概述

Julia 语言及其生态

陈久宁

2021.10.10

《Julia 语言及其应用》— 编程系列讲座

关于我

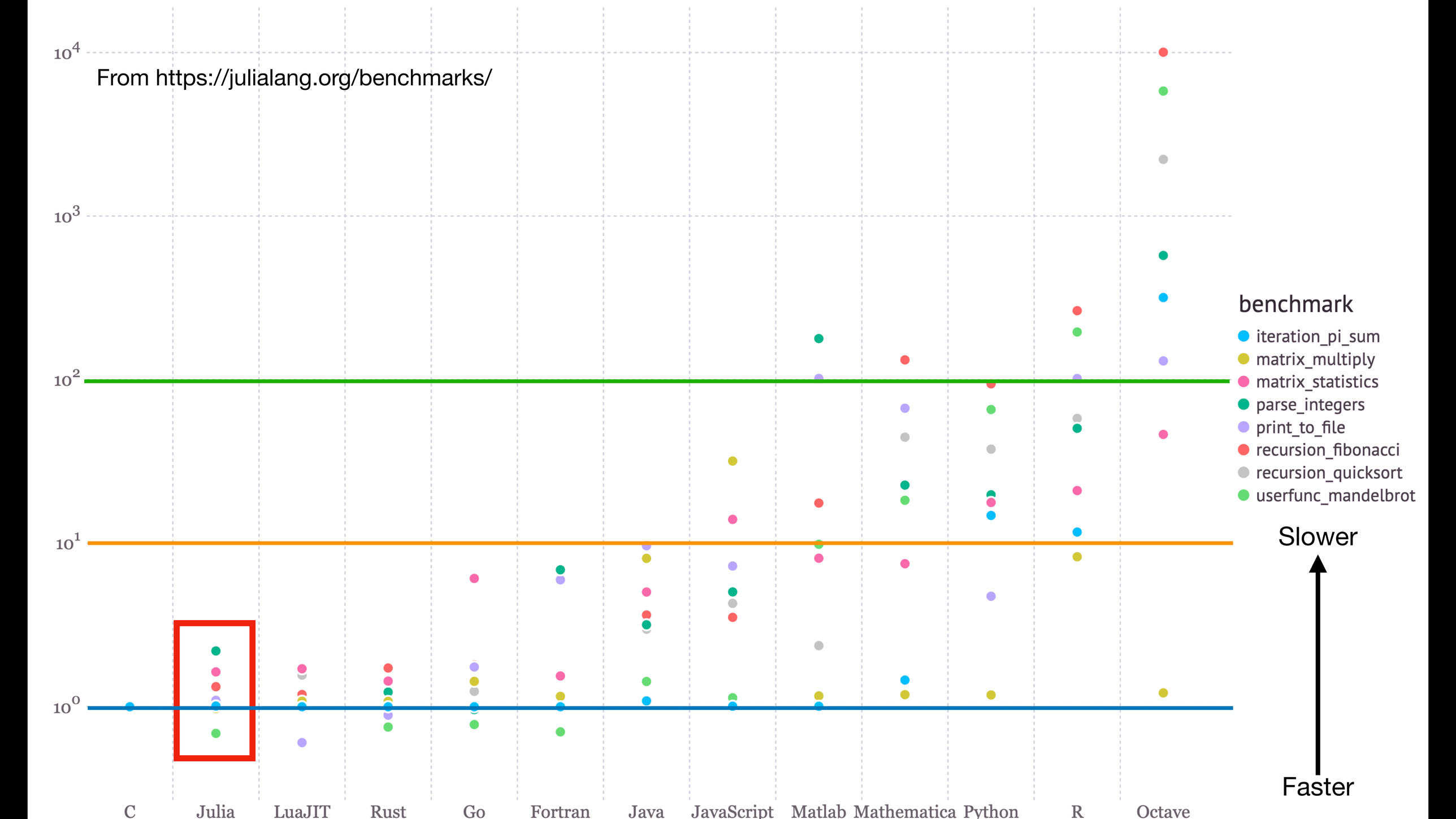
- 本科哲学系，研究生开始在数学系做图像处理，目前博士三年级
- 3年前开始接触 Julia 和开源社区，目前是 JuliaImages 图像处理工具箱的核心开发者，也是 JuliaLang 开发团队的（非核心）成员
- Matlab 图像处理 —> Python 深度学习 —> Julia
- GitHub: johnnychen94

《Julia 语言及其应用》— 编程系列讲座

大纲

- (1) Julia 概述：这是一门什么样的语言，为什么要有这门语言，以及当前的生态
- (2) Julia 入门：Julia 基本语法、数据类型、开发工具以及工作流
- (3-4) Julia 进阶：编程风格、性能优化以及如何写出高质量的 Julia 代码
- (5-6) CPU 并行计算：CPU 硬件模型、SIMD、多线程与异步模型、多进程
- (7-8) GPU 并行计算：GPU 硬件模型及 CUDA
- (9) 自动微分：深度学习的核心组件
- (10 - ?) 待定

