



软 件 工 程 技 术 丛 书



# 软件测试的艺术

(原书第3版)

*The Art of Software Testing* (Third Edition)

(美) Glenford J. Myers Tom Badgett Corey Sandler 著 张晓明 黄琳 译



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3版)

The Art of Software Testing, Third Edition

Glenford J. Myers  
(美) Tom Badgett 著  
Corey Sandler  
张晓明 黄琳 译

HZ BOOKS  
华章图书

本书从第1版付梓到现在已经30余年，是软件测试领域的经典著作。本书结构清晰、讲解生动活泼，简明扼要地展示了久经考验的软件测试方法和智慧。

本书以一次自我评价测试开篇，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、互联网应用测试等高级主题，全面展现了作者的软件测试思想。第3版在前两版的基础上，结合软件测试的最新发展进行了更新，覆盖了可用性测试、移动应用测试以及敏捷开发测试等内容。

本书适合软件开发人员、IT项目经理等相关读者阅读，还可以作为高等院校计算机相关专业软件测试课程的教材或参考书。

Glenford J. Myers, Tom Badgett, Corey Sandler: The Art of Software Testing, Third Edition (ISBN: 978-1-118-03196-4).

Authorized translation from the English language edition published by John Wiley & Sons, Inc.

Copyright © 2012 by Word Association, Inc. Published by John Wiley & Sons, Inc.

All rights reserved.

本书中文简体字版由约翰·威利父子公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

封底无防伪标签均为盗版

版权所有，侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2011-7872

图书在版编目（CIP）数据

软件测试的艺术（原书第3版）/（美）梅耶（Myers, G. J.）等著；张晓明，黄琳译．  
—北京：机械工业出版社，2012.3

（软件工程技术丛书）

书名原文：The Art of Software Testing, Third Edition

ISBN 978-7-111-37660-6

. 软... . 梅... 张... 黄... . 软件测试 . TP311.5

中国版本图书馆CIP数据核字（2012）第040000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：吴 怡

印刷

2012年4月第1版第1次印刷

170mm × 242mm · 12.75印张

标准书号：ISBN 978-7-111-37660-6

定价：39.00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

《软件测试的艺术》(下简称《艺术》)作为元老级的测试书在国内可能没那么出名,但它的确非常经典且很有口碑,书中所提出的“软件测试为求错而非求证”的观点至今仍在学术界被广泛争议与讨论。随着软件测试的重要性越来越受到现代软件企业的重视,这本书就好像尘封已久的宝藏被人们挖掘出来并受到追捧。与此同时,也正是因为测试市场的需求激增,书店里的测试书籍也似乎是“忽如一夜春风来,千树万树梨花开”了。

我和《艺术》的第一次接触并不是在书店里发现了它,而是因为我阅读的一个习惯。我喜欢在阅读之前先看参考文献部分,也由此发现国内的很多软件测试书籍都把《艺术》作为首要的参考书目,这让我不得不对该书刮目相看,现在我终于明白了,原来《艺术》一书就是现在各种测试书籍参考的源头之一。以我个人的观点,今天书店里的软件测试理论书籍(注意我是指理论方面)已经饱和甚至是富营养化,如果你打算系统地学习软件测试理论知识,我不敢向你保证这本书是最全面最详细的,但是绝对是恰到好处的,它精悍凝练的篇幅可以让你在最短时间内获得关于软件测试的真知灼见。

对于那些已经成为测试工程师甚至是高级测试工程师的人来说,本书同样值得一读,书中的很多内容读起来仿佛醍醐灌顶,本书所涵盖的测试知识经过千锤百炼和时间的考验,而把这些理论知识结合你的测试经验,能系统化并巩固加深你对测试这门学科的理解,而这种对软件测试技术系统的、深刻的理解,将使你在今后的工作以及事业中受益匪浅。

因此,作为一名测试工程师,我对读者的建议是,初学者可将本书作为入门书;而有经验者更应该将本书作为理论指南,花点时间翻阅一下,梳理自己的经验和知识;本书对开发人员也相当有用,可以让你在最短的时间内建立起对测试的框架认知,从而在编码的过程中能够在脑海里多一些测试的思想,十分有益,当然有些测试类型本身就需要开发者参与,比如本书所介绍的极限编程与测试,开发者需

要编写单元测试用例。对于测试管理者而言，本书的重要性不言而喻，本书内容非常精炼，将有助于你根据项目情况制定更合理、更有效的测试计划。

作为本书第3版的译者，我十分有幸能够接到这样的经典书籍翻译任务，而且还是本人的处女译，心中的忐忑自不必多说，只希望我的翻译能够不辱原著的经典，更能够得到读者的认可。

此次第3版相对于之前的第2版增加了全新的两章（第7章和第11章），由此使得本书包含了当下最新的测试分类和技术。除了第9章的改动增补较大外，其他的章节基本是略有改动。

在翻译过程中，我尽量保证在准确翻译原文的基础上增加了一些个人的见解，通常以译者注的形式给出，我相信这些注释有助于国内的读者理解和消化书中内容；也有一些地方做了勘误。有时候直译的效果并不那么好，我也会尝试一些意译，比如在11.4节，作者强调了移动应用必然越来越火这一现象和趋势，我便借用著名诗歌《见与不见》来意译：

你用，或者不用  
移动应用就在那里

.....

因为是站在巨人的肩膀上（前两版译文）进行翻译，所以我必须感谢前两版的翻译人员，这使得我此次翻译的精力主要聚焦在新增的两章以及改动较大的章节上。同时要感谢黄琳为我的原始译稿所做的详尽审稿和勘误工作，这也是我们之间的二度合作。

最后请大家注意书名中的“艺术”二字，这暗示本书并不是那么晦涩难懂或写得很深奥，我觉得即使你没多少计算机基础知识，只要用过电脑软件，本书的很多章节读起来应该都不会太吃力。我希望有越来越多各行各业的人们加入测试的世界，从本书第7章的可用性测试我们看到，测试需要领域内经验，如果某公司针对音乐爱好者推出了一款混音软件，他们肯定会想，要是能够把周杰伦请来做首席用户体验师该有多好。总之，测试，这是个相比较开发来讲门槛不算太高的职业（当然要做到精深未必容易，甚至难度还要高），而且收入还算不错，在这里与各位读者共勉了。

张晓明

2012年2月6日

# 序 言

1979年，Glenford J. Myers出版了一本现在仍被证明为经典的著作，这就是本书第1版。本书经受住了时间的考验，25年来一直列在出版商提供的书目清单中。这个事实本身就是对本书可靠、精粹和珍贵品质的佐证。

在同一时期，本书第3版的几位合著者共出版了200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8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）。然而，那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？那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变动和更新。

在此期间出版的有关软件测试的书籍已数以百计，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3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、极限编程与测试及移动应用测试。

但是，我们永远不会忘记，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。

# 前言

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。

30多年后的今天，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，本书再版的目的是与前两版一样：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，还包括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。

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。

读者可能会意识到，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。第5章探讨了如何测试单个模块或子例程，第6章讲述了如何测试更大的对象。第7章围绕用户体验或可用性测试这一重要的软件测试概念进行阐述，在更复杂且拥有更广大用户量的软件不断涌现的今天，可用性测试变得越来越重要。第8章介绍了一些程序调试的实用建议，第9章着重研究了极限编程及其测试（一种在今天称为敏捷开发环境中的编程和测试方法）。第10章介绍如何将本书所涵盖的软件测试知识运用到Web开发中，包括电子商务系统以及社交网络<sup>⊖</sup>的开发。第11章描述了如何测试移动设备上的应用。

本书主要面向三类读者。第一类是专业的程序员。尽管我们希望本书的内容对于他们来说不是全新的知识，但本书能使专业程序员对测试技术增强了解。如果这些材料能使软件开发人员在某个程序中多发现了一个错误，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，他们将会直接受益于本书所介绍的实用的测试管理理论与知识。第三类读者是软件或计算机专业的学生，我们的目的在于向学生展示程序测试的问题，并提供一系列有效的技术。对于最后一类读者群，我们建议本书作为程序设计课程的补充教材，使学生在学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

