= f
        a, b, c, d = f*a+c, f*b+d, a, b
        if y == 0 || abs(a/b-x) <= tol
            return a//b
        end
        y = 1/y
    end
end
convert{T<:Integer}(rt::Type{Rational{T}}, x::FloatingPoint) = convert(rt,x,eps(x))

convert{T<:FloatingPoint}(::Type{T}, x::Rational) = convert(T,x.num)/convert(T,x.den)
convert{T<:Integer}(::Type{T}, x::Rational) = div(convert(T,x.num),convert(T,x.den))
```

前四个定义可确保 `a//b == convert(Rational{Int64}, a/b)` 。后两个把分数转换为浮点数和整数类型。

类型提升

类型提升是指将各种类型的值转换为同一类型。它与类型等级关系无关，例如，每个 `Int32` 值都可以被表示为 `Float64` 值，但 `Int32` 不是 `Float64` 的子类型。

Julia 使用 `promote` 函数来做类型提升，其参数个数可以是任意多，它返回同样个数的同一类型的多元组；如果不能提升，则抛出异常。类型提升常用来将数值参数转换为同一类型：

```
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)
```

浮点数值提升为最高的浮点数类型。整数值提升为本地机器的原生字长或最高的整数值类型。既有整数也有浮点数时，提升为可以包括所有值的浮点数类型。既有整数也有分数时，提升为分数。既有分数也有浮点数时，提升为浮点数。既有复数也有实数时，提升为适当的复数。

数值运算中，数学运算符 +, -, * 和 / 等方法定义，都“巧妙”的应用了类型提升。下例是 promotion.jl 中的一些定义：

```
+(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)...)
```

promotion.jl 中还定义了其它算术和数学运算类型提升的方法，但 Julia 标准库中几乎没有调用 promote 。promote 一般用在外部构造方法中，便于使构造函数适应各种不同类型的参数。 rational.jl 中提供了如下的外部构造方法：

```
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)
```

此方法的例子：

```
julia> Rational(int8(15),int32(-5))
-3//1

julia> typeof(ans)
Rational{Int64} (constructor with 1 method)
```

对自定义类型来说，最好由程序员给构造函数显式提供所期待的类型。但处理数值问题时，做自动类型提升比较方便。

定义类型提升规则

尽管可以直接给 promote 函数定义方法，但这太麻烦了。我们用辅助函数 promote_rule 来定义 promote 的行为。 promote_rule 函数接收类型对象对儿，返回另一个类型对象。此函数将参数中的类型的实例，提升为要返回的类型：

```
promote_rule(::Type{Float64}, ::Type{Float32} ) = Float64
```

提升后的类型不需要与函数的参数类型相同。下面是 Julia 标准库中的例子：

```
promote_rule(::Type{Uint8}, ::Type{Int8}) = Int
promote_rule(::Type{Char}, ::Type{Uint8}) = Int32
```

不需要同时定义 promote_rule(::Type{A}, ::Type{B}) 和 promote_rule(::Type{B}, ::Type{A}) —— promote_rule 函数在提升过程中隐含了对称性。

promote_type 函数使用 promote_rule 函数来定义，它接收任意个数的类型对象，返回它们作为 promote 参数时，所应返回值的公共类型。因此可以使用 promote_type 来了解特定类型的组合会提升为哪种类型：

```
julia> promote_type(Int8, Uint16)
Int64
```

promote 使用 promote_type 来决定类型提升时要把参数值转换为哪种类型。完整的类型提升机制可见 promotion.jl ，一共有 35 行。

案例：分数类型提升

我们结束 Julia 分数类型的案例：

```
promote_rule{T<:Integer}(::Type{Rational{T}}, ::Type{T}) = Rational{T}
promote_rule{T<:Integer,S<:Integer}(::Type{Rational{T}}, ::Type{S}) = Rational
↪{promote_type(T,S)}
promote_rule{T<:Integer,S<:Integer}(::Type{Rational{T}}, ::Type{Rational{S}}) =␣
↪Rational{promote_type(T,S)}
promote_rule{T<:Integer,S<:FloatingPoint}(::Type{Rational{T}}, ::Type{S}) = promote_
↪type(T,S)
```

CHAPTER 15

模块

Julia 的模块是一个独立的全局变量工作区。它由句法限制在 `module Name ... end` 之间。在模块内部，你可以控制其他模块的命名是否可见（通过 `import`），也可以指明本模块的命名是否为 public （通过 `export`）。

下面的例子展示了模块的主要特征。这个例子仅为演示：

```
module MyModule
using Lib

using BigLib: thing1, thing2

import Base.show

importall OtherLib

export MyType, foo

type MyType
    x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1

show(io, a::MyType) = print(io, "MyType $(a.x)")
end
```

注意上述例子没有缩进模块体的代码，因为整体缩进没有必要。

这个模块定义了类型 `MyType` 和两个函数。 `foo` 函数和 `MyType` 类型被 export ， 因此可以被 import 进其他模块使用。 `bar` 是 `MyModule` 的私有函数。

语句 `using Lib` 表明， `Lib` 模块在需要时可用来解析命名。若一个全局变量在当前模块中没有被定义，系统会在 `Lib` export 的变量中搜索，并在找到后把它 import 进来。在当前模块中凡是用到这个全局变量时，都会去找 `Lib` 中变量的定义。

语句 `using BigLib: thing1, thing2` 是 `using BigLib.thing1, BigLib.thing2` 的缩写。

The `import` keyword supports all the same syntax as `using`, but only operates on a single name at a time. It does not add modules to be searched the way `using` does. `import` also differs from `using` in that functions must be imported using `import` to be extended with new methods.

In `MyModule` above we wanted to add a method to the standard `show` function, so we had to write `import Base.show`. Functions whose names are only visible via `using` cannot be extended.

The keyword `importall` explicitly imports all names exported by the specified module, as if `import` were individually used on all of them.

Once a variable is made visible via `using` or `import`, a module may not create its own variable with the same name. Imported variables are read-only; assigning to a global variable always affects a variable owned by the current module, or else raises an error.

Summary of module usage

To load a module, two main keywords can be used: `using` and `import`. To understand their differences, consider the following example:

```
module MyModule

export x, y

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

end
```

In this module we export the `x` and `y` functions (with the keyword `export`), and also have the non-exported function `p`. There are several different ways to load the Module and its inner functions into the current workspace:

Import Command	What is brought into scope	Available for method extension
`using MyModule`	All `export` ed names (`x` and `y`), `MyModule.x`, `MyModule.y` and `MyModule.p`	`MyModule.x`, `MyModule.y` and `MyModule.p`
`using MyModule.x, MyModule.p`	`x` and `p`	
`using MyModule: x, p`	`x` and `p`	
`import MyModule`	`MyModule.x`, `MyModule.y` and `MyModule.p`	`MyModule.x`, `MyModule.y` and `MyModule.p`
`import MyModule.x, MyModule.p`	`x` and `p`	`x` and `p`
`import MyModule: x, p`	`x` and `p`	`x` and `p`
`importall MyModule`	All `export` ed names (`x` and `y`)	`x` and `y`

模块和文件

大多数情况下,文件和文件名与模块无关；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
```

在不同的模块中包含同样的代码，会带来类似 mixin 的特征。可以利用这点，在不同的环境定义下运行同样的代码，例如运行一些操作的“安全”版本来进行代码测试：

```
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：Main, Core, 和 Base 。

Main 是顶级模块，Julia 启动时将 Main 设为当前模块。提示符模式下，变量都是在 Main 模块中定义，`whos()` 可以列出 Main 中的所有变量。

Core 包含“内置”的所有标志符，例如部分核心语言，但不包括库。每个模块都隐含地调用了 `using Core` ，因为没有这些声明，什么都做不了。

Base 是标准库（ 在 base/ 文件夹下） 。所有的模块都隐含地调用了 `using Base` ，因为大部分情况下都需要它。

默认顶级声明和裸模块

除了 `using Base` ，模块显式引入了所有的运算符。模块还自动包含 `eval` 函数的定义，这个函数对本模块中的表达式求值。

如果不想要这些定义，可以使用 `baremodule` 关键字来定义模块。使用 `baremodule` 时，一个标准的模块有如下格式：

```
baremodule Mod

using Base

importall Base.Operators

eval(x) = Core.eval(Mod, x)
eval(m,x) = Core.eval(m, x)

...
```

```
end
```

模块的相对和绝对路径

输入指令 `using foo`, Julia 会首先在 `Main` 名字空间中寻找 `Foo` 。如果模块未找到, Julia 会尝试 `require("Foo")` 。通常情况下, 这会从已安装的包中载入模块.

然而，有些模块还有子模块，也就是说，有时候不能从 `Main` 中直接引用一些模块。有两种方法可以解决这个问题：方法一，使用绝对路径，如 `using Base.Sort` 。方法二，使用相对路径，这样可以方便地载入当前模块的子模块或者嵌套的模块：

```
module Parent

module Utils
...
end

using .Utils

...
end
```

模块 `Parent` 包含子模块 `Utils` 。如果想要 `Utils` 中的内容对 `Parent` 可见, 可以使用 `using` 加上英文句号。更多的句号表示在更下一层的命名空间进行搜索。例如， `using ..Utils` 将会在 `Parent` 模块的子模块内寻找 `Utils` 。

模块文件路径

全局变量 `LOAD_PATH` 包含了调用 `require` 时 Julia搜索模块的目录。可以用 `push!` 进行扩展

```
push!(LOAD_PATH, "/Path/To/My/Module/")
```

将这一段代码放在 `~\.juliarc.jl` 里能够在每次 Julia启动时对 `LOAD_PATH` 扩展。 此外，还可以通过定义环境变量 `JULIA_LOAD_PATH` 来扩展 Julia的模块路径。

小提示

如果一个命名是有许可的(qualified)（如 `Base.sin` ），即使它没被 export ，仍能被外部读取。这在调试时非常有用。

import 或 export 宏时，要在宏名字前添加 `@` 符号，例如 `import Mod.@mac` 。Macros in other modules can be invoked as `Mod.@mac` or `@Mod.mac`.

形如 `M.x = y` 的语法是错的，不能给另一个模块中的全局变量赋值；全局变量的赋值都是在变量所在的模块中进行的。

直接在顶层声明为 `global x` ，可以将变量声明为“保留”的。这可以用来防止加载时，全局变量初始化遇到命名冲突。

CHAPTER 16

元编程(Release 0.4.0)

在Julia语言中，对元编程的支持，是继承自Lisp语言的最强大遗产。类似Lisp，Julia自身的代码也是语言本身的数据结构。由于代码是由这门语言本身所构造和处理的对象所表示的，因此程序也可以转换成和生成自身语言的代码。这样不用额外的构建步骤，依然可以生成复杂而精细的高级代码，并且也可以让真正Lisp风格的宏在抽象语法树 (abstract syntax trees) 层面进行操作。与此相反的是，称之为预处理器"宏"系统，例如C和C++就采用了这种系统。它所实现的是，在执行任何实际内插 (inter-pretation) 操作或者从语法上解析 (parse) 操作之前，执行文本处理和代入操作（Julia与此相反）。因为所有在julia中的数据类型和代码都是通过julia数据结构来表示的，所以用反射 (reflection) 功能可以探索程序内部的内容以及这些内容的类型，就像任何其他类型的数据一样。

程序的表示

每一个Julia程序都是从一个字符串开始它的生命的（所有的程序源代码都是字符串）：

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

下一步将发生什么呢?

下一步是把每一个字符串解析 (parse) 成一种被称之为表达式 (Expression) 的对象，用Julia类型 `Expr` 来表示：

```
julia> ex1 = parse(prog)
:(1 + 1)

julia> typeof(ex1)
Expr
```

`Expr` 对象包含三部分：

- 一个 `Symbol` 用来表示表达式对象的种类。符号 (symbol) 是驻留字符串 (interned string) 的标识符（详见下文）。

```
julia> ex1.head
:call
```

- （一堆）表达式对象的参数, 他们可能是符号，其他表达式, 或者立即数/字面值:

```
julia> ex1.args
3-element Array{Any,1}:
 :+
 1
 1
```

- 最后，是表达式对象的返回值的类型, 它可能被用户注释或者被编译器推断出来（而且可以被完全忽略，比如在本章里）：

```
julia> ex1.typ
Any
```

通过前缀符号，表达式对象也可以被直接构建:

```
julia> ex2 = Expr(:call, :+, 1, 1)
:(1 + 1)
```

通过上述两种方式 – 解析或者直接构建 – 构建出的表达式对象是等价的:

```
julia> ex1 == ex2
true
```

这里的要点是，Julia语言的代码内在地被表示成了一种，可以被外界通过Julia语言自身所获取的数据结构

这个 `dump()` 函数可以显示带缩进和注释的表达式对象:

```
julia> dump(ex2)
Expr
  head: Symbol call
  args: Array(Any,(3,))
    1: Symbol +
    2: Int64 1
    3: Int64 1
  typ: Any
```

表达式对象也可以是嵌套形式的:

```
julia> ex3 = parse("(4 + 4) / 2")
:((4 + 4) / 2)
```

另一种查看表达式内部的方法是用 `Meta.show_sexpr` 函数, 它可以把一个给定的表达式对象显示成S-expression形式, Lisp用户肯定会觉得这个形式很眼熟。这有一个例子，用来说明怎样显示一个嵌套形式的表达式对象:

```
julia> Meta.show_sexpr(ex3)
(:call, :/, (:call, :+, 4, 4), 2)
```

符号对象

在Julia中，这个字符： `:` 有两个语法的功能. 第一个功能是创建一个 `Symbol` 对象, 把一个驻留字符串(interned string)用作表达式对象的构建块:

```
julia> :foo
:foo

julia> typeof(ans)
Symbol
```

符号对象也可以被 `symbol()` 函数构建, 它把所有参数的值（数、字符、字符串、现有的符号对象，或者是用 `:` 新构建的符号对象）链接起来，整体创建的一个新的符号对象：

```
julia> :foo == symbol("foo")
true

julia> symbol("func",10)
:func10

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

在表达式对象的语境里, 符号被用来表明变量的值; 当计算一个表达式对象的时候, 每个符号都被替换成它在当前变量作用范围内 (scope) 所代表的值。

有时用额外的圆括号包住的 `:` 来表示 `:` 的字符意义（而不是语法意义，在语法意义中，它表示把自己之后的字符串变成一个符号） 从而避免在解析时出现混淆。

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

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

表达式及其计算

引用 (Quote)

这个 `:` 字符的第二个语法功能是，不用显式的 (explicit)表达式对象构建器，从而构建一个表达式对象。这被称之为引用。 通过使用这个 `:` 字符, 以及后面跟着的由一对圆括号所包住的一条 julia 表达式语句（注意表达式语句和表达式对象不一样，表达式语句就是一条 julia 程序/脚本的语句）, 生成一个基于这条被包括住的语句的表达式对象。 这个例子表明了对一个简短的算数运算的引用:

```
julia> ex = :(a+b*c+1)
:(a + b * c + 1)

julia> typeof(ex)
Expr
```

（为了查看这个表达式对象的结构, 请尝试上文提到过的 `ex.head`、 `ex.args` 或者 `dump()`）

注意：用这种方法构建出来的表达式对象，和用“Expr“对象构建器直接构建，或者用 `parse()` 函数构建，构建出来的表达式对象是等价的:

```
julia>       :(a + b*c + 1)  ==
       parse("a + b*c + 1") ==
       Expr(:call, :+, :a, Expr(:call, :*, :b, :c), 1)
true
```

由解析器 (parser) 生成的表达式对象通常把符号对象、其他表达式对象、或者字面值作为他们的参数, 然而用 julia 代码（即 `Expr()`, `:()` 这些方法）构建的表达式可以不通过字面形式，把任意实时值 (run-time values) 作为参数（比如可以把变量 `a` 的实时值当做参数，而不是变量 `a` 这一字面形式作为参数，后文有详细描述）。 在上面这个具体的例子里, `+` 和 `a` 都是符号对象, `*(b,c)` 是一个子表达式对象, 以及 `1` 是一个字面值（64位有符号整数）

引用的另一种语法是通过“引用块”实现多重表达式： 在引用块里，代码被包含在 quote ... end中。 注意，当直接操作由引用块生成的表达式树时，一定要注意到，这种形式把 QuoteNode 元素引入了表达式树。其他情况下比如 `:( ... )` 和 `quote .. end` 块会被当做一样的对象来处理。

```
julia> ex = quote
           x = 1
           y = 2
           x + y
       end
quote  # none, line 2:
    x = 1 # none, line 3:
    y = 2 # none, line 4:
    x + y
end

julia> typeof(ex)
Expr
```

内插 (Interpolation)

用参数值直接构建表达式对象，这种方法是非常强大的, 但是与“正常”的julia 语法相比， `Expr` 对象构建器就可能显得冗长。作为另一个选择， julia 允许, 把字面值或者表达式对象 “拼接 (splicing) ” 或者“内插” 进 一个被引用的表达式语句（即表达式对象）。 内插的内容之前加 `$` 前缀.

在这个例子里，被内插的是变量 `a` 的值:

```
julia> a = 1;

julia> ex = :($a + b)
:(1 + b)
```

对于没有被引用的表达式语句，是不能做“内插”操作的，并且如果对这种表达式语句做内插操作，将会导致一个编译错误 (compile-time error)。:

```
julia> $a + b
ERROR: unsupported or misplaced expression $
```

在这个例子里, tuple `(1,2,3)` 作为一个表达式语句，先被 `:` 引用成了表达式对象 `b` ，再被内插进一个条件测试:

```
julia> b = :(1,2,3)
julia> ex = :(a in $b )
:($(Expr(:in, :a, :((1,2,3)))))
```

把符号对象内插进一个嵌套的表达式对象需要在一个封闭的引用块（如下所示的 `:(:a + :b)` ）内附每一个符号对象：

```
julia> :( :a in $( :(:a + :b) ) )
                   ^^^^^^^^^^
                被引用的内部表达式
```

用于表达式内插的 `$` 的用法，令人想起字符串内插和指令内插。表达式内插这一功能，使得复杂julia表达式，得以方便，可读，程序化的被构建。

`eval()` 函数及其效果

使用 `eval()` 函数,可以在全局作用域，让 julia 执行 (evaluate) 一个表达式对象：

```
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
```

每个模块都有自己的 `eval()` 函数，用来在全局作用域执行表达式对象。 用 `eval()` 函数执行表达式对象，不仅可以得到返回值 — 而且还有这样一个附带后果：在当前作用域，修改在这个表达式对象中所被修改的状态量：

```
julia> ex = :(x = 1)
:(x = 1)

julia> x
ERROR: UndefVarError: x not defined

julia> eval(ex)
1

julia> x
1
```

这里, 对表达式对象所进行的计算给全局变量 `x` 赋了一个值（1）。

既然表达式语句都是可以通过先程序化的构建表达式对象，再计算这个对象从而生成的， 这也就是说，可以动态的生成任意代码（动态的构建表达式对象），然后这些代码可以用 `eval()` 函数执行。 这里有一个简单的例子：

```
julia> a = 1;

julia> ex = Expr(:call, :+, a, :b)
:(1 + b)
```

```
julia> a = 0; b = 2;

julia> eval(ex)
3
```

a 的值被用来构建表达式对象 ex， ex 用 + 函数来加"值1"和"变量 b "。 注意 a 和 b 的用法有着重要的不同点:

在构建表达式时，变量 a 的值，被用作一个用在表达式中的立即数。 因此, 当计算这个表达式的时候，变量 a 的值是什么都无所谓了： 在表达式中，这个值已经是1了，与‘‘a‘‘这个变量的值后来变成什么就没关系了。

另一方面而言, 符号 :b 被用在了表达式里, 所以在构建表达式时，变量 b 的值就无所谓是多少了 — :b 只是一个符号对象，甚至变量 b 在那个时候（计算 ex 之前）都没必要被定义。然而在计算 ex 的时候, 把这个时候变量 b 的值当做符号 :b 的值，来进行计算。

表达式的函数

正如上文所提示过的, julia 的一个极其有用的特性是用 julia 程序有能力自己生成和操作这个程序自己的代码。我们已经见过这样的一个例子，一个函数的返回值是一个表达式对象：parse() 函数，它输入的是一个 julia 代码构成的字符串，输出的是这些代码所对应的表达式对象。 一个函数也可以把一个或者更多的表达式对象当做参数，然后返回另一个表达式对象。这是一个简单的有启发性的例子:

```
julia> function math_expr(op, op1, op2)
         expr = Expr(:call, op, op1, op2)
         return expr
       end

 julia>  ex = math_expr(:+, 1, Expr(:call, :*, 4, 5))
 :(1 + 4*5)

 julia> eval(ex)
 21
```

比如另一个例子,这里有一个函数，把任何数值参数都翻倍，其他部分不变，只返回新的表达式对象:

```
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
```

Macros

Macros provide a method to include generated code in the final body of a program. A macro maps a tuple of arguments to a returned expression, and the resulting expression is compiled directly rather than requiring a runtime eval() call. Macro arguments may include expressions, literal values, and symbols.

** （以下是0.3.0版本内容） **

宏

宏有点儿像编译时的表达式生成函数。 Just as functions map a tuple of argument values to a return value, macros map a tuple of argument *expressions* to a returned *expression*. They allow the programmer to arbitrarily transform the written code to a resulting expression, which then takes the place of the macro call in the final syntax tree.调用宏的语法为：

```
@name expr1 expr2 ...
@name(expr1, expr2, ...)
```

注意，宏名前有 `@` 符号。第一种形式，参数表达式之间没有逗号；第二种形式，宏名后没有空格。这两种形式不要记混。例如，下面的写法的结果就与上例不同，它只向宏传递了一个参数，此参数为多元组 `(expr1, expr2, ...)`：

```
@name (expr1, expr2, ...)
```

程序运行前， `@name` 展开函数会对表达式参数处理，用结果替代这个表达式。使用关键字 `macro` 来定义展开函数：

```
macro name(expr1, expr2, ...)
    ...
    return resulting_expr
end
```

下例是 Julia 中 `@assert` 宏的简单定义：

```
macro assert(ex)
    return :($ex ? nothing : error("Assertion failed: ", $(string(ex))))
end
```

这个宏可如下使用：

```
julia> @assert 1==1.0

julia> @assert 1==0
ERROR: Assertion failed: 1 == 0
 in error at error.jl:22
```

宏调用在解析时被展开为返回的结果。这等价于：

```
1==1.0 ? nothing : error("Assertion failed: ", "1==1.0")
1==0 ? nothing : error("Assertion failed: ", "1==0")
```

That is, in the first call, the expression `:(1==1.0)` is spliced into the test condition slot, while the value of `string(:(1==1.0))` is spliced into the assertion message slot. The entire expression, thus constructed, is placed into the syntax tree where the `@assert` macro call occurs. Then at execution time, if the test expression evaluates to true, then `nothing` is returned, whereas if the test is false, an error is raised indicating the asserted expression

that was false. Notice that it would not be possible to write this as a function, since only the *value* of the condition is available and it would be impossible to display the expression that computed it in the error message.

The actual definition of `@assert` in the standard library is more complicated. It allows the user to optionally specify their own error message, instead of just printing the failed expression. Just like in functions with a variable number of arguments, this is specified with an ellipses following the last argument:

```
macro assert(ex, msgs...)
    msg_body = isempty(msgs) ? ex : msgs[1]
    msg = string("assertion failed: ", msg_body)
    return :($ex ? nothing : error($msg))
end
```

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()` function:

```
julia> macroexpand(:(@assert a==b))
:(if a == b
        nothing
    else
        Base.error("assertion failed: a == b")
    end)

julia> macroexpand(:(@assert a==b "a should equal b!"))
:(if a == b
        nothing
    else
        Base.error("assertion failed: a should equal b!")
    end)
```

There is yet another case that the actual `@assert` macro handles: what if, in addition to printing "a should equal b," we wanted to print their values? One might naively try to use string interpolation in the custom message, e.g., `@assert a==b "a ($a) should equal b ($b)!"`, but this won't work as expected with the above macro. Can you see why? Recall from *string interpolation* that an interpolated string is rewritten to a call to the `string` function. Compare:

```
julia> typeof(:("a should equal b"))
ASCIIString (constructor with 2 methods)

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: ASCIIString "a ("
    2: Symbol a
    3: ASCIIString ") should equal b ("
    4: Symbol b
    5: ASCIIString ")!"
  typ: Any
```

So now instead of getting a plain string in `msg_body`, the macro is receiving a full expression that will need to be evaluated in order to display as expected. This can be spliced directly into the returned expression as an argument to the `string` call; see error.jl for the complete implementation.

The `@assert` macro makes great use of splicing into quoted expressions to simplify the manipulation of expressions inside the macro body.

卫生宏

卫生宏 是个更复杂的宏。In short, macros must ensure that the variables they introduce in their returned expressions do not accidentally clash with existing variables in the surrounding code they expand into. Conversely, the expressions that are passed into a macro as arguments are often *expected* to evaluate in the context of the surrounding code, interacting with and modifying the existing variables. Another concern arises from the fact that a macro may be called in a different module from where it was defined. In this case we need to ensure that all global variables are resolved to the correct module. Julia already has a major advantage over languages with textual macro expansion (like C) in that it only needs to consider the returned expression. All the other variables (such as `msg` in `@assert` above) follow the *normal scoping block behavior*.

来看一下 `@time` 宏，它的参数是一个表达式。它先记录下时间，运行表达式，再记录下时间，打印出这两次之间的时间差，它的最终值是表达式的值：

```
macro time(ex)
  return quote
    local t0 = time()
    local val = $ex
    local t1 = time()
    println("elapsed time: ", t1-t0, " seconds")
    val
  end
end
```

`t0`, `t1`, 及 `val` 应为私有临时变量，而 `time` 是标准库中的 `time` 函数，而不是用户可能使用的某个叫 `time` 的变量（`println` 函数也如此）。

Julia 宏展开机制是这样解决命名冲突的。首先，宏结果的变量被分类为本地变量或全局变量。如果变量被赋值（且未被声明为全局变量）、被声明为本地变量、或被用作函数参数名，则它被认为是本地变量；否则，它被认为是全局变量。本地变量被重命名为一个独一无二的名字（使用 `gensym` 函数产生新符号），全局变量被解析到宏定义环境中。

但还有个问题没解决。考虑下例：

```
module MyModule
import Base.@time

time() = ... # compute something

@time time()
end
```

此例中， `ex` 是对 `time` 的调用，但它并不是宏使用的 `time` 函数。它实际指向的是 `MyModule.time` 。因此我们应对要解析到宏调用环境中的 `ex` 代码做修改。这是通过 `esc` 函数的对表达式“转义”完成的：

```
macro time(ex)
    ...
    local val = $(esc(ex))
    ...
end
```

这样，封装的表达式就不会被宏展开机制处理，能够正确的在宏调用环境中解析。

必要时这个转义机制可以用来“破坏”卫生，从而引入或操作自定义变量。下例在调用环境中宏将 x 设置为 0 :

```
macro zerox()
  return esc(:(x = 0))
end

function foo()
  x = 1
  @zerox
  x  # is zero
end
```

应审慎使用这种操作。

非标准字符串文本

字符串 中曾讨论过带标识符前缀的字符串文本被称为非标准字符串文本，它们有特殊的语义。例如:

- `r"^\s*(?:#|$)"` 生成正则表达式对象而不是字符串
- `b"DATA\xff\u2200"` 是字节数组文本 `[68,65,84,65,255,226,136,128]`

事实上，这些行为不是 Julia 解释器或编码器内置的，它们调用的是特殊名字的宏。例如，正则表达式宏的定义如下:

```
macro r_str(p)
  Regex(p)
end
```

因此，表达式 `r"^\s*(?:#|$)"` 等价于把下列对象直接放入语法树:

```
Regex("^\\s*(?:#|\$)")
```

这么写不仅字符串文本短，而且效率高：正则表达式需要被编译，而 Regex 仅在 代码编译时 才构造，因此仅编译一次，而不是每次执行都编译。下例中循环中有一个正则表达式:

```
for line = lines
  m = match(r"^\s*(?:#|$)", line)
  if m == nothing
    # non-comment
  else
    # comment
  end
end
```

如果不想使用宏，要使上例只编译一次，需要如下改写:

```
re = Regex("^\\s*(?:#|\$)")
for line = lines
  m = match(re, line)
  if m == nothing
    # non-comment
  else
    # comment
  end
end
```

由于编译器优化的原因，上例依然不如使用宏高效。但有时，不使用宏可能更方便：要对正则表达式内插时必须使用这种麻烦点儿的方式；正则表达式模式本身是动态的，每次循环迭代都会改变，生成新的正则表达式。

不止非标准字符串文本，命令文本语法（`` `echo "Hello, $person"` ``）也是用宏实现的：

```
macro cmd(str)
  :(cmd_gen($shell_parse(str)))
end
```

当然，大量复杂的工作被这个宏定义中的函数隐藏了，但是这些函数也是用 Julia 写的。你可以阅读源代码，看看它如何工作。它所做的事儿就是构造一个表达式对象，用于插入到你的程序的语法树中。

反射

In addition to the syntax-level introspection utilized in metaprogramming, Julia provides several other runtime reflection capabilities.

Type fields The names of data type fields (or module members) may be interrogated using the `names` command. For example, given the following type:

```
type Point
  x::FloatingPoint
  y
end
```

`names(Point)` will return the array `Any[:x, :y]`. The type of each field in a `Point` is stored in the `types` field of the Point object:

```
julia> typeof(Point)
DataType
julia> Point.types
(FloatingPoint,Any)
```

Subtypes The *direct* subtypes of any DataType may be listed using `subtypes(t::DataType)`. For example, the abstract DataType `FloatingPoint` has four (concrete) subtypes:

```
julia> subtypes(FloatingPoint)
4-element Array{Any,1}:
 BigFloat
 Float16
 Float32
 Float64
```

Any abstract subtype will also be included in this list, but further subtypes thereof will not; recursive applications of `subtypes` allow to build the full type tree.

Type internals The internal representation of types is critically important when interfacing with C code. `isbits(T::DataType)` returns true if *T* is stored with C-compatible aligment. The offsets of each field may be listed using `fieldoffsets(T::DataType)`.

Function methods The methods of any function may be listed using `methods(f::Function)`.

Function representations Functions may be introspected at several levels of representation. The lowered form of a function is available using `code_lowered(f::Function, (Args...))`, and the type-inferred lowered form is available using `code_typed(f::Function, (Args...))`.

Closer to the machine, the LLVM Intermediate Representation of a function is printed by `code_llvm(f::Function, (Args...))`, and finally the resulting assembly instructions (after JIT'ing step) are available using `code_native(f::Function, (Args...)`.

CHAPTER 17

多维数组

类似于其它科学计算语言，Julia语言提供了内置的数组。相较于很多科学计算语言都很关注数组在其它容器上的开销。Julia语言并不特别地对待数组。如同其它Julia代码一样，数组基本完全使用Julia本身实现，由编译器本身进行性能优化。同样的，这也使得通过继承 `AbstractArray` 来定制数组成为可能。 更多的细节，请参照 :ref: 抽象数组接口。

数组是一个存在多维网格中的对象集合。通常，数组包含的对象的类型为 `Any` 。对大多数计算而言，数组对象一般更具体为 `Float64` 或 `Int32` 。

总的来说，不像其它的科学计算语言，Julia不需要为了获得高性能而将程序被写成向量化的形式。Julia的编译器使用类型推断生成优化的代码来进行数组索引，这样的编程风格在没有牺牲性能的同时，可读性更好，编写起来更方便，有时候还会使用更少的内存。

有一些科学计算语言会通过值来传递数组，这在很多情况下很方便，而在 Julia 中，参数将通过引用传递给函数，这使得函数中对于一个数组输入的修改在函数外部是可见的。Julia 的库函数不会修改传递给它的输入。用户写代码时，如果要想做类似的功能，要注意先把输入复制一份儿。

数组

基础函数

函数	说明
`eltype(A)`	`A` 中元素的类型
`length(A)`	`A` 中元素的个数
`ndims(A)`	`A` 有几个维度
`size(A)`	返回一个元素为 `A` 的维度的多元组
`size(A,n)`	`A` 在某个维度上的长度
`stride(A,k)`	在维度 `k` 上，邻接元素（在内存中）的线性索引距离
`strides(A)`	返回多元组，其元素为在每个维度上，邻接元素（在内存中）的线性索引距离

函数	说明
eltype(A)	A 中元素的类型
length(A)	A 中元素的个数
ndims(A)	A 的维数
size(A)	返回一个包含 A 中每个维度元素个数的多元组
size(A,n)	A 在某个维度上的大小
indices(A)	返回一个包含 A 中可能的索引的多元组
indices(A,n)	返回一个在 n 维上可能的索引范围
eachindex(A)	一个能够高效地访问每个 A 中的元素的迭代器
stride(A,k)	第‘‘k‘‘维的跨度（相临元素间的索引距离）
strides(A)	返回一个包含每一维度跨度的多元组

构造和初始化

下列函数中调用的 dims... 参数，既可以是维度的单多元组，也可以是维度作为可变参数时的一组值。

函数	描述
Array{type}(dims...)	未初始化的稠密数组
zeros(type, dims...)	指定类型的全 0 数组. 如果未指明 type, 默认为 Float64
zeros(A)	全 0 数组, 元素类型和大小同 A
ones(type, dims...)	指定类型的全 1 数组. 如果未指明 type, 默认为 Float64
ones(A)	全 1 数组, 元素类型和大小同 A
trues(dims...)	全 true 的 Bool 数组
trues(A)	全 true 的 Bool 数组，大小和 A 相同
falses(dims...)	全 false 的 Bool 数组
falses(A)	全 false 的 Bool 数组，大小和 A 相同
reshape(A, dims...)	将数组 A 中的数据按照指定维度排列
copy(A)	复制 A
deepcopy(A)	深度拷贝，递归地复制 A 中的元素
similar(A, element_type, dims...)	属性与输入数组（稠密、稀疏等）相同的未初始化数组，但指明了元素类型和维度。 第二、三参数可省略，省略时默认为 A 的元素类型和维度
reinterpret(type, A)	二进制数据与输入数组相同的数组，但指定了元素类型
rand(dims)	在 [0,1) 上独立均匀同分布的 Float64 类型的随机数组
randn(dims)	Float64 类型的独立正态同分布的随机数组，均值为 0 ，标准差为 1
eye(n)	n x n 单位矩阵
eye(m, n)	m x n 单位矩阵
linspace(start, stop, n)	从 start 至 stop 的由 n 个元素构成的线性向量
fill!(A, x)	用值 x 填充数组 A
fill(x, dims)	创建指定规模的数组, 并使用 x 填充

一维数组（向量）可以通过使用‘‘[A, B, C, ...]‘‘这样的语句来构造。

连接

使用下列函数，可在任意维度连接数组:

Function	Description
`cat(k, A...)`	在第 `k` 维上连接给定的n维数组
`vcat(A...)`	‘‘cat(1, A...)‘‘的简写
`hcat(A...)`	‘‘cat(2, A...)‘‘的简写

传递给这些函数的参数值将被当做只有一个元素的数组

由于连接函数使用的次数很频繁，所以有一些专用的语法来调用它们

表达式	所调用的函数
`[A B C ...]`	`hcat`
`[A, B, C, ...]`	`vcat`
`[A B; C D; ...]`	`hvcat`

`hvcat` 同时连接第一维 (用分号隔开) 和第二维度 (用空格隔开).

指定类型的数组初始化

指定类型为‘‘T‘‘的数组可以使用‘‘T[A, B, C, ...]‘‘来初始化. 这将会创建一个元素类型为‘‘T‘‘，元素初始化为‘‘A‘‘, B, ‘‘C‘‘等的一维数组。比如‘‘Any[x, y, z]‘‘将创建一个包含任何类型的混合数组。

类似地，连接语句也能通过加前缀来指定元素类型

```
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
```

列表推导

列表推导为构造数组提供了一种更加一般，更加强大的方法。它的语法类似于数学中的集合标记法:

```
A = [ F(x,y,...) for x=rx, y=ry, ... ]
```

`F(x,y,...)` 根据变量 `x`, `y` 等来求值。这些变量的值可以是任何迭代对象，但大多数情况下，都使用类似于 `1:n` 或 `2:(n-1)` 的范围对象，或显式指明为类似 `[1.2, 3.4, 5.7]` 的数组。它的结果是一个 N 维稠密数组。

下例计算在维度 1 上，当前元素及左右邻居元素的加权平均数:

```
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
```

输出的数组类型由所计算出的元素类型决定。显式地控制类型可以通过在列表推导的前面加上类型前缀完成。例如，我们可以这样来使得结果都是单精度的浮点数

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

生成器表达式

列表推导也可以被用不闭合的方括号写出，从而产生一个称为生成器的对象。这个对象可以通过迭代来产生所需的值，而不需要提前为一个数组分配内存。（参见 man-interfaces-iteration）。例如下面的表达式会对一列没有分配内存的数求和

```
julia> sum(1/n^2 for n=1:1000)
1.6439345666815615
```

在生成器参数列表中有多个维度的时候，需要通过括号来分割各个参数:

```
julia> map(tuple, 1/(i+j) for i=1:2, j=1:2, [1:4;])
ERROR: syntax: invalid iteration specification
```

所有在 `for` 之后通过逗号分割的表达式将被解释成范围。通过增加括号能够使得我们给 `map` 增加第三个参数:

```
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)
```

生成器和列表推导的范围可以通过多个‘‘for‘‘关键字对外层范围依赖:

```
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)
```

在上面发的情况中，结果都会是一维数组

生成的值可以通过 `if` 关键字过滤

```
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)
```

索引

索引 n 维数组 A 的通用语法为:

```
X = A[I_1, I_2, ..., I_n]
```

其中 I_k 可以是:

1. 标量
2. 满足 `:`, `a:b`, 或 `a:b:c` 格式的 `Range` 对象
3. 能够选取整个维度的“:或者``Colon()``
4. 任意整数数组，包括空数组 `[]`
5. 能够输出所在位置为“true“的索引所对应元素的布尔数组

如果所有的索引都是标量，那么结果 `X`` 就是 ``A 中的单个元素。不然 “X“就是一个和索引有相同维度的数组。

例如如果所有的索引都是向量，那么 `X``的大小就会是``(length(I_1), length(I_2), ..., length(I_n))`, `X``位于``(i_1, i_2, ..., i_n)``的元素具有``A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]``的值。如果``I_1``被变为一个两维的矩阵，这个矩阵就会给``X``增加一个维度，那么``X``就会是一个``n+1``维的数组，大小为``(size(I_1, 1), size(I_1, 2), length(I_2), ..., length(I_n))`。位于“(i_1, i_2, i_3, ..., i_{n+1})“的元素就会有“A[I_1[i_1, i_2], I_2[i_3], ..., I_n[i_{n+1}]]“的值。所有用标量索引的维度的大小会被忽略。比如，`A[2, I, 3]``的结果是一个具有 ``size(I)`` 大小的数组。它的第 ``ith 个元素是“A[2, I[i], 3]“。

使用布尔数组“B“通过:func:‘find(B) <find>‘进行索引和通过向量索引实际上是类似的。它们通常被称作逻辑索引，这将选出那些“B“中值为“true“的元素所在的索引在“A“中的值。一个逻辑索引必须是一个和对应维度有着同样长度的向量，或者是唯一一个和被索引数组的维度以及大小相同的索引。直接使用布尔数组进行索引一般比用:func:‘find(B) <find>‘进行索引更快。

进一步，多维数组的单个元素可以用“x = A[I]“索引，这里“I“ 是一个 “CartesianIndex“（笛卡尔坐标）。它实际上类似于一个 整数“n“元组。具体参见下面的:ref:*man-array-iteration*

“end“关键字是这里比较特殊的一个语法，由于最内层被索引的数组的大小会被确定，它可以在索引的括号中用来表示每个维度最后一个索引。不使用“end“关键字的索引与使用“getindex“一样:

```
X = getindex(A, I_1, I_2, ..., I_n)
```

例子:

```
julia> x = reshape(1:16, 4, 4)
4×4 Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}:
 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[map(ispow2, x)]
5-element Array{Int64,1}:
  1
  2
  4
```

```
  8
 16

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

类似于“n:n-1“的空范围有时可以用来表示索引之间的位置。例如“searchsorted“函数使用这个方法来表示在有序数组中没有出现的元素：

```
julia> a = [1,2,5,6,7];

julia> searchsorted(a, 3)
3:2
```

赋值

给 n 维数组 A 赋值的通用语法为：

```
A[I_1, I_2, ..., I_n] = X
```

其中 I_k 可能是：

1. 标量
2. 满足 `:`, `a:b`, 或 `a:b:c` 格式的 `Range` 对象
3. 能够选取整个维度的“:或者``Colon()`
4. 任意整数数组，包括空数组 `[]`
5. 能够输出所在位置为“true“的索引所对应元素的布尔数组

如果 `X` 是一个数组，它的维度应为 `(length(I_1), length(I_2), ..., length(I_n))` ，且 `A` 在 `i_1, i_2, ..., i_n` 处的值被覆写为 `X[I_1[i_1], I_2[i_2], ..., I_n[i_n]]` 。如果 `X` 不是数组，它的值被写进所有 `A` 被引用的地方。

用于索引的布尔值向量与 `getindex` 中一样（先由 `find` 函数进行转换）。

索引赋值语法等价于调用 `setindex!` ：

```
setindex!(A, X, I_1, I_2, ..., I_n)
```

例如：

```
julia> x = reshape(1:9, 3, 3)
3x3 Array{Int64,2}:
 1  4  7
 2  5  8
 3  6  9

julia> x[1:2, 2:3] = -1
-1

julia> x
3x3 Array{Int64,2}:
 1  -1  -1
```

```
 2  -1  -1
 3   6   9
```

迭代

我们建议使用下面的方法迭代整个数组:

```
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
```

在你需要使用具体的值而不是每个元素的索引的时候，使用第一个方法。在第二种方法里，如果“A“是一个有快速线性索引的数组， i``将是一个``Int``  类型，否则将会是``CartesianIndex:

```
A = rand(4,3)
B = view(A, 1:3, 2:3)
julia> for i in eachindex(B)
           @show i
       end
       i = Base.IteratorsMD.CartesianIndex_2(1,1)
       i = Base.IteratorsMD.CartesianIndex_2(2,1)
       i = Base.IteratorsMD.CartesianIndex_2(3,1)
       i = Base.IteratorsMD.CartesianIndex_2(1,2)
       i = Base.IteratorsMD.CartesianIndex_2(2,2)
       i = Base.IteratorsMD.CartesianIndex_2(3,2)
```

相较“for i = 1:length(A)“，使用“eachindex“更加高效。

数组的特性

如果你写了一个定制的 AbstractArray 类型，你可以用下面的方法声明它有快速线性索引:

```
Base.linearindexing{T<:MyArray}(::Type{T}) = LinearFast()
```

这个设置会让 MyArray``（你所定义的数组类型）的 ``eachindex`` 的迭代使用整数类型。如果你没有声明这个特性，那么会默认使用 ``LinearSlow()。

向量化的运算符和函数

数组支持下列运算符。逐元素进行的运算，应使用带“点”（逐元素）版本的二元运算符。

1. 一元： -, +, !
2. 二元： +, -, *, .*, /, ./, \, .\, ^, .^, div, mod
3. 比较： .==, .!=, .<, .<=, .>, .>=
4. 一元布尔值或位运算： ~
5. 二元布尔值或位运算： &, |, $

有一些运算符在没有‘‘.‘‘运算符的时候，由于有一个参数是标量同样是是逐元素运算的。这些运算符是‘‘*‘‘, +, -，和位运算符。 / 和 ‘‘‘‘ 运算符在分母是标量时也是逐元素计算的。

注意比较运算，在给定一个布尔值的时候，是对整个数组进行的，比如‘‘==‘‘。在逐元素比较时请使用‘‘.‘‘运算符。

Julia为将操作广播至整个数组或者数组和标量的混合变量中，提供了 `f.(args...)` 这样的兼容语句。这样会使调用向量化的数学操作或者其它运算更加方便。例如 `sin.(x)`` 或者 ``min.(x,y)`。（广播操作）详见 man-dot-vectorizing

注意 `min` `max` 和 `minimum` `maximum` 之间的区别，前者是对多个数组操作，找出各数组对应的的元素中的最大最小，后者是作用在一个数组上找出该数组的最大最小值。

广播

有时要对不同维度的数组进行逐元素的二元运算，如将向量加到矩阵的每一列。低效的方法是，把向量复制成同维度的矩阵:

```
julia> a = rand(2,1); A = rand(2,3);

julia> repmat(a,1,3)+A
2x3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846
```

维度很大时，效率会很低。Julia 提供 `broadcast` 函数，它将数组参数的维度进行扩展，使其匹配另一个数组的对应维度，且不需要额外内存，最后再逐元素调用指定的二元函数:

```
julia> broadcast(+, a, A)
2x3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846

julia> b = rand(1,2)
1x2 Array{Float64,2}:
 0.867535  0.00457906

julia> broadcast(+, a, b)
2x2 Array{Float64,2}:
 1.71056  0.847604
 1.73659  0.873631
```

逐元素的运算符，如 `.+` 和 `.*` 将会在必要时进行 broadcasting 。还提供了 `broadcast!` 函数，可以明确指明目的，而 `broadcast_getindex` 和 `broadcast_setindex!` 函数可以在索引前对索引值做 broadcast 。

并且，‘‘broadcast‘‘ 不仅限于数组（参见函数的文档），它也能用于多元组和并将不是数组和多元组的参数当做“标量”对待。

```
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"
```

实现

Julia 的基础数组类型是抽象类型 `AbstractArray{T,N}` ，其中维度为 `N` ，元素类型为 `T` 。`AbstractVector` 和 `AbstractMatrix` 分别是它 1 维 和 2 维的别名。

`AbstractArray` 类型包含任何形似数组的类型， 而且它的实现和通常的数组会很不一样。例如，任何具体的 `AbstractArray{T, N}` 至少要有 `size(A)` (返回 `Int` 多元组)， `getindex(A,i)` 和 `getindex(A,i1,...,iN)` (返回 `T` 类型的一个元素), 可变的数组要能 `setindex!` 。 这些操作都要求在近乎常数的时间复杂度或 O(1) 复杂度，否则某些数组函数就会特别慢。具体的类型也要提供类似于 `similar(A,T=eltype(A),dims=size(A))` 的方法用来分配一个拷贝。

“DenseArray“ 是“AbstractArray“的一个抽象子类型，它包含了所有的在内存中使用常规形式分配内存，并且也因此能够传递给C和Fortran语言的数组。子类型需要提供“stride(A,k)“方法用以返回第“k“维的间隔：给维度 “k“ 索引增加 “1“ 将会给 `getindex(A,i)` 的第 “i“ 个索引增加 `stride(A,k)` 。 如果提供了指针的转换函数:func:*Base.unsafe_convert(Ptr{T}, A) <unsafe_convert>* 那么，内存的分布将会和这些维度的间隔相同。

`Array{T,N}` 类型是 `DenseArray` 的特殊实例，它的元素以列序为主序存储（详见 代码性能优化 ）。`Vector` 和 `Matrix` 是分别是它 1 维 和 2 维的别名。

`SubArray` 是 `AbstractArray` 的特殊实例，它通过引用而不是复制来进行索引。使用 `sub` 函数来构造 `SubArray` ，它的调用方式与 `getindex` 相同（使用数组和一组索引参数）。 `sub` 的结果与 `getindex` 的结果类似，但它的数据仍留在原地。 `sub` 在 `SubArray` 对象中保存输入的索引向量，这个向量将被用来间接索引原数组。

`StridedVector` 和 `StridedMatrix` 是为了方便而定义的别名。通过给他们传递 `Array` 或 `SubArray` 对象，可以使 Julia 大范围调用 BLAS 和 LAPACK 函数，提高内存申请和复制的效率。

下面的例子计算大数组中的一个小块的 QR 分解，无需构造临时变量，直接调用合适的 LAPACK 函数。

```
julia> a = rand(10,10)
10×10 Array{Float64,2}:
 0.561255   0.226678   0.203391  0.308912   ...  0.750307  0.235023   0.217964
 0.718915   0.537192   0.556946  0.996234       0.666232  0.509423   0.660788
 0.493501   0.0565622  0.118392  0.493498       0.262048  0.940693   0.252965
 0.0470779  0.736979   0.264822  0.228787       0.161441  0.897023   0.567641
 0.343935   0.32327    0.795673  0.452242       0.468819  0.628507   0.511528
 0.935597   0.991511   0.571297  0.74485    ...  0.84589   0.178834   0.284413
 0.160706   0.672252   0.133158  0.65554        0.371826  0.770628   0.0531208
 0.306617   0.836126   0.301198  0.0224702      0.39344   0.0370205  0.536062
 0.890947   0.168877   0.32002   0.486136       0.096078  0.172048   0.77672
 0.507762   0.573567   0.220124  0.165816       0.211049  0.433277   0.539476

julia> b = view(a, 2:2:8,2:2:4)
4×2 SubArray{Float64,2,Array{Float64,2},Tuple{StepRange{Int64,Int64},StepRange{Int64,
↪Int64}},false}:
 0.537192  0.996234
 0.736979  0.228787
 0.991511  0.74485
 0.836126  0.0224702
```

```
julia> (q,r) = qr(b);

julia> q
4×2 Array{Float64,2}:
 -0.338809   0.78934
 -0.464815  -0.230274
 -0.625349   0.194538
 -0.527347  -0.534856

julia> r
2×2 Array{Float64,2}:
 -1.58553  -0.921517
  0.0       0.866567
```

稀疏矩阵

稀疏矩阵 是其元素大部分为 0，并以特殊的形式来节省空间和执行时间的存储数据的矩阵。稀疏矩阵适用于当使用这些稀疏矩阵的表示方式能够获得明显优于稠密矩阵的情况。

列压缩（CSC）存储

Julia 中，稀疏矩阵使用 列压缩（CSC）格式 。Julia 稀疏矩阵的类型为 `SparseMatrixCSC{Tv,Ti}`，其中 `Tv` 是非零元素的类型， `Ti` 是整数类型，存储列指针和行索引：

```
type SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column i is in colptr[i]:(colptr[i+1]-1)
    rowval::Vector{Ti}      # Row values of nonzeros
    nzval::Vector{Tv}       # Nonzero values
end
```

列压缩存储便于按列简单快速地存取稀疏矩阵的元素，但按行存取则较慢。把非零值插入 CSC 结构等运算，都比较慢，这是因为稀疏矩阵中，在所插入元素后面的元素，都要逐一移位。

如果你从其他地方获得的数据是 CSC 格式储存的，想用 Julia 来读取，应确保它的序号从 1 开始索引。每一列中的行索引值应该是排好序的。如果你的 *SparseMatrixCSC* 对象包含未排序的行索引值，对它们进行排序的最快的方法是转置两次。

有时，在 *SparseMatrixCSC* 中存储一些零值，后面的运算比较方便。 `Base` 中允许这种行为（但是不保证在操作中会一直保留这些零值）。这些被存储的零被许多函数认为是非零值。 `nnz` 函数返回稀疏数据结构中存储的元素数目，包括被存储的零。要想得到准确的非零元素的数目，请使用 `countnz` 函数，它挨个检查每个元素的值（因此它的时间复杂度不再是常数，而是与元素数目成正比）。

构造稀疏矩阵

稠密矩阵有 `zeros` 和 `eye` 函数，稀疏矩阵对应的函数，在函数名前加 `sp` 前缀即可：

