HZ BOOKS
华章科技
PEARSON

Amazon五星级畅销书，作者权威，在全球iOS/Mac开发者社区享有盛誉

完美地展现了测试驱动开发方法与iOS开发的结合，能使iOS开发者在产品需求、软件设计、测试有效性与开发效率之间达到很好的平衡

华章程序员书库

Test-Driven iOS Development

测试驱动的iOS开发

（美） Graham Lee 著
爱飞翔 译

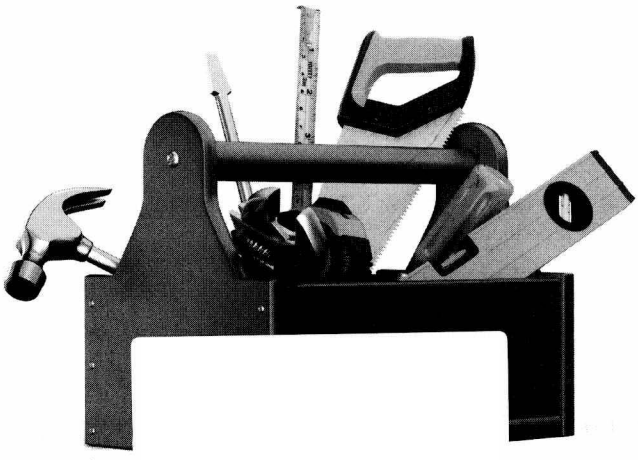
机械工业出版社
China Machine Press

华章程序员书库

Test-Driven iOS Development

测试驱动的iOS开发

（美） Graham Lee 著
爱飞翔 译



机械工业出版社
China Machine Press

本书是目前仅有的一本讲解如何将测试驱动的开发方法应用到 iOS 开发中的权威教程，从测试工具、测试驱动开发方法和技巧、基于测试驱动开发方法的软件设计等多角度完美地展现了测试驱动开发方法与 iOS 开发的结合，不仅能使 iOS 开发者迅速掌握测试驱动的开发方法，还能使他们在产品需求、软件设计、测试有效性与开发效率之间达成达到很好的平衡。

全书一共 13 章：第 1 章简单介绍了通用的软件测试知识，以及软件测试的目标；第 2 章介绍如何利用测试驱动开发与单元测试来达到这个目标；第 3 章将如何进行单元测试的设计与编写；第 4 章将深入学习如何使用 Apple 的开发工具中绑定的 OCUnit 测试框架；第 5 章揭示在 iOS 应用程序的开发过程中如何从最初的需求规格书演进到最终产品；第 6 章阐述如何实现从需求描述中提取出来的数据模型；第 7 章将实现应用程序的业务逻辑；第 8 章以测试用例为指导，为 BrowseOverflow 应用程序设计并实现网络通信功能；第 9 章讲述如何编写视图控制器的代码，让 BrowseOverflow 程序将这些信息展示给用户；第 10 章将以上述类整合起来，实现一个功能完备且能够正常运行的应用程序；第 11 章讲解一些用于移除类之间的依赖性、代码运行环境依赖性以及线程依赖性的设计范式；第 12 章回答了在什么情况下适合采用 TDD 开发方式；第 13 章展望了业界一些更为新颖的功能、一些对已有技术的扩展以及一些有用的开发工具。

本书适合从事 iOS 测试驱动的软件开发人员和程序员阅读。

本书版权登记号：图字：01-2012-4658

图书在版编目（CIP）数据

测试驱动的 iOS 开发 /（美）李（Lee, G.）著；爱飞翔译 . —北京：机械工业出版社，2012.10
书名原文：Test-Driven iOS Development

ISBN 978-7-111-39919-3

I. 测…　II. ①李…　②爱…　III. 移动电话机－应用程序－程序设计　IV. TN929.53

中国版本图书馆 CIP 数据核字（2012）第 232380 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）
责任编辑：谢晓芳
北京市荣盛彩色印刷有限公司印刷
2012 年 10 月第 1 版第 1 次印刷
186mm×240mm • 13.5 印张
标准书号：ISBN 978-7-111-39919-3
定价：49.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

译 者 序

测试驱动开发是近年来受关注程度较高的一种开发过程，它与“敏捷软件开发”与“极限编程”等软件设计方法学都有紧密关联。如何才能保证软件产品的代码质量，这是富有责任心的程序员一直都在探索的问题。**测试驱动开发的妙处即在于，它以需求为引领，通过测试的形式，来指导开发者进行软件的设计与架构，并编写出最为精炼的代码，使得测试用例运行通过。经过适当的重构之后，测试用例与产品代码均可达到较为健康的状态。**

