Gabriel Giorisatto De Angelo
Luiz Otávio Gerhardt Fernandes

Histórico

- Desenvolvida em 2012 por Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman
- Gratuita, Open Source e licenciada sob a Licença MIT.
- Versão atual 0.5.0

Comunidade

- Possui várias listas de discussão abertas.
- Comunidade pequena porém ativa.
- YouTube, Twitter, GitHub, StackOverflow e encontros de usuários.
  - São Paulo Julia Meetup e Rio de Janeiro Julia Meetup

JuliaCon

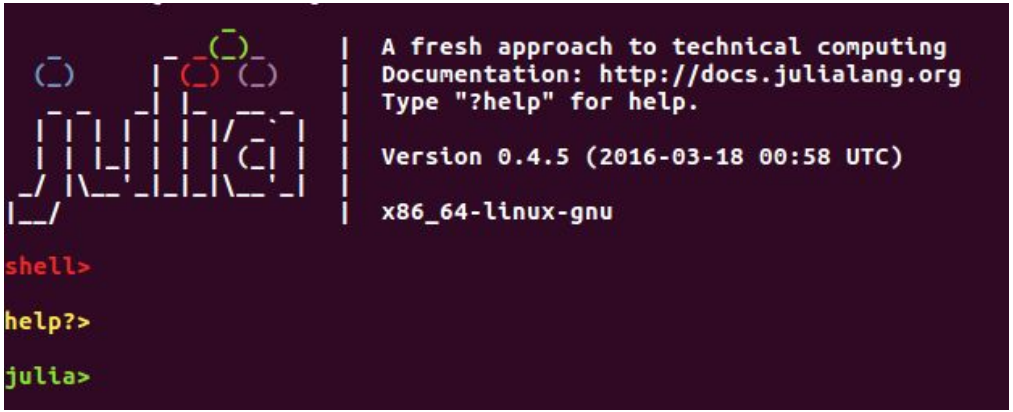
- Conferência anual sobre Julia
- Teve sua terceira edição em Junho de 2016
- Workshops, palestras e hackathons sobre diversos temas de informática, engenharia, matemática, economia entre outros

Introdução

- Julia é uma linguagem de programação dinâmica de alto nível projetada para atender aos requisitos de computação numérica e científica de alto desempenho, além de ser eficaz para programação de uso geral.

Implementação

- Implementação Híbrida
- Compilação JIT (Just in time)
- LLVM (Low Level Virtual Machine)



Paradigmas

- Multi-paradigma, contendo características de
  - Funcional
  - Imperativo
  - Orientação a Objetos

- Não é necessariamente uma linguagem orientada a objetos. Em Julia, todos os valores são objetos, mas as funções não são atreladas aos objetos em que operam.

Palavras Reservadas

- Exemplos: return, break, while, function, else, try, for, true, false

- Nomes permitidos para variáveis:
  - Começar com uma letra, underline, caracteres em UNICODE.
  - Demais letras podem ser números ou até mesmo marcadores de pontuação como !

Cuidado!

```
julia> 2 + 2
4

julia> + = 3
WARNING: imported binding for + overwritten in module Main
3

julia> 2 + 2
ERROR: MethodError: `call` has no method matching call(::Int64, ::Int64, ::Int64)
Closest candidates are:
  BoundsError(::Any...)
  TypeVar(::Any...)
  TypeConstructor(::Any...)
  ...
```

Escopo de Variáveis

Tipo de Escopo	Onde é observado	
Global	Arquivos .jl, módulos ou variáveis do prompt interativo	
Local	Soft	for, while, try-catch
	Hard	functions

Escopo Global

```
julia> module A
       a = 1
       end
A

julia> module B
       b = A.a
       end
ERROR: UndefVarError: A not defined

julia> module B
       import A
       b = A.a
       end
WARNING: replacing module B
B

julia> B.b
1
```

Escopo Global

```
module A
local a = 1
end

A

module B
import A
b = A.a
end

UndefVarError: a not defined
```

Escopo Local

```
julia> for i=1:10
         z = i
         end

julia> z
ERROR: UndefVarError: z not defined
```

Escopo Local

```
> for i=1:10
      global z = i
  end

> z

✓ 10
```

Soft vs Hard

```
x,y = 0,1
(0, 1)
for i = 1:10
    x = i + y + 1
end
x
12
```

Soft
Variáveis são herdadas do escopo pai podendo ser lidas e modificadas.

```
x,y = 0,1
(0, 1)
function foo()
  x = 2
  return x + y
end
foo
foo()
3
x
0
```

