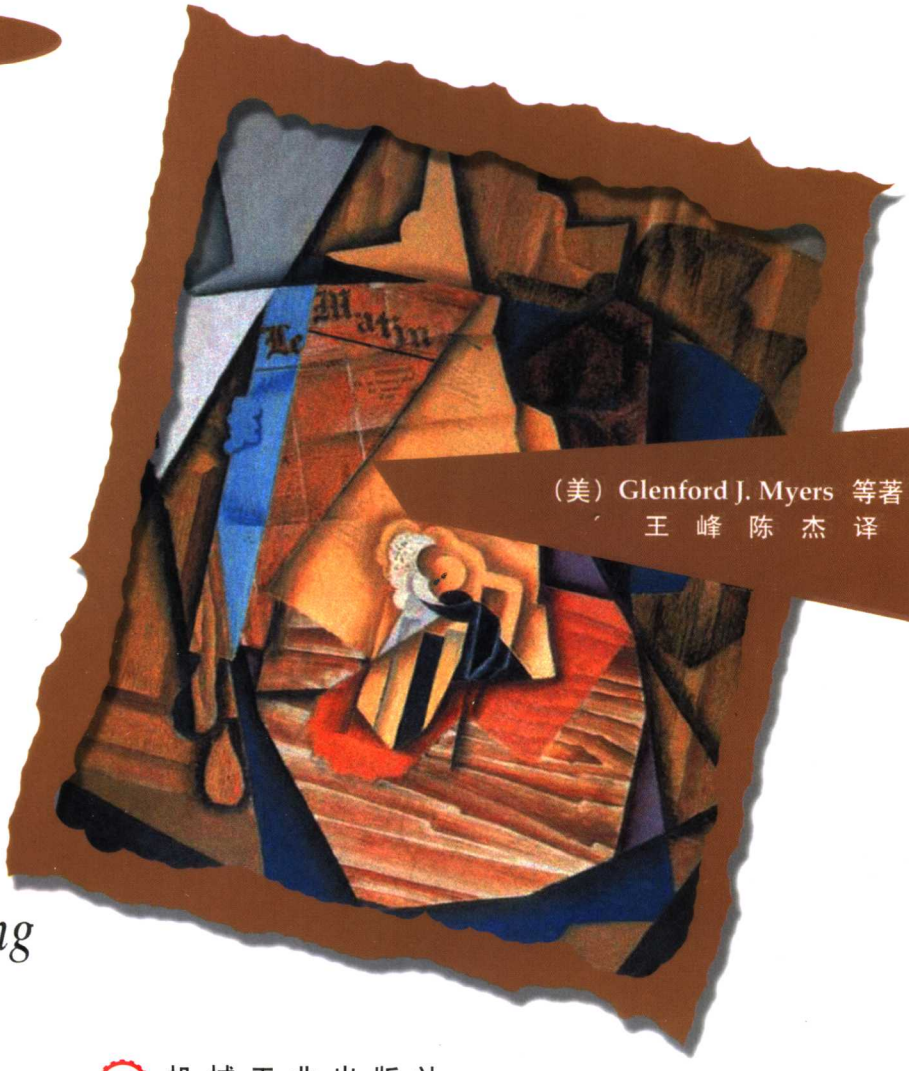
软 件 工 程 技 术 丛 书

HZ BOOKS

软件测试的艺术

（原书第2版）

软件测试系列



（美）Glenford J. Myers 等著
王 峰 陈 杰 译

The Art of Software Testing

(Second Edition)

