第3章

白盒测试

3.1 白盒测试概述

3.1.1 白盒测试简介

白盒测试是对一类软件测试方法的统称，这类测试方法针对的测试对象是程序代码，它要求已知程序的逻辑结构、工作过程，检查程序所有内部成分是否符合相关标准和要求，验证程序的每种内部操作是否符合设计规格。

白盒测试把被测软件看成是透明的盒子，内部是可视的，测试人员需要清楚盒子内部的结构以及程序流程是如何执行的。白盒测试对程序及其执行过程做细致的检查，对程序的执行过程进行测试覆盖，检查程序中的每条通路是否符合预定要求，能正确工作，并可以通过在程序中的不同位置设立检查点，来检查程序执行的内部状态，以确定实际运行状态与预期状态是否一致。

通过白盒测试，要尽可能检查发现程序代码中的问题和缺陷，让代码达到正确性、高效性、清晰性、规范性、一致性等要求。

3.1.2 静态白盒测试和动态白盒测试

白盒测试既有静态方法也有动态方法，如图 3-1 所示。

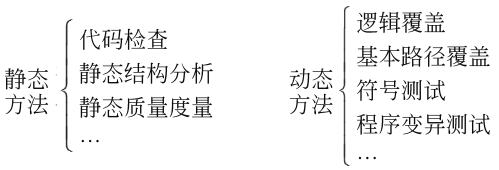


图 3-1 白盒测试方法

静态白盒测试是指在不执行程序的情况下，对程序进行检查和分析。

动态白盒测试是指先针对程序的内部逻辑结构设计测试用例，然后运行程序，输入测试用例，检验程序执行过程及最终结果是否符合预期要求，查找问题和缺陷。逻辑覆盖、基本路径覆盖、程序变异测试等都是动态白盒测试方法。

3.2 静态白盒测试

静态白盒测试是指在不执行被测试程序的情况下，对程序代码进行检查和分析，并争取发现问题，找出缺陷的过程，如图 3-2 所示。

对程序最基本的检查是找出源代码的语法错误，这类检查可由编译器来完成，编译器可以逐行分析程序的语法，找出错误并报告。除此之外，有许多非语法方面的错误，编译器无法发现，开发或者测试人员还需采用人工或自动化的方法来检查、分析源代码，争取能够检测发现，如图 3-3 所示。

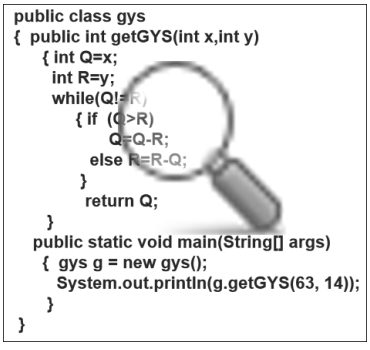


图 3-2 静态白盒测试示意图

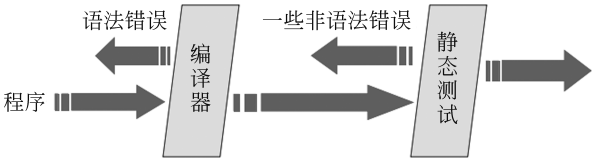


图 3-3 静态白盒测试针对的是非语法错误

静态白盒测试通过检查或分析程序代码的逻辑、结构、过程、接口、编码规范等来发现问题，找出缺陷和可疑之处，如不匹配的参数、不适当的循环嵌套和分支嵌套、不允许的递归、未使用过的变量、空指针的引用和可疑的计算等。静态白盒测试的结果可用于进一步的查错，并为测试用例选取提供指导。最常见的静态测试包括代码检查、静态结构分析、静态质量度量等。

3.2.1 代码检查

代码检查就是直接对源程序代码进行各项检查，看是否符合相关规范要求。以前的代码检查又可分为桌面检查、代码审查和走查方式等不同形式，随着技术的发展，现在的代码检查一般都是由测试工具软件自动完成。代码检查的主要内容如下。

（1）代码和设计的一致性。

（2）代码逻辑表达的正确性。

（3）代码的可读性。

（4）代码风格、格式的规范性和一致性。

（5）代码对编码规则、编码规范等的遵循情况。

（6）代码结构的合理性。

（7）程序中是否存在不安全、不明确和模糊的部分。

代码检查一旦发现错误，通常能在代码中对其进行精确定位，这与动态测试只能发现

错误的外部征兆不同，因而可以降低修正错误的成本。另外，在代码检查过程中，有时可以发现成批的错误，典型的如分散在多处的同一类错误，而动态测试通常只能一个一个地测试和报错。

1. 静态结构分析

一个软件通常由多个部分组成，总是存在着一定的组织结构，各个部分之间也总是存在一定关联，在静态结构分析中，通常通过使用测试工具，分析程序代码的控制逻辑、数据结构、模块接口、调用关系等，生成控制流图、调用关系图、模块组织结构图、引用表、等价表、常量表等各种图表，清晰地呈现软件的组织结构和内在联系，使得程序便于被宏观把握和微观分析。

借助这些图表，可以进行控制流分析、数据流分析、接口分析、表达式分析等，可以发现程序中的问题或者不合理的地方，然后通过进一步检查，就可以确认软件中是不是存在缺陷或错误。

