The Art of Software Testing

Second Edition

Glenford J. Myers

Revised and Updated by

Tom Badgett and Todd M.Thomas

with Corey Sandler

Copyright© 2004 by Word Association, Inc. All rights reserved.
Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

整理：丁兆杰,黄晰晖,黄米吉,林嘉,马晓靖,王博,吴航,余伟辰,俞培源

v 1.2 7/1/2008

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》自付印以来已出版到第 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功和更新。

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。

未经同意，严禁以任何形式拷贝

引言

在本书 1979 年第 1 版出版的时侯，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”的集度来进行探讨。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夫的对象，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

未经同意，严禁以任何形式拷贝

第 1 章 一个自我评价测试

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

未经同意，严禁以任何形式拷贝

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，代表了二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. 是否能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 表示一个不规则二角形，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

未经同意，严禁以任何形式拷贝

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。

未经同意，严禁以任何形式拷贝

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，将耗费大最的时间和人力资源，以至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：

未经同意，严禁以任何形式拷贝

“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，达就说明为什么人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。大多数的项日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。

未经同意，严禁以任何形式拷贝

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某些实验检测，却没有诊断出任何病因，我们就不会认为这此实验检测是“成功的”。之所以是“不成功的”检侧，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l0 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，软件测试更适合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

未经同意，严禁以任何形式拷贝

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

未经同意，严禁以任何形式拷贝

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，测试投人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）。

未经同意，严禁以任何形式拷贝

在这里我们的目标是针对达种测试策略，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最可能达到天文数字。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 $5^{20}+5^{19}\cdots$ 十 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。

未经同意，严禁以任何形式拷贝

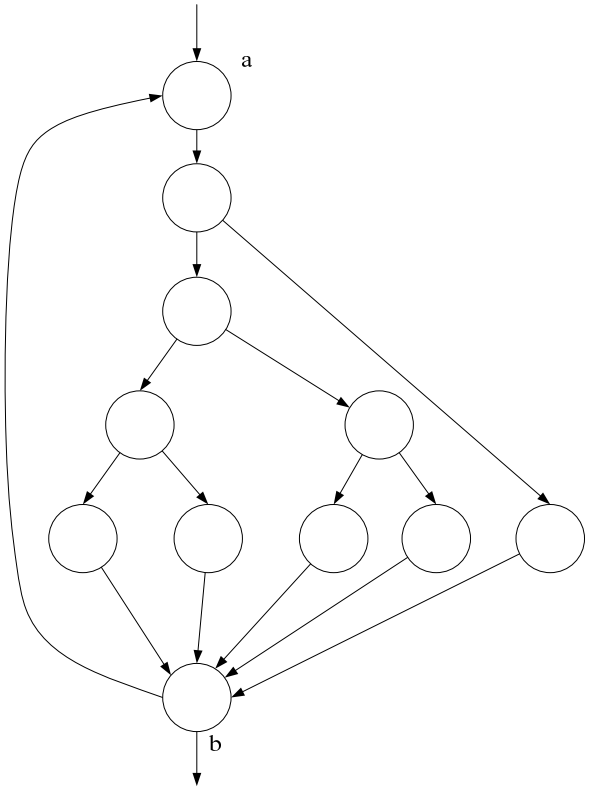


图 2-1 一个小型程序的控制流图

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。

未经同意，严禁以任何形式拷贝

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。

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-l 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
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，极具智力的挑战性的工作

未经同意，严禁以任何形式拷贝

原则 l：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：由干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

未经同意，严禁以任何形式拷贝

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

未经同意，严禁以任何形式拷贝

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负作用。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，饱含我们宝贵投人的测试用例，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人大量的工作，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更动后重新执行，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目成正比。

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。

未经同意，严禁以任何形式拷贝

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对软件测试过程的洞察或反馈。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。

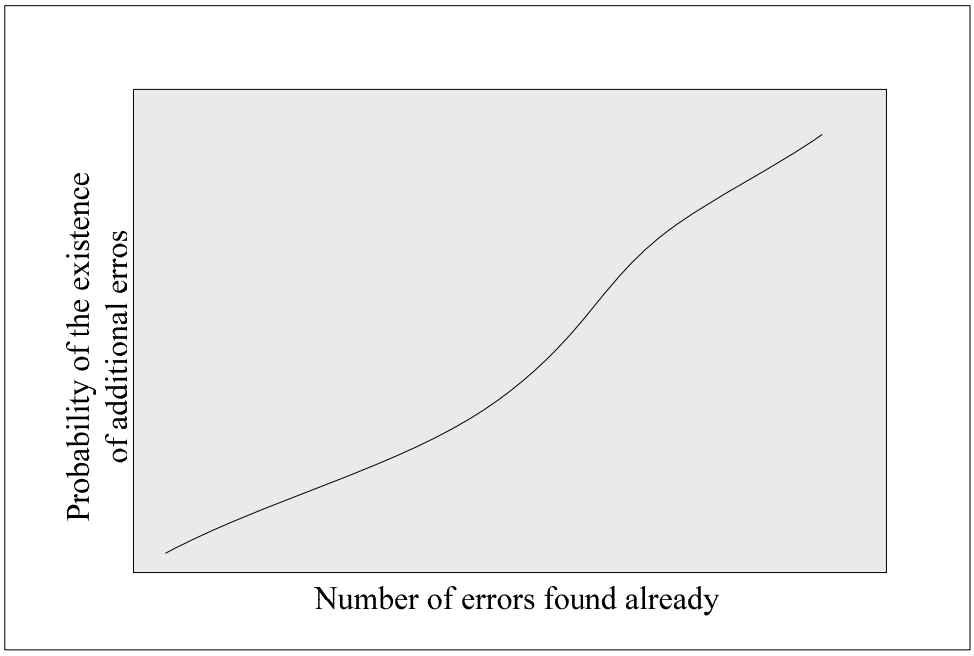


图 2-2 残存错误与已知错误间令人惊奇的联系

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，请牢记以下三个重要的测试原则：

- 软件测试是为发现错误而执行程序的过程。
- 一个好的测试用例具有较高的发现某个尚未发现的错误的可能性。
- 一个成功的测试用例能够发现某个尚未发现的错误。

未经同意，严禁以任何形式拷贝

第 3 章 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“尽可能快地修正这个缺陷”。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。

未经同意，严禁以任何形式拷贝

3.1 检查与走查(Inspections And Walkthroughs)

代码检查与走查是两种主要的人工测试方法。由于这两种方法具有很多的共同之处，在这里我们将一起讨论它们的相似点，而它们的不同之处将在后续章节中进行介绍。

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，它们的成功与本文第 2 章所述的那些原则有关。

在代码走查中，一组开发人员（三至四人为最佳）对代码进行审核。参加者当中只有一人是程序编写者。因此，软件测试的主要工作是由其他人，而不是软件编写者本人来完成．这符合“软件编写者往往不能有效地测试自己编写的软件”的测试原则。

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。

未经同意，严禁以任何形式拷贝

当然，可能存在对这统计数字的批评，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(Code Inspections)

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。

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。

协调人就像质量控制工程师。小组中的第二个成员是该程序的编码人员。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。

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

未经同意，严禁以任何形式拷贝

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. 对着历来常见的编码错误列表分析程序（该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 分钟之间。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。

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

未经同意，严禁以任何形式拷贝

了。

除了可以发现错误这个主要作用之外．代码检查还有几个有益的附带作用。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。其他参与者也可以通过接触其他程序员的错误和编程风格而同样受益匪浅。还有，代码检查还是早期发现程序中最易出错部分的方法之一，有助于在基于计算机的测试过程中将更多的注意力集中在这些地方（本文第 2 章中的测试原则之一）。

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

未经同意，严禁以任何形式拷贝

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，而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，引用的内存单元者都存在。

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作整数，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。

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？

未经同意，严禁以任何形式拷贝

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？没有明确声明虽然不一定是错误，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作程序执行。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的类型一致？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;
```

未经同意，严禁以任何形式拷贝

```
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，10×0.1 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丁 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住了一起？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 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 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(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 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 GOTO 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

```
GOTO (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

未经同意，严禁以任何形式拷贝

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++} {
...
}
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

```
D0 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 l 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？

未经同意，严禁以任何形式拷贝

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. 此模块传递给被模块的实参数量，是否等于被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/1 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。

未经同意，严禁以任何形式拷贝

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？

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？

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”信息可能会罗列山没有声明的变量，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？

未经同意，严禁以任何形式拷贝

5. 程序是否遗漏了某个功能？

这些检查列表在表 3-l 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？	1.是否存在非算术变量间的运算？
2.下标的值是否在范围之内？	2.是否存在混合摸式的运算？
3.是否存在非整数下标？	3.是否存在不同字长变量问的运算？
4.是否存在虚调用？	4.目标变量的大小是否小于赋值大小？
5.当使用别名时属性是否正确？	5.中间结果是否上溢或下溢？
6.记录和结构的属性是否匹配？	6.是否存住被 0 除？
7.是否计算位串的地址？是否传递位串参数？	7.是否存在二进制的不精确度？
8.基础的存储属性是否正确？	8.变量的值是否超过了有意义的范围？
9.跨过程的结构定义是否匹配？	9.操作符的优先顺序是否被正确理解？
10.索引或下标操作是否有“仅差一个”的错误？	10.整数除法是否正确？
11.继承需求是否得到满足？	

数据声明错误	比较错误
1.是否所有的变量都已声明？	1.是否存在不同类型变量间的比较？
2.默认的属性是否被正确理解？	2.是否存在混合模式的比较运算？
3.数组和字符串的初始化是否正确？	3.比较运算符是否正确？
4.变量是否赋予了正确的长度，类型和存储类？	4.布尔表达式是否正确？
5.初始化是否与存储类相一致？	5.比较运算是是否与布尔表达式相混合？
6.是否有相似的变量名？	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.编译器对布尔表达武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？	1.文件的属性是否正确？
2.是否每个循环都终止了？	2.0PEN 语句是否正确？
3.是否每个程序都终止了？	3.I/O 语句是否符合格式规范？
4.是否存在由于入口条件不满足而跳过循环体？	4.缓冲大小与记录大小是否匹配？
5.可能的循环越界是否正确？	5.文件在使用前是否打开？
6.是否存在“仅差一个”的迭代错误？	6.文件在使用后是否关闭？

未经同意，严禁以任何形式拷贝

7.DO/END 语句是否匹配？	7.文件结束条件是否被正确处理？
8.是否存在不能穷尽的判断？	8.是否处理了 I/O 错误？
9.输出信息中是否有文字或语法错误？	

接口错误	**其他检查**
1.形参的数量是否等于实参的数量？	1.在交叉引用列表中是否存在未引用过的变量？
2.形参的量纲是否与实参的量纲相匹配？	2.属性列表是否与预期的相一致？
3.形参的量纲是否与实参的量纲相匹配？	3.是否存在“警告”或“提示”信息？
4.传递给被调用模块的实参个数是否等于其形参个数？	4.是否对输入的合法性进行了检查？
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(Walkthroughs)

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小间断会议的形式。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。关于这二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：(1)一位极富经验的程序员；(2)一位程序设计语言专家；(3)一位程序员新手（可以给出新颖，不带偏见的观点），(4)最终将维护程序的人员；(5)一位来自其他不同项目的人员；(6)一位来自该软件编程小组的程序员。

未经同意，严禁以任何形式拷贝

开始的过程与代码检查相同：参与者在走查会议之前的几天得到材料，他们可以专心钻研程序。然而走查会议的程则不相同。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逻辑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，这些错误应被看作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(Desk Checking)

人工查找错误的第二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，而不是桌面检查自己的程序）。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

未经同意，严禁以任何形式拷贝

气氛，人们喜欢通过发现问题来展示自己的能力。而在桌面检查中，由于没有其他人可供展示，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 同行评分(Peer Ratings)

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。

同行评分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分过程的管理员，管理员又会挑选出大约 6～20 名参与者（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（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从 1～7 的分值（代表明确的“是”，7 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？

还要要求评审人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

未经同意，严禁以任何形式拷贝

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。同行评分的目的，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。大多数人都以为，因为程序是为了供机器执行而编写的，那么也该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。

未经同意，严禁以任何形式拷贝

第 4 章 测试用例的设计

除了第 2 章探讨的软件测试的心理学问题以外，软件测试中最重要的因素是设计和生成有效的测试用例。

然而，无论软件测试进行得如何具有创造性、如何完全，也不能保证软件中不存在任何错误。测试用例的设计如此重要，原因在于完全的测试是不可能的，对任何程序的测试必定是不完全的。那么，最显然的测试策略就是努力使测试尽可能完全。

由于时间和成本的约束，软件测试的最关键问题是：

在所有可能的测试用例中，哪个子集最有可能发现最多的错误？

对软件测试用例设计方法的研究为这个问题提供了答案。

一般而言，在所有的方法中效率最低的是随机输入测试，即在所有可能的输入值中随机选取某个子集来对程序进行测试的过程。就发现最多错误的可能性而言，随机选取而产生的测试用例集很少有可能是理想的或接近理想的子集。在本章中，我们将提出一套思考过程，该过程有助于更加睿智地选取测试数据。

本书第 2 章已经证明穷举的黑盒和白盒测试通常都是不可能的，但同时也建议：将这两种测试的要素组合起来得到一种合理的测试策略。本章将对这种策略进行研究。我们可以通过使用特定的面向黑盒测试的测试用例设计方法，而后使用白盒测试方法对程序的逻辑结构进行检查以补充这些测试用例，借此来设计出一个相当严格的测试。

本章将要讨论的测试方法如下：

黑盒测试	**白盒测试**
等价类划分	语句覆盖
边界值分析	判定覆盖
因果图分析	条件覆盖
错误测试	判定/条件覆盖
	多重条件覆盖

未经同意，严禁以任何形式拷贝

尽管上述方法将分开来进行讨论，但我们建议综合最多的（如果不能是全部的话）测试方法来设计严格的程序测试，因为每一种测试方法都有其独特的优势和弱点。举例来说，某种方法遗漏掉的错误，而用其他的方法就可能找出来。

没有人曾承诺说：软件测试会是容易的事。引用一位智者的话，“如果你觉得设计和编写程序很困难，你就并非一无所知。”

我们推荐的步骤是先使用黑盒测试方法来设计测试用例，然后视情况需要使用白盒测试方法来设计补充的测试用例。下面首先讨论较为有名的白盒测试方法。

4.1 白盒测试(White-Box Testing)

4.1.1 逻辑覆盖测试(Logic-Coverage Testing)

白盒测试关注的是测试用例执行的程度或覆盖程序逻辑结构（源代码）的程度。如同我们在本书第 2 章所看到的，完全的白盒测试是将程序中每条路径都执行到，然而对一个带有循环的程序来说，完全的路径测试并不切合实际。

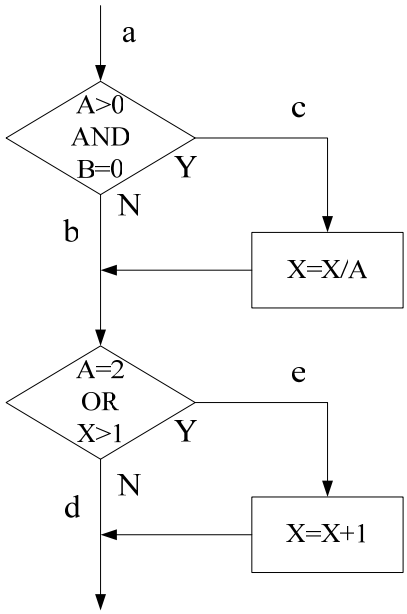


图 4-1 被测试的小程序

如果完全从路径测试中跳出来看，那么有价值的目标似乎就是将程序中的每条语句至少执行一次。遗憾的是，这恰是合理的白盒测试中较弱的准则。图 4-1 描述

了这种思想。假设图 4-1 代表了一个将要进行测试的小程序，其等价的 Java 代码段如下：

```
Public void foo(int a, int b, int x) {
    if (a>1 && b == 0) {
        x=x/a;
    }
    if (a==2 || x>1) {
       x=x+1;
    }
}
```

通过编写单个的测试用例遍历程序路径 ace，可以执行到每一条语句。也就是说，通过在点 a 处设置 A=2，B=0，X=3，每条语句将被执行一次（实际上，X 可被赋任何值）。

遗憾的是，这个准则相当不足。举例来说，也许第一个判断应是“或”，而不是“与”。如果这样，这个错误就会发现不到。另外，可能第二个判断应该写成“X>0”，这个错误也不会被发现。还有，程序中存在一条 X 未发生改变的路径（路径 abd），如果这是个错误，它也不会被发现。换句话说，语句覆盖这条准则有很大的不足，以至于它通常没有什么用处。

判定覆盖或**分支覆盖**是较强一些的逻辑覆盖准则。该准则要求必须编写足够的测试用例，使得每一个判断都至少有一个为“真”和为“假”的输出结果。换句话说，也就是每条分支路径都必须至少遍历一次。分支或判定语句的例子包括 switch，do-while 和 if-else 语句. 在一些程序语言如 FORTRAN 中，多重选择 GOTO 语句也是合法的。

判定覆盖通常可以满足语句覆盖。由于每条语句都是在要么从分支语句开始，要么从程序入口点开始的某条子路径上，如果每条分支路径都被执行到了，那么每条语句也应该被执行到了。但是，仍然还有至少三种例外情况：

- 程序中不存在判断。
- 程序或子程序/方法有着多重入口点。只有从程序的特定入口点进入时，某条特定的语句才能执行到。
- 在 ON 单元（ON-unit）里的语句。遍历每条分支路径并不一定能确保所有的 ON 单元都能执行到。由于我们将语句覆盖视为一个必要条件，那么，

作为似乎更佳准则的判定覆盖的定义理应涵盖语句覆盖。因此，判定覆盖要求每个判断都必须有“是”和“否”的结果，并且每条语句都至少被执行一次。换一种更简单的表达方式，即每个判断都必须有“是”和“否”的结果，而且每个入口点（包括 ON 单元）都必须至少被调用一次。

我们的探讨仅针对有两个选择的判断或分支，当程序中包含有多重选择的判断时，判定/分支覆盖准则的定义就必须有所改变。典型的例子有包含 select(case)语句的 Java 程序，包含算术（三重选择）IF 语句、计算或算术 GOTO 语句的 FORTRAN 程序，以及包含可选 GOTO 语句或 GO-TO-DEFENDING-ON 语句的 COBOL 程序。对于这些程序，判定/分支覆盖准则将所有判断的每个可能结果都至少执行一次，以及将程序或子程序的每个入口点都至少执行一次。

在图 4-1 中，两个涵盖了路径 ace 和 abd，或涵盖了路径 acd 和 abe 的测试用例就可以满足判定覆盖的要求。如果我们选择了后一种情况，两个测试用例的输入是 A=3，B=0，X=3 和 A=2，B=1，X=1。

判定覆盖是一种比语句覆盖更强的准则，但仍然相当不足。举例来说，我们仅有 50%的可能性遍历到那条 X 未发生变化的路径（也即，仅当我们选择前一种情况）。如果第二个判断存在错误（例如把 X>1 写成了 X<1，那么前面例子中的两个测试用例都无法找出这个错误。

比判定覆盖更强一些的准则是条件覆盖。在条件覆盖情况下，要编写足够的测试用例以确保将一个判断中的每个条件的所有可能的结果至少执行一次。因为，就如同判定覆盖的情况一样，这并不总是能让每条语句都执行到，因此作为对这条准则的补充就是对程序或子程序，包括 ON 单元的每一个入口点都至少调用一次。举例来说，分支语句

```
DO K=0 to 50 WHILE (J+K<QUEST)
```

包含两种情况：K 是否小于或等于 50？以及 J+K 是否小于 QUEST？因此，需要针对 K <= 50、K >50（达到循环的最后一次迭代）以及 J+K<QUEST、J+K>=QUEST 的情况设计测试用例。

图 4-1 有四个条件：A>1、B=0、A=2 以及 X>1。因此需要足够的测试用例，使得在点 a 处出现 A=2、A<2、X>1 及 X<=1 的情况。有足够数量的测试用例满足此准

未经同意，严禁以任何形式拷贝

则，用例及其遍历的路径如下所示：

1．A=2，B=0，X=4 ace

2．A=1，B=1，X=1 adb

请注意，尽管在本例中生成的测试用例数量是一样的，但条件覆盖通常还是要比判定覆盖更强一些。因为，条件覆盖可能（但并不总是这样）会使判断中的各个条件都取到两个结果（“真”和“假”），而判定覆盖却做不到这一点。举例来说，在相同的分支语句

```
DO  K=0 to 50 WHILE (J+K<QUEST)
```

中，存在一个两重分支（执行循环体，或者跳过循环体）。如果使用的是判定覆盖测试，将循环从 K = 0 执行到 K = 51 即可满足该准则，但从未考虑到 WHILE 子句为假的情况。如果使用的是条件覆盖准则，就需要设计一个测试用例为 J+K<QUEST 产生一个为假的结果。

虽然条件覆盖准则乍看上去似乎满足判定覆盖准则，但并不总是如此。如果正在测试判断条件 `IF (A&B)`，条件覆盖准则将要求编写两个测试用例：A 为真，B 为假；A 为假，B 为真。但是这并不能使 IF 语句中的 THEN 被执行到。对图 4-1 所示例子所进行的条件覆盖测试涵盖了全部判断结果，但这仅仅是偶然情况。举例来说，两个可选的测试用例：

1．A=2，B=0，X=3

2．A=1，B=1，X=1

涵盖了全部的条件结果，却仅涵盖了四个判断结果中的两个（这两个测试用例都涵盖到了路径 abe，因而不会执行第一个判断结果为真的路径，以及第二个判断结果为假的路径）。

显然，解决上面左右为难局面的办法就是所谓的判定/条件覆盖准则。这种准则要求设计出充足的测试用例。将一个判断中的每个条件的所有可能的结果至少执行一次，将每个判断的每个条件的所有可能的结果至少执行一次，将每个判断的所有可能的结果至少执行一次，将每个入口点都至少调用一次。

判定/条件覆盖准则的一个缺点是尽管看上去所有条件的所有结果似乎都执行

到了，但由于有些特定的条件会屏蔽掉其他的条件，常常并不能全部都执行到。请参见图 4-2 来观察此种情况。图 4-2 中的流程图描述的是编译器将图 4-1 中的程序编译生成机器代码的过程。源程序中的多重条件判断被分解成单个的判断和分支，因为大多数的机器都没有能执行多重条件判断的单独指令。那么，更为完全的测试覆盖似乎是将每个基本判断的全部可能的结果都执行到，而前两个判定覆盖的测试用例都做不到这点，它们未能执行到判断 H 中的结果为“假”的分支，以及判断 K 中结果为“真”的分支。

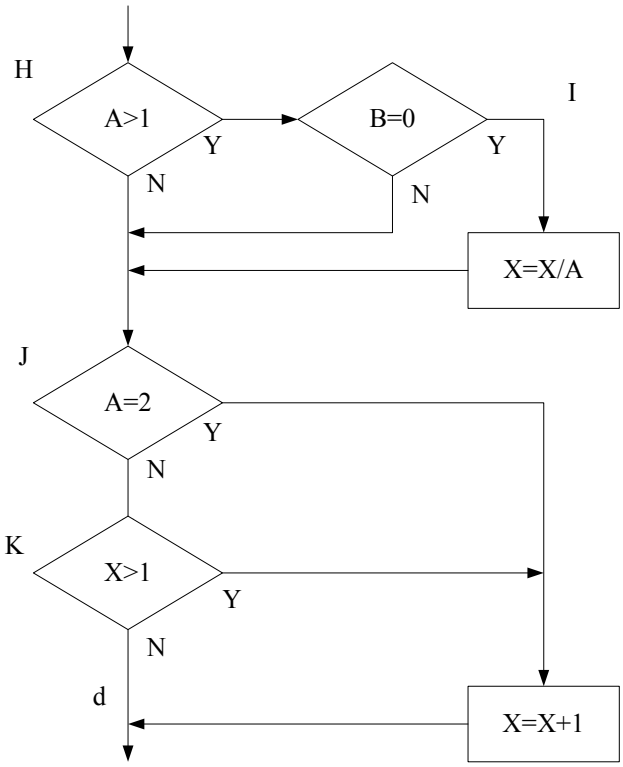


图 4-2 图 4-1 中程序的机器码

如图 4-2 所示，其中的原因是“与”和“或”表达式中某些条件的结果可能会屏蔽掉或阻碍其他条件，的判断。举例来说，如果“与”表达式中有个条件为“假”，那么就无须计算该表达式中的后续条件。同样，如果“或”表达式中有个条件为“真”，那么后续条件也无须计算。因此，条件覆盖或判定/条件履盖谁则不一定会发现逻辑表达式中的错误。

所谓的多重条件覆盖准则能够部分解决这个问题。该准则要求编写足够多的测

未经同意，严禁以任何形式拷贝

试用例，将每个判定中的所有可能的条件结果的组合，以及所有的入口点都至少执行一次。举例来说，考虑下面的伪代码程序；

```
NOTFOUND=TRUE;
DO I=1 TO TABSIZE WHILE (NOTFOUND); /*SEARCH TABLE*/
 ……searching logic……;
END
```

要测试四种情况：

1. I<=TABSIZE，并且 NOTFOUND 为真；
2. I<=TABSIZE，并且 NOTFOUND 为假（在到达表格尾部前查询指定条目）；
3. I>TABSIZE，并且 NOTFOUND 为真（查询了整个表格，未找到指定条目）；
4. I>TABSIZE，并且 NOTFOUND 为假（指定条目位于表格的最后位置）。

很容易发现，满足多重条件覆盖准则的测试用例集，同样满足判定覆盖准则、条件覆盖准则以及判定/条件覆盖淮则。

再次回到图 4-1 中，测试用例必须覆盖以下 8 种组合：

1. A>1，B=0
2. A>1，B<>0
3. A<=1，B<>0
4. A=2，X>1
5. A=2，X<=1
6. A<>2，X>1
7. A<>2，X<=1

注意，与左边的情况一样，第 5 至第 8 组合表示了第二个 if 语句的值。由于 X 可能在该 if 语句之前发生了改变，因此这个 if 语所需的值必需对程序逻辑进行回溯，以找到相对应的输入值，要测试的这 8 种组合并不一定意味着需要设计出 8 个测试用例。实际上，用 4 个测试用例就可以覆盖它们。下面是这些测试用例的输入，以及它们覆盖的组合：

A=3，B=0，X=4	覆盖组合 1，5
A=2，B=1，X=1	覆盖组合 2，6
A=1，B=0，X=2	覆盖组合 3，7
A=1，B=1，X=1	覆盖组合 4，8

未经同意，严禁以任何形式拷贝

图 4-1 的程序存在 4 条不同的路径，需要 4 个测试用例，这样的情况纯属巧合。事实上，这 4 个用例也没有覆盖到每条路径，路径 acd 就被遗漏掉了。举例来说，对于如下所示的判断语句，尽管它只包含两条路径，仍可能需要 8 个测试用例：

```
if (x==y && length(z)==0 && FLAG) {
    j=1
else
    i=1;
}
```

在存在循环的情况下，多重条件覆盖准则所需要的测试用例的数量通常会远远小于其路径的数量。

总的来说，对于包含每个判断只存在一种条件的程序，最简单的测试准则就是设计出足够数量的测试用例，实现：（1）将每个判断的所有结果都至少执行一次；（2）将所有的程序入口（例如入口点或 ON 单元）都至少调用一次，以确保全部的语句都至少执行一次。而对于包含多重条件判断的程序，最简单的测试准则是设计出足够数量的测试用例，将每个判断的所有可能的条件结果的组合，以及所有的入口点都至少执行一次（加入“可能”二字，是因为有些组合情况难以生成）。

4.1.2 等价划分(Equivalence Partitioning)

本书第 2 章将一个好的测试用例描述为具有相当高的可能性发现某个错误来，此外还讨论了对程序的穷举输入测试是无法实现的。因此，当测试某个程序时，我们就被限制在从所有可能的输入中努力找出某个小的子集。理所当然，我们要找的子集必须是正确的，并且是可能发现最多错误的子集。

确定这个子集的一种方法，就是要意识到一个精心挑选的测试用例还应具备另外两个特性：

1. 严格控制测试用例的增加，减少为达到“合理测试”的某些既定目标而必须设计的其他测试用例的数量。
2. 它覆盖了大部分其他可能的测试用例。也就是说，它会告诉我们，使用或不使用这个特定的输入集合，哪些错误会被发现，哪些会被遗漏掉。

虽然这两个特性看起来很相似，但描述的却是截然不同的两种思想。第一个特

性意味着，每个测试用例都必须体现尽可能多的不同的输入情况，以使最大限度地减少测试所需的全部用例的数量。而第二个特性意味着应该尽量将程序输入范围进行划分，将其划分为有限数量的等价类，这样就可以合理地假设（但是，显然不能绝对肯定）测试每个等价类的代表性数据等同于测试该类的其他任何数据。也就是说，如果等价类的某个测试用例发现了某个错误，该等价类的其他用例也应该能发现同样的错误。相反，如果测试用例没能发现错误，那么我们可以预计，该等价类中的其他测试用例不会出现在其他等价类中，因为等价类是相互交迭的。

这两种思想形成了称为等价划分的黑盒测试方法。第二种思想可以用来设计一个“令人感兴趣的”输入条件集合以供测试，而第一个思想可以随后用来设计涵盖这些状态的一个最小测试用例集。

本书第 1 章中三角形程序的一个等价类的例子是集合“三个值相等、都大于 0 的整型数据”。将此作为一个等价类后，我们就可以说，如果对该集合中某个元素所进行的测试没有发现错误的话，那么对该集合中其他元素所进行的测试也不大可能会发现错误。换言之，我们的测试时间最好花在其他地方（其他的等价类）。

使用等价划分方法设计测试用例主要有两个步骤：（1）确定等价类；（2）生成测试用例。

外部条件	有效等价类	无效等价类

图 4-3 等价类列举表

1．确定等价类

确定等价类是选取每一个输入条件（通常是规格说明中的一个句子或短语）并将其划分为两个或更多的组。可以使用图 4-3 中的表格来进行划分。注意，我们确定了两类等价类：**有效等价类**代表对程序的有效输入，而**无效等价类**代表的则是其他任何可能的输入条件（即不正确的输入值）。这样，我们遵循了本书第 2 章阐述

未经同意，严禁以任何形式拷贝

的测试原则，即要注意无效和未预料到的输入情况。

在给定了输入或外部条件之后，确定等价类大体上是一个启发式的过程。下面给出了一些指导原则：

1. 如果输入条件规定了一个取值范围（例如，“数量可以是从 1 到 999”），那么就应确定出一个有效等价类（1<数量<999），以及两个无效等价类（数量<1，数量>999）。
2. 如果输入条件规定了取值的个数（例如，“汽车可登记一至六名车主”），那么就应确定出一个有效等价类和两个无效等价类（没有车主，或车主多于六个）。
3. 如果输入条件规定了一个输入值的集合，而且有理由认为程序会对每个值进行不同处理（例如，“交通工具的类型必须是公共汽车、卡车、出租车、火车或摩托车”），那么就应为每个输入值确定一个有效等价类和一个无效等价类（例如，“拖车”）。
4. 如果存在输入条件规定了“必须是”的情况，例如“标识符的第一个字符必须是字母”，那么就应确定一个有效等价类（首字符是字母）和一个无效等价类（首字符不是字母）。

如果有任何理由可以认为程序并未等同地处理等价类中的元素，那么应该将这个等价类再划分为小一些的等价类，稍后我们将给出这个过程的例子。

2．生成测试用例

第二步是使用等价类来生成测试用例，其过程如下：

1. 为每个等价类设置一个不同的编号。
2. 编写新的测试用例，尽可能多地覆盖那些尚未被涵盖的有效等价类，直到所有的有效等价类都被测试用例所覆盖（包含进去）。
3. 编写新的用例，覆盖一个且仅一个尚未被覆盖的无效等价类，直到所有的无效等价类都被测试用例所覆盖。

用单个测试用例覆盖无效等价类，是因为某些特定的输入错误检查可能会屏蔽或取代其他输入错误检查。举例来说，如果规格说明规定了“请输入书籍类型（硬皮、软皮或活页）及数量（1～999 ）”，代表两个错误输入（书籍类型错误，数量