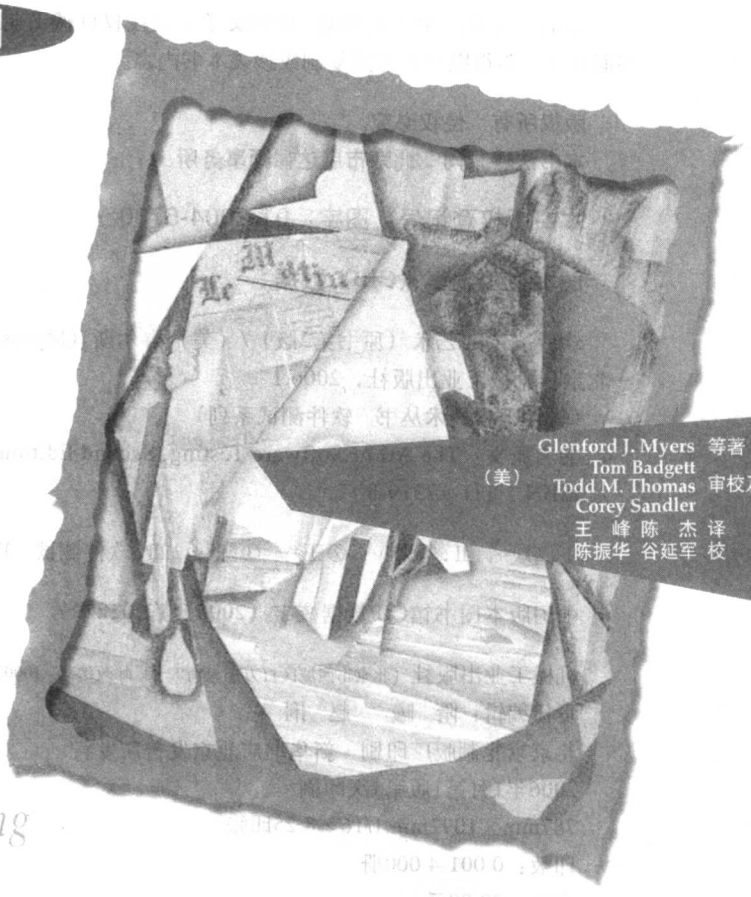
机械工业出版社
China Machine Press

软 件 工 程 技 术 丛 书

软件测试的艺术

（原书第2版）

软件测试系列



（美）Glenford J. Myers 等著
Tom Badgett
Todd M. Thomas 审校及更新
Corey Sandler
王 峰 陈 杰 译
陈振华 谷延军 校

The Art of Software Testing
(Second Edition)

机械工业出版社
China Machine Press

本书以一次自评价测试开篇，从软件测试的心理学和经济学入手，探讨了代码检查、走查与评审、测试用例的设计、模块（单元）测试、系统测试、调试等主题，以及极限测试、因特网应用系统测试等高级主题，全面展现了作者的软件测试思想。本书是软件测试领域的佳作，其结构合理、内容简洁、语言流畅。本书适合作为软件测试从业人员的参考手册，以及高等院校软件测试课程的教材或参考书。

Glenford J. Myers, et,al.:The Art of Software Testing, Second Edition (ISBN:0-471-46912-2).

本书版权登记号：图字：01-2004-5730

图书在版编目（CIP）数据

软件测试的艺术（原书第2版）/（美）梅尔斯（Myers, G. J.）等著；王峰，陈杰译. -北京：机械工业出版社，2006.1
（软件工程技术丛书　软件测试系列）
书名原文：The Art of Software Testing, Second Edition
ISBN 7-111-17319-8

Ⅰ.软…　Ⅱ.①梅…　②王…　③陈…　Ⅲ.软件测试　Ⅳ.TP311.5

中国版本图书馆CIP数据核字（2005）第102229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：隋　曦　　赵　俐
北京京北制版厂印刷·新华书店北京发行所发行
2006年1月第1版第1次印刷
787mm×1092mm 1/16·8.25印张
印数：0 001-4 000册
定价：22.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

译者序

按照一般的规矩，出版社的编辑要我作为译者为本书的中译本写一个“译者序”。说什么呢？

书中的真知灼见，是作者原创的。作为译者，我们只是把他们的思想尽量用我们的母语原封不动地传达给读者，实在没有什么非说不可的东西。是好是坏，全在书中，读了之后自有论断。

然而，当我翻译完原书封底最后一行文字的时候，还是想对读者朋友们说几句话。

首先，这是我表示不再翻译“软件测试”同类书之后，又“旧病重犯”接的活。两年前，我们应机械工业出版社华章分社编辑之约，翻译了Cem Kaner等人的《Testing Computer Software》第2版，由机械工业出版社和中信出版社于2004年5月出版。当时翻译工作基本上是在编辑的不断催促下在八小时工作以外完成的，很辛苦。结束之后，觉得太累，决定这种活以后不干了。2004年12月，编辑打电话约我去华章分社，递给我这本Glenford J. Myers的《The Art of Software Testing》。又是一个第2版，还是自1979年首版25年之后的第2版，当时我想这肯定是本好书。在我的记忆当中，除了大学的教材，计算机专著很少能“活”25年的。因此，当编辑问我翻不翻译时，我不加思索地答应了。另外，心想这本书薄，肯定没有翻译上本书那么累。

