计算机软件测试技术

郑人杰

清华大学出版社

前　言

正如任何生产过程离不开产品质量检验一样，在计算机软件的开发中测试工作是必不可少的一步。软件测试工作做得怎样，决定着软件产品质量的好坏。然而，人们重视软件测试的原因还不仅于此。事实说明，在软件测试阶段投入的成本和工作量往往要占软件开发总成本和总工作量的一半，甚至更多。尽管这样，测试技术仍然一直落后于人们对它的期望。到现在为止作为软件工程学科中的一个分支，它远未成熟。不仅测试理论，而且已有的测试方法都无法满足当前软件开发的实际需求。这就要求我们密切关注它的发展，继续研究它的规律。

十分遗憾的是在国内大量的出版物中，至今很少见到有关软件测试的技术资料。几年前我们翻译的 Myers 著作(《计算机软件测试技巧》，清华大学出版社，1985)。在普及测试技术方面起过一定作用。近年来，在完成教学和科研任务的过程中，我们围绕着软件测试开展了调查、研究、分析和实践活动，同时也积累了一些资料。我们愿意把这些资料整理出来，与同行们共享，使其发挥应有的作用。

由于本书编写的时间十分紧迫，有些部分深入研究得不够，其中的错误和不妥之处在所难免。我们真诚地希望读者批评指正，批评意见请寄北京清华大学软件技术中心(邮政编码：100084)。

参加本书编写工作的还有姜凡(第一章的软件测试发展的历史回顾、第四章的功能测试、第五章的测试覆盖准则、域测试及程序变异)，徐仁佐(第九章的程序可靠性模型及软件可靠性在软件测试中的应用)，马素霞(第八章程序正确性证明)，殷人昆(第四章的正交实验设计法)和刘挺(第七章 COSTE 简介)。郑晨参加了部分书稿的誊写。在此对他们的辛勤劳动和密切配合表示衷心的感谢。

郑人杰

1990 年底于清华园

目　录

第一章 绪 论

本世纪中计算机刚一问世，尽管当时还没有软件这一名称，但软件的重要组成部分——计算机程序就开始为我们服务了。然而，计算机软件作为一种人类创造的复杂逻辑实体和人们长期以来已经熟悉的大多数产品有着许多不同的特点。认识和掌握这些特点需要大量的实践和研究。只是70年代以来形成了软件工程的概念，才使软件产业得以初步建立。

软件测试是保证软件质量的重要手段。本章将对软件生存期、软件测试的目的和意义、软件测试的发展简史以及应该如何正确对待软件测试的心理学等问题作一概括的描述。其目的在于，使读者在着手研究和掌握具体的测试技术以前，对软件测试建立起正确的、全面的基本概念；同时澄清在用户甚至在计算机界中仍然流行着的一些错误的和有害的观点。

为了便于读者进一步地研究，本章提供了十分详尽的参考资料目录。

1.1 软件危机和软件生存期

计算机软件在近30多年来经历了曲折的发展道路。在这期间值得一提的是在60年代出现的"软件危机"。

我们知道，随着计算机硬件技术的进步，计算机的元器件质量逐步提高，整机的容量、运行速度以及工作可靠性有了明显的提高。与此同时，硬件生产的成本降低了。计算机价格的下跌为它的广泛应用创造了极好的条件。在此形势下自然要求软件与之相适应。一方面适应高速度、大容量、高可靠度的高性能硬件；另一方面要适应在广泛应用情况下出现的大型、复杂问题对软件技术提出的迫切需求。然而，事实上软件技术的发展未能满足这些需求。和硬件技术的快速发展相比，软件的确大大地落后了。多年来由于问题未得到及时解决，致使矛盾日益尖锐。这些矛盾归结起来主要表现在以下几个方面：

① 由于缺乏大型软件开发的经验和软件开发数据的积累，使得开发工作的计划很难制定。主观盲目地制定计划，执行起来和实际情况有很大差距，使得经费使用常常突破预算。由于工作量的估计不够准确，进度计划难以遵循，开发工作完成的期限一再拖延。延期的项目，为了尽快赶上去，便要增加人力，结果适得其反，不仅进度未能加快，反而更加延误了。在这种情况下，软件开发的投资者和用户对开发工作从不满意发展到不信任。

② 作为软件设计依据的软件需求，在开发的初期提得不够明确，或是未能做出确切的表达。开发工作开始后，软件人员又未能和用户及时交换意见，使得一些问题得不到及时解决而隐藏起来，造成开发后期矛盾的集中暴露。这时对多个错综复杂的问题既难于分析，又难于解决。

③ 开发过程中没有遵循统一的、公认的方法论或是开发规范，参加工作的人员之间

的配合不够严密，约定不够明确。加之不重视文字资料工作，使得开发文档很不完整。发现了问题，未能从根本上去找原因，只是修修补补。显然，这样开发出的软件无法维护。

④ 缺乏严密有效的软件质量检测手段，交付给用户的软件质量差，在运行中暴露出各种各样的问题。在各个应用领域的不可靠软件，可能带来不同程度的严重后果。轻者影响系统的正常工作，重者发生事故，甚至酿成生命财产的重大损失。

这些矛盾表现在具体的软件开发项目上。最为突出的实例便是美国 IBM 公司在 1963 年至 1966 年开发的 IBM 360 机操作系统。这一项目在开发期中每年花费五千万美元，总共投入的工作量为 5 千人-年。参加工作最多时有 1 千人。总共写出了一百万行源程序。尽管如此多的开销，却拿不到开发成果。这是一次失败的记录。项目负责人 F. P. Brooks 事后总结了他在组织开发过程中的沉痛教训，写成《神秘的人-月》一书。这个反映软件危机的典型事例成为软件技术发展过程中一个重要的历史性标志。

从上述软件危机的现象和发生危机的原因分析，要摆脱危机不是一件简单的事，要从多方面着手解决，把握好软件的特点，抓住它与其它产业部门工作对象的相同与相异之处加以对比分析，排除人们的一些传统观念和某些错误认识是非常重要的。

除去那些规模很小的项目以外，通常开发一个软件要在不同层次的多个开发人员的配合与协作中完成；开发各阶段之间的工作应当有严密的衔接关系；开发工作完成以后，软件产品应该面向用户，接受用户的检验。所有这些活动都要求人们根本改变那种把软件当作个人才智产物的看法，抛弃那些只按自己的工作习惯，不顾与周围其它人员密切配合的作法。事实上，这里所列举的情况与研制计算机硬件，甚至与完成一项建筑工程项目并没有本质的差别。任何参加工程项目的人员，他的才能只能在工程的总体要求和技术规范的约束下充分发挥和施展。既然我们已经积累了几千年的工程学知识，能不能把它运用在软件开发工作上呢？实践表明，按工程化的原则和方法组织软件开发工作是有效的，也是摆脱软件危机的一个主要出路。

参照工程学的概念，研究软件工程工作的特点进一步改变了原来受到束缚的传统观念。当我们全面分析软件开发工作的各个“工序”时，认识到程序编写只是整个工作的一部分。在它的前后还有更重要的工序。正如同其它事物一样，从它的发生、发展到达成熟阶段，以至老化和衰亡，有一个历史发展的过程，任何一个计算机软件有它的生存期（Life Cycle）。这个生存期包括 6 个步骤，即：

① 计划（Planning）

② 需求分析（Requirement Analysis）

③ 设计（Design）

④ 程序编写（Coding）

⑤ 测试（Testing）

⑥ 运行与维护（Run and Maintenance）

这些步骤的主要任务是：

① 计划：确定软件开发的总目标；给出软件的功能、性能、可靠性以及接口等方面的设想。研究完成该项软件任务的可行性，探讨问题解决的方案；对可供开发使用的资源（如计算机硬、软件、人力等）、成本、可取得的效益和开发的进度作出估计；制定完成开发

任务的实施计划（Implementation Plan）。

② 需求分析：对开发的软件进行详细的定义，由软件人员和用户共同讨论决定，哪些需求是可以满足的，并且给予确切的描述；写出软件需求说明书（Software Requirement Specifications）或称软件规格说明书，以及初步的用户手册（System User's Manual），提交管理机构审查。

③ 软件设计：设计是软件工程的技术核心。在设计阶段应把已确定的各项需求转换成相应的体系结构，在结构中每一组成部分是功能明确的模块。每个模块都能体现相应的需求。这一步称为概要设计（Preliminary Design）。进而进行详细设计（Detail Design），即对每个模块要完成的工作进行具体的描述，以便为程序编写打下基础。上述两步设计工作均应写出设计说明书，以供后继工作使用并提交审查。

④ 程序编写：把软件设计转换成计算机可以接受的程序，即写成以某个程序设计语言表示的源程序清单。这一工作也称为编码。当然，写出的程序应该是结构良好、清晰易读，且与设计相一致的。

⑤ 测试：测试是检验开发工作的成果是否符合要求，它是保证软件质量的重要手段。通常测试工作分为三步，即：

单元测试（Unit Testing）——单独检验各模块的工作。

集成测试（Integrated Testing）——将已测试的模块组装起来进行检验。

确认测试（Validation Testing）——按规定的需求，逐项进行有效性测试，以决定开发的软件是否合格，能否提交用户使用。

⑥ 运行和维护：已交付用户的软件投入正式使用以后便进入运行阶段。这阶段可能持续若干年，甚至几十年。在运行中可能有多种原因需要对它进行修改。其原因包括：运行中发现了软件中有错误，需要修正；为了适应变化了的软件工作环境，需要作相应的变更；为进一步增强软件的功能，提高它的性能，使它进一步完善和扩充。

以上 6 步表明每个软件从它的酝酿开发，直至使用相当长时间以后，被新的软件代替而退役的整个历史过程。按此顺序逐步转变的过程可用一个软件生存期的瀑布模型加以形象化描述。图 1.1 给出了这一瀑布模型。从图中可看出，从左上到右下如同瀑布流水

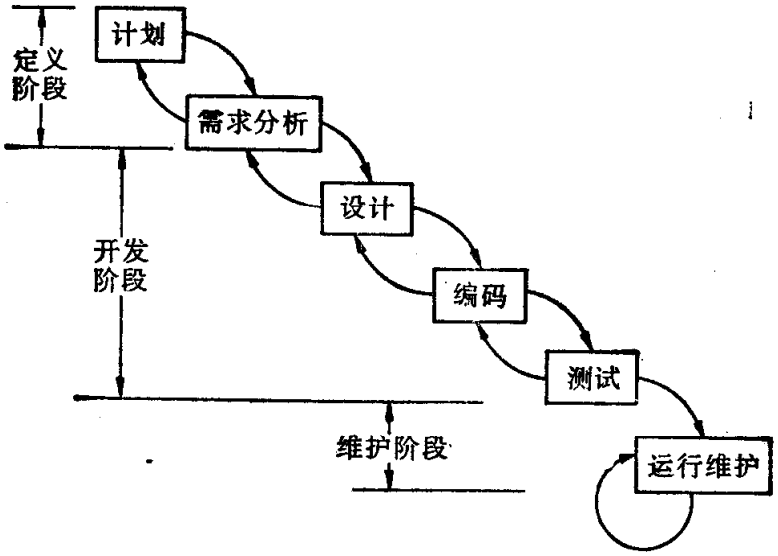


图 1.1 软件生存期的瀑布模型

逐级下落。在最后的运行中可能需要多次维护，图中用环形箭头表示。此外，在实际的项目开发中，为了确保软件的质量，每一步骤完成以后都要进行复查，如果发现了问题就要及时解决，以免问题积压到最后造成更大的困难。每一步骤的复查及修正工作正是图中给出的向上箭头。此外，图中还指明了6个步骤划分的3个阶段：软件定义阶段、软件开发阶段及软件维护阶段。

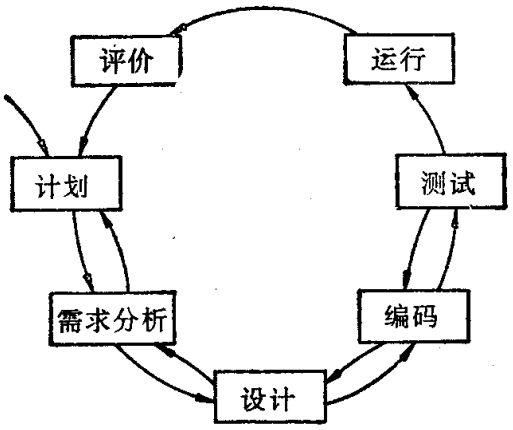


图 1.2　软件生存周期

值得注意的是，上述软件维护工作不可简单地看待。原因在于维护工作不仅仅是修改程序。在软件运行的过程中若有必要修改，得提出充分的修改理由，经过审核，才能肯定下来。接着需要经历制订修改计划、确定新的需求、修改软件设计、修改编码、进行测试以及重新投入运行等一系列步骤。这些步骤正是上述开发一个新软件的步骤。若是运行中多次提出修改，将多次经历这些步骤。我们把这一过程用图1.2来表示，并称此为软件生存周期。实际上软件生存周期和软件生存期表达的是同一内涵，为简便起见通常只称为软件生存期。

1.2　软件测试的意义

软件测试在软件生存期中占有非常突出的重要位置，究其原因是多方面的。根据 Boehm 的统计，软件开发总成本中，用在测试上的开销要占 30% 到 50%。如果我们把维护阶段也考虑在内，讨论整个软件生存期，开发测试的成本比例会有所降低，但不要忘记，维护工作相当于二次开发，乃至多次开发，其中必定也包含有许多测试工作。因此，有人估计软件工作有 50% 的时间和 50% 以上的成本花在测试工作上。

软件测试究竟是什么意思，它包括哪些工作？这些问题常常被一般人误解，甚至从事计算机工作的人员也可能弄不清楚。

测试（Test）一词最早出于古拉丁字，它有“罐”或“容器”的含义。人们当时用一种特殊的容器检验金属中含有某种元素是多少。到现在测试一词已经普遍使用了，在工业生产和制造业中测试被当作一个常规的生产活动，它常常和产品的质量检验密切相关。在这些行业中测试的含义似乎是明确的，但在计算机软件领域内则不然。比如，什么是程序测试，什么是软件测试，它们之间有什么差别？测试和调试是一回事吗？它们之间又有什么差别？对于这些概念的不同解释可能会涉及到测试的目的和方法。

“程序测试”的说法最早几乎是和“程序编写”同时出现的。从当时的观点来看，谁写出的程序，谁去测试它，谁去使用它，测试被当作编写程序以后很自然要做的一步工作。并且在测试时不仅要发现程序里的错误，而且要排除错误。这时测试与调试混为一谈，完全不加区分。一段时间里人们谈论和写文章所涉及到的测试一词，实际上是调试即排错（debug）（请参看本节最后两段）。其实这两者是完全不同的两个概念。我们还注意到，那时的计算机多用来完成数值计算任务。程序运行后所得的计算结果常常采用以手

工计算其中一部分数据的办法来比较，从而判断程序的正确性。这在当时还是简单易行的。但随着计算机应用领域的拓广，所写程序要解决的问题也不仅仅限于数值计算了。上述手工计算进行校验的办法难于适应了，然而对于程序正确性检验的基本模式并没有改变。这就是给出测试数据，运行被测程序，将所得结果与预期结果进行比较，从而判断程序的正确性。我们称此为程序正确性测试(参看图 1.3)。

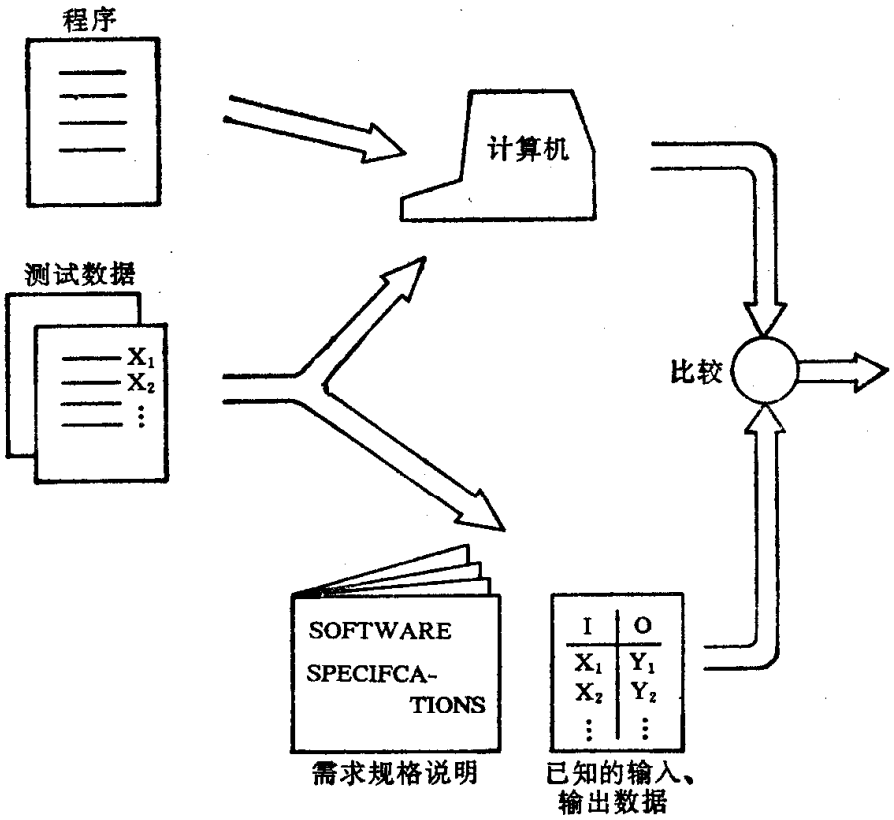


图 1.3　程序正确性测试

70 年代中以来，形成了软件生存期概念。这时人们对于软件测试的认识更广泛也更深刻了。这对于软件产品的质量保障以及组织好软件开发工具有着重要的意义。首先，由于能够把整个开发工作明确地划分成若干个开发步骤，就把复杂的问题按阶段分别加以解决。使得对于问题的认识与分析、解决的方案与采用的方法以及如何具体实现在各个阶段都有着明确的目标。其次，把软件开发划分成阶段，就对中间产品给出了若干个监控点，提高了开发过程的可见度，为各阶段实现目标的情况提供了检验的依据。各阶段完成的软件文档成为检验软件质量的主要对象。这时对软件质量的判断决不只限于程序本身。即使只谈程序本身的正确性，它也和编码以前所完成的需求分析及软件设计工作进行得如何密切相关。很显然，表现在程序中的错误，并不一定是编码所引起的。很可能是详细设计、概要设计阶段，甚至是需求分析阶段的问题引起的。因此，即使针对源程序进行测试，所发现的问题其根源可能在开发前期的各个阶段。解决问题、纠正错误也必须追溯到前期的工作。正是考虑到这一情况，我们通常把测试阶段的工作分成若干步骤进行。这些步骤包括：模块测试，集成测试、确认测试和系统测试。对程序的最小单

Unit Integrated
Testing testing

位——模块进行测试时，检验每个模块能否单独工作，从而发现模块的编码问题和算法问题；进而将多个模块联结起来，进行集成测试，以检验概要设计中对模块之间接口设计的问题；确认测试则应以需求规格说明书中的规定作为检验尺度，发现需求分析的问题；最后进行的系统测试是将开发的软件与硬件和其它相关因素（如人员的操作，数据的获取等）综合起来进行全面的检验，这样的作法必将涉及到软件的需求以及软件与系统中其它方面的关系。图1.4给出了测试工作与软件开发前期工作的关系。图中开发工作是自上而下进行的，而几种不同的测试都会涉及到前期工作的不同阶段。

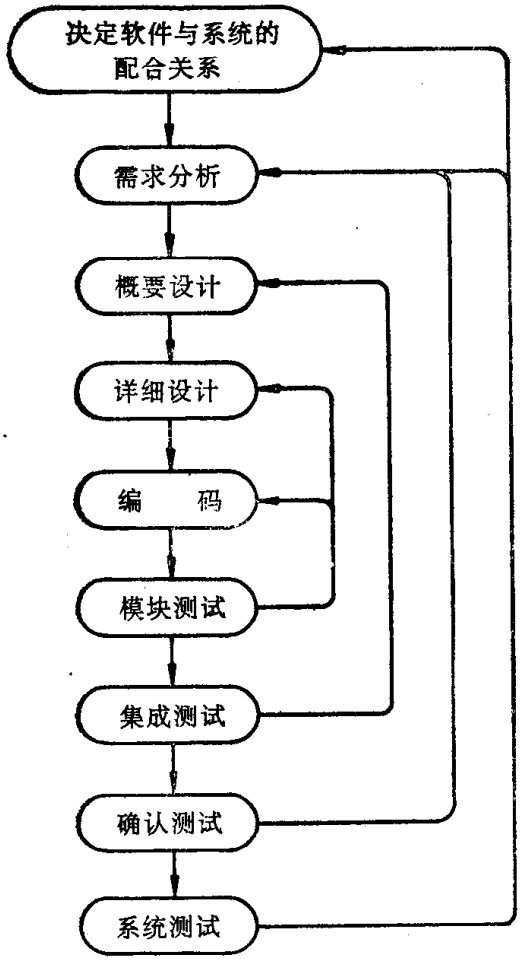


图1.4　测试与开发前期工作的关系

如果我们着眼于整个软件生存期，特别是着眼于编码以前各开发阶段的工作来保证软件的质量，就需要突破原来对测试的理解。也就是在开发的过程中，需要不断地复查与评估、不断地进行检验，以利于把发现的错误和问题得到及时的解决，而不让这些错误和问题隐藏起来，影响后期的开发工作。总之，贯穿在整个开发各阶段的复查、评估与检测活动，远远地超出了程序测试的范围，可以统称为确认、验证与测试活动（V，V&T——Validation, Verification and Testing）有时为了方便，也简称为测试，不过这是广义的测试概念。

所谓确认，它是指如何决定最后的软件产品是否正确无误。比如，编写出的程序相对于软件需求和用户提出的要求是否符合，或者说程序输出的信息是用户所要的信息吗？这个程序在整个系统的环境中能够正确稳定地运行吗？这里自然包含了对软件需求满足程度的评价。在软件产品开发完成以后，为了对它在功能、性能、接口以及限制条件等方面是否满足需求作出切实的评价，需要在开发的初期，在软件需求规格说明书中明确地规定确认的标准。

所谓验证，它是指如何决定软件开发的每个阶段、每个步骤的产品是否正确无误，并与其前面的开发阶段和开发步骤的产品相一致。验证工作意味着在软件开发过程中开展一系列活动，旨在确保软件能够正确无误地实现软件的需求。

确认和验证是有联系的，但也有明显的差别。Boehm 在1981年是这样来描述两者差别的：确认要回答的是：我们正在开发一个正确无误的软件产品吗？而验证要回答的是：我们正开发的软件产品是正确无误的吗？

总之，确认、验证与测试在整个软件开发过程中作为质量保证的手段，应当最终保证软件产品的正确性、完全性和一致性。我们可以把确认、验证与测试活动分为三类（见图

1.5):

① 完整性检验——验证每一开发阶段(或开发步骤)中产品的完整性；分析每一产品,确保其内部的一致与完全。例如,分析需求规格说明书，以找出不一致的需求或矛盾的需求。比如,其中规定输出报告,但该报告所需要的数据是无法得到的。

② 进展检验——保证各个开发阶段(或开发步骤)之间其规格说明书的完全性和一致性。例如,后一阶段的工作确是前阶段工作的进一步细化。

③ 适用性与充分性检验——把取得的结果与对问题的理解作比较,保证所完成的结果是必要而充分的解。

在整个软件生存期各阶段中确认、验证与测试活动包括:

① 需求分析阶段

• 制定项目的 V,V&T 计划: 确定 V,V&T 的目标; 安排 V,V&T 的活动;选择采用的方法和工具;制定进度并作出预算。

• 确定与测试用例相关的需求: 这些需求构成了一组基础的测试用例。从而有助于澄清并且确定软件需求的可度量性,同时也作为验收测试的基础。

• 复审并分析需求: 其目的在于确保规定的需求能够对整个问题取得有指望的和可用的结果,针对问题叙述的清晰性、完全性、正确性、一致性、可测试性和可跟踪性进行复审。

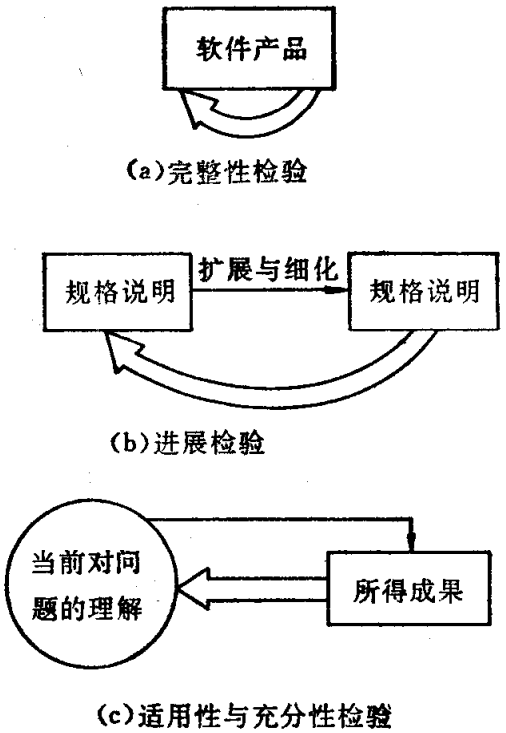


图 1.5 三类确认、验证与测试活动

② 概要设计阶段

• 复审并修订 V,V&T 计划。

• 针对要执行的逻辑功能而生成的测试数据,补充软件需求。

• 复审并分析概要设计: 确保内部的一致性、完全性、正确性及清晰性; 检验已进行的设计是否满足需求。

③ 详细设计阶段

• 针对功能测试数据进行设计: 设计要考虑到基于系统物理结构的测试数据。

• 复审并分析详细设计规格说明: 确保内部的一致性、完全性、正确性及清晰性; 检验详细设计是否对概要设计做出了正确无误的细化;确认所做的设计满足于需求。

④ 编码及测试阶段

• 完成测试用例规格说明: 针对编码过程中对设计的修改补充或修正测试用例规格说明。

• 复审、分析并测试程序: 检查是否遵循了编码标准; 自动或手工分析程序;运行测试用例,以保证满足验收要求。

• 进行产品验收。

⑤ 运行及维护阶段

• 软件评估：对软件的运行情况作出评估，以保证它能继续满足用户的需求。

• 软件修改评估：当提出修改的要求时也要进行评估，修改以后要进行复审和测试，以确保修改正确无误。

• 回归测试：重新运行前已正确无误的测试用例，以便消除由于软件修改而带来的各种错误。

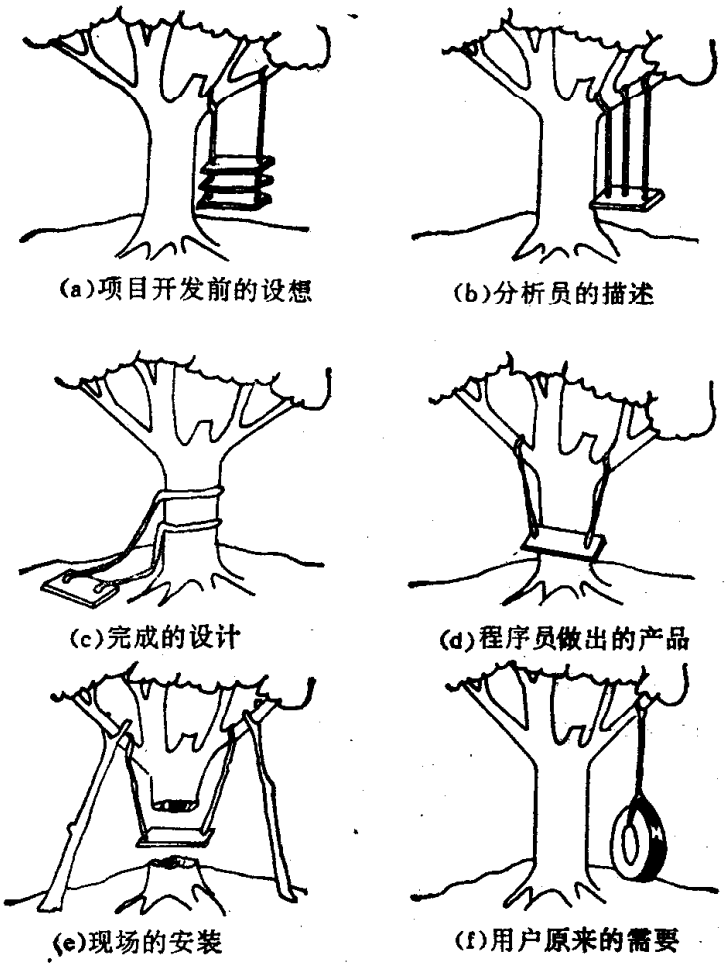


图 1.6 软件开发面临的实际问题

最后，为了较为形象地描述软件开发面临的实际问题，请读者参看图 1.6[注]。虽然这只是一个比喻，但的确在一定程度上反映了实际情况。软件项目的实践一再告诉我们，为了确保软件产品能够符合用户的需要，必须着眼于整个软件生存期，在每个开发阶段采取措施，进行各个阶段的 V，V&T 活动。使之不致在开发完成后，发现和用户的需求有如此大的差距。

在这一节的最后，我们讲一下关于排错（debug）一词的来源。程序中排错，我们现在都称为 debug，其原始意义是捉住小虫（bug）。这里有关于该词的一件轶事，它对理解排错与测试的区别是有益的。

1945 年夏在美国弗吉尼亚某地海军水上武器研究中心运行着 MarkII 计算机，这是以继电器为元件的老式计算工具。由于没有空调设备，夏夜中的机房很热。当时正值大战期间，计算任务十分繁忙。可是 MarkII 突然停止了工作，在多方查找后发现了原因：一只飞蛾从窗外进入，落在继电器的触点上。电磁式继电器触头将其打扁，致使电路中断而停机。机务人员捉到飞蛾，放于机器运行日志，并记载了这一情况。G. M. Hopper 为此创一新词，把排除机器运行的故障统称为“捉虫”——debugging。此后，人们也用该术语称呼程序排错。其实，它和测试一词的含义完全不同。

1.3 什么是软件测试

经过了多年的软件开发实践，积累了许多成功的开发经验，同时也总结出不少失败的

注：图1.6引自伦敦大学《计算中心通讯》，第 53 期(The University Computer Centre Newsletter, No. 53)。

教训。在此过程中，软件测试的重要意义逐渐被人们普遍认识。看到软件测试在开发成本中占有如此大的比例，同时它又是保证软件质量的主要手段，因而逐渐受到重视。然而，究竟什么是软件测试，这一基本概念很长时间以来存在着不同的观点。一些人为软件测试给出了定义，但由于强调的方面不完全一致，至今难于给出统一论述。即使那些公认的测试定义，也还存在问题。特别是目前仍然有许多人对于"什么是软件测试?"持有不正确的观点，这也恰恰是不能很好地做好软件测试工作的原因。

回答这一问题的典型说法有：

- 对照规格说明检查程序；
- 找出程序中的隐藏错误；
- 确定用户接受的可能性；
- 确认系统已经能够提供使用了；
- 取得该软件已能工作的信心；
- 表明程序执行得正确无误；
- 表明程序中错误并未出现；
- 理解程序运行的限制；
- 弄清该软件不能做什么；
- 检验该软件的能力；
- 验证软件文档；
- 使自己确信开发任务已经完成；

等等。

必须承认，以上这些说法并非都错，有的也有其正确的方面，但毕竟含有缺陷。

1973年 W. Hetzel 曾经指出，测试是对程序或系统能否完成特定任务建立信心的过程。这种认识在一段时间内曾经起过作用，但后来有人提出异议。认为我们不应该为了对一个程序建立信心或显示信心而去作测试。此后他又修正了自己的观点，他说："测试是目的在于鉴定程序或系统的属性或能力的各种活动，它是软件质量的一种度量"。这一定义实际上是使测试依赖于软件质量的概念。但软件质量又是什么呢？对于很多从事实际工作的人员来说，质量一词和测试一样抽象，也难以捉摸。事实上，尽管我们可以为软件质量的含意给出确切的解释，但影响软件质量的因素也不是一成不变的。在某一环境下得到的高质量产品，在换了另一环境以后，很可能变成低质量的，甚至是完全不适用的。如果我们把质量理解成"满足需求"，那将是可以接受的。假定软件项目的需求是完全的，开发出的产品又能满足这些需求，那就是高质量的软件产品。1983年 IEEE 提出的软件工程标准术语中给软件测试下的定义是："使用人工或自动手段来运行或测定某个系统的过程，其目的在于检验它是否满足规定的需求或是弄清预期结果与实际结果之间的差别"。这就非常明确地提出了软件测试以检验是否满足需求为目标。

G. J. Myers 则持另外的观点，他认为："程序测试是为了发现错误而执行程序的过程"。这一测试定义明确指出"寻找错误"是测试的目的。相对于"程序测试是证明程序中不存在错误的过程"，Myers 的定义是对的。因为把证明程序无错当作测试的目的不仅是不正确的，是完全做不到的，而且对做好测试工作没有任何益处，甚至是十分有害

的(关于这一点我们在本章下一节还要进一步说明)。从这方面讲,我们接受 Myers 的定义以及它所蕴含的方法论和观点。不过,这个定义规定的范围似乎过于狭窄,使得它受到很大限制。因为如前所述,除去执行程序以外,还有许多方法去评价和检验一个软件系统。按照 Myers 的定义,测试似乎只有在编码完成以后才能开始。另一方面,"测试"归根结底包含"检测"、"评价"和"测验"的意思,这和"找错"显然是不同的。

以上讨论的软件测试定义都是强调软件的正确。有些测试专家认为软件测试的范围应当包括得更广泛些。J. B. Goodenough 认为测试除了要考虑正确性以外,还应关心程序的效率、健壮性(robustness)等因素,并且应该为程序调试提供更多的信息。他列出了一张测试组成表(图 1.7)。

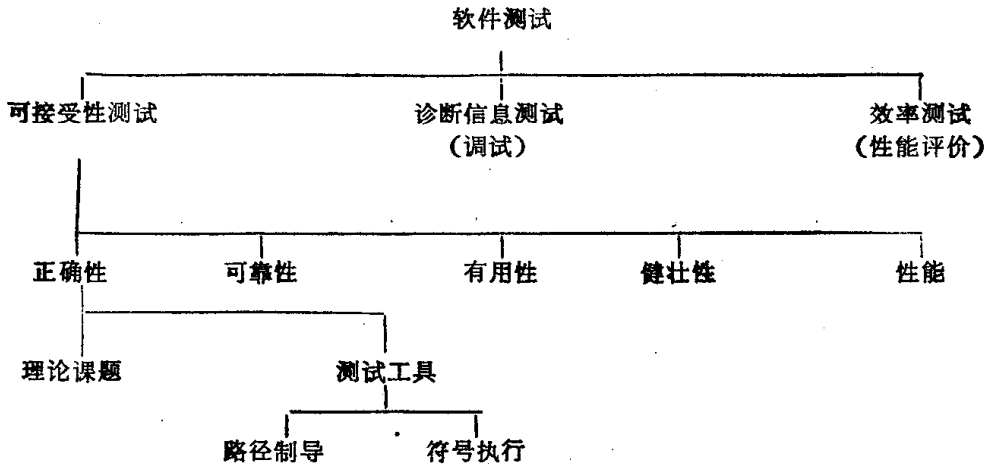


图 1.7 Goodenough 关于软件测试的定义

S. T. Redwine 认为,软件测试应该包含以下几种测试覆盖:

① 功能覆盖;

② 输入域覆盖;

③ 输出域覆盖;

④ 函数交互覆盖;

⑤ 代码执行覆盖;

他还给出了检查表。

至于测试的范围,A. E. Westley 将测试分为四个研究方向,即:

① 验证技术(目前验证技术仅适用于特殊用途的小程序);

② 静态测试(应逐步从代码的静态测试往高层开发产品的静态测试发展);

③ 测试数据选择;

④ 测试技术的自动化。

为了进一步明确测试的范围和具体目标,G. J. Myers 和 B. Beizer 都详细列出了各种软件错误的类型,并指出软件测试的目的就是要找出这些错误。

1.4 应该怎样认识软件测试

如何正确地认识和对待软件测试常常是做好软件测试工作的前提。事实上,到现在

为止仍然有一些不正确的看法和错误的态度，在不同程度地妨碍着测试工作的开展。这包括：

·认为测试工作不如设计和编码那样具有开拓性，也不容易看到进展。在测试中花了多大力气也很难让人看到。这是没有真正认识到软件测试的意义，如果以这种认识指导工作，那是非常有害的。

·以发现软件错误为目标的测试是非建设性的，甚至是破坏性的。简单地以为软件错误都属于责任事故，因此测试中发现了错误是对责任者工作的一种否定。其实，建设性与破坏性之间存在着辩证关系，并且追究错误的个人责任通常无益于问题的根本解决。

·"测试工作枯燥无味，不能引起人的兴趣"。持这种观点的人常常缺乏耐心，也没有认清软件测试的重要意义。

·测试工作的确是艰苦而细致的工作。但有人不愿在这上面花力气，指望着碰运气过关，这当然做不好测试工作。

·"这程序不会有问题，因为是我写的"。实际上，这是没有根据的盲目自信。在发现了程序错误以后，他们便立刻会顾虑别人对自己的看法。这常常是开发小组人员之间配合不好而影响工作开展的原因。要求人们的思维和他们的行为完全附合客观规律，而不犯任何错误是不可能的。一段时间以来，人们常说：要允许别人犯错误，也要允许别人改正错误。这才是我们对待错误的正确态度。

为了澄清认识和端正态度，有必要对以下几个方面的问题加以说明。

一、能够彻底测试程序吗？

如果我们认为测试的目的在于查找错误，并且找出的错误越多越好，很自然就会提出这样的问题：能不能把所有隐藏的错误全部找出来呢？或者问：能不能把所有可能做的测试无遗漏地一一做完，来找出所有的错误呢？

以下举两个例子进行具体的分析，从中可看出"穷举测试"的不现实性。

若一程序 P 有输入量 X 和 Y，并有输出量 Z，在字长为 32 位的计算机上运行。如果 X 和 Y 均只取整数，考虑把所有可能的 X 值和 Y 值作为测试数据，逐个用以驱动程序 P，进行穷举测试。力图全面、无遗漏地"挖掘"出程序中的所有错误。

这样做可能采用的测试数据组 X_i 和 Y_i，其中 i 的最大值为：

$$2^{32}\times 2^{32}=2^{64}\approx 10^{20}$$

如果程序 P 测试一组 X、Y 数据需用 0.001 秒，要完成 10^{20} 组测试，共需 5 亿年的时间！

若另一程序 Q 有四个条件判断（图 1.8 中的 d_1、d_2、d_3 和 d_4）和一个最多重复 20 次的循环。在循环体内有五条路径。若考虑到循环，从入口到出口总计有路径数为：

$$5^{20}\approx 10^{14}$$

假定我们为每条路径设计一组测试数据，再对其实施测试。如果这一过程需时两分钟，则测试完 10^{14} 条路径也要用 5 亿年的时间。

以上两种情况说明，由于穷举测试工作量过大，需用时间过长，致使实施成为不现实，因而也就失去了实用价值。

我们知道，软件工程的总目标是充分利用有限的人力和物力资源，高效率、高质量地

完成软件开发项目。在测试阶段既然穷举测试是不可实现的，为了节省时间和资源，提高测试效率，就必须精心设计测试用例，使得采用这些测试数据能够取得最佳的测试效果。但这又如何掌握呢？

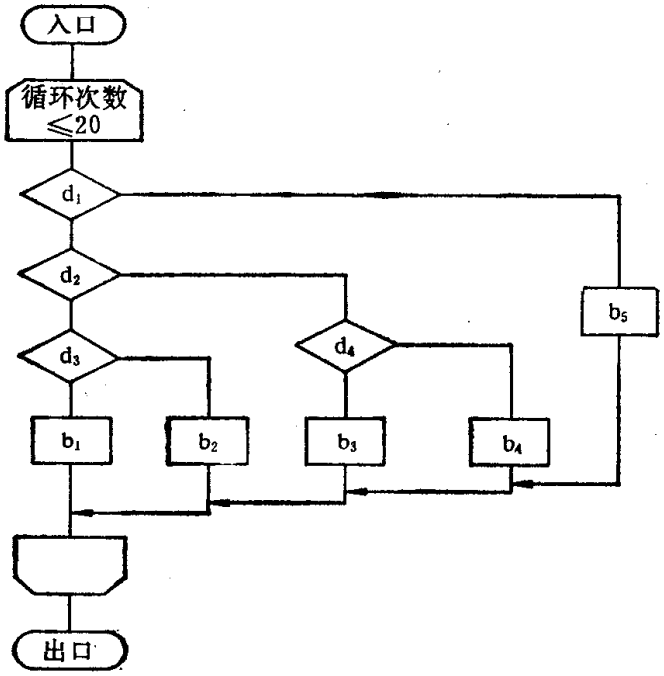


图 1.8　具有四个判断和一个循环的程序

一位有经验的软件开发管理人员在谈到软件测试时曾这样说过："不充分的测试是愚蠢的，而过度的测试则是一种罪孽。"其原因在于，不足测试势必使软件带着一些未揭露的隐藏错误投入运行，这将孕育着更大的危险要让用户承担。过度测试则会浪费许多宝贵的资源。到测试阶段的后期，即使找到了错误，然而付出了过高的代价。关于这一点，我们将在第六章测试管理中作进一步分析。

不过，从这里可以看出，软件测试有着一个不可克服的突出欠缺。即通常我们不可能找出程序中所有的错误，也就无法证实程序是完全正确的。也就是说，既然无法做到"彻底"的排错，当然不能说明程序中没有错误。E. W. Dijkstra 在 1972 年曾说过一句名言："程序测试只能表明错误的存在，而不能表明错误不存在"。

软件测试在当前软件开发中的重要地位仍然是其它质量保证手段所不能代替的。人们寄希望于软件测试，力图在测试中找出尽可能多的错误，这个愿望也是很自然的。然而，上述软件测试的欠缺是不可克服的，无论怎样违背了人们的主观愿望，但这毕竟是客观的现实。

二、证实程序的正确性是测试的目的吗？

有人认为，测试的目的在于证实程序的正确性，测试是为了说明程序是没有问题的。因此，在程序编写完以后，随便找几个数据，使程序走通便认为程序没有问题了，也就达到测试目的。这种认识在观念上是错误的，在实践上则是十分有害的。因为，若是出于这一目的，人们会自觉或不自觉地寻找容易使程序通过的测试数据，回避那些易于暴露程序错误的数据。致使隐藏的错误无法发现，自然也就得不到排除。与此相反，如果我们测试活动的目标始终围绕着揭露程序中的错误，那么在选取测试数据时，自然要考虑那些易于发现程序错误的数据。并且认为，能够发现程序错误的数据是好的数据；能够高效率揭露程序错误的测试是成功的测试。而持相反观点的人必然认为那些是坏的数据，找出程序错误的测试是失败的测试。

前去医院看病的病人总是希望早些作出诊断，弄清病因，以利于对症施治。但如果病人讳疾忌医，在医院里虽然进行了一些检查，但未作出诊断，或是随便作了一些检查，就被告知没病。病人自己可能一时心欢，实际上很可能蕴蓄着危险的后患。就医院来说，这不

是对病人负责任的做法。

心理学研究告诉我们，人们常常愿意看到想要看到的事物。在根据观察到的事物作出判断时，他的期望、观察的动机都会影响到观察的结果。1966 年 Green 和 Swets 曾作过很有典型意义的信号侦察研究试验。他们要求受试者注视雷达屏幕，在发现接收信号时，立即报告。试验结果表明，做到准确无误地报告是很难的。原因在于，如果受试者期待着能够找到许多目标，或是当他感到由于报告找到目标而得到高额奖酬时，他会发现并报告许多目标。其实其中有些并不是真的目标信号。与此相反，如果受试者得知，实际上并没有很多目标信号，或是报告不真实信号要受罚时，他常常会疏漏一些确已在屏幕上出现的目标信号。实验心理学家用了 80 年艰辛的实验，认识到在这类实验中不要再去责备那些犯有错误的受试者。同时还认识到，受试者本人的心态以及如何组织安排实验决定了错误报警和疏漏的比率。

软件测试工作又何尝不是这样呢？如果你能在测试中找出许多错误，并因此而受到赞扬，那你定会努力找错。甚至在找到的错误中也许还包含假的错误。另一种情况是，如果你一直期望某个程序能正确无误地工作，或是如果因找出了错误受到别人抱怨，也许因为你报了假错而受罚，那你定会疏漏一些真的错误。对于以上两种情况我们宁愿要前者。因为我们总是希望通过测试把错误找出来，而不愿让它隐藏下来。也正是这个原因我们一直认为，前节提到的 Myers 的观点有它的合理性和积极作用。

在我们前面说的以去医院就诊做比喻时，也许有人提出疑问，病人前往就诊是以有病为前提的，而写出的程序能否肯定其中必定有错误呢？下面的讨论对回答这一问题是有益的。

三、软件错误可以避免吗？

面对软件测试工作所遇到的一些困难，希望最好有一种全新的软件开发技术能够代替目前广泛采用的传统开发方法，从而避免软件生存期的开发方式，也就从根本上避开了产生软件错误的根源。但这毕竟是我们为之奋斗的前景。当前，我们仍然必须沿着软件生存期的瀑布模型逐级推进。与此同时在各阶段的开发工作中，不断地与发生的各种错误作斗争，以利于最终得到高质量的软件产品。

必须看到，开发过程中软件错误之所以不可避免，从客观上说，是由于所开发的软件具有相当的复杂性。特别是软件的抽象性，它是看不见、摸不着的逻辑实体。发现错误和纠正错误都不是容易作到的事。从主观方面讲，则是由于人们思维的局限性。即使是经验丰富而又细心的软件人员也不能保证他们的思维是绝对周密的，在他们参与的工作中不出现任何差错。软件开发的实践也一再表明，指望整个开发过程完全不出现人为的差错是不可能的，问题在于如何能够及时地发现和排除各种隐匿的差错。

就软件开发的几个阶段而论，经历了一系列步骤，各个步骤之间的衔接会有许多出现差错的可能。开发结束后，得到的软件在机器上运行的结果是否能够令用户满意，依赖于各个步骤的工作，特别是各步骤之间的各种正确性是否能够保持。但实际情况常常告诉我们，所有正确性均能保持是非常困难的，就是说，在软件开发过程中完全避免出错是难以做到的。请读者参看图 1.9，其中表明了软件开发经历的若干主要环节，以及这些环节

之间接口处需要保证的各种正确性。实际上，不言而喻，还应保证其间的各种一致性关系。

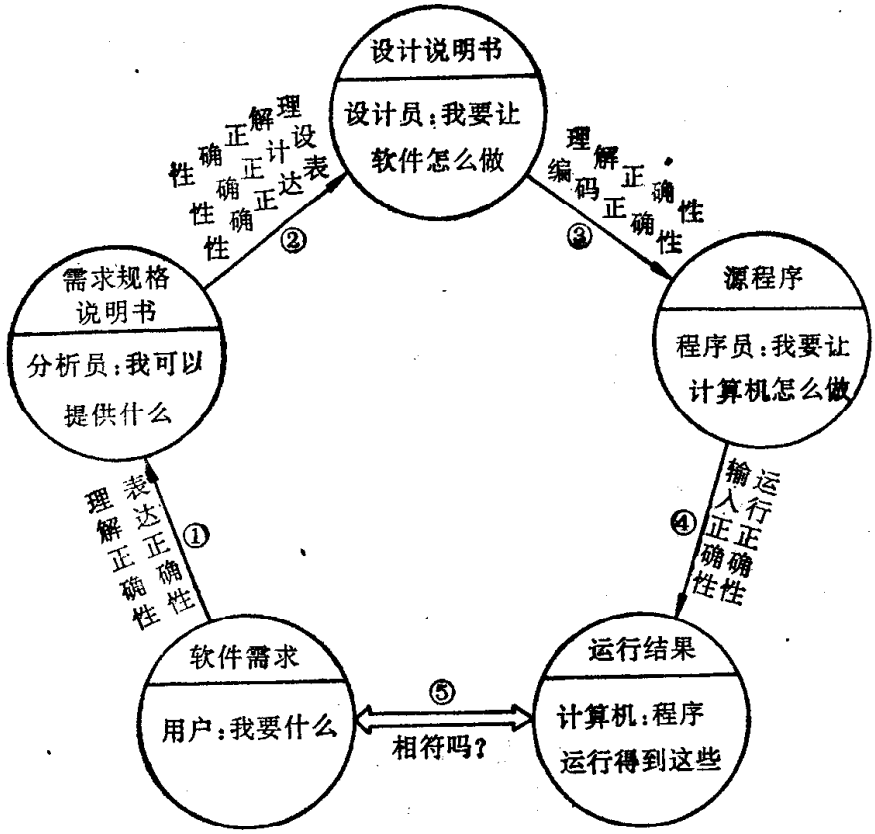


图 1.9　软件开发主要环节之间需要保持的正确性

四、开发过程出现的错误何必急于解决？

遵循软件生存期模型，各阶段的开发工作有着明确的目标，并按一定的顺序进行。比如，软件设计总是在需求分析之后，并且要在需求分析工作的基础上进行。需求分析的结果是软件设计的依据，软件设计是需求分析的扩展。软件生存期中相邻两个阶段工作之间的依赖关系是十分明显的。

