这章包括

- 使用 Scala REPL(读取-求值-打印-循环)
- Scala 基础
- For 推导 (For-comprehensions)和模式匹配
- 用 Scala 构建一个完整的 REST 客户端

在第1章你学习了什么是Scala及它与其它的编程语言相比如何.我以一个较高层次描述了这个概念,在下面的章节中你将重新回顾这些概念,使用示例更详细的探索它们.在这章你将学习基础的Scala类型,函数,for推导,模式匹配,和其它常见的语法(suspects).

这章的某些概念,比如模式匹配和for推导,是函数式编程的概念.但是因为Scala是一个混合模式的语言,我介绍他们其它的非函数式编程概念.这章的目标是让你熟悉Scala语言的基础和它的运行环境,这样你就可以开始使用Scala编程了.在开始阶段,我只用Scala解释器和它的REPL(读取-求值-打印-循环)环境(你将在下一节看到),为了保持事情的简单性.

在进一步开始之前,保证你安装的Scala是可以工作的.这章自始至终你将试验各种Scala示例,同时如果你在Scala解释器中试验它们,这将会更好.现在让我们愉快的体验下Scala解释器.

2.1 使用Scala解释器的REPL

开始使用Scala的简单的方式就是使用Scala解释器,一个用于编写Scala表达式和程序的交互式shell.为了以交互式模式开始Scala解释器,在命令提示符中输入scala.如果你安装的没有问题,你将看到如下的一些东西:

```
1.  Welcome to Scala version 2.10.0.final (Java ...).
2.  Type in expressions to have them evaluated.
3.  Type :help for more information.
4.  scala>
```

这意味着你的Scala安装是成功的.我运行的是 Scala version 2.10.0,所有的代码示例用这个版本都可以工作.在Scala提示符中输入42然后回车,你将看到下面的东西:

```
1.  scala> 42
2.  res0: Int = 42
```

第一行就是你输入的内容.Scala解释器读取输入的42,将它解释为一个整型字面量,创建一个代表数字42的 `Int` 类型对象,然后将它打印到控制台, `res0` 是由Scala解释器创建的变量名称(变量的名称对你来说可能是不同的,因为变量名称是由Scala解释器在运行时生成的),它保存了值42.如果你输入变量名称,在这种情况下是 `res0` , 在提示符中,你将得到类似的输出:

```
1.  scala> res0
2.  res1: Int = 42
```

这些步骤加起来称为读取-求值-打印-循环(read-evaluate-print loop-REPL).你可以在Scala解释器中重复循环 read-evaluate-print 步骤.现在你将在Scala中编写你的第一个 "Hello world"程序:

```
1.  scala> println("Hello world")
2.  Hello world
```

你将判定println函数通过传递"Hello world"字符串作为一个参数,Scala输入相同的字符串.

说明(DEFINITION) `println` 是定义在 `scala.Console` 中的函数, 依次使用 `System.out.println`

打印消息到控制台.Scala `Predef`(部分的标准库)为你将 `println` 映射到了 `Console.println`,所以这样当你使用的时候你不用使用 `Console` 作为前缀.

在第一章,我提到了 Scala 可以和 Java结合,但是没有提供一个示例.我在这里补充:

```
1.  scala> val myList = new java.util.ArrayList[String]()
2.  myList: java.util.ArrayList[String] = []
```

在这个例子中你创建了一个 `java.util.ArrayList` 的实例,保存了 `String` 类型对象.如果你记不得所有的方法,你可以在 myList 上调用,不要担心,因为Scala解释器将帮助你完成.输入myList,后面跟着一个点号,然后按下Tab;Scala解释器将列出你可以调用的所有方法.它不仅仅可以列出与一个数据类型相关的方法,也可以自动完成对解释器已知的变量名称和类名.我鼓励你花些时间在Scala解释器上面,探索下可用的选项,在本书中当使用示例的时候这将很方便.考虑到REPL作为学习一门新的语言的基础部分,因为它给出了快速的反馈.表2.1说明了对你来说可用的REPL选项.

补充:列出类的方法

```
scala> myList.
add             addAll          asInstanceOf    clear           clone           contains        containsAll
ensureCapacity  get             indexOf         isEmpty         isInstanceOf    iterator        lastIndexOf
listIterator    remove          removeAll       retainAll       set             size            subList
toArray         toString        trimToSize
```

表2.1 重要的Scala解释器命令

命令	描述
:help	这条命令打印在Scala解释器中可用的所有命令行的帮助信息
:cp	使用这条命令为Scala解释器将一个JAR文件加入到classpath中.比如, `cp tools/junit.jar` 将尝试找到一个相对于你的当前位置的JUnit JAR文件,如果找到的话,它将把JAR文件加入到classpath中,这样你将可以引用这个JAR文件中的类.
:load 或:l	运行你加载一个Scala文件到解释器中.如果你想研究已经存在的Scala代码,你可以把文件加载到Scala解释器中,所有的定义(definitions)对你来说将是可得的.
:replay 或 :r	重置解释器,然后回放(replays)所有之前的命令.
:quit 或 :q	退出解释器.
:type	显示一个表达式的类型而不用计算.比如, `:type+2` 将决定表达式的类型为 `Int` 而不用执行相加的操作.

2.2 Scala 基础

在这节我将使用示例完成基础的Scala信息,这样你可以逐渐熟悉它.你将使用 Scala REPL去尝试示例,但是你可以使用任意的在之前章节提到的适合你的开发环境.

在下面的小节中,你将探索基础的Scala类型,包括 `String` 和值类型 `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Boolean`, 和 `Char`.你将学习两种类型的Scala变量, `var` 和 `val`,如何使用它们,如何定义它们,和你可以调用它们的方式.我们从Scala的基础数据类型开始.

2.2.1 基本类型

如果你是一个Java程序员,你将会非常高兴,知道Scala支持所有的基本的值类型(基本类型): `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Boolean`, 和 `Char`(原文中缺少了 `Long`).表2.2展示了Scala支持的所有的八种基本的值类型.在Scala中所有的基本类型都是对象,它们定义在 `scala` 包中.

表2.2 Scala基本类型

值类型	描述和范围
Byte	8-bit有符号的2进制补码.它的最小值为-128,最大值为127(包括)
Short	16-bit有符号的2进制补码.它的最小值为-32,768,最大值为32,767(包括)
Int	32-bit有符号的2进制补码.它的最小值为-2,147,483,648,最大值为2,147,483,6477(包括)
Long	32-bit有符号的2进制补码.它的最小值为-9,223,372,036,854,775,808, 最大值9,223,372,036,854,775,80777(包括)
Float	单精度的32-bit IEEE 754 浮点型数
Double	单精度的64-bit IEEE 754 浮点型数
Boolean	两个可能的值：true和false
Char	一个16位Unicode字符.最大值为 "\u0000"(或0), 最大值为 "\uffff"(或65,535,包括)

关于 Scala Predef 的一个小事实

Scala 编译器隐式地为每个编译单元或者Scala程序导入 java.lang, scala 包和称为 scala.Predef 的对象.在 .NET 中,不是导入 java.lang, 而是导入 system 包. Predef 对象为 Scala 定义了标准的函数和类型别名.因为这个对象是被自动导入的,这个对象的所有成员对你都是可用的. Predef 是很有趣的,你可以通过 scaladoc 或 scala.Predef 对象的源码了解学习它.

在Scala中所有的基本类型声明的时候都是第一个字母大写.在Scala中它被声明为 `Int` , 而不是 `int` .在Scala的早期版本中,程序可以使用小写和大写字母交换,但是从版本2.8开始,如果你使用 `int` 声明任意的变量,你将得到一个编译错误:

```
scala> val x:int = 1
<console>:4: error: not found: type
int
val x:int = 1
```

下面的是可以编译的:

```
scala> val x:Int = 1
x: Int = 1
```

即使 `Int` 的全限定名是 `scala.Int` , 你可以只使用 `Int` ,因为 `scala` 包被自动导入到 Scala 源码中,所以你没有必要用全限定的基本类型.
为了看下所有的包都是自动导入的,使用 `:imports` 命令:

```
scala> :imports
 1) import java.lang._             (153 types, 158 terms)
```

```
3.    2) import scala._                (798 types, 806 terms)
4.    3) import scala.Predef._         (16 types, 167 terms, 96 are implicit)
```

在这种情况中， `java.lang` ， `scala` ，和 `scala.Predef` 包是自动导入的,当你开始一个REPL会话.

```
1.    scala> val decimal = 11235
2.    decimal: Int = 11235
```

说明 一个字面量是描述一个对象的速记法.标准的表达式匹配项目的结构.你可以通过使用字符串字面量 "one" 创建一个字符串对象,然后也可以使用 `new` 关键字，如 `new String("one")` .

