高等学校计算机应用规划教材

软件测试技术

杨怀洲　编著

清华大学出版社

北　京

内 容 简 介

本书系统地介绍软件测试的基本原理与方法，重点讲解软件测试的基本技术、测试用例的设计方法、软件测试的主要过程、软件缺陷的报告以及测试的评估方法。同时，结合软件测试工程实践，讲解测试项目管理、自动化测试原理以及测试工具的分类和选择。书后附录部分给出了常用软件中测试术语的中英文对照、与测试相关的软件工程国家标准目录、实用的软件测试计划模板和验收测试报告模板，供读者学习参考。

本书融入作者十余年软件工程领域实践与教学经验，内容精炼实用、条理清晰并且通俗易懂。通过丰富的实例和实践要点描述，方便读者理解测试理论和技术的具体应用方法，力求使软件测试初学者可以在短时间内掌握软件测试技术核心内容，为进一步适应高级软件测试工作打下坚实基础。

本书可作为软件工程、计算机科学与技术以及相关专业的本科生教材和硕士研究生参考教材，也可以作为各类软件工程技术相关人员的参考书。

本书对应的课件可以到 http://www.tupwk.com.cn/downpage 网站下载，也可通过扫描前言中的二维码下载。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

软件测试技术 / 杨怀洲 编著. —北京：清华大学出版社，2019
(高等学校计算机应用规划教材)
ISBN 978-7-302-52501-1

Ⅰ. ①软…　Ⅱ. ①杨…　Ⅲ. ①软件—测试—高等学校—教材　Ⅳ. ①TP311.55

中国版本图书馆 CIP 数据核字(2019)第 043098 号

责任编辑：胡辰浩　李维杰
装帧设计：孔祥峰
责任校对：牛艳敏
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市少明印务有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：17.75　　**字　　数**：455 千字
版　　次：2019 年 4 月第 1 版　　**印　　次**：2019 年 4 月第 1 次印刷
定　　价：59.00 元

产品编号：083108-01

前　言

现阶段，软件测试基础人才不足，已成为制约我国软件产业发展的瓶颈。在国内，虽然软件测试仍然处于起步阶段，但是毫无疑问，就 IT 产业发展前景来看，软件测试是软件行业中的朝阳产业。信息产业目前已成为我国的支柱性产业，特别是伴随着“互联网+”战略上升至国家战略，软件行业正在以前所未有的速度蓬勃发展，因此也极大地带动了软件测试行业的快速发展。软件测试对于软件质量保障的重要性越来越多地得到软件企业和软件研发团队的重视，专业的软件测试人才需求不断扩大，各种软件测试培训机构和网站数量不断增多，软件测试已成为 IT 产业中的一个重要行业分支。

但是与软件测试发展和人才需求不相适应的是，很多软件企业认为，大量软件测试岗位应聘者缺乏对于软件测试技术的系统培训，未能系统化地掌握软件测试正规流程，一些应聘者虽然有一些软件研发经验，但是不了解软件测试岗位需求。从软件测试人员的现状来看，也存在着很多问题。测试人员的专业知识不够扎实，只懂得一些表面上的测试技术，不能全面胜任软件测试工作，更无法胜任软件测试项目管理工作；测试人员没有建立相对完整的测试体系概念，对软件测试的基本定义和目的不清晰，不了解如何具体开展软件测试工作；忽视软件测试理论知识，认为理论知识没有用而不去深入理解软件测试的基本原理。软件测试人才知识能力结构不健全的根本原因是人才培养途径不健全，因此，急须加强高等院校软件测试技术相关课程的建设。

软件测试远比人们直观想象的复杂，测试工作具有很高的组织管理和技术难度，测试理论也比较庞杂，具有理论和实践高度联系的特点。软件测试工具相比于软件开发工具来讲，分类更为细致并且数量众多，一个软件项目的测试工作往往需要多种软件测试工具的配合使用才能达到全面和深入测试的效果。同时，高等院校软件测试技术课程的讲授或多或少地受到讲授形式和有限课时的限制，与培训机构动辄 4~6 个月的软件测试培训周期和以实验练习为主的培训方式有很大的不同。

上述实际情况使得高等院校软件测试技术课程的讲授具有一定的挑战性，需要精心编排和组织授课内容，设定合理的授课目标，力争在有限的学时内，使学生掌握软件测试的基本原理、方法和技术，熟悉软件测试的正规流程以及相关标准和规范，了解软件测试项目的管理方法，并且能够学习掌握自动化测试的原理和基本的软件测试工具，为今后更为深入地学习软件测试知识技能和胜任高级软件测试工作打下坚实基础。为此，本书在内容上进行了精心组织，摒弃了一些复杂深奥和实用性不强的理论内容，加强了对软件测试基本技术的讲解，力求通过丰富的典型实例和通俗易懂的语言，使读者快速理解重点内容、切实掌握相关难点。同时，重视理论与实践相结合，根据当前软件测试行业技术应用现状和未来发展趋势，使读者既能够系统地

掌握软件测试的基本理论和方法，又能够明晰这些理论和方法是如何在实际应用中发挥作用的。本书主要包括以下内容：

(1) 测试基础知识。在第 1 章中通过分析软件测试工程师的职业发展前景和当前我国软件测试行业现状，使读者首先了解学习本课程的意义，增强学习兴趣；介绍软件测试的发展历程、基本概念、原则和术语；详细说明软件测试的目的、分类、流程和基本的软件测试过程；细致讲解常见的软件测试模型；阐述什么是测试用例、如何正规书写测试用例、如何保障测试用例的设计质量。

(2) 测试基本技术。在第 2 章和第 3 章中结合经典实例，重点讲解常用的白盒测试技术和黑盒测试技术以及相应的测试用例设计方法，对难以掌握和应用中易错的知识点进行实例化说明。总结和分析白盒测试和黑盒测试的优缺点，在此基础上给出白盒测试和黑盒测试技术的应用策略。

(3) 测试过程。在第 4 章中，从单元测试、集成测试、系统测试和验收测试 4 个阶段详细介绍软件测试执行过程，说明各测试阶段依据的主要技术文档、参与人员、典型测试数据和采用的主要技术，对回归测试的方法和注意事项进行介绍。

(4) 功能与非功能测试。在第 5 章中，对各种典型的功能和非功能测试技术进行说明，重点讲解性能测试的分类以及常用的性能测试指标。

(5) 缺陷报告与测试评估。在第 6 章中，详细说明报告软件缺陷的方法，重点说明如何完成定量化测试评估，介绍测试总结报告的编写方法。

(6) 测试管理。在第 7 章中，介绍测试管理中一些最为重要的管理内容和相关知识，主要包括软件质量管理标准和管理体系、如何制定测试计划、测试项目中的测试文档以及测试配置管理等内容。

(7) 软件测试自动化。在第 8 章中，介绍自动化测试的原理，说明测试工具的分类和选择方法，给出一些常用测试工具的说明。

本书在编写过程中参考了很多专著、教材、论文和大量的网上资料，由于篇幅所限，一些细节之处未能一一列出。在此，向所有作者表示衷心的感谢和诚挚的敬意。由于作者专业水平有限，书中难免有缺点和欠妥之处，恳请读者批评指正，以便于今后不断修正和改进。我们的电话是 010-62796045，信箱是 huchenhao@263.net。

本书对应的课件可以到 http://www.tupwk.com.cn/downpage 网站下载，也可通过扫描下方的二维码下载。

作　者

2018 年 12 月

目　录

第 2 章

白 盒 测 试

本章从静态测试和动态测试两个方面介绍白盒测试基本技术。静态白盒测试主要包括代码检查和静态结构分析两种方法。动态白盒测试主要包括程序插桩、逻辑覆盖测试、基本路径测试、循环结构测试等。动态白盒测试方法是本章重点内容，也是白盒测试中发现软件缺陷的主要手段。其中，逻辑覆盖测试和基本路径测试方法是实际工作中最常用到的两种动态白盒测试技术，本章将通过多个示例对上述两种白盒测试方法进行详细说明。

2.1 对于白盒测试的基本认识

在第 1 章介绍的软件测试分类中，我们已经简单介绍了什么是白盒测试。在这里，我们给出对于白盒测试更为详细的说明，以便于加深对白盒测试的认识。

白盒测试一般用来分析程序的内部结构，因此有时也被称为基于程序的测试。白盒测试的前提条件是已知程序的内部工作过程，清楚其语句、变量状态、逻辑结构和执行路径等关键信息，因此也被称为玻璃盒测试。白盒测试主要是根据程序内部的逻辑结构和相关信息，检验程序中的各条通路是否都能够按设计要求正确工作，从这个意义上讲，白盒测试又常被称为结构测试或逻辑驱动测试。当然，白盒测试还包括对程序所有内部成分和内部操作的检查过程，例如代码评审也属于白盒测试，是对编码规范性和正确性的综合检查过程。

白盒测试针对的是程序的内部结构和运行过程，可以对程序的每一条语句、每一个条件、每一个分支甚至每一条路径进行测试。白盒测试重视测试覆盖率的度量，被看作“基于覆盖的测试”，要求对被测程序的结构能够做到一定程度的覆盖，通过不同类型的覆盖准则来判断测试执行的充分性。白盒测试有利于引导测试人员向提高覆盖率的方向完成测试工作，不断发现那些潜在的程序错误。理论上如果有充分的时间，白盒测试能够保证所有的语句和条件得到测试，使测试的覆盖程度达到很高。

白盒测试主要用于测试过程早期的单元测试阶段，是进一步完成功能测试和性能测试的基础，基本测试原则包括：

- 保证程序模块中的所有独立路径都至少使用一次；
- 保证程序中的所有逻辑值都能测试 True 和 False 两种情况；
- 在循环的边界和运行的界限内执行循环体；

- 测试程序内部数据结构的有效性以及完成边界数据取值情况下的测试。

白盒测试用例的设计经常需要参考软件详细设计说明，对测试人员软件编程经验和综合业务能力的要求很高，白盒测试工程师已经属于高级测试工程师。另外，满足一定覆盖准则和覆盖率的白盒测试经常比程序编码本身所花费的时间还要长，因此测试成本很大，往往给白盒测试的正常实施带来一定的困难。

需要注意的是，白盒测试方法试图穷举所有程序路径进行测试，这往往是不可能的。图 2-1 所示的程序流程图包含 30 次循环，虽然程序流程结构很简单，但是可能存在的执行路径数高达 3^{30}。如果对所有路径进行穷举测试，假设测试每条路径需要 1ms，总共需要约 6528 年。因此，需要根据特定的原则设计测试用例，使得用例数量尽可能少。

