

前言
序言
致谢
关于此书
关于封面插图

第1部分基石

第1章基础概念

- 1.1 什么是程序员测试
 - 1.1.1 对象测试的目的
 - 1.1.2 “对象测试”的节奏
 - 1.1.3 单元测试的框架
 - 1.1.4 进入JUnit
 - 1.1.5 理解测试驱动开发
- 1.2 开始使用JUnit
 - 1.2.1 下载和安装Juni
 - 1.2.2 编写一个简单的测试
 - 1.2.3 理解 TestCase 类
 - 1.2.4 失败信息
 - 1.2.5 JUnit 怎样表示一个失败的断言
 - 1.2.6 失败和错误的区别
- 1.3 一些好的实践
 - 1.3.1 测试和测试类的命名约定
 - 1.3.2 测试的是行为，而不是方法
- 1.4 总结

第2章码元测试

- 2.1 测试你的 equals 方法
- 2.2 测试一个没有返回值的方法
- 2.3 测试构造函数
- 2.4 测试获取器
- 2.5 测试设置器
- 2.6 测试接口
- 2.7 测试 JavaBean
- 2.8 测试是否抛出正确的异常
- 2.9 容器自己进行比较
- 2.10 测试一个巨型对象的相等性
- 2.11 测试一个拥有其他对象的对象

第3章组织和编译JUnit 测试

- 3.1 将测试类与产品代码放在同一个包中
- 3.2 为测试代码创建独立的源代码树
- 3.3 区分测试包和产品包

- 3. 4 抽取一个测试模块
- 3. 5 提取一个测试装置层次结构
- 3. 6 引入一个基本测试用例
- 3. 7 将对特殊用例的测试放到一个单独的测试装置里
- 3. 8 从命令行编译测试代码
- 3. 9 使用 Ant 编译测试代码
- 3. 10 使用 Eclipse 编译测试代码

第 4 章管理 Test Suites

- 4. 1]~JUnit 创建 Test Suite
- 4. 2 归纳专用的 TestCase
- 4. 3 收集一个 package 中所有的测试
- 4. 4 收集系统中的所有测试
- 4. 5 为测试扫描文件系统
- 4. 6 分离不同的 Test Suite
- 4. 7 控制某些测试的顺序
- 4. 8 创建数据驱动的 Test Suite
- 4. 9 使用 XML 定义 TestSuite

第 5 章使用测试数据进行测试

- 5. 1 使用 Java 的系统属性
- 5. 2 使用环境变量
- 5. 3 使用内联数据文件
- 5. 4 使用属性文件
- 5. 5 使用 ResourceBundle API
- 5. 6 使用基于文件的测试数据仓库
- 5. 7 使用 XML 描述测试数据
- 5. 8 使用 Ant 的任务来操作数据库
- 5. 9 使用 JUnitPP
- 5. 10 为整个 Test Suite 建立设置实体
- 5. 11 为多个测试执行单次环境设置
- 5. 12 使用 DbUnit

第 6 章运行 JUnit 测试

- 6. 1 运行时看见测试的名字
- 6. 2 在用基于文本的测试运行器时，怎样观察每个被执行的测试的名字
- 6. 3 执行单个的测试
- 6. 4 在单独的 JVM 中执行每个测试
- 6. 5 在每个测试前重新装载类
- 6. 6 略过一个测试

第 7 章汇报 JUnit 结果

- 7. 1 使用带日志功能的基础测试类
- 7. 2 使用 Log4Unit

- 7. 3 用 Ant 获取纯文本的结果
- 7. 4 使用 Ant 的任务将结果输出为 HTML 格式
- 7. 5 使用 XSLT 自定义 junit>XML 报告
- 7. 6 拓展 Ant 的 Junit 结果格式
- 7. 7 实现 TestListener 拓展 TestRunner
- 7. 8 报告断言的总数

第 8 章为 JUnit 排除疑难

- 8. 1 JUnit 无法找到你的测试
- 8. 2 JUnit 不执行你自定义的 Test Suite
- 8. 3 JUnit 没有设置你的测试实体
- 8. 4 覆盖 runTest(、)方法后测试建立失败
- 8. 5 第一个断言失败后测试停止了
- 8. 6 图形化的 Test Runner 没有正确地装载你的类
- 8. 7 当测试类使用 JAXP 的时候 JUnit 失败了
- 8. 8 当指向一个 EJB 引用时 JUnit 失败

第 2 部分测试 J2EE

第 9 章测试与 XML

- 9. 1 验证文档中元素的顺序
- 9. 2 忽略 XML 文档中元素间的顺序
- 9. 3 在 XML 文档中忽略特定种类的差异
- 9. 4 从 XMLUnit 中获得更详细的错误消息
- 9. 5 测试静态 Web 页面的内容
- 9. 6 单独测试 XSL 样式表
- 9. 7 在测试中验证 xML 文档

第 10 章测试与 JDBC

- 10. 1 测试从 ResultSet 创建 domain object
- 10. 2 验证你的 SQL 语句
- 10. 3 测试你的数据库
- 10. 4 确定测试释放了 JDBC 资源
- 10. 5 核实你的产品代码释放了 JDBC 资源
- 10. 6 在你的测试功能中管理外部数据
- 10. 7 管理测试数据库中的测试数据
- 10. 8 测试部署 schema 对象时的权限
- 10. 9 脱离数据库来测试 legacy JDBC 代码
- 10. 10 联合数据库测试遗留的 JDBC 代码
- 10. 11 联合 DbUnit 使用 schema-qualified 的表
- 10. 12 测试存储过程

第 11 章测试 EJB

- 11. 1 脱离容器测试一个 session bean 的方法
- 11. 2 测试一个遗留的 session bean

- 11. 3 在实际的容器中测试 session bean
- 11. 4 测试 CMP entity bean
- 11. 5 在容器外测试 CMP meta data
- 11. 6 测试 BMP entity bean
- 11. 7 在容器中测试 message-driven bean
- 11. 8 在容器外测试一个 message-driven bean
- 11. 9 测试遗留的 message-driven bean
- 11. 10 脱离消息服务器测试一个 JMS 消息使用者
- 11. 11 测试 JMS 消息的处理
- 11. 12 测试一个 JMS 消息生成器
- 11. 13 测试你的 JNDI 目录中的内容

第 12 章测试 web 组件

- 12. 1 脱离容器测试会话数据的更新
- 12. 2 测试 HTTP session 数据对象的更新
- 12. 3 测试解析 JSP
- 12. 4 测试对 Velocity 模板的解析
- 12. 5 测试一个 JSP tag handler
- 12. 6 测试你的 JSP 标签库的部署
- 12. 7 测试 servlet 的初始化
- 12. 8 测试 ServletContext
- 12. 9 测试对 request 的处理
- 12. 10 脱离服务器来验证网页内容
- 12. 11 验证表单属性
- 12. 12 校验传递到页面模板的数据
- 12. 13 测试 web 资源过滤器

第 13 章测试 J2EE 应用

- 13. 1 测试页面流
- 13. 2 在 Struts 应用中测试导航规则
- 13. 3 测试你的网站以寻找失效的链接
- 13. 4 测试 web 资源的安全性
- 13. 5 测试 EJB 资源的安全性
- 13. 6 测试容器管理的事务处理

第 3 部分其他 JUnit 技术

第 14 章测试设计模式

- 14. 1 测试一个 Observer(Event Listener)
- 14. 2 测试可观测的事件源
- 14. 3 测试一个 Singleton
- 14. 3 测试一个 Singleton 的客户端
- 14. 5 测试一个对象工厂
- 14. 6 测试一个 Template 方法的实现

第 15 章 GSBase

- 15. 1 用 EventCather 检查事件源
- 15. 2 测试序列化
- 15. 3 测试克隆对象
- 15. 4 用“appears equal”来比较 JavaBean

第 16 章 JUnit-addons

- 16. 1 测试你的类的 compareTo()方法
- 16. 2 从档案文件里自动收集测试
- 16. 3 用 PropertyManager 来组织测试数据
- 16. 4 管理共享的测试资源
- 16. 5 保证你的共享测试组件会把自己卸掉
- 16. 6 在执行每个测试时报告其名称

第 17 章 补遗

- 17. 1 在测试前清理文件系统
- 17. 2 不依赖文件系统的情况下测试基于文件的应用。
- 17. 3 检验你的测试用例类的语法
- 17. 4 提取定制的断言
- 17. 5 测试一个无返回值的继承方法
- 17. 6 如果你必须测试一个 private 方法

附录 A 完整方案

- A. 1 用 XML 定义一个测试
- A. 2 覆盖 runTest()的参数化测试用例
- A. 3 忽略 XML 文档中的元素次序
- A. 4 隔离测试一个 XSL 样式表
- A. 5 使你的测试中的 XML 文档生效
- A. 6 基于方面的通用 Spy
- A. 7 测试一个 BMP 实体 bean

附录 B 测试的文章

- B. 1 简单得不能拆分
- B. 2 奇特性与传递性
- B. 3 隔离高开销的测试
- B. 4 仿制对象概览

附录 C 阅读清单

参考文献

索引

如果要写 JUnit 测试代码，应该从哪里开始呢？

本书的第一部分为有效的使用 JUnit 设计和测试 Java 代码奠定了基础。一旦你能理解这部分介绍的技巧，并能熟练地在实际工作中应用它们，你就会永远使用下去——因为所有的测试，其实都可以分解为后面几章中将要介绍的一个或者几个技巧。问题在于，你如何在工程级的特定 Java 应用的代码和类结构中，认出这些简单的小模式。因此，在对付大的问题之前，让我们首先处理这些小问题。

到第 1 部分结束的时候，你将学习到 60 多个重要的 JUnit 技巧，它们涵盖了测试的各个方面：编写、组织、创建和执行测试，还有如何维护测试数据以及如何汇报测试结果。第 2、第 3 部分的技巧经常会用到第 1 部分介绍过的技巧，因此请做好随时回头查阅的准备。很快，你就会对这部分介绍的技巧非常熟悉。

本章主要内容：

- 有关程序员测试的介绍
- JUnit 入门
- JUnit 的一些经验之谈
- 什么时候使用测试而不是调试

我们痛恨调试程序。

你抬头看看墙上的钟，发现已经很晚了，因为今天晚上还有一堆的错误需要纠正。现在已经到了“编码与纠正”阶段的纠正部分了，并且这个阶段已经持续了三个月。到了这个时候，你可能几乎已经忘记你们家是什么样的了。你现在比以前任何时候都熟悉的是办公室的四面墙——假设你真的有四面墙可以看到。当你盯着“关键错误”列表的时候，发现有一个问题又回来了，你还以为一个星期前就已经解决了呢！这些该死的测试人员什么时候能够让你安静一会儿呢！

先启动调试工具，再启动应用服务器——赶紧呷一口咖啡，你可能只有 5 分钟的空闲时间——然后你需要设置一两个断点，在 10 个文本区输入一些数据，然后点击“GO”按钮。调试器会在第一个断点停下来。你的目的是找到哪个对象老是产生错误的数。当你一步步地调试代码的时候，偶尔性的肌肉痉挛——明显是睡眠不足造成的——会让你忽略那些可能产生问题的代码。现在你必须将应用服务器停下来，重新启动调试器，再重新启动应用服务器。利用这个间隙，抓起一块已经变味的糕点，喝下六个小时以前冲的苦咖啡。这样的工作有意思吗？

哦不！就算最有毅力的人也会说，“我宁愿做一个测试人员！”

1.1 What is Programmer Testing 什么是程序员测试

程序员测试并不是对程序员进行测试，而是有关程序员所进行的测试工作。近几年，一些编程人员发现如果自己去编写测试程序会带来很多好处，这种好处是以前“让测试部门来管”所没有的。修正错误非常困难，主要是因为花时间太多：不仅测试人员需要花很多时间来发现错误，并且要给程序员提供足够的信息来重现这个错误；编程人员也要花费很多时间，从几个月前看过的代码中找到错误产生的原因。还有很多时间花在讨论这些究竟是不是一个错误，弄清楚为什么程序员会犯如此幼稚的错误，是不是测试人员干扰了他们的工作？是不是需要让测试人员远离程序员而不去打扰他们等等。如果程序员自己测试他们自己的代码，那么就可以节省很多时间。

程序员所做的测试一般叫做“单元测试”。但是我们在这里不想使用这个词语，因为这个词被过度地使用和滥用，通常它带来的混乱比它所能解释的意思还要多。在程序员社区中，我们不能统一地确定“单元”是什么——是一个方法吗？是一个类吗？还是一段代码呢？如果我们不能就“单元”达成共识，那么就不用说“单元测试”了！这也是为什么我们使用“程序员测试”来表示程序员自己所做的测试工作。这也是我们为什么使用“对象测试”来表示对单独对象的测试。将不同的对象隔离开单独测试，是这本书中主要涉及到的测试方式。也许这和你所想象的测试工作有出入。

有一些程序员是这样测试他们的程序的：先在某些特定的行上设置一些断点，然后再将应用程序运行在“调试”的模式上，一行一行地步入他们的程序，并且检查相应变量的数值。严格地说，这就是“程序员测试”。因为这是程序员在测试自己的程序。但是这样做会带来很多缺点，包括：

- 需要调试工具，这并不是所有的人都已经安装或愿意安装的。
- 在执行测试以前，需要有人去设置断点；在测试完成以后再将断点去掉。如果要进行多次测试的话，这样做非常麻烦。
- 需要知道和记住这些变量所希望的数值。这就使得其他人来执行这些测试非常困难，除非他们能够知道和记下同样的数值。
- 想要测试任何的代码片段，就必须了解整个应用程序是怎样工作的，需要花一段漫长的时间来进行一系列键盘的输入和鼠标的点击。这会使一个测试的执行过程非常容易出错。

1.1.1 对象测试的目的

我们把“对象测试”定义为将对象单独隔离进行测试。正是“隔离”这个词使得“对象测试”和前面提到的手工测试有所不同。“对象测试”的思想就是单独测试每个对象，而不用考虑这个对象在周边的系统中所起的作用。如果创建的每一个对象都符合其定义说明书（或者协议），那么将它们组合在一起形成一个大系统，这个系统多半也能像你期望的那样正常工作。要编写一个“对象测试”包括编写一些

代码通过直接调用函数来操纵这个单独的对象，而不是“从系统外部”测试整个系统。那么一个“对象测试”具体是什么样的呢？

1.1.2 “对象测试”的节奏

当编写“对象测试”的时候，程序员通常是这样想的，“如果我在那个对象上调用了这个方法，它返回的应该是这样的结果。”这样的想法形成了一种节奏——一个通用的不断重复的结构，包括以下几个步骤：

1. 创建一个对象。
2. 调用一个方法。
3. 检查调用的结果。

Bill Wake, “*Refactoring Workbook*”的作者，使用了三个“A”来描述这个节奏：“安排（arrange），行动（act），断定（assert）。”请记住这三个“A”，这能够让你在使用 JUnit 来编写对象测试的时候更加有效。这种模式非常有效，因为相应的测试能够反复多次进行，都不会超出可以预见的行为范围之外：

如果这个对象处于这个状态的时候，我做了那样的操作，那么某种情况一定会发生。对象测试的挑战之一就是将整个系统的行为浓缩为那些重要的，可以预测的用例。你可以将这整本书都看成是如何找到方法从复杂的软件中抽取简单的、可以预测的测试，然后使用 JUnit 来编写这些测试。

那么，应该怎样来编写对象测试呢？

1.1.3 单元测试的框架

在一篇标题为“简单的 Smalltalk 测试：使用各种模式（Simple Smalltalk Testing: With Patterns）”的论文中，Kent Beck 描述了怎样使用 Smalltalk 来写一个对象测试。这篇论文还介绍了一个简单的测试框架的进展情况，这个框架叫做“*SUnit*”。Kent 和 Erich Gamma 一起将这个框架移植到 Java 上，被称为“*JUnit*”。自从 1999 年以来，JUnit 已经发展成业界标准的 Java 测试和设计的工具，获得了广泛的接受，不仅仅在开源的项目（www.opensource.org）中，还经常被商业软件公司所使用。

Kent Beck 的测试框架被移植到 30 多种语言和环境。在这个框架结构后面所表现的一些概念，被抽象成 xUnit，慢慢演化成一些简单的编写测试的规则。

测试必须自动化

在编程人员社区当中，通常认为测试就是输入一些文字，按一下按钮，然后观察发生的结果。

虽然这也叫做测试，但这仅仅是测试的方法之一，适用于通过最终用户的界面进行端到端的测试。这不是一种用于测试对象层次的有效的方法。手动的代码级别的测试通常包括设置断点，将代码运行在调试的模式下，然后分析各个变量的数值。这个过程是非常耗时的操作，它中断了编程人员正常的工作流程，浪费了大量本应该用于编写工作代码的时间。如果你能够让计算机去运行这些测试，将会大大提高你的效率。如果将这些测试的程序都用 Java 编写，你就可以让计算机来运行这些测试了。因为你已经是 Java 程序员了，你的代码已经是用 Java 来编写的，编写 Java 代码来调用对象中的方法是很有意义的，这样你就不用手动地去调用这些方法。

注意：“探索型测试”——通常认为自动测试和探索型测试是截然相反的两种技术，但是如果我们查看 James Bach 在他的文章“什么是探索型测试（What is Exploratory Testing）”中所给出的定义，发现其实并不是那么回事。探索型测试将重点放在决定要写什么测试，写完后，根据此测试的反馈信息再决定下一步怎么做。这跟“测试驱动开发”非常类似。“测试驱动开发”是一种编程的技术，主要是通过编写测试来帮助驱动对类的设计。“探索型测试”可能进行一些手动的测试，感知被测试的系统，将获得的知识保存起来，然后放弃这些测试。他认为通过测试获得的知识比测试本身更有价值。“测试驱动开发”中，一个“测试驱动”通过测试来保障代码改动的安全性，所以开发和保留一系列测试非常重要。除了这些不同以外，这两种方法拥有共同的特点：测试是用来感知软件的。“探索型测试”感知软件是怎样工作的，而“测试驱动”是用来感知软件应该怎样被设计。我们推荐在总体上采用“探索型测试”的方法，然后尽可能将结果自动化，以提供持续的保护防止衰退。如果想要在以前没有测试程序的代码中加入测试，你会发现“探索型测试”非常有用，可用来逆向生成你所想要的自动测试程序。

除了自动化，测试还需要可重复性。在相同的条件下多次执行相同的测试应该得到相同的结果。如果测试不能重复，你会发现你自己会花很多时间来解释，为什么今天的测试结果和昨天不一样呢？其实并没有什么错误需要修正。你需要通过测试来发现和防止错误。如果运行一个测试要花费大量的时间和精力，而且并不能很好的发现和防止错误，那为什么还需要测试呢？

测试必须自我校验

许多编程人员已经将测试自动化了。他们已经认识到执行一个稳定的、可重复的测试是多么地重要。因为程序员需要分析各种变量的值，他们经常编写一段代码来将那些重要的变量的数值打印在屏幕上，并且观察这些数值，判断它们是否正确。这个过程虽然简单而直接，却打断了程序员的正常的编程工作；更加糟糕的是，这种测试必须依赖程序员对程序的了解（和记忆力），知道正确的数值是什么。当程序员最开始在这一部分系统工作的时候，应该不会有任何问题，但是四个月以后，他可能记不得返回的值到底是 37 还是 39 了——他也不知道当前的测试应该是算通过还是失败了。要解决这个问题，测试本身必须知道它所期望的返回结果，并且告诉我们这个测试是通过还是失败了。这个很容易做到：在测试的末尾加一行代码说，“这个变量的值应该是 39：如果结果是 39 就打印‘OK’，如果不是就打印‘Oops’”。

多个测试必须能够很容易同时运行

只要有了那些可以自动运行并且自我校验的测试，你会发现经常需要运行这些测试。你会慢慢依靠这些测试提供直接的反馈，来判断这段产品代码是否如你所愿那样工作正常。你会创建大量的测试，不仅用来校验一些简单的用例，大量的边界条件，还包括那些你所关心的异常情况。你会希望连续运行所有这些测试，让它们告诉你是否有一些测试失败了。你可能一个一个地运行这些测试——你甚至会写一小段脚本来连续运行许多个测试。但是最终，你一定会集中在如何编写测试上，而不需要关心怎样执行它们。要实现这个功能，你可以将这些测试统一放到一个共同的地方，例如同一个 Java 类中，然后提供一些自动的机制从这个类中抽取这些测试，并且统一执行它们。这个“测试抽取”的引擎只需要写一次，然后可以不断地重新使用。

1.1.4 进入 JUnit

JUnit 是一个框架结构，是让你用 Java 来编写自动运行，自我校验的测试，这些测试在 JUnit 中被叫做“测试用例”。JUnit 提供一个机制能够将相关测试组合到一起，称之为“测试套件（*test suite*）”。JUnit 还提供了一个“运行器”来执行一个测试套件。如果有测试失败了，这个测试运行器就会报告出来，如果没有测试失败，就会简单地说个“OK”。当编写 JUnit 测试的时候，你将所有的相关知识都放到测试当中去，这样，这些测试就变得完全与编程人员无关了。这就意味着任何人不需要知道这些测试是做什么的，就可以运行它们，并且如果他们好奇的话，他们只需要阅读测试代码。JUnit 测试是用大家所熟悉的语言——Java——来编写的，并且容易阅读，就算是对这种测试不熟悉的人员也能读懂。

1.1.5 理解测试驱动开发

我们在这本书中的许多建议都是来自“测试驱动开发”的经验。这是一种编程的风格，这种风格大致的意思是：如果你在编写要通过测试的产品代码之前，先编写测试本身，这样会自然而然地带来以下好处：

- 你的整个系统都会被测试所覆盖
- 你可以从一些松耦合但是互相关联的对象来构建你的系统。
- 整个过程非常稳健，在通过了一个又一个的测试之后，整个系统逐步扩大和提高。
- 一个刚刚通过的测试肯定是发生在几分钟前，给你更多的自信和不断的正面的反馈信息。

除了先编写测试程序以外，你还需要在编写代码的时候执行重构；这意味着，为了达到降低添加新特性成本的目标，在你构建系统的时候，要尽量改善系统的设计。一个好的设计应该是没有重复代码，没有多余的类，自我描述得很清楚，从类名和方法名就能清楚地知道它们是做什么的系统。

这样一个系统是很容易修改，很容易扩展，并且很容易理解的。这些特点不仅能取悦于编程人员，取悦于项目经理，取悦于最终用户，更重要的是能取悦于 CEO 们。

“测试驱动开发”（Test-Driven Development，以下简称 TDD）拥有大量的并且还在快速增长的用户群体，包括本书的所有作者。尽管我们热衷于这种编程的风格，这并不是有关 TDD 的书，而是一本描述怎样有效地使用 JUnit 的书。我们极力推荐 Kent Beck 的“测试驱动开发：示例（*Test-Driven Development:By Example*）”这本书，使大家对 TDD 有更加全面的了解。我们希望本书能够成为 Beck 的著作的姊妹篇，至少为 Java 编程人员带来一些价值。

1.2 Getting started with JUnit1 开始使用 JUnit

到目前为止，我们已经认为对于测试来说，不仅仅是设置断点，然后查看变量值。我们还介绍了 JUnit，一个可重复的、自我校验的、面向对象的测试框架结构。下一步可以开始编写一些 JUnit 测试了，所以让我们看看如何下载和安装 JUnit，然后编写和执行一些测试。

1.2.1 下载和安装 JUnit

JUnit 很容易安装和使用。要想准备好运行 JUnit，你必须：

1. 下载 JUnit 包
2. 将 JUnit 包解开到文件系统中。
3. 当你构建和运行测试的时候，将 JUnit 中的所有 *.jar 文件放到你的类路径中。

下载 JUnit

目前，要想找到 JUnit 的最好的地方就是 www.junit.org。你会发现一个下载的链接链到此产品的最新版本。点击下载链接来将这个软件下载到文件系统中。

解包 JUnit

通过流行的*.zip 文件工具，你可以将 JUnit 解包到文件系统中的任何目录下。表 1.1 描述了 JUnit 发布中的一些主要文件和目录。

表 1.1 JUnit 发布中包含了什么

文件/目录	描述
junit.jar	JUnit 框架结构、扩展和测试运行器的二进制发布

src.jar	JUnit 的源代码，包括 Ant buildfile
JUnit	JUnit 自带的用 JUnit 编写测试示例程序
Javadoc	JUnit 完整的 API 文档
Doc	一些文档和文章，包括“Test Infected: Programmers Love Writing Tests”和其他的一些资料可以帮助你入门

要检验你的安装是否正确，需要执行 JUnit 的测试套件。没错：JUnit 的发布已经自带了用 JUnit 写的测试程序！要执行这些程序，请按照以下步骤进行：

1. 打开一个命令行提示。
2. 转到包含 JUnit 的目录下（D:\junit3.8.1 或/opt/junit3.8.1 或任何你决定的地方）。
3. 执行下列命令：

```
> java -classpath junit.jar;. junit.textui.TestRunner
```

```
    junit.tests.AllTests
```

你会看到类似于以下的结果：

```
.....
```

```
.....
```

```
.....
```

```
Time:2.003
```

```
OK (91 tests)
```

对于每个测试，测试运行器都会打印出一个点，让你知道现在正在执行的过程中。在执行完所有的测试之后，测试运行器会说“OK”，并且告诉你它总共执行了多少个测试和花费了多少时间。

将*.jar 文件包含在你的类路径下

看看你用来执行测试的命令：

```
>java -classpath junit.jar;. junit.textui.TestRunner junit. tests. AllTests
```

类路径包括了 junit.jar 和当前的目录。在编译和运行你的测试的时候，这个文件必须放到类路径下。这也是仅有的一个需要放到类路径下的文件。这是一个简单的过程，因为当前的目录——你解包 JUnit 的那个目录——恰恰就是 JUnit 测试的 *.class 文件所在的目录。

接着的一个参数，JUnit.textui.TestRunner，是基于文本的 JUnit 测试运行器的类名。这个类会执行所有的 JUnit 测试，并将结果报告给控制台。如果你想保存测试结果以供以后回顾，可以将此输出重新定向到文件中去。如果测试运行得不正确，参见第 8 章，来获得详细的信息。如果在执行测试的时候有问题，或者需要以特殊的方式来执行测试，可参见第 6 章。

最后一个参数，junit.tests.AllTests，是需要运行的测试套件的名字。这些 JUnit 的测试包括了一个叫做 AllTests 的类，它会构建一个包含 100 多个测试的测试套件。要想获得更多有关怎样组织测试的信息，请参见第 3 章。

1.2.2 编写一个简单的测试

现在你能够执行这些测试了。但你可能想要编写一个你自己的测试。让我们从清单 1.1 所示的例子开始吧。

清单 1.1 你的第一个测试

```
package junit.cookbook.gettingstarted.test;

import junit.cookbook.util.Money;

import junit.framework.TestCase;

public class MoneyTest extends
TestCase {

    public void testAdd() {

        Money addend = new Money(30,
0);

        Money augend = new Money(20, 0);

        Money sum = addend.add(augend);

        assertEquals(5000, sum.inCents());
    }
}
```

← 创建 TestCase 类的子类

← 每个测试都是一个方法

← 30 dollars, 0 cents

← 参数必须相等

```
}  
  
}
```

这个测试遵循了基本的“对象测试”的风格：

- 创建了一个对象，叫做“addend”。
- 调用了一个方法，叫做“add()”。
- 通过比较“inCent()”的返回值来检查结果是否和期望的数值“5 000”一样。

用简单的话，这个测试在说，“如果我将\$30.00 和\$20.00 相加，我应该得到\$50.00，恰好是 5 000 美分。”

这个例子演示了 JUnit 的不同的几个方面，包括：

- 要创建一个测试，需要编写一个方法来表示这个测试。我们将这个测试命名为“testAdd()”，使用了 JUnit 的命名约定来让 JUnit 来自动发现和执行你的测试。这个约定规定了，一个实现测试的方法，它的名字必须以“test”开头。
- 测试需要一个放置的地方。你将这个方法放到一个类中，这个类扩展了 JUnit 框架类 `TestCase`。我们一会儿再详细描述 `TestCase` 这个类。
 - 要表达你对被测试对象行为的期望，需要做一些断言。一个断言就是对你的期望的一个简单陈述。JUnit 提供了很多做断言的方法。在本例子中，你使用了“`assertEquals()`”，这就等于告诉 JUnit，“如果这两个值不相等，这个测试就算失败”。
- 当 JUnit 执行一个测试时，如果断言失败了——在本例中，如果 `inCents()` 没有返回 5 000——那么测试就算失败；如果没有断言失败，那么测试就算通过了。

这些都是重点，但是跟往常一样，越到细节越是麻烦。

1.2.3 理解 `TestCase` 类

`TestCase` 类在整个 JUnit 框架中处于核心的地位。你可以在 `junit.framework` 包中发现 `TestCase` 这个类。在 JUnit 开发人员当中，甚至是在一些高级的程序员当中，都有一个疑惑，那就是测试用例（`test case`）和 `TestCase` 类之间的关系。事实上，这是个命名冲突问题。“测试用例”（`test case`）这个术语通常指的是单个的测试，通过编码来校验某个特定的行为。那么，将多个测试用例都收集到一个类中，这个类是 `TestCase` 的子类，而且每一个测试用例都被实现为 `TestCase` 类中的一个方法，这的确有些奇怪。既然这个类包含了多个测试用例（`test case`），那为什么把它叫成“`TestCase`”而不是叫做“测试集”之类的名字呢？

在这里我们给出最好的解释：虽然在编写测试的时候，你需要创建一个“TestCase”的子类，并把每个测试用例都实现为这个新类的方法；但是在运行的时候，每个测试用例都会被当作 TestClass 子类的一个实例来运行。使用面向对象的术语来说，每个测试用例都是一个 TestCase 的对象，因此，这个名字还是合适的。不过，一个 TestCase 包含了很多的测试，这的确引起了术语的疑惑。这也是为什么我们如此小心地区分一个测试用例（test case）和 TestCase 这个类：前者是单个的测试，而后者则包含了多个测试，每个测试都用不同的方法来实现。为了更清楚的区分这两个概念，我们不会再（尽量做到不再）使用测试用例（test case）这个术语。要么就用测试（test）这个词，要么就用“TestCase”这个类名。在这本书的后面，我们还会把这个“TestCase”类称为“固定器（fixture）”。不再过多地解释这个描述，我们以后再谈这个“固定器”。现在我们把固定器想象成将多个测试自然组合到一起的方式，然后 JUnit 可以统一执行这些测试。“TestCase”类提供了一个默认的方式来确定哪些方法属于测试，但是你可以以自己的方式来组织测试。在第 4 章中，描述了多种不同的方法从你的测试中来创建测试套件。

在 JUnit 框架结构中，TestCase 类扩展了一个叫做 Assert 的工具类。这个 Assert 类提供了很多方法来让你对当前的对象的状态做“断言”。正因为 TestCase 扩展了 Assert，因此你可以不需要引用外部的类就能编写自己的断言了。在 JUnit 中，基本的断言方法如表 1.2 所描述。

表 1.2 JUnit 中的 Assert 类提供了多种做断言的方法

方法	它实做什么的
assertTrue(boolean condition)	如果 condition 为 false 则失败；否则通过测试
assertEquals(Object expected, Object actual)	根据 equals() 方法，如果 expected 和 actual 不相等则失败；否则通过测试
assertEquals(int expected, int actual)	根据 == 操作符，如果 expected 和 actual 不相等则失败；否则通过测试。对每一个原始类型：int、float、double、char、byte、long、short 和 boolean，这个方法都会都一个函数的重载。（参见 assertEquals() 的注释）
assertSame(Object expected, Object actual)	如果 expected 和 actual 引用不同的内存对象则失败；如果它们引用相同的内存对象则通过测试。两个对象可能并不是相同的，但是它们可能通过 equals() 方法仍然可以是相等的
assertNull(Object object)	如果对象为 null 则通过测试，反之看作失败

JUnit 还提供了正好和表中列出的逻辑相反的其他断言操作：assertFalse()、assertNotSame() 和 assertNotNull()；但是没有 assertNotEquals()，如果想有这个方法，需要更多的有关定制 JUnit 的知识，请参见第 3 章。

注意：有两个 `assertEquals()` 的重载函数稍有不同。用来比较 `double` 和 `float` 的版本需要第三个参数：“精确程度”。这个精确程度确定了两个浮点值需要有多接近才能被认为它们是相等的。因为浮点运算并不是完全精确的，你可能会决定“这两个值如果相差不超过 0.0001，那么它们就足够相近了”，用函数的方式来表达就是 `assertEquals(expectedDouble, actualDouble, 0.0001d)`。

1.2.4 失败信息

当一个断言失败了，加上一个简短的信息来描述这个失败的一些属性甚至是失败的原因，可以带来一些好处。每一个断言的方法都接受一个“String”类型的参数，这个参数不是必需的，它包含了一些信息，在断言失败的时候显示出来。当编写断言的时候，程序员是否应该把编写一个失败的信息作为通用的规定，这是一个有争论的问题。支持者申明，这样做能够增加代码的自我描述的能力，然而其他人认为，在许多的情况下，上下文就能将失败的实质表达得很清楚。读者可以自己去试试，然后比较两种结果来决定。

我们可以在 `assertEquals()` 方法中加入自己定制的失败信息，如果一个相等性的断言失败的时候，你就能看见这样的信息：

```
junit.framework.AssertionFailedError: expected:<4999> but was:<5000>
```

看看，这个定制的失败消息将问题的原因描述得再清楚不过了。

1.2.5 JUnit 怎样表示一个失败的断言

要想理解 JUnit 是怎样确定一个测试到底是通过了还是失败了，关键在于要了解这些断言的方法是怎样表示一个断言已经失败的了的。

要让 JUnit 测试能够自我校验，就必须对你的对象的状态做断言。当产品代码没有像你断言的那样，JUnit 必须做一个红色警告标记。在 Java 语言中和 C++、Smalltalk 一样，做警告标记的方法就是抛出一个异常。当 JUnit 断言失败了，断言的方法就会抛出一个异常来表示这个断言失败了。

更准确地说，当断言失败的时候，断言的方法会抛出一个错误：AssertionFailed-Error。下面是 `assertTrue()` 的源代码：

```
static public void assertTrue(boolean condition) {  
  
    if (!condition)  
  
        throw new AssertionError();  
  
}
```

当你断言某个条件应该是“true”而实际却不是的时候，这个方法会抛出 `AssertionFailedError` 来表示这个失败的断言。JUnit 框架会捕获这个错误，将这个测试标记为失败，并且记住哪个测试失败了，

然后再做下一个测试。在测试运行结束后，JUnit 会列出所有失败的测试，其他的测试被认为已经通过测试了。

1.2.6 失败和错误的区别

通常情况下是不希望自己编写的 Java 代码抛出错误的，而只应该抛出异常。通常来说，应该让 Java 虚拟机本身来抛出错误。因为一个错误意味着低级别的、不可恢复的问题，例如：无法装载一个类，这种问题我们也不期望被恢复。出于这个原因，也许我们对 JUnit 通过抛出错误来表示一个断言的失败而感到有些奇怪。

JUnit 抛出的是错误而不是异常，这是因为在 Java 中，错误是不需要检查的；因此，不是每个方法都必须声明它会抛出哪些错误。你可能会认为 `RuntimeException` 可以完成相同的功能，但是如果 JUnit 抛出这种在产品代码中可能抛出的异常的类型，JUnit 测试就会和你的产品相互影响。这种影响会降低 JUnit 的价值。

当你的测试包含了一个失败的断言，JUnit 会将它算做一个失败的测试；但是如果你的测试抛出了一个异常（并且没有捕获它），JUnit 将它看成是一个错误。这个区别是很细微的，却是非常有用的：一个失败的断言通常表示产品代码中有问题，而一个错误却表示测试本身或周围的环境存在着问题。也许你的测试期望了一个不该期望的错误的异常对象，或者错误地在一个 `null` 的应用上调用了函数。也有可能磁盘已经满了，或者网络连接不可用，或者是一个文件找不到了。JUnit 并不把这个算做产品代码的缺陷，因此它只是在表达，“有些不对劲了。我不能分辨这个测试是否通过。请解决这个问题并再重新测试一次。”这就是失败和错误的区别。

JUnit 的测试运行器会报告一个测试运行的结果，格式如下：“78 run, 2 failures, 1 error。”从这个结果可以判断有 75 个测试已经通过，有两个失败，还有一个没有结论。我们的建议是先去调查那个错误，解决了此问题以后再重新运行这个测试。也许在解决了这个错误以后，所有的测试都通过了！

1.3 A few good practices 一些好的实践

这本书中的一些诀窍来自于我们辛苦实践才获得的一些好的实践经验。冰冻三尺非一日之寒，我们也不期望你在一天内学会它们，但是遵守一些好的实践会给你更合适的起点。

1.3.1 测试和测试类的命名约定

命名非常重要。你的对象的名字，方法和参数的名字以及包的名字在交流的时候都起着非常重要的作用，用来说明你的系统是什么，是做什么的。别的程序员也可能坐在计算机旁边浏览你的代码，就从这些名字就能在心里形成一个你的系统的模型。如果不是这样，说明这些命名是错误的。也许这话说得太强硬了，我们也知道要找到合适的名字远不是一件容易的事情。对系统的各个部分的命名是件费时费力的工作，但我们相信找到合适的命名是值得我们去做的。

在测试中所要做的是：精确地给这些测试命名。这些测试的名称应该用少量的词语对测试的一个总结。如果不能总结的话，你就需要编写额外的文档来解释这个测试。一个测试到底是做什么的，你通过名字就能解释清楚，比起要看实现的代码来要好得多！你的目标是让代码像这本书一样容易

