A linguagem de programação Scala e o arcabouço para concorrência Akka

Nelson Lago — MAC 5742

1 Introdução

A linguagem de programação Scala é uma linguagem multi-paradigma, oferecendo recursos de programação orientada a objetos e de programação funcional [7]. Ela é compilada para bytecodes Java para interpretação pela JVM, podendo também rodar no Android, e tem desempenho similar ao de Java. Classes escritas em Java e em Scala podem ser misturadas livremente, o que permite o uso de bibliotecas, arcabouços e outras ferramentas Java existentes diretamente em Scala. Ao mesmo tempo, ferramentas desenvolvidas em Scala podem ser utilizadas em Java, como é o caso do Akka [10].

A linguagem e suas principais ferramentas são software livre, disponíveis sob licenças permissivas (Apache e BSD). Ela foi concebida em 2001 e sua versão inicial foi lançada em 2003 [6, 3]. Segundo o criador, sua experiência anterior com Funnel [1], outra linguagem criada por ele, o convenceu que o sucesso de uma nova linguagem depende da existência de um grande conjunto de bibliotecas-padrão; esse foi um dos fatores que incentivou a adoção de um modelo que permitisse o uso de bibliotecas Java existentes, mas a própria biblioteca-padrão de Scala também é bastante abrangente.

Scala é fortemente tipada e totalmente orientada a objetos: todo valor é um objeto e toda operação é uma chamada de método. Ao mesmo tempo, Scala é também uma linguagem funcional completa: funções são entidades de primeira classe, a biblioteca-padrão oferece estruturas de dados imutáveis, há suporte nativo para o casamento de padrões e o estilo da linguagem favorece a imutabilidade. Essas características foram incorporadas no intuito de favorecer técnicas de programação mais próximas do modelo funcional, geralmente mais afeitas ao processamento paralelo, dado que um dos objetivos principais da linguagem é simplificar o desenvolvimento de aplicações paralelas. Segundo os autores, novos usuários tendem a "migrar" gradativamente de um estilo de programação similar ao do Java para um estilo mais funcional.

1.1 Tecnologias relacionadas

Programas escritos em Scala podem ser compilados com ferramentas usuais de Java, como maven ou ant, mas mais comumente é usado o **sbt** [12], ferramenta de *build* específica de Scala. Há diversas bibliotecas e arcabouços interessantes disponíveis para a linguagem; dois deles são o arcabouço Play [11], para o desenvolvimento de aplicações *web*, e o arcabouço de concorrência Akka; este último foi recentemente incorporado à biblioteca padrão da linguagem.

```
println(if (4 > 3) "4 > 3 is true" else "4 > 3 is false")
```

Figura 1: A última expressão em um escopo é o valor do escopo.

A empresa Typesafe Inc. hoje é uma das mantenedoras da linguagem, participando também do desenvolvimento dessas ferramentas, e oferece serviços, componentes e ferramentas adicionais (também como software livre). Em particular, eles oferecem o Activator [13], um sistema para construção de pacotes baseado no sbt com modelos pré-prontos de projetos e com uma interface de usuário que, na presente versão, inclui alguns tutoriais [14].

2 Sintaxe básica[1]

Qualquer valor em Scala (incluindo números, funções etc.) é um objeto, ou seja, qualquer valor é uma instância de uma classe. A classe raiz da hierarquia é `scala.Any`, que tem duas subclasses principais: `scala.AnyVal` (números) e `scala.AnyRef` (demais objetos), que corresponde a `java.lang.Object`.

Em Scala, a última expressão em uma função ou método (na verdade, em qualquer escopo) é o seu valor, como se pode ver na Figura 1. Assim, em várias situações, não é necessário utilizar a palavra `return` explicitamente ao final de um método. De fato, uso de `return` é desencorajado, pois trata-se na verdade de um comando com significado similar a `break` [5]. Scala também diferencia variáveis, declaradas com a palavra-chave `var`, de valores imutáveis, declarados com a palavra-chave `val`. Embora seja estaticamente tipada, Scala é capaz de realizar inferência de tipos, o que significa que em diversas situações não é preciso declarar tipos explicitamente, como visto nas variáveis x e y na Figura 2. Finalmente, chamadas de métodos sem parâmetros não necessitam de parênteses.

No exemplo da Figura 2, podemos ver a definição de uma classe (`Point()`). A declaração da classe inclui o construtor, que recebe dois inteiros como parâmetros. Métodos herdados de uma superclasse que precisam ser redefinidos, como `toString()` no exemplo, devem ser declarados com a palavra `override`. Classes podem ser parametrizáveis (como os *Generics* do Java) e Scala também possui *Traits*, que são similares a *Interfaces* em Java mas com a possibilidade de ter implementações parciais. Traits permitem o uso de *composição misturada* (*mixin composition*), uma forma de viabilizar herança múltipla sem os problemas normalmente associados a ela.