未经同意，严禁以任何形式拷贝

错误）的测试用例“XYZ 0”，很可能不会执行对数量的检查，因为程序也许会提示“XYZ 是未知的书籍类型”，就不检查输入的其余部分了。

4.1.3 一个范例

作为一个例子，假设我们正在为 FORTRAN 语言的一个子集开发编译器，我们希望对 DIMENSION 语句的语法检查进行测试。该语句的规格说明如下所示(这不是 FORTRAN 语言中的完整 DIMENSION 语句，我们对其进行了适当的剪裁，使其适合作为教科书的样例。不要被其误导，以为测试实际的程序就像测试本书中的样例一样容易)。在规格说明中，斜体字中的项是在实际语句中必须被特定实体取代的语法单元，使用括弧代表可选项，省略号代表前面的项可能会连续重复出现多次。

DIMENSION 语句用来定义数组的大小

DIMENSION 语句的格式如下:

DIMENSION *ad*[,*ad*]…

其中 *ad* 是数组描述符，其格式如下:

n(d[,d] …)

其中 n 是数组的符号名，d 是数组的维说明符。符号名可以由 1~6 个字母或数字组成，其中首字符必须是字母。一个数组最少有 1 个维，最多有 7 个维。维说明符的格式如下:

[lb:]ub

其中 lb 与 ub 分别是维的下边界和上边界。边界可以是-65534~65535 之间的一个常数，或是一个整型变变量名（但不能走数组元素名）。如果未指定 1b，则其默认值为 1。ub 的值必须大于或等于 lb 。如果指定了 lb，则其值可为负数、零或正数。就全部语句而言，DIMENSION 语句可写成连续多行（规格说明结束）。

第一步应该是确定输入条件，然后为输入条件确定等价类。这些步骤都以表格形式记录在表 4-1 中。括号中的数字代表不同等价类的标识符。

表 4-1 等价类

Input Condition	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
Number of array descriptors	one (1), > one (2)	none (3)
Size of array name	1–6 (4)	0 (5), > 6 (6)
Array name	has letters (7), has digits (8)	has something else (9)
Array name starts with letter	yes (10)	no (11)

未经同意，严禁以任何形式拷贝

Number of dimensions	1–7 (12)	0 (13), > 7 (14)
Upper bound is	constant (15), integer variable (16)	array element name (17),something else (18)
Integer variable name	has letter (19), digits (20)	has has something else (21)
Integer variable starts with letter	yes (22)	no (23)
Constant	-65534－65535 (24)	≤ 65534 (25), >65535 (26)
Lower bound specified	yes (27),	no (28)
Upper bound to lower bound	greater than (29), equal (30)	less than (31)
Specified lower bound	negative (32), zero (33), > 0 (34)	
Lower bound is	constant (35), integer variable (36),	array element name (37) something else (38)
Multiple lines	yes (39),	no (40)

下一个步骤应该是编写一个测试用例以覆盖一个或多个有效等价类。举例来说，测试用例

```
DIMENSION A(2)
```

覆盖了第 1，4，7，10，12，15，24，28，29，40 等价类。

再下一个步骤应该是设计一个或更多的测试用例以覆盖剩余的有效等价类，如以下形式的测试用例

```
DIMENSION A 12345 (1,9,J4XXXX,65535,1,KLM，
X   100),BBB[-65534:100,0:1000,10:10,I:65535]
```

覆盖了剩余的等价类。而无效输入等价类及其测试用例如下所示：

```
(3): DIMENSION
(5): DIMENSION (10)
(6): DIMENSION A234567(2)
(9): DIMENSION A.1(2)
(11): DIMENSION 1A(10)
(13): DIMENSION B
(14): DIMENSION B(4,4,4,4,4,4,4,4)
(17): DIMENSION B(4, A(2))
(18): DIMENSION B(4, 7)
(21): DIMENSION C(1, 10)
(23): DIMENSION C(10, 1J)
(25): DIMENSION C(-65535:1)
(26): DIMENSION C(65536)
(31): DIMENSION D(4:3)
(37): DIMENSION D(A(2):4)
```

未经同意，严禁以任何形式拷贝

(38)：D(.:4)

因此，所有的等价类都被 18 个测试用例全部所覆盖了。读者可以考虑一下，如何将这些测试用例与用特殊方法生成的测试用例集进行比较。

尽管等价划分方法要比随机选取测试用例优越得多，但它仍然存在不足。例如，这种方法忽略了某些特定类型的高效测试用例，下面介绍的两种方法（边界值分析与因果图）可以弥补其中的很多不足。

4.1.4 边界值分析(Boundary-Value Analysis)

经验证明，考虑了**边界条件**的测试用例与其他没有考虑边界条件的测试用例相比，具有更高的测试回报率。所谓边界条件，是指输入和输出等价类中那些恰好处于边界、或超过边界、或在边界以下的状态。边界值分析方法与等价划分方法存在两方面的不同：

1. 与从等价类中挑选出任意一个元素作为代表不同，边界值分析需要选择一个或多个元素，以便等价类的每个边界都经过一次测试。
2. 与仅仅关注输入条件（输入空间）不同，还需要考虑从结果空间（输出等价类）设计测试用例。

很难提供一份如何进行边界值分析的“详细说明’，因为这种方法需要一定程度的创造性，以及对问题采取一定程度的特殊处理办法（因此，就像测试的许多其他方面一样，这更多的是项智力工作，并非其他的什么）。然而，我们还是给读者提供一些通用指南：

1. 如果输入条件规定了一个输入值范围，那么应针对范围的边界设计测试用例，针对刚刚越界的情况设计无效输入测试用例。举例来说，如果输入值的有效范围是-1.0 至+1.0，那么应针对-1.0、1.0、-1.001 和 1.001 的情况设计测试用例。
2. 如果输入条件规定了输入值的数量，那么应针对最小数量输入值、最大数量输入值，以及比最小数量少一个、比最大数量多一个的情况设计测试用例。举例来说，如果某个输入文件可容纳 1~255 条记录，那么应根据 0、1、255 和 256 条记录的情况设计测试用例。

未经同意，严禁以任何形式拷贝

3. 对每个输出条件应用指南 1。举例来说，如果某个程序按月计算FICA[1]的扣除额，且最小金额是$0.00，最大金额为$1165.25，那么应该设计测试用例来测试扣除$0.00 和$1165.25 的情况。此外，还应观察是否可能设计出导致扣除金额为负数或超过$1165.25 的测试用例。注意，检查结果空间的边界很重要，因为输入范围的边界并不总是能代表输出范围的边界情况（例如，三角正弦函数sin的情况就如此）。同样，总是产生超过输出范围的结果也是不大可能的，但无论如何，应该考虑这种可能性。
4. 对每个输出条件应用指南 2。如果某个信息检索系统根据输入请求显示关联程度最高的信息摘要，而摘要的数量从未超过 4 条，则应编写测试用例，使程序显示 0 条、1 条和 4 条摘要，还应设计测试用例，导致程序错误地显示 5 条摘要。
5. 如果程序的输入或输出是一个有序序列（例如顺序的文件、线性列表或表格），则应特别注意该序列的第一个和最后一个元素。
6. 此外，发挥聪明才智找出其他的边界条件。

本书第 1 章中的三角形分析程序可以说明边界值分析的必要性。作为代表三角形的输入值，它们必须是大于 0 的整数，而且其中任意两个之和应大于第三个。如果定义了等价划分，可能会确定一个满足此条件的等价类，以及另一个两个输入之和不大于第三个的等价类。因此，3-4-5 和 1-2-4 两个都是可能的测试用例。然而，我们遗漏了一个可能的错误，即如果程序中表达式写成了 A+ B>=C，而不是 A+ B>C，那么程序就会错误地告诉我们 1-2-3 表示的是一个有效的不规则三角形。因此，边界值分析方法和等价划分之间的重要区别是，边界值分析考察正处于等价划分边界或在边界附近的状态。

作为边界值分析的一个例子，考虑下面的程序规格说明：

MTEST 是一个多项选择考试的评分程序。程序的输入是一个名为 OCR 的数据文件，包含多个长度为 80 个字符的记录。按照文件的格式要求. 第一个记录的内容是标题，作为每份输出报告的标题。后面的一组记录描述了试题的标准答案，这些记录的最后一个字符是 “2”。在这组记录的首条记录中，第 1～第 3 列存储的是试题的数量（一个 1～999 的数），第 10～第 59 存储的是第 1～第

[1]美国联邦社会保险捐款法. 纳税人应依据此项法律交纳一定金额。—译者注

50 道试题的标准答案（任何字符都为有效答案），后续记录的第 10～第 59 列存储的是第 51～第 100 道试题、第 101～第 150 道试题的标准答案等等。

第三组记录描述的是每个学生的答案，这些记录的最后一个字符皆为“3”。对于每个学生来说，第一条记录的第 1～第 9 列存储的是学生的名字或编号（任意字符），第 10～第 59 列存储的是该学生对第 1～第 50 道试题的答案。如果本次考试试题超过 50 个，该学生的后续记录的第 10～第 59 列存储的是第 51～第 100、第 101～第 150 道试题的答案等等。学生的人数最多是 200。输入数据如图 4-4 所示。四个输出报告分别是:

1. 按学生的编号排序的报告，显示每名学生的成绩（正确答案的百分比）和名次。
2. 按成绩排序的报告。
3. 显示成绩的平均值、中间值和标准偏差的报告。
4. 按问题的编号排序的报告，显示正确回答每个问题的学生比例。

（规格说明结束）

未经同意，严禁以任何形式拷贝

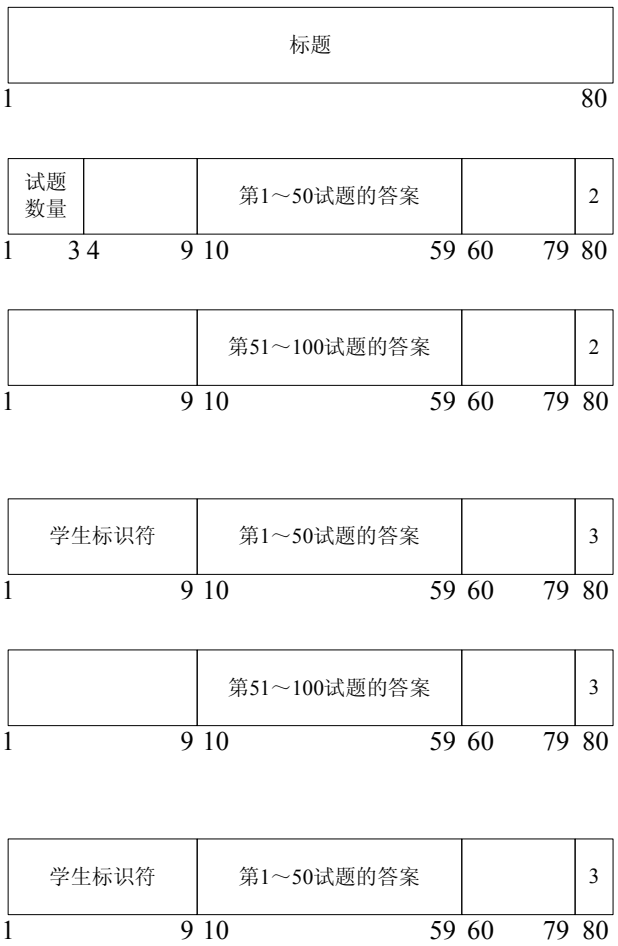


图 4-4 MTEST 程序的输入

我们从仔细阅读规格说明开始，寻找输入条件。第一个边界输入条件是一个空输入文件。第二个输入条件是该标题记录，边界条件是标题记录不存在、可能的最短标题和最长标题。后面的输入条件是存储标准答案的记录，以及第一个标准答案记录里的“试题数量”域是否存在。试题数量的等价类不应是 1～999，因为在每个 50 的倍数处会出现某些特殊情况（例如，需要多个记录）。这种输入等价类的一个合理划分是 1～50、51～999。因此，我们需要针对试题数量为 0、1、50、51 和 999 的情况设计测试用例。这样就覆盖了标准答案记录数量的大多数边界条件。然而，三个最令人感兴趣的输入条件是标准答案记录不存在、记录多了一个以及记录少了一个（例如，试题数量是 60 个，然而在某个情况下有三个标准答案记录，而在另一种情况下只有一个）。到目前为止，我们生成的测试用例有：

1.输入文件为空。

2.没有标题记录。

未经同意，严禁以任何形式拷贝

3.标题只有 1 个字符。

4.标题有 80 个字符。

5.考试试题数量为 1。

6.考试试题数量为 50。

7.考试试题数量为 51。

8.考试试题数量为 999。

9.考试试题数量为 0。

10.试题数量域的值为非数字类型。

11.标题记录后无标准答案记录。

12.标准答案记录数量多一个。

13.标准答案记录数量少一个。

下面的输入条件是有关学生的答案的，其边界值测试用例可以是：

14.学生人数为 0 。

15.学生人数为 1。

16.学生人数为 200。

17.学生人数为 201。

18.某个学生只有一条答案记录，但却存在两条标准答案记录。

19.上面那个学生是文件中第一个学生。

20.上面那个学生是文件中的最后一个学生。

21.某个学生有两条答案记录，但只有一条标准答案记录。

22.上面那个学生是文件中第一个学生。

23.上面那个学生是文件中最后一个学生。

尽管有些输出边界（例如第一份输出报告为空）已被已有的测试用例覆盖到，但我们仍然可以通过检查输出边界而得到有用的测试用例集。第一份输出报告与第二份输出报告的边界条件是：

学生人数为 0（同第 14 号测试样例）。

学生人数为 1（同第 15 号测试样例）。

学生人数为 200（同第 16 号测试样例）。

24.所有学生的成绩相同。

未经同意，严禁以任何形式拷贝

25.所有学生的成绩都不相同。

26.部分、但不是全部学生的成绩相间（检查名次的计算是否正确）。

27.某个学生的成绩为 0 。

28.某个学生的成绩为 100 。

29.某个学生的标识符值为可能的最低值（检查排序）。

30.某个学生的标识符值为可能的最高值。

31.学生的数量恰好够一份报告占满一页（检查是否打印出多余页）。

32.学生的数量除够一份报告占满一页外，还多一个。

第三份输出报告（平均值、中间值和标准偏差）的边界条件是；

33.平均值为其最大值（全部学生都得满分）。

34.平均值为 0（全部学生都得 0 分）。

35.标准偏差为其最大值（一个学生成绩为 0 分，其他都为 100 分）。

36.标准偏差为 0（全部学生成绩相同）。

第 33 和第 34 号测试用例同时也覆盖了中间值的边界条件。另外一个有用的测试用例是学生人数为 0 的情况（检查程序在计算平均值时是否有被 0 除的情况），只是这种情况与第 14 号测试用例相同。

对第四份输出报告的检查可以生成下列边界值测试用例：

37.全部学生都回答正确第一道试题。

38.全部学生都回答错误第一道试题。

39.全部学生都回答正确最后一道试题。

40.全部学生都回答错误最后一道试题。

41.试题的数量恰好够一份报告占满一页.

42.试题的数量除够一份报告占满一页外，还多一道。

有经验的程序员很可能会认同这一点，即 42 个测试用例的大部分代表了在开发该程序的过程中可能会犯的共性错误，但如果我们采用的是随机生成或特殊的测试用例设计方法，这些错误中的大多数都不会被检查出来。如果使用得正确，边界值分析是最为有效的测试用例设计方法之一。然而，这种方法常常使用得不好，因为表面上它听起来比较简单。我们应该认识到，边界条件可能非常微妙，因此把它们确定下来需要煞费一番脑筋。

未经同意，严禁以任何形式拷贝

4.1.5 因果图(Cause-Effect Graphing)

边界值分析和等价划分的一个弱点是未对输入条件的组合进行分析。举例来说，上一节中介绍的 MTEST 程序由于试题数量与学生数量的乘积超过某个阈值时可能会发生失效（例如，程序耗尽了内存）。边界值测试不一定能检查出此类错误。

对输入组合进行测试并不是简单的事情，因为即使对输入条件进行了等价划分，这些组合的数量也是个天文数字。如果在选择输入条件的子集时没有采用一个系统的方法，很可能选择出一个任意的输入条件子集，这样会使测试没有什么成效。

因果图有助于用一个系统的方法选择出高效的测试用例集。它还有一个额外的好处，就是可以指出规格说明的不完整性和不明确之处。

因果图是一种形式语言，用自然语言描述的规格说明可以转换为因果图。因果图实际上是一种数字逻辑电路（一个组合的逻辑网络）、但没有使用标准的电子学符号，而是使用了稍微简单点的符号。除了了解布尔逻辑（了解逻辑运算符“与”、“或”、“非”）之外，读者不必掌握电子学方面的知识。

生成测试用例时采用的过程如下：

1. 将规格说明分解为可执行的片段。这是必须的步骤，因为因果图不善于处理较大的规格说明。举例来说，当测试一个电子商务系统时，“可执行的片段”可能是指对挑选和确认购物车中的单件商品的规格说明。在测试一个 Web 页面设计时，我们可能会测试一个单独的菜单树，甚至是一个不太复杂的导航序列。
2. 确定规格说明中的因果关系。所谓“因”，是指一个明确的输入条件或输入条件的等价类。所谓“果”，是指一个输出条件或系统转换（输入对程序或系统状态的延续影响）。举例来说，如果某个事务引起文件或数据库记录被修改，那么这种改变就是一个系统转换，而系统反馈的确认信息就是一个输出条件。通过逐字逐句地阅读规格说明，同时标识出描述“因”和“果”的文字或句子，就可以将“因”和“果”确定出来。因果关系一旦确定下来，每个“因”和“果”都被赋予一个惟一的编号。
3. 分析规格说明的语义内容，并将其转换为连接因果关系的布尔图。这就是

所谓的因果图。

4. 给图加上注解符号，说明由于语法或环境的限制而不能联系起来的“因”和“果”。
5. 通过仔细地跟踪图中的状态变化情况，将因果图转换成一个有限项的判定表。表中的每一列代表一个测试用例。
6. 将判定表中的列转换成测试用例。

因果图中的基本符号如图 4-5 所示。设想一下，每个结点的值为 0 或为 1，0 代表“不存在”状态，1 代表“存在”状态。identity 函数表示如果 a 等于 1，则 b 也为 1，否则 b 为 0。not 函数表示如果 a 等于 1，则 b 为 0，否则 b 为 1。or 函数表示如果 a 或 b 或 c 等于 1，则 d 为 1，否则 d 为 0。and 函数表示如果 a 和 b 都等于 1，则 c 为 1，否则 c 为 0。后两个函数（or 和 and）允许存在任意数量的输入。

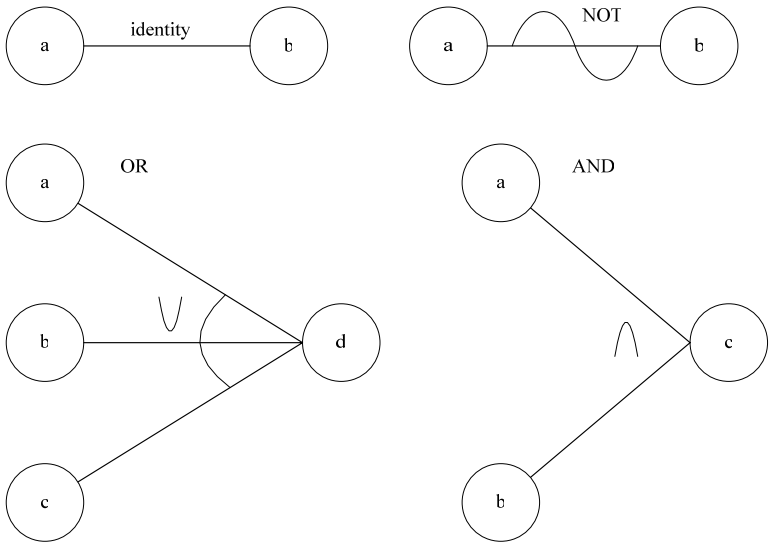


图 4-5 基本的因果图符号

为描述一个小的因果图，考虑下面的规格说明：

第一列中的字符必须是“A”或“B”，第二列中的字符必须是一个数字。在这种情况下，对文件进行更新。如果第一个字符不正确，产生提示信息 X12。如果第二个字符不是数字，产生提示信息 X13 。

“因”如下：

1——第一列的字符是“A”

2——第一列的字符是“B”

3——第二列的字符是一个数字

“果”如下：

70——对文件做了更新

71——产生提示信息 X12

72——产生提示信息 X13

因果图如图 4-6 所示。请注意生成的中间结点 11。通过设置“因”的全部可能状态，观察“果”得到了正确的值，就可以确认该图代表了规格说明。对于熟悉逻辑图的读者来说，图 4-7 是一个等价的逻辑电路。

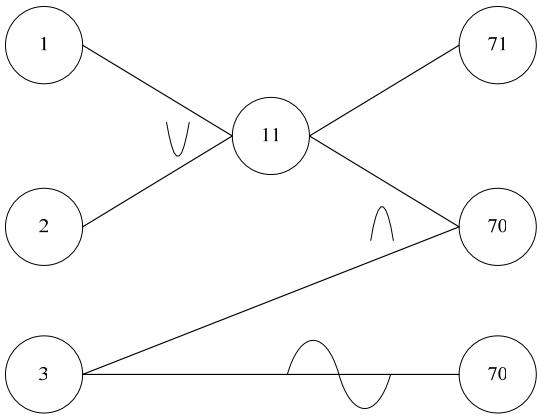


图 4-6 因果图范例

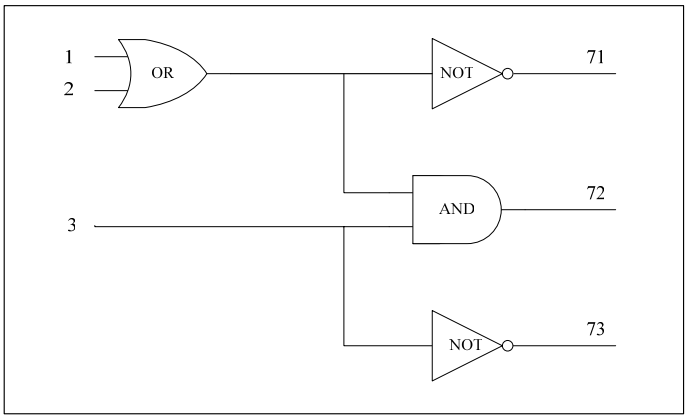


图 4-7 与图 4-6 等价的逻辑图

尽管图 4-6 所示的因果图代表了规格说明，但图中包含了一个不可能的原因组合，即原因 1 和原因 2 不可能同时设置为 1。在大多数程序中，由于语法或环境的原因，某些原因的组合是不可能存在的（一个字符不能同时为“A”和“B”）。为了对此做出解释，我们采用图 4-8 所示的符号。约束 E 表示其必须总为真，而 a 和 b 最多只有一个为 1(a 与 b 不能同时为 1)。约束 I 表示其为真时，a、b、c 中至少有一个应为 1（a、b、c 不能同时为 0)。约束 0 表示 a、b 中有且仅有一个必须为 1,约束 R 表示如果 a 为 1，b 也必须为 1(例如，a 为 1 而 b 为 0 的情况是不可能的)。

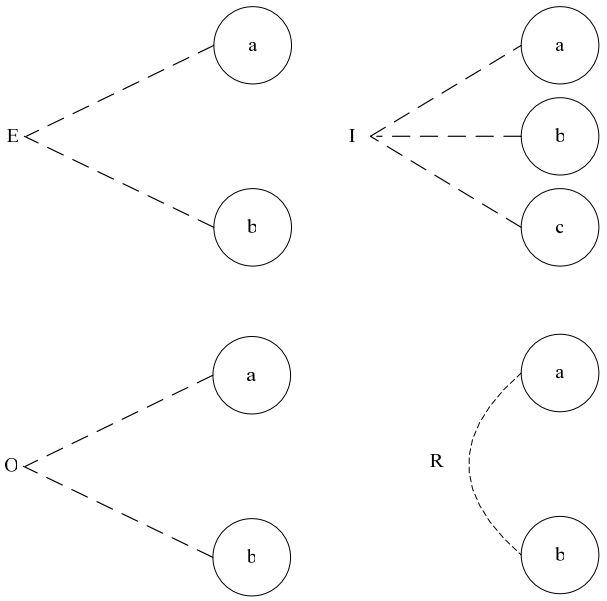


图 4-8 约束符号

在结果之间通常需要建立约束关系。图 4-9 中的约束 M 表示，如果结果 a 为 0,则 b 强制为 0。

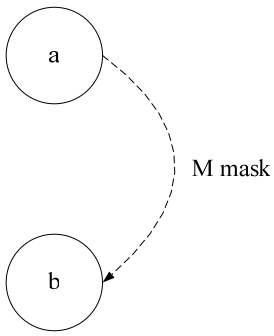


图 4-9 屏蔽约束的符号

再回到前面那个简单例子中来，我们看到，原因 1 和原因 2 实际上是不可能同时成立的，而两者都不成立却是可能的。因此，它们之间应该用约束 E 来连接，如图 4-10 所示。

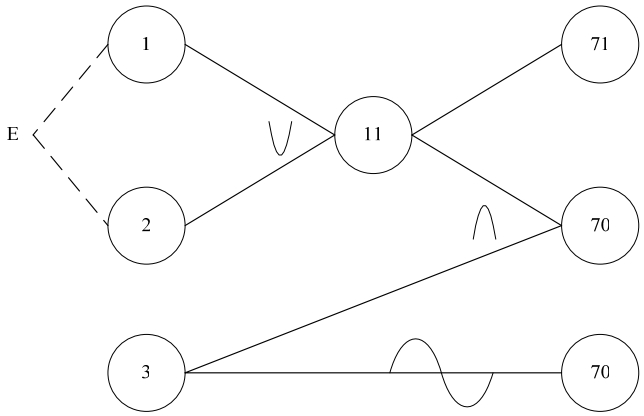


图 4-10 带有“排斥性”约束条件的因果图范例

为了说明如何从因果图中导出测试用例，需要使用下面介绍的规格说明。该规格说明用于某个交互系统的一条调试命令。

DISPLAY 命令用于从一个终端窗口中观察内存空间的内容。该命令的语法见图 4-11。括弧表示可替换的可选操作时象。大写字母表示操作对象的关健字; 小写字母表示操作时象的值（即要被取代的实际值）。带下划线的操作对象代表默认值（即操作对象默认时所使用的值）。

第一个操作对象（hexloc1）规定了待显示的内容的首字节地址。该地址可以是 1～6 位长度的十六进制（0～9，A～F）数。如果地址没有指定，默认地址为 0。地址必须在机器实际内存地址的范围之内。

第二个操作对象规定了要显示的内存的数量。如果规定了 hexloc2 的值，也就确定了要显示的内存空间范围内的末字节地址。该地址可以是 1～6 位长度的十六进制数，且必须大于或等于起始地址（hexloc1）。同时，hexloc2 也必须在机器实际内存地址的范围之内。如果定义了 END，那么从起始位置(hexloc1)直到机器内存中最后字节的内容都将显示出来。如果规定了 bytecount 的值，也就规定了要显示的内存字节数量（从 hexloc1 指定的位置开始计算），该操作对象是

一个十六进制整数（长度为 1～6 位）。bytecount 与 hexloc1 之和不能超过实际的内存容量加 1，而 bytecount 的值至少为 1。

当显示内存内容时，在屏幕上按如下格式分一行或多行输出:

xxxxxx = word1 word2 word3 word4

xxxxxx 是以十六进制表示的 word1 的地址。无论 hexloc1 为何值，或者要显示的内存容量是多大，总是显示整数个字（一个字由 4 个字节排列组成，字中首字节的地址是 4 的倍数）。每一输出行总是包含 4 个字（16 个字节）。被显示范围的首字节包含在第一个字中。

（规格说明结束）

可能产生的错误信息如下：

M1 无效的命令语法。

M2 所需内存超出了实际的内存范围。

M3 所需内存为 0 或为负数。

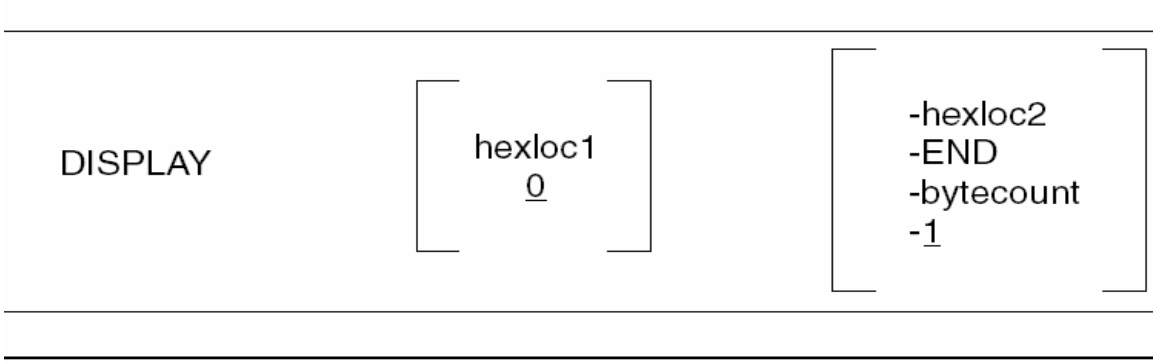


图 4-11 DISPLAY 命令的语法

例如:

DISPLAY

显示内存的前四个字（默认的起始位置为 0，默认的字节数为 1）。

DISPLAY 77F

显示首字节地址为 77F 的字以及后续的 3 个字。

DISPLAY 77F-407A

显示从字节 77F-字节 407A 间的字。

未经同意，严禁以任何形式拷贝

DISPLAY 77F.6

显示从字节 77F 起六个字节的字。

DISPLAY 50FF-END

显示从字节 50FF 开始直到内存结束的字。

第一步骤是认真地分析规格说明以确定出“因”和“果”。“因”如下：

1. 存在第一个操作对象。
2. hexloc1 操作对象仅包含十六进制数字。
3. hexloc1 操作对象包含 1-6 个字符。
4. hexloc1 操作对象在机器实际的内存范围之内。
5. 第二个操作对象为 END。
6. 第二个操作对象为 hexloc2。
7. 第二个操作对象为 bytecount 。
8. 第二个操作对象默认。
9. hexloc2 操作对象仅包含十六进制数字。
10. hexloc2 操作对象包含 1-6 个字符。
11. hexloc2 操作对象在机器实际的内存范围之内。
12. hexloc2 操作对象大于或等于 hexloc1 操作对象。
13. bytecount 操作对象仅包含十六进制数字。
14. bytecount 操作对象包含 1-6 个字符。
15. bytecount + hexloc1 ＜＝ 内存容量＋1。
16. bytecount >= 1。
17. 定义的内存范围大到足够要求显示多行输出。
18. 显示内存的起始位置不在字单元的边界位置。

每个“因”都按不同的数字进行了编号。注意，其中四个“因”（第 5～第 8）是第二个操作对象所必需的，因为第二个操作对象可能是：(1) END；(2)hexloc2；（3）bytecount；(4)不存在；(5)以上情况都不是。“果”如下：

91. 显示了信息 M1。
92. 显示了信息 M2。

未经同意，严禁以任何形式拷贝

93. 显示了信息 M3。

94. 内存内容显示在一行上。

95. 内存内容显示在多行上。

96. 显示范围的首字节正好在字单元的边界位置。

97. 显示范围的首字节不在字单元的边界位置。

下一个步骤是建立因果图。“因”结点垂直排列在纸的左边，“果”结点垂直排列在纸的右边。应仔细分析规格说明所表达的意思，以便建立起“因”与“果”的连接关系（即说明在何种情况下产生何种结果）。

图 4-12 显示了因果图的最初形式。中间结点 32 表示一个语法上有效的第一个操作对象；结点 35 表示一个语法上有效的第二个操作对象；结点 36 表示一条语法上有效的命令。如果结点 36 为 1,结果 91(错误信息)就不会显示出来，反之就会显示。

