

# 探索GDK



Groovy不仅给Java虚拟机（JVM）带来了动态语言的优势，还增强了存在已久的Java开发包（JDK）的性能。因为享受得到更好、更轻量级、更优雅的Java API，使用Groovy编程的效率很高。

前面已经介绍过Groovy通过便捷方法对JDK的增强，其中很多方法大量使用了闭包。Groovy的这一扩展称作Groovy Java开发包（Groovy Java Development Kit，Groovy JDK）或GDK。<sup>①</sup>

图7-1显示了JDK和GDK之间的关系。GDK是基于JDK的，所以在Java代码和Groovy代码之间传递对象时，无需任何转换。当处在同一JVM中时，Java端和Groovy端使用的是同一对象。对于Groovy端看到的对象，因为Groovy向其中添加了便于使用、可以提高开发效率的方法，所以看上去更时髦。



图7-1 JDK和GDK

GDK扩展了一些JDK中的类，我们会在本书的不同章节中讨论其中的一部分。本章主要关注两个方面：一个是对`java.lang.Object`类的扩展，另一个是对常用类的其他各种扩展。

## 7.1 使用 Object 类的扩展

本节将探索Groovy对众类之母`java.lang.Object`的一些扩展。在第6章中，我们看到了Groovy在`Collection`上添加的方法：`each()`、`collect()`、`find()`、`findAll()`、`any()`和`every()`。其实不仅`Collection`上提供了这些方法，我们在任何对象上都可以使用它们。这为

<sup>①</sup> <http://groovy.codehaus.org/groovy-jdk>

我们以类似方式使用不同对象和集合类提供了一致的API——这是组合模式的优点之一（参见*Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]）。Groovy还在Object上添加了一些和集合类无关的便捷方法。我们无意将本章变成GDK库的完整参考手册，因此这里暂不介绍所有这些方法。相反，重点将放在那些有可能激发我们的兴趣，或者对日常任务很有帮助的方法上。

### 7.1.1 使用 dump 和 inspect 方法

如果想知道类的一个实例包含哪些内容，可以在运行时使用dump()方法轻松获得：

#### ExploringGDK/ObjectExtensions.groovy

```
str = 'hello'

println str
println str.dump()
```

看一下这段代码打印的该对象的详细信息：

```
hello
<java.lang.String@5e918d2 value=hello offset=0 count=5 hash=99162322>
```

dump()使我们得以一窥对象内部。我们可以将其用于调试、日志和学习。它会给出目标实例的类型（class）、散列码及字段。

Groovy还向Object类添加了另一个方法——inspect()。该方法旨在说明创建一个对象需要提供什么。如果类没有实现该方法，会简单地返回toString()所返回的内容。如果对象要接收大量输入，该方法可以帮助类的使用者在运行时确定他们应该提供的内容。

### 7.1.2 使用上下文 with()方法

7

JavaScript和VBScript都有with这个友好的特性，支持创建一个上下文（context）。在with的作用域内调用的任何方法，都会被定向到该上下文对象，这样就去掉了对该实例的多余引用。在Groovy中，Object的with()方法提供了同样功能。（Groovy中with()方法是作为identity()的同义词引入的，所以它们可以互换使用。）该方法接受一个闭包作为参数。在闭包内调用的任何方法都会被自动解析到上下文对象。我们看一个例子，先从没有利用这种简洁性的代码开始：

#### ExploringGDK/Identity.groovy

```
lst = [1, 2]
lst.add(3)
lst.add(4)
println lst.size()
println lst.contains(2)
```

在前面的代码中，我们一直在调用lst上的方法，其中lst引用的是一个ArrayList实例。这里没有隐含的上下文，我们重复地（或者说冗余地）使用了对象引用lst。在Groovy中，可以使用with()方法设置一个上下文，因此代码可以改成下面这样：

#### ExploringGDK/Identity.groovy

```
lst = [1, 2]
lst.with {
    add(3)
    add(4)
    println size()
    println contains(2)
}
```

这段代码噪音很少，其输出如下：

```
4
true
```

with()方法是怎样知道把闭包内的调用路由到上下文对象的呢？魔力在于该闭包的delegate属性（更多信息，参见4.9节）。我们检查一下附到with()上的闭包中的delegate属性、this属性以及owner属性。

#### ExploringGDK/Identity.groovy

```
lst.with {
    println "this is ${this},"
    println "owner is ${owner},"
    println "delegate is ${delegate}."
}
```

输出说明了我们所关注的引用的详细信息：

```
this is Identity@ce56f8,
owner is Identity@ce56f8,
delegate is [1, 2, 3, 4].
```

当我们调用with()方法时，它会将该闭包的delegate属性设置到调用with()的对象上。正如4.9节所讨论的，delegate会负责this不处理的方法。

with()方法使我们更方便地在一个对象上调用多个方法，利用上下文来减少混乱吧。在构建领域特定语言（DSL）时，会发现该方法非常有用。还可以实现类似脚本的调用，隐式地将其路由到幕后的对象，第19章中将详述。

### 7.1.3 使用 sleep

添加到Object上的sleep()方法应该叫作酣睡（soundSleep），因为在给定的毫秒数时间（近似）内睡眠时，该方法会忽略中断。

我们看一个使用sleep()方法的例子：

#### ExploringGDK/Sleep.groovy

```
thread = Thread.start {
    println "Thread started"
    startTime = System.nanoTime()
    new Object().sleep(2000)
    endTime = System.nanoTime()
    println "Thread done in ${(endTime - startTime)/10**9} seconds"
}
new Object().sleep(100)
println "Let's interrupt that thread"
thread.interrupt()
thread.join()
```

输出说明，该线程忽略了中断，并完成执行：

```
Thread started
Let's interrupt that thread
Thread done in 2.000272 seconds
```

这里我们使用了Groovy添加的Thread.start()方法。这是在一个不同的线程中执行一段代码的方便方法。在Object上调用sleep()与使用Java提供的Thread.sleep()的区别在于：如果有InterruptedException，前者会压制下来。如果我们确实想被中断，也不必受try-catch之苦。相反，可以在前面的sleep()方法上使用一个变种版本，它接受一个处理中断的闭包：

#### ExploringGDK/Sleep.groovy

```
def playWithSleep(flag)
{
    thread = Thread.start {
        println "Thread started"
        startTime = System.nanoTime()
        new Object().sleep(2000) {
            println "Interrupted... " + it
            flag
        }
        endTime = System.nanoTime()
        println "Thread done in ${(endTime - startTime)/10**9} seconds"
    }

    thread.interrupt()
    thread.join()
}

playWithSleep(true)
playWithSleep(false)
```

在输出中可以看到闭包是如何处理中断的：

```

Thread started
Interrupted... java.lang.InterruptedException: sleep interrupted
Thread done in 0.00437 seconds
Thread started
Interrupted... java.lang.InterruptedException: sleep interrupted
Thread done in 1.999077 seconds

```

在中断处理器内，我们可以采取任何适当的动作。如果需要访问`InterruptedException`，也可以，它作为闭包的一个参数存在。如果我们在闭包内返回一个`false`值，`sleep()`将继续，就好像没有被中断，在前面代码中，通过第二个`playWithSleep()`调用语句可以看到这一点。

### 7.1.4 间接访问属性

我们知道，Groovy使访问属性变得非常容易。例如，对于一个`Car`类型的`car`实例，要获得其`miles`属性，可以简单地调用`car.miles`。然而，如果编写代码时不知道属性名，这种语法就派不上用场了，比如属性名依赖于用户的输入，而且我们不想为所有可能的输入硬编码一个处理分支。这时可以使用`[]`操作符（该操作符映射到Groovy添加的`getAt()`方法）动态地访问属性。如果将该操作符用于赋值语句的左侧，它则映射到`putAt()`方法。

我们看一个例子：

```

ExploringGDK/IndirectProperty.groovy
class Car {
    int miles, fuelLevel
}

car = new Car(fuelLevel: 80, miles: 25)

properties = ['miles', 'fuelLevel']
// 上面的列表可能通过一些输入来填充
// 或者来自一个Web应用中的动态表单

properties.each { name ->
    println "$name = ${car[name]}"
}

car[properties[1]] = 100

println "fuelLevel now is ${car.fuelLevel}"

```

我们能够间接地与该实例交互，如输出所示：

```

miles = 25
fuelLevel = 80
fuelLevel now is 100

```

这就是使用`[]`操作符访问`miles`和`fuelLevel`属性的过程。这种方法可应用于通过输入接收

的属性名，例如动态创建和填充Web表单。可以轻松地编写一个高阶函数，让它接受属性名列表和一个实例，并以XML、HTML或任何其他我们期望的格式来输出这些属性名和它们的值。可以通过对象的**properties**属性（也就是**getProperties()**方法）获得其所有属性的列表。