Como mencionado, funções e métodos em Scala são objetos como quaisquer outros, o que faz o estilo de programação funcional ser natural em Scala. No exemplo da Figura 3, `() => println("Time goes by!")` é a declaração de uma função anônima que é passada como parâmetro para a função `oncePerSecond`. Por sua vez, a definição de `oncePerSecond` determina que ela recebe como parâmetro uma função sem parâmetros e que devolve Unit (ou seja, cujo valor devolvido é irrelevante, de forma similar a void em Java) — e `() => println("Time goes by!")` corresponde exatamente a isso. Assim, se `oncePerSecond` for exe-

[1] Baseado no tutorial de Scala [2], no Wikibook de Scala [15], no artigo da IBM Developerworks [4] e no livro do criador de Scala [8].

```
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 + ")";
}
```

Figura 2: Definição de uma classe (`Point()`) em Scala.

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

Figura 3: Funções como parâmetros.

cutada, a função anônima será executada periodicamente. Esse tipo de construção permite abstrações interessantes, como visto na Figura 4.

3 Akka e o modelo de atores[2]

Uma abordagem que tem ganhado popularidade para a programação paralela e concorrente é o modelo de atores, introduzido comercialmente com a linguagem Erlang. O modelo é baseado em eventos e um ator é uma entidade que, ao receber uma mensagem:

- Envia mensagens a outros atores

[2]Baseado na documentação do Akka [9].

```
val list = List(1, 2, 3, 4)
if (list.exists((a:Int) => a == 4))
  println("Existe ao menos um elemento igual a 4")

val even = list.filter((a:Int) => (a % 2) == 0)
println(even)

val double = list.map((a:Int) => a * 2)
println(double)
```

Figura 4: Utilizando funções ao invés de iteradores em coleções.

- Cria novos atores
- Determina o comportamento que ele tomará ao receber a próxima mensagem

Atores possuem:

- Uma caixa de entrada de mensagens
- Estado
- Comportamento (que pode mudar dependendo do estado do ator)
- Filhos (outros atores)

A comunicação entre os atores ocorre exclusivamente através da troca de mensagens; a execução dos atores e a troca de mensagens entre eles é totalmente assíncrona. No modelo, atores são pequenos e criá-los e destruí-los são operações baratas e é normal que haja um grande número (milhões) de atores em um sistema ("tudo são atores"). Como não há sincronização explícita (semáforos etc.) nem estado compartilhado, programas concorrentes complexos capazes de ser executados de forma distribuída e com boa tolerância a falhas podem ser desenvolvidos mais facilmente nesse modelo.

Akka [10] é um arcabouço para programação concorrente escrito em Scala (mas também disponível para Java) com ênfase no modelo de atores. O desenvolvimento de uma aplicação usando Akka não leva em conta a localidade, ou seja, não há distinção entre atores "locais" e "remotos": o arcabouço permite que atores em execução em uma única máquina ou em várias trabalhem em conjunto de maneira transparente para o programador. Apenas a configuração determina como Akka vai decidir onde instanciar este ou aquele ator. A interconexão entre nós separados segue um modelo *peer-to-peer*: qualquer nó pode iniciar uma comunicação com qualquer outro e a comunicação é bidirecional. Ou seja, não faz sentido pensar em dividir os nós seguindo uma arquitetura cliente-servidor (isso pode ser feito de outras formas).

Um ator pode delegar uma tarefa a um outro ator "filho"; nesse caso, ele assume o papel de *supervisor* desse filho, e todo ator tem exatamente um supervisor. No caso de uma falha de um filho, o supervisor pode reiniciar o processamento do filho, tratar a falha de alguma outra maneira ou falhar ele mesmo, delegando o tratamento da falha para seu próprio supervisor.

Finalmente, cada ator pode ter um estado interno (e seu comportamento pode mudar em consequência disso), mas não há estado compartilhado e todas as mensagens são imutáveis. Não há garantias sobre a ordem de chegada das mensagens, mas duas mensagens enviadas por um dado ator para um outro são entregues em ordem (mas possivelmente com outras mensagens de outros atores entre elas). Como o envio de mensagens é assíncrono, os atores não devolvem nada ao fim do tratamento de uma mensagem; eles podem apenas enviar outra mensagem.

