

**注意** 在比较对象时，请首先问一下自己，要比较的是引用还是值。然后再问一下，是不是使用了正确的操作符。

## 2.11.2 编译时类型检查默认为关闭

Groovy的类型是可选的。然而，Groovy编译器groovyc大多数情况下不会执行完整的类型检查(第3章会介绍Groovy如何支持选择性的类型检查)，而是在遇到类型定义时执行强制类型转换。如果要把一个字符串赋给一个Integer类型的变量：

**GroovyForJavaEyes/NoTypeCheck.groovy**

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

这段代码可以正常编译，没有错误。但当尝试运行编译生成的Java字节码时，会出现GroovyCastException异常，在输出中可以看到：

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

Groovy编译器不会验证类型，相反，它只是进行强制类型转换，然后将其留给运行时处理。这可以通过分析生成的字节码来验证（使用javap -c ClassFileName命令可以一窥可读的字节码形式）：

```
...
35: ldc #71 // String hello
37: astore_3
38: aload_3
39: ldc #65 // class java/lang/Integer
41: invokestatic #75 // Method ...castToType:(...)... 
44: checkcast #65 // class java/lang/Integer
...
```

所以在Groovy中，`x = y`在语义上等价于`x = (ClassOfX)(y)`。类似地，如果调用了一个不存在的方法（比如下面例子中调用了不存在的blah方法），也不会出现编译错误：

**GroovyForJavaEyes/NoTypeCheck.groovy**

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

不过运行时会出现MissingMethodException异常：

```
groovy.lang.MissingMethodException:
No signature of method: java.lang.Integer.blah() is applicable
for argument types: () values: []
Possible solutions: each(groovy.lang.Closure), with(groovy.lang.Closure),
```

```
plus(java.lang.Character), plus(java.lang.String), plus(java.lang.Number),
wait()
```

在第13章你将看到，这实际上是个优点。我们可以在代码编译时和执行时动态注入缺失的方法。

Groovy编译器可能看上去不够严格，但是对于Groovy的动态和元编程等强项而言，这种行为是必要的<sup>①</sup>。在2.x版本中，我们可以关闭这种动态类型特性，并增强编译时类型检查，3.8节及3.8.1节将会介绍。

2

### 2.11.3 小心新的关键字

`def`和`in`都是Groovy中的新关键字。`def`用于定义方法、属性和局部变量。`in`用于在`for`循环中指定循环的区间，比如`for(i in 1..10)`。

将这些关键字用作变量名或方法名可能会带来问题，尤其是当把现有的Java代码当作Groovy代码时。

定义名为`it`的变量也是不明智的。尽管Groovy不会抱怨什么，但是如果在闭包内使用了这样的变量，它引用的是闭包的参数，而不是类中的一个字段——隐藏变量可无助于偿还技术债<sup>②</sup>。

### 2.11.4 别用这样的代码块

下面是合法的Java代码：

#### GroovyForJavaEyes/Block.java

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

    {
        System.out.println("in block");
    }
}
```

Java中的代码块定义了一个新的作用域，但是Groovy会感到困扰。Groovy编译器会错误地认为我们是要定义一个闭包，并给出编译错误。在Groovy中，方法内不能有任何这样的代码块。

### 2.11.5 闭包与匿名内部类的冲突

Groovy的闭包是使用花括号（`{...}`）定义的，而定义匿名内部类也是使用花括号。如下面例子所示，当构造器接收一个闭包作为参数时，就出现问题了：

① <http://groovy.codehaus.org/Runtime+vs+Compile+time,+Static+vs+Dynamic>

② <http://martinfowler.com/bliki/TechnicalDebt.html>

**GroovyForJavaEyes/Calibrator.groovy**

```
class Calibrator {
    Calibrator(calculationBlock) {
        print "using..."
        calculationBlock()
    }
}
```

正常情况下，可以通过把一个代码块附到函数调用末尾，将闭包传递给函数：`instance.method() {...}`。按照这种习惯，我们可以通过向`Calibrartor`的构造器传递一个闭包来实例化一个实例，如下面代码所示：

**GroovyForJavaEyes/AnonymousConflict.groovy**

```
def calibrator = new Calibrator() {
    println "the calculation provided"
}
```

在这个例子中，我们是要调用`Calibrator`类的构造器，它接受一个闭包作为参数。事与愿违，Groovy却认为我们是要创建一个匿名内部类，因而报告了一个错误。

```
org.codehaus.groovy.control.MultipleCompilationErrorsException: startup failed:
.../code/GroovyForJavaEyes/AnonymousConflict.groovy:
2: unexpected token: println @ line 2, column 3.
    println "the calculation provided"
^

1 error
```

要绕开这个陷阱，必须修改调用方式，将闭包放在构造器调用语句的圆括号内。我们仍然可以在调用时定义闭包，或传递一个引用该闭包的变量。

**GroovyForJavaEyes/AnonymousConflictResolved.groovy**

```
def calibrator1 = new Calibrator({
    println "the calculation provided"
})
def calculation = { println "another calculation provided" }
def calibrator2 = new Calibrator(calculation)
```

运行这段代码，验证这个版本没有让Groovy迷惑，并且达到了将闭包传递给构造器的预期结果。

```
using...the calculation provided
using...another calculation provided
```

这只是个小麻烦；与传递内联的闭包相比，传递引用给闭包噪音会小一些。

### 2.11.6 分号总是可选的

使用从C语言派生而来的语言的程序员，小拇指可受了不少年的罪，在Groovy中他们可以轻松一下了。因为不必在语句末尾放一个分号（;）。丢掉分号是有好处的，这有助于创建领域特定语言（DSL）。尽管分号是可选的，但是在一行中放置多条语句时，分号还是有用的。至少有一个地方，分号是必不可少的，如下面例子所示：

#### GroovyForJavaEyes/SemiColon.groovy

```
class Semi {
    def val = 3

    {
        println "Instance Initializer called..."
    }
}

println new Semi()
```

我们本打算让这个代码块成为类的一个实例初始化器，但是Groovy迷惑了，它把实例初始化器看成了一个闭包，并给出了如下错误：

```
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)
```

如果把def val = 3改写为def val = 3;，代码即可正常运行。现在Groovy把这个代码块识别成了实例初始化器，而不是附加到属性定义上的一个部分。

如果要使用的是一个静态初始化器，而不是实例初始化器，那就没有这个问题了。如果有理由同时使用这两种初始化器，可以将静态初始化器放在实例初始化器之前，从而避免使用分号。

### 2.11.7 创建基本类型数组的不同语法

要在Groovy中创建基本类型的数组，不能使用我们在Java中所习惯的符号。

在Java中，可以像下面这样创建整型的数组：

#### GroovyForJavaEyes/ArrayInJava.java

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

而在Groovy中，上述代码会导致编译错误。Groovy以如下方式定义基本类型的数组：

**GroovyForJavaEyes/ArrayInGroovy.groovy**

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

println arr
println "class is " + arr.getClass().name
```

输出表明，所创建实例的类型为[I，JVM用它表示int[]]。

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

如果省略掉左侧的类型信息int[]，Groovy会假设我们在创建ArrayList的实例（参见2.9.4节），所以在这个例子中指定类型非常关键。作为一种选择，还可以使用as操作符来创建数组：

```
def arr = [1, 2, 3, 4, 5] as int[]
println arr2
println "class is " + arr2.getClass().name
```

Groovy使得创建ArrayList类型的实例更为容易了，但是要创建数组，则必须付出更多努力。

以上是一些在Groovy中编程时可能遇到的陷阱。因为我们有Java背景，所以可以通过了解Groovy与Java的不同来获益。<http://groovy.codehaus.org/Differences+from+Java>列出了Groovy与Java的差别，很不错。

本章介绍了很多东西。你现在知道了在Groovy中如何编写类，学到了一些Groovy的习惯用法，也了解了一些Groovy编写代码的方式。你还了解到，必要的情况下我们可以回到Java语法。现在，你就可以开始实验和把玩Groovy了，不必等到学完这本书。然而我们还有很多东西要学。术语动态类型和可选类型已经出现过多次，所以下一章将介绍这些主题，并探讨如何在Groovy中利用它们。

动态类型语言中的类型是在运行时推断的，方法及其实参也是在运行时检查的。通过这种能力，可以在运行时向类中注入行为，从而使代码比严格的静态类型具有更好的可扩展性。

本章介绍动态类型的优点，以及如何在Groovy中使用动态类型。借助动态类型，可以用比Java更少的代码创建灵活的设计。将实参的类型验证推迟到运行时这一特性为Groovy中的多态注入了活力。利用多方法（multimethods）这一工具，可以为与实参的运行时类型相关的操作提供替换行为。本章还将介绍如何使用Groovy中的静态编译选项，妥善地利用这些强大的特性。

## 3.1 Java 中的类型

编译时类型检查所提供的“安全性”让人产生了依赖心理。但是类型安全中的安全性和社会保障中的保障同样让人“放心”。

假设Car类有两个属性，一个是year，一个是一个Engine类的实例。现在要实现对Car类对象的复制。先忽略Java中的深度复制问题<sup>①</sup>。为了提供复制功能，需要实现Cloneable接口，并提供一个public的clone()方法。Object类的clone()方法可以创建对象的一个浅副本（shallow copy）。不过这里希望不同的Car实例所包含的Engine也不同。因此，在使用clone()这个基本的方法实现复制时，还需要稍作修改，使其拥有自己的Engine，如下面代码所示：