其次，这是我惟一一本从头到尾彻底看完的外文专业书，正文内容至少读了三遍，有些话反复揣摩，不下五遍。在我桌上一直放着商务印书馆出的两本词典，一本是中国人编的《英华大词典》，一本是外国人编的《朗文当代高级英语辞典（英英.英汉双解）》，为了翻译得准确些，很多单词我都同时查过这两本词典。在本书中，作者提出了“测试是为发现错误而执行程序的过程”、“测试的目标是建立‘软件做了其应该做的，未做其不应该做的’信心”等观点，与众不同。同时书中明确定义并区分了“代码检查”、“代码走查”、“同行评审”等概念，这在一般的书中是很少讲到的。由于工作的原因，译者参加过很多软件的测试工作，曾自认对软件测试还有所了解，自从读完本书之后，深深感到自己的认识原来是多么肤浅，所完成工作的质量原来是多么有待提高。当然，不要指望读了一两本书就能改变现状。

再次，关于全书的内容和结构，浏览一下目录就清楚了，我在这里就不赘言了。这本书“活”了25年，自有它的道理，请慢慢体会。

最后，如果条件允许，最好阅读原著。译者水平再高，也很难将原书所有的优美之处完全准确地表达出来。

本书由王峰、陈杰翻译，陈振华校读了全部译稿的内容，谷延军校读了前五章译稿的内容。感谢机械工业出版社华章分社的编辑对我们翻译工作的鼓励以及对我们拖延进度的宽容。中译本中的错误由译者负责。

王　峰

2005年10月15日于北京

前　言

1979年，Glenford J. Myers出版了一本现在仍被证明为经典的著作，这就是本书的第1版。本书经受住了时间的考验，25年来一直被列在出版商可供书目的清单中。这个事实本身就是对本书稳定、基础和珍贵品质的佐证。

在同一时期，本书第2版的几位合著者共出版了120余本著作，大多数都是关于计算机软件的。其中有一些很畅销，再版了多次（例如Corey Sandler的《Fix Your Own PC》自付梓以来已出版到第7版，Tom Badgett关于微软PowerPoint及其他Office组件的著作已经出版到第4版以上）。然而，这些作者的著作中没有哪一本书能够像本书一样持续数年之后仍畅销不衰。

区别究竟在哪里呢？这些新书只涵盖了短期性的主题：操作系统、应用软件、安全性、通信技术及硬件配置。20世纪80年代和90年代以来的计算机硬件与软件技术的飞速发展，必然使得这些主题频繁地变动和更新。

在此期间出版的有关软件测试的书籍已数以十计、甚至数以百计。这些书也对软件测试的主题进行了简要的探讨。

然而，本书为计算机界一个最为重要的主题提供了长期、基本的指南：如何确保所开发的所有软件做了其应该做的，并且同样重要的是，未做其不应该做的？

本书第2版中保留了同样的基本思想。我们更新了其中的例子以包含更为现代的编程语言。我们还研究了在Myers编著本书第1版时尚无人了解的主题：Web编程、电子商务及极限编程与测试。

但是，我们不会忘记，新的版本必须遵从其原著，因此，新版本依然向读者展示Glenford Myers全部的软件测试思想，这个思想体系以及过程将适用于当今乃至未来的软件和硬件平台。我们也希望本书能够顺应时代，适用于当今的软件设计人员和开发人员掌握最新的软件测试思想及技术。

引　言

在本书1979年第1版出版的时候，有一条著名的经验，即在一个典型的编程项目中，软件测试或系统测试大约占用50%的项目时间和超过50%的总成本。

25年后的今天，同样的经验仍然成立。现在出现了新的开发系统、具有内置工具的语言以及习惯于快速开发大量软件的程序员。但是，在任何软件开发项目中，测试依然扮演着重要角色。

在这些事实面前，读者可能会以为软件测试发展到现在不断完善，已经成为一门精确的学科。然而实际情况并非如此。事实上，与软件开发的任何其他方面相比，人们对软件测试仍然知之甚少。而且，软件测试并非热门课题，本书首次出版时是这样，遗憾的是，今天仍然如此。现在有很多关于软件测试的书籍和论文，这意味着，至少与本书首次出版时相比，人们对软件测试这个主题有了更多的了解。但是，测试依然是软件开发中的“黑色艺术”。

这就有了更充足的理由来修订这本关于软件测试艺术的书，同时我们还有其他一些动机。在不同的时期，我们都听到一些教授和助教说：“我们的学生毕业后进入了计算机界，却丝毫不了解软件测试的基本知识，而且在课堂上向学生介绍如何测试或调试其程序时，我们也很少有建议可提供。”

因此，本书再版的目的与1979年时一样：填充专业程序员和计算机科学学生的知识空缺。正如书名所蕴涵的，本书是对测试主题的实践探讨，而不是理论研究，连同了对新的语言和过程的探讨。尽管可以根据理论的脉络来讨论软件测试，但本书旨在成为实用且“脚踏实地”的手册。因此，很多与软件测试有关的主题，如程序正确性的数学证明都被有意地排除在外了。