```
julia> spzeros(3,5)
3x5 sparse matrix with 0 Float64 entries:
```

```
julia> speye(3,5)
3x5 sparse matrix with 3 Float64 entries:
        [1, 1]  =  1.0
        [2, 2]  =  1.0
        [3, 3]  =  1.0
```

sparse 函数是比较常用的构造稀疏矩阵的方法。它输入行索引 I，列索引向量 J，以及非零值向量 V。sparse(I,J,V) 构造一个满足 S[I[k], J[k]] = V[k] 的稀疏矩阵:

```
julia> I = [1, 4, 3, 5]; J = [4, 7, 18, 9]; V = [1, 2, -5, 3];

julia> S = sparse(I,J,V)
5x18 sparse matrix with 4 Int64 entries:
        [1 ,  4]  =  1
        [4 ,  7]  =  2
        [5 ,  9]  =  3
        [3 , 18]  =  -5
```

与 sparse 相反的函数为 findn，它返回构造稀疏矩阵时的输入:

```
julia> findn(S)
([1,4,5,3],[4,7,9,18])

julia> findnz(S)
([1,4,5,3],[4,7,9,18],[1,2,3,-5])
```

另一个构造稀疏矩阵的方法是，使用 sparse 函数将稠密矩阵转换为稀疏矩阵:

```
julia> sparse(eye(5))
5x5 sparse matrix with 5 Float64 entries:
        [1, 1]  =  1.0
        [2, 2]  =  1.0
        [3, 3]  =  1.0
        [4, 4]  =  1.0
        [5, 5]  =  1.0
```

可以使用 dense 或 full 函数做逆操作。 issparse 函数可用来检查矩阵是否稀疏:

```
julia> issparse(speye(5))
true
```

稀疏矩阵运算

稠密矩阵的算术运算也可以用在稀疏矩阵上。对稀疏矩阵进行赋值运算，是比较费资源的。大多数情况下，建议使用 findnz 函数把稀疏矩阵转换为 (I,J,V) 格式，在非零数或者稠密向量 (I,J,V) 的结构上做运算，最后再重构回稀疏矩阵。

稠密矩阵和稀疏矩阵函数对应关系

接下来的表格列出了内置的稀疏矩阵的函数, 及其对应的稠密矩阵的函数。通常，稀疏矩阵的函数，要么返回与输入稀疏矩阵 S 同样的稀疏度，要么返回 d 稠密度，例如矩阵的每个元素是非零的概率为 d 。

详见可以标准库文档的 *Sparse Matrices* 章节。

稀疏矩阵	稠密矩阵	说明
`spzeros(m,n)`	`zeros(m,n)`	构造 *m* x *n* 的全 0 矩阵 (`spzeros(m,n)` 是空矩阵)
`spones(S)`	`ones(m,n)`	构造的全 1 矩阵 与稠密版本的不同， `spones` 的稀疏 度与 *S* 相同
`speye(n)`	`eye(n)`	构造 *m* x *n* 的单位矩阵
`full(S)`	`sparse(A)`	转换为稀疏矩阵和稠密矩阵
`sprand(m,n,d)`	`rand(m,n)`	构造 *m*-by-*n* 的随机矩阵（稠密度为 *d*） 独立同分布的非零元素在 [0, 1] 内均匀分布
`sprandn(m,n,d)`	`randn(m,n)`	构造 *m*-by-*n* 的随机矩阵（稠密度为 *d*） 独立同分布的非零元素满足标准正 态（高斯）分布
`sprandn(m,n,d,X)`	`randn(m,n,X)`	构造 *m*-by-*n* 的随机矩阵（稠密度为 *d*） 独立同分布的非零元素满足 *X* 分 布。（需要 `Distributions` 扩展包）
`sprandbool(m,n,d)`	`randbool(m,n)`	构造 *m*-by-*n* 的随机矩阵（稠密度为 *d*），非零 `Bool``元素的概率为 *d* (``randbool` 中 *d* =0.5)

CHAPTER 18

线性代数

矩阵分解

矩阵分解 是将一个矩阵分解为数个矩阵的乘积，是线性代数中的一个核心概念。

下面的表格总结了在 Julia 中实现的几种矩阵分解方式。具体的函数可以参考标准库文档的 *Linear Algebra* 章节。

`Cholesky`	Cholesky 分解
`CholeskyPivoted`	主元 Cholesky 分解
`LU`	LU 分解
`LUTridiagonal`	LU factorization for Tridiagonal matrices
`UmfpackLU`	LU factorization for sparse matrices (computed by UMFPack)
`QR`	QR factorization
`QRCompactWY`	Compact WY form of the QR factorization
`QRPivoted`	主元 QR 分解
`Hessenberg`	Hessenberg 分解
`Eigen`	特征分解
`SVD`	奇异值分解
`GeneralizedSVD`	广义奇异值分解

特殊矩阵

线性代数中经常碰到带有对称性结构的特殊矩阵，这些矩阵经常和矩阵分解联系到一起。 Julia 内置了非常丰富的特殊矩阵类型，可以快速地对特殊矩阵进行特定的操作.

下面的表格总结了 Julia 中特殊的矩阵类型，其中也包含了 LAPACK 中的一些已经优化过的运算。

Hermitian	埃尔米特矩阵
Triangular	上/下 三角矩阵
Tridiagonal	三对角矩阵
SymTridiagonal	对称三对角矩
Bidiagonal	上/下 双对角矩阵
Diagonal	对角矩阵
UniformScaling	缩放矩阵

基本运算

矩阵类型	+	-	*	\	其它已优化的函数
Hermitian				XY	inv, sqrtm, expm
Triangular			XY	XY	inv, det
SymTridiagonal	X	X	XZ	XY	eigmax/min
Tridiagonal	X	X	XZ	XY	
Bidiagonal	X	X	XZ	XY	
Diagnoal	X	X	XY	XY	inv, det, logdet, /
UniformScaling	X	X	XYZ	XYZ	/

图例:

X	已对矩阵-矩阵运算优化
Y	已对矩阵-向量运算优化
Z	已对矩阵-标量运算优化

矩阵分解

矩阵类型	LAPACK	eig	eigvals	eigvecs	svd	svdvals
Hermitian	HE		ABC			
Triangular	TR					
SymTridiagonal	ST	A	ABC	AD		
Tridiagonal	GT					
Bidiagonal	BD				A	A
Diagonal	DI		A			

图例:

A	已对寻找特征值和/或特征向量优化	例如 eigvals(M)
B	已对寻找 ilth 到 ihth 特征值优化	eigvals(M, il, ih)
C	已对寻找在 [vl, vh] 之间的特征值优化	eigvals(M, vl, vh)
D	已对寻找特征值 x=[x1, x2,...] 所对应的特征向量优化	eigvecs(M, x)

缩放运算

A UniformScaling operator represents a scalar times the identity operator, $\lambda*I$. 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.

CHAPTER 19

网络和流

Julia 提供了一个丰富的接口处理终端、管道、tcp套接字等等I/O流对象。 接口在系统层的实现是异步的，开发者以同步的方式调用该接口、一般无需关注底层异步实现。 接口实现主要基于Julia支持的协程(coroutine)功能。

基本流 I/O

所有Julia流都至少提供一个‘read‘和一个‘write‘方法，且第一个参数都是流对象，例如:

```
julia> write(STDOUT,"Hello World")
Hello World

julia> read(STDIN,Char)

'\n'
```

注意我又输入了一次回车，这样Julia会读入换行符。现在，由例子可见， ‘write‘方法的第二个参数是将要写入的数据，‘read‘方法的第二个参数是即将读入的数据类型。 例如，要读入一个简单的字节数组，我们可以:

```
julia> x = zeros(Uint8,4)
4-element Uint8 Array:
 0x00
 0x00
 0x00
 0x00

julia> read(STDIN,x)
abcd
4-element Uint8 Array:
 0x61
 0x62
 0x63
 0x64
```

不过像上面这么写有点麻烦，还提供了一些简化的方法。例如，我们可以将上例重写成:

```
julia> readbytes(STDIN,4)
abcd
4-element Uint8 Array:
 0x61
 0x62
 0x63
 0x64
```

或者直接读入整行数据:

```
julia> readline(STDIN)
abcd
"abcd\n"
```

注意这取决于你的终端配置，你的TTY可能是行缓冲、需要多输入一个回车才会把数据传给julia。

如果想要读入 STDIN 中的每一行，你可以使用 eachline 函数:

```
for line in eachline(STDIN)
    print("Found $line")
end
```

当然，你有可能会想一个字符一个字符地读:

```
while !eof(STDIN)
    x = read(STDIN, Char)
    println("Found: $x")
end
```

文本 I/O

注意上面所说的write方法是用来操作二进制流的，也就是说读入的值不会转换成任何其他格式，即使输出的时候看起来好像转换了一样:

```
julia> write(STDOUT,0x61)
a
```

对于字符 I/O，应该使用‘print‘或‘show‘方法 (关于它们有什么区别，你可以去看看标准库的文档):

```
julia> print(STDOUT,0x61)
97
```

处理文件

很自然地，Julia也会有一个‘open‘函数，可以输入一个文件名，返回一个‘IOStream‘对象。 你可以用这个对象来对文件进行输入输出，比如说我们打开了一个文件‘hello.txt‘，里面就一行”Hello, World!”:

```
julia> f = open("hello.txt")
IOStream(<file hello.txt>)

julia> readlines(f)
```

```
1-element Array{Union(ASCIIString,UTF8String),1}:
 "Hello, World!\n"
```

如果你想往里面输出些东西，你需要在打开的时候加上一个(“*w*”):

```
julia> f = open("hello.txt","w")
IOStream(<file hello.txt>)

julia> write(f,"Hello again.")
12
```

如果你这时手动点开‘hello.txt‘你会看到并没有东西被写进去，这是因为IOStream被关闭之后，真正的写入才会完成:

```
julia> close(f)
```

现在你可以去点开看看，此时文件已经写入了内容。

打开一个文件，对其内容做出一些修改，然后关闭它，这是很常用的操作流程。 为了简化这个常用操作，我们有另一个使用‘open‘的方式，你可以传入一个函数作为第一个参数，然后文件名作为第二个参数。打开文件后，文件将会传入你的函数，做一点微小的工作，然后自动‘close‘。 比如说我们写出下面这个函数:

```
function read_and_capitalize(f::IOStream)
    return uppercase(readall(f))
end
```

你可以这样用:

```
julia> open(read_and_capitalize, "hello.txt")
"HELLO AGAIN."
```

打开了‘hello.txt‘，对它施放‘read_and_capitalize‘，然后关闭掉‘hello.txt‘，然后返回大写的文字，在REPL显示出来。

为了省去你打函数名的劳累，你还可以使用‘do‘语法来创建一个匿名函数，此处f是匿名函数的形参:

```
julia> open("hello.txt") do f
          uppercase(readall(f))
       end
"HELLO AGAIN."
```

简单的 TCP 例子

我们来看看下面这个使用Tcp Sockets的例子，首先创建一个简单的服务器程序:

```
julia> @async begin
         server = listen(2000)
         while true
           sock = accept(server)
           println("Hello World\n")
         end
       end
Task

julia>
```

对于了解Unix socket API的人来说，我们用到的方法名看起来很熟悉， 尽管它们用起来比Unix socket API简单。首先在这个例子中，‘listen‘方法将会创建一个监听(2000)端口等待连接的服务器。它还可以用于创建各种各样其他种类的服务器:

```
julia> listen(2000) # Listens on localhost:2000 (IPv4)
TcpServer(active)

julia> listen(ip"127.0.0.1",2000) # Equivalent to the first
TcpServer(active)

julia> listen(ip"::1",2000) # Listens on localhost:2000 (IPv6)
TcpServer(active)

julia> listen(IPv4(0),2001) # Listens on port 2001 on all IPv4 interfaces
TcpServer(active)

julia> listen(IPv6(0),2001) # Listens on port 2001 on all IPv6 interfaces
TcpServer(active)

julia> listen("testsocket") # Listens on a domain socket/named pipe
PipeServer(active)
```

注意，最后一个调用的返回值是不一样的，这是因为这个服务器并不是监听TCP，而是监听一个Named Pipe(Windows黑科技术语)，也叫Domain Socket(UNIX术语)。The difference 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
```

As expected we saw "Hello World" printed. So, let's actually analyze what happened behind the scenes. When we called connect, we connect to the server we had just created. Meanwhile, the accept function returns a server-side connection to the newly created socket and prints "Hello World" to indicate that the connection was successful.

A great strength of Julia is that since the API is exposed synchronously even though the I/O is actually happening asynchronously, we didn't have to worry callbacks or even making sure that the server gets to run. When we called *connect* the current task waited for the connection to be established and only continued executing after that was done. In this pause, the server task resumed execution (because a connection request was now available), accepted the connection, printed the message and waited for the next client. Reading and writing works in the same way. To see this, consider the following simple echo server:

```
julia> @async begin
         server = listen(2001)
         while true
           sock = accept(server)
           @async while true
             write(sock,readline(sock))
           end
         end
       end
Task

julia> clientside=connect(2001)
```

```
TcpSocket(open, 0 bytes waiting)

julia> @async while true
          write(STDOUT,readline(clientside))
       end

julia> println(clientside,"Hello World from the Echo Server")

julia> Hello World from the Echo Server
```

解析 IP 地址

One of the *connect* methods that does not follow the *listen* methods is *connect(host::ASCIIString,port)*, which will attempt to connect to the host given by the *host* parameter on the port given by the port parameter. It allows you to do things like:

```
julia> connect("google.com",80)
TcpSocket(open, 0 bytes waiting)
```

At the base of this functionality is the getaddrinfo function which will do the appropriate address resolution:

```
julia> getaddrinfo("google.com")
IPv4(74.125.226.225)
```

CHAPTER 20

并行计算

Julia 提供了一个基于消息传递的多处理器环境，能够同时在多处理器上使用独立的内存空间运行程序。

Julia 的消息传递与 MPI[1] 等环境不同。Julia 中的通信是“单边”的，即程序员只需要管理双处理器运算中的一个处理器即可。

Julia 中的并行编程基于两个原语：*remote references* 和 *remote calls* 。remote reference 对象，用于从任意的处理器，查阅指定处理器上存储的对象。 remote call 请求，用于一个处理器对另一个（也有可能是同一个）处理器调用某个函数处理某些参数。 remote call 返回 remote reference 对象。 remote call 是立即返回的；调用它的处理器继续执行下一步操作，而 remote call 继续在某处执行。可以对 remote reference 调用 `wait` ，以等待 remote call 执行完毕，然后通过 `fetch` 获取结果的完整值。使用 `put` 可将值存储到 remote reference 。

通过 `julia -p n` 启动，可以在本地机器上提供 `n` 个处理器。一般 `n` 等于机器上 CPU 内核个数：

```
$ ./julia -p 2

julia> r = remotecall(2, rand, 2, 2)
RemoteRef(2,1,5)

julia> fetch(r)
2x2 Float64 Array:
 0.60401   0.501111
 0.174572  0.157411

julia> s = @spawnat 2 1 .+ fetch(r)
RemoteRef(2,1,7)

julia> fetch(s)
2x2 Float64 Array:
 1.60401  1.50111
 1.17457  1.15741
```

[1] In this context, MPI refers to the MPI-1 standard. Beginning with MPI-2, the MPI standards committee introduced a new set of communication mechanisms, collectively referred to as Remote Memory Access (RMA). The motivation for adding RMA to the MPI standard was to facilitate one-sided communication patterns. For additional information on the latest MPI standard, see http://www.mpi-forum.org/docs.

remote_call 的第一个参数是要进行这个运算的处理器索引值。Julia 中大部分并行编程不查询特定的处理器或可用处理器的个数，但可认为 remote_call 是个为精细控制所提供的低级接口。第二个参数是要调用的函数，剩下的参数是该函数的参数。此例中，我们先让处理器 2 构造一个 2x2 的随机矩阵，然后我们在结果上加 1 。两个计算的结果保存在两个 remote reference 中，即 r 和 s 。 @spawnat 宏在由第一个参数指明的处理器上，计算第二个参数中的表达式。

remote_call_fetch 函数可以立即获取要在远端计算的值。它等价于 fetch(remote_call(...)) ，但比之更高效：

```
julia> remotecall_fetch(2, getindex, r, 1, 1)
0.10824216411304866
```

getindex(r,1,1) 等价于 r[1,1] ，因此，这个调用获取 remote reference 对象 r 的第一个元素。

remote_call 语法不太方便。 @spawn 宏简化了这件事儿，它对表达式而非函数进行操作，并自动选取在哪儿进行计算：

```
julia> r = @spawn rand(2,2)
RemoteRef(1,1,0)

julia> s = @spawn 1 .+ fetch(r)
RemoteRef(1,1,1)

julia> fetch(s)
1.10824216411304866 1.13798233877923116
1.12376292706355074 1.18750497916607167
```

注意，此处用 1 .+ fetch(r) 而不是 1 .+ r 。这是因为我们不知道代码在何处运行，而 fetch 会将需要的 r 移到做加法的处理器上。此例中， @spawn 很聪明，它知道在有 r 对象的处理器上进行计算，因而 fetch 将不做任何操作。

（ @spawn 不是内置函数，而是 Julia 定义的 宏 ）

所有执行程序代码的处理器上，都必须能获得程序代码。例如，输入：

```
julia> function rand2(dims...)
         return 2*rand(dims...)
       end

julia> rand2(2,2)
2x2 Float64 Array:
 0.153756  0.368514
 1.15119   0.918912

julia> @spawn rand2(2,2)
RemoteRef(1,1,1)

julia> @spawn rand2(2,2)
RemoteRef(2,1,2)

julia> exception on 2: in anonymous: rand2 not defined
```

进程 1 知道 rand2 函数，但进程 2 不知道。 require 函数自动在当前所有可用的处理器上载入源文件，使所有的处理器都能运行代码：

```
julia> require("myfile")
```

在集群中，文件（及递归载入的任何文件）的内容会被发送到整个网络。可以使用 @everywhere 宏在所有处理器上执行命令：

```
julia> @everywhere id = myid()

julia> remotecall_fetch(2, ()->id)
2

@everywhere include("defs.jl")
```

A file can also be preloaded on multiple processes at startup, and a driver script can be used to drive the computation:

```
julia -p <n> -L file1.jl -L file2.jl driver.jl
```

Each process has an associated identifier. The process providing the interactive julia prompt always has an id equal to 1, as would the julia process running the driver script in the example above. The processes used by default for parallel operations are referred to as `workers`. When there is only one process, process 1 is considered a worker. Otherwise, workers are considered to be all processes other than process 1.

The base Julia installation has in-built support for two types of clusters:

- A local cluster specified with the `-p` option as shown above.
- A cluster spanning machines using the `--machinefile` option. This uses a passwordless `ssh` login to start julia worker processes (from the same path as the current host) on the specified machines.

Functions `addprocs`, `rmprocs`, `workers`, and others are available as a programmatic means of adding, removing and querying the processes in a cluster.

Other types of clusters can be supported by writing your own custom ClusterManager. See section on ClusterManagers.

数据移动

并行计算中，消息传递和数据移动是最大的开销。减少这两者的数量，对性能至关重要。

`fetch` 是显式的数据移动操作，它直接要求将对象移动到当前机器。`@spawn` （及相关宏）也进行数据移动，但不是显式的，因而被称为隐式数据移动操作。对比如下两种构造随机矩阵并计算其平方的方法：

```
# method 1
A = rand(1000,1000)
Bref = @spawn A^2
...
fetch(Bref)

# method 2
Bref = @spawn rand(1000,1000)^2
...
fetch(Bref)
```

方法 1 中，本地构造了一个随机矩阵，然后将其传递给做平方计算的处理器。方法 2 中，在同一处理器构造随机矩阵并进行平方计算。因此，方法 2 比方法 1 移动的数据少得多。

并行映射和循环

大部分并行计算不需要移动数据。最常见的是蒙特卡罗仿真。下例使用 `@spawn` 在两个处理器上仿真投硬币。先在 `count_heads.jl` 中写如下函数：

```
function count_heads(n)
    c::Int = 0
    for i=1:n
        c += randbool()
    end
    c
end
```

在两台机器上做仿真，最后将结果加起来：

```
require("count_heads")

a = @spawn count_heads(100000000)
b = @spawn count_heads(100000000)
fetch(a)+fetch(b)
```

在多处理器上独立地进行迭代运算，然后用一些函数把它们的结果综合起来。综合的过程称为 约简 。

上例中，我们显式调用了两个 `@spawn` 语句，它将并行计算限制在两个处理器上。要在任意个数的处理器上运行，应使用 并行 *for* 循环 ，它在 Julia 中应写为：

```
nheads = @parallel (+) for i=1:200000000
  int(randbool())
end
```

这个构造实现了给多处理器分配迭代的模式，并且使用特定约简来综合结果（此例中为 `(+)` ）。

注意，尽管并行 for 循环看起来和一组 for 循环差不多，但它们的行为有很大区别。第一，循环不是按顺序进行的。第二，写进变量或数组的值不是全局可见的，因为迭代运行在不同的处理器上。并行循环内使用的所有变量都会被复制、广播到每个处理器。

下列代码并不会按照预想运行：

```
a = zeros(100000)
@parallel for i=1:100000
  a[i] = i
end
```

如果不需要，可以省略约简运算符。但此代码不会初始化 `a` 的所有元素，因为每个处理器上都只有独立的一份儿。应避免类似的并行 for 循环。但是我们可以使用分布式数组来规避这种情形，后面我们会讲。

如果“外部”变量是只读的，就可以在并行循环中使用它：

```
a = randn(1000)
@parallel (+) for i=1:100000
  f(a[randi(end)])
end
```

有时我们不需要约简，仅希望将函数应用到某个范围的整数（或某个集合的元素）上。这时可以使用 并行映射 `pmap` 函数。下例中并行计算几个大随机矩阵的奇异值：

```
M = {rand(1000,1000) for i=1:10}
pmap(svd, M)
```

被调用的函数需处理大量工作时使用 `pmap` ，反之，则使用 `@parallel for` 。

Synchronization With Remote References

Scheduling

Julia's parallel programming platform uses 任务（也称为协程） to switch among multiple computations. Whenever code performs a communication operation like `fetch` or `wait`, the current task is suspended and a scheduler picks another task to run. A task is restarted when the event it is waiting for completes.

For many problems, it is not necessary to think about tasks directly. However, they can be used to wait for multiple events at the same time, which provides for *dynamic scheduling*. In dynamic scheduling, a program decides what to compute or where to compute it based on when other jobs finish. This is needed for unpredictable or unbalanced workloads, where we want to assign more work to processes only when they finish their current tasks.

As an example, consider computing the singular values of matrices of different sizes:

```
M = {rand(800,800), rand(600,600), rand(800,800), rand(600,600)}
pmap(svd, M)
```

If one process handles both 800x800 matrices and another handles both 600x600 matrices, we will not get as much scalability as we could. The solution is to make a local task to "feed" work to each process when it completes its current task. This can be seen in the implementation of `pmap`:

```
function pmap(f, lst)
    np = nprocs()  # determine the number of processes available
    n = length(lst)
    results = cell(n)
    i = 1
    # function to produce the next work item from the queue.
    # in this case it's just an index.
    nextidx() = (idx=i; i+=1; idx)
    @sync begin
        for p=1:np
            if p != myid() || np == 1
                @async begin
                    while true
                        idx = nextidx()
                        if idx > n
                            break
                        end
                        results[idx] = remotecall_fetch(p, f, lst[idx])
                    end
                end
            end
        end
    end
    results
end
```

`@async` is similar to `@spawn`, but only runs tasks on the local process. We use it to create a "feeder" task for each process. Each task picks the next index that needs to be computed, then waits for its process to finish, then repeats until we run out of indexes. Note that the feeder tasks do not begin to execute until the main task reaches the end of the `@sync` block, at which point it surrenders control and waits for all the local tasks to complete before returning from the function. The feeder tasks are able to share state via `nextidx()` because they all run on the same process. No locking is required, since the threads are scheduled cooperatively and not preemptively. This means context switches only occur at well-defined points: in this case, when `remotecall_fetch` is called.

分布式数组

并行计算综合使用多个机器上的内存资源，因而可以使用在一个机器上不能实现的大数组。这时，可使用分布式数组，每个处理器仅对它所拥有的那部分数组进行操作。

分布式数组（或 全局对象 ）逻辑上是个单数组，但它分为很多块儿，每个处理器上保存一块儿。但对整个数组的运算与在本地数组的运算是一样的，并行计算是隐藏的。

分布式数组是用 DArray 类型来实现的。 DArray 的元素类型和维度与 Array 一样。 DArray 的数据的分布，是这样实现的：它把索引空间在每个维度都分成一些小块。

一些常用分布式数组可以使用 d 开头的函数来构造：

```
dzeros(100,100,10)
dones(100,100,10)
drand(100,100,10)
drandn(100,100,10)
dfill(x, 100,100,10)
```

最后一个例子中，数组的元素由值 x 来初始化。这些函数自动选取某个分布。如果要指明使用哪个进程，如何分布数据，应这样写：

```
dzeros((100,100), [1:4], [1,4])
```

The second argument specifies that the array should be created on processors 1 through 4. When dividing data among a large number of processes, one often sees diminishing returns in performance. Placing DArrays on a subset of processes allows multiple DArray computations to happen at once, with a higher ratio of work to communication on each process.

The third argument specifies a distribution; the nth element of this array specifies how many pieces dimension n should be divided into. In this example the first dimension will not be divided, and the second dimension will be divided into 4 pieces. Therefore each local chunk will be of size (100,25). Note that the product of the distribution array must equal the number of processes.

distribute(a::Array) 可用来将本地数组转换为分布式数组。

localpart(a::DArray) 可用来获取 DArray 本地存储的部分。

localindexes(a::DArray) 返回本地进程所存储的维度索引值范围多元组。

convert(Array, a::DArray) 将所有数据综合到本地进程上。

使用索引值范围来索引 DArray （方括号）时，会创建 SubArray 对象，但不复制数据。

构造分布式数组

DArray 的构造函数是 darray ，它的声明如下：

```
DArray(init, dims[, procs, dist])
```

init 函数的参数，是索引值范围多元组。这个函数在本地声名一块分布式数组，并用指定索引值来进行初始化。 dims 是整个分布式数组的维度。 procs 是可选的，指明一个存有要使用的进程 ID 的向量 。 dist 是一个整数向量，指明分布式数组在每个维度应该被分成几块。

最后俩参数是可选的，忽略的时候使用默认值。

下例演示如果将本地数组 fill 的构造函数更改为分布式数组的构造函数：

```
dfill(v, args...) = DArray(I->fill(v, map(length,I)), args...)
```

此例中 `init` 函数仅对它构造的本地块的维度调用 `fill` 。

分布式数组运算

At this time, distributed arrays do not have much functionality. Their major utility is allowing communication to be done via array indexing, which is convenient for many problems. As an example, consider implementing the "life" cellular automaton, where each cell in a grid is updated according to its neighboring cells. To compute a chunk of the result of one iteration, each process needs the immediate neighbor cells of its local chunk. The following code accomplishes this:

```
function life_step(d::DArray)
    DArray(size(d),procs(d)) do I
        top   = mod(first(I[1])-2,size(d,1))+1
        bot   = mod( last(I[1])  ,size(d,1))+1
        left  = mod(first(I[2])-2,size(d,2))+1
        right = mod( last(I[2])  ,size(d,2))+1

        old = Array(Bool, length(I[1])+2, length(I[2])+2)
        old[1      , 1      ] = d[top , left]   # left side
        old[2:end-1, 1      ] = d[I[1], left]
        old[end    , 1      ] = d[bot , left]
        old[1      , 2:end-1] = d[top , I[2]]
        old[2:end-1, 2:end-1] = d[I[1], I[2]]   # middle
        old[end    , 2:end-1] = d[bot , I[2]]
        old[1      , end    ] = d[top , right]  # right side
        old[2:end-1, end    ] = d[I[1], right]
        old[end    , end    ] = d[bot , right]

        life_rule(old)
    end
end
```

As you can see, we use a series of indexing expressions to fetch data into a local array `old`. Note that the `do` block syntax is convenient for passing `init` functions to the `DArray` constructor. Next, the serial function `life_rule` is called to apply the update rules to the data, yielding the needed `DArray` chunk. Nothing about `life_rule` is `DArray`-specific, but we list it here for completeness:

```
function life_rule(old)
    m, n = size(old)
    new = similar(old, m-2, n-2)
    for j = 2:n-1
        for i = 2:m-1
            nc = +(old[i-1,j-1], old[i-1,j], old[i-1,j+1],
                   old[i  ,j-1],             old[i  ,j+1],
                   old[i+1,j-1], old[i+1,j], old[i+1,j+1])
            new[i-1,j-1] = (nc == 3 || nc == 2 && old[i,j])
        end
    end
    new
end
```

Shared Arrays (Experimental, UNIX-only feature)

Shared Arrays use system shared memory to map the same array across many processes. While there are some similarities to a `DArray`, the behavior of a `SharedArray` is quite different. In a `DArray`, each process has local access to just a chunk of the data, and no two processes share the same chunk; in contrast, in a `SharedArray` each "participating" process has access to the entire array. A `SharedArray` is a good choice when you want to have a large amount of data jointly accessible to two or more processes on the same machine.

`SharedArray` indexing (assignment and accessing values) works just as with regular arrays, and is efficient because the underlying memory is available to the local process. Therefore, most algorithms work naturally on `SharedArrays`, albeit in single-process mode. In cases where an algorithm insists on an `Array` input, the underlying array can be retrieved from a `SharedArray` by calling `sdata(S)`. For other `AbstractArray` types, `sdata` just returns the object itself, so it's safe to use `sdata` on any Array-type object.

The constructor for a shared array is of the form:

```
SharedArray(T::Type, dims::NTuple; init=false, pids=Int[])
```

which creates a shared array of a bitstype `T` and size `dims` across the processes specified by `pids`. Unlike distributed arrays, a shared array is accessible only from those participating workers specified by the `pids` named argument (and the creating process too, if it is on the same host).

If an `init` function, of signature `initfn(S::SharedArray)`, is specified, it is called on all the participating workers. You can arrange it so that each worker runs the `init` function on a distinct portion of the array, thereby parallelizing initialization.

Here's a brief example:

```
julia> addprocs(3)
3-element Array{Any,1}:
 2
 3
 4

julia> S = SharedArray(Int, (3,4), init = S -> S[localindexes(S)] = myid())
3x4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  3  4  4

julia> S[3,2] = 7
7

julia> S
3x4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  7  4  4
```

`localindexes` provides disjoint one-dimensional ranges of indexes, and is sometimes convenient for splitting up tasks among processes. You can, of course, divide the work any way you wish:

```
julia> S = SharedArray(Int, (3,4), init = S -> S[myid()-1:nworkers():length(S)] =␣
↪myid())
3x4 SharedArray{Int64,2}:
 2  2  2  2
 3  3  3  3
 4  4  4  4
```

Since all processes have access to the underlying data, you do have to be careful not to set up conflicts. For example:

```
@sync begin
    for p in workers()
        @async begin
            remotecall_wait(p, fill!, S, p)
        end
    end
end
```

would result in undefined behavior: because each process fills the *entire* array with its own `pid`, whichever process is the last to execute (for any particular element of `S`) will have its `pid` retained.

ClusterManagers

Julia worker processes can also be spawned on arbitrary machines, enabling Julia's natural parallelism to function quite transparently in a cluster environment. The `ClusterManager` interface provides a way to specify a means to launch and manage worker processes. For example, `ssh` clusters are also implemented using a `ClusterManager`:

```
immutable SSHManager <: ClusterManager
    launch::Function
    manage::Function
    machines::AbstractVector

    SSHManager(; machines=[]) = new(launch_ssh_workers, manage_ssh_workers, machines)
end

function launch_ssh_workers(cman::SSHManager, np::Integer, config::Dict)
    ...
end

function manage_ssh_workers(id::Integer, config::Dict, op::Symbol)
    ...
end
```

where `launch_ssh_workers` is responsible for instantiating new Julia processes and `manage_ssh_workers` provides a means to manage those processes, e.g. for sending interrupt signals. New processes can then be added at runtime using `addprocs`:

```
addprocs(5, cman=LocalManager())
```

which specifies a number of processes to add and a `ClusterManager` to use for launching those processes.

CHAPTER 21

运行外部程序

Julia 使用倒引号 ` 来运行外部程序:

```
julia> `echo hello`
`echo hello`
```

它有以下几个特性:

- 倒引号并不直接运行程序，它构造一个 Cmd 对象来表示这个命令。可以用这个对象，通过管道将命令连接起来，运行，并进行读写
- 命令运行时，除非指明， Julia 并不捕获输出。它调用 libc 的 system ，命令的输出默认指向 stdout 。
- 命令运行不需要 shell 。 Julia 直接解析命令语法，对变量内插，像 shell 一样分隔单词，它遵循 shell 引用语法。命令调用 fork 和 exec 函数，作为 julia 的直接子进程。

下面是运行外部程序的例子:

```
julia> run(`echo hello`)
hello
```

hello 是 echo 命令的输出，它被送到标准输出。 run 方法本身返回 nothing 。如果外部命令没有正确运行，将抛出 ErrorException 异常。

使用 readall 读取命令的输出:

```
julia> a=readall(`echo hello`)
"hello\n"

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

More generally, you can use open to read from or write to an external command. For example:

```
julia> open(`less`, "w", STDOUT) do io
           for i = 1:1000
```

```
            println(io, i)
        end
    end
```

内插

将文件名赋给变量 `file`，将其作为命令的参数。像在字符串文本中一样使用 `$` 做内插（详见 字符串）：

```
julia> file = "/etc/passwd"
"/etc/passwd"

julia> `sort $file`
`sort /etc/passwd`
```

如果文件名有特殊字符，比如 `/Volumes/External HD/data.csv`，会如下显示：

```
julia> file = "/Volumes/External HD/data.csv"
"/Volumes/External HD/data.csv"

julia> `sort $file`
`sort '/Volumes/External HD/data.csv'`
```

文件名被单引号引起来了。Julia 知道 `file` 会被当做一个单变量进行内插，它自动把内容引了起来。事实上，这也不准确： `file` 的值并不会被 shell 解释，所以不需要真正的引起来；此处把它引起来，只是为了给用户显示。下例也可以正常运行：

```
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'`
```

如果 要 内插多个单词，应使用数组（或其它可迭代容器）：

```
julia> files = ["/etc/passwd","/Volumes/External HD/data.csv"]
2-element ASCIIString Array:
 "/etc/passwd"
 "/Volumes/External HD/data.csv"


julia> `grep foo $files`
`grep foo /etc/passwd '/Volumes/External HD/data.csv'`
```

如果数组内插为 shell 单词的一部分，Julia 会模仿 shell 的 `{a,b,c}` 参数生成的行为：

```
julia> names = ["foo","bar","baz"]
3-element ASCIIString Array:
 "foo"
 "bar"
```

```
 "baz"

julia> `grep xylophone $names.txt`
`grep xylophone foo.txt bar.txt baz.txt`
```

如果将多个数组内插进同一个单词，Julia 会模仿 shell 的笛卡尔乘积生成的行为：

```
julia> names = ["foo","bar","baz"]
3-element ASCIIString Array:
 "foo"
 "bar"
 "baz"

julia> exts = ["aux","log"]
2-element ASCIIString Array:
 "aux"
 "log"

julia> `rm -f $names.$exts`
`rm -f foo.aux foo.log bar.aux bar.log baz.aux baz.log`
```

不构造临时数组对象，直接内插文本化数组：

```
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`
```

引用

命令复杂时，有时需要使用引号。来看一个 perl 的命令：

```
sh$ perl -le '$|=1; for (0..3) { print }'
0
1
2
3
```

再看个使用双引号的命令：

```
sh$ first="A"
sh$ second="B"
sh$ perl -le '$|=1; print for @ARGV' "1: $first" "2: $second"
1: A
2: B
```

一般来说，Julia 的倒引号语法支持将 shell 命令原封不动的复制粘贴进来，且转义、引用、内插等行为可以原封不动地正常工作。唯一的区别是，内插被集成进了 Julia 中：

```
julia> `perl -le '$|=1; for (0..3) { print }'`
`perl -le '$|=1; for (0..3) { print }'`

julia> run(ans)
0
1
2
```

```
3

julia> first = "A"; second = "B";

julia> `perl -le 'print for @ARGV' "1: $first" "2: $second"`
`perl -le 'print for @ARGV' '1: A' '2: B'`

julia> run(ans)
1: A
2: B
```

当需要在 Julia 中运行 shell 命令时，先试试复制粘贴。Julia 会先显示出来命令，可以据此检查内插是否正确，再去运行命令。

管道

Shell 元字符，如 |, &, 及 > 在 Julia 倒引号语法中并是不特殊字符。倒引号中的管道符仅仅是文本化的管道字符 “|” 而已：

```
julia> run(`echo hello | sort`)
hello | sort
```

在 Julia 中要想构造管道，应在 Cmd 间使用 |> 运算符：

```
julia> run(`echo hello` |> `sort`)
hello
```

继续看个例子：

```
julia> run(`cut -d: -f3 /etc/passwd` |> `sort -n` |> `tail -n5`)
210
211
212
213
214
```

它打印 UNIX 系统五个最高级用户的 ID 。 cut, sort 和 tail 命令都作为当前 julia 进程的直接子进程运行，shell 进程没有介入。 Julia 自己来设置管道并连接文件描述符, 这些工作通常由 shell 来完成。也 因此, Julia 可以对子进程实现更好的控制, 也可以实现 shell 不能实现的一 些功能. 值得注意的是, |> 仅仅是重定向了 stdout. 使用 .> 来 重定向 stderr.

Julia 可以并行运行多个命令：

```
julia> run(`echo hello` & `echo world`)
world
hello
```

输出顺序是非确定性的。两个 echo 进程几乎同时开始，它们竞争 stdout 描述符的写操作，这个描述符被两个进程和 julia 进程所共有。使用管道，可将这些进程的输出传递给其它程序：

```
julia> run(`echo world` & `echo hello` |> `sort`)
hello
world
```

来看一个复杂的使用 Julia 来调用 perl 命令的例子：

```
julia> prefixer(prefix, sleep) = `perl -nle '$|=1; print "'$prefix' ", $_; sleep '
↪$sleep';'`

julia> run(`perl -le '$|=1; for(0..9){ print; sleep 1 }'` |> prefixer("A",2) &␣
↪prefixer("B",2))
A   0
B   1
A   2
B   3
A   4
B   5
A   6
B   7
A   8
B   9
```

这是一个单生产者双并发消费者的经典例子：一个 `perl` 进程生产从 0 至 9 的 10 行数，两个并行的进程消费这些结果，其中一个给结果加前缀 "A"，另一个加前缀 "B"。我们不知道哪个消费者先消费第一行，但一旦开始，两个进程交替消费这些行。（在 Perl 中设置 `$|=1` ，可使打印表达式先清空 `stdout` 句柄；否则输出会被缓存并立即打印给管道，结果将只有一个消费者进程在读取。）

再看个更复杂的多步的生产者-消费者的例子：

```
julia> run(`perl -le '$|=1; for(0..9){ print; sleep 1 }'` |>
           prefixer("X",3) & prefixer("Y",3) & prefixer("Z",3) |>
           prefixer("A",2) & prefixer("B",2))
B   Y   0
A   Z   1
B   X   2
A   Y   3
B   Z   4
A   X   5
B   Y   6
A   Z   7
B   X   8
A   Y   9
```

此例和前例类似，单有消费者分两步，且两步的延迟不同。

强烈建议你亲手试试这些例子，看看它们是如何运行的。

CHAPTER 22

调用 C 和 Fortran 代码

Julia 调用 C 和 Fortran 的函数，既简单又高效。

被调用的代码应该是共享库的格式。大多数 C 和 Fortran 库都已经被编译为共享库。如果自己使用 GCC （或 Clang ）编译代码，需要添加 `-shared` 和 `-fPIC` 选项。Julia 调用这些库的开销与本地 C 语言相同。

调用共享库和函数时使用多元组形式： `(:function, "library")` 或 `("function", "library")` ，其中 `function` 是 C 的导出函数名， `library` 是共享库名。共享库依据名字来解析，路径由环境变量来确定，有时需要直接指明。

多元组内有时仅有函数名（仅 `:function` 或 `"function"` ）。此时，函数名由当前进程解析。这种形式可以用来调用 C 库函数， Julia 运行时函数，及链接到 Julia 的应用中的函数。

使用 `ccall` 来生成库函数调用。 `ccall` 的参数如下:

1. (:function, "library") 多元组对儿（必须为常量，详见下面）
2. 返回类型，可以为任意的位类型，包括 `Int32` ， `Int64` ， `Float64` ，或者指向任意类型参数 `T` 的指针 `Ptr{T}` ，或者仅仅是指向无类型指针 `void*` 的 `Ptr`
3. 输入的类型的多元组，与上述的返回类型的要求类似。输入必须是多元组，而不是值为多元组的变量或表达式
4. 后面的参数，如果有的话，都是被调用函数的实参

下例调用标准 C 库中的 `clock` :

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

julia> t
2292761

julia> typeof(ans)
Int32
```

`clock` 函数没有参数，返回 `Int32` 类型。输入的类型如果只有一个，常写成一元多元组，在后面跟一逗号。例如要调用 `getenv` 函数取得指向一个环境变量的指针，应这样调用:

```
julia> path = ccall( (:getenv, "libc"), Ptr{Uint8}, (Ptr{Uint8},), "SHELL")
Ptr{Uint8} @0x00007fff5fbffc45

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

注意，类型多元组的参数必须写成 `(Ptr{Uint8},)` ，而不是 `(Ptr{Uint8})` 。这是因为 `(Ptr{Uint8})` 等价于 `Ptr{Uint8}` ，它并不是一个包含 `Ptr{Uint8}` 的一元多元组：

```
julia> (Ptr{Uint8})
Ptr{Uint8}

julia> (Ptr{Uint8},)
(Ptr{Uint8},)
```

实际中要提供可复用代码时，通常要使用 Julia 的函数来封装 `ccall` ，设置参数，然后检查 C 或 Fortran 函数中可能出现的任何错误，将其作为异常传递给 Julia 的函数调用者。下例中， `getenv` C 库函数被封装在 env.jl 里的 Julia 函数中：

```
function getenv(var::String)
  val = ccall( (:getenv, "libc"),
               Ptr{Uint8}, (Ptr{Uint8},), var)
  if val == C_NULL
    error("getenv: undefined variable: ", var)
  end
  bytestring(val)
end
```

上例中，如果函数调用者试图读取一个不存在的环境变量，封装将抛出异常：

```
julia> getenv("SHELL")
"/bin/bash"

julia> getenv("FOOBAR")
getenv: undefined variable: FOOBAR
```

下例稍复杂些，显示本地机器的主机名：

```
function gethostname()
  hostname = Array(Uint8, 128)
  ccall( (:gethostname, "libc"), Int32,
         (Ptr{Uint8}, Uint),
         hostname, length(hostname))
  return bytestring(convert(Ptr{Uint8}, hostname))
end
```

此例先分配出一个字节数组，然后调用 C 库函数 `gethostname` 向数组中填充主机名，取得指向主机名缓冲区的指针，在默认其为空结尾 C 字符串的前提下，将其转换为 Julia 字符串。 C 库函数一般都用这种方式从函数调用者那儿，将申请的内存传递给被调用者，然后填充。在 Julia 中分配内存，通常都需要通过构建非初始化数组，然后将指向数据的指针传递给 C 函数。

调用 Fortran 函数时，所有的输入都必须通过引用来传递。

`&` 前缀说明传递的是指向标量参数的指针，而不是标量值本身。下例使用 BLAS 函数计算点积：

```
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,
                   (Ptr{Int32}, Ptr{Float64}, Ptr{Int32}, Ptr{Float64}, Ptr{Int32}),
                   &n, DX, &incx, DY, &incy)
  return product
end
```

前缀 `&` 的意思与 C 中的不同。对引用的变量的任何更改，都是对 Julia 不可见的。 `&` 并不是真正的地址运算符，可以在任何语法中使用它，例如 `&0` 和 `&f(x)` 。

注意在处理过程中，C 的头文件可以放在任何地方。目前还不能将 Julia 的结构和其他非基础类型传递给 C 库。通过传递指针来生成、使用非透明结构类型的 C 函数，可以向 Julia 返回 `Ptr{Void}` 类型的值，这个值以 `Ptr{Void}` 的形式被其它 C 函数调用。可以像任何 C 程序一样，通过调用库中对应的程序，对对象进行内存分配和释放。

把 C 类型映射到 Julia

Julia 自动调用 `convert` 函数，将参数转换为指定类型。例如：

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

会按如下操作：

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

如果标量值与 `&` 一起被传递作为 `Ptr{T}` 类型的参数时，值首先会被转换为 `T` 类型。

数组转换

把数组作为一个 `Ptr{T}` 参数传递给 C 时，它不进行转换。Julia 仅检查元素类型是否为 `T` ，然后传递首元素的地址。这样做可以避免不必要的复制整个数组。

因此，如果 `Array` 中的数据格式不对时，要使用显式转换，如 `int32(a)` 。

如果想把数组 不经转换 而作为一个不同类型的指针传递时，要么声明参数为 `Ptr{Void}` 类型，要么显式调用 `convert(Ptr{T}, pointer(A))` 。

类型相关

基础的 C/C++ 类型和 Julia 类型对照如下。每个 C 类型也有一个对应名称的 Julia 类型，不过冠以了前缀 C 。这有助于编写简便的代码（但 C 中的 int 与 Julia 中的 Int 不同） 。

与系统无关：

unsigned char	Cuchar	Uint8
short	Cshort	Int16
unsigned short	Cushort	Uint16
int	Cint	Int32
unsigned int	Cuint	Uint32
long long	Clonglong	Int64
unsigned long long	Culonglong	Uint64
intmax_t	Cintmax_t	Int64
uintmax_t	Cuintmax_t	Uint64
float	Cfloat	Float32
double	Cdouble	Float64
ptrdiff_t	Cptrdiff_t	Int
ssize_t	Cssize_t	Int
size_t	Csize_t	Uint
void		Void
void*		Ptr{Void}
char* (or char[], e.g. a string)		Ptr{Uint8}
char** (or *char[])		Ptr{Ptr{Uint8}}
struct T* (where T represents an appropriately defined bits type)		Ptr{T} (call using &variable_name in the parameter list)
struct T (where T represents an appropriately defined bits type)		T (call using &variable_name in the parameter list)
jl_value_t* (any Julia Type)		Ptr{Any}

Julia 的 Char 类型是 32 位的，与所有平台的宽字符类型 (wchar_t 或 wint_t) 不同。

返回 void 的 C 函数，在 Julia 中返回 nothing 。

与系统有关：

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)

对应于字符串参数（ char* ）的 Julia 类型为 Ptr{Uint8} ，而不是 ASCIIString 。参数中有 char** 类型的 C 函数，在 Julia 中调用时应使用 Ptr{Ptr{Uint8}} 类型。例如，C 函数：

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

在 Julia 中应该这样调用：

```
argv = [ "a.out", "arg1", "arg2" ]
ccall(:main, Int32, (Int32, Ptr{Ptr{Uint8}}), length(argv), argv)
```

For wchar_t* arguments, the Julia type should be Ptr{Wchar_t}, and data can be converted to/from ordinary Julia strings by the wstring(s) function (equivalent to either utf16(s) or utf32(s) depending upon the width of Cwchar_t. Note also that ASCII, UTF-8, UTF-16, and UTF-32 string data in Julia is internally NUL-terminated, so it can be passed to C functions expecting NUL-terminated data without making a copy.

通过指针读取数据

下列方法是“不安全”的，因为坏指针或类型声明可能会导致意外终止或损坏任意进程内存。

指定 Ptr{T}，常使用 unsafe_ref(ptr, [index]) 方法，将类型为 T 的内容从所引用的内存复制到 Julia 对象中。 index 参数是可选的（默认为 1），它是从 1 开始的索引值。此函数类似于 getindex() 和 setindex!() 的行为（如 [] 语法）。

返回值是一个被初始化的新对象，它包含被引用内存内容的浅拷贝。被引用的内存可安全释放。

如果 T 是 Any 类型，被引用的内存会被认为包含对 Julia 对象 jl_value_t* 的引用，结果为这个对象的引用，且此对象不会被拷贝。需要谨慎确保对象始终对垃圾回收机制可见（指针不重要，重要的是新的引用），来确保内存不会过早释放。注意，如果内存原本不是由 Julia 申请的，新对象将永远不会被 Julia 的垃圾回收机制释放。如果 Ptr 本身就是 jl_value_t*，可使用 unsafe_pointer_to_objref(ptr) 将其转换回 Julia 对象引用。（可通过调用 pointer_from_objref(v) 将Julia 值 v 转换为 jl_value_t* 指针 Ptr{Void}。）

逆操作（向 Ptr{T} 写数据）可通过 unsafe_store!(ptr, value, [index]) 来实现。目前，仅支持位类型和其它无指针（ isbits ）不可变类型。

现在任何抛出异常的操作，估摸着都是还没实现完呢。来写个帖子上报 bug 吧，就会有人来解决啦。

如果所关注的指针是（位类型或不可变）的目标数据数组， pointer_to_array(ptr,dims,[own]) 函数就非常有用啦。如果想要 Julia “控制”底层缓冲区并在返回的 Array 被释放时调用 free(ptr)，最后一个参数应该为真。如果省略 own 参数或它为假，则调用者需确保缓冲区一直存在，直至所有的读取都结束。

Ptr 的算术(比如 +) 和 C 的指针算术不同， 对 Ptr 加一个整数会将指针移动一段距离的 字节 ， 而不是元素。这样从指针运算上得到的地址不会依赖指针类型。

用指针传递修改值

因为 C 不支持多返回值， 所以通常 C 函数会用指针来修改值。 在 ccall 里完成这些需要把值放在适当类型的数组里。当你用 Ptr 传递整个数组时， Julia 会自动传递一个 C 指针到被这个值:

```
width = Cint[0]
range = Cfloat[0]
ccall(:foo, Void, (Ptr{Cint}, Ptr{Cfloat}), width, range)
```

这被广泛用在了 Julia 的 LAPACK 接口上， 其中整数类型的 info 被以引用的方式传到 LAPACK， 再返回是否成功。

垃圾回收机制的安全

给 ccall 传递数据时，最好避免使用 pointer() 函数。应当定义一个转换方法，将变量直接传递给 ccall 。ccall 会自动安排，使得在调用返回前，它的所有参数都不会被垃圾回收机制处理。如果 C API 要存储一个由 Julia 分配好的内存的引用，当 ccall 返回后，需要自己设置，使对象对垃圾回收机制保持可见。推荐的方法为，在一个类型为 Array{Any,1} 的全局变量中保存这些值，直到 C 接口通知它已经处理完了。

只要构造了指向 Julia 数据的指针，就必须保证原始数据直至指针使用完之前一直存在。Julia 中的许多方法，如 unsafe_ref() 和 bytestring() ，都复制数据而不是控制缓冲区，因此可以安全释放（或修改）原始数据，不会影响到 Julia 。有一个例外需要注意，由于性能的原因， pointer_to_array() 会共享（或控制）底层缓冲区。

垃圾回收并不能保证回收的顺序。例如，当 a 包含对 b 的引用，且两者都要被垃圾回收时，不能保证 b 在 a 之后被回收。这需要用其它方式来处理。

非常量函数说明

(name, library) 函数说明应为常量表达式。可以通过 eval ，将计算结果作为函数名：

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

表达式用 string 构造名字，然后将名字代入 ccall 表达式进行计算。注意 eval 仅在顶层运行，因此在表达式之内，不能使用本地变量（除非本地变量的值使用 $ 进行过内插）。 eval 通常用来作为顶层定义，例如，将包含多个相似函数的库封装在一起。

间接调用

ccall 的第一个参数可以是运行时求值的表达式。此时，表达式的值应为 Ptr 类型，指向要调用的原生函数的地址。这个特性用于 ccall 的第一参数包含对非常量（本地变量或函数参数）的引用时。

调用方式

ccall 的第二个（可选）参数指定调用方式（在返回值之前）。如果没指定，将会使用操作系统的默认 C 调用方式。其它支持的调用方式为: stdcall , cdecl , fastcall 和 thiscall 。例如 (来自 base/libc.jl)：

