



# Linguagens Erlang e Scala

Bernardo Amorim e Fernando Buzato



Linguagem Erlang



# História

Criada na Ericsson em 1988 para ser usada em sistemas de telecomunicação





# Características

- Tipagem: dinâmica, fraca
- Tipos: Inteiros, átomos, floats, referências, binários, pids, portas, funções, tuplas, listas, mapas.
- Strings são implementadas por açúcar sintático.
- Variáveis só podem ter um único valor associado a elas.



# Uma Linguagem Concorrente

Como Sistemas de Telecomunicação são altamente distribuídos, a linguagem que os implementa deve ter alto suporte à concorrência.

# Metas de Projeto

- Processos devem utilizar o mínimo de memória possível
- O sistema deve ser robusto a eventuais erros em seus diferentes agentes.





# Utilização de Memória

- Se quisermos processos leves, podemos confiar no sistema operacional para criá-los e gerenciá-los?



# Utilização de Memória

```
File Edit Options Buffers Tools C Help
[Icons: New, Open, Save, Undo, Cut, Copy, Paste, Find]
#include <stdio.h>
#include <unistd.h>
#define N_FORKS 4

void main(){
    int i;
    for(i=0; i<N_FORKS;i++)
        fork();
    sleep(10);
}

U:--- binge_forking.c All (1,0) (C/l AC Abbrev)
Quit
```





# Utilização de Memória

```
Terminal
bernardo erlang_presentation $ ./binge_forking &
[6] 3173
[5] Done
bernardo erlang_presentation $ ps aux|grep ./binge|grep -v grep
bernardo 3173 0.0 0.0 4228 620 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3174 0.0 0.0 4228 84 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3175 0.0 0.0 4228 84 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3177 0.0 0.0 4228 84 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3178 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3179 0.0 0.0 4228 84 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3180 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3181 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3182 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3183 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3184 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3185 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3186 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3188 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3189 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo 3190 0.0 0.0 4228 88 pts/1 S 21:51 0:00 ./binge_forking
bernardo erlang_presentation $
```



# Utilização de Memória

4.2 MB por processo (Absurdo!)



# Utilização de Memória

A criação e o gerenciamento de processos, portanto, deve ser feita independente do sistema operacional. Mas como?





# BEAM, a VM de Erlang

- Todos os processos são encapsulados pela BEAM, que utiliza apenas um processo do SO.
- O tamanho mínimo de um processo da BEAM é de apenas ~1200B.





# BEAM, a VM de Erlang

A screenshot of an Erlang IDE window. The window has a menu bar with 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'Erlang', and 'Help'. Below the menu bar is a toolbar with icons for file operations (new, open, save, close), editing (undo, redo, cut, copy, paste), and search. The main text area contains the following Erlang code:

```
module(binge_forking).  
-compile(export_all).  
  
binge_fork() ->  
    [spawn(fun() -> timer:sleep(10) end) || _ <- lists:seq(1,16)].
```

The status bar at the bottom of the window displays the text: `-:--- binge_forking.erl All (1,0) (Erlang)` and `(No changes need to be saved)`.



# Tolerância a Erros

Qualquer tipo de sincronia entre os processos (memória compartilhada, mutexes, etc.) faz com que o sistema seja drasticamente mais sensível a eventuais erros em algum de seus agentes.



# Tolerância a Erros

```
File Edit Options Buffers Tools Python Help
[Icons: New, Open, Save, Close, Save As, Undo, Cut, Copy, Paste, Find]
def worker(lock):
    while True:
        try:
            lock.acquire()
            work()
        finally:
            lock.release()
```

U:--- lock\_example.py All (7,26) (Python AC)  
(No changes need to be saved)



# Tolerância a Erros

O try/finally resolve apenas parcialmente o problema.

Classes com **case**:

- E se o SO não fechar o processo gentilmente?
- E se houver falhas no hardware?





# Tolerância a Erros

Os criadores do Erlang solucionaram este problema:

- impossibilitando que processos compartilhem estados
- possibilitando apenas uma forma de interação entre eles:  
a troca de mensagens, com cada processo possuindo sua própria caixa de correio.



# A Interface

- `spawn(module, function, args)`: cria um processo, executando nele a função `module:function(args)`. Retorna o pid do novo processo.
- `Process ! Message`: Envia para o processo de PID `Process` a mensagem `Message`, retornando imediatamente. Não há restrições sobre `Message`, podendo esta ter tipo arbitrário.
- `receive`
  - `pattern1 → action1;`
  - `pattern2 → action2;`
  - `end`: retira uma mensagem da caixa de correio do processo, bloqueando sua execução caso não haja nenhuma. Sendo retirada a mensagem, calcula-se em qual padrão a mensagem se encaixa, executando a ação associada a este padrão.



# A Interface

Como exemplo, implementemos um contador em Erlang.



