Scala入门

本文源自Michel Schinz和Philipp Haller所写的A Scala Tutorial for Java programmers，由Bearice成中文，dongfengyee(东风雨)整理。

1 简介

本文仅在对 Scala 语言和其编译器进行简要介绍。本文的目的读者是那些已经具有一定编程经验，而想尝试一下 Scala 语言的人们。要阅读本文，你应当具有基础的面向对象编程的概念，尤其是 Java 语言的。

2 第一个Scala例子

作为学习 Scala 的第一步，我们将首先写一个标准的 HelloWorld，这个虽然不是很有趣，但是它可以让你对 Scala 有一个最直观的认识而不需要太多关于这个语言的知识。我们的 Hello world 看起来像这样：

```
object HelloWorld {
      def main(args: Array[String]) {
              println("Hello, world!")
      }
}
```

程序的结构对 Java 程序员来说可能很令人怀念：它由一个 main 函数来接受命令行参数，也就是一个 String 数组。这个函数的唯一一行代码把我 们的问候语传递给了一个叫 println 的预定义函数。main 函数不返回值（所以它是一个 procedure method）。所以，也不需要声明返回类型。

对于 Java 程序员比较陌生的是包含了 main 函数的 object 语句。这样的语句定义了一个单例对象：一个有且仅有一个实例的类。object 语 句在定义了一个叫 HelloWorld 的类的同时还定义了一个叫 HelloWorld 的实例。这个实例在第一次使用的时候会进行实例化。

聪明的读者可能会发现 main 函数并没有使用 static 修饰符，这是由于静态成员（方法或者变量）在 Scala 中并不存在。Scala 从不定义静态成员，而通过定义单例 object 取而代之。

2.1 编译实例

我们使用Scala编译器“scalac”来编译Scala代码。和大多数编译器一样，scalac 接受源文件名和一些选项作为参数，生成一个或者多个目标文件。scala编译生成的产物就是标准的 Java 类文件。

假设我们吧上述代码保存为文件 HelloWorld.scala，我们使用下面的命令编译它（大于号“>”表示命令提示符，你不必输入它）

```
> scalac HelloWorld.scala
```

这将会在当前目录生成一系列.class 文件。其中的一个名为 HelloWorld.class 的文件中定义了一个可以直接使用 scala 命令执行的类。下文中你可以看到这个例子。

2.2 运行实例

一旦完成编译，Scala 程序就可以使用 scala 命令执行了。scala 的用法和 java 很相似，并且连选项也大致相同。上面的例子就可以使用下面的命令运行，这将会产生我们所期望的输出。

```
> scala -classpath . HelloWorld
Hello, world!
```

3 Scala与Java交互

Scala 的一个强项在于可以很简单的于已有的 Java 代码交互，所有 java.lang 中的类都已经被自动导入了，而其他的类需要显式声明导入。

来看看演示代码吧。我们希望对日期进行格式化处理，比如说用法国的格式。

Java 类库定义了一系列很有用的类，比如 Date 和 DateFormat。由于 Scala 于 Java 能够进行很好的交互，我们不需要在 Scala 类库中实现等效的代码，而只需直接吧 Java 的相关类导入就可以了：

```
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)
      }
}
```

Scala的import语句看上去与Java的非常相似，但是它更加强大。你可以使用大括号来导入同一个包里的多个类，就像上面代码中第一行所做的那样。另一个不同点是当导入一个包中所有的类或者符号时，你应该使用下划线（_）而不是星号（*）。这是由于星号在Scala中是一个有效的标识符（例如作为方法名称）。这个例子我们稍后会遇到。

第三行的 import 语句导入了 DataFormat 类中的所有成员，这使得静态方法 getDateInstance 和静态变量 LONG 可以被直接引用。

在 main 函数中，我们首先建立了一个 Java 的 Date 实例。这个实例默认会包含当前时间。接下来我们一个使用刚才导入的静态函数 getDateInstance 定义了日期格式。最后我们将使用 DataFotmat 格式化好的日期打印了出来。最后一行代码显示了 Scala 的一个有趣 的语法：只有一个参数的函数可以使用下面这样的表达式来表示：

```
df format now
```

