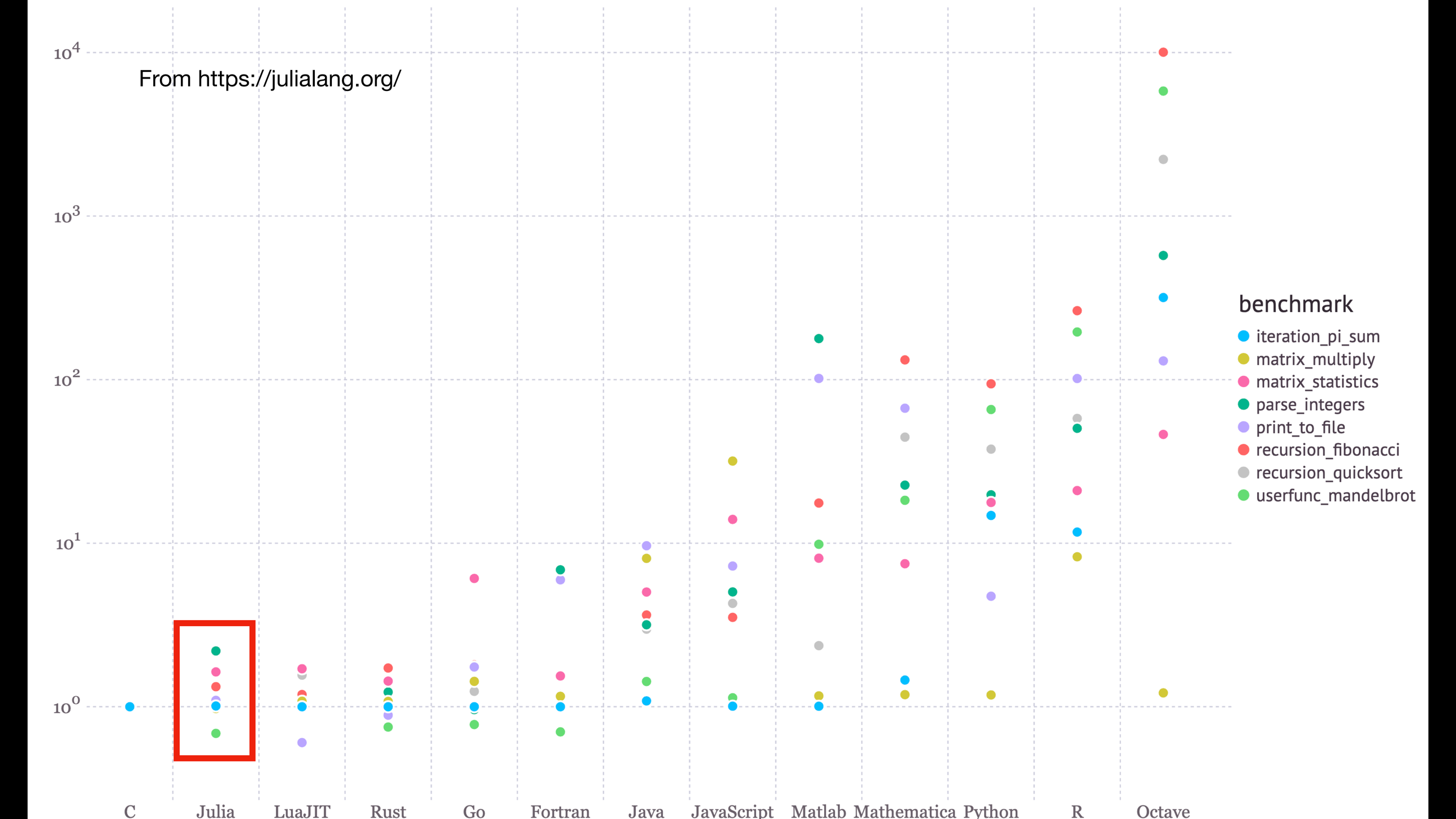
Julia 语言及其生态

陈久宁 @JuliaImages (2021.04.18)

Julia 语言及其生态

Outline

- Julia 的代码风格
- Julia 为什么快
- 生态与社区


From https://julialang.org/
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

Julia 代码风格

Topics

- 动态编译型语言
- Math language!
- 函数式编程 + 多重派发
- Oh my broadcasting!

Julia 代码风格 — 动态编译型语言

以 Python/Matlab 的 REPL(read-execute-print loop) 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 ru
```

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 — A math language!

