

```

expectedRevenue = new BigDecimal("30");
assertEquals(expectedRevenue, venueRevenue.estimateTotalRevenue(1));
}

@Test
public void tenTicketsSoldIsThreeHundredInRevenue() {           ← 销量为N
    expectedRevenue = new BigDecimal("300");
    assertEquals(expectedRevenue, venueRevenue.estimateTotalRevenue(10));
}

@Test(expected=IllegalArgumentException.class)
public void failIfMoreThanOneHundredTicketsAreSold() {           ← 销量大于100
    venueRevenue.estimateTotalRevenue(101);
}
}

```

为通过所有测试（绿）写的基本实现版看起来应该如代码清单11-5所示。

### 代码清单11-5 通过测试的第一版TicketRevenue

```

import java.math.BigDecimal;

public class TicketRevenue {

    public BigDecimal estimateTotalRevenue(int numberOfTicketsSold)
        throws IllegalArgumentException {

        BigDecimal totalRevenue = null;
        if (numberOfTicketsSold < 0) {
            throw new IllegalArgumentException("Must be > -1");           ← 异常情况
        }
        if (numberOfTicketsSold == 0) {
            totalRevenue = BigDecimal.ZERO;
        }
        if (numberOfTicketsSold == 1) {
            totalRevenue = new BigDecimal("30");
        }
        if (numberOfTicketsSold == 101) {
            throw new IllegalArgumentException("Must be < 101");           ← 销量为N
        }
        else {
            totalRevenue =
                new BigDecimal(30 * numberOfTicketsSold);           ← 销量为N
        }
        return totalRevenue;
    }
}

```

有了刚刚完成的实现，现在你的测试就变成通过测试了。

按照TDD循环周期，现在该重构这个实现了。比如说，可以把不合法的numberOfTicketsSold情况（负数或者大于100）放到一个if语句中，并用公式(TICKET\_PRICE \* numberOfTicketsSold)返回所有合法numberOfTicketsSold的收入。代码清单11-6应该跟重构之后的代码很像。

### 代码清单11-6 重构后的TicketRevenue版本

```
import java.math.BigDecimal;
```

```

public class TicketRevenue {
    private final static int TICKET_PRICE = 30;
    public BigDecimal estimateTotalRevenue(int numberOfTicketsSold)
        throws IllegalArgumentException {
        if (numberOfTicketsSold < 0 || numberOfTicketsSold > 100) {
            throw new IllegalArgumentException
                ("# Tix sold must == 1..100");
        }
        return new BigDecimal
            (TICKET_PRICE * numberOfTicketsSold);
    }
}

```

新的TicketRevenue类更加紧凑，并且还通过了所有测试！现在你已经完成了整个红—绿—重构循环，可以信心满满地开始实现下一个业务逻辑了。另外，如果你（或会计）发现漏掉了任何边界情况，比如有浮动票价，也可以再次开始一个循环。

我们强烈建议你弄明白红—绿—重构的TDD方式背后的原理，也就是我们接下来要讨论的内容。但如果你没什么耐心，可以直接跳到11.1.4节学习JUnit，或11.2节了解用来测试第三方代码的测试替身。

### 11.1.3 深入思考红—绿—重构循环

这一节会在前面例子的基础上探索TDD背后的一些思想。我们会再次谈论红—绿—重构循环，你应该还记得第一步是写失败测试。但这也有几种不同的方式。

#### 1. 失败测试（红）

一些开发人员真的喜欢编写编译失败的测试，喜欢等到绿色步骤才提供实现代码。也有一些开发人员喜欢先把测试调用的方法存根写出来，这样虽然测试代码能编译，但还是会失败。我们觉得怎么样都行，随意就好。

**提示** 这些测试代码是实现的第一个客户，所以应该认真考虑该怎么设计它们：方法定义看起来应该是什么样的。还应该问自己几个问题：该传什么参数进去？期望的返回值是什么？会不会有异常情况？另外，不要忘了测试重要领域对象的equals()和hashCode()方法。

一旦写完失败测试，就该进入下一阶段了：让它通过。

#### 2. 通过测试（绿）

这一步应该尽量少写代码，只要保证测试通过就行。也就是说你不用把实现做到完美，那是重构阶段的工作。

测试通过之后，你就可以告诉同事，你的代码已经实现了它应该实现的功能，他们可以拿去用了。

#### 3. 重构

在这一步中应该重构实现代码。可以重构的地方数不胜数，但有几个应该重点关注的，比如

去掉硬编码的变量或把大方法拆分开。如果是面向对象的代码，则应该遵循SOLID原则。

SOLID原则是Bob大叔（Robert Martin）提出来的，请参见表11-2。要了解更详细的信息，可以参考他的文章“The Principles of OOD”(<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>)。

表11-2 面向对象代码的SOLID原则

原 则	描 述
单一职责原则（SRP）	每个对象都应该做一件事，并且只做一件事
开放/封闭原则（OCP）	对象应该是可扩展、但不可修改的
里氏替换原则（LSP）	对象应该可以被它的子类型实例替换
接口隔离原则（ISP）	特定的小接口更好
依赖倒置原则（DIP）	不要依赖具体实现（请参见第3章关于依赖注入的内容）

**提示** 我们还要向你推荐Checkstyle和FindBugs这两个静态代码分析工具（第12章还有更多）。

Joshua Bloch的*Effective Java, Second Edition*<sup>①</sup> (Addison-Wesley, 2008) 也是好资源，其中有很多Java语言的技巧和窍门。

测试代码本身的重构是个容易被人遗忘的角落。你可以把通用的设置和拆卸代码提取出来，可以重命名测试以更准确地反应它的测试意图，还可以根据静态分析工具的分析结果做些小修订。

现在你已经能跟上TDD的三步走了，该去熟悉一下JUnit了，它可是在Java里写TDD代码的默认工具。

#### 11.1.4 JUnit

JUnit是公认的Java项目测试框架。当然，除了JUnit还有其他测试框架，比如拥有不少追随者的TestNG，但目前JUnit还是Java测试界的主流。

**注意** 如果你熟悉JUnit，可以跳到11.2节。

JUnit有三个主要特性：

- 用于测试预期结果和异常的断言，比如`assertEquals()`；
- 设置和拆卸通用测试数据的能力，比如`@Before`和`@After`；
- 运行测试套件的测试运行器。

JUnit用简单的注解模型提供了很多重要的功能。

大多数IDE（比如Eclipse、IntelliJ和NetBeans）都内置了JUnit，如果你用的正好是其中之一，就不用自己去下载、安装或配置JUnit了。如果你的IDE没有安装JUnit，可以访问[www.junit.org](http://www.junit.org)查

<sup>①</sup> 《Effective Java中文版》由机械工业出版社于2003年出版。——编者注

看它的下载和安装指导<sup>①</sup>。

**注意** 我们用的是JUnit 4.8.2。如果你要练习本章中的例子，建议也用这个版本。

一个基本的JUnit测试包含下面这些元素：

- 用@Before标记设置方法，在每个测试运行前准备测试数据；
- 用@After标记拆卸方法，在每个测试运行完成后拆卸测试数据；
- 测试方法本身（用@Test注解标记）。

为了多了解一下上面这些元素，我们来看几个非常基本的JUnit测试。

比如OpenJDK团队要你给BigDecimal类写个单元测试。第一个测试是检查加法( $1.5 + 1.5 == 3.0$ )，第二个测试是检查用非数字值创建BigDecimal实例时会抛出NumberFormatException异常。

**注意** 我们在本章的例子中经常同时给出多个失败测试，实现（绿）和重构。这违背了纯粹的TDD单个测试贯穿红—绿—重构循环的原则，但却可以让我们在本章中放入更多例子。不过在你编码时，应该尽可能地遵守单个测试循环的开发模型。

要运行代码清单11-7，可以在IDE里的源码文件上点击右键，选择运行或测试选项（三个主流IDE中都有显眼的Run Test或Run File选项）。

### 代码清单11-7 JUnit测试的基本结构

```
import java.math.BigDecimal;
import org.junit.*;
import static org.junit.Assert.*;

public class BigDecimalTest {
    private BigDecimal x;

    @Before
    public void setUp() { x = new BigDecimal("1.5"); } ① 每个测试之前的设置

    @After
    public void tearDown() { x = null; } ② 每个测试之后的拆卸

    @Test
    public void addingTwoBigDecimals() {
        assertEquals(new BigDecimal("3.0"), x.add(x));
    } ③ 执行测试

    @Test(expected=NumberFormatException.class)
    public void numberFormatExceptionIfNotANumber() {
        x = new BigDecimal("Not a number");
    } ④ 处理意料中的异常
}
```

<sup>①</sup> 第12章会讲到JUnit和Maven的集成。

在每个测试运行之前，`x`在`@Before`区域中被设置为`BigDecimal("1.5")`**①**。这会确保每个测试处理的都是已知值`x`，而不是被之前运行的测试修改过的中间值。在每个测试运行之后，在`@After`区域中确保`x`被设为`null`**②**（以便`x`可以被垃圾收集）。然后用`assertEquals()`（JUnit众多静态`assertX`方法之一）测试`BigDecimal.add()`的返回结果是否符合期望**③**。为了处理预期的异常，在`@Test`上加上了可选的`expected`参数**④**。

进入TDD最佳状态的最好办法就是动手实践。把TDD原则牢牢印在你的脑海里，把JUnit框架搞明白，你就可以开始了！通过这些例子你也能看出来，单元测试级的TDD很容易掌握。

但所有TDD从业者最终都要测试使用依赖项或子系统的代码。下一节就会讲到那些代码的测试技术。

## 11.2 测试替身

如果你继续用TDD风格编码，很快就会遇到需要引用（经常是第三方的）依赖项或子系统的情况。在这种情况下，你肯定想把测试代码跟依赖项隔离开，以保证测试代码仅仅针对于实际构建的代码。你肯定还想让测试代码尽可能快速运行。而调用第三方依赖项或子系统（比如数据库）可能会花很长时间，也就是说会丧失TDD快速响应的优势（在单元测试层面尤其如此）。测试替身（test double）就是为解决这个问题而生的。

你在这一节将学会如何用测试替身有效隔离依赖项和子系统，看到使用四种测试替身（虚设、伪装、存根和模拟）的例子。

在最复杂的情况下，也就是测试有外部依赖项（比如分布式服务或网络服务）的代码时，依赖注入技术（见第3章）会和测试替身联手来拯救你，即便是看上去大得吓人的系统，它们也能保你安全无虞。

### 为什么不用Guice？

如果对第3章还记忆犹新，你应该不会忘了Guice——Java DI框架的参考实现。阅读这一节时你很可能边看边想：“他们怎么不用Guice呢？”

简言之，对于这些代码，即便引入像Guice这样简单的框架都显得过于复杂。记住，DI是一项技术。不要纯粹为了使用框架而使用它。

11

Gerard Meszaros在他的《xUnit Test Patterns<sup>①</sup>》（Addison-Wesley Professional, 2007）一书中给出了测试替身的简单解释，我们很荣幸能在这里引用他的说法：“测试替身（想一想特技演员）泛指任何出于测试目的替换真实对象的假冒对象。”

Meszaros接着定义了四种测试替身，如表11-3所示。

虽然看起来很抽象，但见到例子你就知道了，它们非常容易理解。让我们先从虚设对象开始讲起。

<sup>①</sup> 本书中文版《xUnit测试模式：测试码重构》已由清华大学出版社于2009年出版。——译者注

表11-3 四种测试替身

类 型	描 述
虚设替身	只传递不使用的对象。一般用于填充方法的参数列表
存根替身	总是返回相同预设响应的对象，其中可能也有些虚设状态
伪装替身	可以取代真实版本的可用版本（当然在品质和配置上达不到生产环境要求的标准）
模拟替身	可以表示一系列期望值的对象，并且可以提供预设响应

### 11.2.1 虚设对象

在这四种测试替身里，虚设对象用起来最容易。记住，它是用来填充参数列表，或者填补那些总也不会用的必填域。大多数情况下，你甚至可以传入一个空对象或null。

我们回到剧院门票那个例子中。能估算出一个售票亭带来的收入非常好，但剧院老板考虑得更长远。售出门票和预期收入的模型要做得更好，并且你还听到有人抱怨：随着需求增多，系统越来越复杂了。

你接到一项任务，要对售出票进行跟踪，并且某些票可以打9折。看起来你需要一个带有价格打折方法的Ticket类。你又从TDD循环的失败测试开始了，测试重点是新的getDiscountPrice()方法。你知道还需要两个构造方法：一个用于常规价格的门票，一个用于可能会打折的门票。Ticket对象最终需要两个参数：

- 客户姓名，测试中绝不会用到的String；
- 正常价格，测试中会用到的BigDecimal。

你非常确定getDiscountPrice()方法肯定不会引用客户姓名，也就是说可以给构造方法传入一个虚设对象（我们用的是固定字符串“Riley”），如代码清单11-8所示。

#### 代码清单11-8 用虚设对象实现的TicketTest

```
import org.junit.Test;
import java.math.BigDecimal;
import static org.junit.Assert.*;

public class TicketTest {
    @Test
    public void tenPercentDiscount() {
        String dummyName = "Riley";
        Ticket ticket = new Ticket(dummyName,
            new BigDecimal("10"));
        assertEquals(new BigDecimal("9.0"), ticket.getDiscountPrice());
    }
}
```

创建虚设对象

传入虚设对象

看到了吧，虚设对象的概念很平常。

为了让你彻底明白这个概念，我们在代码清单11-9中给出了部分实现的Ticket类。

### 代码清单11-9 用虚设对象测试Ticket类

```

import java.math.BigDecimal;

public class Ticket {
    public static final int BASIC_TICKET_PRICE = 30;           ← 默认价格
    private static final BigDecimal DISCOUNT_RATE =
        new BigDecimal("0.9");                                ← 默认折扣

    private final BigDecimal price;
    private final String clientName;

    public Ticket(String clientName) {
        this.clientName = clientName;
        price = new BigDecimal(BASIC_TICKET_PRICE);
    }

    public Ticket(String clientName, BigDecimal price) {
        this.clientName = clientName;
        this.price = price;
    }

    public BigDecimal getPrice() {
        return price;
    }

    public BigDecimal getDiscountPrice() {
        return price.multiply(DISCOUNT_RATE);
    }
}

```

有些开发人员会被虚设对象搞糊涂——他们预期的复杂度并不存在。虚设对象非常直接，它们就是过去为了避免出现`NullPointerException`的古老对象，只是为了让代码能跑起来。

我们转入下一个测试替身的讨论吧。存根对象（从复杂度来讲）向前迈出了一步。

#### 11.2.2 存根对象

在使用能够做出相同响应的对象代替真实实现的情况下，就会用到存根对象。让我们回到剧院门票价格的例子中，看一下实际应用。

写完Ticket类后，领导给你放了个假。你度完假刚回来，打开邮箱就看到一个bug单，报告说代码清单11-8中的`tenPercentDiscount()`测试时好时坏。你一检查代码库，发现`tenPercentDiscount()`已经被改掉了。现在新写了一个Price接口，而Ticket实例是由该接口的实现类HttpPrice创建的。

经过调查，你又发现一些变化，为了从一个外部网站上的第三方类HttpPricingService获得最初的价格，要调用HttpPrice的`getInitialPrice()`方法。

因此每次调用`getInitialPrice()`都会返回不同的价格。此外，它时好时坏还有几个原因，有时是公司防火墙规则变了，有时是第三方网站无法访问了。

所以测试就失败了，测试的目的也不幸受到了污染。记住，你所要的单元测试只是针对打9折的价格。

**注意** 涉及第三方价格网站调用的情景肯定超出了测试的责任范围。但你可以考虑做一个单独覆盖HttpPrice类和第三方的HttpPricingService的系统集成测试。

在用存根替换HttpPrice类之前，先看一下代码的当前状态，如下面三段代码（代码清单11-10至代码清单11-12）。除了跟Price接口有关的修改，剧院老板的想法也变了，觉得没必要记录是谁买了票，代码如下所示。

#### 代码清单11-10 实现了新需求的TicketTest

```
import org.junit.Test;
import java.math.BigDecimal;
import static org.junit.Assert.*;

public class TicketTest {
    @Test
    public void tenPercentDiscount() {
        Price price = new HttpPrice();
        Ticket ticket = new Ticket(price);
        assertEquals(new BigDecimal("9.0"),
                    ticket.getDiscountPrice());
    }
}
```

实现了Price的  
HttpPrice

创建Ticket

测试可能会失败

下面是新的Ticket，现在它包括了一个私有类FixedPrice，用来处理价格已知并固定的情况，即不需要从外部源中获取这些信息。

#### 代码清单11-11 实现了新需求的Ticket

```
import java.math.BigDecimal;

public class Ticket {
    public static final int BASIC_TICKET_PRICE = 30;
    private final Price priceSource;
    private BigDecimal faceValue = null;
    private final BigDecimal discountRate;

    private final class FixedPrice implements Price {
        public BigDecimal getInitialPrice() {
            return new BigDecimal(BASIC_TICKET_PRICE);
        }
    }

    public Ticket() {
        priceSource = new FixedPrice();
        discountRate = new BigDecimal("1.0");
    }

    public Ticket(Price price) {
        priceSource = price;
        discountRate = new BigDecimal("1.0");
    }
}
```

修改过的构造方法

```

public Ticket(Price price,
              BigDecimal specialDiscountRate) {           ← 修改过的构造方法
    priceSource = price;
    discountRate = specialDiscountRate;
}

public BigDecimal getDiscountPrice() {
    if (faceValue == null) {
        faceValue = priceSource.getInitialPrice();   ← 新的getInitialPrice
    }                                               方法调用
    return faceValue.multiply(discountRate);       ← 计算没变化
}
}

```

### 代码清单11-12 Price接口及其实现HttpPrice

```

import java.math.BigDecimal;

public interface Price {
    BigDecimal getInitialPrice();
}

public class HttpPrice implements Price {
    @Override
    public BigDecimal getInitialPrice() {           ← 返回结果随机
        return HttpPricingService.getInitialPrice(); ←
    }
}

```

那么，怎么才能做出跟HttpPricingService一样的响应？关键是想清楚测试的真实意图是什么？在这个例子中，你要测的是Ticket类中getDiscountPrice()方法所做的乘法跟预期一致。

因此你可以用总是返回同一价格的存根StubPrice换掉HttpPrice类，以调用getInitialPrice()。这样就可以把价格经常变化且时好时坏的HttpPrice类从测试中隔离出去了。使用代码清单11-13中的实现，测试就可以通过了。

### 代码清单11-13 使用存根对象的TicketTest实现

```

import org.junit.Test;
import java.math.BigDecimal;
import static org.junit.Assert.*;
public class TicketTest {

    @Test
    public void tenPercentDiscount() {           ← StubPrice存根
        Price price = new StubPrice();          ←
        Ticket ticket = new Ticket(price);       ← 创建Ticket
        assertEquals(9.0,                         ←
                     ticket.getDiscountPrice().doubleValue(),
                     0.0001);                          ← 检查价格
    }
}

```

StubPrice是个简单的小类，返回的初始价格总是10，如代码清单11-14所示。

**代码清单11-14 存根StubPrice**

```

import java.math.BigDecimal;

public class StubPrice implements Price {
    @Override
    public BigDecimal getInitialPrice() {
        return new BigDecimal("10");
    }
}

```

返回同一价格

咻！现在测试又能通过了，重要的是你又可以毫不畏惧地重构剩下的实现细节了。

存根是种挺实用的测试替身，但有时候我们会希望存根所做的工作可以尽可能地接近生产系统，这时可以用伪装替身。

### 11.2.3 伪装替身

伪装对象可以看做是存根的升级，它所做的工作几乎和生产代码一样，但为了满足测试需求会走些捷径。如果你想让代码的运行时环境非常接近生产环境（连接真实的第三方子系统或依赖项），伪装替身特别有用。

大部分Java开发人员迟早都要编写跟数据库交互的代码，特别是在Java对象上执行CRUD操作。在DAO（Data Access Object，数据访问对象）代码跟生产数据库连接之前，证明其可用的工作通常会留到系统集成测试阶段，或者根本就不做检查！如果能在单元测试或集成测试阶段对DAO代码进行检查，那将会有很多好处，最重要的是你能快速响应。

在这种情况下可以用伪装对象：用来代表跟你交互的数据库。但自己写一个代表数据库的伪装对象相当困难！好在经过数年的演进，内存数据库的轻巧易用已经足以胜任这一工作。HSQLDB（[www.hsqldb.org](http://www.hsqldb.org)）是广泛用于这一用途的内存数据库。

剧院门票应用进展良好，下一阶段的工作就是把门票保存在数据库中，以便后期获取。Java中最常用的数据库持久化框架是Hibernate（[www.hibernate.org](http://www.hibernate.org)）。

#### Hibernate与HSQLDB

如果你不了解Hibernate或HSQLDB，请不要惊慌！Hibernate是一个对象关系映射（ORM）框架，实现了Java持久化API（JPA）标准。简而言之，你可以调用简单的save、load、update，还有很多其他的Java方法来执行CRUD操作。这和用原始的SQL和JDBC不同，并且它经过抽象隔离了特定数据库的语法和语义。

HSQLDB只是个Java内存数据库。只要把hsqldb.jar放到你的CLASSPATH下就可以用了。尽管在关闭之后数据会全部丢失，但它的表现跟一般的RDBMS很像。（其实数据是可以保存下来的，请访问HSQLDB的网站了解更多细节。）

虽然我们可能又扔给你两项新技术，但随书源码中的构建脚本会帮你把正确的JAR依赖项和配置文件放到正确的地方。

首先，你需要一个Hibernate配置文件来定义到HSQLDB数据库的连接，如代码清单11-15所示。

### 代码清单11-15 用于HSQLDB的Hibernate配置文件

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.HSQLDialect
    </property>                                ← 设置方言
    <property name="hibernate.connection.driver_class">
      org.hsqldb.jdbcDriver
    </property>
    <property name="hibernate.connection.url">
      jdbc:hsqldb:mem:wgjd
    </property>                                ← 指定要连接的URL
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.connection.autocommit">true</property>
    <property name="hibernate.hbm2ddl.auto">
      create
    </property>                                ← 自动创建数据表
    <property name="hibernate.show_sql">true</property>
    <mapping resource="Ticket.hbm.xml"/>
  </session-factory>
</hibernate-configuration>

```

你应该注意到了，清单中的最后一行语句引用了Ticket类的映射资源(<mapping resource="Ticket.hbm.xml"/>)①。这个资源会告诉Hibernate怎么把Java文件映射到数据库列。在Hibernate配置文件里，除了方言(HSQLDB)，还有所有Hibernate需要用来在幕后自动构建SQL的信息。

尽管Hibernate允许你在Java类里直接用注解添加映射信息，但我们还是更喜欢下面这种XML映射方式，如代码清单11-16所示。

**警告** 注解跟XML映射之间的选择之战在邮件列表中已经打了很久了，所以你最好选个自己喜欢的，然后就由它去吧。

11

### 代码清单11-16 用于Ticket的Hibernate映射文件

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class
    name="com.java7developer.chapter11
    .listing_11_18.Ticket">
    ← 标出要映射的类
  </class>
</hibernate-mapping>

```

```

<id name="ticketId"
    type="long"
    column="ID" />

<property name="faceValue"
    type="java.math.BigDecimal"
    column="FACE_VALUE"
    not-null="false" />

<property name="discountRate"
    type="java.math.BigDecimal"
    column="DISCOUNT_RATE"
    not-null="true" />

</class>
</hibernate-mapping>

```

指定**ticketId**为关键字  
**faceValue**映射  
**discountRate**映射

弄完配置文件，该想想测什么了。用唯一ID获取Ticket是业务需要。为了满足这一业务（和Hibernate映射）要求，必须将Ticket类改成代码清单11-17这样。

### 代码清单11-17 带有ID的Ticket

```

import java.math.BigDecimal;
public class Ticket {
    public static final int BASIC_TICKET_PRICE = 30;
    private long ticketId; ←加上ID
    private final Price priceSource;
    private BigDecimal faceValue = null;
    private BigDecimal discountRate;
    private final class FixedPrice implements Price {
        public BigDecimal getInitialPrice() {
            return new BigDecimal(BASIC_TICKET_PRICE);
        }
    }
    public Ticket(long id) {
        ticketId = id;
        priceSource = new FixedPrice();
        discountRate = new BigDecimal("1.0");
    }
    public void setTicketId(long ticketId) {
        this.ticketId = ticketId;
    }
    public long getTicketId() {
        return ticketId;
    }
    public void setFaceValue(BigDecimal faceValue) {
        this.faceValue = faceValue;
    }
    public BigDecimal getFaceValue() {
        return faceValue;
    }
}

```

```

    }

    public void setDiscountRate(BigDecimal discountRate) {
        this.discountRate = discountRate;
    }

    public BigDecimal getDiscountRate() {
        return discountRate;
    }

    public BigDecimal getDiscountPrice() {
        if (faceValue == null) faceValue = priceSource.getInitialPrice();
        return faceValue.multiply(discountRate);
    }
}

```

现在Ticket的映射有了，Ticket类也改过了，可以调用TicketHibernateDao里的findTicketById方法进行测试了。哦，还要写JUnit测试设置的准备代码，如代码清单11-18所示：

#### 代码清单11-18 TicketHibernateDaoTest测试类

```

import java.math.BigDecimal;
import org.hibernate.cfg.Configuration;
import org.hibernate.SessionFactory;
import org.junit.*;
import static org.junit.Assert.*;

public class TicketHibernateDaoTest {

    private static SessionFactory factory;
    private static TicketHibernateDao ticketDao;
    private Ticket ticket;
    private Ticket ticket2;

    @BeforeClass
    public static void baseSetUp() {
        factory =
            new Configuration().
                configure().buildSessionFactory();
        ticketDao = new TicketHibernateDao(factory);
    }

    @Before
    public void setUpTest()
    {
        ticket = new Ticket(1);
        ticketDao.save(ticket);
        ticket2 = new Ticket(2);
        ticketDao.save(ticket2);
    }

    @Test
    public void findTicketByIdHappyPath() throws Exception {
        Ticket ticket = ticketDao.findTicketById(1);
        assertEquals(new BigDecimal("30.0"),
                    ticket.getDiscountPrice());
    }
}

```

① 使用Hibernate配置

② 设置测试Ticket的数据

③ 找到Ticket

```

    @After
    public static void tearDown() {
        ticketDao.delete(ticket);
        ticketDao.delete(ticket2);
    }

    @AfterClass
    public static void baseTearDown() {
        factory.close();
    }
}

```

在运行任何测试之前，先用Hibernate的配置创建所要测试的DAO①。然后，在每个测试运行之前，都在HSQLDB数据库里存两条门票的记录（作为测试数据）②。运行测试，测试DAO的findTicketById方法③。

因为你还没写TicketHibernateDao类及其方法，所以测试一开始会失败。使用Hibernate框架不需要SQL，也不需要提及用的是HSQLDB数据库。因此，DAO的实现应该和代码清单11-19类似。

### 代码清单11-19 TicketHibernateDao类

```

import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.criterion.Restrictions;

public class TicketHibernateDao {

    private static SessionFactory factory;
    private static Session session;

    public TicketHibernateDao(SessionFactory factory)
    {
        TicketHibernateDao.factory = factory;
        TicketHibernateDao.session = getSession();
    }

    public void save(Ticket ticket)
    {
        session.save(ticket);
        session.flush();
    }

    public Ticket findTicketById(long ticketId)
    {
        Criteria criteria =
            session.createCriteria(Ticket.class);
        criteria.add(Restrictions.eq("ticketId", ticketId));
        List<Ticket> tickets = criteria.list();
        return tickets.get(0);
    }
}

```

```

public void delete(Ticket ticket) {
    session.delete(ticket);
    session.flush();
}

private static synchronized Session getSession() {
    return factory.openSession();
}
}

```

DAO的save方法特别不起眼，就是调用Hibernate的save方法，然后用flush确保对象能存到HSQLDB数据库中❶。要取出Ticket，可以用Hibernate的Criteria（相当于SQL里的WHERE从句）❷。

写完DAO之后，测试就能通过了。你可能已经注意到了，save方法也已经被部分测试到了。你可以继续写更加完整的测试，比如检查一下从数据库中取回的票是否带有正确的discountRate。现在可以提前测试数据库访问代码了，所以数据库访问层也得到了TDD方式的所有好处。

我们接着讨论下一个测试替身：模拟对象。

#### 11.2.4 模拟对象

模拟对象跟前面提过的存根对象是亲戚，但存根对象一般都特别呆。比如在调用存根时它们通常总是返回相同的结果。所以不能模拟任何与状态相关的行为。

看个例子：假设你想用TDD方式写一个文本分析系统。其中一个单元测试要求文本分析类对某篇博文中出现的“Java 7”进行计数。但这篇博文是第三方资源，所以很多失败都跟你写的计数算法没太大关系。换句话说，测试代码不是孤立的，并且获取第三方资源可能很费时间。下面是一些很常见的失败：

- 由于防火墙限制，你的代码可能无法访问互联网上的这篇博文；
- 这篇博文可能被挪走了，而链接没有重定向；
- 博文可能被编辑过，“Java 7”出现的次数可能增加了，也可能减少了。

用存根几乎不可能把这个测试写出来，即便能写也极其繁琐，模拟对象此时登场。这是一种特殊的测试替身，你可以把它当做可以预编程的存根或超级存根。使用模拟对象非常简单：在准备要用的模拟对象时，告诉它预计会有些调用，以及每个调用该如何响应。模拟会跟DI结合得很好，你可以用它注入一个虚拟的对象，这个对象将完全按照已知方式行动。

让我们看一个剧院门票的例子。我们会用一个流行的模拟类库Mockito (<http://mockito.org/>)，请看代码清单11-20。

#### 代码清单11-20 用于剧院门票的模拟对象

```

import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

import java.math.BigDecimal;
import org.junit.Test;

```

```

public class TicketTest {
    @Test
    public void tenPercentDiscount() {
        Price price = mock(Price.class);
        when(price.getInitialPrice()).
            thenReturn(new BigDecimal("10"));

        Ticket ticket = new Ticket(price, new BigDecimal("0.9"));
        assertEquals(9.0, ticket.getDiscountPrice().doubleValue(), 0.000001);

        verify(price).getInitialPrice();
    }
}

```

创建模拟对象需要调用静态的`mock()`方法①，并将模拟目标类型的`class`对象作为参数传给它。然后要把模拟对象需要表现出来的行为记录下来，通过调用`when()`方法表明要记录哪些方法的行为，然后用`thenReturn()`指定所期望的结果是什么②。最后要证实在模拟对象上调用了预期的方法。这是为了确保你的正确结果不是经由不正确的路径得到的。

你可以像使用常规对象那样使用模拟对象，并且无需任何其他步骤就可以把它传给你调用的`Ticket`构造方法。这使得模拟对象成为了TDD的得力工具，有些从业者实际上更喜欢所有事情都用模拟对象来做，完全放弃了其他测试替身。

不管你是不是选择这种“最模拟”的TDD风格，完整的测试替身（需要的话加上一点DI）知识会让你毫不畏惧地进行重构和编码，即便面对复杂的依赖和第三方子系统也不怕。

Java开发人员会发现TDD的工作方式非常容易上手。但Java经常伴随着一个反复出现的问题——有些繁琐。在纯粹的Java项目中用TDD会导致大量的套路化代码。好在现在你已经学了一些其他的JVM语言，能用它们做出更精炼的TDD。实际上，从测试开始将非Java语言带入项目中是推动多语言项目的经典方式之一。

在下一节中，我们会讨论ScalaTest，这个测试框架具有广泛的测试用途。我们会从介绍ScalaTest开始，并会向你展示如何用它运行JUnit测试来测试Java类。

### 11.3 ScalaTest

如果你还记得，我们在7.4节说过TDD是动态语言的理想用例。实际上，Scala先进的类型推断让它在做测试上同样也有很多优势，尽管它是静态类型系统，还是经常会让人觉得它是动态语言。

Scala中的主测试框架是ScalaTest。它为做各种测试提供了一些极其实用的特质和类——从JUnit风格的单元测试到全面的集成和验收测试。我们来看一个ScalaTest的实战例子。

代码清单11-21用ScalaTest重写了11.4节中的代码，并且加了一个新的`sellTicket()`方法测试`fiftyDiscountTickets()`。

## 代码清单11-21 ScalaTest风格的JUnit测试

```

import java.math.BigDecimal
import java.lang.IllegalArgumentException
import org.scalatest.junit.JUnitSuite
import org.scalatest.junit.ShouldMatchersForJUnit
import org.junit.Test
import org.junit.Before
import org.junit.Assert._

class RevenueTest extends JUnitSuite with ShouldMatchersForJUnit {
    var venueRevenue: TicketRevenue = _

    @Before def initialize() {
        venueRevenue = new TicketRevenue()
    }

    @Test def zeroSalesEqualsZeroRevenue() {
        assertEquals(BigDecimal.ZERO, venueRevenue.estimateTotalRevenue(0))
    }

    @Test def failIfTooManyOrTooFewTicketsAreSold() {
        evaluating { venueRevenue.estimateTotalRevenue(-1) } ← 预期的异常
        → should produce [IllegalArgumentException]
        evaluating { venueRevenue.estimateTotalRevenue(101) }
        → should produce [IllegalArgumentException]
    }

    @Test def tenTicketsSoldIsThreeHundredInRevenue() {
        val expected = new BigDecimal("300");
        assert(expected == venueRevenue.estimateTotalRevenue(10));
    }

    @Test def fiftyDiscountTickets() {
        for (i <- 1 to 50)
            venueRevenue.sellTicket(new Ticket())
        for (i <- 1 to 50)
            venueRevenue.sellTicket(new Ticket(new StubPrice(),
            new BigDecimal(0.9)))
        assert(1950.0 ==
            venueRevenue.getRevenue().doubleValue()); ← Scala风格的断言
    }
}

```

我们还没讲过Scala如何处理注解。它们看起来跟Java注解一样。这没什么好说的。你的测试也是放在扩展了JUnitSuite的类中，这就是说ScalaTest会把这个类当做它能运行的东西。

你可以在命令行中用本地ScalaTest运行器轻松运行ScalaTest：

```

ariel:scalatest boxcat$ scala -cp /Users/boxcat/projects/tickets.jar:/Users/
    boxcat/projects/wgjd/code/lib/scalatest-1.6.1.jar:/Users/boxcat/
    projects/wgjd/code/lib/junit-4.8.2.jar org.scalatest.tools.Runner -o -s
    com.java7developer.chapter11.scalatest.RevenueTest

```

在这条命令中，所测试的Java类放在tickets.jar文件中，所以要把它跟ScalaTest和JUnit文件一起放在类路径中。

这条命令用-s选项指定了要运行的测试集(省略-s选项会运行所有测试集中的所有测试)。-o选项把测试输出发送到标准输出中(用-e把测试结果输出到标准错误流中)。ScalaTest参照这个配置输出报道途径(包括其他途径,比如图形化界面)。前面的例子产生的输出如下所示:

```
Run starting. Expected test count is: 4
RevenueTest:
- zeroSalesEqualsZeroRevenue
- failIfTooManyOrTooFewTicketsAreSold
- tenTicketsSoldIsThreeHundredInRevenue
- fiftyDiscountTickets
Run completed in 820 milliseconds.

Total number of tests run: 4
Suites: completed 1, aborted 0
Tests: succeeded 4, failed 0, ignored 0, pending 0
All tests passed.
```

这些测试已经被编译进了一个类文件中。只要类路径中有JUnit和ScalaTest两者的JAR,就可以用scala运行这些测试,而不用在JUnit运行器中。

```
ariel:scalatest boxcat$ scala -cp /Users/boxcat/projects/tickets.jar:/Users/
    boxcat/projects/wgjd/code/lib/scalatest-1.6.1.jar:/Users/boxcat/
    projects/wgjd/code/lib/junit-4.8.2.jar org.junit.runner.JUnitCore
    com.java7developer.chapter11.scalatest.RevenueTest
JUnit version 4.8.2
...
Time: 0.096

OK (4 tests)
```

当然,输出会稍有不同,因为执行测试用的是不同的工具(JUnit运行器)。

**注意** 在用Maven构建第12章的java7developer项目时,我们会用这个JUnit运行器。

### 用ScalaTest测试Scala代码

我们在这一节主要讨论用ScalaTest测试Java代码。但如果你用Scala作为项目中的主要编程语言会怎么样?

