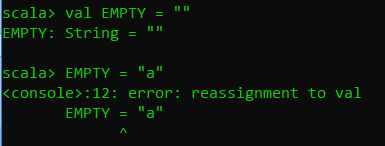
# 1. 基础

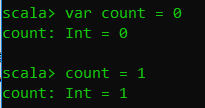
## 1.1 声明变量

以val对应的值实际上是一个常量，如果试图修改值的会报以下的错

赋值语句的类型是Uint，var x = {}；var y = 0; x = y =1;//此时表达式成立



如果要声明其值可变的变量，可以用var



指定变量类型 变量名在前，类型在后



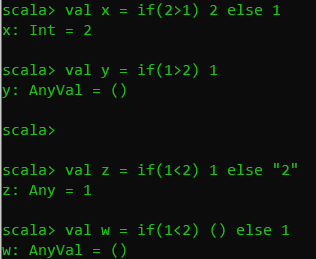
多个变量放在一起声明



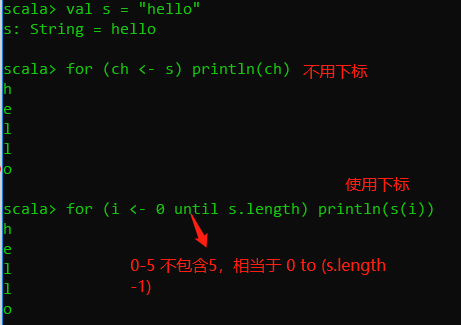
## 1.2 算术和操作符重载

scala中没有 ++，我们需要使用 +=1 或者 -=1

## 1.3 条件表达式

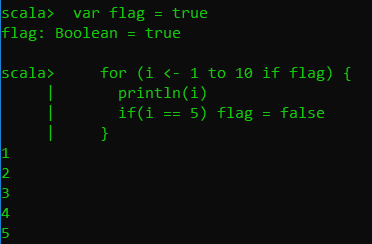


## 1.4 循环

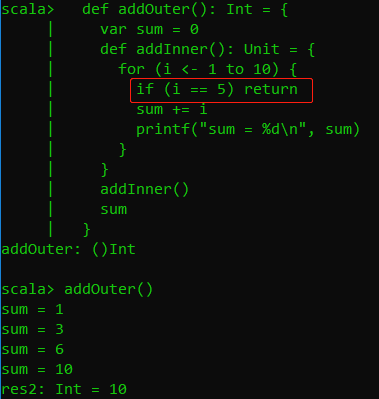


### 1.4.1 退出循环

#### 1.4.1.1 使用Boolean 控制变量



#### 1.4.1.2 使用嵌套函数以及return

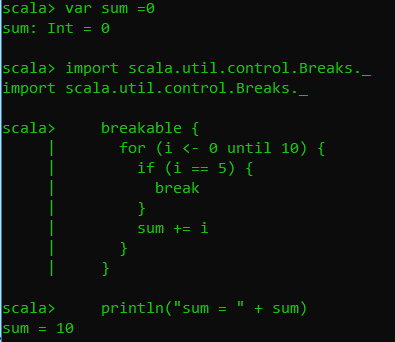


#### 1.4.1.3 使用Breaks对象的break方法

使用时需要导入这个包

import scala.util.control.Breaks.\_

注意：在这里，控制权的转移是通过抛出和捕获异常完成的，因此，如果时间很重要的话，应该尽量避免使用这套机制。

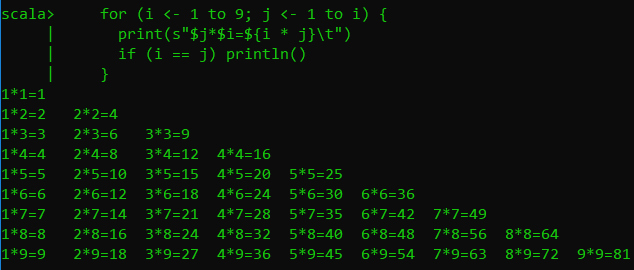


### 1.4.2 高级for循环和for推导式

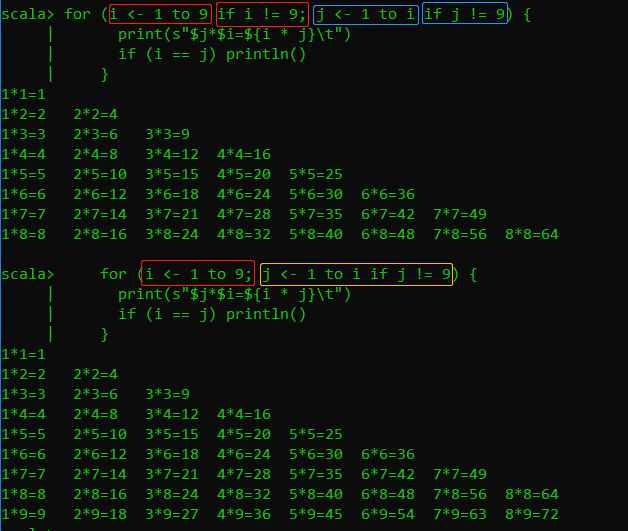
可以将生成器和守卫都定义在花括号当中，并可以以换行的方式而不是分号来隔开。

#### 1.4.2.1 可以以变量 <- 表达式的形式提供多个生成器，用分号隔开

|  |