读懂——或更加容易。

测试命名

从测试本身开始：测试的名称应该描述了你想要测试的行为，而不是怎样实现一个测试。换句话说，一个测试的名字应该展示了测试的目的。如果你正在编写一个银行系统的对象，你可能会对“有人企图提出过多的钱款”这一特殊用例编写测试。你可以将这个测试命名为“testWithdrawWhenOverdrawn()”或“testWithdrawWith Insufficient- Funds()”。也许“testWithdrawTooMuch()”就够了。但“testWithdraw200Dollars ()”是一个值得怀疑的选择：这个名字可能描述了此测试要做什么，但没有说明是为什么。第一次见到这个测试的名称的时候，程序员可能要问，“提款\$200 是个特殊的用例呢？”我们建议要使测试的名称很明显才好。对于那些正常的用例（没有发生错误的直接场景），我们推荐简单地命名这些测试的行为就够了。对于从帐户提款成功的测试，“testWithdraw()”的名字就很好了。如果你不给出其他的描述，读者就会假设你在测试一个正常的操作。

通常约定使用下划线字符来放在行为名称和特殊用例之间。一些程序员喜欢用“testWithdraw Overdrawn()”或“testWithdraw_Zero()”。这样能使测试的方法更容易读懂：它隔离了测试的行为和被测试的特殊条件。我们采用此命名约定。在同一个行为名称之后可能会出现三四个特殊的用例，你可能会考虑将这些特殊用例移到一个特殊的“测试固定器”中——一个定义测试和测试所操作的对象的类。参见 3.7 节，“将对特殊用例的测试放到一个单独的测试装置里”，来得到这个技术的完整的描述和讨论。

命名测试用例的类

当命名测试用例的类时，最简单的规则就是以被测试的类来命名测试用例的类。换句话说，对“Account”类做的测试叫“AccountTest”，对“FileSystem”类做的测试的类叫做“FileSystemTest”，等等以此类推。这个命名约定带来的主要好处是，只要你知道被测试的类，就很容易找到这个类的测试。尽管我们推荐开始使用这个约定，重要的是要明白这仅仅是个约定，并不是 JUnit 的强制要求。我们感到惊奇的是，有大量的程序员都来问我们，如果测试用例的类变得非常大和难以管理应该怎么办？不管是 JUnit 还是其他，任何变得太大的类都应该拆分成一些小类。我们建议找出那些有共同结构的测试，将它们重新构造成一些单独的测试用例类。如果有人问，“不是说所有的这些都应该在同一个 TestCase 类中吗？”回答是“不是！”

我们以前提到过每一个测试都是 TestCase 子 类的一个实例。但那时我们没有提到，测试用例类仅仅是一个测试的容器，怎样将不同的测试分布到不同的容器中都由自己决定。在第三章中，我们描述了一些有用的方法。如果发现有三个测试应该在一起，应该将它们从其他的测试中分离出去，那就把它们移走吧。我们建议以这些测试的公共的部分来命名这个新的测试结构。如果有六个测试用例都是从账户中提钱，那么将它们移到一个新的测试用例中，叫“AccountWithdrawalTest”或“WithdrawFromAccountTest”。究竟选哪个取决于你是喜欢名词短语还是动词短语。我们推荐的是动词短语，一般会选“WithdrawFromAccountTest”。

1.3.2 测试的是行为，而不是方法

下面是我们另外一个编写测试的建议：测试应该以行为为中心，而不必担心是哪个类在测试。这也是为什么我们的测试名称建议用动词而不是名词：我们测试的是行为（动词），而不是类（名词）。另外，行为和方法之间的区别也许不那么明显：我们一般是用方法来实现行为的，所以测试行为必须要测试方法。但是并不完全是这样的。我们的确是用方法来实现行为的，但是怎样实现特定的行为取决于很多的因素，有些因素甚至归结为个人的偏好。当用方法实现行为的时候，我们有很多的决定要做：它们的名称，它们的参数清单，哪些方法是 public 的，哪些方法是 private 的，我们把方法放到哪些类中等——这就是一些方法有可能不同的地方，甚至就算背后的行为可能是一样的。实现的方式多种多样，测试并不需要知道这些。

有的时候单个的方法实现了所有需要的行为，在那种情况下，只需要直接测试那个方法就行了。更加复杂一点的行为需要许多不同的方法甚至是不同的对象一起协作来实现。如果你的测试太依赖于特定的实现，一旦需要重构（来提高设计结构）的时候，你会为自己找很多的麻烦。进一步说，有些方法只是参与到一个特定的功能中，而不是实现它。要单独测试这些方法是不值得花这么大力气的。这样做会使你的测试套件更加复杂（使用更多的测试），使得重构工作更加困难——所有这些都是因为没有在更高的一个层次上来测试行为。将重点放在测试行为上而不是每一个单独的方法上，你可以更好地平衡测试的覆盖度和重构需要的自由度。

为了说明这一点，考虑对一个栈（stack）的测试。一个栈（stack）提供了一些基本操作：压栈 push（添加到栈的顶端），弹出 pop（从栈的顶端移走），查看 peek（查看栈的顶端）。当要决定怎样测试一个栈的实现，下列的测试会浮现到脑海：

- ☐ 要弹出一个空的栈应该会失败。
- ☐ 要查看一个空的栈应该什么也找不到。
- ☐ 要将一个对象压栈，然后查看它，希望得到和刚才压栈相同的对象。
- ☐ 要将一个对象压栈，然后弹出它，希望得到和刚才压栈相同的对象。
- ☐ 要将两个对象压栈，然后弹出两次，希望得到对象的顺序和刚才压栈的顺序正好相反。

注意到这些测试都集中在固定状态——当操作执行的时候栈的状态——而不是操作本身。还需要注意这些方法是怎样组合起来提供特定行为的。如果“push()”不能工作，那么很难去校验“pop()”和“peek()”方法。其他的方法也是一样的。进一步来说，当使用栈时，一般会使用所有的三个方法，所以单独的测试它们，而不是根据它的内容总体上测试栈的行为是不是正确，是没有什么意义的。这是不是意味着设计得不好，而导致这些方法互相耦合呢？不是这样的，其实它再一次说明了这样一个事实：一个对象是一组互相紧密关联的方法作用在同一组数据上。对象的整体行为——在一定的状态下，对象的多个方法的组合行为——才是重要的。所以我们推荐将测试的工作重点放在整个对象上，而不是对象的一部分。

1.4 Summary 总结

无论是自己的代码还是别人的程序，我们都有过跟踪一个错误的经历。这个可怕的行为一般我们把它叫做“调试（*debugging*）”，一般包含两个不同的却相关的行为：一个是推断有什么可能已经出错了，另外一个是在黑暗中顺着细微的线索来断定哪些真的是出错了。当我们调试的时候，一般是从第一个行为开始；但是一旦发现事情比我们要担心的还要糟糕的时候，我们会开始第二个行为。在这个时候，没有办法知道我们什么时候能解决这个问题，甚至是 能不能解决这个问题。

最基本的调试技术应该是使用名为“`printf`”的 C 库函数，来将文本打印到屏幕上。就算是 Java 程序员也会说，“将一些 `printf` 放到那里，看看会发生什么”（尽管说 `println` 应该更合适，因为在 Java 中是这样调用的）。这个方法的注意是将错误的范围缩小到代码当中足够小的范围内，然后随意放置一些临时的代码来将一些变量的值打印到屏幕上。然后运行这个系统，重现那个错误，并且分析变量的值。

这个过程是不是听起来有点熟悉呢？

正如 James Bach 描述的那样，你所做的是探测性的测试。唯一不同的是你可以访问一个调试器来使整个过程更加容易：你可能只有通过日志文件或控制台来观察，并且最好观察的速度一定要快，因为 `println` 语句有可能在你准备好以前就打印出来了，而使你忽略了重要的信息。

为什么不编写一个测试呢？通过将范围缩小到对象的某个方法的调用，你已经定位到了出现问题的代码。编写一个测试来检验那个方法。并不需要搜索每一个你所能找到的变量的值，只要确定发生问题的那个方法的输入值是什么，在测试中就使用这个输入值。对你所希望的结果做一个断言，然后运行这个测试。不断地增加测试直到找到问题的根源为止。当你改变了那段该死的代码以后，运行你的测试来校验问题是不是解决了。这样做，每次你认为已经解决了问题的时候，能够增加反馈的频率（和效率）。当你解决了问题以后——最重要的是——保留这个测试！

没错：当你完成这个调试工作以后，你带走的不仅仅是又一个能够向程序员兄弟们炫耀的斗争故事，还应该得到的是这些能够防止这个错误以后再次入侵到你的系统的那些测试。你可能会忘记这个问题的原因，以及你是如何解决这个问题的，但是那些测试永远不会。将你辛苦工作的结果保存下来，以避免在产品发布的前两天在客户那里，或者当你向 CEO 做下一个产品演示的时候，又重新遇到这个问题。

停止调试吧。取而代之的是编写测试。

[2.1 测试你的 equals 方法\(1\)](#)

本章主要内容：

- 测试单个方法的通用技巧
- 测试 JavaBeans 的特殊技巧
- 测试接口的特殊技巧
- 如何在测试中验证两个对象是否相等

最简单的测试莫过于验证一个方法的返回值，这种测试是所有程序测试的基础。在本书的其余部分，我们会尝试将每一个复杂的测试问题进行拆分，并最终分解成这种最简单的测试。我们首先详细介绍这种最简单的测试，其余的测试技巧其实都是这种最简单测试的逐渐演进。即将介绍的测试技巧不仅告诉我们如何处理常见的测试问题，还将引入大量的、基本的专业术语，掌握这些术语非常有助于描述和解决更复杂的测试问题。如果验证一个方法的返回值可以称为“测试的原子”的话，那么这些测试方法就可以称为“测试的分子”。因为这些方法最基本，并且是构建复杂的功能组件的基本单位。到本书结束的时候，你将掌握绝大部分的测试问题的处理策略。

这里是如何测试一个有返回值的方法的例子，先调用这个方法，然后比较它的实际返回值和你预期的返回值：

```
public void testNewListIsEmpty() {  
  
    List list = new ArrayList();  
  
    assertEquals(0, list.size());  
  
}
```

size()方法返回列表中的元素的个数。如果我们手工进行测试，我们会把这个结果打印到屏幕上，通过眼睛去验证：“是 0 么？是！Ok，通过”。使用 JUnit 你可以走得更远，你可以制定一个“断言 (assertion)”来捕捉返回值：写一行代码描述你的期望值，然后让 JUnit 比较预期值与实际的返回值。JUnit 提供名称以“assert”开头的断言方法，你可以依此在测试中写断言。如果一个测试通过了，那就说明它的所有断言都为真。参照 Bill Wake 的三“A”理论：arrange (创建)、act (执行)、assert (断言)，我们前面举的例子实际上尽可能简单的遵循了这个模式：首先创建一个空列表 (arrange)，然后取得它的大小 (act)，最后验证它的大小是不是为零 (assert)。这大概是最简单的 JUnit 测试了。

assertEquals()方法既允许你比较基本类型的值，也允许你比较对象。回想一下，基本的类型包括：int、float、boolean、double、long、char、byte。你可以打开 JUnit 的 junit.framework 类，验证那些“assert”方法是如何实现的。framework 将基本类型当作数值来比较（而不是对象），所以只要 value 是一个值为“3”的基本数据类型，assertEquals(3, value)就会通过。如果你希望测试的方法返回一个对象而不是一个基本数据类型，你就需要做更多的工作来验证实际返回值和期望值是否一致。

如果方法的返回值是对象，那么有两种技巧可以用来验证期望值与该方法的返回值是否相等。第一种技巧是，将该方法的返回值的所有可读的属性值拿出来，依次与你期望值的相应属性值相比较。第二种是创建一个代表期望值的对象，然后使用一行代码将它与实际返回值进行比较。我们推荐第二种方法。为了举例说明他们的差别，请看下面的代码：

```
public void testSynchronizedListHasSameContents() {
```

```

List list = new ArrayList();

list.add("Albert");

list.add("Henry");

list.add("Catherine");


List synchronizedList = Collections.synchronizedList(list);

assertEquals("Albert", synchronizedList.get(0));

assertEquals("Henry", synchronizedList.get(1));

assertEquals("Catherine", synchronizedList.get(2));

}

```

这里我们测试 `Collections.synchronizedList()` 方法，这个方法被认为可以为列表（List）添加一个线程安全的新特性，而不影响列表的内容。我们通过测试来验证该方法是否真的做到了这一点。

`synchronizedList()` 方法非常简单，因为它直接返回了一个对象，可以让我们检查其正确性。我们将对象的每一个元素都视为可读的属性值，并将它们逐个与原来列表中的元素进行比较。这实际上就是我们前面介绍的第一种方法。它工作得很好，但也有缺陷，最显著的缺点是：它涉及到了太多的代码输入——尤其是对这么简单的测试而言，它的输入实在太多了。当然，我们也可以使用一个循环来比较两个列表中的相应元素。

```

public void testSynchronizedListHasSameContents() {

    List list = new ArrayList();

    list.add("Albert");

    list.add("Henry");

    list.add("Catherine");


    List synchronizedList = Collections.synchronizedList(list);

    for (int i = 0; i < list.size(); i++) {

        assertEquals(list.get(i), synchronizedList.get(i));
    }
}

```

```
}  
  
}
```

我们使用循环避免了重复的代码，这样好了一些。但同我们的思维模式相比，它似乎仍然有太多的代码，不管读还是输入都很费劲。如果用我们的语言描述这个测试，我们会说：“我们将一些对象放入一个列表，并把它传递给 `synchronizedList` 方法，那么返回的列表应该与原列表相同”。使用“相同”这个词可能不太严谨，因为 Java 编程中，只有当两个引用指向内存中的同一个对象时，才被认为“相同”。可能更准确一点的说法应该是：“新的列表中可比较的元素与原列表都相等”。更好的说法是：“两个列表中相应的元素应该相等”。而我们的代码似乎仅仅表明：“列表应该与同步后的列表相等”，如果我们将“相等”定义为“有相同的元素”的话。令人欣慰的是，Java 本身就有这样一个方法：`Object.equals()`。更令人振奋的是，`List.equals()` 方法要求：当且仅当两个列表的相同索引指向的元素相同的时候，两个列表才算“相等”。太棒了！这样我们就不仅可以正确地完成测试，还可以将判断相等性的责任推给 `List` 类自身，因为它们自己已经配备了做这件事的最好方法。

```
public void testSynchronizedListHasSameContents() {  
  
    List list = new ArrayList();  
  
    list.add("Albert");  
  
    list.add("Henry");  
  
    list.add("Catherine");  
  
  
    List synchronizedList = Collections.synchronizedList(list);  
  
    assertEquals(list, synchronizedList);  
  
}
```

这样做极大地简化了测试流程。如果使用 `Arrays.asList()` 方法，我们还可以进一步简化：

```
public void testSynchronizedListHasSameContents() {  
  
    List list = Arrays.asList(  
  
        new String[] { "Albert", "Henry", "Catherine" });  
  
    assertEquals(list, Collections.synchronizedList(list));  
  
}
```

```
}
```

这里只写了一句声明来完成测试，但这样看上去却最可读。我们已经开始了好的思路。

如果你要写最简单的测试代码，那么请遵循如下步骤：

1. 创建一个对象并初始化它。
2. 调用一个方法，它返回实际的结果（`actualResult`）。
3. 创建一个“预期结果”，该结果可以是基本类型，也可以是对象。
4. 调用 `assertEquals(expectedResult, actualResult)` 方法进行比较。

如果坚持这样编写代码，你就会慢慢养成使用对象的 `equals()` 方法的习惯。如果了解 `equals()` 方法应该如何实现，请参考 2.1 节“测试你的 `equals` 方法”。如果你掌握了这种方法，其他的 JUnit 测试几乎都可以化简为这种方法。

当测试 `get` 和 `set` 方法时，你也可以直接使用这一技巧（参考 2.4 节“测试获取器”和 2.5 节“测试设置器”）。为了有效地使用这一技巧，你需要恰当地实现你的 `equals()` 方法（参考 2.1 节“测试你的 `equals` 方法”）。如果不能直接采用这一技巧，你需要借助其他的办法来获取返回值，然后与你的期望值进行比较（请参考 2.2 节“测试一个没有返回值的方法”）。本章的其余节主要介绍各种基本的测试技巧，这些技巧是本书其余章节的基础。此外，这些技巧阐述了那些最简单、最常见的测试问题。

2.1 Test your equals method

2.1 测试你的 `equals` 方法

◆ 问题

如何测试 `equals()` 方法。

◆ 背景

令人震惊的是，虽然一个强大的面向对象的设计要求恰当的实现 `equals()` 方法，许多程序员却没有正确地做到这一点。如果你想正确地掌握本书介绍的技巧，那么你就必须恰当地实现一些类的 `equals()` 方法，尤其是那些用来存放数据的类。一般来讲，你不会去比较那些面向行为或者面向处理过程的类，但你需要去比较那些面向数值对象的类，因为那些面向处理过程的类会使用面向数值对象的类作为输入和输出。为了将这些数值对象存储到容器（比如 `List`，`Set`，`Map` 等）中，你需要恰当地实现其 `equals()` 和 `hashCode()` 方法。当你对数值对象进行测试的时候，这就显得尤为重要，因为你将会花费大量的时间来判断两个数值对象或者两个基本数据类型是否相等。

注意：一个数值对象代表一个值，比如 `Integer`、`Money` 对象或者 `Timestamp` 对象。数值对象与其他的对象的不同在于：如果你有二十个不同的对象代表

同一个值，那么这些对象就是可相互替代的。你是否使用内存中的同一个对象并不重要：如果你在内存中有三个代表 5 的数值对象，那么你使用一个来代替另一个不会影响程序的功能。换句话说，即使是内存中的不同对象，我的 20 美元与你的 20 美元也不会有什么不同。不过你需要通过恰当的实现 `equals()` 方法来反映这一点。

基本类型（包括整型，浮点型等）的封装类具有数值对象的特性，`String` 也是这样。这本书中的许多例子中都使用一个“`Money`”类型的对象，它也是一个数值对象。

如果你的对象不是数值对象，那么就无须去测试它们的相等性，当然也就不需要重载其 `equals()` 方法。这个原则对大多数情况都适用。

注意：`equals()` 方法的快速回顾——`equals()` 方法的要求并不复杂，但对于那些在数学课上没有认真学习相等性的程序员来说，可能也不大容易，所以我们在这里作一个简单的回顾。`equals()` 方法必须具备三个著名的属性：即反身性（*reflexive*）、对称性（*symmetric*）和传递性（*transitive*）。为了方便记忆，数学界也把它称为“*RST*”原则。

- 反身性的意思是一个对象等于其自身。
- 对称性的意思是，如果我等于你，那么你也等于我，以此类推。
- 传递性的意思是，如果我等于你，你等于那边的那个家伙，那么我跟那个家伙也相等。用古代的话说，“等量与等量是相等的”。

除了这些数学特性，`equals()` 方法还必须是一致的：也就是说，不论你调用了多少次，只要比较的对象不变，`equals()` 方法的结果就必须是一致的。最后一个定理是，任何一个对象都不等于“`null`”。

知道了这些，我们就可以开始测试数值对象的 `equals()` 方法了。

◆ 诀窍

只要你曾经尝试过编写各种的测试程序，你就会马上发现这实在是一件繁重的工作。如果你想在本文提供的诀窍之前，先了解一下这样的工作，请跳到本章的讨论部分，然后再回到这里。如果你想直接获得答案，那么我告诉你，我们借助于 Mike Bowler 的杰出贡献，极大地简化了这一节的内容。他写的开源包 `GSBase`（<http://gsbase.sourceforge.net>）为 JUnit 测试提供了许多工具，其中就包括一个相等性测试器（`EqualsTester`）。这个测试器能在数值对象上运行一整套的测试，以确定该对象的 `equals()` 方法是否具备所有必要的属性。

清单 2.1 使用 `EqualsTester` 测试 `Money` 类的 `equals()` 方法

```
package junit.cookbook.common.test;
```

```
import junit.cookbook.util.Money;

import junit.framework.TestCase;

import com.gargoylesoftware.base.testing.EqualsTester;

public class MoneyTest extends TestCase {

    public void testEquals() {

        Money a = new Money(100, 0);

        Money b = new Money(100, 0);

        Money c = new Money(50, 0);

        Object d = null;

        new EqualsTester(a, b, c, d);

    }

}
```

让我们详细解释一下传递给 EqualsTester 的参数：

- ☐ a 是一个基准对象，其他的三个对象都要与它进行比较。
- ☐ b 是内存中另外一个对象，它与 a 的值相等。
- ☐ c 是不等于 a 的一个对象。
- ☐ 如果你要测试的类是 final 型（即不能被继承），那么 d 应该是“null”。实际上 d 应该代表一个看上去与 a 相等，但实际上不相等的对象。“看上去相等”的意思是，举例来讲，d 与 a 有相等的属性值，但 d 是 a 的子类的对象，有比 a 更多的属性值，因此不等于 a。

返回到我们的例子中，因为 Money 允许被继承，因此若将 d 设为“null”，这个测试就不会通过。我们必须考虑将 Money 设置为 final 型，或者将测试代码改成如下形式：

```

public class MoneyTest extends TestCase {

    public void testEquals() {

        Money a = new Money(100, 0);

        Money b = new Money(100, 0);

        Money c = new Money(50, 0);

        Money d = new Money(100, 0) {

            // Trivial subclass

        };

        new EqualsTester(a, b, c, d);

    }

}

```

`EqualsTester` 为数值对象的相等性测试提供了一个便捷的解决方案，这里“相等性”的定义为：“具有相同属性值的特性”。这个方案可以应用于绝大多数应用程序。

◆ 讨论

为了理解为什么我们推荐使用 `EqualsTester`，让我们尝试一下自己编写测试代码。

首先编写反身性的测试代码：

```

public class MoneyEqualsTest extends TestCase {

    private Money a;

    protected void setUp() {

        a = new Money(100, 0);

    }

}

```

```
        public void testReflexive() {  
            assertEquals(a, a);  
        }  
    }  
}
```

下一个属性是对称性，我们需要创建另外一个 Money 对象：

```
package junit.cookbook.common.test;  
  
import junit.cookbook.util.Money;  
import junit.framework.TestCase;  
  
public class MoneyEqualsTest extends TestCase {  
    private Money a;  
    private Money b;  
  
    protected void setUp() {  
        a = new Money(100, 0);  
        b = new Money(100, 0);  
    }  
  
    public void testReflexive() {  
        assertEquals(a, a);  
        assertEquals(b, b);  
    }  
}
```

```

    public void testSymmetric() {

        assertEquals(a, b);

        assertEquals(b, a);

    }
}

```

也许你会将 `testSymmetric()` 方法用逻辑关系来实现，就像下面的代码一样：

```

public void testSymmetric() {

    assertTrue(!a.equals(b) || b.equals(a));

}

```

当然，这就要求你能够只用逻辑与、逻辑或和逻辑非表示的关系式实现一个方法，并不是所有人都能做到这一点。令人欣慰的是，这并不重要，因为你可以按照我们前面介绍的办法来实现。在那里，任何一个断言不通过，整个测试就会失败。换句话说，那些断言之间存在逻辑与关系，因此相等性的各个要素都会得到测试。既然简单直接的方法能够完成任务，我们就应该去使用它。其中，我们使用 `b` 进行反身测试，仅仅使用了一个对象就达到了目的。

下一个特性是传递性，这一般要求三个对象。但是，反身性加上对称性就足以证明传递性。例外的情况极其少见，因此我们在这里不作阐述，而把它留作单独的一节进行讨论。详细情况请参考附录 B.2：“奇异性与传递性”。

现在我们已经确认了：如果我们认为对象应该相等，那么它们就被 `equals()` 方法看作相等的。接着我们要确认：如果我们认为对象不相等的话，`equals()` 方法也不认为它们相等。这就要求找一个与前两个对象不相等的对象。就像前面的例子一样，我们在测试代码中为这个新对象添加一个声明：

```

public class MoneyEqualsTest extends TestCase {

    private Money a;

    private Money b;

    private Money c;

    protected void setUp() {

        a = new Money(100, 0);

```

```

        b = new Money(100, 0);

        c = new Money(200, 0);
    }

    public void testReflexive() {

        assertEquals(a, a);

        assertEquals(b, b);

        assertEquals(c, c);
    }

    public void testSymmetric() {

        assertEquals(a, b);

        assertEquals(b, a);

        assertFalse(a.equals(c));

        assertFalse(c.equals(a));
    }
}

```

这个测试说明 c 应该等于它自身（因为这对任何一个对象都应该成立），但 a 不等于 c，反之亦然。你也许注意到了 `assertFalse(a.equals(c))`。在 JUnit 中没有 `assertNotEquals()` 方法，但你可以在插件中找到它，比如 JUnit-addons。JUnit 社区中已经有人呼吁添加这项功能，但到目前为止，JUnit 还没有实现它。这个方法其实很容易实现，如果你愿意自己实现它，我们建议你不妨去试试。

目前为止，我们已经验证了三个主要的特性。为了证明一致性，我们需要在两个对象上重复多次相等性测试。因为不能使用无限循环，因此选择一个足够大的循环次数，以保证结果的可信度，同时

不会导致测试运行过慢：

```
public class MoneyEqualsTest extends TestCase {

    // No changes to the other tests

    public void testConsistent() {

        for (int i = 0; i < 1000; i++) {

            assertEquals(a, b);

            assertFalse(a.equals(c));

        }

    }

}
```

虽然进行了 1 000 次循环，但在普通的膝上电脑上也仅仅需要 0.05 秒，所以这个测试应该算足够快了。最后要做的是，将对象与 null 进行比较：

```
public class MoneyEqualsTest extends TestCase {

    // No changes to the other tests

    public void testNotEqualToNull() {Elementary tests

        assertFalse(a.equals(null));

        assertFalse(c.equals(null));

    }

}
```

我们不用比较 b 和 null，因为前面的测试早就证明了 a 和 b 是相等的。如果你想加上这个额外的测试，也没有什么坏处，但我们不认为这能说明任何问题。

到这里，我们已经验证了 equals() 方法的所有特性。虽然这不是一个穷举测试，但已经能在绝大多数应用程序中找出 equals() 方法存在的缺陷。你可以参考 EqualsTester 的源代码，看看 Mike Bowler 添加的测试中，还有哪些是值得实现的。

注意：除了测试 `equals()` 方法，`EqualsTester` 还验证了 `hashCode()` 方法对 `equals()` 而言是否也是一致的。也就是说，如果两个对象是相等的，那么它们的哈希码也相等，但反之不必成立。没有这种测试，就有可能导致我们在 `Map` 中用一个关键字对象存入一个对象后，无法用另外一个关键字对象将它取回，即使这两个关键字对象是相等的。我们就有过这样的经历。如果你不想或者不必自己去实现一个哈希算法的话，我们推荐你将 `hashCode()` 方法实现为抛出一个 `UnsupportedOperationException` 异常。这符合对象的基本要求，但如果你把这个对象当作关键字添加到 `Map` 中就会显式地失败。另一个办法是返回一个常数，比如 0，但这样做实际上将哈希算法变成了线性搜索，如果经常这样干的话，会带来效率上的问题。

替代方案

JUnit-addons 也提供了一个相等性测试器，称为 `EqualsHashCodeTestCase`。它与 `EqualsTester` 实质是一样的，但是代码不同。你可以继承这个测试类，但需要重载用来取得数值对象的方法。其中，一个方法是 `createInstance()`，它返回一个基准对象；另一个方法是 `createNotEqualInstance()`，它返回一个与基准对象不相等的对象。每次使用数值对象，你都需要调用这些方法来返回一个新的对象，而不能创建一个对象并一直使用它。一般来讲，测试程序会调用 `createInstance()` 方法两次来生成两个对象，然后使用 `equals()` 方法判断它们是否相等。但如果你直接使用同一个对象，测试就会失败！所以你必须牢记这一点，才能正确地进行测试。清单 2.2 展示了我们如何使用这个测试器来测试 `Money` 类。

清单 2.2 使用 `EqualsHashCodeTestCase` 测试 `Money.equals()` 方法

```
package junit.cookbook.common.test;

import junit.cookbook.util.Money;

import junitx.extensions.EqualsHashCodeTestCase;

public class MoneyEqualsTestWithJUnitAddons
    extends EqualsHashCodeTestCase {

    public MoneyEqualsTestWithJUnitAddons(String name) {
        super(name);
    }
}
```



```

        protected Object createInstance() throws Exception {

            return Money.dollars(100);

        }

        protected Object createNotEqualInstance() throws Exception {

            return Money.dollars(200);

        }

    }

```

现在总结一下两种工具的主要差别：你在测试中可以直接使用 `EqualsTester`，但如果要使用 `EqualsHashCodeTestCase`，你必须继承它。另外，`EqualsTester` 测试了子类的相等性问题，而 `EqualsHashCodeTestCase` 没有。除了这个，两种工具实际上是没有差别的。

◆ 相关

- ☐ B.2—奇特性与传递性
- ☐ GSBase (<http://gsbase.sourceforge.net>)
- ☐ JUnit-addons (<http://junit-addons.sourceforge.net>)

2.2 Test a method that returns nothing 测试一个没有返回值的方法

◆ 问题

如何测试一个没有返回值的方法。

◆ 背景

JUnit 新手常问的一个问题是：“我怎么测试一个返回值类型为 `void` 的方法？”这个问题之所以存在，主要原因是没有一个直接的办法可以断定这种方法到底做了什么，因为你不能将一个期望值与一个实际的返回值进行比较。这就需要你想其他的办法，来描述这种方法的期望行为，并将其与实际的行为进行比较。

◆ 诀窍

如果一个方法没有返回值，它一定有一些可观测的副作用，比如说改变了一个对象的状态，否则的话就说明它什么也没有做。如果一个方法什么也没有做，就没有必要去测试它，同样也就没有必要去使用它，所以你可以放心地忽略它。如果它有一个可观测的副作用，你须要识别出这一点，并以此为基础，设置合适的断言。

让我们测试一下将一个对象放入容器。Collection.add(Object)方法没有返回值。尽管如此，我们有一个直接的方法来验证这个对象是否被成功地添加到了容器中：

1. 首先创建一个空容器。
2. 查询该容器应该得不到任何项。
3. 将对象添加到容器中，并查询容器。
4. 现在容器包含了刚才添加的项。

将该过程写成代码，用来测试 ArrayList.add()方法：

```
public class AddToArrayListTest extends TestCase {  
  
    public void testListAdd() {  
  
        List list = new ArrayList();  
  
        assertFalse(list.contains("hello"));  
  
        list.add("hello");  
  
        assertTrue(list.contains("hello"));  
  
    }  
  
}
```

我们看到，List.contains(Object)可以验证 add()方法正确执行时带来的副作用。虽然 add()方法没有返回值，我们仍然可以验证：add()方法的正确执行会带来列表状态的变化。

如果一个方法从某个地方获取数据，但并不显式地声明数据是否被成功的装载，我们也可以使用这一技巧进行验证。这个测试从一个配置文件中装载数据，然后找出我们期望的、关于配置文件的属性值，并对其设置断言：

```
public void testLoadProperties() throws Exception {  
  
    Properties properties = new Properties();  
  
    properties.load(new FileInputStream("application.properties"));
```

```

    assertEquals("jbrains", properties.getProperty("username"));

    assertEquals("jbra1ns", properties.getProperty("password"));

}

```

注意，这个测试有些不可靠，因为它从测试程序之外，也就是从文件系统中的文件里获取数据。我们将在 5.3 节“使用内联数据文件”中阐述这一问题。这个测试使用 `load()` 方法，并验证它的主要副作用：期望的数据被正确地配置文件中装载进来。（此外，我不使用密码做任何事，所以我并不验证它。）

◆ 讨论

这个办法是否有效，取决于要测试的方法是否有可观测的副作用。任何一个方法（在这一点上，也可以说任何一段代码），要么有返回值，要么有副作用。否则的话，按照定义，它就没有任何行为。仅有的问题是，你的应用程序是否允许你去查看它的副作用。比如，缓存就通常不提供查看其内容的方法，因为缓存机制通常是实现上的细节问题。如果没有验证是否命中某个缓存的方法，就没有办法观察缓存机制的行为了。当然了，如果你的缓存根本不提供验证缓存命中的手段，你怎么知道它到底是否缓存了什么东西呢？

即使没有可观测的副作用，你还可以使用另一个方法的返回值来测试这个方法。回到前面那个列表的例子，我们使用 `contains()` 方法来测试 `add()` 方法。通常 `add()` 方法会被用来测试 `contains()` 方法，但我们前面写的测试恰恰是验证 `add()` 方法的！这好像掉进了一个“鸡生蛋，蛋生鸡”的问题：如果我们事先不能保证 `contains()` 方法的正确性，我们就不能保证 `add()` 方法的正确，反之亦然。这是否意味着我们的测试不能说明任何问题呢？

答案是否定的。我们要测试的是行为，而不是方法。当我们说“我们测试向容器中添加对象”时，将导致大量的操作：添加、删除、清空、包含、查找等等。所有这些容器操作都属于测试添加对象的行为。总的来讲，方法实现了特定的操作，比如将多个对象包含在容器中或者删除重复元素（比如对 `set` 类来讲）。我们建议你考虑更高的层次——测试行为，而不是测试单个方法。当然，某些行为非常直接，并被一个方法完美地实现，但我们关心的是对象，而不仅是方法。如果我们需要调用 `add()`，`contains()` 和 `clear()` 方法来验证元素被添加的时候，容器是否正确地整理了它们，那我们就调用这些方法。我们会在整本书中更多的讨论这个问题。

注意： 测试才是规范！——如果 `add()` 方法和 `contains()` 方法存在缺陷，以至于一个方法中止会导致另一个也跟着中止，结果测试通过了，我们就认为行为是正确的。我们寄希望于未来的测试能找出这些缺陷。如果一直没有找出这些缺陷，我们的错误的代码就会一直执行错误的行为。如果代码执行错误的行为，测试却没有失败，那么代码还有缺陷吗？越来越多的程序员说：“没有”。他们会说，“测试就是标准”，也就是说，他们根据代码测试是否通过，来说明代码是否具有某种行为。直到测试通过，一个特性才会被展示。每一个测试都能证明代码能执行某个特定的操作。因此，如果没有测试能够证明，一个对象被添加到同一个列表中两次，那么我们就不能假设可以这样做。这就会给程序带来更多的压力，迫使他们写足够的测试代码。

如果没有办法可以观测一个方法的副作用，那么请你为了测试创建一个出来。虽然你可能不喜欢“仅仅为了测试”而刻意添加代码，但我们坚信：如果查询方法是简单的（实际上也确实是简单的），那么添加一个简单的查询方法往往是值得的。在大多数情况下，将一个不可观测的副作用转化为可观测的，不仅仅对测试有益。一个例子是，可以允许一个可重用的类在一个更大的、获取了它的类中出现。当你有了更多的JUnit使用经验的时候，你就会自己发现这一点。在整本书中，我们会根据需要更多的阐述这个问题。

◆ 相关

- 5.3 —使用内联数据文件
- 14.2—测试可观测的数据源
- B.4 —仿制对象概览

2.3 Test a constructor 2.3 测试构造函数

◆ 问题

如何测试一个构造函数。

◆ 背景

测试一个构造函数有点像自己咬自己的尾巴。

直接的测试构造函数的办法莫过于使用构造函数创建一个对象，然后使用 `assertEquals()` 方法将它与期望值进行比较。这种测试将非常简单，因为你想这样干的话，写一行代码就够了。这其中只有一个问题：如果你要创建期望值，那么没错，就也需要使用你要测试的构造函数。这样的测试是难以令人满意的，这就好比说：“如果构造函数正常的话，那么它是正常的。”

那好，我们应该如何测试一个构造函数呢？

◆ 诀窍

如果你的类有可读的属性值的话，那么办法很简单：你可以将它们与你传递给构造函数的参数进行比较。在这里，可读的属性值可以是一个直接的变量域，也可以通过 `get` 方法获取。

注意：仅对测试而言——最常见的测试类的方法是，完全通过它们的 `public` 接口。如果你把这个作为准则的话，有些情况下你就必须给类添加 `public` 接口（一般是一个 `get` 方法）以取得需要的数据，用来验证特定的行为。这是一个编程中有争议的话题：添加这种方法似乎“仅仅为了测试”而破坏了数据的封装，人们对此看法迥异。这种情况下，只能由你自己根据具体的情况来衡量其中的差别，并决定采用那种办法。

最简单的构造函数测试程序是：暴露所生成对象的内部状态，以验证被传递给构造函数的参数是否被设置成正确的属性值。下面是一个简单的例子：

```
public void testInitializationParameters() {  
  
    assertEquals(762, new Integer(762).intValue());  
  
}
```

我们使用 `intValue()` 方法获取 `Integer` 对象内部的属性值，以验证我们传递给构造函数的参数是否被正确的储存到了对象中。虽然这里没有名为 `get` 的方法，但实际上 `intValue()` 就相当于 `get` 方法。

如果你的类不能看到可读的属性值，那你就必须另外创建一些可观测的副作用，用来验证构造函数的正确性。你可以写一个名为 `isValid()` 的方法来验证所创建对象是否正常，而且这种做法不会破坏数据隐藏的原则。这样的测试一般是如下的样子：

```
public void testInitializationParameters() {  
  
    BankDepositCommand command = new BankDepositCommand(  
  
        "123", Money.dollars(125, 50), today());  
  
    assertTrue(command.isValid());  
  
}
```

