三、函数和闭包

作者：shiweichn
原文链接：https://ld246.com/article/1533372694621
来源网站：链滴

三、函数和闭包

标签（空格分隔）： Scala学习笔记

---

1. 方法

定义函数最通用的方法是作为某个对象的成员，这种函数被称为方法。

```
def processFile(fileName: String, width: Int): Unit = {
  val source = Source.fromFile(fileName)
  for (line <- source.getLines()) {
    processLine(fileName, width, line)
  }
}

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

2. 本地函数

上面代码演示了函数式编程风格的重要设计原则：程序应该被解构成若干小的函数，每块实现一个定完备的任务，每块都非常小。 但是这样的函数太多的话就可能污染程序的命名空间。但Scala支持函中定义函数，就像定义本地变量那样定义本地函数。函数中的函数的作用域只限于其上一层函数内。

```
def processFile(fileName: String, width: Int): Unit = {
  def processLine(line: String) = {
    if (line.length > width)
      println(fileName + ": " + line.trim)
  }

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

3. 头等函数

Scala的函数是头等函数，不仅可以定义和调用，还可以写成匿名的字面量，并把它们作为值传递。

函数字面量被编译进类，并在运行期间实例化为函数值。因此函数字面量与值的区别在于函数字面量在于源代码，而函数值作为对象存在运行期。这个区别很像类和对象。

=> 左边参数，右边方法体。

函数值是对象，可以存进变量。它们也是函数，所以可以使用通常的函数调用方式来调用它们。

4. 占位符语法

把下划线当做一个或更多参数的占位符，可以让函数字面量更简洁；只要每个参数在函数字面量内仅现一次。

```
def processFile(fileName: String, width: Int): Unit = {
   def processLine(line: String) = {
     if (line.length > width)
       println(fileName + ": " + line.trim)
   }

   val lines = Source.fromFile(fileName).getLines()
   lines.foreach(processLine _)    // 占位符
 }
```

多个下划线指代多个参数，而不是单个参数的重复使用；第一个下划线代表第一个参数，以此类推。

5. 部分应用函数

尽管上面使用下划线代替了单个参数，但是还可以使用下划线来代替整个参数列表。

```
lines.foreach(processLine _)    // 占位符
```

以这种方式使用下划线时，就正在写一个部分应用函数。Scala中，当调用函数，传入任何需要的函，实际是把函数应用到参数上。

部分应用函数是一种表达式，不需要提供函数需要的所有参数。代之以提供部分，或不提供所需参数。

```
def sum(a: Int, b: Int, c: Int) = a + b + c

val a = sum _

println(a(1, 2, 3))
```

实际发生的事情是这样的：名为a 的变量指向一个函数值对象。这个函数值是由 Scala 编译器依部分应用函数表达式 sum _ ，自动生成的类的一个实例。编译器产生的类有一个 apply 方法带 3 个数。

Scala 把 a(1,2,3) 翻译成对函数值 apply 方法的调用，传入三个参数a.apply(1,2,3) ，Scala编译根据表达式 sum_ 自动生成的类里的 apply 方法，简单的把这三个缺失的参数传递到sum，并返回结。也就是在apply中调用了sum(1,2,3) ,并返回sum返回的6。

这种一个下划线代表全部参数列表的表达式的另一种用途，就是把它当做转化def为函数值的方。就像上面的，把 sum 包装成与 apply 方法具有相同的参数列表和结果类型的函数值。当把这个函值应用到某些参数上时，它一次把 sum 应用到同样的参数，并返回结果。仅管不能把方法或嵌套函赋值给变量，或当做参数传递给其它方法， 但是如果通过在名称后面加下划线的方式把方法或嵌套数包装在函数值中，就可以做到了。

现在，sum 就是一个偏函数，这个名字源自于函数未被应用于它所有的参数。在sum _里，它没有应于任何参数。不过还可以提供某些但不是全部需要的参数表达一个偏函数。如：

```
def sum(a: Int, b: Int, c: Int) = a + b + c

val b = sum(1, _, 3)

println(b(2))
```

如果你正在写一个省略所有参数的偏程序表达式，如println _ 或 sum _ ，而且在代码的那个地方确

需要一个函数，那就可以去掉下划线从而表达的更简明。如：

```
def sum(a: Int, b: Int, c: Int) = a + b + c

val b = sum _

println(b)
```

6. 闭包

```
val more = 1
val f1 = (x: Int) => x + 1
val f2 = (x: Int) => x + more
```

第三行这个函数来看，more是一个自由变量，因为函数字面量没有给出其含义，相对的，x变量是一绑定变量，因为它在函数的上下文中有声明：被定义为函数的唯一参数。

依照这个函数字面量在运行时创建的函数值(对象)被称为闭包：closure。

其实有对外部变量进行捕获的函数都叫做闭包，很显然第二行就不是个闭包。

7. 重复参数（可变参数）

```
def sum(num: Int*) = {}
```

最后一个参数类型上面加个星号，代表是可变参数。可变参数的实质就是传递一个数组，所有也可以递一个数组到方法中，但是要稍微改一下。如下：

```
sum(arr: _*)
```

8. 尾递归

尾递归：递归调用中函数体执行的最后一件事是调用自己本身。Scala可对尾递归进行优化，Scala编器编译器检测到尾递归就用新值更新函数参数，然后把它替换成一个会到函数开头的跳转。但是Scal里尾递归的使用局限性也很大，如果递归是间接的或者最后一个调用是一个函数值，那么都不能获得递归优化。因此,尾递归优化限定了方法或嵌套函数必须在最后一个操作调用本身，而不是转到某个函值或什么其它的中间函数的情况。