### 7.1.5 间接调用方法

如果以**String**形式接收到方法名，而且想调用该方法，使用反射要这样实现——首先必须从实例取到**Class**元对象，然后调用**getMethod()**方法得到**Method**实例，最后在该实例上调用**invoke()**方法。噢，还有，别忘了那些不得不处理的异常。

在Groovy中不需要做这些，而只需简单地调用**invokeMethod()**方法。Groovy中的所有对象都支持该方法。这是一个例子：

```
ExploringGDK/IndirectMethod.groovy

class Person {
    def walk() { println "Walking..." }
    def walk(int miles) { println "Walking $miles miles..." }
    def walk(int miles, String where) { println "Walking $miles miles $where..." }
}

peter = new Person()

peter.invokeMethod("walk", null)
peter.invokeMethod("walk", 10)
peter.invokeMethod("walk", [2, 'uphill'] as Object[])
```

下面是间接调用该方法的输出：

```
Walking...
Walking 10 miles...
Walking 2 miles uphill...
```

因此，如果编写代码时不知道方法名，而在运行时获得，那就可以使用几行代码将其变为实例上的动态调用。

Groovy还提供了**getMetaClass()**方法，用以获得元类（metaclass）对象，这是在Groovy中利用动态能力的一个关键对象，在后面的章节中我们将看到。

Groovy的扩展API远不止扩展了JDK中最基础的**Object**类，下面我们会看到。

## 7.2 其他扩展

GDK所做的扩展远不止**Object**类，其他一些JDK的类和接口也得到了增强。再次重申，GDK的扩展很广，本节所涉及的只是一个子集。这里要介绍的是那些最常用的扩展。

### 7.2.1 数组的扩展

在所有数组类型上，比如int[]、double[]和char[]等，都可以使用Range对象作为索引（创建数组的语法请参见2.11.7节）。下面演示了如何使用索引的范围访问一个int数组中的连续几个值：

#### ExploringGDK/Array.groovy

```
int[] arr = [1, 2, 3, 4, 5, 6]
```

```
println arr[2..4]
```

输出显示了给定范围内的值：

```
[3, 4, 5]
```

GDK向List、Collection和Map添加了很多便捷方法，通过第6章的学习我们已经很熟悉了。

### 7.2.2 使用 java.lang 的扩展

对于基本类型的包装器，如Character、Integer等，有一个值得注意的补充，那就是重载操作符映射的方法，比如+操作符映射的plus()、++操作符映射的next()等。当创建DSL时，我们会发现这些方法（或者应该说是操作符）很有用。

Number（Integer和Double就扩展了该类）加上了迭代器方法upto()和downto()。它还有step()方法（参见2.1.2节）。这些方法有助于对一个范围内的值进行迭代。

在2.1.3节中，我们看了一些与系统级进程交互的例子。Process类提供了访问stdin、stdout和stderr命令的便捷方法，分别对应out、in和err属性。它还有一个text属性，可以为我们提供完整的标准输出或来自进程的响应。如果想一次性读取完整的标准错误，可以在进程实例上使用err.text。使用<<操作符可以以管道方式链接到一个进程中。（管道——|，在类Unix系统上用于将一个进程的输出链接到另一个进程的输入。）下面通过一个例子来看一下与一个wc进程的通信——wc程序是类Unix系统上一个流行的实用程序，它会向标准输出打印从其标准输入中发现的单词数、行数和字符数：

#### ExploringGDK/UsingProcess.groovy

```
process = "wc".execute()

process.out.withWriter {
    // 将输入发送到进程
    it << "Let the World know...\\n"
    it << "Groovy Rocks!\\n"
}
```

```
// 从进程读取输入
println process.in.text
// 或
//println process.text
```

前面代码的输出是wc返回的结果——2行，6个单词，36个字符：

```
2      6      36
```

在这段代码中，首先，通过调用String的execute()获得了一个进程实例。我们想向wc的标准输入写内容，所以需要来自程序的一个OutputStream。可以通过调用out属性从进程获取。

要写入内容，可以使用<<操作符。然而，一旦把数据写到了流中，我们就想刷新（flush）并关闭该流。可以使用一个方法——withWriter()，同时完成这两方面的处理。该方法会将OutputStreamWriter附到OutputStream上，同时将其传给闭包。当我们从闭包返回时，它会自动刷新并关闭流（参见4.5节）。

试试用Java实现前面这段代码，你就能体会到，Groovy不仅节省时间，还带来了优雅的享受。

如果想将命令行参数发送给进程，有两个选择：把参数格式化为一个字符串，或者创建一个参数的String数组。String[]也支持execute()方法，数组的第一个元素会被当作要执行的命令，其余元素则被视作该命令的参数。作为替代，可以使用List的execute()方法。

这是一个向groovy命令传递命令行参数的例子：

#### ExploringGDK/ProcessParameters.groovy

```
String[] command = ['groovy', '-e', '"print \'Groovy\'"]'
println "Calling ${command.join(' ')}"
println command.execute().text
```

上述代码执行的命令及输出如下：

```
Calling groovy -e "print 'Groovy'"
Groovy
```

在Groovy中，我们可以非常轻松地启动一个进程、发送参数，以及与该进程交互。只需要几行代码。

如果想创建多个线程，并将任务分派给单独的线程执行，Groovy可以让我们少敲很多字。使用start()方法启动一个线程，并为其提供一个将在单独的线程中执行的闭包。如果希望该线程是守护线程（daemon thread），可以使用startDaemon()方法代替。如果当前已经没有活跃的非守护线程，守护线程会退出——有点像只有老板在的时候才干活的员工。以下是这两个方法的实际演示例子：

#### ExploringGDK/ThreadStart.groovy

```
def printThreadInfo(msg) {
```

```

def currentThread = Thread.currentThread()
println "$msg Thread is ${currentThread}. Daemon? ${currentThread.isDaemon()}"
}

printThreadInfo 'Main'

Thread.start {
    printThreadInfo "Started"
    sleep(3000) { println "Interrupted" }
    println "Finished Started"
}

sleep(1000)

Thread.startDaemon {
    printThreadInfo "Started Daemon"
    sleep(5000) { println "Interrupted" }
    println "Finished Started Daemon" // 不会执行到这里
}

```

下面的输出说明了线程信息：

```

Main Thread is Thread[main,5,main]. Daemon? false
Started Thread is Thread[Thread-1,5,main]. Daemon? false
Started Daemon Thread is Thread[Thread-2,5,main]. Daemon? true
Finished Started

```

在这个例子中，主线程和我们创建的非守护线程一退出，守护线程就被中止了。可见，在Groovy中创建线程，我们不需要使用**Thread**或**Runnable**的实例。处理线程创建非常简单，而且方便。

### 7.2.3 使用 `java.io` 的扩展

`java.io`包中的**File**类也加入了很多方法。其中**eachFile()**和**eachDir()**（及其变种）这样的方法，可以接受闭包，为目录和文件的导航与迭代提供了方便的方式。

假设我们想读取一个文件的内容。下面是实现该功能的Java代码：

```

// Java代码
import java.io.*;
public class ReadFile {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(
                new FileReader("thoreau.txt"));
            String line = null;
            while((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        }
    }
}

```

```
    } catch(FileNotFoundException ex) {  
        ex.printStackTrace();  
    } catch(IOException ex) {  
        ex.printStackTrace();  
    }  
}
```

这么读文件可真费劲。Groovy通过向BufferedReader、InputStream和File添加一个text属性，使之简单了许多，我们可以把文件的全部内容都读到一个String中。这对处理或打印整个输出很有用。下面用Groovy重写了前面的代码：

## Exploring GDK/ReadFile.groovy

```
println new File('thoreau.txt').text
```

上面代码输出了thoreau.txt文件的内容，具体如下：

"I went to the woods because I wished to live deliberately,  
to front only the essential facts of life, and see if I could  
not learn what it had to teach, and not, when I came to die,  
to discover that I had not lived. "

- Henry David Thoreau

如果不想一次性读入整个文件，而想一次读取并处理一行，可以使用`eachLine()`方法，它会对读入的每行文本调用一个闭包：

## Exploring GDK/ReadFile.groovy

```
new File('thoreau.txt').eachLine { line ->
    println line // 或者在这里执行自己想对该行进行的任何处理
}
```