其实就是下面的这个冗长的表达式的简洁写法

```
df.format(now)
```

这看起来是一个语法细节，但是它导致一个重要的后果，我们将在下一节进行说明。

另外，我们还应当注意到 Scala 中可以直接继承或者实现 Java 中的接口和类。

4 Scala：万物皆对象

Scala 作为一个纯面向对象的语言，于是在 Scala 中万物皆对象，包括数字和函数。在这方面，Scala 于 Java 存在很大不同：Java 区分原生类型（比如 boolean 和 int）和引用类型，并且不能把函数当初变量操纵。

4.1 数字和对象

由于数字本身就是对象，所以他们也有方法。事实上我们平时使用的算数表达式（如下例）

```
1 + 2 * 3 / x
```

是由方法调用组成的。它等效于下面的表达式，我们在上一节见过这个描述。

```
(1).+(((2).*(3))./(x))
```

这也意味着 +，-，*，/ 在Scala中也是有效的名称。

在第二个表达式中的这些括号是必须的，因为 Scala 的分词器使用最长规则来进行分词。所以他会把下面的表达式：

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

理解成表达项 1. ，+，和 2 的组合。这样的组合结果是由于 1.是一个有效的表达项并且比表达项 1 要长，表达项 1.会被当作 1.0 ，使得它成为一个 double 而不是 int。而下面的表达式阻止了分析器错误的理解

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

4.2 函数与对象

函数在 Scala 语言里面也是一个对象，也许这对于 Java 程序员来说这比较令人惊讶。于是吧函数作为参数进行传递、把它们存贮在变量中、或者当作另一个函数的返回值都是可能的。吧函数当成值进行操作是函数型编程语言的基石。

为了解释为什么吧函数当作值进行操作是十分有用的，我们来考虑一个计时器函数。这个函数的目的是每隔一段时间就执行某些操作。那么如何吧我们要做的 操作传入计时器呢？于是我们想吧他当作一个函数。这种目前的函数对于经常进行用户界面编程的程序员来说是最熟悉的：注册一个回调函数以便在事件发生后得到 通知。

在下面的程序中，计时器函数被叫做 oncePerSceond，它接受一个回调函数作为参数。这种函数的类型被写作 () => Unit ，他们不接受任何参数也没有任何返回（Unit 关键字类似于 C/C++中的 void）。程序的主函数调用计时器并传递一个打印某个句子的函数作为回调。换 句话说，这个程序永无止境的每秒打印一个“time flies like an arrow”。

```
object Timer {
      def oncePerSecond(callback: () => Unit) {
              while (true) { callback(); Thread sleep 1000 }
      }
      def timeFlies() {
              println("time flies like an arrow...")
      }
      def main(args: Array[String]) {
              oncePerSecond(timeFlies)
      }
}
```

注意，我们输出字符串时使用了一个预定义的函数 println 而不是使用 System.out 中的那个。

4.2.1 匿名函数

我们可以吧这个程序改的更加易于理解。首先我们发现定义函数 timeFlies 的唯一目的就是当作传给 oncePerSecond 的参数。这么看来 给这种只用一次的函数命名似乎没有什么太大的必要，事实上我们可以在用到这个函数的时候再定义它。这些可以通过匿名函数在 Scala 中实现，匿名函数顾名 思义就是没有名字的函数。我们在新版的程序中将会使用一个匿名函数来代替原来的 timeFlise 函数，程序看起来像这样：

```
object TimerAnonymous {
      def oncePerSecond(callback: () => Unit) {
              while (true) { callback(); Thread sleep 1000 }
      }
      def main(args: Array[String]) {
              oncePerSecond(() =>
                      println("time flies like an arrow..."))
      }
}
```

本例中的匿名函数使用了一个箭头（=>）吧他的参数列表和代码分开。在这里参数列表是空的，所以我们在右箭头的左边写上了一对空括号。函数体内容与上面的 timeFlise 是相同的。