人们通常认为Scala是稳定层语言,所以如果你在使用Scala代码,应该也可以像测试Java代码那样测试Scala代码库。所以用ScalaTest代替JUnit是使用TDD方式的不二之选。

快速了解ScalaTest后,我们对TDD的讨论就结束了。第14章对行为驱动开发(BDD)的讨论就是建立在这些内容之上的,从逻辑关系上可以将BDD看成TDD的下一步。

## 11.4 小结

测试驱动开发能消除或减轻开发过程中的恐惧。遵从TDD风格,比如单元测试的红—绿—重

构循环，开发人员可以把自己从思维定式中解放出来，不会步入临时拼凑代码的窘境。

JUnit是Java开发人员的主要测试类库。它可以指定设置和拆卸挂钩，运行一个测试集里相互独立的测试。JUnit的断言机制会判断调用实现逻辑后是否能产生想要的结果。

不同类型的测试替身可以帮你写出恰当的测试。你可以用四种测试替身（虚设、存根、伪装和模拟）取代依赖项，从而让测试精准运行。在编写测试代码时，借助模拟对象可以实现终极的灵活性。

ScalaTest始终秉持大量减少套路化测试代码的观念，有助于开发人员深入理解测试的行为驱动开发风格。

我们在下一章讨论自动构建，以及建立在TDD基础之上的持续集成（CI）开发方法。使用CI开发方法，你能立即得到每个新变化的自动反馈，并且它鼓励开发团队成员之间彻底透明化。

## 构建和持续集成

12

### 本章内容

- 构建管道和持续集成（CI）的重要性
- Maven 3：惯例优先于配置的构建工具
- Jenkins：得到公认的CI工具
- 使用FindBugs和Checkstyle等静态代码分析工具
- Leiningen：Clojure构建工具

我们接下来要讲的故事取材于MegaCorp的真实事件，出于对当事人的保护隐去了真实姓名。故事的主角是：

- Riley，刚毕业的新人；
- Alice和Bob，两个“经验丰富”的开发老手；
- Hazel，紧张的项目经理。

时间是周五下午两点，新开发的Sally支付功能要在周末跑批前上线。

Riley：我能为上线做点什么吗？

Alice：当然，我想最后一版是Bob构建的。Bob？

Bob：是的，是我几周前用Eclipse生成的。

Riley：但现在我们都用IntelliJ了；那么，该怎么构建呢？

Bob：哦，这需要些经验！总之我们会搞定它，年轻人！

Riley：好。我没这方面的经验，但支付功能的构建应该没问题，对吧？

Alice：当然没问题。我在两周前刚创建的代码分支，其他人对代码的改动肯定还不多。

Bob：但是，实际上，你知道我们添了些泛型的修改，对不对？

[尴尬的沉默]

Hazel：改完你们要经常在一起试试。这个我们强调过很多次了！

Riley：要不要我订外卖？貌似今晚我们得加班了。

Hazel：你说对了，学得挺快嘛！

Alice：实际上，我已经将订餐电话设成快速拨号状态了，这是常态！

Hazel：赶紧把它搞定！我们已经因为延迟发布和bug太多损失很多了，高管正想找机会杀鸡儆猴呢。

Alice、Bob和Riley明显没有优秀的构建和持续集成（CI）经验，但“构建和CI”究竟是什么意思？

**构建和持续集成** 快速和重复地为各种环境产生高质量二进制部署工件的过程。

开发团队经常谈论“构建”或“构建过程”<sup>①</sup>。就本章而言，我们在提到构建时是指遵循构建周期用构建工具将源码转化成二进制工件的过程。像Maven这种构建工具有很长的、详细的构建周期，它们中的大多数对于开发人员来说是不可见的。一个相当基础的、典型的构建周期如图12-1所示。

清除 → 编译 → 测试 → 打包

图12-1 一个简化的典型构建周期

持续集成是指团队成员按照“尽早提交，经常提交”的口号频繁地集成工作成果。每个开发人员至少按天把代码提交到版本控制系统中，CI服务器会自动定期构建，以尽快检查集成错误<sup>②</sup>。CI服务器通常会在大屏幕上显示开心/悲伤的表情给团队以反馈。

那么构建和CI为什么重要？本章的每一节都会强调某些独特的好处，表12-1中列出了其中最为重要的几个。

表12-1 构建和CI的主要优势

主 题	解 释
重复性	任何人都可以随时随地运行构建。也就是说整个开发团队都可以自如地运行构建，而不需要一个专门的“构建负责人”做这件事。如果一个新加入的团队成员需要在周日的凌晨三点运行构建，他可以毫不犹豫地这么干
尽早反馈	一旦出了问题，你马上就能知道。在开发者处理需要集成的代码时这跟CI尤其相关
一致性	你知道部署的软件是什么版本，并且完全清楚每个版本的代码
依赖管理	大多数Java项目都有几个依赖项，比如log4j、Hibernate、Guice等。手工管理这些依赖项可能会非常困难，而且有一个版本发生变化就可能会导致软件不可用。良好的构建和CI能确保你总是针对同一个第三方依赖项进行编译和运行

为了将源码部署到运行时环境中，需要经过构建周期将其转变成二进制工件（JAR、WAR、RAR、EAR等）。比较老的Java项目通常都使用Ant，而比较新的则使用Maven或Gradle。很多开发团队还有夜间集成构建，有些已经升级成用CI服务器定期执行构建了。<sup>③</sup>

① 如果你的团队在谈论这些内容时或虔诚、或害怕，或话不多，那这一章就是为你准备的。

② 构建时间可配置：间隔可以是几分钟，也可以在提交时触发，或在其他特定时间运行。

③ 合作极其默契的项目团队能让非技术队友运行构建。

**警告** 如果你从IDE中构建JAR文件或其他工件，那是在自找麻烦。从IDE中构建得到的不是与本地IDE设置无关的可重用构建，那简直就是埋下了祸根。作为朋友，我再怎么强调这一点都不为过：不允许你用IDE构建工件！

但大多数开发人员都觉得构建和CI不值得他们投入精力，他们觉得这个工作做起来不够爽，也得不到什么回报。构建工具和CI服务器经常是在项目一开始的时候搭起来，但很快就被遗忘了。这么多年来，我们听到过很多类似的说法：“我们为什么还要在构建和CI服务器上花时间呢？现在弄得也挺好用的。够用就好，对不对？”

我们坚信良好的构建和CI能加快编码速度，提高代码质量。跟TDD（见第11章）相结合的构建和CI意味着你可以毫无后顾之忧地快速重构。你可以把它当做在你身后默默提供支持的导师，它为你营造一个安全的环境，让你可以快速编写并大胆修改代码。

本章，我们会首先介绍Maven 3。Maven 3是一个流行（还有争议，有些开发人员挺讨厌它）的构建工具，会强迫你按照严格定义好的构建周期工作。介绍Maven 3的内容中，除了常见的Java代码，还会涉及Groovy和Scala代码的构建。

Jenkins是CI界的流行天王，可以通过多种方式配置（以插件系统的方式）持续执行构建，并产生质量指标。在学习Jenkins时，我们还会深入了解FindBugs和Checkstyle产生的代码质量指标。

在学完Maven和Jenkins之后，你应该会彻底熟悉典型的Java构建和CI流程。之后我们会重点讨论Clojure的构建工具Leiningen，完全从另一个角度看构建和部署工具。你会看到它如何在提供工业级强度的构建和部署能力的同时实现极其迅速、易用的TDD风格。

与Maven 3的相遇将开启你的构建和CI之旅！

## 12.1 与 Maven 3 相遇

Maven是流行的Java及JVM语言相关的构建工具，然而反对它的人和支持它的人态度同样坚决。它的设计理念是，严格的构建周期辅以强大的依赖管理是成功构建的必要条件。Maven不仅是构建工具，更是项目技术组件的管理工具。实际上，Maven的构建脚本叫做POM（Project Object Model，项目对象模型）文件。这些POM文件是用XML写的，并且每个Maven项目或模块都有一个pom.xml文件。

**注意** POM文件中马上要加入对备选语言的支持，从而满足用户对灵活性的要求（就像Gradle提供的那些功能）。

### Ant和Gradle怎么样？

Ant是个流行的构建工具，特别是在早年的Java项目里。它作为公认的标准存在了相当长的一段时间。我们不准备在这里再讲了，因为之前已经有人讲过上百次了。更关键的是，我们

觉得Ant没有强制实行通用的构建周期，也没有一组通用（强制的）构建目标。这就是说开发人员必须研究手头每个Ant构建的细节。如果你要用Ant，Ant网站（<http://ant.apache.org>）列出了所有必需的细节。

Gradle是这一领域的新生。它有意选择了和Maven相反的路线，限制不会那么严格，你可以按自己的方式声明构建过程。它也跟Maven一样提供依赖管理和很多其他特性。如果你想尝试下Gradle，可以访问Gradle网站（[www.gradle.org](http://www.gradle.org)）了解更多细节。

要学习优秀的构建实践，Maven是适合的工具。它强制你遵循Maven构建周期，一旦掌握这个构建周期，你就可以轻松地构建世界上任何一个Maven项目。

Maven采取了惯例优先配置的策略，并希望你能融入到它的世界观，在源码该怎么布局、属性如何过滤等设置上都能接受它的安排。这可能会吓着某些开发人员，但Maven的构建周期是经过多年深思熟虑总结出来的，沿着它提供的路径走往往是最合理的。而对于那些极力反对墨守成规的人，Maven确实提供了覆盖默认值的办法，但那样会做出更加繁琐，并且标准化程度更低的构建脚本。

用Maven执行构建就是让它执行一个或几个目标（代表特定任务，比如编译源码、运行测试等）。目标都是绑到默认构建周期中的，如果你要求Maven执行测试（如mvn test），它会先编译源码和测试代码。简言之，它会强迫你遵守正确的构建周期。

如果你还没装Maven 3，请参见附录A中的A.2节。在完成下载和安装之后，再回到这里来创建你的第一个Maven项目。

## 12.2 Maven 3 入门项目

Maven遵循惯例优先的原则，你只要创建一个快速启动项目，马上就能看到它惯用的项目结构。它喜欢的典型项目结构看起来和下面的布局类似。

```
project
|--- pom.xml
`--- src
    |--- main
    |   |--- java
    |   |   |--- com
    |   |   |   |--- company
    |   |   |   |   |--- project
    |   |   |   |       |--- App.java
    |   |--- resources
    `--- test
        |--- java
        |   |--- com
        |   |   |--- company
        |   |   |   |--- project
        |   |   |       |--- AppTest.java
        |--- resources
`--- target
```

依照惯例，Maven把代码分成了main和test两部分。它还创建了一个特别的resources目录，构建工作所需的其他任何文件（比如用于日志的log4.xml文件、Hibernate配置文件以及其他类似资源文件）都放在这个目录下。pom.xml是Maven的构建脚本，关于这个文件的详情，请参见附录E。

如果你是多语言程序员，Scala和Groovy源码跟Java源码的结构一样，只是Java源码放在java目录下，而它们的根目录分别是scala和groovy。Java、Scala和Groovy代码可以高高兴兴地手拉手出现在同一个Maven项目中。

target目录是构建运行后才会创建的。所有的类、工件、报告和构建产生的其他文件都会出现在这个目录下。对于Maven项目结构的完整列表，请参见Maven网站上的Introduction to the Standard Directory Layout（标准目录布局介绍）页面（<http://t.cn/aKJYxo>）。

要为新项目创建这个结构，请执行下面的目标（注意其中的参数）：

```
mvn archetype:generate
-DgroupId=com.mycompany.app
-DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart
-DinteractiveMode=false
```

然后你会看到Maven开始刷屏，它在疯狂下载插件和第三方类库。Maven需要它们来运行这个目标，它的默认下载地址是Maven Central（工件的在线资源库）。

### 为什么Maven看起来像要把整个互联网都下载下来？

“哦，又来了，Maven又开始下载了。”这是构建Java项目的兄弟之间常说的模因<sup>①</sup>。但这真是Maven的错吗？我们认为它这样做有两个根本原因。一是第三方类库开发人员对包和依赖的管理很烂（比如在他们的pom.xml文件里指定一个实际上并不需要的依赖项）。另一个是继承自JAR为主的包系统自身的缺陷，没办法做更细化的依赖项控制。

除了“正在下载……”，控制台应该还会有下面这种声明：

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.703s
[INFO] Finished at: Fri Jun 24 13:51:58 BST 2011
[INFO] Final Memory: 6M/16M
[INFO] -----
```

如果这一步失败了，很可能是你的代理服务器不允许访问Maven Central，插件和第三方类库都放在那上面。要解决这个问题，只要编辑settings.xml文件（见附录A的A.2节），把下面这部分内容加上去，请根据你的实际情况为各元素填上恰当的值：

<sup>①</sup> 模因（Meme）也称为米姆、弥、弥因、弥母、迷因以及谜米等，是文化资讯传承单位。这个词是1976年理查德·道金斯在《自私的基因》一书中创造的，以生物学中的演化规则类比文化传承的过程。模因包含甚广，包括宗教、谣言、新闻、知识、观念、习惯、习俗，甚至口号、谚语、用语、用字、笑话。——译者注

```

<proxies>
  <proxy>
    <active>true</active>
    <protocol></protocol>
    <username></username>
    <password></password>
    <host></host>
    <port></port>
  </proxy>
</proxies>

```

重新运行上面的目标，这次应该能看到my-app项目出现在了目录中。

**提示** 如果团队中的所有人都遇到了这个问题，请在\$M2\_HOME/conf/settings.xml中加上代理配置。

Maven支持的原型（项目布局）几乎是无限的。如果要生成某个特定类型的项目（比如JEE6的项目），可以执行mvn archetype:generate目标，然后只要遵照它给你的提示就行了。

为了探索Maven的更多细节，我们来看一个源码和测试代码都已经准备好的项目，用它把整个构建周期走一遍。

## 12.3 用 Maven 3 构建 Java7developer 项目

还记得图12-1中的构建周期吗？Maven的构建周期跟那个类似，你马上就要经历构建周期中的每个阶段了。尽管本书中的源码不是一个应用程序，我们还是会把它们统一放到一个叫做java7developer项目中。

这一节的重点是：

- 探索Maven POM文件（即构建脚本）的基础；
- 如何编译、测试和打包代码（包括Scala和Groovy）；
- 如何用环境配置处理多个环境；
- 如何生成一个包含各种报告的项目网站。

首先你要搞明白定义java7developer项目的pom.xml文件。

### 12.3.1 POM

java7developer项目用pom.xml表示，包括各种插件、资源，以及构建所需的其他元素。可以在解压或签出本书项目代码的根目录（从现在开始我们用\$BOOK\_CODE指代这个位置）中找到这个pom.xml文件。POM主要由四部分组成：

- 项目基本信息；
- 构建配置；
- 依赖项管理；
- 环境配置。

这是个相当长的文件，但实际上它没有看起来那么复杂。如果你想了解POM中可以包含哪些内容的完整细节，请参见Maven网站上的POM Reference (<http://maven.apache.org/pom.html>)。

接下来我们就要解释java7developer项目pom.xml文件的这四部分，先从项目基本信息开始。

### 1. 项目基本信息

pom.xml文件中可以放入一系列的基本项目信息。代码清单12-1列出的是最起码的起步信息。

#### 代码清单12-1 项目基本信息

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.java7developer</groupId>
  <artifactId>java7developer</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>java7developer</name>
  <description>
    Project source code for the book!
  </description>
  <url>http://www.java7developer.com</url>

  <properties>
    <project.build.sourceEncoding>
      UTF-8
    </project.build.sourceEncoding>
  </properties>
  ...

```

这个工件在Maven资源库中的唯一标识符由三部分组成：第一部分是<groupId>的值com.java7developer①；第二部分是<artifactId>的值java7developer。<packaging>的值jar告诉Maven你要构建一个JAR文件（这里可能出现的值有war、ear、rar、sar和har）。唯一标识的最后一部分是<version>的值1.0.0<sup>②</sup>，表明版本号（执行Maven发布时会在这个值后面加上SNAPSHOT）。

文件中还指定了<projectName>和<url>，以及一些其他可选的项目信息③。<sourceEncoding>为UTF-8，这样可以确保在所有平台上的构建都是一致的③。

总的来说，这个配置会指导Maven构建出java7developer-1.0.0.jar工件，并把它存在Maven资源库中的com/java7developer/1.0.0目录下。

#### Maven版本和快照

作为Maven惯例优先原则的一部分，它倾向于以主要.次要.琐碎的格式来设置版本号，并依照惯例在版本号后面加上-SNAPSHOT表示这是一个临时性的工件。比如说，在你的团队为

① 版本号的格式遵循Major.Minor.Trivial风格，这是我们的最爱！

即将发布的1.0.0版本持续构建JAR时，Maven会依照惯例将版本号设置为1.0.0-SNAPSHOT。这样，各种Maven插件就知道这还不是生产版本，从而可以正确处理它。在把这个工件发布到生产环境中时，要发布为1.0.0，下一个修订bug的版本要从1.0.1-SNAPSHOT开始。

Maven通过它的发布插件把这些都自动化了。要了解更多细节，请参见发布插件页面(<http://maven.apache.org/plugins/maven-release-plugin/>)。现在你已经明白项目基本信息部分是什么样的了，接下来我们来看看<build>吧。

## 2. 构建配置

<build>中包含执行Maven构建周期目标所需的插件<sup>①</sup>及相应的配置。在大多数项目中，这部分内容都相当少，因为通常用默认插件的默认设置就够了。

在java7developer项目中，<build>中有几个覆盖了默认值的插件，以便可以：

- 构建Java 7代码；
- 构建Scala和Groovy代码；
- 运行Java、Scala和Groovy测试；
- 提供Checkstyle和FindBugs代码指标报告。

插件是以JAR为主的工件（主要是用Java写的）。要配置构建插件，需要把它放在pom.xml文件的<build><plugins>中。跟所有Maven工件一样，每个插件都有唯一标识，所以需要指定<groupId>、<artifactId>和<version>信息。对插件的所有配置都放在<configuration>中，并且每个插件的具体配置元素是不同的。比如编译插件的配置元素有<source>、<target>和<showWarnings>，这是编译器独有的配置信息。

代码清单12-2列出的是java7developer项目的构建配置部分（完整的代码清单及相应的解释在附录E中）。

### 代码清单12-2 POM：构建信息

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
        <showDeprecation>true</showDeprecation>
        <showWarnings>true</showWarnings>
        <fork>true</fork>
    
```

① 所用插件

② 编译Java 7代码

③ 编译器警告

12

<sup>①</sup> 如果你需要对构建进行配置，可以访问Maven的插件页面查看插件的完整列表（<http://maven.apache.org/plugins/index.html>）。

```

<executable>${jdk.javac.fullpath}</executable>
</configuration>
</plugin>

<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-surefire-plugin</artifactId>
<version>2.9</version>
<configuration>
<excludes>
<exclude>
com/java7developer/chapter11/
=> listing_11_2/TicketRevenueTest.java
</exclude>
<exclude>
com/java7developer/chapter11/
=> listing_11_7/TicketTest.java
</exclude>
...
</excludes>
</configuration>
</plugin>
</plugins>
</build>

```

因为Maven 3默认是编译Java 1.5的代码，而我们要编译Java 1.7②，所以需要指明编译器插件（的版本）①。

既然你打破了惯例，所以还要加上几个编译警告选项③。接下来要指定Java 7安装在哪儿④。只需要把sample\_<os>\_build.properties文件另存为build.properties，并编辑其中的jdk.javac.fullpath属性，因为Maven会用到它。

Surefire插件是测试用的。在配置中我们排除了几个失败测试⑤（有两个第11章的TDD测试）。现在构建部分已经讲完了，可以进入POM中非常重要的部分了：依赖管理。

### 3. 依赖管理

大多数Java项目的依赖项列表都很长，java7developer项目也不例外。Maven Central Repository中有各种各样的第三方类库，所以Maven可以帮你管理这些依赖项。最重要的是，这些第三方类库都有它们自己的pom.xml文件，会声明各自的依赖项，Maven可以据此找出任何需要下载的其他类库。

这些依赖项一开始主要分为两个作用域（compile和test）<sup>①</sup>。设置作用域跟把JAR文件放到CLASSPATH下是一样的效果。代码清单12-3是java7developer项目的<dependencies>部分。完整的代码清单及相应的解释在附录E中。

#### 代码清单12-3 POM：依赖项

```

<dependencies>
<dependency>
```

<sup>①</sup> J2EE/JEE项目通常也会用到runtime作用域的依赖项。

```

<groupId>com.google.inject</groupId>
<artifactId>guice</artifactId>
<version>3.0</version>
<scope>compile</scope>
</dependency>
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.8.5</version>
    <scope>test</scope>
</dependency>
...
</dependencies>

```

为了让Maven找到你引用的工件,需要给它正确的<groupId>、<artifactId>和<version>**①**。我们在之前提到过,把<scope>设置为compile**②**会把这些JAR加到CLASSPATH中用于代码的编译。将<scope>设置为test**③**会在Maven编译和运行测试时把这些JAR加到CLASSPATH中。

但你怎么知道该指定什么<groupId>、<artifactId>和<version>? 答案是搜索Maven Central Repository (<http://search.maven.org/>), 你几乎总能找到答案。

如果找不到合适的工件,可以用install:install-file目标自己手工下载和安装插件。这里有个安装asm-4.0\_RC1.jar类库的例子。

```

mvn install:install-file
-Dfile=asm-4.0_RC1.jar
-DgroupId=org.ow2.asm
-DartifactId=asm
-Dversion=4.0_RC1
-Dpackaging=jar

```

这个命令运行完后,你应该能在本地资源库的\$HOME/.m2/repository/org/ow2/asm/asm/4.0\_RC1/中找到安装好的工件,就像Maven下载的一样。

### 工件管理器

在你手工安装一个第三方类库时,你只是为自己装的,团队里的其他人呢? 在你做要跟同事共享的工件时也面临相同的问题,但你也不能把它放到Maven Central中(因为那是你们的私有代码)。

用二进制工件管理器可以解决这个问题，比如Nexus (<http://nexus.sonatype.org/>)。工件管理器就像你和团队自有的本地Maven Central，外界无法访问。大多数工件管理器还会缓存 Maven Central和其他资源库，你的开发团队所需的依赖项都可以从它那里得到。

环境配置是要搞懂的最后一部分POM了，它可以有效处理不同环境下的构建。

#### 4. 环境配置

环境配置是Maven用来处理环境化（比如UAT跟生产环境之间的构建差异）或其他与普通构建稍有不同的构建变体的。java7developer项目中有个例子，其中一个环境配置会关闭编译器和作废警告，如代码清单12-4所示。

**代码清单12-4 POM：环境配置**

```

<profiles>
  <profile>
    <id>ignore-compiler-warnings</id>           ① 该环境配置的ID
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.3.2</version>
          <configuration>
            <source>1.7</source>
            <target>1.7</target>
            <showDeprecation>false</showDeprecation>
            <showWarnings>false</showWarnings>
            <fork>true</fork>
            <executable>${jdk.javac.fullpath}</executable>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>

```

在执行Maven时用-P <id>可以指定要启用的环境配置（比如mvn compile -P ignore-compiler-warnings）①。在这个环境配置被激活后，会使用指定的编译器插件，作废警告和其他编译器警告都会被关闭②。

在Introduction to Build Profiles（构建环境配置介绍）页面可以找到更多关于环境配置和其他环境化目的如何使用它们的信息（<http://maven.apache.org/guides/introduction/introduction-to-profiles.html><sup>①</sup>）。

终于完成了java7developer项目的pom.xml文件之旅，你是不是已经迫不及待地想构建它了？

① 短链接：<http://t.cn/zl7MyO1>。——译者注

### 12.3.2 运行示例

希望你已经把代码下载下来了。你会在其中看到一些pom.xml文件，就是它们控制着Maven构建。

你会在这一节中经历最常用的Maven构建周期目标（clean、compile、test和install）。第一个目标就是清除上次构建遗留的所有工件。

#### 1. 清除

目标clean会删掉target目录。请换到\$BOOK\_CODE目录并执行clean目标。

```
cd $BOOK_CODE  
mvn clean
```

与你要用到的其他Maven构建目标不同，clean不会自动调用。如果想清除上次构建的工件，必需手动加上clean目标。

上次构建遗留的残余物已经清除了，接下来一般是执行编译源码的构建目标。

#### 2. 编译

目标compile用pom.xml文件中的编译器插件配置编译在src/main/java、src/main/scala和src/main/groovy下的源码。也就是说它会带着加到CLASSPATH中的编译作用域依赖项执行Java、Scala和Groovy编译器（javac、scalac和groovyc）。Maven还会处理在src/main/resources目录下的资源文件，确保它们作为编译CLASSPATH的一部分。

编译后的类最终会出现在target/classes目录下。请执行下面的Maven目标：

```
mvn compile
```

compile目标的执行应该相当快，并且在控制器中应该有类似下面这种输出。

```
...  
[INFO] [properties:read-project-properties {execution: default}]  
[INFO] [groovy:generateStubs {execution: default}]  
[INFO] Generated 22 Java stubs  
[INFO] [resources:resources {execution: default-resources}]  
[INFO] Using 'UTF-8' encoding to copy filtered resources.  
[INFO] Copying 2 resources  
[INFO] [compiler:compile {execution: default-compile}]  
[INFO] Compiling 119 source files to  
      C:\Projects\workspace3.6\code\trunk\target\classes  
[INFO] [scala:compile {execution: default}]  
[INFO] Checking for multiple versions of scala  
[INFO] includes = [**/*.scala,**/*.java,]  
[INFO] excludes = []  
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\java:-1: info: compiling  
[INFO] C:\Projects\workspace3.6\code\trunk\target\generated-sources\groovy-  
stubs\main:-1: info: compiling  
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\groovy:-1: info:  
      compiling  
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\scala:-1: info: compiling  
[INFO] Compiling 143 source files to  
      C:\Projects\workspace3.6\code\trunk\target\classes at 1312716331031  
[INFO] prepare-compile in 0 s
```

```
[INFO] compile in 12 s
[INFO] [groovy:compile {execution: default}]
[INFO] Compiled 26 Groovy classes
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 43 seconds
[INFO] Finished at: Sun Aug 07 12:25:44 BST 2011
[INFO] Final Memory: 33M/79M
[INFO] -----
```

在这一阶段，src/test/java、src/test/scala和src/test/groovy目录下的测试类还没编译。尽管专门有一个test-compile目标编译这些类，但最常见的方式是运行test目标。

### 3. 测试

在目标test中能见到Maven构建周期的实际效果。在你要求Maven运行测试目标时，它知道为了保证test目标成功运行，需要把前面的所有构建周期目标都执行一下（包括compile、test-compile和一系列其他目标）。

Maven会通过神火（Surefire）插件，使用pom.xml文件中的测试提供者（作为测试作用域的依赖项，此例中为JUnit）运行测试。Maven不仅会运行测试，还会产生报告文件，测试完成后你可以分析这些报告，以便对失败测试展开调研，并收集测试指标。

请执行下面的Maven命令：

```
mvn clean test
```

一旦完成测试类的编译和运行，应该就能见到下面这种输出。

```
...
Running com.java7developer.chapter11.listing_11_3.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_4.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_5.TicketTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec

Results :

Tests run: 20, Failures: 0, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 16 seconds
[INFO] Finished at: Wed Jul 06 13:50:07 BST 2011
[INFO] Final Memory: 24M/58M
[INFO] -----
```

测试结果存在target/surefire-reports目录下。现在你可以去看看那里的文本文件。稍后你能在[一个更棒的Web页面上](#)看到这些结果。

**提示** 你应该注意到了，命令中还有clean目标。我们这么做是出于习惯，以防有遗留的残余物欺骗我们。

现在你的代码编译过，也测试过了，并且已经准备好打包了。尽管你可以直接用package目标，但我们会用install目标。想知道为什么，且看下文分解！

#### 4. 安装

目标install的任务主要有两个。按pom.xml文件中<packaging>指定的方式（此例中为JAR文件）对编译结果打包。然后把打包好的工件安装到本地Maven资源库中（在\$HOME/.m2/repository下），以便其他项目可以把它当做依赖项用。跟其他目标一样，如果它发现之前的构建步骤还没执行，它也会先执行这些相关目标。请执行下面的Maven命令：

