开始 Groovy

1. 使用入门

你可能急着的编写一些 Groovy 代码, 对吧? 好的, 首先，你必须安装 Groovy. 在这章, 我将告诉你如何快速安装 Groovy 并确保一切在你的系统上运行良好.

1.1 使用 Groovy

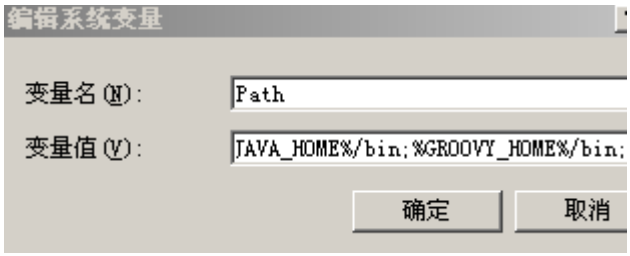
获取稳定的 Groovy 版本非常简单:只要浏览 Groovy 主页 http://groovy.codehaus.org, 点击 Download 链接。 你可以下载 binary release 或者 source release。 假如你想在自己的电脑上构建 Groovy 或者想研究它的源代码可以下载 source release。否则, 推荐你下载 binary release (假如你使用的是 Windows, 你可以下载 Windows 安装版本, 但我建议获取 binary release 自己来设置必要的环境)。 同时你需要 JDK 1.4 或最新的(see http://java.sun.com/javase/downloads/index.jsp). 假如你想体验 Groovy 内 Java 5 支持功能 ，我建议至少 JDK 1.5。最后，确定你已经安装了 Java 。

1.2 安装 Groovy

在下面的章节里， 我假设你已经下载了 Groovy 1.5.7 并已经安装了 JDK .

在 Windows 安装 Groovy

假如你下载的是 Windows 安装版本, 运行它, 按照指示安装。
假如你下载的是 binary release , 解压. 把 groovy-1.5.7 复制到一个你想要的位置。比如, 我的 Windows 系统：C:\groovy-1.5.7 .
下面就是设置 GROOVY_HOME 和 Path 环境变量. 编辑你的系统环境变量 (这里的位置就不要我说了吧，如果不会的话，去看下 Java 怎么设置). 创建环境变量并命名为 GROOVY_HOME, 并设置成你的 Groovy
安装路径(例如, 我设置成 C:\groovy-1.5.7 .). 同样,在 Path 中设置为%GROOVY_HOME%\bin . 注意使用(;)分隔你的环境变量.
接下来，配置的 JAVA_HOME 为你的 JDK 目录 (假如你还没有设置).



在Unix-like 系统安装Groovy

解压你下载的 binary release . 把 groovy-1.5.7 复制到一个你想要的位置. 例如： 我的 Mac 系统, /opt/groovy 目录.
下一步是设置 GROOVY_HOME 和 Path 环境变量. 根据你所使用的 shell , 你必须编辑不同的配置文件. 我使用 bash,因此,编辑 ~/.bash_profile 文件. 在这个文件里, 我添加了 GROOVY_HOME="/opt/groovy/groovy-1.5.7" 来设置 GROOVY_HOME 环境变量. 同样在 Path 中添加$GROOVY_HOME/bin 环境变量.
接下来就是配置 JAVA_HOME （这里不再介绍）

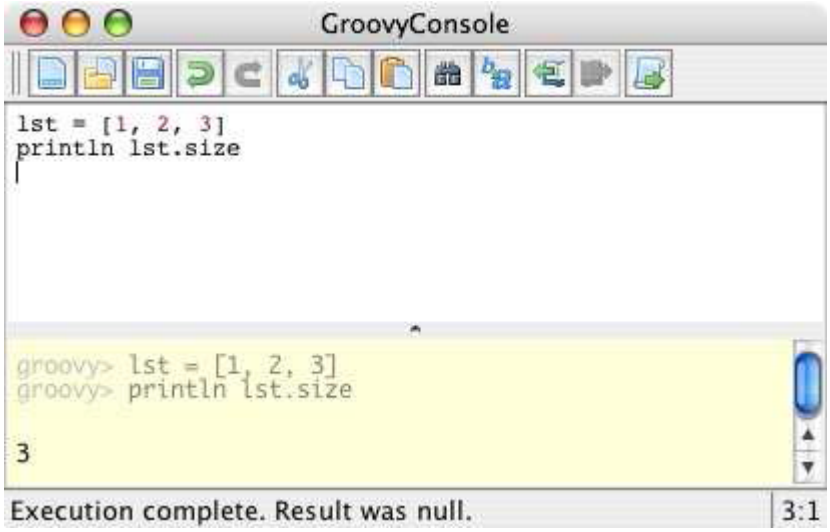
1.3 试运行 Using groovysh

OK, 你已经安装了 Groovy，为了检测安装是否正确可以使用命令行工具 groovysh. 打开 Window 命令提示符, 键入 groovysh; 将看到下图的内容：

```
~>groovysh
Groovy Shell (1.5.4, JVM: 1.6.0_01-41-release)
Type 'help' or '\h' for help.
```