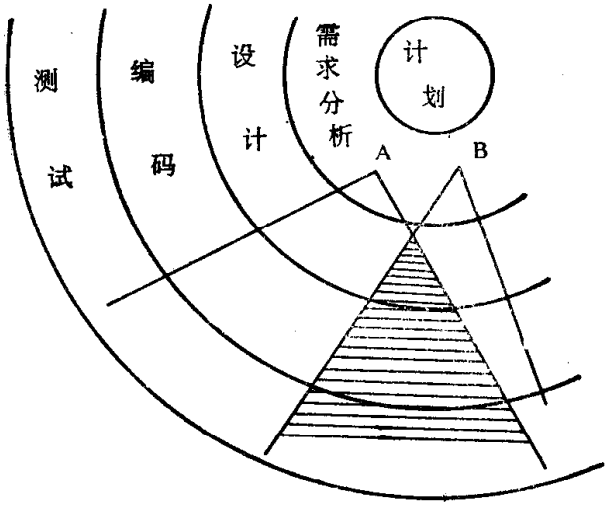


图 1.10　开发前期出现错误的扩展

由于各开发阶段工作的连续性，以及逐步扩展，早期开发中出现的错误如不能得到及时发现和解决，必然要在后面各阶段中逐步传递，并且也会逐步扩展。图 1.10 表明了这一扩展的现象。如果在需求分析阶段有某一需求的问题，例如图中的 A 点，在当时也只是范围很小的隐藏问题。但将其带到设计、编码以及测试各阶段，那就要逐步扩展。不仅要付出不必要的人力、物力作代价，所造成的影响还会扩大到更广的范围。特别值得注意的是，如果当时有两个问题没有得到解决，

例如图中的A和B点，它们在后期阶段的扩展中，两个扇形的影响范围有一部分相互重叠。由于两者的相互交错，表现出来的现象更加复杂，这就造成了更大的麻烦。致使问题不仅难于找出，也难于解决。自然，这些问题的解决必定要付出更大的代价。由此可以看出，开发前期出现错误带来的严重后果。

十分明显，及时排除早期开发中出现的错误，可以排除它给后期工作带来的麻烦，也就避免付出高额的代价，从而大大提高了开发的效率。这对降低整个软件的开发成本有着特别重要的意义。

五、程序验证方法能否取代测试问题

鉴于软件测试具有的局限性，一些人力图避开测试，另辟新径，这就是程序正确性验证方法。有人认为，程序正确性验证能够准确地指明程序是否存在问题；而且程序验证使用的是一种形式化方法，不象软件测试只是以实践为主的方法，因此可以指望程序验证取代程序测试。这种看法实际上是错误的。

首先，必须看到程序验证方法也是有局限性的。实际上，真正能够通过程序验证方法而得到程序正确性确认的例子是不多的。并且在大多数情况下，程序验证还仅仅停留在一些小的例子上。大中型的软件开发项目的验证无论从方法上还是从验证效率上看，都还不能应用于实际。

其次，已经验证为正确的程序，仍然可能存在错误。这就需要用软件测试方法对其进行补充。经验证为正确的程序，还可能存在着以下几种错误：

① 需求规格说明书 (Requirement Specifications) 错误

在软件开发的需求分析阶段应该制定的一份重要文件是软件需求规格说明书，它既是反映用户要求的文件，又是进行软件设计的依据。采用程序验证方法，需将非形式的规格说明转换成形式化的规格说明书。在转换过程中可能会出现错误。而验证过程使用的是程序本身和形式化说明书，因而形式化说明书的错误并不容易查出。与此相反，软件测试依据的标准输出结果通常是从非形式化说明书中得到的，这就有可能发现程序验证方法未能捕捉到的错误。

② 接口错误

程序验证方法由于工作量的限制，常常局限于程序模块或是子系统上，很少有对整个软件进行验证的实例。然而在局部被验证为正确的模块，常常还存在着接口的错误。事实上，程序模块的接口并不是很容易作出形式化描述的。这类接口错误只有通过模块的集成测试和系统测试，才能发现和解决。

③ 目标错误

程序验证方法实施的过程也可能发生错误，这种验证错误将会导致程序中的错误被掩盖，无法发现，也可能会出现虚报错误的错误。即使程序验证实施过程是完全正确无误的，还可能会出现程序验证目标不正确的现象。也就是说，验证得出的结论并不能说明程序的实现与要求的功能一致。因此，程序中存在的错误并不一定能检测出来。而程序测试方法由于它是实际运行程序，便有可能查出这些错误。

从上述分析可以看出，要想用程序正确性验证方法代替程序测试是不可能的。在实

际的软件开发项目中，测试仍然是最为现实、有效的质量保障手段。

六、软件测试信息流

为进一步说明软件测试的过程，这里给出软件测试的信息流图(见图 1.11)。

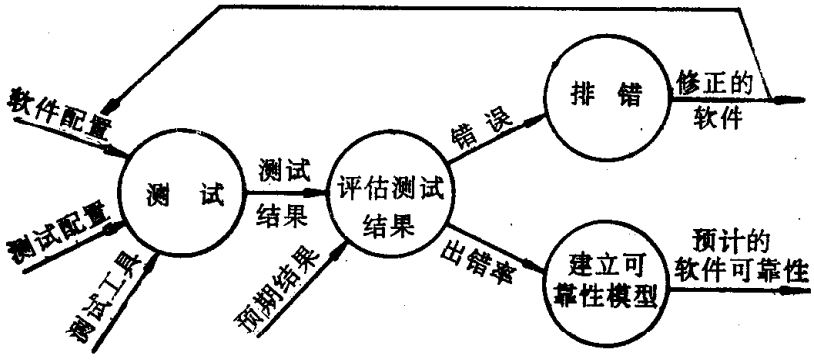


图 1.11　软件测试信息流

实施测试应给出三类信息：

① 软件配置：这是测试的对象，包括软件需求规格说明书、设计规格说明书和被测的源程序。

② 测试配置：包括测试计划、测试步骤、测试用例(测试数据)，以及具体实施测试的测试程序等。

③ 测试工具：为高效率完成测试所采用的测试工具软件。

测试的结果与预期结果进行比较以后，可以判断是否存在错误，继而进入排错。这包括具体确定错误的位置，并着手修正。修正后的程序必须重新测试。不能认为修正总是没有问题的，恰恰相反，修正也常常带来新的问题。

根据出错情况得到的出错率可用以建立被测软件的预计可靠性，这在软件投入运行以后的维护工作中有着参考价值。

1.5　软件测试发展的历史回顾

一、自从计算机作为强大的计算工具在本世纪中出现以来，程序的编制与程序的测试课题就同时摆在人们面前。不过早期的计算机运行可靠性差，当时使用计算机的突出矛盾是元器件质量不高，工作不稳定。不幸的是这一情况又和运行有毛病的程序交织在一起，令人十分为难。计算程序得不到正确结果，不容易分辨是计算机本身的问题，还是程序编写的问题。然而，这两者的性质毕竟是完全不同的，必须分别加以研究解决。后来的实践表明，元器件的质量在若干年后有了很大的提高，而程序测试问题的解决相比之下显得远不能令人满意。

早在 50 年代，英国著名的计算机科学家图灵就曾给出程序测试的原始定义。他认为，测试是正确性确认的实验方法的一种极端形式。自然，初期的测试都是针对机器语言程序或是汇编语言程序的。测试的主要方式是给出特定的输入或测试用例，运行被测程

序，再检查程序的输出，是否与预期的结果一致。测试用例的选取方法是在随机选取的基础上，吸取测试者的经验或是凭直觉判断，突出某些重点测试区域。50 年代以后，随着高级语言的诞生和广泛使用，测试工作自然把重心逐渐转移到用高级语言编写的程序系统了。尽管随机选取测试数据的方法一直是低效的，然而测试在程序的开发过程中仍然没有受到应有的重视。测试方法和理论研究发展缓慢。一些实际应用项目，除去非常关键的程序系统以外，一般程序系统所经历的测试大都是很不完备的。在开发工作结束以后，含有各种大小错误的程序投入运行了。如果运行中那些隐藏的缺陷并未暴露出来，也只不过是一时的幸运。无论如何这些隐患毕竟是不可靠因素，一旦暴露出来就会给用户和维护者带来不同程度的严重后果。早年探测火星所用的运载火箭因控制程序中错写逗号而爆炸，对空防御系统曾把月亮当作洲际弹道导弹的目标来轰击，这些都已成为人们谈论测试的笑柄。

测试工作在当时考虑不足的另一个原因是人们的心理因素。从软件系统开发者的角度看，研制工作的目标是使其能够运转起来，这是富有刺激性和创造性的任务，当付出相当的精力逐渐变为成果时，他们往往充满信心。他们不愿做那些后续的既麻烦又可能否定自己成果的测试工作，也不愿意让别人给自己开发的软件挑毛病。正如 Myers 所说的那样，人们把软件测试看成是设法从程序中找错的破坏性过程。成功的测试就是力图让程序执行失败。测试人员和开发人员的这一对抗心理在一段时间内成为测试工作取得成绩的障碍，极大地影响着测试技术的发展。70 年代以来，由于加深了对测试工作的认识，测试的意义逐渐被人们理解，加之一些测试工具陆续出现，上述矛盾得到了缓和。

即使存在上述矛盾，软件开发项目仍然必须进行基本的测试活动，以便向开发人员自己，也向别人表明软件系统的正确性。开发项目的实践成为发展测试技术的基础，尽管所用的测试方法并非都是高效的。另一方面，测试技术的研究开始受到重视，也取得了一些初步的成果。1982 年 6 月在美国北卡罗来纳大学召开了首次软件测试的正式技术会议，关心软件测试和软件质量的研究与开发人员第一次聚会，专门讨论他们感兴趣的问题。这次会议成为软件测试技术发展中的一个重要里程碑。

会议以后，软件测试的研究更加活跃，有关软件测试技术的论文如雨后春笋。F. P. Brooks 总结了开发 IBM OS/360 操作系统中的经验，在著名的《神秘的人-月》一书中阐明了软件测试在研制大系统中的重要意义。B. W. Bohem 指出，由于测试工作主要集中在系统的递交阶段，使得测试的效果完全依赖于运行情况，这是当时测试工作的一个致命弱点。美籍华人学者 J. C. Huang（黄荣昌）教授在 1975 年讨论了测试准则、测试过程、路径谓词、测试数据及其生成等问题。首次全面地论述了软件测试的有关问题。在 1972 年会议上发表多篇论文的 W. C. Hetzel 于 1975 年整理出版了“Program Test Methods”一书，书中纵览了测试方法以至各种自动测试工具。这是专题论述软件测试的第一本著作。E. F. Miller 在测试管理和普及方面作了大量工作，他为把现代测试概念推向实业界作出了重要贡献。在软件工程学科颇有影响的美籍印人 C. V. Ramamoorthy 和华人 R. T. Yeh（叶祖尧）等则分别在早期的测试自动化工具的研究和测试技术的资料汇集方面做了大量的工作。

70 年代中期，软件测试技术的研究达到高潮。J. B. Goodenough 和 S. L. Gerhart

首先提出了软件测试的理论，从而把软件测试这一实践性很强的学科提高到理论的高度，被认为是测试技术发展过程中具有开创性的工作。此后不久，著名测试专家 W. E. Howden 指出了上述理论的缺陷，并进行了新的开创性工作。以后，又有 Weyuker 和 Ostrand; Geller，以及 Gerhart 进一步总结原有的测试理论并进一步加以完善，使软件测试成为有理论指导的实践性学科。

在软件测试理论迅速发展的同时，各种高级的软件测试方法也将软件测试技术提高到了初期的原始方法无法比拟的高度。J. C. Huang 提出了程序插装（Program Instumentation）的概念，使被测程序在保持原有逻辑完整性的基底上，插入"探测仪"，以便获取程序的控制流及数据流信息，并可得到测试的覆盖率。W. E. Howden 对路径测试进行了深入的分析，提出了系统功能测试及代数测试等概念。W. E. Howden、L. A. Clarke 和 J. A. Darringer 等人把符号执行的概念引入到软件测试中，提出了符号测试方法，并且建立了 DISSET 等符号测试系统。Woodward 和 Hedley 等人分析了路径测试中的不可行路径，并提出了衡量软件测试水平的层次度量方法。R. A. Demillo 首先提出了耦联效应假设，并以此为基础发展出基于程序变异（Program Mutation）的测试方法，使传统的测试技术领域增加了新的成员——错误驱动测试。接着 T. A. Budd 和 F. Sayward 等人进一步发展了程序变异的思想，论述了能采用其它测试方法的地方基本上都能使用程序变异方法进行测试，并且开发了变异测试系统的原型。后来，W. E. Howden 把 DeMillo 等人的变异方法称为强变异，依据原始的变异思想提出了弱变异的测试技术。使用弱变异方法无需生成被测程序的变异因子，而是分析程序中易于出错的部位进行变异测试。在这以后，S. J. Zeil 提出了一种新的程序模型，把程序描述为环境的改变，通过扰动减少错误空间的维数，这就是与程序变异相似的程序扰动测试方法（Permutation of Program Statements）。1977 年 L. Osterweil 和 L. D. Fosdick 等人首先引入了数据流测试方法，该方法通过对数据流的静态测试找出程序中潜藏的错误。Osterweil 还把这一方法推广到并发程序的数据流分析。1983 年日本人 Ryoichi Hosoya 等在数据流测试方法中加入了变量值域分析，使数据流方法检测的错误类型更多。

如何决定测试数据，选取测试点，仍然是实施测试，使之提高测试效率和缉错命中率的关键问题。L. White 和 E. Cohen 提出了一种新的计算机程序测试策略，这就是域测试方法（Domain Testing Strategy）。输入域分析是把程序的输入域按谓词划分，进一步以此划分为依据，给出各个域的测试点。E. J. Weyuker 和 T. J. Ostrand 接着发展了他们的方法。1985 年 D. J. Richardson 和 L. A. Clarke 在此基础上又提出了划分分析的概念，将形式化程序说明书和程序本身的输入变量都划分为域，通过其交的测试找出它们之间的不一致性。L. A. Clarke 还深入讨论并改进了 White 和 Cohen 提出的域测试方法。此外，S. Redwine Jr 总结了一套工程化测试方法。

60 年代末，70 年代初软件危机的矛盾日益突出，软件工程化的概念逐渐形成。把软件工程活动分为需求分析、设计、编码、测试和维护几个阶段的软件生存期的概念被人们广泛接受。同时，从软件开发的实践中，人们还认识到，在开发初期发现并解决软件错误所付出的代价远比在编码以后经过测试而发现错误并加以改正的代价小得多。从这个认识出发，各种有关生存期前几个阶段的测试理论和测试方法应运而生。W. E. Howden

给出了软件生存期各个阶段的测试目标。D. J. Richardson 和 L. A. Clarke 指出，应当为划分分析测试方法（Partion Analysis）提供形式化的可执行的程序说明书。S. C. Ntafos 提出，结构测试的弱点在于只注重源程序的代码；纯粹的黑盒测试对某些特殊情况的检验是无能为力的；错误驱动测试的可靠性又不得而知，因而应该从软件工程的思想中推出需求元素测试的理论。1985 年 M. S. Karasik 提出了环境测试的概念，其重点不在于验证程序是否与说明书一致，而是看程序是否与其运行环境一致。1986 年 I. J. Hayes 提出说明书制导的模块测试方法，主张在早期开发中就处处想到测试的需求。此外，P. Jalote 和 M. Majaros 在抽象数据类型的测试和在形式化说明书背景下的测试分别进行了深入的研究。

近年来，尽管软件测试技术有了长足的进步，但总的来说，仍然和软件开发实践提出的要求有相当大的距离。测试手段的进展也远远没有达到令人满意的程度。

二、以下概括地介绍一些软件测试工具的发展情况

为了提高软件测试的效率，加快软件开发过程，一些测试工具逐渐问世。有些测试工具用以检查规格说明书的一致性和完整性，有的用以检查编码的静态特征，也有些则支持已提出的测试方法。R. E. Fairley 曾经系统地论述了软件工具的作用，重点介绍了 ADA 支撑环境。L. G. Stucki 所设计的程序评估和测试器 PET 能够自动进行 FORTRAN 程序的测试，并能给出测试覆盖信息。RXVP 原是用于检测 FORTRAN 程序静态错误的，扩充以后的 RXVP 80 能兼顾 FORTRAN 和 COBOL 的程序测试。类似的工具还有 C. V. Ramamoothy 教授指导开发的 FACES 和 L. J. Osterweil 等人开发的 DAVE 系统。H. Kao 和 T. Chen 开发了支持 COBOL 程序数据流分析的工具。

有关动态测试的工具的开发更为普遍，其中的一部分工作力图自动产生测试数据，比如 SMOTL 测试编译程序的工具，以及一些支持符号测试的工具 SELECT，DISSECT，ATTEST 和 SADAT 等。由于模块测试时需要用到驱动模块和桩模块（参看本书第三章 3.3 节），一些模拟测试环境的工具出现了，如 R. G. Hamlet 设计的 Test-master 系统和 D. J. Pauzl 设计的 AUT。

近年来，对软件测试理论的研究和测试新方法的探讨陷入困境，人们的注意力更多地转向了软件测试工具，促使测试工具进一步推陈出新。C. Wilson 和 L. J. Osterweil 不久前提出了支持C 程序数据流分析工具 Omega，主要解决了C 程序中指针引用的测试问题。E. F. Miller 设计了交互式测试环境 ITB。接着出现的有 M. A. Hennel 和 D. Hedley 开发用于 PFORT 语言程序的 LORA 软件测试环境以及 B. Montel 等人设计测试 Petri 网的软件包 OVIDE。D. J. Reifer 对软件测试工具的特点和使用范围作了说明，这对于了解测试工具的全面情况很有帮助。

值得注意的是，近几年国内软件测试工具的开发取得了显著进展。在此仅列举几个有代表性的课题：

- 西安交通大学开发的 COBOL 测试系统 CTPS-1
- 华中科技大学开发的C 编译程序的测试系统
- 北京航空航天大学与清华大学开发的C 软件综合测试系统

· 清华大学开发的 FORTRAN 静态分析系统及动态特性分析系统 DTFG（见本书第七章 7.3 节）

· 清华大学开发的 COBOL 软件测试环境 COSTE（见本书第七章 7.2 节）

三、与软件测试相关课题的发展

与软件测试相关的课题主要包括：图论的应用、正确性证明、程序排错、软件质量保证以及软件复杂性度量。其发展情况简述如下：

图论在计算机软件各个领域中有着广泛的应用，在软件测试中，图论更是非常有用的工具。我们知道，对程序进行测试时，首先要为程序建立模型。为此可把程序看作一个有向图，通常把程序中的语句看作为有向图上的点，把语句间的控制顺序关系当作图上的边。M. R. Paige 进一步提出了程序图的若干分割方法。R. L. Probert 引入并且改进了 Böhm 和 Jacopini 定义的 BJ 图语法 $G=(V_N,V_T,S,R)$，其中 V_N 和 V_T 分别表示非终结图符和终结图符，S 是开始符号，R 是产生式集。并且利用所建立的图模型分析了探针的最优插装方法(关于程序插装请参看本书第五章 5.6 节)。

程序正确性证明历来是软件理论研究者感兴趣的课题。程序正确性证明的三个主要流派是 R. W. Floyd 的归纳断言方法、C. A. R. Hoare 的公理化方法以及 D. Scott 的定点归纳方法。对于实践性很强的软件测试来说需要有理论性较强的正确性证明来作为提高可靠性的支持，因而正确性证明颇受一些人推崇。然而，它毕竟只能与软件测试互相补充，并不能完全取代软件测试技术。这一点已经是很明显的了。任何过分夸大程序正确性证明效能的说法都是不恰当的。S. L. Gerhart 和 L. Yelowitz 在 1976 年列举了很多详实的例子，说明不完全形式化的正确性证明也会导致很多谬误。事实上，完全形式化的正确性证明也会在推理过程中引进错误，因而需要有软件测试的辅助。正确性证明必须包括对程序运行环境的完全严格的证明，才能确保经过证明的程序完全正确。要做到这一点，目前还很困难。此外，程序正确性证明工作开销巨大，我们还只能用在关键程序中。值得重视的是 W. E. Howden 曾多次实践，验证各种测试方法的有效性，而许多错误并不一定能在证明中发现。实际上，符号测试正是正确性证明与软件测试的最好结合。M. Geller 在两者之间的联系方面开展了十分有益的工作。

排错是软件开发中必不可少的步骤。但排错与测试有着密切的关系，这已在本章 1.2 节中作了详尽的说明。在此仅需指出它们之间的交互关系，那就是在测试中发现的错误，要经过排错消除。然而排错以后的程序还必须再测试，若发现仍有错误那就要再排错。总之，它们是交互循环进行的。J. B. Goodenough 提出，测试应该为排错提供必要的诊断信息。

保证软件质量可从严格开发过程和改善测试方法两方面着手。测试方法的好坏直接影响着软件的可靠性。M. L. Shooman 和 Rault 等人认为可将程序的可靠性与程序测试中错误发生率和测试覆盖率联系起来，并根据这一认识提出了可靠性模型。A.Cicu 索性把可测试性考虑在软件质量中，并将其作为衡量软件质量的一个标准。C. V. Ramamoorthy 特别强调指出，在要求高可靠的软件项目中，为确保安全，可以针对同一目标开发两个并行的软件，通过两套系统的测试比较来确保软件质量。

软件复杂性度量对软件测试起着指导作用。当前最为著名的软件复杂性模型是

Halstead 的软件科学度量模型和 McCabe 的程序环路模型。对于软件复杂性的初步估算能够预计测试的复杂程度。显然，这对于测试计划的制定和测试工作的组织管理都是非常有用的。

参 考 文 献

[1] F. Bazzichi et al., "An Automatic Generator for Compiler Testing", IEEE Transactions on Software Engineering, Vol. SE-8, No. 4, 1982.

[2] Boris Beizer, Software System Testing and Quality Assurance, Van Nostrand, 1984.

[3] Boris Beizer, Software Testing Techniques, Van Nostrand, 1983.

[4] B. W. Boehm, The High Cost of Software, In Practical Strategies for Developing Large Software, Addison-Wesley, 1975.

[5] R. S. Boyer et al., "SELECT——A Formal System for Testing and Debugging Programs by Symbolic Execution", Proc. International Conference on Reliable Software, Apr. 1975.

[6] F. P. Brooks, The Mythical Man-month, Addison-Wesley, 1975.

[7] T. A. Budd et al., "The Design of a Prototype Mutation System for Program Testing", Proc. National Computer Conference, 1978.

[8] A. Cicu, "The Quality of a Computer Program: The USER View and the Software Engineering View", Computer Program Testing.

[9] L. A. Clarke et al., "A Close Look at Domain Testing", IEEE Transaction on Software Engineering, July 1982.

[10] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, Vol. SE-2, No 3, Sep. 1976.

[11] J. A. Darringer, "The Use of Symbolic Execution in Program Testing", Software Testing, Vol. 2, Infotech State of the Art Report, 1979.

[12] R. A. Demillo et al., "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, Apr. 1978.

[13] R. A. Demillo et al., "Program Mutation: A New Approach to Program Testing", Software Testing, Vol. 2, Infotech State of Art Report, 1979.

[14] E. W. Dijkstra, "Notes on Structured Programming", Structure Programming, Academic Press, 1972.

[15] J. W. Duran et al., "An evaluation of Rundom Testing", IEEE Transactions on Software Engineering, July 1984.

[16] R. E. Fairley, "Software Testing Tools", Computer Program Testing, Edited by S. Radcchi, North-Holland, 1981.

[17] K. A. Foster, "Error Sensitive Test Cases Analysis", IEEE Transactions on Software Engineering, Vol. SE-6(3), May 1980, pp. 258—264.

[18] L. D. Fosdick et al., "Data Flow Analysis in Software Reliability", Computing Surveys, Vol. 8, Sept. 1976.

[19] S. L. Gerhart et al., "Observations of Fallibility in Application of Modern Programming Methodologies", IEEE Transactions on Software Engineering, Sept. 1976.

[20] M Geller, "Test Data As an aid in Proving Program Correctness", CACM, Vol. 21, May 1978.

[21] J. B. Goodenough, "A Survey of Program Testing Issues", Research Directions in Software Technology, MIT Press, 1980.

[22] J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, June 1975.

[23] J. S. Gourlay, "A Mathematical Framework for the Investigation of Testing", IEEE Transactions on Software Engineering, Vol. SE-9, Nov. 1983.

[24] M. Halstead, Elements of Software Science, North-Holland, 1977.

[25] R. G. Hamlet, "Testing Programs with the Aid of a Compiler", IEEE Transactions on Software Engineering, Vol. SE-3, No. 4, 1977.

[26] I. J. Hayes, "Specification Directed Module Testing", IEEE Transactions on Software Engineering, Jan. 1986.

[27] W. C. Hetzel, Program Test Methods, Prentice-Hall Inc., 1973.

[28] R. Hosoya and H. Hotta, "Static Detection of Ada Programming Error Through Joint Analysis of Data Flow and Value Range", Proc. Compsac '83, Chicago.

[29] W. E. Howden, "Algebraic Program Testing", Acta Informatica, Vol. 10, Jan. 1978.

[30] W. E. Howden, "Functional Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No. 2, 1980.

[31] W. E. Howden, "Weak Mutation Testing and Completeness of Test Sets", IEEE Transactions on Software Engineering, July 1982.

[32] W. E. Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-2, Sept. 1976.

[33] W. E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System", IEEE Transactions on Software Engineering, Vol. SE-3, No. 4, 1977.

[34] J. C. Huang, "Program Instrumentation and Software Testing" Computer, Apr. 1978.

[35] J. C. Huang, "Instrumenting Programs for Data Flow Analysis", TR-UH-CS-77-4, Univ. of Houston, May 1977

[36] P. Jalote, "Specification and Testing of Abstract Data Types", Proceedings of COMPSAC 83, Chicago.

[37] H. Kao and T. Y. Chen, "Data Flow Analysis for COBOL". ACM SIGPLAN Notices, Vol. 19, No. 7, 1984.

[38] M. S. Karasik, "Environment Testing Techniques for Software Certification", IEEE Transactions on Software Engineering, Sept. 1985.

[39] R. A. Kemmerer, "Testing Formal Specifications to Detect Design Errors", IEEE Transactions on Software Engineering, Jan. 1985.

[40] Kuo-Cheng Tai "Program Testing Complexity and Test Criteria", IEEE Transactions on Software Engineering, Nov. 1980.

[41] J. W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, 1983.

[42] M. Majaros et al, "Testing Program Against a Formal Specification", Proc. COMPSAC 83, Chicago.

[43] T. J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Dec. 1976.

[44] E. F. Miller, "Structural Techniques of Program Validation", Proc. COMPCON 74, San Francisco.

[45] E. F. Miller et al., "A Software Test Bed: Philosophy, Implementation and Application", Computer Program Testing, North-Holland, 1981.

[46] E. F. Miller, "Program Testing: Art Meets Theory", Computer, Vol. 10(7), July 1977.

[47] E. F. Miller, "A Service Concept for Software Auditing", Proc. of the Workshop on Computer Auditing, San Francisco, 1976.

[48] B. Montel et al., "Ovide: A Software Package For Verifying and Validating Petri Nets", Software Fair.

[49] G. J. Myers, The Art of Software Testing, John Wiley & Sons, 1979.

[50] S. C. Ntafos, "On Required Element Testing", Proceedings of COMPSAC 81.

[51] L. J. Osterweil, and L. D. Fosdick, "DAVE: A Validation Error Detection and Documentation System for FORTRAN Programs", Software Practice and Experience, Vol. 6(4), 1976.

[52] L. J. Osterweil, and L. D. Fosdick "A Validation Error Detection and Documentation System for FOR TRANPrograms", Software P & E, Vol. 6, 1976.

[53] L. J. Osterweil, "The Detection of Unexecutable Program Paths Through Static Data Flow Analysis", Proc. COMPSAC 77, 1977.

[54] L. J. Osterweil, "A Strategy for Integrating Program Testing and Analysis", Computer Program Testing, Edited by S. Raduhi, North-Holland.

[55] M. R. Paige, "On Partitioning Program Graphs", IEEE Transactions on Software Engineering, Nov. 1977.

[56] D. J. Pauzl, "Automatic Software Test Drivers", Computer, Vol. 11, No. 4, 1978.

[57] R. L. Probert, "Optimal Insertion of Software Probes in Well-Delimited Programs", IEEE Transactions on Software Engineering, Vol. SE-8(1), Jan. 1982.

[58] Qiang Zeng et al., "An Experimental C Validation System", Proc. International Workshop on Software Engineering Environment, Beijing, 1986.

[59] C. V. Ramamoorthy and S. F. Ho, "FORTRAN Automatic Code Evaluation System", ERL-M466, Univ. California at Berkeley, 1974.

[60] C. V. Ramamoorthy et al., "On the Automated Generation of Program Test Data", IEEE Transactions on Software Engineering, Vol. SE-2(4), Dec. 1976.

[61] C. V. Ramamoorthy et al., "Testing Large Software with Automated Software Evaluation Systems", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, Mar. 1975.

[62] S. T. Redwine, "An Engineering Approach to Software Test Data Design", IEEE Transa. ctions on Software Engineering, Mar. 1983.

[63] D. J. Reifer et al., "A Glossary of Software Tools and Techniques", COMPSAC 77, Tutorial on Program Testing Techniques, 1977.

[64] D. J. Richardson and L. A. Clarke, "Partition Analysis: A Method Combining Testing and Verification", IEEE Transactions on Software Engineering, Dec. 1985.

[65] M. L. Shooman, Software Engineering, McGraw-Hill, 1983.

[66] A. M. Turing, "Checking A Large Routine", Report of a Conference on High Speed Automatic Calculating Machines, University Mathematical Laboratory Cambridge, Jan. 1950.

[67] U. Voges et al., "SADAT-An Automated Testing Tool", IEEE Transactions on Software Engineering, Vol. SE-6(3), May 1980.

[68] A. E. Westley, Foreward To'Software Testing', Infotech State of the Art Report, 1979.

[69] E. J. Weyuker et al., "A Data Flow Oriented Program Testing Strategy", IEEE Transactions on Software Engineering, May 1983.

[70] E. J. Weyuker et al., "Theories of Program Testing and the Application of Revealing Subdomains", IEEE Transactions on Software Engineering, Vol. SE-6(3), May 1980.

[71] L. J. White and E. Cohen, "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, 1980.

[72] William E. Perry, "A Structured Approach to Systems Testing", QEP Information Sciences, 1983.

[73] C. Wilson et al., "Omega-A Data Flow Analysis Tool For the C Programming Language", IEEE Transactions on Software Engineering, Vol. SE-11, Sept. 1985.

[74] M. R. Woodward et al., "Experience With Path Analysis and Testing of Programs", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, May 1980.

[75] R. T. Yeh (editor), Current Trends in Programming Methodology, Vol. 2, Prentice-Hall, 1977.

[76] S. J. Zeil, "Testing For Pertubations of Program Statements", IEEE Transactions on Software Engineering, Vol. SE-9(3), May 1983.

[77] L. G. Stucki, "New Directions in Automated Tools for Improving Software Quality", Current Trends in Programming Methodology, Vol. 2, Edited by R. T. Yeh, Prentice-Hall, 1977.

[78] E. F. Miller, "Structural Techniques of Program Validation", IEEE COMPCON 74, 1974.

[79] C. L. Andrews et al., "RXVP 80-The Verification and Validation System For FORTRAN

and COBOL", Proc. Softfair.

[80] J. Bicevskis et al., "Smotl-A System To Construct Samples for Data Processing Program Debugging", IEEE Transactions on Software Engineering, Vol. SE-5, No. 1, 1979.

[81] F. Bazzichi et al., "An Automatic Generator For Compiler Testing", IEEE Transactions on Software Engineering, Vol. SE-8, No. 4, 1982.

[82] C. V. Ramamoorthy et al., "Application of a Methodology For the Development and Validation of Reliable Process Control Software", IEEE Transactions on Software Engineering, Vol. SE., Nov. 1981.

[83] Bosheng Zhou, Renjie Zheng et al., "C Software Testing Environment", Proc. International Workshop on Software Engineering Environment, Beijing, 1986.

[84] Li Yu-ren, Liu Shao-ying, "Design and Implementation of COBOL Program Testing System CPTS-1", Proc. International Workshop on Software Engineering Environment, Beijing, 1986.

[85] 周之英、刘爱武、王静，"Fortran 测试系统",«计算机软件开发方法、工具和环境»西北大学出版社,1985年8月。

[86] Richard A. DeMillo et al., Software Testing and Evaluation, Benjamin/Cummings Publishing Company, 1987.

[87] 郑人杰,实用软件工程,清华大学出版社,1991。

[88] R. S. Pressman, Software Engineering: A Practitioner's Approach, Second Edition, McGraw-Hill, New York, 1987.

[89] IEEE Standard Glossary, IEEE Std 729—1983.

[90] W. C. Hetzel, The Complete Guide to Software Testing, QED Information Sciences, Inc. Wellesley, MA, 1984.

[91] Cem Kaner, Testing Computer Software, TAB Books Inc., 1988.

第二章　软件错误与软件质量保证

软件开发项目经历了计划、需求分析、设计和编码以后，已经取得了初步成果。但这些成果究竟能否满足用户的需求，能够在多大程度上满足用户的需求，这不仅是开发人员，也是管理人员和用户十分关心的问题。质量检验和质量保证活动无疑是非常必要的。

很明显，软件错误是质量检验活动中的"稽查"目标。本书在以后几章展开讨论各种测试技术以前，需要对这一目标进行初步的分析，同时给出软件质量保证的基本概念。

本章首先分析软件错误的类型和软件错误的原因，进而讨论程序中错误数量的估计。关于软件质量和软件质量保证是个很大的题目，本章只是扼要地给出一些概念。最后简要地讨论程序排错的方法。

2.1　软件错误类型分析

一、程序正确性的差异

我们编写出的源程序在测试中受到检验，它能不能正确无误地工作，实践表明，可能出现许多种不同的情况。这些情况列举如下：

① 程序编写得无语法错误

要求程序编写得无语法错误是程序运行的最起码条件。我们知道，具有语法错误的程序在上机运行时，无法通过编译系统的语法检查。编译程序会列举出程序中各种语法错误现象，而停止编译，也就谈不到程序的执行。

② 程序执行中未发现明显的运行错误

这是指程序运行过程中没有因产生过大或过小的数据，由于溢出而无法继续执行；也没有遇到循环不已而障碍运行等情况。

③ 程序中无不适当的语句

程序尽管符合语法规则，也未出现上述运行错误，但有些语句是不适当的。例如，有的变量未经说明而引用；有的虽已作说明，却未曾引用；或者有的变量未赋初始值而被引用；以及有的变量被多次赋值，并未引用等等。

④ 程序运行时，能通过典型的有效测试数据，而得到正确的预期结果。

这表明程序能够接受规格说明所规定正常条件下的合理数据，并给出正确结果。

⑤ 程序运行时能通过典型的无效测试数据，而得到正确的结果。

程序运行中可接受规格说明所规定异常条件下的不合理数据，并给出相应的结果。

⑥ 程序运行时能通过任何可能的数据，并给出正确的结果。

以上给出的6种情况表明了程序正确性的差别。我们写出的程序如果能达到第6条要求是很不容易的，这一条要求意味着程序中没有任何隐含的缺陷或差错。这是我们编

写程序希望达到的最高目标。实际上，要达到这个目标必须付出相当大的代价。

十分明显，提高程序的正确性就是要尽可能发现和消除程序中隐藏的各种差错。

如上所述，编译系统接受用户的源程序后所作的语法检查能够发现程序编写时出现的语法错，但也仅仅是一些语法错，更多情况的差错编译系统是无法查出的。比如，在程序中往往会出现的逻辑性差错、名字拼写错、不正确的初始化或未作初始化、数据格式或文件格式不对、循环次数有错、调用了错误的程序块或是纯粹属于语义上的错误等。这些错误正是要在测试中发现，在排错中消除。

二、软件错误的分类

软件人员在编写程序中出现的错误常常一再重复。例如，数组超界，变址或移位差1，用错特征位，补码运算错，初始化不对，使用指针的问题，控制转移的问题以及间接寻址出错的问题等。实际上，这些程序编写中的错误远远不是软件的主要错误。如果我们把眼光稍微扩展一下，从软件错误的性质看，可以发现软件错误又分为以下几种类型：

① 软件需求错误

软件需求制定得不合理或不正确；需求不完全；其中含有逻辑错误；需求分析的文档有误等等。

② 功能和性能错误

功能或性能规定得有错误，或是遗漏了某些功能，或是规定了某些冗余的功能；为用户提供的信息有错，或信息不确切；对意外的异常情况处理有误等等。

③ 软件结构错误

程序控制流或控制顺序有误；处理过程有误等。

④ 数据错误

数据定义或数据结构有误；数据存取或数据操作有误等。例如动态数据与静态数据混淆，参数与控制数据混淆等。

⑤ 软件实现和编码错误

编码错或按键错；违背编码风格要求或是编码标准的问题。包括语法错、数据名错、局部量与全局量混淆，或是程序逻辑有误等。

⑥ 软件集成错误

软件的内部接口、外部接口有误；软件各相关部分在时间配合、数据吞吐量等方面不协调。

⑦ 软件系统结构错误

操作系统调用错或使用错、恢复错误、诊断错误、分割及覆盖错误以及引用环境的错误等。

⑧ 测试定义与测试执行错误

测试的错误往往被人们忽视，它可能包括测试方案设计与测试实施的错误、测试文档的问题、测试用例不够充分等等。

上述这些类型的软件错误中最值得重视的是软件结构错误、数据错误和功能与性能错误，因为这三种错误最为普遍。

表 2.1　软件错误分类统计

错　误　分　类	错 误 数	百 分 比
1.需求错误	1317	8.1
需求有误	649	4.0
需求逻辑错	153	0.9
需求不完备	224	1.4
需求文档描述问题	13	0.1
需求更改	278	1.7
2.功能和性能错误	2624	16.2
功能或性能有误	456	2.8
性能不完全	231	1.4
功能不完全	193	1.2
适应范围有问题	778	4.8
用户信息和诊断信息有误	857	5.3
异常情况处理有误	79	0.5
其它功能错误	30	0.2
3.结构错误	4082	25.2
控制流和控制顺序错	2078	12.8
处理错	2004	12.4
4.数据错误	3638	22.4
数据定义及数据结构错	1805	11.1
数据存取及数据操作错	1831	11.3
其它数据问题	2	0.0
5.实现与编码错误	1601	9.9
编码及按键错	322	2.0
违背编码风格或标准	318	2.0
文档有误	960	5.9
其它实现的问题	1	0.0
6.软件集成错误	1455	9.0
内部接口错	859	5.3
外部接口,时间、吞吐量不匹配	518	3.2
其它集成错误	78	0.5
7.系统结构错误	282	1.7
操作系统引用或使用错	47	0.3
软件结构错误	139	0.9
恢复错误	4	0.0
执行错误	64	0.4
诊断错误	16	0.1
分割与覆盖错误	3	0.0
引用环境错	9	0.1
8.测试定义与测试执行错误	447	2.8
测试设计错误	11	0.1
测试执行错误	355	2.2
测试文档有误	11	0.1
测试用例不充分	64	0.4
其它测试错误	6	0.0
9.其它类型错误	763	4.7
总　计	16,209	100.0

为分析软件错误的来源，有人作了大量的统计工作。例如，对含有 6,877,000 个语句(其中包括注释行)的某一软件进行了单元测试、集成测试和系统测试以后，将发现的各种错误进行了统计分析。发现错误总数为 16,209 个，平均每千行语句含有错误 2.36 个。表 2.1 给出这些统计分析的数字。

三、软件错误的后果

软件的错误常常给软件的运行带来不同程度的后果。

按错误发生的影响和后果，可能区分以下几种类型：

① 较小错误：这类错误只是对系统的输出结果有一些非实质性影响，比如，输出的数据格式不符合要求等。

② 中等错误：对系统的运行有局部的影响。如输出的某一部分数据有错误或出现冗余。

③ 较严重错误：系统的行为由于错误的干扰而出现明显不合情理的现象。如开出 0.00 元的支票。系统的输出结果完全不可信赖。

④ 严重错误：系统运行不可跟踪，一时不能掌握其规律，时好时坏。

⑤ 非常严重的错误：系统运行中突然停机，其原因不明，且无法软起动。

⑥ 最严重错误：运行被测的软件导致环境遭到破坏，或是造成事故，引起生命、财产的损失。

以上只是列举一些测试可能发生的现象，但我们可以通过这些现象来估计错误的严重程度，作到心中有数。

需要指出的是，错误的大小与其后果严重程度并不是成比例的。就是说，很小的软件错误也可能带来非常严重的后果。1963 年在美国发生的一个实例充分地说明了这一点。

一个 FORTRAN 程序的循环语句

```
DO  5  I = 1,3
```

被误写为

```
DO  5  I = 1.3
```

由于空格对 FORTRAN 编译程序没有实际意义，误写的语句被当作了赋值语句：DO5 I = 1.3。这里只是“,”“错成“.”，结果一点之差竟造成极为严重的后果。因为该 FORTRAN 程序控制了金星探测火箭的飞行，火箭的事故使得损失一千万美元。

其实，更能说明问题的还是本书第一章中提到的 IBM 360 机的操作系统。由于技术，特别是管理上的原因，在程序中含有许多错误。在一再修改之后，又推出新的版本。事实上，新版本的质量不见得好多少。据统计，这个操作系统每次发行的新版本都是从前一版本中找出一千个程序错误后得到的。致使人们对软件的质量产生怀疑，该项目竟成为软件危机的典型事例和标志。

上述典型事例已经成为人们谈论软件测试的笑柄。它告诉我们，哪个软件项目不重视测试工作，那就迟早要得到严重的教训，尝到苦果。

2.2 程序中隐藏错误数量估计

为便于组织测试与排错工作，我们常常需要了解已开发程序中含有多少错误。但实际上要回答这一问题并非易事。本节将讨论几种估算程序中错误数量的模型和方法。

一、撒播模型 (Seeding Models)

这里介绍的程序错误撒播方法是从估算池塘中养鱼数或是野生动物数所用方法得来的。假定在池中有某个品种的鱼N尾，而且池中并没有其它品种的鱼。为了估算池中这一品种鱼的数量N，首先从池中取出N_t尾(当然，这是一个合理的，实际上可以取得的数量)，然后给这些鱼均作上有利于辨认的标记，再放回池中，使其与未作标记的鱼充分混游。几天后，再从池中任意取出一些鱼样，并且根据标记加以区别，得到带标记者n_t尾，无标记者n尾。如果这一取样仍然是随机进行的，并不因是否带有标记而易于(或难于)捕获。于是，可以得到以下关系式：

$$\frac{N_t}{N+N_t}=\frac{n_t}{n+n_t} \tag{2-1}$$

它表明样鱼中带标记鱼的比例等于作标记鱼的比例。进而得到N的计算值：

$$\hat{N}=\frac{n}{n_t}N_t \tag{2-2}$$

其中，N_t、n和n_t均为已知量。

我们完全可以模仿上述方法估算在开始排错以前其中含有的错误数N。首先，在不让排错程序人员知道的情况下，往程序中播入N_s个错误(这相当于为N_t尾鱼作标记后放归池中)。经过τ个月的错误工作以后，派持有播入错误清单的专人(并非排错程序员)检查排错的情况。它把排出的错误分为两类，一类是属于播入的错误n_s，另一类是非播入错误n。这样，利用前面估算鱼的公式便得到：

$$\hat{N}=\frac{n}{n_s}N_s \tag{2-3}$$

在程序中撒播错误，并借此来估算初始错误数量的方法最早是Mills提出来的。但是一开始所作实验并没有得到什么结果，其原因在于，一方面人为制造错误并不容易。而在池中捕鱼时即使只有一个品种，也还有大鱼和小鱼的差别，除非假定不管什么鱼捕到的概率是一样的。另一方面，这个方法并未用于机器测试，而仅仅当作排错手段用以评价代码阅读的有效性。

Hyman 所提出的方法则有其优越的地方。他建议让两个(或者多个)程序人员一开始针对同一程序分别独立地完成排错工作。假定估计需要4个月完成排错，则安排第一人4个月的工作，安排第二人只是开头的一至两个月。在分别工作几周以后，由一位分析员来评价他们的工作，他可根据类似于(2-3)式来估算错误的数量。这样的估算每隔几周便进行一次，直到取得满意的N值为止。然后，把第二人的工作结果交给第一人，并给第二

人安排另外的任务。这样，该程序的调试完成了 $\frac{1}{4}$ 或者一半以后，就可得到该程序中错误数量的合理估值。如果从中减去已排除的错误数，就可知道仍然残存在程序中的错误数。此外，第二人发现的错误只有一部分和第一人相同，因而这样做以后；第一个人就能很快地做完第二人的工作。从而可以大大地加速排错过程。在多数情况下，获得的效益会远远超过付出的代价，约为一两个人月的额外排错时间(如果第二个人能发现许多独立的错误，大大减轻第一人工作量，则此代价可能很小，也许是零或为负)。下面讨论如何得到计算公式的细节。

为描述两位排错人员协同工作的估算过程，需要引入以下几个符号：

τ　以月数计算的开发时间，0 到 τ_1 为量测期

B_0　τ 为 0 时程序中的错误数

B_1　到 τ_1 为止第一位排错人员找到的错误数

b_c　到 τ_1 时第二人所找到的两人共识错误数(即 b_c 也在 B_1 中)

b_1　到 τ_1 时第二人所找到的不在 B_1 中的错误数

$B_2 = b_c + b_1$　到 τ_1 时第二人发现的错误数

如果我们确信，在撒播实验中播入的程序错误与程序的固有错误在形式上是没有差别的，那么就可以找出某些错误，对其做上标记，充当种子。当然，要找到一组错误得花费一定的力气。由于我们选定哪组错误作为标记错误组无关紧要(只要这些错误是有代表性的)，我们应该能像处理标记错误组那样处理可辨认的错误组。这里需要几个基本假定：

① 在排错过程中错误特性不变

在对大型程序排错时，最初几周或最初几个月发现的错误可作为全部错误的代表。

② 相似的程序具有独立的排错结果

两个独立工作的排错人员在同一大型程序的不同版本上工作，可以发现程序虽有不同，但两个版本之间的差异对排错工作几乎没有什么影响。

③ 共同的错误是典型的错误代表

两个独立工作的排错人员对同一大型程序进行排错时，他们找出的共同错误是全部错误中的代表。

如果上述的假定成立，我们可以把 B_1 当作前面(2-3)式中的 N_s，b_c 当作 n_s，B_2 当作 n_o。得到原始错误数：

$$\hat{B}_0 = \frac{B_2}{b_c} B_1 \tag{2-4}$$

于是得到关于 B_0 方差的近似表达式。假定 B_1 和 B_2 的值不再改变，即是说，让两个排错人员工作直到取得 B_1 和 B_2。这样，B_0 的变化将取决于 b_c，可以猜想，该方差为：

$$\text{方差 } B_0 = \frac{B_0^2}{b_c} \tag{2-5}$$

将(2-4)式代入(2-5)，得到：

$$\text{方差 } B_0 = \frac{B_1^2 B_2^2}{b_c^3} \tag{2-6}$$

采用现代统计估值理论进一步研究这一问题，便可得到一套改进的估值公式：

$$\hat{B}_0 = \frac{B_2(B_1+1)}{b_c} - 1 \tag{2-7}$$

$$\text{方差 } B_0 = \frac{(B_1 - b_c + 1)(B_0+1)(B_0 - B_1)}{b_c(B_1+2)} \tag{2-8}$$

由于 $B_1 \gg 1$，$B_2B_1/b_c \gg 1$，所以(2-7)式便可以简为(2-4)式。同样，在(2-8)中，$B_0 \gg 1$，又 $B_1 \gg 2$，该式变成：

$$\text{方差 } B_0 \approx \left(\frac{B_1}{b_c} - 1\right)\frac{B_0^2}{B_1}\left(1 - \frac{B_1}{B_0}\right) \tag{2-9}$$

若将(2-4)式代入(2-9)中，令 $B_1 > b_c$，$B_2 > b_c$，得到：

$$\text{方差 } B_0 \approx B_1^2B_2^2/b_c^3$$

因此，如果 B_1 和 B_2 大于1，且 b_c 比较小，(2-6)与(2-8)就一致了。

二、回归模型

预测任何时间函数未来特性时广泛采用的一个方法是测定前面的一些数据，描出数据的曲线，然后推测出曲线的未来部分。往往用一条曲线表示那些数据，再用最小二乘法确定曲线的参数，使其与这些数据能够取得最好的吻合。在统计学中，这一过程已经标准化，称为回归分析。如果数据模型是一直线，则采用线性回归分析，求得直线的斜率和截距。如果是多项式模型，则采用多项式回归，求得多项式系数。

如果我们有一组数据，就可画出随时间变化的累积错误曲线。如图 2.1 所示，图中的三条曲线分别表示三个项目按月排错的累积数。Ellingson 使用回归分析方法处理累积排错数，并预测在未来开发阶段中要排除的错误数。Coutinho 把曲线画在对数座标线上，来研究累积错误的特性。Nathan 则试图利用增长曲线的概念给出了软件随错误的消除而逐步改进的模型。

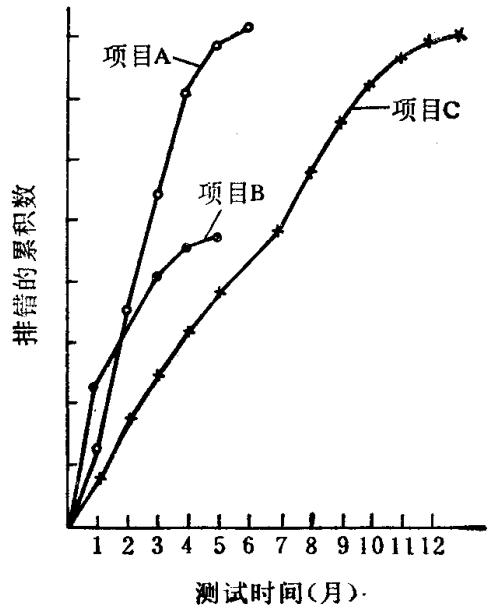


图 2.1 排错累积曲线

2.3 软件质量因素和质量特性

既然我们的各项软件工程活动，都朝着希望得到高质量软件这样一个总目标，我们就必须真正弄清楚，究竟什么是软件质量。由于计算机软件是极为复杂的产品，它的质量很难用一句话表达清楚。不过我们可以从以下三个方面把握软件质量的概念：

① 软件需求是衡量软件质量的基础。如果开发出的软件与需求不一致，就谈不到软件的质量。

② 规定了的标准是软件开发必须遵循的准则，如果软件项目未能按标准开发，软件质量必定是低劣的。

③ 软件通常有着一些不作明文规定的隐含需求。例如，要求软件有较好的可维护性。如果已开发的软件已经满足了那些明文规定的需求，却没有满足那些隐含的需求，那么软件产品的质量仍然是有问题的。