```
mvn install
```

一旦完成install目标，你应该见到下面这种输出报告。

```
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\Projects\workspace3.6\code\trunk\target\java7developer-1.0.0.jar
[INFO] [install:install {execution: default-install}]
[INFO] Installing C:\Projects\workspace3.6\code\trunk\target\java7developer-1.0.0.jar
to C:\Documents and Settings\Admin\.m2\repository\com\java7developer\java7developer-1.0.0\java7developer-1.0.0.jar
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 17 seconds
[INFO] Finished at: Wed Jul 06 13:53:04 BST 2011
[INFO] Final Memory: 28M/66M
[INFO] -----
```

在target目录（package目标的结果）和本地Maven资源库的\$HOME/.m2/repository/com.java7developer/1.0.0目录下应该能找到java7developer-1.0.0.jar工件。

**提示** 你可能希望把Scala和Groovy代码分别放到它们自己的JAR文件中。Maven支持这种操作，但你必须记住，对于Maven来说，每个独立的JAR工件都应该对应一个独立的项目。也就是说你必须用到Maven中多模块项目的概念。请参见Maven的Guide to Working with Multiple Modules（多模块处理指南）页面了解细节（[http://maven.apache.org/guides/mini/guide-multiple-modules.html<sup>①</sup>](http://maven.apache.org/guides/mini/guide-multiple-modules.html)）。

我们大多数人都是在团队中工作，并且经常要共享代码库，那我们怎么才能保证快速、可靠地构建大家共享的代码呢？这要靠CI服务器来保证，并且到目前为止，对于Java开发人员来说现在最流行的CI服务器是Jenkins。

<sup>①</sup> 短链接<http://t.cn/zjIIx70>。——译者注

## 12.4 Jenkins：满足CI需求

CI的成功要靠管理（开发纪律）和工具相结合。为了让CI过程达到优秀标准，Jenkins提供了很多必需的支持，如表12-2所示。

表12-2 衡量CI构建是否优秀标准及Jenkins如何达成这些标准

标 准	Jenkins如何达成
自动构建	Jenkins会在你需要的任何时间运行构建。它可以通过构建触发器实现自动构建
一直测试	Jenkins能运行任何你想要的目标，包括Maven的test。它有强大的测试失败趋势报告，只要有一个测试没通过，它就会报告构建失败
定期提交	这是开发人员的事
每次提交都构建	每次检测到版本控制库的新提交时Jenkins都可以执行构建
快速构建	这对基于单元测试的构建更加重要，因为你想要它们有更快的往返时间。Jenkins可以把工作发送给从属节点从而提高速度，但更主要的是开发人员做出精益的、有意义的构建脚本，并配置Jenkins在执行构建时调用恰当的构建周期目标
结果可视化	Jenkins有基于Web的仪表板，还有一套发送通知的办法

所有CI服务器都能轮询版本控制资源库，并执行构建周期目标compile和test。让Jenkins脱颖而出的是它易于使用的UI和可扩展的插件生态系统。

在配置Jenkins和它的插件时，UI的帮助非常大，它经常会在你输入完成后用Ajax检查输入的有效性。它还提供了大量的情景式帮助信息，运行Jenkins根本就不需要专业技能。

Jenkins的插件包罗万象，几乎可以轮询任何版本控制资源库，并且可以生成一系列非常有价值的代码报告。

### Jenkins和Hudson

在网上和某些书中，这个CI服务器的名字有些混乱。Jenkins实际上是Hudson项目最近出现的一个副本，主流的开发人员和活跃的社区现在都集中在Jenkins上。Hudson本身仍然是一个优秀的CI服务器，但相较而言Jenkins项目更活跃。

Jenkins是自由的开源软件，其社区充满活力，对新手帮助很大。

关于如何下载和安装Jenkins，请参阅附录D。完成下载和安装后，马上回来继续！

**警告** 假定你会把Jenkins的WAR文件装到Web服务器上，那么Jenkins安装的根URL是`http://localhost:8080/jenkins/`。如果是直接运行WAR文件<sup>①</sup>，根URL应该是`http://localhost:8080/`。

本节会讨论Jenkins安装的基础配置，然后是如何设置、执行构建任务。我们会以java7developer

<sup>①</sup> 指运行`java -jar Jenkins.war`。——译者注

项目为例，但你可以随意选用自己喜欢的项目。

为了让Jenkins监测源码资源库并执行构建，需要先配置基础设置。

### 12.4.1 基础配置

我们会从Jenkins的主页`http://localhost:8080/jenkins/`开始。要配置Jenkins，请点击左边菜单中的Manage Jenkins（管理Jenkins）链接（`http://localhost:8080/jenkins/manage`）。管理页中列出了很多设置选项。

现在，选择Configure System（系统配置）链接（`http://localhost:8080/jenkins/configure`）。你应该能进入类似于图12-2的界面中。

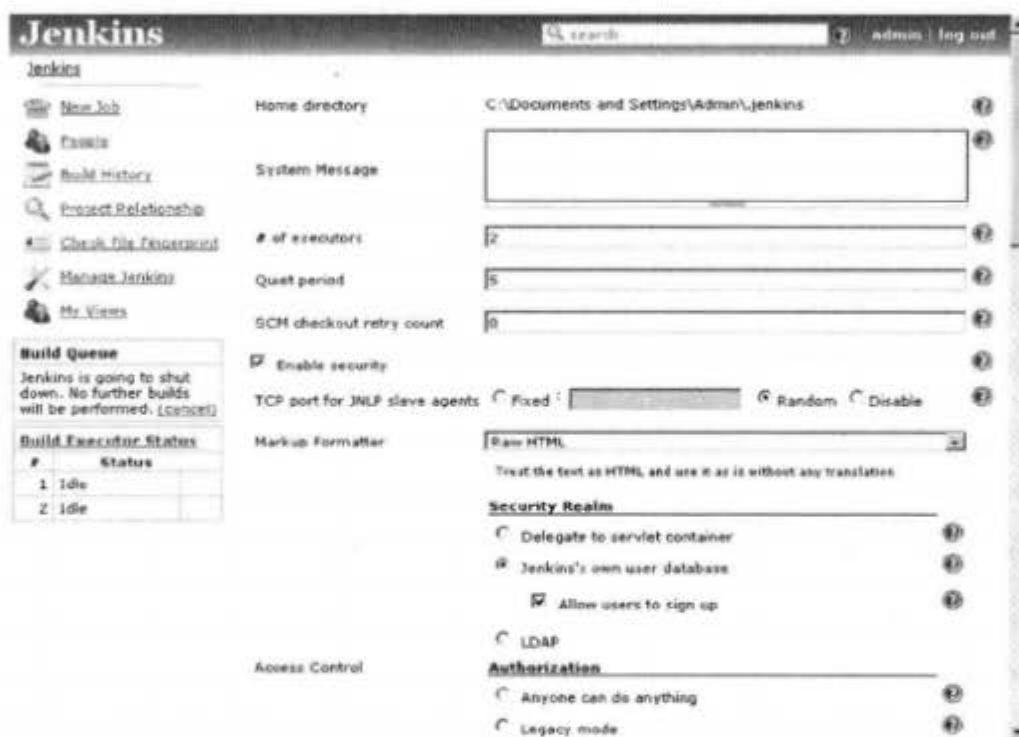


图12-2 Jenkins配置页

从界面的顶部可以看到Jenkins的home目录的位置。如果需要在UI之外进行配置，可以到这个目录中去。

**提示** 如果你是为团队安装Jenkins，并且需要考虑安全性，应该选中Enable Security（安全保护）和Prevent Cross Site Request Forgery Exploits（阻止跨域攻击请求）多选框，并进行相应的配置。对初学者来说，用Jenkins自身的数据库最容易。以后你可以随时切换到企业LDAP上，或基于Active Directory（活动目录）进行认证和授权。

为了执行构建，Jenkins需要知道构建工具放在哪里。这还要在配置页中设置，找到单词“Maven”。

#### 1. 构建工具配置

Jenkins内置了对Ant和Maven（可以用插件支持其他构建工具）的支持。在java7developer项

目中，我们用的是Maven（在Windows上），所以对Jenkins的配置如图12-3所示。

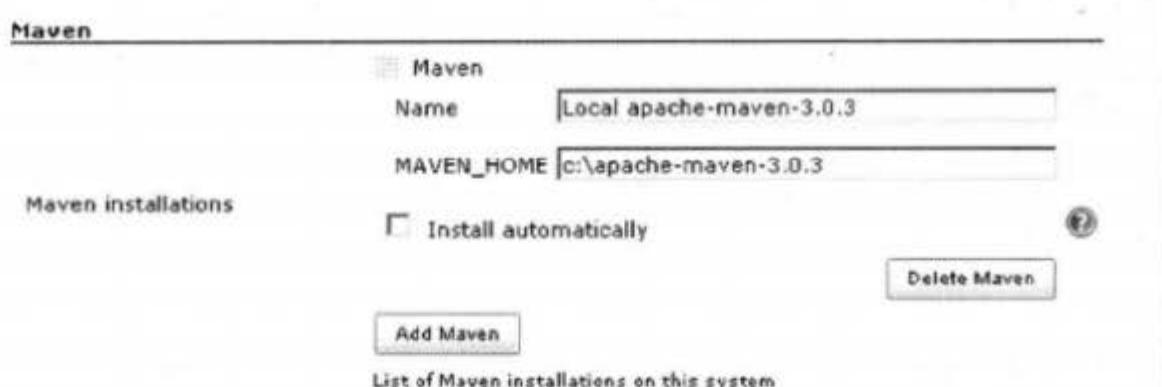


图12-3 设置构建工具Maven

注意，Jenkins有一个自动安装Maven的选项，在没装Maven的机器上，这个选项还是挺方便的。

现在Maven配置好了，需要告诉Jenkins你用什么版本控制资源库。这在配置页的下面。找到单词SVN。

## 2. 版本控制配置

Jenkins内置了对CVS和Subversion（SVN）的支持。像Git和Mercurial这样的版本控制系统也有插件。java7developer项目用SVN 1.6，配置如图12-4所示。

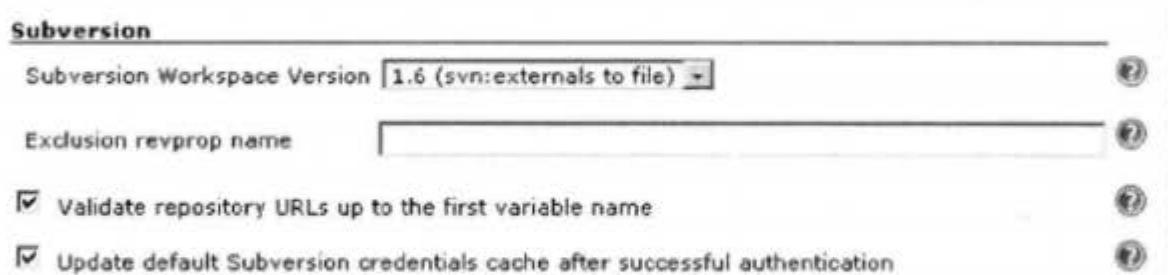


图12-4 SVN版本控制配置

在设置好这些配置后，点击屏幕底部的Save按钮，以确保这些配置会被保存下来。

现在Jenkins的基础设置已经做好了，可以创建你的第一个任务了。

### 12.4.2 设置任务

要设置新任务，请回到仪表板中并点击左手菜单的New Job（新任务）链接，进入任务设置页面（<http://localhost:8080/jenkins/view/All/newJob>）。这里有很多选项可供选择。

要设置一个任务来构建java7developer项目，先要给任务确定一个标题（java7developer），选择Build a Maven 2/3 Project（构建一个Maven 2/3项目）选项并点击OK按钮继续。你应该会进入一个类似图12-5的配置界面。这里有一些输入项要填，但下面这些应该是你先填好的内容：

- 源码管理；
- 构建触发器；
- 构建。

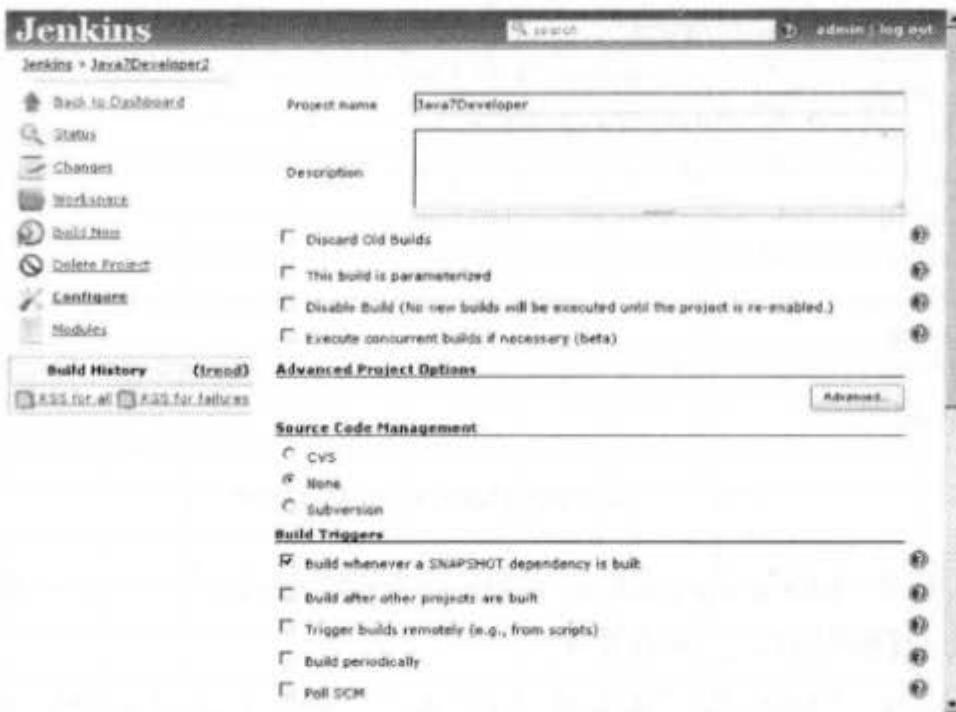


图12-5 Maven 2/3任务配置页面

我们从源码管理的配置开始。

## 1. 源码管理

源码管理主要设置要构建的源码来自版本控制的那个分支、标记或标签。随着你的团队向版本控制系统中稳步添加源码，它就是持续集成中的“集成”。对于java7developer项目，我们用SVN构建主干中的源码。设置如图12-6所示。

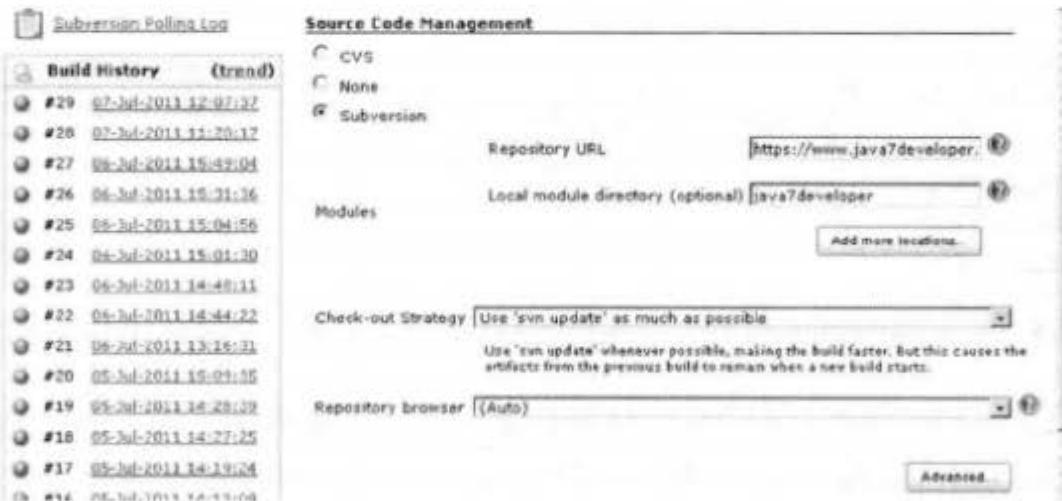


图12-6 java7developer源码管理配置

一旦告诉Jenkins从哪里获取源码，接下来要配置的就是Jenkins应该隔多长时间构建一次，这是通过构建触发器完成的。

## 2. 构建触发器

构建触发器把“持续”引入了持续集成。你可以要求Jenkins在源码控制库每次有新提交时就进行构建，或者采用更悠闲的方式，设为每日构建一次。

我们对java7developer项目的设置，只是要求Jenkins每隔15分钟轮询SVN一次，如图12-7所示。

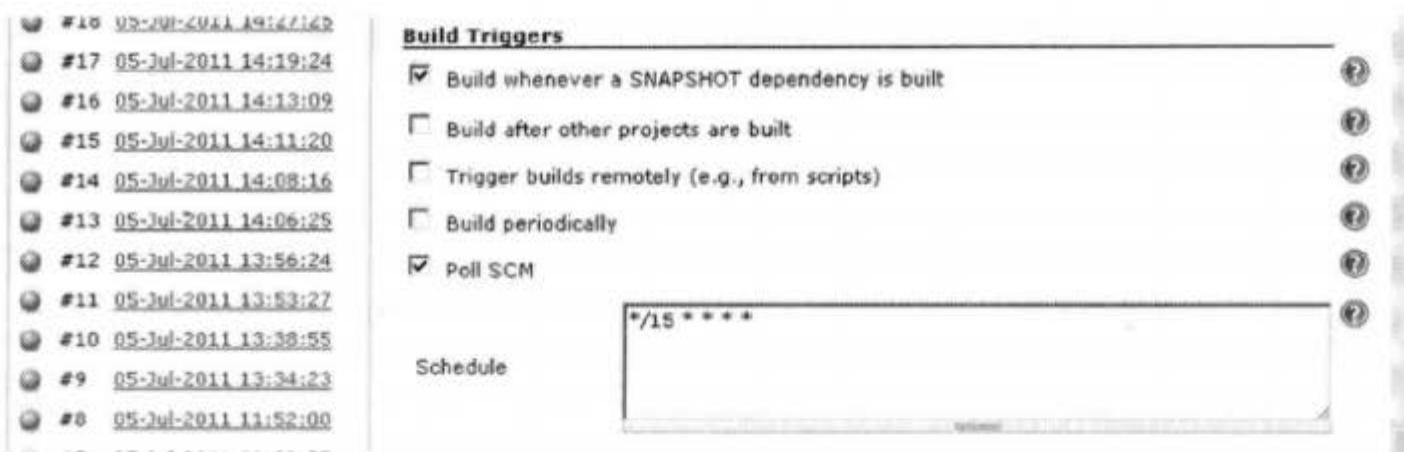


图12-7 java7developer构建触发器配置

你可以点击输入项旁边的帮助图标(表示为?)查看帮助信息。在这个例子中,编写类似cron的表达式来指定轮询周期时你可能需要帮助。

到这个阶段,Jenkins已经知道了到哪里去找源码,隔多长时间构建一次。接下来就该告诉Jenkins应该执行哪个构建阶段(构建脚本中的目标或目的)。

### 3. 构建

用Jenkins可以设置很多任务来执行构建周期的不同阶段。你可能想要一个每晚执行一次完整的系统集成测试的任务。但更多情况下,你可能想要执行频率更高的任务,在每次有新的源码提交到版本控制系统时编译源码并运行单元测试。

对于java7developer项目,我们要求Jenkins执行Maven的clean和install目标,如图12-8所示。

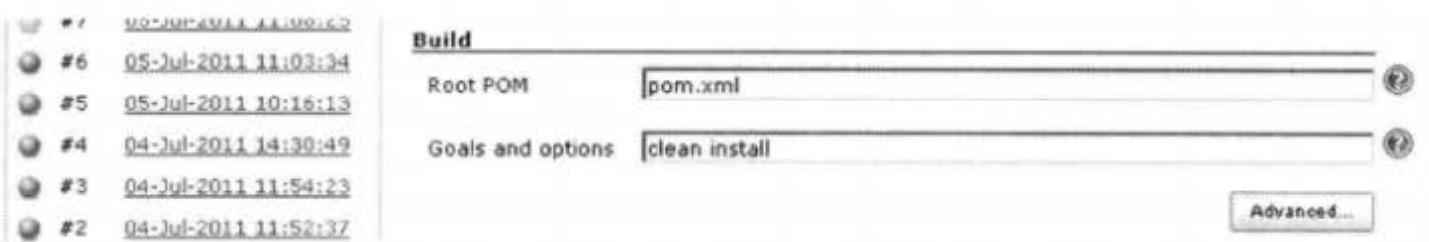


图12-8 Java7developer任务中要执行的Maven构建目标(clean、install)

Jenkins现在有java7developer项目的所有信息了,它可以每隔15分钟轮询一次SVN代码库的主干,执行Maven的clean和install目标。不要忘了点击Save按钮把任务存下来!

现在可以回到仪表板,在那里看看你的任务,它应该非常像图12-9。

在Last Success(S)一栏(最近一次成功),圆形图标表示任务最后一次构建的状态。在Weather(W)(天气)一栏,天气图标表示项目的总体健康状况,它是由构建失败的频率、测试是否通过,还有一系列其他可能情况(取决于你配置的插件)决定的。要进一步了解这些图标的含义,请点击仪表板中的Legend(图例)链接(<http://localhost:8080/jenkins/legend>)。

现在任务已经准备好了,你可能想看看它运行起来是什么样!你可以等15分钟后的第一次轮询,也可以强制执行一次构建。

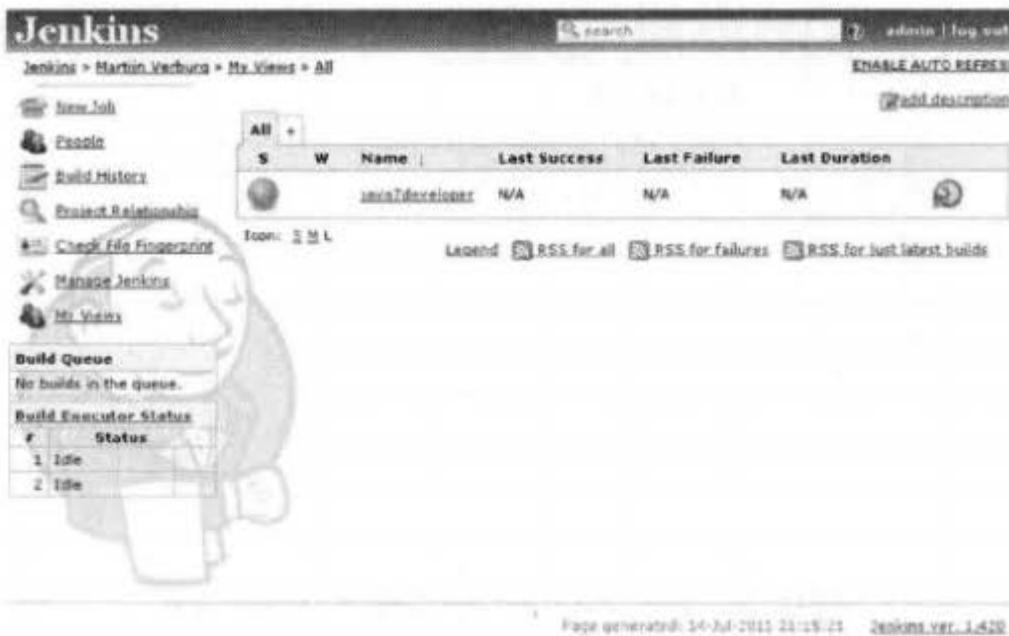


图12-9 有java7developer任务的仪表板

### 12.4.3 执行任务

要检查新配置，强迫任务执行是个好办法。对于java7developer任务，只要到仪表板中，点击Schedule a Build（调度构建）按钮（在Last Duration旁边带绿色箭头的钟表图标）。然后你就能刷新页面去看看构建的执行结果了。

**提示** 点击仪表板右上角的Enable Auto Refresh（允许自动刷新），可以让仪表板自动刷新。这样你就可以及时看到当前构建的状态了。

在构建执行时，java7developer任务的第一个图标会闪，表明构建正在处理中。在页面左侧还能看到Build Executor Status（构建执行器状态）。构建一完成，Last Success (S)（最近一次成功）一栏那个圆形图标就变成了红色，表明构建失败了。

这个失败是因为漏掉了build.properties文件。如果你阅读12.2节时没按照书中的指示做，现在也可以很快解决掉这个问题，找到build.properties文件样本，编辑它，以便能找到要用到的Java 7 JDK。下面是在Unix上执行这些操作的例子：