这种技巧也可以用于测试 `JavaBeans`。详细内容请参考 2.7 节“测试 `JavaBean`”。

◆ 讨论

大多数这样的测试都显得过于简单，甚至有人认为根本不值得写。许多程序员都这样认为，实际上我们也经常这样做。某些情况下，可能会恰恰因为我们没有写测试程序而导致程序的缺陷。因此，你应该根据你自己的经验和错误的概率来决定：是否应该写这样的测试，以保证在进行下一步工作之前，已经对程序进行了足够多的测试。一个流行的说法是：“一直测试到你的恐惧变成厌烦。”

如果你想获得一个更快、更有效的测试方案，请使用如下的方法，这个方法可以验证任何对象初始化时被赋予的默认值。考虑一个 `Money` 类，这个类如果在构造时没有获得任何参数，它的对象默认值为 0 美元。测试程序大致为如下的结构：

```
public void testDefaultInitializationParameters() {  
  
    assertEquals(0L, new Money().inCents());  
  
}
```

这里，我们使用的 `inCents()` 方法获取 `Money` 对象的基本值，以分为单位（这样可以避免使用浮点数），以验证对象的默认值为 0 美元，也就是 0 美分。

然而，一般来讲，一个构造函数本身很难完成自己的相关测试。大多数构造函数要么不接受参数，

要么将参数存入实例变量，或者将参数传给其他对象，还有其他更有趣的例子。如果你担心将参数传递给实例对象可能导致程序异常，那么请参考附录 B。如果你的构造函数实际上代表一些更有趣的方法，那么请测试这些方法而不要去测试构造函数。如果你的构造函数初始化它的参数，那么请参考 2.8 节“测试是否抛出正确的异常”，以获取一些关于那些会抛出某种特殊异常的方法的测试实例。

替代方案

你可能不愿意暴露你的对象中的数据，因为你（除了测试）没有其他的理由来做这件事。那么在这种情况下，你如何才能验证是否你传递给构造函数的参数被正确的储存了呢？你可能会马上想到这样的方法：“既然测试不能提取出实际的值，那就应该将期望值传递给对象，并让它进行比较。下面就是采用这种办法的一个例子，仍然使用 Money 为例：

```
public void testValueInCents() {  
  
    assertTrue(new Money(0, 50).valueInCentsIs(50));  
  
}
```

这个测试使用了一个新方法：valueInCentsIs()，这个方法比较了参数和对象的内部状态，返回它们是否相等。这种办法同样有效，但我们的观点是，相比使用 inCents()方法和在测试程序中比较结果的方案，这种办法显得不够清晰。这个是个人的倾向问题，你不一定要采纳我们的意见。最好两种办法都试一试，比较一下它们的结果有什么不同。

关于这一点我还想说，valueInCentsIs()方法并不比 inCents()方法更好地隐藏了数据。这个方法想实现类似 equals()方法的功能，但有些失败。如果你想用这样的功能，你应该直接去使用 equals()方法，但就像我们前面说的一样，这将导致使用构造函数来测试构造函数。既然这条路走不通，那我们就应该走另一条最简单有效的路：添加 inCents()方法。这个方法看上去似乎直接暴露了域内的数据，但实际上这是数据存储方案的事情。如果 Money 类改成使用两个域（元和分）存储数据，并相应地修改 inCents()的实现代码，Money 的使用者并不需要知道这种差异。就算你把这个方法的名称直接改成 getCents()（这样看上去更像直接获取其中的数据）也没有关系，因为这个方法只有读的权限，而且数据是被快速计算出来的，而不仅仅代表域内的一个值。实际上数据仍然是隐藏的。

陷阱

在结束这个话题之前，我们想展示一些我们经常看到的、新手写的构造函数的测试程序。如果你仔细审查，会发现这些测试不仅没有必要，而且在测试错误的东西：

下面就是一个常见的构造函数“测试程序”：

```
public void testConstructorDoesNotAnswerNull() {  
  
    assertNotNull(new Integer(762));  
  
}
```

根据 Java 的语言规范，一个构造函数要么在内存中创建一个新的对象，要么抛出一个异常，所以除非 Java 的翻译器或者编译器出错，构造函数决不会返回 null。其他类型的创建方法可能会返回空表示对象没有被创建，但我们认为这不至于造成混淆。

下面是另一个常见的构造函数“测试程序”：

```
public void testConstructorAnswersRightType() {  
  
    assertTrue(new Integer(762) instanceof Integer);  
  
}
```

同样根据 Java 语言规范，如果一个构造函数返回一个对象，那么它只能返回自己类的对象。在 Java 中只有通过工厂方法才能生成多种结构的类。因此，没有必要验证构造函数的返回值类型，因为语言本身已经保证了它的正确性。

前面这两个测试实际上是在测试语言平台。只有在 Java 语言本身有错误的情况下，这种测试才会失败。除非你要测试 Java 本身，否则费劲去测试那些你控制不了的东西是完全没用的。还有足够的代码等着你去测试，所以不要浪费时间去测试语言平台。

◆ 相关

- ☐ 2.4—测试获取器
- ☐ 2.5—测试设置器
- ☐ 2.7—测试 JavaBean
- ☐ 14.2—测试可观测的数据源
- ☐ B.1—简单得不能拆分

2.4 Test a getter2.4 测试获取器

◆ 问题

你想测试一个对象的 *get* 方法，但许多测试都显得过于简单。你想知道哪些需要测试，哪些不需要。

◆ 背景

JUnit 的新手通常也是程序测试的新手。他们瞪大了眼睛，满腔热情，什么东西都想测试一番。因为他们刚刚开始测试，因此他们主要测试程序中最简单的部分，而没有几个方法比那些程序中随处可见的、名字以“get”开头的方法更简单了。满腔热情加上缺乏经验，导致了程序员编写了非常非常多

的关于这种方法的测试。结果导致了他们情绪的低落：因为他们觉得没有从测试中发现多少问题。或许本节的内容可以帮助解决这个困惑。

关于是否应该编写关于 *get* 方法的测试程序，我们的探讨基于如下通用原则：不要去测试那些过于简单以至于很难出现错误的方法（详细内容请参考附录 B 中的相关短文）。如果一个方法只是简单地返回域中的属性值，那么除非编译器或者解释器工作不正常，否则很难出错。就像下面这个方法，很难有编码上的失误（除了排版问题）会导致它不能正常工作：

```
public class Money {  
  
    private int cents;  
  
    // Code omitted for brevity  
  
    public long inCents() {  
  
        return cents;  
  
    }  
  
}
```

*inCents()*方法本身确实不会出现异常。因此，我们不推荐你特意为它编写测试程序。此外，因为像 *get* 之类的方法非常简单，以至于很难有错误，因此一般就假设它能正常工作，并使用它来测试构造函数。详细情况请参考 2.3 节“测试构造函数”。

◆ 诀窍

第一原则是：如果一个 *get* 方法只是简单的返回域内的值，那么就不用考虑去为它写测试程序；但是，如果这个方法做了一些更为复杂的操作，那么还是请考虑一下。如果你决定为它写测试程序，其实也很简单：因为一个 *get* 方法有返回值，所以可以直接比较期望值和实际返回值。我们在本章的介绍中已经描述了如何写这类测试程序。

如果需要验证的 *get* 方法确实做了一些实在的工作，那么我们就肯定有办法测试它。这种类型的测试一般使用本章开头部分介绍的技巧：一般来讲，你可以向构造函数中传递参数，然后使用 *get* 方法获取相应的属性值。下面的例子摘自 Dave Thomas 和 Andrew Hunt 所著的 *Programming Ruby* 一书，不过我们把它改成了 Java 语言版。在类 *Song* 中，构造函数接收三个参数：歌名，唱片的参与艺术家名和歌曲的时长（以秒记）：

```
package junit.cookbook.common.test;
```



```

import junit.cookbook.common.Song;

import junit.framework.TestCase;

public class SongTest extends TestCase {

    public void testDurationInMinutes() {

        Song song = new Song("Bicyclops", "Fleck", 260);

        assertEquals(4.333333d, song.getDurationInMinutes(),

            0.000001d);

    }

}

```

就像我们在第一章提到的那样，如果要比较浮点数，你就必须设定精度。这个例子中，我们设定误差为百万分之一分钟。

在这个例子中，歌曲的时长（以秒计）存储在域中，以分计的时长则是被计算出来的。这里我们使用 260 代表歌曲时长的秒数，并作为参数传递给构造函数，`getDurationInMinutes` 方法接收这个参数，并计算出相应的以分钟为单位的时长。作者建议写这个测试程序的主要原因是，歌曲的时长是以秒为单位存储的，而 `get` 方法通过计算返回以分为单位的时长。这个方法有两种不同的实现。第一种很直接：`getDurationInMinutes()` 方法根据要求执行运算，如清单 2.3 所示：

清单 2.3 `Song.getDurationInMinutes()` 方法

```

package junit.cookbook.common;

public class Song {

    private String name;

    private String artistName;

    private int duration;

```

```

    public Song(String name, String artistName, int duration) {

        this.name = name;

        this.artistName = artistName;

        this.duration = duration;

    }

    public double getDurationInMinutes() {

        return (double) duration / 60.0d;

    }

}

```

清单 2.4 是另一种可行的实现方法。

清单 2.4 Song.getDurationInMinutes()的另一种实现

```

package junit.cookbook.common;

```

```

public class Song {

    private String name;

    private String artistName;

    private int duration;

    private double durationInSeconds;

    public Song(String name, String artistName, int duration) {

        this.name = name;

        this.artistName = artistName;
    }
}

```

```

        this.duration = duration;

        this.durationInSeconds = (double) duration / 60.0d;

    }

    public double getDurationInMinutes() {

        return durationInSeconds;

    }

}

```

这种实现在构造函数中就将时长进行了格式转换，以备将来使用。这是一种效率优化的策略。这种情况下，虽然 *get* 方法已经变得极为简单，很难有错误，但你仍然应该进行这个测试，因为你现在验证的是构造函数是否进行了正确的运算。这时候测试程序认为 *get* 方法是可信的（为什么不呢？），并用它来测试构造函数。两种途径下测试都是有效的。

◆ 讨论

JUnit 社区在讨论到底多简单的方法才不值得测试，关于这一点一直有大量不同的意见。如果你在这一点上还没有自己的标准，那么我们建议你多做一些测试。你可以多读一些书——包括本书的其余部分和其他几本不错的著作，但你还是会发现，如果你不写测试代码，那么你就永远没法培养自己的洞察力，以便做出正确的抉择。如果在这一点上你认为我们说得不对，烦请告知我们！

◆ 相关

- ☐ 2.3—测试构造函数
- ☐ B.1—简单得不能拆分

2.5 Test a setter

2.5 测试设置器

◆ 问题

你想测试你的 *set* 方法，但这种测试似乎过于简单，没有多大的价值。

◆ 背景

JUnit 新手常问的问题是：“我是否应该测试我的 *set* 方法？”没想到这个竟然也是 JUnit 社区争论不休的话题之一。我们坚决认为：基本的设置方法过于简单，很难有错误。但是，如果你还是决定测试它们，那么最好了解什么是有效的诀窍。

◆ 诀窍

最常见的使用 *set* 方法的地方是简单的 JavaBean 中。在这个例子中，bean 对象有点像一个数据包：它包装了一堆数据，这些数据对外可读可写。在这个例子中，每一个 *set* 方法都有一个对应的 *get* 方法，使这个模型可以如此简单地实现：

```
public void testSetProperty() {  
  
    Bean bean = new Bean();  
  
    bean.setProperty(newPropertyValue);  
  
    assertEquals(newPropertyValue, bean.getProperty();)  
  
}
```

斜体部分的代码依具体的测试而定，而 *Bean* 是你使用的 JavaBean 的类名，*Property* 是你测试的属性值。下面是实现步骤的具体描述：

1. 恰当的命名你的测试方法：将 *Property* 改成你要测试的 bean 属性值的名称。
2. 创建一个类的实例。
3. 如果 *newPropertyValue* 是复杂类型，比如另外一个 JavaBean 的索引，就相应地初始化 *newPropertyValue*。
4. 如果 *property* 是一个比字符串更复杂的对象，那么你要保证 *property* 所属的类的 *equals()* 方法已经被恰当地实现。详细内容请参考第 2.1 节“测试你的 *equals* 方法”。

◆ 讨论

如果你没有相应的 *get* 方法，并且不愿仅仅为了测试 *set* 方法而添加它，那么你需要找出 *set* 方法的可观测的副作用，并验证它。

另外一种常见的 *set* 方法应用是在 Command 设计模式中。如果命令被提交给命令解释器以获得执行，那么命令应该有一个相应的 *get* 方法：否则的话，命令解释器如何获取输入的参数呢？另外，如果命令遵从“行为”模式（也就是说提供自己的执行方法），那么命令本身就能完全囊括输入的参数，这样就没有办法直接验证 *set* 方法的行为了。这种情况下，就必须执行该命令（或者行为，如果你非要用更严格的学术定义的话）并分析其副作用，以验证输入的参数是否正确地 *set* 方法使用。

看下面这个简单的、执行银行转账行为的类。你会看到，如果数据被完好地封装，那么测试 *set* 方法就困难了：

```
package junit.cookbook.common;
```

```
import junit.cookbook.util.Bank;
```

```
import junit.cookbook.util.Money;
```

```
public class BankTransferAction {
```

```
    private String sourceAccountId;
```

```
    private String targetAccountId;
```

```
    private Money amount;
```

```
    public void setAmount(Money amount) {
```

```
        this.amount = amount;
```

```
    }
```

```
    public void setSourceAccountId(String sourceAccountId) {
```

```
        this.sourceAccountId = sourceAccountId;
```

```
    }
```

```
    public void setTargetAccountId(String targetAccountId) {
```

```
        this.targetAccountId = targetAccountId;
```

```

    }

    public void execute() {

        Bank bank = Bank.getInstance();

        bank.transfer(sourceAccountId, targetAccountId, amount);

    }

}

```

虽然这个实现中的方法都很简单，但我们不得不使用 `execute()` 方法来测试各种 `set` 方法，因为我们没有其他可以观测的副作用。

注意这个实现严格遵照了 `execute()` 方法的规范：不接受任何参数。一般来讲，程序员执行这个方法是为了让超类或者接口执行当前的 `execute()` 方法。因为这个动作自己获取参数并自动执行，因此我们也不清楚，按照这个规范进行设计到底带来了什么好处，但既然有人认为这是一个好的规范，我们就不再加以评判。为了验证这个类的方法，我们将提交我们自己的 `Bank` 类，以验证传递给 `transfer` 方法的参数。这种办法仅有的问题是，我们提交的是一个“假的”银行对象，而不是一个产品代码中真正使用的银行对象。我们可以选择使用一个新的 `execute()` 方法，并在 `bank` 类中公用一个 `setInstance()` 方法供我们自由地提交 `bank` 对象。两种选择其实都不是很好。尤其当我们使用全局数据的时候，这种不幸的折衷经常出现。我们给出第一种办法的结果，第二种留给读者作为练习。

首先，清单 2.5 给出了一个更开放的行为类。

清单 2.5 一个易于测试的 `BankTransferAction` 类

```

package junit.cookbook.common;

import junit.cookbook.util.Bank;

import junit.cookbook.util.Money;

public class BankTransferAction {

    private String sourceAccountId;

    private String targetAccountId;

    private Money amount;

```

```
public void setAmount(Money amount) {  
    this.amount = amount;  
}  
  
public void setSourceAccountId(String sourceAccountId) {  
    this.sourceAccountId = sourceAccountId;  
}  
  
public void setTargetAccountId(String targetAccountId) {  
    this.targetAccountId = targetAccountId;  
}  
  
public void execute() {  
    execute(Bank.getInstance());  
}  
  
public void execute(Bank bank) {  
    bank.transfer(sourceAccountId, targetAccountId, amount);  
}  
}
```

下面，清单 2.6 给出了验证输入参数的测试。

清单 2.6 BankTransferActionTest 类

```
package junit.cookbook.common.test;

import junit.cookbook.common.BankTransferAction;

import junit.cookbook.util.Bank;

import junit.cookbook.util.Money;

import junit.framework.TestCase;

public class BankTransferActionTest extends TestCase {

    public void testSettingInputParameters() {

        BankTransferAction action = new BankTransferAction();

        action.setSourceAccountId("source");

        action.setTargetAccountId("target");

        action.setAmount(Money.dollars(100));

        action.execute(new Bank() {

            public void transfer(String sourceAccountId,

                                String targetAccountId,

                                Money amount) {

                assertEquals("source", sourceAccountId);

                assertEquals("target", targetAccountId);

                assertEquals(Money.dollars(100), amount);

            }

        });

    }

}
```


}

突然间，需要写 `get` 方法的额外要求似乎显得不那么难以接受了。

我们已经统计了，到底有多少失败的 `set` 方法和诊断这些错误所要投入的时间，发现花这么多时间测试这些方法其实并不值得。我们的建议是，你应该主要去测试那些更容易出错的代码。如果你发现 `set` 方法是最严重的问题，那么不要担心，这说明你的代码写得比目前大多数项目都优秀得多。

◆ 相关

- ☐ 2.1—测试你的 `equals` 方法
- ☐ 2.4—测试获取器
- ☐ B.1—简单得不能拆分

2.6 Test an interface

2.6 测试接口

◆ 问题

你想测试一个接口（`interface`），但接口没有办法初始化。或者你不仅想测试这个接口当前的实现，你想测试“所有可能的实现”。

◆ 背景

写接口的目的一般是，让这个接口的所有实现都具备某个共同的行为。这个行为不仅目前实现的类具备，将来要写的实现也都必须具备。因此，你就需要为这个接口编写一个通用的测试程序，这个测试程序不仅能测试当前已经实现的类的通用属性，而且可以不加修改应用于将来要实现的类。

◆ 诀窍

你应该首先引入一个抽象的测试类，该测试类的方法用于测试接口的共同行为。然后使用工厂方法创建接口的对象，以完成测试程序。下面是详细步骤：

1. 选定测试程序要测试的已经具体实现的类。
2. 创建一个抽象的测试类，声明要验证的功能的测试方法。在具体的测试程序实现中继承这个测试类，并修改相应的实现方法。

3. 在接口的每一个具体实现中都运行该测试程序，但在每个实现中都只验证“接口范围内”的行为。
4. 在测试程序内，找到创建（接口）对象的代码，将该代码改成具体的、已经实现的类的创建方法，但记住将该对象声明为接口的对象，而不是具体实现的类的对象。重复这一过程，直至测试程序中没有已经实现的类的对象。
5. 声明你要在测试中调用的抽象方法。
6. 现在，测试只涉及接口和一些抽象的测试方法，请将测试程序移入抽象的测试类。
7. 重复这一过程直至所有的测试都移入抽象的测试类。
8. 重复前面的全部过程，直至除了验证具体实现的特有的方法的测试程序外，所有的测试代码都已完成。

下面我们通过测试 `java.util.Iterator` 接口来具体说明这种技巧。首先让我们看清单 2.7，这是测试 `java.util.ListIterator` 接口的一个具体实现。

清单 2.7 ListIterator 测试程序

```
package junit.cookbook.test;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

import java.util.NoSuchElementException;

import junit.framework.TestCase;

public class ListIteratorTest extends TestCase {

    private Iterator noMoreElementsIterator;
```

```
protected void setUp() {  
    List empty = new ArrayList();  
    noMoreElementsIterator = empty.iterator();  
}
```

```
public void testHasNextNoMoreElements() {  
    assertFalse(noMoreElementsIterator.hasNext());  
}
```

```
public void testNextNoMoreElements() {  
    try {  
        noMoreElementsIterator.next();  
        fail("No exception with no elements remaining!");  
    }
```

```
        catch (NoSuchElementException  
expected) {  
    }  
}
```

← 见2.8节, “测试是否
抛出正确的异常”

```
public void testRemoveNoMoreElements() {  
    try {  
        noMoreElementsIterator.remove();  
        fail("No exception with no elements remaining!");  
    }
```

```

        }

        catch (IllegalStateException expected) {

        }

    }

}

```

接着让我们引入抽象的 `IteratorTest` 测试类，并将 `ListIteratorTest` 类的实际的实现添加到 `IteratorTest`。结果如下所示：

```

package junit.cookbook.test;

import java.util.Iterator;
import java.util.NoSuchElementException;
import junit.framework.TestCase;

public abstract class IteratorTest extends TestCase {

    private Iterator noMoreElementsIterator;

    protected abstract Iterator makeNoMoreElementsIterator();

    protected void setUp() {

        noMoreElementsIterator = makeNoMoreElementsIterator();

    }

    public void testHasNextNoMoreElements() {

        assertFalse(noMoreElementsIterator.hasNext());
    }
}

```

```
}
```

```
public void testNextNoMoreElements() {  
    try {  
        noMoreElementsIterator.next();  
        fail("No exception with no elements remaining!");  
    }  
    catch (NoSuchElementException expected) {  
    }  
}
```

```
public void testRemoveNoMoreElements() {  
    try {  
        noMoreElementsIterator.remove();  
        fail("No exception with no elements remaining!");  
    }  
    catch (IllegalStateException expected) {  
    }  
}  
}
```

只要我们实现了 `makeNoMoreElementsIterator()` 方法，我们就可以将所有的测试移入 `IteratorTest` 类。我们只需要将这个类封装到 `ListIteratorTest` 类中：

```
package junit.cookbook.test;
```

```
import java.util.ArrayList;
```

```
import java.util.Iterator;
```

```
import java.util.List;
```

```
public class ListIteratorTest extends IteratorTest { //继承 IteratorTest 而不是 TestCase
```

```
    protected Iterator makeNoMoreElementsIterator() { //应该返回合适的 iterator
```

```
        类型
```

```
        List empty = new ArrayList();
```

```
        return empty.iterator();
```

```
    }
```

```
}
```

□ ListIteratorTest 继承我们的抽象类 IteratorTest，而不是直接继承 junit.framework.TestCase 类。

- 我们的 ListIteratorTest 类中实现的创建方法返回一个 iterator 而不是一个空列表。类似地，如果以测试一个基于 Set 类的 iterator，你应该创建一个继承 IteratorTest 的 SetIteratorTest 类，这个类的 makeNoMoreElementsterator() 方法也应该返回相应的 iterator 而不是一个空的 Set 对象。

◆ 讨论

这个抽象的 test case 能正常工作的原因是 Junit 中的测试等级规定。一个 TestCase 类在继承其父类时将同时继承父类所有的测试。在我们的例子中，ListIteratorTest 继承 IteratorTest，所以只要我们在 test runner 中运行 ListIteratorTest，IteratorTest 中的测试都将得到运行。

值得一提的是 Eric Armstrong 的观点，他是 Yahoo! 的 JUnit 社区常客：“一个接口只定义语法，而不指定语义，虽然他们经常被实现。另一方面，一个相关的 test Suite 可以指定语义。我们应该给每一个公用的接口配备一个 test

suite！”当我们在框架中发布一个接口或者抽象类的时候，最好同时写一个相关的抽象的 Test Case，以验证框架在所有客户端中的最重要的期望值。

最后，让我们看看 Java 的联机帮助文档中介绍的 `Iterator.remove()` 方法。这个方法抛出两个异常：`IllegalStateException` 异常表示你没有正确地使用该方法，`UnsupportedOperationException` 异常则表示这个 `Iterator` 的实现不支持删除元素。你在实现 `iterator` 类的时候，在 `remove()` 方法的验证时，应该期望一个 `UnsupportedOperationException`，而不能是 `IllegalStateException`。你还可以做得更好：在抽象的 Test Case 类中添加 `supportsRemove()` 方法，如果 `iterator` 支持 `remove()`，该方法就返回 `true`，否则返回 `false`。（由具体的实现来实现该方法）现在，如清单 2.8 中所示，`remove()` 方法的预期行为将取决于 `supportsRemove()` 的返回值：

清单 2.8 一个更完整的 `IteratorTest` 程序

```
public abstract class MoreCompleteIteratorTest extends TestCase {
```

← 测试下的 iterator
需要支持 `remove()` ?

```
// Other tests as in IteratorTest
```

```
    protected abstract boolean supportsRemove();
```

```
    public void testRemoveNoMoreElements() {
```

```
        try {
```

```
            noMoreElementsIterator.remove();
```

```
            if (supportsRemove()) {
```

```
                fail("No exception with no elements remaining!");
```

```
            } else {
```

```
                fail("No exception when attempting to remove!");
```

```
            }
```

← 假定不支持 `remove`

```
        }
```

```
    catch (IllegalStateException expected1) {
```


2.7 测试 JavaBean

◆ 问题

你要测试一个 JavaBean，但你写的测试似乎既千篇一律又不可靠。

◆ 背景

因为 JavaBean 跟一堆 *get* 和 *set* 方法差不多，所以你在为 bean 编写的测试代码看起来重复性很大。这肯定无法让你感到满意，其实有很多人也都这么认为。

◆ 诀窍

在多数情况下，测试 JavaBean 没什么特别的。

- bean 的一个属性就是直接与域内的数据进行交互而且不进行计算，因此测试相应的 *set* 和 *get* 方法是没多大用的。不要浪费时间做这件事情。
- 对于需要计算的属性值，写简单的测试程序验证属性值是否被正确的计算。
- 对于 bean 方法，没有什么特殊的要做；因此，将它的方法当作普通的简单方法来测试就可以了。
- 对于 bean 的事件方法，比如属性改变的事件，请参考 14.2 节中介绍的技巧，将 bean 当作一个事件源，因为它确实也就是！

在 JavaBeans 方面惟一的测试技巧是正常性测试。JavaBeans 规范要求无参数的构造函数，然而大多数的 bean 都要求属性值非空。这就意味着：当构造函数完成以后，得到的对象其实并没有被完全初始化。我们称之为“不正常对象”，因为如果你这时候使用它的任何方法，你都无法确定你将获得什么结果。

为了实现正常性测试，请引入一个名为 *isValid()* 的方法，用它来确认必要的属性值是否确实为非空。你的测试使用的 *isValid()* 方法，实际上表明了哪些属性值是必要的，哪些是可选的。就像有人说的一样，如果 `java.util.Calendar` 也提供了这种方法就太好了！

在一个简单的 JavaBean 类中，如果属性值直接映射为类中的一个域，那么一般来讲你无须为 *set* 方法编写测试，因为它们过于简单而很难出错。但是，如下所示的 JavaBean 测试程序仍然是值得写的。在这个例子中，我们使用 *Command* 对象：

```
public void testBankTransferCommandIsValid() {  
  
    BankTransferCommand command = new BankTransferCommand();
```

```

        command.setSourceAccountId("123-456A");

        command.setTargetAccountId("987-654B");

        command.setAmount(Money.dollars(1000));

        assertTrue(command.isReadyToExecute());
    }

```

这个测试向读它的程序员表明了一些重要的事情：如果程序已经设置了 account ID， target account ID 和要转账的金额，那就说明 command 已经准备好，可以执行了。在这里，为了更好地表明意图，我们将 `isValid()` 改成了 `isReadyToExecute()`。这个测试验证了是否已经为转账准备了充足的条件，这个命令是否已经有效、是否可以被执行了。为了完整起见，我们进行如下测试：

```

public void testNeedsAmount() {

    BankTransferCommand command = new BankTransferCommand();

    command.setSourceAccountId("123-456A");

    command.setTargetAccountId("987-654B");

    // 不要设置数量

    assertFalse(command.isReadyToExecute());
}

```

这个测试说明，如果没有提供转账的数额，命令就不能执行。虽然没有注释，但它已经说明了哪个属性值还没有被设置。其实这种测试很少需要写注释，因为代码本身一般就说明问题了。但是，为了说明代码中需要补充的东西，注释还是必要的。

◆ 讨论

使用这种技巧时，你应该为每种需要测试的属性都提供一个测试。每项属性都验证一下：如果这项属性没有被设置，其对象就不可用。因为这里有三项属性我们就应该有四个测试（一个验证可用，三个验证不可用）。

这个技巧描述了如何按需进行测试。在这种建议下，记住不管什么对象，只要你不能保证它的正确性，就应该测试一下。我们只是告诉你什么测试可以忽略但如果你还是觉得不放心，那就去测试一下。不要把我们的建议当作准则，那只是指导方针而已。要记住这个方针本身的深层准则是：测试到你的担心变成厌烦为止。如果你担心你的 JavaBean 不能正常工作，你可以将所有东西都纳入测试范围，直到你找到足够的理由来忽略一些。最后你会降低自己的标准，这时候缺陷就会在你的测试没有发觉的情况下不知不觉的被引入。结果，你就不得不写更多的测试，直到你找回自己的信心。这个循环会周而复始，永无穷尽。

◆ 相关

- 2.4—测试获取器
- 2.5—测试设置器
- 14.2—测试可观测的数据源
- B.1—简单得不能拆分

2.8 Test throwing the right exception

2.8 测试是否抛出正确的异常

◆ 问题

你想验证一个方法是否在某种特定的情况下抛出期望的异常，也许你正在找“最简单的办法”来写这种测试。

◆ 背景

要想知道如何实现这种测试，你需要了解 JUnit 如何判定一个测试是通过还是失败。如果一个断言失败或者抛出一个异常的时候，测试就会失败（可能是 failure 也可能是 error）；否则，测试就会通过。换句话说，如果一个测试全部走完，就是说程序从头运行到了尾，而没有从中间跳出，那么它就通过了。知道了这些，就足够你推断出如何写这种测试了：如果该抛出异常的代码段没有抛出异常，那么这个测试就应该失败；测试只能捕捉期望的异常；任何其他的异常都应该由 JUnit 框架捕捉。

◆ 诀窍

下面的代码展示了，如何写这种验证是否抛出正确异常的测试示例：

```
public void testConstructorDiesWithNull() throws Exception {  
  
    try {  
  
        Fraction oneOverZero = new Fraction(1, 0);  
  
        fail("Created fraction 1/0! That's undefined!");  
  
    }  
  
    catch (IllegalArgumentException expected) {  
  
        assertEquals("denominator", expected.getMessage());  
  
    }  
  
}
```

你已经看过了示例，下面让我们解释一下：

1. 找到可能抛出异常的代码段，将它放入一个 try 语句内。
2. 调用了应该抛出异常的方法以后，在 try 语句内写一个 fail()方法来说明：“如果运行到了这里，那么说明期望的异常没有被抛出。”
3. 添加一个 catch 语句以捕获期望的异常。
4. 在 catch 语句内，如果需要的话，验证捕获的异常的属性与你期望的相同。
5. 声明该测试方法会抛出异常，这可以让代码适应性更强。有人可能会在测试程序外声明这个方法可抛出其他的异常。这种变化不应该影响你的测试，因此它不应该导致你的测试不能被编译。

◆ 讨论

如果测试的方法抛出其他的异常——不同于你要捕获的异常——那么 JUnit 将报告一个 error，而不是 failure，因为测试方法将一个异常抛给了 JUnit 框架。记住这样的 error 一般是环境或者测试程序自身的错误，而不是产品代码的问题。如果产品代码抛出了一个预期之外的异常，那么一般可能是潜在的问题阻碍了测试的正常运行。

回到前面的例子，注意我们使用的异常的名字：`expected`。这很重要，这个信息对要抛出（或者捕获，由你怎么看）这个异常的程序员来说，很容易看懂，这很好。因为异常是不正常代码的执行路径，我们已经习惯了将异常看作不好的事情。而这种测试就是为了验证异常。

许多情况下都需要为期望的异常设置断言。只要你确认了捕捉的异常类型相符，那就足以说明你的代码是正确的，你不需要在异常处理部分写任何的代码。这种情况下，有些程序员喜欢添加注释说明这是“正确的路径”。我们觉得将异常对象命名为“`expected`”就能有效地说明问题，而不需要添加注释。这是个人的习惯问题，你可以照自己的习惯来写。

另一个备选的方案是捕获所有可能抛出的异常，然后为所有期望之外的异常添加一个 `failure` 声明。如果一个程序员这样做，就会将一个期望之外的异常报成 `failure`，而不是 `error`。基于下面两个简单的原因，我们不推荐这样做：

- 与简单地将异常抛给 JUnit 框架相比，这样做需要输入更多的代码。它们的效果是一样的，但更少的代码更容易维护。
- 将预期之外的异常报为 `failure` 没有额外的好处。多数情况下，它们两个是没有区别的。有些预期之外的异常是产品代码的问题，有些是环境配置问题。如果不依靠底层的实现细节，一般很难区分它们。

一个更面向对象的办法

就在这个主题似乎可以结束的时候，JUnit 社区一位很有声望的成员 Ilja Preuß (IL-ya PROYSS) 给出了一个更面向对象的建议代码：

```
public void testForException() {  
  
    assertThrows(MyException.class, new ExceptionalClosure() {  
  
        public Object execute(Object input) throws Exception {  
  
            return doSomethingThatShouldThrowMyException();  
  
        }  
  
    });  
  
}
```

虽然有人觉得 Java 的匿名内部类不太好读，但这个方法的意图再明显不过了：“测试这段代码是否抛出期望的异常。”可以按如下的方式实现 `assertThrows()` 方法：

```
public static void assertThrows(  

```

Class expectedExceptionClass,

ExceptionalClosure closure) {

String expectedExceptionClassName

= expectedExceptionClass.getName();

try {

closure.execute(null);

fail(

"Block did not throw an exception of type "

```

        + expectedExceptionClassName);
    }
    catch (Exception e) {
        assertTrue(
            "Caught exception of type <"
                + e.getClass().getName()
                    + ">, expected
one of type <"
                + expectedExceptionClassName
                    + ">",
            expectedExceptionClass.isInstance(e));
    }
}

```

迫使做这个

细节错误信息

验证得到异常的类

我们捕捉了所有的异常，而不仅是期望的异常，这是因为当编译的时候，我们还不知道代码会抛出什么样的异常。如果这样设置断言，那么错误信息就很重要，因为你取走了错误信息的控制权。另外一个可选的方案是，在 `assertThrows()` 方法中添加另外一个参数，用来接收自定义的异常。最后，因为我们必须测试所有的异常，我们必须将捕获的异常与我们的期望值进行比较。`Class.isInstance(Object)` 方法说明我们捕获的异常是否是期望的异常的实例，这与使用 `instanceof()` 方法是一样的。

除了意图明确，这种办法还避免了在大量的测试中重复输入异常检测代码。这是很大的进步！

也许你已经注意到了代码中使用了 `ExceptionalClosure`，我们就此作一些说明。`Closure` 只是一段代码的封装，某些语言，像 `Smalltalk` 和 `Ruby`，直接支持这种语法。为了避免“制造相同的轮子”，我们尽可能地使用 `Jakarta Commons` 编写的 `Closure` 接口（<http://jakarta.apache.org>）。不幸的是，在这里我们无法使用

它，因为它的 `execute()` 方法没有声明可以抛出未知的异常，因此只能抛出已知异常，实际上是运行时异常的派生。为了补偿这个缺陷，我们向 Diasparsoft 工具包中添加了一个 `ExceptionalClosure` 接口，它具备 `Closure` 的功能，但它的 `execute()` 方法可以抛出任何异常。

一旦你实现了自己的 `assertThrows()` 方法，你就可以将它放入自己定义的 `assertion` 超类，并随时可以使用它。关于自定义断言的讨论，请参看 17.4 节。

注意你的断言

如果你的断言与特定的异常相关，那么你要小心：如果验证异常对象的结构过于紧凑，那么可能导致测试与产品代码耦合得过强。这时候，测试的结果可能有点不太可靠。假设异常信息是给终端用户看的，而你要写直接验证该消息的断言。一般来讲，你会照如下的方式编写测试代码：

```
public void testNameNotEntered() {  
  
    try {  
  
        login("", "password");  
  
        fail("User logged in without a user name!");  
  
    }  
  
    catch (MissingEntryException expected) {  
  
        assertEquals("userName", expected.getEntryName());  
  
        assertEquals(  
  
            "Please enter a user name and try again.",  
  
            expected.getMessage());  
  
    }  
  
}
```

测试的代码很清楚：如果一个用户在不输入用户名的情况下登录，那么登录模块就抛出一个 `MissingEntryException`，这个异常包含了登录所缺少的必要项的名称，以及提供给终端用户的信息。这看起来很好，因为这个异常对象包含的信息非常简单，它包含的数据像终端用户读到的信息一样易于理解。虽然 `catch` 语句中的第一个断言写得不错，但我们对第二个有不同的意见。虽然 `userName` 是程序的内部名称，并且是行为的一部分，但给终端用户看的信息可以在不影响登录功能的

前提下随时更改。换句话说，如果 `userName` 改变的话，测试程序就需要随之改变。然而，测试似乎不应该随着终端用户信息的改变而改变。由于测试是现在写的，将来任何属性值的改变，都会要求测试程序随之改变。

这个例子中，我们建议去除第二个断言而仅保留第一个，因为设计的一般准则是：将给终端用户看的信息与内部对象的行为相分离。你可以经常简单地初始化一个异常对象，然后检查它的 `toString()` 以及 `getMessage()` 方法的返回值。但是，我们没有必要仅仅为了验证异常类这方面的行为去创建异常。

◆ 相关

- 17.4—提取定制的断言
- Jeff Langr 和 Essential Java Style 写的“*Patterns for Implementation*”，Prentice Hall 出版社，1999 年。
- Jakarta Commons 项目（<http://jakarta.apache.org>）