交流/通知群


From https://julialang.org/benchmarks/
10^4
10^3
10^2
10^1
10^0
C
Julia
LuaJIT
Rust
Go
Fortran
Java
JavaScript
Matlab
Mathematica
Python
R
Octave
benchmark
iteration_pi_sum
matrix_multiply
matrix_statistics
parse_integers
print_to_file
recursion_fibonacci
recursion_quicksort
userfunc_mandelbrot
Slower
Faster

Julia 语言及其生态

大纲

- Julia 代码风格
- Julia 为什么快
- 生态与社区

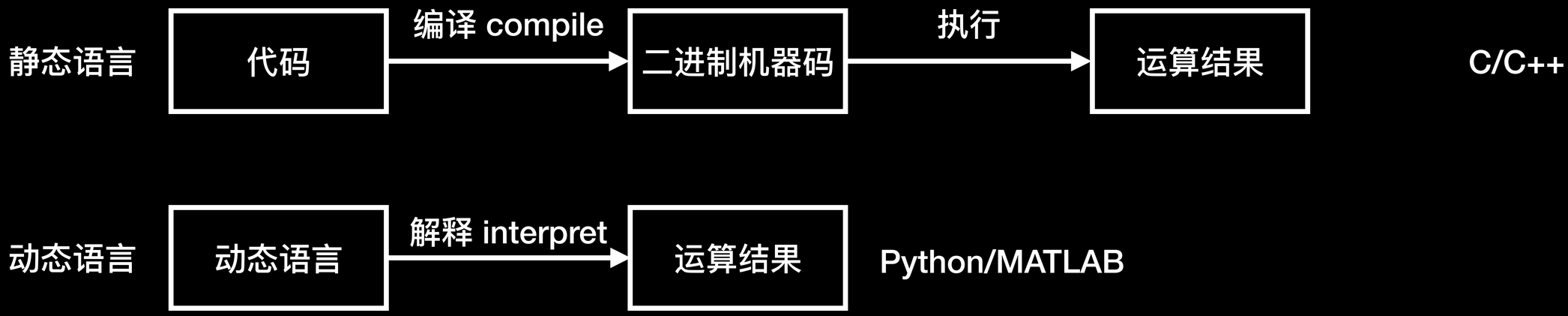
Julia 代码风格

Topics

- 动态编译型语言
- 函数式编程 + 类型 + 多重派发
- 广播 (Broadcasting)

Julia 代码风格 — 动态类型的编译型语言

编译型 vs 解释型



Julia 代码风格 – 动态编译型语言

编译优化

```
function add(x, y)
    z = 0
    z = x + y
    return z
end
```

Dead code elimination

```
julia> @code_llvm add(1, 2)
;  @ REPL[1]:1 within `add'
define i64 @julia_add_658(i64 signext %0, i64 signext %1) {
top:
;  @ REPL[1]:3 within `add'
; ┌ @ int.jl:87 within `+'
   %2 = add i64 %1, %0
; └
;  @ REPL[1]:4 within `add'
  ret i64 %2
}
```

```
function my_sum(A)
    rst = zero(eltype(A))
    @inbounds @simd for x in A
        rst += x
    end
    return rst
end
```

SIMD (CPU parallelization)

Loop unrolling

```
vector.body:                                      ; preds = %vector.body, %vector.ph
; └
; ┌ @ simdloop.jl:78 within `macro expansion'
; │┌ @ int.jl:87 within `+'
    %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body ]
    %vec.phi = phi <4 x double> [ zeroinitializer, %vector.ph ], [ %14, %vector.body ]
    %vec.phi9 = phi <4 x double> [ zeroinitializer, %vector.ph ], [ %15, %vector.body ]
    %vec.phi10 = phi <4 x double> [ zeroinitializer, %vector.ph ], [ %16, %vector.body
    %vec.phi11 = phi <4 x double> [ zeroinitializer, %vector.ph ], [ %17, %vector.body
; └└
```