```
cd $USER/.jenkins/jobs/java7developer/workspace/java7developer
cp sample_build_unix.properties build.properties
```

现在你可以回到仪表板中，再次手工运行构建。这次构建应该能成功，并且仪表板中java7developer任务的Last Success（最近一次成功）一栏应该是个蓝色图标，表示构建成功了。

你还可以马上去看一下构建的测试报告，因为Jenkins知道如何读取Maven产生的输出。要看测试结果，可以点击java7developer任务Last Success一栏的链接（<http://localhost:8080/jenkins/job/java7developer/lastSuccessfulBuild/>）。该链接会将你带入Latest Test Result（最新测试结果）页面，其界面如图12-10所示。

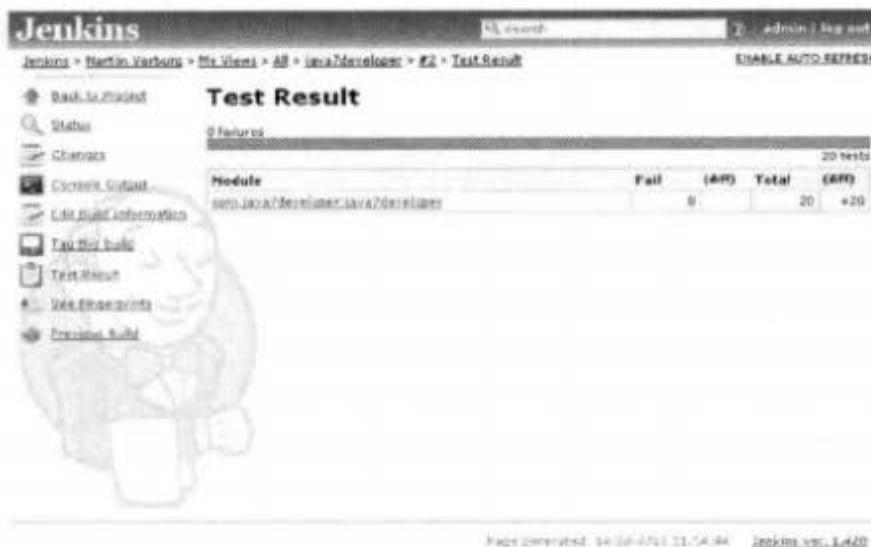


图12-10 java7developer成功构建的测试结果

测试全部通过，非常棒！如果有任何一个失败，你可以深入了解每个测试的细节。

我们对运行失败和成功构建的基础都进行了总结。对于java7developer项目来说，Jenkins会继续轮询SVN并在发现新提交时执行新构建。

你已经见过Jenkins如何运行构建，构建因不同原因失败时如何在界面上发出警告，如何检查测试成功或失败。但Jenkins可以做得不止这些，它还能给出一系列实用的代码指标，让你对代码的质量有更深入的认识。

## 12.5 Maven 和 Jenkins 代码指标

Java和JVM已经存在很长时间了，并且这么些年积累下来，已经开发出了一些强大的工具和类库来指导开发人员编写优质的代码。我们宽泛地把这个领域定义为代码指标或静态代码分析。Maven和Jenkins都支持目前最流行的工具。尽管这些流行的工具（或新的专门化工具）对其他语言的支持越来越多，但它们仍然主要是针对Java语言自身。

**提示** 现代IDE（比如Eclipse、IntelliJ和NetBeans）也支持几种静态语言分析工具和类库，值得花些时间研究一下。

代码指标工具主要是为了消除所有开发人员都会犯的小错误。它能帮你树起最低的代码质量标杆，告诉你下面这些情况：

- 测试对代码的覆盖率<sup>①</sup>；
- 代码的格式是否清晰（有助于差异比较和可读性）；
- 是否很可能会出现NPE；
- 是否忘记了域对象中的equals()和hashCode()方法。

<sup>①</sup> 本书不讨论普通的代码覆盖工具，因为它们和Java 7还不兼容。

各种工具提供的检查列表都很长，但开发团队要自己决定对项目做哪些检查。

### 代码指标的局限性

一些团队因为解决了代码指标工具警告过的问题就认为他们的代码库臻于完善了，这是不对的。代码指标警告能帮你去掉很多低级bug，避免糟糕的编码实践。但它们不能保证代码质量，或者判断业务逻辑实现是否正确。

另外一个问题是管理层可能想把这些指标放到报告里。为了管理层和你们自己考虑，请把代码指标留在开发人员这一层面。它们不是项目管理指标。

Maven和Jenkins结合得很好，既可以提供代码指标的概览，也能告诉你其中的细节。本节的两个主要内容如下：

- 如何安装并配置Jenkins插件；
- 如何配置代码一致性（Checkstyle）和bug查找（FindBugs）插件。

我们仍以java7developer为例。先来看看如何安装Jenkins插件，这是得到代码指标报告功能的前提条件。

#### 12.5.1 安装 Jenkins 插件

Jenkins基于UI的插件管理器很不错，它可以帮你下载和安装插件，所以安装Jenkins插件并不复杂。安装Jenkins插件时需要重启Jenkins，所以应该先进入Jenkins管理页（<http://localhost:8080/jenkins/manage>），并点击Prepare to Shutdown（准备关闭）链接。这会终止所有即将执行的任务，以便你可以安全地安装插件，重启Jenkins。

Jenkins准备好关闭之后，就可以访问插件管理器了。在管理页中点击Manage Plugins（管理插件）链接（<http://localhost:8080/jenkins/pluginManager/>）。应该能见到如图12-11所示的界面。



图12-11 Jenkins插件管理器

第一个是Updates(更新)标签。切换到Available(可用)标签，能见到一长串可用插件的列表。为学习本章内容，需要选中下面这些插件：

- Checkstyle；
- FindBugs。

然后点击界面底部的Install按钮开始安装。安装完成后，可用通过链接http://localhost:8080/jenkins/restart重启Jenkins。

重启Jenkins之后插件就安装好了。现在该去配置这些插件了，先从Checkstyle插件开始。

### 12.5.2 用Checkstyle保持代码一致性

Checkstyle是静态代码分析工具，主要关注代码布局、Javadoc层次是否恰当，以及其他语法糖的检查。它还会检查常见的代码错误，但FindBugs检查得更加全面。

Checkstyle的重要性体现在两个方面。首先，它有助于强化小组的编码风格规范，以便团队成员可以很容易地读懂彼此的代码（易读性是Java得以流行的一个主要原因）。其次，如果代码元素的位置和空格保持一致，diffs和patches用起来就更容易了。

我们在Maven的pom.xml文件中已经配置过Checkstyle插件了，所以你只需要在java7developer任务中加上checkstyle:checkstyle目标。要配置该任务，点击在仪表板中列出的java7developer链接，然后在新界面上点击左侧菜单中的Configure(配置)链接。

接着配置报告，并确定违规的情况出现次数太多时是否应该放弃构建。图12-12中是我们对java7developer项目中Maven构建及报告的配置。



图12-12 Checkstyle配置

别忘了点击Save把这个配置存下来！Checkstyle的默认规则是Sun公司最初提出来的Java编码规范。Checkstyle能微调到第n级，所以应该能准确表示团队的编码规范。

**警告** 最新版的Checkstyle还没全面支持Java 7语法，所以你见到对try-with-resources、钻石语法和其他Coin项目语法的肯定可能是假的。

让我们来看看默认规则集如何把Java7developer项目组织起来。跟往常一样，你可以回到Jenkins的仪表板中，手工执行构建。构建完成后，回到最近构建成功页面（记住，可以通过Last Success一栏的链接访问该页面），点击左侧菜单上的Checkstyle Warnings（Checkstyle警告）链接进入Checkstyle报告页。java7developer项目的Checkstyle报告页看起来应该如图12-13所示。

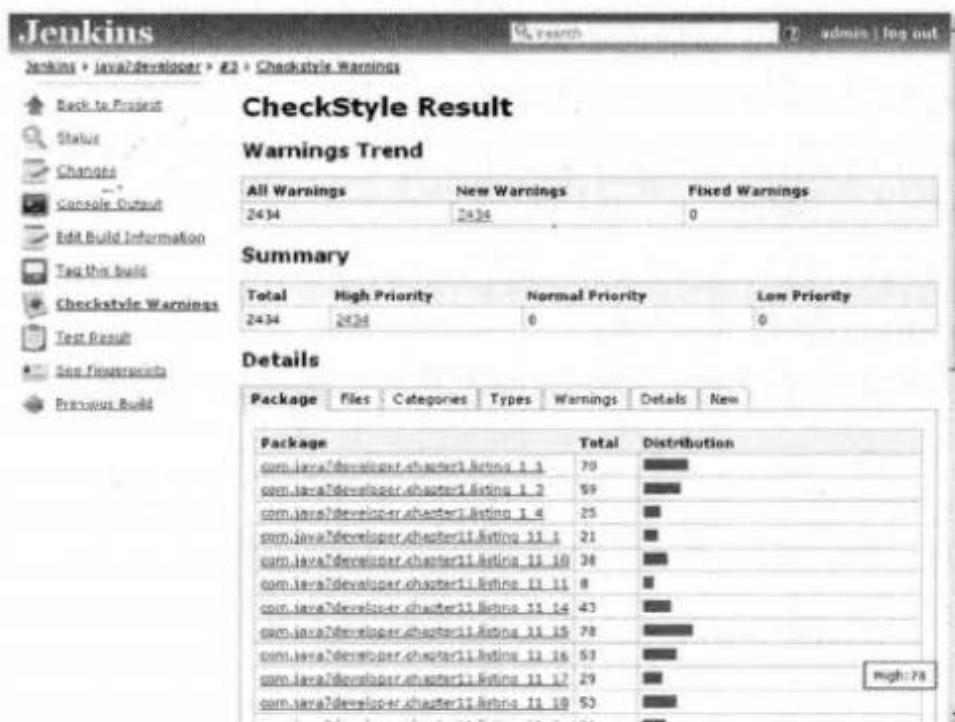


图12-13 Checkstyle报告

如你所见，在Java7developer的代码上有些警告信息。看起来我们还有些工作要做！你可以深入到每个警告中，看看为什么会发生违规，并在下一次构建之前改正它。

Checkstyle肯定在这方面有所帮助，但它的重点不是潜在的代码错误。这种重要的代码错误检查最好用FindBugs插件。

### 12.5.3 用FindBugs设定质量标杆

FindBugs（Bill Pugh出品）是为了找出代码中潜在的bug而做的字节码分析工具。由于它分析的是字节码，所以也能用在Scala和Groovy上。但因为它所设置的规则是为了捕获Java语言中的bug，所以如果你用它来分析Groovy和Scala代码，要对其持审慎的态度，因为它可能发现不了其中的bug。

FindBugs背后有大量研究成果的支持，都是由特别熟悉Java语言的开发人员做的。它能检测出下面这些状况：

- 会导致NPE的代码；
- 赋值给一个从来没用到的变量；
- 用==而不是用equals方法比较字符串对象；
- 在循环中用基本的+操作符而不是用StringBuffer合并字符串。

你应该先试试FindBugs的默认设置，然后再根据要检测的规则进行微调。

**警告** 即便是Java语言，FindBugs也会误报。你应该认真研究它发出的警告，如果确信可以忽略它们，可以显式排除这些情况。

FindBugs的重要性体现在两个方面。首先，它以结对程序员的角色教开发人员养成好习惯(尽可能帮助检测出潜在bug)。其次，项目的总体质量变好了，并且问题跟踪单里不会再充满烦人的小bug，让团队可以解决实际问题，比如业务逻辑的变化。

跟Checkstyle插件一样，你可以点击仪表盘任务列表中的java7developer链接，然后在新界面中点击左侧菜单上的Configure链接。

为了执行FindBugs插件，还要在Jenkins中添加Maven构建目标compile findbugs:findbugs(需要compile以便FindBugs能处理字节码)。

除了定义违例过多构建是否失败，还可以配置报告。如图12-14所示。



图12-14 FindBugs配置

别忘了点击Save把这个配置存下来！FindBugs预定义的规则集可以大范围调整，以准确表示团队的编码规范。让我们来看看默认规则集如何应用到Java7developer项目上。

跟往常一样，你可以回到Jenkins的仪表盘中，手工执行构建。构建完成后，回到最近构建成功页面(记住，可以通过Last Success栏的链接访问该页面)，点击左侧菜单上的FindBugs Warnings链接进入FindBugs报告页。java7developer项目的FindBugs报告页看起来应该如图12-15所示。

如你所见，Java7developer的代码有些警告信息。写书的作者也可能写出不完美的代码！你可以仔细检查每个警告，看看为什么会发生违规，并且如果你愿意，可以在下一次构建之前改正它。

FindBugs会把大部分常见的Java陷阱和编码错误都找出来。随着开发团队对这些错误的了解程度不断加深，报告中的警告数量会逐步减少。你们不仅提升了代码的品质，还完善了自身的编码能力！

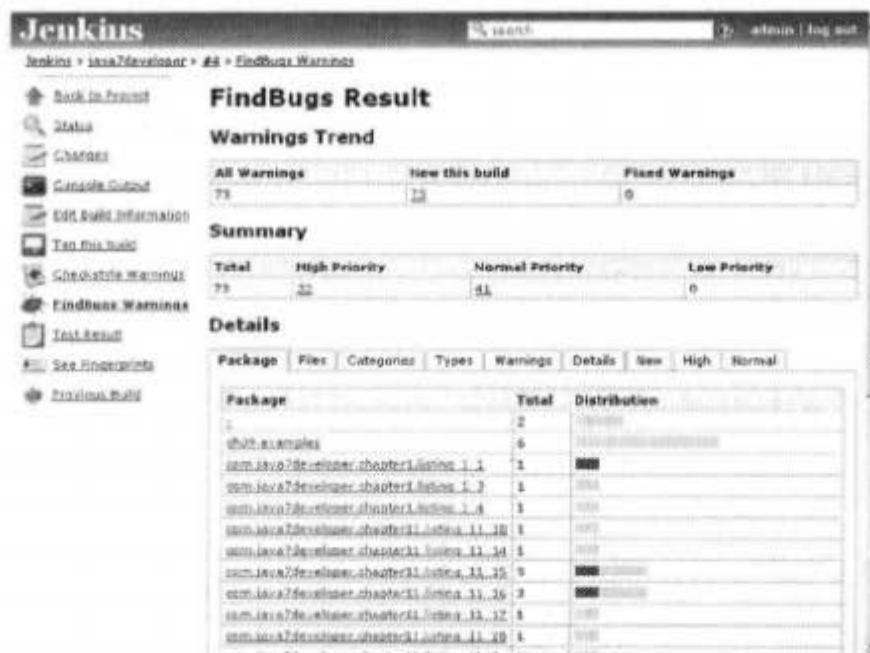


图12-15 FindBugs报告页

Jenkins、Maven和代码指标这一节到此就完成了。这一领域的工具化程度相当高（Scala和Groovy还需要更多支持），启动和运行也非常容易。如果你是CI迷，想探索Jenkins的完整能力，我们强烈推荐John Ferguson Smart不断更新的*Jenkins: The Definitive Guide* (O'Reilly)。你可能已经注意到了，与JVM多语言编程相关的构建和CI的拼图还缺了一片——我们还没处理过Clojure项目。好在Clojure社区已经专门针对纯粹的Clojure项目制作了几个构建工具，并已得到了广泛应用。其中最流行的就是Leiningen，一个专为Clojure写的构建工具。

## 12.6 Leiningen

要成为对开发人员有用的构建工具，关键是要具备下面这几种能力：

- 依赖管理；
- 编译；
- 测试自动化；
- 部署打包。

Leiningen对此采取的策略是分而治之。它重用已有的Java技术实现了每种功能，但却没有把所有功能都放在一个包里。

这听上去可能比较复杂，还有点恐怖，但开发人员并不会受到这种复杂性的影响。实际上，甚至没用过底层Java工具的开发人员也能使用Leiningen。我们一开始先通过一个非常简单的过程来安装Leiningen。然后讨论Leiningen的组件和整体架构，最后用Hello World项目来小试牛刀。

你将看到如何开始一个新项目，添加依赖项，使用Leiningen提供的Clojure REPL内部依赖项。这自然会让我们转而讨论如何用Leiningen在Clojure内做TDD。作为本章的收尾，我们会看一下如何打包代码，产生一个应用程序部署或供人调用的类库。

我们来看看如何开始使用Leiningen吧。

### 12.6.1 Leiningen 入门

Leiningen非常容易上手。对于类Unix系统(包括Linux和Mac OS X),开发人员可以从掌握lein脚本开始。在GitHub上可以找到它Leiningen(在<https://github.com/>页面中或用自己喜欢的搜索引擎搜索Leiningen)。

把lein脚本放到PATH中并设为可执行文件后,它就可以运行了。在第一次运行lein时,它会检查需要安装哪些依赖项(还有哪些已经装上了)。只要需要,它甚至会把其他不属于Leiningen核心部分的组件也给装上。因为要安装依赖项,首次运行可能比后续运行稍慢一些。

在下一节,我们会介绍Leiningen的架构,以及为它提供核心功能的Java技术。

#### 在Windows上安装Leiningen

从一个Unix老黑客的角度来看,Windows的烦人之处是它没有为钟爱命令行的人提供赖以生存的、标准的、简单的工具。比如说,基本的Windows安装中没有通过HTTP下载文件的curl或wget工具(Leiningen需要用它们从Maven资源库中下载jar)。解决办法是用Leiningen Windows安装——带有lein.bat文件和预置的wget.exe压缩文件,为了让自行安装的lein正确工作,需要把它们放到Windows的PATH中的目录下。

### 12.6.2 Leiningen 的架构

我们说过,Leiningen封装了一些主流的Java技术并做了简化。它封装的主要组件是Maven(版本2)、Ant和javac。

如图12-16所示,Maven用来做依赖项解析和管理,javac和Ant用来构建、运行测试和完成构建过程中的其他工作。

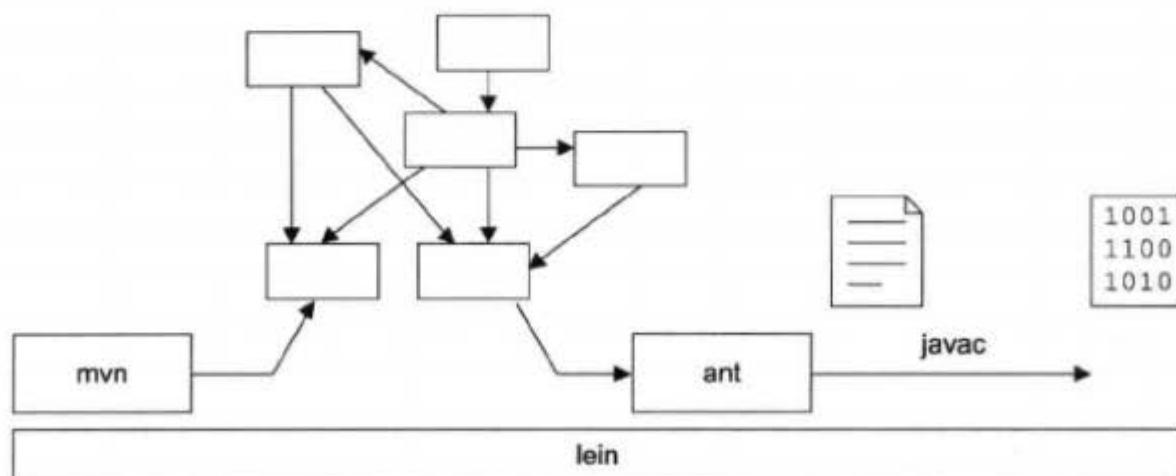


图12-16 Leiningen及其组件

高级用户可以穿过抽象层,直接使用Leiningen的底层工具。但Leiningen的基本语法和应用非常简单,不需要使用者具备使用任何底层工具的经验。

我们来看一个简单的例子，看看project.clj文件如何工作，以及在Leiningen项目生命周期中如何使用那些基本的命令。

### 12.6.3 Hello Lein

把lein放在PATH上之后，我们可以用它的new命令开始一个新项目：

```
ariel:projects boxcat$ lein new hello-lein
Created new project in: /Users/boxcat/projects/hello-lein
ariel:projects boxcat$ cd hello-lein/
ariel:hello-lein boxcat$ ls
README  project.clj  src  test
```

这个命令创建了一个叫做hello-lein的项目。它有项目目录，里面有个简单的描述文件README、一个project.clj文件（马上就会详细讨论），还有并列的src和test目录。

如果你把Leiningen刚刚创建的项目导入Eclipse中（比如用CounterClockwise插件），项目的布局应该如图12-17所示。



图12-17 新创建的Leiningen项目

这个项目结构是直接从Java项目上照搬过来的：有带有core.clj文件的并列test和src结构（分别用于测试和顶层代码）。另外一个重要的文件是project.clj，Leiningen用它来控制构建、保存元数据。

我们来看一下lein的新命令生成的骨架文件。

```
(defproject hello-lein "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]])
```

这个Clojure形式解析起来相当直白：有一个(defproject)的宏负责制作表示Leiningen项目的新值。这个宏需要知道项目名称（在这里是hello-lein），还需要知道项目的版本（默认是1.0.0-SNAPSHOT，12.3.1节讨论过的Maven版本号），然后是描述项目的元数据映射。

lein自带了两个元数据：一个描述字符串和一个依赖项向量，后者对于添加新的依赖项很方便。我们现在就来加一个clj-time类库。这个类库为Clojure提供Java日期和时间类库（Joda-Time，但在这个例子中你没必要知道这个Java类库）的访问接口。加上新的依赖项后，project.clj看起来应该是这样的：

```
(defproject hello-lein "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]
    [clj-time "0.3.0"]])
```

向量的第二个元素描述了要用的新依赖项类库版本。如果Leiningen在本地依赖项资源库中找不到它，会按这个版本从外部资源库获取该依赖项。

Leiningen默认从位于http://clojars.org/的资源库获取缺失的类库。因为Leiningen底层用的是Maven，所以这本质上就是一个Maven资源库。Clojars提供了一个搜索工具，可以在你知道所需类库但不知道具体版本号时提供帮助。

在这个新的依赖项就位后，你需要更新本地构建环境，可以执行lein deps命令。

```
ariel:hello-lein boxcat$ lein deps
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.pom from central
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.pom from clojure
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.pom from clojars
Transferring 2K from clojars
Downloading: joda-time/joda-time/1.6/joda-time-1.6.pom from clojure
Downloading: joda-time/joda-time/1.6/joda-time-1.6.pom from clojars
Transferring 5K from clojars
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.jar from central
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.jar from clojure
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.jar from clojars
Transferring 7K from clojars
Downloading: joda-time/joda-time/1.6/joda-time-1.6.jar from clojure
Downloading: joda-time/joda-time/1.6/joda-time-1.6.jar from clojars
Transferring 522K from clojars
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
ariel:hello-lein boxcat$
```

Leiningen已经用Maven下载了Clojure的接口，还有底层的Joda-Time JAR。我们在代码中用一下它，展示在依赖项存在的情况下如何用Leiningen作为REPL进行开发。

需要把主要源文件src/hello\_lein/core.clj改成下面这样：

```
(ns hello-lein.core)
(use '[clj-time.core :only (date-time)])
(defn isodate-to-millis-since-epoch [x]
  (.getMillis (apply date-time
  (= (map #(Integer/parseInt %) (.split x "-")))))
```

它提供了一个Clojure函数，将ISO标准日期（格式为YYYY-MM-DD）转换成自Unix纪元（1970年）以来的毫秒数。

我们用Leiningen的REPL风格测试一下。先在project.clj文件中加上一行，改成下面这样：

```
(defproject hello-lein "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]
    [clj-time "0.3.0"]]
  :repl-init hello-lein.core)
```

加上这一行后，可以启动一个能访问所有依赖项的REPL，并且它已经把命名空间hello-lein.core中的函数引入了作用域：

```
ariel:hello-lein boxcat$ lein repl
REPL started; server listening on localhost:10886.

hello-lein.core=> (isodate-to-millis-since-epoch "1970-01-02")
86400000
hello-lein.core=>
```

这是以天为单位的日期的正确毫秒数，并且它阐明了在真实项目中使用REPL的核心原则。我们在这上面稍微展开一点，再看一个使用Leiningen REPL面向测试的工作方式。

#### 12.6.4 用 Leiningen 做面向 REPL 的 TDD

任何优秀的TDD方法，其核心都应该是这样一个用来开发新功能的简单基本的循环。具体到Clojure和Leiningen，其基本循环应该如下所示：

- (1) 添加任何所需的新依赖项（并重新运行lein deps）；
- (2) 启动REPL（lein repl）；
- (3) 草拟一个新函数，并把它放到REPL的作用域中来；
- (4) 在REPL内测试这个函数；
- (5) 重复步骤3和4，直到该函数表现正确；
- (6) 把该函数的最终版加到恰当的.clj文件上；
- (7) 把刚才运行的测试用例加到测试集.clj文件中；
- (8) 重启REPL，再次从第三步开始（或者第一步，如果需要新的依赖项）。

这是测试驱动开发的风格，但却避开了先写测试还是先写代码的问题，用REPL风格的TDD，这两件事是同时进行的。

之所以要在添加新函数时重启REPL（第八步），是为了能干净地编译新函数。创建新函数时，有时为了支持它，会对其他函数或环境做轻微的修改。而这些修改在把函数加入最终的源码库时很容易被忘掉。重启REPL能帮我们尽早记起这些被忘掉的修改。

这个过程清晰而简单，但还有个问题我们没提到，无论是在这里还是第11章讨论TDD时，我们都还没讨论过怎么编写Clojure测试。好在这非常简单。我们来看一下用lein new创建新项目时它所提供的模板：

```
(ns hello-lein.test.core
  (:use [hello-lein.core])
  (:use [clojure.test]))

(deftest replace-me ;; FIXME: write
  (is false "No tests have been written.))
```

我们就用lein test命令来测试这个自动生成的用例，看看会发生什么（实际上你应该能猜出来）。

```
ariel:hello-lein boxcat$ lein test
Testing hello-lein.test.core
FAIL in (replace-me) (core.clj:6)
No tests have been written.

expected: false
actual: false
Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
```

如你所见，自动生成的测试用例失败了，并且它絮叨着让你写些测试用例。那就写吧，在test文件夹里写个core.clj文件：

这个测试非常简单：使用了(deftest)宏，给测试命名为(one-day)，并且有一个跟断言语句非常相似的形式。Clojure代码的结构使得(is)形式读起来非常自然——几乎就像DSL一样。这个测试可以读作“自1970年1月2日以来的毫秒数等于86 400 000对吗？”我们来看一下这个测试的实际效果：

```
(ns hello-lein.test.core
  (:use [hello-lein.core])
  (:use [clojure.test]))

(deftest one-day
  (is true
    (= 86400000 (isodate-to-millis-since-epoch "1970-01-02"))))
```

这里的关键包是clojure.test，它提供了一些在更复杂的环境或需要用到测试固件时用来构建测试用例的形式。如果想了解得更深入，请参考Amit Rathore写的*Clojure in Action* ( Manning, 2011 )，其中对Clojure中的TDD有全面的论述。

```
ariel:hello-lein boxcat$ lein test
Testing hello-lein.test.core
Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

在面向REPL的TDD流程就绪后，现在可以用Clojure构建一个具有相当规模且能用的应用程序了。但你终归要做一些需要跟人分享的东西。好在Leiningen有一些命令可以让你很容易地进行打包和部署。

## 12.6.5 用 Leiningen 打包和部署

Leiningen主要提供了两种代码分发办法。这两种办法本质上的区别是带不带依赖项。对应的命令分别是lein jar和lein uberjar。

我们先看一下lein jar：

```
ariel:hello-lein boxcat$ lein jar
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
Created /Users/boxcat/projects/hello-lein/hello-lein-1.0.0-SNAPSHOT.jar
```

下面这些是被打包进JAR文件中的东西：

```
ariel:hello-lein boxcat$ jar tvf hello-lein-1.0.0-SNAPSHOT.jar
  72 Sat Jul 16 13:38:00 BST 2011 META-INF/MANIFEST.MF
 1424 Sat Jul 16 13:38:00 BST 2011 META-INF/maven/hello-lein/hello-lein/
   pom.xml
  105 Sat Jul 16 13:38:00 BST 2011
META-INF/maven/hello-lein/hello-lein/pom.properties
  196 Fri Jul 15 21:52:12 BST 2011 project.clj
  238 Fri Jul 15 21:40:06 BST 2011 hello_lein/core.clj
ariel:hello-lein boxcat$
```

其中最明显的就是Leiningen的基本命令把Clojure源文件，而不是编译后的.class文件发出去了。这是Lisp代码的传统，因为系统的读时组件和宏会因为要处理编译后的代码而受到阻碍。

现在，我们来看看用lein uberjar会发生什么。它所产生的JAR不仅包含代码，还有依赖项。

```
ariel:hello-lein boxcat$ lein uberjar
Cleaning up.
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
Created /Users/boxcat/projects/hello-lein/hello-lein-1.0.0-SNAPSHOT.jar
Including hello-lein-1.0.0-SNAPSHOT.jar
Including clj-time-0.3.0.jar
Including clojure-1.2.1.jar
Including clojure-contrib-1.2.0.jar
Including joda-time-1.6.jar
Created /Users/boxcat/projects/hello-lein/
➥ hello-lein-1.0.0-SNAPSHOT-standalone.jar
```

看到了吧，这个JAR中不仅有代码，还有依赖项，以及依赖项的依赖项，这称为依赖的传递闭包图。也就是说它是一个可以完全独立运行的包。

当然，因为所有依赖项都打包了，所以这也意味着lein uberjar打包的文件要比lein jar的文件大很多。即便是我们这个简单的小例子，其差异也相当鲜明：

```
ariel:hello-lein boxcat$ ls -lh h*.jar
-rw-r--r-- 1 boxcat staff 4.1M 16 Jul 13:46
hello-lein-1.0.0-SNAPSHOT-standalone.jar
-rw-r--r-- 1 boxcat staff 1.7K 16 Jul 13:46
hello-lein-1.0.0-SNAPSHOT.jar
```

你可以这样理解lein jar和lein uberjar：如果要构建一个类库（构建在其他类库之上），或者要将它作为依赖项，就用lein jar。如果是构建一个最终用户使用的Clojure应用程序，而不是交给用户去扩展的工件，就用lein uberjar。

你已经见过用Leiningen如何开始、管理、构建和部署Clojure项目了。Leiningen还有很多内置的实用命令，还有一个强大的插件系统让你可以对它进行定制。想要对Leiningen能做什么有更多了解，只要调用时不带命令就可以了，单用lein。

我们在下一章构建Clojure的Web应用时还会遇到Leiningen。

## 12.7 小结

有优秀Java开发人员参与的项目肯定有快速、可重复、简单的构建。如果软件不能快速稳定地构建，就会浪费大量的时间和金钱，包括你自己的！

理解编译—测试—打包这一基本构建周期是确立良好构建流程的关键。毕竟，你不能测试还没编译的代码！

Maven将构建周期的概念发扬光大，扩展为在所有Maven项目中都能保持一致的项目周期。这种惯例优先（于配置）的方式对于大型软件团队非常有帮助，但有些项目可能需要更多的灵活性。

Maven还解决了依赖管理的问题，因为几乎所有项目都要依赖第三方类库，这个难题一直困扰着Java和开源世界。

把构建流程挂到CI环境中，开发人员就能得到迅捷无比的反馈，还可以毫无畏惧地快速合并修改。

Jenkins是一个流行的CI服务器，不仅能构建几乎所有类型的项目，还能通过它庞大的插件系统提供丰富的报告。假以时日，开发团队就能让Jenkins执行各种构建，覆盖范围可以从快速单元测试构建到系统集成构建。

Leiningen是Clojure项目的自然之选。它用一个非常清爽的构建和部署工具，把紧凑的TDD循环和REPL方式结合在了一起。

我们接下来会讨论快速Web开发，自从第一个基于Java的Web框架出现以来，大多数优秀的Java开发人员都曾为这一主题奋斗过。

# 快速Web开发

# 13

## 本章内容

- 为什么Java不是快速Web开发的理想选择
- Web框架的选择标准
- 基于JVM的Web框架比较
- 认识Grails（与Groovy）
- 认识Compojure（与Clojure）

快速Web开发很重要，非常重要。在全球的商业和社交活动中，数量庞大的网站和由Web技术驱动的应用程序占据着主导地位。企业（特别是创业公司）的生死取决于他们向市场投放新产品或新特性的速度。如今的终端用户希望新功能的出现和bug的消失能像变魔术一样快，他们越来越没耐心了。

可大多数Java上的Web框架在支持快速Web开发的能力上都有限，为了不在激烈的竞争中死掉，很多组织都纷纷转向PHP和Rails之类的技术。

作为一个优秀的Java开发人员，你该何去何从？好在最近JVM上出现了动态层语言，现在JVM上也有快速Web开发的理想选择。Grails（Groovy）和Compojure（Clojure）就是这样的框架，它们能满足你要求的快速Web开发能力。也就是说你不用放弃强大而又灵活的JVM，在跟PHP和Rails这样的技术竞争时也不用比它们多花几个小时了。

### Java EE 6：Java的快速Web开发是否向前迈进了一步？

相比J2EE（曾因JSP、Servlet和EJB API饱受诟病），Java企业版（Java EE）6已经有了长足的发展。尽管Java EE 6所做的改进（在JSP、Servlet和EJB API上有明显体现）仍然受限于Java静态类型系统和编译方面的问题。

本章一开始会解释一下为什么Java上的Web框架不是快速Web开发的理想选择。顺着这个解释，你会了解优秀的Web框架应该满足哪些标准。通过一些定量的研究，以及Matt Raible的工作，你会理解如何用Web框架的20条标准对各种JVM Web框架进行评级。

Grails是快速Web开发框架的领导者之一，它满足了其中的很多标准。我们会带你过一遍这个基于Groovy的Web框架，炙手可热的Rails框架对它产生了很大的影响。

我们还会讨论作为Grails备选的Compojure，它是一个基于Clojure的Web框架，可以实现非常精炼的Web编程和快速开发。

让我们先来看看为什么基于Java的Web框架不一定是现代Web项目的理想选择。

## 13.1 Java Web框架的问题

第7章讨论过多语言编程金字塔和编程语言的三个层次，我们在图13-1中再次重复一下。



图13-1 多语言编程金字塔

Java位于稳定层，所以它的所有Web框架也属于这一层。作为一门流行而又成熟的语言，Java有很多Web框架，比如：

- Spring MVC
- GWT
- Struts 2
- Wicket
- Tapestry
- JSF（及其他与Faces相关的类库）
- Vaadin
- Play
- 以前那些普通的JSP/Servlet

在这一领域，Java没有公认的领头羊，并且源于Java的这一分支根本就不是快速Web开发的理想选择。Struts 2曾经流行一时，该项目的前任领导者说过这样一段话：

我堕落了:-)，我现在更喜欢用Rails——因为前面提到的简洁性，因为我再也不用“构建”和“部署”了。兄弟姐妹们，我要提醒你们，如果你们想吸引Rails开发人员……或者要避免像我这样的“背叛Java的Web开发人员”流失:-)，这类事情是你们必须克服的障碍。

——Craig McClanahan，2007年10月23日  
 ( <http://markmail.org/thread/qfb5sekad33eobh2> )

本节会谈到Java为什么不是快速Web开发的理想选择。我们先来看看编译型语言为什么会在开发Web应用时拖后腿。

### 13.1.1 Java 编译为什么不好

Java是编译型语言，这就是说你每次修改代码，都必须重复下面的步骤：

- 重新编译Java代码；
- 停止Web服务器；
- 把改过的应用重新部署到Web服务器中；
- 启动Web服务器。

这会浪费大量的时间！特别是有很多小修改时，比如修改控制权的目标或界面的微调。

**注意** 应用服务器和Web服务器之间的界限已经变得模糊了。这是因为JEE 6的出现（可以在Web容器内运行EJB），也因为大多数应用服务器都是高度模块化的。我们提到的“Web服务器”是指任何有Servlet容器的服务器。

如果你是一位经验丰富的Web开发人员，应该知道有些技术可以解决这个问题。其中大多数都是不用停止和启动Web服务器就能应用代码修改，也被称为热部署。热部署或者把全部资源（比如整个WAR文件）都换掉，或者只选其中几个（比如单个JSP页面）资源进行替换。但热部署不是100%可靠（因为类加载的限制和容器中的bug），并且大多数情况下，Web服务器仍然需要执行昂贵的代码重编译操作。

#### 用JRebel和LiveRebel执行热部署

如果你必须要用Java Web框架，我们强烈向你推荐JRebel和LiveRebel（<http://www.zeroturnaround.com/jrebel/>）。JRebel确实具备一些惊艳的JVM技巧，它处在IDE和Web服务器之间，源码发生变化时能自动将这些变化反应到正在运行的Web服务器上（LiveRebel用于生产环境的部署）。这种热部署基本没什么问题，并且这些工具实际上是解决热部署问题的行业标准。

一般来说，Java Web框架为代码修改而产生的周转时间太长。但这不是唯一的问题，另外一个拖慢Web开发速度的不利因素是语言的灵活性，这是静态类型系统的弱项。

### 13.1.2 静态类型为什么不好

在开发新产品或新功能的早期阶段，保持用户展示层设计的开放性（对于类型而言）通常都是明智之举。对于用户来说，要求数值精确到小数位，或让书单变成图书和玩具混合的清单都是非常合理的要求。静态类型可能会成为这类需求的巨大障碍。如果你必须把一个Book对象列表

变成BookOrToy<sup>①</sup>对象的列表，则只能在代码中把这个静态类型全改掉。

尽管在容器类里能用基本类型（比如Java的Object类）作为对象的类型，但这肯定不是最佳实践——这简直是一下子回到了泛型。

因此，选择基于动态层语言编写的Web框架无疑是个有效选项。

**注意** Scala当然是静态类型语言。但由于其先进的类型推断能力，它能规避很多由Java静态类型实现方式引发的问题。也就是说，Scala可以是、也确实是可用的Web层语言。

在你继续深入研究和选择Web开发的动态语言之前，我们先退一步，让视野更开阔一点。想一想优秀的快速Web开发框架应该符合哪些标准。

## 13.2 选择Web框架的标准

Java这么多年的顶级编程语言地位不是白给的，Java中可供选择的Web框架有很多。最近基于其他JVM语言（比如Groovy、Scala和Clojure）的Web框架也崛起了。但不幸的是，这么多年来还没有一个能在这一领域确立自己的霸主地位，选哪个框架完全看个人偏好。

Web框架应该能提供大量帮助。必须能用一些标准进行评估，它能满足的标准越多，开发Web应用的速度可能会越快。

Matt Raible<sup>②</sup>总结出了评判Web框架的20条标准<sup>③</sup>。表13-1对这些标准作了简要介绍。

表13-1 Web框架的20条标准

标 准	示 例
开发人员的工作效率	能用1天或5天搭出一个CRUD页面吗
开发人员的看法	用起来有意思吗
学习曲线	学了一个礼拜或一个月后能干活吗
项目健康状况	项目陷入绝境了吗
开发人员的充足性	能找到经验丰富的开发人员吗
就业趋势	将来能招到人吗
模板化	遵循DRY（不重复自己）原则吗
组件	自带日期选择器之类的东西吗

① 小心！如果你的域对象中有Or或And这样的字眼出现，那你很可能违反了我们在第11章讨论的SOLID原则。

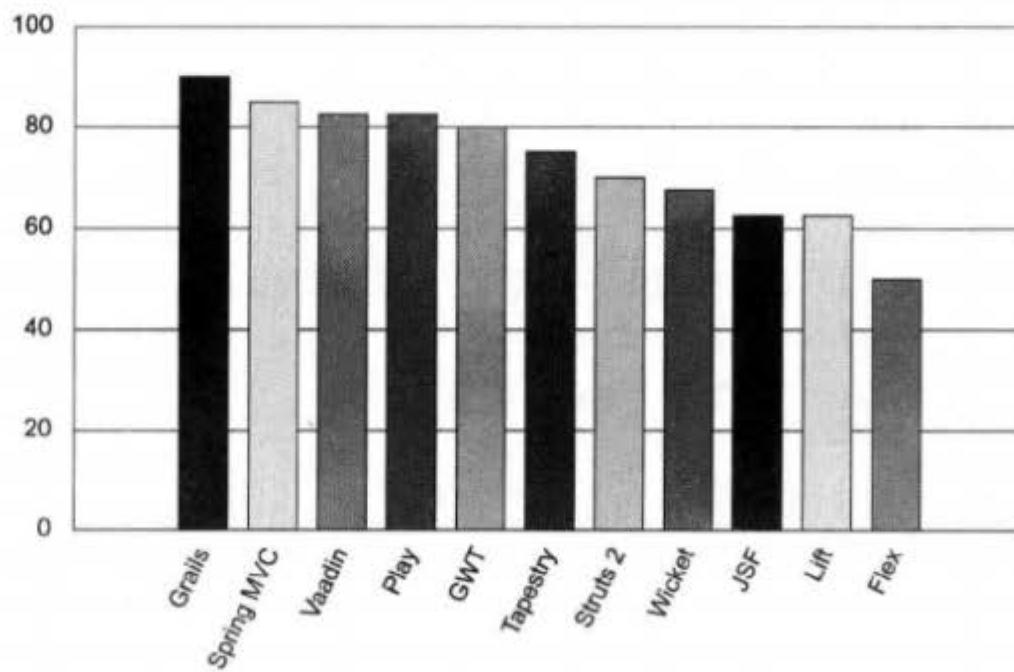
② AppFuse (<http://appfuse.org>) 的作者，AppFuse是一个Web开发基础平台，它集成了Java中各种流行的Web框架，并提供了所有Web系统开发过程中都需要开发的一些功能，比如登录、用户密码加密、用户管理、为不同的用户展现不同的菜单。它可以自动生成40%~60%的代码，还自带了一些默认的CSS样式，使用这些样式能快速改变整个系统的外观，还具备自动测试的能力。——译者注