1.4 使用 groovyConsole

假如你不喜欢命令行而是喜欢 GUI, —那么在 Windows 下双击 groovyConsole.bat (在 %GROOVY_HOME%\bin 目录下能找到他).如果是 Unix-like 系统可以双击 groovyConsole 执行脚本.一个控制台 GUI 弹出, 如图：



在上面的控制平台窗口输入些代码, 在 Windows 系统上可以按下 Ctrl+R 或者 Ctrl+Enter （Mac 系统上 Command+R 或者 Command+Enter）执行代码.
你也可以使用工具条上的按钮执行脚本.
使用时间长后你便会喜欢上 groovyConsole —你可以保存你的脚本, 打开已存在的脚本等等, , 所以可以花些时间来研究它

2. Groovy for the Java Eyes

在这章我会帮助你轻松的认识 Groovy. 首先，从熟悉的 Java 然后再转换到 Groovy 的写法。由于 Groovy 保留 Java 语法和语义, 你可以随时混合 Java 样式 和 Groovy 样式.现在开始我们的 Groovy 之旅.

2.1 从 Java 到 Groovy

Groovy 能直接接受你的 Java 代码. 因此, 从一段你熟悉的代码开始, 但是通过 Groovy 运行. 当你工作时,你将发现做同样事情的代码，使用 Groovy 方式来书写后非常小巧.

Hello, Groovy

这是 Java 代码同样也是 Groovy 代码:

```
// Java code
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!" );
}
}
FRO
```

默认导入

在你书写 Groovy 代码时没有必要导入一些公用的 类/包. 例如, Calendar 直接指向 java.util.Calendar.

Groovy自动导入以下Java包: java.lang, java.util, java.io, 和java.net. 它同样也导入了java.math.BigDecimal和 java.math.BigInteger.
另外, Groovy的 groovy.lang 和groovy.util 包也被导入.

下面是上面代码的输出:ho ho ho Merry Groovy!
如此简单的工作，却需要这么多代码. 然而, Groovy 会乖乖的接受并运行它. 以 Greetings.groovy 的文件名保存上面的代码, 使用命令 groovy Greetings 来执行.
其实, 你可以去除上面程序的大部分代码并得到相同的结果.
第一步,先去除上面的分号. 去除分号不仅可以减少干扰同时帮你使用 Groovy 实现内部 DSLs(后面会介绍 DSLs in Groovy).
然后再删除类和方法的定义.

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

你可以更进一步.
Groovy 知道 println( )因为它已经被添加到 java.lang.Object.
它同样有轻巧的 For 循环格式（使用 Range 对象--- Groovy 有宽松的圆括号）. 因此上面的代码可以减少成这样:

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

这个输出和你一开始的Java代码是相同的, 只是这里的代码更加简洁.

2.2 循环方式

在 Groovy 中你不局限于传统的 For 循环. 你已经在 For 循环中使用了 range 0..2 .别急,还有更精彩的.
Groovy 已经在 java.lang.Integer 中添加了便捷的 upto( )实体方法, 因此，你可以像下面这样使用这个方法:

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

这里，你用 Integer 实体 0 调用 upto( ). 上面代码的输出:0 1 2

upto( )方法接受一个闭包作为参数. 假如闭包只有一个参数, 在 Groovy 中可以使用默认名 it 代表这个参数. 记住这点, 我将在后面章节讨论如何使用 Closures（闭包）. 变量前的 $ 告诉 println( ) 打印出变量的值，而不是变量的字符 “it”—允许你嵌入到 strings，你将在后面章节了解到 upto( )方法允许你设置上下界限. 假如你从 0 开始, 你还可以使用 times( ) 方法, 如下:

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

代码输出如下:0 1 2
如果你想在循环的时候跳读值, 可以使用 step( )方法:

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

代码输出如下:0 2 4 6 8

你已经在实践中看到了简单的循环.你同样可以在集合的对象中使用简单的迭代和横切方法, 你将在后面的章节看到
更进一步, 你可以重写 greetings 实例 使用早前学过的方法.看看 Groovy 是如何简化在开始时的 Java 代码:

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

代码输出如下:ho ho ho Merry Groovy!

