中文(简体)

快速入门

一份简单而粗略的语言概览

julia 1.0

页面的源文件放在 Github 上 . 欢迎提交改进!

...是什么?

Julia 是一种为科学计算而生的，开源、多平台、高性能的高级编程语言。

Julia 有一个基于 LLVM 的 JIT 编译器，这让使用者无需编写底层的代码也能拥有像 C 与 FORTRAN 那样的性能。因为代码在运行中编译，你可以在 shell 或 REPL 中运行代码，这也是一种推荐的工作流程。

Julia 是动态类型的。并且提供了为并行计算和分布式运算设计的多重派发机制。

Julia 自带包管理器。

Julia 有许多内置的数学函数，包括特殊函数 (例如：Gamma 函数)。并且支持开箱即用的复数运算。

Julia 允许你通过类似 Lisp 的宏来自动生成代码。

Julia 诞生于 2012 年。

基础语法

```
answer = 42
x, y, z = 1, [1:10; ], "A string"
x, y = y, x     # 交换 x, y
```
赋值语句

常量定义
```
const DATE_OF_BIRTH = 2012
```

行尾注释
```
i = 1           # 这是一行注释
```

多行注释
```
#= 这是另一行注释 =#
```

链式操作
```
x = y = z = 1   # 从右向左
0 < x < 3       # true
5 < x != y < 5  # false
```

函数定义
```
function add_one(i)
    return i + 1
end
```

插入 LaTeX 符号 `\delta` + [Tab] `# δ`

运算符

基本算数运算	`+, -, *, /`
幂运算	`2^3  # 8`
除法	`3/12  # 0.25`
反向除法	`7\3 == 3/7 # true`
取余	`x % y` 或 `rem(x,y)`
取反	`!true  # false`
等于	`a == b`
不等于	`a != b` 或 `a ≠ b`
小于与大于	`<` 与 `>`
小于等于	`<=` 或 ≤
大于等于	`>=` 或 ≥
逐元素运算(点运算)	`[1, 2, 3] .+ [1, 2, 3] == [2, 4, 6] # true` `[1, 2, 3] .* [1, 2, 3] == [1, 4, 9] # true`
检测非数值(NaN)	`isnan(NaN) # true` **而不是** `NaN == NaN # false`
三元运算符	`a == b ? "Equal" : "Not equal"`
短路 AND 和 OR 表达式	`a && b` 和 `a || b`
对象等价	`a === b`

shell/REPL 环境

上一次运算的结果	`ans`
中断命令执行	[Ctrl] + [C]
清屏	[Ctrl] + [L]
运行程序文件	`include("filename.jl")`
查找 `func` 相关的帮助	`?func`
查找 `func` 的所有定义	`apropos("func")`
命令行模式	`;`
包管理模式	]
帮助模式	?
查找特殊符号输入方式	`?☆ # "☆" can be typed by \bigwhitestar<tab>`
退出特殊模式 返回到 REPL	在空行上按 [Backspace]
退出 REPL	`exit()` 或 [Ctrl] + [D]

标准库

为了让 Julia 能加载的更快，许多核心组件都放在与 Julia 捆绑在一起的标准库中。当你想用某一个标准库时，就输入 `using PackageName`。以下是一些标准库及其常用的函数。

`Random`	`rand`, `randn`, `randsubseq`
`Statistics`	`mean`, `std`, `cor`, `median`, `quantile`
`LinearAlgebra`	`I`, `eigvals`, `eigvecs`, `det`, `cholesky`
`SparseArrays`	`sparse`, `SparseVector`, `SparseMatrixCSC`
`Distributed`	`@distributed`, `pmap`, `addprocs`
`Dates`	`DateTime`, `Date`

包管理

一个程序包必须先注册，然后才能在包管理器中看到它。

在 Julia 1.0 中，有两种使用包管理器的方法：

- 一是通过 `using Pkg` 导入 `Pkg` 模块，然后用它的函数管理其他包；
- 或者在 REPL 中输入 `]`，然后按回车。进入特殊的交互式包管理模式。 (要从包管理模式返回 REPL，只需要在空行上按退格键 `BACKSPACE` 就行了)

