在线实验，请到PC端体验

函数

一、实验介绍

1.1 实验内容

当程序越来越大时，你需要将代码细化为小的容易管理的模块。Scala支持多种方法来细化程序代码，这些方法也为有经验的程序员已经掌握的：使用函数。

在本节实验中，我们将为你介绍类成员函数、局部函数等具有Scala特色的函数使用方法。

1.2 实验知识点

- 类成员函数
- 局部函数
- 函数的头等公民地位
- 函数字面量的一些简化写法
- 部分应用的函数
- 闭包
- 可变参数、命名参数和缺省参数

1.3 实验环境

- Scala 2.11.7
- Xfce 终端

1.4 适合人群

本课程难度为一般，属于初级级别课程，适合零基础或具有 Java 编程基础的用户。

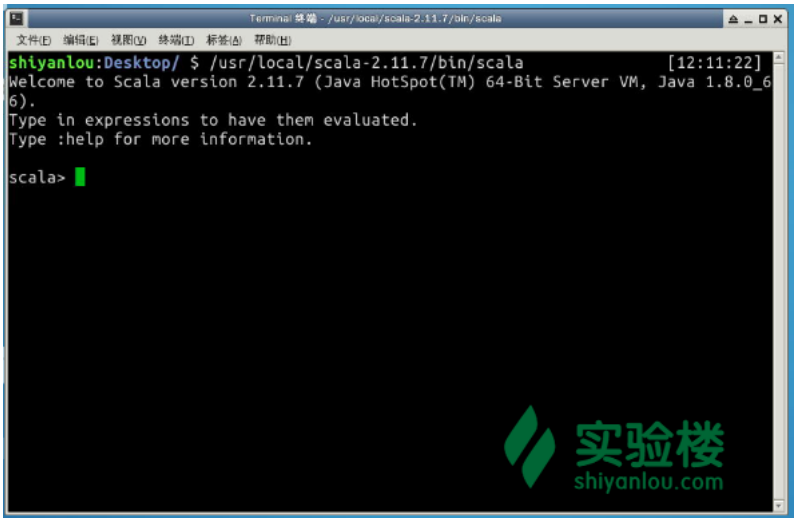
二、开发准备

为了使用交互式 Scala 解释器，你可以在打开的终端中输入命令：

```
cd /usr/local/scala-2.11.7/bin/

scala
```

当出现 `scala>` 开始的命令行提示符时，就说明你已经成功进入解释器了。如下图所示。



三、实验步骤

3.1 函数的具体使用 之 类成员函数

和 Java 相比，Scala 提供了多种 Java 不支持的方法来定义函数，除了类成员函数外，Scala 还支持嵌套函数，函数字面量，函数变量等。

本节先介绍类或对象的成员函数。这也是最常见的定义函数的方法。例如，下面的例子定义了两个成员函数：

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

   private def processLine(filename:String,
      width:Int, line:String){
     if(line.length > width)
       println(filename + ":" +line.trim)
   }

}
```

成员函数 `processFile` 使用两个参数，一个是文件名，另一个为字符长度，其作用是打印出文件中超过指定字符长度的所有行。它调用另外一个私有成员函数 `processLine` 完成实际的操作。

这个成员函数，如果作为脚本使用，可以使用如下代码：

```
LongLines.processFile(args(0),args(1).toInt)
```

可以看到，Scala 类成员函数的使用方法和其它面向对象的程序语言（如 Java ）基本一致。在后面的内容中，我们将继续介绍 Scala 函数不同于 Java 的一些特性。

3.2 局部函数

上个例子中， `processFile` 使用了一个非常重要的设计原则——应用程序可以分解成多个小的函数，每个小的函数完成一个定义完好的功能。

使用这种程序设计风格，可以让程序中有相当数量的程序构造模块。通过这些小的构造模块的组合来完成较复杂的功能。每个小的构造模块应该足够简洁，以帮助理解。

这样带来的一个问题是：这些小的辅助函数的名称可能会影响到程序空间，你不能在同个程序中使用两个相同名称的函数，即使你定义私有函数。如果你设计函数库，你也不希望有些辅助函数被库函数的用户直接调用。

对于 Java 来说，你可以通过私有成员函数来达到目的。而Scala 除了支持私有成员函数外，还支持局部函数（其作用域和局部变量类似）。

也就是说，你可以在函数的内部再定义函数，如同定义一个局部变量。例如，修改前面的 `processFile` 的例子如下：

