软件开发方法学精选系列

PEARSON

优质代码

软件测试的原则、实践与模式

Quality Code

Software Testing Principles, Practices, and Patterns

[美] Stephen Vance 著

伍斌 译

人民邮电出版社
POSTS & TELECOM PRESS

目　录

软件开发方法学精选系列

Quality Code:Software Testing Principles,Practices,and Patterns

优质代码　软件测试的原则、实践与模式

［美］Stephen Vance　著

伍斌　译

人民邮电出版社

北京

图书在版编目（**CIP**）数据

优质代码：软件测试的原则、实践与模式/（美）万斯(Vance,S.)著；伍斌译.--北京：人民邮电出版社，2015.1

（软件开发方法学精选系列）

书名原文：Quality code:software testing principles,practices,and patterns

ISBN 978-7-115-37558-2

Ⅰ.①优… Ⅱ.①万…②伍… Ⅲ.①软件—测试 Ⅳ.①TP311.5

中国版本图书馆CIP数据核字（2014）第273844号

内容提要

本书讲述如何对所有的软件进行轻松的例行测试，书中为读者提供一些工具——一些实现模式，这些工具几乎可以测试任何代码。本书分为三个部分：第一部分讨论了测试的一些原则和实践，包括首次优质、代码意图、测试攻略和测试与设计之间的关系等；第二部分讨论了有关测试实践方面的一些模式，包括测试构造器和getter/setter、处理宁符串、封装与覆写、调整代码可见性、测试单例模式、验证错误条件，以及利用各种接缝和测试多线程等；第二部分展示了两个实例的编程过程，其中一个是用测试驱动开发方法编写新的Java应用程序WebRetriever，另一个是为一个未写测试的JavaScript开源项目iQuery Timepicker Addon添加测试代码。

本书适合对测试驱动开发有初步了解或实践并想提升测试代码编写技能的程序员和自动化测试工程师阅读，也适合想通过本书在GitHub上的微量提交的代码来学习用测试驱动开发方法编写Java新项目和用测试来驯服JavaScript遗留代码的详细过程的任何读者阅读。

◆著 ［美］Stephen Vance

译 伍斌

责任编辑 杨海玲

责仟印制　张佳莹　彭志环
◆人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京铭成印刷有限公司印刷
◆开本：720×960　1/16
印张：13.5
字数：241千字　　2015年1月第1版
印数：1-4000册　　2015年1月北京第1次印刷
著作权合同登记号　图字：01-2014-1729号
定价：49.00元
读者服务热线：**(010)81055410**　印装质量热线：**(010)81055316**
反盗版热线：**(010)81055315**

版权声明

译者序

2014年仲夏前的一个下午，当我正在为自己的处女作《驯服烂代码》的收官忙得焦头烂额时，接到了策划编辑杨海玲老师发出的翻译本书的邀请。由于之前翻译过海玲老师策划的《测试驱动数据库开发》，了解译书的辛苦意味着什么。看了本书的页数，我就知道，要翻译它需要我在写书的同时，额外再花3个月每天用五六个小时来翻译。我想时间真的不够用，要是这书不合我的口味，就找个理由谢绝吧。于是就在亚马逊网站上找到了本书的目录和书评。

本书在亚马逊网站的评价是4颗星，一个不错的分数。再看给最高星级的评价说本书中的例子很清晰。嗯，我喜欢讲述清晰的作品。

再看看目录吧。这一看不要紧，本来是要找不翻译本书的理由，结果反倒找到了下面这些对我胃口的关键词：软件匠艺、首次优质、代码意图、单例模式测试、错误条件验证、接缝的使用、多线程测试等。这些关键词成了我要译本书的理由。

回想我自己写的书，在讨论测试技术实现的广度和深度方面与本书相比真是望尘莫及。比如前面提到的单例模式测试、错误条件验证和多线程测试，都是我基本从未考虑过的。由此可见，本书作者在测试驱动开发领域的实践方面功力深厚。

我和本书作者都认同编程是门手艺。而手艺人所最看重的，不是最后精彩绝伦的成果，而是成就这个结果的精雕细琢的过程。因为“授人以鱼，不如授之以渔”，所以我在写自己的《驯服烂代码》时，就刻意通过详述若干个带有 GitHub 微量代码提交的实例的编写和思考过程，

来在编程操练中悟道。可喜的是，本书作者也看到了这一点，在本书最后两章中，提供了两个带有 GitHub 微量代码提交并能涵盖前面所讨论的主要测试原则的编程实例，来供读者体会作者用测试驱动开发方法开发Java新项目和用测试来驯服已有JavaScript烂代码的详细编写过程。这对读者正是 “授之以渔”。这意味着，在阅读本书前面讨论原则和模式的内容时，遇到不懂的地方也不要紧，在重现最后两章的带有详尽提交注释的微量代码提交的实例的编写过程中，就能体悟到前面所讨论的原则，并能从其中悟出更多的“道”。

重点突出的测试实践原则，富有特色的测试实践模式和微量提交的编程实例，令我对本书爱不释手，最终选择翻译本书，并在本书的启发下完成了我自己的拙作《驯服烂代码》。

伍斌

匠艺程序员，公益编程操练社区“bjdp.org北京设计模式学习组”创办者

前言

在过去的几十年里，精益（Lean）生产方式已经革命性地彻底改变了制造业。像全面质量管理（TQM）、准时生产（Just-In-Time）、约束理论（Theory of Constraints）和丰田生产方式（Toyota Production System）这样的框架已经大幅地改善了汽车质量和制造质量的总体状况，并造就了一个充满活力的竞争局面。各种敏捷软件开发方法都将精益生产的原则引入到软件产品开发的知识体系中，但这些原则需要改造，以适应软件开发这个不同于机械制造的领域。

像那些为了提升客户满意度和减少产品整个生命周期的维护成本，而将质量内建到产品中的想法，已经造就了测试驱动开发（TDD）和其他测试先行（test-first）及尽早测试（test-early）的方法的诞生。无论遵循哪种方法，都需要了解可测试的软件看起来是什么样子的，并掌握广泛的技术来成功地实现测试。笔者已经发现，在各种测试的调理方案中，上述原则和实践现状之间所存在的差距，往往是一个未被引起重视的失败根源。“使用测试驱动开发”这句话说起来容易，但是当面对一个项目时，许多开发人员都不知从何下手。

当向人们展示如何运用测试驱动开发或者尽早测试开发时，经常发现那些，其中的一个障碍就是编写测试的技术性细节。如果以数学函数的方式来执行一个方法，单纯地将输入转换成所期望的输出，那是没有问题的。但是很多软件都有副作用、行为特征或上下文约束，这些都使得对其进行测试并不是那么容易的。

本书源自下面这样经常性的需求，即要为开发人员展示如何针对那

些曾经让他们挠头的具体场景进行测试。当这种需求来临时，我们总是会坐下来，为那些惹麻烦的代码编写一个单元测试，使开发人员能够拥有一个新的工具。

本书内容

本书讲述了如何对所有的软件进行轻松的例行测试。主要侧重于单元测试，但是其中的许多技术也同样适用于较高层次的测试（higher-level test）。本书将为读者提供一些工具——一些实现模式（implementation pattern），这些工具几乎可以测试任何代码，用它们还能够识别出代码什么时候需要改成可测试的。

本书目标

本书将帮助读者：

了解如何对所有的代码进行轻松的单元测试；

提高软件设计的可测试性；

针对代码找到适用的测试替代方案；

以和应用程序的生长相适应的方式来编写测试。

为了达到这些目标，本书将提供：

20多个测试代码的具体技术和许多例子；

在各种场景下使用正确的测试技术的指南；

一种能帮助读者更加专注地进行单元测试的方法。

本书读者

本书主要针对那些希望提升自己在代码层面的测试技能以增强代码质量的、专业的软件开发人员和在测试领域的软件开发人员。本书尤其对那些测试驱动开发（TDD）和尽早测试的实践者们特别有用，可以帮助他们从一开始就确保其代码的正确性。本书中的许多技术也同样适用于集成测试和系统测试。

读者背景

本书假定读者具有以下特点。

熟练掌握面向对象的编程语言，能够阅读和理解 Java、C++和其他语言的样例代码，并能将从中学到的知识运用到自己所运用的编程语言中。

熟悉代码层面测试的概念和像JUnit这样的xUnit测试框架的工具。

了解或可以查阅到有关设计模式和重构的信息。

本书章节

第一部分（第1～5章）涵盖了能成功指导测试的原则和实践。第1章将本书所讨论的方法放到工程的背景下，讨论了工程、匠艺（craftsmanship）和首次优质[1]（first-time quality）的一般概念，以及一些针对软件的具体问题。第2章探讨了意图（intent）的作用。第3章描述了一种能帮助读者更加专注地工作的测试方法。第4章讨论了设计和可测试性之间的相互作用，其中包括了一些能够帮助拓展测试工作的想法。第5章介绍了一些能够用来指导做出测试决策的测试原则。

第二部分（第6～13章）详细介绍了测试的实现模式。首先，第6章介绍了bootstrapping测试，并讨论了相关技术的基本类别，而第7～12章则对第6章介绍的内容进行了详细的阐述。同时在第9章对意图这个概念进行了更加深入的研究。第13章则通过引入一些能确定性地重现竞态条件[2]（race condition）的技术，在技术上更深入地探究了许多人认为不可能办到的事情。

第三部分（第14～15章）详细描述了如何把本书前面所讨论的原则和技术，运用到两个真实工作中的例子上。第14章探讨了使用测试驱动开发来从头创建一个Java应用，展示了如何开始以及如何将上述技术运用到一个严格定义类型的语言上。第15章选择了一个未写测试代码的开源JavaScript jQuery插件，来为其添加测试代码，展示了驯服用动态语言编写的遗留代码的方法。在描述过程中，这两个例子都包含了详细的GitHub的代码提交历史信息，以供参考。

同道前辈

人类所有的进步都是建立在他人先前努力的基础之上的。本书所讨论的内容正是在过去15年计算机发展的影响下成长起来的。下面的列表并不详尽，希望列表中有遗漏或宣传得不够到位的地方不会冒犯这些前辈。

敏捷宣言（Agile Manifesto）的推动者们和签署者们。

早期敏捷开发方法的先驱们，如极限编程（eXtreme Programming）的先驱Kent Beck。

提出设计模式的“四巨头”[3]（Gang of Four）和提出重构的Martin Fowler。

软件匠艺运动和人称“Bob大叔”的Robert C.Martin。

最近Michael Feathers和Gerard Meszaros在软件测试领域所做出的开创性的工作。

很有幸曾与上述杰出前辈的一些原团队的同事们一起共事过。

---

[1].首次优质，是指第一次就把产品质量做好的理念，以消除后来检查产品质量缺陷的成本和时间。——译者注

[2].竞态条件，指一种电子或软件系统的行为，该行为的输出依赖于其他不可控事件的发生顺序或发生时间（引自wikipedia.org）。——译者注

[3].四巨头，指《设计模式》（Design Patterns）一书的四位合著者Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides。——译者注

致谢

所有的作者都会说，如果没有生活伴侣的支持，他们是不可能把书写完的。对于这一点，直到自己亲身经历写书的过程后，才真正地体会到其中的滋味。如果没有妻子珍妮的持续支持和鼓励，没有她对写书所费时间的耐心，没有她每日所说的“我来刷盘子，你去写书吧。”这样的话语，真的没法完成这本书的写作。

感谢Greg Doench对于头次写书的作者所表现出的耐心，以及他在本书的编辑过程中所提供的经验丰富的指导。Zee Spencer、Eric E.Meyer和Jerry Eshbaugh针对本书的审阅反馈帮助改进了本书的内容，并使其更加专注。希望本书没有辜负他们所提供的反馈。

开源软件项目jQuery Timepicker Addon插件是本书第15章中使用的样例的主体。该插件作者Trent Richardson一直都对把这个开源项目的代码纳入测试的保护之下这件事抱有极大的热心和支持。他接受了迄今为止的所有pull requests[1]。在撰写本书之时，该项目的第一个带有测试的版本上线了。

多年来，笔者曾辅导和管理过多个团队。一个人只有在去教会别人的过程中，才能让自己真正明白一些东西。感谢这些团队让笔者得以成长。

有了《Dr.Dobbs’Journal》和《Software Development》杂志的多位作者的激发，笔者才产生了有关尽早测试方法一些早期的想法。Eric E.Meyer、Jean-Pierre Lejacq和 Ken Faw 展示了有关 TDD 的一些训练有素的方法。笔者的机会，许多都是Christopher Beale 提供的，或者和他

一起创造的。我们的事业交织在一起，这包括我们和Joe Dionise在一起所做的工作。许多早期的设计和架构经验，都是在他们的指导下形成的。

最后，两位教授在不知不觉中对笔者产生了很大的影响，这种影响在本书中达到了极致。20世纪90年代初，曾在密歇根大学任教的Lee教授，向笔者展示了计算机科学可以不仅仅是一种爱好。在硕士课程的第一堂课中，奥克兰大学的 Janusz Laski教授介绍了形式化验证（formal verification）和静态与动态分析方法，对理解和宣传那些支持软件开发过程的工具很有帮助。

---

[1].pull request指的是程序员在GitHub修改代码后所发出的用来通知其他程序员有关该代码变动的请求。GitHub是一种基于Web的使用代码版本控制系统Git的软件开发项目托管服务，用来管理源代码的版本变更。——译者注

第一部分 测试的原则和实践

测试，尤其是用代码编写的自动化测试，贯穿软件工程的始终。无论是通过测试驱动开发、持续集成、行为驱动开发、持续交付、验收测试驱动开发、实例化需求（specification by example）、集成测试、系统测试，还是单元测试，每一个参与基于软件的产品开发的人，都会有机会编写自动化测试。敏捷、精益（lean）[1]和软件匠艺运动都赞成使用高水平的测试，来支持快速开发和高品质的产品。

软件工程领域的思想领袖们，把基于代码的测试作为专业开发人员的保留节目来加以推动。而人称“Bob 大叔的”Martin[CC08，CC11]则把它称为软件匠艺。Gerard Meszaros[xTP]整合了与其相关的词汇。Steve Freeman和Nat Pryce[GOOS]描述了如何使用健康的测试剂量来培育软件产品的开发。Michael Feathers[WEwLC]为大家展示了如何通过测试来拯救旧代码，他甚至把遗留代码（legacy code）定义为没有对其编写测试的代码。Kent Beck[XPE]等人阐释了如何通过使用包括测试驱动开发方法在内的手段，来将编程发挥到极限。

上述杰出人物中的每一位，都描述了在软件开发中使用测试的一个重要部分。他们所撰写的每一本技术书籍，都使用了大量的示例来帮助大家领会他们各具特色的教导。

然而，当笔者用测试的调理方案辅导了多个团队之后，却一次次地发现阻碍采用测试的障碍，既不是对测试流程缺乏了解，也不是对相关概念的误解，或者词汇的缺失，更不是对测试实践的价值的怀疑。虽然所有这些障碍都因时因人而异，但是一个最常见的、一直没有很好地加

以面对的障碍，就是没有充分地理解测试代码的机制和编写可测试代码的机制。

虽然本书专注于有关编程测试和可测试性方面的多种机制——各种实现模式，但是如果能带着对这些技术背后的概念框架的理解来阅读本书，那么就可以更加巧妙地运用这些技术。本书第一部分探讨了工程和匠艺之间的关系，并讨论了测试在匠艺中所起的作用。随后这一部分内容又展示了如何分别为既有代码和未写代码编写测试，如何更好地理解代码意图，可测试性是什么样子的，以及在编写测试时应该记住的指导原则。

第1章 工程、匠艺和首次优质

我们这个行业在很大程度上，把诸如软件开发人员、计算机程序员、编程牛人[2]和软件工程师这些头衔当作同义词而混为一谈。然而，由于“工程师”一词在制度上的严格规定，导致“软件工程师”一词在有些地方是不能使用的。软件这门学问一直以来都渴望成为一门工程学科，跻身于比较传统的工程学科（如土木、机械、电气和航空航天）行列。这一点在最近这些年里开始得到人们的承认，但是这期间还是伴随着相当数量的有关执照、认证和所需最小知识体系这些问题的争论。

与此同时，软件匠艺运动一直在稳步发展。伴随着在世界各地纷纷出现的软件匠艺组织和像CodeMash[3]和Global Day of Code Retreat[4]这样的活动的举办，越来越多的开发人员希望专注于提升自己的编程技艺。

1.1 工程与匠艺

有一件事可以用来区分软件工程与其他工程学科，那就是软件开发人员需要经常和全面地实践软件构建过程的各个环节；而来自其他工程学科的工程师们，一般只对和他们的领域相关的构建过程感兴趣，并拥有其技能，但是他们很少需要将其作为一项每日必做的活动。

汽车工程师可能会将时间花费在装配线上，但他们不会经常呆在那里。土木工程师可以监督和检查桥梁或建筑物的建造过程，但他们很少花时间去铆铆钉、浇筑混凝土或者穿挂电缆。与软件工程师这样的全程投入的职位最接近的，恐怕也就是极少数的几个身兼航空工程师的试飞

员了，因为他们会参与自己将要试飞的飞机的设计、施工、检查和验证。

当在软件领域出现了这种复兴运动[5]后，其结果就是我们往往会淡化手艺（craft）、工程和创作之间的界限。另外，在软件开发中，还要去参与了解问题领域（problem domain）的内容，这就难怪会在工作中把所应关注的问题与其他问题纠缠在一起了。但讨论这些的用意何在呢？

专业的软件从业者，会在多个层次上进行编码、设计、架构、测试、度量和分析。这其中的一些内容，很明显就是工程。进行设计和由他人进行验证也很显然就是工程。但不管怎样切分，编码的行为，就是匠艺的行为，就是做出来的行为。当我们的手指在键盘上敲击的同时，我们或许是在并发地做一些“工程”，但这还是在做一门手艺。

1.2 匠艺在首次优质中的作用

匠艺蕴含着技巧。在编码中所使用的技巧，决定着编码的结果。无论对软件做了多少架构、设计或构思的工作，一个差劲的实现就能够破坏掉所有这一切。

作为传统，我们过去会依赖这样的验证，即通过自己的注意力或一个质量保证小组来手工确保软件行为的正确性。一位开发人员曾对笔者说：“我先编写代码，然后再训练它做正确的事情。”虽然他那种最终实现正确性的用意值得称颂，但是此话也表露出他对最初就把产品做好缺乏意愿和关注。当他花了1周时间在一个复杂系统中修改了一些核心代码后，令人哭笑不得的事情发生了，他又要求花3周时间来测试修改的结果。

精益制造是运作在将优质构建到产品中的原则之下的。返工是一种形式的浪费，需要从系统中消除[6]。而先编写软件以待将来发现bug时

再重写或修补，就是返工。

测试那种本应正确的软件，就可以被看作是一种浪费。由于尚未找到创建没有缺陷的软件的方法，所以在编程之后进行测试在一定程度上是必需的。然而，当软件缺陷被发现、修复和重测时，对该软件所做的多于一次的测试显然是低效的。

另一种形式的浪费是库存，可以将其视为机会成本[7]。在修复 bug 上所花费的时间，也正是该软件中那些被正确编写的部分无法交付给客户或提供商业价值的时间。这个时间同时也是开发人员原本可以做包括职业发展在内的其他有价值的行为的时间。

提升工作中的匠艺水平，能为个人和公司带来成倍的回报。它能令人用更少的时间和更少的浪费，来交付更多的价值，并带来更多的个人满足感。

关于方法的一点看法

首先需要事先声明，笔者是软件开发中敏捷和精益方法的热衷者。然而，笔者也坚信，世界上不存在一个或一组方法，能够适合每个团队、公司、项目或产品的情况。笔者从事过同时使用敏捷和瀑布开发方法的相当标准的软件项目，也把敏捷开发过程嵌入到瀑布式开发过程中，甚至在很大程度上，把敏捷技术运用到涉及实时控制和有关人身安全的产品上，还曾经雄心勃勃地以整个组织转型为目标，把敏捷和精益方法运用到前沿的Web开发上。

无论使用哪种方法，项目中可能都已经有了某种形式的自动化测试，哪怕就是为了省却自己去按下按钮的麻烦。而更有可能的是，为了频繁地运行单元测试、集成测试、系统测试、组件测试、功能测试和其他形式的测试，项目已经拥有了某种形式的持续集成系统。

只要有测试自动化的地方，就会有人编写代码来测试其他的代码。对于这种情况，本书所讨论的技术将会很有用。对于单元测试或隔离测试（isolation testing），本书所讨论的几乎所有技术都适用。而对于更

大范围的测试，或许可以选择来运用的技术就会少很多。一些技术只适合于某些编程语言，而其他技术则可被用于几乎任何编程语言或编程范型（programming paradigm）。

笔者会提及一些敏捷软件开发中常见的测试方法。即使有些方法不合某些人的开发口味，也不要让这一点成为去尝试它们的拦路虎。无论是实践测试驱动开发、测试先行、尽早测试或者测试后行（test after），都仍然需要驯服被测代码。祝读者有一个快乐的测试！

1.3 支持软件匠艺的实践

现在问题变成了这个样子：如何才能不断地提升软件从一开始就能被正确编写的可能性？首先，让计算机做它们所擅长的事情，即那些对人类来说，耗时易错的、需要死记硬背的、机械的行为和详细的分析工作。而利用自动化代码质量工具的不同方式，会导致有关软件卫生（software hygiene）的不同分类[8]。

风格（style）：空白字符、大括号、缩进等。

语法（syntax）：编译、静态分析。

简单性（simplicity）：圈复杂度、耦合、YAGNI[9]。

解决方案（solution）：运行时、正确性、是否能工作、TDD、BDD。

努力的伸缩性（scaling the effort）：持续集成、持续交付。

集成开发环境（Integrated Development Environment，IDE）[10]能让我们轻松地拥有从“风格”到“解决方案”的丰富功能。代码补全（code completion）、语法加亮（syntax highlighting）和项目与文件的组织与管理这些仅仅是最基本的，重构支持（refactoring support）和其他工具集成才是最诱人的。如果能够办到的话，可以在IDE中配置所有的工具，使它们能够提供直接的实时反馈。编写代码和获得反馈之间所消耗

的时间越短，就越接近首次优质的目标。

不要让自己担心大括号的位置和空白字符的数量和类型[11]，可以配置IDE或构建工具来做这些有关代码格式的事情。理想情况下，IDE会做这些事情，使得代码格式的设置能够立即生效。而随时能在本地运行的构建工具也会为构建目标做这些事情。

可以用静态分析工具进一步增强对代码“语法”和“简单性”这两个分类的保护。像下面这些工具，如针对Java语言的Checkstyle，针对JavaScript语言的jshint或jslint，针对Perl语言的perlcritic，针对C和C++语言的lint，或者针对其他所选用语言的类似的工具，都可以用来帮助检查有关代码风格方面的问题。而像 PMD[12]或 FindBugs这样针对 Java语言的更加高级的工具，则在代码格式规范和风格检查之外更进了一步，它们直接检查诸如未使用的和未初始化的变量、框架特定的约定（framework-specific conventions）和复杂性度量等各个方面。有些工具甚至可以进行扩展。特别是PMD这个工具，具有非常灵活的功能，它能基于使用抽象语法树的XPath表达式来定义规则。PMD还有一个称为CPD的模块，即复制-粘贴探测器（copy–paste detector），能用来加亮那些复制而来的代码，以便进行重构并复用。

代码覆盖率工具能通过数值和图形的方式显示代码中哪些部分已经被执行，来指导测试并帮助开发人员找到正确的解决方案。本书后面的部分将更加密切地关注代码覆盖率。

代码生成工具能够根据源代码或其领域特定语言[13]（Domain-Specific Language，DSL）来创建成千上万行代码，将可能帮助完成“简单性”和“解决方案”这两个分类的工作。目前已经有可以生成网络服务接口、数据库访问代码和更多其他代码的工具了，尽管在许多情况下，这样生成的代码都具有代码腐臭[14]（code smell），因而丧失了复用的机会，以及完成“简单性”分类工作的机会。在某些情况下，我们甚至能够找到帮助生成测试代码的工具。如果能够信任代码生产器，那么就可

以跳过对所生成的代码进行测试的步骤，从而节省同时在两条战线作战所耗费的时间和精力。

对于工具要讨论的最后一点是，持续集成系统除了能够针对代码进行构建和测试工作以外，还能运行所有这些工具，以使代码能够独立于本地开发环境，从而帮助完成“伸缩性”的工作。持续集成所构建的结果，应该可以通过Web、监视器和电子邮件来获取到。许多团队已经将他们失败的持续集成的结果与警笛或警灯绑定在一起，使其更加明显。为取得最佳效果，持续集成的构建应该在几分钟之内就能报告结果。持续集成提供了另一种提供快速反馈的形式，用牺牲一些响应时间作为代价来换取对代码所做出的更加全面的评估。

测试

在所有能够利用的有助于提升匠艺的实践中，令人最为获益的是测试。有人可能会说：“我们已经对软件进行了多年的测试，但还是能够发现许多 bug。”取得进步的关键其实就在于此。

在一个精益系统中，我们要做的是努力阻止缺陷，而不是捕获缺陷。而传统以来，软件测试一直是作为捕获缺陷的一种方法。所以，我们一直沿用着一个低效的过程这一点，不应该令我们感到惊讶。越早地进行测试，就能越早地捕获缺陷。

这是不是在说我们还是仅仅在捕获缺陷？如果是在编写生产代码后才编写测试，那么就是仅仅在捕获缺陷。如果能弥合创建缺陷和捕获缺陷这两者之间的间隙，那么代码在这两者之间所发生的变化就会更小，那么就更有可能让刚刚创建的代码的上下文在头脑中记忆犹新。如果在编写代码后立即就编写测试，那么测试就能马上验证有关上述代码的概念和假设，并保证开发走在正确的方向上。

然而实际上，在bug发生之前，我们就有机会捕获到它们。测试先行和测试驱动的开发方法，都是在编写代码之前先编写测试代码。当使用测试先行的方法时，可以用自动化和可执行的形式来捕获设计意图。

此时，专注于用阻止缺陷而不是创建缺陷的方法来编写代码并令其能够工作。这里所编写的测试，就是对下一步的意图所进行的持久化的加固，它不仅能帮助我们把事情做好，还能帮助我们把事情做对。记住，一件错误的事情，不管做得多好，它都仍然是一个bug。

在上述模型中，测试驱动开发（Test-Driven Development，TDD）是测试先行方法的一个子集。测试先行允许我们在编写生产代码之前，根据自己的想法，编写出又多又全的测试代码。虽然测试先行把质量工作放在了首位，但它在推测要做的事情和所要采取的形式方面，也给了我们更大的自由度。一旦所设计的解决方案比所需的还通用，那么就不仅要面临在该方案的创建上花费更多时间的风险，还要面临背上更沉重维护负担的风险，这是一个需要一直承担的无限期的成本。此外，若前面编写了太多的测试，一旦发现解决问题的更好方法，就会增加修改的成本。因为有这些原因，TDD为编写软件带来了更加有纪律性和精益性的方法。这种方法仅仅当需要时才编写恰好所需的代码。正是对以前不足够的预见和不完整的理解的一个个发现，指引着编程的方向。

无论编写测试的时间与编写生产代码的时间这两者之间处于怎样的关系，本书所提出的原则，都将会有助于进行有效的测试。但是，测试进行得越早，本书所提供的帮助才会越大。敏捷开发和测试的其中一条准则是自动化测试三角（Automated Testing Triangle，见图 1-1）。传统上，我们一直专注于系统级别的测试，而这种测试不是因为验证得过于深入而趋向于脆弱，就是因为要达到便于维护的目的而趋向于粗略。虽然我们永远都无法取代系统级别的测试，但是可以将一部分测试转移到级别低一些的单元测试里。

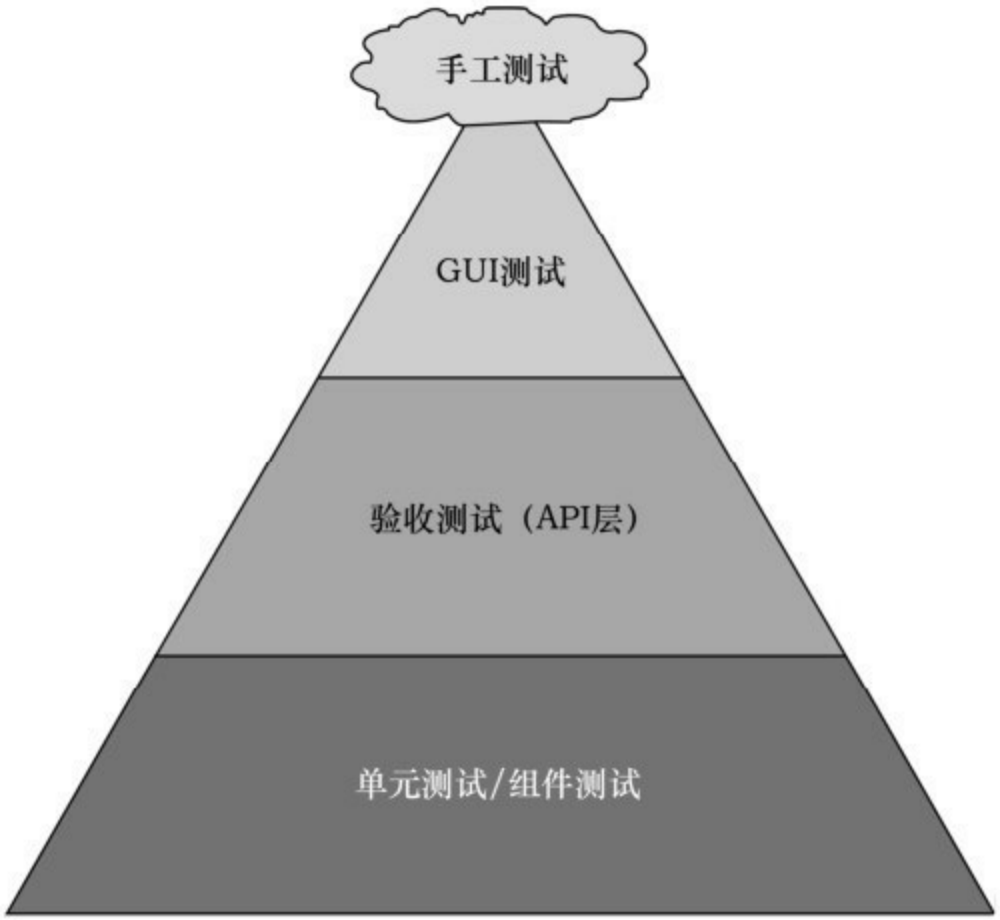

图1-1 自动化测试三角对更小级别的测试进行了高投入，因为只要编写得当，这些测试就会不那么脆弱，且更容易加以维护

笔者听说有人反对在上图的底部进行大量的单元测试，因为它们很脆弱、难以维护。然而编写得当的单元测试的特点恰恰与上面所说的相反。本书中所讨论的原则和技术将有助于理解单元测试变得难缠的原因，以及如何避免这种情况发生。

1.4 在代码检查器的约束下进行单元测试

在前面笔者曾经提到，应该使用工具来将匠艺所关注的内容尽可能多地进行自动化。如果用测试来驱动开发的质量，那么也需要将良好的匠艺运用到测试上。我们的工具不仅应该检查应用程序的代码，还应该

以几乎同样的标准检查测试代码。笔者在这里说“几乎”，是因为一些针对生产代码的用启发方式对代码进行复用的方法，需要在测试代码中放得宽松一些。例如，在两个不同的测试里使用重复的字符串值，能够使这些测试彼此相互独立，尽管使用不同的字符串会起到更好的效果。同样，虽然针对复用的重构能够为测试代码带来好处，但是不应该将测试重构得把设置（setup）、执行（execution）和验证（verification）这三者[15]彼此之间的界限给搞模糊了。

将静态检查器的约束应用到单元测试时，需要缩小该约束可以应用的解决方案的范围。例如，假设我们想在嵌套类中使用一些常量值，同时使用一系列默认的 PMD规则，那么我们就会被迫将许多上述常量值被移到测试方法之外。可以用许多方法来解决这个问题，但如果这个问题成为了团队的负担，且没有带来多少价值，那么就可以将正常规则的一个子集或者甚至是一个等价的定制集合，应用到测试代码上。

不管怎样，用与应用程序代码相同的标准对待测试代码，会在将来很长的时间里得到回报。如果读者觉得本书中一些例子的实现形式得看起来很奇怪，那么可以试着想象一下静态检查器打开时实现这些代码会出现的情况，或许就能明白原因了。

1.5 针对覆盖率的单元测试

代码覆盖率工具为编写、雕琢和维护测试提供了极好的帮助。这些工具用图形化的方式显示了哪些生产代码在测试中得到了运行，从而让我们得知测试是否命中了目标以及哪些生产代码还未被命中。

代码覆盖率所提供的并不是一个有关生产代码如何被执行的详尽情况。代码的许多构造会以不能被静态确定的方式分支到多个代码路径上。各种编程语言中的一些最强大的功能（那些提供最有用的抽象的功能）会以各种让静态分析无从探知的方式来工作。当使用代码覆盖率这

个机制时，虽然覆盖率工具能够显示出哪些生产代码已经被执行，但是没有办法预先知道会存在多少排列组合，来从代码运行中计算代码覆盖的程度。

数据驱动执行（data-driven execution）通常以从数组或表中进行查询的方式来替换链式的条件语句（chained conditional）。在代码清单1-1中，覆盖率工具虽然能给出针对这个情况的语句覆盖率的准确评估，但是无法给出表中那些转换点所代表的每个条件，是否像用if...else语句链实现那样被覆盖。假如rateComment数组中的那些函数是在其他地方定义的，而且有被复用的可能性，那么覆盖率工具就会漏掉更多的情况。

代码清单1-1 一个展示代码覆盖率盲点的用JavaScript编写的数据驱动执行的例子

```
function commentOnInterestRate(rate) {
  var rateComment =[
    [-10.0,function() { throw " I’m ruined!!! " ; }],
    [0.0,
      function() { gripe( " I hope this passes quickly. " ); }],
    [3.0,function() { mumble( " Hope for low inflation. " ); }],
    [6.0,function() { state( " I can live with this. " ); }],
    [10.0,
      function() { proclaim( " Retirement,here I come. " ); }],
    [100.0,function() { throw " Jackpot! " ; }]
  ];
  for (var index = 0; index ＜ rateComment.length; index++) {
    var candidateComment = rateComment[index];
    if (rate ＜ candidateComment[0]) {
      candidateComment[1]();
      break;
```

```
        }
      }
    }
```

动态分派（dynamic dispatch）提供了另一种机制，在这种机制下，代码的执行无法得到静态分析。看看下面这行简单的JavaScript代码：

```
someObject[method]();
```

这个变量method的值只有在运行时才能知道，这使得在一般情况下，覆盖率工具不可能确定代码可用路径的数目。在JavaScript中，方法的数目会随着代码运行的过程发生改变，所以即使看到一个对象的已知方法，也不能凭借它来计算覆盖率。这个问题并不限于动态语言。即使是像C++和Java这样的标准的静态类型语言，都有动态分派功能，其中，C++可以通过虚函数和指向成员的指针引用（pointer-to-member reference）获得动态分派功能，Java可以通过反射（reflection）来获得。

类似的其他情况会发生得更加自然一些，可以称为边界情况的语义处理（semantically handled edge cases）。在这些情况中，编程语言或运行环境会把一些异常条件自动翻译为一些变体，系统在处理这些变体时，会采用与正常执行所不同的方法。Java语言的unchecked异常和那些不必在方法签名中声明的异常，都会发生这种情况。这还包括在试图使用一个 null 引用时所抛出的最臭名昭著的、可怕的 Null PointerException（空指针异常）。而对于被零除（divide-by-zero）这样的错误，不同的语言则有不同的处理方式，这些方式包括整个应用崩溃、抛出可捕获的异常和从计算中返回NaN[16]。

另外，代码覆盖率也是具有欺骗性的。覆盖率仅显示了被执行的代码，而不是那些被验证过的代码。覆盖率的有用之处仅仅和驱动它的那些测试相同。即使是富有善意的开发人员，在覆盖率报告面前也会变得沾沾自喜。下面的这些轶事，是笔者以前带过的一些团队无意中所犯的

错误，通过它们，可以想象到滥用覆盖率所造成的人为的后果。

开发人员在编写了测试的setup和execution部分之后，就开始有些分心了，因为一会儿就要回家过周末了。等到了周一上午，在他运行代码包时，因为失去了过周末之前的那些背景信息的提醒，他在验证实现了全覆盖后，就把代码提交了。事后对代码进行检查发现，他所提交的测试虽然全部执行了被测代码，但是却没有包含任何断言。该测试实现了代码覆盖并运行通过，但是代码包含了bug。

开发人员编写了一个Web控制器，用来控制这个应用程序内两个页面之间的交换。由于不知道一个特定条件的目标页面，他使用了空字符串作为占位符，并且编写了一个测试来验证该占位符按照预期返回。该测试运用通过且显示全覆盖。两个月之后，一位用户报告说，应用程序会在一个晦涩的条件组合下返回主页面。根本原因分析显示，那个空字符串的占位符即使在正确的页面被定义后，也未被替换。那个空字符串会被拼接到应用程序的域和上下文路径后，重定向到主页面。

开发人员最近发现了 mock[17]的价值，并在 mock 魔力的吸引下编写了一个测试。他在不经意间mock了被测代码，并让测试运行通过。而其他测试偶然间使用了该被测代码，结果导致该代码被计入了覆盖率的统计。该系统其实没有达到全覆盖。后来为了有意义地增加覆盖率而对测试进行检查，结果发现了这个过失，并编写了一个测试去执行被测代码，而不是仅运行mock。

代码覆盖率是指南，但不是目标。覆盖率有助于编写正确的测试来执行代码的语法运行路径。人的大脑还是需要多用一用的。同样，人们所编写的测试的质量，依赖于这些人在编写测试时所投入的技能和注意力。覆盖率对于检查意外或故意造成的低劣的测试无能为力。

请注意在覆盖率这一点上，前面并没有谈及要使用哪种覆盖率指标。几乎所有人都认为要使用语句或行数覆盖率。几乎所有的覆盖率工具都提供语句覆盖率的统计，但这仅仅是起点，就好比一进赌场就能得

到的那些入场筹码而已。不幸的是，许多人就到此止步了，有时甚至用更加薄弱的类和方法/函数覆盖率来作为对语句覆盖率的补充。而笔者则更喜欢最低限度地使用分支和条件覆盖率[18]作为补充。一些工具包含分支覆盖率功能。可悲的是，没什么工具能包含条件及更多覆盖率的功能。一些额外的指标包括循环覆盖率（每个循环必须执行0次、1次和多次）和像def-use链[19]这样的数据路径指标[LAS83，KOR85]。在Java 语言中，开源工具 CodeCover （http://codecover.org）和Atlassian公司的商业工具Clover在这方面做得不错。Perl语言的Devel::Cover工具也处理了多个指标。PMD会对UR、DU和DD异常[20]的数据流分析发出错误和警告信息，尽管这些信息需要再改进一下。

笔者似乎十分喜爱那些高可用、高可靠和注重安全性的软件。笔者以前带领团队开发过紧急响应软件、实时机器人控制（有时需要与非人眼安全的激光技术相结合）和高利用率的构建和测试系统，对于这些项目来说，停机时间意味着真实的业务遭受延误和损失。笔者还曾带过一些项目，在这些项目中，100%的语句、分支、条件和循环覆盖率仅仅被当作全面单元测试的一个里程碑。相对于应用到这些系统上的许多其他级别的测试，那时除了仅从单元测试中统计覆盖率以外，还要仅从一个类的测试中来统计该类的测试覆盖率。那些其他类偶然使用的覆盖率不会被纳入统计。

一般情况下，笔者发现当语句覆盖率在50%左右时，优质代码的回报就开始显现出来。当这个数达到80%时，回报就会变得有意义。如果达到了100%这个里程碑，我们就能获得显著的回报，但是这样做的成本取决于开发人员在测试和编写可测试的、低复杂度的、松耦合的代码这些方面的技能[21]。先暂且不管决定回报的成本需要依赖于特定的问题领域这件事，现在大多数团队还没有对覆盖率准确地做出评估以便进行权衡的经验。

通常情况下，团队会基于不值得做　100%覆盖率这样的观点，来选

择一个小于100%的数作为目标覆盖率。一般来说，有观点认为，不宜做测试的情况分为两种：琐碎的和困难的。反对编写困难的测试的观点，则集中在算法复杂项或错误路径这两个方面。

琐碎的测试包括对像getter和 setter 这样的简单方法的测试。确实，测试它们很无聊，但是对它们进行测试花不了多少时间，而且这样一来，就永远不必怀疑覆盖率的缺口会是由这些琐碎的测试的疏漏而造成的了。

算法复杂的代码最有可能是系统的独家秘方的心脏——就是靠这个独家秘方把自己的软件与其他公司的软件区分开来。这听起来好像是需要测试的东西。如果实现[22]的复杂性阻碍了测试，那么代码就可能需要进行设计的改进，而这个改进可以通过可测试性的需求来驱动。

错误路径的测试是用来验证系统中那些最有可能令客户心烦的部分的。很少能在有关软件的线上的评论中看到赞誉系统没有崩溃并优雅地处理了错误这样的内容。发生了崩溃、丢失数据和其他严重失效的软件，会得到非常差劲和公开的评论。抱着永远乐观的心态，开发人员希望并几乎假设那些错误路径很少会被走到，但现实是，这个世界是十分不完美的，所以这些错误路径肯定会被走到。测试错误路径就是在逆市中对客户的良好意愿的投资。

最终，自己要对业务所需的测试程度做出决策。建议在做决策时，要从一个熟练工匠（即能够实现业务所要求的任何覆盖率的人）的角度来考虑，而不要因为看起来很难就避免高覆盖率。本书的目的就是要用模式、原则和技术把读者武装起来，使读者能站到熟练工匠的角度上做决策。

第2章 代码的意图

当读者用所喜爱的IDE来工作，并用到了其中各种吸引人的附加功能（语法检查、自动完成、静态分析和其他功能）时，会不会感叹缺少一个尚未被发明的特定功能？对，笔者指的就是意图检查器。对此大家都很了解。当我们在思考时，就会需要这样的功能：“我希望它能按照我的意思而不是按照我敲的东西来编程！”或许在奋力编写一个棘手的算法时，就会想要这个功能。可能在调用了这个功能后，就发现了一个愚蠢的敲错一个字符这样的 bug。不管在什么情况下，所面临的就是将意图转化成实现的复杂性。

另一方面，大家以前都问过像这样的问题：“这段代码是做什么的？”或者更极端地问：“这个开发人员当时是怎么想的？”测试所做的所有事情，就是要验证实现与显性或隐性的开发意图之间的匹配性：显性表现在代码应该要完成一些目标；隐性表现在代码在完成上述目标时，也应该具有特定的可用性、强壮性等这些特征，而不管这些是否被特别地考虑过[23]。

2.1 意图都被放到哪里去了

意图是一个捉摸不定的东西。在更广阔的社会里，存在着“意图式生活”（intentional living）这个词。在实践意图式生活时，会试图把所做的每一个行为当作是刻意的，而不是当作习惯性的或偶然的，同时要考虑到行为发生的地点和在这些行为的背景下会出现的后果。意图的明确性也常常会与极简主义的那些方法联系起来。上述道理虽然显而易

见，但是要实践这种生活，需要用更多的纪律和努力。

软件需要同样的专注力。笔者在参与开发了多个有关人身安全方面的项目和产品之后，对于所写代码的后果变得很敏感。这些后果可以在某种程度上扩展到整个软件。如果电子邮件程序将邮件发送给了错误的收件人，那么就会违反保密性、破坏信任，有时还会带来重大的金融和政治影响。一个字处理程序的恼人崩溃似乎微不足道，特别是在没有造成数据丢失的情况下，但如果这种情况出现几百万次，那么它就会演变成大量的刺激，丧失生产力。

极简主义也适用于软件。代码编写得越少，要维护的内容也就越少。要维护的内容越少，必须要理解的内容也就越少。必须要理解的内容越少，犯错误的机会也就越少。更少的代码成就更少的bug。

笔者有意囊括了超乎安全性、金钱和生产力之外的因素。渐渐地，软件被集成到了周围一切的事物中，包括被集成到伴随日常生活的各种设备中。这种无处不在使得软件对我们的生活质量和精神状态所施加的影响不断增大。所以上文中包含了像“信任”和“刺激”这样描述影响的字眼。产品的意图包含了非功能性的方面。那些取得巨大成功的公司，不仅撷取了用户的头脑，也俘获了他们的内心[24]。

2.2 将意图与实现分离

实现仅仅是完成意图的众多方式中的一种而已。如果一个人能够对实现与意图之间的边界有一个清楚的认识，那么他在编写和测试软件时就能有一个更好的心智模式（mental model）。实现与意图极其相像的情况屡见不鲜。例如，一个“奖励获胜玩家”的意图可以实现为“查询获胜用户，遍历他们，然后为每人的成绩清单中添加一个徽章”。实现的语言与意图的表述紧密对应，但这两者并不相同。

明确地划分意图与实现之间的界限，能有助于将测试工作的规模与

软件相匹配。在不掺杂实现元素的前提下，对意图测试得越多，测试耦合到代码的情况就越少。耦合得少了，就不用被迫随着实现中的变化更新或重写测试了。测试改变得越少，花费在测试上的精力也就越少，而这会增加测试保持正确的可能性。所有这一切都会使得在验证、维护和扩展软件上的花费更少，在长远来看更是如此。

也可以将代码的意图与功能相分离。此处所说的分离是将代码原本的工作用意与它实际的行为分隔开。当需要测试实现时，应该对代码本应做的事情进行测试。而对代码所编写出的那些行为进行测试时，若该代码编写得不正确，那么这种测试就会造成一个安全的假象。一个运行通过的测试会告诉我们一些有关代码质量和代码与目的之间契合度的信息。而一个不应运行成功的测试若运行通过了，那么该测试就会在上述信息上对我们撒谎。

当编写代码时，要使用编程语言和框架中的特性来最清晰地表达意图。在 Java语言中将变量声明为 final或 private，在C++中声明为const，在Perl中声明为my，或者在JavaScript中声明为 var，都是表达了有关该变量用途的意图。在像Perl和 JavaScript 这样带有弱参数需求的动态语言中，在 Perl 中传递哈希参数[PBP]和在JavaScript 中传递对象参数[JTGP]时，参数值本身的命名能用于在代码内部更加清晰地记录意图。

2.3 一个能引发思考的简单例子

让我们看看一个使用Java语言的例子。代码清单2-1显示了一个简单的Java类，该类带有几条线索，展示了在构造该类时所表现出来的意图。考虑ScoreWatcher类，它是一个跟踪体育比赛分数系统的一部分。它封装了从一个新闻源（a news feed）获得比赛分数的功能。