③ Matt Raible, “Comparing JVM Web Frameworks” (March 2011), presentation. <http://raibledesigns.com/rd/page/publications>。

(续)

标 准	示 例
Ajax	支持客户端的异步Javascript调用吗
插件或附加项	能加上Facebook集成之类的功能吗
扩展性	默认的控制器处理的并发用户数能达到500+吗
测试支持	能做测试驱动的开发吗
I18n和I10n	自带其他语种和地域的支持吗
校验	能轻松校验用户输入并迅速反馈吗
多语言支持	能同时用(比如说)Java和Groovy吗
文档/教程的质量	常见的用例和问题在文档中都有体现吗
出版图书	有没有行业专家用过它，并分享了自己的战斗事迹
REST支持(服务器端和客户端)	它能按HTTP的设计宗旨使用该协议吗
移动支持	是否很容易就能支持Android、iOS和其他移动设备
风险程度	是用来做“保存食谱”的应用程序或是“核电站控制器”

你看到了，这个清单很长，在做决定时，你需要想好各个标准的权重。不过Matt很勇敢<sup>①</sup>，他最近在这一领域做了一些研究，尽管其研究结果引发了激烈的争论，但真相总算是开始浮现了。如果给那些对快速Web开发最重要的标准赋予较高的权重，各种框架的得分(总分为100)如图13-2所示。这些标准应该是：开发人员的工作效率、测试支持和文档的质量。



不同的人需求可能会不同，在`http://bit.ly/jvm-frameworks-matrix`上可以很容易地修改Matt的权重，运行自己的分析，产生自己的图形。

① 你应该能想象得到，人们对于自己喜爱的Web框架有多大的热情！

**提示** 在你选定框架之前，我们强烈建议你在两到三个框架上按自己的标准做一些功能原型。

现在你知道该用哪些标准进行评估了，并且还能利用Matt提供的工具，所以在选择快速Web开发框架时，你可以做出明智的选择。在我们的加权标准分析中脱颖而出的是Grails框架（Compojure没有名列前茅，但因为它还非常年轻，所以我们预计它在不久的将来能迅速蹿升到领导阵营中）。

我们来看看获胜者Grails！

### 13.3 Grails入门

Grails是基于Groovy的快速Web应用框架，它集成了多个第三方类库，包括Spring、Hibernate、JUnit和Tomcat服务器等。它是一个完备的Web框架，13.2节列出的20条标准它全都满足。还有一点很重要，Grails在很大程度上借鉴了Rails中惯例优先的原则。如果能依照惯例编码，框架会帮你做很多套路化的工作。

我们在这一节中会讨论如何搭建你的第一个快速启动应用。在搭建快速启动应用的过程中，你会看到很多可以证明Grails“快速”的证据。我们还会指出Grails中那些需要进一步探索的重要技术，从而让你可以构建出能用于生产环境的、正儿八经的应用程序。

#### 不喜欢Groovy？试试Spring Roo

Spring Roo ([www.springsource.org/roo](http://www.springsource.org/roo)) 是跟Grails基于同样原则开发的快速Web开发框架，但它的核心语言是Java，并且向开发者开放了更多的Spring DI框架。我们认为它没Grails成熟，但如果你确实不喜欢Groovy，这也是个不错的备选。

如果不熟悉Groovy，可能需要认真温习一下第8章。在你能跟Groovy融洽相处后，请下载Grails并安装。附录C中有该过程的完整指导。

装好Grails之后，就该开始你的第一个Grails项目了！

### 13.4 Grails快速启动项目

这一节会介绍一个Grails快速启动项目，重点展示Grails作为快速Web框架的亮点。用Grails创建Web应用所需的步骤如下：

- 创建域对象；
- 测试驱动开发；
- 域对象的持久化；
- 创建测试数据；
- 控制器；

- GSP视图；
- 脚手架和自动化的UI创建；
- 快速开发的周转时间。

说得具体点，我们准备搞一个角色扮演游戏<sup>①</sup>中的基本构件（PlayerCharacter）。到本节结束的时候，你会创建一个具备以下能力的简单的域对象（PlayerCharacter）：

- 进行一些运行时测试；
- 预先准备好测试数据；
- 可以保存到数据库中；
- 具有可以进行CRUD操作的基本UI。

Grails节省时间的第一个法宝就是自动创建好项目结构。运行grails create-app <my-project>命令，马上就能得到一个可以构建的项目！你需要做的唯一一件事情就是保证能接入互联网，因为它要下载标准的Grails依赖项（比如Spring、Hibernate、JUnit、Tomcat服务器等）。

Grails用来管理和下载依赖项的工具是Apache Ivy。它下载和管理依赖项的概念跟第12章介绍的Maven非常像。下面这个命令会创建一个叫做pcgen\_grails的应用程序，包括一个依照Grails的传统优化过的项目结构。

```
grails create-app pcgen_grails
```

依赖项下载完成，其他自动安装步骤也完成之后，你应该就会得到一个如图13-3所示的项目结构。

```
pcgen_grails
  application.properties    --> basic application info/versioning
  + grails-app
    + conf                  --> location of configuration artifacts
    + hibernate             --> optional hibernate configuration
    + spring                --> optional spring configuration
    + controllers           --> location of controller artifacts
    + domain                --> location of domain classes
    + i18n                  --> location of message bundles for i18n
    + services              --> location of services
    + taglib                --> location of tag libraries
    + util                  --> location of special utility classes
    + views
      + layouts             --> location of views
                                --> location of layouts
  + lib
  + scripts                --> scripts
  + src
    + groovy               --> optional; location for Groovy source files
                                (of types other than those in grails-app/*)
    + java                 --> optional; location for Java source files
    + test                  --> generated test classes
  + web-app
    + WEB-INF
```

图13-3 Grails项目的布局

有了项目结构就可以开始生产一些能运行的代码了！首先要创建域对象类。

<sup>①</sup> 想想《龙与地下城》或《指环王》。

### 13.4.1 创建域对象

Grails以域对象为应用程序的核心，因此鼓励你按域驱动设计（Domain-Driven Design, DDD）的方式来考虑问题<sup>①</sup>。执行grails create-domain-class命令可以创建域对象。

下面的例子创建了一个PlayerCharacter类，用来表示游戏中的角色：

```
cd pcgen_grails
grails create-domain-class com.java7developer.chapter13.PlayerCharacter
```

Grails会自动为你创建下面的文件：

- 一个表示域对象的PlayerCharacter.groovy源文件（在目录grails-app/domain/com/java7developer/chapter13下）；
- 开发单元测试用的PlayerCharacterTests.groovy源文件（在目录test/unit/com/java7developer/chapter13下）。

看，Grails在鼓励你写单元测试！

还需要给PlayerCharacter定义一些属性，比如strength、dexterity和charisma。有了这些属性，你就可以开始构想游戏中的角色如何跟想象的世界交互<sup>②</sup>。但刚刚看过第11章，你当然想先写测试！

### 13.4.2 测试驱动开发

按TDD的方式，我们要先写个失败测试，然后实现PlayerCharacter让测试通过。

我们还准备利用Grails的域对象自动校验特性。在Grails中，可以自动在任何域对象上调用validate()方法，以确保该对象的有效性。代码清单13-1会测试strength、dexterity和charisma三项统计量都是3到18之间的数值。

#### 代码清单13-1 PlayerCharacter的单元测试

```
package com.java7developer.chapter13
import grails.test.*
class PlayerCharacterTests extends GrailsTestCase {
    PlayerCharacter pc;
    protected void setUp() {
        super.setUp()
        mockForConstraintsTests(PlayerCharacter)
    }
    protected void tearDown() {
        super.tearDown()
    }
}
```

<sup>①</sup> 想了解DDD（由Eric Evans提出）的更多内容，请访问域驱动设计社区（<http://domaindrivendesign.org/>）。

<sup>②</sup> Gweneth是不是应该善于摔跤、杂耍，或面带微笑地解除对手的武装？

```

void testConstructorSucceedsWithValidAttributes {
    pc = new PlayerCharacter(3, 5, 18)
    assert pc.validate()
}

void testConstructorFailsWithSomeBadAttributes() {
    pc = new PlayerCharacter(10, 19, 21)
    assertFalse pc.validate()
}

```

Grails的单元测试都应该扩展自GrailsUnitTestCase①。跟所有标准的JUnit测试一样，它也有setUp()和tearDown()方法。但为了在单元测试阶段用Grails内置的validate()方法，必须通过mockForConstraintsTest方法把它拉进来②。这是因为Grails把validate()当做集成测试的关注点，通常只有这样才能用它。但如果想要更快地得到反馈，可以把它放到单元测试阶段。接下来，可以调用validate()来检查域对象是否有效③④。

现在可以执行下面的命令来运行测试了：

```
grails test-app
```

这个命令既运行单元测试也会运行集成测试（不过我们现在只有单元测试），并且从控制台中的输出来看，测试失败了。

要了解测试失败的原因，需要到target/test-reports/plain目录下去找。对于这个程序，要找到TEST-unit-unit-com.java7developer.chapter13.PlayerCharacterTests.txt文件。这个文件会告诉你测试失败是因为在尝试创建新的PlayerCharacter时，没找到匹配的构造方法。这很容易理解，因为PlayerCharacter域对象还有什么都没有呢！

现在你可以把PlayerCharacter搭起来，重复运行测试直到通过。按你的想法加上strength、dexterity和charisma三个属性。但为了在这些属性上设定minimum(3)和maximum(18)的限制，需要用特殊的限定语法。那样就可以用Grails提供的默认validate()方法了。

### Grails中的限定

Grails中的限定是在Spring validator API基础上实现的。可以用它们指定域类型属性的校验需求。Grails的限定很多（在代码清单13-2中用到了min和max），你还可以根据需要自行编写限定。参见<http://grails.org/doc/latest/guide/validation.html>了解详情。

下面这段代码中的PlayerCharacter类中仅包含了让它可以通过测试的最基本的属性和限定。

#### 代码清单13-2 PlayerCharacter类

```

package com.java7developer.chapter13

class PlayerCharacter {

    Integer strength
    Integer dexterity
    Integer charisma

    PlayerCharacter() {}
}

```

① 要持久化的类型变量

```

PlayerCharacter(Integer str, Integer dex, Integer cha) {
    strength = str
    dexterity = dex
    charisma = cha
}

static constraints = {
    strength(min:3, max:18)
    dexterity(min:3, max:18)
    charisma(min:3, max:18)
}
}

```

② 可以通过测试的构造方法

③ 用于校验的限定

PlayerCharacter类相当简单。有三个会自动保存到PlayerCharacter表中的基本属性①。有一个带三个参数的构造方法②。那个特殊的static代码块确定了validate()方法要检查的min和max值③。

PlayerCharacter类变具体后，测试应该能很痛快地通过了（再次运行grails test-app）。如果遵循TDD方式，到这个阶段就该着手重构PlayerCharacter和测试了，以便让代码更加清爽。

Grails还会确保域对象保存到数据存储中。

### 13.4.3 域对象持久化

持久化是由Grails自动处理的，因为Grails认为类中所有具有明确类型的域变量都应该保存到数据库中。Grails会自动把域对象映射到同名的表中。对于PlayerCharacter域对象而言，三个属性（strength、dexterity和charisma）全部是Integer类型，所以都会映射到PlayerCharacter表中。Grails默认使用Hibernate，并会提供一个HSQLDB内存数据库（我们在第11章提到过它，那时用做伪装测试替身），但你可以用自己的数据源取代默认数据源。

grails-app/conf/DataSource.groovy文件里是数据源的配置。可以在这里为每种环境设定数据源。记住，Grails已经在pcgen\_grails里给出了默认使用HSQLDB的实现，所以无需任何修改就可以运行它。但代码清单13-3中给出了使用其他数据库的配置供参照。

#### 代码清单13-3 可能的pcgen\_grails数据源

```

dataSource {}

environments {
    development { dataSource {} }
    test { dataSource {} }

    production {
        dataSource {
            dbCreate = "update"
            driverClassName = "com.mysql.jdbc.Driver"
            url = "jdbc:mysql://localhost/my_app"
            username = "root"
            password = ""
        }
    }
}

```

← 生产数据源

← 数据库驱动

← JDBC连接URL

比如说，可以在生产环境中使用MySQL数据库，而开发和测试环境中还用HSQLDB。这些都是相当标准的Java数据库连接（JDBC）配置，你对它们应该已经很熟悉了。

Grails开发者也考虑到了手工创建测试数据的问题，所以他们提供了一种机制，可以在应用启动时将数据预填充到数据库中。

#### 13.4.4 创建测试数据

测试数据的创建通常是由Grails的Bootstrap类完成的，它在grails-app/conf/Bootstrap.groovy中。只要Grails应用或Servlet容器启动，就会运行init方法。这和大多数Java Web框架用的启动servlet所起的作用是一样的。

**注意** 可以用Bootstrap类做所有初始化操作，但现在我们主要讨论测试数据。

代码清单13-4在初始化阶段生成了两个PlayerCharacter域对象，并把它们存到了数据库里。

#### 代码清单13-4 为pcgen\_grails引导测试数据

```
import com.java7developer.chapter13.PlayerCharacter
class BootStrap {
    def init = { servletContext ->
        if (!PlayerCharacter.count()) {
            new PlayerCharacter(strength: 3, dexterity: 5, charisma: 18)
                .save(failOnError: true)
            new PlayerCharacter(strength: 18, dexterity: 10, charisma: 4)
                .save(failOnError: true)
        }
    }
    def destroy = {}
}
```



① 在Servlet上下文  
启动时引导

每次把代码部署到Servlet容器中都会执行init方法（即应用启动和Grails自动部署时）①。为了确保不会覆盖掉任何已有数据，可以对已有的PlayerCharacter实例执行简单的count()方法。如果确定没有实例，可以创建一些。这里有个很重要的特性：如果有异常抛出，或所构造的对象无法通过校验，则可以肯定对象不会保存到数据库中。如果愿意，可以在destroy方法中执行清除操作。

有了一个带有存储支持的基本域对象后就可以进入下一阶段了：在Web页面上显示域对象。为此需要构建一个Grails控制器，你应该不会对这个源自MVC设计模式的术语感到陌生。

#### 13.4.5 控制器

Grails遵循MVC设计模式，用控制器来处理来自客户端（一般是浏览器）的Web请求。Grails的惯例是给每个域对象配一个控制器。

要创建域对象PlayerCharacter的控制器只需要执行下面这条命令：

```
grails create-controller com.java7developer.chapter13.PlayerCharacter
```

重要的是指明域对象的完全限定类名，包括包名。

命令执行完成后应该能发现下面这些文件：

- PlayerCharacter域对象的控制器的PlayerCharacterController.groovy源文件（在grails-app/controller/com/java7developer/chapter13目录下）；
- 开发控制器单元测试的PlayerCharacterControllerTests.groovy源文件（在test/unit/com/java7developer/chapter13目录下）；
- grails-app/view/playerCharacter文件夹（稍后会用到）。

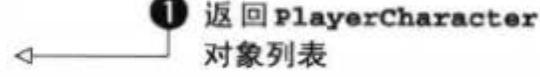
控制器以简单的方式支持REST风格的URL和操作映射。假设要把REST风格的URL http://localhost:8080/pcgen\_grails/playerCharacter/list映射到一个返回PlayerCharacter对象列表的方法上。按照Grails惯例优于传统的方式可以用最少的源码把URL映射到PlayerCharacterController类中。这个URL是由下面这些元素组成的：

- 服务器（http://localhost:8080/）；
- 基础项目（pcgen\_grails/）；
- 控制器名称的衍生部分（playerCharacter/）；
- 在控制器里声明的操作块变量（list）。

要在代码中看到这些元素，请用代码清单13-5替换已有的PlayerCharacterController.groovy源码。

#### 代码清单13-5 PlayerCharacterController

```
package com.java7developer.chapter13
class PlayerCharacterController {
    List playerCharacters
    def list = {
        playerCharacters = PlayerCharacter.list() } }
```



**①** 返回PlayerCharacter对象列表

使用Grails的惯例处理方式，playerCharacter的属性会用在REST风格的URL指向的页面中**①**。

但如果现在就启动程序，然后访问http://localhost:8080/pcgen\_grails/playerCharacter/list，是不会成功的，因为还没创建JSP或GSP页面。现在我们就来解决这个问题。

#### 13.4.6 GSP/JSP 页面

用Grails既可以创建GSP页面，也可以创建JSP页面。这一节会创建一个简单的GSP页面，用来显示PlayerCharacter对象的列表（设计师、Web开发者和HTML/CSS大拿们，现在请移开你们的视线！）

代码清单13-6是GSP页面grails-app/view/playerCharacter/list.gsp的代码。

## 代码清单13-6 PlayerCharacter列表的GSP页面

```

<html>
  <body>
    <h1>PC's</h1>
    <table>
      <thead>
        <tr>
          <td>Strength</td>
          <td>Dexterity</td>
          <td>Charisma</td>
        </tr>
      </thead>
      <tbody>
        <% playerCharacters.each({ pc -> %>
          <tr>
            <td><%"${pc?.strength}"%></td>
            <td><%"${pc?.dexterity}"%></td>
            <td><%"${pc?.charisma}"%></td>
          </tr>
          <% } ) %>
        </tbody>
      </table>
    </body>
</html>

```

HTML非常简单，关键是如何用Groovy脚本。你会注意到我们在第8章介绍的Groovy函数字面值语法，它简化了集合循环操作①。接着是对角色属性的引用（注意安全的null解引用操作符的使用）②，然后结束函数字面值③。

既然域对象、控制器和它的显示页面都准备好了，接下来就可以启动Grails应用了！执行下面这条命令即可：

```
grails run-app
```

Grails会自动在http://localhost:8080上启动一个Tomcat，并把pcgen\_grails应用部署上去。

**警告** 很多开发人员已经装过Tomcat服务器了。如果想同时启动多个Tomcat实例，就要修改端口号，端口8080只能有一个实例监听。

如果你打开浏览器访问http://localhost:8080/pcgen\_grails/，会看到页面上列出了PlayerCharacterController，如图13-4所示。

点击com.java7developer.chapter13.PlayerCharacterController链接，就会进入PlayerCharacter域对象的列表页。

尽管做这个GSP页面相当快，但如果框架能帮你做岂不是更好？用Grails的脚手架功能可以迅速做出域对象CRUD页面的原型。

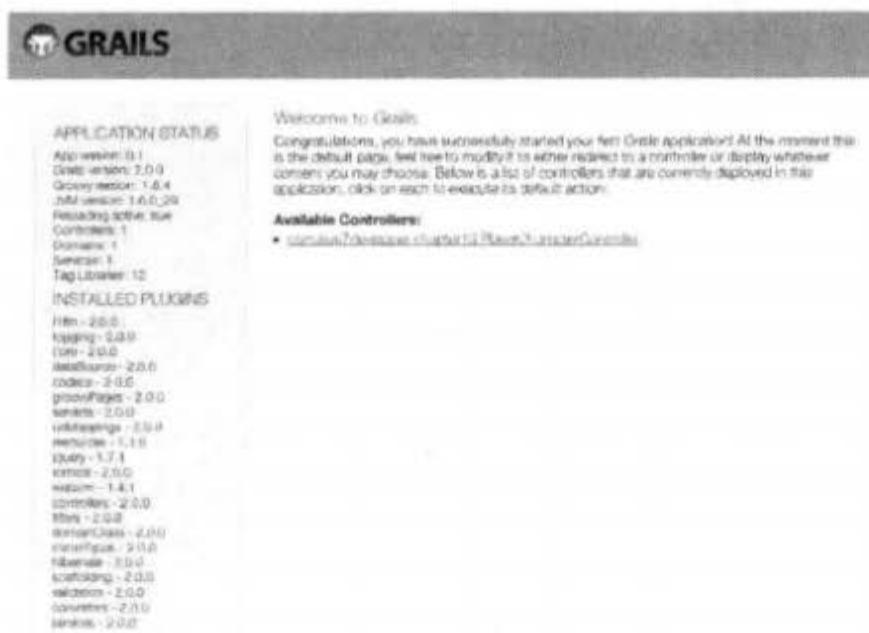


图13-4 pcgen\_grails主页

### 13.4.7 脚手架和UI的自动化创建

Grails可以用它的脚手架（scaffolding）特性自动创建用来执行域对象CRUD操作的UI。

要使用脚手架特性，请用代码清单13-7替换PlayerCharacterController.groovy源文件中的代码：

#### 代码清单13-7 带脚手架的PlayerCharacterController

```
package com.java7developer.chapter13
class PlayerCharacterController {
    def scaffold = PlayerCharacter
}
```

① 用于PlayerCharacter的脚手架

PlayerCharacterController类非常简单。依照惯例将域对象的名称赋值给脚手架变量

①，Grails马上就可以构建默认UI。

请暂时把list.gsp改成list\_original.gsp，以防它会妨碍脚手架产生相应的文件。改好之后，刷新http://localhost:8080/pcgen\_grails/playerCharacter/list页面，就会看到自动生成的PlayerCharacter域对象列表，如图13-5所示。

The screenshot shows a table titled 'PlayerCharacter List' with three columns: Strength, Dexterity, and Charisma. There are two rows of data:

Strength	Dexterity	Charisma
3	5	18
15	10	4

图13-5 PlayerCharacter实例列表

在这个页面中也可以创建、更新和删除PlayerCharacter对象。请确保添加了两个PlayerCharacter域对象记录，然后进入下一节了解与代码修改的快速周转有关的内容。

### 13.4.8 快速周转的开发

Grails的run-app命令为快速Web开发中的“快速”贡献了一点儿特殊的东西。用Grails的run-app命令运行的应用程序，其源码会和服务器连接起来。尽管这在生产环境中不是什么明智之举（因为会影响性能），但对于开发和测试来说非常重要。

**提示** 对于生产环境，一般都是用grails war创建WAR文件，然后通过标准的开发流程进行部署。

如果Grails应用中的源码改了，这些变化会自动反映到服务器上<sup>①</sup>。我们来试试，改一下PlayerCharacter域对象：在PlayerCharacter.groovy文件中加一个变量name，存一下。

```
String name = 'Gweneth the Merciless'
```

现在刷新http://localhost:8080/pcgen\_grails/playerCharacter/list页面，就能看到PlayerCharacter对象上新加了name属性这一列。注意到了吗？不用停Tomcat，不用重新编译代码，其他的什么也不用做。Grails就是靠这种几乎即时生效的速度确立了它快速Web开发框架的领导地位。

我们对快速启动项目的介绍就到此为止了，你应该体验了一把用Grails做快速Web开发。当然，还有很多可以对默认行为进行定制的方法值得探索。现在我们就去看看吧。

## 13.5 深入 Grails

可惜啊，短短一章的篇幅无法承载Grails框架的所有内容，因为它需要一本书！在这一节，我们再为新加入Grails阵营的开发人员讲一些值得探索的领域：

- 日志；
- GORM：Grails对象—关系映射；
- Grails插件。

另外，也可以到http://www.grails.org网站上去看看，上面有关于这些主题的基本教程。Glen Smith和Peter Ledbrook写的*Grails in Action* ( Manning, 2009 ) 也值得仔细阅读。

我们从Grails的日志入手吧。

### 13.5.1 日志

Grails的日志功能是由log4j提供的，在grails-app/conf/Config.groovy文件中配置。

<sup>①</sup> 对于大多数源码来说都是如此，只要没改出错来就行。

比如说，你可能想要chapter13包中的代码显示WARN消息，而域对象类PlayerCharacter只显示ERROR消息。要满足这一要求，可以把下面这段代码放到log4j的配置文件Config.groovy文件中。

```
log4j = {
    ...
    warn  'com.java7developer.chapter13'
    error 'com.java7developer.chapter13.PlayerCharacter',
           'org.codehaus.groovy.grails.web.servlet', // 控制器
    ...
}
```

日志配置就跟你过去用log4j的log4j.xml配置一样灵活。

接下来我们会看看Grails中的对象关系映射技术GORM。

### 13.5.2 GORM：对象关系映射

GORM是用Spring/Hibernate实现的，这是Java开发人员非常熟悉的技术组合。它所涵盖的功能非常广泛，但其核心功能非常像Java的JPA。

要想马上实验一下它的持久化行为，可以执行如下命令打开Grails控制台：

```
grails console
```

还记得第8章讲的Groovy控制台吗？这个Grails应用环境跟那个非常类似。

首先，我们保存一下PlayerCharacter域对象：

```
import com.java7developer.chapter13.PlayerCharacter
new PlayerCharacter(strength:18, dexterity:15, charisma:15).save()
```

PlayerCharacter保存好后有很多种办法可以读取它。最简单的办法是通过Grails添加到域对象类中的隐含id属性取回可写的完整实例。在控制台用下面这段代码换掉前面那段并执行。

```
import com.java7developer.chapter13.PlayerCharacter
def pc = PlayerCharacter.get(1)
assert 18 == pc.strength
```

要更新对象，修改一些属性然后再次调用save()方法。请再次清空控制台并运行下面这段代码。

```
import com.java7developer.chapter13.PlayerCharacter
def pc = PlayerCharacter.get(1)
pc.strength = 5
pc.save()
pc = PlayerCharacter.get(1)
assert 5 == pc.strength
```

要删除对象请用delete()方法。再次清空控制台并运行下面的代码，删除PlayerCharacter。

```
import com.java7developer.chapter13.PlayerCharacter
def pc = PlayerCharacter.get(1)
pc.delete()
```

GORM具备完整丰富的多对一、多对多关系声明能力，以及其他我们熟悉的Hibernate/JPA所支持的关系声明能力。

现在我们去看看从Rails“拿来”的插件概念。

### 13.5.3 Grails 插件

Grails有大量插件，可以帮开发人员完成常见的Web开发任务。其中最流行的插件有：

- Cloud Foundry Integration（用于将应用部署到云服务上）；
- Quartz（用于计划调度）；
- Mail（用于处理电子邮件）；
- Twitter、Facebook（用于社交网络集成）。

要查看有哪些插件可用，请执行如下命令：

```
grails list-plugins
```

然后可以执行`grails plugin-info [名称]`查看插件的更多信息，用感兴趣的插件名称替换[名称]就可以了。此外，也可以访问<http://grails.org/plugins/>深入了解这些插件及其生态系统的信  
息。

要安装插件，请运行`grails install-plugin [名称]`，用要安装的插件名称替换[名称]。比如说，为了更好地支持日期和时间，可以安装Joda-Time插件。

```
grails install-plugin joda-time
```

装上Joda-Time插件后，可以给PlayerCharacter加上LocalDate属性。把下面的import语句加到域对象类中。

```
import org.joda.time.*  
import org.joda.time.contrib.hibernate.*
```

把下面这个属性加到PlayerCharacter中。

```
LocalDate timestamp = new LocalDate()
```

为什么这跟引用JAR文件中的API不同呢？因为Joda-Time插件会确保该类型跟Grails惯例优先的原则兼容。这就是说Joda-Time的类型是映射到数据库类型上的，并且完全支持它的映射和脚手架处理。如果现在回到[http://localhost:8080/pcgen\\_grails/playerCharacter/list](http://localhost:8080/pcgen_grails/playerCharacter/list)页面中，会看到列出了日期。

借助插件的这类支持，Grails开发人员可以用很短的时间构建出数量惊人的功能。

我们对Grails的初次拜访结束了，但本章中快速Web开发的故事还没讲完。下一节会讨论Clojure的快速Web开发类库Compojure。熟悉Clojure的开发人员可以借助它用简洁的Clojure代码迅速构建出小到中型的Web应用。

## 13.6 Compojure入门

开发Web最致命的想法就是把什么网站都当成Google来设计。对于Web应用来说，过度设计和设计不足都是错误的。

务实而优秀的开发人员会考虑Web应用的上下文，不会增加任何不必要的复杂性。认真分析所有应用的非功能需求是避免构建错误的关键前提。

Compojure就是那种不妄想征服世界的Web框架。对于Web仪表板、操作监控，以及很多更加注重简单性和开发速度、而不是大规模扩展能力及其他非功能需求的简单任务来说，Compojure是非常理想的选择。从这种描述中你应该能猜出来，Compojure介于多语言编程金字塔的领域特定层和动态层之间。

在这一节我们会搭建一个简单的Hello World应用，然后讨论Compojure把Web应用串起来的简单规则。在用这些规则搭建示例程序之前，先介绍一个实用的Clojure HTML类库（Hiccup）。

如图13-6所示，Compojure构建在Ring框架之上，Ring框架是Clojure连接到Jetty Web容器的中间件。但使用Compojure/Ring并不需要对Jetty有多深入的了解。我们先用一个简单的Hello World作为Compojure的入门应用吧。

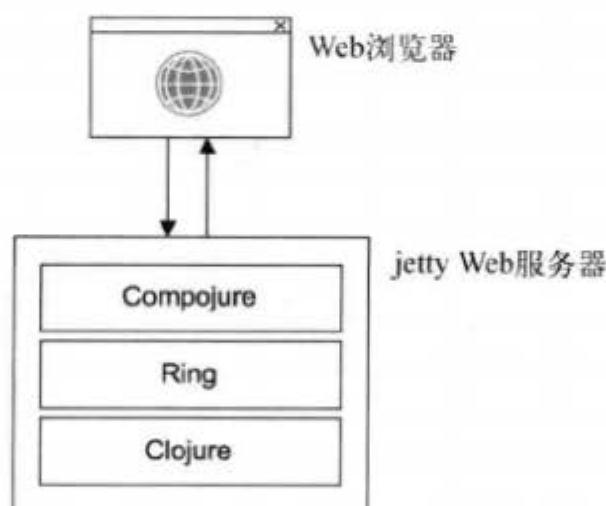


图13-6 Compojure和Ring

### 13.6.1 Hello Compojure

开始一个新的Compojure项目非常容易，因为Compojure跟Leiningen的工作流程自然融合。如果你还没装Leiningen，也没看第12章中的那一节，那你现在就应该去把这两件事做了，因为接下来的内容要求你熟悉Leiningen。

要开始一个新项目，只要执行一个普通的Leiningen命令：

```
lein new hello-compojure
```

在project.clj中可以轻松指明项目的依赖项。代码清单13-8显示了如何在project.clj文件中指定Hello World项目的依赖项。

**代码清单13-8 简单的Compojure project.clj**

```
(defproject hello-compojure "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]
                 [compojure "0.6.2"]]
  :dev-dependencies [[lein-ring "0.4.0"]])
:ring {:handler hello-compojure.core/app})
```

宏(defproject)跟第12章那个很像，不过多了两个元数据。

□ :dev-dependencies确保开发人员可以在开发时使用lein命令。稍后我们讨论lein ring server时你就能见到实例了。

□ :ring引入了Ring类库所需的挂钩。它将Ring特定的元数据映射为参数。

这个例子中给Ring传入了一个:handler属性。看起来它希望得到hello-compojure.core命名空间中的app符号。我们来看看代码清单13-9中core.clj中对它的声明，以便找出它们是如何相互配合的。

**代码清单13-9 Compojure Hello World中简单的core.clj文件**

```
(ns hello-compojure.core
  (:use compojure.core)
  (:require [compojure.route :as route]
            [compojure.handler :as handler]))

(load "hello")

(defroutes main-routes
  (GET "/" [] (page-hello-compojure))           | 主路由定义
  (route/resources "/")
  (route/not-found "Page not found"))

(def app (handler/site main-routes))             | 注册路由
```

这种把关联信息和其他信息保存在core.clj中的惯例非常实用。当有URL请求时再加载一个包含对应函数(页面函数)的单独文件很简单。这确实只是一个为了提高可读性，简单实现关注点分离的惯例。