未经同意，严禁以任何形式拷贝

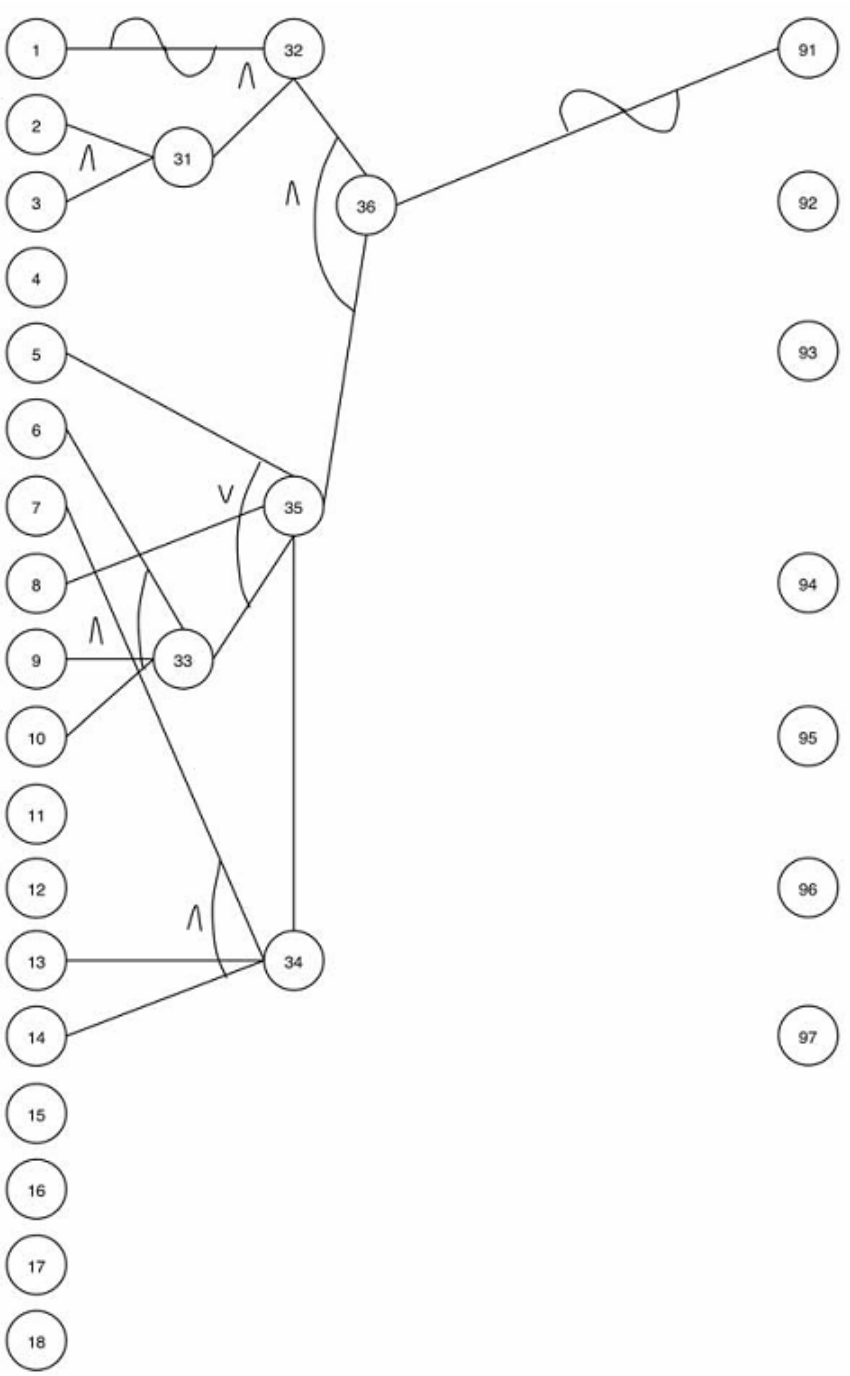


图 4-12 DISPLAY 命令的初样图

完整的因果图如图 4-13 所示。应当仔细地研究该图，以确认它如实地反映了规格说明的内容。

未经同意，严禁以任何形式拷贝

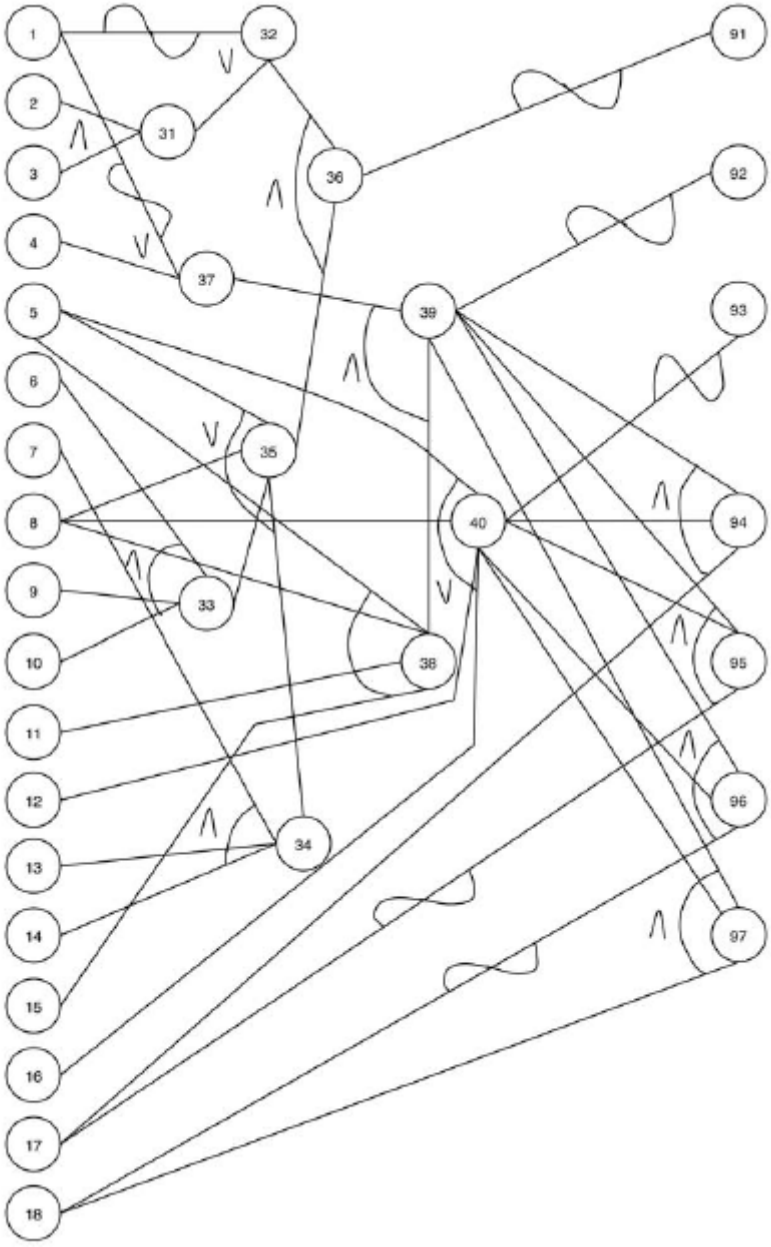


图 4 -13 不带约束条件的完整因果图

如果我们利用图 4-13 来生成测试用例，就会生成很多不可能实现的测试用例出来。原因是由于语法的制约，有些特定的原因组合是不可能的。举例来说，除非出现了原因 l，原因 2 和原因 3 就不可能出现。而原因 4 只有当原因 2 和原因 3 都出现时才成立。图 4-14 显示的就是带有约束条件的完整的因果图。请注意，原因 5、6、7、8 中最多仅能出现一个。其余所有的原因约束条件都是“要求”关系（require，即图 4-8 中的约束 R。--译者注）。注意原因 17（多行输出）要求原因 8（第二个操作对象默认）不成立；仅当原因 8 不出现时，原因 17 方才出现。再一次提醒，应

未经同意，严禁以任何形式拷贝

该仔细地检查这些约束条件。

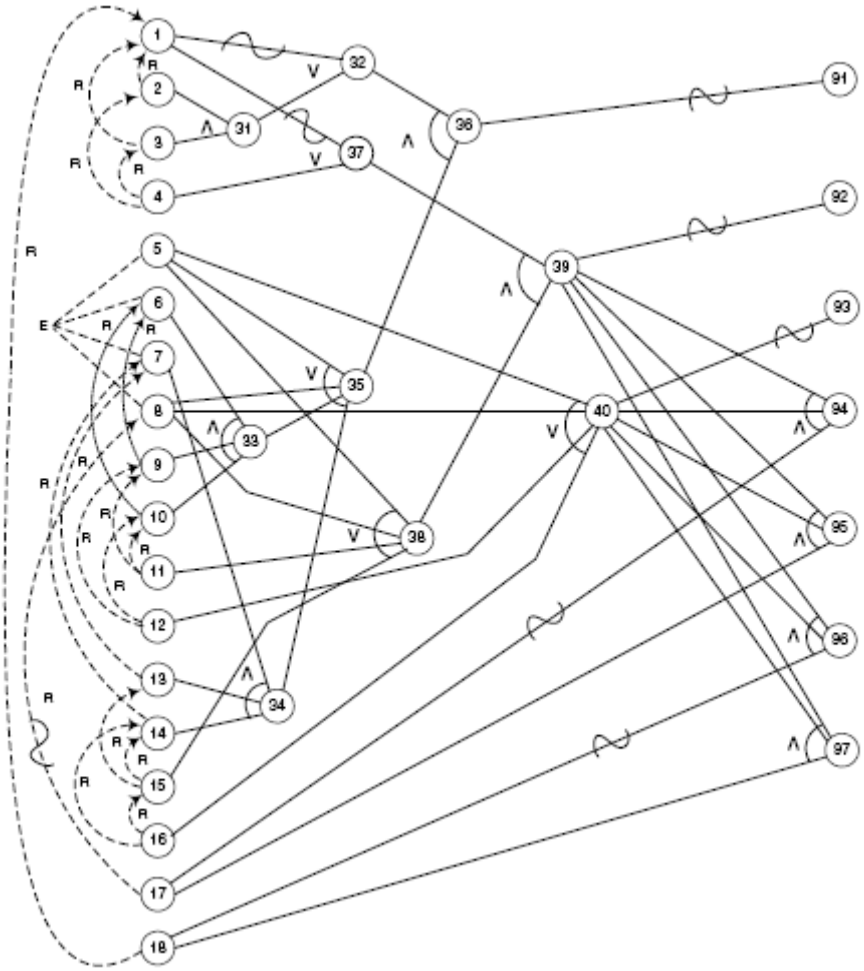


图 4-14 DISPLAY 命令的完整因果图

下一步骤是建立有限项的判定表。读者如果熟悉判定表的话，就会知道“因”是条件，而“果”是动作。采用的过程如下：

1. 选择一个“果”作为当前状态（1）。
2. 对因果图进行回溯，查找导致该“果”为 1（根据约束条件）的所有“因”的组合。
3. 在判定表中为每个“因”的组合生成一列。
4. 对于每种“因”的组合，判断所有其它“果”的状态，并放置在每一列中。

在执行第 2 步时，需要做以下考虑：

1. 当回溯经过一个结果应为 1 的 or 结点时，不要同时将该 or 结点的一个以上的输入设置为 1。这就是所谓的路径敏感性（path sensitizing），其目的是

避免由于原因之间的屏蔽而漏掉某些错误。

2. 当回溯经过一个结果应为 0 的 and 结点时，显然应列举出导致结果为 0 的所有的输入组合情况。然而，如果碰到的情况是一个输入为 0，其它的输入中有一个或更多为 1，那么就无须罗列出其它输入可能为 1 的所有情况。
3. 当回溯经过一个结果应为 0 的 and 结点时，仅有一种所有输入皆为 0 的情况需要列举出来（如果这个 and 结点位于因果图的中部，其输入来自于其他中间结点，那么所有输入都为的情况就会非常多）。

这些复杂的思路在图 4-15 中总结出来，图 4-16 是一个范例。

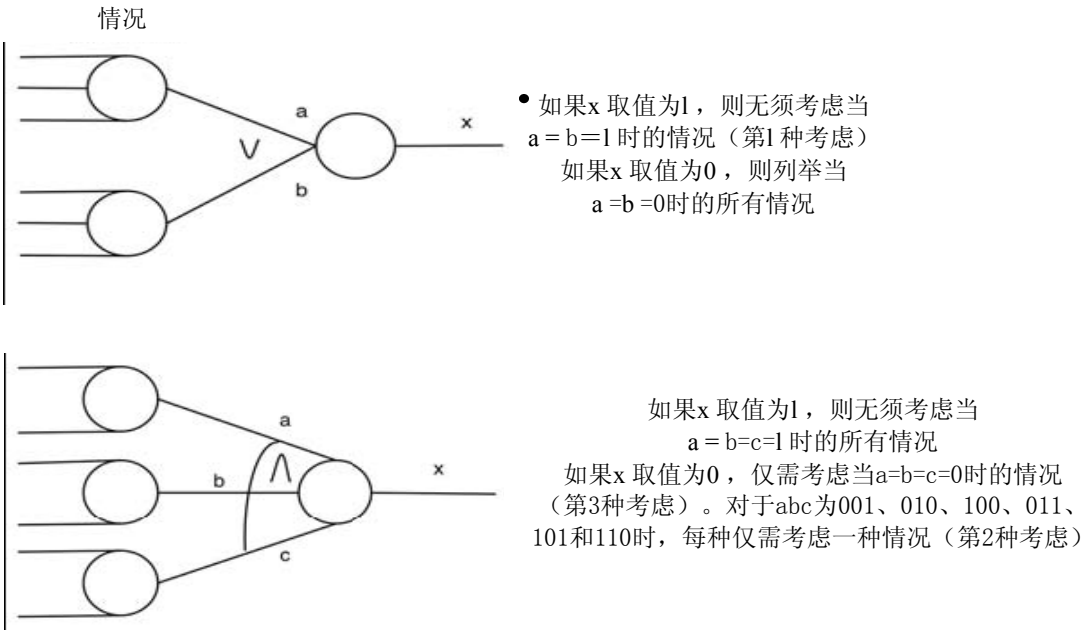


图 4-15 追溯因果图的思路

假设我们需要找出所有导致输出状态为 0 的输入条件。根据上述第 3 条思路，我们只需列出一种情况，即结点 5 和结点 6 都是 0 的情况。根据第 2 条思路，对于结点 5 为 1 而结点 6 为 0 的情况，我们只需列出结点 5 为 l 的这一种情况，而不必罗列出结点 5 可能为 l 的所有情况。同样地，对于结点 5 为 0 而结点 6 为 l 的情况，我们也只需列出结点 6 为 1 的这一种情况（尽管在本例中只有一种）。根据第 1 条思路，当结点 5 应被设置为 l 时，我们不应将结点 1 和结点 2 同时设置为 1。因此，举例来说，我们可以处于从结点 1 到结点 4 间的 5 种状态，而并非是从结点 1 到结点 4 间导致输出为 0 的 13 种可能的状态，其值如下：

未经同意，严禁以任何形式拷贝

0	0	0	0	（5=0，6=0）
1	0	0	0	（5=l，6=0）
1	0	0	1	（5=1，6=0）
1	0	1	0	（5=1，6=0）
0	0	1	1	（5=0，6=1）

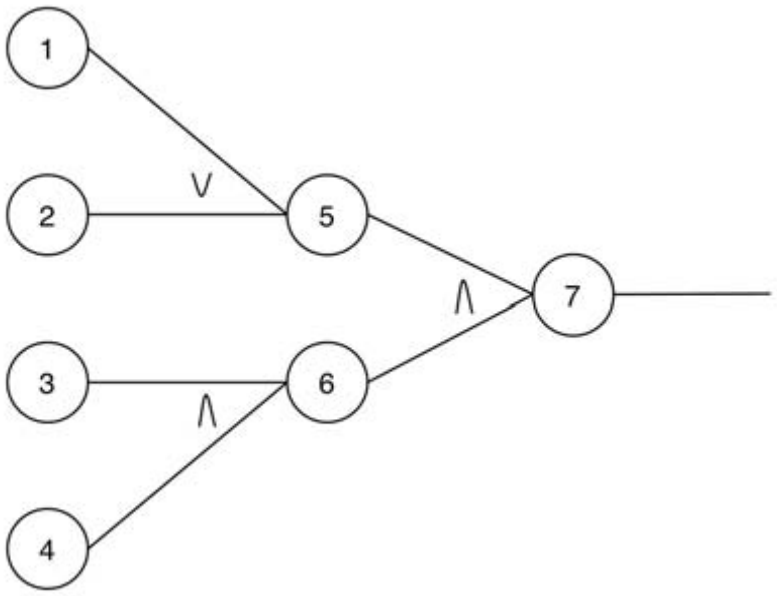


图 4-16 描述追溯思路的因果图范例

这些思路也许看起来反复无常，但都有一个重要的目标：即减少因果图中的组合关系。它们排除了会导致生成低效测试用例的状态。如果不能排除低效测试用例，那么一个因果关系复杂的大因果图会生成天文数字的测试用例。如果测试用例的数量大得不切合实际，就只得从中挑出一些子集来，而这又不能保证低效的测试用例会被排除在外。因此，最好在分析因果图的阶段就将其排除掉。

现在，可将图 4-14 所示的因果图转化为判定表。最先应选择结果 91。当结点 36 为 0 时，才会出现结果 91。当结点 32 和 35 为 0，0；0，l 或 l，0 时，结点 36 才会为 0，此处可以应用第 2 条和第 3 条思路。通过对原因的回溯以及对原因间约束关系的考虑，可以找出导致结果 91 出现的原因组合，尽管这样做是个颇费力气的过程。

针对结果 91 出现的情况，其判定表如图 4-17 所示（第 1 列～第 11 列）。第 l 列～第 3 列（也是 1～3 号测试用例）代表结点 32 为 0 而结点 35 为 l 的情况，而第 4 列～第 10 列代表结点 32 为 1 而结点 35 为 0 的情况。根据第 3 条思路，在结点 32 和 35 皆为 0 的全部 21 种情况中只需确定一种（第 11 列）。表格中的空白处表示

未经同意，严禁以任何形式拷贝

“无关紧要”的情况（即与该原因的状态并不相关），或指出由于其他相依赖原因的关系，该原因的状态是显而易见的（例如在第1 列中，我们知道由于与原因 6 存在“至多一个”的关系，原因 5、7 和 8 必定为 0 ）。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	0	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1
4												1	1	0	0	1	1
5				0										1			
6	1	1	1	0	1	1	1				1	1			1	1	
7				0				1	1	1			1				1
8				0													
9	1	1	1		1	0	0				0	1			1	1	
10	1	1	1		0	1	0				1	1			1	1	
11												0			0	1	
12																0	
13								1	0	0			1				1
14								0	1	0			1				1
15													0				
16																	0
17																	
18																	
91	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
94	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 4-17 生成的判定表的前半部分

第 12 列～第 15 列代表结果 92 出现了的情况。第 16 列～第 17 列代表结果 93 出现了的情况。图 4-18 描述了判定表的剩余部分。

未经同意，严禁以任何形式拷贝

	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1	1
2	1	1	1	1					1	1	1	1	1	1	1				1	1	1
3	1	1	1	1					1	1	1	1	1	1	1				1	1	1
4	1	1	1	1					1	1	1	1	1	1	1				1	1	1
5	1				1				1				1			1			1		
6			1				1				1			1			1			1	
7				1				1				1			1			1			1
8		1				1				1											
9			1				1				1			1			1			1	
10			1				1				1			1			1			1	
11			1				1				1			1			1			1	
12			1				1				1			1			1			1	
13				1				1				1			1			1			1
14				1				1				1			1			1			1
15				1				1				1			1			1			1
16				1				1				1			1			1			1
17	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
18	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
94	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
96	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1
97	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0

图 4-18 生成的判定表的后半部分

最后一个步骤，是将判定表转化为 38 个测试用例。下面列举了一组 38 个测试用例。每个测试用例右边的数字指出期望出现的结果。我们假定所用计算机内存的最末地址是 7FFF 。

1	DISPLAY 234AF74-123	(91)
2	DISPLAY 2ZX4-3000	(91)
3	DISPLAY HHHHHHHH-2000	(91)
4	DISPLAY 200 200	(91)
5	DISPLAY 0-22222222	(91)
6	DISPLAY 1-2X	(91)
7	DISPLAY 2-ABCDEFGHI	(91)
8	DISPLAY 3.1111111	(91)
9	DISPLAY 44.$42	(91)
10	DISPLAY 100.$$$$$$$	(91)
11	DISPLAY 10000000-M	(91)
12	DISPLAY FF-8000	(92)

未经同意，严禁以任何形式拷贝

13	DISPLAY FFF.7001	(92)
14	DISPLAY 8000-END	92)
15	DISPLAY 8000-8001	(92)
16	DISPLAY AA-A9	(93)
17	DISPLAY 7000.0	(93)
18	DISPLAY 7FF9-END	(94, 97)
19	DISPLAY 1	(94, 97)
20	DISPLAY 21-29	(94, 97)
21	DISPLAY 4021.A	(94, 97)
22	DISPLAY –END	(94, 96)
23	DISPLAY	(94, 96)
24	DISPLAY –F	(94, 96)
25	DISPLAY .E	(94, 96)
26	DISPLAY 7FF8-END	(94, 96)
27	DISPLAY 6000	(94, 96)
28	DISPLAY A0-A4	(94, 96)
29	DISPLAY 20.8	(94, 96)
30	DISPLAY 7001-END	(95, 97)
31	DISPLAY 5-15	(95, 97)
32	DISPLAY 4FF.100	(95, 97)
33	DISPLAY –END	(95, 96)
34	DISPLAY –20	(95, 96)
35	DISPLAY 11	(95, 96)
36	DISPLAY 7000-END	(95, 96)
37	DISPLAY 4-14	(95, 96)
38	DISPLAY 500.11	(95, 96)

注意，在大多数情况下，如果对同样一组原因执行两个或更多的测试用例，为改进测试用例的效率，这些原因应取不同的值。还应注意到，由于受到实际存储空间的限制，第 22 号测试用例是不能实现的（就如同第 33 号测试用例，其产生的结果是 95 而不是 94）。因此，最终确定了 37 个测试用例。

评语

因果图方法是一个根据条件的组合而生成测试用例的系统性的方法。可以替代这种方法的是特殊选取的条件组合，但在这个过程中，很可能会遗漏很多可由因果图方法确定的“令人感兴趣的”测试用例。

由于因果图方法需要将规格说明转换为一个布尔逻辑网络，因此它使我们从不同的视角，以更多的洞察力来审视规格说明。事实上，建立因果图是一个暴露规格

未经同意，严禁以任何形式拷贝

说明中模糊和不完整之处的好方法。举例来说，聪明的读者也许已经注意到，上文讨论的过程已经发现了 DISPLAY 命令规格说明中的一个问题。该规格说明规定，所有的输出行都包含 4 个字。然而，这并不是对所有的情况都成立；在测试用例 18 和 26 中就不会发生，因为其起始地址距内存最末位置不足 16 个字节。

尽管因果图方法确实能产生一组有效的测试用例，但通常它不能生成全部应该被确定的有效测试用例。举例来说，在上面的例子中，我们并未提到验证显示出来的内存值是否与内存中的实际值一致，也未提到对程序能否显示出内存空间中任何可能的值进行判断。另外，因果图方法没有充分考虑边界条件。当然，在此过程中我们可以尝试覆盖边界状态。例如，不将

hexloc2 ≥ hexloc1

确定成一个“因”，而将其确定成两个“因”：

hexloc2 = hexloc1

hexloc2 > hexloc1

然而，这样做所带来的问题是使因果图急剧复杂化，导致生成的测试用例的数量非常庞大。鉴于此，最好是单独考虑边界值分析。举例来说，可以从 DISPLAY 命令规格说明中确定出下面的边界条件：

1. *hexloc1* 为 1 位数字。
2. *hexloc1* 为六位数字。
3. *hexloc1* 为七位数字。
4. *hexloc1* = 0。
5. *hexloc1* = 7FFF。
6. *hexloc1* = 8000。
7. *hexloc2* 为一位数字。
8. *hexloc2* 为六位数字。
9. *hexloc2* 为七位数字。
10. *hexloc2* = 0。
11. *hexloc2* = 7FFF。
12. *hexloc2* = 8000。

未经同意，严禁以任何形式拷贝

13. *hexloc2* = *hexloc1*。

14. *hexloc2* = *hexloc1* + 1。

15. *hexloc2* = *hexloc1* − 1。

16. *bytecount* 为一位数字。

17. *bytecount* 为六位数字。

18. *bytecount* 为七位数字。

19. *bytecount* = 1。

20. *hexloc1* + *bytecount* = 8000。

21. *hexloc1* + *bytecount* = 8001。

22. 显示 16 个字节 (一行)。

23. 显示 17 个字节 (两行)。

注意，这并不意味着需要编写出 60 (37+23)个测试用例来。由于因果图方法给我们提供了选择操作对象具体值的灵活性，在由因果图生成测试用例时，可以将边界条件分析一并考虑进去。在上面的例子中，通过对最初 37 个测试用例的一部分进行重新编写，可以覆盖所有的 23 种边界条件，而且不必增加任何测试用例。因此，我们得到了一组虽然不多但却很有效的测试用例，并满足了两方面的目标。

注意，因果图方法是与本书第 2 章中的几个测试原则相一致的。确定每个测试用例的预期输出是因果图方法的固有部分（判定表中的每一列指明了预期的结果）。同时还应注意到，此方法鼓励我们查找未预料到的结果。举例来说，第 1 列（也是第 1 号测试用例）指出预期应出现结果 91，而不应出现结果 92 至结果 97。

此方法中最具难度的部分是将因果图转化为判定表。这个过程是有算法的，即意味着我们可以编写程序来自动完成这个过程。已经有些商业软件可以帮我们完成这一转化。

4 .2 错误猜测(Error Guessing)

常常可以看到这种情况，有些人似乎天生就是测试的能手。这些人没有用到任何特殊的方法（比如对因果图进行边界值分析），却似乎有着发现错误的诀窍。

对此的一个解释是这些人更多是在下意识中，实践着一种称为**错误猜测**的测试

用例设计技术。接到具体的程序之后，他们利用直觉和经验猜测出错的可能类型，然后编写测试用例来暴露这些错误。

由于错误猜测主要是一项依赖于直觉的非正规的过程，因此很难描述出这种方法的规程。其基本思想是列举出可能犯的错误或错误易发情况的清单，然后依据清单来编写测试用例。例如，程序的输入中出现 0 这个值就是一种错误易发情况。因此，可以编写测试用例，检查特定的输入值中有 0，或特定的输出值被强制为 0 的情况。同样，在出现输入或输出的数量不定的地方（如某个被搜索列表的条目数量）。数量为“没有”和“一个”（例如空列表，仅包含一个条目的列表）也是错误易发情况。另一个思想是，在阅读规格说明时联系程序员可能做的假设来确定测试用例（即规格说明中的一些内容会被忽略，要么是由于偶然因素，要么是程序员认为其显而易见）。

由于无法给出一个规程来，次优的选择是讨论错误猜测的实质，最好的做法是举出实例。假设在测试一个排序程序，要探讨的情况如下：

- 输入列表为空。
- 输入列表仅包含一个条目。
- 输入列表所有条目的值都相同。
- 输入列表已经排过序。

换言之，上面列举出的这些特殊情况可能在程序设计时被忽略。如果要测试的是一个二进制搜索程序，需要检查的情况包括：(1) 被搜索的表中只有一个条目；（2）表的大小是 2 的幂（如 16)；(3）表的大小是 2 的幂差 1 和 2 的幂多 1（如 15 和 17)。

想一想“边界值分析”一节中的 MTEST 程序。当使用错误猜测方法之后，我们会想到以下增加的测试：

- 程序是否接受“空白”作为答案？
- 一个第 2 类型的记录(标准答案)出现在第 3 类型的记录集中(学生答案)。
- 除了首条记录（标题）外，存在最后一列中没有“2”或“3”的记录。
- 两位学生名字或编号相同。
- 由于中间值的计算根据数据项的数量是奇数还是偶数而有所不同，因此针

未经同意，严禁以任何形式拷贝

对学生数量为奇数和偶数的情况分别对程序进行测试。

- “问题数量”域的值为负数。

对前一节中的 DISPLAY 命令，所想到的错误猜测测试如下：

- DISPLAY 100- （第二个操作对象不全）
- DISPLAY 100. （ 第二个操作对象不全）
- DISPLAY 100-10A 42 （多余的操作对象）
- DISPLAY 000-0000FF （以 0 打头）

4.3 测试策略

本章讨论的测试用例设计方法可以组合为一个整体的策略。之所以组合，原因现在已经很清楚了，每一种方法都可以提供一组具体的有用的测试用例，但是都不能单独提供一个完整的测试用例集。一组合理的策略如下：

1. 如果规格说明中包含输入条件组合的情况，应首先使用因果图分析方法。
2. 在任何情况下都应使用边界值分析方法。应记住，这是对输入和输出边界进行的分析。边界值分析可以产生一系列补充的测试条件，但是，也正如“因果图分析”一节所述，多数甚至全部条件都可以被整合到因果图分析中。
3. 应为输入和输出确定有效和无效等价类，在必要情况下对上面确认的测试用例进行补充。
4. 使用错误猜测技术增加更多的测试用例。
5. 针对上述测试用例集检查程序的逻辑结构。应使用判定覆盖、条件覆盖、判定/条件覆盖或多重条件覆盖准则（最后的一个最为完整）。如果覆盖准则未能被前四个步骤中确定的测试用例所满足，并且满足准则也并非不可能（由于程序的性质限制，某些条件的组合也许是不可能实现的），那么增加足够数量的测试用例，以使覆盖准则得到满足。

再一次声明，使用上述策略并不能保证可以发现所有的错误，但实践证明这是一个合理的折中方案。同时，它也代表了客观的艰巨工作量，虽然没人说软件测试是一件容易的事。

未经同意，严禁以任何形式拷贝

第 5 章 模块（单元）测试

到目前为止，我们在很大程度上忽视了软件测试的机制及被测程序的规模。然而，大型的软件程序（即超过 500 条语句块的程序）需要特别的测试对策。在本章中我们将探讨构建大型程序测试的第一个步骤：模块测试，而剩余的步骤将在本书第 6 章中介绍。

模块测试（或单元测试）是对程序中的单个子程序、子程序或过程进行测试的过程，也就是说，一开始并不是对整个程序进行测试，而是首先将注意力集中在对构成程序的较小模块的测试上面。这样做的动机有三个。首先，由于模块测试的注意力一开始集中在程序的较小单元上，因此它是一种管理组合的测试元素的手段。其次，模块测试减轻了调试（准确定位并纠正某个已知错误的过程）的难度，这是因为一旦某个错误被发现出来，我们就知道它在哪个具体的模块中。第三，模块测试通过为我们提供同时测试多个模块的可能，将并行工程引入软件测试中。

模块测试的目的是将模块的功能与定义模块的功能规格说明或接口规格说明进行比较。为了再次强调所有测试过程的目的，这里的测试目标不是为了说明模块符合其规格说明，而是为了揭示出模块与其规格说明存在着矛盾。在本章中，我们从以下三个方面来探讨模块测试：

1. 测试用例的设计方式。
2. 模块测试及集成的顺序。
3. 对执行模块测试的建议。

5.1 测试用例设计

在为模块测试设计测试用例时，需要使用两种类型的信息：模块的规格说明和模块的源代码。规格说明一般都规定了模块的输入和输出参数以及模块的功能。

模块测试总体上是面向白盒测试的。其中一个原因是如果对大一点的软件进行测试，例如一个完整的程序（其实是后续的测试过程所针对的对象），白盒测试不容易展开。第二个原因是，后续的测试过程着眼于发现其他类型的错误（举例来说，

未经同意，严禁以任何形式拷贝

这些错误不一定与程序的逻辑结构有关，比如程序未能满足其用户需求）。因此，模块测试的测试用例的设计过程如下：使用一种或多种白盒测试方法分析模块的逻辑结构，然后使用黑盒测试方法对照模块的规格说明以补充测试用例。

由于所需的测试用例设计方法已经在本书第 4 章中讨论过，我们在这里通过一个例子来描述这些方法在模块测试中的应用。假设我们要测试一个名为BONUS的模块。其功能是为销售额最高的部门的雇员的薪水增加$2,000，但是如果某个符合条件的雇员的当前工资已经达到或超过了$150,000，则薪水只增加$1000 。[2]

