

我们会逐一深入探讨这些标准。对于自己应该回答的问题，你得做到心中有数。

7.4.1 低风险

假设你有一个核心的支付处理规则引擎，每天要处理一百万笔交易。这是一个大概运行了七年、稳定的Java软件，但并没做过太多测试，代码里有大量死角。对于引入新语言来说，支付处理引擎显然是高危区，更不用说它本来跑得好好的，而且测试没做到全面覆盖，开发人员也还没完全弄明白它是怎么回事儿。

但系统不可能只有核心部分。比如更完善的测试明显会对系统有帮助。Scala有一个非常好的测试框架：ScalaTest（我们会在第11章介绍），可以用来测试Java或Scala代码。开发人员能用它写出跟JUnit相似但简洁得多的测试代码。所以一旦度过了ScalaTest的学习曲线，开发人员就能非常高效地增加测试覆盖面。而且ScalaTest对于逐步在代码库中引入行为驱动开发这样的概念很有办法。在将来要对核心的某些部分进行重构或替换时，不管最终新的处理引擎是用Java还是用Scala写，能用上现代测试特性真的很有帮助。

或者假设你需要建一个Web控制台，以便操作员能管理支付处理系统后台一些不太重要的静态数据。开发人员都知道Struts和JSF，可对这两种技术都提不起兴趣。这是另外一个试用新语言和技术栈的低风险区。Grails是个很抢眼的选择（基于Groovy的Web框架，受Ruby on Rails启发）。开发人员在经过一些研究后（Matt Raible也做过一个非常有趣的调研），一致认为Grails是生产率最高的Web框架。7

因为是集中在低风险区的有限试点上做实验，如果所尝试的技术栈不适合自己的团队或系统，经理可以随时终止项目，不用中断太久就可以转移到不同的交付技术上。

7.4.2 与 Java 的交互操作

你肯定不想把原来写的那些Java代码弃之不用！很多组织都是因为这个原因迟迟不肯引入新的编程语言。但因为备选语言是跑在JVM上的，所以可以充分发挥原有代码的作用，这样问题变成了怎么让已有代码库的价值最大化，而不是抛弃正在使用的代码。

JVM上的备选语言跟Java之间的互操作简单利落，当然也能部署到原先的环境中。在讨论这个问题时一定要请管理生产环境的同仁到场参与。在把非JavaJVM语言加入到系统中时，你需要充分运用他们的专业经验。这也有助于消除他们对支持新方案的担忧，还能降低风险。

注意 DSL一般都是用动态层语言构建的（某些情况下也有稳定层语言），所以它们大多数是通过其内置语言运行在JVM上。

有些语言跟Java交互操作起来更容易。我们发现最流行的JVM备选语言（比如Groovy、Scala、Clojure、Jython和JRuby）都跟Java互操作得很好（而且其中某些语言的集成做得非常棒，几乎天衣无缝）。如果你确实是个谨小慎微的人，可以先做几个实验，很快，也很容易，而且你肯定能明白集成是如何工作的。

我们以Groovy为例。在Groovy代码里可以用我们熟悉的import语句直接导入Java包。用基于Groovy的Grails框架可以快速搭建一个网站，而引用对象仍然是Java模型对象。反过来说，在Java代码里调用Groovy代码也非常容易，有各种各样的办法，得到的也是熟悉的Java对象。比如从Java里调用Groovy代码处理JSON数据，并让它返回一个Java对象。

7.4.3 良好的工具和测试支持

大多数开发人员一旦习惯了已有环境，就会低估他们能节省的时间。有强大的IDE和构建工具、测试工具帮他们快速生产出高质量的软件。Java开发人员多年来受益于优秀的支持工具，所以一定要记得其他语言的成熟度可能还没达到相同的水平。

有些语言（比如Groovy）在编译、测试和最终结果部署方面长期以来都有IDE的支持。而其他语言的工具可能羽翼未丰。比如说Scala的IDE就不像Java的那么好用，但Scala粉觉得它的强大和简洁完全可以弥补当前IDE的不足。

还有个相关的问题，当备选语言社区开发出供自己使用的大工具（比如Clojure的构建工具Leiningen）后，可能不太好调整它去处理其他语言。这就是说开发团队要认真考虑该如何分配项目，特别是在部署各自独立但又相互关联的组件时。

7.4.4 备选语言学习难度

学一门新语言总归需要时间，而且如果开发团队不熟悉该语言的范式，时间会更长。如果新语言是面向对象的，并有类C的语法（比如Groovy），那大多数Java开发团队都能轻松掌握。

可如果偏离了这一范式，偏离程度越大，Java开发人员觉得越难学。Scala试图在面向对象和函数式两个世界之间架起一座桥梁，但这种融合对于大规模软件项目是否可行仍然没有定论。在特别流行的几种备选语言中，Clojure可能会带来不可思议的好处，但开发团队在学习Clojure的函数式属性和Lisp语法时，也需要非常多的再培训工作。

还有一种选择是看看重新实现已有语言的JVM语言。Ruby和Python都是非常成熟的语言，有大量的材料可供开发人员学习。这些语言的JVM替身对于想采用易学的非Java语言的开发团队来说，是不错的起点。

7.4.5 使用备选语言的开发者

组织必须考虑现实情况：他们不可能总能雇到前2%的人（不管他们在广告里怎么忽悠），而且开发团队的成员也不会整年一成不变。某些语言，比如Groovy和Scala，已经足够成熟了，所以有相当的开发人员可以招募。但像Clojure这样还在想办法推广自己的语言，要找到优秀的Clojure开发人员仍然不太容易。

警告 对重新实现的语言的警告：比如说，很多现有的用Ruby写的包和应用程序，只在原始的基于C的实现下测试过。这就是说要在JVM上使用它们可能会有问题。在选择平台时，如果你计划采用整个用重新实现的语言编写的技术栈，那就应该把额外的测试时间考虑在内。

重新实现的语言（JRuby、Jython等）在这个问题上很可能再一次发挥作用。简历上有JRuby的开发人员可能很少，但因为它就是JVM上的Ruby，而Ruby开发人员非常多（熟悉C版本的Ruby开发人员很容易掌握在JVM上运行导致的差异）。

要理解备选JVM语言的设计选择及限制，你要先理解JVM如何支持多种语言。

7.5 JVM 对备选语言的支持

一种语言要在JVM上运行可能有两种方式：

- 有一个产生类文件的编译器；
- 有一个用JVM字节码实现的解释器。

无论哪种方式，通常都会有个运行时环境为执行该语言编写的程序提供支持。图7-4展示了Java和一种典型非Java语言的运行时环境栈。



7

图7-4 非Java语言运行时支持

这些运行时支持系统复杂度各不相同，主要取决于给定的非Java语言运行时所要掌握的资源数量。几乎在所有情况下，运行时都会作为可执行程序类路径（classpath）上的一组JAR文件，并且要在程序执行开始之前启动。

本书的重点是编译型语言。至于Rhino等解释型语言，只是为了内容的完整性才提一下，所以我们不会在上面花太多篇幅。在本节剩下的内容中，我们会介绍备选语言所需的运行时支持（甚至还包括编译型语言），然后探讨编译器小说（compiler fiction）——那些可能不会出现在底层字节码中、由编译器合成的该语言特有的功能。

7.5.1 非 Java 语言的运行时环境

有一种评估语言运行时环境复杂度的简单办法：看运行时实现中JAR文件的大小。按这个标准，Clojure的运行时环境量级很轻，而JRuby语言则需要更多支持。

这个标准其实不是特别公平，因为有些语言把很多类库和功能都打包在标准发布版里了，而有些却不这么做。但如果不去深究的话，它是个挺有用的经验。

通常来说，运行时环境是要帮助非Java语言的类型系统和其他特性符合期望的语义。备选语言对基本编程概念的看法有时和Java并不完全一致。

比如，Java的面向对象方式并不是放之四海而皆准的。在Ruby中，运行时可以往一个单独的对象里添加一个额外的方法，而这个方法在定义类时还不知道是什么，也不是在相同类的其他实例上定义的。JRuby的实现需要把这个属性（不太明白为什么叫“开放类”）照搬过来。而这种高级支持只能放在JRuby的运行时中。

7.5.2 编译器小说

语言的某些特性是由编程环境和高层语言合成的，在底层JVM中根本不存在。这些特性称为编译器小说。我们在第6章已经遇到过一个例子了——Java的字符串合并。

提示 你应该了解一下这些特性是如何实现的，否则你的代码可能跑得慢，甚至可能毁掉整个过程。有时运行时环境要做大量的工作来合成某个特性。

Java中的编译器小说还包括检查型异常和内部类（如有必要，内部类总会被转换成带有特殊合成访问方法的顶层类，如图7-5所示）。如果你曾经探究过JAR文件的内部（用jar tvf），见到过许多名字中有\$的类，它们就是被取出并转换成“常规”类的内部类。

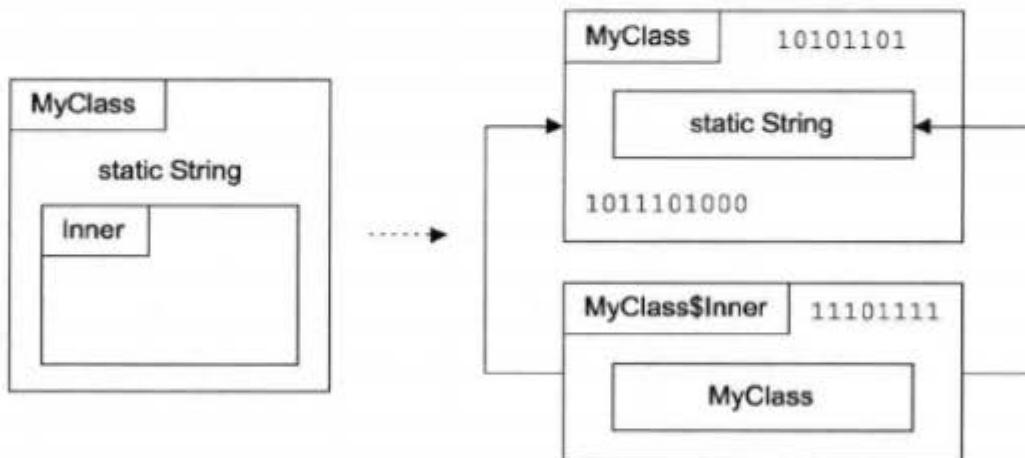


图7-5 用编译器小说实现的内部类

备选语言也有编译器小说。某些情况下，这些编译器小说甚至形成了语言的核心功能。我们来看两个重要的例子。

1. 函数是一等值

我们在7.1节介绍了函数式编程的关键概念——函数应该是能放到变量中的值。这通常说成“函数是一等值”。我们也指出Java对函数的建模方式并不太好。

本书第三部分讨论的所有非Java语言都把函数当做一等值。也就是说函数可以放在变量中、传给方法，并可以像操作其他任何值一样操作。JVM只能把类当做最小的代码和功能单元，所以现在所有的非Java语言都用小型匿名类作为函数的载体（不过这在Java 8中可能有所改变）。

解决源码和JVM字节码之间这种差异的办法是，记住对象只是把数据和操作数据的方法绑在一起的东西。请想象一个没有状态、只有一个方法的对象，比如第4章Callable接口的匿名实现

类。把这样一个对象放到变量中，作为参数传递，然后调用它的call()方法，这一切都很正常，像这样：

```
Callable<String> myFn = new Callable<String>() {
    @Override
    public String call() {
        return "The result";
    }
};

try {
    System.out.println(myFn.call());
} catch (Exception e) {
}
```

我们把异常处理忽略掉了，因为在这个例子中myFn的call()方法不可能抛出异常。

注意 在这个例子中，myFn变量是一个匿名类型，所以在编译后它看起来应该是个类似NameOfEnclosingClass\$1.class的东西。类的序号从1开始，并且编译器每遇到一个就加1。如果它们是动态创建的，并且数量很多（就像在JRuby语言中那样），会对存放类定义的PermGen内存区域造成压力。

尽管Java对此没有任何特殊的语法，但Java程序员经常用这个技巧创建匿名实现类。这就是说结果可能会有点冗长。我们所讨论的所有语言都提供了编写这些函数值（也叫函数字面值、匿名函数）的特殊语法。它们是函数式编程风格的支柱，Scala和Clojure做得都不错。

2. 多继承

还有一个例子，在Java（和JVM）中没办法表示实现的多继承。实现多继承的唯一办法就是使用接口，可它不允许有任何具体的方法。

相反，在Scala中特性机制（trait）允许把方法的实现混合到类中，所以它提供了不同的继承视图。我们会在第9章全面介绍。现在只要记住这种行为必须由Scala的编译器和运行时合成，在VM层面不提供这种特性。

对JVM上可用的不同类型的语言及其独特功能的实现方式就介绍到这里。

7.6 小结

JVM上的备选语言已经有了长足的发展。对于某些特定的问题，它们现在提供的解决方案要比Java好，并且还能与原来用Java技术实现的系统及投资兼容。这就是说，即便对于Java的推销者而言，Java也不总是所有编程任务的首选。

了解语言的不同分类方式（静态类型与动态类型、命令式与函数式、编译型与解释型）是为不同任务挑选正确语言的基础。

对于多语言程序员来说，编程语言大致可以分为三层：稳定层、动态层和特定领域层。Java和Scala这样的语言最好用来做稳定层的软件开发，而诸如Groovy和Clojure等其他语言更适合完成动态层或特定领域层的任务。

某些编程难题属于特定的层次，比如快速Web开发属于动态层，而建模企业消息属于特定领域层。

有必要再次强调一下，不要在已有生产系统的核心业务功能中引入新语言。对于核心功能区而言，支持级别高、测试覆盖率优异，并且有稳定的良好记录非常重要。与其从这里入手，还不如选一个风险低的领域部署备选语言。

不要忘了每个团队和项目都有自己独特的个性，这会影响选择语言时的决策。所以这个问题没有标准答案。在选择一门新语言时，项目经理和技术负责人必须把项目和团队的特性考虑在内。

一个都是经验丰富的技术人员组成的小团队可能会选择Clojure，因为它设计清晰、精巧并且强大（他们才不管概念的复杂性和招人的难度呢）。而一个Web团队，希望团队能快速扩充，能吸引年轻人，他们可能会因为生产率和储备相对较丰富的人才库而选择Groovy和Grails。

Groovy、Scala和Clojure是JVM语言中的领头羊。读完本书后，你能学到这三种最有前途的JVM备选语言的基础知识，并让自己的编程工具箱越来越有意思。

下一章我们会学习第一种语言：Groovy。

Groovy：Java的动态伴侣

本章内容

- 为什么学习Groovy
- Groovy的基本语法
- Groovy和Java之间的差别
- Groovy之于Java独有的特性
- Groovy为何又是脚本语言
- Groovy与Java的互操作性

Groovy是一种面向对象的动态类型语言，跟Java一样运行在JVM上。实际上，你可以把它看成是给Java静态世界补充动态能力的语言。Groovy项目最初是由James Strachan和Bob McWhirter在2003年末创建的，2004年其领导者变成了Guillaume Laforge。如今<http://groovy.codehaus.org/>上的Groovy社区仍在蓬勃发展。人们认为Groovy是继Java之后最流行的JVM语言。

受到Smalltalk、Ruby和Python的启发，Groovy已经实现了几个Java不具备的语言特性，比如：

- 函数字面值；
- 对集合的一等^①支持；
- 对正则表达式的一等支持；
- 对XML处理的一等支持。

注意 在Groovy中，函数字面值也叫闭包。正如第7章所说的，它们是能够放进变量中的函数，能传递给方法，能像其他值一样操作。

那么我们为什么要用Groovy呢？如果你还记得第7章的多语言编程金字塔，也应该还记得Java不是解决动态层问题的理想语言。这些问题包括快速Web开发、原型设计、脚本处理以及其他很多问题。Groovy就是要解决这些问题。

这里有一个体现Groovy价值的例子。比如老板让你写一个Java例程，把一堆Java bean转成XML。用Java当然可以完成这个任务，而且有很多API和类库可以选用：

^① 这里所说的“一等”是指内置到语言的语法中，不需要调用类库。

- Java 6中的Java Architecture for XML Binding (JAXB) 和Java API for XML Processing (JAXP);
- Codehaus上的XStream类库;
- Apache的XMLBeans类库。

当然还不止这些.....

这个过程很费工。比如说，要在JAXB下输出Person对象对应的XML，你必须写出：

```
JAXBContext jc = JAXBContext.newInstance("net.teamsparq.domain");
ObjectFactory objFactory = new ObjectFactory();
Person person = (Person) objFactory.createPerson();
person.setId(2);
person.setName("Gweneth");
Marshaller m = jc.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(person, System.out);
```

Person类必须是标准的Java bean，有完整的获取和设置方法。

Groovy的方式与此不同，因为它把XML当做一等公民对待。这是上面那段代码的Groovy实现：

```
def writer = new StringWriter();
def xml = new groovy.xml.MarkupBuilder(writer);
xml.person(id:2) {
    name 'Gweneth'
    age 1
}
println writer.toString();
```

看到了吧，用这种语言写代码非常快，并且它和Java很像，Java开发人员很容易掌握。

Groovy还可以帮你减少套路代码的编写工作，比如Groovy处理XML和循环遍历集合的方式要比Java更简洁。因为Groovy跟Java的互操作性很好，所以在Java中很容易利用Groovy的动态性和语言特性。

因为跟Java的语法很像，所以对于Java开发人员来说，Groovy的学习曲线很平滑，并且只要有Groovy的JAR就可以开始了。希望你看完本章后能跟新语言拍档默契！

既然Groovy在由JVM执行之前经过了完整的分析、编译和生成过程，有些开发人员会想：“为什么它不能在编译时把那些明显的错误挑出来呢？”要记住Groovy是动态语言，它的类型检查和绑定都是在运行时做的。

Groovy的性能

如果你的软件性能要求很严格，Groovy语言并不是最好的选择。Groovy的对象都扩展自GroovyObject，它有一个invokeMethod (String name, Object args)方法。Groovy的方法不能像Java中那样直接调用，而是通过之前提到的invokeMethod (String name, Object args)方法执行。这个方法会自行执行一些反射调用和查找，这自然会降低处理速度。Groovy语言的开发者已经做了一些优化，而且在接下来的新版本中，Groovy会利用JVM中新字节码invokedynamic做更多优化。

Groovy在做某些重活时还是靠Java，并且它要调用已有的Java类库很容易。因为Groovy能和Java一起使用它的动态类型和新语言特性，所以它是一种卓越的快速原型设计语言。Groovy还能作为脚本处理语言使用，因此它以Java的灵活、动态伴侣而著称。

本章一开始会跑几个简单的Groovy例子。一旦你熟悉了简单Groovy程序的运行，就可以开始学习Groovy的具体语法了，还有对于Java开发人员来说比较难缠的Groovy内容。本章接下来就深挖一下Groovy的真金真银，讨论Java不具备的几个语言特性。最后，你可以学习在JVM上混合使用Java和Groovy代码，成为一名多语言程序员。

8.1 Groovy入门

如果你还没装Groovy，请先参照附录C在你的机器中把它搭起来，然后再编译和运行本章的第一个例子。

本节会向你展示如何用命令行编译和执行Groovy，以便你在任何操作系统上都能应用自如。我们还会介绍Groovy控制台，一个宝贵的、操作系统无关的暂存器环境，非常适合用来练手。

装好了吗？那我们就来编译一些Groovy代码，让它们跑起来吧！

8.1.1 编译和运行

这里有些你应该了解的Groovy命令行工具，特别是编译器（groovyc）和运行时执行器（groovy）。它们两个基本上就相当于javac和java。

为什么代码示例的编码风格变了？

越往后，本章中的示例代码的语法和语义越像纯粹地道的Groovy。希望这样能让你更容易从Java向Groovy转移。再向你推荐一本非常优秀的书：Kenneth A. Kousen编著的*Making Java Groovy* (Manning, 2012)。

我们来看一个简单的Groovy脚本，它可以输出下面的内容^①，也借此熟悉一下命令行工具：

It's Groovy baby, yeah!

打开命令行提示符，执行如下操作。

- (1) 随便找个目录，在里面创建一个HelloGroovy.groovy文件。
- (2) 编辑这个文件，加上这一行：

```
System.out.println("It's Groovy baby, yeah!");
```

- (3) 保存HelloGroovy.groovy。
- (4) 用下面这个命令编译它：

```
groovyc HelloGroovy.groovy
```

^①感谢《王牌大贱谍》！

(5)用下面这个命令运行它:

```
groovy HelloGroovy
```

提示 如果Groovy源文件在CLASSPATH下, 可以跳过编译。如果需要, Groovy运行时会先在源文件上执行groovyc。

恭喜, 你刚刚运行了有生以来第一行Groovy代码!

跟Java一样, 你可以在命令行中编写、编译和执行Groovy代码, 但要处理CLASSPATH之类的事情时, 你很快就会觉得这么做太笨了。主流的Java IDE (Eclipse、IntelliJ和NetBeans) 对Groovy的支持都很好, 但Groovy也提供了一个控制台供你运行代码。这个控制台非常适合快速演练小型解决方案或原型, 因为用它比用正式的IDE快得多。

8.1.2 Groovy 控制台

本章会用Groovy控制台运行示例代码, 因为它是一个好用、轻量的IDE。要启动控制台, 请在命令行中执行groovyConsole。

它会弹出一个类似图8-1这样的独立窗口。

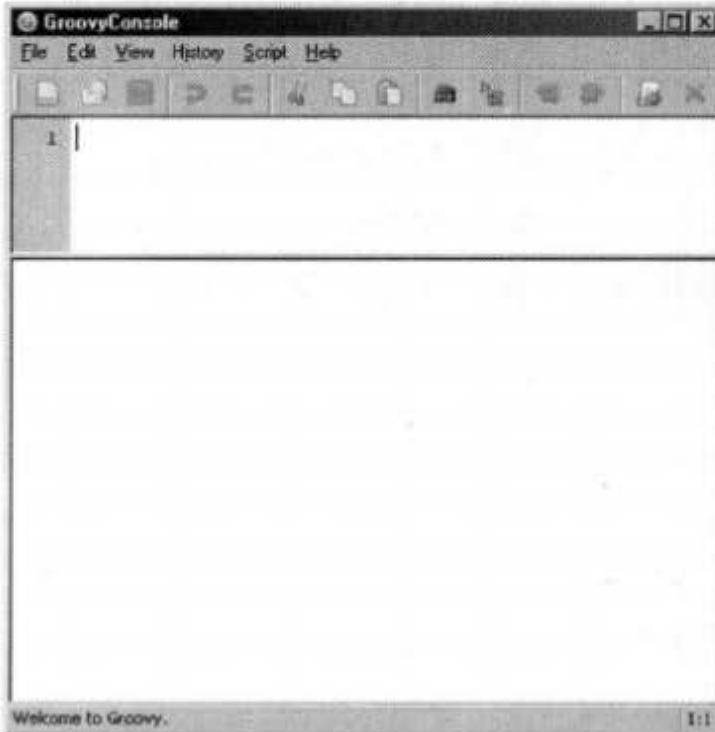


图8-1 Groovy控制台

首先, 你应该取消勾选View (视图) 菜单中的Show Script in Output (在输出中显示脚本) 选项。这会让输出简单一点儿。现在你可以运行一下前面那个例子中的Groovy代码, 以确保控制台能正常工作。在控制台的顶部面板中输入下面这行代码:

```
System.out.println("It's Groovy baby, yeah!");
```

然后点击Execute Script（执行脚本）按钮，或者用快捷键Ctrl-R。Groovy控制台就会在底部面板中显示如下输出：

```
It's Groovy baby, yeah!
```

如你所见，输出面板显示了刚刚执行的那个表达式的计算结果。

现在你已经知道如何快速执行Groovy代码了，是时候学一些Groovy的语法和语义了。

8.2 Groovy 101：语法和语义

上一节只写了一行Groovy语句，没有任何类或方法之类的结构（用Java时会需要）。实际上你写的是一个Groovy脚本。

Groovy脚本

跟Java不同，Groovy的源码可以当做脚本执行。比如说，如果你有一段代码放在类定义之外，那段代码还是可以执行。像其他动态脚本语言（比如Ruby或Python）一样，Groovy脚本在JVM上执行之前要在内存中经过完整的分析、编译和生成过程。任何能在Groovy控制台执行的代码都可以保存到.groovy文件，经过编译后，就可以作为脚本运行。一些开发人员已经用Groovy脚本取代了shell脚本，因为它们功能更强，更易于编写，并且只要装了JVM，就可以在任何平台上运行。给你一个性能方面的小提示，请使用groovyserv类库，它会启动JVM和Groovy扩展，让脚本运行得更快。

Groovy的一个关键特性是可以使用跟Java中一样的结构，语法也类似。为了突出这种相似性，请在Groovy控制台中执行下面这段类似Java的代码：

```
public class PrintStatement
{
    public static void main(String[] args)
    {
        System.out.println("It's Groovy baby, yeah!");
    }
}
```

结果和前面那个只有一行的Groovy脚本一样，都是输出"It's Groovy baby, yeah!"。除了使用Groovy控制台，你还可以把源码放到PrintStatement.groovy源文件中，用groovyc编译它，然后用groovy执行。换句话说，你能像Java中那样带着类和方法编写Groovy源码。

提示 在Groovy中几乎可以使用所有Java普通语法，所以while/for循环、if/else结构、switch语句等，都会按你期望的方式工作。所有新语法及主要差异都会在本节及相应章节中重点阐述。

随着本章内容的深入，我们会向你介绍Groovy特有的语法惯用语，例子也会从类似Java的语法向更纯粹的Groovy语法转变。你已经习惯了结构沉重的Java代码，再见到像脚本一样简洁的Groovy语法，很容易发现两者的差异。

本节的剩余部分会介绍Groovy的基本语法和语义，以及它们为什么能帮助开发人员。具体来说，我们会探讨：

- 默认导入；
- 数字处理；
- 变量、动态与静态类型，以及作用域；
- 列表与映射的语法。

首先，理解Groovy提供了哪些开箱即用的东西很重要。我们先来看看Groovy脚本或程序的默认导入。

8.2.1 默认导入

Groovy会默认导入一些语言包和工具包，以提供基本的语言支持。Groovy还会导入一系列的Java包，以便为其初始功能提供更广泛的基础。下面这个导入列表总是隐含在Groovy代码之中：

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

要使用更多的包和类，可以像Java一样用import语句。比如要从Java中得到所有Math类，只要在Groovy源码里加上`import java.math.*;`就行了。

设置可选的JAR文件

为了添加功能（比如内存数据库及其驱动），可以在Groovy安装中添加可选JAR。Groovy为此提供了一个惯用语：通常是在脚本中使用@Grab注解。另外一种办法（在你仍在学习Groovy时）是效仿Java，把JAR文件加到CLASSPATH中。

下面就来使用一下默认的语言支持，并看看Java和Groovy在数字处理上的差异。

8.2.2 数字处理

Groovy能动态计算数学表达式，并且它采用最小意外原则。这一原则在处理浮点数时（比如

3.2) 尤其明显。Groovy在底层用Java中的`BigDecimal`表示浮点数，但它会确保`BigDecimal`的行为尽量符合开发人员的期望。

1. Java和`BigDecimal`

我们来看一个经常会让开发人员头疼的数字处理问题。在Java中，如果在`BigDecimal`上加0.2，你觉得答案应该是什么？缺乏经验的Java开发人员在没看Javadoc的情况下很可能执行下面这种代码，它会返回一个极其恐怖的结果：3.20000000000000011102230246251565404236316680908203125。

```
BigDecimal x = new BigDecimal(3);
BigDecimal y = new BigDecimal(0.2);
System.out.println(x.add(y));
```

经验丰富的Java开发人员知道最好用`BigDecimal(String val)`，而不是用将数字作为参数的`BigDecimal`构造方法。以字符串为参数的构造方法写出来的代码会产生预期答案3.2：

```
BigDecimal x = new BigDecimal("3");
BigDecimal y = new BigDecimal("0.2");
System.out.println(x.add(y));
```