```
hn = Array(Uint8, 256)
err=ccall(:gethostname, stdcall, Int32, (Ptr{Uint8}, Uint32), hn, length(hn))
```

更多信息请参考 LLVM Language Reference.

Accessing Global Variables

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.

Passing Julia Callback Functions to C

It is possible to pass Julia functions to native functions that accept function pointer arguments. A classic example is the standard C library qsort function, declared as:

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

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 1d 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 worry about calling `qsort` and passing arguments, we need to write a comparison function that works for some arbitrary type T:

```
function mycompare{T}(a_::Ptr{T}, b_::Ptr{T})
    a = unsafe_load(a_)
    b = unsafe_load(b_)
    return convert(Cint, a < b ? -1 : a > b ? +1 : 0)
end
```

Notice that we have to be careful about the return type: `qsort` expects a function returning a C `int`, so we must be sure to return `Cint` via a call to `convert`.

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

```
const mycompare_c = cfunction(mycompare, Cint, (Ptr{Cdouble}, Ptr{Cdouble}))
```

`cfunction` accepts three arguments: the Julia function (`mycompare`), the return type (`Cint`), and a tuple of the argument types, in this case to sort an array of `Cdouble` (Float64) elements.

The final call to `qsort` looks like this:

```
A = [1.3, -2.7, 4.4, 3.1]
ccall(:qsort, Void, (Ptr{Cdouble}, Csize_t, Csize_t, Ptr{Void}),
      A, length(A), sizeof(eltype(A)), mycompare_c)
```

After this executes, `A` is changed to the sorted array `[ -2.7, 1.3, 3.1, 4.4]`. Note that Julia knows how to convert an array into a `Ptr{Cdouble}`, how to compute the size of a type in bytes (identical to C's `sizeof` operator), 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).

Thread-safety

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. Your callback needs to be written to take two inputs (which you'll most likely just discard) and then wrapped by `SingleAsyncWork`:

```
cb = Base.SingleAsyncWork(data -> my_real_callback(args))
```

The callback you pass to C should only execute a `ccall` to `:uv_async_send`, passing `cb.handle` as the argument.

More About Callbacks

For more details on how to pass callbacks to C libraries, see this blog post.

C++

Cpp 和 Clang 扩展包提供了有限的 C++ 支持。

处理不同平台

当处理不同的平台库的时候，经常要针对特殊平台提供特殊函数。这时常用到变量 `OS_NAME` 。此外，还有一些常用的宏： `@windows`, `@unix`, `@linux`, 及 `@osx` 。注意， linux 和 osx 是 unix 的不相交的子集。宏的用法类似于三元条件运算符。

简单的调用：

```
ccall( (@windows? :_fopen : :fopen), ...)
```

复杂的调用：

```
@linux? (
          begin
              some_complicated_thing(a)
          end
        : begin
              some_different_thing(a)
          end
        )
```

链式调用（圆括号可以省略，但为了可读性，最好加上）：

```
@windows? :a : (@osx? :b : :c)
```

CHAPTER 23

嵌入式 Julia

我们已经知道 (调用 *C* 和 *Fortran* 代码) Julia 可以用简单有效的方式调用 C 函数。但是有很多情况下正好相反:需要从C 调用 Julia 函数。这可以把 Julia 代码整合到更大型的 C/C++ 项目中去， 而不需要重新把所有都用C/C++写一遍。 Julia提供了给C的API来实现这一点。正如大多数语言都有方法调用 C 函数一样， Julia的 API 也可以用于搭建和其他语言之间的桥梁。

高级嵌入

我们从一个简单的C程序入手，它初始化Julia并且调用一些Julia的代码:

```
#include <julia.h>

int main(int argc, char *argv[])
{
    jl_init(NULL);
    JL_SET_STACK_BASE;

    jl_eval_string("print(sqrt(2.0))");

    return 0;
}
```

编译这个程序你需要把 Julia的头文件包含在路径内并且链接函数库 `libjulia`。 比方说 Julia安装在 `$JULIA_DIR`， 就可以用gcc编译:

```
gcc -o test -I$JULIA_DIR/include/julia -L$JULIA_DIR/usr/lib -ljulia test.c
```

或者可以看看 Julia 源码里 `example/` 下的 `embedding.c`。

调用Julia函数之前要先初始化Julia， 可以用 `jl_init` 完成，这个函数的参数是Julia安装路径，类型是 `const char*` 。如果没有任何参数，Julia会自动寻找Julia的安装路径。

The second statement initializes Julia's task scheduling system. This statement must appear in a function that will not return as long as calls into Julia will be made (`main` works fine). Strictly speaking, this statement is optional, but

operations that switch tasks will cause problems if it is omitted.

The third statement in the test program evaluates a Julia statement using a call to `jl_eval_string`.

类型转换

Real applications will not just need to execute expressions, but also return their values to the host program. `jl_eval_string` returns a `jl_value_t*`, which is a pointer to a heap-allocated Julia object. Storing simple data types like `Float64` in this way is called `boxing`, and extracting the stored primitive data is called `unboxing`. Our improved sample program that calculates the square root of 2 in Julia and reads back the result in C looks as follows:

```
jl_value_t *ret = jl_eval_string("sqrt(2.0)");

if (jl_is_float64(ret)) {
    double ret_unboxed = jl_unbox_float64(ret);
    printf("sqrt(2.0) in C: %e \n", ret_unboxed);
}
```

In order to check whether `ret` is of a specific Julia type, we can use the `jl_is_...` functions. By typing `typeof(sqrt(2.0))` into the Julia shell we can see that the return type is `Float64` (`double` in C). To convert the boxed Julia value into a C double the `jl_unbox_float64` function is used in the above code snippet.

Corresponding `jl_box_...` functions are used to convert the other way:

```
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);
```

As we will see next, boxing is required to call Julia functions with specific arguments.

调用 Julia 的函数

While `jl_eval_string` allows C to obtain the result of a Julia expression, it does not allow passing arguments computed in C to Julia. For this you will need to invoke Julia functions directly, using `jl_call`:

```
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);
```

In the first step, a handle to the Julia function `sqrt` is retrieved by calling `jl_get_function`. The first argument passed to `jl_get_function` is a pointer to the `Base` module in which `sqrt` is defined. Then, the double value is boxed using `jl_box_float64`. Finally, in the last step, the function is called using `jl_call1`. `jl_call0`, `jl_call2`, and `jl_call3` functions also exist, to conveniently handle different numbers of arguments. To pass more arguments, use `jl_call`:

```
jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t nargs)
```

Its second argument `args` is an array of `jl_value_t*` arguments and `nargs` is the number of arguments.

内存管理

..As we have seen, Julia objects are represented in C as pointers. This raises the question of who is responsible for freeing these objects.: 正如我们看到的，Julia 的对象在 C 中是以指针形式呈现的。而这也就给出了一个问题：由谁来负责释放这些对象对应的内存呢?

..Typically, Julia objects are freed by a garbage collector (GC), but the GC does not automatically know that we are holding a reference to a Julia value from C. This means the GC can free objects out from under you, rendering pointers invalid.: 一般情况下，Julia 的对象由垃圾回收机制来释放，但垃圾回收机制并不能自动获知我们正在 C 中使用 Julia 对象的引用。这意味着垃圾回收机制可能会释放我们正在使用的对象，造成该指针失效。

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 hold a reference to a Julia 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` is working on the stack, so it must be exactly paired with a `JL_GC_POP` before the stack frame is destroyed.

Several Julia values can be pushed at once using the `JL_GC_PUSH2` , `JL_GC_PUSH3` , and `JL_GC_PUSH4` 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();
```

控制垃圾回收

..There are some functions to control the GC. In normal use cases, these should not be necessary.: 有一些函数可以帮助控制垃圾回收。在一般情况下，它们都不需要被用到。

`void jl_gc_collect()`	Force a GC run
`void jl_gc_disable()`	Disable the GC
`void jl_gc_enable()`	Enable the GC

处理数组

Julia and C can share array data without copying. The next example will show how this works.

Julia arrays are represented in C by the datatype `jl_array_t*`. Basically, `jl_array_t` is a struct that contains:

- Information about the datatype
- A pointer to the data block
- Information about the sizes of the array

To keep things simple, we start with a 1D array. Creating an array containing Float64 elements of length 10 is done by:

```
jl_value_t* array_type = jl_apply_array_type(jl_float64_type, 1);
jl_array_t* x          = jl_alloc_array_1d(array_type, 10);
```

Alternatively, if you have already allocated the array you can generate a thin wrapper around its data:

```
double *existingArray = (double*)malloc(sizeof(double)*10);
jl_array_t *x = jl_ptr_to_array_1d(array_type, existingArray, 10, 0);
```

The last argument is a boolean indicating whether Julia should take ownership of the data. If this argument is non-zero, the GC will call `free` on the data pointer when the array is no longer referenced.

In order to access the data of x, we can use `jl_array_data`:

```
double *xData = (double*)jl_array_data(x);
```

Now we can fill the array:

```
for(size_t i=0; i<jl_array_len(x); i++)
    xData[i] = i;
```

Now let us call a Julia function that performs an in-place operation on `x`:

```
jl_function_t *func  = jl_get_function(jl_base_module, "reverse!");
jl_call1(func, (jl_value_t*)x);
```

By printing the array, one can verify that the elements of `x` are now reversed.

访问返回的数组

If a Julia function returns an array, the return value of `jl_eval_string` and `jl_call` can be cast to a `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);
```

Now the content of `y` can be accessed as before using `jl_array_data`. As always, be sure to keep a reference to the array while it is in use.

高维数组

Julia's multidimensional arrays are stored in memory in column-major order. Here is some code that creates a 2D array and accesses its properties:

```
// 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;
```

Notice that while Julia arrays use 1-based indexing, the C API uses 0-based indexing (for example in calling `jl_array_dim`) in order to read as idiomatic C code.

异常

Julia code can throw exceptions. For example, consider:

```
jl_eval_string("this_function_does_not_exist()");
```

This call will appear to do nothing. However, it is possible to check whether an exception was thrown:

```
if (jl_exception_occurred())
    printf("%s \n", jl_typeof_str(jl_exception_occurred()));
```

If you are using the Julia C API from a language that supports exceptions (e.g. Python, C#, C++), it makes sense to wrap each call into libjulia with a function that checks whether an exception was thrown, and then rethrows the exception in the host language.

抛出 Julia 异常

When writing Julia callable functions, it might be necessary to validate arguments and throw exceptions to indicate errors. A typical type check looks like:

```
if (!jl_is_float64(val)) {
    jl_type_error(function_name, (jl_value_t*)jl_float64_type, val);
}
```

General exceptions can be raised using the funtions:

```
void jl_error(const char *str);
void jl_errorf(const char *fmt, ...);
```

`jl_error` takes a C string, and `jl_errorf` is called like `printf`:

```
jl_errorf("argument x = %d is too large", x);
```

where in this example `x` is assumed to be an integer.

CHAPTER 24

扩展包

Julia 内置了一个包管理系统，可以用这个系统来完成包的管理，当然，你也可以用你的操作系统自带的，或者从源码编译。 你可以在 http://pkg.julialang.org 找到所有已注册（一种发布包的机制）的包的列表。 所有的包管理命令都包含在 `Pkg` 这个module里面，Julia的 Base install 引入了 `Pkg` 。

扩展包状态

可以通过 `Pkg.status()` 这个方程，打印出一个你所有安装的包的总结。

刚开始的时候，你没有安装任何包:

```
julia> Pkg.status()
INFO: Initializing package repository /Users/stefan/.julia/v0.3
INFO: Cloning METADATA from git://github.com/JuliaLang/METADATA.jl
No packages installed.
```

当你第一次运行 `Pkg` 的一个命令时， 你的包目录（所有的包被安装在一个统一的目录下）会自动被初始化，因为 `Pkg` 希望有这样一个目录，这个目录的信息被包含于 `Pkg.status()` 中。

这里是一个简单的，已经有少量被安装的包的例子:

```
julia> Pkg.status()
Required packages:
 - Distributions                 0.2.8
 - UTF16                         0.2.0
Additional packages:
 - NumericExtensions             0.2.17
 - Stats                         0.2.6
```

这些包，都是已注册了的版本，并且通过 `Pkg` 管理。 安装了的包可以是一个更复杂的"状态"，通过"注释"来表明正确的版本；当我们遇到这些"状态"和"注释"时我们会解释的。

为了编程需要，`Pkg.installed()` 返回一个字典，这个字典对应了安装了的包的名字和其现在使用的版本:

```
julia> Pkg.installed()
["Distributions"=>v"0.2.8","Stats"=>v"0.2.6","UTF16"=>v"0.2.0","NumericExtensions"=>v
↪"0.2.17"]
```

添加和删除扩展包

Julia's package manager is a little unusual in that it is declarative rather than imperative. This means that you tell it what you want and it figures out what versions to install (or remove) to satisfy those requirements optimally – and minimally. So rather than installing a package, you just add it to the list of requirements and then "resolve" what needs to be installed. In particular, this means that if some package had been installed because it was needed by a previous version of something you wanted, and a newer version doesn't have that requirement anymore, updating will actually remove that package.

Your package requirements are in the file `~/.julia/v0.3/REQUIRE`. You can edit this file by hand and then call `Pkg.resolve()` to install, upgrade or remove packages to optimally satisfy the requirements, or you can do `Pkg.edit()`, which will open `REQUIRE` in your editor (configured via the `EDITOR` or `VISUAL` environment variables), and then automatically call `Pkg.resolve()` afterwards if necessary. If you only want to add or remove the requirement for a single package, you can also use the non-interactive `Pkg.add` and `Pkg.rm` commands, which add or remove a single requirement to `REQUIRE` and then call `Pkg.resolve()`.

You can add a package to the list of requirements with the `Pkg.add` function, and the package and all the packages that it depends on will be installed:

```
julia> Pkg.status()
No packages installed.

julia> Pkg.add("Distributions")
INFO: Cloning cache of Distributions from git://github.com/JuliaStats/Distributions.
↪jl.git
INFO: Cloning cache of NumericExtensions from git://github.com/lindahua/
↪NumericExtensions.jl.git
INFO: Cloning cache of Stats from git://github.com/JuliaStats/Stats.jl.git
INFO: Installing Distributions v0.2.7
INFO: Installing NumericExtensions v0.2.17
INFO: Installing Stats v0.2.6
INFO: REQUIRE updated.

julia> Pkg.status()
Required packages:
 - Distributions                 0.2.7
Additional packages:
 - NumericExtensions             0.2.17
 - Stats                         0.2.6
```

What this is doing is first adding `Distributions` to your `~/.julia/v0.3/REQUIRE` file:

```
$ cat ~/.julia/v0.3/REQUIRE
Distributions
```

It then runs `Pkg.resolve()` using these new requirements, which leads to the conclusion that the `Distributions` package should be installed since it is required but not installed. As stated before, you can accomplish the same thing by editing your `~/.julia/v0.3/REQUIRE` file by hand and then running `Pkg.resolve()` yourself:

```
$ echo UTF16 >> ~/.julia/v0.3/REQUIRE

julia> Pkg.resolve()
INFO: Cloning cache of UTF16 from git://github.com/nolta/UTF16.jl.git
INFO: Installing UTF16 v0.2.0

julia> Pkg.status()
Required packages:
 - Distributions                 0.2.7
 - UTF16                         0.2.0
Additional packages:
 - NumericExtensions             0.2.17
 - Stats                         0.2.6
```

This is functionally equivalent to calling `Pkg.add("UTF16")`, except that `Pkg.add` doesn't change `REQUIRE` until *after* installation has completed, so if there are problems, `REQUIRE` will be left as it was before calling `Pkg.add`. The format of the `REQUIRE` file is described in *Requirements Specification*; it allows, among other things, requiring specific ranges of versions of packages.

When you decide that you don't want to have a package around any more, you can use `Pkg.rm` to remove the requirement for it from the `REQUIRE` file:

```
julia> Pkg.rm("Distributions")
INFO: Removing Distributions v0.2.7
INFO: Removing Stats v0.2.6
INFO: Removing NumericExtensions v0.2.17
INFO: REQUIRE updated.

julia> Pkg.status()
Required packages:
 - UTF16                         0.2.0

julia> Pkg.rm("UTF16")
INFO: Removing UTF16 v0.2.0
INFO: REQUIRE updated.

julia> Pkg.status()
No packages installed.
```

Once again, this is equivalent to editing the `REQUIRE` file to remove the line with each package name on it then running `Pkg.resolve()` to update the set of installed packages to match. While `Pkg.add` and `Pkg.rm` are convenient for adding and removing requirements for a single package, when you want to add or remove multiple packages, you can call `Pkg.edit()` to manually change the contents of `REQUIRE` and then update your packages accordingly. `Pkg.edit()` does not roll back the contents of `REQUIRE` if `Pkg.resolve()` fails – rather, you have to run `Pkg.edit()` again to fix the files contents yourself.

Because the package manager uses git internally to manage the package git repositories, users may run into protocol issues (if behind a firewall, for example), when running `Pkg.add`. The following command can be run from the command line to tell git to use 'https' instead of the 'git' protocol when cloning repositories:

```
git config --global url."https://".insteadOf git://
```

安装未注册的扩展包

Julia packages are simply git repositories, clonable via any of the protocols that git supports, and containing Julia code that follows certain layout conventions. Official Julia packages are registered in the METADATA.jl repository, available at a well-known location[1]. The `Pkg.add` and `Pkg.rm` commands in the previous section interact with registered packages, but the package manager can install and work with unregistered packages too. To install an unregistered package, use `Pkg.clone(url)`, where `url` is a git URL from which the package can be cloned:

```
julia> Pkg.clone("git://example.com/path/to/Package.jl.git")
INFO: Cloning Package from git://example.com/path/to/Package.jl.git
Cloning into 'Package'...
remote: Counting objects: 22, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 22 (delta 8), reused 22 (delta 8)
Receiving objects: 100% (22/22), 2.64 KiB, done.
Resolving deltas: 100% (8/8), done.
```

By convention, Julia repository names end with `.jl` (the additional `.git` indicates a "bare" git repository), which keeps them from colliding with repositories for other languages, and also makes Julia packages easy to find in search engines. When packages are installed in your `.julia/v0.3` directory, however, the extension is redundant so we leave it off.

If unregistered packages contain a `REQUIRE` file at the top of their source tree, that file will be used to determine which registered packages the unregistered package depends on, and they will automatically be installed. Unregistered packages participate in the same version resolution logic as registered packages, so installed package versions will be adjusted as necessary to satisfy the requirements of both registered and unregistered packages.

更新扩展包

When package developers publish new registered versions of packages that you're using, you will, of course, want the new shiny versions. To get the latest and greatest versions of all your packages, just do `Pkg.update()`:

```
julia> Pkg.update()
INFO: Updating METADATA...
INFO: Computing changes...
INFO: Upgrading Distributions: v0.2.8 => v0.2.10
INFO: Upgrading Stats: v0.2.7 => v0.2.8
```

The first step of updating packages is to pull new changes to `~/.julia/v0.3/METADATA` and see if any new registered package versions have been published. After this, `Pkg.update()` attempts to update packages that are checked out on a branch and not dirty (i.e. no changes have been made to files tracked by git) by pulling changes from the package's upstream repository. Upstream changes will only be applied if no merging or rebasing is necessary – i.e. if the branch can be "fast-forwarded". If the branch cannot be fast-forwarded, it is assumed that you're working on it and will update the repository yourself.

Finally, the update process recomputes an optimal set of package versions to have installed to satisfy your top-level requirements and the requirements of "fixed" packages. A package is considered fixed if it is one of the following:

1. **Unregistered:** the package is not in `METADATA` – you installed it with `Pkg.clone`.
2. **Checked out:** the package repo is on a development branch.
3. **Dirty:** changes have been made to files in the repo.

[1] The official set of packages is at https://github.com/JuliaLang/METADATA.jl, but individuals and organizations can easily use a different metadata repository. This allows control which packages are available for automatic installation. One can allow only audited and approved package versions, and make private packages or forks available.

If any of these are the case, the package manager cannot freely change the installed version of the package, so its requirements must be satisfied by whatever other package versions it picks. The combination of top-level requirements in `~/.julia/v0.3/REQUIRE` and the requirement of fixed packages are used to determine what should be installed.

Checkout, Pin and Free

You may want to use the `master` version of a package rather than one of its registered versions. There might be fixes or functionality on master that you need that aren't yet published in any registered versions, or you may be a developer of the package and need to make changes on `master` or some other development branch. In such cases, you can do `Pkg.checkout(pkg)` to checkout the `master` branch of `pkg` or `Pkg.checkout(pkg,branch)` to checkout some other branch:

```
julia> Pkg.add("Distributions")
INFO: Installing Distributions v0.2.9
INFO: Installing NumericExtensions v0.2.17
INFO: Installing Stats v0.2.7
INFO: REQUIRE updated.

julia> Pkg.status()
Required packages:
 - Distributions                 0.2.9
Additional packages:
 - NumericExtensions             0.2.17
 - Stats                         0.2.7

julia> Pkg.checkout("Distributions")
INFO: Checking out Distributions master...
INFO: No packages to install, update or remove.

julia> Pkg.status()
Required packages:
 - Distributions                 0.2.9+             master
Additional packages:
 - NumericExtensions             0.2.17
 - Stats                         0.2.7
```

Immediately after installing `Distributions` with `Pkg.add` it is on the current most recent registered version – `0.2.9` at the time of writing this. Then after running `Pkg.checkout("Distributions")`, you can see from the output of `Pkg.status()` that `Distributions` is on an unregistered version greater than `0.2.9`, indicated by the "pseudo-version" number `0.2.9+`.

When you checkout an unregistered version of a package, the copy of the `REQUIRE` file in the package repo takes precedence over any requirements registered in `METADATA`, so it is important that developers keep this file accurate and up-to-date, reflecting the actual requirements of the current version of the package. If the `REQUIRE` file in the package repo is incorrect or missing, dependencies may be removed when the package is checked out. This file is also used to populate newly published versions of the package if you use the API that `Pkg` provides for this (described below).

When you decide that you no longer want to have a package checked out on a branch, you can "free" it back to the control of the package manager with `Pkg.free(pkg)`:

```
julia> Pkg.free("Distributions")
INFO: Freeing Distributions...
INFO: No packages to install, update or remove.
```

```
julia> Pkg.status()
Required packages:
 - Distributions                 0.2.9
Additional packages:
 - NumericExtensions             0.2.17
 - Stats                         0.2.7
```

After this, since the package is on a registered version and not on a branch, its version will be updated as new registered versions of the package are published.

If you want to pin a package at a specific version so that calling `Pkg.update()` won't change the version the package is on, you can use the `Pkg.pin` function:

```
julia> Pkg.pin("Stats")
INFO: Creating Stats branch pinned.47c198b1.tmp

julia> Pkg.status()
Required packages:
 - Distributions                 0.2.9
Additional packages:
 - NumericExtensions             0.2.17
 - Stats                         0.2.7              pinned.47c198b1.tmp
```

After this, the `Stats` package will remain pinned at version `0.2.7` – or more specifically, at commit `47c198b1`, but since versions are permanently associated a given git hash, this is the same thing. `Pkg.pin` works by creating a throw-away branch for the commit you want to pin the package at and then checking that branch out. By default, it pins a package at the current commit, but you can choose a different version by passing a second argument:

```
julia> Pkg.pin("Stats",v"0.2.5")
INFO: Creating Stats branch pinned.1fd0983b.tmp
INFO: No packages to install, update or remove.

julia> Pkg.status()
Required packages:
 - Distributions                 0.2.9
Additional packages:
 - NumericExtensions             0.2.17
 - Stats                         0.2.5              pinned.1fd0983b.tmp
```

Now the `Stats` package is pinned at commit `1fd0983b`, which corresponds to version `0.2.5`. When you decide to "unpin" a package and let the package manager update it again, you can use `Pkg.free` like you would to move off of any branch:

```
julia> Pkg.free("Stats")
INFO: Freeing Stats...
INFO: No packages to install, update or remove.

julia> Pkg.status()
Required packages:
 - Distributions                 0.2.9
Additional packages:
 - NumericExtensions             0.2.17
 - Stats                         0.2.7
```

After this, the `Stats` package is managed by the package manager again, and future calls to `Pkg.update()` will upgrade it to newer versions when they are published. The throw-away `pinned.1fd0983b.tmp` branch remains in your local `Stats` repo, but since git branches are extremely lightweight, this doesn't really matter; if you feel like cleaning them up, you can go into the repo and delete those branches.

CHAPTER 25

开发扩展包

Julia's package manager is designed so that when you have a package installed, you are already in a position to look at its source code and full development history. You are also able to make changes to packages, commit them using git, and easily contribute fixes and enhancements upstream. Similarly, the system is designed so that if you want to create a new package, the simplest way to do so is within the infrastructure provided by the package manager.

Initial Setup

Since packages are git repositories, before doing any package development you should setup the following standard global git configuration settings:

```
$ git config --global user.name "FULL NAME"
$ git config --global user.email "EMAIL"
```

where `FULL NAME` is your actual full name (spaces are allowed between the double quotes) and `EMAIL` is your actual email address. Although it isn't necessary to use GitHub to create or publish Julia packages, most Julia packages as of writing this are hosted on GitHub and the package manager knows how to format origin URLs correctly and otherwise work with the service smoothly. We recommend that you create a free account on GitHub and then do:

```
$ git config --global github.user "USERNAME"
```

where `USERNAME` is your actual GitHub user name. Once you do this, the package manager knows your GitHub user name and can configure things accordingly. You should also upload your public SSH key to GitHub and set up an SSH agent on your development machine so that you can push changes with minimal hassle. In the future, we will make this system extensible and support other common git hosting options like BitBucket and allow developers to choose their favorite.

Generating a New Package

Suppose you want to create a new Julia package called `FooBar`. To get started, do `Pkg.generate(pkg, license)` where `pkg` is the new package name and `license` is the name of a license that the package generator

knows about:

```
julia> Pkg.generate("FooBar","MIT")
INFO: Initializing FooBar repo: /Users/stefan/.julia/v0.3/FooBar
INFO: Origin: git://github.com/StefanKarpinski/FooBar.jl.git
INFO: Generating LICENSE.md
INFO: Generating README.md
INFO: Generating src/FooBar.jl
INFO: Generating test/runtests.jl
INFO: Generating .travis.yml
INFO: Committing FooBar generated files
```

This creates the directory `~/.julia/v0.3/FooBar`, initializes it as a git repository, generates a bunch of files that all packages should have, and commits them to the repository:

```
$ cd ~/.julia/v0.3/FooBar && git show --stat

commit 84b8e266dae6de30ab9703150b3bf771ec7b6285
Author: Stefan Karpinski <stefan@karpinski.org>
Date:   Wed Oct 16 17:57:58 2013 -0400

    FooBar.jl generated files.

        license: MIT
        authors: Stefan Karpinski
        years:   2013
        user: StefanKarpinski

    Julia Version 0.3.0-prerelease+3217 [5fcfb13*]

 .travis.yml      | 16 ++++++++++++++
 LICENSE.md       | 22 ++++++++++++++++++++++
 README.md        |  3 +++
 src/FooBar.jl    |  5 +++++
 test/runtests.jl |  5 +++++
 5 files changed, 51 insertions(+)
```

At the moment, the package manager knows about the MIT "Expat" License, indicated by `"MIT"`, the Simplified BSD License, indicated by `"BSD"`, and version 2.0 of the Apache Software License, indicated by `"ASL"`. If you want to use a different license, you can ask us to add it to the package generator, or just pick one of these three and then modify the `~/.julia/v0.3/PACKAGE/LICENSE.md` file after it has been generated.

If you created a GitHub account and configured git to know about it, `Pkg.generate` will set an appropriate origin URL for you. It will also automatically generate a `.travis.yml` file for using the Travis automated testing service. You will have to enable testing on the Travis website for your package repository, but once you've done that, it will already have working tests. Of course, all the default testing does is verify that `using FooBar` in Julia works.

Making Your Package Available

Once you've made some commits and you're happy with how `FooBar` is working, you may want to get some other people to try it out. First you'll need to create the remote repository and push your code to it; we don't yet automatically do this for you, but we will in the future and it's not too hard to figure out[3]. Once you've done this, letting people try out your code is as simple as sending them the URL of the published repo – in this case:

[3] Installing and using GitHub's "hub" tool is highly recommended. It allows you to do things like run `hub create` in the package repo and have it automatically created via GitHub's API.

```
git://github.com/StefanKarpinski/FooBar.jl.git
```

For your package, it will be your GitHub user name and the name of your package, but you get the idea. People you send this URL to can use `Pkg.clone` to install the package and try it out:

```
julia> Pkg.clone("git://github.com/StefanKarpinski/FooBar.jl.git")
INFO: Cloning FooBar from git@github.com:StefanKarpinski/FooBar.jl.git
```

Publishing Your Package

Once you've decided that `FooBar` is ready to be registered as an official package, you can add it to your local copy of `METADATA` using `Pkg.register`:

```
julia> Pkg.register("FooBar")
INFO: Registering FooBar at git://github.com/StefanKarpinski/FooBar.jl.git
INFO: Committing METADATA for FooBar
```

This creates a commit in the `~/.julia/v0.3/METADATA` repo:

```
$ cd ~/.julia/v0.3/METADATA && git show

commit 9f71f4becb05cadacb983c54a72eed744e5c019d
Author: Stefan Karpinski <stefan@karpinski.org>
Date:   Wed Oct 16 18:46:02 2013 -0400

    Register FooBar

diff --git a/FooBar/url b/FooBar/url
new file mode 100644
index 0000000..30e525e
--- /dev/null
+++ b/FooBar/url
@@ -0,0 +1 @@
+git://github.com/StefanKarpinski/FooBar.jl.git
```

This commit is only locally visible, however. In order to make it visible to the world, you need to merge your local `METADATA` upstream into the official repo. The `Pkg.publish()` command will fork the `METADATA` repository on GitHub, push your changes to your fork, and open a pull request

```
julia> Pkg.publish()
INFO: Validating METADATA
INFO: No new package versions to publish
INFO: Submitting METADATA changes
INFO: Forking JuliaLang/METADATA.jl to StefanKarpinski
INFO: Pushing changes as branch pull-request/ef45f54b
INFO: To create a pull-request open:

  https://github.com/StefanKarpinski/METADATA.jl/compare/pull-request/ef45f54b
```

For various reasons `Pkg.publish()` sometimes does not succeed. In those cases, you may make a pull request on GitHub, which is not difficult.

Once the package URL for `FooBar` is registered in the official `METADATA` repo, people know where to clone the package from, but there still aren't any registered versions available. This means that `Pkg.add("FooBar")` won't work yet since it only installs official versions. `Pkg.clone("FooBar")` without having to specify a URL for it.

Moreover, when they run `Pkg.update()`, they will get the latest version of `FooBar` that you've pushed to the repo. This is a good way to have people test out your packages as you work on them, before they're ready for an official release.

Tagging Package Versions

Once you are ready to make an official version your package, you can tag and register it with the `Pkg.tag` command:

```
julia> Pkg.tag("FooBar")
INFO: Tagging FooBar v0.0.1
INFO: Committing METADATA for FooBar
```

This tags `v0.0.1` in the `FooBar` repo:

```
$ cd ~/.julia/v0.3/FooBar && git tag
v0.0.1
```

It also creates a new version entry in your local `METADATA` repo for `FooBar`:

```
$ cd ~/.julia/v0.3/FooBar && git show
commit de77ee4dc0689b12c5e8b574aef7f70e8b311b0e
Author: Stefan Karpinski <stefan@karpinski.org>
Date:   Wed Oct 16 23:06:18 2013 -0400

    Tag FooBar v0.0.1

diff --git a/FooBar/versions/0.0.1/sha1 b/FooBar/versions/0.0.1/sha1
new file mode 100644
index 0000000..c1cb1c1
--- /dev/null
+++ b/FooBar/versions/0.0.1/sha1
@@ -0,0 +1 @@
+84b8e266dae6de30ab9703150b3bf771ec7b6285
```

If there is a `REQUIRE` file in your package repo, it will be copied into the appropriate spot in `METADATA` when you tag a version. Package developers should make sure that the `REQUIRE` file in their package correctly reflects the requirements of their package, which will automatically flow into the official metadata if you're using `Pkg.tag`. See the *Requirements Specification* for the full format of `REQUIRE`.

The `Pkg.tag` command takes an optional second argument that is either an explicit version number object like `v"0.0.1"` or one of the symbols `:patch`, `:minor` or `:major`. These increment the patch, minor or major version number of your package intelligently.

As with `Pkg.register`, these changes to `METADATA` aren't available to anyone else until they've been included upstream. Again, use the `Pkg.publish()` command, which first makes sure that individual package repos have been tagged, pushes them if they haven't already been, and then opens a pull request to `METADATA`:

```
julia> Pkg.publish()
INFO: Validating METADATA
INFO: Pushing FooBar permanent tags: v0.0.1
INFO: Submitting METADATA changes
INFO: Forking JuliaLang/METADATA.jl to StefanKarpinski
INFO: Pushing changes as branch pull-request/3ef4f5c4
INFO: To create a pull-request open:

  https://github.com/StefanKarpinski/METADATA.jl/compare/pull-request/3ef4f5c4
```

Fixing Package Requirements

If you need to fix the registered requirements of an already-published package version, you can do so just by editing the metadata for that version, which will still have the same commit hash – the hash associated with a version is permanent:

```
$ cd ~/.julia/v0.3/METADATA/FooBar/versions/0.0.1 && cat requires
julia 0.3-

$ vi requires
```

Since the commit hash stays the same, the contents of the `REQUIRE` file that will be checked out in the repo will **not** match the requirements in `METADATA` after such a change; this is unavoidable. When you fix the requirements in `METADATA` for a previous version of a package, however, you should also fix the `REQUIRE` file in the current version of the package.

依赖关系

The `~/.julia/v0.3/REQUIRE` file, the `REQUIRE` file inside packages, and the `METADATA` package `requires` files use a simple line-based format to express the ranges of package versions which need to be installed. Package `REQUIRE` and `METADATA requires` files should also include the range of versions of `julia` the package is expected to work with.

Here's how these files are parsed and interpreted.

- Everything after a `#` mark is stripped from each line as a comment.
- If nothing but whitespace is left, the line is ignored.
- If there are non-whitespace characters remaining, the line is a requirement and the is split on whitespace into words.

The simplest possible requirement is just the name of a package name on a line by itself:

```
Distributions
```

This requirement is satisfied by any version of the `Distributions` package. The package name can be followed by zero or more version numbers in ascending order, indicating acceptable intervals of versions of that package. One version opens an interval, while the next closes it, and the next opens a new interval, and so on; if an odd number of version numbers are given, then arbitrarily large versions will satisfy; if an even number of version numbers are given, the last one is an upper limit on acceptable version numbers. For example, the line:

```
Distributions 0.1
```

is satisfied by any version of `Distributions` greater than or equal to `0.1.0`. Suffixing a version with `-` allows any pre-release versions as well. For example:

```
Distributions 0.1-
```

is satisfied by pre-release versions such as `0.1-dev` or `0.1-rc1`, or by any version greater than or equal to `0.1.0`.

This requirement entry:

```
Distributions 0.1 0.2.5
```

is satisfied by versions from `0.1.0` up to, but not including `0.2.5`. If you want to indicate that any `0.1.x` version will do, you will want to write:

```
Distributions 0.1 0.2-
```

If you want to start accepting versions after `0.2.7`, you can write:

```
Distributions 0.1 0.2- 0.2.7
```

If a requirement line has leading words that begin with `@`, it is a system-dependent requirement. If your system matches these system conditionals, the requirement is included, if not, the requirement is ignored. For example:

```
@osx Homebrew
```

will require the `Homebrew` package only on systems where the operating system is OS X. The system conditions that are currently supported are:

```
@windows
@unix
@osx
@linux
```

The `@unix` condition is satisfied on all UNIX systems, including OS X, Linux and FreeBSD. Negated system conditionals are also supported by adding a `!` after the leading `@`. Examples:

```
@!windows
@unix @!osx
```

The first condition applies to any system but Windows and the second condition applies to any UNIX system besides OS X.

Runtime checks for the current version of Julia can be made using the built-in `VERSION` variable, which is of type `VersionNumber`. Such code is occasionally necessary to keep track of new or deprecated functionality between various releases of Julia. Examples of runtime checks:

```
VERSION < v"0.3-" #exclude all pre-release versions of 0.3

v"0.2-" <= VERSION < v"0.3-" #get all 0.2 versions, including pre-releases, up to the␣
↪above

v"0.2" <= VERSION < v"0.3-" #To get only stable 0.2 versions (Note v"0.2" == v"0.2.0")

VERSION >= v"0.2.1" #get at least version 0.2.1
```

See the section on *version number literals* for a more complete description.

CHAPTER 26

代码性能优化

以下几节将描述一些提高 Julia 代码运行速度的技巧。

避免全局变量

全局变量的值、类型，都可能变化。这使得编译器很难优化使用全局变量的代码。应尽量使用局部变量，或者把变量当做参数传递给函数。

对性能至关重要的代码，应放入函数中。

声明全局变量为常量可以显著提高性能:

```
const DEFAULT_VAL = 0
```

使用非常量的全局变量时，最好在使用时指明其类型，这样也能帮助编译器优化:

```
global x
y = f(x::Int + 1)
```

Writing functions is better style. It leads to more reusable code and clarifies what steps are being done, and what their inputs and outputs are.

使用 `@time` 来衡量性能并且留心内存分配

衡量计算性能最有用的工具是 `@time` 宏. 下面的例子展示了良好的使用方式

```
julia> function f(n)
           s = 0
           for i = 1:n
               s += i/2
           end
           s
```

```
        end
f (generic function with 1 method)

julia> @time f(1)
elapsed time: 0.008217942 seconds (93784 bytes allocated)
0.5

julia> @time f(10^6)
elapsed time: 0.063418472 seconds (32002136 bytes allocated)
2.5000025e11
```

在第一次调用时 (`@time f(1)`), `f` 会被编译. (如果你在这次会话中还 没有使用过 `@time`, 计时函数也会被编译.) 这时的结果没有那么重要. 在 第二次调用时, 函数打印了执行所耗费的时间, 同时请注意, 在这次执行过程中 分配了一大块的内存. 相对于函数形式的 `tic` 和 `toc`, 这是 `@time` 宏的一大优势.

出乎意料的大块内存分配往往意味着程序的某个部分存在问题, 通常是关于类型 稳定性. 因此, 除了关注内存分配本身的问题, 很可能 Julia 为你的函数生成 的代码存在很大的性能问题. 这时候要认真对待这些问题并遵循下面的一些个建 议.

另外, 作为一个引子, 上面的问题可以优化为无内存分配 (除了向 REPL 返回结 果), 计算速度提升 30 倍

```
julia> @time f_improved(10^6)
elapsed time: 0.00253829 seconds (112 bytes allocated)
2.5000025e11
```

你可以从下面的章节学到如何识别 `f` 存在的问题并解决.

在有些情况下, 你的函数可能需要为本身的操作分配内存, 这样会使得问题变得 复杂. 在这种情况下, 可以考虑使用下面的 工具 之一来甄别问题, 或者将函数拆分, 一部分处理内存分 配, 另一部分处理算法 (参见 预分配内存). 工具 —

Julia 提供了一些工具包来鉴别性能问题所在 :

- *Profiling* 可以用来衡量代码的性能, 同时鉴别出瓶颈所在. 对于复杂的项目, 可以使用 *ProfileView <https://github.com/timholy/ProfileView.jl>* 扩展包来直观的展示分析 结果.
- 出乎意料的大块内存分配, – `@time`, `@allocated`, 或者 -profiler - 意味着你的代码可能存在问题. 如果你看不出内存分配的问题, -那么类型系统可能存在问题. 也可以使用 `--track-allocation=user` 来 -启动 Julia, 然后查看 `*.mem` 文件来找出内存分配是在哪里出现的.
- **'TypeCheck <https://github.com/astrieanna/TypeCheck.jl'_** 可以帮助找 出部分类型系统相关的问题. 另一个更费力但是更全面的工具是 `code_typed`. 特别留意类型为 `Any` 的变量, 或者 `Union` 类型. 这些问题可以使用下面的建议解决.
- Lint 扩展包可以指出程序一 些问题.

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

```
a = Real[]     # typeof(a) = Array{Real,1}
if (f = rand()) < .8
    push!(a, f)
end
```

Because a is a an array of abstract type Real, it must be able to hold any Real value. Since Real objects can be of arbitrary size and structure, a must be represented as an array of pointers to individually allocated Real objects. Because f will always be a Float64, we should instead, use:

```
a = Float64[] # typeof(a) = Array{Float64,1}
```

which will create a contiguous block of 64-bit floating-point values that can be manipulated efficiently.

See also the discussion under 参数化类型.

类型声明

在 Julia 中，编译器能推断出所有的函数参数与局部变量的类型，因此声名变量类型不能提高性能。然而在有些具体实例中，声明类型还是非常有用的。

给复合类型做类型声明

假如有一个如下的自定义类型：

```
type Foo
    field
end
```

编译器推断不出 foo.field 的类型，因为当它指向另一个不同类型的值时，它的类型也会被修改。这时最好声明具体的类型，比如 field::Float64 或者 field::Array{Int64,1} 。

显式声明未提供类型的值的类型

我们经常使用含有不同数据类型的数据结构，比如上述的 Foo 类型，或者元胞数组（Array{Any} 类型的数组）。如果你知道其中元素的类型，最好把它告诉编译器：

```
function foo(a::Array{Any,1})
    x = a[1]::Int32
    b = x+1
    ...
end
```

假如我们知道 a 的第一个元素是 Int32 类型的，那就添加上这样的类型声明吧。如果这个元素不是这个类型，在运行时就会报错，这有助于调试代码。

显式声明命名参数的值的类型

命名参数可以显式指定类型：

```
function with_keyword(x; name::Int = 1)
    ...
end
```

函数只处理指定类型的命名参数，因此这些声明不会对该函数内部代码的性能产生影响。 不过，这会减少此类包含命名参数的函数的调用开销。

与直接使用参数列表的函数相比，命名参数的函数调用新增的开销很少，基本上可算是零开销。

如果传入函数的是命名参数的动态列表，例如“f(x; keywords...)“，速度会比较慢，性能敏感的代码慎用。

把函数拆开

把一个函数拆为多个，有助于编译器调用最匹配的代码，甚至将它内联。

举个应该把“复合函数”写成多个小定义的例子：

```
function norm(A)
    if isa(A, Vector)
        return sqrt(real(dot(A,A)))
    elseif isa(A, Matrix)
        return max(svd(A)[2])
    else
        error("norm: invalid argument")
    end
end
```

如下重写会更精确、高效：

```
norm(x::Vector) = sqrt(real(dot(x,x)))
norm(A::Matrix) = max(svd(A)[2])
```

写“类型稳定”的函数

尽量确保函数返回同样类型的数值。考虑下面定义：

```
pos(x) = x < 0 ? 0 : x
```

尽管看起来没问题，但是 `0` 是个整数（`Int` 型），`x` 可能是任意类型。因此，函数有返回两种类型的可能。这个是可以的，有时也很有用，但是最好如下重写：

```
pos(x) = x < 0 ? zero(x) : x
```

Julia 中还有 `one` 函数，以及更通用的 `oftype(x,y)` 函数，它将 `y` 转换为与 `x` 同样的类型，并返回。这仨函数的第一个参数，可以是一个值，也可以是一个类型。

避免改变变量类型

在一个函数中重复地使用变量，会导致类似于“类型稳定性”的问题：

```
function foo()
    x = 1
    for i = 1:10
        x = x/bar()
    end
    return x
end
```

局部变量 `x` 开始为整数，循环一次后变成了浮点数（`/` 运算符的结果）。这使得编译器很难优化循环体。可以修改为如下的任何一种：

- 用 `x = 1.0` 初始化 `x`
- 声明 `x` 的类型: `x::Float64 = 1`
- 使用显式转换: `x = one(T)`

分离核心函数

很多函数都先做些初始化设置，然后开始很多次循环迭代去做核心计算。尽可能把这些核心计算放在单独的函数中。例如，下面的函数返回一个随机类型的数组:

```
function strange_twos(n)
    a = Array(randbool() ? Int64 : Float64, n)
    for i = 1:n
        a[i] = 2
    end
    return a
end
```

应该写成:

```
function fill_twos!(a)
    for i=1:length(a)
        a[i] = 2
    end
end

function strange_twos(n)
    a = Array(randbool() ? Int64 : Float64, n)
    fill_twos!(a)
    return a
end
```

Julia 的编译器依靠参数类型来优化代码。第一个实现中，编译器在循环时不知道 `a` 的类型（因为类型是随机的）。第二个实现中，内层循环使用 `fill_twos!` 对不同的类型 `a` 重新编译，因此运行速度更快。

第二种实现的代码更好，也更便于代码复用。

标准库中经常使用这种方法。如 abstractarray.jl 文件中的 `hvcat_fill` 和 `fill!` 函数。我们可以用这两个函数来替代这儿的 `fill_twos!` 函数。

形如 `strange_twos` 之类的函数经常用于处理未知类型的数据。比如，从文件载入的数据，可能包含整数、浮点数、字符串，或者其他类型。

Access arrays in memory order, along columns

Multidimensional arrays in Julia are stored in column-major order. This means that arrays are stacked one column at a time. This can be verified using the `vec` function or the syntax `[:]` as shown below (notice that the array is ordered `[1 3 2 4]`, not `[1 2 3 4]`):

```
julia> x = [1 2; 3 4]
2x2 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.

Consider the following contrived example. Imagine we wanted to write a function that accepts a `Vector` and and returns a square `Matrix` with either the rows or the columns filled with copies of the input vector. Assume that it is not important whether rows or columns are filled with these copies (perhaps the rest of the code can be easily adapted accordingly). We could conceivably do this in at least four ways (in addition to the recommended call to the built-in function `repmat`):

```
function copy_cols{T}(x::Vector{T})
    n = size(x, 1)
    out = Array(eltype(x), n, n)
    for i=1:n
        out[:, i] = x
    end
    out
end

function copy_rows{T}(x::Vector{T})
    n = size(x, 1)
    out = Array(eltype(x), n, n)
    for i=1:n
        out[i, :] = x
    end
    out
end

function copy_col_row{T}(x::Vector{T})
    n = size(x, 1)
    out = Array(T, n, n)
    for col=1:n, row=1:n
        out[row, col] = x[row]
    end
    out
end

function copy_row_col{T}(x::Vector{T})
    n = size(x, 1)
    out = Array(T, n, n)
    for row=1:n, col=1:n
        out[row, col] = x[col]
    end
    out
end
```

Now we will time each of these functions using the same random `10000` by `1` input vector:

```
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
```

Notice that `copy_cols` is much faster than `copy_rows`. This is expected because `copy_cols` respects the column-based memory layout of the `Matrix` and fills it one column at a time. Additionally, `copy_col_row` is much faster than `copy_row_col` because it follows our rule of thumb that the first element to appear in a slice expression should be coupled with the inner-most loop.

Pre-allocating outputs

If your function returns an Array or some other complex type, it may have to allocate memory. Unfortunately, oftentimes allocation and its converse, garbage collection, are substantial bottlenecks.

Sometimes you can circumvent the need to allocate memory on each function call by pre-allocating the output. As a trivial example, compare

```
function xinc(x)
    return [x, x+1, x+2]
end

function loopinc()
    y = 0
    for i = 1:10^7
        ret = xinc(i)
        y += ret[2]
    end
    y
end
```

with

```
function xinc!{T}(ret::AbstractVector{T}, x::T)
    ret[1] = x
    ret[2] = x+1
    ret[3] = x+2
    nothing
end

function loopinc_prealloc()
    ret = Array(Int, 3)
    y = 0
    for i = 1:10^7
        xinc!(ret, i)
        y += ret[2]
    end
    y
end
```

Timing results:

```
julia> @time loopinc()
elapsed time: 1.955026528 seconds (1279975584 bytes allocated)
50000015000000

julia> @time loopinc_prealloc()
elapsed time: 0.078639163 seconds (144 bytes allocated)
50000015000000
```

Pre-allocation has other advantages, for example by allowing the caller to control the "output" type from an algorithm. In the example above, we could have passed a `SubArray` rather than an `Array`, had we so desired.

Taken to its extreme, pre-allocation can make your code uglier, so performance measurements and some judgment may be required.

Avoid string interpolation for I/O

When writing data to a file (or other I/O device), forming extra intermediate strings is a source of overhead. Instead of:

```
println(file, "$a $b")
```

use:

```
println(file, a, " ", b)
```

The first version of the code forms a string, then writes it to the file, while the second version writes values directly to the file. Also notice that in some cases string interpolation can be harder to read. Consider:

```
println(file, "$(f(a))$(f(b))")
```

versus:

```
println(file, f(a), f(b))
```

处理有关舍弃的警告

被舍弃的函数，会查表并显示一次警告，而这会影响性能。建议按照警告的提示进行对应的修改。

小技巧

注意些有些小事项，能使内部循环更紧致。

- 避免不必要的数组。例如，不要使用 `sum([x,y,z])` ，而应使用 `x+y+z`
- 对于较小的整数幂，使用 `*` 更好。如 `x*x*x` 比 `x^3` 好
- 针对复数 `z` ，使用 `abs2(z)` 代替 `abs(z)^2` 。一般情况下，对于复数参数，尽量用 `abs2` 代替 `abs`
- 对于整数除法，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)`.

Performance Annotations

Sometimes you can enable better optimization by promising certain program properties.

- Use `@inbounds` to eliminate array bounds checking within expressions. Be certain before doing this. If the subscripts are ever out of bounds, you may suffer crashes or silent corruption.
- Write `@simd` in front of `for` loops that are amenable to vectorization. **This feature is experimental** and could change or disappear in future versions of Julia.

Here is an example with both forms of markup:

```
function inner( x, y )
    s = zero(eltype(x))
    for i=1:length(x)
        @inbounds s += x[i]*y[i]
    end
    s
end

function innersimd( x, y )
    s = zero(eltype(x))
    @simd for i=1:length(x)
        @inbounds s += x[i]*y[i]
    end
    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        = ",2.0*n*reps/time*1E-9)
    time = @elapsed for j in 1:reps
        s+=innersimd(x,y)
    end
    println("GFlop (SIMD) = ",2.0*n*reps/time*1E-9)
end

timeit(1000,1000)
```

On a computer with a 2.4GHz Intel Core i5 processor, this produces:

```
GFlop        = 1.9467069505224963
GFlop (SIMD) = 17.578554163920018
```

The range for a `@simd for` loop should be a one-dimensional range. A variable used for accumulating, such as `s` in the example, is called a *reduction variable*. 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`.
- No iteration ever waits on another iteration to make forward progress.

