

`trampoline()`方法帮我们在享受递归之强大的同时避免了其缺点。这是很大的进步，但是在这一过程中，简洁性也失去了。现在不再是像`factorial(5)`这样简单地调用该方法，我们必须提供一个额外的参数，比如`factorial(5, 1)`。除了这种额外的负担，还很容易出错，比如我们可能为第二个参数提供了别的值，而不是1。

这个问题有个权宜之计——可以为闭包的第二个参数定义一个默认值，就像这样：`BigInteger theFactorial = 1`。调用者现在可以省略第二个参数，但还是无法防止他们提供错误的值。可以将闭包封装到一个函数内来解决这个问题。

UsingClosures/trampoline.groovy

```
def factorial(int factorialFor) {
    def tailFactorial
    tailFactorial = { int number, BigInteger theFactorial = 1 ->
        number == 1 ? theFactorial :
        tailFactorial.trampoline(number - 1, number * theFactorial)
    }.trampoline()
    tailFactorial(factorialFor)
}
println "factorial of 5 is ${factorial(5)}"
println "Number of bits in the result is ${factorial(5000).bitCount()}"
```

这里定义了一个函数`factorial()`，同时在该函数内定义了尾递归闭包，并调用了`trampoline()`方法。在该函数的最后，我们调用了闭包，并将想要计算阶乘的数传给它。闭包的第二个参数会被传入默认值1。

某些语言是通过编译技术来实现尾递归优化的。与它们不同，Groovy是通过闭包上的一个便捷方法实现的。这避免了对语言及其语法的整体影响，在减少内存占用的同时使编程变得更为优雅。不过使用`trampoline()`特性还是有个问题——它的性能要比简单递归或纯粹的迭代差一些。对于较大的输入值，这可能是个很大的限制。（不需要任何编译器技巧的实现固然是好事，但是希望不影响性能却并不现实。）要使`trampoline()`适用于较大的输入值，或者需要大大改进该实现的性能，或者需要一种编译器变换来改进效率。

下一节将介绍闭包对一种特殊类型的递归算法的运行时影响。

4.11 使用记忆化改进性能

我曾经看过一场鸟类表演，表演者让一个小伙子和一只鹦鹉进行算数比赛。这只鹦鹉可以快速地回答出“200乘以50是多少”之类问题，观众很开心，不过这个小伙子很懊恼。在鹦鹉又答对了接下来的一些问题之后，表演者向大家透露了其中的秘密——对于这些照本宣科的问题，鹦鹉只是在简单地重复所记住的答案。本节会使用一种类似的技术，不过我们不会称其为“记住”，而是继续本领域会为概念起个奇怪名字的传统，使用“记忆化”（Memoization）这个术语。

上一节介绍了一种可以使递归调用更高效地使用内存的技巧。递归本质上是一种使用子问题的解决方案来解决问题本身的方式。这种技巧有一个变种（被奇怪地命名为动态规划）^①，将问题分解为可以多次重复解决的若干部分。在执行期间，我们将子问题的结果保存下来，当调用到重复的计算时，只需要简单地使用保存下来的结果。这就避免了重复运行，因此极大地减少了计算时间。记忆化可以将一些算法的计算时间复杂度从输入规模（n）的指数级（ $O(k^n)$ ）降低到只有线性级（O(n)）。

为帮助理解记忆化这一概念以及Groovy中的相关设施，我们来看一下卖杆这种业务。不同长度的杆零售价格不同。我们会批发某一特定长度的杆，比如说27英寸（1英寸=2.54厘米），然后将其分割成长度不一的杆销售，以实现收入最大化。

首先使用简单递归来解决这个问题，然后再使用记忆化。因为记忆化是要减少计算时间，所以我们需要一种手段来度量所消耗的时间。就从一个可以帮助我们度量时间的小函数入手吧。

4

UsingClosures/rodCutting.groovy

```
def timeIt(length, closure) {
    long start = System.nanoTime()
    println "Max revenue for $length is ${closure(length)}"
    long end = System.nanoTime()
    println "Time taken ${(end - start)/1.0e9} seconds"
}
```

`timeIt()`方法会报告给定闭包运行所消耗的时间，并报告对于给定长度的杆，我们可以期望的最大收入，这里的最大收入是该闭包输出的。

作为示例，下面使用一个数组为各种长度的杆[从0英寸到30英寸（不包含）]定义一组零售价格。之所以要包含长度为0的杆，是为了弥补基于0的数组索引所带来的问题：^②

UsingClosures/rodCutting.groovy

```
def rodPrices = [0, 1, 3, 4, 5, 8, 9, 11, 12, 14,
                15, 15, 16, 18, 19, 15, 20, 21, 22, 24,
                25, 24, 26, 28, 29, 35, 37, 38, 39, 40]

def desiredLength = 27
```

`rodPrices`变量指向一个列表，其中保存了不同长度的杆的零售价格。变量`desiredLength`保存的是我们想要将其分割销售以实现收入最大化的原始杆的长度。

根据给定的零售价格，如果直接销售27英寸长的杆，收入为38美元。如果将其分割成分别长

^① 动态规划的英文表达是Dynamic Programming，如果不清楚其背景，可能会与本书所谈的动态特性编程相混淆，所以说命名有点奇怪。——译者注

^② 加入这个0之后，我们就可以以杆长度作为索引值来获得其零售价格了，比如`rodPrices[1]`表示的就是长度为1的杆的价格，也就是1，依此类推。——译者注

1英寸和26英寸的杆来销售，收入还是38美元，因此实际上不值得分割。如果分割成长4英寸和23英寸的两段，还会损失一些钱。如果分割成六段，五段5英寸长的，一段2英寸长的，总共会带来43美元的销售收入，超过了前面的38美元。人工计算需要相当长的时间。如果给定了一个任意的长度和价格，我们希望有一个程序能快速地计算出使得收入最大化的最优分割方案。

对于一个给定的长度，下面将要编写的代码会给出最大收入和实现最大收入的分割方法，也就是分割后各杆的长度。

先从一个保存价格和分割后各段长度的类入手：RevenueDetails。

UsingClosures/rodCutting.groovy

```
@groovy.transform.Immutable
class RevenueDetails {
    int revenue
    ArrayList splits
}
```

要找出一个最优的最大收入，需要尝试各种组合。对于一个给定的长度，其最大收入，就是以不同方式对杆进行分割，分别计算所得收入，然后求出的最大值。例如，我们可以先将杆分割为一段2英寸的和一段25英寸的。不过这种分割方式可以获得的最大收入，不是这两种长度的零售价格之和，而是它们分别可以获得的最大收入之和。可见解决方案中已经浮现出递归结构。

为确定25英寸的最大收入，我们将其分割成更小的段，然后依次重复地计算较小的段（比如长2英寸的段）的最大收入。为节省时间，可以把这种重复计算记录下来，我们一会就会看到。

首先为杆切割问题实现简单递归方案。

UsingClosures/rodCutting.groovy

```
def cutRod(prices, length) {
    if(length == 0)
        new RevenueDetails(0, [])
    else {
        def maxRevenueDetails = new RevenueDetails(Integer.MIN_VALUE, [])
        for(rodSize in 1..length) {
            def revenueFromSecondHalf = cutRod(prices, length - rodSize)
            def potentialRevenue = new RevenueDetails(
                prices[rodSize] + revenueFromSecondHalf.revenue,
                revenueFromSecondHalf.splits + rodSize)
            if(potentialRevenue.revenue > maxRevenueDetails.revenue)
                maxRevenueDetails = potentialRevenue
        }
        maxRevenueDetails
    }
}

timeIt desiredLength, { length -> cutRod(rodPrices, length) }
```