因为整型字面量通常是整型,Scala推断类型为整型,但是如果你想使用一个 `Long` 类型,你可以加上后缀 `L` 或 `l` :

```
1.    scala> val decimal = 11235L
2.    decimal: Long = 11235
```

十六进制数以 `0x` 开始,八进制数以前缀 `0` 开始:

```
1.    scala> val hexa = 0x23
2.    hexa: Int = 35
3.    scala> val octa = 023
4.    hexa: Int = 19
```

一个要注意的事情是当使用十六进制和八进制数的时候,Scala解释器总是将结果计算为十进制数.你可以指定你想要的变量的类型(有时候只称作值(value);稍后详述),当你认为Scala类型结果不会产生你想要的结果的时候.如果你使用一个整型字面量声明一个变量,比如,Scala创建一个整型类型的变量除非它不能满足整型值的范围.在那种情况下,Scala编译器抛出一个错误.你会怎么做如果你想要一个 `Byte` 类型变量?你声明类型 `Byte` 当创建变量的时候:

```
1.    scala> val i  = 1
2.    i: Int = 1
3.    scala> val i2: Byte = 1
4.    i2: Byte = 1
```

浮点型字面量

浮点型字面是由一个数字使用小数点和指数组成的.但是小数点和指数部分的可选的.浮点字面量是 `Float` 类型的字面量当他们使用前缀 `F` 或 `f` 的时候,否则是 `Double` :

```
1.    scala> val d  = 0.0
2.    d: Double = 0.0
3.    scala> val f = 0.0f
4.    f: Float = 0.0
```

你也可以使用一个指数部分创建一个 `Double` 变量.为了声明 1 乘以 10的30次幂(1 times 10^30)的值的变量,它看上去像这样:

```
1.    scala>val exponent = 1e30
2.    exponent: Double = 1.0E30
```

使用浮点字面量的特殊情况

在浮点字面量中你可以定义 Double 值为 1.0 或 1,而不用后面加上 0 或数字.在 Scala 中你可以使用一个小数点(.)调用一个方法,后面跟着方法的名称,因为所有的基本类型(原始类型)在 Scala 中是对象,和所有其他的 Scala 对象一样都有一个 toString 方法.这引发了一个有趣的特殊情况当在浮点字面量上调用方法的时候.为了在 1. 上调用一个 toString 方法,你该怎么做?你必须在点和 toString 方法之间放一个空格,像这样: 1. toString().如果你尝试使用 1.toString 而没有空格,它调用定义在 Int 对象中的 toString 方法.这个是必须的只有当方法名称以字母开头的时候.比如,1.+1 工作正常,产生预期的输出, 2.0.

字符常量

一个字符常量是被引号引起来的单个字符.字符可以是一个可打印的Unicode字符或一个转义序列:

```
scala> val capB = '\102'
capB: Char = B
scala> val capB = 'B'
capB: Char = B
```

你也可以分配一个特殊的字符字面量转义序列给 `Char` 类型的变量:

```
scala> val new_line = '\n'
new_line: Char =
```

因为 `new_line` 字符不像其它的字符, 是不可用打印的, `new_line` 变量的值不会在Scala解释器中显示.所有的字符转义在Java和.NET中都被支持.Scala把使用Unicode字符的编程带到了下一个水平阶段(Scala takes programming with Unicode characters to the next level).你不仅可以使用字面量也可以使用可打印的Unicode字符作为变量和方法的名称.为了使用Unicode字符创建一个名为 `ans` 值为 42的变量,你可以像下面这么做:

```
scala> val \u0061\u006e\u0073 = 42
ans: Int = 42
```

使用Unicode字符在程序中用于命名变量和函数是一种会被你的同行对你大喊大叫的方式,但是在某些环境中它提高了可读性.在下面的示例中Unicode字符被用在变量和方法名称中:

```
val ? = scala.math.Pi
def ?(x:Double) = scala.math.sqrt(x)
```

在你尝试这些示例之前,保证你的编辑器支持Unicode编码.

字符串字面量

一个字符串字面量是在双引号中的一个字符序列.字符是可以打印的字符或转义字符.如果字符串字面量包含了一个双引号,它必须使用一个反斜杠(`\`)转移:

```
scala> val bookName = "Scala in \"Action\""
bookName: java.lang.String = Scala in "Action"
```

字符串字面量的值是 `String` 类型.不像其它的基本类型, `String` 是 `java.lang.String` 的一个实例.如之前提到的,它被自动导入.Scala也支持一个特殊的多行字符串字面量,被附在了三重引号中(`"""`).字符

串序列是任意的,除了它不能包含一个三重引号外,另外它甚至不一定是可打印的:

```
scala> val multiLine = """This is a
     |                    multi line
     |                    string"""
multiLine: java.lang.String =
This is a
            multi line
            string
```

多行变量的输出有前导的空白,可能不是你想要的.有一种简单的方式修复-调用一个称为 `stripMargin` 的方法去除前导空白.

```
scala> val multiLine = """This is a
     | |multi line
     | |string""".stripMargin
multiLine: String =
This is a
multi line
string
```

这个代码看起来有点混乱.第一个 | (竖线) 或边缘的空白(margin)是由解释器加上的当你按下回车而不用完成表达式,第二个是你添加作为一个margin用于多行的字符串.当 `stripMargin` 方法找到这些 margin 字符,它去除前导的空白.我发现多行字符串非常有用当创建数据集用于单元测试的时候.

字符串插值(String interpolation)

Scala 2.10 已经支持字符串插值.你可以使用这种特性像这样:

```
scala> val name = "Nilanjan"
name: String = Nilanjan
scala> s"My name $name" # 前面的 s 可以不要
res0: String = My name Nilanjan
```

这里是一个在类 `StringContext` 上的方法调用通过传递包含在双引号中的字符串字面量.任何以 `$` 为前缀的token或使用 `${...}` 包装的字符串将被替换为相应的值.同样的预加 `f` 到任意的字符串字面量,允许简单的格式化字符串的创建.类似于其它语言中的 `printf`:

```
scala> val height = 1.9d
height: Double = 1.9
scala> val name = "James"
name: String = James
scala> println(f"$name%s is $height%2.2f meters tall") #
前面必须要有 f
James is 1.90 meters tall
```

作为一个细心的读者,你可能会对这个 `stripMargin` 方法感到有点惊奇,因为我说过一个Scala `String` 对象什么都不是但是代表了一个 `java.lang.String` 对象;我们是在哪里取得这个新的 `stripMargin` 方法的?在 `java.lang.String` 中没有一个称为 `stripMargin` 的方法.再次说明, `Predef` 施了一点魔法,通过将 `java.lang.String` 包装为另一种类型,称为 `scala.collection.immutable.StringLike` .如果你查看Scala文档,你将看到Scala提供了 `stripMargin` 方法和许多其它有用的方法给字符串对象,连同定义在

`java.lang.String` 类中的方法.

RichString vs. StringLike
如果你使用过 Scala 的以前的版本,你将回忆起一个早期的类称为 scala.RichString,向 Scala 字符串对象提供了额外的方法,但是从 Scala 2.8 开始,它被称为 scala.collection.immutable.StringLike.将一个字符串作为一个不变的集合更有意义,因为它是字符的一个集合,并且一个字符串是一个不变的对象.Scala 仍然有其它基本类型的 Rich 类型的包装,像 RichInt, RichBoolean, RichDouble,等.

XML字面量

通常,使用XML意味着使用第三方的解析器和库,但是在Scala中,它是语言的一部分.Scala支持XML字面量,你可以将XML片段作为代码的一部分:

```
1.  val book = <book>
2.             <title>Scala in Action</title>
3.             <author>Nilanjan Raychaudhuri</author>
4.             </book>
```

当你在Scala解释器中输入这个表达式,你将得到以下输出:

```
1.  book: scala.xml.Elem =
2.      <book>
3.         <title>Scala in Action</title>
4.         <author>Nilanjan Raychaudhuri</author>
5.    </book>
```

Scala将XML字面量转换为一个 `scala.xml.Elem` 类型的对象.这里还没有结束.你可以将有效的Scala代码放到花括号中, `{}` , 里面是XML标记,它工作的很好当你需要动态生成XML的时候:

```
1.  scala> val message = "I didn't know xml could be so much fun"
2.  scala> val code = "1"
3.  scala> val alert = <alert>
4.              <message priority={code}>{message}</message>
5.              <date>{new java.util.Date()}</date>
6.              </alert>
7.  alert: scala.xml.Elem =
8.  <alert>
9.         <message priority="1">
10.             I didn't know xml could be so much fun
11.          </message>
12.         <date>Fri Feb 19 19:18:08 EST 2010</date>
13.   </alert>
```

如你看到的,Scala执行花括号中的代码,用输出的代码替换它.在花括号中定义的代码被称为 *Scala 代码块 (Scala code blocks)*.

当使用Scala代码生成属性值,保证你不要使用双引号(priority={code}),因为如果你这样做,Scala将忽略它,并将它作为一个字符串值.本书自始至终你将看到XML字面量的各种使用和Scala支持的其它的XML东西.

2.2.2 定义变量

你已经看到许多关于定义变量示例.在Scala中有两种方式定义变量: `val` 和 `var` . `val` 是单次分配变量,有时候称为 `值(value)` .一旦初始化, 一个 `val` 不能被改变或重新分配给其它的值(类似于Java中的 `final` 变量).另一方面, `var` 是可重新分配的;你可以通过一次又一次的改变变量的值在初始化分配后:

```
scala> val constant = 87
constant: Int = 87
scala> constant = 88
<console>:5: error: reassignment to val
       constant = 88
                ^
scala> var variable = 87
variable: Int = 87
scala> variable = 88
variable: Int = 88
```

Scala解释器在基于值推断变量类型上做了很好的工作,但是有时候你喜欢,需要,指定类型,你可以在变量名称后面指定变量的类型,通过冒号(`:`)分隔.
存在一些情况你想声明一个没有分配值的变量,因为你也不知道值.这种情况下你可以使用Scala占位符(`_`)分配一个默认的值:

```
scala> var willKnowLater:String = _
willKnowLater: String = null
```

Scala解释器在基于值推断变量类型上做了很好的工作,但是有时候你喜欢,需要,指定类型,你可以在变量名称后面指定变量的类型,通过冒号(`:`)分隔.
存在一些情况你想声明一个没有分配值的变量,因为你也不知道值.这种情况下你可以使用Scala占位符(`_`)分配一个默认的值:

```
scala> var willKnowLater:String = _
willKnowLater: String = null

// 补充:
scala> var willKnowLater:String
<console>:7: error: only classes can have declared but undefined members
(Note that variables need to be initialized to be defined)
       var willKnowLater:String
           ^

scala> var willKnowLater:String  =
     | _
willKnowLater: String = null

scala> var willKnowLater
     |
     |
You typed two blank lines.  Starting a new command.
```