静态结构分析通常采用以下一些方法进行程序的静态分析。

(1) 通过生成各种表，来帮助对源程序的静态分析。

① 标号交叉引用表。

② 变量交叉引用表。

③ 子程序(宏、函数)引用表。

④ 等价表。

⑤ 常数表。

(2) 通过分析各种关系图、控制流图来检查程序是否有问题。

① 控制流图：由许多节点和连接节点的边组成的图形，其中每个节点代表一条或多条语句，边表示控制流向，可以直观地反映出一个函数的内部结构。

② 函数调用关系图：列出所有函数，用连线表示调用关系，通过应用程序各函数之间的调用关系展示了系统的结构。

③ 文件或者页面调用关系图。

④ 模块结构图。

(3) 其他常见错误分析。分析程序中是否有某类问题、错误或“危险”的结构。

① 数据类型和单位分析。

② 引用分析。

③ 表达式分析。

④ 接口分析。

2. 程序流程分析

一个程序要能够正常执行，不出现问题、不留下隐患，在流程上会有一些基本要求，下面分别从控制流和数据流的角度来对程序做流程上的分析。

(1) 控制流分析

从控制流的角度来说，程序不应存在以下问题。

① 转向并不存在的函数、方法、页面等。

如果转向并不存在的函数、方法、页面等，程序执行就会意外中止。

② 有从程序入口无法到达的语句。

有从程序入口无法到达的语句，就意味着这些语句根本就不会被执行到，其对应的功能也无法被调用。

③ 有不能退出执行的语句、函数、方法、界面等。

例如，某 App 的主界面没有退出按钮，这会导致用户使用的不便。

（2）数据流分析

数据流分析就是对程序中数据的定义、引用及其之间的依赖关系等进行分析的过程。某一语句执行时能改变变量 V 的值，则称 V 是被该语句定义的。某一语句的执行用到内存变量 V 的值，则称变量被语句引用。其示例如下。

语句　X:＝Y＋Z 定义了变量 X，引用了 Y，Z。

语句　if　Y＞Z　then…，引用了变量 Y 和 Z。

语句　READ　X，引用了变量 X。

语句　WRITE　X，定义了变量 X。

一般而言，变量应当先定义再使用，不会用到的变量，就不要定义。

（3）示例

程序的控制流分析和数据流分析，有的编译器就带有相关功能，并能给出提示。例如，在 Eclipse 中编写如图 3-4 中代码。

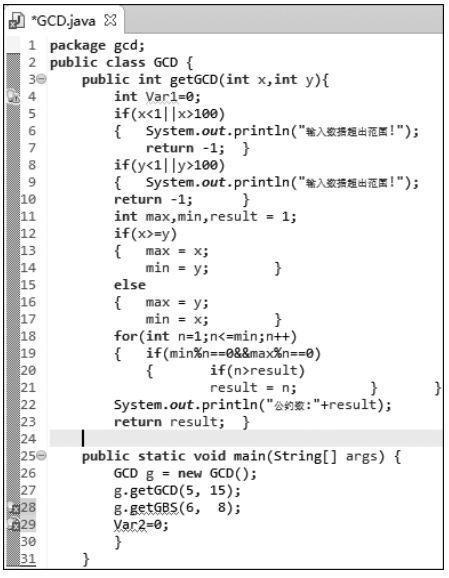


图 3-4　Eclipse 中的控制流、数据流分析提示

鼠标指针放在第 4 行行首的叹号上，开发平台提示：

The value of the local variable Var1 is not used

就是提醒说定义的变量 Var1 没有被使用到。

鼠标指针放在第 28 行行首的×号上，开发平台提示：

The method getGBS(int, int) is undefined for the type GCD

就是提醒说要调用的方法 getGBS()没有被定义。

鼠标指针放在第 29 行行首的×号上，开发平台提示

Var2 cannot be resolved to a variable

就是提醒说变量 Var2 没有被定义。

3.2.2 编码规则和编程规范

在程序编写中，根据代码分析、经验教训等，我们会发现如果遵循某些规则，则程序出现问题的概率会降低，反之，则会导致程序执行出错或者至少是留下了问题隐患，于是可以把这些应当遵守的规则称为编码规则，用这些规则去检查待测试的代码有没有遵守，起到尽可能避开软件编码过程中容易出现的错误和疏漏，提升软件产品质量的效果。

随着软件规模越来越大，很多时候需要很多人参与到同一个项目中，共同完成规模庞大、结构复杂的软件系统。在这种情况下，如何统一程序风格，规范代码的编写，让不同的程序员开发的代码能够合成一个整体，容易阅读和理解，就成为了一个巨大的挑战。于是编程规范应运而生，它要求所有参与编程的人，按照统一的风格、格式、规范编写程序代码，起到统一代码风格，降低协同成本，提高代码的可读性和可理解性的效果。

由于编码规则与编程规范的重要性，部分大型软件开发公司相继提出并开放属于自己的编程标准，例如，Google 公司针对多种语言（包括 Java、C++ 、Object-C 等）提出相应的编码规范，国内如阿里巴巴公司也提出中英文版本的面向 Java 程序的开发手册，同时还提供了 Java 开发规约插件，用以帮助研发人员自动化检测自己编写的代码。

编程规范或者规则分成多种情况，有的是推荐遵循或建议参考的，有的则是要求必须遵守的；有的只是某一个软件开发组织自行制定和使用，有的则是在一定范围内普遍认同并遵循；有的是和某一种编程语言相关的，有的则是与具体的编程语言无关的；有的是由手工来进行检查和确认，有的则可以通过工具软件来进行分析和度量；有的是为了统一代码风格、便于代码阅读和理解，有的则是为了防止编码错误和疏漏、提升软件产品质量；有的针对代码的逻辑结构、异常处理、网络、数据库等，有的针对可测试性、安全性等；有的比较简单，如变量的命名格式，有的较为复杂，如数据库操作等。下面分类列举一些常见的编码规则和编程规范。