Hard
Variáveis são herdadas do escopo pai podendo ser lidas porém não podem ser modificadas

Definições, declarações e tipos

- Atualmente, não é permitida a declaração de tipo em escopo global.
- Fortemente e dinamicamente tipada
- Possui constantes

Constantes

```
> const bar = 1;
> bar
✓ 1
> bar = 2;
⚠ WARNING: redefining constant bar
> bar
✓ 2
```

Tipos inteiros

Type	Signed?	Number of bits	Smallest value	Largest value
`Int8`	✓	8	-2^7	2^7 - 1
`UInt8`		8	0	2^8 - 1
`Int16`	✓	16	-2^15	2^15 - 1
`UInt16`		16	0	2^16 - 1
`Int32`	✓	32	-2^31	2^31 - 1
`UInt32`		32	0	2^32 - 1
`Int64`	✓	64	-2^63	2^63 - 1
`UInt64`		64	0	2^64 - 1
`Int128`	✓	128	-2^127	2^127 - 1
`UInt128`		128	0	2^128 - 1
`Bool`	N/A	8	`false` (0)	`true` (1)

Tipos float

Type	Precision	Number of bits
`Float16`	half	16
`Float32`	single	32
`Float64`	double	64

Tipos padrão

```
# 32-bit system:
julia> typeof(1)
Int32

# 64-bit system:
julia> typeof(1)
Int64
```

Números Complexos e Racionais

- Os tipos primitivos int e float possuem suporte para números complexos e racionais.

Números Complexos e Racionais

```
> typeof(3/2)
✓ Float64
> typeof(3//2)
✓ Rational{Int64}
> typeof(3 + 2im)
✓ Complex{Int64}
```

Declaração de Tipos

- Pode ser inferido pelo compilador ou atribuído pelo programador a fim de melhorar eficiência e aumentar a confiabilidade.

- A declaração de tipo é feita utilizando o operador : :

```
julia> (1+2)::Int
3
```

Conversão de Tipos

```
> x = 12;
> typeof(x)
✓ Int64
> convert(Float64,x);
> typeof(ans)
✓ Float64
> typeof(x)
✓ Int64
```

Caracteres

```
julia> 'x'
'x'

julia> typeof(ans)
Char
```

Strings

```
julia> str = "Hello, world.\n"
"Hello, world.\n"

julia> """Contains "quote" characters"""
"Contains \"quote\" characters"
```

Strings

```
julia> str[1]
'H'

julia> str[6]
','

julia> str[end]
'\n'

julia> str[4:9]
"lo, wo"
```

```
julia> greet = "Hello"
"Hello"

julia> whom = "world"
"world"

julia> string(greet, ", ", whom, ".\n")
"Hello, world.\n"

julia> "$greet, $whom.\n"
"Hello, world.\n"
```

```
> "Hello " * "world!"
✓ "Hello world!"
```

Vetores e Matrizes

```
vet = [1,2,3]
```

```
Int64[3]
  1
  2
  3
```

```
vet[1]
```

```
1
```

```
vet[2:3]
```

```
Int64[2]
  2
  3
```

```
m = [1 2 3;4 5 6;7 8 9]
```

```
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9
```

Vetores e Matrizes

```
julia> vet = []
0-element Array{Any,1}

julia> push!(vet,3)
1-element Array{Any,1}:
 3

julia> push!(vet,"a")
2-element Array{Any,1}:
 3
  "a"

julia> push!(vet,2.2)
3-element Array{Any,1}:
 3
  "a"
 2.2

julia> push!(vet,"hello")
4-element Array{Any,1}:
 3
  "a"
 2.2
  "hello"
```

```
julia> typeof(vet)
Array{Any,1}
```

Vetores e Matrizes

- Existem muitas funções úteis para trabalhar com vetores e matrizes em Julia

size length zeros ones linspace

e muitas outras

Tipo composto

```
type Pessoa
        nome::String
        nascimento::Date
    end
```

```
type Pessoa
  nome::String
  nascimento::Date
end
```

```
fulano = Pessoa("Luiz",Date(1993,09,01))
```

```
Pessoa
    nome → "Luiz"
    nascimento → 1993-09-01
```

