第一部分 Part 1

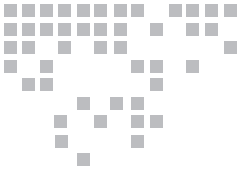
基　础

本书第一部分打算利用各个章节，建立读者和作者之间也就是你我之间共同的上下文。本书最终的目的是帮助你提高编写优秀测试的能力，第 1 章先从整体来看测试先行所带来的价值。然后讨论程序员生产力的动力学，以及各种对测试和测试质量的影响，最后会简单地介绍两种与自动化测试紧密相关的方法：**测试驱动开发**（Test-Driven Development，TDD）和**行为驱动开发**（Behavior-Driven Development，BDD）。

第 2 章挑起重担，定义了如何才能写出优秀的测试。简而言之，我们希望写出可读、可维护、可靠的测试。第 2 部分将会深入兔子洞，反转问题，检视一系列我们不希望看到的反例。

第一部分的末尾是第 3 章，它会谈及现代程序员最基本的工具之一——测试替身（test double）。我们将建立合理的用法，比如隔离代码以使其能够被恰当地测试，并且区分各种可能用到的测试替身的类型。最后，我们指导如何用好测试替身，帮助你绕开常见的坑，同时又能享受替身的好处。

读完前三章，你应当明白哪种测试是你希望编写的，以及为什么是那样的。你应当清楚地理解测试替身，将其作为常用工具。本书其余部分将根据这些基础展开，扶上马，送你一程。



Chapter 1 第 1 章

优秀测试的承诺

本章内容包括：

- 编写单元测试的价值
- 测试如何提高程序员的生产力
- 将测试用作设计工具

当我开始吃上编程这碗饭时，世界看起来与今日大不一样。那是 10 多年以前，人们使用着简单的 Vim 和 Emacs 等文本编辑器，而非如今的 Eclipse、Net Beans 和 IDEA 等集成开发环境。我清楚地记得，某位资深同事在调试软件时摆弄着 Emacs 宏，生成大量的 System.out.println 调用。我甚至还清楚地记得，当一个主要客户报告说他们的订单没有正常生成时，我们要把打印出的日志破译出来。

那时候，“测试”对大多数程序员来说意味着下面两件事之一——要么是由专人在自己完成编码之后所做的事情，要么是在声称编码完成之前对代码所捣鼓的事情。当引入一个 Bug 后，你会再次对自己的代码动手折腾一番——这次要增加一些日志，看看能否找出错误之处。

自动化曾是我们最先进的概念。我们用 makefiles 来重复地编译和打包代码，但尚未将运行自动化测试作为构建的一部分。我们曾用各种 shell 脚本来启动一两个“测试类”——操作生产代码的小程序，用来打印发生了什么，以及对于某些输入，代码的输出是什么。我们完全没有任何标准的测试框架，也没有自验证（self-verifying）的测试可以用来报告断言中的各种失败。

我们走过了漫长的日子，好不容易才到今天。

1.1　国情咨文[⊖]：编写更好的测试

下述概念如今已被广泛推荐，即开发者应该编写自动化测试，以便当发现回归问题时就使构建失败。而且，测试先行的编程风格已有大量的专业研究，使用自动化测试不仅是保护回归，而且是帮助设计，在编写代码之前就指出代码的期望行为，从而在验证实现之前先验证设计。

作为顾问，我见过很多团队、组织、产品和代码。看看今天的我们，很明显自动化测试已经成为主流。这很棒，因为没有自动化测试，大多数软件项目会比现在更糟。自动化测试改善了你的生产力，使你获得并保持开发速度。

救命！我是单元测试新手

如果你还不熟悉如何编写自动化测试，现如今是一个熟悉这种实践的好时节。Manning 出版社出了几本关于 JUnit 的书，那是编写 Java 单元测试的事实标准库，还有《JUnit in Action》（第 2 版，作者 Petar Tahchiev 等，2010 年 7 月出版），是测试各种 Java 代码的优秀入门教程，涵盖从简单 Java 对象到企业级 JavaBeans。

