Broadview
www.broadview.com.cn

PEARSON

Jolt大奖精选丛书

持续集成

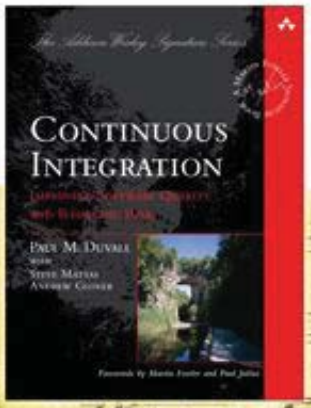
软件质量改进和风险降低之道

Continuous Integration

Improving Software Quality and Reducing Risk

[美] Paul M. Duvall
Steve Matyas
Andrew Glover
著

王海鹏 译



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

持续集成：软件质量改进和风险降低之道

[美] Paul M.Duvall/steve Matyas/Andrew Glover 著

王海鹏 贾立群 译

電子工業出版社
Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

Jolt 大奖素有“软件业之奥斯卡”的美称，本丛书精选自 Jolt 历届获奖图书，以植根于开发实践中的独到工程思想与杰出方法论为主要甄选方向。本书全面深入地讨论持续集成的各个方面，介绍了一种增加项目可见性、降低项目失败风险的有效实践。此外，还介绍了测试驱动、代码审查、数据库集成、信息反馈等实践和工具。全书列举了持续集成系统的优缺点，如何去使用持续集成系统，什么时候使用等，可操作性极强。

本书荣获 2008 年 Jolt 世界图书大奖，适合软件开发人员及团队阅读，还可作为软件工程方面的教材。

图书在版编目（CIP）数据

持续集成：软件质量改进和风险降低之道 /（美）杜瓦尔（Duvall,P.M.），（美）迈耶斯（Matyas,S.），（美）格洛弗（Glover,A.）著；王海鹏译. —北京：电子工业出版社，2012.4
书名原文：Continuous Integration: Improving Software Quality and Reducing Risk
ISBN 978-7-121-14869-9

Ⅰ. ①持… Ⅱ. ①杜… ②迈… ③格… ④王… Ⅲ. ①软件质量－质量管理 Ⅳ. ①TP311.5

中国版本图书馆 CIP 数据核字（2011）第 214922 号

策划编辑：张春雨
责任编辑：董 英
印 刷：
装 订：
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×980 1/16 印张：17 字数：435 千字
印 次：2012 年 6 月第 1 次印刷
印 数：4000 册 定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

经久不息的回荡

今时的读书人，不复有无书可读之苦，却时有品种繁多而无从择优之惑，甚而专业度颇高的技术书领域，亦日趋遭逢乱花迷眼的境地。此时，若得觅权威书评，抑或有公信力的排行榜，可按图索骥，大大增加选中好书的命中率。然而，如此良助，不可多得，纵观中外也唯见一枝独秀——素有“软件业奥斯卡”之美誉的 Jolt 奖！

震撼世界者为谁

在计算设备已经成为企业生产和日常生活之必备工具的今天，专业和大众用户对于软件的功能、性能和用户体验的要求都在不断提高。在这样的背景下，如何能够发挥出软件开发的最高效率和最大效能，已经是摆在每一个从业者面前的重大课题，而这也正是 Jolt 大奖横空出世的初衷及坚持数年的宗旨。

Jolt 大奖历时 20 余年，在图书及软件业知名度极高，广受推崇。奖如其名，为引领计算机科学与工程发展主流，Jolt 坚持将每年的奖项只颁给那些给整个 IT 业界带来震撼结果的图书、工具、产品及理念等，因一流的眼光及超高的专业度而得以闻名遐迩，声名远播。

除图书外，Jolt 针对软件产品设有诸多奖项分类，如配置管理、协作工具、数据库引擎/数据库工具、设计工具/建模、开发环境、企业工具、库/框架、移动开发工具等。但图书历来是 Jolt 大奖中最受瞩目且传播最广的一个奖项分支。Jolt 曾设有通用类图书、技术类图书等分类，每个分类又设有“卓越奖”（Jolt Award，一般为一个）和“生产力奖”（Productivity Award，一般为 2 或 3 个）。

获奖技术图书一经公布，即打上经典烙印，可谓一举“震撼全世界”（赞助商 Jolt 可乐的广告词）。

作为计算机技术图书的厚爱者，我们总在追问——是谁在震撼世界，是谁在照亮明天？Jolt 大奖恰似摆在眼前的橱窗，让我们可以近距离观看潮流在舞蹈，倾听震撼在轰鸣！

朝花夕拾为哪般

Jolt 像是一年一度的承诺，在茫茫书海中为我们淘砺出一批批经得起岁月冲刷的杰作，头顶桂冠的佳作也因而得以一批批引进中国，为国人开阔了眼界，滋补了技术养分。然而，或因技术差距造就的生不逢时、水土不服，或因翻译、制作的不如人意，抑或是疏于宣传等诸多原因，这些经典著作在国内出版后，尽管不乏如获至宝的拥趸，却仍不为诸多人所知，从而与大量本应从中获益的读者擦肩而过。既然这生生错失的遗憾本不该发生，则更不应延续。为此，我们邀国外出版同行、国内技术专家一道，踏上朝花夕拾之路，竭力为广大读者筛选出历久弥新、震撼依旧的 Jolt 图书精品。

Jolt 获奖图书皆由业界专家一致评出，并得到软件从业人员的高度认可，虽然这些书今天读来，不再能看到上世纪史诗时代那般日新月异的理论突破，以及依赖于高深繁复的科学研究所取得的系统化成果，更多是在日复一日的开发实践中总结和提炼出来的工程思想和方法论。重新选材之所以有所弃取，从 Jolt 多年来的评奖规律中可窥端倪——

一万小时真理见

凡是在工程思想领域取得革命性、颠覆性突破的图书，就被归于“震撼”获奖分类。比如，从基于过程的程序设计模型过渡到面向对象的全新模型，就是软件开发思想上的一次带来巨大震撼的革命；再比如，打破传统的瀑布模型而转向持续集成的软件交付模型，这也是一场业界的重大思想转变。像这样的重大思想突破，可以说是数年甚至数十年一遇的，而荣获 Jolt 大奖的图书中更为常见的，则是基于最佳实践的“生产效率”获奖者。获得此类殊荣的图书，都是作者们从平凡的、重复的，甚至用一般人的眼光看来不怎么起眼的日常开

发实践中，以独具的慧眼、过人的耐心和大胆的创新，闯开一条不平常道路的心血与经验总结。

这些图书所涉及的主题，都是普通的软件开发人员每天要面对的工作——代码阅读、撰写测试用例、修复软件问题……但就是这样貌似平淡无奇的工作，是否能每一天、每一个项目都做好，着实拉开了软件开发人员素质的差距，也决定了软件企业开发出来的产品和服务的质量。我们中国有一句古话，叫做熟能生巧；某位著名企业家也说过一句家喻户晓的名言：“把简单的事千百万次地做好，就是不简单的。”这些朴素而实际的真理，同样也是本套丛书最能彰显的所谓程序员精神。它建立在脚踏实地的实践基础之上，也充满了对于自由和创新的向往。

名作可堪比名曲

就不因岁月流逝而褪色来说，与这些 Jolt 名作相媲美者，只有那些百年响彻、震撼古今的经典名曲。希望本丛书带给大家的每部著作，也如百听不厌的乐曲，掩卷良久方余音绕梁，真知存心。仔细想来，软件开发与古典音乐岂非有异曲同工之妙？既是人类心智索问精确科学的探究，亦是寻觅美学享受的追求。工程是艺术的根基，而艺术是工程的极致。衷心地希望各位读者能够认真阅读本丛书的本本珍品，并切实地用于自己的日常工作中，在充分享受大师魅力的同时，为中国的软件事业谱写更多、更震撼的乐章。

电子工业出版社博文视点

二零一二年春

译者序

软件项目开发有两大难题：一是确定软件的需求，即确定目标；二是确定目前离目标还有多远，即确定剩余的工作量。第二个问题就是项目缺少可见性的问题，这对“人月神话”做出了“巨大贡献”。当一个项目经理或一名开发者说已经完成了 80%的任务，您必须保持审慎的乐观。因为剩下的 20%可能还需要 80%的时间，甚至永远也不能完成。您可能迟迟不能拿到可以部署的软件，对此所有的人都无能为力，只能表示深深的遗憾。这确实让专业软件开发者的声誉蒙羞。但是对于大型软件开发这样的复杂工作，我们的经验确实显得有些不够。

早在自顶向下的结构化设计时代，Harlan Mills 就指出，“自顶向下”的含义就在于项目消除了集成困难。如何将诸多知识工作者的工作有效地组织起来，形成一个概念完整的产品，这是越来越多的人们所面对的挑战。软件项目，特别是大型软件项目的高失败率，引起了很多智者的思考。

Grady Booch 说，成功的项目有两个特点：一是具有良好的架构愿景，二是采用迭代增量式的开发过程。越来越多的人认识到，项目的早期就应该验证架构的可行性，得到系统的“可行走的骨架”。持续集成是成功项目不可或缺的实践。

成功的项目，还离不开信任：客户与开发者相互信任，管理层和开发者相互信任，开发者之间相互信任。里根说：“信任，但要检查。”我们也可以反过来说，因为可以随时检查，所以我们信任。因此，采用持续集成策略之后，项目中的信任大为增强。

项目在经过较短的启动阶段之后，就一直可以提交能工作的软件。开发者可以更早听到用户的反馈意见。客户可以根据业务环境和约束条件灵活地决定

下一步的开发重点。系统的架构和功能随着业务的发展而演进。

持续集成也模糊了开发和运营之间的界限。根据具体项目的要求，可以缩短交付间隔，实现持续交付。这对于许多业务来说，是极有价值的。

《持续集成：软件质量改进和风险降低之道》这本书向我们介绍了一种增加项目可见性、降低项目失败风险的有效实践。许多软件开发的资深人士认定，这种方法非常不错。我们不必把宝全部押在最后那一次“大爆炸”式的集成上，而是采用“早集成、常集成”的策略。这样做可以减少缺陷引入和缺陷发现之间的时间，提高开发效率，降低风险。您对项目完成百分比的报告将有更大的信心，而且任何时候，您都可以得到一个可以部署的软件。虽然功能可能还没有全部实现，但它是可用的！

这本书向我们揭示了这样一个道理：如果一件事很难，而您又必须做，不妨经常去做，每次做一点点。其实这也是古老的“分而治之”思想的一种应用。正所谓“滴水穿石，跬步千里”。

敏捷软件开发的许多实践都是互相关联的。持续集成在与其他实践结合时，才能将它的效用发挥到极致。这本书除了介绍持续集成的基本原则和工具之外，也介绍了测试驱动、代码审查、数据库集成、信息反馈等实践和工具。人（思想）、过程和自动化工具的完美结合，将形成一个和谐的开发生态环境。如果您一直在追求效率更高的软件项目管理方法，我相信这本书一定能给您带来一些启发和灵感。

一本好书使您改变。它将改变您的思想，您看待问题的角度和方式，最终，它将改变您的行为。然而，所有具有重要意义的改变都不会在一夜之间发生。改变随时都在发生，但按照您的意志去领导变革却很难。如果您相信这种变革必须发生，不妨朝着这个方向去努力，经常改变，每次改变一点点。

