Scala Language

Dessì Massimiliano

Linux Day Cagliari
24 ottobre 2009

pro-netics group

Author

Software Architect and Engineer
ProNetics / Sourcesense

Presidente
JugSardegna Onlus

Fondatore e coordinatore
SpringFramework Italian User Group

Committer - Contributor
OpenNMS - MongoDB - MagicBox

Autore
Spring 2.5 Aspect Oriented Programming

Topics

Quanti core per processore ci sono sul server

su cui metti le tue applicazioni ?

La tua applicazione gestisce la concorrenza ?

Il tuo codice è scritto pensando alla scalabilità ?

Consideri questi aspetti quando scrivi il tuo codice ?

Se hai riposto affermativamente

fermati ad ascoltare questa presentazione

pro-netics group

SCAlable LAnguage

E' un linguaggio funzionale

ad oggetti puro.

Ogni cosa è un oggetto.

Non ci sono primitive.

Gira sulla Java Virtual Machine

pro-netics
group

Funzionale vs Imperativo

I linguaggi funzionali dicono

cosa deve fare il programma

Java e i linguaggi Object Oriented

sono imperativi,

dicono ***come*** deve farlo

pro-netics group

SCALA - best of both

In Scala, paradigma funzionale e Object oriented
sono complementari.
Invita ad utilizzare l'approccio funzionale,
ma non vincola le scelte.
Tutto è un oggetto
funzioni comprese.

pro-netics
group

Scala

Scala può essere utilizzato sia come linguaggio di scripting,

sia come linguaggio ad alta concorrenza

Possiede le caratteristiche di un linguaggio dinamico

Con la velocità di un linguaggio *statico* e *tipizzato*, grazie

alla inferenza si può scrivere codice

molto più coinciso

pro-netics group

Scala Inference

Scala è un linguaggio

low-ceremony, static typing

```
var anno: Int = 2009
var altroAnno = 2009
var saluto = "Ciao mondo"
```

quando definiamo il tipo
compare su un solo lato
nella assegnazione

```
var builder = new StringBuilder("hello" )
```

pro-netics group

Scala Inference

Scala attraverso l' inferenza, quando
non specificato, comprende
anche il tipo di ritorno dei metodi

```
def metodoUno() { 2 }
def metodoDue() = { 4 }
def metodoTre() = 6
def metodoQuattro : Double = 8
```

pro-netics group

Java

```
public class Persona {

    private String nome, cognome;
    private int eta;

    public Persona(String nome, int eta, String cognome) {
        this.nome = nome;
        this.eta = eta;
        this.cognome = cognome;
    }

    public String getNome() {
        return nome;
    }

    public int getEta() {
        return eta;
    }

    public void setEta(int eta) {
        this.eta = eta;
    }

    public String getCognome() {
        return cognome;
    }
}
```

pro-netics group

Scala

```
Case class Persona(val nome: String, var eta: Int, val cognome: String)
```

Il compilatore scala se non dichiariamo altro,

genera di default (Case Class):

`toString`,

`hashCode`,

`equals`,

Costruttore con i parametri dichiarati

getter dei parametri definti `val`

setter e getter dei parametri defini `var`

pro-netics group

Scala

Scalac Person.scala
javap -private Person

```
Compiled from "Person.scala"
public class org.magicbox.domain.Person extends java.lang.Object implements
scala.ScalaObject,scala.Product,java.io.Serializable{
    private final java.lang.String cognome;
    private int eta;
    private final java.lang.String nome;
    public org.magicbox.domain.Person(java.lang.String, int, java.lang.String);
    private final boolean gd1$1(java.lang.String, int, java.lang.String);
    public java.lang.Object productElement(int);
    public int productArity();
    public java.lang.String productPrefix();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public int hashCode();
    public int $tag();
    public java.lang.String cognome();
    public void eta_$eq(int);
    public int eta();
    public java.lang.String nome();
}
```

pro-netics group

Scala

Se invece vogliamo anche la convenzione classica JavaBean

get<nome>

set<nome>

Utilizziamo l'annotazione

```
@scala.reflect.BeanProperty
```

```
Case class Person(@scala.reflect.BeanProperty val nome: String,
                  @scala.reflect.BeanProperty var eta: Int,
                  @scala.reflect.BeanPropertyval cognome: String)
```