对模块的输入情况如图 5-1 中的表格所示。如果模块正确完成了其功能，返回错误代码 0。如果雇员或部门表中不存在任何条目，模块将返回错误代码 l。如果在某个符合条件的部门中未发现任何雇员，模块将返回错误代码 2。

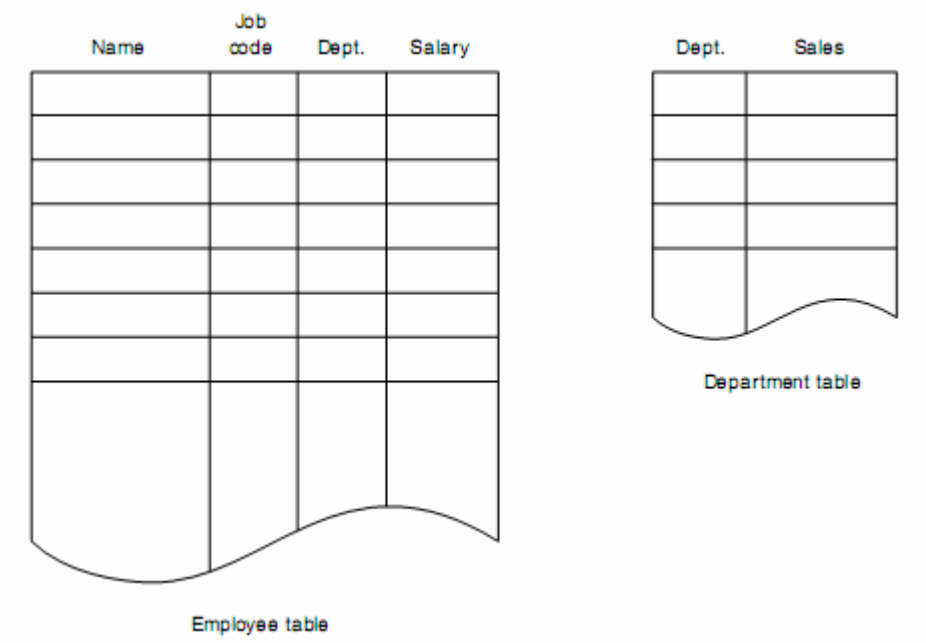


图 5-1 BONUS 模块的输入表

模块的源程序如图 5-2 所示。输入参数 ESIZE 和 DSIZE 分别代表雇员表和部门表内条目的数量。该模块是用 PL/l 语言编写的，但下面的讨论整体上是与编程语言无关的；这些技术可以用在其他语言编写的程序中。同时，由于本模块中的 PL/1 逻辑结构非常简单，实际上任何读者，甚至不熟悉 PL/l 的人也应该能够读懂程序。

[2] 这三个数据与图 5-2 源程序中的数据不一致，图是$200,$100。——译者注。

```
BONUS : PROCEDURE(EMPTAB,DEPTTAB,ESIZE,DSIZE,ERRCODE);
DECLARE 1 EMPTAB (*),
          2 NAME CHAR(6),
          2 CODE CHAR(1),
          2 DEPT CHAR(3),
          2 SALARY FIXED DECIMAL(7,2);
DECLARE 1 DEPTTAB (*),
          2 DEPT CHAR(3),
          2 SALES FIXED DECIMAL(8,2);
DECLARE (ESIZE,DSIZE) FIXED BINARY;
DECLARE ERRCODE FIXED DECIMAL(1);
DECLARE  MAXSALES  FIXED  DECIMAL(8,2)  INIT(0);  /*MAX.  SALES  IN
DEPTTAB*/
DECLARE (I,J,K) FIXED BINARY; /*COUNTERS*/
DECLARE FOUND BIT(1); /*TRUE IF ELIGIBLE DEPT. HAS EMPLOYEES*/
DECLARE SINC FIXED DECIMAL(7,2) INIT(200.00); /*STANDARD INCREMENT*/
DECLARE LINC FIXED DECIMAL(7,2) INIT(100.00); /*LOWER INCREMENT*/
DECLARE  LSALARY  FIXED  DECIMAL(7,2)  INIT(15000.00);  /*SALARY
BOUNDARY*/
DECLARE MGR CHAR(1) INIT('M');
1 ERRCODE=0;
2 IF(ESIZE<=0)|(DSIZE<=0)
3   THEN ERRCODE=1; /*EMPTAB OR DEPTTAB ARE EMPTY*/
4   ELSE DO;
5      DO I = 1 TO DSIZE; /*FIND MAXSALES AND MAXDEPTS*/
6         IF(SALES(I)>=MAXSALES) THEN MAXSALES=SALES(I);
7      END;
8      DO J = 1 TO DSIZE;
9         IF(SALES(J)=MAXSALES) /*ELIGIBLE DEPARTMENT*/
10           THEN DO;
11              FOUND='0'B;
12              DO K = 1 TO ESIZE;
13                 IF(EMPTAB.DEPT(K)=DEPTTAB.DEPT(J))
14                   THEN DO;
15                      FOUND='1'B;
16                      IF(SALARY(K)>=LSALARY)|CODE(K)=MGR)
17                        THEN SALARY(K)=SALARY(K)+LINC;
18                        ELSE SALARY(K)=SALARY(K)+SINC;
19                      END;
20                 END;
21                 IF(-FOUND) THEN ERRCODE=2;
22              END;
23        END;
24     END;
25 END;
```

图 5-2　BONUS 模块

无论采用哪种逻辑覆盖方法，第一步都是要列举出程序中所有的条件判断。该程序中需要列出的对象是所有的 IF 和 DO 语句。通过对程序进行检查，我们可以看出所有的 DO 语句都是此简单的迭代，每一个迭代的上限都等于或大于初始值（意味着每个循环体总是会至少执行一次），而且退出循环的惟一方法是通过 DO 语句。

未经同意，严禁以任何形式拷贝

因此，该程序中的 DO 语句无须特别关注，因为任何导致 DO 语句执行的测试用例最终都会使其进入两个方向的分支路径（即进入循环体和退出循环体）。因此，必须要进行分析的语句有：

```
2  IF (ESIZE<=0) | (DSIZE<=0)
6  IF (SALES(I) >= MAXSALES)
9  IF (SALES(J) = MAXSALES)
13 IF (EMPTAB.DEPT(K) = DEPTTAB.DEPT(J))
16 IF (SALARY(K) >= LSALARY) | (CODE(K) =MGR)
21 IF(-FOUND) THEN ERRCODE=2
```

得到了数量较少的判断后，我们可能会选择多重条件覆盖，但应该检查所有的逻辑覆盖准则（语句覆盖准则除外，其限制太多以至于不易于使用），看看它们的效果如何。为了满足判定覆盖准则，我们需要设计充足的测试用例，来触发上述 6 个判断中每一个的全部输出结果。触发所有判定输出所需的输入状态列举在表 5-1 中。由于有两个输出结果总会发生，故需要测试用例触发的状态只有 10 个。注意，要建立表 5-l ，必须沿程序逻辑结构对判定输出的状态进行回溯，以判别相应的正确输入状态。例如，判断 16 不会被任何符合条件的雇员触发。雇员必须处在满足条件的部门之内。

表 5-1 与判定输出对应的状态

Decision	**True Outcome**	**False Outcome**
2	ESIZE or DSIZE ≤ 0.	ESIZE and DSIZE > 0.
6	Will always occur at least once.	Order DEPTTAB so that a department with lower sales occurs after a department with higher sales.
9	Will always occur at least once	All departments do not have the same sales.
13	There is an employee in an eligible department.	There is an employee who is not in an eligible department.
16	An eligible employee is either a manager or earns LSALARY or more.	An eligible employee is not a manager and earns less than LSALARY.
21	All eligible departments contain no employees.	An eligible department contains at least one employee.

表 5-l 中关注的 10 种情况可以被图 5-3 中所示的两个测试用例触发。注意，每个测试用例都包含了对预期输出的定义，这是符合本书第 2 章讨论的测试原则的。

未经同意，严禁以任何形式拷贝

<table>
<tr><th>测试用例</th><th>输入</th><th>预期的输出</th></tr>
<tr><td>1</td><td>ESIZE = 0
All other inputs are irrelevant</td><td>ERRCODE = 1
ESIZE, DSIZE, EMPTAB, and DEPTTAB are unchanged</td></tr>
<tr><td>2</td><td>ESIZE = DSIZE = 3
EMPTAB
<table><tr><td>JONES</td><td>E</td><td>D42</td><td>21,000.00</td></tr><tr><td>SMITH</td><td>E</td><td>D32</td><td>14,000.00</td></tr><tr><td>LORIN</td><td>E</td><td>D42</td><td>10,000.00</td></tr></table>DEPTTAB
<table><tr><td>D42</td><td>10,000.00</td></tr><tr><td>D32</td><td>8,000.00</td></tr><tr><td>D95</td><td>10,000.00</td></tr></table></td><td>ERRCODE = 2
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table><tr><td>JONES</td><td>E</td><td>D42</td><td>21,100.00</td></tr><tr><td>SMITH</td><td>E</td><td>D32</td><td>14,000.00</td></tr><tr><td>LORIN</td><td>E</td><td>D42</td><td>10,200.00</td></tr></table></td></tr>
</table>

图 5-3 满足判定覆盖准则的测试用例

尽管这两个测试用例都满足判定覆盖准则，但很明显，该模块中仍可能存在着很多类型的错误不能通过这两个用例来发现。举例来说，这两个用例没有对错误代码为 0、某名雇员是管理人员或部门表为空（DSIZE<=0）时的情况进行检查。

使用条件覆盖准则可以获得更为满意的测试效果。因此我们需要设计出足够的测试用例，来触发判断中每个条件的所有输出结果。触发所有输出结果的条件以及所需的输入情况列在表 5-2 中。由于两个输出结果总会出现，需要测试用例强制触发的状态有 14 个。这些状态同样可以仅被两个测试用例触发，如图 5-4 所示。

<table>
<tr><th>测试用例</th><th>输入</th><th>预期的输出</th></tr>
<tr><td>1</td><td>ESIZE = DSIZE = 0
All other inputs are irrelevant</td><td>ERRCODE = 1
ESIZE, DSIZE, EMPTAB, and DEPTTAB are unchanged</td></tr>
<tr><td>2</td><td>ESIZE = DSIZE = 3
EMPTAB
<table><tr><td>JONES</td><td>E</td><td>D42</td><td>21,000.00</td></tr><tr><td>SMITH</td><td>E</td><td>D32</td><td>14,000.00</td></tr><tr><td>LORIN</td><td>E</td><td>D42</td><td>10,000.00</td></tr></table>DEPTTAB
<table><tr><td>D42</td><td>10,000.00</td></tr><tr><td>D32</td><td>8,000.00</td></tr><tr><td>D95</td><td>10,000.00</td></tr></table></td><td>ERRCODE = 2
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table><tr><td>JONES</td><td>E</td><td>D42</td><td>21,100.00</td></tr><tr><td>SMITH</td><td>E</td><td>D32</td><td>14,000.00</td></tr><tr><td>LORIN</td><td>E</td><td>D42</td><td>10,200.00</td></tr></table></td></tr>
</table>

图 5-4 满足条件覆盖准则的测试用例

图 5-4 所示的测试用例是用来说明某个问题的。由于他们的确可以触发表 5-2 中列举的所有输出结果，因此它们满足条件覆盖准则，但是从满足判定覆盖准则的

未经同意，严禁以任何形式拷贝

效果来看，它们要比图 5-3 中的测试用例集差一些。原因是它们没有执行到每条语句。举例来说，语句 18 就从没有被执行过。而且，它们也不比图 5-3 中的测试用例更全面，它们没有触发输出 ERRORCODE=0。如果语句 2 错误地编写成（ESIZE=0）并且（DSIZE=0），这个错误就检查不出来。当然，其他的测试用例集也可能会解决这个问题，但是图 5-4 中的两个测试用例确实可以满足条件覆盖准则，这个事实是存在的。

使用判断/条件覆盖准则可以克服图 5-4 中的测试用例存在的明显缺陷。这里，我们会提供足够多的测试用例，使得所有条件和判断的全部输出结果都至少被触发一次。让 Jones 当管理人员，Lorin 当非管理人员，就可以实现这一点。这样做的结果是，将产生判断 16 的两个输出结果，从而使语句 18 得到执行。

然而，这样做存在一个问题，从根本上讲它并不比图 5-3 中的测试用例更完善。如果我们使用的编译器一旦判断出“或”表达式中某个操作对象结果为真，就停止检查该表达式，那么就会导致语句 16 中 CODE(K)=MGR 表达式永远得不到为真的结果。因此，如果该表达式编码不正确，这些测试用例无法发现此错误。

我们要讨论的最后一个准则式多重条件覆盖准则。这个准则要求设计出足够多的测试用例，以便将每个判断中的所有可能的条件组合至少触发一次。这项工作可以从表 5-2 开始。判断 6、9、13 和 21 每个都有两种组合；而判断 2 和 16 每个都有 4 种组合。设计测试用例的方法是挑选出一个测试用例覆盖到尽可能多的组合情况，然后再挑选出一个测试用例覆盖到尽可能多的剩余组合情况，以此类推。满足多重条件覆盖准则的测试用例集如图 5-5 所示。这个集合要比前面的测试用例集全面得多，也就意味着，我们应该开始就选择多重条件覆盖准则。

表 5-2 与条件输出对应的状态

Decision	*Condition*	*True Outcome*	*False Outcome*
2	ESIZE ≤ 0	ESIZE ≤ 0	ESIZE > 0
2	DSIZE ≤ 0	DSIZE ≤ 0	DSIZE > 0
6	SALES (I) ≥MAXSALES	Will always occur at least once.	Order DEPTTAB so that a department With lower sales occurs after a department with higher sales.
9	SALES (J) = MAXSALES	Will always occur at least	All departments do not have the same

未经同意，严禁以任何形式拷贝

		once.	sales.
13	EMPTAB.DEPT(K) = DEPTTAB.DEPT(J)	There is an employee	There is an employee who in an eligible is not in an department. Eligible department.
16	SALARY (K) ≥LSALARY	An eligible employee earns LSALARY or more.	An eligible employee earns less than LSALARY.
16	CODE (K) = MGR	An eligible employee is a manager.	An eligible employee is not a manager.
21	-FOUND	An eligible department contains no employees.	An eligible department contains at least one employee.

<table>
<tr><th>测试用例</th><th>输入</th><th>预期的输出</th></tr>
<tr><td>1</td><td>ESIZE = 0 DSIZE = 0
All other inputs are irrelevant</td><td>ERRCODE = 1
ESIZE, DSIZE, EMPTAB, and DEPTTAB are unchanged</td></tr>
<tr><td>2</td><td>ESIZE = 0 DSIZE > 0
All other inputs are irrelevant</td><td>Same as above</td></tr>
<tr><td>3</td><td>ESIZE > 0 DSIZE = 0
All other inputs are irrelevant</td><td>Same as above</td></tr>
<tr><td>4</td><td>ESIZE =5 DSIZE = 4
EMPTAB
<table>
<tr><td>JONES</td><td>M</td><td>D42</td><td>21,000.00</td></tr>
<tr><td>WARNS</td><td>M</td><td>D95</td><td>12,000.00</td></tr>
<tr><td>LORIN</td><td>E</td><td>D42</td><td>10,000.00</td></tr>
<tr><td>TOY</td><td>E</td><td>D95</td><td>16,000.00</td></tr>
<tr><td>SMITH</td><td>E</td><td>D32</td><td>14,000.00</td></tr>
</table>
DEPTTAB
<table>
<tr><td>D42</td><td>10,000.00</td></tr>
<tr><td>D32</td><td>8,000.00</td></tr>
<tr><td>D95</td><td>10,000.00</td></tr>
<tr><td>D44</td><td>10,000.00</td></tr>
</table></td><td>ERRCODE = 2
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table>
<tr><td>JONES</td><td>M</td><td>D42</td><td>21,100.00</td></tr>
<tr><td>WARNS</td><td>M</td><td>D95</td><td>12,100.00</td></tr>
<tr><td>LORIN</td><td>E</td><td>D42</td><td>10,200.00</td></tr>
<tr><td>TOY</td><td>E</td><td>D95</td><td>16,100.00</td></tr>
<tr><td>SMITH</td><td>E</td><td>D32</td><td>14,000.00</td></tr>
</table></td></tr>
</table>

图 5-5 满足多重条件覆盖准则的测试用例

BONUS 模块可能存在着大量的错误，即使是满足多重条件覆盖准则的测试用例也检查不出来，认识到这一点很重要。举例来说，没有测试用例会触发 ERRORCODE 返回值为 0 的情况，因此，如果语句 1 漏掉了，该错误就发现不到。如果 LSALARY 被错误地初始化为$150,000.01，这个错误也将无法发现。如果语句 16 中写的是 SALARY(K)> LSALARY 而不是 SALARY(K)>= LSALARY，这个错误

也将无法发现。同样，各种 off-by-one 错误（例如没有正确地处理 DEPTTAB 或 EMPTAB 表中最末的条目）能否检查出来，在很大程度上全凭运气。

现在有两个观点清晰了：多重条件覆盖准则优先于其它准则；任何逻辑覆盖准则尚不足以胜任作为生成模块测试用例的惟一手段。因此，下一个步骤就是用一组黑盒测试用例来补充图 5-5 中的测试用例。要做到这一点，我们将 BONUS 模块的接口规格说明列举在下面：

PL/1 模块 BONUS 接收 5 个参数，分布是 EMPTAB，DEPTTAB、ESIZE、DSIZE 和 ERRORCODE。这些参数的属性如下：

```
DECLARE 1 EMPTAB(*), /*INPUT AND OUTPUT*/
    2 NAME CHARACTER(6),
    2 CODE CHARACTER(1),
    2 DEPT CHARACTER(3),
    2 SALARY FIXED DECIMAL(7,2);
DECLARE 1 DEPTTAB(*), /*INPUT*/
    2 DEPT CHARACTER(3),
    2 SALES FIXED DECIMAL(8,2);
DECLARE (ESIZE, DSIZE) FIXED BINARY; /*INPUT*/
DECLARE ERRCODE FIXED DECIMAL(1); /*OUTPUT*/
```

模块假设传递的参数具有以上属性。ESIZE，DSIZE 分别表示 EMPTAB、DEPTTAB 表中的条目数量，而 EPTAB、DEPTTAB 表中的条目顺序未作任何假设。该模块的功能是为销售额（DEPTTAB、SALES）最高的一个或多个部门的雇员增加薪水（EMPTAB，SALARY）。如果某个符合条件的雇员当前工资已经达到或超过了$150,000，或者该雇员为管理人员（EMPTAB.CODE= ‘M’），则薪水只增加$1,000，否则增加$2,000。模块假设增加的薪水放入 EMPTAB.SALARY 中。如果 ESIZE、DSIZE 不大于 0，则将 ERRCODE 设置为 1，并且不进行任何操作。在所有其他情况下，完整地执行模块的功能。然而，如果某个具有最大销售额的部门没有雇员，程序继续执行，但将 ERRCODE 设置为 2；否则设置为 0。

上述规格说明并不适合于因果图分析方法（没有一组应检查其组合情况的能力分辨出来的输入条件），因此采用边界值分析方法。确定的输入边界如下：

1. EMPTAB 中条目数量为 1。
2. EMPTAB 中条目数量为最大值（65 535）。
3. EMPTAB 中条目数量为 0。
4. DMPTAB 中条目数量为 1。

未经同意，严禁以任何形式拷贝

5. DMPTAB 中条目数量为最大值（65 535）。
6. DMPTAB 中条目数量为 0。
7. 某个销售额最高的部门仅有 1 名雇员。
8. 某个销售额最高的部门有 65535 名雇员。
9. 某个销售额最高的部门没有雇员。
10. DEPTTAB 中所有部门的销售额相同。
11. 销售额最高的部门为 DEPTTAB 的第一条目。
12. 销售额最高的部门为 DEPTTAB 的最末条目。
13. 某个符合条件的雇员为 EMPTTAB 的第一条目。
14. 某个符合条件的雇员为 EMPTTAB 的最末条目。
15. 某个符合条件的雇员为管理人员。
16. 某个符合条件的雇员不是管理人员。
17. 某个不为管理人员、且符合条件的雇员的薪水为$149 999.99。
18. 某个不为管理人员、且符合条件的雇员的薪水为$150 000。
19. 某个不为管理人员、且符合条件的雇员的薪水为$150 000.01。

输出边界如下：

20. ERRCODE＝0。
21. ERRCODE＝1。
22. ERRCODE＝2。
23. 某个符合条件雇员增加后的薪水为$299 999.99。

使用错误猜测技术进一步确定的测试条件如下：

24. 在 DEPTTAB 中，某个没有雇员的销售额最大的部门其后跟着另一个有雇员的销售额最大的部门。

这用来判断当遇到 ERRCODE=2 的情况时，模块是否错误地终止对输入的处理。

评价一下这 24 种条件，其中条件 2、5 和 8 看起来像是不切实际的测试用例。由于它们所代表的条件不可能发生（在测试中作这种假设通常是很危险的，但在此处似乎比较安全），因此可以将它们排除掉。下一步是将剩下的 21 种条件与当前的测试用例集（图 5-5）进行比较，判断哪些边界条件尚未被覆盖到。通过比较，我

未经同意，严禁以任何形式拷贝

们发现需要为条件 1、4、7、10、14、17、18、19、20、23 和 24 设计图 5-5 中没有的测试用例。

再下一步是设计额外的测试用例以覆盖上述 11 种边界条件。一种方法是将这些条件合并到现有的测试用例中去（即对图 5-5 中的第 4 个用例进行修改），但我们不推荐这样做，因为这样做会打乱现有测试用例完整的多重条件覆盖。因此，最保险的方法是增加图 5-5 之外的测试用例。在这样做的过程中，我们的目标是使设计覆盖边界条件所需的最小数量的测试用例。图 5-6 中的三个测试用例实现了这一点。测试用例 5 覆盖了条件 7、10、14、17、18、19 和 20，测试用例 6 覆盖了条件 1、4 和 23，而测试用例 7 则覆盖了条件 24。

在这里我们有理由相信，逻辑覆盖准则或白盒测试用例、图 5-6 所示的测试用例已实现对模块 BONUS 适度的模块测试。

<table>
<tr><th>测试用例</th><th>输入</th><th>预期的输出</th></tr>
<tr><td>5</td><td>ESIZE = 3 DSIZE = 2
EMPTAB
<table><tr><td>ALLY</td><td>E</td><td>D36</td><td>14,999.99</td></tr><tr><td>BEST</td><td>E</td><td>D33</td><td>15,000.00</td></tr><tr><td>CELTO</td><td>E</td><td>D33</td><td>15,000.01</td></tr></table>DEPTTAB
<table><tr><td>D33</td><td>55,400.01</td></tr><tr><td>D36</td><td>55,400.01</td></tr></table></td><td>ERRCODE = 0
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table><tr><td>ALLY</td><td>E</td><td>D36</td><td>15,199.99</td></tr><tr><td>BEST</td><td>E</td><td>D33</td><td>15,100.00</td></tr><tr><td>CELTO</td><td>E</td><td>D33</td><td>15,100.01</td></tr></table></td></tr>
<tr><td>4</td><td>ESIZE =1 DSIZE = 1
EMPTAB
<table><tr><td>CHIEF</td><td>M</td><td>D99</td><td>98,899.99</td></tr></table>DEPTTAB
<table><tr><td>D99</td><td>99,000.00</td></tr></table></td><td>ERRCODE = 0
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table><tr><td>CHIEF</td><td>M</td><td>D99</td><td>99,899.99</td></tr></table></td></tr>
<tr><td>4</td><td>ESIZE =2 DSIZE = 2
EMPTAB
<table><tr><td>DOLE</td><td>E</td><td>D67</td><td>10,000.00</td></tr><tr><td>WARNS</td><td>E</td><td>D22</td><td>33,333.33</td></tr></table>DEPTTAB
<table><tr><td>D66</td><td>20,000.00</td></tr><tr><td>D67</td><td>20,000.00</td></tr></table></td><td>ERRCODE = 2
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table><tr><td>DOLE</td><td>E</td><td>D67</td><td>10,000.00</td></tr><tr><td>WARNS</td><td>E</td><td>D22</td><td>33,333.33</td></tr></table></td></tr>
</table>

图 5-6　BONUS 模块补充的边界值分析测试用例

5.2 增量测试

在执行模块测试过程中,我们主要有两点考虑：第一，如何设计一个有效的测试用例集，这在上一节已经讨论过。第二，将模块组装成工作程序的方式。第二点考

未经同意，严禁以任何形式拷贝

虑很重要，因为它涉及模块测试用例编写的形式、可能用到的测试工具类型、模块编码和测试的顺序、生成测试用例的成本以及调试（定位并修复检查出的错误）的成本等。简而言之，它具有实际重要性。在这一节中，我们将讨论两类方法，增量测试和非增量测试。在下一节中，我们将探讨两种增量方法：自顶向下的和自底向上的开发或测试过程。

这里需要考虑的问题是：软件测试是否应先独立地测试每个模块，然后再将这些模块组装成完整的程序？还是先将下一步要测试的模块组装到测试完成的模块集合中，然后再进行测试？第一种方法称为非增量测试或“崩溃（big-bang ）”测试，而第二种方法称为增量测试或集成。

图 5-7 所示的程序可作为一个例子。矩形框代表程序的 6 个模块（子程序或过程），连接模块间的线条代表程序的控制层次，也就是说，模块 A 调用模块 B、C 和 D，模块 B 调用模块 E 等等。作为传统方法的作增量测试是按如下方式进行的：首先，对 6 个模块中的每一个模块进行单独的模块测试，将每个模块视为一个独立实体。根据环境（例如，是人机交互式的，还是使用批处理计算工具）和参与人数，这些模块可以同时或按次序进行测试。最后，将这些模块组装或集成（例如“连接编辑”）为完整的程序。

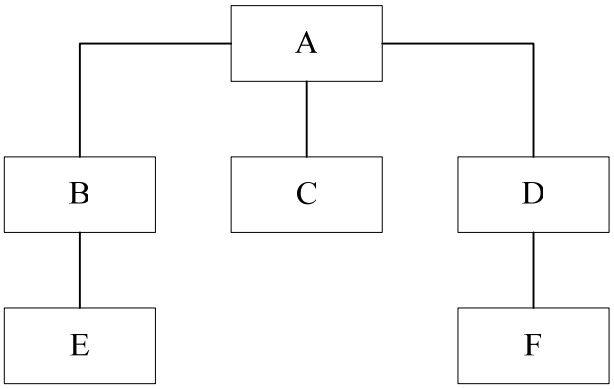


图 5-7 包含 6 个模块的程序范例

测试单独的模块需要一个特殊的**驱动模块**（drive module）和一个或多个**桩模块**（stub module）。举例来说，测试模块 B，首先要设计测试用例，然后将测试用例作

为输入参数由驱动模块传递给模块 B。驱动模块是人们编写的一个小模块，用来将测试用例驱动或传输到被测模块中（也可以用测试工具替代）。驱动模块还必须向测试人员显示模块 B 的结果。此外，由于模块 B 调用了模块 E，所以还必须使用一个额外的组件，该组件在模块 B 调用模块 E 时接受模块 B 的控制指令。这就由桩模块来完成它是一个被命名为“E”的特殊模块，用来模拟模块 E 的功能。当所有 6 个模块的模块测试都完成之后，就将这些模块组装成完整的程序。

另一种可选择的方法是增量测试。不同于独立地测试每个模块，增量测试首先将下一个要测试的模块组装到前面已经测试过的模块集合中去。

现在要给出对图 5-7 所示的程序进行增量测试的步骤还为时太早，因为还有大量可能的增量方法。一个关键问题是我们究竟是从程序的顶部开始、还是从底部开始进行测试。由于这个问题将在下一节中讨论，我们暂且假设从底部开始测试。第二步先测试模块 E、C 和 F，可以并行测试（由三个人进行），也可串行进行。请注意，我们必须要为每个模块准备一个驱动模块，但不是桩模块。下一步是测试模块 B 和 D，但不是单独地测试它们，而是分别将其与模块 E 和 F 组装在一起。换言之，要测试模块 B，应编写驱动模块和集成测试用例，将模块 B 和 E 组合起来测试．将下一个要测试的模块组装到前面已经测试过的模块集合或子集中去，这个增长的过程会一直进行到测试完最后一个模块（本例中是模块 A）为止，请注意，这个过程也可以自顶向下进行。下面是几个显而易见的结论：

1. 非增量测试所需的工作量要多一些。对于图 5-7 所示的程序，需要准备 5 个驱动模块和 5 个桩模块（假设顶部的模块不需要驱动模块）。自底向上的增量测试需要 5 个驱动模块，但不需要桩模块。自顶向下的增量测试需要 5 个桩模块，但不需要驱动模块。增量测试所需的工作量要少一些，因为使用了前面测试过的模块来取代非增量测试中所需要的驱动模块（如果从顶部开始测试）或桩模块（如果从底部开始测试）。
2. 如果使用了增量测试，可以较早地发现模块中与不匹配接口、不正确假设相关的编程错误。这是由于尽早地对模块组合进行了集成测试。然而，如果采用非增量测试，只有到了测试过程的最后阶段，模块之间才能“互相看到”。
3. 因此如果使用了增量测试，调试会进行得容易一些，我们假定存在着与模

未经同意，严禁以任何形式拷贝

块间接口或假设相关的编程错误（根据经验而来的合理假设），那么，如果使用非增量测试，直到整个程序组装之后，这些错误才会浮现出来。到了这个时候，我们就难以定位错误。因为它可能存在于程序内部的任何位置，相反，如果使用增量测试，这种类型的错误就很容易发现，因为该错误很可能与最近添加的模块有关。

4. 增量测试会将测试进行得更彻底。如果当前正在测试模块 B，要么是模块 E，要么是模块 A（取决于测试是从底部还是从顶部开始的）被当作结果而执行。虽然模块 E 或模块 A 先前已经进行了完全的测试，但将其作为 B 的模块测试结果而执行，则会诱发出一个新的情况，可能会暴露出先前测试过的模块 E 或模块 A 中存在的一个新缺陷。另一方面，如果使用的是非增量测试，对模块 B 的测试仅影响到其本身。换言之，增量测试使用先前测试过的模块，取代了非增量测试中使用的桩模块或驱动模块。因此，到最后一个模块测试完成时，实际的模块经受到了更多的检验。
5. 非增量测试所占用的机器时间显得少一些。如果使用自底向上的方法测试图 5-7 中的模块 A，在执行 A 的过程中，模块 B、C、D、E 和 F 也会执行到。而在对模块 A 的非增量测试中，仅会执行模块 B、C 和 E 的桩模块。自顶向下的增量测试的情况也是如此。如果测试的是模块 F，那么在执行模块 F 时还会执行模块 A、B、C、D 和 E；而在对模块 F 的非增量测试中，仅有模块 F 的驱动模块与其一起执行。因此，完成一次增量测试所需执行的机器指令，显然多于采用非增量测试方法所需的指令。但此消彼长的是，非增量测试要比增量测试需要更多的驱动模块和桩模块，开发这些驱动模块和桩模块是要占用机器时间的。
6. 模块测试阶段开始时，如果使用的是非增量测试，就会有更多的机会进行并行操作（也就是说，所有的模块可以同时测试）。对于大型的软件项目〔模块和人员都很多），这可能十分重要，因为在模块测试开始之时，项目的人员数量常常处于最高峰。

总的来说，第 1 条～第 4 条结论是增量测试的优点，而第 5、6 条结论是其不利之处。考虑到计算机行业当前的趋势（硬件成本已经降低而且势必会持续下去，硬件的功能不断增加，而人力劳动成本和软件错误的代价在不断增长），再考虑到

未经同意，严禁以任何形式拷贝

错误发现得越早，改正它的成本也越低，我们会看到第 1 条至第 4 条结论的重要性日益突出，而第 5 条结论越来越显得不那么重要。如果有一个缺点的话，第 6 条结论似乎确是一个薄弱的缺点。从而我们可以得出结论，增量测试要更好一些。