软件业中没有银弹，不可能有某种东西在短时间内让您的开发效率提高 10 倍。但是我们也很容易发现不同人和不同团队之间的开发效率相差巨大，不止 10 倍。那些软件高手和明星团队就像职业围棋选手，他们高得惊人的效率是多年用心改进实践的结果。

参加本书翻译工作的人员除封面署名的外还有：王海燕、李国安、周建鸣、范俊、张海洲、谢伟奇、林冀、钱立强、甘莉萍。在这本书的翻译过程中，我学到了很多，因此郑重地向大家推荐它。如果这本书对于您改进软件开发实践有所帮助，我将十分高兴。

王海鹏

辛卯年夏末于上海

Martin Fowler[1]序

在软件行业发展的初期，软件项目中最棘手、最紧张的时刻就是集成。单独能工作的一些模块被组装在一起，系统整体却常常失败，而且很难找到失败的原因。但在最近几年里，集成基本上已不再是项目中的痛苦之源，而是变成了“小事一桩”。

这种转变的关键在于更为频繁地进行集成。人们曾经认为日构建（daily build）是一个较难达到的目标。但是今天我接触到的项目每天都集成许多次。很奇怪，如果您遇到很痛苦的事情，更频繁地去做这件事似乎是比较好的建议。

关于持续集成，一件有趣的事情就是人们常常会对它产生的影响感到吃惊。我们经常发现人们认为它的好处不大，但它却给项目带来了完全不同的感觉。项目的可见性变得好了很多，因为问题能够更快地检测出来。引入缺陷和发现缺陷之间的时间间隔变短，就更容易发现缺陷，您可以很容易地看见改变了什么，以方便找到问题的根源。当它与良好的测试程序配合时，可以大大减少缺陷的数量。结果是，开发者在调试上花的时间减少了，在增加功能上花的时间更多了，他们相信自己是在一个坚实的基础上开发软件。

当然，光说您应该更频繁地集成是不够的。在这个简单的词语后面有一些原则和实践，正是这些原则和实践使得持续集成变成现实。您可以找到一些建议，它们散布在一些书籍中和 Internet 上（我很自豪，我也在这方面提供过一些内容），但是您必须亲自花力气去寻找。

所以我很高兴看到 Paul 把这些信息收集起来，成为一本完整的书。对于希望执行这些最佳实践的人来说，这是一本参考手册。和许多简单的实践一样，细节之中包含着许多令人烦恼的东西。这些年来，我们已经对这些细节有了许多了解，并学会了如何进行处理。这本书汇集了这些经验，为持续集成奠定了坚实的基础，就像持续集成为软件开发奠定了坚实的基础一样。

---

[1] Martin Fowler 是 ThoughtWorks 公司的首席科学家，在面向对象分析、UML 模式、软件开发方法学等方面都是世界顶级专家，他是著名畅销书《分析模式》、《UML 精粹》和《重构》的作者。

Paul Julius序

我一直希望有人能够抽出时间来写这本书，早写比晚写好。私下里说，我总希望这个人就是我。但是我很高兴 Paul、Steve 和 Andy 最后能够一起完成这本完整的、经过深思熟虑的专著。

我一直投入在持续集成之中，做那些似乎永远也做不完的事情。2001 年 3 月，我和朋友一起创建了开放源代码项目 CruiseControl，并成为项目的管理者。我的日常工作是在 ThoughtWorks 提供咨询，利用持续集成（Continuous Integration，CI）的原则和工具帮助客户设计、构建和部署测试解决方案。

在 CruiseControl 的邮件列表中，2003 年活动开始多了起来。我有机会读到数千个不同的 CI 故事。软件开发者们遇到的问题各不相同，非常复杂。开发者们完成所有这些工作的理由对我来说越来越清楚了。CI 的好处，如快速反馈、快速部署和可重复的自动化测试，要远大于实现 CI 的麻烦。但是，在创建这类环境时却很容易忽视这一点。当我们第一次发布 CruiseControl 时，我从来没想到过人们会通过一些有趣的方式，利用 CI 来改进他们的软件开发过程。

2000 年，我在一个大型的 J2EE 应用程序开发项目中工作，这个项目用到了 J2EE 规范中提供的所有功能。这个应用程序本身就已够让人吃惊，更不必说它的构建了。我所谓的构建，是指编译、测试、打包并执行功能测试。当时 Ant 还处在初期，还没有成为 Java 应用程序构建工具的事实标准。我们使用了一套组合的 shell 脚本来编译所有的东西并执行单元测试，使用另一套 shell 脚本将所有的东西变成可部署的包。最后，我们通过一些手工的步骤来部署 JAR 包并执行功能测试。不用说，这个过程变得费时费力，而且经常容易出错。

从那时起我就希望创建一个可重复的构建过程，只要按“一个按钮”就可以（当时 Martin Fowler 的热门话题之一）。Ant 解决了跨平台的构建脚本的问题。我还想要另外一些东西，能够处理那些烦琐的步骤：部署、功能测试及报告结果。那时，我研究了已有的解决方案，但没有找到我想要的。在那个项目中，我从未找到我所希望的东西。那个应用程序成功地完成了开发并投入使用，但我知道事情还可以做得更好。

在那个项目和下一个项目之间的时间里，我找到了答案。Martin Fowler 和 Matt Foemmel 刚刚发布了他们关于 CI 的第一篇文章。很幸运，我遇到了另一些 ThoughtWorks 的同事，他们正致力于把 Fowler/Foemmel 系统变成可复用的解决方案。我很兴奋，太兴奋了！我知道这就是在上个项目中一直萦绕在我心中的问题的答案。在几周之后，我们已经准备好了所有的东西，并在几个已有的项目中开始使用。我甚至拜访了一个愿意进行 Beta 测试的地方，在一个真正的目标企业中安装了 CruiseControl 的前身。不久之后，我们开放了源代码。对我来说，再也不会回头了。

作为 ThoughtWorks 的一名顾问，我遇到了一些极为复杂的企业级部署结构。我们的客户根据业界的宣传资料承诺的优势，经常希望得到快速的修复。就像所有的技术一样，关于它可以怎样轻易地改变您的企业，实际上存在着一些误导。如果说我从多年的顾问工作中学到了什么，那就是没有什么事情像看起来那么简单。

我想告诉客户如何实际地应用 CI 的原则。我想强调从开发的“韵律”转变到真正享受到那些好处的重要性。如果开发者每个月只签入（check in）一次，不关注自动化的测试，或者并不急于修复失败的构建，那么要完全享受 CI 的好处就很成问题了。

这是否意味着 IT 经理们应该先完成向这些实践的转变，再来碰 CI 呢？不是的。实际上，应用 CI 的实践可以是促成这些改变的最快推进器。我发现，安装像 CruiseControl 这样的 CI 工具会使软件团队变得积极主动，而不是消极被动。这些改变不会在一夜中发生，您必须有正确的预期——包括那些相关的 IT 经理们。通过对底层原理的深入理解，即使是最复杂的环境也可以变得易于理解，易于测试，而且易于很快地投入生产使用。

本书的作者们为您整理好了比赛场地。我发现这本书既全面又深入。这本书深入地讨论了 CI 最重要的方面，它将帮助读者做出理智的决定。书中的各种主题介绍了今天在 CI 领域中运用的各种方法，帮助读者衡量需要进行的折中。最后，我很高兴看到 CI 社区中有这么多的工作被规范化，成为下一步创新的基础。由于这一点，我极力推荐这本书。它是一个重要的资源，利用这些 CI 魔法，可以使得企业级应用程序的复杂配置变得有意思。

前言

在我刚刚参加工作的时候，我看到杂志上有一张整页的广告，展示了键盘上的一个键，类似 Enter 键，上面标着“Integrate（集成）”（参见图 1）。键 下面的文字是“假如一切如此容易。”我已记不清楚这个广告是谁为了什么而做的，但它打动了我的心。在软件开发方面，我曾想，这当然永远不会实现，因为在我们的项目里，我们会花几天的时间在“集成地狱”中挣扎，在接近项目里程碑的时候尝试将大量软件组件拼凑起来。但是我喜欢这个想法，所以我剪下了这张广告，把它贴在我的墙上。对我来说，它代表了我成为一名高效率的软件开发者的主要目标之一：将重复的、容易出错的过程自动化。而且，它包含我的梦想，即将软件集成变成项目中的“小事一桩”（Martin Fowler 曾这样说）——只是自然发生的事情。持续集成（CI）可以帮助您将项目中的集成变成小事一桩。

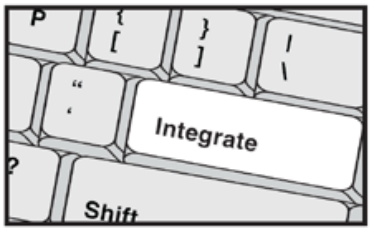


图 1 集成

这本书讲什么

请考虑软件项目中的一些典型的开发过程：编译代码、通过数据库定义数据并操作数据、进行测试、复查代码，最后部署软件。另外，团队肯定需要就软件的状态进行沟通。请想象一下，如果您可以按一个键就完成这些过程。

这本书向您展示了如何创建一个虚拟的集成按钮，将许多软件开过过程都自动化。而且，我们介绍了如何持续地按下这个按钮，从而减少创建可部署的应用程序时的风险，如较晚才发现缺陷、低品质的代码等。在创建 CI 系统时，许多过程都被自动化，在每次修改开发的软件时，都执行这些过程。

什么是持续集成

集成软件的过程不是新问题。在一个人开发的项目中，依赖外部系统又比较少的话，软件集成不会成为太大的问题，但是随着项目复杂度的增加（即使只增加一个人），就会对集成和确保软件组件能够一起工作提出更多的要求——要早集成，常集成。等到项目快结束时才来集成会导致各种各样的软件品质问题，解决这些问题代价很大，常常会导致项目延期。CI 以较小增量的方式迅速地解决这些风险。

在Martin Fowler的热门文章“Continuous Integration”[1]（持续集成）中，他将CI描述为：

……一种软件开发实践，即团队的成员经常集成他们的工作，通常每个成员每天至少集成一次——这导致每天发生多次集成。每次集成都通过自动化的构建（包括测试）来验证，从而尽快地检测出集成错误。许多团队发现，这个过程会大大减少集成问题，让团队能够更快地开发出一致的软件。

根据我的经验，这意味着：

- 所有开发者都先在他们自己的工作站上执行私有构建[2]，然后再将他们的代码提交到版本控制库中，从而确保他们的变更不会导致集成构建失败。
- 开发者每天至少向版本控制库提交一次代码。
- 集成构建每天在一台独立的计算机上进行多次。
- 每次构建都必须 100%通过测试。
- 生成可以进行功能测试的产品（如 WAR、配件、可执行程序等）。
- 修复失败的构建是优先级最高的事情。
- 某些开发者复查构建生成的报告，如编码标准报告和依赖分析报告，寻找可以改进的地方。

---

[1] 参见 www.martinfowler.com/articles/continuousIntegration.html。

[2] 私有（系统）构建和集成构建模式在 Stephen P. Berczuk 和 Brad Appleton 写的 *Software Configuration Management Patterns* 一书中有介绍。