---

⊖ 具有高度的和用户交互性质的Web应用，也即是所谓的Web 2.0。——译者注



# 目 录

译者序

序言

前言

第1章 一次自评价测试 .....	1
第2章 软件测试的心理学和经济学 .....	4
2.1 软件测试的心理学 .....	4
2.2 软件测试的经济学 .....	7
2.2.1 黑盒测试 .....	7
2.2.2 白盒测试 .....	8
2.3 软件测试的原则 .....	10
2.4 小结 .....	14
第3章 代码检查、走查与评审 .....	15
3.1 代码检查与走查 .....	16
3.2 代码检查 .....	17
3.2.1 代码检查小组 .....	17
3.2.2 检查议程与注意事项 .....	18
3.2.3 对事不对人，和人有关的注意事项 .....	19
3.2.4 代码检查的衍生功效 .....	19
3.3 用于代码检查的错误列表 .....	19
3.3.1 数据引用错误 .....	20
3.3.2 数据声明错误 .....	22
3.3.3 运算错误 .....	23
3.3.4 比较错误 .....	23
3.3.5 控制流程错误 .....	24
3.3.6 接口错误 .....	26
3.3.7 输入/输出错误 .....	27

3.3.8 其他检查 .....	27
3.4 代码走查 .....	29
3.5 桌面检查 .....	30
3.6 同行评审 .....	31
3.7 小结 .....	32
第4章 测试用例的设计 .....	33
4.1 白盒测试 .....	34
4.2 黑盒测试 .....	40
4.2.1 等价划分 .....	40
4.2.2 一个范例 .....	43
4.2.3 边界值分析 .....	45
4.2.4 因果图 .....	50
4.3 错误猜测 .....	66
4.4 测试策略 .....	67
4.5 小结 .....	68
第5章 模块（单元）测试 .....	70
5.1 测试用例设计 .....	70
5.2 增量测试 .....	81
5.3 自顶向下测试与自底向上测试 .....	84
5.3.1 自顶向下的测试 .....	84
5.3.2 自底向上的测试 .....	89
5.3.3 比较 .....	90
5.4 执行测试 .....	91
5.5 小结 .....	92
第6章 更高级别的测试 .....	93
6.1 功能测试 .....	96
6.2 系统测试 .....	97
6.2.1 能力测试 .....	99
6.2.2 容量测试 .....	100
6.2.3 强度测试 .....	100
6.2.4 可用性测试 .....	101
6.2.5 安全性测试 .....	101
6.2.6 性能测试 .....	102

6.2.7 存储测试.....	102
6.2.8 配置测试.....	102
6.2.9 兼容性/转换测试.....	103
6.2.10 安装测试 .....	103
6.2.11 可靠性测试 .....	103
6.2.12 可恢复性测试 .....	104
6.2.13 服务/可维护性测试 .....	105
6.2.14 文档测试 .....	105
6.2.15 过程测试 .....	105
6.2.16 系统测试的执行 .....	105
6.3 验收测试 .....	106
6.4 安装测试 .....	107
6.5 测试的计划与控制 .....	107
6.6 测试结束准则 .....	109
6.7 独立的测试机构 .....	114
6.8 小结.....	114
第7章 可用性（或用户体验）测试 .....	116
7.1 可用性测试基本要素 .....	116
7.2 可用性测试流程 .....	118
7.2.1 测试用户的选择 .....	119
7.2.2 需要多少用户进行测试 .....	120
7.2.3 数据采集方法.....	122
7.2.4 可用性调查问卷 .....	124
7.2.5 何时收工，还是多多益善 .....	125
7.3 小结.....	126
第8章 调试 .....	127
8.1 暴力法调试 .....	128
8.2 归纳法调试 .....	129
8.3 演绎法调试 .....	132
8.4 回溯法调试 .....	135
8.5 测试法调试 .....	135
8.6 调试的原则 .....	136
8.6.1 定位错误的原则 .....	136

8.6.2 修改错误的技术 .....	138
8.7 错误分析 .....	139
8.8 小结 .....	140
第9章 敏捷开发模式下的测试 .....	142
9.1 敏捷开发的特征 .....	143
9.2 敏捷测试 .....	144
9.3 极限编程与测试 .....	145
9.3.1 极限编程基础 .....	146
9.3.2 极限测试：概念 .....	149
9.3.3 极限测试的应用 .....	152
9.4 小结 .....	155
第10章 互联网应用测试 .....	156
10.1 电子商务的基本结构 .....	157
10.2 测试的挑战 .....	159
10.3 测试的策略 .....	161
10.3.1 表示层的测试 .....	163
10.3.2 业务层的测试 .....	165
10.3.3 数据层的测试 .....	167
10.4 小结 .....	169
第11章 移动应用测试 .....	171
11.1 移动环境 .....	171
11.2 测试面临的挑战 .....	173
11.2.1 移动设备多样性 .....	173
11.2.2 运营商网络基础设施 .....	174
11.2.3 脚本编程 .....	176
11.2.4 可用性测试 .....	177
11.3 测试方法 .....	177
11.3.1 真机测试 .....	179
11.3.2 基于模拟器的测试 .....	181
11.4 小结 .....	182
附录A 极限编程示例程序 .....	184
附录B 小于1000的素数 .....	190

# 一次自我评价测试

自本书30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？原因在于大量的编程语言、操作系统以及硬件平台的涌现。在20世纪70年代只有相当少的人使用计算机，而在今天几乎人人离不开计算机。而今天计算机不仅仅是指摆在你书桌上的计算机了，几乎所有我们所接触和使用的电子设备都内置了一个“计算机”或者计算芯片，以及运行在其上的软件系统。不妨回想一下在今天的社会中还在使用哪些不需要软件驱动的设备，没错，锤子和手推车是，但是这些工具也大量使用在由软件控制和操作的车间中。软件的普遍应用提升了测试的意义。今天的设备已经千百倍强于它们的“前辈”，今天的“计算机”这个概念也变得越来越广泛和越来越难准确地定义。数字电视、电话、游戏产品、汽车等都有一颗计算机的“心”以及运行其中的软件，以至于在某些情况下它们自己本身也能够被看做是一台特别的计算机。

因此，现在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1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预编程对象，将其作为自定义应用程序的组成部分进行测试的要求就减少了许多。

另外，尽管市场上的测试书籍越来越多，甚至有过剩之嫌，似乎依旧有很多开发人员对全面的测试并不那么欢迎。引入更优秀的开发工具、使用已经通过测试



的GUI（图形界面控件）控件、紧张的交付日期以及高度集成的便利开发环境会让测试变得仅仅是让那些最基本的测试用例走走过场罢了。影响不大的bug也许只不过会让最终用户觉得使用不方便而已，然而严重的bug则可能造成经济损失甚至是人身伤害。本书所阐述的方法旨在帮助设计人员、开发工程师以及项目经理更好地理解全面综合测试的意义所在，并提供行之有效的指南以帮助达成测试的目标。

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。下面是对该程序的描述：

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条边的长度。程序显示提示信息，指出该三角形是何种三角形：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。因此，想象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。

## 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，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，这样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的本质之前（指技术层面），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，应当想到要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做是“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，同样也被称做是“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”，也就是说，能发现错误就证明它是值得设计的。“不成功的”测试用例，会看到程序输出正确的结果

而没发现任何错误。

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检查和化验，却没有诊断出任何病因，我们就不会认为这些检查和化验是“成功的”，因为病人支付了昂贵的检查和化验费用，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，如果医生诊断出病人是胃溃疡，那么这次检测就是“成功的”，医生可以开始进行相应的治疗。因此，医疗行业会使用“成功的”或“不成功的”来表达诊断结果。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,但这还绝不是所谓穷尽的;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循环体内，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 $5^{20}+5^{19}+\dots+5^1$ 计算而来，5是循环体内的路径数量。

