在线实验，请到PC端体验

控制语句

一、实验介绍

1.1 实验内容

和其它语言（比如Java，C#)相比，Scala只内置了为数不多的几种程序控制语句： `if`、 `while`、 `for`、 `try catch` 以及函数调用，这是因为从Scala诞生开始就包含了函数字面量，Scala内核没有定义过多的控制结构，而是可以通过额外的库来扩展程序的控制结构。

在本实验中，我们就将学习如何使用控制语句。

1.2 实验知识点

- if 表达式
- while 循环
- for 表达式
- 用 try 表达式处理异常
- Match 表达式
- 代替“break”和“continue”

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。

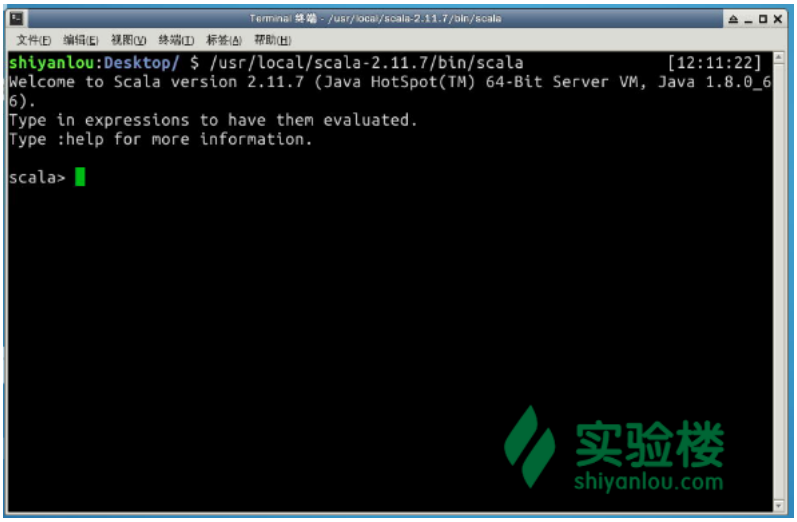
二、开发准备

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 控制语句详解

Scala 的所有控制结构都有返回结果，如果你使用过 Java 或 C# ，就可能了解 Java 提供的三元运算符 `?:` ，它的基本功能和 `if` 一样，当可以返回结果。Scala 在此基础上所有控制结构（ `while` 、 `try` 、 `if` 等）都可以返回结果。这样做的一个好处是，可以简化代码，如果没有这种特点，程序员常常需要创建一个临时变量用来保存结果。

总的来说，Scala 提供的基本程序控制结构，“麻雀虽小，五脏俱全”，虽然少，但足够满足其他指令式语言（如 Java,C++ ）所支持的程序控制功能。而且，由于这些指令都有返回结果，可以使得代码更为精简。

3.2 if 表达式

Scala 语言的 `if` 的基本功能和其它语言没有什么不同，它根据条件执行两个不同的分支。比如，使用 Java 风格编写下面 Scala 的 if 语句的一个例子：

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

上面代码和使用 Java 实现没有太多区别，看起来不怎么像 Scala 风格，我们重新改写一下，利用 `if` 可以返回结果这个特点。

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

首先这种代码比前段代码短，更重要的是这段代码使用 `val` 而无需使用 `var` 类型的变量。使用 `val` 为函数式编程风格。

3.3 while 循环

Scala 的 `while` 循环和其它语言（如 Java）功能一样，它含有一个条件和一个循环体。只要条件满足，就一直执行循环体的代码。

比如，下面的计算最大公约数的一个实现：

```
def gcdLoop (x: Long, y:Long) : Long ={
   var a=x
   var b=y
   while( a!=0) {
      var temp=a
      a=b % a
      b = temp
   }
  b
}
```

动手实践是学习 IT 技术最有效的方式！ 开始实验

Scala 也有 do-while 循环，它和 while 循环类似，只是检查条件是否满足在循环体执行之后检查。

例如：

```
var line=""
do {
   line = readLine()
   println("Read: " + line)
} while (line !="")
```

Scala 的 while 和 do-while 称为“循环”而不是表达式，是因为它不产生有用的返回值（或是返回值为 Unit），可以写成 () 。() 的存在使得 Scala 的 Unit 和 Java 的 void 类型有所不同。

比如，下面的语句在 Scala 中解释器中执行：

```
scala> def greet() { println("hi")}
greet: ()Unit

scala> greet() == ()
<console>:9: warning: comparing values of types Unit and Unit using `==' will always yield true
              greet() == ()
                      ^
hi
res0: Boolean = true
```

可以看到（或者看到警告） greet() 的返回值和 () 比较结果为 true 。

注意另外一种可以返回 Unit 结果的语句为 var 类型赋值语句。如果你使用如下 Java 风格的语句将碰到麻烦：

```
while((line=readLine())!="")
  println("Read: " + line)
```