5.3 自顶向下测试与自底向上测试

在上一节结论的基础上，即增量测试要优于非增量测试，本节将讨论两种增量测试策略：自顶向下的测试和自底向上的测试。然而在讨论它们之前，先要澄清几个误解。

首先，“自顶向下的测试”、“自顶向下的开发”和“自顶向下的设计”常用作近义词。“自顶向下的测试”和“自顶向下的开发”确实是同义词（表示安排模块的编码和测试顺序的策略），但“自顶向下的设计”则完全不同并且是独立的概念，按自顶向下模式设计的程序既可使用自顶向下的方式，也可使用自底向上的方式进行增量测试。

其次，自底向上的测试（或自底向上的开发）常被错误地当作非增量测试。原因在于自底向上的测试的开展方式与非增量测试是相同的（即对底层或终端模块进行测试），但是就如我们从上一节看到的那样，自底向上的测试是一种增量测试。

最后，由于两种策略都属于增量测试，因此增量测试的优点在这里就不再赘述，仅讨论自顶向下测试与自底向上测试的差异。

5.3.1 自顶向下的测试

自顶向下的测试是从程序的顶部或初始模块开始。测试开始之后，挑选哪一个后续模块进行增量测试没有惟一正确的方法：惟一的原则是：要成为合乎条件的下一个模块，至少一个该模块的从属模块（调用它的模块）事先经过了测试。

我们用图 5-8 来说明这种测试策略。A 至 L 代表程序的 12 个模块。假定模块 J 包含程序的 I/O 读操作，而模块 I 包含 I/O 写操作。

第一步是测试模块 A，测试要求必须编写出代表 B、C 和 D 的桩模块。遗憾的是，我们经常会错误理解桩模块的生成。作为佐证，我们可能经常会听到这样的说

未经同意，严禁以任何形式拷贝

法，“一个桩模块仅需要写一条‘我们进行到了这一步’的信息”、“在很多情况下，模拟的桩模块仅仅只是存在而不起任何作用”。在大多数情况下，这些说法都是错误的。由于模块 A 调用模块 B，模块 A 就需要模块 B 执行一些操作，这些操作很可能就是返回给模块 A 的结果（输出参数）。如果桩模块仅仅只是返回了控制，或显示一条出错信息却没有返回一个有意义的结果，模块 A 就会发生失效，这并不是由于模块 A 存在错误，而是因为桩模块未能模拟出相应的模块。此外，桩模块仅仅返回一个“已经连通（wired-in)”的结论是不够的。举例来说，让我们考虑编写一个桩模块，代表一个平方根程序、一个数据库表搜索程序，一个“获取相关主文件记录”程序或诸如此类的程序等。如果这个桩模块仅仅返回一条固定的“已经连通”输出，却没有返回调用模块此次调用所希望的特定值，那么调用模块将会发生失效或是产生一个混乱的结果。因此，编写桩模块是很关键的。

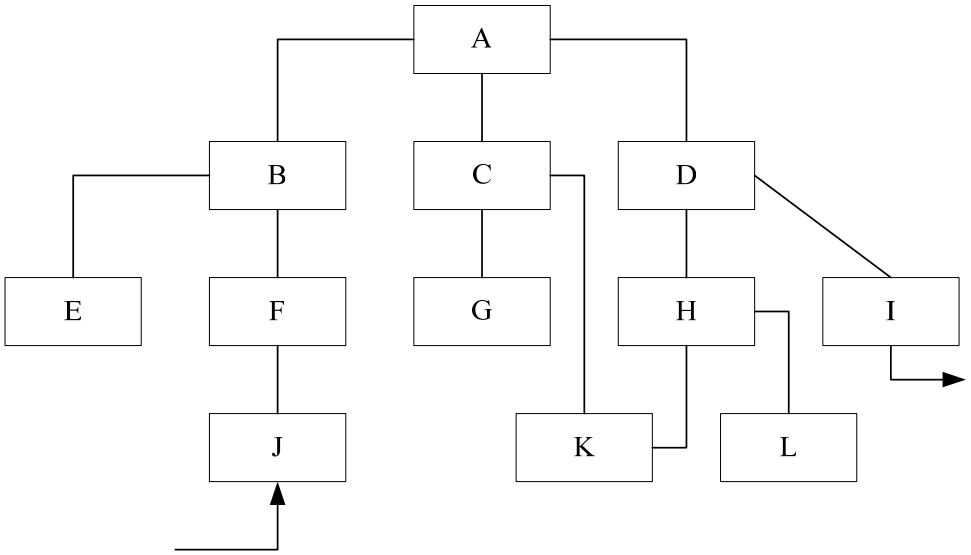


图 5-8 包含 12 个模块的程序范例

另一个需要考虑的地方是采取了什么样的形式将测试用例提交给程序，这是一个非常重要的问题，大多数对自顶向下测试的研究都没有提到这一点。在我们给出的例子中，存在这样的问题：如何向模块 A 提交测试用例？由于在典型的程序中，顶部模块既不接收输入参数，也不执行输入 / 输出操作，因此问题的答案不是显而易见的。答案是：测试数据是通过其一个或多个桩模块提交给模块（此处为模块 A）的。为了说明这一点，假设模块 B、C 和 D 的功能如下：

B － 获取事务文件的概要。

C － 判断每周的状态是否满足限额。

D － 生成每周总结报告。

那么自桩模块 B 返回的一个事务概要就是模块 A 的一个测试用例。桩模块 D 可能包含将其输入数据写到打印机的语句，这样就可以检查每一个测试的结果。

在本程序中还存在另一个问题。由于假定模块 A 仅调用模块 B 一次，问题是如何将多个测试用例提交给模块 A。一个解决方法是编写出桩模块 B 的多个版本，每一个版本都将一个各不相同的有效测试数据集返回给模块 A。为了执行这些测试用例，程序需要执行多次，每次都使用桩模块 B 的不同版本。另一种可选择的方法是将测试数据放置在外部文件中，由桩模块 B 读取并返回给模块 A。根据前面的讨论，对于任何一种情况，开发桩模块通常要比实际理解的更为困难。而且，由于程序的特点所致，通过被测模块之下的多个桩模块来传送测试数据常常是必需的（即被测模块通过调用多个桩模块来获得要处理的测试数据）。

模块 A 测试完成之后，就用一个实际的模块代替其中的一个桩模块，而该模块需要的桩模块也被添加进来。举例来说，图 5-9 就显示了该程序的下一个版本。

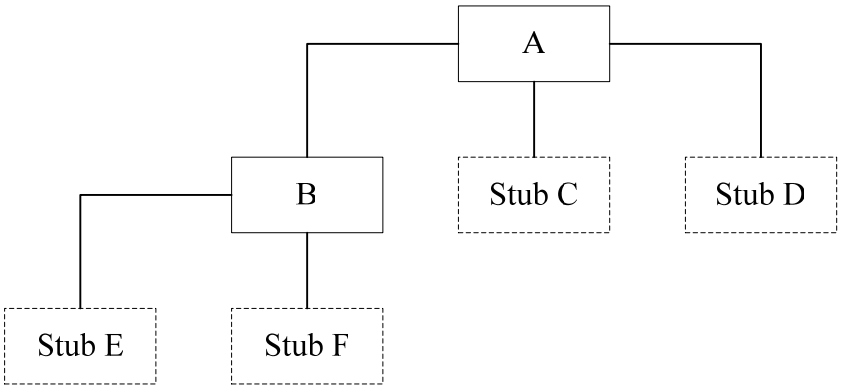


图 5－9 自顶向下测试的第二个步骤

测试完顶部模块之后，接下来可能的测试序列有很多。举例来说，如果我们要执行所有的测试序列，大量可能的模块序列中的四个序列如下：

1.	A	B	C	D	E	F	G	H	I	J	K	L
2.	A	B	E	F	J	C	G	K	D	H	L	I

3.	A	D	H	I	K	L	C	G	B	F	J	E
4.	A	B	F	J	D	I	E	C	G	K	H	L

如果可以进行并行测试，可能还有其他的选择。举例来说，模块 A 测试结束之后，一位程序员可能会选取模块 A，测试模块 A-B 的组合，另一位程序员可能会测试模块 A-C 的组合，而第三位程序员可能会测试模块 A-D 的组合。总的来说，不存在最佳的模块序列，但却有下面可供考虑的两项指南：

1. 如果程序中存在关键部分（例如模块 G），那么在设计模块序列时就应将这些关键模块尽可能早地添加进去。所谓“关键部分”可能是某个复杂的模块、某个采用新算法的模块或某个被怀疑容易发生错误的模块。
2. 在设计模块序列时，应将 I/O 模块尽可能早地添加进来。

第一项指南的动机非常清楚，但第二项指南的动机则需要进一步的讨论。回想一下，桩模块的问题就是一部分桩模块须包含测试用例，而另一部分桩模块则须将其输入写到打印机中或显示出来。然而，接收程序输入的模块一旦被添加进来，测试用例的描述就相当简单了，其采用的形式就与最终程序接收的输入一样（例如，通过事务文件或终端）。相似地，一旦执行程序输出功能的模块被添加进来，桩模块中就可能无需再放置输出测试用例结果的代码。因此，如果模块 J 和模块 I 是 I/O 模块，而模块 G 执行某些关键操作，那么增长序列可能是：

A B F J D I C G E K H L

而第 6 个增量[3]之后，程序可能是如图 5-10 所示的形式。

一旦到达了如图 5-10 所示的中间阶段，测试用例的描述以及测试结果的检查就简单化了。由于有一个程序实际运行的框架版本，也就是执行实际的输入和输出操作，就带来了另一个好处。然而，桩模块依然模拟着部分“内幕”。这个早期的程序框架版本有以下优点：

- 可以使我们发现人为因素的错误和问题。
- 可以将程序演示给最终用户看。
- 证明程序的整体设计是合理的。

[3] 即加入 I。——译者注

- 起到精神上的鼓舞作用。

然而另一方面，自顶向下策略还有一些严重缺陷。假定我们当前的测试状态如图 5-10 所示，下一步是用模块 H 取代桩模块 H。这时（或更早一些）我们所要做的是使用本章前面所述的方法，为 H 设计一个测试用例集。但是请注意，这些测试用例采用的是向模块 J 的实际程序输入的形式。这带来了一些问题。

首先，由于在模块 J 和模块 H 之间存在中间摸块（即模块 F 、B 、A 和 D），我们会发现无法将测试过模块 H 中所有预先确定的情况的测试用例提交到模块 J 中去。举例来说，如果 H 是如图 5-2 所示的 BONUS 模块，由于中间模块 D 的存在，就无法生成图 5-5 和图 5-6 中的 7 个测试用例中的部分用例。

其次，由于 H 和程序中测试数据引入点之间存在着“距离”，即使存在着测试全部状态的可能性，要决定往模块 J 中输入什么样的数据来测试到 H 中的所有状态，通常也是一项困难的脑力劳动。最后，由于一个测试显示出来的输出可能来自于一个与被测模块相距甚远的模块，要将显示出来的输出与此模块的实际执行情况联系起来非常困难，甚至是不可能的。想象一下将模块 E 添加到图 5-10 中，每个测试用例的结果都取决于检查模块 I 的输出，但是由于存在着中间模块，要推演出模块 E 的实际输出（即返回给模块 B 的数据）可能是很困难的。

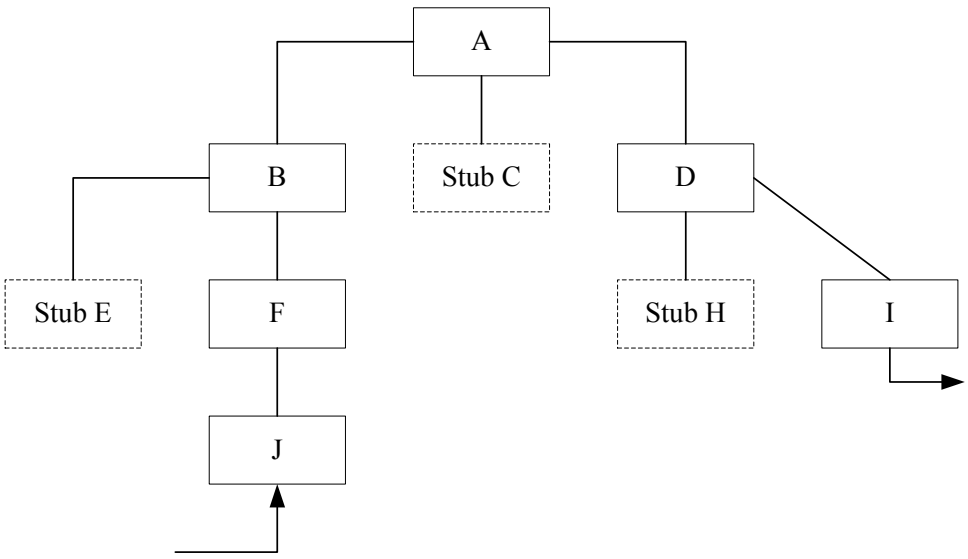


图 5-10 自顶向下测试的中间状态

自顶向下的测试策略取决于其使用的方法，可能还存在两个更深层次的问题。人们会偶尔感觉到它可能与程序的设计阶段重叠。举例来说，如果我们正在设计如图 5-8 所示的程序，可能会觉得在最先的两个层次设计完成之后，在下面层次的设计进行的同时就可以对模块 A 至模块 D 进行编码和测试了。正如我们在其他地方所强调的那样，这往往不是明智之举。程序设计是一个迭代的过程，这意味着当我们在设计程序结构的较低层次时，可能会对较高层次进行合理的变更或改进。如果程序的较高层次已经完成了编码和测试，那么这些理想的改进就会被摒弃，最终成为一个不明智的决策。

实践中时常会发生的一个终极问题是，在进行到下一个模块前未能穷举测试此模块。这来自于两个原因：一是由于将测试数据嵌入桩模块中存在困难，二是由于程序的较高层次通常会为较低层次提供资源。在图 5-8 中，我们看到，对模块 A 的测试需要用到针对模块 B 的多个版本的桩模块。在实践中，我们会倾向于说“由于这需要投人很多工作，我现在就不执行模块 A 的所有测试用例，一直等到将模块 J 添加到程序中，此时引入测试用例就容易多了，我会记得在那时完成对模块 A 的测试”。当然，这里的问题是到了那个较晚的时间点，我们可能会忘记模块 A 中剩下的测试。另外，因为较高的层次常常会提供资源给较低层次（例如打开文件）使用，有时除非到了使用资源的低层次模块测试完成之后，我们很难判断这些资源提供得是否正确（例如，文件是否以正确的属性打开）。

5.3.2 自底向上的测试

下面讨论自底向上的增量测试策略。在大多数情况下，自底向上的策略与自顶向下的策略是相对立的。自顶向下测试的优点成为自底向上测试的缺点，而自顶向下测试的缺点又成为自底向上测试的优点。正因为这一点，我们对自底向上测试的介绍就简短一些。

自底向上的策略开始于程序中的终端模块（此类模块不再调用其他任何模块）。测试完这些模块之后，同样没有最佳的方法来挑选要进行增量测试的下一个模块。惟一正确的原则是，要成为合乎条件的下一个模块，该模块所有的从属模块（它调用的模块）都已经事先经过了测试。回到图 5-8，第一步是测试模块 E、J、G、K、L 和 I 中的部分或全部模块，既可以串行进行，也可以并行进行。要做到这一点，

未经同意，严禁以任何形式拷贝

每一模块都需要一个特殊的驱动模块：即包含着有效的测试输入、调用被测模块且将输出显示出来（或将实际输出与预期输出作比较）的模块。有别于使用桩模块的情况，由于驱动模块可以交迭地调用被测模块，因此不需要为驱动模块提供多个版本。在大多数情况下，开发驱动模块要比开发桩模块更容易些。

如同前面的例子一样，影响测试序列的因素是模块的关键程度。如果我们觉得模块 D 和模块 F 最为关键，那么应该自底向上增量测试的某个中间状态可能如图 5-11 所示。接下来的步骤可能是测试模块 E，然后再测试模块 B，将模块 B 与先前测试过的模块 E、F 和 J 组装起来进行测试。

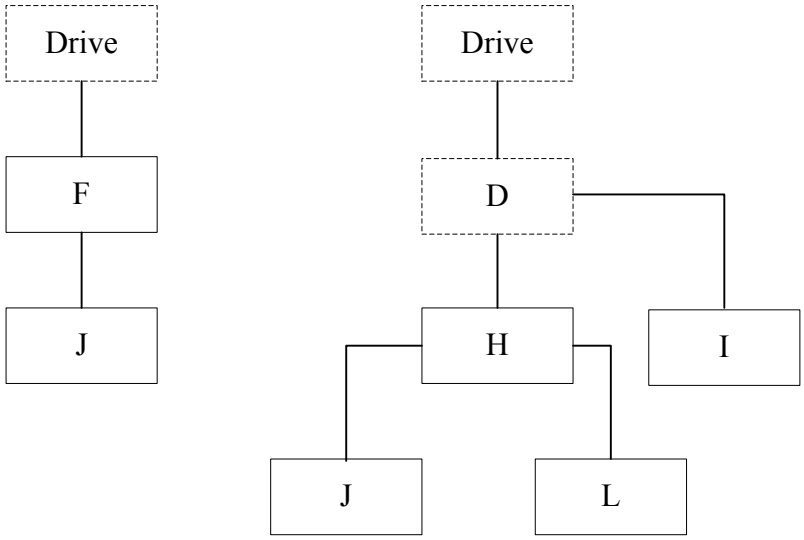


图 5-11 自底向上测试的中间状态

自底向上策略的一个不足是，它没有早期程序框架的概念。事实上，直到最后一个模块（模块 A）被添加进来，才形成了可工作的程序，也就是完整的程序。尽管 I/O 功能可以在整个程序集成之前进行测试（I/O 模块在图 5－11 中用到），早期程序框架的优点在这里体现不出来。

自顶向下方法中无法建立所有测试环境的问题，在这里都不复存在。如果将驱动模块看作是一个测试探针的话，那么该探针是直接放入被测模块中去的，不会受到中间模块的困扰。检查一下与自顶向下方法相关的其他问题，我们再也不会做出让设计和测试重叠的不明智决定，因为自底向上的测试要直到程序底层设计完成之后方才开始。同样，在没有测试完一个模块之前就开始另一个模块测试的问题也不

未经同意，严禁以任何形式拷贝

会存在，这是因为使用自底向上的测试不再有如何将测试数据绑定到桩模块中去的烦恼。

5.3.3 比较

如果自顶向下的方法和自底向上的方法，就象增量测试和非增量测试一样区别分明，那么比较起来很容易，但遗憾的是，情况并非如此。表 5-3 概括了它们之间相对的优点和不足（前面讨论过的两者皆有的优点除外，也就是增量测试的优点）。每种方法的第一个优点似乎是决定性的因素，但是也没有证据表明主要的缺陷会更容易发生在典型程序的顶部或底层。最保险的判断方法是，根据特定的被测程序，对表 5-3 中所示的各因素进行权衡。由于这里缺乏一个规程，自顶向下测试第四个缺点的严重后果，以及有可用的测试工具减少了对驱动模块而不是桩模块的需求，这样似乎给自底向上的策略带来了优势。

除此之外，自顶向下的方法和自底向上的方法很显然都不是惟一可能的增量测试策路。

表 5-3 自顶向下测试与自底向上测试的比较

自顶向下测试	
优点	**缺点**
1. 如果主要的缺陷发生程序的顶层将非常有利 2. 一旦引人 I/O 功能，提交测试用例会更容易 3. 早期的程序框架可以进行演示，并可激发积极性	1. 必须开发桩模块 2. 桩模块要比最初表现的更复杂 3. 在引入 I/O 功能之前，向桩模块中引入测试用例比较用难 4. 创建测试环境可能很难，甚至无法实现 5. 观察测试输出很困难 6. 使人误解设计和测试可以交迭进行 7. 会导致特定模块测试的完成延后
自底向上优点	
优点	**缺点**
1. 如果主要的缺陷发生在程序的底层将非常有利 2. 测试环境比较容易建立 3. 观察测试输出比较容易	1. 必须开发驱动模块 2. 直到最后一个模块添加进去，程序才形成一个整体

5.4 执行测试

按下来介绍模块测试的其他部分如何实际进行测试。这里我们给出了一系列操作的提示和指南。

未经同意，严禁以任何形式拷贝

当测试用例造成模块输出的实际结果与预期结果不匹配的情况时，存在两个可能的解释：要么该模块存在错误，要么预期的结果不正确（测试用例不正确）。为了将这种混乱降低到最小程度，应在测试执行之前对测试用例集进行审核或检查（也就是说，应对测试用例进行测试）。

使用自动化测试工具可以使测试过程中的枯燥劳动减至最小。举例来说，现在已有测试工具可以降低我们对驱动模块的需求。流程分析工具可以列举出程序中的路径、找出从未被执行的语句（“不可达”代码），以及找出变量在赋值前被使用的实例。

在准备模块测试时，重温一下本书第 2 章中讨论的心理学和经济学原则会有所裨益。如同本章前面所做的那样，记住对预期输出进行定义是测试用例必不可少的部分。在执行测试时，应该查找程序的副作用（即模块执行了某些不该执行操作的情况）。一般情况下，这些情况都是很难发现的，但如果在测试用例执行完之后，检查那些不应有变动的模块输入，可能会发现一些错误实例。举例来说，图 5-7 中的测试用例 7 声明 ESIZE、DSIZE 和 DEPTTAB 作为预期结果的一部分，不应发生变更。在执行此测试用例时，不仅要检查输出结果是否正确，还要检查 ESIZE、DSIZE 和 DEPTTAB，判断它们是否被错误地修改了。

因个人试图测试自己编写的程序所带来的心理学问题，也适用于模块测试。程序员不应测试自己编写的模块，而应交换模块进行测试。编写调用模块的程序员始终是测试被调用模块的最佳候选人。注意，这仅仅适用于测试，对模块的调试一般应当由编程人员本人进行。应避免随意丢弃测试用例，应将它们按某种格式记录下来，以便将来可以重新使用它们。回想一下图 2-2 中那个有悖于直观的现象。如果发现某一部分模块存在大量错误，那么很有可能这些模块甚至包含着更多的错误，只是尚未检查出来而已。这样的模块应该进行更进一步的测试，可能还需要进行额外的代码走查或检查。最后，记住模块测试的目的不是证明模块能够正确地运行，而是证明模块中存在着错误。

未经同意，严禁以任何形式拷贝

第 6 章 更高级别的测试

完成了对程序的模块测试之后，整个测试过程才刚刚开始，对于大型或复杂的软件来说尤为如此。考虑下面这个重要概念：

当程序无法实现其最终用户要求的合理功能时，就发生了一个软件错误。

根据这个定义，即使完成了一次非常完美的模块测试，仍然不能保证已经找出了程序中的所有错误。

因此，要结束整个测试任务，还必须进行其他形式的更深入的测试。我们将这些新形式的测试称为“更高级别的”测试。

软件开发过程在很大程度上是沟通有关最终程序的信息、并将信息从一种形式转换到另一种形式。由于这个原因，绝大部分软件错误都可以归因为信息沟通和转换时发生的故障、差错和干扰。

未经同意，严禁以任何形式拷贝

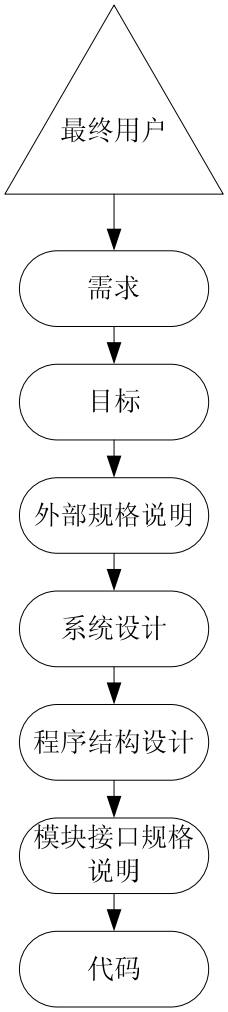


图 6-1 软件开发过程

图 6-1 描述了软件开发的这个观点，它表示了一个软件产品开发周期的模型。过程的流程可归结为以下 7 个步骤：

1. 将软件最终用户的要求转换为一系列书面的需求。这些需求就是该软件产品要实现的目标。
2. 通过评估可行性与成本、消除相抵触的用户需求、建立优先级和平衡关系，将用户需求转换为具体的目标。
3. 将上述目标转换为一个准确的产品规格说明，将产品视为一个黑盒，仅考虑其接口以及与最终用户的交互。该规格说明被称为“外部规格说明”。
4. 如果该产品是一个系统，如操作系统、飞行控制系统、数据库管理系统或雇员人事系统等，而不仅是一个程序（编译器、工资程序、字处理程序等），那么下

未经同意，严禁以任何形式拷贝

一步骤就是系统设计。该步骤将系统分割为单独的程序、部件或子系统，并定义它们的接口。

5. 通过定义每个模块的功能、模块的层次结构以及模块间的接口，来设计程序或程序集合的结构。
6. 设计一份准确的规格说明，定义每个模块的接口与功能。
7. 经过一个或更多的子步骤，将模块接口规格说明转换为每个模块的源代码算法。

以下是从其他角度来审视上述文档的形式：

- 需求规格说明定义了为什么要开发程序。
- 目标定义了程序要做什么，以及应做得怎样。
- 外部规格说明定义了程序对用户的准确表现。
- 与后续阶段相关的文档越来越详细地规定了程序是如何建立起来的。

假定软件开发周期的七个阶段包括了信息的沟通、理解和转换，以及大多数的软件错误都来源于信息处理中的故障，那么现在有三个补充的方法来预防或识别这些错误。首先，我们可以使软件开发过程更加精密，以防其中出现很多错误；其次，在每个阶段结束时可以引入一个独立的验证过程，在进入下一个阶段之前尽可能多地发现问题。这种方法如图 6-2 所示。举例来说，对外部规格说明的验证可以通过与前一个阶段的输出（对目标的叙述）进行比较，然后将任何发现的错误反馈到外部规格说明定义过程中去。在第七阶段结束时，使用本书第 3 章讨论的代码检查和走查方法进行验证。

未经同意，严禁以任何形式拷贝

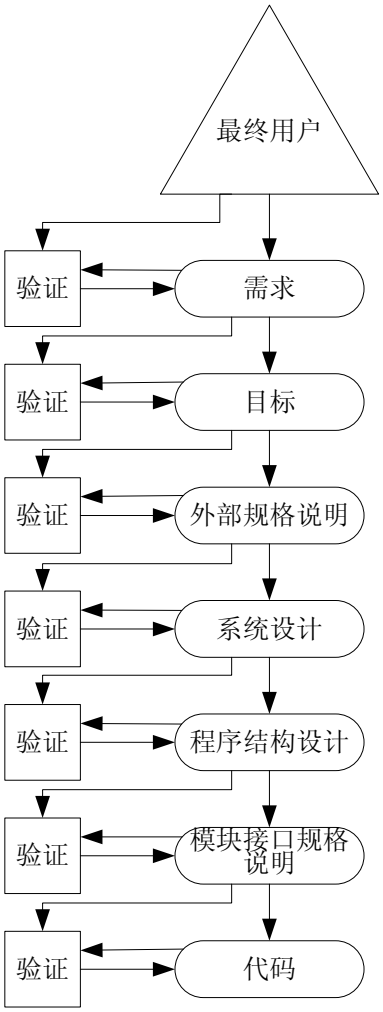


图 6-2 包含中间验证步骤的开发过程

第三个方法是对不同的开发阶段采用不同的测试方法。也就是说，将每一个测试过程都重点针对一个特定的转换步骤，从而也针对一类具体的错误。这种方法如图 6-3 所示。测试周期是模仿软件开发周期建立起来的，换言之，我们应该能够在开发过程和测试过程之间建立起一对一的联系。举例来说：

- 模块测试的目的是发现程序模块与其接口规格说明之间的不一致。
- 功能测试的目的是为了证明程序未能符合其外部规格说明。
- 系统测试的目的是为了证明软件产品与其初始目标不一致。

这种结构的好处是避免了没有效果的多余测试，并使我们不会遗漏掉大量的错误类型。举例来说，不能仅将系统测试定义为“对整个系统的测试”并且可能仅重

未经同意，严禁以任何形式拷贝

复先前的测试，而是针对一种特定类型的错误（在将目标转换为外部规格说明时所犯的错误），并就开发过程中的特定类型的文档进行度量。

图 6-3 所示的更高级别的测试方法最适用于软件产品（作为合同的结果或面向广泛应用而编写的程序，与做试验用的或仅供作者本人使用的程序有所不同）。不作为产品而编写的程序常常没有正规的需求和目标。对于这些程序，功能测试可能就是惟一的更高级别的测试。同时，对更高级别测试的需求是与程序的规模一同增长的。这是由于在大型程序中，设计错误（在早期开发阶段所犯的错误）与编码错误之间的比率要比在小程序中的比率高很多。

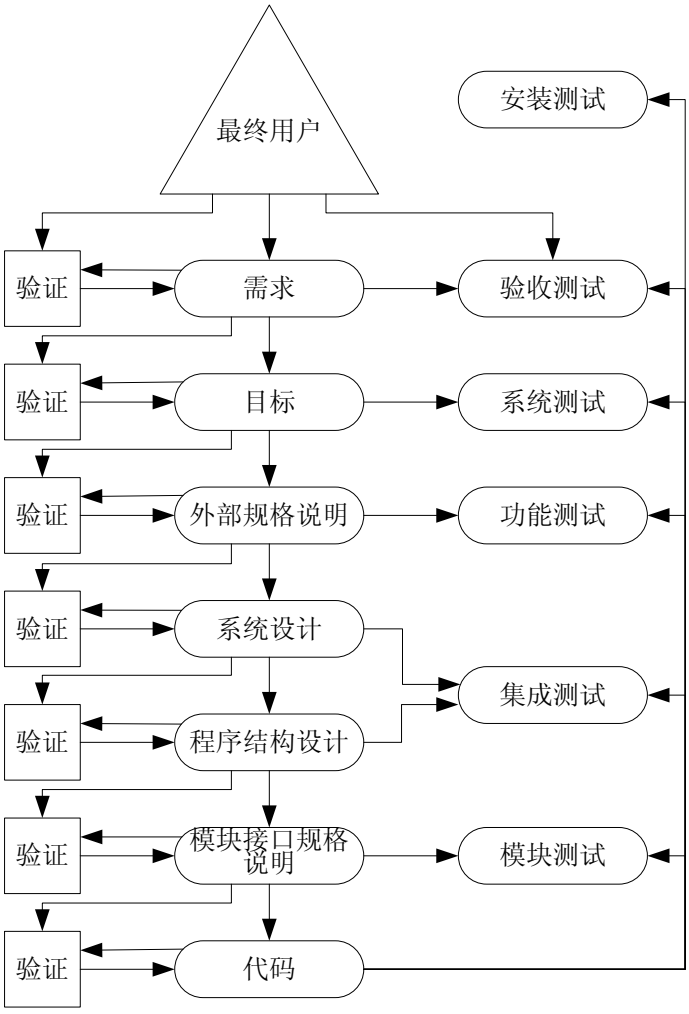


图 6-3 开发过程与测试过程的对应关系

未经同意，严禁以任何形式拷贝

注意，图 6-3 所示的测试过程顺序并不定意味着严格的时间顺序。举例来说，由于系统测试并非定义为“功能测试之后进行的测试类型”，而是定义为一种特定类型的测试，关注于具体类型的错误，因此它很有可能与其他测试过程在时间上发生部分重叠。

在本章中，我们将讨论功能测试、系统测试、验收测试和安装测试的过程。在这里忽略了集成测试，因为集成测试往往并不作为一个独立的测试步骤，而且在进行增量模块测试时，它是模块测试的隐含部分。

我们将简要讨论这些测试过程，并且大多不提供范例，因为这些更高级别的测试所使用的特定测试技术是与具体的被测程序高度相关的。举例来说，对操作系统进行的系统测试的特点（测试用例的类型、测试用例设计的方式、使用的测试工具）与对编译器、核反应堆控制程序或数据库应用程序所进行的系统测试的特点有很大不同。

本章的最后几节将会计论测试计划和测试组织等话题，以及决定何时终止测试这一重要问题。

6.1 功能测试(Function Testing)