事实上，软件质量是多种因素的混合体，或者说是多种因素的综合体。这些因素可能因不同的应用方面和不同的用户观点而有所变化。我们可以粗略地把影响软件质量的因素分为两类：一类是可直接度量的因素，例如，单位时间内每千行源代码所发现的错误个数；另一类则是只能间接度量的因素，例如，可复用性、可维护性等。不论哪一类，都必须能够度量，都应能以具体数据表达软件质量的不同方面。

McCall 提供了影响软件质量因素的实用分类方法。这一分类法是从软件产品的三个重要方面出发的，即软件的运行特性、软件经受修改的能力以及软件适应新环境的能力。图 2.2 便从这三方面表达了有关软件质量的因素。

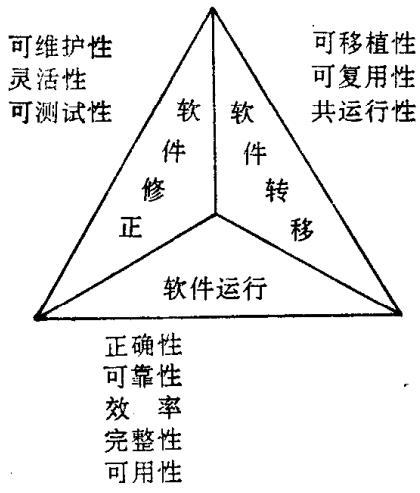


图 2.2 软件质量因素

以下对各质量因素作一简要说明：

① 软件的运行特性

正确性 (Correctness)：软件能满足其规格说明及完成客户提出任务要求的程度。针对正确性，可以问：该软件是按要求去做的吗？

可靠性 (Reliability)：软件能按规定的精确度履行其预期职能的程度。针对软件可靠性，可以提出问题：该软件能够总是精确地工作吗？(有关可靠性问题将在本章后面的部分作更为深入的讨论)

效率 (Efficiency)：为履行软件的职能所需要的计算资源和代码的数量，某个软件是否能在我们机器上运行，内存、外设容量是否够用？

完整性 (Integrity)：控制未经允许人员使用软件或数据的能力。这个软件产品是安全的吗？会被人盗用吗？

可用性 (Usability)：为了掌握该软件、学会操作、为其运行准备输入数据以及解释其输出数据要花多大工作量。任一用户可能提问：此软件容易掌握吗？

② 软件的修正特性

可维护性 (Maintainability)：找到软件错误发生的位置并加以修正所花的工作量大小。用户会问：这个软件能够很容易理解、纠错、改写或扩充吗？能够维修它吗？

灵活性 (Flexibility)：修改某个运行的软件要付出的工作量。用户会问：可以改它吗？容易改吗？

可测试性 (Testability)：为确保软件履行其职能而进行测试所需的工作量。用户会问：可以测试它吗？容易测试吗？

③ 软件转移特性

可移植性（Portability）：从某个硬件和(或)软件系统环境下把一个程序转移到另一环境下所需的工作量。用户常常会提出问题：可以在别的机器上使用这个软件吗？

表 2.2（a） 软件质量因素的特性

特　性	含　意	相关因素
可跟踪性 Traceability	该特性为特定开发项目或运行环境提供了从需求到实现的完整线索	正确性
安全性 Completeness	所要求功能的全面实现	正确性
兼容性 Consistency	统一的设计与实现技术和表达方法	正确性 可靠性 可维护性
精确度 Accuracy	为计算和输出提供了所要求的精确度	可靠性
容错性 Error Tolerance	在非标定条件下继续运行的能力	可靠性
简明性 Simplicity	以最容易理解的方式实现规定的功能（避免复杂化）	可靠性 可维护性 可测试性
模块性 Modularity	独立性很强的模块形式的结构	可维护性、灵活性，可测试性、可移植性、可复用性、共运行性
通用性 Generality	所完成功能的广泛性	灵活性 可复用性
可扩充性 Expandability	数据存储需求和计算功能的扩展能力	灵活性
自检性 Instrumentation	软件监察自身运行情况及发现自身错误的能力	可测试性
自描述性 Self-Descriptiveness	软件解释自身功能被监察情况的能力	灵活性、可维护性、可测试性、可移植性、可复用性
执行效率 Execution Efficiency	软件的最小处理时间	效率
存储效率 Storage Efficiency	软件运行时的最小存储需求	效率
存取控制 Access Control	对软件和数据存取的控制	完整性
存取审查 Access Audit	审查软件和数据的存取	完整性
可操作性 Operability	决定与软件运行有关的操作和步骤，即操作软件的容易程度	可用性
培训 Training	软件协助新用户掌握使用它的能力	可用性
通讯性 Communicativeness	可被吸收的有用输入和输出	可用性
软件系统独立性 Software System Independence	对软件工作环境(如操作系统、实用程序等)的依赖性	可移植性 可复用性
机器独立性 Machine Independence	软件对硬件系统的依赖程度	可移植性 可复用性
相互通讯性 Communications Commonality	使用标准协议及标准接口子程序等的程度	共运行性
数据公用 Data Commonality	使用标准数据表示的程度	共运行性
简洁性 Conciseness	使用最少的代码实现某一功能的能力，即紧凑程度	可维护性

可复用性（Reusability）：某个软件（某个程序或程序的一部分）可以在其它题目中再次利用的程度。用户会问：它能重复使用吗？

共运行性（Interoperability）：某个软件与另一软件联合起来协调工作所需的工作量。用户可能提问：可以把这个软件与另外的软件对接吗？要能作到这样得花多少工作量？

以上提供的质量因素确是从不同的方面反映了软件质量要求，但要注意到，这些质量因素往往很难，甚至不可能直接度量。为此必须作进一步分析。

我们知道，质量因素可以分解成一些独立的质量特性（Quality Characteristics），这些特性是很容易度量的，在这里引出质量特性是考虑到：

① 质量特性为质量因素提供了更完全、更具体的定义。

② 它的引进有助于说明各质量因素之间的关系。

③ 质量特性的提出方便了质量检查和质量的定量观测。

④ 质量特性使我们能够准确地决定质量因素的范围。

表 2.2(a) 列出了软件质量特性，表中同时给出了简要说明以及与其相关的质量因

表 2.2（b） 质量因素与质量特性的关系

质量特性 \ 质量因素	正确性	可靠性	效率	完整性	可用性	可维护性	可测试性	灵活性	可移植性	可复用性	共运行性
可跟踪性	○					○	○	○		○	
完全性	○	○			○						
兼容性	○	○				○	○	○		○	
精确性		○	△		○						
容错性	○	○	△		○						
简明性	○	○	○			○	○	○	○	○	
模块性			△			○	○	○	○	○	○
通用性		△	△	△				○		○	○
可扩充性			△					○		○	
自检性			△		○	○	○				
自描述性			△			○	○	○	○	○	
执行效率			○						△		
存储效率			○				△		△		
存取控制			△	○	○			△			△
存取审查			△	○							
可操作性			△		○					○	
培训					○					○	
通讯性			△		○	○	○	○		○	
软件系统独立性			△					○	○	○	○
机器独立性			△					○	○	○	○
相互通讯性											○
数据公用性				△						○	○
简洁性	○		○								

注：表中△表示对质量因素有负向影响；
○表示对质量因素有正向影响。

素。为了更清楚地表明质量因素与质量特性间的关系，这里又给出了表 2.2(b) 我们可以从中看到哪些质量特性对哪些质量因素有着什么影响。

2.4 软件质量保证的任务

计算机发展的初期，比如 50 年代和 60 年代，软件的质量保证是程序人员的事情。70 年代，军用软件开发提出了软件质量的标准。以此为起点推行质量保证标准迅速在商用软件中展开。

软件质量保证（SQA-Software quality assurance）包括以下多方面的工作：

① 采用技术手段；

② 组织正式技术评审；

③ 软件测试；

④ 推行软件工程标准；

⑤ 对软件的变更进行控制；

⑥ 对软件质量进行度量；

⑦ 对软件质量情况及时记录和报告。

一个软件产品或系统的质量如何，是从设计开始就决定了的，它和整个软件的研制过程密切相关。质量的好坏不是研制出来以后可以任意改变的。因此软件质量保证活动必须从采用技术手段和工具开始。保证系统分析员获得一个高质量的规格说明书和高质量的设计。

在软件规格说明和设计资料完成以后，就要对它们的质量进行评价。这时质量评价的主要活动是正式技术评审（formal technical review）。正式技术评审是按特定形式举行的，技术人员参加评审会，其目的在于揭露软件的质量问题。

软件测试把多层次实施测试与一套测试用例设计方法结合起来，从而保证了有效的错误检测。软件开发人员把软件测试当作质量保证的重要手段。原因是测试可以发现软件中的大多数隐藏错误。不过，正如第一章所述，遗憾的是测试并不能发现所有的软件错误。

采用怎样的软件标准对各个开发部门是不同的，有的是应客户的要求，有的是上级管理部门规定的，也有的是软件开发部门自己制定的。正式的软件工程标准一旦得到确认，就应开展软件质量保证活动，使得标准被人们重视，并在软件开发中得到遵循。评价软件工程标准执行得怎样，可作为正式技术评审的一个内容。有些情况下，软件质量保证机构可以开展独立的审计活动。

对软件质量的一个不可忽视的威胁因素是来自软件的修改和变更，尽管表面上看修改和更动是有理由的和有益的，但实践证明，在修改过程中常常引进一些潜伏的错误，或是带来某些足以传播错误的因素。因此，严格控制软件的修改和变更自然成为十分必要的了。这包括严格掌握修改和变更的请求，仔细研究修改和变更的性质以及控制修改和变更对软件各部分和有关方面引起的冲击。这将是软件维护工作的一个重要问题。

度量是任何工程学科不可缺乏的一项重要活动。软件质量保证的目标是要跟踪软件的质量，也就必须进行软件度量（software metrics）。

对软件质量的记录与报告为收集和传播软件质量保证信息提供了手段。评审的结果、审计的结果、对修改与变更的控制、测试以及其它软件质量保证活动都必须成为软件工程项目历史记录的一部分，并传送给有关人员。例如，对软件设计的正式技术评审结果应按一定格式记录下来，与相关模块的全部技术资料和质量保证资料放在一起，保留起来。

软件评审（Review）是软件质量保证的一个重要手段。评审可针对开发工程的任何一个阶段，目的在于发现其中的隐藏错误，并加以排除。比如，在分析、设计、编码和测试等软件工程活动中，软件评审起到了检查与监督的作用。

目前多数的软件项目都是按传统的瀑布式模型开发的（见图 2.3 靠右的流水线），各个开发阶段依次向下连接。后一阶段紧紧依赖于前面阶段的工作成果。如果后一阶段继承了前面阶段的错误，常常不仅是简单的继承，而且还会扩展。这种扩展使后期的工作很被动。一方面很难发现问题的根源，另一方面，由于影响域的扩大，也很难加以排除。这样势必极大地增加软件开发的成本。为了及时地发现每个阶段工作中出现的错误，并不致给后阶段工作造成影响，在每个开发阶段之后安排一个评审活动。其目的在于未充分揭露出这一阶段工作中的问题时，暂不进入下一阶段。从图 2.3 中左边流水线可看出，各阶段之间穿插了需求评审、设计评审、程序编写评审等。自然这些评审活动将主要针对规格说明书、设计文档以及源程序进行。

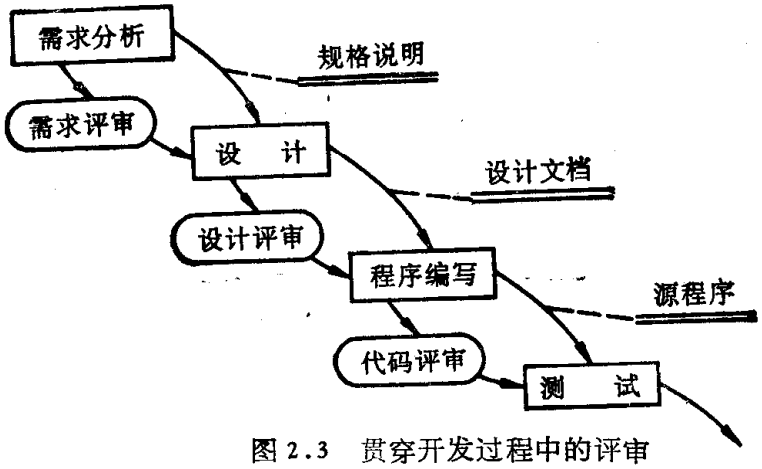


图 2.3　贯穿开发过程中的评审

许多软件开发部门总结了软件评审工作效益，他们一致认为，软件评审可以配合测试工作起到有效地揭露软件错误的作用。例如，设计活动常常带来整个软件工程项目 50% 到 65% 的错误，组织正式的设计评审最多可以找出 75% 的错误。这充分说明，评审不仅可以降低编码和测试的成本，也必定降低了软件维护的成本，对于保证软件质量作出了有益的贡献。IBM 公司对大型软件项目的开发作了统计分析，他们发现，在设计阶段假定发现和纠正一个错误要花一个货币单位的成本；同一个错误在测试开始时发现和纠正，要花 6.5 个单位；在测试的中期发现将花 15 个单位；而在测试结束，软件交付用户以后再发现和纠正，将要花费 67 个单位的成本。由此，足见软件评审的重要意义。

软件错误的扩展模型进一步说明了，在软件开发过程各个阶段中错误的发生、扩展和发现的一些实际情况。图 2.4 给出了这一模型的形式。矩形表示一个软件开发阶段。矩形的左半部分为三部分，分别表示：直接继承前一开发阶段的错误个数；部分错误被扩大M倍的情况；以及在该开发阶段新产生的错误。矩形的右半部表示该阶段找到错误的比率。

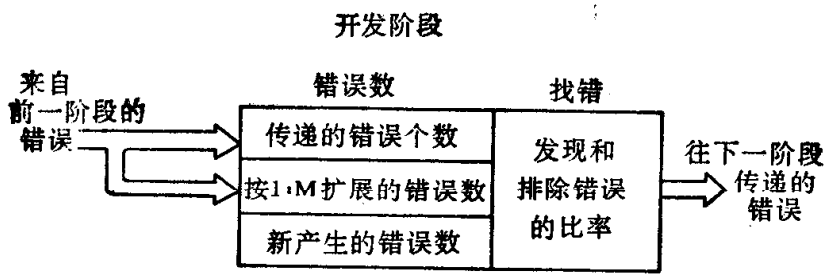


图 2.4　开发过程中错误模型

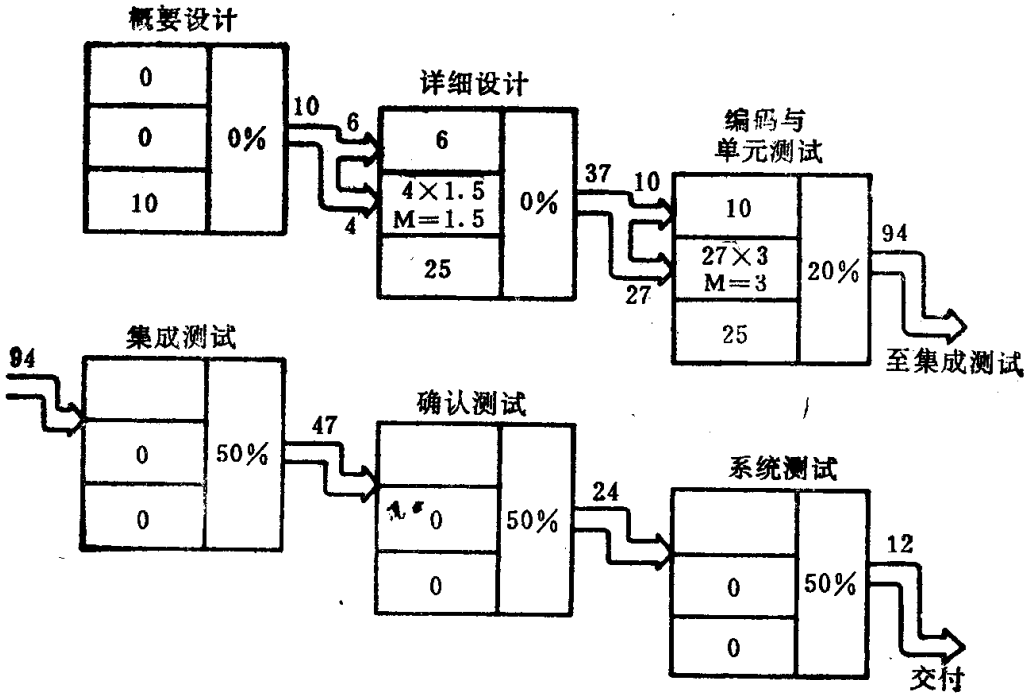


图 2.5　未经软件评审的错误扩展

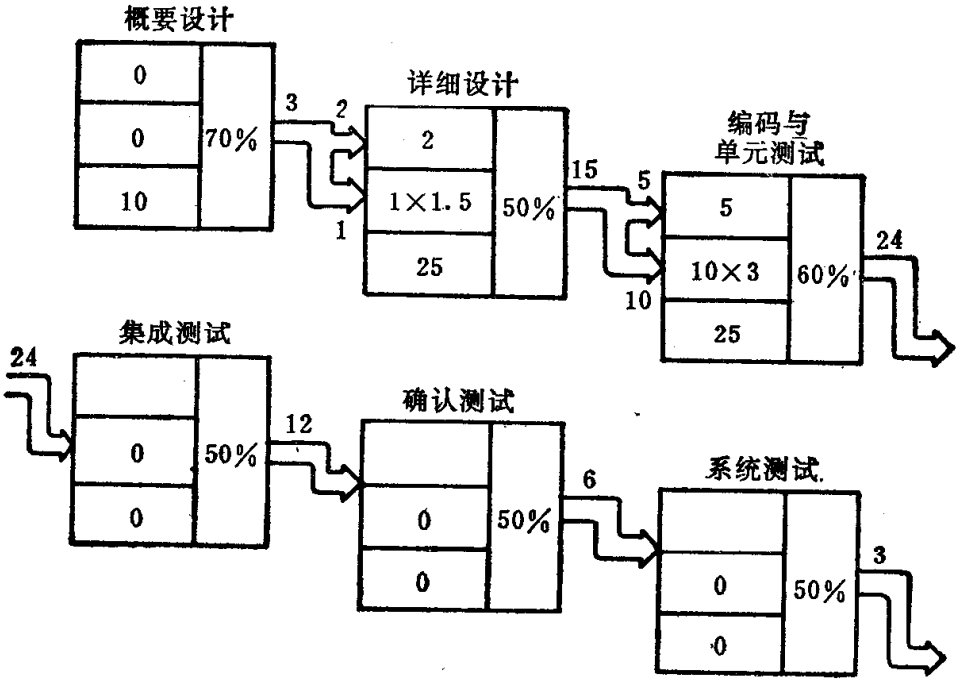


图 2.6　经过软件评审的错误扩展

图 2.5 表明了未经软件评审的错误扩展情况。图中的每个测试步骤假定发现和纠正 50% 的错误，并且未引进新的错误。这样，在概要设计阶段产生的 10 个错误到测试以前已经扩展成为 94 个。经过各种测试，逐步消除了一些，但最后仍然有 12 个错误隐藏起来，被交付给用户。图 2.6 则是同一个例子，只是进行了阶段评审。假定概要设计、详细设计和编码与单元测试的阶段评审分别消除了 70%、50% 和 60% 的错误，那么在开始测试以前，只有 24 个错误。再经过各种测试，便将大部分错误消除，最后仅存 3 个潜伏下来，被交付给用户。

软件评审对于软件质量的保证作用可以由此得到一些感性认识。

软件评审的形式多种多样。有关如何进行评审的问题可参看本书第三章人工测试一节。

2.5 程 序 排 错

前面一章中读者已看出排错与测试有着完全不同的含意，我们不能把两者混为一谈。严格地说，排错已超出了测试工作的范围。但这两者毕竟有着十分密切的关系，常常是在测试以后紧接着要着手排错。实际上，这两种工作还经常交叉进行。很显然，在医院接受体检和化验只是发现病因的手段，对症治疗才是目的。这与测试和排错的关系非常相似。鉴于本书主要以测试为中心论题，在以后几章详细介绍测试技术以前，本节只对排错的有关问题作一简要的描述。

一、排错工作概述

1. 什么是程序排错

关于什么是测试，第一章中已作了充分的说明。简单地说，测试只不过是一种检验。经过测试人们会看到一些现象，这些现象也许是令人可疑的错误征兆。我们常常不能直接从测试的结果中找出错误的根源，这就需要充分利用测试结果和测试提供的信息进行全面的分析。经过分析找到错误的根源和出现错误的原因。紧接着便是纠正已发现的错误。测试以后进行的这些工作称为排错。

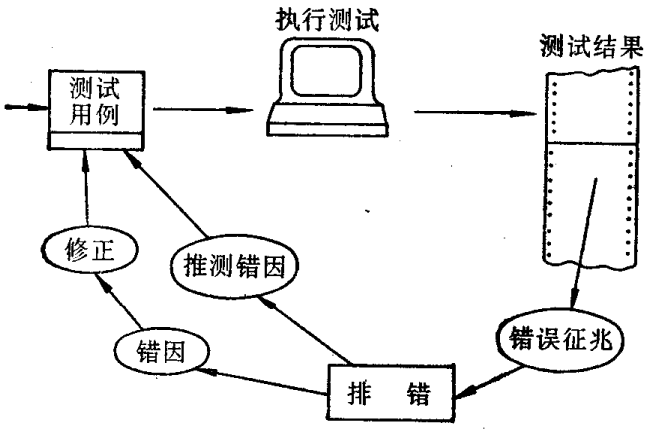


图 2.7 排错过程

如果进一步分析测试与排错的交叉关系，应该把它们看作一个循环过程。图 2.7 表示了这一过程。图中表明，根据测试结果得到错误征兆以后，开始进行排错。事实上，这时有两种情况，一是已经弄清了出错原因和错误根源，便可立即进行修正；另一情况是并未能确定出错原因，这便要作出推测，再根据所作的推测设计测试用例，再次进行测试。

2. 排错中遇到的困难

凡是具有软件开发实践经验的人都体会到，排错不是一件轻松的工作。排错过程中我们会遇到各种困难。这些困难包括：

① 必须克服心理因素的障碍

做好排错工作与人的素质有很大关系。有些人擅长排错工作，有些人则不行。实践表明，具有相同的教育背景和实践的人，在排错能力方面会有很大差异。Shneiderman 曾指出，排错是整个程序工作中比较棘手的一部分，它是在你不高兴承认出了错的情况下，不得不来解决问题。经常会出现的心情急躁，不愿意承认错误等等都会增加排错的困难。但当一个错误完全改正以后，会有一种如释重负的感觉，非常轻松和兴奋。

② 当改正了一个错误时，有可能引进了两个或更多的错误。这种情况又常常不容易发觉。自然，引入了新的错误将使问题更加复杂化。

③ 错误本身的一些特点也增加了排错的困难。以下一些情况可能会遇到：

• 我们从外部观察到程序出错的位置与程序内部导致出错原因的位置相距甚远，后者很难找到。

• 当其它错误改正后，这个错误所表现的现象可能暂时消失，但并未实际排除。

• 可能是非程序的原因（如舍入不当）引起的错误，不容易发现。

• 可能是时序问题引起的错误，而不是处理过程引起的错误，这也容易被忽视。

• 也许是人的行为错误；如操作错误、写错等，这些是不容易追踪的。

• 可能是由于难以精确地模拟的输入条件所引起的，如在实时应用中，其输入是持续不断的。

• 反映错误的现象不稳定，呈间歇状，时有时无，难以捕捉。

在这些困难面前，只靠正规的系统方法常常不能解决问题。难怪有人说，有时实践经验，甚至直觉和运气倒能起作用，帮助找到问题。

二、排错方法

这里简要地描述几种主要排错方法和它们的优缺点。

1. 内存信息转储（core dumps）

内存转储是在执行测试中出现问题以后，设法保留有关的现场信息。把所有寄存器和内存有关部分的内容打印出来，进行分析研究。必要时，这些信息要存档，供以后使用。

这一方法可以取得发现错误的关键信息，内存中的信息有时最能说明问题。当已经对程序中某个错误假设有了进一步认识，用这个方法协助作出推断，常常是很有效的。

但必须说明，使用内存转储需要一些主机时间，相当多的 I/O 时间以及分析时间。内存信息以十六进制码打印，辨认这些信息并且找出和源程序的对应地址也有一定困难。显然，必须有目标、有限制地使用这一方法，否则不恰当地使用会造成很大的浪费，而且也不会取得预计的效果。

2. 跟踪

跟踪和内存信息转储非常相似，只是除去打印内存信息、寄存器信息外，其打印还以某些事件发生为条件。典型事件包括进入、退出或是出现了：

① 特定子程序、语句、宏结构或数据库；

② 与终端、打印机、磁盘或是其它外部设备的通信；

③ 变量或表达式的值；

④ 实时系统中定时启动或随机启动。

跟踪程序有一个值得注意的问题是，上述这些条件均以源语言形式引入，它的任何修改都要求重新编译。

3. 打印语句

为取得关键变量的动态值，在程序中特定的位置上安排标准的打印语句。这是取得动态信息比较简单的方法，可以检验在某事件以后某个变量是否按预计的要求发生了变化。而且多个打印语句可以给出变量的动态特性。

4. 使用排错程序

开发专用的排错程序，使其与被测试的程序并行运行，它还为排错提供一些命令，如检查内存和寄存器，在特定点停止执行程序，对特定常量、变量和寄存器的引用进行查询。

排错程序可以设计成交互式的，这对于检查运算的动态特性具有相当大的灵活性。不过排错程序大多在机器语言程序上工作，若用于高级语言程序必须与解释程序结合。当然，这种工作方式多用于微型机而不是大型机。

三、排错策略

采用哪种排错策略是排错的中心环节，好的排错策略可以协助我们较快地找出错误的原因。这里提供几种常用的策略。

1. 试错法（Trial and error）

在分析出错征兆后，迅速地判断出错误可能发生的位置。选择一个或几个排错方法对程序实施排错处理，如打印相关的信息。很显然，这种作法常常是较慢，而且是低效的。

2. 回溯法（Backtracking）

回溯法也称向后追踪。考察错误征兆，从它在程序显露的位置起沿着程序的控制流向后（反方向）追踪，直到征兆消失处为止。再仔细分析邻接的程序段，进而找出错误的原因。

3. 向前追踪（Forwardtracking）

在程序中设置一些打印语句，输出某些关键变量的值，进而检查这些输出的中间结果，判断哪一点开始发生错误。

4. 二分查找（Binary-Search）逼近法

如果我们已经能够知道程序中某些关键点上变量应取的正确值，那就可以采用二分查找的办法。例如，我们在程序的中间部位引入一组输入，并检验其输出结果，如果输出结果是正确的，那就表示，错误发生在程序的前半部分。如果输出是错的，可以认为错误在程序的后半部分。反复这样的过程，进行多次，直至逼近到错误的准确位置。

5. 归纳法（Induction）

研究与出错相关的信息，找出特征，得到“原因假设”，然后确认或否认这个假设。归

纳法按以下步骤进行：

① 收集有用信息。列举已知失败的测试用例情况和成功的测试用例情况。弄清哪些是观察到的出错征兆，何时出现错误，什么情况下出现错误等。

② 确定出错的类型。检查收集到的信息，注意区别失败的测试用例和成功的测试用例间的差别。

③ 构想出一个或若干假设。根据观察到的关系，推出一个或多个假设。如果假设并不是显而易见的，再来考察已有信息，收集更多的其它信息，也许还需要运行更多的测试用例。如果有几个假设，则将其按成立的可能性排列，最大可能的假设列在前面。

④ 审查假设，看其能否成立。再次检查有关信息，以便确定假设能否解释观察到的问题的各种表现。注意不要忽视可能有多个错误同时存在。这一步若未完成，不要往下进行。

⑤ 作适当的修正。根据假设作出推断，对相关的代码实施修正。

⑥ 验证修正。再次运行前次失败的测试用例，以确认的确消除了错误的征兆。并且还要使用另外一些测试用例运行，以增加信心。此外应该再次运行以前成功的测试用例，目的是要保证所作的修正不会引起新的问题。如果修正是成功的，便可用修正后的程序当作主文本，并删除原来的程序文本。若修正不成功，则需返回第一步。

图 2.8 形象地表达了归纳法进行排错的过程。

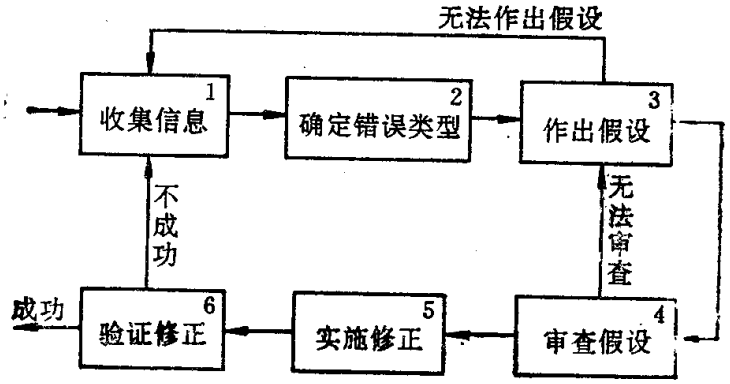


图 2.8 归纳法排错过程

6. 演绎法 (Deduction)

演绎法正好和归纳法的过程相反，首先列举一些可能的原因或假设，然后再进行逐个分析，排除那些不能确立的原因和假设，直到仅剩下一个被证实。大致可按以下步骤进行：

① 枚举若干可能的原因和假设。

首先列出所有可能的出错原因，这些原因并不要求有充分的理由，只是提供一些分析的线索。

② 利用掌握的资料排除一些原因。

仔细地分析已掌握的资料，找出一些矛盾，力图排除那些不能成立的原因。但如果所有的原因都被排除，就应另作推测。如果不是一个原因保留下来，就应把最大可能的原因

列为优先考虑的对象。

③ 精心研究保留的假设。

这时保留的可能原因也许是对的，但对于确认错误可能还不够充分。要充分利用已知的线索作精心的研究，其中估计到一些特殊的情况。

④ 证明保留的假设。

这和归纳法的最后一步是相同的。

图 2.9 表示了演绎法排错的过程。

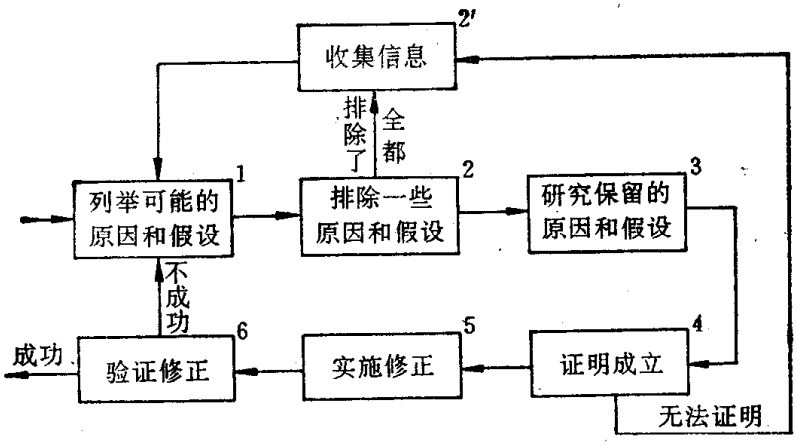


图 2.9 演绎法排错过程

参 考 文 献

[1] Myers, 计算机软件测试技巧，清华大学出版社，周之英、郑人杰译，1985。

[2] Martin L. Shooman, Software Engineering, McGraw-Hill, 1983.

[3] Boris Beizer, Software Testing Techniques, 2nd Ed., Van Nostrand Reinhold Company, 1990.

[4] F. P. Brooks, The Mythical Man-month, Addison-Wesley, 1972.

[5] Cem Kaner, Testing Computer Software, TAB Books Inc, 1988.

[6] James Vincent et al., Software Quality Assurance, Vol. 1, Prentice Hall, 1988.

[7] Roger S. Pressman, Software Engineering——A Practitioner's Approach, Second Edition McGraw-Hill, 1987.

[8] Edited by G. Gordon Schulmeyer and James I. McManus, Handbook of Software Quality Assurance, Van Nostrand Reinhold, 1987.

第三章　软件测试策略

为了检验开发的软件能否符合规格说明书的要求，测试活动可以采用各种不同的策略。这些策略的区别在于它们表明了不同的出发点、不同的思路以及采用不同的手段和方法。本章将针对常用的若干软件测试策略，以对照比较的方式进行阐述，这包括：静态方法与动态方法；黑盒方法与白盒方法；随机测试与穷举测试；测试步骤(单元测试、集成测试、确认测试与系统测试)；自顶向下测试与自底向上测试；累进测试与非累进测试；以及人工测试等专题。

3.1　静态方法与动态方法

原则上讲，我们可以把软件测试方法分为两大类，即静态方法和动态方法。

静态方法的主要特征是不利用计算机运行被测试的程序，而是采用其它手段达到检测的目的。但上述静态方法的特征并不意味着完全不利用计算机作为分析的工具。它与本章第4节介绍的人工测试有着根本的区别。

本节主要给出静态分析的基本概念，动态测试将在后面第四章和第五章全面地展开，在此不作讨论。掌握两种方法的差别则是本节的主要目标。

静态分析是对被测程序进行特性分析的一些方法的总称。这些方法本身各有自己的目标和步骤。比如，有的是要收集一些程序信息，以利于查找程序中的各种欠缺和可疑的程序构造；有的只是从程序中提出语义的或结构要点，供进一步分析；或是以符号代替数值求得程序的结果，便于对程序进行运算规律的检验；以及对程序进行一些处理，为进一步动态分析作准备等等。

对于静态分析在软件测试中究竟占据什么地位，许多人有不同的见解。原因在于，人们已经开发出一些静态分析系统作为软件测试工具，静态分析被当作一种自动化的代码检验方法。对于软件开发人员来说，静态分析只是进行动态分析的预处理工作。他们认为，静态分析并不是要找出程序中的错误，因为编译系统已经能够做到这一点了。实际上，这种看法是片面的，尽管编译系统也能发现某些程序错误，但这些远非软件中存在的大部分错误。静态分析的查错功能是编译程序所不能代替的。为了说明这一点，以下列出静态分析可以做到的一些工作：

- 可能发现的程序欠缺：
  - 用错的局部变量和全程变量；
  - 不匹配的参数；
  - 不适当的循环嵌套和分支嵌套；

不适当的处理顺序；
无终止的死循环；
未定义的变量；
不允许的递归；
调用并不存在的子程序；
遗漏了标号或代码；
不适当的连接。

- 找到潜伏着问题的根源：
  未使用过的变量；
  不会执行到的代码；
  未引用过的标号；
  可疑的计算；
  潜在的死循环。
- 提供间接涉及程序欠缺的信息：
  每一类型语句出现的次数；
  所用变量和常量的交叉引用表；
  标识符的使用方式；
  过程的调用层次；
  违背编码规则。
- 为进一步查错作准备
- 选择测试用例
- 进行符号测试

为使读者对静态分析有进一步的理解，请参阅本书第七章 7.1 节测试工具综述及 7.2 节 COSTE 系统的静态分析工具部分。

3.2 黑盒测试与白盒测试

黑盒测试与白盒测试是很广泛使用的两类测试方法。

黑盒测试（Black-box Testing）又称功能测试、数据驱动测试或基于规格说明的测试（Specification-based Testing）。用这种方法进行测试时，被测程序被当作打不开的黑盒，因而无法了解其内部构造。在完全不考虑程序内部结构和内部特性的情况下，测试者只知道该程序输入和输出之间的关系，或是程序的功能（图 3.1）。他必须依靠能够反映这一关系和程序功能的需求规格说明书考虑确定测试用例，和推断测试结果的正确性。即所依据的只能是程序的外部特性。因此，黑盒测试是从用户观点出发的测试。

白盒测试（White-boxTesting）又称结构测试、逻辑驱动测试或基于程序的测试（Program-based Testing）。采用这一测试方法，测试者可以看到被测的源程序，他可用以分析程序的内部构造，并且根据其内部构造设计测试用例。这时测试者可以完全不顾程序的功能。

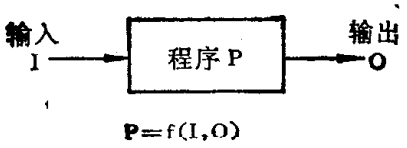


图 3.1 黑盒测试

这两类测试方法是从完全不同的起点出发，并且是两个完全对立的出发点，可以说反映了事物的两个极端。两类方法各有侧重，在测试的实践中都是有效和实用的。我们不能指望其中的一个能够完全代替另一个。在进行单元测试时大都采用白盒测试，而在确认测试或系统测试中大都采用黑盒测试。

以下将分别介绍这两类测试方法，并作对比。

一、黑盒测试

如上所述，黑盒测试是一类重要的测试方法，它因根据规格说明设计测试用例，并不涉及程序的内部构造而得其名。它是一类传统的测试方法，有着严格规定和系统的方式可供参考。但并非采用这类方法在实践中就不存在问题了。

一个突出的问题是要弄清楚，所谓程序的功能究竟是哪些？我们知道，任何软件作为一个系统都是有层次的。在软件的总体功能之下可能有若干个层次的功能，而测试人员常常只看到低层的功能。他们面临的一个实际问题是在哪个层次上进行测试。如果仅在高层次上进行测试，就可能忽略一些细节。若是在低层次上展开测试，又可能忽视各功能之间存在的相互作用和相互依赖的关系。看来，测试人员需要考虑并且兼顾各个层次的功能。如果为测试人员提供的是一个不分层次的杂乱的规格说明，那么他的黑盒测试工作必定要陷入毫无希望的混乱之中，也就不可能取得良好的测试效果。

黑盒测试的另一问题是功能生成。软件开发总是从把原始问题变换成计算机能处理的形式开始的，接着还要进行一系列变换，直至得到编码的程序。在这一系列变换的过程中，每一步都得到不同形式的中间成果。例如，一开始要把原始数据变成表格形式的数据，然后又变成文件上的记录。在此过程中便出现了功能。首先是填表，然后是输入、输出。在送入计算机以后又会出现安全保密、口令、恢复及出错处理等功能。

如果规格说明书是按高层抽象编写的，那就不会涉及到许多具体的技术性功能，如文件处理、出错处理等。如果测试用例是根据这样的规格说明得到的，那么测试便不可能是完全的和充分的。另一种情况，如果规格说明是按低层抽象编写的，其中必定包含许多技术细节。对于这样的规格说明，用户是非常为难的，因为他们无法理解其中的一些技术细节，也就无法判断这个规格说明是否反映了他的真正需求。为了解决这一矛盾，有人建议写出两份规格说明书，一份给用户用，另一份给测试人员用。但即使这样，问题也并没有真正解决，因为很难保证这两份说明书完全一致。

由于这一情况，很长一段时间学术界对黑盒测试抱着不信任的态度。1980 年 W. E. Howden 恢复了功能测试的活力，他发表了题为 "Life Cycle Software Validation" 的文章。文中指出，近年来软件开发出现了一些比较严谨的设计方法，在这当中功能测试完全可以

发挥作用，或是和其它方法结合起来发挥作用。

从策略上说，重要的是要发展可靠的并且高效的功能测试方法。因为，功能测试不仅能够找到大多数其它测试方法无法发现的错误，而且一些外购软件、参数化软件包以及某些生成的软件，由于无法得到源程序，用其它方法进行测试是完全无能为力的。

这里需要说明的是，正是因为黑盒测试的测试数据是根据规格说明书决定的，这一方法的主要缺点则是它依赖于规格说明书的正确性。但实际上，我们并不能保证规格说明书是完全正确的。例如，在规格说明书中规定了多余的功能，或是漏掉了某些功能，这对于黑盒测试来说是完全无能为力的。

以上概括地介绍了几个黑盒测试方法的主要思想，其中的一些主要方法本书第四章将作更详细的描述。

二、白盒测试

前已说明，白盒测试是根据被测程序的内部结构设计测试用例的一类测试。有人也称它为透明盒或玻璃盒测试。因为它涉及到的是软件设计的细节。从道理上讲只涉及到被测源程序，但有时也会用到设计信息。按结构测试来理解，它要求对某些程序的结构特性作到一定程度的覆盖，或说"基于覆盖的测试"。这是从最早所谓"测试整个程序"的原始概念发展而来的。重视测试覆盖率的度量，可能减少测试的盲目性，并引导我们朝着提高覆盖率的方向努力，从而找出那些已被忽视的程序错误。

最为常见的程序结构覆盖是语句覆盖。它要求被测程序的每一可执行语句在若干次测试中尽可能都检验过。这是最弱的逻辑覆盖准则。进一步则要求程序中所有判定的两个分支尽可能得到检验，即分支覆盖或判定覆盖。当判定式含有多个条件时，可以要求每个条件的取值均得到检验，即条件覆盖。在同时考虑条件的组合值及判定结果的检验时，我们又有判定/条件覆盖。在只考虑对程序路径的全面检验时，可使用路径覆盖准则。所有这些逻辑覆盖准则都将在第五章中进行详细的讨论。

为取得被测程序的覆盖情况，最为常用的办法是在测试前对被测程序进行预处理。预处理的主要工作是在其重要的控制点插入"探测器"。这将在第五章 5.6 节程序插装中讨论。

必须说明，无论哪种测试覆盖，即使其覆盖率达到百分之百，都不能保证把所有隐藏的程序欠缺都揭露出来。对于某些在规格说明中已有明确规定，但在实现中被遗漏的功能，无论哪一种结构覆盖也是检查不出的。因此，提高结构的测试覆盖率只能增强我们对被测软件的信心，但它绝不是万无一失的。

三、黑盒测试与白盒测试的比较

前面已经对黑盒测试与白盒测试分别作了说明，现在可以抓住它们的实质，加以对比。

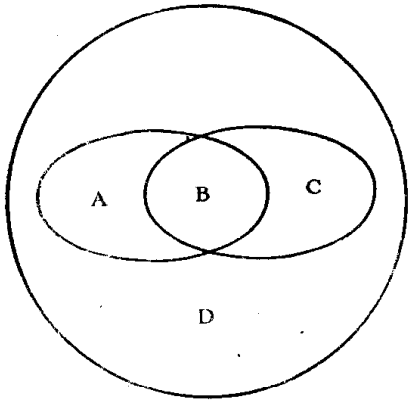
既然黑盒测试是以用户的观点，从输入数据与输出数据的对应关系出发进行测试的，也就是根据程序外部特性进行的测试。它完全不涉及到程序的内部结构。很明显，如果外部特性本身有问题或规格说明的规定有误，用黑盒测试方法是发现不了的。另一方面，

白盒测试完全与之相反，它只根据程序的内部结构进行测试，而不考虑外部特性。如果程序结构本身有问题，比如说程序逻辑有错误，或是有遗漏，那是无法发现的。如果用“黑”与“白”来对比不能明确地表达两者关系，那么用“外”与“内”来对比就再清楚不过了。因为，“外”与“内”的说法更能揭露两类方法的对立本质。可以从这一对比中看出它们各自的优缺点，以及它们之间的互补关系。

表 3.1 给出了黑盒测试与白盒测试两类方法的对比。图 3.2 则表明了它们各自的能

表 3.1 黑盒测试与白盒测试的对比

		黑盒测试	白盒测试
测试依据		根据用户能看到的规格说明，即针对命令、信息、报表等用户界面及体现它们的输入数据与输出数据之间的对应关系，特别是针对功能进行测试。	根据程序的内部结构，比如语句的控制结构，模块间的控制结构以及内部数据结构等进行测试。
特点	优 点	能站在用户立场上进行测试。	能够对程序内部的特定部位进行覆盖测试。
	缺 点	① 不能测试程序内部特定部位。 ② 如果规格说明有误，则无法发现。	① 无法检验程序的外部特性。 ② 无法对未实现规格说明的程序内部欠缺部分进行测试。
方法举例		等价类划分 边值分析 因果图	语句覆盖 判定覆盖 条件覆盖 判定/条件覆盖 路径覆盖 模块接口测试



A	只能用黑盒测试发现的错误
C	只能用白盒测试发现的错误
B	用黑盒测试或白盒测试都能发现的错误
D	黑盒测试与白盒测试均无法发现的错误
A+B	能用黑盒测试发现的错误
B+C	能用白盒测试发现的错误
A+B+C	用两种测试能发现的错误
A+B+C+D	软件中的全部错误

图 3.2 黑盒测试与白盒测试能够发现的错误

力范围，它们的互补关系以及各自的不足。

如果要求被测的软件“做了所有它该做的事，却没有做一点它不该做的事”，则需要把黑盒测试与白盒测试结合起来。

3.3 测试步骤

本书第一章中曾以瀑布模型描述软件工程过程，为了说明软件测试策略，还可以把这个过程表达成一个螺旋(参看图 3.3)。首先，系统工程为软件开发规定了任务，从而把它与硬件要完成的任务明确地划分开。接着便是进行软件需求分析，决定被开发软件的信息域、功能、性能、限制条件并确定该软件项目完成后的确认准则。沿着螺线向内旋转，将进入软件设计和代码编写阶段。从而使得软件开发工作从抽象逐步走向具体化。

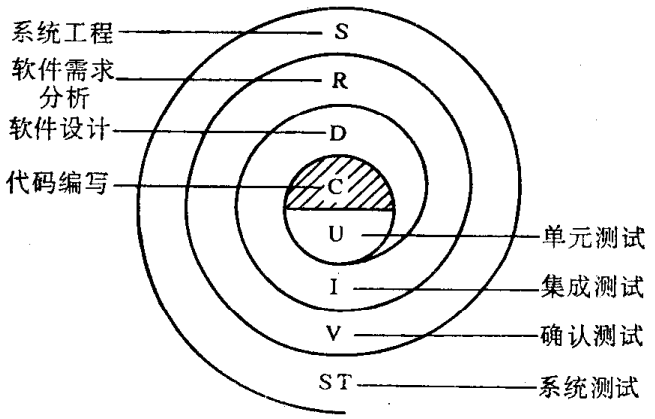


图 3.3 软件测试策略

软件测试工作也可从这一螺旋线上体现出来。在螺线的核心点针对每个单元的源代码，进行单元测试。在各单元测试完成以后，沿螺线向外前进，开始针对软件整体构造和设计的集成测试。然后是检验软件需求能否得到满足的确认测试，最后，来到螺线的最外层，把软件和系统的其它部分协调起来，当作一个整体，完成系统测试。这样，沿着螺旋线，从内向外，逐步扩展了测试的范围。

以上用螺旋线表明的测试过程，按四个步骤进行，即单元测试、集成测试、确认测试和系统测试。图 3.4 表示了测试的 4 个步骤。开始是分别完成每个单元的测试任务，以确保每个模块能正常工作。单元测试大量地采用了白盒测试方法，尽可能发现模块内部的程序差错。然后，把已测试过的模块组装起来，进行集成测试。其目的在于检验与软件设计相关的程序结构问题。这时较多地采用黑盒测试方法来设计测试用例。完成集成测试以后，要对开发工作初期制定的确认准则进行检验。确认测试是检验所开发的软件能否

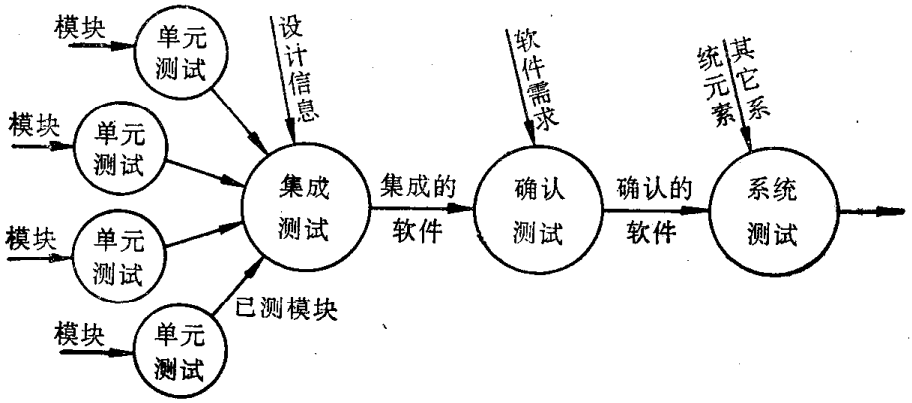


图 3.4 软件测试步骤

满足所有功能和性能需求的最后手段，通常均采用黑盒测试方法。完成确认测试以后，给出的应该是合格的软件产品，但为检验它能否与系统的其它部分（如硬件、数据库及操作人员）协调工作，需要进行系统测试。严格地说，系统测试已超出了软件工程的范围。

以下就测试步骤的几个环节作进一步说明。

一、单元测试

单元(unit)是程序的最小组成单位，它具有以下的特征：

① 通常可分配给某个程序人员开发，在单元之间或与外界除去有技术接口，如相互调用，引用数据库以外，总会有人员工作的界面需要协调。

② 单元接受数据输入后，经过加工，得到一些结果，这可能给出输出数据，也可能仅仅发生一些状态的改变。但如果输入、加工和输出三者缺少任何一个，这个程序单元都不是完整的。

③ 原则上说，每个程序单元都应有正规的规格说明，使之对其输入、加工和输出的关系作出明确的描述。

比程序单元更大的程序单位，其定义更加模糊。但我们常用的一个名称——模块，是为大家所熟悉的。在软件开发前期的概要设计阶段，通常把整个软件设计成一个树状的层次结构。处在树结构每个节点上的则是程序模块。什么是模块？也没有严格的定义，不过按一般的理解，模块应该具有以下的基本属性：

- 名字；
- 明确规定了的功能；
- 内部使用的数据，或称局部数据；
- 与其它模块或与外界存在的数据联系；
- 实现其特定功能的算法；
- 可被其上层模块调用，在其工作过程中也可调用其下属模块协同工作。

模块大小并没有明确的限定，它可处在树结构的上层，也可处在底层。我们可以认为基层模块就是程序单元，或是由程序单元构成。有时不加区分，默认单元测试就是模块测试也不会造成误解。

