

Julia 学习笔记

翟羽

2021 年 4 月 24 日



目录

	4 程序结构	8
1 概览	2 5 变量作用域	10
1.1 工作环境	2	
1.2 输入-求值-打印循环 (REPL)	2	
1.3 基本包管理和安装第一个包 IJulia	2	
1.4 直接在终端运行 Julia 程序与 JIT 编译	3	
2 基本语法和类型	3	
2.1 文本与注释	3	
2.2 字面值常量与基本类型	4	
2.3 变量或名字	5	
3 更复杂的类型	5	
3.1 可索引的容器	5	
3.2 以数为内容的数组与线性代数	6	
3.3 字典	7	
3.4 复合类型	7	
3.5 字符串	7	
	6 模块	11
	7 操作符	12
	8 基本常用函数	13
	9 读写数据	14
	10 随机数	14
	11 统计与机器学习	14
	12 宏	14
	13 绘图	15
	14 处理表格数据	16
	15 扩展阅读	17

本文旨在通过实例向有其他编程语言经验的计算科学家介绍 Julia 语言，也可以作为介绍 Julia 语言的简短课程的讲义。

本文仅介绍 Julia 语言（作者认为的）最基本和最常用功能。读者应该能理解作者的工作背景和读者的潜在（巨大）差异，在阅读本文之外请积极参考其他资料。在本文第15节列出了较为重要的参考资料，供读者进一步学习。

欢迎一切建设性建议，请在 github issues (<https://github.com/zhaiyusci/Julia-Notes/issues>) 中提出。

1 概览

Julia 是一种为科学计算设计的编程语言。相比 C/C++、FORTRAN 等语言，其动态语言的特性和方便的包管理机制极大地降低了其开发痛苦；相比 Python、R 等语言，其编译型语言的特性和出色的性能优化保证了其执行效率；相比 MATLAB 等商业语言，Julia 容易获得，能够在各种工作环境轻松部署。

1.1 工作环境

请访问 Julia 语言官网 <https://julialang.org/> 下载适合您操作系统版本的 Julia 安装包。Julia 的安装过程和一般的软件相似，在此不做过多说明。对于 Windows 用户，您可以在 Windows Subsystem for Linux (WSL) 中安装使用。如果您在公用服务器中学习 Julia，请咨询您的系统管理员。

本文在以下版本的 Julia 下测试通过（您可以在您的 Julia 会话中运行 `versioninfo()` 检查版本）：

```
1 Julia Version 1.6.0
2 Commit f9720dc2eb (2021-03-24 12:55 UTC)
3 Platform Info:
4   OS: Linux (x86_64-pc-linux-gnu)
5   CPU: AMD Ryzen 7 PRO 4750U with Radeon Graphics
6   WORD_SIZE: 64
7   LIBM: libopenlibm
8   LLVM: libLLVM-11.0.1 (ORCJIT, znver2)
```

您的版本可能和本文所述有所不同。但对于本文的大部分内容，这应该不会导致什么问题。如果有，请访问文档的 [github](#) 页面提 issue。

1.2 输入-求值-打印循环 (REPL)

通过本文档学习 Julia 的最简单方法是将它们输入到 Julia 的输入-求值-打印循环 (REPL) 里。在系统的终端中键入 `julia` 并回车将进入 REPL。在 REPL 您可以实时和 Julia 编译器交互。在 REPL 中按 `^D` (`Ctrl-D`) 可以退出 REPL，按 `^C` 可以中断计算。

1.3 基本包管理和安装第一个包 IJulia

刚安装好的 Julia 编译器仅包含很少一部分包 (package)。Julia 社区中其他成员的成果往往以包的形式发布供大家使用。若您需要安装包 `PackageX`，可以在 Julia REPL 中按下 `]` 进入包管理器，然后键入 `add PackageX` 以安装该包。在网络畅通的状态下，安装过程不需要用户过多操心。Julia 的包管理机制会自动下载、安装该包及其所有依赖。¹ 当然您可以把 `PackageX` 换成您需要的包的名称。在包管理器中，还有 `status` (状态)、`remove` (移除)、`update` (升级) 等命令可供使用。按退格键可以退出包管理模式。

例如，另一个可以实时与 Julia 交互的方式是 Jupyter Notebook。在 Julia 的 REPL 中键入 `]` 然后键入 `add IJulia`。安装完毕后，退出包管理模式，键入 `using IJulia; IJulia.notebook()` 即可在默认浏览器中打开 Jupyter。²

本文将不会对 Jupyter 做过多说明，而默认您了解其基本用法。³ 本文的第13节会讨论一些基本的科学绘图，因此 Jupyter 是一个更合适的环境。

¹有时网路极不畅通，您可能需要代理 (proxy)。请参考 <https://github.com/zhaiyusci/Julia-Notes/issues/1>。