如果只是想取得满足某个特定条件的那些行文本，可以使用filterLine()，如下所示：

## Exploring GDK/ReadFile.groovy

```
println new File('thoreau.txt').filterLine { it =~ /life/ }
```

通过前面代码提取出的、过滤后的文本行如下：

to front only the essential facts of life, and see if I could

我们仅过滤出了输入文件由包含life的行

如果想使用完毕时自动刷新并关闭一个输入流，可以使用`withStream()`方法。该方法会调用作为参数传入的闭包，并将`InputStream`的实例作为一个参数发送给该闭包。我们一从闭包返回，它就会刷新并关闭这个流。`Writer`有一个类似的方法，叫做`withWriter()`，我们在本节前面部分已经看过一个例子。

`InputStream`的`withReader()`会创建一个`BufferedReader`（被附到输入流上），并将其传给作为参数接受的闭包。也可以通过调用`newReader()`方法获得一个新的`BufferedReader`实例。

对于`InputStream`和`DataInputStream`中的输入，可以通过调用`iterator()`方法获得一个`Iterator`，然后使用该迭代器对输入进行迭代。说到迭代，我们也可以便利地迭代`ObjectInputStream`中的对象。

如果想使用`Reader`代替，也可以。添加到`InputStream`上的便捷方法，`Reader`上也有。

在Groovy中可以方便地向文件或流写入内容。`OutputStream`、`ObjectOutputStream`和`Writer`类都通过`leftShift()`方法（<<操作符）得到了革新。下面的代码示例使用该操作符向文件中写入信息：

#### ExploringGDK/ShiftOperator.groovy

```
new File("output.txt").withWriter{ file ->
    file << "some data..."
}
```

`java.io`包中的类还有其他一些扩展，它们使我们的生活更轻松了，编写代码用的时间也更少了。

### 7.2.4 使用 `java.util` 的扩展

在第6章中，我们探讨了Groovy对集合类的扩展。这一节，我们来看看`java.util`包中类的一些其他扩展。

`List`、`Set`、`SortedMap`和`SortedSet`都加入了`asImmutable()`方法，用于获得各自实例的一个不可变实例。这些类还加入了一个`asSynchronized()`方法，用于创建线程安全的实例。

`Iterator`支持`inject()`方法，我们在6.4节中讨论过。

`java.util.Timer`上加入了一个`runAfter()`方法。使用的语法更为简单，因为该方法接受一个闭包，该闭包将在一个给定的延迟（以毫秒为单位）之后运行。

正如我们在本章所讨论的，Groovy在`java.lang.Object`层次加入了很多方法。有的方法让我们在调试、日志或获取信息时可以一窥对象内部，有的方法让我们使用一致的接口对待对象和集合，就像组合模式那样。

`Object`也支持用于元编程的方法，可以动态地访问属性和调用方法。这些方法共同构建起来的高层抽象，减少了日常任务的应用代码的体积，也减少了所需时间。

还可以使用不同类上的专用方法，Groovy为一些类和接口增强了API，比如`Matcher`、`Writer`、`Reader`、`List`、`Map`和`Socket`等，举不胜举。GDK对一些JDK的类和接口也有扩展。

GDK太过庞大，本书无法一一覆盖；可以访问<http://groovy.codehaus.org/groovy-jdk>，查看全面且持续更新的GDK API列表。

在使用Groovy编程时，我们需要同时参考JDK和GDK。如果在JDK中没有发现要找的东西，一定要记得检查一下GDK，看它是不是支持我们要的特性。

## 7.3 使用扩展模块定制方法

Groovy 2.x不只赋予我们使用GDK方法的特权。使用扩展模块（extension-modules）特性，我们还可以在编译时向现有类添加实例方法或静态方法，并在运行时在应用中使用它们。下面通过一个例子，看一下必须遵循的一些简单步骤。

要使用该特性，需要做到两点：首先，想要添加的方法必须定义在一个扩展模块类中；其次，需要在清单文件（manifest）中放一些描述信息，告诉Groovy编译器要查找的扩展模块。

让我们在String上创建两个扩展方法，一个是实例方法，一个是静态方法，用于获取给定股票的价格。所有引入的扩展方法，只要基于JDK或GDK，并且将包含这些类的jar文件放在它们的classpath下，就可以被调用。

两类扩展方法都必须定义为static的，而且第一个参数应该是该方法要加入到的类型。定义中还要通过额外的参数来提供该扩展方法要接收的参数。

下面是String类上的一个实例扩展方法，写在一个扩展辅助类PriceExtension中（这里是由Groovy类编写的，也可以使用其他任何JVM语言编写，包括Java）。

```
Extension/com/agiledeveloper/PriceExtension.groovy
package com.agiledeveloper;

class PriceExtension {
    public static double getPrice(String self) {
        def url = "http://ichart.finance.yahoo.com/table.csv?s=$self".toURL()
        def data = url.readLines()[1].split(",")
        Double.parseDouble(data[-1])
    }
}
```

getPrice()方法被定义为static的，第一个参数说明该方法将被添加到哪个类上。这段代码并没有说出要添加的这个方法是实例方法还是静态方法；这个信息会放在清单声明中，一会我们会看到。

再为同样目的定义一个静态扩展方法。

```
Extension/com/agiledeveloper/PriceStaticExtension.groovy
package com.agiledeveloper;

class PriceStaticExtension {
    public static double getPrice(String selfType, String ticker) {
        def url = "http://ichart.finance.yahoo.com/table.csv?s=$ticker".toURL()

        def data = url.readLines()[1].split(",")
        Double.parseDouble(data[-1])
    }
}
```

`getPrice()`方法接收两个参数，第一个说明该方法要加入到哪个类上；第二个是实际的股票值，说明要获得哪支股票的价格。在第一个版本的`getPrice()`方法中，股票隐含地包含在实例中；而在这个版本中，股票信息必须作为一个参数传入，因为该方法要运行在`String`类的静态上下文中。

我们已经准备好了带有扩展方法的辅助类。下面需要声明其存在，并将声明信息和编译好的类打包到一个jar文件中。下面是声明信息，保存在META-INF/services目录下的org.codehaus.groovy.runtime.ExtensionModule文件中：

```
Extension/manifest/META-INF/services/org.codehaus.groovy.runtime.ExtensionModule
moduleName=price-module
moduleVersion=1.0-test
extensionClasses=com.agiledeveloper.PriceExtension
staticExtensionClasses=com.agiledeveloper.PriceStaticExtension
```

声明文件中包含4个信息的键值对。`moduleName`是为模块起的逻辑名称。`moduleVersion`用于检查该版本是否已经加载。`extensionClasses`是用逗号分隔的包含实例扩展方法的辅助类的名字。最后，`staticExtensionClass`是用逗号分隔的包含静态扩展方法的辅助类的名字。

使用下列命令编译这两个辅助类，并创建必要的jar文件：

```
$ groovyc -d classes com/agiledeveloper/*.groovy
$ jar -cf priceExtensions.jar -C classes com -C manifest .
```

`priceExtensions.jar`文件中包含了编译好的辅助类和清单文件。

再创建一个使用这些扩展方法的示例Groovy文件：

```
Extension/FindPrice.groovy
def ticker = "ORCL"

println "Price for $ticker using instance method is ${String.getPrice(ticker)}"
println "Price for $ticker using static method is ${ticker.getPrice()}"
```

我们分别调用了实例方法和静态方法。要加入这些方法，必须将`priceExtensions.jar`文

件包含在classpath下，像下面命令中这样：

```
$ groovy -classpath priceExtensions.jar FindPrice.groovy
```

Groovy会基于清单文件中提供的信息，无缝地加入扩展方法。下面显示了调用这些扩展方法的结果：

```
Price for ORCL using instance method is 34.75  
Price for ORCL using static method is 34.75
```

我们看过了Groovy对JDK方法的扩展，以及如何添加定制的扩展。Groovy还为各种任务提供了强大的类库集。下一章，我们将学习Groovy如何优雅地应对原本乏味的XML处理任务。

## 第8章

# 处理XML

8

处理XML可能很繁琐。使用传统的Java API和库创建和解析XML，往往会让情绪低落。而使用DOM API在文档层次结构中导航，肯定会让抓狂。

Groovy缓解了解析和创建XML文档之苦。我们已经看过一些创建XML文档的方式。这一章，我们将再回到这个主题，学习使用3种不同的设施来解析XML文档，它们带来了不同程度的便捷性与效率。我们还将浏览一下Groovy对创建XML文档的支持。

