Julia中文手册

极客学院出版

前言

Julia 是一个新的高性能动态高级编程语言。语法和其他编程语言类似，易于其他语言用户学习。Julia 拥有丰富的函数库，提供了数字精度、精致的增幅器（sophisticated amplifier）和分布式并行运行方式。核心函数库等大多数库是由 Julia 编写，但也用成熟的 C 和 FORTRAN 库来处理线性代数、随机数产生和字符串处理等问题。

Julia 是个灵活的动态语言，适合科学和数值计算，性能可与传统静态类型语言媲美。

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

本课程是 Julia 官方文档的中文译本。

适用人群

本课程主要适用于希望了解并深入学习 Julia 语言用以编写高性能程序的国内读者。

学习前提

在学习本课程之前，我们假定你对动态编程语言有一定的了解，并对基本的编程知识熟练掌握。

> 英文文档地址: http://docs.julialang.org/en/release-0.3/

目录

1 简介

Julia 是个灵活的动态语言，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 的编译器与像 Python 或者 R 语言的解释器不同，你可能首先会发现 Julia 的性能并不那么直观。如果你发现哪些地方比较慢， 我们强烈建议你在做任何尝试之前通读一下代码性能优化章节。一旦你明白了 Julia 是如何工作的，你就可以写出来速度媲美 C 语言的代码。

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href="https://github.com/JuliaLang/julia/blob/master/LICENSE.md"））
- 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

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()" to list help topics
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.3.0-prerelease+3690 (2014-06-16 05:11 UTC)
 _/ |\__'_|_|_|\__'_|  |  Commit 1b73f04* (0 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
```

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` 默认为当前的用户， `port` 默认为标准 ssh 端口。可选择的，万一是多网主机， `bind_addr` 可被用来精确指定接口。

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! 你好! ??????")' > ~/.juliarc.jl
$ julia
Greetings! 你好! ??????

...
```

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

3

变量

在 Julia 中的一个变量是一个与一个值关联（或绑定）的名称。它的作用表现在当你想存储一个值（例如，你在进行一些数学运算后得到了一些值，你需要在之后使用到这些值）时。例如：

```
# 给变量 x 赋值为 10

julia> x = 10
10

# 用 x 的值做一些数学运算

julia> x + 1
11

# 重新给 x 赋值

julia> x = 1 + 1
2

# 您可以为变量赋给种类型的值，例如文本字符串等

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> ????? = "Hello"
"Hello"
```

在 Julia REPL 和其他几个 Julia 编辑环境中，您可以通过输入反斜杠符号名称后再输入标签来键入很多 Unicode 数学符号。例如，变量名 `δ` 可以通过键入 `\delta` 键入，甚至可以通过输入 `\alpha` − *tab* − `\hat` − *tab* − `\_2` − *tab* 输入 `α̂₂` 。

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

变量名必须以字母（a–z 或 A–Z），下划线，或一个 Unicode 编码指针中指向比 00A0 更大的指针子集开始；特别是 Unicode 字符 Lu/Ll/Lt/Lm/Lo/Nl（字母），Sc/So（货币和其他符号），和其他一些可以看做字符的一些输入（例如 Sm 数学符号的子集）是允许的。首位之后的字符也包括！ 和数字（0–9 和其他字符 Nd/No），以及其他 Unicode 编码指针：变音符号和其他修改标记（字母 Mn/Mc/Me/Sk），一些标点连接器（字母 Pc），素数，和其他的一些字符。

运算符类似 `+` 也是有效的标识符，但需要特别解析。在某些情况下，运算符可以像变量一样使用；例如 `(+)` 是指增加功能，和 `(+) = f` 将重新定义这个运算。大多数的 Unicode 中缀操作符（在 Sm 中），如 `⊕`，会被解析为中缀操作符，同时可以自定义方法（例如，你可以使用 `⊗ = kron` 定义 `⊕` 成为一个中缀 Kronecker 积）。

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

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href="http://zh.wikipedia.org/zh-cn/Unicode)；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*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` 功能）有两个额外的样例：被 0 除，和被最低的负数（ `typemin` ）-1 除。两个例子都抛出了一个 `DivideError` 。余数和模运算（ `rem` 和 `mod` ）当它们的第二个参数为 0 时，抛出了一个 `DivideError` 。

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	Inft32	Inf	正无穷	比所有的有限的浮点数都大
-Inf16	-Inft32	-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(https://gmplib.org/) 和 GNU MPFR Library (http://www.mpfr.org/)。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")
1.2345678901234567890100000000000000000000000000000000000000000000000000000000000000000000000000004e+00 with 256 bits of

julia> BigFloat(2.0^66) / 3
2.4595658764946068821333333333333333333333333333333333333333333333333333333333333333333333333344e+19 with 256 bits of

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
1.100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000003e+00 with 256 bits o

julia> with_rounding(BigFloat,RoundDown) do
       BigFloat(1) + BigFloat("0.1")
       end
1.099999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999986e+00 with 256 bits o

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)
1e+00 with 256 bits of precision
```

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：

表达式	名称
`~x`	按位取反
`x & y`	按位与
`x | y`	按位或
`x $ y`	按位异或
`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 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	`.` 跟随 `::`
幂	`^` 和 `.^` 等效
分数	`//` 和 `.//`
乘除	`/ % & \` 和 `.* ./ .% .\`
位移	`<< >> >>>` 和 `.<< .>> .>>>`
加减	`+ − | $` 和 `.+ .−`
语法	`: ..` 跟随于 `|>`
比较	`> < >= <= == === != !== <:` 和 `.> .< .>= .<= .== .!=`
逻辑	`&&` 跟随于 `||` 跟随于 `?`
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

函数	描述
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)	√x x 的平方根
cbrt(x)	?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 。

另外，`sinpi(x)` 和 `cospi(x)` 各自被提供给更准确的 `sin(pi*x)` 和 `cos(pi*x)` 的计算。

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)	将误差函数定义为 −im * erf(x * im)，其中 im 是虚数单位
erfcx(x)	缩放的互补误差函数，即对较大的 x 值的准确的 exp（x ^ 2）* erfc（x）
dawson(x)	缩放虚误差函数，又名道森函数，即对较大的 x 值求精确的 exp(−x^2) * erfi(x) * sqrt(pi) / 2
gamma(x)	x 处的 gamma 函数
lgamma(x)	当 x 较大时，精确计算 log(gamma(x))
lfact(x)	对较大的 x 求精确的 log(factorial(x)); 与对大于 1 的 x 值求 lgamma(x+1) 相等, 否则等于 0
digamma(x)	x 处的 digamma 函数，即导数的衍生
beta(x,y)	在（x,y）处的 beta 函数
lbeta(x,y)	对较大的 x 或 y 值求准确的 log(beta(x,y))
eta(x)	x 处的 Dirichlet eta 函数
zeta(x)	x 处的 Riemann zeta 函数
airy(z)，airyai(z)，airy(0,z)	z 处的 Airy Ai 函数
airyprime(z)，airyaiprime(z)，airy(1,z)	Airy Ai 函数在 z 处的导数
airybi(z)，airy(2,z)	z 处的 Airy Bi 函数
airybiprime(z)，airy(3,z)	Airy Bi 函数在 z 处的导数
airyx(z)，airyx(k,z)	缩放 Airy Ai 函数 以及 k 对 z 的导数
besselj(nu,z)	对 z 中一阶 nu 的贝塞尔函数
besselj0(z)	besselj(0,z)

函数	描述
besselj1(z)	besselj(1,z)
besseljx(nu,z)	对 z 中一阶 nu 的缩放贝塞尔函数
bessely(nu,z)	对 z 中二阶 nu 的贝塞尔函数
bessely0(z)	bessely(0,z)
bessely1(z)	bessely(1,z)
besselyx(nu,z)	对 z 中二阶 nu 的缩放贝塞尔函数
besselh(nu,k,z)	对 z 中三阶 nu（例如汉克尔函数）的贝塞尔函数； k 必须为 1 或 2
hankelh1(nu,z)	besselh(nu, 1, z)
hankelh1x(nu,z)	缩放 besselh(nu, 1, z)
hankelh2(nu,z)	besselh(nu, 2, z)
hankelh2x(nu,z)	缩放 besselh(nu, 2, z)
besseli(nu,z)	对 z 中一阶 nu 的修正贝塞尔函数
besselix(nu,z)	对 z 中一阶 nu 的缩放修正贝塞尔函数
besselk(nu,z)	对 z 中二阶 nu 的修正贝塞尔函数
besselkx(nu,z)	对二阶 o 的缩放修正贝塞尔函数

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href="http://julia-cn.readthedocs.org/zh_CN/latest/manual/mathematical-operations/#man-mathematical-operations）。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
```

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：

- `String` 是个抽象类型，不是具体类型
- 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 `is_valid_char` 函数：

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

julia> is_valid_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。特别是，Julia 还提供了 `utf16string` 和 `utf32string` 类型，由 `utf16(s)` 和 `utf32(s)` 函数分别支持 UTF-16 和 UTF-32 编码。它还为 UTF-16 或 UTF-32 字符串提供了别名 `WString` 和 `wstring(s)`，两者的选择取决于 `Cwchar_t` 大小。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.
```

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*href="http://julia-cn.readthedocs.org/zh_CN/latest/manual/metaprogramming/#man-non-standard-string-literals2) 。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")
```

当调用 `match` 时，你可以选择指定一个索引，它指示在哪里开始搜索。比如：

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

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

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

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。

... 三重引号的正则字符串，形式为 `r"""..."""`，也是 ... 支持的（可能对于含有 ... 等式标记或换行的正则表达式是方便的）。

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。

版本号文字

版本号可以很容易地用非标准字符串的形式 `v"..."` 表示。版本号会遵循语义版本的规范创建 `VersionNumber` 对象，因此版本号主要是由主版本号，次版本号和补丁的值决定的，其后是预发布和创建的数字注释。例如，`v"0.2.1-rc1+win64"` 可以被分块解释为主版本 `0`，次要版本 `2`，补丁版本 `1`，预发布 RC1 和创建为 Win64。当输入一个版本号时，除了主版本号的其余字段都是可选的，因此，会出现例如 `v"0.2"` 与 `v"0.2.0"` 等效（空预发布/创建注释），`v"2"` 与 `v"2.0.0"` 等效，等等。

`VersionNumber` 对象大多是能做到容易且准确地比较两个（或更多）的版本。例如，恒定的 `VERSION` 把 Julia 版本号作为一个 `VersionNumber` 对象管理，因此可以使用简单的语句定义一些特定版本的行为，例如：

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

既然在上面的示例中使用了非标准的版本号 `v"0.3-"`,它使用了一个后连接号：此符号是一个 Julia 扩展的标准符号，它是用来表示一个低于任何 0.3 发行版的版本，其中包括其所有的预发行版本。所以在上面的例子中的代码只会运行稳定在 `0.2` 版本，并不能运行在这样的版本 `v"0.3.0-rc1"`。为了允许它也在不稳定的（即预发行版）0.2 版上运行，较低的检查应修改为 `v"0.2-" <= VERSION`。

另一个非标准版规范扩展允许对使用尾部 `+` 来表达一个上限构建版本，例如 `VERSION > "v"0.2-rc1+"` 可以被用来表示任何版本在 `0.2-rc1` 之上且任何创建形式的版本：对于版本 `v"0.2-rc1+win64"` 将返回 false ,而对于 `v"0.2-rc2"` 会返回 true 。

使用这种特殊的版本比较是好的尝试（特别是，尾随 `-` 应该总是被使用在上限规范，除非有一个很好的理由不去这样），但这样的形式不得被当作任何的实际版本号使用，因为在语义版本控制方案上它们是非法的。

除了用于 `VERSION` 常数，`VersionNumber` 对象还广泛应用于 `Pkg` 模块，来指定包的版本和它们的依赖关系。

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 + 2x - 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` 的语法创建一个含有参数 `x` 的匿名函数，并将其传给 `map` 作为第一个参数。类似地， `do a,b` 将创建一个含有两个参数的匿名函数，和一个朴素的 `do` 的声明以 `() -> ....` 方式说明如下是一个匿名函数。

如何将这些参数初始化取决于“外部”函数；在这里， `map` 将依次设置 `x` 到 `A` , `B` , `C` ，每个都将调用匿名函数，就像在语法 `map(func, [A, B, C])` 中做的一样。

因为语法的调用看起来像正常的代码块，所以这种语法使它更容易使用函数来有效地扩展语言。这里有许多可能完全不同于 `map` 的用途，如管理系统状态。例如，有一个版本的 `open` ，运行代码来确保打开的文件最终关闭：

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

它可以通过以下定义来实现：

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

对比的 `map` 的例子，这里的 IO 是通过 `open("outfile", "w")` 来实现初始化的。字符流之后会传递给您的执行写入的匿名函数；最后，`open` 的功能确保流在您的函数结束后是关闭状态的。`try/finally` 的构造将在 *控制流* 中被描述。

do 块语法的使用有助于检查文档或实现了解用户函数的参数是如何被初始化的。

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。

请注意，非常短的条件语句（一行）在 Julia 中是会经常使用短的电路评估（Short-Circuit Evaluation）实现的，具体细节在下一节中进行概述。

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 `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 *要不就*)。

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` ）。在任何地方使用一个非布尔值，除非最后一个进入连锁条件的是一个错误：

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

另一方面，任何类型的表达式可以使用在一个条件链的末端。根据前面的条件，它将被评估和返回：

```
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` 可以直接跳出所有循环。

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
```

你可以使用下列方式定义你自己的异常：

```
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 DomainError 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，一些异常类型使用一个或更多个参数用来报告错误：

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

这个机制能被简单实现，通过按照下列所示的 UndefVarError 方法自定义异常类型：

```
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，跟在 `catch` 之后的符号会被解释为一个异常的名称，因此，需要注意的是，在单行中写 `try/catch` 表达式时。下面的代码将不会正常工作返回 `x` 的值为了防止发生错误：

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

我们在 `catch` 后使用分号或插入换行来实现：

```
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，但是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，切换到特定的 task ， 并使这个 task 的最后一次 yeidlto 返回 \特定的 value 。注意 yieldto 是唯一需要的操作来进行 ‘任务风格’的控制流;不需要调用和返回，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。

yeildto 很强大， 但是大多数时候并不直接调用它。 当你从当前的任务切换走，你有可能会想切换回来， 但需要知道切换的时机和任务，这会需要相当的协调。 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。

除此之外，为了高效地使用任务，其他一些基本的函数也同样必须。 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

大多数任务的切换都是在等待像 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，等待数据变得可用。

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，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`，使正在等待的任务变得可以运行。

调用 `Task` 可以生成一个初始对调度器还未知的任务，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，根本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	等待一个特定事件从而阻塞
:queued	在调度程序的运行队列中准备重新启动
:done	成功执行完毕
:failed	由于未处理的异常而终止

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。

多变量可以使用以下语法声明为全局：

```
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。

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。 `::` 最常见的用法是作为一个在函数/方法签名中的断言，例如 `f(x::Int8) = ...` (查看*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
```

并适用于整个当前范围，甚至在声明之前。目前，声明类型不能用于全局范围，例如在 REPL 中就不可以，因为 Julia 还没有定型的全局变量。需要注意的是在函数返回语句中，上述的前两个表达式计算值，还有就是 `::` 是一个类型的断言不是一个声明。

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href="http://zh.wikipedia.org/zh-cn/%E8%A4%87%E5%90%88%E5%9E%8B%E5%88%A5)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

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

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 :ref:`man-methods` 。

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)})
```

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)
```

这个输出告诉我们，`f` 是一个含有两个方法的函数对象。要找出这些方法的签名，可以通过使用 `methods` 函数来实现：

```
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
```

这表明，`f` 有两个方法，一个以两个 `Float64` 类型作为参数和另一个则以一个 `Number` 类型作为参数。它也指示了定义方法的文件和行数：因为这些方法在 REPL 中定义，我们得到明显行数值：`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(De
+{S,T}(A::Union(SubArray{S,N,A<:DenseArray{T,N},I<:(Union(Range{Int64},Int64)...,)},DenseArray{S,N}),B::Union(Sub
+{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::Ran
+{S<:Real,T}(A::Range{S<:Real},B::Union(SubArray{T,N,A<:DenseArray{T,N},I<:(Union(Range{Int64},Int64)...,)},Dense
+(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。

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。构造类型对象：

```
type Foo
  bar
  baz
end

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

julia> foo.bar
1

julia> foo.baz
2
```

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：

```
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
```

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*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。注意第一个参数使用的语法：参数的名称在 `::` 之前是省略的，只给出了参数的类型。这是 Julia 中对于一个函数参数，如果其类型是指定但该参数的值在函数体中从未使用过，那么语法会被使用，在这个例子中，因为参数是单态类型，就永远不会有任何理由会在函数体中使用它的值。

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)
```

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。

`import` 关键字支持与 `using` 所有相同的语法，但只能在一个时间上对一个名称进行操作。它不像 `using` 那样会添加用于搜索的模块。`import` 与 `using` 的不同之处还在于导入这一功能时必须使用新的方法扩展后的 `import` 。