Using `@simd` merely gives the compiler license to vectorize. Whether it actually does so depends on the compiler. To actually benefit from the current implementation, your loop should have the following additional properties:

- The loop must be an innermost loop.
- The loop body must be straight-line code. This is why `@inbounds` is currently needed for all array accesses.
- Accesses must have a stride pattern and cannot be "gathers" (random-index reads) or "scatters" (random-index writes).
- The stride should be unit stride.
- In some simple cases, for example with 2-3 arrays accessed in a loop, the LLVM auto-vectorization may kick in automatically, leading to no further speedup with `@simd`.

CHAPTER 27

代码样式

The following sections explain a few aspects of idiomatic Julia coding style. None of these rules are absolute; they are only suggestions to help familiarize you with the language and to help you choose among alternative designs.

写成函数，别写成脚本

Writing code as a series of steps at the top level is a quick way to get started solving a problem, but you should try to divide a program into functions as soon as possible. Functions are more reusable and testable, and clarify what steps are being done and what their inputs and outputs are. Furthermore, code inside functions tends to run much faster than top level code, due to how Julia's compiler works.

It is also worth emphasizing that functions should take arguments, instead of operating directly on global variables (aside from constants like `pi`).

避免类型过于严格

Code should be as generic as possible. Instead of writing:

```
convert(Complex{Float64}, x)
```

it's better to use available generic functions:

```
complex(float(x))
```

The second version will convert `x` to an appropriate type, instead of always the same type.

This style point is especially relevant to function arguments. For example, don't declare an argument to be of type `Int` or `Int32` if it really could be any integer, expressed with the abstract type `Integer`. In fact, in many cases you can omit the argument type altogether, unless it is needed to disambiguate from other method definitions, since a `MethodError` will be thrown anyway if a type is passed that does not support any of the requisite operations. (This is known as duck typing.)

For example, consider the following definitions of a function `addone` that returns one plus its argument:

```
addone(x::Int) = x + 1              # works only for Int
addone(x::Integer) = x + one(x)     # any integer type
addone(x::Number) = x + one(x)      # any numeric type
addone(x) = x + one(x)              # any type supporting + and one
```

The last definition of `addone` handles any type supporting the `one` function (which returns 1 in the same type as `x`, which avoids unwanted type promotion) and the `+` function with those arguments. The key thing to realize is that there is *no performance penalty* to defining *only* the general `addone(x) = x + one(x)`, because Julia will automatically compile specialized versions as needed. For example, the first time you call `addone(12)`, Julia will automatically compile a specialized `addone` function for `x::Int` arguments, with the call to `one` replaced by its inlined value `1`. Therefore, the first three definitions of `addone` above are completely redundant.

Handle excess argument diversity in the caller

Instead of:

```
function foo(x, y)
    x = int(x); y = int(y)
    ...
end
foo(x, y)
```

use:

```
function foo(x::Int, y::Int)
    ...
end
foo(int(x), int(y))
```

This is better style because `foo` does not really accept numbers of all types; it really needs `Int` s.

One issue here is that if a function inherently requires integers, it might be better to force the caller to decide how non-integers should be converted (e.g. floor or ceiling). Another issue is that declaring more specific types leaves more "space" for future method definitions.

如果函数修改了它的参数，在函数名后加 *!*

Instead of:

```
function double{T<:Number}(a::AbstractArray{T})
    for i = 1:endof(a); a[i] *= 2; end
    a
end
```

use:

```
function double!{T<:Number}(a::AbstractArray{T})
    for i = 1:endof(a); a[i] *= 2; end
    a
end
```

The Julia standard library uses this convention throughout and contains examples of functions with both copying and modifying forms (e.g., `sort` and `sort!`), and others which are just modifying (e.g., `push!`, `pop!`, `splice!`). It is typical for such functions to also return the modified array for convenience.

避免奇葩的类型集合

像 `Union(Function,String)` 这样的类型，说明你的设计有问题。

Try to avoid nullable fields

When using `x::Union(Nothing,T)`, ask whether the option for `x` to be `nothing` is really necessary. Here are some alternatives to consider:

- Find a safe default value to initialize `x` with
- Introduce another type that lacks `x`
- If there are many fields like `x`, store them in a dictionary
- Determine whether there is a simple rule for when `x` is `nothing`. For example, often the field will start as `nothing` but get initialized at some well-defined point. In that case, consider leaving it undefined at first.

Avoid elaborate container types

It is usually not much help to construct arrays like the following:

```
a = Array(Union(Int,String,Tuple,Array), n)
```

In this case `cell(n)` is better. It is also more helpful to the compiler to annotate specific uses (e.g. `a[i]::Int`) than to try to pack many alternatives into one type.

使用和 Julia `base/` 相同的命名传统

- 模块和类型名称以大写开头，并且使用驼峰形式: `module SparseMatrix`, `immutable UnitRange`.
- 函数名称使用小写 (`maximum`, `convert`). 在容易读懂的情况下把几 个单词连在一起写 (`isequal`, `haskey`). 在必要的情况下, 使用下划 线作为单词的分隔符. 下划线也可以用来表示多个概念的组合 (`remotecall_fetch` 相比 `remotecall(fetch(...))` 是一种更有效的 实现), 或者是为了区分 (`sum_kbn`). 简洁是提倡的, 但是要避免缩写 (`indexin` 而不是 `indxin`) 因为很难记住某些单词是否缩写或者怎么 缩写的.

如果一个函数需要多个单词来描述, 想一下这个函数是否包含了多个概念, 这样 的情况下最好分拆成多个部分.

不要滥用 try-catch

It is better to avoid errors than to rely on catching them.

不要把条件表达式用圆括号括起来

Julia doesn't require parens around conditions in `if` and `while`. Write:

```
if a == b
```

instead of:

```
if (a == b)
```

不要滥用 ...

Splicing function arguments can be addictive. Instead of `[a..., b...]`, use simply `[a, b]`, which already concatenates arrays. `collect(a)` is better than `[a...]`, but since `a` is already iterable it is often even better to leave it alone, and not convert it to an array.

Don't use unnecessary static parameters

A function signature:

```
foo{T<:Real}(x::T) = ...
```

should be written as:

```
foo(x::Real) = ...
```

instead, especially if `T` is not used in the function body. Even if `T` is used, it can be replaced with `typeof(x)` if convenient. There is no performance difference. Note that this is not a general caution against static parameters, just against uses where they are not needed.

Note also that container types, specifically may need type parameters in function calls. See the FAQ 如何声明"抽象容器类型"的域 for more information.

Avoid confusion about whether something is an instance or a type

Sets of definitions like the following are confusing:

```
foo(::Type{MyType}) = ...
foo(::MyType) = foo(MyType)
```

Decide whether the concept in question will be written as `MyType` or `MyType()`, and stick to it.

The preferred style is to use instances by default, and only add methods involving `Type{MyType}` later if they become necessary to solve some problem.

If a type is effectively an enumeration, it should be defined as a single (ideally `immutable`) type, with the enumeration values being instances of it. Constructors and conversions can check whether values are valid. This design is preferred over making the enumeration an abstract type, with the "values" as subtypes.

不要滥用 macros

Be aware of when a macro could really be a function instead.

Calling `eval` inside a macro is a particularly dangerous warning sign; it means the macro will only work when called at the top level. If such a macro is written as a function instead, it will naturally have access to the run-time values it needs.

Don't expose unsafe operations at the interface level

If you have a type that uses a native pointer:

```
type NativeType
    p::Ptr{Uint8}
    ...
end
```

don't write definitions like the following:

```
getindex(x::NativeType, i) = unsafe_load(x.p, i)
```

The problem is that users of this type can write `x[i]` without realizing that the operation is unsafe, and then be susceptible to memory bugs.

Such a function should either check the operation to ensure it is safe, or have `unsafe` somewhere in its name to alert callers.

Don't overload methods of base container types

It is possible to write definitions like the following:

```
show(io::IO, v::Vector{MyType}) = ...
```

This would provide custom showing of vectors with a specific new element type. While tempting, this should be avoided. The trouble is that users will expect a well-known type like `Vector` to behave in a certain way, and overly customizing its behavior can make it harder to work with.

Be careful with type equality

You generally want to use `isa` and `<:` (`issubtype`) for testing types, not `==`. Checking types for exact equality typically only makes sense when comparing to a known concrete type (e.g. `T == Float64`), or if you *really, really* know what you're doing.

不要写 `x->f(x)`

高阶函数经常被用作匿名函数来调用，虽然这样很方便，但是尽量少这么写。例如，尽量把 `map(x->f(x), a)` 写成 `map(f, a)` 。

CHAPTER 28

常见问题

会话和 REPL

如何删除内存中的对象?

Julia 没有 MATLAB 的 `clear` 函数；在 Julia 会话（准确来说， `Main` 模块）中定义了一个名字的话，它就一直在啦。

如果你很关心内存使用，你可以用占内存的小的来替换大的。例如，如果 `A` 是个你不需要的大数组，可以先用 `A = 0` 来释放内存。下一次进行垃圾回收的时候，内存就会被释放了；你也可以直接调用 `gc()` 来回收。

如何在会话中修改 type/immutable 的声明?

有时候你定义了一种类型但是后来发现你需要添加一个新的域。当你尝试在REPL里这样做时就会出错

```
ERROR: invalid redefinition of constant MyType
```

`Main` 模块里的类型不能被重新定义。

当你在开发新代码时这会变得极其不方便，有一个很好的办法来处理。模块是可以用重新定义的办法来替换， 所以把你的所有的代码封装在一个模块里就能够重新定义类型以及常数。你不能把类型名导入到 `Main` 里 再去重新定义，但是你可以用模块名来解决这个问题。换句话说，当你开发的时候可以用这样的工作流

```
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)
...
```

函数

我把参数 x 传递给一个函数， 并在函数内修改它的值， 但是在函数外 x 的值并未发生变化， 为什么呢?

假设你像这样调用函数:

```
julia> x = 10
julia> function change_value!(y) # Create a new function
           y = 17
       end
julia> change_value!(x)
julia> x # x is unchanged!
10
```

在 Julia 里， 所有的函数(包括 `change_value!()`) 都不能修改局部变量的所属的类。 如果 `x` 被函数调用时被定义为一个不可变的对象(比如实数), 就不能修改; 同样地， 如果 `x` 被定义为一个 `Dict` 对象， 你不能把它改成ASCIIString。 但是需要主要的是: 假设 `x` 是一个数组(或者任何可变类型)。 你不能让 `x` 不再代表这个数组，但由于数组是可变的对象，你能修改数组的元素:

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

julia> function change_array!(A) # Create a new function
           A[1] = 5
       end
julia> change_array!(x)
julia> x
3-element Array{Int64,1}:
5
2
3
```

这里我们定义了函数 `change_array!()`, 把整数 `5` 分配给了数组的第一个元素。 当我们把 `x` 传读给这个函数时， 注意到 `x` 依然是同一个数组，只是数组的元素发生了变化。

我能在函数中使用 using 或者 import 吗?

不行， 在函数中不能使用 `using` 或 `import`。 如果你要导入一个模块但只是在某些函数里使用，你有两种方案:

1. 使用 `import`

```
import Foo
function bar(...)
    ... refer to Foo symbols via Foo.baz ...
end
```

2. 把函数封装到模块里:

```
module Bar
export bar
using Foo
function bar(...)
    ... refer to Foo.baz as simply baz ....
end
end
using Bar
```

类型，类型声明和构造方法

什么是“类型稳定”?

这意味着输出的类型是可以由输入类型预测出来。特别地，这表示输出的类型不能因输入的值的变化而 变化。下面这段代码 不是 类型稳定的

```
function unstable(flag::Bool)
    if flag
        return 1
    else
        return 1.0
    end
end
```

这段代码视参数的值的不同而返回一个 `Int` 或是 `Float64`。 因为 Julia 无法在编译时预测 函数返回值类型， 任何使用这个函数的计算都得考虑这两种可能的返回类型， 这样很难生成快速的机器码。

为什么看似合理的运算 Julia还是返回 `DomainError` ?

有些运算数学上讲得通但是会产生错误:

```
julia> sqrt(-2.0)
ERROR: DomainError
 in sqrt at math.jl:128

julia> 2^-5
ERROR: DomainError
 in power_by_squaring at intfuncs.jl:70
 in ^ at intfuncs.jl:84
```

这时由类型稳定造成的。对于 `sqrt`， 大多数用户会用 `sqrt(2.0)` 得到一个实数而不是得到一个复数 `1.4142135623730951 + 0.0im`。 也可以把 `sqrt` 写成当参数为负的时候返回复数， 但是这将不再是类型稳定 而且 `sqrt` 会变的很慢。

在这些情况下，你可以选择 输入类型 来得到想要的 输出类型

```
julia> sqrt(-2.0+0im)
0.0 + 1.4142135623730951im

julia> 2.0^-5
0.03125
```

Why does Julia use native machine integer arithmetic?

Julia uses machine arithmetic for integer computations. This means that the range of `Int` values is bounded and wraps around at either end so that adding, subtracting and multiplying integers can overflow or underflow, leading to some results that can be unsettling at first:

```
julia> typemax(Int)
9223372036854775807

julia> ans+1
-9223372036854775808

julia> -ans
-9223372036854775808

julia> 2*ans
0
```

Clearly, this is far from the way mathematical integers behave, and you might think it less than ideal for a high-level programming language to expose this to the user. For numerical work where efficiency and transparency are at a premium, however, the alternatives are worse.

One alternative to consider would be to check each integer operation for overflow and promote results to bigger integer types such as `Int128` or `BigInt` in the case of overflow. Unfortunately, this introduces major overhead on every integer operation (think incrementing a loop counter) – it requires emitting code to perform run-time overflow checks after arithmetic instructions and braches to handle potential overflows. Worse still, this would cause every computation involving integers to be type-unstable. As we mentioned above, *type-stability is crucial* for effective generation of efficient code. If you can't count on the results of integer operations being integers, it's impossible to generate fast, simple code the way C and Fortran compilers do.

A variation on this approach, which avoids the appearance of type instabilty is to merge the `Int` and `BigInt` types into a single hybrid integer type, that internally changes representation when a result no longer fits into the size of a machine integer. While this superficially avoids type-instability at the level of Julia code, it just sweeps the problem under the rug by foisting all of the same difficulties onto the C code implementing this hybrid integer type. This approach *can* be made to work and can even be made quite fast in many cases, but has several drawbacks. One problem is that the in-memory representation of integers and arrays of integers no longer match the natural representation used by C, Fortran and other languages with native machine integers. Thus, to interoperate with those languages, we would ultimately need to introduce native integer types anyway. Any unbounded representation of integers cannot have a fixed number of bits, and thus cannot be stored inline in an array with fixed-size slots – large integer values will always require separate heap-allocated storage. And of course, no matter how clever a hybrid integer implementation one uses, there are always performance traps – situations where performance degrades unexpectedly. Complex representation, lack of interoperability with C and Fortran, the inability to represent integer arrays without additional heap storage, and unpredictable performance characteristics make even the cleverest hybrid integer implementations a poor choice for high-performance numerical work.

An alternative to using hybrid integers or promoting to BigInts is to use saturating integer arithmetic, where adding to the largest integer value leaves it unchanged and likewise for subtracting from the smallest integer value. This is precisely what Matlab™ does:

```
>> int64(9223372036854775807)

ans =

  9223372036854775807

>> int64(9223372036854775807) + 1
```

```
ans =

  9223372036854775807

>> int64(-9223372036854775808)

ans =

 -9223372036854775808

>> int64(-9223372036854775808) - 1

ans =

 -9223372036854775808
```

At first blush, this seems reasonable enough since 9223372036854775807 is much closer to 9223372036854775808 than -9223372036854775808 is and integers are still represented with a fixed size in a natural way that is compatible with C and Fortran. Saturated integer arithmetic, however, is deeply problematic. The first and most obvious issue is that this is not the way machine integer arithmetic works, so implementing saturated operations requires emitting instructions after each machine integer operation to check for underflow or overflow and replace the result with `typemin(Int)` or `typemax(Int)` as appropriate. This alone expands each integer operation from a single, fast instruction into half a dozen instructions, probably including branches. Ouch. But it gets worse – saturating integer arithmetic isn't associative. Consider this Matlab computation:

```
>> n = int64(2)^62
4611686018427387904

>> n + (n - 1)
9223372036854775807

>> (n + n) - 1
9223372036854775806
```

This makes it hard to write many basic integer algorithms since a lot of common techniques depend on the fact that machine addition with overflow *is* associative. Consider finding the midpoint between integer values `lo` and `hi` in Julia using the expression `(lo + hi) >>> 1`:

```
julia> n = 2^62
4611686018427387904

julia> (n + 2n) >>> 1
6917529027641081856
```

See? No problem. That's the correct midpoint between 2^62 and 2^63, despite the fact that `n + 2n` is -4611686018427387904. Now try it in Matlab:

```
>> (n + 2*n)/2

ans =

  4611686018427387904
```

Oops. Adding a >>> operator to Matlab wouldn't help, because saturation that occurs when adding `n` and `2n` has already destroyed the information necessary to compute the correct midpoint.

Not only is lack of associativity unfortunate for programmers who cannot rely it for techniques like this, but it also

defeats almost anything compilers might want to do to optimize integer arithmetic. For example, since Julia integers use normal machine integer arithmetic, LLVM is free to aggressively optimize simple little functions like `f(k) = 5k-1`. The machine code for this function is just this:

```
julia> code_native(f,(Int,))
    .section    __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 1
    push    RBP
    mov RBP, RSP
Source line: 1
    lea RAX, QWORD PTR [RDI + 4*RDI - 1]
    pop RBP
    ret
```

The actual body of the function is a single `lea` instruction, which computes the integer multiply and add at once. This is even more beneficial when `f` gets inlined into another function:

```
julia> function g(k,n)
         for i = 1:n
           k = f(k)
         end
         return k
       end
g (generic function with 2 methods)

julia> code_native(g,(Int,Int))
    .section    __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 3
    push    RBP
    mov RBP, RSP
    test    RSI, RSI
    jle 22
    mov EAX, 1
Source line: 3
    lea RDI, QWORD PTR [RDI + 4*RDI - 1]
    inc RAX
    cmp RAX, RSI
Source line: 2
    jle -17
Source line: 5
    mov RAX, RDI
    pop RBP
    ret
```

Since the call to `f` gets inlined, the loop body ends up being just a single `lea` instruction. Next, consider what happens if we make the number of loop iterations fixed:

```
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,))
```

```
    .section    __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 3
    push    RBP
    mov RBP, RSP
Source line: 3
    imul    RAX, RDI, 9765625
    add RAX, -2441406
Source line: 5
    pop RBP
    ret
```

Because the compiler knows that integer addition and multiplication are associative and that multiplication distributes over addition – neither of which is true of saturating arithmetic – it can optimize the entire loop down to just a multiply and an add. Saturated arithmetic completely defeats this kind of optimization since associativity and distributivity can fail at each loop iteration, causing different outcomes depending on which iteration the failure occurs in. The compiler can unroll the loop, but it cannot algebraically reduce multiple operations into fewer equivalent operations.

Saturated integer arithmetic is just one example of a really poor choice of language semantics that completely prevents effective performance optimization. There are many things that are difficult about C programming, but integer overflow is *not* one of them – especially on 64-bit systems. If my integers really might get bigger than 2^63-1, I can easily predict that. Am I looping over a number of actual things that are stored in the computer? Then it's not going to get that big. This is guaranteed, since I don't have that much memory. Am I counting things that occur in the real world? Unless they're grains of sand or atoms in the universe, 2^63-1 is going to be plenty big. Am I computing a factorial? Then sure, they might get that big – I should use a `BigInt`. See? Easy to distinguish.

How do "abstract" or ambiguous fields in types interact with the compiler?

Types can be declared without specifying the types of their fields:

```
julia> type MyAmbiguousType
           a
       end
```

This allows `a` to be of any type. This can often be useful, but it does have a downside: for objects of type `MyAmbiguousType`, the compiler will not be able to generate high-performance code. The reason is that the compiler uses the types of objects, not their values, to determine how to build code. Unfortunately, very little can be inferred about an object of type `MyAmbiguousType`:

```
julia> b = MyAmbiguousType("Hello")
MyAmbiguousType("Hello")

julia> c = MyAmbiguousType(17)
MyAmbiguousType(17)

julia> typeof(b)
MyAmbiguousType (constructor with 1 method)

julia> typeof(c)
MyAmbiguousType (constructor with 1 method)
```

`b` and `c` have the same type, yet their underlying representation of data in memory is very different. Even if you stored just numeric values in field `a`, the fact that the memory representation of a `Uint8` differs from a `Float64` also means that the CPU needs to handle them using two different kinds of instructions. Since the required information is not available in the type, such decisions have to be made at run-time. This slows performance.

You can do better by declaring the type of a. Here, we are focused on the case where a might be any one of several types, in which case the natural solution is to use parameters. For example:

```
julia> type MyType{T<:FloatingPoint}
         a::T
       end
```

This is a better choice than

```
julia> type MyStillAmbiguousType
         a::FloatingPoint
       end
```

because the first version specifies the type of a from the type of the wrapper object. For example:

```
julia> m = MyType(3.2)
MyType{Float64}(3.2)

julia> t = MyStillAmbiguousType(3.2)
MyStillAmbiguousType(3.2)

julia> typeof(m)
MyType{Float64} (constructor with 1 method)

julia> typeof(t)
MyStillAmbiguousType (constructor with 2 methods)
```

The type of field a can be readily determined from the type of m, but not from the type of t. Indeed, in t it's possible to change the type of field a:

```
julia> typeof(t.a)
Float64

julia> t.a = 4.5f0
4.5f0

julia> typeof(t.a)
Float32
```

In contrast, once m is constructed, the type of m.a cannot change:

```
julia> m.a = 4.5f0
4.5

julia> typeof(m.a)
Float64
```

The fact that the type of m.a is known from m's type—coupled with the fact that its type cannot change mid-function—allows the compiler to generate highly-optimized code for objects like m but not for objects like t.

Of course, all of this is true only if we construct m with a concrete type. We can break this by explicitly constructing it with an abstract type:

```
julia> m = MyType{FloatingPoint}(3.2)
MyType{FloatingPoint}(3.2)

julia> typeof(m.a)
Float64
```

```
julia> m.a = 4.5f0
4.5f0

julia> typeof(m.a)
Float32
```

For all practical purposes, such objects behave identically to those of `MyStillAmbiguousType`.

It's quite instructive to compare the sheer amount code generated for a simple function

```
func(m::MyType) = m.a+1
```

using

```
code_llvm(func,(MyType{Float64},))
code_llvm(func,(MyType{FloatingPoint},))
code_llvm(func,(MyType,))
```

For reasons of length the results are not shown here, but you may wish to try this yourself. Because the type is fully-specified in the first case, the compiler doesn't need to generate any code to resolve the type at run-time. This results in shorter and faster code.

如何声明"抽象容器类型"的域

The same best practices that apply in the *previous section* also work for container types:

```
julia> type MySimpleContainer{A<:AbstractVector}
         a::A
       end

julia> type MyAmbiguousContainer{T}
         a::AbstractVector{T}
       end
```

For example:

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

julia> typeof(c)
MySimpleContainer{UnitRange{Int64}} (constructor with 1 method)

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

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

julia> b = MyAmbiguousContainer(1:3);

julia> typeof(b)
MyAmbiguousContainer{Int64} (constructor with 1 method)

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

julia> typeof(b)
MyAmbiguousContainer{Int64} (constructor with 1 method)
```

For `MySimpleContainer`, the object is fully-specified by its type and parameters, so the compiler can generate optimized functions. In most instances, this will probably suffice.

While the compiler can now do its job perfectly well, there are cases where *you* might wish that your code could do different things depending on the *element type* of `a`. Usually the best way to achieve this is to wrap your specific operation (here, `foo`) in a separate function:

```
function sumfoo(c::MySimpleContainer)
    s = 0
    for x in c.a
        s += foo(x)
    end
    s
end

foo(x::Integer) = x
foo(x::FloatingPoint) = round(x)
```

This keeps things simple, while allowing the compiler to generate optimized code in all cases.

However, there are cases where you may need to declare different versions of the outer function for different element types of `a`. You could do it like this:

```
function myfun{T<:FloatingPoint}(c::MySimpleContainer{Vector{T}})
    ...
end
function myfun{T<:Integer}(c::MySimpleContainer{Vector{T}})
    ...
end
```

This works fine for `Vector{T}`, but we'd also have to write explicit versions for `UnitRange{T}` or other abstract types. To prevent such tedium, you can use two parameters in the declaration of `MyContainer`:

```
type MyContainer{T, A<:AbstractVector}
    a::A
end
MyContainer(v::AbstractVector) = MyContainer{eltype(v), typeof(v)}(v)

julia> b = MyContainer(1.3:5);

julia> typeof(b)
MyContainer{Float64,UnitRange{Float64}}
```

Note the somewhat surprising fact that `T` doesn't appear in the declaration of field `a`, a point that we'll return to in a moment. With this approach, one can write functions such as:

```
function myfunc{T<:Integer, A<:AbstractArray}(c::MyContainer{T,A})
    return c.a[1]+1
end
# Note: because we can only define MyContainer for
# A<:AbstractArray, and any unspecified parameters are arbitrary,
# the previous could have been written more succinctly as
#     function myfunc{T<:Integer}(c::MyContainer{T})

function myfunc{T<:FloatingPoint}(c::MyContainer{T})
    return c.a[1]+2
end

function myfunc{T<:Integer}(c::MyContainer{T,Vector{T}})
```

```
    return c.a[1]+3
end

julia> myfunc(MyContainer(1:3))
2

julia> myfunc(MyContainer(1.0:3))
3.0

julia> myfunc(MyContainer([1:3]))
4
```

As you can see, with this approach it's possible to specialize on both the element type `T` and the array type `A`.

However, there's one remaining hole: we haven't enforced that `A` has element type `T`, so it's perfectly possible to construct an object like this:

```
julia> b = MyContainer{Int64, UnitRange{Float64}}(1.3:5);

julia> typeof(b)
MyContainer{Int64,UnitRange{Float64}}
```

To prevent this, we can add an inner constructor:

```
type MyBetterContainer{T<:Real, A<:AbstractVector}
    a::A

    MyBetterContainer(v::AbstractVector{T}) = new(v)
end
MyBetterContainer(v::AbstractVector) = MyBetterContainer{eltype(v),typeof(v)}(v)


julia> b = MyBetterContainer(1.3:5);

julia> typeof(b)
MyBetterContainer{Float64,UnitRange{Float64}}

julia> b = MyBetterContainer{Int64, UnitRange{Float64}}(1.3:5);
ERROR: no method MyBetterContainer(UnitRange{Float64},)
```

The inner constructor requires that the element type of `A` be `T`.

Nothingness and missing values

How does "null" or "nothingness" work in Julia?

Unlike many languages (for example, C and Java), Julia does not have a "null" value. When a reference (variable, object field, or array element) is uninitialized, accessing it will immediately throw an error. This situation can be detected using the `isdefined` function.

Some functions are used only for their side effects, and do not need to return a value. In these cases, the convention is to return the value `nothing`, which is just a singleton object of type `Nothing`. This is an ordinary type with no fields; there is nothing special about it except for this convention, and that the REPL does not print anything for it. Some language constructs that would not otherwise have a value also yield `nothing`, for example `if false; end`.

Note that `Nothing` (uppercase) is the type of `nothing`, and should only be used in a context where a type is required (e.g. a declaration).

You may occasionally see `None`, which is quite different. It is the empty (or "bottom") type, a type with no values and no subtypes (except itself). You will generally not need to use this type.

The empty tuple (`()`) is another form of nothingness. But, it should not really be thought of as nothing but rather a tuple of zero values.

Julia 发行版

Do I want to use a release, beta, or nightly version of Julia?

You may prefer the release version of Julia if you are looking for a stable code base. Releases generally occur every 6 months, giving you a stable platform for writing code.

You may prefer the beta version of Julia if you don't mind being slightly behind the latest bugfixes and changes, but find the slightly faster rate of changes more appealing. Additionally, these binaries are tested before they are published to ensure they are fully functional.

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.

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.

Links to each of these download types can be found on the download page at http://julialang.org/downloads/. Note that not all versions of Julia are available for all platforms.

何时移除舍弃的函数?

Deprecated functions are removed after the subsequent release. For example, functions marked as deprecated in the 0.1 release will not be available starting with the 0.2 release.

开发 Julia

How do I debug julia's C code? (running the julia REPL from within a debugger like gdb)

First, you should build the debug version of julia with `make debug`. Below, lines starting with `(gdb)` mean things you should type at the gdb prompt.

From the shell

The main challenge is that Julia and gdb each need to have their own terminal, to allow you to interact with them both. One approach is to use gdb's `attach` functionality to debug an already-running julia session. However, on many systems you'll need root access to get this to work. What follows is a method that can be implemented with just user-level permissions.

The first time you do this, you'll need to define a script, here called `oterm`, containing the following lines:

```
ps
sleep 600000
```

Make it executable with `chmod +x oterm`.

Now:

- From a shell (called shell 1), type `xterm -e oterm &`. You'll see a new window pop up; this will be called terminal 2.
- From within shell 1, `gdb julia-debug`. You can find this executable within `julia/usr/bin`.
- From within shell 1, `(gdb) tty /dev/pts/#` where `#` is the number shown after `pts/` in terminal 2.
- From within shell 1, `(gdb) run`
- From within terminal 2, issue any preparatory commands in Julia that you need to get to the step you want to debug
- From within shell 1, hit Ctrl-C
- From within shell 1, insert your breakpoint, e.g., `(gdb) b codegen.cpp:2244`
- From within shell 1, `(gdb) c` to resume execution of julia
- From within terminal 2, issue the command that you want to debug. Shell 1 will stop at your breakpoint.

Within emacs

- `M-x gdb`, then enter `julia-debug` (this is easiest from within julia/usr/bin, or you can specify the full path)
- `(gdb) run`
- Now you'll see the Julia prompt. Run any commands in Julia you need to get to the step you want to debug.
- Under emacs' "Signals" menu choose BREAK—this will return you to the `(gdb)` prompt
- Set a breakpoint, e.g., `(gdb) b codegen.cpp:2244`
- Go back to the Julia prompt via `(gdb) c`
- Execute the Julia command you want to see running.

CHAPTER 29

与其它语言的区别

与 MATLAB 的区别

Julia 的语法和 MATLAB 很像。但 Julia 不是简单地复制 MATLAB，它们有很多句法和功能上的区别。以下是一些值得注意的区别：

- 数组用方括号来索引， `A[i,j]`
- 数组是用引用来赋值的。在 `A=B` 之后，对 `B` 赋值也会修改 `A`
- 使用引用来传递和赋值。如果一个函数修改了数组，调用函数会发现值也变了
- Matlab 把赋值和分配内存合并成了一个语句。比如： `a(4) = 3.2` 会创建一个数组 `a = [0 0 0 3.2]`，即为a分配了内存并且将每个元素初始化为0,然后为第四个元素赋值3.2，而 `a(5) = 7` 会为数组a增加长度，并且给第五个元素赋值7。 Julia 把赋值和分配内存分开了： 如果 `a` 长度为4, `a[5] = 7` 会抛出一个错误。 Julia 有一个专用的 `push!` 函数来向 `Vectors` 里增加元素。并且远比Matlab的 `a(end+1) = val` 来的高效。
- 虚数单位 `sqrt(-1)` 用 `im` 来表示
- Literal numbers without a decimal point (such as `42`) create integers instead of floating point numbers. Arbitrarily large integer literals are supported. But this means that some operations such as `2^-1` will throw a domain error as the result is not an integer (see *the FAQ entry on domain errors* for details).
- 多返回值和多重赋值需要使用圆括号，如 `return (a, b)` 和 `(a, b) = f(x)`
- Julia 有一维数组。列向量的长度为 `N`，而不是 `Nx1`。例如， `rand(N)` 生成的是一维数组
- 使用语法 `[x,y,z]` 来连接标量或数组，连接发生在第一维度（“垂直”）上。对于第二维度（“水平”）上的连接，需要使用空格，如 `[x y z]`。要想构造块矩阵，尽量使用语法 `[a b; c d]`
- `a:b` 和 `a:b:c` 中的冒号，用来构造 `Range` 对象。使用 `linspace` 构造一个满向量，或者通过使用方括号来“连接”范围，如 `[a:b]`
- 函数返回须使用 `return` 关键字，而不是把它们列在函数定义中（详见 *return* 关键字）
- 一个文件可以包含多个函数，文件被载入时，所有的函数定义都是外部可见的
- `sum`, `prod`, `max` 等约简操作，如果被调用时参数只有一个，作用域是数组的所有元素，如 `sum(A)`

- `sort` 等函数，默认按列方向操作。（`sort(A)` 等价于 `sort(A,1)`）。要想排序 `1xN` 的矩阵，使用 `sort(A,2)`
- 如果 `A` 是 2 维数组, `fft(A)` 计算的是 2 维 FFT. 尤其注意的是, 它不等效于 `fft(A,1)`, 后者计算的是按列的 1 维 FFT.
- 即使是无参数的函数，也要使用圆括号，如 `tic()` 和 `toc()`
- 表达式结尾不要使用分号。表达式的结果不会自动显示（除非在交互式提示符下）。`println` 函数可以用来打印值并换行
- 若 `A` 和 `B` 是数组， `A == B` 并不返回布尔值数组。应该使用 `A .== B` 。其它布尔值运算符可以类比， `<`, `>`, `!=` 等
- 符号 `&` 、`|` 和 `$` 表示位运算“和”、“或”以及“异或”。它们和python中的位运算符有着相同的运算符优先级，和c语言中的位运算符优先级并不一样。 它们能被应用在标量上或者应用在两个数组间（对每个相同位置的元素分别进行逻辑运算，返回一个由结果组成的新数组）。 值得注意的是它们的运算符优先级，别忘了括号: 如果想要判断变量 `A` 是等于1还是2, 要这样写 `(A .== 1) | (A .== 2)` 。
- 可以用 `...` 把集合中的元素作为参数传递给函数，如 `xs=[1,2]; f(xs...)`
- Julia 中 `svd` 返回的奇异值是向量而不是完整的对角矩阵
- Julia 中 `...` 不用来将一行代码拆成多行。Instead, incomplete expressions automatically continue onto the next line.
- 变量 `ans` 是交互式会话中执行的最后一条表达式的值；以其它方式执行的表达式的值，不会赋值给它
- The closest analog to Julia's `types` are Matlab's `classes`. Matlab's `structs` behave somewhere between Julia's `types` and `Dicts`; in particular, if you need to be able to add fields to a `struct` on-the-fly, use a `Dict` rather than a `type`.

与 R 的区别

Julia 也想成为数据分析和统计编程的高效语言。与 R 的区别：

- 使用 = 赋值，不提供 `<-` 或 `<<-` 等箭头式运算符
- 用方括号构造向量。Julia 中 `[1, 2, 3]` 等价于 R 中的 `c(1, 2, 3)`
- Julia 的矩阵运算比 R 更接近传统数学语言。如果 `A` 和 `B` 是矩阵，那么矩阵乘法在 Julia 中为 `A * B`，R 中为 `A %*% B` 。在 R 中，第一个语句表示的是逐元素的 Hadamard 乘法。要进行逐元素点乘，Julia 中为 `A .* B`
- 使用 `'` 运算符做矩阵转置。 Julia 中 `A'` 等价于 R 中 `t(A)`
- 写 `if` 语句或 `for` 循环时不需要写圆括号：应写 `for i in [1, 2, 3]` 而不是 `for (i in c(1, 2, 3))` ；应写 `if i == 1` 而不是 `if (i == 1)`
- `0` 和 `1` 不是布尔值。不能写 `if (1)` ，因为 `if` 语句仅接受布尔值作为参数。应写成 `if true`
- 不提供 `nrow` 和 `ncol` 。应该使用 `size(M, 1)` 替代 `nrow(M)` ；使用 `size(M, 2)` 替代 `ncol(M)`
- Julia 的 SVD 默认为非 thinned ，与 R 不同。要得到与 R 一样的结果，应该对矩阵 `X` 调用 `svd(X, true)`
- Julia 区分标量、向量和矩阵。在 R 中， `1` 和 `c(1)` 是一样的。在 Julia 中，它们完全不同。例如若 `x` 和 `y` 为向量，则 `x' * y` 是一个单元素向量，而不是标量。要得到标量，应使用 `dot(x, y)`
- Julia 中的 `diag()` 和 `diagm()` 与 R 中的不同
- Julia 不能在赋值语句左侧调用函数：不能写 `diag(M) = ones(n)`

- Julia 不赞成把 main 命名空间塞满函数。大多数统计学函数可以在 扩展包 中找到，比如 DataFrames 和 Distributions 包：
  - Distributions 包 提供了概率分布函数.
  - DataFrames 包 提供了数据框架
  - GLM 扩展包 提供了广义的线性模型.
- Julia 提供了多元组和哈希表，但不提供 R 的列表。当返回多项时，应该使用多元组：不要使用 `list(a = 1, b = 2)`，应该使用 `(1, 2)`
- 鼓励自定义类型。Julia 的类型比 R 中的 S3 或 S4 对象简单。Julia 的重载系统使 `table(x::TypeA)` 和 `table(x::TypeB)` 等价于 R 中的 `table.TypeA(x)` 和 `table.TypeB(x)`
- 在 Julia 中，传递值和赋值是靠引用。如果一个函数修改了数组，调用函数会发现值也变了。这与 R 非常不同，这使得在大数据结构上进行新函数操作非常高效
- 使用 `hcat` 和 `vcat` 来连接向量和矩阵，而不是 `c`, `rbind` 和 `cbind`
- Julia 的范围对象如 `a:b` 与 R 中的定义向量的符号不同。它是一个特殊的对象，用于低内存开销的迭代。要把范围对象转换为向量，应该用方括号把范围对象括起来 `[a:b]`
- `max`, `min` are the equivalent of `pmax` and `pmin` in R, but both arguments need to have the same dimensions. While `maximum`, `minimum` replace `max` and `min` in R, there are important differences.
- The functions `sum`, `prod`, `maximum`, `minimum` are different from their counterparts in R. They all accept one or two arguments. The first argument is an iterable collection such as an array. If there is a second argument, then this argument indicates the dimensions, over which the operation is carried out. For instance, let `A=[[1 2],[3,4]]` in Julia and `B=rbind(c(1,2),c(3,4))` be the same matrix in R. Then `sum(A)` gives the same result as `sum(B)`, but `sum(A,1)` is a row vector containing the sum over each column and `sum(A,2)` is a column vector containing the sum over each row. This contrasts to the behavior of R, where `sum(B,1)=11` and `sum(B,2)=12`. If the second argument is a vector, then it specifies all the dimensions over which the sum is performed, e.g., `sum(A,[1,2])=10`. It should be noted that there is no error checking regarding the second argument.
- Julia 有许多函数可以修改它们的参数。例如， `sort(v)` 和 `sort!(v)` 函数中，带感叹号的可以修改 `v`
- `colMeans()` 和 `rowMeans()`, `size(m, 1)` 和 `size(m, 2)`
- 在 R 中，需要向量化代码来提高性能。在 Julia 中与之相反：使用非向量化的循环通常效率最高
- 与 R 不同，Julia 中没有延时求值
- 不提供 `NULL` 类型
- Julia 中没有与 R 的 `assign` 或 `get` 所等价的语句

与 Python 的区别

- 对数组、字符串等索引。Julia 索引的下标是从 1 开始，而不是从 0 开始
- 索引列表和数组的最后一个元素时，Julia 使用 `end`， Python 使用 -1
- Julia 中的 Comprehensions （还）没有条件 if 语句
- for, if, while, 等块的结尾需要 `end`； 不强制要求缩进排版
- Julia 没有代码分行的语法：如果在一行的结尾，输入已经是个完整的表达式，就直接执行；否则就继续等待输入。强迫 Julia 的表达式分行的方法是用圆括号括起来

- Julia 总是以列为主序的（类似 Fortran ），而 *numpy* 数组默认是以行为主序的（类似 C ）。如果想优化遍历数组的性能，从 *numpy* 到 Julia 时应改变遍历的顺序（详见 代码性能优化 ）。

Part II

笔记

此笔记不是官方文档，没有任何人为此文档的时效性、准确性等负责。

CHAPTER 30

宏

在这儿列几个推荐的宏

@time

@time 宏可打印出程序运行所需的时间。可用于单行表达式或代码块:

```
julia> @time sqrtm( sum( randn(1000) ) )
elapsed time: 0.0 seconds
5.128676446664007

julia> @time for i=1:1000
                sqrtm(sum(randn(1000)))
            end
elapsed time: 0.04699993133544922 seconds
```

这个宏适合在优化代码等情况时使用。

注: @time expr 与tic(); expr; toc();等价。

@which

@which 宏可以显示对于指定的表达式，Julia调用的是哪个具体的方法来求值:

```
julia> @which rand(1:10)
rand{T<:Integer}(Range1{T<:Integer},) at random.jl:145

julia> @which 1 + 2im
+(Number,Number) at promotion.jl:135

julia> a = [ 1 2 3 ; 4 5 6]
2x3 Int32 Array:
```

```
1  2  3
4  5  6

julia> @which a[2,3]
ref(Array{T,N},Real,Real) at array.jl:246
```

这个宏比较方便 debug 时使用。比使用 `which()` 函数方便。

@printf

@printf 宏用于输出指定格式的字符串，它区别于C里的printf函数，Julia的printf 对所要输出的内容会做相应的优化，使用上和C的printf的格式相差不大:

```
julia> @printf("Hello World!")
Hello World!

julia> A = [1 2 3;4 5 6]
2x3 Array{Int64,2}:
1  2  3
4  5  6

julia> @printf("A has %d elements.",length(a))
A has 6 elements.

julia> @printf("sqrtm(2) is %4.2f.",sqrtm(2))
sqrtm(2) is 1.41.
```

@inbounds

@inbounds 宏作用主要是提高运行速度，不过它是以牺牲程序中数组的bound check为代价. 一般来说速度会有相当的提升:

```
julia>  function f(x)
        N = div(length(x),3)
        r = 0.0
        for i = 1:N
            for j = 1:N
                for k = 1: N
                    r += x[i+2*j]+x[j+2*k]+x[k+2*i]
                end
            end
        end
        r
        end
f (generic function with 1 method)

julia>  function f1(x)
        N = div(length(x),3)
        r = 0.0
        for i = 1:N
            for j = 1:N
                for k = 1: N
                    @inbounds r += x[i+2*j]+x[j+2*k]+x[k+2*i]
```

```
                end
            end
        end
        r
        end
f1 (generic function with 1 method)

julia> a = rand(2000);

julia> @time f(a)
elapsed time: 0.528003738 seconds (211264 bytes allocated)
4.4966644948005027e8

julia> @time f1(a)
elapsed time: 0.307557441 seconds (64 bytes allocated)
4.4966644948005027e8
```

@assert

@assert 宏和C里的用法类似，当表达式的求值非真将显示错误,如果表达式值为真则什么都不返回，一般用于处理代码中的错误:

```
julia> @assert 1==2
ERROR: assertion failed: 1 == 2
 in error at error.jl:21

julia> @assert 1.000000000000001==1.0
ERROR: assertion failed: 1.000000000000001 == 1.0
 in error at error.jl:21

julia> @assert 1.0000000000000001==1.0
```

注：最后这个实例在x64上通过。一般不用assert做测试。做测试的时候一般习惯使用 `@test` 宏， 使用前需要声明 `using Base.Test` 。

@goto & @label

@goto 宏作用和C里的goto一致， 因为goto的滥用会导致程序可读性降低，所以现在的goto基本上只推荐用于跳出嵌套循环

```
julia>  function f()
        i = 0
        while (i < 10)
            @printf("%d\n", i)
            for j = 1:5
                if i > 5
                    @goto loop_end
                end
            end
        end
        @label loop_end
        end
julia>  f()
```

```
0
1
2
3
4
5
```

CHAPTER 31

心得

数字分隔符

数字分隔符可以使数字位数更加清晰：

```
julia> a = 100_000
100000

julia> a = 100_0000_0000
10000000000
```

建议大家在写多位数字的时候，使用数字分隔符。

CHAPTER 32

优化代码

使用宏 @inbounds 加速循环

@inbounds 以牺牲程序的安全性来大幅度加快循环的执行效率。 在对数组结构循环的时候，使用该宏需要确保程序的正确性。

节约空间

Julia提供了很多inplace的函数，合理地使用能够避免分配不必要的空间，从而减少gc的调用且节约了分配空间的时间。比如在数值上的共轭梯度法，其中的矩阵乘法可以使用 A_mul_B! 而不是 *(A,x)，这样避免了每次进入循环的时候分配新的空间

```
julia> A = rand(5000,5000); x = rand(5000); y = rand(5000);
julia> @time y = A*x;
elapsed time: 0.016998118 seconds (40168 bytes allocated)
julia> @time A_mul_B!(y,A,x);
elapsed time: 0.012711399 seconds (80 bytes allocated)
```

手动优化循环(manual loop unrolling)

手动地将循环平铺开能削减循环带来的overhead， 因为每次进入循环都会进行 end-of-loop 检测， 手动增加一些代码会加速循环的运行。缺点是会产生更多的代码，不够简洁，尤其是在循环内部调用inline函数，反而有可能降低性能。一个比较好的例子是 Julia 的sum函数。这里只做个简单的对比

```
function simple_sum(a::AbstractArray, first::Int, last::Int)
    b = a[first];
    for i = first + 1 : last
        @inbounds b += a[i]
    end
```

```
    return b
end

function unroll_sum(a::AbstractArray, first::Int, last::Int)
    @inbounds if ifirst + 6 >= ilast # length(a) < 8
        i = ifirst
        s =  a[i] + a[i+1]
        i = i+1
        while i < ilast
            s +=a[i+=1]
        end
        return s

    else # length(a) >= 8, manual unrolling
        @inbounds s1 = a[ifirst] +  a[ifirst + 4]
        @inbounds s2 = a[ifirst + 1] + a[ifirst + 5]
        @inbounds s3 = a[ifirst + 2] +  a[ifirst + 6]
        @inbounds s4 = a[ifirst + 3] +  a[ifirst + 7]
        i = ifirst + 8
        il = ilast - 3
        while i <= il
          @inbounds  s1 +=  a[i]
          @inbounds  s2 += a[i+1]
          @inbounds  s3 += a[i+2]
          @inbounds  s4 += a[i+3]
            i += 4
        end
        while i <= ilast
          @inbounds  s1 += a[i]
            i += 1
        end
        return s1 + s2 + s3 + s4
    end
end
```

运行结果如下

```
julia>rand_arr = rand(500000);
julia>@time @inbounds ret_1 = simple_sum(rand_arr, 1, 500000)
elapsed time: 0.000699786 seconds (160 bytes allocated)
julia>@time @inbounds ret_2= unroll_sum(rand_arr,1,500000)
elapsed time: 0.000383906 seconds (96 bytes allocated)
```

调用C或Fortran加速

尽管Julia声称速度和C相提并论，但是并不是所有的情况下都能有C的性能。合理地使用像LAPACK,BLAS,MUMPS这类已经高度优化的函数库有助于提升运行速度。

尽量使用一维数组

多维数组相当于多重指针，读取需要多次读地址，用一维数组能节约读取时间，但注意一维数组的排列，Julia和MATLAB以及Fortran一样都是列优先储存。对于可以并行的一维数组操作，Julia提供了宏 `@simd` (0.3.0，可能会在以后版本中取消)。

良好的编程习惯

这是对Julia官方手册的一句话总结，包括声明变量类型，保持函数的健壮性，循环前预分配空间等等。

Part III

The Julia Standard Library

CHAPTER 33

The Standard Library

Introduction

The Julia standard library contains a range of functions and macros appropriate for performing scientific and numerical computing, but is also as broad as those of many general purpose programming languages. Additional functionality is available from a growing collection of available packages. Functions are grouped by topic below.

Some general notes:

- Except for functions in built-in modules (`Pkg`, `Collections`, `Graphics`, `Test` and `Profile`), all functions documented here are directly available for use in programs.
- To use module functions, use `import Module` to import the module, and `Module.fn(x)` to use the functions.
- Alternatively, `using Module` will import all exported `Module` functions into the current namespace.
- By convention, function names ending with an exclamation point (`!`) modify their arguments. Some functions have both modifying (e.g., `sort!`) and non-modifying (`sort`) versions.

Getting Around

`exit`(*[code]*)
Quit (or control-D at the prompt). The default exit code is zero, indicating that the processes completed successfully.

`quit`()
Quit the program indicating that the processes completed succesfully. This function calls `exit(0)` (see `exit()`).

`atexit`(*f*)
Register a zero-argument function to be called at exit.

isinteractive() → Bool
Determine whether Julia is running an interactive session.

whos(*[Module,] [pattern::Regex]*)
Print information about global variables in a module, optionally restricted to those matching `pattern`.

edit(*file::String*[, *line*])
Edit a file optionally providing a line number to edit at. Returns to the julia prompt when you quit the editor.

edit(*function*[, *types*])
Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit.

@edit()
Evaluates the arguments to the function call, determines their types, and calls the `edit` function on the resulting expression

less(*file::String*[, *line*])
Show a file using the default pager, optionally providing a starting line number. Returns to the julia prompt when you quit the pager.

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.

@less()
Evaluates the arguments to the function call, determines their types, and calls the `less` function on the resulting expression

clipboard(*x*)
Send a printed form of `x` to the operating system clipboard ("copy").

clipboard() → String
Return a string with the contents of the operating system clipboard ("paste").

require(*file::String...*)
Load source files once, 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 in the current working directory, then looks for package code under `Pkg.dir()`, then tries paths in the global array `LOAD_PATH`.

reload(*file::String*)
Like `require`, except forces loading of files regardless of whether they have been loaded before. Typically used when interactively developing libraries.

include(*path::String*)
Evaluate the contents of a source file in the current context. 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. All paths refer to files on node 1 when running in parallel, and files will be fetched from node 1. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

include_string(*code::String*)
Like `include`, except reads code from the given string rather than from a file. Since there is no file path involved, no path processing or fetching from node 1 is done.

help(*name*)
Get help for a function. `name` can be an object or a string.

apropos(*string*)
Search documentation for functions related to `string`.

which(*f*, *types*)
Return the method of `f` (a `Method` object) that will be called for arguments with the given types.

@which()
Evaluates the arguments to the function call, determines their types, and calls the `which` function on the resulting expression

methods(*f*[, *types*])
Show all methods of `f` with their argument types.

If `types` is specified, an array of methods whose types match is returned.

methodswith(*typ*[, *showparents*])
Return an array of methods with an argument of type `typ`. If optional `showparents` is `true`, also return arguments with a parent type of `typ`, excluding type `Any`.

@show()
Show an expression and result, returning the result

versioninfo([*verbose::Bool*])
Print information about the version of Julia in use. If the `verbose` argument is true, detailed system information is shown as well.

workspace()
Replace the top-level module (`Main`) with a new one, providing a clean workspace. The previous `Main` module is made available as `LastMain`. A previously-loaded package can be accessed using a statement such as `using LastMain.Package`.