²WSL 用户需要做更多事情。请参考 <https://github.com/jupyter/notebook/issues/4594>。

³如果您原来用 FORTRAN 或 C，或者 Julia 是您的第一门语言，您可能不了解 Jupyter。鉴于 Jupyter 是图形界面，您可能愿意观看一些视频教程来学习其基本用法。目前 Jupyter 的中文视频教程大多是结合 Python 的，但这并不妨碍您通过这些资料了解 Jupyter 本身。

1.4 直接在终端运行 Julia 程序与 JIT 编译

你可以在系统 shell 里通过 `julia script.jl` 运行 Julia 脚本。Julia 将“逐行执行”这些内容。但请注意，Julia 并没有逐行解释（interpret）这些代码，而是**实时编译**（JIT）。

将下列示例脚本保存至文件中并执行（在以后的章节中有更多例子）：

```
1 using LinearAlgebra      # 使用线性代数库
2 f(n)=rand(n)·rand(n)    # 定义函数 f
3 for i in 1:3             # 执行 3 次
4     @time @show f(100)   # 显示结果与时间
5 end
```

您会得到类似这样的输出：

```
1 f(100) = 25.047134233497236
2     0.028204 seconds (27.25 k allocations: 1.405 MiB)
3 f(100) = 24.53621754676729
4     0.000029 seconds (11 allocations: 2.422 KiB)
5 f(100) = 24.16289030865225
6     0.000019 seconds (11 allocations: 2.422 KiB)
```

注意到头一次的执行时间（第 2 行）远大于后两次（第 4、6 行）。这是因为 Julia 会在函数第一次执行前编译该函数，因此第一个执行时间实际上包含了函数的编译时间。Julia 的 JIT 特性是值得一般使用者注意的。简单来说，不建议采用“写脚本”的方式去写 Julia 代码；相反，Julia 鼓励您把最常用的功能包装成函数，反复调用。⁴

2 基本语法和类型

2.1 文本与注释

Julia 的源文件是 **Unicode** 文本文件。这在上一个例子中已经有所展示。因此，在实际工作中，您需要采用一个“现代”的文本编辑器和终端和“现代”的字体来显示这些特殊符号。您自己写程序的时候当然可以仅用美式键盘上的那些符号。但是诸如 $\pi \cdot \alpha \beta \approx \times \Delta$ 等符号可以极大的改善程序的可读性，使代码和公式尽量一致。编辑器的 Unicode 输入支持可能需要插件。⁵

在 Julia 语言中，注释有以下两种写法：

```
1 # 这里是注释
2 #=
3 这里全是注释
4 依然是注释
5 这个结构叫块注释
6 =#
```

Julia 中区分大小写，Code、code和c0dE 不一样。

⁴因此，Julia 有较严重的“首次执行问题”：在使用较大的 Julia 包时（例如绘图包），第一次执行十分耗时。请参考 <https://julialang.github.io/PackageCompiler.jl/dev/sysimages/> 改善体验。

⁵您可以用那些“流行”的编辑器。请参考<https://github.com/JuliaEditorSupport>、<https://junolab.org>、<https://www.julia-vscode.org/>，获得您熟悉的编辑器的插件，乃至将您的编辑器改造成集成开发环境（IDE）。至于字体，推荐 DejaVu Sans Mono 和 JuliaMono（至少，它们支持更多的 Unicode 字符）。

