Meu nome é julia

Lucas de Barros Costa, Nicole Rizzi Nunes e Úrsula Ferreira Abreu

Histórico

- O desenvolvimento de Julia começou em 2009 e uma versão de código aberto foi divulgado em fevereiro de 2012.
- Desenvolvida por: Jeff Bezanson, Stefan Karpinski, Viral B. Shah e Alan Edelman
- Versão atual: 0.5.2 (lançada em 10/05/2017)
- Versão em teste: 0.6.0

Introdução

A linguagem de programação Julia é uma linguagem dinâmica, apropriada para computação numérica e científica, com um desempenho comparável a linguagens estáticas tradicionalmente utilizadas.

Julia é construída com heranças das linguagens de programação matemática, mas também herda muito de outras linguagens dinâmicas populares, incluindo Lisp, Perl, Python, Lua, and Ruby.

A sintaxe de Julia é similar a do GNU Octave ou MATLAB(R) e consequentemente os programadores que que já utilizam estas linguagens devem sentir-se imediatamente confortáveis com Julia.

Características

- Livre e open source (Licença MIT)
- Tipos definidos pelo usuário são rápidos e compactos como tipos nativos
- Ausência da necessidade de vetorizar códigos por desempenho; códigos não vetorizados são rápidos
- Projetado para computação paralela e distribuída
- Suporte eficiente para Unicode, incluindo mas não limitado ao UTF-8

Para aprender

- A instalação de Julia é direta, seja com utilizando binário pré-compilados, seja compilando o código-fonte. Baixe e instale Julia seguindo as instruções (em inglês) em http://julialang.org/downloads/.

- A maneira mais fácil de aprender e experimentar com Julia é iniciando sessão interativa.

- Tutoriais: http://forio.com/julia/tutorials-list

Visão Geral da Linguagem

Visão Geral

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

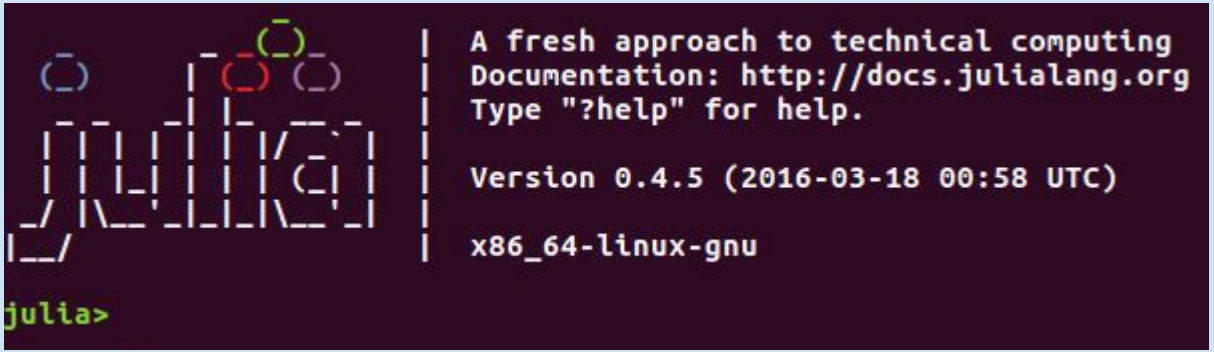
Para baixar e instalar Julia:

$ sudo apt-get install julia

Abrir o terminal julia:

$ julia

Encerrar a sessão: quit()



Paradigmas

- Julia é uma linguagem multiparadigma.
  - Estruturada
  - Orientada a Objetos
  - Funcional

- Julia não permite criação de métodos membros

Palavras Reservadas

- Exemplos de palavras reservadas: for, while, break, if, else, true, false, function, try, type

- Nomes de Variáveis
  - Precisam ser iniciados com uma letra, um underscore ou um conjunto de de caracteres Unicode com código maior que 00A0
  - Podem conter no resto de sua composição números, o caractere ‘!’ e várias outras categorias de caracteres Unicode

Tipos de Dados

- Atualmente, não é permitida a declaração de tipo em escopo global.
- Fortemente e dinamicamente tipada
- Descrevendo Julia na linguagem do sistemas de tipo, é: dinâmico, nominativo e paramétrico.
- Possui constantes

```
julia> const n = 1;

 n = 2; //Erro!
```

Tipos Primitivos

- É definido como um conjunto simples de bits.