本书讨论了 CI 中自动化的方面，因为您会从自动化重复的、容易出错的过程中得到许多好处。但是，正如 Fowler 所指出的，CI 就是经常集成工作的过程，这不一定需要自动化的过程。我们相信，既然已经有许多工具让 CI 成为自动化的过程，那么使用 CI 服务器来自动化 CI 实践就是一种有效的方式。不管怎样，也许手工集成的方式（利用自动化的构建）很适合您的团队。

快速反馈

持续集成增加了您获得反馈信息的机会。这样，您每天都能多次了解项目的状态。CI 可以用来减少引入缺陷和修复缺陷之间的时间间隔，从而改进软件的总体品质。

开发团队不应该相信因为 CI 系统自动化了，就可以避免集成问题。如果团队只使用自动化的工具来编译源代码，就更是如此。有些人把编译称为“构建”，实际上不是的（参见第 1 章）。有效的 CI 实践包含的东西比工具多得多。它包括本书中介绍的实践，如经常向版本控制库提交代码，立即修复失败的构建，以及使用独立的集成构建计算机等。

CI 的实践支持快速反馈。如果应用了有效的 CI 实践，您可以每天多次了解到正在开发的软件的健康状况。而且，CI 与重构、测试驱动开发等实践配合得挺好，因为这些实践的中心思想都是进行小的变更。从本质上来说，CI 提供了一张安全网，确保变更能够与软件的其他部分一起工作。从更高的层面上讲，CI 增加了团队整体的信心，减少了项目所需的人工，因为它通常是一个无人值守的过程，在软件发生变更时执行。

关于“持续”的注释

我们在本书中使用“持续”这个术语，但是这种用法从技术上来说是不对

的。“持续”意味着某事一旦启动就不会停止。这意味着集成过程一直在执行，但是即使对于最密集的 CI 环境来说，也不是这样的。所以，我们在这本书中讲述的更像是“经常集成”。

谁应该读这本书

在我的经验中，把软件开发当成工作的人和把它当成职业的人之间有着显著的差别。本书是为那些把软件开发当成职业，并发现自己在做一些重复的过程的人（或者我们将帮助您意识到您正频繁地这样做）。我们描述 CI 的实践和好处，让您知道如何应用这些实践，这样您就可以将时间和专业知识用在更重要、更有挑战性的问题上。这本书介绍了与 CI 相关的主要话题，包括如何利用持续反馈、测试、部署、审查和数据库集成来实现 CI。不论您在软件开发中承担哪种角色，您都可以将 CI 纳入到自己的软件开发过程之中。如果您是一位软件开发专家，希望变得更有效率（在您拥有的时间里做更多的事情或得到更可靠的结果），您会从本书中学到很多东西。

开发者

如果您已注意到自己宁愿为用户开发软件而不是与软件集成问题搏斗，那么这本书将帮助您消除原来的大部分“痛苦”。这本书不会要求您花更多的时间来集成，它是讲如何让大部分的软件集成工作变成“小事一桩”，让您能够关注最喜欢做的事情：开发软件。这本书中的许多实践和例子向您展示了如何实现一个有效的 CI 系统。

构建/配置/发布管理

如果您的工作是让能工作的软件发布出去，您会发现这本书特别有意思，因为我们在书中展示，通过在每次对版本控制库进行变更时执行一些过程，您

可以生成一致的、能工作的软件。可能许多人在管理构建的同时还承担项目中的其他角色，如开发。CI 将替您完成一些“思考”，不必等到开发生命周期的末尾，它每天都能多次生成可测试的软件。

测试者

CI 为软件开发提供了快速的反馈信息，消除了过去即使进行了“修复”之后，缺陷还会再次出现的痛苦。在实现了 CI 的项目中，测试者通常会提高满意度，并提高对他们的角色的兴趣，因为送来测试的软件更为频繁，每次要测试的范围也较小。在开发生命周期中使用 CI 系统之后，您的测试是一直进行的，而不是像通常那样有时忙得要死，有时又闲着没事。以前测试者要么工作到很晚，要么又没有东西可测试。

经理

对于团队一致地、重复地交付工作软件的能力，如果您希望有更大的信心，那么这本书将给您带来巨大的冲击。您可以更有效地管理时间、费用和品质，因为您决策的基础是能工作的软件、真实的反馈信息和测量指标数据，而不是项目进度计划上的任务项。

本书的组织结构

本书分为两个部分。第一部分介绍了 CI，从头开始讨论了 CI 的概念和实践。第一部分针对的是那些不熟悉 CI 的核心实践的读者。但是，如果没有第二部分，这些 CI 的实践是不完整的。第二部分自然地将核心概念扩展为 CI 系统执行的有效过程，包括测试、审查、部署和反馈等。

第一部分　CI 的背景知识：原则与实践

第 1 章，启程，让您通过一些例子了解如何利用 CI 服务器来持续构建软件。

第 2 章，引入持续集成，让您熟悉常见的实践方法，知道如何开始 CI。

第 3 章，利用 CI 减少风险，通过场景式的例子说明 CI 可以缓解的主要风险。

第 4 章，针对每次变更构建软件，探讨了利用自动化的构建，在每次变更时集成软件。

第二部分　创建全功能的 CI 系统

第 5 章，持续数据库集成，进入一些更高级的概念，涉及构建数据库和应用测试数据等过程，作为每次集成构建的一部分工作。

第 6 章，持续测试，介绍了在每次集成构建时测试软件的概念和策略。

第 7 章，持续审查，介绍了利用不同的工具和技术进行自动化的、持续的审查（静态和动态分析）。

第 8 章，持续部署，探讨了利用 CI 系统部署软件的过程，以便于进行功能测试。

第 9 章，持续反馈，向您展示了如何利用持续反馈设备（如电子邮件、RSS、X10 及 Ambient Orb），这样您在构建成功或失败时就能收到通知。

“尾声”探讨了 CI 将来的可能性。

附录

附录 A，CI 资源，包含了与 CI 有关的 URL、工具和文章的列表。

附录 B，评估 CI 工具，评估了市面上不同的 CI 服务器和相关的工具，讨论了它们支持本书中描述的哪些实践，指出了每种工具的优点和不足，解释了如何利用它们的一些有趣的功能。

其他特点

本书还包括了一些其他特点，帮助您更好地理解和应用书中的内容。

- **实践**——在这本书中，我们介绍了 40 多个 CI 相关的实践。许多章的副标题就是这些实践。在多数章的开始有一张图，说明了这一章要介绍的实践，让您能够方便地寻找感兴趣的内容。例如，“使用专门的集成构建计算机”和“经常提交代码”就是本书中讨论的实践。
- **示例**——我们通过许多不同语言和平台上的例子，展示了如何应用这些实践。
- **问题**——每一章都包含了一个问题列表，帮助您评估项目中 CI 实践的应用情况。
- **Web 站点**——本书的配套 Web 站点 www.integratebutton.com 提供了本书更新、代码示例和其他材料。

您将学到什么

通过阅读这本书，您将学到一些概念和实践，它们能帮助您每天多次创建一致的、能工作的软件。我们首先关注这些实践，然后是这些实践的应用，在可能的时候我们都提供了示例来说明。这些示例使用了不同的开发平台，如 Java、Microsoft.NET，甚至还有 Ruby。CruiseControl（Java 版和.NET 版）是本书中主要使用的 CI 服务器，但是，我们在配套网站（www.integratebutton.com）和附录 B 中提供了使用其他服务器和工具的例子。

当您从头到尾阅读这本书时，您会有下面的发现：

- 实现 CI 如何能够做到在开发生命周期中的每一步都生成可部署的软件。
- CI 如何能够减少缺陷引入和缺陷被发现之间的时间间隔，从而降低修复缺陷的成本。
- 通过经常构建软件，而不是等到开发的后期再构建软件，如何能够提

高软件的品质。

本书没有介绍什么

本书没有介绍构成 CI 系统的全部的工具——构建进度计划、编程环境、版本控制等。它关注的是实现 CI 实践，从而得到一个有效的 CI 系统。首先讨论的是 CI 实践，如果书中提到的某个工具不再使用，或者不能满足您的特别需要，只要使用另一个工具来应用这些实践就行了，目的是达到同样的效果。

本书也不可能介绍 CI 所使用的每一种类型测试、反馈机制、自动化审查工具和部署的类型。即使这样做，用处也不大。通过关注关键实践，利用技术和工具的示例来讲解数据库集成、测试、审查和反馈。项目和团队可以根据自己的理解进行不同的应用。我们希望这样可以实现更宏大的目标。正如这本书中处处提到的，这本书的配套 Web 站点 www.integratebutton.com 包含使用其他工具和语言的例子，这些例子在本书中可能没有提及。

关于作者

本书有三位作者和一位贡献者。我编写了大部分章节。Steve Matyas 参与了第 4、5、7、8 章和附录 A 的编写，并提供了本书中的一些例子。Andy Glover 编写了第 6、7、8 章，提供了例子，并参与了本书其他部分的编写。Eric Tavela 编写了附录 B。这样在读到使用第一人称的句子时，您可以知道是谁在发表观点。

关于封面

当我得知我们的书将作为著名的 Martin Fowler 签名系列中的一本时，我非常兴奋。我知道这意味着我可以挑选一座桥作为这本书的封面。其他几位作者和我都是在华盛顿特区长大的。如果你不是来自这个地区的，可能不知道这个地区是变化很快的。更准确地说，我们来自北弗吉尼亚，所以觉得选择弗吉尼亚的“天然桥”作为封面是很不错的。我直到 2007 年初才去了那里——在

我将它选为封面之后。它有一段有趣的历史，我觉得难以置信，它竟然能每天让汽车从上面开过（当然，我也开车从上面走过几次）。我希望在阅读了这本书之后，您会将 CI 作为下一个软件开发项目中的自然组成部分。

致谢

我说不清楚有多少次在读书时看到作者说“单靠自己是无法完成的”和其他一些事情。我总是对自己说：“他们只是在假谦虚。”可是，我确实错了。这本书是一个浩大的工程，我要感谢这里列出的人。

我要感谢我的出版商，Addison-Wesley。我特别要感谢我的责任编辑 Chris Guzikowski，他和我一起度过了这个耗费心血的过程。他的经验、观点和鼓励对我帮助非常大。而且，我的项目编辑 Chris Zahn，在几个版本和几轮编辑中提供了中肯的建议。我还要感谢 Karen Gettman、Michelle Housley、Jessica D’Amico、Julie Nahil、Rebecca Greenberg，以及我的第一位责任编辑 Mary O’Brien 和其他人。

Rich Mills 为本书提供了 CVS 服务器，并在“头脑风暴”会议上提出了绝妙的想法。我也要感谢我的指导者和朋友 Rob Daly，2002 年让我开始职业写作，并在我写作的过程中进行了详细的复查。John Steven 对促成本书的编写也起了帮助作用。

我想感谢我的合作者、编辑和贡献者。Steve Matyas 和我度过了许多不眠之夜，创作您读到的这本书。Andy Glover 是我们抓过来的作者，他提供了大量有关项目的开发者测试的经验。Lisa Porter 是我们的特约编辑，她不知疲倦地检查每一个版本，进行编辑并提供建议，提高了这本书的品质。我要感谢 Eric Tavela，他写了 CI 工具的附录，感谢 Levent Gurses 在附录 B 中提供了 Maven 2 的经验。