`cutRod()`方法接受`prices`和`length`这两个参数，返回一个`RevenueDetails`实例，其中保存了给定长度的最大收入和各段的可能长度。如果`length`为0，则递归结束。给定一个长度，我们会尝试尽可能多的分割组合：1和`length - 1`、2和`length - 2`以及3和`length - 3`，诸如此类，同时求出每种组合的最大收入。对于每对长度，递归地调用`cutRod()`方法。

这是一种简单的递归，每次调用该方法都会执行全部计算。为同样的长度重复调用该方法，会反复地重新计算结果。以27英寸长的杆为例，运行这段代码，看一下最大收入、最优分割长度以及这段代码求得这些结果所消耗的时间。

```
Max revenue for 27 is RevenueDetails(43, [5, 5, 5, 5, 5, 2])
Time taken 162.89431500 seconds
```

要获得43美元的最大收入，需要将杆分割成六段。这个程序用了两分多钟才求出该结果。我们可以使用记忆化改进其速度。出人意料的是，只需要很小的改动就能实现——将该函数转换为一个闭包，然后在闭包上调用`memoize()`方法：

4

```
UsingClosures/rodCutting.groovy
def cutRod
cutRod = { prices, length ->
    if(length == 0)
        new RevenueDetails(0, [])
    else {
        def maxRevenueDetails = new RevenueDetails(Integer.MIN_VALUE, [])
        for(rodSize in 1..length) {
            def revenueFromSecondHalf = cutRod(prices, length - rodSize)
            def potentialRevenue = new RevenueDetails(
                prices[rodSize] + revenueFromSecondHalf.revenue,
                revenueFromSecondHalf.splits + rodSize)
            if(potentialRevenue.revenue > maxRevenueDetails.revenue)
                maxRevenueDetails = potentialRevenue
        }
        maxRevenueDetails
    }
}.memoize()

timeIt desiredLength, { length -> cutRod(rodPrices, length) }
```

在将函数转换为闭包，并在其上调用了`memoize()`方法之后，我们将结果保存到了`cutRod`变量中。通过这些步骤，我们创建了`Memoize`类的一个专用实例。该实例中有一个指向所提供闭包的引用，还有一个结果的缓存。当我们调用该闭包时，该实例会在返回结果之前将响应缓存下来。在随后的调用中，如果对应某个参数已经有了相应的缓存下来的结果值，则返回该值。

运行修改版的代码，并确认所得的最大收入和各段长度与之前版本相同。该版本应该会节省很多时间：

```
Max revenue for 27 is RevenueDetails(43, [5, 5, 5, 5, 5, 2])
Time taken 0.01171600 seconds
```

记忆化版本产生的结果相同，但是在消耗的时间上，前面的简单递归版本花了2分多钟，而该版本只用了1/100秒。

记忆化技术是以空间换取速度。我们看到了速度的提升。这种技术消耗的空间取决于使用不重复的参数值调用递归方法的次数。对于较大的问题规模，内存需求可能会急剧增长。Groovy对此非常敏感，所以我们提供了一些选项。简单地调用`memoize()`，该方法会使用一个没有限制的缓存。我们可以使用`memoizeAtMost()`方法代替它。该方法会限制缓存的大小，而且当达到该限制时，最近最少使用（Least Recently Used，LRU）的值会从缓存中移出，以容纳新的值。

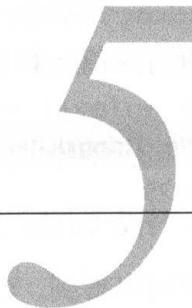
还可以使用诸如`memoizeAtLeast()`和`memoizeAtLeastBetween()`之类的变种，前者可以设置缓存的下限，而后者可以同时设置下限和上限。除了管理缓存，`memoize()`函数的实现还提供了线程安全性；我们可以安全地从多个线程访问缓存。

我们看到，动态类型的Groovy使得动态规划实现起来非常轻松。

本章介绍了Groovy中最重要的概念之一——闭包，这也是将来会反复用到的概念。我们现在已经知道如何在动态上下文中使用闭包，也理解了闭包如何分派方法调用。后面的章节会介绍一些出现闭包的例子，因此我们有的是机会欣赏它的魅力。

在编写程序时，字符串司空见惯，而Groovy为处理它们提供了极大的便利性和灵活性。在下一章中，我们将探讨Groovy提供的涵盖从创建字符串到格式化表达式等不同功能的字符串处理设施。

使用字符串



我们都知道，在Java中使用字符串是种痛苦。本以为像字符串这么基础的东西应该更容易些，但事与愿违。基本的字符串操作，以字符串形式表示多个变量或表达式的求值，甚至连创建跨多行的字符串这么简单的事情，都很费劲。Groovy来拯救我们了！它终结了这些字符串操作的痛苦。通过提供特殊的操作符，利用正则表达式对字符串进行模式匹配也变得更容易了。

本章就依次介绍一下Groovy字符串的基础知识。

5.1 字面常量与表达式

5

Groovy中可以使用单引号创建字符串字面常量，比如'hello'。而在Java中，'a'是一个char，"a"才是一个String对象。Groovy中没有这样的分别。在Groovy中，二者都是String类的实例。如果想显式地创建一个字符，只需要输入'a' as char。当然，如果有任何方法调用需要的话，Groovy可能会隐式地创建Character对象。

对于字符串字面常量中可以放什么，Groovy也很灵活。只要你想，双引号都可以放到字符串中：

WorkingWithStrings/Literals.groovy

```
println 'He said, "That is Groovy"'
```

由输出可见，Groovy处理得相当好：

```
He said, "That is Groovy"
```

来看看使用单引号创建的对象的类型：

WorkingWithStrings/Literals.groovy

```
str = 'A string'  
println str.getClass().name
```

从输出中可以看到，该对象就是常见的String：

```
java.lang.String
```

Groovy会把使用单引号创建的String看作一个纯粹的字面常量。因此，如果在里面放了任何表达式，Groovy并不会计算它们；相反，它就是按所提供的字面内容来使用它们。要对String中的表达式进行求值运算，则必须使用双引号，一会儿就会介绍。

WorkingWithStrings/Literals.groovy

```
value = 25
println 'The value is ${value}'
```

从输出中可以看到，Groovy没有对value进行求值计算：

```
The value is ${value}
```

Java的String是不可变的，Groovy也信守这种不可变性。一旦创建了一个String实例，就不能通过调用更改器等方法修改其内容了。可以使用[]操作符读取一个字符；不过不能修改，从下面的代码中可以看到：

WorkingWithStrings/Literals.groovy

```
str = 'hello'
println str[2]
try {
    str[2] = '!'
} catch(Exception ex) {
    println ex
}
```

尝试修改String导致了一个错误：

```
l
groovy.lang.MissingMethodException: No signature of method:
    java.lang.String.putAt() is applicable for argument types:
        (java.lang.Integer, java.lang.String) values: [2, !]
...