## 8.1 解析 XML

在Groovy中，如果有特殊的需求或原因要依赖较老的API，或者既有代码使用了这些API，那我们可以使用自己熟悉的、基于Java的解析方法与工具。如果已经有了可用的解析XML文档的Java代码，在Groovy中也可以轻松复用。Groovy不会强迫我们重复劳动。

不过，如果要创建新的XML解析代码，Groovy提供的设施则可谓是我们福音。

在最近的一个项目中，我必须使用来自大约400个XML文档的数据填充一个应用。乍看上去，这个活令人生畏；要处理的文件之多，足以让我望而却步。在快速浏览了一些文件后，我决定使用Groovy处理这些文件、解析XML文档并填充应用。使用`XmlSlurper`类，再结合大约30行Groovy代码，就足以把活干完了。

Groovy解析器相当强大，而且使用方便，它们还支持命名空间，下一节我们将看到。

本章中的所有示例都将会用到下面这个包含一系列语言及其作者信息的XML文档：

`WorkingWithXML/languages.xml`

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

```

<language name="Lisp">
  <author>McCarthy</author>
</language>
<language name="Modula-2">
  <author>Wirth</author>
</language>
<language name="Oberon-2">
  <author>Wirth</author>
</language>
<language name="Pascal">
  <author>Wirth</author>
</language>
</languages>

```

### 8.1.1 使用 DOMCategory

使用Groovy的分类（将在13.1节中详细探讨）可以在类上定义动态方法，其中有一个分类可用于处理文档对象模型（Document Object Model, DOM）——**DOMCategory**。Groovy还通过添加便捷方法，简化了DOM应用编程接口（API）。

**DOMCategory**可以通过类GPath（Groovy path expression，即Groovy路径表达式）的符号在DOM结构中导航。

仅通过子元素的名字就能访问所有子元素。例如，不用调用`getElementsByName('name')`，使用属性`name`就能获取该元素，就像`rootElement.language`这样。也就是说，给定根元素`rootElement`，简单地调用`rootElement.language`就能获取所有的`language`元素。<sup>①</sup>DOM解析器会给出`rootElement`；在下面的例子中，我们将使用`DOMBuilder`的`parse()`方法把文档加载到内存中。

在属性名之前放一个@可以获得该属性的值，就像`language.@name`这样。

在下面的代码中，我们使用**DOMCategory**在文档中取得语言的名字和作者：

#### WorkingWithXML/UsingDOMCategory.groovy

```

document = groovy.xml.DOMBuilder.parse(new FileReader('languages.xml'))

rootElement = document.documentElement

use(groovy.xml.dom.DOMCategory) {
  println "Languages and authors"
  languages = rootElement.language
}

```

8

<sup>①</sup> 这里原文有误，根元素是`rootElement`，而非`languages`。从后面的代码也可以知道，`languages = rootElement.language`。——译者注

```

languages.each { language ->
    println "${language.'@name'} authored by ${language.author[0].text()}"
}

def languagesByAuthor = { authorName ->
    languages.findAll { it.author[0].text() == authorName }.collect {
        it.'@name' }.join(', ')
}

println "Languages by Wirth:" + languagesByAuthor('Wirth')
}

```

使用这段代码提取出的数据如下：

```

Languages and authors
C++ authored by Stroustrup
Java authored by Gosling
Lisp authored by McCarthy
Modula-2 authored by Wirth
Oberon-2 authored by Wirth
Pascal authored by Wirth
Languages by Wirth:Modula-2, Oberon-2, Pascal

```

`DOMCategory`对于使用DOM API解析XML文档很有用，同时还结合了GPath查询的便捷与Groovy动态特性的优雅。

要使用`DOMCategory`，必须把代码放在`use()`块内。不过本章将会看到的另外两种方法并无这种限制。在前面的示例中，我们使用GPath语法从文档中提取出了想要的细节信息。还写了一个定制的方法（或者说过滤器），用于获得仅由Wirth创造的那些语言。

### GPath是什么？

与XPath可以帮助导航XML文档的层次结构很类似，GPath可以帮助导航对象（Plain Old Java Object和Plain Old Groovy Object，即POJO和POGO）和XML的层次结构。可以使用句点`(.)`符号遍历层次结构。例如，`car.engine.power`这种写法会帮助我们通过一个`car`实例的`getEngine()`方法访问其`engine`属性，然后再通过所获得的`engine`实例的`getPower()`方法，访问该实例的`power`属性。如果处理的不是对象，而是XML文档，这种写法会帮助我们获得元素`engine`的一个子元素`power`，而`power`又是元素`car`的一个子元素。与访问元素不同的是，可以使用`car.'@year'`（或`car.@year`）这种写法访问`car`的`year`属性。`@`符号说明要访问的是属性，而非子元素。

### 8.1.2 使用 XMLParser

`groovy.util.XMLParser`利用了Groovy的动态类型和元编程能力。可以直接使用名字访问文档中的成员。例如，可以使用`it.author[0]`访问一个作者的名字。

我们使用`XMLParser`从语言的XML文档中取得想要的数据：

```
WorkingWithXML/UsingXMLParser.groovy
languages = new XmlParser().parse('languages.xml')

println "Languages and authors"

languages.each {
    println "${it.@name} authored by ${it.author[0].text()}"
}

def languagesByAuthor = { authorName ->
    languages.findAll { it.author[0].text() == authorName }.collect {
        it.@name }.join(', ')
}

println "Languages by Wirth:" + languagesByAuthor('Wirth')
```

这段代码与我们在“使用`DOMCategory`”部分看到的示例很像。主要区别在于，这里没有使用`use()`块。`XMLParser`向元素添加了便捷的迭代器，所以可以使用诸如`each()`、`collect()`和`find()`等方法轻松实现导航。

使用`XMLParser`也有一些不足之处：它没有保留`XML InfoSet`，而且忽略了文档中的XML注释和处理指令。它带来的便捷性使其成为应对大部分常见处理需求的极好工具。然而，如果有其他特殊需求，我们必须探索更传统的解析器。

### 8.1.3 使用 XMLSlurper

对于较大的文档，`XMLParser`的内存使用可能让人难以忍受。`XMLSlurper`类可以处理这类情况。它在使用上与`XMLParser`类似。下面的代码与前面“使用`XMLParser`”部分的代码几乎相同：

```
WorkingWithXML/UsingXMLSlurper.groovy
languages = new XMLSlurper().parse('languages.xml')
println "Languages and authors"

languages.language.each {
    println "${it.@name} authored by ${it.author[0].text()}"
}
```

```

def languagesByAuthor = { authorName ->
    languages.language.findAll { it.author[0].text() == authorName }.collect {
        it.@name }.join(', ')
}
println "Languages by Wirth:" + languagesByAuthor('Wirth')

```

还可以使用XML文档中的命名空间来解析它们。命名空间让我想起一件往事，当时我接到马来西亚一家公司的电话，他们对涉及大量代码以强化测试驱动开发的培训很感兴趣。交谈中我问到，我需要使用什么语言。顿了一下，那位绅士不情愿地说：“英语，当然是英语。我们团队里的每个人英语都说得很好。”我的意思实际上是：“我要使用什么计算机语言”。这是一个日常交谈中上下文与混淆的例子。XML文档也有同样的问题，而命名空间可以帮助我们处理名字冲突。

记住，命名空间不是URL，但是它们需要保持唯一性。我们在XML文档中使用的命名空间前缀不是独一无二的。我们可以随便起（当然，要有一些命名约束）。因此，要引用查询中的一个命名空间，我们要为其关联一个前缀。可以使用`declareNamespaces()`方法实现这种关联，该方法接受一个以前缀为键，以命名空间为值的映射。一旦定义了前缀，我们的GPath查询也就可以获得名字的前缀了。`element.name`将返回所有带有`name`的子元素，不考虑命名空间；而`element.'ns:name'`则仅返回`ns`关联的命名空间中的元素。来看一个例子，假设我们有一个文档，其中包含计算机语言和自然语言的名字，如下所示：

```

<languages xmlns:computer="Computer" xmlns:natural="Natural">
    <computer:language name="Java"/>
    <computer:language name="Groovy"/>
    <computer:language name="Erlang"/>
    <natural:language name="English"/>
    <natural:language name="German"/>
    <natural:language name="French"/>
</languages>

```

元素名`language`或者是在`Computer`命名空间中，或者是在`Natural`命名空间中。下列代码演示了如何同时取得所有语言，以及如何仅取得`Natural`语言：

#### WorkingWithXML/UsingXMLSlurperWithNS.groovy

```

languages = new XmlSlurper().parse(
    'computerAndNaturalLanguages.xml').declareNamespace(human: 'Natural')

print "Languages: "
println languages.language.collect { it.@name }.join(', ')

print "Natural languages: "
println languages.'human:language'.collect { it.@name }.join(', ')

