

在开始编写Groovy代码之前，首先需要安装Groovy。本章将介绍如何快速安装Groovy，并确保一切工作正常。现在处理好这些基础任务，有助于我们向后面更有意思的内容快速迈进。

1.1 安装 Groovy

获得一份稳定、可用的Groovy副本非常简单：只需访问Groovy主页<http://groovy.codehaus.org>，点击下载链接。可以看到有二进制版本和源代码版本供我们下载：如果想在本地构建Groovy，或者想研究其源代码，请下载源代码版本；否则请下载二进制版本。Windows用户也可以下载Windows安装程序版本。建议访问下载页面的同时，把Groovy的文档也下载下来。

对于加入了Groovy用户邮件列表的前卫程序员，前面提到的版本是不够的。他们需要的是Groovy语言实现^①的最新预发布版本，可以从<http://groovy.codehaus.org/Git>获取快照版本。

还需要JDK 1.5或更高版本，所以请确保本地系统上已经安装了Java^②。

下面来安装Groovy。

1.1.1 在 Windows 系统环境安装 Groovy

我们可以使用针对Windows的一键式安装程序，运行安装程序并按照说明操作即可。希望对安装过程施加更多控制的程序员，可以使用二进制发布包。

下一步，必须设置GROOVY_HOME环境变量和路径。进入“控制面板”，打开“系统”，编辑“系统变量”。创建一个名为GROOVY_HOME的环境变量，然后将其设置为Groovy目录的位置（例如，我将其设置为C:\programs\groovy\groovy-2.1.0）。此外，将%GROOVY_HOME%\bin添加到Path环境变量中，以此把Groovy的bin目录加入到查找路径中。记得路径中的目录以分号（;）分隔。

再下一步，确认环境变量JAVA_HOME指向的是Java开发包（Java Development Kit，JDK）的

^①这里指支撑Groovy语言的编译器和运行时等。——译者注

^② <http://java.sun.com/javase/downloads/index.jsp>

位置。如果不是，请设置。

这就完成了在Windows系统环境的安装。记得关闭所有打开的命令行窗口，因为对环境变量的修改需要重启命令行窗口才会生效。在新的命令行窗口中，输入`groovy -v`，确保报告的是正确的版本。

1.1.2 在类 Unix 系统环境安装 Groovy

解压下载的二进制发布包。同时访问<http://groovy.codehaus.org/Download>，查看是否有针对不同Unix变种的特殊发布版本和说明。将`groovy-2.1.0`目录移到想放的位置。例如，在我的Mac系统上，我将其放在了`/opt/groovy`目录。

下一步，设置`GROOVY_HOME`环境变量与路径。根据所用Shell的不同，需要编辑不同的配置文件。你可能已经知道去哪里找那些配置文件；如果不知道，请参考相应文档。我在OS X上使用bash，所以我编辑的是`~/.bash_profile`文件。在这个文件中，我添加了一项：`export GROOVY_HOME="/opt/groovy/groovy-2.1.0"`，以此设置环境变量`GROOVY_HOME`。我还把`$GROOVY_HOME/bin`添加到了`PATH`环境变量中。

再下一步，确认环境变量`JAVA_HOME`指向的是JDK目录所在位置；如果不是，请设置。`ls -l `which java``这条命令可以帮助确定Java的安装位置。

完成了Groovy的安装，就为使用这门语言做好了基本准备工作。关闭所有打开的终端窗口，因为对环境变量的修改需要重启终端才会生效。也可以使用`source`命令执行一下配置文件，但打开新窗口简单明了。在一个新的命令行窗口中，输入`groovy -v`命令，确保报告的是正确的版本。这就够了！

1.2 管理多个版本的 Groovy

面对不同的项目，经常需要使用同一语言的多个版本。如果不小心，为项目管理正确的语言版本这个任务可能很快就会变成消耗时间的无底洞。GVM (Groovy enVironment Manager) 不仅可以管理Groovy语言的版本，还可以管理与Groovy相关的库和工具（如Grails、Griffon和Gradle等）的版本。

这款工具很容易安装，而且支持各种*nix系统，在Windows系统环境也可以通过Cygwin支持^①。一旦安装了GVM，只需简单地运行`gvm list groovy`命令，就可以看到可用的和已安装的Groovy语言版本。如果想使用Groovy的某个特定版本，也可以指定。例如，要运行本书中的示例，可以输入`gvm install groovy 2.1.1`命令。GVM之后会下载并安装该版本，以供使用。如果已经安装了Groovy的多个版本，要切换到2.1.1版本，则可以使用`gvm use groovy 2.1.1`命令。

① <http://gvmtool.net>

1.3 使用 groovysh

我们已经安装了Groovy，并且核对了版本，现在就来试用吧。命令行工具groovysh是把玩Groovy最为快速的方式之一。打开一个终端窗口，输入groovysh，如下所示，我们会看到一个shell。输入一些Groovy代码，看看它是如何工作的。

```
> groovysh
Groovy Shell (2.1.1, JVM: 1.7.0_04-ea)
Type 'help' or '\h' for help.

-----
groovy:000> Math.sqrt(16)
====> 4.0
groovy:000> println 'Test drive Groovy'
Test drive Groovy
====> null
groovy:000> String.metaClass.isPalindrome = {
groovy:001>   delegate == delegate.reverse()
groovy:002> }
====> groovysh_evaluate$_run_closure1@64b99636
groovy:000> 'mom'.isPalindrome()
====> true
groovy:000> 'mom'.l

lastIndexOf(leftShift(length())
groovy:000> 'mom'.l
```

groovysh是以交互方式尝试一些小型Groovy代码例子的好工具。它也可以用于在编码过程中实验一些代码。然而需要注意的是，groovysh有些特殊之处。如果在使用该命令时遇到问题，可以使用save命令把代码保存到一个文件中，然后尝试使用groovy命令从命令行运行，以避免任何与工具有关的问题。一按回车键，groovysh命令就会编译并执行输入完的语句，打印代码执行过程中的所有输出，并打印这条语句的执行结果。

例如，输入Math.sqrt(16)，它会打印结果——4.0。然而，如果输入println 'Test drive Groovy'，打印的则是引号中的字符串，后面加上null，说明println()什么都没有返回。

也可以输入占据多行的代码，如果groovysh提示存在问题，只需要在每行后面加一个分号，就像定义动态方法isPalindrome()的那行代码一样。当我们输入一个类、一个方法，甚至一个if语句时，groovysh会等我们完成输入再执行那段代码。groovy:提示符后面的数字告诉我们，已经累积的要执行代码的行数。

如果不太确定要输入的命令，可以输入所知道的尽可能多的字符，然后按Tab键。shell会打印以我们输入的部分名字打头的可用方法，就像前面的groovysh交互式shell使用片段中，在输入'mom'.l后，按下Tab键，可以看到提示的内容。如果只输入一个英文句点(.)然后按下Tab键，shell会询问是否要显示所有可用方法。

输入help可以得到所支持命令的列表。可以使用向上箭头查看已经输入的命令，这对于重复前面的语句或命令非常有用。它甚至会记住我们在前面的调用中输入的命令。

使用完毕，输入exit退出该工具。

1.4 使用 groovyConsole

Groovy也有些偏爱图形用户界面（Graphical User Interface，GUI）的用户——只需要在Windows资源管理器中双击groovyConsole.bat（在%GR00VY_HOME%\bin目录下查找该文件）。类Unix系统的用户可以使用他们喜欢的文件或目录浏览工具双击groovyConsole可执行脚本。还可以在命令行输入groovyConsole来启动控制台GUI工具。控制台GUI会弹出来，如图1-1所示。

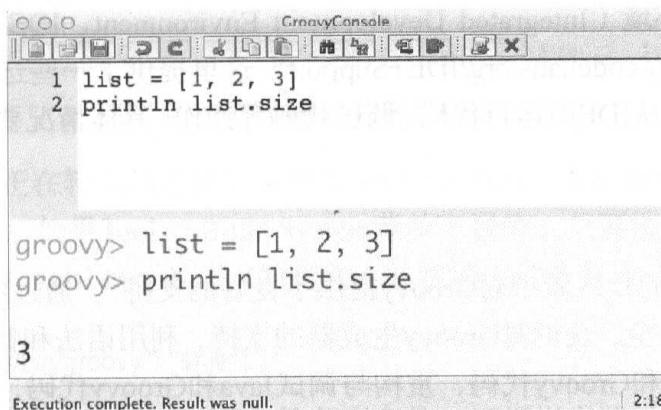


图1-1 使用groovyConsole

让我们在控制台的顶层窗口中输入一些Groovy代码。要执行代码，Windows系统用户按Ctrl+R或Ctrl+Enter组合键，Mac系统用户则按Command+R或Command+Enter组合键。

也可以通过点击相应的工具条按钮来执行脚本。随着时间的推移，groovyConsole变得越来越好，可以执行保存脚本、打开现有脚本等操作，所以花点时间探索一下这个工具吧。

1.5 在命令行中运行 Groovy

当然，对某些程序员而言，再没有比进入命令行并运行程序更令他们开心的了。我们也可以这样做，只需输入groovy命令，后面加上Groovy程序的文件名，如下所示：

```
> cat Hello.groovy
println "Hello Groovy!"
> groovy Hello
Hello Groovy!
>
```

要在命令行中直接尝试一些语句，请使用-e选项。在命令行中输入groovy -e "println

'hello'"，并敲击回车，Groovy将输出“hello”。

然而，实际上groovy命令对于执行较大的Groovy脚本和类非常有用。它希望我们输入的或者是不包含在任何类中的一些可执行代码，或者是一个带有static main(String[] args)方法（即传统的Java main()方法）的类。