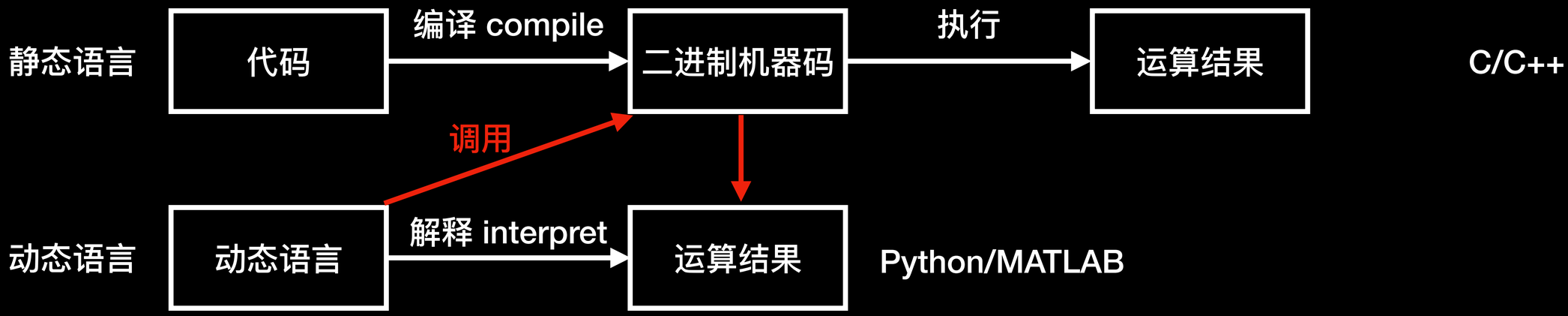
```
function square(x)
    rst = zero(x)
    for i in 1:5
        rst += x*x
    end
    return rst
end
```

Const propagation?

```
; ┌ @ int.jl within `*'
   %1 = mul i64 %0, %0
; └
;  @ REPL[20]:3 within `square'
  %2 = mul i64 %1, 5
;  @ REPL[20]:6 within `square'
  ret i64 %2
```

Julia 代码风格 — 动态编译型语言

Python/MATLAB 中的性能优化



Real world

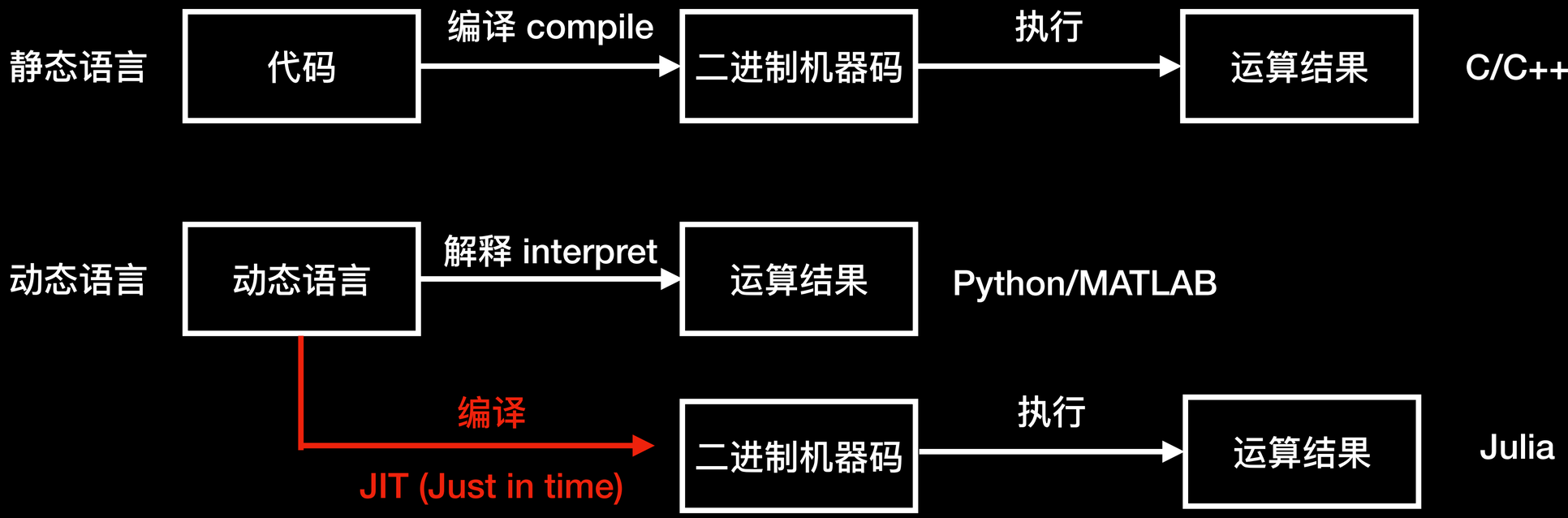
1. 在动态语言中进行快速建模
2. 发现性能太差
3. 切换到静态语言中进行重新实现
4. 接回动态语言

两语言问题

- （对于大部分科研人员来说）在典型的静态语言中 (C/C++) 进行实现太慢且太难
- 直接从动态语言翻译到静态语言并不一定能得到理想的加速
- 跨语言调用本身是一件很麻烦的事情
- 跨语言调用意味着黑盒模型：一些优化策略没办法采用