```

使用这段代码提取出的数据如下：

```

Languages: Java, Groovy, Erlang, English, German, French
Natural languages: English, German, French

```

对于较大的XML文档，我们更愿意使用`XMLSlurper`。它执行惰性求值，所以内存使用比较友好，而且开销较低。

除了漂亮地解析API，相反的方向，即创建XML文档，Groovy也使其变容易了。下一节我们将看一下不同的创建方式。

## 8.2 创建 XML

在创建业务应用时，我们往往有很多原因要以XML格式呈现数据，比如保存应用状态、与Web服务通信以及表示某些配置文件等。无论需求是什么，Groovy都使创建XML文档变得非常容易了。

在生成XML时，我们可以利用Java API的一切强大之处。如果有特别喜爱的基于Java的XML处理器，比如Xerces，在Groovy中也可有使用。<sup>①</sup>如果我们已经有可用的、以特定格式创建XML文档的Java代码，而且想在我们的Groovy项目中使用它，这可能是比较好的方法。

如果想使用纯Groovy的方式创建XML文档，可以借助`GString`在字符串中嵌入表达式的能力，再加上Groovy用于创建多行字符串的设施。对于创建我们可能在代码和测试中需要的小型XML片段，这一设施非常有用。下面是一个简单的例子（更多细节，参见5.3节）：

### WorkingWithStrings/CreateXML.groovy

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

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

下面是生成的XML文档：

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

<sup>①</sup> <http://xerces.apache.org/xerces-j>

```

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

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

```

我们也可以选择MarkupBuilder或StreamingMarkupBuilder，从任意源创建XML格式的数据输出。这是Groovy应用偏爱的方法，因为生成器提供的便捷性使创建XML文档变得非常容易。我们不必搞一堆乱七八糟的复杂API或字符串操作，普通、简单的Groovy就够了。我们再来看一个简单的例子（关于使用MarkupBuilder和StreamingMarkupBuilder的详细信息，参见17.1节）：

#### UsingBuilders/BuildUsingStreamingBuilder.groovy

```

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

xmlDocument = new groovy.xml.StreamingMarkupBuilder().bind {
    mkp.xmlDeclaration()
    mkp.declareNamespace(computer: "Computer")
    languages {
        comment << "Created using StreamingMarkupBuilder"
        langs.each { key, value ->
            computer.language(name: key) {
                author (value)
            }
        }
    }
}
println xmlDocument

```

这段代码生成的XML文档如下：

```

<?xml version="1.0"?>
<languages xmlns:computer='Computer'>
    <!--Created using StreamingMarkupBuilder-->
    <computer:language name='C++'>
        <author>Stroustrup</author>
    </computer:language>
    <computer:language name='Java'>
        <author>Gosling</author>
    </computer:language>
    <computer:language name='Lisp'>
        <author>McCarthy</author>
    </computer:language>
</languages>

```

如果我们的数据保存在数据库中或Microsoft Excel文件中，可以结合将在第9章中介绍的技术来处理。一旦从数据库中取出数据，就可以应用这里讨论的任何一种方式将其插入到文档中了。

在这一章中，我们知道了Groovy是如何辅助解析XML文档的。Groovy的使用使XML处理变得可以忍受了。如果我们的用户不喜欢维护XML配置文件（谁又喜欢呢），他们可以创建并维护基于Groovy的DSL，再将DSL转为底层框架和库期望的XML格式。如果我们处于接收XML的一端，则可以依赖Groovy为我们提供XML数据的对象表示。

一旦手上有了数据，我们就知道如何使用Groovy将其以XML格式呈现出来。贯穿本书，有很多地方会深入探讨这些主题，后面我们将看到更详细的代码示例。下一章，我们学习如何使用Groovy代码从数据库中获取数据。

# 使用数据库

9

我有一个要频繁更新的远程数据库。通过浏览器访问相当慢，不过我已经把更新过程自动化了。我不太情愿用普普通通的Java代码干这个活，因为学不到什么激动人心的东西或新东西。只是在碰到Groovy SQL ( GSQL ) 之前，我得那么做。现在，我的更新是自动化的，而且快速、毫不费力。使用GSQL，更新脚本中的数据要多于代码，信噪比很高。

我们经常会用到数据库，但是很快就会让人感觉乏味无聊。GSQL是JDBC ( Java Database Connectivity, Java数据库连接 ) 的包装器，为轻松访问数据提供了很多便捷方法。全部使用Groovy语法，我们可以快速创建SQL查询，然后使用内建的迭代器遍历结果。

这一章我们就来探索GSQL的力量。你将学到编写SQL select查询、从结果生成XML数据、执行数据的插入和更新，还会看到访问Excel文件中数据的方法。

## 9.1 创建数据库

在本章的例子中，我们将使用MySQL；不过我们可以使用任何能够通过JDBC访问的数据库。首先创建将使用于例子中的数据库，同时创建一个名为weather的表。该表中包含的是某些城市的名字和温度值。

使用自动化脚本设置数据库要比手动设置容易。因此，我们创建一个SQL脚本来构建数据库：

```
create database if not exists weatherinfo;
use weatherinfo;

drop table if exists weather;

create table weather (
    city varchar(100) not null,
    temperature integer not null
);

insert into weather (city, temperature) values ('Austin', 48);
insert into weather (city, temperature) values ('Baton Rouge', 57);
insert into weather (city, temperature) values ('Jackson', 50);
```

```
insert into weather (city, temperature) values ('Montgomery', 53);
insert into weather (city, temperature) values ('Phoenix', 67);
insert into weather (city, temperature) values ('Sacramento', 66);
insert into weather (city, temperature) values ('Santa Fe', 27);
insert into weather (city, temperature) values ('Tallahassee', 59);
```

在这个脚本中，我们为一个命名为weather的表定义了模式，并填充了一些示例数据。将该脚本保存到一个名为createdb.sql的文件中，然后使用mysql--user=root < createdb.sql命令来运行该脚本，创建数据库。

现在数据库准备好了；接下来，我们看一下从Groovy代码访问数据库的不同方式。

## 9.2 连接到数据库

要连接到数据库，只需要调用static的newInstance()方法，创建一个groovy.sql.Sql类的实例。该方法有个版本接收数据库URL、用户ID、密码和数据库驱动的名字作为参数。如果有已经有了一个java.sql.Connection实例，或者有一个java.sql.DataSource实例，就可以使用Sql类的接受相应类型的构造器，而不是使用newInstance()。

可以通过调用Sql实例的getConnection()方法（connection属性）获取连接相关信息。在处理结束后，可以通过调用close()方法关闭该连接。下面是一个例子，演示了如何连接到我们为本章创建的数据库：

### WorkingWithDatabases/Weather.groovy

```
def sql = groovy.sql.Sql.newInstance('jdbc:mysql://localhost:3306/weatherinfo',
                                      userid, password, 'com.mysql.jdbc.Driver')

println sql.connection.catalog
```

该代码报告的数据库名如下：

```
weatherinfo
```

## 9.3 数据库的 Select 操作

我们可以使用Sql对象方便地迭代一个表中的数据。只需要调用eachRow()方法，为其提供要执行的SQL查询，同时提供用于处理每行数据的闭包，就是这样：

9

### WorkingWithDatabases/Weather.groovy

```
println "City          Temperature"
sql.eachRow('SELECT * from weather') {
    printf "%-20s%5s\n", it.city, it[1]
}
```

使用前面代码取到的数据如下所示：

City	Temperature
Austin	48
Baton Rouge	57
Jackson	50
Montgomery	53
Phoenix	67
Sacramento	66
Santa Fe	27
Tallahassee	59

我们让eachRow()在weather表上执行SQL查询，处理该表的所有行。之后对每一行进行迭代（正如名字中的each所示）。还有更Groovy风格的写法，我们可以使用eachRow()提供的GroovyResultSet对象来访问表中的列，可以直接使用列名（如it.city），也可以使用索引（如it[1]）。

在前面的例子中，输出的头部是硬编码的。如果能从数据库中获取，那会更好一些。eachRow()的另一个重载版本会做这么做。它接收两个闭包，一个用于元数据，另一个用于数据。元数据的闭包仅调用一次（在SQL语句执行之后），并以一个ResultSetMetaData实例为参数，而另一个闭包会对结果中的每一行调用一次。我们通过下面的代码尝试一下：