如果我们的类扩展了GroovyTestCase类（参见18.2节），或者实现了Runnable接口，可以跳过main()方法。在这些情况下，如果main()方法仍然出现了，则被优先执行。

1.6 使用IDE

随着我们开始大量编写更为复杂的Groovy代码，我们将很快结束使用上述工具，并希望有一款功能齐全的集成开发环境（Integrated Development Environment，IDE）。幸运的是，我们有多种选择。参见<http://groovy.codehaus.org/IDE+Support>，这里提供了一些选择。应用这些工具，可以执行编辑Groovy代码、从IDE内运行代码、调试代码等操作（具体情况要取决于所选择的工具）。

1.6.1 IntelliJ IDEA

IntelliJ IDEA在免费的社区版中对Groovy提供了良好的支持^①。通过IntelliJ IDEA，可以编辑Groovy代码，使用代码补全，获得对Groovy生成器的支持，利用语法和错误高亮，使用代码格式化与检查，联合编译Java和Groovy代码，重构与调试Java和Groovy代码，以及在同一项目中使用Java和Groovy代码。

1.6.2 Eclipse Groovy 插件

Eclipse用户可以使用Groovy Eclipse插件^②。通过该插件，可以编辑Groovy类和脚本，利用语法高亮，以及编译、运行与测试代码。使用Eclipse调试器，可以单步进入Groovy代码或调试单元测试。此外，还可以在Eclipse内调用Groovy Shell或Groovy控制台，以便快速实验Java和Groovy代码。

1.6.3 TextMate Groovy Bundle

Mac的程序员普遍是在TextMate中使用Groovy Bundle。关于TextMate，可以参考*TextMate: Power Editing for the Mac*[Gra07]^{③,④}一书。（Windows用户可以看一下E Text Editor^⑤。另外，如果

① <http://www.jetbrains.com/idea>

② <http://groovy.codehaus.org/Eclipse+Plugin>

③ <http://docs.codehaus.org/display/GROOVY/TextMate>

④ <http://macromates.com>

⑤ <https://github.com/etexteditor/e>

编辑的是较小的代码片段，可以使用Notepad2^①。)TextMate提供了一些可以节省时间的脚本片段，支持将一些代码展开为标准的Groovy代码，比如闭包。如图1-2所示，在TextMate内，我们可以利用语法高亮，还可以快速运行Groovy代码和测试。可以阅读我的博客文章<http://blog.agiledeveloper.com/2007/10/tweaking-textmate-groovy-bundle.html>来了解如何微调Groovy Bundle以快速显示结果而不弹出窗口。

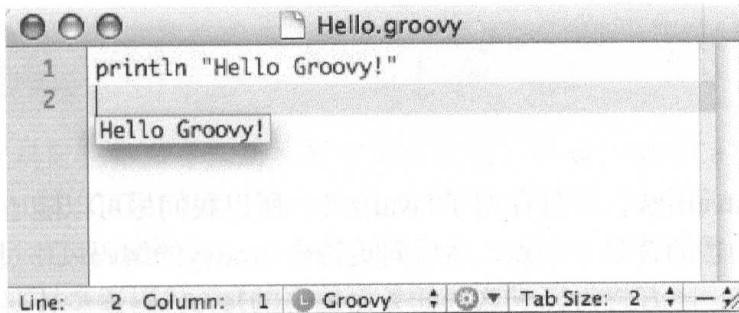


图1-2 在TextMate内执行Groovy代码

很多TextMate用户正在转向新近出的编辑器Sublime Text。要在Sublime Text内运行Groovy代码，需要一个构建脚本。如果Tools > Build System菜单下没有，只要选择New Build System...菜单项创建一个名为groovy.sublime-build的文件，并在该文件中写入一行命令：

```
{ "cmd": ["/opt/groovy/bin/groovy", "$file"] }
```

该命令要求Sublime Text以Groovy源文件名为参数，运行指定路径上的groovy命令。结果将显示在输出窗口内。要运行代码，可以按F7或Command+B。有关Sublime Text中构建配置的更多细节，请参考http://sublimetext.info/docs/en/reference/build_systems.html。

有命令行和IDE工具可供选择真好。然而还需要确定哪款工具是正确的选择。我觉得最简单的方法是，直接在编辑器或IDE内运行Groovy代码，让groovy工具在背后处理代码的编译与执行。这对我的“快速编辑、编码和运行测试”这种开发循环很有帮助。有时我发现我会跳到groovysh实验一些代码片段。但是你不必这么做。自己用着最舒服的工具就是正确的。请从适合自己的简单工具和步骤入手。感觉适应了以后，当需要时可以尝试一些较为复杂的工具。

在本章中，我们安装了Groovy，并且进行了快速测试，还一并看了一些命令行工具和IDE支持。这意味着我们已经为下一章的探索做好了准备。

^① <http://www.flos-freeware.ch/notepad2.html>

第2章

面向Java开发者的Groovy



因为Groovy支持Java语法，并且保留了Java语义，所以我们尽可以随心所欲地混用两种语言风格。本章将以我们熟悉的背景为起点，然后向更符合Groovy的编码风格过渡。即从过去常常使用Java完成的任务入手，随着将为完成这些任务而编写的Java代码逐步转变为Groovy代码，我们会看到Groovy版本更为简洁，而且更具表现力。在本章最后，我们会研究一些陷阱——如果预料不到，这些陷阱会让我们措手不及。

2.1 从 Java 到 Groovy

我们从一段带有一个简单循环的Java代码入手。首先通过Groovy运行这段代码，然后将其从Java风格重构为Groovy风格。在代码演进过程中，每个版本做的事情相同，但是代码越来越简洁，表现力也越来越好。那感觉就像打了兴奋剂。我们开始吧。

2.1.1 Hello, Groovy

让我们从一个以Java代码编写例子开始，它同时也是Groovy代码，保存在一个名为 `Greetings.groovy` 的文件中。

```
// Java代码
public class Greetings {
    public static void main(String[] args) {
        for(int i = 0; i < 3; i++) {
            System.out.print("ho ");
        }

        System.out.println("Merry Groovy!");
    }
}
```

使用 `groovy Greetings.groovy` 命令执行这段代码，看一下输出：

```
ho ho ho Merry Groovy!
```

这么简单的任务，代码可真够多的。不过Groovy依然乖乖地接受并执行了它。

Groovy的信噪比比Java要高，故而可以用较少的代码获得更多结果。实际上，去掉上面程序中的大部分代码，仍然可以得到相同的结果。我们就从去掉表示一行结束的分号开始。去掉分号能减少噪音，代码也会更流畅。

现在去掉类和方法定义，Groovy仍是欣然接受（说不定更喜欢了）。

默认导入

在编写Groovy代码时，不必导入所有的常用类或包。例如，使用`Calendar`，就可以毫无困难地引用`java.util.Calendar`。Groovy自动导入下列包：`java.lang`、`java.util`、`java.io`和`java.net`。它也会导入`java.math.BigDecimal`和`java.math.BigInteger`两个类。此外，它还导入了`groovy.lang`和`groovy.util`这些Groovy包。

`GroovyForJavaEyes/LightGreetings.groovy`

```
for(int i = 0; i < 3; i++) {
    System.out.print("ho ")
}
System.out.println("Merry Groovy!")
```

甚至可以更进一步。Groovy能够理解`println()`，因为该方法已经被添加到`java.lang.Object`中。它还有一种使用`Range`对象的、更为轻量级的`for`循环形式，而且Groovy对括号很宽容。因此，可以把上面代码简化成下面这样：

`GroovyForJavaEyes/LighterGreetings.groovy`

```
for(i in 0..2) { print 'ho ' }
println 'Merry Groovy!'
```

这段代码的输出与本节开始所编写的Java代码相同，但是代码轻便多了。在Groovy中，简单的事情就简单做。

2.1.2 实现循环的方式

Groovy并没有限制使用传统的`for`循环。我们已经在`for`循环中用过了`range 0..2`。其实，Groovy为我们提供了很多优雅的迭代方式。

比如`upto()`，它是Groovy向`java.lang.Integer`类中添加的一个便于使用的实例方法，可用于迭代。

GroovyForJavaEyes/WaysToLoop.groovy

```
0.upto(2) { print "$it "}
```

我们在0上调用了upto()，这里的0就是Integer的一个实例。输出应该显示所选范围内的每个值。

```
0 1 2
```

那代码块中的\$it是什么呢？在这个上下文中，它代表进行循环时的索引值。upto()方法接受一个闭包作为参数。如果闭包只需要一个参数，在Groovy中则可以使用默认的名字it来表示该参数。（先记住这一点，第4章将更详细地讨论闭包。）变量it前面的\$让print()方法打印该变量的值，而非打印it这两个字符。利用该特性，我们可以在字符串中嵌入表达式，第5章有此类用法。

使用upto()方法时，可以设置范围的上限和下限。如果范围从0开始，也可以使用times()，如下面的例子所示：

GroovyForJavaEyes/WaysToLoop.groovy

```
3.times { print "$it "}
```

这个版本将与上个版本产生相同的输出：

```
0 1 2
```

要在循环时跳过一些值，可以使用step()方法。

GroovyForJavaEyes/WaysToLoop.groovy

```
0.step(10, 2) { print "$it "}
```

输出将显示该范围内选定的值：

```
0 2 4 6 8
```

还可以使用类似方法迭代或遍历对象的集合，第6章将介绍相关用法。

接下来，使用前面学到的方法重写Greetings这个例子。与我们开始所写的Java代码相比，看看Groovy代码是多么简短吧：

GroovyForJavaEyes/WaysToLoop.groovy

```
3.times { print 'ho ' }
println 'Merry Groovy!'
```