```

可以使用双引号（" "）或正斜杠（//）创建一个表达式^①。不过，双引号经常用于定义字符串表达式，而正斜杠则用于正则表达式。下面是一个创建表达式的例子：

WorkingWithStrings/Expressions.groovy

```
value = 12
println "He paid \$${value} for that."
```

Groovy会计算该表达式，我们在输出中会看到：

```
He paid $12 for that.
```

变量value在字符串内被计算求值了。这里使用了转义字符（\）来打印\$符号，因为Groovy

^① 这里指的是一前一后两个正斜杠，将字符串内容包围在内。——译者注

会将\$符号用于嵌入表达式。如果定义字符串时使用的是正斜杠，而非双引号，则不必转义\$。如果表达式是一个像value这样的简单变量名，或者是一个简单的属性存取器（accessor），则包围表达式的{}是可选的。因此，我们可以把语句`println "He paid \$\$value for that."`写作`println "He paid \$\$value for that."`或`println (/He paid \$\$value for that/)`。尝试去掉表达式中的{}，看看Groovy是否会报错。需要的时候我们总是可以加上的。

Groovy支持惰性求值（lazy evaluation），即把一个表达式保存在一个字符串中，稍后再打印。来看一个例子：

WorkingWithStrings/Expressions.groovy

```
what = new StringBuilder('fence')
text = "The cow jumped over the $what"
println text

what.replace(0, 5, "moon")
println text
```

通过输出看看Groovy是如何解析该表达式的：

```
The cow jumped over the fence
The cow jumped over the moon
```

当打印text中的字符串表达式时，使用的是what所指对象的当前值。因此，第一次打印text时，得到的是“*The cow jumped over the fence*”。在修改了StringBuilder中的值之后，再打印该字符串表达式时，我们并没有修改text的内容，但得到的输出不同，这次是来自流行的儿歌*Hey Diddle Diddle*中的那句“*The cow jumped over the moon*”。

从这种行为可以看出，使用单引号创建的字符串和使用双引号或正斜杠创建的字符串不同。前者是普通的java.lang.String，而后者有些特殊。Groovy的开发者以他们奇怪的幽默感称其为GString，即Groovy字符串的简称。下面看一下使用不同语法创建的对象的类型：

WorkingWithStrings/Expressions.groovy

```
def printClassInfo(obj) {
    println "class: ${obj.getClass().name}"
    println "superclass: ${obj.getClass().superclass.name}"
}

val = 125
printClassInfo ("The Stock closed at ${val}")
printClassInfo (/The Stock closed at ${val}/)
printClassInfo ("This is a simple String")
```

从输出中可以看到所创建对象的类型：

```
class: org.codehaus.groovy.runtime.GStringImpl
superclass: groovy.lang.GString
```

```

class: org.codehaus.groovy.runtime.GStringImpl
superclass: groovy.lang.GString
class: java.lang.String
superclass: java.lang.Object

```

Groovy并不会简单地因为使用了双引号或正斜杠就创建一个GString实例。它会智能地分析字符串，以确定该字符串是否可以使用一个简单的普通String蒙混过关。在这个例子中，最后一次调用printClassInfo()时，即使我们使用了双引号来创建字符串，但该参数还是一个String实例。

很快你就会熟悉Groovy中不同字符串类型的无缝互动。下一节我们将会看到，使用这些字符串时也必须慎重。

5.2 GString 的惰性求值问题

GString表达式的结果取决于表达式中是否使用了值或引用。如果表达式组织得不够仔细，结果可能会令人惊讶。我在学习Groovy字符串操作时就遇到过多次，现在学习这部分内容，可以避免你像我那样栽跟头。下面是上一节中工作得很好的那个例子：

```

WorkingWithStrings/LazyEval.groovy
what = new StringBuilder('fence')
text = "The cow jumped over the $what"
println text

what.replace(0, 5, "moon")
println text

```

这段代码的输出看上去相当合理：

```

The cow jumped over the fence
The cow jumped over the moon

```

text这个GString实例中包含了变量what。该表达式会在每次被打印时，也就是在其上调用toString()方法时求值。如果修改了what所指向的StringBuilder对象的值，在打印时会有所体现。这看上去很合理吧？

遗憾的是，如果修改的是引用what，而不是被引用对象的属性，结果将出乎意料。但如果对象是不可变的，修改引用就是很自然的做法。下面的例子说明了这个问题：

```

WorkingWithStrings/LazyEval.groovy
price = 684.71
company = 'Google'
quote = "Today $company stock closed at $price"
println quote

```

```

stocks = [Apple : 663.01, Microsoft : 30.95]

stocks.each { key, value ->
    company = key
    price = value
    println quote
}

```

这里使用嵌入的变量company和price在变量quote中保存了一个表达式。第一次打印时，这段代码正确打印了谷歌及其股价。我们还想使用这个表达式来打印其他一些公司的股价，为实现该功能，对stocks这个Map进行迭代。在闭包内，将公司名作为键，将股价作为值。然而，当打印quote时，如下所示，结果并不是我们想要的。在同事们挑起另一场“谷歌已经接管世界”的争论之前，必须修复这个问题。

```

Today Google stock closed at 684.71
Today Google stock closed at 684.71
Today Google stock closed at 684.71

```

先弄清楚它为什么没有按预期方式工作，才能找出解决方案。这里在定义quote这个GString时，使用了company和price两个变量，前者绑定的是值为“Google”的一个String，后者绑定的是一个Integer，其中保存高得惊人的股价。可以将company和price引用（它们指向的均为不可变对象）修改为任何想指向的其他对象，但是不能修改GString实例所绑定的内容。

“The cow jumping over...”可以工作，是因为修改的是GString所绑定的对象。然而这个例子中却不可行。因为不可变，所以无法修改。那如何解决呢？——让GString重新计算引用，毕竟，正如计算机科学家David Wheeler所说：“计算机科学领域的任何问题都可以通过增加一个间接层来解决。”

在修复该问题之前，先花点时间理解一下GString表达式是如何求值的。当对一个GString实例求值时，如果其中包含一个变量，该变量的值会被简单地打印到一个Writer，通常是一个StringWriter。然而，如果GString中包含的是一个闭包，而非变量，该闭包就会被调用。如果闭包接收一个参数，GString会把Writer对象当作一个参数发送给它。如果闭包不接收任何参数，GString会简单地调用该闭包，并打印我们想返回给Writer的结果。如果闭包接收的参数不止一个，调用则会失败，并抛出一个异常，所以别这么做。

下面使用这些知识来解决我们的表达式求值问题。第一次尝试：

WorkingWithStrings/LazyEval.groovy

```

companyClosure = { it.write(company) }
priceClosure = { it.write("$price") }
quote = "Today ${companyClosure} stock closed at ${priceClosure}"
stocks.each { key, value ->
    company = key
    price = value
}

```

```
    println quote
}
```

运行代码看一下输出：

```
Today Apple stock closed at 663.01
Today Microsoft stock closed at 30.95
```

输出合乎预期，但这段代码看上去还不够出色。即使最终版本不想采用这种方式，但是通过观察这个例子，还是会有两方面的收获。首先，可以看到实际发生了什么：当表达式需要求值/打印时，`GString`会调用闭包；其次，如果想做些计算，而不是仅仅显示一下属性的值，也知道了该怎么做。