这有点悖于常理，所以Groovy默认采用了以字符串为参数的构造方法，解决了这一问题。

2. Groovy和`BigDecimal`

在Groovy中处理浮点数（在底层用`BigDecimal`表示）时，会自动使用以字符串为参数的构造方法，`3 + 0.2`会得到3.2。你可以在Groovy控制台中输入下面的指令亲自证实一下：

```
3 + 0.2;
```

你会发现Groovy对BEDMAS^①的支持是正确的。并且在需要时能无缝切换数字类型（比如`int`和`double`）。

用Groovy进行数学运算比Java简单。如果你想了解底层细节，可以访问<http://groovy.codehaus.org/Groovy+Math>，那里有所有的细节信息。

接下来我们学习Groovy如何处理变量和作用域。因为Groovy的动态性和执行脚本的能力，它在这方面的语义规则和Java稍有不同。

8.2.3 变量、动态与静态类型、作用域

因为Groovy是一种能作为脚本语言的动态语言，所以你要清楚动态类型和静态类型一些细微差别，还需要了解Groovy如何限定变量的作用域。

提示 如果你意在让Groovy代码与Java互操作，它也能在可能的情况下使用静态类型，因为它简化了类型重载和调度机制。

^① 回想起你在学校的日子了吧！BEDMAS表示括号、次方、除法、乘法、加法和减法，是我们计算数学题目时所要遵循的顺序（先计算括号和次方，再计算乘除，最后计算加减）。由于地区不同，你的记忆中可能是BODMAS或PEMDAS。

首先你要理解Groovy动态类型和静态类型的差别。

1. 动态类型与静态类型

Groovy是动态语言，所以不必指定变量的类型，变量的类型是在声明（或返回）时确定的。比如说，你可以把一个Date赋值给变量x，然后紧接着再用不同的类型给x赋值。

```
x = new Date();
x = 1;
```

用动态类型能让代码更简洁（忽略显而易见的类型信息），反馈更快，并且很灵活，可以在一个变量上赋予不同类型的对象来完成工作。对于那些想对自己使用的类型更有把握的人，Groovy也确实支持静态类型。比如：

```
Date x = new Date();
```

如果声明了静态类型变量，在用不正确的类型值对它赋值时，Groovy能检查出来。比如：

```
Exception thrown
```

```
org.codehaus.groovy.runtime.typehandling.GroovyCastException: Cannot cast
object 'Thu Oct 13 12:58:28 BST 2011' with class 'java.util.Date' to
class 'double'
```

在Groovy控制台中运行下面的代码，就可以重现上面的输出。

```
double y = -3.1499392;
y = new Date();
```

如你所料，Date类型的值不能赋给double变量。Groovy中的动态和静态类型都讨论到了，那作用域呢？

2. Groovy中的作用域

对于Groovy里的类，其作用域跟Java一样，类、方法、循环作用域的变量，它们的作用域都跟你想的一样。但涉及Groovy脚本时，这个话题就变得比较有意思了。

提示 记住，作为脚本的Groovy代码不在平常的类和方法结构中。8.1.1节已经给过一个例子了。

简单说，Groovy脚本有两种作用域。

□ 绑定域，绑定域是脚本的全局作用域。

□ 本地域，本地域就是变量的作用域局限于声明它们的代码块。对于在脚本代码块内声明的变量（比如在脚本的顶部），如果是定义过的变量，其作用域就是定义它的本地域。

能在脚本中使用全局变量可以极大提高代码的灵活性。它和Java中类范围内的变量有点像。定义变量是指被声明为静态类型，或用特殊的def关键字定义的变量（表明它是未确定类型的定义变量）。

在脚本中声明的方法访问不了本地域。如果你调用一个试图引用本地域中的变量的方法，会提示类似下面的错误消息：

```
groovy.lang.MissingPropertyException: No such property: hello for class:  
listing_8_2  
...
```

下面是产生该异常的代码，说明了作用域的这个问题。

```
String hello = "Hello!";  
void checkHello()  
{  
    System.out.println(hello);  
}  
checkHello();
```

如果用hello = "Hello!"换掉上面代码里的第一行，这个方法可以成功输出“Hello”。因为hello不再定义为string，它现在的应用域是绑定域。

除了编写Groovy脚本时的这些差异，动态和静态类型、应用域、变量声明都跟你想的完全一样。接下来我们去看看Groovy内置的集合（列表和映射）支持。

8.2.4 列表和映射语法

Groovy把列表和映射（包括集合）结构当做语言中的一等公民对待，所以没必要像Java那样显式声明List和Map结构。也就是说，Groovy中的列表和映射在底层是由Java ArrayList和LinkedHashMap实现的。

使用Groovy语法最大的优势在于可以省掉很多套路化的代码，让代码更简洁，但丝毫不影响可读性。

Groovy用方括号[]指定和使用列表结构（是不是想起了Java中的原生数组语法）。下面的代码展示了如何引用第一个元素（Java），获取列表大小（4），以及将列表设置为空[]。

```
jvmLanguages = ["Java", "Groovy", "Scala", "Clojure"];  
println(jvmLanguages[0]);  
println(jvmLanguages.size());  
jvmLanguages = [];  
println(jvmLanguages);
```

看，Groovy将列表作为一等公民处理要比用java.util.List及其实现类的代码轻量得多。

因为Groovy是动态类型语言，我们可以把不同类型的值保存在列表（或映射）中，所以下面的代码也是正确的：

```
jvmLanguages = ["Java", 2, "Scala", new Date()];
```

Groovy处理映射也跟这差不多，用[:]符号，并用冒号（:）来分开键/值对。以映射键的方式引用映射中的值。下面的代码通过相应的操作展示了这些功能：

- 引用键"Java"的值100；
- 引用键"Clojure"的值"N/A"；
- 将键"Clojure"的值变成75；
- 将映射设为空（[:]）。

```
languageRatings = [Java:100, Groovy:99, Clojure:"N/A"];
println(languageRatings["Java"]);
println(languageRatings.Clojure);
languageRatings["Clojure"] = 75;
println(languageRatings["Clojure"]);
languageRatings = [:];
println languageRatings;
```

提示 你有没有注意到映射里的键是不带引号的字符串？为了让代码更简洁，Groovy对这个语法也做了调整，映射键的引号可用可不用。

这种写法很直观，用起来也舒服。Groovy把对映射和列表内置支持的概念更进了一步。

还有一些语法技巧，比如引用集合中一定范围内的元素，甚至可以用负索引引用最后一个元素。下面的代码引用了列表中的前三个元素（[Java, Groovy, Scala]）和最后一个元素（clojure）。

```
jvmLanguages = ["Java", "Groovy", "Scala", "Clojure"];
println(jvmLanguages[0..2]);
println(jvmLanguages[-1]);
```

现在，我们已经了解了Groovy的一些基本语法和语义。但在真正使用Groovy之前，还需要学习更多内容。下一节会更深入地探讨Groovy的语法和语义，重点讲解Java开发人员学习Groovy过程中那些“难缠的内容”。

8.3 与Java的差异——新手陷阱

目前你基本上已经熟练掌握Groovy的基本语法了，当然其中部分原因是它跟Java语法很像。但这种相似有时可能是“诈”，所以本节来讨论那些经常困扰Java开发人员的语法。

Groovy有大量可以省略的语法，比如：

- 语句结束处的分号；
- 返回语句（return）；
- 方法参数两边的括号；
- public访问限定符。

这类设计是为了让源码更简洁，在快速设计原型时都能体现出优势来。

其他修改包括去掉已检查和未检查异常之间的区别，相等概念的替代办法，以及不再使用内部类。我们先从最简单的开始：可选的分号和返回语句。

8.3.1 可选的分号和返回语句

在Groovy中，语句结束处的分号（;）是可选的，除非一行中有多条语句，否则都可以省略。

此外，从方法中返回对象或值时不必使用return关键字。Groovy会自动返回最后一个表达式的计算结果。

代码清单8-1演示了这些可选语法，并返回了方法中最后一个表达式的计算结果3。

代码清单8-1 分号和返回语句可以省略

```
Scratchpad pad = new Scratchpad()
println(pad.doStuff())

public class Scratchpad
{
    public Integer doStuff()
    {
        def x = 1
        def y; def String z = "Hello";
        x = 3
    }
}
```

上面的代码看起来跟Java还是很像，而Groovy的简洁风格其实还要更强。接下来你会看到Groovy如何省略方法参数两边的括号。

8.3.2 可选的参数括号

如果Groovy里的方法调用至少有一个参数，并且没有二义性，则可以省略括号。也就是说下面的代码

```
println("It's Groovy baby, yeah!")
```

可以写成

```
println "It's Groovy baby, yeah!"
```

代码变得更简洁了，可读性仍然没受影响。

下一个特性是可选的public访问限定符，再用上它，Groovy代码看起来就不太像Java了。

8

8.3.3 访问限定符

优秀的Java开发人员都知道确定类、方法和变量的访问级别是面向对象设计的重要组成部分。跟Java一样，Groovy也有public、private和protected级别；但和Java不同，Groovy的默认访问级别是public。所以我们把代码清单8-1改一下，去掉一些默认的public限定符，加几个private限定符，如代码清单8-2所示。

代码清单8-2 public是默认访问限定符

```
Scratchpad2 pad = new Scratchpad2()
println(pad.doStuff())

class Scratchpad2
{
    def private x;
    Integer doStuff()
    {
```

```

x = 1
def y; def String z = "Hello";
x = 3
}
}

```

继续语法精简的主题，作为一名Java开发人员，你对用在方法签名中抛出已检查异常的throws语句熟不熟悉？

8.3.4 异常处理

跟Java不同，Groovy不区分已检查异常和未检查异常。Groovy编译器会忽略方法签名中的所有throws语句。

在保证源码可读的前提下，Groovy采用了一些快捷语法来简化代码。接下来看一个有严重语义影响的语法变化：相等操作符。

8.3.5 Groovy 中的相等

遵循最小意外原则，Groovy把==当做Java中的equals()方法。这是直觉式开发人员的又一项福利，他们不必再像用Java时为原始类型和对象倒腾==和equals()。

检查真实的对象是否相等，需要使用Groovy内置的is()函数。这一规则有个例外，就是你仍然可以用==来检查一个对象是否为null。代码清单8-3说明了这些特性。

代码清单8-3 Groovy中的相等

```

Integer x = new Integer(2)
Integer y = new Integer(2)
Integer z = null
if (x == y)
{
    println "x == y"
}

if (!x.is(y))
{
    println "x is not y"
}

if (z.is(null))
{
    println "z is null"
}

if (z == null)
{
    println "z is null"
}

```

隐含的equals() 调用	检查对象 是否相等	用is()检查 是否为null	检查是否 为null
-------------------	--------------	--------------------	---------------

当然，如果你喜欢，仍然可以用equals()方法检查相等关系。

最后，还有一个应该简单提一下的Java结构——内部类，Groovy中它基本被一个新的语言结构取代了。

8.3.6 内部类

Groovy支持内部类，但大多数情况下我们应该用函数字面值替代它。下一节讨论函数字面值，它是一个很强的现代编程结构，应该占用更多篇幅介绍。

用Groovy可以写出更简洁的代码，而且并不影响代码的可读性，如果你喜欢，仍然可以继续使用Java的（大多数）语法结构。接下来，你会看到一些Java还不具备的Groovy语言特性。其中的某些特性很可能就是你选用Groovy的关键，比如XML处理。

8.4 Java 不具备的 Groovy 特性

Groovy具备一些Java没有的语言特性，起码Java 7还没有。优秀的Java开发人员就是在这些问题上需要向新语言求助，希望能以更优雅的方式解决它们。本节就探索几个这样的特性，包括：

- GroovyBean，更简单的bean；
- 用操作符?.实现null对象的安全访问；
- 猫王^①操作符（Elvis operator），更短的if/else结构；
- Groovy字符串，更强的字符串抽象；
- 函数字面值（即闭包），把函数当做值传递；
- 对正则表达式的本地支持；
- 更简单的XML处理。

我们会从GroovyBean开始，因为Groovy代码中经常见到它们。作为一名Java开发人员，你可能有点儿疑心，因为按JavaBean的标准来衡量的话，它们不太完整。但请你放心，GroovyBean很完整，分毫不差，并且用起来更方便。

8.4.1 GroovyBean

GroovyBean很像JavaBean，不过省略了显式声明的获取和设置方法，提供了自动构造方法，并允许你用点号（.）引用成员变量。如果需要把某个获取方法或设置方法设为private，或者希望改变默认的行为，可以显式声明那个方法，并按你的想法修改它。自动构造方法只是一个用来构造GroovyBean、传入与GroovyBean的成员变量对应的参数的映射。

不论是不辞劳苦自己输入获取方法和设置方法，还是用IDE生成，所有这些都省去了我们处理JavaBean时所要编写的大量套路化代码。

我们以一个角色扮演游戏（RPG）^②里的Character类为例来看一下GroovyBean是如何工作的。代码清单8-4会输出STR[18]，WIS[15]，这是代表GroovyBean力量和智慧的成员变量。

^① Elvis Aron Presley (1935—1977)，美国摇滚乐史上影响力最大的歌手，有摇滚乐之王的誉称。——译者注

^② 这里大力推荐一下PCGen (<http://pcgen.sf.net>)，对于RPG粉来说真是个非常好的开源项目。

代码清单8-4 探索GroovyBean

```
class Character
{
    private int strength
    private int wisdom
}

def pc = new Character(strength: 10, wisdom: 15)
pc.strength = 18
println "STR [" + pc.strength + "] WIS [" + pc.wisdom + "]"
```

它的行为跟Java里的JavaBean非常相似（封装性得以保留），而语法更精简。

提示 可以用@Immutable注解使GroovyBean不可变（意思是它的状态不可修改）。这对于传递线程安全的数据结构很有用，在并发代码中用起来更安全。第10章讨论闭包时我们还会进一步讨论不可变数据结构的概念。

接下来我们会转向Groovy检查null引用的能力。这会进一步减少套路化代码，以便你可以更快地把想法变成原型。

8.4.2 安全解引用操作符

NullPointerException^①（NPE）是所有Java开发人员都挥之不去的梦魔（很不幸）。为了避开NPE，Java程序员通常都会在引用对象之前检查一下它是否为null，特别是在他们不能保证所处理的对象不是null的情况下。如果你准备在Groovy中延续那种开发风格，为了遍历一个Person对象列表，最终编写的代码可能像下面这样（只是输出“Gweneth”）。

```
List<Person> people = [null, new Person(name:"Gweneth")]
for (Person person: people) {
    if (person != null) {
        println person.getName()
    }
}
```

Groovy引入了安全解引用运算符，用?.符号帮你去掉一些套路化的“如果对象为null”检查代码。在使用这个符号时，Groovy引入了一个特殊的null结构，表示“什么也不做”，而不是真的引用null。

在Groovy中，可以用安全解引用语法重写上面的代码：

```
people = [null, new Person(name:"Gweneth")]
for (Person person: people) {
    println person?.name
}
```

Groovy函数也支持这种安全解引用，所以Groovy的默认集合方法（比如max()方法），能自动处理好null引用。

^① Java最大的憾事就是没据实把这个叫做NullReferenceException，本书的一位作者对此一直颇多怨言！

接下来是猫王操作符，看起来和安全解引用差不多，但它是用来减少某些if/else结构中的代码的。

8.4.3 猫王操作符

用猫王操作符(?)可以把带有默认值的if/else结构写得极其短小。为什么叫猫王？因为这个符号看起来明显很像猫王鼎盛时期梳的大背头^①。用猫王操作符不用检查null，也不用重复变量。

假设你要检查王牌大贱谍是不是活跃的侦探。在Java中可能要用三元操作符：

```
String agentStatus = "Active";
String status = agentStatus != null ? agentStatus : "Inactive";
```

Groovy能缩短这个语句，是因为它能在需要时将类型强制转换为boolean，比如if语句的条件判断。在前面的代码中，Groovy把String转换为boolean，假如String是null，它会被转换成Boolean值false，所以可以省略null检查。因而前面的代码可以写成这样：

```
String agentStatus = "Active"
String status = agentStatus ? agentStatus : "Inactive"
```

但这样还是要重复agentStatus变量，Groovy可以让我们不再重复输入。用猫王操作符可以去掉重复的变量名：

```
String agentStatus = "Active"
String status = agentStatus ?: "Inactive"
```

第二个agentStatus没了，代码更简洁了。

好了，现在该去看看Groovy字符串了，看看它们跟Java常规String有什么不同。

8

8.4.4 增强型字符串

Groovy有一个String类的扩展类GString，它比Java中标准的String强，也更灵活。

尽管双引号也有效，但按照惯例，普通字符串是用开闭两个单引号定义的。比如：

```
String ordinaryString = 'ordinary string'
String ordinaryString2 = "ordinary string 2"
```

而GString必须用双引号定义。对于开发人员来说，使用它最大的好处是可以包含可在运行时计算的表达式（用\${}）。如果GString随后被转为普通字符串（比如传给了println），GString中的表达式都会被替换为其计算结果。比如：

```
String name = 'Gweneth'
def dist = 3 * 2
String crawling = "${name} is crawling ${dist} feet!"
```

其中的表达式计算后被转到可以调用toString()的Object上，或是函数字面值上。（请参见<http://groovy.codehaus.org/Strings+and+GString>了解关于函数字面值和GString规则的细节。）

^① 本书的作者都郑重声明，我们根本不知道猫王在鼎盛时期长什么样。我们真没那么老，不开玩笑！

警告 GString的底层并不是Java中的String! 尤其不应该把GString作为映射中的键, 或者比较它们是否相等。结果是不可预料的!

Groovy中另一个有点儿用的结构是三引号String或三引号GString, 它们可以在源码中定义跨行字符串。

```
"""This GString
wraps over two lines!"""
```

接下来我们要向函数字面值进军了。由于最近几年业内兴起了对函数式语言的兴趣, 这个编程技巧也成了一个热门话题。要弄懂函数字面值, 可能需要动动脑筋。如果你没用过, 也就是说如果这是你第一次用, 也许你现在就该先起身将公爵杯加满自己喜欢的饮品。

8.4.5 函数字面值

函数字面值表示一个可以当做值传递的代码块, 也可以像操作任何值一样操作。可以当参数传给方法, 可以给变量赋值, 等等。这个语言特性已经成为Java社区的讨论热点, 但对于Groovy程序员来说, 它们是标配的工具。

举例说明向来都是学习新概念的最好方法, 我们先来看几个例子吧!

假设我们有一个普通的静态方法, 要构建一个String来向作者或读者问好。我们用常规方式从这个类的外部调用该方法, 如代码清单8-5所示:

代码清单8-5 一个简单的静态函数

```
class StringUtils
{
    static String sayHello(String name)           ← 静态方法声明
    {
        if (name == "Martijn" || name == "Ben")
            "Hello author " + name + "!"
        else
            "Hello reader " + name + "!"
    }
}
println StringUtils.sayHello("Bob")             ← 调用者
```

有了函数字面值, 你不用方法或类结构也可以实现同样的功能, 只要把代码放在函数字面值里。而函数字面值又可以赋值给一个变量, 从而可以被传递和执行。

代码清单8-6把函数字面值赋值给sayHello, 传入参数“Martijn”, 并最终输出“Hello author Martijn!”。

代码清单8-6 使用简单的函数字面值

```
def sayHello =                                ← 函数字面值赋值
{
    name ->
        if (name == "Martijn" || name == "Ben")
            "Hello author " + name + "!"
        else
            "Hello reader " + name + "!"
```

① 变量与处理逻辑分开

```

    "Hello reader " + name + "!"
}
println(sayHello("Martijn"))

```

输出结果

注意函数字面值开始处的`{`。把传入函数字面值的参数跟处理逻辑分开的箭头`(->)`^①。最后是函数字面值结束处的`}`。

在代码清单8-6中，函数字面值的定义方式非常像方法的定义方式。因此你可能在想：“它们看起来也不是特别有用！”只有开始用它们创作（用函数方式思考）时，你才能真正发现它们的好，比如说跟Groovy对集合的内置支持结合起来之后，函数字面值会特别强大。

8.4.6 内置的集合操作

Groovy有几个可以用于集合（列表和映射）的内置方法。这种在语言层面对集合的支持，跟函数结合在一起，可以极大减少程序员在Java中必写的那些套路化代码；并且代码仍然很容易看懂，不影响维护。

表8-1是一些使用了函数字面值的内置函数。

表8-1 Groovy中的部分集合函数

方 法	描 述
each	遍历集合，对其中的每一项应用函数字面值
collect	收集在集合中每一项上应用函数字面值的返回结果（相当于其他语言map/reduce中的map函数）
inject	用函数字面值处理集合并构建返回值（相当于其他语言里map/reduce中的reduce函数）
findAll	找到集合中所有与函数字面值匹配的元素
max	返回集合中的最大值
min	返回集合中的最小值

Java编程过程中遍历集合，并对其中每个对象执行某种操作是很常见的任务。比如说，如果你想在Java 7中输出电影名称，很可能会写出如代码清单8-7所示的代码：^②

代码清单8-7 在Java 7中输出一个集合

```

List<String> movieTitles = new ArrayList<>();
movieTitles.add("Seven");
movieTitles.add("Snow White");
movieTitles.add("Die Hard");

for (String movieTitle : movieTitles)
{
    System.out.println(movieTitle);
}

```

Java中有帮你少写代码的技巧，但不管怎样都要用某种循环结构手工遍历电影名称的List。在Groovy里可以用内置的集合遍历功能（each函数），并且函数字面值可以减少大量你需要

^① 不，我们可不会告诉你谁喜欢《白雪公主》（反正不是我俩）！

自己编写的代码。此外，这样还能反转列表和所要执行的算法之间的关系。不再是把集合传递到方法中，而是把方法传入到集合中！

下面的代码和代码清单8-7所做的工作完全一样，但只有短短的两行，很容易读懂：

```
movieTitles = ["Seven", "SnowWhite", "Die Hard"]
movieTitles.each({x -> println x})
```

实际上，如果使用隐含的it变量，这段代码还可以变得更精简，it变量可以用在单参的函数字面值中，代码如下所示^①：

```
movieTitles = ["Seven", "SnowWhite", "Die Hard"]
movieTitles.each({println it})
```

看，这段代码简洁易读，并且效果和Java 7那个版本一样。

提示 只能介绍这么多了，如果你想研究更多例子，推荐你到Groovy的网站上去看看与集合相关的内容(<http://groovy.codehaus.org/JN1015-Collections>)，或者读读Dierk König、Guillaume Laforge、Paul King、Jon Skeet和Hamlet D'Arcy合著的*Groovy in Action, second edition*(Manning, 2012)，这是一本相当不错的书。

下一个语言特性是Groovy内置的正则表达式支持，你可能要花点儿时间才能熟悉，所以借着咖啡劲儿，我们赶紧来看看吧！

8.4.7 对正则表达式的内置支持

Groovy把正则表达式当做语言的一部分，所以用Groovy处理文本要比Java简单得多。表8-2中是Groovy可用的正则表达式语法，以及Java与之对应的东西。

表8-2 Groovy正则表达式语法

方 法	描述及Java中的对等物
~	创建一个模式（创建一个编译的Java Pattern对象）
==	创建一个匹配器（创建一个Java Matcher对象）
==~	计算字符串（相当于在Pattern上调用Java match()方法）

假设你从一个硬件上收到了一些日志数据，要部分匹配其中一些错误日志。比如查找模式1010的实例，然后再找0101。在Java 7中，实现代码可能如下所示。

```
Pattern pattern = Pattern.compile("1010");
String input = "1010";
Matcher matcher = pattern.matcher(input);
if (input.matches("1010"))
{
    input = matcher.replaceFirst("0101");
```

^① Groovy高手会说实际上还可以简化，一行足矣！

```

    System.out.println(input);
}

```

在Groovy中，每行代码都变短了，因为Pattern和Matcher对象是内置在语言中的。当然，输出（0101）还和原来一样，请看代码。

```

def pattern = /1010/
def input = "1010"
def matcher = input =~ pattern
if (input ==~ pattern)
{
    input = matcher.replaceFirst("0101")
    println input
}

```

Groovy支持完整的正则表达式语义，所采用的方式和Java一样，所以你熟悉的那种灵活性还在。

正则表达式跟函数字面值结合得也很好。比如分析String得到一个人的名字和年龄，并输出详细信息。

```

("Hazel 1" =~ /(\w+) (\d+)/).each {full, name, age
    -> println "$name is $age years old."}

```

或许你应该借这个机会稍微放松一下，接下来我们马上就要探索一项完全不同的技术：XML处理。

8.4.8 简单的 XML 处理

Groovy有构建器的概念，用Groovy原生语法可以处理任何树型结构的数据。包括HTML、XML和JSON。Groovy理解开发人员想轻松处理这种数据的需求，所以提供了开箱即用的构建器。

XML：一种被滥用的语言

XML是一种卓越、详细的数据交换语言，但现在已经变得如洪水猛兽一般了。为什么呢？因为软件开发人员已经把XML当成编程语言来用了，可它不是图灵完备^①的语言，所以它不适合干这些事。希望XML能在你的项目中得其所哉，只是用来交换数据。

本节重点是XML，一种常用的交换数据格式。尽管Java语言的核心（通过JAXB和JAXP）以及浩浩荡荡的第三方类库（XStream、Xerces、Xalan等）组成了庞大的XML处理大军，但选哪个方案经常让人难以抉择，并且采用相应方案的Java代码会变得非常冗长。

本节会带你用Groovy创建XML，并告诉你如何把XML解析为GroovyBean。

1. 创建XML

用Groovy构建XML文档非常简单，比如person：

^① 对于一种语言来说，如果是图灵完备的，那它至少必须能做条件分支判断，并能修改内存数据。

```
<person id='2'>
  <name>Gweneth</name>
  <age>1</age>
</person>
```

Groovy能用内置的MarkupBuilder产生这个XML。产生personXML记录的代码如代码清单8-8所示：

代码清单8-8 产生简单的XML

```
def writer = new StringWriter()
def xml = new groovy.xml.MarkupBuilder(writer)
xml.person(id:2) {
  name 'Gweneth'
  age 1
}
println writer.toString()
```

注意看person的起始元素（属性id设置为2）创建起来多么简单，根本不用定义Person对象。Groovy不会强迫你显式地弄一个GroovyBean来支撑XML的创建，再一次节省了时间和精力。

代码清单8-8中的例子相当简单。你可以多做些试验，把输出类型StringWriter改掉，并且可以尝试用不同的构建器，比如groovy.json.JsonBuilder()，即刻创建JSON^①。在处理更复杂的XML结构时，命名空间和其他特定构造的处理上也有额外的辅助方法。

你可能还希望执行反向操作，读取XML并把它解析成GroovyBean。

2. 解析XML

Groovy有几种解析XML输入的办法。表8-3列出了其中三个方法，这是从Groovy的官方文档（<http://docs.codehaus.org/display/GROOVY/Processing+XML>）中拿过来的。

表8-3 Groovy XML解析技术

方 法	描 述
XMLParser	支持XML文档的GPath表达式
XMLSlurper	跟XMLParser类似，但以懒加载的方式工作
DOMCategory	用一些语法支持DOM的底层解析

这三个用起来都很简单，但这一节我们主要关心XMLParser的用法。

注意 GPath是一种表达式语言。Groovy文档（<http://groovy.codehaus.org/GPath>）中有它的全部内容。

我们把代码清单8-8中产生的那个表示“Gweneth”（人名）的XML拿过来，并把它解析到一个GroovyBean Person中，如代码清单8-9所示。

① 关于这一问题，Dustin在他的博客Inspired by Actual Events（<http://marxsoftware.blogspot.com/>）上有一篇很棒的文章，标题是“Groovy 1.8 Introduces Groovy to JSON”。

代码清单8-9 用XMLParser解析XML

```

class XmlExample {
    static def PERSON =
    """
<person id='2'>
    <name>Gweneth</name>
    <age>1</age>
</person>
"""
}
class Person {def id; def name; def age}
def xmlPerson = new XMLParser().
    parseText(XmlExample.PERSON)
Person p = new Person(id: xmlPerson.@id,
                      name: xmlPerson.name.text(),
                      age: xmlPerson.age.text())
println "${p.id}, ${p.name}, ${p.age}"

