Uma introdução à linguagem Scala: mais uma linguagem JVM ou o futuro de JAVA?

Filipe Ferraz Salgado
filipefs@ime.usp.br
08/12/2008

Motivação

Atualmente

- Várias linguagens:
  - JavaScript (cliente)
  - Perl/Python/Ruby/Groovy (script no lado do servidor)
  - JavaFX (UI)
  - Java (lógica de negócios)
  - SQL (banco de dados)

Vantagens e desvantagens

+ Cada uma é usada naquilo que é melhor

- As linguagens precisam se comunicar

  - Geralmente a comunicação é feita através de XML ou strings. Isso torna a manutenção da aplicação difícil.

Ideal

- Uma linguagem que consiga descrever tanto pequenos quanto grandes componentes

- Que utilize bibliotecas ao invés de outras linguagens para desenvolver módulos específicos nas aplicações

Pra que mais uma linguagem?

- O objetivo é criar uma linguagem que suporte melhor aplicações baseadas em componentes

- Scala é baseada em duas hipóteses:
  - Uma linguagem para aplicações baseadas em componentes deve ser escalável
  - Tal escalabilidade pode ser alcançada através da unificação e generalização dos conceitos de programação orientada a objetos e funcional

Scala

Scala é funcional

- Funções de ordem superior

```
def exists[T] ( xs: Array[T], p: T => boolean ) = {
    var i: int = 0
    while (i < xs.length && !p(xs(i))) i = i + 1
    i < xs.length
}
```

- Tipos de dados algébricos (TDA) e casamento de padrões
- Polimorfismo paramétrico

Scala é orientada a objetos

- Herança e subtipos
- Todo valor é um objeto
- Toda operação é uma chamada de método

`1 + 2`  `1.+(2)`

Como eles fizeram isso?

- Concentraram-se em 3 conceitos:
  - Abstração: membros abstratos e *generics*
  - Composição: *objects*, *classes* e *traits*
  - Decomposição: TDA e casamento de padrões

Abstração

Membros abstratos

- Tipicamente orientado a objetos
- Pode haver um membro abstrato que é um tipo e um membro abstrato que é um valor

```
abstract class AbsCell {
    type T;
    val init: T;
    private var value: T = init;
    def get: T = value;
    def set(x: T): unit = { value = x }
}
val cell = new AbsCell {  type T = int; val init  = 1 }
```

Generics

- Tipicamente funcional
- A abstração da classe e dos argumentos dos métodos é feita através de parâmetros que são tipos

```
def swap[T](x: GenCell[T], y: GenCell[T]): unit = {
    val t = x.get; x.set(y.get); y.set(t)
}
```

Limite de tipos

- Tanto em membros abstratos quanto em generics é possível restringir as especializações que poderão ser usadas

```
abstract class Ordered {
    type O;
    def < (that: O): boolean;
    def <= (that: O): boolean =
        this < that || this == that
}
abstract class MaxCell extends AbsCell {
    type T <: Ordered { type O = T }
    def setMax(x: T) = if (get < x) set(x)
}
```

Self-type

- Modifica o tipo de *this* (se não for usado, o tipo de *this* é o da classe declarada)
- Foi introduzido por motivos técnicos, mas acabou sendo bastante utilizado

```
abstract class Graph {
    type Node <: BaseNode;
    class BaseNode requires Node {
        def connectWith(n: Node): Edge = new Edge(this, n);
    }
    class Edge(from: Node, to: Node) {
        def source() = from;
        def target() = to;
    }
}
```

Self-type

- A consistência de tipos é garantida por dois requisitos:

1) o *self-type* de uma classe deve ser um subtipo dos *self-types* de todas as suas classes base

2) quando uma classe é instanciada, é verificado se o *self-type* da classe é um supertipo do tipo do objeto que está sendo criado

Composição

Objects, classes e traits

- Object: classe com uma única instância (singleton)

- Trait: Forma especial de classe abstrata. Pode ser utilizada em qualquer situação onde se utilizaria uma classe abstrata comum. Entretanto, apenas *trait* pode ser usada como *mixin*.