2.9 Let collections compare themselves

2.9 让容器自己进行比较

◆ 问题

你想验证容器的内容，而你第一个想到的办法是逐个检验你期望的项。是否有更简单的方法呢？

◆ 背景

如果你是一个 JUnit 新手，那么可能你也是 Java 新手。我们最近经常看到一些高校已经在他们的 Java 课程中引入了 JUnit。我们坚定地认为，通过写测试来学习一门编程语言是可喜的进步。这是因为，一门新语言的一些基础方面是很奇怪的，比如要将输出打印到控制台或者执行一个程序的入口点。比如，在 Smalltalk 和 Microsoft Foundation Classes（MFC）中做这些事情一点都不简单，但不管你使用什么语言，它们的 xUnit test runners 都是一样的。如果你是一个 Java 新手，你可能不熟悉如何让对象自己进行相互比较，以至于你觉得逐个比较容器的条目是惟一的选择。其实，我们有更简单的办法。

◆ 诀窍

首先使用你期望的内容创建一个列表，你可以使用你期望的顺序，接着使用 `assertEquals()` 方法将其与你获取的容器（最常见的是通过一个方法）进行比较，最

后就让相应的 equals()方法来验证容器是否相等。我们在本章的开头介绍了这一技巧，现在我们就使用它来帮助你实现测试代码。我们总结了各种容器类的 equals()方法的特点，请参考表 2.1。

表 2.1 各种类型的容器的 Equals()方法的行为

容器的种类	Equals()方法的行为
List	如果两个 list 对象的元素全部相同，并且索引全部对应，就认为它们“相等”，而不论比较的对象是否是 list 接口的不同实现。比如，如果一个 ArrayList 对象和一个 LinkedList 对象满足这些条件，那么就认为它们“相等”。
Set	如果两个 set 对象包含完全相同的元素，就认为它们“相等”，不论它们是否是 Set 接口的不同实现。比如，如果一个 HashSet 对象和一个 LinkedList 对象包含完全相同的元素，那么就认为它们是相等的。
Map	如果两个 Map 对象包含完全相同的关键字，并且每个关键字都映射相同的元素，就认为它们“相等”，不论它们是否是 Map 接口的不同实现。比如，如果一个 HashMap 对象和一个 TreeMap 对象满足这些条件，那么就认为它们是相等的。
Collection	如果两个容器对象属于同一种类型（List，Set，Map），并且满足相应的类型的相等条件，才认为它们“相等”。即使一个 List 对象和一个 Set 对象包含完全相等的元素，也不认为它们是相等的。

只要你比较的容器“属于同一种类型”，比较它们就很简单。示例代码请参考本章的介绍，那里有使用 assertEquals()方法比较 List 对象的例子。

◆ 讨论

如果你有两个容器对象都是 List 类型，但你想在不考虑它们的元素顺序的前提下进行比较，那么你有几个选择，这取决于 List 对象的特征。如果你知道 List 对象没有重复项，那么可以简单地将 List 对象转换为 Set 对象再进行比较。这只需要使用 assertEquals- (new HashSet (expectedList),new HashSet(actualList))方法，就能够实现。

然而，如果要比较的 List 对象含有重复项，你就需要首先获取每个重复项的重复次数，并确认它们是否相同。你可以创建自定义的断言方法，或者使用 GSBASE 项目的 BaseTestCase，它含有 assertEquals()方法。这个方法将两个容器对象当作“未排序”的 List 对象进行比较：如果两个容器对象含有每个元素的相同数量的拷贝，那么即使它们的顺序不同，也认为它们是相等的。GSBASE 可以高效地比较两个容器的无序元素。这个方法非常完美，但如果你想实现一个更面向对象的方法，你需要一个封装了无序 List 的容器。

我们一般将这种容器称为 Bag。Bag 是未排序的元素容器，并且允许元素有多个拷贝同时存在。也有人将这种容器称为“Multiset”。如果两个 Bag 对象的每个元素的拷贝数量都相等，那么就认为它们是相等的。因此，它的 equals()方法应该按照这个要求进行实现。你可以自己实现这个容器类，或者在网

上搜索已有的实现。你也可以将 GSBASE 的 `assertCollectionsEqual()` 方法看作 Bag 类（或者接口）的 `equals()` 方法。之所以 GSBASE 项目没有提供一个 Bag 类的完全实现，很可能是因为它的创建者 Mike Bowler 还不需要它。如果你实现了，可以考虑给 GSBASE 项目添加一个补丁——但首先请确定你的补丁通过了 JUnit 的完全测试！

◆ 相关

- GSBASE 项目 (<http://gsbase.sourceforge.net>)

2.10 Test a big object for equality

2.10 测试一个巨型对象的相等性

◆ 问题

你有一个包含许多（比如说超过 6 个）关键属性的值对象（Value Object），你使用 `EqualsTester` 或者 `EqualsHashCodeTestCase` 来写测试代码，但这个测试似乎并不恰当。

◆ 背景

GSBASE 的 `EqualsTester`（参考 2.1 节）接受四个参数：前两个是不同的但是相等的对象，第三个与第一个相等，而第四个是第一个对象的子类的对象，因此跟第一个对象不会相等。虽然这个方法对大多数应用都已够用，但在这里你可能需要一个更全面的测试，你需要测试 $n+3$ 个对象：前两个是相等的对象， n 个与前两个不相等，最后一个子类的对象。这里的 n 就是你的数值对象的关键属性值的个数。JUnit-addons 中的 `EqualsHashCodeTestCase` 方法也存在同样的问题，因为它只处理两个不相等的数值对象。

◆ 诀窍

看起来我们需要总结一下，如何与任意数量的、与“基准对象”不相等的对象进行相等性测试。我们已经将实现这个测试功能的类（`ValueObjectEqualsTest`）添加到了 Diasparsoft Toolkit 中，这个类的核心算法抄袭自 JUnit-addons，当然，我事先获得了授权。简单的讲，这个类可以测试数值对象中任何可能存在差异的地方。首先，让我们来看看如何使用 `ValueObjectEqualsTest` 类。下面的清单 2.9 就是一个例子——尽管是一个抽象而且没有意义的例子。在继承了 `ValueObjectEqualsTest` 之后，你需要实现三个方法。

清单 2.9 使用 `ValueObjectEqualsTest` 测试五个关键属性值对象

```
package com.diasparsoftware.java.lang.test;
```

```
import java.util.*;
```

```
import com.diasparsoftware.util.junit.ValueObjectEqualsTest;
```

```
public class ValueObjectEqualsTestFivePropertiesTest
```

```
    extends ValueObjectEqualsTest {
```

← 测试下的iteretor

```
    protected List keyPropertyNames() {
```

```
        return Arrays.asList(
```

```
            new String[] { "key1", "key2", "key3", "key4", "key5" });
```

```
    }
```

```
    protected Object createControllInstance() throws Exception {
```

```
        return new FiveKeys(1, 2, 3, 4, 5);
```

```
    }
```

```
    protected Object createInstanceDiffersIn(String keyPropertyName)
```

← 在码属性值的控制中每个对象都是不同的

```
        throws Exception {
```

```
            if ("key1".equals(keyPropertyName))
```

```

        return new FiveKeys(6, 2, 3, 4, 5);

    else if ("key2".equals(keyPropertyName))

        return new FiveKeys(1, 6, 3, 4, 5);

    else if ("key3".equals(keyPropertyName))

        return new FiveKeys(1, 2, 6, 4, 5);

    else if ("key4".equals(keyPropertyName))

        return new FiveKeys(1, 2, 3, 6, 5);

    else if ("key5".equals(keyPropertyName))

        return new FiveKeys(1, 2, 3, 4, 6);

    return null;

}

}

```

每个数值对象都由一系列的关键属性值来定义，如果属性的值不等，那么数值对象就不相等。一般情况下，数值对象的所有属性值都是它的关键属性值，但这不绝对。为了给相等性测试提供这些属性值，你需要首先实现 `keyPropertyNames()` 方法，以返回属性值名称的列表。返回的列表的排列顺序并不重要，我们在例子中将其按字母排序。

像在 `EqualsHashCodeTestCase` 中一样，你需要定义一个“基准”对象，而所有其他对象将与这个“基准”对象进行比较。你需要实现的是 `createInstance()` 方法，它每次都返回一个新的对象。你可以随意决定每次返回什么对象，但这个选择将影响你实现其余方法的方式。我们决定放回 1、2、3、4 和 5，作为我们的基准对象。

像在 `EqualsHashCodeTestCase` 中一样，你需要实现的最后一个方法是：返回与基准对象不同的对象的方法。当你实现 `createInstanceDiffersBy(String keyPropertyName)` 方法时，你必须能返回一个与基准对象的指定属性值不同的对象。在我们的例子中，当你要求返回一个 `key1` 属性值不同的对象时，我们返回了 6、2、3、4、5，它的第一个属性值（就是 `key1`，明白么？）与基准对象的第一个属性值不同。希望你看懂了这个例子。

◆ 讨论

如果你希望你的相等性测试不仅能验证几个简单的例子，你可以自己创建一个参数化的 Test Case（请参考 4.8 节“创建数据驱动的 test suite”），在这里每个测试都接受三个参数：两个对象，一个说明两个对象是否应该相等的布尔值。这个比较测试使用称为“示例证明”的办法定义自己的 equals() 方法，因为你完全通过例子来定义 equals() 方法的行为：每个例子都说明，“这个对象与那个相等，或者这个对象与那个不相等”。如果你提供了足够的例子，你最终会找到仅存的合格的 equals() 方法，这些方法似乎对任何对象都适用。一旦你的 equals() 方法返回了错误的结果，就说明找到了能制造缺陷的对象，你就可以添加这个对象以阻止同类错误再度发生。

◆ 相关

- 2.1—测试你的 equals 方法
- 4.8—创建数据驱动的 Test Suite

2.11 Test an object that instantiates other objects

2.11 测试一个拥有其他对象的对象

◆ 问题

你想测试一个对象，但这个对象内部还初始化其他对象，这使得测试变得困难或者代价高昂。

◆ 背景

面向对象的设计是双刃剑。我们使用聚合来表明一个对象拥有另外一个对象比如大多数情况下一个汽车都有自己的轮子。而另一方面，为了单独测试一个对象，我们需要将对象像拼图游戏一样拼凑起来。这就是说，测试更期望对象使用组合而不是聚合。如果你要测试的对象还初始化了其他的对象，那么只有在保证内部对象的正确性的前提下，你才能测试这个对象，而这就违反了单独测试一个对象的原则。

我们不安地发现，仍然有大量的程序员对进行测试的好处一无所知。这致使他们过多地使用了聚合。他们的设计中充满了初始化了其他对象的对象，或者常常从全局的位置获取对象。这种编程办法，如果不改进的话，就会造就强耦合的架构，导致测试难以进行。我们知道：我们继承了大量这种设计，甚至直接创建了它们。下面的诀窍介绍了一个基本的测试技巧，它可以带来程序设计上的一些提高，使单独的测试一个对象成为可能。

◆ 诀窍

为了处理这个问题，你要将要测试的对象的内部对象用其他的对象代替。这就有两个问题需要你解决：

□ 你如何创建这个对象所包含的对象的替身？

□ 你如何将它传递给要测试的对象？

为了简化讨论，我们使用术语“测试对象”（不要与对象测试相混淆），用来指代那些你要测试的类或者接口的实例。有多种不同的测试对象：假的、残缺的、模拟的，我们将在附录 B 中详细讨论其中的差别。

创建一个接口的测试对象很简单：只要创建一个类，并用最简单的方法实现这个接口就可以了，这是最简单的测试对象。在整本书中，我们都使用 EasyMock (www.easymock.org) 来创建接口的测试对象。之所以使用这个包，是因为它不仅可以避免我们重复的劳动，还能保证我们仿造的接口对象的一致性和统一性。让我们的测试程序更易于理解——至少对熟悉 EasyMock 的人来说是这样。你可以在第二部分的 J2EE 接口部分发现大量的使用 EasyMock 的例子。

创建一个类的测试对象，可以创建它的一个子类，然后仿造它的全部方法，或者干脆屏蔽它的全部方法。一个仿造的方法返回某种可预见的、有意义的、硬编码的值，而一个屏蔽的方法不做任何有意义的事情，只需要实现编译要求即可。你也许会发现：在要测试的对象中声明接口，然后在测试时转变为具体类的对象是很有好处的。因为这样你就可以使用前面介绍的 EasyMock 了。此外，如果你在对象中要使用其他的对象，最好将其声明为接口，这样你的设计就更有弹性（也更容易测试）。

至于第二个问题，我们有两种方法可以让你将一个测试对象传递给另一个要测试的对象：一种是改造构造函数，另一种是添加一个 set 方法。我们认为改造构造函数的方法比较简单，这样可以避免在测试中再去调用 set 方法。为了举例说明这个技巧，我们使用 J. B. 的文章“Use Your Singletons Wisely”中的例子。其中，Deployment 使用 Deployer 在文件中部署对象。在例子中，只需要一个 Deployer，因此很容易设计成 Singleton。（关于更多测试以及 Singleton 的例子，请参考 14.3 节的“测试一个 Singleton”）。Deployment.deploy() 方法实现如下：

```
public class Deployment {  
  
    public void deploy(File targetFile) throws FileNotFoundException {  
  
        Deployer.getInstance().deploy(this, targetFile);  
  
    }  
  
}
```

注意，Deployment 使用类级别的方法 `Deployer.getInstance()` 来生成 `Deployer`。如果你想仿造一个 `Deployer`，你需要将一个 `Deployer` 通过某种方法传递给 `Deployment`。我们推荐使用构造函数来传递，所以新加了一个构造函数并创建了一个实例来保存 `Deployer`：

```
public class Deployment {

    private Deployer deployer;

    public Deployment(Deployer deployer) {

        this.deployer = deployer;

    }

    public void deploy(File targetFile) throws FileNotFoundException {

        deployer.deploy(this, targetFile);

    }

}
```

等一下！`Deployer.getInstance()` 方法哪里去了？我们不能丢掉这段代码：现在我们将删除了无参数的构造函数，那么我们就需要添加一个新的构造函数以保证提供默认的单例 `Deployer`：

```
public class Deployment {

    private Deployer deployer;

    public Deployment() {

        this(Deployer.getInstance());

    }

    public Deployment(Deployer deployer) {
```



```

        this.deployer = deployer;
    }

    public void deploy(File targetFile) {

        deployer.deploy(this, targetFile);

    }
}

```

现在，当产品代码使用无参构造函数创建一个 `Deployment` 时，就可以观察到期望的行为了：`Deployment` 将使用 `Singleton Deployer`。不过，在我们的测试中，可以使用一个假的 `Deployer`，这样做可以模拟诸如目标文件不存在等情况。下面的代码是一个“故障测试”的 `Deployer`——它永远都认为目标文件不存在：

```

public class FileNotFoundDeployer extends Deployer {

    public void deploy(Deployment deployment, File targetFile)

        throws FileNotFoundException {

        throw new FileNotFoundException(targetFile.getPath());

    }

}

```

现在，我们可以测试当 `Deployer` 部署失败的时候，`Deployment` 如何表现。我们使用在 2.8 节“测试是否抛出正确的异常”中介绍的技巧：

```

public void testTargetFileNotFound() throws Exception {

    Deployer fileNotFoundDeployer = new FileNotFoundDeployer();

    Deployment deployment = new Deployment(fileNotFoundDeployer);

    try {

```

```

        deployment.deploy(new File("hello"));

        fail("Found target file?!");
    }

    catch (FileNotFoundException expected) {

        assertEquals("hello", expected.getMessage());
    }
}

```

这个测试演示了如何替换一个对象中引用的测试对象，同时也介绍了如何通过继承仿制测试对象。我们将要使用的对象变成了构造函数的一个可选参数：如果我们没有提供的话，那么类也会自动察觉，并创建一个默认的。我们整本书以及我们的实际工作中都使用这个技巧和替代方法。

◆ 讨论

我们使用了一个测试对象来模拟没有办法找到目标文件的情况，而不是想办法重现这种情况，因为重现需要我们真的去在文件系统中寻找一个不存在的文件。我们强烈推荐您模拟这种场景，因为重现很容易出错。如果你在 Windows 中指定一个不存在的文件，但别人在 Unix 中执行这个测试将会怎样？在 Unix 中，你的文件名甚至根本不是一个有效的文件名。更糟糕的是，如果你设定的“不存在”文件正好在别人的机器上存在呢？当然，你可以使用 JVM 来寻找本机的缓存路径，但总的来讲模拟错误比重现错误要简单。

假冒一个类级别或者全局的数据和方法比较困难，因为你不能通过继承来重载一个类级别的方法；即使你能，使用类级别方法的代码硬编码了类的名称，不允许你通过继承来重载。关于类级别的方法，以及如何在应用中克服其缺点，在 JUnit 和“测试驱动开发方法”社区中有大量的讨论。他们的结论是，最好将类级别的方法移到新的类中，并将它们变成实例级别，以减少其使用，至少当这样做有意义时。有好几种复制策略，包括将类级别的方法隐藏在接口之后，让这个方 法看起来像是实例级别。这就需要大量的机械式的转换，一般都没有事先通过全面的测试检查！幸运的是，Chad Woolley 已经开始创建一个工具包，并承诺使假冒类级别方法成为可能。他的工具包称为 Virtual Mock

(www.virtualmock.org/)，声称可以提供安全转换的简单的方法——尤其是对那些大量使用类级别方法和数据的设计而言。虽然当我们这么说的时候这个项目还处于 alpha 阶段，但 Chad 的工作很出色，因此，如果你下次获得了这样的一个设计，我们建议你 将 Virtual Mock 装备到你的武器库中。换句话说，我们强烈推荐使 用 Virtual Mock 来建立安全转换，但不要使用它来掩盖设计上的臭味，而是要彻底清除它。

◆ 相关

- 2.8—测试是否抛出正确的异常
- 14.3—测试一个 Singleton
- B.4—仿制对象概览
- 虚拟模拟项目 (www.virtualmock.org)

本章主要内容：

- 你的 JDBC 客户端代码的哪些内容不需要测试
- 测试领域对象与 ResultSet 间的映射
- 使用 Gold Master 技术来验证 SQL 语句
- 使用数据库的 meta data 来编写简单的测试
- 管理不同数据库中的测试数据和使用 DbUnit
- 测试遗留的 JDBC 代码

有人说数据库是任何 Java 应用的核心。系统需要数据，许多人设计系统都是围绕着数据库来进行的，似乎数据库是一切的中心。这样就导致了应用与源数据之间的高耦合，而我们常说的高耦合与高重复是程序员测试失败的两个主要原因。在这一章的内容中，我们将会介绍那些用于测试数据组件的策略与技术，以及将你的应用重构为更容易测试的设计的方法。

测试 Java 的数据库代码最头痛的问题之一就是 Java 没有一个独立的、成熟的 SQL 解析器。如果一定要找，也许 HSQLDB 和 Mimer 可以胜任。HSQLDB (hsqldb.sourceforge.net) 是一个纯 Java 的数据库平台，所以我们认为可以直接使用它的解析器。在行动之前，我们查看了 HSQLDB 的源代码来了解它是如何解析 SQL 语句的。但这个解析器与其数据库以及 SQL 查询器的联系是如此的紧密，仅仅靠它是不能够解析 SQL 语句的。这不是在诋毁 HSQLDB 或者它的作者——也许他们根本就不需要一个独立的解析器。如果我们能够从其中独立出这个功能将会很棒。

Upright Database Technology 基于其 Mimer (www.mimer.com) 数据库引擎提供了一个联机的 SQL 查询验证器；但在本书出版时，还仅仅提供了 Web Service 的版本，而我们真正想要的是可嵌入的模式。在 Google 中搜索数小时后，我们比较不负责任的下了如此结论：在我们编写此部分内容时，还没有可以用来验证 SQL 语句的独立的解析器。在我们想要的 SQL 解析器或者验证器出现之前，我们将使用一种替代的方法。

测试驱动的程序已经针对测试数据库做了很多工作。它通常是程序员进行的第一个复杂的测试。你可以在 Richard Dallaway 的 *Unit Testing Database Code* 中找到一些优秀的指导性的规则。这篇文章的大纲中提到“你需要 4 个数据库”，现在我们引用在这里：

1. 产品数据库中有真实数据，不需要进行测试。
2. 你大部分的测试是在你本地的开发用的数据库中执行的。
3. 公用的开发数据库，会被所有的开发者共享，因为你可以检验你的代码在真实数据下能否顺利工作。而不是在你的本地的测试用的数据库中的少量数据。你可能并不需要如此，但这样的结果是可以验证你的应用能够在大量数据下工作。
4. 无论是外部的数据库还是继承的数据库，测试都应当优先于部署以确保任何本地数据库的更改的有效性。如果你是一个人单独工作，这一点你可以不去考虑但你必须确保任何数据库的结果以及存储过程都被移植到了产品数据库中了。

本章中的实例可以划分为两个类别：如何编写针对你的数据库组件以及它们的客户端的对象的测试，以及如何对你的数据访问层进行单元测试。前一个类别的实例对于那些将要建立数据库组件或者需要重构现有数据库组件使其更容易测试的程序员很有帮助。而后一个类别的实例是为那些拥有继承于不可修改的

数据库组件的代码或者项目中不允许对数据库组件进行重构的程序员准备的。（即使 Martin Fowler 自己在其 *Refactoring: Improving the Design of Existing Code* 中也讲述到了一些不应当进行重构的情况）。

我们先前有一些建议，许多刚刚接触到 JUnit 的程序员选择他们的数据访问测试来作为他们尝试的第一个复杂的测试。他们为他们的每一个 JDBC 调用都编写测试：每个创建、查询以及删除。没有多久他们就建立了一系列重复和相互依赖的测试。对于每一张表，他们陷入了下面的这种模式：

1. 首先直接在数据库中添加一条记录。
2. 验证你的 DAO 能够查询出这条记录。
3. 通过你的 DAO 创建一条记录。
4. 打开数据库查看用 DAO 添加记录是否成功。

这其实是一个简单的测试！对每一张表都像这样做是重复劳动，而且这个测试的第二步还要依赖于第一步的顺利通过。请注意这个测试不仅仅验证了你的代码，同时也验证了数据库提供商关于 JDBC 的实现。你应当集中精力来测试你自己写的代码，而不是去测试平台。

我们以一个 DELETE 语句来说明这一点。在我们的 Coffee Shop 应用中，我们在一个名称为 catalog.discount 的表中保存用于表示商品的折扣价格与促销价格的信息。根据实际情况，市场部门会提前给出新的针对最近一段有限时间内的折扣价格与促销价格。一旦所提供的信息过期，我们希望能够将它从 vcatalog.discount 表中删除。当然我们也希望如果出现了什么例外的情况——比方说，由于某种原因，我们提供的咖啡豆的价格变成了 \$ 1/千克——我们可以中止继续提供这样的商品信息。我们需要一个简单的工具，它能够删除那些已经过期的信息。最后，我们还希望对这个工具中数据访问的部分做一些测试，比如下面的“happy path”测试。这个测试假设了在一个空的 vcatalog.discount 表中删除所有的在 2003 年 1 月 1 日过期的折扣，然后再次搜索 2003 年 1 月 1 日过期的折扣，这时返回的结果应该是空的。

```
public class DeleteDiscountsTest extends CoffeeShopDatabaseFixture {

    // other code omitted

    public void testDiscountsExpiredThirtyDaysAgo() throws Exception {

        // FIXTURE: getDataSource() returns our test data source

        // FIXTURE: Table cleanup occurs in setUp() and tearDown()

        DiscountStore discountStore =

            new DiscountStore(getDataSource());
```

```

        Date expiryDate = DateUtil.makeDate(2003, 1, 1);

        discountStore.removeExpiredDiscountAsOf(expiryDate);

        Collection expiresByDate =

            discountStore.findExpiresNoLaterThan(expiryDate);

        assertTrue(expiresByDate.isEmpty());

    }
}

```

我们希望 DiscountStore 类使用 JDBC 与数据库交互，这样，这个类将包含如下的一些行为：

- ☐ DiscountStore 将 Value Object 中的属性转换为 PreparedStatement 中的参数以便在 PreparedStatement 中使用它们来执行 SQL 语句。
- ☐ DiscountStore 使用 JDBC API 来执行 SQL 语句。
- ☐ 对于查询（SELECT），DiscountStore 将 ResultSet 转换回 Value Object 中的属性，这样就可以将 Value Object（业务逻辑对象）返回给业务逻辑层。

类似我们前面的 DeleteDiscountTest 的测试会有一个烦人的依赖问题，它依赖数据库中的数据来判断“Delete Discount”的功能正确性。我们的测试依赖与一个实际运行中的数据库，这不仅降低了我们的测试执行速度，而且使得我们的测试变得比较的脆弱（可以想像如果某人为了测试在数据库中也加入了一些测试数据会出现什么结果）。另外，为了验证这个类的每个部分，我们不得不去执行一条 SQL 语句，这是这个类的一项不在我们控制中的行为：我们得需要测试 JDBC 驱动，但它并不是我们写的代码。我们应当完全信任 JDBC API 以及我们的数据库提供的 JDBC 实现，也就是说，我们不应测试它们，或者去简单的测试它们，我们应当对其大部分的功能测试一次，以确保它能够正常的工作，然后将注意力集中到测试我们自己的代码中来。

注意： 你信任 *lib* 吗？——如果你不信任你的 JDBC 实现，那么你可以选择一个你信任的驱动或者自己编写测试来验证你的 JDBC 提供的行为是否满足你的期望。如果你需要经常升级你的 JDBC 驱动，或者你需要用相同的 JDBC 代码来支持多种数据库，这一点十分重要。

许多很熟悉 JUnit 的人可能会建议使用 Mock JDBC provider 来减少与具体产品的 JDBC 的耦合，但我们建议用另外一种方式：重构你的代码使得它对 JDBC API 的依赖减少到最小，然后减少那些

需要实际数据库以进行的测试。我们喜欢 Mock Object，但总是存在着 Mock JDBC provider 与具体的产品数据库的 JDBC provider 在某些重要的方面有差异的可能性。如果说你使用的 JDBC 驱动存在一个缺陷，你可以抱怨它，但是最终你还得面对这个缺陷，升级驱动或者换用别的驱动。你使用 Mock JDBC provider 测试得出的正确性的结果并不会对你使用正式的产品数据库的驱动有任何帮助。当然这种问题是可以得到控制的，但我们更愿意不遇见这种问题。我们通过对以上的行为进行单独的测试来减少需要实际数据库测试的数量。

第一种测试验证我们通过 Discount 的 Value Object 所创建的 PreparedStatement 的正确性。对于每一个 PreparedStatement 我们都需要这样一个测试。我们为我们想要测试的 DELETE 语句直接创建一个 PreparedStatement 来验证它的行为，但很快就不得不停下来。这个测试使用一个用 JDBC 方式实现 discount 存储的 DiscountStoreJdbcImpl 类：

```
public class DiscountStorePreparedStatementTest

    extends CoffeeShopDatabaseFixture {

    public void testCreate_RemoveExpiredDiscountAsOf()

        throws Exception {

        DiscountStoreJdbcImpl discountStoreJdbcImpl =

            new DiscountStoreJdbcImpl(getDataSource());

        PreparedStatement removeExpiredDiscountAsOf =

            discountStore.prepareJdbcStatement(

                "removeExpiredDiscountAsOf",

                expiryDate);

        // How do we check the PreparedStatement?

    }
```

```
}
```

但不幸的是，JDBC API 的设计并没有提供得到 `PreparedStatement` 的参数方法。虽然 `MockPreparedStatement` 能够提供我们需要的其他信息。我们还是更希望用另一种方式。我们验证将要传入给 `PreparedStatement` 的参数，而不是实际的创建一个 `PreparedStatement`。与其去测试一个 `DiscountStore` 的 JDBC 实现，我们宁愿选择去测试一个 `DiscountStore` 的 JDBC 的 *query builder*。下面是我们为另一个类编写的测试：

```
public void testParametersForRemoveExpiredDiscountAsOf()

    throws Exception {

    Date expiryDate = DateUtil.makeDate(2003, 1, 1);

    List domainParameters = Collections.singletonList(expiryDate);

    DiscountStoreJdbcQueryBuilder discountStoreJdbcQueryBuilder =

        new DiscountStoreJdbcQueryBuilder();

    List removeExpiredDiscountAsOfParameters =

        discountStoreJdbcQueryBuilder

            .createPreparedStatementParameters(

                "removeExpiredDiscountAsOf",

                domainParameters);

    List expectedParameters =

        Collections.singletonList(

            JdbcUtil.makeTimestamp(expiryDate));
```



```

    assertEquals(

        expectedParameters,

        removeExpiredDiscountAsOfParameters);

}

```

这个被测试的类是 `DiscountStoreJdbcQueryBuilder`，这个类能够创建 `DiscountStore` 支持的所有功能中要用到的 `PreparedStatement` 的参数，将基于 Java 业务对象的领域级的参数映射到匹配各种 SQL 数据类型的 SQL 级参数。我们将这个希望测试的功能命名为 `removeExpiredDiscountAsOf`，这样 `DiscountStore` 就可以通过它的 query builder 来得到与 `removeExpiredDiscountAsOf` 有关的 `PreparedStatement` 的参数。

你应该会注意到，虽然这是一个 JDBC 方式实现的 `DiscountStore` 的 query builder，但它里面并不存在对 JDBC API 的依赖。这个测试中并没有用到一个实际的数据库，而且 `PreparedStatement` 也不需要。通过了这个测试的产品代码相当简单：

```

public List createPreparedStatementParameters(

    String statementName,

    List domainParameters) {

    Date expiryDate = (Date) domainParameters.get(0);

    return Collections.singletonList(

        JdbcUtil.makeTimestamp(expiryDate));

}

```

编写出这个简单的测试你可能不会有成就感，但我们的确转换了 domain object（`Java.util.Date`）使之成为数据库对象（`Java.sql.Timestamp`）而且这个转换过程是必须的。你可能需要为这些错误编写测试——domain Parameters 可能没有得到期望的日期对象——如果你担心是否有人会传入不当的值。

第二种情况是查询的正确执行：也就是说，创建正确的 `DiscountStore Prepared-Statement` 并执行这个 Statement。由于这是特定 JDBC 的实现，我们现在将这个类重命名为 `DiscountStoreJdbcImpl` 并从中抽象出 `DiscountStore` 接口[Refactoring, 341]。同时，我们更改 `DeleteDiscountsTest` 使得它使用 JDBC 的 `DiscountStore` 实现。

```

public void testDiscountsExpiredThirtyDaysAgo() throws Exception {

```



```

        queryBuilder.createPreparedStatementParameters(
            "removeExpiredDiscountAsOf",
            Collections.singletonList(expiryDate));

        deleteStatement.clearParameters();

        int columnIndex = 1;

        for (Iterator i = parameters.iterator();
            i.hasNext();
            columnIndex++) {

            Object eachParameter = (Object) i.next();

            deleteStatement.setObject(columnIndex,
eachParameter);

        }

        deleteStatement.executeUpdate();

    }

    // Handle exceptions and clean up resources

}

```

我们可以将代码中的宋体部分抽取出来成为一个只负责执行 Prepared Statement 的方法。这里产生的这个方法对 domain object 没有任何的依赖，它是纯粹的 JDBC 客户端的代码：

```

private void executeDeleteStatement(

    PreparedStatement deleteStatement, List parameters)

```

```

throws SQLException {

    deleteStatement.clearParameters();

    int columnIndex = 1;

    for (Iterator i = parameters.iterator();

         i.hasNext();

         columnIndex++) {

        Object eachParameter = (Object) i.next();

        deleteStatement.setObject(columnIndex, eachParameter);

    }

    deleteStatement.executeUpdate();

}

```

我们还可以将它放到一个新的类中，并命名为 JDBC query executer，然后再也不对其进行任何测试！

我们可以拓展一下。下面回顾一下我们所做的：

1. 在 `removeExpireDiscountAsOf()` 中保存的与 discount 有关的代码只剩下了那个用来表达 DELETE 语句的 String。
2. 在上一步之后，我们就可以将所有用于捕获 Exception 的代码放到 `executeDeleteStatement()` 中。
3. 然后我们发现了一个问题：`DiscountStoreJdbcImpl` 提供了 SQL 的查询字符串，但是它的参数却是由 `DiscountStoreJdbcQueryBuilder` 处理的。这样看起来似乎不太合理，所以我们应该将 SQL 的查询字符串移动到 query builder 中去。

现在的 `removeExpiredDiscountAsOf()` 的代码就如下所示：

```

public void removeExpiredDiscountAsOf(Date expiryDate) {

```

```

String deleteExpiredDiscountsSql =

    queryBuilder.getSqlString("removeExpiredDiscountAsOf");

List parameters =

    queryBuilder.createPreparedStatementParameters(

        "removeExpiredDiscountAsOf",

        Collections.singletonList(expiryDate));

executeDeleteStatement(deleteExpiredDiscountsSql, parameters);
}

```

很短，不是吗？但是仍然存在一个问题，字符串“removeExpiredDiscountAsOf”出现了两次。为了解决这个问题，我们可以让 query builder 只提供一个单独的方法来同时返回查询字符串和参数，下面我们添加两个类，一个是 PreparedStatementData 类用于存储返回的查询字符串和参数。另一个是 JdbcQueryExecuter 类用于执行查询。下面我们需要修改 DiscountStoreJdbcImpl 以便使用它：

```

public class DiscountStoreJdbcImpl implements DiscountStore {

    // Some code omitted for brevity

    private DiscountStoreJdbcQueryBuilder queryBuilder =

        new DiscountStoreJdbcQueryBuilder();

    private JdbcQueryExecuter queryExecuter;

    public DiscountStoreJdbcImpl(DataSource dataSource) {

        queryExecuter = new JdbcQueryExecuter(dataSource);
    }
}

```

```

    }

    public void removeExpiredDiscountAsOf(Date expiryDate) {

        PreparedStatementData
removeExpiredDiscountAsOfStatementData =

        queryBuilder.getPreparedStatementData(

            "removeExpiredDiscountAsOf",

            Collections.singletonList(expiryDate));

        queryExecutor.executeDeleteStatement(

            removeExpiredDiscountAsOfStatementData);

    }

}

```

你可以看到在 DiscountStore 中保留的原有的代码是如此的少！建立 SQL 查询字符串然后再解析参数——将它们从 domain parameter 转换为 SQL parameter——全部都在 query builder 中完成，并且它的测试不需要数据库的支持。你只需要少量的测试就可以验证删除语句是否顺利执行。就是这样。你可能有 100 个不同的 delete 语句，但是可能只有一个方法使用 JDBC API 来执行这些语句。这就意味着你可以有许多的运行于内存中的测试和只有很少量的——小于十个——需要数据库支持的测试。这使得测试的执行节省了时间并且减少了复杂度。

不要再去测试你的 JDBC 的供应商，应该去测试你自己的代码。

10.1 Test making domain objects from a ResultSet

10.1 测试从 **ResultSet** 创建 **domain object**

◆ 问题

你想要验证你从 SELECT 语句返回的 ResultSet 中创建的 domain object 的正确性。

◆ 背景

在你使用 JDBC 来执行查询时只有两种出错的可能性。你可能使用了不正确的 SQL 语句，也可能是因为将排序不当的数据传入了对象。对其测试最直接的方式是同时测试这两个问题：在数据库中初始一些数据，然后创建 SELETE 语句，执行它们，再然后验证得到的结果是不是你想要的结果。这是最直接的方式，但却并不优化。

很快，每个 SELECT 测试看起来几乎都一样了：不同的地方仅仅是查询字符串以及是否需要调用 getString()、getInt()、getBlob().....等等。应该使用一种方法来消除这种重复的工作。我们在这一章的引言部分就详细描述了这方面的内容！一旦你应用了这些技术，并且只有你的数据访问层的一部分会去执行 SQL 语句，你剩下的任务就只是对我们前面提到的两部分的内容的测试：SQL 语句以及无序的逻辑。我们的解决办法是针对后者的；10.2 节“验证你的 SQL 语句”是有关于第一个问题的详细讨论。

◆ 诀窍

我们需要对将 ResultSet 转换为其所表示的领域对象的 Set 进行测试。最简单的方法是用已知的数据构造一个 ResultSet，调用“生成 domain object”的方法，然后将生成的结果与期望的结果相比较。不幸的是，JDBC 并没有提供一个对 ResultSet 的单独的实现以使得我们可以对其直接增加数据。这是一个典型的 mock object 适用的情况。还好，mock object 提供了一些容易使用的 ResultSet 的实现：针对于返回结果为多条记录的 MockMultiRowResultSet，针对于返回结果为单条记录的 MockSingleResultSet。而后者还提供了一个简单的接口并且其效率要比前者高。

注意： Mock Object 项目——这个项目是从对 Tim Mackinnon，Steve Freeman，和 Philip Craig 在他们的论文“Endo-Testing: Unit Testing with Mock Objects.”中描述的一个想法的实现开始的。这个工程中包含了对 J2SE 以及 J2EE 中许多的库（lib）的 mock 的实现。我们对这些包的依赖并不强——我们大部分情况只是使用 EasyMock 来实现 mock object——但是 Mock Object 项目的确提供了一套优秀的预定义的 mock object。我们建议在你有了 mock object 的概念之后先选择使用他们的 object 而不是构建你自己的 mock object。了解 mock object 对于测试的影响的方式很重要。Mock Object 的作者指出过，Mock Object 能够帮助你去关注对象之间的交互，而不必担心每个测试的实现细节。

现在我们拥有了一个可以将数据硬编码于其中的结果集，我们可以开始写我们转换 ResultSet 对象到领域对象的测试。我们将验证从 catalog.discount 表中的数据生成一个 Discount 对象的过程。清单 10.1 是一个很好的开始：