#### WorkingWithDatabases/Weather.groovy

```
processMeta = { metaData ->
    metaData.columnCount.times { i ->
        printf "%-21s", metaData.getColumnLabel(i+1)
    }
    println ""
}

sql.eachRow('SELECT * from weather', processMeta) {
    printf "%-20s %s\n", it.city, it[1]
}
```

输出中显示了使用元数据创建的头部，后面是各行数据：

city	temperature
Austin	48
Baton Rouge	57
Jackson	50
Montgomery	53
Phoenix	67
Sacramento	66
Santa Fe	27
Tallahassee	59

如果想处理所有行，但是不想使用迭代器，我们可以在Sql实例上使用rows()方法。该方法返回一个结果数据的ArrayList实例，如下所示：

**WorkingWithDatabases/Weather.groovy**

```
rows = sql.rows('SELECT * from weather')

println "Weather info available for ${rows.size()} cities"
```

这段代码的输出如下：

```
Weather info available for 8 cities
```

如果改为调用`firstRow()`方法，则仅得到结果的第一行。我们可以使用`Sql`的`call()`方法执行存储过程。使用`withStatement()`方法，可以设置一个将在查询执行之前调用的闭包。如果想在执行之前拦截并修改SQL查询，该方法会有所帮助。

## 9.4 将数据转为 XML 表示

我们可以从数据库中获得数据，然后使用Groovy生成器创建数据的不同表示。下面这个例子演示了如何创建`weather`表中数据的一个XML表示（参见17.1节）：

**WorkingWithDatabases/Weather.groovy**

```
bldr = new groovy.xml.MarkupBuilder()

bldr.weather {
    sql.eachRow('SELECT * from weather') {
        city(name: it.city, temperature: it.temperature)
    }
}
```

这段代码输出的XML如下：

**WorkingWithDatabases/Weather.output**

```
<weather>
    <city name='Austin' temperature='48' />
    <city name='Baton Rouge' temperature='57' />
    <city name='Jackson' temperature='50' />
    <city name='Montgomery' temperature='53' />
    <city name='Phoenix' temperature='67' />
    <city name='Sacramento' temperature='66' />
    <city name='Santa Fe' temperature='27' />
    <city name='Tallahassee' temperature='59' />
</weather>
```

几乎毫不费力，Groovy和GSQL就帮我们创建了数据库中数据的一个XML表示。

## 9.5 使用 DataSet

在9.3节中，我们看到了如何处理执行一条Select查询所获得的结果集。如果想接收的只是一个过滤后的一些行，比如只要温度值低于32<sup>①</sup>的城市，我们可以相应地设置查询。作为一种选择，我们可以将结果接收为一个groovy.sql.DataSet，以此过滤数据。我们进一步看一下。

Sql类的dataSet()方法接收一个表名，并返回一个虚拟代理——直到迭代时，它才去取实际的行。之后我们可以使用DataSet的each()方法（就像Sql的eachRow()方法）在行上迭代。不过在下面的代码中，我们将使用findAll()方法来过滤结果，只获得温度在零下的城市。当我们调用findAll()时，DataSet会通过基于我们提供的select谓词确定的专用查询做进一步提炼。直到我们在获得的对象上调用each()方法时，才会去取实际的数据。因此，DataSet非常高效，仅提取选中的数据。

### WorkingWithDatabases/Weather.groovy

```
dataSet = sql.dataSet('weather')
citiesBelowFreezing = dataSet.findAll { it.temperature < 32 }
println "Cities below freezing:"
citiesBelowFreezing.each {
    println it.city
}
```

使用本节介绍的DataSet，这段代码的输出如下：

```
Cities below freezing:  
Santa Fe
```

## 9.6 插入与更新

我们可以使用DataSet对象来添加数据，而不仅仅是过滤数据。add()方法接收一个数据的Map，用其中的数据创建一行，如下列代码所示：

### WorkingWithDatabases/Weather.groovy

```
println "Number of cities : " + sql.rows('SELECT * from weather').size()
dataSet.add(city: 'Denver', temperature: 19)
println "Number of cities : " + sql.rows('SELECT * from weather').size()
```

下面输出说明了这段代码的执行效果：

```
Number of cities : 8  
Number of cities : 9
```

<sup>①</sup> weather表中保存的温度是华氏温度，与摄氏温度的换算关系为：摄氏温度=（华氏温度-32）\*5/9。因此华氏温度低于32相当于摄氏温度的零下。——译者注

然而更传统的方式是使用Sql类的execute()或executeInsert()方法，如下所示：

```
WorkingWithDatabases/Weather.groovy
temperature = 50
sql.executeInsert("""INSERT INTO weather (city, temperature)
    VALUES ('Oklahoma City', ${temperature})""")
println sql.firstRow(
    "SELECT temperature from weather WHERE city='Oklahoma City'")
```

前面代码的输出如下：

```
[temperature:50]
```

通过发出相应的SQL命令，也可以以类似方式执行更新和删除操作。

## 9.7 访问 Microsoft Excel

我们还可以使用Sql类来访问Microsoft Excel。如果想了解与COM或ActiveX交互的信息，可以看一下Groovy的Scriptom应用编程接口。<sup>①</sup>在这一节中，除了用Excel替换掉MySQL，我们将使用已经见过的东西创建一个非常简单的例子。首先，创建一个名为weather.xlsx（如果是老版本的Excel，则是weather.xls）的Excel文件。

把该文件创建在c:\temp目录下。文件中将包含一个名为temperatures的工作表（见工作表底部），数据内容如图9-1所示。访问Excel的代码如下。

	A	B	C	D	E	F	G
1	City	Temperature					
2	Denver	19					
3	Boston	12					
4	New York	22					
5							
6							

图9-1 我们要使用GSQL访问的一个Excel文件

<sup>①</sup> <http://groovy.codehaus.org/COM+Scripting>

**WorkingWithDatabases/Excel/Windows/AccessExcel.groovy**

```
def sql = groovy.sql.Sql.newInstance(  
    """jdbc:odbc:Driver=  
    {Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)};  
    DBQ=C:/temp/weather.xlsx;READONLY=false""", '', '')  
  
println "City\tTemperature"  
sql.eachRow('SELECT * FROM [temperatures$]') {  
    println "${it.city}\t${it.temperature}"  
}
```

下面是使用前面的代码从Excel文件中获取到的数据：

City	Temperature
Denver	19.0
Boston	12.0
New York	22.0

在调用`newInstance()`时，我们指定了Excel的驱动和Excel文件的位置。也可以不这么做，愿意的话，可以为Excel文件设置一个数据源名（Data-Source Name，DSN），并使用古老的JDBC-ODBC驱动桥。

如果那样做的话，我们就不用将文件位置写在代码里了。相反，我们要在Windows上配置数据源名（DSN）。其余执行查询和处理结果的代码都是耳熟能详的了。

在这一章中，我们使用了GSQL来访问关系数据库。对于数据访问，我们可以从该API的简单但强大的功能中受益。只需要几行代码，几分钟的时间，我们的应用就可以读写真正的数据了。

经过这么多章，我们学习了很多API和Groovy编程技术。Groovy的关键优势之一，在于它能够与Java集成和共存。下一章，我们将探讨如何在这两门语言间集成代码。

# 使用脚本和类

# 10

就算不是最流行的，Java也是最流行的主流企业级语言之一。尽管Groovy可以单独使用，但是我们很可能会混合使用Groovy和Java。在使用Groovy的项目中，Java代码一起演进是这种使用场景很常见。学习如何混合使用以这些语言编写的代码，能够帮助我们在应用中快速采用Groovy。

在Groovy中调用Java代码，非常简单、直接。而乍看上去，在Java中调用Groovy代码就显得没那么简单了。Groovy方法可以接受闭包，Groovy类可以有动态方法，也就是在运行时才出现的方法。Java中是不是可以访问这些东西呢，如果可能的话，又有多困难？我们脑海中蹦出了很多问题。本章将回答这些问题。

我们将看到如何联合编译Java和Groovy代码，如何在Java中使用Groovy代码，以及如何在Java中创建Groovy闭包。我们还将探索如何在Java代码中调用Groovy动态方法，都不用费什么劲。

### 10.1 Java 和 Groovy 的混合

在应用中，我们可以在一个Java类、一个Groovy类或者一个Groovy脚本中实现某个特定功能。之后可以在Java类、Groovy类或Groovy脚本中调用该功能。图10-1展示了混合使用Java类、Groovy类和Groovy脚本的各种选择。