- Apesar de conter todos os tipos primitivos usuais, Julia permite que o usuário declare novos tipos primitivos
  - Tamanho definido em bits, porém sempre em múltiplos de 8

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	true

Float

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

Números complexos e racionais

Os tipos primitivos int e float possuem suporte para números complexos e racionais, como podemos ver abaixo:

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

Vetores e Matrizes

- Utiliza-se colchetes para a criação de vetores e/ou matrizes

```
julia> vet = [1 , 2 , 3 , 4]
4-element Array{Int64,1}:
 1
 2
 3
 4
```

Vetores e Matrizes

- Utiliza-se colchetes para a criação de vetores e/ou matrizes

```
julia> mat = [1 2 ; 3 4]
2x2 Array{Int64,2}:
 1  2
 3  4
```

Vetores e Matrizes

- Existem várias funções de manipulação e criação de vetores e matrizes
  - *zeros* e *ones*
  - *length*
  - *fill* e *fill!*
  - várias outras
- Possível utilizar compreensão para criar vetores

```
julia> vet = [x for x=1:3]

3-element Array{Int64,1}:

  1

  2

  3
```

Strings

- Strings literais podem ser escritas com uma ou três aspas duplas
- Suporta Unicode no padrão UTF-8 e permite conversões entre várias codificações UTF
- Concatenação: *string()* ou operador *
- Permite interpolação utilizando o operador $ dentro de uma string.

```
julia> name = “Julia”

“Julia”

julia> "Hello, $name.\n"

"Hello, Julia.\n"
```

Strings

- Strings literais escritas com 3 aspas duplas se comportam de maneira diferente:
  - Remoção de caractere de nova linha (\n) inicial
  - Indentação relativa à linha menos indentada

```
julia> text = """
               Julia

                is a beautiful name.

              """

" Julia\n  is a beautiful name.\n"
```

Declaração de tipos

- Pode ser inferido pelo compilador ou atribuído pelo programador a fim de melhorar eficiência e aumentar a confiabilidade.
- Descrevendo Julia na linguagem do sistemas de tipo, é: dinâmico, nominativo e paramétrico.

```
julia> age::Int8 = 32

32

julia> typeof(age)

Int8
```

Tuplas

- Funcionam como um vetor de itens
- Sempre Imutável
- Geralmente mais útil em quantidades fixas e pequenas

Conversão de dados

- Checagem de tipo
- Função de conversão
- Promoção
  - Definições gerais
  - Regras de Promoção
- OBS: Promoção != Upcasting

Tipos compostos

- Conjunto de informações relacionadas
- Similar a *struct* em C
- *Immutable* vs *Mutable*

```
julia> type Ponto

           x::Float32

           y::Float32

           Ponto(x,y) = new(x,y)

       end
```

Tipos abstratos

- Servem como nós na árvore de hierarquia de tipos
- Definem tipos “mais abrangentes”, sendo papel fundamental no funcionamento do sistema de tipos de Julia

```
julia> abstract Quadrilátero

julia> type Retângulo <: Quadrilátero

            base::Float32

            altura::Float32

            Retângulo (base,altura) = new(base,altura)

        end
```

Tipos Paramétricos

- São tipos que utilizam algum outro tipo que lhe é passado pelo seu usuário.

```
julia> type Ponto{T}
            x::T
            y::T
            Ponto(x,y) = new(x,y)
       end
julia> a = Ponto{T}
```

Curiosidades sobre tipos

- *DataType*
  - Supertipo para todo tipo que semanticamente carrega algum dado
- União de Tipos
  - Capaz de representar mais de um tipo, mesmo que estes não sejam relacionados.
- *Any* e *Bottom*
  - *Any* age como supertipo para todos os tipo, enquanto *Bottom* age como subtipo para todos os tipos
  - *Bottom* é representado por *Union*{}