为确认这段代码的效果，运行并查看输出：

```
ho ho ho Merry Groovy!
```

2.1.3 GDK一瞥

Java平台的核心优势之一就是其Java开发包(JDK)。Groovy并没有强迫我们学习一组新的类和库。通过向JDK的各种类中添加便捷方法,Groovy扩展了强大的JDK。这些扩展可以在称作GDK(或Groovy JDK, <http://groovy.codehaus.org/groovy-jdk>)的库中获得。通过使用Groovy提供的便捷方法,我们甚至可以更深入地利用JDK。下面通过一个用于与外部进程通信的GDK便捷方法来激发我们的兴趣。

我生命中有一部分时间花在了版本控制系统的维护上。每当有文件签入时,后端的钩子会使用一些规则,执行进程,然后发出通知。简而言之,我必须创建进程,并与这些进程交互。来看一下Groovy在这里是如何帮助我们的。

Java中可以使用`java.lang.Process`与系统级进程交互。假设我们想在代码中调用Subversion的`help`,下面是实现该功能的Java代码:

```
//Java代码
import java.io.*;
public class ExecuteProcess {
    public static void main(String[] args) {
        try {
            Process proc = Runtime.getRuntime().exec("svn help");
            BufferedReader result = new BufferedReader(
                new InputStreamReader(proc.getInputStream()));
            String line;
            while((line = result.readLine()) != null) {
                System.out.println(line);
            }
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

`java.lang.Process`类非常有用,但是在前面的代码中使用它时,真是大费周章;实际上,异常处理代码以及为实现输出所做的努力,所有这些让我们头晕目眩。通过在`java.lang.String`类上添加一个`execute()`方法,GDK使这一切变得非常简单了。

GroovyForJavaEyes/Execute.groovy

```
println "svn help".execute().text
```

比较这两个代码段,让我想起了电影《夺宝奇兵》中持剑打斗的场景。Java代码就像带剑的反派,搞了一个大大的噱头。而另一方面,Groovy就像印第安纳·琼斯博士,不费吹灰之力就把任务完成了。不要误解我的意思——我当然不是说Java是反派。在Groovy代码中,我们仍然使用了`Process`和JDK。那些让利用JDK和Java平台的力量更困难且更耗时的不必要的复杂性,才是我们的敌人。

在我维护的一个Subversion钩子中，一次重构把50多行Java代码减少到了只有3行Groovy代码。这两个版本我更喜欢哪一个呢？当然是简洁明了的那个了（除非我们是按代码行数收费的咨询师……）。

当在String实例上调用execute()方法时，Groovy创建了一个扩展了java.lang.Process的类的实例，就像Java代码中Runtime类的exec()方法所做的那样。可以使用如下代码验证这一点：

GroovyForJavaEyes/Execute.groovy

```
println "svn help".execute().getClass().name
```

当在类Unix机器上运行时，输出如下：

```
java.lang.UNIXProcess
```

在Windows机器上，输出则是：

```
java.lang.ProcessImpl
```

当调用text时，我们是在调用Groovy在Process类上添加的getText()方法，其功能是将该进程的整个标准输出读到一个String对象中。如果只是想等待进程结束，waitFor()或Groovy添加的waitForOrKill()方法（该方法接受一个以毫秒表示的超时值）会有所帮助。这就来试一下上面的代码吧。

除了使用Subversion，还可以尝试其他命令：只需要将svn help替换为其他程序，比如groovy -v。

GroovyForJavaEyes/Execute.groovy

```
println "groovy -v".execute().text
```

我们在Groovy脚本内调用的独立Groovy进程将报告其版本号：

GroovyForJavaEyes/Execute.output

```
Groovy Version: 2.1.1 JVM: 1.7.0_04-ea Vendor: Oracle Corporation OS: Mac OS X
```

这个例子在类Unix系统和Windows系统上均可工作。类似地，在一个类Unix系统上，要得到当前目录下内容的列表，可以调用ls：

GroovyForJavaEyes/Execute.groovy

```
println "ls -l".execute().text
```

在Windows上，简单地把ls替换为dir是不起作用的。原因在于，尽管ls是一个可以在类Unix系统上执行的程序，但dir并不是一个程序，它只是一个shell命令。所以除了调用dir，还得干点活。明确地说，我们需要调用cmd，并让它来执行dir命令：

GroovyForJavaEyes/Windows/ExecuteDir.groovy

```
println "cmd /C dir".execute().text
```

我们已经看到GDK扩展能使我们的编码工作更为轻松，但是仅仅触及了GDK的一点皮毛。第7章将研究GDK的更多精彩内容。

2

2.1.4 安全导航操作符

Groovy有很多激动人心且能帮助简化开发工作的小特性，继续阅读本书，你会发现这些特性遍布各个章节。安全导航（safe-navigation）操作符（`?.`）就是其中之一。我们经常需要检查引用是否为空值（`null`）。这种操作单调乏味，如下面例子所示，使用该操作符，可以避免这种操作：

GroovyForJavaEyes/Ease.groovy

```
def foo(str) {
    //if (str != null) { str.reverse() }
    str?.reverse()
}

println foo('evil')
println foo(null)
```

`foo()`方法（介绍编程的书籍中，往往至少会出现一个名为`foo`的方法）中的`?.`操作符只有在引用不为`null`时才会调用指定的方法或属性。运行这段代码，看一下输出：

```
live
null
```

使用`?.`在空引用上调用`reverse()`，其结果是产生了一个`null`，而没有抛出空指针异常（`NullPointerException`），这是Groovy减少噪音、节省开发者力气的另一手段。

2.1.5 异常处理

与Java相比，Groovy少了很多繁文缛节。这一点在异常处理上极其明显。Java强制我们处理所有受检查异常（Checked Exception）。试想一个简单的例子：我们想调用`Thread`的`sleep()`方法。（Groovy提供了一个备选的`sleep()`方法，参见7.1.3节。）Java坚持让我们捕获`java.lang.InterruptedException`。当不得已之时，Java程序员会怎么做呢？想办法处理呗。结果就是大量空的`catch`块，是不是？看看这个：

GroovyForJavaEyes/Sleep.java

```
// Java代码
try {
```

```

    Thread.sleep(5000);
} catch(InterruptedException ex) {
    // 啊？这里该做什么？我都因为这个寝食难安了。
}

```

使用空的catch块比不处理异常更糟糕。如果放一个空的catch块，异常就被压制了下来。而如果我们在异常出现的第一个位置不予处理，该异常会被传递给调用者。这样调用者就可以做些处理工作，或是将其再传递给更上层的调用者。

对于那些我们不想处理，或者不适合在代码的当前层次处理的异常，Groovy并不强制我们处理。我们不处理的任何异常都会被自动传递给更高一层。下面用一个例子来说明一下，Groovy针对异常处理给出的答案：

```

GroovyForJavaEyes/ExceptionHandling.groovy
def openFile(fileName) {
    new FileInputStream(fileName)
}

```

openFile()方法没有处理声名狼藉的FileNotFoundException异常。如果产生了该异常，它并不会被压制下来。相反，它会被传递给调用代码，由调用代码来处理，如下面的例子所示：

```

GroovyForJavaEyes/ExceptionHandling.groovy
try {
    openFile("nonexistentfile")
} catch(FileNotFoundException ex) {
    // 关于该异常，在这里想做什么就做什么
    println "Oops: " + ex
}

```

如果有兴趣捕获可能抛出的所有异常（Exception），可以简单地在catch语句中省略异常的类型：

```

GroovyForJavaEyes/ExceptionHandling.groovy
try {
    openFile("nonexistentfile")
} catch(ex) {
    // 关于该异常，在这里想做什么就做什么
    println "Oops: " + ex
}

```

利用catch(ex)（变量ex前面没有任何类型）可以捕获摆在我们面前的任何异常。注意：它不能捕获Exception之外的Error或Throwable。要捕获所有这些，请使用catch(Throwable throwable)。

可见，Groovy让我们专注于完成工作，而不是将精力浪费在处理恼人的系统级细节上。

2.1.6 Groovy 是轻量级的 Java

Groovy还有其他一些使这门语言更为轻量级、更为易用的特性，试举几例如下。2

- `return`语句几乎总是可选的（参见2.11.1节）。
- 尽管可以使用分号分隔语句，但它几乎总是可选的（参见2.11.6节）。
- 方法和类默认是公开（`public`）的。
- `?.`操作符只有对象引用不为空时才会分派调用。
- 可以使用具名参数初始化JavaBean（参见2.2节）。
- Groovy不强迫我们捕获自己不关心的异常，这些异常会被传递给代码的调用者。
- 静态方法内可以使用`this`来引用`Class`对象。在下面的例子中，`learn()`方法返回的是`Class`对象，所以可以使用链式调用：

```
class Wizard {
    def static learn(trick, action) {
        //...
        this
    }
}
Wizard.learn('alohomora', {/*...*/})
    .learn('expelliarmus', {/*...*/})
    .learn('lumos', {/*...*/})
```

在领教了Groovy的表现力和简洁这两大特性之后，下一节将介绍Groovy是如何在Java的一大最基本特性中减少混乱的。

2.2 JavaBean

JavaBean概念的引入令我们异常兴奋，那些能够按照特定约定暴露出其属性的Java对象就被视作JavaBean。这一概念唤起了很多希望，但是不久我们发现，要访问这些属性，开发者还是需要调用访问器（`Getter`）和更改器（`Setter`）方法。我们的兴奋轰然倒塌，而开发者还是要继续在其应用中创建成千上万傻瓜似的方法^①。如果把JavaBean比作人，那么他们应该吃百忧解^②了。公平地说，JavaBean的意图是好的，它使基于组件的开发、应用装配和集成变得切实可行，也为优秀的集成开发环境（IDE）和插件开发铺平了道路。

在Groovy中，JavaBean获得了应有的尊重。而且Groovy中的JavaBean确实是有属性的。我们就从Java代码入手，然后将其简化成Groovy代码，这样就能看到差别了。

^① <http://www.javaworld.com/javaworld/jw-09-2003/jw-0905-toolbox.html>

^② 一种治疗精神抑郁的药物。——编者注

GroovyForJavaEyes/Car.java