# A Implementação em Java

File Edit Options Buffers Tools Java Help



```
public class Counter {  
    private int nextVal = 0;  
    public synchronized int getNext() {  
        return nextVal++;  
    }  
    public synchronized void reset() {  
        nextVal = 0;  
    }  
}
```

U:--- counter.java All (10,0) (Java/l AC Abbrev)





# A Implementação em Erlang

File Edit Options Buffers Tools Erlang Help



```
-module(counter1).
-export([make_counter/0, get_next/1, reset/1]).

% Cria um contador, em um novo processo.
make_counter() ->
    spawn(fun() -> counter_loop(0) end).

% Loop principal do contador.
counter_loop(N) ->
    receive
        {From, get_next} ->
            From! {self(), N},
            counter_loop(N+1);
        reset ->
            counter_loop(0);
    end.

% Incrementa o contador e retorna seu valor incrementado.
get_next(Counter) ->
    Sequence ! {self(), get_next},
    receive
        {Sequence, N} -> N.
    end.

% Reseta o contador
reset(Sequence) ->
    Sequence ! reset.
```

U:--- counter1.erl All (28,0) (Erlang)



# A Implementação em Erlang


A implementação ficou menos concisa do que a de Java! Podemos tornar o código mais claro separando a lógica do contador daquela mais genérica.



# A Implementação em Erlang

File Edit Options Buffers Tools Erlang Help

```


-module(server).
-export([start/1, loop/2, call/2, cast/2]).

% Inicia um servidor.
start(Module) ->
    spawn(fun() -> loop(Module, Module:init()) end).

% Loop principal do servidor, com request divididos
% entre tipos 'call' e 'cast'
loop(Module, State) ->
    receive
        {call, {Client, Id}, Params} ->
            {Reply, NewState} = Module:handle_call(Params, State),
            Client ! {Id, Reply},
            loop(Module, NewState);
        {cast, Params} ->
            NewState = Module:handle_cast(Params, State),
            loop(Module, NewState)
    end.

% Manda uma mensagem do tipo 'call' ao servidor
call(Server, Params) ->
    Id = make_ref(),
    Server ! {call, {self(), Id}, Params},
    receive
        {Id, Reply} -> Reply
    end.

% Manda uma mensagem do tipo 'cast' ao servidor
cast(Server, Params) ->
    Server ! {cast, Params}.

```

U:\*\*- server.erl All (32,0) (Erlang)



# A Implementação em Erlang

File Edit Options Buffers Tools Erlang Help



```
-module(counter2).  
-export([make_sequence/0, get_next/1, reset/1]).  
-export([init/0, handle_call/2, handle_cast/2]).  
  
make_sequence() -> server:start(counter2).  
get_next(Counter) -> server:call(Sequence, get_next).  
reset(Sequence) -> server:cast(Sequence, reset).  
  
init() -> 0.  
handle_call(get_next, N) -> {N, N+1}.  
handle_cast(reset, _) -> 0.  
|
```

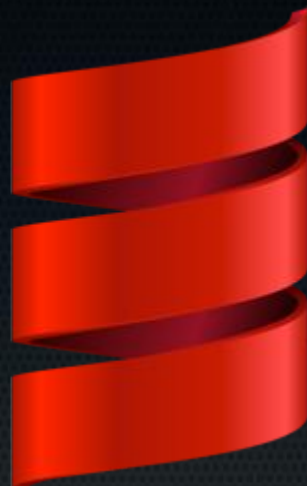
U:--- counter2.erl All (12,0) (Erlang)



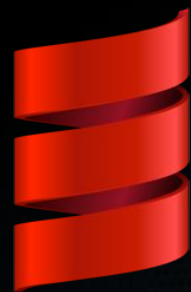


# A Implementação em Erlang

Código concorrente em Erlang não é livre das problemáticas comuns a este paradigma – porém, sendo fácil na linguagem abstrair a lógica de passagem de mensagens das callbacks associadas ao recebimento, eventuais erros são mais explícitos do que em casos análogos para outras linguagens.



# Linguagem Scala



# História

Linguagem Criada por Martin Odersky em 2001

- . Professor da Escola Politécnica Federal de Lausanne (Suiça): onde publicou seus trabalho relacionados a linguagem Scala.
- . Um dos criadores do Generics em Java e um dos colaboradores na criação do compilador Java.
- . Trabalhou com Niklaus Wirth (Criador do Pascal) e com Philip Wadler (Designer do Haskell).

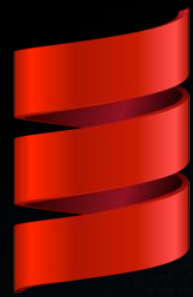




# Motivador

Criar uma linguagem combinando Orientação a Objetos e Programação funcional



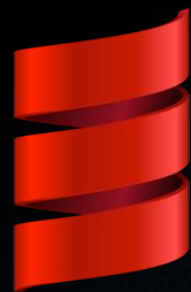


# O que é Scala?

- Linguagem Multiparadigma que incorpora recursos de linguagem Orientada a Objetos e Linguagem Funcional.