有了这样的测试套件作为代码质量与代码安全的保证，开发者即可在面对各式各样的需求冲击时，更为稳健地添加产品功能；同时，新功能的实现又可受益于既有的架构与设计，因为由测试驱动开发方式所催生出来的架构，既富有扩展性，又显得很简洁。译者在4年有余的工作经历中，也逐步感觉到：在项目中引入测试驱动开发方式，可以使软件的设计与功能的实现之间，形成良性循环，这二者都在测试驱动的引领与既有测试的保护下，可以高效地应对持续变化的需求。

与20世纪90年代末产生的测试驱动开发技术相比，iOS平台的应用程序开发则是一个相对较新的领域。自2007年诞生以来，iOS平台迅速成为业界的热门话题。Apple系列的产品一贯以其精美的操作界面与良好的用户体验著称，截至2012年6月，App Store已经有超过65万个iOS应用程序了，其累计下载量突破300亿次。

与Android、Windows Phone等移动平台相比，目前的iOS平台，从产品质量和营收情况看，都显得更具吸引力，因此有一大批公司、团队与个人开发者都涌入了这个领域。与传统软件行业相比，移动应用项目在开发资金、开发人员、开发周期方面都更为紧缩。在新产品的数量与既有产品的更新频度都急速增加的情况下，软件项目的可维护性与可扩展性却因为过分追求“短周期产品开发”而大幅度下降。

如果这些移动软件项目的代码质量得到了提高，那么它就会有效地降低项目维护所需的成本，而且已有的测试用例与数次重构之后所产生的高内聚、低耦合的模块化代码，又大大减少了后续版本研发新功能时所需的开发、调试时间。高质量的代码还可以封装起来，为新的软件项目所复用，从而降低新项目的研发周期与人力资源消耗。

近些年市面上讲述测试驱动开发的专著很多，从Kent Beck的经典入门读物《测试驱动开发》到Steve Freeman与Nat Pryce所写的那本强调使用TDD来培养面向对象思维的《测试驱

动的面向对象软件开发》，各种关注点不同的经典教材都使大家受益匪浅。然而，我们却很少能够见到像本书这样的优秀教程，本书将 TDD 开发方式全面贯穿于移动开发平台中。

读者通过研习本书，既可以掌握提升项目素质的先进 TDD 开发方式，又能够迅速制作出符合市场需求的高质量软件产品。无论你对测试驱动开发与 iOS 应用程序开发这两个领域的掌握程度如何，阅读完本书之后，都会获得开发技能与思维水平的双重提高。**在初识书名时，译者就被这两种流行技术的相互结合所吸引，在翻译完全书之后，更是觉得它在产品需求、软件设计、测试有效性与开发效率之间达成了相当完美的平衡，是一本独具视角的佳作。**

在翻译过程中，对于众多的术语，译者都尽可能地查找相关资料，给予适当的注释。如果某个专有名词存在多种译法，且尚未有普遍认可的中文译法，则会将英文单词与各个译法都列出，供大家参考。本书主要由爱飞翔翻译，王鹏、舒亚林及张军也参与了部分翻译工作。感谢家人和朋友对我翻译工作的支持与鼓励。由于译者水平有限，错误与疏漏在所难免，恳请广大读者批评指正。

译 者

前　言

我向其他开发者讲授测试驱动开发很大程度上是出于偶然。本来安排我在会议上关于另一个话题发表演讲，而我的一个朋友则讲述 TDD（Test Driven Development，测试驱动开发）。不过朋友的妻子选择那周末生双胞胎（我认为是这样的，我可不是这方面的专家），所以 Chunk，也就是委托我写这本书的人，问我能不能也关于 TDD 发表演讲。由此机缘，最终使我开始了为期一年的写书过程。

通常来说，真实状况并不像我们彼此之间讲述的那样美好。实际上，我初次接触单元测试是在很多年前了。在成为专业的软件开发者之前，我曾在一家制作基于 GNUstep（由自由软件基金会所制作，可运行于 Linux 与其他平台的 Cocoa 库）产品的公司里当测试人员。据我当时所知，单元测试就是确保软件产品的每个小部件都能正常运行的一种开发方法。当这些小部件合并为大的组件时，按理说它也能正常运行才是。

这种理解一直持续到了我从事第一份编程工作时，那个工作是负责某个跨平台安全软件的 Mac 系统版本开发。（这里我又一次省略了一件事——几年前，我曾做过一个为期 6 周的 LISP 外包程序。每个人都曾做过一些并不为之骄傲的事情。）在做这份工作时，我参加了一个 TDD 培训课程，这项课程是由经常活跃于面向对象编程讨论会上的 Kevlin Henney 所开设的。他除了写过很多文章之外，还曾编辑过一本书，名叫《每个程序员都应该知道的 97 件事》[⊖]。就是通过学习这个课程，我才最终意识到测试驱动开发的主旨是使得开发者对其编写的代码更有信心。随着学习的深入，我还领悟到测试驱动开发可以使开发者在修改代码时心里更加踏实。当对 TDD 有足够的了解，并能够从实践的教训中总结经验时，我终于将 TDD 当成了日常开发的一部分，并弄懂了其中哪些方法适合我，哪些不适合。几年之后，我能够接受 Chuck 的邀请，来做一场关于 TDD 的演讲了。