2.3 快速查看 GDK

Groovy 扩展的 JDK 被称为 GDK2 或者 Groovy JDK. 在 Java 中, 你可以使用 java.lang.Process 与系统级进程相互作用. 假如你在代码中调用 Subversion Help:

```
//Java code
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 非常有用,但是 GDK, 却使它意想不到的简单:

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

比较两段代码.那段代码你会更喜欢了?
当你在 String 实体上调用 execute( )时, Groovy 创建一个继承自 java.lang.Process 的实体, 就如在 Java 代码中 Runtime 的 exec( )方法:

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

在 Unix-like 机器上的输出:
java.lang.UNIXProcess
在 Windows 机器上的输出:
java.lang.ProcessImpl

当你调用 text, 你将在 Process 中调用 Groovy 添加的方法 getText( ) 来读取进程的整个标准，并把结果输出进一个 String 中.
来吧, 试下你上面的代码.
假如你没有使用 Subversion, 而是使用 groovy -v 来代替:

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

输出:Groovy Version: 1.5.4 JVM: 1.6.0_01-41-release

上面代码在 Unix-like 系统和 Windows 都能正常工作.
类似的, 在 Unix-like 系统, 获取当前目录列表,你可以调用 ls:

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

如果是 Windows, 用 dir 替代 ls 将无法工作. 原因是 ls 是 Unixlike 系统的程序 , 而 dir 却不是一个程序—它只是一个 shell 命令. 因此，你必须调用 cmd，然后要求它执行 dir 命令:

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

在这节, 你只不过是接触到 GDK的表面. 在后面的章节还会接触更多的GDK.

2.4 安全导航操作符

安全导航操作符 (?.). 消除普通的 null 检查:

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


foo( )方法中的?. 操作符当 str 不为 null 时才调用 str.reverse().

输出如下:

```
live
null
```

在 null 对象上使用?.来替代 NullPointerException.

另一个用处是异常处理.

在 Java 中会强迫你进行异常处理检查. 考虑一个简单的案例: 当你调用 Thread 的 sleep( ) 方法.Java 强迫你捕获 java.lang.InterruptedException.

这导致了大量空的 catch 块:

```
// Java code
try
{
Thread.sleep(5000);
}
catch(InterruptedException ex)
{
// eh? I'm losing sleep over what to do here.
}
```

空的 catch 块比没有进行异常处理更糟糕 .

Groovy 不会强迫你处理异常. 任何异常你都不需要处理，它被传递到高的层次. 这里有个实例来解答 Groovy 的异常处理:

```
def openFile(fileName)
{
new FileInputStream(fileName)
}
```

openFile( ) 没有处理臭名昭著的 FileNotFoundException.

假如发生异常, 并不会阻止.而是传递给呼叫代码处理:

```
try
{
openFile("nonexistentfile" )
}
catch(FileNotFoundException ex)
{
// Do whatever you like about this exception here
```

```
println "Oops: " + ex
}
```
如果你关心所有抛出的异常,你可以写一个 catch:
```
try
{
openFile("nonexistentfile" )
}
catch(ex)
{
// Do whatever you like about this exception here
println "Oops: " + ex
}
```
catch(ex)变量 ex 没有使用任何类型, 因此，可以捕获任何异常. 小心, 这里并不是捕获 Errors 或者 Throwables 以外的 Exceptions. 如果你需要捕获所有这些可以使用 catch(Throwable t).
正如你说看到的, Groovy允许你把重点放在让您的工作，而不是解决恼人的系统级细节.

2.5 Groovy 作为轻量级的 Java

Groovy 的其他功能,使它更轻,更易于使用:

- 返回声明几乎是可选的 .
- ; 几乎是可选的 虽然可以用于单行声明
- Methods（方法）和 Classes（类） 默认为 public.
- ?.运算符只在对象应用不为空时调用.
- 初始化 JavaBeans 可以使用命名参数.
- 如果你不想，它不会强迫你捕获异常.

他们传递给调用的代码.

- 你可以使用内部 static 方法来引用这个 Class 对象.

例如,下面的代码, learn( ) 返回类,因此,你可以使用链式调用 learn( )方法:

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

3.4 实现接口

在 Groovy 中你可以使用 Map 和代码块快速的实现接口.在这个部分, 首先会用 Java 方式来实现接口, 接着教你如何利用 Groovy 工具.
这里是一个熟悉得不能再熟悉的 Swing JButton 事件处理程序. 为了调用 addActionListener( ) 你必须实现 ActionListener 接口. 因此, 你就必须得创建匿名的内部类来实现 ActionListener, 而且还必须实现
actionPerformed( ) 方法.