我们有一个平衡的技术复查者核心团队，他们在本书的编写过程中提供了很好的反馈意见。他们是 Tom Copeland、Rob Daly、Sally Duvall、Casper Hornstrup、Joe Hunt、Erin Jackson、Joe Konior、Rich Mills、Leslie Power、David

Sisk、Carl Tallis、Eric Tavela、Dan Taylor 和 Sajit Vasudevan。

我还要感谢 Charles Murray 和 Cristalle Belonia 的帮助，以及来自 Urbancode 公司的 Maciej Zawadzki 和 Eric Minick 的帮助。

我要感谢几个很好的人，我在 Stelligent 公司工作的每一天里，他们对我提供了支持和帮助。他们是 Burke Cox、Mandy Owens、David Wood 和 Ron Wright。还有许多人这些年在我的工作中给了我启发，他们是 Rich Campbell、David Fado、Mike Fraser、Brent Gendleman、Jon Hughes、Jeff Hwang、Sherry Hwang、Sandi Kyle、Brian Lyons、Susan Mason、Brian Messer、Sandy Miller、John Newman、Marcus Owen、Chris Painter、Paulette Rogers、Mark、Simonik、Joe Stusnick 和 Mike Trail。

我也感谢 Addison-Wesley 的技术复查团队提供的反馈信息，包括 Scott Ambler、Brad Appleton、Jon Eaves、Martin Fowler、Paul Holser、Paul Julius、Kirk Knoernschild、Mike Melia、Julian Simpson、Andy Trigg、Bas Vodde、Michael Ward 和 Jason Yip。

我想感谢参加了 CITCON 芝加哥 2006 年会议的人，他们和我们分享了在 CI 和测试方面的经验。我特别要感谢 Paul Julus 和 Jeffrey Frederick 组织了这次会议，以及其他所有参加了这次会议的人。

最后，我要感谢 Jenn 在本书编写的日子里始终提供了坚定的支持。

Paul M. Duvall

于弗吉尼亚州费尔费克斯县

2007 年 3 月

作者简介

Paul M. Duvall 是 Stelligent 公司的 CTO。Stelligent 公司是一家咨询公司，他们通过优化软件开发过程，帮助开发团队可靠地、快速地开发出更好的软件。他几乎担任过软件开发项目中的所有职务，从开发者到测试者再到架构师和项目经理。Paul 向各个行业的客户提供咨询，包括金融业、房地产业、政府、医疗卫生业，以及大型的独立软件提供商。他是许多知名软件会议的特邀讲演者。他为 IBM developerWorks 撰写了一系列的文章，名为“Automation for the People”，他是 *NFJS 2007 Anthology*（Pragmatic Programmers，2007）的合著者，也是 *UML 2 Toolkit*（Wiley，2003）的贡献作者。他是临床研究数据管理系统和方法的发明者之一，这个系统和方法正在申请专利。他经常在 www.testearly.com 和 www.integratebutton.com 上写日志。

Stephen M. Matyas III 是 AutomateIT 的副总裁。AutomateIT 是 5AM Solutions 公司的一个服务机构，它帮助组织机构通过自动化来改进软件开发。Steve 在应用软件工程方面有多重背景，他的客户包括商业客户和政府客户。Steve 担任过许多种不同的角色，从业务分析师和项目经理到开发者、设计者和架构师。他是 *UML 2 Toolkit*（Wiley，2003）的贡献作者。他实践了许多迭代增量式的方法，包括敏捷方法和 Rational Unified Process（RUP）。他的大部分第一手的职业经验来自于 Java/J2EE 定制软件开发和服务，在方法学、软件品质和过程改进方面具有特珠的要求。他拥有弗吉尼亚理工大学的计算机科学学士学位。

Andrew Glover 是 Stelligent 公司的总裁。Stelligent 公司是一家咨询公司，他们通过优化软件开发过程，帮助开发团队可靠地、快速地开发出更好的软件。Andy 经常在北美的各种会议上作为讲演嘉宾，他也是 No Fluff Just Stuff Software Symposium 小组的讲演者。他还是 *Groovy in Action*（Manning，2007），*Java Testing Patterns*（Wiley，2004）及 *NFJS 2006 Anthology*（Pragmatic Programmers，2006）的合著者之一。他也是一些在线文章的作者，这些文章发布在 IBM 的 developerWorks，O’Reilly 的 ONJava、ONLamp 和 Dev2Dev 门户站点上。他经常在 www.thediscoblog.com 和 www.testearly.com 上写有关软件品质的日志。

贡献者简介

Lisa Porter 是一个顾问团队的高级技术作者，这个团队向美国政府提供网络安全的解决方案。Lisa 在本书出版之前提供了技术编辑服务。她早些年曾支持一个包含多个应用程序的大型软件开发项目，在需求确定和项目成熟度/能力等活动方面受到了好评。她也进行外语翻译和架构/工程方面的技术写作。Lisa 从 2002 年开始就从事编辑书籍和在线出版物的工作。

Eric Tavela 是 5AM Solutions 公司的首席架构师。5AM Solutions 公司是一个软件开发公司，该公司致力于将软件工程的最佳实践应用于生命科学的研究工作。Eric 的主要工作背景是实现 Java/J2EE 应用程序及指导面向对象软件开发和 UML 建模。

目录

第 2 部分 创建全功能的 CI 系统

第 6 章

持续测试

自动化单元测试 | 自动化组件测试 | 自动化系统测试

自动化功能测试 | 对开发者测试分类 | 先执行较快的测试

为缺陷编写测试 | 让组件测试可重复 | 将测试用例限制为一个断言

实践造就完美。

——英语格言

re·li·a·ble —形容词—在连续的试验中给出同样的结果。[1]

系统工程有一项原则，即线性系统的可靠性是每个系统组件的可靠性的乘积。例如，假设一个系统由三个组件组成，如图 6-1 所示。

图 6-1　具有三个组件的系统

[1] 摘自 www.m-w.com/cgi-bin/dictionary?va=reliable。

这个示例系统中的组件的可靠性都进行了测量，每个都是 90%（不管这个 90%是怎样得到的）。如果您不是一名系统工程师，您可能认为整个系统的可靠性就是 90%。但是，这个答案是不正确的：0.90×0.90×0.90 的结果实际上是 0.73。系统的整体可靠性是 73%。

您是否开车驶过可靠性为 73%的大桥？如果您有一支钢笔，只有 73%的时候能写出字，您不会扔掉它吗？我们假定我们开车路过的绝大多数大桥都是 100%可靠的，绝大多数钢笔在墨水用完之前也是 100%可靠的。为了获得这样的可靠性，大桥的建造者和钢笔的制造者必须从最低层开始确保组件的可靠性，因为这才是确保整体可靠性的唯一方式。

这就是为什么 20 世纪 70 年代日本汽车的销售开始超过美国汽车的原因。日本制造商发现并应用了这一原则，结果日本制造的汽车比美国竞争对手的产品要可靠得多。日本制造商认识到他们必须从最低层开始确保可靠性。

下面请设想一个软件系统（顺便提一句，软件系统是非线性的——这基本上意味着您必须还要考虑每个组件之间的接口或连接组件的可靠性）。可能我们都没有开发过像图 6-1 那样只有三个组件（如对象）的软件系统。大部分软件系统如果没有几千个对象，也有几百个对象。对于包含 100 个组件的线性系统来说，如果每个组件的可靠性是 99%，整个系统的可靠性就只有 37%。

如果您想构建一个软件系统，它在服务层面的承诺达到 100%（或接近），您绝对需要在单个对象的层面上确保可靠性。如果您不能从最低层确保并测量可靠性，您就不可能在系统层面上达到要求。然而这正是我们整个行业一直以来构建和交付软件的主要方式。设计、构建，然后抛给质量保证（QA）小组。QA 小组在系统层面上进行测试，然后不可避免地发现许多缺陷。到了一定的时候，我们就向客户宣布这个系统，他们肯定也会发现缺陷，这不奇怪，有时候会降低公司利润，或者毁坏你的声誉，或者同时发生。

所以作为底线，如果我们要构建真正可靠的软件系统，我们就必须确保对象层面的可靠性，这只能通过成功的单元测试来实现。否则，我们就不能期望

构建出高可靠性的应用程序。当然，只是为对象编写单元测试不一定能保证可靠性。测试必须有效地使用了被测对象，而且必须经常执行。

因为软件系统中的对象相互之间进行通信，每次当系统发生变更时，测试都必须执行。在 CI 系统中包含持续的测试让您能够做到这一点。图 6-2 展示了我们在实现完整的自动化构建和 CI 系统时，持续测试所处的位置。

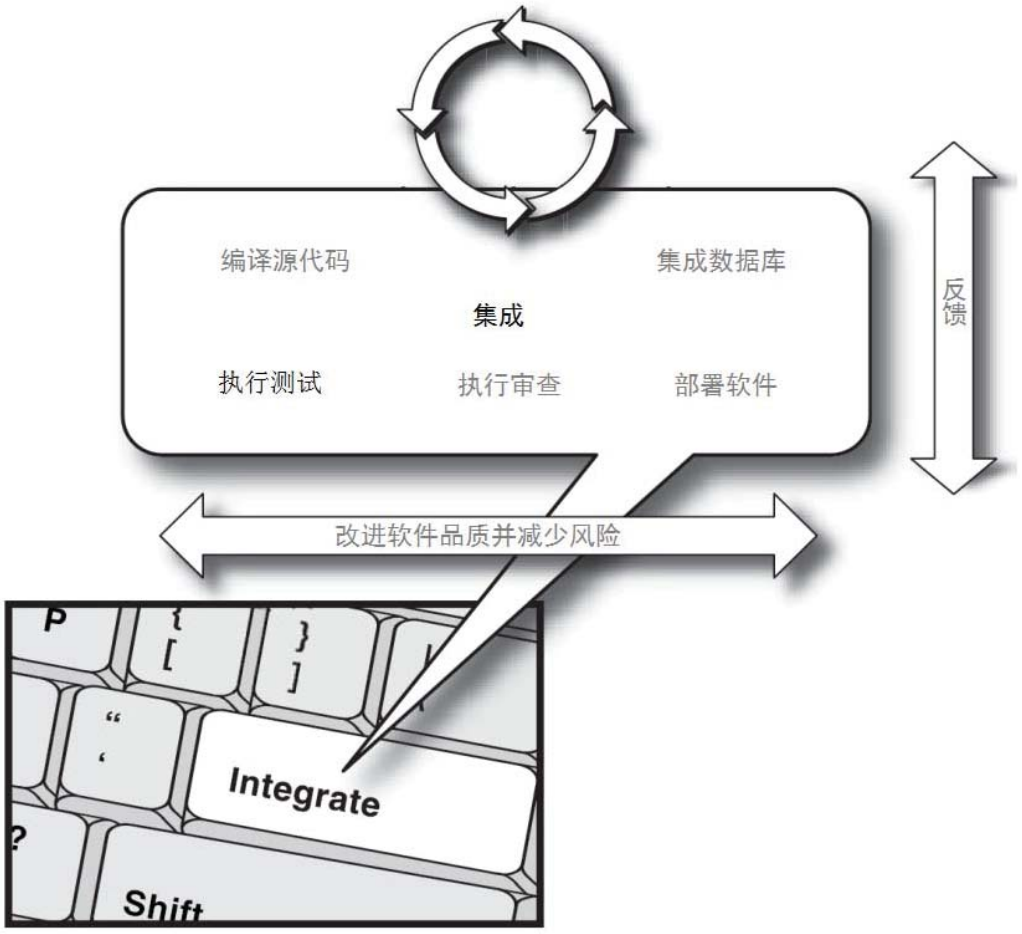


图 6-2 集成按钮——执行自动化的开发者测试

6.1 自动化单元测试

人们在使用“单元测试”这个术语时比较随意。这可能会导致困惑，特别是当人们开始声称他们的单元测试“花了很长的时间执行”时。为开发者测试定义一个常用词汇表有助于将测试进行有效的分类，从而创建一个有效的 CI 系统，能够执行快速的构建。