TypesAndTyping/Car.java

```
//Java代码
public Object clone() {
    try {
        Car cloned = (Car) super.clone();
        cloned.engine = (Engine) engine.clone();
        return cloned;
    } catch(CloneNotSupportedException ex) {
        return null; // 不会发生这种情况，这是为了取悦编译器
    }
}
```

<sup>①</sup> 参见我的文章“Why Copying an Object Is a Terrible Thing to Do”（为什么复制对象很可怕），其中讨论了在Java中复制对象的问题：<http://www.agiledeveloper.com/articles/cloning072002.htm>。

这段代码噪音很大。首先，编译器坚持让我们处理`CloneNotSupportedException`，而且就在实现复制的方法中处理。其次，当在`Car`类的实例方法中调用`super.clone()`时，其实已经能够明确目标是另一个`Car`。然而编译器还是固执地要求对调用结果进行强制类型转换。下一条复制`Engine`的语句也是如此。此外，当准备好在一个`Car`实例上调用`clone()`方法时，还需要再次强制类型转换，以便把调用的结果保存到一个`Car`引用中。有时候静态类型检查只会带来烦恼，而且还会降低效率。好的类型检查应该像一个好政府——只做必要的事，而不能阻碍发展。然而，在大部分时间内，Java编译器都是一个障碍。

编译时类型检查自有其价值。不过如今的集成开发环境（IDE）让开发代码和运行测试变得非常容易，所以往往让IDE来保存相关的编辑文件，并在必要时编译代码。当尝试运行测试失败时，再来解决问题。因此，在重复快速的“编辑-运行-测试”循环的同时，不必再对编译错误、运行时错误和测试失败作太多区分。关键在于让代码持续工作并让所有测试通过。

## 3.2 动态类型

动态类型放宽了对类型的要求，使语言能够根据上下文判定类型。

动态类型有什么优点？放弃在编译时或代码编辑时对类型进行验证或确认的优势，这值得吗？动态类型有两大优点，使这种舍弃利大于弊。

首先，可以在不知道方法具体细节的情况下编写对象上的调用语句。在运行期间，对象会动态地响应方法或消息。在静态类型语言中，使用多态可在某种程度上实现这种动态行为。然而，大部分静态类型语言把继承和多态捆绑在了一起。它们强迫我们遵循某个结构，而不是遵循实际行为。真正的多态并不关注类型——把一个消息发送给一个对象，在运行时，它自会确定所要使用的相应实现。因此，动态类型语言可以实现比传统的静态类型语言更高程度的多态。

其次，不必用大量的强制类型转换操作来取悦编译器，就像3.1节中的例子那样。

一种聪明而且愿意配合程序员的语言，当然谁都愿意使用。工作效率会更为高效，在一定程度上，这都得益于少了很多繁文缛节。

在使用静态语言工作时，那感觉就像是有个唠叨的婆婆站在旁边，盯着我们的一举一动。对于将某些实现推迟到后面某个时间（代码执行前），静态类型语言也没有提供充分的灵活性。相反，使用动态类型语言工作时，就像有位和蔼的爷爷站在旁边，让我们做实验，把事情弄清楚，保持创造性，而他只是站在一旁，在必要时才提供协助。

第一个优点（真正实现了多态）极大地改进了设计应用的方式，3.4节还会详细讨论。

### 3.3 动态类型不等于弱类型

在静态类型语言中，要在编译时指定变量、引用等数据的类型，而且很多编译器会坚持要求这么做。比如在C/C++中，必须指定变量类型，要么是诸如int或double等基本类型，要么就是某个具体类的类型。然而，如果把该变量强制转换为一个错误类型，又会怎么样呢？编译器会阻止吗？并不会。运行时，程序的命运又当如何？那得看运气。如果幸运的话，程序会崩溃。如果不幸，可能会一直等到我们做重要的演示时才出现崩溃或错误行为。根据内存如何配置，调用是否多态，虚函数表如何组织<sup>①</sup>等不同条件，最终表现也很难预测。如果仔细去听，可能还会听到编译器的嘲笑，嘲笑我们依赖它假装提供的类型安全。这是静态类型和运行时弱类型相结合的一个例子。

在下面的图中，我们基于静态对比动态、强类型对比弱类型对常见语言做了分类。

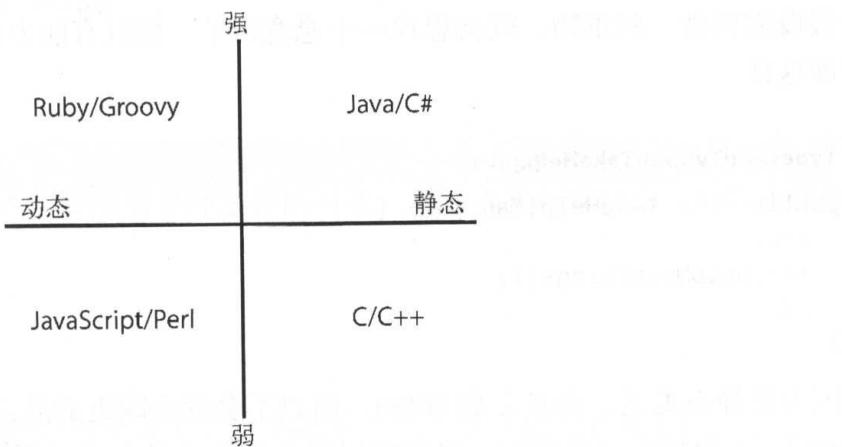


图3-1 所选语言的分类：静态对比动态、强类型对比弱类型

Java是一种静态类型语言，但它是强类型的。编译器会检查类型，但是如果我们强制转换成了某个错误类型，运行时会给我们抓出来。

动态类型语言，比如Groovy，不会在代码编辑时或编译时执行类型检查。然而，如果把一个对象当作错误的类型，Groovy会在运行时毫不含糊地提示出来。把实际的验证推迟到了运行时；我们得以在编码和编译时，以及在代码执行时修改程序的结构。JVM上的动态类型语言说明，动态类型并不意味着弱类型。

<sup>①</sup> 有些语言，比如C++，会维护一个方法分派表，其中保存了多态方法的地址，参见Margaret A. Ellis和Bjarne Stroustrup合著的*The Annotated C++ Reference Manual* [ES90]一书。

## 3.4 能力式设计

作为Java程序员，我们严重依赖接口。我们推崇“契约式设计”（Design By Contract），在这种设计中，接口定义了交流的契约，类负责实现并遵守这些契约——参见Bertrand Meyer的*Object-Oriented Software Construction*[Mey97]一书。

商业契约是个好东西，可以帮助确保特定的预期目标得以实现。然而，契约最好不要太严格，要有一定的灵活性，以便以可接受的方式满足或超出预期。

软件契约也是类似的。基于接口的编程，尽管非常强大，但往往有很多限制。下面的例子突出了静态类型和动态类型之间的差别。

### 3.4.1 使用静态类型

假设需要搬一些重物，就会想找一个愿意帮忙、而且有能力的帮手。在Java中，代码可以写成下面这样：

#### TypesAndTyping/TakeHelp.java

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

因为是静态类型，而且参数为`Man`，所以不会理会附近的某位愿意帮忙、而且有能力的女人（`Woman`）。下面扩充这段代码，以便可以寻求男人或是女人的帮助，来创建一个`Human`抽象类，其中包含`helpMoveThings()`方法。`Man`和`Woman`都可以提供对该方法的实现：

#### TypesAndTyping/Human.java

```
// Java代码
public abstract class Human {
    public abstract void helpMoveThings();

    //...
}
```

下面是接受一个`Human`的帮助的代码：

#### TypesAndTyping/TakeHelp.java

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

好了，现在任何人都可以帮我们搬东西了。然而，如果我们是塞伦盖蒂平原上的流浪者，一头友好的大象或许能提供帮助，但我们却没法利用它的友善——因为依赖的是Human，大象不会遵从这个契约。又该加以扩充了，这次引入一个Helper接口，其中包含helpMoveThings()方法：

#### TypesAndTyping/Helper.java

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

然后Human、Elephant，只要可以提供帮助，都可以实现Helper。我们现在依赖Helper，可以接受来自实现了该接口的实例的帮助：

3

#### TypesAndTyping/TakeHelp.java

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

至此，扩展需要付出很多努力。要使用多种多样的对象，意味着要创建接口，并修改依赖接口的代码。

### 3.4.2 使用动态类型

我们使用Groovy的动态类型能力再来看上节所举的那个“获得帮助”的例子：

#### TypesAndTyping/TakeHelp.groovy

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

takeHelp()接受一个helper，但是没有指定其类型，这样类型默认为Object。此外，这里在它上面调用了helpMoveThings()方法。这就是能力式设计（Design By Capability）<sup>①</sup>。不同于让helper遵守某些显式的接口，我们利用了对象的能力——依赖一个隐式的接口。这被称作鸭子类型，它基于这一观点：“如果它走路像鸭子，叫起来也像鸭子，那它就是一只鸭子。”<sup>②</sup>

