

# 目录

|        |                                   |    |
|--------|-----------------------------------|----|
| 第一章    | Groovy 之旅.....                    | 5  |
| 1.1.   | groovy 背景.....                    | 6  |
| 1.1.1. | groovy 是什么? .....                 | 7  |
| 1.1.2. | 与 java 共事: 无缝集成.....              | 7  |
| 1.1.3. | 强劲代码: 一个特性丰富的语言.....              | 9  |
| 1.1.4. | 后台公司支持的社区驱动.....                  | 12 |
| 1.2.   | groovy 能为你做什么.....                | 13 |
| 1.2.1. | java 内行人士的 groovy.....            | 13 |
| 1.2.2. | 对于脚本编写人员的 groovy .....            | 14 |
| 1.2.3. | 为敏捷开发人员 .....                     | 14 |
| 1.3.   | 运行 groovy.....                    | 15 |
| 1.3.1. | 使用 groovysh 编写 “HelloWorld” ..... | 16 |
| 1.3.2. | 使用 groovyConsole .....            | 19 |
| 1.3.3. | 使用 groovy.....                    | 20 |
| 1.4.   | 编译和运行 groovy .....                | 21 |
| 1.4.1. | 使用 groovyc 编译 groovy.....         | 22 |
| 1.4.2. | 使用 java 运行编译好的 groovy 脚本 .....    | 22 |
| 1.4.3. | 使用 Ant 编译和运行 .....                | 23 |
| 1.5.   | Groovy 的 IDE 和编辑器支持.....          | 24 |
| 1.5.1. | 集成 IntelliJ IDEA .....            | 25 |
| 1.5.2. | Eclipse 插件 .....                  | 25 |
| 1.5.3. | 其他文本编辑器的支持.....                   | 26 |
| 1.6.   | 摘要.....                           | 26 |
| 第一部分   | groovy 语言.....                    | 28 |
| 第二章    | 前奏: groovy 基础.....                | 30 |
| 2.1    | 一般代码外观 .....                      | 30 |
| 2.1.1  | groovy 代码的注释 .....                | 30 |
| 2.1.2  | 比较 groovy 和 java 的语法.....         | 31 |
| 2.1.3  | 简洁优雅的代码 .....                     | 32 |
| 2.2    | 探索语言的断言功能.....                    | 33 |
| 2.3    | Groovy 预览.....                    | 35 |
| 2.3.1  | 声明类 .....                         | 36 |
| 2.3.2  | 使用脚本.....                         | 36 |
| 2.3.3  | GroovyBeans .....                 | 37 |
| 2.3.4  | 处理文本.....                         | 38 |
| 2.3.5  | 数字也是对象.....                       | 40 |
| 2.3.6  | 使用 lists/maps/ranges .....        | 41 |
| 2.3.7  | 代码块: 闭包.....                      | 45 |
| 2.3.8  | groovy 结构控制.....                  | 47 |
| 2.4    | 在 java 环境中运行 groovy.....          | 49 |
| 2.4.1  | 我的类就是你的类.....                     | 49 |

|                               |     |
|-------------------------------|-----|
| 2.4.2 GDK:groovy 类库 .....     | 50  |
| 2.4.3 groovy 的生命周期 .....      | 52  |
| 2.5 概要 .....                  | 55  |
| 第三章 groovy 数据类型.....          | 56  |
| 3.1 无处不在的对象.....              | 56  |
| 3.1.1 java 类型——专有类型和引用类型..... | 56  |
| 3.1.2 Groovy 的答案：一切都是对象 ..... | 57  |
| 3.1.3 自动装箱和拆箱.....            | 59  |
| 3.1.4 没有中间层的拆箱.....           | 60  |
| 3.2 可选类型的概念.....              | 61  |
| 3.2.1 指定类型.....               | 61  |
| 3.2.2 静态类型 VS 动态类型.....       | 62  |
| 3.3 重载操作符.....                | 62  |
| 3.3.1 可重写的操作符一览.....          | 63  |
| 3.3.2 重写操作符实战.....            | 64  |
| 3.3.3 确保正确的工作 .....           | 66  |
| 3.4 使用字符串.....                | 68  |
| 3.4.1 字符串的样式.....             | 68  |
| 3.4.2 使用 Gstring 进行工作 .....   | 71  |
| 3.4.3 从 java 到 groovy .....   | 72  |
| 3.5 使用正则表达式.....              | 74  |
| 3.5.1 在字符串中使用模式 .....         | 76  |
| 3.5.2 应用模式.....               | 78  |
| 3.5.3 模式实战.....               | 80  |
| 3.5.4 模式和性能 .....             | 82  |
| 3.5.5 模式分类.....               | 83  |
| 3.6 使用数字 .....                | 84  |
| 3.6.1 造型和数字运算符.....           | 84  |
| 3.6.2 GDK 为数字提供的方法 .....      | 86  |
| 3.7 概要 .....                  | 87  |
| 第四章 集合类型 .....                | 89  |
| 4.1 使用 ranges .....           | 89  |
| 4.1.1 规定 ranges.....          | 90  |
| 4.1.2 range 是对象 .....         | 92  |
| 4.1.3 range 实战.....           | 93  |
| 4.2 使用 list .....             | 95  |
| 4.2.1 声明 list.....            | 95  |
| 4.2.2 使用 list 操作符 .....       | 96  |
| 4.2.3 使用列表方法.....             | 100 |
| 4.2.4 list 实战.....            | 104 |
| 4.3 使用 map.....               | 106 |
| 4.3.1 声明 map.....             | 106 |
| 4.3.2 使用 map 操作符 .....        | 108 |
| 4.3.3 map 实战.....             | 112 |
| 4.4 groovy 集合中需要注意的地方.....    | 114 |

|   |     |
|---|-----|
| 4.4.1 了解并发修改.....                       | 114 |
| 4.4.2 识别副本和修改在语义上的不同.....               | 115 |
| 4.5 摘要 .....                            | 115 |
| 第五章 闭包 .....                            | 117 |
| 5.1 出身名门的闭包.....                        | 117 |
| 5.2 闭包例子 .....                          | 118 |
| 5.2.1 使用迭代 (iterator) .....             | 119 |
| 5.2.2 处理资源.....                         | 120 |
| 5.3 声明闭包 .....                          | 123 |
| 5.3.1 简单的声明方式.....                      | 123 |
| 5.3.2 使用赋值的方式声明闭包 .....                 | 124 |
| 5.3.3 引用一个方法作为闭包.....                   | 125 |
| 5.3.4 比较.....                           | 127 |
| 5.4 应用闭包 .....                          | 128 |
| 5.4.1 调用闭包.....                         | 128 |
| 5.4.2 更多的闭包方法 .....                     | 131 |
| 5.5 理解范围 .....                          | 134 |
| 5.5.1 简单的变量范围 .....                     | 135 |
| 5.5.2 闭包范围.....                         | 135 |
| 5.5.3 工作中的范围使用：典型的累加测试.....             | 138 |
| 5.6 从闭包返回结果.....                        | 140 |
| 5.7 设计模式的支持.....                        | 141 |
| 5.7.1 Visitor 模式 .....                  | 142 |
| 5.7.2 Builder 模式 .....                  | 142 |
| 5.7.3 其他相关模式.....                       | 143 |
| 5.8 结束语.....                            | 143 |
| 第六章 groovy 的控制结构 .....                  | 145 |
| 6.1 groovy 真相.....                      | 145 |
| 6.1.1 评估 Boolean 测试 .....               | 145 |
| 6.1.2 将 Boolean 测试分配给变量 .....           | 147 |
| 6.2 条件执行结构 .....                        | 148 |
| 6.2.1 普通的 if 语句 .....                   | 148 |
| 6.2.2 三元条件操作符 .....                     | 149 |
| 6.2.3 switch 语句.....                    | 150 |
| 6.2.4 使用断言进行安全检查.....                   | 153 |
| 6.3 循环 .....                            | 157 |
| 6.3.1 while 循环 .....                    | 157 |
| 6.3.2 for 循环.....                       | 158 |
| 6.4 退出代码块和方法 .....                      | 160 |
| 6.4.1 正常终止： return/break/continue ..... | 160 |
| 6.4.2 异常： throw/try-catch-finally ..... | 161 |
| 6.5 总结 .....                            | 162 |
| 第七章 groovy 风格的动态面向对象 .....              | 163 |
| 7.1 定义类和脚本 .....                        | 163 |
| 7.1.1 定义属性和本地变量 .....                   | 164 |

|  |     |
|--|-----|
| 7.1.2 方法和参数 .....  | 167 |
| 7.1.3 安全的引用符号 (?) .....                                  | 171 |
| 7.1.4 构造器（构造方法） .....                                    | 173 |
| 7.2 组织类和脚本 .....   | 175 |
| 7.2.1 文件到类的关系 .....                                      | 175 |
| 7.2.2 在包中组织类 .....                                       | 177 |
| 7.2.3 类路径更长远的考虑 .....                                    | 181 |
| 7.3 高级 OO 特性 .....                                       | 183 |
| 7.3.1 使用继承 .....   | 183 |
| 7.3.2 使用接口 .....   | 183 |
| 7.3.3 Multimethods .....                                 | 184 |
| 7.4 使用 GroovyBean 工作 .....                               | 186 |
| 7.4.1 声明 Bean .....                                      | 187 |
| 7.4.2 使用 bean 工作 .....                                   | 189 |
| 7.4.3 为任何对象使用 bean 方法 .....                              | 192 |
| 7.4.4 属性、访问方法、隐射和扩展（Fields/accessors/maps/Expando） ..... | 193 |
| 7.5 使用强劲的特性 .....  | 195 |
| 7.5.1 使用 GPath 来查询对象 .....                               | 195 |
| 7.5.2 注入展开操作符 .....                                      | 200 |
| 7.5.3 使用 use 关键字进行混入 .....                               | 201 |
| 7.6 在 groovy 进行元程序编程 .....                               | 204 |
| 7.6.1 理解元类（MetaClass）的概念 .....                           | 205 |
| 7.6.2 方法调用和拦截 .....                                      | 207 |
| 7.6.3 方法拦截实战 .....                                       | 208 |
| 7.7 总结 .....   | 212 |

# 第一章 Groovy 之旅

欢迎进入 groovy 世界。

也许你已经在某些博客和邮件列表中听说过 groovy，已经在这里或者哪里看到了 groovy 的片段报道，大概你的同事已经告诉你用 java 写的一页代码在 groovy 中只需要短短的几行就可以完成相同的工作，也许你仅仅因为这本书的名字比较好记而翻阅这本书，为什么你应该学习 groovy？你期望的回报是什么？

Groovy 将让你迅速的获得成功，groovy 比用 java 写代码更加简单，更易进行自动化重复的任务，还可以作为日常工作用来编写特别脚本，groovy 的代码阅读起来更加自然易懂，当然更重要的是，groovy 用起来更加有趣。

学习 groovy 是一项明智的投资。Groovy 给给 java 平台带来高级语言强大的特性，如闭包、动态类型和元对象协议，你现有的 java 知识在 Groovy 中仍然有效，而不会过时。Groovy 建立在你现有的 java 经验和熟悉程度之上，这样你可以在适当的时候进行选择，或者混合使用 java 和 groovy。

如果对在 Ruby 上实现一个 web 应用不会感到惊奇的，Python 变戏法似的容器，Perl 人员通过少数的键盘输入来进行服务器管理，或者 Lisp 领袖利用极少的代码改变转向他们整个代码库，那么想想他们使用的语言的语言特性。Groovy 的目标是在 java 平台有提供类似功能的语言，尽量保留 java 对象模型和对 java 编程人员保持透明。

第一章介绍 groovy 的背景信息和开始使用 groovy 需要知道的一些知识。本章以 groovy 的故事开始：为什么创建 groovy，驱动的设计因素是什么，并且 groovy 在语言界的位置是怎样的。接下来的部分阐述 groovy 的优势和如何使用 groovy 来把你的工作变得更轻松。

我们坚信学习一门程序语言的唯一途径是：动手尝试它，我们介绍编译程序、解释程序和处理程序处理脚本时各种各样的变化；列出了在广泛使用的 IDE 上使用的各种插件和找到 groovy 最新信息的地方。

在本章的最后你将对 groovy 的使用有一个基本的认识。

衷心希望你在用 groovy 编程时和使用本书做指定参考时有一个愉快的时光。

## 1.1.groovy 背景

在 GroovyOne 2004——一次 groovy 开发人员在伦敦的集会上；James Strachan 发表演讲时说他已经有了怎么样发明 groovy 的主意。

在这不久前，James Strachan 和他的妻子有个计划，当她去购物的时候，James Strachan 上网时不由自主的转到 Python 网站并且决定开始学习这门语言，在学习过程中，他越来越喜欢 Python 了，作为一个成熟的 java 开发人员，他承认他的主攻语言缺乏 Python 中有趣的和有用的特性，如为通用的数据类型的自然支持，更加重要的是动态行为，他的想法是赋予 java 这些有趣的特性。

这些想法指导了 groovy 开发的主要方向：更加丰富的特性和比 java 更友好的语言，为已经十分成熟的平台带来动态语言的特性。

图 1.1 展示了 James Strachan 怎样独特的定位 groovy 在 java 编程界的位置，我们不想通过准确的描述来触犯任何人，我们相信任何其他的语言也许在未来会更好，但是我们肯定 groovy 的位置

一些语言也许比 groovy 有更多的特性，一些语言也许主张与 java 进行更好的集成。当考虑两方面集成在一起的时候目前没有哪个语言比 groovy 做的更好：没有哪个语言提供了比 groovy 的更好的 java 友好性和完整的现代语言特性。

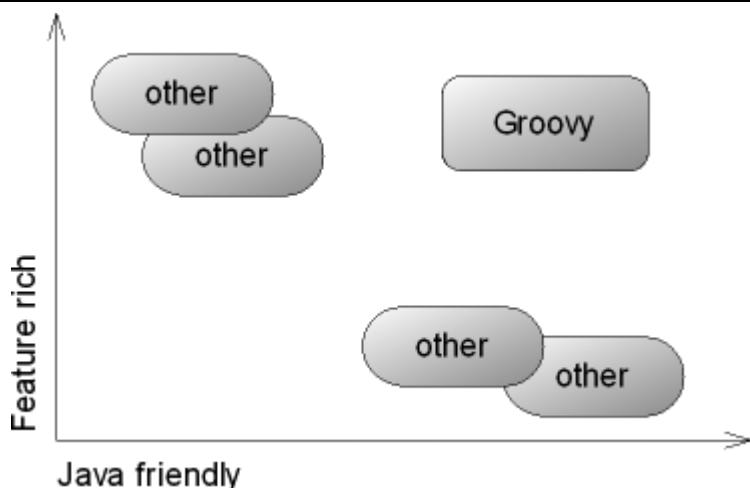


图 1.1.

看了 groovy 的一些目标，我们来看看具体有那些。

### 1.1.1. groovy 是什么？

Groovy 的网站 (<http://groovy.codehaus.org>) 给出了 groovy 的最好定义：groovy 是在 java 平台上的、具有象 Python, Ruby 和 Smalltalk 语言特性的灵活动态语言，groovy 保证了这些特性象 java 语法一样被 java 开发者使用。

Groovy 经常被认为是脚本语言——它也像脚本一样工作的很好。但是，把 Groovy 理解为脚本语言是一种误解，groovy 代码被编译成 java 字节码，然后能集成到 java 应用程序中或者 web 应用程序，整个应用程序都可以是 groovy 编写的——groovy 是非常灵活的。

groovy 与 java 平台非常融合，groovy 的许多代码是使用 java 实现的，其余部分是用 groovy 实现的，当你使用 groovy 编程的时候，许多情况下你正在写特殊的 java 程序，在 java 平台上的所有强大功能——包括大量的 java 类库也可以直接在 groovy 中使用。

这样说来 groovy 只是为 java 增加了语法糖吗？不完全对，虽然你在 groovy 中做的每一件事情通过 java 也可以做，但是用 java 代码实现 groovy 的魔术工作是会让人发狂的，groovy 在后台做了许多工作来完成敏捷性和动态性，就像你读这本书一样，有时设法想想 groovy 使用 java 模仿的效果要做的事情，首先，groovy 的许多特性特别神奇——透明的对方封装逻辑，建立层次的代码，在一般应用程序语言进行中高效的数据库查询，在每个对象被创建之后控制每个对象的运行时行为——所有这些任务都不能在 java 中执行，你也许认为 groovy 象一个“彩色的”语言，那么相比较而言 java 就是黑白的——在后面是黑白的点上创建了彩色特性。

我们来看看 groovy 的魅力，一步一步开始 groovy 和 java 的工作。

### 1.1.2. 与 java 共事：无缝集成

与 java 的友好性有俩方面：与 java 运行时环境无缝集成和与 java 相似的语法。

#### 无缝集成

图 1.2 显示 groovy 的集成情况：groovy 运行在 java 虚拟机，java 的类库也可以继续使用，Groovy 仅仅是创建 java 类的一种新的途径——通过在运行时创建，groovy 是使用了额外 jar 文件为依赖的 java。

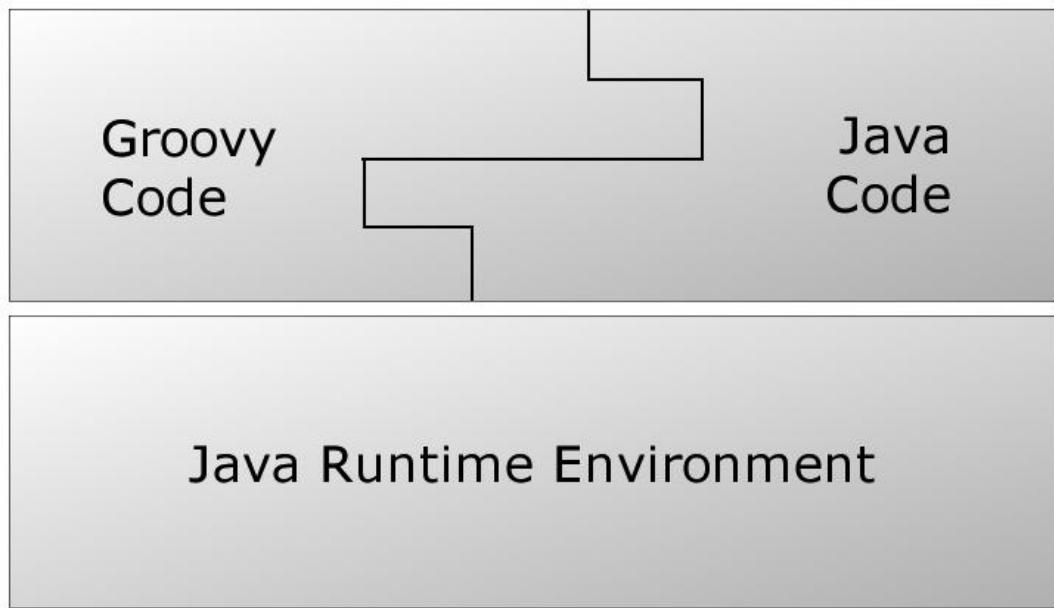


图 1.2

因此，从 groovy 调用 java 是没有任何问题的。当使用 groovy 开发的时候，你不需要得到任何通知就可以这样做，每一个 groovy 的类型（groovy type）都是 `java.lang.Object` 的子类。每一个 groovy 对象都是一个类的实例。一个 groovy date 对象是 `java.util.Date` 的实例等等。

反向集成也是非常容易，假设一个 groovy 类 `MyGroovyClass` 被编译到一个以 “.class” 结尾的文件中，并且把它放在 java 的 classpath 中，那么你能在 java 类中以一个类的形式使用这个 groovy 类

```
new MyGroovyClass();      //create from java
```

换句话说，实例化一个 groovy 类与实例化一个 java 类是一样的，在这之后，一个 groovy 类就是一个 java 类，你能在这个实例上调用类的方法，传参数给方法等等，JVM 根本不知道运行的代码是通过 groovy 编写的。

### 相似的语法

Groovy 的第二个友好的方面是语法友好，让我们比较一下 groovy 与 java 获得当前日期方面的不同机制，我们将 groovy 和 ruby 的排在一起：

```
import java.util.*; // Java
Date today = new Date(); // Java
today = new Date() // a Groovy Script
```

```

require 'date' # Ruby
today = Date.new # Ruby

```

groovy 的解决方法最少，并且比 java 代码更加紧凑，groovy 不需要导入“java.util”包或者指定 Date 类型；此外，当 groovy 能理解没有分号的代码的时候，groovy 不要求书写分号，尽管代码十分紧凑，但 groovy 的代码完全能够让 java 开发者理解。

Ruby 的代码也列出来以说明 groovy 想避免的问题：一个不同的包概念（require），不同的注解方式，不同的对象创建语法，虽然 ruby 能够识别这些代码（也许比 java 更加相容），但是与 groovy 比较起来，他的语法和结构与 java 不一致

现在，对 groovy 与 java 集成和语法的友好性有了一个了解，但是特性增强怎么样呢？

### 1.1.3. 强劲代码：一个特性丰富的语言

列出 groovy 的特性列表就有点像列出一个舞蹈演员舞蹈的每一步，虽然每一个特性在他自身看来都是重要的，这些特性在 groovy 中是非常的融合，groovy 在 java 上有三个主要类型的特性：语言特性，groovy 类库和附加到已经存在的 java 标准类上的功能（GDK），图 1.3 显示了这些特性的关系，阴影圆标示了特性之间相互之间引用，比如，许多类库严重依赖语言特性，符合 groovy 语言的代码很少独立的使用一个特性，相反，通常 groovy 同时使用这些特性。

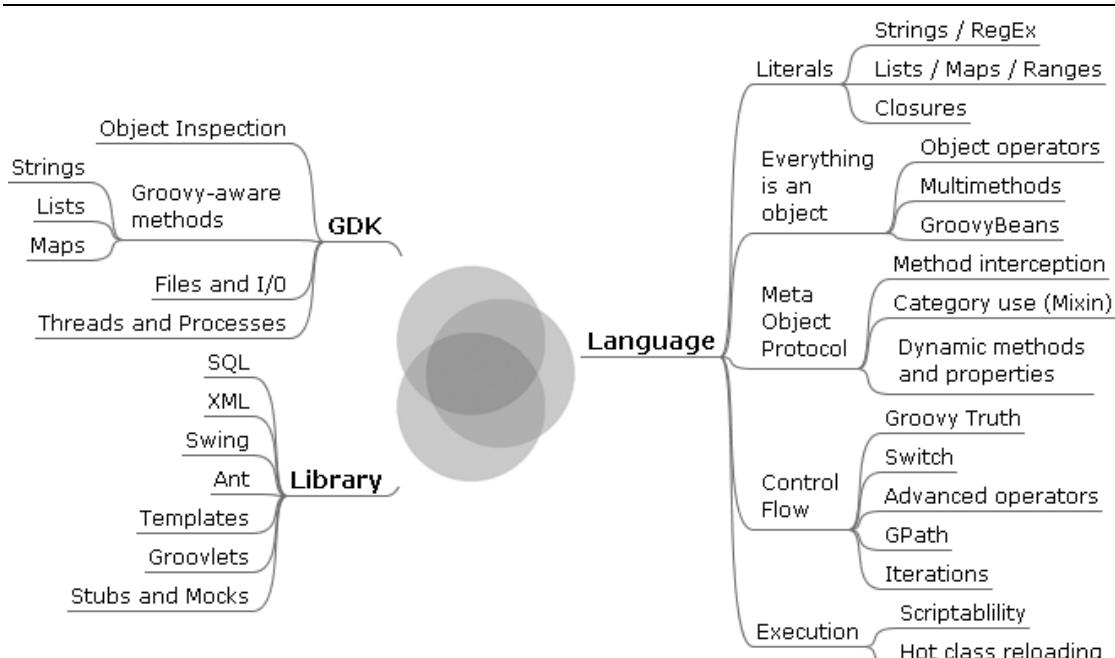


图 1.3

遗憾的是，许多特性用几句话不能说明白，比如闭包（Closures）——groovy 中一个非常重要的语言概念，但是它的字面意思不能告诉你任何事情，我们现在不打算介绍闭包的所有细节，但是有一些闭包的例子将使你感兴趣。

### 列出文件内容：闭包和附加的 IO 特性

闭包是一个代码块，它能作为一个正常类处理：作为引用传递，存储，在任何时候运行等等，java 的匿名内部类经常也是这样使用的，特别是适配器类，但是 java 匿名内部类的语法是丑陋的，并且被限制在他们能访问和改变的数据之内。在 groovy 的文件处理变得容易了很多，因为 groovy 增加了很多方法到 java.io 包的类中，一个典型的例子是 File.eachLine 方法，你经常需要读一个文件，一次一行，并且在每一行进行相同的处理，在文件结束的时候关闭文件吗？这是一个非常通用的任务，它不应该困难——所以在 groovy 中，它不困难。

我们来把这两个特性放在一起并且创建一个完整的程序，该程序列出一个文件中每一行的数据，在每一行前面输出行号：

```
def number=0
new File ('test.groovy').eachLine { line ->
    number++
    println "$number: $line"
}
```

在读取文件的一行之后，花括号中的闭包代码都被执行，这是通过 File 的新的 eachLine 方法来确保闭包代码被执行的。

### 打印列表：容器迭代和简单的属性访问方式

java.util.List 和 java.util.Map 大概是 java 中使用得最广泛的两个接口，但是几乎没有直接支持的语言，groovy 增加了通过语法来申明 list 和 map 的能力，groovy 也增加了许多别的方法到 collection 类上。

同样的，javaBean 的属性访问机制也无处不在，但是 java 没有很好的利用他们，groovy 简化属性访问，同时充分考虑到了代码的可读性。

下面的例子用这两个特性打印类列表中的每一个类的包名称，注意 package 需要用引号包括起来，因为它在 java 中是一个关键字，但是它仍旧能被用作属性名称，尽管 java 允许像第一行那样声明一个数组，但是我们在这里作为一个真正的 list 使用——不需要額

外的工作就可以增加和删除 list 中的对象：

```
def classes = [String, List, File]
for (clazz in classes)
{
    println clazz.'package'.name
}
```

在 groovy 中，你甚至能避免通过 for 循环来访问 list 中每个对象的属性——这样得到的结果是 list 中每个对象的属性值的一个列表。使用这个特性，上面代码的等价解决代码为：

```
println( [String, List, File].'package'.name )
```

在控制台输出的结果为：

```
["java.lang", "java.util", "java.io"]
```

非常酷！！

## Groovy 的 xml 处理方式： GPath 和动态属性

是否你用 java 进行过大量的 XML 处理工作，如果选择 W3C DOM 使工作会变得更容易一些，但是 java 本身不在语言层面对你提供任何帮助——它不能适应你的需要。Groovy 允许类在编译时没有的属性在运行时起作用。GPath 就是建立在这个特性之上的，并且允许像 Xpath 那样来浏览 XML 文档。

假定你有一个名称为 “customers.xml” 的文件，内容像这样：

```
<?xml version="1.0" ?>
<customers>
    <corporate>
        <customer name="Bill Gates" company="Microsoft" />
        <customer name="Steve Jobs" company="Apple" />
        <customer name="Jonathan Schwartz" company="Sun" />
    </corporate>
    <consumer>
        <customer name="John Doe" />
        <customer name="Jane Doe" />
    </consumer>
```

```
</customers>
```

你能使用下面的代码输出所有的“corporate”客户的名称和公司（从一开始使用 groovy 的 Builder 生成 xml 文件也是比 java 容易许多）。

```
def customers = new XmlSlurper().parse(new File('customers.xml'))
for (customer in customers.corporate.customer)
{
    println "${customer.@name} works for ${customer.@company}"
}
```

这里仅仅证实了 groovy 的少数特性，你已经在前面的例子中看到了 groovy 别的特性——使用 GString 对 String 进行功能增强，简化了的 for 循环处理，可选的类型声明，可选的语句结束符（在 java 中为分号），这些仅仅是个开始，这些特性彼此之间工作的十分融洽，以至于你很难注意到你正在使用他们。

尽管比 java 友好和功能增强是 groovy 的主要目标，但也考虑到许多有价值的方面，迄今为止，我们已经注意到 groovy 的许多技术手段，但是一个语言需要更成功，吸引大量的人，在计算机语言世界，更多更好的特性不保证会让你有大量的用户，必须对开发人员和管理人员两个方面都具有吸引力才行。

#### 1.1.4. 后台公司支持的社区驱动

对于一些人来说，他们投资学习的语言是一个标准化的语言是让人十分欣慰的，这是 groovy 的承诺之一，自从 groovy 通过了 JSR-241，groovy 成为了 java 平台的第二个标准语言（第一个是 java 语言）。

大量的用户是第二个准则，由于有大量的用户，就会有大量机会获得好支持的和可持续开发，Groovy的用户有相当的规模。一个好的指示是邮件列表的活跃程度和大量的相关项目（请参考<http://groovy.codehaus.org/Related+Projects>）。

一个更吸引人的是超过战略上的考虑，不管怎样，凭感觉将导致你享受编程。

Groovy开发者知道这种感觉，并且仔细的考虑什么时候决定使用上面的语言特性，毕竟，语言的名称也是一个原因。

有人说groovy是“java风格+Ruby感受”，我不认为这是一个好的描述，groovy的工作就像你和程序语言之间的合作伙伴，groovy的工作是准确无误的让计算机明白你的意图。

当然，你仍旧需要支付账单，有时让你美好的心情感觉像“feel the groove”，在下一节，

我们将来看看groovy带给你那些的优点。

## 1.2.groovy 能为你做什么

根据你的工作经验,你也许对不同的 groovy 的特性感兴趣,无论谁都不可能需要 groovy 的所有特性,正像没有人使用 java 提供的整个框架类库一样。

这一节介绍 java 内行人士、脚本程序员和敏捷开发人员感兴趣的 groovy 特性和适用范围,我们认识到开发人员在工作中很少从事一个角色的工作,有时不得不承担不同角色的工作,无论如何, groovy 致力将各种角色联合在一起在许多情况下是有帮助的。

### 1.2.1. java 内行人士的 groovy

如果你认为自己是一个 java 专家,那么你也许有了多年的 java 行业经验。你了解 java 类库中所有重要的部分和许多常用的扩展 java 包。

坦率的说,你不能改变像每天在当前目录下递归搜索所有文件的任务,如果你想要达到这样的效果,在 java 中将需要编写很多代码。

在这本书中你将了解到,通过 groovy 你能快速的打开控制台,通过输入如下代码来输出所有的文件名(递归处理):

```
groovy -e "new File('.').eachFileRecurse { println it }"
```

即使 java 有一个名称为 eachFileRecurse 的方法和一个 FileListener 的接口,你仍旧需要显示的创建一个类,声明一个 main 方法,保存源代码到一个文件中,并且编译它,然后你才能运行它,基于比较,我们来看看 java 做相同工作的代码,我们假设存在单独的 eachFileRecurse 方法:

```
public class ListFiles { // JAVA !!
    public static void main(String[] args) {
        new java.io.File(".").eachFileRecurse( //假设java存在该方法
            new FileListener() {
                public void onFile (File file) {
                    System.out.println(file.toString());
                }
            }
        );
    }
}
```

```
}
```

注意 `java` 框架代码是如何的作用（打印每一个文件）是不重要的，这里是为了达到完成程序的目的。

还有命令行可用和漂亮的代码，`groovy` 允许你赋予 `java` 应用程序动态行为，例如通过业务规则进行动态配置，甚至实现领域模型语言（domain specific languages）。

你可用使用静态的或者动态的类型来编译代码，作为应该开发人员，为了无论在什么地方和什么时间，增加代码适当的灵活性，`groovy` 让你有了选择。

这应该给你足够的信心让你增加 `groovy` 到你的项目中，以便从 `groovy` 的特性中获取优势。

### 1.2.2. 对于脚本编写人员的 `groovy`

作为一个脚本编写人员，你也许使用 `Perl`、`Ruby`、`Python`，或者其他动态（非脚本）语言如 `Smalltalk`、`Lisp`、`Dylan` 做工作。

但是 `java` 平台有无可争辩的市场占有率，使用 `java` 工作是相当普遍，客户通常运行一个 `java` 标准平台（如 J2EE），`java` 将被开发和部署在产品环境，你没有机会应用你的脚本解决方案，你不得不编写无休止的 `java` 代码，一整天的想“如果这里我有自己的语言，我能用一行替换整个方法！” ，我们不得不承认会有这种挫败感。

`Groovy` 能给你轻松，并且让你在需要高级语言的地方找回用高级语言特性编程的乐趣：在你的日常工作中，允许你在任何对象上进行方法调用，为立即执行或者延后执行的程序提供代码块，增加存在的库代码给你的专业语义，并且使用许多别的强大的特性，`groovy` 让你清晰的、并且通过少量代码就能表达自己的意思。

仅仅需要把“`groovy-all-*.jar`”文件放在你的项目的类路径中。

今天，软件开发不再是一个人的事情，你的同事（和你的老板）需要知道你使用 `groovy` 做什么和 `groovy` 是什么，这本书的目标也是成为你和别人一起学习的工具。（当然，如果你不想和别人一起分享这本书，告诉他们让他们自己买一本，我们不会介意）

### 1.2.3. 为敏捷开发人员

如果你已经进入这个范围，你也许已经有一个不堪重负的书架，到处贴着列有任务的索引卡，一个自动测试的套件在适当的时机将会运行，下一个释放版本已经关闭。这里是

`groovy` 的时代，甚至完全的让你和同事愉悦的进行结对编程。

还有一种情况，实用性、极限编程或者敏捷编程不时让你不得不返回。放轻松，然后评估你的工具是否还足够锋利，不管多个紧迫的项目计划，你需要定期磨练你的工具，在软件领域，这意味现有的知识和资源需要有一个正确的使用方法，不管是工具、技术还是语言。

`Groovy` 是一个适合你的项目的、非常宝贵的自动化任务工具。包括简单的自动构建，持续集成，报告和自动更新文档，部署和安装。`groovy` 自动支持强大的诸如 Ant 和 Maven 解决方案，尽可能提供简单和简明语言意思来控制他们，`groovy` 甚至有助于测试，包括单元测试和功能测试，帮助我们方便自在的进行测试驱动开发。

在使用 `groovy` 之前，我使用过其他的脚本语言（更好的 `ruby`）来设计我的思路，做一个试验来评估任务的可行性——并且运行一个函数原型，我从来没有想过我写的这些代码也可以在 `java` 中工作，更坏的情况是，最后我从构架重做工作，通过 `groovy`，我能直接在我的目标工作平台做所有的探索工作。

`Groovy` 和 `java` 优化后的代码相互合作：`java` 代码需要在运行时进行代码优化工作，使用 `groovy` 编写的代码在灵活性和可读性方便进行优化。

除了这些实际的好处之外，学习 `groovy` 的理由还有很多，它开阔你的思路，使你有了新的解决方案，在开发的时候帮助你理解新概念，无论你使用哪个语言。

你是哪种类型的程序员不重要，我们希望你立刻开始编写一些 `groovy` 代码，如果你不能这样做，那么回到第二章看看真实的 `groovy` 代码。

## 1.3. 运行 `groovy`

首先，我们先要介绍运行和编译（编译时可选的）`groovy` 代码的工具，如果你想试试这些工具，你需要先安装 `groovy`，当然，附录 A 提供了一个安装指南。

在表 1.1 列出了执行 `groovy` 代码和脚本的三个命令，在接下来的章节将通过例子和屏幕截图来演示如果使用这三种不同方式来运行 `groovy`，`groovy` 也能像原始的 `java` 程序一样“运行”，同样你在 1.4.2 节将看到这一点，并且在 1.4.3 节也有与 ant 进行集成的说明。

我们将在 11 章见到一些集成 `groovy` 到 `java` 程序的办法。

表 1.1 运行 `groovy` 的命令

| 命令            | 描述  |
|---------------|---|
| groovysh      | 开始 groovysh 命令行 shell, 它用来交互式执行 groovy 代码, 在 shell 中一行一行的输入语言或者整个代码, 通过 go 命令立即执行输入的代码。 |
| groovyConsole | 运行一个图形界面用来交互执行 groovy 代码; 此外, groovyConsole 也可以加载和运行 groovy 脚本文件。                       |
| groovy        | 开始解释执行 groovy 脚本, 单行 groovy 脚本可以作为命令行参数来运行。   |

### 1.3.1. 使用 groovysh 编写 “HelloWorld”

首先看 groovysh, 因为它是我们手边十分方便进行实验 groovy 的工具, 在 groovysh shell 中进行编辑和运行 groovy 是非常容易的, 并且这样做也减轻了工作量, groovysh 不需要创建和编辑 groovy 的脚本文件。

启动 shell, 在命令行运行 groovysh(UNIX)或者运行 groovysh.bat(windows), 然后你应该在命令行看到像下面这样的提示信息:

```
Lets get Groovy!
=====
Version: 1.0-RC-01-SNAPSHOT JVM: 1.4.2_05-b04
Type 'exit' to terminate the shell
Type 'help' for command help
Type 'go' to execute the statements

groovy>
```

译者注: 在翻译这本书的时候, groovy 的最新版本是 1.6.0, 提示稍微信息有些不一样。

传统意义上的“HelloWorld”程序在 groovy 中通过一行代码就可以写出来, 并且在 groovysh 中通过 go 命令进行运行:

```
groovy> "Hello, World!"
groovy> go
====> Hello, World!
```

go 命令是 groovy 可以识别的少数命令中的一个，剩余的命令通过 help 命令能够显示出来：

```
groovy> help
Available commands (must be entered without extraneous
characters):
exit/quit      - terminates processing
help          - displays this help text
discard       - discards the current statement
display        - displays the current statement
explain        - explains the parsing of the current statement
                  (currently disabled)
execute/go     - temporary command to cause statement execution
binding        - shows the binding used by this interactive shell
discardclasses - discards all former unbound class definitions
inspect        - opens ObjectBrowser on expression returned from
                  previous "go"
```

go 与 execute 命令是等效的，discard 命令告诉 groovy 丢弃最后输入的一行脚本，在你录入一个大的脚本的时候这个命令是有用的，因为该命令方便的清除一小块代码，这比必须重新从头录入整个脚本要好，我们再来看看别的命令。

### Display 命令

display 命令显示最后录入的非命令的语句：

```
groovy> display
1> "Hello World!"
```

### Binding 命令

binding 命令显示在 groovysh 会话中可使用的变量，在我们的简单例子中没有使用任何变量，但是，为了演示这个命令，我们修改我们的“Hello World! ”，使用变量 greeting 来作为信息的一部分，然后我们再输出这个信息：

```
groovy> greeting = "Hello"
groovy> "${greeting}, World!"
groovy> go
==> Hello, World!
groovy> binding
```

```
Available variables in the current binding
greeting = Hello
```

当在groovysh会话中编写长的脚本时，或者忘记了正在使用的变量的值的时候  
binding命令是有用的。

为了清除binding中的变量，需要退出shell，然后重新开始一个新的shell。

### Inspect 命令

inspect命令打开“Groovy Object Browser” groovy对象浏览器查看最后运行的表达式，这个浏览器是一个swing用户界面，这让你直接可以看到一个对象的原始javaAPI和通过groovy的GDK附加在这个对象上的任何可用特性：

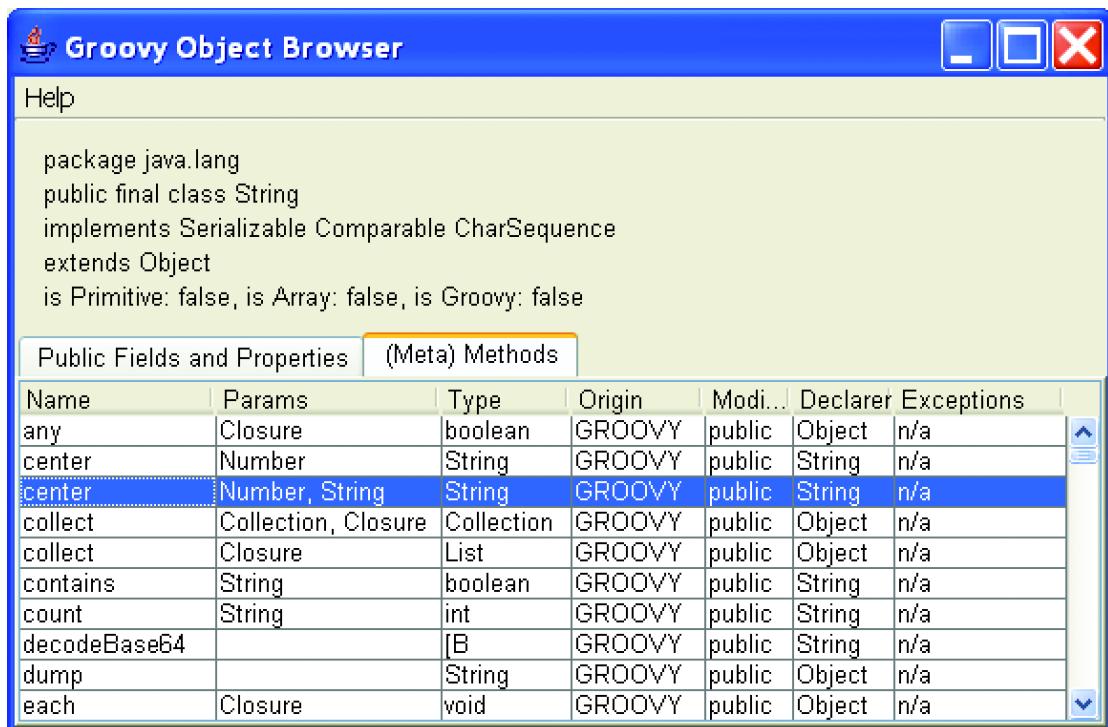


图1.4 groovy对象浏览器打开了String对象，显示了它可用的方法列表和注册的元数据方法

图1.4显示了对象浏览器透视String的一个实例，在顶部包括了String类的信息和两个表，这两个表显示了String可用方法和字段的列表。

看第二行和第三行，在一个字符串上有一个名称为center的方法是可用的，这个方法接受一个Number类型的参数（第二行）和一个可选的String类型的参数（第三行），方法返回一个String类型对象，groovy在String类上定义了这个新的、公共的方法。

如果你像我们一样，迫不及待的想试试在groovysh中的新知识并且输入：

```
groovy> 'test'.center 20, '-'
groovy> go
====> -----test-----
```

几乎和IDE工具支持的一样好！！

为了方便浏览，你可以点击列头来对数据进行排序，再点击一下按刚才的顺序进行倒序排列，你也可以通过按顺序点击列头进行多列排序，并且通过拖动列头来重新排安排列的顺序。

Groovy对象浏览器未来的版本也许提供更多的高级特性。

### 1.3.2. 使用 groovyConsole

groovyConsole是一个swing界面，是一个小型的groovy交互式解释程序。

groovyConsole不支持groovysh所支持的命令行选项；但它有一个“File”菜单来加载，创建和保存groovy脚本文件。有趣的是，groovyConsole是使用groovy编写的，它的实现是通过一个示范性的Builder来进行的，这将在第七章进行讨论。

groovyConsole不接受参数，它启动一个面板窗口（图1.5），在上方的面板接受键盘输入，为了运行一个脚本，通过“Ctrl+R”，“Ctrl+Enter”或者“Action”菜单的“Run”命令来运行这个脚本，如果脚本代码中的一部分是被选中的时候，那么仅仅是被选中的文本被执行，这个特性在进行简单调试或者连续选择一行或者多行进行单步执行的时候是有用的。

groovyConsole “File”菜单有新建（new）、打开（open）、保存（save）和退出（Exit）命令，**新建（New）**用来打开一个新的groovyConsole窗口。**打开（Open）**用来浏览和打开在文件系统中已经存在的groovy脚本到编辑面板进行编辑和运行，**保存（Save）**用来保存在编辑面板中的文本到一个文件中，，**退出（Exit）**用来结束groovyConsole的运行。

在图1.4显示的Groovy对象浏览器在groovyConsole中同样有效，并且也是显示最后计算的表达式结果，为了打开浏览器，按下Ctrl+I（为了查看透视对象）或者在Actions菜单中选择透视（Inspect）命令。

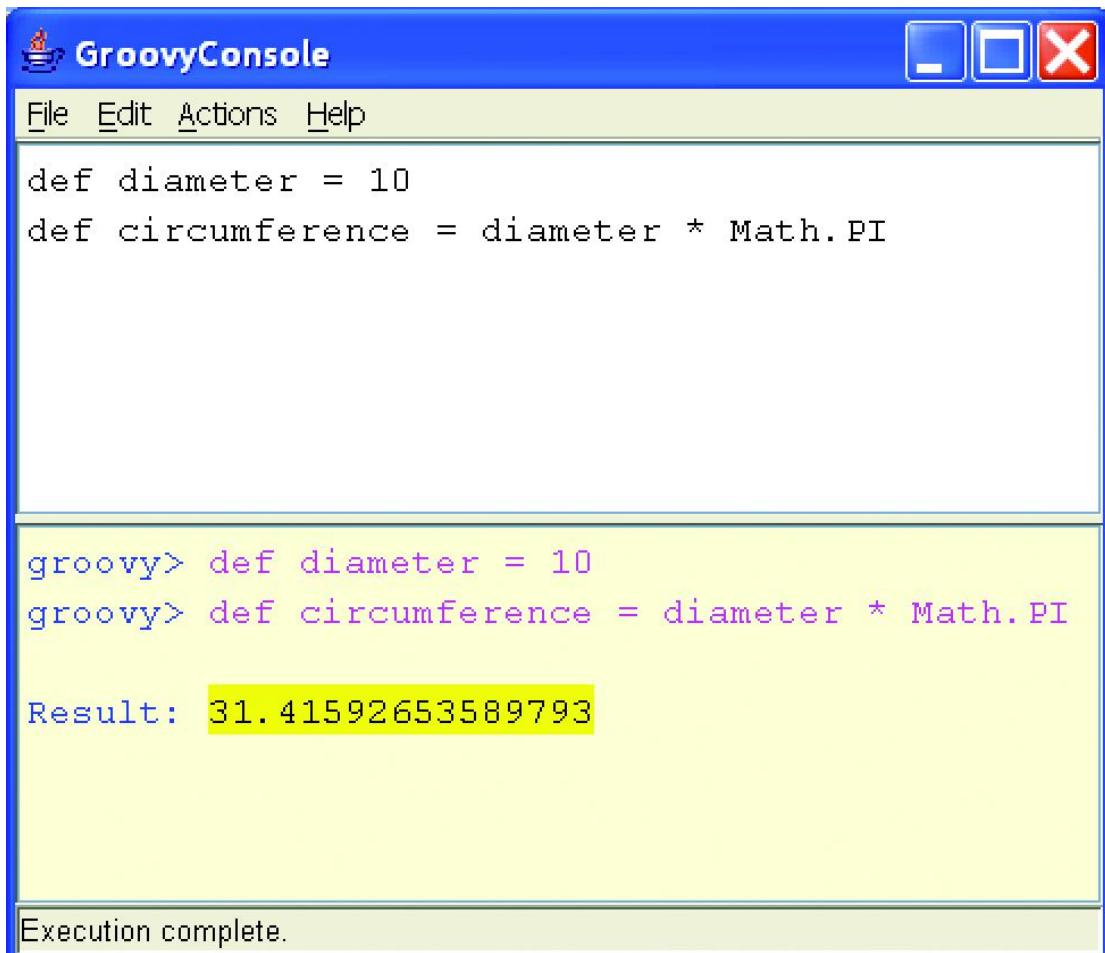


图1.5 groovyConsole和在编辑面板中的简单脚本，这个脚本用来根据一个圆的直径来计算圆的周长，结果在输出面板中显示出来。

groovyConsole就介绍到这里了，选择在groovysh中，或者groovyConsole中进行工作根据个人喜好进行选择，脚本程序员更喜欢在shell中处理他们的工作。

除非做出了明确的说明，否则你能把这本书中的任何代码直接放到groovysh或者groovyConsole中并且运行它，更多的时候你会这样做，早期时用来感受groovy语言的魅力。

### 1.3.3. 使用 groovy

Groovy命令用来执行groovy程序和脚本，例如，列表1.1 显示的使用groovy打印头10个斐波那契数的数列，斐波那契数列中，头2个数是1和1，后面的每一个数是这个数前面两个数之和。

如果你喜欢这个例子，拷贝代码到一个文件中，并且把这个文件保存为Fibonacci.groovy，文

件扩展名不一定要使用groovy，但是使用“.groovy”作为groovy脚本的后缀是一种约定，使用“.groovy”作为扩展名的一个好处是在命令行指定脚本名称的时候你可以省略扩展名——比如“groovy MyScript.groovy”，你可以这样输入“groovy MyScript”就行了。

表1.1 Fibonacci.groovy

```
current = 1
next = 1
10.times {           //循环10次
    print current + ' '
    newCurrent = next
    next = next + current
    current = newCurrent
}
println ''
```

作为一个groovy程序，运行这个文件只需要把文件名称传递给groovy命令，你应该得到下面的输出结果：

```
> groovy Fibonacci
1 1 2 3 5 8 13 21 34 55
```

Groovy命令有许多附加的选项对命令行脚本是有用的，例如，输入“groovy -e “println 1+1””表达式能被直接执行，这将在控制台输出2，12.3节将介绍完整的选项列表，包括了许多的例子。