注意新的工具总是先添加到交互式模式中，然后才会加入 `Pkg` 模块。

在 Julia 会话中使用 Pkg 管理包

列出已安装的包 (人类可读版)	`Pkg.status()`
列出已安装的包 (机器可读版)	`Pkg.installed()`
更新所有包	`Pkg.update()`
安装包	`Pkg.add("PackageName")`
重新构建包	`Pkg.build("PackageName")`
(在安装之后) 使用包	`using PackageName`
删除包	`Pkg.rm("PackageName")`

交互式包管理模式

添加包	`add PackageName`
删除包	`rm PackageName`
更新包	`update PackageName`
使用开发版本	`dev PackageName` 或 `dev GitRepoUrl`
停止使用开发板，返回普通的发行版	`free PackageName`

字符与字符串

字符	`chr = 'C'`
字符串	`str = "A string"`
字符 => 编码	`Int('J') # 74`
编码 => 字符	`Char(74) # 'J'`
任意的 UTF 字符	`chr = '\uXXXX'     # 4 位 HEX` `chr = '\UXXXXXXXX' # 8 位 HEX`

逐字符迭代	`for c in str`
	`  println(c)`
	`end`

字符串拼接	`str = "Learn" * " " * "Julia"`
字符插值	`a = b = 2` `println("a * b = $(a*b)")`
第一个匹配的子串或正则表达式	`findfirst(isequal('i'), "Julia")  # 4`
替换字串或正则表达式	`replace("Julia", "a" => "us")` `# "Julius"`
收集的最后一个索引值	`lastindex("Hello")  # 5`
字符串的长度	`length("Hello")     # 5`
正则表达式	`pattern = r"l[aeiou]"`

字字符串	`str = "+1 234 567 890"`
	`pat = r"\+([0-9]) ([0-9]+)"`
	`m = match(pat, str)`
	`m.captures  # ["1", "234"]`

所有匹配	`[m.match for m = eachmatch(pat, str)]`
所有匹配的迭代器	`eachmatch(pat, str)`

要当心 UTF-8 中的多字节 Unicode 编码：

```
Unicode_string = "Ångström"
lastindex(Unicode_string) # 10
length(Unicode_string)    # 8

Unicode_string[10]  # 'm': ASCII/Unicode U+006d
Unicode_string[9]   # ERROR: StringIndexError("Ångström", 9)
Unicode_string[8]   # 'ö': Unicode U+00f6
```

字符串是不可变的。

数字相关

整数类型	`IntN` 和 `UIntN`, 且 `N ∈ {8, 16, 32, 64, 128}`, `BigInt`
浮点类型	`FloatN` 且 `N ∈ {16, 32, 64}` `BigFloat`
类型的最大和最小值	`typemin(Int8)` `typemax(Int64)`
复数类型	`Complex{T<:Real}`
虚数单位	`im`
机器精度	`eps() # 等价于 eps(Float64)`
圆整	`round()       # 浮点数圆整` `round(Int, x) # 整数圆整`
类型转换	`convert(TypeName, val) # 尝试进行转换/可能会报错` `TypeName(val) # 调用类型构造器转换`

全局常量	`pi # 3.1415...`
	`π  # 3.1415...`
	`im # real(im * im) == -1`

更多常量	`using Base.MathConstants`

Julia 不会自动检测数值溢出。使用 SaferIntegers 包可以得到带溢出检查的整数。

随机数

许多随机数函数都需要 `using Random`。

设置随机数种子	`Random.seed!(seed)`
产生随机数	`rand()   # 均匀分布 [0,1)` `randn()  # 正态分布 (-Inf, Inf)`

产生特定分布的随机数	`using Distributions`
	`my_dist = Bernoulli(0.2) # 举例`
	`rand(my_dist)`