“单元测试”验证软件系统中所有小元素的行为，这些小元素通常都是一个类。但是有时候，单元测试和被测试的类之间的这种一对一的关系会被放大，因为一些被测试的类耦合程度较高。

代码清单 6-1 展示了利用 TestNG 框架编写的单元测试。TestNG 是基于 annotation 的，因此@testng.test 这样 Javadoc 一样的注释出现在了 startPatternTest 方法中。通过 Java1.4 的 assert 语句，这个测试用例验证了 RegexPackageFilter 类通过一个正则表达式模式正确地过滤了字符串。

代码清单 6-1　使用 TestNG 的隔离单元测试

```
public class RegexPackageFilterTestNG {
    /**
     * @testng.test
     */
    public void starPatternTest() throws Exception{

        Filter filter = new RegexPackageFilter("java.lang.*");

        assert filter.applyFilter("java.lang.String"):
            "filter returned false";

        assert !filter.applyFilter("org.junit.TestCase"):
            "filter returned true for org.junit.TestCase";
    }
}
```

某些单元测试需要较少的外部依赖关系，这些依赖关系通常是其他的类。这些依赖的类本身比较简单，没有很复杂的类间关系。有时候，单元测试甚至使用模拟对象（mock），它们是一些简单的对象，用于替换真实的、复杂的对象。如果依赖的对象本身确实依赖于外部的实体，如一个文件系统或数据库，而这些外部对象又没有虚拟化，测试就变成了组件测试（后面定义）。

代码清单 6-2 展示了一个用 Ruby 写的单元测试的例子，它验证了一个过滤器的行为。这个测试仍然是一个单元测试，虽然它使用了两个类，RegexFilter 和 SimpleFilter，因为它只使用了一个类型来验证行为。

代码清单 6-2　使用 Ruby 的隔离单元测试

```
require "test/unit"
require "filters"

class FiltersTest < Test::Unit::TestCase

  def test_regex
    fltr = RegexFilter.new(/Google|Amazon/)
    assert(fltr.apply_filter("Google"))
  end

  def test_simple
    fltr = SimpleFilter.new("oo")
    assert(fltr.apply_filter("google"))
  end

  def test_filters
    fltrs = [SimpleFilter.new("oo"), RegexFilter.new(/Go+gle/)]
    fltrs.each{ | fltr |
      assert(fltr.apply_filter("I love to Gooogle on the Internet"))
    }
  end
end
```

单元测试的关键在于没有外部的依赖关系，如数据库。这些外部的依赖关系通常会使测试建立和执行的时间变长。单元测试可以在开发周期的早期创建并执行（例如第一天）。因为编码和看到单元测试结果之间的时间很短，所以单元测试是一种有效的除错方法。

6.2 自动化组件测试

“组件测试”或“子系统测试”验证系统的各个部分，可能需要安装整个系统或某些外部依赖关系，如数据库、文件系统或网络终端等。这些测试验证组件间的交互能产生预期的组合行为。典型的组件测试需要底层数据库支持，甚至可能跨越架构边界。因为每个测试用例执行的代码量更大，每个测试的代码覆盖率也更大，所以这些测试一般比单元测试执行的时间长。

代码清单 6-3 展示了一个组件测试的例子，它利用 DbUnit 框架来生成数据库中的基底数据，然后尝试基于数据库中的内容找到数据。DbUnit 使用了一些 XML 文件，它读入这些文件并将对应的数据插入到匹配的数据库表中。

代码清单 6-3　使用 DbUnit 的组件测试

```
public class DefaultWordDAOImplTest extends DatabaseTestCase {
    protected IDataSet getDataSet() throws Exception {
       return new FlatXmlDataSet(new File("test/conf/wseed. xml"));
    }

   protected IDatabaseConnection getConnection() throws Exception {
        final Class driverClass =
              Class.forName("org.gjt.mm.mysql.Driver");
        final Connection jdbcConnection =
        DriverManager.getConnection(
           "jdbc:mysql://localhost/words",
           "words", "words");
         return new DatabaseConnection(jdbcConnection);
    }

    public void testFindVerifyDefinition() throws Exception{
        final WordDAOImpl dao = new WordDAOImpl();
        final IWord wrd = dao.findWord("pugnacious");
        for(Iterator iter =
              wrd.getDefinitions().iterator();
                                iter.hasNext();){
           IDefinition def = (IDefinition)iter.next();
           TestCase.assertEquals(
                "def is not Combative in nature; belligerent.",
                "Combative in nature; belligerent.",
                def.getDefinition());
        }
    }
    public DefaultWordDAOImplTest(String name) {
        super(name);
    }
}
```

组件级的测试比单元测试用到更多的依赖关系，但不一定像更高级的系统测试（稍后定义）用到的那么多。组件级的测试通过 API 来执行代码，但这些

API 可能暴露给客户，也可能不暴露给客户。在代码清单 6-3 中，主要通过暴露出的接口测试了数据访问对象（DAO）层的一个对象。另一个组件测试的例子通过 StrutsTestCase 框架测试了 Struts 架构中一个动作类，如代码清单 6-4 所示。这个测试明显需要一个数据库才能执行，但 Web 容器则是虚拟的，执行的 API 也不必暴露给客户。

代码清单 6-4 使用 StrutsTest 的组件测试

```
public class ProjectViewActionTest extends DeftMeinMockStrutsTestCase {
   public void testProjectViewAction() throws Exception {
      this.addRequestParameter("projectId", "100");
      this.setRequestPathInfo("/viewProjectHistory");
      this.actionPerform();
      this.verifyForward("success");

        Project project = (Project)this.getRequest()
          .getAttribute("project");
           assertNotNull(project);
           assertEquals(project.getName(), "DS");
  }

  protected String getDBUnitDataSetFileForSetUp() {
      return "dbunit-seed.xml";
  }

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

在代码清单 6-4 中，StrutsTestCase 框架与 DbUnit 一起提供了向数据库填充基底数据的功能和一个虚拟的容器。DeftMeinMockStrutsTestCase 类是一个模板，它要求实现 getDBUnitDataSetFileForSetUp 方法。

这种类型的测试也被称为“集成测试”。这种类型的测试与系统测试的不同之处在于，集成测试（或组件测试、子系统测试）并不总是执行那些期望公开的 API。例如，系统测试可以通过 Web 应用程序的 Web 页面执行它，但组件测试可以执行应用程序 Web 页面底下的业务层。

6.3　自动化系统测试

“系统测试”运行整个软件系统，因此需要完整地安装系统，如 servlet 容器和相关的数据。这些测试验证外部的接口，如 Web 页面、Web 服务及 GUI 能够像设计的那样工作。系统测试可能需要较长的执行时间和准备时间。但如果您已成功地执行了自动化的单元测试和组件测试，就已经事先解决了一些底层的问题，只要计划定期执行这个耗时较长的测试就可以了，也许作为次级集成构建的一部分，甚至在下班以后，例如夜间执行。

系统测试与“功能测试”有着根本的区别，功能测试是按照更像客户使用系统的方式来测试系统的。例如，在代码清单 6-5 中，测试模拟了一个浏览器，通过HTTP来操作网站。但是，这个测试并没有使用浏览器。像Selenium[2]这样的框架会驱动一个浏览器，可以用于创建功能测试。在完成了自动化的系统测试之后，您仍需执行自动化和手工的功能测试——这两者之间不能互相代替。

代码清单 6-5　使用 JWebUnit 的系统测试

```
public class LoginTest extends WebTestCase {

  protected void setUp() throws Exception {
     getTestContext().
         setBaseUrl("http://pone.acme.com/meinst/");
  }

  public void testLogIn() {
    beginAt("/");
    setFormElement("j_username", "aader");
    setFormElement("j_password", "a1445");
    submit();
    assertTextPresent("Logged in as aader");
  }
}
```

[2] Selenium 是一个基于 Web 的、跨浏览器的功能测试工具，可以从 www.openqa.org/selenium/获得。

代码清单 6-5 包含了一个 JWebUnit 测试用例的例子，它尝试登录一个网站，然后验证这次尝试是否成功。虽然在代码里看起来并不明显，但是必须安装并运行整个系统（sevlet 容器和数据库），这个测试用例才能执行。请注意，安装的过程并不在这个测试用例中，而是作为构建的一个较大部分的工作完成的。

6.4 自动化功能测试

“功能测试”正如其名，它从客户的视角来测试应用程序，这意味着测试将模仿客户的行为。这些测试也被称为“验收测试”。

正如前面曾提到的，像 Selenium 这样的框架实际上控制了一个浏览器，让它与 Web 站点交互。Selenium 测试是以表格的形式编写的，它代表了一个工作流程，其中填入了命令和断言。代码清单 6-6 是 Selenium 的测试用例，它尝试登录一个 Web 站点，然后验证登录是否成功。

代码清单 6-6 使用 Selenium 的功能测试

TestLoginSuccess		
open	/ib/app	
verifyTitle	Integrate Button - Welcome	
verifyTextPresent	Welcome to The IntegrateButton.com. Please log in to access exclusive material for the book.	
clickAndWait	link=Log In	
type	inputUserId	admin
type	inputPassword	admin
clickAndWait	loginSubmit	
assertTextPresent	Logout	
clickAndWait	Link=Logout	
assertTextPresent	Log In	
verifyTitle	Integrate Button - Welcome	
assertTextPresent	Welcome to The IntegrateButton.com. Please log in to access exclusive material for the book.	

如代码清单 6-6 所示，Selenium 利用表格的方式来测试，这是一种非常高效的沟通方式，这个表格的作者可以不是开发人员。您可以看到，这个测试做了一些事情：它验证了网页中的内容，填写了表单，并且验证了数据。

我们都需要理解的一点是，测试是根据它们所需的准备条件（如准备数据库基底数据等）来区分的，这直接关系到执行测试所需的时间。测试分类在 CI 的环境下特别重要——如果在很多情况下构建的执行需要较长时间，就会大大影响您和您的小组对 CI 的感受。

6.5　对开发者测试分类

编写并执行测试显然是件好事，但是我们必须将它作为一种架构的组成部分，需要正确地分类和组织；否则，它们就会变成成功的障碍，而不是钥匙。当项目的代码数量增加时，我们就会面对许许多多的测试——如果您在 CI 系统中随时执行所有写下的测试，完成构建的时间就会变得越来越长。

将开发者测试进行相应的分组（单元测试、组件测试、系统测试和功能测试），可以让您能够先执行较快的测试，然后执行较慢的测试。例如，每次在版本控制库变更时执行系统测试将耗费时间和资源，如果构建中发生问题，相关人员收到通知的时间也比较晚。如果时间延迟得太厉害，开发人员已经转向其他的开发任务，那么持续集成的一个主要好处就没有实现。为什么不在签入时执行单元测试（它们执行较快），然后安排一个时间定期执行组件测试（或在提交构建之后），再安排另一个时间定期执行系统测试呢？定期执行的时间间隔可以随着开发的深入而加大，您可能在项目的初始阶段希望更频繁地执行。

像.NET 平台下的 NUnit 框架和 Java 平台下的 JUnit（新版本）和 TestNG 框架利用了 annotation 的特征，可以很容易地对测试进行分组。在其他的框架中，对测试分组有一些难度。例如，对于较早版本的 JUnit，在框架内部或 Ant 中没有提供一种机制将测试分成三个组。但是，这仍然是可以实现的，可以通过对测试命名的方式，或者更简单，利用合适的目录结构。

开发者测试的一项实践是将单元测试和源代码分别放在两个目录下。例

如，一个项目的目录结构中有一个 src 目录，用于存放源代码；还有一个 test 目录，存放相关的测试。代码清单 6-7 是一个示例项目的根目录。

代码清单 6-7 项目结构示例