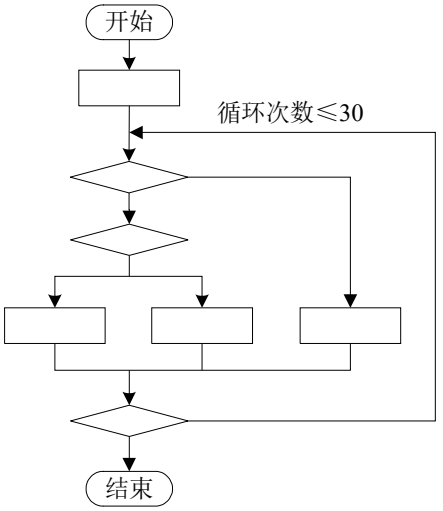


图 2-1　穷举路径测试示例

当然，产生上述天文数字路径数的原因是程序包含循环结构。那么，如果不考虑循环结构，100%覆盖所有路径，是否就能够发现所有程序问题呢？答案是令人失望的，程序仍然可能存在错误！原因是穷举路径测试查不出违反设计规范的错误，不能发现程序中已实现但不是用户所需要的功能，不可能查出程序中因遗漏路径而产生的错误，可能发现不了一些与数据相关的异常错误。

因此，尽管白盒测试方法深入程序内部，针对程序细节的逻辑结构进行测试，对代码的测试比较彻底，但仍然存在着一定的局限性。

2.2 静态测试

根据测试时是否运行源程序，白盒测试可以分为静态测试和动态测试，而静态测试方法又主要分为代码检查和静态结构分析等。静态测试就是不实际运行被测试的软件，而只是静态地检查程序代码、界面或文档中可能存在的错误的过程。

2.2.1 代码检查法

代码检查法主要包括桌面检查、代码走查和代码审查，主要检查代码的规范性、可读性、结构的合理性、逻辑表达的正确性等内容。实践证明，代码检查比动态测试更为有效，能快速发现30%~70%的逻辑设计和编码缺陷，应当在程序编译和动态测试之前进行。但是，代码检查对技术能力和经验的要求很高，并且非常耗时。表2-1给出了常见的三种代码检查法的对比。

表2-1 桌面检查、代码走查和代码审查的对比

项目	桌面检查	代码走查	代码审查
准备	程序的规格说明、编码规范、错误列表、源代码	参加人员事先阅读设计和源代码，准备代表性测试用例	需求与设计文档、源代码、编码规范、缺陷检测表、会议计划和流程
形式	无	非正式会议	正式会议
参加人员	程序编写者本人	开发组内部人员	开发、测试和相关人员
主要技术方法	无	逻辑运行测试用例	缺陷检测表
注意事项	注释与编码规范	限时、不当场修改代码	限时、不当场修改代码
生成文档	无	静态分析错误报告	结果报告
目标	无	代码标准规范、无逻辑错误	代码标准规范、无逻辑错误
优点	省时	便于项目组成员交流，共同理解软件产品	有计划地对软件产品进行编码质量控制
缺点	不正式、依赖个人能力、效率低	耗时	耗时

1) 桌面检查

桌面检查是最不正式，也是最省时的静态测试技术。桌面检查就是程序员对自己的代码进行一次自我检查，对编码成果进行自我完善。程序员根据程序的规格说明、编码规范、常见错误列表等，仔细阅读源代码，发现程序中的问题和错误。由于桌面检查没有任何约束，依赖程序员个人的经验和技术能力，因此对于大多数人而言，检查效率很低。由编程者本人完成的桌面检查明显违背了软件测试的独立性原则,因此最好由其他编程人员通过伙伴检查的方式进行。即便如此，其效果仍然远远逊色于代码走查和代码审查。

通常桌面检查包含以下内容：

- 对变量和标号的交叉引用表的检查。检查变量的定义和使用以及转向特定位置的标号；
- 对子程序、宏、函数的检查；
- 等价性检查。检查全部等价变量类型的一致性；
- 常量检查；
- 设计标准检查。检查程序是否违反设计标准；
- 风格检查；
- 控制流检查；
- 选择、激活路径检查。检查每条控制流路径是否都能被程序激活，达到语句覆盖；
- 规格符合性检查。检查是否符合程序规格说明以及编码规范；
- 补充文档检查。

2) 代码走查

代码走查的过程是非正式的，一般在开发组内部进行，通过代码走查小组，以会议的方式来检查代码。小组成员一般提前阅读设计规格书、源程序等文档，准备一些代表性的测试用例，通过逻辑运行程序的方式共同交流、讨论和发现程序问题。借助典型测试用例可以帮助发现程序在逻辑和功能上的问题，但是对所发现的问题并不做现场修改。

代码走查有利于项目组人员共同理解项目所涉及的业务信息和具体代码实现过程，交换代码编写思路，帮助开发人员找出程序错误和解决方法。很多软件缺陷并非需要到运行程序进行测试时才能发现，通过合理的代码走查可以发现和解决相当多的程序问题。

3) 代码审查

代码审查是一种正式的评审活动，通过正式会议的方式进行，事先一般具有制定好的会议计划和流程，会议中应用预先定义好的标准和检查技术检查程序和文档，发现软件缺陷，会后形成正式的审查结果报告。与代码走查相比，代码审查也是通过组成审查小组的形式对程序进行检查，但是组成人员更多，包括项目开发组、测试组和相关人员(QA、产品经理等)。

会议前，审查小组成员需要提前阅读需求和设计规格说明书、源程序等文档。另外，还需要准备一份缺陷检查表，在表中分类列出所有可能发生的错误供审查小组对照检查。开发人员是会议中检查项目的生成者，因此一般由开发人员负责提供有关检查项目资料，并且在会议过程中回答审查小组成员的问题。程序员在会议中讲解程序的逻辑实现，审查小组通过提问、讨论和争论的方式促进问题的暴露，往往能够发现 30%~70%的逻辑设计错误和编码错误。与代码走查一样，为了节约时间不跑题，避免无休止的争论，代码审查限时并且不对所发现的问题进行现场修改。

图 2-2　Mark Chu-Carroll 博士

代码审查是软件开发过程中必不可少的环节。谷歌前资深软件开发工程师 Mark Chu-Carroll 博士(见图 2-2)认为，Google 的程序之所以如此优秀的一个重要原因看起来很简单：代码审查。在 Google，没有任何项目的程序源代码可以在没有经过有效的代码审查前就提交到代码库中。代码审查最重要的作用和需要注意的事项为：

- 因为知道存在代码审查，编码者编写代码更为规范；
- 代码审查能传播知识，使模块编写者之外的审查者也能熟悉程序的设计和架构；
- 确保程序作者自己写出的代码是正确的；
- 不应过于匆忙地完成代码审查；
- 需要遵循严格的编码规范。

2.2.2　静态结构分析法

静态结构分析法实际上是通过白盒测试工具辅助进行程序检查的一种方法。阅读和理解代码是一件很困难的工作。代码以文本格式写成，包含很多复杂的数据结构和逻辑结构，即使是非常符合编码规范的源代码，理解起来也具有一定的困难。研究表明，程序员 38%的时间都被用于理解源代码。白盒测试工具能够在分析程序的基础上提供各种直观的图表，全面详细地展

示程序各方面的情况，使开发和测试人员更容易找出其中的问题。

在静态结构分析中，测试人员通过测试工具分析程序的系统结构、数据结构、数据接口、控制逻辑等内部结构，生成函数调用关系图、程序控制流图、内部文件调用关系图、子程序表、宏和函数参数表等各类图表，可以清晰地呈现整个系统的组成结构，方便阅读和理解。通过分析这些图表，测试人员可以快速和有效地发现程序中潜在的错误。表 2-2 给出了主要的静态结构分析图表及其内容与作用。

表 2-2 静态结构分析图表及其内容与作用

分类	名称	内容与作用
图	函数调用关系图	● 列出所有函数，用连线表示调用关系，展示系统的结构 ● 发现系统是否存在结构缺陷、区分函数的重要程度、确定测试覆盖级别 ● 检查函数的调用关系是否正确 ● 是否存在递归调用 ● 函数的调用层次是否过深 ● 检查是否存在孤立而未被调用的函数 ● 确定函数调用频度，重点检查被频繁调用的函数
	模块控制流图	● 由节点和边组成，每个节点代表一条或多条语句，边表示控制流 ● 能够直观地反映模块的内部逻辑结构
表	标号交叉引用表	● 列出所有模块中用到的标号 ● 标号的属性，包括已说明、未说明、已使用、未使用 ● 模块以外的全局标号、计算标号
	变量交叉引用表	● 展示所有变量的定义和引用情况 ● 变量的属性，包括是否已说明、是否已使用、类型、是否属于公共变量、全局变量等
	子程序(宏、函数)引用表	列出所有子程序、宏和函数的属性，包括类型、是否已定义、是否已引用、引用次数、输入输出参数的数量、顺序和类型
	等价表	列出在等价语句或等值语句中出现的全局变量和标号
	常数表	列出所有数字和字符常数

借助图表信息，测试人员可以完成如下静态错误分析：

- 数据类型和单位分析；
- 引用分析。找出变量引用错误，例如变量赋值以前被引用或赋值后未被引用；
- 表达式分析。发现表达式中括号使用不正确、数据下标越界等错误；
- 接口分析。检查模块之间接口的一致性，以及模块与外部数据库之间接口的一致性。

2.3 程序插桩

在白盒测试中，程序插桩是一种最基本的动态测试手段，有着广泛的应用。

程序插桩技术，是在保证被测程序原有逻辑完整性的基础上在程序中插入一些语句，这些语句被称为“探针”“探测器”或“探测点”，其本质就是进行信息采集的代码段。通过探针

的执行，记录和显示程序语句的执行情况、变量的变化等特征数据。通过对这些数据的分析，可以获得程序的控制流和数据流信息，进而得到逻辑覆盖等动态信息，从而对程序运行状态和运行逻辑的正确性进行判断，发现其中的问题。

测试人员常常借助程序插桩的方法来收集程序动态运行行为，一些与运行环境相关的程序行为只能通过程序插桩的方法来收集，静态程序分析无法完成这样的工作。通过程序插桩技术，能够获取各种程序信息，是对程序进行白盒测试的一种有效手段。例如，如果想知道程序中所有语句是否都被执行，也就是语句或程序分支覆盖的情况，或者想了解每个语句的实际运行次数，就可以利用程序插桩技术。

图 2-3 是计算两个整数的最大公约数的程序插桩示例，左边是函数源程序，右边是程序的流程图。最大公约数是指两个或多个整数共有约数中最大的一个，例如，12 和 30 的最大公约数是 6，算法的含义在这里并不重要。虚线框代表在源程序中插入的一些探针语句，用于记录语句执行的次数，是一些计数器，可以用数组的方式实现。

```
int Get_GCD(int X,int Y)
{
 while(X!=Y)
 {
 if(X>Y)
     X=X-Y;
  else
    Y=Y-X;
  }
  return X;
}
```