Tipo composto

```
type OrderedPair
  x::Real
  y::Real

  OrderedPair(x,y) = x > y ? error("out of order") : new(x,y)
end
```

Tuplas

```
function retornaTupla(a,b)
  return a,b
end
```

```
retornaTupla
retornaTupla(a, b) at base\console:2
```

```
retornaTupla(2,2.5)
```

```
(2, 2.50)
```

```
typeof(ans)
```

```
Tuple{Int64,Float64}
```

Tipos Imutáveis

```
> A = (1,2)
✓ (1, 2)
> A[1]
✓ 1
> A[1] = 12
✗ MethodError: no method matching setindex!(::Tuple{Int64,Int64}, ::Int64, ::Int64)
```

```
immutable Tuple2{A,B}
    a::A
    b::B
end
```

Union

```
> IntOuString = Union{Int,String}
✓ Union{Int64,String}
> 1::IntOuString
✓ 1
> "Oi"::IntOuString
✓ "Oi"
> 3.2::IntOuString
✗ TypeError: typeassert: expected Union{Int64,String}, got Float64
```

Tipo Paramétrico

```
type Point{T}
  x::T
  y::T
end
```

```
pontoFloat = Point(1.0,2.0)
```

```
Point{Float64}
    x → 1.00
    y → 2.00
```

Tipo Abstrato

```
abstract Supimpa

type supimpinha <: Supimpa
  x
end

function supimpar(x::Supimpa)
  if typeof(x) == supimpinha
    println(x.x+=1)
  end
end

s = supimpinha(1)

supimpinha
  x → 1

supimpar(s)

2
```

Funções

```
function somar(x,y)
    return x+y
end
```

```
f(x,y) = x+y
```

```
somar(2,3)
5
f(2,3)
5
```

Tipo de parâmetros

```
function somar(x::Int64,y::Int64)
  return x+y
end
```

```
somar
  somar(x::Int64, y::Int64) at base\console:2
```

```
somar(2,3)
```

```
5
```

Passagem de Parâmetros

```
function funcao(;x=0,y=1)
  return x,y
end
```

```
funcao(y=2,x=3)
```

```
(3, 2)
```

```
funcao(y=2)
```

```
(0, 2)
```

Passagem de Parâmetros

- Pass-by-sharing
- Mesmo comportamento de Python, Ruby e Lisp

Passagem de Parâmetros

```
f(x) = x = 2
a = 1;
f(a)
2
a
1
```

Novo objeto **x** tem valor alterado em **f**, porém o objeto **a** não é alterado

```
f(x) = x[1] = 2;
a = [1,2,3];
f(a)
2
a
Int64[3]
  2
  2
  3
```

Novo objeto **x** tem o objeto da posição 1 alterado, que é o mesmo objeto da posição 1 de **a**

Tipo de retorno de uma função

```
function sinc(x)::Float64
    if x == 0
        return 1
    end
    return sin(pi*x)/(pi*x)
end
```

Funções Anônimas

```
julia> x -> x^2 + 2x - 1
(::#1) (generic function with 1 method)

julia> function (x)
           x^2 + 2x - 1
       end
(::#3) (generic function with 1 method)

julia> map(x -> x^2 + 2x - 1, [1,3,-1])
3-element Array{Int64,1}:
  2
 14
 -2
```

Closures

```
julia> function somar()
       i = 0
       return function()
       i+=1
       end
       end
somar (generic function with 1 method)

julia> closure = somar()
(anonymous function)

julia> closure()
1

julia> closure()
2

julia> closure()
3
```

Varags

```
function func(a,b,c,x...)
  return x
end
```

```
> func(1,2,3,4,5,6,7)
✔ (4, 5, 6, 7)
```

Operadores

Expression	Name	Description
`+x`	unary plus	the identity operation
`-x`	unary minus	maps values to their additive inverses
`x + y`	binary plus	performs addition
`x - y`	binary minus	performs subtraction
`x * y`	times	performs multiplication
`x / y`	divide	performs division
`x \ y`	inverse divide	equivalent to `y / x`
`x ^ y`	power	raises `x` to the `y` th power
`x % y`	remainder	equivalent to `rem(x,y)`
`!x`	negation	changes `true` to `false` and vice versa

Operadores

Expression	Name
`~x`	bitwise not
`x & y`	bitwise and
`x | y`	bitwise or
`x $ y`	bitwise xor (exclusive or)
`x >>> y`	logical shift right
`x >> y`	arithmetic shift right
`x << y`	logical/arithmetic shift left