代码清单2-1 一个简单的Java类展示了带有意图的构造

```
class ScoreWatcher {
    private final NewsFeed feed;
    private int pollFrequency; // Seconds
    public ScoreWatcher(NewsFeed feed,int pollFrequency) {
        this.feed = feed;
        this.pollFrequency = pollFrequency;
    }
    public void setPollFrequency(int pollFrequency) {
        this.pollFrequency = pollFrequency;
    }
    public int getPollFrequency() {
        return pollFrequency;
    }
    public NewsFeed getFeed() {
        return feed;
    }
    ...
}
```

首先，看一下该类所定义的那些属性（attribute）。编写该类的作者把feed定义为final[25]的属性，却没有把pollFrequency也定义为final。这告诉了我们什么？它表达了这样一个意图：feed应该只能在类构建时被赋值一次，但pollFrequency能够在该对象的整个生命周期中被修改。接下来，在代码中所看到的 pollFrequency 同时具备getter和setter，而feed仅有一个getter，又强化了这一点。

但这仅仅让我们了解了实现上的意图。上面的做法可能会支持哪个功能性的意图呢？可以根据代码的上下文做出一个合理的结论，即对于每一个能够使用的新闻源，应该只恰好分配一个类来封装它。还可以继

续推论，或许对于每一个要被监测的比赛分数，也应该恰好只存在一个用来初始化 ScoreWatcher的 NewsFeed。还可以继续推测，如果存在多个新闻源，那么多个源的管理可能会隐藏在一个新闻源的接口后。这一点需要验证，但是在目前的情况下看起来是合理的。

然而，或许是由于Java语言在表达能力上的限制，上述假设有一个弱点。即便不知道 NewsFeed类的构造情况，我们也能推测出：即使feed这个引用本身不能被改变，但还是有可能通过它来修改它所引用的对象。在C++语言中，可以这样声明属性：

const NewsFeed * const feed;

这个声明不仅表达了指针不能被改变，而且还表达了不能使用指针来改变它所指向的对象。这在C++语言中提供了一个额外的上下文不变性（contextual immutability）的标记，而这一点在Java语言中并不存在。在Java语言中，想让一个类的所有实例都不可变（immutable）还是比较容易的。但是想让一个特定的引用所引用的对象不可变，就需要花费相当多的努力了，或许需要创建一个处理不可变性的代理来封装该对象实例。

然而，这些又是如何改变测试的呢？类的构造——实现——清楚地规定了赋给那个类的 feed在该类的整个声明周期中不应改变。这是意图吗？让我们看看如代码清单2-2所示的验证这个假设的测试。

代码清单2-2 验证代码清单2-1中的新闻源不会改变的测试

```
class TestScoreWatcher {
  @Test
  public void testScoreWatcher_SameFeed() {
    // Set up
    NewsFeed expectedFeed = new NewsFeed();
    int expectedPollFrequency = 70;
    // Execute SUT[26]
```

```
        ScoreWatcher sut = new ScoreWatcher(expectedFeed,
        expectedPollFrequency);
        // Verify results
        Assert.assertNotNull(sut); // Guard assertion
        Assert.assertSame(expectedFeed,sut.getFeed());
        // Garbage collected tear down
    }
}
```

在JUnit中，assertSame断言验证的是，期望的引用和实际的引用都指向同样的对象。回到有关该类的意图的推测上，假设引用到同样的feed很重要这一点是合理的，但是同样的NewsFeed这一点是不是在这种情况下有些超出规格所规定的范围？例如，要是代码的实现为了选择加强初始新闻源的不变性，从 getter将其返回之前就克隆其属性，从而确保任何变化都不会影响ScoreWatcher的NewsFeed的内部状态，那该怎么办？在这种情况下，测试构造器的参数是相同的这一点就不正确了。这种设计的意图，更有可能需要验证feed的深度相等性[27]（deep equality）。

第3章 从哪里开始

知道如何处理问题，具有解决计划，能够简化并增强测试的有效性。在这个意义上说，测试驱动开发提供了一个用来做测试的更简单的框架，因为那些测试都是由下一步希望实现的内容来指导的。然而相比之下，未测试的现有代码给我们提出了一个更加严峻的挑战，因为在这种情况下如何开始有太多的选择。

本章在合理地选择测试方法的形式方面，提供了指导。所用的策略不是激进的、崭新的，但是它会确立一个本书后面所讨论的那些技术所能应用的上下文。希望它能解决一些有关测试策略的问题，以便那些测试技术能被有效地使用。

本章，实际上也是本书，会专注于自动化测试，即Lisa Crispin和Janet Gregory[AT]所描绘的敏捷测试象限图（Agile Testing Quadrant）中的第1象限（Q1）和第2象限（Q2），这两个象限都支持团队[28]。

3.1 一种测试的方法

还没有人发现完美测试的魔法配方（magic formula）。这意味着，作为从业人员，当编写测试时，需要组合运用推理、经验和直觉。本节记录了许多编写自动化测试时所需考虑的问题。

3.1.1 了解范围

需要问自己的第一个问题是：“我要测什么？”这个宽泛的问题需要回答一些更详细的问题，比如：

要验证哪些用例？

要测试全栈、一个集成的子集，还是一个单元？

要验证哪些技术？

要验证哪些架构层次？

是测试新代码、整洁的编写良好的代码，还是拯救一堆遗留代码？

能否将测试问题分解为多个有用的更小的、更易处理的部分？

虽然全栈测试能针对一个完全集成的系统进行终极的验证，但是必须付出代价。全栈测试经常运行缓慢，原因是所有组件之间都存在着交互，这些组件包括网络、数据库和文件系统。全栈测试经常难以区分细微的行为，而且由于必须管理大量的条件来获得期望的结果，所以需要有复杂的攻略来再现错误发生时的条件。最后，全栈测试会因为用户接口和时序交互的问题而变得很脆弱，或者难以实现。所处理的测试范围越小，上述因素的重要性就会越小。

这些技术和架构层次自然会涉及范围，但是它们会引导对测试工具的使用，并会影响对测试替身[29]（test double）的选择。用于测试 Web 接口的测试框架和用于测试ReST服务的框架有所不同，同时也与测试Java类的框架有所不同。针对业务层的测试或许会有一些针对数据库的可替代的选项，然而针对存储过程或ORM[30]映射的测试或许就没有这种选项。

代码成熟度和整洁度能显著地指导所使用的方法。如果开发者正开始一个新项目，那么开发者就拥有创建未来的力量。当专注于手头的任务而不是逆向工程和调试时，一个现有的示范性代码库会让工作更轻松，或许甚至有些乏味。而当面对一个陌生的、如同意大利面条般凌乱的烂摊子时，在有信心对其开始清理和添加功能特性之前，需要具备耐心、洞察力和一系列 Michael Feathers[WEwLC]所介绍的技术，比如特征测试[31]（characterization testing）和接缝识别[32]（seam identification）。

系统中各个部分越彼此独立，就越有信心来分别对这些部分进行测

试。就像系统的分解，将测试任务分解为多个子任务能简化工作并减少整体的耦合性。

3.1.2 测试的概念框架

在概念性的指导方面，可以使用两个原则来指导整体的测试工作。首先，专注于要测试的软件的目的；其次，积极地减少因测试而引入的耦合的程度。现在先考虑目的，下一章将讨论耦合。

代码的目的在不同的层面上会有不同的应用。在系统层面，代码的目的是软件存在的理由：功能特性、需求和用例。对于目的进行严格的评估有助于限定测试用例和验收标准。一个用户界面元素使用一种特定的颜色、大小或以某种特定的方式对齐这一点很重要吗？如果应用程序的值仅仅简单地被持久化并能够被获取到，或许就不需要验证特定的数据库字段，只要执行获取和验证结果的操作就足够了。

在模块或组件的层面来说，代码的目的指的是整个系统的功能。集成测试满足了这个层面的需求。一个模块可以公开实现一个功能特性，直接承担用户期望的功能的职责。此外，一个模块也可以实现一个底层起到支持作用的设计组件。无论采用哪种实现方式，该模块在整个系统中都起到了一定的作用。这个作用应该是明确的，并需要被测试。

在单元测试或隔离测试的层面，笔者喜欢把代码的目的理解为该代码所增加的价值。像墨西哥或欧盟，是有增值税（Value-Added Tax，VAT）的，生活在那里的人或许能明晰这个概念。公司仅对其增加到产品上的那部分价值上缴增值税。也就是说，在计算税费时，会从公司所销售的产品的价值中扣除掉那些原材料或零部件本身所具有的价值。同样，开发者的代码会在代码库和协作者[33]（collaborator）的基础上进行构建，并为其客户增添额外的功能或目的：通过该代码所增添的价值。

定义单元测试已经引发了相当大的争论，但是增值的角度给了我们

对单元测试下定义的另一种方法。许多人使用像Michael Feathers[34]提出的那种单元测试不做什么的定义。笔者认为该定义只是编撰了一些有关测试设计的启发式规则，而笔者更喜欢在一个单元测试的包容性的定义上使用增值的概念。

单元测试是用来验证被测代码所增添的价值的。任何对独立可测试的协作者的使用，仅仅是为了方便而已。

根据这个定义，在同一个类中对一个不可测试的方法的使用，其实就体现了被测代码所增添的价值。由被测代码所调用的一个可测试的方法，应该拥有针对它的独立的测试，从而就不必将其纳入该被测代码的验证范围，除非它给该被测代码增添了价值。可以用mock、stub[35]或其他测试替身的方式来使用其他类，因为它们都是独立可测试的。可以用牺牲测试性能的代价来使用数据库、网络或文件系统，尽管笔者不推荐这样做。

系统的每一个子集，从最小的方法到整个系统本身，都应该有一个明确定义的目的。如果发现自己正在苦苦地测试某个子集，那么或许是因为还不了解它的目的。如果不了解这个子集的目的，那么它可能就有一些实现、设计或架构方面的问题。测试软件的同时，就是透过一个特定的镜头来观察并诊断有关维护和扩展该软件的当前或未来的那些问题。

3.1.3 状态和行为测试

将手头的测试任务进行分类能有助于识别编写测试的最佳方法。确定一个测试是否会验证状态或行为，能有效地引导测试过程。

纯粹的状态验证会执行代码，并检查与操作关联的数据值结果的变化。那些仅仅把输入转换为输出的数学函数体现了状态验证的目标。在函数式的模型中，验证仅仅判断返回值与输入参数正确地相关。在面向对象的模型中，还可以额外验证对象的属性已经被适当地改变了（或没

有被改变！）。进行状态验证的测试通常要提供输入，并验证输出以期望的方式对应到了输入。

行为验证检查被测代码执行了正确的操作，即以超出数据传输之外的各种方式表现出正确的行为。行为验证最纯粹的例子是仅编排一些行为，如代码清单3-1所示。这个方法只调用那些独立可测试的其他方法，以协调它们之间的执行顺序，并将一个方法的输出传送到另一个方法作为输入。行为测试在很大程度上依赖测试替身，经常会使用mock。

代码清单3-1用JavaScript编写的一个纯粹的行为函数的例子，使用了Promise模式的jQuery.Deferred()实现来协调异步计算。为了确保依赖得到满足，并行任务得到协调，它精心编排了该过程的步骤，但该函数自己不做计算

```
function submitTestScore() {
  verifyAllQuestionsAnswered();
  $.when(
    computeScore(),// Returns deferred object
    allocateAttemptID() // Server call returns deferred object
  ).done(
    sendScore()
  );
}
```

一个高度模块化的软件往往不是基于行为的，就是基于状态的。将来有可能会遇到这两者都多少有一点的情况，但是所测试的大部分软件都只会基于前者或后者，了解这一点，就能找到最有效的测试风格。

3.1.4 测试还是不测试

虽然本书的大部分内容都假定读者想要或需要对软件进行测试，但

不应盲目地认定测试是必要的。或许与直觉相反，自动化测试的主要目的不是去验证新的功能。在创建了测试后，该测试的每一次运行，主要是为了验证现有的功能没有遭到破坏。

但是，如果对保护功能性这一点并不介意时该怎么办？如果编写的代码是会被扔掉的，就要考虑一下是否应该为它编写自动化测试。原型、概念验证、演示和试验，这些代码的生命周期或许都很短暂有限。

另一方面，对于想要持续使用的代码就应该进行测试。如果能适当地对代码进行测试，就能增加安全修改代码的能力。对于那些期望能作为客户业务基础的代码和想让其他人来使用的代码，应该用可持续的方法来开发。这也意味着如果打算将上面提到的那些编完即扔的代码产品化，那么就应该在测试的保护下来开发或重写。

另外还需要决定如何对代码进行测试，而在做这个决定时，就要考虑经济因素了。对于不经常改变的高耦合、低质量的代码来说，系统测试和集成测试通常更加便宜和容易。而如果能将单元测试运用到经常会改变的低耦合、高质量的代码上，那么将能获得更多的收益。第1章中介绍的自动化测试金字塔是假定要创建的是可持续的软件，但是这种在经济上完全合理的情况或许不会总出现。或者读者可能已经接手了需要挽救的遗留代码[36]，在这种情况下，可以先将其用特征测试[WEwLC]进行覆盖——基本上是以系统测试或集成测试的形式——直到能将其重构成更加有利于单元测试的形式时为止。

3.2 攻略

可以使用许多不同的方法来进行测试。可以先粗略地进行功能性的测试，然后再有选择性地进行完善。也可以先选择一个功能、架构或设计部分，然后再彻底地深入下去。而使用类似于手工探索性测试这样更加临时和随机的方法也是有价值的。另外还有一种方法可以通过诸如缺

陷密度（defect density）、复杂性或关键性（criticality）这样的各种指标来指导测试。

本节会详细介绍一种无论怎样选择待测的代码或功能都能产生高测试覆盖率的很有用的方法，如图3-1所示。相对于进行广度的测试，这种方法更加适合于进行深度的测试。该方法不仅在使用测试驱动的方法来编写新代码时，效果良好，而且在重现bug、增强既有代码的功能或为既有代码编写测试时，也同样适用。

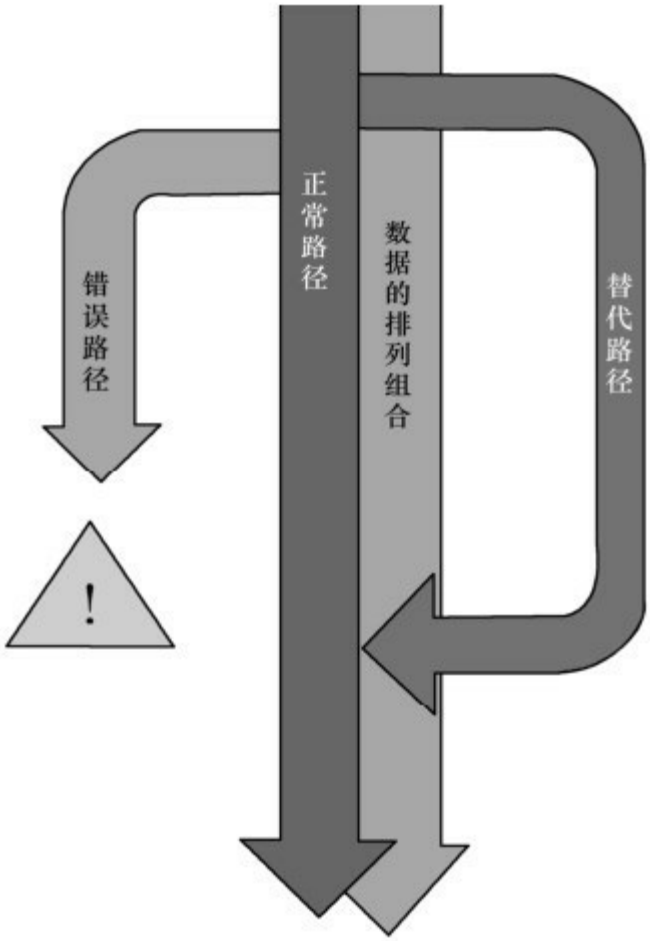

图3-1 测试攻略示意图

3.2.1 测试“正常路径”

代码或功能的“正常路径”（happy path）就是该软件之所以存在的首要目的和主要原因。如果我们用一句话来描述软件所做的事情，那么

这句话所描述的就是正常路径。

对正常路径进行的测试，为余下的测试的构建奠定了基础，也建立了供所有这些由此衍生出来的测试进一步增值的上下文。如果从测试即文档的角度看，它用一种能在控制回归和功能演进方面可以执行的方式表达了软件的目的。

若要完全验证正常路径可能需要多个测试，这取决于测试的范围。在做初始测试时，可以先从验证一两个特征目的[37]（characteristic purpose）开始下手。对于一个单元测试来说，一个测试应该就能够控制它。而对于一个全栈系统测试，或许需要一整套的测试套件。但是如果能按照多个功能范围将该套件进行分解，那么就能更加容易地管理手头的任务。

3.2.2 测试替代路径

一旦确定软件的主要功能能够按照预期进行工作，就能对付那些从正常行为所衍生出来的各种有用的行为变体。例如，如果首要的功能是保存文件，那么那些针对网络文件系统的特殊保存位置或许就是一个不错的替代路径。而在单元测试层面，或许在没有事件排队的情况下确保一个事件处理循环能够正常工作，就是一个替代路径。

在这一点上，测试的覆盖目标会指导测试的全面性。如果用覆盖率来驱动单元测试，那么就会试图进行彻底的测试。这将很有可能在针对全栈系统测试的功能覆盖的意识下指导自己进行测试。而那些子系统的集成测试将会追求更加本地化的完整性定义。

3.2.3 测试错误路径

许多人在开始测试软件的错误处理之前就停了下来。不幸的是，人们对软件质量的很多感知，不是由软件是否失败（因为失败总是会发生的）来形成的，而是通过软件是否能处理这些失败来形成的。这个世界充满了种种意外事件。即使软件运行在单独的、硬件化的计算机上，最

终也会失败。电源波动、电磁干扰和元件故障这些仅仅是那些经常成串发生的众多事情中的一小部分而已。

错误处理验证能够确保那些针对环境中异常变化的响应是刻意设计的，而不是偶然发生的。对错误处理进行刻意的设计，应该能给予用户那些开发人员所期望的体验，但愿这也是用户所期望的体验。

许多组织都会跳过或者吝啬地进行错误路径的测试，因为测试会涉及种种困难。一般来说，相比较小的范围来说，在较大的范围内引入错误会更加困难。相比一个全栈应用来说，在类的级别上模拟各种网络错误会更加容易。如果能从架构的角度关注错误处理，那么就会得到针对各种组件如何组成框架的明确定义的方针，从而勾勒出在较低层次上进行验证的意图，能够令人专注于本地行为的正确性。

3.2.4 测试数据的排列组合

数据[38]驱动几乎所有的软件。在测试用户接口和公共API时，边界条件和验证条件会显著地影响软件的安全性和稳定性。在离编程较近的那些层面上，不同形式的数据控制行为能够产生大量的软件功能。甚至在像Java和C#这样的静态类型的语言中，系统设计方面的更高层次的抽象，会自然而然地降低作为全面测试指南的代码覆盖率的有效性。而动态语言和动态特性（像基于反射执行），则更是加大了上述挑战的难度。

1.边界条件

在软件行为中，一种更加常见的数据变化形式来自于边界条件（boundary condition）。边界条件会由于各种广泛的原因而产生。正常路径和替代路径的测试在正常的输入值中验证行为，但可能不会测试所有的输入值。边界条件测试验证软件在如下条件下该如何做出行为。

在正常输入的边界检查问题，比如“相差为一”[39]（off-by-one）这样的错误。

在非正常输入的边界也检查相差为一的错误。

出于安全的考虑使用预料中的非正常输入的变体。

使用特定的无法正常工作的非正常输入，诸如被零除错误，或使用能触发由上下文确定限制的输入，如数值精度或表示范围。

当进行错误路径的测试时，或许已经测试了一些边界条件。然而，如果能以边界条件的角度审视上述变化，就能让在错误处理逻辑中所发生的遗漏凸显出来，并能驱动出更加彻底的测试覆盖。

自然或实用的数值和资源限制能够产生丰富的边界条件。当使用自然的有限状态集的数值时，就会产生自然的限制。“真/假”和“是/否”是这些状态中最小的单元。那些要求用户从有限数量的选项中进行选择的菜单选项也提供了上下文自然的限制。像字段长度这样的实用的限制形成了边界条件的丰富源泉，特别是在操作或添加要输入到软件内部的数据时。而对于有限资源或范围的极端边界，可以测试像内存和文件大小这样的限制。

数值和数学上的变化形式被认为是自然的或实用的，它们出现广泛，具有特点，使其值得拥有独特的处理方法，并引来程序员们特别的关注。被零除错误也许是编程中最常见的数学问题了，不管其表示格式或大小是怎样的，这个问题都是需要关注的。需要持续考虑由于离散表达方法所导致的数值的限制，因为当把数值迁移到更大的表示范围后，必然会增加要测试的数据量。而精度呈现了一系列更加复杂的待测条件，因为精度问题会影响正在被测的代码和测试代码。

基于标准和基于约定的格式产生了一些结构化的、可预测的、有时还很复杂的模式，特别是随着这些模式的发展，从其中可以派生出许多边界条件。例如，域名系统[40]（Domain Name System,DNS）的语法规则比较简单。然而，即使在这样简单的情况下，也有发现其中令人吃惊的各种变体的机会。出于安全的考虑，人们会试图去验证这些域。而那些选择不去通过查询来验证域的人，不管他们理由是好是坏，都必须假

设域名的规则会超出语法约定。笔者曾经见过有代码假设所有的顶级域名（Top-Level Domain，TLD）的长度必须是2或3个字符，而这一点在过去对于顶级域名集合中的大部分域名都是正确的。但它忽略了原先被分配的用于管理用途的单字母域名，而且也没有自动考虑到那些已经或将被添加的更长的顶级域名，如.name 和.info。另外DNS语法对于允许非欧洲字符集的扩展又为验证增添了另一个麻烦。

更加临时或非结构化的来源为预测提供了一些最富有挑战性的输入。任何自由格式的文本字段都有许多需要验证的因素。最简单的情况或许只涉及限制或剥离空白字符或从一个有限字符集中选择的一些字符。而比较复杂的情况则包括评判输入以检查SQL注入或跨站脚本攻击和对语义内容的自然语言处理。

2.数据驱动的执行

通过代码覆盖来指导测试，特别是在单元的层面，对于测试从代码结构派生出来的行为的变化来说效果很好。然而，许多结构提供了显著的行为变化，在代码中却没有明显的分支。所谓的软件工程基本定理[41]是这样说的：“我们通过引入一个额外的间接层面就能解决任何问题。”

在处理命令行或某些远程调用的接口时，调度器（dispatcher）使用抽象工厂（Abstract Factory）来生成用于运行的命令模式[DP]对象，这就是一个常见的数据驱动场景。如代码清单3-2所示。虽然CommandFactory类的功能和每一个可以被获取到的Command类的实现应该被单独测试，但是CommandDispatcher将一些行为集成起来，创建了一组更大的行为，且这组行为无法通过静态分析或覆盖率的评估识别出来。

代码清单3-2 用数据驱动的方式使用抽象工厂模式来创建命令模式对象以进行工作的调度器

```
class CommandDispatcher {
```

```
    private CommandFactory commandFactory;
    public void dispatch(String commandName) {
      Command command =
        commandFactory.createCommand(commandName);
      command.execute();
    }
}
```

当在单元层面上测试这些结构时，应该验证调度机制的正确性。理想情况下，调度目标的定义应该是动态的或分离的，并以一种有利于进行独立测试的方式呈现出来。应该独立地测试每一个调度目标。

对于更大范围的测试，比如系统测试或集成测试，必须测试每一个动态变化，以确保对软件进行了全面的测试。当一个通常工作在单元层面上的调度机制被集成到一个组件或系统中时，一般都有一个定义明确的和有限的可能性集合。

3.运行时和动态绑定

大多数运行在虚拟机上并且/或者具有像脚本语言那样的动态绑定特性的编程语言，都具有一个被称为反射的特性。反射提供了在运行时检查程序的命名空间的能力，以发现或验证像类、函数、方法、变量、属性、返回类型和参数这样的元素的存在性，并且在合适的情况下调用这些元素。

访问或调用任意符号的能力，类似于一个内置的、基于运行时系统维护的数据的、数据驱动执行的形式，这会比大多数靠自己来实现上述功能的应用，具有更高程度的能力和灵活性。反射的力量，已经导致许多团队在应用程序中阻止或取缔它，以避免一些毫无疑问的令人反感的使用。在像Java（见代码清单3-3）或Perl这样的语言中，这不太会阻碍大部分应用程序的开发。而如果不使用上述特性，那么像 Smalltalk 和JavaScript（见代码清单3-4）这样的编程语言就会遭受麻烦。即使团队

避免编写基于反射的代码，许多框架（像Java　Spring和Quartz）还是大量地使用反射来实现基于配置的应用装配（application　assembly）和依赖注入的功能。

代码清单3-3　用Java编写的使用反射的基本动态调用，省略了错误处理和异常

```
class Invoker {
   public static void invokeVoidMethodNoArgs(String className,String
methodName) {
      Class clazz = Class.forName(className);
      Object object = clazz.newInstance();
      Method method = class.getMethod(methodName,null);
      method.invoke(object,null);
   }
}
```

代码清单3-4 用JavaScript编写的基本动态调用

```
function invokeNoArgsNoReturn(object,func) {
   if (object[func] & & typeof object[func] === “function”) {
      object[func]();
   }
}
```

即使是像C和C++这样的在反射方面功能较弱的编程语言，通过使用像dlopen(3)这样的POSIX动态库的API，也能表现出一些支持反射的编程语言所具有的动态绑定特性，如代码清单3-5所示。该API为应用程序赋予了动态加载共享库、调用其中函数的能力，而这一切是在已知调用签名这个约束下，通过将库和函数名指定为字符串来做到的。

代码清单3-5　不考虑错误处理，在C语言中使用POSIX动态库API的运行时绑定

```
#include ＜dlfcn.h＞
int main(int argc,char **argv)
{
   void *lib;
   void (*func)(void);
   lib = dlopen(argv[0],RTLD_LAZY);
   func = dlsym(lib,argv[1]);
   (*func)();
   dlclose(lib);
   return 0;
}
```

正如在数据驱动执行中所做的那样，测试需要验证动态调用机制在单元层面上也能工作，并且各个部件组装在一起后能够在更高的层面上一同工作。

3.2.5 对缺陷进行测试

无论对代码进行多少遍测试，代码也还是会有缺陷。如果团队的工作完成得好，那么就能在软件投产之前，找出所有显著的缺陷。不管缺陷是在何时被何人发现的，都要编写一个测试来让缺陷重复出现，然后修复缺陷并让该测试运行通过，这样做能帮助我们了解到该缺陷已经得到修复，并且能确保随着时间的推移，该缺陷总是保持着被修复的状态。

笔者比较喜欢测试每一个缺陷，至少在单元层面上笔者会这么做。每一个缺陷，包括那些可以被大致描述为集成或交互问题的缺陷，都能跟踪到一个代码单元的一个或多个缺陷。或许调用者传递了错误的参数，或以错误的顺序调用了功能。或许调用者以错误的格式或数值使用了参数或返回值，从而做错了事情。也许处理同步时所采用的方式允许

了偶尔的竞态条件。所有这些问题以及许多未能尽述的其他问题，都能在单元层面重现并修复。

第4章 设计和可测试性

有关软件设计的书已经很多了。有关设计问题的知识体系和词汇已经演进了几十年。结构化设计、面向对象设计和面向方面设计（aspect-oriented design）都已经应用到了软件中。我们把各种设计模式和模式语言当成编目和指导原则来讲述。Meszaros[xTP]汇编了有关软件测试的各种模式。

而上面这一切，却很少有如何将软件设计成可测试的讨论。可测试性经常被当成是一种事后补充，或者为了设计的纯洁性干脆就牺牲了测试。实际上，许多“良好设计”的原则都阻碍了可测试性。本章将探讨一些会导致这种问题的做法。

4.1 关于设计范型

本章将在面向对象（Object-Oriented，OO）设计的背景下讨论很多内容。在本书撰写之前的20年里，OO就是主流的设计范型（paradigm）。本章所讨论的这些问题并不是OO所独有的。如果说有什么区别的话，那就是，大多数OO编程语言中的各种封装机制可以为本章所讨论的问题提供更广泛的解决方案。另一方面，机制的数量和通常伴随这些机制的传统智慧，也提供了许多让代码难以测试甚至是不可测试的方式。

随着我们这个行业的成熟，对于测试的总体态度正在发生着巨大的改变。现有的持续日益增长的计算能力，使得人们可以很容易地决定持续地运行自动化测试。受精益思想启发而产生的像极限编程这样的方

法，和像测试驱动开发这样的实践，推动着我们首先编写测试来内建质量（build quality in），并驱动设计。社会中无处不在的软件增加了广泛快速测试软件的需求。

然而，我们的设计范型和实现它们的编程语言却没有跟上。在当今世界里，软件比以往任何时候都更重要，软件质量就是商业决策，在这个前提下，我们需要了解我们那些设计方法的种种限制。无论是在过程式的、函数式的框架下工作，还是在面向对象式的框架下工作，都需要将可测试性作为一个设计因素来考虑。

4.2 封装和可观察性

实现隐藏（implementation hiding）是面向对象封装的一项基本原则。一个对象作为一个整体代表了一个事物。它或许拥有特征或属性，但是对其进行表示的底层数据不一定与其外部表示一致。同样，要把类的那些不会直接在外部表示中体现出来的行为的子集隐藏起来，因为它们只是许多可能的实现方法中的一种而已，否则，若将它们隔离起来分别对待的话，会变得没有意义。

4.2.1 表示性的封装

数据类型也不会告诉我们有关表示（representation）的所有事情。尽管有关面向对象设计的大部分讨论内容都集中在结构和语法这些方向上（因为这些是封装的机制），但是可以肯定地说，含义才是其中最重要的方面。当设计并使用一个对象的目的和特征时，需要把它们放到意识的最前沿。理想情况下，对象的设计能引导我们正确地使用它，但是当前各种主流的编程语言对于完整的语义意图的表达方面做得并不多。

至于对象的用法，其所做的一切会使我们避免创建带有不一致的内部状态的对象，并针对保持兼容性这样的增强特性，提供对象的实现途

径。使用对象的方法来控制对属性实现的访问，隐藏了管理内部状态方面的复杂性。如果处置得当，就算内部表示发生变化，使用接口的那部分用户也不用做什么更改。

然而，这个相同层面的间接性却干扰了实现的验证。只有极少数的简单对象才只需要对其规格进行彻底的测试。通过进一步的测试覆盖来驱动测试才能确保发现更加深入的需求。

考虑一个对I/O进行缓冲的对象的例子，类似于Java的BufferedReader[42]类。该类不应该暴露缓冲的内部细节，因为典型用户只是想更加有效地读取一个数据流。注意到BufferedReader仅仅实现了Closeable和Readable这两个接口，这一点更加突出了上述意图。

上述这个用于缓冲I/O的类的实现者，会十分关心这个对象的内部行为。诸如将字符读入缓冲区、缓冲区中字符的数目和当缓冲区被排放或装满时系统有何行为这一切，都决定了功能的正确性。严密地封装这些实现会让这个对象难以测试。为了让其更加容易测试，开发者可以放松一点严密的封装，并用文档注明这些暴露的入口点不应被使用，但是该类的用户们却可以，而且很有可能，在某个时间点上忽视这些警告。不同的编程语言会提供一些其他的可以用来访问类的内部细节的机制，如同第9章所讨论的那样，但是这些机制的大部分所做的充其量就是把一个锁好的箱子强行撬开而已。

4.2.2 行为的封装

当面对各种方法和从属对象时，也要需要考虑这些问题。随着对象行为的不断实现，其各种方法的大小和复杂性也会不断增加，从而导致重构的发生。这种重构会创建一些新的方法和类，这其中会有一些代表了该对象行为的纯粹的内部模型。Extract Method[REF]的重构手法创建了一些受限制的方法。而Extract Class[REF]的重构手法创建了一种新的表示，这种表示可能不需要在原来的上下文之外可见。

像这样的重构能改良软件的规模和复杂性。然而，既然庞大的规模和复杂性会导致重构，那么就肯定也需要进行测试。虽然是通过重构来简化软件的组件或单元，但在这个过程中，我们也能识别出一个过程中的某个关键步骤或一个关系中的某个角色。这些新元素的关键性都需要进行测试。

在面向对象的开发社区中，有很多人说不应该测试私有方法。这种观点当然是支持对象的严格封装的。然而，当我们建立条件，用所有可能的方法调用那些被提取出来的元素时，这种情况也会更有可能增大测试的复杂性和维护的工作量。在测试中所产生的复杂性于是会转变成测试中发生错误的机会，使得测试用例不那么可靠。在测试中对重构后的实现直接测试，和在生产代码中进行重构，一样简单。

4.2.3 测试的灰度

测试是一门色彩单一却灰度各异的学科。黑盒测试、白盒测试和灰盒测试是经常会出现的词汇。黑盒测试的目的是，基于暴露给外界的功能和外部协议对软件进行测试。另一方面，白盒测试则是在充分了解内部情况的基础上对软件进行验证。在色谱中间的某个地方，灰盒测试可以看成是凭已掌握的知识推测内部情况的黑盒测试，也可以看成是在部分了解内部情况的基础上，进行合理的推断或预期的白盒测试。

让测试与代码之间的耦合尽量小的愿望促使了这种有灰度变化的分类的产生。如果仅测试那些对象对外承诺的内容，那么在编写测试时就不会做出没有根据的假设。许多人觉得，当从较细粒度的单元测试向外过渡到粗粒度的系统测试的过程中，应该仅编写黑盒测试。在后面的章节中，将会讨论，黑盒测试还不足以避免把测试耦合到实现上，而且标准的面向对象设计的准则所带来的耦合程度会比通常所意识到的大。

黑盒测试不能充分地测试实现，就像前面那个I/O缓冲的例子所展示的那样。所以为了达到彻底的功能覆盖，无论是否使用代码覆盖率的

目标，我们都得使用白盒测试或灰盒测试来作为补充。然而，针对内部可见的代码的测试，经常需要用到那些封装所阻止的内部访问权限。

测试的维度

当讨论测试时，一些考虑因素会彼此交织。例如，是否曾经听过有人问，一个测试是单元测试还是回归测试（regressiontest）？他们是否真的是相互排斥的？让我们来看看在讨论测试时，会遇到的不同的维度。

第一个维度是目的。为什么要编写或运行这个测试？回归测试可以确保行为不会随着时间的推移而改变（即回归）。当练习测试驱动开发（TDD）时，测试将使用场景进行了文档化，并对底层设计进行指导。随着时间的推移，测试的目的会发生变化。原先为TDD而编写的测试，在TDD阶段的初期很快就变成了回归测试。验收测试从用户的角度验证软件是否符合其功能目的；它们验证软件是否已经准备好被接受。性能测试检查系统是否能用很低的延迟来提供充分的响应，等等。冒烟测试提供了快速但粗略的验证。

粒度是又一个维度。试图测试的范围和边界是什么呢？当谈论单元测试时，术语“单元”（unit）指的是正在测试的代码的最小单元，如函数、方法或类。因为单元测试的定义存在着广泛的争论，所以一些人更喜欢“隔离测试”（isolation test）这个术语，因为它隔离了一段明确定义的代码段。而较大粒度的测试包括执行单元或组件测试组合的集成测试，和验证整个系统的系统测试。

另一个维度是透明性，就如同在4.2.3节中所讨论的那样。黑盒测试仅测试那些显性的行为而不考虑实现。灰盒测试会考虑那些可能的实现和算法陷阱。白盒测试则由实现来对其进行指导。

还有一个维度描述了测试的手段。Monte Carlo 测试[43]、猴子测试[44]（monkey testing）和随机事件测试都指的是这样的技术，即用各种过程很确定的测试所无法做到的方式来“惊扰”（surprising）一下代码。

当谈论测试时，要认识到这些重要的维度。尽管一个维度中的某些点可能经常会与另一个维度的某些点相关（例如验收测试和性能测试通常是系统测试），但是不应该假设典型的相关性总是会起作用的。尤其应该避免调换相关因素的位置，或忘记它们会随着时间的推移和上下文的不同而发生变化。

4.2.4 封装、可观察性和可测试性

总而言之，常规的面向对象的封装指南，会使良好设计和可测试性之间的关系变得紧张。一个类的可测试性直接关系到验证以下方面的能力：该类的状态、隐藏在getter和setter后面的代码，以及隐藏在受限制访问的方法中的该类的行为。

理想情况下，编程语言都会支持可测试性的需要。越来越多地，框架为可测试性提供准备，工具利用编程语言的特性来增加我们的测试能力。而我们要做的则是在良好设计的规则与验证的需要之间找到一个平衡点。

本书提倡尽可能就使用语言和工具的特性来测试代码。当无法使用标准语言特性时，可以研究各种方式，在尽量少地违反封装精神和宗旨的情况下，放宽设计的限制。

4.3 耦合和可测试性

耦合是一段代码依赖另一段代码的程度。显然，基于自然形成的关系，测试代码会依赖其所测试的代码。对于黑盒测试来说，测试代码依赖测试目标的接口，以及测试目标所使用的任何数据类型、其所依赖的各种接口或在测试目标的接口中所使用的签名。如果一个方法在其参数列表中使用了一个类型，那么调用方代码（calling code）（以及测试）需要能够获得或创建该类型。当一个方法返回一个类型时，调用方代码

会使用该值，从而表达出其对该类型的了解程度。甚至当使用接口时，上述讨论也同样有效，因为接口规定了类型。函数式、过程式和动态语言也都有这种效果。那些回调函数、监听器（listener）和函子（functor）等的签名（即那些返回类型和参数列表）都给代码引入了额外的耦合性。

当测试那些不属于公共协议的、访问受限的方法时，如果测试的实现使用的是一般情况下不会暴露的类型，那么白盒或灰盒测试就会创建额外的依赖关系。拥有支持实现的内部的完全私有的抽象的情况其实并不少见。

所有这些测试代码和被测代码之间的关系，都增加了测试与测试目标之间的耦合程度。彻底的测试倾向于最大化两个参与者之间潜在的耦合。

软件设计的目标是尽量减少耦合。耦合度量了两个组件之间的依赖关系。依赖关系可以表示为，针对达成软件的商业目的而对所提供的功能的必要使用。然而，针对任何给定问题将软件分解为组件仅仅是很多可能的方案中的一种而已。代码的初始实现经常需要随着软件的生长和功能的增加而变化。如果在设计接口时有先见之明，而且运气足够好，那么组件就能够被重构，并对所有的调用者保持外部行为和协议。但是对于灰盒或白盒测试来说，就不存在这样的保证，因为重构并不保证实现的细节能够得到保持。在更加极端的情况下，必须重写组件，很可能会令所有的调用代码都无法工作或失效，这包括黑盒测试。

软件产品的演进会强制代码发生变化，特别是对于测试代码。经常有人基于这种效应来评价更低层次的测试。但情况不必是这样的。使测试变得很脆弱的原因，既不是因为测试的存在，也不是因为代码有变化的事实。就像当软件在演进时可以将其内部设计得不那么脆弱一样，也可以在可测试性上应用设计的原则来让测试代码不那么脆弱。这需要更改“良好设计”的理念来将可测试性的特征扩展为首要的代码驱动者。耦

合是对测试实践进行调整的敌人，特别是针对单元测试。

让我们更加细致地看看这个耦合的某些方面。在理想情况下，单元测试是耦合到被测单元[45]的，而且也只耦合到被测单元。然而，除非是限制自己完全使用编程语言的基本数据类型，否则这种情况在实际中很少会发生。像代码清单4-1所示的Java编写的接口，或是代码清单4-2所示的JavaScript编写的接口，经常会指向那些最根本的抽象中的其他结构。

代码清单 4-1 Swing 包的 Java 接口展示了对其他类的使用。注意PropertyChange Listener来自一个独立于Swing的包

```
package javax.swing;
import java.beans.PropertyChangeListener;
public interface Action
    extends java.awt.event.ActionListener {
  java.lang.Object getValue(java.lang.String s);
  void putValue(java.lang.String s,java.lang.Object o);
  void setEnabled(boolean b);
  boolean isEnabled();
  void addPropertyChangeListener(
    PropertyChangeListener propertyChangeListener);
  void removePropertyChangeListener(
    PropertyChangeListener propertyChangeListener);
}
```

代码清单 4-2 jQuery 的 JavaScript 功能接口展示了对事件对象的使用和表示子集。尽管JavaScript是动态类型的，事件对象内容的约定也将调用者耦合到了签名上

```
jQuery.on( events[,selector][,data],
    handler(eventObject) )
```

```
{
   target:
   relatedTarget:
   pageX:
   pageY:
   which:
   metaKey:
}
```

我们忽视了在接口中对类型的使用其实构成了该接口协议的一部分这一点。对于类型的使用会将使用该接口的那些用户耦合到这个附加的类型上，增加系统整体的耦合性。因为要获得良好的代码覆盖率，测试需要最大限度地使用接口的特性，所以这些测试会使用所有的参数和返回类型。通常情况下，测试必须创建参数的实例，调用一些构造器、setter和其他方法，这里每一个创建和调用都能使测试耦合到那些实现和表示上。构建器（builder）能够将这些耦合抽象化，但是它们又引入了对构建器的耦合。测试的工厂方法整合了这些依赖关系，提供了一个避免耦合的薄薄的绝缘层，但是耦合依然存在。

同样，测试需要使用接口中的方法的返回值，来验证操作的结果。在最好的情况下，这些测试能够通过编程语言所固有的等值判断手段（如Java的Object.equals()方法）来验证返回值，这是一个经常被隐藏起来但是相对无害的耦合。更多的时候，测试会验证那些单个的属性或结果，而这些都是在做部分比较的需求驱动下产生的。也许正在被测试的那个方面的这些效果不需要做完整的相等性比较，或者尽管与固有的等值判断手段相关，但就不存在多少有关类的状态的确定性的方面（如GUID[46]或UID），来区分一个个不同的实例。现有的比较器（comparator）对象和方法，能够通过在其所在位置用一个更小的依赖作为替代的方式来将耦合最小化。作为改进，仅用于测试的比较器整合

并缓解了耦合。但并不是所有的验证都完全依赖于状态，而是需要使用在那些在返回接口中出现的次生（secondary）类型之上的操作。

这些概念中的一个更加常见而隐匿的表现形式，是通过最简单的面向对象的约定而产生的：属性封装。通常情况下，开发人员习惯于为他们的每一个属性都编写访问器（accessor）。其原因包括：能将调用者与属性的内部表示相隔离，允许编写虚的或带有计算能力的属性，隐藏了副作用，强制实施了不变性约束，等等。虽然，最常见的情况是，getter 仅仅简单地返回内部表示，但也有可能通过对更复杂的内部结构的引用来提供对其直接的访问。正如我们所看到的，对于这些内部表示的验证有时会通过接口访问来进行。其中null安全性（null safety）或者通过表示来保证，或者通过测试来忽略。让我们来看看下面这行代码：

A.getB().getC().getD()

尽管公然违反最少知识原则[47]（Principle of Least Knowledge），但经常能在测试和生产代码中发现像这样的代码，当然我们自己是不会这样写的！尽管对象A、B和C的状态按照面向对象的标准明确地封装好了，但这种getter链（getter chaining）的结构还是会传递性地将A耦合到B、C甚至D的实现上[48]。

所有这一切都旨在加剧良好的设计原则与实用的开发驱动力之间的紧张关系。通常情况下，我们或许会因为一个迂腐的有关性能损耗的原因，而免掉在内部表示与接口表示之间所进行的转换。本章促使我们考虑测试的调理方案在未来的可维护性会如何影响那些设计决策。

幸运的是，存在解决这些问题的轻量级的方法。通过重构，当栽培（grow）代码和测试时，可以在根据需要的基础上，将这些方法引入进来。这里的关键并不是削弱在编程上的表现来加强代码与测试的彻底隔离，从而让测试不再与其直接测试对象之外的代码耦合。相反，需要刻意地用平衡考虑各种因素的方式来进行设计与实现。

第5章 测试的原则

虽然本书包含了许多用于测试各种特定情况的专门的攻略，但其中的一些技术也可以应用于多种情况。像任何技能一样，精湛的技艺来自于对机制的学习和实践。真正的专业来自于理解机制背后的道理。本章将讨论这些道理。

将这些原则应用到测试的编写过程中，将有助于找出哪些测试模式是相关的，哪些特定的技术是可用的。在更大的系统之中，这些原则将有助于调整测试工作的规模，以匹配系统的成长，不会有那些经常会发生的不成比例的开销。

这些原则是按照重要性的大小来排序的。一般情况下，当要做出有关测试方法的决策时，笔者都是让先给出的原则比后给出的原则拥有更多的权重。尽管如此，一个坚实而明确的理由应该胜过排序先后的差别。最终，最重要的是理由充分的设计决策，而不是规则。

5.1 把测试雕琢好

对于测试代码，我们应该有这样的认识：测试代码几乎和生产代码一样重要。这意味着应该用与生产代码同样的标准来雕琢测试代码。特别是应该做到：

一定要在测试代码中表达出测试的意图；

尽量仅表达意图，而不表达其他的；

保持测试代码的简单与整洁，使用配置几乎相同的同样的工具，并且随时进行重构。

为了使本书保持简单，这里不会重复有关雕琢良好的代码所有其他的那些原则。然而，可以稍有不同地运用一些软件匠艺的原则。

对于测试代码，应该禁用某些静态检查器规则。需要注意的一点是，大多数静态检查器都不知道断言语句会抛出异常，于是它们就不会将其当作一个函数或方法中最末端的语句来对待。或许还会发现，这种情况也是可以接受的：让运行环境告诉我们测试中的一些值是null的，还是undefined的，而不是到处去写断言来判断其有效性。

无论是否做重构，都应该保留四阶段测试（Four-Phase Test）[xTP]的结构——setup （设置）、execute（执行）、verify（验证）和tear down（拆除）。如果读者比较熟悉AAA结构——arrange（布置）、act（行为）和assert（断言）[49]，那么这些就是需要保留的边界。这意味着不应跨过这些阶段的边界来重构。一点点的重复能够更好地传达测试的意图。

下面两点值得进行更加深入的讨论。

5.1.1 将输入关联到输出

曾有多少次看到过类似下面这样的测试？

```
assertEquals(4,transformIt(2));
```

其中字面值[50]（literal value）2和4都是什么意思？当然，假设文档存在的话，能够阅读有关transformIt()方法的文档来了解其要做的事情是什么。而在文档缺失的情况下，可以看该方法的实现，对代码做逆向工程，但如果这样的话，测试的就是实现而不是意图了。而且，上面这段代码的作者需要每一位后来的开发者除了提取被测代码的意图外，还要去破译他自己原本的意图。