如果你试图编译或是执行这段代码会有如下警告：

```
/root/scala/demo.scala:2: warning: comparing values of types Unit and String using `!=' will always yield true
while((line=readLine())!="")
```

意思 Unit （赋值语句返回值）和 String 做不等比较永远为 true 。上面的代码会是一个死循环。

正因为 while 循环没有值，因此在纯函数化编程中应该避免使用 while 循环。Scala 保留 while 循环，是因为在某些时候使用循环代码比较容易理解。而如果使用纯函数化编程，需要执行一些重复运行的代码时，通常就需要使用回溯函数来实现，回溯函数通常看起来不是很直观。

比如前面计算最大公倍数的函数使用纯函数化编程使用回溯函数实现如下：

```
def gcd (x :Long, y:Long) :Long =
   if (y ==0) x else gcd (y, x % y
```

总的来说，推荐尽量避免在代码使用 while 循环，正如函数化编程要避免使用 var 变量一样。 而使用 while 循环时通常也会使用到 var 变量，因此在你打算使用 while 循环时需要特别小心，看是否可以避免使用它们。

3.4 for 表达式

Scala 中的 for 表达式有如一把完成迭代任务的瑞士军刀，它允许你使用一些简单的部件以不同的方法组合可以完成许多复杂的迭代任务。简单的应用，比如枚举一个整数列表，较复杂的应用可以同时枚举多个不同类型的列表，根据条件过滤元素，并可以生成新的集合。

3.4.1 枚举集合元素

这是使用 for 表示式的一个基本用法，和 Java 的 for 非常类型，比如下面的代码可以枚举当前目录下所有文件：

```
val filesHere = (new java.io.File(".")).listFiles

for(file <-filesHere)
  println(file)
```

其中如 `file <- filesHere` 的语法结构，在 Scala 中称为“生成器 (generator)”。本例中，`filesHere` 的类型为 `Array[File]`。每次迭代中，变量 `file` 会初始化为该数组中一个元素，`file` 的 `toString()` 为文件的文件名，因此 `println(file)` 打印出文件名。

Scala 的 `for` 表达式支持所有类型的集合类型，而不仅仅是数组，比如下面使用 `for` 表达式来枚举一个 `Range` 类型。

```
 scala> for(i <- 1 to 4) println("Interation" +i)
Interation 1
Interation 2
Interation 3
Interation 4
```

3.4.2 过滤

某些时候，你可能不想枚举集合中的每一个元素，而是只想迭代某些符合条件的元素。在 Scala 中，你可以为 `for` 表达式添加一个过滤器——在 `for` 的括号内添加一个 `if` 语句，例如：

修改前面枚举文件的例子，改成只列出 `.scala` 文件如下：

```
val filesHere = (new java.io.File(".")).listFiles

for( file   println(file)
```

如果有必要的话，你可以使用多个过滤器，只要添加多个 if 语句即可。比如，为保证前面列出的文件不是目录，可以添加一个 `if` ，如下面代码：

```
val filesHere = (new java.io.File(".")).listFiles

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

3.4.3 嵌套迭代

`for` 表达式支持多重迭代。下面的例子使用两重迭代，外面的循环枚举 `filesHere` ，而内部循环枚举该文件的每一行文字。实现了类似 Unix 中的 `grep` 命令：

```
val filesHere = (new java.io.File(".")).listFiles

def fileLines (file : java.io.File) =
   scala.io.Source.fromFile(file).getLines().toList

def grep (pattern: String) =
  for (
    file <- filesHere if file.getName.endsWith(".scala");
    line <- fileLines(file)
    if line.trim.matches(pattern)
  ) println(file + ":" + line.trim)

grep (".*gcd.*")
```

注意上面代码中，两个迭代之间使用了 `;` ，如果你使用 `{}` 替代 `for` 的 `()` 的括号，你可以不使用 `;` 分隔这两个“生成器”。这是因为，Scala 编译器不推算包含在括号内的省掉的 `;` 。使用 `{}` 改写的代码如下：

```
val filesHere = (new java.io.File(".")).listFiles

def fileLines (file : java.io.File) =
   scala.io.Source.fromFile(file).getLines().toList

def grep (pattern: String) =
  for {
    file <- filesHere if file.getName.endsWith(".scala")
    line <- fileLines(file)
    if line.trim.matches(pattern)
  } println(file + ":" + line.trim)

grep (".*gcd.*")
```

这两段代码是等效的。

3.4.4 绑定中间变量

你可能注意到，前面代码使用了多次 `line.trim` 。如果 `trim` 是个耗时的操作，你可以希望 `trim` 只计算一次。Scala 允许你使用 `=` 号来绑定计算结果到一个新变量。绑定的作用和 `val` 类似，只是不需要使用 `val` 关键字。例如，修改前面的例子，只计算一次 `trim` ，把结果保存在 `trimmed` 变量中。