以概率 p 从 A 中进行伯努利抽样	`randsubseq(A, p)`
随机重排 A 中的元素	`shuffle(A)`

数组

声明数组	`arr = Float64[]`
预分配内存	`sizehint!(arr, 10^4)`
访问与赋值	`arr = Any[1,2]` `arr[1] = "Some text"`

数组比较	`a = [1:10;]`
	`b = a  # b 指向 a`
	`a[1] = -99`
	`a == b  # true`

复制元素(而不是地址)/深拷贝	`b = copy(a)` `b = deepcopy(a)`
从 m 到 n 的子数组	`arr[m:n]`
n 个 0.0 填充的数组	`zeros(n)`
n 个 1.0 填充的数组	`ones(n)`
n 个 #undef 填充的数组	`Vector{Type}(undef,n)`
n 个从 start 到 stop 的等间距数	`range(start,stop=stop,length=n)`
n 个随机 Int8 填充的数组	`rand(Int8, n)`
用值 val 填充数组	`fill!(arr, val)`
弹出最后一个元素	`pop!(arr)`
弹出第一个元素	`popfirst!(a)`
将值 val 作为最后一个元素压入数组	`push!(arr, val)`
将值 val 作为第一个元素压入数组	`pushfirst!(arr, val)`
删除指定索引值的元素	`deleteat!(arr, idx)`
数组排序	`sort!(arr)`
将 b 连接到 a 后	`append!(a,b)`
检查值 val 是否在数组 arr 中	`in(val, arr)` 或 `val in arr`
改变维数	`reshape(1:6, 3, 2)' == [1 2 3; 4 5 6]`
转化为字符串，并以 delim 分隔	`join(arr, delim)`

线性代数

想要使用线性代数相关的工具，请用：`using LinearAlgebra`。

单位矩阵	`I # 直接用 I 就好。会自动转换到所需的维数。`
定义矩阵	`M = [1 0; 0 1]`
矩阵维数	`size(M)`
选出第 `i` 行	`M[i, :]`
选出第 `j` 列	`M[:, j]`
水平拼接	`M = [a b]` 或 `M = hcat(a, b)`
竖直拼接	`M = [a ; b]` 或 `M = vcat(a, b)`
矩阵转置	`transpose(M)`
共轭转置	`M'` 或 `adjoint(M)`
迹(trace)	`tr(M)`
行列式	`det(M)`
秩(rank)	`rank(M)`
特征值	`eigvals(M)`
特征向量	`eigvecs(M)`
矩阵求逆	`inv(M)`
解矩阵方程 `M*x == v`	`M\v` 比 `inv(M)*v` 更好。
求 Moore-Penrose 伪逆	`pinv(M)`

Julia 有内置的矩阵分解函数。

Julia 会试图推断矩阵是否为特殊矩阵(对称矩阵、厄米矩阵等)，但有时会失败。为了帮助 Julia 分派最优的算法，可以声明矩阵具有特殊的结构。如：对称矩阵、厄密矩阵(Hermitian)、上三角矩阵、下三角矩阵、对角矩阵等。

控制流与循环

条件语句	`if-elseif-else-end`

`for` 循环	`for i in 1:10`
	`  println(i)`
	`end`

嵌套循环	`for i in 1:10, j = 1:5`
	`  println(i*j)`
	`end`

枚举	`for (idx, val) in enumerate(arr)`
	`  println("the $idx-th element is $val")`
	`end`

`while` 循环	`while bool_expr`
	`  # 做点啥`
	`end`

退出循环	`break`
退出本次循环	`continue`

函数

函数的所有参数都是传参(passed by reference)。

以 `!` 结尾的函数会改变至少一个参数，一般是改变第一个参数：`sort!(arr)`。

必须的参数以逗号分隔，通过位置传入。

可选的参数均需要一个默认值，用 `=` 定义。

关键字参数使用名称标记，它们放在函数签名中的分号后面：

```
function func(req1, req2; key1=dflt1, key2=dflt2)
    # 做点啥
end
```