Compojure使用了一组规则，称为路由，来确定如何处理接入的HTTP请求。这些规则是由Compojure依赖的Ring框架提供的，它们既简单又实用。你可能已经猜出来了，规则GET"/"告诉Web服务器如何处理对根URL的GET请求。我们下一节会对路由做更多的讨论。

为了完成这个例子的代码，还需要在src/hello\_compojure目录中创建hello.clj文件。在这个文件中要定义一个如下所示的页面函数(page-hello-compojure)：

```
(ns hello-compojure.core)

(defn page-hello-compojure []
  "<h1>Hello Compojure</h1>")
```

这个页面函数是个常规的Clojure函数，它会返回一个字符串作为HTML文档的<body>标签中的内容，而这个文档会作为响应的一部分返回给用户。

让我们把这个例子跑起来。在Compojure中这是个十分简单的操作。先确保所有依赖项都装上了：

```
ariel:hello-compojure boxcat$ lein deps
Downloading: org/clojure/clojure/1.2.1/clojure-1.2.1.pom from central
Downloading: org/clojure/clojure/1.2.1/clojure-1.2.1.jar from central
Copying 9 files to /Users/boxcat/projects/hello-compojure/lib
Copying 17 files to /Users/boxcat/projects/hello-compojure/lib/dev
```

到目前为止一切都好。现在需要把它跑起来，可以用Ring提供的`ring server`方法。

```
ariel:hello-compojure boxcat$ lein ring server
2011-04-11 18:02:48.596:INFO::Logging to STDERR via org.mortbay.log.StdErrLog
2011-04-11 18:02:48.615:INFO::jetty-6.1.26
2011-04-11 18:02:48.743:INFO::Started SocketConnector@0.0.0.0:3000
Started server on port 3000
```

这会启动一个简单的Ring/Jetty Web服务器（默认端口3000），以实现快速反馈。默认情况下，这个服务器会自动重载被修改的文件。

**警告** 需要知道开发服务器的重载是在文件这一层实现的。这意味着正在运行的服务器可能会因为重新加载页面导致其状态被冲掉（或更糟，被部分冲掉）。如果你怀疑发生了这种情况，并因此出现了问题，应该关掉服务器重新启动。启动Ring/Jetty很快，应该不会对开发时间有太大影响。

如果用浏览器访问开发机上的3000端口（或本机`http://127.0.0.1:3000`），应该会看到页面中显示出了“Hello Compojure”。

### 13.6.2 Ring 和路由

我们来看看如何配置Compojure应用的路由。路由的定义应该能让你想到一种领域特定语言：

```
(GET "/" [] (page-hello-compojure))
```

这些路由规则应当被看做匹配接入请求的规则。其构成方式非常简单：

```
(<HTTP method> <URL> <params> <action>)
```

- HTTP方法，通常是GET或POST，但Compojure也支持PUT、DELETE和HEAD。如果要匹配这条规则，这个HTTP方法必须跟传入的请求相匹配。
- URL，请求对应的URL。如果要匹配这条规则，这个URL必须跟传入的请求相匹配。
- 参数，一个表示参数应该如何处理的表达式。很快我们就会对它展开讨论。
- 动作，与这条规则匹配时返回的表达式（通常表示为传入参数的函数调用）。

对这些规则的匹配按从上到下的顺序逐一比对，直到找到匹配项。Compojure会执行第一个匹配项的动作，表达式的值会作为返回文档`<body>`标签中的内容。

Compojure中规则的定义很灵活。比如说，创建一个从URL中提取函数参数的规则非常简单。我们来改一下代码清单13-5中的Hello World路由：

```
(defroutes main-routes
  (GET "/" [] (page-hello-compojure))
  (GET ["/hello/:fname", :fname #"[a-zA-Z]+" ]
  => [fname] (page-hello-with-name fname))
  (route/resources "/")
  (route/not-found "Page not found"))
```

这个新规则只匹配包含/hello/<名称>的URL。其中的名称只能包含字母（大写、小写或大小写组合都行），这是由Clojure的正则表达式#"[a-zA-Z]+"限定的。

如果匹配了这一规则，Compojure会以匹配的名称为参数调用(page-hello-with-name)。函数定义非常简单：

```
(defn page-hello-with-name [fname]
  (str "<h1>Hello from Compojure " fname "</h1>"))
```

只有非常简单的应用才能用这种内联HTML，否则很快就会变成一种痛。好在有Hiccup模块，它为需要输出HTML的Web应用提供了很多实用的功能。马上我们就去看看。

### 13.6.3 Hiccup

要在hello-compojure应用中挂上Hiccup，需要做三件事：

- 在project.clj上加上依赖项，如[hiccup "0.3.4"]；
- 再次运行lein deps；
- 重启Web容器。

很好。现在我们来看看在Clojure内部怎么用Hiccup写出更好的HTML形式。

Hiccup提供的关键形式之一是(html)。用它可以非常直接地编写HTML。下面是用Hiccup重写的(page-hello-with-name)：

```
(defn page-hello-html-name [fname]
  (html [:h1 "Hello from Compojure " fname]
    [:div [:p "Paragraph text"]]))
```

现在这些嵌套格式的HTML标签看起来很像Clojure代码，所以把它放到代码里自然多了。(html)形式以一个或更多的（标签）向量为参数，并且标签的嵌套深度不受限制。

接下来，我们会向你介绍一个稍微大一点儿的应用，一个给水獭投票的网站。

## 13.7 我是不是一只水獭

互联网上似乎有两件事永远都不会让人厌烦：在线投票和可爱的动物图片。有个创业公司想把这两件事结合起来，让人们给水獭图片投票，然后靠广告回报赚钱，他们雇了你。勇敢面对吧，这毕竟还算不上是创业公司所尝试过的最傻的主意。

我们先想想这个水獭投票网站所需的基本页面和功能：

- 网站首页应该展示两张水獭供用户选择；
- 用户应该能给自己喜欢的那只水獭投票；
- 应该有个单独的页面允许用户上传水獭的新照片；

- 应该有个仪表板页面显示每张水獭图片的当前得票。

图13-7中展示了如何安排构成应用的页面和HTTP请求。

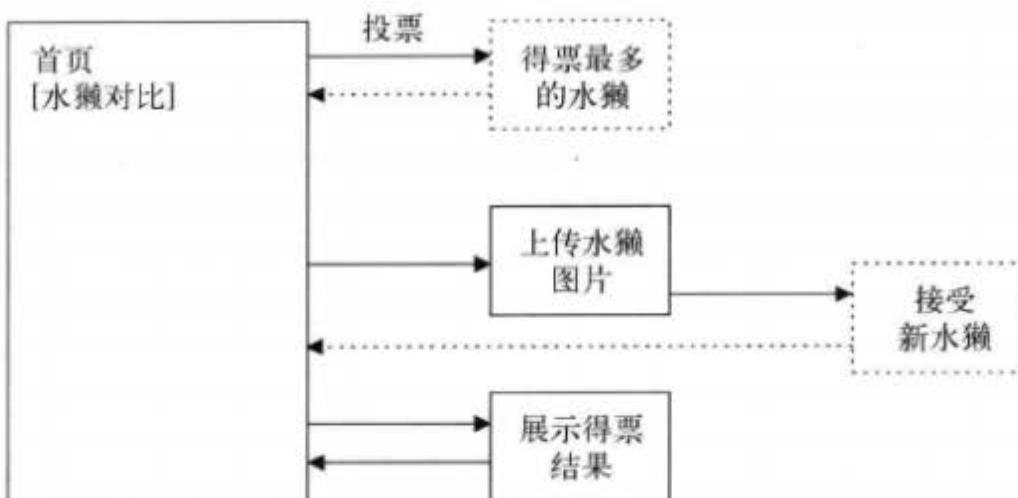


图13-7 “我是不是一只水獭？” 的页面流

我们暂不考虑该应用的非功能性需求。

- 该网站不做访问控制。
- 对新上传的水獭图片文件不做安全检查。它们会以图片的形式在页面上显示，但上传对象的内容或安全性都没有经过检查。我们相信用户，他们不会上传任何不合适的东西。
- 该网站没有持久化。如果Web容器崩溃了，所有投票数据就都没了。但在应用启动时，它会扫描硬盘，预先填充水獭图片的存储。

尽管我们会在这一章中介绍其中的重要文件，但github.com上有这个项目，你可能会发现那个更好用。

### 13.7.1 项目设置

要开始这个Compojure项目，需要定义基本项目：它的依赖项、路由，还有一些页面函数。我们先来看看project.clj文件，如代码清单13-10所示。

#### 代码清单13-10 项目project.clj

```

(defproject am-i-an-otter "1.0.0-SNAPSHOT"
  :description "Am I an Otter or Not?"
  :dependencies [[org.clojure/clojure "1.2.0"]
                 [org.clojure/clojure-contrib "1.2.0"]
                 [compojure "0.6.2"]
                 [hiccup "0.3.4"]
                 [log4j "1.2.15" :exclusions [javax.mail/mail
                                              javax.jms/jms
                                              com.sun.jdmk/jmxtools
                                              com.sun.jmx/jmxri]]]
  [org.slf4j/slf4j-api "1.5.6"]
  [org.slf4j/slf4j-log4j12 "1.5.6"]]
  :dev-dependencies [[lein-ring "0.4.0"]])
:ring {:handler am-i-an-otter.core/app})
  
```

这个文件中没什么新鲜玩意，除了log4j类库，其他在前面的例子里都有。

接下来我们看看core.clj文件里的连接和路由逻辑，如代码清单13-11所示。

### 代码清单13-11 core.clj的路由

```
(ns am-i-an-otter.core
  (:use compojure.core)
  (:require [compojure.route :as route]
            [compojure.handler :as handler]
            [ring.middleware.multipart-params :as mp]))

(load "imports")
(load "otters-db")
(load "otters")

(defroutes main-routes
  (GET "/" [] (page-compare-otters))
  (GET ["/upvote/:id", :id #"[0-9]+" ] [id] (page-upvote-otter id))
  (GET "/upload" [] (page-start-upload-otter))
  (GET "/votes" [] (page-otter-votes))

  (mp/wrap-multipart-params
    (POST "/add_otter" req (str (upload-otter req)
      (page-start-upload-otter)))))

  (route/resources "/"))
  (route/not-found "Page not found"))

(def app
  (handler/site main-routes))
```

文件上传处理程序展示了一种新的参数处理方式。我们在下一小节还会展开来讲，但现在，可以把它看做“将整个HTTP请求传给页面函数处理”。

core.clj中的关联关系让你可以看清哪个页面函数跟哪个URL相关。所有页面函数都以page打头——这只是函数命名的惯例。

代码清单13-12给出了该应用程序的页面函数。

### 代码清单13-12 项目的页面函数

```
(ns am-i-an-otter.core
  (:use compojure.core)
  (:use hiccup.core))

(defn page-compare-otters []
  (let [otter1 (random-otter), otter2 (random-otter)]
    (.info (get-logger) (str "Otter1 = " otter1 " ; Otter2 = "
    " " otter2 " ; " otter-pics)))
    (html [:h1 "Otters say 'Hello Compojure!'"]
      [:p [:a {:href (str "/upvote/" otter1)}
        [:img {:src (str "/img/"
        (get otter-pics otter1))} ]]]
      [:p [:a {:href (str "/upvote/" otter2)}
        [:img {:src (str "/img/"
        (get otter-pics otter2))} ]]]))
```

```

[:p "Click " [:a {:href "/votes"} "here"]
   " to see the votes for each otter"]
[:p "Click " [:a {:href "/upload"} "here"]
   " to upload a brand new otter")))

(defn page-upvote-otter [id]                                ← 处理投票
  (let [my-id id]
    (upvote-otter id)
    (str (html [:h1 "Upvoted otter id=" my-id]) (page-compare-otters)))))

(defn page-start-upload-otter []                         ← 选择水獭上传
  (html [:h1 "Upload a new otter"]
    [:p [:form {:action "/add_otter" :method "POST"
      :enctype "multipart/form-data"}]
      [:input {:name "file" :type "file" :size "20"}]
      [:input {:name "submit" :type "submit" :value "submit"}]])
    [:p "Or click " [:a {:href "/"} "here"] " to vote on some otters"]))

(defn page-otter-votes []                                     ← 显示投票结果
  (let []
    (.debug (get-logger) (str "Otters: " @otter-votes-r))
    (html [:h1 "Otter Votes"]
      [:div#votes.otter-votes
        (for [x (keys @otter-votes-r)]
          [:p [:img {:src (str "/img/" (get otter-pics x))}]]
        (get @otter-votes-r x))]])))

```

代码中还有两个Hiccup特性。第一个可以对一组元素进行循环，在这儿是刚上传的水獭图片。Hiccup在下面的代码片段中表现得非常像简单的模板语言（带有嵌入的(for)形态）：

```

[:div#votes.otter-votes
  (for [x (keys @otter-votes-r)]
    [:p [:img {:src (str "/img/" (get otter-pics x))}]]
  (get @otter-votes-r x))])

```

第二个特性是:div#votes.otter-votes语法。这是指明某一标签的id和class属性的快捷办法。它会变成HTML标签<div class="otter-votes" id="votes">。开发人员可以借此把最可能由CSS使用的属性分离出来，不会让HTML结构变得太乱。

CSS和其他代码（比如JavaScript源文件）通常会放在静态内容目录中等待读取。在Compojure项目中默认是在resources/public目录下。

### HTTP方法的选择

水獭投票这个例子在架构上有缺陷。我们为投票页面指定的路由规则是GET规则。这是错误的。

应用程序绝不应该用GET请求修改服务器端的状态（比如水獭的投票数）。因为Web浏览器在觉得服务器没有响应时是可以重发GET请求的（比如当请求进来时它正因为垃圾收集而暂停呢）。这一重发请求的行为可能会导致同一水獭收到重复投票，可实际上用户只点了一次。对于电子商务应用来说，这会引发灾难！

记住这条原则：有意义的服务器端状态绝不能用GET请求修改。

我们已经看过了关联起来的应用和它的路由，以及页面函数。我们再来看一些处理水獭投票的后台函数，继续讨论这个应用。

### 13.7.2 核心函数

在讨论应用的核心功能时，我们提到应用应该扫描图片目录找出磁盘里已有的水獭图片。代码清单13-13是扫描目录并进行预填充的代码。

**代码清单13-13 目录扫描函数**

```
(def otter-img-dir "resources/public/img/")
(def otter-img-dir-fq
  (str (.getAbsolutePath (File. "."))
        "/"
        otter-img-dir))
(defn make-matcher [pattern]
  (.getPathMatcher (FileSystems/getDefault) (str "glob:" pattern)))

(defn file-find [file matcher]
  (let [fname (.getName file (- (.getNameCount file) 1))]
    (if (and (not (nil? fname)) (.matches matcher fname))
        (.toString fname)
        nil)))
  ← 如果匹配，返回去掉
  ← 两边空格的文件名
  ← 用(toString)启用标签

(defn next-map-id [map-with-id]
  (+ 1 (nth (max (let [map-ids (keys map-with-id)]
                    (if (nil? map-ids) [0] map-ids))) 0)))
  ← 取下一个
  ← 水獭的ID

(defn alter-file-map [file-map fname]
  (assoc file-map (next-map-id file-map) fname))
  ← 修改函数并将文件名
  ← 加到映射中

(defn make-scanner [pattern file-map-r]
  (let [matcher (make-matcher pattern)]
    (proxy [SimpleFileVisitor] []
      (visitFile [file attrs]
        (let [my-file file,
              my-attrs attrs,
              file-name (file-find my-file matcher)]
          (.debug (get-logger) (str "Return from file-find " file-name))
          (if (not (nil? file-name))
              (dosync (alter file-map-r alter-file-map file-name) file-map-r)
              nil)
          (.debug (get-logger)
        ← (str "After return from file-find " @file-map-r)
        ← FileVisitResult/CONTINUE))

        (visitFileFailed [file exc] (let [my-file file my-ex exc]
          (.info (get-logger)
            (str "Failed to access file " my-file " ; Exception: " my-ex))
          FileVisitResult/CONTINUE)))))

      (visitFileFailed [file exc] (let [my-file file my-ex exc]
        (.info (get-logger)
          (str "Failed to access file " my-file " ; Exception: " my-ex))
        FileVisitResult/CONTINUE)))))

(defn scan-for-otters [file-map-r]
  (let [my-map-r file-map-r]
    (Files/walkFileTree (Paths/get otter-img-dir-fq
    ← (into-array String [])) (make-scanner "*.jpg" my-map-r)).
    my-map-r))

(def otter-pics (deref (scan-for-otters (ref {})))))
  ← 设置水獭图片
```

这段代码的入口是(scan-for-otters)。它用Java 7中的Files类从otter-img-dir-fq开始遍历文件系统，并返回一个映射。这里用了一个简单的惯例，以-r结束的标记名称表示这是对某个结构的引用。

遍历文件的代码是SimpleFileVisitor类（在java.nio.file包中）的Clojure代理，这个类在第2章就出现过。我们自行实现了其中两个方法：(visitFile)和(visitFileFailed)，对这个例子来说足够了。

其他有趣的函数是实现投票功能的那些，如代码清单13-14所示。

#### 代码清单13-14 水獭投票函数

```
(def otter-votes-r (ref {}))

(defn otter-exists [id] (contains? (set (keys otter-pics)) id))

(defn alter-otter-upvote [vote-map id]
  (assoc vote-map id (+ 1 (let [cur-votes (get vote-map id)]
    (if (nil? cur-votes) 0 cur-votes)))))

(defn upvote-otter [id]
  (if (otter-exists id)
    (let [my-id id]
      (.info (get-logger) (str "Upvoted Otter " my-id))
      (dosync (alter otter-votes-r alter-otter-upvote my-id)
        otter-votes-r)
      (.info (get-logger) (str "Otter " id " Not Found " otter-pics))))
    (defn random-otter [] (rand-nth (keys otter-pics)))

(defn upload-otter [req]
  (let [new-id (next-map-id otter-pics),
        new-name (str (java.util.UUID/randomUUID)
        ".jpg"),
        tmp-file (:tempfile
        (get (:multipart-params req) "file")))
    (.debug (get-logger) (str (.toString req) " ; New name = "
    new-name " ; New id = " new-id))
    (ds/copy tmp-file (ds/file-str
    (str otter-img-dir new-name)))
    (def otter-pics (assoc otter-pics new-id new-name))
    (html [:h1 "Otter Uploaded!"])))
```

在(upload-otter)函数中处理的是完整的HTTP请求映射。其中有很多信息可供Web开发人员使用，不过有些可能是你已经熟悉的了：

```
{:remote-addr "127.0.0.1",
:scheme :http,
:query-params {},
:session {},
:form-params {},
:multipart-params {"submit" "submit", "file" {:filename "otter_kids.jpg",
:size 122017, :content-type "image/jpeg", :tempfile #<File /var/tmp/
upload_646a7df3_12f5f51ff33_8000_00000000.tmp>}},
:request-method :post,
:query-string nil,
```

```

:route-params {},
:content-type "multipart/form-data; boundary=-----
  WebKitFormBoundaryvKKZehApamWrVFT0",
:cookies {},
:uri "/add_otter",
:server-name "127.0.0.1",
:params {:file {:filename "otter_kids.jpg", :size 122017, :content-type
  "image/jpeg", :tempfile #<File /var/tmp/
  upload_646a7df3_12f5f51ff33_8000_00000000.tmp>}, :submit "submit"},
:headers {"user-agent" "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_6;
  en-US) AppleWebKit/534.16 (KHTML, like Gecko) Chrome/10.0.648.205
  Safari/534.16", "origin" "http://127.0.0.1:3000", "accept-charset" "ISO-
  8859-1,utf-8;q=0.7,*;q=0.3", "accept" "application/xml,application/
  xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5", "host"
  "127.0.0.1:3000", "referer" "http://127.0.0.1:3000/upload", "content-
  type" "multipart/form-data; boundary=-----
  WebKitFormBoundaryvKKZehApamWrVFT0", "cache-control" "max-age=0",
  "accept-encoding" "gzip,deflate, sdch", "content-length" "122304",
  "accept-language" "en-US,en;q=0.8", "connection" "keep-alive"},
:content-length 122304,
:server-port 3000,
:character-encoding nil,
:body #<Input org.mortbay.jetty.HttpParser$Input@206bc833>}

```

从这个请求映射中能看到容器已经把上传的文件内容放到了 /var/tmp 的临时文件中。可以通过 (:tempfile (get (:multipart-params req) "file")) 访问相应的 File 对象。然后简单地用 clojure.contrib.duck-streams 中的 (copy) 函数把它保存到文件系统中。

水獭投票不大，但它是一个完整的应用程序。在本节开头提出的功能性和非功能性需求的限制下，它的表现符合我们的预期。我们对 Compojure 及一些相关类库的探索就到此为止了。

## 13.8 小结

快速 Web 开发应该是所有优秀 Java 开发人员都能做的事情。但如果选了糟糕的语言或框架，很快你就会落在 Rails 和 PHP 这种非 Java/JVM 技术人员的后面。尤其是静态类型的编译型 Java 语言，它有时候不是做 Web 开发的理想选择。相反，选对了语言或框架，就可以在保证质量的前提下快速实现新功能，助你攀上 Web 开发食物链的顶端，可以针对用户所需快速做出反应。

优秀的 Java 开发人员不希望扔掉强大灵活的 JVM。幸好，随着 JVM 上的其他语言及其 Web 框架的出现，你可以留着它了！像 Grails 和 Compojure 这样的动态层框架提供了你所需要的快速 Web 开发能力。

特别是 Grails，可以非常迅速地搭建一个完整的（UI 到数据库）原型，然后开发人员就可以用强大的展示层技术（GSP）、存储层技术（GORM）和一大堆实用的插件把各个部分撑起来。

Compojure可以很自然地跟Clojure编写的项目相结合。也非常适合用来向Java或其他语言的项目中添加小型Web组件，比如仪表板和操作控制台。简洁的代码和快速的开发能力是Compojure的主要优势。

我们就这样学习了JVM多语言编程的各种示例，走到了各章的结尾。在最后一章，我们会把所有的线索都抓到一起，看一些超前的知识。那里有超出我们现有经验之外的挑战，但现在我们掌握的工具已经可以处理它们了。

**本章内容**

- Java 8对开发者的意义
- 多语言编程的未来
- 并发性的发展分方向
- JVM层的新特性

要走在时代的前列，优秀的Java开发人员总是应该对即将到来的东西保持清醒的认识。本书最后一章会讨论几个在我们看来指引Java语言及平台未来发展方向的主题。

因为我们既没有TARDIS<sup>①</sup>也没有水晶球，所以本章的内容主要集中在据我们所知已经在开发的语言特性和平台修改上。也就是说这只能是当下的观点，客气的说法是这在某种程度上来说算是科幻作品。

撰写本书过程中我们所讨论的观点只是代表将来的一种可能。事情如何发展还有待时间验证。毫无疑问的是，在某些重要方式上事情的发展会跟我们此处的讨论有所不同，并且以非常有趣的方式到达那一点。通常都是这样。

让我们先去看看第一个主题吧，快速浏览一下很可能出现在Java 8中的一些主要特性。

## 14.1 对 Java 8 的期待

2010年秋，Java SE执行委员会商议决定执行B计划。这个决定是尽快发布Java 7，并把一些主要特性延迟到Java 8中。这一结论是在对社区进行广泛征询和投票后得出的。

在Java 7中发起的某些特性已经被推到Java 8中了，还有些特性已经缩减了范围以便为将来的特性打下基础。在这一节中，我们会对一些期望Java 8突出的特性做简要介绍，包括那些被延迟的特性。在这一阶段，没有什么是板上钉钉的，特别是语法。所有示例代码都只是初步构想，可能跟Java 8的最终写法差别很大。欢迎来到风口浪尖！

<sup>①</sup> TARDIS是英国科幻电视剧《神秘博士》( *Doctor Who* )中的时间机器和宇宙飞船，是时间和空间相对维度 ( Time and Relative Dimension In Space ) 的缩写。——译者注

### 14.1.1 lambda表达式（闭包）

将在Java 8中构建的Java 7特性中最有代表性的是MethodHandles和invokedynamic。它们本身是非常实用的特性（在Java 7中，invokedynamic主要用在语言和框架实现上）。

在Java 8中，这些特性是将lambda表达式引入Java语言的基础。可以这样理解，lambda表达式跟前面在备选语言中讲的函数字面值类似，它们也能用来解决我们在前面重点强调的那类问题。

就Java 8的语法而言，还需要决定lambda表达式在代码中如何表示。但其基本特性已经确定下来了，所以我们先来看看基本的Java 8语法，如代码清单14-1所示。

**代码清单14-1 在Java中用lambda表达式实现Schwartzian变换**

```
public List<T> schwartz(List<T> x, Mapper<T, V> f) {
    return x.map(w -> new Pair<T, V>(w, f.map(w)))
        .sorted((l, r) -> l.hashed.compareTo(r.hashed))
        .map(l -> l.orig).into(new ArrayList<T>());
}
```

schwartz()方法看起来应该眼熟，它是在10.3节中用闭包实现的Schwartzian变换。代码清单14-1展示了Java 8中lambda表达式的下列基本语法：

- 在lambda表达式前部有个参数列表；
- 组成lambda表达式的主体代码块用括号括起来；
- 用箭头（->）来分隔参数列表和lambda表达式的主体；
- 参数列表中参数的类型是可推断的。

第9章中Scala的函数字面值和这个写法很像，所以这种语法应该不会让你觉得特别陌生。代码清单14-1中的lambda表达式非常短，全都只有一行。实际上，lambda表达式是可以包含多行代码的，其主体甚至可以很大。经过初步分析，那些适于改造成lambda表达式的代码改造后的长度都应该在1到5行之间。

代码清单14-1中还介绍了另外一个新特性。变量x的类型是List<T>。我们在x上调用了方法map()。map()方法接受了一个lambda表达式作为其参数。停！List接口根本就没有map()方法，并且在Java 7及之前都不存在lambda表达式。

我们来仔细看看这个问题是如何解决的。

#### 1. 扩展和默认方法

我们所面临的问题本质是：怎么向已有接口中添加方法以使其“lambda化”而又不破坏其向后兼容性？

答案来自于Java的一个新特性：扩展方法。它可以为没有提供扩展方法的接口实现提供一个可用的默认方法。

这些默认的方法实现必须在接口本身内部定义。比如跟List搭配的AbstractList，跟Map搭配的AbstractMap，跟Queue搭配的AbstractQueue。这些类是为各自的接口存放新的扩展方法默认实现的理想之所。Java内置的集合类是扩展方法和lambda化的主要应用场景，但看起来这种模型在最终用户代码中也适用。

### Java怎么实现扩展方法

扩展方法会在类加载时进行处理。当加载一个带有扩展方法的接口实现时，类加载器会检查它是否实现了自己的扩展方法。如果没有，类加载器会定位到默认方法，并把一个桥接方法插入新加载类的字节码中。这个桥接方法会用invokedynamic调用默认方法。

扩展方法无需打破向后兼容性就可以让发布后的接口得到进化。开发人员可以借此带着lambda表达式为老旧的API注入新的活力。但lambda表达式对于JVM来说是什么呢？是对象吗？如果是，它们的类型是什么？

#### 2. SAM转换

lambda表达式提供了一种紧凑的办法，可以声明少量内联代码并将其作为数据传递。也就是说lambda是一个对象，就像我们在本书第三部分中对lambda表达式和函数字面值的解释一样。具体说来，你可以把lambda表达式当做Object的一个子类，它没有参数（因此也没有状态），只有一个方法。

还有一种理解这个问题的方式：通过术语SAM（Single Abstract Method，单例抽象方法）。SAM的概念在各种Java API中都有体现，是一种常见主题。很多API中都有只声明了一个单例方法的接口。Runnable、Comparable、Callable和ActionListener之类的监听器都只声明名了一个方法，因此都算SAM类。

在刚开始用lambda表达式时，可以把它们当做语法糖——为给定接口编写匿名实现的简便写法。过一段时间后，你可以掌握更多的函数式技术，甚至可能会从Scala或Clojure中把自己喜欢的技巧引入Java代码中。学习函数式编程是个循序渐进的过程：从集合的映射、排序和过滤技术开始学起，然后慢慢向外开疆拓土。

现在让我们进入下一个大主题：模块化编程，它正在Jigsaw项目的支持下如火如荼地展开。

#### 14.1.2 模块化（拼图 Jigsaw）

处理classpath有时毫无疑问是不太理想的。围绕着JAR文件和classpath构建的生态系统有些众所周知的问题：

- JRE自身的规模就比较大；
- JAR文件提倡整体式部署模型；
- 有些繁琐且极少会用到的类仍然必须加载；
- 启动慢；
- classpath是脆弱的野兽，并且跟机器上的文件系统结合得过于紧密；
- classpath基本上是一个扁平化的命名空间；
- JAR不具有固有的版本；
- 即便逻辑上没有关联的类之间也有复杂的相互依赖关系。

要解决这些问题，需要一个新的模块系统。但要先解决架构上的问题。其中最重要的如图14-1所示。

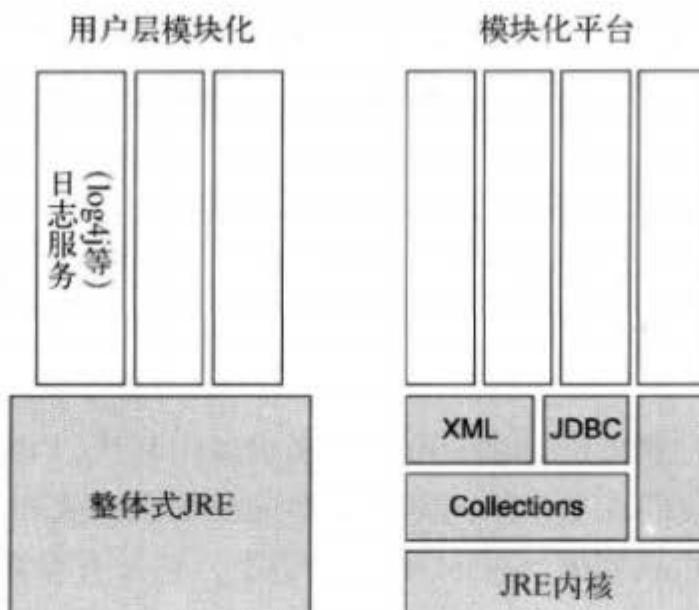


图14-1 模块化系统的架构选择

我们是应该引导VM然后再使用用户层模块化系统（比如OSGi），还是应该彻底迁移到模块化平台上？

后一种方式需要启动一个能支持模块的最基本的“内核”VM，然后根据启动应用程序的需要添加特定的模块。这要求对VM，以及JRE中很多现有的类做颠覆性的修改，但潜在收益更大。

- JVM应用程序的启动时间可以跟shell和脚本语言相媲美。
- 能显著降低应用程序部署的复杂性。
- 针对性的Java安装所占用的资源可以显著减少（对硬盘、内存和安全性都有积极影响）。  
如果你不需要CORBA或RMI，就不用装！
- 可以以更加灵活的方式升级Java安装。如果在Collections中发现了一个严重的bug，只有那个模块需要升级。

撰写本书时看起来Jigsaw项目会选择第二种方式。但在它发布并能投入使用之前，还有很长的路要走。下面是一些仍在讨论的重要问题：

- 平台或应用的正确发布单元是什么？
- 是不是需要一种跟包和JAR都不同的新结构？

这一设计决策的影响极为重要：Java无处不在，因而这种模块化的设计要渗透到所有地方。它也要支持跨OS平台。

Java平台最起码要能在Linux、Solaris、Windows、Mac OS X、BSD Unix和AIX上部署模块化应用。这些平台中有些有需要Java模块集成的包管理器（比如Debian的apt、Red Hat的rpm，以及Solaris包）。而其他平台，比如Windows，没有可供Java使用的包管理系统。

这一设计还有其他的限制。不过这一领域已经有一些成熟的项目了：比如Maven和Ivy这样的

依赖项管理系统，还有发起倡议的OSGi。新的模块化系统应该尽一切可能跟现有项目集成，即便在完全的集成和兼容被证明不可能之后，也应该提供一个顺畅的升级途径。

不管将来会怎样，Java 8的发布应该会给Java应用程序的交付和部署带来革命性的变化。

我们去看看JDK 8应该给JVM的其他公民带来的一些特性，包括我们在前面研究的那些语言。

## 14.2 多语言编程

从第5章开始，你已经无数次见证了JVM作为语言运行时平台的奇妙。第1章介绍的OpenJDK项目在Java 7的发布周期中成了Java的参考实现。非常有趣的是JVM已经发展成了一个语言无关的、真正支持多语言编程的虚拟机。