```
// Java code
button.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent ae)
{
JOptionPane.showMessageDialog(frame, "You clicked!" );
}
});
```

Groovy 带来了迷人而通用的方法—不需要 actionPerformed( ) 方法或创建任何的匿名类!

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

这里你只需要在 addActionListener 方法中提供一个代码块，通过 as 操作符来实现 ActionListener .
就这么简单—Groovy 帮你搞定其他的一切. Groovy 拦截所有的接口调用(actionPerformed( ))
假如你想在鼠标点击或离开更新 Label 上的信息. 要是用 Java,你必须实现 MouseListener 和 MouseMotionListener 接口总共 7 个方法. 还好,Groovy 更加容易的处理(非常牛 B):

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

在这里, 你创建 displayMouseLocation 变量引用一个代码块. 然后使用 as 来实现 MouseListener 和 MouseMotionListener. 在上面的代码, 你又一次看见了 it 变量. it 代表方法的参数.现在你不必了解这个 it 变量，因为在后面介绍闭包的时候会详细的讲解（现在你只要记住怎么实现接口的）OK, 现在看上去感觉非常好,不过很多现实情况是： 你需要实现接口的不同方法. 不用担心, Groovy 都能帮你搞定. 只要简单的创建一个 Map—简单的使用(:)来分隔方法名和代码块. . 同样, 你没有必要实现所有的方法（做过 Swing 的人都知道，很多事件处理接口方法其实我们并不需要，我们关心的只是那些需要的，如果所有方法都实现，无疑让我们感觉浪费，也增加我们对那长串代码的厌倦）. 现在让我们来看看实例：

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

```
]
button.addFocusListener(handleFocus as FocusListener)
```

当 button 获得焦点,与之关联的 Key-- focusGained 被调用. 当 button 失去焦点, 与之关联的 Key-- focusLost 被调用. 这里的 Key FocusListenerInterface 接口中的方法.

牛年到了，再让我们看看更牛 B 的用处。

假如你想添加事件处理方式给不同的事件:WindowListener,ComponentListener, ... 当然这些列表是可以动态改变的. 假设你的处理程序执行某些共同的操作,比如 logging 或者更新状态栏—有些任务是测试或调试你的应用程序.你可以使用单个代码块动态添加事件处理器给多个事件：

```
events = ['WindowListener' , 'ComponentListener' ]
// Above list may be dynamic and may come from some input
handler = { msgLabel.setText("$it" ) }
for (event in events)
{
handlerImpl = handler.asType(Class.forName("java.awt.event.${event}" ))
frame."add${event}" (handlerImpl)
}
```

你想处理的事件在一个事件列表 events 中. 列表是动态的; 这些不同的事件的事件处理程序在变量 handler 所引用的代码块中. 循环 events 中的每个事件,使用 asType( ) 方法创建每个接口的实现. 在代码块上调用此方法 并使用 forName( )方法获得接口的 Class metaobject 作为参数.

上面的代码, 在代码块上使用 asType( ) 方法.

假如你不同的方法有不同的实现, 可以使用 Map 代替单一的代码块. 这样的话, 你可以用简单的方式在 Map 上调用 asType( )方法 . 最后, 给出一个实例让大家慢慢研究:

```
import javax.swing.*
import java.awt.*
import java.awt.event.*
frame = new JFrame(size: [300, 300],
layout: new FlowLayout(),
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' ]
// Above list may be dynamic and may come from some input
handler = { msgLabel.setText("$it" ) }
for (event in events)
{
handlerImpl = handler.asType(Class.forName("java.awt.event.${event}" ))
frame."add${event}" (handlerImpl)
}
frame.show()
```

3.5 Groovy boolean 值

Groovy 与 Java 的 boolean 值有着本质的区别. 根据不同的情况, Groovy 会自动将表达式的值作为 boolean 看待.
让我们来看看具体的实例. 下面的Java代码不能正常运行:

```
//Java code
String obj = "hello" ;
int val = 4;
if (obj) {} // ERROR
if(val) {} //ERROR
```

Java 必须要你在 if 条件语句中提供一个 boolean 表达式值. 上面的代码必须改成 if(obj != null) 和 if(val > 0) 才能正常运行.
Groovy 就不会那么挑剔了. 如果你把一个对象放在一个 boolean 表达式中,Groovy 会检查这个对象是否为 null. 它会把 null 当做 false,否则为 true, 就像下面的代码:

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

输出为: hello

上面是相对于单个对象，那么对于集合(比如 java.util.ArrayList)这样的类型，它会怎么判断了.Groovy 会检查集合是否为空来判断，因此,像 if (obj)为 true 的条件只在 obj 不为 null 且集合必须拥有至少一个元素:

```
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'
```

Type	为 true 的条件
Boolean	true
Collection	不为空
Character	值不为 0
CharSequence	长度> 0
Enumeration	有更多元素
Iterator	有文件
Number	双精度值不为 0
Map	不为空
Matcher	至少一个匹配

Object[ ]	长度> 0
任何其他类型	引用不为 null

上面的代码输出:

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

3.6 操作符重载

你可以使用 Groovy 实现操作符重载, 那么 Groovy 是怎么做到的了？其实非常简单, 事实上—每个运算符都有一个标准的映射方法.因此, 在 Java 中你可以使用这些方法,对于 Groovy 你可以使用操作符或操作符对应的方法,这两者任意一种.
这里一个实例展示操作符重载:

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

你使用'++'操作符来循环'a'---'c'中的字符.这个'++'操作符映射到 String 类的 next( ) 方法. 上面代码的输出:

```
a
b
c
```

我们再来看一个，其他这个 JDK1.5 的循环在这里也是使用 String 的 next( ) 方法:

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

String 类有许多重载操作符, 你会在后面看到. collection 类—ArrayList 和 Map—也有方便的重载操作符.假如你想添加一些元素到 collection 中, 你可以使用 << 操作符,它会转化为 Groovy 为 Collection 添加的额外方法 leftShift( ) ,像这里:

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

输出:["hello" , "there"]

你可以通过添加映射方法来为自己的类提供操作符, 像 plus( ) 对应+.
这里将展示一个操作符重载:

```
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
```

因为你 ComplexNumber12 类中添加了 plus( ) 方法,Groovy 允许你使用 + 获取两个复数相加的结果 ,输出:5 + 3i

在某些方面操作符重载使表达式更具表现力. 不过, 通常情况下我并不喜欢操作符重载,因为它很难被理解.

3.8 一些小小的问题

return 并不总是可选的

Groovy 中，方法最后的 return 语句是可选的:

```
def isPalindrome(str) { str == str.reverse() }
println "mom is palindrome? ${isPalindrome('mom')}"
```

输出:

```
mom is palindrome? true
```

不过上面的代码并不总是正确的，如果是在条件语句中的声明:

```
def isPalindrome2(str)
{
if (str)
{
str == str.reverse()
}
else
{
false
}
}
println "mom is palindrome? ${isPalindrome2('mom')}"
```

输出:

```
mom is palindrome? null
```

如果我们添加return的话:

```
def isPalindromeOK(str)
{
if (str)
{
return str == str.reverse()
}
else
{
return false
}

}

println "mom is palindrome? ${isPalindromeOK('mom')}"
```

输出:

```
mom is palindrome? true
```

个人认为，如果你不至于懒到连几个字都不愿意打的话，最好明确的返回.

Groovy 的 == Is 相当于 Java 的 equals

== 和equals( ) 已经在 Java中引起混乱, 而Groovy增加你混乱. Groovy 映射 == 操作符为Java中的equals( ). 如果想进行对象的比较，那你就得使用is( ):

```
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)}"
```

This is the output from the previous code:

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

有人认为, Groovy 的 == 映射为 equals( ) 只是部分正确—那就是看你的类是否实现 Comparable 接口. 如果实现, 它便会映射为 compareTo( )方法.

比如:

```
class A
{
boolean equals(other)
{
println "equals called"
false
}
}
class B implements Comparable
{
boolean equals(other)
{
println "equals called"
}
int compareTo(other)
{
```

```
println "compareTo called"
0
}
}
new A() == new A()
new B() == new B()
```

代码输出可以看出，如果你的类实现 Comparable 接口，则 compareTo( ) 方法优先于 equals( ) 方法:

```
equals called
compareTo called
```

所以在使用时，你就得非常谨慎，首先得看看到底是比较对象的引用，还是值，来选择正确的操作符.

没有编译时类型检查

Groovy 是可选类型; 不过, Groovy 编译器, groovc, 不会进行完整的类型检查. 作为替代, 只有在遇到类型定义时才执行类型转换. 它还会检查你使用的类是否存在. 例如:

```
Integer val = 4
val = 'hello'
```

这段代码编译没有问题. 当你运行创建的 Javabytecode created, 将会抛出 GroovyCastException 异常:

```
org.codehaus.groovy.runtime.typehandling.GroovyCastException:
Cannot cast object 'hello' with class 'java.lang.String'
to class 'java.lang.Integer'
```

Groovy 编辑器, 不会检查类型, 只是简单的转换它并把它留在运行时处理. 你可以在产生的 bytecode 证实这点 (可以使用 javap -c ClassFileName 命令查看):