在上述的 `MyModule` 中我们想向标准的 `show` 功能增加一个方法，所以我们必须写下 `import Base.show` 。

那些函数名只有通过 `using` 功能才能看到的函数是不能被扩展的。

`importall` 关键字显式地导入导出指定模块的所有名称，其效果就像 `import` 单独使用在它们的所有名称一样。

一旦一个变量是通过 `using` 或 `import` 使其可见的，一个模块就可能无法创建它自己的同名的变量了。输入变量必须是只读的；对全局变量赋值总是会影响当前模块所拥有的变量，否则就会引发错误。

模块使用方法的总结

我们要加载一个模块时，可以使用两个主要关键字： `using` 和 `import` 。要了解他们的差异，可以考虑下面的例子：

```
module MyModule

export x, y

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

end
```

在这个模块中我们（使用关键字 `export` ）导出 `x` 和 `y` 功能，也包含了非导出函数 `p` 。我们有几个不同的方法来加载该模块及其内部功能到当前工作区，具体如下：

导入命令	导入变量	方法扩展可用项
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` 。在其他模块中的宏可以被调用为 `Mod.@mac` 或 `@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。

16

元编程

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并生成自身语言的代码。元编程的另一个功能是反射，它可以在程序运行时动态展现程序本身的特性。

表达式和求值

Julia 代码表示为由 Julia 的 `Expr` 类型的数据结构而构成的语法树。下面是 `Expr` 类型的定义：

```
type Expr
  head::Symbol
  args::Array{Any,1}
  typ
end
```

`head` 是标明表达式种类的符号； `args` 是子表达式数组，它可能是求值时引用变量值的符号，也可能是嵌套的 `Expr` 对象，还可能是真实的对象值。 `typ` 域被类型推断用来做类型注释，通常可以被忽略。

有两种“引用”代码的方法，它们可以简单地构造表达式对象，而不需要显式构造 `Expr` 对象。第一种是内联表达式，使用 `:` ，后面跟单表达式；第二种是代码块儿，放在 `quote ... end` 内部。下例是第一种方法，引用一个算术表达式：

```
julia> ex = :(a+b*c+1)
:(a + b * c + 1)

julia> typeof(ex)
Expr

julia> ex.head
:call

julia> typeof(ans)
Symbol

julia> ex.args
4-element Array{Any,1}:
  :+
  :a
  :(b * c)
 1

julia> typeof(ex.args[1])
Symbol

julia> typeof(ex.args[2])
Symbol
```

```
julia> typeof(ex.args[3])
Expr

julia> typeof(ex.args[4])
Int64
```

下例是第二种方法：

```
julia> quote
        x = 1
        y = 2
        x + y
      end
quote  # none, line 2:
    x = 1 # line 3:
    y = 2 # line 4:
    x + y
end
```

符号

`:` 的参数为符号时，结果为 `Symbol` 对象，而不是 `Expr` ：

```
julia> :foo
:foo

julia> typeof(ans)
Symbol
```

在表达式的上下文中，符号用来指示对变量的读取。当表达式被求值时，符号的值受限于符号的作用域（详见*变量的作用域*）。

有时, 为了防止解析时产生歧义， `:` 的参数需要添加额外的括号：

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

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

`Symbol` 也可以使用 `symbol` 函数来创建，参数为一个字符或者字符串：

```
julia> symbol('\'')
:'
```

```
julia> symbol("'")
:'
```

求值和内插

指定一个表达式，Julia 可以使用 `eval` 函数在 global 作用域对其求值。

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

julia> eval(ans)
3

julia> ex = :(a + b)
:(a + b)

julia> eval(ex)
ERROR: a not defined

julia> a = 1; b = 2;

julia> eval(ex)
3
```

每一个组件 有在它全局范围内评估计算表达式的 `eval` 表达式。传递给 `eval` 的表达式不限于返回一个值 – 他们也会具有改变封闭模块的环境状态的副作用：

```
julia> ex = :(x = 1)
:(x = 1)

julia> x
ERROR: x not defined

julia> eval(ex)
1

julia> x
1
```

表达式仅仅是一个 `Expr` 对象，它可以通过编程构造，然后对其求值：

```
julia> a = 1;

julia> ex = Expr(:call, :+,a,:b)
```

```
:(+(1,b))

julia> a = 0; b = 2;

julia> eval(ex)
3
```

注意上例中 `a` 与 `b` 使用时的区别：

- 表达式构造时，直接使用*变量* `a` 的值。因此，对表达式求值时 `a` 的值没有任何影响：表达式中的值为 `1`，与现在 `a` 的值无关
- 表达式构造时，使用的是*符号* `:b` 。因此，构造时变量 `b` 的值是无关的—— `:b` 仅仅是个符号，此时变量 `b` 还未定义。对表达式求值时，通过查询变量 `b` 的值来解析符号 `:b` 的值

这样构造 `Expr` 对象太丑了。Julia 允许对表达式对象内插。因此上例可写为：

```
julia> a = 1;

julia> ex = :($a + b)
:(+(1,b))
```

编译器自动将这个语法翻译成上面带 `Expr` 的语法。

代码生成

Julia 使用表达式内插和求值来生成重复的代码。下例定义了一组操作三个参数的运算符： ::

```
for op = (:+, :*, :&, :|, :$)
  eval(quote
    ($op)(a,b,c) = ($op)(($op)(a,b),c)
  end)
end
```

上例可用 `:` 前缀引用格式写的更精简： ::

```
for op = (:+, :*, :&, :|, :$)
  eval(:(($op)(a,b,c) = ($op)(($op)(a,b),c)))
end
```

使用 `eval(quote(...))` 模式进行语言内的代码生成，这种方式太常见了。Julia 用宏来简写这个模式： ::

```
for op = (:+, :*, :&, :|, :$)
  @eval ($op)(a,b,c) = ($op)(($op)(a,b),c)
end
```

`@eval` 宏重写了这个调用，使得代码更精简。 `@eval` 的参数也可以是块代码：

```
@eval begin
  # multiple lines
end
```

对非引用表达式进行内插，会引发编译时错误：

```
julia> $a + b
ERROR: unsupported or misplaced expression $
```

宏

宏有点儿像编译时的表达式生成函数。正如函数会通过一组参数得到一个返回值,宏可以进行表达式的变换，这些宏允许程序员在最后的程序语法树中对表达式进行任意的转化。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")
```

上面的代码的意思是，当第一次调用表达式 `:(1==1.0)` 的时候，会被拼接为条件语句，而 `string(:(1==1.0))` 会被替换成一个断言。因此所有这些表达式构成了程序的语法树。然后在运行期间，如果表达式为真，则返回 `noth`

ing ，如果条件为假，一个提示语句将会表明这个表达式为假。注意，这里无法用函数来代替，因为在函数中只有值可以被传递，如果这么做的话我们无法在最后的错误结果中得到具体的表达式是什么样子的。

在标准库中真实的 @assert 定义要复杂一些，它可以允许用户去操作错误信息，而不只是打印出来。和函数一样宏也可以有可变参数，我们可以看下面的这个定义：

```
macro assert(ex, msgs...)
    msg_body = isempty(msgs) ? ex : msgs[1]
    msg = string("assertion failed: ", msg_body)
    return :($ex ? nothing : error($msg))
end
```

现在根据参数的接收数目我们可以把 @assert 分为两种操作模式。如果只有一个参数，表达式会被 msgs 捕获为空，并且如上面所示作为一个更简单的定义。如果用户填上第二个参数, 这个参数会被作为打印参数而不是错误的表达式。你可以在下面名为 macroexpand 的函数中检查宏扩展的结果:

```
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)
```

在实际的 @assert 宏定义中会有另一种情况：如果不仅仅是要打印 "a should equal b,"，我们还想要打印它们的值呢？有些人可能天真的想插入字符串变量如: @assert a==b "a ($a) should equal b ($b)!" ，但是这个宏不会如我们所愿的执行。你能看出是为什么吗？回顾字符串的那一章，一个字符串的重写函数，请进行比较：

```
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
```

所以现在不应该得到一个面上的字符串 `msg_body`，这个宏接收整个表达式且需要如我们所期望的计算。这可以直接拼接成返回的表达式来作为 `string` 调用的一个参数。通过看 error.jl源码得到完整的实现。

`@assert` 宏极大地通过宏替换实现了表达式的简化功能。

卫生宏

卫生宏是个更复杂的宏。一般来说，宏必须确保变量的引入不会和现有的上下文变量发送冲突。相反的，宏中的表达式作为参数应该可以和上下文代码有机的结合在一起，进行交互。另一个令人关注的问题是，当宏用不同方式定义的时候是否被应该称为另一种模式。在这种情况下，我们需要确保所有的全局变量应该被纳入正确的模式中来。Julia 已经在宏方面有了很大的优势相比其它语言（比如 C）。所有的变量（比如 `@assert` 中的 `msg`）遵循这一标准。

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

除了使用元编程语法层面的反思，朱丽亚还提供了一些其他的运行时反射能力。

类型字段 数据类型的域的名称（或模块成员）可以使用 `names` 命令来询问。例如，给定以下类型：

```
type Point
  x::FloatingPoint
  y
end
```

`names(Point)` 将会返回指针 `Any[:x, :y]` 。在一个 `Point` 中每一个域的类型都会被存储在指针对象的 `types` 域中：

```
julia> typeof(Point)
DataType
julia> Point.types
(FloatingPoint,Any)
```

亚型

任何数据类型的*直接*亚型可以使用 `subtypes(t::DataType)` 来列表查看。例如，抽象数据类型 `FloatingPoint` 包含四种（具体的）亚型：：

```
julia> subtypes(FloatingPoint)
4-element Array{Any,1}:
 BigFloat
 Float16
 Float32
 Float64
```

任何一个抽象的亚型也将被列入此列表中，但其进一步的亚型则不会；“亚型”的递归应用程序允许建立完整的类型树。

类型内部

当使用到 C 代码接口时类型的内部表示是非常重要的。`isbits(T::DataType)` 在 `T` 存储在 C 语言兼容定位时返回 true 。每一个域内的补偿量可以使用 `fieldoffsets(T::DataType)` 语句实现列表显示。

函数方法

函数内的所有方法可以通过 `methods(f::Function)` 语句列表显示出来。

函数表示

函数可以在几个表示层次上实现内部检查。一个函数的更低形式在使用 `code_lowered(f::Function, (Args...))` 时是可用的，而类型推断的更低形式在使用 `code_typed(f::Function, (Args...))` 时是可用的。

更接近机器的是，LLVM 的中间表示的函数是通过 `code_llvm(f::Function, (Args...))` 打印的，并且最终的由此产生的汇编指令在使用 `code_native(f::Function, (Args...)` 时是可用的。

17

多维数组

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` 。

因为性能的原因，Julia 不希望把程序写成向量化的形式。

在 Julia 中，通过引用将参数传递给函数。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
nnz(A)	A 中非零元素的个数
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

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	说明
Array(type, dims...)	未初始化的稠密数组
cell(dims...)	未初始化的元胞数组（异构数组）
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
falses(dims...)	全 false 的 Bool 数组
reshape(A, dims...)	将数组中的数据按照指定维度排列
copy(A)	复制 A
deepcopy(A)	复制 A ，并递归复制其元素
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，但指明了元素类型

函数	说明
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

连接

使用下列函数，可在任意维度连接数组：

函数	描述
cat(k, A...)	在 k 维度上连接输入 n-d 数组
vcat(A...)	cat(1, A...) 的简写
hcat(A...)	cat(2, A...) 的简写

传递给这些函数的标量值被视为一元阵列。

级联功能非常常用，所以为它们设计了特殊的语法：

表示	调用
[A B C ...]	hcat
[A, B, C, ...]	vcat
[A B; C D; ...]	hvcat

hvcat 可以实现一维上的（使用分号间隔）或二维上的（使用空格间隔）的级联。

Comprehensions

Comprehensions 用于构造数组。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 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> const 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
```

> 注解： 上例中，`x` 被声明为常量，因为对于非常量的全局变量，Julia 的类型推断不怎么样。

可在 comprehension 之前显式指明它的类型。如要避免在前例中声明 `x` 为常量，但仍要确保结果类型为 `Float64`，应这样写：

```
Float64[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
```

使用花括号来替代方括号，可以将它简写为 `Any` 类型的数组：

```
julia> { i/2 for i = 1:3 }
3-element Array{Any,1}:
 0.5
 1.0
 1.5
```

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. 任意整数向量，包括空向量 `[]`
4. 布尔值向量

结果 X 的维度通常为 `(length(I_1), length(I_2), ..., length(I_n))`，且 X 的索引 `(i_1, i_2, ..., i_n)` 处的值为 `A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]`。缀在后面的标量索引的维度信息被舍弃。如，`A[I, 1]` 的维度为 `(length(I),)`。布尔值向量先由 `find` 函数进行转换。由布尔值向量索引的维度长度，是向量中 `true` 值的个数。

索引语法与调用 `getindex` 等价：

```
X = getindex(A, I_1, I_2, ..., I_n)
```

例如：

```
julia> x = reshape(1:16, 4, 4)
4x4 Array{Int64,2}:
 1  5  9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[2:3, 2:end-1]
2x2 Array{Int64,2}:
 6  10
 7  11
```

`n:n-1` 形式的空范围有时用来表示相互索引位置在 `n-1` 和 `n` 之间。例如，在 `searchsorted` 函数使用本习惯指出插入点的值不在排序后的数组中：

```
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. 任意整数向量，包括空向量 `[]`
4. 布尔值向量

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
```

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：`&`，`|`，`$`

一些没有“点”（逐元素）操作运算符当一个参数是一个标量时会被使用。这些运算符有 `*`，`/`，`\` 和按位运算符。

请注意，像 `==` 操作这样的比较运算是操作在整个阵列上的，它会给出一个布尔返回值。逐位的比较使用点操作符。

下列内置的函数也都是向量化的, 即函数是逐元素版本的：

```
abs abs2 angle cbrt
airy airyai airyaiprime airybi airybiprime airyprime
acos acosh asin asinh atan atan2 atanh
acsc acsch asec asech acot acoth
cos cospi cosh sin sinpi sinh tan tanh sinc cosc
csc csch sec sech cot coth
acosd asind atand asecd acscd acotd
cosd sind tand secd cscd cotd
besselh besseli besselj besselj0 besselj1 besselk bessely bessely0 bessely1
exp erf erfc erfinv erfcinv exp2 expm1
beta dawson digamma erfcx erfi
exponent eta zeta gamma
hankelh1 hankelh2
 ceil floor round trunc
iceil ifloor iround itrunc
isfinite isinf isnan
lbeta lfact lgamma
log log10 log1p log2
copysign max min significand
sqrt hypot
```

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。

Julia 提供了 `@vectorize_1arg` 和 `@vectorize_2arg` 两个宏，分别用来向量化任意的单参数或两个参数的函数。每个宏都接收两个参数，即函数参数的类型和函数名。例如：

```
julia> square(x) = x^2
square (generic function with 1 method)

julia> @vectorize_1arg Number square
square (generic function with 4 methods)

julia> methods(square)
```

```
# 4 methods for generic function "square":
square{T<:Number}(::AbstractArray{T<:Number,1}) at operators.jl:359
square{T<:Number}(::AbstractArray{T<:Number,2}) at operators.jl:360
square{T<:Number}(::AbstractArray{T<:Number,N}) at operators.jl:362
square(x) at none:1

julia> square([1 2 4; 5 6 7])
2x3 Array{Int64,2}:
 1   4  16
 25  36  49
```

Broadcasting

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。

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` 是一个抽象的 `AbstractArray` 类型的亚型，它应该包括在内存的常规偏移上的所有数组，因此可以被传递到外部在此内存布局上的 C 和 Fortran 函数。
亚型应该提供一个方法 `stride(A,k)`，使之返回“跨越”的维度 `k`：向给出的维度 `k` 加 `1` 应该使 `getindex(A,i)` 中的 `i` 增加 `stride(A,k)`。如果提供了一个指针转换方法 `convert(Ptr{T},A)`，那么内存布局应该以相同的方式对应于这些扩展。

`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)
10x10 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 = sub(a, 2:2:8,2:2:4)
4x2 SubArray{Float64,2,Array{Float64,2},(StepRange{Int64,Int64},StepRange{Int64,Int64})}:
 0.537192  0.996234
 0.736979  0.228787
 0.991511  0.74485
 0.836126  0.0224702

julia> (q,r) = qr(b);

julia> q
4x2 Array{Float64,2}:
 −0.338809   0.78934
 −0.464815  −0.230274
 −0.625349   0.194538
 −0.527347  −0.534856

julia> r
2x2 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 的矩阵。

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 *stdlib-sparse* 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 ）

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	三对角矩阵的 LU 因子分解
UmfpackLU	稀疏矩阵的 LU 分解（使用 UMFPACK 计算）
QR	QR 分解
QRCompactWY	QR 分解的紧凑 WY 形式
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

一个 `UniformScaling` 运算符代表了一个单位算子的标量次数，`λ*I` 。单位算子 `I` 被定义为一个常量且是 `UniformScaling` 的一个实例。 这些运算符的尺寸是一般大小，可匹配 `+`，`-`，`*` 和 `\` 等其它二元运算符中的矩阵。对于 `A+I` 和 `A-I` 这意味着 `A` 必须是一个方阵. 使用了单位算子 `I` 的乘法运算是一个空操作(除非缩放因子为一)，因此基本没有开销。

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。

如果你想要通过 STDIN 去读每一行，你可以使用 eachline 方法：