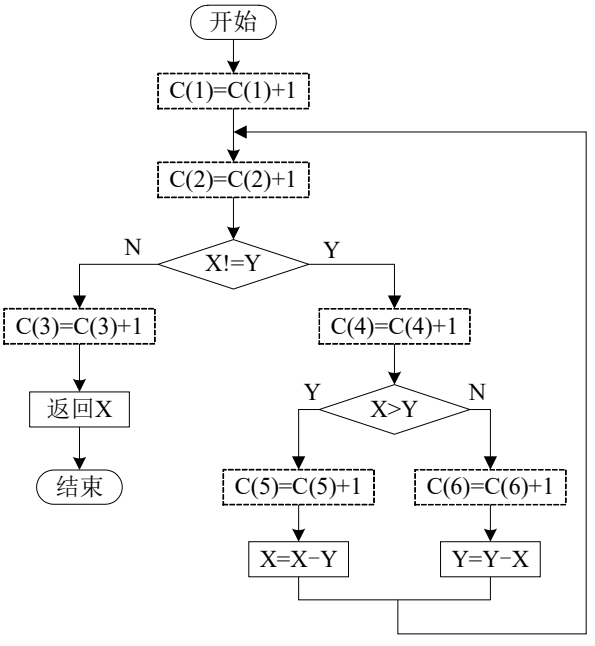


图 2-3　最大公约数计算函数的程序插桩过程

C(1)用于记录函数被调用的次数，C(2)用于记录循环执行的次数，C(3)是函数出口计数器，C(4)和 C(5)是主要程序分支上的计数器。如果在程序开始时插入对计数器数组的初始化程序，在程序返回前输出各计数器的值，就构成了完整的插桩程序。

在程序插桩时，需要注意的要点如下。

1) 需要探测哪些信息

这需要根据具体的测试目标来决定。例如，如果需要知道各种覆盖标准下对应元素的覆盖情况，就需要探测相应元素是否运行；如果需要知道程序运行到每个位置的执行结果是否正确，就需要输出该位置的特定信息。

2) 在什么位置设置探测点

通常在如下位置设置探测点：

- 程序的第一条可执行语句之前，用于判断程序是否被执行；
- 有标号的可执行语句之前；
- for、while、do until 等循环语句处；
- if、then、else 等条件分支语句处；
- 输入语句之后，用于检验输入数据的正确性；
- 输出语句之前，用于检验将要输出的数据是否正确；
- 函数、过程等程序调用语句之后，用于判断调用结果是否正确；
- return 语句之前，判断程序是否正常返回。如果探针设置在 return 语句之后，它将无法被执行。

3) 需要设置多少个探测点

一般情况下，在没有分支的程序段中只需要在首尾各设置一个探测点，用于确定程序执行时该段程序是否被覆盖。这样的程序段由一些顺序执行的语句组成，又称为“顺序块”。如果顺序块的第一条语句被执行，那么其他语句也会被执行。因此，无须对每条语句都进行插桩操作。但是，如果程序中有各种分支控制结构，如各种循环和条件判断分支结构，那么为了插入最少的探测点，需要针对程序的控制结构进行具体分析。

4) 如何在程序的特定位置插入用于判断变量特性的语句

通过程序插桩技术，可以在程序的特定位置插入某些用于判断变量特性的断言语句，以便证实程序运行时的某些特性，从而帮助排除故障。这种方法是证明程序正确性的基本步骤，虽然不是一种严格证明方法，但具有一定的实用性。

需要注意的是，程序插桩并不是独立的白盒测试方法，一般要和诸如覆盖测试等方法结合起来使用。在实现程序覆盖测试时，经常需要获得一些特定信息，例如：程序中语句被执行(也就是被覆盖)的情况，程序运行的路径，变量的定义和引用情况等。要想获得这些信息，就需要在被测程序中插入完成相应工作的代码，即运用代码插桩技术，如今大多数的覆盖测试工具均采用代码插桩技术。

还需要注意的是，代码插桩虽然不影响程序的逻辑结构和复杂性，但是会破坏程序的时间特性。因此，在用程序插桩辅助完成一些性能监视测试工作时，有时需要考虑插桩代码对程序运行效率的影响。

2.4 逻辑覆盖测试

逻辑覆盖测试是一种常用的动态白盒测试方法，主要包括语句覆盖、判定覆盖、条件覆盖、判定-条件覆盖、条件组合覆盖和路径覆盖。逻辑覆盖是基于程序的内部逻辑结构进行的测试，要求在设计测试用例时，对被测程序的逻辑结构有清晰的了解。

在图 2-4 中，左边是源程序，右边是程序流程图。流程图中的字符“a~e”标记了程序逻辑结构的节点，用于表示程序运行经过的路径。我们将这个程序作为被测程序，讲解上述几种常见的逻辑覆盖技术。

```
float js(float A, float B, float X)
{
   if(A>1&&B==0)
      X=X/A;
   if(A==2||X>1)
      X=X+1;
   return X;
}
```

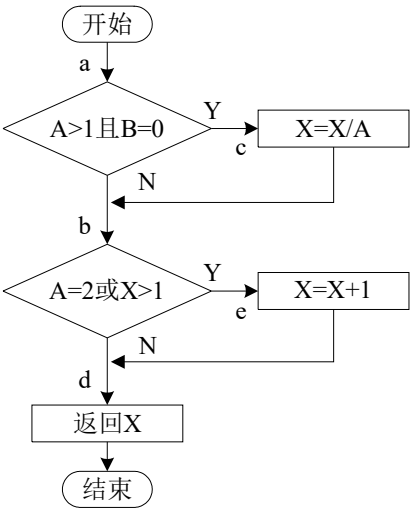


图 2-4　逻辑覆盖测试的被测程序及其流程图

在逻辑覆盖测试中，决定程序分支走向的整体布尔型表达式被称为判定，取值为 True 或 False。判定不考虑其内部是否包含“与”“或”等逻辑操作符。上述例程中包含两个判定：

- “A>1 且 B=0”，为方便表达记为 P1；
- “A=2 或 X>1”，记为 P2。

但是，大部分的判定表达式是由多个逻辑条件组合而成的。虽然上述例程中只有 3 个变量，但是却包含 4 个条件表达式：

- “A>1”，记为 C1；
- “B=0”，记为 C2；
- “A=2”，记为 C3；
- “X>1”，记为 C4。

2.4.1　语句覆盖

语句覆盖的含义是指，设计若干个测试用例，使被测程序中的每一条可执行语句至少执行一次。

需要说明的是，为了尽可能减少设计、实施和维护测试用例的成本，逻辑覆盖测试用例的数量应当越少越好，只要能够满足相应的覆盖标准即可。关于被测程序中语句的个数统计，图 2-4 所示程序中共有 3 条可执行语句，分别以分号结尾。“if (A>1&&B==0)”和“if(A==2|| X>1)”只是判定条件而不是语句。

对于图 2-4 中的被测程序来讲，为了使程序中的每条语句都被执行一次，只需要选取一组测试用例输入数据，使得程序沿着路径“a-c-b-e-d”运行即可。例如，可以选取(A=2，B=0，X=3)作为输入数据，这样，P1 和 P2 两个判定的值都为 True，3 条语句都被执行。这样的测试用例即可达到语句覆盖的标准。当然，测试用例不是唯一的，例如，(A=2，B=0，X=2)同样满足要求。

语句覆盖是最弱的逻辑覆盖标准。运行测试用例(A=2，B=0，X=3)，虽然能够执行所有语句，但是不能覆盖所有的判定分支。例如，判定 P1 和 P2 的 False 分支都没有被覆盖到。因此，

语句覆盖只针对程序中显式存在的语句，而无法测试隐藏的条件和可能的逻辑分支。例如，如果判定 P1 中的“&&”被错误写成“||”，上述测试用例仍可覆盖所有语句。又例如，如果判定 P2 中的“X>1”被错误写成“X>0”，上述测试用例仍然无法发现这个错误。

2.4.2 判定覆盖

判定覆盖又称为分支覆盖，是指设计若干个测试用例，使被测程序中每个判定的取真分支和取假分支至少被执行一次，即每个判定的真假值均被满足。

对于上述被测程序，设计如表 2-3 所示的判定覆盖测试用例，使程序执行路径“a-c-b-e-d”和“a-b-d”。这样，判定 P1 和 P2 的真假分支就都能被执行到。

表 2-3 判定覆盖测试用例(一)

测试用例	P1	P2	执行路径
A=2，B=0，X=4	T	T	a-c-b-e-d
A=3，B=1，X=1	F	F	a-b-d

或者，换一种设计思路，设计如表 2-4 所示的测试用例，使程序执行路径“a-c-b-d”和“a-b-e-d”。这样，判定 P1 和 P2 的真假分支也同样都能被执行到。

表 2-4 判定覆盖测试用例(二)

测试用例	P1	P2	执行路径
A=3，B=0，X=1	T	F	a-c-b-d
A=2，B=1，X=3	F	T	a-b-e-d

需要注意的是，程序中包含两种类型的判定语句。一种是双值判定语句，取值是 True 或 False。例如，If-Then-Else、Do-While、Do-Until 等。另一种是多值判定语句，例如 Case 语句。因此，判定覆盖更一般的含义是设计测试用例，使每一个分支获得每一种可能的结果。

判定覆盖比语句覆盖具有更好的测试充分性。相比语句覆盖测试用例，判定覆盖测试用例驱动被测程序执行几乎多一倍的测试路径。由于可执行语句要不就在判定的真分支上，要不就在判定的假分支上；因此，只要满足判定覆盖标准的测试用例，就一定满足语句覆盖标准，反之则不然。

判定覆盖的测试充分性仍然很弱，它只是判断整个判定表达式的最终取值结果，而不考虑表达式中每个条件的取值情况，因此必然会漏检一些条件错误。判定表达式往往由多个条件组合而成，某个条件的取值结果可能会掩盖其他条件的取值结果情况。例如，“或”关系的第一个条件为真则不再判断第二个条件，“与”关系的第一个条件为假则不再判断第二个条件。在上述程序中，将判定 P2 中的条件“X>1”错误写成“X<1”，使用表 2-3 所示的测试用例仍然能够达到判定覆盖标准，无法发现该错误。

2.4.3 条件覆盖

条件覆盖是指，设计足够多的测试用例，使每个判定中每个条件的真假取值都至少被满足

一次。

上述程序中，4 个条件 C1~C4 中的每一个都有真假两种取值可能，分别为：

- C1 取真值(即 A>1)记为 T1，取假值(即 A≤1)记为 F1；
- C2 取真值(即 B=0)记为 T2，取假值(即 B≠0)记为 F2；
- C3 取真值(即 A=2)记为 T3，取假值(即 A≠2)记为 F3；
- C4 取真值(即 X>1)记为 T4，取假值(即 X≤1)记为 F4。