1. 常见 Java 编码规则违背示例

常见 Java 编码规则违背示例见表 3-1。

2. 常见编程规范

(1) 注释

① 注释要简单、清楚、明了，含义准确，防止二义性。

表 3-1　常见 Java 编码规则违背示例

分　类	规　则	说　明	代码示例
Input Validation and Representation	Cross-Site Scripting：DOM	向一个 Web 浏览器发送未经验证的数据会导致该浏览器执行恶意代码	insert = $(nNewNode);
	Cross-Site Scripting：Persistent	向一个 Web 浏览器发送未经验证的数据会导致该浏览器执行恶意代码	<buttonclass="btn btn-mini btn-danger" type="button" onclick="dormBuildDelete(${dormBuild.dormBuildId})">删除</button></td>
	Cross-Site Scripting：Reflected	向一个 Web 浏览器发送未经验证的数据会导致该浏览器执行恶意代码	<td><input type="text" id="dormBuildName" name="dormBuildName" value="${dormBuild.dormBuildName }" style="margin-top:5px;height:30px;" /></td>
	SQL Injection	通过不可信赖的数据源输入构建动态 SQL 语句，攻击者就能够修改语句的含义或者执行任意 SQL 命令	PreparedStatement pstmt = con.prepareStatement(sb.toString().replaceFirst("and", "where"));
	Header Manipulation：Cookies	包含未验证的数据，这可产生 Cookie Manipulation 攻击，并导致其他 HTTP 响应头文件操作攻击	Cookie user =new Cookie ("dormuser", userName +"-"+password+"-"+userType+"-"+"yes");
Security Features	Password Management：Password in HTML Form	对 HTML 表单中的密码字段进行填充可能会危及系统安全	<td><input type="password" id="password"name="password" value="${dormManager.password}" style="margin-top:5px;height:30px;" /></td>
	Privacy Violation	对机密信息（如客户密码或社会保障号码）处理不当会危及用户的个人隐私	<td>${dormManager.id}</td>
Environment	Password Management：Password in Configuration File	在配置文件中存储明文密码，可能会危及系统安全	dbUserName=sa dbPassword=123456

② 在必要的地方注释，注释量要适中。

③ 修改代码的同时修改相应的注释，以保证注释与代码的一致性。

④ 注释的就近原则，即保持注释与其对应的代码相邻，并且应放在上方或者与代码同行，不可放在下面。

⑤ 全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。

⑥ 在每个源文件的头部要有必要的注释信息，包括文件名，版本号，作者，生成日期，模块功能描述（如功能、主要算法、内部各部分之间的关系、该文件与其他文件关系等），主要函数或过程清单及本文件历史修改记录等。

⑦ 在每个函数或过程的前面要有必要的注释信息，包括函数或过程名称，功能描述，输入、输出及返回值说明，调用关系及被调用关系说明等。

（2）命名

① 命名应有统一的规则。

② 避免使用不易理解的名称。

③ 较短的单词可通过去掉“元音”形成缩写。

④ 较长的单词可取单词的头几个字符形成缩写。

⑤ 需要包含多个单词的命名可采用下画线来进行分段。

（3）变量

① 去掉没必要的公共变量。

② 构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的公共变量，防止多个不同模块或函数都可以修改、创建同一公共变量的现象。

③ 仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。

④ 明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等。

⑤ 当向公共变量传递数据时，要十分小心，防止赋予不合理的值或越界等现象发生。

⑥ 防止局部变量与公共变量同名。

⑦ 仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起误用现象。

⑧ 结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保留余地（如预留一些空间等）。

⑨ 注意具体的编程语言及编译器处理不同数据类型的原则及有关细节。

⑩ 严禁使用未经初始化的变量。声明变量的同时应对变量进行初始化。

⑪ 编程时，要注意数据类型的强制转换。

（4）函数、过程

① 单个函数的规模尽量限制在200行以内。

② 一个函数最好仅完成一个功能。

③ 为简单但常用功能编写函数。

④ 尽量不要编写依赖于其他函数内部实现的函数。

⑤ 尽量减少函数的参数，降低函数调用时出错的概率。

⑥ 用注释详细说明每个参数的作用、取值范围及参数间的关系。

⑦ 应检查函数所有参数输入的有效性。

⑧ 应检查函数所有非参数输入的有效性，如数据文件、公共变量等。

⑨ 函数名应准确描述函数的功能。

⑩ 函数的返回值要清楚明了，尤其是出错返回值的意义要准确。

⑪ 明确函数功能，代码应能精确（而不是近似）地实现函数功能。

⑫ 减少函数本身或函数间的递归调用。

⑬ 编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即 P、V 操作）等手段对其加以保护。

（5）代码可测性

① 采用漏斗型设计，公共逻辑归一化。

② 降低模块耦合度。

③ 面向接口编程，使用函数接口将外部依赖隔离。

④ 在编写代码之前，应预先设计好程序调试与测试的方法和手段，并设计好各种调测手段及相应测试代码，如测试脚本、输出语句等。

（6）程序效率

① 编程时要经常注意代码的效率。尤其是需要反复执行、并发执行的代码。

② 应在保证软件系统的正确性、稳定性、可读性及可测性的前提下，提高代码效率。而不能一味地追求代码效率，却对软件的正确性、稳定性、可读性及可测性造成影响。

③ 要仔细地构造或直接用汇编语言编写调用频繁或性能要求极高的函数。

④ 通过对系统数据结构划分与组织的改进，以及对程序算法的优化来提高效率。

⑤ 在多重循环中，应将最忙的循环放在最内层。

⑥ 尽量减少循环嵌套层次。

⑦ 尽量用乘法或其他方法代替除法，特别是浮点运算中的除法。

3.2.3 质量度量

软件质量度量是指按照某种质量模型、质量标准指标体系对软件的各个质量特性进行测度，并给出度量结果。随着技术的发展，一般可以借助自动化测试工具来对程序代码进行质量度量，并得到质量度量报告。例如，图 3-5 为使用静态白盒测试工具 Logiscope 对某软件源代码进行测试后得到的质量报告示例。