```
val filesHere = (new java.io.File(".")).listFiles

def fileLines (file : java.io.File) =
   scala.io.Source.fromFile(file).getLines().toList

def grep (pattern: String) =
  for {
    file <- filesHere if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed=line.trim
    if trimmed.matches(pattern)
  } println(file + ":" + trimmed)

grep (".*gcd.*")
```

3.4.5 生成新集合

`for` 表达式也可以用来生产新的集合，这是 Scala 的 `for` 表达式比 Java 的 `for` 语句功能强大的地方。它的基本语法如下：

```
for clauses yield body
```

关键字 `yield` 放在 body 的前面， `for` 每迭代一次，就产生一个 `body` 。 `yield` 收集所有的 `body` 结果，返回一个 `body` 类型的集合。比如，前面列出所有 `.scala` 文件，返回这些文件的集合：

```
def scalaFiles =
  for {
    file     if file.getName.endsWith(".scala")
  } yield file
```

`scalaFiles` 的类型为 `Array[File]` 。

3.5 用 try 表达式处理异常

Scala 的异常处理和其它语言比如Java类似，一个方法可以通过抛出异常的方法而不返回值的方式，来终止相关代码的运行。调用函数，可以捕获这个异常作出相应的处理，或者直接退出。在这种情况下，异常会传递给调用函数的调用者，依次向上传递，直到有方法处理这个异常。

3.5.1 抛出异常

Scala 抛出异常的方法和 Java 一样，使用 `throw` 方法。例如，抛出一个新的参数异常：

```
throw new IllegalArgumentException
```


尽管看起来似乎有些自相矛盾，Scala 中， `throw` 也是一个表达式，也是有返回值的。比如下面的例子：

```
val half =
  if (n % 2 == 0)
    n/2
  else
    throw new RuntimeException("n must be even")
```

当 n 为偶数时，n 初始化为 n 的一半；而如果 n 为奇数，将在初始化 half 之前就抛出异常。正因为如此，可以把 throw 的返回值类型视作任意类型。技术上来说，抛出异常的类型为 Nothing 。对于上面的例子来说，整个 if 表达式的类型为可以计算出值的那个分支的类型。如果 n 为 Int ，那么 if 表示式的类型也是 Int 类型，而不需要考虑 throw 表达式的类型。

3.5.2 捕获异常

Scala 捕获异常的方法和后面介绍的“模式匹配”的使用方法是一致的。比如：

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

try {
  val f = new FileReader("input.txt")
} catch {
  case ex: FileNotFoundException => //handle missing file
  case ex: IOException => //handle other I/O error
}
```

模式匹配将在后面介绍，try-catch 表达式的基本用法和 Java 一样。如果 try 块中代码在执行过程中出现异常，将逐个检测每个 catch 块。在上面的例子，如果打开文件出现异常，将先检查是否是 FileNotFoundException 异常。如果不是，再检查是否是 IOException 。如果还不是，再终止 try-catch 块的运行，而向上传递这个异常。

注意： 和Java异常处理不同的一点是，Scala 不需要你捕获 checked 的异常。这点和 C# 一样，也不需要使用 throw 来声明某个异常。当然，如果有需要，还是可以通过 @throw 来声明一个异常，但这不是必须的。

3.5.3 finally语句

Scala 也支持 finally 语句，你可以在 finally 块中添加一些代码。这些代码不管 try 块是否抛出异常，都会执行。比如，你可以在 finally 块中添加代码保证关闭已经打开的文件，而不管前面代码中是否出现异常。

```
import java.io.FileReader

val file = new FileReader("input.txt")

try {
  //use the file
} finally {
  file.close()
}
```

3.5.4 生成返回值

和大部分 Scala 控制结构一样，Scala 的 try-catch-finally 也生成某个值。比如下面的例子尝试分析一个 URL ，如果输入的 URL 无效，则使用缺省的 URL 链接地址：

```
import java.net.URL
import java.net.MalformedURLException

def urlFor(path:String) =
  try {
    new URL(path)
  } catch {
    case e: MalformedURLException =>
      new URL("http://www.scala-lang.org")
  }
```

通常情况下，finally 块用来做些清理工作，而不应该产生结果，但如果在 finally 块中使用 return 来返回某个值，这个值将覆盖 try-catch 产生的结果，比如：

```
scala> def f(): Int = try { return 1 } finally { return 2}
f: ()Int

scala> f
res0: Int = 2
```

而下面的代码：

```
scala> def g() :Int = try 1 finally 2
g: ()Int

scala> g
res0: Int = 1
```