*“Object-Oriented Meets Functional*

*Have the best of both worlds. Construct elegant class hierarchies for maximum code reuse and extensibility, implement their behavior using higher-order functions. Or anything in-between”*

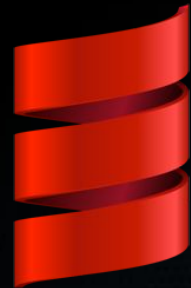


# O que é Scala?

O nome Scala significa "Scalable Language"

- . Projetada para escalar de acordo com a necessidade dos usuários/desenvolvedores.
- . A linguagem pode ser utilizada para escrita de simples scripts até a construção de grandes sistemas complexos.

A linguagem roda sobre a JVM padrão do Java e é totalmente interoperável com a linguagem Java.



# Quem está utilizando?







# Principais Características

## Concisão

```
public class Pessoa {  
  
    private String nome;  
    private String sobrenome;  
    private Integer idade;  
  
    public Pessoa(String nome, String sobrenome, Integer idade) {  
        super();  
        this.nome = nome;  
        this.sobrenome = sobrenome;  
        this.idade = idade;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public String getSobrenome() {  
        return sobrenome;  
    }  
  
    public Integer getIdade() {  
        return idade;  
    }  
  
}
```

Java

```
class Pessoa (val nome: String, val sobrenome: String, val idade: Int)
```

Scala





# Principais Características

## Tipagem Estática:

- Os tipos das variáveis são conhecidos em tempo de compilação assim como nas linguagens tradicionais como Java, C e C ++.

## Inferência de Tipos:

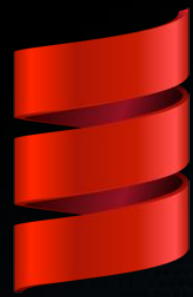
```
val x = 1 + 2 * 3           // o tipo de x é Int
println(x)
val y = x.toString()       // o tipo de y é String
println(y)
def succ(x: Int) = x + 1    // o método succ retorna um valor Int
println (succ(x))
```



# Principais Características

Imutabilidade:

```
val variavelImutavel = "Eu sou Imutável"  
variavelImutavel = "Será que sou?"
```



# Principais Características

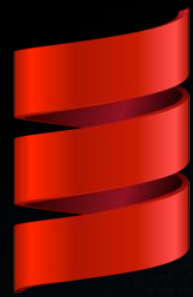
Imutabilidade:

```
val variavelImutavel = "Eu sou Imutável"  
variavelImutavel = "Será que sou?"
```



reassignment to val





# Principais Características

## Imutabilidade:

```
val variavelImutavel = "Eu sou Imutável"  
variavelImutavel = "Será que sou?"
```



reassignment to val

## Mutabilidade:

```
var variavelMutavel = "Eu sou Mutável"  
variavelMutavel = "Será que sou?"  
variavelMutavel = "É claro que sou"  
println (variavelMutavel)
```





# Principais Características

## Imutabilidade:

```
val variavelImutavel = "Eu sou Imutável"  
variavelImutavel = "Será que sou?"
```



reassignment to val

## Mutabilidade:

```
var variavelMutavel = "Eu sou Mutável"  
variavelMutavel = "Será que sou?"  
variavelMutavel = "É claro que sou"  
println (variavelMutavel)
```

```
sbt: [info] Running Main  
É claro que sou
```



# Principais Características

## Imutabilidade:

```
val variavelImutavel = "Eu sou Imutável"  
variavelImutavel = "Será que sou?"
```

✗ reassignment to val

## Mutabilidade:

```
var variavelMutavel = "Eu sou Mutável"  
variavelMutavel = "Será que sou?"  
variavelMutavel = "É claro que sou"  
println (variavelMutavel)
```

```
sbt: [info] Running Main  
É claro que sou
```

val = Utilizado para variáveis imutáveis

var = Utilizado para variáveis mutáveis



# Principais Características

## Orientação a Objetos:

- Tudo em Scala é tratado como um objeto, inclusive valores e funções.
- Não existem tipos primitivos, todos os tipos são objetos.





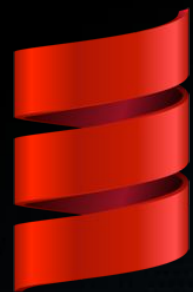
# Principais Características

## Classes:

```
class Pessoa (val nome: String, val sobrenome: String, val idade: Int) {  
    def comoString(): String =  
        s"Pessoa($nome,$sobrenome,$idade)"  
}  
  
val pessoa = new Pessoa("Fernando","Buzato",32)  
println(pessoa.comoString)
```

```
sbt: [info] Running Main  
Pessoa(Fernando,Buzato,32)
```





# Principais Características

Classes com **case**:

```
case class Pessoa (nome: String, sobrenome: String, idade: Int)

val pessoa = Pessoa("Fernando", "Buzato", 32)
println(pessoa)
```

```
sbt: [info] Running Main
Pessoa(Fernando,Buzato,32)
```



# Principais Características

## Classes com **case**:

```
case class Pessoa (nome: String, sobrenome: String, idade: Int)

val pessoa = Pessoa("Fernando", "Buzato", 32)
println(pessoa)
```

```
sbt: [info] Running Main
Pessoa(Fernando,Buzato,32)
```