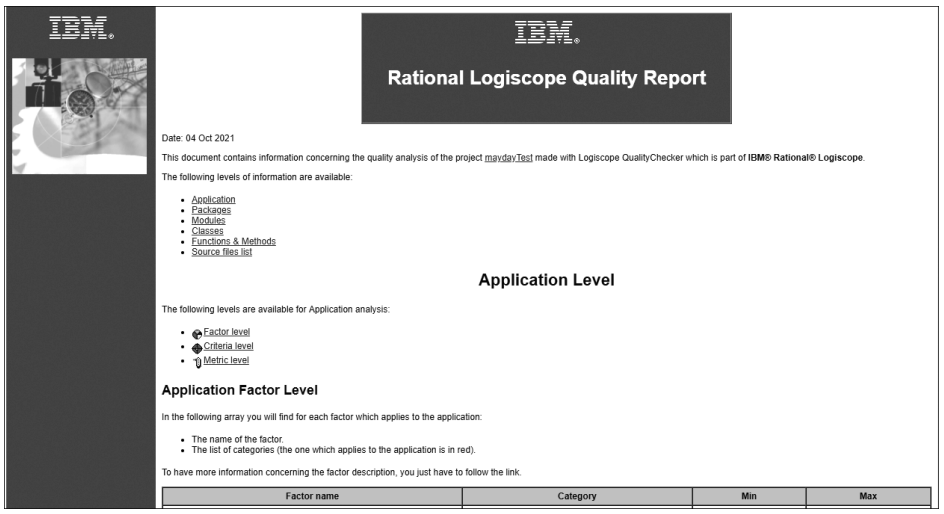


图 3-5　Logiscope 质量报告

3.3 逻辑覆盖

3.3.1 简介

逻辑覆盖是白盒测试中主要的动态测试方法之一，是以程序内部的逻辑结构为基础的测试技术，是通过对程序逻辑结构的遍历来实现对程序的测试覆盖，所谓覆盖就是作为测试标准的逻辑单元、逻辑分支、逻辑取值都被执行到。这一方法要求测试人员对程序的逻辑结构有清楚的了解。逻辑覆盖的标准有语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖等。设有程序段 P1 如下。

```
IF ( x>0 OR   y>0  ) then a =10
IF ( x<10 AND   y<10  ) then b =0
```

其中，变量 a，b 的初始值在其他地方已经定义了，都为－1。程序段 P1 对应的流程图如图 3-6 所示。

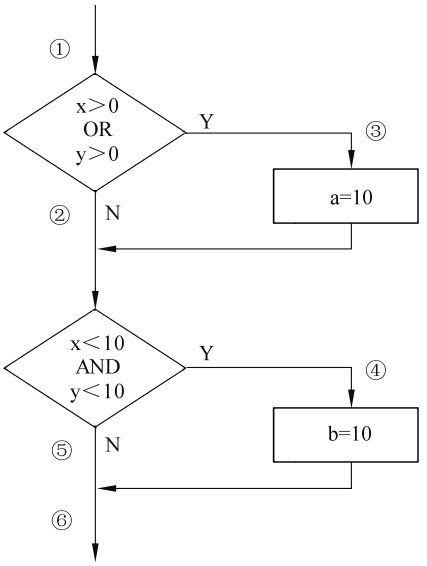


图 3-6 程序段 P1 流程图

下面看一下应如何分别实现语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖和条件组合覆盖。

3.3.2 语句覆盖

语句覆盖要求设计若干测试用例，使得程序中的每个可执行语句至少都能被执行一次。对图 3-6 所示程序段 P1 流程图，按照这一标准，程序需要执行通过的位置有①③④⑥，而②⑤位置，由于没有语句，所以不需要覆盖。

首先可能想到的是，可以设计以下两个测试用例，分别覆盖第一个 IF 结构有执行语句的分支③，和第二个 IF 结构有执行语句的分支④。

```
case1: x=1 ,y=1, 覆盖 ③。
case2: x=-1,y=-1, 覆盖 ④。
```

这样即可达到语句覆盖要求，但从节约测试成本的角度出发，可以优化测试用例设计，实际上只需要一个测试用例，具体如下。

```
case3: x=8,y=8。
```

即可同时覆盖①③④⑥，执行通过路径如图 3-7 所示。

一方面，对于一个具有一定规模的软件而言，要达到 100％的语句覆盖，可能是相当难的，例如，有的代码是用来进行错误处理或是应对某些特殊情况的，如果这种错误或者特殊情况不出现，这些代码就不会被执行到。此时要提高语句覆盖率，需要有针对性地进行测试用例设计。另一方面，语句覆盖实际上是一种比较弱的覆盖准则，从图 3-7 中可以

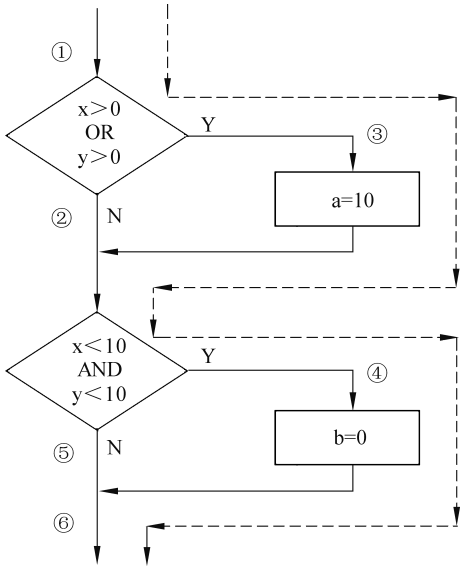


图 3-7　语句覆盖执行路径

看出，两个判断语句的都只执行了一个分支，而另外一个分支根本就没有被执行到。语句覆盖说起来是测试了程序中的每一个可执行语句，似乎能够比较全面地对程序进行检验，但实际上，它并不是一个测试很充分的覆盖标准，有时一些明显的错误语句覆盖测试也发现不了。

如果程序段 P1 中，两个判断语句的逻辑运算符号由于疏忽写错了，第一个判断语句中的 OR 错写成了 and，第二个判断语句中的 AND 错写成了 OR，用测试用例 case3 进行测试，则执行的路径仍然是①③④⑥，如图 3-8 所示，测试结果也依然正确，测试没有能够发现程序中的错误。