假如你在家自己编写单元测试，但却不熟悉 Java 或 JUnit，或许你该先看看本书的附录 A，这样在阅读例子时就不会有麻烦了。

自动化测试成为主流，并不意味着我们的测试覆盖率已达到理想状态，或者生产力无法再改善了。事实上，我在过去五年中的大量工作正是帮助人们编写测试，在编码之前写测试，特别是编写更好的测试。

为什么编写更好的测试这么重要？如果我们不注意测试的质量，那又怎样？我们现在谈谈测试带给我们什么价值，以及测试质量为什么重要。

1.2　测试的价值

来认识一下 Marcus。Marcus 是著名的编程奇才，两年前刚毕业，然后加入了本地一家投行的 IT 部门，为银行开发用于在线自助服务的 Web 应用程序。作为团队中最年轻的成员，起初他保持低调，集中精力学习银行的领域知识，熟悉“这里做事的方式”。

⊖ 国情咨文（State of the Union）是美国总统每年在美国国会联席会议召开之前于美国国会大厦中的众议院发表的报告。这里指下文对软件自动化测试的现状做概要介绍。——译者注

几个月后，Marcus 注意到团队的工作很多都是返工：修复程序员的错误。[㊀]他开始关注团队修复错误的类型，发现单元测试可以轻易地捕获到大多数的错误。当他感觉到哪里存在特别容易出错的代码时，Marcus 就对其编写单元测试。

测试帮助我们捕获错误。

一段时间以后，团队其他人也开始到处编写单元测试。Marcus 已被测试感染（test-infected）了，他碰过的代码几乎都具有相当高的自动化测试覆盖率。[㊁]除了在第一次时犯错，他们不会再花费时间修复过去的错误，待修复缺陷的总数在下降。测试开始清晰可见地影响着团队工作的质量。

自从 Marcus 编写第一个测试，已经过去近一年了。在前往公司圣诞派对的路上，他意识到时光匆匆，于是开始回想这段时间内发生的变化。团队的测试覆盖率在近几个月快速提高，达到了 98% 的分支覆盖率。

Marcus 曾认为他们应该推动那个数字直到 100%。但过去几周，他打定了主意——那些缺失的测试不会给他们带来更多价值，花费更多精力来编写测试不会带来额外的收益。许多没有测试覆盖的代码，只因所用的 API 要求实现某些接口，但 Marcus 的团队并未用到它们，那么何必测试那些空方法桩呢？

100% 测试覆盖率不是目标

100% 听起来肯定比 95% 要好，但是区别在于那些测试的价值对你可能是微不足道的。这要看哪种代码没有被测试覆盖，以及你的测试能否暴露程序的错误。100% 的覆盖率并不能够确保没有缺陷——它只能保证你所有的代码都执行了，不论程序的行为是否满足要求。与其追求代码覆盖率，不如将重点关注在确保写出有意义的测试。

团队已经达到稳定水平——曲线的平坦部分显示出额外投资的收益递减。在本地的 Java 用户组会议中，许多团队报告了类似的经验，并画出如图 1.1 所示的草图。

高级软件架构师 Sebastian 曾是投行的顾问，他改变了 Marcus 的想法。Sebastian 加入了自助服务团队，并快速地成为初级成员的导师，包括 Marcus 在内。Sebastian 这位老手貌似使用过几乎所有主要的编程语言，用以开发 Web 应用程序。Sebastian 编写单元测试的方式影响了 Marcus。

㊀ 出于一些原因，人们称之为错误、缺陷、bug 或问题。

㊁ 测试感染（test-infected）这个术语，由 Erich Gamma 和 Kent Beck 提出，出自 1998 年《Java Report》的文章，《Test-Infected：Programmers Love Writing Tests.》。

Marcus 习惯于先写代码，在提交到版本控制系统之前再补充单元测试。但 Sebastian 的风格是先写一个会失败（很明显是这样的）的测试，再写足够使测试通过的代码，然后写另一个失败的测试。他重复这个循环直到完成任务。

与 Sebastian 共事，Marcus 意识到自己的编程风格开始转变。他的对象结构不同了，代码看起来稍微不同了，只是因为他开始从调用者的角度来审视代码的设计和行为了。

测试帮助我们针对实际使用来塑造设计。

想到这些，Marcus 觉得好像明白了一些道理。当他试图用语言表达出他的编程风格是如何改变的，以及产生了哪些效果时，Marcus 明白了他写的测试不仅仅是质量保证的工具，或是对错误及回归的保护。测试作为一种设计代码的方式，提供了具体的目的。编写使失败测试通过的代码比以前的方式简单多了，而且该做的也都做了。