Com **case**, o compilador já implementa automaticamente:

- toString()
- equals() e hashCode()
- atributos públicos e imutáveis por padrão
- não é necessário o new para instanciar um objeto



# Principais Características

## Programação Funcional:

- Funções de Ordem Superior.
- Funções anônimas.
- Currying





# Principais Características

## Funções de Ordem Superior:

- Funções que podem receber funções como parâmetro ou que podem devolver funções como resultado.

```
def funOrdemSuperior(x: Int, y: Int, f: (Int, Int) => Int): Int = f(x, y)
def soma(a: Int, b: Int) = a + b
val resultado = funOrdemSuperior(2, 3, soma)
println(resultado)
```

```
sbt: [info] Running Main
5
```





# Principais Características

## Funções Anônimas ou Lambdas:

- Funções que não precisam ser nomeadas.

```
def funOrdemSuperior(x: Int, y: Int, f: (Int, Int) => Int): Int = f(x, y)

val resultado = funOrdemSuperior(2, 3, (a: Int, b: Int) => a + b)

println (resultado)
```

```
sbt: [info] Running Main
5
```



# Principais Características

## Currying:

- Técnica para reescrita de funções com múltiplos parâmetros como a composição de funções de um parâmetro

```
//Função sem Currying
def soma(a: Int, b: Int) = a + b

val resultado_soma = soma (2,3)

println ("Resultado Soma sem Currying: " + resultado_soma)

//Função com Currying
def soma(a: Int) = (b: Int) => a + b

val resultado_soma_currying = soma(2)(3)

println("Resultado Soma com Currying: " + resultado_soma_currying)
```

```
sbt: [info] Running Main
Resultado Soma sem Currying: 5
Resultado Soma com Currying: 5
```

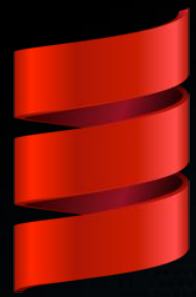


# Exemplo OO + Programação Funcional

```
case class Pessoa (nome: String, sobrenome: String, idade: Int) {  
    def concatenaTudo(a: String, b: String, c: Int, f: (String, String, Int) => String) = f(a, b, c)  
    def concatena (a: String, b: String, c: Int) = s"" "0 nome completo da pessoa é $a $b  
    |e a pessoa tem $c anos de idade"".stripMargin  
}  
  
val pessoa1 = Pessoa("Fernando", "Buzato", 32)  
  
println(pessoa1.concatenaTudo(pessoa1.nome, pessoa1.sobrenome, pessoa1.idade, pessoa1.concatena))
```

```
sbt: [info] Running Main  
0 nome completo da pessoa é Fernando Buzato  
e a pessoa tem 32 anos de idade
```

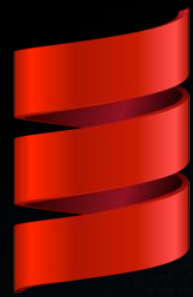




# Concorrência, Paralelismo e Distribuição

- Scala's Parallel Collections Library
- Futures
- Actors

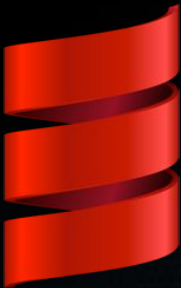




# Concorrência, Paralelismo e Distribuição

## Scala's Parallel Collection Library

- Scala como outras linguagens possui bibliotecas para trabalhar com coleções de objetos como Listas, Vetores, Mapas, etc.
- Scala criou uma abstração sobre as bibliotecas de Coleções permitindo o uso de paralelismo ao manipular estas coleções.



# Concorrência, Paralelismo e Distribuição

Scala's Parallel Collections Library (Exemplo):

## *Vetor Sequencial*

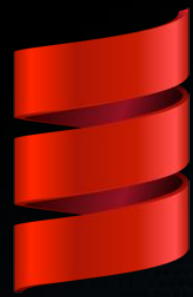
```
val v = Vector.range(0, 10)
v.foreach(println)
```

```
sbt: [info] Running Main
0
1
2
3
4
5
6
7
8
9
```

## *Vetor Paralelo*

```
val v = Vector.range(0, 10)
v.par.foreach(println)
```

```
sbt: [info] Running Main
7
2
0
5
6
3
4
1
8
9
```

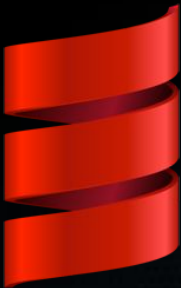


# Concorrência, Paralelismo e Distribuição

## Futures

- Futures proporcionam uma maneira de executar operações em paralelo de forma assíncrona e não bloqueante.
- Funciona como uma espécie de objeto reservado que é criado para um resultado que ainda não existe.
- Permite a criação de tarefas simultâneas de forma assíncrona.
- Permite a combinação das tarefas através de Maps, For-Comprehensions, Reduces de forma imutável e não blocante.