在调用函数时，关键字参数前的分号 **不是** 必须的。

`return` 语句是可选的，但我们强烈建议你为每一个函数都加上 `return`。

在单一的 `return` 语句中，可以通过元组返回多种数据结构。

从命令行输入的参数 `julia script.jl arg1 arg2...`，可以通过常量 `ARGS` 访问：

```
for arg in ARGS
    println(arg)
end
```

匿名函数可以用于收集函数(collection functions)或列表推断(list comprehensions)： `x -> x^2`.

函数可以接收可变数量的参数

```
function func(a...)
    println(a)
end

func(1, 2, [3:5]) # tuple: (1, 2, UnitRange{Int64}[3:5])
```

函数可以嵌套

```
function outerfunction()
    # 在外面做点啥
    function innerfunction()
        # 在内面做点啥
        # 可以访问之前在外部定义的东西
    end
    # 在外面继续做点啥
end
```

函数可以显示指定返回类型

```
# 接收 Number 的任何子类型，返回一个字符串
function stringifynumber(num::T)::String where T <: Number
    return "$num"
end
```

函数可以通过点语法 向量化

```
# 这里通过点语法广播了减去均值的操作
julia> using Statistics
julia> A = rand(3, 4);
```

字典

字典	`d = Dict(key1 => val1, key2 => val2, ...)` `d = Dict(:key1 => val1, :key2 => val2, ...)`
所有的键 (迭代器)	`keys(d)`
所有的值 (迭代器)	`values(d)`

按键值对迭代	`for (k,v) in d`
	`  println("key: $k, value: $v")`
	`end`

是否存在键 `:k`	`haskey(d, :k)`
将键/值复制到数组	`arr = collect(keys(d))` `arr = [k for (k,v) in d]`

字典是可变的。当使用符号(:symbol)作为键时，键不可变。

集合

声明集合	`s = Set([1, 2, 3, "Some text"])`
并集 `s1 ∪ s2`	`union(s1, s2)`
交集 `s1 ∩ s2`	`intersect(s1, s2)`
补集 `s1 \ s2`	`setdiff(s1, s2)`
对称差 `s1 △ s2` (symmetric difference)	`symdiff(s1, s2)`
子集? `s1 ⊆ s2`	`issubset(s1, s2)`

检查元素是否在集合(set)内可以在 O(1) 的时间内完成。

收集相关函数

将 f 应用到 coll 中的每一个元素上	`map(f, coll)` 或
	`map(coll) do elem`
	`  # 处理 elem`
	`  # 必须有返回值`
	`end`

滤出 coll 中使 f 为真的每一个元素	`filter(f, coll)`
列表推导	`arr = [f(elem) for elem in coll]`

类型

Julia 没有类，因此也没有类相关的方法。

类型就像是没有方法的类。

抽象类型可以作为子类型，但不能被实例化。

具体的类型不能作为子类型。

`struct` 默认是不可变的。

不可变类型能能改善程序的性能，并且它们是线程安全的，因为它们在跨线程使用时不需要同步。

可能是一组类型之一的对象称为 `Union` (联合)类型。

类型注释	`var::TypeName`

类型声明	`struct Programmer`
	`  name::String`
	`  birth_year::UInt16`
	`  fave_language::AbstractString`
	`end`

可变类型声明	将 `struct` 替换为 `mutable struct`
类型别名	`const Nerd = Programmer`
类型构造器	`methods(TypeName)`
类型实例	`me = Programmer("Ian", 1984, "Julia")` `me = Nerd("Ian", 1984, "Julia")`

子类型声明	`abstract type Bird end`
	`struct Duck <: Bird`
	`  pond::String`
	`end`

参数化类型	`struct Point{T <: Real}`
	`  x::T`
	`  y::T`
	`end`
	
	`p = Point{Float64}(1,2)`

联合类型	`Union{Int, String}`
遍历类型层级	`supertype(TypeName)` 和 `subtypes(TypeName)`
默认的超类型	`Any`
所有字段	`fieldnames(TypeName)`
所有字段类型	`TypeName.types`