本书第1章介绍了一个供自我评价的测试，每位读者在继续阅读之前都须进行测试。它揭示出我们必须了解的有关软件测试的最为重要的实用信息，即一系列心理和经济学问题，这些问题在第2章中进行了详细讨论。第3章探讨的是不依赖计算机的代码走查或代码检查的重要概念。不同于大多数研究都将注意力集中在概念的过程和管理方面，第3章则是从技术上“如何发现错误”的角度来进行探讨。

聪明的读者都会意识到，在软件测试人员的技巧中最为重要的部分是掌握如何编写有效测试用例的知识，这正是第4章的主题。本书第5章和第6章分别探讨了如何测试单个模块或子程序及测试更大的对象，而第7章则介绍了一些程序调试的实用建议，第8章讨论了极限编程和极限测试的概念，第9章介绍了如何将本书其他章节中详细讨论的软件测试的知识运用到Web编程，包括电子商务系统中去。

本书面向三类主要的读者。尽管我们希望本书中的内容对于专业程序员而言不完全是新的知识，但它应增强专业人员对测试技术的了解。如果这些材料能使软件人员在某个程序中多发现一个错误，那么本书创造的价值将远远超过书价本身。第二类读者是项目经理，因为本书中包含了测试过程管理的最新的、实用的知识。第三类读者是计算机科学的学生，我们的目的在于向学生们展示程序测试的问题，并提供一系列有效的技术。我们建议将本书作为程序设计课程的补充教材，让学生在学习阶段的早期就接触到软件测试的内容。

Glenford J. Myers
Tom Badgett
Todd M. Thomas
Corey Sandler

目　录

第1章 一次自评价测试

自本书25年前首次出版以来，软件测试变得比以前容易得多，也困难得多。

软件测试何以变得更困难？原因在于大量编程语言、操作系统以及硬件平台的出现。在20世纪70年代只有相当少的人使用计算机，而今天在商业界和教育界，如果不使用计算机，几乎没有人能完成日常工作。况且，计算机本身的功能也比以前增强了数百倍。

因此，我们现在编写的软件会潜在地影响到数以百万计的人，使他们更高效地完成工作，反之也会给他们带来数不清的麻烦，导致工作或事业的损失。这并不是说今天的软件比本书第一版发行时更重要，但可以肯定地说，今天的计算机——以及驱动它的软件——无疑已影响到了更多的人、更多的行业。

就某些方面而言，软件测试变得更容易了，因为大量的软件和操作系统比以往更加复杂，内部提供了很多已充分测试过的例程供应用程序集成，无须程序员从头进行设计。例如，图形用户界面（GUI）可以从开发语言的类库中建立起来，同时，由于它们是经过充分调试和测试的可编程对象，将其作为用户应用程序的组成部分进行测试的要求就减少了许多。

所谓软件测试，就是一个过程或一系列过程，用来确认计算机代码完成了其应该完成的功能，不执行其不该有的操作。软件应当是可预测且稳定的，不会给用户带来意外惊奇。在本书中，我们将讨论多种方法来达到这个目标。

好了，在开始阅读本书之前，我们想让读者做一个小测验。

我们要求设计一组测试用例（特定的数据集合），适当地测试一个相当简单的程序。为此要为该程序建立一组测试数据，程序须对数据进行正确处理以证明自身的成功。下面是对程序的描述：

> 这个程序从一个输入对话框中读取三个整数值。这三个整数值代表了三角形三边的长度。程序显示提示信息，指出该三角形究竟是不规则三角形、等腰三角形还是等边三角形。

注意，所谓不规则三角形是指三角形中任意两条边不相等，等腰三角形是指有两条边相等，而等边三角形则是指三条边相等。另外，等腰三角形等边的对角也相等（即任意三角形等边的对角也相等），等边三角形的所有内角都相等。

用你的测试用例集回答下列问题，借以对其进行评价。对每个回答“是”的答案，可以得1分：

1. 是否有这样的测试用例，代表了一个有效的不规则三角形？（注意，如1，2，3和2，5，10这样的测试用例并不能确保“是”的答案，因为具备这样边长的三角形不存在。）