```
for line in eachline(STDIN)
    print("Found $line")
end
```

或者如果你想要以字符为单位去读，则如下：

```
while !eof(STDIN)
    x = read(STDIN, Char)
    println("Found: $x")
end
```

文本 I/O

注意上面提到的写方法是对二进制流进行操作的。特别是，值不会被转换成任何规范的文本表示，而是会被写成如下：

```
julia> write(STDOUT,0x61)
a
```

对于文本 I/O，可以使用 print 或 show 方法，这取决你的需求（可以查看标准库中对于两者不同之处的细节描述）：

```
julia> print(STDOUT,0x61)
97
```

使用文件

像其他的环境一样，Julia 有一个开放的函数，它以一个文件名作为参数并且返回一个 IO 流对象，通过这个 IO 流，你可以从文件中读或者写其中的内容。举个例子来说，如果我们有一个文件，hello.txt，它的内容为 “Hello, World!”：

```
julia> f = open("hello.txt")
IOStream(<file hello.txt>)

julia> readlines(f)
1-element Array{Union(ASCIIString,UTF8String),1}:
 "Hello, World!\n"
```

如果你想要写入某些内容到文件当中，你可以用写标志 (“w”) 打开它：

```
julia> f = open("hello.txt","w")
IOStream(<file hello.txt>)

julia> write(f,"Hello again.")
12
```

如果你用这种方式检查 hello.txt 的内容，你将会注意到它是空的；其实没有任何东西被写入磁盘。这是因为 IO 流在数据真正写到磁盘之前必须被关掉：

```
julia> close(f)
```

再次检查 hello.txt 将会显示它的内容已经被改变了。

打开一个文件，对它的内容做一些改变，然后关闭它是一个常见的模式。为了让这个过程更简单，这里存在另一个 open 的调用，用一个方法作为其第一个参数，用文件名作为他的第二个参数，打开文件，调用该文件的方法作为一个参数，然后再次关闭它。举一个例子，给出一个方法：

```
function read_and_capitalize(f::IOStream)
    return uppercase(readall(f))
end
```

你可以调用：

```
julia> open(read_and_capitalize, "hello.txt")
"HELLO AGAIN."
```

为了打开 *hello.txt*，调用它的 *read_and_capitalize* 方法，关闭 *hello.txt* 然后返回大写的内容。

为了避免去定义一个已经命名的函数，你可以使用 *do* 语法，动态的去创建一个匿名函数：

```
julia> open("hello.txt") do f
         uppercase(readall(f))
       end
"HELLO AGAIN."
```

简单的 TCP 例子

让我们直接用一个简单的 Tcp Sockets 的示例来说明。我们首先需要创建一个简单地服务器：

```
julia> @async begin
        server = listen(2000)
        while true
          sock = accept(server)
          println("Hello World\n")
        end
      end
Task

julia>
```

那些熟悉 Unix socket API 的人，会觉得方法名和 Unix socket 很相似，尽管他们的用法比原生 Unix socket API 要简单。第一次调用 *listen* 将会创建一个服务器来等待即将到来的连接，在这个案例中监听的端口为 2000。相同的方法可能会用来去创建不同的其他种类的服务器：

```
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)
```

注意最后一次调用的返回值类型是不同的。这是因为这个服务器没有监听 TCP，而是在一个命名管道（Windows 术语）– 同样也称为域套接字（UNIX 的术语）。他们的不同之处非常微小，并且与他们的接收和连接方法有关系。接受方法会检索一个到客户端的连接，连接到我们刚刚创建的服务器端，而连接到服务器的函数使用的是特定的方法。连接方法和监听方法的参数是一样的，所以使用的环境（比如主机，cwd 等等）能够传递和监听方法相同的参数来建立一个连接。所以让我们来尝试一下（前提是已经创建好上面的服务器）：

```
julia> connect(2000)
TcpSocket(open, 0 bytes waiting)

julia> Hello World
```

正如我们预期的那样，我们会看到 “Hello World” 被打印出来了。所以我让我们分析一下在后台发生了什么。当我们调用连接函数时，我们连接到了我们刚刚创建的服务器。同时，接收方法返回一个服务器端的连接到最新创建的套接字上，然后打印 “Hello World” 来表明连接成功了。

Julia 的一个强大功能是尽管 I/O 实际上是异步发生的，但 API 仍然是同步的，我们甚至不必担心回调或服务器是否继续正常运行。当我们调用连接时，当前任务会等待连接建立，并且在连接建立之后，当前任务才会继续执行。在暂停期间，服务器任务会恢复执行（因为现在一个连接请求可用），接受这个连接，打印出信息并且等待下一个客户端。读和写的工作是相同的。为了更好地理解，请看以下一个简单的 echo 服务器：

```
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

一种不伴随监听方法的 *connect* 函数为 *connect(host::ASCIIString,port)*,它会尝试去连接到主机端口参数给出的端口提供的主机参数给出的主机。它允许你如下操作:

```
julia> connect("google.com",80)
TcpSocket(open, 0 bytes waiting)
```

这个功能的基础是 getaddrinfo 方法，将提供适当的地址解析:

```
julia> getaddrinfo("google.com")
IPv4(74.125.226.225)
```

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
```

`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) :ref: 等价于 <man-array-indexing> 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 :ref: 宏 <man-macros> ）

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")
```

文件也可以在多个进程启动时预加载,并且一个驱动脚本可以用于驱动计算：

```
  julia -p <n> -L file1.jl -L file2.jl driver.jl
```

每个进程都有一个关联的标识符。这个过程提供的 Julia 提示总是有一个 id 值为 1 ,就如上面例子中 julia 进程会运行驱动脚本一样。这个被默认用作平行操作的进程被称为 `workers` 。当只有一个进程的时候，进程 1 就被当做一个 worker。否则，worker 就是指除了进程 1 之外的所有进程。

Julia 内置有对于两种集群的支持：

- 如上文所示,一个本地集群指定使用 `—p` 选项。
- 一个集群生成机器使用 `--machinefile` 选项。它使用一个无密码的 `ssh` 登来在指定的机器上启动 julia 工作进程(以相同的路径作为当前主机)。

函数 `addprocs` , `rmprocs` , `workers` ,当然还有其他的在一个集群中可用的以可编程的方式进行添加,删除和查询的函数。

其他类型的集群可以通过编写自己的自定义 ClusterManager。请参阅 ClusterManagers 部分。

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` 。

与远程引用同步

调度

Julia 的平行编程平台使用任务（也成为协程），其可在多个计算中切换。每当代码执行一个通信操作，例如 `fetch` 或者 `wait`，当前任务便暂停同时调度器会选择另一个任务运行。在事件等待完成后，任务会重新启动。

对于很多问题，没必要直接考虑任务。然而，由于提供了动态调度，可以同时等待多个事件。在动态调度中，一个程序决定计算什么和在哪计算，这是基于其他工作何时完成的。这是被不可预知的或不可平衡的工作荷载所需要的，只有当他们结束当前任务我们才能分配更多的工作进程。

作为一个例子,考虑计算不同大小的矩阵的奇异值:

```
M = {rand(800,800), rand(600,600), rand(800,800), rand(600,600)}
pmap(svd, M)
```

如果一个进程要处理 800 x 800 矩阵和另一个 600 x 600 矩阵,我们不会得到很多的可伸缩性。解决方案是让本地的任务在他们完成当前的任务时去“喂”每个进程中的工作。`pmap` 的实现过程中可以看到这个:

```
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
```

只有在本地运行任务的过程中， `@async` 才与 `@spawn` 类似。我们使用它来为每个流程创建一个“供给”的任务。每个任务选择下一个需要被计算的指数,然后等待它的进程完成,接着一直重复到用完指数。注意,“供给”任务只有当主要任务到达 `@sync` 块结束时才开始执行,此时它放弃控制并等待所有的本地任务在从函数返回之前完成。供给任务可以通过 `nextidx()` 共享状态,因为它们都在相同的进程上运行。这个过程不需要锁定,因为线程是实时进行调度的而不是一成不变。这意味着内容的切换只发生在定义好的时候:在这种情况下,当 `remotecall_fetch` 会被调用。

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])
```

第二个参数指定了数组应该在处理器 1 到 4 中创建。划分含有很多进程的数据时,人们经常看到性能收益递减。把 `DArrays` 放在一个进程的子集中，该进程允许多个 `DArray` 同时计算,并且每个进程拥有更高比例的通信工作。

第三个参数指定了一个分布;数组第 n 个元素指定了应该分成多少个块。在本例中,第一个维度不会分割,而第二个维度将分为四块。因此每个局部块的大小为 `(100，25)` 。注意,分布式数组必须与进程数量相符。

`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

在这个时候,分布式数组没有太多的功能。主要功能是通过数组索引来允许进行通信，这对许多问题来说都很方便。作为一个例子,考虑实现“生活”细胞自动机,每个单元网格中的细胞根据其邻近的细胞进行更新。每个进程需要其本地块中直接相邻的细胞才能计算一个迭代的结果。下面的代码可以实现这个功能:

```
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
```

可以看到,我们使用一系列的索引表达式来获取一个本地数组中的数组 `old` 。注意, `do` 块语法方便 `init` 函数传递给 `DArray` 构造函数。接下来,连续函数 `life_rule` 被调用以提供数据的更新规则，产生所需的 `DArray` 块。 `life_rule` 与 `DArray-specific` 没有关系,但为了完整性，我们在此仍将它列出:

```
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
```

共享数组 (用于试验, 仅在 unix 上)

共享阵列使用在许多进程中共享内存来映射相同数组的系统。虽然与 DArray 有一些相似之处,但是 SharedArray 的行为是完全不同的。在一个 DArray 中,每个进程只能本地访问一块数据,并且两个进程共享同一块;相比之下,在 SharedArray 中，每个“参与”的进程能够访问整个数组。当你想要在同一台机器上大量数据共同访问两个或两个以上的进程时， SharedArray 是一个不错的选择。

SharedArray 索引(分配和访问值)与常规数组一样工作,并且是非常高效的,因为其底层内存可用于本地进程。因此,大多数算法自然地在 SharedArrays 上运行,即使在单进程模式中。当某个算法必须在一个 Array 输入的情况下,可以从 SharedArray 检索底层数组通过调用 sdata(S) 取回。对于其他 AbstractArray 类型, sdata 返回对象本身,所以在任何数组类型下使用 sdata 都是很安全的。

共享数字构造函数数的形式:

```
SharedArray(T::Type, dims::NTuple; init=false, pids=Int[])
```

创建一个被 pids 进程指定的，bitstype 为 T 并且大小为 dims 的共享数组。与分布式阵列不同,共享数组只能用于这些参与人员指定的以 pid 命名的参数(如果在同一个主机上，创建过程也同样如此)。

如果一个签名为 initfn(S::SharedArray) 的 init 函数被指定,它会被所有参与人员调用。你可以控制它，每个工人可以在数组的不同部分运行 init 函数,因此进行并行的初始化。

这里有一个简单的例子:

```
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` 提供不相交的一维索引的范围,它有时方便进程之间的任务交流。当然,你可以按你希望的方式来划分工作:

```
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
```

因为所有进程都可以访问底层数据,你必须小心不要设置冲突。例如:

```
@sync begin
    for p in workers()
        @async begin
            remotecall_wait(p, fill!, S, p)
        end
    end
end
```

这有可能导致未定义的行为:因为每个进程有他自己的 `pid` 来充满整个数组,无论最后执行的是哪一个进程(任何特定元素 `S`)都将保留他的 `pid` 。

ClusterManagers

Julia 工作进程也可以在任意机器中产生,让 Julia 的自然并行功能非常透明地在集群环境中运行。`ClusterManager` 接口提供了一种方法来指定启动和管理工作进程的手段。例如, `ssh` 集群也使用 `ClusterManager` 来实现:

```
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
```

`launch_ssh_workers` 负责实例化新的 Julia 进程并且 `manage_ssh_workers` 提供了一种方法来管理这些进程,例如发送中断信号。在运行时可以使用 `addprocs` 添加新进程:

```
addprocs(5, cman=LocalManager())
```

来指定添加一批进程并且 `ClusterManager` 用于启动这些进程。

脚注

[1]:在这边文中, MPI 是指 MPI-1 标准。从 MPI-2 开始,MPI 标准委员会引入了一系列新的通信机制,统称为远程内存访问 (RMA) 。添加 RMA MPI 标准的动机是改善单方面的沟通模式。最新的 MPI 标准的更多信息,参见 http://www.mpi-forum.org/docs。

日期和时间

`Dates` 模块提供了两种关于时间的数据类型: `Date` 和 `DateTime`, 精度分别为天和毫秒, 都是抽象数据类型 `TimeType` 的子类型. 使用两种数据类型的原因很简单: 某些操作本身很简单, 无论是从代码上看还是逻辑上, 使用高精度的数据类型是完全没有必要的. 例如, `Date` 只精确到天 (也就是说, 没有小时, 分钟或者秒), 所以使用时就不需要考虑时区, 夏令时和闰秒.

`Date` 和 `DateTime` 都不过是 `Int64` 的简单封装, 仅有的一个成员变量 `instant` 实际上的类型是 `UTInstant{P}`, 代表的是基于世界时的机器时间 [1]. `Datetime` 类型是 *不考虑时区* 的 (根据 Python 的讲法), 或者说是 Java 8 里面的 *本地时间*. 额外的时间日期操作可以通过 Timezones.jl 扩展包来获取, 其中的数据来自 Olsen Time Zone Database . `Date` 和 `DateTime` 遵循 ISO 8601 标准. 值得注意的一点是, ISO 8601 关于公元前日期的处理比较特殊. 简单来说, 公元前的最后一天是公元前 1-12-31, 接下来第二天是公元 1-1-1, 所以是没有公元 0 年存在的. 而 ISO 标准认定, 公元前 1 年是 0 年, 所以 `0000-12-21` 是 `0001-01-01` 的前一天, `-0001` 是公元前 2 年, `-0003` 是公元前 3 年, 等等.

[1] 一般来说有两种常用的时间表示法, 一种是基于地球的自转状态 (地球转一整圈 = 1 天), 另一种基于 SI 秒 (固定的常量). 这两种表示方法是不一样的. 试想一下, 因为地球自转, 基于世界时的的秒可能是不等长的. 但总得来说, 基于世界时的 `Date` 和 `DateTime` 是一种简化的方案, 例如闰秒的情况不需要考虑. 这种表示时间的方案的正式名称为世界时 . 这意味着, 每一分钟有 60 秒, 每一天有 60 小时, 这样使得关于时间的计算更自然, 简单.

构造函数

`Date` 和 `DateType` 可以通过整数或者 `Period` 构造, 通过直接传入, 或者作为与特定时间的差值:

```
julia> DateTime(2013)
2013-01-01T00:00:00

julia> DateTime(2013,7)
2013-07-01T00:00:00

julia> DateTime(2013,7,1)
2013-07-01T00:00:00

julia> DateTime(2013,7,1,12)
2013-07-01T12:00:00

julia> DateTime(2013,7,1,12,30)
2013-07-01T12:30:00

julia> DateTime(2013,7,1,12,30,59)
2013-07-01T12:30:59

julia> DateTime(2013,7,1,12,30,59,1)
2013-07-01T12:30:59.001

julia> Date(2013)
2013-01-01

julia> Date(2013,7)
2013-07-01

julia> Date(2013,7,1)
2013-07-01

julia> Date(Dates.Year(2013),Dates.Month(7),Dates.Day(1))
2013-07-01

julia> Date(Dates.Month(7),Dates.Year(2013))
2013-07-01
```

`Date` 和 `DateTime` 解析是通过格式化的字符串实现的. 格式化的字符串是指 *分隔* 的或者 *固定宽度* 的 "字符段" 来表示一段时间, 然后传递给 `Date` 或者 `DateTime` 的构造函数.

使用分隔的字符段方法, 需要显示指明分隔符, 所以 `"y-m-d"` 告诉解析器第一个和第二个字符段中间有一个 `-`, 例如 `"2014-07-16"`, `y`, `m` 和 `d` 字符告诉解析器每个字符段的含义.

固定宽度字符段是使用固定宽度的字符串来表示时间. 所以 `"yyyymmdd"` 相对应的时间字符串为 `"20140716"`.

同时字符表示的月份也可以被解析, 通过使用 `u` 和 `U`, 分别是月份的简称和全称. 默认支持英文的月份名称, 所以 `u` 对应于 `Jan`, `Feb`, `Mar` 等等, `U` 对应于 `January`, `February`, `March` 等等. 然而, 同 `dayname` 和 `monthname` 一样, 本地化的输出也可以实现, 通过向 `Dates.MONTHTOVALUEABBR` 和 `Dates.MONTHTOVALUE` 字典添加 `locale=>Dict{UTF8String, Int}` 类型的映射.

更多的解析和格式化的例子可以参考 tests/dates/io.jl .