```

我们一开始抄了点儿近路，把XML文档直接放在代码中了，这样它就会出现在CLASSPATH中①。真正的第一步是用XMLParser中的parseText()方法读取XML数据②。然后创建新的Person对象，给它赋值③，最后输出Person，以便你能用肉眼检查一下。

我们对Groovy的介绍到此就完成了。现在，你可能觉得心里痒痒的，想在自己的Java项目里使用一些Groovy特性！下一节，我们会带你看看Java如何跟Groovy互操作。由此你将迈出作为优秀Java开发者的重要一步：成为一名JVM多语言程序员。

8

8.5 Groovy 与 Java 的合作

这一节很短，但不要低估它的重要性！如果你是按顺序看到这里的，这里就是你要跳的龙门，跳过去你就不是只在JVM上做Java开发的人了。JVM上有多种语言作为Java的补充，优秀的Java开发者要具备使用它们的能力，Groovy是个很好的起点！

首先，你会重温一下从Groovy中调用Java是多么简单。之后你会看到Java与Groovy交互的三种常用途径，使用GroovyShell、GroovyClassLoader和GroovyScriptEngine。

我们先来重温一下Groovy里怎么调用Java。

8.5.1 从 Groovy 调用 Java

还记得吗？我们说过从Groovy调用Java很简单，你只要把JAR放到CLASSPATH中，然后用标准的import语句就行了。这儿有个例子，引入流行的Joda日期时间类库中org.joda.time包里的类^①：

```
import org.joda.time.*;
```

^① 在Java 8发布之前，实际上Joda一直都不是Java的标准日期时间类库。

可以跟在Java中一样使用这些类。下面的代码会输出当前月份的数值表示。

```
DateTime dt = new DateTime()
int month = dt.getMonthOfYear()
println month
```

哦，肯定要比这个更复杂点儿吧？

阿克巴上将：“陷阱！”^①

开个玩笑，这里没什么陷阱！真的就这么简单，那我们是不是应该看看更困难的情况？从Java调用Groovy并得到有意义的结果还是有点儿技术含量的。

8.5.2 从Java调用Groovy

从Java程序调用Groovy需要把Groovy及其相关的JAR放到这个程序的CLASSPATH下，因为它们都是运行时依赖项。

提示 只需要把GROOVY_HOME/embeddable/groovy-all-1.8.6.jar文件放到CLASSPATH中。

下面是几种从Java调用Groovy代码的办法：

- 使用Bean Scripting Framework (BSF)，即JSR 223；
- 使用GroovyShell；
- 使用GroovyClassLoader；
- 使用GroovyScriptEngine；
- 使用嵌入式Groovy控制台。

我们在这一节重点讨论最常用的办法（GroovyShell、GroovyClassLoader和GroovyScriptEngine）。先从最简单的GroovyShell开始。

1. GroovyShell

在临时性快速调用Groovy并计算表达式或类似于脚本的代码时，可以用GroovyShell。比如说，有些开发人员可能更喜欢用Groovy做数值处理，就可以调用GroovyShell执行一些数学计算。代码清单8-10会返回用Groovy的数值相加得到的结果10.4。

代码清单8-10 在Java中用GroovyShell执行Groovy代码

```
import groovy.lang.GroovyShell;
import groovy.lang.Binding;
import java.math.BigDecimal;

public class UseGroovyShell {
    public static void main(String[] args) {
```

^① 星战迷对这句在网上广为流传的话应该不会感到陌生。

```

Binding binding = new Binding();
binding.setVariable("x", 2.4);
binding.setVariable("y", 8);
GroovyShell shell = new GroovyShell(binding);
Object value = shell.evaluate("x + y");
assert value.equals(new BigDecimal(10.4));
}
}

设置shell上的binding
计算并返回表达式

```

用GroovyShell只能应付快速执行小段Groovy代码的情况，如果要与一个完整的Groovy类交互，该怎么办呢？这时可以用GroovyClassLoader。

2. GroovyClassLoader

从开发人员的角度看，GroovyClassLoader的表现很像Java的ClassLoader。找到类和想要调用的方法，然后调用就行了。

下面的代码中有一个简单的CalculateMax类，其中有个getMax方法，会使用Groovy内置的max函数。要在Java里通过GroovyClassLoader运行这个方法，需要用下面的代码创建一个Groovy文件（CalculateMax.groovy）：

```

class CalculateMax {
    def Integer getMax(List values) {
        values.max();
    }
}

```

现在我们有了要执行的Groovy脚本，可以从Java调用它了。在代码清单8-11中，从Java调用CalculateMax getMax函数，返回了传入参数中的最大值10。

8

代码清单8-11 在Java中用GroovyClassLoader执行Groovy代码

```

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import groovy.lang.GroovyClassLoader;
import groovy.lang.GroovyObject;
import org.codehaus.groovy.control.CompilationFailedException;

public class UseGroovyClassLoader {

    public static void main(String[] args) {
        GroovyClassLoader loader = new GroovyClassLoader(); // 准备GroovyClassLoader
        try {
            Class<?> groovyClass = loader.parseClass(
                new File("CalculateMax.groovy")); // 得到Groovy类
            GroovyObject groovyObject = (GroovyObject)
                groovyClass.newInstance(); // 得到Groovy类的实例
            ArrayList<Integer> numbers = new ArrayList<>();
            numbers.add(new Integer(1));
            numbers.add(new Integer(10));
            Object[] arguments = {numbers}; // 准备参数

```

```

Object value =
    groovyObject.invokeMethod("getMax", arguments);
assert value.equals(new Integer(10));
}
catch (CompilationFailedException | IOException | InstantiationException
    | IllegalAccessException e) {
    System.out.println(e.getMessage());
}
}
}
}

调用Groovy方法

```

这种技术在调用几个Groovy实用类时可能会有用。但如果要用大量的Groovy代码，我们推荐使用完整的GroovyScriptEngine。

3. GroovyScriptEngine

使用GroovyScriptEngine要指明Groovy代码的URL或所在目录。Groovy脚本引擎会加载那些脚本，并在必要时进行编译，包括其中的依赖脚本。比如说你修改了脚本B，而脚本A依赖于B，则引擎会全重新编译它们。

假设有一个Groovy脚本（Hello.groovy）定义了一个简单的“Hello”语句，后面跟着一个名字（要从Java应用程序中传入的参数）。

```
def helloStatement = "Hello ${name}"
```

然后Java程序会通过GroovyScriptEngine使用Hello.groovy，并输出一句问候，如代码清单8-12所示：

代码清单8-12 在Java中用GroovyScriptEngine执行Groovy代码

```

import groovy.lang.Binding;
import groovy.util.GroovyScriptEngine;
import groovy.util.ResourceException;
import groovy.util.ScriptException;
import java.io.IOException;

public class UseGroovyScriptEngine {
    public static void main(String[] args)
    {
        try {
            String[] roots = new String[] {"src/main/groovy"};
            GroovyScriptEngine gse =
                new GroovyScriptEngine (roots);

            Binding binding = new Binding();
            binding.setVariable("name", "Gweneth");
            Object output = gse.run("Hello.groovy", binding);
            assert output.equals("Hello Gweneth");
        }
        catch (IOException | ResourceException | ScriptException e) {
            System.out.println(e.getMessage());
        }
    }
}

设置根目录
初始化引擎
运行脚本

```

GroovyScriptEngine监控之下的任何Groovy脚本都可能被程序员一时兴起改掉。比如说，将Hello.groovy改成这样：

```
def helloStatement = "Hello ${name}, it's Groovy baby, yeah!"
```

这段Java代码下次再运行时，它就会用这个新的，更长的消息。这样Java应用程序就具备了以前根本不可能出现的动态灵活性。这在某些情况下简直是无价之宝，比如调试生产环境下的代码，在运行时修改系统属性，还有很多……

至此，对Groovy的介绍真要结束了。我们已经走了很远了！

8.6 小结

Groovy有多种引人注目的特性，这使它成为一门可以和Java共用的出色语言。你可以用和Java非常相近的语法，也可以用更精简的代码实现相同的逻辑。这种精简并不以牺牲可读性为代价，而且Java开发人员在采用跟集合、null引用处理和GroovyBean相关的新语法时不存在什么困难。然而，Groovy给Java开发人员设了几个陷阱，但你已经搞定了大多数情况，希望你能带领同事走进这片新大陆。

很多Java开发人员都对Groovy中的几个语言特性感到眼馋，希望有朝一日Java语言中也能有这些特性。其中最难掌握、也最强大的就是函数字面值，它是一种能在集合上轻松进行操作的强大编程技术（跟其他技术一起）。当然，集合享受的是一等公民的待遇，你能用更短小易用的语法来创建、修改和操作它们。

大多数Java开发人员都要在Java程序里生成或解析XML，对此Groovy也能助你一臂之力，它能用内置的XML支持帮你挑起大部分重担。

借助各种技术把Java代码和Groovy代码集成在一起解决编程问题，你已经向多语言程序员迈出了一步。

我们的Groovy旅程还没有结束。在第13章讨论快速Web开发时，还有更多的Groovy特性等着我们去使用和探索。

接下来，我们请出Scala，另外一门已在业内造成小轰动的JVM语言。Scala既是面向对象语言，也是函数式语言，要解决现代编程中进退两难的问题，Scala是值得一看的语言。

Scala：简约而不简单



本章内容

- Scala不是Java
- Scala语法及更加函数化的风格
- match表达式与模式
- Scala的类型系统和集合
- Scala并发之actor

Scala是出自学术界和编程语言研究社区的语言。由于其强大的类型系统和先进特性，又有精英团队证明了其价值，它已经赢得了一定数量开发者的青睐。

现在Scala里有很多有意思的地方，但要评判它能否完全渗入Java生态系统，以及能否挑战Java语言的霸主地位，还为时尚早。

最合理的预测是Scala会被更多的团队接受，并最终被项目采纳。在接下来的三四年中，我们预计会有大量开发人员在项目中见到Scala的身影。也就是说作为优秀的Java开发者，你应该了解它，能判断出它是否适用于自己的项目。

例子 金融风险模拟应用可能需要采用Scala新颖的面向对象方式、类型推断、灵活的语法、新的集合类（包括自然的函数式编程风格，比如映射/过滤器惯用语），以及基于actor的并发模型。

对于Java开发人员来说，刚开始接触Scala时最应该牢记于心的是：Scala不是Java。

这看起来似乎是显而易见的。毕竟，每种语言都不同，Scala当然也和Java不一样。但我们在第7章提到过，有些语言非常相似。第8章介绍Groovy时也在强调它和Java相似的地方。希望这些内容在你首次探索Groovy这门非Java JVM语言时能够对你有所帮助。

本章我们想做点不一样的事情，先突出Scala中一些非常独特的语言特性。我们喜欢把这比喻成“Scala的宜居之所”，告诉你怎么写Scala代码才不会像是从Java翻译过来的。之后我们会阐述项目问题，搞明白Scala是不是适用于你的项目。然后，我们看一些Scala在语法上的创新，这些创新让Scala代码变得即简洁又漂亮。接下来是Scala处理面向对象的方式，然后是一节介绍集合

和数据结构的内容。最后收尾的一节是关于Scala并发和它强大的actor模型的内容。

在讨论Scala的独特性时，我们会一并解释它的语法（以及其他必要的概念）。跟Java比，Scala是一门相当庞大的语言，需要掌握的基础概念和语法点更多。这就是说你要做好心理准备，随着接触到的Scala代码越来越多，你需要自己花时间和精力去更多地探索这门语言。

我们先来大体看一下后面会遇到的一些主题。这能帮你熟悉Scala不同的语法和思维方式，并帮你打下基础，以便更好地学习新知识。

9.1 走马观花 Scala

下面是我们准备展示的主要内容：

- Scala语言的精炼，包括类型推断的能力；
- match表达式，以及模式和case类等相关概念；
- Scala的并发，采用消息和actor机制，而不是像Java代码那样用老旧的锁机制。

这些不是Scala的全部内容，只掌握它们也不可能让你变成Scala开发高手。它们是用来吊你胃口的，只是给你几个具体示例表明Scala可能适用于哪些场合。要走得更远，就得做更深入的探索。你可以找些在线资源，也可以找本完整讲述Scala的书，比如Joshua Suereth的*Scala in Depth* (Manning, 2012)。

我们要解释的第一个特性，也是Scala跟Java最重要的差别，就是它语法上的精炼性，我们就直奔主题吧。

9.1.1 简约的 Scala

Scala是采用静态类型系统的编译型语言。也就是说Scala代码应该和Java代码一样详细。可Scala偏偏很精炼，它太精炼了，看起来简直和脚本语言一样。因此Scala开发人员更加快速和高效，写代码的速度几乎可以跟用动态语言编程媲美了。

我们来看一些非常简单的代码，了解一下Scala的构造方法和类。比如要写一个简单的现金流模型类。需要用户提供两项信息：现金流的额度和货币。用Scala应该这样写：

```
class CashFlow(amt : Double, curr : String) {
    def amount() = amt
    def currency() = curr
}
```

这个类只有四行（其中一行还是用来结束的右括号）。不管怎样，它有获取方法（但没有设置方法）作为参数，还有一个单例构造方法。跟Java比起来，这简直太划算了（就这么几行代码）。请看相应的Java代码：

```
public class CashFlow {
    private final double amt;
    private final String curr;

    public CashFlow(double amt, String curr) {
```

```

        this.amt = amt;
        this.curr = curr;
    }

    public double getAmt() {
        return amt;
    }

    public String getCurr() {
        return curr;
    }
}

```

跟Scala相比，Java代码中的重复信息太多了，就是这种重复导致了Java代码的冗长。

选择Scala，让开发人员尽量减少重复信息的输入，IDE的界面中就可以显示更多内容。面对稍微复杂点的逻辑时，开发人员就能见到更多代码，因此也有望能掌握理解它所需的更多线索。

要不要省1500美元？

CashFlow类的Scala版长度几乎比Java版短75%。据估计，一行代码每年的成本是32美元。如果我们假定这段代码的生命期是5年，那在这个项目的生命周期内，Scala版代码的维护成本就会比Java代码少花1500美元。

既然说到这儿了，我们就来看看第一个例子中展示的语法点。

- 类的定义（就它的参数而言）和类的构造方法是同一个东西。Scala中可以有其他的“辅助构造方法”，稍后就会谈到。
- 类默认是公开的，所以没必要加上public关键字。
- 方法的返回类型是通过类型推断确定的，但要在定义方法的def从句中用等号告诉编译器做类型推断。
- 如果方法体只是一条语句（或表达式），那就没必要用大括号括起来。
- Scala不像Java一样有原始类型。数字类型也是对象。

Scala的精炼不止体现在这些方面。甚至像HelloWorld这样简单的经典程序中都有所体现：

```

object HelloWorld {
    def main(args : Array[String]) {
        val hello = "Hello World!"

        println(hello)
    }
}

```

即便在这个最基本的例子中，也有几个帮我们去除套路化代码的特性。

- 关键字object告诉Scala编译器这个类是单例类。
- 调用println()没必要说明完整路径（感谢默认引入）。
- 没必要在main()方法前指明关键字public和static。
- 不必声明hello的类型，编译器会自己找出来。

- 不必声明main()的返回类型，编译器会自动设为Unit（等价于Java中的void）。这个例子中还有些相关语法需要注意一下。
- 跟Java和Groovy不一样，变量的类型在变量名之后。
- Scala用方括号来表示泛型，所以类型参数的表示方法是Array[String]，而不是String[]。
- Array是纯正的泛型。
- 集合类型必须指明泛型（不能像Java那样声明生类型^①）。
- 分号绝对是可选的。
- val就相当于Java中的final变量，用于声明一个不可变变量。
- Scala应用程序的初始入口总是在object中。

在后续几节中，我们会详细解释这些语法是如何工作的，并且我们还会再选几个让你更省手指头的Scala创新介绍一下。我们也会讨论Scala的函数式编程，它对于编写精炼的代码非常有帮助。现在，我们先来讨论一个强大的Scala“本地”特性。

9.1.2 match 表达式

Scala有一种非常强大的结构：match表达式。最简单的match用法跟Java的switch差不多，但match的表达力要强得多。match表达式的形式取决于case从句中的表达式结构。Scala调用不同类型的case从句模式，但要注意，这些所谓的模式跟正则表达式里的“模式”是截然不同的（尽管在match表达式里也可以用正则表达式模式）。

先看一个熟悉的例子。1.3.1节那个带字符串的switch被翻译成了Scala代码，请看：

```
var frenchDayOfWeek = args(0) match {
    case "Sunday"      => "Dimanche"
    case "Monday"       => "Lundi"
    case "Tuesday"      => "Mardi"
    case "Wednesday"   => "Mercredi"
    case "Thursday"     => "Jeudi"
    case "Friday"       => "Vendredi"
    case "Saturday"     => "Samedi"
    case _              => "Error: '" + args(0) + "' is not a day of the week"
}
println(frenchDayOfWeek)
```

9

我们在这个例子中只用到了两种最基本的模式：用来确定是周几的常量模式和处理默认情况的_模式，后面我们还会遇到其他模式。

从语言的纯粹性来看，可以说Scala的语法比Java更清晰，也更正规，至少从下面这两点来看是这样的：

^① 生类型（raw type）是指不带类型参数的泛型类或接口。比如泛型类Box<T>，创建它的参数化类型时要指明类型参数的真实类型：Box<Integer> intBox = new Box<>();。如果忽略了类型参数，Box rawBox = new Box();则是创建了一个生类型。——译者注

- 默认case不需要另外一个不同的关键字；
 - 单个case不会像Java中那样进入下一个case，所以也不需要break。
- 这个例子中的其他语法点如下所示。
- 关键字var用来声明一个可变（非final）变量。没有必要尽量不要用它，但有时候确实需要它。
 - 数组用圆括号访问，比如args(0)是指main()的第一个参数。
 - 总应该包括默认case。如果Scala在运行时在所有case中都找不到匹配项，就会抛出MatchError。这绝不是你想看到的。
 - Scala支持间接方法调用，所以可以把args(0).match({ ... })写成args(0) match { ... }。

到目前为止一切都好。match看起来就像稍微简洁些的switch。但这只是它众多模式中最像Java的。Scala中有大量使用不同模式的语言结构。比如说，有一种类型化模式，对于处理类型不确定的数据很有用，不用像Java那样弄一堆乱糟糟的类型转换或instanceof测试：

```
def storageSize(obj: Any) = obj match {
    case s: String => s.length
    case i: Int      => 4
    case _            => -1
}
```

这个极其简单的方法以一个Any类型（即未知类型）的值为参数，然后用模式分别处理String和Int类型的值。每个case都给要处理的值绑定了一个临时别名，以便必要时可以调用其中的方法。

在Scala的异常处理代码中有一个跟变量模式非常相似的语法形式。下面是一段改编自第11章ScalaTest框架的类加载代码：

```
def getReporter(repClassName: String, loader: ClassLoader): Reporter = {
    try {
        val reporterCl: java.lang.Class[_] = loader.loadClass(repClassName)
        reporterCl.newInstance.asInstanceOf[Reporter]
    }
    catch {
        case e: ClassNotFoundException => {
            val msg = "Can't load reporter class"
            val iae = new IllegalArgumentException(msg)
            iae.initCause(e)
            throw iae
        }
        case e: InstantiationException => {
            val msg = "Can't instantiate Reporter"
            val iae = new IllegalArgumentException(msg)
            iae.initCause(e)
            throw iae
        }
    ...
    }
}
```

在`getReporter()`中，要加载一个定制的`report`类（通过反射），以便在运行测试集时输出报告。在类加载和实例化过程中很多事都可能出错，所以要有个`try-catch`块来保护程序执行。

`catch`块起到的作用就跟在异常类型上放`match`表达式类似。`case`类的这种思路还可以进一步延伸，接下来我们就来讨论这个。

9.1.3 case 类

`match`表达式的最强用法之一就是跟`case`类（可以看成是枚举概念面向对象的扩展）相结合。我们来看一个温度过高发出报警信号的例子：

```
case class TemperatureAlarm(temp : Double)
```

单这一行代码就可以定义一个绝对有效的`case`类。在Java中相应的类大概应该是这样子：

```
public class TemperatureAlarm {
    private final double temp;
    public TemperatureAlarm(double temp) {
        this.temp = temp;
    }

    public double getTemp() {
        return temp;
    }

    @Override
    public String toString() {
        return "TemperatureAlarm [temp=" + temp + "]";
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        long temp;
        temp = Double.doubleToLongBits(this.temp);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        TemperatureAlarm other = (TemperatureAlarm) obj;
        if (Double.doubleToLongBits(temp) !=
            Double.doubleToLongBits(other.temp))
            return false;
        return true;
    }
}
```

只需加个case关键字就可以让Scala编译器生成这些额外的方法。它还会生成很多额外的架子方法。大多数情况下，开发人员都不会直接使用这些方法。它们是为某些Scala特性提供运行时支持的——能以“自然的Scala”方式使用case类。

创建case类实例不需要关键字new，像这样：

```
val alarm = TemperatureAlarm(99.9)
```

这进一步强化了case类是类似于“参数化枚举类型”或某种形式的值类型的观点。

Scala中的相等

Scala认为Java用==表示“引用相等”是个错误。所以在Scala中，==和>equals()是一样的。如果需要判断引用相等，可以用==。case类的.equals()方法只有在两个实例的所有参数值都一样时才会返回true。

case类跟构造器模式非常合，请看：

```
def ctorMatchExample(sthg : AnyRef) = {
    val msg = sthg match {
        case Heartbeat => 0
        case TemperatureAlarm(temp) => "Tripped at temp " + temp
        case _ => "No match"
    }
    println(msg)
}
```

我们去看看Scala观光之旅的最后一站：基于actor的并发结构。

9.1.4 actor

Scala选择用actor机制来实现并发编程。它们提供了一个异步并发模型，通过在代码单元间传递消息实现并发。很多开发人员都发现这种并发模型比Java提供的基于锁机制、默认共享的并发模型易用（不过Scala的底层模型也是JMM）。

来看个例子。假设我们在第4章遇到的兽医需要监控诊所里动物的健康状况（尤其是体温）。按我们的想法，温度感应器应该会将它们的读数消息发送给中心监控软件。

在Scala中，我们可以用一个actor类TemperatureMonitor对这种设置建模。应该有两种不同的消息：一种是标准的“心跳”消息，一种是TemperatureAlarm消息。第二种消息会带一个参数，表明那个警报器的温度超出了限值。代码清单9-1中列出了这些类的代码。

代码清单9-1 与actor的简单通信

```
case object Heartbeat
case class TemperatureAlarm(temp : Double)

import scala.actors._

class TemperatureMonitor extends Actor {
```

```

var tripped : Boolean = false
var tripTemp : Double = 0.0

def act() = {
    while (true) {
        receive {
            case Heartbeat => 0
            case TemperatureAlarm(temp) =>
                tripped = true
                tripTemp = temp
            case _ => println("No match")
        }
    }
}

```

监控actor会对三种不同的case做出响应（通过receive）。第一个是心跳消息，告诉你一切正常。因为这个case类没有参数，所以技术上来说它是一个单例实例，可以按case对象引用。actor在收到心跳消息时什么也不用做。

如果收到TemperatureAlarm消息，actor会保存警报器上的温度值。你应该想象得出，兽医有另外的代码定期检查TemperatureMonitor actor，看有没有警报被触发。

最后还有个default case。这是为了确保有任何不期而至的消息溜进actor环境时能被捕获到。如果没有这个一切全包的case，actor如果看到不认识的消息类型就会抛出异常。我们在本章的最后还会再次讨论actor的更多细节，但Scala的并发是个非常大的主题，而且在这本书里我们也不想让你浅尝辄止。

我们快速浏览了Scala的一些亮点。希望其中的某些特性已经燃起了你的兴趣之火。在下一节，我们会花点时间聊聊你可能会（也可能不会）在自己的项目中选择使用Scala的原因。

9.2 Scala 能用在我的项目中吗

9

在Java项目里增加一门语言总该有正当的理由和根据。在这一节，我们希望你想想那些理由，以及如何把它们应用到你的项目中。

一开始我们会快速比较一下Scala跟Java，然后看看什么时候，以及如何开始使用Scala。为了让这一篇幅较短的章节更完整，我们会看一些示警信号，当出现这些信号时，Scala可能不是最适合你项目的语言。

9.2.1 Scala 和 Java 的比较

我们在表9-1中对这两种语言的主要差异做了汇总。语言的“表皮层”是指该语言关键字的数量和开发人员用它干活必须掌握的独立语言结构的数量。

这些差异是Scala得到Java开发人员青睐，可以把它当做某些项目或组件的备选语言的部分原因。接下来我们会给出把Scala引入项目的更多细节。

表9-1 比较Scala和Java

特 性	Java	Scala
类型系统	静态的、非常繁琐	静态的，但使用了大量类型推断
多语言金字塔层级	稳定	稳定、动态
并发模型	基于锁机制	基于actor机制
函数式编程	需要严格遵守的特殊编码方式，不自然	内置支持、语言的一部分（纯天然）
表皮层	小型/中型	大型/超大型
语法风格	简单、常规、比较繁琐	灵活、精炼、很多特殊情况

9.2.2 何时以及如何开始使用 Scala

我们在第7章讨论过，在已有项目中引入新语言时，最好从风险比较低的区域开始。ScalaTest测试框架（见第11章）就是这样的低风险区。如果Scala实验不顺利，那所有的成本就是开发人员浪费的时间（这些单元测试有可能变成普通的JUnit测试）。

一般来说，适合在项目中引入Scala的组件应该基本满足下面这些条件：

- 你有信心评估所需的工作量；
- 问题域边界明确，定义清晰；
- 需求说明正确；
- 与其他组件的互操作性需求已知；
- 确定了愿意学习新语言的开发人员。

经过深思熟虑选定合适区域后，你就可以开始实现自己的第一个Scala组件了。下面有些指导原则，事实证明它们能让初始组件按部就班地进行：

- 以快速扣杀开始；
- 尽早跟已有的Java组件测试交互操作；
- 有定义扣杀成功或失败的入门标准（基于需求）；
- 如果扣杀失败，要有B计划；
- 在预算中为新组件留出额外的重构时间（用新语言编写的第一个项目欠下的技术债几乎肯定会比用团队已经熟悉的语言编写来得高）。

在评估Scala时，另外一个应该考虑的是检查那些明显让Scala对项目来说不太理想的迹象。

9.2.3 Scala 可能不适合当前项目的迹象

下面这些迹象表明Scala可能并不适合你的项目。如果出现了其中一个或多个迹象，你应该慎重考虑引入Scala的时机是否恰当。如果超过两个，那基本就没什么戏了。

- 受到了业务小组和其他程序支持小组的抵制，或缺乏动力。
- 开发团队没有明显的学习Scala的动力。
- 小组中分帮结派或政治上存在巨大分歧。

- 小组中高级技术人员的支持力度不够。
- 截止日期太紧张（没时间学习新语言）。

另外一个要密切关注的因素是，团队是否分散在全球各地。如果你用来开发（或支持）Scala代码的员工分散在几个地方，那会增加Scala培训人员的成本和负担。

现在我们已经讨论过把Scala引入项目的机制了，接下来该去看看Scala的语法了。我们会重点关注让Java开发人员更轻松的特性，鼓励更加紧凑的代码，少点套路化和挥之不去的繁琐。

9.3 让代码因 Scala 重新绽放

我们在这一节会先介绍一下Scala编译器和交互环境（REPL）。然后讨论类型推断，接着是方法声明（跟你所熟悉的Java方式不太一样）。这两个特性能帮你减少大量的套路化代码，从而提高生产力。

我们会谈到Scala的代码封装方式和更强大的import语句，然后详细讲解一下Scala中的循环和控制结构。这些特性植根于跟Java差异巨大的编程传统，所以我们会借此机会讨论一下Scala的函数式编程，包括函数式的循环结构、match表达式和函数字面值。