```
import scala.io.Source
object LongLines {
    def processFile(filename: String, width: Int) {
        def processLine(filename:String,width:Int, line:String){
            if(line.length > width)
            println(filename + ":" +line.trim)
        }

    val source= Source.fromFile(filename)
    for (line <- source.getLines())
        processLine(filename,width,line)
    }
}
```

这个例子中，私有成员函数 processLine 移动到 processFile 内部，成为了一个局部函数。也正因为如此， `processLine` 可以直接访问到 `processFile` 的参数 `filename` 和 `width` 。因此，代码可以进一步优化如下：

```
import scala.io.Source
object LongLines {
    def processFile(filename: String, width: Int) {
        def processLine(line:String){
            if(line.length > width)
            println(filename + ":" +line.trim)
    }

    val source= Source.fromFile(filename)
    for (line <- source.getLines())
        processLine(line)
    }
}
```

代码变得更简洁了，对不对？局部函数的作用域和局部变量作用域一样，局部函数访问包含该函数的参数是非常常见的一种嵌套函数的用法。

3.3 头等公民

Scala 中，函数是头等公民。你不仅可以定义一个函数然后调用它，你还可以写一个未命名的函数字面量，然后把它当成一个值，传递到其它函数或是赋值给其它变量。

下面的例子为一个简单的函数字面量（参考整数字面量， `3` 为一整数字面量）。

```
(x :Int ) => x +1
```

这是个函数字面量，它的功能为 `+1` 。符号 `=>` 表示这个函数将符号左边的东西（本例为一个整数），转换成符号右边的东西（加 1 ）。

函数字面量为一个对象（就像 `3` 是个对象）。因此，如果你愿意的话，可以把这个函数字面量保持在一个变量中。这个变量也是一个函数，因此你可以使用函数风格来调用它，比如：

```
scala> var increase = (x :Int ) => x +1
increase: Int => Int = <function1>

scala> increase(10)
res0: Int = 11
```

注意，函数字面量 `(x:Int) => x + 1` 在 Scala 内部，表示为带有一个参数的类 `function1` 的一个对象。其它情况比如 `functionN` 代表带有 N 个参数的函数， `function0` 代表不含参数的函数类型。

如果函数定义需要多条语句，可以使用 `{}` ，比如：

```
scala> var increase = (x :Int ) => {
     |    println("We")
     |    println("are")
     |    println("here")
     |    x + 1
     |    }
increase: Int => Int = <function1>

scala> increase (10)
We
are
here
res0: Int = 11
```

从上面的内容中，我们了解到了函数字面量的基本概念。它可以作为参数传递给其它函数，比如很多 Scala 的库允许你使用函数作为参数（比如 foreach 方法，它使用一个函数参数，为集合中每个运算调用传入的函数）。例如：

```
scala> val someNumbers = List ( -11, -10, - 5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.foreach((x:Int) => println(x))
-11
-10
-5
0
5
10
```

再比如，Scala 的集合也支持一个 `filter` 方法用来过滤集合中的元素， `filter` 的参数也是一个函数，比如：

```
scala> someNumbers.filter( x => x >0)
res1: List[Int] = List(5, 10)
```

使用 `x => x >0` ，过滤掉小于 `0` 的元素。如果你熟悉 `lambda` 表达式 (http://baike.baidu.com/view/3048187.htm)， `x => x >0` 为函数的 `lambda` 表达式。

3.4 函数字面量的一些简化写法

Scala 提供了多种方法来简化函数字面量中多余的部分。比如前面例子中， `filter` 方法中使用的函数字面量，完整的写法如下：

```
(x :Int ) => x +1
```

首先可以省略到参数的类型，Scala 可以根据上下文推算出参数的类型，函数定义可以简化为：

```
(x) => x +1
```

这个函数可以进一步去掉参数的括号，这里的括号不起什么作用：

```
x => x +1
```

Scala 还可以进一步简化：Scala 允许使用“占位符”下划线“ _ ”来替代一个或多个参数，只要这个参数值函数定义中只出现一次，Scala 编译器可以推断出参数。比如：

```
scala> val someNumbers = List ( -11, -10, - 5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> someNumbers.filter(_ >0)
res0: List[Int] = List(5, 10)
```

可以看到，简化后的函数定义为 `_ > 0` ，你可以这样来理解：就像我们以前做过的填空题，“ _ ”为要填的空，Scala 会来完成这个填空题，而你来定义填空题。

有时，如果你使用 _ 来定义函数，可能没有提供足够的信息给 Scala 编译器。此时， Scala 编译器将会报错。比如，定义一个加法函数如下：