时间间隔/比较

计算两个 `Date` 或者 `DateTime` 之间的间隔是很直观的, 考虑到他们不过是 `UTInstant{Day}` 和 `UTInstant{Millisecond}` 的简单封装. 不同点是, 计算两个 `Date` 的时间间隔, 返回的是 `Day` , 而计算 `DateTime` 时间间隔返回的是 `Millisecond` . 同样的, 比较两个 `TimeType` 本质上是比较两个 `Int64`

```
julia> dt = Date(2012,2,29)
2012-02-29

julia> dt2 = Date(2000,2,1)
2000-02-01

julia> dump(dt)
Date
  instant: UTInstant{Day}
    periods: Day
      value: Int64 734562

julia> dump(dt2)
Date
instant: UTInstant{Day}
  periods: Day
    value: Int64 730151

julia> dt > dt2
true

julia> dt != dt2
true

julia> dt + dt2
Operation not defined for TimeTypes

julia> dt * dt2
Operation not defined for TimeTypes

julia> dt / dt2
Operation not defined for TimeTypes

julia> dt - dt2
4411 days
```

```
julia> dt2 - dt
-4411 days

julia> dt = DateTime(2012,2,29)
2012-02-29T00:00:00

julia> dt2 = DateTime(2000,2,1)
2000-02-01T00:00:00

julia> dt - dt2
381110402000 milliseconds
```

访问函数

因为 `Date` 和 `DateTime` 类型是使用 `Int64` 的封装, 具体的某一部分可以通过访问函数来获得. 小写字母的获取函数返回值为整数:

```
julia> t = Date(2014,1,31)
2014-01-31

julia> Dates.year(t)
2014

julia> Dates.month(t)
1

julia> Dates.week(t)
5

julia> Dates.day(t)
31
```

大写字母的获取函数返回值为 `Period` :

```
julia> Dates.Year(t)
2014 years

julia> Dates.Day(t)
31 days
```

如果需要一次性获取多个字段, 可以使用符合函数:

```
julia> Dates.yearmonth(t)
(2014,1)

julia> Dates.monthday(t)
(1,31)

julia> Dates.yearmonthday(t)
(2014,1,31)
```

也可以直接获取底层的 `UTInstant` 或 整数数值 :

```
julia> dump(t)
Date
```

```
instant: UTInstant{Day}
  periods: Day
  value: Int64 735264

julia> t.instant
UTInstant{Day}(735264 days)

julia> Dates.value(t)
735264
```

查询函数

查询函数可以用来获得关于 `TimeType` 的额外信息, 例如某个日期是星期几:

```
julia> t = Date(2014,1,31)
 2014-01-31

 julia> Dates.dayofweek(t)
 5

 julia> Dates.dayname(t)
 "Friday"

 julia> Dates.dayofweekofmonth(t)
 5  # 5th Friday of January
```

月份信息：

```
julia> Dates.monthname(t)
 "January"

 julia> Dates.daysinmonth(t)
 31
```

年份信息和季节信息：

```
julia> Dates.isleapyear(t)
 false

 julia> Dates.dayofyear(t)
 31

 julia> Dates.quarterofyear(t)
 1

 julia> Dates.dayofquarter(t)
 31
```

`dayname` 和 `monthname` 可以传入可选参数 `locale` 来显示

```
 julia> const french_daysofweek =
 [1=>"Lundi",2=>"Mardi",3=>"Mercredi",4=>"Jeudi",5=>"Vendredi",6=>"Samedi",7=>"Dimanche"];

 # Load the mapping into the Dates module under locale name "french"
```

```
julia> Dates.VALUETODAYOFWEEK["french"] = french_daysofweek;

julia> Dates.dayname(t;locale="french")
"Vendredi"
```

`monthname` 与之类似的, 这时, `Dates.VALUETOMONTH` 需要加载 `locale=>Dict{Int, UTF8String}`.

时间间隔算术运算

在使用任何一门编程语言/时间日期框架前, 最好了解下时间间隔是怎么处理的, 因为有些地方需要特殊的技巧.

`Dates` 模块的工作方式是这样的, 在做 `period` 算术运算时, 每次都做尽量小的改动. 这种方式被称之为 *日历* 算术, 或者就是平时日常交流中惯用的方式. 这些到底是什么? 举个经典的例子: 2014 年 1 月 31 号加一月. 答案是什么? JavaScript 会得出 3月3号 (假设31天). PHP 会得到 `3月2号 <http://stackoverflow.com/questions/5760262/php-adding-months-to-a-date-while-not-exceeding-the-last-day-of-the-month> _` (假设30天). 事实上, 这个问题没有正确答案. `Dates` 模块会给出 2月28号的答案. 它是怎么得出的? 试想下赌场的 7-7-7 赌博游戏.

设想下, 赌博机的槽不是 7-7-7, 而是年-月-日, 或者在我们的例子中, 2014-01-31. 当你想要在这个日期上增加一个月时, 对应于月份的那个槽会增加1, 所以现在是 2014-02-31, 然后检查年-月-日中的日是否超过了这个月最大的合法的数字 (28). 这种方法有什么后果呢? 我们继续加上一个月, `2014-02-28 + Month(1) == 2014-03-28` . 什么? 你是不是期望结果是3月的最后一天? 抱歉, 不是的, 想一下 7-7-7. 因为要改变尽量少的槽, 所以我们在月份上加1, 2014-03-28, 然后就没有然后了, 因为这是个合法的日期. 然而, 如果我们在原来的日期(2014-01-31)上加上2个月, 我们会得到预想中的 2014-03-31. 这种方式带来的另一个问题是损失了可交换性, 如果强制加法的顺序的话 (也就是说,用不用的顺序相加会得到不同的结果). 例如 ::

```
julia> (Date(2014,1,29)+Dates.Day(1)) + Dates.Month(1)
2014-02-28

julia> (Date(2014,1,29)+Dates.Month(1)) + Dates.Day(1)
2014-03-01
```

这是怎么回事? 第一个例子中, 我们往1月29号加上一天, 得到 2014-01-30; 然后加上一月, 得到 2014-02-30, 然后被调整到 2014-02-28. 在第二个例子中, 我们 *先* 加一个月, 得到 2014-02-29, 然后被调整到 2014-02-28, *然后* 加一天, 得到 2014-03-01. 在处理这种问题时的一个设计原则是, 如果有多个时间间隔, 操作的顺序是按照间隔的 *类型* 排列的, 而不是按照他们的值大小或者出现顺序; 这就是说, 第一个加的是 `Year` , 然后是 `Month` , 然后是 `Week` , 等等. 所以下面的例子 *是* 符合可交换性的 ::

```
julia> Date(2014,1,29) + Dates.Day(1) + Dates.Month(1)
2014-03-01

julia> Date(2014,1,29) + Dates.Month(1) + Dates.Day(1)
2014-03-01
```

很麻烦? 也许吧. 一个 `Dates` 的初级用户该怎么办呢? 最基本的是要清楚, 当操作月份时, 如果强制指明操作的顺序, 可能会产生意想不到的结果, 其他的就没什么了. 幸运的是, 这基本就是所有的特殊情况了 (UT 时间已经免除了夏令时, 闰秒之类的麻烦).

调整函数

时间间隔的算术运算是很方便, 但同时, 有些时间的操作是基于 *日历* 或者 *时间* 本身的, 而不是一个固定的时间间隔. 例如假期的计算, 诸如 "纪念日 = 五月的最后一个周一", 或者 "感恩节 = 十一月的第四个周四". 这些时间的计算牵涉到基于日历的规则, 例如某个月的第一天或者最后一天, 下一个周四, 或者第一个和第三个周三, 等等.

`Dates` 模块提供几个了 *调整* 函数, 这样可以简单简洁的描述时间规则. 第一组是关于周, 月, 季度, 年的第一和最后一个元素. 函数参数为 `TimeType`, 然后按照规则返回或者 *调整* 到正确的日期。

```
# 调整时间到相应的周一
julia> Dates.firstdayofweek(Date(2014,7,16))
2014-07-14

# 调整时间到这个月的最后一天
julia> Dates.lastdayofmonth(Date(2014,7,16))
2014-07-31

# 调整时间到这个季度的最后一天
julia> Dates.lastdayofquarter(Date(2014,7,16))
2014-09-30
```

接下来一组高阶函数, `tofirst`, `tolast`, `tonext`, and `toprev`, 第一个参数为 `DateFunction`, 第二个参数 `TimeType` 作为起点日期. 一个 `DateFunction` 类型的变量是一个函数, 通常是匿名函数, 这个函数接受 `TimeType` 作为输入, 返回 `Bool`, `true` 来表示是否满足特定的条件. 例如 ::

```
julia> istuesday = x->Dates.dayofweek(x) == Dates.Tuesday  # 如果是周二, 返回 true
(anonymous function)

julia> Dates.tonext(istuesday, Date(2014,7,13)) # 2014-07-13 is a 是周日
2014-07-15

# 同时也额外提供了一些函数, 使得对星期几之类的操作更加方便
julia> Dates.tonext(Date(2014,7,13), Dates.Tuesday)
2014-07-15
```

如果是复杂的时间表达式, 使用 `do-block` 会很方便:

```
julia> Dates.tonext(Date(2014,7,13)) do x
       # 如果是十一月的第四个星期四, 返回 true (感恩节)
       Dates.dayofweek(x) == Dates.Thursday &&
       Dates.dayofweekofmonth(x) == 4 &&
       Dates.month(x) == Dates.November
```

```
    end
2014-11-27
```

类似的, `tofirst` 和 `tolast` 第一个参数为 `DateFunction`, 但是默认的调整范围位当月, 或者可以用关键字参数指明调整范围为当年 :

```
julia> Dates.tofirst(istuesday, Date(2014,7,13)) # 默认位当月
2014-07-01

julia> Dates.tofirst(istuesday, Date(2014,7,13); of=Dates.Year)
2014-01-07

julia> Dates.tolast(istuesday, Date(2014,7,13))
2014-07-29

julia> Dates.tolast(istuesday, Date(2014,7,13); of=Dates.Year)
2014-12-30
```

最后一个函数为 `recur`. `recur` 函数是向量化的调整过程, 输入为起始和结束日期 (或者指明 `StepRange`), 加上一个 `DateFunction` 来判断某个日期是否应该返回. 这种情况下, `DateFunction` 又被经常称为 "包括" 函数, 因为它指明了 (通过返回 true) 某个日期是否应该出现在返回的日期数组中。

```
# 匹兹堡大街清理日期; 从四月份到十一月份每月的第二个星期二
 # 时间范围从2014年1月1号到2015年1月1号
 julia> dr = Dates.Date(2014):Dates.Date(2015);
 julia> recur(dr) do x
          Dates.dayofweek(x) == Dates.Tue &&
          Dates.April <= Dates.month(x) <= Dates.Nov &&
          Dates.dayofweekofmonth(x) == 2
        end
 8-element Array{Date,1}:
  2014-04-08
  2014-05-13
  2014-06-10
  2014-07-08
  2014-08-12
  2014-09-09
  2014-10-14
  2014-11-11
```

更多的例子和测试可以参考 test/dates/adjusters.jl .

时间间隔

时间间隔是从人的角度考虑的一段时间, 有时是不规则的. 想下一个月; 如果从天数上讲, 不同情况下, 它可能代表 28, 29, 30, 或者 31. 或者一年可以代表 365 或者 366 天. `Period` 类型是 `Int64` 类型的简单封装, 可以通过任何可以转换成 `Int64` 类型的数据构造出来, 比如 `Year(1)` 或者 `Month(3.0)`. 相同类型的时间间隔的行为类似于整数 :

```
julia> y1 = Dates.Year(1)
1 year

julia> y2 = Dates.Year(2)
2 years

julia> y3 = Dates.Year(10)
10 years

julia> y1 + y2
3 years

julia> div(y3,y2)
5 years

julia> y3 - y2
8 years

julia> y3 * y2
20 years

julia> y3 % y2
0 years

julia> y1 + 20
21 years

julia> div(y3,3) # 类似于整数除法
3 years
```

另加详细的信息可以参考 :mod:`Dates` 模块的 API 索引.

22

可空类型

在很多情况下, 你可能会碰到一些可能存在也可能不存在的变量. 为了处理这种 情况, Julia 提供了参数化的数据类型 `Nullable{T}` , 可以被当做是一种特 殊的容器, 里面有 0 个或 1 个数据. `Nullable{T}` 提供了最小的接口以保 证对可能是空值的操作是安全的. 目前包含四种操作 :

- 创建一个 `Nullable` 对象.
- 检查 `Nullable` 是否含有数据.
- 获取 `Nullable` 内部的数据, 如果没有数据可被返回, 抛出 `NullException` .
- 获取 `Nullable` 内部的数据, 如果没有数据可被返回, 返回数据类型 `T` 的默认值.

创建 Nullable 对象

使用 `Nullable{T}()` 函数来创建一个关于类型 `T` 的可空对象：

```
x1 = Nullable{Int}()
x2 = Nullable{Float64}()
x3 = Nullable{Vector{Int}}()
```

使用 `Nullable(x::T)` 函数来创建一个非空的关于类型 `T` 的可空对象：

```
x1 = Nullable(1)
x2 = NUllable(1.0)
x3 = Nullalbe([1, 2, 3])
```

注意上面两种构造可空对象方式的不同: 对第一种方式, 函数接受的参数是类型 `T`; 另一种方式中, 函数接受的是单个参数, 这个参数的类型是 `T`.

检查 Nullabe 对象是否含有数据

使用 isnull 函数来检查 Nullable 对象是否为空：

```
isnull(Nullable{Float64}())
isnull(Nullable(0.0))
```

安全地访问 Nullable 对象的内部数据

使用 get 来安全地访问 Nullable 对象的内部数据：

```
get(Nullable{Float64}())
get(Nullable(1.0))
```

如果没有数据, 正如 Nullable{Float64}, 抛出 NullException 错误. get 函数保证了任何访问不存在的数据的操作立即抛出错误。

在某些情况下, 如果 Nullable 对象是空的, 我们希望返回一个合理的默认值。我们可以将这个默认值穿递给 get 函数作为第二个参数：

```
get(Nullable{Float64}(), 0)
get(Nullable(1.0), 0)
```

注意, 这个默认的参数会被自动转换成类型 T。例如, 上面的例子中, 在 get 函数返回前, 0 会被自动转换成 Float64. get 函数可以 设置默认替换值这一特性使得处理未定义变量变得非常轻松.

23

交互

Julia 有一个全功能的交互式命令行 REPL（read-eval-print 循环）内置在可执行的 `julia` 内。除了允许快速并且简易的评定 Julia 语句，他还有一个可搜索历史的功能，tab 补齐功能，以及更多有用的快捷键，和专门的帮助，并且还有 shell 模式。REPL 能够通过简单的无参数调用或双击执行来进行启动:

```
$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "help()" to list help topics
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.3.0-prerelease+2834 (2014-04-30 03:13 UTC)
 _/ |\__'_|_|_|\__'_|  |  Commit 64f437b (0 days old master)
|__/                   |  x86_64-apple-darwin13.1.0

julia>
```

如果要退出互动会话，敲击 `^D` 即 control 键加上 d 键 - 或者是编辑 `quit()`，然后在敲击回车键。REPL 会给你 `julia>` 提示。

不同的提示模式

Julia 模式

REPL 有四种主要的操作模式。第一种并且最常见的一种是 Julian 提示。它是默认的操作模式；每一行的开始都会是 `julia>` 。在这里，你可以输入 Julia 表达式。在一个完整的表达式已经输入好之后敲打回车将会评估该条目并且显示最后一个表达式的结果。

```
julia> string(1 + 2)
"3"
```

这里有许多独特的特点可以来进行交互工作。除了显示结果外，REPL 同样将结果绑定到变量 `ans` 。 在该行尾部的分号可以用作一个标志来抑制显示结果。

```
julia> string(3 * 4);

julia> ans
"12"
```

帮助模式

当指针在一行的开始位置时，敲击 `?` 提示将会变为帮助模式。Julia 将会尝试打印在帮助模式中的帮助或是文档：

```
julia> ? # upon typing ?, the prompt changes (in place) to: help>

help> string
Base.string(xs...)

  Create a string from any values using the "print" function.
```

除了方法名，完成方法调用可以看到哪一个方法被指定的参数调用了。宏,类型和变量也可以查询。

```
help> string(1)
string(x::Union(Int16,Int128,Int8,Int32,Int64)) at string.jl:1553

help> @printf
Base.@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.

help> String
DataType   : String
  supertype: Any
  subtypes : {DirectIndexString,GenericString,RepString,RevString{T<:String},RopeString,SubString{T<:String},UTF16S
```

想要退出帮助模式可以在一行的开始按下退格键。

Shell 模式

帮助模式适用于快速访问文档，另一个常见任务是使用系统 Shell 来执行系统命令。就像当光标在一行的开始位置时 `?` 来进入帮助模式，使用分号 `(;)` 可以进入 shell 模式。并且想要退出模式时可以在一行的开始按下退格键。