单元测试是要检验程序最小单位有无错误，它是在编码完成后，首先要施行的测试工作。通常由编码人员自己来完成，因而有人把编码与单元测试考虑成一个开发阶段。单元测试大多从程序的内部结构出发设计测试用例，即多采用白盒测试方法。多个程序单元可以并行地独立开展测试工作。以下集中在单元测试要解决的问题和单元测试的步骤两方面作进一步地说明。

① 单元测试要解决的问题

单元测试是要针对每个模块的程序，解决以下 5 个方面的问题(参看图 3.5)

- 模块接口——对被测的模块，信息能否正常无误地流入和流出。
- 局部数据结构——在模块工作过程中，其内部的数据能否保持其完整性，包括内部数据的内容、形式及相互关系不发生错误。
- 边界条件——在为限制数据加工而设置的边界处，模块是否能够正常工作。

·覆盖条件——模块的运行能否作到满足特定的逻辑覆盖。

·出错处理——模块工作中发生了错误，其中的出错处理设施是否有效。

模块与其周围环境的接口有无差错应首先得到检验，否则其内部的各种测试工作也将是徒劳的。Myers 提供的模块接口检查表是很有用的，以下简要地引出：

1）模块接受的输入参数个数与模块的变元个数是否一致？

2）参数与变元的属性是否匹配？

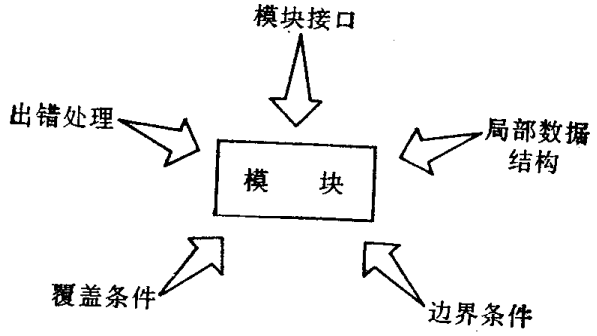


图 3.5　单元测试

3）参数与变元所使用的单位是否一致？

4）传送给另一被调用模块的变元个数与参数的个数是否相同？

5）传送给另一被调用模块的变元属性与参数的属性是否匹配？

6）传送给另一被调用模块的变元，其单位是否与参数的单位一致？

7）调用内部函数时，变元的个数、属性和次序是否正确？

8）在模块有多个入口的情况下，是否有引用与当前入口无关的参数？

9）是否会修改只是作为输入值的变元？

10）出现全程变量时，这些变量是否在所有引用它们的模块中都有相同的定义？

11）有没有把常数当作变量来传送？

当模块执行了外部的输入、输出时，Myers 提出还需考虑：

1）文件属性是否正确？

2）OPEN 语句是否正确？

3）格式说明与输入、输出语句给出的信息是否一致？

4）缓冲区的大小是否与记录的大小匹配？

5）是否所有的文件在使用前均已打开了？

6）对文件结束条件的判断和处理是否正确？

7）对输入、输出错误的处理是否正确？

8）有没有输出信息的正文错误？

对于局部数据结构应该在单元测试中注意发现以下几类错误：

1）不正确的或不相容的说明。

2）不正确的初始化或省缺值。

3）错误的变量名，如拼写错或缩写错。

4）不相容的数据类型。

5）下溢、上溢或是地址错误。

除局部数据结构外，在单元测试中还应弄清楚全程数据（如 FORTRAN 的 COMMON）对模块的影响。

如何设计测试用例，使得模块测试能够高效率地发现其中的错误，这是非常关键的问

题。无论考虑何种逻辑覆盖都应注意发现以下一些典型的计算错误：

1）对运算优先性的错误理解，或是错误的处理。

2）运算方式（mode）未加区分，发生了混合运算的情况。例如，实型量和复型量混淆。

3）初始化错误。

4）计算精度不够。

5）表达式中符号表示的错误。比较和控制流常常是彼此密切相关的，比较的错误势必导致控制流的错误。

需要特别注意发现的错误包括：

1）不同数据类型的数据进行比较。

2）逻辑运算符或其优先级用错。

3）本应相等的数据，由于精确度原因而不相等。

4）变量本身或是比较有错。

5）循环终止不正确，或循环不已 。

6）在遇到发散的循环时，不能摆脱出来。

7）循环控制变量修改有错。

程序运行中出现了异常现象并不奇怪，良好的设计应该预先估计到投入运行后可能发生的错误，并给出相应的处理措施，使得用户不致于束手无策。检验程序中出错处理这一问题解决得怎样，可能出现的情况有：

1）对运行发生的错误描述得难以理解。

2）指明的错误并非实际遇到的错误。

3）出错后，尚未进行出错处理便引入系统干预。

4）意外的处理不当。

5）提供的错误信息不足，以致无法找到出错的原因。

边界测试是单元测试的最后一步，是不容忽视的。实践表明，软件常常在边界地区发生问题。例如，处理 n 维数组的第 n 个元素时很容易出错，循环执行到最后一次执行循环体时也可能出错。这可按前面讨论过的，利用边值分析方法来设计测试用例，以便发现这类程序错误。

② 单元测试的步骤

单元测试常常被当作代码编写的附属步骤，它是在完成了程序编写，经过了复查，确认没有语法错误以后，针对每个程序模块单独进行的测试工作。

由于每个模块在整个软件中并不是孤立的，在对每个模块进行单元测试时，也不能完全忽视它们和周围模块的相互联系。为模拟这一联系，在进行单元测试时，需设置若干辅助测试模块。辅助模块有两种，一种是驱动模块(driver)，用以模拟被测模块的上级模块。另一种是桩模块(stub)，用以模拟被测模块工作过程中所调用的模块。图 3.6 表示了一个被测模块进行单元测试时的环境状况。其中设置了一个驱动模块和 3 个桩模块。驱动模块在单元测试中接受测试数据，把相关的数据传送给被测模块，启动被测模块，并打印出相应的结果。桩模块由被测模块调用，它们仅作很少的数据处理，例如打印入口和返回，以便

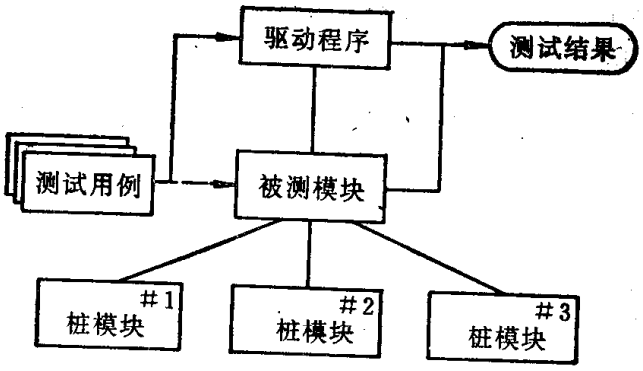


图 3.6　单元测试环境

于检验被测模块与其下级模块的接口。

自然，驱动模块和桩模块是一种额外的负担。就是说，虽然在单元测试中必须编写这些辅助模块，但却不作为最终的软件产品提供用户。不过这些模块的结构十分简单，模块间接口的全面检验可在集成测试时去进行。

二、集成测试

在每个模块完成单元测试以后，需要按照设计时作出的结构图，把它们联接起来，进行集成测试(integrated testing)。实践表明，一些模块能够单独地工作，并不能保证连接起来也能正常工作。程序在某些局部反映不出的问题，在全局上很可能暴露出来，影响功能的发挥。

怎样合理地组织集成测试，这里提供两种不同的方法，即非增式测试和增式测试。

非增式测试方法是这样进行的：在配备辅助模块的条件下，对所有模块进行个别的单元测试。然后在此基础上，按程序结构图将各模块联接起来，把联接后的程序当作一个整体进行测试。图 3.7 表示采用这种非增式的集成测试方法的一个例子。被测程序的结

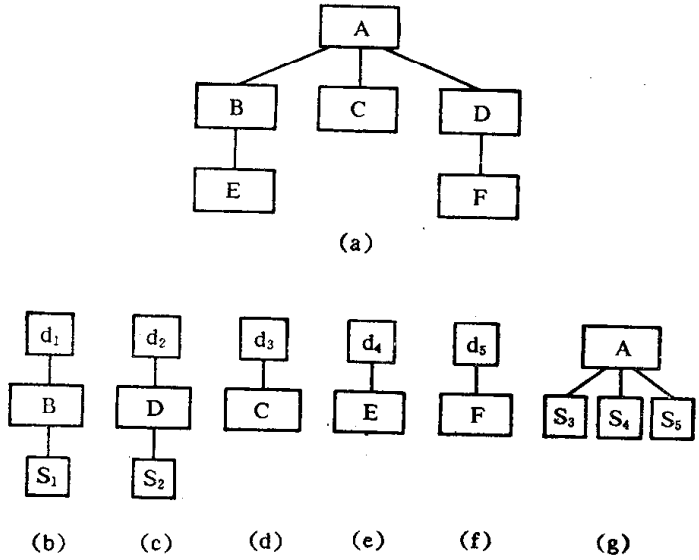


图 3.7　非增式测试例

构由图 3.7(a) 表示，它由六个模块构成。在进行单元测试时，根据它们在结构图中的地位，对模块 B 和 D 配备了驱动模块和桩模块，对模块 C、E 和 F 只配备了驱动模块。对主模块 A，由于它处在结构图的顶端，无其它模块调用它，因此仅为它配备了 3 个桩模块，以模拟被它调用的 3 个模块 B、C和 D。分别进行单元测试以后，再按图 3.7(a)的结构图形式联接起来，进行集成测试。

增式测试的作法与非增式测试有所不同。它的集成是逐步实现的，集成测试也是逐步完成的。也可以说它把单元测试与集成测试结合起来进行。增式集成测试可按不同的

次序实施，因而可以有两种：

① 自顶向下增式测试表示逐步集成和逐步测试是按结构图自上而下进行的。图 3.8 给出的程序，其结构图如图 3.8(c)所示。集成测试分为 3 步：首先，对顶层的主模块 A 进行单元测试，这时需配以桩模块 S_1、S_2 和 S_3（参看图 3.8(a)），以模拟被它调用的模块 B、C 和 D。其后，把模块 B、C 和 D 与顶层模块 A 联接起来，再对 B 和 D 配以桩模块 S_4 和 S_5，以模拟对模块 E 和 F 的调用。这样按图3.8（b）的形式完成测试。最后，去掉桩模块 S_4 和 S_5，把模块 E 和 F 联上即对完整的结构图进行测试。

② 自底向上增式测试表示逐步集成和逐步测试的工作是按结构图自下而上进行的。图 3.9 以同一实例表明了这一过程。图 3.9（a）、（b）和（c）表示：树状结构图中处在

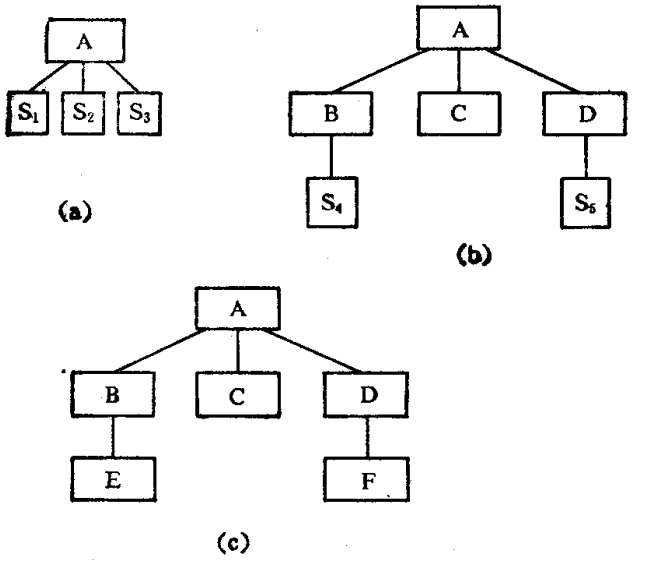


图 3.8 自顶向下增式测试例

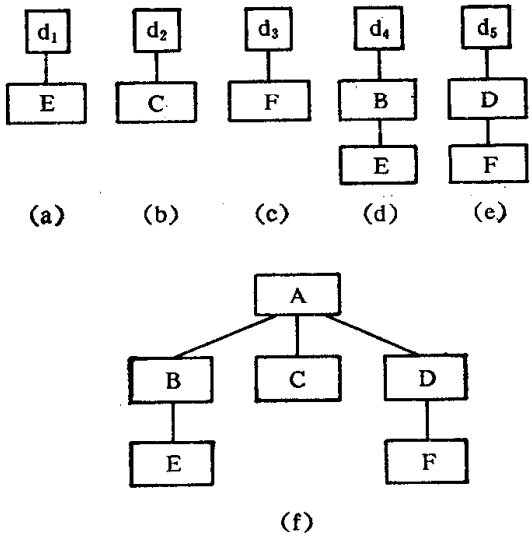


图 3.9 自底向上增式测试例

最下层的叶结点模块 E、C 和 F，由于它们不再调用其它模块，对它们进行单元测试时，只需配以驱动模块 d_1、d_2 和 d_3，用来模拟 B、A 和 D 对它们的调用。完成这 3 个单元测试以后，再按图 3.9 中(d) 和(e)的形式，分别将模块 B 和 E 及模块 D 和 F 联接起来，在配以驱动模块 d_4 和 d_5 的条件下实施部分集成测试。最后再按图 3.9 (f)的形式完成整体的集成测试。

比较以上几种集成测试的作法，我们可以看出：

① 非增式测试的做法是先分散测试，再集中起来一次完成集成测试。如果在模块的接口处存在差错，只会在最后的集成时一下子暴露出来。与此相反，增式测试的逐步集成和逐步测试的办法，把可能出现的差错分散暴露出来，便于找出问题和修改。其次，从前面的实例中也可看出，增式测试使用了较少的辅助模块，也就减少了辅助性测试工作。并且一些模块在逐步集成的测试中，得到了较为频繁的考验，因而可能取得较好的测试效果。总的说来，增式测试比非增式测试具有一定的优越性。

② 自顶向下测试的主要优点在于它可以自然地做到逐步求精，一开始便能让测试者看到系统的雏型。这个系统模型的检验有助于增强程序人员的信心，它的不足是一定要提供桩模块。并且在输入、输出模块接入系统以前，在桩模块中表示测试数据有一定困难。由于桩模块不能模拟数据，如果模块间的数据流不能构成有向的非环状图，一些模块

的测试数据难于生成。同时观察和解释测试输出往往也是困难的。

另一方面，自底向上测试的优点在于，由于驱动模块模拟了所有调用参数，即使数据流并未构成有向的非环状图，生成测试数据也没有困难。如果关键的模块是在结构图的底部，自底向上测试是有优越性的。

自底向上方法的缺点在于，当最后一个模块尚未测试时，还没有呈现出被测软件系统的雏型。由于最后一层模块尚未设计完成时，无法开始测试工作，因而设计与测试工作不能交叉进行。

三、确认测试

集成测试完成以后，分散开发的模块被联接起来，构成完整的程序。其中各模块之间接口存在的种种问题都已消除。于是测试工作进入最后阶段——确认测试（validation testing）。什么是确认测试，说法众多，其中最简明、最严格的解释是检验所开发的软件是否能按顾客提出的要求运行。若能达到这一要求，则认为开发的软件是合格的。因而有的软件开发部门把确认测试称为合格性测试（qualification testing）。这里所说的顾客要求通常指的是在软件规格说明书中确定的软件功能和技术指标，或是专门为测试所规定的确认准则。

① 确认测试准则

怎样来判断被开发的软件是成功的？为了确认它的功能、性能以及限制条件是否达到了要求，应进行怎样的测试？在需求规格说明书中可能作了原则性规定，但在测试阶段需要更详细、更具体地在测试规格说明书（Test specification）中作进一步说明。例如，制定测试计划时，要说明确认测试应测试哪些方面，并给出必要的测试用例。除了考虑功能、性能以外，还需检验其它方面的要求。例如，可移植性、兼容性、可维护性、人机接口以及开发的文件资料等是否符合要求。

经过确认测试，应该为已开发的软件作出结论性评价。这也无非是两种情况之中的一个：（1）经过检验的软件功能、性能及其它要求均已满足需求规格说明书的规定，因而可被接受。认为是合格的软件。（2）经过检验发现与需求说明书有相当的偏离，得到一个各项缺陷清单。对于第二种情况，往往很难在交付期以前把发现的问题纠正过来。这就需要开发部门和顾客进行协商，找出解决的办法。

② 配置审查

配置审查是确认过程的重要环节。其目的在于确保已开发软件的所有文件资料均已编写齐全，并得到分类编目、足以支持投入运行以后的软件维护工作。这些文件资料包括：用户所需资料（如用户手册、操作手册），设计资料（如设计说明书等），源程序以及测试资料（如测试说明书、测试报告等）。配置审查（Configuration review）有时也称配置审计（Configuration audit）。图 3.10 给出了它和确认测试的关系。

有关验收测试的问题我们将在第六章测试管理中作进一步阐述。

四、系统测试

由于软件只是计算机系统中的一个组成部分，软件开发完成以后，最终还要与系统中

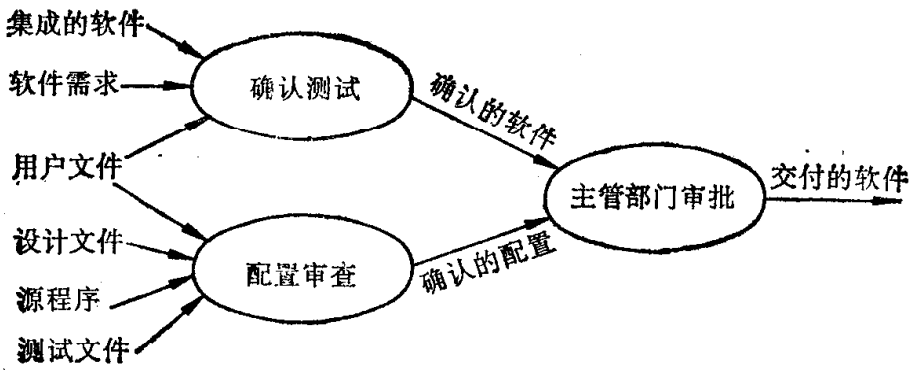


图 3.10　配置审查与确认测试

的其它部分配套运行。系统在投入运行以前各部分需完成组装和确认测试，以保证各组成部分不仅能单独地受到检验，而且在系统各部分协调工作的环境下也能正常工作。这里所说的系统组成部分除去软件外，还可能包括计算机硬件及其相关的外围设备、数据及其收集和传输机构、掌握计算机系统运行的人员及其操作等，甚至还可能包括受计算机控制的执行机构。显然，系统的确认测试已经完全超出了软件工作的范围。然而，软件在系统中毕竟占有相当重要的位置，软件的质量如何，软件的测试工作进行得是否扎实势必与能否顺利、成功地完成系统测试关系极大。另一方面，系统测试实际上是针对系统中各个组成部分进行的综合性检验。尽管每一个检验有着特定的目标，然而所有的检测工作都要验证系统中每个部分均已得到正确的集成，并能完成指定的功能。以下分别简要说明几种系统测试：

① 恢复测试

恢复测试是要采取各种人工干预方式使软件出错，而不能正常工作，进而检验系统的恢复能力。如果系统本身能够自动地进行恢复，则应检验：重新初始化，检验点设置机构、数据恢复以及重新启动是否正确。如果这一恢复需要人为干预，则应考虑平均修复时间是否在限定的范围以内。

② 安全测试

安全测试的目的在于验证安装在系统内的保护机构确实能够对系统进行保护，使之不受各种非常的干扰。系统的安全测试要设置一些测试用例试图突破系统的安全保密措施，检验系统是否有安全保密的漏洞。

③ 强度测试

检验系统的能力最高实际限度。进行强度测试时，让系统的运行处于资源的异常数量、异常频率和异常批量的条件下。例如，如果正常的中断平均频率为每秒一到二次，强度测试设计为每秒 10 次中断。又如某系统正常运行可支持 10 个终端并行工作，强度测试则检验 15 个终端并行工作的情况。

④ 性能测试

性能测试检验安装在系统内的软件运行性能。这种测试常常与强度测试结合起来进行。为记录性能需要在系统中安装必要的量测仪表或是为度量性能而设置的软件（或程序段）。

3.4 人 工 测 试

一、人工测试技术概述

人工测试技术的含意是什么，我们可以从字面上有所了解。它是针对自动测试技术而言的。本书除这一节的内容以外，所有的测试技术都要利用计算机，在这个意义上说那是自动测试技术。然而，严格地说，每一种自动测试技术都离不开人的干预。到目前为止，完全的自动测试过程尚未实现。本节所强调的是不依赖于计算机的测试技术。

其实，在计算机发展的初期，对程序人员来说机时是非常宝贵的。为了节省用于程序查错的机时，当时人们写好程序上机之前总要仔细检查、反复检查可能出现的各种程序错误，这就是通常说的静态检查。

70 年代以来，软件生存期的概念逐步形成。人们认识到，软件开发不仅要看到程序编写阶段的质量问题，更重要的还必须重视开发前期的质量检验问题。必须找出适用的方法检查开发前期各阶段工作的质量，而这些检查的方法一般是无法利用计算机的。另一方面，各阶段软件产品作为一种脑力劳动的成果，如果仅仅停留在开发者自己检验自己产品的方式上，那便会有很大的局限性。常常会出现个人的思路狭窄，不容易发现问题的情况。

1974 年 M.E.Fagan 首先提出了软件审查会 (software inspections) 的想法，接着在 IBM 公司和其它一些软件开发机构中试行，取得了较好的效果。此后逐渐被更多的软件开发部门接受，至今成为在软件开发过程中把握产品质量的有效方法。有人统计，这一方法用于检验程序时，能有效地发现 30% 到 70% 的逻辑设计错误和编码错误。IBM 公司代码审查会的查错效率更高，竟能查出全部错误的 80%。由于人工测试技术在检查某些编码错误时，有着特殊的功效，它常常能够找出利用计算机测试不容易发现的错误，我们必须重视它。绝不能认为，有了充足的计算机机时，就只考虑采用上机测试的办法；或者认为不利用计算机测试就是落后的陈旧测试方法。

事实上，人工测试可用于软件开发的各个阶段，也有着各种不同的形式和不同的名称。审查会 (inspection) 与评审 (review) 并没有本质的差别。用于检查编码的错误，程序审查 (program inspection) 也称代码审查会 (code inspection) 与人工运行 (walkthrough) 也只是实施步骤上有些差别。以上这些都是组成小组进行集体审查的。代码的静态检查 (desk checking) 则是个人完成的。等级评定 (peer ratings) 也是针对程序的，但它并不是为了检查出程序错误，而是由评议小组的成员分别对程序的质量给出具体的评价。

人工运行与审查会的差别主要在于侧重点和目标有所不同。人工运行通常被当作开发人员在开发过程中使用的技术(尽管实际上不只是开发者本人参加)，其目的在于提高编码的质量。审查会则常被当作一种管理工具，经过审查不仅可以提高各阶段软件产品(不仅是编码)的质量，而且还可以收集到一些有关该软件产品质量的数据。项目管理人员可借以合理和定量地作出决策。因此，软件开发过程中每个阶段的审查都必须十分正

规地、严格地加以定义，并且根据规格说明实施。

二、软件审查

审查会是使非开发人员的力量与开发人员结合起来，利用集体的智慧查找软件产品中存在的问题，从而保证软件产品质量的有效手段。在此我们对审查会和评审会不加区别。审查的对象可以是各开发阶段的成果，例如：需求分析、概要设计、详细设计等阶段的成果以及编码、测试计划和测试用例等。

软件审查工作大致要经历以下几个步骤：

- 制定计划；
- 预审；
- 准备；
- 审查会；
- 返工；
- 终审。

这里对每个步骤作出简要介绍。

在软件产品确已具备阶段审查的条件后，可以着手制定审查工作计划。首先要确定审查会主持人。主持人负有组织审查会，并最后决定审查结论的责任。他应该是客观公正的。为能作到这一点，主持人不应是被审查软件的开发人员。他的首要责任是检验该软件的阶段产品是否确已具备了进行审查的条件。若能肯定这一点，则可决定预审，并确定参加审查的其它人员。通常包括开发者，但还需有“局外人”参加，以4至5人组成审查组为宜。接着便需决定审查工作的日程。

预审是正式审查的初步，为了能公正地准确地完成评审，应当把与评审相关的资料提供给参审人员。例如，如果审查的对象是某一分时系统的软件模块，就需要在每个模块的审查开始以前，让所有参审人员对整个系统有个全面的了解。必要时，可举行专题报告，以介绍和讲解该软件设计和实现中所采用的特定技术和方法。比如，若是同步独立处理采用了特定的入队和出队技术，那就要在审查前，作出专题介绍。

软件审查的准备工作主要是单独进行的活动。开发人员收集有关的审查资料，并填写“软件审查概要”表（参看本书附录A中表一）。其它参审者则应认真阅读和研究所提供的资料，填写“软件审查准备工作记录“（附录A中表二）。其目的在于对被审软件有充分的了解，记录那些阅读中发现了的问题。在这些问题中，有些可能是明显的错误，有些是不可理解的部分。这些问题都将在审查会上提出，以求得进一步分析和讨论。

召开审查会自然是软件审查的中心环节。它的主要工作包括：

① 审查会主持人了解会议准备情况，包括各位参审人员在会前准备工作中所花的时间。如果他认为准备工作不够充分，那么很可能决定推迟审查会召开的时间。

② 仔细阅读并记录所发现的不妥之处是审查会的主要活动。在审查会上，由一位审查员逐段朗读被审资料。参审人员，可随时提出问题，这时中断朗读。对于发现了的问题，可以立即解决或记录下来。只要审查组同意，或是主持人认为有必要，便由记录员将所发现问题的位置、简要情况和问题类别等登记在“审查会发现问题报告”（参看本书附录

A中表四)上。朗读结束后,主持人指示记录员检查,看是否确已把所有发现的问题均已作了记录,并正确地作了分类。

③ 此后,参审人员要决定对这次审查的结论。它可以是:符合要求、需要返工或需要再次审查等。只有当被审查资料确已满足事先规定的一些"通过条件"(也称出口条件)时(或是基本满足,但只需进行非常简单的修正时),才会作出"符合要求"的结论。若在审查后所作的返工中有较大的修改,就必须进行"再次审查"。

由于审查会要找出并且记录被审查软件产品存在的问题,开会期间常常会引起人际关系的紧张。有经验的审查会主持人能够注意到这一点,在审查会之后可能安排一个开发人员的研制报告。其目的在于,使得参审人员从中吸取有益的开发经验。对于好的经验,自然会得到热情的赞扬,从而使紧张的空气得到调节。另一方面,管理人员无需参加审查,因为如果把发现的具体技术问题和开发人员的责任分开,只会促进问题的发现和解决,而不会形成人为的紧张并影响正常的工作。

审查会虽严格但并不可怕。所有参审人员只要明确了审查的目的在于找出软件产品的问题,开发者和审查者的目标是一致的。并且认识到审查的对象是软件产品而不是开发者,则都能认识到这样做是完全必要的。

审查返工自然是在审查会以后由开发者完成的。这项工作通常只是把记录在"审查会发现问题报告"的错误改正过来。

终审是由主持人完成的最后审查活动。在返工完成以后,他要检验所有需要改正的地方是否确已改正。并且最后填写"审查结果报告"和"审查总结报告"(附录A中的表三和表五)。

三、软件审查的作用

1. 软件审查所得数据的使用

任何一个软件审查都将取得一些数据,这些数据如图 3.11 所示,它真实地反映了开发过程中出现的各种问题。充分利用这些数据指导和改进开发工作,将是十分有益的。图 3.12 表示了处在两个开发阶段之间的审查,其所获数据有三方面用途,即反馈、前馈和馈入。利用审查所得数据来改进前期的开发过程,其反馈作用是很容易理解的。例如,设计审查发现,数据定义的问题特别突出。这就要求更好地控制数据打印格式的文档,即需要作些调整,以保证随后的产品有较高的质量。前馈可以对软件产品的调整提供有用的信息。例如,发现某一软件产品中逻辑问题所占的比例很大,那就要进行更为严格的测试,并重点检验其中的逻辑关系。审查数据的第三个用途是馈入,它将为审查工作自身提供有效性数据和质量数据,可以从中了解到这次审查做得怎样。例如,软件产品在审查中是否得到了严格的检验,参审人员在审查会前是否充分地做好了准备工作等等。根据审查数据的收集和分析,再加上其它形式的开发阶段度量信息(如测试后得到的有关质量和生产率的报告)可以对特定的审查作些调整,以求得更好的有效性。这一点如果用得恰当,软件审查便具有自调节功能。事实已经反复表明,软件审查在提高软件质量和生产率方面都不失为一个十分有效的方法。

2. 作为软件开发进程控制的审查

系统名: _____ 审查日期: _____

被审查单元名: _____

审查会主持人: _____

参加审查人员: _____

审查类别: _____

本阶段审查发现的问题统计				
问题类型	遗漏	错误	多余	共计
接口				
数据				
逻辑				
输入/输出				
性能				
人为因素				
标准				
文档				
语法				
功能				
测试环境				
测试覆盖				
其它				
			总计	

开发中前阶段发现的问题			
所属阶段	问题类型	问题性质	修改次数

估计返工完成日期: _____

返工工作量估计: _____

实际返工工作量: _____

图 3.11 软件审查取得的数据

为了使软件审查成为软件开发进程的控制工具，就要在开发过程的若干控制点实施

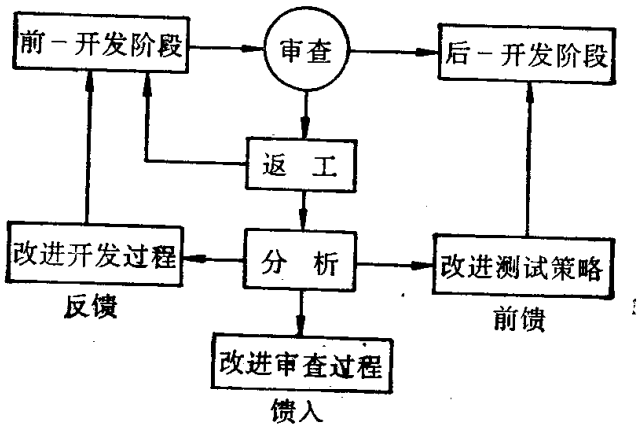


图 3.12 审查数据的用途

前面介绍过的阶段审查。图 3.13 给出了安排有阶段审查的软件开发过程。图中涉及到的开发阶段包括：需求分析、概要设计、详细设计、编码、制定测试计划以及设计测试用例。其中的每个阶段审查都应规定它的进入条件、出口条件、推荐的参审人员、查出问题

分类以及查找问题策略。

所谓阶段审查的进入条件是指准备好审查时必须具备的条件。如果并未安排对某个开发阶段进行审查，那个开发阶段的出口条件，即是这个阶段工作完成时所应具备的条件。除此以外，进入条件还包括一些作好审查工作的需求。例如，编码审查的进入条件要求代码成功地通过编译，被审查的代码要有清晰可见的行号。同时，与代码相关的需求、设计及信息修改的资料都应备齐，作为审查材料的组成部分。

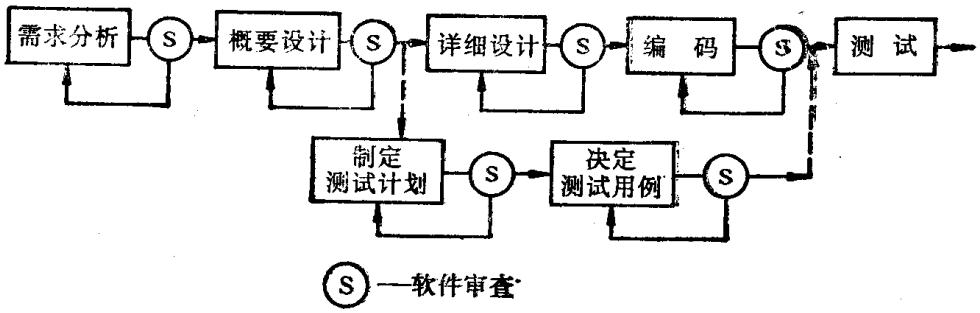


图 3.13 软件开发过程中的审查

出口条件是审查工作完成的条件。审查的出口条件主要指的是确已改正了所有发现的问题。

由于软件审查是集体进行的活动，审查小组内部协作得如何至关重要，因而，选定参审人员常常是搞好审查的关键。在软件开发初期的阶段审查中（例如需求分析审查或概要设计审查），用户和系统测试的观点，以及实现人员的观点都应得到反映。为能做到这一点，审查组成员有时多达 7 至 8 人。而开发的后期阶段，如编码阶段的审查则只涉及到详细设计、编码和单元测试。由 3 至 4 位开发人员参加就可以了。

——采用什么办法查找存在的问题，与所作审查属于哪个开发阶段、被审查资料的质量以及对查出问题的分类有关。有些审查活动围绕着按问题类型编排的检查表（checklists）进行。有些审查则强调在审查的准备工作中主要是理解送审资料。这时如果使用按问题类型编排的检查表就很不合适。可以给出一份按送审资料编排的“审查准备工作指南”，例如，在作概要设计审查时，验证所设计的功能是否与规定的需求一一对应是很重要的。那么概要设计审查的准备工作指南就应包括询问子程序的设计和模块的设计是否反映这种对应关系。

正如图 3.11 所给出数据，阶段审查所发现的问题主要有三种表现形式，即

遗漏——在规格说明或标准中指明应该有的内容，送审资料中丢掉了。

多余——超出规格说明和标准，多给出的信息。

错误——应该有，也的确有，但内容有误的信息。

图 3.11 对审查中发现的问题归纳成 13 类，即接口、数据、逻辑、输入和输出、功能、性能、人为因素、标准、文档、语法、测试环境、测试覆盖和其它。这里结合某一项目概要设计的审查，举例说明其分类方法。图 3.14 给出了这一分类的实例。其项目概要设计说明则在图 3.15 中给出。

性能

·有一似乎合理的变元使该程序不能满足需求中描述的性能目标

数据

·在 INPUTS,OUTPUTS 或 UPDATES 节中遗漏或多出信息项

·在 INPUTS,OUTPUTS 或 UPDATES 节中遗漏或有不正确的数据类型

接口

·在 PROCESSING 节中遗漏或多出子程序调用

·INVOCATION 中的信息有误

功能

·在 PROCESSING 中遗漏、多余或有错误操作

·在 PROCESSING 中的逻辑条件有遗漏、多余或有误

文档

·在 DESCRIPTION 节中的内容不完整或使人误解

·对数据项的描述有多义性

·处理步骤描述不够清晰

标准

·违背了采用的设计标准: Project Standards Manual

语法

·语法、标点或拼写方面有误

其它

·不属于以上所述的其它问题

图 3.14 概要设计审查发现问题分类实例

```
SUBPROGRAM:prcl
TITLE: 按列打印
DESCRIPTION: 本子程序将输入文件变换格式成按列输出文件
INPUTS:
    -l      任选的长度命令
    -k      任选的列数
    -w      任选的页宽命令
    filename 输入文件
UPDATES:N/A
OUTPUTS:
    格式化输出    排入标准输出文件
    错误信息      至标准错误文件
    状态          处理状态码
PROCESSING:
    1.将输入文件格式化按标准输出每页 k 列
    2.若发现错误,则将错误信息送往标准错误文件
    3.若从正常出口则回送状态码 0,若经错误出口则回送状态码 1
INVOCATION: 使用以下格式调用
    Prcl[-l  integer] [-k integer] [-w integer] filename
```

图 3.15 概要设计说明书例

参 考 文 献

[1] A.Frank Ackerman et al., "Software Inspections and the Industral Production of Software", Software Validation, Edited by H.L.Hausen, Elsevier Science Publishers B.V. (North-Holland), GMD, 1984.

[2] Fagan, M.E., "Design and Code Inspections to Reduce Errors in Program Development", IBM System Journal, Vol.15, No.3, 1976.

[3] Yourdon, E., Structured Walkthroughs, Yourdon, Inc., 1978.

[4] Edited by Stephen J. Andriole. Software Validation, Verification, Testing and Documentation, Petrocelli Books, 1986.

[5] Robert H.Dunn, Software Defect Removel, McGraw-Hill Book company,1984.

[6] Richard A.DeMillo et al., Software Testing and Evaluation, Benjamin/Cummings Company, Inc., 1987.

[7] R.E. Fairley, "Tutorial:Static Analysis and Dynamic Testing of Computer Software",Computer, Vol. 11(4), Apr.1978.

[8] Cem Kaner, Testing Computer Software, TAB Professional and Reference Books, 1988

[9] Myers, G., The Art of Software Testing, John Wiley & Sons, 1979.

[10] William E.Howden, Functional Program Testing and Analysis, McGraw-Hill Book Company, 1987.

第四章　黑盒测试

关于黑盒测试的基本概念在前一章中已和白盒测试方法相对照作了说明。本章将在此基础上简要介绍几个具体的黑盒测试方法，其中包括等价类划分、因果图、正交实验设计法、边值分析、判定表驱动法等，最后介绍功能测试。应该说，这些方法是比较实用的。掌握和采用这些方法并不困难，它们能够较好地解决我们软件开发中的测试问题。使用时自然要针对开发项目的特点对方法加以适当的选择。

4.1　等价类划分

等价类划分是一种典型的黑盒测试方法，用这一方法设计测试用例完全不考虑程序的内部结构，而是只根据对程序的要求和说明，即平时我们常常说的需求规格说明书(Requirement Specifications)。我们必须仔细分析和推敲说明书的各项需求，特别是功能需求。把说明中对输入的要求和输出的要求区别开来并加以分解。

既然穷举测试的办法由于数量太大，以致于无法实际完成，自然促使我们在大量的可能数据中选取其中的一部分作为测试用例。问题在于如何选取。等价类划分的办法是把程序的输入域划分成若干部分，然后从每个部分中选取少数代表性数据当作测试用例。使用这一方法设计测试用例，首先必须在分析需求规格说明的基础上划分等价类，列出等价类表。以下按其步骤对方法加以说明，并给出两个实例。

一、方法简介

1. 划分等价类

首先把数目极多的输入情况划分成若干个等价类。所谓等价类是指某个输入域的集合。它表示，如果用集合中的一个输入条件作为测试数据进行测试不能发现程序中的错误，那么使用集合中的其它输入条件进行测试也不可能发现错误。也就是说，对揭露程序中的错误来说，集合中的每个输入条件是等效的。如果我们的测试数据全都从同一个等价类中选取，除去其中的一个测试数据对发现程序错误有意义以外，使用其余的测试数据进行测试都是徒劳的，因为它们对测试工作的进展没有任何益处。我们不如把测试的时间花在其它等价类输入条件的测试中。例如，如果以 2 和 3 分别作为输入某一程序的两个测试数据，它们具有等价的测试效果(即如果 2 作为测试数据，在程序测试中能暴露某个错误，3 若作为测试数据也能发现同一个错误)。我们宁愿只取它们中的一个作为测试数据，作一次测试，而不是取两个，分别作两次测试。

在考虑等价类时，应该注意区别两种不同的情况：

• 有效等价类：有效等价类指的是对程序的规格说明是有意义的、合理的输入数据

所构成的集合。在具体问题中，有效等价类可以是一个，也可以是多个。

·无效等价类：无效等价类指对程序的规格说明是不合理的或无意义的输入数据所构成的集合。对于具体的问题，无效等价类至少应有一个，也可能有多个。

如何确定等价类，这是使用等价类划分方法的一个重要问题。以下结合具体实例给出几条确定等价类的原则：

① 如果输入条件规定了取值范围或值的个数，则可确定一个有效等价类和两个无效等价类。例如，程序的规格说明中提到的输入条件包括"…项数可以从 1 到 999…，"则可取有效等价类为"1＜项数＜999"。无效等价类为 "项数＜1"及"项数＞999"。又如，程序规格说明中提到"…学生允许选修 2 至 4 门课…"，有效等价类可取"选课 2 至 4 门"，无效等价类为"只选一门或未选课"及"选课超过 4 门"。

② 输入条件规定了输入值的集合，或是规定了"必须如何"的条件，则可确定一个有效等价类和一个无效等价类。例如，某程序的规格说明中提到的输入条件包括"…统计全国各省、市、自治区的人口…"，则应取"国内省、市、自治区"为有效等价类，"非国内省、市、自治区"为无效等价类。又如，某程序涉及到标识符，其输入条件规定"标识符应以字母开头…"，则"以字母开头者"作为有效等价类，"以非字母开头"为无效等价类。

③ 如果我们确知，已划分的等价类中各元素在程序中的处理方式是不同的，则应将此等价类进一步划分成更小的等价类。

等价类划分完以后，可按下面的形式列出等价类表：

输入条件	有效等价类	无效等价类
…	…	…
…	…	…

2. 确定测试用例

根据已列出的等价类表，按以下步骤确定测试用例：

① 为每个等价类规定一个唯一的编号。

② 设计一个测试用例，使其尽可能多地覆盖尚未覆盖的有效等价类。 重复这一步，最后使得所有有效等价类均被测试用例所覆盖。

③ 设计一个新的测试用例，使其只覆盖一个无效等价类。重复这一步使所有无效等价类均被覆盖。

注意，这里规定每次只覆盖一个无效等价类，是因为一个测试用例中如果含有多个错误，有可能在测试中只发现其中的一个，另一些被忽视。例如，程序的规格说明中规定了"…每类科技用书 10 至 50 册，…"，若一个测试用例为"小说 5 册"，在测试中很可能只检测出书的类型错误，而忽略了书的册数错误。

二、以下给出两个应用等价类划分方法进行测试用例设计的实例

① 某程序规定："输入三个整数作为三边的边长构成三角形。当此三角形为一般三角形、等腰三角形及等边三角形时，分别做计算…"。试用等价类划分方法为该程序的构成

表 4.1　例 1 的等价类表

		有效等价类	号码	无效等价类	号码
输入条件	输入三个整数	整数	1	一边为非整数：a 为非整数	12
				一边为非整数：b 为非整数	13
				一边为非整数：c 为非整数	14
				两边为非整数：a,b 为非整数	15
				两边为非整数：b,c 为非整数	16
				两边为非整数：a,c 为非整数	17
				三边 a,b,c 均为非整数	18
		三个数	2	只给一边：只给 a	19
				只给一边：只给 b	20
				只给一边：只给 c	21
				只给两边：只给 a,b	22
				只给两边：只给 b,c	23
				只给两边：只给 a,c	24
				给出三个以上	25
		非零数	3	一边为零：a 为 0	26
				一边为零：b 为 0	27
				一边为零：c 为 0	28
				二为边零：a,b 为 0	29
				二为边零：b,c 为 0	30
				二为边零：a,c 为 0	31
				三边 a,b,c 均为 0	32
		正数	4	一边<0：a<0	33
				一边<0：b<0	34
				一边<0：c<0	35
				二边<0：a<0 且 b<0	36
				二边<0：a<0 且 c<0	37
				二边<0：b<0 且 c<0	38
				三边均<0：a<0 且 b<0 且 c<0	39
输出条件	构成一般三角形	a + b>c	5	a + b<c	40
				a + b = c	41
		b + c>a	6	b + c<a	42
				b + c = a	43
		a + c>b	7	a + c<b	44
				a + c = b	45
	构成等腰三角形	a = b 且两边之和大于第三边	8		
		b = c 且两边之和大于第三边	9		
		a = c 且两边之和大于第三边	10		
	构成等边三角形	a = b = c	11		

三角形部分进行测试用例设计。

使用等价类划分方法必须仔细分析和推敲题目所给出的要求。本题的输入条件要求的关键之处有:

1) 整数;

2) 三个数;

3) 非零数;

4) 正数。

输出条件要求的关键之处有:

5) 应满足两边长之和大于第三边边长;

6) 等腰;

7) 等边。

其中,3)、4)和 5) 并没有在题目上明显给出,但这些条件是必要的。

以下分两步进行:

(1) 列出等价类表(见表 4.1, 表中号码为等价类编号)

(2) 列出覆盖上述等价类的测试用例

覆盖有效等价类的测试用例:

a	b	c	覆盖等价类号码
3	4	5	①——⑦
4	4	5	①——⑦,⑧
4	5	5	①——⑦,⑨
5	4	5	①——⑦,⑩
4	4	4	①——⑦,⑪

覆盖无效等价类的测试用例见表 4.2。

② 某 FORTRAN 编译系统的设计与程序编写已经完成,现需对其中 DIMENSION 语句的编译设计测试用例。已知 DIMENSION 语句的语法规则是:

DIMENSION 语句用以规定数组的维数,其形式为:

DIMENSION ad [,ad]···

其中, ad 是数组描述符,形式为:

n(d[,d]···)

其中, n 是数组名,由 1 个至 6 个字母或数字组成,为首的必须是字母;

d 是维数说明符。数组维数最大为 7,最小为 1。它的形式为:

[lb:]ub

lb 和 ub 分别表示数组下标的下界和上界,均为−65,534 至 65,535 之间的整数,也可是整型变量名(但不可是数组元素名)。若未规定 lb, 则认为其值为 1, 且 ub ≥ lb。若已规定 lb, 则它可为负数、零或正数。

DIMENSION 语句也和其它语句一样,可连续写多行。

表 4.2 覆盖无效等价类的测试用例

a	b	c	覆盖等价类号码	a	b	c	覆盖等价类号码
2.5	4	5	12	0	0	5	29
3	4.5	5	13	3	0	0	30
3	4	5.5	14	0	4	0	31
				0	0	0	32
3.5	4.5	5	15				
3	4.5	5.5	16	−3	4	5	33
3.5	4	5.5	17	3	−4	5	34
3.5	4.5	5.5	18	3	4	−5	35
3			19	−3	−4	5	36
	4		20	−3	4	−5	37
		5	21	3	−4	−5	38
3	4		22	−3	−4	−5	39
	4	5	23				
3		5	24	3	1	5	40
3	4	5	25	3	2	5	41
				3	1	1	42
0	4	5	26	3	2	1	43
3	0	5	27				
3	4	0	28	1	4	2	44
				3	4	1	45

表 4.3 例 2 的等价类表

输入条件	有效等价类	无效等价类
数组描述符个数	1(1),>1(2)	无数组描述符(3)
数组名字符个数	1～6(4)	0(5),>6(6)
数组名	有字母(7),有数字(8)	有其它字符(9)
数组名以字母开头	是(10)	否(11)
数组维数	1～7(12)	0(13),>7(14)
上界是	常数(15),整型变量(16)	数组元素名(17),其它(18)
整型变量名	有字母(19),有数字(20)	其它(21)
整型变量名以字母开头	是(22)	否(23)
上下界取值	−65,534～65,535(24)	<−65,534(25)
		>65,535(26)
是否定义下界	是(27),否(28)	
上界对下界关系	>(29),=(30)	<(31)
下界定义为	负数(32),0(33),正数(34)	
下界是	常数(35),整型变量(36)	数组元素名(37),其它(38)
语句多于一行	是(39),否(40)	

以上规则中，小写字母代表语法单位。方括号内是任选项；省略号表示它前面的项可以重复出现多次。

按下面两步用等价类划分方法设计测试用例。

第一步　确定输入条件，并确定等价类。

等价类表在表 4.3 中给出(表中括号内数字为等价类号)：

第二步　确定测试用例。先设计一个测试用例，使它覆盖一个或多个有效等价类。如：

```
DIMENSION A(2)
```

能覆盖有效等价类 1,4,7,10,12,15,24,28,29 和 40。为覆盖其它有效等价类，需设计另外的测试用例。如：

```
DIMENSION A12345(I,9,J4XXXX,65535,1,KLM,100),
          BBB(-65534:100,0:1000,10:10,I:65535)
```

它能覆盖其余的有效等价类。

以下一个个地设计测试用例，使每个测试用例只覆盖一个无效等价类，直至覆盖完为止。这些测试用例是(以下测试用例左端括号内的数字为它所代表的等价类号)：

```
(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(I.,10)
(23) DIMENSION C(10,1J)
(25) DIMENSION D(-65535:1)
(26) DIMENSION D(65536)
(31) DIMENSION D(4:3)
(37) DIMENSION D(A(2):4)
(38) DIMENSION D(.:4)
```

以上共计 18 个测试用例，它们覆盖了全部等价类。

从这个实例中可以看到，使用等价类方法能够全面、系统地考虑黑盒测试的测试用例设计问题。它是一个很容易掌握的有效方法。

4.2 因 果 图

前节介绍的等价类划分方法并没有考虑到输入情况的各种组合，也没考虑到各个输入情况之间的相互制约关系。这样做尽管考虑到各个输入条件可能出错的多种情况，但多个输入条件组合起来出错的情况却被忽略了。采用因果图方法 (Cause-Effect Grap-

hing）能够帮助我们按一定步骤，高效率地选择测试用例，同时还能为我们指出，程序规格说明描述中存在着什么问题。

利用因果图导出测试用例需要经过以下几个步骤：

① 分析程序规格说明的描述中，哪些是原因，哪些是结果。原因常常是输入条件或是输入条件的等价类。而结果是输出条件。

② 分析程序规格说明的描述中语义的内容，并将其表示成连接各个原因与各个结果的"因果图"。

③ 由于语法或环境的限制，有些原因和结果的组合情况是不可能出现的。为表明这些特定的情况，在因果图上使用若干个特殊的符号标明约束条件。

④ 把因果图转换成判定表（有关判定表的知识请读者参看本章 4.5节）。

⑤ 把判定表中每一列表示的情况写成测试用例。

这里首先介绍因果图。在因果图中出现的 4 种符号分别表示了规格说明中的 4 种因果关系。图 4.1 给出了这些符号所代表的关系。图中 c_i 表示原因，通常置于图的左部；e_i 表示结果，通常在图的右部。c_i 和 e_i 均可取值 0 或 1，0 表示某状态不出现，1 表示某状态出现。