清单 10.1 unmarshalling ResultSet 数据的测试

```
public void testDiscountJoinWithDiscountDefinition()

    throws Exception {
```

PercentageOffSubtotalDiscountDefintion

expectedDiscountDefinition =

new PercentageOffSubtotalDiscountDefintion();

expectedDiscountDefinition.percentageOffSubtotal = 25;

Date expectedFromDate = DateUtil.makeDate(1974, 5, 4);

Date expectedToDate = DateUtil.makeDate(2000, 5, 5);

Discount expectedDiscount =

new Discount(

expectedFromDate,

expectedToDate,

expectedDiscountDefinition);

DiscountStoreJdbcMapper mapper = new DiscountStoreJdbcMapper();

← 存放 name/value
一行一对

Map rowData = new HashMap();

rowData.put(

"typeName",

PercentageOffSubtotalDiscountDefintion.class.getName());

rowData.put("fromDate", JdbcUtil.makeTimestamp(1974, 5, 4));


```
rowData.put("toDate", JdbcUtil.makeTimestamp(2000, 5, 5));
```

```
rowData.put("percentageOffSubtotal", new Integer(25));
```

```
rowData.put("suspended", null);
```

```
MockSingleRowResultSet resultSet = new MockSingleRowResultSet();
```

← 向 ResultSet 中填充源代码数据

```
resultSet.addExpectedNamedValues(rowData);
```

← 指向 ResultSet 的第一行

```
assertTrue(resultSet.next());
```

```
Discount actualDiscount = mapper.makeDiscount(resultSet);
```

```
assertEquals(expectedDiscount, actualDiscount);
```

```
}
```

忽略掉使用 mock object 带来的复杂性，测试有一个一般形式，即：我们创建期望中的 Discount 对象，再通过 JDBC 的数据来创建一个我们将要去处理的 ResultSet，再去检查映射器的行为，我们将使用 MockSingleRowResultSet，因为我们的测试仅仅只涉及到单条记录。相应的，如果你的测试中涉及到多条记录，你可以使用 MockMultiRow- ResultSet。

清单 10.2 展示了测试中涉及的 DiscountStoreJdbcMapper 类：

清单 10.2 DiscountStoreJdbcMapper

```
package junit.cookbook.coffee.jdbc;
```

```
import java.sql.ResultSet;
```

```
import java.sql.SQLException;
```

```
import junit.cookbook.coffee.data.*;
```

```
import com.diasparsoftware.jdbc.JdbcMapper;
```

```
public class DiscountStoreJdbcMapper extends JdbcMapper {
```

```
    public Discount makeDiscount(ResultSet discountResultSet)
```

```
        throws SQLException {
```

```
        Discount discount = new Discount();
```

```
        discount.fromDate = getDate(discountResultSet, "fromDate");
```

```
        discount.toDate = getDate(discountResultSet, "toDate");
```

```
        discount.discountDefinition =
```

```
            makeDiscountDefinition(discountResultSet);
```

```
        return discount;
```

```
    }
```

```
    public DiscountDefinition makeDiscountDefinition(
```

```
        ResultSet resultSet)
```

```
        throws SQLException {
```

```
        String discountClassName = resultSet.getString("typeName");
```

```
        if (PercentageOffSubtotalDiscountDefintion
```

```

        .class.getName().equals(discountClassName)) {

    PercentageOffSubtotalDiscountDefintion

        discountDefinition =

            new
PercentageOffSubtotalDiscountDefintion();

        discountDefinition.percentageOffSubtotal =

            resultSet.getInt("percentageOffSubtotal");

        return discountDefinition;

    }

    else

        throw new DataMakesNoSenseException(

            "Bad discount definition type name: "

                + discountClassName + "");

    }

}

```

◆ 讨论

我们应该提醒你当你使用这项技术时，确保你硬编码的结果集和从产品数据库中获得的结果集是相同的，这一点至关重要。应当警惕那些数据库编程中经常出现的问题（而不仅仅是 JDBC）。比方说时区的差别，数字的格式的差别以及超过长度限制的字符串（举个例子，某些数据库会默认将超过长度限制的字符串的超过部分截取掉）等问题。理解你实际的数据库的运行机制并且找出你的 mock JDBC object 与你的数据库提供的 JDBC 的实现之间的差别都很重要。你可能还需要创建一个自定义的 mock object 来更好的模仿你的数据库提供的 JDBC 的特性。

同样，这项技术告诉了我们高耦合就意味着低的复用率。JDBC 的 ResultSet 是一个拥有很多责任的对象的典型例子。它至少要做三件重要的事情：再现表的一条记录，用 Iterator 再现表中的数据集合，以及与底层数据库交互以获得查询结果。对于一个简单对象，它的任务显得太多了。当我们编写测试时，我们经常仅仅只想使用其中的某一条功能。一个内部耦合严重的类将会阻碍它的复用。

我们想要的仅仅是数据，我们并不希望去和数据库交互来得到它，如果 JDBC 提供了一个单独的类来表示 ResultSet 中的记录，我们就可以使用产品级别的实现来替代我们在测试中使用的 mock object。我们可以测试将 SQL 数据类型转换为 Java 数据类型的方式，以及相应的表的属性转换为 Java 对象的属性。不幸的是，并不存在这样的单独的类，所以我们不得不去使用 mock ResultSet object。鉴于 Mock Object 对接口的实现数量，我们应当感谢它。为了编写一个自定义的 ResultSet 的 mock object 我们需要做很多的工作，即使我们仅仅是需要使用那个接口中的很小的一部分。MockResultSet 的实现中包含了一个实际的 ResultSet 从数据库中取得数据。如果 JDBC 先前有了那些包含单独责任的对象，我们就不需要在这里使用 MockResultSet 了；而且我们更希望使用产品代码来测试而不是使用模仿的对象。

◆ 相关

- 10.2—验证你的 SQL 语句

10.2 Verify your SQL commands

10.2 验证你的 SQL 语句

◆ 问题

你希望验证你的 SQL 语句但不想每次都去和数据库交互。

◆ 背景

保证 JDBC 代码的正确性的三个要素是：

1. 正确的 SQL 语句。
2. 正确地将领域对象转换为 PreparedStatement 的参数。
3. 正确地将 ResultSet 转换为领域对象。

我们已经介绍过了，其他的使用 JDBC API 来与数据库交互的代码可以被抽取出来到一个单独的类或者方法，这样就仅仅需要测试一次。用 JUnit 来验证你的 SQL 语句显得有些奇怪。当我们编写测试的时候，就类似于做如下的断言：

```
assertEquals(  
    expectedSqlString,  
    queryBuilder.getSqlString(statementName);
```

这和将一对 key-value 加入 Map 中然后测试能不能用 key 得到 value 是一样的。它仅仅测试了 key 的 hashCode() 方法以及 HashMap 或者 TreeMap 的 Java 的实现，但这些都与 JDBC 和 SQL 无关。应该考虑一下用 JUnit 来验证 SQL 语句的合理性。

◆ 诀窍

我们建议不编写 JUnit 的测试来验证你的针对某个数据库的 SQL 语句。因为在这里 JUnit 不是一个很合适的工具。我们建议你用你的数据库提供的 SQL 语句执行器来验证你的 SQL 语句。比如 Mimer 的 BatchSQL，ORACLE 的 sqlplus。

我们建议手工来测试，有没有搞错？

当然没有。可能你不同意这个观点。一般来说，验证 SQL 语句的最好的办法是多次执行它，修改它，最后确定它是正确的。那时你可能会去编写测试来验证 SQL 语句是否和它上一个正确运行的版本相同。简单的说就是应用 *Golden Master* 技术。

注意：*Golden Master*——或者是“golden result”。一个 Golden Master（或者 Gold Master）依赖于你访问的对象——它是由你手工测试一次的测试结果，然后将它作为你将来做其他测试的基础。将来执行的测试中如果其结果与 Gold Master 的输出结果是吻合的，就通过。不要将这项技术与 anti-pattern *Guru Checks Output* 相混淆了。我们讲的是先用手工的方式验证一次，然后使用其输出的结果来实现自我验证的测试。而 *Guru Checks Output* 中你需要 guru 去检验每个测试的输出，而如果没有 guru，所有的测试都不会被执行。在 Gold Master 中，我们使用一次 guru 技术，然后在整个测试中都保留它。

我们以咖啡店应用程序为实例来说明这点。当顾客从在线商店采购咖啡的同时，某处的产品管理员却在更新目录。当我们决定购买新型咖啡豆时，她需要添加产品到数据库。在系统某处有对应的 SQL 语句来向适当的表插入新的咖啡豆产品。让我们将它作为遗留的程序方案（legacy coding scenario），这就意味着，编码已经存在并且是“正确的”——至少暂时是这样。我们想要添加测试来帮助我们修改 SQL 语句后，能够立即看到效果。在遗留代码的情况下，这是我们首先要做的。

首先我们找出那个执行 SQL 语句的方法：

```
public void addProduct(Product toAdd) {

    Connection connection = null;

    PreparedStatement insertStatement = null;

    try {

        connection = dataSource.getConnection();

        insertStatement =

            connection.prepareStatement(

                "insert into catalog.beans "

                + "(productId, coffeeName, unitPrice) values "

                + "(?, ?, ?)");

        insertStatement.clearParameters();

        insertStatement.setString(1, toAdd.productId);

        insertStatement.setString(2, toAdd.coffeeName);

        insertStatement.setInt(3, toAdd.unitPrice.inCents());

        if (insertStatement.executeUpdate() != 1)

            throw new DataMakesNoSenseException(
```

```

        "Inserted more than 1 row into catalog.beans!");
    }

    catch (SQLException e) {
        throw new DataStoreException(e);
    }

    finally {
        try {
            if (insertStatement != null)
                insertStatement.close();

            if (connection != null)
                connection.close();

        }

        catch (SQLException ignored) {
        }

    }
}

```

如果我们重构这个方法，我们就可以将 SQL 语句移动到如下的一个简单方法中：

```

public String getAddProductSqlString() {
    return "insert into catalog.beans "
        + "(productId, coffeeName, unitPrice) values "
        + "(?, ?, ?)";
}

```

```
}
```

注意一般情况下，我们倾向将这些字符串移动到一个方法中而不是符号常量,因为为了某种通用的目的（比如：一个基于关键字的查找方法），重构一个方法更简单。这只是接口与实施分离的一种特殊情况.....可能我们偏离主题了。

现在我们可以将 SQL 字符串抽取出来，我们就可以编写下面的测试：

```
public void testAddProductSqlString() throws Exception {  
  
    CatalogStoreJdbcImpl store = new CatalogStoreJdbcImpl(null);  
  
    assertEquals("", store.getAddProductSqlString());  
  
}
```

注意, 这里我们期待的是一个空字符串, 很明显这并不是我们真正期望目录数据库执行的字符串。我们写这个测试因为我们对我们得到的实际 SQL 指令缺乏信心。由于我们判断实际执行的 SQL 语句是正确的, 我们应该让数据库告诉我们那个字符串的含义, 而不是去猜测它。执行测试然后获得错误消息可能会更加容易。

expected:<> but was:<insert into catalog.beans

⇒ (productId, coffeeName, unitPrice) values (?, ?, ?)>

现在我们知道了期望的 SQL 语句，所以我就可以将它加入到我们的测试中以便将来使用：

```
public void testAddProductSqlString() throws Exception {  
  
    CatalogStoreJdbcImpl store = new CatalogStoreJdbcImpl(null);  
  
    assertEquals(  
  
        "insert into catalog.beans "  
  
        + "(productId, coffeeName, unitPrice) values "  
  
        + "(?, ?, ?)",  
  
        store.getAddProductSqlString());  
  
}
```

现在这个测试通过了, 这样我们就可以做一些重构的工作了, 譬如从数据库中获得查询。如果我们的重构在某种情况下修改了 SQL 语句, 这个测试会立刻告诉我们, 在什么地方我们可以更新 Gold Master 的 String, 或者我们最近加入的某些代码产生了错误。

◆ 讨论

下一步我们要将 Gold Master String 存储到 properties 文件中。一旦你所有的 SQL 语句都保存在文件里了，你可以将它作为你的 SQL 查询的源来使用，而不是将它们硬编码在你的 JDBC 客户端代码中。这种重构可以使得你的设计更加的灵活。如果明天数据库小组决定重新定义数据库，使用不同的 schema 名称或不同的表名，你只需要修改一个文件然后就可以再次运行所有测试。

如果你无法像我们这样重构 SQL 语句又会怎样？你应该如何去确定 Gold Master String？答案是替代掉那个为你收集信息的 DataSource 的实现。为了避免重复发明轮子，你可以使用 Mock Object 来编写我们已经描述过的必不可少的测试：

当我们执行这个测试，我们会得到下面的信息：

```
public void testAddProductSqlString() throws Exception {
```

```
     存储 Gold Master 字符串 String expectedSqlString = "";
```

```
    MockDataSource dataSource = new MockDataSource();
```

```
    MockConnection2 connection = new MockConnection2();
```

```
    MockPreparedStatement expectedStatement =
```

```
        new MockPreparedStatement();
```

```
    expectedStatement.addUpdateCount(1);
```

```
    dataSource.setupConnection(connection);
```

```
    connection.setupAddPreparedStatement(
```

```
        expectedSqlString,
```

```
     告诉 Mock Connection 期望哪一个 SQL 字符串 expectedStatement);
```

```
    CatalogStoreJdbcImpl store =
```

```
        new CatalogStoreJdbcImpl(dataSource);
```

```

Product product =
    new Product(
        "762",
        "Expensive New Coffee",
        Money.dollars(10, 50));

store.addProduct(product);

expectedStatement.verify();
}

```

产品代号将提供 SQL 字符串

根据产品代号来审核 Gold Master

这就是 Gold Master String！现在我们就可以在我们的测试中使用它了，把它的值赋给上面的变量 expectedSqlString。当我们做完这个修改然后再次执行测试时，测试通过了！现在 Gold Master String 被放在测试中来防止后续的更改。由于客户端不能够直接访问 SQL 字符串，这里我们还需要再做一些工作，但是这项技术是相同的：执行一次测试来获得一个返回结果，然后修改测试，以刚才执行测试得到的返回结果作为以后做测试的期望结果。

请记住 Gold Master 与 Guru Checks Output 之间的差别：Gold Master 中你使用手工的方式获得一次执行结果，然后测试就以这个执行结果做自我验证；而 Guru Checks Output 则是你需要在每次执行测试后都对其结果进行手工的检验。如果你使用着 Guru Checks Output，如果你是 guru，那么我恐怕你就得天天守在你的代码旁边，不用考虑旅行计划之类的了。

◆ 花絮

我们的评论者中的一个——George Latkiewicz，建议了一种 JUnit 的替代技术。由于我们未在一个真正的项目中尝试过这个技术，我们不能真正地推荐它，但它听起来很值得尝试。

这项技术很简单。从数据库的连接中向 JDBC 驱动请求使用 PreparedStatement 编译你的 SQL 语句。当你调用 prepareStatement() 时，如果 statement 不是正确的，Connection 对象应当会抛出 SQLException 异常。

这种情况下，验证的地方并不仅仅局限于 SQL 的语法，但是 JDBC 驱动应该能够报告不正确的列和表名称。你可以创建一个参数化的测试用例（参见 4.8 节，“创建数据驱动的 Test Suite”）来测试你的应用中的各个需要执行的 SQL 语句。测试将会简单地为每个 SQL 语句调用 `prepareStatement()`，但实际上并不会去执行它们。George 发现这很有用，尤其是当 statement 需要频繁地更新或者需要支持多种数据库系统或 JDBC 驱动时（包括平台差异）。但 George 发现在动态生成 SQL 语句的情况下，这个技术毫无价值。

现在我们仍然认为 Gold Master 方法，总体上来说，无论从投入编写测试所花费的精力还是执行测试所花费的时间上来讲都是一个更好的选择，但我们只是推测。为了公平起见，我们在下结论之前会尝试一下 George 的方法。因此，我们将它提供给你作为另一种选择并且希望这会对你有所帮助。

◆ 相关

□ Keith Stobie, “*Test Result Checking Patterns*”中详细描述了 Gold Master，或者说 Batch Check, Golden Results 以及 Reference Checking。

10.3 Test your database schema

10.3 测试你的数据库

◆ 问题

你想要测试你的数据库，校验其 null 列、索引、外键以及触发器等。

◆ 背景

我们敏捷人士喜欢生活在一个梦想的世界，在那里 *team* 共同拥有数据库。但在现实里程序员和数据库管理员需要一起融洽的工作来为我们的顾客（或者 boss）建立完美的数据库。数据库是灵活的并且每个人对变动都很敏感。当变动发生时，大家都会立刻知道。如果真的是这样，那该多好啊。

现实的情况是，大多数组织中，在“数据库管理员”与“程序开发人员”之间存在着很大的分歧，甚至不能够让他们在一起和谐的工作。即便不是因为一些不合理的管理控制策略的原因，如果你的应用是围绕着数据库开发的，那么你就有一个很好的机会来让存在分歧的团队来共同维护数据库：所有的修改都让他们经手。每当你有需求需要变更数据库，你需要确定，你要做的变更被正确地理解和执行了，而且和使用测试的方式比较，哪一种方式更好呢？

注意：实际发生的是……在你相信将变更提交给开发团队中的这两部分人而一定就不会出问题之前，让我们来看看这个故事。一个程序员——我们就叫他 Joe 吧——根据存储在内存中的对象创建了他的组件，然后将结果转移到关系数据库中。于是他创建了数据库，使用了简单的 DDL 定义了他的数据库，然后

将变更递交给了数据库小组。一个星期后, 在他将变更后的数据库设计加入到他这一周的测试驱动中, 在他针对这个变更过后的数据库执行他的测试时。瞧——有一个失败了！惊讶的 Joe 重新检查了测试驱动中的数据库创建脚本, 却发现数据库小组错误地设置了某个表中一个属性的 unique index。于是惊讶的 Joe 就问数据库小组的头儿怎么回事。那个头儿说：“我们维护的时候使用的是 ERwin, 可能 Erwin 在我们将你的 DDL 导入到工具中, 然后再将整个数据库定义导出到测试驱动的过程中, 出现了什么差错。”即使 Joe 很精确地提交了 DDL 到数据库表中, 在转换的过程中还是可能会出现信息的丢失。于是 Joe 得到了一个可贵的教训：数据库 schema 不可靠！

如果你有类似 Joe 的经验, 我们建议你增加一些测试保护你自己, 以免“惊讶”。

◆ 诀窍

或许解决这个问题的最佳办法与测试无关：对你的应用中的数据模型做一个简单的、清晰的描述, 然后从这个描述出发创建 DDL 脚本。Martin Fowler 描述了使用 XML 文件——容易解析, 也因此容易使用 XPath 验证（参见第 9 章）——作为应用中数据表现形式的一种简单描述, 也许可以根据这种数据表现形式生成数据库 [PEAA, 49]。现在我们可以找到解析 XML 的工具, 但却找不到解析 DDL 的 Java 工具。

现在让我们假设, 你需要在没有 XML 帮助将数据库中的数据再现的情况下去测试数据库。这种情况下通常的策略是为你的数据库的以下每个方面都编写测试, 你需要验证：

- ☐ 表以及表的属性是否存在
- ☐ 主键的正确性
- ☐ 外键的正确性
- ☐ 触发器的正确性
- ☐ 默认值以及数据的限制的正确性
- ☐ 存储过程的正确性
- ☐ 数据库对象的权限的正确性

这些测试中的任何一个, 大体上有两种策略: 或者针对数据库的 meta data 做断言, 或者针对数据库做测试, 通过执行查询和检查结果来判断其正确性。我们更喜欢 meta data 的方式, 因为它并不依赖于数据库中的任何数据, 但不同的数据库提供商所支持的 meta data 各不相同。我们的 Coffee Shop 应用中使用了一个叫做 Mimer (<http://www.mimer.se>) 的数据库来存放数据, 并且我们不肯肯定它的 JDBC 提供者支持数据库 meta data 的情况如何, 所以我们尝试了清单 10.3 中的 Learning Test。

清单 10.3 数据库的 Learning Test

```
public void testTablesAndColumnsExist() throws Exception {

    MimerDataSource coffeeShopDataSource = new MimerDataSource();

    coffeeShopDataSource.setDatabaseName("coffeeShopData");

    coffeeShopDataSource.setUser("admin");

    coffeeShopDataSource.setPassword("adm1n");


    Connection connection = coffeeShopDataSource.getConnection();

    DatabaseMetaData databaseMetaData = connection.getMetaData();

    ResultSet schemasResultSet = databaseMetaData.getSchemas();


    Map databaseSchemaDescriptors = new HashMap();

    while (schemasResultSet.next()) {

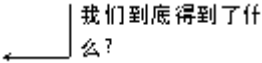
        databaseSchemaDescriptors.put(

            schemasResultSet.getString("TABLE_SCHEM"),

            schemasResultSet.getString("TABLE_CATALOG"));

    }

    schemasResultSet.close();

    connection.close();


    fail(databaseSchemaDescriptors.toString());

}
```

本质上来说，这只是一个“printf”，不同的是它更容易被转换到测试中来帮助我们发现不兼容的问题或者将来的版本中的 Mimer JDBC provider 的变化。在看到 fail() 语句的输出结果之后，我们就能确定应该对什么做断言。我们发现的第一个问题是，字段 TABLE_CATALOG 并不在结果集中——但 DatabaseMetaData.getSchemas() 的 Javadoc 中说是应该有的。我们需要仔细的检查一下 meta data。幸运的是，我们能使用 Diasparsoft 工具里的 JdbcUtil 得到一个可以读懂的 JDBC 结果集。我们在处理结果集之前添加了这行代码：

```
fail(JdbcUtil.resultSetAsTable(schemasResultSet).toString());
```

结果集中只有一个列：TABLE_SCHEM，这就足够了，我们已经可以确定我们期望的 CATALOG schema 就在那里。于是我们修改我们的测试，来印证这一点并将跟踪代码去除。清单 10.4 展示了这个修改后的测试。

清单 10.4 在学到一些东西后的 Learning Test

```
public void testTablesAndColumnsExist() throws Exception {  
  
    MimerDataSource coffeeShopDataSource = new  
MimerDataSource();  
  
        coffeeShopDataSource.setDatabaseName("coffeeShopData");  
  
        coffeeShopDataSource.setUser("admin");  
  
        coffeeShopDataSource.setPassword("adm1n");  
  
        Connection connection =  
coffeeShopDataSource.getConnection();  
  
        DatabaseMetaData databaseMetaData = connection.getMetaData();  
  
        ResultSet schemasResultSet = databaseMetaData.getSchemas();  
  
        List schemaNames = new LinkedList();  
  
        while (schemasResultSet.next()) {  
  
            schemaNames.add(schemasResultSet.getString("TABLE_SCHEM"));  
  
        }  
}
```

```
schemasResultSet.close();
```



```
connection.close();
```

```
assertTrue(schemaNames.contains("CATALOG"));
```

```
}
```

我们将“schema descriptors”改为了“schema names”——我们认为应该不只仅有一个属性。它们都是字符串。我们不再需要 Map 了，因为我们在 collection 中存放的内容都是简单的数据值，一个 List 就可以胜任。我们的断言很直接和容易理解：我们期望那里存在一个叫做 CATALOG 的 schema。现在这个测试是很脆弱的，因为它是假设在 schema meta data 返回的结果为大写的前提下进行的。你可以使用 Diasparsoft 工具集中的 CollectionUtil，它为字符串的集合提供了大小写不敏感的查寻功能。使用下面的代码替换上面的断言：

```
assertTrue(
    CollectionUtil.stringCollectionContainsIgnoreCase(
        schemaNames,
        "catalog"));
```

上面讲到的这两个测试现在都通过了，并且我们成功地验证了我们的数据库中存在名为 CATALOG 的 schema。你还可以使用 ResultSetMetaData API 的其他部分来验证表的存在性，表的属性，以及约束条件——当然还要取决于你的数据库提供商对这些功能的支持程度。并不是所有的提供商都支持这些功能。如果 meta data 让你很失望怎么办？回到出发点来：使用特定的词语来描述一个数据库应有的行为并且为其编写相应的测试。清单 10.5 中的测试验证了 coffeeName 在表 CATALOG.BEANS 中是否是 unique 的，即使 coffeeName 不是一个主关键字。

清单 10.5 校验 unique 索引

```
public void testCoffeeNameUniquenessConstraint() throws Exception {

    MimerDataSource coffeeShopDataSource = new
MimerDataSource();

    coffeeShopDataSource.setDatabaseName("coffeeShopData");

    coffeeShopDataSource.setUser("admin");

    coffeeShopDataSource.setPassword("adm1n");
```

```
        Connection connection =  
coffeeShopDataSource.getConnection();
```

```
        PreparedStatement createBeanProductStatement =  
            connection.prepareStatement(  
                "insert into catalog.beans "  
                + "(productId, coffeeName, unitPrice) "  
                + "values (?, ?, ?)");  
  
        createBeanProductStatement.clearParameters();  
  
        createBeanProductStatement.setString(1, "000");  
  
        createBeanProductStatement.setString(2, "Sumatra");  
  
        createBeanProductStatement.setInt(3, 725);  
  
        assertEquals(1, createBeanProductStatement.executeUpdate());  
  
        // How will Mimer react to the duplicate entry?  
    }
```

由于我们对 Mimer 也是刚刚接触，我们也不确信它将对重复的数据记录会做何种反应，于是我们也就不知道在测试中将什么来作为断言中的比较目标。这就意味着我们还得摸索着测试的手段开始。让我们将同一条 update 语句执行两次再看看会发生什么。我们以下面这行代码替换掉测试中的 comment 部分：

```
        createBeanProductStatement.executeUpdate();
```

当我们执行测试时，Mimer 抛出一个 java.sql.SQLException: UNIQUE constraint violation，这样我们就知道应当期望 SQLException，但我们不知道应该与哪个 SQLState 对应，如此我们又修改了这个测试，将前面的那条语句替换为下面的代码：


```

try {

    createBeanProductStatement.executeUpdate();

}

catch (SQLException expected) {

    fail(expected.getSQLState());

}

```

再次执行测试，我们又得到同样的异常：UNIQUE constraint violation，这是为什么？

是的，原因就在我们前面做的测试中向数据库插入的数据。这就是为什么我们推荐写尽可能多的不包含实际数据库的测试，即便是在这样简单的测试中，我们都会碰到问题。参见 10.6 节，“在你的测试功能中管理外部数据”来了解一些如何处理引入数据库的测试的策略。现在回到我们的测试，我们增加一些代码，以便在测试的开始将数据库 CATALOG.BEANS 表中的所有数据删除，然后再执行我们的测试。这时 SQLState 的值是 23 000。我们立刻查找 Mimer 的文档发现这个值代表“integrity constraint violation（违反数据完整性约束）”。参见清单 10.6，这个测试的最终版本：

清单 10.6 CoffeeShopDatabaseSchemaTest 最终版

```

package junit.cookbook.coffee.jdbc.test;

import java.sql.*;

import java.util.LinkedList;

import java.util.List;

import junit.framework.TestCase;

import com.diasparsoftware.java.util.CollectionUtil;

import com.mimer.jdbc.MimerDataSource;

public class CoffeeShopDatabaseSchemaTest extends TestCase {

    public void testCoffeeNameUniquenessConstraint()

```

```

throws Exception {

    MimerDataSource coffeeShopDataSource = new
MimerDataSource();

    coffeeShopDataSource.setDatabaseName("coffeeShopData");

    coffeeShopDataSource.setUser("admin");

    coffeeShopDataSource.setPassword("adm1n");


    Connection connection =
coffeeShopDataSource.getConnection();

    connection.createStatement().executeUpdate(

        "delete from catalog.beans");


    PreparedStatement createBeanProductStatement =

        connection.prepareStatement(

            "insert into catalog.beans "

                + "(productId, coffeeName, unitPrice) "

                + "values (?, ?, ?)");

    createBeanProductStatement.clearParameters();

    createBeanProductStatement.setString(1, "000");

    createBeanProductStatement.setString(2, "Sumatra");

    createBeanProductStatement.setInt(3, 725);


    assertEquals(1,
createBeanProductStatement.executeUpdate());

```

```

        try {

            createBeanProductStatement.executeUpdate();

            fail("Added two coffee products with the same name?!");

        }

        catch (SQLException expected) {

            assertEquals(

                String.valueOf(23000),

                expected.getSQLState());

        }

    }

    // JDBC resource cleanup code omitted for brevity

}

```

有一件事情可能还会让你担心，这个测试可能假设了数据库的一些信息。比方说，失败消息意味着 coffee name 字段重复，但相反的，可能是产品 ID 的问题。这种类型的依赖将可能导致对 JDBC 代码的测试的增加，尤其是针对一个实际运行的数据库做测试时。假设三个月之后某个表的 unique 约束改变了，那么这个测试很可能就会失败。力图提供精确帮助的失败消息，现在却带来了误导。对于一张只有三个字段的表来说，那不是什么大问题；但是，对于一张拥有许多属性的表，这可能会使程序员在查找问题的时候很费力以至于在调试上花费很多的时间。我们倾向于错误信息中加入更多相关信息，但就像任何其他习惯一样，有时这会变成坏习惯。如果你能意识到潜在的问题，如果问题发生，就应该积极的处理它，避免让问题出现。

◆ 讨论

在针对一个实际运行的数据库进行测试的过程中，我们已经遇到几个考虑值得的问题，即便这个数据库完全属于你自己。为了保证测试的独立性，你需要在测试之前将数据库中的数据清空。在实际测试中有两个关键：

- 表之间的依赖的增长——你必须增加逻辑来清理你的数据库的表，并且如果你的表的外关键字限制越是复杂，你需要增加的逻辑也就越多。在中等大小应

用程序中拥有超过 40 张数据库表，而其中只有很少的表中存在外关键字约束的情况并不多见。

□ 数据库是一种昂贵的外部资源——你针对数据库所编写的测试越多，你的测试执行的速度就会越慢。请记住程序员测试的目标是保证测试足够快速以便程序员在编程的过程中能够经常的执行它们。这就是为什么你需要保持你的设计灵活性和重构你的 safety net 来减少增加功能的费用。

注意： 统计数据——更加直接的测试方式是针对一个实际的数据库来编写测试，特别是对于那些没有将 JDBC 客户代码从物理数据库分离的程序员。一些统计数据对于我们理解远离数据库的重构的优势是非常重要的。我们写了两个测试来核实数据库的哪个字段可以为空（ nullable ）。第一方法采取了 Martin Fowler 的建议将数据库的定义转移到 XML 中；第二种方法使用了上面描述的数据库 meta data。在 Pentium4 1.7GHz，512 MB RAM 的计算机上运行，前一个测试平均耗时 0.05 秒，而后者却用了 0.5 秒。差别看上去很小，但这个测试仅仅是测试同一个数据库表中的七个字段。假设每张表中有 1000 个这样的字段需要检查并且有大约 50 张到 200 张表——当然，这取决于数据库设计者的设计理念以及涉及设计风格。乘以 143（1000/7），然后差别就是 $143 \times 0.45 = 64.35$ 秒也就是超过了一分钟！现在随着你的测试的数量的增长，无论你的测试的规模有多大，初始化过程，譬如建立数据库连接所花费的时间是不会变的。即使我们有将差别夸张，其实它可能是 30 秒左右，就假设每个测试执行需要 30 秒，那么在平均项目生命周期的一半过程中，每个成员每天执行的每个测试都需要 30 秒，拿出你的计算器，看看把它们加起来的结果。

所以当我们想提供你一些验证一个实际运行的数据库 schema 的例子时，一般来说在测试中不实际引用数据库是明智的选择。如果你确实担心可能不能与数据库协同运行，就针对数据库编写一个 Learning Test，然后让它们作为你的后台的一部分来执行。至少，你可以通过测试你的应用的用户接口，也就是 End-to-End 的测试来验证你的应用程序与数据库之间的交互是正常的。如果对于你的数据访问层的测试以同样的引入一个实际运行中的数据库的方式来进行，那就造成了测试的重复，造成了你的努力的浪费。你应当将你的 Object Test（ Object Test ）的重点放在独立的对象上。

◆ 相关

□ 10.6—在你的测试功能中管理外部数据

10.4 Verify your tests clean up JDBC resources

10.4 确定测试释放了 JDBC 资源

◆ 问题

你已经编写了创建 JDBC 对象的测试，譬如 connection、statement、resultset，你希望通过确认你的测试清理了自身以避免资源泄露。

◆ 背景

泄漏 JDBC 资源会造成很多的问题。举个例子，你可以不释放连接直到数据库放弃连接等待，并标记其不可用，来轻易的破坏 JDBC 连接池。我们在网上发现的这篇文章对这个问题进行了比较详细的描述。

注意： 连接池性能——许多用户报告说 JDBC 连接池在提高性能方面很有限，当处理了上百个用户事务之后，用户响应时间实际上是在下降。似乎是服务器性能问题，但这实际上是程序中的问题：程序中忘记了关闭 JDBC 会话以及释放 JDBC 资源。

获得 JDBC 资源是一个耗资源的过程，这就是为什么 JDBC 连接池存在的原因。没有连接池的时候，如果你关闭和释放 JDBC 连接失败，将不会有明显的性能下降，但你可能会耗尽内存，不过那个情况不会经常发生在服务器中。

但是有了连接池，如果你不关闭 JDBC 会话，你将最终用尽池中的连接，那时你的应用将等待直到其他事务释放连接。

这样的问题可能出现的第一个地方是在结束你的 HTTP 事务的过程中。即使你还有 application level 的会话没有关闭，但你必须关闭 JDBC 会话，让它们归还给连接池，以便它们再在其他的 HTTP 会话中使用。

如果你看不出任何明显的泄漏 JDBC 资源的地方，那么问题可能出在 Exception 的代码部分。结束 HTTP 会话的任何 Exception 的处理者都需要释放 JDBC 资源。但这是一个很冒险的形式，因为你可能很少会碰到这样的 exception：允许在 server 启动和应用变慢之间出现这么久的延迟。

虽然这种说法是建立在 IBM 的 WebSphere 应用服务器上下文之上的，但是它应该可以适用于更加宽广的上下文环境。但不论什么平台，你泄漏的 JDBC 资源越多，你泄漏的内存也就越多，而内存问题就会影响到性能！你会需要更频繁的进行垃圾回收，这样就悄悄地占用了你的应用完成其正常工作所需要的内存。

而且，如果你不释放 JDBC 资源，JDBC API 也不会自动的去回收这个资源，可以想像它坐在那里嘲笑你，看着你的代码一点一点地耗尽你内存的每个可用的 byte。而幸运地是，在你的代码中释放 JDBC 资源不是什么问题。在你的测试中，你在测试开始的时候分配连接，然后在测试结束的时候将它关闭。如果你分配了一个 statement，然后在最后将它也关闭了，这样就不会出问题了！

问题是，即使你的测试失败了你也需要释放这些资源！你对于将释放 JDBC 资源的代码放在 finally 里面没有异议。但 finally 的代码可能就不那么合理了：它们自己也许也会抛出 SQLExceptions。这看上去不那么好，一定有一个比较好的方式。

◆ 诀窍

你可以使用下面的这些要点来确保你使用了 JDBC 资源的测试在其使用完那些资源后能够正确的将它们释放，不仅仅是清除数据库中的记录，同时也要清除你用来与数据库交互的 Java 对象。

- 在 setUp() 中创建数据源 (DataSource)。
- 在 setUp() 中创建连接，当然应该是同一个 setUp()，除非你是要测试多连接下的事务行为。
- 在 setUp() 中创建一些集合变量用来存放 resultSet、statement，以及你想要释放的连接——当然应该是为每种资源创建一个集合变量。
- 当你在你的测试中创建 JDBC 资源时，将它们添加到那些你创建的用于存放那些需要释放资源的对象的集合变量中。
- 在 tearDown() 中，针对你在测试中使用的任何资源都调用 close()。请务必首先测试是否为 null，因为你并不知道测试什么时候结束！

在你这样做了两三次之后，你将会发现你的代码的模式简直可以被重构到基本测试用例 (Basic Test Case) 当中，(参看 3.6, “引入一个基本测试用例”)。当然，这里面还有一些重复的内容，它们应当被抽取出来放到工具类当中，但我们做的远不止这些。首先，让我们来看一个例子，我们为 Coffee Shop 的数据库写了两个测试：一个用于核实数据库和表的存在性，另一个用于核实表属性中的 unique 约束。两个测试都需要数据库的支持。清单 10.7 为这两个测试的代码，但从中去除了一些额外的代码。

清单 10.7 Two database schema tests

```
public class CoffeeShopDatabaseSchemaTest extends TestCase {

    public void testTablesAndColumnsExist() throws Exception {

        MimerDataSource coffeeShopDataSource = new
MimerDataSource();

        coffeeShopDataSource.setDatabaseName("coffeeShopData");

        coffeeShopDataSource.setUser("admin");

        coffeeShopDataSource.setPassword("adm1n");
```

```

        Connection connection =
coffeeShopDataSource.getConnection();

        DatabaseMetaData databaseMetaData =
connection.getMetaData();

        ResultSet schemasResultSet =
databaseMetaData.getSchemas();

        List schemaNames = new LinkedList();

        while (schemasResultSet.next()) {

schemaNames.add(schemasResultSet.getString("TABLE_SCHEM"));

        }

        assertTrue(

            CollectionUtil.stringCollectionContainsIgnoreCase(

                schemaNames,

                "catalog"));

        schemasResultSet.close();

        connection.close();

    }

    public void testCoffeeNameUniquenessConstraint() throws Exception {

        MimerDataSource coffeeShopDataSource = new
MimerDataSource();

        coffeeShopDataSource.setDatabaseName("coffeeShopData");

        coffeeShopDataSource.setUser("admin");