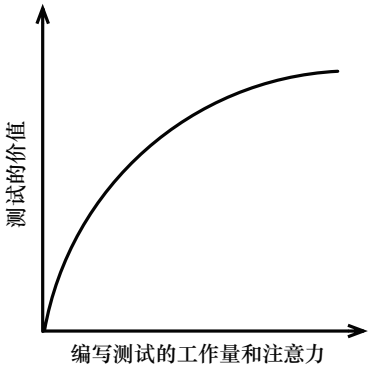


图 1.1　测试越多，额外测试的价值越少。第一个测试最有可能是针对代码最重要的区域，因此带来高价值与高风险。当我们为几乎所有事情编写测试后，那些仍然没有测试覆盖的地方很可能是最不重要和最不可能破坏的

通过明确地指出所需的行为，测试帮助我们避免镀金。

主人公 Marcus 的经历正如许多被测试所感染的编程人员一样，在不断地认识和理解测试。我们经常因为相信单元测试有助于避免尴尬、耗时的错误而开始编写它们。随后我们会学到，将测试作为安全网只是等式的一部分，而另一部分——或许是更大部分——其好处是我们将测试表达为代码的思考过程。

大多数开发者确实都理解了编写自动化测试的好处。不太能达成一致的是写多少和写哪种测试。在顾问工作中，我注意到大多数程序员都同意他们应该亲自去编写一些测试。相类似，我注意到几乎没有人认为应该达成 100% 覆盖率——那意味着测试将执行生产代码中每个可能的执行路径。

那么我们能够由此而做些什么吗？大多数人同意说编写一些测试是不费脑筋的。但随着我们接近完全的代码覆盖率，我们不那么确定了——我们差不多已经为一切都编写了测试，而剩下的没有测试的代码是微不足道，几乎不会破坏的。这就是某些人所谓的收益递减，通常显示为如图 1.1 所示的曲线。

换句话说，向代码基增加 100 个精挑细选的自动化测试是明显的改善，但当我们已有

30 000 个测试时，这些额外的 100 个测试就无足轻重了。投行的年轻程序员 Marcus 在与导师工作时发现，事情没那么简单。之前的原因不能支持所谓的**双稳态定律**（Law of the Two Plateaus）：

> 编写测试的最大价值不在于结果，而在于编写过程中的学习。

双稳态定律涉及 Marcus 的故事中对于测试的两层思考。为了实现测试的全部潜力，我们要爬两座山而非一座山，如图 1.2 所示。

只要我们认为测试仅仅是质量保证工具，我们的潜力就被图 1.2 中下面的弧线给限制住了。将我们对测试的认识转变为编程工具（一种设计工具）能促使我们从质量稳态移动到第二个以测试为设计工具的稳态。

遗憾的是，大多数代码似乎沿第一条曲线停滞不前，开发者并未发力去使用测试来驱动设计。相应地，对于不断增长的测试套件的维护成本，开发者没有引起足够的注意。

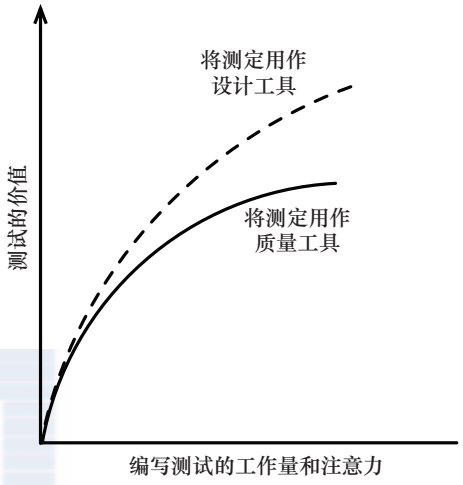


图 1.2 第一个稳态表明编写更多更好的测试不再带来额外的价值。第二个稳态进一步爬升，从我们想法的改变中发现更多的回报——将测试认为是丰富的资源，而不仅仅是验证工具