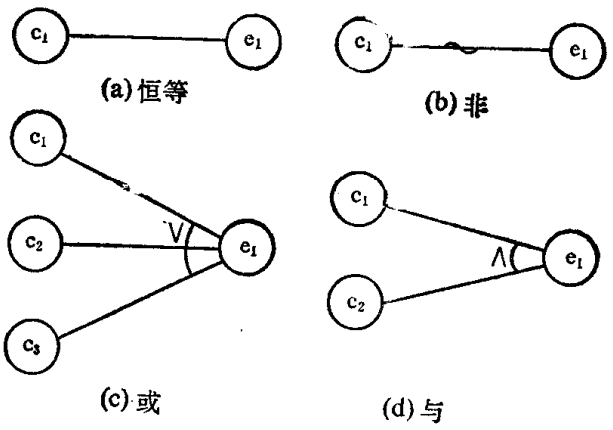


图 4.1 因果图的基本符号

① 恒等：若 c_1 是 1，则 e_1 也是 1；否则 e_1 为 0。

② 非：若 c_1 是 1，则 e_1 是 0；否则 e_1 是 1。

③ 或：若 c_1 或 c_2 或 c_3 是 1，则 e_1 是 1；否则 e_1 为 0。"或"可有任意个输入。

④ 与：若 c_1 和 c_2 都是 1，则 e_1 为 1；否则 e_1 为 0。"与"也可有任意个输入。

因果图中使用了简单的逻辑符号，以直线联接左右结点。左结点表示输入状态（或称原因），右结点表示输出状态（或称结果）。

我们注意到，在实际问题中，输入状态相互之间还可能存在某些依赖关系，我们称之为"约束。"比如，某些输入条件本身不可能同时出现。输出状态之间也往往存在约束。在因果图中，用特定的符号标明这些约束（见图 4.2）。

对于输入条件的约束有以下 4 类：

①E 约束（异）：a 和 b 中至多有一个可能为 1，即 a 和 b 不能同时为 1。

②I 约束（或）：a、b 和 c 中至少有一个必须是 1，即 a、b 和 c 不能同时为 0。

③O 约束（唯一）：a 和 b 必须有一个，且仅有 1 个为 1。

④R 约束（要求）：a 是 1 时，b 必须是 1，即不可能 a 是 1 时 b 是 0。

输出条件的约束只有：

M 约束（强制）：若结果 a 是 1，则结果 b 强制为 0。

以下给出一个简单的实例，借以说明因果图方法的步骤。

某个软件的规格说明中包含这样的要求：

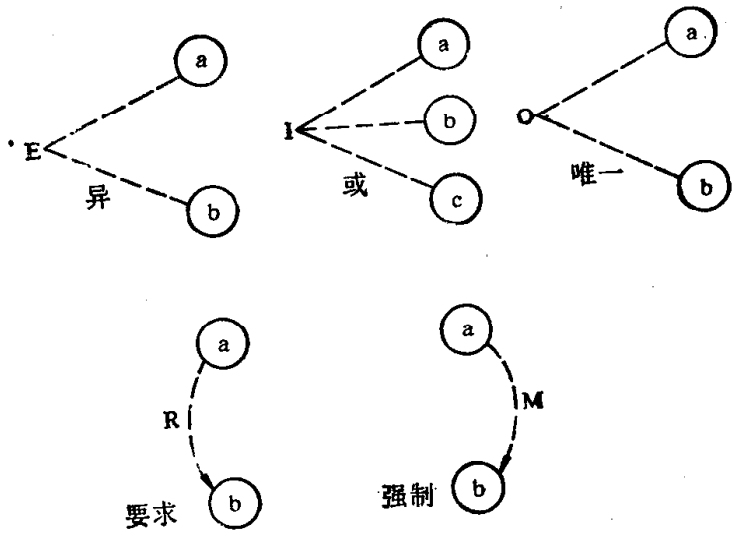


图 4.2　约束符号

“第一列字符必须是A或 B，第二列字符必须是一个数字，在此情况下进行文件的修改。但如果第一列字符不正确，则给出信息 L；如果第二列字符不是数字，则给出信息 M。”

首先根据这一规格说明画出因果图。在分析以上的要求以后，我们可以明确地把原因和结果分开。以下分别给出(左端数字是编号)，

原因：

1——第一列字符是 A；

2——第一列字符是 B；

3——第二列字符是一数字。

结果：

21——修改文件；

22——给出信息 L；

23——给出信息 M。

我们把原因和结果用上述的逻辑符号联接起来，画成因果图(图 4.3)。图中左列表示原因，右列为结果，编号为 11 的中间结点是导出结果的进一步原因。

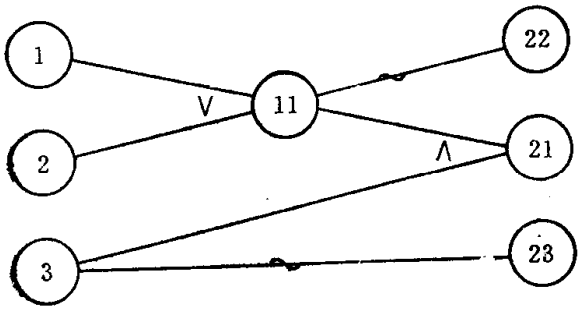


图 4.3　因果图实例

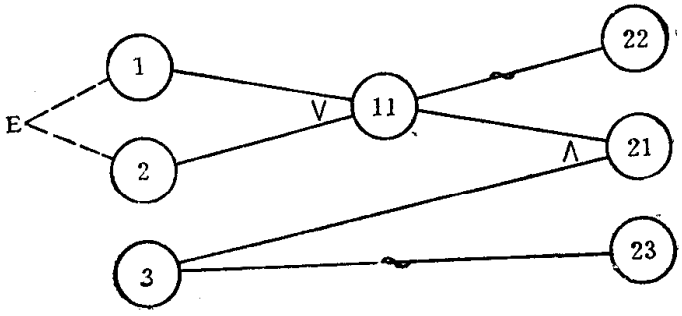


图 4.4　具有 E 约束的因果图

考虑到原因 1 和原因 2 不可能同时为 1 (即第一列字符不可能是A和 B)，我们在因果图上可对其施加 E 约束，这样便得到了具有约束的因果图(参看图 4.4)。

根据因果图建立如下的判定表：

		1	2	3	4	5	6	7	8
条件(原因)	①	1	1	1	1	0	0	0	0
	②	1	1	0	0	1	1	0	0
	③	1	0	1	0	1	0	1	0
	⑪			1	1	1	1	0	0
动作(结果)	㉒			0	0	0	0	1	1
	㉑			1	0	1	0	0	0
	㉓			0	1	0	1	0	1
测试用例				A3 A8	AM A?	B5 B4	BN B!	C2 X6	DY P;

注意，表中 8 种情况的左面两列情况中，原因①和原因②同时为1，这是不可能出现的，故应排除这两种情况。

表的最下一栏给出了 6 种情况的测试用例，这是我们所需要的数据。

以上所述只是因果图应用的一个简单实例，限于篇幅，不可能在此举出更复杂的例子。但绝不要因此而误解，以为没有必要经过上面给出的步骤，以为因果图和判定表都是多余的。事实上，在较为复杂的问题中，这个方法常常是十分有效的，它能有力地帮助我们确定测试用例。当然，如果哪个开发项目在设计阶段就采用了判定表，也就不必再画因果图，而是可以直接利用判定表设计测试用例了。

4.3 正交实验设计法

利用因果图来设计测试用例时，作为输入条件的原因与输出结果之间的因果关系，有时很难从软件需求规格说明书得出，而且即使是对于一般中小规模的软件，画出的因果图也可能非常庞大，以致于据此因果图而得到的测试用例数目将达到惊人的程度，给软件测试工作带来在人工、机时、费用上的沉重负担。

为了有效地、合理地减少测试的工时与费用，可以利用在实际生产活动中行之有效的正交实验设计法，进行测试用例的设计。

所谓正交实验设计法，是从大量的实验点中挑选出适量的、有代表性的点，应用依据伽罗瓦理论导出的“正交表，”合理地安排实验的一种科学的实验设计方法。利用这种方法，可使所有的因子和水平在实验中均匀地分配与搭配，均匀地有规律地变化。

在正交实验设计方法中，通常把判断实验结果优劣的标准叫做实验的指标，把有可能影响实验指标的条件称为因子，而影响实验因子的，叫做因子的水平(或状态)。在进行实验优化设计时，为了完成明确的实验目的，必须有合理的实验指标，加上合理的基准来挑选实验因子以及相应的水平。

软件功能测试，作为实验的一种，完全可以利用正交实验设计法，来进行测试数据的选择，以提高测试的效率。

软件功能测试的目的是检查被测软件是否满足其规格说明书中规定的功能需求。因此，利用正交实验设计法来设计测试用例时，首先要根据被测软件的规格说明书找出影响其功能实现的操作对象和外部因素，把它们当作因子，而把各个因子的取值当作状态，构造出二元的因素分析表。然后，利用正交表进行各因子的状态的组合，构造有效的测试输入数据集，并由此建立因果图。这样得出的测试用例集中，测试用例的数目将大大减少。

一、提取功能说明，构造因子-状态表

软件的功能测试是基于被测软件的规格说明书来进行的测试，因此，用户必须保证提供完整的、准确的被测软件各种功能的详细说明。否则，将无法进行测试结果的分析、比较，更无法得到准确的测试结论。

然而在实际测试时，用户所提供的被测软件的功能说明，往往是非形式化的，很难满足构造因素分析表的需要。因此需要对软件规格说明书中的功能要求进行划分，把整体的概要性的功能要求进行层层分解与展开。分解成具体的，有相对独立性的基本的功能要求，这样就可以把被测软件中所有的因子都确定下来，并为确定各因子的权值提供参考的依据。

接下来，由用户会同测试人员根据软件规格说明书，确定各个因子的取值，即因子的状态。由于有些因子的取值范围较广，我们必须进行采样取值，在各个不同的取值区间上取典型值与边界值，并重点选取某些具有特定意义的取值点。

确定因子与状态是设计测试用例的关键，因此需要尽可能全面、准确地确定取值，以确保测试用例的设计做到完整与有效。

因子与状态填入用二维表格形式表示的因子-状态表。

二、加权筛选，生成因素分析表

对于大中规模的软件，它们所包含的因子个数较多，其取值范围也较广，而且各个因子与状态在被测软件中所起的作用也大不一样，因此必须对众多的因子及其状态加以选择，否则若把所有的因子与状态都收入因素分析表中，则最后生成的测试用例集可能会相当庞大，从而降低测试效率。

对因子与状态的选择可按其重要程度分别加权。具体来说，可根据各个因子及状态的作用大小，出现频率的大小以及测试的需要，确定权值的大小。这样在必要时，可删去一部分权值较小，也就是重要性较小的因子或状态，使最后生成的测试用例集缩减到允许范围。

通常软件系统中各因子与状态在不同的子系统或模块中重要程度不同，因而所起的

作用也不同。这样在进行子系统或模块的功能测试时，可以加大在该模块中起决定作用的因子或状态的权值，便于较准确地测定出出错位置，降低每次测试的复杂程度。

在不同测试阶段对因子及状态的权值的输入、改变都会有不同的要求。例如，初次测试时，可以根据被测软件规模大小，确定是否要进行加权。若需要加权，则由测试人员确定权值并输入之，若无需加权，则应以 0 作为各因子及状态的权值的缺省值。在进行模块或子系统测试时，有时需要修改部分因子或状态的权值，有时必须修改所有因子及状态的权值，有时可以借用上次测试的权值重复进行测试。因此，必须考虑各个测试阶段可能出现的各种情况，分别进行处理(见图 4.5)。

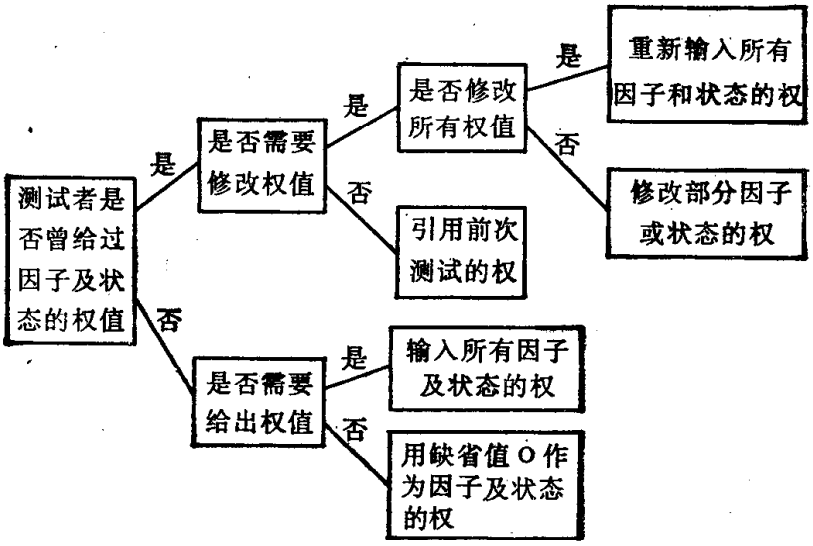


图 4.5　权值处理流程

在确定了各因子及状态的权值之后，便开始进行权值计算。可选用如下方法进行加权筛选。首先将各个因子的权值（ω_i）相加，得到权总和（SUM），即 $SUM = \sum_{i=1}^{n} \omega_i$，n 是因子个数。然后将各个因子的权分别除以权总和，得到各因子的权在权总和中所占的比例，即权比例 r_i。

$$r_i = \frac{\omega_i}{SUM} \qquad (i = 1,2,\cdots,n)$$

接着，选择筛选标准，可采用因子数倒数的一半，即 $\frac{1}{2n}$，作为权比例标准值。将各因子的权比例 r_i 与标准值 $\frac{1}{2n}$ 进行比较。若 $r_i > \frac{1}{2n}$，则保留该因子；若 $r_i \leqslant \frac{1}{2n}$，则舍去该因子及其状态。继而对于保留下来的各个因子也可用同样方法对其状态进行筛选。

最后将保留下来的因子与状态以二维表格形式表示，可得设计测试用例所需的因素分析表。

下面将举例说明。如某被测工程数据库查询语言软件由规格说明书得到的因子-状态表如图 4.6 所示。

因子 / 状态	A	B	C	D
	查询类别	查询方式	元胞类别	打印方式
1	功能	简单	门	终端显示
2	结构	组合	功能块	图形显示
3	逻辑符号	条件		行式打印

图 4.6　因子-状态表

其中 A、B、C、D 分别是各个因子的代码。其状态编号为 1、2、3。

经过加权筛选后，得到因素分析表如图 4.7。

因子 / 状态	A	B	C
	查询类别	查询方式	元胞类别
1	功能	简单	门
2	结构	组合	功能块
3		条件	

图 4.7　因素分析表

可以看出，在因子-状态表中，有 4 个因子。经过加权筛选后，保留下来的因子有“查询类别”、“查询方式”、“元胞类别”，而因子“打印方式”因作用不大被舍去了。在因子“查询类别”中的状态“逻辑符号”也被舍去了。

三、利用正交表构造测试数据集

正交表的推导过程依据近世代数中的伽罗瓦（Galois）理论，在此不予介绍。我们可以从一般的数理统计书中查到如图 4.8 的正交表。

表中，每列表示一个因子，每行表示一个项目。

根据正交表的推导过程，可得正交表的生成规律如下：

① 项目数完全是由因子个数决定的，而且总是 2 的整数次幂。

② 设因子个数为 m：

则当 $m = 2^{i-1} - 1$ 时，项目数为 2^{i-1}；（$i \geqslant 2$）

当 $2^{i-1} - 1 < m \leqslant 2^i - 1$ 时，项目数为 2^i；

当 $2^i - 1 < m \leqslant 2^{i+1} - 1$ 时，项目数为 2^{i+1}。

③ 正交表的生成也是有规律的。大小不一的正交表都可以从一个二行一列的子表逐步推导、构造出来。

设已有某一个子表 $A_{(i-1)}$，它满足①、②。则由它可推导出表 A_i：

因子 / 项目	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
T_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
T_2	1	0	1	0	0	1	1	0	1	0	1	0	1	0	1	
T_3	0	1	1	0	1	0	1	0	0	1	1	0	0	1	1	
T_4	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	
T_5	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
T_6	1	0	1	1	1	0	0	0	1	0	1	1	0	1	0	
T_7	0	1	1	1	0	1	0	0	0	1	1	1	1	0	0	
T_8	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1	
T_9	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
T_{10}	1	0	1	0	0	1	1	1	0	1	0	1	0	1	0	
T_{11}	0	1	1	0	1	0	1	1	1	0	0	1	1	0	0	
T_{12}	1	1	0	0	1	1	0	1	0	0	1	1	0	0	1	
T_{13}	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	
T_{14}	1	0	1	1	1	0	0	1	0	1	0	0	1	0	1	
T_{15}	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	
T_{16}	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0	
⋮																

图 4.8　正交表

$$A_i = \begin{array}{|c|c|c|} \hline A_{(i-1)} & 0 & A_{(i-1)} \\ \hline A_{(i-1)} & 1 & \bar{A}_{(i-1)} \\ \hline \end{array}$$

设最基本的二行一列子表是 $A_1 =$

	1
1	0
2	1

则直接由它推导出的 4 行 3 列的子表 A_2 是：

$A2 =$

	1	2	3
1	0	0	0
2	1	0	1
3	0	1	1
4	1	1	0

由此，当因素分析表中的因子数 m 确定下来之后，相应的正交表的大小就可以确定，再根据上述规律，自动生成正交表。

例如，前面所举被测工程数据库查询语言例，因素分析表中因子数为 3，则生成的正交表大小为 4 行 3 列，见图 4.9。把这个正交表应用到因素分析表中，就可以设计满足要求的测试数据了。但是，在一般情况下，各个因子的状态数是杂乱的、不统一的，几乎不可能出现均匀的情况。为此，必须首先用逻辑命令来组合各因子的状态，作出布尔图，然后再考虑正交表在布尔图中各结点上的应用。

因子 / 项目	A	B	C
1	0	0	0
2	1	0	1
3	0	1	1
4	1	1	0

图 4.9　正交表

将因素分析表中各因子的状态作为输入，结果作为输出，用逻辑命令把输入与输出结合在一起，画出布尔图。假设图中最左部并列的结点是各因子的所有可能状态，把它们按因子分组，每组是某一个因子的所有状态，组内各状态间是逻辑或（OR）的关系，如果一个因子的状态数超过 2，还要增加中间结果结点，以保证所有中间结点都是两个输入端。图的最右部是结果，它的输入端个数等于因子个数，结果的所有输入之间是逻辑与(AND)的关系。

由前例而得到图 4.10 表示的布尔图。结点 11 是结果，当它为 1 时，则输入应全部为 1，由此可知，结点 21、22、23 的输出应都是 1。为了保证它们是 1，则这几个结点的输入必须是 $\begin{bmatrix}0\\1\end{bmatrix}$ 或者 $\begin{bmatrix}1\\0\end{bmatrix}$，这里排除 $\begin{bmatrix}1\\1\end{bmatrix}$ 的情形，因为同一因子的所有状态不能在一次测试时同时出现。

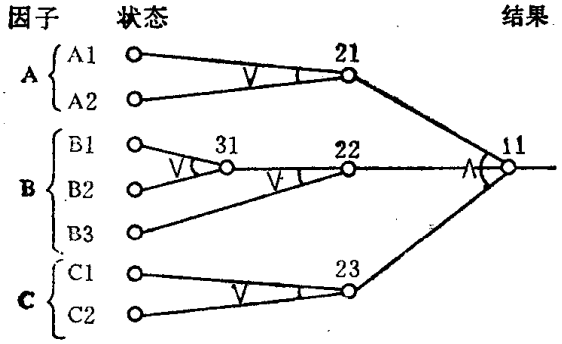


图 4.10　布尔图

把结点 21、22、23 当作因子，则它的输入可以当作状态，这时就可以使用正交表了。

正交表(图 4.8）中的 0 和 1 分别代表两种不同的状态，故可以把状态为 0 的替换成结点输入值 $\begin{bmatrix}1\\0\end{bmatrix}$，把状态为 1 的替换成结点输入值 $\begin{bmatrix}0\\1\end{bmatrix}$。见图 4.11。

针对中间结点 31，可做同样的工作(图 4.12)。最后可得到具有 6 组测试数据的测试数据

结点 / 组合号	21	22	23
	A1 A2	31 B3	C1 C2
1	1 0	1 0	1 0
2	0 1	1 0	0 1
3	1 0	0 1	0 1
4	0 1	0 1	1 0

图 4.11　替换中的正交表

结点 / 组合号	31
	B1 B2
1	1 0
2	0 1

图 4.12　继续替换中的正交表

集，见图 4.13。

因子 / 测试组号	A	B	C
1	A1	B1	C1
2	A1	B2	C1
3	A2	B1	C2
4	A2	B2	C2
5	A1	B3	C2
6	A2	B3	C1

图 4.13　测试输入数据集

利用这些测试数据，构造因果图，可得到实际可用的测试用例。

四、方法评价

利用正交实验设计法设计测试用例，比使用等价类划分、边值分析、因果图等方法，具有一定的优点：

① 节省测试工作工时。由于把实验设计法的正交表直接应用到因素分析表，简单地设定测试用例，比其他设计测试用例的方法要单纯，不一定非要具备丰富的经验与创造力。可以减轻测试者的负担。

② 可控制生成的测试用例的数量。若考虑各因子的所有状态的组合，来设计测试用例，从测试工作量和所用时间等方面来看，是不可能实现的。利用加权筛选，考虑各因子与状态的重要程度及因子间的相互影响，作出一定的筛选，可以把生成的测试用例的数量，控制在允许的范围。

因子 / 项目	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…
T_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
T_2	1	0	1	1	0	1	1	0	1	1	0	1	1	0	
T_3	0	1	1	0	1	1	0	1	1	0	1	1	0	1	
T_4	1	1	0	1	1	0	1	1	0	1	1	0	1	1	
T_5	0	0	1	1	1	0	0	0	1	1	1	0	0	0	
T_6	1	1	0	0	0	1	1	1	0	0	0	1	1	1	
T_7	0	1	1	1	0	1	1	0	0	0	1	0	0	1	
T_8	1	0	0	0	1	0	0	1	1	1	0	1	1	0	
T_9	0	1	0	0	1	0	1	1	0	0	1	0	1	0	
T_{10}	1	0	1	1	0	1	0	0	1	1	0	1	0	1	

图 4.14　改造的正变表

③ 测试用例具有一定的覆盖度。实验设计法是一种合理的、有效的抽样方法，对被测软件来说，测试用例的涉及范围在整体上比较均匀，可排除偏向于功能的某个局部的可能性，与结构测试相配合，可检出软件中大部分错误。检出故障率达 50% 以上。

使用正交实验设计法设计测试用例的问题是当因子数增加时，正交表中的项目数会急剧增加。例如，因子数从 3 增加到 4，项目数将从 4 增加到 8；同样地，因子数从 7 增加到 8 时，项目数将从 8 增加到 16。

为了减缓当因子数增加时项目数急剧增加的趋势，可以对图 4.8 所示的正交表予以改造。图 4.14 就是一个改造了的正交表。

把它应用于因素分析表，依照设计测试用例的步骤，就可以得到一组测试用例，但其数目比正交表未改造前就减少了许多。

4.4 边值分析

在软件设计和程序编写中，常常对于规格说明中的输入域边界或输出域的边界不够注意，以致形成一些差错。实践表明，在设计测试用例时，对边界附近的处理必须给予足够的重视，为检验边界附近的处理专门设计测试用例，常常取得良好的测试效果。

比如，在作三角形计算时，要输入三角形的三个边长：A、B 和 C。我们应注意到这三个数值应当满足 A＋B＞C、A＋C＞B 及 B＋C＞A 才能构成三角形。但如果把三个不等式的任何一个大于号＞错写成大于等于号≥，那就无法构成三角形。问题恰恰出现在那些容易被疏忽的边界上。这里所说的边界是指，相对于输入等价类和输出等价类而言，稍高于其边界值及稍低于其边界值的一些特定情况。

针对边界值设计测试用例时，应注意遵循以下几条原则：

① 如果输入条件规定了取值范围，或是规定了值的个数，则应以该范围的边界内及刚刚超出范围的边界外的值，或是分别对最大、最小个数及稍小于最小、稍大于最大个数作为测试用例。例如，如果程序的规格说明中规定："重量在 10 公斤至 50 公斤范围内的邮件，其邮费计算公式为……"。作为测试用例，我们应取 10 及 50，还应取 10.01，49.99，9.99及50.01 等。如果另一问题规格说明规定："某输入文件可包含 1 至 255个记录，……"，则测试用例可取 1 和 255，还应取 0 及 256 等。

② 针对规格说明的每个输出条件使用前面的第①条原则。例如，某程序的规格说明要求计算出"每月保险金扣除额为 0 至 1165.25 元"，其测试用例可取 0.00 及 1165.25，还可取－0.01 及 1165.26 等。如果另一程序属于情报检索系统，要求每次"最多显示 4 条情报摘要"，这时我们应考虑的测试用例包括 1 和 4，还应包括 0 和 5 等。

③ 如果程序规格说明中提到的输入或输出域是个有序的集合（如顺序文件、表格等），就应注意选取有序集的第一个和最后一个元素作为测试用例。

④ 分析规格说明，找出其它的可能边界条件。

以下给出实例，借以说明在具体问题中怎样从边界值的分析中考虑测试用例。

某一为学生考试试卷评分和成绩统计的程序，其规格说明指出了对程序的要求：

"程序的输入文件由 80 个字符的一些记录组成，这些记录分为三组：

① 标题

这一组只有一个记录，其内容为输出报告的名字。

② 试卷各题标准答案记录

每个记录均在第 80 个字符处标以数字“2”。该组的第一个记录的第 1 至第 3 个字符为题目编号（取值为 1—999）。第 10 至第 59 个字符给出第 1 至第 50 题的答案（每个合法字符表示一个答案）。该组的第 2，第 3，……个记录相应为第 51 至第 100，第 101 至第 150，……题的答案。

③ 每个学生的答卷描述

该组中每个记录的第 80 个字符均为数字 “3”。 每个学生的答卷在若干个记录中给出。如甲的首记录第 1 至第 9 字符给出学生姓名及学号，第 10 至第 59 字符列出的是甲所做的第 1 至第 50 题的答案。若试题数超过 50，则其第 2，第 3，……记录分别给出他的第 51 至第 100，第 101 至第 150，……题的解答。然后是学生乙的答卷记录。

若学生最多为 200 人，输入数据的形式如图 4.15 所示。

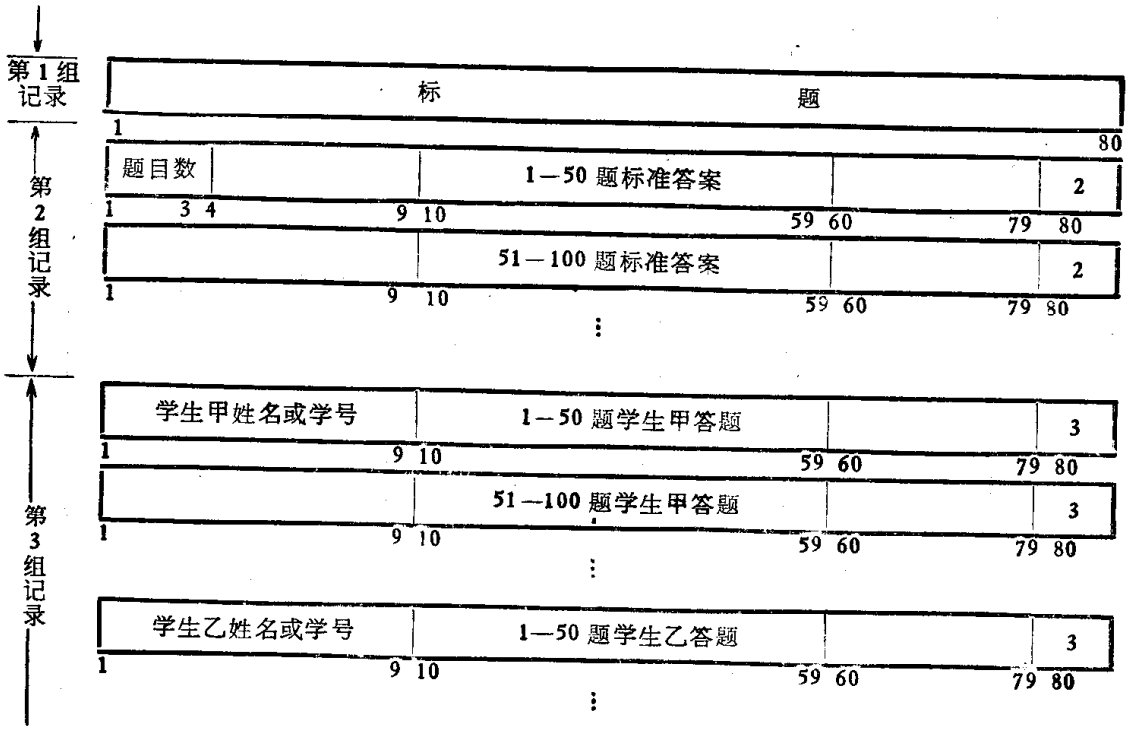


图 4.15　学生考卷评分和成绩统计程序输入数据形式

该程序应给出 4 个输出报告，即：

① 按学生学号排序，每个学生的成绩（答对的百分比）和等级报告。

② 按学生得分排序，每个学生的成绩。

③ 平均分数，最高与最低分之差。

④ 按题号排序，每题学生答对的百分比。

表 4.4 及表 4.5 分别针对输入条件和输出条件，根据其边界值设置了测试用例。

这里共 43 个测试用例，其中有些若不用边值分析方法就很难考虑到。但它们常常是会出现的情况，由此可见，边值分析确是有效而实用的方法。

表 4.4

输入条件	测试用例
输入文件	空输入文件
标题	无标题记录 只有 1 个字符的标题 具有 80 个字符的标题
出题个数	出了 1 个题 出了 50 个题 出了 51 个题 出了 100 个题 出了 999 个题 没有出题 题目数是非数值量
答案记录	标题记录后没有标准答案记录 标准答案记录多 1 个 标准答案记录少 1 个
学生人数	学生人数为 0 学生人数为 1 学生人数为 200 学生人数为 201
学生答题	某学生只有 1 个答卷记录，但有 2 个标准答案记录 该学生是文件中的第 1 个学生 该学生是文件中最后 1 个学生
学生答题	某学生有 2 个答卷记录，但仅有 1 个标准答案记录 该学生是文件中的第 1 个学生 该学生是文件中最后 1 个学生

表 4.5

输出条件	测试用例
学生得分	所有学生得分相同 所有学生得分都不同 一些学生(不是全部)得分相同(用以检查等级计算) 1 个学生得 0 分 1 个学生得 100 分
输出报告 ①，②	1 个学生编号最小(检查排序) 1 个学生编号最大 学生数恰好使报告印满 1 页(检查打印) 学生人数使报告 1 页打印不够，尚多 1 人
输出报告 ③	平均值取最大值(所有学生均得满分) 平均值为 0 (所有学生都得 0 分) 标准偏差取最大值(1学生得 0 分，1 学生得 100) 标准偏差为 0 (所有学生得分相同)

续表

输出条件	测　试　用　例
输出报告④	所有学生都答对第1题 所有学生都答错第1题 所有学生都答对最后1题 所有学生都答错最后1题 报告打印完1页后，恰剩1题未打 题数恰好使得报告打印在1页上

4.5　判定表驱动测试

一、什么是判定表

在一些数据处理问题中，某些操作是否实施依赖于多个逻辑条件的取值。也即在这些逻辑条件取值的组合所构成的多种情况下，分别执行不同的操作。处理这类问题的一个非常有力的分析和表达工具是判定表（Decision Table）。

让我们首先看一个简单的实例，用以说明什么是判定表。

在翻开一本技术书的几页目录之后，读者看到一张表，名为“本书阅读指南”。表的内容给读者指明了在读书过程中可能遇到的种种情况，以及作者针对各种情况给读者的建议(参看图4.16)。表中列举了读者读书时可能遇到的三个问题，若读者的回答是肯定的(判定取真值)，标以字母“Y”；若回答是否定的(判定取假值)，标以字母“N”。三个判定条件，其取值的组合共有8种情况。该表为读者提供了4条建议，但并不需要每种情况都施行。这里把要实施的建议在相应栏内标以“×”，其它建议相应的栏内什么也不标。例如，表中的第3种情况，当读者已经疲劳，对内容又不感兴趣，并且还没读懂，这时作者建议去休息。

		1	2	3	4	5	6	7	8
问题	你觉得疲倦吗?	Y	Y	Y	Y	N	N	N	N
	你对内容感兴趣吗?	Y	Y	N	N	Y	Y	N	N
	书中的内容使你糊涂吗?	Y	N	Y	N	Y	N	Y	N
建议	请回到本章开头重读	×				×			
	继续读下去		×				×		
	跳到下一章去读							×	×
	停止阅读，请休息			×	×				

图4.16　“读书指南”判定表

早在程序设计发展的初期，判定表就已被当作编写程序的辅助工具使用了。由于它可以把复杂的逻辑关系和多种条件组合的情况表达得既明确又具体，因而给编写者、检查者和读者均带来很大方便。

判定表通常由4个部分组成，如图4.17所示，用双线分割的四个部分是：

条件茬 (Condition Stub)

动作茬 (Action Stub)

条件项 (Condition Entry)

动作项 (Action Entry)

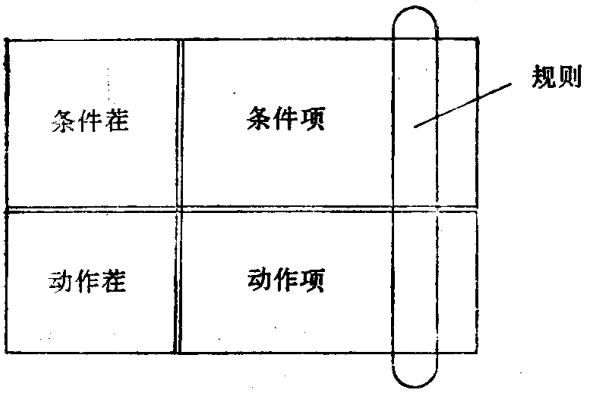


图 4.17　判定表的4个组成部分

条件茬部分列出了问题的所有条件，除了某些问题对条件的先后次序有特定的要求外，通常在这里列出的条件，其先后次序无关紧要。条件项部分是针对它左列条件的取值，在所有可能情况下，给出真假值。动作茬则列出了问题规定可能采取的操作。这些操作的排列顺序一般并没有什么约束，但为了便于阅读也可令其按适当的顺序排列。动作项和条件项紧密相关。它指出了在条件项的各组取值情况下应采取的动作。我们把任何一个条件组合的特定取值及其相应要执行的操作称为规则。在判定表中贯穿条件项和动作项的一列就是一条规则。显然，判定表中列出多少组条件取值，也就有多少条规则，即条件项和动作项有多少列。

在实际使用判定表时，常常先把它化简。化简工作是以合并相似规则为目标的。若表中有两条或多条规则具有相同的动作，并且其条件项之间存在着极为相似的关系，我们便可设法将其合并。例如，在图4.18中左端的两规则其动作项一致，条件项中前两条件取值一致，只是第三条件取值不同。这一情况表明，在第1、2条件分别取真值和假值时，无论第3条件取任何值，都要执行同一操作。也即要执行的动作与第3条件的取值无关。于是，我们便将这两个规则合并。合并后的第3条件项用特定的符号“—”表示与取值无关。

与此类似，无关条件项“—”在逻辑上又可包含其它的条件项取值，具有相同动作的规则还可进一步合并，如图4.19所示。

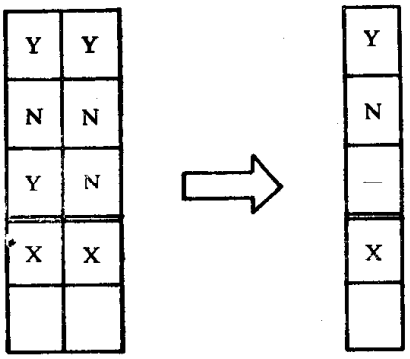


图 4.18　两条规则合并成一条

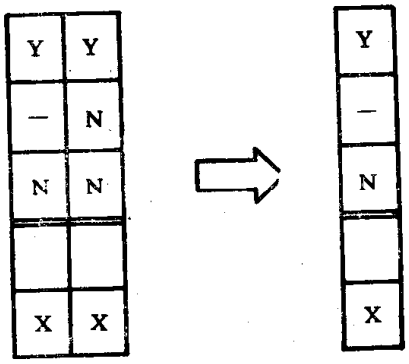


图 4.19　两条规则的进一步合并

按上述合并规则的办法，可把“读书指南”判定表加以简化，简化后的判定表见图4.20。

		1	2	3	4
问题	你觉得疲倦吗?	—	—	Y	N
	你对内容感兴趣吗?	Y	Y	N	N
	书中内容使你糊涂吗?	Y	N	—	—
建议	请回到本章开头重读	×			
	继续读下去		×		
	跳到下一章去读				×
	停止阅读,请休息			×	

图 4.20　化简后的"读者指南"判定表

以下结合实例给出构造判定表的方法。若问题要求："……对功率大于 50 马力的机器、维修记录不全或已运行 10 年以上的机器，应给予优先的维修处理……"。这里假定，"维修记录不全"和"优先维修处理"均已在别处有更严格的定义。按以下 5 步建立判定表。

① 确定规则的个数。这里有 3 个条件，每个条件有两个取值，故应有 $2^3 = 8$ 种规则。

② 列出所有的条件茬和动作茬。

③ 填入条件项。为防止遗漏可从最后 1 行条件项开始，逐行向上填满。如第三行是：

Y N Y N Y N Y N

第二行是：

Y Y N N Y Y N N

等等。

④ 填入动作茬和动作项。这样便得到形如图 4.21 的初始判定表。

⑤ 化简。合并相似规则后得到图 4.22。

判定表最突出的优点是，它能把复杂的问题按各种可能的情况一一列举出来，简明而易于理解，也可避免遗漏。它的不足之处在于，不能表达重复执行的动作，例如循环结构。

二、判定表在功能测试中的应用

一些软件的功能需求可用判定表表达得非常清楚，在检验程序的功能时判定表也就成为一个非常有力的工具。

如果一软件的规格说明指出：

① 当条件 1 和条件 2 满足，并且条件 3 和条件 4 不满足，或者当条件 1、3 和条件 4 满足时，要执行操作 1。

		1	2	3	4	5	6	7	8
条件	功率大于 50 马力吗?	Y	Y	Y	Y	N	N	N	N
	维修记录不全?	Y	Y	N	N	Y	Y	N	N
	运行超过 10 年?	Y	N	Y	N	Y	N	Y	N
动作	进行优先维修	×	×	×		×		×	
	作其它处理				×		×		×

图 4.21　初始判定表

		1	2	3	4	5
条件	功率大于 50 马力吗?	Y	Y	Y	N	N
	维修记录不全吗?	Y	N	N	—	—
	运行超过 10 年吗?	—	Y	N	Y	N
动作	进行优先维修	×	×		×	
	作其它处理			×		×

图 4.22　化简后的判定表

	规则 1	规则 2	规则 3	规则 4
条件 1	Y	Y	N	N
条件 2	Y	—	N	—
条件 3	N	Y	N	—
条件 4	N	Y	N	Y
操作 1	×	×		
操作 2			×	
操作 3				×

图 4.23　根据规格说明得到的判定表

② 在任一个条件都不满足时，要执行操作 2。

③ 当条件 1 不满足，而条件 4 被满足时，要执行操作3。

根据上述说明，我们可给出图 4.23 表示的判定表：表中 "Y" 表示"是"，"N" 表示"否"。"—"表示条件取值是无关紧要的。

很显然，这个判定表中4个条件的16种规则并未一一给出。只是给出了16种规则中的4种。事实上，除这4条以外的一些规则是指当不能满足指定的条件，执行3种操作时，要执行1个默许的操作。在没有必要时，判定表通常可以略去这些规则。但如果我们打算用判定表来支持设计测试用例，那就有必要列出这些默许的规则(见图4.24)。

	规则5	规则6	规则7	规则8
条件1	–	N	Y	Y
条件2	–	Y	Y	N
条件3	Y	N	N	N
条件4	N	N	Y	–
默许操作	×	×	×	×

图4.24 默许的规则

B.Beizer 指出了适合于使用判定表设计测试用例的条件：

① 规格说明以判定表形式给出，或是很容易转换成判定表。

② 条件的排列顺序不会也不应影响执行哪些操作。

③ 规则的排列顺序不会也不应影响执行哪些操作。

④ 每当某一规则的条件已经满足，并确定要执行的操作后，不必检验别的规则。

⑤ 如果某一规则得到满足要执行多个操作，这些操作的执行顺序无关紧要。

B.Beizer 提出这5个必要条件的目的是为了使操作的执行完全依赖于条件的组合。其实对于某些不满足这几条的判定表，同样可以借以设计测试用例，只不过尚需增加其它的测试用例罢了。

4.6 功能测试

功能测试是解决模块测试问题的一类重要测试方法。与结构测试不同，它完全不顾模块内部的实现逻辑，以检验输入输出信息是否符合规格说明书中有关功能需求的规定为目标。测试者要根据规格说明书，从不同的方面提出普通情况和特定情况下的测试用例。为了使测试取得成功，测试者必须对规格说明书非常熟悉。

近年来已经发展了多种功能测试方法。事实上本章前面介绍的等价类划分、边值分析以及利用判定表或因果图等方法均属功能测试，本节将对功能测试的系统化和模块功能分解测试展开讨论。

一、功能测试的系统化

既然功能测试的依据是规格说明书，我们就必须十分重视规格说明书，并对其进行仔细的分析。规格说明书尽管可以有各种各样的形式，但都必须有一个共同的特征，就是要对模块的功能进行无遗漏、无冗余、无矛盾的描述。如果说明书是形式化的，我们可以严格按照形式的要求，有步骤地生成测试数据，以检验各功能的实现情况。如果说明书不是

形式化的，则需要从中提取有关功能的准确信息，并据此构造相应的测试数据。一般说来，在模块的各变量输入域内，如果变量取某个子域的值，程序执行的路径不变，那么这一子域就可看作是一个等价类。便可以从中找出若干个代表点，在不严重地影响可靠性的前提下(实际上，只有对输入值的所有组合进行测试，才能保证测试的完全可靠性)，大大地减少测试的工作量。但仅仅这样做是不够的。我们还要注意到输入变量之间的关系，并生成测试用例去检验这些关系约束下程序运行的情况及结果。另外，对于不正确的输入值，也需选其代表，考察模块对这些异常值的处理能力。其实，这些在前面讨论过的等价类划分方法中已经有所体现了。

以下我们对数学计算型函数的功能测试进行分析。这些分析的思想也可扩充到其它类型的函数测试中去。

在功能测试中，一个模块可以看作是一组变量到另一组变量之间的映射。我们这样来表示一个模块 M:

$$M=:(x_1,x_2,\cdots,x_i\xrightarrow{f}y_1,y_2,\cdots,y_j)$$

其中，=:表示定义为，$x_k(k=1,\cdots,i)$ 是模块的输入变量，$y_k(k=1,\cdots,j)$ 是模块的输出变量，f 是模块所计算的函数。

对于每个 $x_k(k=1,\cdots,i)$，都有一对应的定义域 D_k。对于每个 $y_k(k=1,\cdots,j)$，有一对应的域值 v_k。功能测试的准则就是要在D中找出代表性的点，并且在D中找点，使得 f 产生 v 中有代表性的点。这里 $D=D_1\times D_2\times\cdots\times D_i$，$v=v_1\times v_2\times\cdots\times v_j$，其中，×是笛卡尔积。

以下按输入变量是数字型简单变量、数组或向量型变量及多维数组等情况分别加以说明:

1. 数字型简单变量

当输入变量的定义域 D_k 是由数字型变量组成时，域中元素的唯一重要特性就是其实际值。定义域可以是由一些离散的点组成，也可以由实数开、闭区间组成。如果定义域由离散点组成，这些离散点在一般情况下都是各自独立的。因此，要想完整地测试模块，我们就必须把每一个离散点都当做是测试点。如果输入域是由一组整、实数区间组成的，按照边界值原理，应当将每一区间的边界点都作为测试点，同时，每一区间内部至少还应有一个测试点。

除了依据输入变量定义域 D_k 进行测试以外，还应该考虑输出变量的值域 v_k，以便于找出漏掉程序分支等错误。对于由离散值组成的输出域，我们应该精心选择输入变量定义域中的测试点，使得这些离散值都能产生。对于由一组整实数区间组成的输出域，应该这样选择测试点，使每个区间的边界点值和至少一个内部值生成。

此外，还应该产生不合法的测试点，以检查程序对错误的处理能力。如某一程序对6位数字的电话号码进行处理，我们就应产生7位数、5位数或是有字母的字符串，进行测试。如果输入输出变量域中有不同类型的元素，则对每一类型都应进行测试。

对于数字计算程序，一个区间内部仅有一个测试点是不够的。某些具有特殊性质的点也应在测试点之列。如某一函数计算的输出域有一拐点，则应将产生这一拐点的输入值和其两端近邻的值作为测试点，以测试拐点位置的正确性。其它常用的特殊点为 0、1，

绝对值很大或很小的点。如果两个输入变量存在某种内部联系，我们在选择测试点时，也要注意充分展示这两个变量之间的关系。在某些程序类型中，一个变量的定义域取决于另一个变量的值。例如，在一个程序中，x 和 y 是两个输入变量。其中，x 的定义域为[0，∞)，y 的定义域为 [0,y]，取决于 x。这时，测试点的选取应注意到这一依赖关系所引出的边界点。对于这一问题，应该至少考虑下列几组测试数据：

① x = 0, y = 0
② x > 0, y = 0
③ x > 0, 0 < y < x
④ x > 0, y = x
} 正确输入

⑤ x < 0, y 任意
⑥ x > 0, y > x
⑦ x = 0, y > 0
} 非法输入

2. 数组、向量型变量

对于数组和向量型变量的测试要更复杂一些。比如，可变数组的测试，除去数组中各元素的值是测试的考虑对象以外，其本身大小也是应该测试的。对于多维可变数组，则其每一维的大小都应得到测试。例如，数组 A 定义为：

```
M:2..20;
N:2..100;
A: Array[1..M][1..N] OF integer;
```

则我们不仅要考虑 A 的各分量的值，还要考虑 A 的大小，即对 M、N 的值选取相应的测试点。通常，对一个 K 维可变数组，我们就其每一维的大小选取两个边界点和一个内点进行测试，需要 3^k 个不同的测试点。在上例中，M 选取两个边界点 2,20 和一个内点 10；A 选取两个边界点 2,100 和一个内点 40，则相对于维数测试的点就至少要有以下 9 组组合：

M	2	2	2	10	10	10	20	20	20
N	2	40	100	2	40	100	2	40	100

数组元素值组成一个集合。此集合可以看作单个实体，也可看作单个变量值的集合。在某些特殊情况下，数组各元素之值可能是相同的。在另外一些场合，数组各元素的定义域都一致，这时也可选取测试数据使数组各元素的值都一致。其实，这种情形也是很常见的。选取的标准类似于单个变量测试点的选择标准，一般情况下是定义域区间的边界值和特殊内点值。特殊内点值与单个变量的特殊内点值有相似之处，如数组各元素的值均为零。

数组各元素之间还会有一定的关系。对于选择测试点来说，某些关系是不依赖于具体问题的。如通常我们都应选择测试数据，使数组元素的值两两不相等。上面所说的数组各元素之间都一致实际上也是数据各元素之间的一种特殊关系。对于特殊的问题，相

对于数组的测试数据也会有特殊的要求。比如，有的模块就要求其输入数组的值单调升降；有的要求其输入数组各元素的值满足某一函数关系，如输入数组是裴波那奇数组(Fibonacci array)。对于这一类有特殊关系的输入数组，应构造测试数据，使其满足所要求的关系。还应构造测试数据，使其不满足那些关系。

在很多场合下，数组的某一元素可能会取特殊值。这时也要构造测试数据来测试与此相对应的情况。

总之，在有关输入数组的测试中，应当精心选择测试数据，使测试数组各维的大小与测试数组的元素相结合起来。

3. 多维数组

在输入变量是多维数组时，我们往往会遇到这种情况，即其某一复合分量满足一定的规范结构。这时，我们就应以此复合分量为基本的测试数据生成单位，而不应在更低的一层上去产生测试数据。比如，某一 $m \times n$ 矩阵是用来表示m个 $n-1$ 次多项式系数的。某一程序模块的功能就是检验这m个 $n-1$ 次多项式是否线性相关。输入矩阵用一个二维数组 A[m, n] 表示。我们知道，此问题中可以把每个多项式系数看作一个向量。问题转化为检验m个n维向量是否线性相关。A[m, n] 可以表示为：

$$(\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_m)$$

其中，$\vec{v}_i (i = 1, \cdots, m)$ 是由n个分量组成的向量，即：

$$\vec{v}_i = \begin{pmatrix} v_{i1} \\ v_{i2} \\ \vdots \\ v_{in} \end{pmatrix}$$

要测试这一模块，如果产生测试数据时，考虑对象为A的基本元素 A[i,j]，则测试既抓不住要害，还可能遗漏一些重要测试点。因此，我们的基本对象应该是 $\vec{v}_i$。即测试时应考虑的是 $\vec{v}_i$ 取什么值，而不是 A[i,j] 取什么值。除了把测试的对象抽象了一层外，别的测试点选择策略大多并未失效，它们在抽象的一层上仍然发挥作用。在多维数组的测试点选择过程中，有时这样的抽象要进行多次才能找到基本元素。

如果输入变量较少，我们可以按上述方法对输入变量的各种组合进行测试。但如果输入变量很多，这种组合就会导致通常所说的"组合爆炸"现象。那么怎样来解决这一问题呢？一种简单易行的方法就是对各种组合划分等价类，将执行同一函数功能的组合值划分到同一等价类中去，从每一类中找出代表来进行测试。显而易见，找等价类是紧密依赖于具体功能模块的，因而不会有统一的方案。实际上，在上面所提到的测试基本方法中，等价类的隐藏概念散落在各处。我们所说的找输入区间的边界点进行测试，实际上就是找等价类。因为模块处理边界点的函数与处理内点的函数是否一致需得到有效的检验。