```

```
coffeeShopDataSource.setPassword("adm1n");
```

```
Connection connection =  
coffeeShopDataSource.getConnection();
```

```
connection.createStatement().executeUpdate(  
    "delete from catalog.beans");
```

```
PreparedStatement createBeanProductStatement =
```

```
    connection.prepareStatement(  
        "insert into catalog.beans "  
        + "(productId, coffeeName, unitPrice) "  
        + "values (?, ?, ?)");
```

```
createBeanProductStatement.clearParameters();
```

```
createBeanProductStatement.setString(1, "000");
```

```
createBeanProductStatement.setString(2, "Sumatra");
```

```
createBeanProductStatement.setInt(3, 725);
```

```
assertEquals(1,  
createBeanProductStatement.executeUpdate());
```

```
try {
```

```
    createBeanProductStatement.executeUpdate();
```

```
    fail("Added two coffee products with the same name?!");
```



```

        }

        catch (SQLException expected) {

            assertEquals(String.valueOf(23000),
expected.getSQLState());

        }

    }

}

```

代码中的宋体部分重复出现在了两个测试中，所以我们要提取这些相同的部分成为一个测试辅助工具（参看 3.4 节“抽取一个测试模块”）。

我们已经处理完 `connection` 了，现在需要去处理 `statement`，下面是我们一般的方法：

1. 创建一个集合对象用来存放那些需要关闭的 `statement`。
2. 从测试中查找 `statement`，在它们被创建之后和被执行之前将它们加入到上面创建的集合中去。
3. 我们在 `tearDown()` 中增加代码，迭代集合中的 `statement`，然后将它们依次关闭。

当然，我们对 `ResultSet` 也使用同样的三个步骤。清单 10.8 展示了最终的代码。

清单 10.8 在测试模块中处理 JDBC 资源

```

public class CoffeeShopDatabaseSchemaTest extends TestCase {

    public void testTablesAndColumnsExist() throws Exception {

        DatabaseMetaData databaseMetaData =
connection.getMetaData();

        ResultSet schemasResultSet =
databaseMetaData.getSchemas();

        ← 每个测试结束后将
           被关闭

resultSetsToClose.add(schemasResultSet);

```

```

        List schemaNames = new LinkedList();

        while (schemasResultSet.next()) {

schemaNames.add(schemasResultSet.getString("TABLE_SCHEM"));

        }

        assertTrue(

            CollectionUtil.stringCollectionContainsIgnoreCase(

                schemaNames,

                "catalog"));

    }

    public void testCoffeeNameUniquenessConstraint() throws Exception {

        Statement statement = connection.createStatement();

        ← 每个测试结束后将被关闭

statementsToClose.add(statement);

        statement.executeUpdate("delete from catalog.beans");

        PreparedStatement createBeanProductStatement =

            connection.prepareStatement(

                "insert into catalog.beans "

                    + "(productId, coffeeName, unitPrice) "

```

```
+ "values (?, ?, ?)");
```

每个测试结束
后将被关闭

```
statementsToClose.add(createBeanProductStatement);
```

```
createBeanProductStatement.clearParameters();
```

```
createBeanProductStatement.setString(1, "000");
```

```
createBeanProductStatement.setString(2, "Sumatra");
```

```
createBeanProductStatement.setInt(3, 725);
```

```
assertEquals(1,  
createBeanProductStatement.executeUpdate());
```

```
try {
```

```
createBeanProductStatement.executeUpdate();
```

```
fail("Added two coffee products with the same name?!");
```

```
}
```

```
catch (SQLException expected) {
```

```
assertEquals(String.valueOf(23000),  
expected.getSQLState());
```

```
}
```

```
}
```

```
private Connection connection;
```

存储对象关闭

```
private MimerDataSource dataSource;

private List statementsToClose = new LinkedList();

private List resultSetsToClose = new LinkedList();

protected void setUp() throws Exception {

    dataSource = new MimerDataSource();

    dataSource.setDatabaseName("coffeeShopData");

    dataSource.setUser("admin");

    dataSource.setPassword("adm1n");

    connection = dataSource.getConnection();

}
```

关闭每一个可关闭
的对象

```
protected void tearDown() throws Exception {

    for (Iterator i = statementsToClose.iterator(); i.hasNext();) {

        Statement each = (Statement) i.next();

        each.close();

    }

    for (Iterator i = resultSetsToClose.iterator(); i.hasNext();) {
```

```

        ResultSet each = (ResultSet) i.next();

        each.close();

    }

    if (connection != null)

        connection.close();

}

public MimerDataSource getDataSource() {

    return dataSource;

}

}

```

当你知道了如何去做时，你还可以更进一步将 JDBC 资源的释放放到一个单独的类中。你的数据库测试中只需要拥有这个新类——我们将它命名为 `JdbcResourceRegistry`——的一个实例就可以了，在这个类中，你可以将需要释放的 JDBC 资源都添加进去。在 `setUp()` 方法中，创建一个新的资源登记；在 `tearDown()` 方法中调用 `JdbcResourceRegistry.cleanup()` 方法。

最后你还可以将你的数据库辅助功能代码移动到一个数据库辅助功能类中。这些辅助功能代码一般是应用级别或者组件级别的，因为它们有用到你的数据源并且准备了一些必要的数据。我们将 `CoffeeShopDatabaseSchemaTest` 功能推进，形成了另一个测试辅助，我们把它叫做 `CoffeeShopDatabaseFixture`，如清单 10.9 所示。

清单 10.9 `CoffeeShopDatabaseFixture`

```

package junit.cookbook.coffee.jdbc.test;

import java.sql.*;

import junit.framework.TestCase;

```

```

import com.diasparsoftware.java.sql.JdbcResourceRegistry;

import com.mimer.jdbc.MimerDataSource;


// You should only need one database fixture for the entire project

public abstract class CoffeeShopDatabaseFixture extends TestCase {

    private Connection connection;

    private MimerDataSource dataSource;

    private JdbcResourceRegistry jdbcResourceRegistry;


    protected void setUp() throws Exception {

        dataSource = new MimerDataSource();

        dataSource.setDatabaseName("coffeeShopData");

        dataSource.setUser("admin");

        dataSource.setPassword("adm1n");

        jdbcResourceRegistry = new JdbcResourceRegistry();


        connection = dataSource.getConnection();

        getJdbcResourceRegistry().registerConnection(connection);

    }


    protected void tearDown() throws Exception {

        ◀—— 简单，不是吗？      getJdbcResourceRegistry().cleanUp();

    }

```

```
public MimerDataSource getDataSource() {  
    return dataSource;  
}
```

```
protected Connection getConnection() {  
    return connection;  
}
```

```
protected JdbcResourceRegistry getJdbcResourceRegistry() {  
    return jdbcResourceRegistry;  
}
```

便利
的方法

```
protected void registerConnection(Connection connection) {  
    jdbcResourceRegistry.registerConnection(connection);  
}
```

```
protected void registerStatement(Statement statement) {  
    jdbcResourceRegistry.registerStatement(statement);  
}
```

```

    }

    protected void registerResultSet(ResultSet resultSet) {

        jdbcResourceRegistry.registerResultSet(resultSet);

    }

}

```

让我们回顾一下我们所做的工作。在重构我们的数据库测试的过程中，我们建立了一个数据库辅助功能类，其他所有的与数据库相关的测试都可以去继承这个类。如果你是在测试遗留的 JDBC 代码，那么这将是一个很有用的设计。如果你打算把你的 JDBC 代码重构为一个小型的 JDBC 引擎，这个功能类也能够帮助你编写那些支持重构的测试。总之，这个辅助功能类是个好东西。清单 10.10 中的 `CoffeeShopDatabaseSchemaTest` 使用了这个新的辅助功能类，从中你可以看到一些区别。

清单 10.10 `CoffeeShopDatabaseSchemaTest` 使用新的功能

```

package junit.cookbook.coffee.jdbc.test;

import java.sql.*;

import java.util.LinkedList;

import java.util.List;

import com.diasparsoftware.java.util.CollectionUtil;

public class CoffeeShopDatabaseSchemaTest

    extends CoffeeShopDatabaseFixture {

    protected void tearDown() throws Exception {

```



```

        Statement statement = getConnection().createStatement();

        registerStatement(statement);

        statement.executeUpdate("delete from catalog.beans");

        super.tearDown();
    }

    public void testTablesAndColumnsExist() throws Exception {

        DatabaseMetaData databaseMetaData =

            getConnection().getMetaData();

        ResultSet schemasResultSet =
databaseMetaData.getSchemas();

        registerResultSet(schemasResultSet);

        List schemaNames = new LinkedList();

        while (schemasResultSet.next()) {

schemaNames.add(schemasResultSet.getString("TABLE_SCHEM"));

        }

        assertTrue(

            CollectionUtil.stringCollectionContainsIgnoreCase(

                schemaNames,

```

```
        "catalog"));  
    }
```

```
public void testCoffeeNameUniquenessConstraint() throws Exception {  
    Statement statement = getConnection().createStatement();  
    registerStatement(statement);  
  
    statement.executeUpdate("delete from catalog.beans");  
  
    PreparedStatement createBeanProductStatement =  
        getConnection().prepareStatement(  
            "insert into catalog.beans "  
            + "(productId, coffeeName, unitPrice) "  
            + "values (?, ?, ?)");  
  
    registerStatement(createBeanProductStatement);  
  
    createBeanProductStatement.clearParameters();  
    createBeanProductStatement.setString(1, "000");  
    createBeanProductStatement.setString(2, "Sumatra");  
    createBeanProductStatement.setInt(3, 725);  
  
    assertEquals(1,  
        createBeanProductStatement.executeUpdate());  
}
```

```

        try {

            createBeanProductStatement.executeUpdate();

            fail("Added two coffee products with the same name?!");

        }

        catch (SQLException expected) {

            assertEquals(String.valueOf(23000),
expected.getSQLState());

        }

    }

}

```

◆ 讨论

如果你需要这种（几乎）自动清理的工具，可以看一下 GSBase 的 JDBC 资源包装器，可以在 gsbase.sourceforge.net 上找到。这些资源包装器将会在它们使用完这些资源之后将其释放。如果你能改变你需要测试的 JDBC 代码，那么我们推荐使用这些资源包装器，甚至可以将它作为我们这里将要介绍的方法的一个替换措施。

◆ 相关

- ☐ 3.4—抽取一个测试模块
- ☐ 3.6—引入一个基本测试用例
- ☐ GSBase (<http://gsbase.sourceforge.net>)

10.5 Verify your production code cleans up JDBC resources

10.5 核实你的产品代码释放了 **JDBC** 资源

◆ 问题

你想要测试你的生产代码，核实它释放了所有分配给它的 JDBC 资源：resultset、statement 还有 connection。

◆ 背景

好消息是如果在你的应用中，你将所有的用于执行查询的代码都移动到一个地方，正如我们在这个章节前面的部分推荐和描述到的，剩下的工作就不多了。在产品代码中唯一需要释放 JDBC 资源的地方就是那个用于执行查询的代码，这样的话，你只需要在少量的方法中运用我们介绍的那种方法。

坏消息是如果你应用中到处都有对 JDBC 调用的话——而且这应该是现在比较常见的情形吧，那么你将需要做更多的工作。你需要非常仔细地考虑是否要花费更多努力去编写你需要的所有测试还是丢弃你所有的数据访问代码，然后使用我们这里开发出的 JDBC 框架来把它替换掉。仔细的衡量一下——多实施几次这个方法，看看这样做会需要多少时间。重写一个数据访问类，也看看会需要多长的时间，然后比较一下两次的结果。

如果你决定继续测试你所有分散的 JDBC 客户端代码，而不是使用 JDBC 框架，那么我们下面要介绍的方法将会对你有一个好的指导。

◆ 诀窍

你可以使用 Mock Object JDBC 的实现来验证为各种不同的 JDBC 资源所调用的 close()。清单 10.11 是这样测试的一个例子。

清单 10.11 审核 **CatalogStore** 关闭其 **PreparedStatement**

```
package junit.cookbook.coffee.jdbc.test;

import java.sql.*;

import junit.cookbook.coffee.data.*;

import junit.cookbook.coffee.data.jdbc.CatalogStoreJdbcImpl;

import junit.framework.*;

import com.diasparsoftware.java.util.Money;

import com.mockobjects.sql.*;
```

```

public class AddProductTest extends TestCase {

    public void testHappyPathWithPreparedStatement() {

        Product toAdd =

            new Product("999", "Colombiano", Money.dollars(9, 0));

        final MockPreparedStatement addProductStatement =

            new MockPreparedStatement();

        addProductStatement.addUpdateCount(1);

        ← 设置期望值

        addProductStatement.setExpectedCloseCalls(1);

        MockConnection2 connection = new MockConnection2();

        connection.setupAddPreparedStatement(

            "insert into catalog.beans "

                + "(productId, coffeeName, unitPrice) values "

                + "(?, ?, ?)",

            addProductStatement);

        CatalogStore store = new CatalogStoreJdbcImpl(connection);

        store.addProduct(toAdd);

        ← 验证期望值

        addProductStatement.verify();

        connection.verify();

    }

```

```
}
```

我们注意到了用于设置和验证我们测试中的期望（expectation）的代码，它使用了一个常见的 Mock Object 模式。首先我们调用 `setExpectedCloseCalls()` 来决定我们测试的代码应该结束 `PreparedStatement` 多少次——答案是一次。在测试的最后我们调用 `verify()`，这样，如果结果不能够与期望（expectation）匹配，Mock `PreparedStatement` 和 Mock `Connection` 将返回失败。也就是说，如果我们不关闭 `PreparedStatement`——只需一次，或者如果我们设法关闭 `Connection`——测试将失败，我们不希望 `CatalogStore` 关闭 `Connection` 的原因有两个：第一，`Connection` 是由谁获得的就应该由谁去关闭，而 `CatalogStore` 没有获得 `Connection`；第二，我们希望在同一个事务中实现多个存取的并发，这样的话就必须使用同一个 `Connection`，也就意味着 `Connection` 不应该被关闭！

◆ 讨论

我们还没有一个用来验证是否关闭了 `ResultSet` 对象的测试例子，不过这个例子很容易创建。请记住，按照顺序来关闭 `ResultSet`、`Statement` 以及 `Connection` 是十分重要的。但这并不能保证我们是以这种特定的顺序来关闭这些资源的：Mock Object 并没有为验证各种对象所调用的各种方法之间的顺序提供直接的支持。

所以为了保持你的测试完整性，你需要每个 JDBC 客户端代码编写这种类型的测试。想像你的 JDBC 客户端代码是如何设计的：你可能会需要编写成百上千的测试。幸运的是如果你注意到了测试中的编码模式，你就能够将它重构为一个参数化的测试用例（参见 4.8，“创建数据驱动的 Test Suite”）并且，如果设计中包含重复的内容，你就可以从你的系统中提取出几个有代表性的测试用例然后只编写这一部分的测试。如果你还不曾领略到重构带来的好处，那么这里省去的要编写的测试可能会让你有所体会。你现在可以走到你的经理面前说：“我刚刚为我们节省了大约 150 个小时的工作。”可能，她还会问你是如何做到的。

我们看到 JDBC 客户端代码在整个应用程序中到处都是，而我们觉得这是一个严肃的设计问题。你可能已经感觉到了，我们看不起那些制造出这些设计问题的人，并且事实可能会更糟。如果你就是那个编写了导致需要再编写成百上千的测试的数据存取代码的人，也不应该灰心。相反，你应该去重构你的代码让它脱离那个糟糕的模式。在当时的状况下，根据你知道的，你写出了最好的代码。不要气馁，因为你不能够做出你不知道如何去做的事情。那么谁能呢？你可以从这种有点好笑的经历中多多学习，而且我们可以帮助你。

◆ 相关

- 4.8—建立数据驱动的 Test Suite

10.6 Manage external data in your test fixture

10.6 在你的测试功能中管理外部数据

◆ 问题

你想要针对测试数据库，但在每个测试之后数据库的状态都会有一些轻微地变化。

◆ 背景

JUnit 实践者经常说“共享的测试设置散发着臭味”。这是我们用调侃的方式说：在不同的测试之间共享辅助数据将会导致设计上的问题。即使现在没有，将来也会有的（但也不是以后一直存在，因为你总是可以去重构它们的）。这里的“共享辅助”指的是当测试#1 执行，增加了一些数据，然后测试#2 去读测试#1 增加的数据。这打破测试独立性，关于这一点的看法我们应该是清楚的。如果你还不是很清楚，请继续阅读下面的内容。

放弃共享辅助数据，而去编写你的测试以便它们共享相同的起点，并且在这些相互独立的测试中按需求增加或者删除数据。有时，共同的起点是什么也没有，而有时是已知的一套数据。你的这些辅助功能很可能可以都组织起来，这样它们就会被很容易的放到一个单独的类中，并且被很多测试使用。下面我们将介绍如何做。

◆ 诀窍

先做重要的事：提取你针对一个实际运行的数据库的特性所编写的所有测试。参见 3.4 节，关于如何做的更多细节。一旦你拥有了数据库测试的 fixture，那么一般情况下，你可以这样来做。

1. 在 `setUp()` 中连接到数据库并且为其准备好必要的数据库。
2. 在 `tearDown()` 中从数据库删除所有数据。

如果你想做一点改进，你可以在 `setUp()` 中开始一个事务，然后在 `tearDown()` 中使用“回滚”技巧，这样 JDBC 实际上并没有被执行。这样也就没有必要去清理数据了！清单 10.12 展示了一个将所有方式都放在一起的例子。

清单 10.12 使用“回滚”技巧

```
package junit.cookbook.coffee.jdbc.test;
```

```
import java.sql.*;
```

```
public class SelectCoffeeBeansTest extends CoffeeShopDatabaseFixture {
```

← 需要设置超类实体

```
    protected void setUp() throws Exception {
```

```
        super.setUp();
```

```
        Connection connection = getConnection();
```

← 不能“回滚”操作

```
        connection.setAutoCommit(false);
```

```
        PreparedStatement insertStatement =
```

```
            connection.prepareStatement(
```

```
                "insert into catalog.beans "
```

```
                    + "(productId, coffeeName, unitPrice) values
```

```
"
```

```
                    + "(?, ?, ?)");
```

隐藏准备好了
的技术细节

```
        registerStatement(insertStatement);
```

```
        insertCoffee(insertStatement, "001", "Sumatra", 750);
```

```
        insertCoffee(insertStatement, "002", "Special Blend", 825);
```

```
        insertCoffee(insertStatement, "003", "Colombiano", 810);
```

```
    }
```



```
protected void tearDown() throws Exception {
```

← 不暴露文档

```
getConnection().rollback();
```

```
super.tearDown();
```

```
}
```

```
public void testFindExpensiveCoffee() throws Exception {
```

```
Connection connection = getConnection();
```

```
PreparedStatement findExpensiveCoffeeStatement =
```

```
connection.prepareStatement(
```

```
    "select * from catalog.beans where unitPrice >  
2000");
```

```
registerStatement(findExpensiveCoffeeStatement);
```

```
ResultSet expensiveCoffeeResults =
```

```
findExpensiveCoffeeStatement.executeQuery();
```

```
registerResultSet(expensiveCoffeeResults);
```

```
assertFalse(expensiveCoffeeResults.next());
```

```
}
```

```
public void testFindAllCoffee() throws Exception {
```

```
Connection connection = getConnection();
```

```
        PreparedStatement findAllCoffeeStatement =  
            connection.prepareStatement("select * from  
catalog.beans");
```

```
        registerStatement(findAllCoffeeStatement);
```

```
        ResultSet allCoffeeResults =  
            findAllCoffeeStatement.executeQuery();  
        registerResultSet(allCoffeeResults);
```

```
        int rowCount = 0;  
        while (allCoffeeResults.next())  
            rowCount++;
```

```
        assertEquals(3, rowCount);
```

```
    }
```

```
private void insertCoffee(  
    PreparedStatement insertStatement,  
    String productId,  
    String coffeeName,  
    int unitPrice)  
    throws SQLException {
```

```

        insertStatement.clearParameters();

        insertStatement.setObject(1, productId);

        insertStatement.setObject(2, coffeeName);

        insertStatement.setObject(3, new Integer(unitPrice));

        insertStatement.executeUpdate();

    }

}

```

如果你测试的 JDBC 代码需要向数据库提交数据——比方说测试事务行为——那么你就可以在 `tearDown()` 中将 `Connection.rollback()` 替换为必要的 JDBC 代码来删除你插入的数据，或者取消你刚刚做的任何更新。这些“取消”可能很快就会变得很复杂。所以注意把握好分寸。如果你的测试中清除数据要花超过一分钟的时间那么就去尝试其他的办法，因为在测试中不值得花这么长时间去清理数据。从经验上来讲，如果你有大约 40 个到 50 个的测试，而且其中一部分还有复杂的“取消”代码，这些代码的维护对于使用它们所带来的益处显得负担过重，也就是说这是个问题。

你最佳的选择是拥有一个数据库的实例，你随时都可以创建或者销毁它，那样的话在最坏的情况下你的 `tearDown()` 代码只需要删除整个表就可以了。如同一个评论者写到，“如果重建数据库不容易，那我们就让它变得容易。”当然，如果你没有条件这么做，我们也不会丢下你不管——参看 10.7 节“管理测试数据库中的测试数据”

◆ 讨论

如同本章节中许多其他针对实际运行的数据库的测试方法一样，当你有你不能修改的遗留 JDBC 代码，或者你希望创建重构 safety net 的时候，这个技术非常有用。如果你有机会替换你的 JDBC 存取代码，我们建议你使用我们在本章开头描述的方法。理想的话，你不会针对一个实际运行的数据库编写很多的代码，而是将需要数据库的代码隔离出来，然后测试那些不需要实际运行的数据库支持的代码。本章的其他方法中谈论到了相关的技术。

要注意你数据库表中的 IDENTITY 或者自增字段。你的测试运行的次数越多，这个字段的下一个值将会越来越高。如果你像我们希望的那样经常执行你的测试，那么很可能就会导致这个字段的 ID 值达到最大，导致 ID 用尽！如果你认为这个问题很严重，最简单的办法是隔一段时间就重建一次数据库，这样会重新设置 ID。如果你在你的测试数据库中不“拥有 plug”，那么你将需要一个更加老练的策略。参见 10.7 节中的建议。

◆ 相关

- 3.4—抽取一个测试模块
- 10.7—管理测试数据库中的测试数据

10.7 Manage test data in a shared database

10.7 管理测试数据库中的测试数据

◆ 问题

你想要针对某个数据库来测试，但你却不能够访问到这个特定的数据库。

◆ 背景

总是有很多原因使得你不得不去共享数据库。

考虑到数据库平台的 license 的花费问题。购买更多的 license 可能会很贵，并且管理上，在成本/收益的衡量上通常很不清晰，与试图在不容易判断的事情（比方说让所有的程序员都共享一个数据库所带来的效率降低的程度）上减少开支相比，他们宁愿在他们容易判断的事情（比方说 license 的确切价格）上节约开支，虽然很可能在他们不容易判断的事情上需要的投入更多。我们可以抱怨我们想要的，但除非我们能提供确切的成本/收益分析并且确定我们的分析准确性，情况才可能有所改变。

在管理上可能会有一些问题围绕着数据库的“归属权”问题。有一些项目经理认为数据库维护小组必须对数据库全权掌握，否则将会出现混乱。他们很害怕看到程序员向数据库管理小组的人员要求如何构造数据库，这种情况可能会增加两个小组之间的摩擦。经验告诉这些项目经理，为了使得两个小组能够系统工作，数据库小组必须完全拥有和完全控制数据库。但却没有明确的解决这个问题的办法。

无论什么原因，你感觉你自己还是不得不使用一个共享的测试数据库，而且你想知道应该如何来处理这种情况。

◆ 诀窍

在你绝望之前，考虑一下下面的方法。

- 下载一个免费的数据库产品，这样你就有了你自己的数据库。你可以从 Mimer、MySQL、HSQLDB 以及其他的数据库中选择一个。有很多公司都提供免费的数据库产品（至少可以做开发用）使得你可以自由编写你需要的测试。
- 在数据库服务器上为你的测试创建一个独立的表空间或者数据库。这样你就能够做想要做的事情了。这个选择将强制你的 SQL 代码是独立于表空间或者 schema 的，这样就避免了对登录的用户的依赖问题。

□ 在与其他测试人员的数据库测试碰撞的可能性很小的几个小时之内去执行你的针对数据库的测试。这里我们假设，你与其他的程序员小组共享一个测试数据库，通过交叉你们执行测试的时间的方式，你就能够使用同一个 license，而不会导致问题。将数据库的测试的执行限制在几个小时之内将减少你执行你测试的频率，但你的测试的编写将会变得比较困难。

如果你不得不与其他的程序员共享一个唯一的数据库、表空间以及 schema，并且你们的测试将会同时执行，那么你能够做的应对措施就不多了：每个小组都必须保证他们的测验数据不会与其他小组的测验数据冲突。这意味着会有这样的一些规范：“顾客名字使用 A 至 D，而咖啡产品号我们将使用 000 到 099。”你可能需要将所有的这些规范都收集起来，然后把它放在一个醒目的地方，以便所有人都能够随时看见它，一个好的办法是把它挂在网站上。当一些粗心的程序员导致了冲突，你就可以镇静地在网站上指给他们看，并礼貌地要求他更加小心。但是如果他第二次又……

◆ 讨论

虽然将数据库分割成很多小的部分是可行的，但仍需要考虑几个限制：

□ 不将大量的数据包含在你的测试中将可能会使得问题被隐藏起来——如果你正在编写从合同中检索和处理数据的测试，你会发现合同中有一个特别的在第一季度失效的规则。如果你的测验数据只包括从 10 月到 12 的合同月，那么你将永远不会测试到那个特别的规则。

□ 你的数据可能不够用——当你希望用顾客购买 1000 个品种的咖啡豆来进行压力测试的时候会怎样？如果你只有 100 个产品，你就不能编写这个测试了——或者至少你必须与其他小组协调来执行你的测试，当然应该不能在高峰期执行。

□ 你的数据库的主键的 ID 可能会不够用——如果你的数据库存在 ID 字段（并且是自增字段），那么在你每天执行你的测试的 10 到 20 次之后，ID 将很可能超出最大值。诚然，共享测试数据库也许迫使你执行你的测试的频率减少，但这里要谈的并不是这个问题。

□ 你无法测试临界状态——如果你不能够将数据库的表清空，却需要去测试你的代码在表为空的情况下的情形，你应该怎么做？你可以采用 Mock Object 方法（参见 10.9 节，“脱离实际运行的数据库测试遗留的 JDBC 代码”），但并不是每个人都会对这种解决方案满意，而我们并不很介意。

□ 数据冲突很不容易被发现——如果两个测试发生冲突，将会是一片混乱。而实际上是根本没有办法去检测，“其他人正在进行测试。”你应该去找谁？情况会变得多坏？那些问题都没有答案，对于花费在执行测试和防止缺陷上的时间和精力是一种浪费。

我们推荐通过所有必要的手段来实现使用分开的数据库进行测试。我们认为强调它的重要性是很有必要的。

注意： 你需要拥有控制权——Ward Cunningham 在给 Kent Beck 的 *Sorted Collection* 中写到了关于“拥有控制权”的重要性，你可以在 <http://c2.com/doc/forewords-/-beck2.html> 查阅到相应的内容。Ward Cunningham 写到，“程序员的程序是在计算机上，在硬件上运行的，为了完全控制你的程序的运行，你就必须能够控制你运行程序的计算机。因此，你应当针对于某个计算机来编写你的程序，如果你对你的计算机的性能并不满意，那么你可以决定使用另一台计算机”。对于数据库也是一样的，为了能更好的实现代码和数据库的交互，你需要能够 unplug（或者清除）数据库，这样做的结果是测试将不能够继续被执行，但是你省去了与其他程序员共享一个测试数据库所耗费的时间。

◆ 相关

- 10.9——脱离数据库来测试 legacy JDBC 代码

10.8 Test permissions when deploying schema objects

10.8 测试部署 **schema** 对象时的权限

◆ 问题

你注意到了部署 schema 时的一些问题：第一个用户第一次使用它时，很可能会告诉你，你刚刚部署的 schema 对象用不了。

◆ 背景

我们从 Carl Manaster 收到了下面的 email，他简短的描述了这个问题。

“我在开发数据库中编写了一个存储过程，测试它，通过了。然后我就将这个存储过程复制到了正式数据库服务器中，然后又测试了一下，还是好的，于是就把它发布了。但是我忘记了赋予一般用户调用这个存储过程的权限了，因此第一个使用它的用户就来告诉我，这个存储过程不能用。”

如果你有相同的情形，那么你将需要更多的测试，而下面的诀窍将告诉你如何进行你需要的测试。

◆ 诀窍

忘记赋予一般用户操作数据库表或者其他 schema 对象的权限是很常见的问题，SQL 标准中并没有提供一个关于给定的用户是否具有操作某个 schema 对象的权限的支持。因此，你不得不先在没有限权的条件下去执行这个测试。但有个问题，就是你的数据库将如何报告权限不足的问题。

为了找出数据库报告权限不足的方式，我们将使用 Learning Test，首先创建一个存储过程，然后在你的权限下去调用它，之后你的 JDBC 提供商也就是你的数据库将会报告权限问题，清单 10.13 显示了这个测试。

清单 10.13 测试存储过程的特权

```
public class StoredProcedurePrivilegesTest

    extends CoffeeShopDatabaseFixture {

    protected void setUp() throws Exception {

        super.setUp();

        Statement statement = getConnection().createStatement();

        registerStatement(statement);

        try {

            statement.executeUpdate("drop procedure
NOT_ALLOWED");

        }

        catch (SQLException doesNotExist) {

            if ("42000".equals(doesNotExist.getSQLState()) == false)

            {

                throw doesNotExist;

            }

        }

        statement.executeUpdate(
```

```

        "create procedure NOT_ALLOWED() begin end");
    }

    protected void tearDown() throws Exception {
        Statement statement = getConnection().createStatement();
        registerStatement(statement);
        statement.executeUpdate("drop procedure NOT_ALLOWED");
        super.tearDown();
    }

    public void testSeePermissionProblem() throws Exception {
        Connection connection =
            getDataSource().getConnection("programmer",
            "pr0grammer");
        Statement statement = connection.createStatement();
        registerStatement(statement);

        statement.execute("call
NOT_ALLOWED()");
    }
}

```

我们执行了这个测试，然后获得了 `Java.sql.SQLException: The procedure NOT_ALLOWED does not exist (or no execute privilege)`，这些告诉我们要去捕获 `SQLException` 并且需要查看错误代码号以及 SQL 状态码。我们将它们加入到我们的测试中，如清单 10.14 所示。

清单 10.14 加入代码来预期一个 SQLException

```
public void testSeePermissionProblem() throws Exception {  
  
    Connection connection =  
  
        getDataSource().getConnection("programmer", "programmer");  
  
    Statement statement = connection.createStatement();  
  
    registerStatement(statement);  
  
    try {  
  
        statement.execute("call NOT_ALLOWED()");  
  
        fail("User 'programmer' allowed to call NOT_ALLOWED?!");  
  
    }  
  
    catch (SQLException e) {  
  
        assertEquals(0, e.getErrorCode());  
  
        assertEquals("", e.getSQLState());  
  
    }  
  
}
```

我们能够确定我们刚刚加入的断言将返回失败，但是当它失败后我们就能够将测试中的期望值替换为正确的值。这也正是 Gold Master 的一种运用方式。当我们找到这两个期望值后——这两个值根据数据库的不同而不同，因此不要将这里的错误代码号复制到你的测试中去——我们将 catch 块修改如下：

```
try {  
  
    statement.execute("call NOT_ALLOWED()");  
  
    fail("User 'programmer' allowed to call NOT_ALLOWED?!");  
  
}  
  
catch (SQLException e) {
```

```

        assertEquals(-12743, e.getErrorCode());

        assertEquals("42000", e.getSQLState());
    }

```

现在我们有了一个测试是在用户有权限的条件下试图去调用某个特定的存储过程，而且这个测试会返回失败。我们这样做的目的是为了了解 Mimer 是如何报告权限不足的问题的。然后我们就可以编写我们确切需要的测试，这个测试将使用刚才得到的信息来作为测试的期望值，清单 10.15 的测试验证了一个拥有权限的用户去调用一个特定的存储过程的情况。

清单 10.15 StoredProcedurePrivilegesTest

```

package junit.cookbook.coffee.jdbc.test;

import java.sql.*;

public class StoredProcedurePrivilegesTest
    extends CoffeeShopDatabaseFixture {

    // setUp and tearDown omitted

    public void testCanCall() throws Exception {

        Connection connection =
            getDataSource().getConnection("programmer",
            "pr0grammer");

        Statement statement = connection.createStatement();

        registerStatement(statement);

        try {

```

```

        statement.execute("call NOT_ALLOWED()");
    }

    catch (SQLException e) {

        if (isNoPrivilegesException(e))

            fail("User 'programmer' cannot call procedure "

                + "NOT_ALLOWED");

        else

            throw e;

    }

    finally {

        connection.close();

    }

}

}

private boolean isNoPrivilegesException(SQLException e) {

    return (-12743 == e.getErrorCode())

        && ("42000".equals(e.getSQLState()));

}

}

```

← “不可预测的异常” 信号

❌ 数据库不可用的异常

我们很少决定去捕获不在期望中的异常，但在这里我们将会这样做——然后让测试返回失败而不是将这个异常向上传递到 JUnit 框架。不过，这纯粹是一个个人习惯的问题，在这种情况下，我们也可以简单的报告权限不足。我们能借助于 Mimer 的错误信息来报告这个信息，但是我们这里使用的方式能够保证即使 Miner 更改了，我们的信息也仍然有效。

下一步是从这个测试中提取出一个测试引擎，然后针对所有有权限的用户去调用存储过程。这个测试的输入是由用户名称、存储过程和执行这个存储过程所需要的权限组成的。你可以从表 10.1 中看到一个访问控制清单的简单例子。这些数据将用于你的测试。

表 10.1 访问控制清单的简单例子

User	Description	Stored procedure	Allowed to execute?
Admin	Administrator	addProduct	Yes
Csr	Customer service representative	addProduct	No
Clerk	Data entry clerk	addProduct	No
Marketing	Marketing professional	addProduct	Yes

现在你有了一套表格化的数据，这样你就可以创建一个参数化的测试用例（参看 4.8 节“创建数据驱动的 Test Suite”），其中的测试所需要的测试数据就是表中的各行。将这个表格化的数据转化为文件形式，这样就使得测试数据与你要测试的存储过程很容易的保持同步。

◆ 讨论

这里讲到的这个诀窍中有一个很大的问题，就是严格的管理可能会禁止你在正式的服务器上去执行这些测试，但这正是你需要做的！坦白的讲，这里我们并没有办法帮你解决这个问题，因为我们自己在有效的谈判方面也并不见长。你能够做的就是在不影响正式数据库的其他应用的情况下来执行你的测试。当然，如果测试的执行对数据库有比较大的影响，那么不允许你去执行这些测试将是正确的。这就是一个需要去测试一个测试的地方。

尽管我们这里的例子是关于测试一个存储过程的权限问题，但在你的测试中你应当测试所有的 schema 对象的权限，而不仅仅是存储过程。

◆ 相关

□ 10.12—测试存储过程

10.9 Test legacy JDBC code without the database

10.9 脱离数据库来测试 legacy JDBC 代码

◆ 问题

你现在拥有一些遗留的 JDBC 代码，然后你又想在不借助于数据库的情况下对其进行测试。

◆ 背景

有一个问题——虽然不是由于你的疏忽造成的，但你必须去面对的问题——就是你不能将我们本章中讲到的重构方法应用到你要测试的 JDBC 代码中。我们觉得非常遗憾，但我们觉得这些还是能够对你有所帮助的。下面的诀窍中介绍了如何在不借助于数据库的情况下使用 Mock Object (<http://www.mockobject.com>) 来测试 JDBC 调用。

◆ 诀窍

在开始之前，我们向你推荐一篇关于 Mock Object 的文章 *Developing JDBC Applications Test First* (www.mockobjects.com/wiki/DevelopingJdbcApplicationsTestFirst)。即使你并不是首先编写你的应用测试，这篇文章也能够提供一些很好的关于 Mock JDBC API 的例子，这些 API 是由 Mock Object 提供的。我们并不希望在这里重复的将这些文档陈列出来。相反，我们将展示一个使用 Mock JDBC API 对我们的 CatalogStore 的 JDBC 实现进行测试的例子。毕竟，可能你此时此刻不能够上网。

让我们来回顾我们在 10.2 节“验证你的 SQL 语句”中用到的例子。在这里，我们将不仅仅去验证 SQL 语句的字符串，我们还会增加断言验证使用 JDBC API 的正确性。这里同样，假设我们想要测试的 JDBC 代码不会改变，在遗留代码中，管理上很害怕这些代码被修改。我们向我们的 CatalogStore 的 JDBC 实现提交一个 mock 数据源。这个数据源已经被预先由 mock 的 Connection 以及 mock 的 PreparedStatement 初始化，这里我们不需要去过多的关注这些 mock 对象——这些对象最终应该会使你想知道究竟是要测试什么——这里的重点是去验证你的 CatalogStore 的 JDBC 实现知道如何去与 JDBC API 的类交互。清单 10.16 展示了向 catalog 中添加一种咖啡豆的测试。