想要这种能力的类只需要实现该方法，而不需要扩展或实现任何东西。这样的结果就是少了

<sup>①</sup> 此方式与“契约式设计”相对应，书中多处对比这两种设计方式，因此参考“契约式设计”译为“能力式设计”。  
——译者注

<sup>②</sup> <http://c2.com/cgi/wiki?DuckTyping>

繁文缛节，增加了生产效率。如果机器有这种能力，可以帮我们搬东西，则不再需要修改代码，就能拿来使用。来看一些提供了我们想要的能力的类。

#### TypesAndTyping/TakeHelp.groovy

```
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..."
    }
    //...
}
```

下面是一个调用takeHelp()方法的例子：

#### TypesAndTyping/TakeHelp.groovy

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

再来看一下各自的效果：

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

这些类没有扩展任何公共类，也没有实现任何公共接口，但是借助Groovy的动态特性，我们能够在takeHelp()方法中使用所有这些类。

因为来自Java背景，我们可能需要付出些努力才能习惯Groovy的动态特性，但是一旦感觉对了，就可以好好利用了。例如，在一个订单处理系统中，可以使用一个模拟（Mock）对象毫不费力地替换掉一个信用卡处理对象，以便进行快速的自动化测试，而不必提前做出优雅的设计决策。这也意味着，可以比较方便地加入一些设计反思，这为创建容易扩展的代码提供了更多的灵活性和动力。

### 3.4.3 使用动态类型需要自律

迄今为止，见识到利用动态类型能使代码多么简单、优雅和灵活了吧？但这其中是否存在风险呢？

3

- 在创建一个**helper**时，可能会敲错方法的名字。
- 没有类型信息，怎么知道给方法发什么呢？
- 如果把方法发给一个不能提供帮助的事物（一个不能搬动物体的对象），又会怎么样呢？

这些担心都是合理的，但是不必恐慌。这一节，我们来看看解决之道。

在编写代码时经常会出现拼写错误。大脑也经常会愚弄人：往往看到的其实是想看到的，而不是真正摆在那里东西。因此，必须确保方法名的大小写正确，接受的形式参数正确。静态类型语言中的编译器会代为检查。而在动态类型语言中，要么不让编译器做，要么就是编译器本来就不做。这就需要依赖单元测试（参见18.2节）来确保其正确性。仅为这一目的编写单元测试相当古怪。不过，编译器生成的字节码也不一定就完全正确。仍然需要验证代码符合预期目标，不仅要符合输入的代码，还要符合我们的真实意图。

我在编程时是很依赖单元测试的，甚至在静态类型语言中也是如此。缺乏编译器支持（或是缺乏某个编译器）来验证这些内容，并没有令我苦恼。单元测试是很好的方法，而且动态类型需要开发者自律地做单元测试。要知道，使用动态类型语言编程，却没有使用单元测试的自律，就像是在玩火。

在一定程度上，类型可以帮助我们确定需要给一个方法发送什么对象或什么值。但这只是一个方面。在实践中，知道必须给一个方法发送一个**double**类型的值一般是不够的（除非我们想因毁掉了卫星而出名）<sup>①</sup>。自律的单元测试和良好的命名约定可以起到帮助。

最后，再来看一下是否一致的问题：如果发来的是一个不支持预期方法的对象，会怎么样？可以假定调用方负责保证所发送内容的合法性。如果发送了非法的对象，代码会失败，而且会抛给调用方一个异常。即使在编译的代码中，也必须处理前置条件破坏的问题，现在也一样，而且要求更多了。在特殊的情况下，如果想处理一些可供替换的或可选的行为，可以问一下对象，了解它是否能够完成我们预期的功能。Groovy的**respondsTo()**方法可以帮忙（参见11.2节）。假

<sup>①</sup> <http://www.cnn.com/TECH/space/9909/30/mars.metric.02/>

定有一个甘蔗农场，想分给帮助者一些甘蔗，但不是所有的帮助者都会吃甘蔗。我们可以问问帮助者是否喜欢：

```
TypesAndTyping/TakeHelp.groovy
def takeHelpAndReward(helper) {
    //...
    helper.helpMoveThings()

    if (helper.metaClass.respondsTo(helper, 'eatSugarcane')) {
        helper.eatSugarcane()
    }
    //...
}

takeHelpAndReward(new Man())
takeHelpAndReward(new Woman())
takeHelpAndReward(new Elephant())
```

我们询问了帮助者是不是可以接受甘蔗，如果可以，就分给它一些，如下面输出所示：

```
Man's helping
Woman's helping
Elephant's helping
I love sugarcanes...
```

如果使用时加以自律，能力式设计可以创建高度可扩展的、简洁的代码。代码中的强制类型转换和噪音会少一些，类层次结构也会更短。我们开始感觉编译器是在积极地辅助我们，而不是唱反调。

## 3.5 可选类型

Groovy是动态类型的，同时也是可选类型的。这意味着，既可以将类型的转盘拨到一个极端——不指定任何类型，让Groovy确定；也可以将它拨到另一个极端，精确地指定所要使用的变量或引用的类型。

请记住，Groovy是一门运行在JVM上的语言。可选类型有助于集成Groovy代码和Java的库、框架以及工具。有时候，Groovy的动态类型映射与当前使用的库、框架或工具并不匹配。这种情况在Groovy中并不突出——开发者可以轻松地切换类型模式，指明类型信息。可选类型在其他情况下也是有用的，比如有时候需要类型信息来生成数据库模式，或者创建GORM/Grails中的验证器。

试想，正在使用Groovy编写一个JUnit测试（参见18.2节）。在定义方法时，可以使用**def**关键字来说明返回类型为**Object**。因为JUnit要求测试方法为**void**，如果尝试运行使用**def**定义的测

试，就会遇到一个错误。相反，必须把方法定义为void，以满足JUnit。这里Groovy的可选类型就派上用场了。

回看一下图3-3，你可能会疑惑，如果Groovy是可选类型的，那为什么不放在静态类型和动态类型中间呢？这是因为，Groovy编译器（groovyc）不会进行完整的类型检查（具体细节参见2.11.2节）。如果我们写下X obj = 2，其中X是一个类，groovyc会简单地放一个强制类型转换操作，如X obj = (X) 2，让运行时动态确定这条语句是否合法。因此，即便Groovy允许使用类型，它仍然是动态类型的。

## 3.6 多方法

3

动态类型和动态类型语言改变了对象响应方法调用的方式。

Groovy支持多态，就像Java那样，但是它比基于目标对象的类型简单地分派方法走得更远。来看一下Java中的多态：

### TypesAndTyping/Employee.java

```
// Java代码
public class Employee {
    public void raise(Number amount) {
        System.out.println("Employee got raise");
    }
}
```

Employee类中的raise()方法仅仅报告了一下它被调用了。现在看一下Executive类：

### TypesAndTyping/Executive.java

```
// Java代码
public class Executive extends Employee {
    public void raise(Number amount) {
        System.out.println("Executive got raise");
    }

    public void raise(java.math.BigDecimal amount) {
        System.out.println("Executive got outlandish raise");
    }
}
```

Executive覆盖了Employee中的raise(Number amount)方法，它会报告Executive中的这个方法被调用了。同时提供了一个重载的版本——raise(java.math.BigDecimal amount)，它会打印获得了不同寻常的加薪。你预计下面的代码会调用哪个版本呢？

最后来看一下使用这些类的Java代码：

**TypesAndTyping/GiveRaiseJava.java**

```
// Java代码
import java.math.BigDecimal;
public class GiveRaiseJava {
    public static void giveRaise(Employee employee) {
        employee.raise(new BigDecimal(10000.00));
    }

    public static void main(String[] args) {
        giveRaise(new Employee());
        giveRaise(new Executive());
    }
}
```

代码中创建了一个Employee和一个Executive，调用了同样的giveRaise()方法，而该方法又会调用这些对象上的raise()方法。输出正如我们的预期：

```
Employee got raise
Executive got raise
```

Employee中的raise()方法是多态的，这意味着在运行时，被调用的方法依赖的并不是目标的引用类型，而是所引用对象的实际类型。不过这里有个限制。在运行时调用的方法必须接收Number作为一个参数，因为这是由基类Employee定义的。因此，编译器会把BigDecimal的实例看作Number。

这是一个标准的、日常使用的Java操作。没什么大不了的，对吧？但是当涉及Groovy的动态特性时，一切都不同了。Groovy深知Tony Hoare的那句名言：“过早的优化是万恶之源。”<sup>①</sup>

当在Groovy中调用raise()方法时，它不会像Java中的代码那样按前面的顺序走。相反，形象地说，它会走到一个对象跟前，问一下：“嘿，咱是不是有一个接收java.math.BigDecimal的raise()方法啊？”一个Employee对象会说，“没有，不过我可以接收一个Number。”而另一方面，一个Executive对象确实有一个接收BigDecimal的raise()方法，所以调用会被路由到这个对象。下面代码说明了这一行为，我们仍然使用前面Java定义的Employee类和Executive类，它们没有变化：

**TypesAndTyping/GiveRaise.groovy**

```
void giveRaise(Employee employee) {
    employee.raise(new BigDecimal(10000.00))
    // 和下面这条语句效果相同
    //employee.raise(10000.00)
}
```

<sup>①</sup> 这里的“过早的优化”指的是提前确定调用的版本，在GiveRaiseJava.java这个例子中，raise(java.math.BigDecimal amount)方法在重载解析时就被排除掉了，而Groovy则直到最后调用时才根据目标对象和所提供的参数确定实际要调用的方法版本。——译者注

```
giveRaise new Employee()
giveRaise new Executive()
```

Groovy报告的输出和Java不同：

```
Employee got raise
Executive got outlandish raise
```

如果一个类中有重载的方法，Groovy会聪明地选择正确的实现——不仅基于目标对象（调用方法的对象），还基于所提供的参数。因为方法分派基于多个实体——目标加参数，所以这被称作多分派或多方法（Multimethods）。