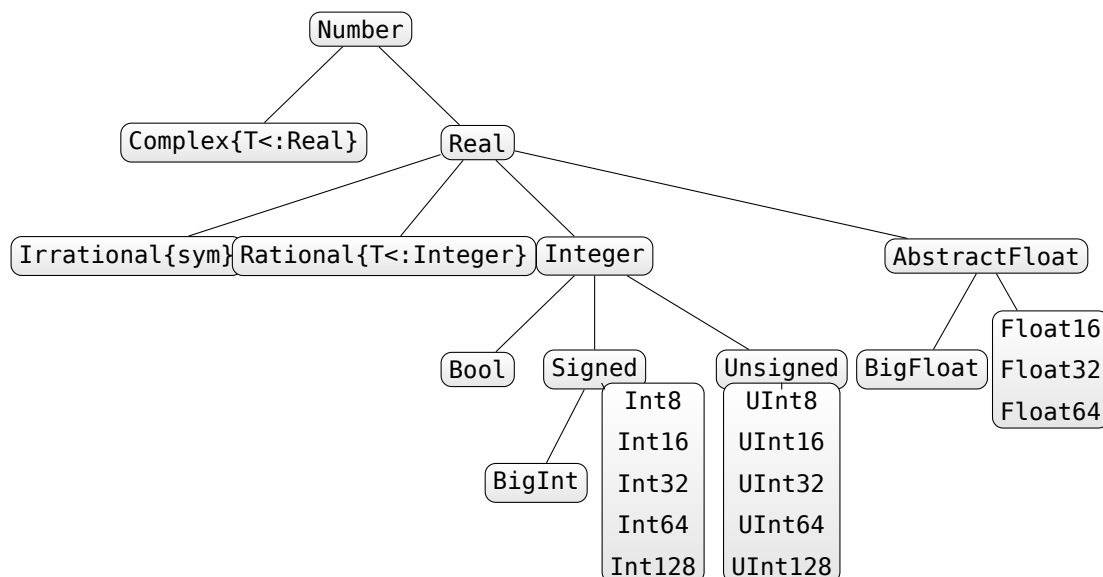


图 1: 数值类型的层级结构

2.2 字面值常量与基本类型

您可以直接在程序中写**字面值** (literal) 常量。语法 `1::Int` 指字面值 `1` 且断言其为 `Int` 类型。通常，类型断言并非必要，Julia 会自动推断某字面值为何种类型。注意类型断言没有类型转化的功能。将 `x` 转为 `T` 类型的最简单方法是 `T(x)`。字符串的解读可通过 `parse(Type, str)` 完成。多个参数自动提升为相同类型（若可行）可用 `promote` 达成，许多操作（算术、赋值）会自动进行类型提升。

```

1  1                # 整数，在 64 位系统中等价于 Int64
2  1.0              # 64 位浮点数
3  true             # 布尔型，可以是 true 或 false
4  'c'              # 字符，可以是 Unicode
5  "abc"            # 抽象字符串，可以有 Unicode，参见下述字符串 (String)
6
7  1::Int            # 正确，1 确实是 Int 类型
8  1::Float          # 错误，1 不是 Float 类型
9  Float64(1)        # 将 1 转化为浮点数的正确方式
10 Int64(1.2)        # 试图将 1.2 转化为整数，但会报错
11
12 parse{Int64}("1") # 以 Int64 类型解读字符串 "1"
13 promote(true, BigInt(1)//3, 1.0)
14                  # BigFloat 类型元组（参见 Tuple），true 转换为 1.0

```

可通过如下方式查验值的类型：

```

1  typeof("abc")      # 返回 String，String 是 AbstractString 的子类型
2  isa("abc", AbstractString) # true
3  isa(1, Float64)    # false，1 是整数而不是浮点数
4  isa(1.0, Float64)  # true
5  1.0 isa Number     # 另一种语法；true，Number（数）是一种抽象类型
6  supertype{Int64}    # Int64 的父类型
7  subtypes{Real}      # 抽象类型 Real 的子类型

```

全部标准数值类型的层级结构见图 1。

Julia 当然支持常见的数学运算和函数，我们可以把 Julia REPL 当作计算器使用。

```
1 2+1.2*3-4^3/5          # -7.200000000000001
2 3*π                     # 3π, Julia 支持 Unicode
3 log(MathConstants.e)    # 1
4 2im^2                   # -4+0im, 对复数的支持, im 是虚数单位
```

Julia 遵循基本的算术运算优先顺序，以及一些省略乘号的语法糖，但依然推荐您在任何有必要情况加括号。

2.3 变量或名字

上述常量都可以存到变量中，例如 `x=3.4`。

在 Julia 中，变量应该被简单地理解为“名字”。变量只是一笔数据的代号，本身并不和特定的类型有一对一的关系，这和 Python 等动态语言一样，而和 C、FORTRAN 等语言不同。因此，变量可以先后和不同类型的值绑定。

```
1 x=3                      # 3
2 typeof(x)                # Int64
3 x=3.3                    # 3.3
4 typeof(x)                # Float64
5 x=3.3+3.3im              # 3.3 + 3.3im
6 typeof(x)                # ComplexF64 (alias for Complex{Float64})
```

3 更复杂的类型

除基本类型以外，Julia 原生提供了多种实用的**复合类型**。例如，上述复数即为复合类型。除此之外，还有几种重要的复合类型。

3.1 可索引的容器

元组 (Tuple) 是不可变的序列，而**数组** (Array) 是可变的序列。而**可索引**是指可以用方括号来获得序列中的具体数值。在 Julia 中，索引从 1 开始数，这和 FORTRAN、Matlab 等一致，而和 C 不一样。此外，元组和数组还有各种细微的区别，见下面的例子。

```
1 a=(1,2,3.0, "abc") # 用圆括号构造元组
2 typeof(a)          # Tuple{Int64, Int64, Float64, String}
3                    # 不同类型可以共处
4 a[2]                # 第二个元素是 2
5 a[2]=20             # 报错，元组内容不可变
6
7 b=[1,2,3.0,"abc"]  # 用方括号构造数组
8 typeof(b)           # Vector{Any} (alias for Array{Any, 1})
9                    # 各类型被提升为共同的父类型
10                   # 在 Julia 中，所有类型共同的父类型是 Any
11 b[2]="bcd"         # 可以改变数组的内容
12 b                 # [1, "bcd", 3.0, "abc"]
```