2. 是否有这样的测试用例，代表一个有效的等边三角形？

3. 是否有这样的测试用例，代表一个有效的等腰三角形？（注意如2，2，4的测试用例无效，因为这不是一个有效的三角形。）

4. 是否至少有三个这样的测试用例，代表有效的等腰三角形，从而可以测试到两等边的所有三种可能情况（如3，3，4；3，4，3；4，3，3）？

5. 是否有这样的测试用例，某边的长度等于0？

6. 是否有这样的测试用例，某边的长度为负数？

7. 是否有这样的测试用例，三个整数皆大于0，其中两个整数之和等于第三个？（也就是说，如果程序判断1，2，3表示一个不规则三角形，它可能就包含一个缺陷。）

8. 是否至少有三个第7类的测试用例，列举了一边等于另外两边之和的全部可能情况（如1，2，3；1，3，2；3，1，2）？

9. 是否有这样的测试用例，三个整数皆大于0，其中两个整数之和小于第三个整数（如1，2，4；12，15，30）？

10. 是否至少有三个第9类的测试用例，列举了一边大于另外两边之和的全部可能情况（如1，2，4；1，4，2；4，1，2）？

11. 是否有这样的测试用例，三边长度皆为0（0，0，0）？

12. 是否至少有一个这样的测试用例，输入的边长为非整数值（如2.5，3.5，5.5）？

13. 是否至少有一个这样的测试用例，输入的边长个数不对（如仅输入了两个而不是三个整数）？

14. 对于每一个测试用例，除了定义输入值之外，是否定义了程序针对该输入值的预期输出值？

当然，测试用例集即使满足了上述条件，也不能确保能查找出所有可能的错误。但是，由于问题1至问题13代表了该程序不同版本中已经实际出现的错误，对该程序进行的充分测试至少应该能够暴露这些错误。

开始关注自己的得分之前，请考虑以下情况：以我们的经验来看，高水平的专业程序员平均得分仅7.8（满分14）。如果读者的得分更高，那么祝贺你。如果没有那么高，我们将尽力帮助你。

这个测验说明，即使测试这样一个小的程序，也不是件容易的事。如果确实是这样，那么想像一下测试一个十万行代码的空中交通管制系统、一个编译器，甚至一个普通的工资管理程序的难度。随着面向对象编程语言（如Java、C++）的出现，测试也变得更加困难。举例来说，为测试这些语言开发出来的应用程序，测试用例必须要找出与对象实例或内存管理有关的错误。

从上面这个例子来看，完全地测试一个复杂的、实际运行的程序似乎是不太可能的。情况并非如此！尽管充分测试的难度令人望而生畏，但这是软件开发中一项非常必需的任务，也是可以实现的一部分工作，通过本书我们可以认识到这一点。

第2章 软件测试的心理学和经济学

软件测试是一项技术性工作，但同时也涉及经济学和人类心理学的一些重要因素。

在理想情况下，我们会测试程序的所有可能执行情况。然而，在大多数情况下，这几乎是不可能的。即使一个看起来非常简单的程序，其可能的输入与输出组合可达到数百种甚至数千种，对所有的可能情况都设计测试用例是不切合实际的。对一个复杂的应用程序进行完全的测试，将耗费大量的时间和人力资源，以致于在经济上是不可行的。

另外，要成功地测试一个软件应用程序，测试人员也需要有正确的态度（也许用“愿景(vision)”这个词会更好一些）。在某些情况下，测试人员的态度可能比实际的测试过程本身还要重要。因此，在深入探讨软件测试更加技术化的本质之前，我们先探讨一下软件测试的心理学和经济学问题。

2.1 软件测试的心理学

测试执行得差，其中一个主要原因在于大多数的程序员一开始就把“测试”这个术语的定义搞错了。他们可能会认为：

- “软件测试就是证明软件不存在错误的过程。”
- “软件测试的目的在于证明软件能够正确完成其预定的功能。”
- “软件测试就是建立一个‘软件做了其应该做的’信心的过程。”

这些定义都是本末倒置的。

每当测试一个程序时，总是想为程序增加一些价值。通过测试来增加程序的价值，是指测试提高了程序的可靠性或质量。提高了程序的可靠性，是指找出并最终修改了程序的错误。

因此，不要只是为了证明程序能够正确运行而去测试程序；相反，应该一开始就假设程序中隐藏着错误（这种假设对于几乎所有的程序都成立），然后测试程序，发现尽可能多的错误。

那么，对于测试，更为合适的定义应该是：

“测试是为发现错误而执行程序的过程”。

虽然这看起来像是个微妙的文字游戏，但确实有重要的区别。理解软件测试的真正定义，会对成功地进行软件测试有很大的影响。

人类行为总是倾向于具有高度目标性，确立一个正确的目标有着重要的心理学影响。如果我们的目的是证明程序中不存在错误，那就会在潜意识中倾向于实现这个目标；也就是说，我们会倾向于选择可能较少导致程序失效的测试数据。另一方面，如果我们的目标在于证明程序中存在错误，我们设计的测试数据就有可能更多地发现问题。与前一种方法相比，后一种方法