对于被测程序，设计如表 2-5 所示的条件覆盖测试用例，使程序中的每个条件都满足真假两种取值情况。

表 2-5　条件覆盖测试用例(一)

测试用例	C1	C2	C3	C4	P1	P2	执行路径
A=2，B=0，X=4	T1	T2	T3	T4	T	T	a-c-b-e-d
A=1，B=1，X=1	F1	F2	F3	F4	F	F	a-b-d

条件覆盖一般比判定覆盖要强，因为它更为细致地考虑了判定表达式中每个条件的取值情况。但需要注意的是，虽然表 2-5 所示的测试用例也同时满足判定覆盖标准，但是满足条件覆盖标准的测试用例并不能总是保证满足判定覆盖标准。这是由于，条件覆盖只考虑每个条件都取得真假两种值，而不考虑所有的判定结果取值情况。例如，表 2-6 所示的测试用例满足条件覆盖标准，但是由于判定 P2 只有取真值一种情况，其 False 分支未被执行，因此不满足判定覆盖标准。

从测试充分性来讲，既然条件覆盖标准不能完全包含判定覆盖标准，那么也就不能保证达到 100%的语句覆盖标准了。

表 2-6　条件覆盖测试用例(二)

测试用例	C1	C2	C3	C4	P1	P2	执行路径
A=2，B=0，X=1	T1	T2	T3	F4	T	T	a-c-b-e-d
A=1，B=1，X=4	F1	F2	F3	T4	F	T	a-b-e-d

2.4.4　判定-条件覆盖

由判定覆盖和条件覆盖可知，条件覆盖不一定包含判定覆盖，反之亦然。因此，需要一种能将两者结合起来的逻辑覆盖标准，这就是判定-条件覆盖，也称为分支-条件覆盖或条件判定组合覆盖。其基本思想是，设计足够多的测试用例，使被测程序中每个判定的每个条件的可能取值至少被执行一次，并且每个可能的判定结果也至少被执行一次。

细心的读者可能已经发现，表 2-5 所示的测试用例本身就已经满足判定-条件覆盖标准，因为条件 C1~C4 和判定 P1 与 P2 都取得真值和假值两种情况，在这里不再给出其他判定-条件覆盖测试用例。

从表面看，判定-条件覆盖测试所有判定和条件的取值，但事实并非如此。无论是“与”关系逻辑表达式还是“或”关系逻辑表达式，某些条件的取值都可能会掩盖另一些条件的取值情

况，判定-条件覆盖测试并没有覆盖所有条件的真假取值组合情况。因此，判定-条件覆盖并不一定能够查出逻辑表达式中的所有错误。

从测试充分性来讲，满足判定-条件覆盖就一定能够满足条件覆盖、判定覆盖和语句覆盖。

2.4.5 条件组合覆盖

条件组合覆盖是指，设计足够多的测试用例，使被测程序中每个判定的所有可能的条件取值组合至少被执行一次。条件组合覆盖与条件覆盖的区别是，不仅要求每个条件都能有真假两种取值结果，而且要求这些结果的所有可能组合都至少出现一次。

表 2-7 给出了上述被测程序的 8 种条件取值组合情况。

表 2-7 条件取值组合情况

组合编号	1	2	3	4	5	6	7	8
条件取值组合	T1，T2	T1，F2	F1，T2	F1，F2	T3，T4	T3，F4	F3，T4	F3，F4

这里需要注意以下几点：

- 条件取值组合只针对同一个判定表达式内存在多个条件的情况，将这些条件的取值进行笛卡尔乘积组合；
- 不同判定表达式内的条件取值之间无须组合；
- 对于单条件的判定表达式，只需要满足自己的所有取值即可。

根据表 2-7 所示的条件取值组合情况，可以设计表 2-8 所示的条件组合覆盖测试用例。从表 2-8 中的“覆盖条件组合”项可以看出，8 种条件取值的情况都被覆盖到了。

表 2-8 条件组合覆盖测试用例

测试用例	C1	C2	C3	C4	覆盖条件组合	P1	P2	执行路径
A=2，B=0，X=4	T1	T2	T3	T4	1，5	T	T	a-c-b-e-d
A=2，B=1，X=1	T1	F2	T3	F4	2，6	F	T	a-b-e-d
A=1，B=0，X=2	F1	T2	F3	T4	3，7	F	T	a-b-e-d
A=1，B=1，X=1	F1	F2	F3	F4	4，8	F	F	a-b-d

条件组合覆盖是一种很强的覆盖标准，能够有效地测试各种条件取值组合是否正确。同时，从表 2-8 中的“P1”和“P2”项也可以看出，条件组合覆盖还可以覆盖所有判定的真假分支。也就是说，条件组合覆盖标准能够完全包容判定-条件覆盖标准。

但是，条件组合覆盖也线性增加了测试用例的数量，提高了测试用例设计、实施和维护的成本。即便如此，条件组合覆盖仍然可能漏测部分程序可执行路径，测试还不够充分。例如，被测程序中有 4 条可执行路径，分别如下。

- 路径 1：a-c-b-e-d。
- 路径 2：a-b-d。
- 路径 3：a-c-b-d。

- 路径 4：a-b-e-d。

表 2-8 所示的测试用例中只有 3 条执行路径，路径“a-b-e-d”是重复的，漏测了可执行路径“a-c-b-d”。

2.4.6 路径覆盖

路径覆盖就是设计足够多的测试用例，使被测程序的每条可执行路径都至少执行一次。被测程序中有 4 条可执行路径，因此表 2-9 所示的路径覆盖测试用例由 4 个测试用例构成。

表 2-9 路径覆盖测试用例

测试用例	C1	C2	C3	C4	P1	P2	执行路径
A=2，B=0，X=3	T1	T2	T3	T4	T	T	a-c-b-e-d
A=1，B=1，X=1	F1	F2	F3	F4	F	F	a-b-d
A=3，B=0，X=3	T1	T2	F3	F4	T	F	a-c-b-d
A=2，B=1，X=1	T1	F2	T3	F4	F	T	a-b-e-d

路径覆盖是经常使用的覆盖测试方法，相比于其他逻辑覆盖方法，测试覆盖率最大。但是路径覆盖并不一定能保证条件组合覆盖，例如上面的测试用例中，“F1，T2”和“F3，T4”两种条件取值组合情况就未能覆盖到。另外，路径覆盖也不一定能保证条件覆盖。举一个简单的例子，假设被测程序仅含有一个判定表达式 P：(C1 or C2)，因此程序中有两条可执行路径，可以通过两个条件取值分别为(T1，F2)和(F1，F2)的测试用例驱动被测程序分别沿着判定 P 的真假分支执行，满足路径覆盖标准。但是因为条件 C2 没有取真值 T2 的情况，所以不满足条件覆盖标准，进而也就不满足判定-条件覆盖标准。由于路径覆盖必然经历所有判定的各个分支，因此路径覆盖能够完全包容判定覆盖和语句覆盖。

另外需要注意的是，随着代码复杂度的增加，程序可执行路径的数量可能呈指数级增长。例如包含 10 条 if 语句的程序，可执行路径可达到 2^{10}=1024 条。如果被测程序中包含循环结构，随着循环嵌套层次和循环次数的增加，程序可执行路径数可能达到天文数字。这种情况下，一般通过 Z 路径覆盖方法进行测试。

通过以上 6 种逻辑覆盖测试方法的讲解我们会发现，没有十全十美的覆盖测试方法，每一种方法都有优点和局限性。因此，在实际的测试用例设计过程中，需要根据实际情况将几种逻辑覆盖测试方法配合使用，以达到最高的覆盖率。实际工作中，语句覆盖、判定覆盖和路径覆盖使用最多，一般有如下要求。

- 语句覆盖率：100%。
- 判定覆盖率：85%以上。
- 路径覆盖率：80%以上。

基于对上述 6 种逻辑覆盖方法的测试充分性的分析，可以将它们之间的强弱关系用图 2-5 表示。

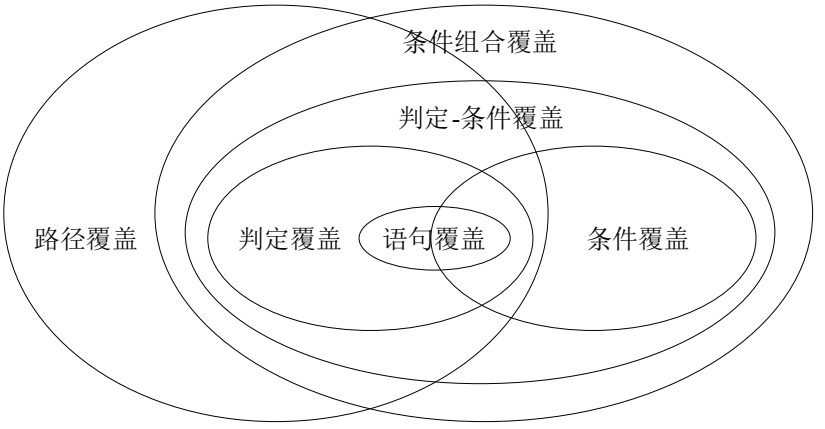


图 2-5　6 种逻辑覆盖测试的强弱关系

2.4.7　Z 路径覆盖

Z 路径覆盖是一种简化循环意义下的路径覆盖测试方法。

普通的路径覆盖是要设计若干测试用例，用来覆盖程序中所有可能的执行路径。当程序比较小，只有为数不多的选择结构时，实现路径覆盖是可以做到的。但是，当程序中含有多个循环或循环次数很多时，可能的执行路径数量将以指数级增长，往往达到天文数字(参考图 2-1 所示的穷举路径测试示例)，想要实现完全的路径覆盖是不可能的。

为了解决这一问题，Z 路径覆盖舍掉了路径覆盖的一些次要因素，对循环机制进行了简化。通过限制循环的次数，最大化地减少路径的数量，使得覆盖这些有限的路径成为可能。无论循环的形式和循环体实际执行的次数如何，在 Z 路径覆盖测试中，只考虑执行循环体一次和零次两种情况，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。