```
//Java代码
public class Car {
    private int miles;
    private final int year;

    public Car(int theYear) { year = theYear; }
    public int getMiles() { return miles; }
    public void setMiles(int theMiles) { miles = theMiles; }

    public int getYear() { return year; }

    public static void main(String[] args) {
        Car car = new Car(2008);

        System.out.println("Year: " + car.getYear());
        System.out.println("Miles: " + car.getMiles());
        System.out.println("Setting miles");
        car.setMiles(25);
        System.out.println("Miles: " + car.getMiles());
    }
}
```

这是我们再熟悉不过的Java代码，不是吗？看一下Car实例属性的输出：

```
Year: 2008
Miles: 0
Setting miles
Miles: 25
```

前面的Java代码可以在Groovy中运行，但是如果用Groovy重写，则可以去掉很多乱七八糟的东西：

GroovyForJavaEyes/GroovyCar.groovy

```
class Car {
    def miles = 0
    final year

    Car(theYear) { year = theYear }
}

Car car = new Car(2008)

println "Year: $car.year"
println "Miles: $car.miles"
println 'Setting miles'
car.miles = 25
println "Miles: $car.miles"
```

这段代码和前面的Java代码做的事情是一样的（从输出可以看出），但是少了很多混乱和繁文缛节。

```
Year: 2008
Miles: 0
Setting miles
Miles: 25
```

`def`在这个上下文中声明了一个属性。我们可以像例子中这样使用`def`声明属性，还可以像`int miles`或`int miles = 0`这样给出类型（以及可选的值）。Groovy会在背后默默地为其创建一个访问器和一个更改器（就像在Java中，如果没有编写任何构造器，则Java编译器会创建一个）。当在代码中调用`miles`时，其实并非引用一个字段，而是调用该属性的访问器。要把属性设置为只读的，需要使用`final`来声明该属性，这和Java中一样。在这种情况下，Groovy会为该属性提供一个访问器，但不提供更改器。修改`final`字段的任何尝试都会导致异常。可以根据需要向声明中加入类型信息。可以把字段标记为`private`，但是Groovy并不遵守这一点^①。因此，如果想把变量设置为私有的，必须实现一个拒绝任何修改的更改器。可以通过下面的代码验证这些概念：

```
GroovyForJavaEyes/GroovyCar2.groovy
class Car {
    final year
    private miles = 0

    Car(theYear) { year = theYear }

    def getMiles() {
        println "getMiles called"
        miles
    }

    private void setMiles(miles) {
        throw new IllegalAccessException("you're not allowed to change miles")
    }

    def drive(dist) { if (dist > 0) miles += dist }
}
```

这里使用`final`声明了`year`，使用`private`声明了`miles`。在`drive()`实例方法中，无法修改`year`，但是可以修改`miles`。`miles`的更改器不允许在类的外部对该属性的值进行任何修改。下面来使用这个版本的`Car`类。

^① Groovy的实现不区分`public`、`private`和`protected`，参见<http://groovy.codehaus.org/Things+you+can+do+but+better+leave+undone>。——译者注

GroovyForJavaEyes/GroovyCar2.groovy

```

def car = new Car(2012)

println "Year: $car.year"
println "Miles: $car.miles"
println 'Driving'
car.drive(10)
println "Miles: $car.miles"

try {
    print 'Can I set the year? '
    car.year = 1900
} catch(groovy.lang.ReadOnlyPropertyException ex) {
    println ex.message
}

try {
    print 'Can I set the miles? '
    car.miles = 12
} catch(IllegalAccessException ex) {
    println ex.message
}

```

从输出可以看到，我们能够读取两个属性的值，但是不能设置其中任何一个。

```

Year: 2012
getMiles called
Miles: 0
Driving
getMiles called
Miles: 10
Can I set the year? Cannot set readonly property: year for class: Car
Can I set the miles? you're not allowed to change miles

```

要想存取属性，再也不需要在调用中使用访问器和更改器了。下面代码说明了其优雅：

GroovyForJavaEyes/UsingProperties.groovy

```

Calendar.instance
// 代替Calendar.getInstance()
str = 'hello'

str.class.name
// 代替str.getClass().getName()
// 注意：不能用于Map、Builder等类型
// 为保险起见，请使用str.getClass().name

```

然而请谨慎使用class属性，像Map、生成器等一些类对该属性有特殊处理（参见6.5节）。因此，为避免任何意外，一般使用getClass()，而不是class。

2.3 灵活初始化与具名参数

Groovy中可以灵活地初始化一个JavaBean类。在构造对象时，可以简单地以逗号分隔的名值对来给出属性值。如果类有一个无参构造器，该操作会在构造器之后执行。^①也可以设计自己的方法，使其接受具名参数。要利用这一特性，需要把第一个形参定义为Map。下面通过代码来实际地看一下。

GroovyForJavaEyes/NamedParameters.groovy

```
class Robot {
    def type, height, width
    def access(location, weight, fragile) {
        println "Received fragile? $fragile, weight: $weight, loc: $location"
    }
}
robot = new Robot(type: 'arm', width: 10, height: 40)
println "$robot.type, $robot.height, $robot.width"

robot.access(x: 30, y: 20, z: 10, 50, true)
//可以修改参数顺序
robot.access(50, true, x: 30, y: 20, z: 10)
```

运行上面代码，看一下输出：

```
arm, 40, 10
Received fragile? true, weight: 50, loc: [x:30, y:20, z:10]
Received fragile? true, weight: 50, loc: [x:30, y:20, z:10]
```

Robot实例把type、height和width等实参当作了名值对。这里使用了Groovy编译器为我们创建的灵活的构造器。

access()方法有3个形参，但如果第一个是Map，则可以将这个映射中的键值对展开放在放在实参列表中。在第一次调用access()方法时，我们依次放了这个映射、weight以及fragile的值。不过如果我们愿意，用于这个映射的实参可以继续往后移，就像第2次调用access()方法那样。

如果发送的实参的个数多于方法的形参的个数，而且多出的实参是名值对，那么Groovy会假设方法的第一个形参是一个Map，然后将实参列表中的所有名值对组织到一起，作为第一个形参的值。之后，再将剩下的实参按照给出的顺序赋给其余形参，正如我们在输出中看到的那样。

尽管在Robot的例子中，这种灵活性非常强大，但是可能会给人带来困惑，所以请谨慎使用。如果想使用具名参数，那最好只接受一个Map形参，而不要混用不同的形参。在这个例子中，如

^① 要使此类操作正确执行，类中必须有一个无参构造器。在示例代码中，因为没有定义构造器，编译器会提供一个无参的构造器。如果定义了带参数的构造器，则编译器不会再为我们创建无参构造器，所以一定要记得自己提供。读者可以自行修改代码测试。——译者注

果传递的是3个整型实参，该特性会导致一个问题。这种情况下，编译器将按顺序传递实参，而不会从这些实参创建一个映射，因而结果也就不是我们想要的了。

通过显式地将第一个形参指定为Map，可以避免这种混乱：

GroovyForJavaEyes/NamedParameters.groovy

```
def access(Map location, weight, fragile) {
    print "Received fragile? $fragile, weight: $weight, loc: $location"
}
```

现在，如果实参包含的不是两个对象外加一个任意的名值对，代码就会报错。

正如我们所见，由于Groovy给JavaBean换上了新装，JavaBean在Groovy中又生机勃发了。

2.4 可选形参

Groovy中可以把方法和构造器的形参设为可选的。实际上，我们想设置多少就可以设置多少，但这些形参必须位于形参列表的末尾。利用这一特性，可以在演进式设计中向已有方法添加新的形参。

要定义可选形参，只需要在形参列表中给它赋上一个值。下面是一个例子，`log()`方法带有一个可选的`base`形参。如果调用时不提供相应实参，则Groovy会假定其值为10：

GroovyForJavaEyes/OptionalParameters.groovy

```
def log(x, base=10) {
    Math.log(x) / Math.log(base)
}

println log(1024)
println log(1024, 10)
println log(1024, 2)
```

如输出所示，Groovy使用这个可选的值填充了缺失的实参：

```
3.0102999566398116
3.0102999566398116
10.0
```

Groovy还会把末尾的数组形参视作可选的。所以在下面的例子中，可以为最后一个形参提供零个或多个值：

GroovyForJavaEyes/OptionalParameters.groovy

```
def task(name, String[] details) {
    println "$name - $details"
```

```

}

task 'Call', '123-456-7890'
task 'Call', '123-456-7890', '231-546-0987'
task 'Check Mail'

```

从输出可以看出，Groovy会把末尾的实参收集起来，赋给数组形参：

2