```
root
     build.xml
     build.properties
     src/
     test/
```

src 目录下包含了存放源代码的子目录，而 test 目录下进一步划分为更具体的子目录，如 unit、component 和 system。例如，test 目录下的内容如代码清单 6-8 所示。

代码清单 6-8 test 目录的内容

```
test/
     unit/
     component/
     system/
```

清单 6-8 中的 unit、component 和 system 目录保存了各个分类的相关测试。如 system 目录下会有一个目录结构，对应于系统测试的包名（这通常对应于测试包下面相应的类），如代码清单 6-9 所示。

代码清单 6-9 system 目录的结构示例

```
test/
     system/
      test/
       com/
        acme/
         stock/
           LogInTest.java
           AccountTest.java
```

既然测试已经分别放到了不同的目录中，您选择的构建系统也需要更新一下。以 Ant 为例，执行分类测试就变成了利用 Ant 的 JUnit 任务中的 batchtest 元素来定义一些目标，如代码清单 6-10 所示。

代码清单 6-10　JUnit 任务的 batchtest 元素

```
<batchtest todir="${testreportdir}">
   <fileset dir="test/unit">
     <include name="**/*Test.*"/>
        </fileset>
</batchtest>
```

include 元素中所引用的命名模式是一般性的——fileset 的 dir 属性中所指的目录确定了执行哪些测试，在这个例子里就是执行单元测试。

不要忘记您也可以自动执行功能测试，例如那些利用 Selenium 定义的测试；然而，这些测试执行的方式有所不同，测试的工具也不同，可以分离到独立的 Ant 任务中。通过定义分类测试的共同形式，例如通过 annotation 或命名模式，您可以告诉 CI 系统在恰当的时候执行每一类测试，您的构建次数是完全可以管理的。这意味着测试可以定期地执行，而不是当它们需要很长时间执行时就抛弃它们。

6.6　先执行较快的测试

通常，构建的主要时间花在测试上，耗时最长的测试是那些依赖于外部对象的测试，如依赖于数据库、文件系统和 Web 容器等。单元测试要求的准备时间最少（根据定义是不需要准备时间的），系统测试要求的准备时间最多（需要准备好所有的东西）。按类型对测试进行定义和分组（单元测试、组件测试和系统测试），开发团队就能形成一个构建过程，执行不同的测试分类，而不是执行所有测试。单元测试执行的频率最高（每次提交都会执行），组件测试、系统测试和功能测试可以在次级构建中执行，或者定期执行。

单元测试

真正的单元测试应该在少于 1 秒的时间内完成。如果单元测试花了较长的时间，就要仔细检查一下——要么它失败了，要么它实际上是一个组件级的测试，而不是单元测试。XP 的咒语“测试一点、编码一点、测试一点……”就是建立在快速测试的基础上的。如果单元测试花的时间足以让开发者抽出空来

关注其他事情，那它就太长了。它开始成为负担，很快就会成为大家避免去做的事情，而不是值得依靠的事。

在 CI 的环境中，构建在有人向版本控制库提交变更时就会执行。因此，单元测试应该在每次有人签入代码时执行（这称为提交构建）。配置需要一些代价，但执行这些测试的资源代价可以忽略不计。

组件测试

组件测试通常有一些依赖关系，执行的时间比较长。因此，它们应该作为次级构建的一部分执行，或者定期执行。不论是哪种方式，它们都应该在您向版本控制库提交代码之前执行（在您的私有构建中）。我们在第 4 章中曾讨论到，组件测试可以作为次级构建的一部分执行。作为更“重量级”的集成构建，次级构建可以跟在提交构建之后执行。组件测试有一些具体的代价：依赖关系必须安装配置好。这些测试本身可能只要花数秒的时间，但是集中在一起，总的时间就比较长了。某些项目中的一些轻量级组件测试可以在每次提交构建时执行。

例如，代码清单 6-11 中所示的组件测试平均的执行时间为 4 秒。

代码清单 6-11 组件测试示例

```
using System;
using System.Collections;
using NUnit.Framework;
using NHibernate.Cfg;
using NDbUnit.Core.OleDb;
using NDbUnit.Core;

namespace NHibernate.words
{
   [TestFixture]
   public class WordTest
   {
      private const string CONN = @"Provider=SQLOLEDB..";
      private const string SCHEMA = @"Dataset2.xsd";
      private const string XML = @"XMLFile2.xml";
```

```
        private OleDbUnitTest fixture;
        private ISessionFactory sessFact;

        [SetUp]
        public void SetUp()
        {
           this.fixture = new OleDbUnitTest(CONN);
           this.fixture.ReadXmlSchema(SCHEMA);
           this.fixture.ReadXml(XML);
           this.sessFact =
               new Configuration().Configure().BuildSessionFactory();
        }

        [Test]
        public void verifyFinder()
        {
          this.fixture.PerformDbOperation(DbOperationFlag. CleanInsert);
          ISession session = this.sessFact.OpenSession();

          IQuery qry = session.GetNamedQuery("word.finder. bySpelling");
          qry.SetAnsiString("spelling", "pugnacious");
          IList list = qry.List();
          Assert.AreEqual(((Word)(list[0])).PartOfSpeech, "adj");
          session.Close();
        }
    }
}
```

这个测试做了一些事情，导致总体测试时间增加，而且它配置起来也更复杂。首先，测试利用NDbUnit[3]为数据库填充了基底数据。NDbUnit是一个数据库填充的框架。在这个例子中，NDbUnit插入了从XML文件XMLFile2.xml中读出的数据，这也意味着需要进行XML解析。这个测试用例接下来配置了NHibernate，然后执行了一个测试，从数据库中取到了一个单词。

对这个测试为什么要花 4 秒钟执行感到奇怪吗？在这个类中增加测试用例不会增加多少执行时间。但是如果将这个测试执行 10 多次，总的时间就要接

[3] NDbUnit 是针对.NET 平台的一个开放源代码项目，可以从 www.ndbunit.org/获得。

近 1 分钟了。

系统测试

系统测试和功能测试需要完全安装系统，执行的时间最长。另外，配置一个全功能的系统是相当复杂的，有时候这一点使得这些测试不能够完全自动化。在每次提交构建时执行系统测试可能会成为灾难，但有时候这些类型的测试是在次级构建或定期构建中执行的。或者，每晚（下班时间）执行这些测试也是很好的选择。

下次您在构建中加入测试用例时，请考虑执行所有的测试从长期来看意味着什么，然后开始优化构建，对测试进行分类，以便能够分阶段执行这些测试。

6.7 为缺陷编写测试

开发者测试和 CI 可以减少软件缺陷发生的频率，但说实话，缺陷还会产生。但这并没有什么问题——错误会发生，错误会被修复，在理想情况下，我们可以从错误中学到东西。但是同样的错误犯两次就是不可原谅的。

有些人用术语“缺陷驱动的开发”来指为缺陷编写测试，但是这个术语听起总是比较消极。缺陷不会驱动开发——防止这些讨厌的偏差才能驱动开发！如果要说它们之间的关系，那么应该说缺陷阻碍了开发——定位缺陷并防止它们再度出现，这才能保持项目继续向前进。这里有一个预防策略，保证缺陷一旦被发现，就再也不会再次出现。

当缺陷被发现时，找出并隔离有问题的代码。如果项目有相当数量的测试用例，可以保持项目健康，也许可以假设缺陷存在于没有测试的代码之中（可能是未考虑的路径）——最有可能是在组件与组件的交互中。例如，代码清单 6-12 展示了一个 Hibernate DAO 对象的 find 方法，它试图从数据库中取出一个单词。

代码清单 6-12　有一个缺陷的 DAO

```
public IWord findWord(String word) throws FindException{
  Session sess = null;
  try{
    sess = WordDAOImpl.sessFactory.getHibernateSession();

    final Query qry = sess.getNamedQuery("word.finder. bySpelling");
    qry.setString("spelling", word);

    final List lst = qry.list();
    final IWord wrd = (IWord)lst.get(0);
    sess.close();
    return wrd;
  }catch(Throwable thr){
    try{sess.close();}catch(Exception e){}
    throw new FindException("Exception while finding word: "
        + word + " "+ thr.getMessage(), thr);
  }
}
```

这个类在一系列利用 DbUnit 的组件级测试中进行了相当程度的测试。这些测试验证了基本的 CRUD（创建、读取、更新和删除）操作。例如，代码清单 6-13 展示了 find 方法的一个测试。

代码清单 6-13　好心情测试用例示例

```
public void testFindVerifyDefinition() throws Exception{
  final WordDAOImpl dao = new WordDAOImpl();
  final IWord wrd = dao.findWord("pugnacious");

  for(Iterator iter = wrd.getDefinitions().iterator();
      iter.hasNext();){
      IDefinition def = (IDefinition)iter.next();
      TestCase.assertEquals(
        "def is Combative in nature; belligerent.",
        "Combative in nature; belligerent.",
        def.getDefinition());
  }
}
```

在更大应用（在这个例子中是一个字典）的功能测试中发现，如果用户试图查找一个不在字典中的单词时，应用程序会报异常并显示调用栈信息，这让

用户相当困惑。在一段时间的仔细检查之后，某人发现如果 HibernateAPI 没有返回任何单词，WordDAOImpl 的 findWord 方法就会抛出一个未预料到的 IndexOutOfBoundsException（这被 FindException 掩盖了）。

这种偏离预期的行为没有考虑到！发现了一个缺陷！但我们没有什么损失。要记住，我们原谅这个错误的产生，但只原谅一次。我们有机会来修复这个可恶的小故障，但是如果它再次出现，我们就应该重新考虑我们的做法。

第一步是编写一个测试用例来暴露这个缺陷，这让您能够重拾信心。慢慢地读这一句话。您的第一反应可能是修复这段有问题的代码，然后转向别的工作，去做更开心的事情（快乐时光！）。然而，如果您这样做的话，就失去了防止这个缺陷再次出现的机会。开始要先写一个测试用例，触发缺陷报告中描述的同样的行为。在这个例子中，我们需要让代码抛出 IndexOutOfBounds Exception，如代码清单 6-14 所示。要记住，我们打算写一个能通过的测试，而不是失败的测试。

代码清单 6-14 验证缺陷的测试用例