pro-netics group

Scala

```
scalac Person.scala
javap -private Person
```

```
Compiled from "Person.scala"
public class org.magicbox.domain.Person extends java.lang.Object implements
scala.ScalaObject,scala.Product,java.io.Serializable{
    private final java.lang.String cognome;
    private int eta;
    private final java.lang.String nome;
    public org.magicbox.domain.Person(java.lang.String, int, java.lang.String);
    private final boolean gd1$1(java.lang.String, int, java.lang.String);
    public java.lang.String getCognome();
    public void setEta(int);
    public int getEta();
    public java.lang.String getNome();
    public java.lang.Object productElement(int);
    public int productArity();
    public java.lang.String productPrefix();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public int hashCode();
    public int $tag();
    public java.lang.String cognome();
    public void eta_$eq(int);
    public int eta();
    public java.lang.String nome();
}
```

pro-netics group

Val e Var

Scala favorisce l' immutabilità.

Questo significa assegnare i valori definitivi al momento della creazione per non avere “side-effects”.

Per ottenere questo risultato utilizziamo la keyword `val`.

Quando usiamo `val`, il compilatore creerà una variabile final.

Se invece abbiamo necessità di poter riassegnare dei valori utilizziamo `var`, ma è consigliabile utilizzare il più possibile `val` per scrivere codice in stile funzionale

pro-netics group

Singleton

Posso avere una sola istanza (per classloader)
con la keyword `Object`,
l' oggetto è già pronto all' uso senza essere istanziato.

```
class Marcatore private(val color: String) {
    override def toString() : String = "colore marcatore " + color
}

object Marcatore {
    private val marcatori = Map(
        "red" -> new Marcatore("red" ),
        "blue" -> new Marcatore("blue" ),
        "green" -> new Marcatore("green" ))

    def getMarcatore(colore: String) =
    if (marcatori.contains(colore)) marcatori(colore) else null

}
```

In questo caso `object` Marcatore agisce anche da Companion
Object di `class` Marcatore

pro-netics group

Gerarchia e alcune keyword

`Any` è il tipo base

Classi figlie di Any sono `AnyVal` e `AnyRef`

`AnyVal` mappa i tipi primitivi Java

`AnyRef` mappa i tipi reference

Alla fine della gerarchia abbiamo `Nothing`

Unit corrisponde al void

pro-netics group

Traits

Sono come le interfacce Java ma con una parziale implementazione

Ci consentono di avere i benefici

della ereditarietà multipla,

ovvero avere classi con più funzionalità,

ma senza i corrispettivi problemi.

Vengono usate anche per la creazione di DSL

pro-netics group

Traits

```
trait Amico {
    val nome: String
    def ascolta() = println("Il tuo amico " + name + " ti ascolta" )
}

class Umano(val nome: String) extends Amico

class Uomo(override val nome: String) extends Umano(nome)

class Donna(override val nome: String) extends Umano(name)

class Animale


trait Verso {
    val verso: String
    def tono() = println("sgnanfuz" )
}

class Cane(val nome: String) extends Animale with Amico with Verso {
    override def ascolta = println(nome + " ascolta in silenzio" )
    override def tono = println("basso" )
     val verso = "bau bau"
}
```

pro-netics group

Pattern Matching e XML

Scala utilizza il pattern matching

in maniera molto estesa, l'utilizzo

più comune è quello

con gli Actor che vedremo nelle pagine seguenti.

Per l' XML Scala fornisce il supporto nativo per la dichiarazione,

lettura, lettura da file e ovviamente creazione/scrittura.

pro-netics group

Collection

Scala fornisce collection basate su quelle Java, sia di tipo immutabile che mutabile.

I tipi principali sono map, sequenze e liste, ma tutte arricchite nelle funzionalità.

Anche nel caso delle collection lo scopo è fornire collezioni per l'alta concorrenza.

pro-netics group

Function value

Possiamo passare una funziona come parametro

```
def store(arr: Array[Int], start: Int, funzione: (Int, Int) => Int) :
Int = {
    var precedente = start
    arr.foreach(element => precedente = funzione(precedente, element) )
    precedente
}
```

Passando funzioni anonime diverse possiamo avere:

```
scala> val array = Array(2, 3, 5, 1, 6, 4, 13)

scala> val somma = store(array, 0, (precedente, elem) => precedente + elem)
somma: Int = 34

scala> val max = store(array, Integer.MIN_VALUE, (precedente, elem) =>
Math.max(precedente, elem))
max: Int = 13
```

pro-netics group

Function value

Possiamo anche ordinare meglio riusando le funzioni

```
def somma(precedente: Int, elem : Int) = { precedente + elem }

def max(precedente: Int, elem : Int) = { Math.max(precedente, elem)}


scala> val sum = store(array, 0, somma)
sum: Int = 34

scala> val massimo = store(array, Integer.MIN_VALUE, max)
massimo: Int = 13
```

pro-netics group

Concorrenza

Scala Concurrency

Scala rende semplice la programmazione multithread.