```

Call - [123-456-7890]
Call - [123-456-7890, 231-546-0987]
Check Mail - []

```

多次调用某个方法时，如果需要提供相同的实参，这会让人厌烦。可选的形参减少了噪音，而且允许提供合理的默认值。

2.5 使用多赋值

向方法传递多个参数，这在很多编程语言中都司空见惯。但是从方法返回多个结果，尽管可能非常实用，却不那么常见。要想从方法返回多个结果，并将它们一次性赋给多个变量，我们可以返回一个数组，然后将多个变量以逗号分隔，放在圆括号中，置于赋值表达式左侧即可。

后面的例子中有一个负责将全名分割为名字（First Name）和姓氏（Last Name）的方法。不出所料，`split()`方法就返回一个数组。可以把`splitName()`的结果赋给一对变量：`firstName`和`lastName`。Groovy会把结果中的两个值分别赋给这两个变量。

```

GroovyForJavaEyes/MultipleAssignments.groovy
def splitName(fullName) { fullName.split(' ') }

def (firstName, lastName) = splitName('James Bond')
println "$lastName, $firstName $lastName"

```

要将结果设置到两个变量中，不必创建临时变量并编写多条赋值语句，如输出所示：

```
Bond, James Bond
```

还可以使用该特性来交换变量，无需创建中间变量来保存被交换的值，只需将欲交换的变量放在圆括号内，置于赋值表达式左侧，同时将它们以相反顺序放于方括号内，置于右侧即可。

```

GroovyForJavaEyes/MultipleAssignments.groovy
def name1 = "Thomson"
def name2 = "Thompson"

println "$name1 and $name2"
(name1, name2) = [name2, name1]
println "$name1 and $name2"

```

从输出中可以看到，name1和name2的值被交换了。

```
Thomson and Thompson
Thompson and Thomson
```

我们已经看到，当赋值表达式左侧的变量与右侧的值数目相同时，Groovy是如何处理多赋值的。而当变量与值的数目不匹配时，Groovy也可以优雅地处理。如果有多余的变量，Groovy会将它们设置为null，多余的值则会被丢弃。

如下面例子所示，还可以指定多赋值中定义的单个变量的类型。下面使用来自著名卡通系列《猫和老鼠》中的动物来说明这一点。

GroovyForJavaEyes/MultipleAssignments.groovy

```
def (String cat, String mouse) = ['Tom', 'Jerry', 'Spike', 'Tyke']
println "$cat and $mouse"
```

左侧只有两个变量，所以狗狗Spike和Tyke将被丢弃。

```
Tom and Jerry
```

GroovyForJavaEyes/MultipleAssignments.groovy

```
def (first, second, third) = ['Tom', 'Jerry']
println "$first, $second, and $third"
```

第三个变量的值将被设置为null。

```
Tom, Jerry, and null
```

如果多余的变量是不能设置为null的基本类型，Groovy将抛出一个异常。这是一种新行为，在Groovy 2.x中，只要可能，int会被看作基本类型，而非Integer。

可见，Groovy使发送和接收多个参数变得非常容易了。

2.6 实现接口

在Groovy中，可以把一个映射或一个代码块转化为接口，因此可以快速实现带有多个方法的接口。本节，你将看到Java实现接口的方式，然后学习如何利用Groovy实现接口。

下面是用于向Swing的 JButton注册事件处理器的Java代码，我们再熟悉不过了。调用addActionListener()时，需要一个实现了ActionListener接口的实例。其中创建了一个实现了该接口的匿名内部类，同时提供了所需的actionPerformed()方法。该方法要求提供一个ActionEvent实例作为参数，即便我们在这个例子中并未用到。

```
// Java代码
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
```

```
JOptionPane.showMessageDialog(frame, "You clicked!");
}
});
```

Groovy提供了一个与之不同的惯用法，非常迷人，不需要`actionPerformed()`方法声明，也不需要显式地用`new`来创建匿名内部类的实例。

```
button.addActionListener(
    { JOptionPane.showMessageDialog(frame, "You clicked!") } as ActionListener
)
```

调用了`addActionListener`方法，同时为该方法提供了一个代码块，借助`as`操作符，相当于实现了`ActionListener`接口。

就是它了！Groovy自会处理剩下的工作。它会拦截对接口中任何方法的调用（这个例子中就是`actionPerformed()`），然后将调用路由到我们提供的代码块。要运行这段代码，还需要创建窗体（Frame）及其组件。完整的代码清单参见本节末尾。

对于有多个方法的接口，如果打算为其所有方法提供一个相同的实现，和上面一样，不需要特殊的操作。

假设想实现一个功能，随着鼠标被点击或者在应用中移动而显示鼠标指针的位置。在Java中，我们必须实现`MouseListener`和`MouseMotionListener`接口中的总共7个方法。因为Groovy对所有这些方法的实现都是相同的，所以它给我们带来了方便。

```
displayMouseLocation = { positionLabel.setText("$it.x, $it.y") }
frame.addMouseListener(displayMouseLocation as MouseListener)
frame.addMouseMotionListener(displayMouseLocation as MouseMotionListener)
```

这段代码创建了变量`displayMouseLocation`，它指向的是一个代码块。使用`as`操作符将其转化了2次，分别转化为`MouseListener`和`MouseMotionListener`。Groovy又一次处理了剩下的事情，我们从而可以把更多精力转向其他事情了。这里用了3行代码，而不会像在Java中那样用掉……不好意思，我又数行数了。

前面的例子中又出现了`it`变量。`it`表示方法的参数。如果正在实现的一个接口中的方法需要多个参数，那么可以将其分别定义为独立的参数，也可以定义为一个数组类型的参数，具体情况将在第4章讨论。

Groovy没有强制实现接口中的所有方法：可以只定义自己关心的，而不考虑其他方法。如果剩下的方法从来不会被调用，那也就没必要去实现这些方法了。当在单元测试中通过实现接口来模拟某些行为时，这项技术非常有用。

好了，这挺不错的，但是在大多数实际情况下，接口中的每个方法需要不同的实现。不用担心，Groovy可以摆平。只需要创建一个映射，以每个方法的名字作为键，以方法对应的代码体作为键值，同时使用简单的Groovy风格，用冒号（`:`）分隔方法名和代码块即可。此外，不必实现

所有方法，只需实现真正关心的那些即可。如果未予实现的方法从未被调用过，那么也就没有必要浪费精力去实现这些伪存根。当然，如果没提供的方法被调用了，则会出现`NullPointerException`。下面把这些内容放到一个例子里看看：

```
handleFocus = [
    focusGained : { msgLabel.setText("Good to see you!") },
    focusLost : { msgLabel.setText("Come back soon!") }
]
button.addFocusListener(handleFocus as FocusListener)
```

每当例子中的按钮获得焦点时，与`focusGained`键关联的第一个代码块就会被调用。当按钮失去焦点时，与`focusLost`键关联的代码块则会被调用。在这种情况下，这里的键相当于`FocusListener`接口中的方法。

如果知道所实现接口的名字，使用`as`操作符即可，但如果应用要求的行为是动态的，而且只有在运行时才能知道接口的名字，又该如何呢？`asType()`方法可以帮忙。通过将欲实现接口的`Class`元对象作为一个参数发送给`asType()`，可以把代码块或映射转化为接口。我们来看一个例子。

```
events = ['WindowListener', 'ComponentListener']
//上面的列表可能是动态的，而且可能来自某些输入
handler = { msgLabel.setText("$it") }
for (event in events) {
    handlerImpl = handler.asType(Class.forName("java.awt.event.${event}"))
    frame."add${event}"(handlerImpl)
}
```

想实现的接口（也就是想处理的事件）在列表`events`中。该列表是动态的，假设它会在代码执行期间通过输入来填充。事件公共的处理器位于变量`handler`指向的代码块中。我们对事件进行循环，对于每个事件，都使用了`asType()`方法为该接口创建了一个实现。在代码块上调用`asType()`方法，同时把使用`forName()`方法获得的、该接口的`Class`元对象传给它。一旦手头有了监听器接口的实现，就可以通过调用相应的`add`方法（如`addWindowListener()`）来注册该实现。调用`add`方法本身就是动态的。11.2节将介绍这些方法的更多细节。

上面的代码使用了`asType()`方法。如果不同方法有不同实现，就会使用一个映射代替单个代码块。在那种情况下，可以用类似的方式在映射上调用`asType()`方法。最后，如约分享本节开发的Groovy Swing代码的完整清单。

GroovyForJavaEyes/Swing.groovy

```
import javax.swing.*
import java.awt.*
import java.awt.event.*

frame = new JFrame(size: [300, 300],
    layout: new FlowLayout(),
    title: "Groovy Swing Example")
```

```

defaultCloseOperation: javax.swing.WindowConstants.EXIT_ON_CLOSE)
button = new JButton("click")
positionLabel = new JLabel("")
msgLabel = new JLabel("")
frame.contentPane.add button
frame.contentPane.add positionLabel
frame.contentPane.add msgLabel

button.addActionListener(
    { JOptionPane.showMessageDialog(frame, "You clicked!") } as ActionListener
)

displayMouseLocation = { positionLabel.setText("$it.x, $it.y") }
frame.addMouseListener(displayMouseLocation as MouseListener)
frame.addMouseMotionListener(displayMouseLocation as MouseMotionListener)

handleFocus = [
    focusGained : { msgLabel.setText("Good to see you!") },
    focusLost : { msgLabel.setText("Come back soon!") }
]
button.addFocusListener(handleFocus as FocusListener)
events = ['WindowListener', 'ComponentListener']
// 上面的列表可能是动态的，而且可能来自某些输入
handler = { msgLabel.setText("$it") }
for (event in events) {
    handlerImpl = handler.asType(Class.forName("java.awt.event.${event}"))
    frame."add${event}"(handlerImpl)
}

frame.show()

```

现在已经介绍完了Groovy实现接口的方式。它使注册事件和传递接口的匿名实现变得非常简单。将代码块和映射转化为接口实现也相当省时。

2.7 布尔求值

Groovy中的布尔求值与Java不同。根据上下文，Groovy会自动把表达式计算为布尔值。

来看一个具体的例子，下面的Java代码是不工作的：