在这节，我们已经看了groovy简化操作的许多脚本，但这不是全部，下一节将详细讲述groovy的代码编译、运行的整个过程。

## 1.4. 编译和运行 groovy

迄今为止，我们已经直接使用了groovy，代码是直接执行而没有产生任何可执行文件，在这节，你将接触到第二种使用groovy的方法：编译成java字节码并且作为正常java应用程序运行在java虚拟机上，这就是预编译模式，这两种方法最终都是在java虚拟机上运行，两种方法都编译groovy代码为java字节码，主要的区别是编译产生结果类的时间和结果类是保存在内存还是磁盘上。

### 1.4.1. 使用 groovyc 编译 groovy

编译groovy是很简单的，因为groovy包括了一个名叫groovyc的编译器，groovyc编译器为每个groovy源文件产生至少一个类文件。作为例子，我们通过运行groovyc来编译上节的Fibonacci.groovy文件为一个一般的java字节码：

```
> groovyc -d classes Fibonacci.groovy
```

在我们的这个例子中，groovyc产生一个父类为groovy.lang.Script的java类，这个类包括一个main方法，因此java能直接运行它，编译出来的class的名称同脚本的名称是一致的，在这里为Fibonacci。

根据编写的groovy代码，编译器也许会产生更多的类，不管怎样，我们不需要关心它，因为它是java平台的话题，本质上，groovyc的编译方式与javac的编译方式是相同的。

注意：Fibonacci脚本包括了代码“10.times{}”结构导致groovyc产生一个类型为Closure的类，代码的实现是花括号内的代码，这个类嵌套在Fibonacci.class中，在第五章将详细的介绍Closure(闭包)，如果现在难以理解，可以暂时跳过这部分。

类文件的实现映射显示在表1.2中，包括了每一个类的详细解释。

表1.2 groovyc为Fibonacci.groovy文件产生的类

| 类文件                            | 父类                  | 作用                         |
|--------------------------------|---------------------|----------------------------|
| Fibonacci.class                | groovy.lang.Script  | 为了Java命令中运行，包括了一个main方法    |
| Fibonacci\$_run_closure1.class | groovy.lang.Closure | 处理在10次循环中每次循环所做的工作，目前可以忽略它 |

现在我们已经有了一个编译好的程序，我们看看该如何运行它。

### 1.4.2. 使用 java 运行编译好的 groovy 脚本

运行一个编译后的groovy程序与运行一个编译后的java程序是一样的，有一个额外的要求是把groovy-all\*.jar文件放在你的JVM类路径中，这将确保groovy依赖的第三方类库能够在运行时被解析，同时，也确保你编译的程序所在的文件夹也在JVM类路径中，然

后你使用java命令运行程序的方法和你运行其他java程序的方法是相同的，

```
> java -cp %GROOVY_HOME%/embeddable/groovy-all-1.0.jar;classes  
Fibonacci  
1 1 2 3 5 8 13 21 34 55
```

注意：当通过java运行时，不应该指定主类（包含main方法的类）文件的扩展名（为.class）。

### 1.4.3. 使用 Ant 编译和运行

使用groovy运行groovyc是一个ant任务，为了使用Ant，你需要先安装Ant，我们建议使用1.6.2或者更高的ant版本。

表1.2显示了一个ant脚本，这个脚本用来编译groovy文件为java字节码，并且作为java字节码运行Fibonacci.groovy。

**Listing 1.2 build.xml for compiling and running a Groovy program as Java bytecode**

```
<project name="fibonacci-build" default="run">  
    <property environment="env"/>  
    <path id="groovy.classpath">  
        <fileset dir="${env.GROOVY_HOME}/embeddable/" />  
    </path>  
  
    <taskdef name="groovyc"  
            classname="org.codehaus.groovy.ant.Groovyc"  
            classpathref="groovy.classpath"/>  
  
    <target name="compile"  
            description="compile groovy to bytecode">  
        <mkdir dir="classes"/>  
        <groovyc  
            destdir="classes"  
            srcdir=".."  
            includes="Fibonacci.groovy"  
            classpathref="groovy.classpath"/>  
        </groovyc>  
    </target>  
  
    <target name="run" depends="compile"  
            description="run the compiled class">  
        <java classname="Fibonacci">  
            <classpath refid="groovy.classpath"/>  
            <classpath location="classes"/>  
        </java>  
    </target>  
</project>
```

保存这个文件在当前目录中，文件名为build.xml，这个文件夹下面也要包括

Fibonacci.groovy源文件，然后在命令行运行ant命令。

建立过程将首选从运行“run”目标开始，“run”目标依赖compile目标，因此compile目标讲先被调用，compile目标只运行groovyc一个任务，为了保证ant识别这个任务。这里引入了“taskdef”，taskdef用来在定义的类路径“groovy.classpath”中找到groovyc任务。

当在compile目标中所有的编译工作都执行成功之后，run目标然后调用一个java任务来运行编译好的程序。

输出的结果应该象这样：

```
> ant
Buildfile: build.xml
compile:
    [mkdir] Created dir: ...\\classes
    [groovyc] Compiling 1 source file to ...\\classes
run:
    [java] 1 1 2 3 5 8 13 21 34 55
    BUILD SUCCESSFUL
Total time: 2 seconds
```

再次执行ant将没有编译信息输出，因为groovyc任务能保证只在需要的时候（源文件被修改，译者注）进行编译，为了进行完整的编译，你必须在编译之前删除相应的目标文件夹。

groovyc任务有许多可选项，这些选项大多数与java任务类似，其中srcdir和destdir选项是必须的。

为了进行自动构建，在使用ant（或者Maven）集成groovy到java项目中的时候，使用groovyc来进行编译是很方便的。关于groovy与ant和Maven进行集成的更多信息将在14章提到。

## 1.5. Groovy 的 IDE 和编辑器支持

如果你有经常用groovy进行编码的计划，你肯定希望groovy对你使用的IDE或者编辑器有一定支持。在目前这个阶段有些编辑器仅仅支持groovy语法的高亮显示，但是在groovy编码提供便利性方面这也是相当有用的，groovy一些常用的IDE和文本编辑器的列表在下节列出。

这节在印刷好之后可能很快过时，时刻保持更新你喜欢的IDE，因为近来groovy对主要的IDE工具的支持性正在提高。Sun公司最近也宣布通过项目Coyote (<https://coyote.dev.java.net>) 在Netbeans中支持groovy，这是特别有趣的，因为这是第一个直接有IDE厂商进行集成的工具。

### 1.5.1. 集成 IntelliJ IDEA

在groovy社区，有个名叫GroovyJ的开源插件在正在开发中，通过这个插件和IDEA内置的特性，groovy程序员能够获取到下面的好处：

- 基于用户喜好的语法高亮显示：GroovyJ目前使用java 5的语法高亮，这包括了groovy的大部分的语法，1.0版将支持完全的groovy语法，并且也可以通过Color&Fonts面板进行高亮颜色的自定义，就像java语法一样。
- 代码自动完成：迄今为止，代码的自动完成仅仅局限在单词的自动完成，IDEA的单词自动完成是基于一个字典来处理的。
- 和IDEA的编译、运行、建立和生成配置和输出视图紧密的集成。
- 在java中使用的高级编辑器动作也能使用。
- 在项目高效的查询，或者依赖类库中的java类。
- 高效的文件导航，包括了groovy文件
- 一个groovy文件图标

GroovyJ有美好的未来，对于groovy来说有非常多的IDEA的PSI（程序构建接口）相关特性，由于这个目的，专门为groovy生成语法文件和产生一个专门的解析器，因为IDEA在基于PSI的高级特性（如重构支持，代码检查，导航等等），我们有希望在groovy中也存在这些特性。

GroovyJ是一个有趣的项目，它由Franck Rasolo领导，这个插件是groovy众多高级插件中的一个，请关注<http://groovy.codehaus.org/GroovyJ+status>了解更多的信息。

### 1.5.2. Eclipse 插件

Eclipse的插件要求Eclipse 3.11或者更高的版本，这个插件也可以运行在Eclipse派生工具如IBM Rational的Rational应用程序开发者和Rational软件架构上，到写这

本书的时候，Groovy的eclipse插件支持以下特性：

- Groovy文件的语法高亮显示
- 在包浏览器和资源视图中提供了一个groovy文件的图标
- 直接在IDE中运行groovy脚本
- Groovy文件的自动创建
- 集成调试功能

可以在<http://groovy.codehaus.org/Eclipse+Plugin>下载可用的插件。

### 1.5.3. 其他文本编辑器的支持

虽然文本编辑器没有开发环境的高级特性，许多有针对性的编辑器也支持程序语言一般特性。

UltraEdit容易进行自定义，提供语法高亮显示和在编辑器内部启动或者编译groovy脚本，输出信息将被输出到一个集成的输出窗口中，一个小型工具条能够方便的定位到在文件中声明方法的地方，支持groovy的代码缩进和括号匹配。参考

<http://groovy.codehaus.org/UltraEdit+Plugin>了解更多的信息。

JEdit的groovy插件支持在编辑器内部执行groovy脚本和代码快照功能，有一个可用的独立的语法高亮显示配置，参考<http://groovy.codehaus.org/JEdit+Plugin>了解更多。

TextPad、Emacs、Vim和一些别的文本编辑器的语法高亮配置在groovy的网站<http://groovy.codehaus.org/Other+Plugins>也可以找到。

在javaIDE的高级特性如调试，单元测试，和动态代码完成等方面将继续得以发展。

## 1.6. 摘要

我们确信你已经真的在使用groovy，作为一个建立在sun公司支持的牢固的java语言基础上的现代语言，groovy和java可以很好的融合在一起。

知道了为什么开发和设计groovy语言，你应该能够知道接下来的章节要介绍groovy的哪些特性，记住java集成原则和功能增强。使一般性任务更简单并且代码更具有描述性。

由于你已经安装了groovy，你可以直接运行一个脚本，也可以编译成类进行运行，如果你有足够的精力，你甚至可以在你常用的IDE中安装groovy插件，当这些工作完成之后，

做好了解更多语言自身知识的准备，在下一章，我们介绍groovy的功能，让你了解groovy的魅力，第一部分介绍了groovy剩下的知识。

# 第一部分 **groovy** 语言

学习新的程序语言好比学习说外语，你必须学习新的词汇，语法和语言习惯。学习初的努力是必不可少的，无论如何，学习新语言唯一方法是自我练习，接受新概念和风格，并且你探索新想法，这些都是 groovy 为我们做到的，并且我们希望 groovy 也帮助你做到。

这本书的第一部分介绍了语言基础：groovy 的语法和用法习惯，我们通过例子介绍，不使用学术式的方式进行介绍。

也许你在开始的时候跳过这部分，以后再回来浏览这部分，如果决定跳过这部分，那么务必要看看第二章及其中的例子，这一章的内容交错，有深度，能找到你感兴趣的任何主题。

通过例子解释程序语言的困难之一是你必须从某个地方开始，不管你的起点在哪里，最终需要使用例子说明一些概念和特性。

我们使用语言的数据类型和介绍表达式、操作符和关键字来讲解语言，本章以大多数语言相似的地方开始，我们希望你在进入新的领域的时候总是充满信心。

第三章介绍 groovy 的类型和 groovy 在语言层面支持的数字类型。

第四章继续介绍 groovy 丰富的内建类型，groovy 的容器类型：ranges，lists 和 maps。

第五章在前面章节的基础上深入介绍闭包（closure）概念。

第六章涉及到逻辑、分支、循环和程序执行流程的捷径。

最后，在第七章介绍 groovy 建立在 java 基础上的对象特性和他们的动态执行流程。

在第一部分的结尾，你将了解到 groovy 语言的整个体系，这是学习第二部分——有关 groovy 类库的基础：groovy 增加到 java 平台的类和方法。第三部分，标题为“groovy 的每一天”将应用第一部分和第二部分的知识到日常编程工作中。

# 第二章 前奏：groovy 基础

在这一章，我们将介绍 groovy 语言的基础知识，我们讲过开始使用 groovy 需要知道的两件事：代码外观和断言。在本章，我们通过例子来开始学习 groovy；每一个例子仅仅只有一部分代码会进行详细的描述——刚好能保证你入门，如果有些例子介绍的不够清楚，在读完本章之后再回过头看看这些例子。

## 2.1 一般代码外观

计算机语言从外观来说有明显的分类，例如，一个 C 程序员看到 java 代码也许不理解许多关键字，但是能够识别象花括号、操作符、圆括号、注释、语句结束符这些符号，groovy 能够让你通过一种途径从 java 平滑的过渡过来，联想、相似的语言习惯是成长的知识，我们也将看到一些基本的东西——怎么样注释代码，java 和 groovy 的注释的不同之处和相似之处。groovy 代码更简洁，因为它让你忽略一些语法元素。

首选，groovy 不区别语法缩进，缩进用于代码块是好的实践，groovy 大多数时候也不知道多余的空格，例外的是语句结束和单行的注释，我们现在来看看这些知识。

### 2.1.1 groovy 代码的注释

Groovy 的单行注释和多行注释与 java 非常相似，groovy 也支持在脚本文件的第一行的一个附加注释：

```
#!/usr/bin/groovy
// some line comment

/*
some multiline
comment
*/
```

编写 groovy 注释的一些指导方针：

- “#!”注释只允许在脚本文件的第一行出现，通过这种方式 Unix shell 能定位 groovy 的启动脚本并且运行这些脚本。

- “//” 单行注释，注释文字必须在这一行结束。
- 多行注释使用 “/\*.....\*/” 包围起来。
- Javadoc 风格的注释 “/\*\*.....\*/” 和多行注释是相似的，但是这种注释支持 groovydoc 用来生成文档，groovydoc 与 javadoc 使用的语法是等价的。  
注释仅仅是 groovy 语法比 java 语法友好的一部分，而不是全部。

## 2.1.2 比较 groovy 和 java 的语法

一些 groovy 代码（不是所有）太像 java 了，这经常给人的感觉就是：groovy 的语法是 java 语法的超集。尽管相似，但 groovy 的语法不是 java 语法的超集。例如，groovy 现在不支持 java 经典的 `for(init;test;inc)` 循环。你在 2.1 章将看到，两种语言会有稍微的不同（例如，`==`操作符）。

除了这些细微的差别，许多主要 java 语法也是 groovy 语法的一部分，包括：

- 相同的包处理机制（包括包的声明和 `import` 语句）
- 类和方法的定义（嵌套类除外）
- 控制结构语句【`for (init;test;inc)` 循环除外】
- 操作符、表达式和赋值
- 异常处理
- 变量声明（也有一些不同）
- 对象实例化，引用和取消引用对象，方法调用

Groovy 语法增加的部分：

- 通过新的表达式和操作符访问 java 对象
- 多种途径声明对象
- 提供新的控制结构来进行流程控制
- 引入新的数据类型和相应的操作符与表达式
- 把所有事物都看成对象来处理

大体上来说，groovy 可以看成是 java 和这些附加功能的和，这些附加的语法元素确保代码更紧凑更易读，一个有趣的方面是 groovy 的加法运算能应用到前面没有提到的对象上。

### 2.1.3 简洁优雅的代码

Groovy 允许忽略一些在 java 中必须的语法元素，忽略这些元素的结果是代码更简短、更少冗余和更清晰的表述。例如，将一个 String 对象编码为一个 URL 对象，比较它们的 java 和 groovy 的代码：

```
Java:  
java.net.URLEncoder.encode("a b");
```

```
Groovy:  
URLEncoder.encode 'a b'
```

groovy 的代码不但更简短，而且以尽可能简单的方法表述了我们的目标。忽略了包的前缀，圆括号和分号，代码量减少到最小。

Groovy 中方法调用的圆括号是可选的，这是建立在消除模棱两可的情况和 groovy 语言规范中概述优先处理规则上的。虽然这些规则清晰，但是有时不够直观，省略括号可能引起误解，即时编译器喜欢这样的代码，我们更愿意在所有的情况下使用圆括号，但是大多数时候这样做是没有意义的，编译器不会为了可读性来鉴定你的代码——必须自己做这些工作。

在第七章，我们将谈论可选的返回语句。

Groovy 自动导入以下包和类：

```
groovy.lang.*  
groovy.util.*  
java.lang.*  
java.util.*  
java.net.*  
java.io.*  
java.math.BigInteger  
java.math.BigDecimal
```

可以直接使用在这些包中的类，而不需要指定包的名称，在这本书中我们始终都会用到这个特性，我们使用全限定类名称来消除模棱两可的情况，或者是需要指定源的地方。注意 java 只自动导入 java.lang.\*。

这一章已经列出了足够的信息来了解每一个单独的特性，我们仍旧会用更详细的方式来讨论这些特性，但是你应该能够接受这本书一致的代码风格，接下来我们将了解到进行测试每个功能点的最重要的工具：assertions（断言）。

## 2.2 探索语言的断言功能

如果使用 `java1.4` 或者更高的 `java` 版本进行工作，那么你也许已经熟悉 `assertions`（断言），断言用来测试你的程序的结果是否是正确的结果，通常，断言贯穿在代码中用来保证代码没有任何逻辑上的矛盾，用来执行例如在方法的开始和结束的时候检查常量或者保证方法的参数是有效的。在这本书，我们使用断言来演示 `groovy` 的特性，正像在测试驱动开发一样，测试被认为是保证单元代码正确性的最后手段，在这本书中，断言用来验证执行一段特定的 `groovy` 代码而获得的结果，我们使用断言来表明不但代码可以运行，而且也要验证代码运行的结果是否正确，这节将为你读这本书的例子做准备，这节解释了在 `groovy` 中断言是如何工作的并且应该怎样使用断言。

开始学习一门语言时就介绍断言好像比较奇怪，但这是必须的，除非你理解断言，否则你不会懂得本书其余的例子，`groovy` 使用关键字“`assert`”来提供断言的功能，表 2.1 显示他们看起来的样式。

表 2.1 使用断言

```
assert(true)
assert 1 == 1
def x = 1
assert x == 1
def y = 1 ; assert y == 1
```

我们来一行一行的看看这些代码。

```
assert(true)
```

这里介绍的“`assert`”关键字显示你需要提供一个断定结果为 `true` 表达式。

```
assert 1 == 1
```

这显示 `assert` 关键字能够接受一个完整的表达式，而不仅仅是直接值或者简单的变量。毫不意外的是，1 等于 1，这象 Ruby，不象 `java`，“`==`”操作符表示两个数的值等价（“`==`”号调用了对象的 `equals` 进行比较），而不是判断引用是否相等，这里省略了圆括号，因为在顶级语句中他们是可选的。

```
def x = 1
assert x == 1
```

这里定义了一个变量 `x`, 将值 1 赋值给 `x`, 并且使用 `x` 作为断言的表达式。注意我们没有透露关于 `x` 的类型的任何信息, “`def`” 关键字的意思是“动态的类型”。

```
def y = 1 ; assert y ==1
```

这是典型断言当前行的风格, 在一行有两个语句, 通过分号进行分隔。分号是 `groovy` 语句的结束符, 当语句在当前行结束的时候分号可以忽略。

断言的多个作用:

- 断言用来显示程序当前的状态, 就像我们在这本书中的例子一样, 先前的断言显示变量 `y` 有一个值, 这个值为 1。
- 断言经常用来替换行注释, 因为他们显示期望的结果并且在同一时间进行验证, 注释在没有人注意的时候也许会过期——断言总是检查正确性, 在真实的代码里它们象极小的单元测试。

**真实的生活经历:** 写这本书是真正的体验了断言, 这本书被构建为允许运行包含断言的例子代码, 工作是这样的: 这本书的初稿是 `MS-word` 格式, 不包括代码, 但是包括一个符号指向相应的脚本文件, 通过少量的脚本, 所有的符号被扫描并且加载相应的文件, 使用这些脚本来运算和替换文档中的符号, 例如, 在列表 2.1 的断言被评估, 并且相应文件被找到, 然后进行正确的替换处理, 如果断言失败, 那么处理过程给出一个错误信息后停止。

由于这本书是一个成品书, 这就意味着产品的处理不会停止并且所有的断言都是成功的。这让你对书中的例子有足够的信心, 这不但证明了断言, 也使用脚本 (15 章) 来控制 `ms-word` 和 `AntBuilder` (8 章) 来进行建立工作——正如我们前面说的那样, `groovy` 的特性是非常棒的。

大多数例子使用的断言——表达式的一部分用来描述的功能特性, 另外一部分将简单化至能够理解的地步, 如果理解一个例子有任何困难, 那么停下来, 想一想我们正在讨论的语言功能和你希望得到的结果, 然后看看我们对结果的说明是什么。断言在运行时被执行, 图 2.1 将一个复杂的断言语句分解为不同的部分来解释。

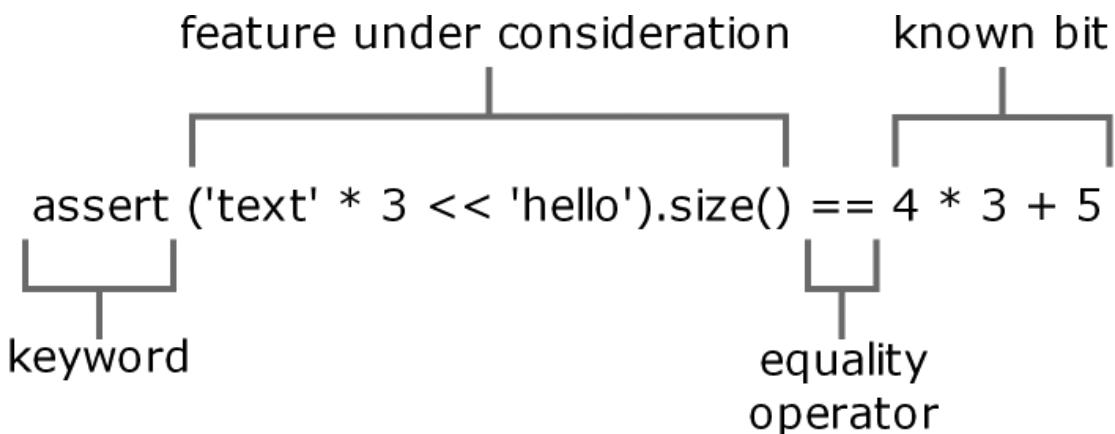


图 2.1 一个复杂的断言，分解成不同的部分

这是一个极端的例子——我们经常在独立的语句中执行断言，并且保持断言本身的简单。原则是一样的，这里只是用来我们证明代码的功能，代码本身没有价值，不需要知道后面知识就容易理解。

假如不相信断言或者不明白断言表达式，通常可以通过在控制台输出相应信息来代替，例如，这样的一个断言：

```
assert x == 'hey, this is really the content of x'
```

可以通过下面的语句代替：

```
println x
```

这会在控制台打印出 `x` 的值，在本书中，为了检查代码，我们经常通过控制台输出来代替断言，这不是一个常用来介绍书中代码的方法，但是我们能进一步感受到代码和结果，对我们进行最原始的测试驱动有帮助。

断言有一些有趣的特性能影响到你的程序风格。6.2.4 节将深入讨论断言，现在已经解释了用在 groovy 中的工具，我们可以开始看看一些有用的功能了。

## 2.3 Groovy 预览

像许多开发语言一样，groovy也有一个语言规范来分类语句、表达式等等，从一个大的规范中来学习一个语言将是非常枯燥的，并且不会让你在短时间内写出groovy代码，相反，我们通过简单的例子来介绍groovy典型的结构：classes, scripts, beans, strings, regular

expressions, numbers, lists, maps, ranges, closures, loops, and conditionals。

这一节只是一个预览，没有回答你的任何问题，但将让你有自己开始写groovy程序的经验。我们鼓励自己动手进行实验——如果你对你的代码是否发生什么感到苦恼，进行测试，通过动手获取最好的经验，我们保证在接下来的章节给出一个更详细的解释。

### 2.3.1 声明类

类是面向对象编程的基础。因为类用来定义了一个对象的结构。

列表2.2包括一个名称为Book的groovy类，它有一个属性title，一个构造方法用来给属性title赋值，一个title的访问方法(getter方法)。注意，在这里代码非常像java，除了没有方法的访问范围修饰符：groovy中缺省的方法访问修饰符是public。

表 2.2 一个简单的 Book 类

```
class Book {  
    private String title  
    Book (String theTitle) {  
        title = theTitle  
    }  
    String getTitle(){  
        return title  
    }  
}
```

把这段代码保存在名称为Book.groovy的文件中，因为将在下一节中使用到它。

这段代码没有什么特别的地方，类的声明与大多数面向对象语言是非常相似的。类声明的详细说明和具体细节将在第七章进行解释。

### 2.3.2 使用脚本

脚本是一个扩展名为.groovy的文本文件，这个文件能够在命令行执行：

> **groovy myfile.groovy**

注意：这与java很不一样，在groovy中，我们正在执行的是源代码！事实上，一个普通的java类被JVM产生并且被运行。但是在用户看来，它就像我们正在执行groovy源代码。

脚本包括了groovy语句，这些语句没有在一个声明的类中。脚本甚至可以在类定义的外面包含方法的定义，在第七章将进一步了解到脚本的知识，在那之前不要把这当成一回事。

列表2.3显示怎样容易的使用在脚本中使用Book类，我们创建了一个新的实例并且在这个对象上使用java的“.”语法调用getter方法。然后我们定义一个方法来读取title。

表 2.2 在脚本中使用 Book 类

```
Book gina = new Book('Groovy in Action')
assert gina.getTitle() == 'Groovy in Action'
assert getTitleBackwards(gina) == 'noitcA ni yvoorG'

String getTitleBackwards(book) {
    title = book.getTitle()
    return title.reverse()
}
```

注意，我们能够在声明方法getTitleBackwards之前调用该方法，注意groovy这后面的基本原理与其他脚本语言如果Ruby是不相同的，一个groovy脚本被完整构建——也就是说，在执行之前脚本被转换、编译和产生类，7.2节有关于这方面更详细的描述。

另外一个重点是我们在没有明确编译Book类的时候使用Book类！唯一的前提是Book类的文件Book.groovy必须在类路径中，groovy运行时系统将自动找到这个文件，显式的将它编译成类，并且产生一个新的Book对象实例。Groovy联合了脚本和面向对象的优点。

Book类和脚本的使用是非常简单的，很难相信它能变得更简单，但是这是做到的，在下一节你将能看到这一点。

### 2.3.3 GroovyBeans

JavaBeans是一个显露出属性的普通java类，那么什么是属性呢？这还有点不好解释，因为属性不能单独存在，它是一个命名概念，如果一个类暴露了名称结构为getName（）和setName（name）的方法，那么概念描述的name就是类的一个属性。get-和set-方法叫做访问

者方法。(有些人区别了访问者方法和修改者方法，我们在这里不进行区别)

一个GroovyBean是在groovy中定义的JavaBean，在Groovy中，使用bean的工作比java中容易了很多，groovy使用beans工作便利表现在三种途径：

- 自动生成访问者方法
- JavaBeans（包括GroovyBeans）的简化访问方式
- 事情处理器的简化注册

列表2.4显示Book类在一行代码中定义了一个名称为title的属性，这个属性的访问者方法

getTitle()和setTitle(title)将自动生成。

我们也说明怎样通过标准的访问者方法访问bean，又是简单的方法。

表 2.4 定义 Book 类为 GroovyBean

```
class Book {  
    String title //声明一个属性  
}  
def groovyBook = new Book()  
  
//通过显示的方法调用来使用属性  
groovyBook.setTitle('Groovy conquers the world')  
assert groovyBook.getTitle() == 'Groovy conquers the world'  
  
//通过groovy的快捷方式来使用属性  
groovyBook.title = 'Groovy in Action'  
assert groovyBook.title == 'Groovy in Action'
```

注意列表2.4的代码是一个完全有效的脚本并且能够运行，尽管含了类声明和附加的代码，在第七章将了解到这钟结构的更多知识。

同时注意groovyBook.title不是类属性的直接访问，它是访问该属性访问者方法的快捷方式。

关于方法和beans的更多信息将在第七章介绍。

### 2.3.4 处理文本

就像在java中一样，字符数据大多数时候作为java.lang.String类型来处理，groovy提供了

一些途径使字符串工作更容易，赋予了字符串更多的选项和一些有用的操作符。

## GStrings

在groovy中，字符串能出现在单引号或者双引号中，在双引号的字符串中允许使用占位符，占位符在需要的时候将自动解析，这是一个**GString**类型，下列代码说明一个简单变量替换，虽然这不是Gstrings能做的全部工作：

```
def nick = 'Gina'  
def book = 'Groovy in Action'  
assert "$nick is $book" == 'Gina is Groovy in Action'
```

第三章将描述更多关于字符串的信息——包括**GString**的更多选项，怎样转换特殊的字符，怎样跨越多行声明字符串，在字符串可用的方法和操作符。正如你期望的那样，**GStrings**是相当漂亮整洁的。

## 正则表达式

如果你熟悉正则表达式的相关概念，你将非常高兴的看到groovy在语言级别对正则表达式的支持，如果你初次接触正则表达式，你可以暂时放心的跳过这一节，在第三章你能找到这个主题的完整的介绍。

Groovy提供容易声明正则表达式的手段，作为一个操作符来应用表达式模式，图2.2使用/.../语法声明了一个模式并且使用“=~”来根据模式匹配给定字符串，第一行保证字符串包含一个数字的系列；第二行使用“x”来替换每一个数字。

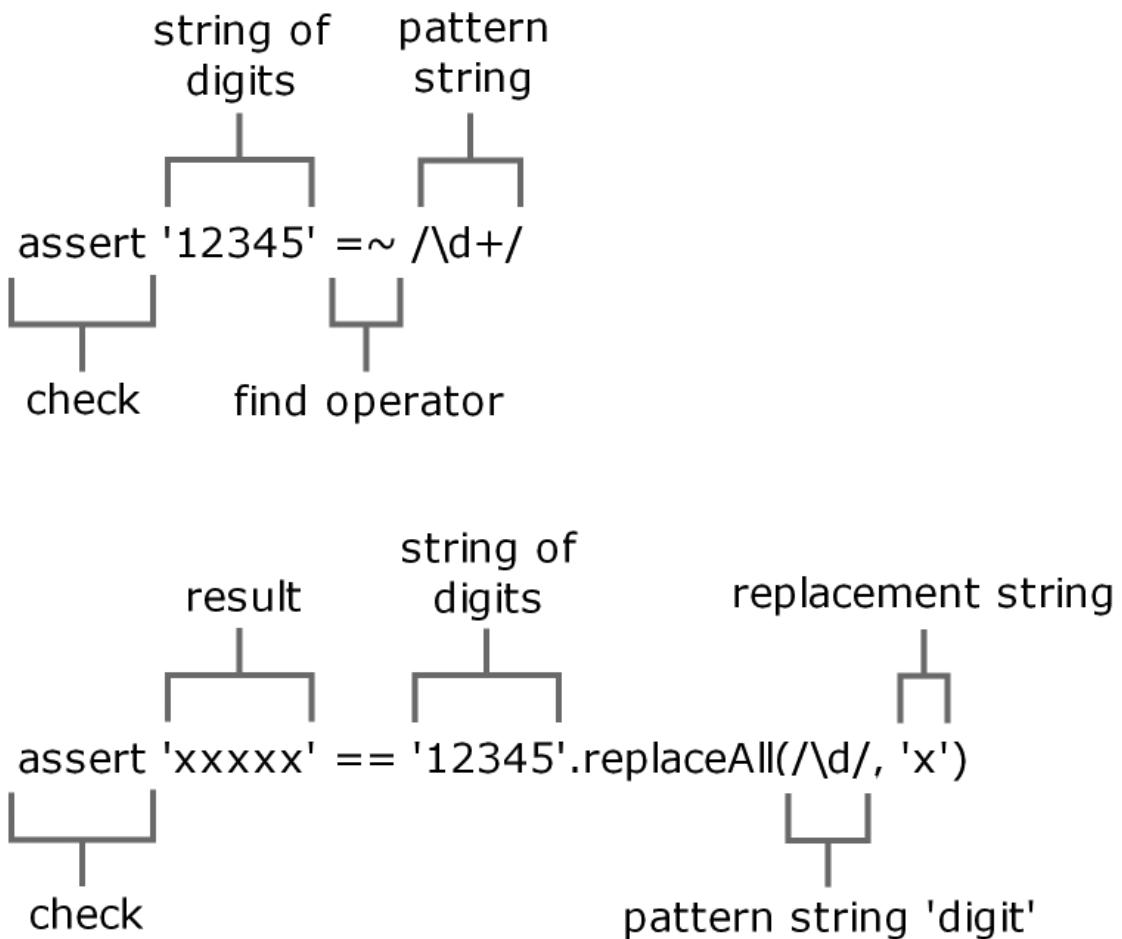


图2.2 在groovy中通过操作符和.../字符串来支持正则表达式

注意方法`replaceAll`定义在`java.lang.String`中，并且接受两个字符串参数，它清晰的表明‘12345’是一个字符串，`\d/`是一个正则表达式。

第三章说明怎样声明和使用正则表达式及如何应用。

### 2.3.5 数字也是对象

所有的程序都离不开数字，大多数情况下，为了计算和索引。Groovy的数字有一般的表现形式。Groovy的数字不像java，他们都是类对象，而不是专有类型。

在java中，不允许在专有类型上调用方法，如果x是一个专有的类型int，不能写这样的代码`x.toString()`。另一方面，如果y是一个对象，你不能像“`2*y`”这样用。

在groovy中，这两者都是允许的，能够在数字上使用数字操作符，并且也可以在数字实例上调用方法。

```
def x = 1
```

```
def y = 2
assert x + y == 3
assert x.plus(y) == 3
assert x instanceof Integer
```

变量x和y都是`java.lang.Integer`的实例，因此，可以调用`plus`方法，但是也可以方便的使用“+”操作符。

这让人十分意外，并且是对`java`平台面向对象的一个重大提升，而`java`有一小部分不是面向对象的，因此`java`不是纯粹意义上的面向对象的语言，`groovy`中保证了一切事物都是对象，在第三章将学习到`groovy`处理数字的更多的知识。

### 2.3.6 使用 lists/maps/ranges

许多开发语言，包括`java`，都有容器类型——数组，在语言层有这个功能，但是仅仅能应用到给的的类型。在现实中，别的容器也广泛使用，并且使用容器比使用数组更困难，也没有说明原因。`Groovy`简化了容器的处理，增加了操作符的支持，并且在`java`类库的基础上补充了更多的方法。

#### Lists

`Java`通过使用方括号和下标来索引数组，这种方式我们叫做下标操作符（`subscript operator`），`groovy`采用了相同的语法来支持`list`——`java.util.List`的实例，`java.util.List`允许向列表增加对象或者从列表中删除对象，允许在运行时改变列表的大小，保存在列表中的对象不受类型的限制；另外，`groovy`可以通过超出列表范围的数来索引列表，再一次表明可以改变列表的大小，此外，列表也可以直接在代码中指定。

下面的例子声明一个罗马数字列表并且用开头7个数来初始化这个列表，如图2.3所示

| <b>Index</b> | <b>Roman numeral</b>         |
|--------------|------------------------------|
| <b>0</b>     |                              |
| <b>1</b>     | I                            |
| <b>2</b>     | II                           |
| <b>3</b>     | III                          |
| <b>4</b>     | IV                           |
| <b>5</b>     | V                            |
| <b>6</b>     | VI                           |
| <b>7</b>     | VII                          |
| <hr/>        |                              |
| <b>8</b>     | <b>VIII</b> <b>New entry</b> |

图2.3 列表中每一个位置对应一个描述该位置的罗马数字

列表中每一个位置对应一个罗马数字，做起来就像使用数组一样，但是在groovy中，还有更多以前只有用数组的操作也被应用到list列表：

```
//罗马数字列表
def roman = ['', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII']

//访问列表
assert roman[4] == 'IV'
```

```
//扩展列表  
roman[8] = 'VIII'  
assert roman.size() == 9
```

注意，在我们为列表位置索引为8的位置赋值时，该位置其实并不存在，代码中操作了超出列表范围的位置，在4.2节讨论列表的更多信息。

### 简单的映射（maps）

一个map是用来给一个键（key）分配值（value）的强类型，map通过key存储和找回值（values），但是list通过位置找回值。

不像java，groovy在语言级别支持map，允许使用特定的操作符来操作map，这是一个清晰的和简单的语法，maps的操作语法像键—值对数组，通过冒号分隔键（key）和值（values）。他们一起被获取。

下面的例子存储http返回的代码描述在一个map中，如图2.4描绘的那样：

| Key<br>(Return code) | Value<br>(message)    |
|----------------------|-----------------------|
| 100                  | CONTINUE              |
| 200                  | OK                    |
| 400                  | BAD REQUEST           |
| 500                  | INTERNAL SERVER ERROR |
|                      | New entry             |

图2.4 一个用来表示http返回代码和他们的描述信息的map

你看到的map的声明和初始化，获取值和增加新的实体，这些的都在源代码中通过一个方法完成，甚至是检查map的size：

```
def http = [  
    100 : 'CONTINUE',  
    200 : 'OK',
```

```
400 : 'BAD REQUEST' ]  
  
assert http[200] == 'OK'  
http[500] = 'INTERNAL SERVER ERROR'  
assert http.size() == 4
```

注意声明、访问和修改列表是如何统一在一起的，map和list之间的差别非常小。因此，这两个都很容易记住，这个例子很好的说明了groovy语言设计者怎样设计一般的请求操作，并且为程序员提供简单的、一致的语法，让程序员的工作更轻松。4.3节介绍map和groovy相关容器功能的更多信息。

### 范围（Ranges）

尽管ranges没有出现在java标准类库中，大多数程序员对range的概念有一个直观的感觉——一个有效的开始点和一个结束点，那么range是如何从开始点移动到结束点，groovy又一次在语法层面对这个概念提供了支持，range就像for语句一样容易理解。

下面的代码说明了range的语法格式、怎样知道range的内容数量、说明是否包含一个特定的值、找到range的开始点和结束点、并且对内容进行反向排序：

```
def x = 1..10  
assert x.contains(5)  
assert x.contains(15) == false  
assert x.size() == 10  
assert x.from == 1  
assert x.to == 10  
assert x.reverse() == 10..1
```

这个例子有一定的局限性，因为我们仅仅显示了range自身可以做的工作，range通常和别的groovy功能联合一起使用，纵览这本书，你会看到许多range的应用。

接下来我们将介绍闭包，这是在java中不存在的概念，但在groovy中却广泛使用。

### 2.3.7 代码块：闭包

闭包不是一个新的概念，但是它通常用在函数式语言中，它允许执行一个任意指定好的代码块。

在面向对象语言，方法对象模式（Method-Object pattern）通常用来模拟一个行为：为一个目的定义一个独立的方法接口，这个接口的实例能够传递一组参数给方法，然后调用这个接口方法。

一个好的例子是`java.io.File.list(FilenameFilter)`方法，`FilenameFilter`接口只有一个方法，这个方法只有一个目的，就是从`list`方法返回的列表可以通过`FilenameFilter`的方法进行过滤。

遗憾的是，这导致了一些不必要的增殖类型，并且由于广泛使用独立逻辑指针导致代码难以理解。`java`使用匿名内部类来达到这种效果，但是语法是难以理解的，并且他们的意义被限制在调用方法的内部，和访问调用方法的内部变量。`groovy`允许将闭包简明的内联在代码中，清晰而强大，实际上提升方法对象模式到语言顶级类的位置。

由于闭包对于`java`程序员来说是全新的概念，所以`java`程序员也许需要一定的时间来适应闭包。一个好消息是使用闭包的起步非常容易以至于你很难注意到闭包是一个新的概念，当发现闭包的优势之后，你会感觉非常酷。

通俗的说，一个闭包是一个用花括号围起来的语句块，像别的任何代码块，为了传递参数给闭包，闭包有一组可选的参数列表，通过“`->`”表示列表的结束，

通过例子最容易理解闭包了，图2.5显示一个简单的闭包，这个闭包被传递给`list.each`方法，这个列表有1, 2, 3三个元素。

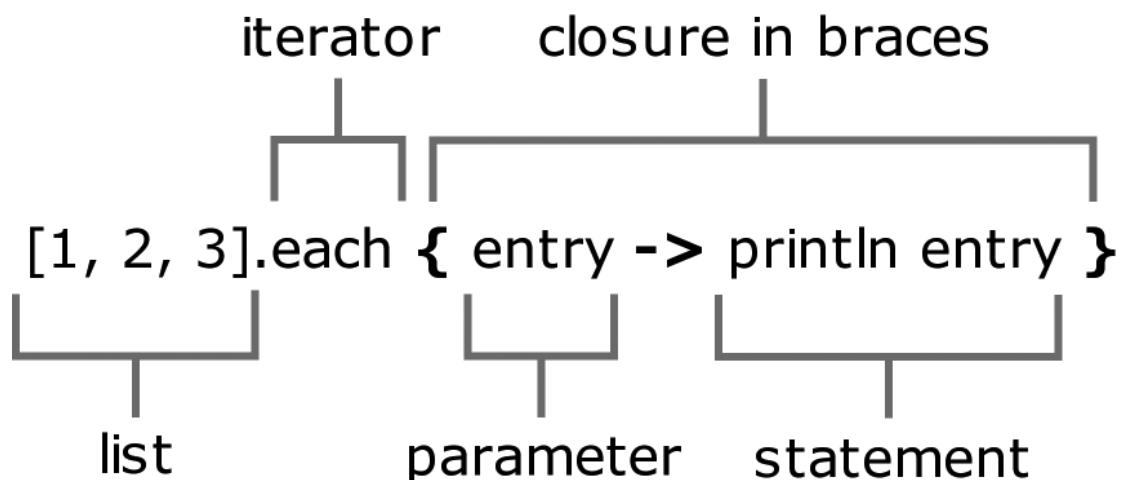


图2.5用来打印数字1、2和3的简单闭包例子

List.each方法需要一个简单的参数——闭包，然后遍历list的元素，每次遍历时将当前的元素传递给闭包，有多少个元素，闭包就被调用多少次。在这个例子中，闭包的主体是打印传递给闭包的参数，这个参数在这个例子中的名称为“entry”。

思考一个稍微复杂点的问题：如果n个人在会议室，并且每两个人之间进行碰杯，有多少次碰杯？图2.6 是5个人的描述图，每一条线代表一次碰杯。

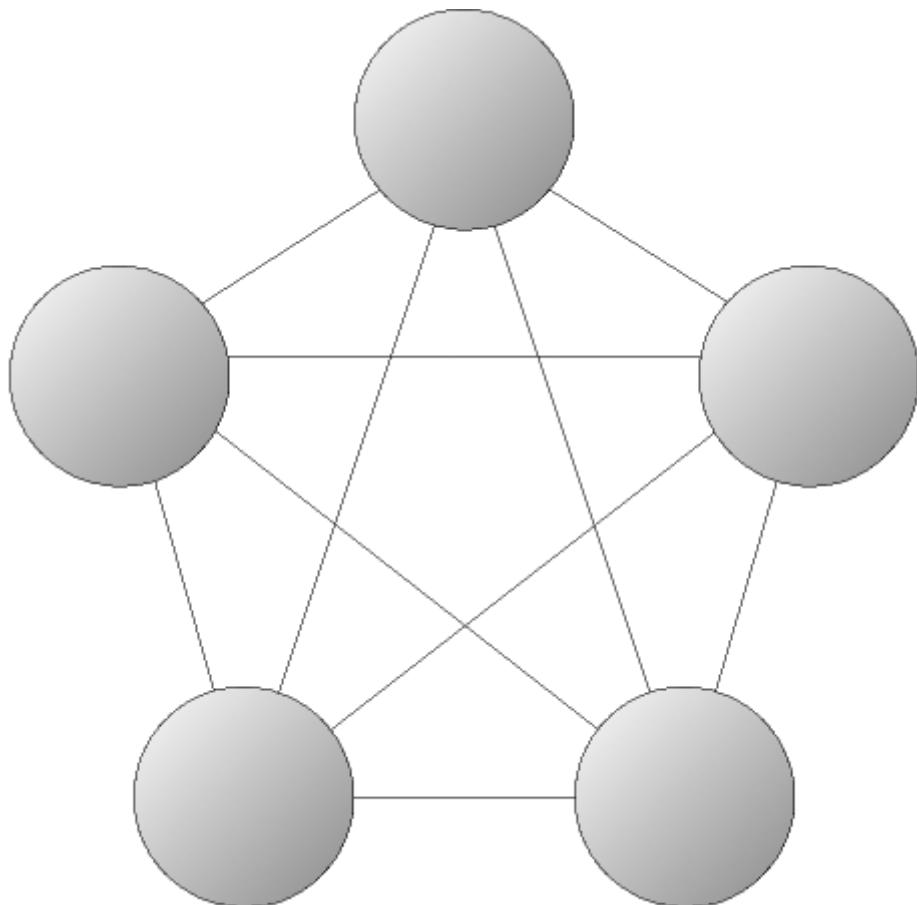


图2.6

为了回答这个问题。我们使用类Integer的upto方法，这个方法用来从当前整数值到结束的整数值之间，为每个整数做一些工作，我们应用这个方法来处理所有到达会议室的人员，在一个人员到达会议室之后，这个人和已经到达会议室的每个人进行碰杯，通过这个方法，每一个人都将与会议室的所有人碰杯。

列表2.5显示了计算碰杯数的代码，我们保留碰杯的总数量，并且在每个客人到达的时候，我们累加这个客户的碰杯数（会议室的客人数 - 1），最后，我们使用高斯原则来验证这个结果——100个人有4950次碰杯。

表 2.5 使用闭包计算所有在会议室的客人之间的碰杯数

```
def totalClinks = 0
def partyPeople = 100
1.upto(partyPeople) { guestNumber ->
    clinksWithGuest = guestNumber-1
    totalClinks += clinksWithGuest
}
assert totalClinks == (partyPeople*(partyPeople-1))/2
```

这个代码在java中 怎么实现呢？在java中，也许使用像下面代码一样的处理方式，类声明和main方法的声明被忽略：

```
//Java
int totalClinks = 0;
for(int guestNumber = 1;
    guestNumber <= partyPeople;
    guestNumber++) {
    int clinksWithGuest = guestNumber-1;
    totalClinks += clinksWithGuest;
}
```

注意guestNumber在java代码中出现了4次而在groovy代码中只出现了2次，不要觉得这好像是一件小事情，程序员应该通过最简单的代码尽可能清晰的描述问题，使用2次表述比使用4次是一个重要的简化。

upto方法封装和隐藏了怎么样迭代一个整数的逻辑，这也意味着这个逻辑在代码中只出现了一次（在upto的实现中），结果与java计算的结果是一样的，但在java中确出现了重复的变量数量。

闭包有许多重要的概念将在第五章进行描述。

### 2.3.8 groovy 结构控制

结构控制是让一个程序语言控制代码执行的流程，groovy中的结构控制简单的如if-else, while, switch和try-catch-finally，这与java的结构控制是一样的。在条件中，null被处理成false，not-null被处理成true，for循环有一个for(i in

`x) {body}`的标记，`x`可以是任何对象，`groovy`知道怎样迭代它，例如`Iterator/enumeration/collection/range/map/`任何对象，正如第六章介绍的那样，在`groovy`中，`for`循环经常被有闭包作为参数的迭代方法替代，列表2.6给出了一个例子。

表 2.6 控制结构

```
//在一行的if语句
if (false) assert false

//null表示false
if (null){
    assert false
}
else{
    assert true
}

//典型的while
def i = 0
while (i < 10) {
    i++
}
assert i == 10

//迭代一个range
def clinks = 0
for (remainingGuests in 0..9) {
    clinks += remainingGuests
}
assert clinks == (10*9)/2

//迭代一个列表
def list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for (j in list) {
    assert j == list[j]
}

//以闭包为参数的each方法
list.each() { item ->
    assert item == list[item]
}
```

```
//典型的switch
switch(3) {
    case 1 : assert false; break
    case 3 : assert true; break
    default: assert false
}
```

表2.6的代码本身就能说明问题，groovy控制结构跟java的语法十分相似，另外，在第六章将了解到groovy控制结构的完整信息。

能感受到groovy的印象，它是混合了java友好性和一些新的功能的语言

明白了如何写第一个groovy代码，现在该看看如何在java平台运行groovy代码了。

## 2.4 在 java 环境中运行 groovy

有趣的Groovy背后是恐怖的java世界，检查groovy类是如何开始进入java环境的，groovy是如何扩充java类库的，最后是groovy的老一套：groovy类动态性的初步解释。

### 2.4.1 我的类就是你的类

“Mi casa es su casa”——我家就是你家，这是西班牙语中表示好客的方式，groovy类和java类之间也存在大量相似的地方。

直到现在，当谈论groovy和java的区别的时候，比较的是两者源代码的区别，但是groovy联合java之后非常强大，在后台，所有的groovy代码都运行在JVM中并且使用的是java对象模型，不管你写的是groovy类，或者是groovy脚本，它们都作为java类在JVM中运行。

在JVM中运行groovy类有两种方式：

- 使用groovyc编译所有的\*.groovy为java的\*.class文件，把这些\*.class文件放在java类路径中，通过java类加载器来加载这些类。
- 通过groovy的类加载器在运行时直接加载\*.groovy文件并且生成对象，在这种方式下，没有生成任何\*.class，但是生成了一个java.lang.Class对象的实例，也就是说，当groovy代码中包括一个new MyClass()的表达式时，并且也有一个MyClass.groovy的文件，这个文件将被解释，一个MyClass的类型将被产生并且

增加到类加载器中，在代码中将像从\*.class一样获取到MyClass对象。

图2.7介绍了转换\*.groovy文件为java类的两种方法，任何一种途径产生的类都有相同的java类格式，groovy在源代码级增强java，但是在字节码是与java一样的。