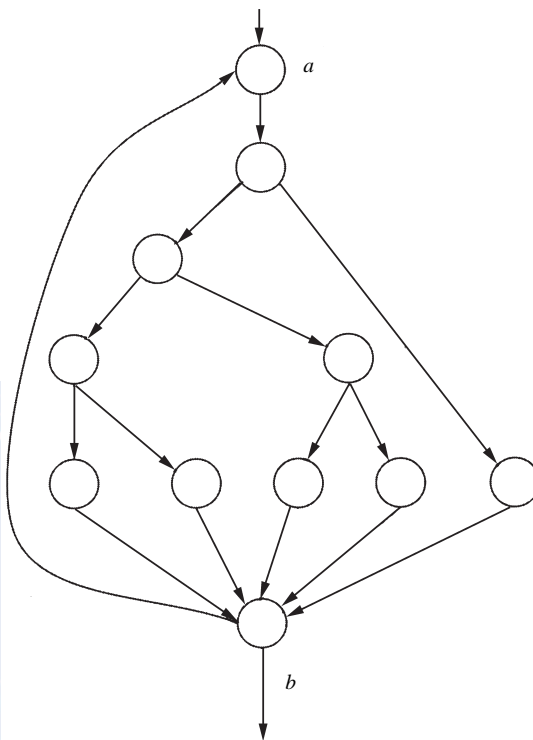


图2-1 一个小型程序的控制流图

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。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，这条语句明显错了，因为程序原意是将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。

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
3	编写软件的组织不应当测试自己编写的软件
4	应当彻底检查每个测试的执行结果

(续)

编 号	原 则
5	测试用例的编写不仅应当根据有效和预料到的输入情况，而且也应当根据无效和未预料到的输入情况
6	检查程序是否“未做其应该做的”仅是测试的一半，测试的另一半是检查程序是否“做了其不应该做的”
7	应避免测试用例用后即弃，除非软件本身就是一个一次性的软件
8	计划测试工作时不应默许假定不会发现错误
9	程序某部分存在更多错误的可能性，与该部分已发现错误的数量成正比
10	软件测试是一项极富创造性、极具智力挑战性的工作

原则1：测试用例中一个必需部分是对预期输出或结果的定义。

这条显而易见的原则在软件测试中是最常犯的错误之一。同样，这个问题也是基于人们的心理的。如果某个测试用例的预期结果事先没有得到定义，由于“所见即所想”现象的存在，某个似是而非、实际上是错误的结果可能会被解释成正确的结论。换句话说，尽管“软件测试是破坏性”的定义是合理的，但人们在潜意识中仍然渴望看到正确的结果。克服这种倾向的一种方法，就是通过事先精确定义程序的预期输出，鼓励人们对所有的输出进行仔细检查。因此，一个测试用例必须包括两个部分：

1. 对程序的输入数据的描述。
2. 对程序在上述输入数据下的正确输出结果的精确描述。

所谓“问题”，可以归纳为一个或一组我们不能给出可信服的解释、看上去不太正常或不符合我们期望或预想的事实。应当明确的是，在确定事物存在“问题”之前，人们必须已经形成了特定的认识。没有期望，也就没有所谓的意外。

原则2：程序员应当避免测试自己编写的程序。

任何作者都知道或应该知道，亲自编辑或校对自己的作品确实是个不好的做法。作者清楚某段文字要说明的是什么，实际表达出来的意思却南辕北辙，而自己可能却意识不到。况且实际上也不会想在自己的作品中找出什么错误来。对程序员而言，也存在相同的问题。

如果我们对软件项目关注的重点发生变化，就会产生另外一个问题。当程序员“建设性”地设计和编写完程序之后，很难让他突然改变视角以一种“破坏性”的眼光来审查程序。



正如许多房屋业主都知道的那样，撕下屋里的墙纸（这是个破坏性的过程）并不容易，如果这些墙纸又恰恰是业主第一个亲手贴的，尤其令其沮丧不已。同样，大多数程序员都不能有效地测试自己编写的程序，因为他们无法改变思维方式来尽力暴露自己程序中的错误。另外，程序员可能会下意识地避免找出错误来，担心受到同事、上司、客户或正在开发的程序或系统的主管的惩罚。

仅次于上面的心理学问题，还有一个重要的问题：由于程序员错误地理解了疑难定义或规范，导致程序中存在错误。如果情况是这样，程序员可能会带着同样的误解来测试自己的程序。

这并不意味着程序员测试自己的程序是不可能的。当然，我们的言下之意是，让其他人来测试程序会更加有效，也会更容易测试成功。

请注意，我们的论据并不适合于“调试”（纠正已知的错误）。“调试”由程序的编写人员来完成会有效得多。

原则3：编写软件的组织不应当测试自己编写的软件。

这里的论据与前面的论据相似。从很多方面来讲，一个软件项目或编程组织是一个有机的机构，具有与个体程序员相似的心理问题。而且在大多数情况下，主要是根据其在给定时间、特定成本范围内开发软件的能力来衡量编程组织或项目经理。其中的一个原因是，度量时间和成本目标比较容易，而定量地衡量软件的可靠性则极其困难。即便是合理规划和实施的测试过程，也可能被认为降低了完成进度和成本目标的可能性，因此，编程组织难以客观地测试自己的软件。

同样，我们并不是说编程组织发现程序中的问题是不可能的，事实上很多组织已经在某种程度上成功地做到了这一点。当然，我们的言下之意是，更经济的方法是由客观、独立的第三方来进行测试。

原则4：应当彻底检查每个测试的执行结果。

这个原则可能是最显而易见的原则，但也同样常常被忽视。我们见过大量的例子，即便错误的症状在输出清单中可以清楚地看到，但还是没有找出那些错误来。换言之，在后续测试中发现的错误，往往是前面的测试遗漏掉的。

原则5：测试用例的编写不仅应当根据有效和预期的输入情况，而且也应当根据无效和未预料到的输入情况。

在测试软件时，有一个自然的倾向，即将重点集中在有效和预期的输入情况上，而忽略了无效和未预料到的情况。比如，在本书第1章三角形程序的测试中，

总是出现这个倾向。

例如，很少有人会向程序输入1, 2, 5以证明程序不会错误地将其解释为一个不规则三角形，而不是一个无效三角形。此外，在软件产品中突然暴露出来的许多问题是当程序以某些新的或未预料到的方式运行时发现的。因此，针对未预料到的和无效输入情况的测试用例，似乎比针对有效输入情况的那些用例更能发现问题。

原则6：检查程序是否“未做其应该做的”仅是测试的一半，测试的另一半是检查程序是否“做了其不应该做的”。

这条原则是上条原则的必然结果。必须检查程序是否有我们不希望的副作用。比如，某个工资管理程序即便可以生成正确的工资单，但是如果也为非雇员生成工资单或者它覆盖掉了人员文件的第一条记录，这样的程序仍然是不正确的程序。

原则7：应避免测试用例用后即弃，除非软件本身就是一个一次性的软件。

这个问题在采用交互式系统来测试软件时最常见。人们通常会坐在终端前，匆忙地编写测试用例，然后将这些用例交由程序执行。这样做的问题在于，饱含我们宝贵投入的测试用例，在测试结束后就消失了。一旦软件需要重新测试（例如，当改正了某个错误或作了某种改进后），又必须重新设计这些测试用例。情况往往是这样的，由于重新设计测试用例需要投入大量的工作，人们总是避免这样做。因此，对该程序的重新测试极少会同上次一样严格。这就意味着，如果对程序的更改导致了程序某个先前可以执行的部分发生了故障，这个故障往往是不会被发现的。保留测试用例，当程序其他部件发生变动后重新执行，这就是我们所谓的“回归测试”。

原则8：计划测试工作时不应默许假定不会发现错误。

项目经理经常容易犯这个错误，这也是使用了不正确的测试定义的一个迹象——也就是说，假定“测试是一个证明程序正确运行的过程”。我们再一次重申，所谓测试，就是为发现错误而执行程序的过程。