你愿意看到代码清单5-1所示的这种代码吗？

代码清单5-1 在测试中将输入关联到输出

```
int operand = 2;
```

```
int expectedSquare = operand * operand;
int actual = transformIt(operand);
assertEquals(expectedSquare,actual);
```

虽然这个例子有点儿简单和做作，但是代码能以一种更加能揭示其意图的方式来供人阅读。或许更重要的是，代码清楚地向将来所有的维护者表达了测试的意图：要测试operand这个输入与expectedSquare这个输出之间的关系。如果transformIt()方法发生了变化，那么该测试先前的预期对于所有参与修改该方法的人来说都是清晰的。

5.1.2 使用命名约定

笔者差点儿就不想写这一节了。毕竟，每个人都知道使用命名约定，对吧？而且每个人都知道其在自文档化代码（self-documenting code）、可维护性、意图表达等方面的好处。

此时笔者想起了曾经见到过的那些代码库，其中的测试，有的使用了无意义的命名，如test1和test2，有的使用了不完整的约定，如testMethod，仅表达出了它是 Method的测试，但没有告诉我们它正在测试该方法的哪个方面，另外还有一些使用了不一致的约定。所以需要在这里讨论一下。

使用一种约定，并坚持使用下去。确保这个约定捕获了所有需要捕获的内容。要认真地考虑使用行业标准的约定，无论是明确提出的还是事实上的标准，因为只有使用了约定，才能将开发准备时间最小化，减少对新员工的再培训。如果约定需要改变，需要确保有如何和何时修改那些使用旧约定的实例的计划。在合理的情况下，可以使用静态检查器来强制代码遵守约定。

对于测试来说，一个良好的命名约定的元素可能包括以下几项。

语法标记，用于区分测试与非测试。常见的例子包括，给类的名字添加Test前缀或后缀，给测试方法添加test 前缀，以及使用元数据标

记，比如JUnit和TestNG里面的@Test标注（annotation）或NUnit里面的[Test]属性。

被测类，诸如使用被测类这样的指向被测符号的引用可以作为基本名字，然后为其添加Test前缀或后缀，并且把被测方法的名字用作测试方法命名的一部分。

条件或变体的描述，用于区分特定的测试，如有意义的值或业务规则的上下文。

分隔符，用于分隔命名中的不同组成部分。

指南，列出在各种具体的非显而易见的情况下该如何应用约定。例如，对于类Class的构造器，其测试命名的起始部分，是应该用testClass呢，还是应该用testConstructor呢？

下面是笔者目前在Java中使用JUnit或TestNG编写测试（见代码清单5-2）时，所采用的的特别约定。

给被测类添加Test后缀来形成测试类名。笔者之所以用它来当后缀，是因为a）这是一种比较常见的约定，b）笔者接触的关于测试系统的大量代码的命名都已经自然而然地以 Test开头，c）这样可以避免和那些带有后缀规则的测试相混淆，如异常测试。

给测试方法添加test前缀。这有一点从JUnit4之前的版本保留下来的风格，但是笔者喜欢它能独立于标注之外，清晰地界定测试方法与非测试方法这一点。

将被测方法的名字用作测试方法的基本名字。

如果包括描述信息，就将“_”（下划线）用作基本的测试名字和测试变体的描述之间的分隔符。

添加测试变体的简短描述。对于独特的和明显的“正常路径”的情况，可以省略这个描述。

使用类的名字来命名构造器测试。

对于属性的那些成对出现的getter和setter，可以用以testSetGet作为

命名开头的方法来一同测试。

代码清单5-2　笔者目前实际使用的Java测试命名约定的简短展示。注意，为简洁起见格式已被压缩

```
class Payment {
   public Payment(Double amount) { ...}
   public void setAmount(Double amount) { ...}
   public Double getAmount() { ...}
   public PaymentTransmissionResult transmit() { ...}
}
class PaymentTest {
   @Test public void testPayment () { ...}
   @Test public void testPayment_ZeroAmount() { ...}
   @Test public void testSetGetAmount() { ...}
   @Test public void testTransmit(){ ...}
}
```

5.2 避免在生产代码内出现测试代码

你见过如代码清单5-3所示的这种代码吗？

代码清单5-3 在生产代码内的测试代码的例子

```
public class MyClass {
   private boolean testing = false;
   public void doSomething() {
      if (testing) {
         // Mock something up
      } else {
         // Run the real code
```

```
        }
      }
    }
```

笔者肯定读者见过。遗憾的是，大多数代码库中都有像上面这样的代码或其变体。让我们从软件验证的角度仔细看看由于有这样的代码而导致的所有问题。

首先，testing是如何被设置的？笔者可能遗漏了一个允许通过接口直接控制这个行为的 setter。那或许是管理它的最安全的方式了。笔者也曾见过有人取而代之地使用一个开关接口（toggle interface）。那有一点不太安全，因为为了正确地使用它，必须得总是检查这个值。有时这两种方法的实现中会缺乏相应的 getter。或许它能通过某种形式的依赖注入来得到控制，在这种情况下，它的值会与实现相分离，这使得代码中明显给它赋值的默认值可能并不是它最终真正的值。另一个方法是在代码中简单地改变它的初始化部分。但如果这样的话，要是开发人员在提交代码前忘记把它改回去该怎么办？如果mock足够好，可能就没人能发现这种情况，直到在部署到生产环境后有不好的事情发生。上面这些可能中的每一种都会带来测试代码在生产环境中被打开运行的风险。

接下来，问问自己我们是否可以测试所有的代码。答案是不能。如果我们打开测试标志，这也是测试时表面上想要做的事情，那么运行的也仅仅是 if 语句的第一个分支。第二个分支仅当不测试时才会执行。这意味着在 else分支中的任何代码在部署之前都得不到测试，从而引入了漏检bug的风险。

最后，如果使用代码覆盖率来驱动测试，那么这种技术会创建一个“暗斑”（dark patch），即一块不可进行测试覆盖的代码。如果不把代码测试覆盖率作为目标，或者制定足够低的覆盖率目标使得else分支中的代码被漏过，那么这或许是可以接受的。笔者曾经见过的大量被这样

处理的代码，都是下面这样对系统行为有显著影响的代码，即系统初始化代码或资源管理代码。除了存在上面提到的未被测试的代码块这样的问题以外，这些测试覆盖率所导致的漏洞会导致另一个问题，即令人感觉测试覆盖总是不完全的，这会削弱覆盖率作为一种指导性指标的价值。

这种技术使用得越多，上述风险全部出现的可能性就越大。其中一种风险体现出来的机会或许很小，但是这种技术的每一次使用都会增加全局风险。如果在足够大的代码库中广泛地使用这些技术，那么全局风险就几乎一定会出现。

基本的原则是测试代码应与其正在测试的代码分开存放。测试钩连代码（testing hook）不应以公然要测试的方式侵入被测代码。针对可测试性做出的一些设计考量应该尽可能多地保持代码的意图。

5.3 通过实现来验证意图

正如在第3章中所讨论的那样，所有的软件都有意图，而且测试的首要目标是验证意图。黑盒测试之所以很流行、很有效，是因为它纯粹通过软件的意图而不是实现来验证软件。然而，大多数中等复杂的算法或者逻辑都无法单纯地用黑盒进行完全测试。以覆盖率为导向的单元测试需要运用白盒的洞察力来测试实现代码。

请注意，如果要寻找有指导性的表述形式，那么这个原则是说“意图高于实现”，而不是说“要意图而不要实现”。当做一些层面的测试，特别是单元测试时，根据实现来做指导是明智的、必要的，但是我们应该永远不要让代码正试图产生的意图走出视线。

5.4 将耦合最小化

反对进行广泛测试的一个主要意见是，测试的维护工作会产生开销，而这个开销的增长比源代码库的增长都迅速。听得更多的可能是这样的话：“维护所有这些测试真是太难了。”

根据笔者经验，这种后果几乎可以通过控制测试的耦合[51]来消除掉。控制耦合的方法根据所讨论的测试类型的不同而有所不同。

当把有关实现的知识用白盒或灰盒的方式纳入到测试中时，系统测试和集成测试中的耦合就通常会发生。有时过分热心的测试者会超越系统必要的行为保证，但通常由于无法验证一个操作的所有效果，让耦合成为一个软件没有针对可测试性进行构建的征兆。

对于单元测试来说，耦合经常会以各种更加隐秘的形式出现。例如，面向对象设计原则建议所有的属性都应该用getter和setter来封装。虽然这种封装会有一些好处，包括逻辑的实现和延迟加载属性的实现，但是在大多数情况下，这种只针对一个值或引用的封装还是非常薄。一个指向复杂内部属性的引用的返回值，经常是为了使用方便而设计的一个实现细节，而不是接口本身的需求。当这种事情发生时，被测类的接口就会被耦合到被内部引用的对象的定义上。单元测试反过来会调用这个接口，造成该测试与内部对象之间发生传递耦合（transitive coupling）。

这些可传递的耦合关系类似于，一个团队的人数增加时团队沟通关系的增加情况。如果一个团队有n个人，且每一个人都与另一个人直接沟通的话，那么会有（n^2-n）/2条一对一的沟通线。如果团队有4个人，那么就只有6条沟通线，这是一个比可管理的数量。如果有10个人，其45条沟通连接线的结果就会引入低效率。20个人会有240条沟通线，这个数字就已经无法管理了。

本书所讨论的这些技术几乎都集中在有关减少测试与其对应的实现之间的耦合[52]的各种方法上。通过减少组件之间的耦合，就可以减少

测试中的耦合。运用这一原则能够让测试工作获得更多的可伸缩性。

5.5 要最小的、新的和瞬态fixture

在《xUnit Test Patterns: Refactoring Test Code》一书中，Meszaros[xTP]建立了一套用于讨论测试的词汇表。特别地，他所定义的有关 fixture[53]的术语澄清并隔离了一些可以分离的问题。本节所讨论的原则正是使用了他所定义的术语。

fixture 建立了测试的上下文。它们是运行测试所必需的 setup 部分。它们用尽可能快速和相互隔离的方式对运行测试所必需的组件进行构造、mock、dummy[54]、注入或其他操作。

最小的fixture应该与所需的一样大，并且不能再大。每一个必要的fixture都拥有能够将其耦合到实现的一些元素。尽管按协议进行测试这一点很好，但是正在测试的是带有真实实现决策的真实的软件。一个fixture与被测组件隔离得越远，它的行为就越像系统的其他部分。这一点会将被测组件耦合到系统的其他部分，从而以传递的方式将测试耦合到系统的其他部分。让fixture保持最小不仅能让测试更快更简单，还能够降低测试与系统其他部分的耦合。

使用新的fixture的测试能够针对每一个测试重新创建它们的fixture。fixture越新，测试之间相互影响的可能性就越小，就会更容易地让每个测试保持相互隔离，从而减少测试之间的耦合。

瞬态fixture仅存在于测试的运行期间。它们也能减少潜在的相互影响和两个测试之间的耦合。

将那些最小的、新的和瞬态的fixture结合在一起，会有助于减少测试中的耦合，并令测试工作具有伸缩性，另外还可以提供快速和隔离的好处。

5.6 利用现有设施

这一点通常被表达为“通过接口测试”。在一般意义上，这两句话是等价的，但是面向对象的编程语言也给出了一个更加狭义的不够充分的解释。这个原则会比简单地使用接口中声明的方法要更加宽泛一些。

当编写测试时，希望将改变被测代码以适应可测试性的程度最小化。为此，最好是使用已经存在的特性。这对于构造器、setter和getter来说，可能都是显而易见的，并且对于方法的参数、返回值和其他直接的功能来说，也是完全自然的和必要的。经常能见到代码的那些能够用作潜在的钩连代码来支持可测试性的其他特性被忽略了。日志记录、监测模块、工厂、依赖注入、回调和模板方法都提供了载体来针对被测代码执行更多的控制。为了封装复杂性和维护有意图的设计，在创建新设施之前，应该使用现有设施。

5.7 要完整的验证而不要部分的验证

许多被高度封装的系统都隐藏了内部状态，因为它们不希望让其客户端代码来操作。如果这种状态是瞬态的或针对具体实现的，那么这或许是正确的决策。另外开发人员有时也会出于安全的考虑隐藏这些属性。安全为隐藏造了一个令人信服的理由，但是如果以验证困难作为代价的话，那么这样做还是应该的吗？只有getter的访问操作和只提供被克隆的返回值是两种提供属性可视性的方式，而且不会削弱很多安全动机，允许更加完整的验证。

另一种常见的不完整的验证类别是用来验证结果的子字符串、集合成员和其他子集合技术的使用。包含性（containment）是一种比较弱的验证形式。针对完整验证的技术能够提供更加强壮的测试。在后面的章节中会讨论这些技术。

5.8 编写小测试

小测试更容易让人理解，它们都被限定在自己的范围之内，主要被用来测试行为的某一个方面，不大可能产生耦合。当小测试运行失败时更容易诊断。还需要说更多吗？

5.9 分离关注点

在设计和编写软件时，我们中的很多人都能想起分离关注点，但在测试时就会忘记。在测试中一个常见的复杂性来源就是没有分离关注点。

例如，假设要创建一个日期转换库。该库中的两个方法，一个将日期字符串解析为底层的表示，另一个将底层表示格式化为可以显示的字符串。可以编写测试来验证这样一个来回的结果，如代码清单5-4所示。

代码清单5-4　用JavaScript和Jasmine[55]通过一个来回的转换来测试日期解析和格式化功能

```
describe('DateLib',function() {
  it('converts the date back into itself',function() {
    var expectedDate = '17 Apr 2008 10:00 +1000';
    var actualDate = DateLib.format(
      DateLib.parse(expectedDate));
    expect(actualDate).toBe(expectedDate);
  });
});
```

然而，上面实际测试的是功能的两个独立的部分：解析和格式化。仅仅由于可以将它们放到一起进行测试这一点，并不意味着应该这样

做。取而代之的是，像代码清单5-5所示的那样对它们分别进行测试。

代码清单5-5 针对代码清单5-4以分离关注点的方式测试日期解析和格式化功能

```
describe('DateLib',function() {
  it('parses a date properly',function() {
    var expectedTimestamp = 1208390400000;
    var actualDate = DateLib.parse('17 Apr 2008 10:00 +1000');
    expect(actualDate.getTime()).toBe(expectedTimestamp);
  });
  it('formats a timestamp properly for GMT',function() {
    var expectedDateString = 'Thu,17 Apr 2008 00:00:00 GMT';
    var inputDate = new DateLib.Date(1208390400000);
    expect(inputDate.toGMTString()).toBe(expectedDateString);
  });
});
```

通过对功能进行分别测试，能够独立地验证每一个操作，并且找出一个bug的问题恰好弥补了另一个bug所带来的问题的情况。另外也更加直接地表达了每一块功能应该如何被使用的方式。

5.10 使用唯一值

当用于构造那些 fixture 的值各自都很独特时测试的可读性和实用性就会得到增强。在测试中，那些独特的值能够加强各种断言之间的差异，并且有助于突出实现中被错误地传入的参数或属性这些问题。在那些数据驱动测试[56]（data-driven test）的两个迭代之间，独特的值有助于更加清晰地突出失败的测试用例。同样的效果也适用于测试之间，在这种情况下，独特的值也有助于识别被共享的状态所造成的影响。

断言失败时，通常能很好地反馈所涉及的值和在代码中的位置。然而，一些断言，比如 assertTrue，并不会显示它们正在验证的表达式的值。循环中的那些断言所对应的多个数据值会共享一个代码位置。自定义的断言和在fixture或工具方法中出现的断言，所显示的栈跟踪信息（stack trace）的层次比通常期望的深。在上述的这些情况中，独特的值都有助于更加快速地定位故障。

数据驱动测试放大了循环中断言的共享位置属性。整个测试会重复执行，而不是仅仅在一个循环中执行几行代码。在行中，独特的值有助于区分异常的属性。无论哪个断言运行失败，列中的那些独特的值都能起到隔离测试用例的作用。

尽管最佳实践指导我们远离那些相互之间有影响的测试，但是各种共享资源，从静态变量到单例（singleton）再到文件或数据库，都会引起交叉，特别是当测试以并行的方式运行时。当一个值出现在错误的测试或上下文中时，独特的值能让我们更加容易地注意到这个问题。

5.11 保持简单：删除代码

可以通过删除代码来简化测试，减少测试负担。如果代码不存在，就不必对它进行测试或维护了。删除那些未被使用的、被注释掉的、做了超出所需事情的和复杂得超过需要的代码。删除代码是一个降低成本的好方法。

此外，删除冗余的测试。更理想的做法是，从一开始就不编写它们。额外的测试不但帮不上验证的忙，反倒会花费运行时间，消耗维护的注意力。试想一下，为了试图在同一地方找出冗余测试与其他测试之间的区别，会浪费多少时间。

如果使用代码覆盖率来指导测试，那么删除代码也能提高测试覆盖率。通常认为通过增加测试的数量就可以增加覆盖率。但是覆盖率是一

个百分比数值，它是用执行特性[57]（feature）的数量除以存在特性的数量得到的。有两个自由度来控制覆盖率，包括减少特性的总数。在有冗余测试的情况下，覆盖率不会增加，但也不会减少。

5.12 不要测试框架

通过利用他人的工作成果而不是复制他们的工作，才更有可能获得成功。虽然编写自己的宠物软件项目[58]（pet project）可能会很好玩，但是下面这种情况的可能性会很大，即具备更多专业知识或经验的其他人已经把这个项目编写好了，并且成百或成千的人正在使用它。或许它能作为所选编程语言的标准特性或库的一部分来获取到。

框架、库、插件、代码生成器和与它们关系紧密的软件，越来越多地让我们工作在一个越来越高的抽象层面。无论软件是内置的、商业的，还是开源的，我们都能将其纳入开发的生态系统并相信它们能正常工作。

商业软件会自带技术支持和维护（不管它在授权许可中的免责声明说了什么），以及一些符合用户目的的期望。通常我们的系统库也同样如此，这些库经常会有后端设计，而且在它们的设计与规格中还有一个标准体系的支持。开源软件会自带用户社区，而且在他们的构建和安装中经常会有测试套件。

一般情况下，不需要测试第三方代码。笔者曾经不计其数地看到开发人员，包括笔者在内，很快就跳到这样的结论去，即编译器或函数库有 bug，因为这不可能是我们的错误，要在过了很久之后才会意识到应该先看看自己的代码。当然，有时错误是由第三方代码的可用性或文档质量的缺陷造成的。曾经有段时间，bug 真的就在第三方的代码中。然而，大部分问题都出在我们这里。

别人已经完成了保证软件质量的工作，我们就不要再重复他们的工

作了。

不要测试生成的代码

请注意，生成的代码也属于不要测试的这一类。生成源代码的好处在过去的几十年里已经被其自身所证明了。早期的编译器就是汇编代码生成器。从生成的代码的语义和语法，产生了许多编译器。XML解析器、正则表达式生成器，还有CORBA、SOAP和REST接口在许多实现中都使用代码生成。向领域特定语言（DSL）和更高层面的构造发展的趋势是经常使用生成器，这一点至少在它们的早期形态中是这样。那么，为何不测试这些代码呢？

首先，代码生成通常源自坚实的理论或实验基础。那些生成器是用实验上或算法上正确的方法来解决问题的，虽然这并不能保证给出的实现就是正确的，但它为获得更可靠的解决方案提供了一个坚实的基础。

其次，生成的代码经常会去解决一些问题，即代码作者或行业尚未给出通用实现的那些问题。这意味着生成的代码通常是重复的，从而测试起来很乏味。

最后，笔者发现很多生成的代码会忽视许多良好的设计原则，因此很难对其进行测试。很多基于理论的生成器来自于过程式的开发遗产，对于我们在当前编程范型中所看到的对使用小函数的痴迷，这些遗产并没有表现出同样水平的喜好。而且这些代码根本就不是为人类的维护工作而设计的。

测试是否已经使用生成器正确地达到了系统目标。如果正在使用一个编译器的生成器，那么要确保生成的编译器能够接受试图要实现的那种语言。对于远程API，可以通过生成的stub来测试API的业务目的。

5.13 有时测试框架

每一个规则都会有例外。在讨论框架时，我们讨论了诸如信任、拥

有测试、保证和正确性这些事情。有时，那些条件不是真实的。而其他时候，也不敢假定它们是真实的。让我们看看都什么时候需要测试框架。

设想你是一个团队的一份子，正在参与开发一个大型的软件系统，这也许是一个由数百万行代码所组成的产品，动用了几百个开发人员，并且在整个代码库中使用了数百个第三方软件包。在两次发布之间的任何一个给定的时间段内，可能都会有几十个第三方软件包发生变化，而且可能需要更新。如何才能安全地更新软件包，而不会破坏系统呢？可以依赖那些已经编写的针对自己使用部分软件包的测试，但是在其他情况下，特别是在选择把框架的功能（如一个服务的访问库）进行stub或mock的情况下，或者在被测系统参与了测试范围之外的事情（例如像一个依赖注入框架可能做的事情）的情况下，那些测试可能就不足以维护新版本的质量。

一个没有测试代码的开源软件包，特别是那种很新、很少得到维护或者带有不兼容历史的软件包，可能也不会符合那些不用验证就会采纳的信任标准。为框架编写自己的测试可能是确保符合需要的最谨慎的做法。甚至可以考虑把自己编写的测试贡献给该项目。

那些项目所依赖的高风险的框架大概也应该在正确性方面得到更多的重视。一个被普遍使用的简单框架会影响系统的所有部分，因此被全面使用的框架需要该框架的所有部分都能按照预期来运作。

最后，测试第三方代码不可避免地要做这样的风险评估，即对它进行测试的成本与冒着出故障的风险来对其进行复用的好处相比是否值得。仔细考虑，并明智地做出判断。

---

[1].本书所讨论的精益是指从精益制造（Lean Manufacturing）运动派生而来的精益生产（Lean Production）方式，这有别于精益创业（Lean Startup）。但如果精益创业就是公司的经营方针的话，当从精益创业的试验中体悟到其中的运作之道后，就会需要本章所讨论的这些适用于可

持续产品化的软件的质量观点。

[2].编程牛人（code-slinger）指的是那些编起程序来像牛仔一样技艺精湛却狂放不羁、不走常规的编程高手。sling作动词表示“随意地投掷”，slinger则表示那种像牛仔一样拿出绳索甩出去套动物的人。（感谢正在美国Google公司工作的译者的大学同学姚芳对该词的解释。）——译者注

[3].http://codemash.org

[4].http://globalday.coderetreat.org

[5].即前面提到的软件跻身于更加传统的工程学科的行列的运动。——译者注

[6].关于这一点，可以找到许多有关精益、丰田生产方式和浪费的参考资料。可以参考http://en.wikipedia.org/wiki/Toyota_Production_System。

[7].机会成本，是指为了得到某种东西而所要放弃另一些东西的最大价值，也可以理解为在面临多方案择一决策时，被舍弃的选项中的最高价值者是本次决策的机会成本（引自百度百科）。——译者注

[8].感谢Zee Spencer在这一方面的建议。

[9].来自极限编程的“You aren’t going to need it”（你将来不会需要它）。[XPI，第190页]

[10].或者被选用的其他编辑器。一些文本编辑器，如Sublime Text（www.sublimetext.com），在一个更加轻量的安装包中，支持许多IDE风格的功能。

[11].空白字符的类型一般包括空格和Tab字符。——译者注

[12].PMD是一个针对Java和其他编程语言的静态分析工具，它的官方网站是http://pmd.sourceforge.net。该工具的开发人员声称，PMD不是一个缩写，而是一个“反向缩写”（“backronym”，即先有PMD这个缩写，然后再找出3个英文单词来凑出这个缩写。——译者注）。关于PMD的解释和这3个字母所可能表示的意思，参见http://pmd.sourceforge.net/meaning.html。

[13].领域特定语言是一种专为特定应用领域而设计的计算机语言。引自wikipedia.org。——译者注

[14].即代码中存在的更深层次问题的症状，这是Kent Beck对软件匠艺所做出的另一个贡献。

[15].一个测试的代码一般分为三个部分：（1）测试前的各种条件准备，如实例化待测的类等；（2）调用待测对象的某个方法，以触发待测的代码行为；（3）验证上述代码行为是否符合预期。这三个部分在本书中分别称为setup、execution和verification，又称准备（arrange）、行动（act）和断言（assert），或假如（given）、当（when）和那么（then）。——译者注

[16].NaN是IEEE浮点规范中“非数字”的符号表示。

[17].mock对象指的是取代被测系统所依赖的那个真实组件的对象，使得测试能够验证其间接输出。引自Meszaros所著《xUnit 测试模式》（xUnit Test Patterns）一书。——译者注

[18].分支覆盖率评估那些语法流程控制语句中的每一个选项，看它们是否都被覆盖到了。条件覆盖率着眼于，在复杂的条件下，发生短路真值估值（short-circuit evaluation）之后，那个完整有效的真值表是否被执行了。

[19].definition-use、define-use或def-use链是一些从变量定义到该变量使用之间的路径，在这些路径上该变量未被重新定义。与此相反的有关变量使用的分析另见use-define链（http://en.wikipedia.org/wiki/Use-define_chain），即从一个变量的使用到该变量的定义之间的所有路径，在这些路径上该变量未被重新定义。对于这两个指标，路径的集合是相同的，但分组是基于相反的终点的。相比大多数工具的实现来说，这些路径的覆盖率是覆盖率中比较严格的形式。

[20].见http://pmd.sourceforge.net/pmd-4.2.5/rules/controversial.html#DataflowAnomalyAnalysis。

[21].这些里程碑纯属传闻，但与其他人的意见具有很好的相关性。包括http://brett.maytom.net/2010/08/08/unrealistic-100-code-coverage-with-unit-

tests/和许多组织内部常见的70%～80%的覆盖率目标。对于上述效果，还有一些可能的解释，这些解释涵盖了从测试的事实到像TDD这样更加注重设计的实践等各个方面。

[22].实现指的是完成接口功能的实现代码。——译者注

[23].重要的是要考虑到，上述意图会发生在软件开发过程的许多层面上。开发人员将用户所想要的，转化为前者所认为后者所想要的，哪些功能应该被添加以解决后者的需要，如何将这些特性映射到现有或期望的应用上，如何将这些纳入系统的架构和设计之中，以及如何编写代码。本章所讨论的许多观点，都可以外推到其他层面上。

[24].若想深入了解这个话题，可以参考Wiley出版社1995年出版的Alan Cooper的著作《交互设计精髓》（About Face）及其后续版本。

[25].在Java语言中，final关键字本身就表明了，由它所声明的变量一旦初始化，其值就不能再改变了。

[26].SUT是Software Under Test（被测软件）的缩写。[xTP]

[27].验证两个对象的深度相等性，即验证这两个对象内部所保存的各个数据的值都一一相等。——译者注

[28].敏捷测试象限图的第1象限表示支持团队且面向技术的自动化测试，包括单元测试和组件测试；第2象限表示支持团队且面向业务的自动化或手工测试，包括功能性测试等；第3象限表示评价产品且面向业务的手工测试，包括探索性测试、可用性测试、用户验收测试、Alpha/Beta测试等；第4象限表示评价产品且面向技术的依赖工具的测试，包括性能和压力测试、安全性测试和非功能性测试等。参见http://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/。——译者注

[29].Meszaros建议使用术语“测试替身”来涵盖诸如stub、fake和mock等这些用来支持测试的不同的说法。[xTP]

[30].ORM即Object-Relational Mapping，指用于在面向对象编程的对象模型与在数据库系统中的关系模型之间进行转换的软件系统。——译者注

[31].特征测试即重现系统目前的行为的测试。——译者注

[32].虽然术语“接缝”的字面意思非常直观，但确切地说，Michael Feathers所定义的接缝是指“程序中的那些能够改变行为而无需在该处进行编辑的位置”（第31页）。

[33].协作者即代码库所依赖的其他组件。——译者注

[34].参见Michael Feathers 有关这个主题的博客：www.artima.com/weblogs/viewpost.jsp?thread=126923。另见[WEwLC]。

[35].测试用的stub指的是取代被测系统所依赖的那个真实组件的对象，使得测试能够控制被测系统的间接的输入。它允许该测试强制被测系统进入到或许在其他情况下无法执行的那些路径。引自Meszaros所著的《xUnit测试模式》一书。——译者注

[36].Michael Feathers[WEwLC] 将“遗留代码”定义为任何没有测试的代码。

[37].特征目的即软件既有行为所表达的目的。——译者注

[38].仅讨论本书测试中用到的那些数值。数据库测试不在本书的讨论范围之内。

[39].相差为一的错误，即涉及边界条件的离散等价问题的逻辑错误。该错误经常发生于计算机编程中，比如一个循环多循环了一次或少循环了一次这种情况。引自wikipedia.org。——译者注

[40].参见http://tools.ietf.org/html/rfc1035#section-2.3.1。

[41].并不真的是一个定理。有关这句话的出处存在着争论。参见下面两条出处：http://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering和 http://en.wikipedia.org/wiki/David_Wheeler_%28computer_scientist%29。

[42].http://docs.oracle.com/javase/6/docs/api/java/io/BufferedReader.html。

[43].可参考Monte Carlo方法，即依赖重复进行的随机取样的算法，来获

得数值结果。为了获取一个概率情况未知的实体的分布，通常会反复多次运行模拟系统。引自wikipedia.org。——译者注

[44].猴子测试指的是运行时没有具体给出测试内容的单元测试。其中猴子指的是任意输入值的提供者。引自wikipedia.org。——译者注

[45].本节将使用术语“单元”，因为无论是测试对象、组件、模块、函数，还是一些其他的作为元素的构建模块，也无论是使用哪种编程范型，这里所讨论的概念都是适用的。

[46].GUID（Globally Unique Identifier）是在计算机软件中用作标识符的一个唯一的参考号码（引自wikipedia.org）。——译者注

[47].又称迪米特法则（Law of Demeter）。最少知识原则为在软件中将耦合最小化方面，提供了有益的启发：
http://en.wikipedia.org/wiki/Law_of_Demeter。

[48].注意getter链与调用链（call chaining）有着显著的不同。在后者中，每一个调用返回一个指向下一步将要对其进行调用的表示（representation）的引用。这一点经常作为下述函数式编程风格的实现方式，即紧凑地对状态进行设置。

[49].由Bill Wake于2001年提出，也被称作AAA或3A。参见
http://xp123.com/articles/3a-arrange-act-assert/。

[50].字面值指的是计算机科学中的一种符号，用来表示源代码中的一个固定值。——译者注

[51].Edward Yourdon和Larry Constantine将耦合的概念引入到了计算机科学词汇之中，这个概念是结构设计[SD]的一部分。耦合描述了软件系统中两个模块之间的依赖程度。

[52].耦合可以被测量。其中一种方法参见
http://en.wikipedia.org/wiki/Coupling_%28computer_programming%29#Module_coupling。

[53].不同的TDD和ATDD框架对fixture的定义有所不同。一般按照Meszaros在《xUnit测试模式》一书中对其下的定义来理解：一切准备到

位的用来执行被测试系统的东西，如新创建的被测对象等。fixture原意指住所中像浴缸和马桶这样无法在搬家中被带走的东西。在测试中指每个测试中被创建的那些相互独立的测试准备条件。勉强可译为“固定设施”。——译者注

[54].dummy对象指的是以参数或参数的属性的形式被传递给被测系统，但是从未被实际使用的占位对象（placeholder object）。引自Meszaros所著《xUnit Test Patterns: Refactoring Test Code》一书。——译者注

[55].Jasmine是一个针对JavaScript的开源测试框架。——译者注

[56].数据驱动测试是计算机软件测试的一个术语，用来描述这样被完成的测试，即直接把一张条件表用作测试输入和可验证的测试输出，同时测试的过程中那些测试环境的设置和控制都不是硬编码的。引自wikipedia.org。——译者注

[57].在这里使用“特性”这个术语，是因为覆盖率并不仅仅与代码的行数或语句相关。一套完整的测试调理方案可能也要考虑分支、条件、循环迭代、def-use链或其他可度量的代码特性。

[58].宠物软件项目指的是开发人员像养宠物那样自己写一些感兴趣的软件项目。——译者注

第二部分 测试与可测试性模式

有了理念和原则，现在是深入匠艺的精髓——代码的时候了。第二部分的开始，先来建立用于构建测试的一致性的根本基础。接下来会使用各种编程语言的例子来讨论逐渐复杂的主题。这一部分用如何测试并发性的一个讨论来结束。

第6章 基础知识

有关测试的最难的事情之一是有时候所知道的仅仅是从哪里开始。本章重点讨论这一点。从一个类开始，看看如何能提供一个坚实的基础来测试该类余下的功能。

然后本章会深入探讨一些基本的工具，这些工具能被直接使用，并且经常被用在比较复杂的测试场景中。我们会研究一些结构化的方法来暴露那些被隐藏的值，并接触一些将会在后面章节中展开讨论的话题。

在讨论过程中，我们会看到，通过使用测试框架特性，能得到一些可用的变体。我们还采用了一些来自Meszaros的《xUnit测试模式》[xTP]一书中的测试模式。

6.1 bootstrapping构造器

在测试一个封装良好的类的时候，有一个基本的难题，这个难题是如此基本以至于经常被忽视：怎么知道已经正确地进行了测试？没有类的属性的访问权，那怎么验证构造器正确地完成了它的工作？

当然，可以用构造器最基本的影响作为起点，并验证一个有效的对象被构造好了。在这个过程中，同时也验证了没有抛出异常。但是，怎样验证内部状态被正确地初始化了呢（见代码清单6-1）？

代码清单6-1 测试对象的构造和缺乏异常

```
public void testMyObject() {
    MyObject sut = new MyObject();
    assertNotNull(sut);
```

```
}
```

如果不坚持严格的封装，那么就可以直接查询属性。然而，在面向对象灌输课程的第一天，我们就都学到了这是一件坏事。我们学到唯一允许访问那些属性的方法应该是使用getter方法。

但是别急，是否真的应该使用那些未被测试的方法来验证另一个方法的行为？还有其他选择吗？简而言之，可能吧，这得取决于语言，但是在这样一个根本的、一致的基础上，没有一个选择是用起来足够好的（见代码清单6-2）。

代码清单6-2 使用getter来验证构造器的状态

```
public void testMyObject() {
    MyObject sut = new MyObject();
    assertNotNull(sut);
    assertEquals(expectedValue,sut.getAttribute());
}
```

在继续往下讨论之前，让我们解决这个看似是循环依赖（circular dependency）的问题。解开这个显而易见的难题的关键，是要认识到在正常情况下，构造器和简单的setter和 getter 以与预期一致的方式来做出行为就可以了。实际上，我们会说：“如果看起来像是一个类，而且做出的行为也像一个类，那么就可以放心地把它当作一个类。”

哪些情况不属于前面所说的“正常情况”呢？遇到这些偏离正常情况的时候应该怎样做？当构造器初始化隐藏的状态，或getter做了简单地将值传递给一个属性之外的事情时，异常情况就会发生。在前一种情况下，可以使用一些技术来赋予这些测试访问那些类的特权，这一点在第9章会讨论。在后一种情况下，建议这样的方法应该被重命名，以揭示其更加复杂的特性和不应被伪装成getter这一点。

一个观察测试的数学视角

对于那些有数学倾向的人来说，这里有一个有关测试的比喻，能够

让两种传统的方法与数学证明很相似。我们都熟悉传统的演绎证明。假如有某个由一些基本事实或假设所组成的集合，就可以构建一个完善的逻辑论证，从而推导出结论。只要初始的假设是有效的，并且逻辑论证保持可靠，就可以认为该结论是真实的。系统测试或集成测试就像演绎证明。假如有一个软件初始条件的集合，就可以应用一个对软件进行操作的集合，如果结果状态符合预期的话，声明软件是正确的。实际上，那些行为驱动开发（behavior-driven development）框架中的“假如/当/那么”（given/when/then）这种格式[1]强化了上述解读。这些测试通常会单独存在。总体来说，它们验证了系统的不同方面，从而合在一起，在理想情况下，通过覆盖系统的所有方面，来验证整个系统。

而单元测试则不同。单元测试可以被看作是被测单元的一个演绎证明，但即使是这样也需要做一点点推理。刚刚在前面描述的bootstrapping构造器的测试方法，就可以被看作是一个归纳证明的基础命题（basis case）。归纳证明首先给出了基础命题的前提，然后又给出了归纳的步骤：对于任何命题n成立的前提，对于n+1 的命题也成立。每下一步都是建立在上一步的基础之上的。在针对一个单元的那些测试中，这个单元中比较复杂的那些行为的正确性，依赖于构造和基本属性管理的正确性，如图6-1所示。在更大的上下文中，那些仅依赖于系统库的叶子组件的单元测试就是基础命题。这种启发式地利用之前单元测试结果的方法，仅对被测单元所添加的价值进行测试，从而每一个单元的成功测试，都会提供一个归纳的步骤，来证明系统的正确性。

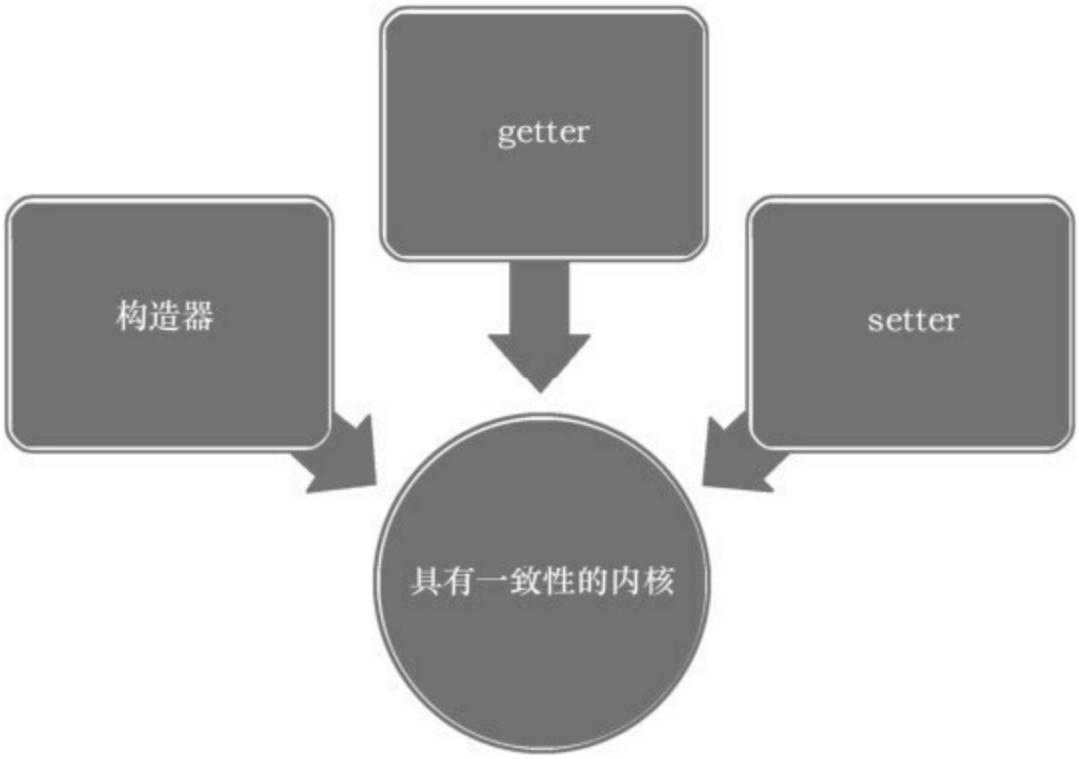

图6-1 一个构造器引导作为一个归纳的基础模型

在数学上，上面两种形式的证明在功能上是等效的，并且都是支持结论的正确方式。然而，软件系统并不是在数学意义上很严格的系统。这个比喻可以作为一个有用的模型，来区分测试的不同方法，但是没有一种方法能够仅凭自己就能足以验证系统。

6.2 测试简单的getter和setter

bootstrapping难题也同样会在测试setter和getter的时候出现，但是在前一节中介绍的策略给了我们一个解决方案。如果不调用 getter，属性的封装会使我们无法验证修改器[2]（mutator）的操作。如果不使用setter，也会妨碍我们通过getter来验证对属性值的设置。因此，只能使用getter来测试setter，反之亦然。有效地将它们一起进行测试的代码如代码清单6-3所示。

代码清单6-3 一起测试setter和getter

```
1 public void testSetGetAttribute() {
2 MyObject sut = new MyObject();
3 assertNotNull(sut);
4 int expectedValue = 17;
5 assertThat(sut.getAttribute(),
        is(not(equalTo(expectedValue))));
6 sut.setAttribute(expectedValue);
7 int actualValue = sut.getAttribute();
8 assertEquals(expectedValue,actualValue);
9 }
```

首先，请注意，代码中的空行将测试分割成了四阶段测试[xTP]（也称作arrange-act-assert 模式）所描述的几部分。尽可能明确地划定这几个阶段有助于提高测试的清晰性和可读性。甚至对在技术上不是必要的中间变量actualValue的引入，也有助于划分阶段并增强对意图的理解。

缺少的第4阶段是Garbage-Collected Teardown[xTP]。强调这一点是因为，即使运行时的系统能够为我们完成拆除工作 ，也需要对这个工作有意识。有垃圾回收功能的编程语言的盛行，已经通过消除大量显式内存管理的开销，显著地简化了代码。然而，与其相应的析构器（destructor）的去除，已经将那些范围边界上的非内存资源的管理工作（如文件句柄和线程）变得很复杂。在测试中，这会涉及那些做清理工作的任务，在测试结尾需要被调用，以便做那些不能施加到测试套件中的所有测试的事情。

在上面第 3 行和第 5 行代码中对断言的使用，强化了测试的前提条件。JUnit提供了一个称作 assumption（假定）的变体，但它仅让测试失效，以好像测试被忽略的方式来处理假定。虽然在测试的前提条件未得

到满足时，assumption不会运行测试，但是它也并没有非常强烈地突出这一点。所以取而代之的还是用断言来达到这个目的。

第5行通过使用Hamcrest匹配器[3]（matcher），让Java语言能体现出更加易读和达意的形式。请注意，这条断言语句读起来是多么的自然。而在许多xUnit变体中，如果在不用assertNotEquals()方法的前提下想做到这样易读的表达，是非常困难的。

在测试的execution阶段，上述代码先后调用了setter和getter，这解释了为何起testSetGetAttribute()这样的测试方法名，甚至名字中两个动词的顺序也与实际的代码行为相一致。这个测试最终以verification阶段中的一个标准相等性断言结束。

有了这两个测试——构造器的验证和setter与getter的验证，就已经“证明”被测单元的基本功能基础是坚实的、一致的。这就为将来构建该类余下的验证提供了基础。

6.3 共享常量

当构造器设置了独立于getter 和 setter的状态时，其中一个挑战就来了。一个典型的例子是，当构造器使用默认值时这种情况会发生。而另一个常见的例子是，当有多个构造器时，如果简单的构造器调用了复杂的构造器，这种情况也会发生（见代码清单6-4）。

代码清单6-4 在构造器的实现中使用字面默认值

```
public class FixedThreadPool {
   private final int poolSize;
   public FixedThreadPool(int poolSize) {
      this.poolSize = poolSize;
   }
   public FixedThreadPool() {
```

```
        this(10);
    }
    public int getPoolSize() {
        return poolSize;
    }
}
```

测试第一个构造器轻而易举。可以传进一个线程池的大小，然后用getter来对其进行验证。不过不能使用getter和setter的组合来进行测试，因为poolSize属性一旦设置就不能改变。但是如何测试上面的默认构造器能够正确地完成它的工作呢？初步进行尝试的测试可能看起来像代码清单6-5所示。

代码清单6-5 测试一个内部使用默认值的构造器

```
@Test
public void testFixedThreadPool() {
    FixedThreadPool sut = new FixedThreadPool();
    int actualPoolSize = sut.getPoolSize();
    assertEquals(10,actualPoolSize);
}
```

这当然可以完成要做的任务。但是若默认的线程池大小发生了改变那该怎么办？就像在前面这个例子中看到的那样，把测试的输入关联到期望的值上能让我们在可维护性上获得好处[4]。这一点对于“被隐藏”的体现了默认值的输入来说也是如此。幸运的是，能够很容易地重构代码，让那些默认值可见，做到自文档化描述。可以看看原来的生产代码经过重构后的片段，如代码清单6-6所示。

代码清单6-6 把默认值提取到常量中的重构后的代码

```
public class FixedThreadPool {
    public static final int DEFAULT_POOL_SIZE = 10;
```

```
    ...
    public FixedThreadPool() {
      this(DEFAULT_POOL_SIZE);
    }
    public int getPoolSize() {
      return poolSize;
    }
  }
```

DEFAULT_POOL_SIZE常量的声明和使用，为我们提供了可公开访问的、用于默认构造器值的能揭示其意图的命名。当测试写成代码清单6-7的样子时，就会变得更加清晰可读，而且也不那么脆弱。

代码清单6-7 使用默认值常量来测试构造器

```
@Test
public void testFixedThreadPool() {
  FixedThreadPool sut = new FixedThreadPool();
  int actualPoolSize = sut.getPoolSize();
  assertEquals(FixedThreadPool.DEFAULT_POOL_SIZE,
    actualPoolSize);
}
```

同样的技术不仅可以用在构造器中，还可以用来公开字符串常量甚至对象常量的访问权限。任何能够使软件工作的固定值或“魔法数字”（magic number）都能成为上述技术的候选使用对象。经常能在定义良好的库中看到这样的，仅仅基于它们的便利性和可读性的目的而被定义的一些常量，但它们也有助于增进软件的可测试性。这是一个很难被滥用的技术[5]。

有时常量值的形式会有所不同。当有多个值来表示一个固定的集合时，枚举就会经常被使用。代码清单6-8展示了一种C++中常见的使用嵌

套匿名枚举的用法。

代码清单6-8　在C++中针对一些紧密相关的常量而使用嵌套匿名枚举

```
class 3DPoint {
   enum {
      X_INDEX = 0,
      Y_INDEX,
      Z_INDEX
   };
   ...
};
```

默认情况下，enum的第1个成员会被自动赋值为0，但是指定初始值也是一种常见的用法。后继的值都会在其前一个值的基础上增1。

6.4 在局部重新定义

一些编程语言允许在一个作用域之内并且仅在该作用域的持续期间重新定义任何变量或函数。例如，比方说想要测试一个使用open函数的Perl子程序。请想想怎样才有可能在下面地道的Perl代码行中，强制让测试中的那个错误发生。

```
open $fh,“＜”,“input.txt”
      or die “Could not open input file.”;
```

open函数成功时返回一个非零值，否则返回一个Perl的未定义的值。open成功时，or关键字的短路求值会将其估值为true，因而会跳过die语句。此时如何才能让open返回undef呢？