5 Scala类

正如我们所见，Scala 是一门面向对象的语言，因此它拥有很多关于“类”的描述 。Scala 类使用和 Java 类似的语法进行定义。但是一个重要的不同点在于 Scala 中的类可以拥有参数，这样就可以得出我们下面关于对复数类（Complex）的定义：

```
class Complex(real: Double, imaginary: Double) {
      def re() = real
      def im() = imaginary
}
```

我们的复数类（Complex）接受两个参数：实部和虚部。这些参数必须在实例化时进行传递，就像这样：new Complex(1.5, 2.3)。类定义中包括两个叫做 re 和 im 的方法，分别接受上面提到的两个参数。

值得注意的是这两个方法的返回类型并没有显式的声明出来。他们会被编译器自动识别。在本例中他们被识别为 Double 但是编译器并不总是像本例中的那样进行自动识别。不幸的是关于什么时候识别，什么时候不识别的规则相当冗杂。在实践中这通常不会成为一个问题，因为 当编译器处理不了的时候会发出相当

的抱怨。作为一个推荐的原则，Scala 的新手们通常可以试着省略类型定义而让编译器通过上下文自己判断。久而久之，新 手们就可以感知到什么时候应该省略类型，什么时候不应该。

5.1 无参方法

关于方法 re 和 im 还有一个小问题：你必须在名字后面加上一对括号来调用它们。请看下面的例子：

```
object ComplexNumbers {
      def main(args: Array[String]) {
              val c = new Complex(1.2, 3.4)
              println("imaginary part: " + c.im())
      }
}
```

你可能觉得吧这些函数当作变量使用，而不是当作函数进行调用，可能会更加令人感到舒服。事实上我们可以通过定义无参函数在 Scala 做到这点。这类 函数与其他的具有 0 个参数的函数的不同点在于他们定义时不需要在名字后面加括弧，所以在使用时也不用加（但是无疑的，他们是函数），因此，我们的 Complex 类可以重新写成下面的样子；

```
class Complex(real: Double, imaginary: Double) {
      def re = real
      def im = imaginary
}
```

5.2 继承和覆盖

Scala 中的所有类都继承一个父类，当没有显示声明父类时（就像上面定义的 Complex 一样），它们的父类隐形指定为 scala.AnyRef。

在子类中覆盖父类的成员是可能的。但是你需要通过 override 修饰符显示指定成员的覆盖。这样的规则可以避免意外覆盖的情况发生。作为演示，我们在 Complex 的定义中覆盖了 Object 的 toString 方法。

```
class Complex(real: Double, imaginary: Double) {
      def re = real
      def im = imaginary
      override def toString() =
              "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

6 Scala的模式匹配和条件类

树是在程序中常用的一个数据结构。例如编译器和解析器常常吧程序表示为树；XML 文档结构也是树状的；还有一些集合是基于树的，例如红黑树。

接下来我们将通过一个计算器程序来研究树在 Scala 中是如何表示和操纵的。这个程序的目标是处理一些由整数常量、变量和加号组成的简单的算数表达式，例如 1 + 2 和 (x + x) + (7 + y)。

我们首先要决定如何表示这些表达式。最自然的方法就是树了，树的节点表示操作符（在这里只有加法），而树的叶节点表示值（这里表示常数和变量）。 在 Java 中，这样的树可以表示为一个超类的树的集合，节点由不同子类的实例表示。而在函数式语言中，我们可以使用代数类型（algebraic data-type）来达到同样的目的。Scala 提供了一种介于两者之间的叫做条件类（Case Classes）的东西。

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

我们实际上定义了三个条件类 Sum ,Var 和 Const 。这些类和普通类有若干不同：

1. 实例化时可以省略 new 关键字（例如你可以使用 Const(5)而不必使用 new Const(5) ）
2. 参数的 getter 函数自动定义（例如你可以通过 c.v 来访问类 Const 的实例 c 在实例化时获取的参数 v）
3. 拥有默认的预定义 equals 和 hashCode 实现，这些实现可以按照值区别类实例是否相等，而不是通过用。
4. 拥有默认的 toString 实现。这些实现返回值的代码实现（例如表达式 x+1 可以被表达成 Sum(Var(x),Const(1))）
5. 条件类的实例可以通过模式匹配进行分析，我们接下来就要讲这个特性。

现在我们已经定义了表示我们算数表达式的数据类型，于是我们可以开始给他们定义对应的操作。我们将会首先编写一个在上下文中下计算表达式的函数。这里的上下文指的是变量与值的绑定关系。例如表达式 x+1 在 x=5 上下文中应该得出结果 6。

这样一来我们需要找到一个表示这种绑定关系的方法。当然我们可以使用某种类似 hash-table 的数据结构，不过我们也可以直接使用函数！一个上下文无非就是一个吧名称映射到值的函数。例如上面给出的{x → 5}的这个映射我们就可以在 Scala 中表示为：