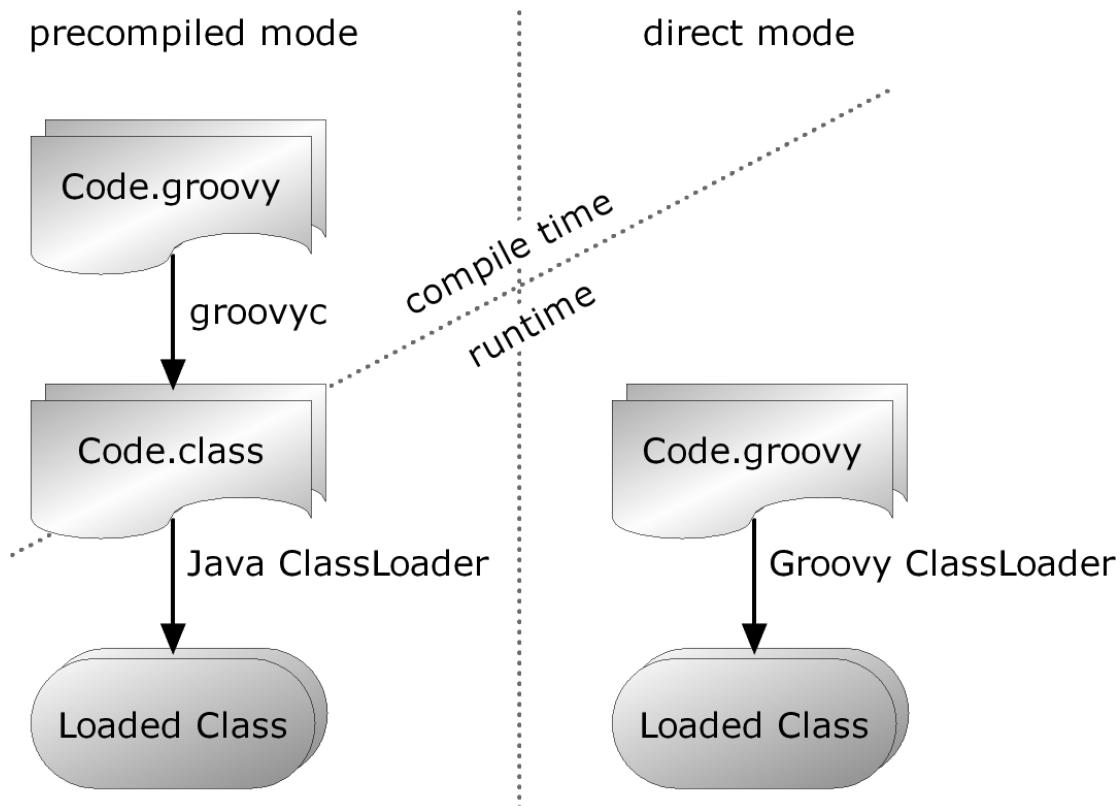


图2.7 groovy源代码可以使用groovyc编译，然后通过java类加载器加载，也可以直接通过groovy类加载器直接加载

## 2.4.2 GDK:groovy类库

Groovy紧密与java联系，在groovy中十分容易的使用java类，反过来也一样的容易，因为他们有相同的对象，groovy对象和java对象之间不存在桥梁，在这本书的例子代码中，每一个groovy对象都是一个java对象实例，在运行时他们都指向同一个对象。

对于java程序员来说这是一个巨大的优势，因为他们学习到的java知识能起到杠杆作用，考虑一下在groovy中一个字符串：

```
'Hello world!'
```

因为这是一个java.lang.String, java程序员知道可以在这个字符串对象上使用JDK方法String.startsWith:

```
if ('Hello World!'.startsWith('Hello')) {  
    // Code to execute if the string starts with 'Hello'  
}
```

Groovy的类库是JDK类库的扩展, groovy类库提供了一些新的类(例如, 简化数据库访问和XML处理的类), 也为已经存在的java类增加了功能, 这些附加的功能就是GDK, 为java类在兼容性、功能性和可表达性方面提供了重要的优势。

一个有趣的例子是在GDK中被使用的size方法, 这个方法可以用于任何对象:  
strings, arrays, lists, maps和别的容器, 在背后, 他们都是JDK的类, 这是对JDK的重大改进, 列表2.1显示了groovy的size方法对应的java对象的不同方法。

我想你应该赞同GDK的解决方案, 这样更加一致, 并且更容易记住。

表2.1 在JDK中size方法的不同表现形式

| Type         | Determine the size in JDK via...         | Groovy        |
|--------------|--|---------------|
| Array        | length field                             | size() method |
| Array        | java.lang.reflect.Array.getLength(array) | size() method |
| String       | length() method                          | size() method |
| StringBuffer | length() method                          | size() method |
| Collection   | size() method                            | size() method |
| Map          | size() method                            | size() method |
| File         | length() method                          | size() method |
| Matcher      | groupCount() method                      | size() method |

Groovy通过一个名叫MetaClass的装备来技巧性的过滤对对象所有方法的调用, 这样允许动态分配方法, 包括解决附加方法到存在的类, 在下章将了解到MetaClass的更多信息。

在这本书后面描述内建的数据类型的时候, 我们也会提到突出的GDK属性, 附录C包括一个完整的列表。

为了帮助读者理解Groovy对象的巨大优势, 我们在接下来的部分将描述groovy对象是如何开始存在的。

### 2.4.3 groovy 的生命周期

虽然java在运行时理解编译后的groovy类没有任何问题，但是它不能懂得.groovy的源文件，如果要在运行时动态加载“.groovy”文件，groovy需要在后台做更多的工作。

这一节要求读者了解java相关的高级知识，如果一点都不了解类加载器，那么跳过这个主题，假定groovy源代码被正确的转换为java字节码。不需要完全懂到底发生了什么，不至于为学习groovy而失眠。

Groovy的语法是面向行的，但是执行groovy代码确不是这样的，不像别的脚本语言，groovy代码不是一行一行的解释执行的。

相反，groovy代码被完整的转换，通过转换器产生一个java类，产生的这个类是groovy和java之间的粘合剂。产生的groovy类的格式与java字节码的格式是一样的。

在java运行的时候，类通过类加载器进行管理，当请求一个java类加载器加载一个类的时候，java类加载器从\*.class文件中加载这个类，存储在缓存中，然后返回这个类。由于一个groovy产生的类也是一个java类，所以这个类也可以通过类加载器的相同行为进行管理。差别在于groovy的类加载器能够从\*.groovy文件加载类（在放入缓存之前进行转换和生成类）。

Groovy在运行时可以像读取\*.class文件一样读取\*.groovy文件，在运行的groovyc编译器产生了相应的类，编译器使用转换和产生类的相同的机制简单的把\*.groovy文件转换为\*.class文件。

#### 在运行时生成groovy类

假设有一个groovy脚本，这个脚本保存在MyScript.groovy文件中，通过命令groovy MyScript.groovy 来运行这个脚本，下面是这个类产生的步骤，就像图2.7显示的那样：

- 1、 MyScript.groovy被传递给groovy的转换器；
- 2、 转换器产生一个抽象语法树（AST）来表示在MyScript.groovy中的所有代码；
- 3、 Groovy类生成器根据AST产生java字节码，根据脚本的内容，结果可能是多个类，现在类通过groovy类加载器是可以使用的了；

4、 Java运行时像调用一个java类MyScript一样来调用第三部产生的类；

图2.8显示了第二种途径，当用groovyc代替groovy时，产生的类被写到\*.class文件中，这两种方式使用同样的类生成机制。

所有的这些工作都在后台进行，以至于你会感觉groovy像解释执行语言。但是实际上它不是，在使用之前类已经完整的构建并且在运行时不能进行更改（不排除在.groovy被修改之后对类在运行时的替换）。

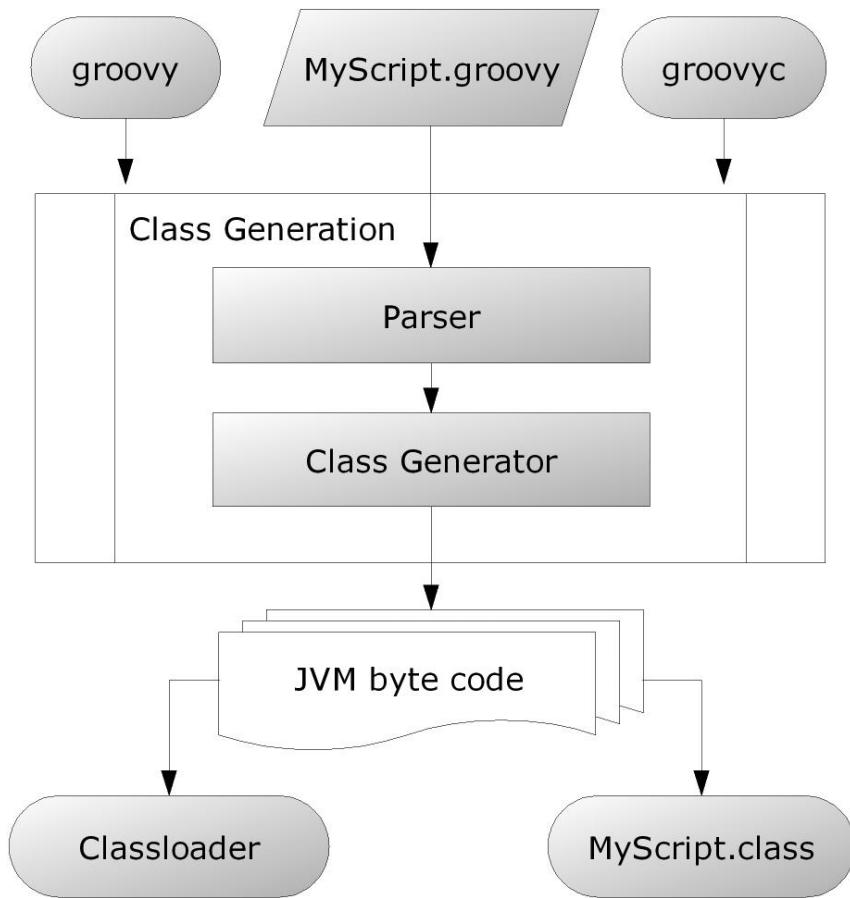


图2.8 在运行环境或者编译到一个class文件的时候，groovy字节码生成流程

根据这个描述，你能恰当的了解到groovy是怎样的动态语言，所有groovy代码都是静态java类格式，正如你所见，groovy通过聪明的途径来执行类的构建和方法调用。

### Groovy是动态的

使动态语言如此强大的原因是在表面上看来在运行时修改类的能力，例如，增加新的方

法，但是，刚刚说到了，groovy类一次性产生并且在加载之后不能改变字节码，类不能改变，那么又怎么能增加方法呢？答案非常简单而又微妙。

Groovy类生成器生成的字节码必然不同于java编译器生成的字节码（不是在格式方面，而是在内容方面），假设一个groovy文件包括一个像foo语句，groovy产生的字节码不是直接调用这个方法的，而是像这样：

```
getMetaClass().invokeMethod(this, "foo", EMPTY_PARAMS_ARRAY)
```

也就是说，方法的调用被重定向为通过对象的MetaClass进行处理，这个MetaClass现在能在运行时处理如拦截、重定向、增加/删除方法等等之类的方法调用。这个原则运用到所有通过groovy代码的处理，不管方法是否在groovy对象中或者在java对象中，记住，这没有区别。

动态代码的第二个选项是把代码放在一个字符串中并且通过groovy来运行它，在第11章将了解到这个工作是如何完成的。一个字符串可以直接根据一定的逻辑进行构建，但是当心：可能你会被动态代码的复杂性吓到。

这里有一个构建两个字符串并且计算结果的例子：

```
def code = '1 + '  
code += System.getProperty('os.version')  
  
//prints "1 + 5.1"  
println code  
  
//prints "6.1"  
println evaluate(code)
```

注意代码是一个原始的字符串，这个字符串的内容为“1 + 5.1”，这是一个有效的groovy表达式（其实是一个脚本），替代一个程序员编写这个表达式（`println 1 + 5.1`），程序在运行时把它们放在一起，`evaluate`方法最后来执行这个语句。

等等——我们不是说一行一行的执行代码是不可能的，并且代码已经作为一个类被完整的构建了吗？这个代码怎么能执行？答案非常简单，还记得图2.7左边的路径吗？类能够在运行时生成，唯一的区别是输入的字符串也可以像\*.groovy一样生成类。

根据代码的任意字符串来运行的能力是脚本语言的显著特征，这意味着groovy能像脚

本语言一样使用，虽然groovy本身是一个一般的编程语言。

## 2.5 概要

这就是我们首先了解到的groovy的知识，不必担心没有理解我们讨论到的每一件事，在接下来的章节将详细的讨论这些内容。

先介绍了groovy代码使用的断言特性，这样我们在展示我们的的代码结果的时候，通过断言，能自动的判断代码的结果是否正确。

对groovy代码的第一印象是通过在同一时间对比java和groovy的异同。Groovy在定义类、对象、和方法的时候与java是相似的，groovy使用的关键字、花括号、方括号和圆括号也与java非常相似；groovy显得更轻巧，代码更少，较少的声明方式和较少的代码，这也许意味着你需要改变阅读代码的速度：groovy通过较少的代码做了更多的事情，因此你必须慢慢的阅读代码——至少开始是这样的。

在运行之前，Groovy是兼容于java并且符合java类构建协议的，但是groovy在运行时仍旧是动态生成类，通过MetaClass，groovy能修改groovy调用者调用的任何方法。在第七章将深入讨论这方面的知识，groovy使用这种机制增加新的能力给现有的JDK类库，这些就是GDK。

现在，你自己能独立编写第一个groovy脚本了，试试吧！通过groovy shell (groovysh) 或者groovy console (groovyConsole)，编写自己的groovy代码。另外，你也许获得在接下来的章节才会深入讨论的知识。

在第一部分剩余的章节，我们将深入groovy，也许你还不了解，不必担心，我们的groovy代码会保持足够的简单的。

# 第三章 groovy 数据类型

Groovy在语言层面支持一套数据类型，这意味着groovy提供了直接声明和特定的操作符，包括了字符串、正则表达式和数字这样的简单类型，也包括范围（ranges）、列表和映射相关的集合类型，这一章将讨论简单类型，下一章将讨论集合类型。

在进行详细讨论之前，先介绍groovy的通用的处理方式，这样使你能意识到在groovy中一切事物都是对象，所有的操作符都是作为方法调用进行处理的。然后你将明白，相对于java分开处理专有类型和引用类型，groovy怎样在语言层面改善面向对象的级别。

然后分别介绍groovy原生支持的数据类型，这学习了这章之后，你一定能自信的使用groovy的简单数据类型并且对“1+1”有一个全新的理解。

## 3.1 无处不在的对象

在groovy中，万事万物皆对象，毕竟groovy是一个面向对象的语言。Groovy不像java那样除了内建数据类型之外才是面向对象。为了说明groovy设计者的选择，我们首先看看java的基本数据类型，然后介绍groovy的处理方式，最后介绍groovy和java怎么样进行自动封装和拆装基本类型。

### 3.1.1 java 类型——专有类型和引用类型

Java对专有类型（比如int/double/char/byte）和引用类型（比如Object/String）采用不同的处理方式，java有固定的专有类型，并且它们是值类型——这种类型的变量的值实际上只是一个数字（或者字符，或者true/false值），在java中不能创建自定义的值类型。

引用类型（除了专有类型之外的任何类型）——这种类型的变量实际上是指向一个对象。有C/C++背景的读者也许把引用类型想象成指针——这是相似的概念。如果改变引用类型的变量的值，这是没有改变到先前引用的对象——这仅仅是改变变量本身，让这个变量引用一个不同的对象，或者根本不引用任何对象。

不能在专有类型上进行方法调用，并且在java中也不能像java.lang.Object对象一样对待专有类型，在使用容器的时候不能处理专有类型时特别痛苦，比如

java.util.ArrayList。java的每个专有类型都有一个相应的包装类型，这个包装类型是一个用来存储专有类型值的引用类型，例如：int的包装类型是java.lang.Integer。

另一方面，比如在 $3*2$  或者 $a*b$ 中的星号“\*”操作符不是任何引用对象都支持，而是仅仅支持专有类型（+号除外，字符串支持+号），作为一个为什么这么痛苦的例子，让我们考虑有两组整数列表的例子，并且希望将这两组数字加起来之后放在第三个列表中，java代码也许是这样的：

```
// Java code!
ArrayList results = new ArrayList();
for (int i=0; i < listOne.size(); i++)
{
    Integer first = (Integer)listOne.get(i);
    Integer second = (Integer)listTwo.get(i);
    int sum = first.intValue() + second.intValue();
    results.add (new Integer(sum));
}
```

Java5的新特性会使这个工作简单一些，但是仍旧有两种类型密切关联（int和Integer），这导致了这个加法的复杂性，java这么处理的理由是：c的遗传并且基于性能上的考虑，groovy的答案是将更多的工作留给计算机，让程序员的工作更少。

### 3.1.2 Groovy 的答案：一切都是对象

Groovy使上面的例子简单到难以置信，目前我们仅仅看到为什么一切都是对象有助于保持代码的紧凑和可读，看看上一节的代码块，能看到在循环体的最后两行的代码的问题。为了求和，必须将Integer转换为int，然后为了保存在另一个list中，我们也不得不创建一个新的Integer，groovy给java.lang.Integer增加了plus方法，然后你可以这样写代码：

```
results.add (first.plus(second))
```

到目前为止，没有任何java类库设计者考虑增加一个plus方法，groovy允许操作符

**运用在对象上**，作为上一节循环体的最后部分的替代

```
// Java  
int sum = first.intValue() + second.intValue();  
results.add (new Integer(sum));
```

使用更加可读的Groovy的解决办法

```
results.add (first + second)
```

在3.3节将学习到操作符更多的知识，并且可以根据需要自己来实现操作符。

为了使groovy实现完全的面向对象，由于JVM不支持在专有类型上的面向对象操作（如方法调用），groovy设计者决定废除专有类型，当groovy需要存储一个java专有类型的值的时候，groovy使用java平台已经存在的包装类型进行包装，表3.1是包装类型的完整列表。

表3.1 java专有类型和它们的包装类型

| Primitive type | Wrapper type        | Description                                    |
|----------------|---------------------|--|
| byte           | java.lang.Byte      | 8-bit signed integer                           |
| short          | java.lang.Short     | 16-bit signed integer                          |
| int            | java.lang.Integer   | 32-bit signed integer                          |
| long           | java.lang.Long      | 64-bit signed integer                          |
| float          | java.lang.Float     | Single-precision (32-bit) floating-point value |
| double         | java.lang.Double    | Double-precision (64-bit) floating-point value |
| char           | java.lang.Character | 16-bit Unicode character                       |
| boolean        | java.lang.Boolean   | Boolean value (true or false)                  |

在groovy代码中，任何时候看到专有类型值（比如数字5，或者布尔值true），它都是相应的包装类型实例的一个引用。为了简洁和习惯，groovy允许声明变量为专有类型的变量，但是不要被欺骗，后台真正使用的类型是包装类型，字符串和数组（arrays）没有在表3.1中列出，因为它们已经是引用类型，不是专有类型，所以不需要包装类。

尽管有了java的语法，但是仔细检查java和groovy数字的格式，也能发现它们稍微有些不同，因为groovy允许通过字面格式实例化java.math.BigDecimal和

`java.math.BigInteger`, 表3.2列出了通过字面格式可以在groovy使用的数字类型

表3.2 在groovy在的数字格式

| Type                              | Example literals                                    |
|-----------------------------------|---|
| <code>java.lang.Integer</code>    | <code>15, 0x1234ffff</code>                         |
| <code>java.lang.Long</code>       | <code>100L, 200l<sup>a</sup></code>                 |
| <code>java.lang.Float</code>      | <code>1.23f, 4.56F</code>                           |
| <code>java.lang.Double</code>     | <code>1.23d, 4.56D</code>                           |
| <code>java.math.BigInteger</code> | <code>123g, 456G</code>                             |
| <code>java.math.BigDecimal</code> | <code>1.23, 4.56, 1.4E4, 2.8e4, 1.23g, 1.23G</code> |

注意groovy是怎么样决定是使用`BigInteger`或者`BigDecimal`的，这是通过数字后面的“G”，或者是否出现小数点来决定的，此外，注意`BigDecimal`是默认的非整数类型——除非指定后缀强制类型为`Float`或者`Double`，否则将使用`BigDecimal`。

### 3.1.3 自动装箱和拆箱

在java和别的支持相同概念的语言中，转换一个专有类型的值为一个包装类型的动作叫做装箱，反向动作——把一个包装实例的值让一个专有类型接收，这样的动作叫做拆箱。Groovy在需要的时候自动进行装箱和拆箱处理，主要用在从groovy中调用java方法时，这种自动进行装箱和拆箱就是我们知道的自动包装（autoboxing）。

我们已经看到groovy被设计得与java一样好，因此，当一个java方法接受一个专有类型参数或者返回一个专有类型的时候发生了什么？怎样从groovy中调用这个方法呢？考虑一下在`java.lang.String`类中已经存在的方法：`int indexOf(int ch)`。

可以在groovy中像这样调用这个方法：

```
assert 'ABCDE'.indexOf(67) == 2
```

从groovy的观点来看，传递一个包含值67（67是字母C的Unicode值）的`Integer`实例，尽管方法期望的参数类型是专有类型`int`，groovy处理拆箱工作，方法返回一个专有类型`int`的值，在这个值返回到groovy的时候立刻自动进行装箱处理。然后在groovy

脚本中比较值为2的Integer，图3.1显示了从groovy到java和从java到groovy的处理过程。

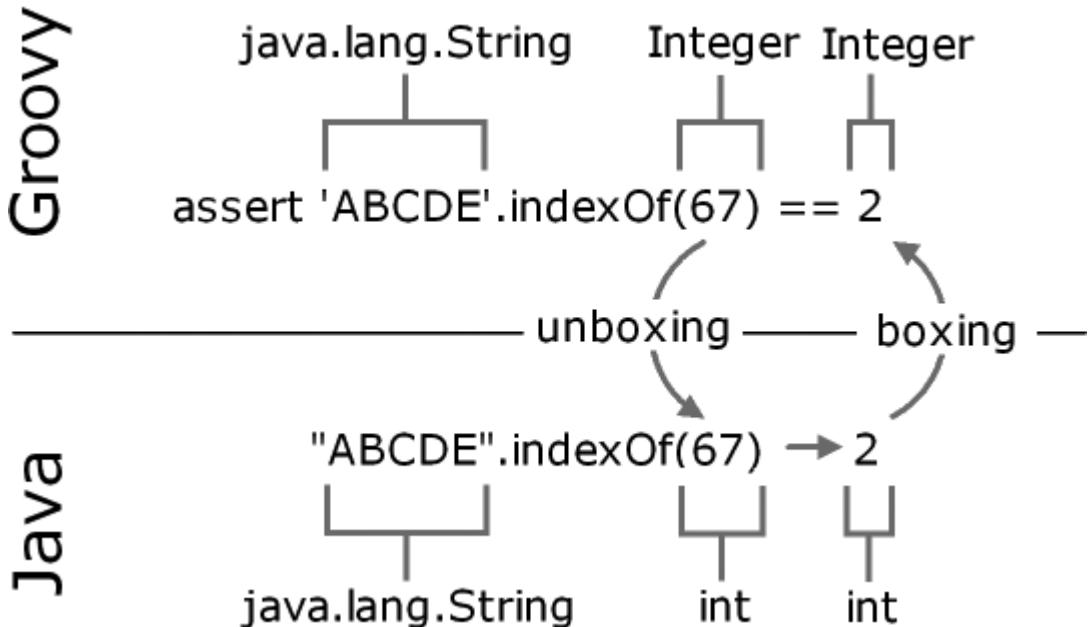


图3.1

所有这些处理都是透明的，这意味着不需要在groovy中进行任何特殊处理来触发它，现在明白了自动装箱，在一个对象上怎样应用操作符变得十分有趣，我们将在下一节探讨这个问题。

### 3.1.4 没有中间层的拆箱

如果在“1+1”中，两个数都是Integer的实例，这两个Integer自动拆箱，然后在专有类型上执行加操作的吗？

不，groovy比java更加面向对象，groovy通过“`1.plus(1)`”来执行这个表达式，调用第一个Integer对象上的`plus()`方法，并且将第二个Integer对象作为参数传递给该方法，这个方法返回一个值为2的新的Integer对象。

这是非常强大的模型，在一个对象上调用方法是面向对象语言应该做的工作，这为对象应用操作符提供了方便。

总结一下，在groovy中出现的字面符号（数字、字符串等等）没有任何问题，他们都是对象，他们仅仅传递给java时才进行装箱和拆箱，操作符是方法调用的快速途径，现在

已经弄明白了在给定期望信息的时候groovy是如何处理类型的了，我们来看看没有给定任何信息的时候是怎么做的

## 3.2 可选类型的概念

到现在为止，在groovy的例子代码中没有使用任何类型信息，没有使用与java相似的任何显示的静态类型声明，我们将字符串和数字赋值给变量，并且没有关心这些变量的类型，在后台，groovy悄悄的指定这些变量的类型为java.lang.Object，这节讨论当指定了类型的时候将发生什么，并且说明赞成和反对动态类型的理由。

### 3.2.1 指定类型

Groovy中允许像java那样显示的指定类型，表3.3给出了可选的声明静态类型和在运行时使用动态类型的例子，关键字“def”用来标记没有特定的类型要求。

表3.3groovy语句和运行时结果的类型的例子

| Statement      | Type of value     | Comment              |
|----------------|-------------------|----------------------|
| def a = 1      | java.lang.Integer | 隐式声明类                |
| def b = 1.0f   | java.lang.Float   |                      |
| int c = 1      | java.lang.Integer | 使用java专有类型的名称显示的声明类型 |
| float d = 1    | java.lang.Float   |                      |
| Integer e = 1  | java.lang.Integer | 使用引用类型的名称显示的声明类型     |
| String f = '1' | java.lang.String  |                      |

按照早期约定，声明或者造型到int或者Integer类型是不重要的，不管是int还是Integer，groovy使用的都是引用类型(Integer)，如果你喜欢简洁，并且相信阅读者能够理解自己的代码，那么使用int，如果希望groovy的新手明白正在使用的是对象，那么使用Integer。

一个变量是否显示的指定了类型是很重要的，系统是类型安全的。不像没类型的语言，groovy不允许将一个定义好的类型的变量看成另外一种类型，例如，你绝不能把

`java.lang.String`类型的“1”看成一个`java.lang.Integer`的类型。

### 3.2.2 静态类型 VS 动态类型

可以在静态类型还是动态类型之间进行选择是groovy的一项关键优势。在web上有大量关于静态类型好还是动态类型好的讨论。换句话说，立场不同，观点也不同。

静态类型为性能优化、编译时安全检查和IDE支持提供更多有用的帮助，也显示变量或者方法参数相关的附加信息及方法重载，静态类型也是从反射获取有用信息的前提。

动态类型，在另一方面，这不仅仅便于延迟程序员编写一些特定的脚本，也对保护和规避类型有用，假设从一个方法调用的结果中获取到一个对象，并且必须将这个对象作为参数不做任何修改的传递给别的方法：

```
def node = document.findMyNode()  
log.info node  
db.store node
```

在这个例子中，对`node`的类型和包名称是不感兴趣的，这样可以节省很多工作——类型的声明，导入需要的包。你只需要传达：“就只做这些事”。

动态类型第二个有用的地方是在一个对象上进行没有固定类型的方法调用，在7.3.2将进行详细的说明，这样可以实现一个高度重用的函数。

对于有java背景的程序员来说，在开始使用groovy时直接使用静态类型是十分常见的，并且逐步使用更多的动态类型，这是合理的，因为每个人都有信心使用groovy。

不管是使用动态类型还是静态类型，都能发现groovy能用较少的代码做较多的工作，接下来看看操作符重载。

## 3.3 重载操作符

重载是面向对象的一个概念，一个类型有一个特定的行为，子类型为了达到更多的目的重写这个行为，当一门语言的操作符是基于一个方法调用时，如果这些方法能够被覆盖，这种行为叫做操作符重载。

更多的常用习惯是叫做操作符过载 (operator overloading) , 这表达的意思是相同的, 不同的是过载可以有一个方法的多个实现, 不同的仅仅是方法的参数类型。

接下来将说明那些操作符可以重载, 有一个完整的例子用来显示如果在工作中进行重写, 并且在使用操作符与多个类型工作时给出一些指导。

### 3.3.1 可重写的操作符一览

就像你在3.1.2看到的那样, `1+1`仅仅是`1.plus(1)`的便利书写方式, `Integer`有一个`plus`方法的实现。

对于其他操作符来说这也是十分变量的, 表3.4显示了一个预览表。

读者可容易的在自己的类上随意的使用这些操作符, 仅仅需要实现相应的方法, 不像java, 需要实现一个特定的接口。

表3.4 操作符的方法

| Operator                             | Name                            | Method                       | Works with  |
|--------------------------------------|---------------------------------|------------------------------|---|
| <code>a + b</code>                   | Plus                            | <code>a.plus(b)</code>       | Number, string, collection  |
| <code>a - b</code>                   | Minus                           | <code>a.minus(b)</code>      | Number, string, collection  |
| <code>a * b</code>                   | Star                            | <code>a.multiply(b)</code>   | Number, string, collection  |
| <code>a / b</code>                   | Divide                          | <code>a.div(b)</code>        | Number  |
| <code>a % b</code>                   | Modulo                          | <code>a.mod(b)</code>        | Integral number   |
| <code>a++</code><br><code>++a</code> | Post increment<br>Pre increment | <code>a.next()</code>        | Number, string, range   |
| <code>a--</code><br><code>--a</code> | Post decrement<br>Pre decrement | <code>a.previous()</code>    | Number, string, range   |
| <code>a**b</code>                    | Power                           | <code>a.power(b)</code>      | Number  |
| <code>a   b</code>                   | Numerical or                    | <code>a.or(b)</code>         | Integral number   |
| <code>a &amp; b</code>               | Numerical and                   | <code>a.and(b)</code>        | Integral number   |
| <code>a ^ b</code>                   | Numerical xor                   | <code>a.xor(b)</code>        | Integral number   |
| <code>-a</code>                      | Bitwise complement              | <code>a.negate()</code>      | Integral number, string (the latter returning a regular expression pattern) |
| <code>a[b]</code>                    | Subscript                       | <code>a.getValueAt(b)</code> | Object, list, map, String, Array  |
| <code>a[b] = c</code>                | Subscript assignment            | <code>a.putAt(b, c)</code>   | Object, list, map, StringBuffer, Array                                      |

|                                    |                          |                                      |   |
|------------------------------------|--------------------------|--------------------------------------|---|
| <code>a &lt;&lt; b</code>          | Left shift               | <code>a.leftshift(b)</code>          | Integral number, also used like "append" to <code>StringBuffers</code> , <code>Writers</code> , <code>Files</code> , <code>Sockets</code> , <code>Lists</code>            |
| <code>a &gt;&gt; b</code>          | Right shift              | <code>a.rightShift(b)</code>         | Integral number   |
| <code>a &gt;&gt;&gt; b</code>      | Right shift unsigned     | <code>a.rightShiftUnsigned(b)</code> | Integral number   |
| <code>switch(a) { case b: }</code> | Classification           | <code>b.isCase(a)</code>             | Object, range, list, collection, pattern, closure; also used with collection c in <code>c.grep(b)</code> , which returns all items of c where <code>b.isCase(item)</code> |
| <code>a == b</code>                | Equals                   | <code>a.equals(b)</code>             | Object; consider <code>hashCode()</code> <sup>a</sup>   |
| <code>a != b</code>                | Not equal                | <code>! a.equals(b)</code>           | Object  |
| <code>a &lt;=&gt; b</code>         | Spaceship                | <code>a.compareTo(b)</code>          | <code>java.lang.Comparable</code>   |
| <code>a &gt; b</code>              | Greater than             | <code>a.compareTo(b) &gt; 0</code>   |   |
| <code>a &gt;= b</code>             | Greater than or equal to | <code>a.compareTo(b) &gt;= 0</code>  |   |
| <code>a &lt; b</code>              | Less than                | <code>a.compareTo(b) &lt; 0</code>   |   |
| <code>a &lt;= b</code>             | Less than or equal to    | <code>a.compareTo(b) &lt;= 0</code>  |   |
| <code>a as type</code>             | Enforced coercion        | <code>a.asType(typeClass)</code>     | Any type  |

当重写`equals`方法时，java强烈建议开发人员也重写`hashCode()`方法，这样相等的两个对象也有相同的`hashCode`（而有相同`hashCode`的方法未必相等）参考javaAPI文档关于`java.lang.Object#equals`的描述。

注意：严格意义上来说，表3.4中显示groovy已经有了很多操作符，比如“.”操作符用来引用字段和方法，他们的行为也可以重写，在第七章将介绍这些知识。

这是非常好的理论，接下来看看在工作中如何应用这些知识。

### 3.3.2 重写操作符实战

列表3.1演示了Money类的`equals`（`==`）和`plus`（`+`）操作符的实现，这是值对象模式（参考<http://c2.com/cgi/wiki?ValueObject>）一个低级别的实现，这允许相同货币构成的Money可以相加，而不同货币构成的Money不能相加。

实现equals的目的是保证这个对象可以与null对象进行对比。这是groovy风格，equals操作的缺省实现是不抛出NullPointerException，记住：“==”或者equals指示对象是否同等（值相等），而不是是否指向同一个对象。

### Listing 3.1 Operator override

```
class Money {  
    private int amount  
    private String currency  
    Money(amountValue, currencyValue) {  
        amount = amountValue  
        currency = currencyValue  
    }  
    boolean equals(Object other) {  
        if (null == other) return false  
        if (!(other instanceof Money)) return false  
        if (currency != other.currency) return false  
        if (amount != other.amount) return false  
        return true  
    }  
    int hashCode() {  
        return amount.hashCode() + currency.hashCode()  
    }  
    Money plus(Money other) {  
        if (null == other) return null  
        if (other.currency != currency) {  
            throw new IllegalArgumentException(  
                "cannot add ${other.currency} to $currency")  
        }  
        return new Money(amount + other.amount, currency)  
    }  
}  
  
def buck = new Money(1, 'USD')  
assert buck  
assert buck == new Money(1, 'USD')  
assert buck + buck == new Money(2, 'USD')
```

① 重写==操作符  
② 实现+操作符  
③ 调用重写的==操作符  
④ 调用+操作符

重写的equals是易懂的，就像（1）显示的那样，例子代码也提供了一个hashCode方法用来保证等值的Money对象有相同的hashCode。（3）显示了这个操作符的用法。

在（2）处，plus操作符严格上说没有重写，因为在Money的父类（Object）上没有任何操作符，在这个例子里，操作符实现是最佳的说法，在（4）处使用这个操作符将两个Money对象进行相加。

为了解释重写和重载的区别，这里可能为Money的plus操作符进行了重载，在列表3.1，Money仅仅能和另外一个Money对象相加，在例子中，我们像这样加Money：

```
assert buck + 1 == new Money(2, 'USD')
```

我们能提供一个另外的方法

```
Money plus (Integer more) {  
    return new Money(amount + more, currency)  
}
```

这是第二个实现方法，这个方法接受一个Integer类型的参数，这个plus方法是重载的，groovy在运行时根据方法的实现进行正确的方法分派。

*注意：在这两个例子中，Money的plus操作都返回Money对象，这样的操作叫Money的plus操作，这是在封闭的类型中进行的，无论在Money实例上执行什么操作，都是得到一个Money实例的结果。*

这个例子说明了在实现一个操作符方法的时候怎么样处理不同的参数类型的问题。在一节看看这些方面的问题。

### 3.3.3 确保正确的工作

当两个数的类型相同时实现操作符是非常简单的，不同类型之间进行操作要复杂一些，如：

```
1+1.0
```

这是一个Integer和BigDecimal的加法，将返回什么类型呢？3.6节回答了这个问题，但这是个一般问题，两个参数中的一个需要升到更通用的类型，这叫做类型强制转换（coercion）。

在实现操作符的时候，类型强制转换有三个主要的问题需要考虑。

#### 支持的参数类型

实现操作符的时候需要考虑哪种参数类型和值是被允许的，如果一个操作符接收了一个不适当的参数类型，那么在必要的时候要抛出IllegalArgumentException。例如，在我们的Money的例子中，尽管能使用Money可以作为plus操作符的参数，但这里不运行不

同的货币进行相加。

### 设置更明确的参数

如果参数类型比定义的更明确（译者注：定义的参数类型是父类（或者接口），传递给操作符的参数是子类（或者实现类）），那么groovy将使用和返回定义的类型，为了明白这个意思，思考一下如何为BigDecimal类设计plus操作符，该操作符接受一个Integer类型的参数。

Integer比BigDecimal更明确，每一个Integer都能表示为一个BigDecimal，但反过来是不行的，所以对于BigDecimal.plus(Integer)操作符来说，我们考虑把Integer提升到BigDecimal来执行加操作，然后返回另外一个BigDecimal——即使是一个整数结果。

### 使用双派发处理更多的通用参数

如果参数的类型更加通用，使用当前对象作为参数调用参数自身的操作符方法（用this表示当前对象），将操作符方法派给参数的操作符方法，这也叫做双派发，并且这能避免代码重复，和可能不一致的代码，反过来考虑一下前面的例子Integer.plus(BigDecimal operand)。

考虑表达式operand.plus(this)表达式的结果，把工作代理给BigDecimal的plus(Integer)方法，结果是一个BigDecimal，这是合理的， $1 + 1.5$ 返回一个Integer，但是 $1.5 + 1$ 返回BigDecimal。

当然，这只适用于可交换的操作符，必须严格测试，避免死循环。

### Groovy的示量行为

Groovy的类型处理策略是返回一个更一般的类型，别的语言如Ruby试图聪明的、在不丢失精度和范围的情况下返回一个小一点的类型。Ruby的方式在程序处理的时候节省了内存。这样如果数字范围溢出要求语言自动提升数字的类型，否则，返回的结果是一个被截取了精度的数。

现在，明白了groovy是如何处理类型的差异，我们能深入研究在语言级别为每个数据类型提供了怎样的支持，我们开始使用的类型也许是另外一个非数字类型：字符串。

## 3.4 使用字符串

字符串使用得非常广泛，许多语言——包括java提供了语言级别的支持，以使字符串工作更容易一些。脚本语言在字符串处理方面比主流的应用程序开发语言做的更好，groovy提供了许多附加的特性，这节看看groovy在字符串处理方面提供了那些有用特性。

Groovy的字符串有两种风格：一般字符串和GString，一般的字符串是java.lang.String的实例，GString是groovy.lang.GString的实例，GString允许有占位符并且允许在运行时对占位符进行解析和计算。许多脚本语言都有相似的功能，通常叫做字符串修改，但是这些功能比groovy的GString的特性更原始，下面我们来看看字符串的每一种风格和他们在代码中是如何出现的。

### 3.4.1 字符串的样式

Java仅仅允许将文本放在双引号——"like this"这样风格的字符串，如果你想嵌入动态值到字符串中，必须调用格式化方法（在java1.5中提供，容易了一些但是仍然有些复杂）或者进行字符串拼接。如果在字符串中包括许多斜杠“\”（比如windows文件名和正则表达式），代码将变得非常难读，因为你必须使用双斜杠。如果在源代码中有大量的文本需要分布在多行，那么必须在每一行包含一个完整的字符串（或者几个完整的字符串）。

Groovy认识到这中间的不足，所以它提供了更多的选择，总结在表3.5中

| Start/end characters       | Example                                       | GString aware? | Backslash escapes?        |
|----------------------------|---|----------------|---------------------------|
| Single quote               | 'hello Dierk'                                 | No             | Yes                       |
| Double quote               | "hello \$name"                                | Yes            | Yes                       |
| Triple single quote (''')  | '''-----<br>Total: \$0.02<br>-----'''         | No             | Yes                       |
| Triple double quote ("""") | """first line<br>second line<br>third line""" | Yes            | Yes                       |
| Forward slash              | /x(\d*)y/                                     | Yes            | Occasionally <sup>a</sup> |

<sup>a</sup> 这种类型的看点是避免进行字符转义，所以在语言层面进行避免是可以做到的，剩余的就是用\u表示的unicode的支持和\\$，除非\\$用来标明是模式的结束，参考groovy的

语言规范。

每一种表示法的目的都是为了使文本内容最小化，每一个表示法都有一个和其他表示法不一样的属性：

- 单引号所表示的字符串不会按照GString的类型来处理内容，这等价于java的字符串；
- 双引号表示的意思与单引号表示的意思是等价的。如果字符串内容中包括没有被转义的\$符号，那么它被加工成GString实例，GString更详细的信息将在下节介绍；
- 三组引号（或者是多行字符串）允许字符串的内容在多行出现，新的行总是被转换为“\n”，其他所有的空白字符都被完整的按照文本原样保留，多行字符串也许是一个GString实例，这根据是使用单引号或者多双引号而定，多行字符串事实上像Ruby或者Perl中的HERE-document。
- “/”表示的字符串，指明字符串内容不转义反斜杠“\”，这在正则表达式的使用中特别有用，就象后面看到的那样，只有在一个反斜杠接下来是一个字符u的时候才需要进行转义——这稍微有点麻烦，因为\u用来表示一个unicode转义。

就像早些时候提到的那样，groovy使用与java相似的机制处理特殊的字符，如换行符和制表符。另外，“\$”符号在groovy也可进行转义，避免编译器将字符串处理成GString实例，列表3.6是完整的转义字符列表。

列表3.6 在groovy已知的转义字符

| Escaped special character | Meaning  |
|---------------------------|--|
| \b                        | Backspace  |
| \t                        | Tab  |
| \r                        | Carriage return  |
| \n                        | Line feed  |
| \f                        | Form feed  |
| \\\                       | Backslash  |
| \\$                       | Dollar sign  |
| \uabcd                    | Unicode character U+abcd (where a, b, c and d are hex digits)                          |
| \abc <sup>a</sup>         | Unicode character U+abc (where a, b, and c are octal digits, and b and c are optional) |
| \'                        | Single quote   |
| \"                        | Double quote   |

注意，在双引号字符串中，单引号不需要进行转义，反过来也是一样。换句话说，'I said,"hi."'和"don't"这两者都会有你希望的结果，基于一致性的原因，这两者仍然可以进行转义，同样的，\$符号在单引号字符串中也能进行转义，即使它们不是必须的，这样使得切换工作会十分容易。

注意，java使用单引号表示一个字符，但是就像你看到的那样，在groovy中不能这样做，因为单引号已经用来表示字符串了，不过在显式声明类型的时候，在groovy中能够获得和java相同的结果：

```
char a = 'x'
```

或者

```
Character b = 'x'
```

java.lang.String对象'x'将被强制转换为一个java.lang.Character，如果在别的时候想将一个字符串转换为一个字符，可以采用下面的两种方式：

```
'x' as char
```

或者

```
'x'.toCharacter()
```

作为GDK吸引人的特性，groovy在字符串中包括许多有用的方法，如toInteger, toLong, toFloat和toDouble。

究竟使用那种方式，除非编译器决定他是Gstring对象，否则将是一个java.lang.String对象实例，就像java字符串一样。到现在，我们仅仅间接的提到Gstring的功能，现在是介绍它的时候了。

### 3.4.2 使用 Gstring 进行工作

Gstring像增加了额外功能的字符串（实际上是groovy.lang.Gstring，而不是java.lang.String的子类，因为String的类是final的，不管怎样，Gstring能像String一样使用，在需要的时候Groovy自动将Gstring转换为String），通过双引号进行声明，决定双引号声明的字符串是一个Gstring实例的特征是字符串内容中是否出现占位符，占位符也许通过完整的表达式语法出现“\${表达式}”，或者以一个简单的“\$reference”语法形式出现，看看表3.2的例子：

Listing 3.2 Working with GStrings

```
me      = 'Tarzan'  
you    = 'Jane'  
line   = "me $me - you $you"  
assert line == 'me Tarzan - you Jane'  
  
date = new Date(0)  
out  = "Year $date.year Month $date.month Day $date.date"  
assert out == 'Year 70 Month 0 Day 1'  
  
out = "Date is ${date.toGMTString()}!"  
assert out == 'Date is 1 Jan 1970 00:00:00 GMT !'  
  
sql = """  
SELECT FROM MyTable  
  WHERE Year = $date.year  
"""  
assert sql == """  
SELECT FROM MyTable  
  WHERE Year = 70  
"""  
  
out = "my 0.02\$"  
assert out == 'my 0.02$'
```

① 简单的\$语法

② 扩展的简单语法

③ 使用花括号表示的完整语法

④ 多行的GString

⑤ \$作为字符串的一部分

在一个Gstring中，可以通过\$符号简单的引用到一个变量，这种最简单的方式如（1）显示，而（2）显示了通过“.”符号访问到变量对象的属性，在第七章将学习到

更多关于属性访问的知识。

(3) 显示了通过 \${ } 符号表示的完全表达式，在花括号中可以是任意的groovy 表达式，花括号表示一个闭包。

在实际生活中，`Gstring`是方便的模板脚本，在(4)中使用`Gstring`创建了一个SQL查询语句，groovy提供了更多的模板支持，第八章将介绍这些模板，如果在模板（或者别的`Gstring`）中需要使用到一个\$符号，必须像(5)示例那样使用“\”进行转义。

虽然`Gstring`通常被像`java.lang.String`那样使用，但他们的实现不同于固定的字符串，并且动态的部分（也叫`values`）是独立的，如下代码的显示：

```
me = 'Tarzan'  
you = 'Jane'  
line = "me $me - you $you"  
assert line == 'me Tarzan - you Jane'  
assert line instanceof GString  
assert line.strings[0] == 'me '  
assert line.strings[1] == ' - you '  
assert line.values[0] == 'Tarzan'  
assert line.values[1] == 'Jane'
```

(`Gstring`中的每个值都是在声明的时候进行绑定的，在`Gstring`被转换为`java.lang.String`的时候（`toString`方法被显式调用或者隐式调用）的时候，每个值被写到字符串中，由于写一个值的逻辑是复杂的，这种行为能用于各种高级的途径，参考13章）。

现在你已经了解到了groovy对字符串的支持，接下来说说在groovy类库中是如何使用这些字符串的，这将让你对java和groovy的相互作用有第一印象，我们将从典型的java风格逐步过渡到groovy模式，仔细的观察每一个步骤。

### 3.4.3 从 java 到 groovy

现在声明字符串是很容易的，这些字符串都是`java.lang.String`对象，我们可以调用在这些字符串上的方法或者把它们作为参数传递给一个期望的字符串，比如容易的控制台输出：

```
System.out.print("Hello Groovy!");
```

这行代码在java和groovy中是等价的，也可以使用单引号传递一个groovy字符串：

```
System.out.print('Hello Groovy!');
```

由于这是十分通用的一个任务，GDK提供了一个快捷的语法：

```
print('Hello Groovy!');
```

也可以删除圆括号和分号，因为这是可选的，并且在这个例子中对可读性也没有什么帮助。Groovy可以将结果浓缩为：

```
print 'Hello Groovy!'
```

只看最后一行，我们不能分辨出这是groovy、Ruby、Perl或者别的脚本语言中的哪一种，看起来这行代码在某种程度上也不优雅，但是它在最简单的途径来表述问题。

列表3.3提供了java和附加的GDK交织在一起的功能，你如何来分辨每一行代码？

### Listing 3.3 What to do with strings

```
greeting = 'Hello Groovy!'

assert greeting.startsWith('Hello')

assert greeting.indexOf('Groovy') >= 0
assert greeting.contains('Groovy')

assert greeting[6..11] == 'Groovy'

assert 'Hi' + greeting -> 'Hi Groovy!'

assert greeting.count('o') == 3

assert 'x'.padLeft(3) == ' x'
assert 'x'.padRight(3, '_') == 'x__'
assert 'x'.center(3) == ' x '
assert 'x' * 3 == 'xxx'
```

这些具有自我描述的例子给出了在groovy的字符串中的印象，如果读者使用过任

何别的脚本语言，也许已经注意到在列表3.3缺失一个有用的函数：修改一个字符串。

groovy不能这样做，因为字符串是一个java.lang.String对象实例，并且java.lang.String在java中是不可变的。

在说是“蹩脚的理由”之前，groovy修改字符串的答案是：虽然不能使用String，但是可以使用StringBuffer！在StringBuffer中，可以使用<<操作符追加文本和下标操作符进行文本替换，在一个字符串上使用<<操作符将返回一个StringBuffer，下面是表3.3通过使用StringBuffer表示的等价的例子：

```
greeting = 'Hello'  
greeting <<= ' Groovy'    ← (1) 追加文本和赋值一起完成  
assert greeting instanceof java.lang.StringBuffer  
greeting << '!'           ← (2) 在StringBuffer上追加文本  
assert greeting.toString() == 'Hello Groovy!'  
greeting[1..4] = 'i'        ← 将子串"ello"替换为"i"  
assert greeting.toString() == 'Hi Groovy!'
```

注意：虽然表达式stringRef<<string 返回的是一个StringBuffer，但是这个StringBuffer并没有自动赋值给stringRef(如(1)所示)，在使用一个字符串的时候，这需要显示的进行赋值；但在使用StringBuffer的时候不需要这样做，在使用StringBuffer的时候，原来存在的对象被修改(如(2)所示)，在使用字符串的时候我们不能修改原来存在的对象，所以必须返回一个新的对象。

