A linguagem Scala e o arcabouço Akka

Nelson Lago

MAC 5742

Scala

- **Linguagem multi-paradigma**
  - Funcional e orientada a objetos
  - A ideia é induzir programadores a migrar gradativamente para o modelo funcional
- **Roda na JVM**
  - Integração “transparente” com bibliotecas Java
  - Desempenho similar ao do Java
  - Pode rodar no Android
- **Criada em 2001 – 2003**

Scala

```
object Olá {
    def main (Array[String]): Unit = {
        println(“Alô, galera!”)
    }
}
```

- Compilação: scalac
- Execução: scala Olá
- Pode usar ant/maven, mas tem sistema de build próprio (sbt)
- Várias ferramentas e arcabouços extras

Scala

- **Fortemente tipada**
- **Inferência de tipos**
  - Muitas vezes não é preciso ser explícito
  - var x = 6 // x é um inteiro!

```
class Point(xc: Int, yc: Int) {
  var x = xc
  var y = yc
  def move(dx: Int, dy: Int) {
    x = x + dx
    y = y + dy
  }
  override def toString(): String = "(" + x + ", " + y + ")";
}
```

- **Funções são objetos como quaisquer outros**
- **Vários recursos típicos de linguagens funcionais**
- **Amor pelos tipos imutáveis**

```
def oncePerSecond(callback: () => Unit): Unit = {
    while (true) {
        callback()
        Thread.sleep(1000)
    }
}
oncePerSecond(() => println("Time goes by!"))
```

Modelo de atores

- **Atores rodam concorrentemente de forma assíncrona**
  - Não há sincronização
  - Não há estado compartilhado
  - Mensagens são imutáveis
- **Comunicação apenas através de mensagens**
- **Pode haver milhões de atores em um sistema, pois eles são “baratos”**
- **Permite concorrência e processamento distribuído de forma transparente para o programador**
- **Não há garantias sobre a ordem de entrega das mensagens**
  - Apenas é garantido que duas mensagens enviadas por um ator A para um ator B são entregues nessa ordem

Modelo de atores

- **Atores têm**
  - Caixa de entrada
  - Estado
  - Comportamento
  - Filhos

Akka

- **Arcabouço para criação de aplicações concorrentes, paralelas e distribuídas**
- **Fortemente baseado no modelo de atores**

```
class MyActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}
```

- **“Tell” (!) e “Ask” (?)**
  - Mas mesmo ask depende de troca de mensagens, não é um “retorno” do método!

```
// Tipos de mensagens
case object Greet
case class WhoToGreet(who: String)
case class Greeting(message: String)
class Greeter extends Actor {
  var greeting = ""
  def receive = {
    case WhoToGreet(who) => greeting = s"hello, $who"
    case Greet           => sender ! Greeting(greeting)
  }
}
// Inicialização do sistema de atores e criação do ator
val system = ActorSystem("helloakka")
val greeter = system.actorOf(Props[Greeter], "greeter")
// Envio da mensagem
greeter ! WhoToGreet("akka")
greeter ! Greet
```