看过这些之后，本章剩下的大部分内容对你来说都没什么问题了，你可以自信地说自己有能力成为一名Scala程序员了。来吧，现在我们就开始讨论编译器和内置的交互环境。

9.3.1 使用编译器和 REPL

Scala是编译型语言，所以执行Scala程序通常要把它们先编译成.class文件，然后在类路径上有scala-library.jar（Scala运行时类库）的JVM环境中执行。

如果你还没装Scala，请在继续阅读之前参见附录C，了解如何安装Scala。样例程序（9.1.1中的HelloWorld）可以用scalac HelloWorld.scala编译（如果你正好在HelloWorld.scala文件所在的目录中）。

一旦得到.class文件，就可以用命令scala HelloWorld执行它了。这个命令会启动带着Scala运行时环境的JVM，然后进入类文件指定的main方法。

除了编译和运行，Scala还有个内置的交互环境，有点像第8章讲的Groovy控制台。但不像Groovy，Scala是在命令行环境里实现的。这就是说在典型的Unix/Linux环境（Path设置正确）中，你可以敲入scala，它就会在终端窗口内打开，而不会再弹出一个新窗口。

9

注意 这类交互环境有时被称为读入—计算—输出（Read-Eval-Print）循环，或简称为REPL。

这在动态语言中很常见。在REPL环境中，前面输入的那些行的计算结果还在，在后面的表达式和计算中还可以用。在本章的剩余部分，我们偶尔会用REPL环境来演示Scala语法。

现在我们开始讨论下一个大特性：Scala的高级类型推断。

9.3.2 类型推断

在读前面的代码时你可能已经注意到了，我们在声明变量hello为val时，没有指明它是什么类型。因为它很“明显”是个字符串。表面上来看这有点像Groovy，变量没有类型（Groovy是动态类型语言），但其实Scala代码中所发生的事情完全不同。

Scala是静态类型语言（所以变量确实有明确的类型），但它的编译器能分析源码，并且一般都能根据上下文推断出应该是什么类型。如果Scala自己能确定是什么类型，就不用你亲自告诉它了。

这就是类型推断，我们已经提过好几次了。Scala在这方面的能力非常突出—以致于开发人员经常在行云流水一样的代码中忘记静态类型。这经常让Scala更有动态语言的“感觉”。

Java中的类型推断

Java也有类型推断的能力，虽然有限，但确实有。最明显的例子就是我们在第1章见到的泛型钻石语法。Java的类型推断通常是在赋值语句等号右边的值上。Scala通常是推断变量而不是值的类型，但它的的确也能推断值的类型。

你已经见过其中最简单的例子了：关键字var和val，Scala根据赋给变量的值来推断它们的类型。Scala类型推断的另一个重要应用是方法声明。我们来看个例子（Scala的AnyRef就是Java中的Object）：

```
def len(obj : AnyRef) = {
    obj.toString.length
}
```

这是一个类型推断的方法。通过检查它返回代码中的java.lang.String#length的类型（int），编译器知道这个方法要返回Int类型的值。注意，这个方法没有显式指定返回类型，我们也不需要用return关键字。实际上，如果你放了一个显式的return在这里，像这样：

```
def len(obj : AnyRef) = {
    return obj.toString.length
}
```

会得到一个编译时错误：

```
error: method len has return statement; needs result type
      return obj.toString.length
      ^
```

如果你连def中的=也省略了，编译器会假定这个方法会返回Unit（就跟Java里返回void一样）。除了前面那些限制，还有两个类型推断受限的区域：

- 方法声明中参数的类型——传给方法的参数必须指定类型；
- 递归函数——Scala编译器不能推断递归函数的返回类型。

关于Scala的方法，我们讨论的东西已经不少了，但还算不上系统化的讨论，所以我们来巩固一下你已经学过的东西。

9.3.3 方法

你已经见过怎么用`def`关键字定义方法了。随着你对Scala越来越熟悉，关于Scala的方法，还有些你应该知道的重要事实。

- Scala没有`static`关键字。跟Java中的`static`方法对应的方法必须放在Scala的`object`(单例)结构中。稍后我们会向你介绍相关概念：伴生对象。
- 跟Groovy(或Clojure)相比，Scala语言的运行时要重得多。Scala类中可能会有很多由平台自动生成的额外方法。
- 方法调用是Scala的核心概念。在Scala中没有Java中那种意义的操作符。
- 对于哪些字符可以出现在方法的名称中，Scala比Java更灵活。特别是那些在其他语言中作为操作符的字符，在Scala中可能是合法的方法名(比如加号`+`)。

间接方法调用(前面讲过)中有Scala把方法调用和操作符合并到一起的线索。举个例子，比如要把两个整型相加。在Java中，应该是写一个`a+b`这样的表达式。在Scala中你也可以这样写，但不止这样，还可以写成`a.+(b)`。换句话说，你调用了`a`上的`+()`方法，并把`b`作为参数传给它。这就是Scala不再把操作符当做一个独立概念的秘密。

注意 你可能已经注意到了，`a.+ (b)`是在`a`上调用方法。但原始类型的变量`a`怎么会有方法呢？9.4节会给出完整的解释。但现在，你只要知道Scala的类型系统认为所有东西都是对象，所以你可以在任何东西上调用方法，即便是Java里的原始类型变量也行。

你已经见过一个用`def`关键字声明方法的例子了。我们再来看一个例子，一个实现阶乘函数的简单递归方法：

```
def fact(base : Int) : Int = {
    if (base <= 0)
        return 1
    else
        return base * fact(base - 1)
}
```

对于所有负数，这个函数都返回1，这算是作弊吧。实际上，负数的阶乘是不存在的，但大家都是朋友嘛。它看起来有点像Java：有返回类型(`Int`)，并用`return`关键字表明把哪个值交回给调用者。唯一需要注意的就是在函数体代码块定义之前额外符号`=`。

Scala中还有另外一个Java中没有概念：局部函数。它是在另外一个函数内部(并且仅在这一作用域内有效)定义的函数。如果开发人员想要一个辅助函数，又不想把实现细节暴露给外部，这是一个简单的办法。在Java中除了用`private`方法之外别无选择，但这个函数对于同一类的其他方法都是可见的。但在Scala中，你只要这样写就行了：

```

def fact2(base : Int) : Int = {
    def factHelper(n : Int) : Int = {
        return fact2(n-1)
    }
    if (base <= 0)
        return 1
    else
        return base * factHelper(base)
}

```

`factHelper()`在`fact2()`的封闭作用域之外绝对是不可见的。

接下来，我们去看看Scala如何处理代码的组织和导入。

9.3.4 导入

Scala对包的使用跟Java一样，关键字也一样，分别是`package`和`import`。Scala可以毫无障碍地导入和使用Java的包和类。Scala的`var`或`val`变量可以引用任何Java类的实例，不需要任何特殊的语法或处理：

```

import java.io.File
import java.net._
import scala.collection.{Map, Seq}
import java.util.{Date => UDate}

```

头两行代码跟Java里的标准导入和通配符导入一样。第三行用一行导入一个包里的多个类。最后一行在导入时指定了类的别名（避免缩写冲突出现）。

跟Java不一样，Scala中的`import`可以出现在代码中的任何位置（不仅限于文件顶部），这样你就可以把`import`当做文件的一部分分离出来。Scala也有默认导入，即所有`.scala`文件默认都会导入`scala._`。这里有很多有用的函数，包括我们已经讨论过的一些，比如`println`。对于所有默认导入的完整细节，请参见`www.scala-lang.org`上的API文档。

我们接下来讨论怎么控制Scala程序的执行流。这可能和你熟悉的Java跟Groovy有些差异。

9.3.5 循环和控制结构

Scala在控制和循环结构上引入了几个有点绕的创新。在我们向你介绍这些不熟悉的形式之前，先来看几个老朋友，比如标准的`while`循环：

```

var counter = 1
while (counter <= 10) {
    println("." * counter)
    counter = counter + 1
}

```

还有`do-while`形式：

```

var counter = 1
do {
    println("." * counter)
    counter = counter + 1
} while (counter <= 10)

```

另一个是基本的for循环：

```
for (i <- 1 to 10) println(i)
```

看起来都很好。但Scala更灵活，比如条件for循环：

```
for (i <- 1 to 10; if i % 2 == 0) println(i)
```

还能在多个变量上循环，比如：

```
for (x <- 1 to 5; y <- 1 to x)
  println(" " * (x - y) + x.toString * y)
```

这些多出来的形式源于Scala实现这些结构的根本性差异。Scala用函数式编程中的概念（列表推导式）来实现for循环。

列表推导式的一般概念是对一个列表中的元素进行转换（或过滤，比如在用条件for循环时）。这会产生一个新列表，然后在其中的每个元素上逐次运行for循环体中的代码。

甚至把要过滤的列表和for代码块分开都是有可能的，用yield关键字。比如下面这段代码：

```
val xs = for (x <- 2 to 11) yield fact(x)
for (factx <- xs) println(factx)
```

这段代码先设置新集合xs，然后用第二个for循环逐一输出其中的值。如果你需要一个创建一次、使用多次的集合，这个极其好用。

这一结构能成立是因为Scala支持函数式编程，我们接下来就去看看Scala如何实现函数式思想。

9.3.6 Scala 的函数式编程

我们在7.5.2节提起过，Scala把函数当做内置的值。这就是说函数可以放进var或val中，并和其他任何值所受的对待毫无二致。这被称为函数字面值（或匿名函数），它们是Scala世界观的重要组成部分。

在Scala中写函数字面值非常简单。其中的关键是箭头=>，Scala用它来表示取得参数列表并传递到代码块中：

```
(<函数参数列表>) => { ... 作为代码块的函数体 ... }
```

我们用Scala的交互环境来演示一下。下面这个例子中定义的函数接受一个Int参数，然后乘以2：

```
scala> val doubler = (x : Int) => { 2 * x }
doubler: (Int) => Int = <function1>

scala> doubler(3)
res4: Int = 6

scala> doubler(4)
res5: Int = 8
```

注意看Scala怎么推断doubler的类型。它的类型是“接受一个Int并返回Int的函数”。这样

的类型用Java的类型系统还不能以令人完全满意的方式表示。你看，调用doubler就是用标准的调用语法。

我们把这个概念再向前推进一点。在Scala中，函数字面值只是值。并且是函数返回的值。这就是说你可以写一个生产函数的函数——接受一个值并返回一个新的函数字面值。

比如说，可以定义一个命名为adder的函数字面值。adder()能生产一个给它们的参数加上一个常量的函数：

```
scala> val adder = (n : Int) => { (x : Int) => x + n }
adder: (Int) => (Int) => Int = <function1>

scala> val plus2 = adder(2)
plus2: (Int) => Int = <function1>

scala> plus2(3)
res2: Int = 5

scala> plus2(4)
res3: Int = 6
```

看到了吧，Scala对函数字面值支持得很好。实际上，Scala代码一般都能用非常函数的世界观来编写，同时也能用更加命令式的风格编写。现在我们所做的不过是刚刚涉足Scala的函数式编程能力，但重要的是知道它们在那里。

在下一节中，我们会讨论Scala的对象模型和面向对象方式的细节。在一些重要方面，Scala的一些先进的特性使得它对面向对象的处理方式跟Java差异很大。

9.4 Scala 对象模型：相似但不同

Scala有时被称为“纯粹”的面向对象语言。也就是说所有的值都是对象，所以面向对象的概念几乎随处可见。本节一开始，我们会探索一下“一切皆对象”的后果。这个主题会很自然地引导我们去思考Scala的类型层级。

Scala的类型层级跟Java有几个重要差异，包括装箱和拆箱等Scala处理原始类型的方式。之后我们会考虑Scala的构造方法和类定义，以及它们怎么帮你少写代码。接着是关于trait（特质）的话题，然后再讨论Scala的单例、伴侣和包对象。本节最后我们会看一下怎么用case类再进一步减少套路化代码，并以一个有警示意义的Scala语法故事作为结尾。

让我们开始吧。

9.4.1 一切皆对象

Scala的观点是所有类型都是对象类型。包括Java所谓的原始类型。图9-1展示了Scala的类型继承关系，包括所有值类型（即原始类型）和引用类型，并标注了与Java中类型的对应关系。

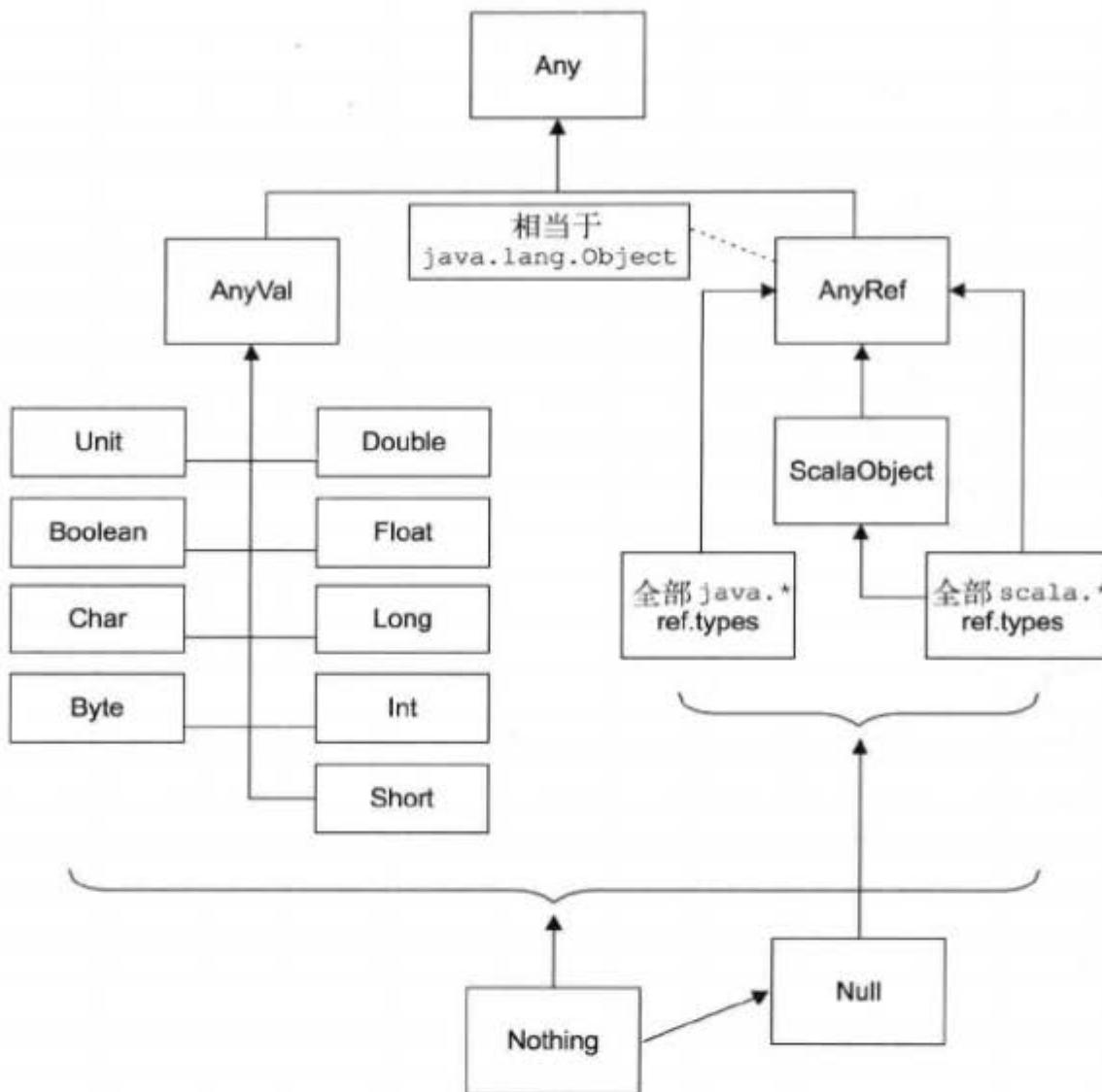


图9-1 Scala中的继承层级

从图中可以看到，Unit和其他值类型在Scala中都是正确的类型。AnyRef类相当于java.lang.Object。每次见到AnyRef，你都应该在心里把它换成Object。它之所以没叫Object，是因为Scala也要运行在.NET运行时平台上，所以它要给这个概念再起个名字。

Scala用extends关键字表示类的继承关系，而且它的用法跟Java很像：所有的非私有成员都会被继承下来，两种类型之间也会建立起父类/子类的关系。如果类定义中没有显式扩展其他类，则编译器会认定它是AnyRef的直接子类。

“一切皆对象”的原则可以解释使用中缀符号的方法调用。9.3.3节中的`obj.meth(param)`和`obj meth param`是方法调用的两种方式，其含义是一样的。现在你应该明白了，Java中的表达式`1+2`是数值原始类型和加法操作符的表达式，而Scala中与之对应的`1.+(2)`是`Scala.Int`类上的方法调用。

Scala中没有因数值的装箱操作而引起的困扰，而这在Java里很常见。请看下面的Java代码：

```

Integer one = new Integer(1);
Integer uno = new Integer(1);
System.out.println(one == uno);
    
```

你可能觉得很奇怪，这段代码的输出结果居然是`false`。而Scala中对数值装箱及相等判断的

方式符合我们的常识，这有以下几个好处。

- 数值类不能由构造方法实例化。它们是有效的abstract和final类（Java中不允许这种组合）。
- 得到数值类实例的唯一办法就是作为字面值。这能确保2总是同一个2。
- 判断两个值是否相等所用的==方法的定义和equals()一样，不是引用相等。
- ==不能重写，但equals()可以。
- 对于引用相等的判断，Scala中有eq方法。但一般不太会用到它。

现在我们已经讨论了Scala中一些最基本的面向对象概念，还需要再多介绍一点儿Scala的语法。最简单的就是Scala的构造方法。

9.4.2 构造方法

Scala的类必须有个主构造方法来定义该类所需的参数。此外，类还可以有额外的辅助构造方法。这些辅助构造方法都用this()表示，但它们比Java的重载构造方法限制更严格。

Scala辅助构造方法的第一条语句必须调用同一个类中的另一个构造方法（或者是主构造方法，或者是另一个辅助构造方法）。这种限制是为了把控制流引导到主构造方法上，因为它是类的唯一真正入口。也就是说，辅助构造方法的真实作用是为主构造方法提供默认参数。

请看CashFlow上的这些辅助构造方法：

```
class CashFlow(amt : Double, curr : String) {
  def this(amt : Double) = this(amt, "GBP")
  def this(curr : String) = this(0, curr)

  def amount = amt
  def currency = curr
}
```

这个例子中有个辅助函数可以只给出金额，CashFlow会假定货币是英镑。另一个辅助函数可以只给出货币，假定金额为0。

注意我们定义的amount和currency方法，都没有括号或参数列表（甚至连空的都没有）。这是告诉编译器，在调用这个类的amount和currency方法时不需要括号，像这样：

```
val wages = new CashFlow(2000.0)
println(wages.amount)
println(wages.currency)
```

Scala对类的定义基本都能对应到Java中。但在面向对象的继承方式上，Scala所采用的方式跟Java有显著的差异。下一节就来讨论它们的差异。

9.4.3 特质

特质是Scala面向对象编程方式的主要组成部分。广义上来说，它们和Java接口一样。但跟Java接口不同的是，特质中可以给出方法的实现，并且这些实现可以由具备该特质的不同类共享。

要理解它所解决的Java问题，请看图9-2中从不同的基础类继承而来的两个Java类。如果这两

个类都要具备额外的相同功能，Java中的做法是声明它们实现了相同的接口。

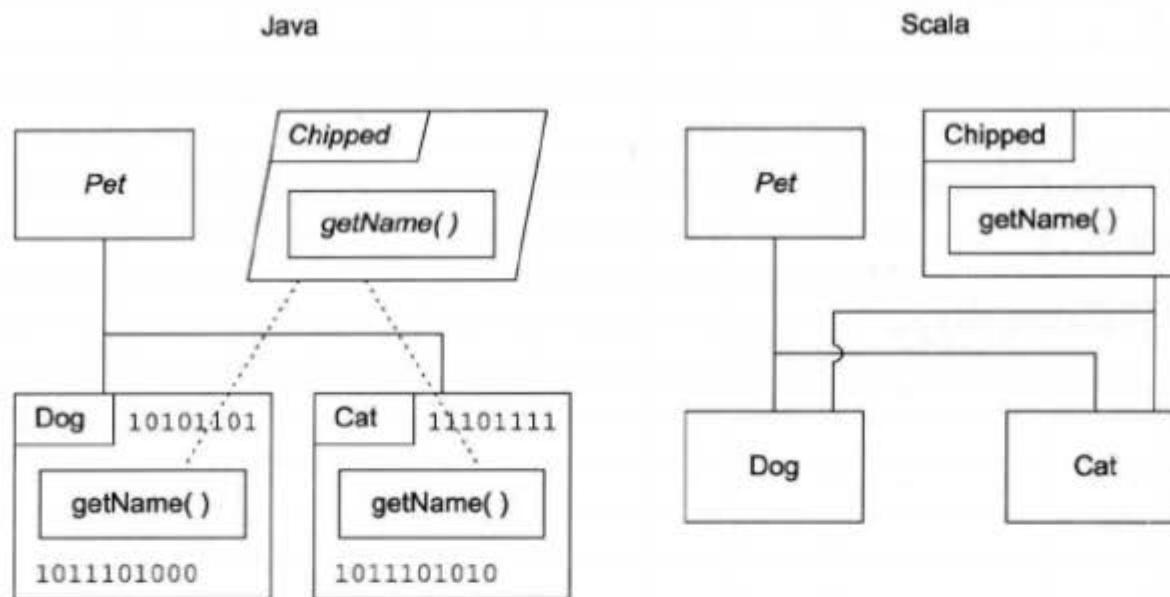


图9-2 Java模型中的实现复制

代码清单9-2是一个简单的Java例子，就是上面这种情况的代码。回忆一下4.3.6节那个兽医诊所的例子。很多带到诊所的动物都会被植入芯片，以便于识别。比如猫和狗几乎肯定会这么处理，但其他物种可能不会。

植入芯片的功能需要提取到单独的接口中。我们来修改一下代码清单4-11中的Java代码，加入这一功能（为了让代码看起来更清晰，我们省略了examine()方法）。

代码清单9-2 说明实现代码的复制

```

9
public abstract class Pet {
    protected final String name;
    public Pet(String name_) {
        name = name_;
    }
}

public interface Chipped {
    String getName();
}

public class Cat extends Pet implements Chipped {
    public Cat(String name_) {
        super(name_);
    }
    public String getName() {
        return name;
    }
}

public class Dog extends Pet implements Chipped {
    public Dog(String name_) {
        super(name_);
    }
}
  
```

```

    }

    public String getName() {
        return name;
    }
}

```

Dog和Cat中都有同样的getName()代码，因为Java接口中不能有实现代码。代码清单9-3是Scala用特质实现的版本。

代码清单9-3 用Scala实现的宠物类

```

class Pet(name : String)

trait Chipped {
    var chipName : String
    def getName = chipName
}

class Cat(name : String) extends Pet(name : String) with Chipped {
    var chipName = name
}

class Dog(name : String) extends Pet(name : String) with Chipped {
    var chipName = name
}

```

Scala要求在子类中必须给父类构造方法中出现的参数赋值。但在特质中声明的方法都会被子类继承。这样就减少了重复实现。你看，Cat和Dog类都要给参数name赋值。两个子类都可以访问Chipped中的实现——在此例中，参数chipName可以用来保存写在芯片上的宠物的名字。

9.4.4 单例和伴生对象

我们来看看Scala中的单例对象（即用关键字object定义的类）是如何实现的。回想一下9.1.1中的HelloWorld：

```

object HelloWorld {
    def main(args : Array[String]) {
        val hello = "Hello World!"
        println(hello)
    }
}

```

如果这是Java，你会觉得这段代码应该变成一个HelloWorld.class文件。实际上，Scala会把它编译成两个文件：HelloWorld.class和HelloWorld\$.class。

因为这就是普通的类文件，所以你可以用第5章介绍的反编译工具javap看看Scala编译器产生的字节码。这会让你对Scala的类型模型及其实现方式有更多的了解。代码清单9-4是对这两个文件运行javap -c -p产生的结果：

代码清单9-4 反编译Scala的单例对象

```

Compiled from "HelloWorld.scala"
public final class HelloWorld extends java.lang.Object {
    public static final void main(java.lang.String[]);
        Code:
            0: getstatic      #11
            1: ldc           #12
            2: invokevirtual #13
            3: return
    }

    // Field HelloWorld$.MODULE$:LHelloWorld$;
    4: aload_0
    5: invokevirtual #13
    6: return
}

Compiled from "HelloWorld.scala"
public final class HelloWorld$ extends java.lang.Object
    implements scala.ScalaObject {
    public static final HelloWorld$ MODULE$;
        Code:
            0: new          #9   // class HelloWorld$
            1: invokespecial #12  // Method "<init>":()V
            2: return

    public void main(java.lang.String[]);
        Code:
            0: getstatic      #19  // Field scala/Predef$.MODULE$:Lscala/Predef$;
            1: ldc           #22
            2: invokevirtual #26
            3: ldc           #22
            4: invokevirtual #26
            5: ldc           #22
            6: invokevirtual #26
            7: ldc           #22
            8: return

    private HelloWorld$();
        Code:
            0: aload_0
            1: invokespecial #33  // Method java/lang/Object."<init>":()V
            2: ldc           #22
            3: putstatic      #35  // Field MODULE$:LHelloWorld$;
            4: ldc           #22
            5: invokevirtual #35
            6: ldc           #22
            7: invokevirtual #35
            8: return
}

```

取得单例伴生模块

调用伴生的main()方法

单例伴生实例

私有构造方法

9

明白“Scala没有静态方法或域”这话是从何而来的了吗？除了这些结构，Scala编译器还自动生成了单例模式代码（不可变静态实例和私有构造方法），并把它们插到以\$结尾的类中。`main()`方法仍然是常规的实例方法，但是是在单例的`HelloWorld$`类实例上调用的。

这意味着在这一对.class文件之间有二元性：一个和Scala文件的名字相同，另外一个加了个\$。静态方法和域被放在了第二个单例类中。

Scala中名字相同的class和object非常常见。在这种情况下，单例类被当做了伴生对象。Scala源文件和两个VM类（主类和伴生对象）之间的关系如图9-3所示。

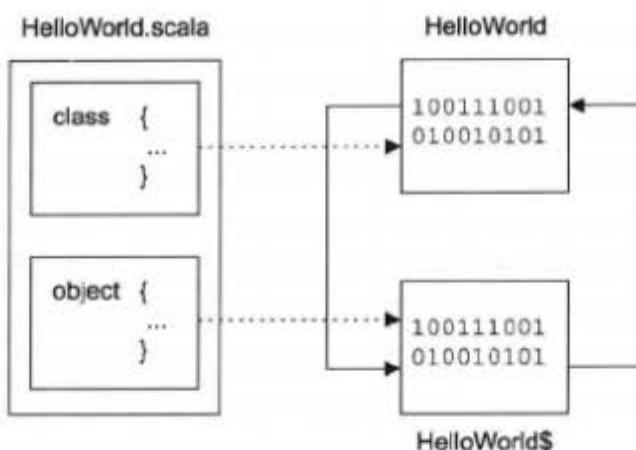


图9-3 Scala单例对象

尽管你不知道，但你确实已经遇到过伴生对象了。在HelloWorld中，你没必要指定println()方法在哪个类中。它看起来像个静态方法，所以你应该能想到它是伴生对象中的方法。

让我们再看一下代码清单9-2中与main()方法对应的字节码：

```

public void main(java.lang.String[]);
Code:
  0: getstatic      #19   // Field scala/Predef$.MODULE$:Lscala/Predef$;
  3: ldc            #22   // String Hello World!
  5: invokevirtual #26
-> // Method scala/Predef$.println:(Ljava/lang/Object;)V
  8: return

```

这段代码中的println()及其他随时可用的Scala函数都在Scala.Predef类的伴生对象中。

伴生对象在其相关类那里有特权。它能访问该类的私有方法。这使得Scala能以合理的方式定义私有辅助构造方法。Scala定义私有构造方法的语法是在其参数列表之前加上关键字private，像这样：

```

class CashFlow private (amt : Double, curr : String) {
    ...
}

```

如果私有的构造方法是主方法，那就只有两种办法可以创建该类的实例：或者通过伴生对象里的工厂方法（可以访问私有构造方法），或者调用一个公开的辅助构造方法。

接下来我们要进入下一主题：Scala的case类。你已经遇到过了，但为了刷新一下你的记忆，我们再重复一次，它们是通过自动提供一些基本方法来减少套路化代码的有效办法。

9.4.5 case类和match表达式

我们用Java实现一个简单的实体，比如Point类，如代码清单9-5所示。

代码清单9-5 一个用Java实现的简单类

```

public class Point {
    private final int x;
    private final int y;
}

```

```

public Point(int x, int y) {
    this.x = x;
    this.y = y;
}

public String toString() {
    return "Point(x: " + x + ", y: " + y + ")";
}

@Override
public boolean equals(Object obj) {
    if (!(obj instanceof Point)) {
        return false;
    }
    Point other = (Point) obj;
    return other.x == x && other.y == y;
}

@Override
public int hashCode() {
    return x * 17 + y;
}
}

```

The diagram shows two annotations: `@Override` and `@ToString`. Arrows point from each annotation to its respective code block below it. The first arrow points from the `@Override` annotation above the `equals` method to the start of that method. The second arrow points from the `@ToString` annotation above the `toString` method to the start of that method.

