Scala學習筆記：重要語法特性

2017-09-28 / VIEWS: 5

1.變量聲明

Scala 有兩種變量， val 和 var val的值聲明後不可變，var可變

```
val msg: String = "Hello yet again, world!"
```

或者類型推斷

```
val msg = "Hello, world!"
```

2.函數定義

如果函數僅由一個句子組成，你可以可選地不寫大括號。

```
def max2(x: Int, y: Int) = if (x > y) x else y
```

3.for循環

打印每一個命令行參數的方法是：

```
args.foreach(arg => println(arg))
```

如果函數文本由帶一個參數的一句話組成，

```
args.foreach(println)
```

Scala 裏只有一個指令式 for的函數式近似。

```
for (arg <- args)
  println(arg)
```

<- 的左邊是變量，右邊是數組。

再比如帶類型的參數化數組

```
val greetStrings = new Array[String](3)
  greetStrings(0) = "Hello"
  greetStrings(1) = ", "
  greetStrings(2) = "world!\n"
  for (i <- 0 to 2)
    print(greetStrings(i))
```

注意這裏的數組定義，只要new的時候帶類型Array[String]就行了，val後面自動推斷類型。
注意這裏的數組索引用的是()而不是java裏面的[]。
因為scala裏面根本沒有傳統意義上的操作符，取而代之的是他們都可以轉換為方法。例如greetStrings(i)可以轉換成 greetStrings.apply(i)，
greetStrings(0) = "Hello" 將被轉化為 greetStrings.update(0, "Hello")

儘管實例化之後你無法改變 Array 的長度，它的元素值卻是可變的。因此，Array 是可變的對象。

4.List對象

創建一個 List 很簡單。 List裏面元素不可變。

```
val oneTwoThree = List(1, 2, 3)
```

List有個叫“ :::”的方法實現疊加功能。

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour

//結果是List(1, 2, 3, 4)
```

Cons 把一個新元素組合到已有 List的最前端，然後返回結果 List。 例如，若執行這個腳本：

```
val twoThree = list(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
//你會看到： List(1, 2, 3)
```

一個簡單的需記住的規則：如果一個方法被用作操作符標註，如 a* b，那么方法被左操作數調用，就像 a.*(b)——除非方法名以冒號結尾。這種情況下，方法被右操作數調用。因此， 1 :: twoThree 裏， ::方法被 twoThree 調用，傳入 1，像這樣： twoThree.::(1)。
類 List 沒有提供 append 操作，因為隨着列表變長 append 的耗時將呈線性增長，而使用::做前綴則僅花費常量時間。如果你想通過添加元素來構造列表，你的選擇是把它們前綴進去，當你完成之後再調用 reverse；

5.元組

與列表一樣，元組也是不可變的，但與列表不同，元組可以包含不同類型的元素。

```
val pair = (99, "Luftballons", 55)
    println(pair._1)
    println(pair._2)
    println(pair._3)
```

注意這裏第一個元素是從_1開始而不像List那樣從0開始。

6.Set和Map

```
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
```

默認set或者HashSet可變。即jetSet = jetSet + "Lear"

如果要用不可變的set或者HashSet，要import

```
import scala.collection.immutable.HashSet
val hashSet = HashSet("Tomatoes", "Chilies")
println(hashSet + "Coriander")

//這裏就不能再賦值給hashSet了
```

7.訪問級別

Public 是 Scala 的缺省訪問級別。C++中struct默認是public，class默認是private。java類中的變量默認是default類型，只允許在同一個包內訪問，一般用的時候跟private差不多。所以java用的時候要想用public必須要指出。

```
class ChecksumAccumulator {
private var sum = 0
...