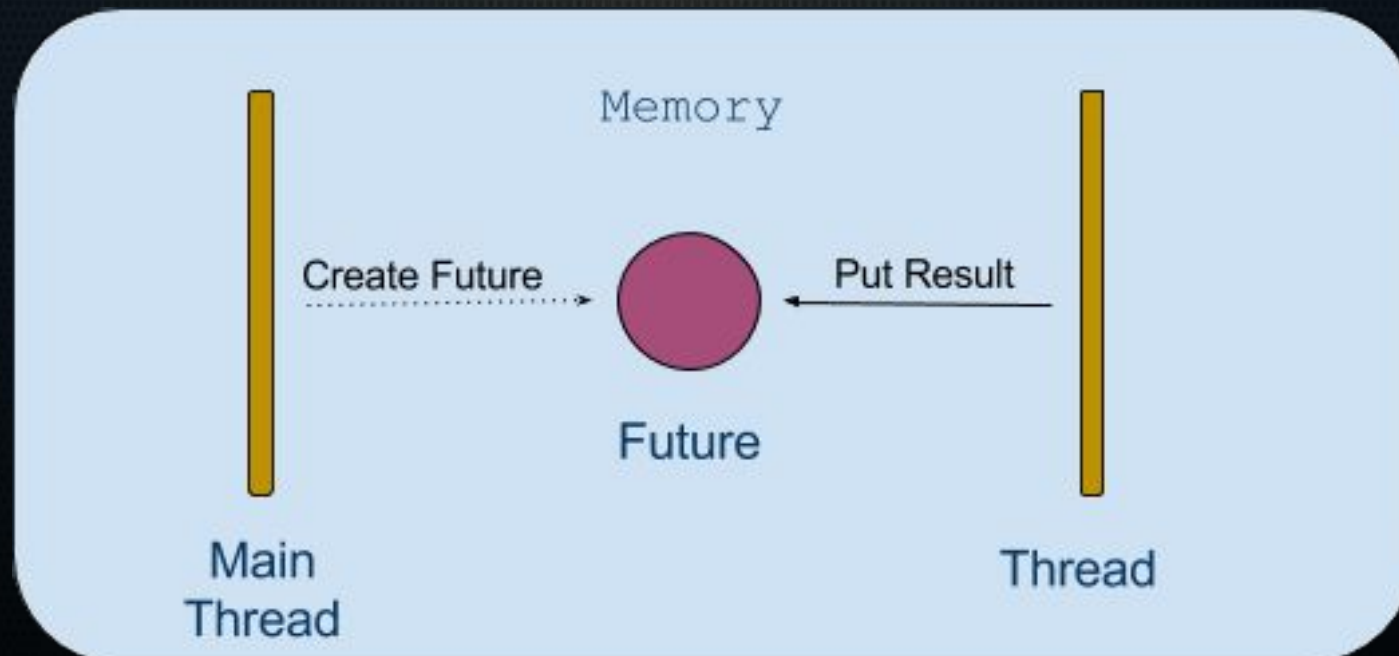




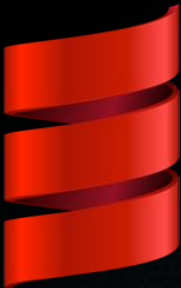
# Concorrência, Paralelismo e Distribuição

## Futures

- As Threads são criadas através de um ExecutionContext - Espécie de ThreadPool.
- Por padrão, o ExecutionContext cria uma Thread para cada CPU onde a JVM está executando.







# Concorrência, Paralelismo e Distribuição

## Futures (Exemplo):

```
def delta = System.currentTimeMillis - startTime
def time = System.currentTimeMillis
def sleep(time: Long): Unit = Thread.sleep(time)

def multiplicaPorDois(x: Int, delay: Int, name: String): Future[Int] = Future {
  println(s"Future $name")
  sleep(delay)
  x * 2
}

val startTime = time

// Future #1
val f1 = multiplicaPorDois(x=1, delay=10000, name="f1")

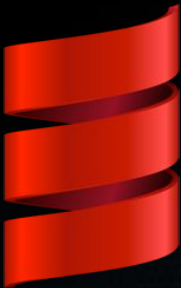
// Future #2
val f2 = multiplicaPorDois(x=2, delay=10000, name="f2")

// Future #3
val f3 = multiplicaPorDois(x=5, delay=10000, name="f3")

val resultado = for {
  r1 <- f1
  r2 <- f2
  r3 <- f3
} yield (r1 + r2 + r3)

resultado.onComplete {
  case Success(x) => {
    println(s"\nResultado = $x (tempo total(ms): = $delta)")
  }
  case Failure(e) => e.printStackTrace
}

val sync = Await.result(resultado, Duration.Inf)
```