因为String的默认值为null,在这个示例中, `willKnowLater` 的值为 null.作为一个练习,使用其它基本类型的时候尝试使用Scala占位符,然后看下你将得到什么值.要注意的一点是何时声明变量(`val` 和 `var`),你需要指定值或 `_` (Scala占位符);否则,Scala解释器将抱怨.你可以有一个没有分配值的变量的唯一情况是当变量(只能是 `var` ,因为 `val` 总是需要一个值当被声明的时候)被声明在一个类的内部.
有时候你想声明一个类型,它的值基于某些耗时(time-consuming)操作被计算,你不想那么做当你声明变量的时候;你想延迟初始化它因为默认情况下Scala计算分配给 `var` 或 `val` 的值当它被声明的时候.为了改变这种默认行为,使用 `lazy val` :

```
scala> lazy val forLater = someTimeConsumingOperation()
forLater: Unit = <lazy>
```

`someTimeConsumingOperation()` 将被调用当变量 `forLater` 被任意表达式使用的时候.这里是另一个例子,说明了延迟性:

```
scala> var a = 1
a: Int = 1
scala> lazy val b = a + 1
b: Int = <lazy>
scala> a = 5
a: Int = 5
scala> b
res1: Int = 6
```

在最后一行,输入 `b` 强制计算b,因为它没有被计算当 `b` 被声明的时候,它使用 `a` 最后的值. `lazy` 关键字只允许和 `val` 使用;在Scala中不能声明 `lazy var` 变量.
变量声明可能有时候有一左边模式(a pattern on the left side).也就是你想提取一个 `List` 的第一个元素然后分配给一个变量.你可以使用左边模式这样做,并进行变量分配:

```
scala> val first :: rest = List(1, 2, 3)
first: Int = 1
rest: List[Int] = List(2, 3)
```

Scala中的 `List` 是一个不可变的集合类型(类似于Java和C#中的 `List`),这上面的情况中它保存了从1到3的数字集合.左边模式匹配了 `List` 的第一个元素,在这种情况是1分配给变量 `first` , `rest` 为列表的尾部2和3. `::` (称为 cons)是定义在 `List` 中的方法.在本章的后面我还将说明关于模式匹配的更多内容.
之前是将参数作为不可变的以及为什么你应该总是优先使用不可变性.牢牢记住这个,在Scala中总是以 `val` 开始当声明变量的时候,改变为 `var` 当完全必要的时候.

2.2.3 定义函数

函数在Scala中以块构建,在这节,你将准备探索这个主题.在Scala中为了定义一个函数,使用 `def` 关键字,然后是方法的名称,参数,可选的返回类型,=和方法体.图2.1 展示了Scala函数声明的语法.
使用一个冒号(`:`)分隔参数列表和返回类型.多个参数用逗号(`,`)分隔.等于号(`=`)作为方法签名和方法体的分隔符.
现在让我们先丢弃参数;稍后将会回到参数上来.你将不使用参数创建你的第一个Scala函数:

```
scala> def myFirstMethod():String = { "exciting times ahead" }
myFirstMethod: ()String
```

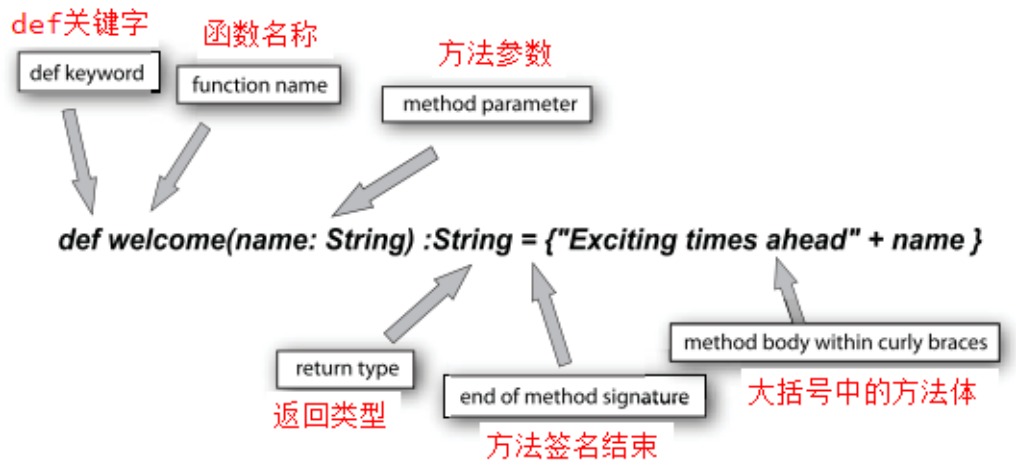

Figure 2.1 **The syntax of the Scala function declaration**

Scala函数声明的语法

Scala函数的返回类型是可选的,因为Scala可自动推断一个函数的返回类型.不过有一些情况不行,但是不要担心,后面将会说明.通过移除返回类型改进 `myFirstMethod` 方法:

```
1.  scala> def myFirstMethod() = { "exciting times ahead" }
2.  myFirstMethod: ()java.lang.String
3.  scala> myFirstMethod()
4.  res6: java.lang.String = exciting times ahead
```

方法签名后面的 `=` 签名不仅仅只是用来分离方法体的签名,同样告诉Scala编译器去推断你的函数的返回类型.如果你忽略它,Scala不会推断你的返回类型:

```
1.  scala> def myFirstMethod(){ "exciting times ahead" }
2.  myFirstMethod: ()Unit
3.  scala> myFirstMethod()
```

在这种情况下,当你使用函数名称和 ()调用函数,你得不到结果.在REPL输出中,注意你的函数的返回类型不再是 `java.lang.String` ;它是 `Unit` .Scala中的 `Unit` 像Java中的 `void` ,这意味着方法不会返回任何东西.

小技巧 Scala类型推断是强大的,但是要小心使用.如果你创建一个库,打算暴露你的函数作为公共API,为用户指定库的返回类型是一个好的实践.在任何情况下,如果你认为函数的返回类型是不明确的,尝试改进函数名称,这样它传达其目的会更好,或指定返回类型.

你的 `myFirstMethod` 是很简单的:它返回字符串 "exciting times ahead",当你有一个像这样的函数,你也可以从方法体中去除大括号:

```
1.  scala> def myFirstMethod() = "exciting times ahead"
2.  myFirstMethod: ()java.lang.String
```

如果你调用函数,你将得到相同的结果.在Scala中,总是可以从代码中去掉不必要的语法干扰.因为你没有传递任何参数,你可以从声明中去除未使用的 `()` ,这样它几乎看起来看一个变量声明,除了你使用的是 `def` 而不是 `var` 或 `val` :

```
1.  scala> def myFirstMethod = "exciting times ahead"
2.  myFirstMethod: java.lang.String
```

当你调用函数的时候,你也可以不用圆括号:

```
scala> myFirstMethod
res17: java.lang.String = exciting times ahead
```

如果函数有副作用(side effects),常用的约定是使用 `()` 即使它不是必须的.
回到函数的参数,下面的函数称为 `max` , 有两个参数,返回两个当中较大的一个:

```
scala> def max(a: Int, b: Int) = if(a > b) a else b
max: (a: Int,b: Int)Int
scala> max(5, 4)
res8: Int = 5
scala> max(5, 7)
res9: Int = 7
```

现在你可能已经理解指定返回类型在Scala中是可选的.你不必指定 `return` 关键从函数中返回什么.它将返回最后的表达式的值.在之前的例子中,如果条件计算为 `true`, `a` 是被执行的最后的表达式,所以 `a` 被返回;否则 `b` 被返回.即使返回类型是可选的,但是你必须指定参数的类型当定义函数的时候.Scala类型推导将会理解参数的类型当你调用函数的时候,但不是在函数声明的期间.

类型推导
如果你有 Haskel,Ocaml,或者任意其它的推断类型编程语言的背景,你会觉得 Scala 参数定义的方式有点怪.原因是 Scala 没有使用 Hindley-Milner 算法进行类型推断;取而代之的是 Scala 的类型推导是基于声明-局部(declaration-local)信息,也称为本地类型推导(local type interence).类型推导超出了本书的范围,但是你如果你感兴趣,你也看的关于 Hindley-Milner 类型推导的算法,了解它为什么这么有用.

有时候有必要创建一个有一个输入的函数,并从它创建一个 `List` .但是问题是你不能决定输入的类型.有人可能使用你的函数去创建 `Int` 类型的 `List` ,另外还有人可能使用它创建 `String` 类型的 `List` .在这种情况下,在Scala中你可以通过参数化(parameterized)类型创建一个函数.参数类型将在你调用的时候被决定:

```
scala> def toList[A](value:A) = List(value)
toList: [A](value: A)List[A]
scala> toList(1)
res16: List[Int] = List(1)
scala> toList("Scala rocks")
res15: List[java.lang.String] = List(Scala rocks)
```

当生命函数的时候,你指示未知的参数化类型为 `A` .现在当你的 `toList` 被调用,它使用给定的参数的参数类型替换 `A` .在方法体中你通过传递参数创建了一个不可变的 `List` ,从REPL的输出中它很清楚 `List` 同样使用参数化的类型.

注意 如果你是一个Java程序员,你将发现Java泛型和Scala参数化类型之间的许多相似之处.目前要记住的唯一的差异是Java使用尖角括号(`<>`),而Scala使用方括号(`[]`).另一种Scala约定用于命名参数化类型是如有必要从 `A` 开始,直到 `Z` .这个和Java约定使用 `T` , `K` , `V` 和 `E` 形成对比.