代码清单6-9中的测试展示了如何在测试的运行期间并仅在该测试的作用域内，使用 local关键字来覆写（override）open。Perl 有块级作

用域，所以任何在块内对open的引用都会调用代码中的那个仅返回undef的匿名子程序。

代码清单6-9 在测试中使用Perl的local来重新定义open函数

```
sub test_override_open {
   local *open = sub { return undef; };
   # Invoke the code that uses open
   ...
} # local definition of open goes away
```

Perl允许这样处理变量和函数。不管将要被重新定义的函数和变量的作用域是怎样的，对这两者的重新定义都是能够进行的。例如，下面这行代码是用来关闭Archive::Extract模块的警告的，但是只对含该行代码的作用域中这行代码之后的代码起作用。

```
local $Archive::Extract::WARN = 0;
```

这种本地对符号重新定义的功能非常强大，但是仅有少数编程语言有这种功能。

6.5 暂时替换

一些编程语言提供了另一种方法，在不触发代码所调用的其他代码的情况下，测试前面的代码所做的事情。上一节展示了一种在代码块的活动期间在本地重新定义函数的技术。但是如果用面向对象的方式来使用模块的话，那么该函数所对应的方法将会针对下面所有的实例被重新定义，即包含重新定义的函数的模块创建出来的那些实例。

相比之下，具有动态特性和原型继承（prototypal inheritance）特点的JavaScript，允许独立定制一个对象的每一个实例。可以先创建一个对象的一个实例，然后在不修改它的任何其他实例的方法的行为的情况下，替换该实例的方法的实现（见代码清单6-10）。

代码清单6-10 替换一个JavaScript对象的toString()方法的简单例子

```
var testObject = new Number();
testObject.toString = function() {
   return " I’m a test object. " ;
};
```

这种技术可以非常强大。Jasmine测试框架[6]有一个spyOn()方法，可用于捕获和验证一个方法被调用的情况、调用的次数以及每次调用的参数，如代码清单6-11所示。spyOn()方法将原有的方法进行包装（wrap）并放在一个Spy对象中，该对象拦截了方法的调用，并在可选择性地调用原有方法之前记录了这些调用的特性。拆除（teardown）阶段能自动将方法恢复成原来的实现。

代码清单6-11　Jasmine的spyOn()方法使用对象重定义在调用者和被调者之间插入一个有记录功能的代理

```
describe('Spy example',function() {
   it('calls toString',function() {
      var testObject = new Number();
      spyOn(testObject,'toString');
      var result = testObject.toString();
      expect(testObject.toString).toHaveBeenCalled();
   });
});
```

6.6 封装和覆写

在身份验证、缓存维护和其他一些方面中，一个常见的操作是让生命周期超出某个阈值的对象过期。代码清单6-12展示了一个生命周期为两周的身份验证令牌的典型实现。

代码清单6-12 让生命周期为两周的身份验证令牌过期的典型实现

```
public class AuthenticationManager {
   public static final EXPIRATION_DAYS = 14;
   ...
   // Return indicates if the token was expired
   public boolean expire(AuthenticationToken token) {
      Calendar expirationHorizon = Calendar.getInstance();
      expirationHorizon.add(Calendar.DAY,-EXPIRATION_DAYS);
      Calendar createdDate = token.getCreatedDate();
      if (expirationHorizon.after(createdDate)) {
         authorizedTokens.remove(token);
         return true;
      }
      return false;
   }
}
```

上述方法的一个好的测试套件会包含有下面这些日期的测试，即在过期边界之前最近的、之上的和之后最近的日期。让我们专门来看看在过期边界之前最近的日期的测试（见代码清单6-13）。

代码清单6-13 针对代码清单6-12的过期边界测试

```
public void testExpire_ImmediatelyBefore() {
   Calendar barelyValid = Calendar.getInstance();
   barelyValid.add(Calendar.DAY,
      -AuthenticationManager.EXPIRATION_DAYS);
   barelyValid.add(Calendar.MILLISECOND,1);
   AuthenticationToken token = new AuthenticationToken();
   token.setCreatedDate(barelyValid);
```

```
    AuthenticationManager sut = new AuthenticationManager();
    Boolean expired = sut.expire(token);
    assertFalse(expired);
}
```

这看起来很合理，对吧？如果有人说上面这段代码导致了竞态条件，使得在相对较少的测试时间中该测试会运行失败，那该怎么办？能看到这种情况吗？在某些方面，这是下面描述的有关“碰巧相等”（coincidental equality）的一个微妙的变体。触发测试运行失败的事件序列如下所示。

1.初始化barelyValid。

2.创建fixture的余下部分时，用了1毫秒或几毫秒的时间。

3.调用 expire()方法，在此期间初始化 expirationHorizon，其值会比barelyValid初始化的值还晚。

4.与调整后的令牌创建日期的值进行比较，expirationHorizon会让该令牌过期。

5.expire()方法返回true，让测试运行失败。

碰巧相等是基于下面这样的假设，即在测试中的 barelyValid 的初始化与实现中的expirationHorizon的初始化这两者之间，“现在”的定义不会发生变化。 “甚至在两个相邻的语句之间，时间也不会有意义地改变”——这个假设出现得太频繁了，一般可以通过在测试中使用Extract Variable[REF]这种重构方法来修复。然而，在上面的代码中，其中一个Calendar对象的初始化在测试的作用域之外，这需要在实现代码中做一些重构。

让我们应用Extract Method[REF]这种重构方法，把expirationHorizon的初始化替换为一个对下述方法的调用（见代码清单6-14）。

代码清单6-14 为了增强可测试性对期限的初始化进行重构

```
protected Calendar computeNow() {
```

```
    return Calendar.getInstance();
  }
```

现在能在测试类中创建并使用嵌套类来重写我们的测试了，不用再使用真正的AuthenticationManager了，如代码清单6-15所示。

代码清单6-15 重写代码清单6-13的测试来使用重构后的代码

```
  private class TimeFrozenAuthenticationManager
      extends AuthenticationManager {
    private final Calendar now;
    public TimeFrozenAuthenticationManager(Calendar now) {
      this.now = now;
    }
    @Override
    protected Calendar computeNow() {
      return now.clone();
    }
  };
  public void testExpire_ImmediatelyBefore() {
    Calendar now = Calendar.getInstance();
    Calendar barelyValid = now.clone();
    barelyValid.add(Calendar.DAY,
      -AuthenticationManager.EXPIRATION_DAYS);
    barelyValid.add(Calendar.MILLISECOND,1);
    AuthenticationToken token = new AuthenticationToken();
    token.setCreatedDate(barelyValid);
    AuthenticationManager sut =
      new TimeFrozenAuthenticationManager(now);
    Boolean expired = sut.expire(token);assertFalse(expired);
```

}

通过封装原本无法进行覆写的系统类，引入了将过期算法在时间上进行冻结的能力，使其更容易测试。再加上一些文档或基于标注的基础设施，就能建议或强制computeNow()[7]方法只应该被测试所调用。我们以少量的对内部状态的限制访问换取了高度的可测试性。这种封装和覆写的方法在单元测试中是最常见的工具之一。而通常情况下，封装已经是实现的一部分了。

6.7 调整可见性

许多我们已经了解到的有关软件设计的封装的内容都没有考虑可测试性。我们被鼓励让很多东西成为私有的，通过方法来提供访问，并且将行为用额外的间接层次封装起来。虽然大部分启发、设计和模式都提出了最佳实践，并且提供了有用的抽象，但是很少给出对它们进行全面测试的手段和机制。

就拿在一个类中对过程进行简单的分解来作为例子进行说明。将一个接口方法分解为富有逻辑的和具有目的的一些子方法，会涉及根据封装的准则来创建私有方法。然而，如果这些子方法就如同它们本应成为的富有逻辑和具有目的的话，那么它们也会是进行直接测试的极好的候选者，来简化调用这些子方法的整个该接口方法的验证。

直接测试需要在类之外的可见性（visibility），或者至少需要打破类的封装的那些技术。在像Java这样的编程语言中，在安全设置的允许下，通过反射是能够打破封装的。在一些更加动态的语言中，弱封装是一个语言特性，或者就像在Perl中一样，封装更多地是一种事后的补充，而且能更加容易地被打破。但无论如何，即使是最简单的打破封装的技术，也比直接调用更加丑陋和繁琐。

在后面的章节中将研究用来调整可见性的技术。现在，让我们看一

个例子。在这个例子中会简单地放松理想封装的强度。考虑一个在 Java 中常规的数据导出的方法（如代码清单6-16所示）。

代码清单6-16 一个假想的数据导出方法

```
public DataExport exportData(Search parameters,Format cues) {
   RawData raw = retrieveData(parameters);
   PreparedData prepared = prepareData(raw);
   return formatData(prepared,cues);
}
private RawData retrieveData(Search parameters) {
   ...
}
private PreparedData prepareData(RawData raw) {
   ...
}
private DataExport formatData(PreparedData prepared,
   Format cues) {
   ...
}
```

exportData()方法读起来十分整洁，它以逻辑发展的顺序解释了自己的工作流。每一个子方法在封装时，都对客户端代码隐藏了实现细节。这样一来，同样也是客户端代码的那些测试代码，就必须执行大量的fixture的设置工作，来测试导出功能的所有这三个步骤。

对于不熟悉Java的人可能需要介绍一下，Java有包（package）的概念。包所提供的不仅有代码的组织结构和命名空间，还有共享同一个包的那些类。相比那些位于不同的包中的那些类来说，共享一个包的那些类相互之间会有一定的访问优势。具体地说，不带访问修饰符（即不带public、private或protected）的方法[8]具有“包默认”（package　default）

的可见性。在同一个包中的任何类都能够访问和调用该方法。protected修饰符的一个经常被忽视的特性是，它也能被包中的其他类访问。

在前面的例子中，要么把那三个子方法的 private 访问修饰符去掉，要么把它们都改成 protected[9]的，这两种做法都能够让该包中的其他类来访问这些方法。使用protected也能有助于应用前面讨论的封装和覆写策略。在Java和许多其他的编程语言中，同样的包会被用于不同的物理源文件树，这使得下面的事情变得很容易——实际上这也是标准的做法，即将测试进行隔离，但为了便于访问又把它们放到同样的包中。

这个简单的变化将类的封装削弱了一点，这主要是针对其他与之密切相关的类来说的，但会极大地增加该类的可测试性。笔者更喜欢把这些针对常规做法的调整，看作是支持提升测试质量的务实的工程改变。或许下一代编程语言就能将测试内置进去了，那时我们就不必做这些妥协了。

6.8 通过注入验证

系统中除了最外层的叶子组件，所有组件都依赖其他组件来完成它们的任务。这些其他组件称为协作者，在Meszaros的术语表[xTP]中称为Depended-On Components （DOC，被依赖的组件）。不管是编写那些只测试新增价值的单元测试，还是测试一个集成好的系统，经常都需要创建称为测试替身的协作者的替代品。特别是在单元的层面，注入是引入测试替身的最常见的方式。

一般来说，依赖注入是一种软件设计模式，在这种模式中，为了满足所需的功能，会在运行时插入（即注入）协作者。在进行测试时，我们可以利用各种形式的依赖注入来插入mock或stub，或者干脆切断一个重量级的、耗时的或带来不便的依赖。

让我们看一个简单的使用依赖注入进行测试的例子。我们将在第

12　章中研究有关依赖注入的各种技术。试想我们正在编写一个用来管理计算服务器集群的系统。其中部分的分配算法依赖于服务器的CPU负载。用于确定一个远程的CPU是否还在分配阈值之内的代码可能类似于代码清单6-17。

代码清单6-17 用于确定一个远程CPU是否在某一阈值之内的代码

```
class CPUManager {
   private final HostConnection connection;
   private int warningThreshold; // Percent
   public CPUManager(HostConnection connection,
      int warningThreshold) {
   this.connection = connection;
   this.warningThreshold = warningThreshold;
   }
   public boolean isHostUnderThreshold()
      throws RemoteException {
   // May throw RemoteException
   int load = connection.getCPULoad();
   return load ＜ warningThreshold;
   }
}
```

很明显，该类所拥有的方法比上面列出的要多（至少会有一个setter和两个getter，可能还有更多的功能），但是上面这些已经足以用来说明基本的依赖注入了。HostConnection.getCPULoad()可能会抛出RemoteException 异常——这个声明是在告诉Java开发人员，它封装了一个网络连接。如果想对这个类进行单元测试，先不提它要连接网络另一头的机器这个假设，和试图控制 CPU 负载这件事，单就这个网络连接就十分不便。

如果完全放弃网络连接，并接管产生 getCPULoad()方法的返回值的控制权，那会怎样？毕竟，这不是isHostUnderThreshold()方法所增添的价值的一部分，它仅仅是一个数据源而已。

可以创建自己的HostConnection版本，测试所有的阈值变化，如代码清单6-18所示。

代码清单6-18 注入各种阈值变化的自定义的HostConnection

```
public class HostConnectionStub implements HostConnection {
   private final int cpuLoad;
   public HostConnectionStub(int cpuLoad) {
      this.cpuLoad = cpuLoad;
   }
   @Override
   public int getCPULoad() {
      return cpuLoad;
   }
}
```

虽然对于单独一个变化来说，创建一个测试stub类可能有些大材小用，但是可以想象一下测试下面这些 CPU 的负载，即低于、等于和高于阈值的负载，还有针对 0负载的测试，以及在每个CPU有多个核的情况下，潜在的对超过100%的值进行的测试。对于这些测试，这是一个有用的类。让我们把它注入到一个测试中吧（见代码清单6-19）。

代码清单6-19 使用一个自定义的实现来注入数据值

```
public void testIsHostUnderThreshold_Over() {
   int threshold = 50;
   int load = threshold + 20;
   HostConnection connection = new HostConnectionStub(load);
   CPUManager sut = new CPUManager(connection,threshold);
```

```
        boolean result = sut.isHostUnderThreshold();
        assertFalse(result);
    }
```

通过几行额外的代码，我们已经构造了一个满足功能需求的stub，并避免了网络连接和对外部硬件的依赖。通过将这些依赖提供给stub的构造器，我们已经将该stub注入到了被测软件中，简化了针对数据变化的整个测试类别。我们将在后面第9章和第12章展示注入错误和其他行为的方式。

第7章 字符串处理

字符串遍布于计算机编程中。无论是拼接名称，还是生成或解析XML文档，对于世上大部分数据来说，字符串都是一种基础的表示形式。这就给我们留下了在测试中验证字符串数据的任务。

本章将考察几种在测试中验证字符串的方式。先从最简单、最脆弱的方法开始，然后一点点地增大方法的复杂性来增加验证的强度。最后，会得到一个能带来严格验证的方法，同时又能富有弹性地应对代码的变化，还能对字符本地化策略提供友好的支持。

7.1 通过包含关系来验证

让我们考虑代码清单7-1所示的代码。

代码清单7-1 一个简单的字符串组合函数

```
public String describeAddition(int left,int right) {
   return " The sum of " +left + " and "  + right
      +  " is " + (left + right);
}
```

虽然上面这个例子有些微不足道，而且有点做作，但是它还是很能代表字符串通常被处理的方式的。如何测试这样的方法呢？有关这个方法我们知道些什么呢？虽然知道了一些文本片段，但在每次运行时那些都不应该变化。我们最关心的或许是能产生输出的那些正确的值。可以编写一个像代码清单7-2那样的测试。

代码清单7-2 针对代码清单7-1的代码的测试

```
public void testDescribeAddition() {
    int left = 3;
    int right = 5;
    int expectedSum = left + right;
    String expectedResult = " The sum of " +left
      + " and " +right
      + " is " +expectedSum;
    String actualResult = describeAddition(left,right);
    assertEquals(expectedSum,actualResult);
}
```

理想的测试不会因细微的变化而中断。如果我们打算把起始的字符串片段改为“The sum of the numbers”，那会怎么样？我们将不得不修改两个地方，修改的维护成本会增加一倍。如果这个方法有多个测试，如测试负数、零值或特定的总和，那么维护成本就会增加数倍。

笔者喜欢把这种测试中的字符串与生产代码中的字符串相同的情况称为“碰巧相等”或“碰巧相似”（coincidental similarity）。我们都知道这些值在构造时都是一样的。然而，我们应用复制粘贴复用（copy-paste reuse）的方式，增加了维护负担和测试的脆弱性。必须有一个更好的方式来解决这个问题。

让我们将我们的相等性断言修改得宽松一些，让它不那么脆弱（见代码清单7-3）。

代码清单7-3 不那么脆弱了的代码清单7-2中的测试断言

```
assertTrue(actualResult.contains(left));
assertTrue(actualResult.contains(right));
assertTrue(actualResult.contains(expectedSum));
```

现在能够抵御来自起始字符串（或者与此有关的任何字符串字面值）的变化了，但是也削弱了验证的严密性。我们只验证了部分结果。

我们以验证的强度作为交换，获得了可维护性上的改善。

此外，如果其中一个操作数是0，那会怎么样？期望的总和会与其中一个操作数相同。或者如果两个操作数相等，或者结果总和里面有一个子字符串与其中一个操作数相同，比如9加1，那会怎么样？结果字符串10与操作数1在一起，会带来一个有二义性的匹配。在这些情况下，我们不知道断言是否验证了正确之处，还需要进行不可靠值的测试。

这种技术可能会有用，肯定比什么都没有好。当使用它时，应该明白它的局限性，明智地使用它。当结果足够复杂，使得一个匹配能带给我们较大可能的正确性时，使用这个技术就特别合适。如果是这样的话，就应该当心不要过度地规定正确性，例如测试一个字符串的包含关系，其中空白字符无关紧要，但是我们所期望的字符串有一个特定的布局。让我们看看另一种方法。

7.2 通过模式[10]来验证

仅仅是为了体现语言的多样性，并且因为正则表达式会更容易一些，让我们用Perl语言重写上面的例子。为了展示一些能应用于面向对象的上下文之外的技术，我们也编写了过程化的Perl（见代码清单7-4）。

代码清单7-4 使用Perl重新编写的代码清单7-1

```
package pattern;
sub describeAddition {
    my $left = shift;
    my $right = shift;
    my $sum = $left + $right;
    return "The sum of $left and $right is $sum";
}
```

代码清单7-5显示了原来的精确相等性测试在使用了Test::Unit之后的样子，Test::Unit是xUnit家族中Perl版本。

代码清单7-5 使用Perl的Test::Unit重写了代码清单7-2的代码

```
use Test::Unit::Procedural;
use pattern;
sub test_describeAddition {
   my $left = 13;
   my $right = 17;
   my $expectedSum = $left + $right;
   $result = pattern::describeAddition($left,$right);
   assert($result eq
      "The sum of $left and $right is $expectedSum");
}
create_suite();
run_suite();
```

这个测试也有前面提到的那个问题。我们对数字拼接成的字符串做了硬编码的“假设”。这样一来，该测试对于在拼接字符串时发生的变化将不再具有弹性。如何才能用一个不错的Perl语言的方式来解决这个问题呢？

Perl语言的一个最强大的功能是能够很容易地使用正则表达式。让我们来发挥这种优势。首先，需要以能被表示为正则表达式的方式来考虑我们对期望的结果有多少了解，而这独立于那些用于拼接的字符串。给定了已知的输出后，就能期望会有第一个操作数、第二个操作数和总和这样的顺序，另外在它们三者之间还会有一些字符。此外，总和会出现在结果字符串的结尾处。这可以让我们将原先测试中的断言替换为下面这行代码：

```
assert($result =~ /$left.*$right.*$expectedSum$/);
```

对于不熟悉Perl语言正则表达式操作符的人可能需要介绍一下，=~操作符会产生一个Boolean结果，指示其左侧的字符串是否能匹配其右侧的正则表达式。在上面这行代码中，我们断言了$result与给定的正则表达式[11]相匹配。

最终的结果是我们对下面的事情进行了验证，即我们期望的值以所期望的相互之间的顺序出现在结果字符串中。比起前一节所介绍的包含关系，这个测试更加强壮。现在我们以下面的方式验证了所有这三个值的存在，即将这三者相互区分，而不会有零、重复的值或无意中嵌套子字符串产生假阳性[12]（false positive）。我们或许还能通过添加空格来将其验证能力增强一些，尽管这样会在正则表达式中添加一些元素，而这些元素未必是测试的重要方面。

然而，还是有一些测试的维护性方面的问题。这个测试在面对拼接字符串的变化时或许不那么脆弱，但是参数顺序的变化会使其不堪一击。我们已经解决了“偶然相等”（accidental equality）的问题，但还是有一个“偶然相似”的问题。如果把上面那个句式改成“You get 8 when you add 3 and 5”会发生什么？或者如果将代码国际化并在自然语序会发生变化的语言环境下执行测试，那么会发生什么？在本章的最后，将给出这些问题的解决方案。

7.3 通过值来精确验证

在进一步讨论之前，让我们把一个来自第6章的技术运用到当前处理字符串的主题上。其中一个基本的技术是共享常量。在前面的例子中，展示了如何使用下面的方式针对一个默认值来使用常量，即在阐明代码意图的同时使其更容易测试。不怕明说，同样的技术也非常适用于字符串验证——而不仅是适用于默认值验证。

让我们看一个Web控制器的例子，由于兼容性的原因，需要把URL

参数从较旧版本转换到较新版本。为了简便起见，假设只需转换一个参数，并且假设在类中有一个处理参数转换的方法（见代码清单7-6）。

代码清单7-6 在Web控制器中的参数转换

```
public class Controller {
   ...
   public String translateParameterName(String name) {
      if ("oldname".equals(name)) {
         return "newname";
      }
      return name;
   }
}
```

这是一个简单的转换方法，其输出完全由输入决定，是一个理想的可测试方法。它的两个测试如代码清单7-7所示。

代码清单7-7 针对代码清单7-6中的代码的两个测试

```
@Test
public void testTranslateParameterName_NewParam() {
   String expectedParam = "notoldname";
   Controller sut = new Controller();
   String actualParam =
      Sut.translateParameterName(expectedParam);
   assertEquals(expectedParam,actualParam);
}
@Test
public void testTranslateParameterName_OldParam() {
   String inputParam = "oldname";
   Controller sut = new Controller();
```

```
        String actualParam = sut.translateParameterName(inputParam);
        assertEquals("newname",actualParam);
    }
```

虽然这些测试揭示了意图，但是它们不是很能适应变化。一方面，第一个测试假定 expectedParam 不是那个要被转换的值。可以通过一个断言或假设来让这个假定更加明显，但是要将 expectedParam 和谁来进行比较呢？另一方面，第二个测试方法深受碰巧相等之苦。输入值和期望的参数值都是字面字符串，它们都（希望如此！）碰巧与实现类中的值发生关联。

通过快速地做一点重构，就能够改善translateParameterName()的可测试性，如代码清单7-8所示。

代码清单7-8 重构代码清单7-6以获得更好的可测试性

```
public class Controller {
    public static final String OLD_PARAMETER_NAME = "oldname";
    public static final String NEW_PARAMETER_NAME = "newname";
    ...
    public String translateParameterName(String name) {
        if (OLD_PARAMETER_NAME.equals(name)) {
            return NEW_PARAMETER_NAME;
        }
        return name;
    }
}
```

这很简单，对吧？两个简单的Extract Constant[REF]重构给我们带来了显著提高的可测试性。从任何意义上讲，所讨论的这两个字符串根本就不是私有值。因为当它们被包含在访问请求的URL中时，就已经最大可能地将自己公开了。将其变成类的公共常量不会打破封装。现在可以

重写测试来利用新常量了（见代码清单7-9）。

代码清单7-9 重构代码清单7-7的测试以利用目前我们在可测试性方面所做的改善

```
@Test
public void testTranslateParameterName_NewParam() {
   String expectedParam = "notoldname";
   assertThat(expectedParam,
      not(equalTo(Controller.OLD_PARAMETER_NAME)));
   Controller sut = new Controller();
   String actualParam =
      sut.translateParameterName(expectedParam);
   assertThat(actualParam,equalTo(expectedParam));
}
@Test
public void testTranslateParameterName_OldParam() {
   String inputParam = Controller.OLD_PARAMETER_NAME;
   Controller sut = new Controller();
   String actualParam = sut.translateParameterName(inputParam);
   assertThat(actualParam,
      equalTo(Controller.NEW_PARAMETER_NAME));
}
```

通过加入一个前提守护（precondition guard）断言[13]，并且将三处字面字符串改为常量引用，我们就新增了验证一个测试的前提的能力，同时两个测试也有了抵御正被使用的值发生变化的能力。现在让我们看看这个技术还能被发挥到何种程度。

7.4 使用格式化的结果来精确验证

回到前面那个例子上来，从带有方法describeAddition()的代码清单7-1开始，甚至包括那个Perl的例子，都给我们留下了完整性、顺序和本地化的问题。虽然与开始所遇到的问题相比，这些都是不太严重的问题，但是日益增长的全球市场和快速的发展步伐使得存在这些问题还是不够理想。

回顾一下考虑过的方法。包含关系比较薄弱，而且容易导致谎报成功。虽然正则表达式提供了多一些的强壮性，但还是有完整性和顺序的问题。我们可以使用常量代替字面字符串，但如果仅将其运用到结果的拼接片段上，那么还是解决不了前面提到的那些突出的问题。

但是如果考虑一个能够代表整个结果的常量那会怎样？就像前一节中的那个URL参数，结果信息的文本不是私有的，所采用的方式也不是把参数插入到文本中。如果让代表整个结果的字符串带有能保存要插入的值的占位符会怎样？这种技术至少从C语言库定义printf的格式字符串的时候就已经存在了。如果像代码清单7-10那样重构方法来利用格式字符串会怎样？

代码清单7-10 重构代码清单7-1来使用格式字符串

```
public static final String ADDITION_FORMAT
        = "The sum of %d and %d is %d";
public String describeAddition(int left,int right) {
   return String.format(ADDITION_FORMAT,
          left,right,left + right);
}
```

我们尚未解决顺序的问题，但是已经解决了前面包含关系和正则表达式这两者的完整性和假阳性的弱点。测试现在看起来像代码清单7-11所示的那样。

代码清单7-11 针对代码清单7-10的基于格式的测试

```
public void testDescribeAddition() {
```

```
    int left = 3;
    int right = 5;
    String expectedResult = String.format(ADDITION_FORMAT,
      left,right,left + right);
    String actualResult = describeAddition(left,right);
    assertEquals(expectedResult,actualResult);
}
```

但对于排序问题该如何处理？一些语言和库，特别是那些专为国际化设计的，已经扩展了C风格的printf格式化，包含了位置引用。Java就有这样的位置引用。让我们像代码清单7-12所展示的那样修改原有的实现代码。

代码清单7-12 修改代码清单7-10中的实现来解决参数顺序问题

```
public static final String ADDITION_FORMAT
  = " The sum of %1$d and %2$d is %3$d " ;
public String describeAddition(int left,int right) {
  return String.format(ADDITION_FORMAT,
      left,right,left + right);
}
```

上面在格式字符串中所添加的后跟美元符号的数字表示了顺序，它是参数列表中的参数从1开始的索引。

从这种形式开始，就可以做一个很小的步骤来将格式字符串提取到一个本地化的资源文件中，并根据常量去国际化框架中查找相应的键值。这些相对简单的变化一旦完成，我们就能得到一个在任何语言环境下都能对代码进行测试的测试。

当面对代码库所必然要发生的演进时，在实现中做出很小的变化，就能使测试的可维护性和稳定性发生很大的变化。我们已经从子字符串包含这样薄弱的验证中走了出来，到达了完备的和能抵抗变化的验证，

甚至都能够在多语言代码库[14]中工作。

注意事项

本节所使用的简单例子和测试的重点是字符串测试技术。然而，所有的这些有时会与在生产代码中所发现的内容很类似。这些也表明了在简单实现中所发现的危险，即测试有关期望状态的计算可能会与生产代码过于相似。

从其本身来看，这未必是个问题。但是这是一个代码腐臭。在理想情况下，应该通过使用与实现并不相同的算法来计算期望的结果。这是财会领域中的复式记账法在软件测试领域中的等价物。如果能通过两个不同的途径得到同样的答案，那么对于答案是正确的这一点就能有较高的保证。

上述危险和代码腐臭的根源是下面这些诱惑，即只是把实现复制到测试中，或者带着看完实现后所产生的过于强烈的偏见来编写测试。这会造成去验证实现而不是意图这样最极端的情况。

在我们的字符串格式化的解决方案中，可能不存在另一种能够合理实现同样结果的方法。从头开始编写一个字符串格式化器，或者寻找和使用一个不同的格式化库，都是矫枉过正。然而，如果某方法的使用次数多于一次或两次，那么这种情况还是比较复杂的，不鼓励做Inline Method[REF]的重构。

最后，需要带着意图来测试。如果能使用黑盒方法来编写测试，那就用黑盒。如果不能，那么就问问自己是否还有一个合理的替代方式来计算预期的答案。如果写出来的fixture的setup步骤看起来很像生产代码，那么就请一位伙伴来看一下，以确保这些细节是正确的。

第8章 封装和覆写变化

第6章介绍了一个封装功能的技术，其工作方式是将功能进行覆写，使得可以在测试中获得更好的控制。本章将详细讨论这种方法和它的几种变体。使用封装在代码中创建接缝，然后利用它来进行验证。另外也能看到能够帮助我们的测试替身（参阅提示内容“测试替身与基于接口的设计”）的使用。

8.1 数据注入

最简单的覆写封装形式允许将简单的数据注入到被测软件中。不管将系统中的参与者（actor）建模为对象的程度有多大，属性仍然是基本的构建模块。让我们使用带有基本数据类型的封装和覆写来开始本节的讨论，如代码清单8-1所示。

代码清单8-1 针对基本数据类型注入的被测代码

```
public class TemperatureWatcher {
    public static final double REALLY_COLD_F = 0.0;
    public static final double REALLY_HOT_F = 100.0;
    public static final String REALLY_COLD_RESPONSE =
        “Really cold!”;
    public static final String REALLY_HOT_RESPONSE =
        “Really hot!”;
    public static final String NORMAL_RESPONSE = “Okay”;
private double readThermometerF() {
```

```
        // Read the thermometer
        return temperature;
        }
        public String describeTemperature() {
            double temperature = readThermometerF();
            if (temperature <= REALLY_COLD_F) {
                return REALLY_COLD_RESPONSE;
            }
            if (temperature >= REALLY_HOT_F) {
                return REALLY_HOT_RESPONSE;
            }
            return NORMAL_RESPONSE;
        }
    }
```

这个例子演示了与外部世界的一个交互行为，这是使用封装来支持可测试性的一个绝佳机会。与物理世界的交互包括各种各样的相互作用，比如检查 CPU 负载或内存使用率、得到语音或键盘输入、通过网络接收数据、从相机获取图像，或者读取像温度计或联合启动器（joint actuator）这样的外部传感器上的数据。

因为封装已经完成，现在可以调整readThermometerF()的可见性，并通过在测试中将其覆写来把它用作测试接缝。针对这个例子我们的做法会有些不同。从输入到输出的映射能很容易地用一对值来规定，并且我们想尝试某个范围内的一些值。可以为每个数据对编写一个测试方法，但这样会掩盖这样的事实，即实际上针对不同的数据执行了同样的测试。可以编写一个带有一个循环的测试，遍历不同的数据用例。但是这个循环真的不是测试的一部分，而只是我们创建出来的用来补充测试框架的一个机制。与以往不同，这次让我们使用TestNG[15]和其所支持

的数据驱动测试（有时也称为表驱动测试）的功能（见代码清单8-2）。

代码清单8-2 针对代码清单8-1的使用覆写注入的测试

```
public class TemperatureWatcherTest {
   @Test(dataProvider = " escribeTemperatureData " )
   public void testDescribeTemperature(
         double testTemp,String result) {
      TemperatureWatcher sut =
            new SoftTemperatureWatcher(testTemp);
      String actualResult = sut.describeTemperature();
      assertEquals(actualResult,result);
   }
   private Object[][] describeTemperatureData() {
      return {
         {TemperatureWatcher.REALLY_COLD_F - 10.0,
            TemperatureWatcher.REALLY_COLD_RESPONSE},
         {TemperatureWatcher.REALLY_COLD_F,
            TemperatureWatcher.REALLY_COLD_RESPONSE},
         {TemperatureWatcher.REALLY_COLD_F + 10.0,
            TemperatureWatcher.NORMAL_RESPONSE},
         {TemperatureWatcher.REALLY_HOT_F,
            TemperatureWatcher.REALLY_HOT_RESPONSE},
         {TemperatureWatcher.REALLY_HOT_F + 10.0,
            TemperatureWatcher.REALLY_HOT_RESPONSE}
      };
   }
   private class SoftTemperatureWatcher {
```

```
        private double temperature;
        public SoftTemperatureWatcher(double temperature) {
            this.temperature = temperature;
        }
        @Override
        protected double readThermometerF() {
            return temperature;
        }
    }
```

@Test标注的 dataProvider参数指出要使用名为describeTemperatureData的数据提供器，来驱动测试方法。TestNG将去寻找一个将上述数据提供器的名称当作其标记的方法，或者去寻找具有该名称的方法。该数据提供器方法返回一个对象的数组的数组。其中这些对象的数组的大小必须是测试方法的参数列表的大小，而且数据必须是赋值兼容的类型[16]。我们的数据需要温度计返回的温度和预期的相应。数据提供器定义了我们感兴趣的一些值的一个范围，这些值都以与那些在类中定义的常量相关的方式来表示。

这个例子的核心位于 SoftTemperatureWatcher类。代码中所定义的这个类的构造器允许我们指定温度，而这个温度在正常情况下会来自于温度计硬件。它覆写了readThermometerF()方法来返回指定的温度。对该方法的可见性的提升和覆写，就是为了达到下面的目的而所需做的所有事情，即让TemperatureWatcher类更容易测试。

可以用多种方式来扩展这种方法。

8.2 封装循环条件

有时一个循环的终止条件是独立于该循环中那些被操作的数据的。

当寻找外部触发器（如事件、信号或传感器的读数）来做停止循环的操作时，就会出现这样的例子。设想正在为一个商用的烤箱编写预热控制器。针对初始加热周期的简化的伪代码可能类似于代码清单8-3所示。

代码清单8-3 针对外部触发的循环终止的简单的伪代码

```
while(underSetTemperature()) {
   keepHeating();
}
```

虽然我们把问题域稍微改变了一下，但这个例子还是非常近似于前面一节的例子。underSetTemperature()方法将持续加热的逻辑封装起来，并返回一个简单的true或false。可以想象，这种方法所做的事情就如同读取一个温度计，并将其与设定温度进行比较，以确定是否还需要增加更多的热量。

另一个常见的例子经常发生在通信处理和其他类型的处理中，即需要无限期地持续处理，或其中的终止条件对于一个简单的条件语句来说过于复杂。在这种情况下，就有了理论上的无限循环（见代码清单8-4）。

代码清单8-4 针对“无限”循环的伪代码

```
void infiniteLoop() {
   while (true) {
      // Do something
      if (/* Some condition */) {
         break;
      }
   }
}
```

这种形式的循环特别难以测试。一方面，可能很难强制其终止，进而有可能产生一个无限循环的测试。如果看到某些特定的循环迭代有奇

怪的行为，那么说明这个true 条件限制了我们控制循环迭代的能力。如果代码覆盖率要求我们对每个循环执行0次、1次和多次迭代，就像一些覆盖率统计工具的循环指标所要求的那样，那么在这种情况下是不可能执行这个循环0次的。

在所有这些情况下，我们可以封装循环条件，哪怕是对return true的一个微不足道的封装，这能带来以测试为目的的对循环的细粒度的控制，如代码清单8-5所示。针对0次执行覆盖条件的简单覆写将返回false而不是true。而更加复杂的以测试为目的的覆写，如循环的第5次迭代，可以计算其被调用的次数，并在5次迭代后返回false。

代码清单8-5 重构代码清单8-4使其具有可测试的循环控制

```
boolean shouldContinueLoop() {
   return true;
}
void infiniteLoop() {
   while(shouldContinueLoop()) {
      // Do something
      if (/* Some condition */) {
         break;
      }
   }
}
```

8.3 错误注入

可以使用注入来为软件测试引入我们所选择的数据，但也能用它来强制产生错误条件。尽管最佳实践建议让方法尽量保持小型和简单，然而很多时候，为了防止出现复杂的设计，会让一个方法的内部稍微复杂

一些。现在考虑如代码清单8-6所示的假想的代码，获取一个专有网络协议的响应。

代码清单8-6 用Java实现的面向连接的协议典型管理

```
public class NetRetriever {
   public NetRetriever() {
   }
   public Response retrieveResponseFor(Request request)
      throws RetrievalException {
   try {
      openConnection();
      return makeRequest(request);
   } catch(RemoteException re) {
      logError(“Error making request”,re);
      throw new RetrievalException(re);
   } finally {
      closeConnection();
   }
   }
}
```

这种逻辑是典型的面向连接的多协议的情况。实际上，它是这种情况的一个相当简单的变体。例如，如果正使用针对数据库连接的JDBC从一个准备语句获取一个结果集，那么可能有几样东西需要清理，其中包括结果集对象、准备语句对象、数据库连接，可能还有一个带提交或回滚路径的事务。

即使在这样一个简单的例子中，一个异常的几种后果会证实很难使用当前代码进行验证。如上面代码所示，没有办法验证连接被正确地清理了。如果不知道日志记录的实现细节，也无法知道验证日志记录行为

的难易程度。

但为了演示错误注入，让我们看看能很容易地验证什么。我们知道，如果makeRequest()方法调用抛出了 RemoteException 异常，那么retrieveRe sponseFor()方法应该抛出一个外覆了该异常的RetrievalException的实例。代码清单8-7显示了这个测试的样子。

代码清单8-7 测试代码清单8-6的错误条件

```
@Test(expected = RetrievalException.class)
public void testRetrieveResponseFor_Exception()
   throws RetrievalException {
NetRetriever sut = new NetRetriever() {
   @Override
   public Response makeRequest(Request request)
      throws RemoteException {
         throw new RemoteException();
      }
   };
   sut.retrieveResponseFor(null);
}
```

这里有两件事情可能看起来很奇怪，特别是在不大习惯Java的时侯。首先，@Test标注的expected属性指明JUnit应该只考虑当RetrievalException被抛出时测试就会运行通过。如果没有抛出异常或者抛出任何其他种类的异常都会导致测试失败。这允许我们用流畅的形式来规定对一个异常的期望，而不用繁琐地用try/catch逻辑来塞满测试。另外，该测试有局限性，即它无法验证上述异常中的任何事情，但是对于目前的目的来说它已经足够好了。

我们用来创建测试对象的构造被称作匿名内部类。它创建了一个派生自NetRetriever的类，覆写了makeRequest()方法。在C++这样的语言

中，针对这个目的，可以创建一个内部类或嵌套类，但该类需要一个类型名称。

最后，我们没有捕获retrieveResponseFor()方法的返回值，而且把null值作为参数传给了它。对于这个测试的目的来说，这两者都无关紧要。当一个方法抛出一个异常时，该方法不会有值返回。另外，我们正在覆写一个方法，来从这个在正常情况下应该处理request的方法中抛出异常。这些测试只有在被需要的地方才会被详细编写。

使用覆写来注入错误能极大地拓展测试的潜力。通常情况下，错误处理代码是最少被深思熟虑且最少被测试的方面。然而，应用系统在处理错误和维护客户业务方面的优雅程度，决定着公司声誉的破立。即使在难以避免的编程错误面前，测试这些错误路径也能为我们带来一个赢得客户内心和注意力的途径。

8.4 替换协作者

协作者是软件与其进行交互的组件。出于测试的目的，我们关心被测组件为了实现其目标所使用的那些组件。让我们看看前面NetRetriever例子的一个经过改写的版本（见代码清单8-8）。

在比较实际的例子中，会把连接（connection）与NetRetriever分开管理。除了支持关注点分离（separation of concerns）的原则，这样做还可以把连接用下面的方式抽象出来，即允许该连接被复用并能从一个更大的上下文中传递进来。虽然在这个设计的上下文中，对连接进行打开和关闭的管理，可能不是最合适或最合理的做法，但是现在让我们先忍耐一下。

代码清单8-8 改写代码清单8-6将Connection传递给构造器

```
public class NetRetriever {
    private Connection connection;
```

```
    public NetRetriever(Connection connection) {
      this.connection = connection;
    }
    public Response retrieveResponseFor(Request request)
      throws RetrievalException {
      try {
        connection.open();
        return makeRequest(connection,request);
      } catch(RemoteException re) {
        logError("Error making request",re);
        throw new RetrievalException(re);
      } finally {
        if (connection.isOpen()) {
          connection.close();
        }
      }
    }
  }
```

现在我们创建 NetRetriever对象的时候，就可以把 Connection对象传递给构造器。makeRequest()方法使用该Connection对象来发出请求。retrieveResponseFor()方法中的连接管理代码能够更加直接地管理这个连接。下面让我们来改变测试策略，通过Connection实例来注入一个错误，而不是像前面那样使用makeRequest()方法（见代码清单8-9）。

代码清单8-9 通过替换Connection协作者来注入异常

```
@Test(expected = RetrievalException.class)
public void testRetrieveResponseFor_Exception()
    throws RetrievalException {
```

```
    Connection connection = new Connection() {
      @Override
      public void open() throws RemoteException {
        throw new RemoteException();
      }
    };
    NetRetriever sut = new NetRetriever(connection);
    sut.retrieveResponseFor(null);
  }
```

我们没有覆写NetRetriever类的方法，而是覆写了协作者的方法，并用它来向被测方法注入了一个错误。在这个例子中，传统的封装和关注点分离的设计原则并没有阻碍可测试性，而是增强了它。用来支持被测类的功能的一个额外对象的出现，使得该对象能更加容易地在执行流中注入各种影响——在本例中是一个错误条件。

测试替身与基于接口的设计

注入协作者这一方法引入了测试替身（Test Double）[xTP]的概念，这是由称为stub、spy、fake和mock的替代组件（substitute component）所组成的家族的一个通用术语。虽然可能会听到许多人有些随意地使用这些术语，但Meszaros[xTP]还是有效地为每一个术语定义了其具体的特征。

stub仅充当协作者。

spy针对测试作用在协作者上的效果提供事后访问。

fake经常出于测试性能的原因用较轻量级的实现替代原有实现。

mock完整地记录了与协作者的交互过程，并在可能的情况下验证这些发生在测试期间或之后的交互。

mock——和特别是像Java的JMock 、EasyMock和Mockito这样的mocking框架——受益于基于接口的设计。如果能通过由协作者的接口

所规定的能力来识别出这些协作者，那么mocking框架就可以很容易地创建mock来进行注入。这些框架也在mocking具体类方面取得了一些进展，但是这种方法还是有局限性的。

像 JavaScript 这样的动态语言很少会被绑定到显式的接口上，尽管在使用时还定义了一个事实上的接口。通过把要被代替的函数替换为用于测试的记录对象，让我们能定义其被调用时所做的行为，Jasmine框架的spyOn功能提供了全方位的测试替身的变体。

在代码清单8-8所示的测试代码中，在检查open()和close()方法被调用了正确的次数以确保资源管理的正确性方面，mock特别有用。

8.5 使用现有的无操作类

在笔者看来，一个最没有得到充分利用的常见设计模式是空对象（Null Object）模式[17]。该模式的概念很简单。它不再返回普遍存在的null（或者是与其等价的各种形式，如under、nil或NULL）并将其作为一个特定的值来进行测试，而是返回一个期望类型的对象，该对象实现了与一个未知的、缺失的或模糊的值相关联的行为。在一个给定的情况下，几个这样的对象可以合理地存在，从而引发Martin Fowler所提出的作为范化（generalization）的Special Case模式[18]。

空对象的实现在测试stub的时候特别有用。例如，可以看一看把一个接口或抽象类作为参数进行传递的一些代码。如果代码清单8-7中使用的那个Connection是一个接口而不是一个类会怎样？甚至为了覆写单独一个方法，也必须实现所有的方法。在一些编程语言中，可以使用像Java语言中的JMock、EasyMock或Mockito这样的mocking框架。让我们研究一个几乎可以用于所有编程语言的技术。

人们可以很容易地把NetRetriever类想象成整个应用系统的一部分，它或许可以管理多个连接。并且至少为了实现应用系统用户界面的目

的，该应用系统可能有一个当前连接的概念。在一个实际的连接被选择或者甚至被定义之前，人们如何才能启用当前连接？现在我们不再像代码清单 8-10 所示的那样使用 null 到处对这个特殊值进行测试了，我们来创建一个NoConnection类，自动初始化当前连接。该类看起来像代码清单8-11所示的样子。

代码清单8-10 使用null值来表示错误情况的一个例子

```
if (currentConnection == null) {
   throw new InvalidConnectionException();
}
currentConnection.doSomething();
```

代码清单8-11 一个用空对象模式来实现Connection的例子

```
class NoConnection implements Connection {
   @Override
   public void open() throws RemoteException {
      throw new InvalidConnectionException();
   }
   @Override
   public void close() {
      // Who cares if we close a nonexistent connection?
      return;
   }
   @Override
   public void doSomething() throws RemoteException {
      throw new InvalidConnectionException();
   }
   ...
}
```

现在用于检查 currentConnection 是否为空的条件测试可以删除了。NoConne ction类的行为和人们预料的在没有连接的情况下发生的行为一样。实际上，在本例中，该类的行为更像是一个行为异常的连接所做出的行为。这不仅简化了整个应用系统的代码，也减少了发生困扰许多应用系统的可怕的NullPointerException（空指针异常）的可能性。

这对于编写测试的目的来说，还有一个额外的好处。现在我们有了这个接口的一个完整的实现来作为被注入对象的基础。有了这个空对象的实现，代码清单8-9的测试就可以把NoConnection用作一个stub了（见代码清单8-12）。

代码清单8-12 使用代码清单8-11的实现来进行测试

```
@Test(expected = RetrievalException.class)
public void testRetrieveResponseFor_Exception()
   throws RetrievalException {
Connection connection = new NoConnection() {
   @Override
   public void open() throws RemoteException {
      throw new RemoteException();
   }
   };
   NetRetriever sut = new NetRetriever(connection);
   sut.retrieveResponseFor(null);
}
```

实际上，就像从方法签名中的throws语句所推测的那样，如果InvalidConnection Exception派生自 RemoteException，那么我们甚至都不必覆写 open()方法，就能把测试简化成代码清单8-13所示的样子。也可以让这个异常派生自RuntimeException，但只有当在这个上下文中抛出RuntimeException是一个合理的期望的时候，才能这样做。而在这个上

下文中没有什么能够表明抛出RuntimeException是所预期的。

代码清单8-13 依赖空对象的实现来简化测试

```
@Test(expected = RetrievalException.class)
public void testRetrieveResponseFor_Exception()
    throws RetrievalException {
  Connection connection = new NoConnection();
  NetRetriever sut = new NetRetriever(connection);
  sut.retrieveResponseFor(null);
}
```