# Concorrência, Paralelismo e Distribuição

## Futures (Exemplo):

```
def delta = System.currentTimeMillis - startTime
def time = System.currentTimeMillis
def sleep(time: Long): Unit = Thread.sleep(time)

def multiplicaPorDois(x: Int, delay: Int, name: String): Future[Int] = Future {
  println(s"Future $name")
  sleep(delay)
  x * 2
}

val startTime = time

// Future #1
val f1 = multiplicaPorDois(x=1, delay=10000, name="f1")

// Future #2
val f2 = multiplicaPorDois(x=2, delay=10000, name="f2")

// Future #3
val f3 = multiplicaPorDois(x=5, delay=10000, name="f3")

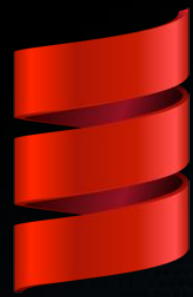
val resultado = for {
  r1 <- f1
  r2 <- f2
  r3 <- f3
} yield (r1 + r2 + r3)

resultado.onComplete {
  case Success(x) => {
    println(s"\nResultado = $x (tempo total(ms): = $delta)")
  }
  case Failure(e) => e.printStackTrace
}

val sync = Await.result(resultado, Duration.Inf)
```

```
Future f2
Future f3
Future f1
```

```
Resultado = 16 (tempo total(ms): = 10049)
```

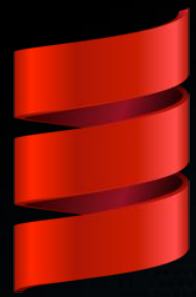


# Concorrência, Paralelismo e Distribuição

## Actors

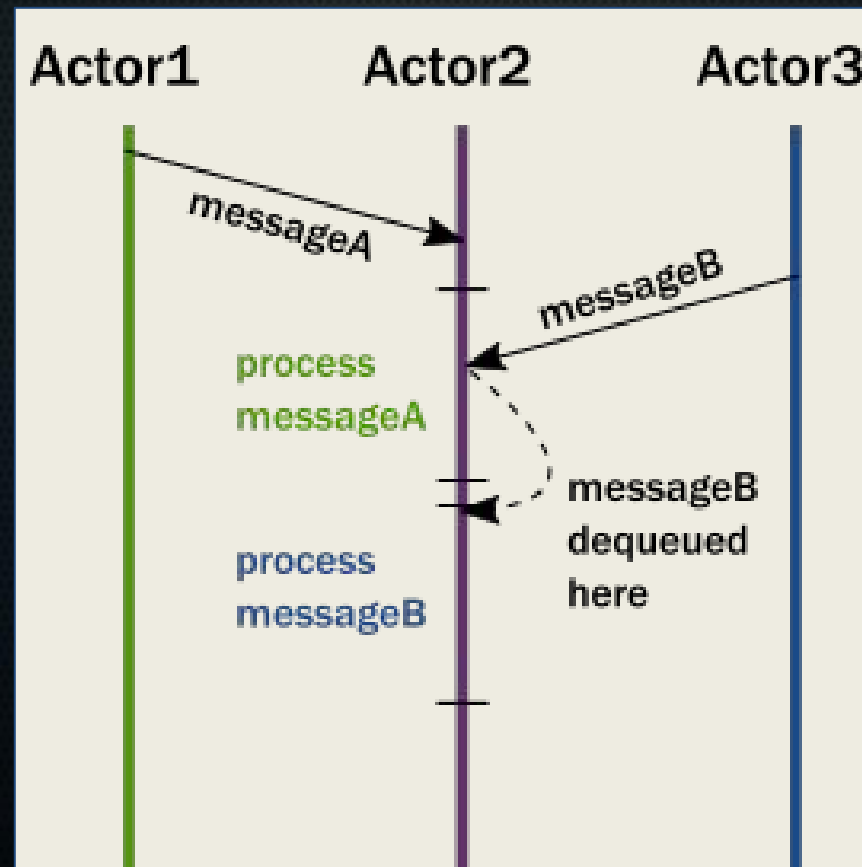
- Baseado no modelo Actor-Model (Modelo matemático para concorrência criado em 73 por Carl Hewitt).
- Fundamentalmente um Actor é um objeto que recebe uma mensagem e toma uma ação sobre aquela mensagem.
- Um Actor pode processar uma mensagem internamente, encaminhar uma mensagem para outro Actor, enviar uma nova mensagem, ou criar novos Actors para processar a mensagem recebida.
- Um Actor trabalha com uma mensagem de cada vez e sem troca de contexto.



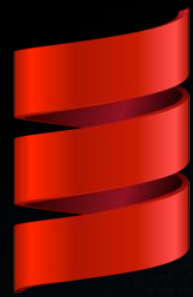


# Concorrência, Paralelismo e Distribuição

## Actors







# Concorrência, Paralelismo e Distribuição

## Actors

- Até a versão 2.10 da linguagem Scala, havia uma implementação para trabalhar com Actors dentro da biblioteca padrão da linguagem.
- Apartir da versão 2.11, a linguagem descontinuou a biblioteca e começou a adotar o framework Akka.
- O framework Akka foi criado pela Typesafe, empresa que mantém o Scala e o Akka.
- O framework pode ser utilizado para implementação do Actor-Model em Scala e Java.