Julia 代码风格 — 动态编译型语言

以 Python/Matlab 的动态交互方式写代码，并获得 C/C++ 的执行效率



Julia: 能编译的时候编译，不能编译的时候解释

Julia 代码风格 — 动态编译型语言

以 Python/Matlab 的动态交互方式写代码，并获得 C/C++ 的执行效率

```
In [1]: import numpy as np

In [2]: A = np.random.rand(10000, 10000)

In [3]: %timeit np.sum(A)
39 ms ± 281 µs per loop (mean ± std. dev. of 7 runs,

In [4]: B = np.random.rand(1000, 1000)

In [5]: %timeit np.sum(B)
217 µs ± 3.58 µs per loop (mean ± std. dev. of 7 run
```

Python + NumPy C routine

```
In [8]: %timeit my_sum(B)
215 ms ± 1.19 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

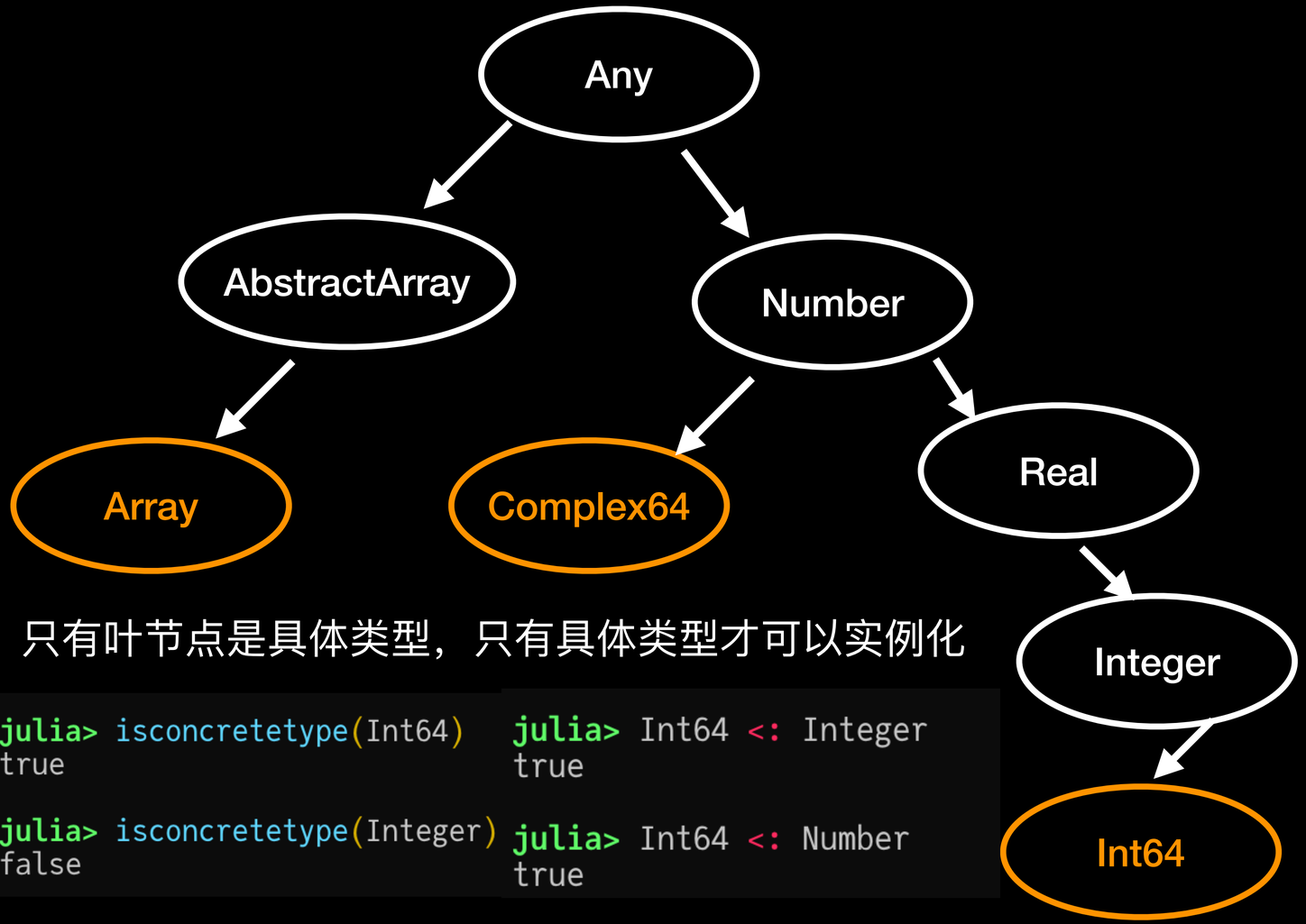
Python manual written

```
julia> A = rand(10000, 10000);

julia> my_sum(A) ≈ sum(A)
true

julia> @btime my_sum($A);
  47.302 ms (0 allocations: 0 bytes)

julia> B = rand(1000, 1000);

julia> @btime my_sum($B);
  117.991 µs (0 allocations: 0 bytes)
```

Julia manual written

Julia 代码风格 — 动态编译型语言

类型树



只有叶节点是具体类型，只有具体类型才可以实例化