语句覆盖的优点是分析和应用起来比较简单，缺点是它对控制结构是不敏感的，对程序执行逻辑的覆盖很低，往往发现不了判断中逻辑运算符可能出现的错误。语句覆盖率的计算公式如下。

$$\text{语句覆盖率} = \frac{\text{被测试到的可执行语句数}}{\text{可执行语句总数}} \times 100\%$$

3.3.3　判定覆盖

比语句覆盖稍强的覆盖标准是判定覆盖。判定覆盖，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真值结果和假值结果都至少出现一次。判定覆盖又称为分支覆盖，因为判断取真值结果就会执行取真分支，判断取假值结果就会执行取假分支，每个判断的真值结果和假值结果都至少出现一次也就相当于每个判断的取真分支和取假分支至少经历一次。

以程序段 P1 为例，对照流程图，按照这一标准，程序需要执行通过的位置有①②③④⑤⑥。程序段 P1 中存在 IF 语句，由于每个判断有真假两种判断结果，所以至少需要两

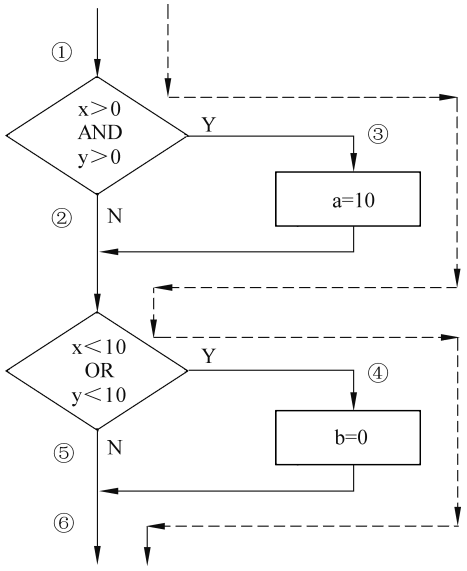


图 3-8 语句覆盖测试未能发现错误

个测试用例。P1 中的两个 IF 语句是串联的，而不是嵌套，所以如果设计合理的话，两个测试用例也确实足够，如下两个测试用例可以达到判定覆盖要求。

case4: x=20,y=20,覆盖 ①③⑤⑥;　　case5: x=-2,y=-2, 覆盖 ①②④⑥。

具体覆盖情况见表 3-2。

表 3-2 判定覆盖表

测试用例编号	x	y	第 1 个判定表达式 x>0 OR y>0	第 2 个判定表达式 x<10 AND y<10
case4	20	20	Y	N
case5	−2	−2	N	Y

如果测试达到判定覆盖，则显然程序流程的所有分支都是会被测试到的，各个分支上的所有语句都会被测试到，所以只要满足判定覆盖，就必定会满足语句覆盖，这一点从图中可以直观地看出来。

在判定覆盖中，如果一个判定表达式中有多个条件，由于只关注这个判断表达式的最终结果，而不是每个条件的判定结果，所以有的条件可能始终只取过真值或者假值，而另外一种取值根本就没有出现过，如果这个条件写错，那么判定覆盖测试显然是发现不了的。也就是说，当程序中的判定表达式是由几个条件组合而成时，判定覆盖对各个条件的测试是不充分的，它未必能发现每个条件中可能存在的错误。判定覆盖率的计算公式如下。

$$\text{判定覆盖率}=\frac{\text{被测试到的判定分支数}}{\text{判定分支总数}}\times 100\%$$

3.3.4 条件覆盖

条件覆盖就是要求判断表达式中的每个条件都要至少取得一次真值和一次假值，需要注意的是，每个条件都要至少取得一次真值和一次假值并不等于每一个判定也都能至少取得一次真值和一次假值，即条件覆盖并不比判定覆盖强，两者只是关注点不同，不存在严格的强弱关系。

例如，对于程序段 P1，设计如下测试用例可以达到条件覆盖要求。

```
case6: x=20,y =-20;    case7: x=-2,y =20。
```

具体覆盖情况见表 3-3。

表 3-3 条件覆盖测试用例表

测试用例编号	x	y	条件 x>0	条件 y>0	条件 x<10	y<10
case6	20	−20	Y	N	N	Y
case7	−2	20	N	Y	Y	N

case6 和 case7 对第 1 个 IF 语句，只覆盖了 Y 分支，对第 2 个 IF 语句，只覆盖了 N 分支，因此并不满足判定覆盖。条件覆盖率的计算公式如下。

$$\text{条件覆盖率} = \frac{\text{被测试到的条件取值数}}{\text{条件取值总数}} \times 100\%$$

3.3.5 条件/判定覆盖

条件覆盖并不比判定覆盖强，两者只是关注点不同，有时会把条件覆盖和判定覆盖结合起来使用，称为条件/判定覆盖，它的含义是指：设计足够多的测试用例，使得判定表达式中每个条件的真/假取值至少都出现一次，并且每个判定表达式自身的真/假取值也都要至少出现一次。

对于程序段 P1，在做判定覆盖时设计的测试用例 case4 和 case5，实际上也同时是满足条件/判定覆盖的，因为每个条件的真/假取值都出现一次，并且每个判定的真/假取值结果也都出现一次，具体覆盖情况见表 3-4。

表 3-4 case4、case5 满足条件/判定覆盖情况

测试用例编号	x	y	条件 x>0	条件 y>0	条件 x<10	y<10
case4	20	20	Y	Y	N	N
case5	−2	−2	N	N	Y	Y

来看一个三角形判定问题的案例，有程序段 P2 如下。

```
if ((a<b+c) && (b<a+c) && (c<a+b))
    is_Triangle =true;
else
```

```
is_Triangle = false;
```

对该程序段进行测试时，如果要满足条件/判定覆盖，则4个条件表达式(见表3-5)都要既有true取值，也有false取值。

表3-5 4个条件表达式

条件表达式编号	条件表达式
1	a<b+c
2	b<a+c
3	c<a+b
4	(a<b+c) && (b<a+c) && (c<a+b)