清单 10.16 使用 Mock 对象 JDBC API 的数据库测试

```
public void testAddProduct() {  
  
    Product toAdd =  
  
        new Product("999", "Colombiano", Money.dollars(9, 0));  
  
    MockDataSource dataSource = new MockDataSource();  
  
    MockConnection2 connection = new MockConnection2();  
    connection.setExpectedCloseCalls(1);  
  
    final MockPreparedStatement addProductStatement =  
  
        new MockPreparedStatement();
```

```

addProductStatement.setExpectedClearParametersCalls(1);

    addProductStatement.addExpectedSetParameters(
        new Object[] { "999", "Colombiano", new Integer(900)});

addProductStatement.addUpdateCount(1);

addProductStatement.setExpectedCloseCalls(1);


dataSource.setupConnection(connection);

connection.setupAddPreparedStatement(
    "insert into catalog.beans "
        + "(productId, coffeeName, unitPrice) values "
        + "(?, ?, ?)",
    addProductStatement);

CatalogStore store = new CatalogStoreJdbcImpl(dataSource);

store.addProduct(toAdd);


addProductStatement.verify();

connection.verify();

dataSource.verify();

}

```

这个测试的大部分工作是初始化，这在基于 Mock 对象的测试中是很普遍的。我们创建了一个 mock 数据源，mock Connection，mock prepared statement。我们希望这个 prepared statement 是以如下方式被使用的：

1. 这个 CatalogStore 将会调用一次 clearParameters()方法。
2. CatalogStore 将根据我们希望添加到 Catalog 中的 Product 对象来设置相应的参数。请注意 Money 中的单价属性应该是以分为单位的。

3. CatalogStore 根据 MockPreparedStatement 的 updateCount 属性所标识的内容更新一条记录。
4. CatalogStore 关闭一次 statement。

类似的，connection 以及 datasource 也会有特定的被使用的方式。当我们执行使用这些 JDBC 对象的操作（这里就是 addProduct()）之后，我们要求它们对自己进行验证，并且如果它们的期望没有达到，要及时响应。所有这些都没有使用一个真正的数据库。

◆ 讨论

在这些测试中，我们仍旧对一件事情不满意，那就是即便是我们没有引入数据库，测试本身依然是脆弱的：每个测试都依赖于你的 JDBC 客户端代码的正确性以及它正确的映射数据的能力。同时执行两项操作的代码是很容易造成混乱的。我们更希望对不同的操作进行单独的测试，当然我们知道在遗留代码中你没有别的选择。使用 Mock 对象将会为这个问题提供一种相同的机制，但是如果你有机会，你不应该停止在这里。

我们建议，如果设计上能够实现，将面向业务的接口从 JDBC 客户端代码中提取出来。也就是说，将那些仅仅为了表达领域概念的方法、参数以及返回值都提取出来。比方说，假设你有一个数据访问对象，它可以查询到所有的账户将会在 30 天后过期的客户，然后你从中提取出 CustomerStore 接口，其中包含一个方法 findPastDue(int days)，然后将你的遗留 JDBC 代码放在对这个接口的实现中，比方说叫 CustomerStoreLegacy-Impl。下面你就可以认为你的对“Store”的遗留实现是一个巨大的黑盒子。期间你可以通过执行你实际的测试来部分的替换它！而且，你能在你空闲的时候来完成它，不必赶时间。对遗留代码进行全面的重构可能会需要很长时间，但至少你知道了是可行的——只不过是时间的问题。

◆ 相关

- 10.2—验证你的 SQL 语句
- Mock Object 项目 (www.mockobjects.com)
- 首先开发 JDBC 应用的测试
(www.mockobjects.com/wiki/DevelopingJdbcApplicationsTestFirst)

10.10 Test legacy JDBC code with the database

10.10 联合数据库测试遗留的 JDBC 代码

◆ 问题

你通过继承得到了遗留的 JDBC 代码，现在你希望联合数据库来对其进行测试。

◆ 背景

如果你打算联合一个实际运行的数据库来测试你遗留的 JDBC 代码，请首先确定你拥有测试数据。参看 10.7 节一些相关的内容。然后你需要一种机制来完成测试数据的建立。我们已经尝试过使用 JDBC 代码本身来建立测试数据，但这条经验算不上是一条优秀的经验。我们需要去维护大量的数据，仅仅是为了建立一个根本不起作用的环境。幸运的是我们已经从经验中总结出一些有用的东西，下面我们将它们传授给你。

◆ 诀窍

在这个诀窍中我们假设你想要在不进行重构的情况下测试你的 JDBC 代码，并且你拥有一个能够由你支配的测试数据库。这个方法很直接：为你要执行的每套测试创建数据集，然后使用 DbUnit (<http://dbunit.sourceforge.net>) 来在文件系统上管理你的测试数据。

DbUnit 提供了将测试数据存储为简单的文本数据的功能。这样就减少了重复使用 JDBC 代码建立数据库中数据，因为我们这里只需要将测试数据文件拷贝一份就可以了。你可以以 XML 文件的形式来存储你的数据，或者建立一个能够插入到你的测试中的数据集文件。下面是一个使用“简单 XML 格式”的例子——就是使用 XML 格式来标识数据，其中每个标签表示一张表以及其中的数据。

```
<?xml version="1.0" ?>
```

```
<dataset>
```

```
  <catalog.beans productId="000"
```

```
    coffeeName="Sumatra"
```

```
    unitPrice="750" />
```

```
  <catalog.beans productId="001"
```

```
    coffeeName="Special Blend"
```

```
    unitPrice="825" />
```

```
  <catalog.beans productId="002"
```

```
    coffeeName="Colombiano"
```



```
unitPrice="925" />
```

```
</dataset>
```

这个例子很小，里面标识了三种咖啡。每种咖啡都是表 `catalog.beans` 中的一条拥有三个属性的记录：`productId`、`coffeeName` 以及 `unitPrice`。你的数据集不会有体积限制或者其他的复杂性。

为了在 JUnit 测试中使用 DbUnit 数据集，你可以继承 `org.dbunit.Database- TestCase`，然后重载两个方法来帮助 JUnit 提取数据：`getConnection()`，将返回数据库的一个 `connection`；以及 `getDataSet()`，将返回你的数据集的描述。在你执行你的测试之前，DbUnit 将把你的数据集中的数据加载到数据库中。清单 10.17 中展示了 `DatabaseTestCase` 使用刚才的示例数据集的情况。

清单 10.17 使用 DbUnit 数据库的一个测试案例

```
public class FindProductsTest extends DatabaseTestCase {

    private DataSource dataSource;

    private JdbcResourceRegistry jdbcResourceRegistry;

    public FindProductsTest(String name) {

        super(name);

    }

    protected void setUp() throws Exception {

        jdbcResourceRegistry = new JdbcResourceRegistry();

        super.setUp();

    }

    protected void tearDown() throws Exception {

        jdbcResourceRegistry.cleanUp();

        super.tearDown();

    }

}
```

```
}
```

```
private DataSource getDataSource() {  
    if (dataSource == null)  
        dataSource =  
CoffeeShopDatabaseFixture.makeDataSource();  
    return dataSource;  
}
```

```
private Connection makeJdbcConnection() throws SQLException {  
    Connection connection = getDataSource().getConnection();  
    jdbcResourceRegistry.registerConnection(connection);  
    return connection;  
}
```

```
protected IDatabaseConnection getConnection() throws Exception {  
    Connection connection = makeJdbcConnection();  
    return new DatabaseConnection(connection);  
}
```

```
protected IDataset getDataSet() throws Exception {  
    return new FlatXmlDataSet(  
        new File("test/data/datasets/findProductsTest.xml"));  
}
```

```

public void testFindAll() throws Exception {

    Connection connection = makeJdbcConnection();

    CatalogStore store = new CatalogStoreJdbcImpl(connection);

    Set allProducts = store.findAllProducts();

    assertEquals(3, allProducts.size());

}
}

```

参看 10.4 节“确定测试释放了 JDBC 资源”关于对 `JdbcResourceRegistry` 的讨论，其中最重要的方法是 `getDatabaseConnection()`，以及 `getDataSet()`。`getDatabaseConnection()` 向数据源请求获得一个 `connection`，而 `getDataSet()` 负责从你的硬盘中的 XML 文件中将数据集读取出来。如果在代码中构建一个 `DefaultDataSet`，而其长度可能会超过 10 行，那么我们建议将你的数据集存储为外部文件，在这里，希望将测试数据包含在测试中与希望将令人厌烦的资源放到测试代码之外的两种想法产生了冲突。尽管这里的测试数据不是令人厌烦的资源，但需要你去编写解析这些数据的代码将可能会使得你厌烦，因此，我们仍然建议你去分别尝试一下两种方法。我们觉得你最终会选择将数据集都存储在硬盘上。

◆ 讨论

如果你现在正好有通过 JDBC 代码来建立测试数据的测试，我们建议你使用 `DbUnit` 来修改其中的一个，然后比较这两种方法。你会很明显地发现，`DbUnit` 才是你要的，尤其是当你考虑到不能够在测试中重构你的 JDBC 代码时。避免测试初始化代码与测试中的 JDBC 代码的重复的唯一方法是将你的产品代码中的 SQL 语句暴露给你的测试，但是如果你不能够去重构你的测试代码的话，就没有直接的方法来使得这些 SQL 语句能够被访问到。这样的话，你就不得不在你的测试中将 SQL 语句以及相应的 JDBC 代码再复制一次。显然这样做并不值得。

注意： `DbUnit` 限制——如果你的数据库的表的属性中存在自增字段或者 `IDENTITY` 字段，你可能需要在使用 `DbUnit` 来向你的表中添加数据之前将它们 `disable` 去掉。在本书出版的时候，`DbUnit` 只支持 MS SQL Server 中的 `IDENTITY` 字段。更多详细内容参见 `DbUnit` 的网站中的 FAQ 部分。

◆ 相关

- 10.4—确定测试释放了 JDBC 资源
- 10.7—管理测试数据库中的测试数据
- `DbUnit` (<http://dbunit.sourceforge.net>)

10.11 Use schema-qualified tables with DbUnit

10.11 联合 DbUnit 使用 **schema-qualified** 的表

◆ 问题

你希望使用 DbUnit 来存储数据集以便测试遗留数据库中多个 schema 里面的表。当你执行测试时，DbUnit 却没有找到相应的表。

◆ 背景

尽管这里讲到的诀窍在 DbUnit 的网站上有明确的讲述到 (<http://dbunit.sourceforge.net>)，但由于你的迫不及待可能让你直接在一个有多个 schema 的数据库上去使用 DbUnit。还有另外一种可能性就是，当你在一个只有一个 schema 的数据库中使用 DbUnit 的时候，发现没有问题，然后你就转向一个具有多个 schema 的数据库，但却发现，先前正常的功能现在已经不起作用了。

◆ 诀窍

你需要在这个测试从数据库取得 connection 之前，通过将系统的属性——`dbunit.qualified.table.names` 的值设置为 `true` 来取消对 schema 的限制。因此，我们建议将它们放在你的测试中的 `setUp()` 方法中，如同下面的一样：

```
public class FindProductsTest extends DatabaseTestCase {

    // other code omitted

    protected void setUp() throws Exception {

        System.setProperty("dbunit.qualified.table.names", "true");

        super.setUp();

    }

    protected IDatabaseConnection getConnection() throws Exception {

        return new DatabaseConnection(
```

```

CoffeeShopDatabaseFixture.makeDataSource().getConnection());

    }

    protected IDataset getDataSet() throws Exception {

        return new FlatXmlDataSet(

            new File("test/data/datasets/findProductsTest.xml"));

    }

    // tests omitted

}

```

现在你可以在你的数据集中指定有 schema 的表名称了，就像下面的一样。

```
<?xml version="1.0" ?>
```

```
<dataset>
```

```

    <catalog.beans productId="000"

        coffeeName="Sumatra"

        unitPrice="750" />

    <catalog.beans productId="001"

        coffeeName="Special Blend"

        unitPrice="825" />

    <catalog.beans productId="002"

        coffeeName="Colombiano"

```

```
unitPrice="925" />
```

```
</dataset>
```

为了完整的表示，清单 10.18 给出了一个简单的使用了这种数据集的测试。

清单 10.18 使用 schema-qualified 数据库

```
public class FindProductsTest extends DatabaseTestCase {
```

```
    private DataSource dataSource;
```

```
    private JdbcResourceRegistry jdbcResourceRegistry;
```

← 和 JUnit 4 的旧版本
兼容

```
    public FindProductsTest(String name) {
```

```
        super(name);
```

```
    }
```

```
    protected void setUp() throws Exception {
```

```
        System.setProperty("dbunit.qualified.table.names", "true");
```

← 跟踪 JDBC 资源
并清除它们

```
        jdbcResourceRegistry = new
```

```
JdbcResourceRegistry();
```

```
        super.setUp();
```

```
    }
    ←
```

```
    protected void tearDown() throws Exception {
```

```
        jdbcResourceRegistry.cleanUp();
```

```
        super.tearDown();
```

```
}
```

```
private DataSource getDataSource() {  
    if (dataSource == null) {  
        dataSource =  
CoffeeShopDatabaseFixture.makeDataSource();  
    }  
    return dataSource;  
}
```

```
private Connection makeJdbcConnection() throws SQLException {  
    Connection connection = getDataSource().getConnection();  
    jdbcResourceRegistry.registerConnection(connection);  
    return connection;  
}
```

```
protected IDatabaseConnection getConnection() throws Exception {  
    return new DatabaseConnection(makeJdbcConnection());  
}
```

```
protected IDataset getDataSet() throws Exception {  
    return new FlatXmlDataSet(  
        new File("test/data/datasets/findProductsTest.xml"));  
}
```

```

public void testFindAll() throws Exception {

    Connection connection = makeJdbcConnection();

    CatalogStore store = new CatalogStoreJdbcImpl(connection);

    Set allProducts = store.findAllProducts();

    assertEquals(3, allProducts.size());

}
}

```

◆ 讨论

这仅仅是我们意外遭遇中的一个。在你没有遇到这个问题的情况下，顺利的使用 DbUnit 的时间越长，当你最终遇到这个问题时，你越会觉得糟糕，因为那时你可能已经以你对这个包的理解建立起了安全机制。这正是使用 pairprogramming 会很有帮助的一种情况：它来检查关于对你的工作环境的假设，这样常常会让你更快的发现问题的所在。我们都经历过在自己强烈的愿望下独立去寻找一个愚蠢问题的诀窍。培养自己发现在恰当的时间去寻求帮助是很重要的，那样你就能够避免在那个问题上低效率的去摸索诀窍。

◆ 相关

□ DbUnit (<http://dbunit.sourceforge.net>)

10.12 Test stored procedures

10.12 测试存储过程

◆ 问题

你希望测试存储过程。

◆ 背景

当 你的应用中通过存储过程来实现你的一些业务逻辑时，你就必须使用一个实际运行的数据库来测试这些业务逻辑。我们知道这允许数据库对业务规则进行重构来缩短 响应时间；但是，这却使

得对其测试变得更加困难。我们建议将业务逻辑放在它最容易被测试的地方。这应当是你的软件中十分重要的一点。

相反，如果在你的应用中使用存储过程仅仅是为了通过一个 CRUD 接口来将数据库 schema 的具体详细内容隐藏起来，我们会觉得这是个不错的方法。这些存储过程减轻了数据访问层生成 SQL 语句的压力，而只剩下数据映射。我们认为这是一个明智的选择，这使得测试变得容易！

但此时要你用相同的方式来实现你的应用是不大可能的，但是如果可以的话，我们推荐你从第一种方式重构到第二种方式。

◆ 诀窍

如果可以，不要使用 JUnit 来测试存储过程。我们发现 JDBC 对于执行这样一个简单的任务来说过于详细冗长。相反，我们可以使用 shell 脚本以及你的数据库供应商的 SQL 语句命令行工具来做测试。请不要担心是否存在一个关于 shell 脚本的框架，我们使用简单的方法来测试：向数据库发出一些 SQL 语句，然后将查询或者结果与已知的特定值比较，然后错误代码号或者错误级别来表示测试的通过或者失败。我们推荐下面的这些书来作为你使用 shell 脚本做测试的参考：Timothy Hill 的 *Windows NT Shell Scripting* 以及 David Tansley 的 *Linux and Unix Shell Programming*。

注意： *BashUnit*？——Curtis Cooley 讲述了一个关于为 Unix shell 编写测试的故事。故事的大体是这样的：你不需要什么框架；你需要的是测试。然后框架就产生了。

“我刚刚完成了一项有趣的试验。我和一个 DBA 在进行极限编程的时候，他正在试图使得一组 shell 脚本能够工作。这些脚本是数据库的安装以及备份的一部分。”

他竭尽全力想去弄清楚它们是如何工作的，以及它们执行目的是什么。我就笑到，“我们应该对它们进行单元测试。编写出一个 shell 框架应该没有那么困难”。迷惑了数分钟后，他说到，“就这么做吧”。

在接下来的一天半里，我们为这些脚本编写了单元测试，以这种方式建立起了一个小的单元测试框架。

这些测试发现了脚本上的很多错误，而且会需要花比较长的时间来将这些错误清理掉。

在一天半的时间里，我们将另一个 DBA 花了两个星期完成的脚本完全重写了，并将其进行了测试。

你是否由于某种原因被要求使用 Java 来对你的存储过程进行测试，那么你能做的就是建立一些数据，调用 CallableStatement，然后检查结果。参看 10.10 节，“联合数据库测试遗留的 JDBC 代码”。注意维护好你的测试数据以及 JDBC 资源的释放。

如果你想要验证你的业务逻辑调用存储过程的正确性，我们建议你使用 MockCallableStatement 对象来进行测试。验证 callable statement 是否接受到了恰当的参数，以及调用代码是否能够处理这个

callable statement 返回的不同类型的参数。清单 10.19 展示了一个例子，例子中验证 CatalogStore 通过调用特定的存储过程来添加一个产品的过程。

清单 10.19 **AddProductTest** 使用存储过程

```
public class AddProductTest extends TestCase {

    public void testHappyPath() {

        Product toAdd =

            new Product("999", "Colombiano", Money.dollars(9, 0));

        MockDataSource dataSource = new MockDataSource();

        final MockCallableStatement addProductStatement =

            new MockCallableStatement();

        addProductStatement.setExpectedClearParametersCalls(1);

        addProductStatement.setExpectedCloseCalls(1);

        addProductStatement.addExpectedSetParameters(

            new Object[] { "999", "Colombiano", new Integer(900)});

        addProductStatement.addUpdateCount(1);

        MockConnection2 connection = new MockConnection2() {

            public CallableStatement prepareCall(String sql)

                throws SQLException {
```

验证正确地
设置参数

验证SQL
字符串

```

        Assert.assertEquals("call addProduct(?, ?, ?)", sql);

        ◀ 验证SQL字符串
        return addProductStatement;

    }

};

connection.setExpectedCloseCalls(1);

dataSource.setupConnection(connection);

CatalogStore store =

    new CatalogStoreStoredProcedureImpl(dataSource);

store.addProduct(toAdd);

addProductStatement.verify();

◀ 执行事务逻辑
connection.verify();

dataSource.verify();

}

}

```

我们可以将这个测试分为三步：

1. 建立数据源——创建一个 Mock 的数据源，让它能够返回一个 mock 的 callable statement。这个 statement 是通过特定的参数建立的，对 CatalogStoreStored- ProcedureImpl 将 domain 级对象 Product 映射到存储过程中的参数（其中包括了将 Money 对象转换为 Integer）的能力进行测试。请注意，使用 mock 对象，这里将会需要做比较多的工作。

2. 执行业务逻辑——调用 `addProduct()` 方法。这样最终会调用 `MockConnection2.preparedCall()`，这个方法会验证 SQL 语句的正确性。另外，`addProduct()` 方法会将适当的参数传递给我们上一步建立的 `MockCallableStatement`。
3. 验证 JDBC 对象——这是使用 mock 对象的标准：对你在测试开始时设置的每个对象验证所有的期望。

我们在 mock JDBC 对象中设置的各种不同的期望值确保了 catalog store 将 callable statement 的参数重置，以及执行这个 statement，更新一条记录，然后将 statement 和 connection 都关闭。

◆ 讨论

我们并不推崇使用 JUnit 来测试存储过程，原因很简单，使用 Java 来编写这个测试会在测试人员与测试对象（代码）中间增加一层不必要的复杂逻辑。我们在本章讲述过，添加这层不必要的复杂逻辑，还会需要我们做更多的工作。使用 JUnit 对存储过程进行测试还会包括：（1）向数据库中插入数据，调用存储过程，然后清理掉刚才对数据库的变动。或者（2）建立起 mock JDBC 对象，调用存储过程，然后验证这些 mock JDBC 对象。

最后产生的 JDBC 代码以及 mock 对象的代码却毫无用处，仅仅是一些与 SQL 有关的，包括数据类型转换、处理 null 值以及其他功能的冗长的代码——许多程序员都对这些 JDBC 的东西感到厌烦。总的来说，测试存储过程的工作最好交给以数据库为中心的工具，最接近的就是 shell 脚本。

如果由于某种原因，你必须要使用 Java 来对其进行测试，那就想尽一切办法做吧，但是经验告诉我们使用 shell 脚本来做测试是最佳的选择。

当然，不要错误的理解为“不要测试你的存储过程”。代码就是代码，代码是需要被测试的；不然，当你的应用出现问题的时候，你怎么会知道究竟是你的应用中的问题还是你的存储过程的问题？如果你还不能够使用 shell 脚本来测试它们，那么你还是先使用 JUnit 来测试吧，但是我们相信如果立刻学习如何使用 shell 脚本来测试它们，会节省很多时间和精力。

◆ 相关

- 10.10—联合数据库来测试遗留的 JDBC 代码
- Timothy Hill, *Windows NT Shell Scripting* Sams, 1998
- David Tansley, *Linux and Unix Shell Programming*, Toronto, Canada: Pearson Education, 2000

本章主要内容：

- 测试 web 应用页面流，包括 Struts

- 测试你的网站以寻找失效的链接
- 测试 web 和 EJB 资源的安全
- 测试容器管理下的事务处理

当你阅读这本书的时候，有一个观点会越来越清晰，那就是我们提倡通过彻底地测试各组件来测试一个应用，随后以最简单的方式整合这些组件。特别地，“整合”对于我们来讲最多不过是选择使用众多接口中的哪个实现，随即据此创建应用入口对象。我们使用哪种日志策略？Log4J 如何！我们知道我们组件要与日志策略接口的任意实现一起工作。何种模式？一个基于 JDBC 的模式，控制器只需要知道模式的接口，因而一个内存中的实现，或一个基于 Prevayler (www.prevayler.org) 就可以完成。对于我们来说，这就是整合。因此，我们倾向于不强调端到端的测试，而通过构造完成我们需要的特征来获得自信。Object Test 告诉你你是否正确地构造了东西；然而，端到端测试帮助你判断你是否构造了正确的东西。

J2EE 应用的某些方面，人们使用端到端测试而非 Object Test。包括：页面流——或者 web 应用间导航——和使用容器的服务，如安全和事务处理。本章我们将讨论这些主题的解决方案，告诉你怎么隔离地测试这些行为——Object Test。我们不想给你一种我们畏惧端到端测试的印象——模拟终端用户通过终端用户接口与之交互的方式测试一个应用。如同我们早先写的，与其他程序员相比，我们使用端到端测试充当一个另外角色：使用端到端测试帮助我们确定是否构建了用户真正需要的东西。我们将讨论使用 HtmlUnit 为 web 应用书写端到端测试（见 13.1），但不再视 JUnit 为一种端到端测试的最佳可用工具。我们使用 Fit (<http://fit.c2.com>) 及其伴侣工具，FitNesse (www.fitnesse.org)。

注意：当然我们不是唯一的把端到端测试看成此种角色的人：这个观点基本上来自于 Agile community (www.agilealliance.org)。不过 Agile 开发者仍旧是少数，各组织仍将继续把端到端测试作为其批准软件的主要工具。我们认为端到端测试根本就是浪费资源，而这些资源本可以被更好地用于通过程序员测试内在的质量以保证其正确性。

因为这本书是关于 JUnit 的，而不是写 Fit 的，所以我们将简要地描述 Fit。设想将测试整个地写成电子数据表和文字处理器文件。你可以以平实的语言来注释你的测试，描述他们在验证什么——把代码和文本混合在一起以便程序员和非程序员都可以理解。现在想像将那些测试在某个软件中执行，这个软件将用绿色标注正确的部分（测试通过），用红色标注错误的部分（测试失败）。这就是 Fit，它还可以让那些具有业务知识的人写出可执行的测试，即使他们不是程序员。FitNesse 是 Wiki (www.wiki.org) 的一个开源项目，它可以执行 Fit 测试程序，为组织这些测试程序和有效地运行它们提供一个绝好的方式。许多人将 FitNesse 作为“他们执行端到端测试的方式”我们也在他们之中。

但这是一本关于 JUnit 的书，并且这是一个从 J2EE 应用的角度写 Object Test 的章节，J2EE 应用通常是从端到端测试的。这里我们提供了一个解决方案的集合，以帮助你更加有效地测试 J2EE 应用的某些方面。这些行为倾向于广泛出现在应用中：页面流，关闭链接，安全措施，事务处理，和 JNDI。我们增加了一个与 Struts 应用框架 (<http://jakarta.apache.org/struts>) 有关的解决方法，但对于多数的 Struts 应用测试，我们推荐 StrutsTestCase 项目 (<http://strutstestcase.sourceforge.net>)。它同时提供了一个 mock 对象方法（在应用服务器之外测试）和一个 Cactus 方法（在应用服务器里面测试）。它具体实现了我们这本书中描述到许多技术，我们并不想复制他们的美好工作，我们向你推荐他们。

13.1 Test page flow

13.1 测试页面流

◆ 问题

你想要确认用户在浏览你的 web 应用程序时，页面之间的流转的正确性，而且你也希望同时将整个应用程序自身的机制包含在测试内。

◆ 背景

即使你会执行能够发现页面流跳转错误的端到端测试，在不涉及应用程序的业务逻辑、表现层、外部资源等的情况下确认页面流的正确跳转会容易很多。我们建议，将你的 web 应用程序转换为一张大的状态图，在这张状态图里针对页面流跳转我们只需要关心两件事情：1) 用户请求的资源。2) 返回给用户的资源。图 13.1 是一个示意图。方框代表页面，箭头代表请求。

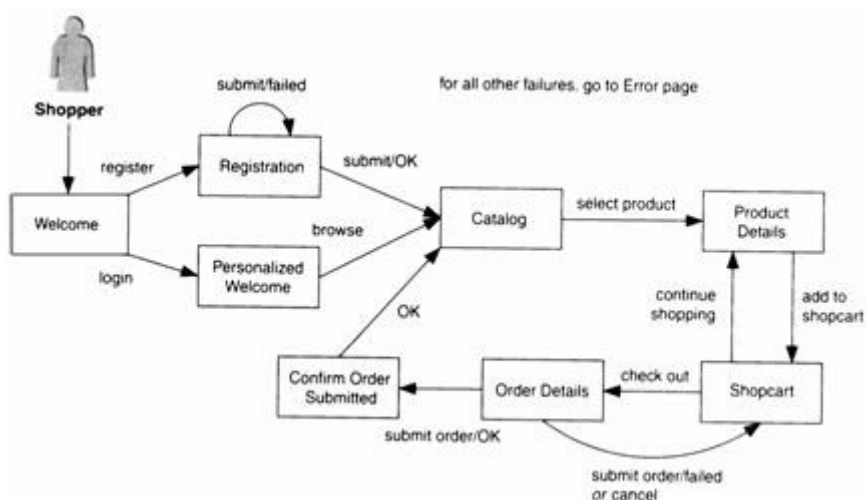


图 13.1 页面流示意图

一些请求只会有一种返回结果，比方说点击一个超链接来浏览目录，多数情况下，这种请求不会失败——当你点击超链接，页面跳转到目录的页面。另一些请求会有多种可能返回结果，比如图中的“OK”和“Failed”。图中注册的页面，当提交表单失败的时候，用户将停留在注册的页面，我们将这种“箭头”标记上“请求/返回结果”的形式，这样“submit/OK”就代表当提交表单执行“submit”操作并且成功的话，指向这个返回结果。当我们像这样来组织我们的页面流的跳转逻辑时，问题不会变得复杂，而且这样也会相应地变得容易测试——毕竟他们只是请求的名称，action 的名称、返回结果的名称，仅仅只是字符串。这样做难度会有多少？

实际上不难，如果你将你的注意力集中在你系统的页面流上，那么你就可以将这些页面之间跳转的代码抽取出来成为一个专门操作导航数据的导航引擎，Struts 框架就是基于这一规则建立的，

我们将在 13.2 节中讨论在 Struts 框架中去确认页面流的正确性是多么的容易。在那一节中将展示一个简单的例子，例子是关于怎样去创建一个重构了的简单的网、将导航逻辑抽取到一个简单的类中，然后再去校验返回的结果数据。

◆ 诀窍

我们从最简单的测试开始：一个校验从一个页面跳转到另一个页面的能力的端到端测试。回到我们的咖啡店程序上来，首先我们检查从欢迎页（welcome）到目录页（catalog）是否顺利跳转。我们可以使用 HtmlUnit 这个工具来加载欢迎页（welcome），然后点击“查看目录”的按钮，确认返回的页面是否就是我们所要得到的目录页（catalog）。当然，在执行这个测试之前，我们必须先将应用程序部署到服务器上并且启动这个服务器。你可以在清单 13.1 中看到正在讨论的这个测试。

清单 13.1 **NavigationTest**, 一个样本页面流测试

```
package junit.cookbook.coffee.endtoend.test;

import java.net.URL;

import junit.framework.TestCase;

import com.gargoylesoftware.htmlunit.*;
import com.gargoylesoftware.htmlunit.html.*;

public class NavigationTest extends TestCase {

    private WebClient webClient;

    protected void setUp() throws Exception {

        webClient = new WebClient();

        webClient.setRedirectEnabled(true);

    }

    public void testNavigateToCatalog() throws Exception {
```



```
Page page =

    webClient.getPage(

        new URL("http://localhost:8080/coffeeShop/"));

assertTrue(

    "Welcome page not an HTML page",

    page instanceof HtmlPage);

HtmlPage welcomePage = (HtmlPage) page;

HtmlForm launchPointsForm =

    welcomePage.getFormByName("launchPoints");

HtmlInput htmlInput =

    launchPointsForm.getInputByName("browseCatalog");

assertTrue(

    "'browseCatalog' is not a submit button",

    htmlInput instanceof HtmlSubmitInput);

HtmlSubmitInput browseCatalogSubmit =

    (HtmlSubmitInput) htmlInput;

Page page2 = browseCatalogSubmit.click();
```

```

        assertTrue(
            "Catalog page not an HTML page",
            page2 instanceof HtmlPage);

        HtmlPage catalogPage = (HtmlPage) page2;

        assertEquals(
            "Coffee Shop - Catalog",
            catalogPage.getTitleText());
    }
}

```

在这个测试用例中有很多有意义的东西值得关注。

为了执行这个测试，我们首先要将它部署到服务器中并且将服务器启动。虽然这看起来是测试页面流的一个合理需求，但是有许多不相关的问题会导致无法执行这个测试：比如 EJB 部署的问题，servlet/URL 映射的问题等等。这并不是说这些问题就不重要而可以置之不理，而是说我们更愿意分开来测试他们。这个测试的目的仅仅是测试页面流。

在测试代码中会包含程序所部署到的服务器的名称（localhost）、监听端口（8080）的信息。但这些信息会随着应用程序的环境改变而改变，所以一个好的方法是，你应该将这些信息都抽取到一个外部的配置文件中。虽然这样会使测试的配置和正确执行变得稍稍复杂。

这个测试依赖于 web 页面本身的正确性。如果欢迎页或者目录页中有问题，即使问题不出现在导航上，这个测试也会执行失败。当然你会使用到第 12 章讲述的“测试 web 组件”的方式来独立测试那些页面，所以这里将不再考虑这些错误。这个测试的代码包含了 web 页面的结构信息——名称为“launchPoints”的表单和名称为“browseCatalog”的提交按钮。如果这些名称改变了，这个测试也会执行失败，这样这个测试变得很脆弱，页面中的按钮应该可以被替换成为超链接，获得正确的 URL，测试执行失败的原因只能出自于页面本身。这个测试是不允许那种情况发生的。

请不要错误地理解我们的意思：以上的内容并不是在指责 HtmlUnit，恰恰相反，HtmlUnit 是一个十分优秀的端到端的测试工具。由于 HtmlUnit 是着重分析 web 请求（全面的页面对象模型）的结果的，在对 web 程序做自动化端到端测试时它是一个理想的选择，因此 HtmlUnit 在这里并不是问题。问题是在这里使用它就好比我们要用锤子打苍蝇一样：用 HtmlUnit 来测试页面流。所以，我们应该写那些遵循于导航规则的测试。这才是我们要写的测试。

```

public void testNavigateToCatalog() {

    assertEquals(

```

```

        "Catalog Page",

        navigationEngine.getNextLocation(

            "Browse Catalog",

            "OK"));

    }

```

这个测试好比是：“如果我点击‘查看目录’的按钮，而且一切正常的情况下，将返回目录页面”。这个测试以一种抽象化的形式来表达地址（*locations*）、*actions* 以及返回结果：地址（*locations*）通常是一个页面，*action* 是一个提交表单或者点击超链接的操作，地址（*locations*）是机器的代码，而 *action* 和返回结果都是一种形式的转换。我们可以使用地址（*locations*）和 *actions* 这两个术语来对这个系统的导航建模。

地址（*locations*）就相当于网页的 URI 或者页面的模板。*action* 相当于提交表单或者点击连接所指向的 URI。这就意味着我们需要通过某种方式将传入的 URI 请求转换为 *locations* 和 *actions*，一旦我们给了每个 URI 一个 *location* 或者 *action* 的名称，我们就可以忽略哪一个 JSP 显示目录或者哪一个请求代表着“添加商品到购物车中”的具体内容。我们可以分开测试它们。

首先，清单 13.2 给我们展示了 Coffee Shop 控制器使用了一个独立的“请求-操作”映射。

清单 13.2 CoffeeShopController 使用操作映射

```

private void handleRequest(

    HttpServletRequest request,

    HttpServletResponse response)

    throws IOException, ServletException {

    String forwardUri = "index.html";

    String userName = "jbrains";

    try {

        String actionName =

actionMapper.getActionName(request);

```

```
log("Performing action: " + actionName);

if ("Browse Catalog".equals(actionName)) {

    CoffeeCatalog catalog = model.getCatalog();

    CatalogView view = new CatalogView(request);
    view.setCatalog(catalog);

    forwardUri = view.getUri();
}

else if ("Add to Shopcart".equals(actionName)) {

    AddToShopcartCommand command =
        makeAddToShopcartCommand(request);
    executeCommand(command);
}

else {

    log("I don't understand action " + actionName);
}

}

catch (Exception wrapped) {

    throw new ServletException(wrapped);
}
```

```

        request.getRequestDispatcher(forwardUri).forward(
            request,
            response);
    }

```

这里的 `actionMapper` 是一个 `HttpServletRequestToActionMapper` 的对象，我们在清单 13.3 中将其测试。

清单 13.3 MapRequestToActionTest

```

package junit.cookbook.coffee.web.test;

import java.util.*;
import java.util.regex.*;
import java.util.regex.Pattern;
import javax.servlet.RequestDispatcher;
import javax.servlet.http.*;
import junit.cookbook.coffee.HttpServletRequestToActionMapper;
import junit.framework.TestCase;
import org.apache.catalina.connector.HttpRequestBase;
import com.diasparsoftware.java.util.*;
import com.diasparsoftware.javax.servlet.http.*;

public class MapRequestToActionTest extends TestCase {

    private HttpServletRequestToActionMapper actionMapper;

    protected void setUp() throws Exception {

```

```

        actionMapper = new HttpServletRequestToActionMapper();
    }

    public void testBrowseCatalogAction() throws Exception {

        Map parameters =

            Collections.singletonMap(

                "browseCatalog",

                new String[] { "catalog" });

        doTestMapAction(

            "Browse Catalog",

            "/coffeeShop/coffee",

            parameters);
    }

    public void testAddToShopcart() throws Exception {

        HashMap parameters = new HashMap() {

            {

                put("addToShopcart-18", new String[] { "Buy!" });

                put("quantity-18", new String[] { "5" });

            }

        };
    }

```

```

doTestMapAction(

    "Add to Shopcart",

    "/coffeeShop/coffee",

    parameters);

}

private void doTestMapAction(

    String expectedActionName,

    String uri,

    Map parameters) {

    HttpServletRequest request =

        HttpUtil.makeRequestIgnoreSession(uri, parameters);

    assertEquals(

        expectedActionName,

        actionMapper.getActionName(request));

}
}

```

这些测试使用来自 Diasparsoft 的工具 HttpUtil 创建了一个假的 HttpServletRequest 对象。就像名称的含义一样，我们创建了一个请求而不关心会话（session）的跟踪，因为这个测试中我们不需要关心会话（session）的信息。测试中所涉及的变量越少越好。这里的“请求-操作”的映射是十分简单的：将请求转换为一个操作（action）的名称。这个测试用例是一个很容易转换成包含参数的测试用例（见解决方法 4.8 节）。

使用同一种方式，我们已经建立了一个从地址到 URI 的映射：将 location 转换为 URI 的名称。在加入这些机制之后，我们的 servlet 请求接收方法就成为清单 13.4 所示。