Composição *mixin*

- Espécie de herança múltipla
- Herda os métodos da superclasse + os novos das classes *mixin*
- Mixin: Classe que aparece associada à palavra with na declaração de classes.

Composição *mixin*

```
object Test {
    def main(args: Array[String]): unit = {
        class Iter extends StringIterator(args(0)) with RichIterator[char]
        val iter = new Iter
        iter foreach System.out.println
    }
}
```

```
class StringIterator(s: String) extends
        AbsIterator {
    type T = char;
    private var i = 0;
    def hasNext = i < s.length();
    def next = { val x = s.charAt(i); i = i + 1; x }
}
```

```
trait RichIterator extends AbsIterator {
def foreach(f: T => unit): unit =
while (hasNext) f(next);
}
```

```
trait AbsIterator {
    type T;
    def hasNext: boolean;
    def next: T;
}
```

Linearização...

- São as classes alcançáveis na hierarquia de classes de Scala a partir de uma classe C

$$L(\text{Iter}) = \{\text{Iter}\} \vec{+} L(\text{RichIterator}) \vec{+} L(\text{StringIterator})$$

{ Iter, RichIterator, StringIterator, AbsIterator, AnyRef, Any }

... é utilizada em

- Chamadas *super*
  - Não são resolvidas estaticamente, depende da composição na qual está sendo usada
  - Ex: StringIterator(string) with SyncIterator with RichIterator
- Escolha de métodos que se sobrescrevem
  - Concretos sobre abstratos
  - Aquele que vier antes na linearização caso sejam ambos concretos ou abstratos
- Tudo que dependa da ordem das classes

Decomposição

TDA e casamento de padrões

- Geralmente criticado por desenvolvedores que gostam de orientação a objetos
  - TDA's não são extensíveis
  - TDA's violam a pureza do modelo de dados de OO
  - Casamento de padrões quebra encapsulamento

TDA's e casamento de padrões em Scala

```
abstract class Term
case class Num(x: int) extends Term
case class Plus(left: Term, right: Term) extends Term

object Interpreter {
    def eval(term: Term): int = term match {
        case Num(x) => x
        case Plus(left, right) => eval(left) + eval(right)
    }
}
```

TDA's e casamento de padrões em Scala

- Pureza: todos os *cases* são classes e objetos
- Extensibilidade: pode-se definir mais *cases* em outros lugares
- Encapsulamento: apenas os parâmetros das classes *case* são mostrados

Outras características

XML

- É possível escrever em XML diretamente nos programas Scala
- Também é possível escrever usando Scala dentro do XML

```
val labPhoneBook =
    <phonebook>
    <descr>Phone numbers of<b>XML</b> hackers.</descr>
    <entry>
        <name>Burak</name>
        <phone where="work"> +41 21 693 68 67 </phone>
        <date> { df.format(new java.util.Date()) } </date>
    </entry>
    </phonebook>;
```

XML

- Decomposição dos nós através de casamento de padrões

```
import scala.xml.Node ;

def add(phonebook: Node, newEntry: Node): Node =
    phonebook match {
        case <phonebook>{ cs @ _* }</phonebook> =>
            <phonebook>{ cs }{ newEntry }</phonebook>
    }

val newPhoneBook =
    add(scala.xml.XML.loadFile("savedPhoneBook"),
        <entry>
            <name>Sebastian</name>
            <phone where="work">+41 21 693 68 67</phone>
        </entry>);
```

Views

- Converte um formato em outro
- São inseridas automaticamente pelo compilador
- Modificador `implicit`

Código

```
val s: Set = xs;
xs contains x;
```

Compilador

```
val s: Set = view(xs);
view(xs) contains x;
```

Por que bibliotecas ao invés de uma linguagem especializada?