真心希望这本书能够让读者领悟到测试驱动开发的好处，并将其纳入日常编码工作中，同时也希望读者能稍花点时间就掌握它，不要像笔者这样花费了 5 年左右的时间。有很多关于单元测试的书，其中某些书的作者也曾参与测试框架的编写与设计。这些书都很好，不过，它们都没有特别针对 Cocoa Touch 的开发者。笔者的这本书提供了以 Objective-C 语言所编写的范

⊖ 英文书名为《97 Things Every Programmer Should Know》，由 O'Reilly Media 于 2010 年出版。——译者注

例，讲述了 Xcode 及相关工具的用法，并以 Cocoa 的编程风格书写代码。但愿本书能够将测试驱动开发的原理讲得更加易懂，并使它与 iOS 开发者的工作联系起来。

另外，还有测试工具的问题。对于写单元测试有很多种工具可用，具体采用哪个，还要取决于众多不同的工具与框架所提供的功能。尽管本书会提到它们之间的一些差别，但是笔者打算专注于讲解 Apple 所提供 Xcode 开发环境及其所附 OCUnit 测试框架的用法。这么做的原因很简单，那就是适用性：任何想尝试单元测试或 TDD 的开发者，只要学习了本书所讲的知识，使用标准的开发工具，并具备一定的决心，就可以立刻投入工作中。如果读者觉得标准的测试工具缺少某些功能或者不太好用，那你当然可以研究其他工具的用法，甚至自己写一个测试工具——只是记得要测试它哦！

在成长为一个迷恋测试的程序员的过程中，我学到了很多。其中之一就是，想要当一个好的软件工程师，最好的办法就是与其他开发者交流。如果你对本书内容或者通用的 TDD 开发方法有任何评论或建议，欢迎在 Twitter 上与我交流讨论（我的用户名是 iamleeg[㊀]）。

致谢

牛顿曾说过，“如果我比别人看得更远，那是因为我站在巨人的肩上”，当然了，他的这个说法综合了数个世纪以来作者们不断拓展与完善的那个隐喻[㊁]。与此相似，本书也不是凭空写出来的，有很多要感谢的人，若全部写出他们的名字的话，要从爱达·勒芙蕾丝伯爵夫人[㊂]开始，用好多页才能写完。如果要写一份简洁一些的致谢表，那么首先要感谢的就是培生（Pearson）出版集团的诸位同仁，是他们促成了本书的出版发行。还要感谢 Chunk、Trina 与 Olivia，是他们持续督促我完成此书的。此外还有本书的技术审校者 Saul、Tim、Alan、Andrew、两位名为 Richard 的先生、Simon、Patrick 以及 Alexander，感谢你们在排查本书草稿中错误的过程中所做的出色工作，如果还有错误的话，则应归咎于我。感谢 Andy 与 Barbara，你们所做的文字润色工作，将一些由程序员草草写就的文字变成了优雅的英语文句。

感谢 xUnit 测试框架的设计者 Kent Beck 先生，没有他对于测试的见解，我将无处下笔，同样，也感谢 xUnit 框架的 Objective-C 版本制作方 Sente SA 公司。还必须提到的是 Apple 的开发工具制作团队，他们为了让全世界的 iOS 开发者能够使用单元测试所做的努力比其他人都要多。与别人相比，Kevlin Henney 更让我领略到了测试驱动开发之美，让我避免了很多 bug，谢谢你！

最后，感谢 Freya 在没日没夜的写书过程中对我的支持与理解。如果此刻你正在读这段话，我想你可能会更加了解我的。

㊀ 作者现已改为使用名叫 secboffin 的 Twitter 账户了。——译者注

㊁ 这个隐喻的本来形式是“Dwarfs standing on the shoulders of giants”（小矮人站在巨人的肩膀上），是由 Bernard of Chartres 首先写出的。参考 http://en.wikipedia.org/wiki/Standing_on_the_shoulders_of_giants。——译者注

㊂ Ada, Countess Lovelace（1815—1852），是著名英国诗人拜伦之女。被后人公认为第一位计算机程序员。——译者注

目　录

第 1 章　软件测试与单元测试简介

要想最大限度地从单元测试中受益，就必须理解它的目标及它是如何改进软件开发过程的。在本章中，读者将会学到一些通用的软件测试知识，这些知识也适用于单元测试。这一章也会讲到软件测试的优点和缺点。