Operadores

Operator	Name
`==`	equality
`!=` `≠`	inequality
`<`	less than
`<=` `≤`	less than or equal to
`>`	greater than
`>=` `≥`	greater than or equal to

Expressões e Comandos

- Compound Expressions: `begin` and `(;)`.
- Conditional Evaluation: `if` - `elseif` - `else` and `?:` (ternary operator).
- Short-Circuit Evaluation: `&&`, `||` and chained comparisons.
- Repeated Evaluation: Loops: `while` and `for`.
- Exception Handling: `try` - `catch`, `error()` and `throw()`.

Expressões e Comandos

```
julia> z = begin
           x = 1
           y = 2
           x + y
       end
3
```

```
julia> z = (x = 1; y = 2; x + y)
3
```

Expressões e Comandos

```
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
```

Expressões e Comandos

```
julia> if 1
           println("true")
       end
ERROR: TypeError: non-boolean (Int64) used in boolean context
 ...
```

Expressões e Comandos

```
> a < b ? println("sim") : println("nao")
"sim"
> a > b ? println("sim") : println("nao")
"nao"
```

Expressões e Comandos

```
> a = 1;
> true || (a+=1);
> a
✓ 1
> true | (a+=1);
> a
✓ 2
```

Expressões e Comandos

```
julia> i = 1;

julia> while i <= 5
           println(i)
           i += 1
       end
1
2
3
4
5
```

```
> for i=1:2:10
    println(i)
  end
```

```
1
3
5
7
9
```

Expressões e Comandos

```
julia> i = 1;

julia> while i <= 5
           println(i)
           i += 1
       end
1
2
3
4
5
```

```
> for i=1:2:10
    println(i)
  end
```

```
1
3
5
7
9
```

Expressões e Comandos

```
julia> for i in [1,4,0]
         println(i)
       end
1
4
0

julia> for s ∈ ["foo","bar","baz"]
         println(s)
       end
foo
bar
baz
```

Desvios Incondicionais

```
julia> for i = 1:1000
         println(i)
         if i >= 5
           break
         end
       end
1
2
3
4
5
```

```
julia> for i = 1:10
         if i % 3 != 0
           continue
         end
         println(i)
       end
3
6
9
```

Exceções

`Exception`	`LoadError`
`ArgumentError`	`OutOfMemoryError`
`BoundsError`	`ReadOnlyMemoryError`
`CompositeException`	`RemoteException`
`DivideError`	`MethodError`
`DomainError`	`OverflowError`
`EOFError`	`ParseError`
`ErrorException`	`SystemError`
`InexactError`	`TypeError`
`InitError`	`UndefRefError`
`InterruptException`	`UndefVarError`
`InvalidStateException`	`UnicodeError`
`KeyError`	

Exceções

```
> type minhaException <: Exception end
> Base.showerror(io::IO,e::minhaException) = print(io,"Erro da minha exceção");
> throw(minhaException())
✖ ERROR: Erro da minha exceção
```

Try-catch-finally

```
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
```

```
f = open("file")
try
    # operate on file f
finally
    close(f)
end
```

Polimorfismo

Inclusão	Não, pois não possui hierarquia de classes.
Paramétrico	Sim, pois possui tipo paramétrico.
Sobrecarga	Sim, pois a linguagem possui despacho múltiplo.
Coerção	Não possui. Só existe conversão explícita de tipos.

Métodos

```
function SomaOuConc(a::Number,b::Number)
  return a+b
end

function SomaOuConc(a::String,b::String)
  return a*b
end
```

```
SomaOuConc(1,2)
```

```
3
```

```
SomaOuConc(1.5,3.4)
```

```
4.90
```

```
SomaOuConc("Hello ","World!")
```

```
"Hello World!"
```

Métodos Ambíguos

```
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)
ERROR: MethodError: g(::Float64, ::Float64) is ambiguous. Candidates:
  g(x, y::Float64) at none:1
  g(x::Float64, y) at none:1
 ...
```