虽然这是一个强大的技术，但也应该知道这引入了测试与一个额外的生产代码类之间的耦合。然而，在空对象实现的情况下，这个耦合往往是无足轻重的。

第9章 调整可见性

面向对象封装的理念，通常会引导我们最大程度地限制方法和属性的可见性，而且仍然要满足软件所需的功能。一般情况下，这种引导是好的。它能让我们基本上可以做到将数据元素进行虚拟化，使得所有针对这些元素的操作都能符合一致的期望。公共的方法代表了外部世界与正被建模的对象之间进行关联的那些固有的行为。在设计目的的驱使下，其他方法的可见性会较小。一些方法仅能让相同类型的其他类（如派生出来的类）可见。不同的编程语言在访问级别上会有额外的变化。当设计编程语言的访问级别和使用这些访问级别来实现完美封装的软件时，很少会去考虑可测试性的问题。这让我们面临这样的处境，即需要利用现有编程语言的特性，放松理想情况下的封装，从而提高软件的可测试性。本章将介绍几种用来调整可见性以提高可测试性的技术。

9.1 用包来包装测试

许多编程语言使用包（package）或模块（module）作为帮助组织类的机制。这些语言当中有许多都对同一个包中的代码提供了额外的的访问权限。Java对于同一个包中的代码有特殊的访问级别。Ruby的模块提供了命名空间，而这些模块也是mixin[19]概念的基础。Python的模块很像类，但它的包是专门用于组织代码的。但需要小心，Perl的包声明定义的是类的名字，而不仅仅是它所属的命名空间。一会儿我们就能看到这个差异是至关重要的。

可以利用共享的包成员的权限来访问更多的被测软件中的成员。让

我们考虑这一点是如何影响如代码清单9-1所示的Java代码的可测试性的。

代码清单9-1 Java中包和访问级别的例子

```
package my.package.example;
class MyClass {
   public void runMe() { }
   protected void inheritMe() { }
   /* package */ void packageMe() { }
   private void hideMe() { }
}
```

任何类都能访问runMe()方法。然而在另一个极端，只有MyClass内部的代码才能调用hideMe()方法。对于inheritMe()方法最常见的理解是由派生的类来访问。my.package.example包中的代码能够调用packageMe()方法和inheritMe()方法，后者经常被忽略。而这最后一个经常被忽略的特性，在很大程度上为可测试性提供了支持。

虽然下面这种做法越来越少，但是 Java 开发人员有时会将测试放到一个专门的“test”子包中，如代码清单9-2所示。

代码清单9-2 将测试放入一个专门的“test”子包中

```
package my.package.example.test;
class MyClassTest {
   public void testRunMe() { } // OK
   public void testInheritMe() { } // Not accessible!
   public void testPackageMe() { } // Not accessible!
   public void testHideMe() { } // Not accessible!
}
```

这种做法经常是出于有必要让测试代码与生产代码相分离的考虑。Java的包所在的目录是相对于classpath中定义的根目录的。由于classpath

能够包含多个目录，所以在将测试与被测代码相分离时，可以将二者放到相同的包中，而该包位于不同的classpath的目录条目下，以带来更大的可访问性（见代码清单9-3）。

代码清单9-3 将测试与被测软件放到同一个包中

```
package my.package.example;
class MyClassTest {
   public void testRunMe() { } // OK
   public void testInheritMe() { } // OK
   public void testPackageMe() { } // OK
   public void testHideMe() { } // Not accessible!
}
```

我们现在已经将潜在的测试覆盖率增加到了原先的3倍，而仅仅是通过改变测试代码的包。为测试和生产代码定义单独的classpath根目录允许我们清晰地将它们进行分离。

然而这种方法对 Perl 不起作用。Perl 有一个类似于 Java 的 classpath 的名为PERL5LIB的环境变量，用来帮助定义针对Perl包的搜索路径。然而，包的声明方式略有不同，如代码清单9-4所示。

代码清单9-4 Perl中包声明的方式与Java中的有所不同

```
package My::Package::Example::MyClass;
use Class::Std;
{
sub testMe { }
}
1;
```

这个包语句不仅声明了从根目录到模块的路径，也声明了模块本身的名字。将测试声明在同一个包中，会让其成为该模块的一部分，会潜在地改变被测代码所有实例的行为。这是在测试中非常不愿被看到的

trait[20]（特征）。所以为了保持测试和生产代码在逻辑上的分离，必须声明带有额外组件的包，即使它们在物理上已经分别放到了不同的目录中。

只要编程语言允许，包提供了一个有用的方式来提升测试对其所测代码的访问。编程语言的语义决定了是否能够在特定的编程环境中利用上述行为。

9.2 将其分解

有时候，包级别的访问尚不足够或尚不可用。如果发现自己无法测试被紧密封装的功能，那么就可能表明类做了太多的事情。开发人员经常会创建私有的方法来简化公共接口的实现。然而，方法通常应该是很小的。如果能够在功能上将一个方法分解成一些较小的具有更高抽象层次的方法，那么当它们被放到一起，特别是当它们被复用时，就意味着这些方法构成了另一个类。

将私有实现提取到一个类中，能使其变得直接可测。只要附带的类不是 public的，Java就允许在一个文件中包含多个类，而这些类就处于包级别访问的范围。也可以将被提取的类变成公共的，以便于测试。

一旦提取了新的类，就需要决定如何使用它。可以令其成为继承层次的一部分，或者可以将其组合进原有的类，并通过引用来调用它。代码清单9-5展示了一个可以用这两种方法来分解的类的例子。代码清单9-6展示了该类被分解为3个类。

代码清单9-5 一个等待重构的Java类

```
public class Car {
   public void start() {
   ...
   igniteEngine();
```

```
    ...
    }
    private void igniteEngine() { ...}
}
```

代码清单9-6 重构代码清单9-5来分离关注点以增强可测试性

```
public class Engine {
    public void ignite() { ...}
}
public class MotorVehicle {
    private Engine engine;
    protected void igniteEngine() {
        engine.ignite();
    }
}
public class Car extends MotorVehicle {
    public void start() {
        ...
        igniteEngine();
        ...
    }
}
```

9.3 更改访问级别

调整可见性的下一个技术在概念上最为简单，但在实践中经常是最富有争议的。之所以说它简单是因为这种方法仅简单地改变了方法声明中的关键字。在Java或C++语言中，可以将 private 改为 protected，或将

protected 改为 public[21]。作为private 的升级，Java 也提供包默认（package default）的访问级别。在 Perl 的Class::Std[22]模块中，可以将 PRIVATE 这个 trait 改为 RESTRICTED，或者删除RESTRICTED这个 trait。道理是一样的。

争议出现在本章开头所提到的理由上。软件设计者经常很仔细地选择访问级别，来捕获将要被建模成类的两个实体之间的关系。对于外部世界所看到的实体而言，public方法代表了其所固有的行为。protected 方法所代表的“行为”，被相同类别或相同类型但通常不是外部可见的所有事物所分享：这基本上是改变较高层面上的概念性行为，或“配置”它们的类型或实例的方法。例如，所有的动物都可以移动，但它们移动的方式差别很大。private方法则纯粹是实现的细节。

封装的纯粹主义者会说，这些可见性的级别，代表了实体的一个自然的模型，应该不容侵犯。虽然对于自然界的实体来说，这是一个有说服力的观点，但对于所有那些人工合成的实体，如数据结构、管理器、解析器等，又会怎样呢？这些实体是一些理论结构的实现，是进一步工作的基础，它们中的大多数都有在功能上等价的多个“自然”的表达。

稍稍务实一些的设计者或许会认识到，我们已经针对访问级别建立了简化的语义约定。这意味着只要有三或四个级别，就能大大简化现实世界中访问变化的范围。然而，现实世界却不是分级的。设计者或许会争辩说，由于各种原因这些约定都应该被遵循。

这些约定背后的许多原因都是可信的。例如，笔者尚未遇到过需要改变私有属性的可见性的测试环境。然而，在第6章中所讨论的对测试进行bootstrapping的结构，就是为了避免改变可见性而设计的。

但是，一些约定源自不同的上下文。那些我们最感兴趣的有关上下文的差异，都与运行测试的资源有关。当今面向对象编程语言背后的原则，都可以追溯到Smalltalk的首次引人注目的商业实现。Smalltalk创建于20世纪70年代，在20世纪80年代首次公开发布。那时，个人电脑正处

于萌芽期，而且不是非常强大。那时一般的观念认为，计算机用于计算的时间比人类计算的时间要昂贵得多。而对于测试来说，这意味着测试自动化没有机会成为计算机语言设计的驱动力。换句话说，访问级别是在没有考虑测试访问特权的情况下设计出来的。

笔者现在阐释了一个有些冗长的观点，即可以放宽“完美的”封装原则，来解决现代编程语言中所缺乏的对测试访问特权的显式支持。对于正在阅读本书的读者来说，这是一个好机会，可以来了解为何笔者会有这样的观点。而对于没有阅读过本书的人来说，上述观点可能不会使人信服，除非本书的读者买了本书送给这些人。

一旦跨过了有关编程语言在访问级别描述上的障碍，改变了访问级别，那么问题就变成如何用损害最小的方式来做到这一点。不幸的是，这个攻略尚不存在。然而，在放松一个成员的可见性时，建议找一个方式来表达这个意图。用注释和文档来表达可能就行了。在Java中，一些类库，如Google的Guava[23]框架，包含了一个@VisibleForTesting标注来作为一个显式的指示器，尽管它并没有做强制访问的事情。

9.4 仅用于测试的接口

可以做一些事情来缓解改变访问级别所带来的危害。可以引入仅用于测试的接口，来将访问级别的变更与稍弱一些的类的分解结合起来。在一般情况下，使用接口这样基本的类型而不是具体的类，可以提高可扩展性和可测试性，从一个面向对象设计的角度来看，这样做能将行为规格的定义与其实现相分离。此外，Java的那些 mocking 框架能与接口配合得最好，尽管它们也在处理具体类的方面做得越来越好。

通过以基本类型的方式来使用接口，那些调用者就只能访问在接口中定义的方法，除非它们有直接去使用实现类的坏习惯。这样会减低那些非接口成员的访问级别的重要性。然而，还可以更进一步，通过将这

些成员方法组织到一个备用的具体类实现的接口中，让这些成员方法的角色更加显式一些，如代码清单 9-7所示。

代码清单9-7　在Java中使用仅用于测试的接口来帮助指定那些要提升访问级别的成员。生产代码会引用Toaster接口，但测试代码会使用ToasterTester接口

```
public interface Toaster {
   public void makeToast();
}
public interface ToasterTester extends Toaster {
   public void verifyPower();
   public void heatElements();
   public void waitUntilDone();
   public void stopElements();
   public void popUpToast();
}
public class DefaultToasterImpl implements ToasterTester {
   public void makeToast() {
      verifyPower();
      heatElements();
      waitUntilDone();
      stopElements();
      popUpToast();
   }
   ...
}
```

9.5 命名那些尚未命名的

许多编程语言支持在嵌套的作用域中声明实体。Java支持内部类。C++支持嵌套类。JavaScript允许在几乎任何作用域中声明函数。

一些编程语言，包括 Java 和 JavaScript，都允许匿名声明实体。在Java 中，通常为了简单或使用像Swing事件处理器那样的一次性回调类（见代码清单9-8），可以创建匿名内部类。JavaScript开发人员为了能够立即执行，普遍用匿名函数来实现回调函数（见代码清单 9-9），并将函数声明进行外覆以冻结闭包（freeze closure）。

代码清单9-8 一个Java Swing事件监听者的匿名内部类的例子

```
button.addActionListener(
   new ActionListener() {
      public void ActionPerformed(ActionEvent event) {
         System.out.println(event.getActionCommand());
      }
   });
```

代码清单9-9 在jQuery中用作鼠标单击事件处理器的JavaScript的匿名函数

```
$('.clickable').bind(''click',function(event) {
   alert('Element has class(es): ' + event.target.className);
});
```

匿名实体虽然创建起来很方便，但难以测试。它们经常在其所声明的作用域之外的任何作用域中都没有可见的引用。在许多情况下，执行的上下文决定了它们的签名（对于函数而言）或接口（对于类而言）。一个简单的Extract Class或Extract Method的重构能够通过使用名字来将匿名实体带入一个可见的作用域，从而可以通过引用它们来达到测试的目的。

9.6 变为friend

相对少量的编程语言具有C++语言的friend关键字的等价物。例如，微软的C#具有InternalsVisibleToAttribute类，可用于从指定的上下文中赋予访问权限。

尽管许多人认为friend关键字是“脏编程”（dirty programming），破坏了封装的原则[24]，然而这一点也经常被认为对于测试的目的来说具有正面价值，特别是对于单元测试。考虑一个扑克牌型[25]（poker hand）的C++实现（见代码清单9-10），这里只考虑五牌扑克[26]（five-card stud）的一个简单游戏。当然，玩家都不想向任何人透露手中的牌。

代码清单9-10 带有一些名正言顺的私有成员的C++类

```
class MyPokerHand
{
private:
   Card hand[5];
public:
   void receiveCard(Card& card);
   void layDownCard(Card& card);
   void evaluateHand();
};
```

虽然其他玩家和发牌的庄家不应看这位玩家手中的牌，除非这位玩家把牌都亮出来，但测试应该能够看到这些牌，以便能验证这位玩家在玩的时候能正确地算牌。通过在MyPokerHand类中添加语句

```
friend class MyPokerHandTest;
```

测试就能检查这些牌以确保玩牌策略能像我们所期待的那样赢来财富。

9.7 通过反射来强制访问[27]

Java的反射特性使其从大多数其他的强类型和需编译的编程语言当中脱颖而出。Java 反射允许在运行时对类型做深入的内省[28]（introspection）和操作。如果能明智地加以利用，这一特性能在生产系统中产生精干的通用代码，尽管必须要小心避免其所固有的性能损失。使用反射的力量也可能会犯下大错[29]。然而，还是让我们利用它来做好事情：增加我们的测试能力。

考虑仅有一个私有方法的简单类，如代码清单9-11所示。就其本身而言，这个类没有用处，因为里面的方法无法被调用，但它却很好地刻画了测试私有方法所面临的问题。不管方法是否是私有的，一个足够复杂的方法或者被复用的方法都值得拥有它自己的测试。过程上的分解封装了逻辑功能的各个方面，而这些方面都能被独立地测试，无论这些功能是否应在类之外被暴露出来。根据这个推理，针对私有方法的测试迫在眉睫。

代码清单9-11 仅有一个的私有方法的简单类

```
package com.vance.qualitycode;
public class ReflectionCoercion {
   private int hardToTest() {
      return 42;
   }
}
```

该如何测试这个方法呢？除了类型内省和调用之外，通过AccessibleObject类的一个简单而强大的名为 setAccessible()的方法，Java的反射也允许操纵一个元素的可访问性。在 SecurityManager 检查和一些其他的小限制条件的制约下，将setAccessible()方法的实参设置为 true 能让 hardToTest()方法可被访问。换句话说，几条简单的反射语句就能做到调用一个类的私有方法（见代码清单9-12）。

代码清单9-12 使用反射强制测试来访问私有方法

```
public class ReflectionCoercionTest {
   @Test
   public void testHardToTest()
         throws NoSuchMethodException,
                InvocationTargetException,
                IllegalAccessException {
      ReflectionCoercion sut = new ReflectionCoercion();
      Method targetMethod =
         sut.getClass().getDeclaredMethod("hardToTest");
      targetMethod.setAccessible(true);
      Object result = targetMethod.invoke(sut);Assert.assertTrue(result
instanceof Integer);
      Assert.assertEquals(42,result);
   }
}
```

首先，上面的代码实例化了我们的被测类。接下来，使用反射对要测试方法进行了内省。在更加复杂的情况下，或许要在具有相同方法名的不同的重载（overload）方法之间进行区分，但这仅仅是反射的一个更加复杂的应用而已。屏蔽方法的访问级别简单到只要调用setAccessible()方法就能办到[30]。

如果每次测试私有方法时都要写上面的反射代码，那么这个代码就会既笨拙又重复。幸运的是，一些Java工具以各种形式提供了这个功能，包括dp4j[31]、PowerMock[32]和Privateer[33]。

9.8 声明范围变更

一些更加动态的语言，尤其是笔者所使用的Perl语言，允许以声明

的方式来改变代码块的作用域，这一点很像基于反射的强制。尽管用这个功能来撬开其他人的封装会带来巨大的邪恶隐患，但是我们只用它来做好的事情。它利用有关使用包的规则的一般原则，以一种更加有针对性和目的性的方式，令测试能够访问到它们正在测试的代码。

让我们看一个针对Perl的面向对象开发的使用Class::Std模块的例子（见代码清单9-13）。Class::Std可以将方法关联（ascribe）到代表其可见性的那些trait上，很像Java和C++语言中的protected和private关键字。如果使用Class::Std，就要相应地使用RESTRICTED和PRIVATE这样的trait。Class::Std使用Perl的动态反射能力来截获对这些方法的调用，确保这些调用来自适当的作用域，或者来自定义它们的类，或者在使用RESTRICTED的情况下来自从该类所派生出来的类。

代码清单9-13 带有私有方法的使用Perl的Class::Std的类

```
package My::Class;
use Class::Std;
{
   sub some_internal_method : PRIVATE
   {
      # Do something the world need not see
   }
}
1;
```

如果上面代码中的some_internal_method足够复杂，那么最好是通过其自身来进行测试，而不是仅通过其调用者。该方法的PRIVATE声明使测试变得很困难……除非使用新的技巧（见代码清单9-14）。

代码清单9-14 使用声明性的范围变更来测试代码清单9-10中的代码

```
package My::Test::Class;
use base qw(Test::Unit::TestCase);
```

```
sub test_some_internal_method {
    my $self = shift;
    my $sut = My::Class->new();
    my $result;{
        package My::Class;
        $result = $sut->some_internal_method();
    }
    # Assert something about $result
}
```

测试中的包声明哄骗了Class::Std，使其以为some_internal_method正被其自己的类所调用。此处两侧的括号，能将声明性的重定位仅限制在这个调用的狭小的作用域中，就像良好的代码公民所做的那样。即使选择为了可测试性而破坏封装，我们也仅用最短的时间在所需的时候才这样做。

第10章 间奏：重温意图

第2章讨论了有关意图编程和理解与测试代码意图的主题。有了前几章介绍的测试技术，让我们考虑一个更加复杂的例子。

举目四望，读者会发现那些将各种意图混杂在一起的代码。大多数情况下，这种情况在开始的时候发生是因为职责还未成熟到足以保证分离成功的程度。只要开发人员认识不到这些关注点[34]（concern）正在逐步清晰，这种情况就还会继续。所有这些都与 Bob Martin 大叔的 SOLID 原则[35]中的第一条——单一职责原则（Single Responsibility Principle）有关。

然而，让关注点就那么纠缠在一起的原因，有时候是习惯造成的惰性，即我们一直以来看到的代码就都是那样实现的，所以就继续照那样做下去，有时候是实现语言不支持那些分离职责所必需的结构。在使用设计模式时，后者就会经常发生。虽然模式本身并没有规定一个具体的实现，但是典型的实现已经逐步形成。

本章将使用单例模式（Singleton pattern）[DP]来探讨这种纠缠的问题。典型的单例模式特别难测试，所以下面将讨论测试的可选项和一些安全有效的测试所需的折衷方法。然后再讨论与分离关注点相关的普遍问题。

10.1 测试单例模式

单例模式能够保证对一个对象的唯一实例进行统一受控的访问[36]。在满足上述保证的情况下，它扮演了全局变量在面向对象世界里

的替身。在其最纯粹的应用中，它对应一个真实的唯一的物理资源。它经常用于代表应用程序中唯一的事物，如远程服务的入口点或一个应用的注册表。

有几种方法可以实现单例。在Java中典型的实现[37]如代码清单10-1所示。不像在C++或 C#中，在 Java 中无法创建一个可复用的单例基类，所以编程语言的限制会导致模式的持续再实现（continuous reimplementation）。

代码清单10-1 单例模式的一个典型的Java实现

```
public final class Singleton {
   private static Singleton instance = null;
   private Singleton() {
      // Initialization
   }
   public static synchronized Singleton getInstance() {
      if (instance == null) {
         instance = new Singleton();
      }
      return instance;
   }
   public void doSomething() {
      // What the class does
   }
}
```

让我们快速过一下上面的代码。首先，类的定义本身声明了类不能被子类化，使得这个唯一的实例不能再被扩展以实现更专门的功能。然后，类的变量instance引用了该类唯一的实例。私有构造器能阻止不受控的类的构造，使得 getInstance()方法成为获取该类实例的唯一一个线

程安全的方式。最后，该类会有一些额外的方法，用来达到其在应用程序中的目的。

10.2 单例的意图

对于一个任意给定的单例的实现，一个类中实际存在两个目的。第一个目的很清楚，即单例模式特征的实现。第二个目的就是该类自身的功能：与对单一资源进行建模的类相关联的行为。

通常情况下，如果在一个类中发现这种明确的双重目的，那么就要将其进行重构以分离这些关注点。然而，单例的性质（nature）要求这些分离的意图要处于同一个类中。在大部分编程语言的继承层次中，类变量的行为都存在着编程语言自身的问题，如果不添加一个注册表的复杂性，就会禁止将单例的性质重构到超类（superclass）中。重构到一个组合关系中会在某种程度上暴露类的构造，从而违背了单例的意图。

10.3 测试的策略

在了解了单例的意图之后，就可以专注于这个意图所带来的价值。鉴于有两个分离的意图集合，可以尝试图把重点放在单独测试这两个意图上，并希望能简化所测试的问题。

关于难以测试的设计的一点看法

当遇到一个难以测试的设计时，应该形成一个习惯：问自己是否存在一个更简单、更可测的设计，也可满足需要。当被问到这个问题时，笔者总能惊讶地发现，居然有如此多可以消除的代码，系统居然可以变得如此简单。

在测试先行的方法中，有一个隐藏的很少被认识到的好处，即在测试先行的过程中发生的系统的简化。当生产代码更简单、更少时，测试

就会更容易编写，其数量也会更少。最终的结果系统只需花费较少的时间来编写和维护，且其中不大可能包含bug。

单例模式就是一个很好的例子。虽然单例模式强制要获得一个单独的类的实例，但是在有足够的框架或合作的情况下，这种强制可能并没有什么必要。

依赖注入就是单例模式的一个极好的替代。通过给构造器或正被使用的方法添加另一个参数，或通过使用诸如 EJB3、Spring 或 Google Guice[38]这样的框架所提供的那些依赖注入的基础设施，就能够分配一个单独的对象，并使用注入来确保它被传递到各处。这就能通过约定或配置，而不再是通过构造，来使得对象成为单例。

完全消除模式的使用经常是让代码变得可测的最佳方式。

10.3.1 测试单例的性质

考虑单例的性质，我们需要测试两个基本行为和两个可选行为。两个基本的行为是：

单例的getInstance()方法返回类的一个实例；

对于所有的调用，单例的getInstance()方法返回同样的实例。

让我们先考虑这两个基本行为，然后再谈谈那两个可选行为。注意我们刚才没有说任何有关实例的初始化状态的事情，那部分价值不是getInstance()方法增加的，相反，它是通过类成员初始化和构造器增加的，这些都是与之相分离的另一堆意图。

第二个行为，即单例总是返回同样的实例，正是那个令单例难以测试的行为。为什么会这样呢？

最常见的原因是，不管愿不愿意，共享一个实例从本质上就会强制我们使用共享的 fixture[xTP]。尽管共享的 fixture 可能会有用，但为了避免测试之间的交叉影响，总是会优先选择新的fixture，因为共享的状态会从一个测试被带到另一个测试。此外，如果尝试在不同的线程之间

并发运行测试，那么被共享的实例将会在各个测试之间造成潜在的数据冲突。所有这些都会导致不稳定测试（Erratic Test），从而减少开发人员对自己的测试库（test base）的信心，并有损生产力，因为解决测试失败需要不断的诊断和调试。

要简单地测试单例行为，可以通过在一个单独的测试方法中测试两个行为的做法来解决这个问题（见代码清单10-2）。

代码清单10-2 测试单例的getInstance()方法的基本行为

```
@Test
public void testGetInstance() {
   Singleton sut = Singleton.getInstance();
   Assert.assertNotNull(sut);
   Assert.assertSame(sut,Singleton.getInstance());
}
```

上面单例实现的可选行为是：

延迟加载（lazy loading）行为，即getInstance()方法仅在第一次请求时才分配对象；

单例的getInstance()方法用线程安全的方式执行了对象的分配。

这些行为对于许多单例实现来说都是共同的。我们此时不会直接去解决它们。通过提供针对实例属性的访问，第9章所讨论的强制访问能被用于测试延迟加载。第13章所讨论的线程测试技术可以用来验证线程安全。

10.3.2 对类的目的进行测试

我们已经将针对单例性质的测试与针对类的功能目的的测试相分离，而且已经解决了如何测试单例性质的问题。测试功能目的就会容易一些，对吧？可能吧，除了一件事：除非通过单例机制，否则我们无法构造一个实例。测试单例的目的的困难之处是无法通过新的fixture来对

其进行测试。编写的测试越多，创建不稳定测试的机会也就越高。那么，有什么解决问题的选项吗？

简单地说，我们的选择有：

删除单例的性质；

不做改变且接受对共享的fixture进行测试的现实；

放宽针对实例的约束；

或者放宽针对类的构造的约束。

让我们依次看看每一个选项。

最不经常考虑的选项是简单地改变设计来避免使用单例模式。除非出于明显的提高性能的动机，比如在创建公共可见的API时控制很重要，或者正在与一个物理资源建立关联，否则可能并不真的需要使用单例（参阅提示内容“关于难以测试的设计的一点看法”）。大部分时候对于应用程序来说，仅仅创建一个实例，并通过约定和其他机制来支持唯一性，就足够+了。然而，有时单例却是出于设计而做出的选择[39]。

接受对共享的fixture进行测试的现实，可能在短期内是可行的，但这没有为测试库的增长做出强有力的规划。接下来就剩下放宽单例约束的选项了。

一般情况下，放宽单例约束就放开了对其进行使用的可能性，而这与单例的意图背道而驰。这里是有意在单例的强制与可测试性之间做出权衡的。应该选择一些替代方案，使得在封装上导致恶劣结果的破坏变得最小，并用单例测试解决大部分问题。

1.放宽针对**instance**的约束

一个经常应用的解决方案是为访问受限的实例创建一个 setter。一个针对前面所列单例代码的典型实现如代码清单10-3所示。

代码清单10-3 针对测试而放宽对instance的访问

```
protected synchronized
Singleton setInstance(Singleton newInstance) {
```

```
Singleton oldInstance = instance;
instance = newInstance;
return oldInstance;
}
```

该方法是同步的，确保了非原子性的测试和getInstance()方法中的set行为的一致性。不同于经典的 setter，该方法返回旧实例值。因为这个方法专门用于支持测试，返回旧实例值能允许测试在完成时将其复位。protected的使用显式规定了允许其他包成员来使用它的访问级别。也可以使用包的默认访问级别，但许多有关代码整洁的调理方案有不接受它的充分理由。

现在已经创建了一个机制来允许注入我们自己的类的实例，对吧？但是等等，如何才能创建那个实例呢？构造器是私有的，所以不能直接创建实例。类被声明为final的，所以不能由其派生出子类。但无论如何放宽这两者中的任何一个都能带来构造上的宽松，而最初的情况为达到这样的目的增大了难度。

目前还尚未真正解决我们那个基础性的问题，即共享的fixture的那个问题。现在已经增添了替换（swap）出fixture并将其还原的能力。那不会强制任何特定的测试替换fixture，再保证测试能将其还原回来，或者阻止测试来改变它所还原的fixture。尽管已经同步了fixture的替换，但考虑到测试的完整运行，这并不能确保能实现事务性的操作，所以测试运行的线程将会遇到竞态条件。

此外，这里还产生了代码腐臭——生产代码中出现测试逻辑（Test Logic in Production）[xTP]。现在已经给生产代码添加了一个专门支持测试的方法，但却没有带来令人信服的利益或保障。

这不是一个令人信服的选项。

2.放宽构造的约束

放开对构造器的访问后，就可以用一种支持测试的并发执行的方式

来创建新的fixture，并独立于其单例的性质，全面地测试类的目的。于是问题就变成：我们应该如何才能做到这一点？

一般的答案是：使用暴露得最少的方式。具体细节因所使用编程语言的可用设施而异。所应用的工具在第9章已经有所描述。在代码清单10-1中，有两个主要的选项。

仅对代码做一些简单的修改，就可以将构造器的访问指示符（access specifier）从private改为默认（通过删除private）或者protected。在两者中选择哪一个取决于所能容忍的隐式默认访问的作用域。没有必要将其变为 public。如果将单元测试放到与被测类所在的包相同的包中，那么就能访问构造器来为每一个测试创建一个新的fixture，如代码清单10-4所示。

代码清单10-4 使用被放宽的针对构造器的访问来测试我们的单例

```
@Test
public void testDoSomething() {
   Singleton sut = new Singleton();
   sut.doSomething();
   // Make assertions about the effects of doSomething()
}
```

另一种方法是使用反射来强制对私有构造器的访问。这种方法的代码比较繁琐，或许最好的选择是将它封装到helper方法、fixture创建的方法或隐式的setup方法中。所以这些都有令代码更加难以阅读和理解的副作用，但为了保持单例的封装性这样做或许是值得的。

可以使用第9章的强制访问技术，针对带有基于反射的强制的私有构造器，重新编写访问代码。如果所有运行这些测试的开发人员都使用足够统一的开发环境，那么这就能工作得很好，但针对开源项目这样的代码库来说，这将会提高参与开发的门槛。

C++还提供了一个更具声明性的替代方案，即使用 friend 关键字。

如果一个类将其相应的测试类声明为 friend，那么该测试类就能访问该类的所有私有成员。这种方法并没有通过访问级别的提升或强制访问来提供细粒度的控制，而是保留了比前者更好的封装性，并且把选择权交给实现这一点也比后者要好。只需在 Singleton类中添加一行

friend class TestSingleton;

就能让 TestSingleton 测试类完全访问其内部。可以通过将每一个测试方法声明为一个 friend函数来把这件事做得更精确一些，但是随着时间的推移，这种做法将变得更加难以维护。

Perl的声明性作用域的变更的适应性，随Perl对象的使用方式的不同而有所不同。假如正在用一个 Perl 的面向对象的方式，支持私有构造[40]，那么其可访问性，经常会像代码清单10-5中的代码所展示的那样被绕过。

代码清单10-5 使用包的重新声明来直接调用单例构造器

```
package TestSingleton;
...
sub testDoSomething {
   my sut;
   {
      package Singleton;
      sut = Singleton-＞new();
   }
   ...
}
```

在Perl的面向对象的方法中，Perl的包名几乎在哪里都可被当作类名。假如有一个约定——要求测试在包名前面加一个 Test前缀，那么这样的代码就创建了一个从属的代码块，然后在其中重新声明处于原代码块中的包，从而令Perl认为这个针对构造器的调用来自类的内部。对于

不大习惯Perl的人来说，他的胃这会儿可能会略有恶心的感觉，但这一切都是为了可测试性而做出的权衡。

10.4 独具慧眼的意图

本章做了几件事情。其中很明显的一个是选取了一个常见的——尽管有时会颇具争议的——设计模式，并在保持其意图的本质的情况下，对测试工作进行了分解。然而更重要的是，我们认识到，就算是有人告诉我们一些模式用起来没有问题，在测试时也要仔细检查。借此更进一步，以地道的方式使用那些公认的设计模式（作为组块[41]或出于习惯），哪怕是最低限度的，都会让测试变得复杂，然而这可能是不必要的，还会无意中让那些关注点彼此交织起来。

需要确保任何有关关注点的纠缠都完全是有意而为的，而且代价是合理的。测试将会展示关注点在哪里发生了混合，fixture将会变得庞大笨拙，测试将会变得冗长而复杂。由于那个看似重要的设计决策，使得测试似乎不可能编写出来。

当这些事情发生时，整个测试都会出现代码腐臭。测试表达的或者想去表达的是一个难以满足的意图。此时应该考虑改变设计，并以澄清代码意图的方式分离关注点。

第11章 错误条件验证

可以让一个精心编写的软件以其实现主要功能的同样方式来处理故障（failure）。相比有关产品的博客内容、论坛评论和推特内容，由于软件故障而产生的技术支持请求的数量会更多，而是否能处理好后者对于产品的声誉更加重要。但错误路径往往是代码中被测试得最少的部分。

本章专注于测试故障路径。这在很大程度上依赖于前面章节的材料。实际上，在用于验证和行为注入的新技术方面，本章没有添加什么内容。相反，本章展示了，如何将前面讨论的技术运用到诱导和验证代码中可能存在的故障的问题上。

在本章的讨论过程中，将会看到一些改善错误处理方式让它们更有价值、更可测试的手段。本章最后一节总结了一些在运用这些手段时需要注意的有关设计的关注点。

11.1 检查返回值

面向对象设计的主导地位并未摒弃API设计的过程模型的使用。对于陈旧但无所不在的过程式的API（如POSIX）的使用和适应，会导致使用返回值和/或全局errno （错误号）来表达与错误条件相对应的有意义的值。许多POSIX的调用，如open(2)[42]，在出错时会返回一个整数-1，并使用一个errno的值的范围来表示错误细节。

为了能进行测试，前面已经探索了多种将值注入到代码中的方式。我们使用同样的技术来注入过程错误代码。通常情况下，可以薄薄地封

装诸如 POSIX 这样的过程性的系统API，这样它们就能被覆写以达到错误注入的目的。在其他时候，可以使用更厚重的封装——有时是以使用库的形式——来彻底地将那些返回值的错误语义翻译成一个基于异常的等价物，来更加自然地与面向对象的设计相匹配。

本章剩余部分将侧重于基于异常进行验证的方式。

11.2 验证异常类型

正如第8章所展示的那样，面向对象的测试框架让验证一个特定类型的异常被抛出变得很容易。通常情况下，这就足以确定代码的正确性了。许多方法只抛出一个异常，或仅有一个需要进行测试的有关错误处理的代码路径。

代码清单11-1中的代码代表了一个统一的异常处理策略，其中错误被外覆到一个应用程序特定的、为此上下文量身定做的异常中。

代码清单11-1 一个典型的统一异常处理策略

```
public class SomeClass {
   public void doSomething() throws ServiceFailureException {
      try {
         makeNetworkRequest();
      } catch (RemoteException rexp) {
         throw new ServiceFailureException (rexp);
      }
   }
}
```

所有makeNetworkRequest()方法抛出的RemoteException异常，都将被外覆到一个特定于如下上下文的异常中：在这种情况下，就是“做点什么事情”的老一套的做法。有人可以令人信服地争辩说，通过这个方

法的错误处理而增添的价值就是这样的外覆，而且那个特定的衍生出来的 RemoteException 与该方法的操作并不相关。在这种情况下，如果makeNetworkRequest()方法也不会抛出那个异常，那么验证ServiceFailureException异常被抛出或许就足够了。这段代码的JUnit测试如代码清单11-2所示。

代码清单11-2 针对代码清单11-1的JUnit4测试

```
@Test(expected = ServiceFailureException.class)
public void testDoSomething_Exception() {
   // Create fixture
   sut.doSomething();
}
```

@Test标注的expected属性告诉JUnit，只有在抛出指定类型的异常时，才认为这个测试通过。如果抛出的是另一种类型的异常，或者没有抛出异常，JUnit 就会让这个测试失败。

使用标注能极大地简化有关错误条件测试的编写。如果使用的编程语言或框架不具备与之等效的功能，那么就需要自己编写等效的行为。尽管这里的模式很简单，但在首次将其实现时通常会遗漏一些细节。代码清单11-3演示了，在没有直接框架支持的情况下，如何编写正确的异常测试。

代码清单11-3 针对代码清单11-1的带有手工编写的异常处理的JUnit测试

```
public void testDoSomething_Exception() {
   // Create fixture
   try {
      sut.doSomething();
      fail();
   } catch(ServiceFailureException expected) {
```

```
            // Expected behavior
        }
    }
```

如果抛出的是除ServiceFailureException之外的其他异常，那么我们仍然要依赖框架行为来让测试失败。如果需要考虑这种情况，那么可以使用另一个带有fail()断言的catch子句。

这里所包含的catch子句故意忽略了我们期望的异常。这个子句中的注释清楚地表达了这是那个期望的结果。这条注释也可满足许多静态检查器的检查条件，使得它们不会抱怨存在一个空代码块，而这种做法笔者认为仅在测试的上下文中才可接受。

缺乏经验的测试者经常会遗漏的一个关键点是：在try语句块中使用fail()断言。如果那个方法偶然运行成功且没有抛出异常，那么会发生什么？控制将经过这行代码，离开 try语句块，经过 catch语句块，然后离开这个方法而不再遇到一个断言，从而导致一个运行通过的测试。那条额外的断言语句能防止一个假阳性的结果。

11.3 验证异常消息[43]

如果异常处理的代码不像代码清单11-1所示的那样简单和整洁，那么该怎么做？或许希望在那个依赖RemoteException被抛出的异常里包含一条消息。可能正在编写一个面向对象的外覆来将一个过程式的API 给包裹起来，并希望抛出一个不同的异常，或包含一个随错误值而不同的消息。代码清单11-4给出了前一种情况的一个简单的例子。

代码清单11-4 带有多个同构异常路径的异常处理。另一种实现方法是从ServiceFai lureExc eption开始创建一个有关异常的层级结构，来区分使用前面章节中的技术的各种情况

```
public class SomeClass {
```

```
    public static final String FAILED_TO_CONNECT =
        "Failed to connect";
    public static final String SERVER_FAILURE =
        "Server failure";
    public void doSomething() throws ServiceFailureException {
      try {
        makeNetworkRequest();
      } catch (ConnectException exp) {
        throw new ServiceFailureException(FAILED_TO_CONNECT,
          exp);
      } catch (ServerException exp) {
        throw new ServiceFailureException(SERVER_FAILURE,exp);
      }
    }
}
```

ConnectException 和 ServerException 与许多其他异常都派生自RemoteException。这两个异常就足够能说明问题了。如何才能区分上述方法会抛出的ServiceFailureException 的（与这两个异常相对应的）两种变体呢？利用前面有关字符串处理的讨论，我们已经为答案准备了代码。区分上述有关结果异常的两件事是消息和因果异常（causal exception）。目前，我们先把重点放在消息上。

JUnit 框架只针对所期望的异常类提供直接的支持，所以如果想验证“消息字符串”这样的细节，就需要使用手工的异常验证。代码清单11-5显示了如何做到这一点。

代码清单11-5 在JUnit中验证一个异常消息

```
public void testDoSomething_ConnectException() {
  // Create fixture to inject ConnectException
```

```
    try {
        sut.doSomething();
        fail();
    } catch(ServiceFailureException exp) {
        assertEquals(SomeClass.FAILED_TO_CONNECT,
                exp.getMessage());
    }
}
```

这是如此常见的一个模式，以至于像 TestNG 这样的框架为基于消息的验证提供了更加显式的支持。代码清单11-6显示了使用TestNG如何编写这样的测试。

代码清单11-6 用TestNG验证一个异常消息

```
@Test(expectedExceptions = ServiceFailureException.class,
    expectedExceptionsMessageRegExp =
        SomeClass.FAILED_TO_CONNECT)
public void testDoSomething_ConnectException() {
    // Create fixture to inject ConnectException
    sut.doSomething();
}
```

这造就了一个更加整洁的测试。TestNG 不仅要求抛出一个 ServiceFailureException异常，而且仅当所指定的正则表达式与消息匹配时，测试才会通过。在这个例子中，正则表达式仅仅是一个字符串常量。依托对完整的正则表达式的功能的支持，就能增强这个特性的能力。除了7.2 节中所指出的那些限制之外，Java又在用于标注参数的表达式的种类上增加了更多的限制。所有的标注参数必须是能在编译时加以解析的常量。

11.4 验证异常有效载荷[44]

现在可以概括一下验证异常消息的想法。上一节做了一个深思熟虑的决定，暂时忽略异常的原因（cause），而考虑了两个属性来帮助区分抛出异常的原因。其实原本可以使用异常的原因，来作为验证的替代手段，或者甚至来补充验证。实际上，message 和 cause 是 Throwable 类中仅有的用户可以设置的两个属性[45]，而Throwable类是Java语言中所有异常类[46]的基类。Throwable非常近似于其他编程语言和框架中异常层级结构的基类。例如，C#有具备InnerException和Message属性的System.Exception类。C++只有what()这个与Java的message等效的访问器——但只能被类本身所定义——而且没有异常原因。

无论是什么编程语言，异常都是可扩展的。它们可以具有超出系统的基类所定义的属性之外的一些属性。虽然那些属性或许没有众所周知的接口，但是那些知道特定异常的代码能够合理地知道这些属性。在本章后面将讨论扩展异常的理由和扩展异常的用法。现在，只是简单地假设它们是因为一个超出可测试性的理由而存在的，并以测试的目的来考察它们。

在代码清单11-5所示的测试中，可以通过在catch语句块中添加语句

assertTrue(exp.getCause() instanceof ConnectException);

来验证异常的原因，或者至少能验证这个原因的类型。现在已经验证异常的原因是所期望的类型。当通过类型来进行验证时，这本身就具有与本章前一部分所讨论的内容相一致的缺点。然而，将其放置在有关基本异常的类型和消息的验证的顶部，就能更有力地验证该基本异常。通过下面的做法还能更进一步，即将所有有关异常验证的技术也都运用到因果异常之上。实际上，可以顺着这个链条一直往下深入下去，但是应该记住，验证得越深入，潜在地将测试与实现细节相耦合的程度就越大，这是不恰当的。

Java为异常定义了cause（原因）属性。那我们自己发明的属性该怎么办呢？设想一个API，它在智能烤箱的嵌入式控制器中，烤箱带有精致的触屏界面和人性化的互动展示。这种烤箱的一个特性可能是，允许用户通过键盘而不是旋钮来输入期望的温度。旋钮有内置在其刻度中的温度极限和旋钮旋转范围的概念，然而数字显示可能只有与之相似的基于所能显示的数字个数的固有极限。考虑代码清单11-7所示的烤箱控制器软件的子集。

代码清单11-7 一个假想的用于智能烤箱的烤箱控制器API的子集

```
public class OvenController {
   private final int ratedTemperature[47];
   public OvenController(int ratedTemperature) {
      this.ratedTemperature = ratedTemperature;
   }
   public void setPreHeatTemperature(int temperature)
         throws IllegalArgumentException {
      if (temperature ＞ ratedTemperature) {
         throw new RequestedTemperatureTooHighException(
               ratedTemperature,temperature);
      }
      // Set the preheat temperature
   }
}
```

从这段代码中可以推断，RequestedTemperatureTooHighException派生自IllegalArgumentException，因为setPreHeatTemperature()方法声明了后者[48]，但抛出了前者。但是为何要提供两个温度来作为参数呢？代码清单11-8显示了这个异常的部分实现。

代码清单11-8 RequestedTemperatureTooHighException的部分实现

```
public class RequestedTemperatureTooHighException
   extends IllegalArgumentException {
   private final int ratedTemperature;
   private final int requestedTemperature;
   public RequestedTemperatureTooHighException(
         int ratedTemperature,int requestedTemperature) {
      this.ratedTemperature = ratedTemperature;
      this.requestedTemperature = requestedTemperature;
   }
   public int getRatedTemperature() {
      return ratedTemperature;
   }
   public int getRequestedTemperature() {
      return requestedTemperature;
   }
   ...
}
```

该异常现在带有关于其被抛出的原因的详细信息，该信息与捕获该特定异常类型的那段代码高度相关。这个信息也给了我们一个手段，能更加彻底地验证抛出异常的软件的行为。代码清单11-9包含了针对OvenController的这样一个测试。

代码清单11-9 针对代码清单11-7的OvenController的一个更加严格的测试

```
@Test
public void testSetPreHeatTemperature() {
   int ratedTemperature = 600;
   int requestedTemperature = 750;
```

```
    OvenController sut = new OvenController(ratedTemperature);
    try {
       sut.setPreHeatTemperature(requestedTemperature);
       fail();
    } catch(RequestedTemperatureTooHighException exp) {
       assertEquals(exp.getRatedTemperature(),ratedTemperature);
       assertEquals(exp.getRequestedTemperature(),
             requestedTemperature);
    }
}
```

将输入值与错误绑在一起——就如同异常所表达的那样——就能得到测试的输入与输出之间更高级别的耦合，从而得到下面这些更高级别的信心，即测试验证了软件，且执行了目标中所设定的特定的异常路径。异常有效载荷能够包含任何数据类型，从而带来丰富的验证机会。在本章的结尾处将会看到，可测试性的提高也能改善软件的其他方面。

11.5 验证异常实例

到目前为止，每一个基于异常的技术，基本上都使用下面的方法来试图推断某个异常就是我们认为应该抛出的异常，即检查该异常的如下特征（characteristic）：类型、消息、原因和扩展属性。我们已经用增加细节级别的方式，加大了已经正确地识别了异常的信心。或许有人会想：“为什么不直接注入异常本身呢？为什么不直接用异常本身的身份来验证异常呢？”这就是本节要解决的问题。

不用说，身份本身就是用来验证身份的最强的方式。但是对于对象来说，特别是对于异常来说，可变性和上下文就很重要了。

可变性指的是一个对象被改变的能力。一些像 C++这样的编程语言

支持值传递（pass-by-value）的语义，这些语义作为参数传递时，会创建对象的副本。否则，引用就会被传递，并且就能通过这些引用来改变对象。

也许有人认为对于异常来说这不是一个问题，因为它们通常是不可变的。在创建之后它们的属性不应该可变，或者至少不应该被改变。另外至少在传统意义上来说，很少会把异常作为参数来传递。catch语句的语义通常将异常作为引用来传递，与把参数作为引用来传递类似。

然而，对于我们的目的来说，异常的可变性最重要的方面关系到上下文的问题。在许多编程语言中，运行时环境在创建异常时，会自动插入大量的上下文。这个上下文就把异常和其他对象显著地区分开来。Java包含堆栈跟踪信息。C#也包含一些有关上下文的其他属性，如应用程序的名称和抛出异常的方法[49]。

不同于像 int 或 String 这样的基本数据类型的常量，为了达到注入的目的而预分配一个异常，就会不可改变地令其附带上对象创建处的上下文，而不是抛出异常处的上下文。不建议使用这个上下文来验证异常。例如检查栈跟踪信息会将异常验证很紧密地耦合到实现上。做一个小小的Extract Method[REF]重构就能导致脆弱的测试（Fragile Test）[xTP]。然而，在很多情况下，测试框架会报告作为故障报告一部分的栈跟踪信息，显示针对错误诊断的引发误导和模棱两可的上下文。一个共享的异常常量会为多个异常抛出点显示同样的栈跟踪信息：这样会看到一个实际上并没有指向异常抛出位置的栈跟踪信息。此外，当有多个测试运行失败时，一个共享的异常常量会为每一个异常抛出点显示同样的栈跟踪信息。