原则9：程序某部分存在更多错误的可能性，与该部分已发现错误的数量成正比。

这种现象如图2-2所示。乍看上去，这幅图似乎没有什么意义，但很多程序都存在这种现象。例如，假如某个程序由两个模块、类或子程序A和B组成，模块A中已经发现了五个错误，而模块B中仅仅找到了一处错误。如果模块A所经过的测试并不是故意设计得更为严格，那么该原则告诉我们，模块A与模块B相比，存在更多错误的可能性要大。

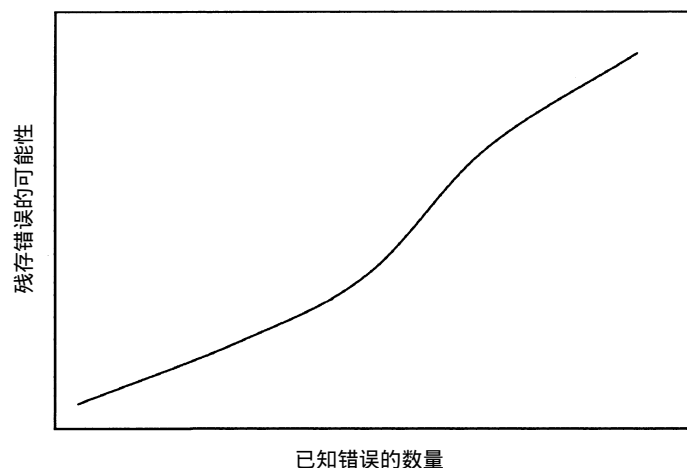


图2-2 残存错误与已知错误间令人惊奇的联系

该原则的另一个说法是，错误总是倾向于聚集存在，而在一个具体的程序中，某些部分要比其他部分更容易存在错误，尽管没有人能够对这种现象给出很好的解释。这种现象之所以有用，是因为它给予了我们软件测试过程的深入理解或反馈信息。如果一个程序的某个部分远比其他部分更容易产生错误，那么这种现象告诉我们，为了使测试获得更大的成效，最好对这些容易存在错误的部分进行额外的测试。

原则10：软件测试是一项极富创造性、极具智力挑战性的工作。

测试一个大型软件所需要的创造性很可能超过了开发该软件所需要的创造性。我们已经看到，要充分地测试一个软件以确保所有错误都不存在是不可能的。本书后续章节讨论的技术使我们能够为某个软件设计出合理的测试用例集，然而这些技术仍然需要大量的创造性。

## 2.4 小结

在阅读本书接下来的内容时，请牢记以下几个重要的测试原则：

- 软件测试是为发现错误而执行程序的过程。
- 尽量避免编码人员测试自己的程序。
- 好的测试用例能够对未发现的错误高度敏感。
- 成功的测试用例能够发现未知的错误。
- 成功的测试需要仔细定义输入输出的期望值。
- 成功的测试需要仔细研究分析测试结果。

## 代码检查、走查与评审

多年以来，软件界的大多数人都持有一个想法，即编写程序仅仅是为了提供给机器执行，并不是供人们阅读的，软件测试的惟一方法就是在计算机上执行它。20世纪70年代早期，一些程序员最先意识到阅读代码对于构成完善的软件测试和调试手段的价值，通过他们的努力，原有的观念开始发生变化。

今天，并不是所有的软件测试人员都要阅读代码，但是研读程序代码作为测试工作的一部分，这个观念已经得到了广泛认同。以下几个因素会影响到特定的测试和调试工作需要人工实际阅读代码的可能性：软件的规模和复杂度、软件开发团队的规模、软件开发的时限（例如时间安排表是松散还是紧密）等，当然还有编程小组的技术背景和文化。

基于这些原因，在深入研究较为传统的基于计算机的测试技术之前，我们首先讨论非基于计算机测试的过程（即“人工测试”）。人工测试技术在查找错误方面非常有效，以至于任何编程项目都应该使用其中的一种或多种技术。应该在程序开始编码之后、基于计算机的测试开始之前使用这些方法。同样，也可以在编程过程的更早阶段就开始设计和应用类似的方法（例如在每个设计阶段的末尾），但是这些内容超出了本书讨论的范围。

在开始讨论人工测试技术之前，有一条重要的注意事项：由于包含了人为因素在内，导致很多方法的正规性要差于由计算机执行的数学证明，人们可能会怀疑某些如此简单和不正规的东西是否有用。反之亦然。这些不正规的方法并没有妨碍测试取得成功；相反，它们从以下两个方面显著地提高了测试的功效和可靠性。

首先，人们普遍认识到错误发现得越早，改正错误的成本越低，正确改正错误的可能性也越大。其次，程序员在开始基于计算机的测试时似乎要经历一个心

理上的转变。从内部产生的压力似乎会急剧增长，并产生一个趋势，要“尽可能地修正这个缺陷”。由于这些压力的存在，程序员在改正某个由基于计算机测试发现的错误时所犯的失误，要比改正早期发现的问题时所犯的失误更多一些。

### 3.1 代码检查与走查

代码检查、走查以及可用性测试是三种主要的人工测试方法。这些测试方法可以应用在软件开发的任何阶段，包括在一个应用程序编码基本结束或者每一个模块（单元）编码结束之后（阅读第5章关于模块或单元测试的更多内容）。本章将主要介绍前两种针对代码的（白盒级别的）测试方法。在第7章我们会讨论可用性测试。

由于代码走查和检查这两种方法具有很多共同之处，所以在这里我们将讨论它们的相似点，而它们的不同之处将在后续章节中介绍。

代码检查与走查都要求人们组成一个小组来阅读或直观检查特定的程序。无论采用哪种方法，参加者都需要完成一些准备工作。准备工作的高潮是在参加者会议上进行的所谓“头脑风暴会”。“头脑风暴会”的目标是找出错误来，但不必找出改正错误的方法。换句话说，是测试，而不是调试。

代码检查与走查已经广泛运用了很长时间。我们认为，它们的成功与本书第2章所述的一些原则有关。

在代码走查中，一组开发人员（三到四人为最佳）对代码进行审核。其中只有一人是代码的作者。因此，代码走查的主要工作是由其他人，而不是作者本人完成的，这和软件测试的第2原则，也即“软件编写者往往不能有效地测试自己的软件”相符合。（参见第2章，表2.1，本书将陆续涉及表中归纳的10条测试原则。）

代码检查与走查是对过去桌面检查过程（在提交测试前由程序员阅读自己程序的过程）的改进。与原方法相比，代码检查与走查更为有效，同样是因为在实施过程中，除了软件编写者本人，还有其他人参与进来。

代码走查的另一个优点在于，一旦发现错误，通常就能在代码中对其进行精确定位，这就降低了调试（错误修正）的成本。另外，这个过程通常发现成批的错误，这样错误就可以一同得到修正。而基于计算机的测试通常只能暴露出错误的某个表症（程序不能停止，或打印出了一个无意义的结果），错误通常是逐个地

被发现并得到纠正的。

在典型的程序中，这些方法通常会有效地查找出30% ~ 70%的逻辑设计和编码错误。但是，这些方法不能有效地查找出高层次的设计错误，例如在软件需求分析阶段的错误。请注意，所谓30% ~ 70%的错误发现率，并不是说所有错误中多达70%可能会被找出来，而是讲这些方法在测试过程结束时可以有效地查找出多达70%的已知错误。请记住，第2章告诉我们，程序中的错误总数始终是未知的。

当然，可能存在对这些统计数字的批评，即人工方法只能发现“简单”的错误（即与基于计算机的测试方法相比，所发现的问题显得微不足道），而困难的、不明显的或微妙的错误只能用基于计算机的测试方法才能找到。然而，一些测试人员在使用了人工方法之后发现，对于某些特定类型的错误，人工方法比基于计算机的方法更有效，而对于其他错误类型，基于计算机的方法更有效。这就意味着，代码检查/走查与基于计算机的测试是互补的。缺少其中任何一种，错误检查的效率都会降低。