- Poucas primitivas
  - Linguagem concisa e leve
- Programadores estendem e adaptam a linguagem de acordo com a necessidade
- Interoperabilidade

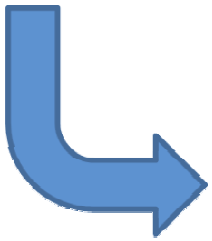
Bibliotecas são mais fáceis para estender e adaptar

- Biblioteca de atores
- Baseada nos atores de Erlang
- Um ator por *thread*
  - Caro
  - Não escalável

Atores

Envio da mensagem

pong ! *message*



```
case object Ping
case object Stop

class Pong extends Actor {
    def act() {
        var pongCount = 0
        while (true) {
            receive {
                case Ping =>
                    if (pongCount % 1000 == 0)
                        Console.println("Pong: ping "+pongCount)
                    pongCount = pongCount + 1
                case Stop =>
                    Console.println("Pong: stop")
                    exit()
            }
        }
    }
}
```

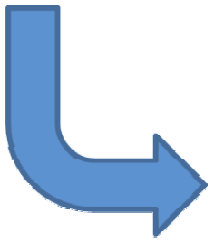
Atores baseados em eventos

- Sem inversão de controle
  - Não implica na reescrita da biblioteca
- Libera a *thread* quando o ator é bloqueado para receber uma mensagem
- Sem muita alteração no código do cliente

Atores baseados em eventos

Envio da mensagem

pong ! *message*



```
case object Ping
case object Stop

class Pong extends Actor {
    def act() {
        var pongCount = 0
        loop {
            react {
                case Ping =>
                    if (pongCount % 1000 == 0)
                        Console.println("Pong: ping "+pongCount)
                    pongCount = pongCount + 1
                case Stop =>
                    Console.println("Pong: stop")
                    exit()
            }
        }
    }
}
```

Modelo unificado de atores

- Permite usar tanto o modelo tradicional quanto o baseado em eventos
- *Trade-off* entre eficiência e flexibilidade de acordo com a necessidade
- Facilita o desenvolvimento de aplicações concorrentes, já que usa troca de mensagens