这套路化代码简直太多了，而且更糟的是，像`hashCode()`、`toString()`、`equals()`以及所有的获取方法通常都是由IDE自动生成的。如果在语言内核的内部完成这些自动生成的工作，用更简单的语法岂不是更好？

Scala的确支持自动生成，`case`类就可以。代码清单9-5可以非常简单：

```
case class Point(x : Int, y : Int)
```

这和Java那段长长的代码功能一样，但除了更短，它还有别的好处。

比如说，用Java那个版本，如果要修改代码（假设要加个z坐标），就必须更新`toString()`和其他方法。实际上，应该要把原来那些方法全部删掉，然后让IDE再重新生成一次。

用Scala这些都没必要，因为根本就没显式定义需要跟着更新的方法。这归结为一个非常强的理论：不可能在没出现的源码中弄出bug来。

在创建新的`case`类实例时，关键字`new`可以省略。代码可以写成这样：

```
val pythag = Point(3, 4)
```

这样看来`case`类更像带一个或多个参数的枚举类型了。实际上`case`类的底层实现机制是提供一个创建新实例的工厂方法。

我们来看一下`case`类的主要用途：模式和`match`表达式。`case`类可以用在叫做构造器（Constructor）模式的Scala模式类型里，请看代码清单9-6。

代码清单9-6 `match`表达式中的Constructor模式

```
val xaxis = Point(2, 0)
val yaxis = Point(0, 3)
val some = Point(5, 12)
```

```

val whereami = (p : Point) => p match {
  case Point(x, 0) => "On the x-axis"
  case Point(0, y) => "On the y-axis"
  case _              => "Out in the plane"
}
println(whereami(xaxis))
println(whereami(yaxis))
println(whereami(some))

```

我们在9.6节讨论actor和Scala的并发观点时会再次拜访Constructor模式和case类。

在结束本节之前，我们要发出一个警告。Scala丰富的语法和聪明的解析器能够用一些非常精炼和优雅的办法来表示复杂的代码。但Scala没有正式的语言规范，并且新特性的增加非常频繁。你应该多加小心——即便是经验丰富的Scala码农有时也会被语言特性出其不意的表现吓到。在语法特性互相结合时尤其如此。

我们来看一个例子：一种在Scala中模拟操作符重载的办法。

9.4.6 警世寓言

我们再想一想刚刚提到的Point case类。你可能想要用一种简单的办法来表示坐标的相加，或者坐标的线性增长。如果你数学好，可能马上就会意识到这是一个平面坐标上的向量空间属性。

代码清单9-7将方法定义得像普通操作符一样。

代码清单9-7 模拟操作符重载

```

case class Point(x : Int, y : Int) {
  def *(m : Int) = Point(this.x * m, this.y * m)
  def +(other : Point) = Point(this.x + other.x, this.y + other.y)
}

var poin = Point(2, 3)
var poin2 = Point(5, 7)
println(poin)
println(poin * 2)
println(poin * 2)
println(poin + poin2)

```

运行这段代码得到的输出应该是：

```

Point(2,3)
Point(5,7)
Point(4,6)
Point(7,10)

```

这下应该能看出Scala的case类跟Java里的等价物相比有多好了吧。只需要很少的代码，就能创造出一个很友好的类，产生合理的输出。定义+和*方法后，你已经可以模拟操作符重载了。但这种方式有问题。请看下面这段代码：

```

var poin = Point(2, 3)
println(2 * poin)

```

这会导致编译错误：

```

error: overloaded method value * with alternatives:
  (Double)Double <and>
  (Float)Float <and>
  (Long)Long <and>
  (Int)Int <and>
  (Char)Int <and>
  (Short)Int <and>
  (Byte)Int
cannot be applied to (Point)
      println(2 * poin)
          ^
one error found

```

尽管在case类Point上已经定义了方法`*(m : Int)`，但不是Scala要找的那个方法，所以出错了。为了让前面的代码编译成功，需要在Int类上实现`*(p : Point)`方法。这是不可能的，所以操作符重载只是一个假象。

这带出了Scala中有一个有趣的问题：很多语法特性的限制在某些情况下可能会让人大吃一惊。Scala的语言分析器和运行时环境在底层做了大量工作，但这些隐藏的机制是建立在尽量做正确的事的基础上的。

我们对Scala面向对象实现方式的介绍到这里就结束了。还有很多先进特性没涉及。很多现代化的类型系统和对象思想在Scala中都有实现，所以如果感兴趣，Scala的广阔天地对你来说大有可为。如果前面的那些内容勾起了你对Scala的类型系统和面向对象实现方式的兴趣，你可以去读一读Joshua Suereth的*Scala in Depth* (Manning, 2012)，或其他专门介绍Scala的图书。

你可能已经想到了，这些语言理论应用的一个重点是Scala的数据结构和集合，这也是我们下一节的主要内容。

9.5 数据结构和集合

9

你已经见过一个简单的Scala数据结构List了。它在任何编程语言中都是一个基本的数据结构，在Scala中也不例外。我们会花点时间探究一下List的细节，然后去研究一下Map。

接着我们会认真研究一下Scala中的泛型，包括与Java泛型的差别及Java泛型所不具备的能力。我们会以一些标准的Scala集合为例展开讨论，以便让你了解其原理。

先从Scala集合的几个一般性原则开始，特别是跟它的不可变性和它与Java集合的交互性相关的原则。

9.5.1 List

Scala中集合的实现方式跟Java很不一样。你可能会有点吃惊，因为在很多其他领域，Scala都在重用和扩展Java的组件、概念。

我们来看看Scala的理念所带来的最大差异：

- Scala集合通常都是不可变的；
- Scala把跟列表类似的集合的方方面面分解成了不同的概念；

- Scala构建List核心所涉及的概念非常少；
- Scala集合的实现方式是不同类型的集合提供的用户体验是一致的；
- Scala鼓励开发人员构建自己的集合类，并让它们用起来像内置的集合类一样。我们会逐一讨论这些差异。

1. 不可变和可变集合

你首先要知道，Scala的集合既有不可变的版本，也有可变的版本，并且不可变版本是默认的（所有Scala源文件都可以随时访问）。

我们需要分辨可变集合和可变内容之间的本质区别。请看代码清单9-8。

代码清单9-8 可变和不可变

```
import scala.collection.mutable.LinkedList
import scala.collection.JavaConversions._
import java.util.ArrayList

object ListExamples {
    def main(args : Array[String]) {
        var list = List(1,2,3)
        list = list :+ 4
        println(list)

        val linklist = LinkedList(1,2,3)
        linklist.append(LinkedList(4))
        println(linklist)

        val jlist = new ArrayList[String]()
        jlist.add("foo")
        val slist = jlist.toList
        println(slist)
    }
}
```



列表追加方法

如上所示，list的引用是可变的（是var）。它指向一个不可变列表实例，所以可以通过重新赋值指向新对象。:+方法返回一个新的（不可变）List实例，这个新实例中含有新追加的元素。

相反，linklist是指向一个LinkedList的不可变引用（是val），而LinkedList实例是不可变的。linklist的内容可以修改，比如在其上调用append()。这种区别如图9-4所示。

代码清单9-8中还演示了一组转换函数：用来对Java集合和相应的Scala集合进行相互转换的JavaConversions类。

2. List的特质

Scala选择强调集合的特质和行为，这是它与众不同的另一个重要之处。我们以Java的ArrayList为例。除了Object，这个类还直接或间接地扩展了：

- java.util.AbstractList;
- java.util.AbstractCollection。

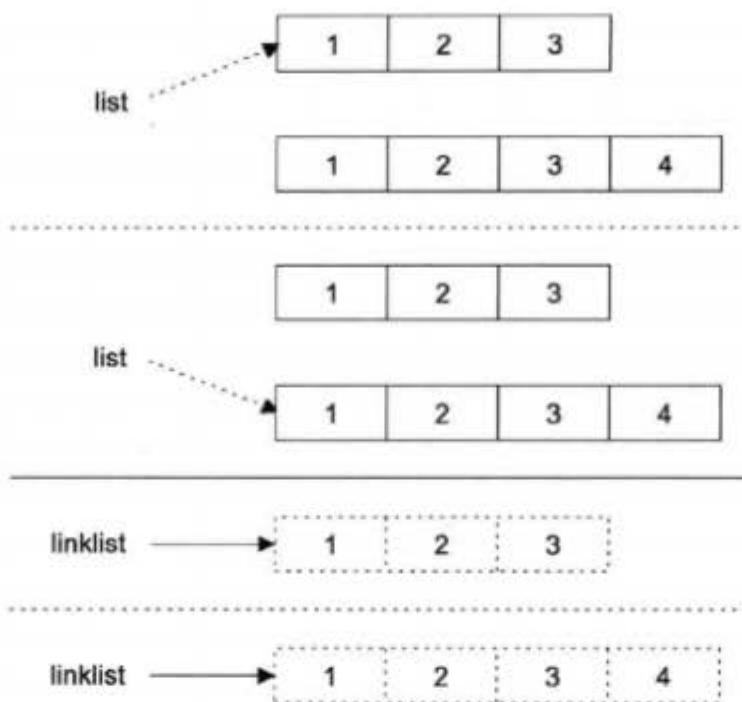


图9-4 不可变和可变集合

还有接口，`ArrayList`或它的某个父类实现了表9-2中列出的接口。

表9-2 `ArrayList`实现的Java接口

<code>Serializable</code>	<code>Cloneable</code>	<code>Iterable</code>
<code>Collection</code>	<code>List</code>	<code>RandomAccess</code>

对于Scala，情况要稍微复杂一点。以`LinkedList`为例，与它提供的功能相关的类或特质多达27个，如表9-3所示。

表9-3 `LinkedList`实现的Scala接口

<code>Serializable</code>	<code>LinkedListLike</code>	<code>LinearSeq</code>
<code>LinearSeqLike</code>	<code>Cloneable</code>	<code>Seq</code>
<code>SeqLike</code>	<code>GenSeq</code>	<code>GenSeqLike</code>
<code>PartialFunction</code>	<code>Function1</code>	<code>Iterable</code>
<code>IterableLike</code>	<code>Equals</code>	<code>GenIterable</code>
<code>GenIterableLike</code>	<code>Mutable</code>	<code>Traversable</code>
<code>GenTraversable</code>	<code>GenTraversableTemplate</code>	<code>TraversableLike</code>
<code>GenTraversableLike</code>	<code>Parallelizable</code>	<code>TraversableOnce</code>

Scala的集合类彼此之间的差异并不像Java那么明显。在Java中，`List`、`Map`、`Set`等，根据使用时的具体类型会有不同的处理模式。但在Scala中，由于使用了特质，类型的细化程度要比Java高得多。因此你可以把注意力放在集合的各种性质上，使用更加贴近需求的类型精确表达你的意图。

因此，Scala的集合处理代码要比Java的看起来更加整齐。

Scala中的set

如你所料，Scala既支持不可变的set，也支持可变set。set的典型用法跟Java里的模式一样：用一个中间对象按顺序遍历集合中的元素。但Java用的是Iterator或Iterable，而Scala用Traversable，它跟Java类型之间不能互操作。

构建列表的两个基础是：Nil表示空列表，::操作符能从已有的列表构建新列表。::操作符的发音是cons，它和Clojure的(concat)函数（见第10章）还有关系。这两者都表明Scala植根于函数式编程——最终可以追溯到Lisp中。

cons操作符有两个参数：一个类型为T的元素和一个类型为List[T]的对象。它会把两个参数合到一起创建一个新的List[T]值：

```
scala> val x = 2 :: 3 :: Nil
x: List[Int] = List(2, 3)
```

另外，也可以直接这样写：

```
scala> val x = List(2, 3)
```

```
x: List[Int] = List(2, 3)
```

```
scala> 1 :: x
res0: List[Int] = List(1, 2, 3)
```

cons操作符和括号

按cons操作符的定义，A :: B :: C的含义是没有歧义的，它的意思是A :: (B :: C)。这是因为::的第一个参数是单个类型为T的值。但A :: B是类型为List[T]的值，所以(A :: B) :: C作为可能的值没有任何意义。学院派的计算机科学家会说::是右相关的。

这也解释了为什么要写成2 :: 3 :: Nil，而2 :: 3不行。::的第二个参数需要是List类型的值，而3不是List。

9.5.2 Map

映射也是一种经典的数据结构。Java最常见的就是它的HashMap。在Scala中，不可变的Map类是默认形态，而HashMap是标准的可变形态。

代码清单9-9中有几种简单、标准的映射定义和操作。

代码清单9-9 Scala中的Map

```
import scala.collection.mutable.HashMap

var x = Map(1 -> "hi", 2 -> "There")
for ((key, val) <- x) println(key + ": " + val)
x = x + (3 -> "bye")

val hm = HashMap(1 -> "hi", 2 -> "There")
hm += (3 -> "bye")
println(hm)
```

看到了吧，Scala定义映射字面值的语法简洁可爱：`Map(1 -> "hi", 2 -> "There")`。用箭头符号直观地表明了每个键“指向”的值。要从映射中取回值，请用`get()`方法，跟Java一样。

可变和不可变映射都用`+=`表示向映射中添加元素（`-`表示移除）。但这个有些微妙，当用在可变映射上时，`+=`修改映射然后返回它。而用在不可变实例上时，返回的是一个包含新的键/值对的新映射。这会导致`+=`操作符出现以下边界情况：

```
scala> val m = Map(1 -> "hi", 2 -> "There", 3 -> "bye", 4 -> "quux")
m: scala.collection.immutable.Map[Int,java.lang.String]
= Map(1 -> hi, 2 -> There, 3 -> bye, 4 -> quux)

scala> m += (5 -> "Blah")
<console>:10: error: reassignment to val
      m += (5 -> "Blah")
           ^

scala> val hm = HashMap(1 -> "hi", 2 -> "There", 3 -> "bye", 4 -> "quux")
hm: scala.collection.mutable.HashMap[Int,java.lang.String]
= Map(3 -> bye, 4 -> quux, 1 -> hi, 2 -> There)

scala> hm += (5 -> "blah")
res6: hm.type = Map(5 -> blah, 3 -> bye, 4 -> quux, 1 -> hi, 2 -> There)
```

这是因为`+=`在不可变和可变映射中的实现是不一样的。对于可变映射，`+=`是一个方便修改映射的方法。这就是说在一个`val`映射上调用这个方法完全合法（就像Java在`final HashMap`上调用`put()`一样）。对于不可变映射，`+=`被分解成`=`和`+`的组合，就像在代码清单9-9里一样。它不能用在`val`上，因为`val`不允许再次赋值。

代码清单9-9中还有一个不错的语法：`for`循环。这用到了列表推导式（见9.3.5节）的思想，但结合了把键值对拆分成键和值的做法。这称为对解构，是Scala中另一个继承自函数式编程的概念。

对于Scala中的映射和它们的能力，我们仅仅触及了冰山一角，但我们要前往下一个主题了：泛型。

9

9.5.3 泛型

你已经知道了，Scala用方括号表示参数化类型，而且你也已经见过一些基本的Scala数据结构了。我们继续深入，看看Scala对泛型的处理方式跟Java有什么不同。

首先，如果在定义函数的参数类型时忽略了泛型，看看会发生什么：

```
scala> def junk(x : List) = println("hi")
<console>:5: error: type List takes type parameters
      def junk(x : List) = println("hi")
           ^
```

在Java中，这是完全合法的。编译器可能会抱怨，但不会报错。而在Scala中，这是一个编译时错误。列表（和其他泛型）必须参数化——故事讲完了，Scala没有Java“生类型”的概念。

1. 泛型的类型推断

把泛型赋值给一个变量时，Scala会对类型参数做出恰当的类型推断。这符合Scala一贯坚持的类型推断和尽可能去掉套路化代码的风格：

```
scala> val x = List(1, 2, 3)
x: List[Int] = List(1, 2, 3)
```

Scala泛型中有个特性乍一看可能觉得奇怪，我们用:::操作符演示一下，看到下面两个列表联接起来产生了新的列表，你就明白为什么说它奇怪了：

```
scala> val y = List("cat", "dog", "bird")
y: List[java.lang.String] = List(cat, dog, bird)
scala> x :: y
res0: List[Any] = List(1, 2, 3, cat, dog, bird)
```

奇怪吧，这样居然都不报错，还产生了新的List。运行时产生了一个Int和String的最小公父类（Any）的列表。

2. 泛型示例：候诊的宠物

假设有些宠物在等着看兽医，而你要建立候诊室里排队队列的模型。代码清单9-10是个不错的起点，用的是一些你已经熟悉的基础类和辅助函数。

代码清单9-10 候诊的宠物

```
class Pet(name : String)
class Cat(name : String) extends Pet(name : String)
class Dog(name : String) extends Pet(name : String)
class BengalKitten(name : String) extends Cat(name : String)

class Queue[T](elts : T*) {
    var elems = List[T](elts : _*)
    def enqueue(elem : T) = elems :: List(elem)           ← 需要类型提示
    def dequeue = {
        val result = elems.head
        elems = elems.tail
        result
    }
}

def examine(q : Queue[Cat]) {
    println("Examining: " + q.dequeue)
}
```

我们来考虑一下在Scala提示符中怎么使用这些类。这些是最简单的例子：

```
scala> examine(new Queue(new Cat("tiddles")))
Examining: line5$object$$iw$$iw$Cat@fb0d6fe

scala> examine(new Queue(new Pet("george")))
<console>:10: error: type mismatch;
       found   : Pet
       required: Cat
              examine(new Queue(new Pet("george")))
^
```

到目前为止都很像Java。我们再多做几个简单的例子：

```
scala> examine(new Queue(new BengalKitten("michael")))
Examining: line7$object$$iw$$iw$BengalKitten@464a149a

scala> var kitties = new Queue(new BengalKitten("michael"))
kitties: Queue[BengalKitten] = Queue@2976c6e4
```

```
scala> examine(kitties)
<console>:12: error: type mismatch;
  found   : Queue[BengalKitten]
  required: Queue[Cat]
     examine(kitties)
               ^
```

这也相当平常。第一个例子没有将kitties作为临时变量，Scala的类型推断把队列的类型作为Queue[Cat]，并接受了michael的加入，因为它的类型是Cat的子类BengalKitten。第二个例子中，创建了变量kitties，显式声明了其类型。也就是说Scala不能用类型推断，所以不能接受类型不匹配的参数。

接下来我们去看看如何用类型系统的类型变体解决这些类型问题，特别是协变（类型变体还有其他形态，但协变最常用）。在Java中，这非常灵活，但也有点神秘。Scala和Java的做法我们都会演示一下。

3. 协变

“在Java中，List<String>是List<Object>的子类吗？”如果你问过类似问题，那这个话题就是为你准备的。

默认情况下，Java对这个问题的回答是“不是”，但你可以让它变成“是”。要知道怎么做，请看下面的代码：

```
public class MyList<T> {
    private List<T> theList;
}

MyList<Cat> katzchen = new MyList<Cat>();
MyList<? extends Pet> petExt = pet1;
```

? extends Pet从句表示petExt是一个部分未知的类型参数（Java类型中的?读作“未知”）。可以确定的是MyList的类型参数必须是Pet或Pet的子类。这样在将类型参数为其子类的值赋给petExt时，Java编译器就不会阻拦。

这就相当于把MyList<Cat>变成了MyList<? extends Pet>的子类。注意，这种子类关系是在使用MyList类型时建立起来的，而不是定义时。类型的这个特性称为协变。

Scala的做法跟Java不同。它不是在使用类型时定义类型变体，而是在类型声明时显式指定协变。这样做有几个优势：

- 编译器可以在编译时检查不符合协变的使用；
- 所有概念上的思虑都交给了类型编写者，而不是抛给类型的使用者；
- 这样可以在基础集合类型间植入直观的关系。

理论上来说，这样的确不如Java那样使用现场的变体更灵活，但在实际应用中，Scala采取的方式所带来的好处完全可以抵消这种不便。大多数程序员很少会使用Java泛型中那些真正先进的特性。

Scala的标准集合，比如List，都实现了协变。这就是说List[BengalKitten]是List[Cat]的子类，而它又是List[Pet]的子类。我们来实际操练一下，请启动解释器：

```

scala> val kits = new BengalKitten("michael") :: Nil
kits: List[BengalKitten] = List(BengalKitten@71ed5401)

scala> var katzen : List[Cat] = kits
katzen: List[Cat] = List(BengalKitten@71ed5401)

scala> var haustieren : List[Pet] = katzen
haustieren: List[Pet] = List(BengalKitten@71ed5401)

```

我们在var上显式声明了类型，以免Scala把类型推断得过窄。

对Scala泛型的简略探讨到这里就结束了。下一个大主题是Scala在并发实现方式上的创新：放弃了多线程显式管理的方式，而选用了actor模型。

9.6 actor介绍

Java的显式锁和同步模型刻下了岁月的痕迹。在最初设计Java语言时，它是一个奇妙的创新，但也埋下了祸根。Java并发模型本质上是面对两难境地时采取折中策略的产物。

锁太少，会导致并发代码不安全，出现竞态条件。锁太多，系统会丧失活力，代码瘫痪，工作毫无进展。这就是我们在第4章讨论过的，安全性与系统活力之间的矛盾。

使用基于锁的模型，必须照顾到给定时间内所有可能发生的并发操作。但随着程序变得越来越大，要做到滴水不漏会变得越来越困难。尽管Java有办法缓解一些问题，但核心问题还在，如果Java语言不能发布一个拒绝向后兼容的版本，就不可能从根本上解决这个问题。

非Java语言有机会从头开始。备选语言可以不向程序员暴露锁和线程的底层细节，而是在自己的运行时环境中提供额外的并发支持。

这应该没什么好奇怪的。毕竟在Java刚刚出现时，Java内存模型就受到过质疑。当时很多C和C++开发人员都对这种想法感到诧异，怎么能由运行时负责管理内存，而让开发人员远离这些细节呢？

我们来看一下Scala基于actor技术的并发模型，看它如何让并发编程变了样（也更简单）。

9.6.1 代码大舞台

actor是扩展scala.actors.Actor，并实现了act()方法的对象。希望这个定义能跟你脑海中对Java线程的定义相呼应。它们最大的差别就是actor在大多数情况下都不会通过共享的数据进行沟通。

程序员在共享数据时必须采用最佳实践。如果你想在actor间共享状态，Scala不会阻止你。我们只是认为这么做不好。actor有沟通的渠道：mailbox，从另一个上下文中发送过来的消息（工作项）可以放在mailbox中交给actor，请参见图9-5。

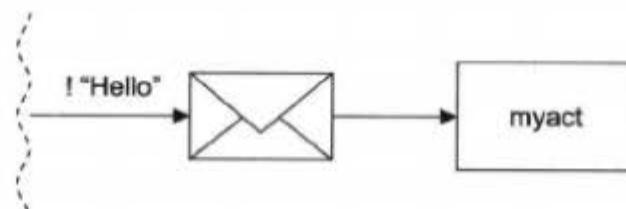


图9-5 scala的actor和mailbox

要创建actor，扩展Actor类就行：

```
import scala.actors._

class MyActor extends Actor {
    def act() {
        ...
    }
}
```

这看起来跟Java代码中声明Thread的子类很像。跟线程一样，我们也要告诉actor开始启动，并进入消息接收的状态，这要调用start()方法。

Scala同样提供了创建actor的工厂方法actor（与Java里创建Runnable匿名实现类的静态工厂方法相对应）。用它写出来的Scala代码很精炼：

```
val myactor = actor {
    ...
}
```

传给actor的代码块会变成act()方法中的内容。另外，这样创建的actor不需要再单独调用start()，它会自动启动。

这是一块香甜的语法糖，但我们还要介绍Scala并发模型的核心部件mailbox，所以别回味了，现在就去看看吧。

9.6.2 用 mailbox 跟 actor 通信

从另一个对象给actor发消息很简单，只要在actor对象上调用!方法就行了。

然而在接收端要有代码处理这些消息，否则它们就会堆在mailbox里。另外，actor方法体通常需要有个循环，以便能处理所有流入的消息。我们在Scala REPL中实际操练一下：

```
scala> import scala.actors.Actor._
         val myact = actor {
             while (true) {
                 receive {
                     case incoming => println("I got mail: " + incoming)
                 }
             }
         }
myact: scala.actors.Actor = scala.actors.Actor$$anon$1@a760bb0
scala> myact ! "Hello!"
I got mail: Hello!
scala> myact ! "Goodbye!"
I got mail: Goodbye!
scala> myact ! 34
I got mail: 34
```

9

上面代码中的receive方法就是actor对消息的处理。而工厂方法的参数（代码块）则是消息处理方法的主体。

注意 总体来说，Scala模型跟我们第4章（代码清单4-13）讨论的处理模式相似，Java处理线程相当于actor的角色，LinkedBlockingQueue相当于Scala中的mailbox。Scala只是以非常直白的方式为这种模式提供了语言和类库层面的支持，可以大量减少使用这种模式时所要编写的套路化代码。

尽管这个例子非常简单，但也包含了很多使用actor的基础知识：

- 在actor方法中要用循环的方式处理接收消息流；
- 用receive方法处理接收到的消息；
- 用一组case作为receive的主体。

最后这点还得继续讨论。这一组case被称为偏函数^①。之所以要这样用，是因为Scala中的actor还有一点比Java方便。具体来说就是mailbox是不区分类型的。也就是说你可以向actor发送任何类型的消息，actor可以用类型化模式和构造器模式接收不同类型的消息。

除了这些基础知识，这里还有一些使用actor的最佳实践。编写代码应该尽量遵循下面几条规则：

- 把传入消息做成不可变的；
- 考虑把消息类型做成case类；
- 不要在actor内部做阻塞操作，一个也别做。

不是每一个程序都需要遵守所有的最佳实践，但大多数应用程序应该都能从这些建议中受益。

对于更加复杂的actor，经常有必要控制它的启动和关闭。关闭actor通常都是用带有Boolean条件判断的循环。如果你喜欢，也可以将actor写成函数式的风格，这样传入的消息就不会影响它的状态。

Scala对基于actor的并发编程提供的支持还有很多。我们在这里看到的只是皮毛。如果想全面了解，请参阅Nilanjan Raychaudhuri的大作*Scala in Action* (Manning, 2010)。

9.7 小结

Scala跟Java有显著的差异：

- 支持更灵活的函数式编程风格；
- 类型推断使静态语言用起来有动态语言的感觉；
- Scala先进的类型系统扩展了Java的面向对象概念。

下一章会介绍最后一门非Java语言：Lisp方言Clojure，这可能是从各方面来看都最不像Java的语言。我们会以不可变性、函数式编程和另一种并发为基础展开讨论，并展示Clojure如何利用这些思想构建起了一个强大无比、美丽异常的编程环境。