在下一节，读者将进一步学到关于字符串的知识，String已经在GDK中获取到了一些新的方法，我们已经看到了一部分，但是通过正则表达式和列表，我们将谈到更多的方法，在字符串上的GDK方法列表在附录C中列出。

在编程的时候使用字符串是非常平常的事情，在特定的编程：读文本、写文本、剪切文本、替换语法、分析内容、查找和替换等等，想想在我们的编程过程中，涉及到字符串的有多少？

Groovy在这些方面都提供了支持，但这不是全部，下一节将介绍正则表达式，正则表达式分解处理文本：这是困难的操作，但是功能极其强大。

## 3.5 使用正则表达式

假设必须准备这本书的一个目录列表，那么需要收集所有的像“3.5 使用正在表

74 / 213

如果满意该文档，请支持(招商银行北京分行双榆树支行 6225881008887381 账户名：吴翊)

达式”这样的标题——以一个数字开始或者一个数字，一个点和另外一个数字，其余的将是标题，这将是复杂的代码：检查书中的每一个字符，看它是否是一行的开始，如果是，那么检查它是否是一个数字，如果是，检查接下来的字符是否是一个点或者数字。——许多的串，我们甚至不能处理大量的数字。

使用正则表达式能解决这个问题，正则表达式允许声明一个模式而不是编程，由于模式只写一次，groovy能做更多方面的工作。

正则表达式是脚本语言显著的特性，并且在JDK1.4之后也是可以使用的，groovy依赖java的正则表达式并且为了便利性增加了三个操作符：

- regex查找操作符“=~”；
- regex匹配操作符“==~”；
- regex模式操作符：~String；

深入讨论正则表达式是在这本书的后面部分，我们关注的是groovy，而不是正则表达式，本书给出尽可能短的介绍，使例子可以被理解并且使读者能入门。

正则表达式通过模式进行定义，一个模式可以是任何一个简单的字符，一个固定的字符串或者一些像构成日期格式的数字和分隔符，模式使用符号序列进行声明，事实上，语言本身就是一个模式描述，在表3.7中显示了一些例子，注意这些都是未处理的模式，他们没有出现在字符串中，换句话说，如果你保存一个模式在一个变量中并且打印这个模式，这是你想要的结果，区别模式自身和在代码中的出现形式是很重要的。

表3.7 简单的正在表达式例子

| Pattern            | Meaning  |
|--------------------|--|
| some text          | 精确的文本“some text”。  |
| some\s+text        | 单词“some”后面紧接着一个或者多个空白字符，然后后面是一个单词“text”。   |
| ^\d+(\.\d+)? (.*)  | 我们介绍的例子：一级或者二级标题，“^”指明为一行的开始，\d表示数字，\d+表示一个或者多个数字，圆括号用于分组，“?”标识前面的分组是可选的，第二个分组包括了标题，“.”表示任何字符，“*”表示任意多个字符。 |
| \d\d/\d\d/\d\d\d\d | 一个日期格式，两个数字后面是一个“/”，后面再是两个数字和一个“/”，接下来是四个数字。   |

列表3.7中的例子用来说明需要查找的内容是什么，而不是必须编程来进行查找，

接下来将看到怎样在代码中应用模式并且能用模式做什么工作，然后通过一个完整的解决办法再看看最初的例子，审查正则表达式的性能和在switch语句中如何使用grep方法进行过滤。

### 3.5.1 在字符串中使用模式

怎样在一个字符串中声明一个符号序列？

在java中，这通常会引起混淆，为了传递一个反斜杠 (\) 给java字符串，模式必须使用许多反斜杠，必须双倍的输入反斜杠，这导致模式在java中十分难以阅读，更坏的结果是在模式中需要获取一个真实的反斜杠的时候，模式语言也转义一个反斜杠，因此，匹配a\b的模式用java字符串表示为“a\\\\b”。

Groovy的处理方式好得多，就像你先前看到的那样，通过斜杠 (/) 构成的字符串，这不需要转义反斜杠 (\) 字符，并且仍然像一个一般的Gstring一样工作，表3.4显示了怎样方便的声明一个模式。

#### Listing 3.4 Regular expression GStrings

```
assert "abc" == /abc/
assert "\\d" == /\d/

def reference = "hello"
assert reference == /$reference/

assert "\$" == /$/
```

这种语法不需要对\$进行转义，注意，可以选择使用这几种字符串风格的声明方式。

#### 符号

使用正则表达式的关键是明白模式的符号，为了方便，表3.8显示了常用的符号列表，把这一页放在特殊的地方以方便查找，很多时候将使用到它。

表3.8 正则表达式符号（摘要）

| Symbol  | Meaning  |
|---------|--|
| .       | Any character  |
| ^       | Start of line (or start of document, when in single-line mode)                   |
| \$      | End of line (or end of document, when in single-line mode)                       |
| \d      | Digit character  |
| \D      | Any character except digits  |
| \s      | Whitespace character   |
| \S      | Any character except whitespace  |
| \w      | Word character   |
| \W      | Any character except word characters   |
| \b      | Word boundary  |
| ()      | Grouping   |
| (x y)   | x or y, as in (Groovy Java Ruby)   |
| \1      | Backmatch to group one: for example, find doubled characters with (.)\1          |
| x*      | Zero or more occurrences of x  |
| x+      | One or more occurrences of x   |
| x?      | Zero or one occurrence of x  |
| x{m,n}  | At least m and at most n occurrences of x  |
| x{m}    | Exactly m occurrences of x   |
| [a-f]   | Character class containing the characters a, b, c, d, e, f                       |
| [^a]    | Character class containing any character except a                                |
| (?is:x) | Switches mode when evaluating x; i turns on ignoreCase, s means single-line mode |

更多的考虑：

- 使用分组属性，像星号和加号这样的扩展操作符进行的是最近匹配原则；  
ab+匹配的是abbbbb，使用(ab)+则匹配的是ababab。
- 在一般情况下，扩展操作符进行贪婪匹配，意思是说这将进行最长的子串匹配，在操作符后面增加一个?表示进行限制匹配，你也许想使用这个模式 href="(.\*)"从一个HTML锚元素中抽取href属性，但href="(.\*?)"也许更好，第一个版本直到在文本的最后一个双引号时才结束，第二个版本直到下一个双引号出现。

这里仅仅对正则表达式进行了简短的描述，在JDK中有完整的说明文档，这个文档在类

`java.util.regex.Pattern`的javadoc中，基于JDK 1.4.2的文档可以在<http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html>找到。

通过javadoc了解更多不同的匹配方式，如反向匹配等等。

在写入代码之前进行测试总是有用的，有一个在线应用程序进行正则表达式的交互式测试：例如，[http://www.nvcc.edu/home/drodgers/ceu/resources/test\\_regex.asp](http://www.nvcc.edu/home/drodgers/ceu/resources/test_regex.asp)，应该意识到不是所有的正则表达式模式语言都是相同的，同一个正则表达式在.NET和JAVA或者Groovy程序中得到的可能不是期望的结果。差异是存在的，如果读者从书上或者网上获取到的一个正则表达式，仍旧应该在代码中进行测试。

现在可以进行模式的声明了，还需要告诉groovy该如何应用，接下来将看看这些用法。

### 3.5.2 应用模式

把正则表达式应用到一个字符串上，groovy支持如下的任务：

- 模式是否完全匹配整个字符串。
- 在字符串中检查是否匹配上一个模式。
- 匹配成功的数目。
- 根据每个成功匹配的结果做一些工作。
- 替换成功匹配的结果。
- 使用匹配的结果作为分隔符将字符串的分割成多个字符串。

列表3.5显示了如何在工作使用groovy来设置模式：

### Listing 3.5 Regular expressions

```
twister = 'she sells sea shells at the sea shore of seychelles'

// twister must contain a substring of size 3
// that starts with s and ends with a
assert twister ==~ /s.a/    <-- ① Regex find operator  
as usable in if
finder = (twister ==~ /s.a/)
assert finder instanceof java.util.regex.Matcher  ② Find expression  
evaluates to a  
matcher object

// twister must contain only words delimited by single spaces
assert twister ==~- /(\w+ \w+)*/   <-- ③ Regex match  
operator
WORD = /\w+/
matches = (twister ==~- /($WORD $WORD)*/)
assert matches instanceof java.lang.Boolean  ④ Match expression  
evaluates to a Boolean

assert (twister ==~- /s.e/) == false      <-- ⑤ Match is full, not  
partial like find
wordsByX = twister.replaceAll(WORD, 'x')
assert wordsByX == 'x x x x x x x x x x x'

words = twister.split(/ /)    <-- ⑥ Split returns a  
list of words
assert words.size() == 10
assert words[0] == 'she'
```

(1) 和 (2) 有些有趣，虽然规则查找操作符将结果放在一个Matcher对象中，它也可以作为一个boolean条件使用，当在第六章“groovy真相”中将看到为什么可以这样使用。

注意：记住=~和==~的不同之处，==~是更严格的匹配，因为需要检查整个字符串，==~像操作符要长一些，所以要求的内容更多一些，呵呵！！

更多的信息请参考java.util.regex.Matcher对象的javadoc文档，该文档包括了所有的匹配方式和如果在每一个匹配上使用分组的讨论。

通用规则陷阱。

读者不需要完整的理解规则，作者已经做了这些了，我们学习了如下内容：

- 当事情很复杂的时候，注释是冗长的
- 使用斜杠/代替字符串语法，否则将迷失在反斜杠中（因为反斜杠需要转义）。
- 不要让模式看起来难以理解，像列表3.5的WORD那样使用子表达式来建立模式。
- 不要猜想结果会正确，要进行测试，为了测试规则正确，写一些断言或者单元测试代码。

### 3.5.3 模式实战

现在已经准备好了用正则表达式做想要做的事情了，String有一个名称为eachMatch的方法，这个方法使用一个规则和一个闭包作为参数，闭包用来定义每一个符合规则的结果需要做的工作。

顺便说一下：匹配不是一个简单的字符串，而是一个字符串列表，在位置0包含了整个匹配，如果模式包括分组，那么可以通过match[n]进行访问，n是分组号，分组的编号通过它们在圆括号中的顺序决定的。

匹配结果将传递给闭包，在列表3.6中，把匹配的结果附加到结果字符串上。

#### Listing 3.6 Working on each match of a pattern

```
myFairStringy = 'The rain in Spain stays mainly in the plain!'

// words that end with 'ain': \b\w*ain\b
BOUNDS = /\b/
rhyme = /$BOUNDS\w*ain$BOUNDS/
found = ''
myFairStringy.eachMatch(rhyme) { match ->           ← ① string.eachMatch
    found += match[0] + ' '
}
assert found == 'rain Spain plain '

found = ''
(myFairStringy =~ rhyme).each { match ->           ← ② matcher.each
    found += match + ' '
}
assert found == 'rain Spain plain '                  ← ③ string.replaceAll
                                                        (pattern_string, closure)

cloze = myFairStringy.replaceAll(rhyme){ it-'ain'+'_ ' } ←
assert cloze == 'The r__ in Sp__ stays mainly in the pl__!'
```

这里有两种不同的途径用来遍历匹配的结果：如（1）使用String.eachMatch(Pattern)，如（2）使用Matcher.each()，Matcher是应用规则到字符串和模式所得到的结果，（3）显示了通过闭包替换规则匹配的每一个结果，变量it引用匹配的子字符串，使用下划线替换“ain”。

为了完全理解Groovy是如何支持正则表达式的，需要查看java.util.regex.Matcher类，这个JDK类包装了如下知识：

- 什么位置和匹配的结果是什么
- 为每一个匹配进行分组

GDK按数组访问的方式增强了Matcher类，这是下面的例子（非常熟悉）匹配所有的非空白字符的结果：

```
matcher = 'a b c' =~ /\S/
assert matcher[0] == 'a'
assert matcher[1..2] == 'bc'
assert matcher.count == 3
```

在匹配结果中的分组是有趣的，如果模式定义分组时包含圆括号，matcher返回的不是为每一个匹配结果返回一个单字符串，而是返回一个数组，完整的匹配是索引为0并且接下来的是分组信息，考虑这个例子，每一个匹配的结果是通过冒号分割的字符串对，为了后面的处理，匹配结果将冒号左边的字符串和右边的字符串分隔在两个组：

```
matcher = 'a:1 b:2 c:3' =~ /(\S+):(\S+)/
assert matcher.hasGroup()
assert matcher[0] == ['a:1', 'a', '1']
```

换句话说，matcher[0]的返回结果依赖与模式是否进行了分组。

这也可以应用到matcher的each方法，每一个组都十分方便，在处理的闭包定义多个参数，参数按照分组列表进行定义：

```
('xy' =~ /(.)(.)/) .each { all, x, y ->
    assert all == 'xy'
    assert x == 'x'
    assert y == 'y'
}
```

仅仅匹配了一次，但是包括了两个组，每一个组一个字符。

注意：groovy内部保存了最近使用的matcher（每线程），可以通过静态方法  
Matcher.getLastMatcher来获取，也可以设置matcher的index属性，使它可以通过  
matcher.index=x来查看各自的匹配结果，这两者在特定的情况下都是有用的，参考  
Matcher的API来了解更详细的信息。

在java和groovy中Matcher和Pattern结合在一起工作是关键概念，读者已经看到了Matcher，在下一节我们将看到Pattern。

### 3.5.4 模式和性能

最后，来看看模式的性能和模式操作符 `~String`。

模式操作符转换一个字符串为 `java.util.regex.Pattern` 对象实例，对于给定的字符串，通过这个模式对象可以查询到 `matcher` 对象。

构建这个理论基础是：模式一个有限状态机，在模式对象被创建的时候编译这个有限状态机，结构更复杂的模式，需要更长的创建时间，通过对比，通过有限状态机进行匹配处理是非常快的。

模式操作符把模式创建的时间从模式匹配的时间中分离出来，通过重用有限状态机提升了性能，列表3.7显示了一个可怜人在这两种方式的性能比较，预编译模式至少快20%。

**Listing 3.7 Increase performance with pattern reuse.**

```
twister = 'she sells sea shells at the sea shore of seychelles'  
// some more complicated regex:  
// word that starts and ends with same letter  
regex = /\b(\w)\w*\1\b/  
  
start = System.currentTimeMillis()  
100000.times{  
    twister =~ regex      | Find operator with implicit  
}                                | pattern construction  
first = System.currentTimeMillis() - start  
  
start = System.currentTimeMillis()  
pattern = ~regex                  | ① Explicit pattern  
100000.times{  
    pattern.matcher(twister)       | construction  
}                                | Apply the pattern on a String  
second = System.currentTimeMillis() - start  
  
assert first > second * 1.20
```

为了找到开始字符和结束字符相同的单词，使用 `\1` 引用到后面的匹配，把单词的第一个字符放在一个分组中，这个分组的编号为1。

注意（1）的书写不是 `a =~ b`，而是 `a = ~b`，小心这个地方！！

敏感的读者也许发现了一个问题：如果写成 `a=~b`（中间没有任何空格）会发生什么事情？它是 `=~` 操作符，还是给 `a` 分配 `~b` 模式？对于人性化的阅读，这是容易引起歧义的，对于 `groovy` 转换器来说不是这样的，`groovy` 转换器进行贪婪匹配，并且将这个转换为 `=~` 操作符。

保留空格是一个好的编程风格，即使转换器能正确的识别，为了代码的可阅读性，应该这样做。

不要忘记性能同时是在次于可读性的——至少在开始是这样的，如果重用一个模式导致你的代码不完美，那么在修改代码之前应该评价各种性能，测量出不同版本的代码在不同情况下的性能，并且平衡可维护性和内存要求。

### 3.5.5 模式分类

列表3.8 完整的显示了模式的用法，Pattern对象是通过模式操作符返回的对象，这个对象实现了一个isCase（String）方法，该方法用来比较一个字符串是否完全匹配一个模式，这个方法是方便使用grep方法和switch语句的前提。

例子中将4个字符的单词进行分类，模式因此是由四个点构成的，这可不是省略号！！

**Listing 3.8 Patterns in grep() and switch()**

```
assert (~/.{4}).isCase('bear')

switch('bear'){
    case ~/.{4}: assert true; break
    default: assert false
}

beasts = ['bear', 'wolf', 'tiger', 'regex']

assert beasts.grep(~/.{4}) == ['bear', 'wolf']
```

在switch和grep的分类读取中，直接使用的是classifier.isCase(candidate)方法，这通常使用的比较少，但是当使用的时候，最佳的方式是从右向左读：条件是classifier的一个例子。

正则表达式是难易掌握的，但是在文本处理任务方面正则表达式是十分有用的，一旦掌握了正则表达式，你就难以想象在编程的生活中没有正则表达式的日子，groovy使正则表达式更容易获取和使用。

在这里结束了文本类型的讨论，但是计算机总是有像文本一样的数字，在所有程序语言中数字的使用都很简单，但这不意味着没有改善空间，来看看groovy在处理数字类型方面的改进吧。

## 3.6 使用数字

在3.1节已经介绍了在groovy中可用的数字类型和他们的声明方式。

已经知道了decimal数字，这种数字的缺省类型为java.math.BigDecimal，大多数错误的理解为浮点数，来看看这个类型的使用和GDK中为数字提供的附加功能。

### 3.6.1 造型和数字运算符

当使用一个数字运算符的时候明白到底发生了什么是十分重要的。

在groovy中，加、乘和减法运算的大多数规则与java一样，但是关于浮点数的行为有点不一样，BigInteger和BigDecimal也需要包含进来，规则是易懂的，先符合的规则总是被使用。

对于+、-、\*运算来说：

- 如果有一个数为Float或者Double，那么结果是Double（在java中，只有操作数都是Float的时候，结果也是Float）。
- 否则，如果一个操作数为BigDecimal，结果为BigDecimal。
- 否则，如果一个操作数为BigInteger，结果为BigInteger。
- 否则，如果一个操作数为Long，那么结果为Long。
- 否则，结果为一个Integer。

表3.9显示了一个查询表，类型用缩写的大写字母表示。

表3.9 数字转换

| + - *      | B  | S  | I  | C  | L  | BI | BD | F | D |
|------------|----|----|----|----|----|----|----|---|---|
| Byte       | I  | I  | I  | I  | L  | BI | BD | D | D |
| Short      | I  | I  | I  | I  | L  | BI | BD | D | D |
| Integer    | I  | I  | I  | I  | L  | BI | BD | D | D |
| Character  | I  | I  | I  | I  | L  | BI | BD | D | D |
| Long       | L  | L  | L  | L  | L  | BI | BD | D | D |
| BigInteger | BI | BI | BI | BI | BI | BI | BD | D | D |
| BigDecimal | BD | D | D |
| Float      | D  | D  | D  | D  | D  | D  | D  | D | D |
| Double     | D  | D  | D  | D  | D  | D  | D  | D | D |

另一方面行为：

- 像java，不像Ruby，当运算结果超出类型表示的范围的时候，不会进行类型提升，幂运算除外。
- 对于除法运算，如果任何一个数是Float（或者Double）类型，那么运算结果为Double类型；否则，结果为BigDecimal类型，精度与两个数的精度值较大的为准，采用四舍五入的方式，结果是规范化的——小数位后面没有无效的0。
- 整数除法（结果保留为整数）通过显示造型或者使用intdiv方法是可以完成的。
- 移位运算符仅仅运用于Integer和Long，它们不能转换为别的类型。
- 幂运算的结果会自动提升到能够容纳结果的范围和精度的类型，提升顺序为Integer, Long, Double。
- 比较运算符在比较之前提升到较大范围的类型。

没有例子，规则是难以理解的，表3.10用例子来说明这些行为。

Table 3.10 Numerical expression examples

| Expression                       | Result type                   | Comments   |
|----------------------------------|-------------------------------|--|
| <code>1f*2f</code>               | <code>Double</code>           | In Java, this would be <code>Float</code> .  |
| <code>(Byte) 1+(Byte) 2</code>   | <code>Integer</code>          | As in Java, integer arithmetic is always performed in at least 32 bits.  |
| <code>1*2L</code>                | <code>Long</code>             |  |
| <code>1/2</code>                 | <code>BigDecimal (0.5)</code> | In Java, the result would be the integer 0.  |
| <code>(int) (1/2)</code>         | <code>Integer (0)</code>      | This is normal coercion of <code>BigDecimal</code> to <code>Integer</code> .   |
| <code>1.intdiv(2)</code>         | <code>Integer (0)</code>      | This is the equivalent of the Java <code>1/2</code> .  |
| <code>Integer.MAX_VALUE+1</code> | <code>Integer</code>          | Non-power operators wrap without promoting the result type.  |
| <code>2**31</code>               | <code>Integer</code>          |  |
| <code>2**33</code>               | <code>Long</code>             | The power operator promotes where necessary.   |
| <code>2**3.5</code>              | <code>Double</code>           |  |
| <code>2G+1G</code>               | <code>BigInteger</code>       |  |
| <code>2.5G+1G</code>             | <code>BigDecimal</code>       |  |
| <code>1.5G==1.5F</code>          | <code>Boolean (true)</code>   | The <code>Float</code> is promoted to a <code>BigDecimal</code> before comparison.   |
| <code>1.1G==1.1F</code>          | <code>Boolean (false)</code>  | 1.1 can't be exactly represented as a <code>Float</code> (or indeed a <code>Double</code> ), so when it is promoted to <code>BigDecimal</code> , it isn't equal to the exact <code>BigDecimal</code> 1.1G but rather 1.100000023841858G. |

唯一的意外就是没有意外，在java中，像第四行的结果总是出乎意外的，例如，`(1/2)`结果总是0，因为进行除法运算的两个操作数都是整数，仅仅是执行了整数除法，在java中为了得到0.5的结果，需要将代码写成`(1f/2)`。

在增强用户自定义输入行为的groovy程序中，这种行为尤其重要，假如允许应用程序的超级用户指定一个规则来计算职员的奖金，并且规则是`businessDone*(1/3)`，在java语义中，这对可怜的职员来说将是糟糕的一年。

### 3.6.2 GDK 为数字提供的方法

在表3.4中显示了GDK定义的全部可用的方法，这些方法实现了数字的加、减、幂运算等等，所有的工作都不意外，另外，`abs`、`toInteger`和`round`方法也像你希望的那样工

作。

非常有趣，GDK也定义了times、upto、downto和step方法，这些方法接受一个闭包参数，列表3.9显示了这些方法的应用，times方法仅仅用于做重复的动作，upto方法是递增一个数字，downto是递减一个数字，step是按一个步进从一个数递增到另外一个数一般形式。

### Listing 3.9 GDK methods on numbers

```
def store = ''  
10.times{      ← Repetition  
    store += 'x'  
}  
assert store == 'xxxxxxxxxx'  
  
store = ''  
1.upto(5) { number ->   ← Walking up with  
    store += number           loop variable  
}  
assert store == '12345'  
  
store = ''  
2.downto(-2) { number ->   ← Walking  
    store += number + ' '     down  
}  
assert store == '2 1 0 -1 -2 '  
  
store = ''  
0.step(0.5, 0.1 ){ number ->   ← Walking with  
    store += number + ' '     step width  
}  
assert store == '0 0.1 0.2 0.3 0.4 '
```

刚从java转过来的时候，在一个数字上调用方法会感到不习惯，只要记住数字也是对象，并且能像普通对象那样看到它们就可以了。

已经了解到在groovy中，数字的处理是很自然的，并且能防止在使用浮点数的时候犯一般性的错误，在大多数情况下，不需要记住运算的所有细节，在需要的时候，这节也许是一个有用的参考手册。

从一个没有想到的角度开始讲一个对象的可用性，在数字这一节已经看到了，在4.1节讲通过相同的原理开始讨论。

## 3.7 概要

简单的数据类型是使用groovy编程的重要组成部分，毕竟，使用字符串和数字工作是

软件开发的根本所在。

提供一般工作的更多便利性是groovy的主要目标之一，所以，groovy提升每个简单的数据类型为一个类对象并且把操作符实现为一个方法调用，这样使得面向对象的优势无所不在。

开发人员更方便的使用已经得到改进的字符串声明方式，不管是通过复杂的Gstring声明或者使用斜杠(/)避免进行转义操作，如正则表达式的模式，Gstring是groovy的另一个中心支柱，简明的、富有表现性的代码，让读者更加清晰的洞察到运行时字符串的值，不需要为了格式化字符串和替换字符串中的值而辛苦的进行字符串的连接操作。

在groovy中，正则表达式也有很好的体现，这又一次证明了它和别的脚本语言一样好用，正则表达式的作用在脚本世界是非常大的，并且语言的行为被完全限制，groovy毫不费劲的组合了java类库到groovy的支持中，保留了java程序员使用正则表达式的惯例。

Groovy在数字处理方面更方便并且使数字的精度处理更直观，甚至对于一个非程序员来说也是这样，这在groovy被用在需要业务客户提供公式的大型系统的快速配置上特别有用——例如，定义共享的评估细节。

字符串、正则表达式和数字可以从JDK之外的GDK获取到大量的有用方法，现在pattern是清晰明了的，groovy在设计之初就考虑了这些问题，在不牺牲java性能的情况下专注于使重复的任务尽可能的简单。

很快就可以看到java开发人员使用内建支持的类型了，groovy的设计者意识到很少考虑程序员的想法是不行的，下一节讨论方便的容器类型：ranges、lists和maps。

# 第四章 集合类型

计算机重复做相同任务从来就不会厌烦，这是一件好事。这或许是让计算机作为我们生活一部分的最重要的特征。搜索数不清的文件或者网页，每10分钟下载一次邮件，实时绘制股票走势图——这仅仅是计算机用集合处理重复工作的一些例子。

由于集合在编程中十分突出，groovy直接提供对集合的使用以缓和使用它们的难度，包括范围（ranges）、列表（lists）和映射（maps）。为了与简单数据类型的一致性，groovy支持通过简单的方式来声明集合类型，为集合类型提供专门的操作符，并且通过GDK增加了许多额外的功能。

这并不是说集合类型的使用对于java开发人员来说是一个全新的知识，就像你看到的那样，在groovy中集合是容易理解和记忆的，你能快速的掌握它，这不像你在接触一个新概念时想象的那样难。

不考虑新的标记语法，lists和maps在语义上与java完全相同，稍微不同的是ranges，因为在java中没有等价的概念，现在开始集合的讨论吧。

## 4.1 使用 ranges

考虑一下经常写的循环代码：

```
for (int i=0; i<upperBound; i++) {  
    // do something with i  
}
```

我们大多数时候做了上千次这样的代码，它太平常了，以至于我们几乎不思考它，现在有机会改进它了，这些代码告诉你将做什么或者怎么样做？

仔细的看看变量、条件和增量，可以看到循环是从0开始的并且没有到达upperBound值的时候进行的，如果不考虑循环体中的i，我们必须进行怎么样的描述呢？

紧接着，考虑一下你经常写的条件代码：

```
if (x >= 0 && x <= upperBound) {  
    // do something with x
```

}

相同的事情发生了：为了懂得计算机在做什么，我们必须认真的检查代码。变量x必须在0和一个upperBound之间的的时候，代码块才被执行，这里包括了upperBound。

我们不是说使用这样的语法是错误的，不是说不能使用C风格的循环。作为有数年工作经验的程序员，很难做出这样的结论，重要的是，range能使用更少的表达式做相同的事情。

通过groovy的range能简单的展现代码的意思，一个范围（range）有一个左边界和一个右边界，开发者可以为range中的每一个元素做一些工作，实际上就是迭代range，你能确定一个条件元素是否落在一个range范围之内，换句话说，一个range是一个间隔加上如何移动这个间隔的策略。

正在介绍的是groovy的新概念ranges，groovy通过range扩展了你在代码中表述自己思想的方式。

接下来将说明怎样规定range，事实上range也是对象，怎样使用这些对象和range在GDK中的典型用法。

### 4.1.1 规定 ranges

范围用两个点“..”表示范围操作符，用来指定从左边到右边的边界，这个操作符具有较低的优先级，因此你常常需要使用圆括号把它们括起来，范围也可以通过相应的构造方法进行声明。

“..<”范围操作符号指定了一个半排除范围——也就是说，右边界值不是range的一部分：

```
left..right  
(left..right)  
(left..<right)
```

范围通常有一个较低的左边界和较高的右边界，当反过来的时候，我们就叫做它是一个反向范围，范围也可以是由已经描述过的类型所组成，列表4.1显示了range怎样使用除了整数之外的类型做边界，如日期和字符串，groovy在语言级别支持ranges和for-in-range循环。

### Listing 4.1 Range declarations

```
assert (0..10).contains(0)
assert (0..10).contains(5)
assert (0..10).contains(10)

assert (0..10).contains(-1) == false
assert (0..10).contains(11) == false

assert (0..<10>.contains(9)
assert (0..<10>.contains(10) == false

def a = 0..10
assert a instanceof Range
assert a.contains(5)

a = new IntRange(0,10)
assert a.contains(5)

assert (0.0..1.0).contains(0.5)

def today      = new Date()
def yesterday = today-1
assert (yesterday..today).size() == 2

assert ('a'..'c').contains('b')    ←③ String ranges

def log = ''
for (element in 5..9){
    log += element
}
assert log == '56789'

log = ''
for (element in 9..5){
    log += element
}
assert log == '98765'

log = ''
(9..<5>.each { element ->
    log += element
}
assert log == '9876'
```

Inclusive  
ranges

Half-exclusive  
ranges

① References  
to ranges

Explicit  
construction

② Date  
ranges

for-in-range  
loop

Loop with  
reverse range

④ Half-exclusive, reverse,  
each with closure

注意在（1）我们将一个range指派给一个变量，换句话说，该变量持有类型 groovy.lang.Range的一个对象实例的引用，我们将查看这个实例并且看看它到底意味着什么。

日期类型也可以在范围中使用，如（2），因为GDK增加了previous和next方法到 java.util.Date，这两个方法是对日期进行增加一天或者减少一天的方法。

顺便说一下：GDK也增加minus和plus方法给java.util.Date，这样可以对日期加数天或者减数天。

字符串的previous方法和next方法也是通过GDK增加的，这样字符串可以为range所

用，如（3），字符串中的最后一个字符被加或者被减，并且在溢出的时候被处理成在字符串的最后增加一个新的字符或者删除字符串的最后一个字符。

通过each方法遍历range，这个方法在每次遍历时候将当前值传递给闭包，如（4），如果范围是反向的，遍历的过程也是反向的，如果range是半排除的，遍历将在到达右边界之前结束。

## 4.1.2 range 是对象

由于每一个range都是一个对象，所以可以直接在每个range对象上进行方法调用，最突出的方法是each，这个方法为range中的每一个元素执行闭包，还有就是contains，这个方法用来检查一个值是否在range中。

作为一个顶级类，range也可以通过实现一个isCase方法进行操作符重写（参考3.3节），这个方法的意思与contains一样，也就意味着，可以像grep过滤器一样在switch语句中使用range，如列表4.2所显示的这样。

**Listing 4.2 Ranges are objects**

```
result = ''  
(5..9).each{ element ->  
    result += element  
}  
assert result == '56789'  
  
assert (0..10).isCase(5)  
  
age = 36  
switch(age) {  
    case 16..20 : insuranceRate = 0.05 ; break  
    case 21..50 : insuranceRate = 0.06 ; break  
    case 51..65 : insuranceRate = 0.07 ; break  
    default: throw new IllegalArgumentException()  
}  
assert insuranceRate == 0.06  
ages = [20,36,42,56]  
midage = 21..50  
assert ages.grep(midage) == [36,42]
```

**Iterating through ranges**

**① Ranges for classification**

**② Filtering with ranges**

（2）是range对象使用grep方法的好例子：midage范围作为一个参数传递给grep方法。

如（1）所示是我们在业务领域中经常看到的使用range进行分类的例子，不同范围的资产分配不同的利率，基于不同的费用收取不同的佣金和基于一个业务范围定义工资红利，尽管技术上可以使用一个函数，业务上使用一个范围。当在软件中包括业务模块的时候，通

过range进行分类是十分便利的。

### 4.1.3 range 实战

列表4.1使用了date类型和string类型的范围，事实上，range可以使用任何类型，只要这个类型满足以下两个条件：

- 该类型实现next和previous方法，也就是说，重写++和--操作符；
- 该类型实现java.lang.Comparable接口；也就是说实现compareTo方法，实际上重写<=>操作符。

在列表4.3的例子中我们用类Weekday表示一周中的一天，一个Weekday有一个值，这个值在‘sun’到‘sat’之间，本质上，它是0到6之间的一个索引，通过list的下标来对应每个weekday的名称。

通过这些准备工作，可以构建一个工作的range，用来报告我们一周所做的工作，我们的老板想访问我们的周报，在最后通过一个断言来表明结果。

### Listing 4.3 Custom ranges: weekdays

```
class Weekday implements Comparable {
    static final DAYS = [
        'Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'
    ]
    private int index = 0           | Construct by name
    Weekday(String day){          | and normalize index
        index = DAYS.indexOf(day)
    }
    Weekday next(){               |
        return new Weekday(DAYS[(index+1) % DAYS.size()])
    }
    Weekday previous(){          |
        return new Weekday(DAYS[index-1])      ← 1 Automatic
    }                                underflow
    int compareTo(Object other){
        return this.index <= other.index
    }
    String toString(){
        return DAYS[index]
    }
}

def mon = new Weekday('Mon')
def fri = new Weekday('Fri')

def worklog = ''                  | Use the range
for (day in mon..fri){           | for iteration
    worklog += day.toString() + ' '
}
assert worklog == 'Mon Tue Wed Thu Fri '
```

这个代码可以放在一个脚本文件中，即使在代码中包括了两个类的声明也是可以的，`Weekday`就像脚本中的一个内部类。

在(1)处`previous`方法的实现有点特殊，虽然`next`方法使用模运算符来从星期六(索引为6)跳到星期天(索引为0)，反方向只是简单的减少索引，索引号-1用来寻找上一个索引所对应的名称，并且`DAYS[-1]`引用到日期列表的最后一个元素，就像将在下一节看到的那样，我们构建了一个`new Weekday('Sat')`，并且构造方法正常的索引指向6。

相比java而言，`range`是一个灵活的解决方案，`for`循环和条件不是对象，不能被重用，并且不能传递参数，但是`range`可以，`range`让我们只关注业务代码该做什么，而不用关心如何做，使你专注于自己的业务，而不用关心边界条件。

在接下来将看看自定义`range`的使用。仔细检查代码，`range`无处不在，并且`range`使你的代码更值得关注，通过一些练习，读者也许会发现在哪里都可以使用`range`，通过新的语言概念改变观点是正常的。

很快读者将在浏览`list`的下标操作符的时候了解到`range`的新知识，`list`是groovy

内建的类型。

## 4.2 使用 list

在最近的一个java项目中，我们必须写一个方法来实现向一个java数组中动态增加元素。这仿佛没有什么价值，但是这是不灵活的java程序，（我们已经被太多的groovy程序宠坏），java数组不能改变长度，因此向java数组中增加元素不是太容易，其中一种方法是将数组转换为java.util.List，再增加元素，然后转回到java数组；第二种方法是构建一个长度为原来数组的长度+1的新数组，然后拷贝原来的值到新的数组中，并且将新数组的最后一个值设置为新的元素，这两种方法都需要写好几行代码。

但是java数组在语言的支持方面也有它们自身的优势。数组使用下标操作符进行工作，使用myarray[index]这样的形式，容易的通过下标接收数组的一个元素，或者通过myarray[index] = newElement的方式快速的存储一个元素在数组指定的位置。

接下来将说明groovy是如何将java.util.List的优势和数组的优势融合到一起的，为实现快速操作符的特性扩展，方法重载和像Boolean一样使用list，通过groovy的list，你将发现对java Collections API有了一种新的途径来提高效率。

### 4.2.1 声明 list

列表4.4显示了声明list的不同方式，主要的方式是使用方括号括起的条目列表购成，条目之间通过逗号进行分隔：

```
[item, item, item]
```

条目列表可以是空，用来声明一个空的list，list的缺省类型是java.util.ArrayList，list也可以通过相应的构造方法进行显式声明，list的结果仍然可以通过下标操作符进行使用，实际上，这适合于所有的list类型如java.util.LinkedList。

list可以通过range的toList方法进行同时创建和初始化。

#### Listing 4.4 Specifying lists

```
myList = [1,2,3]
assert myList.size() == 3
assert myList[0] == 1
assert myList instanceof ArrayList
emptyList = []
assert emptyList.size() == 0

longList = (0..1000).toList()
assert longList[555] == 555

explicitList = new ArrayList()
explicitList.addAll(myList)
assert explicitList.size() == 3
explicitList[0] = 10
assert explicitList[0] == 10

explicitList = new LinkedList(myList)
assert explicitList.size() == 3
explicitList[0] = 10
assert explicitList[0] == 10
```



在(1)处我们使用`java.util.List`的`addAll(Collection)`方法，这个方法非常容易填充list，作为选择，可以将一个collection作为参数传递给构造方法来填充list对象，如我们看到的`LinkedList`。

基于完整性的考虑，在groovy增加了以`java数组`为参数的`list`的构造方法，一个数组像自动装箱一样——一个list将根据数组的元素（专有类型自动进行转换为包装类型）自动的产生。

GDK扩展了所有的数组、容器对象和字符串，为它们增加了一个`toList`方法，这个方法返回一个由数组（容器或者字符串）元素构成新的list对象，字符串对象被处理为一个`character`列表。

### 4.2.2 使用 list 操作符

`List`实现了在3.3节列出的一些操作符，列表4.4是其中的两个：`getAt`和`putAt`方法，这两个方法用来实现下标操作符，这样可以简单的通过索引来访问list，这样比通过其他方式访问list更加便利。

#### 下标操作符

GDK使用`range`和`collection`作为参数重载了`getAt`方法，这样可以通过一个范围或

者索引的一个集合来访问，在列表4.5进行了说明。

同样的策略应用到了putAt方法上，这个方法使用一个Range作为参数的方式进行了重载，这样可以将一个列表值指定给整个子列表。

#### Listing 4.5 Accessing parts of a list with the overloaded subscript operator

```
myList = ['a', 'b', 'c', 'd', 'e', 'f']

assert myList[0..2] == ['a', 'b', 'c']    ↪ getAt(Range)
assert myList[0, 2, 4] == ['a', 'c', 'e']  ↪ getAt(collection of indexes)

myList[0..2] = ['x', 'y', 'z']    ↪ putAt(Range)
assert myList == ['x', 'y', 'z', 'd', 'e', 'f']  ↪ ① Removing elements

myList[3..5] = []
assert myList == ['x', 'y', 'z']  ↪ ② Adding elements

myList[1..1] = ['y', '1', '2']
assert myList == ['x', 'y', '1', '2', 'z']
```

通过范围指定下标的数量不需要与range的数量一样，当指定的列表值小于给定的范围或者是空的时候，这个列表被收缩，如（1）显示的那样，当指定的列表值更大时，列表进行增长，如（2）所示。

范围作为下标使用来访问list中的元素是十分便利的特性，也可以参考javadoc  
java.util.List#sublist。

除了正数作为下标索引之外，list也可以通过负数进行索引，通过负数索引是从列表的最后向前走的。图4.1显示了list[0,1,2,3,4]进行正数索引和负数索引的对照。

因此，通过list[-1]来获取非空列表的最后一个元素，通过list[-2]来获取到倒数第二个元素，负数索引也可以用作表示范围，因此list[-3...-1]这样的表示结果将得到列表的最后三个元素。当使用反向范围的时候，得到的结果列表也是反向的，因此list[4..0]的结果是[4,3,2,1,0]。在这里，结果作为一个新的list对象返回，而不是像JDK的sublist那样返回原来的对象，甚至混合使用正数和负数作为索引都是可以的，如list[1...-2]能够用来去掉原来列表的开始的和最后的元素。

| <b>Example list values</b> | 0                | 1  | 2  | 3  | 4  |    |                      |                       |
|----------------------------|------------------|----|----|----|----|----|----------------------|-----------------------|
| <b>Positive index</b>      | 0                | 1  | 2  | 3  | 4  | 5  | 6                    |                       |
|                            | -7               | -6 | -5 | -4 | -3 | -2 | -1                   | <b>Negative index</b> |
| <b>Out of bounds</b>       | <b>In bounds</b> |    |    |    |    |    | <b>Out of bounds</b> |                       |
|                            |                  |    |    |    |    |    |                      |                       |

图4.1 一个长度为5的列表的正数和负数索引，包括在界内和在界外的分类索引

**提示:** `list`下标操作符中的`range`是`IntRange`的实例，排他性的`IntRange`在构建的时候被映射，这发生在下标操作符起作用和能映射负数索引和到正数索引之前，这在混合正的左边界和负的右边界(不包括)会使人十分惊奇的，例如，`IntRange(0..<2)`得到的结果是`(0..-1)`，因此`list[0..<2]`实际上等同于`list[0..-1]`。

虽然这样没有问题并且是可以预见的工作，但对于代码的阅读者也许会感到迷惑，读者也许预料的结果为`list[0..-3]`，基于这个原因，这种方式应该避免使用。

### 增加和删除列表中的条目

虽然下标操作符能用来改变一个列表中的任意单个元素，也有一些操作符用来改变`list`的内容，它们是

`plus(Object)`, `plus(Collection)`, `leftShift(Object)`, `minus(Collection)`

和`multiply`，列表4.6显示了它们的用法，`plus`方法被重载，这样可以区别是增加一个元素还是增加一个集合的所有元素到列表中，`minus`方法仅仅接受一个集合作为参数。

#### Listing 4.6 List operators involved in adding and removing items

```
myList = []  
myList += 'a'    ↪ plus(Object)  
assert myList == ['a']  
  
myList += ['b', 'c']    ↪ plus(Collection)  
assert myList == ['a', 'b', 'c']  
  
myList = []  
myList << 'a' << 'b'    ↪ leftShift is like  
                           append  
assert myList == ['a', 'b']  
  
assert myList - ['b'] == ['a']    ↪ minus(Collection)  
assert myList * 2 == ['a', 'b', 'a', 'b']    ↪ Multiply
```

当谈论操作符的时候，并不只是已经在列表上使用过的“==”操作符，幸好这正是我们期望的，来看看它是如何工作的：列表的equals方法用来测试两个集合是否有相同的元素，参考javadoc文档java.util.List#equals了解更详细的信息。

#### 控制结构

Groovy的列表有许多灵活的存储方式，它们在组织执行groovy程序流程的时候扮演了重要的角色，列表4.7显示了groovy中的列表的if, switch和for控制结构。

#### Listing 4.7 Lists taking part in control structures

```
myList = ['a', 'b', 'c']  
  
assert myList.isCase('a')  
candidate = 'a'  
switch(candidate){  
    case myList : assert true; break  
    default      : assert false  
}  
  
assert ['x', 'a', 'z'].grep(myList) == ['a']  
  
myList = []  
if (myList) assert false    ↪ ③ Empty lists  
                           are false  
  
// Lists can be iterated with a 'for' loop  
log = ''  
for (i in [1, 'x', 5]) {    ↪ ④ for in Collection  
    log += i  
}  
assert log == '1x5'
```

在(1)和(2)，你已经在正则表达式和range相关章节看到了这种用法：实现isCase方法，得到一个grep过滤器并且可以简单的使用switch进行分类。

(3) 是一个小小的惊喜，内部的boolean值测试，空的list将评估为false。

(4) 显示了在列表或者其他集合上的进行的循环，并且也说明了列表可以包含多个不同类型的元素。

### 4.2.3 使用列表方法

在列表中还有许多有用的方法没有通过例子来说明它们，事实上大量的方法来自java接口java.util.List（在JDK1.4中共有25个方法）。

而且，GDK增加了额外的方法到List接口、Collection接口和Object类上，因此，在List上有许多方法是可以使用的，包括所有在Collection和Object上的方法。

附录C有一个GDK增加到List上的方法的完整列表，java.util.List的javaDoc描述了完整的JDK方法。

当在groovy中使用list的时候，不需要知道方法是来自JDK还是GDK，或者是这个方法定义在List接口还是Collection接口上，不管怎样，作为描述groovy的list类型的目的，我们完全覆盖了在list和collection上的GDK方法，但是不包括重载的方法和在前面的例子中已经讨论过的方法，在这里仅仅提供了我们认为比较重要的JDK部分方法的例子。

#### 维护列表内容

列表4.8展现了第一组方法，涉及到了向列表中增加元素、从列表中删除元素；通过不同的途径组合列表，排序，翻转列表和平整（flattening）嵌套的列表，从存在的列表创建新的列表。

#### Listing 4.8 Methods to manipulate list content

```
assert [1,[2,3]].flatten() == [1,2,3]

assert [1,2,3].intersect([4,3,1]) == [3,1]
assert [1,2,3].disjoint([4,5,6])

list = [1,2,3]          1 Treating a list  
popped = list.pop()    like a stack  
assert popped == 3  
assert list == [1,2]

assert [1,2].reverse() == [2,1]

assert [3,1,2].sort() == [1,2,3]

def list = [[1,0], [0,1,2]]  
list = list.sort { a,b -> a[0] <=> b[0] }          2 Comparing lists  
assert list == [[0,1,2], [1,0]]                         by first element

list = list.sort { item -> item.size() }           3 Comparing  
assert list == [[1,0], [0,1,2]]                      lists by size

list = ['a','b','c']          4 Removing  
list.remove(2)                    by index  
assert list == ['a','b']  
list.remove('b')                  5 Removing  
assert list == ['a']              by value

list = ['a','b','b','c']  
list.removeAll(['b','c'])  
assert list == ['a']

def doubled = [1,2,3].collect{ item ->      6 Transforming one  
    item*2  
}  
assert doubled == [2,4,6]

def odd = [1,2,3].findAll{ item ->      7 Finding every element  
    item % 2 == 1  
}  
assert odd == [1,3]
```

列表中的元素可以是任何类型，包括别的嵌套的list对象，这可以用来实现列表的列表，在groovy中这与java中的多维数组相等价，对于嵌套的list来说，通过flatten方法来获取到一个包括所有元素的列表。

对于同时出现在两个列表中的元素的交集，集合也可以通过方法disjoint方法来获取到交集，不管它们的交集是否为空。

List可以像堆栈（stack）那样使用，常用的堆栈行为是push和pop，如（1）所示，push操作使用list的“<<”操作符替代。

当list中的元素是可比较的（实现Comparable接口），那么可以对list直接进行排

序，也可以通过一个闭包指定一个排序逻辑，如（2）和（3）所示，在（2）中，通过子列表的第一个元素进行排序，在（3）中接受单个参数，这样能在闭包内部完成排序，在这个例子中，排序根据闭包的返回结果进行（返回的结果必须是可比较的（Comparable））。

可以通过索引删除元素，如（4）所示，或者通过值删除元素，如（5）所示，也可以删除这个列表中所有的在另外一个list中出现的元素。在JDK中包括所有的删除方法。

collect方法，如（6）所示，返回一个新的list，这个list中的每个元素是原list相同位置的元素应用闭包的结果。在例子中，在接受的新list中，每一个元素的值都为原list中的相应元素\*2。findAll方法，如（7）所示，这个方法返回的列表中的所有条目是原list根据闭包计算结果为true的元素，在这个例子中，我们使用模运算符来查找所有的奇数。

改变存在的列表的两个相关的问题是删除重复值和删除null值，删除重复值的一种方法是将list转换为一个set，使用list作为参数通过调用set的构造方法来完成。

```
def x = [1,1,1]
assert [1] == new HashSet(x).toList()
assert [1] == x.unique()
```

如果不想创建新的集合但还是想完成相同的工作，可以使用unique方法，这保证实体的顺序没有被这个操作改变。

从list中删除null值能保证列表中没有空元素——例如，在前面已经看到的findAll方法：

```
def x = [1,null,1]
assert [1,1] == x.findAll{it != null}
assert [1,1] == x.grep{it}
```

可以看到通过grep代码更简短，但是为了更好的理解这种机制，读者需要闭包的相关知识（第五章）和第六章相关知识。

## 访问列表的内容

List有许多方法用来访问list中元素，遍历列表和接收结果。

查询方法包括查询list中元素数量的count方法，min和max方法用来查询list中的最小元素和最大元素，find方法用来查找list第一个符合闭包要求的元素，every和any

方法用来确定list中的每一个元素（或者任何一个元素）是否符合闭包的要求。

遍历像通常一样完美，使用each方法进行正向遍历，并且使用eachReverse方法进行反向遍历（译者注：在groovy1.6中并没有发现eachReverse方法，但存在reverseEach方法）。

这些方法是简单优雅的，join方法十分简单：这个方法将所有元素通过给定的字符串进行连接，然后将结果作为字符串返回，inject方法来自Smalltalk，这个方法使用闭包注入一个新的函数，这个函数用来对一个中间结果和遍历的当前元素进行操作，inject方法的第一个参数是中间结果的初始值，在列表4.9中，使用这个方法累加所有元素，后面对所有的元素结果进行相乘。

#### Listing 4.9 List query, iteration, and accumulation

```
def list = [1,2,3]

assert list.count(2) == 1
assert list.max() == 3
assert list.min() == 1

def even = list.find { item ->
    item % 2 == 0
}
assert even == 2

assert list.every { item -> item < 5}
assert list.any { item -> item < 2}

def store = ''
list.each { item ->
    store += item
}
assert store == '123'

store = ''
list.reverseEach{ item ->
    store += item
}
assert store == '321'

assert list.join('-') == '1-2-3'

result = list.inject(0){ clinks, guests ->
    clinks += guests
}
assert result == 0 + 1+2+3
assert list.sum() == 6

factorial = list.inject(1){ fac, item ->
    fac *= item
}
assert factorial == 1 * 1*2*3
```