}
```

8.靜態對象：object

Scala 比 Java 更面向對象的一個方面是 Scala 沒有靜態成員。替代品是， Scala 有單例對象：singleton object。除了用 object 關鍵字替換了 class 關鍵字以外，單例對象的定義看上去就像是類定義。

```
import scala.collection.mutable.Map
object ChecksumAccumulator {
private val cache = Map[String, Int]()
def calculate(s: String): Int =
```

```
if (cache.contains(s))
cache(s)
else {
.....
}
```

可以如下方式調用 ChecksumAccumulator單例對象的calculate方法：

```
ChecksumAccumulator.calculate("Every value is an object.")
```

也不用在實例化了。
類和單例對象間的一個差別是，單例對象不帶參數，而類可以。因為你不能用new關鍵字實例化一個單例對象， 你沒機會傳遞給它參數。

9.伴生對象

當單例對象與某個類共享同一個名稱時，他被稱作是這個類的伴生對象： companion object。你必須在同一個源文檔裏定義類和它的伴生對象。類被稱為是這個單例對象的伴生類： companion class。類和它的伴生對象可以互相訪問其私有成員。
不與伴生類共享名稱的單例對象被稱為孤立對象： standalone object。 由於很多種原因你會用到它

10.Main函數

要執行Scala進程，你一定要提供一個有main方法（僅帶一個參數， Array[String]，且結果類型為 Unit的孤立單例對象名。比如下面這個例子；

```
import ChecksumAccumulator.calculate
object Summer {
    def main(args: Array[String]) {    //對比java裏面的public static void

   main(String[] args){}
       for (arg <- args)
       println(arg + ": " + calculate(arg))
    }
}
```

11.另一種Main函數

Application 特質：在單例對象名後面寫上“ extends Application” 。然後取代main 方法.

```
import ChecksumAccumulator.calculate
object FallWinterSpringSummer extends Application {
  for (season <- List("fall", "winter", "spring"))
  println(season +": "+ calculate(season))
}
```

效果和main函數一樣。不過它也有些缺點。首先， 如果想訪問命令行參數的話就不能用它，因為args數組不可訪問。第二，如果你的進程是多線程的就需要顯式的 main 方法。最後，某些JVM的實現沒有優化被 Application 特質執行的對象的初始化代碼。因此只有當你的進程相對簡單和單線程情況下你才可以繼承 Application 特質。

12.帶參數類聲明

Java 類具有可以帶參數的構造器，而 Scala 類可以直接帶參數。 Scala 的寫法更簡潔——類參數可以直接在類的主體中使用；沒必要定義字段然後寫賦值函數把構造器的參數複製到字段裏。(不需要構造函數)
例如以下分數構造器：

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  override def toString = n +"/"+ d

 val numer: Int = n
  val denom: Int = d

  def add(that: Rational): Rational =
  new Rational(
  numer * that.denom + that.numer * denom,
  denom * that.denom
}
```

require方法帶一個布爾型參數。如果傳入的值為真，require將正常返回。反之，require將通過拋出IllegalArgumentException來阻止對象被構造。 這裏使用了重載。重載了類自帶的toString函數。 這裏that也就是隨便起了個名字的變量，不是關鍵字。this是關鍵字。比如下面這個函數：

```
def lessThan(that: Rational) =
this.numer * that.denom < that.numer * this.denom
```

這裏的this可以省略。但下面這個就不能省略了：

```
def max(that: Rational) =
  if (this.lessThan(that)) that else this
```

13.if返回值

Scala 的 if 可以產生值（是能返回值的表達式）。於是 Scala 持續了這種趨勢讓 for， try 和 match 也產生值。while不產生值，所以用得少。如果實在想用while，在純函數式編程的時候可以考慮遞歸。
指令式風格：

```
var filename = "default.txt"
if (!args.isEmpty)
filename = args(0)
```

函數式風格：

```
val filename =
if (!args.isEmpty) args(0)
else "default.txt"
```

使用 val 而不是 var 的第二點好處是他能更好地支持等效推論。無論何時都可以用表達式替代變量名。如要替代 println(filename)，你可以這么寫：

```
println(if (!args.isEmpty) args(0) else "default.txt")
```

14.break 和 continue

scala裏面也沒有break以及continue。如果想要用到他們的功能，可以使用增加布爾變量控制到循環語句判斷中，類似：

```
var foundIt = false  while (i < args.length && !foundIt) { }
```

15.for過濾器

在for裏面還支持過濾器：

```
val filesHere = (new java.io.File(".")).listFiles    //路徑名的目錄中的文檔的數組。
for (file <- filesHere if file.getName.endsWith(".scala"))
println(file)
```

甚至多個過濾器：

```
for (
  file <- filesHere
  if file.isFile;
  if file.getName.endsWith(".scala")
) println(file)
```