```
    for n = 1:N
        index = SVector(1,2)
        Λn[1:2,1:2] .=  Λ₁[n][index,index]
        Λn[3:4,3:4] .=  Λ₂[n][index,index]
        m    = hom(ℳ[n])
        m′   = hom(ℳ′[n])
        Un   = (m ⊗ m′)
        ∂xun = [(e₁ ⊗ m′) (e₂ ⊗ m′) (m ⊗ e₁) (m ⊗ e₂)]
        Bn   = ∂xun * Λn * ∂xun'
        Σn   = θ' * Bn * θ
        Σn⁻¹ = inv(Σn)
        T₁   = @SMatrix zeros(Float64,l,l)
        for k = 1:l
            ek = Iₗ[:,k]
            ∂ekΣn = (Iₘ ⊗ ek') * Bn * (Iₘ ⊗ θ) + (Iₘ ⊗ θ') * Bn * (Iₘ ⊗
ek)
            # Accumulating the result in T₁ allocates memory,
            # even though the two terms in the
            # summation are both SArrays.
            T₁ = T₁ + Un * Σn⁻¹ * (∂ekΣn) * Σn⁻¹ * Un' * θ * ek'
        end
        T = T + T₁
    end
```

From https://cormullion.github.io/pages/2020-07-26-JuliaMono/

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"
```

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

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
```

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

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])
```

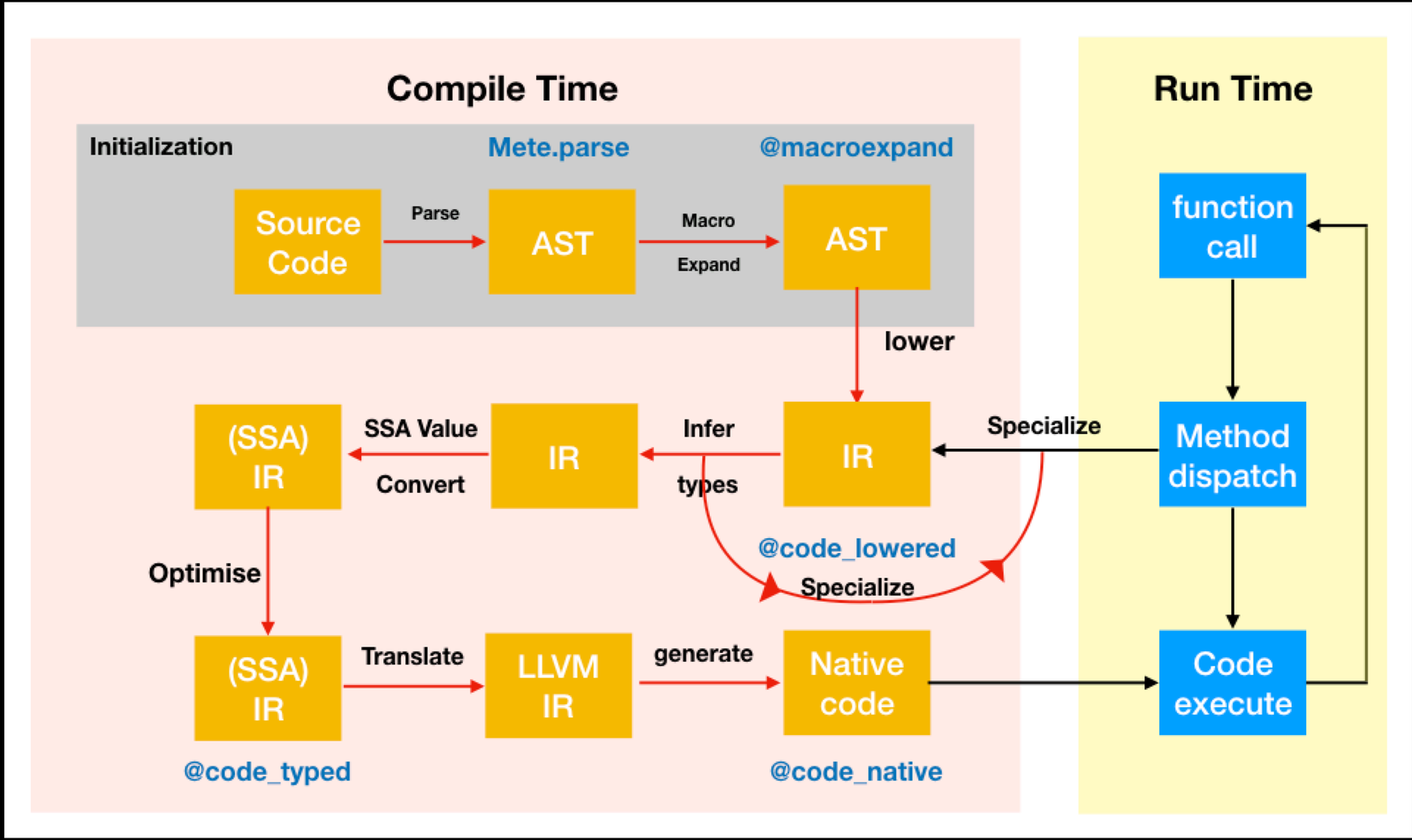
Julia 为什么快

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)
- 在需要的时候，可以通过多重派发来进行优化

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

可以不写 x 的类型

```
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

```
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 has hundreds of array types

```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

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 - 深度学习

- 显卡并行、数据并行、计算并行
- 预处理工具箱
- 预训练的网络 (model zoo)
- 高阶 API

Julia 生态 - 深度学习

这些统统没有

- 显卡并行、数据并行、计算并行
- 预处理工具箱
- 预训练的网络 (model zoo)
- 高阶 API

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

Questions?

Join us

julia 中文