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>

shell> echo hello
hello
```

查找模式

在所有以上的方法中，所有执行行会被保存到历史文件当中，并且能够被查找。为了初始化一个对先前历史的增量搜索，敲击 `^R` – control 键加上键盘上的 `r` 键。提示会被更改为 `(reverse-i-search)':` ，并且随着你敲击，查询请求将会出现在引用中。达到匹配要求的最近一次的结果将会被动态更新在右面的控制台中。想要找到更老的结果就使用相同查询，然后再敲击一次 `^R` 。

`^R` 是反向查询，而 `^S` 是正向查询，提示为 `(i-search)':` 。这两个可以相互结合使用来相对的移动到前一个或是后一个匹配结果。

键绑定

Julia REPL 很好的使用了键绑定功能。上面已经介绍了很多种控制键绑定（ `^D` 用来退出，`^R` 和 `^S` 用来查询），但是这里还有更多的键绑定。除了控制键，这里还有很多 meta - 键绑定。这些键绑定因平台的不同而不同，但是大部分终端默认使用 alt - 或 option - 选一个键来发送 meta - 键（或是通过配置）。

Program control	
^D	退出（当缓冲区为空）
^C	中断或是取消
Return/Enter, ^J	新的一行并且如果上一行已经完成则执行上一行
meta-Return/Enter	新的一行并且不执行
? or ;	进入帮助或是 Shell 模式（在一行的起始位置）
^R, ^S	增量历史搜索
Cursor movement	
Right arrow, ^F	向右移动一个字符
Left arrow, ^B	向左移动一个字符
Home, ^A	移动到该行的起始
End, ^E	移动到该行的末尾
^P	改变先前或下一个历史条目
^N	改变到下一个历史条目
Up arrow	移动到上面一行（或是先前的历史条目）
Down arrow	移动到下面一行（或是之后的历史条目）
Page-up	切换到上一条光标前的文本匹配的历史条目
Page-down	切换到下一条光标前的文本匹配的历史条目
meta-F	向右移动一个词
meta-B	向左移动一个词
Editing	
Backspace, ^H	删除前一个字符
Delete, ^D	向后删除一个字符（当缓冲区有文本时）
meta-Backspace	删除前一个词
meta-D	向后删除一个词
^W	删除先前的直到最近的空白的所有文本
^K	"杀死"到行的末尾，将文本放至缓冲区
^Y	从 kill 缓冲区插入文本
^T	根据光标调换字符
Delete, ^D	向后删除一个字符（当缓冲区内有文本）

自定义快捷键

Julia REPL 的快捷键可以通过向 `REPL.setup_interface()` 传入字典类型的数据来实现自定义. 字典的关键字可以是字符, 也可以是字符串. 字符 `*` 代表默认默认操作. `^x` 代表快捷键 Control 键加 `x` 键. Meta 键加 `x` 键可以写作 `"\\Mx"` . 字典的数据必须是 `nothing` (代表忽略该操作), 或者参数为 `(PromptState, AbstractREPL, Char` 的函数. 例如, 为了实现绑定上下键到搜索历史记录, 可以把下面的代码加入到 `.juliarc.jl` :

```
import Base: LineEdit, REPL

 const mykeys = {
   # Up Arrow
   "\e[A" => (s,o...)->(LineEdit.edit_move_up(s) || LineEdit.history_prev(s, LineEdit.mode(s).hist)),
   # Down Arrow
   "\e[B" => (s,o...)->(LineEdit.edit_move_up(s) || LineEdit.history_next(s, LineEdit.mode(s).hist))
 }

 Base.active_repl.interface = REPL.setup_interface(Base.active_repl; extra_repl_keymap = mykeys)
```

可供使用的按键和操作请参阅 `base/LineEdit.jl` .

Tab 补全

在 Julia REPL (或者帮助模式下的 REPL), 可以输入函数或者类型名的前几个字符, 然后按 Tab 键来显示可能的选项::

```
julia> stri
stride    strides    string     stringmime  strip

julia> Stri
StridedArray    StridedVecOrMat  String
StridedMatrix   StridedVector
```

Tab 键也可以使 LaTeX 数学字符替换成 Unicode 并且显示可能的选项::

```
julia> \pi[TAB]
julia> π
π = 3.1415926535897...

julia> e\_1[TAB] = [1,0]
julia> e₁ = [1,0]
2-element Array{Int64,1}:
 1
 0

julia> e\^1[TAB] = [1 0]
julia> e¹ = [1 0]
1x2 Array{Int64,2}:
 1  0

julia> \sqrt[TAB]2     # √ is equivalent to the sqrt() function
julia> √2
1.4142135623730951

julia> \hbar[TAB](h) = h / 2\pi[TAB]
julia> ħ(h) = h / 2π
ħ (generic function with 1 method)

julia> \h[TAB]
\hat            \heartsuit        \hksearow         \hookleftarrow    \hslash
\hbar           \hermitconjmatrix  \hkswarow         \hookrightarrow   \hspace
```

24

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
```

更普遍的，你可以使用 `open` 从一个外部命令读取或者写到一个外部命令。例如：

```
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 :ref: `man-strings` ）：

```
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。

25

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 C lang ）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

unsigned char	Cuchar	Uint8
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* (T 正确表示一个定义好的 bit 类型)		Ptr{T} (在参数列表中使用 &variable_name 调用)
struct T (T 正确表示一个定义好的 bit 类型)		T (在参数列表中使用 &variable_name 调用)
jl_value_t* (任何 Julia 类型)		Ptr{Any}

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)
```

对于 `wchar_t*` 参数，Julia 类型为 `Ptr{Wchar_t}` ,并且数据可以通过 `wstring(s)` 方法转换为原始的 Julia 字符串(等同于 `utf16(s)` 或 `utf32(s)` ,这取决于 `Cwchar_t` 的宽度)。还要注意 ASCII, UTF-8, UTF-16, 和UTF-32 字符串数据在 Julia 内部是以 NUL 结尾的，所以它能够传递到 C 函数中以 NUL 为结尾的数据，而不用再做一个拷贝。

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

访问全局变量

当全局变量导出到本地库时可以使用 `cglobal` 方法，通过名称进行访问。 `cglobal` 的参数和 `ccall` 的指定参数是相同的符号，并且其表述了存储在变量中的值类型：

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

该结果是一个该值的地址的指针。可以通过这个指针对这个值进行操作，但需要使用 unsafe_load 和 unsafe_store 。

将 Julia 的回调函数传递给 C

可以将 Julia 函数传递给本地的函数，只要该函数有指针参数。一个典型的例子为标准 C 库 qsort 函数，描述如下：

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

base 参数是一个数组长度 nmemb 的指针，每个元素大小为 size 字节。compare 是一个回调函数，带有两个元素 a 和 b 的指针，并且如果 a 在 b 之前或之后出现，则返回一个大于或者小于 0 的整数（如果允许任意顺序的话，结果为 0 ）。现在假设我们在 Julia 值中有一个一维数组 A ，我们想给这个数组进行排序，使用 qsort 函数（不用 Julia 的内置函数）。在我们调用 qsort 和传递参数之前，我们需要写一个比较函数，来适应任意类型 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
```

请注意，我们必须注意返回值类型： qsort 需要的是 C 语言的 int 类型变量作为返回值，所以我们必须通过调用 convert 来确保返回 Cint 。

为了能够传递这个函数给 C，我们要通过 cfunction 来得到它的地址:

```
const mycompare_c = cfunction(mycompare, Cint, (Ptr{Cdouble}, Ptr{Cdouble}))
```

cfunction 接受三个参数：Julia 函数（ mycompare ），返回值类型（ Cint ）,和一个参数类型的元组，在这种情况下对 cdouble （ Float64 ）元素 的数组进行排序。

最终对 qsort 的调用如下：

```
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)
```

执行该操作之后， `A` 会更改为排序数组 `[-2.7, 1.3, 3.1, 4.4]` 。注意 Julia 知道如何去将数组转换为 `Ptr{Cdouble}` ,如何计算字节大小（与 C 的 `sizeof` 是相同的）等等。如果你有兴趣，你可以尝试在 `mycompare` 插入一个 `println("mycompare($a,$b)")` ，这将允许你以比较的方式去查看 `qsort` （并且确认它的确调用了 你传递的 Julia 函数）。

线程安全

一些 C 从不同的线程中执行他们的回调函数，并且 Julia 不含有线程安全，你需要做一些额外的预防措施。特别是，你需要设置两层系统：C 的回调应该只调度（通过 Julia 的时间循环）你“真正”的回调函数的执行。你的回调需要两个输入（你很可能会忘记）并且通过 `SingleAsyncWork` 进行包装::

```
cb = Base.SingleAsyncWork(data -> my_real_callback(args))
```

你传递给 C 的回调应该仅仅执行 `ccall` 到 `:uv_async_send` ,传递 `cb.handle` 作为参数。

关于回调更多的内容

对于更多的如何传递回调到 C 库的细节，请参考 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)
```

26

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。

第二个语句初始化了 Julia 的任务调度系统。这条语句必须在一个不返回的函数中出现，只要 Julia 被调用（ `main` 运行得很好 ）。严格地讲，这条语句是可以选择的，但是转换任务的操作将引起问题，如果它被省略的话。

测试程序中的第三条语句使用 `jl_eval_string` 的调用评估了 Julia 语句。

类型转换

真正的应用程序不仅仅需要执行表达式，而且返回主程序的值。`jl_eval_string` 返回 `jl_value_t`，它是一个指向堆上分配的 Julia 对象的指针。用这种方式存储简单的数据类型比如 `Float64` 叫做 `boxing`，提取存储的原始数据叫做 `unboxing`。我们提升过的用 Julia 计算 2 的平方根和用 C 语言读取结果的样本程序如下所示：

```
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);
}
```

为了检查 `ret` 是否是一个指定的 Julia 类型，我们可以使用 `jl_is_...` 函数。通过将 `typeof(sqrt(2.0))` 输入进 Julia shell，我们可以看到返回类型为 `Float64`（C 中的 double）。为了将装好的 Julia 的值转换成 C 语言中的 double, `jl_unbox_float64` 功能在上面的代码片段中被使用。

相应的 `jl_box_...` 功能被用来用另一种方式转换：

```
jl_value_t *a = jl_box_float64(3.0);
jl_value_t *b = jl_box_float32(3.0f);
jl_value_t *c = jl_box_int32(3);
```

正如我们下面将看到的，调用带有指定参数的 Julia 函数装箱(boxing)是需要的。

调用 Julia 的函数

当 `jl_eval_string` 允许 C 语言来获得 Julia 表达式的结果时，它不允许传递在 C 中计算的参数到 Julia 中。对于这个，你需要直接调用 Julia 函数，使用 `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);
```

在第一步中，Julia 函数 `sqrt` 的处理通过调用 `jl_get_function` 检索。第一个传递给 `jl_get_function` 的参数是一个指向 `Base` 模块的指针，在那里 `sqrt` 被定义。然后，double 值使用 `jl_box_float64` 封装。最后，在最后一步中，函数使用 `jl_call1` 被调用。`jl_call0`，`jl_call2` 和 `jl_call3` 函数也存在，来方便地处理不同参数的数量。为了传递更多的参数，使用 `jl_call`：

```
jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t nargs)
```

第二个参数 `args` 是一个 `jl_value_t*` 参数的数组而且 `nargs` 是参数的数字。

内存管理

正如我们已经看见的，Julia 对象作为指针在 C 中呈现。这引出了一个问题，谁应该释放这些对象。

通常情况下，Julia 对象通过一个 garbage collector(GC) 来释放，但是 GC 不会自动地知道我们在 C 中有一个对 Julia 值的引用。这意味着 GC 可以释放指针，使指针无效。

GC 仅能在 Julia 对象被分配时运行。像 `jl_box_float64` 的调用运行分配，而且分配也能在任何运行 Julia 代码的指针中发生。在 `jl_...` 调用间使用指针通常是安全的。但是为了确认值能使 `jl_...` 调用生存，我们不得不告诉 Julia 我们有一个对 Julia 值的引用。这可以使用 `JL_GC_PUSH` 宏指令完成。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();
```

`JL_GC_POP` 调用释放了之前 `JL_GC_PUSH` 建立的引用。注意到 `JL_GC_PUSH` 在栈上工作，所以它在栈帧被销毁之前必须准确地和 `JL_GC_POP` 成组。

几个 Julia 值能使用 `JL_GC_PUSH2`， `JL_GC_PUSH3`，和 `JL_GC_PUSH4` 宏指令被立刻 push。为了 push 一个 Julia 值的数组我们可以使用 `JL_GC_PUSHARGS` 宏指令，它能向以下那样使用：cro, 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

有一些函数来控制 GC。在普通的使用案例中，这些不应该是必需的。

`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 和 C 能不用复制而分享数组数据。下一个例子将展示这是如此工作的。

Julia 数组通过数据类型 `jl_array_t*` 用 C 语言显示。基本上，`jl_array_t` 是一个包含以下的结构：

- 有关数据类型的信息
- 指向数据块的指针
- 有关数组大小的信息

为了使事情简单，我们用一个 1D 数组开始。创建一个包含长度为 10 个元素的 Float64 数组通过以下完成：

```
jl_value_t* array_type = jl_apply_array_type(jl_float64_type, 1);
jl_array_t* x          = jl_alloc_array_1d(array_type, 10);
```

或者，如果你已经分配了数组你能生成一个对数据的简单包装：

```
double *existingArray = (double*)malloc(sizeof(double)*10);
jl_array_t *x = jl_ptr_to_array_1d(array_type, existingArray, 10, 0);
```

最后一个参数是一个表明 Julia 是否应该获取数据所有权的布尔值。如果这个参数非零，GC 将在数组不再引用时在数据指针上调用 `free` 。

为了获取 x 的数据，我们可以使用 `jl_array_data` ：

```
double *xData = (double*)jl_array_data(x);
```

现在我们可以填写数组：

```
for(size_t i=0; i<jl_array_len(x); i++)
  xData[i] = i;
```

现在让我们调用一个在 `x` 上运行操作的 Julia 函数：

```
jl_function_t *func  = jl_get_function(jl_base_module, "reverse!");
jl_call1(func, (jl_value_t*)x);
```

通过打印数组，我们可以核实 `x` 的元素现在被颠倒了。

访问返回的数组

如果一个 Julia 函数返回一个数组， `jl_eval_string` 和 `jl_call` 的返回值能被转换成一个 `jl_array_t*` ：

```
jl_function_t *func  = jl_get_function(jl_base_module, "reverse");
jl_array_t *y = (jl_array_t*)jl_call1(func, (jl_value_t*)x);
```

现在 `y` 的内容能在使用 `jl_array_data` 前被获取。一如往常，当它在使用中时确保保持对数组的引用。

高维数组

Julia 的多维数组在内存中以列的顺序被存储。这儿是一些创建二维数组和获取属性的代码：

```
// Create 2D array of float64 type
jl_value_t *array_type = jl_apply_array_type(jl_float64_type, 2);
jl_array_t *x  = jl_alloc_array_2d(array_type, 10, 5);

// Get array pointer
double *p = (double*)jl_array_data(x);
// Get number of dimensions
int ndims = jl_array_ndims(x);
// Get the size of the i-th dim
size_t size0 = jl_array_dim(x,0);
size_t size1 = jl_array_dim(x,1);

// Fill array with data
for(size_t i=0; i<size1; i++)
   for(size_t j=0; j<size0; j++)
      p[j + size0*i] = i + j;
```

注意到当 Julia 数组使用 1-based 的索引，C 的 API 使用 0-based 的索引（例如在调用 `jl_array_dim` 时）以作为惯用的 C 代码读取。

异常

Julia 代码能抛出异常。比如，考虑以下：

```
jl_eval_string("this_function_does_not_exist()");
```

这个调用将什么都不做。但是，检查一个异常是否抛出是可能的。

```
if (jl_exception_occurred())
    printf("%s \n", jl_typeof_str(jl_exception_occurred()));
```

如果你用一个支持异常的语言（比如，Python，C#，C++）使用 Julia C API，用一个检查异常是否被抛出的函数包装每一个调用 libjulia 的调用是有道理的，而且它用主语言重新抛出异常。

抛出 Julia 异常

当写一个可调用的 Julia 函数时，验证参数和抛出异常来指出错误是必要的。一个典型的类型检查像这样：

```
if (!jl_is_float64(val)) {
    jl_type_error(function_name, (jl_value_t*)jl_float64_type, val);
}
```

通常的异常能使用函数来引起：

```
void jl_error(const char *str);
void jl_errorf(const char *fmt, ...);
```

`jl_error` 使用一个 C 的字符串，`jl_errorf` 像 `printf` 一样被调用：

```
jl_errorf("argument x = %d is too large", x);
```

在这个例子中 `x` 被假设为一个整型。

27

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 的包管理有一点不同这是因为它是生命而不是必要。这意味着你告诉它你想要什么，它就会知道安装什么版本（或移除）来有选择地满足那些需求 – 最低程度下地。所以不是安装一个包，你只是添加它到需求列表然后“解决”什么需要被安装。特别的，这意味着如果一些包因为它被你想要东西的前一个版本所需要而已经被安装，而且一个更新的版本不再有那个需求了，更新将真正移除那个包。

你的包需求在文件 `~/.julia/v0.3/REQUIRE` 中。你可以手动编辑这个文件，然后调用 `Pkg.resolve()` 方法来安装，升级或者移除包来有选择地满足需求，或者你可以做 `Pkg.edit()`，它将在你的编辑器中打开 `REQUIRE`（通过 `EDITOR` 或者 `VISUAL` 环境变量配置），然后之后自动调用 `Pkg.resolve()`，如果有必要的话。如果你仅仅想要添加或者移除一个单一包的需求，你也可以使用非交互的 `Pkg.add` 和 `Pkg.rm` 命令，它添加或移除一个单一的需求来 `REQUIRE`，然后调用 `Pkg.resolve()`。

你可以用 `Pkg.add` 函数添加一个包到需求列表，这个包和所有它所依赖的包都将被安装：