1.2.1 生产力的因素

从这里来看，最大的问题是哪些因素影响着程序员的生产力，在这个动态中测试扮演什么角色？

编写测试最快的方式是以最快的速度打字，但却不关心代码中重要部分——测试代码——的健康。然而，我们马上将会讨论为什么应该花时间培育你的测试代码、对重复部分做重构、普遍地注意它的结构、清晰度和可维护性。

测试代码往往天生就比生产代码简单。[⊖]无论如何，如果你偷工减料，在测试代码的质量上留下技术债务，它将会把你拖慢。本质上，测试代码的重复和多余的复杂性会降低你的生产力，抵消测试带来的正面影响。

这不仅关乎你的测试可读性，还有可靠性和可信赖性，以及运行所需的时间。图 1.3 展示了测试周围的影响力系统。

⊖ 例如，很少有人在测试中使用条件语句、循环等。

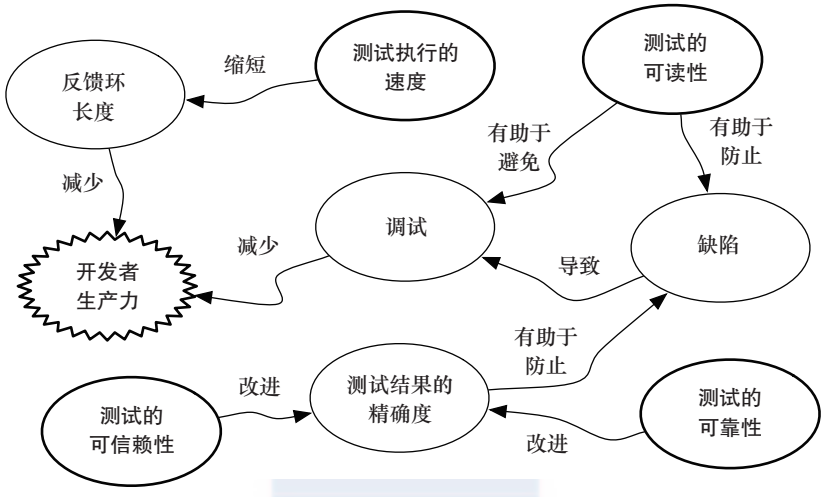


图 1.3　多个关于测试的影响力，它们直接或间接地影响生产力

注意我在图中标出的两个对生产力的直接影响，它们是**反馈环长度**和**调试**。在我的经验中，这两者是在键盘上消耗程序员时间的罪魁祸首。[⊖]如果你在错误发生后迅速学习，那么花在调试上的时间是可以大幅避免的返工——同时，你的反馈环越长，花在调试上的时间越多。

攀升的 bug 修复成本

你认为修复一个 bug 的平均成本是多少？伦敦召开的 2009 XP 日会议上，Google 的 Mark Striebeck 报告了 Google 对延迟修复缺陷的成本估计。

Google 估算出在程序员引入 bug 后马上修复它要花费 5 美元。同样的缺陷，如果当时逃脱了程序员的眼睛，而是在运行了整个构建后才发现它，那么要花费 50 美元来修复。如果在集成测试时发现这个缺陷，成本飙升至 500 美元，而到了系统测试阶段，成本高达 5000 美元。

听听这些数字，无疑最好还是尽快找到 bug 吧！

等待对变更进行确认和验证在很大程度上牵扯到**测试执行的速度**，这是图 1.3 中用粗体强调的根本原因之一。另外三个根本原因都会影响程序员的**调试量**。

缺乏**可读性**自然会降低你的分析速度，并且鼓励你打开调试器，因为阅读测试代码不会让你明白——太难理解了。缺乏可读性也会引入更多的**缺陷**，因为很难看出你造成的错误，

⊖ 有许多人说会议是生产力的终极杀手，不过那是在别的书里。

而缺陷会导致更多的调试工作。

缺陷逃逸数量增多的另一个因素是**测试结果的准确度**。你若要能够依赖于测试套件去识别引入的错误，那么准确度是一个基本要求。剩下的两个影响生产力的测试相关根本原因，都会直接影响测试准确度，它们叫做**可信赖性**和**可靠性**。为了测试报告的准确性，测试需要断言它们的承诺，并以可靠、可重复的方式提供断言。