Curiosidades sobre tipos

- União de Tipos

```
julia> 1::Union{Int64, AbstractString}

1

julia> “Hello”::Union{Int64, AbstractString}

“Hello”

julia> 1.3::Union{Int64,AbstractString}

ERROR: TypeError: typeassert: expected Union{AbstractString,Int64}, got
Float64
```

ARMAZENAMENTO

Escopo das variáveis

Tipo de Escopo	Ocorrências
Escopo Global	`Módulos e Terminal interativo de Julia`
Escopo Local	Soft Local Scope (Escopo Local Leve): `for`, `while`, compreensões, try-catch-finally, `let`
	Hard Local Scope (Escopo Local Forte): funções (em todos os seus tipos de definição) , `structs`, `macros`

Coletor de lixo

- Julia conta com um coletor de lixo, assim como em Java
- Permite forçar a execução chamando a função *gc*()

Serialização

- Possível através da utilização das funções *Base.Serializer.serialize*() e *Base.Serializer.deserialize*()

- Focadas em simplicidade e rapidez, ou seja, não há validação dos dados

Operadores

Operadores Aritméticos

- `+x`
- `-x`
- `x + y`
- `x - y`
- `x / y`
- `x \ y`
- `x ^ y`
- `x % y`

Operadores bit a bit

- `~x`
- `x & y`
- `x | y`
- `x $ y`
- `x >>> y`
- `x >> y`
- `x << y`

Operadores Aritméticos de atualização

- +=
- -=
- *=
- /=
- \=
- ÷=
- %=
- ^=
- &=
- |=
- $=
- >>>=
- >>=
- <<=

Operadores Lógicos

- ==
- !=
- <=
- <
- >=
- >

Curiosidades

```
julia> NaN == NaN
> false
julia> NaN != NaN
> true
julia> NaN < NaN
> false
julia> NaN > NaN
> false
```

- O zero positivo é igual, mas não superior ao zero negativo.
- **Inf** é igual a si mesmo e maior que o resto, exceto **NaN**.
- **-Inf** é igual a si mesmo e menos do que tudo, exceto **NaN**.
- **NaN** não é igual, não inferior a, e não maior do que qualquer coisa, inclusive em si.

CONTROLE DE FLUXO

Controle de fluxo

- `if/else`
- `while`
- `for`
- `break/continue`

if / else

```
julia>  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
```

Operador ternário

```
julia> x = 1 ; y = 2 ;

julia> println ( x < y ? "inferior a" : "não inferior a" )
inferior a

julia> x = 1 ; y = 0 ;

julia> println ( x < y ? "inferior a" : "não inferior a" )
não inferior a
```

While

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

For

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

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

For

```
julia> for i = 1:2, j = 3:4
          println((i, j))
       end
(1,3)
(1,4)
(2,3)
(2,4)
```

Break

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

1
2
3
4
5

FUNÇÕES

Funções

```
f(x::Float64, y::Float64)
    2x + y
end
```

ou

```
f(x::Float64, y::Float64) = 2x + y;
```

Especificando tipo de parâmetro

```
f(x,y)::Float64
    x+y
end
```

Especificando tipo de retorno

Funções

```
julia> f(x::Float64, y::Float64) = 2x + y;

julia> f(2.0, 3.0)

7.0

julia> f(2.0, 3)

ERROR: MethodError: `f` has no method
matching f(::Float64, ::Int64)

Closest candidates are:

  f(::Float64, !Matched::Float64)
```

```
julia> f(x::Number, y::Number) = 2x - y;

julia> f(2.0, 3)

1.0

julia> f

f (generic function with 2 methods)

julia> methods(f)

# 2 methods for generic function "f":

f(x::Float64, y::Float64) at none:1

f(x::Number, y::Number) at none:1
```

