Scala 学习笔记

harli

2015/1/19

# Scala阅读笔记

1. Scala创始人Martin Odersky
2. Scala的设计始终贯穿着一个理念：创造一种更好地支持组件的语言。
3. 特征或语法与Java对比：
4. 抽象类 - 和Java一样
5. Case类 - 意味着你可以在匹配表达式中使用它
6. 密封类 - 意味着你只能使用这里定义的子类，只需要关注当前文件的case类匹配的列举
7. 特征（traits） - 和Java中的接口相似
8. 对象 - 和Java中的static相似

（Scala 2.8）另一方面，Scala抛弃了枚举和注解，它们和Java中的类很相似，但并不完全相似，如果看一看相关的语言规范，我觉得Java在这个领域是更复杂的。

# Scala语法学习

Liskov替代原则：Scala协变和逆变规则的理论基础

它规定T类型是U类型的子类条件是，在U对象出现的所有地方都可以用T对象来代替。同时对于U和T中相同的方法定义，还必须保证T的参数类型需求的比较少，而T的返回类型提供得比较多。

## 视界（“类型类”）

<%

有时候，你并不需要指定一个类型是等/子/超于另一个类，你可以通过转换这个类来伪装这种关联关系。

一个视界指定一个类型可以被“看作是”另一个类型。这对对象的只读操作是很有用的。

隐 函数允许类型自动转换。更确切地说，在隐式函数可以帮助满足类型推断时，它们允许按需的函数应用。

视界，就像类型边界，要求对给定的类型存在这样一个函数。您可以使用<%指定类型限制，例如：

scala> class Container[A <% Int] { def addIt(x: A) = 123 + x }

defined class Container

这是说 A 必须“可被视”为 Int

## 类型的上界和下界

1. U >: T

这是类型下界的定义，也就是U必须是类型T的父类(或本身，自己也可以认为是自己的父类)。

1. S <: T

这是类型上界的定义，也就是S必须是类型T的子类（或本身，自己也可以认为是自己的子类)。

# Scala

http://dirlt.com/scala.html#sec-1-4-2

## Function

所有函数都是继承于Function这个类的，比如(Int,Int) => Long那么继承于Function2[Int,Int,Long]并且实现了apply方法，所以如果想创建函数的话，实际上也可以使用创建对象方式来完成

val x = new Function2[Int,Int,Long] {

def apply(x:Int,y:Int) = x + y

}

note(dirlt)@2013-11-04: much shorter code. Function2[Int,Int,Long] = (Int,Int)=>Long

object x extends((Int,Int)=>Long) {

def apply(x:Int,y:Int) = x + y

}

**Harli**：使用type别名来代替继承

## 控制结构

从函数式编程角度出发，所有的控制结构都应该是能够产生值的。对于scala来说也是，基本上所有的控制结构都能够产生值。

* while # 不产生值
* foreach # args.foreach(<function>).
* for # for(arg <- args). 注意这里的arg以val定义所以不能够修改
* if/else # val x = if (<pred>) <value1> else <value2>
* match/case # pattern matching.
* try/catch/finally # exception handling.
* **note(dirlt):没有break/continue语句**

## operator

* a op b -> a.op(b)
* a(b) -> a.apply(b)
* a(b)=c -> a.update(b,c)
* a op: b -> b.op(a) # If the method name ends in a colon, the method is invoked on the right operand.
  + **note(dirlt):但是evaluation的顺序依然先是a，然后是b**

## import

<http://www.scala-lang.org/old/node/119.html>

| **The clause** | **makes available without qualification..** |
| --- | --- |
| import p.\_ | all members of p (this is analogous to import p.\* in Java). |
| import p.x | the member x of p. |
| import p.{x => a} | the member x of p renamed as a. |
| import p.{x => \_} | the member x of p removed. |
| import p.{x, y} | the members x and y of p. |
| import p1.p2.z | the member z of p2, itself member of p1. |

uthermore the clause import p1.\_, p2.\_ is a shorthand for import p1.\_; import p2.\_.

scala import相比java import更加灵活

* may appear anywhere // 类似Python的import.
* may refer to objects (singleton or regular) in addition to packages
* let you rename and hide some of the imported members

## 模式匹配

pattern总结起来大约以下几类：

1. Wildcard patterns // \_ 统配
2. Constant patterns // 常量
3. Variable patterns // 变量
4. Constructor patterns // 构造函数
5. Sequence patterns // 比如List(\_,\_). 如果需要匹配剩余的话使用List(0,\_\*)
6. Tuple patterns // (a,b,c)
7. Typed patterns // 使用类型匹配 case a:Map[\_,\_]
   * asInstanceOf[<type>]
   * isInstanceOf[<type>]
   * **note(dirlt):这里需要注意容器类型擦除.Array例外因为这个是java内置类型**

pattern matching过程中还有下面几个问题需要注意：

* Patterns are tried in the order in which they are written.
* Variable binding // 有时候我们希望匹配的变量包含外层结构
  + A(1,B(x)) => handle(B(x))
  + A(1, p @ B(\_)) => handle(p) # p绑定了B(x)这个匹配
  + A(1, p @ B()) => handle(p) # **B是可以包含unapply从type(p) => Boolean的类，做条件判断**
* Pattern guards // 有时候我们希望对pattern做一些限制性条件
  + A(1,e,e) 比如希望后面两个元素相等，但是这个在pm里面没有办法表达
  + A(1,x,y) if x == y => <body> // 通过guard来完成

