Análisis de un lenguaje de programación

Alexander Redondo Acuña, Codigo 1703233-7721
Escuela de Ingeniería de Sistemas - Universidad del Valle
Maestria en Ingenieria de sistemas y computación.

RESUMEN

Scala, abreviatura de Scalable Language, es un lenguaje de programación híbrido y funcional. Fue creado por Martin Odersky. Scala integra sin problemas las características de los lenguajes orientados a objetos y funcionales. Scala está compilado para ejecutarse en la máquina virtual Java. Muchas empresas existentes, que dependen de Java para aplicaciones comerciales críticas, recurren a Scala para aumentar su productividad de desarrollo, escalabilidad de aplicaciones y confiabilidad general.

PALABRAS CLAVES: paradigma, declarativa, funcional, concurrencia

INTRODUCCIÓN

Scala está orientado a objetos

Scala es un lenguaje orientado a objetos en el sentido de que cada valor es un objeto. Los tipos y el comportamiento de los objetos se describen por clases y características que se explicarán en los capítulos siguientes.

Las clases se amplían mediante subclases y un mecanismo flexible de composición basado en mixin como un reemplazo limpio para la herencia múltiple.

Scala es funcional

Scala es también un lenguaje funcional en el sentido de que cada función es un valor y cada valor es un objeto, por lo que, en última instancia, cada función es un objeto.

Scala proporciona una sintaxis ligera para definir funciones anónimas, admite funciones de orden superior, permite anidar funciones y admite el currying. Estos conceptos se explicarán más adelante.

Scala está tipificado estáticamente

Scala, a diferencia de algunos de los otros lenguajes de tipo estático (C, Pascal, Rust, etc.), no espera que proporciones información de tipo redundante. No tiene que especificar un tipo en la mayoría de los casos, y ciertamente no tiene que repetirlo. Scala se ejecuta en la JVM

Scala está compilado en Java Byte Code que es ejecutado por Java Virtual Machine (JVM). Esto significa que Scala y Java tienen una plataforma de tiempo de ejecución común. Puede pasar fácilmente de Java a Scala.

El compilador de Scala compila su código Scala en Java Byte Code, que luego puede ejecutarse con el comando 'scala'. El comando 'scala' es similar al comando java, ya que ejecuta el código de Scala compilado.

Scala puede ejecutar código Java

Scala le permite utilizar todas las clases del SDK de Java y también sus propias clases personalizadas.

Scala puede hacer el procesamiento Concurrente y sincronizado

Scala le permite expresar patrones generales de programación de manera efectiva. Reduce el número de líneas y ayuda al programador a codificar de forma segura. Le permite escribir códigos de manera inmutable, lo que facilita la aplicación de simultaneidad y paralelismo.

Cuando consideramos un programa de Scala, se puede definir como una colección de objetos que se comunican mediante la invocación de los métodos de cada uno. Veamos ahora brevemente qué significan clase, objeto, métodos y variables de instancia.

Objeto: los objetos tienen estados y comportamientos. Un objeto es una instancia de una clase. Ejemplo: un perro tiene estados (color, nombre, raza y comportamiento), meneando, ladrando y comiendo.

Clase - Una clase se puede definir como una plantilla / plan que describe los comportamientos / estados que están relacionados con la clase.

Métodos: un método es básicamente un comportamiento. Una clase puede contener muchos métodos. Es en los métodos donde se escriben las lógicas, se manipulan los datos y se ejecutan todas las acciones.

Campos: cada objeto tiene su conjunto único de variables de instancia, que se llaman campos. El estado de un objeto es creado por los valores asignados a estos campos.

Cierre: un cierre es una función cuyo valor de retorno depende del valor de una o más variables declaradas fuera de esta función.

Rasgos: un rasgo encapsula el método y las definiciones de campo, que luego pueden reutilizarse mezclándolos en clases. Los rasgos se utilizan para definir tipos de objetos al especificar la firma de los métodos admitidos.

PARADIGMAS ANALIZADOS

Modelo de programación declarativa.

Comenzamos viendo un ejemplo del lenguaje declarativo, el número más pequeño expresable como la suma de dos cubos de dos maneras diferentes.

```
for {hardyTaxi <- List.range(1000,2000)
        j<-List.range(1,20)
        k<-List.range(1,20)
        l<-List.range(1,20)
        m<-List.range(1,20)
        if ((j*j*j)+(k*k*k) == hardyTaxi &&
(l*l*l)+(m*m*m) ==
        hardyTaxi) && j!=l && k!=m && j!=m}
yield(i,j,k,l,m)

List[(Int, Int, Int, Int, Int)] = List(1729,1,12,9,10)
```

La forma más sencilla de caracterizar la diferencia entre los lenguajes de programación imperativos y los no imperativos es que en el razonamiento sobre la programación imperativa, debe tener en cuenta el "estado" de la máquina como se registra en su memoria.