通过关注对生产力的影响力，你可以扩展质量稳态。使程序员变得高效的关键正是这些已经识别出的根本原因。高产的先决条件是你足够了解你的工具，而不是持续分散你的注意力。一旦你了解了编程语言，你可以浏览其核心 API。当你熟悉了问题域，就专注于根本原因——可读性、可靠性、可信赖性和测试的速度。

这也是本书剩余大部分将要围绕的内容——帮助你提高对测试代码可读性、可信赖性和可靠性的意识和感觉，并确保你能持续使用这种工作方式，以确保可维护性。

在那之前，我们来解释图 1.2 中的第二条曲线。

1.2.2 设计潜力的曲线

假设你先写了最重要的测试——针对最常见和基本的场景，以及软件架构中的关键部位。

你的测试质量很高，你大胆地将重复都重构掉，保持测试精益且可维护。现在想象一下，你积累了如此高的测试覆盖率，唯一没测到的地方只是那些直接针对字段的访问器。平心而论，为那些地方写测试没什么价值。因此，之前的做法倾向于收益递减——你已经不能再从“仅仅”编写测试这件事中获取价值了。

这是由于我们不做的事情而造成的质量稳态。之所以这么说是因为想要达到更高的生产力，你需要换个思路去考虑测试。为了找回丢掉的潜力，你需要从编写测试中找到完全不同的价值——价值来自于创新及设计导向，而非防止回归缺陷的保护及验证导向。

总而言之，为了能同时达到两个稳态，从而完全发挥测试的潜力，你需要：

1. 像生产代码一样对待你的测试代码——大胆地重构、创建和维护高质量测试，你自己要对它们有信心。

2. 开始将测试作为一种设计工具，指导代码针对实际用途进行设计。

如前所述，前者造成了大多数程序员在编写测试时会不知所措，无法顾及高质量，或降低编写、维护、运行测试的成本。这也是本书的重点——编写优秀的测试。

也就是说我们不会花很多时间去讨论利用测试作为设计的方面。[㊀]我只是想让你对这种

㊀ 碰巧我写过一本关于该主题的宝书——《测试驱动开发的艺术》（Test Driven）。另一本不错的书是《测试驱动的面向对象软件开发》（Growing Object-Oriented Software，Guided by Tests，作者 Steve Freeman，Nat Pryce，机械工业出版社，2010）。

动态和工作方式有个全面的了解，那我们详细说说这个话题再前进吧。

1.3　测试作为设计工具

传统上，程序员编写的自动化测试被看做是质量保证工作，用于在编写的时候验证实现的正确性，以及将来代码进化的时候验证正确性。这就是将测试作为验证工具——你设想一份设计，编写代码实现，编写测试验证实现是否正确。

使用自动化测试作为设计工具将世界颠倒过来了。当你用测试设计代码时，你将典型的“设计，编码，测试”序列变换为“测试，编码，设计”。是的，就是那样。测试先于编码，并以追溯性的设计活动来得出结论。那结论性的设计活动称为重构，序列变为“测试，编码，重构”，如图 1.4 所示。

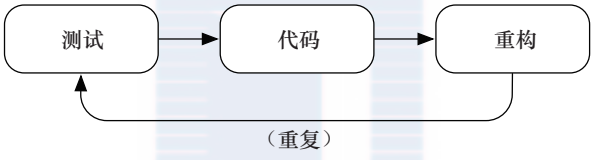


图 1.4　测试驱动开发的过程，先是编写失败的测试，编写代码以通过测试，重构代码来改进设计

如果你对这有点耳熟，大概是听说过测试先行编程或测试驱动开发（Test-Driven Development，TDD）。[⊖]这就是我们所要讨论的，那我们也这么叫吧。

1.3.1　测试驱动开发

TDD，如图 1.5 所示，是一种很有章法的编程技术，它基于一个简单的想法：在编写出能够证明代码存在的失败测试之前，不写生产代码。这也是它有时被称为测试先行编程的原因。

不止这些。先写测试，会向测试所期望的方向来驱动生产代码的设计。这会带来以下令人满意的后果：

- 代码变得可用——代码的设计和 API 适合于你的使用场景。
- 代码变得精益——生产代码仅仅实现场景所需要的功能。

首先，无论你工作在系统蓝图的哪一部分，无论其他组件、类或接口存在与否，你一定是在为一个具体的场景来设计解决方案。你将该场景翻译为一个可执行的例子，以自动化单元测试的方式。运行测试，看着它失败，你具有了一个使之通过的清晰目标，只编写足够的生产代码——不要多写。