Querying

Iteration

Accumulation

如果不熟悉这个概念，那么理解和使用inject方法就稍微有点困难，注意，这不完全是遍历的例子，还用来存储中间计算结果，好处是在累加结果的外面不需要引入额外的变量，并且没有超出闭包的范围。

GDK为list提供了两个更加便利的方法：asImmutable和asSynchronized，这两个方法使用Collections.unmodifiableList和Collections.synchronizedList方法来保护list被无意中的修改和并发访问，更详细的信息参考javadoc的相关主题。

#### 4.2.4 list 实战

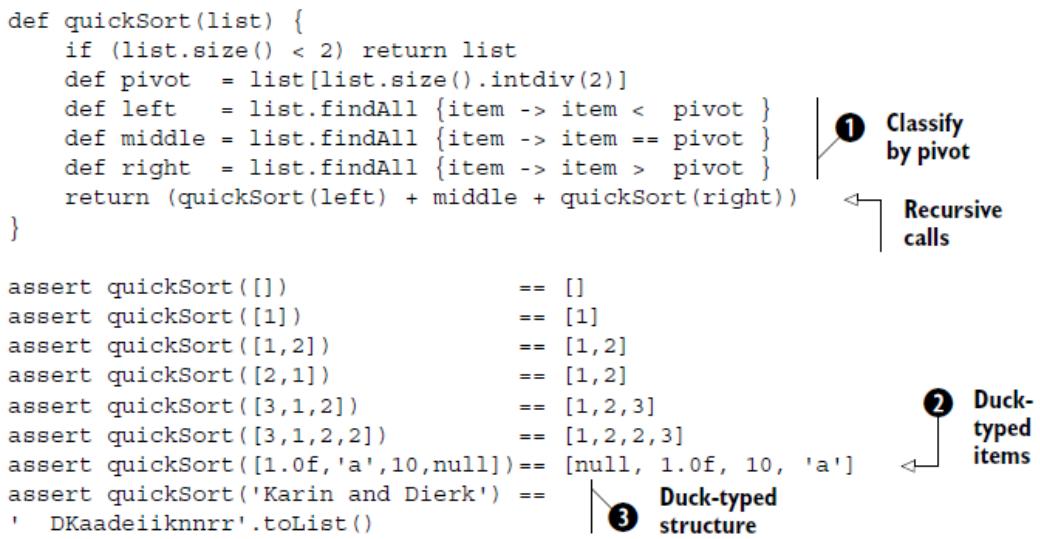
在所有的虚假的例子之后，应该看一个真实的应用：在列表4.10中实现Tony Hoare排序算法，为了使这件事情更有趣，先通过一般的方法来实现，为了排序不需要任何特定的数据类型，这样通过使用<、=和>操作符来完成比较工作。

快速排序的目标是使用较少的比较，这个策略依赖于在list中找到一个好的核心元素用来把list拆分为两个list：一个list中的所有元素都比核心元素小，另外一个list中的所有元素都比核心元素大，然后再递归调用这两个list，这后面的理论基础是不需要把一个列表的元素和另一个列表进行比较，如果总能找到正确的核心元素，这将精确的对半拆分list，这个算法运行一个复杂的 $n * \log(n)$ ，最坏的结果，每次都选择了一个边界元素作为核心元素，那么排序将以 $n^2$ 结束，在列表4.10中，选择了list的中间元素，这通常是一个好的选择。

### Listing 4.10 Quicksort with lists

```
def quickSort(list) {
    if (list.size() < 2) return list
    def pivot = list[list.size().intdiv(2)]
    def left = list.findAll {item -> item < pivot}
    def middle = list.findAll {item -> item == pivot}
    def right = list.findAll {item -> item > pivot}
    return (quickSort(left) + middle + quickSort(right))
}

assert quickSort([]) == []
assert quickSort([1]) == [1]
assert quickSort([1, 2]) == [1, 2]
assert quickSort([2, 1]) == [1, 2]
assert quickSort([3, 1, 2]) == [1, 2, 3]
assert quickSort([3, 1, 2, 2]) == [1, 2, 2, 3]
assert quickSort([1.0f, 'a', 10, null]) == [null, 1.0f, 10, 'a']
assert quickSort('Karin and Dierk') == 'DKaadei knnrr'.toList()
```



对比前面说过的，实际上在（1）处使用了三个列表而不是两个列表，当不想丢失重复出现的元素的时候使用这种实现方式。

可以像（2）那样排序不同的类型，甚至像（3）那样对字符串进行排序，原因是没有为list中的条目指定任何的类型，由于类型实现了size/getAt(Index)/findAll方法，在这里直接作为一个sortable对象对待。

顺便说一下：groovy的sort方法使用JAVA的排序实现，这考虑到了最坏情况下的性能问题，考虑了 $n \log(n)$ 的发生情况。

当然这个的实现也能进行多方面的优化，在这里的目标是整洁和灵活，而不是最好的性能。

如果没有groovy的帮助来解释快速排序算法，我们的伪代码看起来跟列表4.10一样，换句话说，groovy代码本身就是最好的用来描述做了什么工作的描述信息，想象一下，当代码读起来就像文档描述一样，你的代码是多么的重要。

已经讨论了groovy最强大功能之一的list，它们总是在手边；list的使用是一直的，并且也支持操作符，起初，也许会被list过多的可用方法吓住，但这也是list强大的原因。

现在可以使用list来做好自己的编码工作了。

接下来的将介绍与list相似的map，groovy扩展了java集合框架，提供了便利的快捷操作方式。

## 4.3 使用 map

假设你正打算学习一门新语言的词汇，并且你需要高效的进行查找，关注常用的一些词语想必是有益的。因此，你需要一个在文本中出现的单词的集合，并且分析这些单词在文本中出现的频率。

Groovy能为你做什么？我们暂且假定是一个大的字符串，必须多方面拆分字符串为单词，但是该怎样计数并且保存单词的使用频率呢？不可能为碰到的每一个不同的单词设置不同的变量，存储频率在一个list中是可能的，但是不方便，通过map可以轻松的解决这个问题。

像下面的伪代码可以解决这个问题：

```
for each word {  
    if (frequency of word is not known)  
        frequency[word] = 0  
        frequency[word] += 1  
}
```

这看起来像list语法，但是使用字符串作为下标，而不是整数作为下标，事实上，groovy的map的使用就像list一样，map允许任何类型的对象作为下标。

为了描述map类型，我们描述了map是如何规定的，map可以使用的操作方法，令人惊奇的便利的特性。

### 4.3.1 声明 map

Map的规范与前面介绍的list的规范类似，很像list，map保证可以通过下标来获取值或者设置值，不同的是map可以使用任何类型作为下标操作符的参数，而list仅仅支持整数，list知道list中元素的顺序，而map一般不知道，特殊的map如java.util.TreeMap也可知道它们的key的顺序。

简单的map声明方式是使用方括号括起来相应的条目列表，这些条目列表通过逗号进行分隔，map的关键特性是map中的条目都是键值对(key-value)，通过冒号分隔键与值：

[key:value, key:value, key:value]

理论上，任何类型都可以用作map的键（key）或者值（value），当使用特殊的类型作为key的时候，需要遵循java.util.Map在javadoc中的规定。

通过“[:]”来声明一个空的map，map缺省的类型是java.util.HashMap，也可以通过调用构造方法进行显式声明，这样的map仍然可以使用下标操作符进行操作，事实上，这适合于任何类型的map，就像在列表4.1.1使用的java.util.TreeMap一样。

#### Listing 4.11 Specifying maps

```
def myMap = [a:1, b:2, c:3]

assert myMap instanceof HashMap
assert myMap.size() == 3
assert myMap['a'] == 1

def emptyMap = [:]
assert emptyMap.size() == 0

def explicitMap = new TreeMap()
explicitMap.putAll(myMap)
assert explicitMap['a'] == 1
```

在列表4.11中，使用了来自java.util.Map的putAll(Map)方法来填充例子map，另外也可以通过TreeMap的构造方法传递myMap。

一般情况下key的类型都是字符串，在声明map的时候可以忽略字符串标记（单引号或者双引号）：

```
assert ['a':1] == [a:1]
```

如果key没有包括特殊的字符（需要符合有效标识符规则）并且不是groovy的关键字，那么允许通过这么便利的方式进行声明。

这种方式也有不便的地方，例如，本地变量的内容被用作key。假设有一个本地变量x，它的内容为'a'，由于[x:1]等价于['x':1]，那么该怎样保证它等于['a',1]呢？通过把符号放在圆括号中，强制让groovy将符号看做为一个表达式：

```
def x = 'a'
assert ['x':1] == [x:1]
assert ['a':1] == [(x):1]
```

需要进行这样的操作比较罕见，但是当需要从本地符号（本地变量、字段、属性）得到

一个key的时候，忘记插入圆括号可能会导致错误。

### 4.3.2 使用 map 操作符

使用map最简单的操作是使用key存储对象到map中和通过key从map中获取对象。列表4.12演示了如何进行这样的工作，从map中获取对象的一种可选方式是使用下标操作符，也许读者已经猜测到了，这基于map实现了getAt方法；另外一种可选方式是使用点语法像使用属性那样来获取对象，在第七章将学习到属性的相关知识；第三种选择是使用get方法，这个方法允许传递一个缺省值，在map没有相应的key的时候允许返回这个值。如果没有指定缺省值，null将为缺省，如果get(key, default)被调用时，key没有找到并且缺省值被返回，那么key:default对将被增加到map中。

**Listing 4.12 Accessing maps (GDK map methods)**

```
def myMap = [a:1, b:2, c:3]

assert myMap['a']      == 1
assert myMap.a        == 1
assert myMap.get('a')  == 1
assert myMap.get('a',0) == 1 | Retrieve existing elements

assert myMap['d']      == null
assert myMap.d        == null
assert myMap.get('d')  == null | Attempt to retrieve missing elements

assert myMap.get('d',0) == 0
assert myMap.d        == 0 | Supply a default value

myMap['d'] = 1
assert myMap.d == 1
myMap.d = 2
assert myMap.d == 2 | Simple assignments in the map
```

将值分配给map时可以使用下标操作符或者点语法，在使用点语法的时候，如果key包含了特殊字符，需要使用字符串符号（单引号或者双引号）围起来，如下所示：

```
myMap = ['a.b':1]
assert myMap.'a.b' == 1
```

如果仅仅写成myMap.a.b，在这里是不能正常工作的，这其实等价于  
myMap.getA().getB()。

列表4.13显示了来自map的相关使用方法，大量使用到来自JDK的java.util.Map的

方法，在列表4.13中显示的equals/size/containsKey/containsValue的使用是简单易懂的。keySet方法返回一个key的set集合，set是一个像list，但不包括重复的元素，并且集合中的元素没有固定的顺序，更详细的信息请参考java.util.Set的JavaDoc，为了比较keySet是否与给定的list中的key，在这里将这个list转换为一个set，这是通过方法toSet来实现的。

values方法返回map的值（value）的列表，由于map没有办法知道key的顺序，因此不能了解value列表的顺序，为了与预知的list中的值进行比较，在这里将两个list都转换为set。

通过调用entrySet方法可以将map转换为一个集合，这个方法返回一个实体的集合，在集合中的每一个实体都可以访问到key属性和value属性。

#### Listing 4.13 Query methods on maps

```
def myMap = [a:1, b:2, c:3]
def other = [b:2, c:3, a:1]

assert myMap == other    ← Call to equals

assert myMap.isEmpty()  == false
assert myMap.size()     == 3
assert myMap.containsKey('a')
assert myMap.containsValue(1)
assert myMap.keySet()   == toSet(['a', 'b', 'c'])
assert toSet(myMap.values()) == toSet([1, 2, 3])
assert myMap.entrySet() instanceof Collection

assert myMap.any {entry -> entry.value > 2 }
assert myMap.every {entry -> entry.key < 'd'}  ↗ ① Methods added by GDK

def toSet(list){
    new java.util.HashSet(list)
}  ↗ Utility method used for assertions
```

Normal JDK methods

① Methods added by GDK

GDK增加了两个方法到JDK的map类型：any和every，如①所示，它们的作用与list中的同名方法类似：它们都返回一个Boolean值用来表示map中的任何一个（any）实体或者每一个（every）实体都满足闭包要求。

能够通过几种途径来遍历map：遍历实体，或者分别遍历key集合和value集合，由于通过keySet和entrySet返回的set是一个集合，因此能够使用for-in-collection形式的循环，列表4.14列出了一些例子。

#### Listing 4.14 Iterating over maps (GDK)

```
def myMap = [a:1, b:2, c:3]

def store = ''
myMap.each {entry ->
    store += entry.key
    store += entry.value
}
assert store.contains('a1')
assert store.contains('b2')
assert store.contains('c3')

store = ''
myMap.each {key, value ->
    store += key
    store += value
}
assert store.contains('a1')
assert store.contains('b2')
assert store.contains('c3')

store = ''
for (key in myMap.keySet()) {
    store += key
}
assert store.contains('a')
assert store.contains('b')
assert store.contains('c')

store = ''
for (value in myMap.values()) {
    store += value
}
assert store.contains('1')
assert store.contains('2')
assert store.contains('3')
```

Iterate over entries

Iterate over keys/values

Iterate over just the keys

Iterate over just the values

Map的each方法接受两种形式的闭包：传递一个参数给闭包，那么这个参数就是map的一个entry；传递两个参数给闭包，那么参数就是key和value，一般来说，后者在实际工作中更方便。

注意：列表4.14在store字符串上使用了三个断言，而不是一个断言，这是因为map中实体的顺序是不可预估的。

最后，可以通过几种不同的途径来进行map内容的修改，如列表4.15所示，可以通过原始的JDK的方法来移除元素，GDK引入的新的途径如下：

通过给定一个key的集合来创建一个子map（subMap），这个子map的所有实体的key

都来自这个集合；

`findAll`用来查找满足闭包要求的所有map实体；

`find`用来查找任意一个满足闭包要求的map实体，这里不像list那样查找的是第一个满足闭包要求的实体，这是因为map是没有顺序的；

`collect`为map的每一个实体应用闭包，返回每一个闭包应用的结果组成的list（闭包是否返回结果是可选的）

#### Listing 4.15 Changing map content and building new objects from it

```
def myMap = [a:1, b:2, c:3]
myMap.clear()
assert myMap.isEmpty()

myMap = [a:1, b:2, c:3]
myMap.remove('a')
assert myMap.size() == 2

myMap = [a:1, b:2, c:3]
def abMap = myMap.subMap(['a','b'])    ↪ ① Create a view onto
                                         the original map
assert abMap.size() == 2

abMap = myMap.findAll { entry -> entry.value < 3}
assert abMap.size() == 2
assert abMap.a      == 1

def found = myMap.find { entry -> entry.value < 2}
assert found.key   == 'a'
assert found.value == 1

def doubled = myMap.collect { entry -> entry.value *= 2}
assert doubled instanceof List
assert doubled.every {item -> item %2 == 0}

def addTo = []
myMap.collect(addTo)    { entry -> entry.value *= 2}
assert doubled instanceof List
assert addTo.every {item -> item %2 == 0}
```

最初的两个例子（`clear`和`remove`）是JDK的方法，其余的方法全部是GDK的方法，只有`subMap`方法（如（1）所示）是才出现的方法，`collect/find/findAll`实际上在list中已经存在，只是现在操作的是map的实体，而不是list的元素，`subMap`方法的功能与`subList`的方法类似，它通过一个key的集合来过滤原始的map来获取到一个新的map。

为了断定`collect`方法是期望的那样工作，重新回顾一下list中关于`collect`的用法：使用`every`方法确保每一个实体都是一致的，`collect`的第二个版本是附加一个集合

参数，这个版本的方法将闭包的结果加到这个集合中，避免创建临时的list。

也许读者已经看到了别的数据类型的可用方法列表，你也许没有看到为grep和switch使用的isCase方法，我们需要对map进行分类吗？如果有，可以对map的keySet或者values分别独立的进行分类。

GDK还为map类型增加了两个方法：asImmutable和asSynchronized，这两个方法使用Collections.unmodifiableMap和Collection.synchronizedMap来防止对map内容的修改和并发访问保护，更详细的信息请参考javadoc相关的主题。

### 4.3.3 map 实战

在列表4.16中，我们回顾了最初的用于统计单词出现频率的例子，这个策略使用一个map，每一个独立的单词作为map的key，map的value是对应单词在文本中出现的频率，我们检查整个文本中的每一个单词，增加map中相应单词的频率值，同时需要保证当文本中的单词第一次出现时，该单词的频率值要正确的增加，幸运的是get(key,default)可以来完成这个工作。

处理完成所有的单词之后，将这些放在一个list中，然后根据其相应的出现频率进行排序，最后打印出这些统计信息。

### Listing 4.16 Counting word frequency with maps

```
def textCorpus =  
"""  
Look for the bare necessities  
The simple bare necessities  
Forget about your worries and your strife  
I mean the bare necessities  
Old Mother Nature's recipes  
That bring the bare necessities of life  
"""  
  
def words = textCorpus.tokenize()  
def wordFrequency = [:]  
words.each { word ->  
    wordFrequency[word] = wordFrequency.get(word, 0) + 1    ←❶  
}  
def wordList = wordFrequency.keySet().toList()  
wordList.sort { wordFrequency[it] }    ←❷  
  
def statistic = "\n"  
wordList[-1..-6].each { word ->  
    statistic += word.padLeft(12) + ':'  
    statistic += wordFrequency[word] + "\n"  
}  
assert statistic ==  
"""  
    bare: 4  
necessities: 4  
    the: 3  
    your: 2  
    for: 1  
    recipes: 1  
"""
```

(1) 很好的展示了我们在groovy的数据类型中的相关知识，统计单词出现频率实际上是一个铺垫，它甚至比我们在这件开始使用的伪代码还要简短。

(2) 在wordList上的sort方法接受一个闭包是十分有益的，因为这能够在wordFrequency这个map上实现比较逻辑——这是与wordList完全不同的对象，这仅仅是一个练习，在java中也试试，统计行数，并且判断解决方法的可表达性。

List和map是非常强大的，整个语言都建立在这两个数据类型上（如Perl是list和hash）并且实现所有别的类型及对象。

它们强大的功能来自java集合框架，感谢groovy，现在我们可以方便的使用这些功能了。

到目前为止，我们没有转回到groovy和java集合类型，在下节将了解更多它们之间相互作用的地方。

## 4.4 groovy 集合中需要注意的地方

Groovy的list和map的基础是java集合API，事实上，groovy不但使用相同的抽象逻辑，甚至它们使用与java集合框架相同的类。

这对于已经很好的理解了java的程序员来说特别方便，如果你没有理解，你应该对后台的信息更感兴趣，从`java.util.Collection`的javadoc开始看看。

也应该了解JDK中关于java集合的指南和教程，它在你的JDK的doc文件的`guide/collections`下面。

Java集合的一个典型特性是在迭代（iterating）一个集合的时候不应该修改这个集合（包括增加实体、移除实体、改变有序集合中实体的顺序），这也适合建立在集合上的视图，如使用`list[range]`。

### 4.4.1 了解并发修改

如果你没有遇到这种限制，你应该看看`ConcurrentModificationException`，例如，不能通过迭代时（iterating）每次移除第一个元素的方式来从列表中移除所有的元素：

```
def list = [1, 2, 3, 4]
list.each{ list.remove(0) }
// throws ConcurrentModificationException !!
```

注意：在这里并发不是意味着另外一个线程从根本上改变集合，如例子中所显示的那样，甚至一个单线程的控制也能改变结构的稳定性。

在这个例子中，正确的解决方法是使用`clear`方法，Collections API有许多专门的方法，当选择方案时，考虑`collect/addAll/removeAll/findAll/grep`。

这导致第二个问题：一些方法工作在集合的一个副本上并且完成的时候返回这个副本，而另外一些方法直接工作在调用的这个集合对象上（这个对象我们叫做接受者对象）。

## 4.4.2 识别副本和修改在语义上的不同

一般的，不容易预先识别到一个方法修改的是接受者对象还是这个集合的副本，一些语言对于这个有命名约定，但是groovy没有这样做，因为在groovy中直接访问了java的方法，并且java方法的名称没有遵从这种约定，groovy设法适应java集合API：

- 修改接受者对象的典型方法是没有返回一个集合，例如：

`add/addAll/remove/removeAll/retainAll`, 统计数量的sort方法。

- 不修改接受者对象并且返回一个集合的方法，例如：

`grep/findAll/collect`, 统计数量的sort方法，是的sort方法这两者都是，因为它返回一个集合并且修改了接受者。

- 修改接受者的方法有一个必要的名称，例如：

`add/addAll/remove/removeAll/retainAll/sort`,

`collect/grep/findAll`是一个有必要的，但是它们没有修改接受者而是返回的一个修改后的副本。

- 先前的规则也适用于操作符，操作符对应的方法名称：`<<leftShift`方法是必须的并且修改了接受者对象（指list，不是字符串，在java中字符串不能修改）；`plus`方法不是必须的并且返回的是一个副本。

没有清晰的规则为你导航，当你不有疑问的时候，看看文档或者写一些断言代码。

## 4.5 摘要

这一节讨论了groovy的数据类型，采用了许多不同的方式来了解新的知识。

我们介绍了range对象——不是一个控制结构，它有自己生存时间和创建地点，range对象能够作为一个参数传递给方法，并且能够作为一个方法调用的结果返回，这使得range非常灵活，并且由于range是存在的，因此它们可以用于许多无法简单控制的结构中，你已经看到的大多数例子是通过使用range作为list的下标数来萃取list中的元素。

List和map对于java程序员来说是非常熟悉的，但是也缺少后台语言的更好的支持，groovy认为list和map是非常常用的数据类型，它给它们在声明方面的特殊处理，当然为了方便而提供了操作符和附加的方法。List和map在groovy的用法和它们在java中的用法是相同的，并且遵循java的规则和限制，尽管通过附加的方法减少了操作集合的难度。

在我们讨论groovy数据类型的过程中，你已经看到了无处不在的闭包，这些闭包像函数一样简单，在下一章，我们将揭秘这个概念，讲解常用的和不常用的应用程序，并且指出你应该怎样使用闭包写自己的代码。

# 第五章 闭包

闭包是非常重要的，在 groovy 中闭包是最有用的特性之一，在没有理解闭包之前闭包对你来说是陌生的概念，为了最有效的使用 groovy，或者理解其他的 groovy 代码，你必须掌握闭包。

闭包不难——它们仅仅与你之前遇到的其他概念有些不同，在某种程度上，这是陌生的，因为面向对象的最高原则是对象有自己的行为和数据，闭包也是对象，他主要的目的是他们的行为——这几乎是闭包的全部。

在过去的几章中，你已经看到了闭包的一些用法，因此你也许对闭包已经有了一个比较好的认识，由于闭包十分重要，请原谅在这里的重复说明。

在这章，我们将介绍闭包的基本概念，说明闭包的优势，然后使用实例说明如何声明和调用闭包，在介绍了这些基本知识之后，我们将看看在闭包上的可用方法和闭包的范围方面的知识——也就是说，数据和成员变量可以在闭包内被访问，并且闭包可以有返回结果，在本章结束的时候将讨论怎么样用闭包来实现常用的设计模式，怎样通过不同的方式减少解决这个问题的其他需求。

## 5.1 出身名门的闭包

让我们以闭包的简单定义开始，然后通过一个例子进一步阐述。一个闭包是被包装为一个对象的代码块，实际上闭包像一个可以接受参数并且能够有返回值的方法。闭包是一个普通对象，因为你能够通过一个变量引用到它，正如你能够引用到任何别的对象一样。不要忘记 JVM 根本就不知你正在运行 groovy 代码，因此你能处理闭包对象也没有什么特别奇怪，闭包仅仅是一个对象，groovy 提供了一种非常容易创建闭包的方式和启用了一些非常常用的行为

如果把闭包与真实世界相联系的话，可以认为是一个信封和信封中的信纸，对于别的对象来说，信纸上也许有“`x=5,y=10`”等等之内的变量值，对于一个闭包，信纸也许有一个操作说明，你可以把这个信封给某个人，并且这个人也许按照信纸的指导来做工作；或者他们也将信封给别人的人，它们也许根据不同的上下文按照信纸的指导做很许多次工作。例如，信

纸上的内容也许是：“把信送给你思念的人”，这个人也许开始找到他思念的人的地址，然后根据地址列表，一次接一次的按照信纸的说明做——联系在地址列表中的每一个人（每次一个人）。

这个例子的等价的 groovy 代码应该是这样的：

```
Closure envelope = { person -> new Letter(person).send() }
addressBook.each (envelope)
```

相当直观，并且符合 groovy 的语言习惯，这显示了闭包的差异（在这个例子中，闭包是变量 envelope）和闭包的用法（作为 each 方法的参数），当第一次看到闭包的时候，难以理解的部分是他们通常用在一个简短的块中间，groovy 使闭包非常简明，因为闭包使用非常频繁——但是简洁增加了学习难度，通过比较，先前的代码是使用 groovy 提供的方式编写的。当你看到这种方式的时候，它经常是如下所示的结构：

```
addressBook.each { new Letter(it).send() }
```

这仍旧是使用闭包作为单个参数的方法调用，但这是难以发现的——在 groovy 中传递一个闭包给一个方法是非常平常，没有特殊规则。同样，如果闭包仅仅需要一个参数，groovy 提供了一个缺省的名称——it，因此，你不需要进行特别的声明，当我们使用 groovy 的简化操作的时候，我们的例子变得十分简短。

停下来并且想想为什么我们应该在第一个位置有闭包，始终记住：闭包是由一些代码组成的对象，并且 groovy 为闭包提供了简洁的语法。

## 5.2 闭包例子

Java 是一个优秀的平台：轻便的、稳定的、可伸缩的和适当的性能要求，java 语言有许多优势但遗憾的是也有许多缺点。

Java 的一些不足可以通过使用 groovy 的闭包来很好的解决，我们通过两个不同的方面来看看使用闭包的优势：使用集合执行每日例行任务和安全的使用资源，在这两种常用的情况下，你需要能够执行每天做相同工作的逻辑代码，在集合的例子中，逻辑代码在 iterator 块内，在资源处理的例子中，资源在使用之前必须进行请求，并且在使用之后进行释放，概括的说，使用一种“回调”机制来执行工作，闭包是 groovy 提供的进行透明回调的一种方

式。

### 5.2.1 使用迭代 (iterator)

在 java 代码中遍历一个集合的用法是使用迭代 iterator:

```
// Java
for (Iterator iter = collection.iterator(); iter.hasNext();) {
    ItemType item = (ItemType) iter.next();
    // do something with item
}
```

明确的实现了 Collection, 或者如 List 这样的接口, 也许也会通过这样的代码操作:

```
// Java
for (int i=0; i < list.size(); i++) {
    ItemType item = (ItemType) list.get(i);
    // do something with item
}
```

Java 通过两个新的特性改进了这种情况: 泛型和增强的 for 语句, 使用泛型之后不再需要进行强制造型; 增强的 for 语句与 groovy 的循环非常相似, 只需要把 “:” 改成 “in”:

```
// Java 5
for (ItemType item : list) {
    // do something with item
}
```

语法也许不完美——java5 的设计者拘泥于向 java 中增加关键字, 但这需要做更多的工作, 对吗? 对于同一件事情, 这种语法被限制在 java5 平台, 但许多开发者仍然在使用 java1.4 或者更早期的版本, 为了这样一个常用的操作需要强制的写更多的代码, 它相对简单, 熟悉了就会觉得平常, 并且在循环体内部的错误很容易被忽略,, 因为你通常在阅读代码的时候会在看完一个方法之后再来看循环体。

增强的 for 语句的第二个问题是让我们感觉它与闭包非常相似, 不管怎样, 清晰的迭代处理集合中的每一个条目是有用的; 否则, groovy 就没有必要拥有闭包了, 对于初学者来说 (groovy 的 for 语句比 java5 的稍微清晰一点, 参考第六章了解更详细的信息), 这是有用

的，但不是我们希望看到的全部东西。有通用的模式来表明为什么我们想遍历一个集合，比如在集合中的任何一个元素是否符合特定的条件，查找所有符合给定条件的结果，或者转换每一个元素到另外一个集合中，因此需要创建新的集合。

对于这些模式的处理有专门的语法看起来比较疯狂，使语言的实现以没有可扩展性而结束，就像通往森林的公路那样，当设计者知道预先需要做的事情时，这样是挺好的。但很快你将迷路，生活是坚强的，不需要语言对这些模式进行直接的语言支持，每一种模式都依赖于一遍又一遍的执行一个特定的代码块，集合中的每一个元素都执行一次，java 没有“特定的代码块”的概念，除非这些代码块被埋植在一个方法中，这个方法可以使一个接口的实现，但这样每一个代码块都需要一个自己的类（可能为匿名内部类），代码显得十分丑陋。

Groovy 使用闭包来指定这些每次都被执行的代码块，并且增加了许多额外的方法(each、find、findAll、collect 等等) 到集合类上，使他们容易使用，这些方法没有魔力，只是用来简化了 groovy，因为闭包能够把控制逻辑从每次执行的代码块中分离出来，如果你发现你想到的一个相似的结构没有在 groovy 中出现，可以容易的增加到 groovy 中。

把控制逻辑和每次迭代处理的逻辑分开不是介绍闭包概念的唯一原因，第二个原因（可能是更重要的原因）是在处理资源的时候使用闭包。

## 5.2.2 处理资源

许多时候你必须打开一个流并且在方法的结尾的时候调用 close 方法，当处理发生异常的时候也许 close 方法从来就没有执行到。因此，需要使用 try-catch 语句块，不，应该使用 try-finally 语句块，不是吗？在 finally 语句块中，close 方法也能抛出另外一个需要进行处理的异常，有太多的细节需要记住，因此资源处理的实现经常不正确，通过 groovy 的闭包支持，你可以像下面这样处理逻辑代码：

```
new File('myfile.txt').eachLine { println it }
```

file 的 eachLine 方法负责处理文件输入流的打开和关闭，这样避免你偶然的错误处理形成资源泄漏。

流仅仅是资源处理中非常小的一部分，数据库连接、本地处理如图形资源，网络连接——甚至你的 GUI 也是需要进行管理的资源（也就是说，在正确的时候进行重绘），并且观察者和事件监听器需要在适当的时候移除，或者以内存泄漏来结束。

忘记进行正确的清理工作可能仅仅在 java 初学者常犯这样的错误，由于语言对 try-catch-finally 这样的语句没有什么帮助，甚至有经验的开发人员也会犯这样的错误，在一个处好的代码内进行编写逻辑代码是可以的，没有经验的 java 程序员不进行资源的集中处理，代码因此而重复，并且不恰当的实现代码出问题的概率随代码的重复而增加。

资源处理代码经常是难以测试的，只有通过代码覆盖率进行完整的覆盖测试，由于重复，到处分布的资源处理的测试十分困难并且耗尽前期的开发时间，测试集中的处理十分容易并且只要求一个测试。

我们来看看 java 提供了什么样的资源处理和为什么它们经常没有用，然后我们展示相应的 groovy 的解决方案。

### 一般的 java 解决方法：使用内部类。

为了集中进行资源处理，你需要传递资源处理代码给处理器，这听起来有点熟悉——本质上与我们处理集合是一样的：处理器需要知道怎样调用这些代码，因此它必须实现一些知道的接口，在 java 中，常常基于 2 个原因实现为一个内部类：首先，允许资源使用代码靠近调用的代码（通常是为了可读性），其次，允许资源使用代码与调用的上下文进行交互，使用本地变量，调用相关对象的方法等等。

匿名内部类几乎有相同的用法，如果 java 有闭包，匿名内部类可能都不会被发明出来。由于规则和限制使它们有明显的缺点，很快你将看到这样的代码 MyClass.this.doSomething，你明白有些事情不正确，这使你分心于在开始创建的代码，与上下文交互调用的代码是有限制的，比如为了访问本地变量，本地变量必须为 final 的。

通过一次方式可以正确的实现，但代码十分难看，尤其在经常使用的时候，java 的局限性使得这不是一个优雅的解决方案。下面的例子中，从 ResourceHandler 中获取到的 Resource 负责构建和销毁，仅仅是粗体代码才是对工作有用的代码：

```
// Java
interface ResourceUser {
    void use(Resource resource)
}
resourceHandler.handle(new ResourceUser() {
    public void use (Resource resource) {
        resource.doSomething()
    }
})
```

```
});
```

与之等价的 groovy 代码则显示了所有必须的信息，并且没有多余的无用代码：

```
resourceHandler.handle { resource -> resource.doSomething() }
```

groovy 的代码保持足够的灵活和功能，移除了使用内部类时的无用代码。

### 另外一种方式：使用模板方法模式

在 java 中另外一种集中处理资源的策略方式是在父类中进行，而在子类中编写资源使用的代码。这是模板方法模式的典型实现。

在这里，使用子类或者内部（也许是匿名的）子类，这也带来了我们前面说过的问题，也有代码不清晰和实现不自主的问题，当继承复杂的时候也很痛苦，使我们的代码抽象起来特别痛苦。

如果仅仅有一个接口被用作包括处理逻辑，像我们在前面例子中假想的 ResourceUser 接口，这样事情不会太坏，但是在 java 中没有一个单独的 ResourceUser 接口来做所有的事情，该接口的回调方法 use 需要能适应以下目的：参数的数量和类型，声明的异常的数量和类型及返回类型。

因此，随着时间变化而出现了大量的接口：Runnable、Observers、Listeners、Visitors、Comparators、Strategies、Commands、Controller 等等，这使得它们的用法更加复杂，因为对于一个新的接口，这也是新的抽象或者概念，这也需要理解。

通过对比，groovy 的闭包能处理任何形式的方法，并且控制逻辑的行为也可能根据闭包的形式不一样而发生改变，这在后面你将看到。

Java 的这两个痛苦的例子可以通过闭包简单的实现，如果这里的问题仅仅是通过闭包来使工作容易些，那么闭包也是高效的。但真实的情况也许更丰富，在没有闭包的情况下，许多设计模式的实现是难以想象的。

在开始你的梦幻之旅之前，需要学习更多关于闭包的基本知识，让我们先从闭包的声明开始吧。

## 5.3 声明闭包

迄今为止，我们已经使用了闭包的简短语法：在一个方法调用的后面，放置闭包代码在一对花括号里，闭包的参数和代码通过箭头(->)进行分隔。

让我们从你知道的简单的语法形式开始，然后我们将看看另外两种不同的声明闭包的方式：通过变量赋值和引用方法。

### 5.3.1 简单的声明方式

列表 5.1 显示了闭包语法的简单方式和一个新的便利特性，当只有一个参数传递给闭包的时候，这个参数的声明是可选的，魔术变量 `it` 代替了声明，看看在列表 5.1 中两个等价的闭包声明方式。

**Listing 5.1 Simple abbreviated closure declaration**

```
log = ''  
(1..10).each{ counter -> log += counter }  
assert log == '12345678910'  
  
log = ''  
(1..10).each{ log += it }  
assert log == '12345678910'
```

注意不像 `counter`，魔术变量 `it` 不需要进行声明。

这种语法是一个缩写形式，因为闭包对象通过花括号声明，该对象作为方法的最后一个参数并且通常出现在圆括号内，就像你看到的这样，这个闭包对象就像其他放在圆括号内的参数一样有效，尽管很少使用这种方式：

```
log = ''  
(1..10).each({ log += it })  
assert log == '12345678910'
```

这种语法是简单的，因为它只有一个参数——隐式参数 `it`，能够在一个序列中声明多个参数，假如没有值从方法传递到闭包，我们将在 5.4 节看到这样的例子。

**提示:** 箭头是闭包的参数和代码的分隔指示符。方法的传递值给箭头左边参数，箭头的右边是闭包的代码。

### 5.3.2 使用赋值的方式声明闭包

第二种声明闭包的方式是直接将它赋给一个变量：

```
def printer = {line -> println line }
```

闭包声明在花括号中并且赋给了 printer 变量。

**提示:** 什么时候你看到闭包的花括号，思考：`new Closure(){}。`

通过方法的返回值也是一种声明闭包的方式：

```
def Closure getPrinter() {  
    return { line -> println line }  
}
```

再一次，花括号指明构建了一个新的闭包对象，这个对象通过方法的调用返回。

**提示:** 花括号能用来标明构建了一个新的闭包对象或者一个 groovy 代码块，代码块可以是类、接口、`static`、对象的初始化代码、方法体，或者与 groovy 的关键字 (`if`、`else`、`synchronized`、`for`、`while`、`switch`、`try`、`catch` 和 `finally`) 一起出现，其它出现的形式就是闭包。

正如你看的那样，闭包是对象，他们可以通过变量进行引用，能够作为参数传递，并且，就像你猜测的那样，你可以在闭包上调用方法，作为对象，闭包也可以是一个方法的返回结果。

### 5.3.3 引用一个方法作为闭包

第三种声明闭包的方式是重用已有的声明：一个方法。方法有一个方法体，可选的返回值，能够接受参数，并且能够被调用，与闭包的相似性是显而易见的，因此 groovy 让你重用你已经在方法中存在的代码，但是作为一个闭包，引用一个方法作为闭包是使用 reference.& 操作符，reference 是闭包调用时使用的对象实例，正如一个一般的方法调用 reference.someMethod()。图 5.1 显示了声明一个方法闭包。

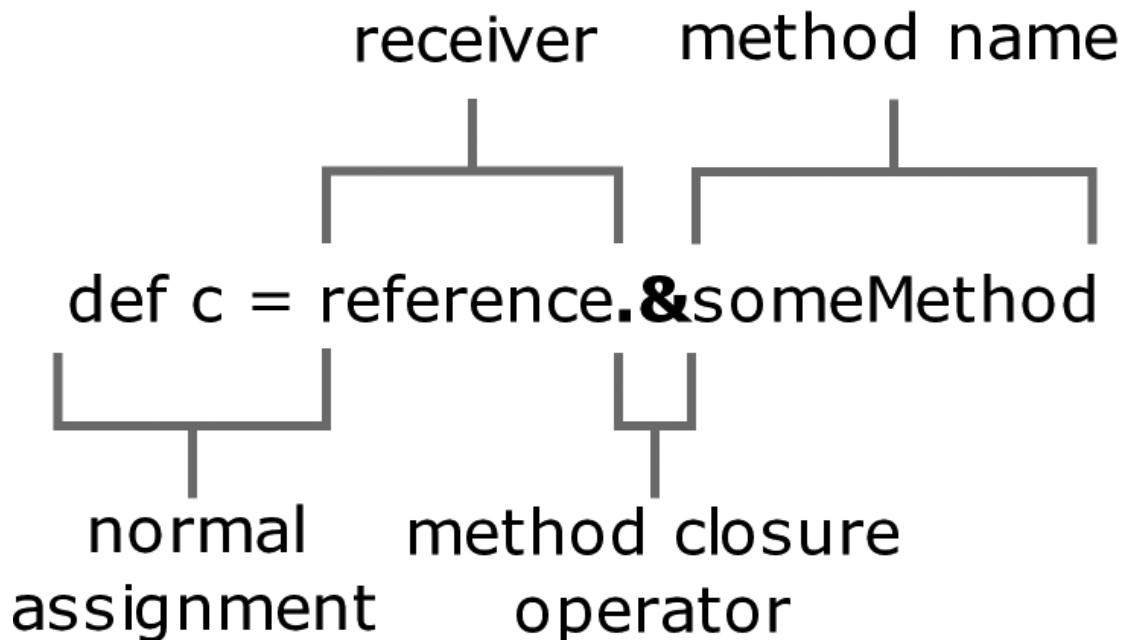


图 5.1 简单方法闭包赋值语句的解析

列表 5.2 演示了方法闭包在实际中的用法，显示两个不同实例被用来获取两个不同的闭包，即使在两个对象上调用的都是相同的方法。

## Listing 5.2 Simple method closures in action

```
class MethodClosureSample {  
    int limit  
  
    MethodClosureSample (int limit) {  
        this.limit = limit  
    }  
  
    boolean validate (String value) {  
        return value.length() <= limit  
    }  
}  
  
MethodClosureSample first = new MethodClosureSample (6)  
MethodClosureSample second = new MethodClosureSample (5)  
  
Closure firstClosure = first.&validate  ←② Method closure assignment  
  
def words = ['long string', 'medium', 'short', 'tiny']  
assert 'medium' == words.find (firstClosure)  ←③ Calling the closure  
assert 'short' == words.find (second.&validate)  ←④ Passing a method closure directly
```

每一个实例（在（1）创建）的 validation 方法都独立的判断给定的字符串是否有效，我们在（2）处使用 first.&validate 创建一个引用，通过 second.&validate 显示这个引用可以作为变量被传递（如(3)），或者作为一个参数传递给 find 方法(如(4))，我们使用简单的单词列表来检查闭包是否像我们期望的那样工作。

方法闭包被限制在一个实例方法，但他们有另外一个有趣的特性——运行时重载，就象我们知道的(方法多态)mymethods，在第七章将了解到更多的(方法多态)mymethods，表 5.3 给出了一个体验。

### Listing 5.3 Multimethod closures—the same method name called with different parameters is used to call different implementations

```
class MultiMethodSample {  
  
    int mysteryMethod (String value) {  
        return value.length()  
    }  
  
    int mysteryMethod (List list) {  
        return list.size()  
    }  
  
    int mysteryMethod (int x, int y) {  
        return x+y  
    }  
}  
  
MultiMethodSample instance = new MultiMethodSample()  
Closure multi = instance.&mysteryMethod  
  
assert 10 == multi ('string arg')  
assert 3 == multi ([['list', 'of', 'values'])  
assert 14 == multi (6, 8)
```

1 Only a single closure is created

2 Different implementations are called based on argument types

这里确实是使用的单个实例，并且确实是一个闭包（在（1）处），但每次调用的时候，不同的方法被调用（在（2）处），在第七章将了解到这种动态行为的更多信息。

现在你已经看到了所有的声明闭包的方式，值得停下来一会儿，并且把它们放在一起，通过不同的声明执行相同的功能。

#### 5.3.4 比较

列表 5.4 显示了创建和使用闭包的所有方式：使用简单声明、赋值给变量和方法闭包。在每一个例子里，我们在一个 map 上调用 each 方法，提供一个闭包，这个闭包将值\*2，在结束的时候，我们已经对值\*2 操作了三次。

### Listing 5.4 Full closure declaration examples

```
map = {'a':1, 'b':2}  
map.each{ key, value -> map[key] = value * 2 } ← 1 Parameter sequence with commas  
assert map == {'a':2, 'b':4}  
  
doubler = {key, value -> map[key] = value * 2 } ← 2 Assign and then call a closure reference  
map.each(doubler)  
assert map == {'a':4, 'b':8}  
  
def doubleMethod (entry){  
    map[entry.key] = entry.value * 2  
}  
doubler = this.&doubleMethod  
map.each(doubler)  
assert map == {'a':8, 'b':16}
```

3 A usual method declaration

4 Reference and call a method as a closure

在（1），我们直接把闭包作为参数传递，这是最常用的一种方式。

在（2）声明闭包的方式是与后面的用法不连贯的，花括号是 groovy 声明闭包的方式，因此我们将闭包对象赋值给变量 `doubler`，一些人不正确的理解这行为：将闭包调用的结果赋值给变量，不要掉进这个陷阱，闭包还没有被调用，仅仅是进行了声明。你看到的通一个实际的引用把闭包作为一个参数传递给 `each` 方法，这种方式与直接在这个地方声明闭包的效果是一样的。

在（3）的方法声明是一个普通的方法，这里没有发现使用闭包的痕迹。

在（4），`reference.&`操作符用来引用方法名称为一个闭包，这一次，方法也没有立即被调用；执行的代码在接下来的一行，这就像（2）一样，闭包被传递给 `each` 方法，这个方法为 `map` 中的每一个实体进行回调。

在 groovy 中类型是可选的，因此闭包的参数是可选的，如果闭包的参数进行了显式的类型声明，那么类型的检查发生在运行时而不是在编译的时候。

为了完全理解闭包怎样工作和怎样使用闭包，你需要明白如何调用它们，这就是接下来一节的主题。

## 5.4 应用闭包

迄今为止，为了传递闭包给方法执行，你已经看到了如何声明一个闭包，如例子中的 `each` 方法。但是在 `each` 内部到底做了什么呢？它们如何调用闭包？如果知道这些，你能很快的提出一个等价的实现，我们首先看看如何简单的调用一个闭包，然后继续看看 `Closure` 类提供的高级方法。

### 5.4.1 调用闭包

假设我们有一个引用 `x` 指向一个闭包，我们能通过 `x.call()` 来调用闭包，或者简单的 `x()`，你大概猜测到可以传递任何参数给闭包方法。

我们通过一个简单的例子开始，列表 5.5 显示了同一个闭包的两种不同的调用方式。

### Listing 5.5 Calling closures

```
def adder = { x, y -> return x+y }

assert adder(4, 3) == 7
assert adder.call(2, 6) == 8
```

我们首先声明了一个十分简单的闭包——这个闭包返回两个参数的和，然后我们直接调用闭包和使用 call 方法调用闭包，这两种方式调用闭包获得的效果是一样的。

现在，我们来试试更复杂的一些东西，在列表 5.6 中，我们从一个方法内部调用一个闭包，例子显示了闭包执行的时机：

### Listing 5.6 Calling closures

```
def benchmark(repeat, Closure worker){
    start = System.currentTimeMillis() ← ① Put closures last
    repeat.times{worker(it)} ← ② Some pre-work
    stop = System.currentTimeMillis()
    return stop - start ← ③ Call closure the
                           given number
                           of times
}
slow = benchmark(10000) { (int) it / 2 } ← ④ Some
fast = benchmark(10000) { it.intdiv(2) }   post-work
assert fast * 15 < slow ← ⑤ Pass different
                           closures for
                           analysis
```

还记得我们在列表 3.7 中进行正则表达式的性能测试吗？我们需要重复基准逻辑，我们没有声明怎样的基准 something。现在你知道了，你可以传递一个闭包到 benchmark 方法，在这个方法里，可以做一些预处理和后处理工作。

在（1）的地方我们把闭包作为最后一个参数，这样在调用该方法的时候可以使用闭包的简单语法声明方式。在这个例子中，我们声明了闭包的类型，这样仅仅是为了让事情显得更明了，Closure 类型是可选的。

实际上，开始基准测试的地方是在（2），从一个常规的方式来说，这里是像打开文件或者连接数据库之类的预处理代码，这通常发生在资源处理的时候。

在（3），我们根据 repeat 参数重复的调用闭包。我们把当前正在重复的次数传递给闭包，这样使得工作更加有趣，一般情况下是把一个资源传递给闭包。

在（4）的地方停止测试并且计算调用闭包所用的时间，这里放置的是后处理工作：关闭文件、冲洗缓冲区（flushing buffers）、将连接返回给连接池等等。

在（5）开始测试，我们能传递逻辑给 benchmark 方法，注意我们使用闭包的简单声明方式并且使用魔幻的 it 来引用当前的计数，我们可以得知一般的数字除法比优化的 intdiv 方

法要多运行 15 倍。

顺便说一下：这种类型的基准测试不太严谨，所有的动作都能严重的影响测试的效果：计算机的配置、操作系统、当前计算机的加载过程、JDK 的版本、JIT 编译器和 Hotspot 设置等等。

图 5.2 显示了一般情况下通过声明方式创建闭包然后调用闭包的 UML 序列图，在 caller 上进行方法调用，并且 caller 回调到给定的闭包。

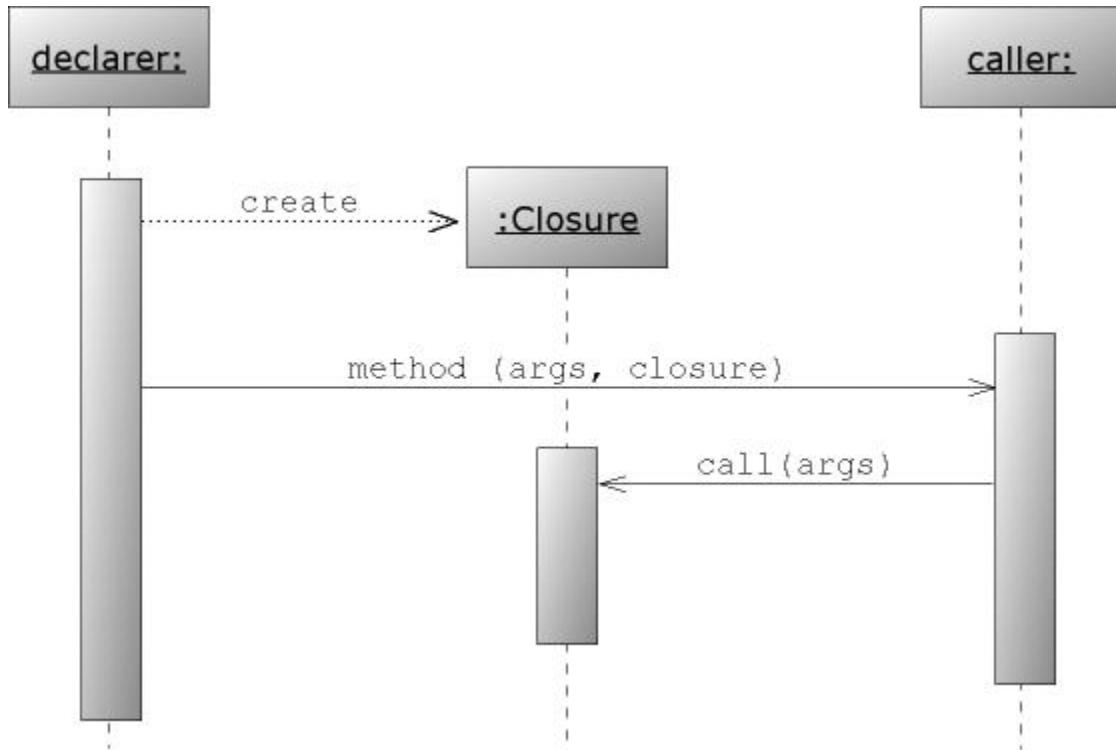


图 5.2 闭包的典型的调用方式

在调用闭包的时候，需要传递所有准确的参数给闭包，除非闭包定义了缺省值，当忽略了某个参数的时候，这个参数的缺省值被使用，下面的代码是列表 5.5 的一个变种，第二个参数有一个缺省值，这样就可以进行两种调用——一是传递两个参数，另外一种是传递一个参数（第二个参数使用缺省值）：

```
def adder = { x, y=5 -> return x+y }
assert adder(4, 3) == 7
assert adder.call(7) == 12
```