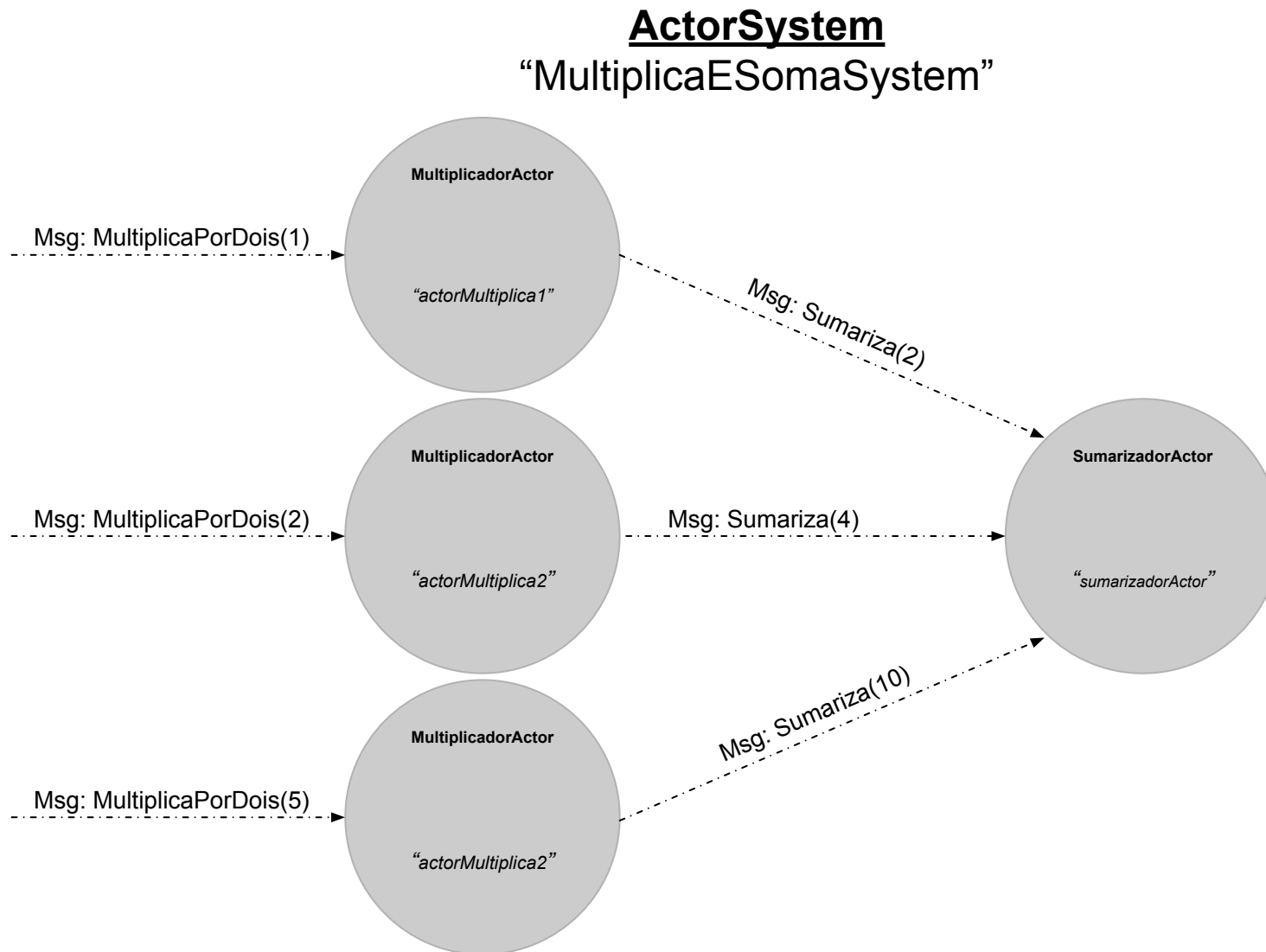
# Concorrência, Paralelismo e Distribuição

## Akka

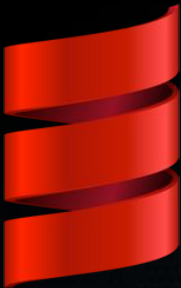
- É um toolkit para construção de aplicações que trabalham com alta concorrência, distribuição e tolerância a falha.
- Roda sobre a JVM e pode ser implementado na linguagem Scala ou Java.
- Possui vários módulos, entre os principais: Actor (core), Remoting, Cluster, Distributed Data e Streams.