```
julia> isconcretetype(Int64)
true

julia> isconcretetype(Integer)
false
```

```
julia> Int64 <: Integer
true

julia> Int64 <: Number
true
```

类型 (Type) 是简单的结构体

```
julia> abstract type AbstractPoint end

julia> struct Point2D <: AbstractPoint
           x::Int
           y::Int
       end

julia> Point2D(1, 2)
Point2D(1, 2)
```

类型没有类方法 (class method)

```
julia> dist_to_origin(p::Point2D) = sqrt(p.x^2 + p.y^2)
dist_to_origin (generic function with 1 method)

julia> dist_to_origin(Point2D(3, 4))
5.0
```

Julia 代码风格 — 函数式编程 + 多重派发

将尽可能多的简短的函数组合一个完整的功能

```
julia> A = rand(1000, 1000);

julia> norm(A) ≈ sqrt(mapreduce(abs2, +, A))
true

julia> norm(A) ≈ sqrt(sum(abs2, A))
true
```

函数式编程：将一个函数作为另一个函数的输入

```
julia> add(x::Number, y::Number) = x + y
add (generic function with 1 method)

julia> add(x::String, y::String) = "$x$y"
add (generic function with 2 methods)

julia> add(1, 2)
3

julia> add("hello", " world")
"hello world"
```

函数式编程 + 多重派发 = 递归?

```
julia> tuple_sum(x::Tuple) = first(x) + tuple_sum(Base.tail(x))
tuple_sum (generic function with 2 methods)

julia> tuple_sum(x::Tuple{}) = 0
tuple_sum (generic function with 2 methods)

julia> tuple_sum((1, 2, 3))
6

julia> function tuple_sum2(x::Tuple)
           isempty(x) && return 0
           return first(x) + tuple_sum2(Base.tail(x))
       end
tuple_sum2 (generic function with 1 method)

julia> tuple_sum2((1, 2, 3))
6
```

多重派发：函数 (Function) 由多个方法 (Method) 共同定义，根据具体的数据类型决定实际调用的方法

```
julia> methods(+)
# 190 methods for generic function "+":
```

Julia 代码风格 — 函数式编程 + 多重派发

单重派发 vs 多重派发

Python 单重派发：根据第一个输入的类型决定具体调用的方法

```python
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def dist(self, p):
        return math.sqrt((self.x - p.x)**2 + (self.y - p.y)**2)
```

Julia 多重派发：根据所有输入的类型决定具体调用的方法

```
julia> dist(p::Point2D, q::Point2D) = sqrt((p.x-q.x)^2 + (p.y-q.y)^2)
dist (generic function with 1 method)

julia> dist(p::Point2D, q::Number) = dist(p, Point2D(q, q))
dist (generic function with 2 methods)

julia> dist(Point2D(3, 4), Point2D(0, 0))
5.0

julia> dist(Point2D(3, 4), 0)
5.0
```

手动派发

```python
def dist(self, p):
    if isinstance(p, Point2D):
        return math.sqrt((self.x - p.x)**2 + (self.y - p.y)**2)
    elif isinstance(p, int):
        return self.dist(Point2D(p, p))
```

单重派发的问题

- 代码过于冗长和复杂
- 扩展问题：C 想基于 B 扩展 A，A 不愿意
- 所有权导致的重复造轮子

Julia 代码风格 — 函数式编程 + 多重派发

多重派发作为核心编程风格

Language	Average # methods (DR)	Choice ratio (CR)	Degree of specialization (DoS)
Cecil[2]	2.33	63.30	1.06
Common Lisp (CMU)[2]	2.03	6.34	1.17
Common Lisp (McCLIM)[2]	2.32	15.43	1.17
Common Lisp (Steel Bank)[2]	2.37	26.57	1.11
Diesel[2]	2.07	31.65	0.71
Dylan (Gwydion)[2]	1.74	18.27	2.14
Dylan (OpenDylan)[2]	2.51	43.84	1.23
Julia[3]	5.86	51.44	1.54
Julia (operators only)[3]	28.13	78.06	2.01
MultiJava[2]	1.50	8.92	1.02
Nice[2]	1.36	3.46	0.33

From https://en.wikipedia.org/wiki/Multiple_dispatch

Julia 代码风格 — 广播

Julia 的广播可以应用到任意函数上