```

//Java代码
String obj = "hello";
int val = 4;
if (obj) {} // 错误
if(val) {} // 错误

```

Java要求if语句的条件部分必须是一个布尔表达式，比如前面例子中的`if(obj != null)`和`if(val > 0)`。

Groovy可没那么挑剔。它会尝试推断，所以我们需要知道它是怎么思考问题的。

如果在需要布尔值的地方放了一个对象引用，Groovy会检查该引用是否为null。它将null视作false，将非null的值视作true，如以下代码所示：

```
str = 'hello'
if (str) { println 'hello' }
```

如输出所示，Groovy会将该表达式计算作布尔值：

```
hello
```

必须承认，前面关于true的说法并不完全正确。如果对象引用不为null，表达式的结果还与对象的类型有关。例如，如果对象是一个集合（如java.util.ArrayList），那么Groovy会检查该集合是否为空。因此，在这种情况下，只有当obj不为null，而且该集合至少包含一个元素时，表达式if(obj)才会被计算为true；请看下面代码示例的输出：

```
lst0 = null
println lst0 ? 'lst0 true' : 'lst0 false'
lst1 = [1, 2, 3]
println lst1 ? 'lst1 true' : 'lst1 false'
lst2 = []
println lst2 ? 'lst2 true' : 'lst2 false'
```

对于集合类(Collection)，Groovy是如何将其处理为布尔值的呢？可以通过这段代码的输出来看看我们的理解是否正确：

```
lst0 false
lst1 true
lst2 false
```

集合类不是唯一受到特殊对待的。那么有哪些类型将被特殊对待，Groovy又是如何计算它们的呢？请参考表2-1。

表2-1 类型与布尔求值对它们的特殊处理

类 型	为真的条件
Boolean	值为true
Collection	集合不为空
Character	值不为0
CharSequence	长度大于0
Enumeration	Has More Elements()为true
Iterator	hasNext()为true
Number	Double值不为0
Map	该映射不为空
Matcher	至少有一个匹配
Object[]	长度大于0
其他任何类型	引用不为null

除了使用Groovy内建的布尔求值约定，在自己的类中，还可以通过实现`asBoolean()`方法来编写自己的布尔转换。

2.8 操作符重载

Groovy支持操作符重载，可以巧妙地应用这一点来创建DSL（领域特定语言，参见第19章）。Java是不支持操作符重载的，那Groovy又是如何做到的呢？其实很简单：每个操作符都会映射到一个标准的方法^①。在Java中，可以使用那些方法；而在Groovy中，既可以使用操作符，也可以使用与之对应的方法。

下面是一个演示操作符重载的例子：

GroovyForJavaEyes/OperatorOverloading.groovy

```
for(ch = 'a'; ch < 'd'; ch++) {
    println ch
}
```

我们通过`++`操作符实现了从字符a到c的循环。该操作符映射的是`String`类的`next()`方法，输出如下：

```
a  
b  
c
```

Groovy中还可以使用简洁的`for-each`语法，不过两种实现都用到了`String`类的`next()`方法：

GroovyForJavaEyes/OperatorOverloading.groovy

```
for (ch in 'a'..'c') {
    println ch
}
```

`String`类重载了很多操作符，5.4节将予以介绍。类似地，为方便使用，集合类（如`ArrayList`和`Map`）也重载了一些操作符。

要向集合中添加元素，可以使用`<<`操作符，该操作符会被转换为Groovy在`Collection`上添加的`leftShift()`方法，如下所示：

GroovyForJavaEyes/OperatorOverloading.groovy

```
lst = ['hello']
lst << 'there'
println lst
```

① <http://groovy.codehaus.org/Operator+Overloading>

在完成追加元素后，可以在输出中看到完整的集合：

```
[hello, there]
```

通过添加映射方法，我们可以为自己的类提供操作符，比如为+操作符添加plus()方法。

下面来为一个类添加一个重载的操作符：

```
GroovyForJavaEyes/OperatorOverloading.groovy
class ComplexNumber {
    def real, imaginary
    def plus(other) {
        new ComplexNumber(real: real + other.real,
                           imaginary: imaginary + other.imaginary)
    }
    String toString() { "$real ${imaginary > 0 ? '+' : ''} ${imaginary}i" }
}
c1 = new ComplexNumber(real: 1, imaginary: 2)
c2 = new ComplexNumber(real: 4, imaginary: 1)
println c1 + c2
```

ComplexNumber类重载了+操作符。对于计算涉及负数平方根的复杂方程式，复数非常有用。复数有实部和虚部，就像人们的收入有实际收入和所得税申报单上的收入之分一样。因为在ComplexNumber类上添加了plus()方法，所以可以使用+操作符把两个复数加到一起，得到又一个作为结果的复数：

```
5 + 3i
```

当应用于某个上下文时，操作符重载可以使代码更富于表现力。应该只重载那些能使事物变得显而易见的操作符。例如，如果对于有上下文背景或领域知识的某些人而言，有些操作符反而不是那么直观，那重载可能就不是很好的选择。

在重载时，必须保留预期的语义。例如，+操作符不可以修改操作中的任何一个操作数。如果操作符必须是可交换的、对称的或传递的，则必须确保重载的方法遵循这些特性。

2.9 对Java 5特性的支持

像枚举和注解等Java 5的语言特性在Groovy中也可以工作。这意味着我们可以非常自然地混用Java和Groovy。回忆一下，Java 5引入了如下语言特性：

- 自动装箱
- for-each循环
- enum
- 变长参数 (varargs)

- 注解
- 静态导入
- 泛型

下面来讨论一下Groovy对这些特性的支持程度。

2

2.9.1 自动装箱

因为Groovy具有动态类型特性，所以它从一开始就支持自动装箱。实际上，必要时Groovy会自动将基本类型视作对象。例如，执行下面代码：

GroovyForJavaEyes/NotInt.groovy

```
int val = 5
println val.getClass().name
```

报告的类型如下：

```
java.lang.Integer
```

在这段代码中，尽管指定的是int，但创建的是java.lang.Integer类的实例，而不是基本类型int的变量。Groovy会根据该实例的使用方式来决定将其存储为int类型还是Integer类型。Groovy在对自动装箱的处理上要比Java略胜一筹。在Java中，自动装箱和自动拆箱会涉及类型之间的转换。而另一方面，Groovy就简单地将其当作对象，所以不需要反复地转换类型。

在2.0版本之前，Groovy中所有基本类型都被看作对象。为了改进性能，也为了能在基本类型的操作上使用更为直接的字节码，从2.0版本起，Groovy做了一些优化。基本类型只在必要时才会被看作对象，比如，在其上调用了方法，或者将其传给了对象引用。否则，Groovy会在字节码级别将其保留为基本类型。

2.9.2 for-each

Groovy对循环的支持优于Java（参见2.1.2节）。在Groovy中仍然可以使用传统的for循环（也就是for(int i = 0; i < 10; i++) {...}）。或者，如果喜欢Java 5支持的更简单的循环形式，也可以使用。在Java 5中，实现了Iterable接口的对象可以用于for-each循环中，如下面的例子所示：

GroovyForJavaEyes/ForEach.java

```
// Java代码
String[] greetings = {"Hello", "Hi", "Howdy"};
for(String greet : greetings) {
    System.out.println(greet);
}
```

Groovy中可以像下面这样重写该例子：

```
GroovyForJavaEyes/ForEach.groovy
String[] greetings = ["Hello", "Hi", "Howdy"]
for(String greet : greetings) {
    println greet
}
```

在Java风格的**for-each**循环中，Groovy要求指明类型（即前面例子中的**String**），或者使用**def**。如果不想指定类型，则要使用**in**关键字代替冒号（**:**），如下面例子所示：

```
GroovyForJavaEyes/ForEach.groovy
for(greet in greetings) {
    println greet
}
```

相对于Java风格的**for-each**语法，在Groovy中我们更喜欢使用带有**in**的**for**语句。作为一种选择，还可以使用**each()**这一内部迭代器（参见第6章）。

2.9.3 enum

Groovy提供了对**enum**的支持，这是Java 5为解决枚举问题而引入的特性。它是类型安全的（比如，我们可以区分得出用**enum**表示的衬衫尺寸和一周中的每一天），还具有可打印、可序列化等特点。

下面例子定义了我们可以购买的咖啡饮品的容量规格：

```
GroovyForJavaEyes/UsingCoffeeSize.groovy
enum CoffeeSize { SHORT, SMALL, MEDIUM, LARGE, MUG }
def orderCoffee(size) {
    print "Coffee order received for size $size: "
    switch(size) {
        case [CoffeeSize.SHORT, CoffeeSize.SMALL]:
            println "you're health conscious"
            break
        case CoffeeSize.MEDIUM..CoffeeSize.LARGE:
            println "you gotta be a programmer"
            break
        case CoffeeSize.MUG:
            println "you should try Caffeine IV"
            break
    }
}
orderCoffee(CoffeeSize.SMALL)
orderCoffee(CoffeeSize.LARGE)
```

```

orderCoffee(CoffeeSize.MUG)
print 'Available sizes are: '
for(size in CoffeeSize.values()) {
    print "$size"
}

```

在前面的代码中，利用switch语句和enum上的迭代，很方便地产生了如下输出：

