

TRABALHO

# LINGUAGEM DE PROGRAMAÇÃO JULIA

Roger da Palma , Guilherme Painko  
Scalcon

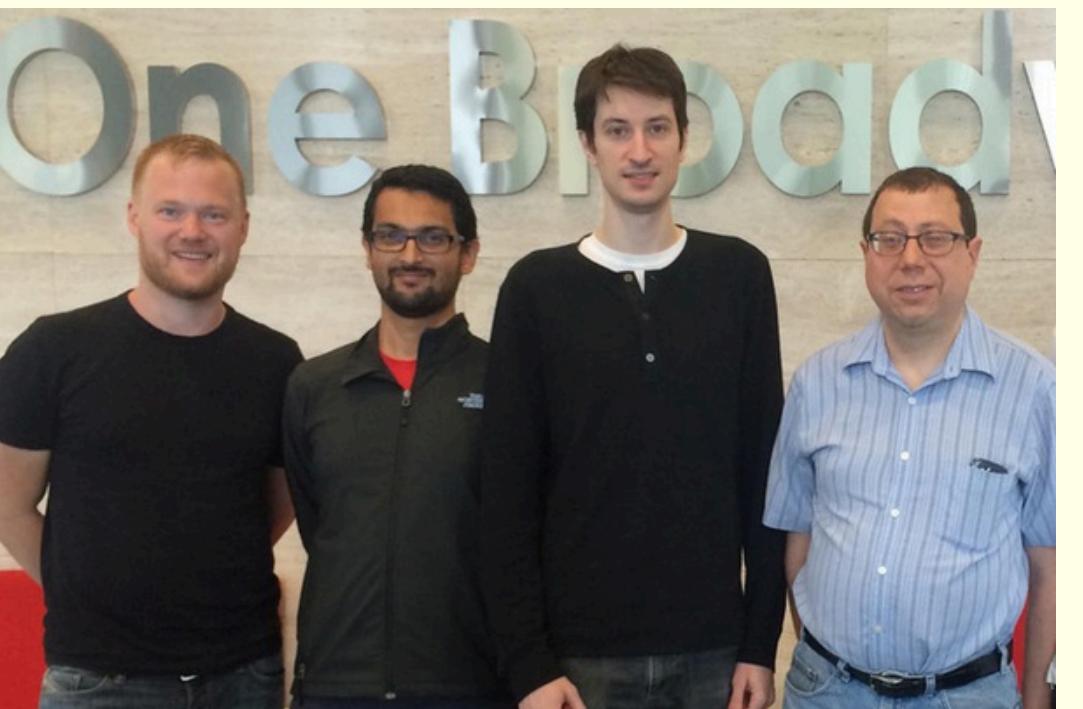
# SUMARIO

## OQUE SERÁ APRESENTADO

1. HISTÓRICO DA LINGUAGEM JULIA
2. PARADIGMA DE PROGRAMAÇÃO
3. PRIMITIVAS PARA COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE PROCESSOS
4. EXEMPLO DE APLICAÇÃO
5. CONCLUSÕES

# HISTÓRICO DA LINGUAGEM JULIA

Julia foi criada com o objetivo de ser uma linguagem **rápida**, de **alto nível** e de **alto desempenho** para computação técnica. Desenvolvida por **Jeff Bezanson, Stefan Karpinski, Viral B. Shah** e **Alan Edelman**, a linguagem foi oficialmente lançada em 2012. O desenvolvimento de Julia começou em 2009, com a intenção de superar os trade-offs entre a facilidade de uso de linguagens como Python e R e a velocidade de linguagens como C e Fortran.