| --- |
| **for** (i <- 1 to 9; j <- 1 to i) {  print(s"$j\*$i=${i \* j}\t")  **if** (i == j) println()  } |



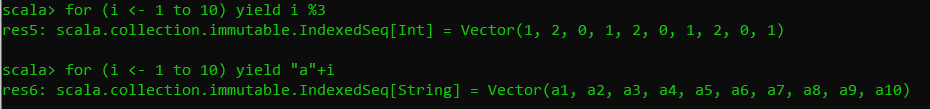
#### 1.4.2.2 每个生成器都可以带一个守卫，以if开头的Boolean表达式 注意在if前面没有分号

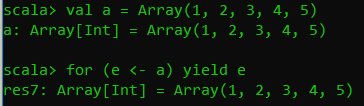


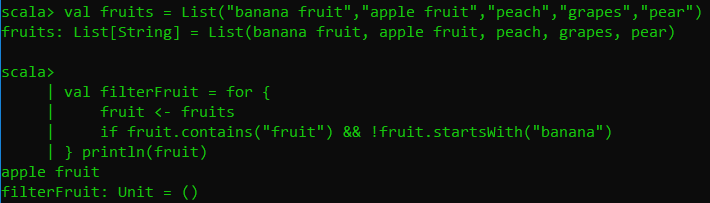
#### 1.4.2.3 可以使用任意多的定义，引入可以在循环中使用的变量

#### 1.4.2.4 for推导式

如果for循环的循环体以yield开始，则该循环会构造出一个集合，每次迭代生成集合中的一个值，类型兼容







## 1.5 函数

方法对对象进行操作，函数不是。Java中用静态方法来模拟函数。

要定义函数，需要给出函数的名称、参数和函数体

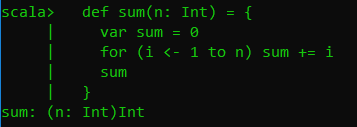
|  |
| --- |
| **def** abs(x: Double) = **if** (x >= 0) x **else** -x |



注意点：

1. 必须给出所有参数的类型。
2. 只要函数不是递归的，不需要指定返回类型，scala编译器可以通过=符号右侧的表达式的类型推断出返回类型。对于递归函数，必须指定返回类型。
3. 如果函数体需要多个表达式完成，可以用代码块。块中的最后一个表达式的值就是函数的返回值
4. 虽然在带名函数中使用return并没有什么不对，但是在匿名函数中return并不返回值给调用者，它跳出到包含它的带名函数中，我们可以把return当做是函数版的break语句，仅当需要时使用。

|  |
| --- |
| **def** sum(n: Int) = {  **var** sum = 0  **for** (i <- 1 to n) sum += i  sum  } |