如前文所述，如果闭包没有任何参数，可以去掉`it`参数，`GString`会使用我们返回的内容。我们已经知道如何创建一个没有参数的闭包——使用`{->}`语法来定义。现在重构前面的代码：

WorkingWithStrings/LazyEval.groovy

```
companyClosure = {-> company }
priceClosure = {-> price }
quote = "Today ${companyClosure} stock closed at ${priceClosure}"
stocks.each { key, value ->
    company = key
    price = value
    println quote
}
```

这是重构版本的输出：

```
Today Apple stock closed at 663.01
Today Microsoft stock closed at 30.95
```

这个版本略胜一筹，但是我们不想单独定义闭包。对于这种简单情形，我们希望代码是自包含的；而如果有更多计算，我们也愿意编写一个单独的闭包。下面是解决该问题的自包含代码（我们称其为“苹果和微软试图接管世界”问题）：

WorkingWithStrings/LazyEval.groovy

```
quote = "Today ${-> company } stock closed at ${-> price }"

stocks.each { key, value ->
    company = key
    price = value
    println quote
}
```

这个简洁的版本会与上一版本产生同样的输出：

```
Today Apple stock closed at 663.01
Today Microsoft stock closed at 30.95
```

`GString`的惰性求值是个非常强大的概念。不过要小心，不要在字符串上栽跟头。如果希望改变表达式中使用的引用，而且希望它们的当前值被用于惰性求值中，请必须记住，不要在表达式中直接替换它们，而要使用一个无参闭包。

我们看到了Groovy的字符串操作和格式化的优雅，不过现在触及的只是一点皮毛而已。在Java中创建多行字符串既可怕又麻烦，下一节将介绍Groovy是如何简化这一过程的。

5.3 多行字符串

要在Java中创建一个多行字符串，我们不得不使用像`str += ...`这样的代码，使用+操作符连接多行，或者多次调用`StringBuilder`或`StringBuffer`的`append()`方法。

我们不得不使用大量的转义字符，这时候我们会抱怨：“一定会有更好的方法。”Groovy就有。可以通过将字符串包含在3个单引号内（' '' ''）来定义多行字面常量，而且Groovy就是这样支持here文档^①：

WorkingWithStrings/MultilineStrings.groovy

```
memo = '''Several of you raised concerns about long meetings.  
To discuss this, we will be holding a 3 hour meeting starting  
at 9AM tomorrow. All getting this memo are required to attend.  
If you can't make it, please have a meeting with your manager to explain.  
'''
```

```
println memo
```

下面是这段代码创建的多行字符串：

```
Several of you raised concerns about long meetings.  
To discuss this, we will be holding a 3 hour meeting starting  
at 9AM tomorrow. All getting this memo are required to attend.  
If you can't make it, please have a meeting with your manager to explain.
```

就像可以使用双引号创建含有表达式的`GString`那样，也可以使用3个双引号创建包含表达式的多行字符串。

WorkingWithStrings/MultilineStrings.groovy

```
price = 251.12

message = """We're very pleased to announce  
that our stock price hit a high of \$${price} per share
```

^① here文档 (here documents)，又称作heredoc、hereis、here-字串或here-脚本，是一种在命令行shell (如sh、csh、ksh、bash、PowerShell和zsh)和程序语言(像Perl、PHP、Python和Ruby)里定义一个字串的方法。参见<http://zh.wikipedia.org/wiki/Here%E6%96%87%E6%A1%A3>。——译者注

```
on December 24th. Great news in time for...
"""
println message
```

Groovy会计算其中的表达式，如输出所示：

```
We're very pleased to announce
that our stock price hit a high of $251.12 per share
on December 24th. Great news in time for...
```

我每个月都要写份通讯。几年前，我决定将用于发送邮件通知的程序转换到Groovy。Groovy能够在多行字符串中嵌入一些值的能力正好派上用场。Groovy甚至使我写的通讯更容易被当作垃圾邮件了！（只是开玩笑，别当真。）

来看一个使用了这个特性的例子。假设有一个语言及其作者的映射，我们想为其创建一个XML表示。下面是一种实现方式：

WorkingWithStrings/CreateXML.groovy

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

content = ''
langs.each { language, author ->
    fragment = """
        <language name="${language}">
            <author>$author</author>
        </language>
    """
    content += fragment
}
xml = "<languages>${content}</languages>"
println xml
```

这太可喜了，但是在17.1节我们将看到更令人惊叹的XML生成器。下面是使用多行字符串表达式产生的XML输出：

```
<languages>
    <language name="C++">
        <author>Stroustrup</author>
    </language>

    <language name="Java">
        <author>Gosling</author>
    </language>

    <language name="Lisp">
        <author>McCarthy</author>
    </language>
</languages>
```

这里使用嵌入了表达式的多行字符串创建了想要的内容，该内容是通过迭代包含数据的映射生成的。

看过了创建字符串的几种方式，下一节将探讨Groovy为操纵字符串提供的便捷方法。

5.4 字符串便捷方法

`String`的`execute()`方法的确好用，它帮我们创建了一个`Process`对象，所以只需要几行代码就可以执行系统级进程（参见2.1.3节）。

想要玩转`String`，还有其他可选方法。例如下面的代码，它使用了`String`的一个重载操作符：

WorkingWithStrings/StringConvenience.groovy

```
str = "It's a rainy day in Seattle"
println str

str -= "rainy "
println str
```

输出说明了这一重载的操作符的效果：

```
It's a rainy day in Seattle
It's a day in Seattle
```

-=操作符对于操纵字符串很有用，它会将左侧的字符串中与右侧字符串相匹配的部分去掉。Groovy在`String`类上添加的`minus()`方法使其成为可能（参见2.8节）。Groovy还向`String`类添加了其他便捷方法：`plus()` (+)、`multiply()` (*)、`next()` (++)、`replaceAll()`和`tokenize()`等。^①

也可以在一系列`String`上迭代，如下所示：

WorkingWithStrings/StringRange.groovy

```
for(str in 'held'..'helm') {
    print "${str} "
}
println ""
```

这段代码生成的序列如下：

```
held hele helf helg helh heli helj helk hell helm
```

这里使用的仍然是`java.lang.String`。Groovy添加的所有这些设施可以帮助我们快速完成工作。

^① <http://groovy.codehaus.org/groovy-jdk/java/lang/String.html>

现在我们知道了如何提取出部分字符串。核心的程序员往往也会接触到正则表达式，Groovy同样令事情更简单了，下一节即将介绍。

5.5 正则表达式

JDK包`java.util.regex`包含了使用正则表达式（RegEx）进行模式匹配的API。关于RegEx的详细讨论，请参考Jeffrey Friedl的*Mastering Regular Expressions*[Fri97]一书。别的方法不说，`String`类的`replaceFirst()`和`replaceAll()`方法就充分利用了RegEx模式匹配。为使编程使用RegEx更为容易，Groovy添加了一些操作符和符号。

`~`操作符可以方便地创建RegEx模式。这个操作符映射到`String`类的`negate()`方法：

WorkingWithStrings/RegEx.groovy

```
obj = ~"hello"
println obj.getClass().name
```

下面输出说明了所创建实例的类型：

```
java.util.regex.Pattern
```

前面的例子说明，将`~`应用于`String`，会创建一个`Pattern`实例。我们可以使用正斜杠、单引号或双引号来创建RegEx。正斜杠有一个优势：不必对反斜杠进行转义。因此，`/\d*\w*/`与`"\\d*\\w*"`等价，但是更优雅。

为方便匹配正则表达式，Groovy提供了一对操作符：`=~`和`==~`。下面就来探索一下这两个操作符之间的差别，以及它们的具体功能：

WorkingWithStrings/RegEx.groovy