1.1　软件测试的目标

很多软件项目的目标都是盈利，实现这个目标的通常方式即通过应用商店来出售软件或者以其他方式授权给用户使用并收取费用。那种为了程序开发者内部使用所制作的软件，则会通过提高某个业务流程的效率，减少该流程所耗的时间来间接地盈利。如果通过提高业务流程效率节省的成本大于开发该软件的花销，那么这个软件项目就是盈利的。开源软件的开发者通常以出售“支援服务包”（support package）来获利，他们也会使用自己开发的软件，在这种情况下，前面的论断依然成立。

所以说，软件开发经济学的基本原则就是，如果某个软件项目的目标是盈利——不管是向客户出售最终产品还是供开发者内部使用，那么它要想成功地达成此目标，必须创造某种高于软件制作开销的价值才行。笔者也知道这并非一个具有非凡意义的论断，不过可以将它推及到软件测试领域中。

如果软件测试（也叫做“质量保证”（Quality Assurance，QA））是为了支持软件项目，则它必须对实现盈利有帮助才行。这一点很重要，因为它对软件测试做出了限定：如果软件测试开销过大，导致项目亏损，那么这种测试就不适合去做。不过对软件进行测试可以保证产品能正常运行，而产品又包含了客户所需的功能。如果你不能展示这些功能的价值，那么客户就不会购买这个产品。

注意，测试的目标是证明产品能够正常运行，而不是发现 bug[⊖]。软件测试是在做“质量保证”，而不是“质量介入”。查找 bug 通常是个坏主意。为什么呢？因为要修复 bug 就必须有开销，而这部分资金本来是付给开发者的，让其一开始就写出无 bug 的软件，现在却被浪费

⊖ 中文为“程序错误”，是指在软件运行中因为程序本身有错误而造成的功能不正常、死机、数据丢失、非正常中断等现象。由于此词是常用的程序设计术语，故全书均保持英文写法，不再译出。——译者注

了。在理想的情况下，大家可能会认为开发者只需写出无 bug 的软件，通过快速的测试确保它们没有问题，然后将其上传到 iTunes Connect 账户，就可以坐等财源滚滚而来了。不过别急，这么做也会以另一种方式导致同样的问题：在测试软件之前，需要多长的时间来编写 100% 无 bug 的软件呢？这样做的开销是多少？

这么说的话，合适的软件测试方案看起来是一种折中：既要保证对软件开发进度有一定程度的控制，又要在工程开销许可的范围内进行一定程度的检查，以确保产品确实能够正常运行。这种平衡应该着眼于将所发行产品的运行风险降低到一个可以接受的水平上。所以说，“最具风险的组件”，也就是那些对于软件的运行至关重要的组件或者那些最有可能隐藏 bug 的组件，应当首先测试，然后测试那些风险稍低的组件，依次测试，直到你觉得所有剩下的风险因素都不值得再投入时间和资金去测试为止。最终的结果应该是让客户看到软件实现了预期功能，从而值得为此付费购买才对。

1.2 软件测试由谁来做

在早期的软件工程实践中，项目都是以图 1-1 所示的“瀑布模型”（Waterfall Model）[⊖]来管理的。在这种模型中，开发过程被划分为一个个独立的”阶段”（phase），上一个阶段的输出即是下一个阶段的输入。所以产品经理或业务分析师可以先建立项目需求，需求建立好之后，将其交给设计师和架构师，依此制作软件需求规格书（software specification）。开发者按照此规格书的要求来编写代码，编好的代码交给测试人员去做质量保证。最后，测试好的软件将会向客户发布（通常会先向一部分预先选定的客户分发，这些人叫做 beta tester [⊜]）。

这种软件项目管理方式将编码者与测试者强制隔开，这对实际的测试工作有利也有弊。好处是通过将代码编写与代码测试的职责分开，可以让更多的人来找 bug。有时候开发人员只将注意力集中在其所写的代码上，而旁观者则可以清楚地找出代码中的错误来。同样，如果需求或规格书中有某一部分描述不够明确，那么测试者与编码者就有机会以不同的方式来理解它，从而增加了其被发现的几率。

这么做的坏处则是成本会增加。表 1-1 是根据 Steve McConnell 所著的《Code Complete, 2nd Edition》（Microsoft Press，2004 年出版）[⊜]一书中的调查数据重制的，它将修复一个 bug 所需的大致花销表示为该 bug 在产品中所潜伏时间的函数。由此表可以看出，在项目快要结束时修复 bug 是开销最大的，这很有道理：测试员找到并汇报某个 bug，开发者必须理解并在源代码中将其定位。如果开发者已经从事这个项目一段时间了，那么就必须进行规格书与代码的复查。修复了 bug 之后的版本必须再次提交给测试者以确保问题确实已经得到解决了。

⊖ 事实上，很多软件项目（包括 iOS 应用程序）仍然是以这种方式进行管理的。不过读者不应该因此就拒绝相信瀑布模型已经是一种过时的历史事物了

⊜ 该名词尚未有统一的中文名称，大意为公众测试员。详情参考 http://en.wikipedia.org/wiki/Beta_tester#Beta。——译者注