```
{ case "x" => 5 }
```

这个定义了一个函数:当参数等于字符串"x"时返回整数 5，否则抛出异常。

在编写求值函数之前我们，我们需要给我们的上下文起个名字，以便在后面的代码里面引用。理所应当的我们使用了类型 String=>Int，但 是如果我们给这个类型起个名字，将会让程序更加简单易读，而且更加容易维护。在 scala 中，这件事情可以通过以下代码完成：

```
type Environment = String => Int
```

从现在开始，类型 Environment 就当作 String 到 Int 的函数类型名来使用了。

现在我们可以开始定义求值函数了。从概念上来说，这是很简单的一个过程：两个表达式之和等于两个表达式分别求值后再求和；变量的值可以从上下文中提取；常量的值就是他本身。在 Scala 中表达这个没有什么难度：

```
def eval(t: Tree, env: Environment): Int = t match {
      case Sum(l, r) => eval(l, env) + eval(r, env)
      case Var(n)     => env(n)
      case Const(v)   => v
}
```

求值函数通过对树 t 进行模式匹配来完成工作。直观的来看，上述代码的思路是十分清晰的：

1. 第一个模式检查传入的树的根节点是否是一个 Sum，如果是，它将会吧树的左边子树赋值给 l,右边的子树赋值给 r,然后按照箭头后面的代码进行处理；这里的代码可以（并且的确）使用了在左边匹配时所绑定的变量，比如这里的 l 和 r。
2. 如果第一个检查没有成功，表明传入的树不是 Sum,程序继续检查他是不是一个 Var；如果是，则吧变量名赋给 n 然后继续右边的操作。
3. 如果第二个检查也失败了，表示 t 既不是 Sum 也不是 Var，程序检查他是不是 Const。如果是着赋值变量并且继续。
4. 最后，如果所有检查都失败了。就抛出一个异常表示模式匹配失败。这只有在 Tree 的其他之类被定义时才可能发生。

我们可以看出模式匹配的基本思想就是试图对一个值进行多种模式的匹配，并且在匹配的同时将匹配值拆分成若干子项，最后对匹配值与其子项执行某些代码。

一个熟练的面向对象的程序员可能想知道为什么我们不吧 eval 定义为 Tree 或者其之类的成员函数。我们事实上可以这么做。因为 Scala 允许条件类象普通类那样定义成员。决定是否使用模式匹配或者成员函数取决于程序员的喜好，不过这个取舍还和可扩展性有重要联系：

1. 当你使用成员函数时，你可以通过继承 Tree 从而很容易的添加新的节点类型，但是另外一方面，添加新的操作也是很繁杂的工作，因为你不得不修改 Tree 的所有子类。
2. 当你使用模式匹配是，形势正好逆转过来，添加新的节点类型要求你修改所有的对树使用模式匹配的函数，但是另一方面，添加一个新的操作只需要再添加一个模式匹配函数就可以了。

下面我们来更详细的了解模式匹配，让我们再给表达式定义一个操作：对符号求导数。读者们也许想先记住下面关于此操作的若干规则：

1. 和的导数等于导数的和，
2. 如果符号等以求导的符号，则导数为 1，否则为 0.
3. 参数的导数永远为 0。

上述规则可以直接翻译成 Scala 代码:

```
def derive(t: Tree, v: String): Tree = t match {
      case Sum(l, r) => Sum(derive(l, v), derive(r, v))
      case Var(n) if (v == n) => Const(1)
      case _ => Const(0)
}
```

这个函数使用了两个关于模式匹配的功能，首先 case 语句可以拥有一个 guard 子句：一个 if 条件表达式。除非 guard 的条件成立，否则该模式不会成功匹配。其次是通配符：_ 。这个模式表示和所有值匹配而不对任何变量赋值。

事实上我们还远没有触及模式匹配的全部精髓。但是我们限于篇幅原因不得不再此停笔了。下面我们看看这个两个函数是如何在一个实例上运行的。为了达到这个目前我们写了一个简单的 main 函数来对表达式(x + x) + (7 + y)进行若干操作：首先计算当{x → 5, y → 7}时表达式的值，然后分别对 x 和 y 求导。

```
def main(args: Array[String]) {
      val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
      val env: Environment = { case "x" => 5 case "y" => 7 }
      println("Expression: " + exp)
      println("Evaluation with x=5, y=7: " + eval(exp, env))
      println("Derivative relative to x:\n " + derive(exp, "x"))
      println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

执行程序，我们能得到以下输出：

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y))) Evaluation
with x=5, y=7: 24
Derivative relative to x:
```

```
Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

通过研究程序输出，我们能看到求导的输出可以在被打印之前简化，使用模式匹配定义一个简化函数是挺有意思的（不过也需要一定的技巧）工作。读者可以尝试自己完成这个函数。

7 Scala Trait

除了从父类集成代码外，Scala 中的类还允许从一个或者多个 traits 中导入代码。

对于 Java 程序员来说理解 traits 的最好方法就是把他们当作可以包含代码的接口（interface）。在 Scala 中，当一个类继承一个 trait 时，它就实现了这个 trait 的接口，同时还从这个 trait 中继承了所有的代码。

让我们通过一个典型的实例来看看这种 trait 机制是如何发挥作用的：排序对象。能够比较若干给定类型的对象在实际应用中是很有用的，比如在进行排 序时。在 Java 语言中可以比较的对象是通过实现 Comparable 接口完成的 。在 Scala 中我们可以通过吧 Comparable 定义为 trait 来做的比 Java 好一些。我们吧这个 trait 叫做 Ord。

在比较对象时，一下六种关系通常使用率最高：小于、小于等于、等于、不等于、大于等于、大于。但是把他们都定义一次无疑是很没用而且繁琐的。尤其是 六种关系中的四种其实是可以通过其他两种关系导出的。例如给定等于和小于的定义后就可以推导出其他的定义。于是在 Scala 中，这些推导可以通过下面这个 trait 实现：

```
trait Ord {
      def < (that: Any): Boolean
      def <=(that: Any): Boolean = (this < that) || (this == that)
      def > (that: Any): Boolean = !(this <= that)
      def >=(that: Any): Boolean = !(this < that)
}
```

这个定义在建立了一个叫做与 Java 中的 Comparable 等效的叫做 Ord 的类型的同时还实现了使用抽象的一种关系推导其他三种的接口。比较相等性的方法没有出现是由于他已经默认存在于所有对象中了。

上面使用的叫做 Any 的类型表示了 Scala 中所有类的共同超类。事实上它就等于 Java 语言中的 Object。

要使的一个类可以被比较，就需要可以比较他们是否相等或者大小关系，而这些都混合在上面的类 Ord 中了。现在我们来写一个 Date 类来表示格利高里历中的日期。这个日期由年、月、日三个部分组成，每个部分都可以用一个整数表示。所有我们就得出了下面这个定义：

```
class Date(y: Int, m: Int, d: Int) extends Ord {
      def year = y
      def month = m
      def day = d
      override def toString(): String = year + "-" + month + "-" + day
}
```