This function should only be used interactively.

All Objects

is(*x*, *y*) → Bool
Determine whether `x` and `y` are identical, in the sense that no program could distinguish them. Compares mutable objects by address in memory, and compares immutable objects (such as numbers) by contents at the bit level. This function is sometimes called `egal`. The `===` operator is an alias for this function.

isa(*x*, *type*) → Bool
Determine whether `x` is of the given `type`.

isequal(*x*, *y*)
Similar to `==`, except treats all floating-point `NaN` values as equal to each other, and treats `-0.0` as unequal to `0.0`. For values that are not floating-point, `isequal` is the same as `==`.

`isequal` is the comparison function used by hash tables (`Dict`). `isequal(x,y)` must imply that `hash(x) == hash(y)`.

Collections typically implement `isequal` by calling `isequal` recursively on all contents.

Scalar types generally do not need to implement `isequal`, unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on `isnan`, `signbit`, and `==`).

isless(*x*, *y*)
Test whether `x` is less than `y`, according to a canonical total order. Values that are normally unordered, such as `NaN`, are ordered in an arbitrary but consistent fashion. This is the default comparison used by `sort`. Non-numeric types with a canonical total order should implement this function. Numeric types only need to implement it if they have special values such as `NaN`.

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.

lexcmp(*x*, *y*)
Compare `x` and `y` lexicographically and return -1, 0, or 1 depending on whether `x` is less than, equal to, or greater than `y`, respectively. This function should be defined for lexicographically comparable types, and `lexless` will call `lexcmp` by default.

lexless(*x*, *y*)
Determine whether `x` is lexicographically less than `y`.

typeof(*x*)
Get the concrete type of `x`.

tuple(*xs...*)
Construct a tuple of the given objects.

ntuple(*n*, *f::Function*)
Create a tuple of length `n`, computing each element as `f(i)`, where `i` is the index of the element.

object_id(*x*)
Get a unique integer id for `x`. `object_id(x)==object_id(y)` if and only if `is(x,y)`.

hash(*x*[, *h*])
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.

finalizer(*x*, *function*)
Register a function `f(x)` to be called when there are no program-accessible references to `x`. The behavior of this function is unpredictable if `x` is of a bits type.

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.

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.

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::ObjectIdDict)` (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.

isdefined([*object*], *index* | *symbol*)
Tests whether an assignable location is defined. The arguments can be an array and index, a composite object and field name (as a symbol), or a module and a symbol. With a single symbol argument, tests whether a global variable with that name is defined in `current_module()`.

convert(*type*, *x*)
Try to convert `x` to the given type. Conversions from floating point to integer, rational to integer, and complex to real will raise an `InexactError` if `x` cannot be represented exactly in the new type.

promote(*xs...*)
Convert all arguments to their common promotion type (if any), and return them all (as a tuple).

oftype(*x*, *y*)
Convert `y` to the type of `x`.

widen(*type* | *x*)
If the argument is a type, return a "larger" type (for numeric types, this will be a type with at least as much range and precision as the argument, and usually more). Otherwise the argument `x` is converted to `widen(typeof(x))`.

```
julia> widen(Int32)
Int64
```

```
julia> widen(1.5f0)
1.5
```

identity(*x*)
The identity function. Returns its argument.

Types

super(*T::DataType*)
Return the supertype of DataType T

issubtype(*type1*, *type2*)
True if and only if all values of `type1` are also of `type2`. Can also be written using the `<:` infix operator as `type1 <: type2`.

<:(*T1*, *T2*)
Subtype operator, equivalent to `issubtype(T1,T2)`.

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.

subtypetree(*T::DataType*)
Return a nested list of all subtypes of DataType T. Note that all currently loaded subtypes are included, including those not visible in the current module.

typemin(*type*)
The lowest value representable by the given (real) numeric type.

typemax(*type*)
The highest value representable by the given (real) numeric type.

realmin(*type*)
The smallest in absolute value non-subnormal value representable by the given floating-point type

realmax(*type*)
The highest finite value representable by the given floating-point type

maxintfloat(*type*)
The largest integer losslessly representable by the given floating-point type

sizeof(*type*)
Size, in bytes, of the canonical binary representation of the given type, if any.

eps([*type*])
The distance between 1.0 and the next larger representable floating-point value of `type`. Only floating-point types are sensible arguments. If `type` is omitted, then `eps(Float64)` is returned.

eps(*x*)
The distance between `x` and the next larger representable floating-point value of the same type as `x`.

promote_type(*type1*, *type2*)
Determine a type big enough to hold values of each argument type without loss, whenever possible. In some cases, where no type exists which to which both types can be promoted losslessly, 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.

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.

getfield(*value*, *name::Symbol*)
Extract a named field from a value of composite type. The syntax `a.b` calls `getfield(a, :b)`, and the syntax `a.(b)` calls `getfield(a, b)`.

setfield!(*value*, *name::Symbol*, *x*)
Assign `x` to a named field in `value` of composite type. The syntax `a.b = c` calls `setfield!(a, :b, c)`, and the syntax `a.(b) = c` calls `setfield!(a, b, c)`.

fieldoffsets(*type*)
The byte offset of each field of a type relative to the data start. For example, we could use it in the following manner to summarize information about a struct type:

```
julia> structinfo(T) = [zip(fieldoffsets(T),names(T),T.types)...];

julia> structinfo(StatStruct)
12-element Array{(Int64,Symbol,DataType),1}:
 (0,:device,Uint64)
 (8,:inode,Uint64)
 (16,:mode,Uint64)
 (24,:nlink,Int64)
 (32,:uid,Uint64)
 (40,:gid,Uint64)
 (48,:rdev,Uint64)
 (56,:size,Int64)
 (64,:blksize,Int64)
 (72,:blocks,Int64)
 (80,:mtime,Float64)
 (88,:ctime,Float64)
```

fieldtype(*value*, *name::Symbol*)
Determine the declared type of a named field in a value of composite type.

isimmutable(*v*)
True if value `v` is immutable. See 不可变复合类型 for a discussion of immutability.

isbits(*T*)
True if `T` is a "plain data" type, meaning it is immutable and contains no references to other values. Typical examples are numeric types such as `Uint8`, `Float64`, and `Complex{Float64}`.

```
julia> isbits(Complex{Float64})
true

julia> isbits(Complex)
false
```

isleaftype(*T*)
Determine whether `T` is a concrete type that can have instances, meaning its only subtypes are itself and `None`

(but `T` itself is not `None`).

typejoin (*T*, *S*)
Compute a type that contains both `T` and `S`.

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.

Generic Functions

apply (*f*, *x...*)
Accepts a function and several arguments, each of which must be iterable. The elements generated by all the arguments are appended into a single list, which is then passed to `f` as its argument list.

```
julia> function f(x, y) # Define a function f
           x + y
       end;

julia> apply(f, [1 2]) # Apply f with 1 and 2 as arguments
3
```

`apply` is called to implement the `...` argument splicing syntax, and is usually not called directly: `apply(f, x) === f(x...)`

method_exists (*f*, *tuple*) → Bool
Determine whether the given generic function has a method matching the given tuple of argument types.

```
julia> method_exists(length, (Array,))
true
```

applicable (*f*, *args...*) → Bool
Determine whether the given generic function has a method applicable to the given arguments.

```
julia> function f(x, y)
           x + y
       end;

julia> applicable(f, 1)
false

julia> applicable(f, 1, 2)
true
```

invoke (*f*, *(types...)*, *args...*)
Invoke a method for the given generic function matching the specified types (as a tuple), on the specified arguments. The arguments must be compatible with the specified types. This 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).

|> (*x*, *f*)
Applies a function to the preceding argument. This allows for easy function chaining.

```
julia> [1:5] |> x->x.^2 |> sum |> inv
0.01818181818181818
```

Syntax

eval([*m::Module*], *expr::Expr*)
Evaluate an expression in the given module and return the result. Every module (except those defined with `baremodule`) has its own 1-argument definition of `eval`, which evaluates expressions in that module.

@eval()
Evaluate an expression and return the value.

evalfile(*path::String*)
Evaluate all expressions in the given file, and return the value of the last one. No other processing (path searching, fetching from node 1, etc.) is performed.

esc(*e::ANY*)
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 man-macros section of the Metaprogramming chapter of the manual for more details and examples.

gensym([*tag*])
Generates a symbol which will not conflict with other variable names.

@gensym()
Generates a gensym symbol for a variable. For example, `@gensym x y` is transformed into `x = gensym("x"); y = gensym("y")`.

parse(*str*, *start; greedy=true*, *raise=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. If `raise` is true (default), syntax errors will raise an error; otherwise, `parse` will return an expression that will raise an error upon evaluation.

parse(*str; raise=true*)
Parse the whole 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.

Iteration

Sequential iteration is implemented by the methods `start`, `done`, and `next`. The general `for` loop:

```
for i = I
  # body
end
```

is translated to:

```
state = start(I)
while !done(I, state)
  (i, state) = next(I, state)
  # body
end
```

The `state` object may be anything, and should be chosen appropriately for each iterable type.

start(*iter*) → state
Get initial iteration state for an iterable object

done(*iter*, *state*) → Bool
Test whether we are done iterating

next(*iter*, *state*) → item, state
For a given iterable object and iteration state, return the current item and the next iteration state

zip(*iters...*)
For a set of iterable objects, returns an iterable of tuples, where the `i`th tuple contains the `i`th component of each input iterable.

Note that `zip` is its own inverse: `[zip(zip(a...)...)...] == [a...]`.

enumerate(*iter*)
Return an iterator that yields `(i, x)` where `i` is an index 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 index `i` of the iterations.

```
julia> a = ["a", "b", "c"];

julia> for (index, value) in enumerate(a)
           println("$index $value")
       end
1 a
2 b
3 c
```

Fully implemented by: `Range`, `Range1`, `NDRange`, `Tuple`, `Real`, `AbstractArray`, `IntSet`, `ObjectIdDict`, `Dict`, `WeakKeyDict`, `EachLine`, `String`, `Set`, `Task`.

General Collections

isempty(*collection*) → Bool
Determine whether a collection is empty (has no elements).

```
julia> isempty([])
true

julia> isempty([1 2 3])
false
```

empty!(*collection*) → collection
Remove all elements from a `collection`.

length(*collection*) → Integer
For ordered, indexable collections, the maximum index `i` for which `getindex(collection, i)` is valid. For unordered collections, the number of elements.

endof(*collection*) → Integer
Returns the last index of the collection.

```
julia> endof([1,2,4])
3
```

Fully implemented by: `Range`, `Range1`, `Tuple`, `Number`, `AbstractArray`, `IntSet`, `Dict`, `WeakKeyDict`, `String`, `Set`.

Iterable Collections

in(*item*, *collection*) → Bool
Determine whether an item is in the given collection, in the sense that it is == to one of the values generated by iterating over the collection. Some collections need a slightly different definition; for example Sets check whether the item is `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)`.

eltype(*collection*)
Determine the type of the elements generated by iterating `collection`. For associative collections, this will be a `(key,value)` tuple type.

indexin(*a*, *b*)
Returns a vector containing the highest index in `b` for each value in `a` that is a member of `b` . The output vector contains 0 wherever `a` is not a member of `b`.

findin(*a*, *b*)
Returns the indices of elements in collection `a` that appear in collection `b`

unique(*itr*[, *dim*])
Returns an array containing only the unique elements of the iterable `itr`, in the order that the first of each set of equivalent elements originally appears. If `dim` is specified, returns unique regions of the array `itr` along `dim`.

reduce(*op*, *v0*, *itr*)
Reduce the given collection `ìtr` with the given binary operator. Reductions for certain commonly-used operators have special implementations which 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, and parallelism will also 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.

reduce(*op*, *itr*)
Like `reduce` but using the first element as v0.

foldl(*op*, *v0*, *itr*)
Like `reduce`, but with guaranteed left associativity.

foldl(*op*, *itr*)
Like `foldl`, but using the first element as v0.

foldr(*op*, *v0*, *itr*)
Like `reduce`, but with guaranteed right associativity.

foldr(*op*, *itr*)
Like `foldr`, but using the last element as v0.

maximum(*itr*)
Returns the largest element in a collection.

maximum(*A*, *dims*)
Compute the maximum value of an array over the given dimensions.

maximum!(*r*, *A*)
Compute the maximum value of `A` over the singleton dimensions of `r`, and write results to `r`.

minimum(*itr*)
Returns the smallest element in a collection.

minimum(*A*, *dims*)
Compute the minimum value of an array over the given dimensions.

minimum!(*r*, *A*)
Compute the minimum value of `A` over the singleton dimensions of `r`, and write results to `r`.

extrema(*itr*)
Compute both the minimum and maximum element in a single pass, and return them as a 2-tuple.

indmax(*itr*) → Integer
Returns the index of the maximum element in a collection.

indmin(*itr*) → Integer
Returns the index of the minimum element in a collection.

findmax(*itr*) -> (*x*, *index*)
Returns the maximum element and its index.

findmax(*A*, *dims*) -> (*maxval*, *index*)
For an array input, returns the value and index of the maximum over the given dimensions.

findmin(*itr*) -> (*x*, *index*)
Returns the minimum element and its index.

findmin(*A*, *dims*) -> (*minval*, *index*)
For an array input, returns the value and index of the minimum over the given dimensions.

maxabs(*itr*)
Compute the maximum absolute value of a collection of values.

maxabs(*A*, *dims*)
Compute the maximum absolute values over given dimensions.

maxabs!(*r*, *A*)
Compute the maximum absolute values over the singleton dimensions of `r`, and write values to `r`.

minabs(*itr*)
Compute the minimum absolute value of a collection of values.

minabs(*A*, *dims*)
Compute the minimum absolute values over given dimensions.

minabs!(*r*, *A*)
Compute the minimum absolute values over the singleton dimensions of `r`, and write values to `r`.

sum(*itr*)
Returns the sum of all elements in a collection.

sum(*A*, *dims*)
Sum elements of an array over the given dimensions.

sum!(*r*, *A*)
Sum elements of `A` over the singleton dimensions of `r`, and write results to `r`.

sum(*f*, *itr*)
Sum the results of calling function `f` on each element of `itr`.

sumabs(*itr*)
Sum absolute values of all elements in a collection. This is equivalent to *sum(abs(itr))* but faster.

sumabs(*A*, *dims*)
Sum absolute values of elements of an array over the given dimensions.

sumabs!(*r*, *A*)
Sum absolute values of elements of `A` over the singleton dimensions of `r`, and write results to `r`.

sumabs2(*itr*)
Sum squared absolute values of all elements in a collection. This is equivalent to *sum(abs2(itr))* but faster.

sumabs2(*A*, *dims*)
Sum squared absolute values of elements of an array over the given dimensions.

sumabs2!(*r*, *A*)
Sum squared absolute values of elements of `A` over the singleton dimensions of `r`, and write results to `r`.

prod(*itr*)
Returns the product of all elements of a collection.

prod(*A*, *dims*)
Multiply elements of an array over the given dimensions.

prod!(*r*, *A*)
Multiply elements of `A` over the singleton dimensions of `r`, and write results to `r`.

any(*itr*) → Bool
Test whether any elements of a boolean collection are true.

any(*A*, *dims*)
Test whether any values along the given dimensions of an array are true.

any!(*r*, *A*)
Test whether any values in `A` along the singleton dimensions of `r` are true, and write results to `r`.

all(*itr*) → Bool
Test whether all elements of a boolean collection are true.

all(*A*, *dims*)
Test whether all values along the given dimensions of an array are true.

all!(*r*, *A*)
Test whether all values in `A` along the singleton dimensions of `r` are true, and write results to `r`.

count(*p*, *itr*) → Integer
Count the number of elements in `itr` for which predicate `p` returns true.

any(*p*, *itr*) → Bool
Determine whether predicate `p` returns true for any elements of `itr`.

all(*p*, *itr*) → Bool
Determine whether predicate `p` returns true for all elements of `itr`.

```
julia> all(i->(4<=i<=6), [4,5,6])
true
```

map(*f*, *c...*) → collection
Transform collection `c` by applying `f` to each element. For multiple collection arguments, apply `f` elementwise.

```
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
```

map! (*function*, *collection*)
In-place version of `map()`.

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.

mapreduce (*f*, *op*, *itr*)
Applies function `f` to each element in `itr` and then reduces the result using the binary function `op`.

```
julia> mapreduce(x->x^2, +, [1:3]) # == 1 + 4 + 9
14
```

The associativity of the reduction is implementation-dependent; if you need a particular associativity, e.g. left-to-right, you should write your own loop. See documentation for `reduce`.

first (*coll*)
Get the first element of an iterable collection.

last (*coll*)
Get the last element of an ordered collection, if it can be computed in O(1) time. This is accomplished by calling `endof` to get the last index.

step (*r*)
Get the step size of a `Range` object.

collect (*collection*)
Return an array of all items in a collection. For associative collections, returns (key, value) tuples.

collect (*element_type*, *collection*)
Return an array of type `Array{element_type,1}` of all items in a collection.

issubset (*a*, *b*)
Determine whether every element of `a` is also in `b`, using the `in` function.

filter (*function*, *collection*)
Return a copy of `collection`, removing elements for which `function` is false. For associative collections, the function is passed two arguments (key and value).

filter! (*function*, *collection*)
Update `collection`, removing elements for which `function` is false. For associative collections, the function is passed two arguments (key and value).

Indexable Collections

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, ...)`.

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, ...)`.

Fully implemented by: `Array`, `DArray`, `BitArray`, `AbstractArray`, `SubArray`, `ObjectIdDict`, `Dict`, `WeakKeyDict`, `String`.

Partially implemented by: `Range`, `Range1`, `Tuple`.

Associative Collections

`Dict` is the standard associative collection. Its implementation uses the `hash(x)` as the hashing function for the key, and `isequal(x,y)` to determine equality. Define these two functions for custom types to override how they are stored in a hash table.

`ObjectIdDict` is a special hash table where the keys are always object identities. `WeakKeyDict` is a hash table implementation where the keys are weak references to objects, and thus may be garbage collected even when referenced in a hash table.

Dicts can be created using a literal syntax: `{"A"=>1, "B"=>2}`. Use of curly brackets will create a `Dict` of type `Dict{Any,Any}`. Use of square brackets will attempt to infer type information from the keys and values (i.e. `["A"=>1, "B"=>2]` creates a `Dict{ASCIIString, Int64}`). To explicitly specify types use the syntax: `(KeyType=>ValueType)[...]`. For example, `(ASCIIString=>Int32)["A"=>1, "B"=>2]`.

As with arrays, `Dicts` may be created with comprehensions. For example, `{i => f(i) for i = 1:10}`.

Given a dictionary `D`, the syntax `D[x]` returns the value of key `x` (if it exists) or throws an error, and `D[x] = y` stores the key-value pair `x => y` in `D` (replacing any existing value for the key `x`). Multiple arguments to `D[...]` are converted to tuples; for example, the syntax `D[x,y]` is equivalent to `D[(x,y)]`, i.e. it refers to the value keyed by the tuple `(x,y)`.

`Dict``()`
`Dict{K,V}()` constructs a hash

table with keys of type K and values of type V. The literal syntax is `{"A"=>1, "B"=>2}` for a `Dict{Any, Any}`, or `["A"=>1, "B"=>2]` for a `Dict` of inferred type.

`haskey`(*collection*, *key*) → Bool
Determine whether a collection has a mapping for a given key.

`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.

`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
```

`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`.

`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
```

getkey(*collection*, *key*, *default*)
Return the key matching argument `key` if one exists in `collection`, otherwise return `default`.

delete!(*collection*, *key*)
Delete the mapping for the given key in a collection, and return the colection.

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.

keys(*collection*)
Return an iterator over all keys in a collection. `collect(keys(d))` returns an array of keys.

values(*collection*)
Return an iterator over all values in a collection. `collect(values(d))` returns an array of values.

merge(*collection*, *others...*)
Construct a merged collection from the given collections.

merge!(*collection*, *others...*)
Update collection with pairs from the other collections

sizehint(*s*, *n*)
Suggest that collection `s` reserve capacity for at least `n` elements. This can improve performance.

Fully implemented by: `ObjectIdDict`, `Dict`, `WeakKeyDict`.

Partially implemented by: `IntSet`, `Set`, `EnvHash`, `Array`, `BitArray`.

Set-Like Collections

Set([*itr*])
Construct a `Set` of the values generated by the given iterable object, or an empty set. Should be used instead of `IntSet` for sparse integer sets, or for sets of arbitrary objects.

IntSet([*itr*])
Construct a sorted set of the integers generated by the given iterable object, or an empty set. Implemented as a bit string, and therefore designed for dense integer sets. Only non-negative integers can be stored. If the set will be sparse (for example holding a single very large integer), use `Set` instead.

union(*s1*, *s2...*)
Construct the union of two or more sets. Maintains order with arrays.

union!(*s*, *iterable*)
Union each element of `iterable` into set `s` in-place.

intersect(*s1*, *s2...*)
Construct the intersection of two or more sets. Maintains order and multiplicity of the first argument for arrays and ranges.

setdiff(*s1*, *s2*)
Construct the set of elements in `s1` but not `s2`. Maintains order with arrays. Note that both arguments must be collections, and both will be iterated over. In particular, `setdiff(set,element)` where `element` is a potential member of `set`, will not work in general.

setdiff!(*s*, *iterable*)
Remove each element of `iterable` from set `s` in-place.

symdiff(*s1*, *s2...*)
Construct the symmetric difference of elements in the passed in sets or arrays. Maintains order with arrays.

symdiff!(*s*, *n*)
IntSet s is destructively modified to toggle the inclusion of integer `n`.

symdiff!(*s*, *itr*)
For each element in `itr`, destructively toggle its inclusion in set `s`.

symdiff!(*s1*, *s2*)
Construct the symmetric difference of IntSets `s1` and `s2`, storing the result in `s1`.

complement(*s*)
Returns the set-complement of IntSet `s`.

complement!(*s*)
Mutates IntSet `s` into its set-complement.

intersect!(*s1*, *s2*)
Intersects IntSets `s1` and `s2` and overwrites the set `s1` with the result. If needed, s1 will be expanded to the size of `s2`.

issubset(*A*, *S*) → Bool
True if `A` ⊆ `S` (A is a subset of or equal to S)

Fully implemented by: `IntSet`, `Set`.

Partially implemented by: `Array`.

Dequeues

push!(*collection*, *items...*) → collection
Insert items at the end of a collection.

pop!(*collection*) → item
Remove the last item in a collection and return it.

unshift!(*collection*, *items...*) → collection
Insert items at the beginning of a collection.

shift!(*collection*) → item
Remove the first item in a collection.

insert!(*collection*, *index*, *item*)
Insert an item at the given index.

deleteat!(*collection*, *index*)
Remove the item at the given index, and return the modified collection. Subsequent items are shifted to fill the resulting gap.

deleteat!(*collection*, *itr*)
Remove the items at the indices given by `itr`, and return the modified collection. Subsequent items are shifted to fill the resulting gap. `itr` must be sorted and unique.

splice!(*collection*, *index*[, *replacement*]) → item
Remove the item at the given index, and return the removed item. Subsequent items are shifted down to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed item.

To insert `replacement` before an index `n` without removing any items, use `splice(collection, n-1:n, replacement)`.

`splice!`(*collection*, *range*[, *replacement*]) → items
Remove items in the specified index range, and return a collection containing the removed items. Subsequent items are shifted down 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-1:n, replacement)`.

`resize!`(*collection*, *n*) → collection
Resize collection to contain `n` elements.

`append!`(*collection*, *items*) → collection.
Add the elements of `items` to the end of a collection.

```
julia> append!([1],[2,3])
3-element Array{Int64,1}:
 1
 2
 3
```

`prepend!`(*collection*, *items*) → collection
Insert the elements of `items` to the beginning of a collection.

```
julia> prepend!([3],[1,2])
3-element Array{Int64,1}:
 1
 2
 3
```

Fully implemented by: `Vector` (aka 1-d `Array`), `BitVector` (aka 1-d `BitArray`).

Strings

`length`(*s*)
The number of characters in string `s`.

`sizeof`(*s::String*)
The number of bytes in string `s`.

`*`(*s*, *t*)
Concatenate strings. The `*` operator is an alias to this function.

```
julia> "Hello " * "world"
"Hello world"
```

`^`(*s*, *n*)
Repeat `n` times the string `s`. The `^` operator is an alias to this function.

```
julia> "Test "^3
"Test Test Test "
```

`string`(*xs...*)
Create a string from any values using the `print` function.

repr(*x*)
Create a string from any value using the `showall` function.

bytestring(*::Ptr{Uint8}*[, *length*])
Create a string from the address of a C (0-terminated) string encoded in ASCII or UTF-8. A copy is made; the ptr can be safely freed. If `length` is specified, the string does not have to be 0-terminated.

bytestring(*s*)
Convert a string to a contiguous byte array representation appropriate for passing it to C functions. The string will be encoded as either ASCII or UTF-8.

ascii(*::Array{Uint8, 1}*)
Create an ASCII string from a byte array.

ascii(*s*)
Convert a string to a contiguous ASCII string (all characters must be valid ASCII characters).

utf8(*::Array{Uint8, 1}*)
Create a UTF-8 string from a byte array.

utf8(*s*)
Convert a string to a contiguous UTF-8 string (all characters must be valid UTF-8 characters).

normalize_string(*s*, *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 *normalize_string(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`.

is_valid_ascii(*s*) → Bool
Returns true if the string or byte vector is valid ASCII, false otherwise.

is_valid_utf8(*s*) → Bool
Returns true if the string or byte vector is valid UTF-8, false otherwise.

is_valid_char(*c*) → Bool
Returns true if the given char or integer is a valid Unicode code point.

is_assigned_char(*c*) → Bool
Returns true if the given char or integer is an assigned Unicode code point.

ismatch(*r::Regex*, *s::String*) → Bool
Test whether a string contains a match of the given regular expression.

match(*r::Regex*, *s::String*[, *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.

eachmatch(*r::Regex*, *s::String*[, *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.

matchall(*r::Regex*, *s::String*[, *overlap::Bool=false*]) → Vector{String}
Return a vector of the matching substrings from eachmatch.

lpad(*string*, *n*, *p*)
Make a string at least `n` characters long by padding on the left with copies of `p`.

rpad(*string*, *n*, *p*)
Make a string at least `n` characters long by padding on the right with copies of `p`.

search(*string*, *chars*[, *start*])
Search for the first occurance of the given characters within the given string. The second argument may be a single character, a vector or a set of characters, a string, or a regular expression (though regular expressions are only allowed on contiguous strings, such as ASCII or UTF-8 strings). The third argument optionally specifies a starting index. The return value is a range of indexes where the matching sequence is found, such that `s[search(s,x)] == x`:

`search(string, "substring")` = `start:end` such that `string[start:end] == "substring"`, or `0:-1` if unmatched.

`search(string, 'c')` = `index` such that `string[index] == 'c'`, or `0` if unmatched.

rsearch(*string*, *chars*[, *start*])
Similar to `search`, but returning the last occurance of the given characters within the given string, searching in reverse from `start`.

searchindex(*string*, *substring*[, *start*])
Similar to `search`, but return only the start index at which the substring is found, or 0 if it is not.

rsearchindex(*string*, *substring*[, *start*])
Similar to `rsearch`, but return only the start index at which the substring is found, or 0 if it is not.

contains(*haystack*, *needle*)
Determine whether the second argument is a substring of the first.

replace(*string*, *pat*, *r*[, *n*])
Search for the given pattern `pat`, and replace each occurrence with `r`. If `n` is provided, replace at most `n` occurrences. As with search, the second argument 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.

split(*string, [chars, [limit,] [include_empty]]*)
Return an array of substrings by splitting the given string on occurrences of the given character delimiters, which may be specified in any of the formats allowed by `search`'s second argument (i.e. a single character, collection of characters, string, or regular expression). If `chars` is omitted, it defaults to the set of all space characters, and `include_empty` is taken to be false. The last two arguments are also optional: they are are a maximum size for the result and a flag determining whether empty fields should be included in the result.

rsplit(*string, [chars, [limit,] [include_empty]]*)
Similar to `split`, but starting from the end of the string.

strip(*string*[, *chars*])
Return `string` with any leading and trailing whitespace removed. If `chars` (a character, or vector or set of characters) is provided, instead remove characters contained in it.

lstrip(*string*[, *chars*])
Return `string` with any leading whitespace removed. If `chars` (a character, or vector or set of characters) is provided, instead remove characters contained in it.

rstrip(*string*[, *chars*])
Return `string` with any trailing whitespace removed. If `chars` (a character, or vector or set of characters) is provided, instead remove characters contained in it.

beginswith(*string*, *prefix* | *chars*)
Returns `true` if `string` starts with `prefix`. If the second argument is a vector or set of characters, tests whether the first character of `string` belongs to that set.

endswith(*string*, *suffix* | *chars*)
Returns `true` if `string` ends with `suffix`. If the second argument is a vector or set of characters, tests whether the last character of `string` belongs to that set.

uppercase(*string*)
Returns `string` with all characters converted to uppercase.

lowercase(*string*)
Returns `string` with all characters converted to lowercase.

ucfirst(*string*)
Returns `string` with the first character converted to uppercase.

lcfirst(*string*)
Returns `string` with the first character converted to lowercase.

join(*strings*, *delim*)
Join an array of strings into a single string, inserting the given delimiter between adjacent strings.

chop(*string*)
Remove the last character from a string

chomp(*string*)
Remove a trailing newline from a string

ind2chr(*string*, *i*)
Convert a byte index to a character index

chr2ind(*string*, *i*)
Convert a character index to a byte index

isvalid(*str*, *i*)
Tells whether index `i` is valid for the given string

nextind(*str*, *i*)
Get the next valid string index after `i`. Returns a value greater than `endof(str)` at or after the end of the string.

prevind(*str*, *i*)
Get the previous valid string index before `i`. Returns a value less than `1` at the beginning of the string.

randstring(*len*)
Create a random ASCII string of length `len`, consisting of upper- and lower-case letters and the digits 0-9

charwidth(*c*)
Gives the number of columns needed to print a character.

strwidth(*s*)
Gives the number of columns needed to print a string.

isalnum(*c::Union(Char, String)*) → Bool
Tests whether a character is alphanumeric, or whether this is true for all elements of a string.

isalpha(*c::Union(Char, String)*) → Bool
Tests whether a character is alphabetic, or whether this is true for all elements of a string.

isascii(*c::Union(Char, String)*) → Bool
Tests whether a character belongs to the ASCII character set, or whether this is true for all elements of a string.

isblank(*c::Union(Char, String)*) → Bool
Tests whether a character is a tab or space, or whether this is true for all elements of a string.

iscntrl(*c::Union(Char, String)*) → Bool
Tests whether a character is a control character, or whether this is true for all elements of a string.

isdigit(*c::Union(Char, String)*) → Bool
Tests whether a character is a numeric digit (0-9), or whether this is true for all elements of a string.

isgraph(*c::Union(Char, String)*) → Bool
Tests whether a character is printable, and not a space, or whether this is true for all elements of a string.

islower(*c::Union(Char, String)*) → Bool
Tests whether a character is a lowercase letter, or whether this is true for all elements of a string.

isprint(*c::Union(Char, String)*) → Bool
Tests whether a character is printable, including space, or whether this is true for all elements of a string.

ispunct(*c::Union(Char, String)*) → Bool
Tests whether a character is printable, and not a space or alphanumeric, or whether this is true for all elements of a string.

isspace(*c::Union(Char, String)*) → Bool
Tests whether a character is any whitespace character, or whether this is true for all elements of a string.

isupper(*c::Union(Char, String)*) → Bool
Tests whether a character is an uppercase letter, or whether this is true for all elements of a string.

isxdigit(*c::Union(Char, String)*) → Bool
Tests whether a character is a valid hexadecimal digit, or whether this is true for all elements of a string.

symbol(*str*) → Symbol
Convert a string to a `Symbol`.

escape_string(*str::String*) → String
General escaping of traditional C and Unicode escape sequences. See `print_escaped()` for more general escaping.

unescape_string(*s::String*) → String
General unescaping of traditional C and Unicode escape sequences. Reverse of `escape_string()`. See also `print_unescaped()`.

utf16(*s*)
Create a UTF-16 string from a byte array, array of `Uint16`, or any other string type. (Data must be valid UTF-16. Conversions of byte arrays check for a byte-order marker in the first two bytes, and do not include it in the resulting string.)

Note that the resulting `UTF16String` data is terminated by the NUL codepoint (16-bit zero), which is not treated as a character in the string (so that it is mostly invisible in Julia); this allows the string to be passed directly to external functions requiring NUL-terminated data. This NUL is appended automatically by the *utf16(s)* conversion function. If you have a `Uint16` array `A` that is already NUL-terminated valid UTF-16 data, then you can instead use *UTF16String(A)*‘ to construct the string without making a copy of the data and treating the NUL as a terminator rather than as part of the string.

utf16(*::Union(Ptr{Uint16}, Ptr{Int16})*[, *length*])
Create a string from the address of a NUL-terminated UTF-16 string. A copy is made; the pointer can be safely freed. If `length` is specified, the string does not have to be NUL-terminated.

is_valid_utf16(*s*) → Bool
Returns true if the string or `Uint16` array is valid UTF-16.

utf32(*s*)
Create a UTF-32 string from a byte array, array of `Uint32`, or any other string type. (Conversions of byte arrays check for a byte-order marker in the first four bytes, and do not include it in the resulting string.)

Note that the resulting `UTF32String` data is terminated by the NUL codepoint (32-bit zero), which is not treated as a character in the string (so that it is mostly invisible in Julia); this allows the string to be passed directly to external functions requiring NUL-terminated data. This NUL is appended automatically by the *utf32(s)* conversion function. If you have a `Uint32` array `A` that is already NUL-terminated UTF-32 data, then you can instead use *UTF32String(A)*‘ to construct the string without making a copy of the data and treating the NUL as a terminator rather than as part of the string.

utf32(*::Union(Ptr{Char}, Ptr{Uint32}, Ptr{Int32})*[, *length*])
Create a string from the address of a NUL-terminated UTF-32 string. A copy is made; the pointer can be safely freed. If `length` is specified, the string does not have to be NUL-terminated.

wstring(*s*)
This is a synonym for either `utf32(s)` or `utf16(s)`, depending on whether `Cwchar_t` is 32 or 16 bits, respectively. The synonym `WString` for `UTF32String` or `UTF16String` is also provided.

I/O

STDOUT
Global variable referring to the standard out stream.

STDERR
Global variable referring to the standard error stream.

STDIN
Global variable referring to the standard input stream.

open(*file_name*[, *read*, *write*, *create*, *truncate*, *append*]) → IOStream
Open a file in a mode specified by five boolean arguments. The default is to open files for reading only. Returns a stream for accessing the file.

open(*file_name*[, *mode*]) → 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:

r	read
r+	read, write
w	write, create, truncate
w+	read, write, create, truncate
a	write, create, append
a+	read, write, create, append

open(*f::function*, *args...*)

Apply the function `f` to the result of `open(args...)` and close the resulting file descriptor upon completion.

Example: `open(readall, "file.txt")`

IOBuffer() → IOBuffer

Create an in-memory I/O stream.

IOBuffer(*size::Int*)

Create a fixed size IOBuffer. The buffer will not grow dynamically.

IOBuffer(*string*)

Create a read-only IOBuffer on the data underlying the given string

IOBuffer([*data*][, *readable*, *writable*[, *maxsize*]])

Create an IOBuffer, which may optionally operate on a pre-existing array. If the readable/writable arguments are given, they restrict whether or not the buffer may be read from or written to respectively. By default the buffer is readable but not writable. The last argument optionally specifies a size beyond which the buffer may not be grown.

takebuf_array(*b::IOBuffer*)

Obtain the contents of an `IOBuffer` as an array, without copying.

takebuf_string(*b::IOBuffer*)

Obtain the contents of an `IOBuffer` as a string, without copying.

fdio([*name::String*], *fd::Integer*[, *own::Bool*]) → 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.

flush(*stream*)

Commit all currently buffered writes to the given stream.

flush_cstdio()

Flushes the C `stdout` and `stderr` streams (which may have been written to by external C code).

close(*stream*)

Close an I/O stream. Performs a `flush` first.

write(*stream*, *x*)

Write the canonical binary representation of a value to the given stream.

read(*stream*, *type*)

Read a value of the given type from a stream, in canonical binary representation.

read(*stream*, *type*, *dims*)

Read a series of values of the given type from a stream, in canonical binary representation. `dims` is either a tuple or a series of integer arguments specifying the size of `Array` to return.

read!(*stream*, *array::Array*)
Read binary data from a stream, filling in the argument `array`.

readbytes!(*stream*, *b::Vector{Uint8}*, *nb=length(b)*)
Read at most `nb` bytes from the stream into `b`, returning the number of bytes read (increasing the size of `b` as needed).

readbytes(*stream*, *nb=typemax(Int)*)
Read at most `nb` bytes from the stream, returning a `Vector{Uint8}` of the bytes read.

position(*s*)
Get the current position of a stream.

seek(*s*, *pos*)
Seek a stream to the given position.

seekstart(*s*)
Seek a stream to its beginning.

seekend(*s*)
Seek a stream to its end.

skip(*s*, *offset*)
Seek a stream relative to the current position.

mark(*s*)
Add a mark at the current position of stream `s`. Returns the marked position.

See also `unmark()`, `reset()`, `ismarked()`

unmark(*s*)
Remove a mark from stream `s`. Returns `true` if the stream was marked, `false` otherwise.

See also `mark()`, `reset()`, `ismarked()`

reset(*s*)
Reset a stream `s` to a previously marked position, and remove the mark. Returns the previously marked position. Throws an error if the stream is not marked.

See also `mark()`, `unmark()`, `ismarked()`

ismarked(*s*)
Returns true if stream `s` is marked.

See also `mark()`, `unmark()`, `reset()`

eof(*stream*) → Bool
Tests 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.

isreadonly(*stream*) → Bool
Determine whether a stream is read-only.

isopen(*stream*) → Bool
Determine whether a stream is open (i.e. has not been closed yet). If the connection has been closed remotely (in case of e.g. a socket), `isopen` will return `false` even though buffered data may still be available. Use `eof` to check if necessary.

ntoh(*x*)
Converts the endianness of a value from Network byte order (big-endian) to that used by the Host.

hton(*x*)
Converts the endianness of a value from that used by the Host to Network byte order (big-endian).

ltoh(*x*)
Converts the endianness of a value from Little-endian to that used by the Host.

htol(*x*)
Converts the endianness of a value from that used by the Host to Little-endian.

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.

serialize(*stream*, *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.

deserialize(*stream*)
Read a value written by `serialize`.

print_escaped(*io*, *str::String*, *esc::String*)
General escaping of traditional C and Unicode escape sequences, plus any characters in esc are also escaped (with a backslash).

print_unescaped(*io*, *s::String*)
General unescaping of traditional C and Unicode escape sequences. Reverse of `print_escaped()`.

print_joined(*io*, *items*, *delim*[, *last*])
Print elements of `items` to `io` with `delim` between them. If `last` is specified, it is used as the final delimiter instead of `delim`.

print_shortest(*io*, *x*)
Print the shortest possible representation of number `x` as a floating point number, ensuring that it would parse to the exact same number.

fd(*stream*)
Returns 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.

redirect_stdout()
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.

redirect_stdout(*stream*)
Replace STDOUT by stream for all C and julia level output to STDOUT. Note that *stream* must be a TTY, a Pipe or a TcpSocket.

redirect_stderr([*stream*])
Like redirect_stdout, but for STDERR

redirect_stdin([*stream*])
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.

readchomp(*x*)
Read the entirety of x as a string but remove trailing newlines. Equivalent to chomp(readall(x)).

readdir([*dir*]) → Vector{ByteString}
Returns the files and directories in the directory *dir* (or the current working directory if not given).

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

skipchars(*stream*, *predicate; linecomment::Char*)
Advance the stream until before the first character for which `predicate` returns false. For example `skipchars(stream, isspace)` will skip all whitespace. If keyword argument `linecomment` is specified, characters from that character through the end of a line will also be skipped.

countlines(*io*[, *eol::Char*])
Read io until the end of the stream/file and count the number of non-empty 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.

PipeBuffer()
An IOBuffer that allows reading and performs writes by appending. Seeking and truncating are not supported. See IOBuffer for the available constructors.

PipeBuffer(*data::Vector{Uint8}*[, *maxsize*])
Create a PipeBuffer to operate on a data vector, optionally specifying a size beyond which the underlying Array may not be grown.

readavailable(*stream*)
Read all available data on the stream, blocking the task only if no data is available.

stat(*file*)
Returns a structure whose fields contain information about the file. The fields of the structure are:

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 preffered 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

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.

ctime(*file*)
Equivalent to stat(file).ctime

mtime(*file*)
Equivalent to stat(file).mtime

filemode(*file*)
Equivalent to stat(file).mode

filesize(*path...*)
Equivalent to stat(file).size

uperm(*file*)
Gets the permissions of the owner of the file as a bitfield of

01	Execute Permission
02	Write Permission
04	Read Permission

For allowed arguments, see `stat`.

gperm (*file*)
Like uperm but gets the permissions of the group owning the file

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

cp (*src::String*, *dst::String*)
Copy a file from *src* to *dest*.

download (*url*[, *localfile*])
Download a file from the given url, optionally renaming it to the given local file name. 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 need, please use a package that provides the desired functionality instead.

mv (*src::String*, *dst::String*)
Move a file from *src* to *dst*.

rm (*path::String; recursive=false*)
Delete the file, link, or empty directory at the given path. If `recursive=true` is passed and the path is a directory, then all contents are removed recursively.

touch (*path::String*)
Update the last-modified timestamp on a file to the current time.

Network I/O

connect ([*host*], *port*) → TcpSocket
Connect to the host `host` on port `port`

connect (*path*) → Pipe
Connect to the Named Pipe/Domain Socket at `path`

listen ([*addr*], *port*) → 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.

listen (*path*) → PipeServer
Listens on/Creates a Named Pipe/Domain Socket

getaddrinfo (*host*)
Gets the IP address of the `host` (may have to do a DNS lookup)

parseip (*addr*)
Parse a string specifying an IPv4 or IPv6 ip address.

IPv4 (*host::Integer*) → IPv4
Returns IPv4 object from ip address formatted as Integer

IPv6 (*host::Integer*) → IPv6
Returns IPv6 object from ip address formatted as Integer

nb_available(*stream*)
Returns the number of bytes available for reading before a read from this stream or buffer will block.

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.

listenany(*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.

watch_file(*cb=false*, *s; poll=false*)
Watch file or directory `s` and run callback `cb` when `s` is modified. The `poll` parameter specifies whether to use file system event monitoring or polling. The callback function `cb` should accept 3 arguments: `(filename, events, status)` where `filename` is the name of file that was modified, `events` is an object with boolean fields `changed` and `renamed` when using file system event monitoring, or `readable` and `writable` when using polling, and `status` is always 0. Pass `false` for `cb` to not use a callback function.

poll_fd(*fd*, *seconds::Real; readable=false*, *writable=false*)
Poll a file descriptor fd for changes in the read or write availability and with a timeout given by the second argument. If the timeout is not needed, use `wait(fd)` instead. The keyword arguments determine which of read and/or write status should be monitored and at least one of them needs to be set to true. The returned value is an object with boolean fields `readable`, `writable`, and `timedout`, giving the result of the polling.

poll_file(*s*, *interval_seconds::Real*, *seconds::Real*)
Monitor a file for changes by polling every *interval_seconds* seconds for *seconds* seconds. A return value of true indicates the file changed, a return value of false indicates a timeout.

Text I/O

show(*x*)
Write an informative text representation of a value to the current output stream. New types should overload `show(io, x)` where the first argument is a stream. The representation used by `show` generally includes Julia-specific formatting and type information.

showcompact(*x*)
Show a more compact representation of a value. This is used for printing array elements. If a new type has a different compact representation, it should overload `showcompact(io, x)` where the first argument is a stream.

showall(*x*)
Similar to `show`, except shows all elements of arrays.

summary(*x*)
Return a string giving a brief description of a value. By default returns `string(typeof(x))`. For arrays, returns strings like "2x2 Float64 Array".

print(*x*)
Write (to the default output stream) a canonical (un-decorated) text representation of a value if there is one, otherwise call `show`. The representation used by `print` includes minimal formatting and tries to avoid Julia-specific details.

println(*x*)
Print (using `print()`) `x` followed by a newline.

print_with_color(*color::Symbol*[, *io*], *strings...*)
Print strings in a color specified as a symbol, for example `:red` or `:blue`.

info(*msg*)
Display an informational message.

warn(*msg*)
Display a warning.

@printf([*io::IOStream*], *"%Fmt"*, *args...*)
Print arg(s) using C `printf()` style format specification string. Optionally, an IOStream may be passed as the first argument to redirect output.

@sprintf(*"%Fmt"*, *args...*)
Return `@printf` formatted output as string.

sprint(*f::Function*, *args...*)
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.

showerror(*io*, *e*)
Show a descriptive representation of an exception object.

dump(*x*)
Show all user-visible structure of a value.

xdump(*x*)
Show all structure of a value, including all fields of objects.

readall(*stream*)
Read the entire contents of an I/O stream as a string.

readline(*stream=STDIN*)
Read a single line of text, including a trailing newline character (if one is reached before the end of the input), from the given `stream` (defaults to `STDIN`),

readuntil(*stream*, *delim*)
Read a string, up to and including the given delimiter byte.

readlines(*stream*)
Read all lines as an array.

eachline(*stream*)
Create an iterable object that will yield each line from a stream.

readdlm(*source*, *delim::Char*, *T::Type*, *eol::Char; header=false*, *skipstart=0*, *use_mmap*, *ignore_invalid_chars=false*, *quotes=true*, *dims*, *comments=true*, *comment_char='#'*)
Read a matrix from the source where each line (separated by `eol`) gives one row, with elements separated by the given delimeter. 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 `ASCIIString`, `String`, 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 `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 `ignore_invalid_chars` is `true`, bytes in `source` with invalid character encoding will be ignored. Otherwise an error is thrown indicating the offending character position.

If `quotes` is `true`, column 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.

readdlm(*source*, *delim::Char*, *eol::Char; options...*)
If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a cell array of numbers and strings is returned.

readdlm(*source*, *delim::Char*, *T::Type; options...*)
The end of line delimiter is taken as `\n`.

readdlm(*source*, *delim::Char; 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 cell array of numbers and strings is returned.

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`.

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 cell array of numbers and strings is returned.

writedlm(*f*, *A*, *delim='t'*)
Write `A` (either an array type or an iterable collection of iterable rows) as text to `f` (either a filename string or an `IO` stream) using the given delimeter `delim` (which defaults to tab, but can be any printable Julia object, typically a `Char` or `String`).

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))`.

readcsv(*source, [T::Type]; options...*)
Equivalent to `readdlm` with `delim` set to comma.

writecsv(*filename*, *A*)
Equivalent to `writedlm` with `delim` set to comma.

Base64Pipe(*ostream*)
Returns 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 `Base64Pipe` stream is necessary to complete the encoding (but does not close `ostream`).

base64(*writefunc*, *args...*)
base64(*args...*)
Given a `write`-like function `writefunc`, which takes an I/O stream as its first argument, `base64(writefunc, args...)` calls `writefunc` to write `args...` to a base64-encoded string, and returns the string. `base64(args...)` is equivalent to `base64(write, args...)`: it converts its arguments into bytes using the standard `write` functions and returns the base64-encoded string.

Multimedia I/O

Just as text output is performed by `print` and user-defined types can indicate their textual representation by overloading `show`, Julia provides a standardized mechanism for rich multimedia output (such as images, formatted text, or even audio and video), consisting of three parts:

- A function `display(x)` to request the richest available multimedia display of a Julia object `x` (with a plain-text fallback).
- Overloading `writemime` allows one to indicate arbitrary multimedia representations (keyed by standard MIME types) of user-defined types.
- Multimedia-capable display backends may be registered by subclassing a generic `Display` type and pushing them onto a stack of display backends via `pushdisplay`.

The base Julia runtime provides only plain-text display, but richer displays may be enabled by loading external modules or by using graphical Julia environments (such as the IPython-based IJulia notebook).

display(*x*)
display(*d::Display*, *x*)
display(*mime*, *x*)
display(*d::Display*, *mime*, *x*)

Display `x` using the topmost applicable display in the display stack, typically using the richest supported multimedia output for `x`, with plain-text `STDOUT` output as a fallback. The `display(d, x)` variant attempts to display `x` on the given display `d` only, throwing a `MethodError` if `d` cannot display objects of this type.

There are also two variants with a `mime` argument (a MIME type string, such as `"image/png"`), which attempt to display `x` using the requesed 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::String` (for MIME types with text-based storage, such as text/html or application/postscript) or `x::Vector{Uint8}` (for binary MIME types).

redisplay(*x*)
redisplay(*d::Display*, *x*)
redisplay(*mime*, *x*)
redisplay(*d::Display*, *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.

displayable(*mime*) → Bool
displayable(*d::Display*, *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.

writemime(*stream*, *mime*, *x*)

The `display` functions ultimately call `writemime` in order to write an object `x` as a given `mime` type to a given I/O `stream` (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 `writemime` method for `T`, via: `writemime(stream, ::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 `stream`. (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 `writemime(stream, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable `Display` (such as IJulia). As usual, be sure to `import Base.writemime` in order to add new methods to the built-in Julia function `writemime`.

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.

mimewritable(*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 `writemime` function for `typeof(x)`.)

reprmime(*mime*, *x*)
Returns a `String` or `Vector{Uint8}` containing the representation of `x` in the requested `mime` type, as written by `writemime` (throwing a `MethodError` if no appropriate `writemime` is available). A `String` 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 `istext(mime)` returns whether or not Julia treats a given `mime` type as text.)

As a special case, if `x` is a `String` (for textual MIME types) or a `Vector{Uint8}` (for binary MIME types), the `reprmime` function assumes that `x` is already in the requested `mime` format and simply returns `x`.

stringmime(*mime*, *x*)
Returns a `String` containing the representation of `x` in the requested `mime` type. This is similar to `reprmime` except that binary data is base64-encoded as an ASCII string.

As mentioned above, one can also define new display backends. For example, a module that can display PNG images in a window can register this capability with Julia, so that calling `display(x)` on types with PNG representations will automatically display the image using the module's window.

In order to define a new display backend, one should first create a subtype `D` of the abstract class `Display`. 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 `reprmime(mime, x)`. A `MethodError` should be thrown if `x` cannot be displayed as that MIME type; this is automatic if one calls `reprmime`. Finally, one should define a function `display(d::D, x)` that queries `mimewritable(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:

pushdisplay(*d::Display*)
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`).

popdisplay()
popdisplay(*d::Display*)
Pop the topmost backend off of the display-backend stack, or the topmost copy of `d` in the second variant.

TextDisplay(*stream*)
Returns a `TextDisplay <: Display`, which can display any object as the text/plain MIME type (only), writing the text representation to the given I/O stream. (The text representation is the same as the way an object is printed in the Julia REPL.)

istext(*m::MIME*)
Determine whether a MIME type is text data.

Memory-mapped I/O

mmap_array(*type*, *dims*, *stream*[, *offset*])
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 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 specifying the size of the array.

The file is passed via the stream argument. 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.