```
pattern = ~"(G|g)roovy"
text = 'Groovy is Hip'
if (text =~ pattern)
    println "match"
else
    println "no match"

if (text ==~ pattern)
    println "match"
else
    println "no match"
```

运行这段代码，可以看出两个操作符之间的差别：

```
match
no match
```

`=~`执行RegEx部分匹配，而`==~`执行RegEx精确匹配。因此，在前面的代码示例中，第一个模式匹配报告的是`match`，而第二个报告的是`no match`。

`=~`操作符会返回一个`Matcher`对象，它是一个`java.util.regex.Matcher`实例。Groovy对`Matcher`的布尔求值处理不同于Java，只要至少有一个匹配，它就会返回`true`（参见2.7节）。如果有多个匹配，则`matcher`会包含一个匹配的数组。这有助于快速获得匹配给定RegEx的文本中的部分内容。

WorkingWithStrings/RegEx.groovy

```
matcher = 'Groovy is groovy' =~ /(G|g)roovy/
print "Size of matcher is ${matcher.size()}"
println "with elements ${matcher[0]} and ${matcher[1]}."
```

前面代码会报告`Matcher`的详细信息，具体如下：

```
Size of matcher is 2 with elements [Groovy, G] and [groovy, g].
```

可以使用`replaceFirst()`方法或`replaceAll()`方法方便地替换匹配的文本（顾名思义，前者仅替换第一个匹配，而后者会替换所有匹配）。

WorkingWithStrings/RegEx.groovy

```
str = 'Groovy is groovy, really groovy'
println str
result = (str =~ /groovy/).replaceAll('hip')
println result
```

5

原始的文本和替换后的文本分别如下：

```
Groovy is groovy, really groovy
Groovy is hip, really hip
```

作为总结，下面列出了与RegEx相关的Groovy操作符：

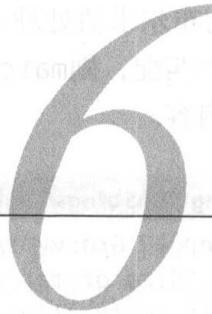
- 要从字符串创建一个模式，使用`~`操作符。
- 要定义一个RegEx，使用正斜杠，像`/[G|g]roovy/`中这样。
- 要确定是否存在匹配，使用`=~`。
- 对于精确匹配，使用`==~`。

这一章介绍了Groovy是如何将字符串的创建和使用变得比在Java中容易得多的。创建多行字符串和带表达式的字符串也是易如反掌。我们也看到Groovy如何简化了RegEx的使用。当着手使用普通字符串操作及正则表达式时，Groovy的字符串会让我们体会到改进。

在编程中，对象的集合也和处理字符串一样基础。Groovy利用闭包提供的便捷与流畅增强了JDK的集合类API，下一章我们将会看到。

第6章

使用集合类



编程时会经常使用集合类。Java开发包（JDK）有很多有用的集合类，Groovy扩展了它们，使它们用起来更方便了。例如，与传统的for循环相比，内部迭代器更简洁、更易用，而且出错的可能性更小。通常可以使用一个不同的专用迭代器find从集合中挑选出一个元素。如果要挑选出几个匹配的元素，只需要简单地把find改为findAll，剩下的代码还一样——这样很简洁，而且没有引入新的集合类时所要面对的负担。一旦熟悉了Groovy中的集合类，再回过头来使用Java的API可就难了。别说我没警告过你！

本章将使用JDK的集合类，不过学习的是Groovy提供的轻量级的、优美自然的方法。首先从List这种有序集合上的各种迭代器和便捷方法入手，之后再来看一下Map这种键值对关联集合上提供的类似方法。

6.1 使用 List

在Groovy中创建一个java.util.ArrayList实例要比在Java中容易。我们不必使用new或指明类名，而是可以简单地列出想包含在List中的初始值，如下所示：

WorkingWithCollections/CreatingArrayList.groovy

```
lst = [1, 3, 4, 1, 8, 9, 2, 6]
println lst
println lst.getClass().name
```

来看一下这个ArrayList的内容以及它的类型，Groovy的输出如下：

```
[1, 3, 4, 1, 8, 9, 2, 6]
java.util.ArrayList
```

在Groovy中声明一个列表时，引用lst指向的是一个java.util.ArrayList实例，如前面的输出所示。

[]操作符可用于获取List中的元素，如下面的例子所示：

WorkingWithCollections/CreatingArrayList.groovy

```
println lst[0]
println lst[lst.size() - 1]
```

输出显示了列表中的第一个元素和最后一个元素的值：

```
1
6
```

其实不必费那么大劲，跳好几个数去获得列表的最后一个元素，Groovy有更简单的方式。可以使用负的索引值，这时Groovy将从右侧开始遍历，而不是从左侧开始：

WorkingWithCollections/CreatingArrayList.groovy

```
println lst[-1]
println lst[-2]
```

这段代码也获得了该列表的最后两个元素，在输出中会看到：

```
6
2
```

甚至可以使用Range对象获得集合中的几个连续的值：

WorkingWithCollections/CreatingArrayList.groovy

```
println lst[2..5]
```

从位置2开始，列表中的4个连续的值如下：

```
[4, 1, 8, 9]
```

Range中还可以使用负的索引值，代码如下，其结果与前面代码相同：

WorkingWithCollections/CreatingArrayList.groovy

```
println lst[-6...-3]
```

快速地查看一下`lst[2..5]`实际返回的是什么：

WorkingWithCollections/CreatingArrayList.groovy

```
subLst = lst[2..5]
println subLst.dump()
subLst[0] = 55
println "After subLst[0]=55 lst = $lst"
```

我们会看到`dump()`方法所报告的该实例的详细信息，以及修改之后的列表：

```
<java.util.ArrayList$SubList@fedbf parent=[1, 3, 4, 1, 8, 9, 2, 6]
parentOffset=2 offset=2 size=4 this$0=[1, 3, 4, 1, 8, 9, 2, 6] modCount=1>
After subLst[0]=55 lst = [1, 3, 55, 1, 8, 9, 2, 6]
```

如果使用一个像`2..5`这样的Range作为索引，`java.util.ArrayList`会返回一个指向原来列表部分内容的实例^①。所以请注意，得到的并非副本，如果修改了其中一个列表的元素，另一个也会受到影响。

可以看到Groovy是如何让List的应用编程接口（Application Programming Interface，API）变得更简单的。我们使用的是同样的、存在已久的ArrayList，但是当以Groovy之角度去看时，它漂亮和轻便了许多。

Groovy提供的便利远不止创建列表这么一点，下一节将介绍更多内容。

6.2 迭代 ArrayList

我们经常要对一组值进行导航或迭代。Groovy提供了优雅的列表迭代方式，还支持迭代时在其中的值上优雅地执行操作。

6.2.1 使用 List 的 each 方法

第4章中介绍过Groovy为迭代集合提供的便捷方式。这个迭代器（以`each()`命名的方法）也称内部迭代器。

内部迭代器与外部迭代器

我们习惯于C++和Java等语言中的外部迭代器，它们能让其用户或客户来控制迭代。我们必须检查迭代是不是要结束了，并且需要显式地移动到下一个元素。

内部迭代器在支持闭包的语言中很流行，迭代器的用户或客户不控制迭代，他们只需要提供一个要在集合中的每个元素上执行的代码块。

内部迭代器更容易使用，我们不必控制迭代。外部迭代器则更为灵活，我们可以更方便地控制迭代顺序、跳过元素、结束迭代或重新迭代等因素。

不要让缺乏灵活性这种表象阻碍我们使用内部迭代器。为了提供更多的灵活性和便捷性，内部迭代器的实现者往往需要付出额外的努力。以List为例，要控制迭代的灵活性，这要通过本章将要介绍的不同便捷方法的形式来实现。

下面创建一个ArrayList，然后使用`each()`方法对它进行迭代。

^① 在较新的Groovy版本中，这种行为有所改变，所以读者一定要明确所使用的版本，具体信息请参考相应文档。

——译者注

WorkingWithCollections/IteratingArrayList.groovy