⊜ 中文书名《代码大全（第 2 版）》，由电子工业出版社于 2006 年出版。——译者注

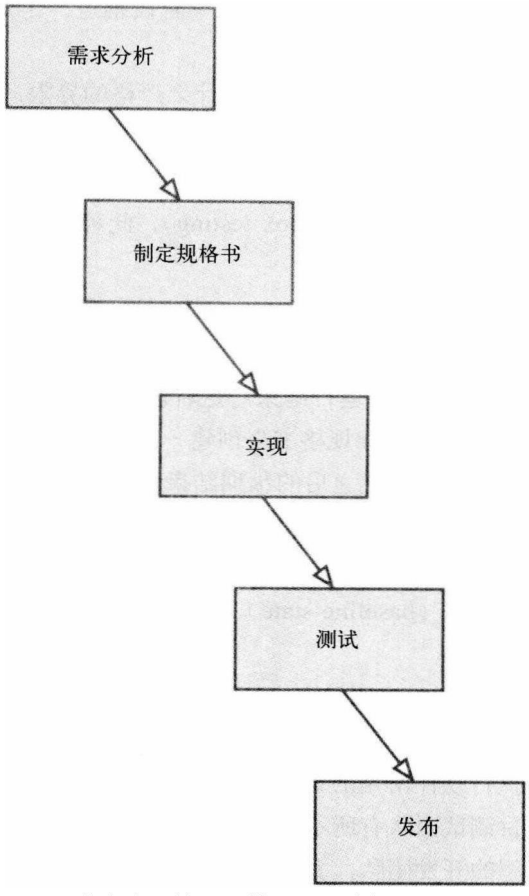


图 1-1　瀑布式软件项目管理流程中的各个开发阶段

表 1-1　在软件开发的不同阶段中修复 bug 的成本

修复 bug 的成本	检测 bug 的时间				
产生 bug 的时间	需求	架构	编码	系统测试	发行之后
需求	1	3	5 ～ 10	10	10 ～ 100
架构	—	1	10	15	25 ～ 100
编码	—	—	1	10	10 ～ 25

这额外的成本从何而来呢？很大程度上是源于不同团队之间的沟通：开发者和测试者使用不同的术语来描述同一个概念，他们会用完全不同的思维模式来理解应用程序中的同一个功能。如果发生了这种情况，那么就需要再花些时间来澄清这些模糊的问题。

从表 1-1 中还可以看出，在项目结束时修复 bug 的成本取决于这个 bug 是在哪个阶段产生的：在需求阶段埋下的错误只能通过重写整个功能来解决，这么做当然会花费很多了。这促使采用瀑布模型的人们在项目的早期持非常保守的态度，他们会彻底检查需求分析及软件规格

书，确保每一个细节都准确无误，否则就不向下一个阶段推进。于是这就导致了“分析麻痹”（analysis paralysis），它会增加项目成本的。

以这种方式划分开发者与测试者，就算不采用什么严格的界限，也会对软件测试的类型产生影响。因为测试者对应用程序内部细节的了解并不如开发者那样彻底，所以他们只会将整个产品看成一个不透明的部件，通过从外部与之沟通来进行“黑盒测试”（black box testing）。第三方测试者很少会进行“白盒测试”（white box testing），此种测试可以通过查看及修改内部代码来验证程序的运行结果。

在黑盒中执行的测试通常叫做系统测试（system test）或集成测试（integration test）。这是个正规的术语，意思是将软件产品当成一个整体来测试（也就是，整个软件系统的各个部件集成起来测试），测试主要针对软件的运行结果。这种测试通常按照预订的计划来执行，这也就是测试者谋生的方式：他们根据软件规格书来创建一系列测试用例，每个用例都会描述执行该测试所需的准备工作，以及执行测试之后的预期结果。这种测试通常是要手动执行的，当测试结果需要让测试者翻译时更是如此，因为测试者需要根据网络服务或当前日期等外部状态来判断测试结果。即便这种测试能自动化，通常运行起来也很耗时：每次测试前，都要把整个软件及其运行环境配置到基准状态（baseline state），而且每个步骤都有可能涉及非常耗时的数据库、文件系统或网络服务操作。

beta 测试（beta testing）[⊖]有时也叫做客户环境测试（customer environment testing），它实际上也是一种特殊的系统测试。其特殊之处在于参与测试的人也许并非专业的软件测试员，而是软件的用户。如果客户执行软件所用的系统配置或运行环境同测试者存在差别，或者用户执行用例所产生的结果与项目测试团队有所不同，那么在 beta 测试阶段都能发现，相关的问题也会回报给项目组。对于小型的开发团队，尤其是无力雇用专职测试员的团队来说，通过 beta 测试可以让软件首次面对不同的使用方式及运行环境所带来的考验。