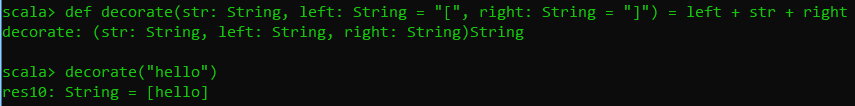


### 1.5.1 默认参数和带名参数

当我们调用某些函数时并不显式地给出所有参数值，对于这些函数我们可以使用默认参数。

|  |
| --- |
| **def** decorate(str: String, left: String = "[", right: String = "]") = left + str + right |

这个函数的两个参数（left和right），带有默认值"["和"]"



如果不喜欢默认值，也可以指定自己的参数



如果相对参数的数量，给出的值不够，默认参数会从从后往前逐个应用进来。

eg：decorate("hello", ">> [") 会使用right的默认值，得到">> [hello]"



可以在提供参数值的时候指定参数名。注意带名参数并不需要跟参数列表的顺序完全一致。带名参数可以让函数更加可读。对于有很多默认参数的函数来说很有用。也可以混用未带名参数和带名参数，只要那些未带名的参数是排在前面的即可。

eg：decorate(left = "<<<", str = "hello", right = ">>>")

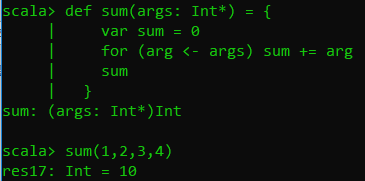




### 1.5.2 变长参数

可以使用任意多的参数来调用该函数

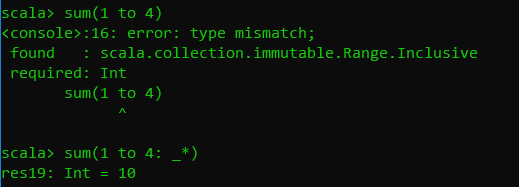
|  |
| --- |
| **def** sum(args: Int\*) = {  **var** sum = 0  **for** (arg <- args) sum += arg  sum  } |



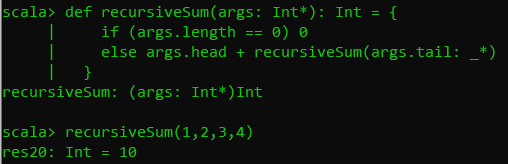
如果sum函数被调用时传入的是单个参数。那么这个参数必须是单个整数，而不是一个整数区间。我们可以把这个参数当做参数序列处理。追加 \_\* 即可。

在递归函数定义中我们会用到上述语法

|  |
| --- |
| sum(1 to 4: \_\*) |

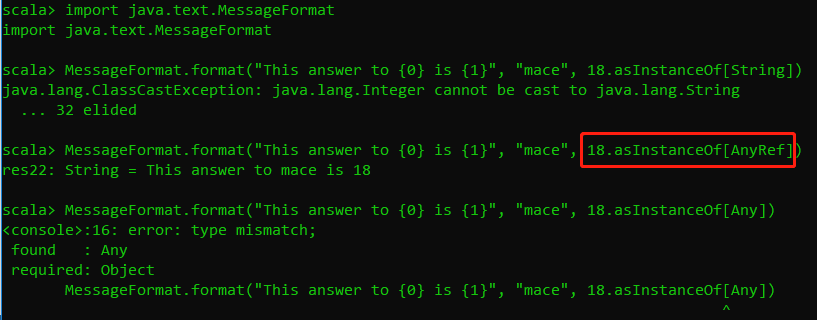


|  |
| --- |
| **def** recursiveSum(args: Int\*): Int = {  **if** (args.length == 0) 0  **else** args.head + recursiveSum(args.tail: \_\*)  } |