```
...
58: ldc #71; //String hello
60: getstatic #74; //Field class$java$lang$Integer:Ljava/lang/Class;
63: ifnonnull 78
66: ldc #76; //String java.lang.Integer
68: invokestatic #21; //Method class$:(Ljava/lang/String;)L...
71: dup
72: putstatic #74; //Field class$java$lang$Integer:Ljava/lang/Class;
75: goto 81
78: getstatic #74; //Field class$java$lang$Integer:Ljava/lang/Class;
81: invokestatic #80; //Method org/codehaus/groovy/runtime/Scri...
84: checkcast #65; //class java/lang/Integer
87: dup
88: astore_3
89: aload_3
90: areturn
```

...
因此, 在 Groovy 中, x = y 在语义上相当于 x = (ClassOfX)(y).假如你调用不存在的方法，在编译时也不会有任何的错误:

```
Integer val = 4
val.blah()
```

它会在运行时抛出MissingMethodException. 其实这实际上算得上是一种优势, 你会在后面章节了解的更清楚. 在代码已经编译，但没有运行时,你可以动态注入方法.

```
groovy.lang.MissingMethodException:
No signature of method: java.lang.Integer.blah() is applicable
for argument types: () values: {}
```

Groovy 编译器看上可能有点脆弱;不过, 这对于 Groovy 的动态性和元编程是必须的.

了解新的关键字

def 用来定义方法,属性和本地变量. in 用于在循环中(i in 1..10).
假如你把他们作为变量名或方法名, 便会出现问题. 这种情况发生在当你把已有的 Java 代码当做 Groovy 代码使用时.
当然你使用 def 或 in 定义变量也不是什么聪明的想法.

没有内部类

Groovy 不支持内部类.当你使用已有的 Java 代码当做 Groovy 代码使用时会遇到麻烦,如果你在写新的 Groovy 代码 ,你可以利用闭包. 你会在后面章节了解的更清楚

没有代码块

```
// Java code
public void method()
{
System.out.println("in method1" );
{
System.out.println("in block" );
}
}
```

代码块在 Java 中定义了一个范围. 而在 Groovy 中就会产生混淆, 因为他会认为是一个闭包.

分号(;)几乎是可选的

你没有必要在表达式后放置分号 (;) . 如果你想把多条表达式放在同一行, 那就可以放置一个分号来分隔表达式.但有时分号 (;)却不能省略 :

```
class Semi
{
def val = 3
{
println "Instance Initializer called..."
}
}
println new Semi()
```

你打算使用代码块来初始化实例.不过, Groovy 会被混淆, 会把这段代码块当中闭包, 并抛出这样的错误:

```
Caught: groovy.lang.MissingMethodException:
No signature of method: java.lang.Integer.call()
is applicable for argument types: (Semi$_closure1)
values: {Semi$_closure1@be513c}
at Semi.<init>(SemiColon.groovy:3)
at SemiColon.run(SemiColon.groovy:10)
at SemiColon.main(SemiColon.groovy)
```

把 def val = 3 改成 val = 3;, 就运行良好了. 现在 Groovy 会正确处理代码块的用处, 假如你使用静态初始化就不会有这样的问题了.

不同的语法创建原始数组

在 Groovy 中, 如果你想创建原始数组, 你不能使用 Java 语法.假如你在 Java 中创建 integer 数组:

```
int[] arr = new int[] {1, 2, 3, 4, 5};
```

这段代码在 Groovy 中就不能运行. 实际上,你会得到编译错误.
如果用 Groovy 来定义原始数组，语法是这样的:

```
int[] arr = [1, 2, 3, 4, 5]
println arr
println "class is " + arr.getClass().name
```

输出的类型[I, 而 JVM 会把它当做 int[ ].