## extractor

模式匹配除了能够直接作用在case class上之外，也可以作用在普通的class上面，但是需要普通的class提供一些辅助的方法将转换成为case class或者是constant/string上面。这个机制在scala里面称为 **extractor**

下面是一个例子

class A(val a:String,

val b:String) {

}

val a = new A("hello","world");

a match {

case A(x,y) => println(x + "," + y);

case \_ => println("!match");

}

这段代码不能够运行，原因在于没有办法告诉scala，如果将A实例和A(x,y)来做匹配。对于case classes来说实现可能相对简单，因为case class的class parameters都是val定义的，也就是说构造参数没有办法改变，编译器内部处理case classes的话可以保存这个构造参数，而general class却不能够像case class一样。所以需要用户提供辅助函数来帮助scala做pattern matching. **用户需要在companion object提供unapply函数**

object A {

def apply(a:String,b:String) = new A(a,b)

def unapply(x:A) = Some((x.a,x.b))

}

unapply和apply通常是配对的函数。apply将参数构造成为一个对象，而unapply将对象解构成为参数。the apply method is called an injection, because it takes some arguments and yields an element of a given set. The unapply method is called an extrac- tion, because it takes an element of the same set and extracts some of its parts. 而companion object则称为extractor.

**unapply的过程可以认为是将unapply参数最用在expression上，抽取出这个expression的构造参数** 比如上面过程可以认为类似

object A {

def apply(a:String,b:String) = new A(a,b)

def unapply(a: A) = Some((a.a,a.b))

}

val a = new A("hello","world");

A.unapply(a) match {

case Some((x,y)) => println(x + "," + y);

case \_ => println("!match");

}

使用上面的unapply方法不能够匹配带有\_\*这种sequence variable的pattern.允许匹配这种pattern的话，那么需要实现unapplySeq方法，返回参数必须是Option[Seq[T]]这个类型

object A {

def apply(a:String,b:String) = new A(a,b)

def unapplySeq(a: A):Option[Seq[String]] = Some(List(a.a,a.b))

}

val a = new A("hello","world");

a match {

case A(x,\_\*) => println(x);

case \_ => println("!match");

}

## annotation

## Implicit conversions

关于隐式转换有下面几个通用规则 Implicit conversions are governed by the following general rules:

1. Marking Rule: Only definitions marked implicit are available. 必须显示指明implicit.
2. Scope Rule: An inserted implicit conversion must be in scope as a single identifier, or be associated with the source or target type of the conver- sion. 隐式转换函数必须能够以单个id来访问，或者是在转换类型共生对象内部有定义
3. Non-Ambiguity Rule: An implicit conversion is only inserted if there is no other possible conversion to insert. 无歧义否则编译出现如下错误“implicit conversions are not applicable because they are ambiguous”
4. One-at-a-time Rule: Only one implicit is tried. 只会尝试做一次隐式转换
5. Explicits-First Rule: Whenever code type checks as it is written, no implicits are attempted. 如果类型匹配就不会做隐式转换

## 字段初始化顺序问题

pre-initialized field.

|  |
| --- |
| trait A{  val a: Int;  val b: Int;  println("A..." + a + "," + b);  }  val b = new {  val a = 1;  val b = 2;  } with A; |

惰性初始化- lazy-evaluation

|  |
| --- |
| trait A{  val a: Int;  val b: Int;  lazy val c = a;  lazy val d = b;  def say() {  println(c + "," + d);  }  }  val b = new A{  val a = 1;  val b = 2;  }  b.say |

但是lazy不允许只有声明必须有定义，这个定义对应expression表示这个lazy value计算方式。

## 修饰符

A modifier of the form private[X] or protected[X] means that access is private or protected “up to” X, where X designates some enclosing package, class or singleton object. 可以用来修饰class, field, method. **其含义是private/protected属性最多作用到X以外，X以内均可以作为public来进行访问。** 其中X还有一个特例就是this，那么标明这个字段只能够在这个实例里面访问。下面是一个例子。

## Enumeration

创建枚举类型非常简单.对于枚举类型来说通常都是单例所以直接使用object较多。You can find more information in the Scaladoc comments of class scala.Enumeration.

object X extends Enumeration {

val A,B,C = Value; // print as A,B,C

val E = Value("hello");

val F = Value("???"); // print as ???

}

Value这里是一个比较特殊的类型path-dependent type.所谓path-dependent type是指这个类型随着路径不同而不同。在X里面，那么Value type全称是X.Value, 这样就可以和其他枚举类型的Value区分开来。

scala提供的枚举类型也相对比较灵活，也可以很容易地访问整个枚举类型内部，也可以很容易地构造出枚举类型