对于在闭包中使用缺省值参数的方式，同样的规则可以应用到方法上面；同样，闭包可  
如果满意该文档，请支持(招商银行北京分行双榆树支行 6225881008887381 账户名：吴翊)

以使用的可变长度的参数列表也适用在方法上，在 7.1.2 节将讨论这些。

在这里，你应该知道传递闭包给方法和怎样回调执行有了一个基本的理解，也可以看看图 5.2 的 UML 图，每当你传递一个闭包给方法的时候，你应该确保有一种方式进行回调（也许仅仅是条件），这依赖方法的逻辑实现，在下一节，将看到闭包更多的功能。

## 5.4.2 更多的闭包方法

类 groovy.lang.Closure 是一个普通的 java 类，尽管它有非常强大的功能和额外的语言支持，它有各种各样不同功能的调用方法，我们将看看最重要的一些方法——即使你通常仅仅声明和调用闭包，在需要的时候知道一些额外的功能是好的。

### 参数的数量

一个有用的简单的例子是 map 的 each 方法作用在闭包上的参数的数量，我们在 4.3.2 节讨论过，这个方法传递一个 Map.Entry 对象的参数或者 key 和 value 分离的参数给闭包，依赖闭包是接受一个参数还是两个参数，通过闭包的 getParameterTypes 方法你能了解到你期望的参数数量信息（类型，如果声明了类型）：

```
def caller (Closure closure) {
    closure.getParameterTypes().size()
}
assert caller { one -> } == 1
assert caller { one, two -> } == 2
```

在 Map.each 例子中，这允许支持不同参数风格的闭包，以适应调用者的需要。

### 怎样使用闭包实现 curry

curry 的基本思想是一个多个参数的函数，传递较少的参数给函数来固定一些值，一个典型的例子是选择一些数 n 并且传递给函数与后面传递的单个参数进行相加，

在 groovy 中，Closure 的 curry 方法返回当前闭包的一个克隆品，这个克隆品已经绑定了一个或者多个给定的参数，参数的绑定是从左向右进行的，列表 5.7 给出了一个实现。

### Listing 5.7 A simple currying example

```
def adder = {x, y -> return x+y}
def addOne = adder.curry(1)
assert addOne(5) == 6
```

我们重用了同一个闭包，在调用 `curry` 方法的时候创建了一个新的闭包，这个闭包实际上像一个简单的加法器，但是第一个参数已经被固定为 1，最后，我们检查我们的结果。

如果你是闭包或者 `curry` 的新手，现在是暂停的好时机——重新回到开始 `curry` 的讨论的地方，再读一遍，这是一个迷惑的概念，慢慢的掌握它，你会感觉到 `curry` 是一个好东西的。

`Curry` 最强大的地方是当闭包的参数是闭包本身的时候，这通常在函数式编程时使用到。

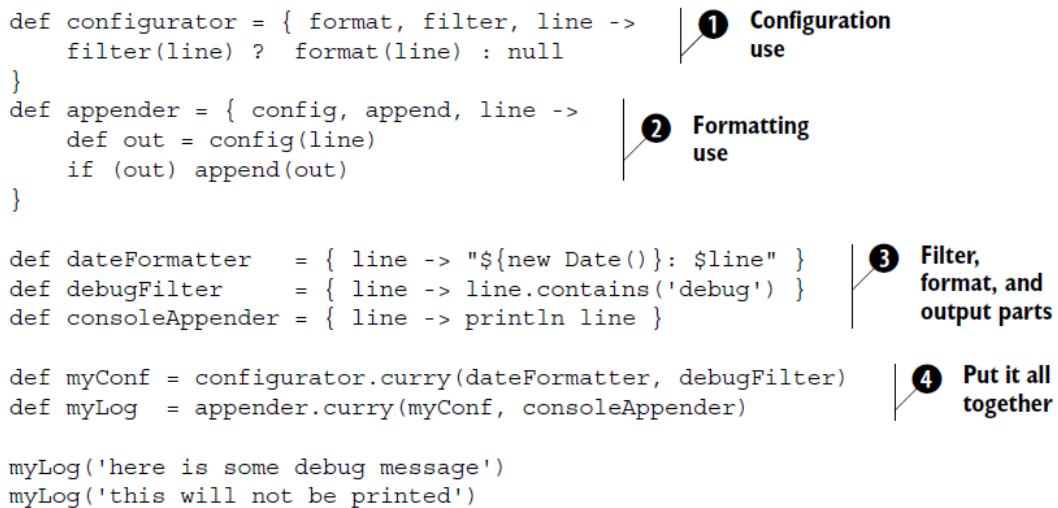
假设你需要实现一个日志记录器，它应该支持行数的过滤，日志的格式化，并且输出它们到一个设备上，每一个记录器都应该是可配置的，办法是提供一个闭包作为每一个记录器自定义版本，这仍旧允许你实现应用一个过滤器，格式化和最终的日志输出，下列的代码显示了 `curry` 怎样被用来注入到自定义的记录器中：

```
def configurator = { format, filter, line ->
    filter(line) ? format(line) : null
}
def appender = { config, append, line ->
    def out = config(line)
    if (out) append(out)
}

def dateFormatter = { line -> "${new Date(): $line}" }
def debugFilter = { line -> line.contains('debug') }
def consoleAppender = { line -> println line }

def myConf = configurator.curry(dateFormatter, debugFilter)
def myLog = appender.curry(myConf, consoleAppender)

myLog('here is some debug message')
myLog('this will not be printed')
```



① Configuration use  
② Formatting use  
③ Filter, format, and output parts  
④ Put it all together

闭包(1)和(2)像是一个食谱：给出任何过滤器、输出格式和目标，一个行数据将可能被记录到日志中，它们执行这个工作：适当的委派，在(3)的闭包是食谱中的配料，它们每次都每使用，但是我们总是使用同一种配料，`curry`（在(4)）允许仅仅记录一个对象，而不是单独的每一个对象。继续类推，我们放所有的配料在一起，结果就是我们需要记录的日志内容。

日志经常作为一个无趣的主题被忽略，但实际上，在前面代码中的一些日志证明这个想法是错误的，作为一名严谨的工程师，你明白日志语句经常使用，并且任何日志组件必须注意性能问题。在这一方面，当没有日志被记录的时候，应该不影响性能。

在这个例子中耗时的操作是格式化和打印，过滤是很快的。借助闭包，我们醉心于代码的模式以保证昂贵的操作不被调用（就是不需要打印的日志），configurator 和 appender 闭包实现了这个模式。

这个模式是极其灵活的，因为怎样过滤的逻辑，怎样应用格式化和这样输出结果都是可配置的（甚至在运行时）。

利用闭包和闭包的 curry 方法，我们完美的解决了最佳的相关性和最低的耦合性的问题。注意每一个闭包实际上都是独立的。

这是函数式编程的开始，在 <http://www-128.ibm.com/developerworks/library/j-pg08235/>（译者注：中文版网址：<http://www.ibm.com/developerworks/cn/java/j-pg08235/>）上面可以了解到使用 groovy 的闭包进行函数式编程的相关文章，这篇文章进一步阐述了如何使用这种方式来实现自己的表达式语言，找准业务规则和检查你的代码。

### 通过 isCase 方法进行分类

闭包实现了 isCase 方法，这样闭包可以在 grep 和 switch 中作为分类器使用，在这种情况下，各自的参数传递给闭包，然后调用闭包进行计算得到一个 Boolean 值（参考 6.1 节），正如你所见：

```
assert [1,2,3].grep{ it<3 } == [1,2]
switch(10){
    case {it%2 == 1} : assert false
}
```

这样可以让我们使用任何逻辑进行分类，又一次证明了这种可能性，因为闭包也是对象。

### 其余的方法

基于完整性考虑，这里说说闭包支持的通常 java 意义下的 clone 方法。

asWriteable 方法返回一个当前闭包的克隆版，这个克隆版有一个附加的 writeTo(Writer) 方法来直接将闭包的结果输出到给定的 Writer 中。

最后，有一个为属性 delegate 的 setter 方法和 getter 方法，在下一节的时候就了解到什么是 delegate 并且在闭包中如何使用它。

## 5.5 理解范围

你已经看到了如何创建闭包及如果使用闭包，这个功能非常强大但是使用起来仍会很简单。

这一节来看看下层的东西，以加深你在使用这种简单的方式创建闭包的时候对后台工作原理的理解，我们仔细看看在闭包的内部有什么数据和方法可以使用，使用 `this` 引用到的是什么，及怎样用你的知识来测试设计的一个典型例子和测试任何语言表达式。

这有点技术难度，你在第一次阅读的时候可以安全的跳过，不管怎样，在适当的时候你也许想来看看这一节并且了解 groovy 是如何提供这些聪明的技巧的。事实上，明白细节经常能让你自己想出特别优美的解决方案。

那么在一个闭包内什么是可用的，这个范围包括：

- 什么样的本地变量可以被访问
- `this`（当前对象）引用到的是什么
- 什么属性和方法是可以访问的

我们以已经看到过的例子来说明，为了说明，我们重新来看看循环 10 次做一些事情的代码：

```
def x = 0
10.times {
    x++
}
assert x == 10
```

很明显，传递给 `times` 方法的闭包可以访问变量 `x`，在声明闭包的时候本地变量是可以访问的，记住：花括号显示了闭包声明的时间，不是执行的时间。在闭包的声明期间，闭包可以对变量 `x` 进行读和写操作。

这引起第二个想法：闭包在执行的时候想必也要访问 `x`，否则 `x` 是如何递增的呢？但是闭包被传递给 `times` 方法，这是在一个值为 10 的 `Integer` 上进行的调用，这个方法，依次回调我们的闭包，但是 `times` 方法没有知道变量 `x` 的机会，因此它不能传递 `x` 给闭包，想必也不知道闭包使用了 `x` 变量。

唯一能使这工作的途径可能是闭包在它的生命周期里以某种方式记住了它的工作上下

文环境，当调用它的时候，闭包可以在它的原始上下文中工作。

这种生命周期的上下文需要闭包记住相应的一个引用，不是一个复制品，如果工作上下文是原始上下文的一个复制品，从闭包中是没有办法改变原始上下文的。但是我们的例子清晰的显示了改变了 x 的值——否则断言将失败，因此，生命周期上下文必须被引用。

### 5.5.1 简单的变量范围

图 5.3 描述了 time 中对象被调用的理解以及它们之间的相互引用关系。

Script 创建了一个 Closure (闭包)，这个 Closure 有一个反向引用到 x，这个 x 是在闭包的声明的范围之内，Script 在 Integer 10 的对象上调用 times 方法，传递声明的闭包作为参数，换句话说，当 times 方法被执行的时候，一个引用指向堆栈中的闭包对象，times 方法使用这个引用执行闭包的 call 方法，传递了本地变量 10 给闭包，在这个特定的例子中，计数变量没有在闭包的 call 方法中被使用，在闭包的 call 方法中仅仅使用了在 Script 中的 x 变量进行工作。

通过分析，你看到闭包在声明的时候，绑定了本地变量的引用。

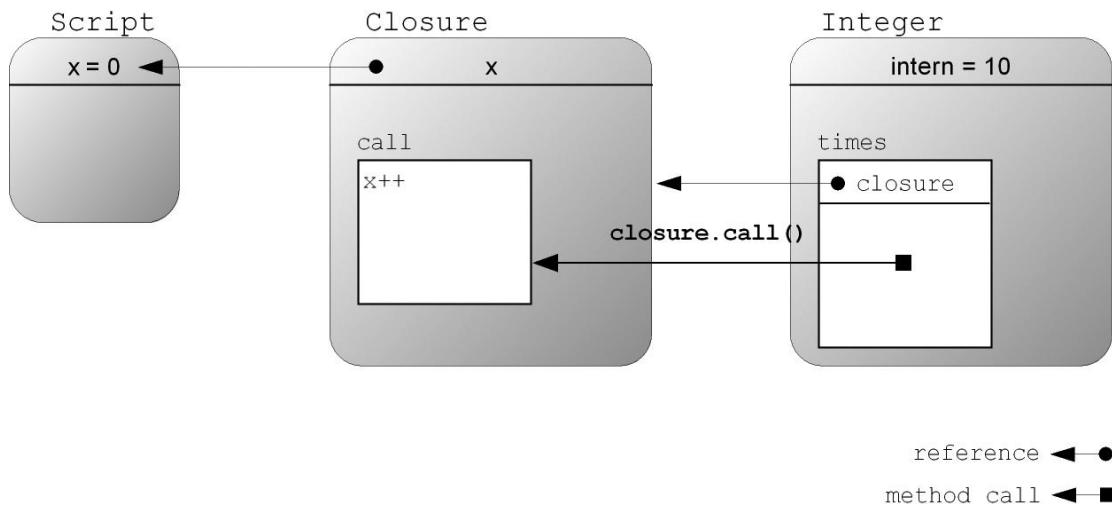


图 5.3 对象引用和方法调用的方案图，在脚本中一个值为 10 的 Integer 对象被使用，闭包作为参数传递给 Integer 的 times 方法，这样闭包可以被调用 10 次。

### 5.5.2 闭包范围

别的范围的元素也可以像本地变量一样被访问是不令人感到意外的：this 的值，属性、

方法和参数。

这通常都是正确的，但是 `this` 引用时有个特例，在闭包中，你正确的猜想 `this` 将引用到当前对象，这是闭包对象本身，在另一方面，使用 `this.reference` 和直接使用 `reference` 来处理本地引用是没有区别的。

通过一个名称为 `owner` 的变量可以获取到声明闭包的对象，列表 5.8 延展了最初例子的目的，显示了其余范围的元素。

我们实现一个类 `Mother`，这个类通过 `birth` 的方法来获取到一个闭包，这个类有一个字段，另外的一个方法，参数列表和本地变量，这个闭包返回这些在当前上下文（或者范围）所有元素的列表，在后台，这些元素在声明的时候被绑定而不是在闭包被调用的时候，让我们来看看调用的结果吧。

#### Listing 5.8 Investigating the closure scope

```
class Mother {  
    int field = 1  
    int foo(){  
        return 2  
    }  
    Closure birth (param) { ← ① This method creates and  
        def local = 3  
        def closure = { caller ->  
            [this, field, foo(), local, param, caller, this.owner]  
        }  
        return closure  
    }  
}  
  
Mother julia = new Mother()  
closure = julia.birth(4) ← ② Let a mother give  
context = closure.call(this) ← ③ Call the closure  
println context[0].class.name ← ④ Script  
assert context[1..4] == [1,2,3,4] ← ⑤ No surprise?  
assert context[5] instanceof Script ← ⑥ The calling object  
assert context[6] instanceof Mother ← ⑦ The declaring object  
firstClosure = julia.birth(4)  
secondClosure = julia.birth(4)  
assert false == firstClosure.is(secondClosure) ← ⑧ Closure braces  
                                            are like new
```

在（1）我们在方法声明的时候增加了返回类型，指明这个方法将返回一个闭包对象，一个返回闭包的方法通常不是闭包的用法，但是有时候这是方便的，注意在这个方法中我们是在声明期间，闭包的列表将被返回，但是闭包仍然没有调用。

在构建了一个新的 `Mother` 实例之后，我们在（2）调用它的 `birth` 方法用来接收一个新

的闭包对象，到现在，闭包还没有被调用，元素列表仍然还没有被构建。

在 (3)，我们显式调用闭包，闭包从它出生的地方构建元素列表，我们将列表保存在一个变量中供后面查阅，注意我们把自身作为参数传递给闭包，这样可以在闭包中作为 `caller` 使用。

在 (4) 输出了这次运行的时候的脚本类的名称，groovy 1.0 之前的版本打印的是闭包类型。

实例变量 `field`, `foo()` 调用的结果，本地变量 `local`, 参数 `param`, 这些所有的都有期望的值，在 (5) 证明了这个结果，当执行闭包的时候它们不知道 `Script`，这是我们预期的重新调用，只有 `foo()` 有点复杂，它总是 `this.foo()` 的缩写，正如我们前面说的那样，`this` 引用到闭包，而不是声明闭包的对象，在这里，闭包有点糊弄人，它们将所有的方法调用代理给 `delegate` 对象，`delegate` 缺省是声明闭包的对象（也就是 `owner`），这样保证了闭包在它的上下文中运行。

显式传递给闭包的 `caller` 参数确保了在闭包中可以进行访问，在 (6) 显示了用法，在这本书中前面的所有闭包的例子中，调用闭包的对象和声明闭包的对象都是相同的，因此，我们可以容易的使用，你也许注意到了调用者对象都是声明对象，如果这样让你感觉很疯狂，不要担心，概念也许稍微有点不熟悉，从 `times` 例子重新开始，如果你想这样做，不算太晚。

在闭包中，魔术变量 `owner` 引用到声明闭包的对象，如 (7) 所示。

如 (8) 每次调用 `birth` 方法的时候，闭包都重新构建，对于闭包来说，闭包的花括号就像 `new` 关键字一样，这后面是闭包和方法的根本差别，方法在类产生的时候就被构建，而且只会构建一次，闭包对象在运行的时候被构建，并且相同的代码也有可能被构建多次。

图 5.4 显示了列表 5.8 的引用关系。

关于闭包的语法来自如 `Lisp`、`smalltalk`、`perl`、`ruby` 和 `python` 等语言。

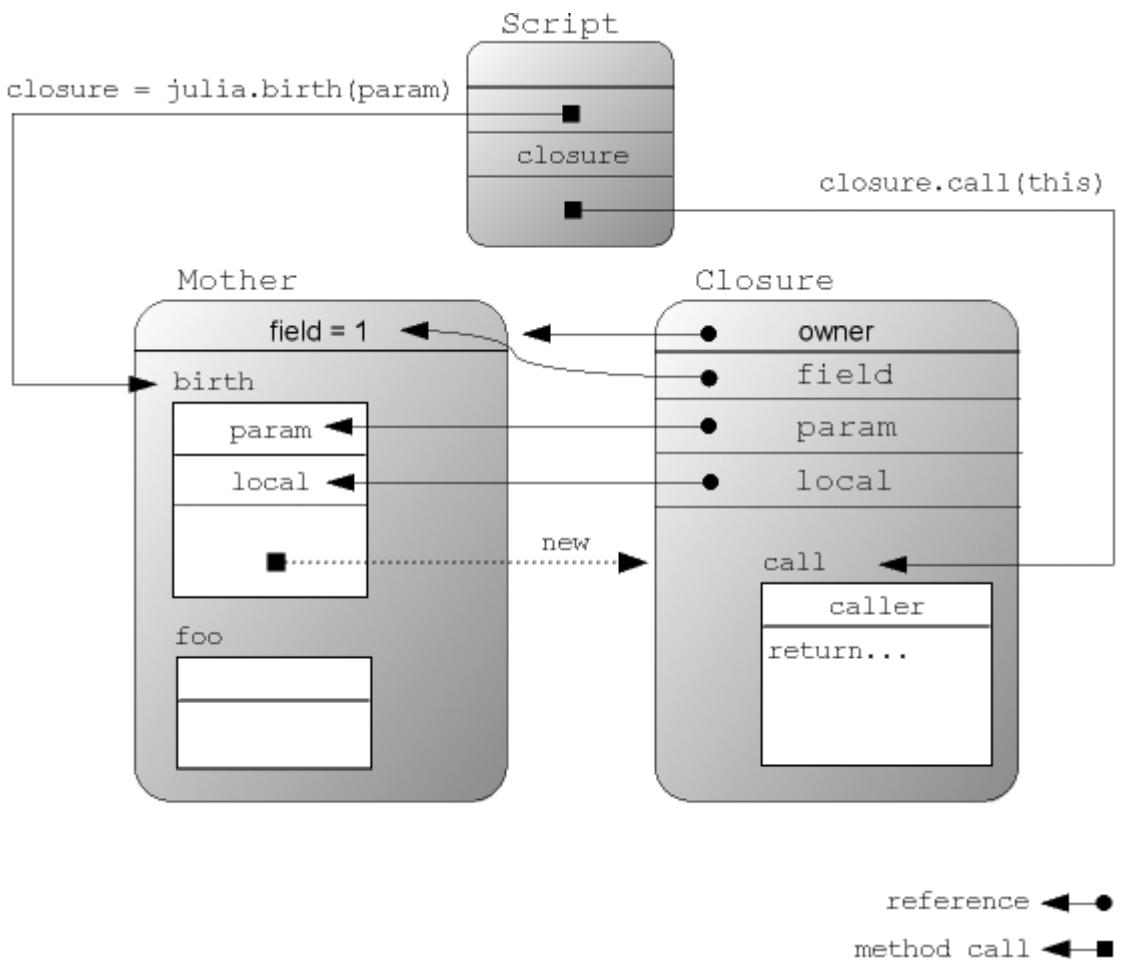


图 5.4 列表 5.8 的例子中对象引用关系图

我们的目的是适当的介绍 groovy 的闭包，这有助你在网上寻找更复杂的例子的时候给你基本的理解，而不是给出一个蓄意费解的例子，无论如何，我们提供一个关于闭包范围的例子，以使别的复杂的任务简单易懂。

### 5.5.3 工作中的范围使用：典型的累加测试

这里有基本的例子来比较各种语言支持闭包的强大程度，其中之一是语言应用闭包是的范围的强大程度，第一个提出这种测试的是Paul Graham，参考(<http://www.paulgraham.com/icad.html>)，在这个测试中，他的想法十分有趣并且和可读。他谈论了不同语言做的方式，在切换到groovy的时候你能发现好的参数。

在一些语言中，这个测试带来了戏剧性的解决方式，但在groovy中不是这样的，groovy的解决方式非常特殊并且实现简单。

这是原始需求的描述：“我们想写一个生成积聚的函数，这个函数接收一个数字n，并且返回一个函数，这个函数接收另外一个参数i，并且返回n+i的结果”

下面是别的语言的解决方法：

在Lisp中：

```
(defun foo (n)
  (lambda (i) (incf n i)))
```

在Perl 5中：

```
sub foo {
  my ($n) = @_;
  sub {$n += shift}
}
```

在Smalltalk中：

```
foo: n
| s |
s := n.
^[:i| s := s+i. ]
```

下列步骤是groovy的解决方法，显示在列表5.9中：

1、 我们需要一个函数，这个函数返回一个闭包，在groovy中，我们没有函数，但是有方法（事实上，groovy中不仅有方法，还有闭包，现在我们保持足够的简单），使用def来定义一个方法，这个方法只有一行，在方法的返回语句中创建了一个新的闭包，我们将调用foo这个方法来实现通过size进行比较的功能，createAccumulator能更好的反映出这个目的。

2、 方法需要一个初始化参数n；  
3、 由于n是声明闭包的方法的一个参数，因此n可以直接在闭包中使用并且进行递增操作。  
4、 递增值被计算同时也作为一个新值赋值给n，这样我们可以得到真实的累加结果。

我们增加了一些断言来检查我们的结果和展示累加器是如何被使用的，列表5.9展示了完整的代码。

### Listing 5.9 The accumulator problem in Groovy

```
def foo(n) {  
    return {n += it}  
}  
  
def accumulator = foo(1)  
assert accumulator(2) == 3  
assert accumulator(1) == 4
```

在学习闭包之后，上面所有的步骤的解决方法是十分简单的。

与别的语言相比，groovy的解决方法简短而又清晰，groovy已经通过了这类语言的异常有效的测试。

这种测试与实际应用有关吗？我们也许认为在某个方面不需要一个加法生成器，但是在不同的方面，通过这种测试意味着语言能动态的增加逻辑到一个对象并且管理这个对象的应用上下文环境，这意味着这个语言能进行十分强大的抽象。

## 5.6 从闭包返回结果

到目前为止，你已经知道了如何声明闭包和如何调用闭包，现在我们还有一个重要的主题没有涉及到：怎样从闭包返回结果。

在理论上，有2种返回结果的方式：

- 闭包最后一个表达式执行的结果，并且这个结果被返回，这叫做结束返回(end return)，在最后的语句前面的return关键字是可选的。
- 使用return关键字从闭包的任何地方返回。

这意味着下面两种对list的操作有相同的效果：

```
[1, 2, 3].collect{ it * 2 }  
[1, 2, 3].collect{ return it * 2 }
```

使用return进行提前返回，例如，仅仅对偶数进行处理：

```
[1, 2, 3].collect{  
    if (it%2 == 0) return it * 2
```

```
    return it  
}
```

在闭包中使用return关键字是简单易懂的，你几乎不会有任何误解，但有些事情需要了解。

**警告：**在闭包内使用return关键字与在闭包外使用是有区别的。

在闭包体外面，任何出现return的地方都会导致离开当前方法，当在闭包体内出现return语句时，这仅仅结束闭包的当前计算，这是更加有限的，例如，当使用List.each时，从闭包的提前返回不会引起从each方法的提前返回——闭包仍旧会在list的下一个元素出现时被调用。

在这本书的后面，我们将再一次遇到这个问题并且探讨更多的相关知识，13.1.8节概述了这些主题。

## 5.7 设计模式的支持

设计模式被广泛的用来提高设计质量，每一种设计模式代表了一种典型的OOP问题和相应的最佳的解决方案。我们来看看闭包在这方面的作用。

如果你以前从来没有看过设计模式，我们建议你看看经典的书《*Design Patterns: Elements of Reusable Object-Oriented Software*》，或者刚出不久的《*Head First Design Patterns*》，或者《*Refactoring to Patterns*》，或者使用你喜欢的搜索引擎搜索关键字“patterns repository”或“patterns catalog”。

尽管许多设计模式在各种语言中广泛的应用，但一些模式特别适合于应用在使用C++和JAVA之类语言编程时遇到的问题。它们经常涉及到实现新的抽象和新类来使原来的程序更具有灵活性和可维护性。通过groovy，在C++和JAVA中的一些约束不再适用。并且这些设计模式更简单活着可以直接通过语言的特性来实现，而不是通过引入新类，我们选择了两个例子来看看他们的不同：Visitor模式和Builder模式，正如你所看到的，闭包和动态类型是在groovy减轻模式使用的关键点。

## 5.7.1 Visitor 模式

当你希望在一个集合类（如tree或者list）中执行一些复杂的商业逻辑的时候Visitor模式特别有用，而不是为了一定的逻辑而修改这个聚合类，一个Visitor对象被引入，这个Visitor对象知道怎样遍历这个集合，也知道怎样为不同的类型执行商业逻辑函数，如果集合改变了或者逻辑函数随着时间而改变了，仅仅Visitor类是固定的。

列表5.10显示了在groovy中Visitor模式的使用是非常简单；集合遍历代码在Drawing类的accept方法中，但是商业函数（在我们的例子中是执行计算形状的面积）包含在两个闭包中，这些函数传递作为参数传递给适当的accept方法，在这里不需要特别的Visitor类。

**Listing 5.10 The Visitor pattern in Groovy**

```
class Drawing {  
    List shapes  
    def accept(Closure yield) { shapes.each{it.accept(yield)} }  
}  
class Shape {  
    def accept(Closure yield) { yield(this) }  
}  
class Square extends Shape {  
    def width  
    def area() { width**2 }  
}  
class Circle extends Shape {  
    def radius  
    def area() { Math.PI * radius**2 }  
}  
  
def picture = new Drawing(shapes:  
    [new Square(width:1), new Circle(radius:1)] )  
  
def total = 0  
picture.accept { total += it.area() }  
println "The shapes in this drawing cover an area of $total units."  
println 'The individual contributions are: '  
picture.accept { println it.class.name + ":" + it.area() }
```

## 5.7.2 Builder 模式

Builder模式提供了通过一个产品的成分逐步组装形成产品的处理。当使用这个模式的时候，一般需要创建一个Builder类，这个类包含了决定什么builder方法被调用和那一种顺序调用来保证适当的产品组装的逻辑。对于每一个产品来说，你必须为Builder类的每一个builder方法提供相应的处理逻辑；每一个builder方法一般返回产品个构成要素之一。

Builder模式中，基于Java的解决方法是不难的，但是java代码啰嗦并且没有突出产品的组装结构，基于这个原因，Builder模式在JAVA中使用的比较少；开发者使用无法组织的代码或者与别的代码混合重复的构建类型逻辑，非常遗憾，因为Builder模式是如此的强大。

Groovy的builder提供了一种解决方案，该方案使用嵌套的闭包来方便的确定复杂的產品。这样的规格是易读的，因为出现的代码直接映射到产品的结构，groovy有基于Builder模式的内建类库，这允许你容易的建立任意节点的结构，制作像HTML或者XML一样的标记，在swing中定义GUI或者别的控件工具箱，甚至在ANT中访问大量的函数，在第8章将看到更多的例子，在8.6节将介绍如何写自己的builder。

### 5.7.3 其他相关模式

在groovy中几乎所有的模式的实现都比在java中实现要容易些，这通常是由于groovy支持更加轻量的解决方案，这使得模式的必需代码大大减少——大多数时候是由于有闭包和动态类型，另外，当需要模式的时候，groovy通常表现的更简单明了。

在这本书的其他章节讨论了别的一些模式，如策略模式（在9.1.1和9.1.3），观察者模式（在13.2.3）和命令模式（在9.1.1），都可以通过使用闭包比通过实现新类更具有优势。如Adapter模式和装饰模式（在7.5.3）从动态类型和动态的方法查找来获取优势。我们也简单的讨论了模板方法模式（在5.2.2）、值对象模式（在3.3.2）、不完全的类库类（在7.5.3）、MVC（在8.5.6）、DTO和DAO模式（第10章）。闭包可以完全替换方法对象模式。

Groovy为你的程序使用模式提供了大量的支持，groovy的类库包含了许多模式实践，如Grails之类的高级别框架做得更远一些，Grails提供了一种建立在groovy类库和模式支持之上的框架。由于使用这类框架避免了你直接面对模式——你仅仅使用框架，这样在不需要了解细节的情况下你将自动的使用模式，尽管那样，理解一些我们接触到的模式有用的，这样你可以最大程度的发挥框架的优势。

## 5.8 结束语

你已经在了解了闭包，闭包是一个逻辑块，可以直接把闭包作为参数到处传递，从方法调用返回闭包，或者存储起来，在后面再使用。

闭包促进集中的资源处理，这样使你的代码更可靠，这不需要任何花费，事实上，代码库从结构重复方面得到了减轻，增强了表达性和维护性。

定义和使用闭包是出乎意料的简单，因为所有困难的任务（如引用的跟踪和方法回调到代理对象）都是透明的，如果你不关心范围规则，每一件事情都天生的自然有序，如果你想了解机制和执行例如调用代理对象的任务，你是可以做到的，当然，高级的用法需要更加注意，当从一个代理返回时你也需要小心一点，特别是在使用了别的语言的for循环或者相似的结果的情况。这比groovy的新手需要更小心，尽管行为是合乎常理的，当有疑惑的时候重新阅读5.6节。

对于许多开发者来说，闭包也许打开了做一些工作的新途径的一扇门，例如currying，刚开始出现的时候也许让人感到害怕，但也可以使用少量的代码实现强大的功能，加之闭包能使相似的设计模式简单甚至不再需要相应的模式。

虽然你对groovy的数据类型和闭包有了一个好的理解，你仍旧需要了解你的程序的执行流程，这通过控制结构来完美的实现，这就是下一章的主题。

# 第六章 groovy 的控制结构

在硬件层面，计算机系统使用简单的算术逻辑操作，例如，如果内存值为 0，那么跳到一个新位值，计算机执行的任何复杂的逻辑流程总是能表示为这种简单的操作，幸运的是，诸如 java 之类的语言提升了程序的抽象级别，这样我们能在更高级别的结构来表示逻辑流程，例如，循环遍历一个数组或者处理一个文件中的所有字符，直到文件结束。

在这章，我们探讨 groovy 的结构带给我们比 java 更简单、更形象的逻辑流程，在开始探讨之前，我们先解释一个哲学上的问题：什么是真相？

## 6.1 groovy 真相

为了理解groovy是怎样处理如if和while之类的控制结构的，你需要知道groovy是如何评估表达式的，这需要有一个Boolean结果，在这章我们碰到的许多控制结构依赖于Boolean测试的结果——一个表达式首先被评估，然后检查结果是否为true或者false。这样的结果后面紧跟相应的处理代码。在java中，这种考虑通常是没有实际意义的，因为java要求表达式的结果必须是专有boolean类型，groovy更自由一些，它通过忽略语言的简单性来简化代码，我们将检查groovy对于Boolean测试的规则并且给出一些建议以避免落入一些常见的陷阱中。

### 6.1.1 评估 Boolean 测试

Boolean测试的表达式可以是任何类型（不是void），这可以应用到任何对象，groovy决定一个表达式的结果是否为true或者false的规则显示在表6.1中，这是基于运行时类型的结果。规则按照给定的顺序应用，一旦一个规则匹配，结果就被完全确定。

表6.1 用来进行Boolean测试规则的顺序

| Runtime type                                 | Evaluation criterion required for truth          |
|--|--|
| Boolean                                      | Corresponding Boolean value is <code>true</code> |
| Matcher                                      | The matcher has a match                          |
| Collection                                   | The collection is non-empty                      |
| Map  | The map is non-empty                             |
| <code>String</code> , <code>GString</code>   | The string is non-empty                          |
| <code>Number</code> , <code>Character</code> | The value is nonzero                             |
| None of the above                            | The object reference is non-null                 |

列表6.1显示了规则的实际应用，使用Boolean的否定操作符“!”来判断表达式的结果应用评估为false。

#### Listing 6.1 Example Boolean test evaluations

```

assert true      | Boolean values
assert !false   | are trivial

assert ('a' =~ /./)    | Matchers must
assert !( 'a' =~ /b/)  | match

assert [1]       | Collections must
assert []        | be non-empty

assert ['a':1]   | Maps must be
assert ![:]      | non-empty

assert 'a'       | Strings must be
assert !!!       | non-empty

assert 1         |
assert 1.1       |
assert 1.2f      |
assert 1.3g      |
assert 2L         |
assert 3G         |
assert !0         | Numbers
                           (any type)
                           must be
                           nonzero

assert new Object() | Any other value
assert !null      | must be non-null

```

这些规则使测试表达式简单易读，无论如何，这些规则都有价值，正如你看到的那样。

## 6.1.2 将 Boolean 测试分配给变量

在我们进入本章的重要部分之前，我们给出了一个警告，正如java，groovy允许进行 Boolean测试的表达式被指定给变量，不像java，Boolean测试的类型没有被严格的限制为 boolean，这意味着你也许再现著名的历史问题，尽管更人性化，也就是说，“==”等号操作符与赋值操作符“=”是一个彻底不同的有效代码。Groovy无法避免这个经常出现的陷阱：在if语句中作为顶级表达式使用的时候。不管怎样，这仍旧出现的比较少。

列表6.2 显示了这个主题的典型变化

### Listing 6.2 What happens when == is mistyped as =

```
def x = 1
if (x == 2) {    ↪ ① Normal comparison
    assert false
}
*****
if (x = 2) {    ↪ ② Not allowed!
    println x      Compiler error!
}
*****
if ((x = 3)) {  ↪ ③ Assign and test in
    println x      nested expression
}
assert x == 3

def store = []
while (x = x - 1) {    ↪ ④ Deliberate assign
    store << x
}
assert store == [2, 1]

while (x = 1) {
    println x    ↪ ⑤ Ouch! This
    break        will print !!
```

在（1）的相等是比较是良好的并且在java中也是允许的，在（2），这个相等是比较是有意的，但是只有一个等号，这将导致groovy编译错误，因为在if语句中的顶级表达式不能是一个赋值表达式。

不管怎样，Boolean测试能够被嵌套在任何深度的表达式中，最简单的如（3）所示，这里有额外的圆括号将赋值语句包围起来，使得赋值语句成为了一个子表达式，这样这个语句符合了groovy的语言规范，在这里，值3分配给变量x，然后x将被用来进行测试，由于3被任务为true，因此值3将被打印出来，这个圆括号的使用欺骗了编译器，这样可以节省了赋值的一行代码，出现额外的圆括号是罕见的，因此这里做了一个警告读者的角色。

赋值表达式被限制不能作为if语句的顶级表达式，但是对于其他的控制结构语句（如while）没有这个限制，这是由于赋值和测试在一个表达式经常被用在while中，如（4），这种风格的典型用法是从转换器中接收处理标记或者从流中读取数据。虽然这是便利的，但这也容易让我们犯如（5）所示的错误，这里为x分配了一个值1，如果没有break语句，将形成一个死循环。

这种潜在的bug在别的语言也存在（如C和C++也存在这种问题），在你希望进行比较的时候在等号的左边放置常量，这样上表中最后一个while语句变为：

```
while (1 = x) {    ←  
    println x  
}
```

**Should  
be ==**

这样将产生一个错误，你不能给一个常量赋值，这样是安全的，遗憾的是，比较的两边都是变量也会失败，这也降低了可读性，不管人类语言的自然习惯，还是条件，大多数人们发现while(x==3)读起来比while(3==x)更加简单。虽然两者都不会引起混淆，后者更容易打断人们的思路。在这本书，我们优先考虑可读性，但是我们的情况与一般的开发稍微有点不同，你必须自己决定使用哪种习惯对你来说更好。

现在我们已经讨论了在groovy中哪种表达式被认为是true，哪种表达式被认为是false，现在我们可以开始看看控制结构了。

## 6.2 条件执行结构

我们首先看到的控制结构是条件执行，它们都评估一个Boolean测试并且根据评估的结果来选择执行下一步工作。对于java开发者来说，这没有什么新的东西，当然groovy增加了自己的一些东西，我们将讨论if语句、条件操作符、switch语句和断言。

### 6.2.1 普通的 if 语句

首先看看if语句和if/else语句，这两个结构在groovy的用法与在java中的用法完全一样，

除了增加的评估Boolean测试之外。

就像在java中一样， Boolean测试表达式必须在圆括号中，条件块一般是在花括号中，如果条件块的组语句只有一个语句，那么花括号是可选的。

“单行语句不需要括号”的特殊应用程序规则是按顺序使用else if，在这种情况下，代码的逻辑块通常采用缩进的方式，也就是说，所有的else if 行有相同的缩进，尽管它们的意义是嵌套的，对于groovy来说缩进是没有区别的，仅仅对代码美观有关。

列表6.3给出了一些例子，使用assert true来表明代码块被执行，并且使用assert false来表明代码块不被执行。

在列表中的显示应该不意外，虽然像字符串和列表这样的非Boolean表达式能被用作 Boolean测试看起来有点奇怪，不要担心，随着时间的过去这会变的很自然。

#### **Listing 6.3 The if statement in action**

```
if (true)      assert true
else          assert false

if (1) {
    assert true
} else {
    assert false
}

if ('non-empty') assert true
else if (['x']) assert false
else          assert false

if (0)          assert false
else if ([])    assert false
else          assert true
```

### **6.2.2 三元条件操作符**

Groovy也支持三元运算符 ?:，如列表6.4显示的那样，这个操作符根据评估的表达式的结果返回冒号左边的或者右边的结果，这依赖于问号前面的测试，如果第一个表达式评估的结果为true，那么中间的表达式被评估，否则最后的表达式被评估。就像在java中一样，最后的两个表达式不会同时被评估。

#### **Listing 6.4 The conditional operator**

```
def result = (1==1) ? 'ok' : 'failed'
assert result == 'ok'

result = 'some string' ? 10 : ['x']
assert result == 10
```

再一次注意Boolean测试（第一个表达式）可以是任何类型，也要注意由于在groovy中任何事物都是对象，中间的和最后的表达式可以是完全不同的对象。

关于对三元运算符的看法非常多，一些人觉得它十分方便并且经常使用，另外一些人却有相反的看法，你也许会在groovy中用的比较少，因为有一个特性使这种应用程序过时了，例如，GString（在3.4.2讨论）允许动态的创建字符串，而这种字符串在java中是通过三元运算符来创建的。

到目前为止，groovy相比java，switch语句有了许多重要的改变。

### 6.2.3 switch 语句

最近在一次乘火车中，我和朋友谈到groovy，提到非常酷的switch特性，朋友甚至没有让我开始说，他就挥着他的手说：“我从来不用switch！”我当初放弃了讨论，因为我没有了讨论的欲望；但后来想想，我也认为我在java中从来不用switch。

switch语句在java中非常受限制，你仅仅可以在switch中应用int类型，byte、char和short能自动的提升为int类型，由于有了这个限制，它的适用性被限制在低级别的任务或者通过一个类型编号进行一些派发，在面向对象语言中，类型编号的用法被认为是丑陋的。

switch结构的一般出现形式仅仅像在java中一样，处理逻辑失败然后进入下一个case，除非显示的退出。我们将看看在6.4节存在的可选项。

列表6.5显示了switch的一般形式。

#### Listing 6.5 General switch appearance is like Java or C

```
def a = 1
def log = ''
switch (a) {
    case 0 : log += '0'      | Fall through
    case 1 : log += '1'
    case 2 : log += '2' ; break
    default : log += 'default'
}
assert log == '12'
```

groovy是支持没有break而直接进入下一个case，这里有一些这样的例子真实的增强了代码的可读性，这样做通常是有害的（也同样也适用于java），作为一个一般惯例，在每一个

case结束的地方放置一个break语句是一种良好的风格。

### 通过分类器使用switch

你已经在3.5.5节看到了groovy通过switch进行分类，如果一个类型实现了isCase方法，那么这个类可以作为switch的分类器，换句话说，groovy的switch像这样：

```
switch (candidate) {  
    case classifier1 : handle1() ; break  
    case classifier2 : handle2() ; break  
    default : handleDefault()  
}
```

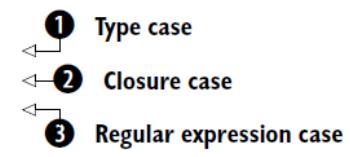
这大致上与下面代码相等：

```
if (classifier1.isCase(candidate)) handle1()  
else if (classifier2.isCase(candidate)) handle2()  
else handleDefault()
```

这样可以进行分类描述并且通过混合分类器来进行一些非传统的用法，不像java中的case，条件也许匹配多个分类器，这意味着case的顺序在groovy是很重要的，但在java中case的顺序是没有影响的。列表6.6给出了多类型分类器的例子，用来检查我们的数字10不是0、不在0..9之间、不在列表[8,9,11]中，不是Float类型，不是3的倍数，最后我们检查是否是2个字符。

#### Listing 6.6 Advanced switch and mixed classifiers

```
switch (10) {  
    case 0 : assert false ; break  
    case 0..9 : assert false ; break  
    case [8,9,11] : assert false ; break  
    case Float : assert false ; break  
    case {it%3 == 0} : assert false ; break  
    case ~/../ : assert true ; break  
    default : assert false ; break  
}
```



在(1)的新特性是通过类型进行归类，Float是java.lang.Class类型，并且GDK增强了Class，增加了isCase方法，这个方法使用isInstance方法来测试条件。

在闭包上存在isCase方法，在(2)的地方传递条件给闭包并且将闭包调用的结果强制

转换为Boolean。

最后的分类器（3）接受两位数，因为`~./`是一个模式（Pattern）并且在模式上的`isCase`方法将参数通过`toString`方法转换为字符串来进行测试。

为了杠杆化`switch`的作用，知道`isCase`方法的实现是必不可少的，groovy不可能提供一个完全的列表，因为你的任何自定义的类型都可以实现这个方法，表6.2列出了在GDK中已知的实现了`isCase`方法的类。

表6.2 实现了`isCase`方法的类

| Class                   | <code>a.isCase(b) implemented as</code>                  |
|-------------------------|--|
| <code>Object</code>     | <code>a.equals(b)</code>                                 |
| <code>Class</code>      | <code>a.isInstance(b)</code>                             |
| <code>Collection</code> | <code>a.contains(b)</code>                               |
| <code>Range</code>      | <code>a.contains(b)</code>                               |
| <code>Pattern</code>    | <code>a.matcher(b.toString()).matches()</code>           |
| <code>String</code>     | <code>(a==null &amp;&amp; b==null)    a.equals(b)</code> |
| <code>Closure</code>    | <code>a.call(b)</code>                                   |

注意：在集合的`grep`也使用`isCase`方法，如`Collection.grep(classifier)`返回符合分类器的一个集合列表。

Groovy的`switch`使用分类器进行处理是一个重大的进步，增加的代码的可读性，读者看到的是简单的分类器而不是混乱的、嵌套的`if`语句，再一次说明通过代码能实现做什么而不是怎么做。

在4.1.2节已经提到过，`switch`通过范围进行分类应用在商业规则上特别方便。这样的代码就像在读一个规范。

在你的代码中实现`isCase`，可以用来替换典型的`else if`结构。

**高级主题：**通过重载条件类型的*isCase*方法来支持不同类型的分类逻辑，如果你提供两个方法：*isCase(String candidate)*和*isCase(Integer candidate)*，那么*switch('1')*和*switch(1)*将有不同的行为。

接下来的主题是断言，猛地一看这个主题不是特别重要，不管怎样，虽然断言没有改变代码的商业能力，但他们可以让代码在产品环境中更强壮。此外，断言能做更好的一些事情：增强了开发者的信心，使开发者有信心在接下来的时间内得到提升和继续维护代码。

## 6.2.4 使用断言进行安全检查

在这本书中包含了几百个断言语句，你已经看到了许多，现在是了解断言的一些额外信息的时候了，我们将审视在断言失败时产生的有意义的错误信息，反映了使用这个关键字的原因，并且显示了如何使用断言进行内联单元测试，我们也将快速的比较groovy的解决方案和java的assert关键字及断言在单元测试中的用法。

### 生成失败相关信息

当断言失败的时候，断言生成堆栈和信息，放置下面这些代码：

```
a = 1  
assert a==2
```

在一个名称为FailingAssert.groovy的文件中，然后通过命令：

```
> groovy Failing-Assert.groovy
```

来运行这个文件，运行的结果会失败，并且生成如下信息：

```
Caught: java.lang.AssertionError: Expression: (a==2). Values: a  
= 1  
at FailingAssert.run(FailingAssert.groovy:2)  
153 / 213
```

如果满意该文档，请支持(招商银行北京分行双榆树支行 6225881008887381 账户名：吴翊)

```
at FailingAssert.main(FailingAssert.groovy)
```

你已经看到了失败，断言将在它出现的代码位置处打印出失败的异常信息和断言表达式中变量的值。堆栈跟踪信息显示了断言失败的位置和导致错误的方法调用顺序，堆栈信息最好是从底部向上阅读：

- 我们是在文件FailingAssert.groovy中；
- 在这个文件中，一个类FailingAssert被构建，这个类包含一个main方法；
- 通过main方法，我们调用了FailingAssert.run方法这个方法在FailingAssert.groovy的第二行（在运行脚本的时候，main和run方法在后台构建）；
- 在这行，断言失败。

这里有许多信息，这些信息大多数情况下足够用来定位和理解错误，但不是全部，我们来试试另外一个读取文件的例子，如果文件不存在或者不能读，那么断言失败：

```
input = new File('no such file')
assert input.exists()
assert input.canRead()
println input.text
```

产生的输出信息：

```
Caught: java.lang.AssertionError: Expression: input.exists()
```

```
...
```

这里没有提供充足的信息，丢失的信息是损坏的文件名称是什么，在这里，断言中可以包含一个跟踪信息：

```
input = new File('no such file')
assert input.exists() , "cannot find '$input.name'"
assert input.canRead() , "cannot read '$input.canonicalPath'"
println input.text
```

产生如下信息：

```
... cannot find 'no such file'. Expression: input.exists()
```

这正是我们需要的信息，不管怎样，这个特殊的例子也显示了断言有时不是必须使用的，因为在这个例子中我们能够容易的输出文件信息，而不是使用断言：

```
input = new File('no such file')
println input.text
```

下面的结果提供了足够的信息：

```
FileNotFoundException: no such file (The system cannot find the
file specified)
```

这里引申出了断言的最佳实践：

- 在写断言之前，先让你的代码失败，并且看看抛出的异常信息是否充足；
- 当写一个断言的时候，第一次先让断言失败，看看失败的信息是否充足，如果不充足，增加一些信息，然后再次让断言失败以检查信息现在是否充足；
- 如果你觉得需要断言来让你的代码更清晰或者保护你的代码，增加断言而不要管前面的规则；
- 如果你觉得需要一个信息来使你的断言的目的和意图更清晰，增加一个这样的信息，而不要管前面的规则。