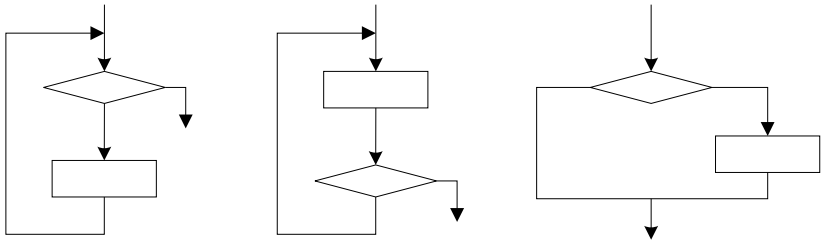


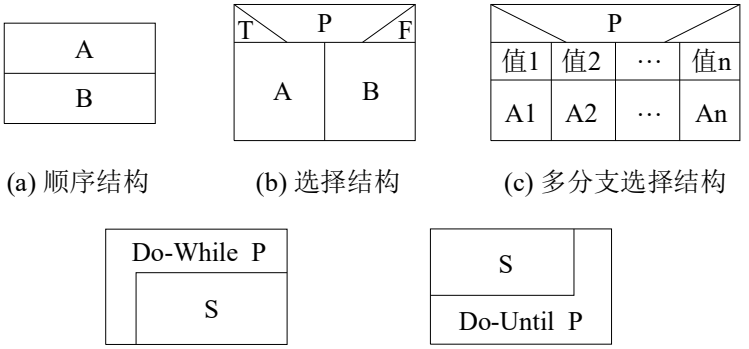
图 2-6　将循环结构简化成选择结构

图 2-6(a)和图 2-6(b)是两种典型的循环结构。前者先做判断，循环体只考虑执行一次或不执行，效果与图 2-6(c)是一样的。后者先执行循环体，也只考虑执行一次，然后再经判断转出，效果与图 2-6(c)中只执行选择结构的右分支一样。经过 Z 路径覆盖方法对循环结构进行简化后，程序中只存在顺序结构和分支结构，其所包含的路径数一般是有限的，因此可以做到对这些路径的覆盖。

2.4.8 计算路径覆盖最少的测试用例数

由于路径覆盖是常用测试方法，因此对于一个较为复杂的被测程序来讲，我们希望能够快速计算出其可执行路径的数量，确定路径覆盖最少的测试用例数，然后基于此数量设计足够多的测试用例，使被测程序的每条可执行路径都至少执行一次。在这里，提供一种计算路径覆盖最少的测试用例数的方法，通过程序盒图(也称为 N-S 图)对该方法进行说明。

图 2-7 中包含 3 种基本逻辑结构：顺序、选择和循环结构。顺序结构无论语句多少，只有一条可执行路径；If-Then-Else 型选择结构包含两条可执行路径；多分支 Case 型选择结构的可执行路径数量由分支数量决定。



(a) 顺序结构　(b) 选择结构　(c) 多分支选择结构

(d) 当型循环结构　(e) 直到型循环结构

图 2-7　由盒图表示的程序的基本逻辑结构

对于循环结构来讲，在 Z 路径覆盖中已经讲过，为了避免测试超大数量的循环路径，在路径覆盖测试时一般将循环结构转变和替换为选择结构，只考虑直接跳过循环体和执行一次循环体这两种情况，并不测试重复执行循环体的路径。因此，我们只需要弄清楚(包含选择结构的)程序的可执行路径数量的计算方法即可。

我们先来观察一个简单的串行选择结构，其盒图如图 2-8 所示。

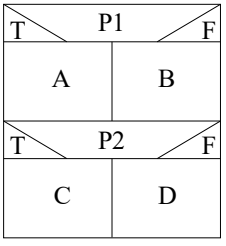


图 2-8　串行选择结构

在这个例子中，可以很容易地看出，程序包含 4 条可执行路径，分别经过语句(块)AC、AD、BC 和 BD。因此，最少的路径覆盖测试用例数为 4。实际上，这样的计算结果是根据“串行分层相乘”的方法计算出来的。图 2-8 所示的程序盒图可以分为上下两层，上层的 P1 选择结构包含 2 条路径，下层的 P2 选择结构地包含 2 条路径，将它们相乘就得到计算结果 4。对于盒图选择结构中并行的语句(块)，如 A 和 B、C 和 D，将它们的数量相加就可以得到该并行层次的路径数。

应用上述计算原理，可以快速计算出更为复杂的程序的可执行路径数。

图 2-9 所示的程序盒图中包含两组串行的选择结构：P1 与 P8，P2 与 P6，分别用分层线 1 和 2 标识。P8 包含的嵌套选择结构的路径数为 3，将它与 P1 包含的路径数相乘即可得到最终路径数。P2 包含的嵌套选择结构的路径数为 5，P6 包含的嵌套选择结构的路径数为 3，将二者相乘后得到 P1 的右分支路径数为 5×3=15。P1 的左分支是一条简单路径，结合右分支计算结果可知，P1 包含的嵌套选择结构的路径数为 15+1=16。因此，结合 P8 路径数计算结果可知，该程序的可执行路径总数为 16×3=48。也就是说，最少需要 48 个测试用例才能保证完成该程序的路径覆盖测试。

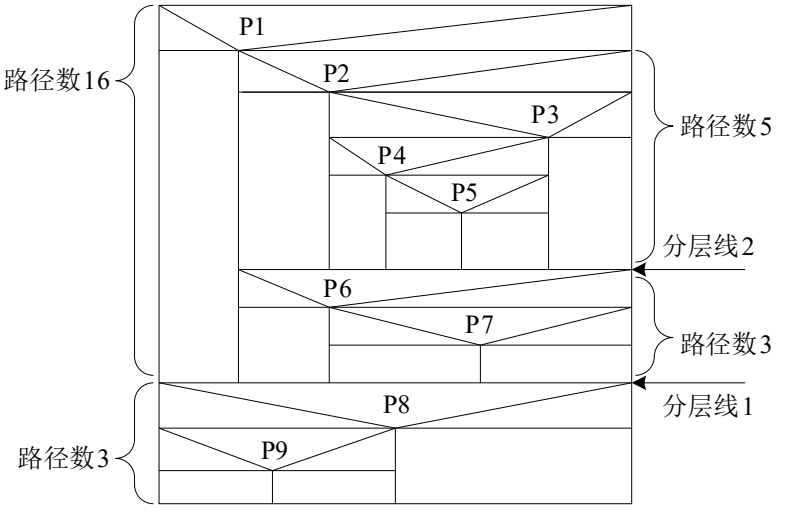


图 2-9　包含复杂结构的程序盒图

2.5 循环结构测试

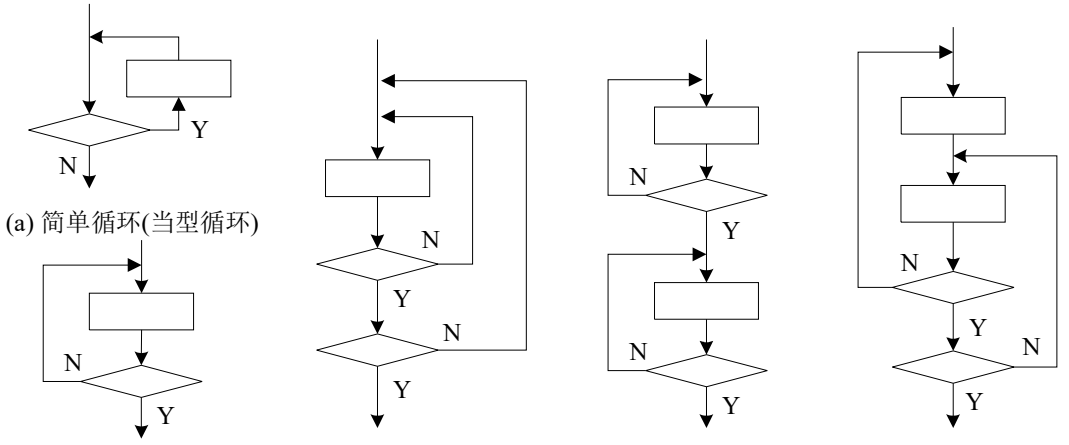
前面讲解了几种主要的逻辑覆盖测试方法，这些方法主要是针对程序选择结构的测试方法。当碰到循环结构时，都进行了大幅度简化，将循环结构转换为选择结构进行测试。但是，当程序中包含比较复杂的循环结构或者循环结构中的程序计算很容易出错时，就需要对其进行更为全面和深入的测试。

循环结构一般有如图 2-10 所示的 4 种形式：简单循环、嵌套循环、串接循环和不规则循环。其中，不规则循环无法进行测试，需要对循环结构进行重新设计，使之成为结构化的程序后再进行测试。下面对简单循环、嵌套循环和串接循环的测试方法进行说明。

1) 简单循环

对简单循环进行测试时，需要考虑循环的次数以及循环边界值和接近边界值的情况。假定循环的最大次数为 n，一般需要设计如下几种测试用例。

- 零次循环：从循环入口直接跳到循环出口。
- 一次循环：只有一次通过循环，用于查找可能的循环初始值错误。
- 两次循环：两次通过循环。



(a) 简单循环(当型循环)

(b) 简单循环(直到型循环)　(c) 嵌套循环　(d) 串接循环　(e) 不规则循环

图 2-10　4 种典型的循环结构

- m 次循环：m 次通过循环，其中 m<n，也就是在 n 次循环中找一个中间值，用于查找在多次循环时才可能暴露的错误。
- n－1 次循环：比最大循环次数少一次通过循环。
- n 次循环：用最大循环次数执行循环。
- n+1 次循环：比最大循环次数多一次通过循环。

测试中，我们还需要关注以下几个问题：

- 循环变量的初值是否正确。
- 循环变量的最大值是否正确。
- 循环变量的增量是否正确。
- 何时退出循环。

下面是一个简单循环，其测试用例如表 2-10 所示。

```
int Sample_Loop( )
{
   int i=1;
   int Sum=0;
   while (i<=10)
{
   Sum=Sum+i;
   i=i+1;
}
   return Sum;
}
```

表 2-10　上述简单循环的测试用例

测试内容	测试用例	备注
整个跳过循环	i=11	0 次通过循环
只有一次通过循环	i=10	

(续表)

测试内容	测试用例	备注
两次通过循环	i=9	
m 次通过循环，其中 m<10	i=5	6 次通过循环
n－1 次通过循环	i=2	9 次通过循环
n 次通过循环	i=1	10 次通过循环
n+1 次通过循环	i=0	11 次通过循环

2) 嵌套循环

如果将简单循环的测试方法用于测试嵌套循环，随着嵌套层数的增加，测试用例数就会呈指数级增长。针对这个问题，一般采用如下嵌套循环测试方法：

- 从最内层循环开始，将所有其他层的循环设置为最小值。
- 对最内层循环使用简单循环测试。测试时，保持所有外层循环的循环变量为最小值。另外，对越界值和非法值增加其他测试。
- 由内向外逐层外推，对其外面的一层循环进行测试。测试时，其他的外层循环变量取最小值，所有其他嵌套内循环的循环变量取“典型”值。
- 反复进行，直到所有各层循环测试完毕。
- 对全部各层循环，同时取最小循环次数和最大循环次数进行测试。