如图 6-3 所示，功能测试是一个试图发现程序与其外部规格说明之间存在不一致的过程。外部规格说明是一份从最终用户的角度对程序行为的精确描述。

除了在小程序中的使用情况之外，功能测试通常是一项黑盒操作。也就是说，要依赖早期的模块测试的过程来实现理想的白盒逻辑覆盖准则。

在进行功能测试时，需要对规格说明进行分析以获取测试用例集。本书第 4 章所讨论的等价类划分方法、边界值分析方法、因果图分析方法和错误猜测方法尤其适合于功能测试。实际上，第 4 章中的例子就是功能测试的范例。对 FORTRAN 语言的 `DIMENSION` 语句、考试评分程序以及 `DISPLAY` 命令的描述实际上就是外部规格说明的例子（但是，请注意它们并不是完全现实的例子：例如真正的评分程序的外部规格说明应该包括对报告格式的准确描述）。因此，在本节中不再列举有关功能测试的例子。

未经同意，严禁以任何形式拷贝

本书第 2 章中的很多原则也特别适合于功能测试。跟踪哪些功能暴露出的错误数量最多，这个信息作常重要，因为它告诉我们这些功能很可能还包含着大多数尚未发现的错误。应记住对无效和未预想到的输入条件给予足够的重视。回想一下，对预期结果的定义是测试用例的重要部分。最后，应始终牢记功能测试的目的是为了暴露程序的错误以及与规格说明不一致之处，而不是为了证明程序符合其外部规格说明。

6.2 系统测试(System Testing)

系统测试最容易被错误理解，也是最困难的测试过程。系统测试并非是测试整个系统或程序功能的过程，因为有了功能测试，这样会显得多余。如图 6-3 所示，系统测试有着特定的目的：将系统或程序与其初始目标进行比较。给定这个目标之后，隐含两方面的含义：

1. 系统测试并不局限于系统。如果产品是一个程序，那么系统测试就是一个试图说明程序作为一个整体是如何不满足其目标的过程。
2. 根据定义，如果产品没有一组书面的、可度量的目标，系统测试也就无法进行。

在寻找程序与其目标之间的不一致的过程中，应重点注意那些在设计外部规格说明的过程中所犯的转换错误。系统测试因而成为一种关键的测试类型，因为就软件产品本身、所犯错误的数量及其严重性而言，开发周期的这个阶段是最易出错的。

这也暗示与功能测试的情况不同，外部规格说明不能作为获得系统测试用例的基础，否则就破坏了系统测试的目标。然而另一方面，也不能利用目标文档本身来表示测试用例，因为根据定义，这些文档并不包含对程序外部接口的准确描述。克服这一两难局面的方法是利用程序的用户文档或书面材料。通过分析目标文档来设计系统测试，分析用户文档来阐明测试用例。该方法能够产生两方面的作用，一是将程序与其目标和用户文档相比较，二是同时也将用户文档与程序目标相比较，如图 6-4 所示。

未经同意，严禁以任何形式拷贝

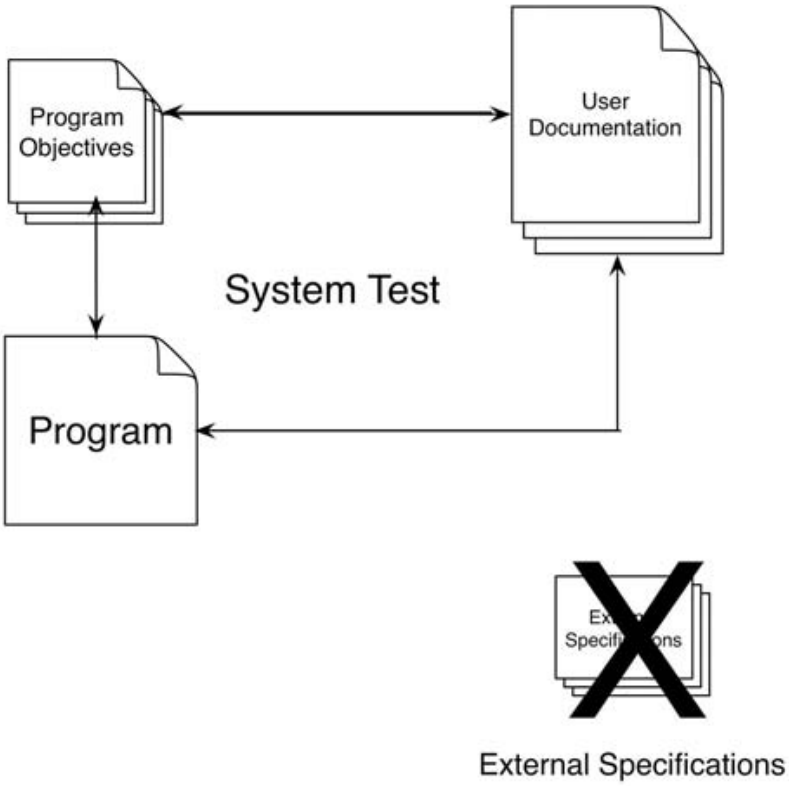


图 6-4 系统测试

图 6-4 说明为什么系统测试是最困难的测试过程。图中最左边的箭头表示将程序与其目标进行比较，是系统测试的核心目的，但是没有说明使用什么样的测试用例设计方法。因为目标文档阐述了程序应该做什么、做到什么程度，却没有说明程序功能如何表现。举例来说，本书第 4 章中定义的 DISPLAY 命令的目标如下：

> 该命令用来从终端查看主存储空间中的内容，其语法应与所有其他系统命令的语法相一致。用户可以通过一个地址范围或者一地址加上一数值来定义空间范围。该命令操作符应具有合理的默认值。
>
> 命令的输出可以分多行显示多个字（以十六进制形式），字与字之间以空格相隔。每一行须包含该行第一个字的地址。该命令是条“不太重要的”指令，意味着其在合理的系统负载下，应在两秒之内开始显示输出，输出各行之间不应有可觉察的延时。命令处理器中发生的编程错误在最坏情况下可能导致该命令失效；而系统以及用户交互则不应受到影响，系统投入使用之后，命令处理器中包含的用户发现的错误不应超过一个。

未经同意，严禁以任何形式拷贝

目标虽已阐明，但并没有确认生成测试用例集的方法，仅有一些含糊却有用的指南来指导如何编写测试用例，以试图证明程序与目标文档中每一条语句都存在着不一致性。因此，系统测试采取了一种不同的测试用例设计方法，不是描述一项技术，我们讨论的是不同类型的系统测试用例。由于没有一个方法，系统测试需要大量的创造性。事实上，设计好的系统测试用例比设计系统或程序需要更多的创造性、智慧和经验。

这里我们将讨论 15 种测试用例，我们并不认为所有的 15 种类型都适用于任何程序，但是为了避免有所遗漏，设计测试用例时应考虑全部的 15 种类型。

6.2.1 能力测试(Facility Testing)

最明显的系统测试类型是判断目标文档提及的每一项能力（或功能，为了避免与功能测试发生混淆而不使用“功能”一词）是否都确实已经实现。能力测试的过程是逐条语句地检查目标文档，当某条语句定义了一个“要做什么”（例如，“语法应该一致……”、“用户应当可以指定一个空间范围……”等），就判断程序是否满足。此种类型的测试常常可以在不使用计算机的情况下进行；有时人工对目标和用户文档进行比较就足够了。尽管如此，利用问题检查单将有助于在下一次进行测试时，确保人工检查的目标是相同的。

6.2.2 容量测试(Volume Testing)

第二类系统测试是使程序经受大容量数据的检验。举例来说，编译器可能要编译规模非常庞大的源程序，连接编辑器可能需要处理一个包含上千模块的程序，电子电路模拟器可能要输入一个包含上千部件的电路，而操作系统的作业队列可能已经达到饱和的容量。如果程序需要处理跨越不同卷的文件，则应产生足够的数据使程序从一个卷转换到另一个中。换言之，容量测试的目的是为了证明程序不能处理目标文档中规定的数据容量。

由于容量测试显然需要大量的资源，鉴于对机器和工时的考虑，不可进行过多的容量测试。当然，每个程序应该至少进行几次容量测试。

未经同意，严禁以任何形式拷贝

6.2.3 强度测试(Stress Testing)

强度测试使程序承受高负载或强度的检验。这不应和容量测试发生混淆；所谓高强度是指在**很短的时间间隔内**达到的数据或操作的数量峰值。类似的情况是测试一名打字员。容量测试是判断打字员能否处理大篇幅的稿子，而强度测试则是判断打字员能否达到每分钟 50 个单词的速度。

由于强度测试涉及时间因素，因此，它不适用于很多程序，如编译器或批处理工资程序。然而，强度测试适用于在可变负载下运行的程序，以及交互式程序、实时程序和过程控制程序。假如某个空中交通控制系统要求在其区域内最多可跟踪 200 架飞机，则可以通过模拟 200 架飞机存在的情况来对其进行强度测试。由于在客观上无法避免第 201 架飞机进入该区域，因此需要进一步的强度测试，以考察系统对这个不速之客的反应。附加的强度测试则会模拟大量飞机同时进入该区域的情况。

如果操作系统要求支持最多 15 个多道程序的作业，则可尝试同时运行 15 个作业对其进行强度测试。可以让学员强行打左舵、后拉节流阀、放下襟翼、抬起机头、放下起落架、打开着陆灯并向左转弯等所有这些操作同时进行，观察系统如何反应，从而对飞行员训练模拟器进行强度测试（这个测试用例可能需要一个长着四只手的飞行员，或者现实一点，需要飞行座舱里有两个测试专家）。可以通过让所有被监视的过程同时产生信号，来对过程控制系统进行强度测试。当对电话交换系统进行强度测试时，可以让大量电话同时打入该系统。

基于 web 的应用程序是最常按受强度测试的软件之一。在这里，我们需要确信的是应用程序及硬件能够处理一定容量的并发用户。读者可能会争辩说，也许有数百万人在同一时刻访问站点，但这是不现实的。我们需要弄清用户群，然后设计一个强度测试，体现出可能访问站点的最大人群的情况。本书第 9 章将提供关于测试基于 Web 应用程序的更多信息。

虽然有很多强度测试体现的是程序在运行过程中可能会遇到的情况，然而也有另一些强度测试确实体现了“不可能发生”的情况，但这并不意味这些测试是无用的。如果在这些不可能发生的情况中检查出了错误，那么这项测试就是有价值的，

未经同意，严禁以任何形式拷贝

因为同样的错误也可能发生在现实的、强度稍低的环境中。

6.2.4 易用性测试(Usability Testing)

系统测试的另一个重要类型是试图发现人为因素或易用性的问题。当本书的第一版出版时，计算机行业并不太注意研究和定义编程系统中良好的人为因素问题。今天的软件系统，尤其是那些为广大商业市场而设计的软件，通常都进行了广泛的人为因素的研究，而现在的软件自然也受益于过去的成千上万的程序和系统。然而，对人为因素的分析依然是一项极为主观的事情。在以下的清单中，我们列举了需要测试的一些问题：

1. 每个用户界面是否都根据最终用户的智力、教育背景和环境要求而进行了调整？
2. 程序的输出是否有意义、不模糊且没有计算机的杂乱信息？
3. 错误诊断（如错误信息）是否直接，用户是否需要有计算机学科的博士学位才能理解它门？举例来说，程序是否产生了诸如“IEK022A OPEN ERROR ON FILE ' SYSIN ' ABEND CODE = 102 ?”此类的信息？象这样的信息在二十世纪七、八十年代的软件系统中并不鲜见。今天的面向大众销售的系统在这方面有了改进，但我们仍然会遇到诸如“出现一个未知错误”或“程序遇到了一个错误，必须重新启动”这样无用的信息。自己编写的程序是由自己控制的，不应加入这些无用的信息。即使我们并不开发软件，如果在测试小组中工作，那么可以推动人机界面这个领域的改进。
4. 整体的用户界面是否在语法、惯例、语义、格式、风格和缩写方面展现出了相当程度的概念完整性，基本的一致性和统一性？
5. 在准确性极为重要的环境里，如网上银行系统，输入中是否有足够的冗余信息？举例来说，该系统可能会要求输入账号、用户名和 PIN（个人识别号）来验证访问账户信息的是合法用户。
6. 系统是否包含过多或不太可能用到的选项？现代软件的一个趋势是，仅向用户提供那些基于软件测试和设计考虑而确定出的最有可能使用的菜单选项。一个设计良好的软件可以向用户学习，并开始向不同的用户展示其经常访问的菜单项。即使已经有了这样智能化的菜单系统，仍需要设计成功

未经同意，严禁以任何形式拷贝

的软件，使得对不同选项的访问合乎逻辑、符合直觉。

7. 对于所有的输入，系统是否返回了某些类型的即时确认信息？举例来说，在点击鼠标进行输入的环境里，被选项可以变换颜色，或者某个按钮对象可以显示凹进或凸起的状态。如果要让用户从列表中选择，那么当用户做出选择后被选序号应显示在屏幕上。还有，如果被选的操作需要一些运行时间（如果软件正在访问一个远程的系统，情况常会如此），那么应显示一条信息通知用户当前正在做什么。
8. 程序是否易于使用？举例来说，如果输入是区分大小字符的，这一点对用户来说是否清楚？此外，如果程序要求浏览一系列的菜单或操作，那么返回到主菜单的方法是否清楚？用户是否可以很容易浏览到上一层或下一层？

6.2.5 安全性测试(Security Testing)

由于社会对个人隐私的日益关注，许多软件都有特别的安全性目标。安全性测试是设计测试用例来突破程序安全检查的过程。举例来说，我们可以设计测试用例来规避操作系统的内存保护机制，破坏数据库管理系统的数据安全机制。设计此种测试用例的方法之一是研究类似系统中已知的安全问题，然后生成测试用例，尽量暴露被测系统存在相似问题。例如，在杂志，聊天室和新闻组中发布的资料，经常包含有操作系统或其他软件系统的已知错误。通过在与被测软件提供相似服务的现有系统中搜寻安全漏洞，可以设计测试用例来判断软件是否受到类似问题的困扰。基于 web 的应用程序常常比绝大多数程序所需的安全测试级别更高。对于电子商务网站尤其如此。尽管已经有了足够多的技术（例如密码学）允许客户在因特网上安全地完成交易，但不能单纯依赖技术的应用来确保安全。除此之外，我们必须向客户群证明软件是安全的，否则就会有失去客户的风险。另外，本书第 9 章提供了更多的有关基于因特网的应用程序的安全性测试的资料。

6.2.6 性能测试(Performance Testing)

很多软件都有特定的性能或效率目标，这终特性描述为在特定负载和配置环境下程序的响应时间和吞吐率。再一次强调，由于系统测试的目的是为了证明程序不

未经同意，严禁以任何形式拷贝

能实现其目标，因此应设计测试用例来说明程序不能满足其性能目标。

6.2.7 存储测试(Storage Testing)

类似地，软件偶尔会有存储目标，举例来说，可能描述了程序使用的内存和辅存的容量，以及临时文件或溢出文件的大小。应设计测试用例来证明这些存储目标没有得到满足。

6.2.8 配置测试(Configuration Testing)

诸如操作系统，数据库管理系统和信息交换系统等软件都支持多种硬件配置，包括不同类型和数量的 I/O 设备和通信线路，或不同的存储容量。通常可能的配置数量非常之大，以至于测试无法面面俱到，但是至少应该使用每一种类型的设备，以最大和最小的配置来测试程序。如果软件本身的配置可忽略掉某些程序组件，或可运行在不同的计算机上，那么该软件所有可能的配置都应测试到。

如今的很多软件都设计成可运行在多种操作系统下，因此如果测试此类程序，应该在该程序面向的所有操作系统环境中对其进行测试。对设计在 Web 浏览器里运行的程序，需要特别的注意，因为 Web 浏览器的种类繁多，并不是所有浏览器都按同样方式运行。除此之外，即使是同一种 Web 浏览器，在不同的操作系统之下，运行方式也会不同。

6.2.9 兼容性/配置/转换测试(Compatibility/Configuration/Conversion Testing)

大多数开发的软件都并不是全新的，常常是为了替换某些不完善的系统。这样的软件往往有着特定的自标，涉及与现有系统的兼容以及从现有系统的转换过程。再次强调，在针对这些目标测试程序时、测试用例的目的是证明兼容性目标未被满足，转换过程并未生效。在将数据从一个系统转移到另一个系统时，应尽力发现错误。升级数据库管理系统就是一个例子。需要确定现有的数据安置到了新的系统中。有很多不同的方法测试这个过程；但这些方法都高度依赖于所使用的数据库系统。

未经同意，严禁以任何形式拷贝

6.2.10 安装测试(Installability Testing)

有些类型的软件系统安装过程非常复杂，测试安装过程是系统测试中的一个重要部分。对于包含在软件包中的自动安装系统而言，这尤其重要。安装程序如果出现故障，会影响用户对软件的成功体验。用户的第一次体验来自于安装软件的过程。如果这个过程进行得很槽糕，用户或顾客就要么寻找其他的产品，要么对软件的有效性不抱太大信心。

6.2.11 可靠性测试(Reliability Testing)

当然，所有类型的测试都是为了提高软件的可靠性，但是如果软件的目标中包含了对可靠性的特别描述，就必须设计专门的可靠性测试。测试可靠性目标可能很困难。举例来说，诸如公司广域网（WAN）或因特网服务供应商（ISP）等现代在线系统在整个运行期间，正常运行时间应占 99.97%。我们现在还不太可能花上数月甚至数年的时间来测试这个目标。今天的关键软件系统的可靠性标准甚至更高，而现今的硬件可以令人信服地保障这个目标的实现。但如果软件或系统有更为适中的平均故障间隔时间（MTBF）目标或合理的（以测试而言）功能错误目标，就有可能对其进行测试。

例如，MTBF 值不超过 20 个小时，或者系统目标是程序在投入使用之后暴露的不同错误的数量不得超过 12 个，那么就可以进行测试，特别是使用了统计的、程序验证的或基于模型的测试方法之后。这些方法都超出了本书的范围，但有些技术文献（网上或其他方面的）对这个领域提供了充分指导。

举例来说，如果读者对软件测试的这个领域感兴趣，可以研究归纳断言的概念。这种方法的目的是设计出有关被测软件的一系列定理，作为判断软件中不存在错误的依据。这种方法首先要对程序的输入条件及其正确结果编写断言。断言用形式逻辑的符号表示，通常是一阶谓词演算。然后需要确定程序中的每个循环，对于每一个循环都写一个断言，描述出在循环中任意点都不变的（总为真）条件。程序现在已经被划分为固定数量的固定长度的路径（在成对的断言中的全部所有路径）。对于每一条路径，取中间程序语句的语义来修改断言，最终到达路径的终点。此时在路径的终点处存在着两条断言：原先的断言以及从路径的另一个终点处断言引申出

未经同意，严禁以任何形式拷贝

的断言。然后写出一条定理，说明原先的断言隐含着引申出的断言，并试图证明这个定理。如果能够证明该定理，就可以认为该程序不存在错误——只要程序最终能够结束，需要单独的证明来说明程序总会结束。

虽然此种类型的软件证明或预测听起来非常复杂，但可靠性测试及软件可靠性工程（SRE）的概念已经为我们所认识，并且对于那些必须维持非常高的正常运行时间的系统，其重要性日益增加。为了说明这一点，请查看表 6-l 中某个系统为支持不同的正常运行时间的需要而每年必须达到的运行小时数。这些数字能够说明对 SRE 的需求。

表 6-1 不同正常运行时间要求的小时数/年

正常运行时间的比例需求	要求的运行小时/年
100	8760.0
99.9	8751.2
98	8584.8
97	8497.2
96	8409.6
95	8322.0

6.2.12 可恢复性测试(Recovery Testing)

诸如操作系统、数据库管理系统和远程处理系统等软件通常都有可恢复性目标，说明系统如何从程序错误、硬件失效和数据错误中恢复过来。系统测试的一个目标是证明这些恢复机制不能够正确发挥作用。我们可以故意将程序错误置入某个系统中，判断系统是否可以从中恢复。诸如内存校验错误或 I/O 设备错误等硬件错误也可以进行模拟。而如通信线路中的嗓音或数据库中的无效指针等数据错误可以故意生成或模拟出来，以分析系统的反应。

这些系统的设计目标之一是使平均恢复时间（MTTR）最小。系统宕机才通常会减少公司的收入。我们的一个测试目标是证明系统不能满足 MTTR 的服务合同。MTTR 往往有上界和下界，所以测试用例应反映出这些界限。

6.2.13 适用性测试(Serviceability Testing)

软件还可能有适用性或可维护性的目标。所有的此类目标都必须测试到。这些

目标可能定义了系统提供的服务辅助功能，包括存储转存程序或诊断程序、调试明显问题的平均时间、维护过程以及内部业务文档的质量等。

6.2.14 文档测试(Documentation Testing)

如同我们在图 6-4 中所描述的那样，系统测试也需要检查用户文档的正确性。完成此任务的主要方法是根据文档来确定系统测试用例的形式。也就是说，一旦设计完成某个具体的测试情况，应该使用文档作为编写实际测试用例的指南。同时，用户文档应成为审查的对象（类似于本书第 3 章中的代码审查的概念），检查其正确性和清晰性。在文档中描述的任何范例应编成测试用例，并提交给程序。

6.2.15 过程测试(Procedure Testing)

最后，很多软件都是较大系统的组成部分，这些系统并不完全是自动化的，包含了很多人员操作过程。在系统测试中，必须对所有已规定的人工过程，如系统操作员、数据库管理员或最终用户的操作过程进行测试。

举例来说，数据库管理员必须记录备份和恢复数据库系统的操作过程。在可能的情况下，应由与数据库管理不相关的人来测试这些过程。然而，公司必须为充分测试这些过程而提供所需的资源，这些资源通常包括硬件和额外的软件许可证。

6.2.16 系统测试的执行

系统测试执行中一个最关键的考虑是决定由谁来进行测试。我们从反面来回答这个问题：(1）不能由程序员来进行系统测试；（2）在所有的测试阶段之中，这是惟一明确不能由负责该程序开发的机构来执行的测试。

第一点基于的事实是，执行系统测试的人思考问题的方式必须与最终用户相同，这意味着必须充分了解最终用户的态度和应用环境，以及程序的使用方式。那么显然的是，如果可行的话，一位或多位最终用户是很好的执行测试的候选人。但是，由于一般的最终用户都不具备执行很多前面所描述的测试类型的能力或专业技术，因此，理想的系统测试小组应由几位专业的系统测试专家（以执行系统测试作为职业)、一位或两位最终用户的代表、一位人类工程学工程师以及该程序主要的

分析人或设计者所组成。将原先的设计者包括进来并不违反先前的测试原则，即不提倡测试由自己编写的程序。因为程序自构思以来已经历经人手，所以原先的设计者不会再受到心理束缚的影响，对程序的测试不会再触及该原则。

第二点基于的事实是，系统测试是一项“随心所欲，百无禁忌”的活动，而软件开发机构会受到心理束缚，有悖于此项活动。而且大多数的开发机构最为关心的是让系统测试进行得尽可能顺利并按时完成，而不会尽力证明程序不能满足其目标。系统测试至少应由很少（如果有的话）受开发机构左右的独立人群来执行。也许最经济的执行系统测试的方式（所谓经济，是指花一定的成本发现最多的错误，或利用更少的费用发现相同数量的错误）是将测试分包给一个独立的公司来完成。这一点将在本章的后面章节进一步讨论。

6.3 验收测试(Acceptance Testing)

让我们回到图 6-3 所示的开发过程的完整模型上来，可以看到验收测试是将程序与其最初的需求及最终用户当前的需要进行比较的过程。这是一种不寻常的测试类型，因为该测试通常是由程序的客户或最终用户来进行，一般不认为是软件开发机构的职责。对于软件按合同开发的情况，由订购方（用户）来进行验收测试，将程序的实际操作与原始合同进行对照。如同其他类型的测试一样，验收测试最好的方法是设计测试用例，尽力证明程序没有满足合同要求。假如这些测试用例都是不成功的，那么就可以接受该程序。对于软件产品的情况，如计算机制造商的操作系统或编译器，或是软件公司的数据库管理系统，明智的用户首先会进行一次验收测试以判断产品是否满足其要求。

6.4 安装测试(Installation Testing)

图 6-3 描述的测试过程的剩余部分是安装测试。安装测试在图 6-3 中的位置有些不寻常，它与所有其他测试过程不同。与设计过程中的任何阶段都没有联系。它的不寻常是由于其目的不是为了发现软件中的错误，而是为了发现在安装过程中出现的错误。

在安装软件系统期间会发生很多事件。作为示例的简短列表包括了下列事件：

未经同意，严禁以任何形式拷贝

1. 用户必须选择大量的选项。
2. 必须分配并加载文件和库。
3. 必须进行有效的硬件配置。
4. 软件可能要求网络联通，以便与其他软件连接。

安装测试应由生产软件系统的机构来设计，作为软件的一部分来发布，在系统安装完成之后进行。除此之外，测试用例需要检查以确认已选的选项集合互不冲突，系统的所有部件全部存在，所有的文件已经创建并包含必需内容，硬件配置妥当等。

6.5 测试的计划与控制

如果认为测试一个大型软件系统可能需要编写、执行和验证数万个测试用例、处理数千个模块、改正数千个错误、雇佣数百人花费一年甚至更长的时间工作，那么很明显我们在计划、监视和控制测试过程方面遇到了巨大的项目管理挑战。事实上，这些问题非常繁杂，我们可以将整本书都用来讨论软件测试的管理问题。本节的初衷是总结其中的一些问题。

正如第 2 章所提到的，在计划测试过程中最常出现的主要错误是默认为不会发现软件缺陷。这个错误带来的显然结果是对计划投人的资源（人力、时间表及计算机时间）明显估计不足，这在计算机行业内是个声名狼藉的问题。造成这个问题的原因是测试阶段处于开发周期的最后阶段，致使调整资源非常困难。另外，可能是更重要的问题，即对软件测试的定义有误，因为很难看到对测试正确定义（测试的目的是发现错误）的人在假定找不到任何错误的情况下去计划一个测试。

与大多数项目的情况一样，计划是管理测试过程中至关重要的一环。一个良好的测试计划应包括：

1. **目标。**必须定义每个测试阶段的目标。
2. **结束准则。**必须制定准则以规定每个测试阶段何时可以结束，该问题将在下一节中讨论。
3. **进度。**每个阶段都须有时间表。应指出何时设计、编写和执行测试用例，某些软件技术，如极限编程（在本书第 8 章中讨论）要求在程序编码开始之前就设计测试用例和单元测试。

未经同意，严禁以任何形式拷贝

4. **责任。**对于每一个阶段，应当确定谁来设计、编写和验证测试用例，谁来修改发现的软件错误。由于在大型项目中讨论特定的测试结果是否代表错误时，有可能出现争端，因此还需要确定一名仲裁者。
5. **测试用例库及标准。**在大型项目中，用于确定、编写以及存储测试用例的系统方法是必须的。
6. **工具。**必须确定需要使用的测试工具，包括计划由谁来开发或采购、如何使用工具以及何时需要使用工具。
7. **计算机时间。**计划每个测试阶段所需的计算机时间，包括用来编译应用程序的服务器（如果需要的话）、用来进行安装测试所需的桌面计算机、用来运行基于 web 应用程序的 web 服务器、联网的设备（如果需要的话）等等。
8. **硬件配置。**如果需要特别的硬件配置或设备，则需要一份计划来描述该需求，该如何满足需求以及何时需要满足。
9. **集成。**测试计划的一部分是定义程序如何组装在一起的方法（例如自顶向下的增量测试）。一个系统如果包含大的子系统或程序，可按增量的方式组装在一起，例如可以使用自顶向下或自底向上的方法，但是这些构造块是程序或子系统，而不是模块。如果是这种情况，就需要一个系统集成计划。系统集成计划规定了系统集成的顺序、系统每个版本的功能以及编写“脚手架”代码以模拟不存在的部件的职责分工。
10. **跟踪步骤。**必须跟踪测试进行中的方方面面，包括对错误易发模块的定位，以及有关进度、资源和结束准则的进展估计。
11. **调试步骤。**必须制定上报已发现错误、跟踪错误修改进程以及将修改部分加入系统中去的机制。调试计划中还应包括进度、责任分工、工具以及计算机时间/资源等。
12. **回归测试。**回归测试在对程序作了功能改进或进行了修改之后进行，其目的是判断程序的改动是否引起了程序其他方面的退步。回归测试通常重新执行测试用例中的某个子集。回归测试很重要，因为对程序的改动和对错误的纠正要比原来的程序代码更容易出错（与报纸排版错误很相似，这些错误通常由于最后所做的编辑改动而引起的，而不是修改先前版本而引起的）。回归测试计划规定了测试人员、测试方法和测试时间，它也是必须的。

未经同意，严禁以任何形式拷贝

6.6 测试结束准则

在软件测试过程中最难回答的一个问题，是判断何时终止测试，因为我们无法知道刚刚发现的错误是否是最后一个错误。事实上，除了非常小的软件，期望所有的错误最终都能被发现是不切实际的。在这种两难情况之下，而且基于经济条件也要求测试必须最终结束的事实，我们可能会产生疑惑，是极其武断地回答此问题呢，还是存在一些有用的终止准则？我们在实际中使用的典型的结束准则既无意义，也不能实现目标。最常见的两个准则是：

1. 用完了安排的测试时间后，测试使结束。
2. 当执行完所有测试用例都未发现错误，测试便结束。也就是说，当所有的测试用例不成功时便结束。

第一条准则没有任何作用，因为我们可以完全什么都不做也可满足它。它并不能衡量测试的质量。第二条准则同样也是无用的，因为它也与测试用例的质量无关，而且也不能够实现测试目标，它下意识里鼓励我们编写发现错误可能性较低的测试用例。

正如本书第 2 章所述，人类是高度受目标驱使的。如果一旦获悉所有的测试用例都不成功就完成了任务，那么人们就会下意识地朝这个目标编写测试用例，避开了有用的、高效的和具破坏性的测试用例。

有三类较为有用的结束准则。

第一类，但不是最佳的准则，根据的是特定的测试用例设计技术。举例来说，我们会这样定义模块测试的结束准则：

测试用例来源于（l）满足多重条件覆盖准则，以及（2）对模块接口规格说明进行边界值分析，产生的所有测试用例最终都是不成功的。

我们会在满足下列情况时规定功能测试结束：

测试用例来源于（l）因果图分析，（2）边界值分析，以及（3）错误猜测，产生的所有测试用例最终都是不成功的。

尽管这种类型的准则要优于前面提到的两条准则，但仍然存在三个问题。首先，

对于那些没有特定方法的测试阶段，如系统测试阶段，这类准则不起作用。第二，它要依赖于主观的度量，因为没有办法保证测试人员适当而又严格地使用特定的方法，如边界值分析方法。第三，不同于设置一个目标再让测试人员选择最佳的实现方法，它的做法正好相反，指定了测试用例设计的方法，却并不设定目标。因此，这种类型的准则对于某些测试阶段有时很有效，但是只有在测试人员根据以往的经历，证明自己可以成功地使用测试用例设计方法时，这些准则方可适用。

第二类，也许也是最有价值的准则，是以确切的数量来描述结束测试的条件。因为测试的目的是发现错误，为什么不将测试结束准则定为发现了某个既定数量的错误呢？举例来说，在对某个具体模块进行模块测试时，直到发现了三个错误才可以认为测试结束了。也许系统测试的结束准则应该规定为发现并修改了 70 个错误，或测试实际进行了 3 个月，无论以后发生什么。应该注意的是，虽然这种准则强化了软件测试的定义，但它也有两个问题，每一个都是可以解决的。一个问题是判断如何获得要发现的错误数量。得到这一数字需要进行下面几个预测：

1. 预测出程序中错误的总数量。
2. 预测这些错误中有多大比例可能通过测试而发现。
3. 预测这些错误中有多少是由各个设计阶段产生的，以及在什么样的测试阶段能够发现这些问题。

可以通过几种方法来大致预测错误的总数。一种方法是利用以前程序的经验来预测出数字。另外，还存在多种预测模型。有些模块需要测试一段时间，记录下连续发现错误的间隔时间，然后将这些时间输入一个公式的参数中。有些模块被置入一些已知但未公开的种子错误，测试一段时间后，检查被发现的种子错误与非种子错误的比例。还有的模型则让两个独立的测试小组分别测试一段时间，然后检查各自找出的错误以及两个组找出的共同问题，再使用这些参数来预测错误的总数。还有一种获得预计数字的粗略方法是使用行业范围内的平均值。举例来说，在编码结束时（在进行代码走查或检查之前），一般程序中的错误数量大致是每 100 行语句中含 4-8 个错误。