要在Groovy代码中使用Groovy类，无需做什么，直接就可以工作。我们只需要确保所依赖的类在类路径(classpath)下，要么是源代码，要么是字节码。要把一个Groovy脚本拉到我们的Groovy代码中，可以使用GroovyShell。而如果要在Java类中使用Groovy脚本，则可以使用JSR 223提供的ScriptEngine API。如果想在Java中使用Groovy类，或者想在Groovy中使用Java类，我们可以利用Groovy的联合编译(joint-compilation)工具。这些都非常简单，本章接下来将一一介绍。

首先，来看一下运行Groovy的各种方法。然后再看一下如何在Java和Groovy中混合使用Groovy的类和脚本。

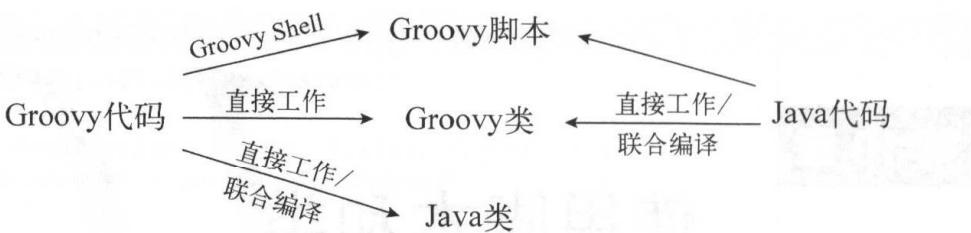


图10-1 混合使用Java类、Groovy类和脚本的方法

## 10.2 运行Groovy代码

要运行Groovy代码，我们有两个选择。一个是对源代码运行groovy命令。Groovy会自动在内存中编译代码并执行。我们不必引入一个明确的编译步骤。

第二个方法，如果想用类似Java的那种更传统的方法，即显式的编译代码来创建字节码(.class文件)，可以使用groovyc编译器。要执行编译好的字节码，可以像执行编译好的Java代码那样使用java命令。唯一的区别是，我们需要把groovy-all-2.1.0.jar文件放在classpath下。记得在classpath里添加一个句点(.)，这样java命令就可以找到当前目录下的类了。这一Java存档文件(JAR)在GROOVY\_HOME下的embeddable目录中。作为一个例子，假设我们有一个名为Greet.groovy的文件，其中的代码如下：

### ClassesAndScripts/Greet.groovy

```
println(['Groovy', ' Rocks!'].join(' '))
```

要运行它，只需要输入groovy Greet。然而，如果想显式地将其编译为Java字节码，则要输入groovyc Greet.groovy来创建一个名为Greet.class的文件，文件名正和我们预料的一样。如果代码中有包声明，则该文件会遵照Java的包目录格式在相应的目录下创建。与Groovy类不同的是，Groovy脚本通常没有包声明。可以使用-d选项指定非当前目录的目标目录。输入如下命令，可以运行生成的字节码：

```
java -classpath $GROOVY_HOME/embeddable/groovy-all-2.1.0.jar:. Greet
```

在Windows上，请用%GROOVY\_HOME%替换掉\$GROOVY\_HOME。输出如下：

```
Groovy Rocks!
```

这些步骤说明，可以以字节码形式编译和分发我们的Groovy代码，这与编译和分发Java代码很像。我们可以将其发布为.class文件，或者打包成JAR文件。java命令看不出区别。如果部署设置有要求的话，可以以这种方法将Groovy代码以字节码形式与项目中的其他字节码一起发布。

下面我们将看一些混合使用Groovy脚本和类的方法。

## 10.3 在 Groovy 中使用 Groovy 类

要在Groovy代码中使用一个Groovy类，只需要确保该类在我们的classpath下。可以使用Groovy源代码，也可以把源代码编译成.class文件并使用该文件——随我们选择。当我们的Groovy代码引用了一个Groovy类时，Groovy会以该类的名字在我们的classpath下查找.groovy文件；如果找不到，则以同样的名字查找.class文件。

假设有一个Groovy源代码文件Car.groovy，内容如下所示，它放在src目录下：

```
ClassesAndScripts/src/Car.groovy
class Car
{
    int year = 2008
    int miles

    String toString() { "Car: year: $year, miles: $miles" }
}
```

再假设我们会在一个名为useCar.groovy的文件中使用该类，像下面这样：

```
ClassesAndScripts/useCar.groovy
```

```
println new Car()
```

要使用这个类，我们输入groovy -classpath src useCar。它会自动取到Car类的源文件，编译它，创建一个实例，然后生成输出：

```
Car: year: 2008, miles: 0
```

如果我们放的是Car的字节码，而不是源代码，步骤是一样的——Groovy可以方便地使用来自.groovy或.class文件中的类。

如果打算在项目中混合使用Groovy和Java，我们可以借助Groovy提供的联合编译工具，下一节将会看到。

## 10.4 利用联合编译混合使用 Groovy 和 Java

如果Groovy类是预先编译好的，那我们就可以方便地在Java中使用.class文件或JAR包。来自Java的字节码和来自Groovy的字节码，对Java而言没什么区别；我们必须把Groovy JAR文件（前面讨论过）放在我们的classpath下，类似于我们使用Spring、Hibernate或其他框架/类库的JAR文件时的做法。

如果我们只有Groovy源代码，而非字节码，又会怎样呢？请记住，当我们的Java类依赖其他Java类时，如果没有找到字节码，javac将编译它认为必要的任何Java类。不过javac对Groovy

可没这么友好。幸好groovyc支持联合编译。当我们编译Groovy代码时，它会确定是否有任何需要编译的Java类，并负责编译它们。因此我们可以自由地在项目中混合使用Java代码和Groovy代码，而且不必执行单独的编译步骤。简单地调用groovyc就好。

要利用联合编译，我们需要使用-J编译标志。使用-J前缀把标志传给Java编译器。例如，假定我们有一个类，保存在AJavaClass.java文件中：

```
ClassesAndScripts/AJavaClass.java
//Java代码
public class AJavaClass {
{
    System.out.println("Created Java Class");
}

public void sayHello() { System.out.println("hello"); }
}
```

我们还有一个保存在UseJavaClass.groovy文件中的脚本，它使用了这个Java类：

```
ClassesAndScripts/UseJavaClass.groovy
new AJavaClass().sayHello()
```

要联合编译这两个文件，我们输入这个命令：groovyc -j AJavaClass.java UseJavaClass.groovy -Jsource 1.6。-Jsource 1.6会把可选的选项source = 1.6发送给Java编译器。使用javap检查生成的字节码，会发现AJavaClass作为一个普通的Java类，扩展了java.lang.Object，而UseJavaClass扩展了groovy.lang.Script。

执行这段代码，确认一切正常。尝试下列命令：

```
java -classpath $GROOVY_HOME/embeddable/groovy-all-2.1.0.jar:. UseJavaClass
```

我们应该会看到如下输出：

```
Created Java Class
hello
```

可以与Java无缝地在项目中混用，使Groovy成为企业级应用中可以与Java漂亮集成的一种奇妙语言。我们可以将精力集中在使用每种语言的优点，而不必忙于与任何集成问题做斗争。集成不仅能在简单情况下简化操作；对于使用了在Java没有直接支持的特性的Groovy代码，我们也可以从Java中调用，下一节我们将看到。

## 10.5 在Java中创建与传递Groovy闭包

Groovy从诞生的第一天起就支持闭包，但Java还没有认真对待这一理念。令人惊讶的是，得

得益于Groovy的动态特性，在Java中创建闭包和调用闭包的Groovy方法非常简单。鉴于Java坚持让我们发送恰当类型的方法实例，Groovy非常友好，而且对于我们使用其特性感到非常开心。

如果仔细检查，我们会发现，当Groovy调用一个闭包时，它只是使用了一个名为call()的特殊方法。要在Java中创建一个闭包，我们只需要一个包含该方法的类。如果Groovy代码要向闭包传递实参，我们必须确保call()方法接受这些实参作为形参。

在下面的例子中我们将看到，在Java中创建并传递闭包非常简单。我们创建一个Groovy类——AGroovyClass，其中有两个接受闭包的方法：

#### ClassesAndScripts/AGroovyClass.groovy

```
class AGroovyClass {
    def useClosure(closure) {
        println "Calling closure"
        closure()
    }

    def passToClosure(int value, closure) {
        println "Simply passing $value to the given closure"
        closure(value)
    }
}
```

useClosure()方法会打印一条消息并调用所提供的闭包。passToClosure()方法会将它接收到的第一个形参传递给所提供的闭包。

要在Java中调用useClosure()方法，我们需要提供一个实现了call()方法的实例，像这样：

#### ClassesAndScripts/UseAGroovyClass.java

```
//Java代码
public class UseAGroovyClass {
    public static void main(String[] args) {
        AGroovyClass instance = new AGroovyClass();
        Object result = instance.useClosure(new Object() {
            public String call() {
                return "You called from Groovy!";
            }
        });

        System.out.println("Received: " + result);
    }
}
```