⊖　或者你读过 Marcus 及其导师 Sebastian 的故事，就在前几页……

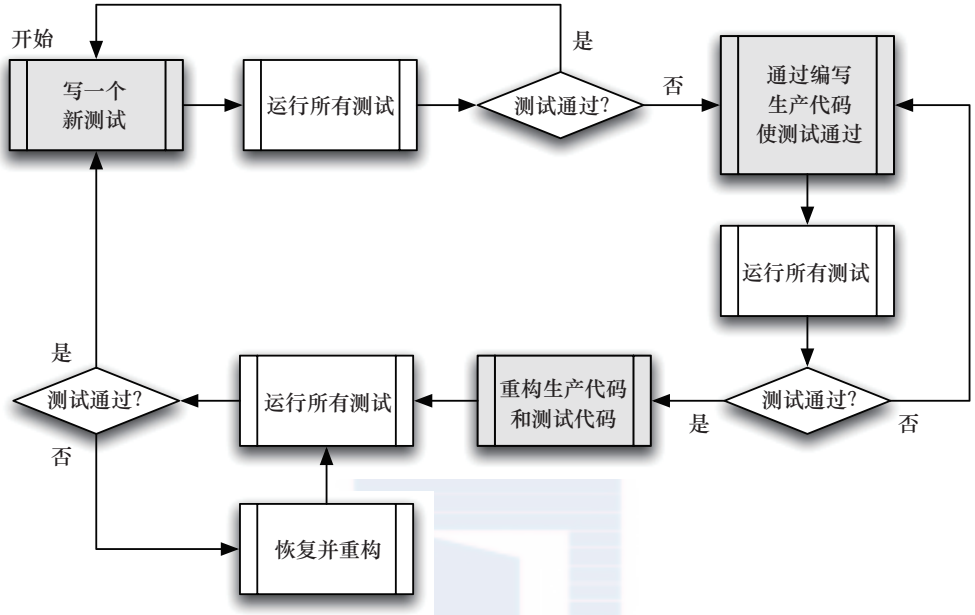


图 1.5 测试驱动开发是一个循环过程。先写失败的测试，使之通过，然后重构代码使意图更明确，同时减少重复。每一步过程中，你都不断地运行测试，确保目前的进展

将场景刻画为可执行的测试代码是一种设计行为。测试代码成为生产代码的调用者，使你在还没写任何代码之前就验证你的设计。用具体例子的形式来表达需求是一种强大的验证工具——我们只能通过将测试作为设计工具才能获取价值。

其次，严格遵循规则，仅仅编写足以使测试通过的代码，你会保持设计简单并适合目的。没有镀金的迹象，因为任何代码都具有测试——场景——来覆盖和保证。代码质量罪魁祸首之一，以及使开发者生产力停滞的主要因素，就是称为**偶发复杂性**。

偶发复杂性是不必要的复杂性。可以通过替换为简单的设计来避免它，同时仍然满足需求。有时我们喜欢通过产出这种复杂设计来展示自己的精神能力，却使自己都理解不了那些设计。我打赌你也认识到自己的原始本能了。复杂设计会扼杀生产力，偶发复杂性也会事与愿违。

测试指出缺陷或代码中缺失的功能，只写足以使测试通过的代码，严格地向简单设计[⊖]来重构，这三者的组合极其强大，可以将偶发复杂性消灭在萌芽中。这不是魔法药剂，程序员的设计观念和经验会影响你最终得到的设计。

关于 TDD 远不止这些——很多书整本都在讲这种方法，包括我自己写的《测试驱动开发的艺术》（Test Driven），以及最近的《测试驱动的面向对象软件开发》（Growing Object-

⊖ 简单设计与过分简单的（simplistic）设计是不同的。

Oriented Software，Guided by Tests，作者 Steve Freeman，Nat Pryce）。如果你想要更紧凑的 TDD 介绍，我推荐你读《测试驱动开发：实战与模式解析》[㊀]（Test Driven Development：By Example，作者 Kent Beck，2004）。