FÁCIL

COMO



julia

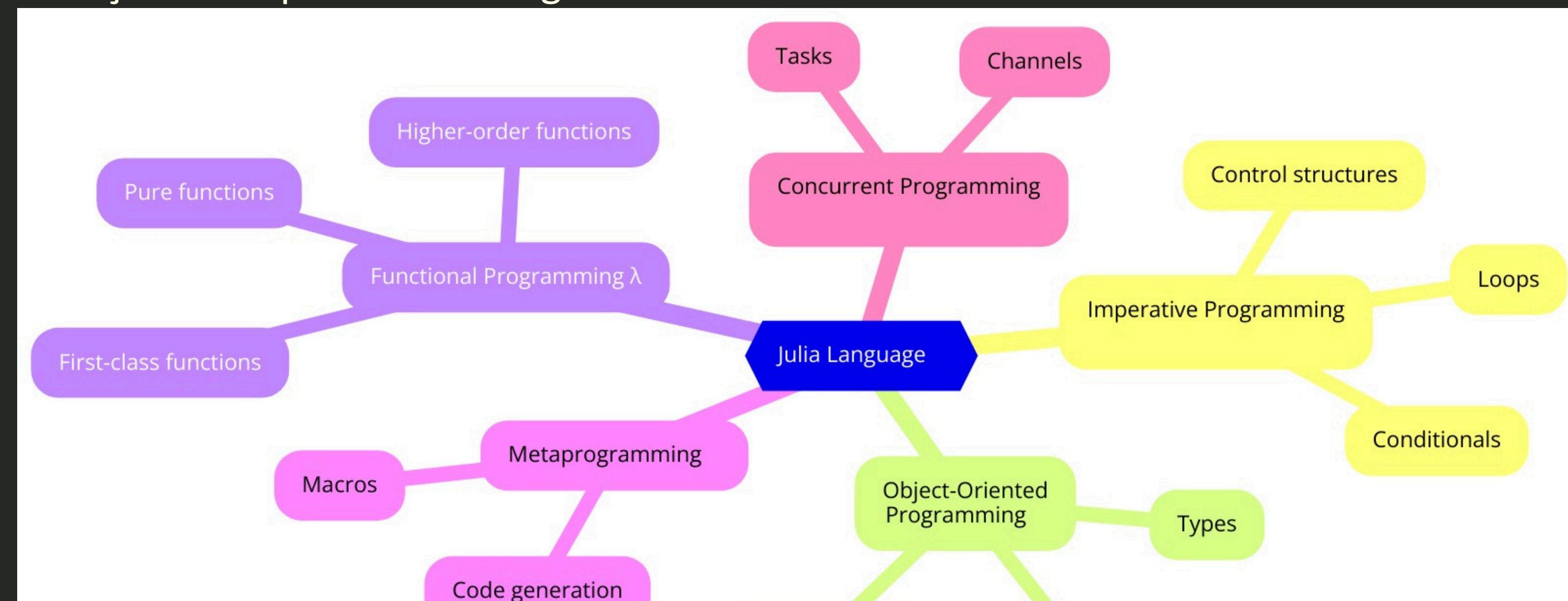
RÁPIDO

COMO



# PARADIGMA DE PROGRAMAÇÃO

Julia é uma linguagem multi-paradigma, suportando programação procedural, funcional e orientada a objetos. É dinâmica, com um sistema de tipos rico e uma forte ênfase em computação numérica e científica. O sistema de despacho múltiplo de Julia, que é central para muitas de suas capacidades, permite que os métodos sejam definidos por combinações de tipos de seus argumentos.



# PRIMITIVAS PARA COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE PROCESSOS

Julia foi projetada para ser boa em computação paralela e distribuída, oferecendo primitivas como:

**@spawn**: Para criar tarefas assíncronas em qualquer processo disponível.

**Canais (Channel)**: Usados para comunicação segura entre tarefas.

**@async** e **@await**: Para programação assíncrona.

**Barreiras de sincronização e locks**: Para sincronização direta entre tarefas.

Além disso, Julia suporta a passagem de mensagens com **RemoteChannel** e **@distributed**

# EXEMPLO DE APLICAÇÃO

```
using Base.Threads

# Define uma função para realizar operações paralelas em matrizes
function parallel_matrix_operations(A::Array{Float64, 2}, B::Array{Float64, 2}, C::Array{Float64, 2})
    # Verifica se as matrizes A, B e C têm o mesmo tamanho e se o número de colunas de B é igual ao número de linhas de C
    @assert size(A) == size(B) == size(C) "As matrizes A, B e C devem ter o mesmo tamanho"
    @assert size(B, 2) == size(C, 1) "O número de colunas de B deve ser igual ao número de linhas de C"

    # Etapa 1: Calcula o produto de matrizes B * C de forma paralela e armazena em D
    D = zeros(Float64, size(B, 1), size(C, 2))
    @threads for i in 1:size(B, 1)
        for j in 1:size(C, 2)
            for k in 1:size(B, 2)
                D[i, j] += B[i, k] * C[k, j]
            end
        end
    end

    # Etapa 2: Calcula a soma de matrizes A + D de forma paralela e armazena em E
    E = zeros(Float64, size(A))
    @threads for i in 1:size(A, 1)
        for j in 1:size(A, 2)
            E[i, j] = A[i, j] + D[i, j]
        end
    end

    # Retorna a matriz resultante E
    return E
end

# Exemplo de uso da função com matrizes A, B e C definidas
A = [1.0 2.0; 3.0 4.0]
B = [5.0 6.0; 7.0 8.0]
C = [1.0 0.0; 0.0 1.0]

# Chama a função e imprime a matriz resultante E
E = parallel_matrix_operations(A, B, C)
println(E)

using CSV # Importa o módulo CSV para ler arquivos CSV
using DataFrames # Importa o módulo DataFrames para manipular dados em formato de tabela
using Plots # Importa o módulo Plots para criar gráficos

# Carregar dados de temperatura de um arquivo CSV chamado 'temperatura_global.csv'
data = CSV.read("temperatura_global.csv", DataFrame)

# Processar e plotar os dados
# Plota os dados de temperatura ao longo dos anos
# 'data[:Ano]' acessa a coluna 'Ano' do DataFrame
# 'data[:Temperatura]' acessa a coluna 'Temperatura' do DataFrame
# 'title' define o título do gráfico
# 'label' define a etiqueta da série de dados
# 'ylabel' define a etiqueta do eixo y
# 'xlabel' define a etiqueta do eixo x
plot(data[:Ano], data[:Temperatura], title="Análise de Temperatura Global", label="Temp", ylabel="Temperatura (°C)", xlabel="Ano")
```

# CONCLUSÕES

Julia combina a facilidade de uso de linguagens de alto nível com o desempenho de linguagens de baixo nível, tornando-a ideal para aplicações que requerem computação intensiva, como modelagem matemática, simulações e aprendizado de máquina. A comunidade Julia está crescendo rapidamente, e sua biblioteca de pacotes continua a expandir, aumentando sua aplicabilidade em diversos campos da ciência e engenharia.



---

# REFERENCIAS

<https://julialang.org/>

[https://run.unl.pt/bitstream/10362/144294/1/Monteiro\\_2022.pdf](https://run.unl.pt/bitstream/10362/144294/1/Monteiro_2022.pdf)

[https://pt.wikipedia.org/wiki/Julia\\_\(linguagem\\_de\\_programa%C3%A7%C3%A3o\)](https://pt.wikipedia.org/wiki/Julia_(linguagem_de_programa%C3%A7%C3%A3o))

[https://github.com/JuliaLangPt/tutorial\\_PT\\_BR](https://github.com/JuliaLangPt/tutorial_PT_BR)

TRABALHO

# LINGUAGEM DE PROGRAMAÇÃO JULIA

Roger da Palma , Guilherme Painko  
Scalcon