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 example, the following code:

```
# Create a file for mmapping
# (you could alternatively use mmap_array 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_array(Int, (m,n), s)
```

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_bitarray([*type*], *dims*, *stream*[, *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_array()`, but the byte representation is different. The `type` parameter is optional, and must be `Bool` if given.

Example: `B = mmap_bitarray((25,30000), s)`

This would create a 25-by-30000 `BitArray`, linked to the file associated with stream `s`.

msync(*array*)

Forces synchronization between the in-memory version of a memory-mapped `Array` or `BitArray` and the on-disk version.

msync(*ptr*, *len*[, *flags*])

Forces synchronization of the `mmap()`ped memory region from `ptr` to `ptr+len`. Flags defaults to `MS_SYNC`, but can be a combination of `MS_ASYNC`, `MS_SYNC`, or `MS_INVALIDATE`. See your platform man page for specifics. The flags argument is not valid on Windows.

You may not need to call `msync`, because synchronization is performed at intervals automatically by the operating system. However, you can call this directly if, for example, you are concerned about losing the result of a long-running calculation.

MS_ASYNC

Enum constant for `msync()`. See your platform man page for details. (not available on Windows).

MS_SYNC
Enum constant for `msync()`. See your platform man page for details. (not available on Windows).

MS_INVALIDATE
Enum constant for `msync()`. See your platform man page for details. (not available on Windows).

mmap (*len*, *prot*, *flags*, *fd*, *offset*)
Low-level interface to the `mmap` system call. See the man page.

munmap (*pointer*, *len*)
Low-level interface for unmapping memory (see the man page). With `mmap_array()` you do not need to call this directly; the memory is unmapped for you when the array goes out of scope.

Standard Numeric Types

`Bool` `Int8` `Uint8` `Int16` `Uint16` `Int32` `Uint32` `Int64` `Uint64` `Int128` `Uint128` `Float16` `Float32` `Float64` `Complex64` `Complex128`

Mathematical Operators

- (*x*)
Unary minus operator.

+ (*x*, *y...*)
Addition operator. `x+y+z+...` calls this function with all arguments, i.e. `+(x, y, z, ...)`.

- (*x*, *y*)
Subtraction operator.

***** (*x*, *y...*)
Multiplication operator. `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`.

/ (*x*, *y*)
Right division operator: multiplication of `x` by the inverse of `y` on the right. Gives floating-point results for integer arguments.

**** (*x*, *y*)
Left division operator: multiplication of `y` by the inverse of `x` on the left. Gives floating-point results for integer arguments.

^ (*x*, *y*)
Exponentiation operator.

.+ (*x*, *y*)
Element-wise addition operator.

.- (*x*, *y*)
Element-wise subtraction operator.

.* (*x*, *y*)
Element-wise multiplication operator.

./ (*x*, *y*)
Element-wise right division operator.

. (*x*, *y*)
Element-wise left division operator.

.^ (*x*, *y*)
Element-wise exponentiation operator.

div (*a*, *b*)
Compute a/b, truncating to an integer.

fld (*a*, *b*)
Largest integer less than or equal to a/b.

mod (*x*, *m*)
Modulus after division, returning in the range [0,m).

mod2pi (*x*)
Modulus after division by 2pi, returning in the range [0,2pi).

This function computes a floating point representation of the modulus after division by numerically exact 2pi, and is therefore not exactly the same as mod(x,2pi), which would compute the modulus of x relative to division by the floating-point number 2pi.

rem (*x*, *m*)
Remainder after division.

divrem (*x*, *y*)
Returns `(x/y, x%y)`.

% (*x*, *m*)
Remainder after division. The operator form of `rem`.

mod1 (*x*, *m*)
Modulus after division, returning in the range (0,m]

rem1 (*x*, *m*)
Remainder after division, returning in the range (0,m]

// (*num*, *den*)
Divide two integers or rational numbers, giving a `Rational` result.

rationalize ([*Type=Int*], *x; tol=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`.

num (*x*)
Numerator of the rational representation of `x`

den (*x*)
Denominator of the rational representation of `x`

<< (*x*, *n*)
Left bit shift operator.

>> (*x*, *n*)
Right bit shift operator, preserving the sign of `x`.

>>> (*x*, *n*)
Unsigned right bit shift operator.

: (*start*[, *step*], *stop*)
Range operator. `a:b` constructs a range from `a` to `b` with a step size of 1, and `a:s:b` is similar but uses a step size of `s`. These syntaxes call the function `colon`. The colon is also used in indexing to select whole dimensions.

colon (*start*[, *step*], *stop*)
Called by `:` syntax for constructing ranges.

range(*start*[, *step*], *length*)
Construct a range by length, given a starting value and optional step (defaults to 1).

linrange(*start*, *end*, *length*)
Construct a range by length, given a starting and ending value.

==(*x*, *y*)
Generic equality operator, giving a single `Bool` result. 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.

Follows IEEE semantics for floating-point numbers.

Collections should generally implement == by calling == recursively on all contents.

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.

!=(*x*, *y*)
Not-equals comparison operator. Always gives the opposite answer as ==. New types should generally not implement this, and rely on the fallback definition `!=(x,y) = !(x==y)` instead.

===(*x*, *y*)
See the `is()` operator

!==(*x*, *y*)
Equivalent to `!is(x, y)`

<(*x*, *y*)
Less-than comparison operator. New numeric types should implement this function for two arguments of the new type. Because of the behavior of floating-point NaN values, < implements a partial order. Types with a canonical partial order should implement <, and types with a canonical total order should implement `isless`.

<=(*x*, *y*)
Less-than-or-equals comparison operator.

>(*x*, *y*)
Greater-than comparison operator. Generally, new types should implement < instead of this function, and rely on the fallback definition `>(x,y) = y<x`.

>=(*x*, *y*)
Greater-than-or-equals comparison operator.

.==(*x*, *y*)
Element-wise equality comparison operator.

.!=(*x*, *y*)
Element-wise not-equals comparison operator.

.<(*x*, *y*)
Element-wise less-than comparison operator.

.<=(*x*, *y*)
Element-wise less-than-or-equals comparison operator.

.>(*x*, *y*)
Element-wise greater-than comparison operator.

.>=(*x*, *y*)
Element-wise greater-than-or-equals comparison operator.

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`. For floating-point numbers, uses < but throws an error for unordered arguments.

~ (*x*)
Bitwise not

& (*x*, *y*)
Bitwise and

| (*x*, *y*)
Bitwise or

$ (*x*, *y*)
Bitwise exclusive or

! (*x*)
Boolean not

x && y
Short-circuiting boolean and

x || y
Short-circuiting boolean or

A_ldiv_Bc (*a*, *b*)
Matrix operator A \ B^H

A_ldiv_Bt (*a*, *b*)
Matrix operator A \ B^T

A_mul_B (...)
Matrix operator A B

A_mul_Bc (...)
Matrix operator A B^H

A_mul_Bt (...)
Matrix operator A B^T

A_rdiv_Bc (...)
Matrix operator A / B^H

A_rdiv_Bt (*a*, *b*)
Matrix operator A / B^T

Ac_ldiv_B (...)
Matrix operator A^H \ B

Ac_ldiv_Bc (...)
Matrix operator A^H \ B^H

Ac_mul_B (...)
Matrix operator A^H B

Ac_mul_Bc (...)
Matrix operator A^H B^H

Ac_rdiv_B (*a*, *b*)
Matrix operator A^H / B

Ac_rdiv_Bc (*a*, *b*)
Matrix operator A^H / B^H

At_ldiv_B(...)
Matrix operator $A^T \backslash B$

At_ldiv_Bt(...)
Matrix operator $A^T \backslash B^T$

At_mul_B(...)
Matrix operator $A^T B$

At_mul_Bt(...)
Matrix operator $A^T B^T$

At_rdiv_B(*a*, *b*)
Matrix operator A^T / B

At_rdiv_Bt(*a*, *b*)
Matrix operator A^T / B^T

Mathematical Functions

isapprox(*x::Number*, *y::Number; rtol::Real=cbrt(maxeps)*, *atol::Real=sqrt(maxeps)*)
Inexact equality comparison - behaves slightly different depending on types of input args:

- For `FloatingPoint` numbers, `isapprox` returns `true` if `abs(x-y) <= atol + rtol*max(abs(x), abs(y))`.
- For `Integer` and `Rational` numbers, `isapprox` returns `true` if `abs(x-y) <= atol`. The *rtol* argument is ignored. If one of `x` and `y` is `FloatingPoint`, the other is promoted, and the method above is called instead.
- For `Complex` numbers, the distance in the complex plane is compared, using the same criterion as above.

For default tolerance arguments, `maxeps = max(eps(abs(x)), eps(abs(y)))`.

sin(*x*)
Compute sine of `x`, where `x` is in radians

cos(*x*)
Compute cosine of `x`, where `x` is in radians

tan(*x*)
Compute tangent of `x`, where `x` is in radians

sind(*x*)
Compute sine of `x`, where `x` is in degrees

cosd(*x*)
Compute cosine of `x`, where `x` is in degrees

tand(*x*)
Compute tangent of `x`, where `x` is in degrees

sinpi(*x*)
Compute $\sin(\pi x)$ more accurately than `sin(pi*x)`, especially for large `x`.

cospi(*x*)
Compute $\cos(\pi x)$ more accurately than `cos(pi*x)`, especially for large `x`.

sinh(*x*)
Compute hyperbolic sine of `x`

cosh(*x*)
Compute hyperbolic cosine of x

tanh(*x*)
Compute hyperbolic tangent of x

asin(*x*)
Compute the inverse sine of x, where the output is in radians

acos(*x*)
Compute the inverse cosine of x, where the output is in radians

atan(*x*)
Compute the inverse tangent of x, where the output is in radians

atan2(*y*, *x*)
Compute the inverse tangent of y/x, using the signs of both x and y to determine the quadrant of the return value.

asind(*x*)
Compute the inverse sine of x, where the output is in degrees

acosd(*x*)
Compute the inverse cosine of x, where the output is in degrees

atand(*x*)
Compute the inverse tangent of x, where the output is in degrees

sec(*x*)
Compute the secant of x, where x is in radians

csc(*x*)
Compute the cosecant of x, where x is in radians

cot(*x*)
Compute the cotangent of x, where x is in radians

secd(*x*)
Compute the secant of x, where x is in degrees

cscd(*x*)
Compute the cosecant of x, where x is in degrees

cotd(*x*)
Compute the cotangent of x, where x is in degrees

asec(*x*)
Compute the inverse secant of x, where the output is in radians

acsc(*x*)
Compute the inverse cosecant of x, where the output is in radians

acot(*x*)
Compute the inverse cotangent of x, where the output is in radians

asecd(*x*)
Compute the inverse secant of x, where the output is in degrees

acscd(*x*)
Compute the inverse cosecant of x, where the output is in degrees

acotd(*x*)
Compute the inverse cotangent of x, where the output is in degrees

sech(*x*)
Compute the hyperbolic secant of `x`

csch(*x*)
Compute the hyperbolic cosecant of `x`

coth(*x*)
Compute the hyperbolic cotangent of `x`

asinh(*x*)
Compute the inverse hyperbolic sine of `x`

acosh(*x*)
Compute the inverse hyperbolic cosine of `x`

atanh(*x*)
Compute the inverse hyperbolic tangent of `x`

asech(*x*)
Compute the inverse hyperbolic secant of `x`

acsch(*x*)
Compute the inverse hyperbolic cosecant of `x`

acoth(*x*)
Compute the inverse hyperbolic cotangent of `x`

sinc(*x*)
Compute $\sin(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

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)`.

deg2rad(*x*)
Convert `x` from degrees to radians

rad2deg(*x*)
Convert `x` from radians to degrees

hypot(*x*, *y*)
Compute the $\sqrt{x^2 + y^2}$ avoiding overflow and underflow

log(*x*)
Compute the natural logarithm of `x`. Throws `DomainError` for negative `Real` arguments. Use complex negative arguments instead.

log(*b*, *x*)
Compute the base `b` logarithm of `x`. Throws `DomainError` for negative `Real` arguments.

log2(*x*)
Compute the logarithm of `x` to base 2. Throws `DomainError` for negative `Real` arguments.

log10(*x*)
Compute the logarithm of `x` to base 10. Throws `DomainError` for negative `Real` arguments.

log1p(*x*)
Accurate natural logarithm of `1+x`. Throws `DomainError` for `Real` arguments less than -1.

frexp(*val*)
Return `(x,exp)` such that `x` has a magnitude in the interval `[1/2, 1)` or 0, and val = $x \times 2^{exp}$.

exp(*x*)
Compute e^x

exp2 (*x*)
Compute 2^x

exp10 (*x*)
Compute 10^x

ldexp (*x*, *n*)
Compute $x \times 2^n$

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.

expm1 (*x*)
Accurately compute $e^x - 1$

round (*x*[, *digits*[, *base*]])
`round(x)` returns the nearest integral value of the same type as `x` to `x`. `round(x, digits)` rounds to the specified number of digits after the decimal place, or before if negative, e.g., `round(pi,2)` is `3.14`. `round(x, digits, base)` rounds using a different base, defaulting to 10, e.g., `round(pi, 1, 8)` is `3.125`.

ceil (*x*[, *digits*[, *base*]])
Returns the nearest integral value of the same type as `x` not less than `x`. `digits` and `base` work as above.

floor (*x*[, *digits*[, *base*]])
Returns the nearest integral value of the same type as `x` not greater than `x`. `digits` and `base` work as above.

trunc (*x*[, *digits*[, *base*]])
Returns the nearest integral value of the same type as `x` not greater in magnitude than `x`. `digits` and `base` work as above.

iround (*x*) → Integer
Returns the nearest integer to `x`.

iceil (*x*) → Integer
Returns the nearest integer not less than `x`.

ifloor (*x*) → Integer
Returns the nearest integer not greater than `x`.

itrunc (*x*) → Integer
Returns the nearest integer not greater in magnitude than `x`.

signif (*x*, *digits*[, *base*])
Rounds (in the sense of `round`) `x` so that there are `digits` significant digits, under a base `base` representation, default 10. E.g., `signif(123.456, 2)` is `120.0`, and `signif(357.913, 4, 2)` is `352.0`.

min (*x*, *y*, ...)
Return the minimum of the arguments. Operates elementwise over arrays.

max (*x*, *y*, ...)
Return the maximum of the arguments. Operates elementwise over arrays.

minmax (*x*, *y*)
Return `(min(x,y), max(x,y))`. See also: `extrema()` that returns `(minimum(x), maximum(x))`

clamp (*x*, *lo*, *hi*)
Return x if `lo <= x <= hi`. If `x < lo`, return `lo`. If `x > hi`, return `hi`. Arguments are promoted to a common type. Operates elementwise over `x` if it is an array.

abs (*x*)
Absolute value of `x`

abs2(*x*)
Squared absolute value of `x`

copysign(*x*, *y*)
Return `x` such that it has the same sign as `y`

sign(*x*)
Return `+1` if `x` is positive, `0` if `x == 0`, and `-1` if `x` is negative.

signbit(*x*)
Returns `1` if the value of the sign of `x` is negative, otherwise `0`.

flipsign(*x*, *y*)
Return `x` with its sign flipped if `y` is negative. For example `abs(x) = flipsign(x,x)`.

sqrt(*x*)
Return $\sqrt{x}$. Throws `DomainError` for negative `Real` arguments. Use complex negative arguments instead. The prefix operator $\sqrt{}$ is equivalent to `sqrt`.

isqrt(*n*)
Integer square root: the largest integer `m` such that `m*m <= n`.

cbrt(*x*)
Return $x^{1/3}$. The prefix operator $\sqrt[3]{}$ is equivalent to `cbrt`.

erf(*x*)
Compute the error function of `x`, defined by $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$ for arbitrary complex `x`.

erfc(*x*)
Compute the complementary error function of `x`, defined by $1 - \operatorname{erf}(x)$.

erfcx(*x*)
Compute the scaled complementary error function of `x`, defined by $e^{x^2} \operatorname{erfc}(x)$. Note also that $\operatorname{erfcx}(-ix)$ computes the Faddeeva function $w(x)$.

erfi(*x*)
Compute the imaginary error function of `x`, defined by $-i \operatorname{erf}(ix)$.

dawson(*x*)
Compute the Dawson function (scaled imaginary error function) of `x`, defined by $\frac{\sqrt{\pi}}{2} e^{-x^2} \operatorname{erfi}(x)$.

erfinv(*x*)
Compute the inverse error function of a real `x`, defined by $\operatorname{erf}(\operatorname{erfinv}(x)) = x$.

erfcinv(*x*)
Compute the inverse error complementary function of a real `x`, defined by $\operatorname{erfc}(\operatorname{erfcinv}(x)) = x$.

real(*z*)
Return the real part of the complex number `z`

imag(*z*)
Return the imaginary part of the complex number `z`

reim(*z*)
Return both the real and imaginary parts of the complex number `z`

conj(*z*)
Compute the complex conjugate of a complex number `z`

angle(*z*)
Compute the phase angle of a complex number `z`

cis(*z*)
Return $\exp(iz)$.

binomial(*n*, *k*)
Number of ways to choose k out of n items

factorial(*n*)
Factorial of n

factorial(*n*, *k*)
Compute factorial(n)/factorial(k)

factor(*n*) → Dict
Compute the prime factorization of an integer n. Returns a dictionary. The keys of the dictionary correspond to the factors, and hence are of the same type as n. The value associated with each key indicates the number of times the factor appears in the factorization.

```
julia> factor(100) # == 2*2*5*5
Dict{Int64,Int64} with 2 entries:
  2 => 2
  5 => 2
```

gcd(*x*, *y*)
Greatest common (positive) divisor (or zero if x and y are both zero).

lcm(*x*, *y*)
Least common (non-negative) multiple.

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 $ux + vy = d = gcd(x, y)$.

```
julia> gcdx(12, 42)
(6,-3,1)
```

```
julia> gcdx(240, 46)
(2,-9,47)
```

注解: Bézout coefficients are *not* uniquely defined. gcdx returns the minimal Bézout coefficients that are computed by the extended Euclid algorithm. (Ref: D. Knuth, TAoCP, 2/e, p. 325, Algorithm X.) These coefficients u and v are minimal in the sense that $|u| < |\frac{y}{d}$ and $|v| < |\frac{x}{d}$. Furthermore, the signs of u and v are chosen so that d is positive.

---

ispow2(*n*) → Bool
Test whether n is a power of two

nextpow2(*n*)
The smallest power of two not less than n. Returns 0 for n==0, and returns -nextpow2(-n) for negative arguments.

prevpow2(*n*)
The largest power of two not greater than n. Returns 0 for n==0, and returns -prevpow2(-n) for negative arguments.

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.

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.

nextprod([*k_1*, *k_2*, ...], *n*)
Next integer not less than `n` that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2, etc.

prevprod([*k_1*, *k_2*, ...], *n*)
Previous integer not greater than `n` that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2, etc.

invmod(*x*, *m*)
Take the inverse of `x` modulo `m`: `y` such that $xy = 1 \pmod m$

powermod(*x*, *p*, *m*)
Compute $x^p \pmod m$

gamma(*x*)
Compute the gamma function of `x`

lgamma(*x*)
Compute the logarithm of absolute value of `gamma(x)`

lfact(*x*)
Compute the logarithmic factorial of `x`

digamma(*x*)
Compute the digamma function of `x` (the logarithmic derivative of `gamma(x)`)

invdigamma(*x*)
Compute the inverse digamma function of `x`.

trigamma(*x*)
Compute the trigamma function of `x` (the logarithmic second derivative of `gamma(x)`)

polygamma(*m*, *x*)
Compute the polygamma function of order `m` of argument `x` (the `(m+1)th` derivative of the logarithm of `gamma(x)`)

airy(*k*, *x*)
kth derivative of the Airy function $\mathrm{Ai}(x)$.

airyai(*x*)
Airy function $\mathrm{Ai}(x)$.

airyprime(*x*)
Airy function derivative $\mathrm{Ai}'(x)$.

airyaiprime(*x*)
Airy function derivative $\mathrm{Ai}'(x)$.

airybi(*x*)
Airy function $\mathrm{Bi}(x)$.

airybiprime(*x*)
Airy function derivative $\mathrm{Bi}'(x)$.

airyx(*k*, *x*)
scaled kth derivative of the Airy function, return $\mathrm{Ai}(x)e^{\frac{2}{3}x\sqrt{x}}$ for `k == 0 || k == 1`, and $\mathrm{Ai}(x)e^{-\left|\mathrm{Re}\left(\frac{2}{3}x\sqrt{x}\right)\right|}$ for `k == 2 || k == 3`.

besselj0(*x*)
Bessel function of the first kind of order 0, $J_0(x)$.

besselj1(*x*)
Bessel function of the first kind of order 1, $J_1(x)$.

besselj(*nu*, *x*)
Bessel function of the first kind of order `nu`, $J_\nu(x)$.

besseljx(*nu*, *x*)
Scaled Bessel function of the first kind of order `nu`, $J_\nu(x)e^{-|\operatorname{Im}(x)|}$.

bessely0(*x*)
Bessel function of the second kind of order 0, $Y_0(x)$.

bessely1(*x*)
Bessel function of the second kind of order 1, $Y_1(x)$.

bessely(*nu*, *x*)
Bessel function of the second kind of order `nu`, $Y_\nu(x)$.

besselyx(*nu*, *x*)
Scaled Bessel function of the second kind of order `nu`, $Y_\nu(x)e^{-|\operatorname{Im}(x)|}$.

hankelh1(*nu*, *x*)
Bessel function of the third kind of order `nu`, $H_\nu^{(1)}(x)$.

hankelh1x(*nu*, *x*)
Scaled Bessel function of the third kind of order `nu`, $H_\nu^{(1)}(x)e^{-xi}$.

hankelh2(*nu*, *x*)
Bessel function of the third kind of order `nu`, $H_\nu^{(2)}(x)$.

hankelh2x(*nu*, *x*)
Scaled Bessel function of the third kind of order `nu`, $H_\nu^{(2)}(x)e^{xi}$.

besselh(*nu*, *k*, *x*)
Bessel function of the third kind of order `nu` (Hankel function). `k` is either 1 or 2, selecting `hankelh1` or `hankelh2`, respectively.

besseli(*nu*, *x*)
Modified Bessel function of the first kind of order `nu`, $I_\nu(x)$.

besselix(*nu*, *x*)
Scaled modified Bessel function of the first kind of order `nu`, $I_\nu(x)e^{-|\operatorname{Re}(x)|}$.

besselk(*nu*, *x*)
Modified Bessel function of the second kind of order `nu`, $K_\nu(x)$.

besselkx(*nu*, *x*)
Scaled modified Bessel function of the second kind of order `nu`, $K_\nu(x)e^x$.

beta(*x*, *y*)
Euler integral of the first kind $\operatorname{B}(x,y) = \Gamma(x)\Gamma(y)/\Gamma(x+y)$.

lbeta(*x*, *y*)
Natural logarithm of the absolute value of the beta function $\log(|\operatorname{B}(x,y)|)$.

eta(*x*)
Dirichlet eta function $\eta(s) = \sum_{n=1}^{\infty}(-)^{n-1}/n^s$.

zeta(*s*)
Riemann zeta function $\zeta(s)$.

zeta(*s*, *z*)
Hurwitz zeta function $\zeta(s,z)$. (This is equivalent to the Riemann zeta function $\zeta(s)$ for the case of `z=1`.)

ndigits(*n*, *b*)
Compute the number of digits in number `n` written in base `b`.

widemul(*x*, *y*)
Multiply `x` and `y`, giving the result as a larger type.

@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.

Data Formats

bin(*n*[, *pad*])
Convert an integer to a binary string, optionally specifying a number of digits to pad to.

hex(*n*[, *pad*])
Convert an integer to a hexadecimal string, optionally specifying a number of digits to pad to.

dec(*n*[, *pad*])
Convert an integer to a decimal string, optionally specifying a number of digits to pad to.

oct(*n*[, *pad*])
Convert an integer to an octal string, optionally specifying a number of digits to pad to.

base(*base*, *n*[, *pad*])
Convert an integer to a string in the given base, optionally specifying a number of digits to pad to. The base can be specified as either an integer, or as a `Uint8` array of character values to use as digit symbols.

digits(*n*[, *base*][, *pad*])
Returns an array of the digits of `n` in the given base, optionally padded with zeros to a specified size. More significant digits are at higher indexes, such that `n == sum([digits[k]*base^(k-1) for k=1:length(digits)])`.

digits!(*array*, *n*[, *base*])
Fills an array of the digits of `n` in the given base. More significant digits are at higher indexes. 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.

bits(*n*)
A string giving the literal bit representation of a number.

parseint([*type*], *str*[, *base*])
Parse a string as an integer in the given base (default 10), yielding a number of the specified type (default `Int`).

parsefloat([*type*], *str*)
Parse a string as a decimal floating point number, yielding a number of the specified type.

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.

bool(*x*)
Convert a number or numeric array to boolean

int(*x*)
Convert a number or array to the default integer type on your platform. Alternatively, `x` can be a string, which is parsed as an integer.

uint(*x*)
Convert a number or array to the default unsigned integer type on your platform. Alternatively, `x` can be a string, which is parsed as an unsigned integer.

integer(*x*)
Convert a number or array to integer type. If `x` is already of integer type it is unchanged, otherwise it converts it to the default integer type on your platform.

signed(*x*)
Convert a number to a signed integer

unsigned(*x*) → Unsigned
Convert a number to an unsigned integer

int8(*x*)
Convert a number or array to `Int8` data type

int16(*x*)
Convert a number or array to `Int16` data type

int32(*x*)
Convert a number or array to `Int32` data type

int64(*x*)
Convert a number or array to `Int64` data type

int128(*x*)
Convert a number or array to `Int128` data type

uint8(*x*)
Convert a number or array to `Uint8` data type

uint16(*x*)
Convert a number or array to `Uint16` data type

uint32(*x*)
Convert a number or array to `Uint32` data type

uint64(*x*)
Convert a number or array to `Uint64` data type

uint128(*x*)
Convert a number or array to `Uint128` data type

float16(*x*)
Convert a number or array to `Float16` data type

float32(*x*)
Convert a number or array to `Float32` data type

float64(*x*)
Convert a number or array to `Float64` data type

float32_isvalid(*x*, *out::Vector{Float32}*) → Bool
Convert a number or array to `Float32` data type, returning true if successful. The result of the conversion is stored in `out[1]`.

float64_isvalid(*x*, *out::Vector{Float64}*) → Bool
Convert a number or array to `Float64` data type, returning true if successful. The result of the conversion is stored in `out[1]`.

float(*x*)
Convert a number, array, or string to a `FloatingPoint` data type. For numeric data, the smallest suitable `FloatingPoint` type is used. Converts strings to `Float64`.

This function is not recommended for arrays. It is better to use a more specific function such as `float32` or `float64`.

significand(*x*)
Extract the significand(s) (a.k.a. mantissa), in binary representation, of a floating-point number or array.

```
julia> significand(15.2)/15.2
0.125

julia> significand(15.2)*8
15.2
```

exponent(*x*) → Int
Get the exponent of a normalized floating-point number.

complex64(*r*[, *i*])
Convert to `r + i*im` represented as a `Complex64` data type. `i` defaults to zero.

complex128(*r*[, *i*])
Convert to `r + i*im` represented as a `Complex128` data type. `i` defaults to zero.

complex(*r*[, *i*])
Convert real numbers or arrays to complex. `i` defaults to zero.

char(*x*)
Convert a number or array to `Char` data type

bswap(*n*)
Byte-swap an integer

num2hex(*f*)
Get a hexadecimal string of the binary representation of a floating point number

hex2num(*str*)
Convert a hexadecimal string to the floating point number it represents

hex2bytes(*s::ASCIIString*)
Convert an arbitrarily long hexadecimal string to its binary representation. Returns an Array{Uint8, 1}, i.e. an array of bytes.

bytes2hex(*bin_arr::Array{Uint8, 1}*)
Convert an array of bytes to its hexadecimal representation. All characters are in lower-case. Returns an ASCIIString.

Numbers

one(*x*)
Get the multiplicative identity element for the type of x (x can also specify the type itself). For matrices, returns an identity matrix of the appropriate size and type.

zero(*x*)
Get the additive identity element for the type of x (x can also specify the type itself).

pi
The constant pi

im
The imaginary unit

e
The constant e

catalan
Catalan's constant

Inf
Positive infinity of type Float64

Inf32
Positive infinity of type Float32

Inf16
Positive infinity of type Float16

NaN
A not-a-number value of type Float64

NaN32
A not-a-number value of type Float32

NaN16
A not-a-number value of type Float16

issubnormal(*f*) → Bool
Test whether a floating point number is subnormal

isfinite(*f*) → Bool
Test whether a number is finite

isinf(*f*) → Bool
Test whether a number is infinite

isnan(*f*) → Bool
Test whether a floating point number is not a number (NaN)

inf(*f*)
Returns positive infinity of the floating point type `f` or of the same floating point type as `f`

nan(*f*)
Returns NaN (not-a-number) of the floating point type `f` or of the same floating point type as `f`

nextfloat(*f*)
Get the next floating point number in lexicographic order

prevfloat(*f*) → FloatingPoint
Get the previous floating point number in lexicographic order

isinteger(*x*) → Bool
Test whether `x` or all its elements are numerically equal to some integer

isreal(*x*) → Bool
Test whether `x` or all its elements are numerically equal to some real number

BigInt(*x*)
Create an arbitrary precision integer. `x` may be an `Int` (or anything that can be converted to an `Int`) or a `String`. The usual mathematical operators are defined for this type, and results are promoted to a `BigInt`.

BigFloat(*x*)
Create an arbitrary precision floating point number. `x` may be an `Integer`, a `Float64`, a `String` or a `BigInt`. The usual mathematical operators are defined for this type, and results are promoted to a `BigFloat`. Note that because floating-point numbers are not exactly-representable in decimal notation, `BigFloat(2.1)` may not yield what you expect. You may prefer to initialize constants using strings, e.g., `BigFloat("2.1")`.

get_rounding(*T*)
Get the current floating point rounding mode for type `T`. Valid modes are `RoundNearest`, `RoundToZero`, `RoundUp`, `RoundDown`, and `RoundFromZero` (`BigFloat` only).

set_rounding(*T*, *mode*)
Set the rounding mode of floating point type `T`. Note that this may affect other types, for instance changing the rounding mode of `Float64` will change the rounding mode of `Float32`. See `get_rounding` for available modes

with_rounding(*f::Function*, *T*, *mode*)
Change the rounding mode of floating point type `T` for the duration of `f`. It is logically equivalent to:

```
old = get_rounding(T)
set_rounding(T, mode)
f()
set_rounding(T, old)
```

See `get_rounding` for available rounding modes.

Integers

count_ones(*x::Integer*) → Integer
Number of ones in the binary representation of `x`.

```
julia> count_ones(7)
3
```

count_zeros(*x::Integer*) → Integer
Number of zeros in the binary representation of `x`.

```
julia> count_zeros(int32(2 ^ 16 - 1))
16
```

leading_zeros(*x::Integer*) → Integer
Number of zeros leading the binary representation of `x`.

```
julia> leading_zeros(int32(1))
31
```

leading_ones(*x::Integer*) → Integer
Number of ones leading the binary representation of `x`.

```
julia> leading_ones(int32(2 ^ 32 - 2))
31
```

trailing_zeros(*x::Integer*) → Integer
Number of zeros trailing the binary representation of `x`.

```
julia> trailing_zeros(2)
1
```

trailing_ones(*x::Integer*) → Integer
Number of ones trailing the binary representation of `x`.

```
julia> trailing_ones(3)
2
```

isprime(*x::Integer*) → Bool
Returns `true` if `x` is prime, and `false` otherwise.

```
julia> isprime(3)
true
```

primes(*n*)
Returns a collection of the prime numbers <= `n`.

isodd(*x::Integer*) → Bool
Returns `true` if `x` is odd (that is, not divisible by 2), and `false` otherwise.

```
julia> isodd(9)
true

julia> isodd(10)
false
```

iseven(*x::Integer*) → Bool
Returns `true` is `x` is even (that is, divisible by 2), and `false` otherwise.

```
julia> iseven(9)
false

julia> iseven(10)
true
```

BigFloats

The *BigFloat* type implements arbitrary-precision floating-point aritmetic using the GNU MPFR library.

precision(*num::FloatingPoint*)
Get the precision of a floating point number, as defined by the effective number of bits in the mantissa.

get_bigfloat_precision()
Get the precision (in bits) currently used for BigFloat arithmetic.

set_bigfloat_precision(*x::Int64*)
Set the precision (in bits) to be used to BigFloat arithmetic.

with_bigfloat_precision(*f::Function*, *precision::Integer*)
Change the BigFloat arithmetic precision (in bits) for the duration of `f`. It is logically equivalent to:

```
old = get_bigfloat_precision()
set_bigfloat_precision(precision)
f()
set_bigfloat_precision(old)
```

Random Numbers

Random number generation in Julia uses the Mersenne Twister library. Julia has a global RNG, which is used by default. Multiple RNGs can be plugged in using the `AbstractRNG` object, which can then be used to have multiple streams of random numbers. Currently, only `MersenneTwister` is supported.

srand([*rng*], *seed*)
Seed the RNG with a `seed`, which may be an unsigned integer or a vector of unsigned integers. `seed` can even be a filename, in which case the seed is read from a file. If the argument `rng` is not provided, the default global RNG is seeded.

MersenneTwister([*seed*])
Create a `MersenneTwister` RNG object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers.

rand() → Float64
Generate a `Float64` random number uniformly in [0,1)

rand!([*rng*], *A*)
Populate the array A with random number generated from the specified RNG.

rand(*rng::AbstractRNG*[, *dims...*])
Generate a random `Float64` number or array of the size specified by dims, using the specified RNG object. Currently, `MersenneTwister` is the only available Random Number Generator (RNG), which may be seeded using srand.

rand(*dims or [dims...]*)
Generate a random `Float64` array of the size specified by dims

rand(*Int32*|*Uint32*|*Int64*|*Uint64*|*Int128*|*Uint128*[, *dims...*])
Generate a random integer of the given type. Optionally, generate an array of random integers of the given type by specifying dims.

rand(*r*[, *dims...*])
Generate a random integer from the inclusive interval specified by `Range1 r` (for example, `1:n`). Optionally, generate a random integer array.

randbool([*dims...*])
Generate a random boolean value. Optionally, generate an array of random boolean values.

randbool!(*A*)
Fill an array with random boolean values. A may be an `Array` or a `BitArray`.

randn(*[rng], dims or [dims...]*)
Generate a normally-distributed random number with mean 0 and standard deviation 1. Optionally generate an array of normally-distributed random numbers.

randn!([*rng*], *A::Array{Float64, N}*)
Fill the array A with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the rand function.

Arrays

Basic functions

ndims(*A*) → Integer
Returns the number of dimensions of A

size(*A*)
Returns a tuple containing the dimensions of A

iseltype(*A*, *T*)
Tests whether A or its elements are of type T

length(*A*) → Integer
Returns the number of elements in A

countnz(*A*)
Counts the number of nonzero values in array A (dense or sparse). Note that this is not a constant-time operation. For sparse matrices, one should usually use `nnz`, which returns the number of stored values.

conj!(*A*)
Convert an array to its complex conjugate in-place

stride(*A*, *k*)
Returns the distance in memory (in number of elements) between adjacent elements in dimension k

strides(*A*)
Returns a tuple of the memory strides in each dimension

ind2sub(*dims*, *index*) → subscripts
Returns a tuple of subscripts into an array with dimensions `dims`, corresponding to the linear index `index`

Example `i, j, ... = ind2sub(size(A), indmax(A))` provides the indices of the maximum element

sub2ind(*dims*, *i*, *j*, *k...*) → index
The inverse of `ind2sub`, returns the linear index corresponding to the provided subscripts

Constructors

Array(*type*, *dims*)
Construct an uninitialized dense array. `dims` may be a tuple or a series of integer arguments.

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,...]`.

cell(*dims*)
Construct an uninitialized cell array (heterogeneous array). `dims` can be either a tuple or a series of integer arguments.

zeros(*type*, *dims*)
Create an array of all zeros of specified type

ones(*type*, *dims*)
Create an array of all ones of specified type

trues(*dims*)
Create a `BitArray` with all values set to true

falses(*dims*)
Create a `BitArray` with all values set to false

fill(*v*, *dims*)
Create an array filled with `v`

fill!(*A*, *x*)
Fill array `A` with value `x`

reshape(*A*, *dims*)
Create an array with the same data as the given array, but with different dimensions. An implementation for a particular type of array may choose whether the data is copied or shared.

similar(*array*, *element_type*, *dims*)
Create an uninitialized array of the same type as the given array, but with the specified element type and dimensions. The second and third arguments are both optional. The `dims` argument may be a tuple or a series of integer arguments.

reinterpret(*type*, *A*)
Change the type-interpretation of a block of memory. For example, `reinterpret(Float32, uint32(7))` interprets the 4 bytes corresponding to `uint32(7)` as a `Float32`. For arrays, this constructs an array with the same binary data as the given array, but with the specified element type.

eye(*n*)
n-by-n identity matrix

eye(*m*, *n*)
m-by-n identity matrix

eye(*A*)
Constructs an identity matrix of the same dimensions and type as `A`.

linspace(*start*, *stop*, *n*)
Construct a vector of `n` linearly-spaced elements from `start` to `stop`. See also: `linrange()` that constructs a range object.

logspace(*start*, *stop*, *n*)
Construct a vector of `n` logarithmically-spaced numbers from `10^start` to `10^stop`.

Mathematical operators and functions

All mathematical operations and functions are supported for arrays

broadcast(*f*, *As...*)
Broadcasts the arrays `As` to a common size by expanding singleton dimensions, and returns an array of the results `f(as...)` for each position.

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)`.

bitbroadcast(*f*, *As...*)
Like `broadcast`, but allocates a `BitArray` to store the result, rather then an `Array`.

broadcast_function(*f*)
Returns a function `broadcast_f` such that `broadcast_function(f)(As...) === broadcast(f, As...)`. Most useful in the form `const broadcast_f = broadcast_function(f)`.

broadcast!_function(*f*)
Like `broadcast_function`, but for `broadcast!`.

Indexing, Assignment, and Concatenation

getindex(*A*, *inds...*)
Returns a subset of array `A` as specified by `inds`, where each `ind` may be an `Int`, a `Range`, or a `Vector`.

sub(*A*, *inds...*)
Returns a SubArray, which stores the input `A` and `inds` rather than computing the result immediately. Calling `getindex` on a SubArray computes the indices on the fly.

parent(*A*)
Returns the "parent array" of an array view type (e.g., SubArray), or the array itself if it is not a view

parentindexes(*A*)
From an array view `A`, returns the corresponding indexes in the parent

slicedim(*A*, *d*, *i*)
Return all the data of `A` where the index for dimension `d` equals `i`. Equivalent to `A[:,:,...,i,:,:,...]` where `i` is in position `d`.

slice(*A*, *inds...*)
Create a view of the given indexes of array `A`, dropping dimensions indexed with scalars.

setindex!(*A*, *X*, *inds...*)
Store values from array `X` within some subset of `A` as specified by `inds`.

broadcast_getindex(*A*, *inds...*)
Broadcasts the `inds` arrays to a common size like `broadcast`, and returns an array of the results `A[ks...]`, where `ks` goes over the positions in the broadcast.

broadcast_setindex!(*A*, *X*, *inds...*)
Broadcasts the `X` and `inds` arrays to a common size and stores the value from each position in `X` at the indices given by the same positions in `inds`.

cat(*dim*, *A...*)
Concatenate the input arrays along the specified dimension

vcat(*A...*)
Concatenate along dimension 1

hcat(*A...*)
Concatenate along dimension 2

hvcat(*rows::(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. For example, `[a b;c d e]` calls `hvcat((2,3),a,b,c,d,e)`.

If the first argument is a single integer `n`, then all block rows are assumed to have `n` block columns.

flipdim(*A*, *d*)
Reverse `A` in dimension `d`.

flipud(*A*)
Equivalent to `flipdim(A,1)`.

fliplr(*A*)
Equivalent to `flipdim(A,2)`.

circshift(*A*, *shifts*)
Circularly shift the data in an array. The second argument is a vector giving the amount to shift in each dimension.

find(*A*)
Return a vector of the linear indexes of the non-zeros in `A` (determined by `A[i]!=0`). A common use of this is to convert a boolean array to an array of indexes of the `true` elements.

find(*f*, *A*)
Return a vector of the linear indexes of `A` where `f` returns true.

findn(*A*)
Return a vector of indexes for each dimension giving the locations of the non-zeros in `A` (determined by `A[i]!=0`).

findnz(*A*)
Return a tuple `(I, J, V)` where `I` and `J` are the row and column indexes of the non-zero values in matrix `A`, and `V` is a vector of the non-zero values.

findfirst(*A*)
Return the index of the first non-zero value in `A` (determined by `A[i]!=0`).

findfirst(*A*, *v*)
Return the index of the first element equal to `v` in `A`.

findfirst(*predicate*, *A*)
Return the index of the first element of `A` for which `predicate` returns true.

findnext(*A*, *i*)
Find the next index >= `i` of a non-zero element of `A`, or `0` if not found.

findnext(*predicate*, *A*, *i*)
Find the next index >= `i` of an element of `A` for which `predicate` returns true, or `0` if not found.

findnext(*A*, *v*, *i*)
Find the next index >= `i` of an element of `A` equal to `v` (using ==), or `0` if not found.

permutedims(*A*, *perm*)
Permute the dimensions of array `A`. `perm` is a vector specifying a permutation of length `ndims(A)`. This is a generalization of transpose for multi-dimensional arrays. Transpose is equivalent to `permutedims(A,[2, 1])`.

ipermutedims(*A*, *perm*)
Like `permutedims()`, except the inverse of the given permutation is applied.

squeeze(*A*, *dims*)
Remove the dimensions specified by `dims` from array `A`

vec(*Array*) → Vector
Vectorize an array using column-major convention.

promote_shape(*s1*, *s2*)
Check two array shapes for compatibility, allowing trailing singleton dimensions, and return whichever shape has more dimensions.

checkbounds(*array*, *indexes...*)
Throw an error if the specified indexes are not in bounds for the given array.

randsubseq(*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`.

randsubseq!(*S*, *A*, *p*)
Like `randsubseq`, but the results are stored in `S` (which is resized as needed).

Array functions

cumprod(*A*[, *dim*])
Cumulative product along a dimension.

cumprod!(*B*, *A*[, *dim*])
Cumulative product of `A` along a dimension, storing the result in `B`.

cumsum(*A*[, *dim*])
Cumulative sum along a dimension.

cumsum! (*B*, *A*[, *dim*])
Cumulative sum of `A` along a dimension, storing the result in `B`.

cumsum_kbn (*A*[, *dim*])
Cumulative sum along a dimension, using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy.

cummin (*A*[, *dim*])
Cumulative minimum along a dimension.

cummax (*A*[, *dim*])
Cumulative maximum along a dimension.

diff (*A*[, *dim*])
Finite difference operator of matrix or vector.

gradient (*F*[, *h*])
Compute differences along vector `F`, using `h` as the spacing between points. The default spacing is one.

rot180 (*A*)
Rotate matrix `A` 180 degrees.

rotl90 (*A*)
Rotate matrix `A` left 90 degrees.

rotr90 (*A*)
Rotate matrix `A` right 90 degrees.

reducedim (*f*, *A*, *dims*, *initial*)
Reduce 2-argument function `f` along dimensions of `A`. `dims` is a vector specifying the dimensions to reduce, and `initial` is the initial value to use in the reductions.

The associativity of the reduction is implementation-dependent; if you need a particular associativity, e.g. left-to-right, you should write your own loop. See documentation for `reduce`.

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`.

sum_kbn (*A*)
Returns the sum of all array elements, using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy.

cartesianmap (*f*, *dims*)
Given a `dims` tuple of integers `(m, n, ...)`, call `f` on all combinations of integers in the ranges `1:m`, `1:n`, etc.

```
julia> cartesianmap(println, (2,2))
11
21
12
22
```

BitArrays

bitpack (*A::AbstractArray{T, N}*) → BitArray
Converts a numeric array to a packed boolean array

bitunpack(*B::BitArray{N}*) → Array{Bool,N}
Converts a packed boolean array to an array of booleans

flipbits!(*B::BitArray{N}*) → BitArray{N}
Performs a bitwise not operation on B. See *~ operator*.

rol(*B::BitArray{1}*, *i::Integer*) → BitArray{1}
Left rotation operator.

ror(*B::BitArray{1}*, *i::Integer*) → BitArray{1}
Right rotation operator.

Combinatorics

nthperm(*v*, *k*)
Compute the kth lexicographic permutation of a vector.

nthperm(*p*)
Return the `k` that generated permutation `p`. Note that `nthperm(nthperm([1:n], k)) == k` for `1 <= k <= factorial(n)`.

nthperm!(*v*, *k*)
In-place version of `nthperm()`.

randperm(*n*)
Construct a random permutation of the given length.

invperm(*v*)
Return the inverse permutation of v.

isperm(*v*) → Bool
Returns true if v is a valid permutation.

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.

ipermute!(*v*, *p*)
Like permute!, but the inverse of the given permutation is applied.

randcycle(*n*)
Construct a random cyclic permutation of the given length.

shuffle(*v*)
Return a randomly permuted copy of `v`.

shuffle!(*v*)
In-place version of `shuffle()`.

reverse(*v*[, *start=1*[, *stop=length(v)*]])
Return a copy of `v` reversed from start to stop.

reverse!(*v*[, *start=1*[, *stop=length(v)*]]) → v
In-place version of `reverse()`.

combinations(*arr*, *n*)
Generate all combinations of `n` elements from an indexable object. Because the number of combinations can be very large, this function returns an iterator object. Use `collect(combinations(a,n))` to get an array of all combinations.

permutations(*arr*)
Generate all permutations of an indexable object. Because the number of permutations can be very large, this function returns an iterator object. Use `collect(permutations(a,n))` to get an array of all permutations.

partitions(*n*)
Generate all integer arrays that sum to `n`. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(n))` to get an array of all partitions. The number of partitions to generete can be efficiently computed using `length(partitions(n))`.

partitions(*n*, *m*)
Generate all arrays of `m` integers that sum to `n`. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(n,m))` to get an array of all partitions. The number of partitions to generete can be efficiently computed using `length(partitions(n,m))`.

partitions(*array*)
Generate all set partitions of the elements of an array, represented as arrays of arrays. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(array))` to get an array of all partitions. The number of partitions to generete can be efficiently computed using `length(partitions(array))`.

partitions(*array*, *m*)
Generate all set partitions of the elements of an array into exactly m subsets, represented as arrays of arrays. Because the number of partitions can be very large, this function returns an iterator object. Use `collect(partitions(array,m))` to get an array of all partitions. The number of partitions into m subsets is equal to the Stirling number of the second kind and can be efficiently computed using `length(partitions(array,m))`.

Statistics

mean(*v*[, *region*])
Compute the mean of whole array `v`, or optionally along the dimensions in `region`. Note: Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `DataArray` package is recommended.

mean!(*r*, *v*)
Compute the mean of `v` over the singleton dimensions of `r`, and write results to `r`.

std(*v*[, *region*])
Compute the sample standard deviation of a vector or array `v`, optionally along dimensions in `region`. The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `v` is an IID drawn from that generative distribution. This computation is equivalent to calculating `sqrt(sum((v - mean(v)).^2) / (length(v) - 1))`. Note: Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `DataArray` package is recommended.

stdm(*v*, *m*)
Compute the sample standard deviation of a vector `v` with known mean `m`. Note: Julia does not ignore `NaN` values in the computation.

var(*v*[, *region*])
Compute the sample variance of a vector or array `v`, optionally along dimensions in `region`. The algorithm will return an estimator of the generative distribution's variance under the assumption that each entry of `v` is an IID drawn from that generative distribution. This computation is equivalent to calculating `sum((v - mean(v)).^2) / (length(v) - 1)`. Note: Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `DataArray` package is recommended.

varm(*v*, *m*)

Compute the sample variance of a vector `v` with known mean `m`. Note: Julia does not ignore `NaN` values in the computation.

median(*v; checknan::Bool=true*)

Compute the median of a vector `v`. If keyword argument `checknan` is true (the default), an error is raised for data containing NaN values. Note: Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `DataArray` package is recommended.

median!(*v; checknan::Bool=true*)

Like `median`, but may overwrite the input vector.

hist(*v*[, *n*]) → e, counts

Compute the histogram of `v`, optionally using approximately `n` bins. The return values are a range `e`, which correspond to the edges of the bins, and `counts` containing the number of elements of `v` in each bin. Note: Julia does not ignore `NaN` values in the computation.

hist(*v*, *e*) → e, counts

Compute the histogram of `v` using a vector/range `e` as the edges for the bins. The result will be a vector of length `length(e) - 1`, such that the element at location `i` satisfies `sum(e[i] .< v .<= e[i+1])`. Note: Julia does not ignore `NaN` values in the computation.

hist!(*counts*, *v*, *e*) → e, counts

Compute the histogram of `v`, using a vector/range `e` as the edges for the bins. This function writes the resultant counts to a pre-allocated array `counts`.

hist2d(*M*, *e1*, *e2) -> (edge1*, *edge2*, *counts*)

Compute a "2d histogram" of a set of N points specified by N-by-2 matrix `M`. Arguments `e1` and `e2` are bins for each dimension, specified either as integer bin counts or vectors of bin edges. The result is a tuple of `edge1` (the bin edges used in the first dimension), `edge2` (the bin edges used in the second dimension), and `counts`, a histogram matrix of size `(length(edge1)-1, length(edge2)-1)`. Note: Julia does not ignore `NaN` values in the computation.

hist2d!(*counts*, *M*, *e1*, *e2) -> (e1*, *e2*, *counts*)

Compute a "2d histogram" with respect to the bins delimited by the edges given in `e1` and `e2`. This function writes the results to a pre-allocated array `counts`.

histrange(*v*, *n*)

Compute *nice* bin ranges for the edges of a histogram of `v`, using approximately `n` bins. The resulting step sizes will be 1, 2 or 5 multiplied by a power of 10. Note: Julia does not ignore `NaN` values in the computation.

midpoints(*e*)

Compute the midpoints of the bins with edges `e`. The result is a vector/range of length `length(e) - 1`. Note: Julia does not ignore `NaN` values in the computation.

quantile(*v*, *p*)

Compute the quantiles of a vector `v` at a specified set of probability values `p`. Note: Julia does not ignore `NaN` values in the computation.

quantile(*v*, *p*)

Compute the quantile of a vector `v` at the probability `p`. Note: Julia does not ignore `NaN` values in the computation.

quantile!(*v*, *p*)

Like `quantile`, but overwrites the input vector.

cov(*v1[, v2][, vardim=1, corrected=true, mean=nothing]*)

Compute the Pearson covariance between the vector(s) in `v1` and `v2`. Here, `v1` and `v2` can be either vectors or matrices.

This function accepts three keyword arguments:

- `vardim`: the dimension of variables. When `vardim = 1`, variables are considered in columns while observations in rows; when `vardim = 2`, variables are in rows while observations in columns. By default, it is set to `1`.
- `corrected`: whether to apply Bessel's correction (divide by `n-1` instead of `n`). By default, it is set to `true`.
- `mean`: allow users to supply mean values that are known. By default, it is set to `nothing`, which indicates that the mean(s) are unknown, and the function will compute the mean. Users can use `mean=0` to indicate that the input data are centered, and hence there's no need to subtract the mean.