在已经考虑了不使用预分配的异常的种种理由后，让我们简要地看一个可以说是合理的例子。代码清单11-10显示了前面的OvenController稍加重构后的版本。

代码清单11-10 OvenController的一个重构后的版本

```
public class OvenController {
   private final int ratedTemperature;
   public OvenController(int ratedTemperature) {
      this.ratedTemperature = ratedTemperature;
   }
   public void setPreHeatTemperature(int temperature)
         throws IllegalArgumentException {
      validateRequestedTemperature(temperature);
      // Set the preheat temperature
   }
   protected void validateRequestedTemperature(int temperature)
         throws IllegalArgumentException {
      if (temperature > ratedTemperature) {
         throw new RequestedTemperatureTooHighException(
               ratedTemperature,temperature);
      }
   }
}
```

上述代码将温度验证逻辑重构到一个单独的protected方法中，从而允许我们通过覆写来注入一个异常。代码清单 11-11 显示了针对重构后的 OvenController的测试。

代码清单11-11 使用预分配的异常来测试重构后的OvenController

```
@Test
public void testPreHeatTemperature_TooHigh() {
   int ratedTemperature = 600;
   int requestedTemperature = 750;
   IllegalArgumentException expectedException = new
```

```
            RequestedTemperatureTooHighException(ratedTemperature,
                requestedTemperature);
        OvenController sut = new FailingOvenController(
                ratedTemperature,expectedException);
        try {
            sut.setPreHeatTemperature(requestedTemperature);
            fail();
        } catch(RequestedTemperatureTooHighException exp) {
            assertSame(exp,expectedException);
        }
    }
    private class FailingOvenController extends OvenController {
        private final IllegalArgumentException toThrow;
        public FailingOvenController(int ratedTemperature,
                IllegalArgumentException toThrow) {
            super(ratedTemperature);
            this.toThrow = toThrow;
        }
        @Override
        protected void validateRequestedTemperature(int temperature)
                throws IllegalArgumentException {
            throw toThrow;
        }
    }
```

上述代码使用了预分配的异常来驱动结果的产生。请注意，以这种方式使用它时，需要假定validateRequestedTemperature()方法已被单独测试。像上面这样覆写该方法能够防止原方法被调用。现在通过创建一个

helper类，以一个不共享的对象的方式来定义一个离抛出点非常近的异常，就已经避免了使用带有极易引发误导的栈跟踪信息的共享异常的陷阱。

这种技术的一个更加引人注目的使用，是注入一个期望作为外覆异常的原因的异常。

11.6 有关异常设计的思考

在探讨异常设计的细节之前，先来考虑一下本章的那些简单例子所没有表达出来的一个方面。异常处理的整体策略，应该能对设计和最低层面的异常处理进行指导。至少有关异常处理架构的一个餐巾纸草图（napkin sketch），就能在将来很长的一段时间里解决那些痛苦。下面是一些需要考虑的问题。

是在诸如UI、API和服务接口（笔者的建议）这样的最顶部的边界层处理异常，还是在低一些的层面以一种更加临时的方式来处理呢？

在像Java这样对异常进行区分的编程语言中，是以checked异常的形式来实现异常，从而将其作为方法签名的一部分加以声明，还是以unchecked 异常的形式来处理呢？

是否允许吞掉异常，从而防止它们的传播？如果是这样的话，该在哪些条件下允许这样做呢？

当较低层面的语义变得不大适用于较高层面的抽象时，转换或外覆异常的策略是什么？

是否需要以不同的方式来处理任何异常，例如需要在 Java 中处理InterruptedException来正常地处理线程生命周期？

随着错误处理策略的不断演进，对这些和其他一些问题的回答，将能在较低层面指导异常的设计，并能将混乱、不一致性和返工降到最低。

本章讨论了异常设计的各个方面，让这些异常和使用它们的代码更加可测。另外还发现或提示了一些衍生自同样的这些设计考量的额外好处。让我们从测试那些实现模式开始，做一个简短的环游，总结那些能被应用的设计元素，从而让异常在整体上更加强大。一个综合这些设计元素的例子如代码清单11-12所示。

从一个应用程序特定的基本异常（base exception）来衍生出各个异常。针对该应用程序的所有异常都能被捕获，而且不会捕获到系统异常。

创建一个明确地与基本异常相关的层级结构。那些表达相似语义概念的异常（如边界错误）或者共享共同操作上下文的异常（如 I/O 操作），应该被分组到一个统一的继承层级结构中。

使用上下文特定的异常类型。一个异常表达了一个有关故障状况（failure condition）的消息。异常的名字应该用人能读懂的方式捕获故障的含义，而且异常类型的特殊性应该能让故障在编程时可隔离和可识别。

使用属性来将异常的意图进行参数化。许多异常都对应于能被一个或多个数据值进行特征化的一些情景。当必要时，以原始属性的方式捕获这些值，能帮助我们在编程时对故障进行响应。

异常属性在最大的实用程度上应该仅可通过构造器来设置。错误的特征不应该需要改变。增强的上下文应该衍生自更深封装的异常。

以字面格式字符串常量来代表一个异常的消息。如果没有必要存在属性，那么一个简单的字面字符串就足够了。如果存在一些属性的话，那么格式字符串应该能格式化这些属性。这有利于进行涉及复杂字符串和数据内容的场景的精确验证。

如果需要国际化和经本地化处理后的错误消息，那么异常消息的常量应该是针对格式化字符串的用于查找的键值。要尽量晚地绑定语言环境，以本地化上下文的错误信息。

代码清单11-12 根据推荐的原则设计的异常

```
public class MyApplicationException
        extends Exception {
    public MyApplicationException() {
        super();
    }
    public MyApplicationException(Throwable cause) {
        super(cause);
    }
    public String formatMessage(Locale locale) {
        throw new UnimplementedException();
    }
}
public class MyCategoryException
        extends MyApplicationException {
    public MyCategoryException() {
        super();
    }
    public MyCategoryException(Throwable cause) {
        super(cause);
    }
}
public ValueTooHighException
        extends MyCategoryException {
    public static final String MESSAGE_KEY = “value.too.high”;
    private final int value;
    private final int threshold;
```

```
    public ValueTooHighException(int value,int threshold) {
        super();
        this.value = value;
        this.threshold = threshold;
    }
    public ValueTooHighException(int value,int threshold,
            Throwable cause) {
        super(cause);
        this.value = value;
        this.threshold = threshold;
    }
    @Override
    public String formatMessage(Locale locale) {
        String formatString = Dictionary.lookup(MESSAGE_KEY);
        return String.format(locale,
                formatString,value,threshold);
    }
    public String getLocalizedMessage() {
        return formatMessage(Locale.getDefault());
    }
    public String getMessage() {
        return formatMessage(Locale.ENGLISH);
    }
}
# The English dictionary entry for the message key
value.too.high = \
    The value %1$d is higher than the threshold %2$d.
```

第12章 利用现有接缝

Michael Feathers 推出了接缝[WEwLC]这个概念，来作为一个能将代码引入测试的框架。代码中的接缝，能给我们提供控制那段代码和在测试的上下文中执行那段代码的机会。任何能够执行、覆写、注入或控制那段代码的地方都可以是一个接缝。接缝到处都有。有时，接缝很明显，以至于我们甚至不认为它们是接缝。有时，接缝很微妙或者很神秘，以至于我们看不到它们。还有时，为了能够运用那些在测试时所需的控制，需要引入它们。

前面已经看过许多能将代码引入测试的技术，现在让我们暂时退后一下，看一幅更大的画面。本章将针对Feathers的对象接缝类别，大致总结出一个接缝类型的排序，但也有一些针对非面向对象的编程模型的扩展。这些接缝出现的先后次序，就是笔者认为，在把代码引入测试时，应该考虑的使用接缝的顺序。这个顺序考虑了诸如易用性和该类别通常会牵涉的与其他接口、类或框架的内在耦合这些因素。

就像任何将启发式方法进行排序所进行的尝试一样，顺序不应被视作一套严格的规则，而是可以作为能在有理由的情况下进行覆写的一个建议。此外，本章针对现有接缝的考虑顺序提出了建议。为了达到可测试性而改变一个现有设计，需要一套更复杂的推理，而不是一个简单的层级结构。

12.1 直接调用

就像人际交往那样，直接的方法效果最好，但经常被绕过或忽视。

希望你不要担心与代码直接沟通就会冒犯这些代码。有时，笔者看到甚至都有可用的直接方法，人们仍然制定缜密的测试方案，让我很惊讶。

12.1.1 接口

对于那些在语法上支持接口的编程语言，已发布的接口通常是最好的起点。除了在通常情况下要被记入文档之外，它们趋向于和其他模块之间具有最小的耦合性。随着时间的推移，它们也具有高度的稳定性，从而将任何耦合所无法避免的影响降到最低。

在一个面向对象的上下文中，你会使用API，这些API是通过类似下面这样的特性发布的：Java接口、C#接口或在C++类的头文件中定义的接口。过程式或函数式的编程语言可能会使用函数原型和签名声明。无论如何，使用已发布的API规格，甚至当测试实现时，也能有助于将对软件的使用限制在规格中最稳定的部分上。甚至一个未被发布的内部使用的接口，也趋向于比实现更稳定。

需要注意的是，在一个过程式或函数式的上下文中，像函数原型和签名这样的构造，也扮演了与面向对象接口同样的角色。与Perl或JavaScript这样的动态语言中的那些更加灵活的声明相比，像C这样严格定义类型的语言提供了更强的绑定和保证，但在强制性上比较宽松的接口的广泛使用，还是针对其在测试中的使用，提供了相对强大的有关稳定性和兼容性的保证。例如，没有什么需求来查询jQuery事件对象的内容，或者想知道它将被如何传递到jQuery事件处理器。

无论正在使用哪种编程模型，在验证实现时，都应该倾向于在测试中尽最大可能地使用接口定义。这样做有双重目的：将测试的耦合最小化；有助于验证代码符合了其所实现的接口的协议。

12.1.2 实现

几乎可以肯定，无法通过一个实现的接口来测试全部实现。除了最简单的实现之外，所有的实现都需要数量可观的代码来将接口的行为实

现出来。在其他情况下，一个接口仅代表一个实现类所起的一个单独的作用。有时候，一个类会用这种方式实现多个接口。

在函数式或过程式的编程语言中，不大可能遇到被测代码只有单独一个目的的现象。然而，如果用基于特性的而不是基于函数或方法的方式来进行测试，那么就会发现需要调用多个函数来测试该特性，而并非所有的这些函数都属于一个已发布的接口。

直接使用实现和实现特定（implementation-specific）的特性还是一个强大的测试代码的方式。这种方式提供了对行为的直接验证，如果不做一些额外的依赖或脆弱的间接层，将无法通过已发布的接口访问这些行为。

12.2 依赖注入

依赖注入[50]已经从一个手工纺制（hand-spun）的、有点不起眼的应用装配方法，成长为一个主流的、普遍的方法。在用Java世界中的Spring框架（见代码清单12-1）和EJB 3.0（见代码清单12-2）规范、.NET中的Windsor[51]容器和许多编程语言中的其他组件，编写实现时，针对依赖注入和控制倒置的框架的使用已司空见惯，尽管这些并不总是很好理解。这些框架的流行和其背后的原则，使得我们在测试上使用这种模式的可能性非常大。

有三种被广泛接受的依赖注入形式，Martin Fowler详细描述过它们，还有一些不大知名的形式，如Yan框架[52]所提供的那些。构造器和setter的注入（两者示于代码清单12-3中）盛行于主流的Java框架中，而且两者都得到了很好的支持。

代码清单12-1 一个服务依赖于另外两个服务的Spring框架配置。一个服务使用构造器注入方法来注入，另一个使用setter注入方法。请注意“¬”表示出于格式化的目的而人为添加的一个换行

```
＜?xml version=”1.0” encoding=”UTF-8”?＞
＜beans xmlns=”http://www.springframework.org/schema/beans”
        xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
        xsi:schemaLocation=
          “http://www.springframework.org/schema/beans ¬
          http://www.springframework.org/schema/beans/¬
            spring-beans-2.0.xsd”＞
  ＜bean id=”userServiceBean” class=”...”/＞
  ＜bean id=”productServiceBean” class=”...”/＞
  ＜bean id=”shoppingCartService” class=”...”＞
    ＜constructor-arg ref=”userServiceBean”/＞
    ＜property name=”productService” ref=”productServiceBean”/＞
  ＜/bean＞
＜/beans＞
```

代码清单12-2 使用JSR 330标注编写的与代码清单12-1等效的配置

```
public class ShoppingCartServiceImpl
    implements ShoppingCartService {
  private UserService userService;
  @Inject
  private ProductService productService;
  @Inject
  public ShoppingCartServiceImpl(UserService userService) {
    this.userService = userService;
  }
}
```

代码清单12-3　直接使用EasyMock来mock各个协作者，从而测试代码清单12-2中所示的那个类中的一个方法

```
import static org.easymock.EasyMock;
public class ShoppingCartServiceImplTest {
   @Test
   public void createCartWithProduct() {
      UserService userService =
         EasyMock.createMock(UserService.class);
      ProductService productService =
         EasyMock.createMock(ProductService.class);
      // Configure mocks
      EasyMock.replay(userService,productService);
      ShoppingCartService sut =
         new ShoppingCartServiceImpl(userService);
      sut.setProductService(productService);
      // Execute and assert something about the cart
   }
}
```

构造器注入允许依赖以参数的形式被传递到一个构造器中。使用这种形式，可以在一个地方声明针对一个对象的所有的依赖，但不支持循环依赖。

setter注入依赖于默认的构造器，使用这种注入的对象至少有一个常规的setter来注入每一个依赖。实例的创建和注入是分开进行的，允许框架更加灵活地解析这些依赖。对于那些支持反射的编程语言来说，setter方法并没有被严格要求，尽管将其忽略需要测试提供对依赖注入的支持，如代码清单12-4所示。

代码清单12-4 Spring框架对基于文件的配置提供的支持，有助于在测试时注入替代的实现

```
package com.vance.qualitycode;
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@Configuration({“test-context.xml”})
public class ContextTest {
...
}
```

这两种形式的注入能被混合到同一个初始化中，但针对我们的目的的重要特性，是要有显式的参数列表来用于将那些协作者注入到测试中。那些允许依赖注入的形式也令我们可以直接测试对象。mock经常用于被注入的组件，简化测试替身的创建过程。

这些框架也为使用注入基础设施来注入协作者提供了一些便利，从而简化那些具有很多依赖的对象的构造。特别是Spring框架能很好地支持这一点[53]。它支持从文件中加载应用的上下文，如代码清单12-4所示。另外Spring 3.1版也支持配置类（见代码清单12-5）。这两种情况都允许测试上下文中的替代实现的注入。

代码清单12-5 Spring框架的3.1版支持配置类，允许在测试代码自身中定义被注入的实例[54]

```
package com.example;
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from the static inner
// ContextConfiguration class
@ContextConfiguration(
   loader=AnnotationConfigContextLoader.class)
public class OrderServiceTest {
   @Configuration
   static class ContextConfiguration {
      // This bean will be injected into
      // the OrderServiceTest class
```

```
    @Bean
    public OrderService orderService() {
        OrderService orderService = new OrderServiceImpl();
        // Set properties,etc.
        return orderService;
    }
}
@Autowired
private OrderService orderService;
@Test
public void testOrderService() {
    // Test the orderService
}
}
```

12.3 回调、观察者、监听者和通告者

任何需要对异步事件进行低延迟响应的系统，都大量使用了某种形式的回调、观察者或通告者。可以在用户界面的工具包和分布式系统中找到它们。在用户界面的工具包中，经常将其称为事件监听者。回调是一个通用机制，当某些众所周知的事件发生时，回调允许我们为系统提供其要调用的代码。回调有各种各样的形式，其中一些创造性地使用了一些语言特定的特性。像C、Perl、JavaScript和其他一些具有过程式或函数式风格的编程语言，都会使用函数指针（见代码清单12-6）或函数引用（见代码清单 12-7）。面向对象的编程语言经常通过那些有特殊目的的接口的实现来实现回调，如代码清单12-8所示。具有反射或动态调用功能的编程语言，能通过那些代表类名、方法名或函数名字的字符串

来提供回调功能。代码清单 12-9 展示了一个使用Spring Scheduling框架的例子。在某些情况下，一些操作系统会提供独立于语言的类似反射的机制，如POSIX的动态加载API（见代码清单12-10）。C++的指向成员的指针（pointer-to-member）函数的语法（见代码清单 12-11）支持一种类型受限（type-constrained）的回调。C++的函数调用操作符的重载允许使用函数对象或函子（functor）[55]，来作为回调的形式，如代码清单12-12所示。

代码清单12-6 接口的C语言标准库[56]需要函数指针的例子

```
#include ＜stdlib.h＞
void qsort(void *base,size_t nmemb,size_t size,
   int (*compar)(const void *,const void *));
```

代码清单12-7　　回调函数在JavaScript中很常见，例如在这里jQuery.each()的使用中所展示的那样

```
$(‘.menu ＞ .menuitem’).each(function(index,element) {
   console.log(‘Item ‘ + index + ‘ says ‘ + $(element).text());
});
```

代码清单12-8　　Java的Swing用户界面框架定义了大量的监听者接口，如这里针对一个简单的按钮反应所使用的ActionListener

```
public class HiThere implements ActionListener {
   ...
   public void init() {
      button.addActionListener(this);
   }
   ...
   public void actionPerformed(ActionEvent e) {
      textArea.append(“Hello,World!”);
   }
```

```
}
```

代码清单12-9　　针对一个基于反射的scheduling回调的Spring Scheduling配置，该回调由字符串定义，详细描述了将被调用的类和方法。 请注意“¬”表示出于格式化的目的而人为添加的一个换行

```
＜bean id=”runMeJob”
    class=”org.springframework.scheduling.quartz¬
      .MethodInvokingJobDetailFactoryBean”＞
  ＜property name=”targetObject” ref=”runMeTask” /＞
  ＜property name=”targetMethod” value=”printMe” /＞
＜/bean＞
```

代码清单12-10 使用POSIX的dlopen(3) API[57]以回调的方式从一个共享库中加载和调用一个函数的例子

```
#include ＜stdio.h＞
#include ＜stdlib.h＞
#include ＜dlfcn.h＞
/* For NAN macro */
#define _GNU_SOURCE
#include ＜math.h＞
double
call_double_return_double(char *libname,
    char *funcname,double arg)
{
  void *handle;
  double (*func)(double);
  char *error;
  double result;
  handle = dlopen(libname,RTLD_LAZY);
```

```
    if (NULL == handle) {
        fprintf(stderr,“%s\n”,dlerror());
        return NAN;
    }
    dlerror(); /* Clear any existing error */
    *(void **) (&func) = dlsym(handle,funcname);
    error = dlerror();
    if (error != NULL) {
        fprintf(stderr,“%s\n”,error);
        return NAN;
    }
    result = (*func)(arg));
    dlclose(handle);
    return result;
}
```

代码清单12-11　使用C++的指向成员的指针函数来实现一个带有签名的回调

```
typedef double (Callback::*CallbackMember)(double);
call_double_returns_double(Callback &cb,
    CallbackMember cbm,double arg)
{
    return cb.*cbm(arg);
}
```

代码清单12-12　基本的C++函子的基类的例子，该函子的参数为字符串，没有返回值。使用模板和指向成员函数的指针，就能在C++中创建一些非常复杂的函子的泛化（generalization）。使用boost::function和boost::bind的组合，Boost库支持函子的创建，这些已被纳入C++11标准

的＜functional＞头文件中[58]

```
class AbstractFunctor
{
public:
   virtual void operator()(const char *string)= 0;
}
```

回调的具体使用，已经通过代表它们特殊目的的更加具体的名字，而变得众所周知。观察者（Observer）模式[DP]用一种专门的方式使用了回调，当一个特定的状态或条件改变时，就能立即进行通信（communicate）。另一种特殊形式的回调是通知者（notifier），它允许客户端代码在一些特定的事件中注册兴趣，并经常作为日志记录或监控策略的一部分。

所有的回调都有共同的目的，即将各种条件同步通信给感兴趣的代码。这些作为通信主题（subject）内容的条件经常被封装起来。出于测试的目的，甚至应该考虑将一个回调的调用过程作为一个需要验证的特性。

然而，代码行为的这个特性也代表了一个可利用的接缝。回调通常接受有关过去发生过的变化或事件的数据。回调也发生在使回调可用的系统的处理过程中，其被充分理解的时候。通常，在没有其他直接的手段可以使用的情况下，可以使用回调来验证系统对数据进行了正确的处理和传递。可以针对行为验证使用回调的mock实现。甚至可以使用回调来对测试和线程执行施加控制，就像将会在第13章所看到的那样。

12.4 注册表

注册表（registry）是查找库，它允许特定的值或实现和一个查找关键字（通常为一种类型的目录路径）相关联，因此可以独立于不同的组

件或进程来对特定的值或实现进行查找。众所周知的例子包括：LDAP、Windows注册表和微软的ActiveDirectory、Java的JNDI及其各种上下文接口（如ServletContext和InitialContext）、Java的RMI注册表，甚至还有像用于Tomcat或Glassfish的运行时配置文件。在一定程度上，甚至连令人肃然起敬的DNS和几乎无所不在的环境变量都可以用作注册表。

注册表通常允许其数据通过层级结构组织起来。像 LDAP 这样的查找库，允许我们将一定程度的语义含义分配到该层级结构的各个层次上，如组织单元（Organizational Unit，OU）。较旧的注册表只允许对值进行存储，原先是以字符串的形式，后来是以具有类型的数据的形式。在那些系统中，应用程序会使用这些值来构造资源定位器（resource locator）。新一些的系统还允许存储可执行的引用，如对象和远程调用的引用，因此这些被存储的资源能被直接调用，而不会产生更多的开销。

注册表的实用性来自于能够动态地修改它们。在存储可调用的引用时，可以将指向一个基于网络服务的一个远程引用，替换为测试创建的一个本地对象引用[59]。如果用这些值来生成资源定位器，就可以替换下面这样的值，即引用到一个在测试控制下的资源，或者至少是已知它适合于参与测试的资源。注册表造就了有益的接缝，因为它们基于双方已知的信息，提供了依赖注入的一种间接形式：查找键。

另一种将注册表用作接缝的方式是简单地替换注册表实现。根据上下文的不同，一些实现做起来会比其他实现要更容易一些。这里关键的区别通常在于如何能自引导（bootstrap）到注册表。如果注册表是一个找到全局资源的方式，而注册表本身也是一个全局资源，那么我们如何才能找到注册表呢？在注册表能通过配置（如J2EE容器）来定义的环境中，就能很容易地注入新注册表的实现，如代码清单12-13和代码清单12-14 所示。那些灵活性较弱的环境会通过静态方法来访问注册表，这

些静态方法是硬连接的（hardwired）且不能被覆写。

代码清单12-13 一个正获取其ServletContext的基本的servlet

```
public class MyServlet extends HttpServlet {
   public void service(HttpServletRequest request,
                       HttpServletResponse response)
      throws IOException,ServletException {
   ServletContext context = getServletContext();
   }
}
```

代码清单12-14 在一个Java的servlet中使用EasyMock来对ServletContext进行mock。这种方法能安全地将调用截获到代表servlet的上下文的注册表中

```
import static org.easymock.EasyMock.*;
public class MyServletTest {
   public void testService() throws Exception {
      ServletConfig servletConfig =
         createMock(ServletConfig.class);
      ServletContext servletContext =
         createMock(ServletContext.class);
      expect(servletConfig.getServletContext())
         .andReturn(servletContext).anyTimes();
      replay(servletContext,servletConfig);
      MyServlet sut = new MyServlet();
      sut.init(servletConfig);
      // Prepare a request and a response also
      ...
      sut.service(request,response);
```

```
            verify(servletContext,servletConfig);
        }
    }
```

当为了测试而把注册表用作接缝时，同样这个允许或阻止注册表替换的特性也会惹来麻烦。甚至可配置的注册表通常情况下也仅针对整个JVM或运行时环境是可配置的。这引入的挑战和测试中单例的情况是相同的。至少在注册表被共享的上下文中，一个行为异常的测试有可能会留下一个替代用的注册表项，该表项会影响未来的测试运行，而且，如果这些测试都在同时修改同一些注册表项，那么这些测试的并发运行也会相互干扰。

12.5 工厂

工厂基于模板的或预先确定的初始化条件按需生成了一些实体（entity）。对于这一作用，通常会想到“四巨头”的抽象工厂[DP]模式。但还存在其他的工厂模式，包括Factory Method和Builder[DP]。

每一种方法和实现都有能满足其目标使用情况的鲜明特征。下面总结了在测试中需要考虑的关键特性。

单例性：就其本质来说，工厂是否是一个将在两个测试之间被共享的单例？

实体初始化的灵活性：针对必要的测试变化情况，实现是否有足够的机制来初始化所生成的实体？

注册管理：工厂是否允许动态地注册和注销所生成的实体类型？

这些特性中的种种变化确定了，测试时，一个特定工厂的实现被期望用作接缝的程度。

抽象工厂模式的许多实现也是单例模式的种种实例。其中的理由是，实体生成的标准化的源应该是这些实体唯一的源。对于许多应用程

序来说，这满足了使用需求，但是却具有所有单例所面临的同样的测试挑战。主要的挑战就是，测试将共享实例，并且会遇到相互干扰的风险。

工厂方法也会苦于有这种限制。根据其定义的本质来说，它们就是单例。通常情况下只存在一个特定方法的一个实例，因为它们通常是静态的，并且驱动它们行为的数据源也需要是单一的[60]。

工厂往往只提供有限的能力来初始化它们所创建的那些实体。它们角色的宽泛本质也支持了这一点。一个能生成各种实体的机制，需要将有时不可接受的约束和需求加到那些实体上，以一致地初始化它们。此外，根据为达到需求目的而对需求进行限制从而创建实体的需要，促使对定制化的程度进行限制，而该定制化能够通过工厂来达到。虽然被创建的实体的受限初始化过程会与应用程序的设计目标吻合得很好，但这会限制测试工厂行为的能力。

许多应用程序使用模板和实例来静态地配置它们的工厂，这些模板和实例对于应用程序的运行时行为是必要的。经静态配置的工厂，会紧紧地将代码耦合到类型和那些创建出来的实体的实现上，从而导致不希望有的测试的耦合。支持被创建实体的动态注册的工厂会更容易测试。开始时它们在测试代码中未被初始化，这样就能使用那些仅在测试中定义的实体来配置它们，从而降低测试库的整体耦合。

通过突出的工厂接缝的概述，可以猜到，只有一组很窄的工厂特征会有力地支持可测试性。为了能最好地支持可测试性，一个工厂实现不应是一个单例，而应支持灵活的初始化和实体注册。持续这样做，就能构成一个稀有的特性组合。这些特性不再频繁地聚在一起，会减少工厂成为理想测试接缝的可能性。这里有一些针对创建工厂来更好地支持可测试性的建议。

因为对可测试性的关切一般都与单例有关，所以应试图按照惯例而不是按照实现，以单例的形式实现工厂。顺便确保它们被构造得只存在

一个实例。许多依赖注入的框架都提供了按照惯例支持单例的机制。

用于测试的有用的工厂实现只需要支持灵活的初始化或动态注册。通过足够灵活的初始化，就能通过改变初始化参数来进行所希望的几乎任何实体的构建。通过动态注册，就能注册一些构建器或模版，来创建完全正确的实体。相比灵活的初始化，笔者个人更喜欢动态注册，其原因仅仅是因为有观点认为，前面所给出的那个有关灵活的初始化有时会违背工厂的目的。不过，当灵活的初始化支持创建更加复杂的实体网络时，那个观点就不是特别能站住脚的了。

当试图测试的组件使用工厂来创建协作者时，工厂会创造一个良好的接缝。修改生成实体的初始化结果或注册可替换的实现，这个能力，允许我们通过已改变的实现将行为注入到被测软件中。

当这看起来像是一个能影响正在测试的行为的整洁的方式时，成本也会接踵而来。如果只是简单地添加工厂来支持测试，这是热切的惯常使用mock的程序员常用的技法[61]，那么这其实也添加了代码和复杂性，其本身也需要被测试和维护。在成本面前，这样的举措应该有充分的理由。此外，工厂是典型的实现细节，它们只是达到目的的手段，而不是软件的需求。因此，把工厂用作接缝的做法，就把测试耦合到了实现上。另外，那些工厂所创建的协作者，经常也是实现细节，即使不是一个全面的实现，这些细节也至少把测试耦合到了接口或模板上。这两个耦合因素抑制了测试调理方案的可扩展性和可维护性。当注入行为时，与使用生产工厂生成的实现类相比，对工厂进行配置能产生更宽松的耦合，但与完全避免使用工厂相比会带来更高的耦合。

工厂是有用的测试接缝，特别是当使用mock或其他形式的测试替身来测试带有行为风格的代码时更是如此。应该抱着审慎的态度来对待工厂所带来的额外的耦合和复杂性，这也是笔者在可利用的接缝列表中，选择将其靠下放置的原因。

12.6 日志记录与最后一手的其他设施

还是能遇到一切合理的接缝都无法能帮助达到目的的时候的。此时一边用脑袋撞墙，一边在挫败中尖叫：“怎样才能测试这段代码？”想要测试这段代码，需要测试这段代码。对掌控这个系统的痴迷不会令人就这样放弃。这样的时刻会导致做出一些绝望中的决定。

首先，休息一下。透透气，散散步，打个盹儿，抽抽烟（如果平时有这习惯），看看电子邮件，读读好文章。总之，就是要做点事情，打破僵化状态，转移注意力，获得一些空间。如果做了所有这些事情之后，仍然需要去战胜挑战，那么请继续读下去，来看一看测试接缝这条绳子的尽头，如图12-1所示[62]。

图12-1 如果考虑使用本节中的技术，请仔细留意这幅图

希望这已经能足够把读者吓着了，从而能令读者在使用下面即将讨论的技术之前，仔细地考虑一下。这些技术是真实的、有用的。让我们首先讨论为何这些技术应被谨慎使用。

正如在第4章和第5章所看到的那样，我们希望通过实现来验证意图、尽量减少耦合、防止测试相互影响，并且避免使用那些会阻碍测试以并行方式可靠运行的技术。但我们下面将要讨论的这些接缝违背了所有这些原则。我们开发框架中的一些设施是为方便程序员而存在的。另

一些方法具有更广泛的适用性，但在这些方法是否能用在我们的代码中以及用在何处这些方面，其要求会更加宽松。

日志记录（logging）、监控，有时还有安全性，这三类设施都属于上述范畴。尽管有少数人会说要避免使用日志记录，但是几乎没人会说，要在何处放置日志记录语句，特别是那些仅支持调试的日志记录语句。

上述这些设施往往是可选的、可实现的。它们不会出现在需求中。它们的存在和处置基本上需要依靠程序员的判断，这是因为根据具体的使用情况，会产生很高程度的实现耦合。同时，它们的集中可配置和可使用的能力所发挥的作用，让它们具有了单例的本质，从而在串行或并行的执行中，会导致测试相互之间错误的交互。

日志记录是我们所讨论的设施中最常见的例子。代码清单12-15显示了使用log4j[63]的Java代码。在这种情况下，可以专门针对测试的目的而添加debug（调试）语句，以验证一个中间的计算结果。虽然大多数开发人员以字符串的形式在调试语句中使用message参数，但接口只要求它是一个Object，所以可以自由地传递intermediate。代码清单12-15 Java代码使用log4j作为一个显式添加的调试接缝

```
public class SUT {
  private static final Logger LOG =
    Logger.getLogger(SUT.class);
  public void testMe() {
    // Bunch of code
    ...
    Integer intermediate = computeSomething();
    // Important: Keep this for testing purposes!
    LOG.debug(intermediate);
    // Continue with computation
```

```
        ...
    }
}
```

怎样才能将日志记录用作一个接缝呢？大多数应用程序会通过一个 Java 属性文件或通过 XML 配置来对 log4j 进行配置。log4j 框架提供了大量有用的追加器（appender）——即那些在消息流上添加消息的软件组件，足以满足大多数应用程序的需要。很少有应用程序需要编写自己的追加器或用编程的方式来操作它们。然而，log4j框架有一个完整的可编程API，它实现了大多数开发人员所了解和喜爱的一些基于配置的特性。

代码清单12-16显示了如何编写一个定制的追加器来捕获被记入日志的值，并在该代码被执行后对其进行验证。即使在一个简化的情况下，这个例子也有些冗长。我们定义了 before 和 after 方法来保证在每一个测试后都清理了追加器。否则，就得使用带有finally的try/catch语句。另外也需要将追加器进行缓存，这样才能在after中将其删除；还需要将前一个日志级别进行缓存，这样才能在after中将其还原。

代码清单12-16 使用一个定制的日志记录追加器来测试代码清单12-15中的代码

```
public class SUTTest {
    private ResultCaptureAppender appender;
    private Level oldLevel;
    @Before
    public void configureAppender() {
        appender = new ResultCaptureAppender();
        Logger log = Logger.getLogger(SUT.class);
        log.addAppender(appender);
        oldLevel = log.getLevel();
```

```
    log.setLevel(Level.DEBUG);
  }
  @After
  public void removeAppender() {
    Logger log = Logger.getLogger(SUT.class);
    log.removeAppender(appender);
    log.setLevel(oldLevel);
  }
  @Test
  public void testTestMe() {
    Integer expectedIntermediate = 17;
    SUT sut = new SUT();
    sut.testMe();
    assertEquals(expectedIntermediate,
      (Integer) appender.getValue());
    }
    private class ResultCaptureAppender
      extends AppenderSkeleton {
    private Object value;
    @Override
    protected void append(LoggingEvent loggingEvent) {
      value = loggingEvent.getMessage();
    }
    public Object getValue() {
      return value;
    }
  }
```

}

然而，上述代码做了一些并不很实用的简化了的假设。首先，它假定所测试的装置是 testMe()方法中的最后一条日志记录语句。在这条语句之后的另一个日志记录语句会覆盖所缓存的中间值。可以通过下面的方法来进行补偿，即存储所有记入日志的对象，甚至存储它们的调用栈。如果这是一个打算复用的测试机制，那么这样做还是值得的。但需要认识到，所构建的复杂的用来支持测试的基础设施，其本身也需要进行测试，以确保持续的正确性。

追加器的实现也作出了将消息类型进行强制转换的事情交给调用者来处理的选择。这虽然达到了让追加器通用的目的，但是会让测试本身变得混乱。而把强制类型转换的事情交给追加器代码来做，也会冒抛出恶劣的强制类型转换异常的风险。可以通过在追加器中对类型进行测试来避免抛出异常，但那也增加了测试装置的复杂性。无论强制类型转换或类型检查在哪里完成，这都已经将测试代码耦合到中间结果的内部数据表示上了。

考虑到额外的维护工作，我们应该避免使用这种技术。先前尽职尽责地添加的日志记录语句能有助于维持其存在。但谁能保证后来的维护者在找到了更好的方法来测试代码的情况下，不会删除这些注释呢？前面已经讨论了，如果另一条日志记录语句被添加到关键的日志记录语句之后会发生什么事情。对于这种修改，没有一种注释形式能够提供一种非常明显的方式来威慑这种改变。如果我们能够足够幸运地让不同的数据出现在下一条日志语句中，那么测试的失败就会将此处突显出来。无论使用哪种方式，这都需要额外的逻辑来将其磨练成完全符合我们所关心的针对测试的日志记录的用法，而使用这项技术所带来的额外测试，会让事情变得更加复杂。

尽管这项技术有所有这些脆弱性和缺点，但有些时候，它就是最好的测试方法。第 13 章展示了一个使用追加器来重现线程竞态条件的例

子，这在正常情况下是做不到的。

第13章 并行性

为了解决一个更难测试的问题——确定性地重现多线程的竞态条件，本章将介绍一些技术。不像本书所提及的大多数其他技术，本章中的技术不太可能被用于测试驱动开发的场景。竞态条件通常会显示成bug，而不是那些我们在开发中预测的或用来获得测试覆盖的行为。然而，对于用户、开发人员和支持人员来说，竞态条件bug是最令人沮丧的。重现这些bug以确保它们被修复并且保持被修复状态，能够极大提高上述人员在软件各个进程和系统工作中的信心。这些技术也能很好地适用于系统测试和集成测试，尽管出于单元测试的目的，每个缺陷都只能被隔离到一个或多个单元中。

大多数现代的平台都显式或隐式地支持多线程。例如，Java就是针对多线程而设计的，并且拥有一套强大的并发编程库。然而，Java的多线程和相关的库并不是从头或凭空开始设计的。例如，Java第5版引入的java.util.concurrent包中的许多特性都是基于 Doug Lea 的研究成果产生的，20 世纪 90 年代 C++的 RogueWave Threads.h++库也受了该成果的启发。

与此相反，JavaScript 仅仅是间接地支持线程。引发 socket 通信的那些操作，像jQuery 的 ajax()方法，是在一个不同的线程上运行 I/O 请求。而且，JavaScript 的window.setTimeout() API的使用，是在后台调用线程。原生的JavaScript通过在异步方法中的回调来解决这个问题。jQuery通过Deferred()实现改善了这一点。

这里所讨论的原则将能适用于一些不同编程语言和平台中的多线程环境，讨论中在细节上会有进一步的解读。

13.1 线程和竞态条件的简介

在深入探讨如何测试并行代码之前，让我们先简单了解一下并行编程[64]的历史，对竞态条件做个剖析。

13.1.1 一些历史

在计算时代的初期，软件一次只运行一条指令。早期的操作系统增加了一定程度的并行性，但是用户编写的程序还是一次只执行一条指令，然后休息一下，好让操作系统能够运行。起初，这个休息仅当上述程序允许时才会发生，这种方式被称作协作式多任务处理（cooperative multi-tasking）。但这还不够，所以人们对这个概念又做了扩展，用几乎相同的方式运行多个用户程序，并将这种方式封装到进程的概念中。这些进程被隔离到不同的地址空间中，由操作系统调度，通过一个称作抢占式多任务处理（preemptive multitasking）的机制，以并行的方式来执行。每一个进程都有其专用的内存和其他资源。它们使用大量的技巧和新的设施来在相互间进行协调，这统称为进程间通信（Inter-Process Communication，IPC）机制。

最终，还需要从计算机中获得更多。整个进程的开销，仅仅并行地运行了一些活动，而且复杂的IPC机制却经常仅仅是共享一些简单的数据结构，这种情形促使动了线程的发明。多个线程在同一个进程内分别执行，共享同样的内存和数据结构。然而，线程并不共享所有东西。每一个线程维护其自身的执行上下文，这个上下文由有限数量的栈内存和其自身的指令路径所组成。

13.1.2 竞态条件

线程会带来bug的可能性较高，因为它们无需显式地协调就能访问共享的数据。而IPC机制需要做出有效的预先计划才能申请进行这些访问。实际上，当看到线程这个字眼儿时，就能联想到在多线程方面编写

具有正确行为的代码的复杂性。如果代码在多个线程同时运行时能够保证正确性，那么这段代码就是线程安全（thread-safe）的。

大多数多线程的bug都是竞态条件。一般来说，如果数据写入操作协调不当，竞态条件就会发生。根据引发冲突的操作及其顺序，存在三种类型的竞态条件：写读（write-read）、读写（read-write）和写写（write-write）。

如果具有顺序依赖的异步操作协调不当，那么开发人员就会遇到读写竞态条件。在使用或读取值之前，应该先生成或写入它。在这种情况下，bug 会出现在协调计算的代码中。代码清单13-1用JavaScript展示了这种情况。jQuery的$.get()方法从给定的 URL 中异步地获取到数据，然后运行所提供的回调函数。在这种情况下，几乎可以肯定，在调用build_menu()之前，上述操作不会结束，未经初始化的变量menu将被传递给menu构建器。

代码清单13-1 JavaScript中的一个写读竞态条件

```
var menu;
$.get('/menu').done(function(response) {
   menu = response;
});
build_menu(menu);
```

读写竞态条件也称为先测试后设置，在代码使用值来决定该值将如何改变，如果没有确保这两个操作以原子化的方式发生，那么这种竞态条件就会发生。让我们看一个简单的例子，这是一个功能异常的资源初始化器的实现（见代码清单13-2）。

代码清单13-2 一个简单的功能异常的资源初始化方法

```
1 public Resource initializeResource() {
2     if (resource == null) {
3         resource = new Resource();
```

```
4    }
5    return resource;
6 }
```

竞态条件发生在第2行和第3行之间。考虑下面的事件序列。

线程A执行第2行并判定resource为null。

操作系统挂起线程A并恢复线程B的运行。

线程B执行第2行并判定resource为null。

线程B执行第3～5行，为Resource类分配一个实例并将其返回给调用者。

操作系统挂起线程B并恢复线程A的运行。

线程A执行第3～5行，为Resource类分配一个不同的实例并将其返回给调用者。

在之后的某个时间点，线程 B 再次调用 initializeResource()方法，resource变量不再为null，于是该方法返回一个与上次调用不同的实例。

那个线程 B 分配的第一个 Resource 类的实例后来怎么样了？这取决于编程语言。在Java中，一旦线程B完成了对它的使用，它就会被垃圾回收。在C++中，除非使用了某种形式的线程安全的智能指针[65]，否则它很可能成为一个内存泄漏，因为线程B没有理由知道它应该释放这块内存。无论在内存中发生了什么，本意是要创建一个资源，现在却创建了两个。

这里演示了临界区的一个最简单的例子，临界区是一段代码，该代码要想正确执行，每次只能由一个线程访问。在一些机器的体系结构中，这种情况甚至能发生在机器码或字节码的级别，如果由编译器生成的低级指令能有效地修改一个与其最终的存储位置相分离的值的话，就像在做一个自赋值的操作（如m += n）时所看到的那样。

写写竞态条件在较高级别的编程语言中发生的频率较低。指令级别的写写竞态发生得更加频繁，但它会以参与者的形式出现在另一个更高

级别的竞态中。代码清单13-3显式了一个在多线程中执行时可能会发生的写写竞态条件的例子。假设有一个队列，其中包含着等待处理的条目。以前，该队列是串行处理的，但为了得到更好的吞吐量，做了并行处理。代码假设队列中的最后一个条目应该最后处理。然而，一旦将条目处理并行化，处理倒数第二个条目的时间就会更长，要到最后一个条目处理结束后才能完成，这样记录下来的最后处理的条目就是错误的。

代码清单13-3 具有特定假设条件的写写竞态条件。例如，如果使用多个线程处理一个队列，那么原本以为的——队列中最后一个条目会被记录成最后一个处理，可能并不能如愿

```
public void recordLastProcessed(Item item) {
   lastProcessed = item;
}
```

13.1.3 死锁

当两个或多个线程停止执行，并互相等待或等待一个被共享的资源时，死锁就会发生。虽然竞态条件普遍是由于缺乏加锁操作而导致的，但死锁却是由于不正确的或过度的加锁操作而造成的。代码清单13-4显式了最终注定会死锁的两个方法。

代码清单13-4 在不同的线程中同时运行在同样的对象上的两个很可能会导致死锁的方法

```
public class ClassA {
   private ClassB that;
   public void doSomething() {
      synchronized(this) {
         synchronized(that) {
            // Do the work
         }
```

```
        }
    }
    public void doSomethingElse() {
        synchronized(that) {
            synchronized(this) {
                // Do other work
            }
        }
    }
}
```

注意这两个方法在获得锁时用了相反的顺序。如果运行doSomething()方法的线程要获取that对象上的锁，而在此之前，运行doSomethingElse()方法的另一个线程已经获取了这个锁，那会怎样？两个线程各自都等待另一个线程所拥有的锁，从而两者都不会继续运行下去。通常情况下，为了运行正确，加锁操作应该在所有的使用情况下都以同样的顺序来加锁——这被称为层级加锁（hierarchical locking）。这里在顺序上的交换是一个有关并发的代码腐臭。

13.2 一个用于重现竞态条件的策略

在讨论那些用于重现竞态条件的特定技术之前，让我们先制定一个总体策略，来指导我们解决这个问题。重现竞态条件需要能仔细地编排临界区中事件的序列，在修复竞态条件问题时，最好能以一种不会改变该序列的方式来进行。理解竞态条件的确切本质，有助于理解临界区的边界，进而理解我们可能控制的范围。我们要寻找存在于临界区内的接缝。

在代码清单13-2中，竞态条件被绑定在第2行和第3行代码上，其中

第2行测试resource是否为null，第3行是resource的赋值语句。在这两行代码之间没有发生太多事情，对吧？不见得，但是如果了解估值（evaluation）序列[66]，就能让我们发现更多可做的事情。

仔细看看第3行，会发现这里调用了一个构造器，且操作符的优先级保证了构造器的调用发生在变量赋值之前。这就是一个接缝。根据该构造器中所发生的事情的不同，可能可以开发出任意数量的接缝。更加复杂的情形经常会富含接缝。此外，底层的线程和执行实现可能会提供一些没有在代码的表面检查中显式表现出来的接缝。

当发现接缝后该如何处理它们呢？可以使用它们来显式地触发或模拟线程的挂起，这种挂起在多线程执行的过程中可能会自然地发生。下面将使用一些技术从根本上停止线程，直到某个其他的执行序列已经完成，以便我们可以再恢复该线程的执行，以复制竞态条件的行为。

首先概括一下这种方式。要先勾画出一种分析方法来诊断竞态条件。这需要花费一些时间来积累经验并树立正确的观点，从而才能很容易地将竞态条件挑拣出来。此外，还需要花些时间来熟悉开发环境中的那些同步API。希望这样做能令人步入处理竞态条件的正途。

（1）识别竞态条件。

确定自己可能有一个竞态条件。有时会因为数据毁损（corruption）而怀疑发生了竞态。有时会看到只偶发的意想不到的故障，例如在本章后边那个日志记录例子中生成的那些异常。其他时候，会看到事情好像以错误的顺序在发生。偶尔，应用程序的某部分会冻结不动。上述任何一个或所有的现象都可能会出现。关键是错误的行为是偶发的，经常跟特定类型的负载有很大的关联，或者以非常特定于运行环境方式发生，例如只在一个更慢或更快的CPU、浏览器或网络连接上发生。如果能总结出问题发生的统计性特征，那么就可以把这些数据当作一个基准。

如果合适的话，确保代码能在单线程的情况下正确地工作。如果花了大量的时间去追踪一个竞态条件，最后却发现其仅仅是算法中的一个

bug，那么真的会让人感到很沮丧。有时候会遇到这样的代码，其中的同步对于行为来说是不可或缺的，以至于很难做出正确的验证。在做进一步的处理之前，先重构代码，以便能够验证单线程的正确性。

创建一个交互本质的假设。竞态条件是顺序操作的问题。当问题涉及数据毁损时，数据修改的序列就需要修正。奇怪的行为仅涉及操作的顺序。通常情况下，创建该假设需要对系统的基础有一个相当深入的理解，例如多任务和时间片处理是如何工作的，各种锁是如何运用到运行时环境的，或者异步回调是如何处理的。确切地阐述出这个问题的一个看似合理的概念。