设计如下测试用例可满足条件/判定覆盖。

case8：a=1,b=1,c=1。

case9：a=1,b=2,c=3。

case10：a=3,b=1,c=2。

case11：a=2,b=3,c=1。

具体覆盖情况见表3-6。

表3-6 满足条件/判定覆盖的测试用例

测试用例编号	a	b	c	条件表达式1	条件表达式2	条件表达式3	条件表达式4
case8	1	1	1	Y	Y	Y	Y
case9	1	2	3	Y	Y	N	N
case10	3	1	2	N	Y	Y	N
case11	2	3	1	Y	N	Y	N

条件/判定覆盖率的计算公式如下。

$$\text{条件/判定覆盖率} = \frac{\text{被测试到的条件取值和判定分支数}}{\text{条件取值总数}+\text{判定分支总数}} \times 100\%$$

3.3.6 条件组合覆盖

条件组合覆盖也叫多条件覆盖，它的含义是要设计足够多的测试用例，使得每个判定中条件取值的各种组合都至少出现一次。显然满足条件组合覆盖的测试用例一定也是满足判定覆盖、条件覆盖和条件/判定组合覆盖的。

对于程序段P1，由于一个判定中有两个条件，而两个条件可能的组合情况有4种，所以，如果要达到条件组合覆盖，至少需要4个测试用例。如果能够合理设计，让4个测试用例在覆盖第1个判定4种条件组合的同时也覆盖第2个判定的4种条件组合，那么4个测试用例就够了，设计如下测试用例可以满足条件组合覆盖。

case12：x= 50,y= 50。

case13：x= −5,y= −5。

case14：x= 50,y= −5。

case15：x= −5,y= 50。

对 2 个判定表达式的条件组合覆盖情况见表 3-7。

表 3-7　条件组合覆盖情况

测试用例编号	x	y	第 1 个判定表达式		第 2 个判定表达式	
			条件 x>0	条件 y>0	条件 x<10	y<10
case12	50	50	Y	Y	N	N
case13	−5	−5	N	N	Y	Y
case14	50	−5	Y	N	N	Y
case15	−5	50	N	Y	Y	N

以上满足条件组合覆盖的 4 个测试用例，虽然能够覆盖判定表达式中条件的各种组合情况，但并不一定能覆盖程序中的每一条可能的执行路径，如路径①②⑤⑥，如图 3-9 所示，就没有被覆盖。

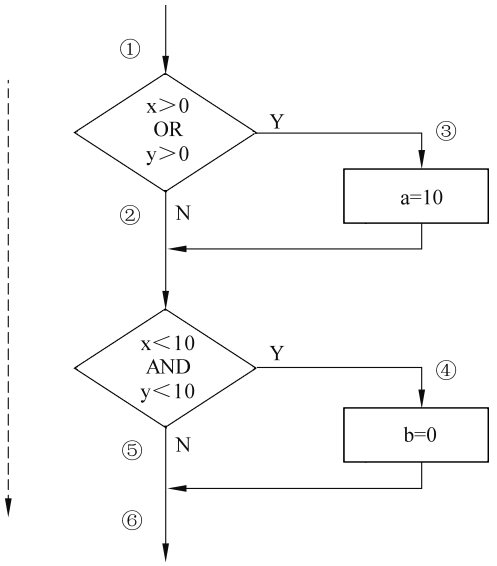


图 3-9　条件组合覆盖未能覆盖的执行路径

条件组合覆盖率的计算公式如下。

$$\text{条件组合覆盖率} = \frac{\text{被测试到的条件取值组合数}}{\text{条件取值组合总数}} \times 100\%$$

如果某个判断表达式由 4 个条件组成，那么对其进行条件组合覆盖测试时，需要设计 2^4 个，也就是 16 个测试用例；如果某个判断表达式由 6 个条件组成，那么对其进行条件组合覆盖测试时，需要设计 2^6 个，也就是 64 个测试用例。条件组合覆盖的缺点是，当一

个判定语句中条件较多时，条件组合数会很大，需要很多的测试用例。从便于测试的角度来说，在编写程序的时候，一个判定表达式中的条件个数不宜太多。

3.3.7 覆盖标准小结

覆盖标准用于描述在测试过程中对被测对象的测试程度，有时候也称为软件测试覆盖准则或者测试数据完备准则，它可以用于衡量测试是否充分，可以作为测试停止的标准之一，同时也是选取测试数据的依据，满足相同覆盖标准的测试数据集是等价的。

白盒测试覆盖标准是针对程序内部结构而言的，可以分为基于控制流的覆盖标准和基于数据流的覆盖标准。基于控制流的覆盖准则，可用于检查程序中的分支和循环结构的逻辑表达式，被工业界广泛采用。语句覆盖、判定覆盖、条件覆盖、条件/判定覆盖、条件组合覆盖、基本路径覆盖这些都属于基于控制流的覆盖标准，而基于数据流的覆盖标准则有 Rapps 和 Weyuker 的标准、Ntafos 的标准、Ural 的标准、Laski 和 Korel 的标准等。

不同的覆盖标准其测试的充分性是不一样的。如果说 A 标准的充分程度比 B 标准高，则意味着满足 A 标准的测试用例集合也满足 B 标准。语句覆盖、判定覆盖、条件覆盖、条件/判定覆盖、修正条件/判定覆盖、条件组合覆盖它们的测试充分程度存在如图 3-10 所示的强弱关系，例如，修正条件/判定覆盖高于条件/判定覆盖，而条件覆盖并不一定比语句覆盖强。