有这么多具体对 TDD 的介绍，我们这里就长话短说，你可以参考那些书来进行更深入的探索。但我在这里讨论 TDD 的原因是，它与我们关于优秀测试的主题紧密相关——称为**行为驱动开发**（Behavior-Driven Development，BDD）的编程风格。

1.3.2 行为驱动开发

你可能听说过 BDD，即行为驱动开发。虽然过去的几十年间，人们曾用过测试先行的方式，但 20 世纪 90 年代才真正提出测试驱动开发的方法。

大约 10 年过去了，身在伦敦的顾问 Dan North，首先意识到了 TDD 思想和词汇表中所讲的“测试”会误导人们，然后成功地将测试先行编程推进了一步。Dan 将这种风格的 TDD 命名为行为驱动开发，在他 2006 年发表于《Better Software》的文章中，他是这样介绍 BDD 的：

> 我突然想到人们对 TDD 的误解几乎总是回到“测试”这个词。
>
> 并不是说测试不是 TDD 的本质——所产生的函数和方法是一种有效确保代码工作的方式。然而，如果函数和方法不能全面地描述系统的行为，那么它们会带给你一种虚假的安全感。
>
> 在我处理 TDD 时，我开始使用“行为”替代“测试”一词，我发现它似乎不仅合适，而且所有的教练（coaching）问题都奇妙地解决了。我现在对 TDD 的一些问题有了答案。给测试命名变得简单——一句话描述下一个你感兴趣的行为。测试的数量变得毫无意义——你只能在一句话中描述这么多行为。当测试失败，你仅需要重走一下上述流程——要么你引入了一个 bug，要么行为改变了，或者测试不再是相关的。
>
> 我发现从思考测试到思考行为，这种转换如此深刻，我开始改称 TDD 为 BDD 或行为驱动开发。

Dan 开始编写和讨论 BDD 之后，全球软件开发者社区中的其他人也纷纷将想出的各种实例驱动、行为需求导向的想法融入 Dan North 的 BDD 中。于是，今天的 BDD 语境和领域远远超出了代码——最引人注目的是将 BDD 提升到需求层面，与业务分析和需求行为结合起来。

㊀ 本书已由机械工业出版社引进出版，ISBN：978-7-111-42386-7。

被 BDD 实践者证明是有效的一个特殊概念，是利用验收测试进行由外向内的开发。这是《Cucumber：行为驱动开发指南》(The Cucumber Book：Behaviour-Driven Development for Testers and Developers）一书作者 Matt Wynne 和 Aslak Hellesøy 所描述的：

> 通过将优秀的 TDD 实践者的良好习惯正式化，测试驱动开发衍生出了行为驱动开发。优秀的 TDD 实践者由外向内地思考，他们先编写一个失败的客户验收测试，用于从客户视角描述系统。作为 BDD 实践者，我们小心地以例子的形式编写验收测试，使任何团队成员都能够理解。我们遵循这个过程，编写例子来从业务干系人那里获得反馈，在动手之前就能了解我们是否在构建正确的东西。

作为这种开发方式的佐证，许多 BDD 工具和框架如雨后春笋般，纷纷将基础的想法、实践和约定嵌入到软件开发者的工具链中。我们会在第 8 章见到那些工具。

当我们在本书说到“优秀测试”时，记住 Dan 的领悟，注意措辞。词汇表很重要。

1.4 小结

“牛仔式编程”由来已久，那时候程序员随意地编码，就算从马鞍上跌落也绝不写测试来接住自己。如今，开发者测试与自动化测试已司空见惯，即使不是标准实践，也绝对是热门话题。为你的代码编写彻底的自动化测试套件，其价值是不可否认的。

本章中你熟悉了双稳态定律。第一个稳态，程序员从测试中获得的价值是有限的，因为你已经具备了完全的测试覆盖率。但你还可以更进一步。

注重测试质量的程序员不只关注第一个稳态，而是瞄准顶峰。具备测试是一回事，具备优秀测试是另一回事。这就是你阅读本书的原因，也是我们将在剩下的章节中接受的挑战。

毕竟，如果你一路登顶而非停留在半山腰，你肯定会获得比登山更多的东西。