会更多地增加程序的价值。

这种对软件测试的定义，包含着无穷的内蕴，其中的很多都蕴涵在本书各处。举例来说，它暗示了软件测试是一个破坏性的过程，甚至是一个“施虐”的过程，这就说明为什么大多数人都觉得它困难。这种定义可能是违反我们愿望的；所幸的是，我们大多数人总是对生活充满建设性而不是破坏性的愿景。大多数人都本能地倾向于创造事物，而不是将事物破坏。这个定义还暗示了对于一个特定的程序，应该如何设计测试用例（测试数据）、哪些人应该而哪些人又不应该执行测试。

为增进对软件测试正确定义的理解，另一条途径是分析一下对“成功的”和“不成功的”这两个词的使用。当项目经理在归纳测试用例的结果时，尤其会用到这两个词。大多数的项目经理将没发现错误的测试用例称为一次“成功的测试”，而将发现了某个新错误的测试称为“不成功的测试”。

这又是一次本末倒置。“不成功的”表示事情不遂人意或令人失望。我们认为，如果在测试某段程序时发现了错误，而且这些错误是可以修复的，就将这次合理设计并得到有效执行的测试称作是“成功的”。如果本次测试可以最终确定再无其他可查出的错误，同样也被称作是“成功的”。所谓“不成功的”测试，仅指未能适当地对程序进行检查，在大多数情况下，未能找出错误的测试被认为是“不成功的”，这是因为认为软件中不包含错误的观点基本上是不切实际的。

能发现新错误的测试用例不太可能被认为是“不成功的”；相反，能发现错误就证明它是值得设计的。一个“不成功的”测试用例，会使程序输出正确的结果，但不能发现任何错误。

我们可以类比一下病人看医生的情况，病人因为身体不舒服而去看医生。如果医生对病人进行了一些实验检测，却没有诊断出任何病因，我们就不会认为这些实验检测是“成功的”。之所以是“不成功的”检测，是因为病人支付了昂贵的实验检测费用，而病状却依然如故。病人会因此而质疑医生的诊断能力。但是，如果实验检测诊断出病人是胃溃疡，那么这次检测就是“成功的”，医生可以开始进行适当的治疗。因此，医疗行业会使用“成功的”或“不成功的”来表达适当的意思。我们当然可以类推到软件测试中来，当我们开始测试某个程序时，它就好似我们的病人。

“软件测试就是证明软件不存在错误的过程”，这个定义会带来第二个问题。对于几乎所有的程序而言，甚至是非常小的程序，这个目标实际上也是无法达到的。

另外，心理学研究表明，当人们开始一项工作时，如果已经知道它是不可行的或无法实现时，人的表现就会相当糟糕。举例来说，如果要求人们在15分钟之内完成星期日《纽约时报》里的纵横填字游戏，那么我们会观察到10分钟之后的进展非常小，因为大多数人都会却步于这个现实，即这个任务似乎是不可能完成的。但是如果要求在四个小时之内完成填字游戏，我们很可能有理由期望在最初10分钟之内的进展会比前一种情况下的大。将软件测试定义为发现程序错误的过程，使得测试是个可以完成的任务，从而克服了这个心理障碍。

诸如“软件测试就是证明‘软件做了其应该做的’的过程”此类的定义所带来的第三个问题是，程序即使能够完成预定的功能，也仍然可能隐藏错误。也就是说，当程序没有实现预期功能时，错误是清晰地显现出来的；如果程序做了其不应该做的，这同样是一个错误。考虑一

下第1章中的三角形测试程序。即使我们证明了程序能够正确识别出不规则三角形、等腰三角形和等边三角形，但是在完成了不应执行的任务后（例如将1，2，3说成是一个不规则三角形或将0，0，0说成是一个等边三角形），程序仍然是错的。如果我们将软件测试视作发现错误的过程，而不是将其视为证明“软件做了其应该做的”的过程，我们发现后一类错误的可能性会大很多。

总结一下，软件测试更适宜被视为试图发现程序中错误（假设其存在）的破坏性的过程。一个成功的测试用例，通过诱发程序发生错误，可以在这个方向上促进软件质量的改进。当然，最终我们还是要通过软件测试来建立某种程度的信心：软件做了其应该做的，未做其不应该做的。但是通过对错误的不断研究是实现这个目的的最佳途径。

有人可能会声称“本人的程序完美无缺”（不存在错误），针对这种情况建立起信心的最好办法就是尽量反驳他，即努力发现不完美之处，而不只是确认程序在某些输入情况下能够正确地工作。