最后，不但这些测试过程对于测试新开发的程序有着不可估量的作用，而且对于测试更改后的程序，这些测试过程具有相同的作用，甚至更大。根据我们的经验，修改一个现存的程序比编写一个新程序更容易产生错误（以每写一行代码的错误数量计）。因此，除了回归测试方法之外，更改后的程序还要进行这些人工方法的测试。

## 3.2 代码检查

所谓代码检查，是以组为单位阅读代码，它是一系列规程和错误检查技术的集合。对代码检查的大多数讨论都集中在规程、所要填写的表格等。这里对整个规程进行简短的概述，之后我们将重点讨论实际的错误检查技术。

### 3.2.1 代码检查小组

一个代码检查小组通常由四人组成，其中一人发挥着协调作用。协调人应该是个称职的程序员，但不是该程序的编码人员，不需要对程序的细节了解得很清楚。协调人的职责包括以下几点：

- 为代码检查分发材料、安排进程。
- 在代码检查中起主导作用。



- 记录发现的所有错误。
- 确保所有错误随后得到改正。

第二个小组成员是代码的作者。小组中的其他成员通常是程序的设计人员（如果设计人员不同于编码人员的话），以及一名测试专家。这名测试专家应该具备较高的软件测试造诣并熟悉大部分的常见编码错误，下文会就这些常见编码错误进行讨论。

### 3.2.2 检查议程与注意事项

在代码检查之前的几天，协调人将程序清单和设计规范分发给其他成员。所有成员应在检查之前熟悉这些材料。在检查进行时，主要进行两项活动：

1. 由程序编码人员逐条语句讲述程序的逻辑结构。在讲述的过程当中，小组的其他成员应提问题、判断是否存在错误。在讲述中，很可能是程序编码人员本人而不是其他小组成员发现了大部分错误。换句话说，对着大家大声朗读程序，这种简单的做法看来是一个非常有效的错误检查方法。
2. 参考常见的编码错误列表分析程序（错误列表将在下一节中介绍）。

协调人负责确保检查会议的讨论高效地进行、每个参与者都将注意力集中于查找错误而不是修正错误（错误的修正由程序员在检查会议之后完成）。

会议结束之后，程序员会得到一份已发现错误的清单。如果发现的错误太多，或者某个错误涉及对程序做根本性的改动，协调人可能会在错误修正后安排对程序进行再次检查。这份错误清单也要进行分析、归纳，用以提炼错误列表，以便提高以后代码检查的效率。

如上所述，这个代码检查过程通常将注意力集中在发现错误上，而不是纠正错误。然而，有些小组可能会发现，当检查出某个小问题之后，有两三个人（包括负责该代码的程序员本人）会建议对设计进行明显的修补以解决这个特例。那么，对这个小问题的讨论，反过来会将整个小组的注意力集中在设计的某个部分。在探讨修补设计来解决这个小问题的最佳方法时，有人可能会注意到另外的问题。既然小组已经发现了设计中同一部分的两个相关问题，那么每隔几段代码就可能需要密集的注释。几分钟之内，整个设计就被彻底检查完，任何问题都会一目了然。

在代码检查的时间及地点的选择上，应避免所有的外部干扰。代码检查会议的理想时间应在90～120分钟。由于开会是一项繁重的脑力劳动，会议时间越长效

率越低。大多数的代码检查都是按每小时大约阅读150行代码的速度进行。因此，对大型软件的检查应安排多个代码检查会议同时进行，每个代码检查会议处理一个或几个模块或子程序。

### 3.2.3 对事不对人，和人有关的注意事项

请注意，要使检查过程有成效，必须树立正确的态度。如果程序员将代码检查视为对其人格的攻击、采取了防范的态度，那么检查过程就不会有效果。正确的做法是，程序员必须怀着非自我本位的态度来对待检查过程，对整个过程采取积极和建设性的态度：代码检查的目标是发现程序中的错误，从而改进软件的质量。正因为这个原因，大多数人建议应对代码检查的结果进行保密，仅限于参与者范围内部。尤其是如果管理人员想利用代码检查的结果，那么就与检查过程的目的背道而驰了。

### 3.2.4 代码检查的衍生功效

除了可以发现错误这个主要作用之外，代码检查还有其他的衍生作用。其一，程序员通常会得到编程风格、算法选择及编程技术等方面的反馈信息。其二，其他参与者也可以通过接触程序员的错误和编程风格而同样受益匪浅。通常来说，这种类型的测试方法能够增强项目中团队的凝聚力，减少消极人际关系滋长的可能性，有利于打造高度合作的、高效的以及信得过的开发模式。（要辩证看待码检查的这些功效，一旦没有做好出现像前面提到的人身攻击之类的事情，则造成恶劣影响，所以进行代码检查一定要准备充分且不断摸索成功经验，摒弃不好的实践。——译者注）

最后还有，代码检查是能够在早期发现程序中脆弱部位的方法之一，有助于在测试过程中将更多的注意力集中在这些脆弱地方（与第2章第9条测试原则不谋而合）。

## 3.3 用于代码检查的错误列表

代码检查过程的一个重要部分就是对照一份错误列表，来检查程序是否存在常见错误。遗憾的是，有些错误列表更多地注重编程风格而不是错误（例如，“注释是否准确且有意义？”，“if-else代码段和do-while代码段是否缩进对

齐？”），错误检查太过模糊而实际上没有用（例如，“代码是否满足设计需求？”）。本节中讨论的错误列表是经多年对软件错误的研究编辑而成的。该错误列表在很大程度上是独立于编程语言的，也就是说，大多数的错误都可能出现在用任意语言编写的程序中。读者可以把自己使用的编程语言中特有的错误，以及代码检查发现的错误补充到这份错误列表中去。

### 3.3.1 数据引用错误

1. 是否有引用的变量未赋值或未初始化？这可能是最常见的编程错误，在各种环境中都可能发生。在引用每个数据项（如变量、数组元素、结构中的域）时，应试图非正式地“证明”该数据项在当前位置具有确定的值。
2. 对于所有的数组引用，是否每一个下标的值都在相应维规定的界限之内？
3. 对于所有的数组引用，是否每一个下标的值都是整数？虽然在某些语言中这不是错误，但这样做是危险的。
4. 对于所有的通过指针或引用变量的引用，当前引用的内存单元是否分配？这就是所谓的“虚调用”（dangling reference）错误。当指针的生命期大于所引用内存单元的生命期时，错误就会发生。当指针引用了过程中的一个局部变量，而指针的值又被赋给一个输出参数或一个全局变量，过程返回（释放了引用的内存单元）结束，尔后程序试图使用指针的值时，这种错误就会发生。与前面检查错误的方法类似，应试图非正式地“证明”，对于每个使用指针值的引用，引用的内存单元都存在。
5. 如果一个内存区域具有不同属性的别名，当通过别名进行引用时，内存区域中的数据值是否具有正确的属性？在FORTRAN语言中对EQUIVALENCE语句使用，或COBOL语言中对REDEFINES语句使用的地方，都可能发生这种错误。例如，一个FORTRAN语言程序包含一个实型变量A和一个整型变量B，两者都通过使用EQUIVALENCE语句而成为同一内存区域的别名。如果程序先对A赋值，然后又引用变量B，由于机器可能会将内存中用浮点位表示的实数当做整数，在这种情况下错误就可能发生。
6. 变量值的类型或属性是否与编译器所预期的一致？当C、C++或COBOL程序将某个记录读到内存中，并使用一个结构来引用它时，由于记录的物理表示与结构定义存在差异，这种情况下错误就可能发生。

### COBOL与Fortran背景资料

COBOL和Fortran是两门分别面向商业处理和科学计算开发方向的元老级别的编程语言，这两门语言为数代的程序员提高开发效率作出了不可估量的贡献。

COBOL（取自COmmon Business Oriented Language的粗体部分）的雏形诞生于1959年前后，其设计的主要目的是为了支撑大型机系统上的商业应用软件的开发工作，其最初的规格设计可谓是集众家之长，当时参与COBOL项目的有知名的计算机制造商以及联邦政府机关，他们共同创造了一门全新的、面向商业的、能够运行在各种硬件和操作系统之上的编程语言。