因为多方法机制，Groovy没有遭受Java的类型混淆问题之苦，感谢Neal Ford提供的这个Java示例。看一下下面使用了泛型的Java代码。`lst`引用的是一个`ArrayList<String>`实例，而`Collection<String>`类型的`col`引用的是同一实例。我们向`lst`中加入3个元素，然后移除1个。移除操作去掉了列表中的第一个元素。现在我们想调用`col.remove(0)`来移除另一个元素。然而，`Collection`接口的`remove()`方法想接收的是一个`Object`，所以Java把`0`装箱成一个`Integer`。因为这个`Integer`实例不是列表中的元素，所以这个方法调用没有移除掉任何东西。

3

#### TypesAndTyping/UsingCollection.java

```
//Java代码
import java.util.*;

public class UsingCollection {
    public static void main(String[] args) {
        ArrayList<String> lst = new ArrayList<String>();
        Collection<String> col = lst;

        lst.add("one");
        lst.add("two");
        lst.add("three");
        lst.remove(0);
        col.remove(0);

        System.out.println("Added three items, removed two, so 1 item to remain.");
        System.out.println("Number of elements is: " + lst.size());
        System.out.println("Number of elements is: " + col.size());
    }
}
```

输出显示出这段代码令人不爽的行为：

```
Added three items, removed two, so 1 item to remain.
Number of elements is: 2
Number of elements is: 2
```

再来看一下这段代码在Groovy中的表现。不需要对前面的代码做任何改动，只是简单地将其复制粘贴到一个名为`UsingCollection.groovy`文件中。然后运行`groovy UsingCollection`并

观察输出。可以看到，其输出与Java版本不同：

```
Added three items, removed two, so 1 item to remain.  
Number of elements is: 1  
Number of elements is: 1
```

Groovy的动态与多方法能力很好地处理了这种情况。在运行时，Groovy知道我们想去掉第一个元素，而不会招惹装箱操作这种不必要的麻烦，也就不会出现前面那种不正确的行为。

## 3.7 动态还是非动态

鉴于Groovy是一门支持可选类型的动态类型语言，那我们是应该指明类型，还是依赖动态类型呢？这方面其实没有真正的规则，但是我们当然可以养成一些偏好。

在使用Groovy编程时，我倾向于省略类型，不过我会为形参或变量选择表达性好的名字。这是因为，不指明类型，一来可以享受到鸭子类型的优点（参见3.4节），二来使应用模拟进行测试变简单了（参见18.2节）。

当然在必要的情况下，我也会选择指明类型。比如，JUnit要求测试方法为void类型，或者指明类型信息有很大的好处，比如在Grails对象关系映射（GORM）中把类型映射到数据库。

如果要为使用静态类型语言的程序员开发API，那我们会在静态类型的、面向客户的API中指明方法形参的类型。

从使用的角度看，社区倾向于总是指明方法签名中的类型。其优势是，在方法调用时知道实参的类型，同时避免了方法内不必要的运行时类型检查。

## 3.8 关闭动态类型

本书所涉及的所有元编程能力，都依赖Groovy的动态类型，但动态类型是有代价的。原本在编译时可以发现的错误被推迟到了运行时。此外，动态方法分派机制也有开销。尽管Java 7为缓解此性能问题而引入了动态调用功能，但是当Groovy在老版本的JVM上运行时，仍然存在性能影响。

可以让Groovy编译器将其类型检查从动态的不严格模式收紧到我们对静态类型编译器（如javac）所期待的水平。还可以权衡动态类型和元编程能力的收益，让Groovy编译器静态编译代码，以便得到更高效的字节码。

本节将介绍两个功能，一个是让Groovy在编译时执行更严格的检查，另一个是让它创建更高效的静态编译的字节码。

### 3.8.1 静态类型检查

对于编译时不存在的方法和属性，基于它们会在运行时被注入应用中的假设，我们可以使用Groovy的动态特性来调用和访问。一方面，这可以十分灵活地为应用提供高级功能，本书第三部分将予以介绍；而另一方面，愚蠢的拼写错误可能会蒙混过关，在运行时却会导致失败。当然，这些错误在我们的单元测试中很快就会浮出水面，然而如果程序中没有使用这样的动态能力，这就是不必要的负担。

其实，完全可以让Groovy自己识别正确的类型，并确保调用的方法和访问的属性在该类型上是合法的。可以使用特殊的注解@TypeChecked，让Groovy去检查这些种错误，这个注解可以用于类或单个方法上。如果用于一个类，则类型检查会在该类中所有的方法、闭包和内部类上执行。如果用于一个方法，则类型检查仅在目标方法的成员上执行。

通过例子来试用一下这个注解。首先创建一个方法，它有一个隐匿的错误，而且没有编译时保护。

#### TypesAndTyping/NoCompiletimeCheck.groovy

```
def shout(String str) {
    println "Printing in uppercase"
    println str.toUpperCase()
    println "Printing again in uppercase"
    println str.toUppercase()
}
try {
    shout('hello')
} catch(ex) {
    println "Failed..."
}
```

`shout()`方法接收一个`String`类型的形参，并调用`toUpperCase()`方法。在第二次调用中，有一个拼写错误。Groovy将在运行时报告一个错误。

```
Printing in uppercase
HELLO
Printing again in uppercase
Failed...
```

这段代码没有使用任何元编程，因此可以利用编译时验证的优势。下面把`@TypeChecked`注解加到这个方法上。

```
@groovy.transform.TypeChecked
def shout(String str) {
//...
```

一旦Groovy看到这个注解，它就会在目标代码上执行严格的检查。如果运行这个版本的代码，Groovy不会像上个版本一样走那么远；编译时就会出现错误。

```

Static type checking] - Cannot find matching method java.lang.String#toUpperCase().
Please check if the declared type is right and if the method exists.
@ line 10, column 11.
    println str.toUpperCase()
               ^
1 error

```

对于使用@TypeChecked注解标记的代码，在编译时，编译器会验证方法或属性是否从属于该类。这会阻止我们使用任何元编程能力。例如，Groovy中默认可以向类中注入方法：

#### TypesAndTyping/Inject.groovy

```

def shoutString(String str) {
    println str.shout()
}

str = 'hello'
str.metaClass.shout = {-> toUpperCase() }
shoutString(str)

```

动态地向String类的实例添加了shout()方法，而且能够从shoutString()方法中调用它，在输出中可以看到：

```
HELLO
```

如果使用@TypeChecked注解shoutString()方法，编译器会阻止我们做进一步的处理。

```

@groovy.transform.TypeChecked
def shoutString(String str) {
    println str.shout() //编译时出错
}

```

虽然当静态类型检查生效时，不能直接调用动态方法，但是有一个变通方案。可以在Groovy对象上使用一个特殊的方法——invokeMethod()，10.6节将会讨论。

静态类型检查会限制使用动态方法。然而，它并没有阻止使用Groovy向JDK中的类添加的方法（参见第7章）。静态类型检查器会检查这些类中的方法和属性。它还会检查一个特殊的DefaultGroovyMethods类，其中包含了一些有用的、优雅的扩展方法。此外，它还会检查开发者能够添加的定制扩展，7.3节将予以讨论。例如，可以自由地在String上调用Groovy添加的reverse()方法。

#### TypesAndTyping/Reverse.groovy

```

@groovy.transform.TypeChecked
def printInReverse(String str) {
    println str.reverse() //没问题
}
printInReverse 'hello'

```

要利用静态类型检查，必须要指明方法和闭包的形参类型。能够在形参上调用的方法，被限制为该类型在编译时已知支持的方法。Groovy会推断闭包的返回类型，并相应地执行类型检查，所以不必担心此类细节。

与Java相比，Groovy的类型检查有一个优势。如果使用`instanceOf`检查类型，在使用该类型特定的方法或属性时，并不需要执行强制转换，下面的例子说明了这一点。

#### TypesAndTyping/NoCast.groovy

```
@groovy.transform.TypeChecked
def use(Object instance) {
    if(instance instanceof String)
        println instance.length() //不必强制转换
    else
        println instance
}
use('hello')
use(4)
```

3

我们介绍了如何让Groovy在编译时执行类型检查。如果注解的是一个完整的类，以进行静态类型检查，我们还可以使用`SKIP`参数去掉一些具体的方法，不对它们进行静态类型检查：

#### TypesAndTyping/Optout.groovy

```
import groovy.transform.TypeChecked
import groovy.transform.TypeCheckingMode

@TypeChecked
class Sample {
    //此处静态类型检查生效
    def method1() {
    }

    @TypeChecked(TypeCheckingMode.SKIP)
    def method2(String str) {
        str.shout()
    }
}
```

我们把`method2()`内的代码从编译时检查中去掉了，因此，除它之外，整个类都会执行静态类型检查。

静态类型检查旨在帮助在编译时识别出一些错误。如果没有错误，编译器为类型检查版本和无类型检查版本生成的字节码是类似的。如果想生成高效的字节码，则必须使用下一节要介绍的静态编译。

### 3.8.2 静态编译

Groovy元编程和动态类型的优点显而易见，但是这些优点需要以性能为代价。性能的下降与代码、所调用方法的个数等因素相关。当不需要元编程和动态能力时，与等价的Java代码相比，性能损失可能高达10%。Java 7的InvokeDynamic特性就旨在缓解这种痛苦，但是对于使用老版本Java的人而言，静态编译可能是个有用特性。

我们可以关闭动态类型，阻止元编程，放弃多方法，并让Groovy生成性能足以与Java媲美的、高效的字节码。

可以使用@CompileStatic注解让Groovy执行静态编译。这样为目标代码生成的字节码会和javac生成的字节码很像。例如，不使用该注解，先来编译一下示例代码。