```
lst = [1, 3, 4, 1, 8, 9, 2, 6]

lst.each { println it }
```

在迭代时打印出每个元素，输出如下：

```
1
3
4
1
8
9
2
6
```

可以使用**reverseEach()**方法反向迭代元素。如果关注迭代过程中的计数或索引，可以使用**eachWithIndex()**方法。

还可以执行其他操作，比如对闭包中的元素求和（参见4.3节），如下所示：

WorkingWithCollections/IteratingArrayList.groovy

```
total = 0
lst.each { total += it }
println "Total is $total"
```

下面是这段代码的执行结果：

```
Total is 34
```

假设想把集合中的每个元素变为原来的2倍，可以使用**each()**方法试试：

WorkingWithCollections/IteratingArrayList.groovy

```
doubled = []
lst.each { doubled << it * 2 }

println doubled
```

结果如下：

```
[2, 6, 8, 2, 16, 18, 4, 12]
```

这里创建了一个名为**doubled**的空**ArrayList**来保存结果。在对集合进行迭代时，将每个元素加倍，并使用**<<**操作符（映射到**leftShift()**方法）将所得的值放到结果中。

如果想在集合中的每个元素上执行一些操作，**each()**方法是一个不错的选择，但如果想让这些操作生成一些结果，则可以求助其他方法。

6.2.2 使用 List 的 collect 方法

如果想在集合中的每个元素上执行操作并返回一个结果集合，Groovy提供了一个简单的解决方案——`collect()`方法，下面我们会看到：

WorkingWithCollections/IteratingArrayList.groovy

```
println lst.collect { it * 2 }
```

`collect()`方法和`each()`一样，会在集合中的每个元素上调用传入的闭包。不过它会把闭包的返回值收集（即`collect`一词的本意）到一个集合中，最后返回这个生成的结果集合。前面例子中的闭包返回给定值的2倍，因为闭包有一个隐式的`return`。得到了一个`ArrayList`，其中的元素值是输入值的2倍，在输出中会看到：

```
[2, 6, 8, 2, 16, 18, 4, 12]
```

如果想在集合的每个元素上执行操作，可以使用`each()`。然而，如果想得到一个进行了这类计算的结果集合，则要使用`collect()`方法。

可用的还不止这两个内部迭代器，请看下节。

6.3 使用查找方法

我们知道了如何在集合上执行迭代，以及如何在每个元素上执行操作。然而，如果想搜索特定的元素，`each()`和`collect()`都不方便。此时应该使用`find()`，像下面这样：

WorkingWithCollections/Find.groovy

```
lst = [4, 3, 1, 2, 4, 1, 8, 9, 2, 6]
println lst.find { it == 2 }
```

这段代码会挑出集合中第一个等于2的元素，在输出中我们会看到：

```
2
```

这段代码会查找一个集合中与2这个值匹配的对象，`find()`会找到第一次出现的匹配对象。在这种情况下，它返回的是在位置3的对象。就像`each()`方法一样，`find()`方法也会对集合进行迭代，但是它只会迭代到闭包返回`true`为止。一得到`true`，`find()`就会停止迭代，并将当前的元素返回。如果遍历结束也没得到`true`，则返回`null`。

我们可以在附到`find()`上的闭包中指定任何条件。例如，下面是如何查找第一个大于4的元素：

WorkingWithCollections/Find.groovy

```
println lst.find { it > 4 }
```

这段代码会报告列表中第一个大于4的数：

8

也可以找到列表中出现的所有2。就像`find()`方法的表现类似`each()`一样，`findAll()`方法的表现类似`collect()`：

WorkingWithCollections/Find.groovy

```
println lst.findAll { it == 2 }
```

可以看到在该列表中找到的所有的2：

[2, 2]

在这个例子中，要查找2，`findAll()`方法返回的是对象，而不是位置。如果想查找第一个匹配对象的位置，可以使用`findIndexOf()`方法。

在最简单的情况下，找到所有的2听上去并不是很有用。然而，一般而言，如果正在查找匹配某些标准的对象，我们就会得到那些对象。例如，如果要查找人口数大于某个特定值的所有城市，结果就会是相应城市的列表。回到前面的例子，如果想要所有大于4的数，应该这样做：

WorkingWithCollections/Find.groovy

```
println lst.findAll { it > 4 }
```

这段代码会报告所有大于4的元素：

[8, 9, 6]

本节介绍了如何迭代集合以及从集合中选择元素。因为对集合的操作远不止这么几个，所以Groovy使用更多便捷方法扩展了其流畅性，下面我们会看到。

6

6.4 List 上的其他便捷方法

Groovy向集合类`Collection`添加了很多便捷方法。（具体列表请参考<http://groovy.codehaus.org/groovy-jdk/java/util/Collection.html>。）下面使用我们已经熟悉的方法`each()`来实现一个例子。之后再使用那些可以使代码自包含、表现力更好的方法来重构这个例子。在此过程中，我们将会看到Groovy是如何像函数式编程语言那样将代码块看作一等公民的。

假设有一个字符串集合，然后想计算其中总的字符数。下面是使用`each()`方法实现的一种方式：

WorkingWithCollections/CollectionsConvenienceMethods.groovy

```
lst = ['Programming', 'In', 'Groovy']
```

```
count = 0
```

```
lst.each { count += it.size() }
println count
```

求出的字符数目如下：

19

Groovy往往会为一个任务提供多种解决方式。下面是另一种方式，使用了`collect()`和`sum()`（都是Groovy在JDK的集合类上添加的方法）：

WorkingWithCollections/CollectionsConvenienceMethods.groovy

```
println lst.collect { it.size() }.sum()
```

在`collect()`方法返回的集合上调用了`sum()`方法，这段代码会产生与前一版本同样的输出：

19

代码有点简洁，但却是自包含的：要处理集合中每个单独的元素，而且要获得一个累积的结果，`each()`很有用。然而，如果想在集合的每个元素上执行一些计算，同时把结果保留为一个集合，`collect()`就很有用了。这一点也可以应用于在集合上进行遍历计算的其他操作（比如`sum()`方法）上。

使用`inject()`方法可以实现同样的功能：

WorkingWithCollections/CollectionsConvenienceMethods.groovy