根据作者的开发经验，元组（及命名元组）更多用于函数与外部的交互。故更多实例请见第4节。

3.2 以数为内容的数组与线性代数

以数为内容的数组可以理解为线性代数中的矢量和矩阵。这部分是科学计算中的重点。

```

1 using LinearAlgebra          # 引入线性代数库
2 x=zeros(10)                  # 10 个元素的零矢量，列矢量
3 a=ones(3)                    # 3 个元素的矢量，全填充 1.0，列矢量
4 H=[1.0 2.0 3.0
5     4.0 5.0 6.0
6     7.0 8.0 9.0]
7 H[2,1]                       # 4.0
8 ndims(H)                     # H 的维度，2
9 eltype(H)                    # H 中元素的类型，Float64
10 length(H)                   # H 中元素数目，9
11 size(H)                     # H 每一维度的大小所构成的元组，(3,3)
12 sum(H, dims=2)               # 对第2维求和，相似地，该用法可以用于以下函数： mean, std,
13                               # prod, minimum, maximum, any, all
14                               # 使用统计函数前需要 using Statistics
15 H[:,1]                      # 矩阵的第1列
16 H[1,:]                      # 矩阵的第1行（仍是列矢量）
17 Hc=H; Hc[1,1]=100.0; H[1,1] # 100.0，数组的赋值是按引用复制
18 Hc=copy(H)                  # 这样才是真的复制一个新的矩阵
19 H*a                         # 矩阵乘法
20 exp(H)                      # 矩阵的函数
21 a*a                         # 出错的矩阵乘法
22                               # 列矢量和列矢量的乘法行列不对应无法进行
23 a·a                         # 但是可以做点积
24                               # · 用 \cdot 可以打出来（在 REPL、Jupyter 中）
25                               # 或者借助插件（见前文开发环境部分）
26 a×a                         # 还可以做叉积
27 sum(a)
28 H'                          # 共轭转置，相当于 adjoint(H)
29 typeof(H')                  # 注意共轭转置是惰性求值
30                               # 它并不直接分配一块内存并写入计算的结果
31                               # 这对计算效率有很大的帮助
32 G=rand(1000,1000); G=G+G'   # 构造一个实对称阵
33 eigvals(G)                  # 这样的小矩阵，我们可以直接求本征值
34 eigvecs(G)                  # 本征矢

```

此外，Julia 支持稀疏矩阵和分布式矩阵。分布式矩阵主要用于分布式计算中，本文不做介绍。稀疏矩阵在科学计算中具有举足轻重的地位。如果您的矩阵足够大又足够稀疏，且仅关心部分本征对，应该考虑用稀疏矩阵。

```

1 using SparseArrays, Arpack    # 引入稀疏矩阵库和 Arpack 数学库的接口
2 Ksp=Symmetric(sprand(5000,5000,0.01)) # 生成一个对称的稀疏矩阵
3                               # 注意，您的内存可能不够大，5000 可以适当调小
4 Kde=Symmetric(Matrix(K))     # 对应的稠密矩阵
5 @time eigen(Kde)              # 计算本征对，计时

```

```
6 # 19.264927 秒, 您的时间大概和这里不同
7 @time Arpack.eigs(Ksp, nev=100)      # 计算前 100 个本征对
8 # 1.389663 秒
```

3.3 字典

字典 (Dict) 是一种关联容器 [键 (key) - 值 (value) 对]:

```
1 x = Dict{Float64, Int64}()          # 将浮点数映射 (map) 为整数的空字典
2 y = Dict{"a"=>1, "b"=>2}            # 非空字典
3 y["a"]                              # 访问元素
4 y["c"]                              # 错误
5 y["c"] = 3                          # 添加元素
6 haskey(y, "b")                      # 检查 y 是否有 "b" 键
7 keys(y), values(y)                 # y 中的键和值
8 delete!(y, "b")                    # 从字典中删除一个键, 参见 pop!
9 get(y, "c", "default")              # 返回 y["c"], 如果 y["c"] 不存在, 返回 "default"
```

3.4 复合类型

你可以定义、访问复合类型**复合类型**。以下是**可变** (mutable) 复合类型的一个实例:

```
1 mutable struct Point
2     x::Int64
3     y::Float64
4     meta
5 end
6 p = Point(0, 0.0, "Origin")
7 p.x          # 访问域
8 p.meta = 2    # 改变域的值
9 p.x = 1.5     # 错误, 数据类型不匹配
10 p.z = 1       # 错误, 没有这个域
11 fieldnames(Point) # 获取所有的域名
```

类似地, 你可以通过删除 `mutable` 关键字来定义**不可变**复合类型。

你可以将你的类型定义为某抽象类型的子类型以将其适当的放在类型层级树中, 你也可以定义自己的抽象类型。

3.5 字符串

字符串 (string) 操作有:

```
1 "Hi " * "there!"      # 字符串连接
2 "Ho " ^ 3             # 重复字符串
3 string("a = ", 123.3) # 使用打印方式构造
4 repr(123.3)           # 返回 show 函数的输出
5 occursin("CD", "ABCD") # 检查第一个字符串是否包含于第二个中
6 "\\n\\t\\$"           # C 语言风格的转义 (escaping) 字符串 (注意 \\$ 的转义)
```



```
7 x = 123
8 "$x + 3 = ${x+3}"      # 没有转义的 $ 用来原位求值 (interpolation)
9 "\$199"                 # 要表示 $ 符号, 你必须转义
10 raw"D:\path"           # 原始 (raw) 字符串字面值, 在表示 Windows 下的路径很有用
11 s = "abc"              # 类型为 String 的字符串
12 chop(s)                # 删除 s 的最后一个字符, 返回一 SubString 类型对象
13 r = r"A|B"             # 创建新的正则表达式 (语法同 Perl)
14 occursin(r, "CD")      # false, 不匹配
15 m = match(r, "ACBD")   # 找到第一处正则表达式匹配
```