Interoperabilidade

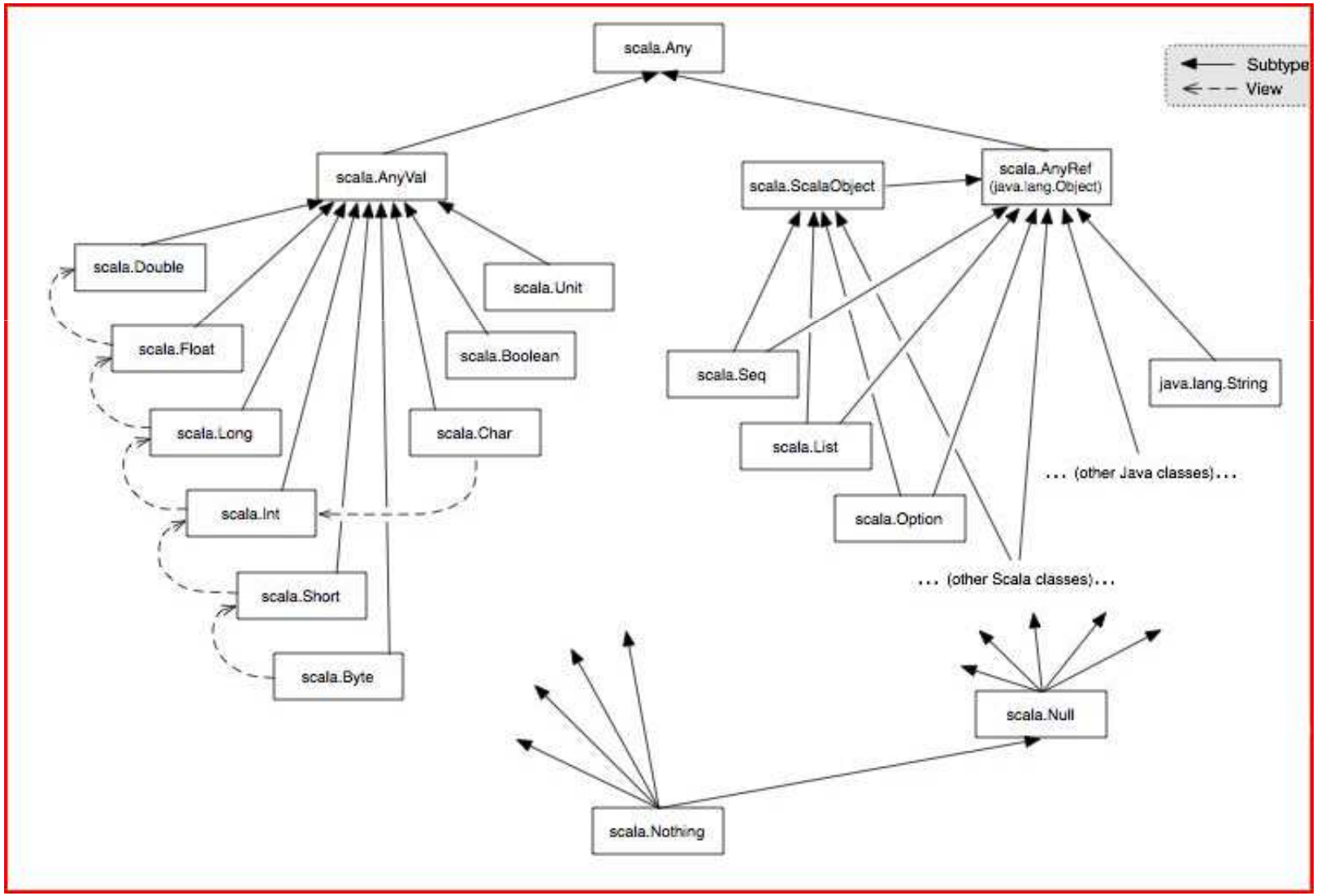
- Funciona normalmente com as bibliotecas de Java
- Importa automaticamente *java.lang*
- Reaproveitamento do código
- Facilita adoção dos usuários

Interoperabilidade

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
    def main(args: Array[String]) {
        val now = new Date
        val df = getDateInstance(LONG, Locale.FRANCE)
        println(df format now)
    }
}
```

Hierarquia de classes de Scala



Aplicação

Lift

- Web Framework
- Inspirado em arcabouços existentes como Rails e Seaside
- Utiliza várias funcionalidades de Scala:
  - Atores -> aplicações Ajax
  - Fechamentos -> elementos HTML
  - Traits -> persistência
- http://demo.liftweb.net/index

Conclusão

- Scala unifica POO e PF em uma linguagem estaticamente tipada
- Espera-se que isso facilite a criação de aplicações baseadas em componentes
- O tempo irá mostrar se é as hipóteses foram validadas ou se é apenas mais uma linguagem que roda na JVM

Experimente

- Scala, http://www.scala-lang.org
- Lift, http://liftweb.net/index.php/Main_Page

Fontes

- Odersky, M., Altherr, P., Cremet, V., Dragos, I., Dubochet, G., Emir, B., McDirmid, S., Micheloud, S., Mihaylov, N., Schinz, M., Spoon, L., Stenman, E., Zenger, M.: **An Overview of the Scala Programming Language (2. Edition). Technical Report** LAMP-REPORT, 2006.

- Odersky, M., Zenger, M.: **Scalable Component Abstractions. OOPSLA, páginas** 41-57, 2005.

- Haller, P., Odersky, M.: **Actors That Unify Threads and Events.** COORDINATION, páginas 171–190, 2007.

- Haller, P., Odersky, M.: **Event-Based Programming without Inversion of Control. LAMP-CONF-2006-004.**

Fontes

- http://www.infoq.com/presentations/jaoo-spoon-scala
- http://www.slideshare.net/pastas9/scala?src=related_normal&rel=585572
- http://www.slideshare.net/pastas9/google06
- Programming with Functional Objects in Scala
  TS-5165 Java One 2008