3) 串接循环

串接循环也称为并列循环，其测试分为两种情况。如果串接循环是相互独立的，则可以简化为两个单独循环来分别处理。但是，如果两个循环串接起来，第一个循环的循环计数是第二个循环的初始值，那么这两个循环不是相互独立的。这种情况下，需要使用嵌套循环的测试方法进行处理。

2.6 基本路径测试

在白盒测试中，基本路径测试是应用非常广泛的一种测试方法。基本路径测试是在程序控制流图的基础上，通过分析控制结构的环路复杂性，导出基本可执行路径的集合，从而设计测试用例的方法。设计出的测试用例需要保证被测程序的每一条可执行语句至少被执行一次。

基本路径测试包含如下 4 个基本步骤：

(1) 以详细设计或源代码为基础，绘制程序控制流图；

(2) 根据程序控制流图，计算程序环路复杂度；

(3) 确定独立路径的集合；

(4) 生成测试用例。

下面我们对以上 4 个步骤的内容分别予以介绍。

2.6.1 程序控制流图与环路复杂度

程序控制流图简称流图，本质上是一种“退化”的程序流程图，用于突出表示程序的控制

结构。流图只呈现程序的控制流程，完全不表现具体的语句以及选择或循环的具体条件。图 2-11 给出了几种典型的程序控制结构的控制流图形式。

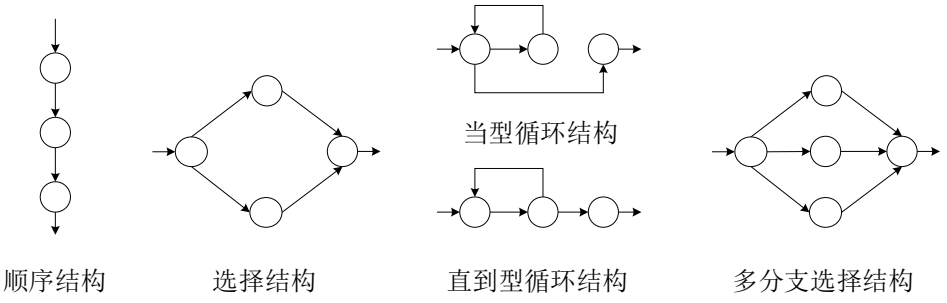


图 2-11　程序控制流图的基本形式

控制流图是一种有向图，由节点和边构成，含义分别如下。

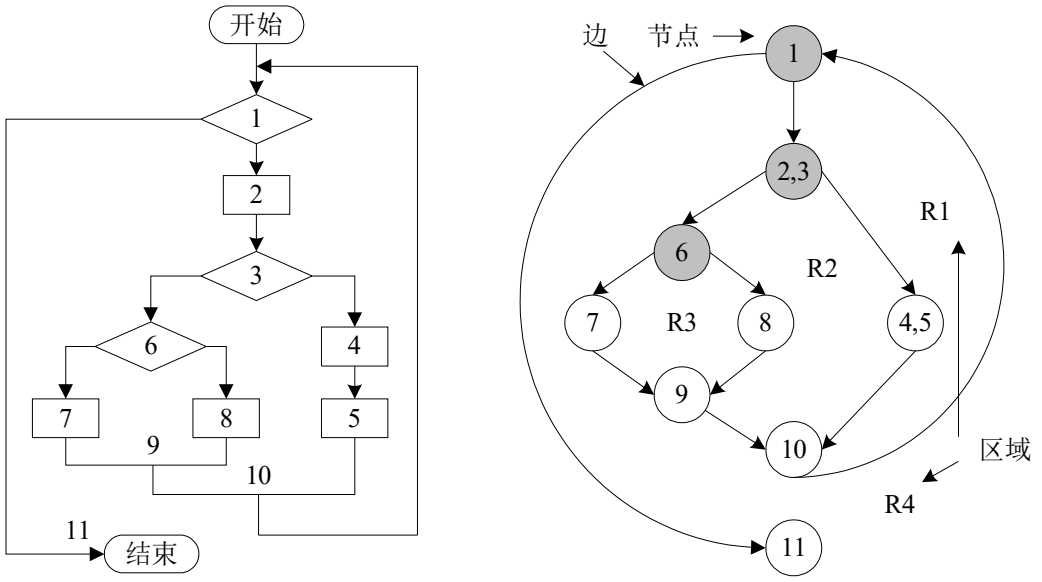
(1) 节点：用圆表示。一个节点代表一条或多条顺序执行的语句。程序流程图中顺序的处理框序列和菱形判定框，可以映射成流图中的节点。

(2) 边：用箭头线表示。边代表控制流，一条边必须终止于一个节点，即使这个节点并不代表任何语句。

当我们将常见的程序流程图转换为控制流图时，需要注意以下两点：

- 在选择或多分支结构中，分支的汇聚处应当添加一个汇聚节点，即使在该处并没有实际的可执行语句也应如此，这样可以使控制结构表现得更为完整和清晰。
- 由边和节点围成的面积称为区域。当计算区域总数时，图形外未围起来的那部分也要记为区域。

图 2-12(a)所示的程序流程图，可以转换为图 2-12(b)所示的控制流图。



(a) 程序流程图　　(b) 控制流图

图 2-12　将程序流程图转换为控制流图

根据程序控制流图，可以定量度量程序的复杂程度，度量结果称为程序的环路复杂度、环形复杂度或圈复杂度。流图一般标记为G，环路复杂度标记为V(G)。一般来讲，模块的环路复杂度 V(G)≤10。

计算环路复杂度有以下 3 种方法：

(1) 控制流图中的区域数等于环路复杂度。

(2) V(G)=E－N+2，其中，E 是控制流图中边的数量，N 是节点的数量。

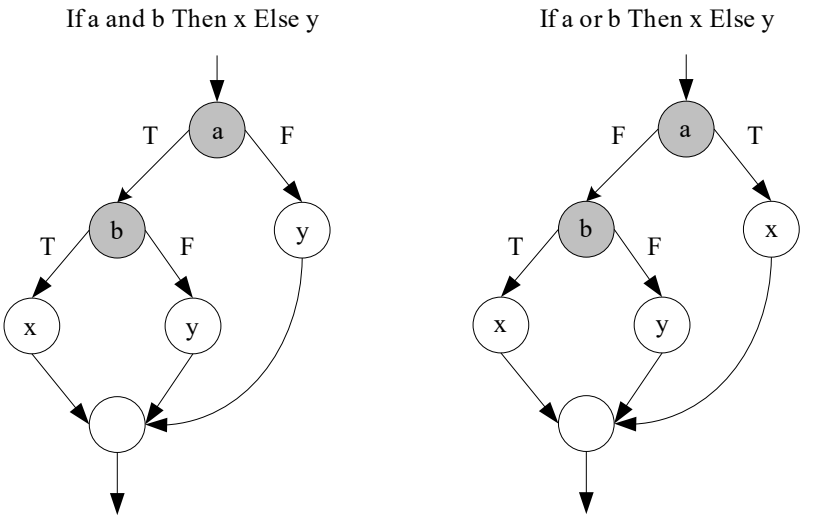
(3) V(G)=P+1，其中，P 是控制流图中判定节点的数量。

例如，通过上述方法计算图 2-12(b)所示程序的环路复杂度：

- 有 R1~R4 共 4 个区域，环路复杂度为 4。
- E=11，N=9，V(G)=11－9+2=4。
- 控制流图中“出度”大于 1 的节点为判定节点，也就是说，起始于判定节点，以之作为箭头线尾端的边的数量一定大于 1。因此，节点 1、(2,3)和 6 是判定节点，P=3，V(G)=3+1=4。

使用以上 3 种计算方法得到的环路复杂度一定是相同的，它们之间可以相互验证。

在图 2-12(a)中，我们实际上假设所有菱形框表示的判定内没有复合条件。但是，需要特别注意的是，如果判定包含复合条件，那么在生成控制流图时，应当把复合条件分解为若干简单条件，每个简单条件对应流图中的一个节点。图 2-13(a)和(b)分别展示了“与”逻辑和“或”逻辑下控制流图的生成方法。



(a)“与”逻辑下的控制流图　　(b)“或”逻辑下的控制流图

图 2-13　复合条件下的控制流图

2.6.2　独立路径集合

独立路径也称为基本路径，其含义包含以下两点：

(1) 独立路径是一条从起始节点到终止节点的路径。

(2) 一条独立路径至少包含一条其他独立路径没有包含的边，也就是说，至少引入了一条

新的执行语句。

图 2-14 所示程序控制流图的环路复杂度为：

- V(G)=图中区域数=5。
- V(G)= E－N+2=10－7+2=5。
- V(G)=P+1=4+1=5。

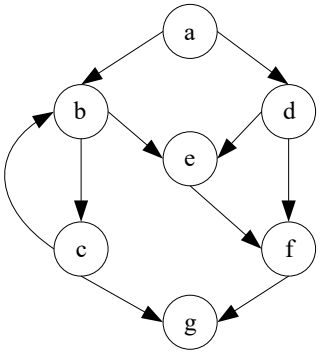


图 2-14　独立路径集合示例

程序的环路复杂度计算结果给出了程序独立路径集合中的独立路径条数，这是保证程序中每条可执行语句至少被执行一次所必需的测试用例数量的上限，也就是说，我们只要最多 V(G)个测试用例就可以满足基本路径覆盖要求。

针对图 2-14，我们可以找出如下 5 条独立路径，构成独立路径集合：

- Path1：a-b-c-g。
- Path2：a-b-c-b-c-g。
- Path3：a-b-e-f-g。
- Path4：a-d-e-f-g。
- Path5：a-d-f-g。

如果能再找出一条路径 Path6=a-b-c-b-e-f-g，我们就会发现，在原有 5 条独立路径的基础上，Path6 并没有引入任何新的边。所以，Path6 不再是一条新的独立路径。同时，我们也会认识到，一个程序的独立路径集合通常并不是唯一的，例如，将 Path2 替换为 Path6 也可以构成一个新的独立路径集合。另外需要注意的是，独立路径集合中的每一条路径都以起始节点“a”开始，以终止节点“g”结束。