^① 在Scala中，偏函数是指类型为PartialFunction[-A, +B]的函数。A是其接受的函数类型，B是其返回的结果类型。偏函数最大的特点就是它只接受其参数定义域的一个子集，而对于这个子集之外的参数则抛出运行时异常。这与case语句非常契合，因为我们在使用case语句时常常是匹配一组具体的模式，最后用“_”来代表剩余的模式。如果一组case语句没有涵盖所有的情况，那么这组case语句就可以被看做是一个偏函数。——译者注

Clojure：更安全地编程

10

本章内容

- Clojure实体和状态的概念
- Clojure的REPL
- Clojure语法、数据结构和序列
- Clojure与Java的交互能力
- Clojure的多线程开发
- 软件事务内存

Clojure跟Java以及我们前面研究的语言差别很大。Clojure是在JVM上重新实现的Lisp。Lisp是最古老的编程语言，如果你对它还不熟悉，没关系。与Lisp语言家族有关的一切，只要是你需要知道的，我们都会告诉你，你可以安心开始Clojure之旅。

除了从Lisp继承的强大编程技术，Clojure还增添了一些令人惊叹的前沿技术。这种组合让Clojure从JVM语言中脱颖而出，成为应用程序开发的诱人选择。

Clojure中的并发工具包和数据结构就是一项新技术。并发抽象层让程序员可以写出更加安全的多线程代码。它和Clojure的序列抽象层（对集合和数据结构上的不同看法）相结合，为开发人员提供了非常强大的工具箱。

想掌握这些力量，先要了解Clojure跟Java在编程方式上截然不同的理念。这种差异使得Clojure学起来很有趣，并且很可能会改变你的思考方式。不管你用的是什么语言，学习Clojure都会让你成为更好的程序员。

我们一开始会先讨论Clojure处理状态和变量的方式。在给出一些简单的例子后，会介绍这门语言的基本词汇表——用来构建语言其余部分的特殊形态。我们将深入到Clojure的语法中，了解它的数据结构、循环和函数。然后介绍序列，这是Clojure最强的抽象概念之一。我们会用两个非常引人注目的特性来给这一章收尾：跟Java的紧密集成以及Clojure惊人的并发支持。

10.1 Clojure 介绍

我们先来看Clojure跟Java在理念上最重要的差别，即对状态，变量和存储的不同认识。如图

10-1所示，Java（跟Groovy和Scala一样）有一个内存和状态模型，把变量当作保存可变内容的“盒子”（内存位置）。

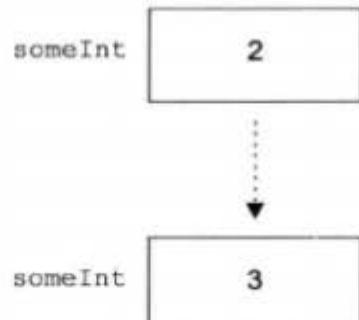


图10-1 命令式语言的内存使用

而Clojure认为值才是真正重要的概念。值可以是数字、字符串、向量、映射、集合，或其他任何东西。一旦创建，值就再也不会改变。这一点真的非常重要，所以我们要再重复一次。一旦创建，Clojure的值就不能再变了，因为它们是不可变的。

这就是说命令式语言那种装着可变内容的盒子模型不是Clojure思考问题的方式。图10-2是Clojure处理状态和内存的方式。它在名字和值之间创建了一个关联关系。

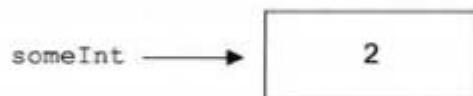


图10-2 Clojure的内存使用

这就是绑定，通过特殊形式(`def`)建立。Clojure中的特殊形式相当于Java的关键字，但请注意，Clojure中的术语“关键字”含义不同，稍后我们会介绍。

`(def)`的句法是：

`(def<名称> <值>)`

如果你觉得这个句法看起来有点怪异，不要担心，这完全是Lisp的普通句法，你很快就会习惯的。现在你可以假装是在调用下面这样一个方法，只是括号的位置不太一样：

`def(<名称>, <值>)`

接下来我们要在Clojure的交互式环境中写一个久经考验的例子，演示一下`(def)`的用法。

10.1.1 Clojure 的 Hello World

如果你还没装Clojure，请参见附录D。然后切换到Clojure所在的目录，运行如下命令：

`java -cp clojure.jar clojure.main`

这个命令会启动Clojure的REPL环境。在编写Clojure代码时，你会在这个交互环境里花上很多时间。

`user=>`是Clojure的会话提示符，你可以把这个会话环境当做高级的调试环境，或者命令行工具：

```

user=> (def hello (fn [] "Hello world"))
#'user/hello
user=> (hello)
"Hello world"

```

这段代码一开始先给标识符hello绑定一个值。(def)就是用来建立标识符(Clojure称为符号)和值之间的绑定关系的。底层实现的时候，它也会创建一个对象var，用来表示这种绑定关系(和符号的名字)。

那这里绑定的值是什么？这个值是：

```
(fn [] "Hello world")
```

这是一个函数，在Clojure中也是一个纯正的值(因此也是不可变的)。这个函数没有参数，返回字符串"Hello world"。

绑定之后，可以用(hello)执行。Clojure运行时会输出该函数的计算结果，也就是"Hello world"。

现在，应该录入这个例子(如果你还没做)，看看它的表现是不是跟我们说的一样。完成之后，我们就可以继续探索了。

10.1.2 REPL 入门

在REPL中可以输入Clojure代码，也可以执行Clojure函数。它是个交互式环境，而且在前面得出的计算结果不会被丢掉。可以用它做探索式编程，我们会在10.5.4节讨论这种编程方式，基本就是不断试验代码。用Clojure开发经常都是先在REPL里把代码调好，然后用正确的构件搭出越来越大的函数。

马上看一个例子。先声明，再次调用def可以改变符号和值的绑定关系，我们在REPL中看一下。代码中用的实际上是(def)的变体(defn)：

```

user=> (hello)
"Hello world"
user=> (defn hello [] "Goodnight Moon")
#'user/hello
user=> (hello)
"Goodnight Moon"

```

注意，hello最初的绑定关系一直都在，直到被你改掉，这是REPL的一个关键特性。这还是状态，只不过换了个说法，变成了哪个符号绑定到哪个值上，并且这个状态存在于用户输入的不同行间。

Clojure中没有可变状态，但有可以改变绑定值的符号。Clojure不是让“内存盒子”中的内容改变，而是让符号绑定到不同的不可变值上。换句话说就是在程序的生命期内，var可以指向不同的值。请参见图10-3。

注意 可变状态和不同绑定两者之间的区别很微妙，但这个概念很重要，一定要掌握。要记住，可变状态是指盒子中的内容变了，而重新绑定是指在不同时间指向不同的盒子。

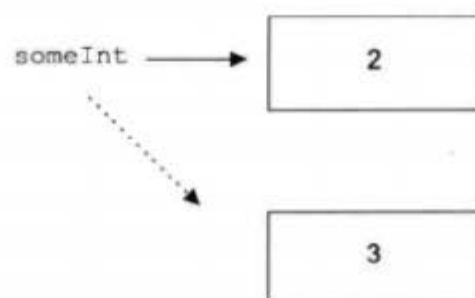


图10-3 可以改变的Clojure绑定

这段代码中还溜进了另一个Clojure概念，“定义函数”宏(`(defn)`)。宏是类Lisp语言的关键概念之一，其核心思想是内置结构和普通代码之间的区别应该尽可能小。

用宏可以创建跟内置语法类似的形式。创建宏是高级话题，但掌握了它之后，你就能制造出非常强大的工具。

这就是说语言真正的原语系统（特殊形式）可以用一种几乎无法察觉的方式构建起整个语言的核心。宏(`(defn)`)就是这种构建的产物。它只是将函数值绑定到符号的相对简单的方法（当然，要创建合适的`var`）。

10.1.3 犯了错误

如果你犯错了，会怎么样？比如你漏掉了`[]`（函数声明的一部分，表明这个函数没有参数）。

```

user=> (defn hello "Goodnight Moon")
#'user/hello
user=> (hello)
java.lang.IllegalArgumentException: Wrong number of args (0) passed to:
user$hello (NO_SOURCE_FILE:0)
  
```

所有后果只是`hello`标识符绑定到了一个未知的东西上。你可以在REPL中重新绑定来修复它：

```

user=> (defn hello [] (println "Dydh da an Nor"))
=> ; "Hello World" in Cornish
#'user/hello
user=> (hello)
Dydh da an Nor
nil
user=>
  
```

跟你猜的一样，上面这段代码中的分号`(;)`表示直到行尾的内容都是注释，`(println)`是输出字符串的函数。注意看`(println)`，它跟所有函数一样，返回了一个值，在函数执行结束后回显到REPL中。结果值是`nil`，相当于Java里的`null`。

10.1.4 学着去爱括号

奇思妙想和幽默感是程序员文化不可或缺的一部分。说Lisp是“很多烦人的傻括号”的缩写就是个很古老的笑话。其实Lisp是列表处理（List Processing）的缩写，真相就是这么平淡无奇。很多Lisp程序员都用这个笑话自嘲，因为它确实戳到了Lisp语法的痛处。

实际上，这个障碍被夸大了。Lisp句法的确特立独行，但也不像看起来那么碍手碍脚。另外，Clojure还为减轻入门的障碍做了几项创新。

我们再看一下Hello World。调用返回“Hello World”的函数写成：

```
(hello)
```

用Java写应该是这样（假设你已经在某个类里定义了hello方法）：

```
hello();
```

但Clojure的表达式不是myFunction(someObj)，而是(myFunction someObj)。这种写法叫波兰表示法，因为它是19世纪的波兰数学家发明的。

如果你研究过编译原理，可能想知道这是否和抽象语法树（AST）之类的概念有关。简单地说是“有”。可以证明，用波兰表示法（Lisp程序员通常管它叫s表达式）写成的Clojure或其他Lisp程序是其简单直接的AST表示。

你可以认为Lisp程序是直接用AST写的。Lisp程序的数据结构表示和代码没有本质上的差别，所以代码和数据是完全可以互换的。这也是Clojure的表示法看起来有点奇怪的原因——类Lisp语盲用它来模糊内置的原生代码、用户代码和类库代码之间的区别。对于Java程序员来说，这股强大力量对他们的引力要远远超过稍微有点古怪的语法。

让我们更深入地学一些Clojure语法，然后用它写一些真正的程序。

10.2 寻找 Clojure：语法和语义

我们上一节介绍了(def)和(fn)两个特殊形式（special form）。这里还有几个需要你马上掌握的特殊形式，它们构成了语言的基础词汇表。Clojure中还有大量实用的形式和宏，用得越多，认识会越来越深刻。

Clojure中的函数非常多，托它们的福，你能想到的任务很多都可以用Clojure完成。不要因此而沮丧，你应该感到庆幸。你要干的活大部分都有人替你干了，不该高兴吗？

我们在这一节会讨论特殊形式的基本工作集，然后是Clojure的原生数据类型（相当于Java的集合）。之后会接着讨论Clojure代码的自然编写风格——以函数而不是变量为中心。JVM面向对象的性质在底层还会存在，但Clojure强调函数的那种力量在纯粹的面向对象语言中表现得不太明显。

10

10.2.1 特殊形式新手营

表10-1给出了一些最常用的Clojure特殊形式。你现在最好快速地把这张表过一遍，然后在10.3节遇到具体例子时再回来看看。

这个特殊形式列表不算详尽，并且其中很多特殊形式都有多种用法。表10-1中只是它们的基本用例，而且都不全面。

表10-1 Clojure一些基本的特殊形式

特殊形式	含 义
(def<符号><值?>)	把符号绑到值上（如果有的话）。如有必要创建与符号对应的var
(fn<名称? [<参数>*]<表达式*>*)	返回带有特定参数的函数值，并把它们应用到表达式上。通常跟(def)相结合，变成形式(defn)
(if<test> <then> <else>?)	如果test的计算结果为true，计算then并产出其结果。否则计算else并产出其结果，当然，前提是else存在
(let[<绑定*>*] <表达式*>*)	给局部名称分配别名值，并隐式定义一个作用域。使得在let作用域内的所有表达式都能获得该别名
(do<表达式*>*)	按顺序计算表达式的值，并产出最后一个的结果
(quote<形式>)	照原样返回形式（不经计算）。它只能接受一个形式参数，其他的参数全都会被忽略
(var<符号>)	返回与符号对应的var（返回一个Clojure JVM对象，不是值）

现在你对一些特殊形式的基本语法有进一步的了解了，让我们转去看看Clojure的数据结构吧，也看看它们怎么操作数据。

10.2.2 列表、向量、映射和集

Clojure中有几个原生数据类型。用的最多的是列表（list），即单向链表。

列表通常都用括号围起来，因为形式一般也是用圆括号，所以这算是一个轻微的语法障碍。况且括号还用来调用函数。所以初学者经常会犯下面这种错误：

```
1:7 user=> (1 2 3)
java.lang.ClassCastException: java.lang.Integer cannot be cast to
clojure.lang.IFn (repl-1:7)
```

之所以会出错，是因为Clojure中的值非常灵活，它希望第一个参数是函数值（或绑定到函数值上的符号），把2和3当做这个函数的参数。可在上例中1不是函数值，所以Clojure无法编译。按我们的说法，这个s表达式是无效的。只有有效的s表达式才能作为Clojure形式。

解决办法是用(quote)形式，它的缩写是'。所以我们可以用两种方式定义列表：

```
1:22 user=> '(1 2 3)
(1 2 3)
1:23 user=> (quote (1 2 3))
(1 2 3)
```

(quote)以一种特殊的方式处理它的参数。具体来说就是它不会计算参数，所以第一个参数不是函数值也没问题。

Clojure的向量（vector）跟数组类似，实际上，基本上可以把Clojure列表等同于Java的LinkedList，向量等同于ArrayList。向量可以用方括号表示，所以下面这些定义都一样：

```
1:4 user=> (vector 1 2 3)
[1 2 3]
1:5 user=> (vec '(1 2 3))
[1 2 3]
1:6 user=> [1 2 3]
[1 2 3]
```

在前面声明Hello World和其他函数时，就是用向量来表示函数的参数。注意，(vec)形式以一个列表为参数，并用这个列表创建向量，而(vector)形式以多个独立符号为参数，并返回包含它们的向量。

函数(nth)有两个参数：集合和索引。它跟Java中List接口的get()方法类似。可以用在向量和列表上，也可以用在Java集合甚至字符串（字符的集合）上，请看下例：

```
1:7 user=> (nth '(1 2 3) 1)
2
```

Clojure也支持映射（map，相当于Java的HashMap），定义很简单：

```
{key1 value1 key2 "value2"}
```

从映射里取值也非常简单：

```
user=> (def foo {"aaa" "111" "bbb" "2222"})
#'user/foo
user=> foo
{"aaa" "111", "bbb" "2222"}
user=> (foo "aaa")
"111"
```

Clojure把前面带冒号的映射键称为“关键字”：

```
1:24 user=> (def martijn {:name "Martijn Verburg",
  :city "London", :area "Highbury"})
#'user/martijn
1:25 user=> (:name martijn)
"Martijn Verburg"
1:26 user=> (martijn :area)
"Highbury"
1:27 user=> :area
:area
1:28 user=> :foo
:foo
```

关于关键字，请记住下面这些知识点。

- Clojure的关键字是只有一个参数的函数，其参数必须是映射。
- 在映射上调用这个函数会返回映射里与该关键字函数对应的值。
- 关键字的使用遵循语法对称性规则，即(my-map :key) 和(:key my-map)都是合法的。
- 关键字作为值使用时返回自身。
- 关键字在使用之前无需声明或def。
- Clojure中的函数也是值，因此可以放在映射里当键用。
- 可以用逗号（但没必要）来分隔键/值对，因为Clojure会把它们当做空格处理。
- 除了关键字，其他符号也能用在映射里做键，但关键字太好用了，所以我们要特别提出来，你应该把它用自己的代码中。

除了映射字面值，Clojure还有个(map)函数。但不要上当，它不像(list)，(map)函数不会产生映射。而是对集合中的元素轮番应用其参数中的函数，并用返回的新值建立一个新集合（实际上是Clojure序列，请参见10.4节）。

```

1:27 user=> (def ben {:name "Ben Evans", :city "London", :area "Holloway"})
#'user/ben
1:28 user=> (def authors [ben martijn])
#'user/authors
1:29 user=> (map (fn [y] (:name y)) authors)
("Ben Evans" "Martijn Verburg")

```

(`map`)还有别的形式，可以一次处理多个集合，但一次输入一个集合的形式最常用。

Clojure也支持集（`set`），跟Java的`HashSet`很像。它的缩写形式是：

```
#{"apple" "pair" "peach"}
```

这些数据结构是构建Clojure程序的基础。

Java土著可能会感到吃惊，居然一直没有提到对象。这不是说Clojure不是面向对象的，但它对面向对象的观点的确和Java不一样。Java认为世界是由封装了数据和代码的静态数据类型组成的。而Clojure强调函数和形式，尽管这些在底层都是由JVM上的对象实现的。

Clojure和Java在世界观上的差别最终会体现在代码里。要充分理解Clojure的观点，必须用Clojure写些程序，并弄明白相比Java的面向对象结构它有哪些优势。

10.2.3 数学运算、相等和其他操作

Clojure没有Java里那种意义上的操作符。所以怎么才能，比如说，让两个数相加呢？在Java里这很容易：

```
3 + 4
```

但Clojure没有操作符，只能用函数：

```
(add 3 4)
```

这也挺好，但我们可以做得更好。因为Clojure里没有操作符，所以我们不用为它们保留任何字符。这就是说Clojure的函数名称可以更加稀奇古怪，所以我们可以这样写^①：

```
(+ 3 4)
```

Clojure函数一般都支持变参（参数数量可变），比如还可以这样：

```
(+ 1 2 3)
```

这个运算结果是6。

Clojure的相等形式（相当于Java里的`equals()`和`==`）状况稍微有点复杂。Clojure有两个跟相等相关的形式：`(=)`和`(identical?)`。注意它们的名字，这全都是因为Clojure不用为操作符保留字符。另外，`(=)`也是等号，而不是赋值符号。

下面这段代码设置了一个列表`list-int`和一个向量`vect-int`，并比较它们是否相等：

^① 例子中的`(+)`是`clojure.core`命名空间下的函数，能够接受0到任意数目的参数，假如没有参数，则返回0。所以虽然Clojure没有操作符，但有很多提供了操作符功能的核心函数，所以你大可不必担心怎么计算`3 * 4`，用早已准备好的函数`(* 3 4)`就行了。——译者注

```

1:1 user=> (def list-int '(1 2 3 4))
#'user/list-int
1:2 user=> (def vect-int (vec list-int))
#'user/vect-int
1:3 user=> (= vect-int list-int)
true
1:4 user=> (identical? vect-int list-int)
false

```

(=)形式会检查集合是否由相同的对象以相同的顺序组成的 (list-int和vect-int符合这一要求)，而(identical?)会检查它们是否真的是同一个对象。

你可能也注意到了，符号名称都没有用驼峰式大小写^①。这在Clojure中很常见，符号通常都用小写，单词之间用连字符连接。

Clojure中的true与false

Clojure中有两个值表示逻辑假：false和nil。其他全是逻辑真。很多动态语言都这样，但对于Java程序员来说这有点奇怪。

掌握了基本的数据结构和操作符，让我们把之前见过的特殊形式和函数拼到一起，写一个稍微长点的Clojure函数吧。

10.3 使用函数和循环

从本节开始，我们会接触到Clojure中一些实质性的内容。从编写函数处理数据开始，让你看到Clojure对函数的重视程度。接着介绍循环结构，以及读取器（reader）宏和派发（dispatch）形式。最后，我们会以Clojure的函数式编程和闭包作为本节的收尾。

举例说明是好办法，所以我们先来几个简单的例子，然后朝Clojure提供的强大函数式编程技术进发。

10.3.1 一些简单的Clojure函数

代码清单10-1中定义了三个函数。其中两个是非常简单的单参函数，另一个稍微有点复杂。

代码清单10-1 定义简单的函数

```

(defn const-fun1 [y] 1)

(defn ident-fun [y] y)

(defn list-maker-fun [x f]
  (map (fn [z] (let [w z]
                 (list w (f w)))
         ) x))

```

^① 驼峰式大小写（Camel-Case）一词来自Perl语言中普遍使用的大小写混合格式，而Larry Wall等人所著的畅销书 *Programming Perl: Unmatched power for text processing and scripting* (O'Reilly, 2012) 的封面图片正是一匹骆驼。

——译者注

在这段代码中，(const-fun1)接受一个参数，返回1，(ident-fun)接受一个数值并返回数值本身。数学家会管它们叫常量函数和恒等函数。还有，函数定义中使用向量表示函数的参数，(let)形式中用的也是向量。

第三个函数比较复杂。函数(list-maker-fun)有两个参数：第一个是包含所处理的值的向量x，第二个一定是函数。

我们来看一下如何使用list-maker-fun，如代码清单10-2所示。

代码清单10-2 使用函数

```
user=> (list-maker-fun ["a"] const-fun1)
(("a" 1))
user=> (list-maker-fun ["a" "b"] const-fun1)
(("a" 1) ("b" 1))
user=> (list-maker-fun [2 1 3] ident-fun)
((2 2) (1 1) (3 3))
user=> (list-maker-fun [2 1 3] "a")
java.lang.ClassCastException: java.lang.String cannot be cast to
clojure.lang.IFn
```

把这些表达式敲到REPL中实际上是在和Clojure的编译器交互。表达式(list-maker-fun [2 1 3] "a")之所以无法编译，是因为(list-maker-fun)的第二个参数应该是函数，而字符串显然不是。看到10.5节你就会知道，对于VM来说，Clojure函数是实现了clojure.lang.IFn的对象。

这个例子表明在跟REPL交互时仍然会涉及一些静态类型问题。因为Clojure不是解释型语言。即便是在REPL中，输入的每个Clojure形式都会被编译成JVM字节码并连接到运行时系统上。Clojure函数在定义完后就被编译成JVM字节码了，所以在出现静态类型冲突时VM会报出ClassCastException异常。

代码清单10-3中的Clojure代码更长。Schwartzian转换可有年头了，从20世纪90年代在Perl中出现后就一直在用。其基本思想是基于向量中元素的某些属性对元素进行排序。排序所依据的属性值是通过在元素上调用键控函数确定的。

代码清单10-3中定义的Schwartzian转换所调用的键控函数是key-fn。在真正调用(schwartz)函数时需要提供一个用作键控的函数。代码清单10-3中用的是我们的老朋友(ident-fun)。

代码清单10-3 Schwartzian转换

```
1:65 user=> (defn schwartz [x key-fn]
  (map (fn [y] (nth y 0)) ← 第三步
        (sort-by (fn [t] (nth t 1)) ← 第二步
                 (map (fn [z] (let [w z] ← 第一步
                               (list w (key-fn w)))
                       ) x))))
```

#'user/schwartz

```
1:66 user=> (schwartz [2 3 1 5 4] ident-fun)
(1 2 3 4 5)
1:67 user=> (apply schwartz [[2 3 1 5 4] ident-fun])
(1 2 3 4 5)
```

这段代码分为三步：

- 创建一个包含键值对的列表；
 - 基于键控函数的值对键值对排序；
 - 仅从排好序的键值对列表中取出原始值，构建新列表（并抛弃键控函数值）。
- 如图10-4所示。

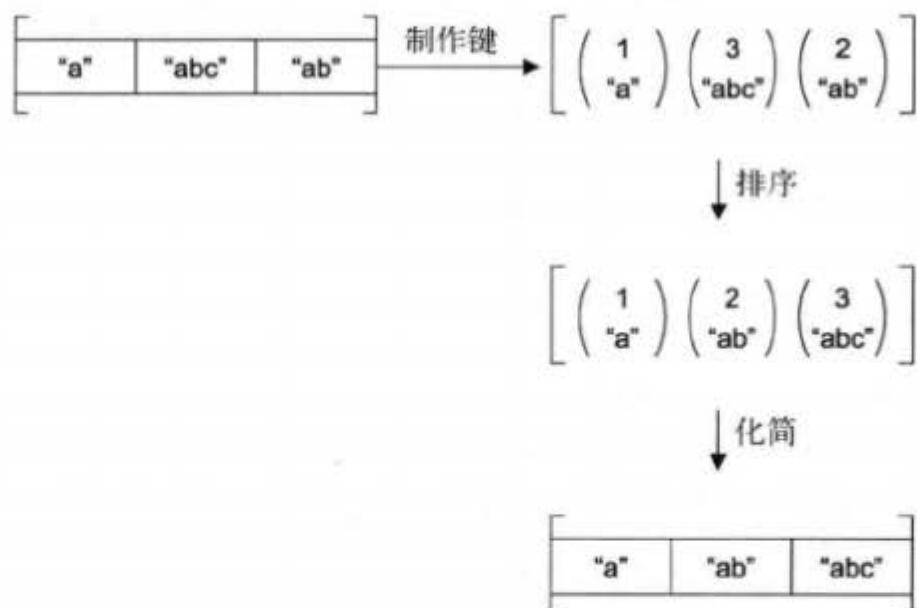


图10-4 Schwartzian转换

代码清单10-3中引入了一个新形式：(sort-by)。这个函数有两个参数：一个是用来排序的函数，一个是要排序的向量。还有(apply)形式，它也有两个参数：一个是要调用的函数，一个是传给它的向量参数。

Randall Schwartz最初用Perl编写Schwartzian转换（该转换以他的名字命名）时在刻意模仿Lisp。我们现在又用Clojure编写，算是绕了一圈又回来了。挺有意思！

Schwartzian转换的示例很实用，我们稍后还会用到它。因为它的复杂性足以用来阐明好几个概念。

接下来我们来讨论下Clojure的循环，可能和你所习惯的循环有点不太一样。

10.3.2 Clojure 中的循环

Java里的循环相当简单直接，可选的循环有for、while，还有其他几种。其核心思想通常是重复一组指令，直到满足某一条件（一般用一个可变变量表示）。

这对Clojure是个小难题：举个例子，对于没有可变变量作为循环索引的Clojure，怎么表示for循环呢？在传统的Lisp中通常用递归形式实现循环。但JVM不能保证尾递归优化（Scheme和其他Lisp语言有这种要求），所以在Clojure中用递归可能会导致栈溢出。

而Clojure有不会增加栈空间占用的结构。最常用的是loop-recur，下面的代码展示了如何用loop-recur构建一个和for循环类似的结构。

```
(defn like-for [counter]
  (loop [ctr counter]
    (println ctr)
    (if (< ctr 10)
        (recur (inc ctr))
        ctr
      )))

```

(loop)形式以包含符号局部名称的向量为参数——像(let)定义的别名。然后当执行到(recur)形式时（本例中只有ctr别名小于10才会执行该形式），它会将控制分支返回到(loop)形式中，但指定了新的值。这样我们就可以搭建循环式结构（比如for和while循环），但实现中仍有递归的味道。

现在我们转入下一主题，看一看Clojure语法的简写，帮你把程序写得更短、更精炼。

10.3.3 读取器宏和派发器

Clojure有些让很多Java程序员吃惊的语法特性。其中之一是没有操作符。它的副作用是放宽了Java对能用在名称中的字符的限制。你已经见过像(identical?)这样的函数了，这在Java中是非法的，但对于哪些字符不能用在符号中，我们还没有说明。

表10-2列出了不能用在Clojure符号中的字符。Clojure分析器保留了这些字符自用，它们通常被称为读取器宏。

表10-2 读取器宏

字 符	名 称	含 义
'	引号	展开为(quote)，产出不进行计算的形式
;	注释	标记直到行尾的注释，就像Java里的//
\	字符	产生一个字面字符
@	解引用	展开为(deref)，接受var对象并返回对象中的值（跟(var)形式的操作相反）。在事务内存上下文中还有其他含义（见10.6节）
^	元数据	将一个元数据的映射附加到对象上。请查阅Clojure文档了解详情
`	语法引用	经常用在宏定义中的引号形式，不太适合初学者。请查阅Clojure文档了解详情
#	派发	有几种不同的子形式，见表10-3