```
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
```

这所做的事情首先是添加 `Distributions` 到你的 `~/.julia/v0.3/REQUIRE` 文件：

```
$ cat ~/.julia/v0.3/REQUIRE
Distributions
```

然后它使用这些新的需求运行 `Pkg.resolve()`，它导向了 `Distributions` 包应该被安装因为它是必需的而且没有被安装的结论。正如之前所声明的，你可以通过手动编辑你的 `~/.julia/v0.3/REQUIRE` 文件完成相同的事情然后自己运行 `Pkg.resolve()`。

```
$ 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
```

这和调用 `Pkg.add("UTF16")` 功能相同，除了 `Pkg.add` 直到在安装完成之后才改变 `REQUIRE`，所以如果有问题的话，`REQUIRE` 将被剩下，正如在调用 `Pkg.add` 之前。`REQUIRE` 文件的格式在 Requirements Specification中被描述；它允许在其他事物中获得特定包版本的范围。

当你决定你不想再拥有一个包，你可以使用 `Pkg.rm` 来从 `REQUIRE` 文件移除它的需求：

```
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.
```

再一次，这和编辑 `REQUIRE` 文件来移除有着包名的那一行然后运行 `Pkg.resolve()` 来更改安装包的集合来匹配相类似。尽管 `Pkg.add` 和 `Pkg.rm` 对于添加和移除单个包的需求来说是方便的，当你想要添加或移除多个

包时，你可以调用 `Pkg.edit()` 来手动地改变 `REQUIRE` 的内容然后根据情况更新你的包。`Pkg.edit()` 不回滚 `REQUIRE` 的内容如果 `Pkg.resolve()` 失效 – 不如说，你不得不再一次运行 `Pkg.edit()` 来修改文档内容。

因为包管理内部使用 git 来管理包 git 仓库，当运行 `Pkg.add` 时，用户可能会碰上协议的问题（比如在一个防火墙后）。接下来的命令可在命令行中被运行来告诉 git 当克隆仓库时使用 'https' 而不是 'git' 协议。

```
git config --global url."https://".insteadOf git://
```

安装未注册的扩展包

Julia 包仅仅是 git 仓库，在任何 git 支持的协议上都是可克隆的，而且包含遵循特定布局惯例的 Julia 代码。官方的 Julia 包在 METADATA.jl 仓库中注册，在可以著名的地方可获得。在之前的段落中，`Pkg.add` 和 `Pkg.rm` 命令和注册的包交互，但是包管理也能安装并使用未注册的包。为了安装未注册的包，使用 `Pkg.clone(url)`，在那里 `url` 是一个包能被克隆的 git URL：

```
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.
```

按照惯例，Julia 仓库用一个 `.jl` 的结尾命名（附加的 `.git` 指示了一个“裸” git 仓库），这防止它们和其他语言的仓库碰撞，也使得 Julia 包在搜索引擎中方便找到。当包在你的 `.julia/v0.3` 目录下安装时，然而，扩展是多余的，所以我们将它留下。

如果未注册的包在它们的资源树的顶部包含 `REQUIRE` 文件，那这个文件将被用来决定未注册的包依赖于哪些注册的包，而且它们将自动被安装。未注册的包和注册的包一样，具有相同版本的解决逻辑，所以安装过的包版本将在必要时调整来满足注册过的和未注册过的包的需求。

[1] 官方的包集在 https://github.com/JuliaLang/METADATA.jl，但是个人和组织能简单地使用一个不同的元数据仓库。这允许包可以自动安装的控制。我们可以仅允许审计通过的和批准的包版本，并使得私人的包和 fork 可被获得。

更新扩展包

当包开发者发布你正在使用的新的注册的包版本时，你当然，想要新的版本。为了获得最新和最棒的包版本，只要 `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
```

更新包的第一步是将新的改变放入 `~/.julia/v0.3/METADATA` 并看看是否有新的注册包版本已经被发布了。在这之后，`Pkg.update()` 通过从包的上游库 pull 一些更改会更新在一个分支上被检查且不 dirty（比如，在 git 下没有对文件更改）的更新包。上游的改变仅仅在如果没有合并或重定基地址是有必要的情况下应用 – 比如，如果分支是 "fast-forwarded"。如果分支不是 fast-forwarded，就假设你正在使用它而且将自己更改仓库。

最后，更新的过程重新计算了一个最佳的包版本的集合来安装以满足你顶级的需求和 “fix” 包的需求。包被认为是 fixed 如果它是下面几条之一：

1.**未注册：**包不在 `METADATA` 中 – 你用 `Pkg.clone` 安装过它。 2.**被检出:**包仓库在一个开发分支上。 3.**Dirty:**在仓库中对文件进行过了修改。

如果这些中的任何一项出现，包管理者不能自由地更改安装好的包版本，所以它的需求必须被满足，无论它所选择的其他包版本是怎样的。在 `~/.julia/v0.3/REQUIRE` 中的顶层需求的组合和修改过的包的需求被用来决定应该安装什么。

Checkout, Pin and Free

你可能想要使用包的 master 版本而不是注册版本中的一个。在 master 上可能有修改或功能,它们是你所需要的且没有在任何注册版本上发布，或者你可能是一个包的开发者且想要改变 master 或一些其他的开发分支。在这些例子中，你能通过 Pkg.checkout(pkg) 来检查 pkg 或 Pkg.checkout(pkg,branch) 的 master 分支以检查一些其他的分支：

```
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
```

一旦在用 Pkg.add 安装 Distributions 之后，在写完的同时它就位于最新的注册版本上 - 0.2.9 。然后在运行 Pkg.checkout("Distributions") 之后，你可以从 Pkg.status() 的输出中看到 Distributions 比起 0.2.9 在一个未注册的版本上更佳。由 “pseudo-version” 数字 0.2.9+ 指示。

当你检查一个未注册的包版本时，包仓库中 REQUIRE 文件的副本地位高于任何其他在 METADATA 中注册的需求，所以开发者保持这个文件的正确性和及时性是很重要的，这反映了目前包版本的真正需求。如果在包仓库中的 REQUIRE 文件是不正确的或者遗失了，当包被检出时依赖性可能会被移除。这个文件也被用来填充新发布的包版本，如果你使用了 Pkg 为此提供的 API（在下面描述）。

当你决定你不再想要让一个包在分支上被检出，你能使用 Pkg “释放”它回到包管理者的控制之下。

```
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
```

在这之后，因为包是在一个注册版本之上而且不在一个分支上，它的版本将被更新作为包的注册版本被发布。

如果你想要在一个指定的版本上 pin 一个包以使调用 `Pkg.update()` 不会改变包所在的版本，你可以使用 `Pkg.pin` 功能：

```
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
```

在这之后，`Stats` 包将以版本 `0.2.7` 保持 pin 的状态 - 或者更具体地说，在提交 `47c198b1` 时，但是自从版本被永久地和一个给定的 git hash 连接后，这就一样了。`Pkg.pin` 通过为你想要 pin 包的提交创建一个 throw-away 分支而运行。默认下，它在当前的提交下 pin 了一个包，但是你能通过传递第二个参数选择一个不同的版本：

```
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
```

现在 `Stats` 包在提交 `1fd0983b` 时被 pin 了，它和 `0.2.5` 版本相一致。当你决定 “unpin” 一个包且让包管理者再一次更新它时，你可以使用 `Pkg.free` 就像你想要离开任何分支一样：

```
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
```

Julia 的包管理者被设计以让当你有一个包需要安装时，你就可以查看它的源代码和完整的开发历史。你也可以对包做出更改，使用 git 提交它们，并能简单地作出修改和增强。相类似的，系统被设计以让如果你想要创建一个新的包，这么做最简单的方法就是在由包管理者提供的基础设施内部。

[2]:不在分支上的包也将被标记为 dirty，如果你在仓库中作出改变，但是那是一件比较少见的事。

28

开发扩展包

Julia 中设有包管理器，当你安装了扩展包时，你可以看到它的源代码和完整的开发历史。你也可以修改扩展包，并使用 git 提交它们，为修复和增加扩展包功能做贡献。相似地，这个系统设计用来当你想要创建一个新扩展包时，最简单的方法就是利用包管理器中提供的基础设施。

初始化设置

由于扩展包存储于 git 仓库中，所以在做扩展包开发之前，你需要先设置如下全局 git 配置：

```
$ git config --global user.name "FULL NAME"
$ git config --global user.email "EMAIL"
```

`FULL NAME` 是你真实的全名(双引号之间允许有空格)并且 `EMAIL` 是你真实的邮箱地址。
尽管创建和发布 Julia 扩展包时使用 GitHub 并不是必要的，然而大多数 Julia 扩展包都存在 GitHub 上并且包管理器知道如何正确地格式化源 URL，并在其他方面上顺利的使用服务。我们建议你创建一个免费账号 在 GitHub 上然后做：

```
$ git config --global github.user "USERNAME"
```

在这里 `USERNAME` 是你 GitHub 上正确的用户名。只要你做了这一点，包管理器就知道你的 GitHub 用户名然后可以配置相关事项。你还需要上传 你的 SSH 公钥到 GitHub 上并设置一个 SSH 代理在你的开发机器上，这样你可以最简单的推送你的修改。在将来，我们会让这个系统具有扩展性，支持更多其它的常见 git 工具例如 BitBucket 并且允许开发者选择他们所喜欢的。

生成新扩展包

假如你想创建一个新的 Julia 扩展包，名为 `FooBar`。首先，你需要 `Pkg.generate(pkg,license)`，其中 `pkg` 是新扩展包的名字并且 `license` 是生成器知晓的许可的名字：

```
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
```

这样创建了一个目录 `~/.julia/v0.3/FooBar`，将它初始化为一个 git 仓库，生成所有包需要有的一系列文件，并把它们提交到仓库：

```
$ 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(+)
```

此时，包管理器知道 MIT "Expat" 证书用 `"MIT"` 表示，Simplified BSD 证书用 `"BSD"` 表示，2.0 版本的 Apache 软件证书用 `"ASL"` 表示。如果你想要使用不同的证书，你可以让我们把它添加到扩展包生成器上，或者就选这三者之一然后在生成之后修改 `~/.julia/v0.3/PACKAGE/LICENSE.md` 文件。

如果你创建了一个 GitHub 账户并且配置了 git, `Pkg.generate` 将会设置一个合适的源 URL 给你。它还会自动生成 `.travis.yml` 文件来使用 Travis 自动测试服务。你可以在 Travis website 上测试你的扩展包仓库，但是只要你做了这个它就已经开始测试了。当然，所有的默认测试是查证 `using FooBar` 能否在 Julia 上工作。

使你的扩展包具有可用性

只要你提交了一些内容，那么你会为测试 `FooBar` 是否可以工作而感到高兴，你可能想要一些其他人来测试一下。首先，你需要创建一个远程仓库并把你的代码推送进去；我们不会自动的为你做这件事，但是未来将会，这配置起来并不难[3]。只要你完成了这个，只需将发布的仓库的 URL 发给他们就可以请让他们来试一下你的代码 – 像这样：

```
git://github.com/StefanKarpinski/FooBar.jl.git
```

对于你的扩展包而言，它将具有你的 GitHub 用户名和你的扩展包名，但是你明白是什么意思。收到你发的 URL 的人们可以使用 `Pkg.clone` 来安装扩展包并测试它:

```
julia> Pkg.clone("git://github.com/StefanKarpinski/FooBar.jl.git")
INFO: Cloning FooBar from git@github.com:StefanKarpinski/FooBar.jl.git
```

> [3]: 极度推荐安装并使用 GitHub 的 "hub" 工具。它允许你在扩展包仓库中像运行 `hub create` 那样做事，然后它会通过 GitHub 的 API 自动创建。

发布你的扩展包

一旦你决定 FooBar 已经准备好注册成为一个官方正式扩展包，你可以把它添加到你的本地 METADATA 的拷贝，并命名为 Pkg.register：

```
julia> Pkg.register("FooBar")
INFO: Registering FooBar at git://github.com/StefanKarpinski/FooBar.jl.git
INFO: Committing METADATA for FooBar
```

这会在 ~/.julia/v0.3/METADATA 仓库中创建一次提交：

```
$ 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
```

然而，这次提交只是本地可见的。为了能将它公诸于世，你需要将你的本地 METADATA 上传到正式库中合并。Pkg.publish() 命令将在 GitHub 上创建 METADATA 仓库的分支，并将你的修改提交到分支上，并打开一个拉取请求：

```
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
```

由于各种各样的原因 `Pkg.publish()` 有时并不会成功。在那些情况下，你可能在 GitHub 上做了一个拉取请求，这并不难。

只要 `FooBar` 扩展包的 URL 在正式 `METADATA` 仓库中注册，人们就知道从哪里克隆这个扩展包，但是这并没有一些注册过的版本可供下载。这意味着 `Pkg.add("FooBar")` 在只安装正式版本时并没有工作。`Pkg.clone("FooBar")` 没有一个指定的 URL 指向它。此外，当他们运行 `Pkg.update()`，他们将会得到你上传到仓库中最新版本的 `FooBar` 。当你还在修改它，在它没有成为正式版之前这是一个比较好的方式测试你的扩展包。

扩展包版本号标签

当你准备好为你的扩展包制作一个正式版本时，你可以使用 `Pkg.tag` 命令为它添加版本号并注册：

```
julia> Pkg.tag("FooBar")
INFO: Tagging FooBar v0.0.1
INFO: Committing METADATA for FooBar
```

这个 `v0.0.1` 标签在 `FooBar` 仓库中：

```
$ cd ~/.julia/v0.3/FooBar && git tag
v0.0.1
```

它也可以为 `FooBar` 在你的本地 `METADATA` 仓库中创建一个新的版本入口：

```
$ 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
```

如果在你的扩展包仓库中有一个 `REQUIRE` 文件，它将会在你标记版本时拷贝到 `METADATA` 中适当的位置。扩展包开发者们需要确定他们的扩展包中的 `REQUIRE` 文件确实反应他们扩展包的需求，如果你使用 `Pkg.tag` 命令，这将自动进入你的正式版。看 Requirements Specification (页 0) 来了解完整格式的 `REQUIRE` 。

`Pkg.tag` 命令有第二个可选参数是一个显示的版本号对象如 `v"0.0.1"` 或者一个标志 `:patch` ， `:minor` 或者 `:major` 。这会智能地添加你的扩展包的补丁、副本或者主版本号。

正如使用 `Pkg.register`，这些对于 `METADATA` 的修改不会对其它任何人可见直到这些修改被上传。再一次使用 `Pkg.publish()` 命令行，它第一次使用的时候要确定每个独立的扩展包仓库已经被标记，如果它们没有被标记要提交它们，然后打开一个到 `METADATA` 的拉取请求：

```
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
```

修改扩展包需求

如果你需要修改一个已发布扩展包版本的注册需求，你只需要修改这个版本的 metadata 即可，这样可以保持相同的提交散列值 – 散列值与一个版本永久相关:

```
$ cd ~/.julia/v0.3/METADATA/FooBar/versions/0.0.1 && cat requires
julia 0.3-

$ vi requires
```

为了保持提交的散列值保持一致，需要检验仓库中的 `REQUIRE` 文件的内容是否与在 `METADATA` 中的在修改之后不匹配；这是不可避免的。

尽管当你在 `METADATA` 中为之前版本的扩展包修改了需求，你仍需要在当前版本的扩展包中修改 `REQUIRE` 文件。

依赖关系

在扩展包中的 `~/.julia/v0.3/REQUIRE` 文件，`REQUIRE` 文件，和 `METADATA` 包 `requires` 文件使用一个简单的基于行的格式来显示需要安装的扩展包版本的范围。包 `REQUIRE` 和 `METADATA requires` 文件也需要包括扩展包兼容的 `julia` 的版本范围。

这里是这些包如何被解析和解释的。

- 所有在 `#` 号后的内容被从行中剥离成为注释。
- 如果出了空白什么都没有，那么这一行被忽略。
- 如果剩下的都是非空字符，那么这一行是一个依赖关系，并且需要用空格分开每个单词。

最简单的有可能的依赖关系是这一行只有扩展包的名字：

```
Distributions
```

这个依赖将被任何版本的 `Distributions` 扩展包满足。这个扩展包的名字可以紧随零活更多升序版本号之后，指明可以接受的那个扩展包的版本间隔。一个版本号开始一个间距，下一个是这个间距的结束，然后下一个又是一个新的开始，然后继续；如果出现了一个奇怪的版本号，那么任意更高的版本都将兼容；如果给出了一个相同的版本号，那么后一个是可以兼容的最高版本。举个例子，这一行：

```
Distributions 0.1
```