## 在代码中内联单元测试

最后，断言的一个有争议的用法是在产品代码中增加适当的断言来进行内联单元测试，列表6.7显示了这个策略，这个策略使用正在表达式从一个URL中提取主机名称，模式（pattern）首先被构建然后在进行真实的执行之前应用一些断言，我们也实现了一个简单的方法assertHost来进行匹配分组的断言。

### Listing 6.7 Use assertions for inline unit tests

```
def host = /\//([a-zA-Z0-9-]+(\.[a-zA-Z0-9-])*?)(:|\/)/ ← Regular  
assertHost 'http://a.b.c:8080/bla', host, 'a.b.c'  
assertHost 'http://a.b.c/bla', host, 'a.b.c'  
assertHost 'http://127.0.0.1:8080/bla', host, '127.0.0.1'  
assertHost 'http://t-online.de/bla', host, 't-online.de'  
assertHost 'http://T-online.de/bla', host, 'T-online.de'  
  
def assertHost (candidate, regex, expected){  
    candidate.eachMatch(regex){assert it[1] == expected}  
}  
← Code to use the regular expression for useful work goes here
```

在没有断言和有断言的情况下阅读这个代码，价值是显然已见的，在审视例子的时候同时从断言中了解到我们的意图，传统上，这个例子应该在测试代码中或者在一个注释中，没有比这更好的了，但是经验证明注释经常过时并且读者不能真正的注意到代码是按说明工作的。外部测试也常常远离代码，一些测试被终止，由于项目计划的压力它们被注释，并且最终不再运行。

在代码中内联测试也许会害怕影响程序的性能，最佳答案是使用性能分析工具检查性能是否真的受到影响，在列表6.7的断言在几毫秒内运行完成并且不一般不是问题，当性能十分重要的时候，一个可能是放内联测试代码并且仅仅在类加载之后执行一次：放在一个静态初始化块中。

#### 断言的其他相关信息

自从JDK1.4之后，java有了一个assert关键字，它与groovy的断言语法上稍微有点不同（冒号代替了逗号来分隔Boolean测试）并且它可以启用或者禁用。Java的断言特性不是那么强大，因为它仅仅与java的Boolean测试一起工作，而groovy的断言完全接受groovy条件（参考6.1节）。

JDK文档使用较长的篇幅讨论了断言的禁用和对编译的影响，启动VM和相关结果，尽管这样做是好的并设计理论基础是清晰的，我们感觉在java中断言的禁用特性是最大的不稳定因素，你永远不能确保你的短语是否真正的执行到。

一些人宣称由于性能原因，断言在产品模式下应该禁用已经测试之后的代码中的断言，基于这个问题，Bertrand Meyer指出这就像离开水之后谈论游泳技巧一样。

在groovy中，断言总是被执行。

在单元测试中断言担当控制中心的角色，groovy包括了Junit，JUnit是一个流行的java测

试框架，JUnit的TestCases有许多专门的断言可用，groovy甚至增加了更多，这些断言的完全描述在14章进行，groovy提供的信息在断言失败的时候特别方便，因为她已经为测试人员编写了许多信息。

断言对个人编程风格有较大的影响，甚至对整个团队也是这样，不管是内联测试还是独立的测试，断言不但使你的代码更可靠，而且也使工作更容易理解。

这些就是条件执行结构，它们是任何逻辑分支的基础和循环的先决条件，这些特性使计算机为你做许多重复的工作，下一节将讨论while和for循环结构。

## 6.3 循环

到目前为止你看到了通过一次Boolean测试并且根据测试的结果更改执行路径的结构，另一方面，循环是重复的执行多次一个代码块，在groovy可用的循环式while和for，这两方面都将讨论。

### 6.3.1 while 循环

while结构与java的while相对应，唯一不同的地方你已经看到了——groovy的Boolean测试表达式更强大。概括起来相当简单，Boolean表达式被评估，如果为true，那么循环体被执行，Boolean表达式然后从新被评估，这样重复处理下去，仅仅在Boolean表达式评估为false的时候控制处理结束while循环，列表6.8显示了从一个列表中移除所有实体的例子，我们在第三章谈到过这个问题，在那里你注意到你不能为了这个目的使用for循环，第二个例子循环体只有一行，并且没有花括号。

#### Listing 6.8 Example while loops

```
def list = [1, 2, 3]
while (list) {
    list.remove(0)
}
assert list == []

while (list.size() < 3) list << list.size() + 1
assert list == [1, 2, 3]
```

这个代码中没有什么让人惊讶的地方，第一个循环式list作为Boolean测试稍微有点例

外。

注意，在groovy没有do{ }while(condition)或者repeat{ }until(condition)循环。

### 6.3.2 for 循环

估计for循环是使用得最广泛的循环类型了，在java中for循环比较难用，虽然使用过具有相似结构的开发语言的人们逐渐发现for循环容易使用，但是常常使用的for循环确没有好好地设计，虽然传统的for循环语句是强大的，但它很少用来表述像集合这样的数据结构，groovy具有这个简单性，这可能是groovy和java之间的最大不同吧。

Groovy允许这样的for循环：

```
for (variable in iterable) { body }
```

这里variable的类型是可选的，groovy的for循环遍历iterable，常常用到的iterable有range/collections/map/array/iterator/enumeration，事实上，任何对象都可以是一个iterable，groovy为对象的遍历应用相同的逻辑，这在第八章进行描述。

如果循环体只有一句，那么花括号是可选的，列表6.9显示了一些例子。

**Listing 6.9 Multiple for loop examples**

```
def store = ''  
for (String i in 'a'..'c') store += i  
assert store == 'abc'  
  
store = ''  
for (i in [1, 2, 3]) {  
    store += i  
}  
assert store == '123'  
  
def myString = 'Equivalent to Java'  
store = ''  
for (i in 0 ..< myString.size()) {  
    store += myString[i]  
}  
assert store == myString  
  
store = ''  
for (i in myString) {  
    store += i  
}  
assert store == myString
```

① Typed, over string range, no braces  
② Untyped, over list as collection, braces  
③ Untyped, over half-exclusive IntRange, braces  
④ Untyped, over string as collection, braces

例子（1）为i使用了现实的类型并且由于循环体只有一句话，所以没有花括号，这个循环用来生成一个字符串。

当使用集合类型时通常出现的for循环如（2）所示，再次感谢自动装箱，这种方式也适用于数组。

一个半排除的整数范围的循环如（3）显示，它与java相当的结构如下：

```
for (int i=0; i < exclusiveUpperBound; i++) { // Java !
    //这里放置使用i的代码
}
```

这是一个经典的for循环，在当前版本的groovy中不支持，以后的版本也许会支持（译者注：在最新的1.6版本中已经支持了）。

例子（4）使例子（3）更加清晰，例子（3）不是groovy典型字符串处理风格，更多时候groovy把字符串作为字符的集合进行处理。

在一般对象上使用for循环将在9.1.3节进行描述，这节提供了一些非常强大的例子。

你可以使用for循环一行一行的输出一个文件的内容：

```
def file = new File('myFileName.txt')
for (line in file) println line
```

或者打印匹配正则表达式的所有数字：

```
def matcher = '12xy3' =~ /\d/
for (match in matcher) println match
```

如果容器对象为null，那么将没有结果输出：

```
for (x in null) println 'This will not be printed!'
```

如果groovy通过任何方法都不能确定一个容器对象是iterable，那么最好的解决方法是仅

仅使用容器对象本身进行一次遍历：

```
for (x in new Object()) println "Printed once for object $x"
```

对象遍历使groovy的for循环成为一个优雅的控制结构，一个有效的对应方式是以闭包为参数调用对象上的遍历方法，比如使用Collection的each方法。

这两者的主要不同之处是for循环不是一个闭包！这意味着方法体是代码块：

```
for (x in 0..9) { println x }
```

但是这样用的循环体就是一个闭包了：

```
(0..9).each { println it }
```

即使它们看起来相似，但它们在结构上时很不相同的。

一个闭包是一个对象，并且有你在第五章看到的所有特性，它可以在不同的地方就行构建然后传递给each方法。

for循环的循环体，相应的，它直接在它出现的地方生成字节码，没有特别的范围规则。

当从循环体中退出来时这种区别十分重要，在下节将显示为什么会这样。

## 6.4 退出代码块和方法

尽管在像阅读操作指南一样阅读代码时不进行跳转时很美好的，经常把控制的当前代码块或者方法围起来是十分重要的，就像在java中，groovy也可以像期望的那样，有序的使用return、break和continue语句，即使是在发生异常的紧急情况下。我们来近距离看看。

### 6.4.1 正常终止：return/break/continue

return、break和continue的正常逻辑与java中的用法类似。一个不同之处是在方法或者闭

包中的最后表达式return关键字是可选的，如果return被忽略，那么返回值是最后表达式的返回结果，显示申明的返回类型为void的方法不返回一个值，而闭包总是返回一个值（如果闭包最后的表达式没有返回值，那么闭包返回null）。

列表6.10显示了当前循环式怎样通过continue快速结束和通过break结束整个循环的，像在java中一样，这是一个可选的标签。

#### Listing 6.10 Simple break and continue

```
def a = 1
while (true) {    ↪ Do forever
    a++
    break    ↪ Forever is
}                ↪ over now
assert a == 2

for (i in 0..10) {
    if (i==0) continue
    a++
    if (i > 0) break    ↪ Proceed
}                ↪ with i
assert a==3
    ↪ Premature
    ↪ loop end
```

在典型的编程风格中，break和continue的用法是必须进行慎重考虑的，不管怎样，这在流程控制方面是有用的，相似的，在方法中从多个点返回时不太妥当的，但有些人发现尽早的返回一个结果，这样能大大的提高代码的可阅读性，我们鼓励你弄明白最容易读懂的是什么，并且讨论无论谁都可以读懂你的代码——对于别人来说，一致性是很重要的。

最后注意返回的处理，记住在遍历方法如each中使用闭包的时候return与控制结构while和for中的return有不同的意思，正如5.6节解释的那样。

### 6.4.2 异常： throw/try-catch-finally

Groovy中的异常处理完全与java相同并且有相同的逻辑，正像java那样，你可以指定完整的try-catch-finally块顺序，或者仅仅使用try-catch，或者仅仅try-finally，注意，不像其他的控制结构是不管否包含处理语句花括号都是必须的，groovy和java不同的地方时在方法申明的时候是否声明异常是可选的，即使是检查的异常，列表6.11显示了常用的行为。

### Listing 6.11 Throw, try, catch, and finally

```
def myMethod() {
    throw new IllegalArgumentException()
}

def log = []
try {
    myMethod()
} catch (Exception e) {
    log << e.toString()
} finally {
    log << 'finally'
}
assert log.size() == 2
```

虽然在groovy的其他地方类型声明是可选的，但在catch表达式中类型是必须的。

当检查的异常没有声明的时候，groovy不能生成编译时或者运行时警告，当一个检查的异常没有处理的时候，这个异常会像RuntimeException那样传播。

在第11章覆盖了集成java和groovy的更详细的信息，在这里讨论异常处理是有价值的，当在java中使用groovy类的时候，你得当心——groovy方法没有声明抛出任何检查异常（除非你显式的增加了声明），即使在运行时抛出了一个检查的异常，不幸的是，如果你在java中捕捉一个检查的异常，但是它发现没有这样一个异常抛出的时候，编译器会聪明的抱怨，如果你运行到这里并且需要显示的捕捉在groovy代码中产生的检查的异常，你也许需要在groovy代码中增加一个throws声明，这样编译器才不会抱怨。

## 6.5 总结

这章我们浏览了groovy的控制结构：条件执行代码、循环、提前退出代码块和方法，没有多少意外的地方因为每一样都与java相似，浓缩了一些groovy的元素。仅仅在结构上不同的是for循环，异常处理与java非常相似，除了不要求对检查的异常进行声明之外。

Groovy的Boolean测试处理在条件执行和循环中式一致的，我们验证了当Boolean测试为true的时候java和groovy的差别，理解这个很重要，因为符合语言习惯的groovy经常被用来测试不是简单的Boolean表达式。

switch关键字被用作一个一般的分类器，带来了新的面向对象的条件处理，通过isCase方法来控制怎样进入条件，虽然在面向对象语言中使用switch是泄气的，groovy给switch注入了新的活力。

总体上，断言有自己的位置，它们属于每一个开发人员都应该了解的工具。

通过你学习到的知识，你可以编写任何类型的程序了，可以肯定的是，你有更高的目标并且想精通面向对象编程，下一章将教你如何做。

## 第七章 groovy 风格的动态面向对象

关于脚本语言有一些普遍的错误认识，因为一个脚本语言也许支持宽松的类型和提供一些令人惊讶的语法，对于计算机迷来说感觉就像一个玩具，而不是适用于面向对象编程的语言。这个声誉来源于早期的 shell 脚本或者是 Perl，它们缺少别的 OO 特性，有时导致粗放的代码管理，频繁的代码复制和难以发现的隐藏 bug，没有相应的语言帮助以让后来人员看明白。

随着时间的过去，脚本语言已经发生了戏剧性的变化，Perl 已经增加了对面向对象的支持，Python 也扩充了面向对象的支持，并且近来的 Ruby（与 java 和 C++ 相比，有重大的生产力的提升）也是一个完全的动态面向对象脚本语言。

Groovy 密切跟踪 Ruby 提供的动态面向对象特性，不但增强了 java 的脚本能力，而且也提供了新的 OO 特性，你已经看到了 groovy 在 java 使用非对象的专有类型的地方提供了引用类型，引入了范围和闭包，并且为对象的集合增加了许多简化符号，但这些增强仅仅表面现象，如果这就是 groovy 的全部，它将与 java 上的其他语法糖没有区别，groovy 的立足点是它的一组动态特性。

在这章中，我们将从熟悉的类、对象、构造方法、引用等等开始，偶尔有一点不同，但在这章结束的时候，我们将到达一个新的领域，在运行时改变对象的能力、拦截方法调用、非常非常多，欢迎来到 groovy 的世界。

### 7.1 定义类和脚本

在 groovy 中类的定义几乎与 java 一样；类的声明使用 class 关键字，在类中可以包含多个属性、构造方法、初始化器和方法。一般方法和构造方法也许使用了他们自己的本地变量作为方法实现代码的一部分。脚本是不同的——提供了额外的灵活性但是也有些限制，它们也许包含代码、变量定义、方法定义和类定义，我们将描述所有的这些成员的声明和前面没

有看到的操作符。

### 7.1.1 定义属性和本地变量

在最简单的情况下，一个变量是内存中一块地址的名称，正如在 java 中那样，groovy 有本地变量、它的范围被限制在所在的方法体内，属性是与类或者这些类的实例相联系的，属性和本地变量的声明有许多相同的地方，因此我们将它们放在一起讨论。

#### 变量的声明

属性和本地变量在使用之前必须进行声明（除了脚本之外，我们在后面将讨论到），这有助于实施作用域规则和防止程序员偶然的拼写错误。声明总是包含指定一个名称，也许还包含一个可选的类型，修饰符和分配初始化值，一旦声明之后，变量就可以通过相应的名称进行引用。

脚本允许使用没有声明的变量，在这种情况下变量被假定从脚本的 binding 属性获取，如果在 binding 中没有发现相应的变量，那么把变量增加到 binding 中，binding 是一个数据存储器，它能把变量在脚本调用者和脚本之间进行传递，11.3.2 节有这个机制的更详细的描述。

Groovy 使用 java 的修饰符——关键字为 private、protected 和 public 用来修饰可访问范围；final 用来限制对变量再次赋值；static 用来标示为类变量，一个非静态的属性为一个实例变量，这些修饰符的意思与 java 中一样。

缺省属性范围在 groovy 有特殊的意思，在没有指定范围修饰符的属性声明的时候，groovy 会根据需要生成相应的访问方法（getter 方法和 setter 方法），你将在 7.4 节关于 groovyBeans 了解到更多有关这方面的信息。

定义变量的类型是可选的，不管怎样，标示符在声明的时候不必独一无二，当没有类型和修饰符时，必须使用 def 来作为替换，实际上用 def 来表明属性或者变量是没有类型的（尽管在内部将被声明为 Object 类型）。

列表 7.1 描述了属性和变量的一般声明方式进行选择性赋值，通过逗号分隔一次性声明多个属性或者变量。

### **Listing 7.1 Variable declaration examples**

```
class SomeClass {  
  
    public      fieldWithModifier  
    String      typedField  
    def         untypedField  
    protected   field1, field2, field3  
    private     assignedField = new Date()  
  
    static      classField  
  
    public static final String CONSTA = 'a', CONSTB = 'b'  
  
    def someMethod(){  
        def localUntypedMethodVar = 1  
        int localTypedMethodVar = 1  
        def localVarWithoutAssignment, andAnotherOne  
    }  
}  
  
def localvar = 1  
boundvar1 = 1  
  
def someMethod(){  
    localMethodVar = 1  
    boundvar2 = 1  
}
```

指定了类型的引用必须符合给定的类型——也就是说，你不能将一个数字对象分配给一个字符串类型的变量，反过来也是一样，在第三章你已经看到 groovy 提供了自动装箱和在需要的时候自动进行类型转换，别的情况下进行不恰当的类型分配将在运行时导致一个 ClassCastException，如列表 7.2 所示。

### **Listing 7.2 Variable declaration examples**

```
final static String PI = 3.14  
assert PI.class.name == 'java.lang.String'  
assert PI.length() == 4  
new GroovyTestCase().shouldFail(ClassCastException.class){  
    Float areaOfCircleRadiusOne = PI  
}
```

就像前面讨论的那样，变量能够通过名称被访问，就像 java 中一样，但 groovy 提供了一些更有趣的可能性。

#### **引用属性和取消对属性的引用**

除了可以通过 obj.fieldName 来引用属性之外，也可以通过下标操作符来引用属性，如

列表 7.3 所示，这样你可以通过动态的名称来访问属性。

### Listing 7.3 Referencing fields with the subscript operator

```
class Counter {  
    public count = 0  
}  
  
def counter = new Counter()  
  
counter.count = 1  
assert counter.count == 1  
  
def fieldName = 'count'  
counter[fieldName] = 2  
assert counter['count'] == 2
```

通过动态的途径来访问属性是动态执行的一部分，是我们这章的课程的组成部分。

如果你用 groovy 的数据类型描述做过工作，你下一个问题可能是：“我怎样重写下标操作符？”，当然可以重写，并且你将是扩展而不是重写一般的属性访问方式（通过 `setter` 和 `getter` 进行），甚至你可以做得更好并且扩展属性访问操作符！

列表 7.4 显示了如何做这个工作，扩展 `set` 和 `get`，提供方法

```
Object get (String name)  
  
void set (String name, Object value)
```

这里没有限制你可以在方法中做的工作；`get` 可以返回一个虚假的值，假装事实上你已经获取到了要求的属性，在列表 7.4，不管访问的那个属性，总是返回相同的值，`set` 方法用来记录写的次数。

#### Listing 7.4 Extending the general field-access mechanism

```
class PretendFieldCounter {  
    public count = 0  
  
    Object get (String name) {  
        return 'pretend value'  
    }  
    void set (String name, Object value) {  
        count++  
    }  
}  
  
def pretender = new PretendFieldCounter()  
  
assert pretender.isNoField == 'pretend value'  
assert pretender.count == 0  
  
pretender.isNoFieldEither = 'just to increase counter'  
  
assert pretender.count == 1
```

通过 `count` 属性，你能看到如果请求的属性出现了 `get/set` 方法好像没有被使用，实际上在我们的这个例子中确实是这样，在 7.4 节，你将看到这个效果的全套规则。

一般来说，重写 `get` 方法意味着重写了 `dot-fieldName` 操作符，重写 `set` 方法意味着重写了 `field assignment` 操作符。

语句 `x.z.y=something` 这个语句执行的结果是什么？这与 `getX().getY().setZ(somethine)` 等价

将属性也连接为一个特性的主题，我们将在 7.4 节解释，在那里我们将讨论新的 `obj.@fieldName` 语法。

### 7.1.2 方法和参数

方法的声明与你看到的变量的声明有相同的概念：通常能使用 `java` 修饰符；返回类型的声明是可选的；如果没有修饰符或者返回类型，那么使用 `def` 关键字来填空白，当使用 `def` 关键字的时候，方法的返回类型被认为是没有类型（尽管没有返回类型，这时与 `void` 方法等价），在这种情况下，groovy 将方法的返回类型定义为 `java.lang.Object`，缺省的方法访问范围为 `public`。

列表 7.5 通过例子的自我描述方式显示了方法声明的典型用法

**Listing 7.5 Declaring methods**

```
class SomeClass {  
    static void main(args) {    ↪ ① Implicit  
        def some = new SomeClass()  
        some.publicVoidMethod()  
        assert 'hi' == some.publicUntypedMethod()  
        assert 'ho' == some.publicTypedMethod()  
        combinedMethod()      ↪ Call static method  
    }                      of current class  
  
    void publicVoidMethod(){  
    }  
  
    def publicUntypedMethod(){  
        return 'hi'  
    }  
    String publicTypedMethod(){  
        return 'ho'  
    }  
  
    protected static final void combinedMethod(){  
    }  
}
```

在（1）处，`main` 方法有些有趣的改变，首先，`public` 修饰符能够忽略，因为方法声明的默认修饰符就是 `public`；其次，为了使 `main` 方法的启动执行 `args` 通常的类型为 `String[]`。感谢 `groovy` 的方法分派，不管如何工作，尽管 `args` 目前的类型隐式为 `java.lang.Object`。第三，方法分派时由于返回类型没有使用，我们能够进一步忽略 `void` 的声明。

因此，这是 `java` 的声明：

```
public static void main (String[] args)
```

在 `groovy` 中浓缩为：

```
static main(args)
```

*注意：如果在 `main` 方法的返回语句中声明了返回类型的话，将通过 `java` 编译器的检查，在 `groovy` 中，返回语句是可选的，因此编译器偶然检查到没有返回类型是不可能的。*

`main` 方法的例子说明了显式声明参数的类型是可选的，当声明类型被忽略的时候，`groovy` 使用 `Object` 作为类型，可以顺序的使用多个参数，通过逗号进行分隔，列表 7.6 显示了显式的声明参数类型和忽略参数类型是可以混合在一起使用的。

#### Listing 7.6 Declaring parameter lists

```
class SomeClass {  
    static void main (args){  
        assert 'untyped' == method(1)  
        assert 'typed' == method('whatever')  
        assert 'two args'== method(1,2)  
    }  
    static method(arg) {  
        return 'untyped'  
    }  
    static method(String arg){  
        return 'typed'  
    }  
    static method(arg1, Number arg2){  
        return 'two args'  
    }  
}
```

在迄今为止的例子中，所有方法调用都与方法参数的位置有关，这意味着做每一个参数已经根据它参数列表中的位置被决定了，这是容易理解的，并且便于简化你已经看到的案例，但对于复杂的脚本有一些痛苦的缺点：

- 你必须记住真实的参数的顺序，当参数列表比较长时这增加了难度；
- 对于非传统的脚本使用，如果根据不同的信息调用方法很有意义，那么需要构建许多不同的方法来处理这些差异，这很快就变得笨重起来并且导致方法的难以维护，特别是当有些参数是可选的时候，如果许多可选的参数具有相同的类型，这会变得特别困难。幸运的是，`groovy` 使用 `map` 作为一个命名参数列表来拯救了这种情况。

**注意：**每当谈论到命名的参数的时候，我们的意思是 `map` 的 `key` 在方法中被用作参数被调用，从一个程序员的视角看，这看起来就像原生的命名参数的支持，但其实不是这样的，这种技巧是必须的，因为 `JVM` 不支持把参数的名称保存在字节码中。

列表 7.7 介绍了 `groovy` 支持的位置参数和命名参数的方法定义和调用，可变长度的参数列表，可选的参数和其默认值，例子提供了四种组合，每一种都突出了不同的调用方法途径。

### Listing 7.7 Advanced parameter usages

```
class Summer {  
    def sumWithDefaults(a, b, c=0){ ← ① Explicit arguments  
        return a + b + c  
    } ← and a default value  
    def sumWithList(List args){ ← ② Define arguments  
        return args.inject(0){sum,i -> sum += i} ← as a list  
    }  
    def sumWithOptionals(a, b, Object[] optionals){ ← ③ Optional arguments  
        return a + b + sumWithList(optionals.toList()) ← as an array  
    }  
    def sumNamed(Map args){ ← ④ Define arguments  
        ['a','b','c'].each{args.get(it,0)} ← as a map  
        return args.a + args.b + args.c  
    }  
}  
  
def summer = new Summer()  
  
assert 2 == summer.sumWithDefaults(1,1)  
assert 3 == summer.sumWithDefaults(1,1,1)  
  
assert 2 == summer.sumWithList([1,1])  
assert 3 == summer.sumWithList([1,1,1])  
assert 2 == summer.sumWithOptionals(1,1)  
assert 3 == summer.sumWithOptionals(1,1,1)  
  
assert 2 == summer.sumNamed(a:1, b:1)  
assert 3 == summer.sumNamed(a:1, b:1, c:1)  
assert 1 == summer.sumNamed(c:1)
```

这四种方案都都有他们的优点和缺点，在（1），`sumWithDefaults`，我们为方法调用有了大多数明显期望的参数声明，这满足了简单脚本的需要——能够增加 2 个或者 3 个参数在一起，但是这种方式受到已经声明的参数的限制。

如（2）这样在 groovy 中使用 `list` 是非常容易的，因为在方法调用中，参数仅仅放置在方括号中，我们也可以支持任意长度的参数列表，不管怎样，方法参数的意义不能明显的表示出来，因此，这个方式最适合于所有参数都有相同的意思的情况，在这里对参数列表进行了累加，参考 4.2.3 节来了解 `List.inject` 方法的详细信息。

在（3）的 `sumWithOptionals` 方法可以通过 2 个或者更多的参数进行调用，声明了这样一个方法，定义最后一个参数为一个数组，groovy 动态方法分派将剩余的参数打包进这个数组中。

命名参数的支持通过（4）的方式进行支持，在使用他们之前设置所有错过的值为一个默认值是一个好的实践。这也较好的显示出 `key` 将被用在方法体中，因为这明显不是通过方法声明得到的。

当设计你的方法的时候，你必须从可选方案中选择一个，你也许希望把你的代码通过文

档的形式记录到项目中，或者还包括了 groovy 的代码风格。

注意：实现可变长度的参数列表的第二种方式是，你可以覆盖 groovy 的分派方法 `invokeMethod(name,params[])`，这是每个 `GroovyObject` 对象都提供的方法，在 7.6.2 将学习到这方面的更多知识。

### 高级命名

当在对象引用上调用一个方法的时候，我们通常使用下面的格式：

```
objectReferenc.methodName()
```

这种格式受到 java 方法命名的格式限制；例如，它们不能包含特殊的如减号(-)或者点号(.)这样的字符，不管怎样，如果你把方法名称放在引号中，那么 groovy 允许你在方法名称中使用这样的字符：

```
objectReferenc.'my.method-Name'()
```

这个特性的目的是支持把调用方法的名称变成函数的一部分，通常情况下不直接使用这个特性，但是 groovy 的其他部分将使用到这个特性，在第八章和第十章将看到这个特性的使用。

说明：有一个字符串，通常是使用 `Gstring`，如果这样 `obj."${var}"()` 会怎么样，是的，这样是可以的，并且 `Gstring` 将被用来确定调用的方法的名称。

这些就是类成员的基本原理，在结束这个主题之前，通过引用对象引用到对象的一个成员的时候有一个便利的操作符是应该介绍的。

### 7.1.3 安全的引用符号 (?)

当一个引用没有指向任何特定对象的时候，它的值为 `null`，当在一个 `null` 引用上调用一

个方法或者访问一个属性的时候，一个 `NullPointerException` (NPE) 将被抛出，这保护了代码在在不明确的先决条件下的工作。但这也容易形成“尽力而为”来检查引用的有效性。

列表 7.8 显示了保护代码不发生 NPE 的不同途径，作为一个例子，我们希望访问在 `map` 中的实体中深度嵌套的对象，这样的路径表达式——通过点来连接互联的引用对象是不能保证不会发生 NPE 的，我们可以使用 `if` 进行显示的检查或者使用 `try-catch` 机制，groovy 提供了额外的 `(?)` 操作符来进行安全的引用，当操作符之前是一个 `null` 引用的时候，当前表达式的评估被终止，并且 `null` 被返回。

#### Listing 7.8 Protecting from `NullPointerExceptions` using the `?.` operator

```
def map = [a:[b:[c:1]]]

assert map.a.b.c == 1

if (map && map.a && map.a.x) {
    assert map.a.x.c == null
}

try {
    assert map.a.x.c == null
} catch (NullPointerException npe) {
}

assert map?.a?.x?.c == null
```

The diagram illustrates three methods for protecting against `NullPointerException` using the `?.` operator:

- Protect with if: short-circuit evaluation**: An arrow points from the first `if` statement to the `assert` statement inside it.
- Protect with try/catch**: An arrow points from the `try` block to the `assert` statement inside it.
- Safe dereferencing**: An arrow points from the final `assert` statement to the `?.` operator used in the path `map?.a?.x?.c`.

通过对比可以发现，在（3）处使用安全引用操作符是最优雅的和最具有表达性的解决方案。

注意在（1）的地方比 java 等价解决方案更紧凑，它需要三个空检查，因为表达式是从左边向右边进行评估的，如果第一个操作数评估为 `false`，那么`&&`操作符终止评估，这是大家已经知道的短路运算符。

在（2）的地方有点啰嗦并且不允许只对路径表达式进行保护控制，这也是滥用了异常处理机制，异常设计的初衷不是应用于这种情况的，引用不为 `null` 时容易避免和验证的。导致一个异常然后再捕捉它等价为在汽车上安装一个大的保险杆然后撞向建筑物。

Groovy 使用安全引用操作符来组合路径并且降低了复杂性，本质上，衡量标准是代码容易理解并且能简单的检查错误。

## 7.1.4 构造器（构造方法）

对象通过类的构造方法进行实例化，如果没有明确的给出一个构造方法，那么编译使用一个无参数的默认构造方法，这是在 java 中就出现了的，由于在 groovy 中，因此增加了一些额外可用的特性也不意外。

在 7.12 节我们比较了命名参数和位置参数的有点，可选参数也是需要的，相同的参数的方法调用也适用与构造方法，groovy 也提供了相同的便利机制，我们将首先看看位置参数的构造方法，然后再看看命名参数的构造方法。

### 位置参数

到目前，我们仅仅使用了隐式构造方法，列表 7.9 介绍了第一个显示构造方法，注意这很像其他所有的方法，构造方法默认是 public 的，我们能够通过三种途径调用构造方法：常用的 java 方式，使用 as 关键字进行强制造型和使用隐式造型。

**Listing 7.9 Calling constructors with positional parameters**

```
class VendorWithCtor {  
    String name, product  
  
    VendorWithCtor(name, product) { ← Constructor  
        this.name = name  
        this.product = product  
    } ← definition  
}  
  
def first = new VendorWithCtor('Canoo', 'ULC') ← Normal  
def second = ['Canoo', 'ULC'] as VendorWithCtor ← ① Coercion with as  
VendorWithCtor third = ['Canoo', 'ULC'] ← ② Coercion in assignment
```

在（1）和（2）的地方也许会让你感到很意外，当 groovy 看到需要将一个列表造型成别的类型的时候，它试图使用列表的所有元素作为参数来调用该类型的构造方法，按照列表的顺序进行。这在使用 as 关键字进行造型或者通过赋值来进行静态类型引用的时候是需要的。我们后面讨论这些叫做隐式构建。

### 命名参数

命名参数在构造方法中是方便的，这在创建一个不可修改的类实例，而这个类有些可选

参数的时候很有用，使用位置参数很快就变得很笨拙，因为你必须考虑到构造方法所有的可选参数。

作为一个例子，假如列表 7.9 的 VendorWithCtor 应该不可变并且 name 和 product 能够是可选的，我们需要 4 个构造方法：一个没有参数，一个有 name 参数，一个有 product 参数，最后一个有 name 和 product 两个参数，事情变得更糟糕的是，我们没有唯一一个只有一个参数的构造方法，由于我们不能区别是否设置 name 还是 product 属性（他们两个都是字符串类型），我们需要一个额外的参数来进行区别，或者我们需要强类型参数（译者注：在声明参数的时候指明参数的类型）。

不要恐慌：groovy 专门提供了命名参数的支持来救援。

列表 7.10 显示了在 Vendor 类的一个简化版本中使用命名参数。它依赖隐式的默认构造方法，不是十分容易吗？

#### Listing 7.10 Calling constructors with named parameters

```
class Vendor {  
    String name, product  
}  
  
new Vendor()  
new Vendor(name: 'Canoo')  
new Vendor(product:'ULC')  
new Vendor(name: 'Canoo', product:'ULC')  
  
def vendor = new Vendor(name: 'Canoo')  
assert 'Canoo' == vendor.name
```

在列表 7.10 的例子中介绍了怎样为你的构造方法使用灵活的命名参数，如果你不想要这种灵活性并且想锁定所有的参数，仅仅需要明确定义你需要的构造方法；隐式的命名构造方法将不再有效。

回到我们这节是如何开始的，默认无参数构造方法调用（new Vendor）的出现是不重要的，虽然它实际上看起来与 java 等价，命名参数的默认无参数构造方法被调用时没有提供任何参数。

### 隐式构造方法

最后，通过简单的提供构造方法的参数列表来隐式的调用构造方法，这意味着已经显式的调用了 Dimension(width,height) 构造方法，例如，你能够这样使用：

```
java.awt.Dimension area  
area = [200, 100]  
assert area.width == 200  
assert area.height == 100
```

当然，groovy 必须知道调用什么构造方法，因此，通过类相应的构造方法来静态的分配类型引用是可用的，这对抽象类和接口是不工作的。

隐式构造方法经常用来作为 builder，正如你将在 8.5.7 中看到的 SwingBuilder 那样。

这就是常用类的成员，这是我们能建立的稳定的基础，但是我们仍然没有在顶峰；我们有四个级别要走，我们通过怎样组织类和脚本的主题来到达高级面向对象特性的级别，它们的基层是命名的 GroovyBean 和涉及到的关于对象的简单 OO 信息，在这个级别，我们能用 groovy 强大的特性来做事，最后，我们将访问到最高级别，这就是在 groovy 中的元数据编程——制作完全动态的环境，通过一种非常奇怪的途径来进行相应的方法调用和属性引用。

## 7.2 组织类和脚本

在 2.4 节，你已经看到 groovy 类在字节码级别就是 java 类，所以，groovy 对象在内存中就是 java 对象，在源代码级别，groovy 类和对象处理几乎是 java 语法的一个超集，嵌套类是一个例外，当前 groovy 语法还不支持嵌套类，并且在数组的定义上有微小的改变，我们将查看类和源文件的组织，及两者的关系，我们也将考虑到在 groovy 中包的使用和类型的别名，groovy 也能不神秘的从它的类路径中加载类。

### 7.2.1 文件到类的关系

在文件和类声明的关系不像在 java 中那样固定的，groovy 文件根据下面的规则能够包含多个公共类的声明：

- 如果一个 groovy 文件不包括类声明，那么它被作为一个脚本处理；也就是说，它透明的包装为 Script 类型的类，自动生成的类的名称与源文件名称相同（不包括扩展名），文件只能够的内容被包装在一个 run 方法中，并且增加了一个 main 方法用来容易

的启动脚本。

- 如果 groovy 文件中只包含一个类声明，并且这个类的名称与文件名相同（不包括扩展名），那么这里有与 java 中一样的一对一的关系。
- 一个 groovy 文件也许包含多个不同访问范围的类声明，这里没有强制规则必须使类的名称与文件名一样，groovyc 编译器完美的为所有在这个文件中声明的类创建 \*.class 文件，如果你希望直接调用你的脚本，例如在命令行或者 IDE 中使用 groovy，那么在你的文件中的第一个类中应该有一个 main 方法。
- 一个 groovy 文件也许混合了类的声明和脚本代码，在这种情况下，脚本代码将变成一个主类被执行，因此不要声明一个与源文件同名的类。

当没有进行显式编译的时候，groovy 通过匹配相应名称的\*.groovy 源文件来查找类，在这点上，名称变得十分重要，groovy 仅仅根据类名称是否匹配源文件名称来查找类，当这样的一个文件被找到的时候，在这个文件中声明的所有类都将被转换，并且 groovy 在后面的时间都将能找到这些类。

列表 7.11 显示一个简单的脚本，这个脚本中包括了两个简单的类，Vendor 和 Address，目前他们没有方法，只有公共的属性。

#### **Listing 7.11 Multiple class declarations in one file**

```
class Vendor {  
    public String name  
    public String product  
    public Address address = new Address()  
}  
class Address {  
    public String street, town, state  
    public int zip  
}  
  
def canoo = new Vendor()  
canoo.name = 'Canoo Engineering AG'  
canoo.product = 'UltraLightClient (ULC)'  
canoo.address.street = 'Kirschgartenstr. 7'  
canoo.address.zip = 4051  
canoo.address.town = 'Basel'  
canoo.address.state = 'Switzerland'  
  
assert canoo.dump() =~ '/ULC/'  
assert canoo.address.dump() =~ '/Basel/'
```

Vendor 和 Address 是简单数据结构类，它们大概等价于 C 的结构或者 Pascal 的记录，我们很快将探索定义类的更优雅的途径。

列表 7.11 介绍了 groovy 支持的源文件，与类的映射规则的便利约定，这在我们早期讨论过的。这样的约定允许在相同的源文件中声明 main 方法类或者当前脚本的助手类。与 java 相比，这样允许你使用嵌套的类供本地类使用，这样不会干扰你的公共类的命名空间，也不会使在代码库中查找源代码文件更加困难，尽管这不是真正的一样，这种约定对于 groovy 开发人员来说有相似的行为。

## 7.2.2 在包中组织类

Groovy 延续了 java 的包结构中组织文件的方式，包结构用来在文件系统中找到相应的类文件。

由于\*.groovy 源文件不需要编译成\*.class 文件，因此在查找类的时候也需要查找 \*.groovy 文件，当这样做的时候，groovy 使用了相同的策略：编译器查找一个 groovy 类 Vendor 的时候，在 business 包对应的文件系统中查找 business/Vendor.groovy 文件。

在列表 7.2 中，我们在脚本代码有两个独立的类 Vendor 和 Address，如列表 7.11 显示的那样，并且把它们移动到了 business 包中。

### 类路径

查找类的时候是从某个地方开始的，java 的类路径就是基于这个目的来使用的，类路径是查找\*.class 文件的开始点的一组列表。Groovy 查找\*.groovy 时重用了类路径。

当我们查找一个给定的类的时候，如果 groovy 同时找到了一个\*.class 和\*.groovy 文件，它使用最后修改的那个文件，也就是说，如果源文件在上一次编译之后做了修改，groovy 将重新编译源文件到\*.class 文件。

### 包

事实上就像在 java 中一样，groovy 的类必须在定义之前指定它们所在的包，当没有声明包的时候，groovy 使用默认包。

列表 7.12 显示了文件 business/Vendor.groovy，这个文件中的第一行有一个包声明语句。

### **Listing 7.12 Vendor and Address classes moved to the business package**

```
package business

class Vendor {
    public String name
    public String product
    public Address address = new Address()
}

class Address {
    public String street, town, state
    public int zip
}
```

为了引用在 business 包中的 Vendor 类，你可以使用 business.Vendor 或者使用 import 语句导入类。

#### **导入 (import)**

Groovy 跟随 java 的导入语句，在声明类之前进行导入。

**注意：**请记住这不像别的一些脚本语言，*import* 在字面上没有包含任何类或者文件，它仅仅是通知编译器如何来解析类的引用。

列表 7.13 显示了 import 语句的用法，使用.\*标记通知编译器在 business 包中试图解析所有不知道的类引用。

### **Listing 7.13 Using import to access Vendor in the business package**

```
import business.*

def canoo = new Vendor()
canoo.name      = 'Canoo Engineering AG'
canoo.product   = 'UltraLightClient (ULC)'

assert canoo.dump() =~ /ULC/
```

**注意：**默认情况下，groovy 导入了 6 个包和两个类，这使得每个 groovy 源代码程序看起来都包含了下面的初始化语句：

```
import java.lang.*
```

```
import java.util.*  
import java.io.*  
import java.net.*  
import groovy.lang.*  
import groovy.util.*  
import java.math.BigInteger  
import java.math.BigDecimal
```

## 类型别名

import 语句有另外一个美妙的转变：可以与 as 关键字一起使用，这可以用来作为类型的别名，而一般的 import 语句运行使用类的名称来应用来，通过类型别名你可以使用任何你喜欢的名称来引用一个类，这个特性解决了类名称冲突的问题并且支持本地修改或者 bug 修复为一个第三方类库。

考虑下面的库类：

```
package thirdparty  
  
class MathLib {  
  
    Integer twice(Integer value) {  
        return value * 3 // intentionally wrong!  
    }  
  
    Integer half(Integer value) {  
        return value / 2  
    }  
}
```

注意那个明显的错误(尽管一般情况下这不是一个错误而仅仅是一个本地想要的改变)，假如现在我们有些已经存在的代码使用了这个类库：

```
assert 10 == new MathLib().twice(5)
```

我们能够使用类型别名来重命名原始的类，然后使用继承来修复它，不能修改原来已经使用的代码，如列表 7.14 显示：

**Listing 7.14 Using import as for local library modifications**

```
import thirdparty.MathLib as OrigMathLib

class MathLib extends OrigMathLib {
    Integer twice(Integer value) {
        return value * 2
    }
}

// nothing changes below here
def mathlib = new MathLib()
assert 10 == mathlib.twice(5)
assert 2 == mathlib.half(5)
```

Usage code for library  
remains unchanged

Invoke fixed  
method

Invoke original method

现在，假设我们需要使用下面的额外的数学库：

```
package thirdparty2

class MathLib {

    Integer increment(Integer value) {
        return value + 1
    }
}
```

虽然它在不同的包中，但是有一个与前面的类型相同的名称，如果不使用别名，那么我们在代码中必须有一个类使用全限定名称或者两个都使用全限定名称，通过别名，我们可以通过一种优雅的方式来避免这种情况，并且在我们的程序中通过为第三方类使用更好的别名也改善了交流，如列表 7.15 显示。

### Listing 7.15 Using import as for avoiding name clashes

```
import thirdparty.MathLib as TwiceHalfMathLib
import thirdparty2.MathLib as IncMathLib

def math1 = new TwiceHalfMathLib()
def math2 = new IncMathLib()

assert 3 == math1.half(math2.increment(5))
```

例如，如果我们后来发现一个 `math` 包 `increment` 和 `twice/half` 函数都应该有，我们能引用这个新库并且保留我们更有意义的名称。

在你的程序中应该考虑使用别名，甚至是在使用简单的内建类型的时候，例如，如果你开发一个冒险游戏，你也许需要将 `Map` 起一个名为 `SatchelContents` 的别名，这样不需要定义一个单独的 `SatchelContents` 的强类型，但同样提高了代码的易读性。

### 7.2.3 类路径更长远的考虑

`groovy` 工作的重要部分之一是查找`*.class` 和`*.groovy`，不幸的是，这可能是问题的根源所在。

如果 你 安 装 的 J2SDK 包 含 了 文 档 ， 你 能 在`%JAVA_HOME%/docs/tooldocs/windows/classpath.html` 中发现类路径的解释说明，在 windows 下，或者对于 Linux 和 Solaris 相似的目录下。文档说的每一件事情都可以等价应用到 `groovy` 中。

表 7.1 也许在你查找可能类路径问题的时候为你提供了一定的参考。

表 7.1 类路径的组成

| Origin        | Definition                             | Purpose and use   |
|---------------|--|---|
| JDK/JRE       | %JAVA_HOME%/lib<br>%JAVA_HOME%/lib/ext | Bootclasspath for the Java Runtime Environment and its extensions |
| OS setting    | CLASSPATH variable                     | Provides general default settings                                 |
| Command shell | CLASSPATH variable                     | Provides more specialized settings                                |
| Java          | -cp<br>--classpath option              | Settings per runtime invocation                                   |
| Groovy        | %GROOVY_HOME%/lib                      | The Groovy Runtime Environment                                    |
| Groovy        | -cp                                    | Settings per groovy execution call                                |
| Groovy        | .                                      | Groovy classpath defaults to the current directory                |

Groovy 使用在%GROOVY\_HOME%/conf 下一个特殊配置文件来定义它自己的配置文件，查看 groovy-starter.conf 文件显示的下列行（不包含其他部分）：

```
# Load required libraries
load ${groovy.home}/lib/*.jar

# load user specific libraries
# load ${user.home}/.groovy/lib/*
```

通过移除最后一行前面的#号标志来启用一个酷特性，在你的个人主目录 user.home 中，你能使用一个子目录.groovy/lib(注意 groovy 前面有一个点)来存储任何\*.class 或者\*.jar 文件，这样无论你什么时候使用 groovy 工作都可以访问到。

如果你不知道自己的个人主目录在什么地方，打开命令行并且运行：

```
groovy -e "println System.properties.'user.home'"
groovy -e "println System.properties.'user.home'"
```

偶然的，你默认就在这个目录下。

第 11 章在内嵌 groovy 在别的环境中（它们有自己的类加载结构）如一个应用程序服务器的时候将有更多高级的类路径相关问题需要考虑。

现在你能够通过不同方式使用构造方法来实例化一个类，类也许属于某个包，并且你已经看到了如何通过导入命令来寻找到这些类，这完全是对象基础的探索，下一步将探索更多 OO 的高级特性，我们将在接下来的章节进行讨论。

## 7.3 高级 OO 特性

在开始拥抱我们已经讨论了的 OO 特性的基础用法的 groovy 类库之前，我们先短暂的停下来看看别的 OO 概念，这些概念一旦在你进入 groovy 世界之后将改变，我们将涉及到继承和接口，这些与 java 是相似的，multimethods 将让你体验到后面将要见到的动态面向对象。

### 7.3.1 使用继承

你已经看到了在定义类的时候如何增加自己的类属性、方法和构造器，继承让你隐式的增加来自父类的类属性和方法，这种机制在一定范围内是有用的，我们不描述它的优势，并且潜在的可能过度使用的时候给你一些告诫。我们简单的让你知道 java 中所有的继承特性（包括抽象类）在 groovy 中都是可用的并且也在 Groovy 和 java 两者中都工作（几乎完美的）。

Groovy 类能够继承 groovy 和 java 的类和接口，java 类也可以继承 groovy 类和接口，在这种情况下，你需要编译你的 java 和 groovy 类，参考第 11.4.2 节了解更详细的信息，当你选择方法进行调用的时候，你仅仅需要知道 groovy 比 java 更动态，这个特性将在 7.3.3 节进行讨论。

### 7.3.2 使用接口

Java 编程频繁的提倡使用接口，通过这种风格写的代码通过使用的接口引用依赖类，依赖类能在以后进行安全的改变而不用修改原来的程序，如果开发者意外的改变到的类不是相应的接口实例，这种错误在编译的时候将会被发现，groovy 完全支持 java 的接口机制。

一些观点认为单独的接口不是足够的强壮，并且基于合约的设计来获取安全对象替换而

不改变你的类库，这样处理更重要。明确的使用抽象方法和继承变得正如使用接口一样重要，groovy 支持 java 的抽象方法，自动启用 assert 语句，并且内建访问测试方法，这意味着也完美的支持方式。

尽管这样，别的一些观点认为动态类型是最佳途径，这样在不降低安全（应该覆盖任何情况的测试）的情况下有更少的类型和更少的脚手架代码。幸运的是 groovy 很好的支持这种风格，为了给你的每天编写的代码增加乐趣，考虑一下如何在 java 和 groovy 中建立一个插件。

在 java 中，一般情况下你为一个插件机制写一个接口，然后每一个插件有一个这个接口的实现类，在 groovy 中，动态类型允许你更容易的创建和使用符合需要的实现。你可能仅仅需要两个类作为开发两个插件实现部分，一般情况下，你会有更少的脚手架代码和更少的类。

**技巧：**如果你决定使用接口，groovy 提供了使他们更动态的途径，如果你有一个接口 `MyInterface`（该接口只有一个方法）和一个闭包 `myClosure`，你能使用 `as` 关键字强制将闭包造型为 `MyInterface`，相似的，如果你有一个接口有几个方法，你能创建一个 `map`（方法名称为 `key`、对应的闭包为 `value`）并且强制造型 `map` 为你的接口类型，参考 groovy 的 wiki 了解更详细的信息。

总之，如果你是从 java 转过来的，你也许使用强接口类型的风格，当使用 groovy 的时候，你不需要强迫使用任何一种风格，在许多情况下，通过使用动态类型你可以最小化类型的数量；并且如果你真的需要，也可以完全使用接口。

### 7.3.3 Multimethods

记住 groovy 的方法查找是方法参数的动态类型化，而 java 是静态类型的，这个 groovy 的特性叫做复合方法 **Multimethods**。

列表 7.16 显示了两个方法，这两个方法都叫做 `oracle`，唯一不同的仅仅是参数的类型，它们通过相同的静态类型和不同的动态类型调用了 2 次。

### Listing 7.16 Multimethods: method lookup relies on dynamic types

```
def oracle(Object o) { return 'object' }
def oracle(String o) { return 'string' }

Object x = 1
Object y = 'foo'

assert 'object' == oracle(x)
assert 'string' == oracle(y)
```