16.for循環生成新集合

for {子句} yield {循環體} 製造新集合
例如:

```
def scalaFiles =
for {
file <- filesHere
```

```
if file.getName.endsWith(".scala")
} yield file
```

這樣每一步就不是打印一個file，而是將file存儲起來，最終產生一個Array[File]

17.try catch finally

```
try {
    val f = new FileReader("input.txt")
    openFile(file)

} catch {
   case ex: FileNotFoundException =>  new FileReader("input.txt")
   //注意這裏依然有返回值。使用=>符號
   case ex: IOException => // Handle other I/O error
}
finally {
   file.close() // 確保關閉文檔。
}
```

這裏catch {
  case ex: 。。。
  case ex: 。。。
}
對比java，catch是這樣用的
catch (Exception e) { 。。。。} 而且經常在catch裏面throw。scala一般不使用throw。

還有，finally裏最好只做一些關閉或打印之類的操作，不要有副作用的表達式，這樣會有無謂的返值。

18.switch語句

match語句就像java裏的switch語句。

```
val firstArg = if (args.length > 0) args(0) else ""
firstArg match {
case "salt" => println("pepper")
case "chips" => println("salsa")
case "eggs" => println("bacon")
case _ => println("huh?")
}
```

差別：
1、Java 的 case 語句裏面的整數類型和枚舉常量。而這裏可以是任意類型，甚至正則匹配
2、在每個可選項的最後並沒有 break。取而代之， break是隱含的。
3、match 表達式也能產生值：例如可以這樣：

```
val friend =
firstArg match {
```

```
  case "salt" => "pepper"
  case "chips" => "salsa"
  case _ => "huh?"
}
println(friend)
```

19.函數嵌套定義

函數：函數式編程風格的一個重要設計原則：進程應該被解構成若干小的函數， 每個完成一個定義良好的任務。在java中通常這樣做：

```
def processFile(filename: String, width: Int) {
  ...
  for (line <- source.getLines)
   processLine(filename, width, line)
}
private def processLine(filename:String, width:Int, line:String) {
  ....
}
```

Scala 提供了另一種方式：你可以把函數定義在另一個函數中。就好象本地變量那樣，這種本地函數僅在包含它的代碼塊中可見。

```
def processFile(filename: String, width: Int) {
def processLine(filename:String, width:Int, line:String) {
  ...
}
val source = Source.fromFile(filename)
for (line <- source.getLines) {
processLine(filename, width, line)
}
}
```

20.Lambda函數

Scala 擁有第一類函數： first-class function。你不僅可以定義函數和調用它們，還可以把函數寫成沒有名字的文本： (跟python裏面的lambda函數差不多)

```
scala> var increase = (x: Int) => x + 1
scala> increase(10)
res0: Int = 11
```

如果你想在函數文本中包括超過一個語句，用大括號包住函數體，當函數值被調用時，所有的語句將被執行，而函數的返回值就是最後一行產生的那個表達式。

```
scala> increase = (x: Int) => {
  println("We")
  println("are")
  println("here!")
  x + 1
}
```

21.集合通配符：_

Scala 提供了許多方法去除冗餘信息並把函數文本寫得更簡短。比如去除參數類型以及被推斷的參數之外的括號：

```
scala> someNumbers.filter(x => x > 0)
```

如果想讓函數文本更簡潔，可以把下劃線當做一個或更多參數的佔位符，只要每個參數在函數文本內僅出現一次。 ：

```
scala> someNumbers.filter(_ > 0)
```

還可以使用一個下劃線替換整個參數列表。叫偏應用函數

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
```

一般調用可以這樣：

```
scala> sum(1, 2, 3)
```

用偏函數取而代之：

```
scala> val a = sum _  //請記住要在函數名和下劃線之間留一個空格
scala> a(1, 2, 3)
```

再比如

```
someNumbers.foreach(println _)
```

22.閉包

閉包：是指可以包含自由（未綁定到特定對象）變量的代碼塊
任何帶有自由變量的函數文本，如(x: Int) => x + more，都是開放術語：由於函數值是關閉這個開放術語(x: Int) => x + more 的行動的最終產物， 得到的函數值將包含一個指向捕獲的 more 變量的參考， 因此被稱為閉包。