函数字面量

在Scala中你也可以传递一个函数作为参数给另一个函数,大多数时候在这些情况中,我提供一个函数的内联定义(inline definition).这种将传递函数作为参数有时候也被很宽松的称为 `闭包(closure)` (传递一个函数不总是一定要闭包(closure),你将在第4章中看到).Scala提供了一种速记的方式创建一个函数,你只需要写函数体,这被称为 `函数字面量` .将那个放到一个test中.在这个test中你想使用函数字面量添加一个 `List` 的所有元素.这里展示了一种在Scala中函数字面量的一个简单使用.

这里你创建了偶数的一个 `List`：

```
1.  scala> val evenNumbers = List(2, 4, 6, 8, 10)
2.  evenNumbers: List[Int] = List(2, 4, 6, 8, 10)
```

为了添加 `List`(scala.collection.immutable.List) 的所有元素,你可以使用定义在 `List` 中的 `foldLeft` 方法. `foldLeft` 方法有两个参数：一个初始值和一个二元操作.它把二元操作应用到给定的初始值和 `List` 的所有元素.它期望二元操作为一个函数有两个参数,并自己执行操作,在我们的例子中是相加.如果你创建一个函数有两个参数,并让它们相加.你可以使用这个test完成. `foldLeft` 函数将为 `List` 中的每个元素调用你的函数,从初始值开始：

```
1.  scala> evenNumbers.foldLeft(0) { (a: Int, b:Int) => a + b }
2.  res19: Int = 30
```

在这种例子中函数 `(a: Int, b:Int) => a + b` 被称为 *匿名* 函数,或者一个没有预定义名称的函数.你可以通过高级的Scala类型推导改进你的函数：

```
1.  scala> evenNumbers.foldLeft(0) { (a, b) => a + b }
2.  res20: Int = 30
```

通常你需要为顶层的(top-level)函数指定参数的类型,因为Scala不会推断参数的类型当被声明的时候,但是对于匿名函数Scala推断可以根据上下文理解类型.在这个例子中你正在使用一个整型的列表,0作为你的初始值,基于这些Scala可以很容易推断 `a` 和 `b` 的类型为整型.Scala允许你更进一步的使用匿名函数：你可以去除参数,只要保留函数体让它作为一个函数字面量.但是在这个例子中参数将使用下划线(`_`)替换.在Scala中下划线有一个特殊的意义,在这个上下文中它是一个参数的占位符;在你的例子中,使用两个下划线：

```
1.  scala> evenNumbers.foldLeft(0) { _ + _ }
2.  res21: Int = 30
```

在你的函数字面量中每个下划线代表一个参数.你已经看到了下划线的另一种使用方式当分配一个默认值给变量的时候.在Scala中你可以在各种位置使用下划线,它们的意思通过上下文和它们在哪里被使用被唯一的决定.有时候它也有点让人困扰,所以总是要记住下划线的值基于它在哪里被使用.本书中你将看到下划线的其它使用方式.函数字面量在Scala中有一种常见的语法,在Scala库和代码基中你可以找到它们.
在第1章你看到了下面的没有足够解释要干什么的例子.现在,使用你对函数字面量的认识,它可能是相当明显, `_.isUpper` 是一个函数字面量：

```
1.  val hasUpperCase = name.exists(_.isUpper)
```

在这个例子中你为在 `name` 字符串中的每个字符调用给定的函数字面量;当它找到一个大写字符,它将退出.在这个上下文中下划线代表了 `name` 字符串的一个字符.

使用Scala闭包和一等函数(first-class functions)：一个例子

在阅读下一节之前,离开"函数定义"小结而没有使用一个闭包的一个简单的工作示例是不公平的.闭包就是任意的函数关闭它所定义的环境.比如,闭包将持续追踪函数之外的,被引用到函数里的任意变量的改变.
在这个例子中你尝试去添加 `break` 的支持.我还没有讨论过关于Scala关键字的内容,但是Scala没有 `break` 或 `continue` .一旦你熟悉了Scala,你就不会怀念它们因为Scala的函数式编程风格的支持减少了 `break` 或 `contiune` 的需求.但是假设你有一种情况你认为 `break` 可能更有用.Scala是一个可扩展的编程语言,所以你可以扩展它去支持 `break` .
在Scala中使用Scala 异常-处理 机制去实现 `break` .抛出异常将帮助你中断执行的序列, `catch` 块将帮助你抵达调用的末端.因为 `break` 不是一个关键字,你可以使用它去定义你自己的函数,这样将抛出一个异常：

```
def break = new RuntimeException("break exception")
```

另一还没有讨论的主题是异常处理,但是如果你已经在Java,C#或Ruby中使用,它应该很容易理解.无论如何,在本章的后面的部分你将阅读关于异常处理的内容.现在创建主函数,这个操作需要一个可中断的特性.显而易见,称它为 `breakable` :

```
def breakable(op: => Unit) { ... }
```

`op: => Unit` 是什么?特殊的右箭头键(`=>`)让Scala知道 `breakable` 函数期望一个函数作为一个参数. `=>` 的右边定义了函数的返回类型-在这个例子中,它是 `Unit` (类似于Java的void)- `op` 是参数的名称.因为你在箭头的左边没有指定任何东西,这意味着你期望函数作为一个参数,而它自己没有任何参数.但是如果你期望一个函数参数是两个参数,比如 `foldLeft` , 你需要像下面这样的定义:

```
def foldLeft(initialValue: Int, operator: (Int, Int) => Int)= { ... }
```

`breakable` 函数被声明为一个没有参数的函数,返回 `Unit` .现在,使用这两个函数,你可以模拟 `break` .让我们看一个示例函数,它需要中断当环境变量 `SCALA_HOME` 没有被设置.否则,它必须工作:

```
def install = {
  val env = System.getenv("SCALA_HOME")
  if(env == null) break
  println("found scala home lets do the real work")
}
```

现在在 `breakable` 函数内我们需要捕获它将引发的异常,当 `break` 从 `install` 函数被调用:

```
try {
    op
} catch { case _ => }
```

下面的清单是完整的代码:

清单 2.1 breakable,break 和 install 函数

```
val breakException = new RuntimeException("break exception")
def breakable(op: => Unit) {
  try {
    op
  } catch { case _ => }
}

def break = throw breakException
def install = {
  val env = System.getenv("SCALA_HOME")
  if(env == null) break
  println("found scala home lets do the real work")
}
```

为了调用 `breakable` 函数,传递的方法名称需要 `breakable` 特性,比如 `breakable(install)` 或你可以内联 `install` 函数,然后将它作为一个闭包传递:

```
breakable {
  val env = System.getenv("SCALA_HOME")
```

```
    if(env == null) break
    println("found scala home lets do the real work")
  }
```

在Scala中如果一个函数的最后一个参数是函数类型,你可以将它作为一个闭包传递.这种语法在DSLs中非常有用.在下一章你将深入闭包如何被转换为对象;记住,在Scala中一切都是对象.

注意 Scala已经提供了 `breakable` 作为库的一部分.查看 `scala.util.control.Breaks` .你应该在你需要 `break` 的时候使用 `Breaks` .再次说明,我认为一旦你深入了Scala函数式编程的细节,你将可能永远不需要 `break` .

2.3 使用 Array 和 List

第4章专门讲解数据结构,但是在那之前我将介绍 `List` 和 `Array` ,这样你可以开始编写有用的Scala脚本.
在Scala中,数组是 `scala.Array` 类的一个实例,它类似于Java的数组:

```
scala> val array = new Array[String](3)
array: Array[String] = Array(null, null, null)
scala> array(0) = "This"
scala> array(1) = "is"
scala> array(2) = "mutable"
scala> array
res37: Array[String] = Array(This, is, mutable)
```

总是记住在Scala中提供的类型信息或参数化使用方括号.类型参数化是一种使用类型信息配置的方式当创建实例的时候.
现在迭代数组中的每个元素是很容易的;调用 `foreach` 方法:

```
scala> array.foreach(println)
This
is
mutable
```

你传递一个函数字面量给 `foreach` 方法去打印数组中的所有元素.在 `Array` 对象中还有其它非常有用的方法;完整的列表在Scaladoc的 `scala.collection.mutable.ArrayLike` 中.目前你脑中最明显的问题可能是为什么我们要去检查 `ArrayLike` ,这个一个在Scala中不同于 `Array` 的类用于检查一个 `Array` 实例的可用的方法.回答是 `Predef` .Scala `Predef` 自动提供了额外的数组函数,当你创建一个 `Array` 实例使用 `ArrayLike` 的时候.再次说明, `Predef` 是理解Scala库的一个很好的开始.

注意 `Predef` 隐式地将 `Array` 转换为 `scala.collection.mutable.ArrayOps` . `ArrayOps` 是 `ArrayLike` 的子类,所以 `ArrayLike` 更像是一个接口用于 `Array` 类型集合的所有的额外的可用方法.

当你编写Scala脚本,你有时候需要用命令行-比如参数.你可以隐式地这么做,因为有一个 `val` 类型的变量称为 `args` .在下面的示例中,你创建了一个Scala脚本,获取用户的输入,然后打印到控制台:

```
args.foreach(println)
```

打开你最喜欢的编辑器,将这行保存到一个称为 `myfirstScript.scala` 的文件中.在文件被保存的位置打开一个命令提示符,然后运行下面的命令:

```
scala  myfirstScript.scala my first script
```

你将看下以下的输出:

```
my
first
script
```

你执行了你的第一个脚本.你使用你开始Scala REPL环境的相同的命令.但是在这种情况下,你通过指定脚本文件名称和三个参数执行一个Scala脚本: my, first 和 script.你将在本章的末尾看到另一个脚本示例.