```

Coffee order received for size SMALL: you're health conscious
Coffee order received for size LARGE: you gotta be a programmer
Coffee order received for size MUG: you should try Caffeine IV
Available sizes are: SHORT SMALL MEDIUM LARGE MUG

```

可以在case语句中使用枚举值。特别地，我们可以使用一个值、一组值的列表或者值的一个区间。前面的代码囊括了上述这些用法。

Groovy也支持为Java 5的enum定义构造器和方法。请看下面例子：

GroovyForJavaEyes/AgileMethodologies.groovy

```

enum Methodologies {
    Evo(5),
    XP(21),
    Scrum(30);

    final int daysInIteration
    Methodologies(days) { daysInIteration = days }

    def iterationDetails() {
        println "${this} recommends $daysInIteration days for iteration"
    }
}

for(methodology in Methodologies.values()) {
    methodology.iterationDetails()
}

```

看一下在这个enum上迭代产生的输出：

```

Evo recommends 5 days for iteration
XP recommends 21 days for iteration
Scrum recommends 30 days for iteration

```

2.9.4 变长参数

利用Java 5的变长参数特性，可以向方法（比如printf()）传递数目不等的参数。要在Java中使用这一特性，我们使用一个省略符号（...）标记方法末尾的形参，比如public static Object max(Object... args)。这是语法糖，Java在调用时会把所有实参放入一个数组中。

Groovy以两种方式支持Java 5的变长参数特性，除了支持使用省略符号标记形参，对于以数

组作为末尾形参的方法，也可以向其传递数目不等的参数。

下面例子演示了Groovy支持的这两种方式：

GroovyForJavaEyes/VarArgs.groovy

```
def receiveVarArgs(int a, int... b) {
    println "You passed $a and $b"
}

def receiveArray(int a, int[] b) {
    println "You passed $a and $b"
}

receiveVarArgs(1, 2, 3, 4, 5)
receiveArray(1, 2, 3, 4, 5)
```

从输出可以看到，这两个版本都接受数目可变的实参：

```
You passed 1 and [2, 3, 4, 5]
You passed 1 and [2, 3, 4, 5]
```

对于接受变长参数或者以数组作为末尾形参的方法，可以向其发送数组或离散的值，Groovy知道该做什么。

在发送数组而非离散值时，请务必谨慎。Groovy会将包围在方括号中的值看作ArrayList的一个实例，而不是纯数组。所以如果简单地发送如[2, 3, 4, 5]这样的值，将出现MethodMissingException。要发送数组，可以定义一个指向该数组的引用，或使用as操作符。

GroovyForJavaEyes/VarArgs.groovy

```
int[] values = [2, 3, 4, 5]
receiveVarArgs(1, values)
receiveVarArgs(1, [2, 3, 4, 5] as int[])
```

大多数情况下，Groovy把类型看作可选的，但是这里我们看到，指定类型可以改变语义。

2.9.5 注解

Java中可以使用注解来表示元数据，而且Java 5带来了一些预定义的注解，比如@Override、@Deprecated和@SuppressWarnings。

Groovy中也可以定义和使用注解，而且定义注解的语法与Java相同。

在使用框架时经常会用到注解，比如，JUnit 4.0使用了@Test注解。如果正在使用像Hibernate、JPA、Seam和Spring这样的框架，我们就会发现Groovy对注解的支持非常有用。

对于Java中与编译相关的注解，Groovy的处理方式有所不同。例如，groovyc会忽略@Override。

2.9.6 静态导入

在Java中，静态导入可以帮助我们把一个类的静态方法导入到我们的命名空间中，所以无需指定类名即可引用它们。例如，如果将下面的语句放到Java代码中：

```
import static Math.random;
```

2

那么就可以像下面这样调用，而非使用`Math.random()`：

```
double val = random();
```

Java中的静态导入改进了工作的安全性。如果我们定义了几个静态导入，或者使用*导入了一个类的所有静态方法，无疑可以让那些想知道这些方法从何而来的程序员感觉莫名其妙。Groovy以两种形式扩展了这一优势。首先，它实现了静态导入。我们可以像在Java中那样使用。当然可以随意丢掉分号，它们在Groovy中是可选的。其次，在Groovy中可以为静态方法和类名定义别名。要定义别名，需要在`import`语句中使用`as`操作符：

```
import static Math.random as rand
import groovy.lang.ExpandoMetaClass as EMC

double value = rand()
def metaClass = new EMC(Integer)
assert metaClass.getClass().name == 'groovy.lang.ExpandoMetaClass'
```

这里为`Math.random()`方法创建了别名`rand()`，也为`ExpandoMetaClass`创建了别名`EMC`。现在可以分别使用`rand()`和`EMC`来代替`Math.random()`和`ExpandoMetaClass`了。

2.9.7 泛型

Groovy是支持可选类型的动态类型语言，作为Java的超集，它也支持泛型。然而，Groovy编译器不会像Java编译器那样执行类型检查（参见2.11.2节），不要期望Groovy编译器会像Java编译器那样一开始就拒绝违规的代码。如有可能，这里Groovy的动态类型特性将与泛型类型相互作用，使我们的代码运行起来。为了解这两种编译器的明显差别，在下面的例子中，我们将向一个保存`Integer`的`ArrayList`中添加一些类型不同的值。

从Java代码开始：

GroovyForJavaEyes/Generics.java

```
// Java代码
import java.util.ArrayList;

public class Generics {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(1);
```

```

list.add(2.0);
list.add("hello");

System.out.println("List populated");
for(int element : list) { System.out.println(element); }
}
}

```

当使用Java编译器编译前面的Java代码时，遇到了编译错误：

```

Generics.java:8: error: no suitable method found for add(double)
    list.add(2.0);
               ^
method ArrayList.add(int,Integer) is not applicable
(actual and formal argument lists differ in length)
method ArrayList.add(Integer) is not applicable
(actual argument double cannot be converted to Integer
by method invocation conversion)
Generics.java:9: error: no suitable method found for add(String)
    list.add("hello");
               ^
method ArrayList.add(int,Integer) is not applicable
(actual and formal argument lists differ in length)
method ArrayList.add(Integer) is not applicable
(actual argument String cannot be converted to Integer
by method invocation conversion)
2 errors

```

因为我们指定了ArrayList只保存Integer，对add()方法而言，除了Integer和int（int会被自动装箱为Integer），Java编译器不接受其他任何类型的数据。

来看一下Groovy是如何处理的。将前面的代码复制到一个命名为Generics.groovy的文件中，然后运行groovy Generics。Groovy不会阻止我们执行这段代码：

```

List populated
1
2
Caught: org.codehaus.groovy.runtime.typehandling.GroovyCastException:
Cannot cast object 'hello' with class 'java.lang.String' to class 'int'
org.codehaus.groovy.runtime.typehandling.GroovyCastException:
Cannot cast object 'hello' with class 'java.lang.String' to class 'int'
    at Generics.main(Generics.java:12)
    at Generics.invokeMethod(Generics.java)
    at RunGenerics.run(RunGenerics.groovy:1)

```

在调用add()方法的过程中，Groovy更大程度上是将类型信息看作一个建议。当对集合进行循环时，Groovy会尝试将其中的元素强制转换为int。如果无法转换，则会导致运行时错误。

Groovy在支持动态行为的同时支持泛型。前面的代码示例也说明了这两种概念有趣的相互作用。对于Groovy的这种双重性，我们一开始可能会感到惊讶，但是当学到Groovy元编程（参见第三部分）的好处时，你会看到其意义。

泛型的用处在Groovy中并没有完全丧失。如果我们愿意对元编程功能做一些折中，Groovy 2.x也在部分代码上提供了严格的类型检查，具体参见3.8节。

2.10 使用 Groovy 代码生成变换

语言设计者往往要面对这样的问题，一方面想让语言演进，而另一方面又不愿意修改语法，因为这会影响性能、复杂性和语义正确性。Groovy巧妙地缓解了这种紧张。Groovy并没有修改语言的核心语法，其编译器会识别选定的注解并生成相应代码。本节将介绍一些这样的注解。第16章将介绍如何为定制的变换创建自己的注解。

Groovy在groovy.transform包和其他一些包中提供了很多代码生成注解。本节将探讨其中的一部分。

2.10.1 使用@Canonical

如果要编写的`toString()`方法只是简单地显示以逗号分隔的字段值，则可以使用`@Canonical`变换让Groovy编译器帮来干这个活。默认情况下，它生成的代码会包含所有字段。不过可以让它仅包含特定字段，而去掉其他字段，比如下面这个例子：

GroovyForJavaEyes/Annotations.groovy

```
import groovy.transform.*

@Canonical(excludes="lastName, age")
class Person {
    String firstName
    String lastName
    int age
}

def sara = new Person(firstName: "Sara", lastName: "Walker", age: 49)
println sara
```

Groovy排除了我们提到的字段，打印了类名，类名后面是剩余字段的值，在输出中会看到：

```
Person(Sara)
```

2.10.2 使用@Delegate

只有当派生类是真正可替换的，而且可代替基类使用时，继承才显示出其优势。从纯粹的代码复用角度看，对于其他大部分用途，委托要优于继承。然而在Java中我们不太愿意使用委托，因为会导致代码冗余，而且需要更多工作。Groovy使委托变得非常容易，所以我们可以做出正确的设计选择。

要更好地理解委托，我们从一个Worker类入手，该类中有几个方法。Expert类有一个与Worker类的名字和签名都相同的方法。不出所料，Manager类什么都不干。但是它擅长把工作委托给别人，所以其中的两个字段使用@Delegate注解标记了。

GroovyForJavaEyes/Annotations.groovy