```
[1, 2, 3, 4, 5]
class is [I
```

4.2 动态类型

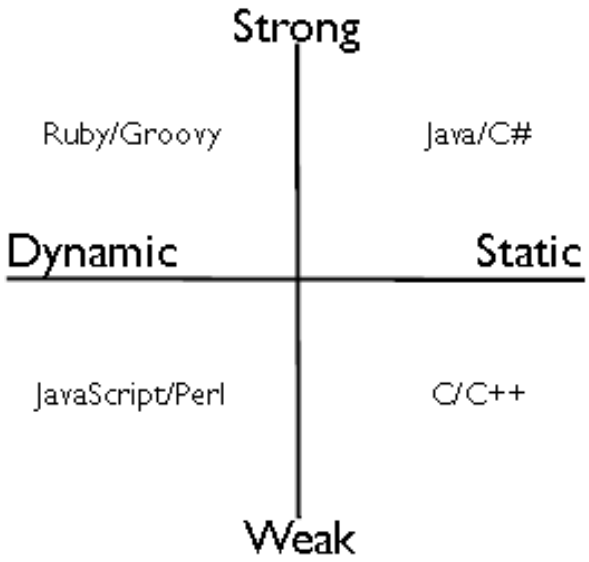
动态类型对类型的要求很宽松. 基本上, 你可以让语言根据上下文来判断类型.很多动态语言是动态类型的, 但也有不少动态语言是静态类型的.那动态语言的优势在哪里? 动态类型提供了 2 个优势，我个人认为利大于弊.

在你书写对象方法调用时，没有必要向下抓取细节，在运行时,对象会自动响应方法或消息. 当然，你可以使用静态语言的多态性来完成. 不过, 大多数静态语言捆绑了继承在多态性上. 他们强迫你符合这些结构而不是真正的行为. 真正的多态性不需要关心类型—发送一条消息给对象, 并在运行时选择合适的实现来使用. 因此,动态类型能帮助你实现比传统静态语言更大程度的多态性.

第 2 个好处是你不需要迫使编译器进行琐碎的类型转换, 在 4.1, Typing in Java, 你已经看到了.

4.3 动态类型 != 弱类型

在静态类型语言中, 你要指定变量，引用等等的类型,因为, 在编译时编译器要求你这样. 拿C/C++来说. 你必须指定变量的原始类型,像 int, double, ..., 或者指定 Class 类型. 然而,如果你把变量转换成一个错误类型会怎样?编译器停止工作? 不. 那么程序在你运行时命运又会怎样了? 这要看情况而定.假如你走运的话, 程序会崩溃.假如不走运, 他会一直等待，直到实例崩溃. Java 同样是静态类型, 但它是强类型. 编译器会检查类型, 假如你给出一个错误类型 会在运行时捕获.



像 Groovy 这样的动态语言不会在代码编辑时或编译时进行类型检查. 不过,你给出一个错误类型, 你会在运行时得到一条错误消息. 也就是说他会把类型检查推迟到运行时; 这就允许你在运行和书写/编译这段时间内动态修改程序结构. 这就告诉我们动态类型未必就是弱类型.

4.4 设计能力

作为 Java 程序员很大程度上依赖接口（interfaces）.“设计规约”的重要意思 :当我们在接口（ interfaces）定义了通信规约 ， 实现接口的类就必须遵守这些规约.
基于接口编程,虽然非常强大，但往往被局限于某一点.让我们看看下面的代码（不同的地方用语法高亮显示）.

使用静态类型

我需要搬动一些沉重的东西. 于是我问，那位男士愿意帮助我. 在 Java 中, 代码如下:

```
public void takeHelp(Man man)
{
//...
man.helpMoveThings();
//...
}
```

因为是强类型, 这里忽视乐意帮助我的女士. 让我们来扩展下，让我所求助的人不分男士和女士. 我将创建一个包含 MoveThings( )方法的 Human 抽象类. Man 和 Woman 会提供 MoveThings( )方法各自的实现:

```
// Java code
public abstract class Human
{
public abstract void helpMoveThings();
//...
}
```

现在代码需要 Human 来帮助:

```
public void takeHelp(Human human)
{
//...
human.helpMoveThings();
//...
}
```

现在任何 Human 都能帮助我. 不过, 假如我是森林管理员, 我就不能依靠大象（Elephant）的优势来帮助我了. 我必须依靠 Human,而大象（Elephant）却不符合规约. 我们只好继续扩展, 现在我们需要的是带有 helpMoveThings( )方法的接口 Helper:

```
// Java code
public interface Helper
{
public void helpMoveThings();
}
```

Human, Elephant 和所有实现 Helper 的帮助者. 现在我就可以得到所有实现 Helper 接口的帮助

者的帮助:

```
public void takeHelp(Helper helper)
{
//...
helper.helpMoveThings();
//...
}
```

使用动态类型

让我们使用 Groovy 的动态能力重写上面的代码:

```
def takeHelp(helper)
{
//...
helper.helpMoveThings()
//...
}
```

takeHelp( )接受一个 helper 但并没有指定它的类型——默认为一个对象. 在这里 helper 并没有显示的实现任何接口,其实是使用了对象隐式接口能力. 这个叫 duck typing（如果它走起来像鸭子，叫起来像鸭子,那么它就是鸭子）.
类只关心他所实现的方法而不必继承或实现任何接口类型. 结果是高产低拘束. 现在, 我们可以使用这些类而不需要改变我的任何代码.