`Array` 是一种可变的数据结构;通过添加每个元素到数组中,你改变了数组实例,并产生了一个副作用.在函数式编程中,方法应该没有副作用.一个方法的唯一的作用是允许计算一个值,然后返回那个值而不会改变实例.像数组一样有一个有序对象的不可变和更多函数选择的是 `List` .在Scala中, `List` 是不可变的,使得函数式编程更容易.创建元素的一个列表如下这么简单:

```
scala> val myList = List("This", "is", "immutable")
myList: List[java.lang.String] = List(This, is, immutable)
```

`List` 是 `scala.collection.immutable.List` 的简写,再次说明, `Predef` 自动使得它对你可用:

```
scala> val myList = scala.collection.immutable.List("This", "is", "immutable")
myList: List[java.lang.String] = List(This, is, immutable)
```

什么是 scala.collection.immutable.$colon$colon ?
如果你在 myList 上调用 getClass 方法看下对象是什么类型,你可能感到好奇.因为不是 scala.collection.immutable.List,你将看到:

```
scala> myList.getClass
res42: java.lang.Class[_] = class
scala.collection.immutable.$colon$colon
```

那是因为 scala.collection.immutable.List 是一个抽象类,它有两种实现: scala.Nil 类和 scala.::.在 Scala 中,::是一个有限的标识符,你可以使用它去命名一个类.Nil 代表了一个空的列表,scala::代表了任意的非空列表.

我们中的大多使用可变的集合,当我们我们添加或移除元素的时候,集合实例扩展或收缩(改变),但是不可变的集合永远不会改变.相反,从一个不可变的集合中添加或移除元素将创建一个新的修改过的集合实例而不是修改原来的那个.下面的示例添加一个元素到已存在的 `List` 中:

```
scala> val oldList = List(1, 2)
oldList: List[Int] = List(1, 2)
scala> val newList = 3 :: oldList
newList: List[Int] = List(3, 1, 2)
scala> oldList
res45: List[Int] = List(1, 2)
```

在这个例子中你使用 `::` 方法添加了3到包含元素1和2的已存在的 `List` 中. `::` 的任务是使用所有已存在的元素加上被添加到 `List` 的前面新元素创建一个新的 `List` .为了在 `List` 的末尾添加, 调用 `:+` 方法:

```
scala> val newList = oldList :+ 3
newList: List[Int] = List(1, 2, 3)
```

Scala提供了一种称为 `Nil` 的特殊对象代表了空的 `List` ,你可以很容易使用它去创建一个新的列表:

```
scala> val myList = "This" :: "is" :: "immutable" :: Nil
myList: List[java.lang.String] = List(This, is, immutable)
```

为了从一个 `List` 中删除一个元素,你可以使用 `-` 方法,但是它是被废弃的.可以使用的一种更好的方式是使用 `filterNot` 方法,采用一个断言(predicate),选择所有不满足断言的元素.为了从 `newList` 中删除3,你可以像如下这么做:

```
scala> val afterDelete = newList.filterNot(_ == 3)
afterDelete: List[Int] = List(1, 2)
```

你将在第4章,4.3小节深入了解Scala集合,但是目前你可有足够的认识使用Scala和脚本做一些事情.在此期间我鼓励你看下定义在 `Scala` 中的方法,然后使用它们.

2.4 使用循环(loop)和if控制流程

使用Scala而不使用 `loop` 和 `if` 很难进行有用的编程或脚本.嗯,你不用再等了.在Scala中, `if` 和 `else` 块和它们在其它编程语言中工作是一样的.如果在 `if` 中的表达式被计算为 `true`,则 `if` 块被执行;否则, `else` 块被执行.关于Scala非常有趣的部分是每个语句都是一个表达式,它是值由语句中的最后的表达式决定.在Scala中分配一个值依赖于某些条件,比如像这样:

```
val someValue = if(some condition) value1 else value2
scala> val useDefault = false
useDefault: Boolean = false
scala> val configFile = if(useDefault) "custom.txt" else "default.txt"
configFile: java.lang.String = default.txt
```

Scala不支持像Java一样的 `?` 操作符,但是我不认为在Scala中你会怀念它.你可以内置 `if/else` 块,你也可以使用 `if` 组合多个 `if/else` 块.
在Scala中循环有和像 `while` 循环和 `do-while` 的同样的语义,但是最有趣的循环结构是 `for` 或 `for-推导` . `while` 和 `do-while` 循环是相当标准的,在Scala中它们不同于 Java或C#.下一节将看下 Scala for-推导.

2.5 For推导(For-comprehensions)

在Scala中的for-推导像一个瑞士军刀:你可以使用基本的简单元素用它做许多事情.在Scala中for表达式由 `for` 关键字组成,后面是括号包围的一个或多个枚举成员,接着是一个表达式块或 `yield` 表达式(见图2.2).在我开始 `yield` 表达式之前,我们深入下 `loop` 的更多的传统形式.在一个 `for` 循环使用的通用规则是迭代一个集合.
为了打印一个目录中的所有以 `.scala` 扩展名结尾的文件,比如,你需要像如下做些事情:

```
val files = new java.io.File(".").listFiles
for(file <- files) {
    val filename = file.getName
    if(fileName.endsWith(".scala")) println(file)
}
```

看起来与Java或C#中循环唯一不同的是表达式 `file <- files` .在Scala中这个被称为 *生成器(generator)*,一个 `generator` 的任务是迭代一个集合. `<-` 右边的部分代表了集合,在这种情况中是 `files` .对于集合中的每个元素(这种情况是 `file`),它执行 `for` 代码块.
这个类似于你在Java中定义一个 `for` 循环的方式:

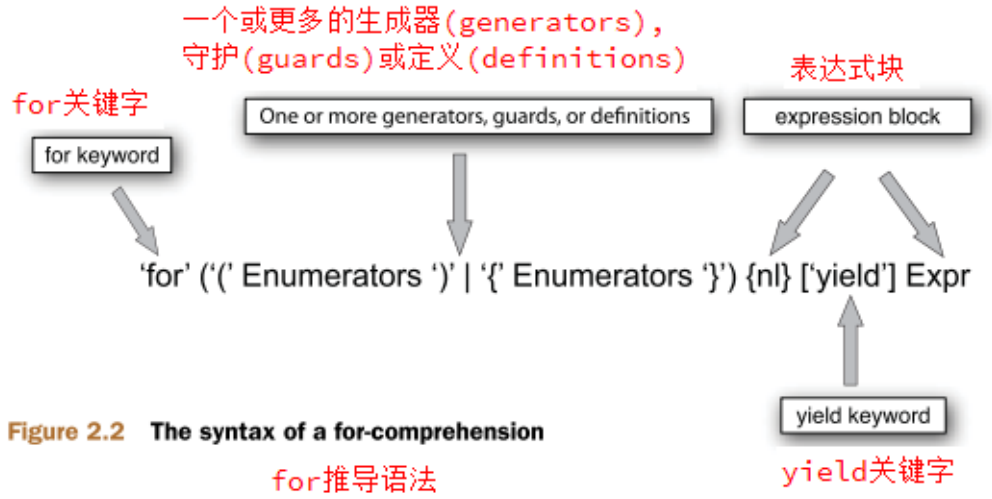

Figure 2.2 The syntax of a for-comprehension

for推导语法

```
1.  for(File file: files) {
2.        String filename = file.getName();
3.        if(filename.endsWith(".scala")) System.out.println(file);
4.  }
```

在Scala中,你不用指定文件对象的类型,因为Scala类型推导会处理它.

除了生成器,你可以在Scala中使用其它的语法简化 `for` 循环.

```
1.  for(
2.      file <- files;
3.      fileName = file.getName;
4.      if(fileName.endsWith(".scala"))
5.  ) println(file)
```

Scala的 `for` 循环允许你在循环中指定定义和守护块(guard clauses).在这种情况中,你定义了一个 `filename` 变量和检查 `filename` 是否以 `.scala` 结束.循环将执行当给定的守护块为 `true`,所以你得到和之前例子一样的结果.注意,在 `for` 表达式中创建的所有的变量都是 `val` 类型,所以你不能改变它,因此减少了副作用的可能性.

如之前提到的,可以在Scala中的 `for` 循环控制中定义多个生成器.下面的示例为每个生成器执行 `for` 循环, 并将它们相加:

```
1.  scala> val aList = List(1, 2, 3)
2.  aList: List[Int] = List(1, 2, 3)
3.  scala> val bList = List(4, 5, 6)
4.  bList: List[Int] = List(4, 5, 6)
5.  scala> for { a <- aList; b <- bList } println(a + b)
6.  5
7.  6
8.  7
9.  6
10. 7
11. 8
12. 7
13. 8
14. 9
```

在这种情况中的生成器是 `aList` 和 `bList`,当你有多个生成器,每个生成器会被其它的生成器重复.当 `a=1`, `b` 的每个值是 `b=4`, `b=5`, `b=6`,循环将被执行,等等.是使用了大括号去包围了 `for` 表达

式,但是你不必这么做;你可与使用 `()` .我倾向于使用大括号当我有一个或多个生成器和守护块的时候.