The size of the result depends on the size of `v1` and `v2`. When both `v1` and `v2` are vectors, it returns the covariance between them as a scalar. When either one is a matrix, it returns a covariance matrix of size `(n1, n2)`, where `n1` and `n2` are the numbers of slices in `v1` and `v2`, which depend on the setting of `vardim`.

Note: `v2` can be omitted, which indicates `v2 = v1`.

cor(*v1[, v2][, vardim=1, mean=nothing]*)

Compute the Pearson correlation between the vector(s) in `v1` and `v2`.

Users can use the keyword argument `vardim` to specify the variable dimension, and `mean` to supply pre-computed mean values.

Signal Processing

Fast Fourier transform (FFT) functions in Julia are largely implemented by calling functions from FFTW.

fft(*A*[, *dims*])

Performs a multidimensional FFT of the array `A`. The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_fft()` for even greater efficiency.

A one-dimensional FFT computes the one-dimensional discrete Fourier transform (DFT) as defined by

$$\mathrm{DFT}(A)[k] = \sum_{n=1}^{\mathrm{length}(A)} \exp\left(-i\frac{2\pi(n-1)(k-1)}{\mathrm{length}(A)}\right) A[n].$$

A multidimensional FFT simply performs this operation along each transformed dimension of `A`.

fft!(*A*[, *dims*])

Same as `fft()`, but operates in-place on `A`, which must be an array of complex floating-point numbers.

ifft(*A*[, *dims*])

Multidimensional inverse FFT.

A one-dimensional inverse FFT computes

$$\mathrm{IDFT}(A)[k] = \frac{1}{\mathrm{length}(A)} \sum_{n=1}^{\mathrm{length}(A)} \exp\left(+i\frac{2\pi(n-1)(k-1)}{\mathrm{length}(A)}\right) A[n].$$

A multidimensional inverse FFT simply performs this operation along each transformed dimension of `A`.

ifft!(*A*[, *dims*])

Same as `ifft()`, but operates in-place on `A`.

bfft(*A*[, *dims*])

Similar to `ifft()`, but computes an unnormalized inverse (backward) transform, which must be divided by the product of the sizes of the transformed dimensions in order to obtain the inverse. (This is slightly more

efficient than `ifft()` because it omits a scaling step, which in some applications can be combined with other computational steps elsewhere.)

$$\mathrm{BDFT}(A)[k] = \mathrm{length}(A)\,\mathrm{IDFT}(A)[k]$$

bfft! (*A*[, *dims*])
Same as `bfft()`, but operates in-place on `A`.

plan_fft (*A*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized FFT along given dimensions (`dims`) of arrays matching the shape and type of `A`. (The first two arguments have the same meaning as for `fft()`.) Returns a function `plan(A)` that computes `fft(A, dims)` quickly.

The `flags` argument is a bitwise-or of FFTW planner flags, defaulting to `FFTW.ESTIMATE`. e.g. passing `FFTW.MEASURE` or `FFTW.PATIENT` will instead spend several seconds (or more) benchmarking different possible FFT algorithms and picking the fastest one; see the FFTW manual for more information on planner flags. The optional `timelimit` argument specifies a rough upper bound on the allowed planning time, in seconds. Passing `FFTW.MEASURE` or `FFTW.PATIENT` may cause the input array `A` to be overwritten with zeros during plan creation.

`plan_fft!()` is the same as `plan_fft()` but creates a plan that operates in-place on its argument (which must be an array of complex floating-point numbers). `plan_ifft()` and so on are similar but produce plans that perform the equivalent of the inverse transforms `ifft()` and so on.

plan_ifft (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_fft()`, but produces a plan that performs inverse transforms `ifft()`.

plan_bfft (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_fft()`, but produces a plan that performs an unnormalized backwards transform `bfft()`.

plan_fft! (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_fft()`, but operates in-place on `A`.

plan_ifft! (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_ifft()`, but operates in-place on `A`.

plan_bfft! (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_bfft()`, but operates in-place on `A`.

rfft (*A*[, *dims*])
Multidimensional FFT of a real array A, exploiting the fact that the transform has conjugate symmetry in order to save roughly half the computational time and storage costs compared with `fft()`. If `A` has size `(n_1, ..., n_d)`, the result has size `(floor(n_1/2)+1, ..., n_d)`.

The optional `dims` argument specifies an iterable subset of one or more dimensions of `A` to transform, similar to `fft()`. Instead of (roughly) halving the first dimension of `A` in the result, the `dims[1]` dimension is (roughly) halved in the same way.

irfft (*A*, *d*[, *dims*])
Inverse of `rfft()`: for a complex array `A`, gives the corresponding real array whose FFT yields `A` in the first half. As for `rfft()`, `dims` is an optional subset of dimensions to transform, defaulting to `1:ndims(A)`.

`d` is the length of the transformed real array along the `dims[1]` dimension, which must satisfy `d == floor(size(A,dims[1])/2)+1`. (This parameter cannot be inferred from `size(A)` due to the possibility of rounding by the `floor` function here.)

brfft (*A*, *d*[, *dims*])
Similar to `irfft()` but computes an unnormalized inverse transform (similar to `bfft()`), which must be divided by the product of the sizes of the transformed dimensions (of the real output array) in order to obtain the inverse transform.

plan_rfft (*A*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized real-input FFT, similar to `plan_fft()` except for `rfft()` instead of `fft()`. The first two arguments, and the size of the transformed result, are the same as for `rfft()`.

plan_brfft (*A*, *d*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized real-input unnormalized transform, similar to `plan_rfft()` except for `brfft()` instead of `rfft()`. The first two arguments and the size of the transformed result, are the same as for `brfft()`.

plan_irfft (*A*, *d*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized inverse real-input FFT, similar to `plan_rfft()` except for `irfft()` and `brfft()`, respectively. The first three arguments have the same meaning as for `irfft()`.

dct (*A*[, *dims*])
Performs a multidimensional type-II discrete cosine transform (DCT) of the array `A`, using the unitary normalization of the DCT. The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_dct()` for even greater efficiency.

dct! (*A*[, *dims*])
Same as `dct!()`, except that it operates in-place on `A`, which must be an array of real or complex floating-point values.

idct (*A*[, *dims*])
Computes the multidimensional inverse discrete cosine transform (DCT) of the array `A` (technically, a type-III DCT with the unitary normalization). The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see `nextprod()`. See also `plan_idct()` for even greater efficiency.

idct! (*A*[, *dims*])
Same as `idct!()`, but operates in-place on `A`.

plan_dct (*A*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized discrete cosine transform (DCT), similar to `plan_fft()` except producing a function that computes `dct()`. The first two arguments have the same meaning as for `dct()`.

plan_dct! (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_dct()`, but operates in-place on `A`.

plan_idct (*A*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized inverse discrete cosine transform (DCT), similar to `plan_fft()` except producing a function that computes `idct()`. The first two arguments have the same meaning as for `idct()`.

plan_idct! (*A*[, *dims*[, *flags*[, *timelimit*]]])
Same as `plan_idct()`, but operates in-place on `A`.

fftshift (*x*)
Swap the first and second halves of each dimension of `x`.

fftshift (*x*, *dim*)
Swap the first and second halves of the given dimension of array `x`.

ifftshift (*x*[, *dim*])
Undoes the effect of `fftshift`.

filt (*b*, *a*, *x*[, *si*])
Apply filter described by vectors `a` and `b` to vector `x`, with an optional initial filter state vector `si` (defaults to zeros).

filt! (*out*, *b*, *a*, *x*[, *si*])
Same as `filt()` but writes the result into the `out` argument, which may alias the input `x` to modify it in-place.

deconv(*b*, *a*)
Construct vector `c` such that `b = conv(a,c) + r`. Equivalent to polynomial division.

conv(*u*, *v*)
Convolution of two vectors. Uses FFT algorithm.

conv2(*u*, *v*, *A*)
2-D convolution of the matrix `A` with the 2-D separable kernel generated by the vectors `u` and `v`. Uses 2-D FFT algorithm

conv2(*B*, *A*)
2-D convolution of the matrix `B` with the matrix `A`. Uses 2-D FFT algorithm

xcorr(*u*, *v*)
Compute the cross-correlation of two vectors.

The following functions are defined within the `Base.FFTW` module.

r2r(*A*, *kind*[, *dims*])
Performs a multidimensional real-input/real-output (r2r) transform of type `kind` of the array `A`, as defined in the FFTW manual. `kind` specifies either a discrete cosine transform of various types (`FFTW.REDFT00`, `FFTW.REDFT01`, `FFTW.REDFT10`, or `FFTW.REDFT11`), a discrete sine transform of various types (`FFTW.RODFT00`, `FFTW.RODFT01`, `FFTW.RODFT10`, or `FFTW.RODFT11`), a real-input DFT with halfcomplex-format output (`FFTW.R2HC` and its inverse `FFTW.HC2R`), or a discrete Hartley transform (`FFTW.DHT`). The `kind` argument may be an array or tuple in order to specify different transform types along the different dimensions of `A`; `kind[end]` is used for any unspecified dimensions. See the FFTW manual for precise definitions of these transform types, at http://www.fftw.org/doc.

The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. `kind[i]` is then the transform type for `dims[i]`, with `kind[end]` being used for `i > length(kind)`.

See also `plan_r2r()` to pre-plan optimized r2r transforms.

r2r!(*A*, *kind*[, *dims*])
Same as `r2r()`, but operates in-place on `A`, which must be an array of real or complex floating-point numbers.

plan_r2r(*A*, *kind*[, *dims*[, *flags*[, *timelimit*]]])
Pre-plan an optimized r2r transform, similar to `Base.plan_fft()` except that the transforms (and the first three arguments) correspond to `r2r()` and `r2r!()`, respectively.

plan_r2r!(*A*, *kind*[, *dims*[, *flags*[, *timelimit*]]])
Similar to `Base.plan_fft()`, but corresponds to `r2r!()`.

Numerical Integration

Although several external packages are available for numeric integration and solution of ordinary differential equations, we also provide some built-in integration support in Julia.

quadgk(*f*, *a*, *b*, *c...; reltol=sqrt(eps)*, *abstol=0*, *maxevals=10^7*, *order=7*, *norm=vecnorm*)
Numerically integrate the function `f(x)` from `a` to `b`, and optionally over additional intervals `b` to `c` and so on. Keyword options include a relative error tolerance `reltol` (defaults to `sqrt(eps)` in the precision of the endpoints), an absolute error tolerance `abstol` (defaults to 0), a maximum number of function evaluations `maxevals` (defaults to `10^7`), and the `order` of the integration rule (defaults to 7).

Returns a pair `(I,E)` of the estimated integral `I` and an estimated upper bound on the absolute error `E`. If `maxevals` is not exceeded then `E <= max(abstol, reltol*norm(I))` will hold. (Note that it is useful to specify a positive `abstol` in cases where `norm(I)` may be zero.)

The endpoints `a` etcetera can also be complex (in which case the integral is performed over straight-line segments in the complex plane). If the endpoints are `BigFloat`, then the integration will be performed in `BigFloat` precision as well (note: it is advisable to increase the integration `order` in rough proportion to the precision, for smooth integrands). More generally, the precision is set by the precision of the integration endpoints (promoted to floating-point types).

The integrand `f(x)` can return any numeric scalar, vector, or matrix type, or in fact any type supporting `+`, `-`, multiplication by real values, and a `norm` (i.e., any normed vector space). Alternatively, a different norm can be specified by passing a *norm*-like function as the *norm* keyword argument (which defaults to *vecnorm*).

The algorithm is an adaptive Gauss-Kronrod integration technique: the integral in each interval is estimated using a Kronrod rule (`2*order+1` points) and the error is estimated using an embedded Gauss rule (`order` points). The interval with the largest error is then subdivided into two intervals and the process is repeated until the desired error tolerance is achieved.

These quadrature rules work best for smooth functions within each interval, so if your function has a known discontinuity or other singularity, it is best to subdivide your interval to put the singularity at an endpoint. For example, if `f` has a discontinuity at `x=0.7` and you want to integrate from 0 to 1, you should use `quadgk(f, 0,0.7,1)` to subdivide the interval at the point of discontinuity. The integrand is never evaluated exactly at the endpoints of the intervals, so it is possible to integrate functions that diverge at the endpoints as long as the singularity is integrable (for example, a `log(x)` or `1/sqrt(x)` singularity).

For real-valued endpoints, the starting and/or ending points may be infinite. (A coordinate transformation is performed internally to map the infinite interval to a finite one.)

Parallel Computing

addprocs(*n; cman::ClusterManager=LocalManager()*) → List of process identifiers

`addprocs(4)` will add 4 processes on the local machine. This can be used to take advantage of multiple cores.

Keyword argument `cman` can be used to provide a custom cluster manager to start workers. For example Beowulf clusters are supported via a custom cluster manager implemented in package `ClusterManagers`.

See the documentation for package `ClusterManagers` for more information on how to write a custom cluster manager.

addprocs(*machines; tunnel=false*, *dir=JULIA_HOME*, *sshflags::Cmd=``*) → 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 host definitions of the form `[user@]host[:port] [bind_addr]. user` defaults to current user, `port` to the standard ssh port. Optionally, in case of multi-homed hosts, `bind_addr` may be used to explicitly specify an interface.

Keyword arguments:

`tunnel` : if `true` then SSH tunneling will be used to connect to the worker.

`dir` : specifies the location of the julia binaries on the worker nodes.

`sshflags` : specifies additional ssh options, e.g. `sshflags=`-i /home/foo/bar.pem` .

nprocs()

Get the number of available processes.

nworkers()

Get the number of available worker processes. This is one less than nprocs(). Equal to nprocs() if nprocs() == 1.

procs()
Returns a list of all process identifiers.

workers()
Returns a list of all worker process identifiers.

rmprocs(*pids...*)
Removes the specified workers.

interrupt([*pids...*])
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.

myid()
Get the id of the current process.

pmap(*f*, *lsts...*; *err_retry=true*, *err_stop=false*)
Transform collections `lsts` by applying `f` to each element in parallel. If `nprocs() > 1`, the calling process will be dedicated to assigning tasks. All other available processes will be used as parallel workers.

If `err_retry` is true, it retries a failed application of `f` on a different worker. If `err_stop` is true, it takes precedence over the value of `err_retry` and `pmap` stops execution on the first error.

remotecall(*id*, *func*, *args...*)
Call a function asynchronously on the given arguments on the specified process. Returns a `RemoteRef`.

wait([*x*])
Block the current task until some event occurs, depending on the type of the argument:

- `RemoteRef`: Wait for a value to become available for the specified remote reference.
- `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, returning its result value.
- `RawFD`: Wait for changes on a file descriptor (see *poll_fd* for keyword arguments and return code)

If no argument is passed, the task blocks for an undefined period. If the task's state is set to `:waiting`, it can only be restarted by an explicit call to `schedule` or `yieldto`. If the task's state is `:runnable`, it might be restarted unpredictably.

Often `wait` is called within a `while` loop to ensure a waited-for condition is met before proceeding.

fetch(*RemoteRef*)
Wait for and get the value of a remote reference.

remotecall_wait(*id*, *func*, *args...*)
Perform `wait(remotecall(...))` in one message.

remotecall_fetch(*id*, *func*, *args...*)
Perform `fetch(remotecall(...))` in one message.

put!(*RemoteRef*, *value*)
Store a value to a remote reference. Implements "shared queue of length 1" semantics: if a value is already present, blocks until the value is removed with `take!`. Returns its first argument.

take!(*RemoteRef*)
Fetch the value of a remote reference, removing it so that the reference is empty again.

isready(*r::RemoteRef*)
Determine whether a `RemoteRef` 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. It is recommended that this function only be used on a `RemoteRef` that is assigned once.

If the argument `RemoteRef` is owned by a different node, this call will block to wait for the answer. It is recommended to wait for `r` in a separate task instead, or to use a local `RemoteRef` as a proxy:

```
rr = RemoteRef()
@async put!(rr, remotecall_fetch(p, long_computation))
isready(rr)  # will not block
```

RemoteRef()
Make an uninitialized remote reference on the local machine.

RemoteRef(*n*)
Make an uninitialized remote reference on process `n`.

timedwait(*testcb::Function*, *secs::Float64; pollint::Float64=0.1*)
Waits till `testcb` returns `true` or for `secs`` seconds, whichever is earlier. `testcb` is polled every `pollint` seconds.

@spawn()
Execute an expression on an automatically-chosen process, returning a `RemoteRef` to the result.

@spawnat()
Accepts two arguments, `p` and an expression, and runs the expression asynchronously on process `p`, returning a `RemoteRef` to the result.

@fetch()
Equivalent to `fetch(@spawn expr)`.

@fetchfrom()
Equivalent to `fetch(@spawnat p expr)`.

@async()
Schedule an expression to run on the local machine, also adding it to the set of items that the nearest enclosing `@sync` waits for.

@sync()
Wait until all dynamically-enclosed uses of `@async`, `@spawn`, `@spawnat` and `@parallel` are complete.

@parallel()
A parallel for loop of the form

```
@parallel [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, @parallel performs local reductions on each worker with a final reduction on the calling process.

Note that without a reducer function, @parallel 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 @parallel for var = range
    body
end
```

Distributed Arrays

DArray(*init*, *dims*[, *procs*, *dist*])
Construct a distributed array. The parameter `init` is a function that accepts a tuple of index ranges. This function should allocate a local chunk of the distributed array and initialize it for the specified indices. `dims` is the overall size of the distributed array. `procs` optionally specifies a vector of process IDs to use. If unspecified, the array is distributed over all worker processes only. Typically, when runnning in distributed mode, i.e., `nprocs() > 1`, this would mean that no chunk of the distributed array exists on the process hosting the interactive julia prompt. `dist` is an integer vector specifying how many chunks the distributed array should be divided into in each dimension.

For example, the `dfill` function that creates a distributed array and fills it with a value `v` is implemented as:

```
dfill(v, args...) = DArray(I->fill(v, map(length,I)), args...)
```

dzeros(*dims*, ...)
Construct a distributed array of zeros. Trailing arguments are the same as those accepted by `DArray()`.

dones(*dims*, ...)
Construct a distributed array of ones. Trailing arguments are the same as those accepted by `DArray()`.

dfill(*x*, *dims*, ...)
Construct a distributed array filled with value `x`. Trailing arguments are the same as those accepted by `DArray()`.

drand(*dims*, ...)
Construct a distributed uniform random array. Trailing arguments are the same as those accepted by `DArray()`.

drandn(*dims*, ...)
Construct a distributed normal random array. Trailing arguments are the same as those accepted by `DArray()`.

distribute(*a*)
Convert a local array to distributed.

localpart(*d*)
Get the local piece of a distributed array. Returns an empty array if no local part exists on the calling process.

localindexes(*d*)
A tuple describing the indexes owned by the local process. Returns a tuple with empty ranges if no local part exists on the calling process.

procs(*d*)
Get the vector of processes storing pieces of `d`.

Shared Arrays (Experimental, UNIX-only feature)

SharedArray(*T::Type*, *dims::NTuple; init=false*, *pids=Int[]*)
Construct a SharedArray of a bitstype `T` and size `dims` across the processes specified by `pids` - all of which have to be on the same host.

If `pids` is left unspecified, the shared array will be mapped across all workers on the current host.

If an `init` function of the type `initfn(S::SharedArray)` is specified, it is called on all the participating workers.

procs(*S::SharedArray*)
Get the vector of processes that have mapped the shared array

sdata (*S::SharedArray*)
Returns the actual `Array` object backing `S`

indexpids (*S::SharedArray*)
Returns the index of the current worker into the `pids` vector, i.e., the list of workers mapping the SharedArray

System

run (*command*)
Run a command object, constructed with backticks. Throws an error if anything goes wrong, including the process exiting with a non-zero status.

spawn (*command*)
Run a command object asynchronously, returning the resulting `Process` object.

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(`cat test.txt` |> DevNull)`

success (*command*)
Run a command object, constructed with backticks, and tell whether it was successful (exited with a code of 0). An exception is raised if the process cannot be started.

process_running (*p::Process*)
Determine whether a process is currently running.

process_exited (*p::Process*)
Determine whether a process has exited.

kill (*p::Process*, *signum=SIGTERM*)
Send a signal to a process. The default is to terminate the process.

open (*command*, *mode::String="r"*, *stdio=DevNull*)
Start running `command` asynchronously, and return a tuple `(stream,process)`. If `mode` is `"r"`, then `stream` reads from the process's standard output and `stdio` optionally specifies the process's standard input stream. If `mode` is `"w"`, then `stream` writes to the process's standard input and `stdio` optionally specifies the process's standard output stream.

open (*f::Function*, *command*, *mode::String="r"*, *stdio=DevNull*)
Similar to `open(command, mode, stdio)`, but calls `f(stream)` on the resulting read or write stream, then closes the stream and waits for the process to complete. Returns the value returned by `f`.

readandwrite (*command*)
Starts running a command asynchronously, and returns a tuple (stdout,stdin,process) of the output stream and input stream of the process, and the process object itself.

ignorestatus (*command*)
Mark a command object so that running it will not throw an error if the result code is non-zero.

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.

setenv (*command*, *env; dir=working_dir*)
Set environment variables to use when running the given command. `env` is either a dictionary mapping strings to strings, or an array of strings of the form `"var=val"`.

The `dir` keyword argument can be used to specify a working directory for the command.

|> (*command*, *command*)

|>(*command*, *filename*)
|>(*filename*, *command*)
Redirect operator. Used for piping the output of a process into another (first form) or to redirect the standard output/input of a command to/from a file (second and third forms).

Examples:

- `run(`ls` |> `grep xyz`)`
- `run(`ls` |> "out.txt")`
- `run("out.txt" |> `grep xyz`)`

>>(*command*, *filename*)
Redirect standard output of a process, appending to the destination file.

.>(*command*, *filename*)
Redirect the standard error stream of a process.

gethostname() → String
Get the local machine's host name.

getipaddr() → String
Get the IP address of the local machine, as a string of the form "x.x.x.x".

pwd() → String
Get the current working directory.

cd(*dir::String*)
Set the current working directory.

cd(*f*[, *dir*])
Temporarily changes the current working directory (HOME if not specified) and applies function f before returning.

mkdir(*path*[, *mode*])
Make a new directory with name `path` and permissions `mode`. `mode` defaults to 0o777, modified by the current file creation mask.

mkpath(*path*[, *mode*])
Create all directories in the given `path`, with permissions `mode`. `mode` defaults to 0o777, modified by the current file creation mask.

symlink(*target*, *link*)
Creates a symbolic link to `target` with the name `link`.

注解: This function raises an error under operating systems that do not support soft symbolic links, such as Windows XP.

---

chmod(*path*, *mode*)
Change the permissions mode of `path` to `mode`. Only integer `mode`s (e.g. 0o777) are currently supported.

getpid() → Int32
Get julia's process ID.

time([*t::TmStruct*])
Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution. When passed a `TmStruct`, converts it to a number of seconds since the epoch.

time_ns()
Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

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.

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.

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`.

tic()
Set a timer to be read by the next call to `toc()` or `toq()`. The macro call `@time expr` can also be used to time evaluation.

toc()
Print and return the time elapsed since the last `tic()`.

toq()
Return, but do not print, the time elapsed since the last `tic()`.

@time()
A macro to execute an expression, printing the time it took to execute and the total number of bytes its execution caused to be allocated, before returning the value of the expression.

@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.

@allocated()
A macro to evaluate an expression, discarding the resulting value, instead returning the total number of bytes allocated during evaluation of the expression.

EnvHash() → EnvHash
A singleton of this type provides a hash table interface to environment variables.

ENV
Reference to the singleton `EnvHash`, providing a dictionary interface to system environment variables.

@unix()
Given `@unix? a : b`, do `a` on Unix systems (including Linux and OS X) and `b` elsewhere. See documentation for Handling Platform Variations in the Calling C and Fortran Code section of the manual.

@osx()
Given `@osx? a : b`, do `a` on OS X and `b` elsewhere. See documentation for Handling Platform Variations in the Calling C and Fortran Code section of the manual.

@linux()
Given `@linux? a : b`, do `a` on Linux and `b` elsewhere. See documentation for Handling Platform Variations in the Calling C and Fortran Code section of the manual.

@windows()
Given `@windows? a : b`, do `a` on Windows and `b` elsewhere. See documentation for Handling Platform Variations in the Calling C and Fortran Code section of the manual.

C Interface

ccall (*(symbol, library) or fptr, RetType, (ArgType1, ...), ArgVar1, ...*)
Call function in C-exported shared library, specified by `(function name, library)` tuple, where each component is a String or :Symbol. Alternatively, ccall may be used to call a function pointer returned by dlsym, but note that this usage is generally discouraged to facilitate future static compilation. Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

cglobal (*(symbol, library) or ptr*[, *Type=Void*])
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{Void}` if no Type argument is supplied. The values can be read or written by `unsafe_load` or `unsafe_store!`, respectively.

cfunction (*fun::Function, RetType::Type, (ArgTypes...)*)
Generate C-callable function pointer from Julia function. Type annotation of the return value in the callback function is a must for situations where Julia cannot infer the return type automatically.

For example:

```
function foo()
    # body

    retval::Float64
end

bar = cfunction(foo, Float64, ())
```

dlopen (*libfile::String*[, *flags::Integer*])
Load a shared library, returning an opaque handle.

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 is `RTLD_LAZY|RTLD_DEEPBIND|RTLD_LOCAL`. An important usage of these flags, on POSIX platforms, is to specify `RTLD_LAZY|RTLD_DEEPBIND|RTLD_GLOBAL` in order for the library's symbols to be available for usage in other shared libraries, in situations where there are dependencies between shared libraries.

dlopen_e (*libfile::String*[, *flags::Integer*])
Similar to `dlopen()`, except returns a `NULL` pointer instead of raising errors.

RTLD_DEEPBIND
Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_FIRST
Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_GLOBAL
Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_LAZY
Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_LOCAL
Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_NODELETE
Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_NOLOAD
Enum constant for `dlopen()`. See your platform man page for details, if applicable.

RTLD_NOW
Enum constant for `dlopen()`. See your platform man page for details, if applicable.

dlsym(*handle*, *sym*)
Look up a symbol from a shared library handle, return callable function pointer on success.

dlsym_e(*handle*, *sym*)
Look up a symbol from a shared library handle, silently return NULL pointer on lookup failure.

dlclose(*handle*)
Close shared library referenced by handle.

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.

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.

c_malloc(*size::Integer*) → Ptr{Void}
Call `malloc` from the C standard library.

c_calloc(*num::Integer*, *size::Integer*) → Ptr{Void}
Call `calloc` from the C standard library.

c_realloc(*addr::Ptr*, *size::Integer*) → Ptr{Void}
Call `realloc` from the C standard library.

c_free(*addr::Ptr*)
Call `free` from the C standard library.

unsafe_load(*p::Ptr{T}*, *i::Integer*)
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]`.

unsafe_store!(*p::Ptr{T}*, *x*, *i::Integer*)
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`.

unsafe_copy!(*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.

unsafe_copy!(*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).

copy!(*dest*, *src*)
Copy all elements from collection `src` to array `dest`. Returns `dest`.

copy!(*dest*, *do*, *src*, *so*, *N*)
Copy `N` elements from collection `src` starting at offset `so`, to array `dest` starting at offset `do`. Returns `dest`.

pointer(*a*[, *index*])
Get the native address of an array or string element. Be careful to ensure that a julia reference to `a` exists as long as this pointer will be used.

pointer(*type*, *int*)
Convert an integer to a pointer of the specified element type.

pointer_to_array(*p*, *dims*[, *own*])
Wrap a native pointer as a Julia Array object. The pointer element type determines the array element type. `own` optionally specifies whether Julia should take ownership of the memory, calling `free` on the pointer when the array is no longer referenced.

pointer_from_objref(*obj*)
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.

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.

disable_sigint(*f::Function*)
Disable Ctrl-C handler during execution of a function, for calling external code that is not interrupt safe. Intended to be called using `do` block syntax as follows:

```
disable_sigint() do
    # interrupt-unsafe code
    ...
end
```

reenable_sigint(*f::Function*)
Re-enable Ctrl-C handler during execution of a function. Temporarily reverses the effect of `disable_sigint`.

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.

systemerror(*sysfunc*, *iftrue*)
Raises a `SystemError` for `errno` with the descriptive string `sysfunc` if `bool` is true

strerror(*n*)
Convert a system call error code to a descriptive string

Cchar
Equivalent to the native `char` c-type

Cuchar
Equivalent to the native `unsigned char` c-type (Uint8)

Cshort
Equivalent to the native `signed short` c-type (Int16)

Cushort
Equivalent to the native `unsigned short` c-type (Uint16)

Cint
Equivalent to the native `signed int` c-type (Int32)

Cuint
Equivalent to the native `unsigned int` c-type (Uint32)

Clong
Equivalent to the native `signed long` c-type

Culong
Equivalent to the native `unsigned long` c-type

Clonglong
Equivalent to the native `signed long long` c-type (Int64)

Culonglong
Equivalent to the native `unsigned long long` c-type (Uint64)

Csize_t
Equivalent to the native `size_t` c-type (Uint)

Cssize_t
Equivalent to the native `ssize_t` c-type

Cptrdiff_t
Equivalent to the native `ptrdiff_t` c-type (Int)

Coff_t
Equivalent to the native `off_t` c-type

Cwchar_t
Equivalent to the native `wchar_t` c-type (Int32)

Cfloat
Equivalent to the native `float` c-type (Float32)

Cdouble
Equivalent to the native `double` c-type (Float64)

Errors

error(*message::String*)
Raise an error with the given message

throw(*e*)
Throw an object as an exception

rethrow([*e*])
Throw an object without changing the current exception backtrace. The default argument is the current exception (if called within a `catch` block).

backtrace()
Get a backtrace object for the current program point.

catch_backtrace()
Get the backtrace of the current exception, for use within `catch` blocks.

assert(*cond*[, *text*])
Raise an error if `cond` is false. Also available as the macro `@assert expr`.

@assert()
Raise an error if `cond` is false. Preferred syntax for writings assertions.

ArgumentError
The parameters given to a function call are not valid.

BoundsError
An indexing operation into an array tried to access an out-of-bounds element.

EOFError
No more data was available to read from a file or stream.

ErrorException
Generic error type. The error message, in the *.msg* field, may provide more specific details.

KeyError
An indexing operation into an `Associative` (`Dict`) or `Set` like object tried to access or delete a non-existent element.

LoadError
An error occurred while *including*, *requiring*, or *using* a file. The error specifics should be available in the *.error* field.

MethodError
A method with the required type signature does not exist in the given generic function.

ParseError
The expression passed to the *parse* function could not be interpreted as a valid Julia expression.

ProcessExitedException
After a client Julia process has exited, further attempts to reference the dead child will throw this exception.

SystemError
A system call failed with an error code (in the `errno` global variable).

TypeError
A type assertion failure, or calling an intrinsic function with an incorrect argument type.

Tasks

Task (*func*)
Create a `Task` (i.e. thread, or coroutine) to execute the given function (which must be callable with no arguments). The task exits when this function returns.

yieldto (*task*, *args...*)
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, `args` are 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.

current_task ()
Get the currently running Task.

istaskdone (*task*) → Bool
Tell whether a task has exited.

consume (*task*, *values...*)
Receive the next value passed to `produce` by the specified task. Additional arguments may be passed, to be returned from the last `produce` call in the producer.

produce (*value*)
Send the given value to the last `consume` call, switching to the consumer task. If the next `consume` call passes any values, they are returned by `produce`.

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.

task_local_storage(*symbol*)
Look up the value of a symbol in the current task's task-local storage.

task_local_storage(*symbol*, *value*)
Assign a value to a symbol in the current task's task-local storage.

task_local_storage(*body*, *symbol*, *value*)
Call the function `body` with a modified task-local storage, in which `value` is assigned to `symbol`; the previous value of `symbol`, or lack thereof, is restored afterwards. Useful for emulating dynamic scoping.

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 `RemoteRef` type does this, and so can be used for level-triggered events.

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.

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 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.

@schedule()
Wrap an expression in a Task and add it to the scheduler's queue.

@task()
Wrap an expression in a Task executing it, and return the Task. This only creates a task, and does not run it.

sleep(*seconds*)
Block the current task for a specified number of seconds. The minimum sleep time is 1 millisecond or input of `0.001`.

Events

Timer(*f::Function*)
Create a timer to call the given callback function. The callback is passed one argument, the timer object itself. The timer can be started and stopped with `start_timer` and `stop_timer`.

start_timer(*t::Timer*, *delay*, *repeat*)
Start invoking the callback for a `Timer` after the specified initial delay, and then repeating with the given interval. Times are in seconds. If `repeat` is `0`, the timer is only triggered once.

stop_timer(*t::Timer*)
Stop invoking the callback for a timer.

Reflection

module_name(*m::Module*) → Symbol
Get the name of a module as a symbol.

module_parent(*m::Module*) → Module
Get a module's enclosing module. `Main` is its own parent.

current_module() → Module
Get the *dynamically* current module, which is the module code is currently being read from. In general, this is not the same as the module containing the call to this function.

fullname(*m::Module*)
Get the fully-qualified name of a module as a tuple of symbols. For example, `fullname(Base.Pkg)` gives `(:Base,:Pkg)`, and `fullname(Main)` gives `()`.

names(*x::Module*[, *all=false*[, *imported=false*]])
Get an array of the names exported by a module, with optionally more module globals according to the additional parameters.

names(*x::DataType*)
Get an array of the fields of a data type.

isconst([*m::Module*], *s::Symbol*) → Bool
Determine whether a global is declared `const` in a given module. The default module argument is `current_module()`.

isgeneric(*f::Function*) → Bool
Determine whether a function is generic.

function_name(*f::Function*) → Symbol
Get the name of a generic function as a symbol, or `:anonymous`.

function_module(*f::Function*, *types*) → Module
Determine the module containing a given definition of a generic function.

functionloc(*f::Function*, *types*)
Returns a tuple `(filename,line)` giving the location of a method definition.

functionlocs(*f::Function*, *types*)
Returns an array of the results of `functionloc` for all matching definitions.

Internals

gc()
Perform garbage collection. This should not generally be used.

gc_disable()
Disable garbage collection. This should be used only with extreme caution, as it can cause memory use to grow without bound.

gc_enable()
Re-enable garbage collection after calling `gc_disable`.

macroexpand(*x*)
Takes the expression x and returns an equivalent expression with all macros removed (expanded).

expand(*x*)
Takes the expression x and returns an equivalent expression in lowered form

code_lowered(*f*, *types*)
Returns an array of lowered ASTs for the methods matching the given generic function and type signature.

@code_lowered()
Evaluates the arguments to the function call, determines their types, and calls the `code_lowered` function on the resulting expression

code_typed(*f*, *types*)
Returns an array of lowered and type-inferred ASTs for the methods matching the given generic function and type signature.

@code_typed()
Evaluates the arguments to the function call, determines their types, and calls the `code_typed` function on the resulting expression

code_llvm(*f*, *types*)
Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to STDOUT.

@code_llvm()
Evaluates the arguments to the function call, determines their types, and calls the `code_llvm` function on the resulting expression

code_native(*f*, *types*)
Prints the native assembly instructions generated for running the method matching the given generic function and type signature to STDOUT.

@code_native()
Evaluates the arguments to the function call, determines their types, and calls the `code_native` function on the resulting expression

precompile(*f*, *args::(Any...,)*)
Compile the given function `f` for the argument tuple (of types) `args`, but do not execute it.

CHAPTER 34

Sparse Matrices

Sparse matrices support much of the same set of operations as dense matrices. The following functions are specific to sparse matrices.

`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 `max(I)` and `max(J)` respectively. If the `combine` function is not supplied, duplicates are added by default.

`sparsevec`(*I*, *V*[, *m*, *combine*])
: Create a sparse matrix `S` of size `m x 1` such that `S[I[k]] = V[k]`. Duplicates are combined using the `combine` function, which defaults to + if it is not provided. In julia, sparse vectors are really just sparse matrices with one column. Given Julia's Compressed Sparse Columns (CSC) storage format, a sparse column matrix with one column is sparse, whereas a sparse row matrix with one row ends up being dense.

`sparsevec`(*D::Dict*[, *m*])
: Create a sparse matrix of size `m x 1` where the row values are keys from the dictionary, and the nonzero values are the values from the dictionary.

`issparse`(*S*)
: Returns `true` if `S` is sparse, and `false` otherwise.

`sparse`(*A*)
: Convert a dense matrix `A` into a sparse matrix.

`sparsevec`(*A*)
: Convert a dense vector `A` into a sparse matrix of size `m x 1`. In julia, sparse vectors are really just sparse matrices with one column.

`full`(*S*)
: Convert a sparse matrix `S` into a dense matrix.

`nnz`(*A*)
: Returns the number of stored (filled) elements in a sparse matrix.

`spzeros`(*m*, *n*)
: Create an empty sparse matrix of size `m x n`.

spones(*S*)

Create a sparse matrix with the same structure as that of `S`, but with every nonzero element having the value `1.0`.

speye(*type*, *m*[, *n*])

Create a sparse identity matrix of specified type of size `m x m`. In case `n` is supplied, create a sparse identity matrix of size `m x n`.

spdiagm(*B*, *d*[, *m*, *n*])

Construct a sparse diagonal matrix. `B` is a tuple of vectors containing the diagonals and `d` is a tuple containing the positions of the diagonals. In the case the input contains only one diagonaly, `B` can be a vector (instead of a tuple) and `d` can be the diagonal position (instead of a tuple), defaulting to 0 (diagonal). Optionally, `m` and `n` specify the size of the resulting sparse matrix.

sprand(*m*, *n*, *p*[, *rng*])

Create a random `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 `rng`. The uniform distribution is used in case `rng` is not specified.

sprandn(*m*, *n*, *p*)

Create a random `m` by `n` sparse matrix with the specified (independent) probability `p` of any entry being nonzero, where nonzero values are sampled from the normal distribution.

sprandbool(*m*, *n*, *p*)

Create a random `m` by `n` sparse boolean matrix with the specified (independent) probability `p` of any entry being `true`.

etree(*A*[, *post*])

Compute the elimination tree of a symmetric sparse matrix `A` from `triu(A)` and, optionally, its post-ordering permutation.

symperm(*A*, *p*)

Return the symmetric permutation of A, which is `A[p,p]`. A should be symmetric and sparse, where only the upper triangular part of the matrix is stored. This algorithm ignores the lower triangular part of the matrix. Only the upper triangular part of the result is returned as well.

nonzeros(*A*)

Return a vector of the structural nonzero values in sparse matrix `A`. This includes zeros that are explicitly stored in the sparse matrix. The returned vector points directly to the internal nonzero storage of `A`, and any modifications to the returned vector will mutate `A` as well.

CHAPTER 35

Linear Algebra

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK. Sparse factorizations call functions from SuiteSparse.

***** (*A*, *B*)
Matrix multiplication

**** (*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`. A direct solver is used for upper- or lower triangular `A`. For Hermitian `A` (equivalent to symmetric `A` for non-complex `A`) the `BunchKaufman` factorization is used. Otherwise an LU factorization is used. For rectangular `A` the result is the minimum-norm least squares solution computed by reducing `A` to bidiagonal form and solving the bidiagonal least squares problem. For sparse, square `A` the LU factorization (from UMFPACK) is used.

dot (*x*, *y*)
Compute the dot product. For complex vectors, the first vector is conjugated.

cross (*x*, *y*)
Compute the cross product of two 3-vectors.

rref (*A*)
Compute the reduced row echelon form of the matrix A.

factorize (*A*)
Compute a convenient factorization (including LU, Cholesky, Bunch-Kaufman, Triangular) of A, based upon the type of the input matrix. The return value can then be reused for efficient solving of multiple systems. For example: `A=factorize(A); x=A\\b; y=A\\C`.

factorize! (*A*)
`factorize!` is the same as `factorize()`, but saves space by overwriting the input `A`, instead of creating a copy.

lu (*A*) → L, U, p
Compute the LU factorization of `A`, such that `A[p,:] = L*U`.

lufact (*A*[, *pivot=true*]) → F
Compute the LU factorization of `A`. The return type of `F` depends on the type of `A`. In most cases, if `A` is

a subtype S of AbstractMatrix 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 <. When A is sparse and have element of type Float32, Float64, Complex{Float32}, or Complex{Float64} the return type is UmfpackLU. Some examples are shown in the table below.

Type of input A	Type of output F	Relationship between F and A
Matrix()	LU	F[:L]*F[:U] == A[F[:p], :]
Tridiagonal()	LU{T, Tridiagonal{T}}	N/A
SparseMatrixCSC()	UmfpackLU	F[:L]*F[:U] == Rs .* A[F[:p], F[:q]]

The individual components of the factorization F can be accessed by indexing:

Component	Description	LU	LU{T, Tridiagonal{T}}	UmfpackLU
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	✓		
F[:q]	left permutation Vector			✓
F[:Rs]	Vector of scaling factors			✓
F[:(:)]	(L,U,p,q,Rs) components			✓

Supported function	LU	LU{T,Tridiagonal{T}}	UmfpackLU
/	✓		
\	✓	✓	✓
cond	✓		✓
det	✓	✓	✓
size	✓	✓	

lufact!(*A*) → LU

lufact! is the same as lufact(), but saves space by overwriting the input A, instead of creating a copy. For sparse A the nzval field is not overwritten but the index fields, colptr and rowval are decremented in place, converting from 1-based indices to 0-based indices.

chol(*A*[, *LU*]) → F

Compute the Cholesky factorization of a symmetric positive definite matrix A and return the matrix F. If LU is :L (Lower), A = L*L'. If LU is :U (Upper), A = R'*R.

cholfact(*A, [LU,][pivot=false,][tol=-1.0]*) → Cholesky

Compute the Cholesky factorization of a dense symmetric positive (semi)definite matrix A and return either a Cholesky if pivot=false or CholeskyPivoted if pivot=true. LU may be :L for using the lower part or :U for the upper part. The default is to use :U. The triangular matrix can be obtained from the factorization F with: F[:L] and F[:U]. The following functions are available for Cholesky objects: size, \, inv, det. For CholeskyPivoted there is also defined a rank. If pivot=false a PosDefException exception is thrown in case the matrix is not positive definite. The argument tol determines the tolerance for determining the rank. For negative values, the tolerance is the machine precision.

cholfact(*A*[, *ll*]) → CholmodFactor

Compute the sparse Cholesky factorization of a sparse matrix A. If A is Hermitian its Cholesky factor is determined. If A is not Hermitian the Cholesky factor of A*A' is determined. A fill-reducing permutation is used. Methods for size, solve, \, findn_nzs, diag, det and logdet. One of the solve methods includes an integer argument that can be used to solve systems involving parts of the factorization only. The optional

boolean argument, `ll` determines whether the factorization returned is of the `A[p,p] = L*L'` form, where `L` is lower triangular or `A[p,p] = L*Diagonal(D)*L'` form where `L` is unit lower triangular and `D` is a non-negative vector. The default is LDL.

`cholfact!` (*A, [LU,][pivot=false,][tol=-1.0]*) → Cholesky

`cholfact!` is the same as `cholfact()`, but saves space by overwriting the input `A`, instead of creating a copy.

`ldltfact` (*A*) → LDLtFactorization

Compute a factorization of a positive definite matrix `A` such that `A=L*Diagonal(d)*L'` where `L` is a unit lower triangular matrix and `d` is a vector with non-negative elements.

`qr` (*A, [pivot=false,][thin=true]*) → Q, R, [p]

Compute the (pivoted) QR factorization of `A` such that either `A = Q*R` or `A[:,p] = Q*R`. Also see `qrfact`. The default is to compute a thin factorization. Note that `R` is not extended with zeros when the full `Q` is requested.

`qrfact` (*A*[, *pivot=false*]) → F

Computes the QR factorization of `A`. The return type of `F` depends on the element type of `A` and whether pivoting is specified (with `pivot=true`).

Return type	`eltype(A)`	`pivot`	Relationship between `F` and `A`
`QR`	not `BlasFloat`	either	`A==F[:Q]*F[:R]`
`QRCompactWY`	`BlasFloat`	`false`	`A==F[:Q]*F[:R]`
`QRPivoted`	`BlasFloat`	`true`	`A[:,F[:p]]==F[:Q]*F[:R]`

`BlasFloat` refers to any of: `Float32`, `Float64`, `Complex64` or `Complex128`.

The individual components of the factorization `F` can be accessed by indexing:

Component	Description	`QR`	`QRCompactWY`	`QRPivoted`
`F[:Q]`	`Q` (orthogonal/unitary) part of `QR`	✓ (`QRPackedQ`)	✓ (`QRCompactWYQ`)	✓ (`QRPackedQ`)
`F[:R]`	`R` (upper right triangular) part of `QR`	✓	✓	✓
`F[:p]`	pivot `Vector`			✓
`F[:P]`	(pivot) permutation `Matrix`			✓

The following functions are available for the `QR` objects: `size`, `\`. When `A` is rectangular, `\` will return a least squares solution and if the solution is not unique, the one with smallest norm is returned.

Multiplication with respect to either thin or full `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 `full()` which has a named argument `thin`.

注解: `qrfact` 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.

The data contained in `QR` or `QRPivoted` can be used to construct the `QRPackedQ` type, which is a compact representation of the rotation matrix:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T)$$

where τ_i is the scale factor and v_i is the projection vector associated with the i^{th} Householder elementary reflector.

The data contained in `QRCompactWY` can be used to construct the `QRCompactWYQ` type, which is a compact representation of the rotation matrix

$$Q = I + YTY^T$$

where `Y` is $m \times r$ lower trapezoidal and `T` is $r \times r$ upper triangular. The *compact WY* representation *[Schreiber1989]* is not to be confused with the older, *WY* representation *[Bischof1987]*. (The LAPACK documentation uses `V` in lieu of `Y`.)

---

qrfact! (*A*[, *pivot=false*])
`qrfact!` is the same as `qrfact()`, but saves space by overwriting the input `A`, instead of creating a copy.

bkfact (*A*) → BunchKaufman
Compute the Bunch-Kaufman *[Bunch1977]* factorization of a real symmetric or complex Hermitian matrix `A` and return a `BunchKaufman` object. The following functions are available for `BunchKaufman` objects: `size`, `\`, `inv`, `issym`, `ishermitian`.

bkfact! (*A*) → BunchKaufman
`bkfact!` is the same as `bkfact()`, but saves space by overwriting the input `A`, instead of creating a copy.

sqrtm (*A*)
Compute the matrix square root of `A`. If `B = sqrtm(A)`, then `B*B == A` within roundoff error.

`sqrtm` uses a polyalgorithm, computing the matrix square root using Schur factorizations (`schurfact()`) unless it detects the matrix to be Hermitian or real symmetric, in which case it computes the matrix square root from an eigendecomposition (`eigfact()`). In the latter situation for positive definite matrices, the matrix square root has `Real` elements, otherwise it has `Complex` elements.

eig (*A,[irange,][vl,][vu,][permute=true,][scale=true]*) → D, V
Compute eigenvalues and eigenvectors of `A`. See `eigfact()` for details on the `balance` keyword argument.

```
julia> eig([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
([1.0,3.0,18.0],
3x3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0)
```

`eig` is a wrapper around `eigfact()`, extracting all parts of the factorization to a tuple; where possible, using `eigfact()` is recommended.

eig (*A*, *B*) → D, V
Computes generalized eigenvalues and vectors of `A` with respect to `B`.

`eig` is a wrapper around `eigfact()`, extracting all parts of the factorization to a tuple; where possible, using `eigfact()` is recommended.

eigvals (*A,[irange,][vl,][vu]*)
Returns the eigenvalues of `A`. If `A` is `Symmetric()`, `Hermitian()` or `SymTridiagonal()`, it is possible to calculate only a subset of the eigenvalues by specifying either a `UnitRange()` `irange` covering indices of the sorted eigenvalues, or a pair `vl` and `vu` for the lower and upper boundaries of the eigenvalues.

For general non-symmetric 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.

eigmax(*A*)
Returns the largest eigenvalue of `A`.

eigmin(*A*)
Returns the smallest eigenvalue of `A`.

eigvecs(*A, [eigvals,][permute=true,][scale=true]*)
Returns the eigenvectors of `A`. The `permute` and `scale` keywords are the same as for `eigfact()`.

For `SymTridiagonal()` matrices, if the optional vector of eigenvalues `eigvals` is specified, returns the specific corresponding eigenvectors.

eigfact(*A,[il,][iu,][vl,][vu,][permute=true,][scale=true]*)
Compute the eigenvalue decomposition of `A` and return an `Eigen` object. If `F` is the factorization object, the eigenvalues can be accessed with `F[:values]` and the eigenvectors with `F[:vectors]`. The following functions are available for `Eigen` objects: `inv`, `det`.

If `A` is `Symmetric`, `Hermitian` or `SymTridiagonal`, it is possible to calculate only a subset of the eigenvalues by specifying either a *UnitRange*' `irange` covering indices of the sorted eigenvalues or a pair `vl` and `vu` for the lower and upper boundaries of the eigenvalues.

For general non-symmetric 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.

eigfact(*A*, *B*)
Compute the generalized eigenvalue decomposition of `A` and `B` and return an `GeneralizedEigen` object. If `F` is the factorization object, the eigenvalues can be accessed with `F[:values]` and the eigenvectors with `F[:vectors]`.

eigfact!(*A*[, *B*])
`eigfact!` is the same as `eigfact()`, but saves space by overwriting the input A (and B), instead of creating a copy.

hessfact(*A*)
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]` 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 `full()`.

hessfact!(*A*)
`hessfact!` is the same as `hessfact()`, but saves space by overwriting the input A, instead of creating a copy.

schurfact(*A*) → 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 unitary/orthogonal 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]`.

schurfact!(*A*)
Computer the Schur factorization of `A`, overwriting `A` in the process. See `schurfact()`

schur(*A*) → Schur[:T], Schur[:Z], Schur[:values]
See `schurfact()`

schurfact(*A*, *B*) → 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[:alpha]./F[:beta]`.

schur(*A*, *B*) → GeneralizedSchur[:S], GeneralizedSchur[:T], GeneralizedSchur[:Q], GeneralizedSchur[:Z]
See `schurfact()`

svdfact(*A*[, *thin*=*true*]) → 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*diagm(S)*Vt`. If `thin` is `true`, an economy mode decomposition is returned. The algorithm produces `Vt` and hence `Vt` is more efficient to extract than `V`. The default is to produce a thin decomposition.

svdfact!(*A*[, *thin*=*true*]) → SVD
`svdfact!` is the same as `svdfact()`, but saves space by overwriting the input A, instead of creating a copy. If `thin` is `true`, an economy mode decomposition is returned. The default is to produce a thin decomposition.

svd(*A*[, *thin*=*true*]) → U, S, V
Wrapper around `svdfact` extracting all parts the factorization to a tuple. Direct use of `svdfact` is therefore generally more efficient. Computes the SVD of A, returning `U`, vector `S`, and `V` such that `A == U*diagm(S)*V'`. If `thin` is `true`, an economy mode decomposition is returned. The default is to produce a thin decomposition.

svdvals(*A*)
Returns the singular values of `A`.

svdvals!(*A*)
Returns the singular values of `A`, while saving space by overwriting the input.

svdfact(*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]'`.

svd(*A*, *B*) → U, V, Q, D1, D2, R0
Wrapper around `svdfact` extracting all parts the factorization to a tuple. Direct use of `svdfact` is therefore generally more efficient. The function returns the generalized SVD of `A` and `B`, returning `U`, `V`, `Q`, `D1`, `D2`, and `R0` such that `A = U*D1*R0*Q'` and `B = V*D2*R0*Q'`.

svdvals(*A*, *B*)
Return only the singular values from the generalized singular value decomposition of `A` and `B`.

triu(*M*)
Upper triangle of a matrix.

triu!(*M*)
Upper triangle of a matrix, overwriting `M` in the process.

tril(*M*)
Lower triangle of a matrix.

tril!(*M*)
Lower triangle of a matrix, overwriting `M` in the process.

diagind(*M*[, *k*])
A `Range` giving the indices of the `k`-th diagonal of the matrix `M`.

diag(*M*[, *k*])
The `k`-th diagonal of a matrix, as a vector. Use `diagm` to construct a diagonal matrix.

diagm(*v*[, *k*])
Construct a diagonal matrix and place `v` on the `k`-th diagonal.

scale(*A*, *b*)

scale(*b*, *A*)
Scale an array `A` by a scalar `b`, returning a new array.