当使用 **内部** 构造器定义类型时，默认的 **外部** 构造器就不能用了，如果你还想用它就需要手工定义一下。内部构造器非常适合于检查参数是否符合特定的(不变的)条件。当然，可以通过直接访问并修改这些字段来改变这些不变量，除非类型定义为不可变。关键字 `new` 可以用于创建相同类型的对象。

类型参数是不可变的，这意味着即使 `Float64 <: Real`，`Point{Float64} <: Point{Real}` 依旧为假(对于元组类型(Tuple)正好相反，它

缺失值与空值

空值(Null)	`nothing`
缺失数据	`missing`
浮点数的非数值	`NaN`
滤除缺失值	`collect(skipmissing([1, missing, 2])) == [1,2]`
替换缺失值	`collect((df[:col], 1))`
检查是否有缺失值	`ismissing(x)` **而不是** `x == missing`

异常处理

抛出异常 SomeExcep

```
throw(SomeExcep())
```

再次引发当前的异常

```
rethrow()
```

定义新异常 NewExcep

```
struct NewExcep <: Exception
    v::String
end

Base.showerror(io::IO, e::NewExcep) = print(io, "A problem with $(e.v)!")

throw(NewExcep("x"))
```

抛出带文本的异常

```
error(msg)
```

异常处理流程

```
try
    # 进行一些可能会失败的操作
catch ex
    if isa(ex, SomeExcep)
        # 处理异常 SomeExcep
    elseif isa(ex, AnotherExcep)
        # 处理另一个异常 AnotherExcep
    else
        # 处理其余的异常
    end
finally
    # 永远执行这些语句
end
```

模块

模块是独立的全局变量工作区，它们将类似的功能组合到一起。

定义

```
module PackageName
# 添加模块定义
# 使用 export 让定义对外可见
end
```

包含文件 filename.jl

```
include("filename.jl")
```

加载

```
using ModuleName        # 导出所有名称
using ModuleName: x, y           # 仅导出 x, y
using ModuleName.x, ModuleName.y: # 仅导出 x, y
import ModuleName       # 仅导出 ModuleName
import ModuleName: x, y          # 仅导出 x, y
import ModuleName.x, ModuleName.y # 仅导出 x, y
# 得到模块导出名称的数组
names(ModuleName)
```

导出

```
# 包含未导出的、弃用的
# 和编译器产生的名称
names(ModuleName, all::Bool)

# 也显示从其他模块显式导入的名称
names(ModuleName, all::Bool, imported::Bool)
```

`using` 和 `import` 只有一点区别：

使用 `using` 时，你需要写 `function Foo.bar(..` 来给 `Foo`模块的函数 `bar` 增添一个新方法；而使用 `import Foo.bar` 时，只需写 `function bar(...` 就能达到同样的效果。

表达式

Julia 具有同像性：程序被表示为语言本身的数据结构。实际上 Julia 语言里的任何东西都是一个表达式 `Expr`。

符号(Symbols)是受限的字符串，以冒号 `:` 为前缀。相对于其他类型来说，符号效率更高。它也经常用作标识符、字典的键或者数据表里的列名。符号不能进行拼接。

使用引用 `:( ... )` 或块引用 `quote ... end` 可以创建一个表达式，就像 `parse(str)`，和 `Expr(:call, ...)`。

```
x = 1
line = "1 + $x"          # 一些代码
expr = Meta.parse(line) # 生成一个 Expr 对象
typeof(expr) == Expr    # true
dump(expr)              # 打印生成抽象语法(AST)
eval(expr) == 2         # 对 Expr 对象求值: true
```