在Scala中for-推导有两个风格.在之前的示例中你已经看到过一种形式: *命令式*(imperative form).在这种形式中你指定了将被循环执行的语句,它不返回任何东西.for-推导的另一种形式被称为 *函数式*(functional form)(有时候也被称为*顺序解读*-sequence comprehension).在函数式中,你倾向于使用值而不是执行语句,并且它返回一个值.看下函数式风格的相同的例子:

```
1.  scala> for { a <- aList; b <- bList } yield a + b
2.  res27: List[Int] = List(5, 6, 7, 6, 7, 8, 7, 8, 9)
```

你使用了 `yield` 关键字从循环中返回了相加的值,而不是打印 `a + b` 的值.你在循环控制中使用了相同的 `aList` 和 `bList` 实例,它返回的结果作为一个 `List` .现在如果你打印结果,如之前的例子所示,你需要循环结果的 `List` :

```
1.  scala> val result = for { a <- aList; b <- bList } yield a + b
2.  result: List[Int] = List(5, 6, 7, 6, 7, 8, 7, 8, 9)
3.  scala> for(r <- result) println(r)
4.  5
5.  6
6.  7
7.  6
8.  7
9.  8
10. 7
11. 8
12. 9
```

看上去函数式比命令式更加冗长,但是思考一下.你从如何使用它中分离出了计算(两个数的相加)-在这种情况中,打印结果.这个提高了函数的可重用性和兼容性或计算指令,这是函数式编程的好处之一.在下面的例子中你重用了由 `for-yield` 循环产生的结果去创建一个XML节点:

```
1.  scala> val xmlNode = <result>{result.mkString(",")}</result>
2.  xmlNode: scala.xml.Elem = <result>5,6,7,6,7,8,7,8,9</result>
```

`mkString` 是一个定义在 `scala.collection.imuutable.List` 中的方法,它需要在列表中的每个元素并将每个元素与你提供的任何分隔符串联-在这种情况下是一个逗号.即使它没有意义,如果你尝试在 `yield` 表达式内部打印会发生什么?将会发生什么?记住,在Scala中以一切都是表达式,并且有一个返回值.如果你尝试下面的例子,你仍然会得到一个结果,但是结果没有什么用,因为它是一个 `units` 的集合.一个 `unit` 等同于 Java中的 `void` , 它是用在yield表达式内部的一个 `println` 函数的返回值.

```
1.  scala> for { a <- aList; b <- bList } yield { println(a+b)}
2.  5
3.  6
4.  7
5.  6
6.  7
7.  8
8.  7
9.  8
10. 9
11. res32: List[Unit] = List((), (), (), (), (), (), (), (), ())
```

目前你只是触及到了for-推导的表面,在第4章我还将回到这个主题去解释函数式数据结构.所以保持你的好奇心直到第4章(或者跳到那一章).下一节将转移到另一个函数式概念: 模式匹配.

2.6 模式匹配

模式匹配是要在Scala中介绍的另一种函数式概念.首先,Scala模式匹配看起来与Java中的 `switch` 类似.在下面清单中的示例,展示了Scala和Java之间的相似性,接收一个整数,然后打印它的原始数字.

清单 2.2 用 Java 写的一个 ordinal 类

```
1.  public class Ordinal {
2.    public static void main(String[] args) {
3.      ordinal(Integer.parseInt(args[0]));
4.    }
5.    public static void ordinal(int number) {
6.      switch(number) {
7.        case 1: System.out.println("1st"); break;
8.        case 2: System.out.println("2nd"); break;
9.        case 3: System.out.println("3rd"); break;
10.       case 4: System.out.println("4th"); break;
11.       case 5: System.out.println("5th"); break;
12.       case 6: System.out.println("6th"); break;
13.       case 7: System.out.println("7th"); break;
14.       case 8: System.out.println("8th"); break;
15.       case 9: System.out.println("9th"); break;
16.       case 10: System.out.println("10th"); break;
17.       default : System.out.println("Cannot do beyond 10");
18.     }
19.   }
20. }
```

这里程序的参数被解析为整数值,然后 `ordinal` 方法返回给定数字的序数文本.目前,它只知道如何处理从1到10的数字.下面的清单展示了在Scala中的相同的示例.

清单 2.3 Scala 中的 ordinal 类

```
1.  ordinal(args(0).toInt)
2.  def ordinal(number:Int) = number match {
3.    case 1 => println("1st")
4.    case 2 => println("2nd")
5.    case 3 => println("3rd")
6.    case 4 => println("4th")
7.    case 5 => println("5th")
8.    case 6 => println("6th")
9.    case 7 => println("7th")
10.   case 8 => println("8th")
11.   case 9 => println("9th")
12.   case 10 => println("10th")
13.   case _ => println("Cannot do beyond 10")
```

这里你做了与之前Java示例一样的事情:从命令行比如 `args` 接收一个输入的整数值,然后决定这个数字的序数文本.因为Scala也可以被作为一个脚本语言使用,你不必定义一个入口点,比如 `main` 方法.另外你也不必为每个case提供break,因为在Scala中你不能溢出到其他的 `case` 子句(引起多次匹配),如Java一样,并且没有默认的 `default` 语句.在Scala中, `default` 使用 `case_` 替换去匹配其它的一切.为了运行 `Ordinal.scala` 脚本,从一个命令提示符中执行下面的命令:

```
1.  scala Ordinal.scala <your input>
```

通配符case是可选的,工作起来像一个安全备选方案(a safe fallback option).如果你就爱那个它移除,已存在的case没有一个匹配,你将得到一个匹配错误:

```
scala> 2 match { case 1 => "One" }
scala.MatchError: 2
    at .<init>(<console>:5)
    at .<clinit>(<console>)
  ...
```

这个相当棒的,因为它告诉你缺少了一个 `case` 子句,不像Java,如果你移除了 `default` ,已存在的case没有一个匹配,它忽略它而不会提供任何的反馈.

Java和Scala模式匹配的相似性到这里结束了,因为Scala让模式匹配进入了另一个层级.在Java中你只能在 `switch` 子句使用基本类型和枚举,但是在Scala你可以模式匹配字符串和复杂的值,类型,变量,常量和构造器.更多的模式匹配概念在下一章介绍.下面的例子定义了一个方法,这个方法接收一个输入,然后检查给定的对象的类型:

```
def printType(obj: AnyRef) = obj match {
  case s: String => println("This is string")
  case l: List[_] => println("This is List")
  case a: Array[_] => println("This is an array")
  case d: java.util.Date => println("This is a date")
}
```

在这个例子中你使用的Scala类型匹配由一个变量和类型组成.这个模式匹配任意的值,通过类型模式匹配-在这个例子中,是 `String` , `List[AnyRef]` , `Array[AnyRef]` ,和 `java.util.Date` .当使用类型的模式匹配,它将变量名绑定到值.你可以使用如Java中的 `instanceof` 操作符,然后转型,不过这是一个更优雅的解决方案.将这个方法保存到问价 `printType.scala` 中,然后把这个文件加载到Scala REPL中:

```
scala> :load printType.scala
Loading printType.scala...
printType: (obj: AnyRef)Unit
```

现在使用各种类型的输入测试 `printType` 函数;

```
scala> printType("Hello")
This is string
scala> printType(List(1, 2, 3))
This is List
scala> printType(new Array[String](2))
This is an array
scala> printType(new java.util.Date())
This is a date
```

Scala也允许 `infix(::)` 操作符,你可以在你的模式中指定一个infix操作符.在infix中,操作符被写在操作数之间-比如, `2 + 2`.在下面的示例中,你从 `List` 中提取第一个和第二个元素:

```
scala> List(1, 2, 3, 4) match {
         case f :: s :: rest => List(f, s)
         case _ => Nil
       }
res7: List[Int] = List(1, 2)
```

这里你将1匹配到f变量,2到s变量,3和4到reset变量.(Think of it as what it will take to create a

List of 1, 2 ,3, and 4 from the expression f :: s :: rest)这个会更有意义.

有时候你需要有一个守护块随着 `case` 语句一起使用,这样会在模式匹配期间有更多的灵活性.在下面的示例中你决定了给定的数字的范围:

```
1.  def rangeMatcher(num:Int) = num match {
2.    case within10 if within10 <= 10 => println("with in 0 to 10")
3.    case within100 if within100 <= 100 => println("with in 11 to 100")
4.    case beyond100 if beyond100 < Integer.MAX_VALUE => println("beyond 100")
5.  }
```

有了这个新的信息,回顾下ordinal的问题.之前的Scala ordinal示例只支持从1到10的数字,但是下面的清单实现了所有的整数.

清单 2.4 Ordianl2.scala 重新实现

```
1.  val suffixes = List(
2.        "th", "st", "nd", "rd", "th", "th", "th", "th", "th", "th")
3.
4.  println(ordinal(args(0).toInt))
5.
6.  def ordinal(number:Int) = number match {
7.    case tenTo20 if 10 to 20 contains tenTo20 => number + "th"
8.    case rest => rest + suffixes(number % 10)
9.  }
```

这里是你使用范围的ordinal的新的实现,这是一个在给定的开始值和终止值之间的整数的集合.表达式 `10 to 20` 是 10.to(20)(记住方法可被用作 infix 操作符).你调用了 `RichInt` 中的 `to` 方法,它创建了一个包含的范围(`scala.collection.immutable.Inclusive`).你在这个范围上调用了 `contains` 方法去检查数字是否属于这个范文.在最后的case中你将低于10超过20的所有数字映射到一个称为 `rest` 的新的变量.这个在Scala中被称为*变量模式匹配(variable pattern matching)*.你可以在 `List` 像数组一样使用索引位置访问一个 `List` 的元素.你将在第3章在看过case类后回顾模式匹配.现在是时候让我们转移到这章的最后的主题:异常处理.

2.7 异常处理

在 `breakable` 的例子中你已经瞥见过Scala异常处理.Scala的异常处理和Java有点不同.Scala允许你用一个单独的 `try/catch` 块,在这个单独的 `catch` 块中你可以使用模式匹配去捕获异常, `catch` 块是一种表面上的匹配块,所以你之前学习过的所有的模式匹配技术都适用于一个 `catch` 块.修改 `rangeMatcher` 去抛出一个异常当它超过100:

```
1.  def rangeMatcher(num:Int) = num match {
2.    case within10 if within10 <= 10 => println("with in 0 to 10")
3.    case within100 if within100 <= 100 => println("with in 11 to 100")
4.    case _ => throw new IllegalArgumentException(
5.              "Only values between 0 and 100 are allowed")
6.  }
7.  `
```

现在当调用这个方法的时候你可以使用 `try/catch` 块包围,然后捕获异常:

```
1.  scala> try {
2.            rangeMatcher1(1000)
3.        } catch { case e: IllegalArgumentException => e.getMessage }
```

```
4.    res19: Any = Only values between 0 and 100 are allowed
```

这个 `case` 语句与 `printType` 例子中使用的类型模式匹配没有任何不同.