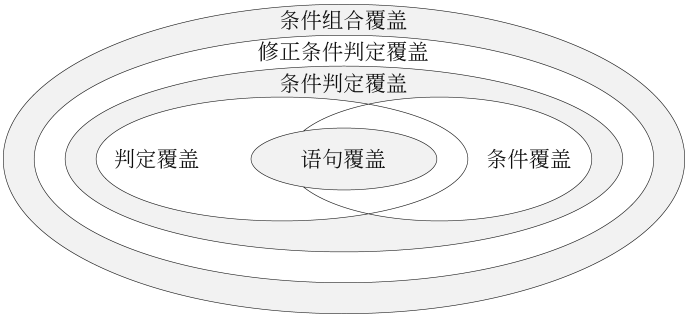


图 3-10 逻辑覆盖标准强弱关系图

测试覆盖标准的作用体现在以下多个方面。

(1) 可以定量地明确软件测试的要求和工作量。

对一段程序进行测试时，按照不同的测试标准，测试的要求和测试的工作量是不一样的。例如，对某一小段程序进行条件组合覆盖可能需要 8 个测试用例，而条件覆盖只需要 2 个测试用例，因为条件组合覆盖标准高于条件覆盖标准。

(2) 可以体现测试的充分程度。

根据逻辑覆盖标准，以及相应的覆盖率统计，可以体现测试进行的充分程度，覆盖标准越高，测试程度越高，覆盖率越高，测试越充分。例如，判定覆盖比语句覆盖测试程度更高。同样是判定覆盖，100%的覆盖率比 95%的覆盖率测试更充分。

(3) 是选取测试数据的依据。

在进行软件测试时，需要设计或者选择很多测试数据，覆盖标准就是选取测试数据的

依据，按照不同的逻辑覆盖标准，就会选取不同的测试数据。

（4）可以作为测试停止的标准。

过度的测试是一种浪费，测试工作不能一直进行下去，测试停止的依据可以有很多种，其中达到某种逻辑覆盖标准就可以作为依据之一。例如，在对程序进行测试时，要求达到修正条件/判定覆盖，那么当测试达到这样的测试标准之后，这项测试任务即算完成，测试可以停止。

（5）对测试结果和软件质量评估具有重要影响。

测试结果是与测试标准挂钩的，不同的覆盖标准对同一个软件的测试结果有可能是不一样的，软件能通过一个覆盖标准的测试，不一定能通过另外一个覆盖标准的测试。不同的覆盖标准在对软件的测试程度上有区别，根据测试通过的覆盖标准的不同，可以对软件质量给出不同的评价意见。

在软件测试实践中，需要按照测试覆盖标准来统计覆盖率，如统计语句覆盖率、判定覆盖率等，这样做的目的如下。

（1）提高测试效率。

通过覆盖率统计，可以发现并去除冗余无效的测试数据，减少测试次数，提高测试效率。例如，张三李四两位测试工程师一起设计测试用例，通过覆盖率统计发现，两人的测试用例合并时李四设计的一部分测试用例对提高覆盖率没有任何贡献，也就是这些测试用例是冗余的，应当去掉，以减少不必要的工作量。

（2）发现更多问题，提高产品质量。

通过覆盖率统计，可以清楚地描述程序被检验到了何种程度，发现软件中尚未测试过的部分，然后针对未测试或者测试不充分的地方继续测试，以发现更多问题，提高软件产品的质量。例如，通过覆盖率统计发现，模块 X 的覆盖率为 0，也就是说这个模块根本就没有被测试到，而模块 Y 的覆盖率也只有 30%，测试不够充分，此时应针对模块 X 和 Y 继续测试。

3.4 基本路径覆盖

在黑盒测试中，对所有可能的输入数据做穷举测试是行不通的，类似地，在白盒测试中，对一个具有一定规模的软件做路径穷举测试也是行不通的，只能在所有可能的执行路径中选取一部分来进行测试，基本路径覆盖就是其中的一种。在对程序做结构分析，尤其是进行基本路径覆盖时，要用到控制流图，下面先来看一下什么是程序的控制流图。

3.4.1 控制流图

控制流图也叫控制流程图，它用图的方式来描述程序的控制流程，是对一个过程或程序的抽象表达。控制流图是一种有向图，其形式化表达如下。

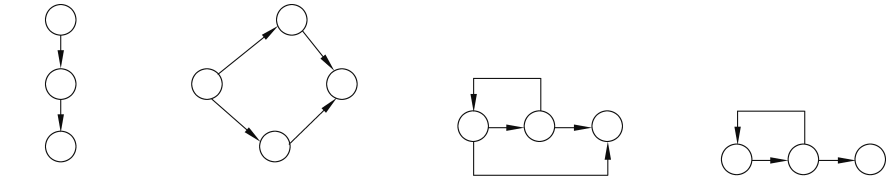
$$G = (N, E, N\_entry, N\_exit)。$$

其中，N 是节点集，程序中的每个语句都对应图中的一个节点，有时一组顺序执行、不存在分支的语句也可以合并为用一个节点表示。E 为边集。

E={<n1,n2> | n1, n2∈ **N** 且 n1 执行后,可能立即执行 n2}。

N_entry 和 N_exit 分别为程序的入口和出口节点,且 G 只具有唯一的入口节点 N_entry 和唯一的出口节点 N_exit。G 中的每个节点至多只能有两个直接后继节点。对于有两个直接后继的节点 v,其出边分别具有属性"T"或"F",并且在 G 中的任意节点 n,均存在一条从 N_entry 经 n 到达 N_exit 的路径。