这些年来，COBOL语言标准不断完善和发展。到2002年，COBOL几乎能够在大多数的操作系统平台进行开发和运行，甚至还推出了一个用来集成.NET开发环境的面向对象版本。

到本书写作之际，目前最新版本的COBOL是Visual COBOL 2010。

Fortran（最早拼写成FORTRAN，不过现在更倾向于首字母大写的规范）的诞生比COBOL还要稍微早点，其规格定义最早可以追溯到20世纪50年代早中期。和COBOL类似，Fortran被设计用来针对某些特殊类型（尤其是科学和数字计算方面）的大型机应用系统开发。Fortran这个名字来源于当时IBM一个叫做Mathematical **FOR**mula **TRAN**slating System（名字取自字体加粗部分）的系统。虽然最初的Fortran只有32种语句，但是这已经远远超越了当时的汇编程序设计语言，是具有划时代意义的进步。

到本书出版之际，最新的Fortran版本是Fortran 2008，已经在2010年被标准委员会正式通过。和COBOL一样，Fortran语言的演进伴随着对更多硬件和操作系统的支持，不过有一点不同的是，不管是现有系统的开发还是老系统的维护，Fortran都可能比COBOL应用得更广泛。

7. 在使用的计算机上，当内存分配的单元小于内存可寻址的单元大小时，是否存在直接或间接的寻址错误？例如，在某些条件下，定长的位串不必以

字节边界为起点，但是地址又总是指向字节边界的。如果程序计算一个位串的地址，稍后又通过该地址引用这个位串，可能会指向错误的内存位置。将一个位串参数传送给一个子程序时，也可能发生这种情况。

8. 当使用指针或引用变量时，被引用的内存的属性是否与编译器所预期的一致？这种错误的一个例子是，当一个指向某个数据结构的C++指针，被赋值为另外的数据结构的地址。
9. 假如一个数据结构在多个过程或子程序中被引用，那么每个过程或子程序对该结构的定义是否都相同？
10. 如果字符串有索引，当对数组进行索引操作或下标引用，字符串的边界取值是否有“仅差一个”(off-by-one)的错误？
11. 对于面向对象的语言，是否所有的继承需求都在实现类中得到了满足？

### 3.3.2 数据声明错误

1. 是否所有的变量都进行了明确的声明？虽然没有明确声明不一定是错误，但通常却是麻烦的源头。举例来说，如果一个程序的子程序接收一个数组参数，却未将该参数定义为数组（如用 `DIMENSION` 语句），对该数组的引用（如 `C=A (I)`）会被解释为一个函数调用，导致计算机试图将此数组当做程序执行。另外，如果某个变量在一个内部过程或程序块中没有明确声明，是否可以理解为该变量在这个程序块中被共用？
2. 如果变量所有的属性在声明中没有明确说明，那么默认的属性能否被正确理解？举例来说，在Java语言中，程序接收到的没有正确声明的默认属性往往是导致意外情况发生的源头。
3. 如果变量在声明语句中被初始化，那么它的初始化是否正确？在很多语言中，数组和字符串的初始化比较复杂，因此也成为容易出错的地方。
4. 是否每个变量都被赋予了正确的长度和数据类型？
5. 变量的初始化是否与其存储空间类型一致？举例来说，如果Fortran语言子程序中的一个变量在每次调用子程序时都需要重新初始化一次，那么必须使用赋值语句对其初始化，而不应该用DATA语句。
6. 是否存在着相似名称的变量（如VOLT和VOLTS）？这种情况不一定是错误，但应被视为警告，这些名称可能会在程序中发生混淆。

### 3.3.3 运算错误

1. 是否存在不一致的数据类型（如非算术类型）的变量间的运算？
2. 是否有混合模式的运算？例如，将浮点变量与一个整型变量做加法运算。这种情况并不一定是错误，但应该谨慎使用，确保程序语言的转换规则能够被正确理解。看看下面的Java程序片段，显示了整数运算中可能发生的取整误差：

```
int x = 1;
int y = 2;
int z = 0;
z = x/y;
System.out.println ("z = " + z);
```

```
OUTPUT:
z = 0
```

3. 是否有相同数据类型、不同字长变量间的运算？
4. 赋值语句的目标变量的数据类型是否小于右边表达式的数据类型或结果？
5. 在表达式的运算中是否存在表达式向上或向下溢出的情况？也就是说，最终的结果看起来是个有效值，但中间结果对于编程语言的数据类型可能过大或过小。
6. 除法运算中的除数是否可能为0？
7. 如果计算机表达变量的基本方式是基于二进制的，那么运算结果是否不精确？也就是说，在一个二进制计算机上， $10 \times 0.1$ 很少会等于1.0。
8. 在特定场合，变量的值是否超出了有意义的范围？例如，对变量PROBABILITY赋值的语句可能需要进行检查，保证赋值始终为正且不大于1.0。
9. 对于包含一个以上操作符的表达式，赋值顺序和操作符的优先顺序是否正确？
10. 整数的运算是否有使用不当的情况，尤其是除法？举例来说，如果i是一个整型变量，表达式  $2 * i / 2 == i$  是否成立，取决于i是奇数还是偶数，或是先运算乘法，还是先运算除法。

### 3.3.4 比较错误

1. 是否有不同数据类型的变量之间的比较运算，例如，将字符串与地址、日期或数字相比较？



2. 是否有混合模式的比较运算，或不同长度的变量间的比较运算？如果有，应确保程序能正确理解转换规则。
3. 比较运算符是否正确？程序员经常混淆“至多”、“至少”、“大于”、“不小于”、“小于”和“等于”等比较关系。
4. 每个布尔表达式所叙述的内容是否都正确？在编写涉及“与”、“或”或“非”的表达式时，程序员经常犯错。
5. 布尔运算符的操作数是否是布尔类型的？比较运算符和布尔运算符是否错误地混在了一起？这是一类经常会犯的错误，这里我们描述几个典型错误的例子：
  - 如果想判断*i*是否在2~10之间，表达式 $2 < i < 10$ 是不正确的；相反，正确的应该是 $(2 < i) \&\& (i < 10)$ 。
  - 如果想判断*i*是否大于*x* 或 *y*，表达式 $i > x || y$ 也是不正确的，正确的应该是 $(i > x) || (i > y)$ 。
  - 如果要比较三个数字是否相等，表达式 $\text{if}(a==b==c)$ 的实际意思却大相径庭。
  - 如果需要验证数学关系 $x > y > z$ ，正确的表达式应该是 $(x > y) \&\& (y > z)$ 。
6. 在二进制的计算机上，是否有用二进制表示的小数或浮点数的比较运算？由于四舍五入，以及用二进制表示十进制数的近似度，这往往是错误的根源。
7. 对于那些包含一个以上布尔运算符的表达式，赋值顺序以及运算符的优先顺序是否正确？也就是说，如果碰到如同 $\text{if}((a==2) \&\& (b==2) || (c==3))$ 的表达式，程序能否正确理解是“与”运算在先还是“或”运算在先？
8. 编译器计算布尔表达式的方式是否会对程序产生影响？例如，语句 $\text{if}((x==0 \&\& (x/y) > z))$ 对于有的编译器来说是可接受的，因为其认为一旦“与”运算符的一侧为FALSE时，另一侧就不用计算；但是对于其他编译器来说，却可能引起一个被0除的错误。

### 3.3.5 控制流程错误

1. 如果程序包含多条分支路径，比如有计算GO TO语句，索引变量的值是否会大于可能的分支数量？例如，在语句  
GO TO (200,300,400), i

中，i的取值是否总是1、2或3？

2. 是否所有的循环最终都终止了？应设计一个非正式的证据或论据来证明每一个循环都会终止。
3. 程序、模块或子程序是否最终都终止了？
4. 由于实际情况没有满足循环的入口条件，循环体是否有可能从未执行过？如果确实发生这种情况，这里是否是一处疏漏？例如，如果循环以下面的语句作为开头：

```
for (i=x ; i<=z; i++) {
    ...
}
or...
while (NOTFOUND) {
    ...
}
```