Scala没有任何像受检查的异常这样的概念;所有的异常都是未检查的.这种方式更强大,和灵活因为作为一个程序员你可以自由决定是否去捕获一个异常.即使Scala异常处理实现是不同的,不过它的行为完全和Java一样,异常是不需要被检查的,这样它允许Scala容易与已存在的Java库交互.本书中你将在示例中看到Scala异常处理的使用方式.

2.8 命令行 REST 客户端: 构建一个工作示例

在这章你已经深入了关于Scala的很多有趣的概念,在实战中同时看到这些概念是一件挺有意思的事情.在这节,你以Scala脚本的方式构建一个基于命令行的REST客户端.你准备使用Apache HttpClient 库去处理HTTP连接和各种HTTP方法.

什么是 REST?

REST 代表了表述性状态转移(REpresentational State Transfer).它是用于分布式超媒体系统比如World Wide Web 的软件架构风格.这个术语首次出现在"Architectural Styles and the Design of Network based Software Architectures,", Roy Fielding 的博士研究论文,HTTP 规范的主要作者之一.

REST 完全引用了这里提到的架构原则的集合.遵循 REST 原则的系统通常被称为 RESTful.

- 应用程序状态和公共被分为资源.
- 使用一种通用语法每个资源是独立可寻址的.
- 所有的资源共享一个客户端和资源直接的状态转换的独立的接口,由众所周知的操作(GET, POST, PUT, DELETE,OPTIONS, 等等, 用于 RESTful web services)组成.和文档类型.
- 一个客户端/服务器协议,无状态缓存,和分层.

为了完成一个RESTful服务的REST调用,你需要注意被服务支持的操作.为了特侧你的客户端你需要有一个RESTful web服务.你可以使用使用免费的公共web服务去试客户端,但是最好控制服务上你创建的操作.你应该使用任意的REST工具或框架去构建REST服务.我将使用Java servlet(Java开发者很熟悉它)去构建服务区测试REST客户端.理解服务如何被实现对于这个例子而已不重要.

创建一个RESTful服务的最简单的方式是使用一个Java servlet,如下面的清单所示.

清单 2.5 Java servlet 作为一个 RESTful 服务

```
1.    package restservice;
2.
3.    import java.io.IOException;
4.    import java.io.PrintWriter;
5.    import java.util.Enumeration;
6.
7.    import javax.servlet.ServletException;
8.    import javax.servlet.http.HttpServlet;
9.    import javax.servlet.http.HttpServletRequest;
10.   import javax.servlet.http.HttpServletResponse;
11.
12.   /**
13.    * Servlet implementation class TestRestService
14.    */
15.   public class TestRestService extends HttpServlet {
16.       private static final long serialVersionUID = 1L;
```

```
17.
18.     public void doGet(HttpServletRequest request, HttpServletResponse response) throws Serv
    letException, IOException {
19.         PrintWriter out = response.getWriter();
20.         out.println("Get method called");
21.         out.println("parameters: " + parameters(request));
22.         out.println("headers: " + headers(request));
23.     }
24.
25.     public void doPost(HttpServletRequest request, HttpServletResponse response) throws Ser
    vletException, IOException {
26.         PrintWriter out = response.getWriter();
27.         out.println("Post method called");
28.         out.println("parameters: " + parameters(request));
29.         out.println("headers: " + headers(request));
30.     }
31.
32.     public void doDelete(HttpServletRequest request, HttpServletResponse response) throws S
    ervletException, IOException {
33.         PrintWriter out = response.getWriter();
34.         out.println("Delete method called");
35.     }
36.
37.     // 构建响应字符串
38.     private String parameters(HttpServletRequest request) {
39.         StringBuilder builder = new StringBuilder();
40.         for (Enumeration e = request.getParameterNames(); e.hasMoreElements();) {
41.             String name = (String) e.nextElement();
42.             builder.append("|" + name + "->" + request.getParameter(name));
43.         }
44.         return builder.toString();
45.     }
46.
47.     // 从headers中构建响应字符串
48.     private String headers(HttpServletRequest request) {
49.         StringBuilder builder = new StringBuilder();
50.         for (Enumeration e = request.getHeaderNames(); e.hasMoreElements();) {
51.             String name = (String) e.nextElement();
52.             builder.append("|" + name + "->" + request.getHeader(name));
53.         }
54.         return builder.toString();
55.     }
56.
57. }
```

在servlet中你支持了三种HTTP方法: `GET` , `POST` , 和 `DELETE` .这些方法相当简单,在响应中返回请求参数和请求头,当测试你的REST客户端的时候这是相当不错的.我添加了两个 `parameters` 和 `headers` 的辅助方法. `parameters` 方法负责解析从客户端传递过来的HTTP请求对象的参数;在这个例子中,它是REST客户端. `headers` 方法检索请求对象的所有的 header 值.一旦servlet被构建,你必须发布WAR文件到一个Java web容器.我已经使用了Maven和Jetty去构建然后运行Java servlet,但是你可以随意使用任何的Java web容器.

2.8.1 HttpClient 库介绍

HttpClient是一个客户端的HTTP传输库.HttpClient的目的是传输和接收HTTP消息.它不是一个浏览器,它不执行JavaScript或尝试猜测文档类型或其它与HTTP传输不相关的功能.HttpClient客户端最基本的功能是执行

HTTP方法.假设用户提供了一个请求对象如 `HttpPost` 或 `HttpGet` ,HttpClient 期望传输请求给目标服务器然后返回对应的响应对象,或者抛出异常如果执行不成功.

HttpClient封装了在一个对象中的每种HTTP方法类型,它们是可用的,位于 `org.apache.http.client.methods` 包中.在这个脚本中你将使用请求的四种类型: `GET` , `POST` , `DELETE` ,和 `OPTIONS` .之前的例子只实现了 `GET` , `POST` , 和 `DELETE` ,因为容器将自动实现 `OPTIONS` 方法,HttpClient客户端提供了默认的客户端,对我们而已足够了.为了执行HTTP `DELETE` 方法,你需要做如下的操作:

```
1.  val httpDelete = new HttpDelete(url)
2.  val httpResponse = new DefaultHttpClient().execute(httpDelete)
```

HTTP `POST` 方法有点不同,因为,根据HTTP规范,它是两个实体封闭(entity-enclosing)方法中的一个.另一个是 `PUT` .为了使用实体HttpClient提供的多个选项,但是在这个例子中,你准备使用实体中的URL-编码.它类似于当你 `POST` 一个表单时所发生的.现在你将深入构建客户端.

为了在你的脚本中使用HttpClient,你需要导入所有必要的类.我还没有讨论关于 `import` 的内容,但是现在将它认为与Java导入类似,除了Scala使用 `_` 导入一个包中的所有类,如下所示:

```
1.  import org.apache.http._
2.  import org.apache.http.client.entity._
3.  import org.apache.http.client.methods._
4.  import org.apache.http.impl.client._
5.  import org.apache.http.client.utils._
6.  import org.apache.http.message._
7.  import org.apache.http.params._
```

你可以使用Scala导入做一些其它有趣的事情,不过这在下一章.

2.8.2 一步一步构建客户端

现在,因为服务正在启动和运行,你可以关注客户端的脚本了.为了保证教程有用,你要知道操作的类型(`GET` 或 `POST`),请求参数,headers参数,和服务的URL.请求参数和header参数是可选的,但是你需要一个操作和一个URL使得任何成功的REST调用:

```
1.  require(args.size >= 2,
2.  "at minimum you should specify action(post, get, delete, options) and url")
3.
4.  val command = args.head
5.  val params = parseArgs(args)
6.  val url = args.last
```

你使用了定义在 `Predef` 中的 `require` 函数去检查输入的大小.记住命令行输入通过一个数组 `arrays` 表示. `require` 函数抛出异常当谓词计算为 `false` 的时候.在这个例子中,因为你期望至少有两个参数,任何少于这个的都会导致一个异常.脚本的第一个参数是命令,接下来是请求和header参数.最后的参数是URL.脚本的输入看起来如下:

```
1.  post -d <comma separated name value pair>
2.         -h <comma separated name value pair> <url>
```

请求参数和header参数通过一个前缀参数决定, `-d` 或 `-h` .一种定义一个 `parseArgs` 方法去解析请求和header参数的方式定义在下面的清单中.