Definición de Función

Una función en Scala tiene la siguiente forma - Sintaxis

```
def funcionNombre ([lista de parametros]) : [return
tipo] = {
  Cuerpo de la función
  return [expr]
}
```

Aquí, el tipo que retorna podría ser cualquier tipo de datos válidos de Scala y la lista de parámetros será una lista de variables separadas por coma y la lista de parámetros y el tipo que retorna son opcionales. Muy similar a Java, se puede usar una declaración de retorno junto con una expresión en caso de que la función devuelva un valor.

Método de Newton

```
def sqrt(x: Double)={
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
      else sqrtIter(improve(guess, x), x)

def improve(guess: Double, x: Double) = (guess + x
/ guess)/2

def isGoodEnough(guess: Double, x: Double) =
abs(square(guess) - x)<0.001

sqrtIter(1.0, x)}
```

Concurrencia

El modelo de actores que fue propuesto por primera vez por Carl Hewitt en 1973; es un modelo de concurrencia computacional que al igual que los hilos, trata de solucionar el problema de la concurrencia.

En el modelo de actores, cada objeto es un actor. Esta es una entidad que tiene una cola de mensajes o buzón y un comportamiento. Los mensajes pueden ser intercambiados entre los actores y se almacenan en el buzón. Al recibir un mensaje, el comportamiento del actor se ejecuta. El actor puede: enviar una serie de mensajes a otros actores, crear una serie de actores y asumir un nuevo comportamiento para el próximo mensaje.
La importancia en este modelo es que todas las comunicaciones se llevan a cabo de forma asincrónica. Esto implica que el remitente no espera a que un mensaje sea recibido en el momento que lo

envío, solo sigue su ejecución.

Una segunda característica importante es que todas las comunicaciones se producen por medio de mensajes: no hay un estado compartido entre los actores. Si un actor desea obtener información sobre el estado interno de otro actor, se tendrá que utilizar mensajes para solicitar esta información. Esto permite a los actores controlar el acceso a su estado, evitando problemas.

Scala implementa el modelo de actores, veamos un poco de código:

```
import scala.actors.Actor

class Saludador extends Actor  {

  def act() = {
       while (true) {
          receive {
             case "conocido" =>
                println("Hola!")
             case _ =>
                println("...")
          }
       }
    }

}

object ActorsTest extends App {

  val saludador = new Saludador
  saludador.start()

  saludador ! "otro"
  saludador ! "conocido"
}
```

el modelo de actores, simplifica el desarrollo concurrente.

Estado

Los estados, transiciones y elementos auxiliares son entidades. De hecho, no son más que objetos, clases, métodos y traits Scala.

Estado Inicial

Los estados son objetos. O bien son instancias de clases u objetos singleton. Por otra parte, también hemos vista que la forma correcta de implementar máquinas de estado es hacer que estos estados, en definitiva objetos, sean inmutables. De esa manera, son las transiciones las responsables de generar nuevos estados totalmente nuevos.

El estado inicial, al diferencia del resto de estados, no se genera por medio de una transición. Es el estado actual al iniciar la interacción con el DSL. Esto indica su naturaleza como singleton, la cual queda confirmada definitivamente por el hecho de que no puede existir ninguna otra instancia de él.

```
object lines extends ToCommandWithSeparator
```

Desde el punto de vista del usuario de Scala, este estado inicial debería ser una palabra indicando el comienzo de una frase en el DSL. Esta es otra razón para justificar el que sea implementado como un objeto singleton.

El estado inicial necesita transitar a otro, es por ello por lo que el singleton *lines* extiende el trait *ToCommandWithSeparator. Para* no adelantar acontecimientos basta que, por el momento, tengamos en cuenta que *ToCommandWithSeparator* es un trait que contiene un conjunto de transiciones.

Estados transitorios y finales

Hay diferentes tipos de estados. Además, muchos de estos estados son bastante parecidos entre sí y podrían construirse a partir de una plantilla común. como implementarlos
.
La clasificación de más alto nivel para los estados no iniciales debería ser la siguiente:

Transitorios y finales.

Conceptualmente, los estados del primer grupo no pueden usarse para generar resultados en tanto que los del segundo sí. Obviamente, esta limitación ocurre igualmente en la implementación de los estados y ello implica que los estados transitorios no pueden generar programas AWK mientras que los finales si.

En cada estado estos atributos pueden estar vacíos o no y, cuando se le pide a un estado final que genere un programa AWK, dichos atributos se utilizarán para generar el resultado en formato de cadena de texto.

```
abstract class AwkCommand protected(
  private[scalawk] val commandOptions:
Seq[String] = Seq.empty,
  private[scalawk] val linePresentation:
Seq[AwkExpression] = Seq.empty,
  private[scalawk] val lineProgram:
Seq[SideEffectStatement] = Seq.empty,
  private[scalawk] val initialProgram:
Seq[SideEffectStatement] = Seq.empty,
  private[scalawk] val endProgram:
Seq[SideEffectStatement] = Seq.empty
) {
  def this(prev: AwkCommand) = this(
    prev.commandOptions,
    prev.linePresentation,
    prev.lineProgram,
    prev.initialProgram,
    prev.endProgram
  )
}
```

Actores y paso de mensajes

Scala proporciona una alternativa basada en un modelo de paso de mensajes, más fácil de programar ya que no hay datos compartidos (y por tanto no tenemos el problema de los interbloqueos ni de las condiciones de carrera).