```
julia> A = reshape(collect(1:9), 3, 3)
3×3 Matrix{Int64}:
 1  4  7
 2  5  8
 3  6  9

julia> sum(A)
45

julia> sum.(eachrow(A))
3-element Vector{Int64}:
 12
 15
 18

julia> sum.(eachcol(A))
3-element Vector{Int64}:
  6
 15
 24
```

```
julia> zeros((2, 3))
2×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> zeros.((2, 3))
([0.0, 0.0], [0.0, 0.0, 0.0])
```

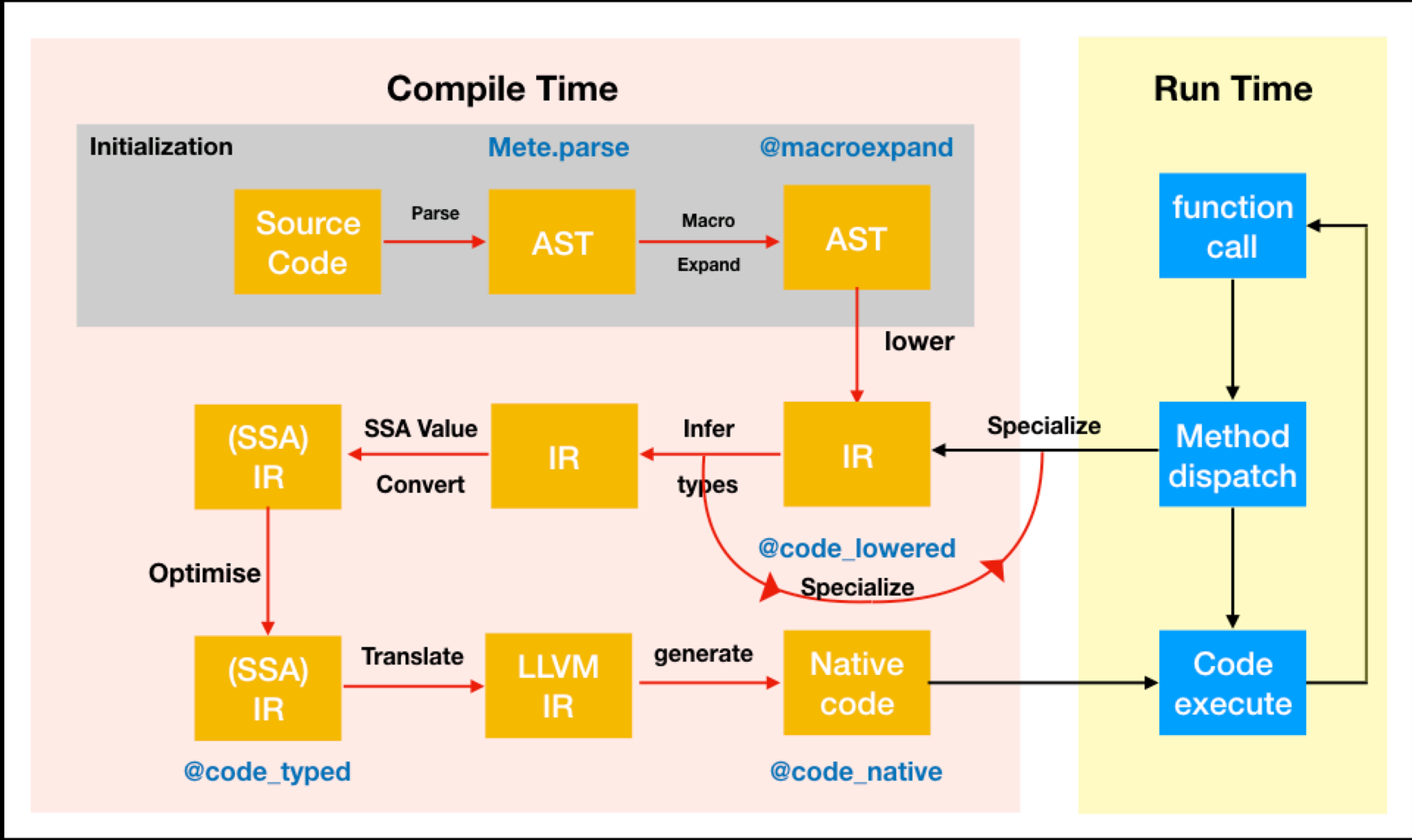
Julia 为什么快

Topics

- 无开销的多重派发
- Abstraction（抽象） + Specialization（特例化）：充分的类型信息
- No first-class citizens （没有一等公民）
- Vectorization/broadcasting （向量化/广播）？
- Julia 一定快吗?

Julia 为什么快

类型稳定的多重派发 == 没有额外开销



JIT (just-in-time)

Julia 为什么快

specialization + abstraction: 充分的类型信息和可维护性

- 定义 Julia 方法的时候可以不给任何类型信息，Julia 会在第一次执行该代码的时候进行一次在线编译(JIT)

```
julia> my_sum(x) = sum(x)
my_sum (generic function with 1 method)
```

可以不写 x 的类型

- 在需要的时候，可以通过多重派发来进行优化