```
scala> val f = _ + _
<console>:7: error: missing parameter type for expanded function ((x$1, x$2) => x$1.$plus(x$2))
       val f = _ + _
               ^
<console>:7: error: missing parameter type for expanded function ((x$1: <error>, x$2) => x$1.$plus(x$2))
       val f = _ + _
```

Scala 编译器无法推断出 _ 的参数类型，所以报错了。但如果你给出了参数的类型，就依然可以使用 _ 来定义函数。比如：

```
scala> val f = (_ :Int ) + ( _ :Int)
f: (Int, Int) => Int = <function2>

scala> f (5,10)
res1: Int = 15
```

因为 _ 替代的参数在函数体中只能出现一次，因此多个“ _ ”代表多个参数。第一个“ _ ”代表第一个参数，第二个“ _ ”代表第二个参数，以此类推。

3.5 部分应用的函数

前面的例子中，我们使用了“ _ ”来代替单个的参数。实际上，你也可以使用“ _ ”来代替整个参数列表。比如说，你可以使用 `print _` 来代替 `println (_)`。

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

Scala 编译器自动将上面代码解释成：

```
someNumbers.foreach( x => println (x))
```

因此这里的“ _ ”代表了 `println` 的整个参数列表，而不仅仅替代单个参数。

当你采用这种方法使用“ _ ”，你就创建了一个部分应用的函数(partially applied function)。在 Scala 中，当你调用函数，传入所需参数，你就把函数“应用”到参数。比如：一个加法函数。

```
scala> def sum = (_:Int) + (_ :Int) + (_ :Int)
sum: (Int, Int, Int) => Int

scala> sum (1,2,3)
res0: Int = 6
```

一个部分应用的函数指的是你在调用函数时，不指定函数所需的所有参数。这样，你就创建了一个新的函数，这个新的函数就称为原始函数的 部分应用函数 。比如说，我们固定 `sum` 的第一和第三个参数，定义如下的部分应用函数：

```
scala> val b = sum ( 1 , _ :Int, 3)
b: Int => Int = <function1>

scala> b(2)
res1: Int = 6
```

变量 `b` 的类型为一函数，具体类型为 `function1` （带一个参数的函数），它是由 `sum` 应用了第一个和第三个参数构成的。

调用 `b(2)` ，实际上调用 `sum(1, 2, 3)` 。

再比如，我们定义一个新的部分应用函数，只固定中间参数：

```
scala> val c = sum (_:Int, 2, _:Int)
c: (Int, Int) => Int = <function2>

scala> c(1,3)
res2: Int = 6
```

变量 `c` 的类型为 `function2` ，调用 `c(1, 3)` 实际上也是调用 `sum(1, 2, 3)` 。

在 Scala 中，如果你定义一个部分应用函数并传入所有参数，你也可以省掉 _ 。比如：

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

可以写成：

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

3.6 闭包

到目前为止，我们介绍的函数都只引用到传入的参数。假如我们定义如下的函数：

```
(x:Int) => x + more
```

这里我们引入一个自由变量 `more` 。它不是所定义函数的参数，而这个变量定义在函数外面。比如：

```
var more =1
```

那么我们有如下的结果：

```
scala> var more =1
more: Int = 1

scala> val addMore = (x:Int) => x + more
addMore: Int => Int = <function1>

scala> addMore (100)
res1: Int = 101
```

这样定义的函数变量 `addMore` 成为一个“闭包”。因为它引用到函数外面定义的变量。定义这个函数的过程，是将这个自由变量捕获而构成一个封闭的函数。有意思的是，当这个自由变量发生变化时，Scala 的闭包能够捕获到这个变化，因此 Scala 的闭包捕获的是变量本身而不是当时变量的值。

比如：

```
scala> more =  9999
more: Int = 9999

scala> addMore ( 10)
res2: Int = 10009
```

同样的，如果变量在闭包在发生变化，也会反映到函数外面定义的闭包的值。比如：