#### TypesAndTyping/NoStaticCompile.groovy

```
def shout1(String str) {
    println str.toUpperCase()
}
```

如果使用groovyc编译前面代码，然后执行javac -p NoStaticCompile，我们会发现，对toUpperCase()方法的调用是通过CallSite()进行的，它会处理Groovy的动态调用机制。

```
...
14: invokeinterface #57, 2; //InterfaceMethod
org/codehaus/groovy/runtime/callsite/CallSite.call:...
19: invokeinterface #61, 3; //InterfaceMethod
org/codehaus/groovy/runtime/callsite/CallSite.callCurrent:...
...
```

再使用@CompileStatic注解标记该方法。

#### TypesAndTyping/StaticCompile.groovy

```
@groovy.transform.CompileStatic
def shout1(String str) {
    println str.toUpperCase()
}
```

现在编译器生成了一个invokevirtual调用，和Java编译器所做的一样。

```
...
2: invokevirtual      #63; //Method java/lang/String.toUpperCase:()...
5: invokevirtual      #67; //Method groovy/lang/Script.println:...
...
```

如果想获得性能可以与Java媲美的代码，静态编译是很好的选择。而对于性能没那么关键，或者想使用元编程的代码，则不用考虑静态编译。

这一章介绍了类型相关的问题、优点以及Groovy的特性。当不愿意指明类型时，Groovy的动态类型如何让类型成为隐式的，以及当需要时，可以很方便地使用可选类型实现类型声明。另外，

Groovy中的方法分派大不一样，而且非常强大，还介绍了如何享用真正的多态，以及如何利用能力式设计。最后，我们还看到了在Groovy中如何选择性地关闭动态类型，以及在希望更多编译器检查或更好的性能的地方，如何获得静态类型的优势。下一章将介绍Groovy最有趣的特性之一——闭包。

## 第4章

# 使用闭包

4

当在Java中定义用于注册事件处理器的方法参数或创建短小的胶水代码时，会创建匿名内部类。该特性在Java 1.1中引入时，看上去像个不错的想法，但是没过多久，人们就意识到，它们变得非常冗长，尤其是对于那种确实非常短的单方法接口的实现而言。Groovy中的闭包就是去掉了那种冗长感的短小的匿名方法。

闭包是轻量级的，短小、简洁，而且将会是我们在Groovy中使用最多的特性之一。过去传递匿名类实例的地方，现在可以传递闭包。

闭包是从函数式编程的Lambda ( $\lambda$ ) 表达式派生而来的。根据Robert Sebesta的*Concepts of Programming Languages* [Seb04]一书，“一个Lambda表达式指定了一个函数的参数与映射”。闭包是Groovy最强大的特性之一，而且语法上非常优雅。或者如计算机科学家和函数式编程先驱Peter J. Landin所言：“（闭包是）可以帮你消化 $\lambda$ 演算的一点语法糖。”

我们会通过Groovy JDK (GDK) 大量使用闭包，因为GDK使用一些以闭包为参数的灵活且便捷的方法扩展了JDK。与其被迫创建接口和很多小型类，不如使用没那么多繁文缛节的小代码块来设计应用，这意味着代码减少了，杂乱冗余也减少了，而复用变多了。

本章将介绍闭包的创建和使用。我们将介绍如何使用闭包来优雅地实现某些设计模式。你还会了解到，闭包不是简单地替代匿名方法，它还可以变成解决有较高内存需求的问题的一种通用工具。所以这就开始了解和学习闭包吧。

## 4.1 闭包的便利性

Groovy中的闭包完全避免了代码的冗长，而且可以辅助创建轻量级、可复用的代码片段。通过对对比闭包与我们所熟悉的传统解决方案在解决同样任务时的表现，就可以理解这种便利性。

### 4.1.1 传统方式

举个简单的例子：求1到某个特定的数n之间所有偶数的和。

下面是传统方式：

```
UsingClosures/UsingEvenNumbers.groovy
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()方法中运行了一个for循环，在偶数上迭代并求和。现在假设我们不想求和了，而是要计算从1到n之间所有偶数的积。代码如下：

```
UsingClosures/UsingEvenNumbers.groovy
4
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)}"
```

又在偶数上进行了迭代，这次计算的是它们的积。现在，假如想得到由这些偶数值的平方所组成的集合，又该怎么办呢？返回平方值所组成数组的代码可能会写成下面这样：

```
UsingClosures/UsingEvenNumbers.groovy
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)}"
```

在前面的几个代码示例中，进行循环的代码是相同的（也是重复的）。差别在于处理的是和、积还是平方。如果想在偶数上执行一些其他操作，我们还要重复这些遍历数字的代码。我们得想办法去掉这种重复。

### 4.1.2 Groovy 方式

以上三个例子产生的是不同的结果，但它们都有一个共同的任务，那就是从给定集合中挑选

出偶数。我们就从解决这一共同任务的一个函数入手。这里不是返回一个偶数的列表，而是这样：当挑选出一个偶数时，直接将其发给一个代码块来处理。现在，可以让代码块简单地打印传入的数字：

```
UsingClosures/PickEven.groovy
def pickEven(n, block) {
    for(int i = 2; i <= n; i += 2) {
        block(i)
    }
}

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

`pickEven()`方法是一个高阶函数，即以函数为参数，或返回一个函数作为结果的函数<sup>①</sup>。该方法对值进行迭代（和前面一样），但不同的是它将值发送给了一个代码块。在Groovy中，我们称这种匿名代码块为闭包（Closure），Groovy设计团队使用了该术语的一个不太严格的定义<sup>②</sup>。

变量`block`保存了一个指向闭包的引用。可以像传递对象一样传递闭包。变量名没必要一定命名为`block`，可以使用任何合法的变量名。当调用`pickEven()`方法时，现在可以像前面代码中演示的那样向其发送代码块。代码块（{}内的代码）被传给形参`block`，就像把值10传给变量`n`。在Groovy中，想传递多少闭包就可以传递多少。例如，方法调用的第一个、第三个和最后一个实参都可以是闭包。如果闭包是最后一个实参，可以用下面这种优雅的语法：

```
UsingClosures/PickEven.groovy
pickEven(10) { println it }
```

当闭包是方法调用的最后一个实参时，可以将闭包附到方法调用上。在这种情况下，代码块看上去就像是附在方法调用上的寄生虫。不同于Java代码块，Groovy的闭包不能单独存在，只能附到一个方法上，或是赋值给一个变量。

代码块中的`it`是什么呢？如果只向代码块中传递一个参数，那么可以使用`it`这个特殊的变量名来指代该参数。如果你喜欢，也可以像下面的例子这样，用其他名字代替`it`：

```
UsingClosures/PickEven.groovy
pickEven(10) { evenNumber -> println evenNumber }
```

变量`evenNumber`现在指代的是在`pickEven()`方法内传递给该闭包的实参。

回到关于偶数的那些计算问题。我们可以使用`pickEven()`来求和，像这样：

<sup>①</sup> 参见<http://c2.com/cgi/wiki?HigherOrderFunction>。

<sup>②</sup> 参见<http://groovy.codehaus.org/Closures+-+Formal+Definition>。

**UsingClosures/PickEven.groovy**

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

我们是从简单打印`pickEven()`生成的偶数值入手的，但是并未对函数做任何修改，就可以求和了。不像传统的方式那样重复地写一段代码，这里的代码很简洁，而且复用性更好。该函数并不限于计算这些值的和，可以复用作其他用途，比如计算积，就像下面的代码这样：

**UsingClosures/PickEven.groovy**

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

除了语法上的优雅，闭包还为函数将部分实现逻辑委托出去提供了一种简单、方便的方式。

前面示例中的代码块所做的事情，要比我们更早之前看到的代码块多。它将触角伸到了`pickEven()`的调用者的作用域之内，使用了变量`product`。这是闭包的一个有趣特性。闭包是一个函数，这里变量都绑定到了一个上下文或环境中，这个函数就在其中执行。

知道了如何创建闭包，下一步将探讨如何在应用中使用闭包。

## 4.2 闭包的应用

本章会一直谈论闭包的强大与优雅，不过首先还是先探讨一下如何将闭包应用于我们的项目之中。在面对某个特定的功能或任务时，需要决定是以普通的函数或方法来实现，还是使用闭包。

闭包能够扩充、优化或增强另一段代码。比如，可以选择对象的操作通过一个谓词或条件提炼出来，而闭包对于表达这样的谓词或条件可能很有用。另外，也可以通过闭包来使用协程(Coroutine)，实现诸如迭代器或循环中的控制流转移。

闭包有两个非常擅长的具体领域：一个是辅助资源清理（参见4.5节），另一个是辅助创建内部的领域特定语言(DSL，详情参见第19章）。

普通函数在实现某个目标明确的任务时优于闭包。重构的过程是引入闭包的好时机。

一旦代码工作起来，就可以重新审视这些代码，看看闭包能否对其加以改进，使之更为优雅。要让闭包在这样的过程中浮现出来，而不是一开始就强制使用。

闭包应该保持短小，有内聚性。闭包应该设计为附到方法调用上的小段代码，只有几行。当编写使用闭包的方法时，最好不要滥用闭包的动态属性，比如在运行时确定参数的数目和类型。在调用方法时实现的闭包一定要非常简单，而且做到显而易见。