String 和 SubString 皆为 AbstractString 的子类型。SubString 可避免复制字符串。通常, 当写自己的程序时, 应假设用户会使用 AbstractString。

注意! 请避免索引字符串, 例如 " $\Delta x a 1 = \Omega$ "[3] 会是 'x' 而不是我们期待的 'a'。这是因为 Julia 是用 UTF-8 来编码标准字符串的, 而索引是按字节而不是字符, 所以你需要理解 UTF-8 编码才能正确地索引。

4 程序结构

在逐行顺序执行所有语句外, 可以用流程控制结构来控制执行的顺序。常见的流程控制结构有 if、while、for 等, 它们以块的形式出现。在 Julia 中, 块总是以 end 结束, 而有不同的开头。

```
1 if false                # if 从句需要 Bool 测试
2     z = 1
3 elseif 1 == 2
4     z = 2
5 else
6     a = 3
7 end                      # 此后 a = 3, z 未定义
8
9 1==2 ? "A" : "B"        # 标准三元操作符
10
11 i = 1
12 while true
13     i += 1
14     if i > 10
15         break
16     end
17 end
18
19 for x in 1:10            # x 在区间中, 这里可以用 = 代替 in
20     if 3 < x < 6
21         continue        # 跳过一次迭代
22     end
23     println(x)
24 end                      # x 在此循环的内部作用域被定义
25
26 [10i+j for i in 1:3, j in 1:3] # 隐式循环构造矩阵
```

表达式可以用 ; 或者 begin end 块结合在一起:


```
1 x = (a = 1; 2 * a) # 此后 x = 2; a = 1
2 y = begin
3     b = 3
4     3 * b
5 end                # 此后 y = 9; b = 3
```

你可以定义自己的函数 (function):

```
1 f(x, y = 10) = x + y          # 新函数 f 的单行定义, y 默认值是 10
2                               # 返回最后一个表达式的结果
3 function f(x, y=10)           # 和上面一样, 但允许多个表达式
4     x + y                     # 在函数体内
5 end
6 f(3, 2)                      # 简单的调用, 返回 5
7 f(3)                          # 返回 13
8 (x -> x^2)(3)                 # 匿名函数的调用
9 () -> 0                       # 没有参数的匿名函数
10 h(x...) = sum(x)/length(x) - mean(x) # 可变参数的函数; x 是元组; 需要 using Statistics
11 h(1, 2, 3)                   # 结果为 0
12 x = (2, 3)                   # 元组
13 f(x)                         # 错误: 我们企图把10加到 (2, 3) 上
14 f(x...)                     # 正确, 元组解包
15 s(x; a = 1, b = 1) = x * a / b # 有关键词参数 a 和 b 的函数
16 s(3, b = 2)                  # 带关键词参数的调用
17 q(f::Function, x) = 2 * f(x) # 函数可以当作参数传递; 这里我们要求 f 是一个函数
18 q(x -> 2x, 10)               # 返回 40, 不必在 2x 中间写乘号 (代表 2*x)
19 q(10) do x                   # 通过 do 结构创建匿名函数, 例如在输入输出有用
20     2 * x
21 end
22 m = reshape(1:12, 3, 4)
23 map(x -> x ^ 2, m)           # 包含转换过数据的 3x4 数组
24 filter(x -> bitstring(x)[end] == '0', 1:12) # 在区间里筛选偶数的亮眼方法
25 ==(1)                       # 返回一个检查是否等于1的函数
26 findall(==(1), 1:10)         # 找到所有等于 1 的元素的索引, 相似地: findfirst, findlast
```

习惯上以 ! 结尾地函数会原位修改其参数。参见本文档中的 `resize!`。

默认函数参数自左向右求值:

```
1 y = 10
2 f1(x=y) = x; f1()            # 10
3 f2(x=y,y=1) = x; f2()        # 10
4 f3(y=1,x=y) = x; f3()        # 1
5 f4(;x=y) = x; f4()           # 10
6 f5(;x=y,y=1) = x; f5()       # 10
7 f6(;y=1,x=y) = x; f6()       # 1
```

Julia 中函数可以有多个方法 (method)。每个方法对应函数的一组参数类型。这种行为叫多重派发 (multiple dispatch) 并且只对位置 (positional) 参数有效。以下是一些简短的例子。详见 Julia 的 Methods 章节。

```
1  g(x, y) = println("all accepted") # g 函数接收任何类型 x 和 y 的方法
2  function g(x::Int, y::Int)        # 当 x 和 y 均为 Int 型调用的方法
3      y, x
4  end
5  g(x::Int, y::Bool) = x * y        # 当 x 为 Int, y 为 Bool 时的方法
6  g(1.0, 1)                        # 第一个定义被调用
7  g(1, 1)                          # 第二个定义被调用
8  g(1, true)                       # 第三个定义被调用
9  methods(g)                       # 列出 g 的所有方法
10 t(; x::Int64 = 2) = x             # 单个关键词参数
11 t()                              # 返回 2
12 t(; x::Bool = true) = x          # 没有对关键词参数的多重派发, 函数定义被覆盖
13 t()                              # true; 旧函数被覆盖
```