```
class Man
{
void helpMoveThings()
{
//...
println "Man's helping"
}
//...
}
class Woman
{
void helpMoveThings()
{
//...
println "Woman's helping"
}
//...
}
class Elephant
{
void helpMoveThings()
{
//...
```

```
println "Elephant's helping"
}
void eatSugarcane()
{
//...
println "I love sugarcanes..."
}
//...
}
```

调用takeHelp( )方法:

```
takeHelp(new Man())
takeHelp(new Woman())
takeHelp(new Elephant())
```

输出:

```
Man's helping
Woman's helping
Elephant's helping
```

态类型需要自律动

现在是否认识到动态类型给你的代码带来的简单, 优雅, 和灵活? 但是, 它会是危险的行为吗?

- 如果你创建 helpers.错误的键入方法名
- 没有类型信息, 你怎么知道什么传递给你的方法?
- 如果你传递一个 nonhelper（不能搬动一些沉重的东西.）会是什么样?

问的很好, 别被这些东西搞得你忧心忡忡的.

现在让我们来讲解这些问题.

Chapter 5

闭包

闭包可能是你以后使用最多的 Groovy 特性之一。你可以传递闭包给方法并调用它们。 实际上，GDK 最大的贡献就在于通过闭包来扩展了 JDK 的方法。闭包为你提供了强大的函数指针，而不是优雅的对象和便捷的 duck typing。 一旦你掌握闭包的诀窍后, 你就会很好的运用在自己的项目中。因此。这章节，将学习闭包 。

5.1 闭包

假如你有个函数用来处理集合的值或者对象集合。可能是在选择的值上执行不同的操作。这些被选择的值可能是一个数组，那么首先看看闭包是如何处理的。
下面的代码是求 1---n 的和

```
def sum(n)
{
total = 0
for(int i = 2; i <= n; i += 2)
{
total += i
}
total
}
println "Sum of even numbers from 1 to 10 is ${sum(10)}"
```

sum( )方法中，你会遍历所有元素并计算他们的和。
如果你现在要求 1 --- n 的乘积:

```
def product(n)
{
```

```
prod = 1
for(int i = 2; i <= n; i += 2)
{
prod *= i
}
prod
}
println "Product of even numbers from 1 to 10 is ${product(10)}"
```

你再一次遍历了所有的元素，然后求的乘积。
如果还想把 1---n 开平方后的结果存入集合中:

```
def sqr(n)
{
squared = []
for(int i = 2; i <= n; i += 2)
{
squared << i ** 2
}
squared
}
println "Squares of even numbers from 1 to 10 is ${sqr(10)}"
```

上面所有的代码，都是使用 1---n 来执行想要的操作. 你们觉得上面 3 个方法有什么不同之处？如果你还想对 1---n 执行其他的操作，还会重复的遍历 。个人认为这不适合极限编程，应该提取出类似的代码。

Groovy 的方式

让我们写一个提取偶数的函数。一旦找出偶数，就立即放进代码块执行，让代码块来简单的输出这个偶数:

```
def pickEven(n, block)
{
for(int i = 2; i <= n; i += 2)
{
block(i)
}
}
pickEven(10, { println it } )
```

pickEven( )迭代所有的值, 然后把合适的值传递给代码块-------闭包。这个可变的代码块持有一个对闭包的引用。 正如你传递对象一样，你也可以传递闭包。你不必为代码块命名，它可以是任何合法变量名。当你调用 pickEven( )时你可以像上面一样传递一个代码块。这个代码块({})传递给参数block，10被传递给变量n。在Groovy中。你想传递多少闭包进来都可以。这里要注意的是，如果方法的最后一个参数是闭包，一个简单的语法就像下面这样,:

```
pickEven(10) { println it }
```

Groovy 中的代码块不像 Java，它不能单独存在，必须依附于一个方法或赋值给一个命名变量。
代码块中的 it 是什么？假如你只传递一个参数给代码块，那么这个参数在代码块中就可以使用 it 来引用。当然你可以自己定义一个变量名来引用这个参数:

```
pickEven(10) { evenNumber -> println evenNumber }
```

再来看看 pickEven( ) 方法。现在可以方便的用于和的计算:

```
total = 0
pickEven(10) { total += it }
println "Sum of even numbers from 1 to 10 is ${total}"
```

同样可以用于乘积:

```
product = 1
pickEven(10) { product *= it }
println "Product of even numbers from 1 to 10 is ${product}"
```

闭包就是一个拥有参数的函数被绑定到上下文环境中来运行。
闭包源于 lambda expressions:“ lambda expression 指定参数和函数映射，就是一个包含若干表达式和语句的匿名函数 ” 。