Java和Groovy代码既可以联合编译，也可以分别编译。要联合编译，我们使用groovyc -j UseAGroovyClass.java AGroovyClass.groovy命令。然后我们可以使用java -classpath \$GROOVY\_HOME/embeddable/groovy-all-2.1.0.jar:. UseAGroovyClass命令运行Java代码。

我们在Java中创建的匿名类的实例会被无缝地传递给Groovy，反过来，调用也会回到该匿名类：

```
Calling closure
Received: You called from Groovy!
```

调用一个接受形参的闭包也没有很大不同，我们会在`passToClosure()`方法中的调用中看到：

#### ClassesAndScripts/UseAGroovyClass2.java

```
//Java代码
System.out.println("Received: " +
    instance.passToClosure(2, new Object() {
        public String call(int value) {
            return "You called from Groovy with value " + value;
        }
    }));
}
```

Java中这一版的`call()`方法接收一个形参，`passToClosure()`方法会在Groovy段给该参数赋一个值，我们在输出中会看到：

```
Simply passing 2 to the given closure
Received: You called from Groovy with value 2
```

我们必须确保`call()`方法接受恰当数目和类型的形参。剩下的具体细节，Groovy会为我们处理。

本节探讨了在Java中调用Groovy闭包。反方向的调用也是如此简单。在[http://www.jroller.com/melix/entry/coding\\_a\\_groovy\\_closure\\_in](http://www.jroller.com/melix/entry/coding_a_groovy_closure_in)中，Cédric Champeau演示了如何把一个Java方法当作Groovy端的一个闭包。

我们看过了如何调用带闭包的方法；下面来看一下如何从Java中调用Groovy动态方法。

## 10.6 在Java中调用Groovy动态方法

Groovy可以在运行时创建方法（第三部分将会介绍）。这些方法不能从Java中直接调用，因为在编译时，这些代码在字节码中还不存在。它们在运行时产生，但是如果我们要从Java中调用它们，需要在编译时编写调用语句（或者使用反射）。要调用动态方法，我们必须先通过Java编译器的编译，然后运行时才能进行分派。这听上去很复杂，但是要相信Groovy！

每个Groovy对象都实现了`GroovyObject`接口，该接口有一个叫作`invokeMethod()`的特殊方法。该方法接受要调用的方法的名字，以及要传递的参数。在Java这一端，`invokeMethod()`可以用来调用Groovy中使用元编程动态定义的方法。

我们通过例子来看一下。创建一个Groovy类，其中包含一个特殊方法——`method Missing()`，当某个不存在的方法被调用时，该方法会介入。

**ClassesAndScripts/DynamicGroovyClass.groovy**

```
class DynamicGroovyClass {
    def methodMissing(String name, args) {
        println "You called $name with ${args.join(', ')}."
        args.size()
    }
}
```

这个类完全是动态的，除了`methodMissing()`，它没有实际的方法。因为这个类可以接受任何方法调用，所以我们几乎可以在它上面调用任何方法。要从Java端调用我们期望的方法，可调用`invokeMethod()`，并将方法的名字和一个由参数组成的数组传给它，如下面的示例所示。

**ClassesAndScripts/CallDynamicMethod.java**

```
public class CallDynamicMethod {
    public static void main(String[] args) {
        groovy.lang.GroovyObject instance = new DynamicGroovyClass();

        Object result1 = instance.invokeMethod("squeak", new Object[] {});
        System.out.println("Received: " + result1);

        Object result2 =
            instance.invokeMethod("quack", new Object[] {"like", "a", "duck"});
        System.out.println("Received: " + result2);
    }
}
```

我们创建了一个`DynamicGroovyClass`实例，并将其赋值给了一个`GroovyObject`类型（所有Groovy对象都支持）的引用。使用这个引用，我们可以在该类上调用任何方法，包括动态的和预定义的。一旦Groovy接收到这些方法，它会通过常规的Groovy方法分派过程来处理方法调用，11.1节会介绍此过程。Groovy会响应我们从Java端进行的调用，从下面的输出可以看出。`invokeMethod()`会将被调用方法返回的任何响应信息返回到Java端。

```
You called squeak with .
Received: 0
You called quack with like, a, duck.
Received: 3
```

如果Groovy因为某些原因无法执行被调用的方法，或者该方法没有执行成功，调用`invokeMethod()`将会失败。请准备好处理该方法可能会抛出的异常。

从Java中可以使用任何Groovy类，这点没有限制，不管它们是否是动态的。下面我们来看一下从Java中使用Groovy类的情况。

## 10.7 在Groovy中使用Java类

在Groovy中使用Java类简单且直接。如果我们想使用的Java类是JDK的一部分，可以像在Java中那样导入这些类或它们的包。Groovy默认会导入很多包和类（参见2.1节），因此，如果我们想使用的类已经导入（比如`java.util.Date`），直接用就可以了，无需再导入。

如果想使用一个自己的Java类，或者不是标准JDK中的类，在Groovy中可以像在Java中那样导入它们。请确保导入了必要的包或类，或者使用类的全限定名来引用它们。当运行groovy时，可以使用`-classpath`选项指定`.class`文件或JAR包的路径。如果类文件和我们的Groovy代码在同一目录下，则不需要通过该选项指定目录。

我们看一个例子。假定我们有一个名为`GreetJava`的Java类，它从属于`com.agiledeveloper`包，而且有一个`static`方法`sayHello()`，如下所示：

```
ClassesAndScripts/GreetJava.java
// Java代码
package com.agiledeveloper;

public class GreetJava {
    public static void sayHello() {
        System.out.println("Hello Java");
    }
}
```

我们想从一个Groovy脚本中调用该方法，因此首先编译Java类`GreetJava`，这样会在`./com/agiledeveloper`目录下生成类文件`GreetJava.class`，其中的句点（`.`）是当前目录。然后在`UseGreetJava.groovy`文件中创建一个Groovy脚本，内容如下：

```
ClassesAndScripts/UseGreetJava.groovy
com.agiledeveloper.GreetJava.sayHello()
```

要运行该脚本，只需要输入`groovy UseGreetJava`。该脚本会顺利运行，并调用`GreetJava`类中的`sayHello()`方法，如下列输出所示：

```
Hello Java
```

如果类文件不在当前目录下，我们仍然可以使用它，只是需要记得设置`-classpath`选项。假定类文件`GreetJava.class`位于`~/release/com/agiledeveloper`目录下，其中`~`是我们的home目录。

要运行前面提到的Groovy脚本（`UseGreetJava.groovy`），请使用下面的命令：

```
$groovy -classpath ~/release UseGreetJava
```

在这个例子中，我们显式地编译了Java代码，然后在Groovy脚本中使用了字节码。如果想显

式地编译Groovy代码，不必分别编译Java和Groovy。可以使用联合编译代替。

并非所有的Groovy代码都需要显式编译。Groovy脚本一般通过groovy命令使用。下面我们将介绍如何混合使用Groovy脚本。

## 10.8 从 Groovy 中使用 Groovy 脚本

Groovy脚本保存的语句和表达式不必局限于源代码中某个特定类。我们可以直接使用groovy命令执行这些脚本。通过GroovyShell类，我们还可以从其他的Groovy脚本和类中调用它们。下面看一个例子：

**ClassesAndScripts/Script1.groovy**

```
println "Hello from Script1"
```

这里有一个命名为Script1.groovy的文件，我们想将其作为另一个脚本——Script2.groovy——的一部分来执行，如下所示：

**ClassesAndScripts/Script2.groovy**

```
println "In Script2"
shell = new GroovyShell()
shell.evaluate(new File('Script1.groovy'))

// 或是更简单点
evaluate(new File('Script1.groovy'))
```

输出如下：

```
In Script2
Hello from Script1
Hello from Script1
```

使用GroovyShell，我们可以在任何文件（或字符串）中对脚本调用evaluate()方法，以执行该脚本。这很容易。但是（凡事总有例外），如果我们想向脚本传递一些参数，又该怎么办呢？

**ClassesAndScripts/Script1a.groovy**

```
println "Hello ${name}"
name = "Dan"
```

这个脚本需要一个变量name。我们可以使用一个Binding实例来绑定变量，如下所示：

**ClassesAndScripts/Script2a.groovy**

```
println "In Script2"

name = "Venkat"

shell = new GroovyShell(binding)
```