5 变量作用域

以下结构创建新的变量作用域 (variable scope) : `function`、`while`、`for`、`try/catch`、`let`、`struct`、`mutable struct`。

你可以将变量定义为:

- `global`: 在当前模块的全局作用域使用变量;
- `local`: 在当前作用域定义新变量;
- `const`: 确保变量类型是常量 (仅全局)。

特殊情况:

```
1  t                                # 错误, 变量 t 不存在
2  f() = global t = 1
3  f()                              # 调用后 t 在全局作用域被定义
4
5  function f1(n)
6      x = 0
7      for i = 1:n
8          x = i
9      end
10     x
11 end
12 f1(10)                           # 10; 在循环内我们使用了外部局域变量
13
14 function f2(n)
15     x = 0
16     for i = 1:n
17         local x
18         x = i
19     end
20     x
```

```
21 end
22 f2(10)          # 0; 在循环内我们使用了新的局域变量
23
24 function f3(n)
25     for i = 1:n
26         h = i
27     end
28     h
29 end
30 f3(10)          # 错误; h 在外部作用域未定义
31
32 const x = 2
33 x = 3 # 警告, 常量的值被改变; 但你永远不应该这样写, 这会导致编译好的代码失效
34 x = 3.0 # 错误, 类型错误
35
36 function f()
37     x::Int = 1
38     x = 2.5 # 当调用 f() 时会报错, 因为 x 必须是 Int
39 end
```

全局常量会加速代码执行, 因为编译器知道其类型。

循环和推导在每次迭代时都重新给变量赋值, 故可以安全地在迭代中用他们创造闭包 (closure):

```
1 Fs = Array{Any}(undef, 2)
2 for i in 1:2
3     Fs[i] = () -> i
4 end
5 Fs[1](), Fs[2]() # (1, 2)
```

6 模块

模块 (module) 将代码封装起来, 每个模块有他们自己的全局命名空间 (Julia REPL 的模块名是 Main)。

```
1 module M # 模块名
2 export x # 模块向外界所暴露的
3 x = 1
4 y = 2 # 隐藏变量
5 end
6
7 varinfo(M) # 列出导出的变量
8 x          # 在全局作用域中找不到
9 M.y        # 可以直接访问变量
10
11 # 导入所有导出的变量
12 # 也可以用此法加载标准包, 但不需要 . 前缀
13 using .M
14
```

```
15 # 将 y 导入全局作用域 （即使其没有被导出）
16 import .M.y
```

给其他模块中的变量绑定值是不可以的。如下是常常让人们惊讶的经典小例子：

```
1 sin(1) # 成功
2 sin = 1 # 失败，在模块 Main 里你不能给模块 Base 里的变量绑定值
3 cos = 1 # 成功，因为cos还没有被调用，故还没有从 Base 导入到 Main
4 cos # 返回 1
5 cos(1) # 失败，cos 在 Main 模块绑定了 1
6 Base.cos(1) # 成功
```

7 操作符

Julia 使用标准的操作符，而有如下的小技巧：

```
1 true || false # 二元或算符（仅单元素使用），|| 和 && 采短路求值
2 [1 2] .& [2 1] # 逐位 (bitwise) 与算符 [以 . 向量化 (vectorize) ]
3 1 < 2 < 3 # 支持连锁条件 (chaining conditions) （仅单元素使用，不可用 . 向量化）
4 [1 2] .< [2 1] # 向量化的操作符需要前缀 .
5 x = [1 2 3]
6 2x + 2(x .+ 1) # 字面值 and 变量间的乘号及字面值或变量与左括号间的乘号可省略
7 y = [1, 2, 3]
8 x + y # 错误：维度不匹配
9 x .+ y # 3x3 矩阵，维度广播 (broadcasting)
10 x + y' # 1x3 矩阵
11 x * y # 数组乘法，单元素矢量（不是标量）
12 x .* y # 逐元素 (element-wise) 乘法，3x3 数组
13
14 x == [1 2 3] # true，对象值一样
15 x === [1 2 3] # false，对象并非同一
16
17 z = reshape(1:9, 3, 3)
18 z + x # 错误：维度不匹配
19 z .+ x # x 纵向广播
20 z .+ y # y 横向广播
21
22 # 长度为1的维度的显式广播
23 # 对每一个数组元素，函数 + 都被调用
24 broadcast(+, [1 2], [1; 2])
25 # 使用 . 操作符广播
26 using Random
27 length([randstring(10) for i in 1:5]) # 5: 数组的长度
28 length.([randstring(10) for i in 1:5]) # 由 10 构成的 5个元素的数组：字符串的长度
```