2.2 软件测试的经济学

给出了软件测试的适当定义之后，下一步就是确定软件测试是否能够发现“所有”的错误。我们将证明答案是否定的，即使是规模很小的程序。一般说来，要发现程序中的所有错误也是不切实际的，常常也是不可能的。这个基本的问题反过来暗示出软件测试的经济学问题、测试人员对被测软件的期望，以及测试用例的设计方式。

为了应对测试经济学的挑战，应该在开始测试之前建立某些策略。黑盒测试和白盒测试是两种最普遍的策略，我们将在下面两节中讨论。

2.2.1 黑盒测试

黑盒测试是一种重要的测试策略，又称为数据驱动的测试或输入/输出驱动的测试。使用这种测试方法时，将程序视为一个黑盒子。测试目标与程序的内部机制和结构完全无关，而是将重点集中放在发现程序不按其规范正确运行的环境条件。

在这种方法中，测试数据完全来源于软件规范（换句话说，不需要去了解程序的内部结构）。如果想用这种方法来发现程序的所有错误，判定的标准就是“穷举输入测试”，将所有可能的输入条件都作为测试用例。为什么这样做？比如说在三角形测试的程序中，试过了三个等边三角形的测试用例，这不能确保正确地判断出所有的等边三角形。程序中可能包含对边长为3842，3842，3842的特殊检查，并指出此三角形为不规则三角形。由于程序是个黑盒子，因此能够确定此条语句存在的惟一方法，就是试验所有的输入情况。

要穷举测试这个三角形程序，可能需要为所有有效的三角形创建测试用例，只要三角形边长在开发语言允许的最大整数值范围内。这些测试用例本身就是天文数字，但这还决不是所谓穷尽的；当程序指出－3，4，5是一个不规则三角形或2，A，2是一个等腰三角形时，问题就暴露出来了。为了确保能够发现所有这样的错误，不仅得用所有有效的输入，而且还得用所有可能的输入进行测试。因此，为了穷举测试三角形程序，实际上需要创建无限的测试用例，这当然是不可能的。

如果测试这个三角形程序都这么难的话，那么要穷举测试一个稍大些的程序的难度就更大

了。设想一下，如果要对一个C++编译器进行黑盒穷举测试，不仅要创建代表所有有效C++程序的测试用例（实际上，这又是一个无穷数），还需要创建代表所有无效C++程序的测试用例（无穷数），以确保编译器能够检测出它们是无效的。也就是说，编译器必须进行测试，确保其不会执行不应执行的操作——如顺利地编译成功一个语法上不正确的程序。

如果程序使用到数据存储，如操作系统或数据库应用程序，这个问题会变得尤为严重。举例来说，在航班预定系统这样的数据库应用程序中，诸如数据库查询、航班预约这样的事务处理需要随上一次事务的执行情况而定。因此，不仅要测试所有有效的和无效的事务处理，还要测试所有可能的事务处理顺序。

上述讨论说明，穷举输入测试是无法实现的。这有两方面的含义，一是我们无法测试一个程序以确保它是无错的，二是软件测试中需要考虑的一个基本问题是软件测试的经济学。也就是说，由于穷举测试是不可能的，测试投入的目标在于通过有限的测试用例，最大限度地提高发现的问题的数量，以取得最好的测试效果。除了其他因素之外，要实现这个目标，还需要能够窥见软件的内部，对程序作些合理但非无懈可击的假设（例如，如果三角形程序将2，2，2视为等边三角形，那就有理由认为程序对3，3，3也作同样判断）。这种思路将形成本书第4章中测试用例设计策略的部分方法。

2.2.2 白盒测试

另一种测试策略称为白盒测试或称逻辑驱动的测试，允许我们检查程序的内部结构。这种测试策略对程序的逻辑结构进行检查，从中获取测试数据（遗憾的是，常常忽略了程序的规范）。

在这里我们的目标是针对这种测试策略，建立起与黑盒测试中穷举输入测试相似的测试方法。也许有一个解决的办法，即将程序中的每条语句至少执行一次。但是我们不难证明，这还是远远不够的。这种方法通常称为穷举路径测试，在本书第4章中将进一步进行深入探讨，在这里就不多加叙述。所谓穷举路径测试，即如果使用测试用例执行了程序中所有可能的控制流路径，那么程序有可能得到了完全测试。