因为 beta 测试是在产品即将发布时进行的，所以处理这种测试反馈信息对于项目开发团队来说是一种煎熬，他们觉得马上就要看到曙光了，都可以闻到庆功宴上比萨饼的香味了。不过话说回来，如果你不打算解决反馈回来的问题，那么也就毫无必要去做 beta 测试了。

开发者也可以自己来做测试。如果在 Xcode 中曾经按下 Build & Debug 按钮运行过程序的话，那么你就等于是做过了某种白盒测试，因为你是在通过检查代码来确定程序的行为是否正确（或者更准确地说，为什么程序运行结果不正确）。编译器产生的警告信息、静态分析器、Instrument [⊜]等工具都可以用于此种测试。

由开发程序员来做测试所具有的优点和缺点与请专职人员做测试所带来的利弊恰恰是相反的：开发者发现软件有问题后，通常能够以较小的成本轻易地解决它。因为他们已经对代码有了一定的理解，能够发现 bug 可能的藏身之处。实际上，开发者可以在编码过程中顺带进行测

⊖ 又名公众测试。——译者注

⊜ 由 Apple 所提供的用于跟踪、分析应用程序与操作系统运行状态的工具。详情参考：https://developer.apple.com/library/mac/#documentation/developertools/conceptual/InstrumentsUserGuide/Introduction/Introduction.html。——译者注

试，这样可以在写好代码之后就及时发现 bug。然而，如果 bug 是由于开发者对软件规格书或者业务范围不理解而产生的，那么必须借他人帮助才能找到这种 bug。

设置正确的运行环境

笔者所编写的代码中最值得一提的一个 bug（迄今为止，但愿以后也别再出现了）就是由于刚说的“开发者对需求不理解”而产生的。当时笔者在为 Mac 开发一款系统管理工具，它独立于用户账号而运行，所以并不会根据用户配置来决定其记录消息所使用的语言，而是从一个文件中读入语言设置。文件类似这样：

```
LANGUAGE=English
```

这看起来很简单。问题是，有些非英语用户回报说尽管已经设置成了其他语言，然而该工具还是会以英语来记录消息。笔者发现读取配置文件的那部分代码同该工具的其他代码之间耦合度很高，所以决定先打破这种依赖关系，然后再插入单元测试来检视代码的具体运行情况。慢慢地，笔者找到了导致语言检测逻辑失败的那段问题代码并将其修复。这样一来所有的单元测试都通过了，所以似乎代码能正常运行了，是吗？实际上不是的，因为笔者没有意识到有时候客户那边的配置文件可能是这样的格式：

```
LANGUAGE=en
```

不只是笔者没发现这个问题，测试人员也没发现，最后是因为程序在用户的操作系统上运行崩溃了才发现这个问题的，这种情况下就算单元测试覆盖了所有代码也发现不了它。

1.3 何时进行软件测试

上一节已经给出了这个问题的部分答案——在产品开发中越早执行测试，发现问题后解决它的开销就越低。如果开发过程中的各个阶段所写的代码都能够正确且稳定地运行，那么在后期将它们集成起来或加入新部件的时候，所产生的错误就要比等项目结束后再一次性执行所有测试要少。不过，上一节也提到，软件产品开发的习惯做法还是只在项目最后才进行测试：在实现阶段之后有一个隐式的 QA 阶段，然后软件会分发给公众测试员试用，最后才会发布正式版。

现今的软件项目管理方法认识到了传统做法的不足之处，力争在产品开发的全过程中对每个部件都进行持续的测试。这就是“敏捷”项目管理方式和传统管理方式之间的主要区别。敏捷管理方式经常将项目划分成数个短期工作阶段，每个阶段叫做一个“迭代”（iteration），有时也叫做“冲刺”（sprint）。每个迭代期都要对需求进行一次复查：将废弃的需求去掉，修改现有需求，根据需要新增一些需求。在该次迭代中，将选择那个最重要的需求，并针对它进行设计、实现和测试。在一轮迭代结束时，要对项目进度进行评估，以决定是将新开发好的这项功能加入到产品中，还是在这项功能上再增加新的需求，在接下来的迭代周期中完善此功能。最为重要的是，“敏捷宣言”（http://agilemanifesto.org/）声称“个体与互动的价值高于流程与工

具”，所以在做重要决策时，客户或其代表都会参与。如果能向客户问清楚软件的功能并向其确认开发好的产品确实能正 常运行，那么就不需要再费心思去雕琢冗长的产品功能规格书了。