```
scala> val someNumbers = List ( -11, -10, -5, 0, 5, 10)
someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)

scala> var sum =0
sum: Int = 0

scala> someNumbers.foreach ( sum += _)

scala> sum
res4: Int = -11
```

可以看到，在闭包中修改 `sum` 的值，其结果还是传递到闭包的外面。

如果一个闭包所访问的变量有几个不同的版本，比如一个闭包使用了一个函数的局部变量（参数），然后这个函数调用很多次，那么所定义的闭包应该使用所引用的局部变量的哪个版本呢？

简单的说，该闭包定义所引用的变量为定义该闭包时变量的值，也就是定义闭包时相当于保存了当时程序状态的一个快照。比如，我们定义下面一个函数闭包：

```
scala> def makeIncreaser(more:Int) = (x:Int) => x + more
makeIncreaser: (more: Int)Int => Int

scala> val inc1=makeIncreaser(1)
inc1: Int => Int = <function1>

scala> val inc9999=makeIncreaser(9999)
inc9999: Int => Int = <function1>

scala> inc1(10)
res5: Int = 11

scala> inc9999(10)
res6: Int = 10009
```

当你调用 `makeIncreaser(1)` 时，你就创建了一个闭包，该闭包定义的 `more` 的值为 `1` ，而调用 `makeIncreaser(9999)` 所创建的闭包的 `more` 的值为 `9999` 。此后你也无法修改已经返回的闭包的 `more` 的值。因此 `inc1` 始终为加一，而 `inc9999` 始终为加 `9999`。

3.7 可变参数、命名参数和缺省参数

前面我们介绍的函数的参数是固定的，本节介绍 Scala 函数支持的可变参数列表、命名参数和参数缺省值定义。

3.7.1 重复参数

Scala 在定义函数时，允许指定最后一个参数重复（变长参数），从而允许函数调用者使用变长参数列表来调用该函数。Scala 中使用“ * ”来指明该参数为重复参数。例如：

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

scala> echo()

scala> echo ("One")
One

scala> echo ("Hello","World")
Hello
World
```

在函数内部，变长参数的类型实际上是一个数组。比如上例的 `String *` 类型实际为 `Array[String]`。 然而，如果你现在试图直接传入一个数组类型的参数给这个参数，编译器会报错：

```
scala> val arr= Array("What's","up","doc?")
arr: Array[String] = Array(What's, up, doc?)

scala> echo (arr)
<console>:10: error: type mismatch;
 found   : Array[String]
 required: String
              echo (arr)
                    ^
```

为了避免这种情况发生，你可以通过在变量后面添加 `_*` 来解决。这个符号告诉 Scala 编译器，在传递参数时，逐个传入数组的每个元素，而不是数组整体。

```
scala> echo (arr: _*)
What's
up
doc?
```

3.7.2 命名参数

通常情况下，调用函数时，参数传入和函数定义时参数列表是一一对应的。

```
scala> def  speed(distance: Float, time:Float) :Float = distance/time
speed: (distance: Float, time: Float)Float

scala> speed(100,10)
res0: Float = 10.0
```

使用命名参数时，允许你使用任意顺序传入参数。比如下面的调用：

```
scala> speed( time=10,distance=100)
res1: Float = 10.0

scala> speed(distance=100,time=10)
res2: Float = 10.0
```

3.7.3 缺省参数值

Scala 在定义函数时，允许指定参数的缺省值，从而允许在调用函数时不指明该参数，此时该参数使用缺省值。缺省参数通常配合命名参数使用，例如：

```
scala> def printTime(out:java.io.PrintStream = Console.out, divisor:Int =1 ) =
     | out.println("time = " + System.currentTimeMillis()/divisor)

printTime: (out: java.io.PrintStream, divisor: Int)Unit

scala> printTime()
time = 1383220409463

scala> printTime(divisor=1000)
time = 1383220422
```

四、实验总结

在本实验中，我们了解了成员函数、局部函数等具有Scala特色的函数使用方法，然而这些对于函数的使用仍然不是足够的。请继续学习下一实验，我们将在下一节继续讲解如何使用函数。

上一节 (/courses/490/labs/1688/document)　　下一节 (/courses/490/labs/1690/document)

课程教师

引路蜂

共发布过**6**门课程

CSDN 专家博主，擅长Java ME, Blackberry ,LWUIT , iPhone, Android, Windows Mobile, Mono , Windows Phone 7等平台开发，主页 http://www.imobilebbs.com/

查看老师的所有课程 > (/teacher/164063)

进阶课程

Scala 专题教程 - Case Class和模式匹配 (/courses/514)

Scala 专题教程 - 隐式变换和隐式参数 (/courses/515)

Scala 专题教程 - 抽象成员 (/courses/516)

Scala 专题教程 - Extractor (/courses/526)