确定对上述假设负责的临界区。这一步要确定的是代码，而不仅仅是概念。对于数据的问题，要找出代码在单独的同步范围内读取和写入数据的位置。要了解那些数据结构是否是线程安全的，以及它们做了哪些线程安全的保证。对于既是数据又是行为的问题，要找出那些没有将依赖强制到顺序上的异步事件。对于冻结不动的问题，要找出过度的同步，特别是在管理多个锁且以不同于它们被加锁的逆序的顺序将其解锁这些方面。此外，要在同步的地方找出循环依赖。

（2）重现竞态条件。

验证假设。有时，可以直接跳到那些基于代码的测试中，略过这一步。大多数时候，这一步会涉及调试器。暂停并运行所验证的单个线程或多个线程，可能会修改所涉及的数据。一旦有了治愈竞态的攻略，就可以继续做下去。如果不能在调试器中复制竞态，那么就去寻找另一个临界区或者确切地阐述出一个新的假设。

确定临界区中的接缝。可以应用本书前面章节所讨论的大部分技术的各种变化形式来强制竞态条件发生。本章的余下内容将讨论这些应用的细节。可以使用几乎任何一个允许注入测试替身的接缝。此外，应该考虑任何允许与临界区上的那些锁进行直接或间接交互的地方。对于锁，在Java中有一个虽然不明显但可用的好例子，即通过wait/notify的

API让所有的对象都能使用的监视器（monitor）。

选择正确的接缝。从那些已经确定的接缝中，清除掉那些在修复竞态条件时会消失的接缝。在验证上述假说的过程中，或许会形成一个有关修复竞态条件的想法。如果该修复会改变一个接缝，那么使用该接缝将不会产生能验证该修复的测试。

利用接缝来注入同步。在验证上述假设时，是在调试器中单步运行代码来重现竞态。既然已经选择了一个要加以利用的接缝，就可以在调试器中使用它来用编程的方式重现所开发的控制序列。编写测试来执行该序列，并断言所期望的结果不会发生。对于死锁，一个在测试中设置的超时值就能很好地处理测试失败。例如，JUnit的@Test标注就能接受一个可选的timeout参数。

（3）修复竞态条件。

既然已经在测试中重现了竞态，那么一旦将其修复，测试就应运行通过。

（4）监测结果。

在第1步中，笔者建议应该将竞态条件发生的特征统计出来。不幸的是，开发人员已经发现、复制并修复的那个竞态条件，可能不是之前试图要寻找的那个。最终，已经将其修复的证据是该竞态条件会很少发生或根本就不发生，但竞态条件所具有的偶发本质使其只能通过持续的监控来验证。还要注意前文所说的“很少发生”这个词。有可能存在多个原因而导致一些类似的症状，而前面只找到一个。

13.3 直接测试线程的任务

面向对象的线程库通常会将一个线程的建模与该线程所运行的任务的建模相分离。通常情况下，也存在仅直接使用线程模型的便利施设，不用针对每个线程显式地创建分离的代表其任务的实例。自Java语言创

建以来，它就有一个Thread类，并且用Runnable接口来代表与其相关的任务。一个Thread类就是一个Runnable类型的类，它的默认实现会将其自身作为其任务来运行。

这种设计的种种不足之处[67]促使Java 5增加了java.util.concurrent包及其子包。除了广泛地增加了一些新的同步方法之外，还增加了作为执行模型的 Executor和作为任务模型的Callable。为了方便使用，同时也是其所必需具备的功能[68]，这些新增的包能够使用并支持那些较早版本的与线程相关的类。

Runnable和Callable这两者都被定义为接口，这帮我们理清了一些测试问题。作为接口，它们代表必须要在单个线程上执行的那些方法，而不是在这个线程上完成的实际工作。这表明，一个任务包含两个不同的目的，可能需要适当地将这两者分开进行测试。第一个目的是要能在单个线程上执行，第二个目的是要在该线程上能执行所需的功能。如果审视一下该任务的功能，通常会看到该任务并不需要在一个单独的线程中执行。

等等！为何单个线程的任务不需要一个单独的线程呢？大量多线程算法的本质是隔离大多数或全部数据，使得不需要对访问进行同步。当数据结果被连接在一起的时候，同步就会发生。不管怎样都必须等这些结果出来。在等待其他结果到来时，先连接已经完成的结果，能够提升计算的速度。这就是MapReduce框架[69]和其他并行数据分区方案（parallel data partitioning scheme）的本质。这意味着许多任务不需要任何同步，我们就能直接针对它们的计算目的进行测试！

那么，这是如何影响我们设计的可测试性的呢？首先，这表明应该确保任务被封装在其自己的可测单元中。虽然以一个简单的 Thread扩展来实现任务很方便，但是我们要确立分离关注点，通过将任务封装到其自身类中，更好地支持可测试性。其次，将内部类和匿名内部类当作任务来使用是很常见的。这种用法，经常对可能无需了解它们的外界，隐

藏任务的细节，但它也对测试隐藏任务。如果我们的任务能以带有名字的类，而不是以隐藏或匿名类来建模，那么测试就能更容易地访问它们，这些任务也就更可测试。

顺便说一句，同样的原则也适用于那些过程式的编程语言。例如，苹果公司针对OS X 和 iOS 的线程提供的线程库，明显就是过程式的，因为它需要函数指针或Objective-C的块（block）[70]来作为任务的定义。尽管以这种内联（inline）的方式定义这些任务很方便，但这会让整个代码更加难以进行单元测试。

让我们看一个将代码清单13-5所示的代码重构成一个单独的Runnable的简单例子。

代码清单13-5 直接用Thread实现并行计算

```
public void parallelCompute() {
    Thread thread = new Thread() {
        public void run() {
            Result result = computeResult();
            storeResult(result);
        }
        private Result computeResult() {
            ...
        }
        private void storeResult(Result result) {
        ...
        }
    };
    thread.start();
    // Do some things while the thread runs
    thread.join();
```

```
}
```

将Runnable从上述Thread中分离到一个命名内部类里，如代码清单13-6所示。

代码清单13-6 重构代码清单13-5，将计算部分提取到Runnable中

```
public void parallelCompute() {
    Thread thread = new Thread(new ComputeRunnable());
    thread.start();
    // Do some things while the thread runs
    thread.join();
}
private class ComputeRunnable implements Runnable {
    public void run() {
        Result result = computeResult();
        storeResult(result);
    }
    private Result computeResult() {
        ...
    }
    private void storeResult(Result result) {
        ...
    }
}
```

对于 result的存储可能需要略做重构，具体取决于实现方式，但是上面那个命名类也能提供更多的封装属性的选项。请注意，即使对于这个小得只剩下骨架的任务，将那个匿名内部类分离出来也能大大提高parallelCompute()方法的可读性。

将命名内部类ComputeRunnable提取到一个单独的类文件中是一个

微不足道的重构。不过从这一步开始，就能使用其他一些成熟的技术来测试这个类的实际功能了。

但是这里我们做了一个以前经常说的但不总成立的明显假设：该任务并不真的需要线程同步来完成其工作。本章余下部分将针对这些场景介绍一些技术。

13.4 通过常见锁来进行同步

现有的显式同步给了我们其中一个最简单的接缝来加以利用。如果被测代码有临界区，且该临界区使用的另一个类已经是线程安全的，那么最常见的形式就会发生。在这种情况下，可以使用在另一个类中的同步来在临界区中挂起。考虑代码清单13-7中的代码。

代码清单13-7 有关常见锁同步的讨论的一个例子

```
public final class HostInfoService { // Singleton
   private static HostInfoService instance;
   private HostInfoService() {
      ...
   }
   public static synchronized HostInfoService getInstance() {
      if (instance == null ) {
         instance = new HostInfoService();
      }
      return instance;
   }
   public HostInfo getHostInfo(String hostname) {
      HostInfo hostInfo = new HostInfo();
      ...
```

```
        return hostInfo;
    }
public class HostCache {
    private Map＜String,HostInfo＞ hostInfoCache;
    public HostInfo getHostInfo(String hostname) {
        HostInfo info = hostInfoCache.get(hostname);
        if (info == null) {
            info = HostInfoService.getInstance()
                    .getHostInfo(hostname);
            hostInfoCache.put(hostname,info);
        }
        return info;
    }
}
```

首先，竞态条件在哪里？反思一下HostCache的意图，这种功能通常被设计成一个单例的实例，即使它没有以一个单例模式的形式来实现。这意味着该应用程序将确保有一个且仅有一个这种功能，至少在某些共享的范围内是如此。其结果是，hostInfoCache这个映射（map）是一个共享的数据结构，即使它不是静态的。在Java中，Map＜K,V＞是一个定义了一个标准键-值存储接口的接口。这里没有给出它实际分配内存的实现，因为其实现类型和它的同步不是竞态条件的源头[71]。它也不是我们将在例子中加以利用的接缝，尽管它看起来应该可以作为一个可行的接缝选择。

假设上述共享范围包括多个线程，就如同在现代网络应用中常见的那样，那个getHostInfo()方法有一个与前面介绍的那个伪单例（pseudo-singleton）相似的竞态条件，但范围更广一些。

假设上面代码中映射的实现本身是线程安全的，临界区的起始部分

就是给 info变量赋初始值，其中那个从映射中获得的值存储在本地。从那一点开始直到存储新值的那个点，定义了临界区的范围。所存储的键值在另一个不同线程上的这两个点之间会发生变化，从而同一个键就有了两个不同的HostInfo缓存值。

那么，在临界区中，我们拥有什么接缝？根据所涉及的方法调用的本质，有3个能立即看到的接缝，如下所示。

（1）HostInfoService.getInstance()

（2）HostInfoService.getHostInfo()

（3）Map.put()

这个例子将使用第一个接缝 HostInfoService.getInstance()。那个在HostInfoService类中临时编写的 getHostInfo()方法的实现，并不是一个非常诱人的接缝，虽然一个更加现实的实现能够提供一个这样的接缝。Map.put()会难以使用，因为只暴露了接口，而没有暴露实现，并且无法控制其代码。在关注将要编写的测试以重现竞态条件的时候，请记住这些想法。

首先，让我们定义一个Callable来封装那个将会重现竞态条件的用法（见代码清单13-8）。下面将以测试类的一个私有内部类的形式来创建它。

代码清单13-8 一个用于帮助重现代码清单13-7中竞态条件的Callable

```
private class GetHostInfoRaceTask
    implements Callable＜HostInfo＞ {
  private final HostCache cache;
  private final String hostname;
  public GetHostInfoRaceTask(HostCache cache,
      String hostname) {
    this.cache = cache;
```

```
            this.hostname = hostname;
        }
        @Override
        public HostInfo call() throws Exception {
            return cache.getHostInfo(hostname);
        }
    }
```

将HostCache传入构造器，能够支持意图中高速缓存的单例用法。hostname参数支持对每个线程定义相同的查找键，这是竞态条件的一个必要特征。

现在已经有了线程中要执行的任务，重现竞态条件的测试方法如代码清单 13-9所示。

代码清单13-9 一个重现代码清单13-7中竞态条件的测试

```
@Test
public void testGetHostInfo_Race()
        throws InterruptedException,ExecutionException {
    HostCache cache = new HostCache();
    String testHost = “katana”;
    FutureTask＜HostInfo＞ task1 =
        new FutureTask＜HostInfo＞(
            new GetHostInfoRaceTask(cache,testHost)
        );
    FutureTask＜HostInfo＞ task2 =
        new FutureTask＜HostInfo＞(
                new GetHostInfoRaceTask(cache,testHost)
        );
    Thread thread1 = new Thread(task1);
```

```
        Thread thread2 = new Thread(task2);
        thread1.start();
        thread2.start();
        HostInfo result1 = task1.get();
        HostInfo result2 = task2.get();
        Assert.assertNotNull(result1);
        Assert.assertNotNull(result2);
        Assert.assertSame(result1,result2);
    }
```

在“四阶段测试”模式中，在我们启动线程之前，所做的一切都是为了设置fixture，线程启动之后，从将来要运行 SUT 的线程中获得结果，并用断言对其进行验证，最后利用Garbage-Collected Teardown释放内存。为了让测试尽量保持简单，我们还选择了在 throws子句中声明异常的做法。如果多次运行该测试，它运行失败的比例应该会相当高，并且总是在assertSame()断言语句上失败。

现在我们有了能够偶尔失败的单元测试。然而，统计数据并不像我们所期望的那样好。当测试像这个例子一样达到25%～50%的失败率时，依赖于统计数据可能还是可以容忍的。而如果细微的竞态条件的失败率表现在1%以下，并且整个测试台（test bed）甚至要花中等长度的时间（如 5 分钟）来运行测试，那么想要获得修复该竞态条件的合理信心，就需要花费平均超过 4 小时[72]的时间。让我们将这个竞态条件变成是可预测的。

察看getInstance()方法的实现，能看到该方法是同步的（synchronized）。该方法也能在特定的静态对象上使用显式的同步来取代现有的同步方式，并且许多人说这是更好的做法。但是对于我们的目的来说，这并不重要，所以我们使用上面这种更简短的表述。

在Java语言中，一个同步方法会在其所在的对象实例上进行同步操

作。但这是一个静态的同步方法，所以它会在它的类对象上进行同步操作。无论在上述两种的哪一种情况下，对于所有类似的同步方法来说，对象充当的都是监视器，用来控制该方法以排他的方式来执行。这些同步原语（primitive）本身会被构建到 Object 类中，而每个Java类都派生自该类。

下面将使用这个有关Java同步的事实，在临界区的中间位置捕获测试线程，以确保它们触发了静态条件故障模式。当这些线程到来时，如何能确保获得HostInfoService的监视器呢？让我们在一个同步块中添加两个对Thread.start()的调用。

但是，那样做只能保证在多个线程启动时，这个类监视器锁定的，但是不能保证在多个线程到达getInstance()方法的调用时，这个类监视器仍是锁定的。在测试的setup 部分就有了一个静态条件！可以很容易地解决这个问题。因为已经预见到了这个问题，所以选择了使用显式的Thread 实例来运行任务，而未使用像ThreadPoolExecutor那样的对象，后者在使用Callable时或许会被建议使用。虽然为了编写生产代码，执行器（executor）在封装和管理线程方面干得不错，但是与封装所允许的知情权和控制权相比，我们还想再多拥有一些。

针对线程的生命周期，Java语言有一个定义良好的状态机。具体来说，当一个Java线程正在等待一个监视器时，它会进入Thread.State.BLOCKED状态，直到它被释放。虽然在生产代码中为这个状态创建忙等待（busy wait）是一种很糟糕的风格，但是它却能让我们的测试既简单又充满确定性。启动线程的这两行现在被纳入fixture的setup部分之中，如代码清单13-10（添加的代码行用粗体字表示）所示。

代码清单 13-10 在进行测试之前先确保被测线程是阻塞的。粗体代码行被添加到代码清单13-9中相应的代码行中

synchronized (HostInfoService.class) {

```
        thread1.start();
        while (thread1.getState() != Thread.State.BLOCKED) { }
        thread2.start();
        while (thread2.getState() != Thread.State.BLOCKED) { }
    }
```

在测试离开 synchronized 块的时候，两个线程都在临界区中间的HostInfo Service.class监视器上等待。其中一个线程将获得锁并继续运行，而另一个线程则会继续等下去直到该锁被释放。无论是哪种情况，我们都已经确定性地重现了静态条件。现在，对被测代码的原有实现进行测试，在原有实现未做任何改动的情况下，该测试会100%地运行失败。

现在可以很容易地用几种方法中的任意一种来修复竞态条件。为了用最简单的方式来进行演示，我们仅为HostCache.getHostInfo()的方法定义添加synchronized，使它看起来像这个样子：

```
public synchronized HostInfo getHostInfo(String hostname) {
```

可以通过只在临界区周围加上显式的synchronized块来对范围做一些优化，把返回语句放到外面，并在该块前面定义info，但不赋值。根据该类整体同步的需要，该块可以在 this上进行同步，或在一个仅仅为保护该临界区而被分配的特殊的锁定对象上进行同步。

尽管Object监视器的具体细节都是以Java为中心的，但是类似的机制存在于所有的线程同步库中。并不是所有的机制都像这样被紧紧地集成到对象模型中，但是监视器、互斥[73]（mutex）和临界区都是一些线程编程中普遍的构建模块。

最后一点是，更复杂的加锁方案可能需要更高水平的奇思妙想。例如，上述测试代码直到线程被阻塞才实现忙等待。我们怎么才能知道线程是在我们认为它会阻塞的那个点上阻塞的呢？这会有什么影响吗？对这两个问题的简短回答分别是：“我们不知道”和“也许吧”。

在我们简单的例子中，测试已经确定性地运行失败了，而且经过检查得知，不存在其他能够对其进行解释的假设。也可以将测试置于调试器中，验证这些线程在离开synchronized块之前，就已经在所期望的位置被阻塞。如果该代码没有发生变化（对任何生产代码来说这都是个很糟糕的假设），那么简单地让测试可重现就能成为足够的证据。然而，出于代码维护的目的，我们应该知道代码的行为会与我们的设计相符，但不存在一个可编程的方式来对此做出断言。

13.5 通过注入来同步

只要常见锁存在且可访问，那么通过常见锁来同步就能工作得很好。但是如果这个锁不可用，或者可用的锁被放到了错误的位置，那该怎么办？创建自己的锁！

第12章讨论了可以将注入点用作接缝的各种方式。无论是通过依赖注入框架、回调、注册表，还是通过日志记录和监控框架，注入测试替身的能力也允许我们注入同步。

例子：通过日志记录来注入同步

尽管笔者曾用很夸张的方式警告过不要将日志记录框架用于测试，但是一些竞态条件还是需要使用它们。让我们看一个笔者遇到的真实的例子。

考虑代码清单13-11中的代码。乍一看，这段代码好像是线程安全的。Java保证Vector 类的单独操作都是线程安全的。第一个元素的获取和删除都在一个synchronized块中。然而，尽管在while条件中判断logRecords是否为空，与那个测试会在其上进行的synchronized块之间的时间间隔可能很小，但是这也代表了一个典型的check-and-modify竞态条件。当该方法被大量地调用时，对logRecords.firstElement()方法的调用有时就会抛出NoSuchElementException异常，表明该容器是空的。

代码清单13-11 一个带有日志记录接缝的竞态条件的例子

```
private static final Logger =
   Logger.getLogger(AuditTrail.class);
private final Vector＜AuditRecord＞ logRecords =
   new Vector＜AuditRecord＞();
public void postFlush(...) {
   ...
   while (!logRecords.isEmpty()) {
      log.debug(“Processing log record”);
      AuditRecord logRecord;
      synchronized (logRecords) {
         logRecord = logRecords.firstElement();
         logRecords.remove(logRecord);
      }
      // Process the record
      ...
   }
}
```

幸运的是，原始代码的作者将一个调试语句包含在循环中。代码清单13-12展示了一个可用来利用接缝的 Appender。它只是等待，每当有消息追加时，它就把自身用作监视器。如果日志语句不止一个，或者想要让锁有条件地介入，那么就可以创建一个更复杂的实现。

代码清单13-12 一个针对线程同步控制的log4j的Appender实现

```
class SynchronizationAppender extends AppenderSkeleton {
   @Override
   protected void append(LoggingEvent loggingEvent) {
      try {
```

```
            this.wait();
        } catch (InterruptedException e) {
            return;
        }
    }
    ...// Other required overrides
}
```

代码清单13-13[74]展示了，如何使用代码清单13-12中的Appender来重现竞态条件并最终验证我们的修复。在配置了log4j来使用自定义的Appender之后，代码清单13-13 中的测试在一个单独的线程中运行被测软件。在确保通过日志语句的诱导让该线程进入等待状态后，程序删除了那个排队记录，并且释放了那个线程。该线程使用Callable来执行，其Future对象被用于确定NoSuchElementException异常是否被抛出。该异常表明测试运行失败。

代码清单13-13 使用代码清单13-12的Appender来强制产生代码清单13-11中的竞态条件

```
@Test
public void testPostFlush_EmptyRace()
            throws InterruptedException,ExecutionException {
    // Set up software under test with one record
    AuditRace sut = new AuditRace();
    sut.addRecord();
    // Set up the thread in which to induce the race
    Callable＜Void＞ raceInducer = new PostFlushCallable(sut);
    FutureTask＜Void＞ raceTask =
        new FutureTask＜Void＞(raceInducer);
    Thread inducerThread = new Thread(raceTask);
```

```
        // Configure log4j for injection
        SynchronizationAppender lock =
            new SynchronizationAppender();
    Logger log = Logger.getLogger(AuditRace.class);
    log.addAppender(lock);
    log.setLevel(Level.DEBUG);
    inducerThread.start();
    while (Thread.State.WAITING != inducerThread.getState());
    // We don't want this failure
    // to look like the race failure
    try {
        sut.getLogRecords().remove(0);
    } catch (NoSuchElementException nsee) {
        Assert.fail();
    }
    synchronized (lock) {
        lock.notifyAll();
    }
    raceTask.get();
}
```

再重复一下，使用这个技术需要谨慎。程序员通常所忽略的作为测试接缝的日志记录语句，同样也会导致脆弱的测试，特别是在强制产生竞态条件时。随意删除用作接缝的日志记录语句会导致不寻常的测试运行失败。包围那条删除排队记录语句的try/catch语句，试图捕获测试运行失败的可能性，但是仍然需要一些仔细的诊断。同样，添加额外的日志记录语句，并与SynchronizationAppender的简单实现相组合，会导致测试无限期地中止（hang），并可能由于超时而运行失败。基于第一次

的执行结果或要记录的消息内容而发生的有条件的等待，可以解决这些问题。

在这个例子中，另一个被省略的重要步骤是删除共享的Appender。像log4j这样的日志记录框架，其能力源自于一个中央单例的本质——协调日志记录行为和配置。这个唯一的实例是一个横跨所有测试的共享资源，这导致追加器的变化会作用到其他测试上。更加强壮的实现会保留现有的追加器结构，并在测试的结尾处将其恢复。

13.6 使用监督控制

有时候，在进行测试时，就是找不到接缝。或许是因为类被封装得太好了。或许是因为被测软件的设计者当时没有意识到，会有允许调用的代码在软件中插入行为或监视挂钩（monitoring hook）的需要。又或许是因为软件的同步已经达到了“不太可能有竞态条件”的程度，至少直到系统的持续演进改变了“系统原本是线程安全的”这个假设之前是这样。

后者曾发生在笔者曾工作过的一个系统中。想像一下，在一个系统中，一些自治代理[75]（autonomous agent）在一个紧密设计的状态机的控制下来进行合作，共同实现应用程序的目标。在这个特定的系统中，各个代理以有效地构建因果关系模型的方式组织起来，耐心地等待来自它们先前的代理所生成的一个阻塞队列上的消息。这种方式曾经工作得不错，很有序，而且在代理的世界中同步得很好。

后来，我们认识到在这个过程中需要加入人类的影响。突然，各个事件可以依序列进入系统，而所依序列由于人类交互中天然的异步本质而无法被状态机所预测。一位特别具有创造性的系统测试人员创建了一个测试装置（test harness），能够像在键盘前的一个愤怒的小孩那样，快速随机地对系统产生外部影响。仅这一个系统测试在一夜之间就生成

了众多的并行性bug。鉴于该系统对健壮性有非常高的需求，并且要允许几百个用户能够同时在其上面进行互动操作，这个测试仅比预计的使用夸张一点点。

这个挑战发生的原因是，在一个非常简单的循环和一个现成的（off-the-shelf）并发组件之间的交互中发现了一个竞态条件，很像代码清单13-14中的Java代码[76]所显示的那样。

代码清单13-14 当使用线程安全的组件时出现的一个竞态条件

```
import java.util.concurrent.BlockingQueue;
class Agent implements Runnable {
   private BlockingQueue＜Event＞ incoming;
   ...
   @Override
   public void run() {
      processEvents();
   }
   public void processEvents() {
      Event event;
      while (event = incoming.take()) { // Blocks
         processEvent(event);
      }
   }
   public void addEvent(Event event) {
      incoming.put(event);
   }
   private void processEvent(Event event) {
      // Do something with the event
   }
```

}

当两个特定类型的事件快速连续地到来时，就会发生一个特别的问题。在自然环境下，两个同样的事件确实会同时到达，且两者都会排队。在孤立的单元测试中，我们决不可能如此快速地填充队列来重现这个测试条件。

那么接缝在哪里？我们在一个由系统提供的黑盒数据结构上，进行了一个几乎是微不足道的、非常紧凑的循环操作。唯一明显的接缝是一个我们不愿意暴露的私有方法[77]。说到这儿，希望读者能先把书放下一会儿，想一想抢占式多任务的本质，然后看看是否能找出那个隐形的接缝。

答案在于线程本身的时间是如何进行分片的。循环阻塞等待一个事件的那个点是一个明确定义的状态。因为Java中的线程生命周期是显式定义的，所以当循环在阻塞时，就能知道该线程的状态。Java拥有一些挂起和恢复线程的方法。这些方法已经被弃用了，因为在生产系统中，用这种方式来同步任务容易导致死锁。然而，由于我们这个测试具有简单和线性的性质，所以，对于我们现在的目的来说，使用这些方法也算是一个理想的方式，且其对死锁不是很敏感。能确定性地重现双事件（double-event） bug的测试可能类似于代码清单13-15。

代码清单13-15 一个能确定性地重现代码清单13-14中的bug的测试

```
public class AgentTest {
  @Test
  public void testProcessEvents() {
    Agent sut = new Agent();
    Thread thread = new Thread(sut);
    thread.start();
    while(thread.getState() != Thread.State.BLOCKED) {}
    thread.suspend(); // Freeze the thread
```

```
        // Queue up the double event
        sut.addEvent(new ProblematicEvent());
        sut.addEvent(new ProblematicEvent());
        thread.resume(); // Bug triggered!
    }
}
```

通过将线程置入一个已知的状态并将其挂起，就能保证在设置这个异常的测试条件时，不会发生任何处理。当然，这个测试缺少对成功事件处理结果的断言，但是读者可以在将本例应用于自己的真实世界的场景时将其添加进去。

13.7 统计性的验证

鉴于竞态条件的偶发本质，有时很难严格确定性地重现。然而，许多因素促成了竞态条件的发生，包括负载、频率、事务量和时序。在显式控制难倒我们的时候，上述因素就能有所帮助。

统计性的验证对于较粗粒度的测试效果最好，如系统测试和集成测试。也能将它有效地运用在较低级别的测试上，但在通常情况下，各种导致竞态条件的确切条件更难隔离和重现。有时，上述集成系统所产生的一些交互和时序，会逃脱诊断和重现。此外，较粗粒度的测试台针对较长的运行时间可能会有更高的耐受性，而较长的运行时间能与统计性的验证相关联。

有几种形式的统计性验证会比较有用。最简单的形式是利用竞态条件偶发的本质，将测试运行足够多的次数，使得几乎能保证出现运行失败。如果了解测试运行失败的频率，那么就可以用这种最简单的形式。如果知道一个测试有 1%运行失败的可能，那么按理将其运行100次，就可能会重现1次失败。但这能保证运行一定失败吗？不能。然而，运行

测试的次数越多，引起测试运行失败的可能性就会逐渐接近100%。一般来说，假设每次测试运行的失败概率P_f是一个常数，那么在经过了n次运行后失败的总概率P_F是

$$P_F = 1 - (1 - P_f)^n$$

所以，对于1%的单独的失败率，需要300次迭代才能达到95%的失败可能性，超过450次迭代才能达到99%的失败机会，如图13-1所示。

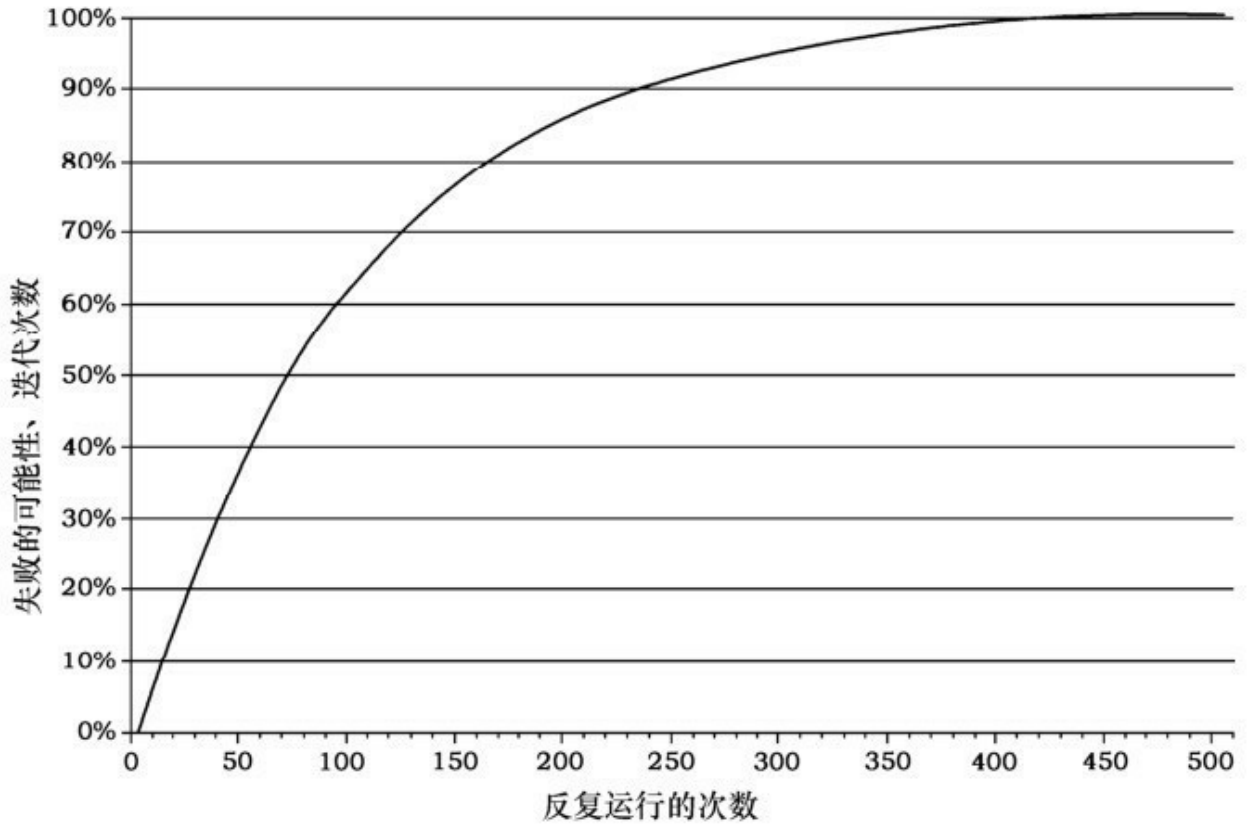

图13-1 对1%单独的失败率而言，失败累积的可能性与迭代次数之间的关系

在许多情况下，竞态条件仅发生在并发负载的情况下。与代码运行的次数相比，竞态条件显示出来的可能性，与并发参与者的数量的关系更大。相关的故障率与参与者数量之间的关系，为确定导致并发失败的负载提供了一个科学的方法。当用图形绘出参与者数量与系统指标（如内存或页面失效）或事件（如垃圾回收、通知或登录）之间的关系时，

能给人带来惊人的洞察力。一个装备良好的系统具有可用于对比的、特定于应用程序的使用指标，另外还有可用于回放的真实的事件序列。如果缺少这样的分析数据，那么可以应用试错法。

对于并发问题来说，“猴子测试”（monkey test）是一种有用的黑盒探索性测试。这种测试形式的名字暗示了无限猴子定理（Infinite Monkey Theorem），该定理展示了一只猴子随机敲击键盘的画面。因为它涉及并发测试，所以，如果能运行时间足够长，那么一个能随机向系统中注入输入和信号的框架，就能为该系统创建一个非常有效的压力测试。笔者曾参与过一个涉及自治代理的图形结构网络系统（即前面13.6节中所讨论的系统）的开发，在这个系统中，当在操作过程中随机启动和停止网络中的节点时，发现了大量的并发bug，包括一些一旦了解就能被确定性地重现的bug。

13.8 调试器API

请读者回想一下自己的一些最初的编程经验，那些你探索或学习如何使用调试器之后得到的经验。调试器改变了开发人员与软件交互的方式。不用先插入打印语句，然后又删除或者注释掉了，现在可以单步调试，跳过与要考察的内容无关的部分，下探到语句层面甚至汇编层面的细节，密切了解软件是如何真正运行的。

你用过能很好地处理多线程的调试器吗？它能极大地简化控制两个线程之间交互的任务。可以在断点上暂停一个线程以及与其交互的其他线程，然后以你所期望的顺序来单步执行它们。

这听起来很像我们使用之前讨论过的所有技术一直在做的事情。每一条指令现在都变成了一个接缝，即一个能够插入某种程度的控制来重现竞态条件的地方。这些地方只受底层机器或虚拟机执行模式的基本规则的制约。

但是，调试器如何能做到这一点呢？简而言之，每一个装扮成多任务的执行环境，当然有些不是装扮的，都会有对执行进行编程控制的API。调试器仅仅是具有成套深奥接口的程序而已。我们平时所遇到的调试器，一般都能基于我们手动发出的命令来执行精细的控制。设置断点，执行到一个特定的代码行，观察一个变量的变化。如果有足够的聪明才智，那么这些同样的API也能用于利用原子接缝。

要具备这种能力是要有代价的。一些调试器需要特殊的权限才能执行，而我们又不希望将这些权限授予任何开发或测试人员都能修改的自动化测试套件。使用这些API经常需要细致地理解底层执行模式，而理解这种模式至少需要针对一门技术进行广泛深入的学习，然而除非是编写编译器或调试器，否则可能不太会用到这门技术。

如果我们跳到代码以外的层面，那么就能获得更多利用这种技术的机会。尽管在有了非常好用的图形化IDE以后，这种方法看起可能很古朴，然而这种低级的命令行调试器能够降低这种方法的成本。许多命令行调试器，包括gdb、jdb和Perl调试器，都能在各种程度上进行脚本化，从而允许在自动化框架中使用它们。

这种技术的一个应用，是使用命令行调试器来设置一个变量监视器（variable watch），在一个蒙特卡洛模拟[78]（Monte Carlo simulation）过程中触发一个宏。该变量监视器建立了刻画偶发故障的异常条件。当检测到异常条件时，与触发上述监视器相关联的那些命令，就组成了期望接收的输出。代码清单13-16中的那些gdb命令[79]，当内存池中假想的垃圾收集的结尾处对象的引用计数非零时，会显示栈跟踪信息，然后继续执行。

代码清单13-16 在一次运行结尾处验证一个对象引用计数的gdb命令

```
break finalize_gc if refs > 0
commands
bt          # Stack trace
```

```
cont
end
```

虽然这肯定不是一个完整的测试自动化或竞态重现的解决方案，但是在隔离竞态条件，甚至在刻画其发生率的过程中，它可以是一个有价值的工具。将前面所讨论的有关垃圾收集的场景稍加变化后的例子，如代码清单13-17所示。

代码清单13-17 使用调试器来计算故障率

```
break gc
commands
set num_gcs = num_gcs + 1
cont
end
break finalize_gc if refs ＞ 0
commands
set num_failures = num_failures + 1
refs = 0
cont
end
break exit
commands
printf “Garbage collection failure rate: %f%%\n”,\
   num_failures/num_gcs
cont
end
```

---

[1].这已经被称为测试规格的Gherkin格式，从其与Cucumber BDD 测试框架之间的关联关系中衍生出来的。

[2].修改器与setter同义。参见http://en.wikipedia.org/wiki/Mutator_method。——译者注

[3].Hamcrest的各种匹配器可以在http://code.google.com/p/hamcrest/找到。它们已经被移植到多个语言中，较新的几个JUnit版本中也包含了它们。其他框架也能提供类似这样针对易读断言（literate assertion）的支持。

[4].参见5.1.1节。——译者注

[5].然而，像任何好东西一样，它也会被滥用。笔者见过的有关共享常量的最严重的滥用，是被预先分配（内存的）异常（preallocated exception）。特别是在Java中，当异常对象被分配内存时，其栈跟踪信息才被捕获。而当异常对象以静态常量的形式被预先分配内存时，其栈跟踪信息显示的是静态初始化时的上下文，而不是异常被实际抛出时的上下文。这一点在诊断一个运行失败的测试时会令人非常困惑。

[6].http://pivotal.github.com/jasmine/

[7].笔者更喜欢将方法命名为computeNow()而不是getNow()，这样getter的语义就可以专用来与get的前缀相关联。随着系统的生长，方法会以无法预知的方式发生变化。严格遵循命名的语义能有助于避开某些类型的技术债务。举个例子，笔者曾经工作过的系统，使用get方法来从数据库中获取复杂的对象。随着系统的生长，获取的行为开始有了需要额外的数据库写操作这样的效果，把get方法变成了也能做set的方法。

[8].或者是成员变量，不过对于这种情况，我们是不会直接对变量进行访问的，对吧？

[9].就个人而言，笔者更喜欢用protected。最不喜欢Java的一点就是，省略一个访问修饰符这样的语法标记竟然会带来功能性的后果。笔者更希望package关键字能够被重载，以显式表示包访问的可见性。

[10].此处的模式指的是正则表达式的模式，而非设计模式。——译者注

[11].如果对正则表达式有些生疏了，那么可以在这里回顾一下。正则表达式通常由斜杠（/）字符分隔。两个斜杠之间的整个表达式就是要匹配的正则表达式。$left、$right和$expectedSum这些表达式会被测试中本

地变量的值以正则表达式中的字面值的形式所替换。点（.）字符表示“任意字符”。星号（*）是说“前面的模式出现了零次或多次”。把这两者放到一起，就表示前面所讨论的“它们之间的一些字符”。最后一个字符是终结美元符号（$）。与用于表示变量引用的美元符号不同，这个在结尾处的美元符号会让正则表达式锚定到字符串的结尾处。

[12].即谎报成功。——译者注

[13].这里使用了Hamcrest匹配器以获得更易读的断言风格。

[14].Java语言具有能原生地处理UTF-16字符串的优势。如果使用像C++这样需要使用单独的API来处理多字节字符串的语言来编程，那么也能运用同样的技术，但是必须要使用相应的API，可能要转换表达形式，也许要编写一些辅助的工具。

[15].JUnit也有类似的功能，同样适用于我们这个特定的例子。但对于现实中最真实的例子来说，JUnit的这个功能使用起来会比较繁琐。详情请参见@RunWith(Parameterized.class)的使用。TestNG允许为每个测试方法定义其对应的测试表，而JUnit却强制为每个测试类定义该表，并且针对这些参数要运行整个类，而不管该类的方法是否使用这些参数。

[16].在Java中，这些方法就是或源自类的类型或可使用自动装箱（autoboxing）进行转换的类型。自动装箱是一个语言特性，它允许在像int这样的普通数据类型与其相应的像Integer这样的包装类（class wrapper）之间进行透明转换。

[17].令很多人感到惊讶的是，这个模式竟然没有出现在四巨头最初所提出的那些设计模式之中，而是在“程序设计的模式语言”（Pattern Languages of Program Design）系列图书中被首次提出。Martin Fowler[REF]、Josh Kerievsky[RTP]和Robert Martin[CC08]都在其著作中特意地使用了该模式。

[18].http://martinfowler.com/eaaCatalog/specialCase.html

[19].在面向对象的编程语言中，mixin是一个类，包含了一个来自其他类的方法的组合。不同语言对于这种组合有不同的实现方式，但不会通过继承来实现。引自wikipedia.org。——译者注

[20].有关trait参见http://en.wikipedia.org/wiki/Trait_(computer_science)。——译者注

[21].在笔者所认识的那些最有才华的人当中，有那么一位程序员仅能将事情做得好那么一点点。他将C++语句#define private public放到一个第三方库的头文件前面，访问那些正常情况下不被暴露的类的成员。尽管笔者并不建议使用这样的技术，但它的有效性真的令人吃惊。这样处理后能够通过编译并不令人感到惊讶，但不同的访问级别的命名在对象层面上被毁掉了，这会导致无法链接。

[22].Damian Conway的Class::Std是Perl的几个对象框架之一，它的创建弥补了Perl在对象方面的先天不足。就像大部分Perl的框架一样，该框架能够在CPAN上找到：http://search.cpan.org/perldoc?Class%3A%3AStd。

[23].http://code.google.com/p/guava-libraries/

[24].可以在此处发现一个很好的讨论：www.cprogramming.com/tutorial/friends.html。Scott Meyers在说 “这也省去了一个friend声明的需要，这一点被很多人看作是纯粹的粗制滥造。”这句话的时候，也暗示了同样的观点[MEC，第131页]。

[25].扑克牌型参见http://en.wikipedia.org/wiki/List_of_poker_hands。——译者注

[26].五牌扑克参见http://en.wikipedia.org/wiki/Five-card_stud。——译者注

[27].强制访问的原文是Coerced Access。——译者注

[28].有关Java的反射和内省的区别，参见http://stackoverflow.com/questions/2044446/java-introspection-and-reflection。——译者注

[29].对于一些犯下大错的很好的例子，参见http://stackoverflow.com/questions/2481862/how-to-limit-setaccessible-to-only - legitimateuses。其中让字符串可变的内容是笔者最喜欢的部分。

[30].这可以用SecurityManager来加以限制，但这一步应该能很容易地在测试之外来完成。

[31].http://code.google.com/p/dp4j/

[32].对于一些犯下大错的很好的例子，参见http://stackoverflow.com/questions/2481862/how-to-limit-setaccessible-to-only - legitimateuses。其中让字符串可变的内容是笔者最喜欢的部分。

[33].http://java.net /projects/privateer/

[34].关注点指的是一段代码所关注的内容，即代码职责。——译者注

[35].SOLID代表Single responsibility（单一职责）、Open-closed（开放-封闭）、Liskov substitution（里氏替换）、Interface segregation（接口隔离）和Dependency inversion（依赖倒置）。参见[ASD]和http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod。

[36].不要将单例模式与单例对象（singleton object）这个更为广泛的概念混淆起来，后者指的是一个对象。单例模式是确保获得单例对象的一种方式，但是还有其他的合理方法。

[37].使用快速搜索，就能够发现相当多的有关在任何给定编程语言中实现单例模式“正确”方式的争论。在Java语言中，静态成员初始化的行为和Java 5所带来的内存模型的变化引来了更多的讨论。其他编程语言也有类似的问题，如在《设计模式》[DP]一书中所提到的C++的问题。对于这些讨论的大部分内容来说，上述变化并不是讨论有关如何测试单例的材料，尽管可测试性可能会哄着令人远离一些具体实现。

[38].http://code.google.com/p/google-guice/

[39].言外之意，仅出于设计的目的而做出的单例选择，往往是不必要的。——译者注

[40].Damian Conway的Class::Std，一个更全面的Perl的面向对象的包，并不显式支持私有构造器。

[41].组块（chunking）是第二语言习得研究中的一个概念。有效的语言学习者会以一个个单元的方式去掌握一些词的“组块”或者序列，而不是

那些语言的基本构建块，以免对学习造成限制。这些组块经常是地道的、惯用的或很难解析的，而且当使用不当或不准确时就会显得相当奇怪。那些增加的抽象级别和加大的设计单元（如设计模式），就是编程世界中的组块，带有同样的好处和害处。

[42].http://pubs.opengroup.org/onlinepubs/000095399/functions/open.html

[43].异常消息的原文是Exception Message。——译者注

[44].原文为Exception Payload。——译者注

[45].存在着仅能被系统所设置的属性，例如Java中的栈跟踪信息，和具有更加彻底描述的C#中的context。

[46].Java将Error、RuntimeException和其他高级别的异常相互区分开来。Error是一个关键的不可恢复的系统问题。RuntimeException不需要被声明，而且被认为是一个有可能无法预知或无法处理的事件。所有其他的异常都必须要被声明。

[47].ratedTemperature意为额定温度。——译者注

[48].从技术上讲，不必声明IllegalArgumentException，因为它是Java的运行时异常。然而，是可以声明运行时异常的，而且在这种情况下，这样做能简化我们的例子。一个更加现实的例子是使用一个应用程序定义的异常来作为基类。

[49].分别是Source和TargetSite。

[50].也称为控制反转（Inversion of Control）。有关这个主题一个早期的统一看法参见www.martinfowler.com/articles/injection.html。

[51].http://docs.castleproject.org/Windsor.MainPage.ashx

[52].http://yan.codehaus.org/Dependency+Injection+Types

[53].http://static.springsource.org/spring/docs/3.1.0.M2/spring-framework-reference/html/testing.html#testco ntext-framework

[54].经过Sam Brannen的许可，这个例子引自

http://www.swiftmind.com/de/2011/06/22/spring-3-1-m2-testing-with-configuration-classes-and-profiles/。

[55].http://en.wikipedia.org/wiki/Function_object

[56].ISO/IEC 9899:1990，7.10.5.2节。

[57].http://linux.die.net/man/3/dlopen

[58].ISO/IEC 14882:2011，20.8节。

[59].这是一个非常明智的策略，不仅将测试从网络的使用中隔离出来，还保护测试以避免它们在并发和顺序执行上相互影响。

[60].请注意，一些编程语言提供了能缓解这些问题的一些特性。在Perl中使用local来覆写子程序的定义，或在JavaScript中在一个特定的对象上用另一种函数的实现来进行替换，都能产生更大的灵活性来解决这些问题。

[61].Arlo Belshee撰写了一篇颇有见地的博客文章，描述了有关用mock与不用mock之间的辩论，参见http://arlobelshee.com/post/the-no-mocks-book，其中包含了在测试中为了进行mock注入而专门创建工厂的例子。

[62].图中文字为：“放弃所有的希望吧，你这个进入此地的人！”出自14世纪意大利诗人但丁的作品《神曲》中的地狱篇。——译者注

[63].注意，像java.util.logging、sl4j和logback这些其他的Java日志记录框架都具有类似的功能。在其他语言中的那些最现代的日志记录框架也随之效仿。

[64].可以将本节看作是一个简写版的理想化的历史。像大多数在知识和实践上所取得的进步那样，实际情况并不像这里所介绍的那样是线性发展的，或是富有逻辑的。有关线程、Java并发库、原子性的原则和不可变性（immutability）等方面更详细的内容，笔者强烈推荐《Java并发编程实战》（Java Concurrency in Practice）一书[JCIP]。

[65].注意，并不是所有的智能指针实现都是线程安全的。要小心！

[66].“估值序列”处可以加一个脚注：即代码在编译、链接和运行时各个变量被系统确定其所包含的值的顺序。——译者注

[67].其中包括，与Runnable外部的异常进行通信的比较笨拙且时有bug的机制，以及没有任务返回值的规范。

[68].Executor并不真的是Thread的替代物，相反，它是Thread的包装类的一个抽象。虽然Callable是Runnable的改进和替代，但还是存在许多使用方便的构造器和方法，它们使用Runnable来使得过渡到新库的工作变得更加容易。另外还存在着大量期望使用Runnable的Java库，它们在Callable出现之前就已经被设计好了。