上面列举的第二个预测（可以通过测试发现的错误比例）包含一定程度的随意猜测，考虑了程序的性质以及未发现的错误造成的后果。

未经同意，严禁以任何形式拷贝

关于错误是如何及何时产生的，我们现在得到的信息还很少，因此第二个预测最为困难。现有的数据表明，在大型程序中，大约有 40％的错误是编码和逻辑设计错误，剩下的错误则产生于早期的设计阶段。

为了使用这个准则，需要根据手头的程序做出自己的预测。这里有一个简单的例子。假设我们要着手测试一个 10,000 行语句的程序，进行代码检查之后剩余的错误数量预计每 100 行语句 5 个错误，并且我们估计，作为测试的目标，要检查出 98％的编码和逻辑设计错误，以及 95％的早期设计错误。这样，错误总数为 500。在这 500 个错误之中，我们假设 200 个错误是编码和逻辑设计错误，300 个是设计缺陷。因此，我们的目标是找出 196 个编码和逻辑设计错误以及 285 个设计错误。表 6-2 显示了对何时可能发现错误的近似合理的预测。

表 6-2 对错误发现时期的推测

	编码和逻辑设计错误	设计错误
模块测试	65％	0％
功能测试	30％	60％
系统测试	3％	35％
总计	98％	95％

如果我们计划进行 4 个月的功能测试、3 个月的系统测试，可以建立如下 3 个结束准则：

1. 当发现并修改了 130 个错误之后（估计的 200 个编码和逻辑设计错误中的 65％），模块测试即告结束。
2. 当发现并修改了 240 个错误之后（200 个错误的 30％加上 300 个错误的 60％），或功能测试进行了 4 个月之后，无论后面发生什么，功能测试即告结束。加上第二条的原因在于，如果我们很快发现了 240 个错误，那么就很有可能表明我们低估了错误的总数，因此，不应很早就结束功能测试。
3. 当发现并修改了 111 个错误之后，或系统测试进行了 3 个月之后，无论以后发生什么，系统测试即告结束。

这类准则的另一个明显问题是过度地预测。在上述例子中，如果在功能测试开始时剩余的错误数量少于 240 个会发生什么情况呢？根据这条准则，我们可能永远也不能结束功能测试。

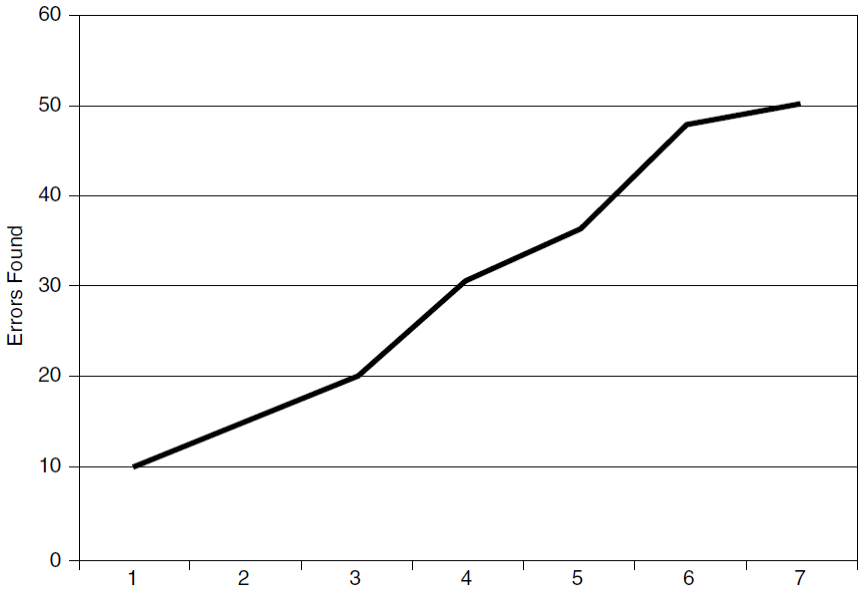
未经同意，严禁以任何形式拷贝

如果你仔细想一想，这里有一个奇怪的问题。这个问题是错误的数量不够，程序的质量过高了。我们可以不将其当成问题，因为这是个很多人都喜欢遇到的“问题”。如果这个问题确实发生了，可以根据常识来解决它。如果我们在 4 个月内没有发现 240 个错误，项目经理可以聘请一个局外人来分析测试用例，判断问题究竟是测试用例不足，还是测试用例很棒却没什么错误可发现。

第三类结束准则表面上似乎很容易，其中却涉及许多判断和直觉。它需要我们在测试过程中记录每单位时间内发现的错误数量。通过检查统计曲线的形状，常常可以决定究竟是继续该阶段的测试，还是结束它并开始下一测试阶段。

假设某个程序正在进行功能测试，对每周发现的错误数量都进行了记录。如果第 7 周的曲线如图 6-5 的上部所示，那么即使发现的错误数量已经达到了结束准则，此时结束测试也会显得草率，因为，在第 7 周里我们似乎仍处于高峰（发现很多错误），此时最明智的决定（记住我们的目标是发现错误）是继续功能测试。如有必要，设计额外的测试用例。

然而另一方面，如果曲线处于图 6-5 的下部，错误发现率明显下降，意味着我们已经“啃干净”了功能测试这块骨头，也许最佳的行动是结束功能测试并开始新的测试类型（也许是系统测试）。当然，我们还必须考虑其他因素，比如错误发现率的降低是否是因为缺少计算机时间，或执行完了可用的测试用例。



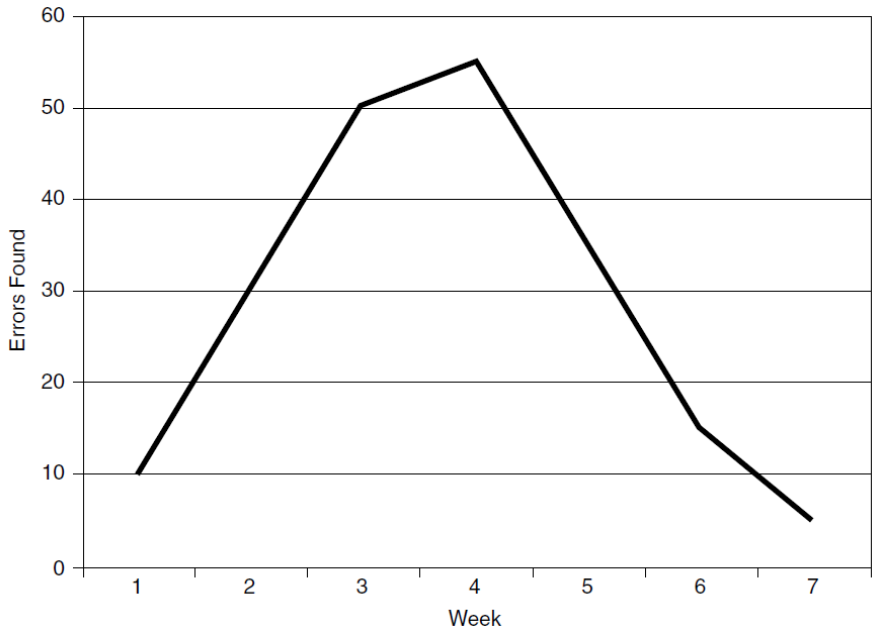


图 6-5 通过记录单位时间内发现的错误来预测测试的结束

图 6-6 显示了如果我们没有记录发现错误的数量，会发生什么情况。该图显示了个非常大的软件系统的三个测试阶段。一个显而易见的结论是，该项目在第 6 时段后不应转到别的阶段。在第 6 时段，错误发现率还很高（对于测试人员而言，发现率越高越好）。然而在这个时候转移到下一个阶段，导致了错误发现率的明显下降。最佳的结束准则可能是上述三种类型的组合。对于模块测试而言，特别是由于多数项目在此阶段都没有正式跟踪已发现的错误，最佳的结束准则可能是第一类。我们应该要求使用一系列具体的测试用例设计方法。而对于功能测试和系统测试而言，结束准则可能是发现了既定数量的错误，或用完了计划的时间，再出现什么都不管，但条件是错误分析与时间图的对比表明测试的效率已很低了。

未经同意，严禁以任何形式拷贝

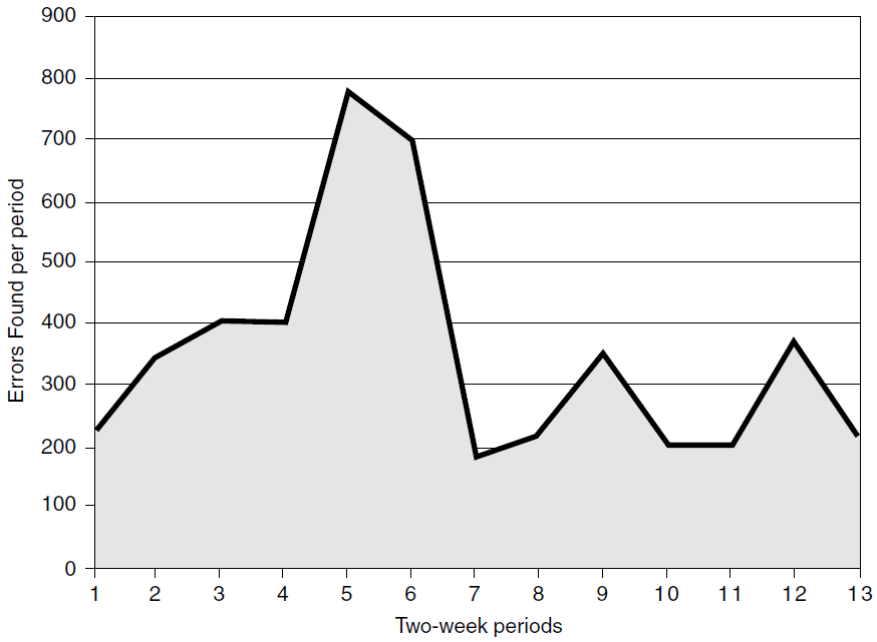


图 6-6 对某个大型项目测试过程的事后研究

6.7 独立的测试机构

在本章的前面章节和第 2 章中，我们强调了软件机构应避免测试自己的软件，其中的原因在于，负责开发程序的机构难以客观地测试同一程序。就公司的架构而言，测试部门应尽可能远离开发部门。事实上，最理想的是测试机构不应是同一个公司的一部分，因为如果不是这样，测试机构仍然会受到与开发部门同样的管理压力的影响。

解决这个矛盾的一个方法是雇佣独立的公司进行软件测试。这是个好主意，不管是系统的设计和使用单位开发的这个软件，还是第三方单位开发的这个软件。这种做法常被提及的好处是提升了测试过程中的积极性、建立了与开发机构的良性竞争、避免了测试过程处于开发机构的管理控制之下，以及独立的测试机构带来的解决问题的专业知识。

未经同意，严禁以任何形式拷贝

第 7 章 调试(DEBUGGING)

简单地讲，调试是执行一次成功的测试之后所要进行的工作。记住，所谓成功的测试，是指它可以证明程序没有实现预期的功能。调试是一个包含两个步骤的过程，从执行了一个成功的测试用例、发现了一个问题之后开始。第一步，确定程序中可疑错误的准确性质和位置；第二步，修改错误。

虽然调试对于程序测试来说非常必要、不可或缺，但它似乎是软件开发过程中最不受程序员欢迎的部分之一。其主要原因可能包括以下几点：

- **个人自尊会从中阻挠**。不管我们是否喜欢，调试都说明了程序员并不完美，要么在软件的设计，要么在程序编码时会犯错。
- **热情耗尽**。在所有的软件开发活动中，调试是最耗费脑力的苦差事，况且，进行调试往往经受着来自机构或自身的巨大压力，必须尽可能快地改正问题。
- **可能会迷失方向**。调试是艰苦的脑力工作，因为发现的错误实际上可能会出现在程序的任何语句中。也就是说，如果不首先检查程序，我们就不能绝对地肯定在一个薪金管理程序出具的支票中出现的数字错误不是由某个子程序引起的，该子程序要求操作员将一个特定的表格传输给打印机。让我们以诊断一个物理系统为例子作对比，如汽车。假如汽车在爬坡时熄火了（症状），那么我们可能会迅速而有效地排除掉某些部件——调频/调幅收音机、速度表或汽车门锁——引起该故障的可能。根据我们对汽车引擎的整体了解，该故障一定是发生在引擎上，我们甚至可以排除掉某些引擎部件，如水箱和滤油器。
- **必须自力更生**。与其他软件开发活动相比，关于调试过程的研究、资料和正式的指南都比较少。

尽管本书是关于软件测试的，并不讨论调试，但这两个过程显然是相互联系的。针对调试的两个步骤，即错误定位和错误修改，对错误进行定位可能解决了 95％的

未经同意，严禁以任何形式拷贝

问题。因此，本章集中讨论错误的定位过程，当然是假定某个成功的测试用例已经发现了一个错误。

7.1 暴力法调试(Debugging by Brute Force)

调试程序的最为普遍的模式是所谓的“暴力”方法。这种方法之所以流行，是因为它不需要过多思考，是耗费脑力最少的方法，但同时也效率低下，通常来讲不是很成功。

暴力调试方法可至少被划分为三种类型：

1. 利用内存信息输出来调试。
2. 根据一般的“在程序中插入打印语句”建议来调试。
3. 使用自动化的调试工具进行调试。

第一种类型，使用内存信息输出（通常使用十六进制或八进制格式粗略地显示所有的存储区域）是最缺乏效率的暴力调试方法，原因如下：

- 难以在内存区域写源程序中的变量之间建立对应关系。
- 即使对下复杂程度较低的程序，内存信息输出也会产生数最非常庞大的数据，其中的大多数都是与调试无关的。
- 内存信息输出显示的是程序的静态快照，仅能显示出在某一个时刻程序的状态；为了发现错误，还需要研究程序的动态状态（随时间的状态变化）。
- 内存信息输出很少可以精确地在错误发生的地方产生，因此无法显示在错误发生时程序的状态。错误发生到输出内存信息这段时间之内程序执行的活动，可能会掩盖掉发现错误所需的线素。
- 通过分析输出的内存信息来发现问题的方法并不大多（因此很名程序员都是密切注视，急切地渴望着错误能神奇地从内存信息输出中自行暴露出来）。

第二种类型，在失效的程序中插入输出变量值的语句，这种做法也不具有很强的优势。它可能比内存信息输出要好一些，因为可以显示程序的动态状态，让我们检查的信息可以相对容易地与源程序联系起来。但是这种方法同样也有很多缺点：

- 它不是鼓励我们去思考程序中的问题，而主要是一种碰运气的方法。

- 它所产生的需要分析的数据量非常庞大。
- 它要求我们修改程序，这些修改可能会掩盖掉错误、改变关键的时序关系，或者会引入新的错误。
- 它可能对小型程序有效，但如果应用到大型程序，成本就相当高。况且对于某些类型的程序，如操作系统或过程控制软件，这种办法甚至无法使用。

第三种类型，自动化调试工具的工作机制类似于在程序中插入打印语句，但是并不修改程序本身。可以使用编程语言的调试功能，或使用特殊的交互式调试工具来分析程序的动态状态。可能会用到的典型的语言功能有：产生可打印的语句执行轨迹的机制、子程序调用以及/或者对特定变量的修改等。调试工具的一个共同的功能是可以设置断点，使程序在执行到某条特定语句或改动了某个特定变量的值时暂停执行，然后程序员就可以检查程序的当前状态。同样，这种方法也主要是在碰运气，常常会生成数量过于庞大的无关数据。

这些暴力调试方法的主要问题在于：它们都忽略了思考的过程。我们可以在调试程序和侦破谋杀案之间找出相似点来。实际上，在几平所有的谋杀悬念小说中，谜案都是通过仔细分析线索，将表面上不重要的细节全联结起来而最终侦破的。这不是一个使用蛮力的方法，要使用蛮力的是寻觅障碍物或搜寻财宝。

还有些证据表明，无论调试小组成员是富有经验的程序员还是学生，肯动脑筋而不是依赖别人帮助的人能够更快、更准确地发现程序错误。因此，我们建议仅在下列情况下使用暴力调试方法：(1）其他的方法都失败了：（2）作为我们下面将会讨论的思考过程的补充，而不是替代方法。

7.2 归纳法调试(Debugging by Induction)

很显然，认真的思考能够发现大部分错误，甚至不需要调试人员使用调试工具．归纳是一种特殊的思考过程，可以从细节转到全局，也就是从线索（即错误的症状，可能是一个或多个测试用例的结果）出发，寻找线索之间的联系。归纳的过程如图 7-1 所示。

未经同意，严禁以任何形式拷贝

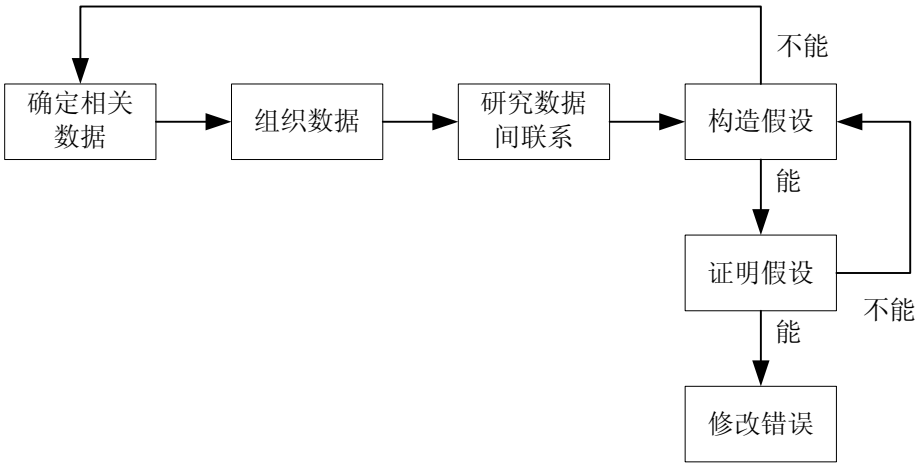


图 7-1 使用归纳法的调试过程

归纳调试的步骤如下：

1. 确定相关数据。调试人员犯的一个主要错误是未能将所有可用的数据或症状都考虑进去。第一步是列举出所有知道的程序执行的正确和不正确之处，这些不正确之处即是症状，让我们相信确实存在错误。那些相似却不相同、且未引起症状出现的测试用例提供了额外的有价值的线索。
2. 组织数据。记住，归纳意味着从特殊到一般，因此第二步是组织这些相关数据，以便观察线索间的模式，尤其重要的是要找到矛盾、事件，比如仅当客户的保险金账户收支不太平衡时出现的错误。我们可以采用图 7-2 所示的表格来组织现有的数据。“是什么”框列举的是总体的症状，“在何处”框描述了这些症状出现的地方，“多大程度”框描述了这些症状的范围和重要性。注意“是”和“否’列，它们所描述的矛盾之处最终可能会导致对错误的假设。

?	**Is**	**Is not**
What		
Where		
When		
To what next		

图 7-2 组织线索的一种方法

3. 做出假设。下一步是研究线索之间的联系，利用线索结构里可能的模式做出一个或多个关于错误原因的假设。如果还无法做出推测，就需要更多的

数据。如果可能有多个假设存在，首先选择最有能的一个。

4. 证明假设。考虑到调试在进行时所承受的压力，这个时期最主要的错误是忽略了这个阶段，直接跳到结论去改正问题。但是在继续下一步之前，证明这些假设的合理性是非常重要的。如果忽略了这一步，可能接下去只修改了问题症状，而没解决问题本身。应将假设与其最初的线索或数据相比较，以此来证明假设的合理性，确定这些假设可以完全解释这些线索的存在。如果无法解释，要么这些假设是无效的或不完整的，要么还有更多的错误存在。

举一个简单的例子、假设在第 4 章描述的考试评分软件报告了一个明显的错误。错误是在某些但不是所有情况下，中间值似乎不正确。在某个特殊的测试用例中，有 51 名学生被评分。正确打印出宋的平均分数为 73.2，但打印出的中间值是 26 分，而不是预期的 82 分。经过对该测试用例及其他一些测试用例结果的检查，线索按图 7-3 所示的形式进行组织。

?	**Is**	**Is not**
What	报告 3 中显示的中间值不正确	计算平均值或标准偏差时出现
Where	仅在报告 3 中出现	在其它报告中出现。学生成绩的计算似乎正确
When	当测试学生为 51 时发生	在测试学生数量为 2 和 200 时未发生
To what next	显示的中间值为 26。当学生数量为 1 时也同样发生，显示的中间值。	

图 7-3 组织线索的例子

下一步是通过寻找模式和矛盾之处，做出关于该错误的假设。我们看到的一个矛盾是这个错误似乎出现在学生人数为奇数的测试用例中，这也许是个巧合，但看来很重要，因为我们要根据学生人数为奇数或偶数而不同地计算中间值。还有一个奇怪的模式：在些测试用例中，计算出来的中间值总是小于或等于学生的人数（26 小等于 51，1 小等于 1）。这时，一个可能的方法是再重新运行一次学生人数为 51 名的测试用例，给学生打与以前不同的分数，看一下是如何影响中间依的计算的。如果中间值仍然是 26，那么“否——多大程度”框可以填上“中间值似乎与实际分数无关”。尽管这个结果提供了一条有价值的线索，但即使没有它，我们可能已经

能够猜出这个错误来。从现有数据计算出的中间值似乎等于学生人数的一半，经过四舍五入后得到最接近的一个整数。换句话说，如果将分数设想为存储在一个分类表里，该程序打印的是中间学生的人数而不是其成绩。因此，我们有了一个关于该错误准确性质的坚定的假设。下一步就是通过检查代码或执行一些附加的测试用例来证明这个假设。

7.3 演绎法调试(Debugging by Deduction)

演绎的过程是从一些普遍的理论或前提出发，使用排除和精炼的过程，达到一个结论（错误的位置），参见图 7- 4 。

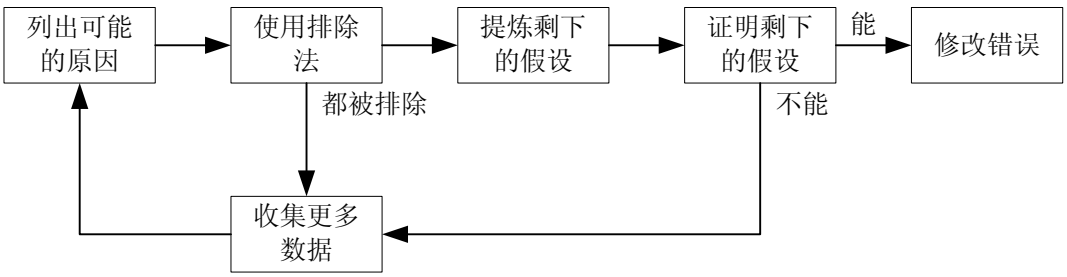


图 7-4 使用演绎法的调试过程

举个谋杀犯的例子，与归纳过程相反，首先从一系列嫌疑人入手，通过排除（花匠有当时不在现场的合理证词）和提炼（罪犯可能是红色头发）的过程，判断出管家可能犯了罪。演绎的步骤如下：

1. 列举出所有可能的原因或假设．第一步是建立一份所有想象得到的错误线索的清单，线索不需要有完整的解释，它们纯粹是一些推测，帮助我们组织和分析现有的数据。
2. 利用数据排除可能的原因。详细检查所有的数据，尤其寻找存在矛盾的地方（图 7-2 可以用在此处），然后尽量排除所有可能的原因，仅留下一条，如果所有的原因都排除掉了，需要增加额外的测试用例，得到更多的数据来设计新的推测。如果剩下的原因多于一个，那么首先选择最有可能的原因，即主要假设。
3. 提炼剩下的假设。此时的可能原因也许是正确的,但可能不够具体，不能指

未经同意，严禁以任何形式拷贝

出错误来。因此，下一步是使用现有的线索来提炼这个推侧．举例来说，我们可能会首先想到“对文件中最后事务的处理可能存在错误”，并将其提炼为“缓冲区中的最后事务被文件结束指示器覆盖” 。

4. 证明剩下的假设。这个重要步骤与归纳法中的第 4 步骤相同。

举个例子，假设我们着手对第 4 章讨论的 DISPLAY 命令进行功能测试。在由因果图分析方法确定的 38 个测试用例中，我们首先使用 4 个测试用例。作为建立输入条件过程的一部分，我们对内存进行初始化，将第一个、第五个、第九个、…字的值设置为 0000，将第二个、第六个、… 字的值设置为 4444 ，将第三个、第七个、…字的值设置为 8888，将第四个、第八个、… 字的值设置为 CCCC。也就是说，每个内存字单元都初始化为每个字的首字节地址中的低位十六进制数字（23FC、23FD、23FE 和 23FF 地址的值为 C）。

图 7-5 显示了这些测试用例、预期的输出及测试用例的实际输出。

显然，我们遇到了一些问题，所有的测试用例都没有产生预期的结果（全部都成功了）。让我们从调试与第一个测试用例相关的错误开始。该命令表明，从 0 地址开始（默认情况），要显示 E（十进制中的 14）个地址（回忆一下，规格说明定义所有的输出应每行包括 4 个字或 16 个字节）。

测试用例的输入	预期的输出	实际的输出
DISPLAY	0000 = 0000 4444 8888 CCCC	M1 INVALID COMMAND SYNTAX
DISPLAY 21v-29	0000020 ＝ 0000 4444 8888 CCCC	000020 = 4444 8888 CCCC 0000
DISPLAY.11	000010 = 0000 4444 8888 CCCC 000010 = 0000 4444 8888 CCCC	000000 = 0000 4444 8888 CCCC
DISPLAY 8000-END	M2 STORAGE REOUESTED IS BEYOND ACTUAL MEMORY LIMITS	008000 = 0000 4444 8888 CCCC

图 7-5 DISPLAY 命令的测试用例输出结果

为出现的不期望的错误信息列举可能的原因：

1. 程序不能接受单词 DISPLAY。
2. 程序不能接受句号。
3. 程序不允许第一个操作数为默认情况。程序要求在句号之前声明一个存储地址。

未经同意，严禁以任何形式拷贝

4. 程序不允许 E 作为有效的字节数量。

下一步是尽力排除这些原因。如果所有原因都排除掉了，那么需要退回去并扩充一下原因的清单。如果剩下来的原因超过了一个，那么就需要检验额外的测试用例以确定惟一的错误假设，或继续使用可能性最大的原因。由于我们手上还有其他测试用例，可以看到图 7-5 中的第二个测试用例似乎可以排除掉第 1 条假设，而第三个测试用例尽管产生了错误的结果，也似乎可以排除掉第 2 和第 3 条假设。

下一步是提炼第 4 条假设。它看上去足够具体，但直觉告诉我们实质的内容要比表面上看到的多。它看上去似乎是一个更为一般的错误实例。那么我们可以认为程序不能正确识别特殊的十六进制字符 A～F 。在其他测试用例中缺少这些字符，使得这听起来是一个行得通的解释。然而我们不能马上得出结论，而应该首先考虑所有的已知信息。第四个测试用例可能代表一个完全不同的错误，也可能提供了一条关于当前错误的线索。假设系统的最高有效地址是 7FFF，那么第四个测试用例将如何显示一个明显不存在的区域呢？显示的值是我们初始化后的值而不是无用的信息，这个事实让我们推测该命令不知何故显示了 0～7FFF 之间的某些内容。我们可能会想到，这种错误也许会发生在程序将命令的操作数当成十进制数（而不是规格说明中要求的十六进制数）的情况，第三个测试用例证实了这种假设，程序并未显示 32 个字节的内存单元的内容，而仅显示了 16 个字节，这与我们的假设是一致的，即“11”被当作了十进制数。因此，提炼后的假设是，程序将字节数当作内存地址处理，并将输出列表中的内存地址当作十进制数。最后的步骤是证明该假设。看一看第四个测试用例，如果 8000 被解读为十进制数，则对应的十六进制数是 1F40 ，这样就会产生我们所看到的输出。作为进一步的证据，检查第二个测试测试用例。输出是不正确的，但如果 21 和 29 被当作十进制数，那么内存地址 15～ID 中的内容将被显示出来，这是与测试用例的错误结果是一致的。因此，我们几乎可以确切地定位错误了：程序认为操作数是十进制数，并将内存地址按十进制的值打印出来，这与规格说明是不符的。而且，这个错误似乎是造成所有四个测试用例产生错误结果的原因。经过一些思考，我们发现了这个错误，同时也解决了其他三个乍看起来毫不相关的问题。

注意，该错误可能在程序中的两个地方显现出来：解释输入命令的部分和在输

未经同意，严禁以任何形式拷贝

出列表上打印内存地址的部分。

说句离题的话，这个可能由于错误理解规格说明而引起的错误进一步印证了我们的建议，即程序员不应该测试自己编写的程序。如果程序员在犯了这个错误之后仍然去设计测试用例，很有可能在编写测试用例时犯同样的错误。换句话说，程序员预料的输出将不同于图 7-5 所示的；这些输出将是按操作数是十进制数的理解而被计算出来的，因此这个基本的错误可能不会被察觉到。

7.4 回溯法调试(Debugging by Backtracking)

在小型程序中定位错误的一种有效方法是沿着程序的逻辑结构回溯不正确的结果，直到找出程序逻辑出错的位置。换句话说，从程序产生不正确结果（如打印了不正确的数据）的地方开始，从该处观察到的结果推断出程序变量应该是些什么值。在头脑中，从这个位置开始逆向执行程序，重复使用“如果程序在此处的状态是这样的，那么程序在上面位置的状态就必然是那样的”过程，就能很快定位出错误。使用这个过程，可以确定程序中从状态符合预期值的位置点，到第一个状态不符合预期值的位置点之间的范围。

7.5 测试法调试(Debugging by Testing)

最后一个“思维型”的调试方法是使用测试用例。这可能听起来有些奇怪，因为从本章一开始就将调试和测试区分了开来。然而，考虑下面两种类型的测试用例。供测试的测试用例，其目的是暴露出以前尚未发现的错误．供调试的测试用例，其目的是提供有用的信息，供定位某个被怀疑的错误之用。两者之间的区别是，供测试的测试用例会“胖”一些，因为我们尽量使用较少数量的测试用例来涵盖较多的条件，而供调试的测试用例则“瘦”一些，因为每个测试用例仅需要覆盖一个或几个条件。

换句话说，当发现了某个被怀疑的错误的症状之后，我们需要编写与原先有所变化的测试用例，尽量确定错误的位置。实际上，这种方法不是一个完全独立的方法；它常常结合归纳法一起使用，以获得进行假设和/或证明假设所需的信息。它也可以和演绎法一起使用，以排除有嫌疑的原因，提炼剩下的假设，并/或证明假设。

未经同意，严禁以任何形式拷贝

7.6 调试的原则

在本节中，我们将讨论一系列的调试原则，在实质上也是心理学的原则。与第2 章的测试原则情况一样，这些调试原则有很多在直观上很明显，但却常常被遗忘或忽略。由于调试的过程由两部分组成，即定位错误及修改错误，因此我们也将讨论两类原则，

7.6.1 定位错误的原则

1. 动脑筋

前面的章节隐含指出，调试是一个解决问题的过程。最为有效的调试方法是动脑筋对错误症状的有关信息进行分析。一个高效的程序调试人员应该不使用计算机就能定位大多数的错误。

2. 如果遇到了僵局，就留到稍后解决

人类的潜意识是一个潜在的问题求解器。我们经常提到的所谓灵感，其实就是当人类的意识停留在诸如吃东西、走路或看电影之上时，潜意识却正在思考另一个向题。如果在合理时间内（也许小型程序为 30 分钟，大一点的程序为几个小时），我们还不能定位某个间题，就丢开它，做些其他的事情，因为思维的效率开始明显下降。忘记这个问题一段时间之后，我们的潜意识可能已经解决了它，或者思维会焕然一新，可以重新检查问题的症状。