`0.1.0` 及其之后的版本的 `Distributions` 都将被兼容。一个版本号以 `-` 作为后缀也允许任何相同前缀的发布版本兼容。例如：

```
Distributions 0.1-
```

兼容相同前缀的版本例如 `0.1-dev` 或 `0.1-rc1`，或 `0.1.0` 及其之后的任何版本。
这个依赖条目：

```
Distributions 0.1 0.2.5
```

兼容从 `0.1.0` 起的任何版本，但是不包括 `0.2.5`。
如果你想要表明任何 `0.1.x` 版本被兼容，你可以这样写：

```
Distributions 0.1 0.2-
```

如果你想要兼容在 `0.2.7` 之后的版本，你可以这样写：

```
Distributions 0.1 0.2- 0.2.7
```

如果一个依赖行以引导字符 `@` 开始，这是一个系统依赖关系。如果你的系统匹配这些系统环境，依赖关系就会被包含，否则将被忽略。例如：

```
@osx Homebrew
```

将仅在操作系统是 OS X 时需要 `Homebrew` 扩展包。当前支持的系统环境包括：

```
@windows
@unix
@osx
@linux
```

`@unix` 环境适应于所有的 UNIX 操作系统，包括 OS X, Linux 和 FreeBSD。在引导字符 `@` 后添加 `!` 表示否定的操作系统。例子：

```
@!windows
@unix @!osx
```

第一个环境应用于任何系统除了 Windows ，第二个环境应用于任何 UNIX 系统除了 OS X。

运行时检查 Julia 的当前版本可以应用在内置 `VERSION` 变量，这是一种 `VersionNumber` 。这些代码偶尔是必要的用来跟踪在发布的 Julia 版本之间的新功能或弃用的功能。运行时检查的例子：

```
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
```

到 *version number literals* 查看跟过更完整的描述细节。

29

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)
```

写函数是一种更好的风格，这会产生更多可重复和清晰的代码，也包括清晰的输入和输出。

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 :ref: 工具 <man-performance-tools> 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 :ref: 预分配内存 <man-preallocation>).

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 <https://github.com/tonyhffong/Lint.jl> _ 扩展包可以指出程序一 些问题.

避免包含一些抽象类型参数

当运行参数化类型时候，比如 arrays，如果有可能最好去避免使用抽象类型参数。 思考下面的代码：

```
a = Real[]    # typeof(a) = Array{Real,1}
if (f = rand()) < .8
    push!(a, f)
end
```

因为 `a` 是一个抽象类型 `Real` 的 array，所以可以包含任何 `Real` 类型的值。既然 `Real` 对象可以是任意的大小和结构， `a` 必须被解释为一个 array 数组指向所有可能的对象。所以我们应该用确定的类型代替，比如 `Float64` ：

```
a = Float64[] # typeof(a) = Array{Float64,1}
```

这样会建立大小为 64 位的浮点值，也会更有效率。

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。

内存列中的访问数组

Julia 中的多维数组是根据以列为主的顺序存储的。这意味着每次数组都占据了一列。我们可以通过如下所示的 `vec` 功能或者是 或是排列 `[:]` 来进行验证（注意到数组的顺序是 `[1 3 2 4]` 而不是 `[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
```

这种给数组排序的约定在许多语言中都是常见的，比如 Fortran ， Matlab ，和 R 语言(举几个例子来说)。以列为主序的另一选择就是以行为主序，其它语言中的 C 语言和 Python 语言(`numpy`)就是选用了这种方式。记住数组的顺序对数组的查找有着至关重要的影响。要记住的一个查找规则就是对于基于列为顺序的数组，第一个指针是变化最快的。这基本上就意味着如果在一段代码中，循环指针是第一个，那么查找速度会更快。

我们来看一下下面这个人为的例子。假设我们想要实现一个功能，接收一个 `Vector` 并且返回一个方形的 `Matrix` ，且行或列为输入矢量的复制。我们假设是行还是列为数据的复制并不重要（或许剩下的代码可以相应地更容易的适应）。我们可以想到有至少四种方法可以实现这一点（除了建议的回访正建的 `repmat` 功能）：

```
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
```

现在我们使用同样的输入向量 `1` 产生的随机数 `10000` 给每个功能计时：

```
julia> x = randn(10000);

julia> fmt(f) = println(rpad(string(f)*": ", 14, ' '), @elapsed f(x))

julia> map(fmt, {copy_cols, copy_rows, copy_col_row, copy_row_col});
copy_cols:    0.331706323
copy_rows:    1.799009911
copy_col_row: 0.415630047
copy_row_col: 1.721531501
```

注意到 `copy_cols` 比 `copy_rows` 快很多。这是意料之中的，因为 `copy_cols` 遵守 `Matrix` 界面的基于列的存储，并且一次就填满一列。除此之外， `copy_col_row` 比 `copy_row_col` 快很多，因为它符合我们的查找规则，即在一段代码中第一个出现的元素应该是与最内部的循环相联系的。

输出预先分配

如果你的功能返回了一个 Array 或其它复杂类型，它可能不得不分配内存。不幸的是，时常分配和它的相反事件，垃圾区收集，是有实质性瓶颈的。

有时候，你可以在访问每个功能时通过预先分配输出来避开分配内存的需要。作为一个很小的例子，比较一下

```
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
```

和

```
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
```

计时结果：

```
    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
```

预先分配有其他好处，比如，允许访问者通过算法控制“输出”类型。在上面的例子中，我们可以按照自己希望的，通过一个 `SubArray` 而不是 `Array` 。

按着最极端的来想，预先分配可以让你的代码看起来丑点，所以需要一些表达方式和判断。

避免输入/输出时的串插入

把数据写入文件（或者其他输入/输出设备）时，中间字符串的形成是额外的开销。而不是：

```
println(file, "$a $b")
```

使用：

```
println(file, a, " ", b)
```

第一种代码形成了一个字符串，然后把它写入了文件，而第二种代码直接把值写入了文件。同样也注意到在某些情况下，字符串的插入很难读出来。考虑一下：

```
println(file, "$(f(a))$(f(b))")
```

对比：

```
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)` .

性能注释

有时你可以设定某些项目属性来获得更好的优化。

- 在检查公式时，使用 `@inbounds` 来消除数组界限。一定要在这之前完成。如果下标越界了，你可能会遇到崩溃或不执行的问题。
- 在 `for` 循环之前写上 `@simd`，这个可以帮你检验。**这个特征是试验性的**而且在之后的 Julia 版本中可能会改变会消失。

这里有一个包含两种形式审定的例子：

```
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)
```

在配有 2.4GHz 的 Intel Core i5 处理器的电脑上，产生如下结果：

```
GFlop       = 1.9467069505224963
GFlop (SIMD) = 17.578554163920018
```

`@simd for` 循环应该是一维范围的。*缩减变数* 是用于累积变量的，比如例子中的 `s`。通过使用 `@simd`，你可以维护循环的几种性能：

- 有缩减变数的特殊考虑后，在任意的或重叠的顺序中执行迭代都是安全的。
- 减少变量的浮点操作可以被重复执行，但是可能会比没有 `@simd` 产生不同的结果。
- 不会有一个迭代在等待另一个迭代，以实现前进。

使用 `@simd` 仅仅是给了编译器矢量化的通行证。它是不是真的会这样做还取决于编译器。要真正从当前的实现中获益，你的循环应该有如下额外的性能：

- 循环必须是内部循环。
- 循环主题必须是无循环程序。这就是为什么当前所有的数组访问都需要 `@inbounds` 的原因了。
- 访问必须有一个跨越模式，而且不能“聚集”（随机指针读取）或者“分散”（随机指针写入）。
- 跨越应该是单元跨越。
- 在一些简单的例子中，例如一个 2-3 数组访问的循环中，LLVM 自动矢量化可能会自动生效，导致无需 `@simd` 的进一步加速。

30

代码样式

以下各节从几方面介绍了符合语言习惯的 Julia 编码风格。这些规则都不是绝对的；它们仅仅是帮您熟悉这门语言，或是帮您可以在许多可替代性设计中能够做出选择的一些建议而已。

写成函数，别写成脚本

编写代码作为在一系列步骤中最高级的办法，是可以快速开始解决问题的，但您应该试着尽快把一个程序分成许多函数。函数具有更好的可重用性和可测试性，并可以更好阐明它们正在做什么，它们的输入和输出是什么。此外，由于 Julia 的编译器工作原理，在函数中的代码往往比最高级别的代码运行得更快。

同样值得强调的是，函数应该以参数来代替，而不是直接在全局变量（除了像 pi 那样的常量）上操作。

避免类型过于严格

代码应尽可能通用。相较于这样的代码书写：

```
convert(Complex{Float64}, x)
```

使用有效的泛型函数是更好的：

```
complex(float(x))
```

第二种写法把 `x` 转换成一个适当的类型，而不是一直用一个相同的类型。

这种类型特点是特别地与函数自变量相关。例如，不声明一个参数是 `Int` 类型或 `Int32` 类型，如果在这种情况下还可以保持是任何整数，那就应该是用 `Integer` 抽象表达出来的。事实上，在许多情况下您都可以把自变量类型给忽视掉，除非一些需要消除歧义的时候，由于如果一个类型不支持任何必要操作就会被忽略，那么一个 `MethodError` 不管怎样也都会被忽略掉。（这被大家认为是 duck typing。）

例如，考虑以下 `addone` 函数中的定义，这个功能可以返回 1 加上它的自变量。

```
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
```

最后一个 `addone` 的定义解决了所有类型的有关自变量的 `one` 函数（像 `x` 类型一样返回 1 值，可以避免不想要的类型提供）和 `+` 函数的问题。关键是要意识到，仅仅是定义通用的 `addone(x) = x + one(x)` 写法也是没有性能缺失的，因为 Julia 会根据需要自主编译到专业的版本。举个例子，您第一次调用 `addone(12)` 的时候， Julia 会自动为 `x::Int` 自变量编译一个 `addone` 函数，通过调用一个内联值 `1` 代替 `one`。因此，上表前三个定义全都是重复的。

在调用程序中解决额外的自变量多样性问题

取代这种写法：

```
function foo(x, y)
    x = int(x); y = int(y)
    ...
end
foo(x, y)
```

利用以下的写法更好：

```
function foo(x::Int, y::Int)
    ...
end
foo(int(x), int(y))
```

第二种写法更好的方式，因为 `foo` 并没有真正接受所有类型的数据；它真正需要的是 `Int` S。

这里的一个问题是，如果一个函数本质上需要整数，可能更好的方式是强制调用程序来决定怎样转换非整数（例如最低值或最高值）。另一个问题是，声明更具体的类型会为未来的方法定义提供更多的“空间”。

如果函数修改了它的参数，在函数名后加 *!*

取代这种写法：

```
function double{T<:Number}(a::AbstractArray{T})
    for i = 1:endof(a); a[i] *= 2; end
a
end
```

利用以下写法更好：

```
function double!{T<:Number}(a::AbstractArray{T})
    for i = 1:endof(a); a[i] *= 2; end
a
end
```

Julia 标准库在整个过程中使用以上约定，并且 Julia 标准库还包含一些函数复制和修饰形式的例子（例如 `sort` 和 `sort!`），或是其它只是在修饰（例如 `push!`，`pop!`，`splice!`）的例子。这对一些也要为了方便而返回修改后数组的函数来说是很典型的。

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。

尽量避免空域

当使用 `x::Union(Nothing,T)` 时，想想把 `x` 转换成 `nothing` 这个选项是否是必要的。以下是一些可供选择的替代选项

- 找到一个安全的默认值来和 `x` 一起初始化
- 介绍另一种缺少 `x` 的类型
- 如果有许多类似 `x` 的域，就把它们存储在字典中
- 确定当 `x` 是 `noting` 时是否有一个简单的规则。例如，域通常是以 `nothing` 开始的，但是是在一些定义良好的点被初始化。在这种情况下，要首先考虑它可能没被定义。

避免复杂的容器类型

通常情况下，像下面这样创建数组是没什么帮助的：

```
a = Array(Union(Int,String,Tuple,Array), n)
```

在这种情况下 `cell(n)` 这样写更好一些。 这也有助于对编译器进行注释这一特定用途，而不是试图将许多选择打包成一种类型。

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

避免错误要比依赖找错好多了。

不要把条件表达式用圆括号括起来

Julia 在 if 和 while 语句中不需要括号。所以要这样写：

```
if a == b
```

来取代：

```
if (a == b)
```

不要滥用 ...

剪接功能参数可以让人很依赖。取代 `[a..., b...]` 这种写法，简单的 `[a, b]` 这样写就已经连接数组了。`collect(a)` 的写法要比 `[a...]` 好，但是因为 `a` 已经是可迭代的了，直接用 `a` 而不要把它转换到数组中也许会更好。

不要使用不必要的静态参数

信号函数：

```
foo{T<:Real}(x::T) = ...
```

应该这样写：

```
foo(x::Real) = ...
```

特别是如果 T 没被用在函数主体。即使 T 被用在函数主体了，如果方便的话也可以被 typeof(x) 替代。这在表现上并没有什么差异。要注意的是，这不是对一般的静态参数都要谨慎，只是在它们不会被用到时要特别留心。

还要注意容器类型，特别是函数调用中可能需要的类型参数。可以到 FAQ 如何声明“抽象容器类型”的域 来查看更多信息。

避免对实例或类型判断的困扰

一些如以下的定义是十分让人困扰的：

```
foo(::Type{MyType}) = ...
foo(::MyType) = foo(MyType)
```

您要决定问题的概念是应被写作 `MyType` 或是 `MyType()`,并要坚持下去。

最好的类型是用默认的实例，并且在解决某些问题需要方法时，再添加包括 `Type{MyType}` 的一些方法好一些。

如果一个类型是一个有效的枚举，它就应该被定义为一个单一的（理想情况下不变的）类型，而枚举变量是它的实例。构造函数和一些转换可以检测值是否有效。这项设计最好把枚举做成抽象类型，把“值”做成其子类型。

不要滥用 macros

您要注意什么时候一个 macros 可以真的代替函数。

在 macros 中调用 `eval` 实在是个危险的标志;这意味着 macros 只有在被最高级调用的时候才会工作。如果这样一个 macros 被写为一个函数，它将自然地访问它需要的运行时值。

不要在接口层暴露不安全的操作

如果您有一个使用本地指针的类型：

```
type NativeType
    p::Ptr{Uint8}
    ...
end
```

不要像下面这样写定义：

```
getindex(x::NativeType, i) = unsafe_load(x.p, i)
```

问题是，这种类型的用户可能在不知道该操作是不安全的情况下就写 `[i]`，这容易导致内存错误。

这样的函数应该能检查操作，以确保它是安全的，或是在它的名字中有不安全的地方时可以提醒调用程序。

不要重载基容器类型的方法

像下面这样书写定义是有可能的：

```
show(io::IO, v::Vector{MyType}) = ...
```

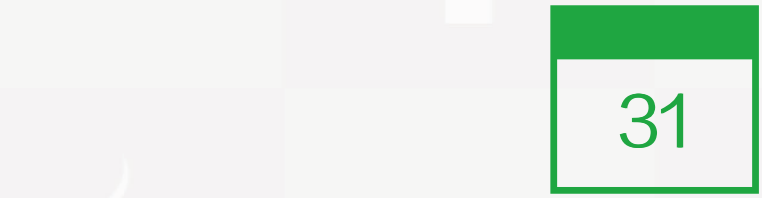
这样写将提供一个特定新元素类型的向量的自定义显示。虽然很让人想尝试，但却是应该避免的。麻烦的是，用户会想用一个众所周知的类型比如向量在一个特定的方式下的行为，也会过度定制它的行为，这都会使工作更困难。

注意类型的相等性

您一般要使用 `isa` 和 `<:`（`issubtype`）来测试类型而不会用 `==`。在与已知的具体类型的类型进行比较时，要精确检查类型的的相等性（例如 `T == Float64`），或者是您真的明白您究竟在干什么。

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)` 。



31 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
```

1. 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
```

Julia 为什么使用本机整数运算?

Julia 会应用机器运算的整数计算。这意味着 int 值的范围是有界的，是在两界之间取值的，所以添加，减去，乘以和除以一个整数都可能导致上溢或下溢，这可能会导致一些不好的后果，这种情况在一开始会让人感到很不安。