看过了使用闭包的便利性与优势，下一节来看一下闭包的不同使用方式。

## 4.3 闭包的使用方式

前面介绍了如何在定义方法调用的参数时即时创建闭包。此外，还可以将闭包赋值给变量并复用，现在就来试试看。

在下面的示例中，`totalSelectValues()`接受一个闭包，用于帮助确定要将哪些值用于计算中：

```
UsingClosures/Strategy.groovy
def totalSelectValues(n, closure) {
    total = 0
    for(i in 1..n) {
        if (closure(i)) { total += i }
    }
    total
}
print "Total of even numbers from 1 to 10 is "
println totalSelectValues(10) { it % 2 == 0 }

def isOdd = { it % 2 != 0 }
print "Total of odd numbers from 1 to 10 is "
println totalSelectValues(10, isOdd)
```

`totalSelectValues()`方法从1迭代到`n`，它会对每个值调用闭包，以确定该值是否要用于计算中，它将选择过程委托给了该闭包。

即便在闭包中，`return`也是可选的。如果没有显式的`return`，最后一个表达式的值（可能是`null`）会自动返回给调用者。

在第一次调用`totalSelectValues()`时，将闭包内联到了方法调用中，该闭包仅选择偶数。另一方面，预先定义了要传给第二个调用的闭包。这个通过变量`isOdd`引用的闭包仅选择奇数。与调用时直接创建的闭包不同，这种预先定义的闭包可以在多个调用中复用。顺便插一句，不费吹灰之力，这个例子就实现了策略模式。<sup>①</sup>

在调用时即时创建闭包之后，我们又了解了预先定义闭包。将闭包缓存下来，以备将来之用，这种方法很有用，下面就会看到。

假设我们正在创建一个模拟器，使用它可以为设备插入不同的计算。我们想执行一些计算，但是希望使用恰当的计算程序。下面是实现该功能的一个例子：

```
UsingClosures/Simulate.groovy
class Equipment {
    def calculator
```

<sup>①</sup> 关于该模式的具体细节，请参考*Design Patterns: Elements of Reusable Object-Oriented Software* [GHJV95]一书。

```

Equipment(calc) { calculator = calc }
def simulate() {
    println "Running simulation"
    calculator() // 可能还会发送参数
}
}
def eq1 = new Equipment({ println "Calculator 1" })
def aCalculator = { println "Calculator 2" }
def eq2 = new Equipment(aCalculator)
def eq3 = new Equipment(aCalculator)

eq1.simulate()
eq2.simulate()
eq3.simulate()

```

`Equipment`的构造器接收闭包作为一个参数，并将其缓存到`calculator`属性中。`simulate()`方法调用该闭包来执行计算。在创建`Equipment`的实例`eq1`时，我们通过一个内联的闭包为其提供了一个计算程序（关于这种语法限制，参见2.11.5节）。如果需要复用该代码块，又该如何呢？可以将闭包保存到一个变量中，就像前面代码中的`aCalculator`，在创建`Equipment`的另外两个实例（即`eq2`和`eq3`）时就使用了它。输出说明设备使用了缓存的计算程序：

```

Running simulation
Calculator 1
Running simulation
Calculator 2
Running simulation
Calculator 2

```

`Collection`相关的类大量使用了闭包，要找使用闭包的例子，这是个好地方。具体细节可以参考6.2节。

介绍完如何创建和复用闭包，下一节来看一下如何向闭包传递参数。

## 4.4 向闭包传递参数

前面几节介绍了如何定义和使用闭包，这一节将探讨如何向闭包发送多个参数。

对于单参数的闭包，`it`是该参数的默认名称。只要知道只传入一个参数，就可以使用`it`。如果传入的参数多于一个，就需要通过名字一一列出了，如下面这个例子：

### UsingClosures/ClosureWithTwoParameters.groovy

```

def tellFortune(closure) {
    closure new Date("09/20/2012"), "Your day is filled with ceremony"
}
tellFortune() { date, fortune ->
    println "Fortune for ${date} is '${fortune}'"
}

```

在调用闭包closure时，`tellFortune()`方法提供了两个参数：一个`Date`实例，一个表示运势信息的`String`。该闭包分别用`name`和`fortune`引用它们。符号`->`将闭包的参数声明与闭包主体分隔开来。运行这段代码，看一下输出<sup>①</sup>：

```
Fortune for Thu Sep 20 00:00:00 MST 2012 is 'Your day is filled with ceremony'
```

因为Groovy支持可选的类型，所以可以在闭包中定义参数的类型，如下面例子所示：

```
UsingClosures/ClosureWithTwoParameters.groovy
tellFortune() { Date date, fortune ->
    println "Fortune for ${date} is '${fortune}'"
}
```

如果为参数选择了表现力好的名字，通常可以避免定义类型。后面会看到，在元编程中，我们可以使用闭包来覆盖或替代方法，而在那种情况下，类型信息对于确保实现的正确性非常重要。

## 4.5 使用闭包进行资源清理

Java的自动垃圾收集是把双刃剑。只要我们释放了引用，就不用担心资源回收问题。但是资源何时会被自动清理并没有保证，因为这要任凭垃圾收集器处理。某些情况下，我们可能希望清理直接进行。这就是我们会在资源密集型的类中看到`close()`和`destroy()`这样的方法的原因。

不过仍然存在一个问题，使用这些类的人可能会忘记调用这些方法。闭包可以帮助确保这些方法被调用。下列代码将创建一个`FileWriter`，并写入一些数据，但是没有在它上面调用`close()`。如果运行这段代码，文件`output.txt`中并没有我们所输入的数据或字符。

```
UsingClosures/FileClose.groovy
writer = new FileWriter('output.txt')
writer.write('!')
// 忘记调用writer.close()
```

使用Groovy添加的`withWriter()`方法重写这段代码。当从闭包返回时，`withWriter()`会自动刷新（`flush`）并关闭这个流。

```
UsingClosures/FileClose.groovy
new FileWriter('output.txt').withWriter { writer ->
    writer.write('a')
} // 不再需要自己调用close()
```

现在不必担心流的关闭了；我们可以集中精力完成工作。也可以在自己的类中实现这样的便捷方法，从而使类的使用者能够开开心心且卓有成效。例如，我们有一个`Resource`类，希望使

<sup>①</sup> 具体输出情况和计算机的时区设置有关，所以读者运行这段代码的结果未必和书上的输出完全一致。——译者注

用者在调用它的任何实例方法之前先调用`open()`，使用完成时再调用`close()`。

这是`Resource`类的一个例子：

#### UsingClosures/ResourceCleanup.groovy

```
class Resource {
    def open() { print "opened..." }
    def close() { print "closed" }
    def read() { print "read..." }
    def write() { print "write..." }
    //...
}
```

下面是一个使用该类的例子：

#### UsingClosures/ResourceCleanup.groovy

```
def resource = new Resource()
resource.open()
resource.read()
resource.write()
```

4

遗憾的是，类的使用者忘记了调用`close()`，资源没有关闭。从下面的输出中可以看到：

```
opened...read...write...
```

这里闭包可以帮得上忙，可以使用Execute Around Method模式（见后文说明）来处理这个问题。

创建一个名为`use()`的静态方法：

#### UsingClosures/ResourceCleanup.groovy

```
def static use(closure) {
    def r = new Resource()
    try {
        r.open()
        closure(r)
    } finally {
        r.close()
    }
}
```

这个静态方法中创建了一个`Resource`实例，然后在该实例上调用`open()`，调用闭包，最后调用`close()`。这里还使用`try-finally`对调用加以保护，所以即使调用闭包时抛出异常，也会正常调用`close()`。

### Execute Around Method模式

如果有一对必须连续执行的动作，比如打开和关闭，我们就可以使用Execute Around Method模式，这是一个Smalltalk模式，Kent Beck的*Smalltalk Best Practice Patterns* [Bec96]一书中曾经讨论过。编写一个Execute Around方法，它接收一个块作为参数。在这个方法中，把对该块的调用夹到对那对方法的调用之间。即先调用第一个方法，然后调用该块，最后调用第二个方法。方法的使用者不必担心这对动作，它们会自动被调用。我们甚至可以在Execute Around方法内处理异常。

来看一下类的使用者如何使用它：

#### UsingClosures/ResourceCleanup.groovy

```
Resource.use { res ->
    res.read()
    res.write()
}
```

下面是调用use()方法的输出，闭包会自动执行：

```
opened...read...write...closed
```

多亏闭包，现在close()的调用是自动的、确定性的，而且会在恰当的时机调用。我们可以将注意力集中于应用领域及其内在复杂性上，让类库去处理诸如确保文件I/O的清理等系统级任务。

学习完如何创建闭包，以及如何将其传递给函数和类，下一节将介绍函数和闭包如何交互。

## 4.6 闭包与协程

调用一个函数或方法会在程序的执行序列中创建一个新的作用域。我们会在一个入口点（方法最上面的语句）进入函数。在方法完成之后，回到调用者的作用域。

协程（Coroutine）则支持多个入口点，每个入口点都是上次挂起调用的位置。我们可以进入一个函数，执行部分代码，挂起，再回到调用者的上下文或作用域内执行一些代码。之后我们可以在挂起的地方恢复该函数的执行。正如Donald E. Knuth所言，“与主例程和子例程之间的不对称关系不同，协程之间是完全对称的，可以互相调用。<sup>①</sup>”

协程对于实现某些特殊的逻辑或算法非常方便，比如用在生产者—消费者问题中。生产者会接收一些输入，对输入做一些初始处理，通知消费者拿走处理过的值做进一步计算，并输出或存储结果。消费者处理它的那部分工作，完成之后通知生产者以获取更多输入。

<sup>①</sup> *The Art of Computer Programming: Fundamental Algorithms* [Knu97]