Funções

Na ausência de uma declaração de tipo com ::, o tipo de parâmetro de um método é **Any** por padrão, o que significa que não é restritivo, pois todos os valores em Julia são instâncias do tipo abstrato **Any**. Assim, podemos definir um método catch-all para f assim:

```
julia>   function f(x,y)

             println("Whoa there, Nelly.");

         end

julia> f("foo", 1)

Whoa there, Nelly.
```

Funções ambiguas

```
julia> g(x::Float64, y) = 2x + y;

julia> g(x, y::Float64) = x + 2y;
WARNING: New definition
    g(Any, Float64) at none:1
is ambiguous with:
    g(Float64, Any) at none:1.
```

To fix, define

```
    g(Float64, Float64)
```

before the new definition.

```
julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
7.0
```

Funções

```
julia> g(x::Float64, y::Float64) = 2x + 2y;
```

```
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)
10.0
```

Funções

```
julia> same_type{T}(x::T, y::T) = true;

julia> same_type(x,y) = false;
```

```
julia> same_type(1, 2)
true
julia> same_type(1, 2.0)
false
julia> same_type(1.0, 2.0)
true
julia> same_type("foo", 2.0)
false
```

Funções

As funções em Julia são objetos de primeira classe : eles podem ser atribuídos a variáveis, chamados de usar a sintaxe de chamada de função padrão da variável para a qual foram atribuídos. Eles podem ser usados como argumentos, e eles podem ser retornados como valores. Eles também podem ser criados anonimamente, sem ter um nome, usando qualquer uma dessas sintaxes:

```
julia> x -> x^2 + 2x - 1

(anonymous function)

julia> function (x)
           x^2 + 2x - 1
       end

(anonymous function)
```

```
julia> map (x -> x^2 + 2x - 1,[1,2,3])

2

7

14

23
```

Parâmetros Default

```
f(a=1,b=2) = a+2b

é equivalente a:

f(a,b) = a+2b

f(a) = f(a,2)

f() = f(1,2)
```

Retorno

```
julia> function foo(a,b)
         a+b, a*b
       end;
julia> foo(2,3)
(5,6)
```

```
julia> x, y = foo(2,3);

julia> x
5

julia> y
6
```

Varargs

```
bar(a,b,x...) = (a,b,x)

julia> bar(1,2)

(1,2,())

julia> bar(1,2,3)

(1,2,(3,))

julia> bar(1,2,3,4)

(1,2,(3,4))

julia> bar(1,2,3,4,5,6)

(1,2,(3,4,5,6))
```

CLOSURES

Closures

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

```
julia> closure = soma()
(::#9) (generic function with 1 method)

julia> closure()
1
julia> closure()
2
```

Polimorfismo

Polimorfismo

Julia suporta os seguintes tipos de polimorfismo:

- Paramétrico
- Sobrecarga
- Inclusão

O polimorfismo de coerção não ocorre. Se uma função só lida com um determinado tipo a conversão não ocorre.

Polimorfismo paramétrico

```
function same_type{T}(x::T, y::T)
    true
end


function f{T}(x::T)::T
    return x
end
```

Polimorfismo de sobrecarga

```
function g(x::Float64, y::Float64)
    2x + 2y
end


function g(x::Int64, y::Int64)
    2x + y
end
```

Polimorfismo de inclusão

```
abstract type Number end
abstract type Real     <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer  <: Real end
abstract type Signed   <: Integer end
abstract type Unsigned <: Integer end

struct MyCustomException <: Exception end
```

EXCEÇÕES

Exceções

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

Exceções - exemplo

```
julia> sqrt(-1)

ERROR: DomainError:

sqrt will only return a complex result if called with a complex argument. Try
sqrt(complex(x)).

Stacktrace:

 [1] sqrt(::Int64) at ./math.jl:434
```

A função Throw()

```
julia> f(x) = x>=0 ? exp(-x) : throw(DomainError())

f (generic function with 1 method)

julia> f(1)

0.36787944117144233

julia> f(-1)

ERROR: DomainError:

Stacktrace:

 [1] f(::Int64) at ./none:1
```