宏

宏允许你在程序中自动生成代码（如：表达式）。

定义

```
macro macroname(expr)
    # 做点啥
end
```

使用 `macroname(ex1, ex2, ...)` 或 `@macroname ex1, ex2, ...`

内置的宏

```
@assert          # assert (单元测试)
@which           # 查看对特定参数使用的方法/查找函数所在的模块
@time            # 运行时间与内存分配统计
@elapsed         # 返回执行用时
@allocated       # 查看内存分配
@async           # 异步任务

using Test
@test            # 精确相等
@test x ≈ y    # 近似相等 isapprox(x, y)

using Profile
@profile         # 优化
```

创建 *卫生宏* (hygienic macros)的规则：

- 在宏的内部只通过 `local` 声明本地变量。
- 在宏的内部不使用 `eval`。
- 转义插值表达式以避免宏变大：`$(esc(expr))`

并行计算

并行计算相关的工具可以在标准库 `Distributed` 里找到。

启动带 N 各 worker 的 REPL	`julia -p N`
可用的 worker 数量	`nprocs()`
添加 N 个 worker	`addprocs(N)`

查看所有 worker 的 pid	`for pid in workers()`
	`  println(pid)`
	`end`

获得正在执行的 worker 的 id	`myid()`
移除 worker	`rmprocs(pid)`

在特定 pid 的 worker 上运行 f(args)	`r = remotecall(f, pid, args...)`
	`# 或:`
	`r = @spawnat pid f(args)`
	`...`
	`fetch(r)`

在特定 pid 的 worker 上运行 f(args) (更高效)	`remotecall_fetch(f, pid, args...)`
在任意 worker 上运行 f(args)	`r = @spawn f(args) ... fetch(r)`
在所有 worker 上运行 f(args)	`r = [@spawnat w f(args) for w in workers()] ... fetch(r)`
让表达式 expr 在所有 worker 上执行	`@everywhere expr`

并行化带规约函数 red 的循环	`sum = @distributed (red) for i in 1:10^6`
	`  # 进行并行任务`
	`end`

将 f 用用到集合 coll 中的所有元素上	`pmap(f, coll)`

Worker 就是人们所说的并行/并发的进程。

需要并行化的模块，最好拆分成包含所有功能与变量的函数文件，和一个用于处理数据的驱动文件。很明显驱动文件需要导入函数文件。

一个有实际意义的规约函数的例子：单词计数 by Adam DeConinck.

输入/输出

读取流

```
stream = stdin
for line in eachline(stream)
    # 做点啥
end
```

读取文件

```
open(filename) do file
    for line in eachline(file)
        # 做点啥
    end
end
```

读取 CSV 文件

```
using CSV
data = CSV.File(filename)
```

写入 CSV 文件

```
using CSV
CSV.write(filename, data)
```

保存 Julia 对象

```
using JLD
save(filename, "object_key", object, ...)
```

读取 Julia 对象

```
using JLD
d = load(filename) # 返回对象的字典
```

保存 HDF5

```
using HDF5
h5write(filename, "key", object)
```

读取 HDF5

```
using HDF5
h5read(filename, "key")
```

DataFrames

想要类似 `dplyr` 的工具，请使用 DataFramesMeta.jl.

读取 Stata, SPSS, 等文件

```
using StatFiles
```

描述(describe) data frame

```
describe(df)
```

得到 `col` 列的向量

```
v = df[:col]
```

按 `col` 排序

```
sort!(df, [:col])
```

分类(Categorical) `col`

```
categorical!(df, [:col])
```

列出 `col` 的级别

```
levels(df[:col])
```

所有满足 `col==val` 的结果

```
df[df[:col] .== val, :]
```

从宽格式转换为长格式

```
stack(df, [1:n; ])
stack(df, [:col1, :col2, ...]
melt(df, [:col1, :col2])
```