结果却是 1 ，上面两种情况常常使得程序员产生困惑，因此关键的一点是避免在 finally 生成返回值，而只用来做些清理工作，比如关闭文件。

3.5.5 Match 表达式

Scala 的 Match 表达式支持从多个选择中选取其一，类似其它语言中的 switch 语句。通常来说，Scala 的 match 表达式支持任意的匹配模式，这种基本模式将在后面介绍。

接下来，为你介绍类似 switch 用法的 match 表达式，它也是在多个选项中选择其一。

例如，下面的例子从参数中读取食品的名称，然后根据食品的名称，打印出该和该食品搭配的食品。比如，输入 salt ，与之对应的食品为 pepper 。如果是 chips ，那么搭配的就是 salsa 等等。

```
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?")
}
```

这段代码和 Java 的 switch 相比有几点不同：

- 一是任何类型的常量都可以用在 case 语句中，而不仅仅是 int 或是枚举类型。
- 二是每个 case 语句无需使用 break ，Scala不支持“fall through”。
- 三是Scala的缺省匹配为 _ ，其作用类似 java 中的 default 。

而最关键的一点，是 scala 的 match 表达式有返回值。上面的代码使用的是 println 打印，而实际上你可以使用表达式，比如修改上面的代码如下：

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

println(friend)
```

这段代码和前面的代码是等效的，不同的是后面这段代码 match 表达式返回结果。

3.6 没有 “break” 和 “continue” 的日子

你也许注意到：到目前为止，我们介绍 Scala 的内置的控制结构时，没有提到使用 break 和 continue 。Scala 特地没有在内置控制结构中包含 break 和 continue ，这是因为这两个控制结构和函数字面量有点格格不入，函数字面量我们将在后面介绍。函数字面量和其它类型字面量，比如数值字面量 4 、 5.6 相比，他们在 Scala 的地位相同。

我们很清楚 break 和 continue 在循环控制结构中的作用。Scala内置控制结构特地去掉了 break 和 continue ，这是为了更好的适应函数化编程。不过，你不用担心，Scala 提供了多种方法来替代 break 和 continue 的作用。

一个简单的方法，是使用一个 `if` 语句来代替一个 `continue` ，使用一个布尔控制量来去除一个 `break` 。比如下面的 Java 代码在循环结构中使用 `continue` 和 `break` ：

```
int i=0;
boolean foundIt=false;
while(i <args.length) {
    if (args[i].startWith("-")) {
        i=i+1;
        continue;
    }
    if(args[i].endsWith(".scala")){
        foundIt=true;
        break;
    }
    i=i+1;
}
```

这段 Java 代码实现的功能，是从一组字符串中寻找以 `.scala` 结尾的字符串，但跳过以 `-` 开头的字符串。

下面我们使用 `if` 和 `boolean` 变量，逐句将这段实现使用Scala来实现（不使用 `break` 和 `continue`)如下：

```
var i=0
var foundIt=false
while (i < args.length && !foundIt) {
    if (!args(i).startsWith("-")) {
    if(args(i).endsWith(".scala"))
        foundIt=true
    }
    i=i+1
}
```

可以看到，我们使用 `if` （与前面的 `continue` 条件相反）去掉了 `continue` ，而重用了 `foundIt` 布尔变量，去掉了 `break` 。

这段代码和前面 Java 实现非常类似，并且使用了两个 `var` 变量。使用纯函数化编程的一个方法是去掉 `var` 变量的使用，而递归函数（回溯函数）是用于去除循环结构中使用 `var` 变量时，通常使用的一个方法。

使用递归函数重新实现上面代码实现的查询功能：

```
def searchFrom(i:Int) : Int =
    if( i >= args.length) -1
    else if (args(i).startsWith("-")) searchFrom (i+1)
    else if (args(i).endsWith(".scala")) i
    else searchFrom(i+1)

val i = searchFrom(0)
```

在函数化编程中，使用递归函数来实现循环是非常常见的一种方法，我们应用熟悉使用递归函数的用法。

如果你实在还是希望使用 `break` ，Scala 在 `scala.util.control` 包中定义了 `break` 控制结构。它的实现是通过抛出异常给上级调用函数。如果希望使用的话，可以参考 Scala 源码。

下面给出使用 `break` 的一个例子：不停的从屏幕读取一个非空行，如果用户输入一个空行，则退出循环。

```
import scala.util.control.Breaks._
import java.io._

val in = new BufferedReader(new InputStreamReader(System.in))

breakable {
  while(true) {
    println("? ")
    if(in.readLine()=="") break
  }
}
```


四、实验总结

在本实验中，我们学习了 Scala 中的主要控制语句的用法。如果你之前学习了 Java ，那么在此处有一点需要特别注意：Scala 是一门“函数式”编程语言。结合这一点，多与之前学过的编程语言作比较，相信你能获得更多。

上一节 (/courses/490/labs/1687/document)

下一节 (/courses/490/labs/1689/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)