```
scala> var more = 1
scala> val addMore = (x: Int) => x + more
scala> addMore(10)
res19: Int = 11
```

每次函數被調用時都會創建一個新閉包。每個閉包都會訪問閉包創建時活躍的 more 變量。

23.重複參數

想要標註一個重複參數，在參數的類型之後放一個星號。例如：

```
scala> def echo(args: String*) =
        for (arg <- args) println(arg)
```

這樣定義， echo 可以被零個至多個 String 參數調用：

```
scala> echo()
scala> echo("one")
scala> echo("hello", "world!")
```

但是如果有一個數組變量
scala> val arr = Array("What's", "up", "doc?")，則不能像scala> echo(arr)這樣調用。
你需要在數組參數後添加一個冒號和一個_*符號，像這樣：

```
scala> echo(arr: _*)
```

24.curry化

Scala 允許你創建新的“感覺像是原生語言支持”的控制抽象：scala提供curry 化：
Scala裏的Curry化可以把函數從接收多個參數轉換成多個參數列表。如果要用同樣的一組實參多次調用一個函數，可以用curry化來減少噪音，讓代碼更有味道。我們要編寫的方法不是接收一個參數列表，裏面有多個參數，而是有多個參數列表，每個裏面可以有一個或多個參數。也就是說，寫的不是def foo(a: Int, b: Int, c: Int){}，而是 def foo(a: Int)(b: Int)(c: Int){}。可以這樣調用這個方法，比如：foo(1)(2)(3)、foo(1){2}{3}，甚至這樣foo{1}{2}{3}。

例如，傳統函數如下：

```
scala> def plainOldSum(x: Int, y: Int) = x + y

scala> plainOldSum(1, 2)     //res5: Int = 3

scala允許你使用curry化的新型函數：

scala> def curriedSum(x: Int)(y: Int) = x + y
scala> curriedSum(1)(2)            //res5: Int = 3
```

結果一樣。

25.帶函數參數的函數

高階函數： higher-order function——帶其它函數做參數的函數

```
def filesMatching(query: String,
  matcher: (String, String) => Boolean) = {
  for (file <- filesHere; if matcher(file.getName, query))
    yield file
}
```

這裏matcher其實是一個函數，這裏做了filesMatching函數的參數。
(String, String) => Boolean)表示matcher函數的參數是(String, String)類型，而返回值是Boolean 類型
你可以通過讓多個搜索方法調用它，並傳入合適的函數：

```
def filesEnding(query: String) =
  filesMatching(query, _.endsWith(_))
```

這就相當於

```
def filesEnding(query: String) =
  for (file <- filesHere; if file.getName.endsWith(query))
    yield file
```

因為上面matcher: (String, String)裏面兩個參數， matcher(file.getName, query)
所以 _.endsWith(_)裏面的第一個_對應於字符串file.getName，第二個_對應於字符串query，
連在一起就是file.getName.endsWith(query)

類似的

```
def filesContaining(query: String) =
  filesMatching(query, _.contains(_))
```

就相當於

```
def filesContaining(query: String) =
   for (file <- filesHere; if file.getName.contains(query))
     yield file
```

26.傳名參數

Scala中允許無參數的函數作為另一函數的參數傳遞進去,也就是傳名參數(call-by-name)
定義一個函數 myAssert ，而其參數則為傳名參數。在這裏，我們想要實現的是斷言，可以將傳名參數寫成函數文本的格式：(…) => Type ，即參數列表 => 類型。

```
def myAssert(check: () => Boolean) =
    if(!check()){
        println("OK ...")
        throw new AssertionError
    }
```

上面的函數定義了一個當客户代碼傳入的函數值（這裏我們用（ ）指明，代表省略了該函數的參數列表。調用方式如下：

```
scala> myAssert(() => 5 < 3)
```

客户端代碼中的 () => 5 < 3 似乎有點繁瑣，如果能夠直接傳入 5 < 3 之類的布爾表達式就更好了。這是可以實現的。只需要將函數定義 def 中空參數列表即小括號對（ ）去掉，直接用 => 而不是（ ）=> 就可以了。此外，if 判斷中的 check 後面的（ ）也要同時去掉。修改後的代碼如下：

```
def myAssert(check: => Boolean) =
    if(!check){
        println("OK ...")
        throw new AssertionError
    }
myAssert(5 < 3)
```