特别是随着Java 7的发布，Java语言丧失了在VM上的特权。平台上的所有语言现在都一视同仁。因此人们对添加之于备选语言非常重要、而对Java本身只有边际效益的VM特性表现出了强烈的兴趣。

这一工作是在达芬奇机（Da Vinci Machine）子项目中开展的，这一项目也叫做mlvm（多语言VM）。在这一项目中培育出的特性会被引入源码主干中。5.5节中的invokedynamic就是这样的例子，但还有很多对非Java语言非常实用的其他特性。也有需要解决的问题。

我们先来看看这些语言特性中的第一个：不同的语言运行在同一个JVM中彼此进行无障碍交流的办法。

### 14.2.1 语言的互操作性及元对象协议

语言平等是向了不起的多语言编程环境迈出的重要一步，但一些棘手的问题仍然存在。其中主要是不同的语言有不同的类型系统。Ruby的字符串是可修改的，而Java的不可修改。Scala把所有东西都当成对象，即便是在Java里作为原始类型的实体也是如此。

处理这些差异，并为同一JVM内的不同语言提供更好的交互和互操作方式，是目前尚未解决的问题，也是正在积极处理有望近期解决的问题。

想象一个将来要做的Web应用：其核心部分可能是Java代码，Web部分是用Compojure写的（即Clojure），所用的JSON处理类库是用纯粹的JavaScript写的，而你想用ScalaTest中一些很酷的TDD功能来对它进行测试。

这形成了一个JavaScript、Clojure、Scala和Java彼此之间都会直接调用的局面。对JVM语言能够互操作并以一种标准的方式调用彼此对象的需求会随着时间逐渐增强。社区内的广泛共识是需要一种元对象协议（Metaobject Protocol，MOP），以便所有这些语言都能以一种标准的方式工作。MOP可以看做是一种在代码内描述特定的语言如何实现面向对象及相关问题的办法。

要实现这一目标，我们需要想想那些能让某种语言中的对象能在另外一种语言中使用的方法。一种简单的方式是把它转换成其他语言中的本地类型（或甚至在外部运行时中创建一个新的“影子”对象）。这种办法简单，但有严重的问题：

- 所有语言必须都有一个通用的“主”接口（或超类），语言内的所有类型都必须实现它（比如JRuby中的`IRubyObject`）；
- 如果用影子对象，那会增加很多内存分配，性能也会受影响。

相反，我们可以考虑为外部运行时构建一个服务作为入口。这一服务会提供一个接口，某一运行时可以通过该接口对外部运行时中的对象执行标准操作，比如：

- 在其他语言运行时中创建一个新对象并返回对它的引用；
- 访问（获取方法或设置方法）外部对象的属性；
- 调用外部对象上的方法，并返回结果；
- 将外部对象转换成不同的相关类型；
- 访问外部对象的其他能力，对一些语言来说可能和方法调用的语义有所不同。

在这样的系统中，可以通过在外部运行时上调用`navigator`来访问外部方法或属性。调用者需要提供一种办法来标识要访问的方法：`someMethod`。通常是个字符串，但某些情况下也可能是`MethodHandle`。

```
navigator.callMethod(someObject, someMethod, param1, param2, ...);
```

要让这种办法起作用，所有协作语言运行时中的`navigator`接口必须都一样。实际上，语言之间的真实联系很可能是用`invokedynamic`建立起来的。

接下来我们去看看多语言JVM和Java 8的模块化子系统组合起来是个什么样子。

### 14.2.2 多语言模块化

随着Jigsaw和平台模块化的出现，不仅仅是Java才会从模块化中受益（并需要参与进来）。其他语言也能加入其中有所表现。

可以想象，`navigator`接口及其辅助类很可能会成为一个模块，对某一非Java语言运行时的支持将会由一个或多个模块实现。图14-2中展示了这一模块系统看起来是什么样子。

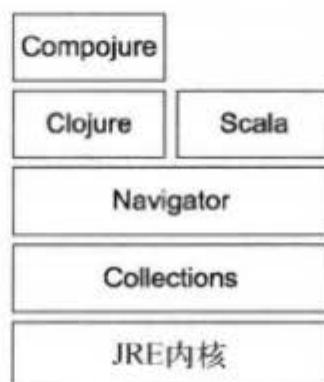


图14-2 实现了多语言解决方案的模块

如你所见，我们可以用模块系统搭建包含多种语言的应用程序。`Clojure`模块提供基本的`Clojure`平台，`Compojure`模块引入运行`webapp`所需的组件，包括特定版本的JAR，在别处运行时

这些JAR可能会用不同的版本。Scala及其XML也出现了，为了实现Scala和Clojure之间的互操作，Navigator模块也出现了。

在下一节中，我们会讨论非Java语言在平台上的爆炸性涌现所推动的另一个编程趋势：并发。

## 14.3 未来的并发趋势

20世纪的语言不一定能充分发挥21世纪的硬件的作用。我们之前讨论的内容已经多次暗示了这一点。在第6章讨论晶体管数量增长的摩尔定律时（6.3.1节），由于一个非常重要的原因，我们当时仅简要讨论了一下。那就是摩尔定律、性能和并发之间的相互作用，这也是我们的第一个主题。

### 14.3.1 多核的世界

尽管晶体管数量的爆炸性增长跟预测一样，但内存的访问速度却没能跟上。在20世纪90年代和21世纪头几年，芯片设计者用大量晶体管解决相对较慢的主存问题。

就像第6章讨论的，这可以确保有稳定的数据流供核心处理。但这根本是一场输掉的战斗：在晶体管上提升的速度变得越来越边际化。这是因为过去用的技术（比如指令级并行和投机式执行）现在已经把容易提升的速度榨光了，投机性变得越来越强。

最近这些年，业界已经把注意力转到了用晶体管在每个芯片上提供更多处理器内核。现在几乎所有笔记本或台式机都至少是双核的，4核和8核也很常见。在更高端的服务器上，可以找到6核或8核的芯片，整机能达到32（或更多）个核心。多核世界就在这里，要充分发挥它的作用，需要以串行处理更少的风格编程。那需要得到语言和运行时环境的支持。

### 14.3.2 运行时管理的并发

我们已经看到了未来并发编程的开始。在Scala和Clojure中，我们讨论了与Java的线程和锁模型有很大差异的并发观点：Scala的actor模型和Clojure的软件事务型内存方式。

Scala的actor模型允许在运行的代码块之间发送消息，而这些代码可能运行在完全不同的核心上（甚至有允许actor远程运行的扩展）。这就是说代码完全是按以actor为中心的方式编写的，因此在多核机器上扩展非常简单。

跟Scala actor一样，Clojure中的代理填补了相同的生态位<sup>①</sup>，但Clojure中还有只能在一个内存事务中修改的共享数据（refs）——软件事务型内存机制。

在这两种并发中，都能见到一种新概念的萌芽：由运行时（而不是开发人员）管理并发。尽管JVM提供了线程调度的底层服务，但它没有提供管理并发程序的高层结构。

这个缺陷在Java语言中能看出来，导致Java程序员基本上在用JVM的底层模型。

<sup>①</sup> 生态位（Ecological niche），又称小生境、生态区位、生态栖位或是生态龛位，是一个物种所处的环境以及其本身生活习性的总称。每个物种都有自己独特的生态位，区别于其他物种。——译者注

### 别对Java并发太过苛求

Java在1996年发布，它是从一开始就考虑并发的主流语言之一。经过业界15年的广泛实操，我们才对可变数据、默认共享状态和用协作锁强制排他执行这种模型的问题有所察觉。但发布Java 1.0的工程师没有这种福利。从很多方面来说，Java在并发上的首次尝试都是我们取得今天这种成就的基础。

现在有大把的代码撒在外面，再为Java做一种全新的机制来强制推行，还要跟现有代码无缝交互，这非常困难。所以大部分注意力都放在了为非Java的JVM语言找寻新的并发出路上。这些语言有两个重要特性：

- 以JMM为底层模型；
- 跟Java相比有“全新设计”的语言运行时，可以提供不同的抽象层（并且强制性更强）。

在VM层面出现更多的并发支持也不是不可能（下一节会讨论到），但目前来看主流还是在以JMM为基础的新语言上做创新，而不是修改底层的基础线程模型。

在JDK 8及以后的版本中，肯定能见到JVM的某些区域会发生变化。其中的一些变化顺延了Java 7的invokedynamic，这也是我们要讨论的下一主题。

## 14.4 JVM的新方向

我们在第1章介绍了VMSpec（JVM规范）。这一文档确切指明了作为JVM标准实现的VM必须遵守的行为准则。当引入新行为时（比如Java 7的invokedynamic），所有实现都必须升级以支持新功能。

在这一节里，我们会谈到那些已经在讨论并有原型的各种修改最终实现的可能性。这项工作是在OpenJDK项目中开展的，该项目是Java参考实现的基础，也是Oracle JDK的起点。除了对规范的可能修改，我们也会涉及对OpenJDK/Oracle JDK代码的显著改动。

### 14.4.1 VM的合并

在Oracle收购了Sun公司之后，它就拥有了两款非常强的Java虚拟机：HotSpot VM（Sun带的）和JRockit（之前收购的BEA带的）。

Oracle很快就决定不再同时维护两个VM来浪费资源，要把它们合并起来。HotSpot VM被选作基础，JRockit特性会在将来发布Java时谨慎地引进。

### 名称有什么关系？

这个合并后的VM没有官方名称，尽管VM粉和Java社区大部分都支持HotRockit这个名称。它也确实挺吸引人，但还是要看Oracle的营销部门同不同意。

所以这对咱开发人员来说，这有什么关系呢？你现在用的VM（很可能是HotSpot VM）将来会增加很多新特性，包括（但不限于）下面这些：

- 去掉PermGen，能防止一大类跟类加载有关的崩溃；
- 加强JMX代理的支持，能让你对运行的VM有更多深入的了解；
- 新的JIT编译方式，从JRockit中引入新的优化；
- 任务控制，提供有助于对生产型应用进行调优和分析的先进工具。这些工具中有些可能是需要付费的外加JVM组件，不包含在免费下载的发布包中。

### 去掉PermGen

就像6.5.2节说的，类的元数据当前保存在VM中一个的特殊内存区里（PermGen）。它很快就会被填满，特别是对于那些在运行时会创建大量类的非Java语言和框架而言。PermGen区不回收，耗光之后还会导致VM崩溃。有关人员正在开展工作，要把元数据保存在自有内存区中，让噩梦一般的“java.lang.OutOfMemory-Error: PermGen space”消息永远地成为过去。

还有很多的小改进全都是为了让VM更小、更快、更灵活。假定HotSpot上大约已经投入了1000人年的工作量，跟投入工作量更多的JRockit结合起来形成的VM前景一定更加光明。

除了合并VM，还有大量的新特性正在制作中。其中之一就是可能会增加称为协同程序的并发特性。

#### 14.4.2 协同程序

Java和JVM语言程序员了解最多的并发形式就是多线程。它依靠JVM的线程调度服务在处理器核心上启动和停止线程，但线程没办法控制这个调度。出于这一原因，多线程被称为“抢占式多任务”，因为调度器可以抢占正在运行的线程，迫使它放弃对CPU的控制。

协同程序的基本思想是允许执行单元部分参与控制对它们的调度。具体来说，协同程序会像普通线程那样运行，直到它遇到了一个“退位”指令。这会导致协同程序把自己挂起，并允许另一个协同程序继续在它的地盘运行。当原来的协同程序再次得到机会运行时，它会从退位之后的下一条语句继续向下执行，而不会从方法开始的地方。

因为这种多线程的方式靠正在运行的协同程序的相互协作，间或把运行机会退让给其他协同程序，这种多线程处理被称为“协作式多任务”。

关于协同程序如何工作的确切设计仍然处于热烈讨论的阶段，没有哪个是肯定要被采纳的。一个可能的模型是在一个单例共享线程（或类似于java.util.concurrent里的线程池）中创建和调度协同程序，如图14-3所示。

正在执行协同程序的线程可能会被系统内的其他任何线程抢占，但JVM线程调度器不能强迫协同程序退位。也就是说，以相信执行池中所有其他协同程序为代价，协同程序就可以控制什么时候切换上下文。

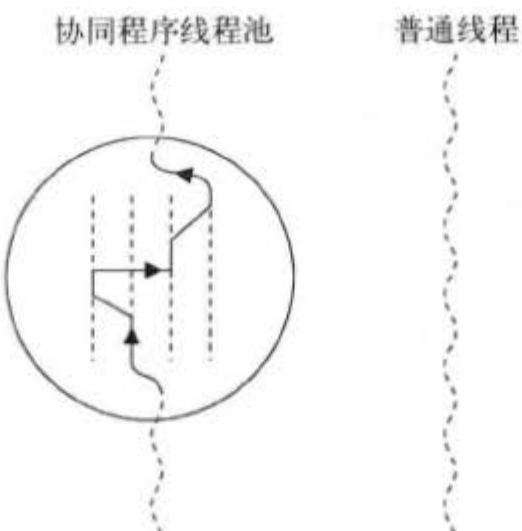


图14-3 一种可能的协程模型

这种控制意味着协程之间的同步可以做得更好。多线程代码必须构建复杂的锁策略来保护数据，但它很脆弱，因为上下文切换可能随时都会发生。这是我们在4.1节讨论的并发类型安全问题。相较而言，协程只要确保退位点数据的一致性，因为它知道其他任何时候自己都不会被抢占。

这个折中的额外担保是以相信其他线程为交换条件的，这是对某些线程编程问题的有益补充。一些非Java语言已经开始支持协程（或与之很贴近的概念“纤维”），特别是Ruby和较新版的JavaScript。在VM层面增加协程的支持（但不一定是对Java语言）会对可以使用协程的语言有很大帮助。

在可能会实现的VM修改中，最后要讨论的是“元组”，这个VM特性提案对性能敏感的计算空间可能会产生很大的影响。

#### 14.4.3 元组

在当今的JVM里，所有数据项不是原始类型就是引用类型（可能是引用对象或数组）。比较复杂的类型只能在类里定义，并传递对这些新类型实例对象的引用。这是一个简单而又相当优雅的模型，过去一直为Java服务得很好。

但要构建高性能系统，这个模型就会暴露几个缺陷。尤其是在游戏和金融软件这样的应用中，遇到这个简单模型局限性的情况十分常见。可以解决这个问题的办法之一就是采用元组。

元组（tuple）有时称为值对象，是能在原始类型和类之间架起桥梁的语言结构。像类一样，用元组可以定义包含原始类型、引用类型和其他元组的自定义复杂类型。像原始类型一样，在将它们传递给方法（或从方法中传递出来），保存在数组和其他对象中时，用的是整个值。如果你熟悉C（或.NET）环境，可以把它们看做结构（struct）的等价物。

我们来看一个例子：一个现有的Java API。

```
public class MyInputStream {
    public void write(byte[], int off, int len);
}
```

这让用户可以将特定数量的数据写到数组中的特定位置，很实用。但它设计得并不好。在理想的面向对象世界，偏移和长度应该被封装在数组内，并且无论是用户还是方法的实现者都应该不用再单独跟踪额外的信息。

实际上，在引入NIO时ByteBuffer就封装了这些信息。可惜这不是白来的，从ByteBuffer中创建新切片需要分配一个新对象，这会给垃圾收集子系统造成压力。尽管大多数垃圾收集器都非常擅长收集短命的对象，但在吞吐率非常高的延迟敏感环境中，这种分配操作会累加并最终导致应用出现令人无法接受的暂停。

如果我们能定义一个保存数组引用、偏移和长度的值对象（也就是元组）类型Slice会发生什么呢？在代码清单14-2中，我们会用新的tuple关键字来表示这个新概念。

#### 代码清单14-2 作为元组的数组切片

```
public tuple Slice {
    private int offset;
    private int length;
    private byte[] array;

    public byte get(int i) {
        return array[offset + i];
    }
}
```

这个切片的构造结合了原始类型和引用类型的很多优势：

- Slice值可以复制到方法中，也可以从方法中复制出来，就跟手工传递数组的引用和int值一样有效；
- Slice元组在退出方法后会被清理掉（因为它们跟值类型一样）；
- 对偏移和长度的处理会干净地封装在元组中。

在日常编程中有很多类型会从元组的使用中受益，比如带有分子和分母的有理数、带有实部和虚部的复数，或者由ID和领域标识引用的用户主档（献给那些MMORPG迷们）。

在处理数组时元组也能对性能有所提升。现在的数组中要放同质的数据值集合——要么是原始类型，要么是引用类型。在使用数组时，元组允许我们对内存的布局做更多的控制。

来看一个例子：一个以原始类型long为键的简单散列表。

```
public class MyHashTable {
    private Entry[] entries;
}

public class Entry {
    private long key;
    private Object value;
}
```

在当前的JVM化身中，entries数组中只能放Entry实例的引用。调用者每次查找表中的key，在用传入的值与相关Entry实例的key比较之前，必须把Entry实例解引用。

当用元组实现时，则可以在数组内展开Entry类型，因此能够省掉访问key产生的解引用开销。图14-4展示了当前情况，以及使用元组后得到的改善。

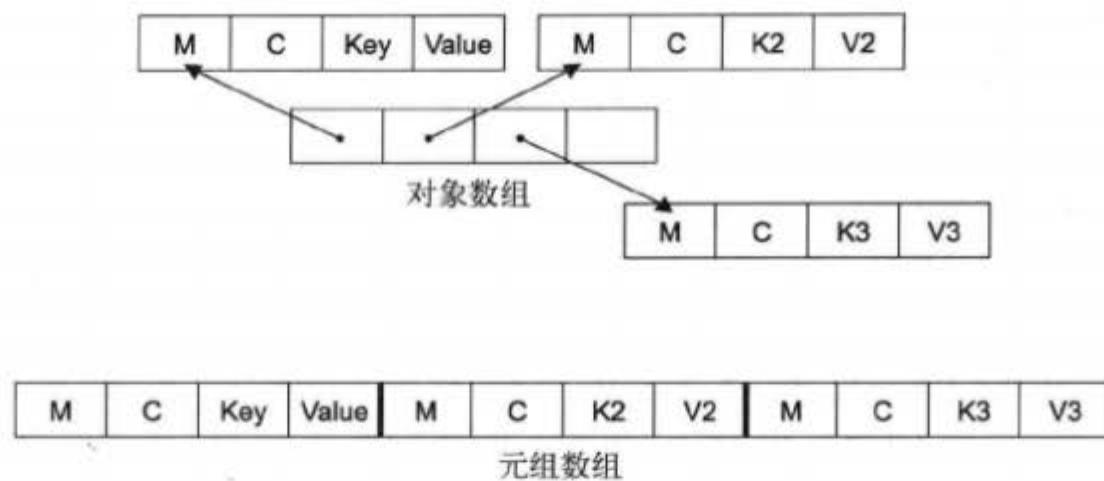


图14-4 JVM数组与元组

在考察元组的数组时，采用元组得到性能优势的关键之处也变得更加清晰。我们在第6章讨论过，大多数应用程序代码的性能都是由一级缓存的命中率决定的。在图14-4中，如果使用元组，扫描散列表的代码效率会更高。它不用再承担额外的缓存读取就能得到key值。这就是元组取得性能优势的本质——程序员在展开内存的数据时可以得到更优的空间局部性。<sup>①</sup>

对可能出现在Java和JDK 8中的新特性，我们的讨论就到此为止了。其中有多少能变成现实也只能等到快要发布时才知道。如果你对特性的演进感兴趣，可以加入OpenJDK项目和Java Community Process，参加这些特性的开发活动。如果你对它们还不熟悉，请找到这些项目并看看如何加入。

## 14.5 小结

Java 8会紧随Java 7的脚步。它会满载着各种改进而来，让开发人员可以更加高效地为现代化硬件编写代码，无论是最小的嵌入式设备还是最大的主机。

一种语言解决所有编程问题的神话已经破灭了。为了搭建有效的解决方案，比如高并发的交易系统，开发人员需要学习能跟核心Java代码互操作的新语言。

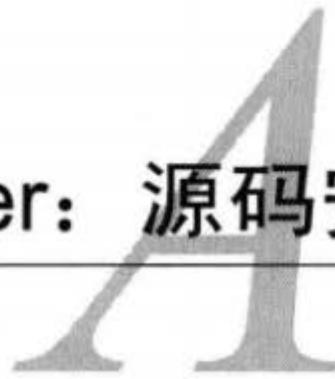
随着多核硬件和OS持续提供高度并行的编码架构，并发仍然是热门话题。想要解决海量数据、复杂运算或高速应用，这是一个必须跟进的领域。

Java VM被看做是当前最好的虚拟机。因为优秀的Java开发人员很可能会开辟高性能计算等新领域，所以有必要密切关注未来的发展态势。

如你所见，这里正在发生着巨变！我们认为Java生态系统正在经历一次规模宏大的涅槃，再过几年，优秀的Java开发人员必将光彩熠熠。

<sup>①</sup> 空间局部性 ( spatial locality )，如果程序访问某个存储器地址后，又在较短时间内访问临近的存储器地址，则程序具有良好的空间局部性。两次访问的地址越接近，空间局部性越好。——译者注

# java7developer：源码安装



在读一本新的技术书时，我们都喜欢真刀真枪地用代码做练习。它能帮我们正确理解书中的内容，光靠读代码达不到那种效果。

本书源码可在[www.manning.com/evans/](http://www.manning.com/evans/)或[www.java7developer.com/](http://www.java7developer.com/)<sup>①</sup>处下载。我们会把你放代码的位置称为\$BOOK\_CODE。

本书中所有源码都放在java7developer项目中。它混合了Java、Groovy、Scala和Clojure的源码，以及它们的支持类库和资源。它不是那种典型的Java项目，你需要按照这个附录中的指令来构建它(即编译源码和运行测试)。我们会用Maven 3来执行各种构建周期目标，比如compile和test。

先来看看java7developer项目的源码布局。

## A.1 java7developer 的源码结构

java7developer项目的结构遵守我们在第12章介绍的Maven规范，因此布局方式如下所示：

```
java7developer
|-- lib
|-- pom.xml
|-- sample_posix_build.properties
|-- sample_windows_build.properties
`-- src
    |-- main
    |   '-- groovy
    |       `-- com
    |           '-- java7developer
    |               '-- chapter8
    |   '-- java
    |       `-- com
    |           '-- java7developer
    |               '-- chapter1
    |               ...
    |               ...
    '-- resources
    '-- scala
        `-- com
```

<sup>①</sup> 也可在图灵社区 ([www.ituring.com.cn](http://www.ituring.com.cn)) 本书网页免费注册下载。——编者注

```

    |   '-- java7developer
    |   '-- chapter9
  '-- test
  '-- java
    '-- com
      '-- java7developer
        '-- chapter1
        '-- ...
        '-- ...
  '-- scala
    '-- com
      '-- java7developer
        '-- chapter9
  '-- target

```

按它的规范，Maven把主代码和测试代码分开了。它还为其他需要包含在构建中的文件设了一个特殊的resources目录(比如日志记录的log4j.xml、Hibernate配置文件以及其他类似资源)。Maven的构建脚本是pom.xml文件，附录E中有对它的详细讨论。

Scala和Groovy源码跟Java源码的目录结构一样，只是Java的根目录是java，而它们的根目录分别是scala和groovy。Java、Scala和Groovy在Maven项目中可以排排坐，和睦相处，Clojure的源码处理起来稍有不同。Clojure大多数都是通过一个交互式环境处理的(所用的构建工具也不同，叫Leiningen)，所以我们只是提供了一个clojure目录，用来存放Clojure源码，做练习的时候可以复制到Clojure REPL中。

在Maven构建运行之前不会创建target目录。构建产生的所有类、工件、报告和其他文件都会出现在这个目录下。

lib目录中放了些类库文件，以防Maven不能访问互联网下载所需类库。

看看项目结构，让自己熟悉一下各章的源码都放在哪里。一旦搞清楚源码的位置，就可以安装和配置Maven 3了。

## A.2 下载并安装 Maven

可以到<http://maven.apache.org/download.html>下载Maven。在第12章的例子中，我们用的是Maven 3.0.3。如果你用的是\*nix操作系统，请下载apache-maven-3.0.3-bin.tar.gz，如果是Windows，则下载apache-maven-3.0.3-bin.zip。文件下载完成后，只要选好目录把文件解压(untar/gunzip或unzip)就行了。

**警告** 跟很多Java/JVM相关软件的安装一样，在安装Maven的目录名称中也不要留空格，否则可能会出现PATH和CLASSPATH错误。比如说，如果你用的是Windows操作系统，不要把Maven装在C:\Program Files\Maven这样的目录中。

在下载和解压完成后，接下来就是设置M2\_HOME环境变量。在\*nix系统中，需要加一些下面这样的东西：

```
M2_HOME=/opt/apache-maven-3.0.3
```

在Windows系统中是这样的：

```
M2_HOME=C:\apache-maven-3.0.3
```

你可能在想：“为什么是M2\_HOME而不是M3\_HOME？毕竟这是Maven 3，对不对？”这是因为Maven的开发团队真的很想跟得到广泛应用的Maven 2保持兼容。

Maven需要Java JDK才能运行。1.5之后的版本都行（当然，到这一阶段，你已经装好JDK 1.7了）。还需要确保环境变量JAVA\_HOME已经设置好了——如果已经装好Java了，那这个环境变量可能已经设置好了。还需要能在命令行中的任何地方执行Maven相关的命令，所以应该在PATH中加上M2\_HOME/bin目录。在\*nix系统中，需要加一些下面这样的东西：

```
PATH=$PATH:$M2_HOME/bin
```

在Windows系统中是这样的：

```
PATH=%PATH%;%M2_HOME%\bin
```

现在可以带着-version参数执行Maven (mvn)，以确保基本安装可用。

```
mvn -version
```

应该能见到Maven输出了类似下面这种信息：

```
Apache Maven 3.0.3 (r1075438; 2011-02-28 17:31:09+0000)
Maven home: C:\apache-maven-3.0.3
Java version: 1.7.0, vendor: Oracle Corporation
Java home: C:\Java\jdk1.7.0\jre
Default locale: en_GB, platform encoding: Cp1252
OS name: "windows xp", version: "5.1", arch: "x86", family: "windows"
```

如你所见，Maven批量输出了很多实用的配置信息，这样你就知道Maven及其依赖项在你的平台上都OK了。

---

**提示** 主流IDE（Eclipse、IntelliJ和NetBeans）都支持Maven，所以熟悉了Maven在命令行中的使用方法之后，可以直接切换到IDE集成的版本。

---

现在Maven已经装好了，该去看看用户设置放在哪里了。为了触发用户设置目录的创建，需要确保Maven插件已经下载并安装好了。执行起来最简单的是帮助（Help）插件。

```
mvn help:system
```

这会下载、安装、并运行帮助插件，它给出的信息要比mvn -version还多。还会确保.m2目录已经创建好了。知道用户设置放哪里很重要，因为有那么几次你可能需要编辑用户设置，比如让Maven能用在一个代理服务器后面。home目录（我们会用\$HOME表示）中能看到表A-1中列出的目录和文件。

表A-1 Maven用户目录和文件<sup>①</sup>

题 材	解 释
\$HOME/.m2	包含Maven用户配置的隐藏目录
\$HOME/.m2/settings.xml	包含用户特定配置的文件。在这个文件中可以指定旁路代理、私有资源库以及定制Maven行为的其他信息
\$HOME/.m2/repository/	Maven的本地资源库。当Maven从Maven Central（或其他的远程Maven资源库）下载插件或依赖项时，它会在本地资源库中保存一份副本。在你用install目标安装本地依赖项时也是这样。这样Maven就可以用本地副本，而不用每次都去下载了

注意，用.m2目录还是因为要保持跟Maven 2的向后兼容（而不是你认为的.m3目录）。

现在已经装好了Maven，也知道用户配置在哪里了，可以开始构建java7developer了。

## A.3 构建 java7developer

这一节会从几个一次性步骤开始，为构建做好准备<sup>②</sup>。这包括手动安装类库、重命名属性文件并编辑它，指向Java 7的本地安装。

然后就是最常见的Maven构建周期目标（clean、compile和test）。第一个构建期目标（clean）用来清理上一次构建遗留下来的工件。

Maven的构建脚本是POM（Project Object Model，项目对象模型）文件。这些POM文件就是XML文件，每个Maven项目或模块都有一个对应的pom.xml文件。POM文件即将会对备选语言提供支持，满足你所需要的更强的灵活性（很像Gradle）。

要用Maven执行构建，可以让它执行一个或几个表示特定任务（比如编译源码、运行测试等）的目标。目标全部都是绑定到默认构建周期中的，所以如果要求Maven运行一些测试（如mvn test），它会在试图运行测试之前把主源码和用于测试的源码都编译一下。简言之，它会迫使你遵循正确的构建周期。

让我们从一个一次性的准备任务开始吧。

### A.3.1 一次性的构建准备工作

要成功运行构建，需要先重命名属性文件并编辑。如果在读12.2节时你没这么做，请转到\$BOOK\_CODE目录下，将sample\_<os>\_build.properties文件（os是你的操作系统）另存为build.properties，修改jdk.javac.fullpath属性，将其值指向Java 7的本地安装。这可以保证Maven构建Java代码时能选择正确的JDK。

准备工作做好了，可以运行clean目标了，执行构建时应该总是把它包括在内。

<sup>①</sup> 向Sonatype致敬，引自Maven: the Complete Reference在线手册（[www.sonatype.com/Request/Book/Maven-The-Complete-Reference](http://www.sonatype.com/Request/Book/Maven-The-Complete-Reference)）。

<sup>②</sup> 尽管Maven构建工具最近有改进，并且也支持多语言编程，但还是有些差距。

### A.3.2 clean

clean目标仅仅是把target目录删掉。要看实际效果，请切换到\$BOOK\_CODE目录并执行clean目标。

```
cd $BOOK_CODE
mvn clean
```

这时候，你会看到控制台中满是Maven下载各种插件和第三方类库的输出信息。Maven需要这些插件和类库运行目标，它默认从Maven Central（这些工件的主要在线资源库）下载。java7developer项目还配置了另外一个资源库，以便可以下载asm-4.0.jar文件。

**注意** Maven偶尔也会为其他目标执行这个任务，所以在执行其他目标时看到它“下载互联网”不要大惊小怪。这些东西它只会下载一次。

除了“正在下载……”的信息，应该还能在控制台中看到类似下面这种信息：

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.703s
[INFO] Finished at: Fri Jun 24 13:51:58 BST 2011
[INFO] Final Memory: 6M/16M
[INFO] -----
```

如果clean目标失败了，很可能是代理服务器阻止你访问Maven Central，使你无法下载插件和第三方类库。要解决这个问题，只需修改\$HOME/.m2/settings.xml文件，加上下面这些配置，为各种元素填上恰当的值。

```
<proxies>
  <proxy>
    <active>true</active>
    <protocol></protocol>
    <username></username>
    <password></password>
    <host></host>
    <port></port>
  </proxy>
</proxies>
```

重新运行这个目标，BUILD SUCCESS消息如期而至。

**提示** 跟其他Maven构建周期目标不同，clean不会自动调用。如果你想清除上一次构建产生的工件，必须把clean目标包括在内。

现在已经把以前构建的残留物都清除掉了，一般接下来要执行的构建周期目标是编译代码。

### A.3.3 compile

compile目标用pom.xml文件中的compiler插件配置编译在src/main/java、src/main/scala和src/main/groovy目录下的源码。这实际上是将compile-scoped的依赖项加到CLASSPATH上执行Java、Scala和Groovy编译器(javac、scalac和groovyc)。Maven也会处理src/main/resources下的资源，确保它们出现在编译时的CLASSPATH中。

编译好的类会放到target/classes目录下。要看实际效果，请执行下面的目标：