[69].参见Google Research上的有关MapReduce的论文原文 http://research.google.com/archive/mapreduce.html。

[70].块是Objective-C针对闭包（closure）的实现。在Java语言中，Java 8的lambda出现以前，与其最接近的相似物是匿名内部类。

[71].假设我们正在使用一个线程安全的Map实现，而HashMap（http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html）这个或许是最常用的Map实现并不是线程安全的。

[72].5分钟乘以100就是500分钟，也就是刚刚超过8小时。在统计上，在前4小时左右的时间里，存在50%的可能性可以看到测试运行失败。

[73].参见http://en.wikipedia.org/wiki/Mutex。——译者注

[74].这个例子最初是用在InformIT杂志的一篇文章中。完整的代码例子可以与该文章一起在http://www.informit.com/articles /article .aspx?p=1882162找到。

[75].自治代理指的是那些能独立工作的软件组件，它们拥有本地定义的任务集，共同影响突发行为。在自然界中，自治代理的典型例子是蚂蚁、蜜蜂和鸟类。例如，可以用算法定义一只鸟相对于距其最近的一些鸟的行动而做出的行为，从而精确有效地模拟整个鸟群的行为。

[76].如果对这里的细节感兴趣，可以参考BlockingQueue接口的Javadoc。参见http://docs.oracle.com/javase/6/docs/

api/java/util/concurrent/BlockingQueue.html。

[77].在现实的系统中，像processEvent()这样的方法是protected的。但是这个bug发生在这个方法的一个具体实现中，而这个实现本身又被强有力地封装起来，且十分复杂，使得还没有一个良好的测试能试图修改这个实现的一个副本。

[78].蒙特卡洛模拟，又叫蒙特卡洛方法，或蒙特卡洛实验，是一大类依赖于重复随机抽样来获取数值的计算算法，通常是多次运行模拟系统以获得未知概率的实体的分布。参见 http://en.wikipedia.org/wiki/Monte_Carlo_ method。——译者注

[79].gdb的文档参见http://sourceware.org/gdb/onlinedocs/gdb/Break-Commands.html#Break-Commands

第三部分 实例

前面对各种技术的单独介绍，不管有没有提供例子，都只是为了传达知识。开发人员需要看到这些技术的实际应用，并亲自使用它们，才能达到真正的融会贯通。在使用技术时无法包办代替，因为那是使用者自己的责任。然而，可以通过实例展示如何在真实的场景下使用这些技术。

为了提供多方面的指导，笔者选择了两个使用不同语言和测试风格的项目。第一个例子是用测试驱动的方法编写的一个新的Java项目。第二个例子选择了一个未经测试的开源JavaScript用户界面组件来进行测试。

这两个例子都保存在笔者的 GitHub 账号中。为了让读者能够详细地看到这两个项目的开展过程，笔者做了一系列的微量提交（micro-commit）。因为活生生的细节都已经公之于众了，所以笔者打算节省几棵树木，仅在本书中叙述这两个项目的开发过程，并穿插一些摘录的代码。结合 GitHub 代码库来看这个叙述能进一步加强学习效果。代码中像[badd0g5]这样的标注引用了Git提交的哈希值。通过将其附加到针对相关代码库的提交路径之后，就可以找到确切的代码提交。例如，如果上述哈希值是笔者一个不存在的“greatsoftware”代码库中的一个提交（当然它不是）的话，能看到代码差异的URL会是

https://github.com/srvance/greatsoftware/commit/badd0g5

并且下面的URL会显示在开发当时那一点上可浏览的源代码树：

https://github.com/srvance/greatsoftware/tree/badd0g5

第14章 测试驱动的Java

第一个例子演示了使用测试驱动开发来创建一个新的Java应用程序。该项目名为WebRetriever，它基本上是一个可以工作的命令行命令curl(1)[1]的子集，在大多数Linux发行版和Mac OS X中都能找到。如果读者不熟悉curl，那么这里简单介绍一下，它是一个用于从互联网上获取内容的工具，获取的内容包括身份验证、cookie管理以及其他更多特征，这些内容都可被指定为URI。

该项目的优点是相对较小和直接，但却很适合在其中运用本书所讨论的一些技术，这主要是因为它使用了网络，更确切地说，它使用了Apache HttpClient库[2]。该项目在GitHub上的代码库参见https://github.com/srvance/QualityCode14，其中一系列的微量提交详述了软件的演进、所犯的错误和其他一切内容。

WebRetriever支持以下特性，其中命令行选项指向了curl文档中相应的选项。

采用那些后跟一个或多个URL的选项。

从URL中获取内容，并将内容写入命令行终端，除非选项另有要求。

仅支持HTTP，默认是GET。

假定HTTP未被指定。

支持-O选项来将文件写入同名的本地文件中。

curl将该选项应用到下一个URI上。如果希望用这种方式处理多个URI，那么就需要将该选项放到每个URI之前。

如果文件已经存在，包括在一次调用中存在多个同名文件时，那么

curl会乐于覆盖这些文件。

如果像访问一个网站的默认主页那样，没有显式地提供文件名，那么curl会报错。

这里省略了一些在正常情况下该项目本应包含和认可的实践。比如在Maven的构建中没有设置静态检查器，针对风格和错误检查仅依赖IDE的默认反馈。通常情况下，笔者会将Checkstyle、PMD和FindBugs（三者皆包括），或者Sonar集成到构建系统中，并将这些配置到IDE中，以获得即时的反馈。正常情况下，笔者也会使用代码覆盖率工具来提供有关测试完整性的反馈。由于要将重点放在讨论技术上，所以此处省略了上述那些实践。最后，为简明起见和以讨论技术为重点，笔者还跳过了针对生产代码编写Javadoc这一步。

14.1 bootstrapping

一个纯粹的测试驱动项目会从一个测试开始。然而，在GitHub中创建repo（代码库）时，如果想克隆该 repo，那么就应该创建一个README.md 文件。这是一个为该项目编写规格（spec），至少是初始规格的好机会，于是笔者首先编写了这个规格。

接下来，笔者使用 Maven 的 archetype 生成，建立了该项目的结构。这里选择了一个能够生成 Java可执行 JAR包项目的 archetype，它生成了主要的类及其测试类。如同在第6章所讨论的那样，这里从构造器开始。由于当时并不需要属性，所以并没有应用完整的bootstrapping构造器的攻略，而是在各种项目设置的提交之间，挤出时间编写了一个简单的默认构造器的测试[445f41f]。测试驱动从[0c2d67e]处正式开始。

请注意，这里一开始就测试一个实例的构造及有关第一个产品特性的主要方法，而并没有测试main()方法。这是为什么？

除非代码中的类拥有显著的静态内容，否则几乎不可能测试一个

main()方法。虽然可以从一个测试方法中以调用静态方法的形式调用main()方法，但是main()方法完成工作的方式通常是创建一个本类或其他类的非静态实例。最好的办法是让main()方法尽量地薄，只把它想成是将程序功能包围起来的顶层异常处理层。这个方法保持得越简单，测试它的机会就会越少。随着项目的进展，将会看到WebRetriever 的main()方法的复杂性会有小幅波动，但最终会稳定在一个非常简单的形式上。

14.2 首要功能

应用第3章的攻略，可以立即跳到正常路径上。该应用程序的目的是从URL上获取内容。最简单的正常路径从一个URL上获取内容。这里在定义基本的retrieve()方法时，使用了该方法的测试（见代码清单14-1）。

代码清单14-1 在定义简单的retrieve()方法时，在其测试中使用了该方法[0c2d67e]。这个定义是“失败”的，因为该方法尚不存在

```
@Test
public void testRetrieve_SingleURI() {
   WebRetriever sut = new WebRetriever();
   String content = sut.retrieve(“http://www.example.com”);
}
```

就目前而言，本着YAGNI的精神，最简单的实现是接受这个URL，并以字符串的形式返回所获取的内容，将有关命令行参数和输出的问题推迟到后面再处理。

接下来的几次提交，一直到[839e4dc]，都对测试进行了演进，使其能表达出所期望的全部意图；并且对实现也进行了演进，使得测试能够通过。此时，测试还需要加强，因为它仅依赖于是否包含一个字符串。

根据大部分定义，该测试不应算作单元测试，因为它还对网页打开了socket。然而，在首先令其能够工作这方面，现在的进展还是很好的。这里使用了Apache HttpClient库来避免自己编写整个协议。毕竟，测试驱动并不意味着应该彻底重做所有事情。

在手工探索以确保测试数据合适的过程中，发现那个测试的 URL实际上是使用了一个HTTP 302响应，来重定向到另一个不同的主机上。对于调用代码，HttpClient库透明地处理了这件事，这是WebRetriever在操作上不同于curl的一个行为。将目标定为创建类似于现有可工作的软件的好处之一就是可以将现有的软件当作基准。下面将忽略这些差异，尽管在发现重定向时，curl的行为是至关重要的。

直到[1b49d37]的一些重构的提交带来了第一个显著的可测试性变化。

14.3 切断网络连接

一般在进行测试驱动开发时，都会创建支持测试意图的接缝。但笔者意识到retrieve()方法（见代码清单14-2）

有点儿长；

没有现成的接缝来为测试切断网络连接；

并且为了支持其目的，进行了各种底层操作。

笔者决定在继续之前，先将其进行简化和封装。[c602d34]的提交是该原则在WebRetriever中的首次应用。

代码清单14-2 WebRetriever.retrieve()的第一个可工作的版本，程序笨拙，可测试性差

```
public String retrieve(String URI) throws IOException {
    HttpClient httpClient = new DefaultHttpClient();
    HttpGet httpGet = new HttpGet(URI);
```

```
    HttpResponse response = httpClient.execute(httpGet);
    HttpEntity entity = response.getEntity();
    InputStream content = entity.getContent();
    StringWriter writer = new StringWriter();
    IOUtils.copy(content,writer);
    return writer.toString();
}
```

纵观该方法，可以将其步骤按照设置、执行和响应处理进行分类，非常像测试的前三个阶段。然而又与测试不同，在重构时，不需要在各个阶段之间保留边界。这里选择了将设置和执行所组成的响应的获取操作提取出来，作为一个单独的方法（见代码清单14-3）。

代码清单 14-3 重构代码清单 14-2 中的代码来提供一个接缝，从而至少能够使我们的一些测试避免网络连接[c602d34]

```
public String retrieve(String URI) throws IOException {
    HttpResponse response = retrieveResponse(URI);
    HttpEntity entity = response.getEntity();
    InputStream content = entity.getContent();
    StringWriter writer = new StringWriter();
    IOUtils.copy(content,writer);
    return writer.toString();
}
protected HttpResponse retrieveResponse(String URI)
        throws IOException {
    HttpClient httpClient = new DefaultHttpClient();
    HttpGet httpGet = new HttpGet(URI);
    return httpClient.execute(httpGet);
}
```

接下来的提交，直到[c1d4b2f]，使用了 EasyMock 来利用接缝，尽管这是一种令人厌恶的耦合到实现的方式。这个问题会得到及时处理。接下来直到[b61f107]的提交，做了一些额外的重构来改善两个测试和被测代码，为开发下一个产品特性做准备。

14.4 转移到处理多个网站的情况

接下来直到[4bdedba]的提交，对将要获取的多个 URL 进行了处理。此时，仅处理不带修饰符（modifier）的直接获取。所以最容易的方法是创建一个新的切入点（entry point），来接收多个URL并做循环，以针对每一个URL调用现有的retrieve()方法。这需要针对多个内容体应如何连续显示来制定一些细化的规格。

针对该功能的驱动测试在提交[5334caa]中。该测试创建了一个stub，通过使用一个计数器来为每次调用返回不同的内容。不同的返回值有助于使该测试避免出现偶然的正确情况。

有些人会反对在测试和生产代码中使用StringUtils.join()。笔者并不以为然。首先，笔者在每处都使用了该方法的不同形式。其次，因为StringUtils.join()是一个第三方工具，仅表现了上述功能的一小部分。笔者不觉得测试与代码实现之间的对应关系如此至关重要。

在开发下一个产品特性之前，笔者决定充实上述 main()方法的内容，以便做一些手工测试。笔者称得上是单元测试的粉丝，因为单元测试是测试策略的基础，读者还需要该策略的其余部分。如果笔者不是试图仅把重点单纯地放到测试技术的应用上，那么可能会使用一个像Cucumber那样的BDD工具，来创建更高层面的验收测试。对于这个项目来说，可以仅依赖于手工测试。提交[ead7038]填补了 main()方法的初始化实现。该实现比它应有的样子笨重了些，因为有些问题需要随着项目的进行逐渐解决，但是目前来说它已经足够好了。

14.5 幽灵协议

规格规定，WebRetriever应该将协议或者那些正执行RFC[3]的语言中的scheme默认定义为HTTP。在最新进行的手工测试中所进行的探索揭示了该特性尚未实现。由于该特性是下一个实现目标，既然被证实尚未实现，那么接下来的工作就是实现该特性。

14.5.1 死胡同

下面将通过重构来创建一个依赖注入接缝[63797de]，以此作为起步。这个想法是注入一个 HttpClient 的 mock 来检查在retrieveResponse()方法中传入HttpClient对象的 execute()方法的参数。为此，笔者将 HttpClient实例提升成为类字段（field），并在默认构造器中将其初始化。然后提出（factor）一个新构造器，用HttpClient实例作为参数，并且修改了默认构造器，通过传入一个全新的实例，将默认构造器链接到了新构造器。

在编写测试来利用接缝的过程中，笔者发现 DefaultHttpClient 和AbstractHttpClient 类的许多方法，包括代码所使用的带有单个参数的execute()方法，都是final的，这会阻止覆写，从而无法使用注入方法。提交[03e0517]删除了新构造器，但保留了默认构造器的新版本，并让HttpClient实例继续保持为类字段。

14.5.2 spy手艺

在某种程度上，之前走的错路帮助了笔者。注入一个mock本来是一个有效的测试技术，但这种做法也会通过一个内部类型的使用将测试更加紧密地耦合到实现上。这是使用mock进行测试时所面临的最大危险：mock引入的耦合程度。提交[7f74b88]引入了一个很好的避免了耦合的测试，如代码清单14-4所示。

代码清单14-4 一个针对将默认协议设定为HTTP的行为的独立于实

现的测试

```
@Test
public void testRetrieveResponse_DomainOnly()
    throws IOException {
  WebRetriever sut = new WebRetriever();
  HttpResponse response =
    sut.retrieveResponse("www.example.com");
  assertThat(response,is(notNullValue()));
}
```

在mock方法背后的概念，是要检查传入到库中执行内容获取的URL。当前的做法是以字符串的形式传入URL，这样做除了不能控制代码之外，这会导致第7章所讨论的所有字符串验证的问题。这会在HttpClient的内部发生。面对这种情况，笔者考虑了以下两点。

（1）笔者宁愿有一个更加结构化的方式来检查URL的格式。

（2）可能需要创建自己的接缝，因为HttpClient不支持将其组件用作测试替身的要求。

第一点令笔者迅速想到 java.net.URI类，它能对URL 有更加结构化的理解。它提供了解析URI字符串的能力，并且HttpClient能够将其作为URL参数的替代格式来接受它。

URI类还提示了如何创建接缝。URI类提供了更多的结构，而且HttpClient能够接受它，但是URL还是以字符串的形式从命令行参数传入。这提示我们需要一个转换层，一个接缝的理想所在！修改retrieveResponse()来将字符串的URL转换成一个URI，并将这个URI传递给HttpGet构造器（见提交[228d5c7]），能令我们重构出一个接受URI参数的retrieveResponse()方法的重载（overload）（见代码清单14-5）。

代码清单14-5　重构后的retrieveResponse()方法带来了一个可以利用的接缝

```
protected HttpResponse retrieveResponse(String URI)
    throws IOException,URISyntaxException {
  URI uri = new URI(URI);
  return retrieveResponse(uri);
}
protected HttpResponse retrieveResponse(URI uri)
    throws IOException {
  HttpGet httpGet = new HttpGet(uri);
  return httpClient.execute(httpGet);
}
```

提交[05e5c03]创建了一个间谍（spy），捕获用于构建HttpGet实例的最终URI，并用它来验证该 URI根据规格对scheme进行了处理。由此产生的测试和spy如代码清单14-6所示。

代码清单14-6 修改代码清单14-4的测试，创建一个spy，以利用在代码清单14-5所引入的接缝

```
@Test
public void testRetrieveResponse_DomainOnly()
    throws IOException,URISyntaxException {
  WebRetrieverURISpy sut = new WebRetrieverURISpy();
  sut.retrieveResponse(“www.example.com”);
  URI actualURI = sut.getSuppliedURI();
  assertThat(actualURI.getHost(),
    is(equalTo(“www.example.com”)));
  assertThat(actualURI.getScheme(),is(equalTo(“http”)));
}
private class WebRetrieverURISpy extends WebRetriever {
  URI suppliedURI;
```

```
    @Override
    protected HttpResponse retrieveResponse(URI uri)
            throws IOException {
        this.suppliedURI = uri;
        return createMockResponse("");
    }
    public URI getSuppliedURI() {
            return suppliedURI;
    }
}
```

上述spy将自己插到中间，并捕获了URI以便稍后进行检查，另外又调用了被覆写的方法的行为，使得实际的行为能畅通无阻地进行下去。这能让测试获得便于在断言中使用的URI的结构化表示。测试捕获了代码中存在的缺陷，笔者在同一个提交中修复了它。接下来的几个提交（一直到[897bfab]）做了一些重构，添加了一个测试以确保受到支持的现有行为仍然有效，并强制唯一支持的scheme就是HTTP。

14.6 执行选项

下一个特性就是支持选项（option）。或许最有用的选项会将所获取到的内容写入一个文件，尽管没有这个选项也始终可以重定向输出。在提交[bb7347e]中，笔者开始实现“-O”选项，具体是通过更新细化curl行为的规格来实现的。

支持这个选项的道路有些迂回曲折。必须承认，笔者起初对如何实现该特性没有一个完整的或者说整洁的概念。相反，笔者开始时只有一个如何处理这些参数的想法，但也只是对输出处理可能会难以测试这一点有清楚的认识。最后还是决定先把这些杂乱的想法和相关的测试变化

给展示出来，不做探索性的spike[4]，然后再顺着一条完整的路线恢复它们。希望这样做能带来更多的信息。

笔者使用了 retrieve()方法的多个 URL 的重载方法来编写测试（见提交[d451e31]），就好像该重载方法就是被设计出来处理选项的。最初的测试并没有完全表达出整体的意图。相反，它只是说，给定两个参数：一个选项和一个URL，应该只得到一个获取结果。为了做到这一点，笔者使用了匿名内部类来创建 mock，但没有使用mocking框架。该mock对有单个URI参数的retrieve()方法的调用次数进行了统计，并内嵌了该方法仅被调用一次的断言。虽然在第一轮的时候漏掉了一些细节，但最后在提交[4102ad1]中得到了所期望的测试运行失败。

提交[3422f2e]添加了测试和逻辑，其中逻辑显式地表明了要写入文件的请求。接下来又前前后后地进行了一些修改——所幸的是前进比后退要多一些——梳理出了emit()方法处理输出的概念（见提交[b994220]），尽管想法还远未完全成熟。

当意识到已经将有关输出的必要状态分散到几个方法调用和抽象的层面上之后，就开始进行一系列的重构，这些重构从提交[87ae64d]开始。这包括通过直接测试rectifyURI()方法来消除以前所创建的spy。然而最值得注意的是，这里创建了一个Target 的概念，即最初是一个嵌套类，而最终成为一个完全的顶层类（见提交[46978e9]），并带有它自己的一套测试（见提交[dde0a09]）。Target 起先作为存储库（repository），存储与一次获取相关的数据，但后来逐渐演变成功能的中心实体（见提交[13df657]）。

一些中间阶段的代码相当丑陋，特别是测试代码。在笔者清楚地意识到大部分功能都属于 Target之后，就开始向其迁移容易迁移的功能。但是其他方法就会更麻烦一些。经过覆写的那些方法无法移动，这证实了将测试耦合到实现上是很危险的。在某些情况下，必须得先注释掉方法中的那些覆写语句，IntelliJ才允许移动这个方法。然后再取消注释，

并将它们移动和调整到新的上下文中。最终，获得了一套更加整洁的代码，并完成了输出标志这一产品特性。

在几个做清理工作的提交之后，又重构了extractContentFromResponse()方法，准备完全切换到基于流的输出方法（见提交[b00689d]）。基础打好后，提交[39461ce]又删除了一些字符串的使用，这些字符串是内容的中间容器。这迫使现有测试做出了相应的变化，使得它们能够使用覆写来注入 ByteArrayOutputStream，进而捕获输出以便进行检查。

14.7 走向下游

最后一组变化为实现输出的各个方面做好了准备。提交[99cb54b]引入了一个有效但有些笨拙的测试，来验证System.out被用于正常输出。之所以说它笨拙，是因为使用了 copiedToOutput变量，该变量需要在retrieve()方法的覆写中进行检查，如代码清单 14-7 所示。这无异于手写mock。但如果不这样处理，就无法保证copyToOutput()方法被调用，因而无法保证其断言被执行。

代码清单14-7 用手写mock来验证System.out被用于输出的有效但笨拙的测试

```
public void testRetrieve_StandardOutput()
    throws IOException,URISyntaxException {
  final String expectedContent = “This should go to stdout”;
  Target sut = new Target(EXAMPLE_URI,false) {
    boolean copiedToOutput = false;
    @Override
    public void retrieve()
        throws IOException,URISyntaxException {
```

```
            super.retrieve();
            assertThat(copiedToOutput,is(true));
        }
        @Override
        protected void retrieveResponse()
                throws IOException,URISyntaxException {
            setResponse(
                WebRetrieverTest.createMockResponse(expectedContent));
        }
        @Override
        protected void copyToOutput(
                InputStream content,OutputStream output)
            throws IOException {
            assertThat(System.out,is(output));
            copiedToOutput = true;
        }
    };
    sut.retrieve();
}
```

下面的提交[f89bf38]将上述的匿名内部类重构为一个显式的嵌套类，并将其转变为一个spy，而不是一个mock。这产生了一个更好看的测试，以及一个能被其他输出变体所复用的spy（见代码清单14-8）。

代码清单 14-8 清理代码清单 14-7 中的测试版本，以显示一个更整洁的测试风格，并准备在即将到来的各种变体中进行复用

```
public void testRetrieve_StandardOutput()
        throws IOException,URISyntaxException {
    OutputSpyTarget sut =
```

```
        new OutputSpyTarget(EXAMPLE_URI,false);
    sut.retrieve();
    OutputStream outputStream = sut.getOutputStream();
    assertThat(outputStream,is(notNullValue()));
    assertThat(System.out,is(outputStream));
}
class OutputSpyTarget extends Target {
    OutputStream outputStream = null;
    public OutputSpyTarget(String URI,boolean writeToFile)
            throws URISyntaxException {
        super(URI,writeToFile);
    }
    @Override
    protected void retrieveResponse()
            throws IOException,URISyntaxException {
        setResponse(WebRetrieverTest.createMockResponse(“”));
}
    @Override
    protected void copyToOutput(
                InputStream content,OutputStream output)
            throws IOException {
        outputStream = output;
    }
    public OutputStream getOutputStream() {
        return outputStream;
    }
}
```

余下的提交使用 OutputSpyTarget 类来验证它写入了一个文件，并且当没有提供文件路径时，它会运行失败。

14.8 回顾

WebRetriever现在已经达到了这样的一种功能状态，即与更加成熟的前任curl相比，WebRetriever多具备了一些额外的功能，至少对于已经实现的特性是这样的。它能默认选择一个scheme，并且自动遵循基于HTTP的重定向。如果想用它来诊断重定向的行为，那么后者就是一个不利因素。但对于大多数以内容为目的的应用来说，它还是有帮助的。对于笔者所编写的代码，只要用Maven来构建并以JAR包的形式来调用它，就能得到适合它运行的环境。

它的功能远没有达到 curl的水平，但是作为测试技术的一个演示来说，如果真要达到 curl的水平，那么这个过程会很乏味。笔者已经测试了一些错误条件，但还有许多没有测试。

从软件设计的角度来看，它经历了一些相当丑陋和混乱的中间形式，但是它的塑造过程是合理的。有时，它也测试了单独使用IntelliJ来重构代码的能力。

笔者并不热衷于那些在WebRetriever.retrieve()方法中的URI参数中被处理的参数。参数解析——可以宽泛地将其描述为解析——使用了一个不具伸缩性的极为简单的方法。但对于一个单个的选项来说，它已经足够了。如果要进一步考虑处理其他选项，那么可以使用像Apache Commons CLI库[5]那样的框架。

看起来似乎是TargetSet类可能会持有一个将要获取的Target的列表，并可能成为那些全局选项的容器，然而特定于一个URI的那些选项才应该与Target类共存。此外，错误处理应该得到加强。

但是就这个练习的重点——测试技术使用的一个展示来说，它还是

涵盖了许多内容。首先，这里展示了如何以测试驱动的方式使用静态类型的编程语言来应用这些技术。笔者使用了对于可测试的代码看起来是什么样子的一个理解，来雕琢那些利用正要创建的接缝的测试。当这些测试在运行通过而变绿的时候，使用了重构的阶段来将代码定位到具备更好的可测试性的状态。对于讨论测试实现模式的那些章节，笔者使用了除第13章以外的其他所有章节中的技术。

第15章 遗留的JavaScript代码

Michael Feathers将遗留代码定义为没有测试的代码[WEwLC]。JavaScript和其他动态语言会受益于这些测试，是因为这些测试会执行范围较广的动态行为，而这些行为有时会令测试出现各种意想不到的行为。开源项目得益于测试，是因为测试使得多个贡献者可以在每人不必对软件所有行为都非常熟悉的情况下进行代码修改。

与此相反，不带测试的那些开源JavaScript项目需要具备简单性或者拥有少数完全投入其中的维护者才能存活下来，而且经常需要同时具备上述两个条件才可以。现在，包含测试的开源JavaScript项目越来越多，但是也有一些非常有用的流行项目仍然没有测试。

Trent Richardson编写的jQuery Timepicker Addon[6]就是这样一个项目。Timepicker Addon在jQuery UI 的Datepicker[7]中注入了时间选择。读者可能想到了，这个项目允许在选择日期的时候选择时间。它除了能够提供简单的时间下拉菜单外，还支持时间格式化、时区、滑块、多域、范围和限制等。幸运的是，该项目包含了非常全面的附带众多例子的文档，所以对它进行测试会比较容易。该项目可在GPL或MIT的许可下使用。一段时间以来，Trent一直想为该项目添加测试，一直热忱地支持这项工作，并且将其纳入到了软件的发行中。

可以在 https://github.com/srvance/jQuery-Timepicker-Addon 上找到笔者原项目中的fork，这是一个包含针对本章代码的哈希引用的代码库。本章所引用的代码提交都位于该代码库的“dev”分支中，这是Trent要求放置将要提交的代码的地方。与第14章一样，这里省略了一些实践——如Grunt构建、jslint或jshint及代码覆盖率——笔者一般将它们用在更多

面向生产的上下文中。为了充分利用 IntelliJ 所提供的重构支持，笔者使用了IntelliJ，并启用了JavaScript intentions功能。IntelliJ也提供一定程度的静态分析，来捕获有问题的语言用法。笔者让Chrome浏览器在另一个显示器上保持运行，使用LivePage[8]插件来持续刷新针对该项目的Jasmine[9]spec runner。

15.1 准备开始

测试一块已有的软件会面临一系列与测试正在编写的软件所不同的挑战，尤其是在其大小不定的情况下。当笔者刚开始为Timepicker Addon添加测试的时候，该项目的代码刚刚超过2100行。

首先需要选择从哪里开始。那些可测试的实体——JavaScript 中的那些函数（function），很少有或根本没有依赖关系，所以先从它们开始会比较简单。有了这个立足点，就可以顺利地开展后面的工作了。当然，第一个代码提交仅仅是建立测试框架（见提交[72007f6]）。

笔者选择先从一系列顶层私有函数开始，这些函数位于立即调用的函数表达式（Immediately Invoked Function Expression，IIFE）之中，而IIFE外覆了每一个精心编写的 jQuery 插件。但这些函数是私有的，所以必须决定如何将它们暴露给测试。在JavaScript中，事物一般情况下要么隐藏，要么不隐藏，很少有介于两者之间的情况。然而，一些程序员使用命名约定来将被暴露的函数指定为私有的。Trent 使用一个下划线来把一些函数标记为私有的。笔者遵循了他的风格。代码清单 15-1 显示了 IIFE的相关子集和笔者用来访问函数的技术。

代码清单15-1 为了可测试性在一个IIFE中暴露私有函数

```
(function($) {
    var extendRemove = function(target,props) { ...};
    var isEmptyObject = function(obj) { ...};
```

```
    var convert24to12 = function(hour) { ...};
    var detectSupport = function(timeFormat) { ...};
    $.timepicker._util = {
      _extendRemove: extendRemove,
      _isEmptyObject: isEmptyObject,
      _convert24to12: convert24to12,
      _detectSupport: detectSupport
    };
})(jQuery);
```

让内部方法成为timepicker上的有下划线前缀对象的有下划线前缀的方法，通过这种方式来暴露内部方法，对于非常简单的测试，笔者就能用像下面这样的测试来访问它们：

```
$.timepicker._util._extendRemove(target,props);
```

最初的 19 个测试，一直到代码提交[d0245b6]，修复了两个 bug，并营造了简单和有测试保护的环境，简化了其他代码。

15.2 DOM的统治

只有6行代码的函数selectLocalTimezone()如代码清单15-2所示，它迫使笔者带来一些额外的技术。该函数设置一个选择列表中所被选择的值。该选择列表代表了当前或指定时区所指向的时区选择。这一小段代码使用了jQuery来在DOM上进行操作。

代码清单15-2　一个使用了jQuery来在DOM上进行操作的小方法。timezone_select属性应该是一个引用到＜select＞标签的jQuery集合

```
var selectLocalTimezone = function(tp_inst,date) {
  if (tp_inst && tp_inst.timezone_select) {
    var now = date || new Date();
```

```
            tp_inst.timezone_select.val(-now.getTimezoneOffset());
        }
    };
```

笔者原本建立了 Jasmine spec runner 来包含 jasmine-jquery[10]，但却错把后者与jasmine-fixture[11]给搞混了，所以下一步是纠正这个问题（见提交[f018316]）。jasmine-fixture模块令笔者定义了基于jQuery selector语法的DOM fixture，而jQuery selector 语法会被自动清理。针对上述方法的操作，笔者使用它建立了一个与所期望的Timepicker类似的结构，并做了测试，确保选中的是正确的值（见提交[34e2ee2]），部分代码如代码清单15-3所示。

代码清单15-3　一个像期望的那样使用jasmine-fixture来建立DOM的测试

```
describe('selectLocalTimezone',function() {
    var timepicker,
        timezoneOffset,
        defaultTimezoneOffset;
    beforeEach(function() {
        timepicker = {
            timezone_select: affix('select')
        };
        var now = new Date();
            timezoneOffset = String(-now.getTimezoneOffset());
            defaultTimezoneOffset = String(timezoneOffset - 60);
        timepicker.timezone_select.affix('option')
            .text(defaultTimezoneOffset);
        timepicker.timezone_select.affix('option')
            .text(timezoneOffset);
```

```
        timepicker.timezone_select.affix('option')
            .text(timezoneOffset + 60);
    });
    it('should select the current timezone with a valid ' +
        'timezone_select and a date',function() {
        util._selectLocalTimezone(timepicker,new Date());
        expect(timepicker.timezone_select.val())
            .toBe(timezoneOffset);
    });
});
```

15.3 在牙膏与测试之上

笔者经常将自底向上的测试方法称为牙膏测试法（toothpaste testing）。从底部开始，慢慢地挤到顶端。这种方法与牙膏重构法（toothpaste refactoring）可以比肩，后者遵循同样的模式，并被牙膏测试法所支持。

牙膏重构法为遗留代码拯救法提供了一种替代方法，即使用特征测试来自顶向下进行驱动。两者都有其用武之地。笔者已经在许多场合中见到过一些看似有益的重构（如添加依赖注入或者修改方法签名），通过调用的层级结构向下渗透，只有当遇到某种奇怪的结构时才会停下来。此外，自顶向下的重构在突破一层又一层并对其自身不断进行改变时，要冒一定的风险，即要在缺少测试安全网的情况下持续工作很长一段时间。

自底向上的工作需要某种程度的自信——或者至少是乐观——把对底层的那些实体进行测试看成是有价值的。还必须做大量有关意图的逆向工程。但如果认为整体的设计和实现是健全的，而仅仅少了测试，那

么牙膏法的效果就会很好。

其他用来解析日期和时间字符串的工具函数稍微有些复杂，所以笔者决定将注意力转向公共的$.timepicker函数。从timezoneOffsetNumber()开始，测试发现了一个看似是意图与注释之间有差异的地方（见提交[83b6840]）。

进行到timezoneOffsetString()（见提交[7bc2133]）的时候，发现了一个不平衡的范围验证，即只检查了该范围的上边界，而没检查下边界。又增强了文档来解释代码中的“魔法”数，这个数对应的是最高实际时区和最低实际时区之间的偏移量，以分钟作为单位。一旦测试保护安装到位，就能够消除更多“魔法”数，简化在结果字符串中将数字进行补零操作的代码。

测试log()函数，触发了Jasmine内置spy能力的首次使用（见提交[1adc385]）。Timepicker 的日志记录是一个很薄的外覆层，用来确保在试图通过 console 来写日志之前，该console是存在的。笔者认为这样做是为了避免Firefox的兼容性问题。为了进行测试，笔者必须确定所请求的信息被输出时是否带有所定义的window.console。从技术上讲，无法证明该函数没有输出那条消息。笔者依赖于下述的白盒知识，即它使用了console.log()来验证它没有调用那个方法。幸运的是，可以对这些种类的“系统”函数使用spy，而且使用起来和其他函数一样容易。

将 timezoneAdjust()纳入测试（见提交[b23be92]），能够允许对计算进行安全的简化（见提交[af26afc]）。笔者有些勉强地测试了这个函数。相对于JavaScript的时区处理，其Date对象似乎并不完整。其API省略了修改时区的能力。构造器将时区设置为浏览器的时区，此后该时区就一直保持只读状态。像timezoneAdjust()这样的方法试图通过修改偏移量对分钟进行调整来弥补这个缺陷。这使得大部分日期数学运算能够以务实的方式产生有用的结果。但是从国际日期和国际时间处理的角度看，其基本原则还是有缺陷的。然而，JavaScript 有这样的限制，而且

我们其他人必须要忍受它。

虽然handleRange()函数比较复杂，但是其他范围函数的值能够以正确的方式简单地调用它。笔者使用spy的产品特性独立地测试了这些调用者（见提交[9254a37]）。代码清单15-4显示了这种用法的一个例子。笔者在提交[afb4c98]中将测试重构进一个子组里，并且通过添加一个占位符将handleRange()进行了推迟（见提交[f363202]）。代码清单15-4 使用一个spy来验证一个函数所添加的值，该函数使用正确的参数简单地调用了另一个函数

```
var startTime = $('<p>start</p>'),
    endTime = $('<p>end</p>'),
    options = {};
beforeEach(function() {
  spyOn($.timepicker,'handleRange');
});
it('timeRange calls handleRange the right way',function() {
  $.timepicker.timeRange(startTime,endTime,options);
  expect($.timepicker.handleRange)
    .toHaveBeenCalledWith('timepicker',
      startTime,endTime,options);
});
```

笔者无视面前明确的文档，最初传递了一些字符串来作为startTime和endTime的参数，但是幸运的是IntelliJ对笔者进行了纠正。

15.4 向上扩展

在驯服遗留代码的过程中，其中一个较大的挑战是现有函数的大小和范围。更大的函数会有更多的依赖和更复杂的逻辑。在将较大的函数

重构为较小的、更专注的函数之前，首先需要编写一些特征测试来捕获意图。到目前为止尚未进行测试的那些函数就属于上述这种较大的函数。简单的函数现在已经处理完了，接下来需要迈出更大的步子了。

下一批最复杂的函数是前面所推迟的对日期和时间进行解析的工具函数（utility function）。在余下的 splitDateTime()和 parseDateInternal()这两个函数中，前者更容易一些。首先，后者调用了前者；另外，后者还调用了datepicker的其他函数，包括原始的和覆写的函数。

就如同其他工具函数一样，笔者开始时通过添加_util命名空间来将其暴露（见提交[ff99363]）。最终对正常路径和两个变体的测试很直接。艰巨的部分是错误路径。这段代码包含了一个神秘的注释，即将一个错误信息的格式用作一个黑客所使用的工具（代码原文即如此！）来避免重复解析的逻辑。然而，为了能对其进行测试，在看不出try代码块中的何处能够生成这样一个异常的情况下，笔者只好暂且留下了一个TODO来标记这个悬案。

笔者选择了继续对付 parseDateTimeInternal()，尽管还没有对其所依赖的那些相当复杂的协作者函数进行测试。幸运的是，这些协作者的意图看起来很清楚，并且从每一个协作者那里获得的返回值都是不经额外操作就直接返回的。所以只需要了解如何传递正确的参数来触发所期望的结果。

前两个测试执行了正常路径的一些变体。对于时间解析的路径，笔者选择了当验证微秒不正确时就抛出异常，而这让setup步骤有一点复杂。

错误路径测试引入了一个新的Jasmine特性，即toThrow()匹配器（见提交[1cc967a]）。为了使用这个匹配器（见代码清单15-5），需要稍微有些不同地构造期望结果。

代码清单15-5　使用Jasmine的toThrow()匹配器来验证一个异常错误路径

```
it('should throw an exception if it cannot parse the time',
    function() {
  var inputDateString ='4/17/2008 11:22:33';
    expect(function() {
    $.timepicker._util._parseDateTimeInternal(
      dateFormat,'q',inputDateString,undefined,undefined
    );
  }).toThrow('Wrong time format');
});
```

与其为expect()提供一个值，不如提供一个Jasmine调用的函数，捕获并保存异常以进行匹配。

在测试错误路径的时候发现了一个bug。如果协作者$.datepicker.parseTime()无法解析时间，那么它所使用的底层解析就会返回false。然而，实现代码是使用===操作符来将该函数的返回值与null进行对比判断的。如果对此不熟悉，那么在此简要介绍一下。在JavaScript中，==和===操作符很相似，只是后者还检查其操作数的类型，而前者会很乐意地强制转换这两个值的类型以便查看是否能匹配。虽然在JavaScript中null是falsey[12]的，且当遇到==操作符时会估值为false，但是严格的相等性比较还是会将这两者区别对待。其结果是，上述实现代码中的比较永远不会触发异常。简单地将这个相等性测试（parsedTime === null）改为一个逻辑非（!parsedTime），就能修复这个问题（见提交[c264915]）。

在测试的保护下，笔者可以重构parseDateTimeInternal()来对其进行简化（见提交[353ff5e]）。现有的timeString变量还未使用，所以笔者在其初始化器重复的地方使用它揭示了更多的意图。另外翻转了if-else语句，使其以肯定条件而不是否定条件开始。这样，笔者就可以使用一个尽早返回的卫条件（guard condition）语句来处理无时间的情况，从而

将一些嵌套的条件语句扁平化。经过扁平化处理的函数读起来更整洁。

在离开这些时间工具函数之前，笔者想要多做几次清理工作。splitDateTime()函数返回一个带有两个元素的数组，内含构造好的日期和时间字符串。parseDateTimeInternal()函数引入了一些中间变量，为那些返回数组中的元素提供了能揭示其意图的命名。笔者决定将返回数组转换为一个对象，以便将那些能揭示意图的命名内建到返回值中（见提交[353ff5e]）。这样就能删除掉中间变量了（见提交[5109082]）。

splitDateTime()函数里的头两个变量初始化给笔者留下了冗长和高度相似的印象。它们通过给出被传递过来的设置来确定一个属性，如果这些设置已被定义，其优先级高于 Timepicker 的默认值。提交[11a2545]将初始化逻辑重构到一个共有的函数computeEffectiveSetting()中。当然，这个新函数需要被直接测试（见提交[ea7babe]），但是笔者还是留下了splitDateTime()的所有测试变体，因为它们也代表了有关行为的重要方面。

15.5 软件考古学[13]

测试 splitDateTime()的错误路径的困难，和无法找出异常应该来自何处的事实，一直困扰着笔者。现在是一支考古探险队开始探索该项目历史的时候了。

经过追溯往事，笔者发现该错误处理是在两年前作为另一个函数的一部分而被添加进来的（见提交[4496926]）。另外也注意到try语句块的块体调用了一个协作者，而该协作者不再被现在的代码所涉及。大约一年以前，它被重构到其当前的函数中（见提交[e7c7d40]），但是上述try语句块仍旧调用上述协作者。最后，笔者发现了10个月前的代码，该代码将原来的实现替换成当前的样子（见提交[b838a21]）。当前的代码无法抛出那个其所捕获且试图要处理的异常。这样一来，笔者就能删除

整个错误处理语句块、包围在实现代码（见提交[6adc077]）之外的try/catch 和用来测试已不复存在的错误路径的占位符。这让笔者进而消除了两个splitDateTime()现在未使用的参数。

15.6 回顾

在撰写本书时，笔者尚未完成Timepicker Addon的测试，而是向该项目笔者的fork里push了37个代码提交。另外创建了两个pull request。Trent已经接受了这两个pull request，且将其中一个合并到了dev分支中。笔者计划要完成代码的测试覆盖，并在发现问题时就进行修复和重构。

在这个例子中，我们看到了一个进行测试的不同方法。将未测试代码纳入测试的流程与用测试来驱动一个新项目的流程是不同的。尽管该项目一开始就被编写得相当好，但是笔者还是发现并修复了一些bug，找出了有二义性的意图，出土了残留的代码，清理了警告信息，并且提高了可读性。

该项目表现出在使用一门动态编程语言时，和在使用一种不同风格的测试框架时，所遇到的一些不同之处。针对结果和错误验证，该测试需要使用各种匹配器，并且还要使用spy来进行行为验证。有时候，笔者认为需要一些其他的spy行为，例如返回一个指定值的能力，但是到提交代码的时候才证明想错了。或许最有趣的是，笔者开始探索那些用于测试 JavaScript 的技术，这些技术通过使用 Jasmine 的扩展来与DOM进行交互，创建了一些能自动被拆除的DOM fixture。

---

[1].http://curl.haxx.se/docs/manpage.html

[2].http://hc.apache.org/httpcomponents-client-ga/index.html

[3].RFC 1630最先定义了URL、URN和URI。之后对其进行的各种改进在2005年最终汇集成在撰写本书时正被执行的规范RFC 3986。

[4].spike是一段固定时间的期限，用来研究一个概念和/或创建一个简单的原型。参见 http://en.wikipedia.org/wiki/Scrum_(software_development)。——译者注

[5].http://commons.apache.org/proper/commons-cli/

[6].该项目带有文档和例子的页面参见 http://trentrichardson.com/examples/timepicker/，源代码参见 http://github.com/trentrichardson/jQuery-Timepicker-Addon。

[7].http://jqueryui.com/datepicker/

[8].https://chrome.google.com/webstore/detail/livepage/pilnojpmdoofaelbinaehl=en

[9].http://pivotal.github.io/jasmine/

[10].https://github.com/velesin/jasmine-jquery

[11].https://github.com/searls/jasmine-fixture

[12].在JavaScript中，falsey的值是false、null、undefined、0、NaN和空字符串。所有其他的值都是truthy的，包括空对象和空数组。

[13].软件考古学原文为Software Archeology。——译者注

参考文献

[ASD] Martin,Robert C.Agile Software Development: Principles,Patterns,and Practices.Upper Saddle River,NJ:Prentice Hall,2003.

[AT] Crispin,Lisa,and Janet Gregory.Agile Testing:A Practical Guide for Testers and Agile Teams.Boston,MA:Addison-Wesley,2009.

[CC08] Martin,Robert C.Clean Code: A Handbook of Agile Software Craftsmanship.Upper Saddle River,NJ: Prentice Hall,2009.

[CC11] Martin,Robert C.The Clean Coder: A Code of Conduct for Professional Programmers.Upper Saddle River,NJ:Prentice Hall,2011.

[DP] Gamma,Erich,Richard Helm,Ralph Johnson,and John Vlissides.Design Patterns:Elements of Reusable Object-Oriented Software.Reading,MA:Addison-Wesley,1995.

[GOOS] Freeman,Steve,and Nat Pryce.Growing Object-Oriented Software,Guided by Tests.Boston,MA:Addison-Wesley,2010.

[JCIP] Goetz,Brian,with Tim Peierls,Joshua Bloch,Joseph Bowbeer,David Holmes,and Doug Lea.Java Concurrency in Practice.Boston,MA:Addison-Wesley,2006.

[JTGP] Crockford,Douglas.JavaScript: The Good Parts.Sebastopol,CA: O'Reilly Media,Inc.,2008.

[KOR85]Korel,B.,and J.Laski."A Tool for Data Flow Oriented Program Testing."ACM Softfair Proceedings(Dec 1985):35–37.

[LAS83] Laski,Janusz W.,and Bogdan Korel."A Data Flow Oriented

Program Testing Strategy.”IEEE Transactions on Software Engineering SE-9(3):347–354.

[MEC] Meyers,Scott.More Effective C++: 35 New Ways to Improve Your Programs and Designs.Reading,MA:Addison-Wesley,1996.

[PBP] Conway,Damian.Perl Best Practices.Sebastopol,CA:O’Reilly Media,Inc.,2005.

[REF] Fowler,Martin.Refactoring: Improving the Design of Existing Code.Reading,MA: Addison-Wesley,1999.

[RTP] Kerievsky,Joshua.Refactoring to Patterns.Boston,MA:Addison-Wesley,2005.

[SD] Yourdon,Edward,and Larry L.Constantine.Structured Design:Fundamentals of a Discipline of Computer Program and Systems Design.Englewood Cliffs,NJ:Prentice Hall,1979.

[WEwLC]Feathers,Michael C.Working Effectively with Legacy Code.Upper Saddle River,NJ: Prentice Hall,2005.

[XPE] Beck,Kent.Extreme Programming Explained: Embrace Change.Boston,MA:Addison-Wesley,2000.

[XPI] Jeffries,Ron,Ann Anderson,and Chet Hendrickson.Extreme Programming Installed.Boston,MA:Addison-Wesley,2001.

[xTP] Meszaros,Gerard.xUnit Test Patterns: Refactoring Test Code.Boston,MA:Addison-Wesley,2007.

索引

D

E

F

K

L

M

Z