函数广播的例子：

```
1 t(x::Float64, y::Float64 = 1.0) = x * y
```

```
2 t(1.0, 2.0) # 成功
3 t([1.0 2.0]) # 错误
4 t.([1.0 2.0]) # 成功
5 t([1.0 2.0], 2.0) # 错误
6 t.([1.0 2.0], 2.0) # 成功
7 t.(2.0, [1.0 2.0]) # 成功
8 t.([1.0 2.0], [1.0 2.0]) # 成功
9 t.([1.0, 2.0], [1.0 2.0]) # 成功
```

8 基本常用函数

```
1 show(collect(1:100)) # 显示对象的字符串表示
2 eps() # 从 1.0 到下一个可表示的 Float64 类型数的差距 (译注: 机器相对误差)
3 nextfloat(2.0) # 下一个可表示的浮点数, 类似地, 提供了 prevfloat
4 isequal(NaN, NaN) # true
5 NaN == NaN # false
6 NaN === NaN # true
7 isequal(1, 1.0) # true
8 1 == 1.0 # true
9 1 === 1.0 # false
10 0.0 == -0.0 # true
11 0.0 === -0.0 # false
12 isfinite(Inf) # false, 类似地提供了 isinf, isnan
13 fld(-5, 3), mod(-5, 3) # (-2, 1), 朝向负无穷的整数除法
14 div(-5, 3), rem(-5, 3) # (-1, -2), 朝向零的整数除法
15 findall(x -> mod(x, 2) == 0, 1:8) # 找到使函数为真的索引
16 x = [1 2]; identity(x) === x # true, 返回参数本身
17 @info "Info" # 打印信息, 类似地有 @warn 和 @error (参见 Logging 模块)
18 ntuple(x->2x, 3) # 通过以值 1、2、3 调用 x->2x 创建元组
19 @isdefined x # 变量 x 是否被定义
20 y = Array{Any}(undef, 2); isassigned(y, 3) # 数组中的第3位是否被赋值 (而不是越界或者 #undef)
21 fieldtype(typeof(1:2), :start) # 获取复合类型中域的类型 (以符号传参)
22 fieldnames(typeof(1:2)) # 获取类型中域的名称
23 zip(1:3, 1:3) |> collect # 将迭代器打包为迭代器元组, 并且将其传给 collect 函数
24 enumerate("abc") # 创建能够遍历由索引和内容元素构成的元组的迭代器
25 collect(enumerate("abc")) # 并将其转换为数组
26 isempty("abc") # 检查一个组合是否为空; 字符串视为字符的组合
27 'b' in "abc" # 检查元素是否在组合里
28 indexin(collect("abc"), collect("abrakadabra")) # [1, 2, nothing] ('c' 没有找到), 需要数组
29 findall(in("abrakadabra"), "abc") # [1, 2] ('c' 没有找到)
30 unique("abrakadabra") # 舍去重复的元素
31 issubset("abc", "abcd") # 检查是否第一个组合里的每一个元素都在第二个组合里 (译注: 判断是否为子集)
32 argmax("abrakadabra") # 最大元素的索引 (3: 对应 'r')
33 findmax("abrakadabra") # 最大元素及其索引构成的元组
34 filter(x->mod(x,2)==0, 1:10) # 保留一个组合里满足条件的元素
35 dump(1:2:5) # 显示该对象所有对用户可见的结构
```

```
36 sort(rand(10)) # 对 10 个随机数排序, sort! 可进行原位排序
```

9 读写数据

有关输入输出的细节, 见 Julia 文档。有大量的包提供了该功能。DelimitedFiles 模块中的基本操作有:

- readlm: 从文件读
- writelml: 写入文件

警告! 如果文件读取时设置 `delim=' '`, 则空格不会被忽略。

10 随机数

基本的随机数用法:

```
1 Random.seed!(1) # 设定随机数种子为 1; 需要先调用 using Random
2 rand()          # 在 U[0,1) 范围内生成随机数
3 rand(3, 4)      # 在 U[0,1) 范围内生成 3x4 随机数矩阵
4 rand(2:5, 10)   # 在 2 到 5 范围内生成 10 个随机整数构成的数组
5 randn(10)       # 依正态分布生成 10 个随机数构成的数组
```

Distributions 包中的进阶随机数:

```
1 using Distributions # 加载包
2 sample(1:10, 10)    # 从集合 1 到 10 中的随机取样
3 b = Beta(0.4, 0.8)  # 参数为 0.4 和 0.8 的 beta 分布
4                     # 支持的分布详见文档
5 mean(b)             # b 分布的期望值
6                     # 支持的统计详见文档
7 rand(b, 100)        # b 分布下 100 个独立随机取样
```

11 统计与机器学习

访问 <http://juliastats.github.io/> 获取更多信息 (尤其是类似于 R 语言的数据框架)。

Julia 中的一个核心语言结构 Missing 专门用于表示缺失的值。

```
1 missing # 缺失的值
2 ismissing(missing) # true
3 coalesce(missing, 1, 2) # 返回第一个没有缺失的值, 当所有值都缺失的时候返回 missing
```