在Java中, `wait()`和`notify()`与多线程结合使用, 可以协助实现协程。闭包会让人产生“协程是在一个线程中执行”的感觉(或者说错觉)。

例如, 看一下这段代码:

```
UsingClosures/Coroutine.groovy
def iterate(n, closure) {
    1.upto(n) {
        println "In iterate with value ${it}"
        closure(it)
    }
}

println "Calling iterate"
total = 0
iterate(4) {
    total += it
    println "In closure total so far is ${total}"
}
println "Done"
```

4

在这段代码中, 控制流在`iterate()`方法和闭包中来回转移:

```
Calling iterate
In iterate with value 1
In closure total so far is 1
In iterate with value 2
In closure total so far is 3
In iterate with value 3
In closure total so far is 6
In iterate with value 4
In closure total so far is 10
Done
```

每次调用闭包, 我们都会从上一次调用中恢复`total`的值。执行序列如图4-1所示, 我们在两个函数的上下文中来回切换。

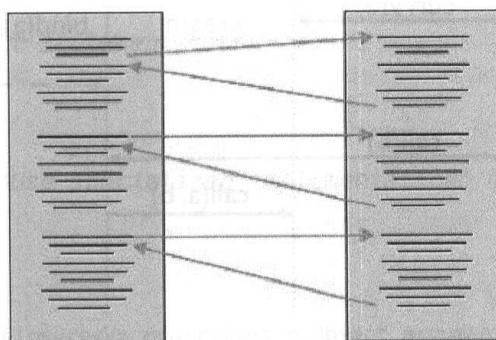


图4-1 一个协程的执行序列

介绍了函数与闭包如何交互, 下一节将介绍如何改变闭包以及如何转换闭包的形参。

## 4.7 科里化闭包

闭包可能不接受任何形参，也可能接受多个形参。每次调用一个闭包时，它会期望我们为其每一个形参传入相应的实参。然而，如果在多次调用之间，有一个或多个实参是相同的，传参就会变得枯燥乏味。预先绑定一些闭包形参可以缓解这种痛苦。

带有预绑定形参的闭包叫做科里化闭包（Curried Closure）。虽然英文单词curry有“咖喱”之意，但科里化闭包与我最喜爱的印度菜并没有什么关系。（术语“科里化”源自对Lambda演算作出重要贡献的著名数学家Haskell B. Curry的名字，Christopher Strachey、Moses Schönfinkel和Friedrich Ludwig创造了这一术语。具体概念则是由Gottlob Frege发明的。）当对一个闭包调用curry()时，就是要求预先绑定某些形参。在预先绑定了一个形参之后，调用闭包时就不必重复为这个形参提供实参了。如图4-2所示，方法调用现在可以接受较少的参数。这有助于去掉方法调用中的冗余或重复，从下面的例子可以看出。

### UsingClosures/Currying.groovy

```
def tellFortunes(closure) {
    Date date = new Date("09/20/2012")
    //closure date, "Your day is filled with ceremony"
    //closure date, "They're features, not bugs"
    // 可以通过科里化避免重复发送date
    postFortune = closure.curry(date)
    postFortune "Your day is filled with ceremony"
    postFortune "They're features, not bugs"
}
tellFortunes() { date, fortune ->
    println "Fortune for ${date} is ${fortune}"
}
```

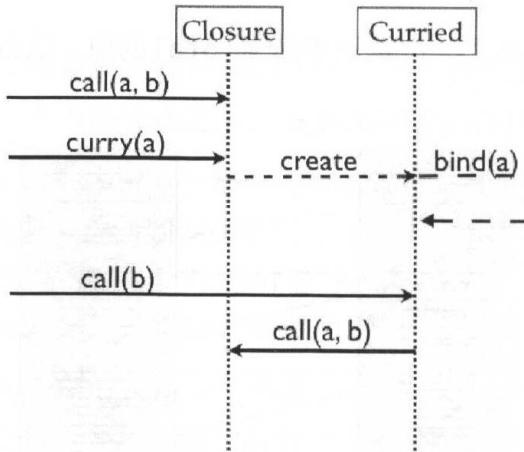


图4-2 对一个闭包进行科里化

`tellFortunes()`方法多次调用了一个闭包。该闭包接受两个形参。因此，每次调用

tellFortunes()时都要提供第一个参数date。作为一种选择，可以以date作为一个参数来调用curry()方法，实现形参date的科里化。postFortune保存着科里化之后的闭包的引用，它已经预先绑定了date的值。

现在可以调用科里化闭包了，只需要传入原来闭包的第二个形参(fortune)。科里化闭包负责把fortune和预先绑定的形参date发送给原来的闭包：

```
Fortune for Thu Sep 20 00:00:00 MST 2012 is 'Your day is filled with ceremony'  
Fortune for Thu Sep 20 00:00:00 MST 2012 is 'They're features, not bugs'
```

可以使用curry()方法科里化任意多个形参，但这些形参必须是从前面开始的连续若干个。也就是说，如果有n个形参，我们可以任意科里化前k个，其中 $0 \leq k \leq n$ 。

如果想科里化后面的形参，可以使用rcurry()方法。如果想科里化位于形参列表中间的形参，可以使用ncurry()<sup>①</sup>方法，传入要进行科里化的形参的位置，同时提供相应的值。

科里化是一种变换，将一个接受多个形参的函数变成了一个接受较少（通常是一个）形参的函数。函数 $f(X, Y) \rightarrow Z$ 上的科里化函数被定义为 $\text{curry}(f): X \rightarrow (Y \rightarrow Z)$ 。科里化有助于简化数学证明方法。就我们的目的而言，在Groovy中，科里化可以减少代码中的噪音。

在形参这个主题之后，我们将学习如何确定一个闭包是否存在，以及如何确定一个闭包可能接收的形参的数目和类型。

## 4.8 动态闭包

可以确定一个闭包是否已经提供。如果尚未提供，比如说是一个算法，我们可以决定使用该算法的一个默认实现来代替调用者未能提供的特殊实现。下面是确定一个闭包是否存在例子：

### UsingClosures/MissingClosure.groovy

```
def doSomething(closure) {  
    if (closure) {  
        closure()  
    } else {  
        println "Using default implementation"  
    }  
}  
  
doSomething() { println "Use specialized implementation" }  
  
doSomething()
```

4

<sup>①</sup> ncurry()有两个版本：public Closure<V> ncurry(int n, Object argument)和public Closure<V> ncurry(int n, Object... arguments)，前者可以科里化位于位置n的形参，后者则可以科里化从位置n开始的多个形参。具体用法可以参考文档：<http://groovy.codehaus.org/api/groovy/lang/Closure.html#ncurry%28int,%20java.lang.Object...%29>。

——译者注

这段代码会确定一个闭包是不是提供了，同时进行相应处理：

```
Use specialized implementation
Using default implementation
```

在传递参数时也有很多灵活性。可以动态地确定一个闭包期望的参数的数目和类型。假设我们使用一个闭包来计算销售税额。税额取决于销售额和税率。再假设该闭包可能需要我们提供税率，也可能不需要。下面是检查参数数目的一个例子：

#### UsingClosures/QueryingClosures.groovy

```
def completeOrder(amount, taxComputer) {
    tax = 0
    if (taxComputer.maximumNumberOfParameters == 2) { // 期望传入税率
        tax = taxComputer(amount, 6.05)
    } else { // 使用默认税率
        tax = taxComputer(amount)
    }
    println "Sales tax is ${tax}"
}
completeOrder(100) { it * 0.0825 }
completeOrder(100) { amount, rate -> amount * (rate/100) }
```

`maximumNumberOfParameters`属性（或`getMaximumNumberOfParameters()`方法）告诉我们给定的闭包接受的参数个数。利用这个方法，我们在`computeOrder()`方法中确定了给定闭包接受的参数个数，并以此确定是否需要提供税率。这可以帮助我们使用正确的参数数目调用给定闭包，在输出中可以看到：

```
Sales tax is 8.2500
Sales tax is 6.0500
```

除了参数个数，还可以使用`parameterTypes`属性（或`getParameterTypes()`方法）获知这些参数的类型。下面这个例子用于检查所提供闭包的参数的类型：

#### UsingClosures/ClosuresParameterTypes.groovy

```
def examine(closure) {
    println "$closure.maximumNumberOfParameters parameter(s) given:"
    for(aParameter in closure.parameterTypes) { println aParameter.name }

    println "--"
}

examine() { }
examine() { it }
examine() { -> }
examine() { val1 -> }
examine() { Date val1 -> }
```

```
examine() {Date val1, val2 -> }
examine() {Date val1, String val2 -> }
```

运行这段代码，看一下参数的个数以及所报告的类型：

```
1 parameter(s) given:
java.lang.Object
--
1 parameter(s) given:
java.lang.Object
--
0 parameter(s) given:
--
1 parameter(s) given:
java.lang.Object
--
1 parameter(s) given:
java.util.Date
--
2 parameter(s) given:
java.util.Date
java.lang.Object
--
2 parameter(s) given:
java.util.Date
java.lang.String
--
```

即便一个闭包没有使用任何形参，就像{}或{ it }中这样，其实它也会接受一个参数（名字默认为it）。如果调用者没有向闭包提供任何值，则第一个形参（it）为null。如果希望闭包完全不接受任何参数，必须使用{->}这种语法：在->之前没有形参，说明该闭包不接受任何参数。