于是，在敏捷项目的开发过程中，软件产品的所有内容都会持续地测试。在每个实现周期，客户都会被询问哪些需求是当前最重要的，然后开发者、分析员和测试者将会协同工作以完这些需求。敏捷软件项目管理采用的其中一种框架称为极限编程（eXtreme Programming，XP），它更进一步要求开发者对代码进行单元测试，并以结对的方式编码：一个人“抱着”键盘输代码，另一个人在旁边提出修改或完善代码的建议，并指出代码中潜在的缺陷。

所以说，问题的实质是要在项目开发的全过程始终进行软件测试。你并不能完全排除用户以非预期的方式使用软件的可能性，而且用户也可能会发现一些受工程期限或预算等因素制约而并未专门处理的逻辑所产生的 bug。不过你仍然可以针对自己所写的这部分代码进行自动化的常规测试，让 QA 团队和公众测试员去试着执行一些实验性的用例，看他们能不能以新奇而巧妙的方式使程序崩溃。而且你可以在每轮迭代结束之后询问客户接下来将要做的工作是否有助于提升产品的价值，是否能实现营销方案上所说的功能而让用户满意。

1.4 测试实践举例

前面已经提到系统测试了，它需要让专业的测试员根据所有用例逐次运行整个软件，以找出异常行为。在开发 iOS 应用程序时，可以利用 Apple 所提供的 Instrument 分析工具中所含的 UI Automation instrument 功能将这种测试在某种程度上自动化。

系统测试并非总是通过一般性的尝试来找到程序中的 bug，有时候测试者会预设一个特定的目标。渗透测试员（penetration tester）通过向程序输入恶意数据、打乱操作顺序或者破坏程序运行环境来寻找安全漏洞。可用性测试员（usability tester）会观察用户如何使用应用程序，记录下让用户操作错误、耗时太久或不知该如何操作的情景。可用性测试中的一个常用技巧就是 A/B 测试（A/B Testing）。让不同的用户使用不同版本的软件，再对使用情况进行统计和比较。Google 就以在软件中运用此种测试而闻名，它甚至会对产品界面所用的色调执行 A/B 测试。注意，可用性测试并不需要等程序全部开发完才能进行，Interface Builder 所创建的界面模型、Keynote 软件制作的幻灯片，甚至是纸上的草图都可以用来评判用户对应用程序界面的接受程度。这些真实度较低的界面虽说不能像在 iPhone 真机上操作那样暴露出一些微妙的细节来，但它们确实是一种及早获取用户反馈的方式。

开发者，尤其是在大型团队中工作的，会在将代码集成入产品之前，先将其提交给同事进行评审。这是一种白盒测试，其他开发者能看到代码，从而可以据此判断这段程序是否处理了某些特定的情况，是否将所有关键的偶发状况都考虑在内了。代码审校并非总能发现逻辑上的 bug，笔者觉得自己在参加过的代码审校中经常发现的是一些代码风格上的问题，以及一些无需改变代码行为即可解决的问题。当要求代码审校者查找某种特定的错误时（例如，按照一张包含 5 ～ 6 项常见错误的核查表来查找错误。在 Mac 与 iOS 代码的核查表中，通常会包含“保

留计数”[⊖]（retain count）问题），尽管他们可能找不到与这些核查项目不相关的问题来，但他们会更容易发现核查范围之内的 bug。

1.5 单元测试的适用范围

单元测试是开发者进行软件测试的另一个工具。第 3 章将会讲解如何设计与编写单元测试，现在只需要知道它是用于测试代码行为的一段小程序即可。这段程序会设置运行环境，然后执行受测代码，最后针对运行状态调用断言（assertion）语句。如果断言成立（也就是说，断言所述的条件得到满足），那么测试就能成功执行。如果实际运行结果与断言所述状态有任何偏差，则会导致测试用例执行失败，这种偏差也包括了由于异常而使程序无法正常运行完毕的情况[⊖]。

这样看来，单元测试有点儿像微缩版的集成测试用例，它也指定了运行程序的步骤及预期的执行结果，不过是以代码而非手工方式来做的。于是可以让电脑来完成这些工作，不必非得由开发者来手动执行每一个步骤。无论怎样，一个好的测试用例同时也应该是一份好的文档：它描述测试员对受测代码执行结果所持的预期。为程序中的某个类编写代码的开发者也可以编写测试用例以确保该类确实能如预期的那样工作。实际上，在下一章中我们会看到，开发者也能在写完受测类之前就先写好测试用例。

单元测试得名于其所测试的代码“单元”，在面向对象软件开发中，这样的“单元”通常指的就是类。这个专有名词源自编译器领域的术语“编译单元”（translation unit），意思是传递给编译器的单个文件。这意味着单元测试本来就是白盒测试，因为它是将应用程序中的某个类单独拿出来以测试其行为。可以将这个类当做一个黑盒，也可以仅通过公共 API 来与之通信，这是个人选择问题，然而不管是哪种方式，都只是在同应用程序的某个小部分进行互动。