Un actor es una entidad similar a un thread que dispone de un buzón para recibir mensajes.

Para implementar un actor, se debe extender la clase scala.actors.Actor e implementar el método act.

Ejemplo básico de un actor

```
import scala.actors._

class SillyActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("Estoy actuando!")
      Thread.sleep(1000)
    }
  }
}
```

Activamos un actor invocando al método start proporcionado por la clase Actor
Se activa un hilo (thread) que se ejecuta en paralelo al shell

```
scala> val actor = new SillyActor
scala> actor.start()
scala> Estoy actuando!
Estoy actuando!
Estoy actuando!
Estoy actuando!
Estoy actuando!
```

La ejecución de los actores es independiente de unos a otros también. Ejemplo:

```
import scala.actors._

class SeriousActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("Ser o no ser.")
      Thread.sleep(1000)
    }
  }
}
```

-
-
- Ejecutamos actores de ambas clases al mismo tiempo:

```
var a = new SillyActor
var b = new SeriousActor

a.start; b.start
scala> Estoy actuando!
Ser o no ser.
Estoy actuando!
Ser o no ser.
Estoy actuando!
Ser o no ser.
Estoy actuando!
Ser o no ser.
Estoy actuando!
Ser o no ser.
```

Comunicación entre actores: mensajes

Ya hemos creado actores que se ejecutan independientemente unos de otros.

¿Cómo pueden trabajar juntos? ¿Cómo se pueden comunicar sin compartir memoria y sin bloqueos?

Los actores se comunican enviándose mensajes.

Se puede enviar un mensaje usando el método !

Para recibir los mensajes los actores deben llamar a la función parcial receive (implementa algo similar a un match)

La llamada a receive deja el actor a la espera del siguiente mensaje que llega a su buzón:

```
import scala.actors.Actor._
val echoActor = actor {
  while(true) {
    receive {
      case msg => println("Mensaje recibido: "+ msg)
    }
  }
}
echoActor ! "hola"
```

Los mensajes son asíncronos

Cuando un actor envía un mensaje no se queda esperando el resultado, sino que continúa su proceso
Esto se denomina comunicación asíncrona
Una invocación a una función o a un método es síncrona: el flujo de ejecución del objeto llamador se detiene hasta que la función devuelve un resultado
Los mensajes se almacenan en una cola en el actor receptor El receptor no interrumpe la ejecución de su proceso; sino que lee los mensajes cuando ejecuta un receive

Mensajes con tipo

Un actor puede esperar mensajes de un tipo en concreto, especificándolo en el tipo del case.
Por ejemplo, un actor que sólo espera mensajes de tipo Int:

```
import scala.actors.Actor._

val intActor = actor {
  receive {
     case x: Int => println("Tengo un Int: "+ x)
  }
}
```

Si recibe mensajes de otro tipo los ignora

```
scala> intActor ! "hola"
scala> intActor ! scala.math.Pi
scala> intActor ! 12
scala> Tengo un Int: 12
```

POO en Scala

Definición de clases

Nombre de la clase
Campos (variables de instancia) privados que se deben inicializar
Métodos, por defecto públicos

Veamos un primer ejemplo, en el que definimos una clase Contador que tiene un campo valor y dos métodos sin argumentos:

```
class Contador {
private var valor = 0
def incrementa() { valor += 1 }
def actual() = valor
}
```

Para usarlo:

```
val c = new Contador
c.incrementa()
c.actual() ⇒ 1
c.actual ⇒ 1. Se puede llamar al
método sin los paréntesis
```

Creamos un nuevo contador, incrementamos su valor y devolvemos su valor actual.
Una diferencia muy importante entre Scala y Java es que Scala permite invocar a los métodos sin argumentos sin poner los paréntesis. Es importante darse cuenta de que la segunda llamada -c.actual- no está accediendo a un campo, sino ejecutando un método. Es idéntica a la invocación anterior.
También sería posible declarar un método que no tiene argumentos sin escribir los paréntesis. Por ejemplo, el método -actual- lo podríamos escribir de la siguiente forma:

```
class Contador {
private var valor = 0
def incrementa() { valor += 1 }
def actual = valor // Sin () en la
declaración
}
```

CONCLUSIÓN

Para terminar este articulo nos queda más que comentar que la experiencia de trabajar con Scala por lo menos en dos ejemplos fue bastante agradable. La elaboración de este artículo nos sirvió.

mucho para aprender lo básico de un nuevo lenguaje de una manera didáctica y entretenida.