当调用变长参数且参数类型为object的Java方法时，需要手工对参数类型进行转换

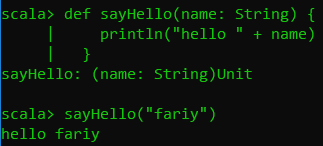
eg：18.asInstanceOf[AnyRef]



1.5.3 过程

对于不返回值的函数即返回类型为Unit时，这样的函数被称为过程。我们可以略去=号。建议显式地声明Unit返回类型

|  |
| --- |
| **def** sayHello(name: String) {  println("hello " + name)  }  **def** sayHello(name: String): Unit = {  println("hello " + name)  } |

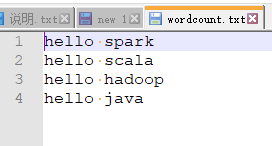


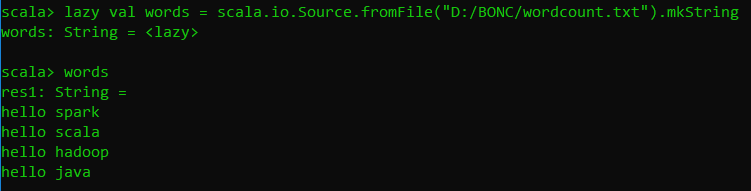
## 1.6懒值

当val被声明为lazy时，它的初始化将被推迟，直到我们首次对它取值。每次访问懒值时，都会有一个方法被调用而这个方法将会以线程安全的方式检查该值是否已被初始化。

lazy是介于val（被定义时取值）和def（每一次被使用时取值）的中间状态。

|  |
| --- |
| **lazy** **val** words = scala.io.Source.fromFile("D:/BONC/wordcount.txt").mkString |





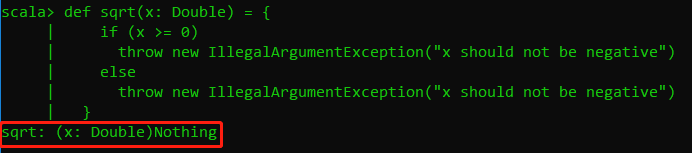
## 1.7 异常

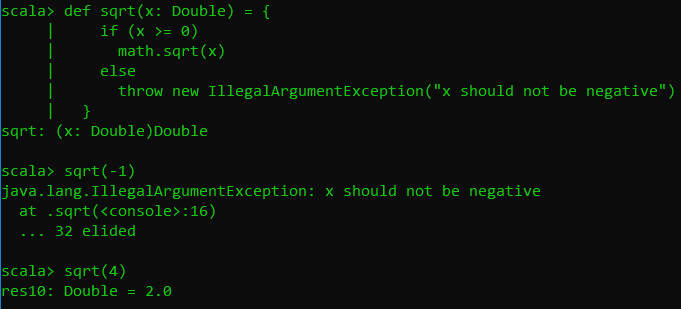
抛出异常

|  |
| --- |
| **throw** **new** IllegalArgumentException("x should not be negative") |

throw表达式有特殊的类型Nothing, 这在if/else表达式很有用。如果一个分支的类型是Nothing，那么if/else表达式的类型就是另一个分支的类型。

|  |
| --- |
| **def** sqrt(x: Double) = {  **if** (x >= 0)  math.sqrt(x)  // throw new IllegalArgumentException("x should not be negative")  **else**  **throw** **new** IllegalArgumentException("x should not be negative")  } |





捕获异常采用的是模式匹配。

|  |
| --- |
| **try** {  // 代码块  } **catch** {  // **TODO**: handle error 处理异常  **case** ex: Exception => ex.printStackTrace()  } **finally** {  // **TODO**: handle finally clause 释放资源  } |