利用maximumNumberOfParameters和parameterTypes属性，我们可以动态地检查闭包，而且在实现某些逻辑时有了更大的灵活性。

谈到检查对象，闭包内的this有什么意义呢？下面我们来看一下。

## 4.9 闭包委托

Groovy中的闭包远远超越了简单的匿名方法，本章余下的几节将关注它们还有哪些强大的功能。Groovy的闭包支持方法委托，而且提供了方法分派能力，这点和JavaScript对原型继承（prototypal inheritance）的支持很像。我们来了解一下其背后的魔法，以及如何好好利用这一点。

this、owner和delegate是闭包的三个属性，用于确定由哪个对象处理该闭包内的方法调用。一般而言，delegate会设置为owner，但是对其进行修改，可以挖掘出Groovy的一些非常好的元编程能力。我们来观察一下闭包的这三个属性：

**UsingClosures/ThisOwnerDelegate.groovy**

```

def examiningClosure(closure) {
    closure()
}

examiningClosure() {
    println "In First Closure:"
    println "class is " + getClass().name
    println "this is " + this + ", super:" + this.getClass().superclass.name
    println "owner is " + owner + ", super:" + owner.getClass().superclass.name
    println "delegate is " + delegate +
        ", super:" + delegate.getClass().superclass.name

    examiningClosure() {
        println "In Closure within the First Closure:"
        println "class is " + getClass().name
        println "this is " + this + ", super:" + this.getClass().superclass.name
        println "owner is " + owner + ", super:" + owner.getClass().superclass.name
        println "delegate is " + delegate +
            ", super:" + delegate.getClass().superclass.name
    }
}

```

在第一个闭包内，取到了该闭包的一些具体信息，确定了this、owner和delegate分别指向的是什么。之后又在第一个闭包内调用了examiningClosure()方法，同时向它传递了另一个闭包。因为第二个闭包是在第一个闭包内定义的，所以第一个闭包成了第二个闭包的owner。在第二个闭包内，我们再次打印了这些具体信息。这段代码的输出如下：

```

In First Closure:
class is ThisOwnerDelegate$_run_closure1
this is ThisOwnerDelegate@55e6cb2a, super:groovy.lang.Script
owner is ThisOwnerDelegate@55e6cb2a, super:groovy.lang.Script
delegate is ThisOwnerDelegate@55e6cb2a, super:groovy.lang.Script
In Closure within the First Closure:
class is ThisOwnerDelegate$_run_closure1_closure2
this is ThisOwnerDelegate@55e6cb2a, super:groovy.lang.Script
owner is ThisOwnerDelegate$_run_closure1@15c330aa, super:groovy.lang.Closure
delegate is ThisOwnerDelegate$_run_closure1@15c330aa, super:groovy.lang.Closure

```

前面的代码示例和相应输出说明，闭包被创建成了内部类。此外还说明，delegate被设置为owner。某些Groovy函数会修改delegate，以执行动态路由，比如with()函数。闭包内的this指向的是该闭包所绑定的对象（正在执行的上下文）。在闭包内引用的变量和方法都会绑定到this，它负责处理任何方法调用，以及对任何属性或变量的访问。如果this无法处理，则转向owner，最后是delegate。下图说明了这种解析顺序。

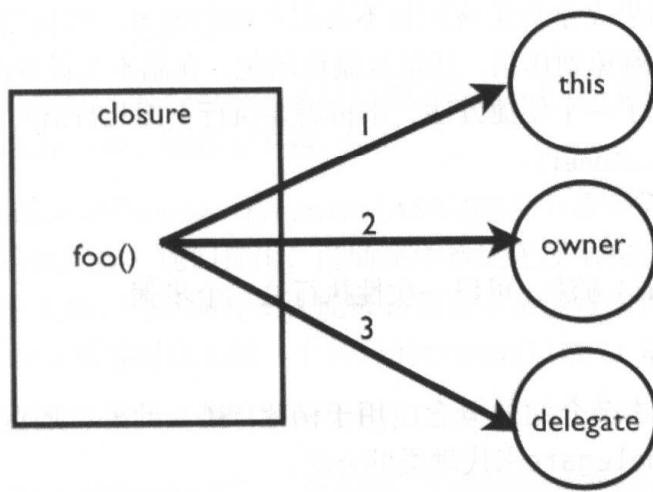


图4-3 闭包内的方法调用解析顺序

下面通过一个例子进一步观察方法解析：

#### UsingClosures/MethodRouting.groovy

```

class Handler {
    def f1() { println "f1 of Handler called ..." }
    def f2() { println "f2 of Handler called ..." }
}

class Example {
    def f1() { println "f1 of Example called ..." }
    def f2() { println "f2 of Example called ..." }

    def foo(closure) {
        closure.delegate = new Handler()
        closure()
    }
}

def f1() { println "f1 of Script called..." }

new Example().foo {
    f1()
    f2()
}

```

在这段代码中，闭包内的方法调用首先被路由到闭包的上下文对象——`this`。如果没找到这些方法，则路由到`delegate`：

```

f1 of Script called...
f2 of Handler called ...

```

前面的示例中设置了闭包的`delegate`属性。不过这会有副作用，尤其是当该闭包还被用于

其他的函数或线程时。如果完全肯定该闭包不会用在别的地方，那自然可以设置`delegate`；如果它用在了其他地方，为避免副作用，还请复制该闭包，在副本上设置`delegate`，并使用副本。Groovy为实现这一点提供了一个便捷方法，不再需要执行下面几行语句：

```
def clone = closure.clone()
clone.delegate = handler
clone()
```

借助一个特殊的`with()`方法，可以一次性执行这三个步骤：

```
handler.with closure
```

19.7节介绍了如何将本节介绍的概念应用于构建DSL。此外，还可以参考7.1节和13.2节。`ExpandoMetaClass`使用`delegate`来代理类的方法。

Groovy的闭包可不仅仅是简单的胶水代码。除了动态方法分派能力，它们还有一些有趣的便捷方法，下一节我们将看到。

## 4.10 使用尾递归编写程序

使用递归会遇到一些较为常见的问题，而借助Groovy中的闭包，我们可以在获得递归之优势的同时避免这些问题。

递归可以通过子问题的解决方案来解决主干问题。递归解决方案的魅力在于非常简洁，而且只需利用输入规模较小的相同问题的解决方案来组合出最终解决方案，这点很酷。尽管存在这些优势，但是程序员往往对递归解决方案敬而远之。在输入规模较大的情况下，潜在的`StackOverflowError`威胁，使得最优秀的程序员都有可能望而却步。

下面是使用一个简单递归实现的极为简化的阶乘函数，我们再熟悉不过了。

### UsingClosures/simpleFactorial.groovy

```
def factorial(BigInteger number) {
    if (number == 1) 1 else number * factorial(number - 1)
}

try {
    println "factorial of 5 is ${factorial(5)}"
    println "Number of bits in the result is ${factorial(5000).bitCount()}"
} catch(Throwable ex) {
    println "Caught ${ex.class.name}"
}
```

5的阶乘是个比较小的值，但5000的阶乘就非常大了。如果计算成功，应该看到的结果是120以及5000的阶乘这个很大的数所包含的位数。当运行程序时，我们发现JVM被数量这么大的递归调用卡住了：

```
factorial of 5 is 120
Caught java.lang.StackOverflowError
```

如果将该函数写为迭代形式，则不会遇到这样的资源限制。但是递归是这么酷，而且富有表现力，要是对资源使用再友好一些，那该多好啊……

*Structure and Interpretation of Computer Programs* [AS96]这本书很不错，书中几位作者论述了一种方法，可以优雅地解决此问题。他们提出，借助编译器优化技术和语言支持，递归程序可以转换为迭代过程。使用这种变换，可以编写出性能极高而且非常优雅的代码，同时还能获得简单迭代的效率优势。Groovy语言通过闭包上的一个特殊的**trampoline()**方法提供了这种技巧。

要使用该特性，首先必须将递归函数实现为闭包：

**UsingClosures/trampolineFactorial.groovy**

4

```
def factorial

factorial = { int number, BigInteger theFactorial ->
    number == 1 ? theFactorial :
        factorial.trampoline(number - 1, number * theFactorial)
}.trampoline()

println "factorial of 5 is ${factorial(5, 1)}"
println "Number of bits in the result is ${factorial(5000, 1).bitCount()}"
```

这里定义了一个名为**factorial**的变量，并将一个闭包赋给它。该闭包接受两个参数：一个是**number**，要计算的就是它的阶乘；一个是**theFactorial**，它表示通过这个递归计算出的部分结果。在闭包中，如果给定的**number**是1，就返回**theFactorial**的值作为结果。否则，就通过调用**trampoline()**方法递归地调用该闭包。将**number - 1**作为第一个参数传给该方法，以缩减计算范围。第二个参数是到目前为止计算出的部分阶乘结果。

**factorial**变量本身被赋的就是在闭包上调用**trampoline()**方法的结果。

Groovy中的尾递归实现非常出彩，没有对语言本身做任何修改就实现了。当我们调用**trampoline()**方法时，该闭包会直接返回一个特殊类**TrampolineClosure**的一个实例。当我们向该实例传递参数时，比如像**factorial(5, 1)**中这样，其实是调用了该实例的**call()**方法。该方法使用了一个简单的**for**循环来调用闭包上的**call**方法，直到不再产生**TrampolineClosure**的实例。这种简单的技术在背后将递归调用转换成了一个简单的迭代。

这种递归之所以叫作尾递归，是因为方法中最后的表达式或者是结束递归，或者是调用自身。相反，在直接递归计算阶乘时，最后的表达式调用的是`*`，即乘法操作符。

运行新的尾递归版本的**factorial()**，我们会发现，这个版本已经没有直接递归版本的缺点：

```
factorial of 5 is 120
Number of bits in the result is 24654
```