二、模块功能的分解测试

以上讨论的功能测试基本上是针对一个程序模块的功能来说的。除了有关多维数组

的测试点选择过程中需要对模块实现的细节有一定了解以外，其余的测试仅依据功能说明就足够了。

但是，功能说明本身常常是有缺陷的。其中一个重要的不足之处就是叙述得过分笼统，自然也就不会涉及到任何有关实现的信息。当然，按理想化的条件，为了使功能测试的测试者不受编程人员实现思路的干扰，他们最好对模块的实现方案和细节一无所知，只需了解作为一个黑盒的模块对外的功能如何。但这一点在实际中往往是做不到的。也许一种折衷的情况更现实，这就是在提醒测试者不受程序编制人员先入之见影响的前提下，尽可能全面地了解程序模块的实现情况。以此来弥补上述规格说明之不足。

一般来说，程序模块或函数常常是由一组子函数合成与联结起来的。按照可维护性的程序编制要求，在子函数的前面都应缀有有关的说明信息。测试者可以把这些说明信息看作是子函数的非规范化功能说明书。从而，可以更进一步地展开对子函数的功能测试，这也称为模块功能分解测试。W.E. Howden 称这些子函数为设计函数。

一个函数的子函数可以以不同的形式出现。但就一个函数各子函数之间的关系来说，设计函数可以分为三类。

第一类设计函数与程序中的某一段直接对应，因而很容易辨认。最简单的例子就是几个设计函数顺序地组成一个简单程序。该程序可以由某一选择变量来决定运行哪一种功能。例如下述程序：

```
Procedure A (int I);
    {  case I = 1:
                    ⋮          }设计函数 1
          I = 2:
          ⋮         ⋮          }设计函数 2
          I = n:
                    ⋮          }设计函数 n
       endcase
    }
```

这类设计函数组成的函数结构如图 4.25(a)所示。其中实线表示控制转换到子函数，虚线表示控制从子函数返回，圆表示决断机制。

另一类比较常见的函数构成是程序被分为有独立意义的多段代码。每一段代码实际上就是一个设计函数，程序顺序地执行诸段代码，如下述程序所示：

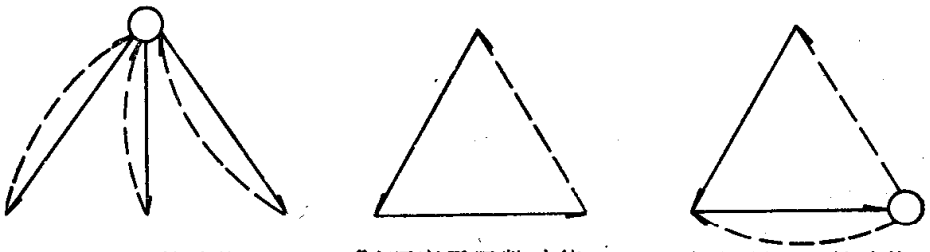
```
Procedure B;
    {  计算支出;——设计函数 1
       计算收入;——设计函数 2
       计算余额(收入、支出);——设计函数 3
    }
```

这类函数的设计函数组成结构如图 4.25 (b) 所示。图中水平线上的箭头代表着控制从一个设计函数传到另一设计函数，而且这些设计函数都是处于同一抽象级的。垂直方向的控制传输表示函数控制向低抽象层次的设计函数的传递。

以上两类设计函数都很容易识别，它们都对应程序中的一段代码。我们可以把它们称为计算型设计函数。因为它们都有计算值，这些值或被作为整个程序模块的输出，或是下一设计函数的输入。第三类设计函数却不是计算型，而是反映函数的某种控制关系，因而可称为控制型设计函数。在数字型计算函数中，控制型设计函数常常被用来选择计算函数或终止循环与递归过程。如下述程序所示：

```
Procedure C (int i, j);
    Boolean T;
    {  IF i>j THEN T = TRUE
              ELSE T = FALSE
       WHILE T DO
         {  i = i - 1;
            j = j + 1;
            IF i≤j THEN T = FALSE
         }
    }
```

由控制型设计函数组成的程序模块如图 4.25 (c) 所示。图中两条虚线中横向者代表循环继续进行，纵向虚线代表控制返回。



(a)平行子函数功能　(b)顺序子函数功能　(c)控制子函数功能

图 4.25　三类设计函数

在将函数的子函数作如上分类后，就可进行设计函数这一层的测试了。由于模块级上的测试过粗，有时并不能测到所有的计算子结构。而且很多变量和数据结构对整个模块并无特殊意义，但在某设计函数上却成为关键点。因此，我们要想比较详细地测试一个模块，就必须进行设计函数一层的测试。例如，一个程序模块用函数逼近方法计算某一值，当满足某一条件时，就应终止逼近过程。但假设逼近过程的控制函数不正确，在应该终止后，还进行了很多次冗余计算。如果我们仅对整个模块进行功能测试，我们会认为该模块是正确的。实际上，整个程序模块效率极低。而在低一层上测试控制函数时，则可轻而易举地找出错误来。又如，某模块计算函数 f，其输入变量 x 也是 f 的一个设计函数 f_0 的输入变量。假设对于 f，x 的定义域是$(-\infty, \infty)$，而对于 f_0，x 的定义域是$[2, \infty)$，即仅当 $x \geqslant 2$ 时，f_0 才执行。这样，如果我们仅在模块级上进行功能测试，根据选两个边界点的原则，我们会选择-k，k 作为测试点（这里 k 的值很大）。如果在设计函数一级上进行测试，我们就应选择 2，k 作为测试点了。此外，还要测试非法状况 $2-\varepsilon$（这里 ε 的值很小）。而 2 和 $2-\varepsilon$ 两个测试点在模块级功能测试上一般是不会有的。

一般说来，模块的设计函数常常是满足一定条件才运行的，这些条件为此设计函数规定了输入域。此输入域的边界点一般不会是模块功能测试的测试点，但它们却反映了模块设计的一些敏感部位，因此是值得进行测试的。设计函数的测试满足了这一要求。

设计函数的功能测试要求每一设计函数的输入输出域都清楚地给出。一般情况下要求程序编制者做到这一点并不难。因为要求程序内给出必要的注释，特别是在模块开头注明输入输出域是完全合理的。至于有的设计函数并不直接对应程序中的某一段代码，对它的注解就得单独进行。

由于对设计函数的测试要求知道进入和退出设计函数时各变量的值，我们必须监视程序变量的值。通常测试环境都能监视程序变量的值。但进入一设计函数时，某些变量必须局限在某一范围之内。怎样选择模块的输入数据，使得该设计函数被执行就成了一个问题。解决这个问题的一个途径是把设计函数的输入变量看成独立的，利用测试驱动器为其独立赋值，而不考虑前面计算对其影响。当然，这样测试以后，还必须检验各设计函数之间的关系。整个测试过程很像是先进行模块测试，后进行系统整体测试方式的一个缩影。

关于这种系统化功能测试在实际应用中的效果，W.E. Howden 曾对统计和数据分析软件包 IMSL 中的程序进行过比较测试，得出功能测试和结构测试效果的对比结果(参看图 4.26)。其中，括号外数字表示用其左边相应方法最容易测得的错误个数，括号内数字表示左边相应方法能够查出的错误总数，也即结合其它较为适合的方法查出错误的个数。

方　　法	错误数
1.功能测试	
普通功能测试	14(20)
粗略计算设计函数功能测试	7(9)
详细计算设计函数功能测试	7(9)
控制设计函数功能测试	3(5)
共计	31(43)
2.结构测试	
分支结构测试	10(13)
路径结构测试	(3)
共计	10(16)

图 4.26　功能测试与结构测试的效果

可以看出，如果设计函数的测试进行得彻底，可以找出更深的隐藏错误，在某些情况下会比结构测试更有效。这里粗略计算设计函数功能测试与模块的主要计算分量有关，而详细计算设计函数功能测试在别的方面都与粗略计算设计函数功能测试一样，只是它将计算分量划分得更细。要找出这样细的计算分量，仅凭模块中的注解是不够的。测试者必须仔细研究模块的构成。

功能测试在 IMSL 类程序的测试中比结构测试更有效。实际上，我们也可想象，计算路径中往往有几个点是正确运行的，而其它点的处理都不正确。结构测试中，如果路径

的代表点选到处理正确的几个点，就无法发现错误。而功能测试迫使程序执行重要的边界点和特殊点，因而极有可能发现此类错误。但必须注意，这是以增加对模块的分析为代价的。

从图 4.26 中可看出，增加设计函数的功能测试后，功能测试发现的错误由 14(20) 提高到 31(38)，大约提高了一倍。由此可见，花力气测试设计函数还是值得的。在结构测试中未曾涉及的一些路径，可能被设计函数的测试得到检验。结构测试一般对循环的测试仅局限于绕过循环体，执行循环体一次或执行多次等。而设计函数功能测试却可能要求循环体执行特定的次数，从而易于查出错误。W.E. Howden 曾举出 IMSL 库中的两个程序的测试实例。其中的某些错误，通过结构测试无法找到，而通过粗略计算设计函数和详细设计函数的测试却可分别找出。

当然，并不是说，功能测试就一定强于结构测试。我们也可举出例子，说明功能测试不能找出的错误却能由结构测试找出。下列三种情况中，分支测试就远比功能测试有效。

① 当一分支执行时，程序总会出错；

② 程序中含有无法执行到的代码段；

③ 错误导致额外分支的执行。

事实表明，功能测试和结构测试各有千秋，谁也不能代替谁。它们只有互相结合，互相补充，才能进一步提高测试的可靠性。

参 考 文 献

[1] W.E. Howden, "Functional Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6(2), March 1980, pp. 162—169.

[2] W.E. Howden, "Functional Testing and Design Abstractions", The Journal of Systems and Software, Vol. 1(4), January 1980, pp. 307—313.

[3] Myers. G., The Art of Software Testing, John Wiley & Sons, 1979.

[4] W.R. Elmendorf, "Cause-Effect Graphs in Functional Testing", TR-00.2487, IBM Systems Development Division, Poughkeepsie, N.Y., 1973.

[5] Roger S. Pressman, Software Engineering—Apractitioner's Approach, Second Edition, McGraw-Hill Book Company, 1987.

[6] Richard B. Hurley, Decision Tables in Software Engineering, Van Nostrand Reinhold Company, 1983.

[7] Boris Beizer, Software Testing Techniques, Van Nostrand Reinhold Company, 1983.

[8] 佐藤忍，下川浩樹，"実験計画法を用いたソフトウエアのテスト項目設定法"，第4回ソフトウエア生産における品質管理シンポジウム，1984。

[9] 大場充，"プログラムの制御構造に基づくテストケース設計法"，第5回ソフトウエア生産における品質管理シンポジウム，1985。

[10] 田口玄一，"実験計画法"，丸善，1976。

[11] 高橋磐郎，"シミュレーションの直交実験による効率化"，数理科学，1979.12。

第五章 白盒测试

有关白盒测试的基本概念也曾在第三章中对照黑盒测试作了说明。本章将在此基础上简要介绍几个具体的白盒测试方法，其中包括程序控制流分析、数据流分析、逻辑覆盖、域测试、符号测试、路径分析、程序插装及程序变异等。其中多数方法比较成熟，也有较高的实用价值，个别的方法仍有些问题没有得到圆满地解决。例如，符号测试和路径测试的分析方法都是很重要的，但在程序分支过多及程序路径过多时，已有的方法将会显示出它们的局限性。相信这些问题总会得到逐步解决。

5.1 程序结构分析

程序的结构形式是白盒测试的主要依据。本节将从控制流分析、数据流分析和信息流分析的不同方面讨论几种机械性的方法分析程序结构。自然，我们的目的总是要找到程序中隐藏的各种错误。

一、控制流分析

由于非结构化程序会给测试、排错和程序的维护带来许多不必要的困难，人们有理由要求写出的程序是结构良好的。70年代以来，结构化程序的概念逐渐为人们普遍接受。体现这一要求对于若干新的语言，如 Pascal、C 等并不困难，因为它们都具有反映基本控制结构的相应控制语句。但对于早期开发的语言来说，要作到这一点，程序编写人员需要特别注意，不应忽视程序结构化的要求。使用汇编语言编写程序，要注意这个问题的道理就更为明显了。

正是由于这个原因，系统地检查程序的控制结构成为十分有意义的工作。

1. 控制流图

程序流程图（flowchart）又称框图，也许是人们最熟悉，也是最容易接受的一种程序控制结构的图形表示了。在这种图上的框内常常标明了处理要求或条件，这些在做路径分析时是不重要的。为了更加突出控制流的结构，需要对程序流程图做些简化。在图 5.1 中给出了简化的例子。其中（a）是一个含有两出口判断和循环的程序流程图，我们把它简化成（b）的形式，称这种简化了的流程图为控制流图（Control-flow graph）。

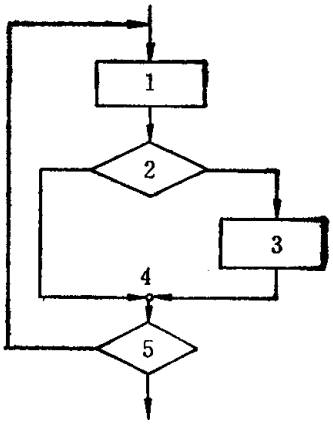
在控制流图中只有两种图形符号，它们是：

①节点：以标有编号的圆圈表示。它代表了程序流程图中矩形框所表示的处理、菱形表示的两至多出口判断以及两至多条流线相交的汇合点。

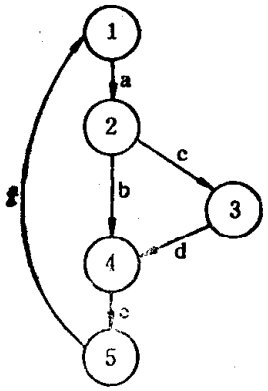
②控制流线或弧：以箭头表示。它与程序流程图中的流线是一致的，表明了控制的顺序。为讨论方便，控制流线通常标有名字，如图中所标的 a、b、c 等。

为便于在机器上表示和处理控制流图，我们可以把它表示成矩阵的形式，称为控制流

图矩阵（control-flow graph matrix）。图 5.2 表示了图 5.1 的控制流图矩阵。这个矩阵有 5 行 5 列，是由该控制流图中含有 5 个节点决定的。矩阵中 6 个元素 a、b、c、d、e



(a)程序流程图



(b)控制流图

图 5.1　程序流程图与控制流图

	a			
		c	b	
			d	
				e
f				

图 5.2　控制流图矩阵

和 f 的位置决定于它们所联接节点的号码。例如，弧 d 在矩阵中处于第 3 行第 4 列，那是因为它在控制流图中联接了节点 3 至节点 4 。这里必须注意方向。图中节点 4 至节点 3 是没有弧的，矩阵中第 4 行第 3 列也就没有元素。

2. 程序结构的基本要求

我们对于程序结构提出以下 4 点基本要求，这些要求是，写出的程序不应包含：

① 转向并不存在的标号；

② 没有用的语句标号；

③ 从程序入口进入后无法达到的语句；

④ 不能达到停机语句的语句。

显然，提出这些要求是合理的。在编写程序时稍加注意，做到这几点也是很容易的。这里我们更为关心的是如何进行检测，把以上 4 种问题从程序中找出来。

其实，前两种情况是很容易发现的，在此不作讨论。第 3 种情况可以利用图 5.3 给出的算法加以检验。该算法使用了栈操作。被检验的程序经此算法处理后，凡未作标记的节点均为不可达代码。

至于第 4 种情况也可用相似的方法找出。在图 5.3 所给出的算法中，只需把“入口节点”改为“出口节点”，“后继”改为“前趋”即可。

3. 结构分析

结构化程序在编写、理解和测试中的优越性是人所共知的。为了得到结构良好的程序，就不能不对程序编写人员提出一些要求。

图 5.4 给出了最初由 Dijkstra 提出的几种结构化程序中若干逐步细化（stepwise refinement）的流程图构造形式。图中每个节点“E”均可扩充，由赋值号右边的结构所代换。

熟悉计算机语言语法定义的读者可以看出，该图作为一个非形式的“流程图语法”（其中每一子图均表示一个产生规则），定义了结构化流程图。

正如使用语言的产生式规则，编译程序可以对程序作语法分析，我们可以利用流程图

```
begin
    清栈;
    为入口节点作标记;
    入口节点进栈;
    while 栈非空 do
        begin
            n: = 栈顶节点;
            出栈;
            为所有未标记的 n 的后继作标记
             且进栈
        end
end
```

图 5.3　检验不可达代码算法

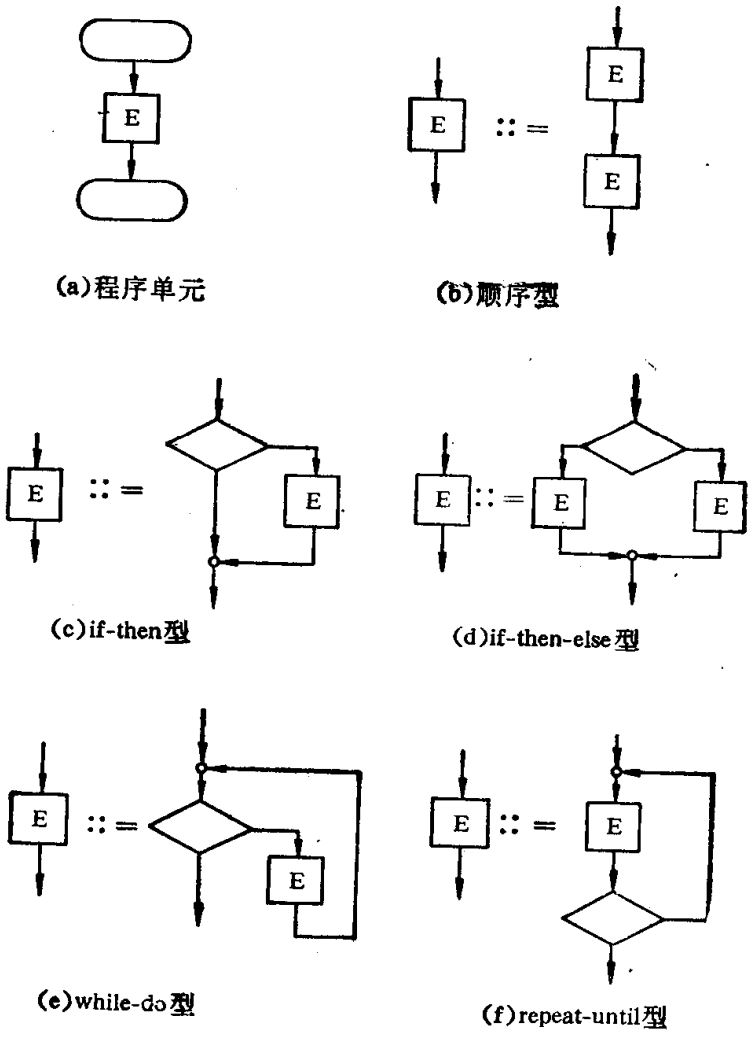


图 5.4　流程图语法

语法的产生式规则进行流程图的自底向上分析。用这种方法能够验证上述结构规则在程序编写中是否得到遵循，如果确已遵循，便可得到有关其构成成分的语法树。此外，还能揭示控制结构的缺欠。这在需作进一步分析时是很有用的。

当然，图 5.4 所给的结构程序规则是很有限的，通常还可作进一步扩充。例如，可以允许使用具有多出口的循环构造(如在 Ada 中的构造。其实，用其它语言的条件转向语句

也可模拟这类循环语句)。正如 Knuth 所指出的那样，使用这种构造可以提高程序的清晰性，也不会使程序正确性证明的工作更加困难。尽管允许这样的结构会使检查流程图结构是否良好更为复杂，但作其它的流程分析还是可行的。

二、数据流分析

数据流分析最初是随着编译系统要生成有效的目标码而出现的，这类方法主要用于代码优化。近年来数据流分析方法在确认系统中也得到成功的运用，用以查找如引用未定义变量等程序错误。也可用来查找对以前未曾使用的变量再次赋值等数据流异常的情况。找出这些错误是很重要的，因为这常常是常见程序错误的表现形式，如错拼名字、名字混淆或是丢失了语句。这里将首先说明数据流分析的原理，然后指明它可揭示的程序错误。

1. 数据流问题

如果程序中某一语句执行时能改变某程序变量V的值，则称V是被该语句定义的。如果一语句的执行引用了内存中变量V的值，则说该语句引用变量 V。例如，语句

X: = Y + Z

定义了 X，引用了Y和 Z，而语句

if Y > Z then goto exit

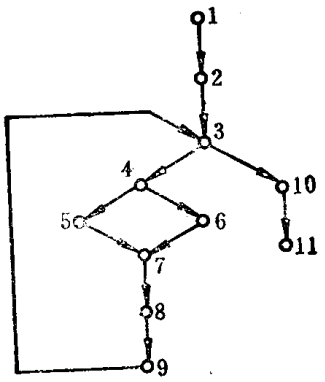
只引用了Y和 Z。输入语句

READ X

定义了 X。输出语句

WRITE X

引用了 X。执行某个语句也可能使变量失去定义，成为无意义的。例如，在 FORTRAN 中，循环语句 DO 的控制变量在经循环的正常出口离开循环时，就变成无意义的。



节　点	被定义变量	被引用变量
1	X,Y,Z	
2	X	W,X
3		X,Y
4		Y,Z
5	Y	V,Y
6	Z	V,Z
7	V	X
8	W	Y
9	Z	V
10	Z	Z
11		Z

图 5.5　控制流图及其定义和引用的变量

图 5.5 给出了一个小程序的控制流图，同时指明了每一语句定义和引用的变量。可以看出，第一个语句定义了 3 个变量 X,Y 和 Z。这表明它们的值是程序外赋给的。例如，该程序是以此三变量为输入参数的过程或子程序。同样，出口语句引用 Z 表明，Z 的值被

送给外部环境。

该程序中含有两个错误:

① 语句 2 使用了变量 W，而在此之前并未对其定义。

② 语句 5、6 使用变量 V，这在第一次执行循环时也未对其定义过。

此外，该程序还包含两个异常:

③ 语句 6 对 Z 的定义从未使用过。

④ 语句 8 对W的定义也从未使用过。

当然，程序中包含有些异常，如③、④也还不会引起执行的错误。不过这一情况表明，也许程序中含有错误；也许可以把程序写得更容易理解，从而能够简化验证工作，以及随后的维护工作（去掉那些多余的语句一般会缩短执行时间，不过在此我们并不关心这些)。

以下介绍自动查找数据流错误和一些异常现象的方法。

2. 可达性定义

严格地说，变量V的定义是修改V值的一个程序语句。如果语句 i 是V的一个定义，我们可以用 V_i 来表示这一定义。如果在控制流图中一路径并未对变量V定义，那么该路径就是变量V的明确定义。如果从节点 i 到节点 j 的入口(或出口)变量V有一明确定义路径，定义 V_i 便说成是达到了节点 j 的入口(或出口)。最后，我们说定义 V_i 被说成是“杀掉”了那些达到节点 i 的变量的所有其它定义(保留了其它变量的所有定义)。

现在让我们来看哪些定义达到了程序的每一节点。对于任一节点，我们用 S_i 和 t_i 分别表示达到 i 的入口和出口的定义集合。设 p_i 是节点 i 保存的所有定义的集合，C_i 是节点 i 中生成的定义集合。

对于每个节点 i，这些集合满足关系:

$$t_i = (S_i \cap p_i) \cup C_i \tag{5-1}$$

很明显，达到节点 i 顶点的定义集合是达到其前趋出口的定义集合的并。若用 x_i 表示节点 i 前趋的集合，则可将此条件表示为:

$$S_i = \bigcup_{j \in x_i} t_j \tag{5-2}$$

现在把 t_i 和 S_i 两式合并，得到:

$$S_i = \bigcup_{j \in i} ((S_j \cap p_j) \cup C_j) \tag{5-3}$$

或

$$S_i = \bigcup_{j \in x_i} (S_j \cap p_j) \cup d_i \tag{5-4}$$

其中

$$d_i = \bigcup_{j \in x_i} C_j$$

若是一个具有 n 个节点的图，我们便有这样的一个联立方程组(i 取值 1, 2,…, n)。正如 Hecht 所指出的，这样的方程组不只有一组解。但总有一个最小解，它给出要求达

到的定义。

这一方程组可用循环迭代算法求解。在该算法中，一开始规定 $S_i = d_i$ ($i = 1, 2, \cdots, n$)。然后依次对 $S_1, S_2, \cdots, S_n$ 求解方程(5-4)。每次令等号右端的 S_i 总取新值，当 S 的值不再改变时，就得到了所求的解。这一算法如图 5.6 所示。

```
begin
    for i:=1 Step 1 until n do Si:=di;
    CHANGE:= true ;
    while CHANGE do
    begin
        CHANGE:=false ;
        for i:=1 step 1 until n do
        begin
            TEMP:= ⋃(Si∩pi)∪di;
                   i∈xi
            if TEMP≠Si
            then
            begin
                CHANGE:=true ;
                Si:=TEMP
            end
        end
    end
end
```

图 5.6 循环迭代算法

可以看出，每个集合 S_i 可用一个二进制向量表示，程序的每个变量定义对应于向量分量的值。如果对应的定义属于 S_i，该分量取值为 1，否则取值为 0。若以类似的方式表示集合 p_i 和 d_i，在循环迭代算法中集合运算 ∪ 和 ∩ 可以分别用操作 OR (按位求和)和 AND (按位求积)来代替。

为说明这一过程，表 5.1 按方程 (5-4) 对图 5.5 给出了集合 d_i 的二进制向量表示。循环迭代算法的第一次完全迭代给出的 S 值如表 5.2 所示。二次迭代结果则在表 5.3 中给出。而第三次迭代它的值并没有什么变化。因此，表 5.3 是对程序每一节点的所有可达定义。

3. 引用未定义变量

根据可达定义表，可按下述方法找出对未定义变量的引用。对每一节点 i，我们依次考虑语句 i 引用的每一变量，如果对任何这样的变量 V，并没有 V 的定义达到 i，那么程序含有一个错误。

在上面的例中，我们考虑节点 2，根据引用变量表看出，这个语句引用变量 W。又从表 5.3 发现，W的定义不可能达到节点 2，因而程序中必定含有一个错误。这属于前述第一类错误(转向并不存在的标号)。

不过某些变量定义 V_i 达到节点 j 并不意味着在语句 j 执行前 V 总是被定义了，除非语句 i 总是在语句 j 之前执行。由于这一原因，上述测试方法并非必定能发现所有对未定义变量的引用。例如，在前面我们程序的例中，它就找不到第二类错误。然而，这类

表 5.1

	V_7	W_8	X_1	X_2	Y_1	Y_5	Z_1	Z_5	Z_9	Z_{10}
2	0	0	1	0	1	0	1	0	0	0
3	0	0	0	1	0	0	0	0	1	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0	1	0	0
8	1	0	0	0	0	0	0	0	0	0
9	0	1	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	1

表 5.2

	V_7	W_8	X_1	X_2	Y_1	Y_5	Z_1	Z_6	Z_9	Z_{10}
2	0	0	1	0	1	0	1	0	0	0
3	0	1	0	1	1	0	1	0	1	0
4	0	1	0	1	1	0	1	0	1	0
5	0	1	0	1	1	0	1	0	1	0
6	0	1	0	1	1	0	1	0	1	0
7	0	1	0	1	1	1	1	1	1	0
8	1	1	0	1	1	1	1	1	1	0
9	1	1	0	1	1	1	1	1	1	0
10	0	1	0	1	1	0	1	0	1	0
11	0	1	0	1	1	0	0	0	0	1

表 5.3

	V_7	W_8	X_1 X_2	Y_1 Y_5	Z_1 Z_6 Z_9 Z_{10}
2	0	[0]	[1 0]	1 0	1 0 0 0
3	1	1	[0 1]	[1 1]	1 0 1 0
4	1	1	0 1	[1 1]	[1 0 1 0]
5	[1]	1	0 1	[1 1]	1 0 1 0
6	[1]	1	0 1	1 1	[1 0 1 0]
7	1	1	[0 1]	1 1	1 1 1 0
8	1	1	0 1	[1 1]	1 1 1 0
9	[1]	1	0 1	1 1	1 1 1 0
10	1	1	0 1	1 1	[1 0 1 0]
11	1	1	0 1	1 1	[0 0 0 1]

错误可以用迭代法解其形状很像方程(5-4)的另一方程组。

4. 未曾使用的定义

可以用下面的方法找出未曾使用的定义。对于每一变量定义 V_i，我们依次考虑由

V_i 达到的每个程序节点 j：如果没有引用变量V的相应语句，则程序中含有一个异常。

例如，在我们的例中，考虑定义 Z_6 时，根据表 5.3 发现，这一定义达到节点 7，8 和 9。然而，由引用变量表我们看出，并没有对应的语句引用变量 Z，这属于第三类错误。

当然，也不可能假定，沿着能使变量V的定义达到V被使用的节点的任何路径，实际上，在程序执行中都可得到循行。因此，上述测试方法并不一定能找出所有的冗余定义。

5. 数据流分析应用的其它方面

在此只是介绍一个最基本的数据流分析方法。例如，上面谈到的关于方法与结构分析相结合，为查找引用未定义变量及未使用的定义提供了更强有力的测试手段。除去这里给出的过程内数据流分析以外，也可以作过程间的数据流分析。基于这些原则的错误查找系统已经开发出来，这些系统甚至能在已经顺利地运行几个月以后，一直被认为是正确的程序中发现有拼写的错误。

在优化的编译系统中，数据流分析除去用于前已说明的以外，还用于多种目的。一个常数传播的例子是：如果变量V的所有定义(该定义达到引用V的一个特定语句)都把同一已知常数赋给该变量，对V的引用便可用这一常数所代替。这里是一个普通的例子。程序段：

```
a: = 4
b: = a +
     :
c: = 3*(a + b)
```

可以用下列程序段代替：

```
a: = 4
b: = 5
     :
c: = 27
```

常数传播除去能节省执行时间外，还能提高程序正文的清晰性，确认系统可以表明进行这种修改的可能性。

另一个例子是找出循环内的不变定义。这种定义并不引用其值在执行循环时可改变的任何变量。在优化的编译系统中查找不变定义是很重要的，因为它可使得将这一定义从循环中移出，放在循环前面或是放在循环后面，从而减少它的执行次数。在程序确认中，我们也对不变定义感兴趣，因为它会提醒我们注意粗心的程序设计。

三、信息流分析

直到目前信息流分析主要用在验证程序变量间信息的传输遵循保密要求。然而，近来发现可以导出程序的信息流关系，这就为软件开发和确认提供了十分有益的工具。

为了说明信息流分析的性质，以下给出整除算法作为例子。图 5.7 是这一算法的程序。图 5.8 是三个关系的表。其中第一个表(图 5.8(a))给出每一语句执行时所用到其输入值的变量。例如，从图 5.7 的算法很明显地看出，M的输入值在语句 2 中得到直接使

```
语句号      begin
  1           Q: = 0;
  2           R: = M;
  3           while R>=N do
                begin
  4               Q: = Q + 1;
  5               R: = R - N
                end
            end
```

图 5.7　整除算法

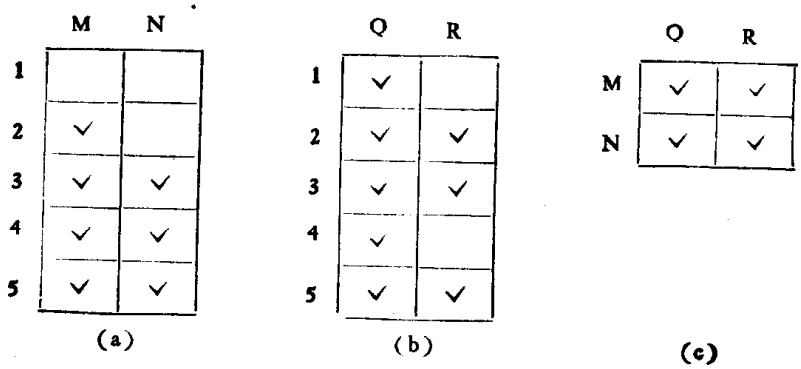


图 5.8　整除算法中输入值、语句与输出值的关系

用，由于这一语句将M的值传送给 R，M的初始值也间接地用于语句 3 和语句 5。而且，语句 3 中表达式 R>=N 的值决定了语句 4 的重复执行次数，也即对Q多少次重复赋值，即是说M的值也间接地用于语句 4。

第二个关系（图 5.8 (b)）给出了其执行可能直接或间接影响输出变量终值的一些语句。可以看出，所有语句都可能影响到商Q的值。而语句 1 和语句 4 并未关系到余数R的值。

最后的关系表明了哪个输入值可能直接或间接地影响到输出值。

针对结构良好的程序快速算法（只需多项式时间）已经开发出来可用以建立这些关系，这在程序的确认中是非常有用的。例如，第一个关系能够表明对未定义变量的所有可能的引用。第二个关系在查找错误中也是有用的，比如假定某个变量的计算值在使用以前被错误地改写了，这可能因为有并不影响任何输出值的语句而被发现。在程序的任何指定点查出其执行可能影响某一变量值的语句，这在程序排错和程序验证中都是很有用的。第三个输入输出关系还提供一种检查，看看每个输出值是否由相关的输入值，而不是其它值导出。

5.2　逻辑覆盖

结构测试是依据被测程序的逻辑结构设计测试用例，驱动被测程序运行完成的测试。结构测试中的一个重要问题是，测试进行到什么地步就达到要求，可以结束测试了。这就是说需要给出结构测试的覆盖准则。

本节将扼要地介绍几种常用的逻辑覆盖测试方法：语句覆盖、判定覆盖、条件覆盖、判定-条件覆盖及路径覆盖。进而给出两个结构测试的覆盖准则。

一、几种常用的逻辑覆盖测试方法

以下给出的几种逻辑覆盖测试方法都是从各自不同的方面出发，为设计测试用例提出依据的。为方便讨论，我们将结合一个小程序段加以说明：

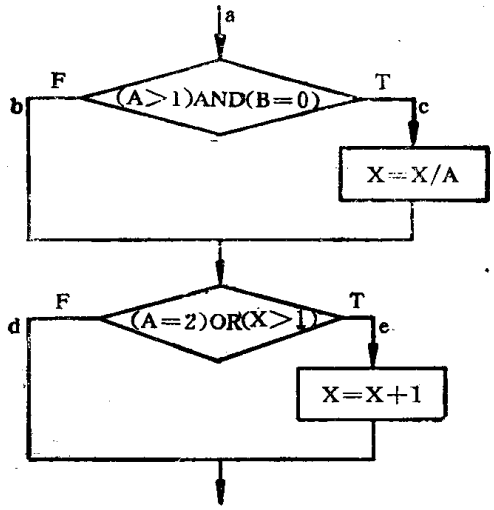


图 5.9　被测程序段流程图

```
IF((A > 1)AND(B = 0)) THEN
        X = X/A
IF((A = 2) OR(X > 1) THEN
        X = X + 1
```

其中“AND”和“OR”是两个逻辑运算符。图 5.9 给出了它的流程图。a、b、c、d 和 e 是控制流上的若干程序点。

1. 语句覆盖

语句覆盖的含意是，在测试时，首先设计若干个测试用例，然后运行被测程序，使程序中的每个可执行语句至少执行一次。这里所谓“若干个”，自然是越少越好。

在上述程序段中，我们如果选用的测试用例是：

```
A = 2
B = 0 }..... CASE1
X = 3
```

则程序按路径 ace 执行。这样该程序段的 4 个语句均得到执行，从而作到了语句覆盖。但如果选用的测试用例是：

```
A = 2
B = 1 }..... CASE2
X = 3
```

程序按路径 abe 执行，便未能达到语句覆盖。

从程序中每个语句都得到执行这一点来看，语句覆盖的方法似乎能够比较全面地检验每一个语句。但它也绝不是完美无缺的。假如这一程序段中两个判断的逻辑运算有问题，例如，第一个判断的运算符“AND”错成运算符“OR”或是第二个判断中的运算符“OR”错成了运算符“AND”。这时仍使用上述前一个测试用例 CASE1，程序仍将按路径 ace 执行。这说明虽然也作到了语句覆盖，却发现不了判断中逻辑运算的错误。

此外，我们还可以很容易地找出已经满足了语句覆盖，却仍然存在错误的例子。如有一程序段：

```
    ⋮
IF(I≥0)
 THEN I = J
    ⋮
```

如果错写成：

```
    ⋮
IF (I>0)
  THEN I = J
    ⋮
```

假定给出的测试数据确使执行该程序段时 I 的值大于 0，则 I 被赋予 J 的值。这样虽然作到了语句覆盖，然而掩盖了其中的错误。

实际上，和后面介绍的其它几种逻辑覆盖比较起来，语句覆盖是比较弱的覆盖原则。作到了语句覆盖可能给人们一种心理的满足，以为每个语句都经历过，似乎可以放心了。其实这仍然是不十分可靠的。语句覆盖在测试被测程序中，除去对检查不可执行语句有一定作用外，并没有排除被测程序包含错误的风险。必须看到，被测程序并非语句的无序堆积，语句之间的确存在着许多有机的联系。

2. 判定覆盖

按判定覆盖准则进行测试是指，设计若干测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次，即判断的真假值均曾被满足。判定覆盖又称为分支覆盖。

仍以上述程序段为例，若选用的两组测试用例是：

```
A = 2
B = 0 ..... CASE 1
X = 3

A = 1
B = 0 ..... CASE 3
X = 1
```

则可分别执行路径 ace 和 abd，从而使两个判断的 4 个分支 c、e 和 b、d 分别得到覆盖。

当然，我们也可以选用另外两组测试用例：

```
A = 3
B = 0 ..... CASE 4
X = 3

A = 2
B = 1 ..... CASE 5
X = 1
```

分别路径 acd 及 abe，同样也可覆盖 4 个分支。

我们注意到，上述两组测试用例不仅满足了判定覆盖，同时还做到了语句覆盖。从这一点看似乎判定覆盖比语句覆盖更强一些，但让我们设想，在此程序段中的第 2 个判断条件 X > 1 如果错写成 X < 1，使用上述测试用例 CASE5，照样能按原路径执行 (abe)，而不影响结果。这个事实说明，只作到判定覆盖仍无法确定判断内部条件的错误。因此，需要有更强的逻辑覆盖准则去检验判断内的条件。

以上仅考虑了两出口的判断，我们还应把判定覆盖准则扩充到多出口判断(如 CASE 语句)的情况。

3. 条件覆盖

条件覆盖是指，设计若干测试用例，执行被测程序以后，要使每个判断中每个条件的可能取值至少满足一次。

在上述程序段中，第一个判断应考虑到：

$A>1$ 取真值，记为 T_1

$A>1$ 取假值，即 $A\leqslant 1$，记为 $\bar{T}_1$

$B=0$ 取真值，记为 T_2

$B=0$ 取假值，即 $B\neq 0$，记为 $\bar{T}_2$

第 2 个判断应考虑到：

$A=2$ 取真值，记为 T_3

$A=2$ 取假值，即 $A\neq 2$，记为 $\bar{T}_3$

$X>1$ 取真值，记为 T_4

$X>1$ 取假值，即 $X\leqslant 1$，记为 $\bar{T}_4$

我们给出 3 个测试用例：CASE6，CASE7，CASE8，执行该程序段所走路径及覆盖条件是：

测试用例	A B X	所走路径	覆盖条件
CASE 6	2 0 3	a c e	T_1, T_2, T_3, T_4
CASE 7	1 0 1	a b d	$\bar{T}_1, T_2, \bar{T}_3, \bar{T}_4$
CASE 8	2 1 1	a b e	$T_1, \bar{T}_2, T_3, \bar{T}_4$

从这个表中可以看到，3 个测试用例把 4 个条件的 8 种情况均作了覆盖。

进一步分析上表，覆盖了 4 个条件的 8 种情况的同时，把两个判断的 4 个分支 b、c、d 和 e 似乎也被覆盖。这样我们是否可以说，做到了条件覆盖，也就必然实现了判定覆盖呢？让我们来分析另一情况，假定选用两组测试用例是 CASE 9 和 CASE 8，执行程序段的覆盖情况是：

测试用例	A B X	所走路径	覆盖分支	覆盖条件
CASE 9	1 0 3	abe	be	$\bar{T}_1, T_2, \bar{T}_3, T_4$
CASE 8	2 1 1	abe	be	$T_1, \bar{T}_2, T_3, \bar{T}_4$

这一覆盖情况表明，覆盖了条件的测试用例不一定覆盖了分支。事实上，它只覆盖了 4 个分支中的两个。为解决这一矛盾，需要对条件和分支兼顾。

4. 判定-条件覆盖

判定-条件覆盖要求设计足够的测试用例，使得判断中每个条件的所有可能至少出现一次，并且每个判断本身的判定结果也至少出现一次。

例中两个判断各包含两个条件，这 4 个条件在两个判断中可能有 8 种组合，它们是：

① $A>1$，$B=0$ 记为 T_1, T_2

② $A>1$，$B\neq 0$ 记为 $T_1, \bar{T}_2$

③ $A \leqslant 1$, $B = 0$ 记为 $\bar{T}_1$, T_2

④ $A \leqslant 1$, $B \neq 0$ 记为 $\bar{T}_1$, $\bar{T}_2$

⑤ $A = 2$, $X > 1$ 记为 T_3, T_4

⑥ $A = 2$, $X \leqslant 1$ 记为 T_3, $\bar{T}_4$

⑦ $A \neq 2$, $X > 1$ 记为 $\bar{T}_3$, T_4

⑧ $A \neq 2$, $X \leqslant 1$ 记为 $\bar{T}_3$, $\bar{T}_4$

这里设计了 4 个测试用例，用以覆盖上述 8 种条件组合：

测试用例	A	B	X	覆盖组合号	所走路径	覆盖条件
CASE 1	2	0	3	① ⑤	ace	T_1, T_2, T_3, T_4
CASE 8	2	1	1	② ⑥	abe	$T_1, \bar{T}_2, T_3, \bar{T}_4$
CASE 9	1	0	3	③ ⑦	abe	$\bar{T}_1, T_2, \bar{T}_3, T_4$
CASE 10	1	1	1	④ ⑧	abd	$\bar{T}_1, \bar{T}_2, \bar{T}_3, \bar{T}_4$

我们注意到，这一程序段共有四条路径。以上 4 个测试用例固然覆盖了条件组合，同时也覆盖了 4 个分支，但仅覆盖了 3 条路径，却漏掉了路径 acd。前面讨论的多种覆盖准则，有的虽提到了所走路径问题，但尚未涉及到路径的覆盖，而路径能否全面覆盖在软件测试中是个重要问题，因为程序要取得正确的结果，就必须消除遇到的各种障碍，沿着特定的路径顺利执行。如果程序中的每一条路径都得到考验，才能说程序受到了全面检验。

5. 路径覆盖

按路径覆盖要求进行测试是指，设计足够多测试用例，要求覆盖程序中所有可能的路径。

针对例中的 4 条可能路径(图 5.10)

ace 记为 L_1

abd 记为 L_2

abe 记为 L_3

acd 记为 L_4

我们给出 4 个测试用例：CASE 1，CASE 7，CASE 8 和 CASE 11，使其分别覆盖这 4 条径：

测试用例	A	B	X	覆盖路径
CASE 1	2	0	3	ace (L_1)
CASE 7	1	0	1	abd (L_2)
CASE 8	2	1	1	abe (L_3)
CASE 11	3	0	1	acd (L_4)

这里所用的程序段非常简短，也只有 4 条路径。但在实际问题中，一个不太复杂的程序，其路径数都是一个庞大的数字。在前面图 5.6 所示的程序竟有 5^{20} 条路径，要在测试中覆盖这样多的路径是无法实现的。为解决这一难题只得把覆盖的路径数压缩到一定限度内，例如，程序中的循环体只执行了一次。

其实，即使对于路径数很有限的程序已经作到了路径覆盖，仍然不能保证被测程序的正确性。例如，在上述语句覆盖一段最后给出的程序段中出现的错误也不是路径覆盖可

以发现的。

由此看出，各种结构测试方法都不能保证程序的正确性。这一严酷的事实对热心测试的程序人员似乎是一个严重的打击。但要记住，测试的目的并非要证明程序的正确性，而是要尽可能找出程序中的错误。确实并不存在一种十全十美的测试方法，能够发现所有的错误。想要撒下几网把湖中的鱼全都捕上来是作不到的。这又涉及到本书第一章里给出的关于软件测试局限性的讨论。

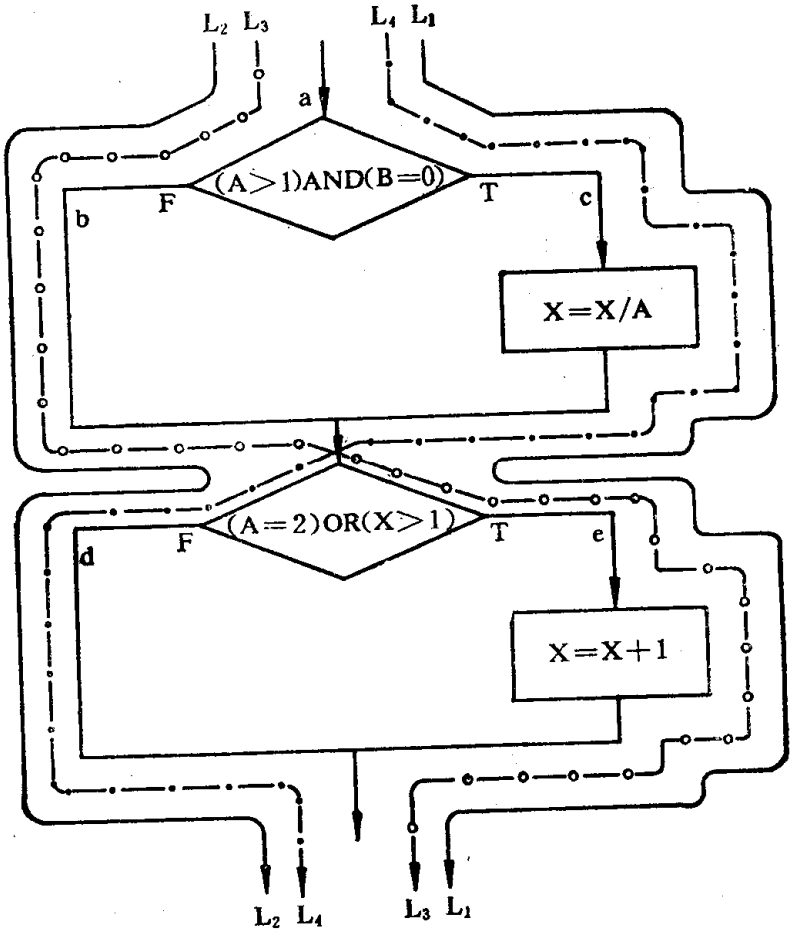


图 5.10　被测程序段的路径

二、最少测试用例数计算

为实现测试的逻辑覆盖，必须设计足够多的测试用例，并使用这些测试用例执行被测程序，实施测试。我们关心的是，对某个具体程序来说，至少要设计多少测试用例。这里提供一种估算最少测试用例数的方法。

我们知道，结构化程序是由 3 种基本控制结构组成。这 3 种基本控制结构就是：

顺序型——构成串行操作；

选择型——构成分支操作；

重复型——构成循环操作。

为了把问题化简，避免出现测试用例极多的组合爆炸，把构成循环操作的重复型结构用选择结构代替。也就是说，并不指望测试循环体所有的重复执行，而是只对循环体检验一

次。这样，任一循环便改造成进入循环体或不进入循环体的分支操作了。

图 5.11 给出了类似于流程图的 N-S 图表示的基本控制结构(图中 A、B、C、D、S 均表示要执行的操作，P 是可取真假值的谓词，Y 表真值，N表假值)。其中图 5.11 (c) 和图 5.11 (d) 两种重复型结构代表了两种循环。在作了如上简化循环的假设以后，对于一般的程序控制流，我们只考虑选择型结构。事实上它已能体现了顺序型和重复型结构了。

例如，图 5.12 表达了两个顺序执行的分支结构。两个分支谓词 P_1 和 P_2 取不同值时，将分别执行 a 或 b 及 c 或 d 操作。显然，要测试这个小程序，需要至少提供 4 个测试用例才能作到逻辑覆盖。使得 ac、ad、bc 及 bd 操作均得到检验。其实，这里的 4 是