```
println lst.inject(0) { carryOver, element -> carryOver + element.size() }
```

输出如下：

19

`inject()`会对集合中的每个元素调用闭包。在这个例子中，集合中的元素是用参数`element`表示的。`inject()`会把将要注入的一个初始值当做一个参数，并通过`carryOver`参数把它放到第一次对闭包的调用中。之后它会把从闭包获得的结果注入到随后对闭包的调用中。如果想在集合中的每个元素上应用某个计算，获得一个累积的结果，与`collect()`方法相比，我们会首选`inject()`方法。

假设想把集合中的元素连接成一个句子。利用`join()`可以很方便地实现：

WorkingWithCollections/CollectionsConvenienceMethods.groovy

```
println lst.join(' ')
```

下面是连接元素的结果：

Programming In Groovy

`join()`会迭代每个元素，然后将每个元素和作为输入参数给定的字符连接起来。在这个例子

中，作为输入参数给定的是空格字符，因此 `join()` 方法返回了字符串 Programming In Groovy。当我们想连接一个由路径组成的集合时，`join()` 方法就派上用场了，例如使用一个分号（;）构造一个 `classpath`，一个简单的调用即可完成。

可以通过索引替换 List 中的一个元素。下面的代码将 `['Be', 'Productive']` 设置给了元素 0：

WorkingWithCollections/CollectionsConvenienceMethods.groovy

```
lst[0] = ['Be', 'Productive']
println lst
```

这会导致集合中包含一个 List，就像下面这样：

```
[['Be', 'Productive'], 'In', 'Groovy']
```

如果这不是我们想要的，可以使用 `flatten()` 方法将 List 拉平：

WorkingWithCollections/CollectionsConvenienceMethods.groovy

```
lst = lst.flatten()
println lst
```

结果是一个拉平了的单一列表：

```
[Be, Productive, In, Groovy]
```

还可以在 List 上使用 - 操作符 (`minus()`) 方法，像这样：

WorkingWithCollections/CollectionsConvenienceMethods.groovy

```
println lst - ['Productive', 'In']
```

右操作数中的元素会被从左侧的集合中移除。如果提供了一个不存在的元素，不用担心——它会被直接忽略掉。- 操作符很灵活，可以为右操作数提供一个列表或是单个的值。该操作的结果如下：

```
[Be, Groovy]
```

可以使用 `reverse()` 方法得到列表的一份副本，其中的元素是反向排列的。

下面是 Groovy 中的另一个便捷方法：可以很方便地在每个元素上执行操作，而不用显式地使用迭代。

WorkingWithCollections/CollectionsConvenienceMethods.groovy

```
println lst.size()
println lst*.size()
```

这段代码会打印元素个数，以及每个元素的大小：

4

```
[2, 10, 2, 6]
```

第一次调用size()是在列表上，它会返回4，即列表中当前的元素个数。第二次调用，因为*的影响，即作用于列表中的每个元素（这个例子中是String）的展开操作符（spread operator）的影响，会返回一个List，其中的每个元素分别保存原始集合中相应元素的大小。lst*.size()的作用与lst.collect { it.size() }相同。

最后来看一下如何在方法调用中使用ArrayList。如果一个方法接收很多参数，不同于发送单个的参数，我们可以将一个ArrayList打散作为参数；也就是说，使用*操作符将集合拆成单个对象，下面我们将看到。要使其正常工作，ArrayList的元素个数必须与方法期望的参数个数相同。

WorkingWithCollections/CollectionsConvenienceMethods.groovy

```
def words(a, b, c, d) {
    println "$a $b $c $d"
}

words(*lst)
```

下面是使用展开操作符的结果：

Be Productive In Groovy

本节探索了Groovy用于处理由对象组成的List的设施，下一节将介绍在Groovy中如何使用Map。

6.5 使用 Map 类

Java的java.util.Map在使用键值对的关联集合时很有用。通过使用闭包，Groovy使Map用起来更为简单和优雅。创建一个Map实例也很简单，因为不需要使用new或指明任何类名。只需简单地创建键值对即可：

WorkingWithCollections/UsingMap.groovy

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

println langs.getClass().name
```

通过输出确认一下所创建集合的类名：

`java.util.LinkedHashMap`

这个例子创建了一个散列映射（hash map），以一些编程语言为键，以语言相应的作者为值。键和值用冒号（：）分隔，整个映射放在一个中括号（[]）中。这种简单的Groovy语法创建了一个java.util.LinkedHashMap实例。通过调用getClass()并获得其name属性便可看到。为什么要这么啰嗦地调用getClass()方法，而没有使用JavaBean的约定直接访问class属性？这里有一

个小陷阱。

可以使用`[]`操作符来访问一个键的值，如下面代码所示：

WorkingWithCollections/UsingMap.groovy

```
println langs['Java']
println langs['C++']
```

下面是所请求的两个键的值：

```
Gosling
Stroustrup
```

如果想看点花哨的，Groovy当然不会让我们失望。可以把键用作好像是Map的一个属性，以此访问该键对应的值：

WorkingWithCollections/UsingMap.groovy

```
println langs.Java
```

Groovy会将该键用作一个属性，返回相应的值：

```
Gosling
```

这很巧妙，把键看作好像是对象的一个属性，然后发送该键，很是方便，而且Map会聪明地返回其值。当然，有经验的程序员立即会问：“有什么陷阱？”这个陷阱已经很明显了：我们不能在Map上调用`class`属性，Map会假定`class`这个名字指向的是一个(不存在的)键，而返回`null`。很明显，后面再调用`class`的`name`属性，因为是在`null`上调用，就会失败了。当调用`class`属性时，Map和其他一些类的实例不会返回`Class`元对象。为避免结果出乎意料，在实例上要总是使用`getClass()`方法，而不是`class`属性。

所以必须调用`getClass()`方法。但是C++这个键又怎么样呢？我们来试一下：

WorkingWithCollections/UsingMap.groovy

```
println langs.C++ // 不合法代码
```

下面是我们得到的输出：

```
java.lang.NullPointerException: Cannot invoke method next() on null object
```

这是什么意思？我们可能会放弃这个代码示例，然后说C++不管在哪都要出问题。

但是，这个问题实际上缘自与Groovy的另一个特性——操作符重载——的冲突(参见2.8节)。Groovy会把前面的请求看作要获取键C的值，而这个键不存在。因此，它会返回`null`，然后尝试调用`++`操作符所映射的`next()`方法。幸运的是，像这样的特殊情况有一个变通方案：只需把带有这种会惹麻烦的字符的键值看作一个`String`。

WorkingWithCollections/UsingMap.groovy

```
println langs.'C++'
```

太棒了！终于得到了正确的输出：

```
Stroustrup
```

Groovy添加了另一个创建Map的便捷方法。当定义一个Map时，对于规规矩矩的键名，可以省略其引号。例如，我们不使用键上的引号，重写编程语言和它们的作者这个Map：

WorkingWithCollections/UsingMap.groovy

```
langs = ['C++' : 'Stroustrup', Java : 'Gosling', Lisp : 'McCarthy']
```

我们知道了如何创建一个Map，以及如何访问这种集合中的单个值。下一节将介绍如何在Map这种集合上迭代。

6.6 在 Map 上迭代

Groovy向Map添加了很多便捷方法^①。我们可以在一个Map上迭代，就像在ArrayList上迭代那样（参见6.2节）。

Map也有自己的each()和collect()方法。

6.6.1 Map 的 each 方法

来看一个使用each()方法的例子：

WorkingWithCollections/NavigatingMap.groovy

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']

langs.each { entry ->
    println "Language ${entry.key} was authored by ${entry.value}"
}
```

输出如下：