根据#后面的字符，派发读取器宏有几种不同的子形式，请见表10-3。

表10-3 派发读取器宏的子形式

派发形式	含 义
#'	展开为(var)
#{} #()	创建一个集字面值，在10.2.2节中用过 创建匿名函数字面值，用在那些使用(fn)太啰嗦的地方
#_	跳过下一个形式。可以用#_(...多行...)来创建多行注释
#"<模式>"	创建一个正则表达式（作为java.util.regex.Pattern对象）

关于派发形式，还有几点要提一下。变量引用形式#’解释了REPL执行(def)之后的表现：

```
1:49 user=> (def someSymbol)
#'user/someSymbol
```

(def)形式返回新创建的var对象，命名为someSymbol，驻留在当前的命名空间中（就是用户所在的REPL），所以#’user/someSymbol是(def)返回的完整值。

匿名函数字面值也是减少繁琐代码的创新。它省略了参数向量，用一种特殊的语法让Clojure读取器推断函数字面值需要多少个参数。

代码清单10-4是我们用这个语法重写的Schwartzian转换。

代码清单10-4 重写Schwartzian转换

```
(defn schwartz [x f]
  (map #(nth %1 0)
       (sort-by #(nth %1 1)
                 (map #(let [w %1]
                         (list w (f w)))
                   ) x))))
```

匿名函数字面值

用%1当做函数字面值参数的占位符（后续参数可以用%2、%3等）真的很好，这样的代码也更容易看懂。这种显而易见的线索对程序员很有帮助，就像你在9.3.6节见过的Scala函数字面值里的箭头符号一样。

Clojure严重依赖于以函数为基本计算单元的概念，而不像Java以对象为语言的根本。这种方式自然会导向函数式编程，也就是我们的下一主题。

10.3.4 函数式编程和闭包

我们现在要进入恐怖的Clojure函数式编程世界。或者，我们没有，因为它不恐怖。实际上，我们这一整章都在学习函数式编程，只是没告诉你，怕把你吓跑。

7.3.2节中说过，函数式编程意味着函数是一个值。函数可以传递，放在变量中操作，就像2或"hello"一样。但那又怎么样？我们回头看看第一个例子：(def hello (fn [] "Hello world"))。我们创建了一个函数（没有参数，返回字符串"Hello world"），把它绑定到符号hello上。函数仅仅是个值，本质上跟2这种值没什么区别。

在10.3.1节，我们以Schwartzian转换为例介绍了以另外一个函数为输入值的函数。这也不过是一个以特定类型为输入参数的函数，唯一的区别不过是这个类型是函数。

关于闭包呢？它们真的很恐怖，是不是？哦，还好吧。我们来看一个简单的例子，这应该能让你想起我们做过的一些Scala例子：

```
1:5 user=> (defn adder [constToAdd] #(+ constToAdd %1))
#'user/adder
1:6 user=> (def plus2 (adder 2))
#'user/plus2
1:7 user=> (plus2 3)
5
1:8 user=> 1:9 user=> (plus2 5)
7
```

上例中先定义了`(adder)`函数。这是一个构造其他函数的函数。如果你熟悉Java语言的工厂方法模式，可以把它当成Clojure的工厂方法实现。以其他函数为函数的返回值没什么好奇怪的，这是将函数作为普通值这一概念的重要体现。

这个例子给匿名函数用了缩写的`#()`形式。函数`(adder)`接受一个数值参数并返回一个函数，并且返回的是带一个参数的函数。

然后用`(adder)`定义了一个新形式：`(plus2)`。这个函数接受一个参数，并在这个参数上加2。这就是说绑定到`(adder)`内部的`constToAdd`的值是2。现在我们来构造一个新函数：

```
1:13 user=> (def plus3 (adder 3))
#'user/plus3
1:14 user=> (plus3 4)
7
1:15 user=> (plus2 4)
6
```

这段代码表明你还可以再构造其他函数`(plus3)`，绑定不同的值到`constToAdd`上。我们说函数`(plus3)`和`(plus2)`已经从它们所在的环境中捕获或“封装”了一个值^①。需要注意的是`(plus3)`和`(plus2)`捕获的值是不同的，并且定义`(plus3)`对`(plus2)`捕获的值没有影响。

在自身环境内“封装”一些值的函数称为闭包，`(plus2)`和`(plus3)`就是闭包。在支持闭包的语言中，用一个制造者函数构造并返回另一个封装了一些东西的函数非常普遍。

接下来我们要讨论Clojure中一个强大的特性：序列。它们使用了跟Java的集合或迭代器类似的东西，但有些不同的属性。在代码中使用序列最能体现Clojure语言的力量，对于习惯了Java处理方式的程序员，Clojure的处理方式会让你耳目一新。

10.4 Clojure 序列

看下面这段代码中的Java迭代器。这是使用迭代器的老套路了。实际上，Java 5里的`for`循环在底层也会被转换成这种实现：

```
Collection<String> c = ...;
for (Iterator<String> it = c.iterator(); it.hasNext();) {
    String str = it.next();
    ...
}
```

对于简单集合的循环处理这就够了，比如`Set`或`List`。但`Iterator`接口只有`next()`和`hasNext()`方法，加上一个可选的`remove()`方法。

1. 残缺的Java迭代器

然而Java迭代器还有缺陷。迭代器接口所提供的集合交互方法满足不了需求。用`Iterator`只能做两件事：

- 查看集合中是否还有更多的元素；

^① 此处的环境即指函数`(adder)`，而捕获的值即绑定到`constToAdd`的值。——译者注

- 取出下一个元素，并把迭代器向前推进。

Iterator最主要的问题是把取得下一个元素和向前推进合在了一起（如图10-5所示）。这意味着无法先对集合中的下一个元素进行检查，然后再决定它是需要特殊处理，还是完好无损地取出去。

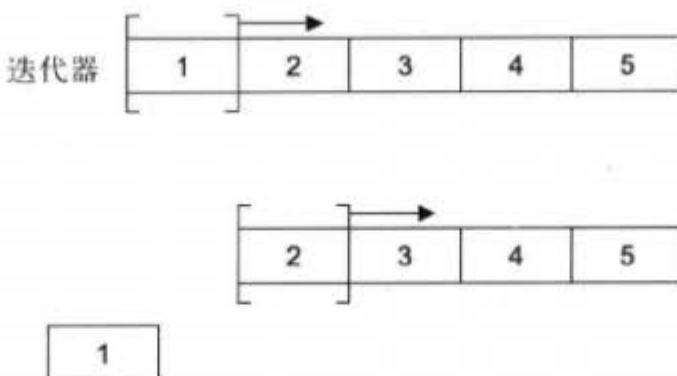


图10-5 Java迭代器的性质

从迭代器中取出下一元素的行为改变了它的状态。也就是说可变已经内建在Java处理集合和迭代器的方法中了，因此不可能用它构建出强健的多路解决方案。

2. Clojure的键抽象

Clojure采用了不同的方式。Clojure与Java中的集合与迭代器相对应的核心概念是序列(sequence)，或者简称seq。它基本上是把两个Java类的一些特性集成到了一个概念里。这样做的动机有三个：

- 更强健的迭代器，特别是对于多路算法而言；
- 不可变能力，可以安全地在函数间传递序列；
- 实现懒序列的可能性（后面还会详细讨论）。

表10-4中列出了跟序列相关的一些核心功能。这些函数都不会改变它们的参数，如果它们需要返回不同的值，那会是一个不同的序列。

表10-4 基本的序列函数

函 数	作 用
(seq <coll>)	返回一个序列，作为所操作集合的“视图”
(first <coll>)	返回集合的第一个元素，如有必要，先在其上调用(seq)。如果集合为nil，则返回nil
(rest <coll>)	返回从集合中去掉第一个元素后得到的新序列。如果集合为nil，则返回nil
(seq? <o>)	如果o是一个序列则返回true（也就是实现了ISeq）
(cons <elt> <coll>)	在集合前面增加新元素，并返回由此得到的序列
(conj <coll> <elt>)	返回将新元素加到合适一端（向量的尾端和列表的头）的新集合
(every? <pred-fn> <coll>)	如果(pred-fn)对集合中的每个元素都返回逻辑真，则返回true

这里有几个例子：

```

1:1 user=> (rest '(1 2 3))
(2 3)
1:2 user=> (first '(1 2 3))
1
1:3 user=> (rest [1 2 3])
(2 3)
1:13 user=> (seq ())
nil
1:14 user=> (seq [])
nil
1:15 user=> (cons 1 [2 3])
(1 2 3)
1:16 user=> (every? is-prime [2 3 5 7 11])
true

```

有一点要重点关注一下，列表是自身的序列，而向量不是。因此从理论上来说，不能在向量上调用(rest)。可实际上是可以的，因为(rest)在操作向量之前先在其上调用了(seq)。这是序列结构中普遍存在的属性：很多序列函数都会接受比序列更通用的对象，并在开始之前先调用(seq)。

我们在这一节中准备探索seq的一些基本属性和用法，尤其会重点关注懒序列和变参函数。其中第一个概念“懒”，是Java中不太会涉及的编程技术^①，所以对你来说它可能比较新颖。现在我们就来看一下吧。

10.4.1 懒序列

在编程语言里，懒是一个强大的概念。其基本思想是将表达式的计算推迟到需要时。体现在Clojure中就是序列可以不是完整的值列表，其中的值可以在被请求时取得（比如根据需要通过调用函数生成它们）。

在Java中，要满足这样的想法就得靠定制的List实现，而且要写大量的套路化代码才可能实现。用Clojure中的宏只要做一点儿工作就能创建出懒序列。

想一想怎么才能创建出一个懒惰的、可能包含无限数量值的序列。很明显，用函数来生成序列内的元素。这个函数应该做两件事：

- 返回序列中的下一个元素；
- 接受数量固定、有限的参数。

数学家会说这样一个函数定义的是递归关系，并且这样的关系用递归的方式处理再恰当不过了。

假设有一台在栈空间和其他能力上都不受限制的机器，并且可以执行两个线程：一个用来生成无限的序列，另外一个使用该序列。那我们就可以在生成线程里用递归定义懒序列，类似下面这段伪代码：

```
(defn infinite-seq <vec-args>
  (let [new-val (seq-fn <vec-args>)]
    (cons new-val (infinite-seq <new-vec-args>))))
```

^① 用过Hibernate的人一定知道懒加载（因为它原来经常爆异常），其基本思路“延迟”跟懒是一样的。——译者注

实际上在Clojure中这是行不通的，因为(infinite-seq)上的递归会导致栈溢出。但要是加上一个结构，告诉Clojure不要疯狂递归，仅根据需要进行处理，是可以做到的。

不仅如此，你还能在一个线程内做到这一点，如下例所示。代码清单10-5中为某个数k定义了懒序列k, k+1, k+2, ...。

代码清单10-5 懒序列的例子

```
(defn next-big-n [n] (let [new-val (+ 1 n)]  
  (lazy-seq  
    (cons new-val (next-big-n new-val))))  
)  
  
(defn natural-k [k]  
  (concat [k] (next-big-n k)))  
  
1:57 user=> (take 10 (natural-k 3))  
(3 4 5 6 7 8 9 10 11 12)
```

(lazy-seq)形式是关键，它标记了发生无限递归的点，还有(concat)，可以安全地处理递归。然后你就可以用(take)形式从懒序列中取出所需的元素了，这个基本上是用(next-big-n)形式定义的。

懒序列是极其强大的特性，实践会告诉你它们是Clojure军火库中的强大武器。

10.4.2 序列和变参函数

Clojure函数有一个强大的特性，它天生就具备参数数量可变的能力，有时称为函数的变元(arity)。参数数量可变的函数称为变参函数(variadic)。

代码清单10-1中讨论过的函数(const-fun1)可以作为一个简单的例子。这个函数接受一个参数并抛弃它，总是返回值1。请看传入多个参数给(const-fun1)时会发生什么：

```
1:32 user=> (const-fun1 2 3)  
java.lang.IllegalArgumentException: Wrong number of args (2) passed to:  
user$const-fun1 (repl-1:32)
```

Clojure编译器仍然会对传给(const-fun1)的参数数量(和类型)做一些检查。对于简单地抛弃所有参数并返回一个常量值的函数来说，这似乎过于严格了。在Clojure中能接受任意数量参数的函数看起来会是什么样的呢？

代码清单10-6展示了如何实现一个这样的(const-fun1)常量函数。我们管它叫(const-fun-arity1)，变元的const-fun1。这是在Clojure标准函数库中(constantly)函数的自产版。

10

代码清单10-6 带有变元的函数

```
1:28 user=> (defn const-fun-arity1  
  ([] 1)  
  ([x] 1)  
  ([x & more] 1)  
)  
 #'user/const-fun-arity1
```

```

1:33 user=> (const-fun-arity1)
1
1:34 user=> (const-fun-arity1 2)
1
1:35 user=> (const-fun-arity1 2 3 4)
1

```

这个函数的定义不是一个参数向量后跟着函数行为的定义。而是有一系列这种组合，每个组合里都是一个参数向量（构成了这一版本函数的有效签名）和这一版本函数的实现。

这跟Java的方法重载类似。传统做法一般是定义几个特殊情况下的形式（没有参数、一个或两个参数）和最后一个参数为序列的额外形式。代码清单10-6中就是参数向量为[x & more]的那个。&符号表明这是该函数的变参版本。

序列是Clojure的创新。实际上，用Clojure编程主要就是要思考怎么用序列解决特定问题。

Clojure的另一项重要创新是Clojure和Java的集成，也就是我们下一节的主题。

10.5 Clojure与Java的互操作

Clojure从一开始就设计为JVM语言，并且不会对程序员完全隐藏JVM特性。这些特殊的设计在几个地方都有体现。比如在类型系统层面，Clojure的列表和向量都实现了Java集合类库中的标准接口List。另外，Clojure使用Java的类库非常容易，反之亦然。

这意味着Clojure程序员可以使用Java中丰富的类库和工具，以及JVM的性能和其他特性。这一节会涉及这种互操作性的几方面内容，特别是：

- 从Clojure中调用Java；
- Java如何见到Clojure函数的类型；
- Clojure代理；
- 用REPL做探索性编程；
- 从Java中调用Clojure。

我们先看看从Clojure中如何访问Java方法，开始它们的集成探索之旅吧。

10.5.1 从Clojure中调用Java

看一下这段在REPL中进行计算的Clojure代码：

```

1:16 user=> (defn lenStr [y] (.length (.toString y)))
#'user/lenStr
1:17 user=> (schwartz ["bab" "aa" "dgfwg" "droopy"] lenStr)
("aa" "bab" "dgfwg" "droopy")
1:18 user=>

```

这段代码用Schwartzian转换对一个字符串向量排序，排序标准是字符串的长度。其中用到了形式(.toString)和(.length)，这都是Java方法，它们是在Clojure对象上调用的。符号开始部分的句号.表示运行时应该在下一个参数上调用该名称的方法，底层是用(.)宏实现的。

所有用(def)或它的变体定义的Clojure值都被放在clojure.lang.Var实例中，它可以承载

任何 `java.lang.Object`, 所以任何可以在 `java.lang.Object` 调用的方法都可以在 Clojure 值上调用。另外一些跟 Java 交互的形式是用来调用静态方法的

```
(System/getProperty "java.vm.version")
( 此处是调用 System.getProperty() ) 和用于访问静态公共变量 ( 比如常量 ) 的
Boolean/TRUE
```

在后面两个例子中已经用到了 Clojure 命名空间的概念。跟 Java 包的概念类似，并且常用的 Java 包都有对应的映射缩写形式，比如前面那些。

Clojure 调用的本质

Clojure 中的函数调用实际上是 JVM 的方法调用。JVM 不能保证像类 Lisp 语言（特别是 Scheme）通常做的那样优化掉尾递归。JVM 上一些其他的 Lisp 方言觉得它们需要真正的尾递归，因此不准备把 Lisp 函数调用跟 JVM 方法调用完全等同起来。而 Clojure 完全以 JVM 为平台，甚至不惜违背通常的 Lisp 实践。

如果你想创建一个新的 Java 对象实例并在 Clojure 中操作它，用 `(new)` 形式就可以轻松做到。它还有个备选的缩写形式，在类名之后跟一个句号，可以归结为 `(.)` 宏的另一个用法：

```
(import '(java.util.concurrent CountDownLatch LinkedBlockingQueue))
(def cdl (new CountDownLatch 2))
(def lbg (LinkedBlockingQueue.))
```

这里还用了 `(import)` 形式，只用一行就可以导入一个包的很多 Java 类。

我们在前面提过，Clojure 的类型系统有些地方跟 Java 是一致的，我们来看看其中的细节。

10.5.2 Clojure 值的 Java 类型

从 REPL 中很容易看到某些 Clojure 值的 Java 类型：

```
1:8 user=> (.getClass "foo")
java.lang.String
1:9 user=> (.getClass 2.3)
java.lang.Double
1:10 user=> (.getClass [1 2 3])
clojure.lang.PersistentVector
1:11 user=> (.getClass '(1 2 3))
clojure.lang.PersistentList
1:12 user=> (.getClass (fn [] "Hello world!"))
user$eval110$fn_111
```

首先要看到所有 Clojure 值都是对象，JVM 的原始类型默认情况下是不对外的（尽管从性能角度来看有办法得到原始类型）。如你所料，字符串和数字值直接映射到对应的 Java 引用类型上了（`java.lang.String`、`java.lang.Double` 等）。

匿名的 "Hello world!" 函数的名字表明它是一个动态生成类的实例。这个类会实现

`clojure.lang.IFn` 接口，Clojure用该接口表明这个值是个函数，你可以把它当做`java.util.concurrent`里的`Callable`接口。

序列会实现`clojure.lang.ISeq`接口。它们通常是抽象类`ASeq`或懒实现`LazySeq`的具体子类。

我们已经看过几种值的类型了，但这些值是怎么保存的呢？就像我们在本章一开始提到的，`(def)`把符号绑到一个值上，这样会创建一个`var`。这些`var`是`clojure.lang.Var`类型（它所实现的接口中也有`IFn`）的对象。

10.5.3 使用Clojure代理

Clojure有一个强大的宏(`proxy`)，你可以用它创建扩展Java类(或实现接口)的Clojure对象。比如代码清单10-7重新实现了之前的一个例子(代码清单4-13)，由于Clojure语法更加紧凑，所以这个例子的核心代码只有一点。

代码清单10-7 重温调度执行者

```
(import '(java.util.concurrent Executors LinkedBlockingQueue TimeUnit))
(def stpe (Executors/newScheduledThreadPool 2))           ← STPE工厂方法
(def lbq (LinkedBlockingQueue.))

(def msgRdr (proxy [Runnable] []
  (run [] (.toString (.poll lbq))))                      ← 定义匿名的Runnable实现
)

(def rdrHndl
  ⇒ (.scheduleAtFixedRate stpe msgRdr 10 10 TimeUnit/MILLISECONDS))

(proxy)的一般形式是：

(proxy [<超类/接口>] [<args>] <命名函数的实现>+)
```

第一个向量参数是这个代理类应该实现的接口。如果这个代理还要扩展Java类(如果可以的话，当然，只能扩展一个Java类)，这个类名必须是向量中的第一个元素。

第二个向量参数包含传给超类构造方法的参数。这个向量经常是空的，并且如果(`proxy`)形式只是实现Java接口的话，那它肯定是空的。

这两个参数之后是一个或多个表示单个方法实现的形式，按接口的要求或超类指定的实现。

用(`proxy`)形式可以做出任何Java接口的简单实现。这促成了一种吸引人的可能性：用Clojure REPL作为实验Java和JVM代码的扩展游戏床。

10.5.4 用REPL做探索式编程

探索式编程的核心思想是减少要编写的代码量，因为Clojure的语法和REPL提供的实时互动环境，REPL不仅是探索Clojure编程的理想环境，也是学习Java类库的极佳选择。

我们来看一下Java列表实现。它们都有返回`Iterator`类型对象的`iterator()`方法。但

Iterator是个接口，所以你可能对真正的实现类型感到好奇。用REPL很容易找出答案：

```
1:41 user=> (import '(java.util ArrayList LinkedList))
java.util.LinkedList
1:42 user=> (.getClass (.iterator (ArrayList.)))
java.util.ArrayList$Itr
1:43 user=> (.getClass (.iterator (LinkedList.)))
java.util.LinkedList$ListItr
```

(import)形式从java.util包中导入了两个类。然后在REPL内用Java的getClass()方法。可以看到迭代器实际上是内部类提供的。也许你不应该对此感到吃惊，因为我们在10.4节讨论过，迭代器和它们的集合绑定很紧密，所以它们也许需要了解这些集合的内部实现细节。

在前面这个例子中值得注意的是，我们一个Clojure结构也没用，只用了一点语法。我们操作的所有东西实际上都是Java结构。尽管如此，我们还是假设你想用不同的方式，在Java程序里用Clojure。下一节将会向你展示如何实现这一目的。

10.5.5 在 Java 中使用 Clojure

Clojure的类型系统跟Java高度一致。Clojure数据结构全是真的Java集合，都实现了对接口的所有必需部分。因为接口的可选部分一般都跟修改数据结构有关，而Clojure数据结构不可变，所以一般都没实现。

类型系统的一致性使得在Java程序里使用Clojure数据结构成为可能。Clojure自身的性质加强了这种可行性——它是采用调用机制的JVM编译型语言。这最大限度地减少了运行时的问题，意味着从Clojure中得到的类几乎跟其他任何Java类一样。解释型语言跟Java的互操作会更加困难，并且通常需要最基本的非Java语言运行时支持。

下面这个例子展示了Clojure的seq结构如何用在一个普通的Java字符串中。要运行这段代码，需要把clojure.jar放在classpath上：

```
ISeq seq = StringSeq.create("foobar");
while (seq != null) {
    Object first = seq.first();
    System.out.println("Seq: " + seq + " ; first: " + first);
    seq = seq.next();
}
```

10

上面的代码使用了StringSeq类中的工厂方法create()。它给出了字符串中字符序列的seq视图。first()和next()方法返回新值，而不是修改已有的seq，就跟我们在10.4节讨论的一样。

截止目前我们只是在处理单线程的Clojure代码。下一节我们要谈论Clojure中的并发。特别是Clojure对状态和可变性的处理方式，这使得它的并发模型跟Java的差别很大。

10.6 Clojure 并发

Java的状态模型从根本上来说是基于对象可变思想的。正如第4章中所提及的，这会直接导致并发代码的安全问题。在一一线程修改对象的状态时，为了防止其他线程看到对象的中间（即不一致）状态，需要引入相当复杂的锁策略。这些策略理解难，调试难，测试更难。

Clojure的并发概念在某些方面不像Java中那么底层。比如说，由Clojure运行时管理线程池的使用（开发人员在这方面几乎或根本不能控制）看起来可能有点奇怪。但是让平台（此处即Clojure运行时）细致地做好内务工作的好处在于，开发人员可以专注于更重要的任务，比如总体设计。

Clojure的指导思想是默认把线程彼此隔开，这种实现并发安全的办法由来已久。假定“没有共享资源”的基线和采用不可变值使Clojure避开了很多Java所面临的问题，从而可以专注于为并发编程安全地共享状态的方法。

注意 为了帮助提升安全性，Clojure的运行时提供了线程协调机制，我们强烈建议你使用这些机制，而不是用Java的惯例或构造自己的并发结构。

实际上，Clojure用不同的方法实现了不同的并发模型：未来式（future）、并行调用（pcall）、引用形式（ref）和代理（agent）。且听我们一道来，先从最简单的开始。

10.6.1 未来式与并行调用

第一个也是最明显的一个状态分享办法就是不分享。实际上，我们一直使用的Clojure结构var本质上是不可以共享的。如果两个不同的线程继承了名字相同的var，并在线程里重新绑定了它，那绑定只在这些线程内部可见，绝不可能被其他线程共享。

可以利用Clojure跟Java的紧密结合启动新线程，也就是说在Clojure中写Java并发代码非常容易。但其中有些抽象在Clojure中有更干净的形式。比如对于第4章介绍的Java未来式（Future），Clojure提供了非常干净的方式。代码清单10-8是个简单的例子。

代码清单10-8 Clojure中的Future

```
user=> (def simple-future
  (future (do
    (println "Line 0")
    (Thread/sleep 10000)
    (println "Line 1")
    (Thread/sleep 10000)
    (println "Line 2"))))
#'user/simple-future
Line 0
user=> (future-done? simple-future)           ← 马上开始执行
user=> false
Line 1
user=> @simple-future                         ← 解引用导致阻塞
Line 2
nil
user=>
```

这段代码用(future)建立了一个Future。创建之后它马上就开始在后台线程中运行，所以在Clojure REPL中看到了输出Line 0（然后是Line 1）——代码已经开始在另一个线程上运行了。接着可以用(future-done?)来检查代码是否已经运行完，这个调用是非阻塞的。然而对

`future`的解引用会阻塞调用线程，直到函数完成。

这实际上是Clojure对Java Future的一个瘦封装，语法更干净。Clojure还提供了对并发程序员非常有帮助的辅助形式。有个简单的函数是(`pcalls`)，可以接受数量可变的零参函数，让它们并发执行。它们在运行时管理的线程池上执行，并返回一个懒序列结果。试图访问序列中的任何还没完成的元素会导致访问线程被阻塞。

代码清单10-9建立了一个单参函数(`wait-with-for`)。它用了一个类似10.3.2节介绍过的`loop`形式。可以用它创建一些零参函数(`wait-1`)、(`wait-2`)等，并把它们传给(`pcalls`)。

代码清单10-9 Clojure中的并行调用

```

user=> (defn wait-with-for [limit]
  (let [counter 1]
    (loop [ctr counter]
      (Thread/sleep 500)
      (println (str "Ctr=" ctr))
      (if (< ctr limit)
          (recur (inc ctr))
          ctr))))
#'user/wait-with-for
user=> (defn wait-1 [] (wait-with-for 1))
user=> #'user/wait-1
user=> (defn wait-2 [] (wait-with-for 2))
user=> #'user/wait-2
user=> (defn wait-3 [] (wait-with-for 3))
user=> #'user/wait-3
user=> (def wait-seq (pcalls wait-1 wait-2 wait-3))
#'user/wait-seq
Ctr=1
Ctr=1
Ctr=1
Ctr=2
Ctr=2
Ctr=2
Ctr=3

user=> (first wait-seq)
1
user=> (first (next wait-seq))
2

```

因为线程睡眠值只有500毫秒，等待函数很快就能完成。通过调整超时（比如延迟到10秒），很容易验证由(`pcalls`)返回的懒序列`wait-seq`是否有上面描述的那种阻塞行为。

对于不需要共享状态的情况，这种简单的多线程结构挺好，但在很多应用中，不同的处理线程都要在运行过程中相互通信。Clojure有几个模型可以处理这种情况，接下来我们先看看其中的一个：借助(`ref`)形式实现的状态共享。

10.6.2 ref形式

`ref`是Clojure在线程间共享状态的办法。它们基于运行时提供的一个模型，在这个模型中，状

态的改变要能被多个线程见到。该模型在符号和值之间引入了一个额外的中间层。也就是说，符号绑定到值的引用上，而不是直接绑到值上。这个系统基本上是事务化的，并且由Clojure运行时进行协调。如图10-6所示。