当NOTFOUND初始时就为假，或者x大于z时，情况会如何呢？

5. 如果循环同时由迭代变量和一个布尔条件所控制（如一个搜索循环），如果循环越界（fall-through）了，后果会如何？例如，伪指令循环以

```
DO I=1 to TABLESIZE WHILE (NOTFOUND)
```

开头，如果NOTFOUND永不为假，会发生什么结果呢？

6. 是否存在“仅差一个”的错误，如迭代数量恰恰多一次或少一次？这在从0开始的循环中是常见的错误。我们会经常忘记将“0”作为一次计数。举例来说，如果想编写一段Java代码执行10次循环，下面的语句是错误的，因为它执行了11次：

```
for (int i=0; i<=10;i++) {
    System.out.println(i);
}
```

正确的应该是执行10次循环：

```
for (int i=0; i <=9;i++) {
    System.out.println(i);
}
```

7. 如果编程语言中有语句组或代码块的概念（例如do-while或{...}），是否每一组语句都有一个明确的while语句，并且do语句也与其相应的语句

组对应？或者，是否每一个左括号都对应有一个右括号？目前的大多数编译器都能识别出这些不匹配的情况。

8. 是否存在不能穷尽的判断？举例来说，如果一个输入参数的预期值是1，2或3，当参数值不为1或2时，在逻辑上是否假设了参数必定为3？如果是这样的话，这种假设是否有效？

### 3.3.6 接口错误

1. 被调用模块接收到的形参（parameter）数量是否等于调用模块发送的实参（argument）数量？另外，顺序是否正确？
2. 实参的属性（如数据类型和大小）是否与相应形参的属性相匹配？
3. 实参的量纲是否与对应形参的量纲相匹配？举例来说，是否形参以度为单位而实参以弧度为单位？
4. 此模块传递给彼模块的实参数量，是否等于彼模块期望的形参数量？
5. 此模块传递给彼模块的实参的属性，是否与彼模块相应形参的属性相匹配？
6. 此模块传递给彼模块的实参的量纲，是否与彼模块相应形参的量纲相匹配？
7. 如果调用了内置函数，实参的数量、属性、顺序是否正确？
8. 如果某个模块或类有多个入口点，是否引用了与当前入口点无关的形参？

下面PL/I程序的第二个赋值语句就存在这种错误：

```
A:  PROCEDURE(W,X);
    W=X+1;
    RETURN
B:  ENTRY (Y,Z);
    Y=X+Z;
    END;
```

9. 是否有子程序改变了某个原本仅为输入值的形参？
10. 如果存在全局变量，在所有引用它们的模块中，它们的定义和属性是否相同？
11. 常数是否以实参形式传递过？在一些用FORTRAN语言编写的程序中，诸如

```
CALL SUBX(J,3)
```

的语句是很危险的，因为如果子程序SUBX对其第二个形参进行赋值，常数3的值将会被改变。

### 3.3.7 输入/输出错误

1. 如果对文件明确声明过，其属性是否正确？
2. 打开文件的语句中各项属性的设置是否正确？
3. 格式规范是否与I/O语句中的信息相吻合？举例来说，在FORTRAN语言中，是否每个FORMAT语句都与相应的READ或WRITE语句相一致（就各项的数量和属性而言）？
4. 是否有足够的可用内存空间，来保留程序将读取的文件？
5. 是否所有的文件在使用之前都打开了？
6. 是否所有的文件在使用之后都关闭了？
7. 是否判断文件结束的条件，并正确处理？
8. 对I/O出错情况处理是否正确？
9. 任何打印或显示的文本信息中是否存在拼写或语法错误？
10. 程序是否正确处理了类似于“File Not Found”这样的错误？

### 3.3.8 其他检查

1. 如果编译器建立了一个标识符交叉引用列表，那么对该列表进行检查，查看是否有变量从未引用过，或仅被引用过一次。
2. 如果编译器建立了一个属性列表，那么对每个变量的属性进行检查，确保没有赋予过不希望的默认属性值。
3. 如果程序编译通过了，但计算机提供了一个或多个“警告”或“提示”信息，应对此逐一进行认真检查。“警告”信息指出编译器对程序某些操作的正确性有所怀疑；所有这些疑问都应进行检查。“提示”信息可能会罗列出没有声明的变量，或者是不利于代码优化的用法。
4. 程序或模块是否具有足够的鲁棒性？也就是说，它是否对其输入的合法性进行了检查？
5. 程序是否遗漏了某个功能？

这些检查列表在表3-1和表3-2中进行了总结。

表3-1 代码检查错误列表总结，第一部分

数据引用错误	运 算 错 误
1. 是否有引用的变量未赋值或未初始化？ 2. 下标的值是否在范围之内？ 3. 是否存在非整数下标？ 4. 是否存在虚调用？ 5. 当使用别名时属性是否正确？ 6. 记录和结构的属性是否匹配？ 7. 是否计算位串的地址？是否传递位串参数？ 8. 基础的存储属性是否正确？ 9. 跨过程的结构定义是否匹配？ 10. 索引或下标操作是否有“仅差一个”的错误？ 11. 继承需求是否得到满足？	1. 是否存在非算术变量间的运算？ 2. 是否存在混合模式的运算？ 3. 是否存在不同字长变量间的运算？ 4. 目标变量的大小是否小于赋值大小？ 5. 中间结果是否上溢或下溢？ 6. 是否存在被0除？ 7. 是否存在二进制的精确度？ 8. 变量的值是否超过了有意义的范围？ 9. 操作符的优先顺序是否被正确理解？ 10. 整数除法是否正确？
数据声明错误	比 较 错 误
1. 是否所有的变量都已声明？ 2. 默认的属性是否被正确理解？ 3. 数组和字符串的初始化是否正确？ 4. 变量是否赋予了正确的长度、类型和存储类？ 5. 初始化是否与存储类相一致？ 6. 是否有相似的变量名？	1. 是否存在不同类型变量间的比较？ 2. 是否存在混合模式的比较运算？ 3. 比较运算符是否正确？ 4. 布尔表达式是否正确？ 5. 比较运算是否与布尔表达式相混合？ 6. 是否存在二进制小数的比较？ 7. 操作符的优先顺序是否被正确理解？ 8. 编译器对布尔表达式的计算方式是否被正确理解？

表3-2 代码检查错误列表总结，第二部分

控制流程错误	输入/输出错误
1. 是否超出了多条分支路径？ 2. 是否每个循环都终止了？ 3. 是否每个程序都终止了？ 4. 是否存在由于入口条件不满足而跳过循环体？ 5. 可能的循环越界是否正确？ 6. 是否存在“仅差一个”的迭代错误？ 7. DO/END语句是否匹配？ 8. 是否存在不能穷尽的判断？ 9. 输出信息中是否有文字或语法错误？	1. 文件属性是否正确？ 2. OPEN语句是否正确？ 3. I/O语句是否符合格式规范？ 4. 缓冲大小与记录大小是否匹配？ 5. 文件在使用前是否打开？ 6. 文件在使用后是否关闭？ 7. 文件结束条件是否被正确处理？ 8. 是否处理了I/O错误？
接 口 错 误	其 他 检 查
1. 形参的数量是否等于实参的数量？ 2. 形参的属性是否与实参的属性相匹配？ 3. 形参的量纲是否与实参的量纲相匹配？ 4. 传递给被调用模块的实参个数是否等于其形参个数？	1. 在交叉引用列表中是否存在未引用过的变量？ 2. 属性列表是否与预期的相一致？ 3. 是否存在“警告”或“提示”信息？ 4. 是否对输入的合法性进行了检查？

(续)

接口 错 误	其 他 检 查
5. 传递给被调用模块的实参属性是否与其形参属性匹配？	5. 是否遗漏了某个功能？
6. 传递给被调用模块的实参量纲是否与其形参量纲匹配？	
7. 调用内部函数的实参的数量、属性、顺序是否正确？	
8. 是否引用了与当前入口点无关的形参？	
9. 是否改变了某个原本仅为输入值的形参？	
10. 全局变量的定义在模块间是否一致？	
11. 常数是否以实参形式传递过？	