3. 如果遇到了困境，就把问题描述给其他人听

与其他人交谈可能会帮助我们发现一些新的东西。事实上，经常是仅仅将问题描述给一个好的倾听者时，我们就会突然找到问题的解决之道，而无需倾听者提供任何帮助。

4. 仅将测试工具作为第二种手段

在试过了其他的方法之后才使用调试上具，并将其作为头脑思考的辅助手段，而不是替代手段。正如本章前面所述，调试工具比如输出和跟踪工具，代表的是一种偶然的调试方法。经验证明，不使用工具的人即使在调试并不熟悉

的程序时，也要比使用工具的人更为成功。

5. 避免使用试验法——仅将其作为最后的手段

调试程序的新手最常犯的错误是为了解决问题而试验性地去修改程序。调试者可能会说：“我知道什么出错了，所以我要改动一下语句．看一看会发生什么”。这种纯粹是无计划的方法甚至不属于调试．它表现的是盲目的行动。它获得成功的机会不仅很小，而且还会将新的错误引入程序，使问题更为复杂。

7.6.2 修改错误的技术

1. 存在一个缺陷的地方，很有可能还存在其他缺陷

这是对本书第 2 章原则的重申，即发现程序某个部分存在一个错误时，该部分存在其他错误的可能性要高于没有发现错误时的可能性。换句话说，错误有扎堆的倾向。在修改某个问题的同时，应检查下紧临的地方，看看有没有任何可能是错误之处。

2. 应纠正错误本身，而不仅是其症状

另一个普遍的错误做法是只修改了错误的症状或仅仅是该错误的一个实例，而不是错误本身。如果所做的改正不符合错误的所有线索，那么可能只修改了错误的一部分。

3. 正确纠正错误的可能性并非 100%

如果将这个观点告诉一些人，他们当然会表示赞同，但是如果将它说给正在修改错误的人听，答案就可能不一样了(“是的，对大多数情况是这样，但这个修改如此之小，它肯定百分之百地正确”)。我们永远也不要假设为纠正错误而增加到程序中的代码是正确的。用新的语句替换原来的语句，这种修改要远比程序中原先的代码更易发生错误。言外之意是应对错误的修改进行测试，也许比对原先程序的测试还要严格。一个严格的回归测试计划可以确保对某个错误的修改没有在程序的其他位置引入另外的错误。

4. 正确修改错误的可能性随着程序规模的增加而降低

未经同意，严禁以任何形式拷贝

换句话说，根据我们的经验，由于修改不正确而引人的错误与原始错误之比，在规模较大的程序中呈递增趋势。对于一个广泛使用的大型程序，每发现6个新错误，其中就有1个错误是由于先前对程序的改正而造成的。

5. 应意识改正错误会引入新错误的可能性

我们不仅需要考虑到不正确的修改，而且还必须考虑到某个看似正确的修改会产生未料到的副作用，比如引入了一个新错误。不仅存在修改无效的可能，还存在修改引入了新错误的可能。言外之意是，不仅应在修改之后对错误的情境进行测试，还应执行回归测试以判断是否引入了新错误。

6. 修改错误的过程也是临时回到设计阶段的过程

我们应该认识到修改错误也是程序设计的一种形式。在认识到修改易产生错误的性质之后，常识告诉我们，在设计阶段使用的任何规程、方法和形式都同样适用于错误修改阶段。举例来说，如果项目证明代码检查很管用，那么在修改错误之后进行代码检查就显得倍加重要。

7. 应修改源代码，而不是目标代码

在调试大型系统，尤其是用汇编语言编写的系统时，偶尔会存在这样的修改错误的倾向，即先立即修改目标代码．稍后再修改源程序．这种方法带来了两个问题，（1）这通常是“通过试验进行调试”的信号；（2）目标代码与源程序不同步，这意味着当程序重新编译或重新汇编之后，同样的错误很容易又浮现出来，这是一种草率的、不专业的调试方法。

7.7 错误分析

有关程序调试最后一个应认识之处是，调试除了有消灭程序中错误的价值之外，还有其他重要作用：它可以告诉我们软件错误的一些本质．我们对此了解得非常之少．关于软件错误本质的信息可以为改进将来的设计、编码和测试过程提供有价值的反馈信息。

任何程序员和编程机构都可以从详细分析发现的错误，或至少一部分错误的过程中获得提高。错误分析是一项困难且费时的工作，相比“有百分之多少的错误是

逻辑设计错误”、“有百分之多少的错误出现在 IF 语句中”这些肤浅的分类，它蕴涵的内容要多得多。详细的错误分析会包括如下内容：

- **错误出现在什么地方？**这个问题是最难回答的问题之一，它需要通过对程序文档和项目历史进行回溯研究，但同时它也是最有价值的问题。它要求我们指出该错误的源头和发生时间。举例来说，错误的源头可能是规格说明中的一个模棱两可的语句，对先前错误的一次修改，或对最终用户需求的一个错误理解，
- **谁制造了这个错误？**如果发现有 60 %的设计错误都是由 10 名软件分析师中的某个人犯下的，或某程序员犯的错是其他程序员的 3 倍，难道这不是相当有用么？（不是为了处罚某人，而是为了进行培训。）
- **哪些做得不正确？**仅仅判断错误的发生时间和出现人员还不够，其中丢失的环节是准确地判断出错误发生的原因。错误是由于某人写得不清楚？是由干某人缺乏对该编程语言的培训？是打字错误？假设做得不对？还是因为没有考虑有效输入？
- **如何避免该错误的出现？**在下一个项目中可以进行哪些调整以避免该问题的出现？此向题的答案就是我们所寻找的最为宝贵的反馈信息或知识。
- **为什么错误没有早些发现？**如果错误是在测试引入段发现的，我们就应该研究为什么在更早些的测试阶段、代码审查和设计评审中没有发现该错误。
- **该如何更早地发现错误？**这个问题的答案是另一个宝贵的反馈信息。该如何改进评审和测试过程以便在将来的项目中更早地发现同类型的错误？假设分析的问题不是由最终用户发现的（也就是说，是由测试用例发现的），我们就应意识发生了一些有价值的事情：我们编写了一个成功的测试用例。这个测试用例为什么会成功？我们是否能从中学习些什么，无论是针对该程序还是将来的程序，设计出更多的成功用例？

再一次申明，分析的过程是很艰难的，但是找到的答案为改进后续的编程实践提供极其宝贵的价值。值得警惕的是，绝大多数的程序员和编程机构都尚未使用这种方法。

未经同意，严禁以任何形式拷贝

第 8 章 极限测试

20 世纪 90 年代出现了一种名为极限编程（XP，Extreme Programming）的新型软件开发方法。一位名叫 Kent Beck 的项目经理设计了这种轻量、敏捷的开发过程，并于 1996 年在戴姆勒• 克莱斯勒公司的项目中进行了首次测试。尽管此后还出现了其他几种敏捷软件开发过程，但 XP 到目前为止最流行的敏捷软件开发过程。事实上，现在已有众多的开源代码工具支持这种方法，这也证明了 XP 在开发人员和项目经理中的流行程度。

开发 XP 是为了支持诸如 Java，Visual Basic 及 C#等编程语言的应用。这些面向对象的语言使开发人员开发大型复杂应用的速度，比使用传统的 C，C++，FORTRAN 或 COBOL 语言更快。使用后者开发程序常常需要构建通用的类库来支持你的工作。诸如打印，排序，联网和统计分析等通用任务的实现方法并不是标准的构件。而 C#和 Java 等编程语言都含有全功能的应用编程接口（API），消除或减少了构建自定义类库的需要。

然而，伴随快速应用开发语言带来的好处，不利之处也随之而来。虽然开发人员可以更快地开发应用程序，但其质量并未得到保证。程序运行时经常不能满足程序规格说明的要求。XP 开发方法的目的是在短时间内开发高质量的程序。传统的软件开发过程依然在发挥作用，但往往会花很多的时间，在竞争激烈的软件开发领域里这就相当于损失了收入。

XP 模型高度依赖模块的单元和验收测试。总的来说，对每个无论多小的递增的代码变更，都必须进行单元测试，以确保代码库满足其规格说明的要求。事实上，测试在 XP 中的地位如此重要，以至于需要首先创建单元（模块）测试和验收测试，然后才创建代码库。这种形式的测试被称为极限测试（XT，Extreme Testing）。

8.1 极限编程基础

如前所述，XP 是一种相当新的软件开发过程，使开发人员可以快速地产生高

质量的代码。在这种情况下，我们可以将“质量”定义为代码库对其规格说明的满足程度。XP 重视采取简单的设计、在开发人员和客户之间建立联系、不断地测试代码库、重构以适应规格说明的变更，以及寻求用户的反馈。XP 更倾向于适合中小规模的软件开发，这些软件的规格说明的变更非常频繁，接近实时的沟通也是可能的。

XP 与传统的开发过程相比有几处不同。首先，它避免了大规模项目的综合症，即在开始编码之前客户与编程小组碰头，设计软件的每一个细节，项目经理们都知道这种做法的缺陷，因为为了反映新的业务谁则或市场情况，客户的规格说明和需求必须不停地变更。举例来说，财务部门可能要求工资报表按处理时间，而不是按支票号码进行排序，而营销部门可能会判断出，如果它没发电子邮件的话．用户将不会购买某产品。XP 的策划阶段将重点放在收集应用程序需求，而不是设计程序上。

XP 方法的另一个不同之处是避免了编写不需要的功能。如果客户认为某个功能虽然需要，但并不是要求实现的，那么在软件发行时通常不包含此项功能。因此我们可以将重点集中在正在进行的任务上，为软件产品增加价植。将精力集中在必需的功能之上，有助于在短时间内开发出高质量的软件。

然而，XP 方法的主要不同之处是其将精力集中在测试上。在经历了一个非常全面的设计阶段之后，传统的软件开发模型会建议首先编码，然后才生成测试接口，但在 XP 方法中，我们必须首先生成单元测试用例，然后才编写代码通过测试。根据本书第 5 章中讨论的概念，可以设计出 XP 环境中的单元测试。

XP 开发模型用 12 个核心实践来驱动该过程。表 8-l 总结了这些实践。简单来说．这 12 个核心的 XP 实践可以归纳为 4 个概念：

1. 聆听客户和其他程序员的谈话。
2. 与客户合作，开发应用程序的规格说明和测试用例。
3. 结对编码。
4. 测试代码库。

表 8-l 中所列的由每个实践提供的注释都是可以自我解释的。然而，有两个更重要的原则，即计划和测试，有必要进一步讨论。一个成功的计划阶段为整个 XP

未经同意，严禁以任何形式拷贝

过程奠定了基础，XP 的计划阶段与传统开发模型不同，通常将需求收集与应用设计结合起来。XP 中的计划重点是确定客户的应用需求，然后设计使用场景（或案例需求）来满足客户的应用需求。通过生成使用场景，可以深入地洞悉应用程序的目的和需求。此外，客户在一个开发周期的最后阶段执行验收测试时也可以用到这些使用场景。最后，计划阶段带来的一个无形的好处是，用户通过深入地参与进来，从而获得对程序的拥有感和信心。

表 8-1 极限编程的 12 个实践

实践	*注解*
1. 计划与需求分析	• 将市场和业务开发人员集中起来，共同确认每个软件特征的最大商业价值 • 以使用场景的形式重新编写每个重要的软件特征 • 程序员估计完成每个使用场景的时间 • 客户根据估计时间和商业价值选择软件的功能特征
2. 小规模，递增地发布	• 努力添加细微的、实在的、可增值的特征，频繁发布新版本
3. 系统隐喻，	• 编程小组确认隐喻，便于建立命名规则和程序流程
4. 简要设计	• 实现最简单的设计，使代码通过单元测试，假设变更即将发生，因此不要在设计上花太多时间、只是不停地实现
5. 连续测试	• 在编写模块之前就生成单元测试用例。模块只有在通过单元测试之后才告完成。此外，程序只有在通过了所有的单元测试和验收测试完成之后才算结束
6. 重构	• 清理和调整代码库；单元测试有助于确保在此过程中不破坏程序的功能。应在任何重构之后重新进行所有的单元测试
7. 结对编程	• 两位程序员协同工作，在同台机器开发代码库，这样就可以对代码进行实时检查，能极大地提高缺陷的发现率和纠正率
8. 代码的集体所有权	• 所有代码归全体程序员所有，没有哪个程序员只致力于开发某一个代码库
9. 持续集成	• 每天在变更通过单元测试之后将其集成到代码库中
10. 每周 40 小时工作	• 不允许加班．如果每周都全力工作了 40 小时，就不需要加班。在重大发布前的一星期例外
11. 客户在现场	• 开发人员和编程小组可以随时接触客户，这样可以快速、准确地解决问题，使开发过程不至于中断
12. 按标准编码	• 所有的代码看上去必须一致。设计一个系统隐喻有助于满足该原则

进行连续的测试是基于 XP 的方法取得成功的关键。虽然连续测试的原则中包含了验收测试，但单元测试占了主要的部分。我们应该确保代码的任何变更都改进了软件的质量，并且没有引入新的缺陷。连续测试的原则同样也支持为优化和调整代码库所进行的重构。持续的测试还会带来一个无形的好处，即建立对软件的信心．编程小组对代码库持有信心，源子用单元测试对代码库不断地进行验证。此外，

未经同意，严禁以任何形式拷贝

客户对投资的信心也会高涨，因为他们知道代码库每天都在通过单元测试。

既然我们已经描述了 XP 过程的 12 个实践，那么一个典型的 XP 项目是如何运作的呢？下面是一个简单的例子，列举了一个基于 XP 的项目可能具有的特征：

1. 程序员与客户会晤，决定产品需求并建立使用场景。
2. 在客户不在场的情况下，程序员进行会晤，将需求分解为独立的任务，并估计完成每项任务所需的时间。
3. 程序员向客户提交任务清单和时间估计，并要求客户产生一个功能优先级清单。
4. 编程小组依据程序员具备的能力，将任务分配给结对的程序员。
5. 每一对程序员依据应用程序的规格说明，对其编程任务生成单元测试用例。
6. 每一对程序员完成其任务，旨在编写出通过单元测试的代码库。
7. 每一对程序员在所有单元测试通过之前，不断修改和重测他们的代码。
8. 所有的结对程序员每天都整合、集成他们的代码库。
9. 编程小组发布应用程序的一个预览版本。
10. 客户进行验收测试，要么确认该应用程序，要么提交一份报告指出存在的缺陷或不足。
11. 程序员在验收测试成功的基础上发布一个产品版本。
12. 程序员根据最新的经验更新时间估计。

尽管 XP 方法魅力无穷，但它并不适合所有的项目和机构。根据 XP 倡导者的总结．如果编程小组充分地进行了 12 个实践，那么成功开发应用程序的机会就会显著提高。而批评者则认为，由于 XP 是一个过程，因此要么全部做到，要么什么也别做。如果漏掉了一个实践，那么 XP 应用得就不彻底，程序的质量就会受到影响。此外，批评者还认为，在未来修改程序以增加新的功能其代价要高于起初就将功能加人需求中并进行编码的代价。最后，一些程序员发现结对编程十分麻烦并侵犯隐私，所以，他们并不接受 XP 思想。

无论个人的观点如何，都应将 XP 视为完成项目的一种方法。应当根据项目的具体特性，仔细衡量 XP 方法的利弊，并做出可能的最佳选择。

未经同意，严禁以任何形式拷贝

8.2 极限测试：概念

为了满足 XP 的流程和思想，开发人员使用了极限测试方法，该方法强调连续测试。正如本章前面所提到的，极限测试主要由两种类型的测试组成：单元测试和验收测试。设计测试用例时所采用的原理与第 5 章描述的原理没有明显差异，但是在开发过程的哪个阶段设计测试用例则有所不同。尽管如此，极限测试和传统测试的目标仍然相同：即确定程序中的错误。本节的余下部分将从极限编程的角度，提供关于单元测试和验收测试的更多信息。

8.2.1 极限单元测试

单元测试是极限测试中采用的主要测试方法，它具有两个简单规则：所有代码模块在编码开始之前必须设计好单元测试用例，在产品发布之前须通过单元测试。乍看起来，这些原则似乎不太极端。但是，极限测试中的单元测试与前面描述的单元测试之间的最大差别在于，极限测试中的单元测试必须在模块编码之前就完成设计和生成。

起初我们可能会迷惑为什么，或者如何为尚未编写出的代码设计测试驱动程序。我们可能还会想到，没有设计测试的时间，因为应用程序的开发必须满足时间限制。这些考虑都是合理的，也容易克服。下面列出了在开始编码之前设计单元测试所带来的一些好处：

- 获得了代码将满足其规格说明的信心。
- 在开始编码之前，就展现了代码的最终结果。
- 更好地理解了应用程序的规格说明和需求。
- 可以先实现一些简单的设计，稍后再放心地重构代码以改善程序的性能，而无须担心破坏应用程序的规格说明。

在这些优点中，获得对应用程序规格说明和需求的洞察和理解不应被低估。举例来说，如果编码开始在先，我们可能无法充分理解程序输入值允许的数据类型和边界值。那么在不理解允许的输入的情况下，如何能够编写单元测试以执行边界分析呢？程序是否只接受数字、字符或两者都可以？如果首先设计单元测试，就必须理解规格说明。首先设计单元测试的做法就是 XP 方法的闪光点，因为它迫使我们

未经同意，严禁以任何形式拷贝

在开始编码之前，首先理解规格说明，排除了混淆。正如第 5 章所述，我们需要确定单元的范围。由于目前常用的编程语言，如 Java，C#，Visual Basic 大部分是面向对象的，因此模块常常就是类，或者甚至是单个的类方法。我们有时可以将“模块”定义为代表特定功能的一组类或方法。仅有程序员本人才知道程序的结构，以及任何最佳地为其设计单元测试。

即使是为最小的程序人工进行单元测试，也是一项让人生畏的工作。随着程序规模的增加，可能要设计数以百计，甚至数以千计的单元测试。因此，通常要采用一个自动化的软件测试套件来减轻连续执行单元测试的负担。在这些测试套件的帮助下，编写测试脚本，然后执行全部或其中的一部分。除此之外，测试套件通常可以生成报告，并对应用程序中频繁出现的缺陷进行分类。该信息可以帮助我们在将来主动清除这些缺陷。

非常有趣的是，一旦设计并确认了单元测试，这些“测试”用例库就与试图编写的应用软件程序一样有价值。因此，应当将这些测试用例保存在一个代码库中。此外，还应确保进行足够的备份，并具备所需的安全保密措施。

8.2.2 验收测试

验收测试是 XP 方法中第二类、也是同等重要的极限测试类型。验收测试的目的是判断应用程序是否满足如功能性和易用性等其他需求。在设计/计划阶段，由开发人员和客户来设计验收测试。

与迄今为止讨论的其他测试形式不同，验收测试是由客户，而不是开发人员或编程搭档来执行的。在这种方式中，客户对应用程序是否满足他们的要求进行客观、公正的确认。客户通过使用场景来设计验收测试。使用场景与验收测试之比通常是一对多，也就是说，每一个使用场景都可能需要不止一个的验收测试。

极限测试中的验收测试可以是自动化或非自动化的。举例来说．当客户必须确认某个用户输入界面的颜色和屏幕布局是否满足其规格说明时，所进行的测试是非自动化的，当应用程序须通过采用某些数据源（比如用二维表格模拟生产数据）作为输入数据来计算工资表格的值时，所进行的测试则是自动化的。

客户使用验收测试来验证应用程序是否得到了预期的结果。如果与预期结果不

未经同意，严禁以任何形式拷贝

一致，即被当作一个缺陷，报告给开发小组，如果客户发现了多个缺陷，那么在将列表传递给开发小组之前，得对缺陷进行优先级别排序。当缺陷被修正，或程序中发生任何变更时，客户都需要重新执行验收测试。从这点来看，验收测试也是回归测试的一种形式。

需要特别提醒的是，程序可能通过所有的单元测试，却不能通过验收测试。为什么会这样呢？因为单元测试是确认程序单元是否满足特定的规格说明，如工资扣除计算是否正确，而并非具体的可操作性或审美特性。对于商用软件来说，产品的外观和感觉是非常重要的部分。理解了规格说明，但却不理解其可操作性，通常会发生这种情况.

8.3 极限测试的应用

在本节中，我们开发了一个小型的 Java 应用程序，使用 JUnit（一个基于 Java 的开源单元测试套件）来描述极限测试的概念，例子本身很小，但其概念可适用于大多数的编程环境。我们的例子是一个命令行的应用程序，仅判断输入值是否为素数。为简洁起见，程序的源代码 check4Primo.java 及其测试配件（test harness) check4PrimeTestjava 列在附录 A 中。在本节中，我们节选了程序片段来说明主要的思路。程序的规格说明如下：

> 开发一个命令行的应用程序，接收一个正整数 n（0<=n<=1000），判断 n 是否为素数。如果为素数，程序应返回信息说明其为素数。如果 n 不是素数，程序也应返回信息说明其不为素数. 知果 n 不是一个有效的输入，程序应显示一条帮助信息。

遵循 XP 方法及第 5 章中列举的原则，我们从设计单元测试开始。对于这个应用程序，可以确定出两个具体的任务：确认输入和判断素数。我们可以分别使用黑盒与白盒测试方法、边界值分析方法和判定覆盖准则。然而，极限测试要求使用一个独立的黑盒测试方法，消除所有的偏见。

8.3.1 测试用例设计

设计测试用例首先从确认测试方法开始. 在本例中，我们将使用边界值分析方

法对输入进行确认，因为程序只能接受固定范围内的正整数．所有其他的输入值，包括字符数据类型和负数都会产生错误，不能被使用。当然，我们可以这样处理本例，将输入确认划归到判定覆盖准则中，因为程序必须判断输入是否有效，这里一个重要的概念是，在设计测试时必须确定使用某个测试方法。

使用确定的测试方法，根据可能的输入和预期的输出结果开发一份测试用例列表。表 8 -2 显示了确认的 8 个测试用例（注意：我们采用了一个非常简单的例子来说明极限测试的基本理论。在实际中，会遇到更详细的程序规格说明，可能包含诸如用户界面需求和输出用语措辞等内容。因此，测试用例列表可能会增长很多）。

表 8 -2 check4Prim.java 的测试用例

用例编号	输入	结果	注解
1	n=3	确认 n 为素数	对有效素数的测试 对边界范围内输入的测试
2	n=1,000	确认 n 不为素数	对等于上边界输入的测试 对 n 不为素数的测试
3	n=0	确认 n 不为素数	对等于下边界的测试
4	n=-1	显示帮助信息	对低于下边界的测试
5	n=1,001	显示帮助信息	对高于下边界的测试
6	两个或两个以上输入	显示帮助信息	对输入值得正确个数的测试
7	n=“a”	显示帮助信息	输入未整数而非字符的测试
8	n 为空（空格）	显示帮助信息	对输入为空的测试

表 8-2 中的测试用例 1 结合了两个测试需求。它检查了输入是否为有效的素数，以及程序如何处理有效的输入值。可以在本测试中使用任何有效的素数。附录 B 提供了一份小于 1,000 的可用素数的清单。

使用测试用例 2 同样会测试到两个需求：当输入值等于上边界，或输入值不是素数时会发生什么情况？这个测试用例本可以被分解为两个单元测试，但软件测试总的目标之一是在足以检查出错误的前提下，使测试用例的数量最小。

测试用例 3 检查有效输入的下边界，以及测试无效的素数。此项检查的第二部分是不需要的，因为测试用例 2 已经处理了此需求，但仍然被默认地包括进来，因为 0 不是素数。

测试用例 4 和测试用例 5 保证了输入是在规定的范围之内，即大于 0，且小于

未经同意，严禁以任何形式拷贝

或等于 1000。

测试用例 6 测试应用程序是否可以正确地处理字符输入值。由于我们所做的是数学运算，因此很明显，程序必须拒绝字符类型的输入。该测试用例假定 Jova 会进行数据类型的检查，当输入无教的数据类型时，应用程序必项能够处理发生的异常。该测试用例确保异常得到了处理。

最后，测试用例 7 和测试用例 8 检查输入值的正确个数，任何超过 t1 个的输入都会发生失效。

JUnit 是一个可以免费获得的开源工具，用于在极限编程环境下对 Java 应用程序进行自动化的单元测试。设计者 Kent Beck 和 Erich Gamma 开发 JUnit 来支持极限编程环境下进行的单元测试。JUnit 非常小，但非常灵活，并且功能丰富。我们可以设计单个测试或一系列的测试。可以自动生成报告，详细描述错误的信息。

在使用 JUnit 或任何测试套件之前，须充分了解如何使用它。JUnit 非常强大，但只有掌握了其 API 之后才会发挥作用。然而，无论是否采纳 XP 方法，JUnit 都是一个对代码进行合理检查的有用工具。

访问 www.JUnit.org 可以获得更多的信息，并可下载该测试套件。另外，该站点还提供关于极限编程和极限测试的丰富信息。

8.3.2 测试驱动器及其应用

既然我们已经设计出了两类测试用例，那么就可以生成测试驱动器类 check4primeTest。表 8-3 将 check4PrlmoTest 中的 JUnit 方法与覆盖的测试用例对应起来。

表 8 -3 测试驱动器的方法

方法	检查到的测试用例
`testCheckPrime_true()`	1
`testCheckPrime_false()`	2,3
`testCheck4Prime_checkArgs_char_input()`	7
`testCheck4Prime_checkArgs_above_upper_bound()`	5
`testCheck4Prime_checkArgs_neg_input()`	4
`testCheck4Prime_checkArgs_2_inputs()`	6

未经同意，严禁以任何形式拷贝

testCheck4Prime_checkArgs_0_inputs()	8

注意 testCheckPrime_false()方法测试了两种状态，因为边界值并不是素数。因此，我们可以采用一个测试方法来检查边界值错误和无效素数．仔细检查该方法可以发现，两个测试

```
public void testCheckPrime_false(){
    assertFalse(check4prime.primecheck(0));
    assertFalse(check4prime.primecheck(1000));
}
```

用例确实在其内部被执行到。以下是附录 A 中列举的 check4JavaTest 类中一个完整的 JUuit 方法。

注意，JUnit 方法 assertFalse()对提供给它的参数进行检查，看看是否有误，参数必须是布尔类型，或是一个返回值为布尔类型的函数。如果返回的是“false"，则测试可视为成功。这个程序片段还提供了一个首先设计测试用例和测试配件所带来好处的例子。可以看到 assertFalse()方法中的参数是 check4prime.primecheck(0)方法。这个方法会出现在应用程序的某个类中。首先进行测试设计迫使我们思考该应用程序的结构。从某些方面来讲，应用程序是按照测试配件设计出来的。这里我们需要一个方法来检查输入是否为素数，因此在应用程序中我们将其包括进来。

测试设计结束之后，就可以开始程序编码了。根据程序规格说明，测试用例、测试配件以

```
public class check4prime{
    public static void main(string[] args)
    public void checkArgs(string[] args) throws Exception
    public boolean primeCheck(int num)
}
```

及最后的 java 程序都由单个 check4Prime 类组成，定义如下：

简单地说，根据 Java 程序的定义。main()过程提供了应用程序的入口点。checkArgs()方法判断输入值 n 是个正数，0<=n<=1000。Primecheck()过程对照一个已计算出的素数列表来检查输入值。我们采用 Eratosthenes 筛选法来快速计算素数。由于涉及的素数数量比较小，此方法是可以接受的。

8.4 小结

随着软件产品间竞争的日益激烈，需要将产品非常快速地投向市场。因此人们设计了极限编程方法来支持快速的应用程序开发。这个轻量级的开发过程强调沟通计划和测试。

极限编程中的测试被称为“极限测试”。极限测试的重点在于单元测试和验收测试。一旦代码库发生变更，就需要进行单元测试。在重要的发布结点，由客户来执行验收测试。

极限测试还要求在开始程序编码之前，根据程序的规格说明设计测试配件。在这种方式中，开发的程序要通过单元测试，从而提高程序满足其规格说明的可能性。

未经同意，严禁以任何形式拷贝

词汇表

black-box testing（黑盒测试）将程序视为一个整体、且忽略其内部结构的测试方法。单纯从软件的规格说明中获取测试数据。

bottom-up testing（自底向上的测试）增量模块测试的一种形式，首先测试底层模块，再测试调用模块等等。

boundary-value analysis（边界值分析），一种黑盒测试方法，重点在于程序输 t 入区间的边界区域。

branch coverage（分支覆盖）参见“判定覆盖”。

cause-effect graphing（因果图分析）使用简化的数字逻辑电路图（组合逻辑网络）辅助生成一组高效测试用例的技术。

code inspection（代码检查）一套供小组代码阅读的规程和错误检查技术，作为检查错误的测试周期的一部分，通常使用一份常见错误的列表来对照代码。

condition coverage（条件覆盖）白盒测试的一项准则，要求编写足够数量的测试用例，确保将一个判断中的每个条件的所有可能的结果至少执行一次。

data-driven testing （数据驱动测试）参见“黑盒测试”。

decision/condition coverage （判定 / 条件覆盖）自盒测试的一项准则，要求编写足够数量的测试用例，确保将每个判断中的每个条件的所有可能的结果至少执行一次，将每个判断的所有可能的结果至少执行一次，将每个入口点都至少调用一次。

decision coverage（判定覆盖）白盒测试的一项准则。要求编写足够数量的测试用例，确保每一个判断都至少有一个为真和为假的输出结果。

desk checking（桌面检查）一种将代码审查和走查技术结合起来，在用户桌面上执行程序的技术。

equivalence partitioning（等价类划分）一种黑盒测试技术，其中每个测试用例都必须体现尽可能多的不同的输入情况，以最大限度地减少全部用例的数量。应该尽量将程序输入范围划分为等价类，这样类中某个输入数据的测试结果等同于同类中所有输入数据的测试结果。

exhaustive input testing（穷举输入测试）黑盒测试的一项准则，通过将每个可能的输入条件都作为测试用例，尽量发现程序中的所有错误。

external specification（外部规格说明）从某个相关系统部件用户的角度对程序功能的精确描述。

facility testing（能力测试）系统测试的一种类型，判断目标文档提及的每一项能力

未经同意，严禁以任何形式拷贝

（或功能）是否都实现了。不要混淆能力测试与功能测试。

function testing（功能测试）发现程序与其外部规格说明之间存在不一致的过程。

incremental testing（增量测试）模块测试的一种形式，将待测模块与已测模块组装在一起进行测试。

input/output testing（输入 / 输出测试）参见“，象盒泪 11 试”。

logical-driven testing（逻辑驱动测试）参见“白盒测试”。

multi-condition coverage（多重条件覆盖）白盒测试的一项准则，要求编写足够数量的测试用例，确保每个判定中的所有可能的条件结果的组合，以及所有的入口点都至少执行一次。

nonincremental testing（非增量测试）模块测试的一种形式，每个模块单独进行测试。

performance testing（性能测试）系统测试的一种形式，尽量证明程序不能满足特定的指标，如在特定负载和配置环境下的响应时间和吞吐率。

random-input testing（随机输入测试）在所有可能的输入值中随机选取一个子集来对程序进行测试的过程。

security testing（安全性测试）系统测试的一种形式，用以考验程序或系统的安全保密机制。

stress testing（强度测试）系统测试的一种形式，使程序经受高负载或强度。高强度是指在很短的时间间隔内达到的数据或操作的数量峰值。因特网应用系统通常需要进行强度测试，因为会有大量用户并发访问系统。

system testing（系统测试）高级测试的一种形式，将系统或程序与其初始目标进行比较。为了完成系统测试，需要一套书面的可度量的目标。

testing（测试）为了发现错误而执行程序（或具体的程序单元）的过程。

top-down testing（自顶向下的测试）增量模块测试的一种形式，首先测试初始模块，再测试下一个子模块等等。

usability testing（易用性测试）系统测试的一种形式，测试程序的人机界面。通常要检查的部件包括界面布局、界面色彩、输出格式、输入字段、程序流程、拼写等等。

volume testing（容量测试）系统测试的一种形式，使用大容量的数据检验程序能否处理目标文档中规定数据容量。容量测试与强度测试并不相同。

walkthrough（走查）一套供小组代码阅读的规程和错误检查技术，作为检查错误的测试周期的一部分。通常一个小组的人起到“计算机”的作用，执行一个小的测试用例集。

white-box testing（白盒测试）一种检查程序内部结构的测试类型。

__END__