清单 2.6 解析传递给程序的 headers 和参数

```
def parseArgs(args: Array[String]): Map[String, List[String]] = {
  def nameValuePair(paramName: String) = {
    def values(commaSeparatedValues: String) =
        commaSeparatedValues.split(",").toList

     val index = args.findIndexOf(_ == paramName)
    (paramName, if(index == -1) Nil else values(args(index + 1)))
  }
  Map(nameValuePair("-d"), nameValuePair("-h"))
}
```

这个清单中在另一参数中定义了另一个函数.Scala允许内嵌的函数,并且内嵌的函数可以访问定义在外部范围函数中的变量-在这个例子中 `parseArgs` 函数的 `args` 参数.内嵌的函数允许你以一种有趣的方式去封装更小的方法和 `break` 计算.这里内嵌的函数 `nameValuePair` 接受参数名称, `-d` , `-h` ,并创建一个请求或header参数的 `name-value` 对的列表. 关于 `nameValuePair` 函数的另一比较有趣的事情是返回类型.这个返回类型是 `scala.Tuple2` ,一个两个元素的元组. `Tupe` 是不可变的,像 `List` 一样,但是不像 `List` ,它(元组)可以包含不同类型的元素;在这个例子中,它包含了一个 `String` 和一个 `List` .Scala提供了通过使用括号 `()` 包装元素创建一个 `Tupe` 的语法糖.

```
scala> val tuple2 = ("list of one element", List(1))
tuple2: (java.lang.String, List[Int]) = (list of one element,List(1))
```

这个类似于:

```
scala> val tuple2 = new scala.Tuple2("list of one element", List(1))
tuple2: (java.lang.String, List[Int]) = (list of one element,List(1))
```

这里是如何创建三个元素的元组;

```
scala> val tuple3 = (1, "one", List(1))
tuple3: (Int, java.lang.String, List[Int]) = (1,one,List(1))
```

我想提的关于 `parseArgs` 方法最后一个有趣的事情是 `Map` .一个 `Map` 是keys和values的不变集合.第4章详细讨论 `Map` .在这个例子中你创建了参数名称(`-d` 或 `-h`)和列出所有作为值的参数的一个 `Map` .当你传递一个两个元素的元组给 `Map` ,它将元组的第一个元素作为key,第二个元素作为value:

```
scala> val m = Map(("key1", "value1"), ("key2", "value2"))
m: scala.collection.immutable.Map[java.lang.String,java.lang.String] =
Map(key1 -> value1, key2 -> value2)
scala> m("key1")
res8: java.lang.String = value1
```

现在,你只支持REST的四种操作: `POST` , `GET` , `DELETE` , 和 `OPTIONS` ,但是我鼓励你实现其它的HTTP方法,比如 `PUT` 和 `HEAD` .

为了检查请求是什么类型,你可以使用简单的模式匹配;

```
command match {
  case "post"    => handlePostRequest
  case "get"     => handleGetRequest
  case "delete"  => handleDeleteRequest
  case "options" => handleOptionsRequest
}
```

这里 `handlePostRequest`，`handleGetRequest`，`handleDeleteRequest`，和 `handleOptionRequest` 都是定义在脚本中的函数.每个都需要被实现但有点差异.比如,在 `GET` 调用中,你将传递请求参数作为查询参数给URL. `POST` 将使用URL-编码表单实体传递参数. `DELETE` 和 `OPTIONS` 不会使用任何的请求参数.看下 `handleGetRequest` 方法,展示在下面的清单中.

清单 2.7 准备一个 GET 请求,然后调用 REST 服务

```
def headers = for(nameValue <- params("-h")) yield {
  def tokens = splitByEqual(nameValue)
  new BasicHeader(tokens(0), tokens(1))
}

def handleGetRequest = {
  val query = params("-d").mkString("&")
  val httpget = new HttpGet(s"${url}?${query} " )
  headers.foreach { httpget.addHeader(_) }
  val responseBody =
      new DefaultHttpClient().execute(httpget,
new BasicResponseHandler())
  println(responseBody)
}
```

在这个方法只能中你检索 `Map` 参数中的所有的请求参数,并创建查询字符串.然后你使用给定的URL和查询字符串创建 `HttpGet` 方法. `DefaultHttpClient` 执行 `http get` 请求,并给出响应. `handlePostRequest` 方法方法更复杂一些,因为它需要创建一个表单实体对象,如下面的清单所示.

清单 2.8 准备一个 POST 请求并调用 REST 服务

```
def formEntity = {
  def toJavaList(scalaList: List[BasicNameValuePair]) = {
    java.util.Arrays.asList(scalaList.toArray:_*)
  }
  def formParams = for(nameValue <- params("-d")) yield {
    def tokens = splitByEqual(nameValue)
    new BasicNameValuePair(tokens(0), tokens(1))
  }
  def formEntity =
       new UrlEncodedFormEntity(toJavaList(formParams), "UTF-8") // 编码POST请求参数
  formEntity
}
def handlePostRequest = {
  val httppost = new HttpPost(url)
  headers.foreach { httppost.addHeader(_) }
  httppost.setEntity(formEntity)
  val responseBody = new DefaultHttpClient().execute(httppost, new
BasicResponseHandler())
  println(responseBody)
}
```

这儿有些有趣和不寻常的东西.首先是 `toJavaList` 方法.Scala `List` 和 Java `List` 是两种不同类型的对象,不能够互相兼容.因为 `HttpClient` 是一个Java库,你需要在调用 `UrlEncodedFormEntity` 之前将它转换为一个Java类型的集合.特殊的 `_*` 告诉Scala编译器发送 `toArray` 的结果作为一个变量参数给 `Arrays.asList` 方法;否则, `asList` 将创建有1个元素的Java `List` .

下下面的例子演示了这一事实:

```
scala> val scalaList = List(1, 2, 3)
scalaList: List[Int] = List(1, 2, 3)

scala> val javaList = java.util.Arrays.asList(scalaList.toArray)
javaList: java.util.List[Array[Int]] = [[I@67826710]

scala> val javaList = java.util.Arrays.asList(scalaList.toArray:_*)
javaList: java.util.List[Int] = [1, 2, 3]
```

下面的清单包含了完整的 RestClient.scala 脚本.

```
import org.apache.http._
import org.apache.http.client.entity._
import org.apache.http.client.methods._
import org.apache.http.impl.client._
import org.apache.http.client.utils._
import org.apache.http.message._
import org.apache.http.params._

// 解析请求参数和headers
def parseArgs(args: Array[String]): Map[String, List[String]] = {
  def nameValuePair(paramName: String) = {
    def values(commaSeparatedValues: String) =
            commaSeparatedValues.split(",").toList
    val index = args.findIndexOf(_ == paramName)
    (paramName, if(index == -1) Nil else values(args(index + 1)))
  }
  Map(nameValuePair("-d"), nameValuePair("-h"))
}

def splitByEqual(nameValue:String): Array[String] = nameValue.split('=')

// 为每个name/value对创建 BasicHeader
def headers = for(nameValue <- params("-h")) yield {
  def tokens = splitByEqual(nameValue)
  new BasicHeader(tokens(0), tokens(1))
}

// 创建URL-编码表单实体
def formEntity = {
  def toJavaList(scalaList: List[BasicNameValuePair]) = {
    java.util.Arrays.asList(scalaList.toArray:_*)
  }
  def formParams = for(nameValue <- params("-d")) yield {
    def tokens = splitByEqual(nameValue)
    new BasicNameValuePair(tokens(0), tokens(1))
  }
  def formEntity =
new UrlEncodedFormEntity(toJavaList(formParams), "UTF-8")
  formEntity
}
def handlePostRequest = {
  val httppost = new HttpPost(url)
  headers.foreach { httppost.addHeader(_) }
  httppost.setEntity(formEntity)
  val responseBody =
new DefaultHttpClient().execute(httppost, new BasicResponseHandler())
```

```
47.   println(responseBody)
48.   }
49.   def handleGetRequest = {
50.     val query = params("-d").mkString("&")
51.     val httpget = new HttpGet(s"${url}?${query}")
52.     headers.foreach { httpget.addHeader(_) }
53.     val responseBody =
54.   new DefaultHttpClient().execute(httpget, new BasicResponseHandler())
55.     println(responseBody)
56.   }
57.   def handleDeleteRequest = {
58.     val httpDelete = new HttpDelete(url)
59.     val httpResponse = new DefaultHttpClient().execute(httpDelete)
60.     println(httpResponse.getStatusLine())
61.   }
62.   def handleOptionsRequest = {
63.     val httpOptions = new HttpOptions(url)
64.     headers.foreach { httpOptions.addHeader(_) }
65.     val httpResponse = new DefaultHttpClient().execute(httpOptions)
66.     println(httpOptions.getAllowedMethods(httpResponse))
67.   }
68.
69.   // ① 校验参数的数目
70.   require(args.size >= 2, "at minmum you should specify
71.       action(post, get, delete, options) and url")
72.   val command = args.head
73.   val params = parseArgs(args)
74.   val url = args.last
75.
76.   // ② 模式匹配命令行参数
77.   command match {
78.     case "post"    => handlePostRequest
79.     case "get"     => handleGetRequest
80.     case "delete"  => handleDeleteRequest
81.     case "options" => handleOptionsRequest
82.   }
```

在这个完整的例子中你实现了支持四种类型的HTTP请求： `POST` ， `GET` ， `DELETE` ， `OPTIONS` . `require` 函数调用 ① 保证你的脚本包含至少两个参数：动作(action)类型和REST服务的URL.脚本最后的模式匹配块 ② 为给定的动作名称选择合适的动作处理器. `parseArgs` 函数处理脚本提供的额外参数,比如请求参数或headers,返回一个包含所有的name-value对的 `Map` . `formEntity` 函数是有趣的因为URL编码请求参数当http请求类型是 `POST` 的时候,因为在 `POST` 请求参数中作为请求体的一部分被发送,它们需要被编码.

为了运行 REST 客户端,你可以使用任意的可以构建Scala代码的构建工具.这个例子使用了一个称为简单构建工具(SBT)的构建工具.你可以在第6章详细地学习这个,但是现在,继续前进并安装 SBT,遵循 wiki（http://www.scala-sbt.org ） 中的说明.看一看这一章为例的代码库(看一看这一章为例的代码库).

2.9 总结

这章包含了大部分的基础的Scala概念,如数据类型,变量,和函数.你看到了如何安装和配置Scala.更为重要的是,你学习了如何在Scala中定义函数,重要的构建块,和函数的概念,包括模式匹配和for-推导.你也学习了关于异常处理和Scala如何使用相同的模式匹配技术用于异常处理.这章也提供了 `List` 和 `Array` 集合类型的基本说明,这样你可以开始构建有用的Scala脚本.第4章详细涵盖了Scala集合.本章中至始至终你使用了Sala REPL,当试验这些例子的时候.Scala REPL是一个重要和便利的工具,全书中你将都使用它.这章以使用你学习的大部分概念构建一个完整的REST客户端结束.这个例子也说明了Scala提供的灵活性,当构建脚本的时候.现在是时候转到

```
Scala类和对象了.
```