3.4 代码走查

代码走查与代码检查很相似，都是以小组为单位进行代码阅读，是一系列规程和错误检查技术的集合。代码走查的过程与代码检查大体相同，但是规程稍微有所不同，采用的错误检查技术也不一样。

就像代码检查一样，代码走查也是采用持续一至两个小时的不间断会议的形式。代码走查小组由三至五人组成，其中一个人扮演类似代码检查过程中“协调人”的角色，一个人担任秘书（负责记录所有查出的错误）的角色，还有一个人担任测试人员。关于这三到五个人的组成结构，有各种各样的建议。当然，程序员应该是其中之一。我们建议另外的参与者应该包括：

- 一位极富经验的程序员；
- 一位程序设计语言专家；
- 一位程序员新手（可以给出新颖、不带偏见的观点）；
- 最终维护程序的人员；
- 一位来自其他不同项目的人员；
- 一位来自该软件编程小组的程序员。

开始的过程与代码检查相同：参与者在走查会议的前几天得到材料，这样可以专心钻研程序。然而走查会议的规程则不相同。不同于仅阅读程序或使用错误检查列表，代码走查的参与者“使用了计算机”。被指定为测试人员的那个人会带着一些书面的测试用例（程序或模块具有代表性的输入集及预期的输出集）来参加会议。在会议期间，每个测试用例都在人们脑中进行推演。也就是说，把测试



数据沿程序的逻辑结构走一遍。程序的状态（如变量的值）记录在纸张或白板上以供监视。

当然，这些测试用例必须结构简单、数量较少，因为人脑执行程序的速度比计算机执行程序的速度慢上若干量级。因此，这些测试用例本身并不起到关键的作用；相反，它们的作用是提供了启动代码走查和质疑程序员思路及其设想的手段。在大多数的代码走查中，很多问题是在向程序员提问的过程中发现的，而不是由测试用例本身直接发现的。

与代码检查相同，代码走查参与者所持的态度非常关键。提出的建议应针对程序本身，而不应针对程序员。换句话说，软件中存在的错误不应被视为编写程序的人员自身的弱点。相反，这些错误应被看做是伴随着软件开发的艰难性所固有的。

与代码检查过程中描述的相似，代码走查应该有一个后续过程。同样，代码检查所带来的附带作用（如可以发现易出错的程序区域，通过接触软件错误、编程风格和方法来获得教育等）同样也会发生在代码走查过程中。

### 3.5 桌面检查

人工查找错误的第三种过程是古老的桌面检查方法。桌面检查可视为由单人进行的代码检查或代码走查：由一个人阅读程序，对照错误列表检查程序，对程序推演测试数据。

对于大多数人而言，桌面检查的效率是相当低的。其中的一个原因是，它是一个完全没有约束的过程。另一个重要的原因是它违反了本书第2章提出的测试原则，即人们一般不能有效地测试自己编写的程序。因此桌面检查最好由其他人而非该程序的编写人员来完成（例如，两个程序员可以相互交换各自的程序，而不是检查自己的程序）。但是即使这样，其效果仍然逊色于代码走查或代码检查。原因在于代码检查和代码走查小组中存在着互相促进的效应。小组会议培养了良性竞争的气氛，人们喜欢通过发现问题来展示自己的 ability。而在桌面检查中，由于没有向其他人展示的机会，也就缺乏这个显而易见的良好效应。简而言之，桌面检查胜过没有检查，但其效果远远逊色于代码检查和代码走查。

### 3.6 同行评审

最后一种人工评审方法与程序测试并无关系（其目标不是为了发现错误），却仍在这里谈到，这是因为它与代码阅读的思想有关。

同行评审是一种依据程序整体质量、可维护性、可扩展性、易用性和清晰性对匿名程序进行评价的技术。该项技术的目的是为程序员提供自我评价的手段。

选出一位程序员来担任这个评审过程的管理员，管理员又会挑选出6~20名参与者（为保持匿名性，6人是最少数量）。这些参与者都应具备相似的背景（例如，不能把Java应用程序员与汇编语言系统程序员编为一组）。要求每名参与者都挑选出两个由自己编写的程序以供评审。其中的一个程序应是参与者自认为能代表其自身能力的最好作品，而另一个则是参与者自认为质量较差的作品。

当所有的程序都收集完毕后，就将这些程序随机分发给参与者。每名参与者拿到4个程序进行评审，其中的两个是“最好”的程序，另外两个则是相对“较差”的程序，但评审人自己并不知道。每名参与者每评审一个程序得花费30分钟，评审完后填写一张评价表。所有4个程序都评审完后，参与者对4个程序的相对质量进行分级。评价表要求评审人用1~10的分值（1代表明确的“是”，10代表明确的“否”），对诸如下面的问题进行回答：

- 程序是否易于理解？
- 高层次的设计是否可见且合理？
- 低层次的设计是否可见且合理？
- 修改此程序对评审者而言是否容易？
- 评审者是否会以编写出该程序而骄傲？

评审人还应给出总的评价和建议的改进意见。

评审结束之后，参与者会收到自己的那两个程序的匿名评价表，此外还会收到一个带统计的总结，说明在所有的程序中其程序的整体和具体得分情况，以及他对其他程序的评价与其他评审人对同一程序打分的比较分析情况。同行评审的目的是让程序员对自身的编程技术进行自我评价。同样，该过程也适用于企业开发和课堂教学环境。

### 3.7 小结

本章讨论了软件开发人员通常不会考虑到的一种测试形式——人工测试。大多数人认为，因为程序是为了供机器执行而编写的，那么也应由机器来对程序进行测试。这种想法是有问题的。人工测试方法在暴露错误方面是很有成效的。实际上，大多数的软件项目都应使用到以下的人工测试方法：

- 利用错误列表进行代码检查。
- 小组代码走查。
- 桌面检查。
- 同行评审。

另一种人工测试（基于人的测试）就是本章开头提到的可用性测试，这是一种黑盒测试技术，需要测试人员站在最终用户实用的角度来评估软件的可用性程度。这一部分将在本书第7章介绍。



# 软件测试的艺术 (原书第3版)

*The Art of Software Testing* (Third Edition)

自从本书第1版付梓到现在30余年,计算机软硬件发生了翻天覆地的变化。不过这本书却经受住了时间的考验。与大多数的软件测试书籍都偏重于介绍某些特别的开发技术、脚本语言或者测试方法不同,本书以简洁明快的写作风格全面而细致地展示了那些久经考验的软件测试方法和智慧。

第3版将经过历史沉淀的经典法则应用到当今最热门的技术领域之中,包括:

- iPhone、iPad、BlackBerry、Android 以及其他移动设备的应用测试
- 可用性测试
- 互联网应用、电子商务和敏捷编程环境的测试

对于软件开发人员、IT项目经理,以及学生或更多相关的读者,本书将成为案头不可缺少的资料,当你通过本书发现第一个bug并修复之后,一定会觉得本书物超所值。

## 作者简介

**Glenford J. Myers** IBM系统研究院前高级研究员,同时还是RadiSys公司的创始人和前CEO。

**Tom Badgett** 曾经主管大型企业软件开发团队,同时他还是PCjr和Digital News等主流计算机杂志的技术编辑。

**Corey Sandler** 计算机新闻界的先锋,他曾经负责Gannett Newspapers和the Associated Press的技术部分以及之后成为PC Magazine的第一任主编。他同时还是Digital News(针对DEC小型机的一份报纸)的编辑创始团队成员。他著作等身,覆盖了从计算机到商业等多个领域。

## 原书封面



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: hzsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书: [www.china-pub.com](http://www.china-pub.com)

华章图书 · 杨宇梅

上架指导: 计算机/软件工程

ISBN 978-7-111-37660-6



9 787111 376606

定价: 39.00元