# Concorrência, Paralelismo e Distribuição

Actors (Exemplo):







# Concorrência, Paralelismo e Distribuição

## Actors (Exemplo):

```
object TesteActor2 extends App {
  def delta = System.currentTimeMillis - startTime
  def time = System.currentTimeMillis

  case class MultiplicaPorDois (n:Int)
  case class Sumariza (n:Int)

  class MultiplicadorActor (sumarizadorActor: ActorRef) extends Actor {
    def receive = {
      case MultiplicaPorDois (n) =>
        println(self)
        println(n)
        Thread.sleep(10000)
        sumarizadorActor ! Sumariza (n * 2)
    }
  }

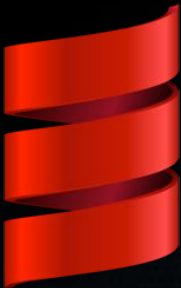
  class SumarizadorActor extends Actor {
    var sum = 0
    def receive = {
      case Sumariza (n) =>
        println (self)
        sum = sum + n
        println(s"Valor Total Somado: $sum")
        println(s"Tempo Total de Execução: $delta")
    }
  }

  val system = ActorSystem("MultiplicaESomaSystem")

  val sumarizadorActor: ActorRef = system.actorOf(Props(new SumarizadorActor), "sumarizadorActor")
  val actorMultiplica1 = system.actorOf(Props(new MultiplicadorActor(sumarizadorActor)), name = "actorMultiplica1")
  val actorMultiplica2 = system.actorOf(Props(new MultiplicadorActor(sumarizadorActor)), name = "actorMultiplica2")
  val actorMultiplica3 = system.actorOf(Props(new MultiplicadorActor(sumarizadorActor)), name = "actorMultiplica3")

  val startTime = time

  actorMultiplica1 ! MultiplicaPorDois(1)
  actorMultiplica2 ! MultiplicaPorDois(2)
  actorMultiplica3 ! MultiplicaPorDois(5)
}
```



# Concorrência, Paralelismo e Distribuição

## Actors (Exemplo):

```
object TesteActor2 extends App {
  def delta = System.currentTimeMillis - startTime
  def time = System.currentTimeMillis

  case class MultiplicaPorDois (n:Int)
  case class Sumariza (n:Int)

  class MultiplicadorActor (sumarizadorActor: ActorRef) extends Actor {
    def receive = {
      case Actor[akka://MultiplicaESomaSystem/user/actorMultiplica1#-220920012]
      1
      Actor[akka://MultiplicaESomaSystem/user/actorMultiplica3#1947027828]
      5
      Actor[akka://MultiplicaESomaSystem/user/actorMultiplica2#-843257318]
      2
    }
  }
  class SumarizadorActor extends Actor {
    var valorTotal = 0
    def receive = {
      case Actor[akka://MultiplicaESomaSystem/user/sumarizadorActor#-500116476]
      Valor Total Somado: 2
      Tempo Total de Execução: 10016
      Actor[akka://MultiplicaESomaSystem/user/sumarizadorActor#-500116476]
      Valor Total Somado: 12
      Tempo Total de Execução: 10016
    }
  }
  Actor[akka://MultiplicaESomaSystem/user/sumarizadorActor#-500116476]
  Valor Total Somado: 16
  Tempo Total de Execução: 10017

  val sumarizadorActor: ActorRef = system.actorOf(Props(new SumarizadorActor), "sumarizadorActor")
  val actorMultiplica1 = system.actorOf(Props(new MultiplicadorActor(sumarizadorActor)), name = "actorMultiplica1")
  val actorMultiplica2 = system.actorOf(Props(new MultiplicadorActor(sumarizadorActor)), name = "actorMultiplica2")
  val actorMultiplica3 = system.actorOf(Props(new MultiplicadorActor(sumarizadorActor)), name = "actorMultiplica3")

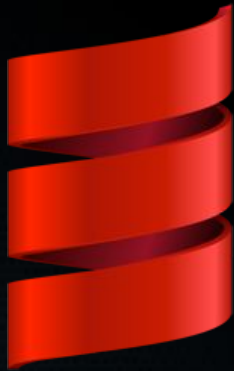
  val startTime = time

  actorMultiplica1 ! MultiplicaPorDois(1)
  actorMultiplica2 ! MultiplicaPorDois(2)
  actorMultiplica3 ! MultiplicaPorDois(5)
}
```





# Bibliografia



- ODESKY, Martin; SPOON, Lex; VENNERS, Bill. *Programming in Scala*. 3 ed. Mountain View: Artima, 2016.
- PAYNE, Alex; WAMPLER, Dean. *Programming Scala*. 2 ed. O'Reilly Media, Inc, 2014.
- THE SCALA PROGRAMMING LANGUAGE. em: <<https://www.scala-lang.org/>>. Acesso em 18 de jun. 2017
- AKKA. em <<http://akka.io/docs/>>. Acesso em 18 de jun. 2017
- Larson, Jim. *Erlang for Concurrent Programming*. em <<http://queue.acm.org/detail.cfm?id=1454463>>. Acesso em 18 de jun. 2017
- Armstrong, Joe. *Erlang: Software for a Concurrent World (apresentação em vídeo)*. em <<https://www.infoq.com/presentations/erlang-software-for-a-concurrent-world>>. Acesso em 18 de jun. 2017
- PROKOPEC, Aleksandar. *Learning Concurrent Programming in Scala*. PACKT, 2014





Obrigado