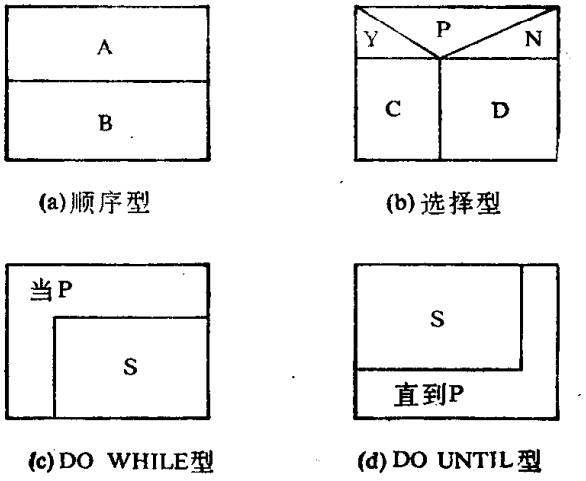


图 5.11 N-S 图表示的基本控制结构

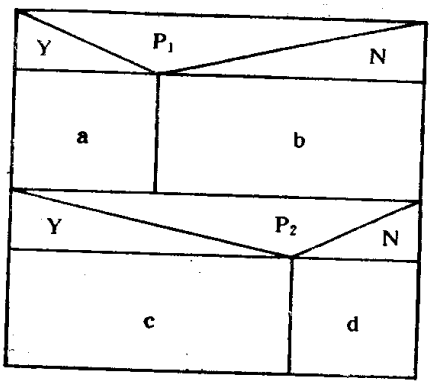


图 5.12 两个串行的分枝结构的 N-S 图

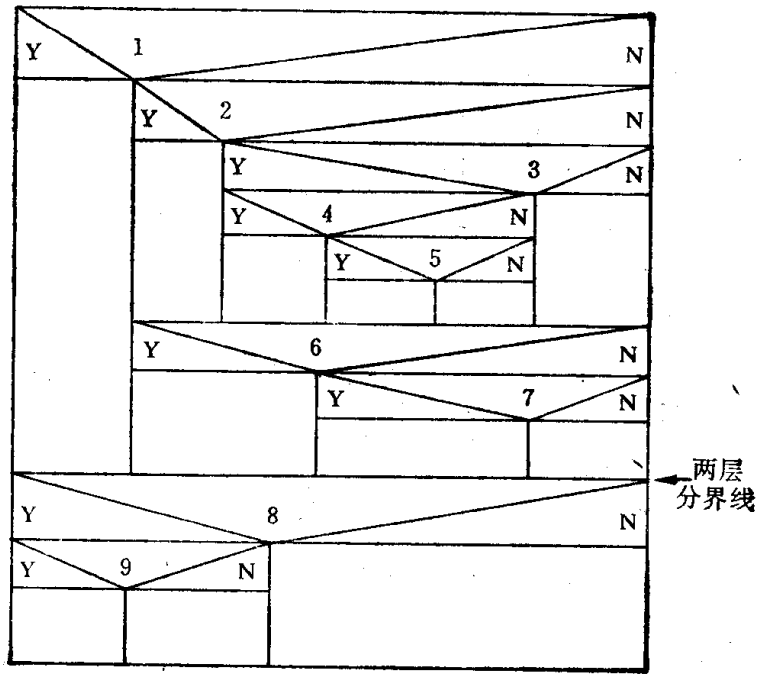


图 5.13 计算最少测试用例数实例

图中第 1 个分支谓词引出的两个操作，及第 2 个分支谓词引出的两个操作组合起来而得到的，即 $2 \times 2 = 4$。并且，这里的 2 是由于两个并列的操作，$1 + 1 = 2$ 而得到的。

对于一般的、更为复杂的问题，估算最少测试用例数的原则也是同样的。现以图 5.13 表示的程序为例。该程序中共有 9 个分支谓词，尽管这些分支结构交错起来似乎十分复杂，很难一眼看出应至少需要多少个测试用例，但如果仍用上面的方法，也是很容易解决的。我们注意到该图可分上下两层：分支谓词 1 的操作域是上层，分支谓词 8 的操作域是下层。这两层正像前面简单例中的 P_1 和 P_2 的关系一样。只要分别得到两层的测试用例个数，再将其相乘即得总的测试用例数。这里需要首先考虑较为复杂的上层结构。谓词 1 不满足时要作的操作又可进一步分解为两层，这就是图 5.14 中的子图 (a) 和 (b)。它们所需测试用例个数分别为 $1 + 1 + 1 + 1 + 1 = 5$ 及 $1 + 1 + 1 = 3$。因而两层组合，得到 $5 \times 3 = 15$。于是整个程序结构上层所需测试用例数为 $1 + 15 = 16$。而下层十分显然为 3。故最后得到整个程序所需测试用例数至少为 $16 \times 3 = 48$。

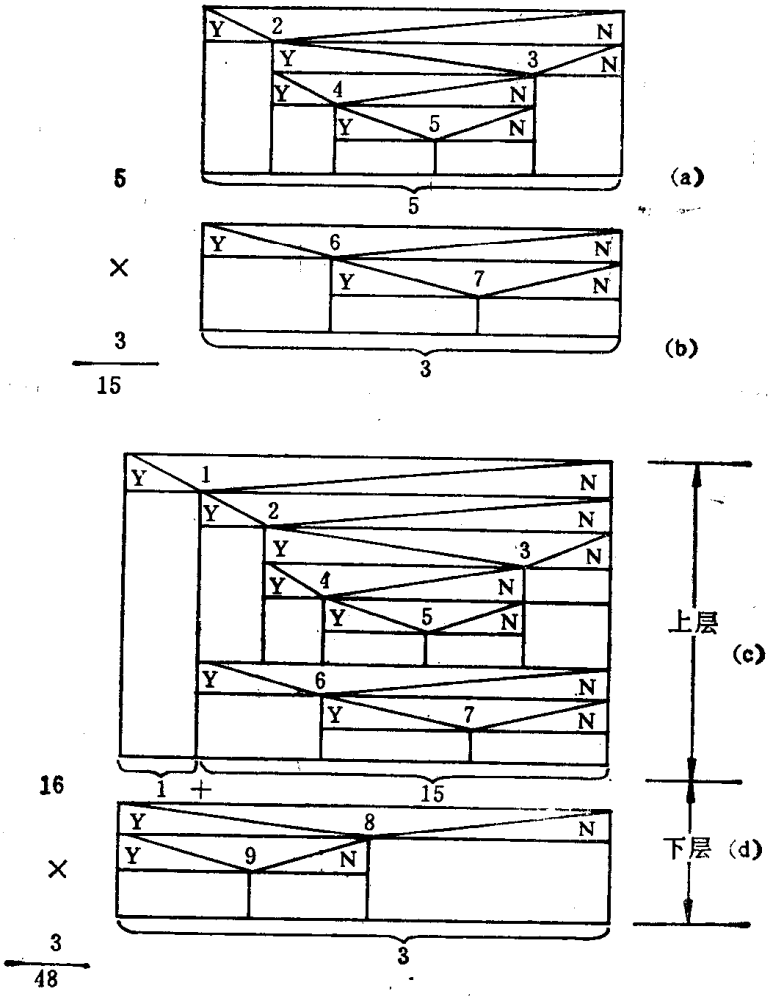


图 5.14　最少测试用例数计算

三、测试覆盖准则

1. FOSTER 的 ESTCA 覆盖准则

前面介绍的逻辑覆盖其出发点似乎是合理的。所谓“覆盖”，就是想要作到全面，而无遗漏。但事实表明，它并不能真的作到无遗漏。甚至象前面提到的将程序段：

```
    ⋮
IF(I≥0)
  THEN I = J
    ⋮
```

错写成：

```
    ⋮
IF (I>0)
  THEN I = J
    ⋮
```

这样的小问题都无能为力。

我们分析出现这一情况的原因在于，错误区域仅仅在 I = 0 这个点上，即仅当 I 取 0 时，测试才能发现错误。它的确是在我们力图全面覆盖来查找错误的测试“网”上钻了空子，并且恰恰在容易发生问题的条件判断那里未被发现。面对这类情况我们应该从中吸取的教训是测试工作要有重点，要多针对容易发生问题的地方设计测试用例。

K. A. Foster 从测试工作实践的教训出发，吸收了计算机硬件的测试原理，提出了一种经验型的测试覆盖准则，较好地解决了上述问题。

Foster 的经验型覆盖准则是从硬件的早期测试方法中得到启发的。我们知道，硬件测试中，对每一个门电路的输入、输出测试都是有额定标准的。通常，电路中一个门的错误常常是“输出总是 0”，或是“输出总是 1”。与硬件测试中的这一情况类似，我们常常要重视程序中谓词的取值，但实际上它可能比硬件测试更加复杂。Foster 通过大量的实验确定了程序中谓词最容易出错的部分，得出了一套错误敏感测试用例分析 ESTCA (Error Sensitive Test Cases Analysis) 规则。事实上，规则十分简单：

[规则 1]对于 A rel B (rel 可以是<，= 和 >)型的分支谓词，应适当地选择 A 与 B 的值，使得测试执行到该分支语句时，A < B，A = B 和 A > B 的情况分别出现一次。

[规则 2]对于 A rel_1 c (rel_1 可以是 < 或是 > ，A是变量，C是常量)型的分支谓词，当 rel_1 为 < 时，应适当地选择 A 的值，使：

$$A = C - M$$

(M 是距 C 最小的机器容许正数，若 A 和 C 均为整型时，M = 1)。同样，当 rel_1 为 > 时，应适当地选择 A，使：

$$A = C + M$$

[规则 3]对外部输入变量赋值，使其在每一测试用例中均有不同的值与符号，并与同一组测试用例中其它变量的值与符号不一致。

显然，上述规则 1 是为了检测 rel 的错误，规则 2 是为了检测“差一”之类的错误（如本应是“IF A > 1”而错成 “IF A > 0”），而规则 3 则是为了检测程序语句中的错误（如应引用一变量而错成引用一常量）。

上述三规则并不是完备的，但在普通程序的测试中确是有效的。原因在于规则本身针对着程序编写人员容易发生的错误，或是围绕着发生错误的频繁区域，从而提高了发现

错误的命中率。

根据这里提供的规则来检验上述小程序段错误。应用规则1，对它测试时，应选取I的值为0，使I = 0的情况出现一次。这样一来就立即找出了隐藏的错误。

当然，ESTCA规则也有很多缺陷。一方面是有时不容易找到输入数据，使得规则所指的变量值满足要求。Foster把这一问题归结为著名的"旅行商问题"，同时提出了一个具体的解决办法。另一方面是仍有很多缺陷发现不了。对于查找错误的广度问题在变异测试中得到较好的解决。

2. Woodward等人的层次LCSAJ覆盖准则

Woodward等人曾经指出结构覆盖的一些准则，如分支覆盖或路径覆盖，都不足以保证测试数据的有效性。为此，他们提出了一种层次LCSAJ覆盖准则。

LCSAJ (Linear Code Sequence and Jump) 意思是线性代码序列与跳转。一个LCSAJ是一组顺序执行的代码，以控制流跳转为其结束点。它不同于判断-判断路径。判断-判断路径是根据程序有向图决定的。一个判断-判断路径是指两个判断之间的路径，但其中不再有判断。程序的入口、出口和分支结点都可以是判断点。而LCSAJ的起点是根据程序本身决定的。它的起点是程序第一行或转移语句的入口点，或是控制流可以跳达的点。几个首尾相接，且第一个LCSAJ起点为程序起点，最后一个LCSAJ终点为程序终点的LCSAJ串就组成了程序的一条路径。一条程序路径可能是由两个、三个或多个LCSAJ组成的。基于LCSAJ与路径的这一关系，Woodward提出了LCSAJ覆盖准则。这是一个分层的覆盖准则：

[第一层]：语句覆盖。

[第二层]：分支覆盖。

[第三层]：LCSAJ覆盖。即程序中的每一个LCSAJ都至少在测试中经历过一次。

[第四层]：两两LCSAJ覆盖。即程序中每两个首尾相连的LCSAJ组合起来在测试中都要经历一次。

⋮

[第n+2层]：每n个首尾相连的LCSAJ组合在测试中都要经历一次。

他们说明了，越是高层的覆盖准则越难满足。

在实施测试时，若要实现上述的Woodward层次LCSAJ覆盖，需要产生被测程序的所有LCSAJ。

5.3 域测试

域测试 (Domain Testing) 是一种基于程序结构的测试方法。最早为L. J. White和E. K. Cohen于1977年提出，以后发展成为一个模块测试的有效方法。但是由于该方法使用时有一些限制条件，并且还涉及到多维空间的概念，不易被人们接受，也就在一定程度上影响了它的实用性和推广。

一、方法简介

Howden 曾对程序中出现的错误进行分类。他将程序错误分为域错误、计算型错误和丢失路径错误三种。这是相对于执行程序的路径来说的。我们知道，每条执行路径对应于输入域的一类情况，是程序的一个子计算。如果程序的控制流有错误，对于某一特定的输入可能执行的是一条错误路径，这种错误称为路径错误，也叫做域错误。如果对于特定输入执行的是正确路径，但由于赋值语句的错误致使输出结果不正确，则称此为计算型错误。另外一类错误是丢失路径错误。它是由于程序中某处少了一个判定谓词而引起的。域测试主要是针对域错误进行的程序测试。

为了域测试的方便，White 和 Cohen 对被测程序规定了一些限制，这些限制是：

① 程序中不出现数组。

② 程序中不含有子函数或子例程。

③ 程序中没有输入和输出错误。

④ 程序的分支谓词是简单谓词，即它不含有布尔运算符 AND 和 OR。

⑤ 程序分支谓词是线性的。

⑥ 程序输入域是连续的，而不是离散的。

⑦ 相邻的两个域(路径)上的计算是不相同的。

事实上，规定这些限制只是为了简化分析的目的，并不是排除域测试无法克服的一些情况。

域测试的"域"指的是程序的输入空间。域测试方法基于对输入空间的分析。自然，任何一个被测程序都有一个输入空间。测试的理想结果就是检验输入空间中的每一个输入元素是否都产生正确的结果。而输入空间又可分为不同的子空间，每一子空间对应一种不同的计算。在考察被测程序的结构以后，我们就会发现，子空间的划分是由程序中分支语句中的谓词决定的。输入空间的一个元素，经过程序中某些特定语句的执行而结束(当然也有可能出现无限循环而无出口)，那都是满足了这些特定语句被执行所要求的条件的。

域测试正是在分析输入域的基础上，选择适当的测试点以后进行测试的。

二、输入域结构

这里结合一个短的程序(图 5.15 所示)说明输入域的分割。

程序中包含三个条件语句的谓词：

谓词 1　　I <= J + 1　　(1)

谓词 2　　K >= I + 1　　(2)

谓词 3　　I = 5　　(3)

该程序的输入变量只是 I 和 J。其输入空间只是由 I 为横轴，J 为纵轴的平面。这一平面域的大小由 I 和 J 的最大和最小取值决定。这里假定，I 在−8 到+8 区间，J 在−5 到+5 之间。图 5.16 给出了输入空间的划分。其中带有指向左上箭头的斜线代表了谓词 1，即 $I \leqslant J+1$。斜线左上方是满足谓词 1 的域。我们注意到，谓词 2 是在对 K 赋值后执

```
READ I,J;
IF I< = J + 1
  THEN K = I + J - 1;
  ELSE  K = 2*I + 1;
IF K> = I + 1
  THEN L = I + 1;
  ELSE  L = J - 1;
IF I = 5
  THEN M = 2*L - K;
  ELSE  M = L + 2*K - 1;
WRITE M
```

图 5.15　说明输入域的程序实例

行的,因而,在 K = I + J − 1 时,谓词 2 表现为:

$$J \geqslant 2 \tag{2-1}$$

在 K = 2*I + 1 时,谓词 2 表现为:

$$I \geqslant 0 \tag{2-2}$$

这在图 5.16 中表现为带有向上箭头的水平线段和带有向右箭头的垂直线段。谓词 3 则很清楚在图中表现为 I = 5 的垂直线段。

这样就完成了输入域的分割。图中共引出了 6 个线段,这是一条水平线段、一条斜线和两条垂线相交分割而成的: 4 个线段关系到不等式,两个线段关系到等式。

三、测试点的选择

有两类测试点可供选择,一类称作 ON 点,这类测试点位于域的边界上;另一类称作 OFF 点,它离边界有一小距离 ε,并在被测域之外。图 5.17 给出了测试点的选择方案。其中 AB 是测试输入空间某一子空间的一条边界(由被测程序决定),PQ 是程序

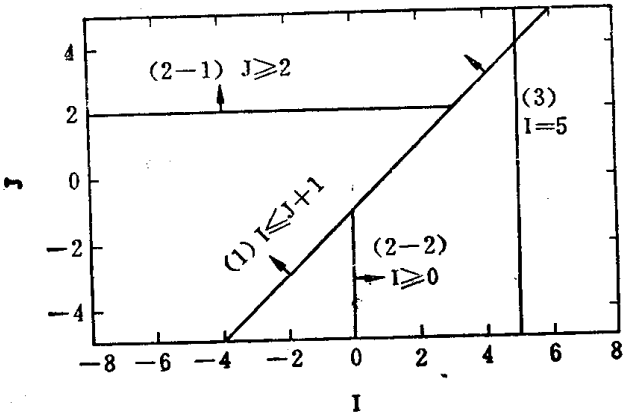


图 5.16　输入空间的划分

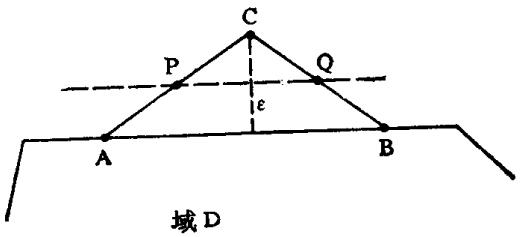


图 5.17　线性边界的测试点选择

正确时实际边界的位置。A、B 是两个 ON 点,C 点是一个 OFF 点,离 AB 边界的距离是 ε。

如果将 ON 点与 OFF 点交替选择,即让 OFF 点在两 ON 点所决定直线上的投影在两 ON 点之间,则可较好地测出由于边界错误而导致的域错误。这就是域测试的基本思想。

从图 5.17 可以看出，如果 A、B、C 三点的测试结果出错，则显然有一个错误存在。反之，若 A、B、C 三点均产生正确的输出值，则这一边界可能是正确边界，也可能正确边界介于C点与线段 AB 之间。这是因为 A、B 在一个域中，而C在与其相邻的域中，因而边界在它们中间。这样，如果我们将 ε 取得尽可能小，就应尽可能地认定边界 AB 的正确性。

我们把这种错误边界离正确边界相差不大的情况称为边界位移。可以将边界位移分为三类，如图 5.18 所示。该图（a）中边界位移使得域 D_1 减小，测试点 A、B 将给出正确结果。（b）中边界位移使得 D_1 增大，测试点C将给出正确结果，而测试点 A、B 将给出错误结果。在子图（c）中，测试点 A、C将给出正确结果，而测试点B将给出错误

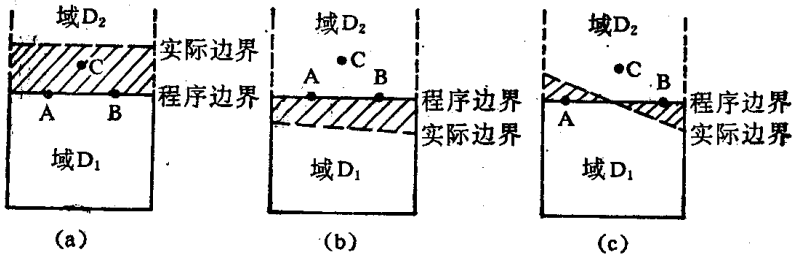


图 5.18　边界位移的三种基本类型

结果。但需注意，判断测试点是否给出正确结果的依据是此测试点是否属于其应属的域中。这表明，ON-OFF-ON 这种交错取点法对边界位移错误是相当有效的。这样，域测试的工作就可归结到 ON-OFF-ON 点的选择了。

同样，程序谓词中运算符的错误也能通过这种域测试方法检测出来。也就是说，常见的错误，如该用“≥”时，错成“≤”；该用“<”处，错成“≤”等等错误都能较容易地查出。

因为一个边界需要三个测试点，若一个域D有P个边界，那么最多需要 3* P 个测试点。但是，相邻的两个边界的交点可以用来作两个边界共同的 ON 点。这样，可以有效地降低所需测试点的个数。

以上是针对二维域情况讨论的。当一程序谓词可归结为多个输入变量的关系时，就需用到多维超平面的概念了。域测试的基本思想仍然适用，只不过对于 n 维情况，我们需要找出 n 个线性无关的 ON 点和一个 OFF 点来测试一边界的位移。通常可把超平面的极值点选作 ON 点，以保证其线性无关。

应当注意，以上讨论的边界都是由不等式谓词产生的，即 exp > 0，exp < 0，exp ≥ 0 以及 exp ≤ 0 之类的谓词。如果谓词形式是 exp = 0，则需要两个 OFF 点，两个 ON 点。两个 OFF 点分别在两个 ON 点连线的两侧，才能确保测出边界的位移。

域测试的步骤可归结为：

① 根据各分支谓词，画出域分割图。

② 对每一个域的每一个边界用 ON-OFF-ON 原则选取测试点进行测试。

③ 在域内取一些点进行测试。

四、划分分析

域测试的一种变形是 D. J. Richardson 和 L. A. Clarke 提出的划分分析 (Partition Analysis)。这是程序测试与程序验证结合的方法。但其基本思想仍然是从划分输入空间出发的。首先，一个过程或函数的功能可以用：

$$F:\quad X \to Z$$

来表示。其中，F 是函数，X 是其定义域，Z 是其值域。F 是由子函数 F_1，F_2，$\cdots$，F_L 组成的。假设实现 F 的程序是 P，F 的形式化说明是 S。我们用 D(X) 表示 X 的定义域，C(X) 表示在 X 定义域上的计算，则可将函数本身，函数的实现和说明表示为：

$F = \{F_1, F_2, \cdots, F_L\}$

$P = \{(D[P_1], C[P_1]), \cdots, (D[P_n], C[P_n])\}$ $P: X \to Z$

$S = \{(D[S_1], C[S_1]), \cdots, (D[S_m], C[S_m])\}$ $S: X \to Z$

也就是说，实现时共有 n 个不同的输入子空间，在其上的计算互不相同。而在说明中有 m 个不同的子空间，在其上的计算也互不相同。实际上，上面分别是对函数、实现及其说明的划分。

在此基础上，可定义过程划分。如果实现的一个输入子空间 $D[P_j]$ 与说明的一个输入子空间 $D[S_i]$ 之交不空，就可将其称为过程的一个子域，我们定义其差异为 C_{ij}。其表示式为：

$$C_{ij} = C[S_i] \ominus C[P_j]$$

这里 '$\ominus$' 表示符号差。

如果 $D(S_i)$ 中的元素未被实现时，任一条路径处理用 D_{i0} 表示，其式子为：

$$D_{i0} = D[S_i] - D[P]$$

这里 '$-$' 表示集合差。同样可定义：

$$D_{0j} = D[P_j] - D[S]$$

这样便可进行过程划分，决定不同输入域上的计算。

过程划分为：

$$\{(D_{ij}, C_{ij}) \mid 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n, D[S_i] \cap D[P_j] \neq 0\}$$
$$\cup\{(D_{i0}, C_{i0}) \mid 1 \leqslant i \leqslant m, D[S_i] - D[P] \neq 0\}$$
$$\cup\{(D_{0j}, C_{0j}) \mid 1 \leqslant j \leqslant n, D[P_j] - D[S] \neq 0\}。$$
$$(C_{i0} = C[S_i], C_{0j} = C[P_j])$$

在每一个划分子域中，可以比较说明与实现的差，如果不一致，则此子域有错。也就是说，若 C_{ij}、C_{i0} 或 C_{0j} 不为零，则对应的域说明与现实不一致。

划分分析的过程是：

① 运用符号执行方法将说明与实现分别进行划分。

② 构造过程划分。

③ 用验证方法验证 $C_{ij} = 0$。

④ 用域测试方法测试一些点，并测试在验证中未得出结果的测试点。

划分分析的优点是将说明考虑在内了，因而有可能查出单纯域测试查不出的丢失路径错误。但是，仔细观察划分分析的着眼点，将会发现，它是从另一个角度去检测边界位移错误的。域测试是直接通过 ON-OFF-ON 点去检测边界位移，而划分分析则是通过说明与实现不同的一测试输入子空间去间接地探索真边界位移。

尽管域测试有着上述优点，但如果不解决两个致命弱点，恐怕将会仅仅以一种测试策略存在，很难具体运用到实践中去。这就是提出的限制过多，另外当程序存在很多路径时，所需的测试点也就很多。

5.4 符 号 测 试

普通的测试方法之所以不容易查出程序中的错误，一个重要的原因是测试点的选择比较困难。我们知道，被测程序与规格说明可能存在差距，无论是用功能测试还是用结构测试方法都不能保证选取到完全有代表性的测试点。由于测试人员的实践经验不足和被测程序逻辑上的复杂性，要使测试工作取得成功，谈何容易。人们公认，测试用例的选择是模块测试的瓶颈问题。能不能避开这个关键问题，符号测试的方法力图另辟捷径。

一、符号测试方法概述

符号测试的基本思想是允许程序的输入不仅仅是具体的数值数据，而且包括符号值，这一方法也正是因此而得名。这里所说的符号值可以是基本符号变量值，也可以是这些符号变量值的一个表达式。这样，在执行程序过程中以符号的计算代替了普通测试执行中对测试用例的数值计算。所得到的结果自然是符号公式或是符号谓词。更明确地说，普通测试执行的是算术运算，符号测试则是执行代数运算。因此符号测试可以认为是普通测试的一个自然的扩充。

如果原来测试某程序时，要从输入数据 X 的取值范围 1 到 500 中选取一个，进行数值运算。现在我们作符号执行，只需用符号值，例如 x1 作为输入数据，代入程序进行代数运算。所得结果是含有 x1 的代数式，它的正确性对于我们判断程序的正确性就直观多了。因为这一代数式本身就表明了运算过程。同时，进行一次符号测试等价于选用具体数值数据进行了大量的普通测试。比如，上述对 x1 的测试也许等价于进行了 500 次普通测试。

符号值可以是初等符号值，也可以是表达式。初等符号是任何变量值的字符串，表达式则是数、算术运算符和符号值的组合。下面短程序中可看到过程 SAMPLE 的变量符号值：

```
Procedure SAMPLE(X,Y)
S = 2*X + 3*Y
T = S - Y
RETURN
END
```

这里把 ▼ 当作程序变量的值，假定：

$$v(X) = a,\ v(Y) = b$$

则：

$$S = 2*a + 3*b$$

$$T = 2*a + 3*b - b$$

简化得到：

$$T = 2*a + 2*b$$

二、符号执行树

取符号值作输入数据，在执行程序的过程中，如果遇到条件语句：

IF 〈谓词〉 THEN……

ELSE……

由于这时的谓词可能是符号表达式，无法确定谓词究竟应该取真值，还是应该取假值，也就无法决定执行哪一个分支。这时我们不能无根据地任意舍去某个分支，因此两个可能的分支都必须保留，并分别做进一步的探索。如果此时能够获得另外的信息，得知程序变量的取值情况，则可据其作出选择。但通常的情况下是得不到这个额外帮助的。也就不得不兼顾两路，继续执行。在后面执行的过程中，凡遇到条件判断，都会出现类似的情况。每遇到一个条件判断出现两个分支，分别继续下去，又会遇到多个分支点。如果程序中仅有 IF 构成的分支，那么符号执行的过程可用一个二叉树来描述。我们称此二叉树为执行树 (execution tree)。把在各分支点的谓词条件累积在一起，用逻辑乘符号联接起来，得到的这个逻辑表达式称为路径条件 (path condition)。

以下让我们结合计算最大公因数的程序，对执行树和路径条件作进一步说明。图 5.19 给出了计算整数 X 和 Y 最大公因数的程序。图 5.20 则是该程序符号执行树，树上有编号的节点对应着流程图的特定语句。除去分支点外，普通节点都有一个后继节点。在

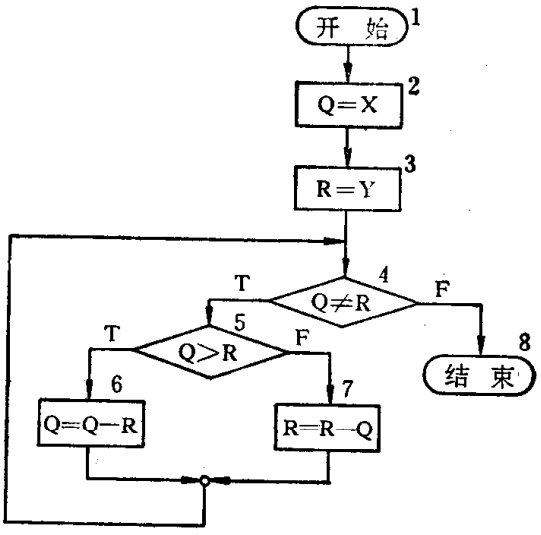


图 5.19 求最大公因数程序流程图

分支点上，由于无法决定谓词的取值，只好两个分支均予保留，但它所依赖的条件应对两个分支分别标明。例如，节点 4 的 $X \neq Y$ 和 $X = Y$。在程序入口处至尚未遇到条件语

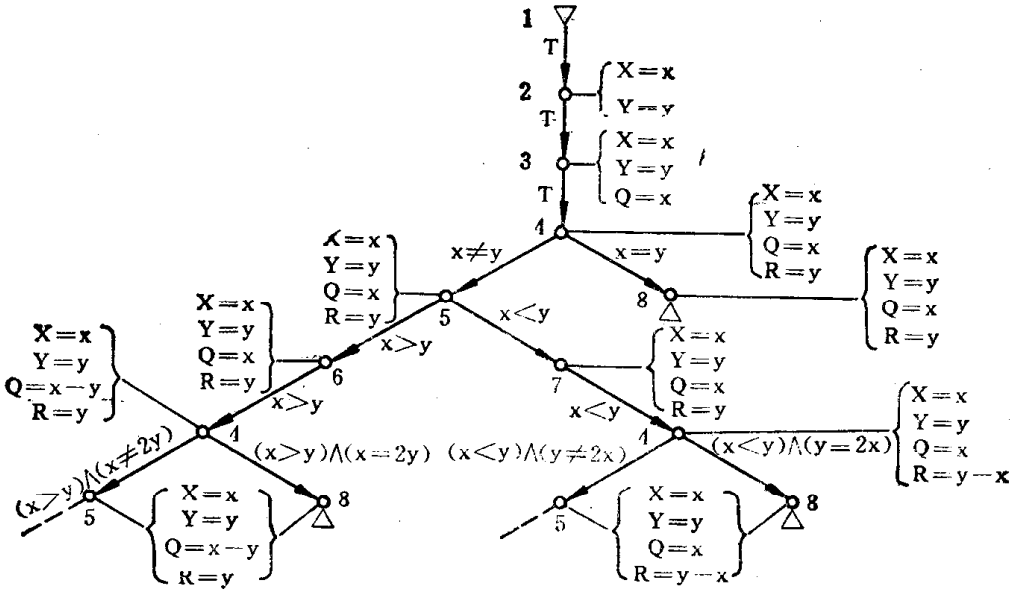


图 5.20　求最大公因数程序的符号执行树

句之前符号执行是无条件的，我们便在相应的控制流线上注以符号“T”。从树的根节点到每个叶都构成了程序的一条路径，控制流上注明的是路径条件。

从这个例子中可以看出，在程序的分支处，一个分支谓词的符号值实际上就是关于输入数据的限制。路径条件是该路径的谓词系统，它是多个等式或不等式的交。如果构成等式或不等式的表达式是线性的，我们可以通过解线性不等式组的办法，得到该路径规定的测试输入值的范围。这常常是符号测试的意义所在。产生符号输出结果的意义对于测试程序来说，不如产生某条路径的谓词系统的意义更大。

三、符号测试方法评述

符号测试可以看作是程序测试与程序验证的一个折衷方法。一方面，它沿用了传统的程序测试方法，通过运行被测程序来检验它的可靠性。另一方面，由于一次符号测试的结果代表了一大类普通测试的运行结果，实际上是证明了程序接受此类输入，所得输出是正确的，还是错误的。最为理想的情况是，程序中仅有有限的几条执行路径。如果对这有限的几条路径都完成了符号测试，我们就能较有把握地确认程序的正确性了。

从符号测试方法使用看，问题的关键在于开发出比传统的编译器功能更强，能够处理符号运算的编译器或解释器。

在实际运用时符号测试可按以下步骤进行：

① 利用符号执行解释器对被测程序进行符号执行。

② 若是遇到程序不能继续执行的情况，要求用户干预，或是遍历执行树的各分支路径。

③ 化简得到的路径条件式。

④ 用解线性不等式方法求解路径条件式，以求得满足各个限制谓词的测试数据。

⑤ 对程序进行测试，若上述不等式无解，则相应的路径不可执行。

从以上的分析看出，符号测试方法的一个优点是，可以很容易地确定，所给的一组测试用例是否覆盖了程序的各条路径。对于任何一组测试用例，可以首先确定它所经历的测试路径。然后，再给出输入变量的符号值，进而得到路径条件。

假设测试用例为 t_1、t_2、$\cdots$、t_n，它们在各自路径上的路径条件为 P_1、P_2、$\cdots$、P_n，其中

$$P_i = C_1 \wedge C_2 \wedge \cdots \wedge C_m$$

C_j 为第 j 个分支上的谓词。

若对于程序中的全部路径：

$$P = P_1 \vee P_2 \vee \cdots \vee P_n$$

为恒真，则表明程序中所有可能的控制路径均已走过。否则，$\neg P$ 是没有走过的路径对应的输入空间。

符号测试方法在使用中会遇到一些问题，这些问题到目前为止尚未得到圆满的解决，因而也就严重地影响着它的发展前景。这些问题包括：

① 分支问题

当采用符号执行方法进行到某一分支点处，分支谓词是符号表达式，这种情况下通常无法决定谓词的取值，也就不能决定分支的走向，需要测试人员作人工干预，或是按执行树的方法进行下去。如果程序中有循环，而循环次数又决定于输入变量，那就无法确定循环的次数。

② 二义性问题

数据项的符号值可能是有二义性的。这种情况常常出现在带有数组的程序中。

我们来看以下的程序段：

```
    ⋮
X(I) = 2 + A
X(J) = 3
C = X(I)
    ⋮
```

如果 I = J，则 C = 3，否则 C = 2 + A。但由于使用符号值运算，这时无法知道 I 是否等于 J。

③ 大程序问题

符号执行中总是要处理符号表达式。随着符号执行的继续，一些变量的符号表达式会越来越庞大。特别是当符号执行树如果很大，分支点很多，路径条件本身变成一个非常长的合取式。如果能够有办法将其化简，自然会带来很大好处。但如果找不到化简的办法，那将给符号测试的时间和运行空间带来大幅度的增长，甚至使整个问题的解决遇到难于克服的困难。

5.5 路径分析

分析程序中的路径是指：检验程序从入口开始，执行过程中经历的各个语句，直到出

口。这是白盒测试最为典型的问题。着眼于路径分析的测试可称为路径测试。完成路径测试的理想情况是做到路径覆盖。从上节的讨论中我们已经看出，对于比较简单的小程序实现路径覆盖是可能做到的。但如果程序中出现多个判断和多个循环，可能的路径数目将会急剧增长，以至实现路径覆盖不可能做到。

本节将从程序的路径表示入手，讨论路径数目的计算，从而看出路径数如何随着程序复杂度增加而增长。实际上我们可以做到的只能是程序中主要路径的测试。

一、程序路径表达式

进行路径分析，首先要解决的是路径如何表示的问题。以下给出几种表示方法。这些方法应该比较直观、形象，以利于人们理解，同时还必须能够容易在计算机中处理。

1. 路径的弧序列表示及节点序列表示

本章 5.1 节一开始已经介绍了程序控制流图。有了控制流图，给出路径的表示就容易多了。如果图 5.1(b) 中的 1 号节点为程序入口，5 号节点为程序出口，这一程序中可能的路径就有许多个。在此列举 4 条路径，分别以弧序列表示及节点序列表示(见表 5.4)。

表 5.4 路径的弧序列表示和节点序列表示

弧序列表示	节点序列表示	进入循环次数	经历左分支次数	经历右分支次数
acde	1-2-3-4-5	1	0	1
abe	1-2-4-5	1	1	0
abefabefabe	1-2-4-5-1-2-4-5-1-2-4-5	3	3	0
abefacde	1-2-4-5-1-2-3-4-5	2	1	1

2. 路径表达式 (path expression)

上述弧序列表示和节点序列表示均指某一路径而言。我们希望找到一种能够给出程序中所有路径的、更加概括的表示方法。这里介绍的路径表达式能够达到这一要求。

路径表达式作为一种表达式，其运算对象指的是控制流图中的弧，此外引入两个运算：乘和加。

弧 a 和弧 b 相乘，所得的乘积为 ab，它表示先沿弧 a，再沿弧 b 所经历的路段。注意，这里也同代数式一样，事实上省略了 a 和 b 之间的乘法运算符。其实这就是前面给出的弧序列表示，形式上没有什么不一样，不过这时我们应该想到，各弧之间是一种相乘的关系。例如，acde 是四个弧的乘积，它表示沿着 a、c、d 和 e 的顺序所经历的路段。

路径表达式中的另一运算是加。弧 a 与弧 b 相加，其和 a + b 表示两弧是或的关系。如图 5.21(a) 中节点 2 至节点 3 有两个弧相联，是并行的路段。这时运算符“+”是不可省的。

可以直观地看出在图 5.21(a) 所表示的控制流中，共有 4 条路径，它们是：eacf，eadf，ebcf 和 ebdf。这 4 条路径是并行的或的关系，我们完全可以用加运算联接它们，从而得到完整的路径表达式：

$$eacf + eadf + ebcf + ebdf$$

另一方面，还可直接从该图中看出，既然弧 a 和弧 b 是并行的，弧 c 和弧 d 也是并行的，它

们的头尾又有弧 e 和弧 f 相联。我们按上述两种运算的定义可立即写出路径表达式:

$$e(a + b)(c + d)f$$

不难发现，这一表达式正是前面 4 项之和表达式的因子提出形式。其实我们发现的并不是一个特殊情况的巧合，而是在路径表达式中普遍适用的分配律。

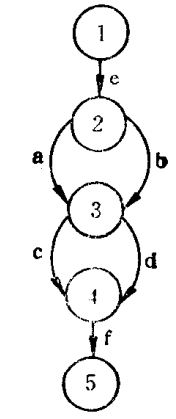
图 5.21(b) 给出的是具有循环的控制流图，它的路径随着执行不同次数循环体而有所不同，如 abd，abcbd，abcbcbd，···分别是执行一次、两次、三次循环体的路径。它的路径表达式可写成:

$$abd + abcbd + abcbcbd + \cdots\cdots$$

进一步化简为:

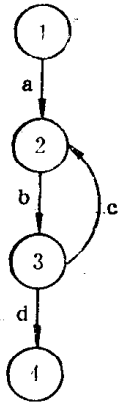
$$ab(1 + cb + cbcb + \cdots\cdots)d \quad 或:$$

$$ab(1 + cb + (cb)^2 + \cdots\cdots)d$$



eacf,eadf,ebcf,ebdf

(a)



abd,abcbd,abcbcbd,···

(b)

图 5.21　两个简单程序的控制流及其路径

事实上，路径表达式中的运算满足以下规律:

① 加法交换律　$a + b = b + a$

② 加法结合律　$a + (b + c) = (a + b) + c = a + b + c$

③ 加法幂等律　$a + a = a$

④ 乘法结合律　$a(bc) = (ab)c = abc$

⑤ 分配律　$a(b + c) = ab + ac$

$$(a + b)c = ac + bc$$

$$(a + b)(c + d) = a(c + d) + b(c + d)$$

注意，路径表达式中乘法是不满足交换律的。上述路径表达式中，a、b、c 和 d 均表示控制流图中的弧，但也可代表路径。例如:

若　$X = abc + def + ghi$

$Y = uvw + z$

则　$X + Y = abc + def + ghi + uvw + z$

$XY = (abc + def + ghi)(uvw + z)$

$= abcuvw + defuvw + ghiuvw + abcz + defz + ghiz$

二、程序中路径数的计算

如果人们提出问题：程序中有多少路径？这个问题表面上看似乎容易回答，其实不然。图 5.21(a) 的程序很容易看出，它有 4 条路径；(b) 所表示的程序路径数与循环的次数有关，如果循环体最多执行三次，那么路径数为3。然而，当程序中既有循环又有分支出现时，其路径数并不容易直觉地从控制流图中看出。比如图 5.1 中的程序就是这样。以下我们给出计算程序中路径数的一般方法。

1. 路径表达式计算

假定所讨论的程序已经得到了它的路径表达式，则可把其中的所有弧均代以数值

"1",然后依表达式的乘法和加法运算,所得数值即为该程序的路径数。

例如,图 5.21(a) 所给的程序,其路径表达式为 e(a + b)(c + d)f。把弧 a, b, c, d, e, f 均代入数值 1,计算表达式的值即为路径数:

$$N = 1 \times (1 + 1)(1 + 1) \times 1 = 4$$

再以图 5.1(b) 为例。如果暂不考虑循环,那么从节点 1 到节点 5,只有两条路径,即 L = a(b + cd)e, 代入数值"1",得到:

$$L = 1 \times (1 + 1 \times 1) \times 1 = 2$$

再加上循环,这时L表示循环体。假定只考虑循环次数小于 3 的情况。路径表达式为:

$$L + LfL + LfLfL = L(1 + fL + (fL)^2)$$

此式中 f 代入"1",L 代入"2",则计算出路径数为:

$$2 \times (1 + 1 \times 2 + (1 \times 2)^2) = 14$$

2. 程序复杂度计算

我们知道,程序中含有的路径数和程序的复杂性有着直接的关系。也就是说程序越复杂,它的路径数越多。但程序复杂性怎样度量呢? McCabe 给出了程序结构复杂性计算公式。

假定不考虑程序控制流图中,联结各节点之间弧的方向,控制流图便成为由节点和边构成的无向图。我们把无向图中任何两个节点之间至少存在一条通路的图称为连通图(connected graph)。McCabe 提出,对程序控制流图的连通图 G,其复杂度 V(G) 可按以下公式计算:

$$V(G) = E - n + 2$$

其中,E 为图 G 中的边数,n 为图 G 中的节点数。

图 5.22 给出了几个简单控制流图所表示程序的复杂性计算实例。其中前 4 个是基本控制结构,第 5 个是含有循环的小程序。需要提请注意的是计算出的复杂度值 V 恰是其控制流图中含有菱形判断的个数加1。同时,这个值也等于控制流图中控制流线把整个图的平面分割的域数。例如,图 5.22 中的第 5 个控制流图,其中有 3 个判断,V 值为 4;并且控制流线将平面分割成 4 个域(I、II、III 和 IV)。

3. 独立路径数

某一程序的独立路径是从程序入口到出口的多次执行中,每次至少一个语句(包括运算、赋值、输入输出或判断)是新的,未被重复的。如果用前面提到的控制流图来描述,独立路径就是在从入口进入控制流图后,至少要经历一个从未走过的弧。

图 5.23 是一个程序的流程图和控制流图。我们可以从图中看出它的 4 条独立路径(这里用节点序列表示):

① 1—11

② 1—2—3—4—5—10—1—11

③ 1—2—3—6—8—9—10—1—11

④ 1—2—3—6—7—9—10—1—11

以上四条路径画有底线的部分是相对于前次执行来说,尚未走过的部分。

事实上,路径 1—2—3—4—5—10—1—2—3—6—8—9—10—1—11 并不能认为是独

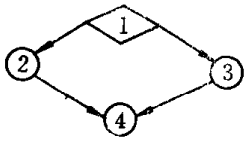
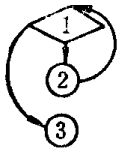
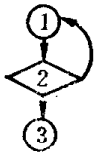
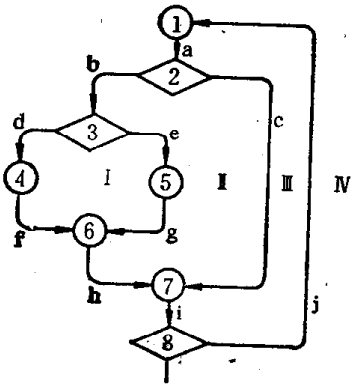
	程序结构	控　制　流　图	边数	节点数	V=E−n+2
1	顺序型		1	2	1
2	选择型		4	4	2
3	WHILE 型循环		3	3	2
4	UNTIL 型循环		3	3	2
5	含有一个循环的小程序		10	8	4

图 5.22　程序复杂度计算实例

立路径。因为它是前面给出的路径②和③的组合。

很明显，从测试工作来说，如果某一程序的每一独立路径都已测试过，那么可以认为程序中的每个语句都已检验过了。

三、程序路径的树表示及路径编码

1. 路径“与/或”树及其特征

程序中的路径也可用树来表示。树作为一种数据结构，我们不仅对它十分熟悉，而且已经掌握了许多对它的处理方法。这里用以表达程序路径的树，我们称之为路径“与/或”树（AND/OR Tree），以下简称为路径树。

前面讨论的路径表达式中，我们知道，只包含两种运算，即加和乘。表达式的加运算反映了它的两个运算对象的并列关系，也即要选择两个运算对象之一执行，其实这就是

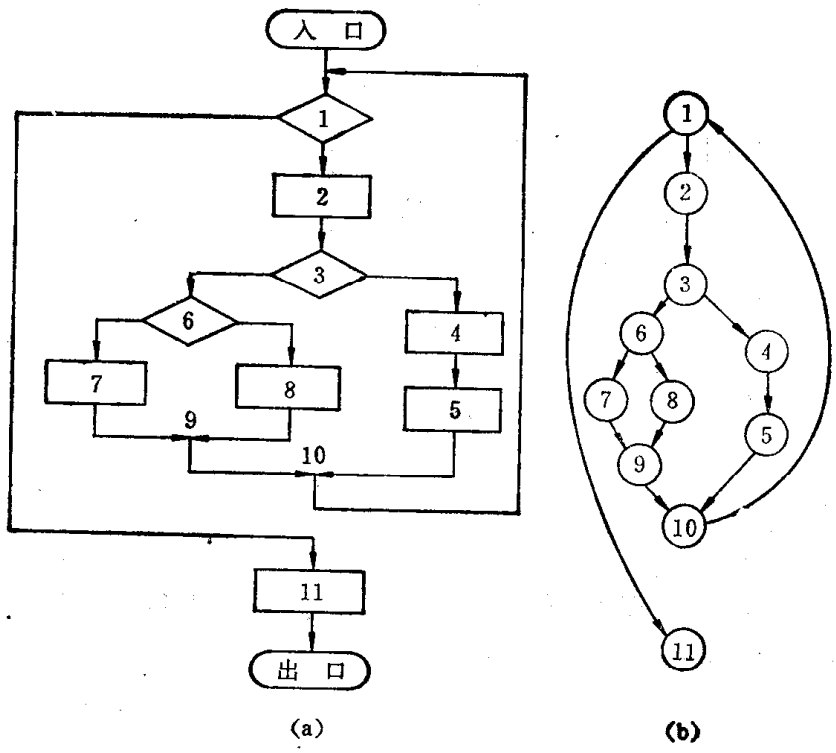


图 5.23 程序流程图和控制流图

"或"的关系。而表达式中乘运算反映的是在它的两个运算对象之间的串行关系，也就是"与"的关系。如果我们进一步把表达式和表达式的语法树联系起来，那就更容易理解了。

例如，我们有表达式

$$(a(b + (c + (e + d))))(f + g) + h$$

它所代表的程序控制流图如图 5.24(b) 所示。假定这里用标有"∧"的结点表示"与"关系，用标有"∨"的结点表示"或"关系，就得到了反映该表达式运算关系的路径树(如图 5.24(a) 所示)。图中方框均表示叶结点，即控制流图中的控制流线(或称弧)。

仔细研究和分析路径树，可以发现它具有以下一些特征：

① 对路径树中任一结点 N，设其进入端个数为 u，流出端分支个数为 v，我们用 N(u,v) 来表示该结点(所图 5.25 所示)。

若某结点 u = 0，则称它为根结点；若一结点 v = 0，则称它为叶结点；否则称为中间结点。

② 路径树中有且仅有一个根结点。

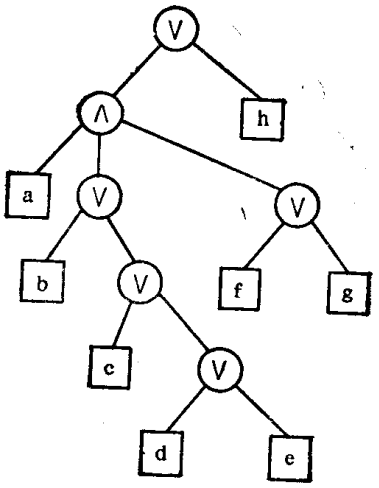
③ 路径树中每一叶结点都对应了它的程序控制流图中的一个弧；反之程序流程图中的任一控制流线也都对应了树中的一个叶结点。

④ 路径树中任一非根结点 N(u,v)，其进入端个数 u 必定为 1。因而，从根结点到树中任一叶结点有且仅有一条路径。实际上，路径树内在地反映了程序的控制流。

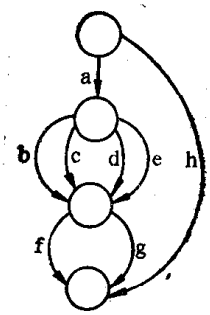
以上这些特征如有必要，可进一步证明。这里只需承认它们，不再证明。

2. 简化路径树

为了便于对路径树的处理，使之易于利用计算机实现对它的遍历和搜索，最终实现路径测试，我们需要将上述路径树作进一步简化。



(a)



(b) 控制流图

图 5.24 路径"与/或"树与控制流图

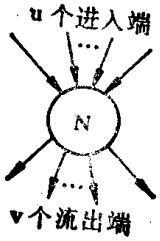


图 5.25 结点 N(u,v)

简化处理主要包括以下三个方面:

① 如果路径树中某 AND 结点的子结点中仍然有 AND 结点，就将孙结点提到子结点的位置，并去掉该 AND 子结点。按照这一原则，我们把图 5.26 中的 (a) 简化成 (b)。

② 如果路径树中某 OR 结点的子结点中仍然有 OR 结点，就将孙结点也提到子结点的位置来，并且去掉该 OR 子结点。按照这一原则，可以把图 5.26 的 (c) 简化成(d)。

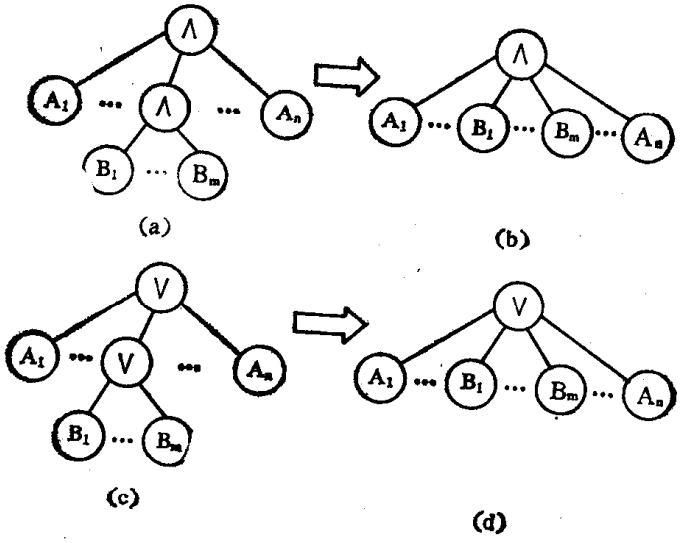


图 5.26 路径树的简化

③ 如果路径树的 AND 结点有一子结点是叶结点，我们可将此叶结点合并到其左、右邻结点中，进而消除 AND 结点和该叶结点构成的分支。在图 5.27 中，若 A 为一叶结点，它的相邻结点可能有以下几种情况:

1) 相邻结点也是叶结点，如 B 结点，则可简化成 AB 结点，如图 5.27(a) 所示。

2) 相邻结点是 OR 结点，如它联结了 B、C、D 叶结点，则可将 A "乘"入这些叶结

点，从而简化成 AB、AC 和 AD 三个叶结点，如图 5.27(b) 所示。

3) 相邻结点是 AND 结点，例如，它联结了 B 和 C 叶结点，那么我们可将 A "乘"入该 AND 结点的最左叶结点，使得它的叶结点变成 AB 和 C，如图 5.27(c) 所示。

任何路径树经过上述规则的简化都会具有新的特征，即：每一 OR 结点的任一分支都不再是 OR 结点，并且每一 AND 结点的任一分支都不再是 AND 结点，很可能是 OR 结点。图 5.28 便是图 5.24 中路径树的简化形式。

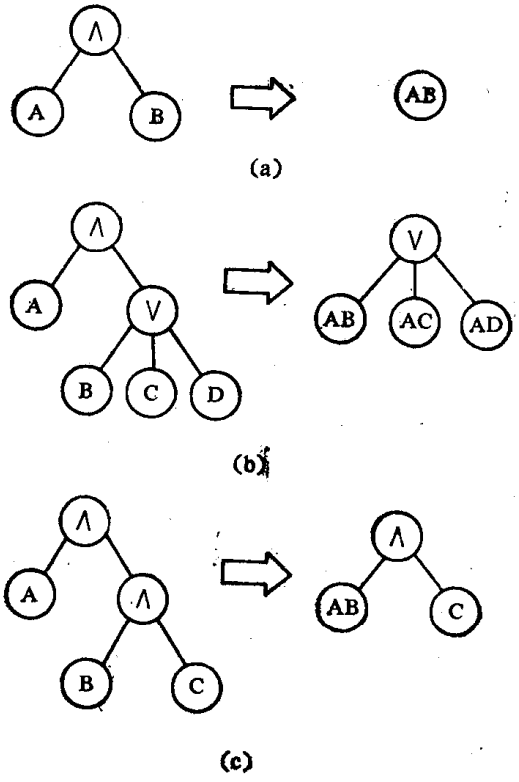


图 5.27 路径树的简化

3. 路径编码

为便于程序路径的机内处理，需要对它进行路径编码。原则上讲，只要求程序的路径与其编码有一一对应关系，可以使用各种编码方法。然而，许多编码方法并不能令人满意。这里我们将介绍一种效率较高的适合机内处理的路径编码方法。

由于路径树充分反映了程序路径的构成，特别是未经简化的路径树，它与程序的控制流图完全是一致的，因而不难发现，执行一次程序，从程序入口到出口经历某一路径，实际上就相当于对路径树的一次遍历。

如果在图 5.24(b) 的控制流图中执行的路径是 abf，那就相当于图 5.24(a) 图的路径树上的一次遍历：从根结点沿其左支到达 AND 结点，再沿左支到达叶结点 a，返回 AND 结点后，沿其中支达到 OR 结点，再进入左支，达到 b，返回 OR 结点，再返回 AND 结点后，进入它的右分支，遇有 OR 结点，进入左分支而达到叶结点 f；接着返回 OR 结点、返回 AND 结点，直至返回根结点。从而结束一次遍历。从这一遍历过程中我们发现，不同的路径，其差别仅仅在于到达 OR 结点后选择哪一分支。在程序中则表现为执行到某一判断时，其逻辑条件当时的取值。掌握了这一关键，就使我们突出 OR 结点的分支走向，并以此为依据进行编码。事实上，在 AND 结点上是没有选择余地的，也就是在 AND 结点附近没有各条路线的选择问题。在为路径编码时，我们不必考虑它们。或者说，在为路径编码时，AND 结点及其分支是多余的。

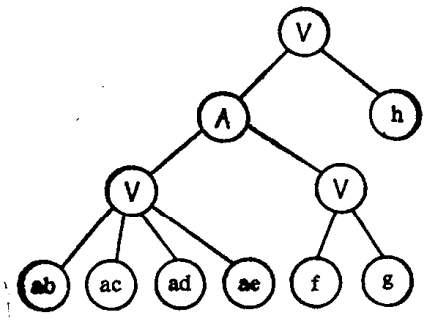


图 5.28 简化后的路径树

以下给出路径编码的方法：

① 路径树中每一 OR 结点的出端分支编码

若 OR 结点的出端分支个数为 v，其第 i 出端分支的编码为十进制数 i－1 的 L 位二进制码。其中，$L=[\log_2(v-1)+1]$，

这里方括号为取整运算。

图 5.29 给出了出端在 5 以内的编码。

② 路径的编码

在对路径树的所有 OR 结点的出端进行编码以后，任一路径的编码是从根结点出发遍历该路径各叶结点而后回到根结点经历的所有 OR 结点出端编码的序列。

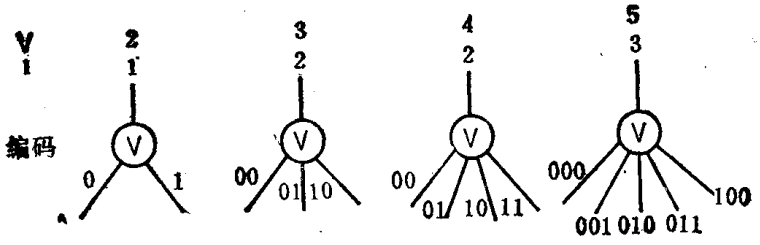


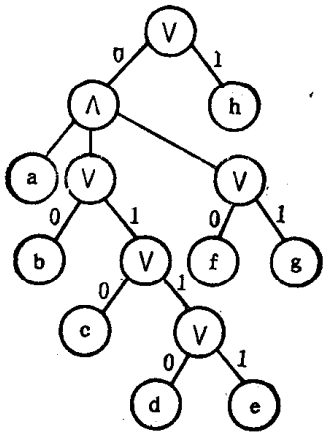
图 5.29 OR 结点出端的编码

作为实例，图 5.30 给出了图 5.24 中路径的编码。对于简化了的路径树，我们可以得到类似的结果（参看图 5.31）。但要特别注意，由于简化树不同于原来的路径树，其编码值也是不同的。

以上介绍的路径编码有着一个明显的优点，这就是在程序执行过程中可以沿着执行的控制流记录相应 OR 结点的编码，从而达到自动地完成动态编码的目的。

4. 路径译码

与路径编码的过程相反，在给出某个路径编码以后，找到与其对应的程序路径，我们



路　径	编　码	路　径	编　码
abf	000	abg	001
acf	0100	acg	0101
adf	01100	adg	01101
aef	01110	aeg	01111
h	1		

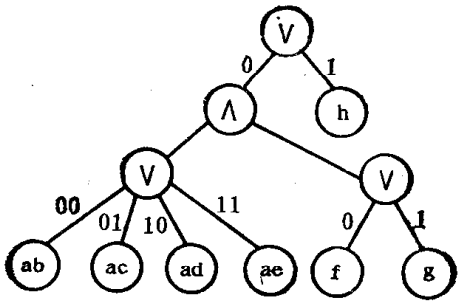
图 5.30 路径编码

称这一逆过程为路径译码。

由于编码过程利用了遍历路径树的方法，其逆过程译码也要利用遍历路径树。其中也必须注意掌握那些起关键作用的 OR 结点。针对上面二进制编码方法得到的路径编码，这里给出它的译码算法。

```
Decode (N(v): 流出端个数为 v 的结点)
int i,n
```

```
begin
  if (N(v) 是 OR 结点) then
    取编码的前 [log₂(v − 1) + 1] 位;
    并将其从编码串中删除;
    取 n 的值为这 [log₂(v − 1) + 1] 位二进制码的十进制数;
    if (N(v) 的第 n + 1 流出端是叶结点) then
      将此叶结点的名字拼入路径
      else Decode (N(v) 的第 n + 1 端);
  else for i = 1 to v do
      begin
          Decode (N(v) 的第 i 端)
      end
end
```



路 径	编 码	路 径	编 码
abf	0000	abg	0001
acf	0010	acg	0011
adf	0100	adg	0101
aef	0110	aeg	0111
h	1		

图 5.31 简化树的路径编码

为具体说明这一译码算法，这里仍以图 5.30 给出的路径树为例，描述过程。假设给定的路径编码为 0100，要译出它所对应的路径。我们从图 5.30 的根结点开始，根据该编码遍历路径树。首先看到根结点是 v = 2 的 OR 结点，取出编码 0100 的前 $[\log_2(2-1)+1]=1$ 位“0”，沿根结点左支，达到 AND 结点，其第一分支得到叶结点 a；再进入第二支，遇到 v = 2 的 OR 结点，再取编码 100 的前 $[\log_2(2-1)+1]=1$ 位“1”，进入该 OR 结点的右支；再遇 v = 2 的另一 OR 结点，取编码 00 的前 1 位“0”，沿其左支得到叶结点 c，因而拼得 ac；返回至 AND 结点后进入其第三支，由于这仍是 v = 2 的 OR 结点，取得编码的最后一个“0”，得到叶结点 f，并拼得字符串 acf；然后返回根结点。至此完成一次遍历，并得到路径 acf。

四、测试路径枚举

1. Z 路径覆盖

在结构测试覆盖中，人们最常用，也是最为熟悉的测试准则常常是语句覆盖、分支覆盖、条件覆盖以及分支/条件覆盖等。至于路径覆盖，由于路径数目过多，使得人们不敢问津。因为路径数目被循环结构搞成天文数字，那是不可能去覆盖它们的。

为了解决这一问题，我们必须舍掉一些次要因素，对循环机制进行简化，从而极大地

减少路径的数量，使得覆盖这些有限的路径成为可能。我们称简化循环意义下的路径覆盖为Z路径覆盖。

这里所说的对循环化简是指，限制循环的次数。无论循环的形式和实际执行循环体的次数多少，我们只考虑执行循环一次和零次两种情况。也即只考虑执行时进入循环体一次和跳过循环体这两种情况。图5.32中（a）和（b）表示了两种最典型的循环控制结构。前者先作判断，循环体B可能执行(假定只限执行一次)，也可能不执行。这就如同（c）所表示的条件选择结构一样。后者先执行循环体B（假定也只执行一次),再经判断转出，其效果也与（c）中给出的条件选择结构只执行右支的效果一样。

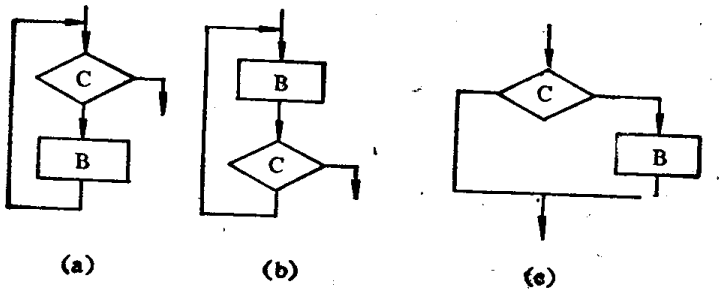


图5.32　循环结构简化成选择结构

2. 路径枚举

前已阐明，路径树能够全面地反映程序中所含路径。我们可以认为，只要得到了路径树，就能够给出所有的路径。办法仍然是遍历。因为在取得某一程序的路径树后，从其根结点开始，一次遍历，再回到根结点时，把所经历的叶结点名排列起来，就得到一个路径。如果我们设法遍历了所有的叶结点，那就得到了所有的路径。

需要说明的是当采用上述对循环的限制以后，路径的数量不会过大。因而实现整个路径树的全部路径遍历，从而得到所有路径是完全可能的。这就是路径枚举要做的事。

五、路径测试系统

本节前面讨论的路径分析问题似乎是一些零散的内容，但分别讨论的目的还是明确的，那就是要解决路径测试问题。这一目标可以通过以下的描述进一步澄清。也许图5.33

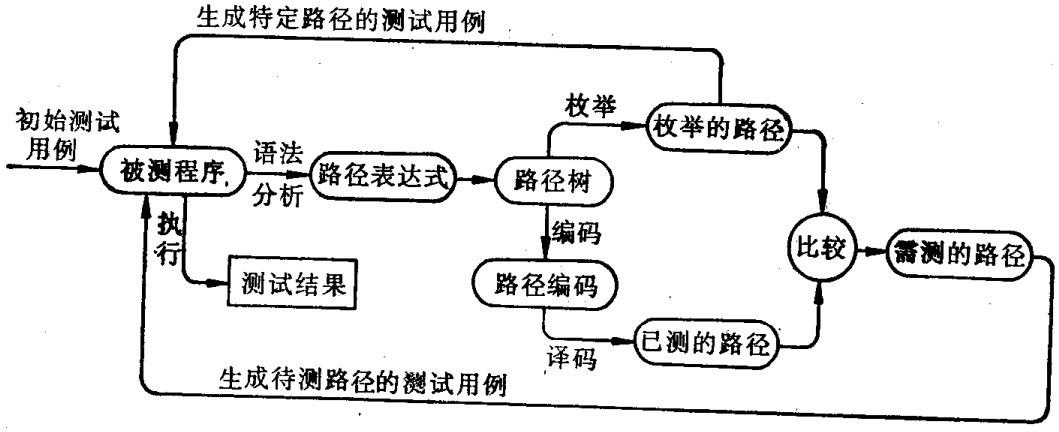


图5.33　路径测试系统

有助于我们理解问题的全貌。

在路径测试中，最关键的问题仍然是如何得到测试用例，使之既能够避免测试的盲

目性，又能有较高的测试效率。在我们的路径测试系统中，可以有三个途径得到测试用例：

1. 通过非路径分析得到的测试用例

这些测试用例也许是应用系统本身在实践中提供的，也许是测试人员凭工作经验得到的，甚至是猜测的。在使用这些测试用例执行被测程序后，一方面可以取得测试结果供进一步分析；另一方面，经过语法分析求得路径表达式，继而生成了该程序的路径树。随着程序的执行得到这次执行路径的编码，在经过译码后就得到了这次执行的路径。

2. 找到尚未测过的路径并生成相应的测试用例

枚举被测程序路径树的所有路径，并与前面已测路径进行对比，便可得知哪些路径尚未测过，针对这些路径生成测试用例，进而完成对它们的测试。

3. 指定特定路径生成相应的测试用例

测试者根据枚举的路径，从中指定某些特定的路径，生成其相应的测试用例。

按以上方法实施测试原则上是可以做到路径覆盖的。其理由是：

① 对程序中的循环作了如上限制以后，程序路径的数量是有限的。

② 程序的路径可经枚举全部得到。

③ 完成若干个测试用例后，所测路径是哪些，尚有哪些路径待测是可以知道的。

④ 在指出要测的路径以后，可以自动生成相应的测试用例。

经过几年的探索和实践，清华大学的程序路径分析科研组已实现了路径表达式生成、路径树生成、路径编码、路径译码和路径枚举的科研项目。由路径生成测试用例是沿路径树的回溯过程，目前尚未实现，需要做进一步探索。

5.6 程 序 插 装

程序插装（Program Instrumentation）是一种基本的测试手段，在软件测试中有着广泛的应用。

一、方法简介

程序插装方法简单地说是借助往被测程序中插入操作来实现测试目的的方法。

我们在调试程序时，常常要在程序中插入一些打印语句。其目的在于，希望执行程序时，随带打印出我们最为关心的信息。进一步通过这些信息了解执行过程中程序的一些动态特性。比如，程序的实际执行路径，或是特定变量在特定时刻的取值。从这一思想发展出的程序插装技术能够按用户的要求，获取程序的各种信息，成为测试工作的有效手段。

如果我们想要了解一个程序在某次运行中所有可执行语句被覆盖(或称被经历)的情况，或是每个语句的实际执行次数，最好的办法是利用插装技术。这里仅以计算整数X和整数Y的最大公约数程序为例，说明插装方法的要点。图5.34给出了这一程序的流程图。图中的虚线框并不是原来程序的内容，而是为了记录语句执行次数而插入的。这些虚线框要完成的操作都是计数语句，其形式为：

$$C(i) = C(i) + 1 \qquad i = 1,2,\cdots,6$$

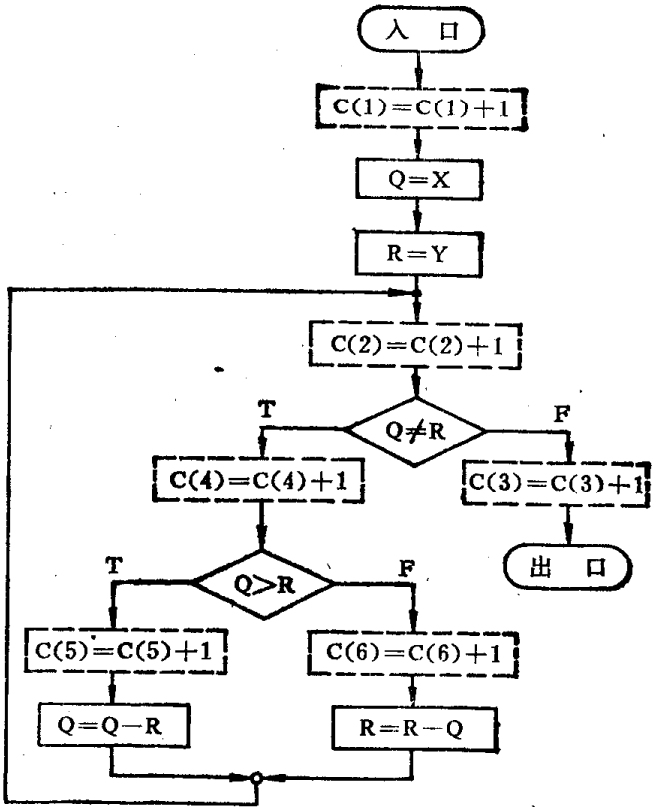


图 5.34　插装后的求最大公约数程序流程图

程序从入口开始执行，到出口结束。凡经历的计数语句都能记录下该程序点的执行次数。如果我们在程序的入口处还插入了对计数器 C(i) 初始化的语句，在出口处插入了打印这些计数器的语句，就构成了完整的插装程序。它便能记录并输出在各程序点上语句的实际执行次数。图 5.35 表示了插装后的程序，图中箭头所指均为插入的语句(原程序的语句已略去)。

通过插入的语句获取程序执行中的动态信息，这一做法正如在刚研制成的机器特定部位安装记录仪表是一样的。安装好以后开动机器试运行，我们除了可以从机器加工的成品检验得知机器的运行特性外，还可通过记录仪表了解其动态特性。这就相当于在运行程序以后，一方面可检验测试的结果数据，另一方面还可借助插入语句给出的信息了解程序的执行特性。正是这个原因，有时把插入的语句称为“探测器”，借以实现“探查”或“监控”的功能。

在程序的特定部位插入记录动态特性的语句，最终是为了把程序执行过程中发生的一些重要历史事件记录下来。例如，记录在程序执行过程中某些变量值的变化情况，变化的范围等。又如本章 5.2 节中所讨论的程序逻辑覆盖情况，也只有通过程序的插装才能取得覆盖信息。实践表明，程序插装方法是应用很广的技术，特别是在完成程序的测试和调试时非常有效。

设计程序插装程序时需要考虑的问题包括：

① 探测哪些信息；

② 在程序的什么部位设置探测点；

③ 需要设置多少个探测点。

其中前两个问题需要结合具体课题解决，并不能给出笼统的回答。至于第三个问题，需要考虑如何设置最少探测点的方案。例如，图 5.34 中程序入口处，若要记录语句 Q＝X 和 R＝Y 的执行次数，只需插入 C(1)＝C(1)＋1 这样一个计数语句就够了，没有必要在每个语句之后都插入一个计数语句。在一般的情况下，我们可以认为，在没有分支的程序段中只需一个计数语句。但程序中由于出现多种控制结构，使得整个结构十分复杂。为了在程序中设置最少的计数语句，需要针对程序的控制结构进行具体的分析。这里我们以 FORTRAN 程序为例，列举至少应在哪些部位设置计数语句：

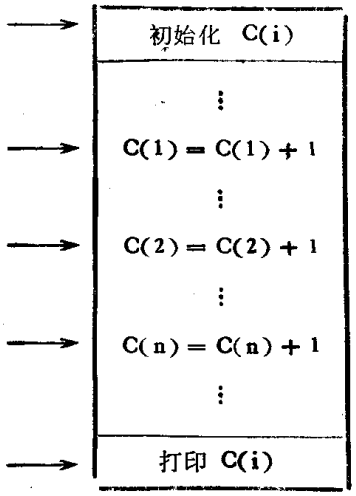


图 5.35 插装程序中插入的语句

① 程序块的第一个可执行语句之前；

② ENTRY 语句的前后；

③ 有标号的可执行语句处；

④ DO、DO WHILE、DO UNTIL 及 DO 终端语句之后；

⑤ BLOCK-IF、ELSE IF、ELSE 及 ENDIF 语句之后；

⑥ LOGICAL IF 语句处；

⑦ 输入/输出语句之后；

⑧ CALL 语句之后；

⑨ 计算 GO TO 语句之后。

在一般情况下应如何掌握呢？Ramamoorthy 提出了插入程序中记录器最少个数的一般方法。

二、断言语句

在程序中的特定部位插入某些用以判断变量特性的语句，使得程序执行中这些语句得以证实，从而使程序的运行特性得到证实。我们把插入的这些语句称为断言（assertions）。这一作法是程序正确性证明的初等步骤，尽管算不上严格的证明，但方法本身仍然是很实用的。我们在后面程序正确性证明一章中还要进一步讨论 FLOYD 的归纳断言法。这里仅以求两个非负数 NUM 和 DEN 之商的 Wensley 迭代算法为例，对断言语句的作用作一简要说明。

假定两个非负数中，NUM 小于 DEN（即所得之商小于 1），算法中只用到加、减及除 2 的运算。该迭代算法的程序如图 5.36 所示。

从程序中可以看出，在每次迭代中由分母得到的变量 B 以及权增量 W 都要缩小一半，而且变量 A 随着迭代次数的增加将接近分子。这些粗略的观察和分析可以用以下 4 个断言语句表达，在每次迭代开始时 4 个断言必定为真：

① $W = 2^{-K}$　　K 是迭代次数，并且 $K \geqslant 0$；

② $A = DEN * Q$；

③ $B = DEN * W/2$；

④ $NUM/DEN - W < Q \leqslant NUM/DEN$。

此外，我们还看出，在循环外 $W < E$，而且结果Q总是从下面逼近真正的商。这就得到了输出断言：

$$NUM/DEN - E < Q \leqslant NUM/DEN$$

它和上面的第④断言很相似。

假定我们所用的编译系统能够处理表达式形式的断言语句，插入断言以后的程序如

```
procedure DIVIDE (NUM,DEN,E,Q)
* E is the accuracy required. E≥0. Q is both   *
* the result at exit and at any interim stage.  *
* A, B and W are the other elements of the pro-*
* gram vector.                                  *
    Q: = 0
    A: = 0
    B: = DEN/2
    W: = 1
    until W<E loop
      if (NUM - A - B)≥0
       then
         Q: = Q + W/2
         A: = A + B
      endif
      B: = B/2
      W: = W/2
endloop
end
```

图 5.36　计算两非负数之商的迭代程序

图 5.37 所示。其中带有标记@的语句是断言语句。新增加的变量K只是在计算第①断言时用到。

首先来检验在初始化以后循环内的断言。

① 由于 $K = 0$，所以 $W = 2^{-K} = 1$ 是初值。

② 由于 $Q = 0$，A 的初值 $A = DEN * Q = 0$。

③ 将W的值代入 $DEN * W/2$，则得B 的初值 $B = DEN/2$。

④ 我们曾假定 $0 \leqslant NUM < DEN$，$NUM/DEN - 1$ (W的值)必定小于零，因而也就小于Q(它是零)。而且，NUM/DEN 必定大于 Q，因为 NUM 和 DEN 均为正，Q 为零。

以上说明了这些断言在初始状态下为真。如果继续迭代，要证明断言为真，就必须证明无论 if-then 结构中执行什么路径这些断言都是真。让我们先来考虑，在初始测试中

```
      procedure DIVIDE (NUM,DEN, E,Q)
*  E is the required accuracy. E≥0. Q is both     *
*  the result at exit and at any interim stage.    *
*  A, B and W are the other elements of the pro-*
*  gram vector.                                    *
      Q: = 0
      A: = 0
      B: = DEN/2
      W: = 1
@     K: = 0
      until K<E loop
@      assert W = 1/2**K
@      assert A = DEN*Q
@      assert B = DEN*W/2
@      assert NUM/DEN - W<Q and Q≤NUM/DEN
       if (NUM - A - B)≥0
        then
         Q: = Q + W/2
         A: = A + B
       endif
       B: = B/2
       W: = W/2
@      K: = K + 1
       endloop
@      assert NUM/DEN - E<Q and Q≤NUM/DEN
       end
```

图 5.37　插入断言后的迭代程序

NUM-A-B ≥ 0 为假,即检验失败。然后给出程序向量的新值 (A′, B′,W′, Q′ 和 K′),我们有:

$$A' = A$$
$$B' = B/2$$
$$W' = W/2$$
$$Q' = Q$$
$$K' = K + 1$$

再来检验 4 个断言:

① $W' = W/2 = 1/2 * *K'$

② $A' = A = DEN * Q = DEN * Q'$

③ $B' = B/2 = DEN * W/4 = DEN * W'/2$

④ 把A和B 代入(NUM − A − B < 0),得到 $NUM - DEN * Q - DEN * W/2 < 0$, 对此关系式两端除以 DEN, 并加 Q, 得到 $NUM/DEN - W/2 < Q$,由于$Q' = Q$, $W' = W/2$, 我们有 $NUM/DEN - W' < Q'$, 且 $Q' \leqslant NUM/DEN$。

如果 if-then 检验成立,再来看 4 个断言。使用 A″, B″, W″, Q″ 和 K″ 作为新

的程序向量，我们有：

① $W'' = W/2 = 1/2 * *K''$

② $A'' = DEN * Q + DEN * W/2 = DEN * (Q + W/2) = DEN * Q''$

③ $B'' = B/2 = DEN * W/4 = DEN * W''/2$

④ 代入 $(NUM - A - B \geqslant 0)$，并作同前的变换，得到 $NUM/DEN - W''/2 < Q'' + W''/2$ 或：

$$NUM/DEN - W'' < Q'', \text{ 并且 } Q'' \leqslant NUM/DEN。$$

总之，无论执行哪一路径，在每一迭代的开始，4 个断言均为真。尽管并未考虑输出断言，但是我们知道，第④断言成立，由于 $W < E$，$NUM/DEN - E < Q$ 和 $Q \leqslant NUM/DEN$ 必定为真，也就必定满足输出断言。

5.7 程 序 变 异

程序变异方法（Program Mutation）与前面提到的结构测试和功能测试都不一样，它是一种错误驱动测试。

所谓错误驱动测试方法，是指该方法是针对某类特定程序错误的。经过了若干年的测试理论研究和软件测试的实践，人们逐渐发现要想找出程序中所有的错误几乎是不可能的。比较现实的解决办法是将错误的搜索范围尽可能地缩小，以利于专门测试某类错误是否存在。这样做的好处在于，便于集中目标于对软件危害最大的可能错误，而暂时忽略对软件危害较小的可能错误。这样可以取得较高的测试效率，并能降低测试的成本。

错误驱动测试主要有两种，即程序强变异和程序弱变异。本节将分别作一介绍。

一、程序强变异

程序强变异通常被简称为程序变异。它是由 R. A. DeMillo 和 T. A. Budd 等人最早提出的。

Demillo 认为，当程序被开发并经过简单测试后，残留在程序中的错误不再是那些很重大的错误，而是一些难以发现的小错误。比如，遗漏了某个操作、分支谓词规定的边界有位移等等。即使是一些稍微复杂一些的错误，也可以看作是这些简单错误的组合。程序变异的目标就是查出这些简单的错误及其组合。

为说明什么是程序变异，首先定义程序的变异因子。程序 P 的变异因子 m(P) 也是一个程序，它是对 P 进行微小改动而得到的。因而 m(P) 是 P 的一个变换，或称是 P 的变异因子。

如果程序 P 是正确的，m(P) 是一个几乎正确的程序。如果 P 不正确，则 P 的某一个变异因子 m(P) 可能是正确的。

假设 P 有一组测试数据，其集合为 D。如果对于 D，P 存在错误，则可对程序进行修改。我们现在关心的是 P 在 D 上无错误的情形。这时，仍然不能肯定程序就是正确的。那么，怎样来确定程序的正确程度呢？我们可以来看看程序变异的思想。

若 P 在 D 上是正确的，可以找出 P 的变异因子的某一集合：

$$m = \{M(P) | m(P) \text{ 是 P 的变异因子}\}$$

若m中每一元素在D上都存在错误，则可认为程序的正确程度较高。若m中某些元素在D上不存在错误，则可能存在三种情况：

① 这些变异因子与P在功能上是等价的。

② 现有的测试数据不足以找出P与其变异因子间的差别。

③ P可能含有错误，而其某些变异因子却是正确的。

可以仔细地核对程序及其变异因子，竭力避免第一种情况出现。第二种可能情况告诉我们，当P与某些变异因子都正确时，可能是因为测试数据太少，并且不够典型造成的。这时，应该增加测试数据，直到"杀掉"所有的变异因子。也就是说，让所有的变异因子都出错。第三种情况则提醒我们，当许多典型的测试数据仍然不能使某一变异因子出错时，此变异因子可能是程序的正确形式。

程序变异意味着对测试数据集中的每一元素，都要对程序P及其变异因子进行测试，所以要求测试数据集D和变异因子集m(P)都需精心挑选。这是强变异方法成功的关键。

使用程序变异方法，最重要的是怎样建立变异因子。我们可以把变异因子看作是变异操作符作用在被测程序上的结果。变异操作符接受被测程序为输入，而产生一系列不同的变异因子。

如果把程序处理的数据元素看作是常量、标量或数组，那么可在数据元素被引用时用其它的数据元素替换。可以将常量增加一小点，或是减少一小点，也可以将数组变量名换为另一相同维数的数组变量名。还可以将操作符作些变换，替换或删除语句，改变GOTO语句的转向点等等。

总之，对程序进行变换的方式是多种多样的，而且还紧紧地依赖于被测程序使用的设计语言。究竟对程序作什么样的变换，很多情况下，与测试人员的实践经验有关。实际上，通过变异分析构造测试数据的过程是一个循环过程。测试人员首先提供被测程序以及初始数据，还有则是要应用于程序的变异运算符。当由此产生的变异因子和程序本身被初始测试数据测试后，可能会有变异因子未被发现错误。这时，用户可以增加测试数据。若所有的变异因子均出错(均被"杀掉")，用户也可以增加新的变异因子。然后进行下一轮变异测试。

Budd 曾列出了如下一些常用的变异运算：

常量之间替换
标量与变量替换
将常量替换为标量
将标量替换为常量
将常量替换为数组分量
将标量替换为数组分量
将数组分量替换为常量
将数组分量替换为标量
数组分量之间替换

数组名替换
算术运算符替换
关系运算符替换
逻辑运算符替换
插入绝对值符号
插入单目运算符
语句分解
语句删除
GO TO
循环终止条件变换

Budd 等人统计了他们设计的程序变异系统在辅助测试人员进行测试的效果。被测程序有 13 个，共包含 30 个错误。统计结果如图 5.38 所示。从中看出程序变异是较为有效的测试方法。

错误类型	总数	查出数
丢失路径错误	7	6
不正确谓词	5	3
不正确计算语句	15	14
丢失计算语句	3	2

图 5.38　变异测试查错效果

程序变异方法也有两大弱点。一是要运行所有的变异因子，从而成倍地提高了测试的成本；二是决定程序与其变异因子是否等价是一个递归不可解问题。但不管怎样，程序变异由于其针对性强、系统性强，正成为软件测试中一种相当活跃的办法。特别是在变异测试系统的支持下，用户可以更有效地测试自己的程序。

二、程序弱变异

程序弱变异方法（Weak Mutation）是 Howden 提出的。由于程序强变异要生成变异因子，为与此相区别，Howden 称只是对被测程序进行测试的变异方法为弱变异。

弱变异方法的目标仍是要查出某一类错误。其主要思想如下所述。设 P 是一个程序，C 是 P 的简单组成部分。若有一变异变换作用于 C 而生成 C′，如果 P′ 是含有 C′ 的 P 的变异因子，则在弱变异方法中，要求存在测试数据，当 P 在此测试数据下运行时，C 被执行，且至少在一次执行中，使 C 产生的值与 C′ 不同。

从这里可以看出，弱变异和强变异有很多相似的地方。它们的主要差别在于，弱变异强调变动程序的组成部分。根据弱变异准则，只要事先确定导致 C 与 C′ 产生不同值的测试数据组，则可将程序在此测试数据组上运行，并不实际产生其变异因子。

在弱变异的实现中，关键问题是确定程序 P 的组成部分集合以及与其有关的变换。组成部分可以是程序中的计算结构、变量定义与引用、算术表达式、关系表达式以及布尔表达式等等。其中一个组成元素可以是另一组成元素的一部分。

Howden 提出了5种最基本的程序组成部分：变量引用、变量定义、算术表达式、关系表达式和布尔表达式。其中前两部分可以包含在后3部分中。算术表达式可以包含在关系表达式中，而关系表达式又可包含在布尔表达式中。以下分别进行讨论：

① 变量引用

变量引用包括变量值的使用。如在语句：

$$A = B + A$$

中，变量B被引用，变量A则先被引用，继而又被定义。再如语句：

$$\text{if} \quad (A > B)$$

中，A、B 的值均被引用，这些语句是变量引用组成部分。

变量引用的弱变异使得被引用的变量变为另一变量。如在语句A＝B中，使B变为另一程序变量。比如说 D，则语句变为 A＝D。假设V是程序P中变量引用组成部分C的一个被引用变量，C′是C的一个变换。要保证C和 C′ 在测试数据上执行时产生的值不同，必须要求程序执行到C时，所有的程序变量的值均与V值不同。因为，如果执行到C时，假设W的值与V相等，若组成部分C为 A＝V，变换 C′ 为 A＝W。那末这组测试值查不出此错误变量引用的错误。

从以上的分析可看出，要保证在一变量引用中，被引用的变量是正确变量，必须产生测试数据组，使得运行到此变量引用时，被引用变量不与任何其它程序变量的值相等。通常一组测试数据仅能保证某几个程序变量值与被引用变量值在运行到变量引用组成部分时不相等。要让所有的程序变量都不与被引用变量相等，这就需要很多组测试数据。实际上，由于程序中有很多变量引用组成部分，要对所有组成部分进行弱变异测试，是非常耗费时间的。

较为经济的办法是，将变量引用组成部分局限在错误数组元素引用与错误输入变量引用两种元素上。据统计，由于下标错误，常常会发生错误的数组元素引用。此外，当函数之间有参数传递时，函数中的形式参数(它是被引用的)，也常常会出现错误。对这两种元素进行弱变异的准则是：选取测试数据，使程序运行到该组成部分时，数组的各元素均不相等，或者是程序的形参互不相等。

② 变量定义

变量定义指的是给变量赋以新值。如在语句 A＝B 中，变量A被赋予新值 B，我们称此为A被定义了，此语句被称为变量定义组成部分。

变量定义的弱变异使得被定义的变量变为另一变量。如在语句A＝B中，使变量A变为另一程序变量，比如说 D，则语句变为 D＝B。假设V是程序P中变量定义组成部分C的一个被定义变量，C′是C的一个变换。要保证C和 C′ 在测试数据上产生的值不同，必须选择测试数据，使程序执行到C前V的值与C执行后V的值不一致。否则，若C的定义是错误定义，V的值仍然未起变化，因而不易查出这一错误。

③ 算术表达式

算术表达式的弱变异考虑三类常见错误：表达式与正确表达式相差一常数；表达式是正确表达式的常数倍；表达式的系数有错误。实际上，后一种情况包括了前两种情况。但由于前两情况要求较少的测试数据，因而将其单独分类。

设 exp 是一算术表达式，exp′ 是在 exp 上加一常数后的表达式，要使运行到 exp 所在的语句时，exp ≠ exp′，任取一组测试数据都可以。如果 exp′ 是 exp 的常数倍，要使运行到 exp 所在语句时，exp ≠ exp′，则要求取一组测试数据，使运行到 exp 所在语句时，exp ≠ 0。

若 exp′ 是 exp 的某些系数变换后的表达式，设K为 exp 中变量指数的上界，若X是 exp − exp′ 的一个根，则在X处，exp = exp′，这时测试数据无法查出此算术表达式错。但是，exp − exp′ 的根仅有有限个，所以任意选取一组测试数据，使得 exp 等于 exp′ 的概率很小。实际测试时，基于这一小概率事件，可以多取几组测试数据，以进行算术表达式的弱变异测试。

④ 算术关系

算术关系的弱变异考虑两类简单的错误：关系运算符错与相差一常数的错误。

设R是一算术关系，R′ 是有错误关系运算符的R的一变换。即：

$$R = exp_1 \quad r \quad exp_2,$$

而：

$$R' = exp_1 \quad r' \quad exp_2, \text{ 但 } r \neq r'。$$

那么，只要选择测试数据，使：

$$exp_1 < exp_2,\ exp_1 > exp_2, \text{ 和 } exp_1 = exp_2$$

的情况都出现一次，则 R′ 和R的真值总会在某一组测试数据下不同。例如：

$$R = exp_1 < exp_2,\ R' = exp_1 \leqslant exp_2,$$

则当取测试值，使：

$$exp_1 = exp_2$$

时，R与 R′ 的真值不同，R为假，而 R′ 为真。其它的情况可以同样推出。

对于相差一常数的错误，可以设：

$$R = exp\ r\ 0$$

其中，r 是关系，那么

$$R' = (exp + K)\ r\ 0$$

其中 K ≠ 0。这里，需要的测试数据与 r 有关。若 r = '<'，则应该取两组测试数据，一组使运行到该组成部分时，exp 的值是小于零的最大可能值；第二组使运行到该组成部分时，exp 的值是大于或等于零的最小可能值（注意，这里的最大、最小可能值均与 exp 的数据类型有关）。如果K大于零，那么第一组测试值使 exp < 0，而 exp + K ≥ 0（即 exp < 0 为假），故能加以区别；如果K小于零，第二组测试值使 exp ≥ 0（即 exp < 0 为假），而 exp + K < 0，也能加以区别。同样，可以找出 r 为其它关系符时应取的测试数据，见图 5.39。

可以将区别错误关系符与相差常数错误的测试数据结合起来，即要求对关系运算：

$$exp_1 \quad r \quad exp_2$$

有测试数据，使运行到该组成部分时，$exp_1 - exp_2$ 为小于零的最大值、零或大于零的最小值。

⑤ 布尔关系

布尔关系是算术关系用 NOT、AND 和 OR 三个逻辑运算符连接起来的复杂关

关系 r	测 试 数 据
$<$	小于零的最大可能数，大于等于零的最小可能数
$>$	大于零的最小可能数，小于等于零的最大可能数
$\leqslant$	大于零的最小可能数，小于等于零的最大可能数
$\geqslant$	小于零的最大可能数，大于等于零的最小可能数
$=$	零
$\neq$	零

图 5.39 对算术关系进行弱变异的测试数据

系。设B是一个布尔关系，

$$B = L(E_1, E_2, \cdots, E_n)$$

其中，E_i $(i = 1,2,\cdots,n)$ 是算术关系，L是含有 NOT、AND 或 OR 运算符的逻辑表达式，B的一个变换：

$$B' = L'(E_1, E_2, \cdots, E_n)$$

则要使B与 B′ 区别开来，必须取测试值，使得 $(E_1, E_2, \cdots, E_n)$ 是各种不同的真值组合。

弱变异测试相对于强变异测试，可以看作是一种测试数据选择的准则。它的优点是，用一组测试数据很可能就会检验出多个程序组成部分的正确性，从而可以减少测试中程序运行的次数。而强变异方法要求的程序运行次数却是随着变异因子的增加而增加的。另外，弱变异的准则对测试数据的选择有一定的指导作用，用户可以有针对性地选择测试数据；而强变异方法对测试数据的选择，其指导作用不大。然而，弱变异测试也有它的不足之处，这表现在由于它重视的对象是程序的组成部分，所以即使当某一组成部分 C_1 有错，而若另有一组成部分 C_2 也有错时，程序在能检验出 C_1 错的测试数据 t 下运行时，可能会由于 C_2 的作用，使程序运行结果正确。而强变异强调的是被测程序，它要求“杀掉”变异因子，所以不会出现这种情况。

我们还可以看出，对算术关系和布尔关系的弱变异准则较之分支覆盖准则强。因此，弱变异准则包含了分支覆盖准则。

为便于测试人员使用变异方法，一些变异测试工具已陆续开发出来。对变异测试工具有兴趣的读者请参阅本书第七章中测试工具综述的相关内容。

参 考 文 献

[1] W. E. Howden, “Weak Mutation Testing and Completeness of Test Sets”, IEEE Transaction Software Eng., July, 1982.

[2] M. R Girigs and M. R. Woodard, “An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis”, Proc. 8th IEEE Int. Conf. Software Eng., London, UK, 1985.

[3] R. A. Demillo et al., “Program Mutation: A New Approach to Program Testing”. Software Testing, Vol 2, Infotech State of the Art Report, 1979.

[4] T.A. Budd et al., “The Design of a Prototype Mutation System for Program Testing”,

Proc. National Computer Conference, 1978.

[5] Edited by P. Depledge, Software Engineering for Microprocessor Systems, Peter Peregrinus Ltd., 1984.

[6] Joe Abbott, Software Testing Techniques, National Computing Centre Limited, 1986.

[7] M. S. Hecht, Flow Analysis of Computer Programs, North-Holland, 1977.

[8] Edited by S S. Muchnick and N. D. Jones, Program Flow Analysis, Prentice-Hall, 1981.

[9] Edited by R. A. Demillo, Foundations of Secure Computation, Academic Press, 1978.

[10] L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, Vol.SE-6, No. 3, May 1980.

[11] D. J. Richardson and L. A. Clarke, "A Partition Analysis Method to Increase Program Reliability", Proceedings 5th International Conference on Software Engineering, 9—12 March 1981, San Diego, California.

[12] Joe Abbott, Software Testing Techniques, The National Computing Centre Limited, 1986.

[13] T. E. Cheatham, G. H. Holloway and J. A. Townley, "Symbolic Evaluation and the Analysis of Programs", IEEE Transactions on Software Engineering, Vol. SE-5(4), July 1979, pp. 402—417.

[14] J. A. Darringer and J. C. King, "Application of Symbolic Execution to Program Testing", Computer,Vol. 11(4), April 1978, pp. 51—60.

[15] J. C. Huang, "An Approach to Program Testing", ACM Computing Surveys, Vol. 7(3), September 1975, pp. 113—128.

[16] Boris Beizer, Software Testing Techniques, Van Nostrand Reinlold Company, 1983.

[17] M. R. Woodward, Hedley D., and M A. Hennell, "Experience with Path Analysis and Testing of Programs", IEEE Transaction on Software Engineering, Vol. SE-6, No. 3, May 1980.

[18] Robert H. Dunn, Software Defect Removal, McGraw-Hill. 1984

[19] R. Glass, Software Reliability Guidebook, Prentice-Hall, 1979.

[20] Infotech State of the Art Report, Software Testing, Vol. 1: Analysis and Bibliography, Infotech International, 1979

[21] J. C. Huang, "Program Instrumentation and Software Testing" Computer, Vol. 11(4),April 1978, pp. 25—31.

附录 2　有关软件测试的术语

以下给出一些软件测试常用的术语及其简要解释。

一　画

一致性 (Consistency)

各组成部分之间的逻辑相关性。一致性也可表明，遵循一组给定的规则。

四　画

认证(Certification)

在软件已被权威机构确认后或是其有效性已向权威机构证实后对软件的认可。

分支测试 (Branch testing)

分支测试是满足特定覆盖准则的方法，该准则要求对于每一个判定点每一可能的分支至少被执行一次。

五　画

正式分析 (Formal Analysis)

使用严格的数学方法分析解的算法。可以对算法的数值性质、效率和正确性进行分析。

正确性 (Correctness)

软件中不含设计缺欠和编码缺欠的程度，即不含故障的程度。同时也是软件满足所规定的需求和满足用户目标的程度。

边界值分析 (Boundary Value Analysis)

边界值分析是一种选择测试数据的方法，按此方法所选的测试数据总是接近于输入域(或输出域)、数据结构、过程参数等的边界或端点，因而常常包括最大值、最小值和一般值或参数。这一方法也叫做强度测试 (Stress testing)。

正确性证明 (Proof of Correctness)

使用数理逻辑方法推出，假定在程序入口处变量间的某个关系是真，隐含着在程序出口处程序变量间的另一关系成立。

功能测试 (Functional Testing)

由特定功能需求导出测试数据而不考虑最后的程序结构的测试。

六　画

有效输入 (Valid Input)

处于程序表示的功能域以内的测试数据。

动态分析（Dynamic Analysis）

使用执行或模拟已开发的阶段产品的方法，通过分析产品对一组输入数据产生的反应来发现错误的方法。

七　画

进化检查（Evolution Checking）

为保证软件产品在由简到繁、逐步细化的过程中各阶段的规格说明保持完整性和一致性所作的测试。

回归测试（Regression Testing）

在软件修改后再次运行以前为查找错误而执行程序曾用过的测试用例。

系统测试（System Testing）

为验证系统满足其规定需求对集成的硬件和软件系统所做的测试。

完备性（Completeness）

问题中所有必要部分均已包括在内的特性。产品的完备性常被用来表示所有的需求均已被满足这一事实。

完整性检查（Integrity Checking）

在每个开发阶段检验软件产品有无缺欠的测试。

八　画

软件（Software）

计算机程序、规程、规则，相关的文件及与计算机系统操作有关的数据。

软件生存期（Software Life Cycle）

从软件产品的最初构想开始，直到产品不能再使用时为止的一段时间。可把软件生存期划分成需求阶段、设计阶段、编码及测试阶段、安装阶段及运行与维护阶段。

驱动程序（Driver）

可以构成测试环境和调用被测模块的代码。

承接程序（Stub）

承接程序是一个特定的代码段。在测试中某一代码段调用它，以便模拟已经设计但尚未完成的模块的行为。

单元测试（Unit Testing）

为消除局部错误、语法错误和逻辑错误，纠正实现中的错误，满足其需求，对模块进行的测试。

组装测试（Integration Testing）

组装测试是按顺序完成的一种测试，测试中要把软件元素、硬件元素或者两者结合在一起，直至所有的模块间联系均已得到检验。

审查（Inspection）

为发现程序错误以十分正规且十分严格的方式检查程序（包括需求、设计或代码）的一种人工分析技术。

九　　画

标准审计 (Standards Audit)

为保证适用的软件得到正确使用的检查。

语句测试 (Statement Testing)

使得程序中每个语句在程序测试时至少执行一次这一准则得到满足的测试方法。

测试 (Testing)

使用抽样数据组(测试用例)执行程序以检验程序的特性。

测试数据集 (Test Data Set)

测试过程中所用的输入元素的集合。

测试驱动程序 (Test Driver)

利用若干测试数据集指引程序执行测试的程序。测试过程中测试驱动程序通常记录输出并组织输出。

十　画

因果图 (Cause Effect Graphing)

画因果图是选择测试数据的一种方法。程序的输入和输出由需求分析决定。一个最小的输入集合被选定,从而避免引起同一输出的多组测试输入。

验证 (Verification)

在开发期的每个阶段及各阶段间论证软件的一致性、完整性及正确性。

验收测试 (Acceptance Testing)

为确定软件系统是否满足验收标准以及使客户决定是否接受而进行的正式测试。

特定测试数据 (Special Test Data)

基于输入值的测试数据,它们可能要求程序给予特定处理。

十　一　画

基于功能测试的设计 (Design Based Functional Testing)

把根据功能分析(参看功能测试)得到的测试数据不仅用于测试需求功能，而且用于测试设计功能。

排错 (Debugging)

纠正测试中发现的法法错或逻辑错的过程。同是为了得到可执行的代码，排错与测试使用了一些共同的方法和策略,但在特定的应用领域和局部范围内两者是不同的。

接口分析 (Interface Analysis)

模块间联系能否正确无误工作的检查。

符号执行或符号演算 (Symbolic Execution or Symbolic Evaluation)

为每一程序路径推演一个符号表达式的分析技术。

断言 (Assertion)

描述必定存在某个程序状态的逻辑表达式或是在程序执行的某个特定点处程序变量

必须满足的一组条件。

十 二 画

插装 (Instrumentation)

在程序中插入附加的代码，以便于程序执行中收集有关程序行为的信息。

黑盒测试 (Black Box Testing)

见“功能测试”。

遍查 (Walkthrough)

由模块开发者向参加审查的同行讲述模块结构及其逻辑的一种人工分析技术。

十 三 画

确认、验证及测试 (VV&T)

为发现错误、确定功能、保证软件产品的质量，在软件生存期中进行复审、分析和测试活动的总称。

路径测试 (Path Testing)

这是一种满足覆盖准则的测试方法，按此准则程序的每一逻辑路径都要测试到。通常将程序的路径分为有限个集合，然后去测试每一集合中的一条路径。

数据流分析 (Data Flow Analysis)

程序执行中变量被初始化、被赋值或被引用，为跟踪这些程序变量各种行为而采用的一种图形分析方法。

十 四 画

静态分析 (Static Analysis)

并不在计算机上执行程序，而直接对软件产品的形式和结构进行分析。可以对需求、设计和代码进行静态分析。

模拟 (Simulation)

用可执行模型显示某个对象的行为特性，在测试中计算机硬件、外部环境，甚至一段代码都可被模拟。

端点测试数据 (Extremal Test Data)

处于输入变量域的端点或边界的测试数据，或是使得到结果处于输出域边界的测试数据。