A função Error()

A função error() é utilizada para produzir um ErrorException que interrompe o fluxo normal de controle.

```
julia> fussy_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not allowed")

fussy_sqrt (generic function with 1 method)

julia> fussy_sqrt(2)

1.4142135623730951

julia> fussy_sqrt(-1)

ERROR: negative x not allowed

Stacktrace:

 [1] fussy_sqrt(::Int64) at ./none:1
```

Avisos e mensagens informativas

```
julia> info("Hi"); 1+1
INFO: Hi
2
julia> warn("Hi"); 1+1
WARNING: Hi
2
julia> error("Hi"); 1+1
ERROR: Hi
Stacktrace:
 [1] error(::String) at ./error.jl:21
```

Try/catch

```
julia> sqrt_second(x) = try
           sqrt(x[2])
       catch y
           if isa(y, DomainError)
               sqrt(complex(x[2], 0))
           elseif isa(y, BoundsError)
               sqrt(x)
           end
       end
```

```
sqrt_second (generic function with 1 method)

julia> sqrt_second([1 4])

2.0

julia> sqrt_second([1 -4])

0.0 + 2.0im

julia> sqrt_second(-9)

ERROR: DomainError:

Stacktrace:

 [1] sqrt_second(::Int64) at ./none:7
```

Finally

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

Tasks

Esse recurso às vezes é chamado por outros nomes, como symmetric coroutines, lightweight threads, cooperative multitasking, ou one-shot continuations.

```
julia> function producer(c::Channel)
           put!(c, "start")
           for n=1:2
               put!(c, 2n)
           end
           put!(c, "stop")
       end;

julia> chnl = Channel(producer);
```

```
julia> take!(chnl)
"start"
julia> take!(chnl)
2
julia> take!(chnl)
4
julia> take!(chnl)
"stop"
```

Tasks

```
julia> for x in Channel(producer)
           println(x)
       end
start
2
4
6
8
stop
```

Tasks

```
julia> a5() = det(rand(1000, 1000));

julia> b = Task(a5);

julia> istaskstarted(b)
false

julia> schedule(b);

julia> yield();

julia> istaskstarted(b)
true

julia> istaskdone(b)
true
```

schedule(b) -> Adiciona uma tarefa na fila Isso faz com que a tarefa seja executada constantemente quando o sistema estiver ocioso, a menos que a tarefa execute uma operação de bloqueio, como wait.

yield() -> Permite que outra tarefa agendada seja executada.

COMPUTAÇÃO PARALELA

Computação paralela

```
$ ./julia -p 2

julia> r = remotecall(rand, 2, 2, 2)


Future(2, 1, 4, Nullable{Any}())
```

```
nheads = @parallel (+) for i =
1:200000000
    Int(rand(Bool))
end
```

```
a = zeros(100000)

@parallel for i = 1:100000

     a[i] = i

end
```

```
a = SharedArray{Float64}(10)

@parallel for i = 1:10

    a[i] = i

end
```

Avaliação da linguagem

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	Estruturado, “OO” e Funcional
Evolutibilidade	Não	Sim	Sim
Reusabilidade	Parcial	Sim	Sim
Integração	Sim	Parcial	Sim

Avaliação da linguagem

Critérios gerais	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, bando de dados, serialização
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

Avaliação da linguagem

Critérios gerais	C	Java	Julia
Encapsulamento e Proteção	Parcial	Sim	Parcial
Sistema de tipos	Não	Sim	Sim
Verificação de tipos	Estática	Estática/Dinâmica	Dinâmica
Polimorfismo	Coerção e sobrecarga	Todos	Paramétrico, sobrecarga e inclusão
Exceções	Não	Sim	Sim
Concorrência	Não	Sim	Sim

Referências

https://pt.wikipedia.org/wiki/Julia_(linguagem_de_programa%C3%A7%C3%A3o)

https://docs.julialang.org/en/stable/

http://www.eripi.com.br/2017/images/anais/minicursos/4.pdf

https://julialang.org/