I thread comunicano tra loro

utilizzando un modello ad eventi

per inviarsi oggetti immutabili come messaggi.

pro-netics group

Actors

Gli actors sono unità di esecuzione

threadless e stackless

che processano messaggi (eventi)

in maniera seriale.

Un actor riceve i messaggi nella sua mailbox,

li processa uno alla volta

in maniera asincrona

prendendoli dalla mailbox.

pro-netics group

Actors

Un actor non esponendo nessuno stato
e venendo modificato o interrogato
attraverso i messaggi,
che sono processati in maniera seriale
non ha bisogno di lock sul suo stato interno
ed è perciò thread safe

pro-netics group

Actor

```
class FooActor extends Actor {

  def act() = {
    var control = true
    while (control) {
      react {
        case "mailbox" => println("Mailbox: " + mailbox.toString)
        case "quit" =>  {
            println("spengo questo FooActor")
            control = false
          }
        case i: Int => println("Ho ricevuto un Int: " + i.toString)
        case _ => println("Nulla di particolare.")
      }
    }
  }
}
```

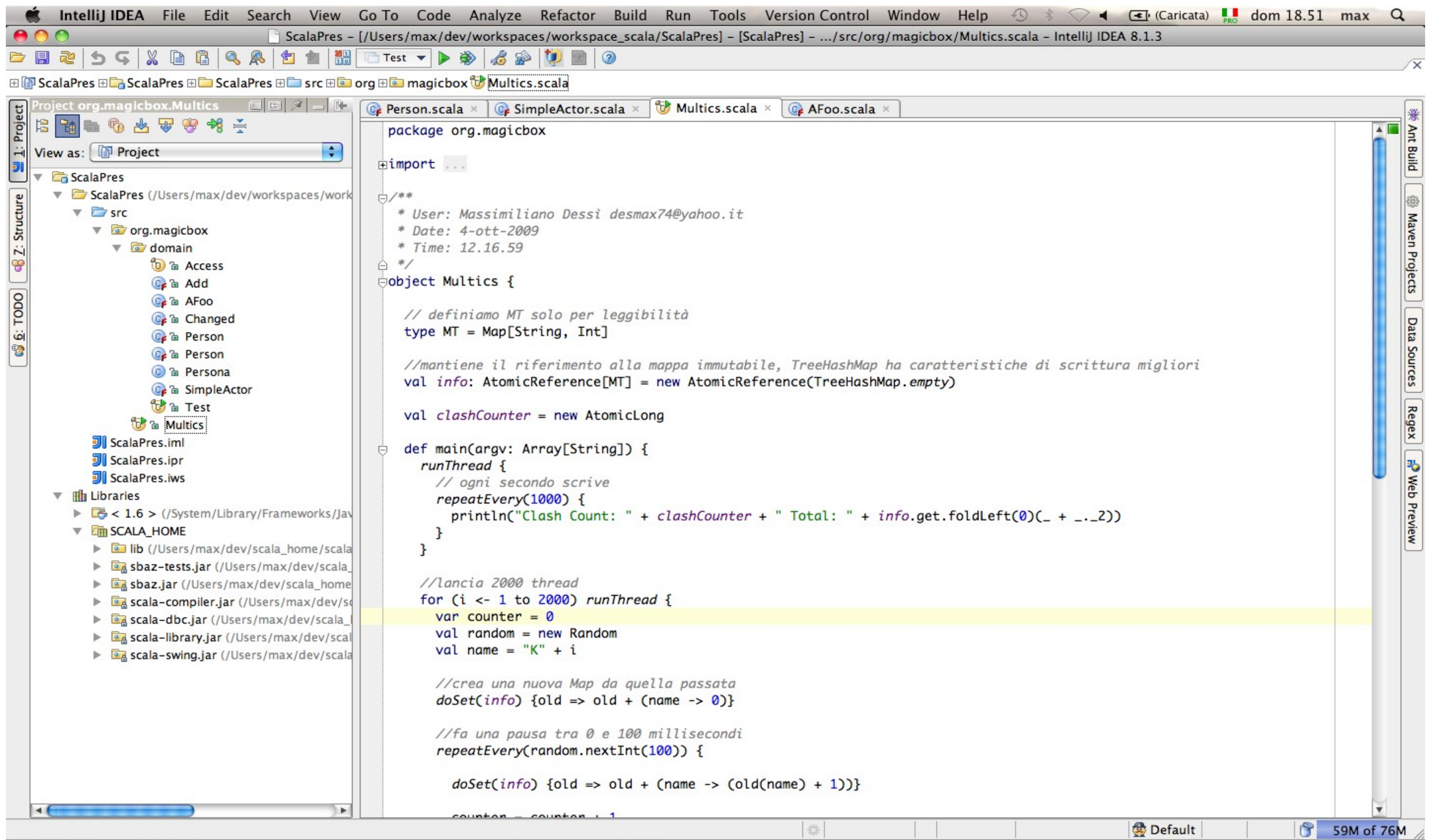
pro-netics group

Actor

```
object Test{
    def main(args: Array[String]) {
        val foo = new FooActor()
        foo.start
        foo ! “mailbox”
        foo ! 1
        foo ! 12
        foo ! 20
        foo ! ”mailbox”
        System.exit(0)
    }
}
```

pro-netics group

Scala - IntelliJ IDEA



pro-netics group

Scala - Netbeans

```scala
/*
 * Main.scala
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package scalaapplication

object Main {

  /**
   * @param args the command line arguments
   */
  def main(args: Array[String]) :Unit = {
    println("Hello, world!")
    val persona = new Person("max","dessi",34)
    println(persona)
  }

}
```

Output - ScalaApplication (run)

```
init:
deps-jar:
Compiling 2 source files to /Users/max/dev/workspaces/workspace_netbeans/ScalaApplication/build/classes
compile:
run:
Hello, world!
Person(max,dessi,34)
BUILD SUCCESSFUL (total time: 5 seconds)
```

pro-netics group

Scala - Eclipse

Eclipse File Edit Navigate Search Project Run Window Help (Caricata) dom 19.06 max