Com Akka, uma nova classe de atores é definida estendendo a classe `Actor` e implementando seu método `receive`. Assim, um ator não possui uma interface ou API definida nos termos da linguagem; ao invés disso, Akka faz uso do recurso de casamento de padrões de Scala: padrões são definidos para identificar os tipos de mensagens que o ator pode processar e, para cada padrão, é definida

```
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

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

Figura 5: Sintaxe para a definição de uma nova classe de atores.

a implementação do que o ator deve fazer, como visto no exemplo da Figura 5. Scala ainda dá suporte a classes `case`, que permitem realizar casamentos de padrões desse tipo de maneira simples (que é o método usado no próximo exemplo). O envio de mensagens pode ser feito de duas formas: "tell" (!) e "ask" (?). Tell significa enviar a mensagem de forma assíncrona; ask significa enviar a mensagem de forma assíncrona e solicitar uma mensagem de resposta, que é convenientemente armazenada em uma variável do tipo `Future`.

Como exemplo, imaginemos um ator cuja função é responder a mensagens com um cumprimento, como visto na Figura 6

4 Conclusões

Scala foi criada com vistas a aproximar o modelo de programação funcional dos programadores acostumados com linguagens imperativas, em particular Java. Entre as supostas vantagens do modelo funcional está uma maior facilidade para o desenvolvimento de programas paralelos, concorrentes e distribuídos. De fato, várias características de Scala herdadas do modelo funcional foram criadas com esse propósito. O modelo de atores faz parte da biblioteca padrão de Scala há muito tempo, mas a implementação original foi subtituída por Akka devido à sua crescente popularidade e flexibilidade. O uso de Scala e Akka em conjunto parece oferecer um ambiente de desenvolvimento favorável para a criação de aplicações distribuídas complexas, como imaginado por seus criadores.

Referências

[1] École Polytechnique Fédérale de Lausanne. *Functional Nets (Funnel Language Homepage)*. URL: `http://lampwww.epfl.ch/funnel/` (acesso em 26/06/2015).

[2] École Polytechnique Fédérale de Lausanne. *Introduction - Scala Documentation*. 2013. URL: `http://docs.scala-lang.org/tutorials/tour/tour-of-scala.html` (acesso em 28/06/2015).

```
// 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")
// Isto só funciona a partir de outro ator,
// porque é preciso haver um "sender"
greeter ! Greet
```

Figura 6: Um ator simples.

[3] École Polytechnique Fédérale de Lausanne. *The Scala Programming Language Homepage*. 2015. URL: http://www.scala-lang.org/ (acesso em 26/06/2015).

[4] Ted Neward. *The busy Java developer's guide to Scala: Functional programming for the object oriented*. 22 de jan. de 2008. URL: http://www.ibm.com/developerworks/java/library/j-scala01228/index.html (acesso em 28/06/2015).

[5] Rob Norris. *Don't use return in Scala*. 2013. URL: https://tpolecat.github.io/2014/05/09/return.html (acesso em 28/06/2015).

[6] Martin Odersky. *A Brief History of Scala*. 9 de jun. de 2006. URL: http://www.artima.com/weblogs/viewpost.jsp?thread=163733 (acesso em 26/06/2015).

[7] Martin Odersky. *What is Scala?* The Scala Programming Language. URL: http://www.scala-lang.org/what-is-scala.html (acesso em 26/06/2015).

[8] Martin Odersky, Lex Spoon e Bill Venners. *Programming in Scala, First Edition*. 2008. URL: http://www.artima.com/pins1ed/ (acesso em 29/06/2015).

[9] Typesafe Inc. *Akka Documentation for Scala*. 2014. URL: http://doc.akka.io/docs/akka/2.3.11/scala.html?_ga=1.39591850.845937081.1435329202 (acesso em 27/06/2015).

[10] Typesafe Inc. *Akka Homepage*. 2015. URL: http://akka.io/ (acesso em 26/06/2015).

[11] Typesafe Inc. *Play Framework - Build Modern & Scalable Web Apps with Java and Scala*. URL: https://www.playframework.com/ (acesso em 28/06/2015).

[12] Typesafe Inc. *sbt - The interactive build tool*. 2015. URL: http://www.scala-sbt.org/ (acesso em 28/06/2015).

[13] Typesafe Inc. *Typesafe Activator and sbt get you started with the Typesafe Reactive Platform*. URL: https://www.typesafe.com/community/core-tools/activator-and-sbt (acesso em 26/06/2015).

[14] Dick Wall. *Typesafe Activator, What *is* it? (and why should I care?)* 7 de out. de 2014. URL: https://www.typesafe.com/blog/typesafe-activator-what-is-it (acesso em 26/06/2015).

[15] Wikimedia Foundation. *Scala - Wikibooks, open books for an open world*. URL: https://en.wikibooks.org/wiki/Scala (acesso em 28/06/2015).