```
public void testFindInvalidWord() throws Exception{
  final WordDAOImpl dao = new WordDAOImpl();
  try{
     final IWord wrd = dao.findWord("fetit");
     TestCase.fail("This should throw an exception");
  }catch(FindException ex){
     Throwable thr = ex.getOriginalException();
     TestCase.assertTrue("Should be instance of " +
       IndexOutOfBoundsException",
        ex.getOriginalException() instanceof
           IndexOutOfBoundsException);
  }
}
```

如果您执行这个测试，它会通过。因此，您已证实确实存在一个缺陷。接下来您可以修复它。

顺便说一句，这种方法与流行的“缺陷驱动开发”有一些区别。后一种方法建议先编写一个会失败的测试用例，然后不断执行这个测试（在修复缺陷的

过程中），直到测试不再失败为止。例如，代码清单 6-15 中的代码就是一个缺陷驱动的测试用例。

代码清单 6-15　测试驱动风格的测试用例示例

```
public void testFindInvalidWordException() {
  final WordDAOImpl dao = new WordDAOImpl();
  try{
     final IWord wrd = dao.findWord("fetit");
  }catch (FindException e){
     TestCase.fail("Didn't find word fetit");
  }
}
```

当然，这个测试用例在第一次执行时会失败（假定缺陷仍然存在）。这种方法确实能工作，但是还有改进的机会。编写一个有意让它在第一次执行时失败的测试用例有下面一些问题。

- 在这种情况下，很难正确地使用一个断言编写将会失败的测试。
  由于这一原因，断言可能不会加入，即使是在测试用例不再失败时，也不会加入断言。这意味着测试用例不一定是通过了，它只是没有失败而已。

- 在这个过程中的某个时刻，您需要知道修复将怎样影响代码的行为，所以在尝试编写会失败的测试时，您需要猜测修复可能是怎样的。
  在代码清单 6-15 中，我们的假定是修复将使代码不再抛出异常。但这只是问题的一部分。

- 当对被测试代码进行修复之后，失败的测试能工作了，但是它并没有真正验证行为的变更。

此时，因为测试用例通过了，大多数人都不会再回过头去更新测试用例。在我们的方法中，为了修复缺陷，我们基本上需要破坏测试，这与缺陷驱动的开发在做法上是不同的。

仔细检查代码之后表明，在取出第一个元素之前，我们需要检查列表是否为空。这时我们遇到了一个设计选择——代码应该返回 null，或是返回一个空

的 Word，还是抛出一个异常呢？我们决定，如果不能通过 Hibernate 从数据库中取出参数中的值，就返回 null（参见代码清单 6-16）。

代码清单 6-16 修复了缺陷的更新代码

```
public IWord findWord(String word) throws FindException{
  Session sess = null;
  try{
     sess = WordDAOImpl.sessFactory.getHibernateSession();

     final Query qry = sess.getNamedQuery("word.finder. bySpelling");
     qry.setString("spelling", word);

     final List lst = qry.list();
     IWord wrd = null;
     if(lst.size() > 0){
       wrd = (IWord)lst.get(0);
     }
     sess.close();
     return wrd;
  }catch(Throwable thr){
     try{sess.close();}catch(Exception e){}
     throw new FindException("Exception while finding word: "
        + word + " "+ thr.getMessage(), thr);
  }
}
```

当被测试的代码确信已被修复时，再次执行测试，这次它会失败。接下来的决定就是这种方法与其他方法之间的差异——在修复我们的测试用例时，我们会断言新的行为。缺陷驱动的例子现在也能工作，我们很可能就让它保持原样了。但是这个测试用例现在已经不再能提供什么价值了。我们需要断言当无效的单词传递给 findWord 方法时，会返回 null。我们也需要断言不会再抛出异常了。更新后的测试用例如代码清单 6-17 所示。

代码清单 6-17 更新测试用例验证修复

```
public void testFindInvalidWord() throws Exception{
  final WordDAOImpl dao = new WordDAOImpl();
  try{
```

```
        final IWord wrd = dao.findWord("fetit");
        TestCase.assertNull("Should have received back a null object", wrd);
      }catch(FindException ex){
        TestCase.fail("This should not throw an exception");
      }
    }
```

现在我们完成了工作，我们一共做了两件事情。首先，缺陷被更正了。恭喜！其次，有了一个回归测试，真正地断言了修复后的正确行为。

我们应该采用哪种方法：缺陷驱动开发还是我们所谓的“持续预防开发”？这两种方法都让您能够：

- 修复缺陷。
- 防止缺陷再次发生。

但是，持续预防开发会促使您完成第三步，即断言缺陷修复后的新行为。

6.8　让组件测试可重复

许多 Web 应用程序需要访问数据库。但是数据库对于测试来说是相当沉重的依赖关系，所以您有两种选择：要么尽量地进行模拟，在尽可能长的时间内完全避免使用数据库，要么使用数据库并承受其开销。第二种选择带来了一系列的新问题——您在测试过程中如何控制数据库？或者更好一点，您如何让这些测试可重复？

到目前为止，实现这种测试最容易的方法是使用某种 xDbUnit 这样的数据库填充框架（如针对.NET 的 NDbUnit、针对 Java 的 DbUnit、针对 Python 的 PDbSeed）。这些框架将数据库的数据集抽象到 XML 文件中，然后为开发者提供细粒度的控制，即在测试过程中如何将这些数据填充到数据库中。例如，代码清单 6-18 中的片断取自于一个 DbUnit 的 XML 数据文件。

代码清单 6-18　DbUnit 数据文件示例

```
<word WORD_ID="1" SPELLING="pugnacious" PART_OF_SPEECH="Adjective"/>
<definition DEFINITION_ID="10"
```

```
  DEFINITION="Combative in nature; belligerent."
  WORD_ID="1"
  EXAMPLE_SENTENCE="The pugnacious youth had no friends left to pick on."/>
<synonym SYNONYM_ID="20" WORD_ID="1" SPELLING="belligerent"/>
<synonym SYNONYM_ID="21" WORD_ID="1" SPELLING="aggressive"/>
```

通过 DbUnit 的 DataBaseTestCase，XML 中的数据可以实现插入、更新和删除等操作。通过实现抽象的 getConnection 方法，配置具体的数据库，XML 文件则通过 getDataSet 方法来定位（参见代码清单 6-19）。

代码清单 6-19 数据库测试用例示例

```
public class DefaultWordDAOImplTest extends DatabaseTestCase
{
    protected IDataSet getDataSet() throws Exception {
        return new FlatXmlDataSet(
          new File("test/conf/words-seed.xml"));
    }

    protected IDatabaseConnection getConnection() throws Exception {
        final Class driverClass =
             Class.forName("org.gjt.mm.mysql.Driver");
        final Connection jdbcConnection =
        DriverManager.getConnection(
           "jdbc:mysql://localhost/words",
           "words", "words");
         return new DatabaseConnection(jdbcConnection);
    }

    public void testFindVerifyDefinition() throws Exception{
       final WordDAOImpl dao = new WordDAOImpl();
       final IWord wrd = dao.findWord("pugnacious");

       for(Iterator iter =
          wrd.getDefinitions().iterator(); iter.hasNext();){
         IDefinition def = (IDefinition)iter.next();
         assertEquals("Combative in nature; belligerent.",
             "Combative in nature; belligerent.",
             def.getDefinition());
       }
```

```
    }

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

但是请注意，这个类假定数据库和执行测试的计算机是同一台机器。这在开发者的工作站上可能是一个安全的假定，但这个配置在 CI 的环境下显然会带来问题。

一种解决方案是分离出硬编码的连接字符串，将它们放入属性文件。但是还有一种更有效率的机制。如果使用 DbUnit 来填充数据库，您可以推断应用程序本身也使用了数据库。如果是这种情况，那么通常的做法是避免在代码中硬编码数据库连接的信息。既然如此，为什么不对 DbUnit 进行配置，让它读取被测试的应用程序所读取的文件呢？

例如，在 Hibernate 应用程序中，数据库连接信息通常是在 hibernate.cfg.xml 文件中定义的。您可以很容易地编写一个工具类，解析该文件并取得正确的连接信息。更好的方法是您可以依靠 Hibernate 来提供想要的信息，如代码清单 6-20 所示。

代码清单 6-20　Hibernate 配置工具类

```
public class DBUnitHibernateConfigurator {
 static Configuration configuration = null;

 private DBUnitHibernateConfigurator() {
  super();
 }

 private static Configuration getConfiguration()
     throws HibernateException {
  if (configuration == null) {
   configuration = new Configuration().configure();
  }
  return configuration;
}

public static IDataSet getDataSet(final String fileName)
    throws ResourceNotFoundException,
     DBUnitHibernateConfigurationException {
```

```
 try{
  return DBUnitConfigurator.getDataSet(fileName);
 }catch(DBUnitConfigurationException e2){
   throw new DBUnitHibernateConfigurationException(
     "DBUnitConfigurationException in getDataSet", e2);
 }
}

 private static String getProperty(final String name)
    throws HibernateException {
  return getConfiguration().getProperty(name);
 }

public static Properties getHibernateProperties()
    throws ResourceNotFoundException,
      DBUnitHibernateConfigurationException{
 try{
  final Properties hProp = new Properties();
  hProp.put("hibernate.connection.driver_class",
    DBUnitHibernateConfigurator.getProperty(
      "hibernate.connection.driver_class"));
  hProp.put("hibernate.connection.url",
    DBUnitHibernateConfigurator.getProperty(
      "hibernate.connection.url"));
  hProp.put("hibernate.connection.username",
    DBUnitHibernateConfigurator.getProperty(
      "hibernate.connection.username"));
  hProp.put("hibernate.connection.password",
    DBUnitHibernateConfigurator.getProperty(
      "hibernate.connection.password"));
  return hProp;
 }catch(HibernateException e){
   throw new DBUnitHibernateConfigurationException(
     "HibernateException in getHibernatePropertiesFile", e);
 }
}

 public static IDatabaseConnection getDBUnitConnection()
    throws DBUnitHibernateConfigurationException{
  try{
   final Properties props =
```

```
      DBUnitHibernateConfigurator.getHibernateProperties();
     return DBUnitConfigurator.getDBUnitConnection(props);
    }catch(DBUnitConfigurationException e1){
     throw new DBUnitHibernateConfigurationException(
      "DBUnitConfigurationException in getDBUnitConnection", e1);
    }catch (ResourceNotFoundException e2) {
     throw new DBUnitHibernateConfigurationException(
       "ResourceNotFoundException in getDBUnitConnection", e2);
    }
   }
  }
```

请注意代码清单 6-20 中的类是如何将 Hibernate 的连接信息放到一个 Properties 对象中去的，这个对象随即在 DbUnitConfigurator 类中被转换到 DbUnit 的 IDatabaseConnection 类型。然后通过 getDbUnitConnection 方法返回 DbUnit 连接类型。DbUnit 的 IDataSet 类型代表那些包含所有数据的 XML 文件，它通过 getDataSet 方法返回。这个方法让开发者不必提供文件的路径——这在不同的环境中是一个特别有用的技巧。