单元测试的细粒度特质可以使开发者非常快速地解决通过运行单元测试所发现的问题。程序员在编写类代码的同时经常也在写该类的单元测试，所以在写测试代码时，就总是会考虑到受测类。笔者有时候甚至不用运行某个单元测试就可以知道它肯定运行失败，而且也知道该如何修改代码使之成功运行，这就是因为我那时的思维仍然在那个受测类上面。与之相对比的情形是，另外一个人来执行测试用例，而写受测程序的那个开发者已经好几个月没接触代码了。尽管开发者所写的单元测试并不会出现在应用程序中，但写测试代码所带来的好处会抵消掉这种开销，因为它可以在代码提交给专职测试员之前就发现并修复代码中的问题。

修复 bug 是每个项目经理都觉得最为恐怖的事情：因为必须在修复 bug 之后产品才能发布，然而又无法为 bug 的修复制订计划，因为无法确定存在多少个 bug 及修复它们需要多长时间。回头看看表 1-1 就知道，在项目即将结束时修复 bug 的成本最高，而且越往后，修复 bug 的成

⊖ retain count，意思为保留计数，是 Cocoa 库为了进行动态内存管理而在 NSObject 基类中所采取的一种对象计数机制。详情参阅：http://en.wikipedia.org/wiki/Cocoa_(API)#Memory_management。——译者注

⊖ 你所用的测试框架也许会将断言失败与执行错误分别列出，不过这没关系，关键是能发现测试用例并没有运行成功。

本就增长得越快。在估算工期的时候可以将写测试用例所花的时间考虑在内，这样在制作软件的过程中就能修复那些 bug，也会减少导致软件延期发行的不确定因素。

几乎可以肯定，单元测试是由程序开发者写的，因为写这种测试需要使用某种测试框架来编写代码，要调用 API，还要表述底层逻辑——这些都是程序员擅长做的事情。不过，某个类与其单元测试的代码则未必要由同一个程序员来写，将这两件事分开也有好处。资深程序员可以通过一系列测试来描述某个类 API 的预期行为，以此指导初级程序员对其编码。有了这些测试之后，初级程序员就编写类的代码依次让这些测试用例成功运行，由此来实现这个类。

集成的过程也可以这么倒置。如果开发者要使用或研究某个已经写好的类，但却不知道它的原理，那么可以将自己对其的假设以测试的形式写出来，由此判断哪些假设是正确的。随着测试用例的增多，这个类的功能与行为会更加完整地呈现出来。不过针对既有类来写测试，通常要比在编码的同时进行并行测试要难。要在测试框架中测试那种依赖于运行环境的类，要花费大量精力，因为必须要将其对周边对象的依赖替换或移除才行。第 11 章将会讲到如何针对既有代码写测试用例。

协同工作的程序员可以更快速地转换角色：一个人写测试，另一个人针对该测试编写实现代码，然后两人交换，由写实现代码的那个人来写测试。具体是哪两个程序员进行结对编程，并不重要。无论如何，都可以将一个或一组测试用例当做某种形式的文档，将某个开发者对该类的想法表达给其结对伙伴。

单元测试的一个关键优势就是它可以让测试的运行自动化。写一份好的手动测试计划书所花的时间，差不多和写一个好的测试用例所花的时间一样多，而电脑则可以在一秒钟内执行完上百个单元测试。开发者可以把其为某个应用所写的测试用例与该应用的产品代码放在同一个版本控制系统里面，这样无论何时想执行测试都会很方便。这使得检测回归 bug（regression bug）的成本变得非常低，这种 bug 指的是那些原本已修复，然而随着后期的开发工作又重新引入的 bug。只要修改了应用程序的代码，就应该将所有的测试用例执行一遍，以确保这次修改没有引入回归 bug。甚至可以通过某种持续集成系统（continuous integration system），在将代码提交至代码库时，自动运行测试用例。第 4 章会讲到这种用法。

反复运行测试不仅有助于找出回归 bug，它还会在要编辑源代码而又不改变其行为（也就是要重构应用程序）的时候，提供一张安全保护网。重构的目标是在不引入新功能及新 bug 的前提下对应用程序的代码进行清理，同时为了使其在将来的工作中更易使用，对其进行结构调整。如果正在重构的代码已经被足够多的单元测试所覆盖，那么任何行为上的改变都会检测到。这意味着可以在重构时就修复软件中的问题，而不是要等到下一个正式版发布之前（或之后）才去修复它们。

然而重构也不是万能的银弹（silver bullet）。如前所述，开发者没办法通过测试来确认他们是否正确地理解了需求。如果测试与实现是由同一个人来完成的，那么其对需求问题的认识和理解，在这两份代码中的体现将会是一致的。还应该意识到，并不存在一种合适的度量指标可以量化地判断某种单元测试方案是否成功。常用的标准（代码覆盖率和成功执行的测试用例