得到独立路径集合后，就可以根据每一条独立路径设计相应的输入数据，形成测试用例，保证每一条独立路径都可以被测试到。

2.6.3　基本路径测试用例

首先，让我们来看一个判定中不含复合条件的被测程序。

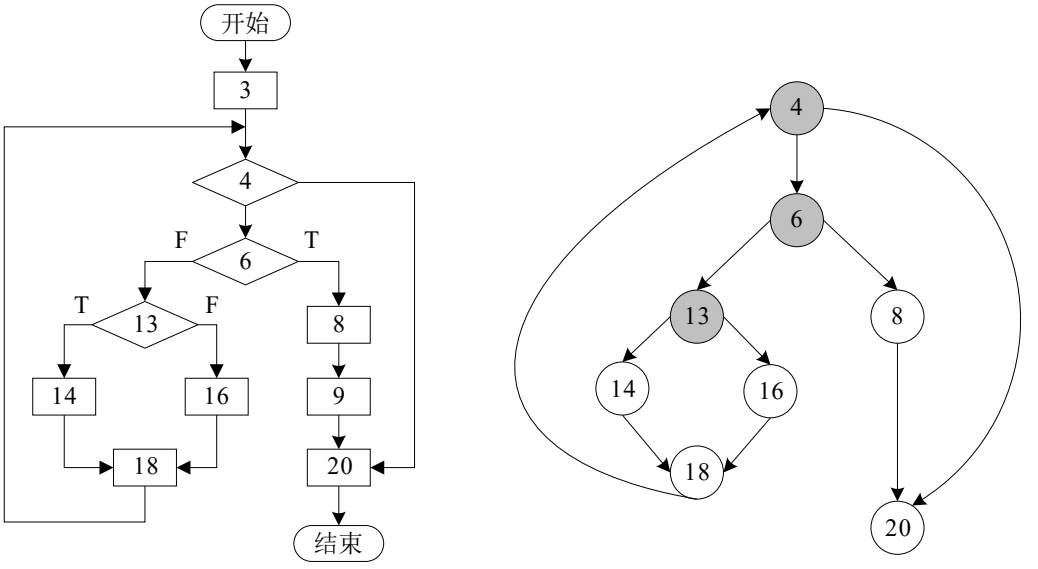
```
1 int Test(int count, int flag)
2 {
3     int temp=0;
4     while (count>0)
5     {
```

```
6       if (flag==0)
7       {
8           temp=count+100;
9           break;
10        }
11        else
12        {
13            if (flag==1)
14                temp=temp+10;
15            else
16                temp=temp+20;
17        }
18        count=count-1;
19    }
20    return temp;
21  }
```

在上述程序中，当 flag=0 时，返回 count+100；当 flag=1 时，返回 count*10；当 flag 是其他值时，返回 count*20。下面按照基本路径测试的 4 个步骤进行说明。

1) 画出上述程序的控制流图

在初期还不能熟练绘制程序控制流图时，可以先绘制出如图 2-15(a)所示的程序流程图，再将其转换为如图 2-15(b)所示的程序控制流图。熟练后，再直接绘制程序控制流图。



(a) 程序流程图　　(b) 控制流图

图 2-15　程序流程图和控制流图

图 2-15 中的数字是源程序中的行号，4、6、13 是判定节点。语句 3 和 4 顺序执行，合并为节点 4；语句 8 和 9 顺序执行，合并为节点 8。

2) 计算程序的环路复杂度

由图 2-15(b)所示的程序控制流图可以计算得出：

- V(G)=图中区域数=4。
- V(G)= E－N+2=10－8+2=4。
- V(G)=P+1=3+1=4。

程序的环路复杂度是 4。因此，只要最多 4 个测试用例就可以达到基本路径覆盖。

3) 确定独立路径集合

在程序控制流图中，从起始节点 4 到终止节点 20 共有 4 条独立路径：

- 4-20
- 4-6-8-20
- 4-6-13-14-18-4-20
- 4-6-13-16-18-4-20

由上面 4 条独立路径构成的集合已经包括流图中所有的边。

4) 设计测试用例

根据上面得到的 4 条独立路径可以设计如表 2-11 所示的 4 个测试用例。

表 2-11　基本路径测试用例

输入数据	预期结果	独立路径
flag=0，或是 flag<0 的某个值	temp=0	4-20
count=1，flag=0	temp=101	4-6-8-20
count=1，flag=1	temp=10	4-6-13-14-18-4-20
count=1，flag=2	temp=20	4-6-13-16-18-4-20

细心的读者会发现，独立路径 3 和 4 实际上已经完全包容独立路径 1。所以，上述测试用例可以简化为 3 项，独立路径 1 的测试用例可以去除。这种情况说明，程序的环路复杂度表示的是最大测试用例个数，是测试用例数量的上界，实际的测试用例数不一定要达到这个上界。不过还要说明的是，测试用例数量越简化，测试的充分性就越低。例如，去掉独立路径 1 的测试用例后，直接跳过循环的情况就无法测试到。因此，需要根据实际情况来确定测试用例数量的简化程度。

接下来，让我们再看一个包含复合条件的被测程序的基本路径测试用例的设计过程。

图 2-16 是计算学生平均成绩的程序流程图。该程序最多可以计算 50 个学生的平均成绩，以－1 作为成绩输入结束标志。程序流程图中，i 是学生序号，n1 是有效成绩数量，n2 是输入的成绩数量，sum 是成绩累加值，Score(i)是第 i 个学生的成绩，Average 是平均成绩。

(1) 画出程序的控制流图。根据图 2-16 所示的程序流程图可以绘制出如图 2-17 所示的程序流图，这一步骤是难点，关键在于将程序流程图中包含复合条件的两个判定分解映射为控制流图中相应的节点。

(2) 计算环路复杂度，如下所示：

- V(G)=图中区域数=6。
- V(G)= E－N+2=16－12+2=6。
- V(G)=P+1=5+1=6。

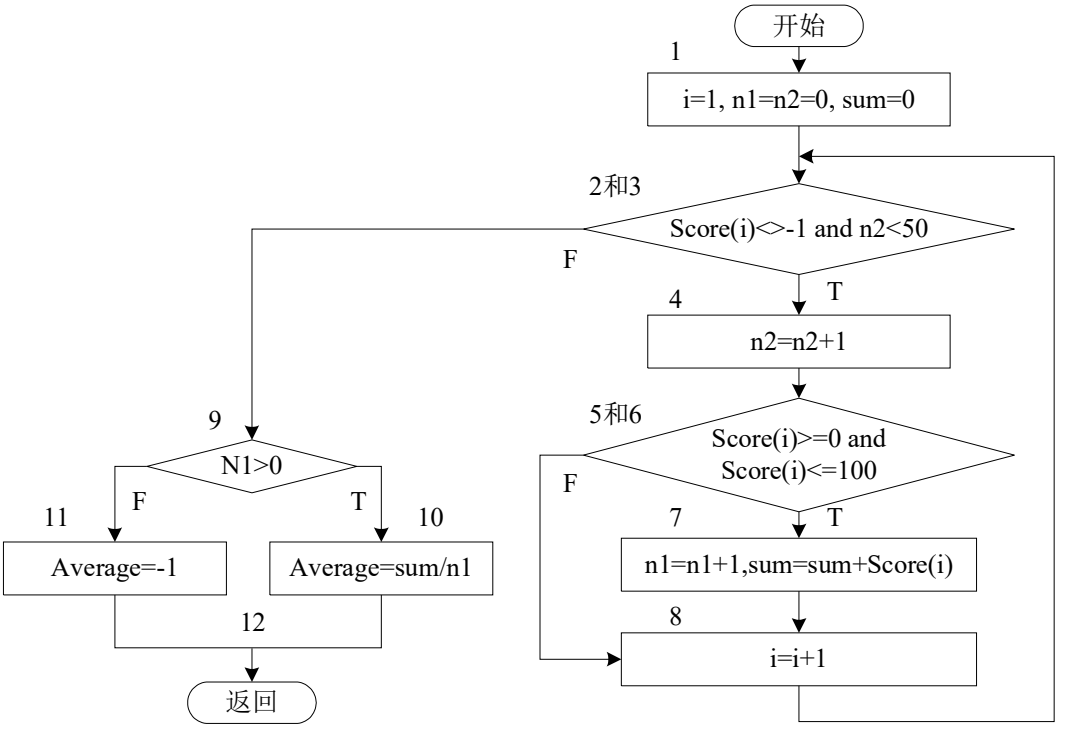


图 2-16 包含复合条件的程序流程图

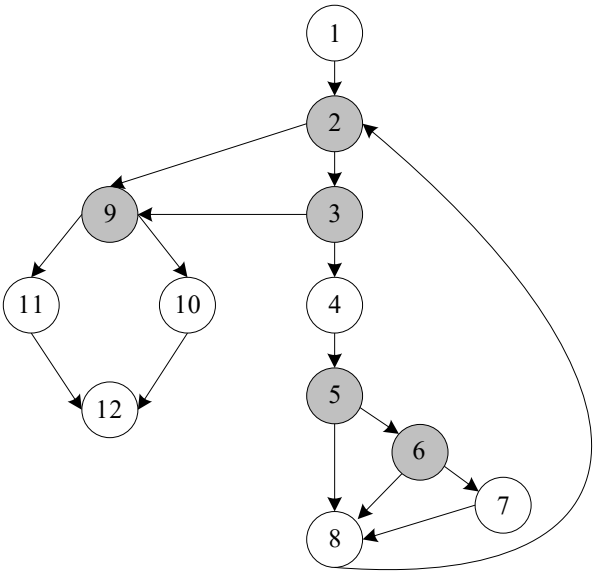


图 2-17 程序的控制流图

(3) 确定独立路径集合。可以确定以下 6 条独立路径：

- 1-2-9-10-12
- 1-2-9-11-12
- 1-2-3-9-10-12

- 1-2-3-4-5-8-2. . .
- 1-2-3-4-5-6-8-2. . .
- 1-2-3-4-5-6-7-8-2. . .

(4) 设计测试用例。为每一条独立路径设计一个测试用例，驱动被测程序沿着该路径至少执行一次。可以设计如表 2-12 所示的 6 个测试用例。

表 2-12　基本路径测试用例

输入数据	预期结果	独立路径
Score(1)=60，Score(2)=-1	n1=1，sum=60，Average=60	1-2-9-10-12
Score(1)=-1	Average=-1，其他变量为初值	1-2-9-11-12
输入多于 50 个有效分数	n1=50，正确的 sum 和 Average 值	1-2-3-9-10-12
Score(1)=-5，Score(2)=70，Score(3)=-1	n1=1，sum=70，Average=70	1-2-3-4-5-8-2. . .
Score(1)=110，Score(2)=80，Score(3)=-1	n1=1，sum=80，Average=80	1-2-3-4-5-6-8-2. . .
Score(1)=80，Score(2)=90，Score(3)=-1	n1=2，sum=170，Average=85	1-2-3-4-5-6-7-8-2. . .

