Linguagem Julia

Jefferson Mika, Lara Brito, Rubens Lavor, Thaiz Izidoro

EMB5631 - Programação III

Introdução

Julia é uma linguagem de programação dinâmica de alto nível projetada para atender aos requisitos de computação numérica e científica de alto desempenho, desenvolvida em 2012.

Implementação e Paradigmas

- Implementação Híbrida
- Compilação JIT (Just in time)
- Multi-paradigma, contendo características Funcional, Imperativo e Orientação a Objetos

Classe, objeto, atributo, método

- Uma **classe** é uma forma de definir um tipo abstrato de dado em uma linguagem orientada a objeto. Sendo formada por dados (**atributos**) e comportamentos (**métodos**).

- **Objeto** refere-se a um molde/cópia da classe, que passa a existir a partir da instanciação da mesma.

- Julia não possui o conceito da entidade de classe. Dessa forma, uma maneira de aplicar o conceito de TAD é criar uma estrutura e criar funções que receba o tipo criado.

main.jl

```julia
struct Exemplo
    atributo1::Int8
    atributo2
end #fim da struct

function funcao_exemplo(e::Exemplo)
  if e.atributo1 == 43
    println(e.atributo2)
  end
end

obj1 = Exemplo(43,"Olá Mundo!")
funcao_exemplo(obj1)
```

Encapsulamento

main.jl

```
1  struct Exemplo
2      atributo1::Int8
3      atributo2
4  end #fim da struct
5
6  obj1 = Exemplo(43,"Olá Mundo!")
7
8  if obj1.atributo1 == 43
9    println(obj1.atributo2)
10 end
11
12 obj1.atributo1=10
13 println(obj1.atributo1)
```

```
Olá Mundo!
error during bootstrap:
LoadError("main.jl", 12, Error
Exception("setfield! immutable
 struct of type Exemplo cannot
 be changed"))
rec_backtrace at /buildworker/
worker/package_linux64/build/s
rc/stackwalk.c:94
record_backtrace at /buildwork
er/worker/package_linux64/buil
d/src/task.c:224 [inlined]
jl_throw at /buildworker/worke
r/package_linux64/build/src/ta
sk.c:461
jl_errorf at /buildworker/work
er/package_linux64/build/src/r
tutils.c:77
jl_f_setfield at /buildworker/
worker/package_linux64/build/s
rc/builtins.c:752
```

Mutable struct

main.jl

```
1  mutable struct Exemplo
2      atributo1::Int8
3      atributo2
4  end #fim da struct
5
6  obj1 = Exemplo(43,"Olá Mundo!")
7
8  if obj1.atributo1 == 43
9    println(obj1.atributo2)
10 end
11
12 obj1.atributo1=10
13 println(obj1.atributo1)
14
```

```
Olá Mundo!
10
>
```

Construtores

- Funções que criam novos objetos de tipos compostos de dados. Um construtor é como qualquer outra função em Julia - seu comportamento geral é definido pela combinação do comportamento de seus métodos.

Construtor default

main.jl

```julia
struct Exemplo1
        atributo1::Int
        atributo2::Int
end

struct Exemplo2
        atributo1::Int
        atributo2::Int
        Exemplo2(atributo1,atributo2) = new(atributo1,atributo2)
end
```

```
julia version 1.3.1
> Exemplo1(1,2)
Exemplo1(1, 2)

> Exemplo2(1,2)
Exemplo2(1, 2)

>
```

- Fornecido quando não há declaração de nenhum construtor interno.

Construtor interno

main.jl

```
1  struct Exemplo
2          x::Real
3          y::Real
4          Exemplo(x,y) = x > y ? error("out of order") : new(x,y)
5      end
6
7
```

```
> Exemplo(1,2)
Exemplo(1, 2)

> Exemplo(2,1)
ERROR: out of order
Stacktrace:
 [1] error(::String) at ./error.jl:33
 [2] Exemplo(::Int64, ::Int64) at ./main.jl:4
 [3] top-level scope at REPL[2]:1

>
```

- É declarado dentro do bloco de declaração do tipo.
- Tem acesso a uma função local chamada de new que cria objetos do tipo desejado.

Construtor externo

main.jl

```
1  struct Exemplo
2          atributo1
3          atributo2
4
5      end
6
7  Exemplo(x) = Exemplo(x,x)
```

```
Exemplo(1)
Exemplo(1, 1)

Exemplo(5.4)
Exemplo(5.4, 5.4)
```

- Métodos adicionais de construtores que são declarados como métodos normais.
- Só podem criar uma nova instância de um objeto chamando um construtor interno.

Destrutores

- Alternativa: Criar função para o comportamento desejado e chamá-la manualmente.

Templates

- Alternativa: Tipos Paramétricos Compostos

main.jl

```julia
struct Exemplo{T}
    atributo1::T
    atributo2::T
end

function template(a::Exemplo, b::Exemplo)
    a.atributo1 + b.atributo1
end

a = Exemplo(5.6, 7.6)
b = Exemplo(4.8, 9.5)
```

```
julia version 1.3.1
‣ Exemplo(1,2)
Exemplo{Int64}(1, 2)

‣ Exemplo(3.5,4.7)
Exemplo{Float64}(3.5, 4.7)

‣ Exemplo("hello","world")
Exemplo{String}("hello", "world")

‣ template(a,b)
10.399999999999999

‣ Exemplo(2.5,8)
ERROR: MethodError: no method matching Exemplo(::Float64, ::Int64)
Closest candidates are:
  Exemplo(::T, ::T) where T at main.jl:2
Stacktrace:
 [1] top-level scope at REPL[7]:1
```

Construtor paramétrico

main.jl

```
1  struct Exemplo{T}
2          atributo1::T
3          atributo2::T
4      end
5
6
7
8
```

```
Exemplo(1,2)
Exemplo{Int64}(1, 2)

Exemplo("hello","world")
Exemplo{String}("hello", "world")

Exemplo(1.54,2.68)
Exemplo{Float64}(1.54, 2.68)

Exemplo{Float64}(1,2)
Exemplo{Float64}(1.0, 2.0)
```

- Para cada tipo de objeto criado, um construtor padrão e distinto para esse tipo que se comporta como um construtor não-paramétrico do tipo em questão também é criado.