在代码清单 6-21 中，可以创建一个定制的抽象测试用例类，它要求实现者为特定的测试用例填充期望的数据集信息。

代码清单 6-21　方便的测试用例

```
public abstract class DefaultDBUnitHibernateTestCase extends
DatabaseTestCase {
 public DefaultDBUnitHibernateTestCase(String name) {
  super(name);
 }

  protected void setUp() throws Exception {
   super.setUp();
   DefaultHibernateSessionFactory.
      closeSessionAndEvictCache();
   DefaultHibernateSessionFactory.
      getInstance().getHibernateSession();
  }

  protected void tearDown() throws Exception {
   DefaultHibernateSessionFactory.
      closeSessionAndEvictCache();
```

```
    super.tearDown();
  }

  protected IDatabaseConnection getConnection() throws Exception {
    return DBUnitHibernateConfigurator.
       getDBUnitConnection();
  }

  protected IDataSet getDataSet() throws Exception {
    final String fileName = this.getDBUnitDataSetFileForSetUp();
    DatabaseTestCase.assertNotNull("data set file was null", fileName);
    return DBUnitHibernateConfigurator.getDataSet(fileName);
  }

  protected abstract String getDBUnitDataSetFileForSetUp();
}
```

代码清单 6-22 展示了一个实现了 DefaultDbUnitHibernateTestCase 的测试用例的例子。

代码清单 6-22 实际工作中的新测试用例

```
public class WordDAOImplTest extends DefaultDBUnitHibernate-

TestCase {
 public void testUpdateWordSpelling() throws Exception{
  WordDAOImpl dao = new WordDAOImpl();
  IWord wrd = dao.findWord("pugnacious");

  wrd.setSpelling("pugnacious-ness");
  dao.updateWord(wrd);

  IWord wrd2 = dao.findWord("pugnacious-ness");
  assertEquals("should be id of 1", 1, wrd2.getId());
 }

 public void testFindVerifyDefinitionsSize() throws Exception{
  WordDAOImpl dao = new WordDAOImpl();
  IWord wrd = dao.findWord("pugnacious");

  Set defs = wrd.getDefinitions();
  assertEquals("size should be one", 1, defs.size());
```

```
 }

 protected String getDBUnitDataSetFileForSetUp() {
  return "words-seed.xml";
 }

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

DbUnit 提供了一组 API（前面曾介绍），可以通过组合有效地应用，这也为强大的组合框架提供了大量的机会。利用这种灵活性，在不同的层次上测试各种架构就变得相当容易。例如，为 Struts 应用程序编写开发者测试可能是一项有挑战性的工作。常见的做法是利用 HttpUnit 这样的框架来模拟 HTTP 请求，但这是一项枯燥的工作，而且没有精确地针对 Struts 架构中大量使用的 Action 类和请求映射配置文件。

StrutsTestCase 项目是针对这个问题而创建的。利用这个框架，您可以方便地隔离测试 Struts 的 Action 类。但是，这个项目要求开发者扩展一个基类，这个基类模拟了一个 servlet 容器。如果一个 Struts 应用程序需要使用数据库，您可能就难办了。

通过 DbUnit 的 API，可以创建一个组合框架，同时利用 DbUnit 的填充数据功能和 StrutsTestCase 项目的 servlet 容器模拟功能（参见代码清单 6-23）。

代码清单 6-23　配置 Struts 和 Hibernate 测试用例

```
public abstract class DefaultDBUnitMockStrutsTestCase
   extends MockStrutsTestCase {

 public DefaultDBUnitMockStrutsTestCase(String testName) {
  super(testName);
 }

 public void setUp() throws Exception {
  super.setUp();
  this.executeOperation(this.getSetUpOperation());
 }
```

```
 public void tearDown() throws Exception{
  super.tearDown();
  this.executeOperation(this.getTearDownOperation());
 }

 private void executeOperation(DatabaseOperation operation)
    throws Exception{
  if (operation != DatabaseOperation.NONE){
   final IDatabaseConnection connection =
     this.getConnection();
  try{
   operation.execute(connection, this.getDataSet());
  }finally{
   closeConnection(connection);
  }
 }
}

protected void closeConnection(IDatabaseConnection connection)
   throws Exception{
  connection.close();
 }

protected abstract Properties getConnectionProperties();

protected abstract String getDBUnitDataSetFileForSetUp();

protected IDatabaseConnection getConnection() throws Exception {
  final Properties dbPrps = this.getConnectionProperties();
  DatabaseTestCase.
    assertNotNull("database properties were null", dbPrps);
  return DBUnitConfigurator.getDBUnitConnection(dbPrps);
 }

protected DatabaseOperation getSetUpOperation() throws Exception {
  return DatabaseOperation.CLEAN_INSERT;
 }

protected DatabaseOperation getTearDownOperation() throws Exception {
  return DatabaseOperation.NONE;
```

```
  }

protected IDataSet getDataSet() throws Exception {
  final String fileName = this.getDBUnitDataSetFileForSetUp();
  DatabaseTestCase.assertNotNull("data set file was null", fileName);
  return DBUnitConfigurator.getDataSet(fileName);
 }
}
```

同样，您又会遇到选择：使用硬编码的连接信息还是复用已有的文件？利用 Hibernate 来测试一个 Struts 应用程序？没问题——只要组合新的 DefaultDBUnitMockStrutsTestCase 和已有的读取 Hibernate 文件的工具就行了。

例如，代码清单 6-24 是一个扩展了 DefaultMerlinMockStrutsTestCase 的类，它结合了 DefaultDBUnitMockStrutsTestCase 的 DbUnit 功能和前面代码清单 6-20 中定义的 Hibernate 文件读取的功能。

现在您得到了一个漂亮的测试用例，任何人都很难抱怨说他们无法以一种可重复的方式来测试这个应用程序。

代码清单 6-24　实际工作中的组合框架

```
public class ProjectListActionTest
  extends DefaultMerlinMockStrutsTestCase {

 public void testProjectListAction() throws Exception{
  this.setRequestPathInfo("/viewProjects");
  this.actionPerform();
  this.verifyForward("success");

  IProject[] projects = (IProject[])this.getRequest().
     getAttribute("projects");
  assertNotNull("object was null", projects);
 }

public ProjectListActionTest(String name) {
  super(name);
 }

 protected String getDBUnitDataSetFileForSetUp() {
  return "dbunit-project-seed.xml";
 }
```

```
}
```

6.9 将测试用例限制为一个断言

由于推动开发的日程表很紧，也由于即将到来的快乐时光，人们趋向于将所有东西都塞到一个测试用例中去。这种随便的做法导致在一个测试用例中有大量的 assert 语句。例如，代码清单 6-25 中的代码试图在一个测试用例中验证 HierarchyBuilder 的 buildHierarchy 方法，以及 Hirarchy 对象的行为。

代码清单 6-25 有太多断言的测试用例

```
public void testBuildHierarchy() throws Exception{
    Hierarchy hier = HierarchyBuilder.buildHierarchy(
      "test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be 2", 2,
        hier.getHierarchyClassNames().length);
    assertEquals("should be junit.framework.TestCase",
        "junit.framework.TestCase",
        hier.getHierarchyClassNames()[0]);
    assertEquals("should be junit.framework.Assert",
        "junit.framework.Assert",
        hier.getHierarchyClassNames()[1]);
    }
```

请注意，代码清单 6-25 中有 3 个断言方法调用。这是一个合法的 JUnit 测试用例，没有什么会阻止一个测试用例中出现多个断言。但是这样做的问题在于，JUnit 是按照“快速失效”的原则设计的。如果第一个断言失败，整个测试用例就在失败处中止了。这意味着下面的两个断言在测试执行时没有执行到。

当您完成代码修复并再次执行该测试时，第二个断言可能失败，这又导致了整个修复/重新执行测试用例的循环。如果第二次执行时，第三个断言失败，同样要重复这个过程。注意到这里低效率的地方了吗？

更有效率的做法是在每个测试用例中限制使用一个断言。这样，不需要将刚才描述的三步过程重复许多次，一次执行就能报告所有的失败，不需要额外的干预。例如，代码清单 6-25 中的代码可以重构为 3 个独立的测试用例（参见代码清单 6-26）。

代码清单 6-26　测试用例重构

```
public final void testBuildHierarchyStrSize() throws Exception{
    Hierarchy hier = HierarchyBuilder.buildHierarchy(
     "test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
      assertEquals("should be 2", 2,
         hier.getHierarchyClassNames().length);
}

public final void testBuildHierarchyStrNameAgain() throws Exception{
    Hierarchy hier = HierarchyBuilder.buildHierarchy(
      "test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be junit.framework.TestCase",
       "junit.framework.TestCase",
       hier.getHierarchyClassNames()[0]);
}

public final void testBuildHierarchyStrName() throws Exception{
    Hierarchy hier = HierarchyBuilder.buildHierarchy(
      "test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be junit.framework.Assert",
       "junit.framework.Assert",
       hier.getHierarchyClassNames()[1]);
}
```

利用 3 个独立的测试用例，在第一次执行测试时，会报告 3 个失败。这样，您就只需要一次修复/执行循环。这也是为什么我们在本章开始时介绍分离的目录结构的原因。测试用例的数目和您的代码保持相同的增长速度，所以您肯定是在前进！

6.10　本章小结

您希望您的软件有多可靠？源代码的可靠性取决于测试的覆盖率，测试的价值取决于它们执行的频率。通过将测试分为 4 个自动执行的分组，即单元测试、组件测试、系统测试和功能测试，CI 系统可以按一种有效的方式进行配置，执行这些测试。单元测试可以在签入时执行，组件测试、系统测试和功能测试可以定期执行，例如作为次级构建的一部分。

表 6-1 总结了本章中介绍的一些实践。

表 6-1 本章中讨论过的 CI 实践

实　践	描　述
自动化单元测试	让您的单元测试自动化，最好是利用 NUnit 或 JUnit 这样的单元测试框架。这些单元测试应该没有外部依赖关系，不会依赖于文件系统或数据库
自动化组件测试	利用 JUnit、NUnit 这样的单元测试框架，让您的组件测试自动化，如果用到数据库，就使用 DbUnit 或 NDbUnit。这些测试涉及更多对象，通常比单元测试需要花更长的时间来执行
自动化系统测试	系统测试比组件测试执行的时间更长，通常涉及多个组件

续表

实　践	描　述
自动化功能测试	利用 Selenium（针对 Web 应用程序）和 Abbot（针对 GUI 应用程序）这样的工具，可以让功能测试自动化。功能测试从用户的角度执行操作，通常是自动化测试套件中执行时间最长的
将开发者测试分组	通过将测试分为一些组，您可以按不同的时间间隔来执行那些较快（如单元测试）和较慢的测试（如组件测试）
先执行较快的测试	先执行单元测试，再执行组件测试、系统测试和功能测试。您可以通过测试分组来做到这一点
为缺陷编写测试	根据新的缺陷编写测试，增加代码覆盖率，确保这个缺陷不会再次出现
让组件测试可重复	利用数据库测试框架确保数据处于“已知状态”，这有助于使组件测试可重复
将测试用例限制为一个断言	将测试用例限制为一个断言，您可以花较少的时间来追踪测试失败的原因

6.11 问题

请利用下面的问题来评估您的 CI 环境中的测试过程，以及它们能为您带来什么：

- 您是否对自动化的测试进行分组，例如分成单元测试、组件测试、系统测试和功能测试？

- 您是否对 CI 系统进行配置，在不同阶段的构建中执行不同的测试分类？
- 您是否为每个缺陷编写自动化的单元测试？
- 每个测试用例中包含多少断言？您是否将测试用例限制为包含一个断言？
- 这些测试用例是否自动化？您的项目是否将自动化的开发者测试提交到版本控制库中？