If `A` is a matrix and `b` is a vector, then `scale(A,b)` scales each column `i` of `A` by `b[i]` (similar to `A*diagm(b)`), while `scale(b,A)` scales each row `i` of `A` by `b[i]` (similar to `diagm(b)*A`), returning a new array.

Note: for large `A`, `scale` can be much faster than `A .* b` or `b .* A`, due to the use of BLAS.

scale!(*A*, *b*)

scale!(*b*, *A*)
Scale an array `A` by a scalar `b`, similar to `scale()` but overwriting `A` in-place.

If `A` is a matrix and `b` is a vector, then `scale!(A,b)` scales each column `i` of `A` by `b[i]` (similar to `A*diagm(b)`), while `scale!(b,A)` scales each row `i` of `A` by `b[i]` (similar to `diagm(b)*A`), again operating in-place on `A`.

Tridiagonal(*dl*, *d*, *du*)
Construct a tridiagonal matrix from the lower diagonal, diagonal, and upper diagonal, respectively. The result is of type `Tridiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `full()`.

Bidiagonal(*dv*, *ev*, *isupper*)
Constructs an upper (`isupper=true`) or lower (`isupper=false`) 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 `full()`.

SymTridiagonal(*d*, *du*)
Construct a real symmetric tridiagonal matrix from the diagonal and upper diagonal, respectively. The result is of type `SymTridiagonal` and provides efficient specialized eigensolvers, but may be converted into a regular matrix with `full()`.

Woodbury(*A*, *U*, *C*, *V*)
Construct a matrix in a form suitable for applying the Woodbury matrix identity.

rank(*M*)
Compute the rank of a matrix.

norm(*A*[, *p*])
Compute the `p`-norm of a vector or the operator norm of a matrix `A`, defaulting to the `p=2`-norm.

For vectors, `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.

For matrices, valid values of `p` are `1`, `2`, or `Inf`. (Note that for sparse matrices, `p=2` is currently not implemented.) Use `vecnorm()` to compute the Frobenius norm.

vecnorm(*A*[, *p*])
For any iterable container `A` (including arrays of any dimension) of numbers, compute the `p`-norm (defaulting to `p=2`) as if `A` were a vector of the corresponding length.

For example, if `A` is a matrix and `p=2`, then this is equivalent to the Frobenius norm.

cond(*M*[, *p*])
Condition number of the matrix `M`, computed using the operator `p`-norm. Valid values for `p` are `1`, `2` (default), or `Inf`.

condskeel(*M*[, *x*, *p*])

$$\kappa_S(M,p) = \left\| |M| \left|M^{-1}\right| \right\|_p$$
$$\kappa_S(M,x,p) = \left\| |M| \left|M^{-1}\right| |x| \right\|_p$$

Skeel condition number κ_S of the matrix `M`, optionally with respect to the vector `x`, as computed using the operator `p`-norm. `p` is `Inf` by default, if not provided. Valid values for `p` are `1`, `2`, or `Inf`.

This quantity is also known in the literature as the Bauer condition number, relative condition number, or componentwise relative condition number.

trace(*M*)
Matrix trace

det(*M*)
Matrix determinant

logdet(*M*)
Log of matrix determinant. Equivalent to `log(det(M))`, but may provide increased accuracy and/or speed.

inv(*M*)
Matrix inverse

pinv(*M*)
Moore-Penrose pseudoinverse

null(*M*)
Basis for nullspace of `M`.

repmat(*A*, *n*, *m*)
Construct a matrix by repeating the given matrix `n` times in dimension 1 and `m` times in dimension 2.

repeat(*A*, *inner = Int[]*, *outer = Int[]*)
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.

kron(*A*, *B*)
Kronecker tensor product of two vectors or two matrices.

blkdiag(*A...*)
Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

linreg(*x*, *y*) → [a; b]
Linear Regression. Returns `a` and `b` such that `a+b*x` is the closest line to the given points `(x,y)`. In other words, this function determines parameters `[a, b]` that minimize the squared error between `y` and `a+b*x`.

Example:

```
using PyPlot;
x = float([1:12])
y = [5.5; 6.3; 7.6; 8.8; 10.9; 11.79; 13.48; 15.02; 17.77; 20.81; 22.0; 22.99]
a, b = linreg(x,y) # Linear regression
plot(x, y, "o") # Plot (x,y) points
plot(x, [a+b*i for i in x]) # Plot the line determined by the linear regression
```

linreg(*x*, *y*, *w*)
Weighted least-squares linear regression.

expm(*A*)
Matrix exponential.

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.

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.

issym(*A*) → Bool
Test whether a matrix is symmetric.

isposdef(*A*) → Bool
Test whether a matrix is positive definite.

isposdef!(*A*) → Bool
Test whether a matrix is positive definite, overwriting `A` in the processes.

istril(*A*) → Bool
Test whether a matrix is lower triangular.

istriu(*A*) → Bool
Test whether a matrix is upper triangular.

ishermitian(*A*) → Bool
Test whether a matrix is Hermitian.

transpose(*A*)
The transposition operator (`.'`).

ctranspose(*A*)
The conjugate transposition operator (`'`).

eigs(*A*[, *B*], *; nev=6*, *which="LM"*, *tol=0.0*, *maxiter=1000*, *sigma=nothing*, *ritzvec=true*, *v0=zeros((0,))*)
-> (*d*[, *v*], *nconv*, *niter*, *nmult*, *resid*)

eigs computes eigenvalues d of A using Lanczos or Arnoldi iterations for real symmetric or general nonsymmetric matri

- `nev`: Number of eigenvalues
- `ncv`: Number of Krylov vectors used in the computation; should satisfy `nev+1 <= ncv <= n` for real symmetric problems and `nev+2 <= ncv <= n` for other problems; default is `ncv = max(20,2*nev+1)`.
- `which`: type of eigenvalues to compute. See the note below.

`which`	type of eigenvalues
`:LM`	eigenvalues of largest magnitude (default)
`:SM`	eigenvalues of smallest magnitude
`:LR`	eigenvalues of largest real part
`:SR`	eigenvalues of smallest real part
`:LI`	eigenvalues of largest imaginary part (nonsymmetric or complex `A` only)
`:SI`	eigenvalues of smallest imaginary part (nonsymmetric or complex `A` only)
`:BE`	compute half of the eigenvalues from each end of the spectrum, biased in favor of the high end. (real symmetric `A` only)

- `tol`: tolerance ($tol \leq 0.0$ defaults to `DLAMCH('EPS')`)
- `maxiter`: Maximum number of iterations (default = 300)
- `sigma`: Specifies the level shift used in inverse iteration. If `nothing` (default), defaults to ordinary (forward) iterations. Otherwise, find eigenvalues close to `sigma` using shift and invert iterations.

- `ritzvec`: Returns the Ritz vectors `v` (eigenvectors) if `true`
- `v0`: starting vector from which to start the iterations

`eigs` returns the `nev` requested eigenvalues in `d`, the corresponding Ritz vectors `v` (only if `ritzvec=true`), the number of converged eigenvalues `nconv`, the number of iterations `niter` and the number of matrix vector multiplications `nmult`, as well as the final residual vector `resid`.

注解: The `sigma` and `which` keywords interact: the description of eigenvalues searched for by `which` do _not_ necessarily refer to the eigenvalues of `A`, but rather the linear operator constructed by the specification of the iteration mode implied by `sigma`.

`sigma`	iteration mode	`which` refers to eigenvalues of
`nothing`	ordinary (forward)	A
real or complex	inverse with level shift `sigma`	$(A - \sigma I)^{-1}$

`peakflops`(*n; parallel=false*)

`peakflops` computes the peak flop rate of the computer by using BLAS 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.

CHAPTER 36

BLAS Functions

This module provides wrappers for some of the BLAS functions for linear algebra. Those BLAS functions that overwrite one of the input arrays have names ending in `'!'`.

Usually a function has 4 methods defined, one each for `Float64`, `Float32`, `Complex128` and `Complex64` arrays.

`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`.

`dotu`(*n*, *X*, *incx*, *Y*, *incy*)
Dot function for two complex vectors.

`dotc`(*n*, *X*, *incx*, *U*, *incy*)
Dot function for two complex vectors conjugating the first vector.

`blascopy!`(*n*, *X*, *incx*, *Y*, *incy*)
Copy `n` elements of array `X` with stride `incx` to array `Y` with stride `incy`. Returns `Y`.

`nrm2`(*n*, *X*, *incx*)
2-norm of a vector consisting of `n` elements of array `X` with stride `incx`.

`asum`(*n*, *X*, *incx*)
sum of the absolute values of the first `n` elements of array `X` with stride `incx`.

`axpy!`(*n*, *a*, *X*, *incx*, *Y*, *incy*)
Overwrite `Y` with `a*X + Y`. Returns `Y`.

`scal!`(*n*, *a*, *X*, *incx*)
Overwrite `X` with `a*X`. Returns `X`.

`scal`(*n*, *a*, *X*, *incx*)
Returns `a*X`.

`syrk!`(*uplo*, *trans*, *alpha*, *A*, *beta*, *C*)
Rank-k update of the symmetric matrix `C` as `alpha*A*A.' + beta*C` or `alpha*A.'*A + beta*C` according to whether `trans` is 'N' or 'T'. When `uplo` is 'U' the upper triangle of `C` is updated ('L' for lower triangle). Returns `C`.

syrk(*uplo*, *trans*, *alpha*, *A*)

Returns either the upper triangle or the lower triangle, according to `uplo` ('U' or 'L'), of `alpha*A*A.'` or `alpha*A.'*A`, according to `trans` ('N' or 'T').

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 whether `trans` is 'N' or 'T'. When `uplo` is 'U' the upper triangle of `C` is updated ('L' for lower triangle). Returns `C`.

herk(*uplo*, *trans*, *alpha*, *A*)

Methods for complex arrays only. Returns either the upper triangle or the lower triangle, according to `uplo` ('U' or 'L'), of `alpha*A*A'` or `alpha*A'*A`, according to `trans` ('N' or 'T').

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` ('N' or 'T'). The matrix `A` is a general band matrix of dimension `m` by `size(A,2)` with `kl` sub-diagonals and `ku` super-diagonals. Returns the updated `y`.

gbmv(*trans*, *m*, *kl*, *ku*, *alpha*, *A*, *x*, *beta*, *y*)

Returns `alpha*A*x` or `alpha*A'*x` according to `trans` ('N' or 'T'). The matrix `A` is a general band matrix of dimension `m` by `size(A,2)` with `kl` sub-diagonals and `ku` super-diagonals.

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/.

Returns the updated `y`.

sbmv(*uplo*, *k*, *alpha*, *A*, *x*)

Returns `alpha*A*x` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`.

sbmv(*uplo*, *k*, *A*, *x*)

Returns `A*x` where `A` is a symmetric band matrix of order `size(A,2)` with `k` super-diagonals stored in the argument `A`.

gemm!(*tA*, *tB*, *alpha*, *A*, *B*, *beta*, *C*)

Update `C` as `alpha*A*B + beta*C` or the other three variants according to `tA` (transpose `A`) and `tB`. Returns the updated `C`.

gemm(*tA*, *tB*, *alpha*, *A*, *B*)

Returns `alpha*A*B` or the other three variants according to `tA` (transpose `A`) and `tB`.

gemm(*tA*, *tB*, *A*, *B*)

Returns `A*B` or the other three variants according to `tA` (transpose `A`) and `tB`.

gemv!(*tA*, *alpha*, *A*, *x*, *beta*, *y*)

Update the vector `y` as `alpha*A*x + beta*x` or `alpha*A'x + beta*x` according to `tA` (transpose `A`). Returns the updated `y`.

gemv(*tA*, *alpha*, *A*, *x*)

Returns `alpha*A*x` or `alpha*A'x` according to `tA` (transpose `A`).

gemv(*tA*, *A*, *x*)

Returns `A*x` or `A'x` according to `tA` (transpose `A`).

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. Returns the updated `C`.

symm(*side*, *ul*, *alpha*, *A*, *B*)
Returns `alpha*A*B` or `alpha*B*A` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

symm(*side*, *ul*, *A*, *B*)
Returns `A*B` or `B*A` according to `side`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

symm(*tA*, *tB*, *alpha*, *A*, *B*)
Returns `alpha*A*B` or the other three variants according to `tA` (transpose `A`) and `tB`.

symv!(*ul*, *alpha*, *A*, *x*, *beta*, *y*)
Update the vector `y` as `alpha*A*y + beta*y`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used. Returns the updated `y`.

symv(*ul*, *alpha*, *A*, *x*)
Returns `alpha*A*x`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

symv(*ul*, *A*, *x*)
Returns `A*x`. `A` is assumed to be symmetric. Only the `ul` triangle of `A` is used.

trmm!(*side*, *ul*, *tA*, *dA*, *alpha*, *A*, *B*)
Update `B` as `alpha*A*B` or one of the other three variants determined by `side` (A on left or right) and `tA` (transpose A). Only the `ul` triangle of `A` is used. `dA` indicates if `A` is unit-triangular (the diagonal is assumed to be all ones). Returns the updated `B`.

trmm(*side*, *ul*, *tA*, *dA*, *alpha*, *A*, *B*)
Returns `alpha*A*B` or one of the other three variants determined by `side` (A on left or right) and `tA` (transpose A). Only the `ul` triangle of `A` is used. `dA` indicates if `A` is unit-triangular (the diagonal is assumed to be all ones).

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` (A on left or right of `X`) and `tA` (transpose A). Only the `ul` triangle of `A` is used. `dA` indicates if `A` is unit-triangular (the diagonal is assumed to be all ones). Returns the updated `B`.

trsm(*side*, *ul*, *tA*, *dA*, *alpha*, *A*, *B*)
Returns the solution to `A*X = alpha*B` or one of the other three variants determined by `side` (A on left or right of `X`) and `tA` (transpose A). Only the `ul` triangle of `A` is used. `dA` indicates if `A` is unit-triangular (the diagonal is assumed to be all ones).

trmv!(*side*, *ul*, *tA*, *dA*, *alpha*, *A*, *b*)
Update `b` as `alpha*A*b` or one of the other three variants determined by `side` (A on left or right) and `tA` (transpose A). Only the `ul` triangle of `A` is used. `dA` indicates if `A` is unit-triangular (the diagonal is assumed to be all ones). Returns the updated `b`.

trmv(*side*, *ul*, *tA*, *dA*, *alpha*, *A*, *b*)
Returns `alpha*A*b` or one of the other three variants determined by `side` (A on left or right) and `tA` (transpose A). Only the `ul` triangle of `A` is used. `dA` indicates if `A` is unit-triangular (the diagonal is assumed to be all ones).

trsv!(*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` (A on left or right of `X`) and `tA` (transpose A). Only the `ul` triangle of `A` is used. `dA` indicates if `A` is unit-triangular (the diagonal is assumed to be all ones). Returns the updated `b`.

trsv(*side*, *ul*, *tA*, *dA*, *alpha*, *A*, *b*)
Returns the solution to `A*X = alpha*b` or one of the other three variants determined by `side` (A on left or right of `X`) and `tA` (transpose A). Only the `ul` triangle of `A` is used. `dA` indicates if `A` is unit-triangular (the diagonal is assumed to be all ones).

blas_set_num_threads(*n*)

Set the number of threads the BLAS library should use.

CHAPTER 37

Constants

OS_NAME
A symbol representing the name of the operating system. Possible values are `:Linux`, `:Darwin` (OS X), or `:Windows`.

ARGS
An array of the command line arguments passed to Julia, as strings.

C_NULL
The C null pointer constant, sometimes used when calling external code.

CPU_CORES
The number of CPU cores in the system.

WORD_SIZE
Standard word size on the current machine, in bits.

VERSION
An object describing which version of Julia is in use.

LOAD_PATH
An array of paths (as strings) where the `require` function looks for code.

CHAPTER 38

Filesystem

isblockdev(*path*) → Bool
Returns `true` if `path` is a block device, `false` otherwise.

ischardev(*path*) → Bool
Returns `true` if `path` is a character device, `false` otherwise.

isdir(*path*) → Bool
Returns `true` if `path` is a directory, `false` otherwise.

isexecutable(*path*) → Bool
Returns `true` if the current user has permission to execute `path`, `false` otherwise.

isfifo(*path*) → Bool
Returns `true` if `path` is a FIFO, `false` otherwise.

isfile(*path*) → Bool
Returns `true` if `path` is a regular file, `false` otherwise.

islink(*path*) → Bool
Returns `true` if `path` is a symbolic link, `false` otherwise.

ispath(*path*) → Bool
Returns `true` if `path` is a valid filesystem path, `false` otherwise.

isreadable(*path*) → Bool
Returns `true` if the current user has permission to read `path`, `false` otherwise.

issetgid(*path*) → Bool
Returns `true` if `path` has the setgid flag set, `false` otherwise.

issetuid(*path*) → Bool
Returns `true` if `path` has the setuid flag set, `false` otherwise.

issocket(*path*) → Bool
Returns `true` if `path` is a socket, `false` otherwise.

issticky(*path*) → Bool
Returns `true` if `path` has the sticky bit set, `false` otherwise.

iswritable(*path*) → Bool
Returns `true` if the current user has permission to write to `path`, `false` otherwise.

homedir() → String
Return the current user's home directory.

dirname(*path::String*) → String
Get the directory part of a path.

basename(*path::String*) → String
Get the file name part of a path.

@__FILE__() → String
`@__FILE__` expands to a string with the absolute path and file name of the script being run. Returns `nothing` if run from a REPL or an empty string if evaluated by `julia -e <expr>`.

isabspath(*path::String*) → Bool
Determines whether a path is absolute (begins at the root directory).

isdirpath(*path::String*) → Bool
Determines whether a path refers to a directory (for example, ends with a path separator).

joinpath(*parts...*) → String
Join path components into a full path. If some argument is an absolute path, then prior components are dropped.

abspath(*path::String*) → String
Convert a path to an absolute path by adding the current directory if necessary.

normpath(*path::String*) → String
Normalize a path, removing "." and ".." entries.

realpath(*path::String*) → String
Canonicalize a path by expanding symbolic links and removing "." and ".." entries.

expanduser(*path::String*) → String
On Unix systems, replace a tilde character at the start of a path with the current user's home directory.

splitdir(*path::String*) -> (*String*, *String*)
Split a path into a tuple of the directory name and file name.

splitdrive(*path::String*) -> (*String*, *String*)
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.

splitext(*path::String*) -> (*String*, *String*)
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.

tempname()
Generate a unique temporary filename.

tempdir()
Obtain the path of a temporary directory.

mktemp()
Returns `(path, io)`, where `path` is the path of a new temporary file and `io` is an open file object for this path.

mktempdir()
Create a temporary directory and return its path.

CHAPTER 39

Punctuation

Extended documentation for mathematical symbols & functions is *here*.

symbol	meaning
`@m`	invoke macro m; followed by space-separated expressions
`!`	prefix "not" operator
`a!( )`	at the end of a function name, *!* indicates that a function modifies its argument(s)
`#`	begin single line comment
`#=`	begin multi-line comment (these are nestable)
`=#`	end multi-line comment
`$`	xor operator, string and expression interpolation
`%`	remainder operator
`^`	exponent operator
`&`	bitwise and
`*`	multiply, or matrix multiply
`()`	the empty tuple
`~`	bitwise not operator
`\`	backslash operator
`'`	complex transpose operator A^H
`a[]`	array indexing
`[,]`	vertical concatenation
`[;]`	also vertical concatenation
`[ ]`	with space-separated expressions, horizontal concatenation
`T{ }`	parametric type instantiation
`{ }`	construct a cell array
`;`	statement separator
`,`	separate function arguments or tuple components
`?`	3-argument conditional operator (conditional ? if_true : if_false)
`""`	delimit string literals
`''`	delimit character literals
``	delimit external process (command) specifications

下页继续

表 39.1 – 续上页

symbol	meaning
`...`	splice arguments into a function call or declare a varargs function or type
`.`	access named fields in objects or names inside modules, also prefixes elementwise operators
`a:b`	range a, a+1, a+2, ..., b
`a:s:b`	range a, a+s, a+2s, ..., b
`:`	index an entire dimension (1:end)
`::`	type annotation, depending on context
`:( )`	quoted expression
`:a`	symbol a

CHAPTER 40

Sorting and Related Functions

Julia has an extensive, flexible API for sorting and interacting with already-sorted arrays of values. For many users, sorting in standard ascending order, letting Julia pick reasonable default algorithms will be sufficient:

```
julia> sort([2,3,1])
3-element Array{Int64,1}:
 1
 2
 3
```

You can easily sort in reverse order as well:

```
julia> sort([2,3,1], rev=true)
3-element Array{Int64,1}:
 3
 2
 1
```

To sort an array in-place, use the "bang" version of the sort function:

```
julia> a = [2,3,1];

julia> sort!(a);

julia> a
3-element Array{Int64,1}:
 1
 2
 3
```

Instead of directly sorting an array, you can compute a permutation of the array's indices that puts the array into sorted order:

```
julia> v = randn(5)
5-element Array{Float64,1}:
  0.297288
```

```
 0.382396
-0.597634
-0.0104452
-0.839027

julia> p = sortperm(v)
5-element Array{Int64,1}:
 5
 3
 4
 1
 2

julia> v[p]
5-element Array{Float64,1}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

Arrays can easily be sorted acording to an arbitrary transformation of their values:

```
julia> sort(v, by=abs)
5-element Array{Float64,1}:
 -0.0104452
  0.297288
  0.382396
 -0.597634
 -0.839027
```

Or in reverse order by a transformation:

```
julia> sort(v, by=abs, rev=true)
5-element Array{Float64,1}:
 -0.839027
 -0.597634
  0.382396
  0.297288
 -0.0104452
```

Reasonable sorting algorithms are used by default, but you can choose other algorithms as well:

```
julia> sort(v, alg=InsertionSort)
5-element Array{Float64,1}:
 -0.839027
 -0.597634
 -0.0104452
  0.297288
  0.382396
```

Sorting Functions

sort!(*v, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false]*)

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.

sort(*v, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false]*)
Variant of `sort!` that returns a sorted copy of `v` leaving `v` itself unmodified.

sort(*A, dim, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false]*)
Sort a multidimensional array `A` along the given dimension.

sortperm(*v, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false]*)
Return a permutation vector of indices of `v` that puts it in sorted order. Specify `alg` to choose a particular sorting algorithm (see *Sorting Algorithms*). `MergeSort` is used by default, and since it is stable, the resulting permutation will be the lexicographically first one that puts the input array into sorted order – i.e. indices of equal elements appear in ascending order. If you choose a non-stable sorting algorithm such as `QuickSort`, a different permutation that puts the array into order may be returned. The order is specified using the same keywords as `sort!`.

sortrows(*A, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false]*)
Sort the rows of matrix `A` lexicographically.

sortcols(*A, [alg=<algorithm>,] [by=<transform>,] [lt=<comparison>,] [rev=false]*)
Sort the columns of matrix `A` lexicographically.

Order-Related Functions

issorted(*v, [by=<transform>,] [lt=<comparison>,] [rev=false]*)
Test whether a vector is in sorted order. The `by`, `lt` and `rev` keywords modify what order is considered to be sorted just as they do for `sort`.

searchsorted(*a, x, [by=<transform>,] [lt=<comparison>,] [rev=false]*)
Returns the range of indices of `a` which compare as equal to `x` according to the order specified by the `by`, `lt` and `rev` keywords, assuming that `a` is already sorted in that order. Returns an empty range located at the insertion point if `a` does not contain values equal to `x`.

searchsortedfirst(*a, x, [by=<transform>,] [lt=<comparison>,] [rev=false]*)
Returns the index of the first value in `a` greater than or equal to `x`, according to the specified order. Returns `length(a)+1` if `x` is greater than all values in `a`.

searchsortedlast(*a, x, [by=<transform>,] [lt=<comparison>,] [rev=false]*)
Returns the index of the last value in `a` less than or equal to `x`, according to the specified order. Returns `0` if `x` is less than all values in `a`.

select!(*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. If `k` is a single index, that values is returned; if `k` is a range, an array of values at those indices is returned. Note that `select!` does not fully sort the input array, but does leave the returned elements where they would be if the array were fully sorted.

select(*v, k, [by=<transform>,] [lt=<comparison>,] [rev=false]*)
Variant of `select!` which copies `v` before partially sorting it, thereby returning the same thing as `select!` but leaving `v` unmodified.

Sorting Algorithms

There are currently three sorting algorithms available in base Julia:

- `InsertionSort`
- `QuickSort`
- `MergeSort`

`InsertionSort` is an O(n^2) stable sorting algorithm. It is efficient for very small `n`, and is used internally by `QuickSort`.

`QuickSort` is an O(n log n) sorting algorithm which is in-place, very fast, but not stable – i.e. elements which are considered equal will not remain in the same order in which they originally appeared in the array to be sorted. `QuickSort` is the default algorithm for numeric values, including integers and floats.

`MergeSort` is an O(n log n) stable sorting algorithm but is not in-place – it requires a temporary array of equal size to the input array – and is typically not quite as fast as `QuickSort`. It is the default algorithm for non-numeric data.

The sort functions select a reasonable default algorithm, depending on the type of the array to be sorted. To force a specific algorithm to be used for `sort` or other soring functions, supply `alg=<algorithm>` as a keyword argument after the array to be sorted.

CHAPTER 41

Package Manager Functions

All package manager functions are defined in the `Pkg` module. None of the `Pkg` module's functions are exported; to use them, you'll need to prefix each function call with an explicit `Pkg.`, e.g. `Pkg.status()` or `Pkg.dir()`.

`dir``()` → String
: Returns the absolute path of the package directory. This defaults to `joinpath(homedir(),".julia")` on all platforms (i.e. `~/.julia` in UNIX shell syntax). If the `JULIA_PKGDIR` environment variable is set, that path is used instead. If `JULIA_PKGDIR` is a relative path, it is interpreted relative to whatever the current working directory is.

`dir``(`*names...*`)` → String
: Equivalent to `normpath(Pkg.dir(),names...)` – i.e. it appends path components to the package directory and normalizes the resulting path. In particular, `Pkg.dir(pkg)` returns the path to the package `pkg`.

`init``()`
: Initialize `Pkg.dir()` as a package directory. This will be done automatically when the `JULIA_PKGDIR` is not set and `Pkg.dir()` uses its default value.

`resolve``()`
: Determines an optimal, consistent set of package versions to install or upgrade to. The optimal set of package versions is based on the contents of `Pkg.dir("REQUIRE")` and the state of installed packages in `Pkg.dir()`, Packages that are no longer required are moved into `Pkg.dir(".trash")`.

`edit``()`
: Opens `Pkg.dir("REQUIRE")` in the editor specified by the `VISUAL` or `EDITOR` environment variables; when the editor command returns, it runs `Pkg.resolve()` to determine and install a new optimal set of installed package versions.

`add``(`*pkg*, *vers...*`)`
: Add a requirement entry for `pkg` to `Pkg.dir("REQUIRE")` and call `Pkg.resolve()`. If `vers` are given, they must be `VersionNumber` objects and they specify acceptable version intervals for `pkg`.

`rm``(`*pkg*`)`
: Remove all requirement entries for `pkg` from `Pkg.dir("REQUIRE")` and call `Pkg.resolve()`.

`clone``(`*url*[, *pkg*]`)`
: Clone a package directly from the git URL `url`. The package does not need to be a registered in `Pkg.`

`dir("METADATA")`. The package repo is cloned by the name `pkg` if provided; if not provided, `pkg` is determined automatically from `url`.

clone(*pkg*)
If `pkg` has a URL registered in `Pkg.dir("METADATA")`, clone it from that URL on the default branch. The package does not need to have any registered versions.

available() → Vector{ASCIIString}
Returns the names of available packages.

available(*pkg*) → Vector{VersionNumber}
Returns the version numbers available for package `pkg`.

installed() → Dict{ASCIIString,VersionNumber}
Returns a dictionary mapping installed package names to the installed version number of each package.

installed(*pkg*) → Nothing | VersionNumber
If `pkg` is installed, return the installed version number, otherwise return `nothing`.

status()
Prints out a summary of what packages are installed and what version and state they're in.

update()
Update package the metadata repo – kept in `Pkg.dir("METADATA")` – then update any fixed packages that can safely be pulled from their origin; then call `Pkg.resolve()` to determine a new optimal set of packages versions.

checkout(*pkg*[, *branch="master"*])
Checkout the `Pkg.dir(pkg)` repo to the branch `branch`. Defaults to checking out the "master" branch. To go back to using the newest compatible released version, use `Pkg.free(pkg)`

pin(*pkg*)
Pin `pkg` at the current version. To go back to using the newest compatible released version, use `Pkg.free(pkg)`

pin(*pkg*, *version*)
Pin `pkg` at registered version `version`.

free(*pkg*)
Free the package `pkg` to be managed by the package manager again. It calls `Pkg.resolve()` to determine optimal package versions after. This is an inverse for both `Pkg.checkout` and `Pkg.pin`.

build()
Run the build scripts for all installed packages in depth-first recursive order.

build(*pkgs...*)
Run the build script in "deps/build.jl" for each package in `pkgs` and all of their dependencies in depth-first recursive order. This is called automatically by `Pkg.resolve()` on all installed or updated packages.

generate(*pkg*, *license*)
Generate a new package named `pkg` with one of these license keys: `"MIT"` or `"BSD"`. If you want to make a package with a different license, you can edit it afterwards. Generate creates a git repo at `Pkg.dir(pkg)` for the package and inside it `LICENSE.md`, `README.md`, the julia entrypoint `$pkg/src/$pkg.jl`, and a travis test file, `.travis.yml`.

register(*pkg*[, *url*])
Register `pkg` at the git URL `url`, defaulting to the configured origin URL of the git repo `Pkg.dir(pkg)`.

tag(*pkg*[, *ver*[, *commit*]])
Tag `commit` as version `ver` of package `pkg` and create a version entry in `METADATA`. If not provided, `commit` defaults to the current commit of the `pkg` repo. If `ver` is one of the symbols `:patch`, `:minor`, `:major` the next patch, minor or major version is used. If `ver` is not provided, it defaults to `:patch`.

publish()

For each new package version tagged in `METADATA` not already published, make sure that the tagged package commits have been pushed to the repo at the registered URL for the package and if they all have, open a pull request to `METADATA`.

test()

Run the tests for all installed packages ensuring that each package's test dependencies are installed for the duration of the test. A package is tested by running its `test/runtests.jl` file and test dependencies are specified in `test/REQUIRE`.

test(*pkgs...*)

Run the tests for each package in `pkgs` ensuring that each package's test dependencies are installed for the duration of the test. A package is tested by running its `test/runtests.jl` file and test dependencies are specified in `test/REQUIRE`.

CHAPTER 42

Collections and Data Structures

The `Collections` module contains implementations of some common data structures.

PriorityQueue

The `PriorityQueue` type is a basic priority queue implementation allowing for arbitrary key and priority types. Multiple identical keys are not permitted, but the priority of existing keys can be changed efficiently.

`PriorityQueue{K,V}`(*[ord]*)
: Construct a new PriorityQueue, with keys of type `K` and values/priorites of type `V`. If an order is not given, the priority queue is min-ordered using the default comparison for `V`.

`enqueue!`(*pq*, *k*, *v*)
: Insert the a key `k` into a priority queue `pq` with priority `v`.

`dequeue!`(*pq*)
: Remove and return the lowest priority key from a priority queue.

`peek`(*pq*)
: Return the lowest priority key from a priority queue without removing that key from the queue.

`PriorityQueue` also behaves similarly to a `Dict` so that keys can be inserted and priorities accessed or changed using indexing notation:

```
# Julia code
pq = Collections.PriorityQueue()

# Insert keys with associated priorities
pq["a"] = 10
pq["b"] = 5
pq["c"] = 15

# Change the priority of an existing key
pq["a"] = 0
```

Heap Functions

Along with the `PriorityQueue` type are lower level functions for performing binary heap operations on arrays. Each function takes an optional ordering argument. If not given, default ordering is used, so that elements popped from the heap are given in ascending order.

`heapify`(*v*[, *ord*])
Return a new vector in binary heap order, optionally using the given ordering.

`heapify!`(*v*[, *ord*])
In-place heapify.

`isheap`(*v*[, *ord*])
Return true iff an array is heap-ordered according to the given order.

`heappush!`(*v*, *x*[, *ord*])
Given a binary heap-ordered array, push a new element `x`, preserving the heap property. For efficiency, this function does not check that the array is indeed heap-ordered.

`heappop!`(*v*[, *ord*])
Given a binary heap-ordered array, remove and return the lowest ordered element. For efficiency, this function does not check that the array is indeed heap-ordered.

CHAPTER 43

Graphics

The `Base.Graphics` interface is an abstract wrapper; specific packages (e.g., Cairo and Tk/Gtk) implement much of the functionality.

Geometry

`Vec2` (*x*, *y*)
: Creates a point in two dimensions

`BoundingBox` (*xmin*, *xmax*, *ymin*, *ymax*)
: Creates a box in two dimensions with the given edges

`BoundingBox` (*objs...*)
: Creates a box in two dimensions that encloses all objects

`width` (*obj*)
: Computes the width of an object

`height` (*obj*)
: Computes the height of an object

`xmin` (*obj*)
: Computes the minimum x-coordinate contained in an object

`xmax` (*obj*)
: Computes the maximum x-coordinate contained in an object

`ymin` (*obj*)
: Computes the minimum y-coordinate contained in an object

`ymax` (*obj*)
: Computes the maximum y-coordinate contained in an object

`diagonal` (*obj*)
: Return the length of the diagonal of an object

aspect_ratio(*obj*)
Compute the height/width of an object

center(*obj*)
Return the point in the center of an object

xrange(*obj*)
Returns a tuple `(xmin(obj), xmax(obj))`

yrange(*obj*)
Returns a tuple `(ymin(obj), ymax(obj))`

rotate(*obj*, *angle*, *origin*) → newobj
Rotates an object around origin by the specified angle (radians), returning a new object of the same type. Because of type-constancy, this new object may not always be a strict geometric rotation of the input; for example, if `obj` is a `BoundingBox` the return is the smallest `BoundingBox` that encloses the rotated input.

shift(*obj*, *dx*, *dy*)
Returns an object shifted horizontally and vertically by the indicated amounts

*****(*obj*, *s::Real*)
Scale the width and height of a graphics object, keeping the center fixed

+(*bb1::BoundingBox*, *bb2::BoundingBox*) → BoundingBox
Returns the smallest box containing both boxes

&(*bb1::BoundingBox*, *bb2::BoundingBox*) → BoundingBox
Returns the intersection, the largest box contained in both boxes

deform(*bb::BoundingBox*, *dxmin*, *dxmax*, *dymin*, *dymax*)
Returns a bounding box with all edges shifted by the indicated amounts

isinside(*bb::BoundingBox*, *x*, *y*)
True if the given point is inside the box

isinside(*bb::BoundingBox*, *point*)
True if the given point is inside the box

CHAPTER 44

Unit and Functional Testing

The `Test` module contains macros and functions related to testing. A default handler is provided to run the tests, and a custom one can be provided by the user by using the `registerhandler()` function.

Overview

To use the default handler, the macro `@test()` can be used directly:

```
julia> using Base.Test

julia> @test 1 == 1

julia> @test 1 == 0
ERROR: test failed: 1 == 0
 in error at error.jl:21
 in default_handler at test.jl:19
 in do_test at test.jl:39

julia> @test error("This is what happens when a test fails")
ERROR: test error during error("This is what happens when a test fails")
This is what happens when a test fails
 in error at error.jl:21
 in anonymous at test.jl:62
 in do_test at test.jl:37
```

As seen in the examples above, failures or errors will print the abstract syntax tree of the expression in question.

Another macro is provided to check if the given expression throws an exception of type `extype`, `@test_throws()`:

```
julia> @test_throws ErrorException error("An error")

julia> @test_throws BoundsError error("An error")
ERROR: test failed: error("An error")
```

```
 in error at error.jl:21
 in default_handler at test.jl:19
 in do_test_throws at test.jl:55

julia> @test_throws DomainError throw(DomainError())

julia> @test_throws DomainError throw(EOFError())
ERROR: test failed: throw(EOFError())
 in error at error.jl:21
 in default_handler at test.jl:19
 in do_test_throws at test.jl:55
```

As floating point comparisons can be imprecise, two additional macros exist taking in account small numerical errors:

```
julia> @test_approx_eq 1. 0.999999999
ERROR: assertion failed: |1.0 - 0.999999999| < 2.220446049250313e-12
  1.0 = 1.0
  0.999999999 = 0.999999999
 in test_approx_eq at test.jl:75
 in test_approx_eq at test.jl:80

julia> @test_approx_eq 1. 0.9999999999999

julia> @test_approx_eq_eps 1. 0.999 1e-2

julia> @test_approx_eq_eps 1. 0.999 1e-3
ERROR: assertion failed: |1.0 - 0.999| <= 0.001
  1.0 = 1.0
  0.999 = 0.999
  difference = 0.0010000000000000009 > 0.001
 in error at error.jl:22
 in test_approx_eq at test.jl:68
```

Handlers

A handler is a function defined for three kinds of arguments: `Success`, `Failure`, `Error`:

```
# The definition of the default handler
default_handler(r::Success) = nothing
default_handler(r::Failure) = error("test failed: $(r.expr)")
default_handler(r::Error)   = rethrow(r)
```

A different handler can be used for a block (with `with_handler()`):

```
julia> using Base.Test

julia> custom_handler(r::Test.Success) = println("Success on $(r.expr)")
custom_handler (generic function with 1 method)

julia> custom_handler(r::Test.Failure) = error("Error on custom handler: $(r.expr)")
custom_handler (generic function with 2 methods)

julia> custom_handler(r::Test.Error) = rethrow(r)
custom_handler (generic function with 3 methods)
```

```
julia> Test.with_handler(custom_handler) do
         @test 1 == 1
         @test 1 != 1
       end
Success on :((1==1))
ERROR: Error on custom handler: :((1!=1))
 in error at error.jl:21
 in custom_handler at none:1
 in do_test at test.jl:39
 in anonymous at no file:3
 in task_local_storage at task.jl:28
 in with_handler at test.jl:24
```

Macros

@test (*ex*)
Test the expression `ex` and calls the current handler to handle the result.

@test_throws (*extype*, *ex*)
Test that the expression `ex` throws an exception of type `extype` and calls the current handler to handle the result.

@test_approx_eq (*a*, *b*)
Test two floating point numbers `a` and `b` for equality taking in account small numerical errors.

@test_approx_eq_eps (*a*, *b*, *tol*)
Test two floating point numbers `a` and `b` for equality taking in account a margin of tolerance given by `tol`.

Functions

with_handler (*f*, *handler*)
Run the function `f` using the `handler` as the handler.

CHAPTER 45

Testing Base Julia

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()`.

`runtests`([*tests=["all"]*[, *numcores=iceil(CPU_CORES/2)*]])

Run the Julia unit tests listed in `tests`, which can be either a string or an array of strings, using `numcores` processors.

CHAPTER 46

Profiling

The `Profile` module provides tools to help developers improve the performance of their code. When used, it takes measurements on running code, and produces output that helps you understand how much time is spent on individual line(s). The most common usage is to identify "bottlenecks" as targets for optimization.

`Profile` implements what is known as a "sampling" or statistical profiler. It works by periodically taking a backtrace during the execution of any task. Each backtrace captures the currently-running function and line number, plus the complete chain of function calls that led to this line, and hence is a "snapshot" of the current state of execution.

If much of your run time is spent executing a particular line of code, this line will show up frequently in the set of all backtraces. In other words, the "cost" of a given line—or really, the cost of the sequence of function calls up to and including this line—is proportional to how often it appears in the set of all backtraces.

A sampling profiler does not provide complete line-by-line coverage, because the backtraces occur at intervals (by default, 1 ms). However, this design has important strengths:

- You do not have to make any modifications to your code to take timing measurements (in contrast to the alternative instrumenting profiler).
- It can profile into Julia's core code and even (optionally) into C and Fortran libraries.
- By running "infrequently" there is very little performance overhead; while profiling, your code will run at nearly native speed.

For these reasons, it's recommended that you try using the built-in sampling profiler before considering any alternatives.

Basic usage

Let's work with a simple test case:

```
function myfunc()
    A = rand(100, 100, 200)
    maximum(A)
end
```

It's a good idea to first run the code you intend to profile at least once (unless you want to profile Julia's JIT-compiler):

```
julia> myfunc()  # run once to force compilation
```

Now we're ready to profile this function:

```
julia> @profile myfunc()
```

To see the profiling results, there is a graphical browser available, but here we'll use the text-based display that comes with the standard library:

```
julia> Profile.print()
      23 client.jl; _start; line: 373
        23 client.jl; run_repl; line: 166
           23 client.jl; eval_user_input; line: 91
              23 profile.jl; anonymous; line: 14
                 8  none; myfunc; line: 2
                  8 dSFMT.jl; dsfmt_gv_fill_array_close_open!; line: 128
                 15 none; myfunc; line: 3
                  2  reduce.jl; max; line: 35
                  2  reduce.jl; max; line: 36
                  11 reduce.jl; max; line: 37
```

Each line of this display represents a particular spot (line number) in the code. Indentation is used to indicate the nested sequence of function calls, with more-indented lines being deeper in the sequence of calls. In each line, the first "field" indicates the number of backtraces (samples) taken *at this line or in any functions executed by this line*. The second field is the file name, followed by a semicolon; the third is the function name followed by a semicolon, and the fourth is the line number. Note that the specific line numbers may change as Julia's code changes; if you want to follow along, it's best to run this example yourself.

In this example, we can see that the top level is `client.jl`'s `_start` function. This is the first Julia function that gets called when you launch julia. If you examine line 373 of `client.jl`, you'll see that (at the time of this writing) it calls `run_repl`, mentioned on the second line. This in turn calls `eval_user_input`. These are the functions in `client.jl` that interpret what you type at the REPL, and since we're working interactively these functions were invoked when we entered `@profile myfunc()`. The next line reflects actions taken in the `@profile` macro.

The first line shows that 23 backtraces were taken at line 373 of `client.jl`, but it's not that this line was "expensive" on its own: the second line reveals that all 23 of these backtraces were actually triggered inside its call to `run_repl`, and so on. To find out which operations are actually taking the time, we need to look deeper in the call chain.

The first "important" line in this output is this one:

```
8  none; myfunc; line: 2
```

`none` refers to the fact that we defined `myfunc` in the REPL, rather than putting it in a file; if we had used a file, this would show the file name. Line 2 of `myfunc()` contains the call to `rand`, and there were 8 (out of 23) backtraces that occurred at this line. Below that, you can see a call to `dsfmt_gv_fill_array_close_open!` inside `dSFMT.jl`. You might be surprised not to see the `rand` function listed explicitly: that's because `rand` is *inlined*, and hence doesn't appear in the backtraces.

A little further down, you see:

```
15 none; myfunc; line: 3
```

Line 3 of `myfunc` contains the call to `max`, and there were 15 (out of 23) backtraces taken here. Below that, you can see the specific places in `base/reduce.jl` that carry out the time-consuming operations in the `max` function for this type of input data.

Overall, we can tentatively conclude that finding the maximum element is approximately twice as expensive as generating the random numbers. We could increase our confidence in this result by collecting more samples:

```
julia> @profile (for i = 1:100; myfunc(); end)

julia> Profile.print()
       3121 client.jl; _start; line: 373
        3121 client.jl; run_repl; line: 166
           3121 client.jl; eval_user_input; line: 91
              3121 profile.jl; anonymous; line: 1
                 848  none; myfunc; line: 2
                  842 dSFMT.jl; dsfmt_gv_fill_array_close_open!; line: 128
                 1510 none; myfunc; line: 3
                  74   reduce.jl; max; line: 35
                  122  reduce.jl; max; line: 36
                  1314 reduce.jl; max; line: 37
```

In general, if you have `N` samples collected at a line, you can expect an uncertainty on the order of `sqrt(N)` (barring other sources of noise, like how busy the computer is with other tasks). The major exception to this rule is garbage-collection, which runs infrequently but tends to be quite expensive. (Since julia's garbage collector is written in C, such events can be detected using the `C=true` output mode described below, or by using ProfileView.)

This illustrates the default "tree" dump; an alternative is the "flat" dump, which accumulates counts independent of their nesting:

```
julia> Profile.print(format=:flat)
 Count File         Function                            Line
  3121 client.jl    _start                               373
  3121 client.jl    eval_user_input                       91
  3121 client.jl    run_repl                             166
   842 dSFMT.jl     dsfmt_gv_fill_array_close_open!      128
   848 none         myfunc                                 2
  1510 none         myfunc                                 3
  3121 profile.jl   anonymous                              1
    74 reduce.jl    max                                   35
   122 reduce.jl    max                                   36
  1314 reduce.jl    max                                   37
```

If your code has recursion, one potentially-confusing point is that a line in a "child" function can accumulate more counts than there are total backtraces. Consider the following function definitions:

```
dumbsum(n::Integer) = n == 1 ? 1 : 1 + dumbsum(n-1)
dumbsum3() = dumbsum(3)
```

If you were to profile `dumbsum3`, and a backtrace was taken while it was executing `dumbsum(1)`, the backtrace would look like this:

```
dumbsum3
    dumbsum(3)
        dumbsum(2)
            dumbsum(1)
```

Consequently, this child function gets 3 counts, even though the parent only gets one. The "tree" representation makes this much clearer, and for this reason (among others) is probably the most useful way to view the results.

Accumulation and clearing

Results from `@profile` accumulate in a buffer; if you run multiple pieces of code under `@profile`, then `Profile.print()` will show you the combined results. This can be very useful, but sometimes you want to start fresh; you can do so with `Profile.clear()`.

Options for controlling the display of profile results

`Profile.print()` has more options than we've described so far. Let's see the full declaration:

```
function print(io::IO = STDOUT, data = fetch(); format = :tree, C = false, combine =
↪true, cols = tty_cols())
```

Let's discuss these arguments in order:

- The first argument allows you to save the results to a file, but the default is to print to `STDOUT` (the console).
- The second argument contains the data you want to analyze; by default that is obtained from `Profile.fetch()`, which pulls out the backtraces from a pre-allocated buffer. For example, if you want to profile the profiler, you could say:

```
data = copy(Profile.fetch())
Profile.clear()
@profile Profile.print(STDOUT, data) # Prints the previous results
Profile.print()                      # Prints results from Profile.print()
```

- The first keyword argument, `format`, was introduced above. The possible choices are `:tree` and `:flat`.
- `C`, if set to `true`, allows you to see even the calls to C code. Try running the introductory example with `Profile.print(C = true)`. This can be extremely helpful in deciding whether it's Julia code or C code that is causing a bottleneck; setting `C=true` also improves the interpretability of the nesting, at the cost of longer profile dumps.
- Some lines of code contain multiple operations; for example, `s += A[i]` contains both an array reference (`A[i]`) and a sum operation. These correspond to different lines in the generated machine code, and hence there may be two or more different addresses captured during backtraces on this line. `combine=true` lumps them together, and is probably what you typically want, but you can generate an output separately for each unique instruction pointer with `combine=false`.
- `cols` allows you to control the number of columns that you are willing to use for display. When the text would be wider than the display, you might see output like this:

```
33 inference.jl; abstract_call; line: 645
  33 inference.jl; abstract_call; line: 645
    33 ...rence.jl; abstract_call_gf; line: 567
       33 ...nce.jl; typeinf; line: 1201
     +1 5  ...nce.jl; ...t_interpret; line: 900
     +3 5 ...ence.jl; abstract_eval; line: 758
     +4 5 ...ence.jl; ...ct_eval_call; line: 733
     +6 5 ...ence.jl; abstract_call; line: 645
```

  File/function names are sometimes truncated (with `...`), and indentation is truncated with a `+n` at the beginning, where `n` is the number of extra spaces that would have been inserted, had there been room. If you want a complete profile of deeply-nested code, often a good idea is to save to a file and use a very wide `cols` setting:

```
s = open("/tmp/prof.txt","w")
Profile.print(s,cols = 500)
close(s)
```

Configuration

`@profile` just accumulates backtraces, and the analysis happens when you call `Profile.print()`. For a long-running computation, it's entirely possible that the pre-allocated buffer for storing backtraces will be filled. If that happens, the backtraces stop but your computation continues. As a consequence, you may miss some important profiling data (you will get a warning when that happens).

You can obtain and configure the relevant parameters this way:

```
Profile.init()            # returns the current settings
Profile.init(n, delay)
Profile.init(delay = 0.01)
```

`n` is the total number of instruction pointers you can store, with a default value of `10^6`. If your typical backtrace is 20 instruction pointers, then you can collect 50000 backtraces, which suggests a statistical uncertainty of less than 1%. This may be good enough for most applications.

Consequently, you are more likely to need to modify `delay`, expressed in seconds, which sets the amount of time that Julia gets between snapshots to perform the requested computations. A very long-running job might not need frequent backtraces. The default setting is `delay = 0.001`. Of course, you can decrease the delay as well as increase it; however, the overhead of profiling grows once the delay becomes similar to the amount of time needed to take a backtrace (~30 microseconds on the author's laptop).

Function reference

`@profile``()`
`@profile <expression>` runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

`clear``()`
Clear any existing backtraces from the internal buffer.

`print``([`*io::IO = STDOUT*`], [`*data::Vector*`]; format = :tree, C = false, combine = true, cols = tty_cols())`
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. `format` can be `:tree` or `:flat`. If `C==true`, backtraces from C and Fortran code are shown. `combine==true` merges instruction pointers that correspond to the same line of code. `cols` controls the width of the display.

`print``([`*io::IO = STDOUT*`], data::Vector, lidict::Dict; format = :tree, combine = true, cols = tty_cols())`
Prints profiling results to `io`. This variant is used to examine results exported by a previous call to `Profile.retrieve()`. Supply the vector `data` of backtraces and a dictionary `lidict` of line information.

`init``(; n::Integer, delay::Float64)`
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)`.

fetch() → data

Returns a reference to the internal buffer of backtraces. Note that subsequent operations, like `Profile.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; `Profile.retrieve()` may be a better choice for most users.

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.

Bibliography

[Bischof1987] 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

[Schreiber1989] 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

[Bunch1977] 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.

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z