然而，这个论断存在两个问题。首先，程序中不同逻辑路径的数量可能达到天文数字。图2-1所示的小程序显示了这一点。该图是一个控制流图，每一个结点或圆圈都代表一个按顺序执行的语句段，通常以一个分支语句结束。每一条边或弧线表示语句段之间的控制（分支）的转换。图2-1描述的是一个有着10～20行语句的程序，包含一个迭代20次的DO循环。在DO循环体中，包含一系列嵌套的IF语句。要确定不同逻辑路径的数量，也相当于要确定从点a～点b之间所有不同路径的数量（假定程序中所有的判断语句都是相互独立的）。这个数量大约是10^{14}，即100万亿，是从$5^{20}+5^{19}\cdots+5^{1}$计算而来，5是循环体内的路径数量。由于大多数的人难以对这个数字有一个直观的概念，不妨设想一下：如果在每五分钟内可以编写、执行和确认一个测试用例，那么需要大约10亿年才能测试完所有的路径。假如可以快上300倍，每一秒就完成一次测试，也得用漫长的320万年才能完成这项工作。

当然，在实际程序中，判断并非都是彼此独立的，这意味着可能实际执行的路径数量要稍微少一些。但是，从另一方面来讲，实际应用的程序要比图2-1所描述的简单程序复杂得多。因此，穷举路径测试就如同穷举输入测试，非但不可能，也是不切实际的。

“穷举路径测试即完全的测试”论断存在的第二个问题是，虽然我们可以测试到程序中的所有路径，但是程序可能仍然存在着错误。这有三个原因。

第一，即使是穷举路径测试也决不能保证程序符合其设计规范。举例来说，如果要编写一个升序排序程序，但却错误地编成了一个降序排序程序，那么穷举路径测试就没多大价值了；程序仍然存在着一个缺陷：它是个错误的程序，因为不符合设计的规范。

第二，程序可能会因为缺少某些路径而存在问题。穷举路径测试当然不能发现缺少了哪些必需路径。

第三，穷举路径测试可能不会暴露数据敏感错误。这样的例子有很多，举一个简单的例子就能说明问题。假设在某个程序中要比较两个数值是否收敛，也就是检查两个数值之间的差异是否小于某个既定的值。比如，我们可能会这样编一条Java语言的IF语句：

```
if (a-b < c)
   System.out.println("a-b < c");
```

当然，这条语句包含一个错误，因为它可能将 c 与a-b的绝对值进行比较。然而，要找出这样的错误，取决于a和b所取的值，而仅仅执行程序中的每条路径并不一定能找出错误来。

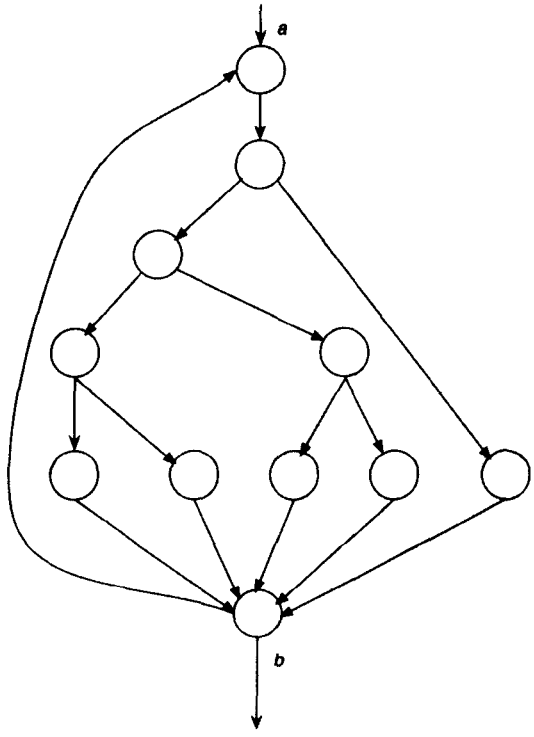


图2-1 一个小型程序的控制流图

总之，尽管穷举输入测试要强于穷举路径测试，但两者都不是有效的方法，因为这两种方法都不可行。那么，也许存在别的方法，将黑盒测试和白盒测试的要素结合起来，形成一个合理但并不十分完美的测试策略。本书的第4章将深入讨论这个话题。

2.3 软件测试的原则

让我们继续本章的话题基础，即软件测试中大多数重要的问题都是心理学问题。我们可以归纳出一系列重要的测试指导原则。这些原则看上去大多都是显而易见的，但常常总是被我们忽视掉。表2-1总结了这些重要原则，每条原则都将在下面的章节中详细介绍。

表2-1 软件测试的重要原则

编 号	原 则
1	测试用例中一个必需部分是对预期输出或结果进行定义
2	程序员应当避免测试自己编写的程序