```
julia> typemax(Int)
9223372036854775807

julia> ans+1
-9223372036854775808

julia> -ans
-9223372036854775808

julia> 2*ans
0
```

显然，这远远不能用数学的方法来表现，您可能会认为 Julia 与一些高级编程语言会公开给用户这一情况相比来说不是那么理想。然而这对于效率和透明度都非常珍贵的数值工作来说，相比之下，替代品更是糟糕。

这里有一个选择是来检查每个整数操作的溢出情况，并且由于溢出情况而提高结果值到大一些的整数类型，例如 `Int128` 或 `BigInt` 。不幸的是，这就引进了在每个整数操作上都会有的主要负担（想想增加一个循环计数器）- 这需要发射代码在算术指令后执行程序时的溢出检查，并且需要一些分支来解决潜在溢出问题。更糟糕的是，这会导致每一个计算，在涉及整数时都是不稳定的。正如我们上面提到的，类型的稳定性是有效的代码生成的关键。如果您不能指望整数运算的结果是整数，那么按 C 和 Fortran 编译器方式做的简单代码，想要生成速度快是不可能的。

这个方法还有一个可以避免不稳定类型外观的变化，就是把 `Int` 和 `BigInt` 合并成一个单一的混合整数类型，当结果不再适合机器整数的大小时，可以由内部改变来表示。然而这只是表面上解决了 Julia 语言的不稳定性水平问题，它也仅仅只是通过强硬地把所有相同的难题汇于 C 语言，使混合整数类型可以成功实现的方式，解决了几个小问题而已。这种方法基本上可以进行工作，甚至可以在许多情况下可以作出相当快的反应，但是还是有几个缺点的。其中一个问题是，在内存中整数和整数数组的表示方法，不再和 C，Fortran 等其它具有本地机器整数的语言的本地表示方法一一对应了。因此，对这些语言进行互操作，我们无论如何最终都需要引入本地的整数类型。任何无界表示的整数都没有一个固定的位，因此它们不能内联地被存储在有固定大小的槽的数组里，较大的整数的值会一直需要单独的堆分配来进行存储。当然，不管一个混合整数的实现有多精妙，总会有性能陷阱的情

况或是性能下降的情况。复杂的表示的话，缺乏与 C 和 Fortran 语言的互操作性，不能代表没有额外堆存储的整数数组，并且不可预知的性能特点使即使最精妙的混合整数来实现高性能计算的工作不管怎样都不是个好办法。

还有一个在使用混合整数或是使其提高到 BigInts 的选择是用饱和的整数运算实现的，这个运算使即使把一个数添加到最大的整数值，值也不会变，同样的，从最小的整数值减去数值，值也不变。这恰恰就是 Matlab™ 可以实现的。

```
>> int64(9223372036854775807)

ans =

  9223372036854775807

>> int64(9223372036854775807) + 1

ans =

  9223372036854775807

>> int64(-9223372036854775808)

ans =

 -9223372036854775808

>> int64(-9223372036854775808) - 1

ans =

 -9223372036854775808
```

乍一看，这似乎很合理,因为 922337203685477580 是比 -922337203685477580 更要接近 9223372036854775807 的，并且整数还是表现在一种用 C 语言和 Fortran 语言兼容的固定大小实现的本地的方式。然而，饱和的整数运算，是非常有问题的。首先的和最明显的问题是，它不是机器的整数算术操作方式，所以每台机器进行整数运算来检查下溢或上溢，并且用 typemin（int）或 typemax（int）适当地取代结果之后，才可以实现发出饱和操作需要发出的指令。这就单独将每一个整数运算从一个单一的、快速的指令扩展到 6 个指令，还可能包括分支。但它会变得更糟 - 饱和的整数算术并不是联想的。来考虑这个 MATLAB 计算：

```
>> n = int64(2)^62
4611686018427387904

>> n + (n - 1)
9223372036854775807
```

```
>> (n + n) - 1
9223372036854775806
```

这使得它很难写很多基本的整数算法，因为很多常见的技术依赖于这样一个事实，即机器加成与溢出是联想的。考虑在 Julia 中利用 (lo + hi) >>> 1 表达式来找到整数值 lo 和 hi 的中间点：

```
julia> n = 2^62
4611686018427387904

julia> (n + 2n) >>> 1
6917529027641081856
```

看见了吗？没有问题。这是 2^62 和 2^63 之间正确的中点，尽管 `n + 2n` 实际应是 -4611686018427387900。现在尝试在 MATLAB 中：

```
>> (n + 2*n)/2

ans =

  4611686018427387904
```

这就出错了。添加一个 a >>> 运算元到 Matlab 上并不会有帮助。因为添加 n 和 2n 已经破坏了必要的计算正确的中点的信息时，饱和就发生了。

这不仅是程序员缺乏结合性而不幸不能依赖这样的技术,而且还打败几乎任何编译器可能想做的优化整数运算。例如，由于 Julia 的整数使用正常的机器整数运算，LLVM 是自由的积极简单的优化小函数如 f(k) = 5k-1。这个函数的机器码就是这样的：

```
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
```

函数的实际体是一个单一的 `lea` 指令，计算整数时立刻进行乘，加运算。当 f 被嵌入另一个函数时，更加有利处：

```
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
```

由于 `f` 调用被内联，循环体的结束时只是一个单一的 `lea` 指令。接下来，如果我们使循环迭代次数固定，我们可以来考虑发生了什么：

```
julia> function g(k)
        for i = 1:10
          k = f(k)
        end
        return k
      end
g (generic function with 2 methods)

julia> code_native(g,(Int,))
    .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
```

因为编译器知道整数的加法和乘法之间的联系并且乘法分配时优先级会高于除法 - 这两者都是真正的饱和运算 - 它们可以优化整个回路使之只留下来的只是乘法和加法。饱和算法完全地打败了这种最优化，这是因为结合性和分配性在每次在循环迭代都可能会失败，而所导致的不同后果取决于在哪次迭代会失败。

饱和整数算法只是一个真的很差的语言语义学选择的例子，它可以阻止所有有效的性能优化。在 C 语言编程中有很多事情是很难的，但整数溢出并不是其中之一，特别是在 64 位系统中。比如如果我用的整数可能会变得比 2^63-1 还要大，我可以很容易地预测到。您要问自己我是在遍历存储在计算机中的实际的东西么？之后我就可以确认数是不会变得那么大的。这点是可以保证的，因为我没那么大的存储空间。我是真的在数实际真实存在的东西么？除非它们是宇宙中的沙子或原子粒，否则 2^63-1 已经足够大了。我是在计算阶乘么？之后就可以确认，它们可能变得特别大-我就应该用 BigInt 了。看懂了么？区分起来是很简单的。

类型的“抽象的”或者不明确的域如何与编译器进行交互?

类型可以在不指定字段的类型的情况下声明：

```
julia> type MyAmbiguousType
           a
       end
```

这允许 `a` 是任何类型。这通常是非常有用的，但它有一个缺点：对于 `MyAmbiguousType` 类型的对象，编译器将无法生成高效的代码。原因是编译器使用对象的类型而不是值来决定如何构建代码。不幸的是， `MyAmbiguousType` 类型只能推断出很少的信息：

```
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` 和 `c` 有着相同的类型，但是它们在内存中数据的基础表示是非常不同的。即使您只在 `a` 的域中储存数值，事实上 `Uint8` 和 `Float64` 的内存表示不同也意味着 CPU 需要用两种不同的指令来处理它们。由于类型中的所需信息是不可用，于是这样的决定不得不在运行时作出。这减缓了性能。

您可以用声明 `a` 的类型的方法做得更好。在这里，我们注意到这样一种情况，就是 `a` 可能是几个类型中的任意一种，在这种情况下自然的解决办法是使用参数。例如：

```
julia> type MyType{T<:FloatingPoint}
         a::T
       end
```

这相对以下代码是一个更好的选择

```
julia> type MyStillAmbiguousType
         a::FloatingPoint
       end
```

因为第一个版本指定了包装对象的类型。例如：

```
julia> m = MyType(3.2)
MyType{Float64}(3.2)

julia> t = MyStillAmbiguousType(3.2)
MyStillAmbiguousType(3.2)

julia> typeof(m)
MyType{Float64} (constructor with 1 method)

julia> typeof(t)
MyStillAmbiguousType (constructor with 2 methods)
```

`a` 的域的类型可以轻而易举地由 `m` 的类型确定，但不是从 `t` 的类型确定。事实上，在 `t` 中是可以改变 `a` 的域的类型的：

```
julia> typeof(t.a)
Float64

julia> t.a = 4.5f0
4.5f0

julia> typeof(t.a)
Float32
```

相反，一旦 `m` 被构造， `m.a` 的类型就不能改变了：

```
julia> m.a = 4.5f0
4.5

julia> typeof(m.a)
Float64
```

`a` 的类型可以从 `m` 的类型知道的事实和 `m.a` 的类型不能在函数中修改的事实允许编译器为像 `m` 那样的类而不是像 `t` 那样的类生成高度优化的代码。

当然，只有当我们用具体类型来构造 `m` 时，这一切才是真实的。我们可以通过明确地用抽象类构造它的方法来打破之一点：

```
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
```

对于一切实际目的，这些对象对 `MyStillAmbiguousType` 的行为相同。

对比一个简单程序所产生的全部代码是很有意义的：

```
func(m::MyType) = m.a+1
```

使用：

```
code_llvm(func,(MyType{Float64},))
code_llvm(func,(MyType{FloatingPoint},))
code_llvm(func,(MyType,))
```

由于长度的原因，结果并没有在这里显示，但您不妨自己尝试一下。因为在第一种情况下，该类型是完全指定的，编译器不需要在运行时生成任何代码来解决类型的问题。这就会有更短的代码更快的编码速度。

如何声明“抽象容器类型”的域

与应用在上一章节中的最好的相同例子在容器类型中也适用：

```
julia> type MySimpleContainer{A<:AbstractVector}
         a::A
       end

julia> type MyAmbiguousContainer{T}
         a::AbstractVector{T}
       end
```

例如：

```
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)
```

对于 `MySimpleContainer`，对象是由其类型和参数完全指定的，所以编译器可以生成优化的功能。在大多数情况下，这可能就足够了。

虽然现在编译器可以完美地完成它的工作，但某些时候您可能希望您的代码能够根据 `a` 的*元素类型*做出不同的东西。通常，达到这一点最好的方法是把您的具体操作（这里是 `foo`）包在一个单独的函数里：

```
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)
```

这在允许编译器在所有情况下都生成优化的代码，同时保持做起来很简单。

然而，有时候您需要根据 `a` 的不同的元素类型来声明外部函数的不同版本。您可以像这样来做：

```
function myfun{T<:FloatingPoint}(c::MySimpleContainer{Vector{T}})
    ...
end
function myfun{T<:Integer}(c::MySimpleContainer{Vector{T}})
    ...
end
```

这对于 `Vector{T}` 来讲不错，但是我们也要给 `UnitRange{T}` 或其他抽象类写明确的版本。为了防止这样单调乏味的情况，您可以在 `MyContainer` 的声明中来使用两个变量：

```
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}}
```

请注意一个有点令人惊讶的事实，`T` 没有在 `a` 的域中声明，一会之后我们将会回到这一点。用这种方法，一个人可以编写像这样的函数：

```
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
```

正如您所看到的，用这种方法可以既专注于元素类型 `T` 也专注于数组类型 `A` 。

然而还剩下一个问题：我们没有强制使 `A` 包括元素类型 `T` ，所以完全有可能构造这样一个对象：

```
julia> b = MyContainer{Int64, UnitRange{Float64}}(1.3:5);

julia> typeof(b)
MyContainer{Int64,UnitRange{Float64}}
```

为了防止这一点，我们可以添加一个内部构造函数：

```
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},)
```

内部构造函数要求 `A` 的元素类型为 `T` 。

无和缺值

Julia 中的“空（null）”和“无（nothingness）”如何工作?

不像许多其他语言（例如，C 和 Java）中的那样，Julia 中没有“空（null）”值。当引用（变量，对象的域，或者数组元素）是未初始化的，访问它就会立即抛出一个错误。这种情况可以通过 `isdefined` 函数检测。

有些函数只用于其副作用，不需要返回值。在这种情况下，惯例返回 `nothing`，它只是一个 `Nothing` 类型的对象。这是一个没有域的普通类型；它除了这个惯例之外，没有什么特殊的，并且 REPL 不会为它打印任何东西。一些不能有值的语言结构也统一为 `nothing`，例如 `if false; end`。

注意 `Nothing`（大写）是 `nothing` 的类型，并且只应该用在一个类型被需求环境中（例如一个声明）。

您可能偶尔看到 `None`，这是完全不同的。它是空（empty，或是“底” bottom）类型，一类没有值也没有子类型（subtypes，除了它本身）的类型。您一般不需要使用这种类型。

空元组（`()`）是另一种类型的无。但是它不应该真的被认为是什么都没有而是一个零值的元组。

Julia 发行版

我想要使用一个 Julia 的发行版本（release），测试版（beta），或者是夜间版（nightly version）？

如果您想要一个稳定的代码基础，您可能更倾向于 Julia 的发行版本。一般情况下每 6 个月发布一次，给您一个稳定的写代码平台。

如果您不介意稍稍落后于最新的错误修正和更改的话，但是发现更具有吸引力的更改的更快一点的速度，您可能更喜欢 Julia 测试版本。此外，这些二进制文件在发布之前进行测试，以确保它们是具有完全功能的。

如果您想利用语言的最新更新，您可能更喜欢使用 Julia 的夜间版本，并且不介意这个版本偶尔不工作。

最后，您也可以考虑从源头上为自己建造 Julia。此选项主要是对那些对命令行感到舒适或对学习感兴趣的个人。如果这描述了您，您可能也会感兴趣在阅读我们[指导方针href="https://github.com/JuliaLang/julia/blob/master/CONTRIBUTING.md")。

这些下载类型的链接可以在下载页面 http://julialang.org/downloads/ 找到。请注意，并非所有版本的 Julia 都可用于所有平台。

何时移除舍弃的函数?

过时的功能在随后的发行版本之后去除。例如，在 0.1 发行版本中被标记为过时的功能将不会在 0.2 发行版本中使用。

开发 Julia

我要如何调试 Julia 的 C 代码？（从一个像是 gdb 的调试器内部运行 Julia REPL）

首先您应该用 `make debug` 构建 Julia 调试版本。下面，以 `(gdb)` 开头的行意味着您需要在 gdb prompt 下输入。

从 shell 开始

主要的挑战是 Julia 和 gdb 都需要有它们自己的终端，来允许您和它们交互。一个方法是使用 gdb 的 `attach` 功能来调试一个已经运行的 Julia session。然而，在许多系统中，您需要使用根访问（root access）来使这个工作。下面是一个可以只使用用户级别权限来实现的方法。

第一次做这种事时，您需要定义一个脚本，在这里被称为 `oterm`，包含以下几行：

```
ps
sleep 600000
```

让它用 `chmod +x oterm` 执行。

现在：

- 从一个 shell（被称为 shell 1）开始，类型 `xterm -e oterm &`。您会看到一个新的窗口弹出，这将被称为终端 2。
- 从 shell 1 之内，`gdb julia-debug`。您将会在 `julia/usr/bin` 里找到这个可执行文件。
- 从 shell 1 之内，`(gdb) tty /dev/pts/#` 里面的 `#` 是在 terminal 2 中 `pts/` 之后显示的数字。
- 从 shell 1 之内，`(gdb) run`
- 从 terminal 2 之内，在 Julia 中发布任何准备好的您需要的命令来进入您想调试的步骤
- 从 shell 1 之内，按 Ctrl-C
- 从 shell 1 之内，插入您的断点，例如 `(gdb) b codegen.cpp:2244`
- 从 shell 1 之内，`(gdb) c` 来继续 Julia 的执行
- 从 terminal 2 之内，发布您想要调试的命令，shell 1 将会停在您的断点处。

在 emacs 之内

- `M-x gdb`，然后进入 `julia-debug`（这可以最简单的从 julia/usr/bin 找到，或者您可以指定完整路径）
- `(gdb) run`
- 现在您将会看到 Julia prompt。在 Julia 中运行任何您需要的命令来达到您想要调试的步骤。
- 在 emacs 的“Signals”菜单下选择 BREAK——这将会使您返回到 `(gdb)` prompt
- 设置一个断点，例如，`(gdb) b codegen.cpp:2244`
- 通过 `(gdb) c` 返回到 Julia prompt
- 执行您想要运行的 Julia 命令。

32

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
- 字面上的数字如果没有小数点，则会被默认为整数类型而不是浮点类型。且支持任意长度的整数类型。但是这也意味着一些如 `2^-1` 的表达式因为不是正式而抛出一个异常。
- 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
- 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
- Julia 的 type 类型和Matlab中的 classes 非常接近。Matlab 中的 structs 行为介于 Julia 的 types 和 Dicts 之间。如果你想添加一个域在 strut 中，使用 Dict 会比 type 好一些。

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` 和 `min` 等价于 R 语言中的 `pmax` 和 `pmin` 。但是所有的参数都应该有相同的维度。而且 `maximum`，`minimum` 可以替代 R 语言的 `max` and `min` ，这是最大的区别。
- 函数 `sum`，`prod`，`maximum`，`minimum` 和 R 语言中的同名函数并不相同。它们接收一个或者两个参数。第一个参数是集合，例如一个 array，如果有第二个参数，这个参数可以指明数据的维度，除此之外操作相似。比如，让 Julia 中的 `A=[[1 2],[3,4]]` 和 R 中的 `B=rbind(c(1,2),c(3,4))` 比较会是一个矩阵。 接着 `sum(A)` 和 `sum(B)` 会有相同的结果, 但是 `sum(A,1)` 是一个包含一列和的行向量，而 `sum(A,2)` 是一个包含行和的列向量. 如果第二个参数是向量，如 `sum(A,[1,2])=10` ， 需要确保第二参数没有问题。
- 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。

极客学院
jikexueyuan.com
中国最大的IT职业在线教育平台

更多信息请访问
http://wiki.jikexueyuan.com/project/julia-manual/