注意在类名后出现的 extends Ord。这表示了这个类继承了 Ord 这个 trait。

然后我们重新定义了 equals 这个从 Object 继承来的方法，好让他能够正确的比较我们日期中的每个部分。原来的 equals 函数的行为与 Java 中的一样，是按照对象的指针进行比较的。我们可以得出下面的代码。

```
override def equals(that: Any): Boolean =
that.isInstanceOf[Date] && {
      val o = that.asInstanceOf[Date]
      o.day == day && o.month == month && o.year == year
}
```

这个函数使用了预定义函数 isInstanceOf 和 asInstanceOf 。第一个 isInstanceOf 类似 Java 中的 instanceof ：当且仅当对象是给定类型的实例时才返回 true。第二个 asInstanceOf 对应 Java 中的类型转换操作：当对象是给定类型的子类时转换，否则抛出 ClassCastException。

最后我们还需要定义测试小于关系的函数，如下面所示。这个函数使用了预定义的函数 error ，它可以使用给定字符串抛出一个异常。

```
def <(that: Any): Boolean = {
      if (!that.isInstanceOf[Date])
            error("cannot compare " + that + " and a Date")
      val o = that.asInstanceOf[Date] (year < o.year) ||
            (year == o.year && (month < o.month ||
            (month == o.month && day < o.day)))
}
```

以上就是 Data 的完整定义了。这个类的实例既可以作为日期显示，也可以进行比较。而且他们都定义了 6 种比较操作：其中两种 ： equals 和< 是我们直接定义的，而其他的是从 Ord 中继承的。

Scala Traits 的应用远不止于此，不过更加深入的讨论不再本文的讨论范围内。

8 Scala的泛型

我们在这文章将要学习 Scala 的最后一个特性是泛型。Java 程序员们可能最近才知道这个东西，因为这个特性是在 Java1.5 中才被加入的。

泛型是一种可以让你使用类型参数的设施。例如当一个程序员正在实现一个链表时，将不得不面对诸如如何决定链表中节点保存数据的类型之类的问题。正由 于这是一个链表，所以往往会在不同的环境中使用，因此，我们不能草率的决定节点数据类型，比如说 Int。这种决定是相当的草率且局限性的。

以前 Java 程序员们通常使用 Object，所有类型的超类，来解决问题。但是这种方法远远算不上是理想方案，例如他无法处理基本类型如 int、 long、float 等（1.6 中的 autobox 特性可以解决这个问题——译者注），而且会让使用者不得不使用大量的动态类型转换。

Scala 中的泛型机制可以很轻松的解决这些个问题。来看下面这个最简单的容器类：一个引用，可以指向某个对象或者指向空。

```
class Reference[T] {
      private var contents: T = _
      def set(value: T) { contents = value }
      def get: T = contents
}
```

Reference 类具有一个叫做 T 的类型参数来表示他说引用的对象的类型。这个类型在 Reference 中作为了变量和函数的参数或者返回类型。

上面的代码还演示了Scala中变量的表达方式，这个无需更多的解释大家都能清楚。不过值得注意的是我们给他赋予的初始值：_ ,这个表示一个默认值，对于数字类型来说是 0，对于boolean来说是false,对于Unit（函数签名）来说是()（无参数无返回），对于其他来说是null。

要使用这个 Reference 类，你需要制定他的类型参数，来告知这个引用到底引用了什么类型。例如要创建一个指向 Int 的引用，你可以这么写：

```
object IntegerReference {
      def main(args: Array[String]) {
            val cell = new Reference[Int] cell.set(13)
            println("Reference contains the half of " + (cell.get * 2))
      }
}
```

就像我们看到的，我们不需要吧 get 的返回值强制转换成 Int，而且由于它被声明成 Int，你不可能在这个引用中放置其他类型的对象。

9 结语

本文简要介绍了 Scala 语言的一些特性，并且同时展示了若干实例。有兴趣的读者可以继续阅读本文的姊妹篇： 《Scala By Example》，该文覆盖了 Scala 的更多的高级特性。如果需要还可以去阅读《Scala Language Specification》。