```
julia> my_sum(x::Tuple) = tuple_sum(x)
my_sum (generic function with 2 methods)

julia> x_arr = [1, 2, 3];

julia> @btime my_sum($x_arr);
  3.016 ns (0 allocations: 0 bytes)

julia> x_tuple = (1, 2, 3);

julia> @btime my_sum($x_tuple);
  1.429 ns (0 allocations: 0 bytes)
```

```
julia> @inline tuple_sum(x::NTuple{3, Int}) = x[1] + x[2] + x[3]
tuple_sum (generic function with 3 methods)

julia> @btime my_sum($x_tuple);
  0.047 ns (0 allocations: 0 bytes)
```

Julia 为什么快

No first-class citizens

- 一等公民 (first-class citizens)：为了达到最佳的性能所单独设计的一个**封闭的**数据类型
- 一等公民带来的是不必要的性能开销以及封闭的生态

```
In [11]: x = list(range(10000))

In [12]: %timeit np.sum(x)
483 µs ± 4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [13]: x = np.arange(10000)

In [14]: %timeit np.sum(x)
6.66 µs ± 71.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

np.array 是 Numpy 的一等公民

Julia 为什么快

No first-class citizens: Julia 有几百种矩阵类型

```julia
# 创建一个对角矩阵类型
struct MyDiagonal{T, AT<:AbstractVector} <: AbstractArray{T,2}
    buffer::AT
end
MyDiagonal(A::AbstractVector{T}) where T = MyDiagonal{T, typeof(A)}(A)

@inline Base.axes(A::MyDiagonal) = (axes(A.buffer,1), axes(A.buffer,1))
@inline Base.size(A::MyDiagonal) = (size(A.buffer,1), size(A.buffer,1))

Base.getindex(A::MyDiagonal, i::Int, j::Int) =
    i == j ? A.buffer[i] : zero(eltype(A))
Base.sum(A::MyDiagonal) = sum(A.buffer)

# 简单的测试
A = MyDiagonal(rand(10000)); # size: (10000, 10000)
@btime sum($A); # 806.011 ns (0 allocations: 0 bytes)

B = rand(10000, 10000); # size: (10000, 10000)
@btime sum($B); # 46.781 ms (0 allocations: 0 bytes)
```

```
julia> MyDiagonal([1, 2, 3, 4])
4×4 MyDiagonal{Int64, Vector{Int64}}:
 1  0  0  0
 0  2  0  0
 0  0  3  0
 0  0  0  4
```

Julia 为什么快

向量化 vs for 循环

for 循环版本

```
function normalize_for!(x::AbstractMatrix)
    min_value, max_value = extrema(x)
    @inbounds @simd for i in eachindex(x)
        x[i]= (x[i] - min_value) / (max_value - min_value)
    end
    return x
end
```

向量化版本

```
function normalize_fuse!(x::AbstractMatrix)
    min_value, max_value = extrema(x)
    @. x = (x - min_value) / (max_value - min_value)
end
```

which is fast?

From: https://discourse.juliacn.com/t/topic/5080/9

Julia 为什么快

向量化 vs for

for 循环版本

```julia
function normalize_for!(x::AbstractMatrix)
    min_value, max_value = extrema(x)
    @inbounds @simd for i in eachindex(x)
        x[i]= (x[i] - min_value) / (max_value - min_value)
    end
    return x
end
```

```julia
function normalize_fus
    min_value, max_value = extrema(x)
    @. x = (x - min_value) / (max_value - min_value)
end
```

```julia
img = rand(10_000, 10_000);
@btime normalize_for!($img); # 345.016 ms (0 allocations: 0 bytes)
@btime normalize_fuse!($img); # 404.579 ms (0 allocations: 0 bytes)
```

which is fast?

From: https://discourse.juliacn.com/t/topic/5080/9

Julia 为什么快

“向量化快” 在 Julia 下并不成立

- Python/Matlab 下向量化是一种非常好的性能加速的手段
- 向量化之所以快是因为它以某种手段告诉了计算机充分的类型信息，从而计算机/CPU可以对代码进行特殊的优化手段
- 向量化不是免费的：运算的中间结果是矩阵而不是标量（额外的缓存或内存开销）

Julia 为什么快 — Julia 一定快吗?