Scala – Foo/src/org/magicbox/Multics.scala – Eclipse SDK – /Users/max/dev/workspaces/workspace_scala

Scala Java

Package Explorer

- Foo
  - src
    - org
    - org.magicbox
      - Multics.scala
    - org.magicbox.domain
      - AFoo.scala
      - Person.scala
      - Persona.java
      - SimpleActor.scala
      - Test.scala
  - Scala Library version 2.7.6.final
    - scala-library.jar – /Users/max/de
    - scala-dbc.jar – /Users/max/dev/
    - scala-swing.jar – /Users/max/de
  - JRE System Library [JVM 1.6.0]
- helloworld
- LinuxDay

AFoo.scala | Test.scala | Person.scala | Multics.scala

```scala
package org.magicbox

import java.util.concurrent.atomic.AtomicReference
import java.util.concurrent.atomic.AtomicLong
import java.util.Random
import scala.collection.immutable.TreeHashMap

/**
 * User: Massimiliano Dessì desmax74@yahoo.it
 * Date: 4-ott-2009
 * Time: 19.16.59
 */
object Multics {

  // definiamo MT solo per leggibilità
  type MT = Map[String, Int]

  //mantiene il riferimento alla mappa immutabile, TreeHashMap ha caratteristiche di scrittura migliori
  val info: AtomicReference[MT] = new AtomicReference(TreeHashMap.empty)

  val clashCounter = new AtomicLong

  def main(argv: Array[String]) {
    runThread {
      // ogni secondo scrive
      repeatEvery(1000) {
        println("Clash Count: " + clashCounter + " Total: " + info.get.foldLeft(0)(_ + _._2))
      }
    }

    //lancia 2000 thread
    for (i <- 1 to 2000) runThread {
      var counter = 0
      val random = new Random
      val name = "K" + i

      //crea una nuova Map da quella passata
      doSet(info) {old => old + (name -> 0)}
```

Outline

Problems | Tasks | Console

No consoles to display at this time.

pro-netics group

Demo concorrenza

Una negozio da barbiere

3 sedie

1 barbiere

n-clienti

pro-netics group

Domande ?

pro-netics
group

Grazie per l'attenzione !

Massimiliano Dessì
desmax74 at yahoo.it
massimiliano.dessi at pronetics.it

http://twitter.com/desmax74

http://jroller.com/desmax

http://www.linkedin.com/in/desmax74

http://www.slideshare.net/desmax74

http://wiki.java.net/bin/view/People/MassimilianoDessi

http://www.jugsardegna.org/vqwiki/jsp/Wiki?MassimilianoDessi

pro-netics group