从长格式转换为宽格式

```
unstack(df, :id, :val)
```

让表格可以有空值 `Nullable`

```
allowmissing!(df) 或
allowmissing!(df, :col)
```

在行上迭代

```
for r in eachrow(df)
    # 干点啥
    # r 是带属性的行名
end
```

在列上迭代

```
for c in eachcol(df)
    # 干点啥
    # c 是列名和列向量的元组
end
```

将函数应用到组

```
by(df, :group_col, func)
```

查询

```
using Query
query = @from r in df begin
    @where r.col1 > 40
    @select {new_name=r.col1, r.col2}
    @collect DataFrame # 默认的迭代器
end
```

自我检查与反射

类型	`typeof(name)`
类型检查	`isa(name, TypeName)`
列出子类型	`subtypes(TypeName)`
列出超类型	`supertype(TypeName)`
函数方法	`methods(func)`
即时编译的字节码	`code_llvm(expr)`
汇编代码	`code_native(expr)`

值得关注的包与项目

许多核心包都是由不同的社区来管理。这些社区以 Julia + [Topic] 的形式命名。

统计	Julia Statistics
自动微分	Julia Diff
数值优化	Julia Opt
绘图	Julia Plots
网络(图)分析	Julia Graphs
Web	Julia Web
地理空间	Julia Geo
机器学习	Julia ML

超级常用的包

DataFrames.jl	线性/逻辑斯蒂(logistic)回归
Distributions.jl	统计分布
Flux.jl	机器学习
Gadfly.jl	类 ggplot2 的画图包
LightGraphs.jl	网络分析
TextAnalysis.jl	自然语言处理(NLP)

命名规范

- Julia 代码风格主要的约定是：尽量避免使用下划线，除非不用就难于理解。
- 变量名小写或使用蛇形命名(snake_case)：`somevariable`。
- 常数全部大写：`SOMECONSTANT`。
- 函数名小写或使用蛇形命名(snake_case)：`somefunction`。
- 宏小写或使用蛇形命名(snake_case)：`@somemacro`。
- 类型名用首字母大写的驼峰命名：`SomeType`。
- Julia 代码文件以 `.jl` 为后缀。

更详细的代码风格规范请参阅手册：代码风格指南

性能改进建议

- 编写 类型稳定 的代码
- 尽可能使用不可变类型
- 大数组用 `sizehint` 预分配内存
- 用 `arr = nothing` 释放大数组的内存
- 使用列访问数组，因为多维数组总是以列优先的顺序储存
- 预分配储存结果用的数据结构
- 在实时应用中使用 `disable_gc()` 关闭垃圾收集器
- 避免使用关键字参数的 splat 操作符(`...`)
- 使用会改变参数的 APIs 以避免复制数据结构。(例如：以 `!` 结尾的函数)
- 使用逐元素的数组操作，而不是列表推断(list comprehensions)
- 避免在计算密集的循环中使用 `try-catch`
- 避免在收集(collections)中出现 `Any`
- 避免在收集(collections)中使用抽象类型
- 避免在 I/O 中使用字符串插值
- 不像 R, MATLAB 或 Python，在 Julia 中向量化 并不会提升运行速度
- 避免在运行时使用 `eval`

IDE、编辑器和插件

在线 IJulia 笔记本

- JuliaBox
- Jupyter

文本编辑器

- Juno
- Emacs \ Julia 模式
- vim \ Julia 模式
- VS Code 插件

学习资源

- Julia 中文文档
- Julia 中文社区

英文资源

- Julia 官方文档
- 学习 Julia 的在线资源
- Julia 之月
- 社区准则
- Julia: 数值计算的新尝试 (pdf)
- Julia: 为科学计算而生的快速动态语言 (pdf)

相关视频

- 科学计算新尝试——Julia语言入门教程 by Roger

YouTube 英文资源

- 第五届 JuliaCon 年会 2018
- 第四届 JuliaCon 年会 2017 (Berkeley)
- 第三届 JuliaCon 年会 2016
- Julia 入门 by Leah Hanson
- Julia 简介 by Huda Nassar
- 给 Python 统计学家的 Julia 简介 by John Pearson