2.6.4　控制流图矩阵

控制流图矩阵是将程序控制流图表达为矩阵的方式。利用控制流图矩阵，可以构造辅助完成路径测试的工具，自动地确定独立路径集合，评估程序的控制结构。

控制流图矩阵实际上是有向图的图形矩阵，由行列数相同的方阵构成，行列数即为控制流图中的节点数，每行和每列依次对应一个节点，矩阵元素反映的是节点间的连接关系。如果节点 i 到节点 j 之间有一条边，那么矩阵第 i 行第 j 列的元素非空。矩阵元素标记为权值 1，表示存在连接；标记为空或权值 0，则表示不存在连接。

例如，可以将图 2-15(b)所示的控制流图表达为如图 2-18 所示的控制流图矩阵。从图 2-18 中可以看出，凡是一行中有大于或等于两个元素的节点就一定是判定节点。通过这一特点，可以方便地确定判定节点的数量，然后计算环路复杂度。

	4	6	8	13	14	16	18	20
4		1						1
6			1	1				
8								1
13					1	1		
14							1	
16							1	
18	1							
20								

图 2-18　控制流图矩阵

对于控制流图矩阵中的元素，除了权值之外，还可以赋予其他属性信息，用于完成对控制结构的一些评估工作。

- 执行连接(边)的概率。

- 执行连接的频率
- 连接的处理时间。
- 执行连接所需的计算资源(如内存等)。

除了利用控制流图矩阵自行开发基本路径测试的辅助工具外，还有一些现成的工具可以利用，例如加拿大麦吉尔(McGill)大学 Sable 研究小组开发的 Soot。通过 Soot 可以进行 Java 程序过程内和过程间的分析优化，以图形化的方式输出程序控制流图，为测试用例的设计提供便捷条件。

2.6.5 基本路径测试的扩展应用

基本路径测试虽然被广泛应用于单元测试阶段的程序路径测试，但是也可以将这种测试方法推广应用到对软件系统流程的测试。

用基本路径测试对程序进行测试时，路径是指函数代码的某个分支。实际上，将软件系统的某个流程也看作路径的话，就可以用基本路径分析的方法来设计测试用例。此时，控制流图中节点的粒度由语句级扩大到模块级，而边反映了软件的系统流程。

采用基本路径测试方法测试系统流程有如下优点：

- 在已知系统流程结构的基础上，可以设计出高质量的测试用例，降低了设计难度。
- 在测试时间紧张的情况下，可以完成对系统重点流程的测试，无须完全根据经验来取舍测试内容。

应用基本路径测试对系统流程进行测试时，一般分为如下 3 个步骤。

1) 将系统运行的流程以控制流图的方式表达出来

将系统流程表达为不同功能或模块的执行关系序列，从最常使用的基本流程入手，再考虑次要和异常的流程。通过逐步理解和细化流程，将各个看似孤立的流程关联起来，形成完整的系统控制流图。

2) 找出所有的系统流程独立路径并为每条路径设定优先级

路径优先级的设定需要考虑以下两个因素：

- 路径使用的频率。使用频率越高，路径优先级越高。
- 路径的重要程度。路径执行失败对系统的影响越大，路径优先级越高。

将上述两个因素确定的路径优先级相加就得到了路径的最终优先级。根据路径优先级的排序就可以确定对所有独立路径的测试顺序以及测试的细致程度。

3) 设计测试用例

为每条独立路径选取测试数据，形成测试用例。每条路径可以对应多个测试用例，相应的测试输入数据应当充分考虑典型值、边界值和特殊值等情况。

2.7 其他白盒测试方法

除了上面介绍的主要白盒测试方法外，还有一些在理论和应用方面具有一定价值的其他白盒测试方法。在这里，我们对这些方法做下简单介绍。

1) 域测试

程序错误可以分为域错误、计算型错误和丢失路径错误三种。

- 域错误。这种错误也称为路径错误。程序的每条执行路径都对应于输入域的一类情况，是程序的子计算。如果程序的控制流有错误，那么对于某一特定的输入，程序可能执行的是一条错误路径。
- 计算型错误。属于常见类型的错误，主要由于赋值语句中的计算错误而导致程序输出结果不正确。
- 丢失路径错误。由于程序中的某处少了判定谓词而造成路径丢失。

域测试主要是针对域错误进行的程序测试，是一种基于程序结构的测试方法。“域”在这里是指程序的输入空间，域测试方法基于对程序输入空间的分析，以及在分析基础上对输入空间进行分割划分，然后选取相应的测试点进行测试。

任何程序都有输入空间，而输入空间又可分为不同的子空间，每一子空间都对应一种不同的程序计算。分析被测程序的结构就会发现，子空间的划分是由程序中分支语句中的谓词决定的。位于输入空间的元素，经过程序中某些特定语句的执行而结束，都是为了满足这些特定语句能够执行所要求的条件。

域测试是一种模块测试的有效方法，但是有两个致命的弱点：一是为了简化分析的目的，域测试对被测程序提出了过多的限制，如要求被测程序不出现数组，分支谓词是不含布尔运算的简单谓词等；二是当程序包含很多路径时，所需的测试点非常多。另外，输入域的分割和划分还涉及多维空间的概念，不易理解。这些都限制了域测试方法的实用性，不易被人们所接受。

2) 符号测试

符号测试的基本思想是允许测试用例的输入数据是符号值，用以代替具体的数值数据。目的是解决测试点不易选取，所选测试点不能保证具有完全代表性的问题。

符号值既可以是基本符号变量值，也可以是符号变量值的表达式。测试过程中，程序执行符号计算而不再执行普通的数值计算，计算结果是符号公式或符号谓词。换言之，符号测试执行的是代数运算，而普通测试执行的是算数运算。符号测试可以看成对普通测试的自然扩充，进行一次符号测试等价于选取具体数值进行的大量普通测试，计算结果可以用于直观地判断程序的正确性。

符号测试可以看成程序测试和程序验证的折中。一方面，符号测试沿用传统的程序测试方法，通过运行被测程序来验证其可靠性。另一方面，因为一次符号测试的结果代表了一大类采用具体数值的普通测试的运行结果，所以实际上也就证明了程序是否能够正确处理此类输入。如果程序中只有有限数量的执行路径，并且通过符号测试验证所有路径都能够正确执行，那么一般就能够确认程序的正确性了。

符号测试方法是否能够得到广泛应用的关键在于能否开发出功能更为强大的程序编译器和解释器，使它们能够处理符号运算。目前，符号测试还存在着分支问题、二义性问题、大程序问题等，使其实际应用受到一定的限制。

3) 程序变异

程序变异是一种错误驱动测试，是针对某类特定程序错误进行的测试。测试理论与实践证明，想要找出程序中的所有错误几乎是不可能的。现实的做法是尽可能缩小错误搜索范围，有针对性地去发现特定错误。这样做的优点是，便于重点发现危害较大的潜在软件缺陷，提高测

试效率，降低测试成本。

程序变异又分为程序强变异和程序弱变异。程序强变异通过对程序进行微小的改变而生成许多程序变异体，而程序弱变异并不实际产生程序变异体，而是分析源程序中易于出错的环节，找出有效的测试数据去执行这些部分。程序变异可以模拟典型的程序错误(例如错误的操作符或变量名)，帮助测试人员发现有效的测试数据，定位测试数据的弱点，或是定位很少或从不使用的代码的弱点。

从另一方面来讲，程序变异也是一种基于错误植入的软件测试技术。可以利用这种技术来衡量测试用例集发现错误的有效性。变异测试通过在程序中逐个引入符合语法的变化，把源程序变异为若干变异程序，利用相应的测试结果检验测试用例集的错误检测能力，预测源程序存在错误的可能性。根据变异对象的类型，变异算子可以分为语句、运算符、常量等类型。根据变异的行为，变异算子又可以分为替换、插入、删除三种类型。

程序变异具有针对性强、系统测试性强的优点。同时，也存在着如果运行所有变异因子会成倍提高测试成本的缺点。因此，在实际工作中，测试人员往往需要借助一些变异测试工具来完成工作。利用变异测试工具，测试人员可以不再考虑复杂的程序变异概念，由工具自动完成对于变异情况的统计、分析，以及用变异结果生成测试数据等工作。

2.8 白盒测试应用策略

白盒测试分为静态测试和动态测试，而静态测试和动态测试又包含多种不同的白盒测试方法。每一种白盒测试方法都具有各自不同的特点，如何根据具体的被测软件，选择合适的方法完成白盒测试是一个应当重视的问题。正确的白盒测试应用策略可以帮助测试人员发现更多的软件缺陷，有效地提高测试效率和测试覆盖率。

下面是一些白盒测试方法综合应用策略，可以在实际测试过程中予以参考。

(1) 开始进行白盒测试时，首先应尽量使用测试工具进行程序静态结构分析。

(2) 在测试中，建议采用先静态后动态的组合方式。先进行静态结构分析和代码检查，再进行覆盖测试。

(3) 利用静态分析的结果作为依据和引导，再使用代码检查和动态测试的方式对静态测试分析结果进行进一步确认，使测试工作更为准确和有效。

(4) 覆盖率测试是白盒测试的重点，是测试报告中可以作为量化指标的依据。一般可以使用基本路径测试达到语句覆盖的标准，对于重点模块应当使用多种覆盖率标准衡量代码的覆盖率。

(5) 在不同的测试阶段，白盒测试的应用侧重点也不同。单元测试以代码检查和逻辑覆盖为主，集成测试需要增加静态结构分析等，而系统测试需要根据黑盒测试结果采取相应的白盒测试。

2.9 思考题

1. 什么是白盒测试？白盒测试为什么又称为结构测试或逻辑驱动测试？

2. 白盒测试都有哪些基本测试原则？

3. 代码检查法主要包括哪些方法？它们各自的特点是什么？

4. 一般在程序的哪些位置进行程序插桩？

5. 逻辑覆盖测试主要有哪几种？它们的覆盖标准是什么？相互之间的强弱关系是怎样的？

6. 请分析一下判定覆盖和条件覆盖之间的关系。

7. 简单循环测试一般需要设计哪些测试用例？

8. 请尝试编写一个利用控制流图矩阵自动获得独立路径集合的小程序，然后利用本章介绍的基本路径测试示例，验证程序的功能是否正确实现。

9. 画出下面程序的流程图，然后设计语句覆盖、判定覆盖、条件覆盖、判定-条件覆盖、条件组合覆盖和路径覆盖测试用例。

```
int LogicExample(int x, int y)
{
    int magic=0;
    if(x>0 && y>0)
    {
        magic = x+y+10;
    }
    else
    {
        magic = x+y-10;
    }
    if (magic < 0)
    {
        magic = 0;
    }
    return magic;
}
```

10. 根据图 2-4 中的被测程序和程序流程图，用基本路径测试设计测试用例。设计过程要求包含绘制控制流图、计算环路复杂度、确定独立路径集合、生成测试用例 4 个基本步骤。