This would return  
object in Java

参数 x 是 Object 静态类型和 Integer 动态类型，参数 y 是 Object 静态类型和 String 动态类型。

两个参数有相同的静态类型，两次都采用与 java 等价的将方法分派给 oracle(Object)，由于 groovy 通过动态类型进行方法分派，oracle(String) 的专门实现被用在第二中情况。

通过这种能力，你能通过适当的覆盖行为更好的避免重复的代码。考虑在列表 7.17 中的 equals 方法的实现，覆盖的 Object 的缺省 equals 方法仅仅用在参数类型为 Equalizer 的情况。

### Listing 7.17 Multimethods to selectively override equals

```
class Equalizer {
    boolean equals(Equalizer e){
        return true
    }
}

Object same = new Equalizer()
Object other = new Object()

assert new Equalizer().equals( same )
assert ! new Equalizer().equals( other )
```

当一个类型为 Equalizer 的对象被传递给 equals 方法的时候，专门的实现被使用。当传递其他类型的对象时，在 Object.equals 的缺省实现方法被调用，这个方法仅仅检查对象的引用。

调用的 equals 方法能完全不知道差异，从一个调用者的角度看，它就像 equals(Equalizer) 覆盖了 equals(Object)，这在 java 中是办不到的，相反，一个 java 程序员也许像这样写：

```
public class Equalizer { // Java
    public boolean equals(Object obj)
```

```

{
    if (obj == null) return false;

    if (!(obj instanceof Equalizer)) return false;

    Equalizer w = (Equalizer) obj;

    return true; // custom logic here
}
}

```

这是不适当的，因为覆盖 `equals` 方法的逻辑对于在 `java` 中每一个自定义类型都会重复，这是使用静态类型并且使用动态类型解决方法的另外一个例子。

注意：无论在什么地方使用 `JAVA` 的静态类型 `Object`，这样代码实际上丢失了静态类型的长处，你不可避免的使用造型来找回类型，折中的方法是编译时类型安全。这就是为什么 `JAVA` 类型概念叫做弱静态类型：丢失了静态类型的优势，但也没有得到动态类型语言的优势如复合方法（multimethods）。

通过对比，在 `groovy` 中通过参数的动态类型进行方法的派发是单一的和一致的实现。

## 7.4 使用 GroovyBean 工作

`JavaBean` 规范在 `JAVA 1.1` 引入，这个规范用来定义应用在 `java` 中的一个轻量级的和一般软件组件模型。这个组件模型建立在命名约定和通过 `JAVA` 类暴露类的属性给别的类和工具的 `API` 上。这大大的增强了定义和使用可重用组件和开发组件工具的可能性。

第一个工具主要是可视化向导，如接受和维护属性的可视化组件的可视化构件。随着时间的过去，`JavaBean` 概念已经被广泛的使用并且扩展了使用范围，包括服务器端组件（在 `JSP` 中），事务行为和持久化（`EJB`），对象关系映射框架（`ORM`）和别的数不清的框架和工具。

`Groovy` 通过特定的语言支持使得 `JavaBean` 的使用更加简单，这种使用包含了三个方面：创建 `JavaBean` 类的特殊的 `Groovy` 语法；不管 `JavaBean` 是在 `groovy` 中还是在 `java` 中声明的，`groovy` 提供了容易的访问 `Bean` 的机制；对 `JavaBean` 的事件处理提供支持。这节将看到 `groovy` 在语言级别支持的每个部分也覆盖了通过 `Expando` 类提供的类库支持。

### 7.4.1 声明 Bean

JavaBean 是符合一定命名约定的普通的 java 类，例如，为了使在 JavaBean 中的一个字符串属性 myProp 可用，bean 类必须有声明为 String getMyProp 和 void setMyProp(String value) 的公共方法，JavaBean 规范也强烈建议 bean 应该是可序列化的 (serializable)，这样它们可以被持久化并且可以通过参数构造器容易的进行构建，典型的 java 实现是下面这样的：

```
// Java

public class MyBean implements java.io.Serializable {

    private String myprop;

    public String getMyprop() {

        return myprop;
    }

    public void setMyprop(String value) {

        myprop = value;
    }
}
```

Groovy 中等价为：

```
class MyBean implements Serializable {

    String myprop
}
```

代码量有非常大的区别的，groovy 中的一行替换了 java 中的七行，这不但少了很多代码量，而且自身也是一个说明文档。在 groovy 中，访问属性是十分容易的：所有的类属性都被声明为缺省可访问范围，三个相关的信息——一个属性有两个访问方法，通过一次声明来保持一致，改变属性的类型或者名称仅仅要求在一个位置改变代码。

注意: groovy 的旧版本使用@Property 语句来标明属性, 这样是难看的, 被移除了并且通过支持属性为默认访问范围。

在底层, groovy 提供了与 java 代码等价的属性访问方法, 但是你不用输入这些代码, 此外, 只有在类中不存在这些方法的时候才自动生成它们, 这样允许你通过自定义逻辑或者访问范围来覆盖标准的访问方法。Groovy 也提供了一个专有的后台属性 (也生成 java 等价代码)。注意, JavaBean 规范只关注了访问方法的可用, 甚至没有要求一个后台属性; 但有一个直观的简单实现方法——因此 groovy 也这样做。

注意:groovy 构造了访问方法并且把它们增加到了字节码中, 这样保证了MyBean 在java 中的使用, groovy 的 MyBean 类是一个验证过的 JavaBean。

列表 7.8 显示了属性的声明选项, 包括可选的类型声明和初始值, 规则与类属性是等价的 (参考 7.2.1 节)

#### Listing 7.18 Declaring properties in GroovyBeans

```
class MyBean implements Serializable {
    def untyped
    String typed
    def item1, item2
    def assigned = 'default value'
}

def bean = new MyBean()
assert 'default value' == bean.getAssigned()
bean.setUntyped('some value')
assert 'some value' == bean.getUntyped()
bean = new MyBean(typed:'another value')
assert 'another value' == bean.getTyped()
```

属性有时可读的, 有时可写的, 这依赖于类是否有相应的 getter 方法或者 setter 方法。Groovy 的属性既是可读的也是可写的, 但是如果你有特殊的需要你总是可以自己写, 当在属性声明的时候使用了 final 关键字的时候, 这个属性仅仅是可读的 (没有创建 setter 方法并且后台属性为 final)。

写 groovyBean 是简单优雅的兼容 JavaBean 支持的解决方案, 可以为特定的需求指定选项。

## 7.4.2 使用 bean 工作

在 java 世界完全采用 JavaBean 概念已经是一个通用编程风格，就是通过访问方法来限制对属性的直接访问（在访问方法中要避免复杂费时的操作），这些访问方法由 groovy 自动生成，如果你有复杂的、额外的逻辑关联到属性，你总是可以覆盖相关的 getter 或者 setter，但是你最好是写一个单独的业务逻辑方法来处理你的高级逻辑。

### 访问方法

即使类没有完全符合 JavaBean 标准，你通常可以假设有访问方法可以调用，这些方法没有大的性能损失或者别的副作用。一个访问方法的规格几乎就像直接的属性访问（不打断一致访问原则）。

Groovy 根据表 7.2 显示的方法调用隐射，在语言级别支持这种风格。

**Table 7.2 Groovy accessor method to property mappings**

| Java                                | Groovy                             |
|-------------------------------------|------------------------------------|
| <code>getPropertynname()</code>     | <code>propertynname</code>         |
| <code>setPropertyname(value)</code> | <code>propertynname = value</code> |

不管是一个 groovy 对象还是 POJO 对象，都可以使用这个隐射进行工作，并且这个 bean 工作也适合于其他类，列表 7.19 显示了 Bean 风格和派生属性的组合。

**Listing 7.19 Calling accessors the Groovy way**

```
class MrBean {  
    String firstname, lastname  
    ↗ Groovy style properties  
    String getName(){  
        ↗ 1 Getter for derived property  
        return "$firstname $lastname"  
    }  
}  
  
def bean = new MrBean(firstname: 'Rowan')  
bean.lastname = 'Atkinson'  
  
assert 'Rowan Atkinson' == bean.name ↗ 3 Call getter
```

② Call setter

③ Call getter

Generic constructor

注意在(2)和(3)是groovy风格的属性访问，看起来就像直接的属性访问。而在(1)解释了没有属性，但是有些派生值，从调用者角度看，这样的访问时真正一致的。

由于属性访问和访问方法快捷方式有相同的语法，你可以选择喜欢的规则进行处理。

**规则：**当属性和相应的访问方法对调用者都是可用的时候，属性引用被解析为对访问方法的调用，如果只存在一种可用，这是可选的。

看起来简单易懂，并且有许多例子，不管怎样，还是有些需要考虑的地方，正如在下节你将看到的那样。

### 使用@进行属性访问

在我们结束属性主题之前，我们有些例子要探讨：列表7.20。在列表中介绍了你怎样提供自己的访问方法和绕开访问方法的机制，在需要的时候你能够使用@操作符直接访问属性。

**Listing 7.20 Advanced accessors with Groovy**

```
class DoublerBean {  
    public value   ← Visible field  
  
    void setValue(value){  
        this.value = value   ← ① Inner field access  
    }  
  
    def getValue(){  
        value * 2   ← ② Inner field access  
    }  
}  
  
def bean = new DoublerBean(value: 100)  
assert 200 == bean.value   ← ③ Property access  
assert 100 == bean.@value   ← Outer field access
```

我们从熟悉的地方入手：在(3)的地方bean.value调用的是getValue并且其结果是返回双精度数，但是getValue的计算结果为value \* 2，如(2)所示，如果value在这里bean快捷方式为getValue，我们将有一个递归的结果。

相似的情况发生在(1)处，赋值语句this.value = 在bean中将被解释为this.setValue，这也让我们掉入一个无限循环中，因此，groovy安排了下列规则。

**规则:** 在类内部, 引用 `fieldName` 或者 `this.fieldName` 将被解释为直接属性访问, 而不是 `bean` 风格的属性访问 (通过访问方法进行); 在类的外部, 可以使用 `reference.@fieldName` 语法直接访问类属性。

需要提到的是, 这些规则能产生符合逻辑但是让人意外的反常的用例, 如从一个静态上下文中使用@或者使用这样的语句 `def x = this; x.@fieldname`, 等等。我们将不进行更详细的讨论, 因为这样的设计是不合理的, 决定是否通过类属性, `bean` 方式的方式, 或者通过显式的访问方法来暴露一个状态, 不要混合使用, 保持访问的一致性。

### Bean 风格的事件处理

除了属性之外, JavaBean 也能作为事件源供给事件监听器, 一个事件监听器是一个有一个回调方法用来通知这个监听器的对象, 一个事件对象用来作为参数传递给回调方法。

JDK 有不同类型的事件监听器, 一个简单在按钮上的事件监听器是 `ActionListener`, 这有一个方法 `actionPerformed (ActionEvent)` 在按钮点击的时候被调用, 更复杂的例子是 `VetoableChangeListener`, 它允许在它的方法 `vetoableChange (PropertyChangeEvent)` 中抛出一个 `PropertyVetoException` 来回滚改变的 `bean` 属性, 别的用法是多方面的, 在这里不可能提供一个完整的列表。

Groovy 通过一种简单但是强大的方式来支持事件监听器, 假设你需要创建一个 Swing JButton, 按钮的标签为 “Push me!”, 当按钮被点击的时候将标签的内容打印在控制台, 一个 Java 的实现时使用匿名内部类:

```
// Java

final JButton button = new JButton("Push me!");

button.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent event) {
        System.out.println(button.getText());
    }
});
```

开发人员需要知道各自注册的监听器和事件类型 (或者接口) 及回调方法。

一个 groovy 开发人员仅仅附上一个闭包给按钮，就像它是按钮的一个属性，属性名称为相应的回调方法名称：

```
button = new JButton('Push me!')  
  
button.actionPerformed = { event ->  
  
    println button.text  
  
}
```

`event` 参数被增加时为了知道如何在需要的时候获取它，在这个例子中，它已经被忽略，因为在闭包中没有使用它。

*注意:* groovy 使用 bean 内省机制来决定是否属性赋值引用为 bean 支持的监听器的回调方法，如果是这样，在通知的时候调用闭包，一个 `ClosureListener` 被透明的增加。一个 `ClosureListener` 是要求的接口的代理实现。

事件处理是 JavaBean 标准构想，不管怎样，在做任何事件处理之前你不需要在 Bean 中以某种方式声明你的对象。相反的：一旦你的对象支持这种事件处理风格，它就是一个 Bean。

尽管 groovy 增加了能将闭包作为监听器注册的能力，java 风格的 bean 事件处理完整的保留，这意味着你仍然可以使用 java 方法来获取所有注册的事件监听器，增加更多的监听器，或者在不需要的时候移除监听器。

### 7.4.3 为任何对象使用 bean 方法

Groovy 不识别 bean 和其他类型的对象的区别，它仅仅依赖相应的 getter 和 setter 方法是否可用。

列表 7.21 显示了怎样使用 `getProperties` 方法和 `properties` 属性来获取到 bean 属性的 map (`key` 为属性名称，`value` 为属性值) 列表。你可以在任何对象上进行这样的操作。

### Listing 7.21 GDK methods for bean properties

```
class SomeClass {  
    def      someProperty  
    public   someField  
    private  somePrivateField  
}  
  
def obj = new SomeClass()  
def store = []  
obj.properties.each { property ->  
    store += property.key  
    store += property.value  
}  
assert store.contains('someProperty')  
assert store.contains('someField')      == false  
assert store.contains('somePrivateField') == false  
assert store.contains('class')  
assert store.contains('metaClass')  
  
assert obj.properties.size() == 3
```

除了显式声明的属性之外，你也看到了 class 和 metaClass 的引用，这两个属性是 groovy 产生的。

这样做的原因将在 9.1 节进行解释。

## 7.4.4 属性、访问方法、隐射和扩展 (Fields/accessors/maps/Expando)

在 groovy 的代码中，你经常发现如 object.name 这样的表达式，当 groovy 解释这样的引用的时候到底发生了什么：

- 如果 object 引用到一个 map，object.name 引用到存储在 map 中的，key 值为 name 对应的 value 值；
- 否则，如果 name 是 object 的一个属性，那么这个属性被引用（优先采用访问方法，如 7.4.2 节所示）；
- 每一个 groovy 对象有机会实现自己的 getProperty(name) 和 setProperty(name,value) 方法，当这样做之后，这些实现用来控制属性的访问。例如，在 map 中使用这个机制来暴露 key 列表为属性列表；
- 如 7.1.1 节显示，属性的访问能够通过提供 object.get(name) 方法进行拦截，这是 groovy 运行时被困扰的最后的对策：它仅仅用在没有适当的 javaBean 属性可用和

getProperty 方法没有实现的情况。

当属性包含特殊字符,而这些特殊字符是无效标识符的时候,这可以通过字符串来应用,如 object.'my-name',你也可以使用 Gstring: def name = 'my-name'; object."\$name",正如你在 7.1.1 节所见,并且我们将在 9.1.1 节进行进一步的讨论,在 Object 上也有一个 getAt 的实现,这样你可以通过 object[name] 的方式来访问对象的属性。

对于 groovy 对象解析依据决定了对象的动态状态和行为,groovy 带来了这样一个特性: Expando,一个 Expando 能够作为 bean 的扩展,尽管仅仅用于 groovy 中而没有直接用在 java 中,它支持 groovy 风格的属性访问和一些扩展,列表 7.22 显示了一个 Expando 对象如何通过赋值方式扩展属性,这与 map 相似,不同的时候将闭包赋值给一个属性,这样在访问属性的时候闭包被执行,闭包可以接受任意参数,在这个例子中, boxer 的 fightBack 按给定的倍数返回。

#### Listing 7.22 Expando

```
def boxer = new Expando()  
  
assert null == boxer.takeThis  
  
boxer.takeThis = 'ouch!'  
  
assert 'ouch!' == boxer.takeThis  
  
boxer.fightBack = {times -> return this.takeThis * times }  
  
assert 'ouch!ouch!ouch!' == boxer.fightBack(3)
```

在某种程度上,Expando 将闭包分配给属性的行为和通过属性访问存储的闭包有点像动态的增加方法给对象。

在避免写脏数据结构如类的时候,Map 和 Expando 是极端的解决方案,因为它们不要求编写任何额外的类,在 groovy 中,访问 map 的 key 或者 Expando 的属性与访问 JavaBean 属性没有什么不同。但是这有代价的: Expando 不能在 java 中用作 bean 并且不支持任何类型。

## 7.5 使用强劲的特性

这节介绍 groovy 在语言级别支持的三个强劲的特性：Gpath、Spread 操作符和 use 关键字。

我们先从 GPath 开始，一个 GPath 是 groovy 代码的一个强劲对象导航的结构，名称的选择与 XPath 相似，XPath 是一个用来描述 XML（和等价物）文档的标准，正如 XPath，GPath 的目标是用在表达式：明确的，紧凑易读的表达式。

GPath 几乎全部建立在你已经看到的概念上：属性访问，短方法调用及增加到 Collection 的 GDK 方法。他们仅仅引入了一个新的操作符：(\*) 操作符，让我们通过正确的途径来开始使用它。

### 7.5.1 使用 GPath 来查询对象

我们通过反射 API 探讨 groovy，目标是获取一个当前对象的所有 getter 方法的简短列表，我们将一步一步的做，因此请打开 groovyConsole 并且跟随向前，你将试图获取到当前对象的信息，输入

```
this
```

并且运行脚本（输入 Ctrl-Enter），在输出窗口，你将看到像这样的一些信息

```
Script1@e7e8eb
```

这是当前对象的字符串表示结果，为了获取到当前对象的 class 的相关信息，你能够使用 this.getClass，但是在 groovy 中你能输入

```
this.class
```

显示的结果为（在你运行该脚本之后）

```
class Script2
```

通过 `getMethods` 来获取到 `class` 对象暴露的方法列表，因此输入：

```
this.class.methods
```

结果输出了一个大的方法对象描述列表，这里有太多的信息，你仅仅对方法的名称感兴趣，每一个方法对象有 `getName` 方法，因此这样调用

```
this.class.methods.name
```

并且得到一个方法名称的列表，作为一个字符串对象的列表返回，你可以容易的使用学到的字符串，正则表达式，列表等相关知识进行工作。由于你仅仅对 `getter` 方法感兴趣，并且想对这些方法排序，输入

```
this.class.methods.name.grep(~/get.*/).sort()
```

你将得到这样的结果

```
["getBinding", "getClass", "getMetaClass", "getProperty"]
```

例如这样的表达式叫做 GPath，一个特殊的事情是你能够在一个方法对象的列表上调用 `name` 属性并且接收到一个字符串对象的列表。

后台规则是：

```
list.property
```

等价于

```
list.collect{ item -> item?.property }
```

这是在列表中进行属性访问的特殊用法的缩写，一般用法为：

```
list*.member
```

这里`*`叫做展开点操作符，并且`member`能够作为属性被访问，一个属性访问，或者一个方法调用，展开点操作符在方法应用到列表中的所有元素上的时候是需要的，而不是应用到列表本身上，它的等价物为：

```
list.collect{ item -> item?.member }
```

为了在实战中明白 GPath，我们看一个适当接近真实情况的例子，假如你正在处理一组发票，每一组引用到固定产品，一个产品有一个价格和名称。

一张发票看起来像表 7.3

**Table 7.3 Sample invoice**

| Name          | Price in \$ | Count | Total |
|---------------|-------------|-------|-------|
| ULC           | 1499        | 5     | 7495  |
| Visual Editor | 499         | 1     | 499   |

图 7.1 使用 UML 类图描述了相应的软件模型，Invoice 类包含多个指向产品的 LineItem。

列表 7.23 是设计的 groovy 实现，定义类为 groovyBean，通过这个结构构建简单的发票，并且最后使用 GPath 表达式通过多种途径查询对象图。

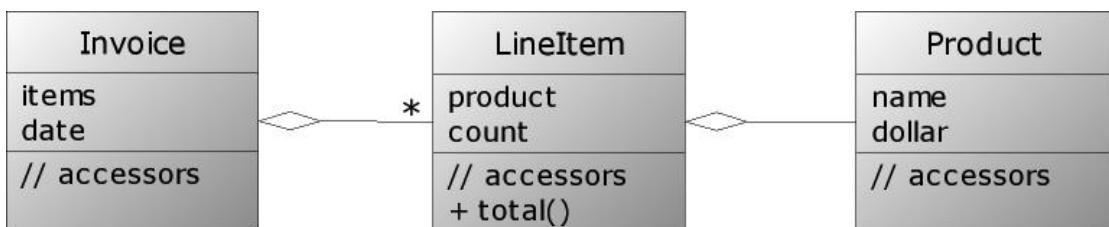


图 7.1

### Listing 7.23 Invoice example for GPath

```
class Invoice {  
    List items  
    Date date  
}  
class LineItem {  
    Product product  
    int count  
    int total() {  
        return product.dollar * count  
    }  
}  
class Product {  
    String name  
    def dollar  
}  
  
def ulcDate = new Date(107, 0, 1)  
def ulc = new Product(dollar:1499, name:'ULC')  
def ve = new Product(dollar:499, name:'Visual Editor')  
  
def invoices = [  
    new Invoice(date:ulcDate, items: [  
        new LineItem(count:5, product:ulc),  
        new LineItem(count:1, product:ve)  
    ]),  
    new Invoice(date:[107, 1, 2], items: [  
        new LineItem(count:4, product:ve)  
    ])  
]  
  
assert [5*1499, 499, 4*499] == invoices.items*.total()  
assert ['ULC'] ==  
    invoices.items.grep{it.total() > 7000}.product.name  
def searchDates = invoices.grep{  
    it.items.any{it.product == ulc}  
}.date*.toString()  
assert [ulcDate.toString()] == searchDates
```

← ① Total for each line item  
← ② Query of product names  
③ Query of invoice date

在列表 7.23 的问题是复杂难懂，首先，在（1），找到每一张发票的总价，这是将所有的项目累加起来。然后我们在（2）查询项目总费用超过 7000 元的所有产品的名称，最后，在（3）查找包含购买了 ULC 产品的每一张发票的日期，并且将它们转换为字符串。

打印出完全的 java 等价实现代码将需要超过 4 页内容并且阅读是非常无趣的，如果你想阅读这个代码，你能在本书的在线资源中找到。

Gpath 和相应的 java 代码的比较是有趣的，Gpath:

```
invoices.items.grep{ it.total() > 7000 }.product.name
```

相应的 java 代码为：

```

// Java

private static List getProductNameWithItemTotal(Invoice[]
invoices) {

    List result = new LinkedList();

    for (int i = 0; i < invoices.length; i++) {

        List items = invoices[i].getItems();

        for (Iterator iter = items.iterator(); iter.hasNext();) {

            LineItem lineItem = (LineItem) iter.next();

            if (lineItem.total() > 7000) {

                result.add(lineItem.getProduct().getName());
            }
        }
    }

    return result;
}

```

表 7.4 给出了两个版本的一些材料，比较代码的行数（LOC），语句数，及嵌套的复杂程度。

也许可以对 java 版本的代码进行裁剪，但是还是有数量级的差别：groovy 需要的代码量至少比 java 少 25%，并且语句少 10%！

写更少的代码不仅仅是一个实践的理由，这也意味着犯更少的错误和更少的测试工作。而有些新开发人员考虑的是一天写多少行代码，而我们考虑的是一天完成的功能。

在许多语言中，较少的代码导致意思的不明确，但在 groovy 中不是这样，Gpath 的例子是最好的证明，它比 java 对应物更容易阅读和理解，即使是复杂的逻辑。

**Table 7.4 GPath example: Groovy and Java metrics compared**

|               | LOC <sup>a</sup> |      | Statements <sup>b</sup> |      | Complexity |      |
|---------------|------------------|------|-------------------------|------|------------|------|
|               | Groovy           | Java | Groovy                  | Java | Groovy     | Java |
| CallingScript | 16               | 84   | 7                       | 72   | 1          | 4    |
| Invoice       | 4                | 16   | 0                       | 4    | 0          | 1    |
| LineItem      | 7                | 19   | 1                       | 5    | 1          | 1    |
| Product       | 4                | 16   | 0                       | 4    | 0          | 1    |
| Total         | 31               | 135  | 7                       | 85   |            |      |

a. Lines of code without comments and newlines

b. Assignments, method calls, and returns

最后，考虑一下可维护性，假设你的客户完善他们的需求，并且你需要改变查找逻辑，在 groovy 中做的工作要比 java 少多少？

## 7.5.2 注入展开操作符

Groovy 提供了一个\*展开操作符来连接到涉及到的 list 的展开点操作符，这看起来就像使用逗号分隔的对象来创建 list 列表的相反操作，展开操作符按顺序分配 list 中所有的条目给接收者，这样接受的方法可以有多个参数，或者有一个 list 的参数。

这有什么好处？假设你有一个方法，这个方法在一个列表中返回多个结果，并且你的代码需要将这些参数传递给第二个方法，展开操作符分配列表中的所有结果为第二个方法的参数：

```
def getList() {  
    return [1,2,3]  
}  
  
def sum(a,b,c) {  
    return a + b + c  
}  
  
assert 6 == sum(*list)
```

这样在接受方法单独声明每一个参数的时候，我们可以方便的传递多个参数给该方法。

使用展开操作符进行展开操作同样在 `range` 上及展开一个 `list` 中的条目到另一个 `list` 中也是可以工作的：

```
def range = (1..3)  
assert [0,1,2,3] == [0,*range]
```

相同的戏法同样可以应用在 `map` 上：

```
def map = [a:1,b:2]  
assert [a:1, b:2, c:3] == [c:3, *:map]
```

展开操作符淘汰了合并 `list`、`range` 和 `map` 需要的代码，你将在 10.3 节看到这个练习，这个操作符有助于实现一个用来进行数据库访问的用户命令语言。

如前面显示的断言，展开操作符在表达式中使用是十分方便的，支持在过程风格上的函数式编程风格，在过程风格中，你可以使用这样的语句 `list.addAll(otherlist)`。

现在，看看 `groovy` 最后的强劲特性，用这个特性你可以指派新的方法给任何 `Groovy` 或者 `java` 类。

### 7.5.3 使用 `use` 关键字进行混入

考虑一个程序从外部设备读取两个整数，然后将它们加在一起，并且回写结果，读和写都是字符串处理；但是加法操作是整数相加，你不能这样写：

```
write( read() + read() )
```

因为这样的结果为在字符串上调用 `plus` 方法来连接另外一个字符串。而不是进行整数加法运算。

`Groovy` 提供了 `use` 方法（像大多数 `groovy` 程序员，我们更喜欢叫做 `use` 关键字，但严格意义上应该叫它是一个方法，这是 `groovy` 增加到 `java.lang.Object` 上的方法），这允许你扩展一个类的可用实例方法，这样的用法被叫做类别（category），在我们的例子中，我们能在

字符串上扩展 plus 方法来得到一个所需的像 Perl 的行为：

```
use (StringCalculationCategory) {  
    write( read() + read() )  
}
```

一个 category 是一个类，这个类包含一组静态方法（叫做 category 方法），use 关键字使这些方法在方法的第一个参数的类上可以作为实例方法使用：

```
class StringCalculationCategory {  
    static String plus(String self, String operand) {  
        // implementation  
    }  
}
```

由于 self 是第一个参数，plus(operand)方法现在在 String 类上是可用的（或者覆盖）。列表 7.24 显示了一个完整的例子，它可靠的实现了这样的需要，使字符串进行真正的数字相加，而不是一般的字符串拼接。

#### Listing 7.24 The use keyword for calculation on strings

```
class StringCalculationCategory {  
    static def plus(String self, String operand) {  
        try {  
            return self.toInteger() + operand.toInteger()  
        }  
        catch (NumberFormatException fallback){  
            return (self << operand).toString()  
        }  
    }  
  
    use (StringCalculationCategory) {  
        assert 1 == '1' + '0'  
        assert 2 == '1' + '1'  
        assert 'x1' == 'x' + '1'  
    }  
}
```

Category 的用法被限制在闭包体内和当前线程上，这样改变之后不会在全局可视，避免了无意识的副作用。

通过这本书的语言基础部分,你已经看到 groovy 增加了新的方法到存在的类,整个 GDK 的实现都是增加到已经存在的 JDK 类上的新方法, use 方法允许任何 groovy 程序员在他们自己的代码中使用相同的策略。

Category 的使用有多个目的:

- 提供专门用途的方法,就像你看到的 StringCalculationCategory 一样,计算方法有相同的接受类并且也许覆盖了已经存在的行为,覆盖操作符方法是一个特例。
- 为已经存在的类库提供附加的方法,有效的解决了类库中没有完成的问题。
- 为不同的接受者提供一组方法,使他们组合工作——例如,一个在 java.io.OutputStream 上的新的 encryptedWrite 方法和在 java.io.InputStream 上的 encryptedRead 方法。
- 实现了 java 的装饰模式,但是不用编写许多麻烦的中继方法。
- 分离一个过度的大类为一个核心类和多方面的类别 (category),这样在需要的时候使 category 和核心类一起使用,注意,use 能接受任意多个 category 类。

在一个 category 方法被指派给 Object 的时候,这个方法在所有对象上都是可用的——也就是说,在哪里都可以用这个方法。这是美好的万能方法如日志,打印,持久化等等,例如,你已经知道每个对象都需要持久化:

```
class PersistenceCategory {  
    static void save (Object self) {  
        // whatever you do to store 'self' goes here  
    }  
}  
  
use (PersistenceCategory) {  
    save ()  
}
```

与 Object 相反,一个更小范围的应用也许更有利,如所有的 Collection 类或者你自己的业务对象(如果他们共享通用的接口)。

注意你可以为 `use` 传递许多 `category` 类，这些类通过逗号进行分隔，或者将这些 `category` 类放在一个 `list` 中。

```
use (ACategory, BCatagory, CCategory) {}
```

现在，你应该对 `groovy` 的强大特性有一些想法了，它们在第一次阅读的时候让人印象深刻，但在真实的应用中使用它们的时候，记住它们在早期提到的价值，这样你不会仅仅由于不熟悉特性和模式而错过一些优雅的代码，不久之后，你将熟练起来这样在你被迫回到 `java` 中的时候你将想念它们，好的消息是在 `java` 中是容易使用 `groovy` 的，我们将在 11 章进行探讨。

`Category` 类在闭包中的使用是一个 `groovy` 能提供的特性，因为他是元程序的概念，在下节将讨论元程序。

## 7.6 在 `groovy` 进行元程序编程

为了完全了解 `groovy` 的作用，了解 `groovy` 的内部是如何工作是有好处的，知道所有的细节不是必须的，但是熟悉大体上的概念将让你在使用 `groovy` 的工作中更自信和找到更多的优雅的解决方案。

这将让你了解到 `groovy` 执行的魔法，目的是解释一下一般性概念的用法，这样你能写出与 `groovy` 内部运行工作的集成解决方案，`groovy` 有许多拦截点，并且你可以选择性的覆盖 `groovy` 内建的行为能力，这样为你在写强大而优雅的解决方案的时候提供了许多选择，我们将描述这些拦截点，并且提供一些在实践中会用到的例子。

在这节描述的能力都是 `groovy` 的元对象协议 MOP (Meta-Object-Protocol) 的组成部分，MOP 是在运行时改变对象和类行为的系统能力。

在写这本书的时候，MOP 的重新设计正在进行中并且这叫做 new MOP，new MOP 主要关注的是内部实现的一致性和运行时的性能问题，我们高度关注程序员期望的地方。

### 7.6.1 理解元类（MetaClass）的概念

在 groovy 中，所有的对象都实现了 GroovyObject 接口，它就像我们提到过的其它类一样，声明在 groovy.lang 包中，GroovyObject 看起来有如下的格式：

```
public interface GroovyObject {  
    public Object invokeMethod(String name, Object args);  
    public Object getProperty(String property);  
    public void setProperty(String property, Object newValue);  
    public MetaClass getMetaClass();  
    public void setMetaClass(MetaClass metaClass);  
}
```

在 groovy 中你的所有类都是通过 GroovyClassGenerator 来构建的，因此它们都实现了 GroovyObject 这个接口并且有接口中的方法的默认实现——除了你自己选择实现它之外。

注意：如果你希望一个一般的 java 类也被作为一个 Groovy 类来组织，你必须实现 GroovyObject 接口，为了便利性，你可以扩展抽象类 GroovyObjectSupport，这个类提供了默认的实现。

GroovyObject 与 MetaClass 协作，MetaClass 是 Groovy 元概念的核心，它提供了一个 Groovy 类的所有的元数据，如可用的方法、属性列表，MetaClass 也实现了下列接口：

```
Object invokeMethod(Object obj, String methodName, Object args)  
Object invokeMethod(Object obj, String methodName, Object[] args)  
Object invokeStaticMethod(Object obj, String methodName, Object[]  
args)  
Object invokeConstructor(Object[] args)
```

这些方法进行真正的方法调用工作，使用 JAVA 反射 API（默认、并且性能更佳）或者通过透明创建一个反射类，GroovyObject 的 invokeMethod 方法默认实现总是转到相应的 MetaClass。

MetaClass 被存储在一个名称为 MetaClassRegistry 的中心存储器中，同时 groovy 也从 MetaClassRegistry 中获取 MetaClass。

图 7.2 显示了一个概要图（当思考 groovy 处理方法调用的时候请想到这个图）

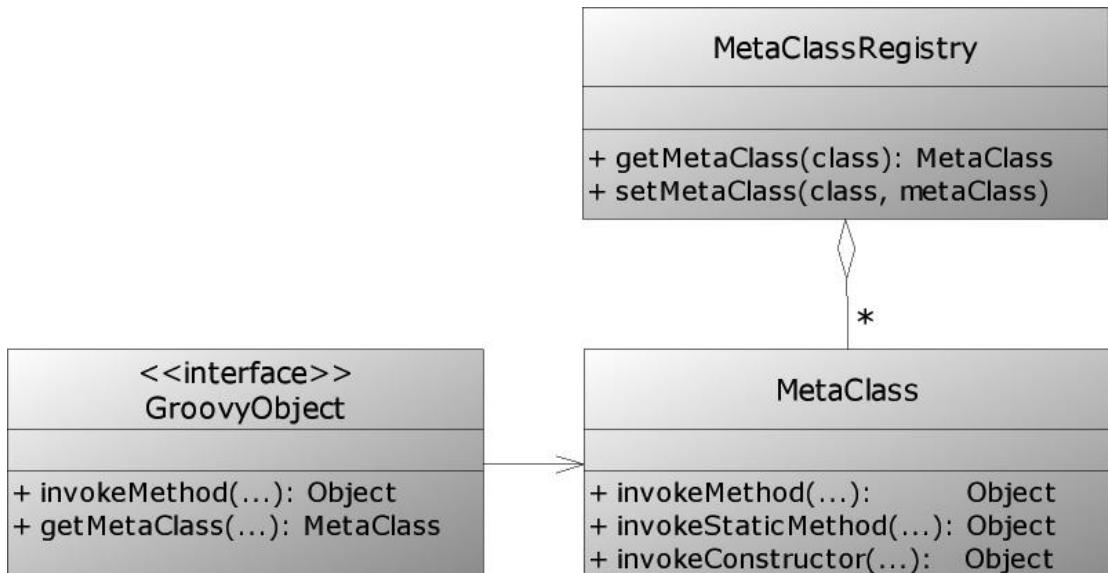


图 7.2

注意： *MetaClassRegistry* 类被设计为单例模式，不能被直接实现，但不管怎样，在代码中，可以使用 *InvokerHelper* 的一个工厂方法来引用到这个单例的注册中心。

通过图 7.2 的结构可以看到每个对象有一个 **MetaClass**，这种能力在默认的实现中没有使用，当前默认实现是在 **MetaClassRegistry** 中每一个类使用一个 **MetaClass**。当你需要定义的方法仅仅在一种类的实例上使用时这样是非常有用的（像 Ruby 的单态方法）。

注意： *GroovyObject* 引用到的 **MetaClass** 是 *GroovyObject* 在 **MetaClassRegistry** 中注册的类型，他是不需要相同的，例如，一个特定的对象可以有一个特殊的 **MetaClass**（这个 **MetaClass** 可以与该对象类的其他对象的 **MetaClass** 不一样）。

## 7.6.2 方法调用和拦截

Groovy 生成 java 字节码，这样每一个方法的处理都遵循下列机制之一：

- 1、类自己实现 invokeMethod 的方法（这也许被代理到 MetaClass）；
- 2、它自己的 MetaClass，通过 getMetaClass().invokeMethod() 进行调用；
- 3、在 MetaClassRegistry 中注册的该类型的 MetaClass。

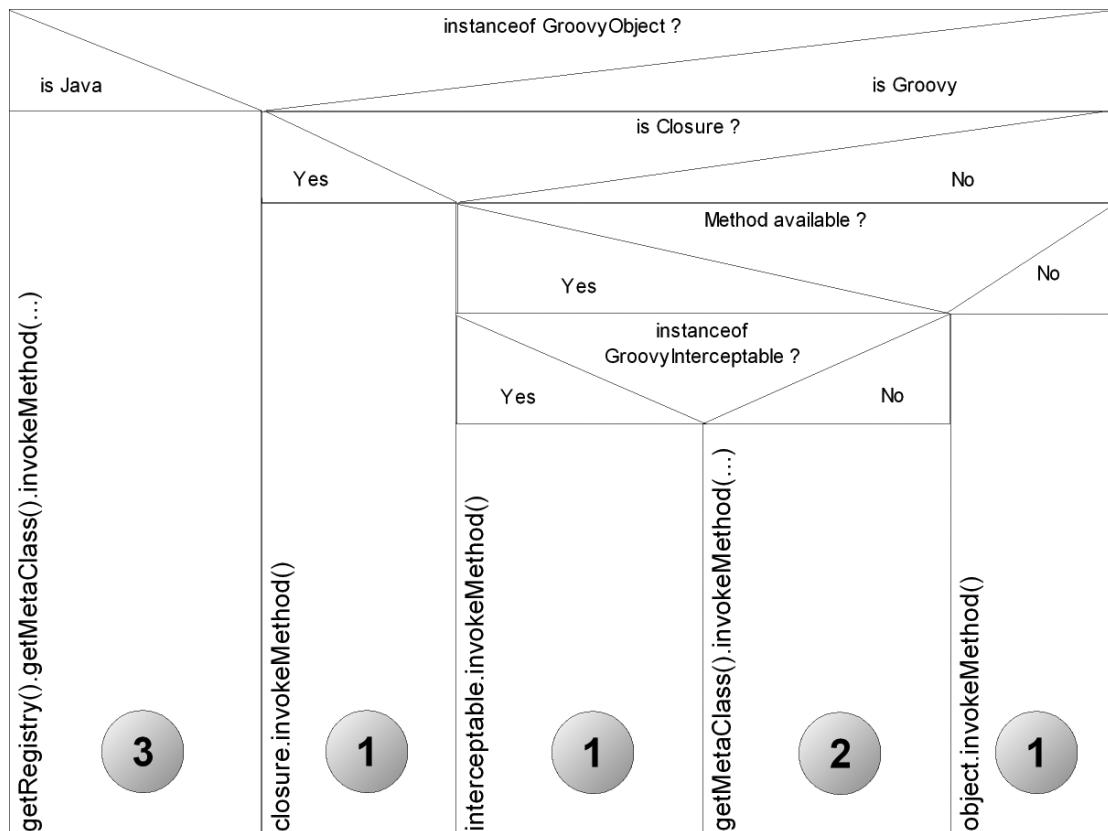


图 7.3 groovy 中三种不同类型的方法（根据被调用对象的类型和方法是否可以）调用逻辑

进行调用的逻辑显示在图 7.3 中，在图中的每个数字对对应在前面的数字列表中的项。

每个方法的调用时复杂的决策，当然很多时候你不需要考虑这些，你不应该在每个方法调用的时候都考虑这些，毕竟，groovy 已经使这些事情变得容易了一些，不管怎样，明白这些细节是有价值的，这样在复杂的情况下你总是可以正确的工作，它也为你在你的类中增加动态行为提供了各种可能。包括了下面的可能性：

- 你能够使用 aspects 进行方法调用的拦截，如日志跟踪、应用安全限制，强制事务控制等等。

- 你能够中转调用别的方法，例如，一个包装类能中转所有方法调用到被包装的对象上。

顺便说一下：闭包也是这样做的，它将方法调用代理给它的 `delegate`。

- 可以假装执行一个方法，这样可以应用一些特殊逻辑，例如，一个 `Html` 类可以假装有一个方法 `body`，当调用 `body` 方法的时候执行 `print('body')`。

顺便说一下，`builder` 就是这样做的，他们假装有相应的方法来定义嵌套产品结构，在第八章将进行详细的解释。

调用逻辑有多种方式来实现拦截、中转或者伪装方法：

- 实现/覆盖在 `GroovyObject` 中的 `invokeMethod` 方法来伪装或者中转方法调用（通常你定义的所有方法仍然被调用）；
- 实现 / 覆盖在 `GroovyObject` 中的 `invokeMethod` 方法，并且也实现 `GroovyInterceptable` 接口来增加方法的拦截调用代码。
- 提供一个 `MetaClass` 的实现类，并且在目标 `GroovyObject` 对象上调用 `setMetaClass`；
- 提供一个 `MetaClass` 的实现类，并且在 `MetaClassRegistry` 中为所有的目标类（Groovy 和 java 类）进行注册，这种方案通过 `ProxyMetaClass` 进行支持。

一般的说，覆盖/实现 `invokeMethod` 方法意味着覆盖点方法操作符。

在下节将教你如何运用这些知识。

### 7.6.3 方法拦截实战

假设我们有一个 groovy 类 `Whatever`，这个类有方法 `outer` 和 `inner`，它们互相调用，并且我们故意丢失了调用顺序，我们希望得到像下面这样的运行时的方法调用过程：

```
before method 'outer'
```

208 / 213

如果满意该文档，请支持(招商银行北京分行双榆树支行 6225881008887381 账户名：吴翊)

```
before method 'inner'  
after method 'inner'  
after method 'outer'
```

这证实了在 `outer` 方法中调用了 `inner` 方法。

由于这是一个 `GroovyObject` 对象，我们能够覆盖 `invokeMethod` 方法，为了保证我们能够拦截定义的所有方法，我们需要实现 `GroovyInterceptable` 接口，这是一个标记接口，没有任何方法需要实现。

在 `invokeMethod` 方法内部，我们在执行方法调用前后写跟踪日志，为了日志的可观察性我们保留了缩进，跟踪输出默认为 `System.out` 或者一个给定的 `Writer`，这样容易进行测试，我们通过提供一个 `writer` 属性来达到这个目标。

为了使我们的代码更清晰，我们将所有的跟踪函数在父类 `Traceable` 中，列表 7.25 显示了最终解决方案。

### Listing 7.25 Trace implementation by overriding invokeMethod

```
import org.codehaus.groovy.runtime.StringBufferWriter
import org.codehaus.groovy.runtime.InvokerHelper
class Traceable implements GroovyInterceptable {    ← Tagged superclass
    Writer writer = new PrintWriter(System.out)    ← Default : stdout
    private int indent = 0
    Object invokeMethod(String name, Object args){    ← Override default
        writer.write("\n" + ' '*indent + "before method '$name'")
        writer.flush()
        indent++
        def metaClass = InvokerHelper.getMetaClass(this)
        def result = metaClass.invokeMethod(this, name, args)
        indent--
        writer.write("\n" + ' '*indent + "after  method '$name'")
        writer.flush()
        return result
    }
    class Whatever extends Traceable {    ← Production class
        int outer(){
            return inner()
        }
        int inner(){
            return 1
        }
    }
    def log = new StringBuffer()
    def traceMe = new Whatever(writer: new StringBufferWriter(log))    ← Test settings
    assert 1 == traceMe.outer()    ← ② Start
    assert log.toString() == """
        before method 'outer'
        before method 'inner'
        after  method 'inner'
        after  method 'outer'"""
}
```

当中转方法调用的时候重要的是不能进入一个无限循环中，在使用同一种方式拦截方法调用的时候（因此在图 7.3 中进入了同一列）这是不可避免的，我们在（1）调用 invokeMethod 方法强制进入了最左边的列。

在（2）我们使用了 groovy 便利的 StringBufferWriter 来进行输出，然后输出到 System.out，使用新的 Whatever 不需要参数。

在（2）开始整个执行过程，我们也断言我们将获取到适当的结果。

不幸的是，这个解决方案有限制，首先，它仅仅工作在 GroovyObject 对象上，不能是任意的 java 类，第二，如果检查的类已经继承了别的类，那么它是不工作的。

回忆图 7.3，我们需要通过替换在 `MetaClassRegistry` 中的 `MetaClass` 来跟踪所  
以的日志，这样的一个类已经在 groovy 代码库中：`ProxyMetaClass`。

这个类担当一个已经存在的 `Meta` 装饰，并且通过使用一个 `Interceptor` 来增加拦截  
能力（参考 groovy javadoc 的 `groovy.lang.Interceptor` 文档），幸运的是，已  
经存在了一个 `TracingInterceptor`，列表 7.26 显示了我们怎样在 `Whatever` 中使用  
这个类。

#### Listing 7.26 Intercepting method calls with `ProxyMetaClass` and `TracingInterceptor`

```
import org.codehaus.groovy.runtime.StringBufferWriter

class Whatever {
    int outer(){
        return inner()
    }
    int inner(){
        return 1
    }
}

def log = new StringBuffer("\n")
def tracer = new TracingInterceptor()           ← Construct the Interceptor
tracer.writer = new StringBufferWriter(log)
def proxy = ProxyMetaClass.getInstance(Whatever.class)   ← Retrieve a suitable ProxyMetaClass
proxy.interceptor = tracer
proxy.use {                                     ← Determine scope for using it
    assert 1 == new Whatever().outer()          ← Start execution
}

assert log.toString() == """
before Whatever.ctor()
after Whatever.ctor()
before Whatever.outer()
before Whatever.inner()
after Whatever.inner()
after Whatever.outer()
"""


```

注意这个解决方案在从 groovy 中调用 java 类的时候也可用。

对于 `GroovyObject` 对象可以不通过 `MetaClassRegistry` 进行调用，你可以使用  
`use` 方法来保证它的工作：

```
proxy.use(traceMe) {
    // call methods on traceMe
}
```

`Interceptor` 和 `ProxyMetaClass` 对调试是有用的，但是仅仅适用简单任务。

**注意：**在应用大范围的改变到 `MetaClassRegistry` 的时候要当心，这可能导致代码产生看起来与它们无关的潜在错误。小心的使用切面，这样可以避免太多切面造成的痛苦。

这就是 Groovy 的元能力，他是有许多魔术的魔术师，并且你自己也可以成为一个魔术师。

MOP 使 Groovy 成为一门动态语言，它是 groovy 带给 java 平台许多发明的基础，这本书的其余部分将展示更重要的部分：建造者、标记、持久化、分布式编程、基于测试目的的透明模拟和存根，建立在已经存在的框架上的动态 API 类型，如用于 Windows Scripting 的 Scriptom 或者通过 Hibernate 建立的 ORM。

这些动态特性使得在 java 平台建立像 Grails（参考 16 章）这样的框架成为可能。

Groovy 通常被认为是 JVM 的一个脚本语言，但它的 java 的脚步能力不是最根本的区别特征，MOP 和结果动态提升使 Groovy 比其他语言更高一等。

## 7.7 总结

恭喜，在看完这一章之后，你已经看完了这本书的第一部分，如果你是动态语言的新手，你的脑海中也许已经转动起来了——这是一个不寻常的旅程。

本章开始部分没有太多的惊喜，仅仅显示了在 java 和 groovy 中定义和组织类的异同，我们介绍了方法和构造器的命名参数，方法的可选参数，通过下标操作符动态的查找类属性，也介绍了 Groovy 的“运行时加载”的能力，她的步骤于 java 有明显的不同。

Groovy 的 JavaBean 的处理约定被增强，也展示了 groovy 类的 JavaBean 风格的属性比相应的 Java 等价物简单和更具有自描述能力。到你看到 groovy 增强功能如 GPath 和 category 的时候，偏差已经很明显，并且 groovy 的动态特性已经开始显示出了痕迹。

最后，通过 groovy 中 MOP 实现的讨论，了解了动态特性的实现。

回顾语言的不同方面是相互依存和支撑的，使用 map 的缺省构造方法，在下标操作符中使用 range，通过操作符覆盖来使用 switch 进行结构控制，使用闭包来过滤 list，在一般的构造方法中使用 list 类型，通过属性语法使用 bean 属性等等，这些功能不但使 groovy 更强大而且也使 groovy 更有趣。

在不改变可读性的情况下 Groovy 代码的紧凑性是十分显著的，也有报告说由于每天只编写了 70 行代码，开发者的生产率没有得到提高。实际上现在提高生产率是在每一行代码中，现在每一行代码表达了更多的内容，现在， groovy 的一行代码可以替换 java 的多行代码，我们能够看到下一个提升开发者生产率的重点内容。

End 7 adapter

后记：

Groovy in action 第一部分已经全部完成，后续的翻译工作还在继续，希望阅读该文档的 java 开发人员对我的翻译感到满意的同时，能从口袋中掏出一块钱来支持我继续翻译下去，也欢迎各位朋友与我联系，我的电话是 13426268797。

我的银行账户为：

招商银行北京分行双榆树支行

6225881008887381

账户名：吴翊