```
Language C++ was authored by Stroustrup
Language Java was authored by Gosling
Language Lisp was authored by McCarthy
```

如果附到each()上的闭包只接收一个参数，each()会把一个MapEntry实例发送给该参数。如果想单独获得键和值，只需要在闭包中提供两个参数，如下面的例子所示：

^① <http://groovy.codehaus.org/groovy-jdk/java/util/Map.html>

WorkingWithCollections/NavigatingMap.groovy

```
langs.each { language, author ->
    println "Language $language was authored by $author"
}
```

下面是使用两个参数的闭包在Map上迭代的输出：

```
Language C++ was authored by Stroustrup
Language Java was authored by Gosling
Language Lisp was authored by McCarthy
```

这个代码示例使用each()方法在langs集合上迭代，迭代时会使用一个键和一个值调用闭包。在闭包中，我们分别使用变量名language和author引用这两个参数。

类似地，对于其他方法，比如collect()、find()等，如果只想要MapEntry，就使用一个参数；如果想分别获得键和值，则使用两个参数。

6.6.2 Map 的 collect 方法

下面试试Map中的collect()方法。首先，它与ArrayList中的collect()方法类似的是，都返回一个列表。不过，如果想让Map的collect()向我们的闭包发送一个MapEntry，就定义一个参数；否则就定义两个参数，分别表示键和值，如下所示：

WorkingWithCollections/NavigatingMap.groovy

```
println langs.collect { language, author ->
    language.replaceAll("[+]", "P")
}
```

6

代码返回如下的列表：

[CPP, Java, Lisp]

在前面的代码中，我们创建了一个键的列表，键中出现的所有+都替换成了字符P。

我们可以方便地将Map中的数据转换成其他表示形式。例如，在17.1节我们会看到创建一个XML表示是多么容易。

6.6.3 Map 的 find 和 findAll 方法

Groovy也向Map添加了find()和findAll()方法。我们看一个例子：

WorkingWithCollections/NavigatingMap.groovy

```
println "Looking for the first language with name greater than 3 characters"
entry = langs.find { language, author ->
    language.size() > 3
}
```

```

}
println "Found $entry.key written by $entry.value"

```

使用`find()`方法的输出如下：

```

Looking for the first language with name greater than 3 characters
Found Java written by Gosling

```

`find()`方法接收一个闭包，而该闭包接收键和值（再次强调，要接收`MapEntry`则使用一个参数）。与`ArrayList`中的对应方法类似，如果闭包返回`true`，它会退出迭代。在前面的示例代码中，我们会找到名字多于3个字符的第一门语言。如果直到最后闭包也没有返回`true`，该方法会返回`null`。否则，它会返回`Map`中一个匹配条目的实例。

可以使用`findAll()`方法获得匹配所查找条件的所有元素，如下面例子所示：

WorkingWithCollections/NavigatingMap.groovy

```

println "Looking for all languages with name greater than 3 characters"
selected = langs.findAll { language, author ->
    language.size() > 3
}
selected.each { key, value ->
    println "Found $key written by $value"
}

```

这段代码会报告所有满足给定条件的语言：

```

Looking for all languages with name greater than 3 characters
Found Java written by Gosling
Found Lisp written by McCarthy

```

除了内部迭代器，Groovy还提供了强大的便捷函数，对`Map`中的元素进行选择和分组，下面我们会看到。

6.7 Map 上的其他便捷方法

我们来看一些`Map`上的便捷方法，并以此结束关于集合的讨论。

要取得满足某个给定条件的一个元素，`find()`方法很有用。然而，如果不是想得到元素，而只是想确定集合中是否有任何元素满足某些条件，就要用`any()`方法。

继续使用6.6节中的语言及其作者的例子。可以使用`any()`方法来确定是否在有些语言的名字中包含着非字母的字符：

WorkingWithCollections/NavigatingMap.groovy

```

print "Does any language name have a nonalphabetic character? "
println langs.any { language, author ->
    language =~ "[^A-Za-z]"
}

```

因为键中包含C++，所以代码会报告如下内容：

```
Does any language name have a nonalphanumeric character? true
```

就像之前所讨论的Map上的其他方法一样，any()也接受了一个带2个参数的闭包。这个示例中的闭包使用了一个正则表达式比较（参见5.5节），来确定是否有语言的名字中包含非字母的字符。

any()方法会查找至少一个满足给定条件（谓词）的Map中的元素，而every()方法则会检查是否所有的元素都满足给定条件：

WorkingWithCollections/NavigatingMap.groovy

```
print "Do all language names have a nonalphanumeric character? "
println langs.every { language, author ->
    language =~ "[^A-Za-z]"
}
```

输出会告诉我们，是否所有元素都满足给定条件：

```
Do all language names have a nonalphanumeric character? false
```

如果想基于某些标准对Map中的元素进行分组，不必费劲地执行迭代或循环，groupBy()就是干这个的。要做的只是通过闭包说明判断标准。这里有一个例子：friends指向的是一个包含一些朋友的Map，其中很多人名字（First Name，不包括姓）相同。如果想根据名字对朋友进行分组，只需要调用一个groupBy()即可完成，如下面的代码所示。在附到groupBy()之后的闭包中，指明了所要进行的分组——在这个例子中，从全名（Full Name）中取出名字并返回。一般而言，会按我们感兴趣的分类返回属性。例如，如果使用属性firstName和lastName将朋友的名字保存在一个Person对象中，而不是使用一个简单的String，我们可以将闭包写成{ it.firstName }。在下面的代码中，groupByFirstname是一个Map，其中名字作为键，而键相应的值本身又是一个名字和相应全名组成的Map。最后，我们对结果进行迭代，并打印出值：

WorkingWithCollections/NavigatingMap.groovy

```
friends = [ briang : 'Brian Goetz', brians : 'Brian Sletten',
            davidb : 'David Bock', daviddg : 'David Geary',
            scottd : 'Scott Davis', scottl : 'Scott Leberknight',
            stuarth : 'Stuart Halloway']

groupByFirstName = friends.groupBy { it.value.split(' ')[0] }

groupByFirstName.each { firstName, buddies ->
    println "$firstName : ${buddies.collect { key, fullName -> fullName }.join(', ')}"
}
```

下面是每个组中的结果：

Brian : Brian Goetz, Brian Sletten

David : David Bock, David Geary

Scott : Scott Davis, Scott Leberknight

Stuart : Stuart Halloway

最后还要记住两个技巧: Groovy将Map用于具名参数(参见2.3节), 以及使用Map实现接口(参见2.6节)。

在本章中, 我们看到了将闭包引入到Java的集合API中的强大。随着将这些概念应用于项目, 我们将发现, 使用集合类更容易、更快速, 代码也会更简短, 这很有趣。你还会发现, 使用Groovy编程, 会让在其他语言中看似平常的遍历和操纵集合这种任务都充满了激情。

看过了Groovy语言的能力以及这门语言向不同API注入的流畅性, 我们已经为进阶打好了基础。下一部分, 将介绍如何将这门语言有效地应用于诸如处理XML和访问数据库等操作。