12 宏

你可以定义宏 (macro) (详见文档)。有用的标准宏有:

断言:

```

1  @assert 1 == 2 "ERROR"           # 2 个宏参数, 引发错误
2  using Test                       # 加载 Test 包
3  @test 1 == 2                     # 与 assert 类似; 错误 similar to assert; error
4  @test_throws DomainError sqrt(-1) # 通过, 不可以使用 sqrt(-1)

```

基准测试:

```

1  @time [x for x in 1:10^6];      # 打印出所用时间和内存
2  @timed [x for x in 1:10^6];     # 返回值、时间和内存
3  @elapsed [x for x in 1:10^6]   # 返回时间
4  @allocated [x for x in 1:10^6] # 返回内存

```

使用 BenchmarkTools 包以获得更强大的基准测试功能。

13 绘图

Julia 有几个绘图包, 例如 Plots (涵盖了多个绘图后端的接口包) 以及 PyPlot. 这里我们介绍 Python 用户会觉得似曾相识的 PyPlot.

```

1  using PyPlot
2  using Random
3  Random.seed!(1) # 使绘图可重现
4  x, y = randn(100), randn(100)
5  scatter(x, y)

```

我们从https://matplotlib.org/1.2.1/examples/pylab_examples/histogram_demo.html借用一个例子:

```

1  using Distributions
2  using PyPlot
3
4  mu, sigma = 100, 15
5  x = mu .+ sigma * randn(10000)
6
7  n, bins, patches = plt.hist(x, 50, density=1,
8                             facecolor="green", alpha=0.75)
9
10 y = pdf.(Ref(Normal(mu, sigma)), bins);
11 plot(bins, y, "r--", linewidth=1)
12
13 xlabel("Smarts")
14 ylabel("Probability")
15 title(raw"\mathrm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$")
16 axis([40, 160, 0, 0.03])
17 grid(true)

```

产生:

14 处理表格数据

Julia 里有多个包支持表格数据。

这里我们介绍 DataFrames (0.21 版) 和 CSV 两个包。

加载一个逗号分隔表 (CSV) 文件:

```
1 using DataFrames
2 using CSV
3 path = joinpath(dirname(pathof(DataFrames)), "../docs/src/assets/iris.csv")
4 df = DataFrame(CSV.File(path));
5 first(df, 5) # 打印数据中的前5行; 使用 last 函数来打最后几行
```

生成如下的输出:

```
1 4x5 DataFrame
2 | Row | SepalLength | SepalWidth | PetalLength | PetalWidth | Species |
3 |     | Float64      | Float64     | Float64     | Float64     | String   |
4 |-----+-----+-----+-----+-----+-----|
5 | 1   | 5.1          | 3.5          | 1.4          | 0.2          | Iris-setosa |
6 | 2   | 4.9          | 3.0          | 1.4          | 0.2          | Iris-setosa |
7 | 3   | 4.7          | 3.2          | 1.3          | 0.2          | Iris-setosa |
8 | 4   | 4.6          | 3.1          | 1.5          | 0.2          | Iris-setosa |
```

读入 DataFrame 后, 让我们介绍几条处理它的最有用的几个操作:

```
1 DataFrame(a=1:10, b=rand(10)) # 由几组数据手动生成 DataFrame
2 describe(df) # 获取数据框架的概要信息
3 df.Species # 获取 df 中的 Species 字段而不进行复制操作
4 df[:, :Species] # 同上
5 df[1, 5] # 从数据框架中获取第1行第5列的值
6 df[1:2, 1:2] # 创建数据框架的子集, 由前两行的前两列构成
7 Matrix(df[:, 1:4]) # 将第1到第4列转换为矩阵
8 names(df) # 以字符串形式获取数据框架中的字段名
9 nrow(df), ncol(df) # 数据框架的行数和列数 number of rows and columns in a data frame
10 sort(df, :SepalWidth) # 返回一个以 SepalWidth 排序的新数据框架
11 filter(:SepalWidth => >(3), df) # 返回一个仅包含满足预设条件的行的新数据框架
12 push!(df, (1, 2, 3, 4, "Some species")) # 在数据框架的末尾加一新行
13 df.key = axes(df, 1) # 向数据框架中添加一个新变量 key
14
15 # 按 Species 计算 SepalLength 的总和, 并将结果存在 x 中
16 combine(groupby(df, :Species), :SepalLength => sum)
17
18 # 把 df 转变成以 SepalLength 为值而 key 与 Species 对为 id 的长格式
19 df2 = stack(df, :SepalLength, [:key, :Species])
20 unstack(df2) # 相反的操作: 宽到长格式
```

15 扩展阅读

请参阅 Julia 最新版文档 <https://docs.julialang.org/>⁶ 以了解这些内容。

⁶译注：亦可参考完善中的中文文档 <https://docs.juliacn.com/latest/>，但请以英文原文为准。