Herança simples e múltipla

- Não possui nenhum tipo de herança por design;
- Para reutilizar código é possível se fazer por composição;

Associação: composição e agregação

- Composição de função é quando você combina funções e aplica a composição resultante aos argumentos;
- Operador de composição de função em julia: (∘)
- Para compor as funções é feito (f ∘ g) (args ...) é o mesmo que f (g (args ...));

main.jl

```
1   x = (sqrt ∘ +)(3, 6)
2
```

```
x
3.0
```

Associação: composição e agregação

Outro exemplo usando composição de funções:

main.jl

```
1   exemplo = map(first ∘ reverse ∘
    uppercase, split("you can compose
    functions like this"))
2
```

```
exemplo
6-element Array{Char,1}:
 'U'
 'N'
 'E'
 'S'
 'E'
 'S'
```

Associação: composição e agregação

Outro exemplo usando composição de funções:

main.jl

```
1   exemplo = 1:10 |> sum |> sqrt
2
3   exemplo2 = (sqrt ∘ sum)(1:10)
4
```

```
exemplo
7.416198487095663

exemplo2
7.416198487095663
```

Associação: composição e agregação

- A agregação é usada para atribuir valores para valores compostos
- Podem ser usadas tanto para inicializar quanto para atualizar esses valores
- Para separar as subexpressões é utilizado o caractere “,”
- No caso de inicialização de tipos cartesianos, as subexpressões ficam entre ’()’
- Para vetores, as subexpressões ficam entre os caracteres ’[]’.
- Podem ser tanto estáticas quanto dinâmicas.

Associação: composição e agregação

Exemplo usando agregação em Julia:

main.jl

```
1  struct Data
2      dia::Int
3      mes::Int
4      ano::Int
5  end
6
7  d = Data(14,2,1996)
8
```

```
d
Data(14, 2, 1996)
```

Associação: composição e agregação

Outros exemplos usando agregação em Julia:

main.jl

```
1   vetor = [10 ,20 ,30 ,40 ,50]
2
3   i = 2
4
5   vetor2 = [2* i ,5 ,3* i ]
6
```

```
vetor
5-element Array{Int64,1}:
 10
 20
 30
 40
 50

vetor2
3-element Array{Int64,1}:
 4
 5
 6
```

Polimorfismo

- Polimorfismo é a capacidade de um código operar ou aparentar operar sobre valores de tipos distintos usando a mesma identificação.

- Existem diferentes tipos de polimorfismo, os quais serão detalhados a seguir no contexto da linguagem Julia.

Polimorfismo em Julia

Inclusão	**Não, pois não possui hierarquia de classes.**
Coerção	**Não, pois só existe conversão explícita de tipos.**
Paramétrico	**Sim, pois possui tipo paramétrico**
Sobrecarga	**Sim, pois Julia possui despacho múltiplo**

Polimorfismo Paramétrico

main.jl

```
1   function parametrico( x :: Any )
2     x
3   end
4   struct Point
5     x :: Int
6     y :: Int
7   end
8   p = Point(1,2)
9
10  println(parametrico(p))
11  println(parametrico(21))
12  println(parametrico(13.7))
```

```
Point(1, 2)
21
13.7
```

Polimorfismo por Sobrecarga

main.jl

```
1  function soma( f1 :: Float64 , f2 :: Float64 )
2    f1 + f2
3  end
4  function soma( i1 :: Int64 , i2 :: Int64 )
5    i1 + i2
6  end
7  println(soma(2.1,3.1))
8  println(soma(1,2))
```

```
5.2
3
```

Sobrecarga de operador

- Sim, há sobrecarga de operador em Julia.

- Se você deseja adicionar métodos a uma função ou em um módulo diferente cujo nome contém apenas símbolos, como exemplo um operador, *Base.*+ (operador de soma), para se referir a ele você deve usar "**:**"

  Por exemplo: ***Base.:+***

  E se o operador tem mais de um caractere, você deve colocá-lo entre parênteses:

  Base.:(==)

Sobrecarga de operador

main.jl

```
1  function Base.:+()
2      println("Sobrecarga do Operador +")
3  end
4
5  let x = 1
6      if x == 1
7          +()
8      end
9  end
```

```
Sobrecarga do Operador +
```

Sobrecarga de operador

Em Julia, operadores são funções.

main.jl

```
1   println(1+2+3)
2
3   println(+(1,2,3))
4
5   f = +
6   println(f(1,2,3))
```

```
6
6
6
```

Conclusão

- Apesar de ser uma linguagem “relativamente nova”, a linguagem de programação Julia já nasceu com muito potencial. Possui boa performance que se aproxima de linguagens com tipagem estática, como a linguagem C.

- Diversas são as aplicações de Julia no mercado. Dentre elas podemos citar:
  - Machine Learning
  - Data Science
  - Computação Científica
  - Visualização de dados

Referências

1. https://www.academia.edu/35619638/Dossi%C3%AA_da_Linguagem_Julia
2. http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-20162-seminario-julia.pdf
3. https://docs.julialang.org/en/v1/
4. https://www.treinaweb.com.br/blog/o-que-e-a-linguagem-de-programacao-julia/

Obrigado!

UFSC
Joinville