清单 13.4 CoffeeShopController 使用本地映射

```
private void handleRequest(  
    HttpServletRequest request,  
    HttpServletResponse response)  
    throws IOException, ServletException {  
  
    String userName = "jbrains";  
    String nextLocationName = "Welcome";  
  
    try {  
        String actionName = actionMapper.getActionName(request);  
        log("Performing action: " + actionName);  
  
        if (knownActions.contains(actionName) == false) {  
            log("I don't understand action " + actionName);  
        }  
        else {  
            String actionResult = "OK";  
            if ("Browse Catalog".equals(actionName)) {  
                CoffeeCatalog catalog = model.getCatalog();  
  
                CatalogView view = new CatalogView(request);  
                view.setCatalog(catalog);  
            }  
        }  
    }  
}
```



```

    }

    else if ("Add to Shopcart".equals(actionName)) {

        AddToShopcartCommand command =

            makeAddToShopcartCommand(request);

            executeCommand(command);

    }

    nextLocationName =

navigationEngine.getNextLocation(actionName, "OK");

    }

}

catch (Exception wrapped) {

    throw new ServletException(wrapped);

}

    String forwardUri = locationMapper.getUri(nextLocationName);

    request.getRequestDispatcher(forwardUri).forward(

        request,

        response);

}

```

请求处理的全部动作就是直接跳转。

1. 解析接收的请求来判断用户想要的操作。
2. 如果没有问题，执行这个操作。假设这个操作由于某种原因执行失败，就对这个错误设定一个简短的错误说明。比如，如果用户试图增加-1kg 的 Sumatra 到他的购物车中，就在返回结果中设定“错误的数量”。
3. 基于 action 的执行和其结果，向导航器询问下一个地址（location）。

4. 返回请求所应得的 URI。

这个设计使得在不实际运行 servlet 的情况下测试你程序中的其它导航规则成为可能——不管他们有多么复杂！当然，你至少要写一组测试用例来验证包含映射和导航的 servlet。使用 ServletUnit 或者 mock object，由你对这些技术的喜好决定。我们在 12 章中对两种技术都做了描述。

◆ 讨论

一旦你写了这些测试而且将这些导航规则抽取到一个简单的测试对象，你会发现这与 Struts 的 web 框架极为相似。在 Struts 中，你将导航规则定义在 struts-config.xml 中，而且导航引擎会操作那些数据，而不是将导航规则散布在整个应用程序中。Struts 框架不但容易理解和维护，而且也容易测试：你可以执行虚无的 action 来获得想要的 ActionForward 来检验期望的导航路径。更多内容将在 13.2 中介绍。

◆ 相关

- 4.8—创建数据驱动的 Test Suite
- 13.2—在 Struts 应用中测试导航规则

13.2 Test navigation rules in a Struts application

13.2 在 Struts 应用中测试导航规则

◆ 问题

你想要在你的 Struts 应用中测试页面间导航，并且最好不用启动 Struts。

◆ 背景

使用端到端测试导航开销很大。我们在解决方法 13.1 节中描述了这个问题。这里，我们感兴趣的是在 Struts 应用中测试导航规则。使用端到端测试对于 Struts 应用和其他类型的 web 应用，其开销是一样的。使用框架并不能使端到端测试变得简单。而 Struts 能做的，是提供一个不用求助于端到端测试的方法来测试导航规则，这个方法使得独立的导航测试变得非常简单。

◆ 诀窍

你可以使用的最直接的方法是使用 XMLUnit 校验 struts-config.xml 的内容。这里我们将展示一些测试样例，要想了解更多关于校验 XML 文档的内容，请参见第 9 章。现在，我们将使用 Struts 网站（<http://jakarta.apache.org/struts/index.html>）上发布的 struts-config.xml 作为被测文档。一些测试见清单 13.5。

清单 13.5 struts-config.xml 的 XMLUnit 测试

```
package junit.cookbook.coffee.web.test;

import java.io.*;

import junit.extensions.TestSetup;

import junit.framework.*;

import org.custommonkey.xmlunit.XMLUnit;

import org.w3c.dom.Document;

import org.xml.sax.InputSource;

public class StrutsNavigationTest extends StrutsConfigFixture {

    private static Document strutsConfigDocument;

    public static Test suite() {

        TestSetup setup =

            new TestSetup(new

TestSuite(StrutsNavigationTest.class)) {

        private String strutsConfigFilename =

            "test/data/sample-struts-config.xml";

        protected void setUp() throws Exception {

            XMLUnit.setIgnoreWhitespace(true);

            strutsConfigDocument =

                XMLUnit.buildTestDocument(

                    new InputSource(
```

```
new FileReader(  
new  
File(strutsConfigFilename))));  
}  
};
```

```
return setup;  
}
```

```
public void testLogonSubmitActionExists() throws Exception {  
    assertXPathExists(  
        getActionXPath("/LogonSubmit"),  
        strutsConfigDocument);  
}
```

```
public void testLogonSubmitActionSuccessMappingExists()  
    throws Exception {  
  
    assertXPathExists(  
        getActionForwardXPath("/LogonSubmit"),  
        strutsConfigDocument);  
}
```

```
public void testLogonSubmitActionSuccessMapsToWelcome()
```

```

throws Exception {

    assertXPathEvaluatesTo(

        "/Welcome.do",

        getActionForwardPathXPath("/LogonSubmit", "success"),

        strutsConfigDocument);

    }

}

```

Struts 的配置文档将导航规则同 locations 和 URI 间的映射结合在一起。当一个 action 跳转到另一个 action，它将利用导航规则；反之，当 actions 跳转到页面模板时（JSP 或 Velocity template）则使用 locations 和 URIs 间的映射规则。在这里，Struts 的配置文档既充当了导航引擎又充当了地址映射者（见解决方法 13.1 节）。你可以使用同样的方法测试地址映射和导航规则。

◆ 讨论

这个解决方法中有几件事要注意。首先，我们在服务器启动时 Struts 一次性载入配置文件（见解决方法 5.10 节，“为整个 Test Suite 建立设置实体”，更多地了解怎么使用 TestSetup）。其次，注意方法 `getActionXPath()`、`getActionForwardXPath()` 和 `getActionForwardPathXPath()`。这些方法将“action”和“action forward”概念转化成了 struts-config.xml 中相应的 XPath locations。你不仅不需要记住 actions 和 action forwards 在 XPath 下多变的表达方式，同时你也避免了当 Struts 配置文件的 DTD 特征改变时，复制那些表达。我们提取了一个 fixture 类 `StrutsConfigFixture`，并将那些方法放入一个新的 fixture 类以供重复使用。清单 13.6 展示了这些方法。

清单 13.6 struts-config.xml 测试的一个样本 fixture

```

package junit.cookbook.coffee.web.test;

import org.custommonkey.xmlunit.XMLTestCase;

public abstract class StrutsConfigFixture extends XMLTestCase {

    protected String getActionForwardPathXPath(

```

```

        String action,

        String forward) {

    return getActionXpath(action)

        + "/forward[@name='" + forward + "']/@path";

    }

    protected String getActionXpath(String path) {

        return "/struts-config/action-mappings/action[@path='"

            + path + "']";

    }

    protected String getActionForwardXpath(String action) {

        return getActionXpath(action) + "/forward";

    }

}

```

注意这种扩展的形式。这是用 XPath 校验 XML 文档的一种很好的方式，因为当一个基于 XPath 判定失败后，很难找出失败的原因。或许你在你的 XML 文档的三层之外的一个地方打错了一个元素的名称，也或许你在输入某个属性值时忘记了加上“@”标志。通过编写许多小的递增的测试，当某个测试失败时就能更轻易地定位出问题的所在。如一个 action mapping，考虑下面三个测试：

1. 这个 action 配置好了没？
2. 这个 action 包含 forwards 么？
3. action 里的 forwards 都正确么？

编写三个测试，而不是仅仅编写第三个测试，这才使得定位成为可能；例如，如果第二和第三个测试都失败了，则问题就出在 Struts 配置文件上——此 action 没有 forwards。

◆ 相关

- 3.4—抽取一个测试模块
- 5.10—为整个 Test Suite 建立设置实体
- 13.1—测试页面流

13.3 Test your site for broken links

13.3 测试你的网站以寻找失效的链接

◆ 问题

你想要校验你网站的所有链接都指向某个地方。

◆ 背景

如果一个 web 应用的终端用户点击某个链接，可这个链接却没有指向任何地方，那么他将很可能被激怒而离开你的网站。你需要不惜任何代价避免让终端用户看到“404 File Not Found”。幸运的是，使用 JUnit 测试你的整个站点非常简单。

◆ 诀窍

这里, HtmlUnit 可以来解救你：一种相当简单的递归算法使这个测试的编写变得令人惊奇地容易。算法的关键部分是：

1. 调用 WebClient.getPage()检索页面。
2. 如果此页是 HtmlPage，得到所有<a>标记，并跟踪每一个链接。
3. 如果你链到了一个域外的页面，则检查终止。
4. 当跟踪某个链接时出错，则定位链接为失效的链接。

这是一种递归算法；但是，当我们设法执行测试时，我们会更注意一些具体的细节，这些细节你是需要知道的。

Jakarta Commons HttpClient 并不处理 mailto 链接，因而此类链接并不能被校验。也许你能做的最多的也就是检验他们的 e-mail 地址是否合法。我们推荐你手工检验它们。

HtmlUnit 1.2.3 有一个缺点，即它不能正确处理本页中的链接()。我们已经将这个问题提交给了 Mike Bowler，很有可能当你读到这个句子时，它已经被很好地解决了。如果还没解决，咨询一下他吧。

许多链接会链回到已被测试过的页面。为了避免无限递归，跟踪每一个已经被检查过的 URL，如果某些 URLs 再次出现，则跳过它们。

有时候失败发生了，但它并非真正的失败——测试失败了，你检查链接，但它并不是一个失效的链接。这有可能是因为网络自身的原因：有时候有那么几秒 URL 不可获得。除此，我们就不知道为什么会发生。你需要多运行几次，观察失败的形式。因为如果你每个星期运行一次，这些失败的假像是不会不断出现的。

同样，请认识到，我们并不是在对提交的表单进行校验，校验它一般是比较复杂的。请参见 12 章中讨论的如何一个个单独地校验 web 表单。

让我们一起来看看清单 13.7 中的代码。简单地将 domainName 改变成你想要开始的 URL。执行这个测试时，我们不推荐你使用 yahoo.com 作为你的起始 URL——这样会花费你很多时间。

清单 13.7 LinksTest

```
package junit.cookbook.applications.test;

import java.io.IOException;

import java.net.URL;

import java.util.*;

import junit.framework.TestCase;

import com.gargoylesoftware.HtmlUnit.*;
import com.gargoylesoftware.HtmlUnit.html.*;

public class LinksTest extends TestCase {

    private WebClient client;
```



```
private List urlsChecked;

private Map failedLinks;

private String domainName;

protected void setUp() throws Exception {

    client = new WebClient();

    client.setJavaScriptEnabled(false);

    client.setRedirectEnabled(true);

    urlsChecked = new ArrayList();

    failedLinks = new HashMap();

}

public void testFindABrokenLink() throws Exception {

    domainName = "yahoo.com";

    URL root = new URL("http://www." + domainName + "/");

    Page rootPage = client.getPage(root);

    checkAllLinksOnPage(rootPage);

    assertTrue(

        "Failed links (from => to): " + failedLinks.toString(),

        failedLinks.isEmpty());

}
```

```
private void checkAllLinksOnPage(Page page) throws IOException {  
    if (!(page instanceof HtmlPage))  
        return;  
  
    URL currentUrl = page.getWebResponse().getUrl();  
    String currentUrlAsString = currentUrl.toExternalForm();  
  
    if (urlsChecked.contains(currentUrlAsString)) {  
        return;  
    }  
  
    if (currentUrlAsString.indexOf(domainName) < 0) {  
        return;  
    }  
  
    urlsChecked.add(currentUrlAsString);  
    System.out.println("Checking URL: " + currentUrlAsString);  
  
    HtmlPage rootHtmlPage = (HtmlPage) page;  
    List anchors = rootHtmlPage.getAnchors();  
    for (Iterator i = anchors.iterator(); i.hasNext();) {  
        HtmlAnchor each = (HtmlAnchor) i.next();  
    }  
}
```

```

String hrefAttribute = each.getHrefAttribute();

boolean isMailtoLink = hrefAttribute.startsWith("mailto:");

boolean isHypertextLink = hrefAttribute.trim().length() > 0;

if (!isMailtoLink && isHypertextLink) {

    try {

        Page nextPage = each.click();

        checkAllLinksOnPage(nextPage);

    }

    catch (Exception e) {

        failedLinks.put(currentUrlAsString, each);

    }

}

}

}

```

◆ 讨论

关于这个测试的一些警告：

- 由于它要在一个实时的网络中检查 URLs，所以它执行起来很慢。
- 如果你的某个失效链接指向的是一个不存在的域或者这个域的 服务器已经完全停止了，这个测试可能报告一个错误前就已经中止网络连接了，这会使测试变得更慢。虽然我们这个版本的测试只校验了指定域的页面，但是这个限制是可以更改的。如果这个限制没有了，那么上面说的就会成为一个问题。

□ 整个的作为一个大的测试在执行，而非一个用于校验每个 URL 的小测试。如果我们不直接执行 Test，我们将无法解决这个问题；然而，当测试执行时在内存中创建一个 TestSuite 是令我们头疼的，因而我们不会选择这样做。

但仍然，我们认为对于一般的使用这是一个好的起点。

◆ 相关

□ 12—测试 web 组件

13.4 Test web resource security

13.4 测试 web 资源的安全性

◆ 问题

你想要校验你已经正确地保护了你的 web 资源。

◆ 背景

校验安全性的典型方式是使用端到端测试：部署应用程序，作为不同的用户注册，并且校验你会在期望的时刻收到“Authorization Failure” or “Forbidden”。这个解决方法中我们将讨论怎么样自动运行这类的测试，不过请记住这些测试可能违反我们不测试平台的原则。如果你正在使用 J2EE 公布的安全特性——如果你没有，我们将感到很惊讶——你就可以在不涉及容器的条件下测试你的整个安全设置。

◆ 诀窍

首先，让我们看看怎样从外部测试安全，我们使用端到端测试。HtmlUnit 对确定请求中的校验权限提供支持，因此我们可以假设已经有一个用户登录了。你还可以使用 HtmlUnit 测试登录过程本身。回到我们的 Coffee Shop 应用，设想一个针对产品价格变化这样简单的日常事务的管理接口。显然，我们需要一些安全机制来保护这些特征。特别地，我们想确定要是一个用户想在没有登录的条件下访问网页，应用程序将迫使我们明确他们的身份。清单 13.8 中的测试校验了那种环境。我们的应用中，所有的管理页面都放在了 URI 为 admin 的路径下面。

清单 13.8 管理页面资源的授权规则测试

```
package junit.cookbook.coffee.endtoend.test;
```

```
import java.net.URL;
```

```

import junit.framework.TestCase;

import com.gargoylesoftware.HtmlUnit.*;

import com.gargoylesoftware.HtmlUnit.html.HtmlPage;


public class AdminWelcomePageTest extends TestCase {

    private static final int AUTHORIZATION_FAILED = 401;


    private WebClient webClient;


    protected void setUp() throws Exception {

        webClient = new WebClient();

    }


    public void testWithoutCredentials() throws Exception {

        try {

            Page page =

                webClient.getPage(

new
URL("http://localhost:8080/coffeeShop/admin/"));


            fail("Got through without a challenge?!");

        }

        catch (FailingHttpStatusCodeException expected) {

            assertEquals(

```

```

        AUTHORIZATION_FAILED,

        expected.getStatusCode());

    }

}

}

```

这里我们使用 HtmlUnit 设法检索页面, 预计将返回一个代码 401: “Authorization Failed”为了做这个测试工作, 我们需要在我们应用的 web 部署描述文件中配置安全性。清单 13.9 展示了相关部分。

清单 13.9 A web deployment descriptor that passes the tests in AdminWelcomePageTest

```

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3/

    /EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app id="WebApp">

    <!-- Most of the file omitted for brevity -->

    <display-name>CoffeeShopWeb</display-name>

    <security-constraint>

        <web-resource-collection>

            <web-resource-
name>CatalogAdministration</web-resource-name>

            <url-pattern>/admin/*</url-pattern>

        </web-resource-collection>

        <auth-constraint>

            <role-name>administrator</role-name>

```

```

        </auth-constraint>

    </security-constraint>

    <login-config>

        <auth-method>BASIC</auth-method>

    </login-config>

</web-app>

```

既然我们知道这些页面要求请求者为登录用户，因而我们需要限制那些扮演管理角色的用户的访问权限。我们的下一个测试使用 HtmlUnit 的 CredentialProvider API 去冒充已经有一个特定的用户登录了。类 SimpleCredentialProvider 允许你确定用户名和密码来模拟所有来自同样 WebClient 对象的请求。假设 admin 是某个管理者的用户名，他拥有对在线商店管理部分的访问权限。这个测试里，我们需要校验 admin 可以登录，并且能够看到欢迎页面。

```

public void testAdminLoggedIn() throws Exception {

    webClient.setCredentialProvider(

        new SimpleCredentialProvider("admin", "adm1n"));

    Page page =

        webClient.getPage(

            new URL("http://localhost:8080/coffeeShop/admin/"));

    assertEquals(

        HttpServletResponse.SC_OK,

        page.getWebResponse().getStatusCode());

    assertTrue(page instanceof HtmlPage);
}

```

```

        HtmlPage welcomePage = (HtmlPage) page;

        assertEquals(

            "Coffee Shop - Administration",

            welcomePage.getTitleText());

    }

```

在这个测试中你会看到两个不同点：首先，我们调用了 `WebClient.setCredential- Provider()` 来模拟用户 admin 使用密码 adm1n 登录。从这点来看，所有来自 WebClient 对象的请求都可以通过权限校验直到你更改 `CredentialProvider`，这些行为你都可以通过那个 API 猜测得到。另一个值得注意的不同点是，我们校验它，返回一个代码 200，意思是“OK”。我们需要它，因为我们对 WebClient 进行了微小而有用的改动。在 `setUp()` 方法中，我们用粗体字让此行显现出来。

```

protected void setUp() throws Exception {

    WebClient = new WebClient();

    webClient.setThrowExceptionOnFailingStatusCode(false);

}

```

这一行配置了当 WebClient 从服务器中收到一个失败的状态代码（一个位于 200—299 范围外的代码）时，它将如何反应。默认状态下，WebClient 会抛出一个异常以解释我们的第一个测试：因为 401 是期望出现的表示失败的代码，所以这个测试期望它的 WebClient 抛出一个 `FailingHttpStatusCodeException` 异常。我们可以通过禁止以上行为来避免 catch 所有的这些异常，但相应地，我们需要检查每个请求的状态代码——包括那些表示成功的代码。对只期望服务器返回失败状态代码的测试进行简化，花销并不大。

下面这个例子将尝试着以购买者的身份登录应用的管理部分。我们期望服务器返回“forbidden”

```

public void testShopperLoggedIn() throws Exception {

    WebClient.setCredentialProvider(

        new SimpleCredentialProvider("shopper", "sh0pper"));

    Page page =

        WebClient.getPage(

            new URL("http://localhost:8080/coffeeShop/admin/"));

```



```

    assertEquals(

        HttpServletResponse.SC_FORBIDDEN,

        page.getWebResponse().getStatusCode());

}

```

测试的最初版本，我们使用硬编码的状态代码，但后面 `HttpServletResponse` 会为它们定义常量，我们用常量取代了硬编码的状态代码。改变后的测试变得更容易理解。你能够看见怎样模拟没有登录，拥有访问权限的情况下登录和没有访问权限的情况下登录。这些是根据测试的复杂程度要求，你需要编写的基本代码，注意力集中在认证和给 web 资源授权上。

◆ 讨论

请认识到，像我们描述的那样从外部测试应用资源的安全性有一个缺点。如果你校验容器的所有用户角色，在所有资源上执行你的安全策略，这将存在一个非常大的危险——复制你测试中的整个访问控制清单。至少从某种程度上说，它触犯了 *declarative security*。因此，你应该去测试你定义的安全信息是否正确，而不是去验证你的应用服务器是否正确的理解了他们。测试后者，需要从本质上引用应用服务器的 *assembly descriptor-processing* 算法。如果你正在创建一个应用服务器，那么以上的努力是合适的；然而，如果你只是在创建一个企业级的应用，让它在应用服务器上运行，那么以上就扯远了。不要测试平台。相反的，使用我们在第 9 章中描述的技术，“测试与 XML”校验你的每个部署描述符的内容。

测试你的服务器端的安全设置时，你需要的具体技术依赖于应用服务器：有些服务器开放度不够，不能使服务器的配置数据对外部组件有效。如对于 JBoss，XMLUnit 和 Plain Old JUnit 的组合可以帮助你校验你在 `jboss-web.xml`、`login-config.xml`、`users.properties` 和 `roles.properties`（在基于 `properties` 文档的用户注册的情况下）中的设置；对于其他应用服务器，检查你的本地文档。

◆ 相关

- 9—测试与 XML

13.5 Test EJB resource security

13.5 测试 EJB 资源的安全性

◆ 问题

你想要测试你的 EJB 安全设置。

◆ 背景

手工测试 EJB 安全性总的来说比手工测试 web 资源的安全性更惹人讨厌（见解决方法 13.4）。典型的解决方法是制作一个 web 前台，使之能够直接调用 EJBs，或者你可以据此目标使用一个支持测试客户端应用程序的开发环境。这些测试客户端本质上是交互的 Java 解释程序，你通过点击超链接来执行测试代码；同时，它对快速校验你的 EJBs 没有完全失效很有用，然而它们并不是一个好的测试工具的替代品（至少我们这样认为）。你需要的是一个编写包含安全因素的测试，这个测试能够通过安全认证、查询和使用 EJB。这些测试可以很容易地自动生成。这个解决方法将描述如何实现这一点。

◆ 诀窍

我们推荐从 Cactus 开始，它可以帮助你冒充已经有一个特定的用户登录到了每个测试上。它采用了一个与 HtmlUnit 类似的机制（见解决方法 13.4）。总体策略是为 Cactus 提供合适的验证信息来假冒一个用户已经登录，然后试着调用 EJB 中的 protected 方法。请记住因为我们正在使用 Cactus，所以这些测试是在容器中执行的。在这个解决方法的讨论部分我们将回到这点上。清单 13.10 展示了第一个测试：如果没有任何人登录，那么调用一个 EJB 方法应当失败。我们正在讨论的 EJB 是一个 session bean，它可以为我们的 Coffee Shop 应用的管理部分提供业务逻辑。此处的 EJB 执行价格的调整。

清单 13.10 PricingOperationsSecurityTest，一个 server-side Cactus 测试

```
package junit.cookbook.coffee.model.ejb.test;
```

```
import java.rmi.ServerException;
```

```
import javax.ejb.EJBException;
```

```
import javax.naming.*;
```

```
import javax.rmi.PortableRemoteObject;
```

```
import junit.cookbook.coffee.model.ejb.*;
```

```
import org.apache.cactus.*;
```

```
public class PricingOperationsSecurityTest extends ServletTestCase {
```

```
    public void testNoCredentials() throws Exception {
```

```

Context context = new InitialContext();

Object object = context.lookup("ejb/PricingOperations");

PricingOperationsHome home =

    (PricingOperationsHome) PortableRemoteObject.narrow(

        object,

        PricingOperationsHome.class);

try {

    PricingOperations pricingOperations = home.create();

    fail("No credentials and you got through?!");

}

catch (ServerException expected) {

    Throwable serverExceptionCause =
expected.getCause();

    assertTrue(

        "This caused the ServerException: "

            + serverExceptionCause,

        serverExceptionCause instanceof

EJBException);

    EJBException ejbException =

        (EJBException) serverExceptionCause;

    Exception ejbExceptionCause =

        ejbException.getCausedByException();

```

```

        assertTrue(
            "This caused the EJBException: " +
            ejbExceptionCause,
            ejbExceptionCause instanceof
            SecurityException);
    }
}

```

用宋体表示的所有代码验证了服务器端的异常，实际上是安全性异常，而并不是偶然的错误。在你的应用的不同层次中，在你将各种 exception 向上传递时，使用上层 exception 来包装下层的 exception，这是 java 中的一个惯例。当我们添加一个或两个测试时，我们会重构上面的代码使之形成一个独立的方法。否则，这些代码使用的是我们在解决方法 2.8 中描述的技术，“测试是否抛出正确的异常”。我们的下一测试验证了一个拥有权限的用户能够创建一个 PricingOperations bean。

你需要能够根据 Cactus 网站的一些简要的介绍在 request 中提供认证信息。当我们扩展了我们的 web 部署描述符后，增加下面的测试，它验证了一个管理者可以调用 home 接口或者直接调用 bean 本身。

```

public void beginAdministrator(WebRequest request) {

    request.setRedirectorName("ServletRedirectorSecure" );

    request.setAuthentication(

        new BasicAuthentication("admin", "adm1n"));

}

public void testAdministrator() throws Exception {

    PricingOperations pricingOperations = home.create();

    pricingOperations.setPrice("762", Money.dollars(12, 50));

}

```

Cactus 站点引导我们使用一个“begin”方法来为这个测试设置认证信息。我们的测试就可以简单地创建一个 PricingOperations bean 并采用一个方法调用它。只要这个测试不抛出任何安全性相关的异常，它就过关了——这就是为什么这中间没有断言。如果这个测试确实抛出了一个异常，我们需要加入逻辑来判断它抛出的是什么异常，使得这个测试只会在它抛出一个安全相关的异常时失败。我们相信额外的付出将不会收到相应的回报，所以就让这个测试这样吧。我们已经把 home 移动到测试的 fixture 里了，并在 setUp() 中对其进行了初始化，如下所示：

```
public class PricingOperationsSecurityTest extends ServletTestCase {

    private PricingOperationsHome home;

    protected void setUp() throws Exception {

        Context context = new InitialContext();

        Object object = context.lookup("ejb/PricingOperations");

        home =

            (PricingOperationsHome) PortableRemoteObject.narrow(

                object,

                PricingOperationsHome.class);

    }

    // Tests omitted

}
```

最后一个测试，我们将试着以一个购买者的身份登录并且调用 PricingOperations EJB，它应该出现“user not authorized.”的失败信息。当我们编写这个测试时，我们意识到我们希望 Cactus 测试转向器能为任何通过认证的用户工作，因为 EJB 层将要执行更严格的安全检查。当试图指出怎么去做失败后，我们创建了一个任何用户都没有扮演的角色，将它命名为 tests。我们用一个安全性限制配置我们的测试 web 应用（包含 Cactus 测试的 web 应用），但授权所有用户使用 web 应用（不禁止任何使用者）。这迫使想要通过测试用户授权认证的用户调用 EJBs。这样就使得要去测试用户调用 EJB 的权限的时候，必须确信用户已经被认证，但任何用户都可以执行这些测试。我们认为基于安全性的测试，这样设计是很好的。

我们对这个“没有安全证书”的测试进行重构，将试图调用 PricingOperationsHome.create() 的代码提取出来，展开期望的异常，并且验证它是一个 SecurityException。我们甚至添加一些代码来检查来

自 `SecurityException` 的消息——现在我们需要区分“没有安全认证”和“没有授权”。清单 13.11 展示了此方法。

清单 13.11 校验 **SecurityException** 信息

```
private void doTestExpectingSecurityException(
    String testFailureMessage,
    String expectedSecurityExceptionMessageContains)
    throws Exception {

    try {
        PricingOperations pricingOperations = home.create();
        fail(testFailureMessage);
    }

    catch (ServerException expected) {
        Throwable serverExceptionCause = expected.getCause();

        assertTrue(
            "This caused the ServerException: "
                + serverExceptionCause,
            serverExceptionCause instanceof EJBException);

        EJBException ejbException =
            (EJBException) serverExceptionCause;

        Exception ejbExceptionCause =
            ejbException.getCausedByException();
    }
}
```

```

        assertTrue(
            "This caused the EJBException: " +
            ejbExceptionCause,
            ejbExceptionCause instanceof SecurityException);

        SecurityException securityException =
            (SecurityException) ejbExceptionCause;

        String securityExceptionMessage =
            securityException.getMessage();

        assertTrue(
            securityExceptionMessage,
            securityExceptionMessage.matches(
                ".*"
                +
            expectedSecurityExceptionMessageContains
                + ".*"));
    }
}

```

我们用粗体突出了检查 SecurityException 消息的额外代码。根据以上的改变，“没有安全认证”的测试代码现在变成：

```

public void testNoCredentials() throws Exception {
    doTestExpectingSecurityException(
        "No credentials and you got through?!",

```

```

        "principal=null");
    }

    然而试图作为购买者调用 EJB 的新测试将变成：

    public void beginShopper(WebRequest request) {

        request.setRedirectorName("ServletRedirectorSecure");

        request.setAuthentication(

            new BasicAuthentication("shopper", "sh0pper"));

    }

```

```

    public void testShopper() throws Exception {

        doTestExpectingSecurityException(

            "Only administrators should be allowed to do this!",

            "Insufficient method permissions");

    }

```

对于“administrator”测试，首先我们执行了 Cactus 的 beginShopper()来扮演一个购买者，然后我们执行了这个测试本身，在 SecurityException 消息中期望看见“Insufficient method permissions”。这个消息文本只对 JBoss 3.2.2 有效，如果我们编写更多的这样的测试，我们就应当将文本提取出来，编写一个属性文件或者至少创建一个符号常量，以避免大量地复制。假设在 JBoss 4.0 中，那些消息文本改变了，我们不想对分散在我们测试中的 25 条字符串都进行改变。

这里我们有三种典型的安全性测试的样例：没有用户登录，一个授权用户登录，一个非授权用户登录。你可以将它们作为模板来编写你自己的测试。

◆ 讨论

Cactus 测试是很复杂的，但只有在运行环境复杂的情况下才复杂。当你试着正确地按照指示去做时，编写测试自身涉及一个轻微弯曲的学习曲线——我们认为，这个曲线并不陡峭——和一两个尝试性的实验。好消息是，一旦你启动了，剩下的烦恼只有测试的执行速度让人不堪忍受。很显然这不是 Cactus 的问题，而是所有的容器中测试的策略所固有的弊病。唯一的缺点是像这样去写一个详尽的安全性测试 suite——尝试所有角色的排列组合等等——会导致重复代码过多以至出问题。

如果你不断地重构，最终你可以制作一个引擎，这个引擎可以从一个描述安全角色的文本创建测试。你可以想像用一个 XML 文档来描述哪个角色授权去执行哪些 actions，它就可以用作创建测试。这个听起来熟悉么？我们就是把它作为 J2EE 公布的安全特性描述符！要使你的测试正确等价于使你的 test-generating XML 文档正确。要不是这样，我们推荐只校验部署符自身。使用 XMLUnit 来校验你已经正确地进行了安全设置，然后使用无论哪种策略，只要是你需要的，来校验你应用服务器特定的配置文件。这些测试提供了一个警告系统，无论何时，有人要改变安全性设置：测试就会失败，然后将由开发小组来确定改变是否正确。这将可以避免：由于某些人在 2 a.m.改变文件而引起偶然安全漏洞并且又忘记警告别人。

◆ 相关

- 9—测试与 XML
- 13.4—测试 web 资源的安全性
- Cactus Security HOWTO (http://jakarta.apache.org/cactus/writing/howto_security.html)

13.6 Test container-managed transactions

13.6 测试容器管理的事务处理

◆ 问题

你想要校验你的容器管理的事务处理。

◆ 背景

基于事务处理的编程可能会很复杂，这是形成 J2EE 容器管理事务处理特征的原因之一。此特征允许你不必对所有讨厌的细节进行编码来指定事务处理。容器管理事务处理的简易性常常使程序员觉得它太容易了甚至他们有时会担心当负载过重时，应用将不会像期望的那样运作。因为他们知道基于事务处理的编程很复杂，因此他们觉得他们应当做更多事情来为应用添加事务处理行为。测试容器管理事务处理根本不复杂，只要你信任容器。这个解决方法将帮助你集中注意力在测试你的工作上，而不是容器的工作。

◆ 诀窍

你可以采取的最直接的方式是在你的部署描述符中校验容器管理事务处理的属性设置。当你做这件事的时候，你并不是在测试你的事务处理设置对你的应用是否可行，而是测试事务处理设置是否符合你的观点，你认为它们应当是怎样的。这样的测试主要防止某些人无意识下（或因为粗心）改变了设置。这样的测试会在任何改变发生时向你发出警告，由此你可以立即判断此改变是否正确。它

仅仅避免了令人讨厌的对测试或产品问题的调试任务，并且回溯地追踪它，看它在哪个事务处理隔离水平下不可能工作。可以通过校验你的部署描述符避免这个问题。你可以使用第 9 章的技术来处理。

你也许会开发额外的 Deployment Tests 来强化你自己的部署规则。例如，你也许会决定所有的实体 bean 组件方法（位于远处和本地的接口）必须随要求的事务处理属性一起部署。

这种情况下，你当然应当编写一个测试来校验那些设置，比如清单 13.12 中的测试校验了任何涉及到某个特定 EJB 的组件方法的容器事务处理属性。请注意它有多复杂。

清单 13.12 EntityBeanTransactionAttributeTest

```
package junit.cookbook.coffee.model.ejb.test;

import java.io.*;

import org.apache.xpath.XPathAPI;
import org.custommonkey.xmlunit.*;
import org.w3c.dom.*;
import org.xml.sax.InputSource;

public class EntityBeanTransactionAttributeTest extends XMLTestCase {

    protected void setUp() throws Exception {

        XMLUnit.setIgnoreWhitespace(true);

    }

    public void testOrder() throws Exception {

        doTestTransactionAttribute(

            "../CoffeeShopLegacyEJB/META-INF/ejb-jar.xml",

            "Order");

    }

}
```

```
}
```

```
private void doTestTransactionAttribute(
    String ejbDeploymentDescriptorFilename,
    String ejbName)
    throws Exception {

    Document ejbDeploymentDescriptor =
        XMLUnit.buildTestDocument(
            new InputSource(
                new FileInputStream(
                    new
File(ejbDeploymentDescriptorFilename))));

    String transactionAttributeXpath =
        "/ejb-jar/assembly-descriptor/container-transaction"
        + "[method/ejb-name='" + ejbName + "' and "
        + "(method/method-intf='Remote' or "
        + "method/method-intf='Local'))]"
        + "/trans-attribute";

    NodeList transactionAttributeNodes =
        XPathAPI.selectNodeList(
            ejbDeploymentDescriptor,
```

```
transactionAttributeXpath);
```

```
assertTrue(
```

```
    "No transaction attribute setting for " + ejbName + " EJB",
```

```
    transactionAttributeNodes.getLength() > 0);
```

```
for (int i = 0;
```

```
    i < transactionAttributeNodes.getLength();
```

```
    i++) {
```

```
    Node each = transactionAttributeNodes.item(i);
```

```
    Text text = (Text) each.getFirstChild();
```

```
    Node assemblyDescriptorNode =
```

```
        each.getParentNode().getParentNode();
```

```
    assertEquals(
```

```
        "Transaction attribute incorrect at "
```

```
        + assemblyDescriptorNode.toString(),
```

```
        "Required",
```

```
        text.getData());
```

```
    }
```

```
}
```

}

对于这个测试，我们已经重构出了一个普遍适用的版本，因而你可以将其转变成为一个 Parameterized Test Case（见解决方法 4.8 节，“创建数据驱动的 Test suite”），在你系统的所有实体 bean 中循环。根据描述容器管理的事务处理所固有的弹性，我们需要一个特别长且费解的 XPath 表达式来匹配它们：任一包含了我们要查询的 ejb-name 的 container-transaction 节点，Remote 或者 Local 的 method-inf。就算经验丰富的 XPath 用户也会花上一段时间来确定这个测试究竟在做什么，因而命名就显得非常重要了。像我们先前所说的，你可以编写这样的测试，但编写工作很复杂。相反，我们建议你用一个能够更加轻易地强化部署规则的过程来创建你的 EJB 部署描述符。如果你使用 XDoclet（<http://xdoclet.sourceforge.net/>）来创建你的 EJB 部署描述符，那么你可以使用 XJavaDoc——一个标准的 JavaDoc 引擎——来分析你的实体 bean 的源文件并对每个实体 bean 的 @ejb.transaction 类型的属性值做断言。我们建议你尝试几种不同的方法，看哪种方法最适合你。

◆ 讨论

在这个解决方法中我们清楚要做什么：设置在你的部署描述符中的事务处理属性是符合你的要求的，你认为它应当是什么样的。剩下的问题是：你怎么测试那样的事务处理属性设置能够得到想要的结果？典型的，一个人选择事务处理属性和事务隔离水平是基于对系统和现实试验的推测。我们将不会探讨怎样辨别哪些事务处理设置是合适的，因为这不是本书的目的。现实试验中，有许多商业的——也许还有非商业的——工具我们可以用来模拟大量负载下的 web 应用。JUnit 中有表现测试包，如 JUnitPerf（www.clarkware.com/software/JUnitPerf.html）；及 JUnitPerf 和 HtmlUnit 的组合可以用来帮助你执行 web 应用的压力测试。无论什么样的工具都无法替代将你的应用程序部署到产品级的服务器上，然后模拟 1000 个用户去访问它，并检测错误日志。正如 Ron Jeffries 说的，“思索和试验——哪一个更能给出正确的答案？”

◆ 相关

- 4.8—创建数据驱动的 Test suite
- 9—测试与 XML
- JUnitPerf（www.clarkware.com/software/JUnitPerf.html）
- XDoclet（<http://xdoclet.sourceforge.net/>）