Paralelismo

```
$ julia -p 4
```

```
julia> nprocs()
5
julia> workers()
4-element Array{Int64,1}:
 2
 3
 4
 5

julia> W1 = workers()[1];
julia> P1 = remotecall(W1, x -> factorial(x), 20)
RemoteRef(2,1,6)
julia> fetch(P1)
2432902008176640000
```

```
julia> @everywhere p = 5
julia> @everywhere println(@sprintf("ID %d: %f %d", myid(), rand(), p))
ID 1: 0.686332 5
        From worker 4:  ID 4: 0.107924 5
        From worker 5:  ID 5: 0.136019 5
        From worker 2:  ID 2: 0.145561 5
        From worker 3:  ID 3: 0.670885 5
```

Paralelismo

```
julia> function findpi(n)
        inside = 0
        for i = 1:n
                x, y = rand(2)
                if (x^2 + y^2 <= 1)
                        inside +=1
                end
        end
        4 * inside / n
       end
```

```
julia> @time findpi(10000)
elapsed time: 0.051982841 seconds (1690648 bytes allocated, 81.54% gc time)
3.14
julia> @time findpi(100000000)
elapsed time: 9.533291187 seconds (8800000096 bytes allocated, 42.97% gc time)
3.1416662
julia> @time findpi(1000000000)
elapsed time: 95.436185105 seconds (88000002112 bytes allocated, 43.14% gc time)
3.141605352
```

```
julia> function parallel_findpi(n)
        inside =  @parallel (+) for i = 1:n
                x, y = rand(2)
                x^2 + y^2 <= 1 ? 1 : 0
        end
        4 * inside / n
       end
parallel_findpi (generic function with 1 method)
```

```
julia> @time parallel_findpi(10000)
elapsed time: 0.45212316 seconds (9731736 bytes allocated)
3.1724
julia> @time parallel_findpi(100000000)
elapsed time: 3.870065625 seconds (154696 bytes allocated)
3.14154744
julia> @time parallel_findpi(1000000000)
elapsed time: 39.029650365 seconds (151080 bytes allocated)
3.141653704
```

Shared Arrays

```
julia> a = SharedArray(Float64,10);

julia> @parallel for i=1:10
         a[i] = i
         println(i)
       end

julia>  From worker 3:  1
        From worker 2:  6
        From worker 2:  7
        From worker 2:  8
        From worker 2:  9
        From worker 2:  10
        From worker 3:  2
        From worker 3:  3
        From worker 3:  4
        From worker 3:  5
julia> a
10-element SharedArray{Float64,1}:
  1.0
  2.0
  3.0
  4.0
  5.0
  6.0
  7.0
  8.0
  9.0
 10.0
```

Avaliação da Linguagem

Critérios Gerais	C	Java	Julia
Aplicabilidade	Sim	Parcial	Sim
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	Parcial
Eficiência	Sim	Não	Sim
Portabilidade	Não	Sim	Sim
Método de projeto	Estruturado	OO	“OO” e Funcional
Evolutibilidade	Não	Sim	Sim

Avaliação da Linguagem

Critérios Gerais	C	Java	Julia
Reusabilidade	Parcial	Sim	Sim
Integração	Sim	Parcial	Sim

Avaliação da Linguagem

Critérios Específicos	C	Java	Julia
Escopo	Sim	SIm	Sim
Expressões e Comandos	Sim	Sim	Sim
Tipos primitivos e compostos	Sim	Sim	Sim
Memória	Programador	Sistema	Sistema
Persistência dos Dados	Biblioteca de funções	JDBC, biblioteca de classes, serialização	Biblioteca de funções, Banco de Dados, serialização,

Avaliação da Linguagem

Critérios Específicos	C	Java	Julia
Passagem de parâmetros	Lista variável e por valor	Lista variável, por valor e por cópia de referência	Lista variável,default,palavra chave por valor e por cópia de referência
Encapsulamento e proteção	Parcial	Sim	Sim

Avaliação da Linguagem

Critérios Específicos	C	Java	Julia
Sistema de tipos	Não	Sim	Sim
Verificação de tipos	Estática	Estática / Dinâmica	Dinâmica
Polimorfismo	Coerção e sobrecarga	Todos	Paramétrico e Sobrecarga
Exceções	Não	Sim	Sim
Concorrência	Não	Sim	Sim

Referências

https://en.wikibooks.org/wiki/Introducing_Julia/Working_with_text_files

http://julialang.org/

http://docs.julialang.org/en/release-0.5/

https://thenewphalls.wordpress.com/2014/02/19/understanding-object-oriented-programming-in-julia-part-1/

http://www.juliabloggers.com/monthofjulia-day-12-parallel-processing/