在类型不稳定的时候，Julia 的执行效率可以像 Python 一样慢

```
julia> rand_int_or_float() = rand() < 0.5 ? Float64(1) : Int(0)
rand_int_or_float (generic function with 1 method)

julia> rand_float() = rand() < 0.5 ? Float64(1) : Float64(0)
rand_float (generic function with 1 method)

julia> @code_warntype rand_int_or_float()
Variables
  #self#::Core.Const(rand_int_or_float)

Body::Union{Float64, Int64}
1 ─ %1 = Main.rand()::Float64
│   %2 = (%1 < 0.5)::Bool
└──      goto #3 if not %2
2 ─ %4 = Main.Float64(1)::Core.Const(1.0)
└──      return %4
3 ─ %6 = Main.Int(0)::Core.Const(0)
└──      return %6
```

```
julia> A_unstable = map(i->rand_int_or_float(), zeros(1000, 1000));

julia> A_stable = map(i->rand_float(), zeros(1000, 1000));

julia> eltype(A_unstable), eltype(A_stable)
(Real, Float64)

julia> @btime sum($A_unstable);
  23.864 ms (999477 allocations: 15.25 MiB)

julia> @btime sum($A_stable);
  127.006 μs (0 allocations: 0 bytes)
```

Julia 为什么快 — Julia 一定快吗?

在类型不稳定的时候，Julia 的执行效率可以像 Python 一样慢

- 没有办法使用最优的内存结构（类型不稳定时没有办法保证内存的连续性）
- 没有办法绕过 runtime check 的额外性能开销

Julia 生态

简介

- 擅长的领域：与科学计算有关的高性能计算领域
- 不擅长的领域：工程、嵌入式
- 未来的发展方向：不会重复造轮子，但会发明新的轮子

Julia 生态 - 混合生态

Zygote - 源到源的自动微分

```
julia> A = MyDiagonal([1, 2, 3, 4])
4×4 MyDiagonal{Int64, Vector{Int64}}:
 1  0  0  0
 0  2  0  0
 0  0  3  0
 0  0  0  4

julia> loss(A) = sum(abs2, A)
loss (generic function with 1 method)

julia> Zygote.gradient(loss, A)[1]
4×4 Matrix{Int64}:
 2  0  0  0
 0  4  0  0
 0  0  6  0
 0  0  0  8
```

CUDA — GPU计算

```
julia> A_gpu = MyDiagonal(CUDA.CuArray([1, 2, 3, 4]))
4×4 MyDiagonal{Int64, CuArray{Int64, 1}}:
 1  0  0  0
 0  2  0  0
 0  0  3  0
 0  0  0  4

julia> Zygote.gradient(loss, A_gpu)[1]
4×4 Matrix{Int64}:
 2  0  0  0
 0  4  0  0
 0  0  6  0
 0  0  0  8
```

NeuralODE: 深度学习 + 微分方程

自动微分 + 概率编程

...

Julia 生态

- 对于库的开发者友好
- 对于性能优化友好
- 新的语言生态还未发展齐全，对调包侠不太友好

Julia 语言及其生态

Summary

Quotes from Julia authors

- Julia's performance is not magic, its language design is.
- In some ways, Julia is just a really great DSL for LLVM’s JIT

关于 Julia 的定位

- Julia 的性能与 C 保持一致
- Julia 的开发效率与 Python 保持一致
- Julia 的泛型设计与可组合性减少了重复造轮子的需求

更多

关于 Julia 的设计、初衷和介绍：

- Bezanson, Jeff, et al. "Julia: A fresh approach to numerical computing." SIAM review 59.1 (2017): 65-98.
- Christopher Rackauckas, Why Numba and Cython are not substitutes for Julia, 2018.
- Julia motivation: why weren't Numpy, Scipy, Numba, good enough? (discourse), 2017.
- Jeff Bezanson Stefan Karpinski Viral B. Shah Alan Edelman, 为什么我们要创造Julia, 2012.
- Lyndon White, JuliaLang: The Ingredients for a Composable Programming Language, 2020
- 罗秀哲，为什么Python这么慢？为什么需要新的语言解决这个问题？, 2019
- 罗秀哲，Julia 解决了 C++/Python/Matlab 的哪些痛点？, 2018

学习 Julia：

- 官方文档 https://docs.julialang.org/en/v1/
- 英文社区论坛：https://discourse.julialang.org/
- 中文社区主页：https://cn.julialang.org/
- GitHub 上 Julia 包的源代码、issue 及文档

关于 Julia 的一些特性：

- （多维数组索引）Tim Holy, Multidimensional algorithms and iteration, 2016
- （带偏置的数组索引（不从1开始））Tim Holy, Knowing where you are: custom array indices in Julia, 2017
- （广播）Steven G. Johnson, More Dots: Syntactic Loop Fusion in Julia, 2017
- （Trait) Lyndon White, The Emergent Features of JuliaLang: Part I, 2018

系列课程与视频：

- Tim Holy, "Why Julia?" A high level description of the features and benefits of programming in Julia., 2021
- Alan Edelman, Introduction to Computational Thinking (MIT), Spring 2021
- Alan Edelman, How the Julia Language Can Fulfill the Promise of Supercomputing, 2019
- Steven Johnson, High performance in dynamic languages (MIT 6.172), 2018
- Steven Johnson, Adventures in Code Generation (JuliaCon), 2019
- Jeff Bezanson, What's Bad About Julia (JuliaCon), 2019