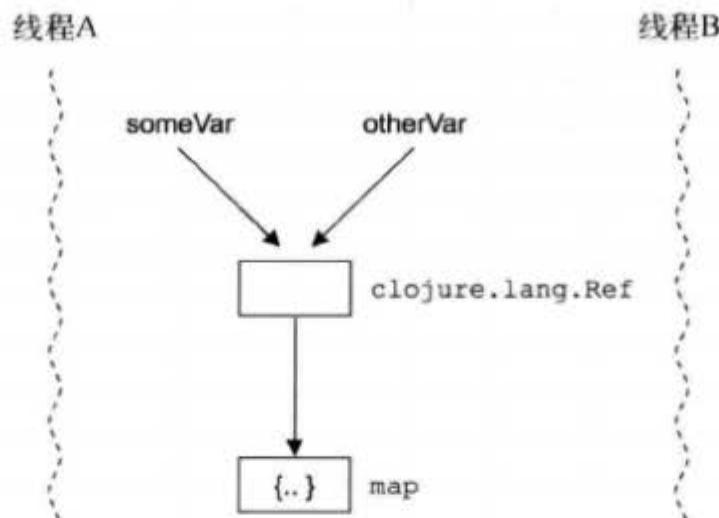


图10-6 软件事务内存

这一中间层意味着改变或更新ref之前必须把它放在一个事务中。当事务完成的时候，或者全变了，或者什么也没变。这跟数据库中的事务是类似的。

这可能有点抽象了，所以我们来看一个模拟ATM的例子。在Java中，要对所有敏感数据加锁保护。代码清单10-10是一个简单的自动提款机模型，包括锁。

代码清单10-10 Java中的ATM模型

```

public class Account {
    private double balance = 0;
    private final String name;
    private final Lock lock = new ReentrantLock();

    public Account(String name_, double initialBal_) {
        name = name_;
        balance = initialBal_;
    }

    public synchronized double getBalance() {
        return balance;
    }

    public synchronized void debit(double debitAmt_) {
        balance -= debitAmt_;
    }

    public String getName() {
        return name;
    }

    public String toString() {
        return "Account [balance=" + balance + ", name=" + name + "]";
    }
}

```

```

public Lock getLock() {
    return lock;
}
}

public class Debitter implements Runnable {
    private final Account acc;
    private final CountDownLatch cdl;

    public Debitter(Account account_, CountDownLatch cdl_) {
        acc = account_;
        cdl = cdl_;
    }

    public void run() {
        double bal = acc.getBalance();
        Lock lk = acc.getLock();

        while (bal > 0) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) { }
            lk.lock();
            bal = acc.getBalance(); ← 必须重新取得余额
            if (bal > 0) {
                acc.debit(1);
                bal--;
            }
            lk.unlock();
        }
        cdl.countDown();
    }
}

Account myAcc = new Account("Test Account", 500 * NUM_THREADS);
CountDownLatch stopl = new CountDownLatch(NUM_THREADS);

for (int i=0; i<NUM_THREADS; i++) {
    new Thread(new Debitter(myAcc, stopl)).start();
}
stopl.await();
System.out.println(myAcc);

```

再来看看用Clojure怎么写。先来个单线程版本。然后我们再开发一个并发版本跟单线程版本比较，这样并发代码应该更容易理解。

代码清单10-11是单线程版本。

代码清单10-11 Clojure中的简单ATM模型

```

(defn make-new-acc [account-name opening-balance]
  {:name account-name :bal opening-balance})

(defn loop-and-debit [account]
  (loop [acc account]
    (let [balance (:bal acc) my-name (:name acc)]

```

```
(Thread/sleep 1)
(if (> balance 0)
  (recur (make-new-acc my-name (dec balance))) ← 用循环/递归代替Java
  acc
  ))))

(loop-and-debit (make-new-acc "Ben" 5000))
```

这段代码跟Java版比起来非常紧凑。必须承认，这是单线程的，但还是比Java的代码少了很多。运行代码会得到期望的结果：一个余额为0的acc映射。现在我们看看并发形式。

要让这段代码并行，需要引入ref。它们是用(ref)形式创建的，并且类型为clojure.lang.Ref的JVM对象。通常建立时会带一个保存状态的映射，此外还需要(dosync)形式来设置事务。在事务之内，还要用到(alter)形式来修改ref。使用ref的多线程ATM函数如代码清单10-12所示。

代码清单10-12 多线程ATM

```
(defn make-new-acc [account-name opening-balance]
  (ref {:name account-name :bal opening-balance}))

(defn alter-acc [acc new-name new-balance]
  (assoc acc :bal new-balance :name new-name)) ← 必须返回值，而不是引用

(defn loop-and-debit [account]
  (loop [acc account]
    (let [balance (:bal @acc)
          my-name (:name @acc)]
      (Thread/sleep 1)
      (if (> balance 0)
        (recur (dosync (alter acc alter-acc my-name (dec balance)) acc))
        acc
        ))))

(def my-acc (make-new-acc "Ben" 5000))

(defn my-loop []
  (let [the-acc my-acc]
    (loop-and-debit the-acc)
    ))

(pcalls my-loop my-loop my-loop my-loop my-loop)
```

就像注释中说的，对值进行操作的(alter-acc)函数必须返回一个值。所操作的值是对当前事务中线程可见的本地值，这称为事务内的值。返回的值是在变更函数返回之后的ref新值。在退出(dosync)所定义的事务块之前，这个值对外界是不可见的。

与此同时，其他事务可能像这个一样也在进行。如果是这样，Clojure STM系统会进行跟踪，并且只允许那些自开始以来已经提交过的事务组成的事物提交。如果不一致，它会回滚，并且可能在得到更新过状态后再次尝试。

如果事物做了任何会产生副作用的事情（比如日志文件或其他输出），这个重试行为可能会引发问题。让事物化部分在函数式编程中（即没有副作用）尽可能地保持简单纯粹是你的责任。

对于某些多线程方式而言，这种持乐观态度的事务行为看起来可能是相当重量级的做法。有些并发应用只需偶尔在线程间进行通信，并且是以相当不对称的风格。幸运的是，Clojure提供了另外一种更好地体现“过后就忘”原则的并发机制，这也是我们下一节的主题。

10.6.3 代理

代理是Clojure中异步的、面向消息的并发原语。Clojure代理不是共享状态，而是属于另外一个线程的一点儿状态，但它会从另外一个线程中接收消息（以函数的形式）。这乍看起来可能是个奇怪的想法，尽管遇到过Scala的actor之后这种感觉可能会少一点。

“我离它们太远了，只能把礼物装进包裹寄给它们，”她想，“这也太滑稽了，给自己的双脚送礼物还需要邮寄！地址写起来就更有趣了！”

——《爱丽丝梦游仙境》，刘易斯·卡罗尔

应用到代理上的函数在代理的线程上运行。这个线程是由Clojure运行时管理的，在一个程序员通常无法访问的线程池里。运行时还会保证代理中那些可以被外界看到的值是孤立的和原子的。这就是说用户代码只会见到状态修改之前或之后的代理值。

代码清单10-13是个简单的代理例子，跟用来讨论future的例子类似。

代码清单10-13 Clojure代理

```
(defn wait-and-log [coll str-to-add]
  (do (Thread/sleep 10000)
       (let [my-coll (conj coll str-to-add)]
         (Thread/sleep 10000)
         (conj my-coll str-to-add))))
(def str-coll (agent []))
(send str-coll wait-and-log "foo")
@str-coll
```

send调用派发了一个(wait-and-log)调用给代理，通过使用REPL解引用，结果就像承诺的那样，你绝不会看到代理的中间状态——只有最后的状态出现了（字符串"foo"被添加了两次）。

实际上，代码清单10-13上的(send)调用很容易让人联想到爱丽丝的脚的地址。刘易斯·卡罗尔很可能是用Clojure代码写的地址：

爱丽丝的右脚收
壁炉前的毛毯上
靠近挡板
(带去爱丽丝的爱)

在你认为一个人的脚是身体的有机组成时，这的确挺怪异的。同样，发消息给Clojure管理的线程池中一个线程上的代理看起来也挺怪异的，两个线程还共享一个地址空间。但你目前多次遇到的一个并发主题就是如果它能让用法更加简单清晰，额外的复杂性可能是件好事。

10.7 小结

作为一门语言，Clojure可以说是我们见过的几门语言中跟Java差别最大的。它对Lisp的传承、对不可变性的强调以及独特的编程方式，让它看起来变成了完全独立的语言。但它和JVM的紧密结合、与类型系统的一致性（即便它提供了序列等替代方案），还有探索式编程的能力，让它成为与Java互补性非常强的一门语言。

任何地方的协同都没有Clojure运行时对线程和并发底层特性的代理控制更清晰。这让程序员可以放手去关注多线程的设计和高层问题。这就跟Java的垃圾收集设施可以让你无需关心内存管理的细节一样。

本部分研究的不同语言间的差别展示了Java平台的进化能力，并且证明了它仍然是应用开发的理想目标。这也是对JVM灵活性和性能的证明。

在本书的最后一部分，我们会向你展示三门新语言为软件工程实践提供的新方式。下一章全部是关于测试驱动开发的内容——你在Java世界中很可能已经碰到过这一主题了。但Groovy、Scala和Clojure提供了全新的视角，有望巩固和加强你已经知道的那些东西。

Part 4

第四部分

多语种项目开发

在最后一部分，我们会把已经学到的平台和多语言编程知识应用到现代软件开发中最常见和最重要的技术上。

要成为一名优秀的 Java 开发人员，不仅仅是掌握 JVM 和它上面跑的语言那么简单。要成功交付软件，还要遵循业界最佳实践。幸好，这些实践中有相当一部分是从 Java 生态系统中开始的，所以我们有很多东西可以聊。

我们会用一整章的内容讨论测试驱动开发（TDD）的基础知识，以及如何把测试概念应用到极其复杂的测试场景中。另一章会集中讨论如何将正规的构建生命周期引入构建流程中，包括持续集成技术。这两章会介绍一些工具，比如用于测试的 JUnit、用于构建的 Maven，以及用于持续集成的 Jenkins。

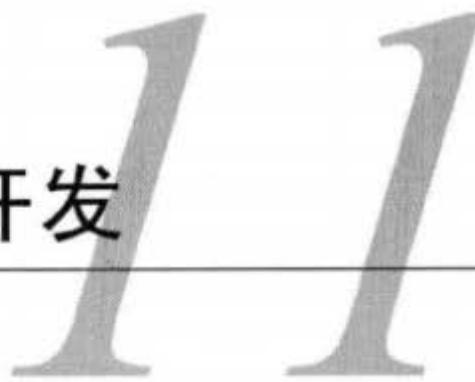
我们还会讨论 Java 7 时代的 Web 开发，会涉及为项目选择最适合框架的标准，还有如何在这个环境中快速开发。

如果你看过第三部分，应该了解非 Java 语言在 TDD、构建生命周期和快速 Web 开发领域都有举足轻重的作用。无论是用于 TDD 的 ScalaTest 框架，或者用于构建 Web 应用的 Grails（Groovy）和 Compojure（Clojure）框架，Java/JVM 生态系统中的很多方面都受到了这些新语言的影响。

我们会向你展示如何把新语言的力量作用到你所熟悉的软件开发工艺上。与 JVM 坚实的基础和 Java 生态系统结合为一个整体，你会发现那些接受多语言观点的开发人员可能会收获颇丰。

最后一章我们会看一看平台的未来，并预测一下将来。第四部分全是前沿内容，所以现在就让我们翻开新的一页，向着地平线推进吧！

测试驱动开发



本章内容

- 实行测试驱动开发的好处
- TDD的核心：红—绿—重构循环周期
- JUnit，公认的Java测试框架
- 四种测试替身：虚设、伪装、存根和模拟
- 用内存数据库测试DAO代码
- 用Mockito模拟子系统
- 使用Scala测试框架ScalaTest

测试驱动开发（TDD）进入软件开发行业已经有相当长的时间了。它的基本前提是在编写真正的功能实现代码之前先写测试代码，然后根据需要重构实现代码。比如要写一段拼接两个String对象（"foo"和"bar"）的实现代码，应该先写测试代码（测试结果必须等于"foobar"），以确保你能判断实现是否正确。

很多开发人员都知道JUnit，也会在开发时不定期用到它。但他们一般是写完实现代码之后才编写测试代码，因此体会不到TDD的益处。

尽管TDD的概念看起来非常普及，但实际上很多开发人员并不清楚为什么要采用TDD。对于很多开发人员来说，“为什么要写测试驱动代码以及有什么好处”一直是个问题。

我们认为消除恐惧和不确定性是编写测试驱动代码的重要原因。Kent Beck（JUnit测试框架的发明人之一）在*Test-Driven Development: by Example^①*（Addison-Wesley Professional，2002）一书中对此总结得很好：

- 恐惧会让你小心试探；
- 恐惧会让你尽量减少沟通；
- 恐惧会让你羞于得到反馈；
- 恐惧会让你脾气暴躁。

TDD可以祛除恐惧，让优秀的Java开发者变得更加自信、善于沟通、乐于接受并更加快乐。

^① 中文版《测试驱动开发》已由中国电力出版社于2004年出版。——编者注

换句话说，TDD能帮你摆脱下面这种心态：

- 在开始新工作时，“我不知道从哪里开始，所以只好将就着做一些修改”；
- 在修改已有代码时，“我不知道现有代码怎么运行，所以我私下认为不能碰它们”。

TDD带来的很多好处并不会马上显现：

- 更清晰的代码——只写需要的代码；
- 更好的设计——有些开发人员管TDD叫测试驱动的设计；
- 更出色的灵活性——TDD鼓励按接口编码；
- 更快速的反馈——不会直到系统上线才知道bug的存在。

刚入门的开发人员有时认为TDD不是“普通”开发人员用的技术，这是他们采用TDD的一个障碍。他们的感觉是只有那些想象中的“敏捷派”或其他神秘组织的成员才会用TDD。这种认识完全错误，我们会在后面解释。TDD是给所有开发人员使用的技术。

另外，敏捷和软件工艺运动都是为了让开发人员活得更轻松。它们肯定不会拒绝别人使用TDD或其他任何技术。

本章首先解释TDD背后的基本思想红—绿—重构循环，然后介绍Java测试框架中的主力JUnit，并用一个简单的例子来阐明其原则。

敏捷宣言和软件工艺运动

敏捷运动（<http://agilemanifesto.org/>）已经开展很长时间了，可以说部分改善了软件开发行业。很多伟大的技术，比如TDD，都是这项运动所倡导的。软件工艺是一项新运动，鼓励参与者编写清晰的代码（<http://manifesto.softwarecraftsmanship.org/>）。

我们喜欢取笑实行敏捷和软件工艺运动的弟兄们。可是，我们自己甚至也拥护它（大多数时候都是如此）。但优秀的Java开发人员，请不要忽视那些对你有用的东西。TDD是一项软件开发技术，仅此而已。

接下来，我们会介绍TDD使用的四大类伪装对象。它们能简化受试代码和第三方类库中代码的隔离，或隔离数据库之类的子系统行为，所以它们很重要。随着依赖项变得越来越复杂，伪装对象也要变得越来越聪明。最终我们会介绍模拟和Mockito类库，它是一个流行的模拟工具，可以让开发人员在不受外部系统影响的环境下进行测试。

开发人员非常熟悉Java测试框架（特别是JUnit），并且一般都有用它们编写测试代码的经验。但对于如何用测试驱动Scala、Clojure等新语言，你可能毫无头绪。因此我们会介绍Scala测试框架ScalaTest，以确保你能在开发Scala代码时应用TDD。

让我们开始了解这个有点奇怪的TDD吧。

11.1 TDD 概览

TDD可以应用在多个层级上。表11-1列出了通常会采用TDD的四个测试层级。

表11-1 TDD的测试层级

层 级	描 述	例 子
单元测试	通过测试验证一个类中包含的代码	测试BigDecimal类中的方法
集成测试	通过测试验证类之间的交互	测试Currency类以及它如何跟BigDecimal交互
系统测试	通过测试验证运行的系统	从UI到Currency类测试会计系统
系统集成测试	通过测试验证运行的系统，包括第三方组件	测试会计系统，包括它与第三方报表系统间的交互

在单元测试中使用TDD是最容易的，如果你对TDD不熟悉，这一层就是个很好的起点。本节主要讲述如何在单元测试层中使用TDD。后续章节会讨论其他层级，包括第三方组件和子系统的测试。

提示 处理没有或只有很少测试的遗留代码是个恐怖的任务。我们几乎不可能把所有测试都追加上，因此，应该只是为添加的新功能加上测试代码。请参阅Michael Feathers的*Working Effectively with Legacy Code*^① (Prentice Hall, 2004) 获取更多帮助。

我们一开始会简单介绍一下TDD的基本前提——红—绿—重构循环——用JUnit测试计算剧院门票销售收入的代码^②。只要遵照红—绿—重构循环，基本上就可以使用TDD！之后我们会探究一下红—绿—重构循环背后的思想，让你对为什么应该采用这种技术有更清楚地认识。最后我们将介绍JUnit这个公认的Java开发者测试框架，讲解它的基本用法。

让我们开始吧，先来一个TDD三步（红—绿—重构）测试计算剧院门票销售收入的实际例子。

11.1.1 一个测试用例

如果你有TDD方面的经验，可以自行决定是否跳过这一节，不过这个小例子中有些新东西。假定有人要你写一个坚若磐石的方法来计算剧院门票的销售收入。剧院会计最初给出的业务规则很简单：

- 门票的底价是30美元；
- 总收入=售出票数*价格；
- 剧院有100个座位。

因为剧院工作人员不懂软件，所以他们现在还必须手工录入门票的销售数量。

如果你做过TDD，应该知道它的三个基本步骤：红、绿、重构。如果刚接触TDD，或者想复习一下，那就请看一下Kent Beck在《测试驱动开发》中对这些步骤的定义：

^① 中文版《修改代码的艺术》已由人民邮电出版社于2007年出版（更多信息请参见<http://www.ituring.com.cn/book/536>）。

——编者注

^② 销售剧院门票在我的家乡伦敦是个大生意，最起码在我们写这本书的时候是。

- (1) 红，写一些不能用的测试代码（失败测试）；
- (2) 绿，尽快让测试通过（通过测试）；
- (3) 重构，消除重复（经过细化的通过测试）。

为了让你了解TicketRevenue应该达到什么效果，请先看一下这些伪代码。

```
estimateRevenue(int numberOfTicketsSold)
if (numberOfTicketsSold is less than 0 OR greater than 100)
then
    Deal with error and exit
else
    revenue = 30 * numberOfTicketsSold;
    return revenue;
endif
```

注意，千万别太深入。测试最终会驱动设计，也会部分影响实现。

注意 我们在11.1.2节会涉及开始失败测试的办法，但在这个例子中我们准备写一个甚至还无法编译的测试！

接下来我们先用JUnit写一个失败单元测试。如果你不了解JUnit，请跳到11.1.4节，然后再回来。

1. 编写失败测试（红）

这一步的要点是以一个会失败的测试开始。实际上，这个测试甚至无法编译，因为你还没有TicketRevenue类！

在跟会计开过一个简短的白板会议后，你意识到测试代码需要覆盖五种情况：售票数量为负数、0、1、2~100，还有大于100。

提示 编写测试代码（特别是牵扯到数值时）有一个很好的经验法则，要考虑值为0/null、1和很多(N)的情况。再进一步考虑 N 上的其他限制，比如数量为负或超出上限。

我们决定先写一个测试覆盖销售一张门票收入的情况。测试代码看起来应该如代码清单11-1所示（记住这个阶段不用编写完美的通过测试）。

代码清单11-1 为TicketRevenue编写的失败单元测试

```
import java.math.BigDecimal;
import static junit.framework.Assert.*;
import org.junit.Before;
import org.junit.Test;
public class TicketRevenueTest {

    private TicketRevenue venueRevenue;
    private BigDecimal expectedRevenue;

    @Before
    public void setUp() {
        venueRevenue = new TicketRevenue();
    }
```

```

    @Test
    public void oneTicketSoldIsThirtyInRevenue() {
        expectedRevenue = new BigDecimal("30");
        assertEquals(expectedRevenue, venueRevenue.estimateTotalRevenue(1));
    }
}

```

销售一张票的情况

测试期望销售一张门票得到的收入等于30。

但这个测试不能编译，因为有`estimateTotalRevenue(int numberOfTicketsSold)`方法的`TicketRevenue`类还不存在呢。为了运行测试，可以先随便写一个让测试可以编译的实现。

```

public class TicketRevenue {
    public BigDecimal estimateTotalRevenue(int i) {
        return BigDecimal.ZERO;
    }
}

```

现在测试代码能编译了，你可以在自己喜欢的IDE中运行它。每种IDE都有自己运行JUnit测试的办法，但一般都能在选中测试类后，从右键弹出菜单中选择运行测试。一旦运行，IDE一般都会更新窗口告诉你测试失败了，因为你所期望的30和`estimateTotalRevenue(1)`返回的值不符，它的返回值是0。

失败测试有了，接下来该做通过测试了（变绿）。

2. 编写通过测试（绿）

这一步的要点是让测试通过，但没必要把实现做到完美。给`TicketRevenue`类一个更好的`estimateTotalRevenue`实现（不会只返回0），可以让测试通过（变绿）。

记住，这一阶段只要让测试通过就行，没必要追求完美。代码可能如代码清单11-2所示：

代码清单11-2 第一版通过测试的TicketRevenue

```

import java.math.BigDecimal;

public class TicketRevenue {

    public BigDecimal estimateTotalRevenue(int numberOfTicketsSold) {
        BigDecimal totalRevenue = BigDecimal.ZERO;
        if (numberOfTicketsSold == 1) {
            totalRevenue = new BigDecimal("30");
        }
        return totalRevenue;
    }
}

```

通过测试的实现

现在再运行测试，通过了！而且在大多数IDE中，会用一个绿条或对勾来表示测试通过。图11-1是在Eclipse中通过测试的界面。

接下来的问题是你能不能说“我搞定了”，然后去做下一项工作？我们可以负责任地告诉你：“不是！”你会忍不住想完善前面的代码，那现在我们就开始吧。

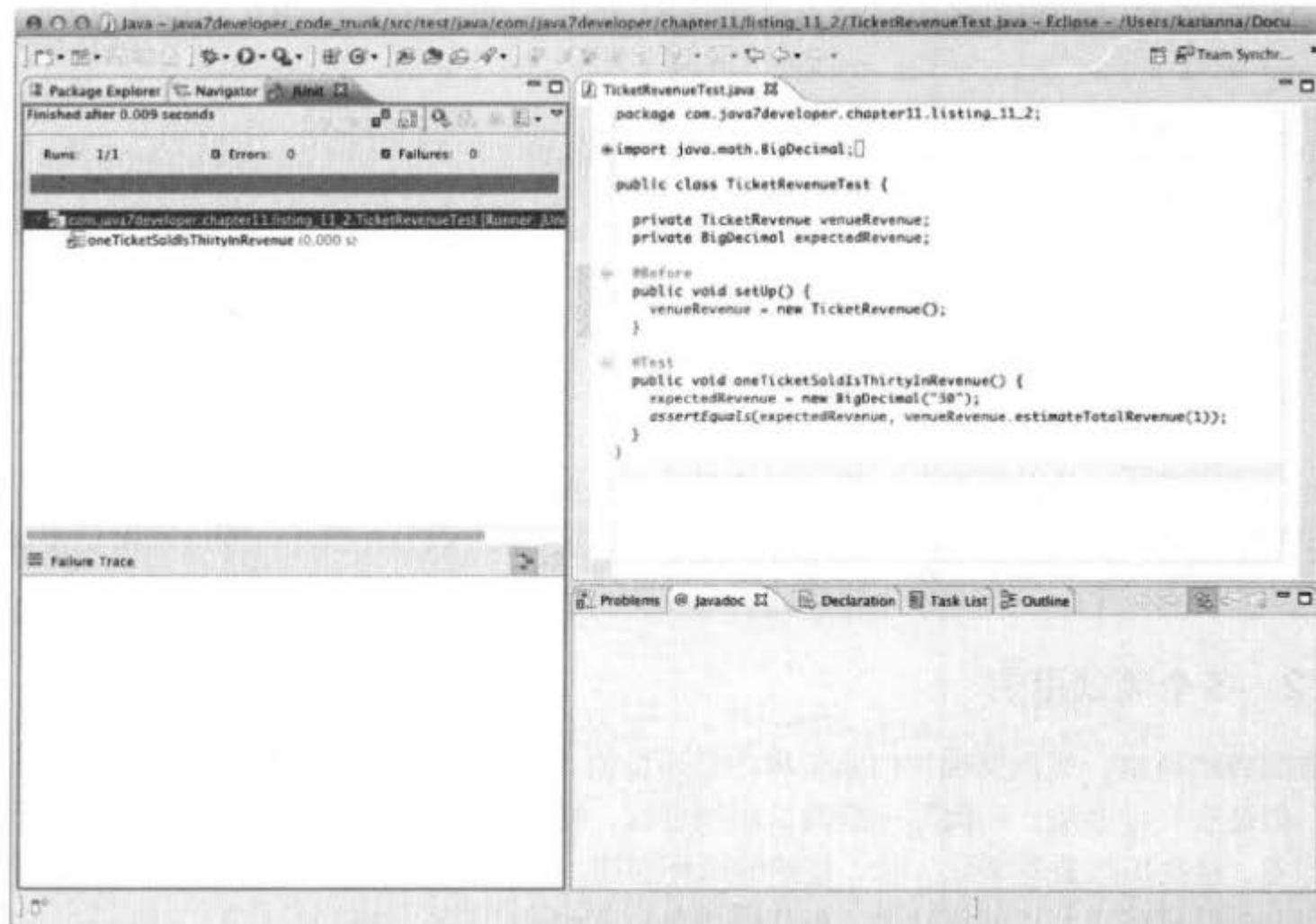


图11-1 Eclipse IDE中表示测试通过的绿条，纸质版印刷出来是中度灰色

3. 重构测试

这一步的要点是看看为了通过测试写的快速实现，确保你遵循了通行的惯例。代码清单11-2中的代码明显可以更清晰、更整洁。你肯定要重构，以减轻自己和他人的技术债务。

技术债务 Ward Cunningham发明的说法，指我们现在临时凑合出来的设计或代码将来会让我们付出更多的成本（工作）。

记住，有了通过测试，可以放心大胆地重构。应该实现的业务逻辑不可能会被忽视。

提示 编写最初的通过测试代码的另一个好处是开发进度可以更快。团队中的其他人可以马上用第一版代码跟更大的代码库一起测试（集成测试及更大范围的测试）。

在代码清单11-3中，我们不想再用魔法数字了——要让票价（30）出现在代码中。

代码清单11-3 通过测试的TicketRevenue重构版

```
import java.math.BigDecimal;
```

```

public class TicketRevenue {
    private final static int TICKET_PRICE = 30;           ← 不用魔法数字了
    public BigDecimal estimateTotalRevenue(int numberOfTicketsSold) {
        BigDecimal totalRevenue = BigDecimal.ZERO;
        if (numberOfTicketsSold == 1) {
            totalRevenue =
                new BigDecimal(TICKET_PRICE *
                    numberOfTicketsSold);           ← 重构的计算
        }
        return totalRevenue;
    }
}

```

经过这次重构，代码得到了改善，但很明显它还没有涵盖所有情况（售票数量为负值、0、2~100和大于100）。你不能只是拼命地猜其他情况下的实现应该是什么样，而应该做更多测试驱动的设计和实现。下一节会继续按照测试驱动设计的方式，带你看更多的测试用例。

11.1.2 多个测试用例

按照TDD风格，应该继续为门票销售数量为负值、0、2~100和大于100的情况依次添加测试用例。但还有一种办法，一次写一组测试用例也行，特别是在它们跟最初的测试有关的时候。

注意，这次仍然要遵循红—绿—重构的循环周期。在把这些用例都加上之后，你应该会得到一个带有失败测试（红）的测试类，如代码清单11-4所示。

代码清单11-4 TicketRevenue的失败单元测试

```

import java.math.BigDecimal;
import static junit.framework.Assert.*;
import org.junit.Test;

public class TicketRevenueTest {
    private TicketRevenue venueRevenue;
    private BigDecimal expectedRevenue;

    @Before
    public void setUp() {
        venueRevenue = new TicketRevenue();
    }

    @Test(expected=IllegalArgumentException.class)
    public void failIfLessThanZeroTicketsAreSold() {           ← 销量为负值
        venueRevenue.estimateTotalRevenue(-1);
    }

    @Test
    public void zeroSalesEqualsZeroRevenue() {                  ← 销量为0
        assertEquals(BigDecimal.ZERO, venueRevenue.estimateTotalRevenue(0));
    }

    @Test
    public void oneTicketSoldIsThirtyInRevenue() {           ← 销量为1
        assertEquals(new BigDecimal("30"), venueRevenue.estimateTotalRevenue(1));
    }
}

```