```
mvn compile
```

compile目标执行起来应该很快，控制台的输出看起来应该像下面这样。

```
...
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 119 source files to
  C:\Projects\workspace3.6\code\trunk\target\classes
[INFO] [scala:compile {execution: default}]
[INFO] Checking for multiple versions of scala
[INFO] includes = [**/*.scala,**/*.java,]
[INFO] excludes = []
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\java:-1: info: compiling
[INFO] C:\Projects\workspace3.6\code\trunk\target\generated-sources\groovy-
  stubs\main:-1: info: compiling
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\groovy:-1: info:
  compiling
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\scala:-1: info: compiling
[INFO] Compiling 143 source files to
  C:\Projects\workspace3.6\code\trunk\target\classes at 1312716331031
[INFO] prepare-compile in 0 s
[INFO] compile in 12 s
[INFO] [groovy:compile {execution: default}]
[INFO] Compiled 26 Groovy classes
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 43 seconds
[INFO] Finished at: Sun Aug 07 12:25:44 BST 2011
[INFO] Final Memory: 33M/79M
[INFO] -----
```

在这一阶段，在src/test/java、src/test/scala和src/test/groovy目录下的测试类还没有编译。尽管针对它们有专门的test-compile目标，但更典型的方式是让Maven运行test目标。

### A.3.4 test

运行test目标能看到Maven的构建周期的真实效果。在要求Maven测试时，它知道自己需要把之前的构建周期目标全都执行过之后才能成功运行test目标(包括compile、test-compile，还有很多其他的)。

Maven会通过Surefire插件，用pom.xml中配置为test-scoped依赖项的测试提供者(此处为

JUnit) 运行测试。Maven不仅运行测试，还会生成报告文件，供以后进行分析，调研失败测试并收集测试指标。

要看实际效果，执行如下目标：

```
mvn clean test
```

Maven一旦完成测试类的编译和运行，就应该能看到类似下面这种输出的报告。

```
...
Running com.java7developer.chapter11.listing_11_3.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_4.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_5.TicketTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec
Results :

Tests run: 20, Failures: 0, Errors: 0, Skipped: 0
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 16 seconds
[INFO] Finished at: Wed Jul 06 13:50:07 BST 2011
[INFO] Final Memory: 24M/58M
[INFO] -----
```

测试结果保存在target/surefire-reports里。你现在可以去看看这个文本文件，能看到测试成功通过了。

## A.4 小结

如果能一边看书一边运行书中的源码示例，你会对书中的内容有更深刻的认识。如果你喜欢冒险，还可以改一改我们的代码，甚至加一些新代码，然后以相同的方式编译和测试。

像Maven 3这样的构建工具，其底层实现复杂得超乎想象。如果你想深入了解这一主题，请阅读第12章，它讨论了构建和持续集成方面的内容。

## 附录 B

# glob模式语法及示例



Java 7 NIO.2类库在循环遍历目录和其他类似任务中用glob模式执行过滤操作，参见第2章。

## B.1 glob 模式语法

glob模式比正则表达式简单，其基本规则如表B-1所示。

表B-1 glob模式语法

语 法	描 述
*	匹配0或更多个字符
**	跨越目录匹配0或更多个字符
?	完全匹配单个字符
{}	限定一个子模式集合，进行匹配时各模式之间有隐含的OR关系，比如匹配模式A、B或C等
[]	匹配一组字符中的单个字符，或者如果字符间有连字符（-），则匹配其所限定范围的字符
\	转义符，在匹配*、?或\之类的特殊字符时使用

要进一步了解glob模式语法，请参见Oracle的在线Java教程（<http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob>）及FileSystem类的Java文档。

## B.2 glob 模式示例

一些使用glob模式的基本例子有时被称为globbing，如表B-2所示。

表B-2 glob模式示例

语 法	描 述
*.java	匹配所有以.java结尾的字符串，比如Listing_2_1.java
??	匹配任意两个字符，比如ab或x1
[0-9]	匹配0到9之间的任意数字
{groovy, scala}.*	匹配所有以groovy.或scala.开头的字符串，比如scala.txt或groovy.pdf
[]a-z, A-Z	匹配一个大写或小写的英文字符
\\	匹配\字符
/usr/home/**	匹配所有以/usr/home/开头的字符串，比如/usr/home/karianna或/usr/home/karianna/docs

要查看更多glob模式匹配的例子，请参见Oracle的在线Java教程及FileSystem类的Java文档。

---

**警告** Java 7规范定义了自己的glob语义（而不是采用已有的标准）。有些可能会变成给程序员挖的坑，特别是在Unix上。比如说，同样是`rm *`，在Java 7中会移除以点(.)开头的文件，而在Unix的`rm/glob`中则不会移除这样的文件。<sup>①</sup>

---

---

<sup>①</sup> 在Unix的glob模式中，如果文件名以“.”开头，则这个字符必须显式匹配。因此`rm *`不会移除`.profile`，并且`tar c *`也不会归档所有文件，用`tar c .`会更好。——译者注



本附录涵盖了三种JVM语言（Groovy、Scala和Clojure）以及Groovy的Web框架（Grails）的下载及安装指导，它们各自分别在第8章、第9章、第10章和第13章讨论。

## C.1 Groovy

装Groovy相当简单，但如果你对设置环境变量不熟，或者刚接触某一操作系统，你应该会觉得这个指南很有帮助。

### C.1.1 下载 Groovy

请先访问<http://groovy.codehaus.org/Download>下载最新的稳定版Groovy。我们的例子用的是Groovy 1.8.6，所以推荐你下载groovy-binary-1.8.6.zip文件。然后把下载好的压缩文件解压到选定的目录中。

---

**警告** 跟很多Java/JVM相关软件的安装一样，在安装Groovy的目录名称中也不要留空格，否则可能会出现PATH和CLASSPATH错误。比如说，如果你用的是Windows操作系统，不要把Groovy装在C:\Program Files\Groovy这样的目录中。

---

剩下没几步了，接下来需要设置环境变量。

### C.1.2 安装 Groovy

完成下载和解压后，需要设置三个环境变量以有效运行Groovy。我们会看看基于POSIX的操作系统（Linux、Unix和Mac OS X）以及微软Windows。

#### 1. 基于POSIX的操作系统（Linux、Unix、Mac OS X）

在一个基于POSIX的操作系统上，在哪里设置操作系统通常取决于打开终端窗口时运行的shell。表C-1中包含了各种POSIX操作系统shell中常见的用户shell配置文件的名称及位置。

表C-1 用户shell配置文件的常见位置

Shell	文件位置
bash	~/.bashrc和/或~/.profile
Korn (ksh)	~/.kshrc和/或~/.profile
sh	~/.profile
Mac OS X	~/.bashrc和/或~/.profile和/或~/.bash_profile

用你喜欢的编辑器打开用户shell配置文件，加上三个环境变量：GROOVY\_HOME、JAVA\_HOME和PATH。

需要先设置环境变量GROOVY\_HOME。加上下面这一行，用Groovy文件的真实位置（即解压文件的位置）换掉<安装目录>：

```
GROOVY_HOME=<installation directory>
```

在下面的例子中，我们将Groovy解压到了/opt/groovy-1.8.6中：

```
GROOVY_HOME=/opt/groovy-1.8.6
```

Groovy需要Java JDK才能运行。任何大于1.5的版本都行（此时你很可能已经装上JDK 1.7了）。你还需要确保环境变量JAVA\_HOME已经设置好了。如果你已经装好了Java，这个可能也已经设置好了，如果还没有，可以添上下面这行：

```
JAVA_HOME=<path to where Java is installed>
```

在下面的例子中，我们将JAVA\_HOME设置为/opt/java/java-1.7.0：

```
JAVA_HOME=/opt/java/java-1.7.0
```

最后，要能在命令行中的任何位置执行Groovy相关命令，所以得把GROOVY\_HOME/bin加到PATH中：

```
PATH=$PATH:$GROOVY_HOME/bin
```

保存用户shell配置文件，在下次启动新shell时，这三个变量就会生效。现在为了确保基本安装可以正常工作，可以在命令行中执行带-version参数的groovy命令：

```
groovy -version
Groovy Version: 1.8.6 JVM: 1.7.0
```

在基于POSIX操作系统上安装Groovy就完成了。现在你可以回到第8章，编译并运行Groovy代码去了！

## 2. Windows

在Windows中，设置环境变量最好的方式是通过管理计算机的GUI。请按照下面这些步骤操作：

- (1) 右键点击“我的电脑”，然后点击“属性”；
- (2) 选择“高级”选项卡；
- (3) 点击“环境变量”；
- (4) 点击“新增”添加新的变量名称和值。

现在需要设置环境变量GROOVY\_HOME。加上下面这一行，用Groovy文件的真实位置（即解压文件的位置）换掉<安装目录>：

```
GROOVY_HOME=<installation directory>
```

在下面的例子中，我们将Groovy解压到了C:\languages\groovy-1.8.6中：

```
GROOVY_HOME=C:\languages\groovy-1.8.6
```

Groovy需要Java JDK才能运行。任何大于1.5的版本都行（此时你很可能已经装上JDK 1.7了）。你还需要确保环境变量JAVA\_HOME已经设置好了。如果你已经装好了Java，这个可能也已经设置好了，如果还没有，可以添上下面这行：

```
JAVA_HOME=<path to where Java is installed>
```

在下面的例子中，我们将JAVA\_HOME设置为C:\Java\jdk-1.7.0：

```
JAVA_HOME=C:\Java\jdk-1.7.0
```

要能在命令行中的任何位置执行Groovy相关命令，所以得把GROOVY\_HOME/bin加到PATH中：

```
PATH=%PATH%;%GROOVY_HOME%\bin
```

一直点击“确定”直到退出“我的电脑”的管理界面。在下次启动新命令行时，这三个变量就会生效。现在为了确保基本安装可以正常工作，可以在命令行中执行带-version参数的groovy命令：

```
groovy -version
Groovy Version: 1.8.6 JVM: 1.7.0
```

在Windows上安装Groovy就完成了。现在你可以回到第8章，编译并运行Groovy代码去了！

## C.2 Scala

Scala环境可从[www.scala-lang.org/downloads](http://www.scala-lang.org/downloads)下载。写本书时的版本是2.9.1，但在你读到这儿时可能已经有新版本发布了。Scala确实倾向于在发布新版本时引入语言的新变化，所以如果你发现某些示例在（较新的）Scala上不能用，请认真检查语言的版本，并确保你为本书装的是2.9.1。

Windows用户应该下载.zip版本的，而基于Unix系统（包括Mac和Linux）的用户应该下载.tgz版本的。解压文件，放到指定的位置。跟Groovy一样，应该避免名称中包含空格的目录。

有几种办法可以在你的机器上设置Scala。最简单的可能就是设一个SCALA\_HOME环境变量指向你安装Scala的目录。然后根据你的操作系统，按照C.1.2节的指令（安装Groovy的），把GROOVY\_HOME全换成SCALA\_HOME。

在完成环境的配置后，可以在命令行中键入scala，应该可以打开Scala交互式会话。如果没开，说明环境配置得不正确，应该重新试一次，确保SCALA\_HOME和PATH的设置是正确的。

现在应该可以运行第9章的Scala代码清单和交互式的代码片段了。

## C.3 Clojure

要下载Clojure，请访问<http://clojure.org/>找到包含最新稳定版的zip文件。我们在示例中用的是Clojure 1.2，所以如果你用的版本比较新，请注意它们可能会稍有差异。

解压刚下载的文件，并进入它刚创建好的目录。假定JAVA\_HOME已经设置好了，java也在PATH上，那么现在你应该可以像第10章那样运行简单的REPL了，像这样：

```
java -cp clojure.jar clojure.main
```

Clojure跟这个附录里的前两种新语言不太一样，要用这门语言真的只需要clojure.jar文件。不用像Groovy和Scala那样设置任何环境变量。

在学习Clojure时，最简单的可能就是用REPL。当你开始考虑用Clojure做生产环境的部署时，需要用一个恰当的构建工具（比如第12章介绍的Leiningen）来管理应用的部署，以及Clojure本身的安装（从远程Maven资源库下载这个JAR文件）。

Clojure的基本安装有些局限性，但好在有几个非常好的Clojure跟IDE的集成。如果你用的是Eclipse，我们衷心向你推荐Eclipse的Counterclockwise插件，它很实用，也非常容易设置。

开发经验稍微丰富一点非常有用，因为在简单的REPL中开发大量代码可能有点容易分散人的注意力。但对于很多应用程序（特别是你正在学的）来说，基本的REPL就足够了。

## C.4 Grails

装Grails相当简单，但如果你对设置环境变量不熟，或者刚接触某一操作系统，应该会觉得这个指南很有帮助。[www.grails.org/installation](http://www.grails.org/installation)上有完整的安装指导。

### C.4.1 下载 Grails

请先访问[www.grails.org](http://www.grails.org)下载最新稳定版Grails。我们在本书中用的版本是2.0.1。下载好后，请把压缩文件解压到选定的目录中。

---

**警告** 跟很多Java/JVM相关软件的安装一样，在安装Grails的目录名称中不要有空格，否则可能会出现PATH和CLASSPATH错误。比如说，如果你用的是Windows操作系统，不要把Grails装在C:\Program Files\Grails这样的目录中。

---

接下来需要设置环境变量。

### C.4.2 安装 Grails

在完成下载和解压后，需要设置三个环境变量以有效运行Grails。我们会看看在基于POSIX的操作系统（Linux、Unix和Mac OS X）以及微软Windows中如何设置环境变量。

### 1. 基于POSIX的操作系统（Linux、Unix、Mac OS X）

在一个基于POSIX的操作系统上，在哪里设置操作系统通常取决于打开终端窗口时运行的shell。表C-2中包含了各种POSIX操作系统shell中常见的用户shell配置文件的名称及位置。

表C-2 用户shell配置文件的常见位置

Shell	文件位置
bash	~/.bashrc和/或~/.profile
Korn (ksh)	~/.kshrc和/或~/.profile
sh	~/.profile
Mac OS X	~/.bashrc和/或~/.profile和/或~/.bash_profile

用你喜欢的编辑器打开用户shell配置文件，加上三个环境变量：GRAILS\_HOME、JAVA\_HOME和PATH。

需要先设置环境变量GRAILS\_HOME。加上下面这一行，用Grails文件的真实位置（即解压文件的位置）换掉<安装目录>

```
GRAILS_HOME=<installation directory>
```

在下面的例子中，我们将Grails解压到了/opt/grails-2.0.1中：

```
GRAILS_HOME=/opt/grails-2.0.1
```

Grails需要Java JDK才能运行。任何大于1.5的版本都行（此时你很可能已经装上JDK 1.7了）。还需要确保环境变量JAVA\_HOME已经设置好了。如果已经装好了Java，这个可能也已经设置好了，如果还没有，可以添上下面这行：

```
JAVA_HOME=<path to where Java is installed>
```

在下面的例子中，我们将JAVA\_HOME设置为/opt/java/java-1.7.0：

```
JAVA_HOME=/opt/java/java-1.7.0
```

最后，要能在命令行中的任何位置执行Grails相关命令，所以得把GRAILS\_HOME/bin加到PATH中：

```
PATH=$PATH:$GRAILS_HOME/bin
```

保存用户shell配置文件，在下次启动新shell时，这三个变量就会生效。现在为了确保基本安装可以正常工作，可以在命令行中执行带-version参数的grails命令：

```
grails -version
Grails version: 2.0.1
```

在基于POSIX操作系统上安装Grails就完成了。现在可以回到第13章开始你的第一个Grails项目了！

### 2. Windows

在Windows中，设置环境变量最好的方式是通过管理计算机的GUI。请按照下面这些步骤

操作：

- (1) 右键点击“我的电脑”，然后点击“属性”；
- (2) 选择“高级”选项卡；
- (3) 点击“环境变量”；
- (4) 点击“新增”添加新的变量名称和值。

现在需要设置环境变量GRAILS\_HOME。加上下面这一行，用Grails文件的真实位置（即解压文件的位置）换掉<安装目录>

```
GRAILS_HOME=<installation directory>
```

在下面的例子中，我们将Grails解压到了C:\languages\grails-2.0.1中：

```
GRAILS_HOME=C:\languages\grails-2.0.1
```

Grails需要Java JDK才能运行。任何大于1.5的版本都行（此时你很可能已经装上JDK 1.7了）。还需要确保环境变量JAVA\_HOME已经设置好了。如果已经装好了Java，这个可能也已经设置好了，如果还没有，可以添上下面这行：

```
JAVA_HOME=<path to where Java is installed>
```

在下面的例子中，我们将JAVA\_HOME设置为C:\Java\jdk-1.7.0：

```
JAVA_HOME=C:\Java\jdk-1.7.0
```

要能在命令行中的任何位置执行Grails相关命令，所以得把GRAILS\_HOME/bin加到PATH中：

```
PATH=%PATH%;%GRAILS_HOME%\bin
```

一直点击“确定”直到退出“我的电脑”的管理界面。在下次启动新命令行时，这三个变量就会生效。现在为了确保基本安装可以正常工作，可以在命令行中执行带-version参数的grails命令：

```
grails -version
```

```
Grails version: 2.0.1
```

在Windows上安装Grails就完成了。现在可以回到第13章开始你的第一个Grails项目了！

## 附录 D

# Jenkins的下载和安装

本附录讲解Jenkins的下载和安装，第12章要用到它。Jenkins的下载和安装很简单。如果你确实碰到了困难，它的维基页面<https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins>应该能提供帮助。

## D.1 下载 Jenkins

可以从<http://mirrors.jenkins-ci.org/>上下载Jenkins。第12章的例子中用的是Jenkins 1.424。

常见的跟操作系统无关的Jenkins安装方式是用jenkins.war包。但如果你不太清楚如何运行自己的Web服务器（如Apache Tomcat或Jetty），可以根据自己的操作系统下载独立的安装包。

---

**提示** Jenkins团队推出新版本的频率令人印象深刻，对于那些想有更稳定的CI服务器的团队来说，这可能让他们觉得内心忐忑。Jenkins团队注意到了这个问题，所以他们现在推出了一个长期支持版本（Long Term Support，LTS）。

---

接下来，你需要按几个简单的步骤来安装Jenkins。

## D.2 安装 Jenkins

不管选的是WAR文件还是独立安装包，下载完之后需要把Jenkins装上。Jenkins要有Java JDK才能运行，任何大于1.5的版本都行（此时你很可能已经装上JDK 1.7了）。我们会先介绍WAR的安装，然后是独立安装包。

### D.2.1 运行 WAR 文件

可以在命令行上运行下面的命令直接执行Jenkins WAR文件，这样安装Jenkins非常快：

```
java -jar jenkins.war
```

这种安装方法仅适用于Jenkins的快速试用，因为它很难对其他相关的Web服务器参数进行配置，所以运行起来可能不会那么顺畅。

## D.2.2 安装 WAR 文件

对于更长久的安装，需要把WAR文件部署到你使用的Web服务器上（支持Java Web应用）。对于java7developer项目而言，我们只要把jenkins.war文件黏贴到Apache Tomcat 7.0.16服务器的webapps目录下就行了。

如果你对WAR文件和能支持Java Web应用的Web服务器不熟悉，可以用独立安装包。

## D.2.3 安装独立安装包

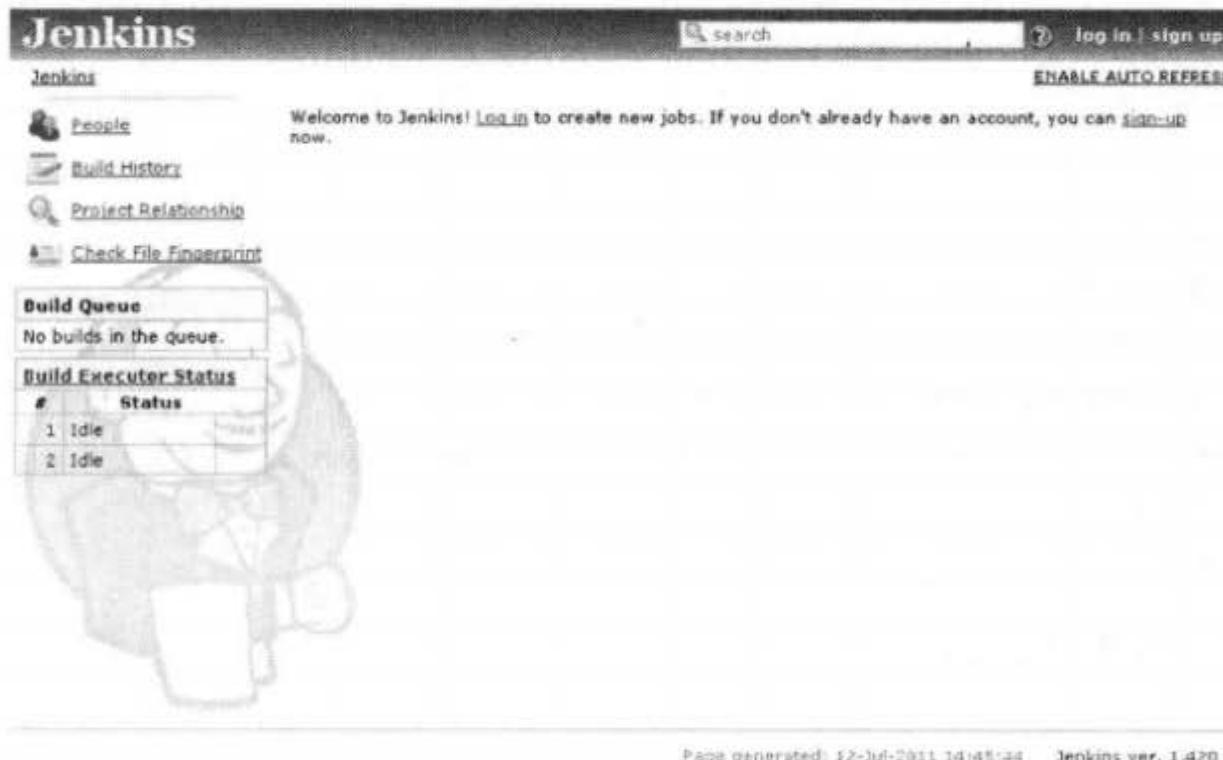
独立安装包装起来也很简单。要在Windows上安装，先解压jenkins-<version>.zip文件，然后运行exe或msi安装文件。要在各种Linux上安装，需要运行合适的yum或rpm包管理器。而对于Mac OS X而言，要运行pkg文件。

无论哪种情况，都可以选择安装文件给出的默认选项，或者按自己的想法选择安装路径或其他设置。

默认情况下，Jenkins会把配置信息和任务保存在用户的主目录（我们用\$USER表示）中的jenkins目录下。要编辑UI之外的任何配置，也可以到这个目录下。

## D.2.4 Jenkins 的首次运行

用你喜欢的浏览器访问Jenkins仪表板（地址通常是http://localhost:8080/或http://localhost:8080/jenkins），装的对不对一看就知道了。安装正确的话应该能见到跟图D-1类似的界面。



图D-1 Jenkins仪表板

装好了Jenkins，现在可以开始创建Jenkins任务了。请回到第12章关于Jenkins的章节去吧！

# java7developer: Maven POM



本附录涵盖了第12章中用来构建java7developer的pom.xml文件，在pom.xml文件的重要部分上展开，以便你能理解整个构建。基本项目信息（12.1节）和环境配置（代码清单12-4）在第12章的讨论已经很充分了，所以我们在这里会讨论POM的下面两个部分：

- 构建配置；
- 依赖项。

我们先从最长的那一部分开始，即构建配置。

## E.1 构建配置

构建部分（<build>）包含一些插件及其配置信息，需要靠它们来执行Maven构建周期目标。对于大多数项目来说，这一部分通常都相当短，因为一般用默认插件的默认设置就够了。但对于java7developer项目而言，<build>部分包含了几个覆盖了默认设置的插件。我们之所以这样做，是为了让java7developer项目可以：

- 构建Java 7代码；
- 构建Scala和Groovy代码；
- 运行Java、Scala和Groovy测试；
- 提供Checkstyle和FindBugs代码指标报告。

如果你的构建中还有更多需要配置的地方，可以在<http://maven.apache.org/plugins/index.html>找到完整的插件列表。

代码清单E-1是java 7 developer项目的构建配置。

### 代码清单E-1 POM: 构建信息

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
```

① 指明要用的插件

```

<configuration>
    <source>1.7</source>
    <target>1.7</target>
    <showDeprecation>true</showDeprecation>
    <showWarnings>true</showWarnings>
    <fork>true</fork>
    <executable>${jdk.javac.fullpath}</executable>
</configuration>
</plugin>

<plugin>
    <groupId>org.scala-tools</groupId>
    <artifactId>maven-scala-plugin</artifactId>
    <version>2.14.1</version>
    <executions>
        <execution>
            <goals>
                <goal>compile</goal>
                <goal>testCompile</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <scalaVersion>2.9.0</scalaVersion>
    </configuration>
</plugin>

<plugin>
    <groupId>org.codehaus.gmaven</groupId>
    <artifactId>gmaven-plugin</artifactId>
    <version>1.3</version>
    <dependencies>
        <dependency>
            <groupId>org.codehaus.gmaven.runtime</groupId>
            <artifactId>gmaven-runtime-1.7</artifactId>
            <version>1.3</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <configuration>
                <providerSelection>1.7</providerSelection>
            </configuration>
            <goals>
                <goal>generateStubs</goal>
                <goal>compile</goal>
                <goal>generateTestStubs</goal>
                <goal>testCompile</goal>
            </goals>
        </execution>
    </executions>
</plugin>

<plugin>
    <groupId>org.codehaus.mojo</groupId>

```

② 编译Java 7代码

③ 设置编译器选项

④ 设置javac的路径

⑤ 强制Scala编译

```

<artifactId>properties-maven-plugin</artifactId>
<version>1.0-alpha-2</version>
<executions>
  <execution>
    <phase>initialize</phase>
    <goals>
      <goal>read-project-properties</goal>
    </goals>
    <configuration>
      <files>
        <file>${basedir}/build.properties</file>
      </files>
    </configuration>
  </execution>
</executions>
</plugin>
</plugins>

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.9</version>
  <configuration>
    <excludes>
      <exclude>
        com/java7developer/chapter11/listing_11_2
        /TicketRevenueTest.java
      </exclude>
      <exclude>
        com/java7developer/chapter11/listing_11_7
        /TicketTest.java
      </exclude>
      ...
    </excludes>
  </configuration>
</plugin>

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <includeTestSourceDirectory>
      true
    </includeTestSourceDirectory>
  </configuration>
</plugin>

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>findbugs-maven-plugin</artifactId>
  <version>2.3.2</version>
  <configuration>

```

⑥ 排除测试

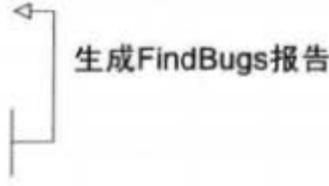
在测试上运行Checkstyle

```

<findbugsXmlOutput>true</findbugsXmlOutput>
<findbugsXmlWithMessages>
    true
</findbugsXmlWithMessages>
<xmlOutput>true</xmlOutput>
</configuration>
</plugin>

</build>

```



因为你要将编译Java 1.5代码的默认行为变为编译Java 1.7②，所以需要指明正在使用（特定版本）的Compiler（编译器）插件①。

因为已经打破了惯例，所以最好加上几个其他的编译器警告选项③。还可以指明Java 7装在哪里④。要想让Maven得到javac的位置，只要将与操作系统对应的sample\_build.properties另存为build.properties，并修改属性jdk.javac.fullpath的值即可。

为了使用Scala插件，需要确保compile和testCompile目标运行时Scala插件能够执行⑤⑥。用Surefire插件可以对测试进行配置。在这个项目的配置中，排除了几个故意失败的测试⑥（你会记起来自第11章的两个TDD测试）。

我们已经讨论过构建部分了，现在让我们转入POM中的另一个关键部分，依赖管理。

## E.2 依赖项管理

大多数Java项目的依赖项清单都很长，java7developer也不例外。Maven可以帮你管理这些依赖项，它在Maven Central Repository中存了数量庞大的第三方类库。重要的是，这些第三方类库都有它们自己的pom.xml文件，其中又声明了它们各自的依赖项，Maven由此可以推断出你还需要哪些类库并下载它们。

最初会用到两个主要作用域是compile和test<sup>⑦</sup>。这跟把JAR文件放到CLASSPATH中编译代码然后运行测试效果是完全一样的。

代码清单E-2为java7developer项目pom文件中的<dependencies>部分。

### 代码清单E-2 POM: 依赖项

```

<dependencies>

    <dependency>
        <groupId>com.google.inject</groupId>
        <artifactId>guice</artifactId>
        <version>3.0</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>javax.inject</groupId>
        <artifactId>javax.inject</artifactId>

```



① 希望这个插件的后续版本能自动挂到这些目标上。

② J2EE/JEE项目中通常还会有些声明为runtime作用域的依赖项。

```

<artifactId>javax.inject</artifactId>
<version>1</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>org.codehaus.groovy</groupId>
<artifactId>groovy-all</artifactId>
<version>1.8.6</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-core</artifactId>
<version>3.6.3.Final</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>org.ow2.asm</groupId>
<artifactId>asm</artifactId>
<version>4.0</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.8.2</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-all</artifactId>
<version>1.8.5</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.scalatest</groupId>
<artifactId>scalatest_2.9.0</artifactId>
<version>1.6.1</version>
<scope>compile</scope>
</dependency>
<dependency>
<groupId>org.hsqldb</groupId>
<artifactId>hsqldb</artifactId>
<version>2.2.4</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>javassist</groupId>
<artifactId>javassist</artifactId>
<version>3.12.1.GA</version>
<scope>test</scope>
</dependency>
</dependencies>

```

3 test作用域

4 compile作用域

为了让Maven找到我们引用的工件，需要给出正确的<groupId>、<artifactId>和<version>**①**。我们在前面说过了，将<scope>设为compile**②**会将那些JAR文件加到编译代码所用的CLASSPATH中。

将<scope>设为test**③**会确保Maven编译和运行测试时将这些JAR文件加到CLASSPATH中。scalatest类库是其中比较奇怪的，它应该放在test作用域中，但要放在compile作用域**④**才能用。<sup>①</sup>

Maven pom.xml文件并不像我们所期望的那么紧凑，但我们执行的是三种语言的构建（Java、Groovy和Scala），还能生成报告。希望随着对这一领域的工具支持不断改善，Maven构建脚本能变得更精简。

---

<sup>①</sup> 希望这个插件的后续版本能解决这个问题。

# 版 权 声 明

Original English language edition, entitled *The Well-Grounded Java Developer: Vital Techniques of Java 7 and Polyglot Programming* by Benjamin J. Evans, Martijn Verburg, published by Manning Publications. 178 South Hill Drive, Westampton, NJ 08060 USA. Copyright © 2013 by Manning Publications.

Simplified Chinese-language edition copyright © 2013 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由Manning Publications授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

“我自认为是一名Java专家：用Java写了15年程序，发表了几百篇文章，在各种会议中演讲，还执教Java高级课程。可阅读Ben和Martijn的这本大作，经常能给我一些意料之外的启发。”

——Heinz Kabutz博士

知名Java技术教育家、The Java Specialists' Newsletter创始人

“如果你想在Java专业领域占有一席之地，本书绝对值得拥有。”

——Stephen Harrison

FirstFuel软件公司首席软件架构师

“本书为那些对于编程有极大热情的Java开发人员提供了绝佳的资源。”

——亚马逊读者

“本书最棒的部分是依赖注入、多语言编程还有现代并发……老实说，这本书的所有内容都很棒！”

——亚马逊读者

今天，掌握JVM上的新语言对Java开发人员的意义非比寻常。因此本书除了深入探讨Java关键技术及Java 7最新特性，还用较大篇幅全面讨论了JVM上的多语言开发和项目控制，包括Groovy、Scala和Clojure这些优秀的新语言。这些技术可以帮助Java开发人员构建下一代商业软件。Java开发人员若要修炼进阶，本书绝对不容错过！



MANNING

图灵社区：[www.ituring.com.cn](http://www.ituring.com.cn)

新浪微博：[@图灵教育](#) [@图灵社区](#)

反馈/投稿/推荐信箱：[contact@turingbook.com](mailto:contact@turingbook.com)

热线：(010)51095186转604

分类建议 计算机/程序设计/Java

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)



The Well-Grounded  
Java Developer

Vital Techniques of Java 7  
and Polyglot Programming

# Java 程序员 修炼之道

**Benjamin J. Evans** 伦敦Java用户组发起人、Java社区过程执行委员会成员。他拥有多年Java开发经验，现在是一家面向金融企业的Java技术公司的CEO。

**Martijn Verburg** jClarity的CTO、伦敦Java用户组领导人。作为一名技术专家和众多初创企业的OSS导师，他拥有十多年的经验。Martijn经常应邀出席Java界的大型会议（JavaOne、Devoxx、OSCON、FOSDEM等）并发表演讲，人送雅号“开发魔头”，赞颂他敢于向行业现状挑战的精神。

**吴海星** 具有10多年的Java软件开发经验，熟悉Java语言规范、基于Java的Web软件开发以及性能调优，曾获SCJP及SCWCD证书。

ISBN 978-7-115-32195-4



ISBN 978-7-115-32195-4

定价：89.00元