```
class Worker {
    def work() { println 'get work done' }
    def analyze() { println 'analyze...' }
    def writeReport() { println 'get report written' }
}
class Expert {
    def analyze() { println "expert analysis..." }
}
class Manager {
    @Delegate Expert expert = new Expert()
    @Delegate Worker worker = new Worker()
}
def bernie = new Manager()
bernie.analyze()
bernie.work()
bernie.writeReport()
```

在编译时，Groovy会检查Manager类，如果该类中没有被委托类中的方法，就把这些方法从被委托类中引入进来。因此，首先它会引入Expert类中的analyze()方法。而从Worker类中，只会把work()和writeReport()方法引入进来。这时候，因为从Expert类带来的analyze()方法已经出现在Manager类中，所以Worker类中的analyze()方法会被忽略。

对于引入的每个方法，Groovy会简单地把对该方法的调用路由给实例上的相应方法，就像这样：public Object analyze() { expert.analyze() }。委托类会对新获得的方法做出响应，在下面的输出中可以看到：

```
expert analysis...
get work done
get report written
```

因为有了@Delegate注解，Manager类是可扩展的。如果在Worker或Expert类上添加或去掉了方法，不必对Manager类做任何修改，相应的变化就会生效。只需要重新编译代码，剩下的事Groovy会处理。

2.10.3 使用@Immutable

不可变对象天生是线程安全的，将其字段标记为final是很好的实践选择。如果用@Immutable注解标记一个类，Groovy会将其字段标记为final的，并且额外为我们创建一些便捷方法，从而使得“做正确的事情”变得更容易了。

下面在CreditCard类中使用一下这个注解。

```
GroovyForJavaEyes/Annotations.groovy
@Immutable
class CreditCard {
    String cardNumber
    int creditLimit
}

println new CreditCard("4000-1111-2222-3333", 1000)
```

作为反馈，Groovy给我们提供了一个构造器，其参数以类中字段定义的顺序依次列出。在构造时间过后，字段就无法修改了。此外，Groovy还添加了hashCode()、equals()和toString()方法。运行所提供的构造器和toString()方法，看一下输出：

```
CreditCard(4000-1111-2222-3333, 1000)
```

可以使用@Immutable注解轻松地创建轻量级的不可变值对象。在基于Actor模型的并发应用中，线程安全是个大问题，而这些不可变值对象是作为消息传递的理想实例。

2.10.4 使用@Lazy

我们想把耗时对象的构建推迟到真正需要时。完全可以懒惰与高效并得，编写更少的代码，同时又能获得惰性初始化的所有好处。

下面的例子将推迟创建Heavy实例，直到真正需要它时。既可以在声明的地方直接初始化实例，也可以将创建逻辑包在一个闭包中。

```
GroovyForJavaEyes/Annotations.groovy
class Heavy {
    def size = 10
    Heavy() { println "Creating Heavy with $size" }
}

class AsNeeded {
    def value

    @Lazy Heavy heavy1 = new Heavy()
    @Lazy Heavy heavy2 = { new Heavy(size: value) }()

    AsNeeded() { println "Created AsNeeded" }
}

def asNeeded = new AsNeeded(value: 1000)
println asNeeded.heavy1.size
println asNeeded.heavy1.size
println asNeeded.heavy2.size
```

Groovy不仅推迟了创建，还将字段标记为`volatile`，并确保创建期间是线程安全的。实例会在第一次访问这些字段的时候被创建，在输出中可以看到：

```
Created AsNeeded
Creating Heavy with 10
10
10
Creating Heavy with 10
1000
```

另一个好处是，`@Lazy`注解提供了一种轻松实现线程安全的虚拟代理模式（virtual proxy pattern）的方式。

2.10.5 使用@Newify

在Groovy中，经常会按照传统的Java语法，使用`new`来创建实例。然而，在创建DSL时，去掉这个关键字，表达会更流畅。`@Newify`注解可以帮助创建类似Ruby的构造器，在这里，`new`是该类的一个方法。该注解还可以用来创建类似Python的构造器（也类似Scala的applicator），这里可以完全去掉`new`。要创建类似Python的构造器，必须向`@Newify`注解指明类型列表。只有将`auto=false`这个值作为一个参数设置给`@Newify`，Groovy才会创建Ruby风格的构造器。

可以在不同的作用域中使用`@Newify`注解，比如类或方法，如下面例子所示：

```
GroovyForJavaEyes/Annotations.groovy
@Newify([Person, CreditCard])
def fluentCreate() {
    println Person.new(firstName: "John", lastName: "Doe", age: 20)
    println Person(firstName: "John", lastName: "Doe", age: 20)
    println CreditCard("1234-5678-1234-5678", 2000)
}

fluentCreate()
```

输出表明，借助该注解，可以使用Ruby和Python风格创建实例。

```
Person(John)
Person(John)
CreditCard(1234-5678-1234-5678, 2000)
```

在创建DSL时，`@Newify`注解非常有用，它可以使得实例创建更像是一个隐式操作。

2.10.6 使用@Singleton

要实现单件模式，正常来讲，我们会创建一个静态字段，并创建一个静态方法来初始化该字段，然后返回单件实例。我们必须确保该方法是线程安全的，同时还要决定是否要惰性创建该单

件。而通过使用@Singleton变换则完全可以避免这种麻烦，如下面例子所示：

GroovyForJavaEyes/Annotations.groovy

```
@Singleton(lazy = true)
class TheUnique {
    private TheUnique() { println 'Instance created' }

    def hello() { println 'hello' }
}

println "Accessing TheUnique"
TheUnique.instance.hello()
TheUnique.instance.hello()
```

当运行这段代码时，实例会在第一次调用instance属性时被创建，该属性会映射到getInstance()方法。

```
Accessing TheUnique
Instance created
hello
hello
```

这里使用@Singleton注解标记了TheUnique类，以生成静态的getInstance()方法。因为此处将lazy属性的值设为了true，所以会将实例的创建延迟到请求时。可以检查一下生成的代码：把前面的代码复制粘贴到groovyConsole中，然后选择Script菜单下的Inspect AST菜单项。

```
public class TheUnique implements
    groovy.lang.GroovyObject extends java.lang.Object {

    private static volatile TheUnique instance
    //...

    private TheUnique() {
        metaClass = /*BytecodeExpression*/
        this.println('Instance created')
    }

    public java.lang.Object hello() {
        return this.println('hello')
    }

    public static TheUnique getInstance() {
        if ( instance != null) {
            return instance
        } else {
            synchronized (TheUnique) {
                if ( instance != null) {
                    return instance
                } else {
```

```

        return instance = new TheUnique()
    }
}
}
//...

```

Groovy不仅将实例创建延迟到了最后责任时刻，还保证创建部分是线程安全的。

警告 使用@Singleton注解，会使目标类的构造器成为私有的，这在我们意料之中，不过因为Groovy实现并不区分公开还是私有，所以在Groovy内仍可使用new关键字来创建实例。但是，必须谨慎恰当地使用这个类，并留心代码分析工具和集成开发环境给出的警告。

除了前面提到的这些，Groovy还提供了一个便于使用的注解，用以解决在继承带有多个构造器的类时所需要做的苦差事。在Java中，即使我们几乎不想调用到相应父类的构造器，它还是会强制我们实现这多个构造器。如果使用@InheritConstructors注解该类，则Groovy会为我们生成这些构造器。

以上是Groovy优秀的一面，但是作为客观的程序员，我们也必须承认，有些东西可能会绊我们一下。下一节就来介绍一些陷阱，帮助我们在需要时保持警惕。

2.11 陷阱

阅读本书过程中，我们将看到Groovy的很多不错的功能，但是在使用Groovy时也确实存在一些“陷阱”——从小小的烦恼到可能令你吃惊的问题。接下来，我们将探讨一些陷阱。

2.11.1 Groovy 的==等价于Java的equals()

在Java中，==和equals()是一个混乱之源，而Groovy加剧了这种混乱。Groovy将==操作符映射到了Java中的equals()方法。假如我们想比较引用是否相等（也就是原始的==的语义），该怎么办呢？必须使用Groovy中的is()。下面通过一个例子来理解其区别。

GroovyForJavaEyes/Equals.groovy

```

str1 = 'hello'
str2 = str1
str3 = new String('hello')
str4 = 'Hello'

println "str1 == str2: ${str1 == str2}"
println "str1 == str3: ${str1 == str3}"
println "str1 == str4: ${str1 == str4}"

```

```
println "str1.is(str2): ${str1.is(str2)}"
println "str1.is(str3): ${str1.is(str3)}"
println "str1.is(str4): ${str1.is(str4)}"
```

来看一下Groovy中`==`操作符的行为以及使用`is()`方法的结果：

```
str1 == str2: true
str1 == str3: true
str1 == str4: false
str1.is(str2): true
str1.is(str3): false
str1.is(str4): false
```

观察发现，Groovy的`==`映射到`equals()`，这个结论并不总是成立，当且仅当该类没有实现`Comparable`接口时，才会这样映射。如果实现了`Comparable`接口，则`==`会被映射到该类的`compareTo()`方法。

下面例子说明了这种行为。

GroovyForJavaEyes/WhatsEquals.groovy

```
class A {
    boolean equals(other) {
        println "equals called"
        false
    }
}

class B implements Comparable {
    boolean equals(other) {
        println "equals called"
        false
    }

    int compareTo(other) {
        println "compareTo called"
        0
    }
}

new A() == new A()
new B() == new B()
```

下面的输出显示，`Comparable`的优先级高：

```
equals called
compareTo called
```

通过输出可以看到，在实现了`Comparable`接口的类上，`==`操作符选择了`compareTo()`，而不是`equals()`。