在控制流图中,用节点来代表操作、条件判断及汇合点,用弧或者控制流线来表示执行的先后顺序关系。程序基本的控制结构对应的控制流图图形符号如图 3-11 所示。



(a) 顺序结构　(b) IF选择结构　(c) WHILE循环结构　(d) UNTIL循环结构

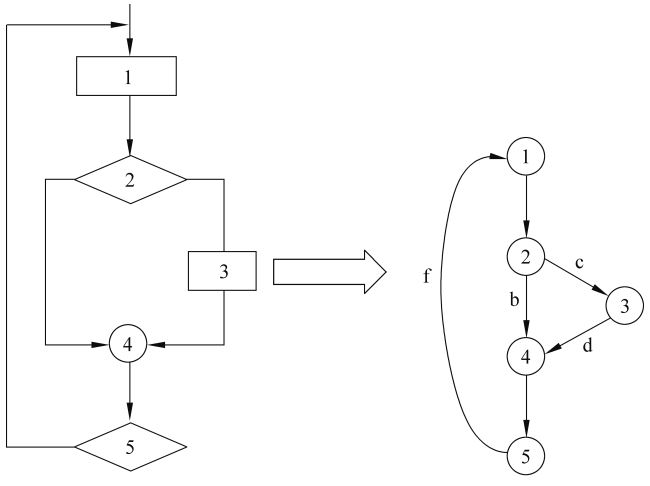
图 3-11　程序基本控制结构对应的控制流图

在图 3-11 所示的图形符号中,圆圈称为控制流图的一个节点,它表示一个或多个无分支的语句;有向箭头称为弧或者控制流线,表示执行的先后顺序关系。可以根据程序得出其控制流图,也可以由程序流程图来转换得到控制流图,但需要注意如下两点。

(1) 在将程序流程图转换成控制流图时,在选择或多分支结构中,分支的汇聚处应有一个汇聚节点。

(2) 如果判断中的条件表达式是由一个或多个逻辑运算符连接的复合条件表达式,则需要改为一系列只有单条件、嵌套的判断。

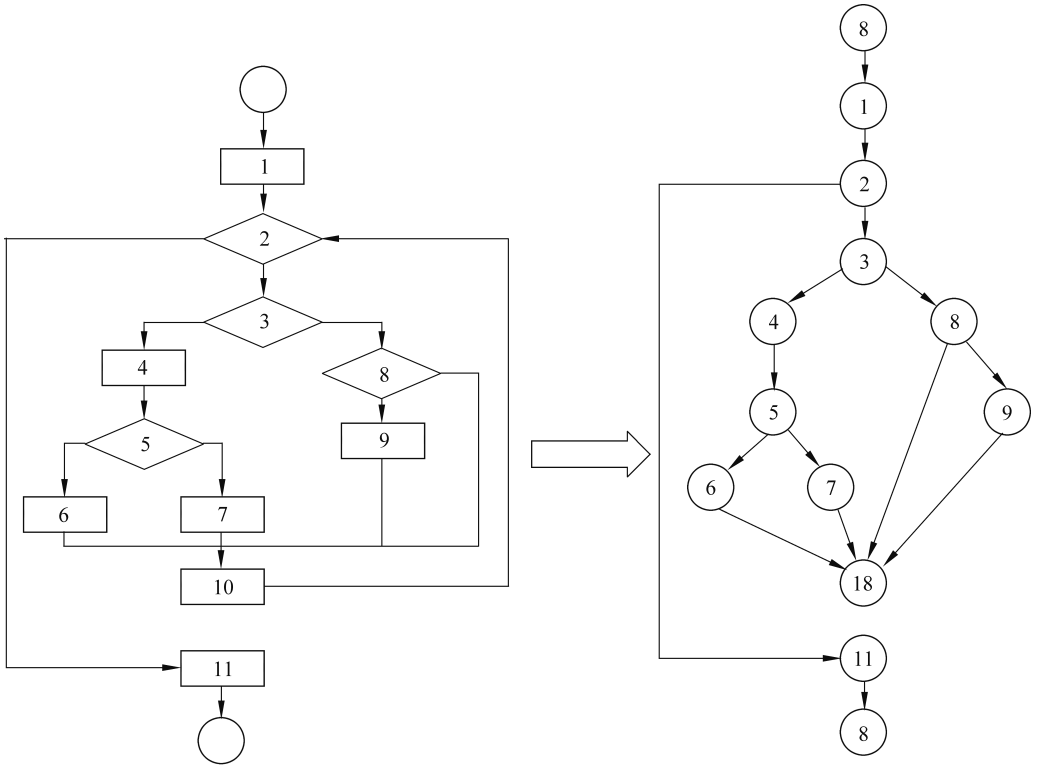
下面看几个例子。图 3-12(a)为一个局部的程序流程图,图 3-12(b)为由图 3-12(a)转换得到的控制流图。其中图 3-12(a)中标④的位置是没有节点的,只是分支的汇聚点,图 3-12(b)中的④号节点是由它转换得到的控制流图节点。



(a) 程序流程图　(b) 控制流图

图 3-12　程序流程图分支的汇聚点转换得到控制流图节点

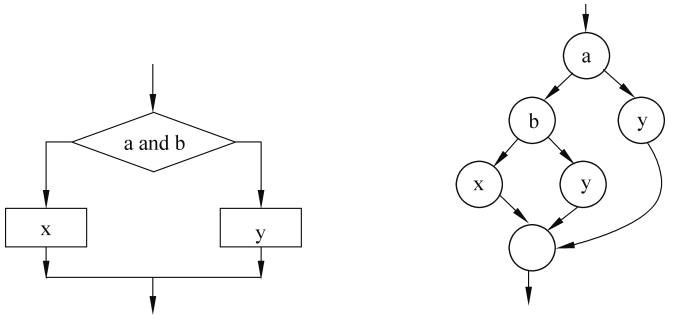
图 3-13(a)为一个完整的程序流程图，图 3-13(b)为由图 3-13(a)转换得到的控制流图。



(a) 程序流程图　　(b) 控制流图

图 3-13　完整的程序流程图及转换得到的控制流图

图 3-14(a)为一个带多条件判断框的程序流程图局部，图 3-14(b)为把多条件判断分解为多个单条件判断后得到的控制流图。



(a) 程序流程图　　(b) 控制流图

图 3-14　将多条件判断拆解为多个单条件判断示意图

3.4.2 环路复杂度

程序的复杂度如何度量呢？是否程序的大小就能准确反映程序的复杂程度呢？一个1000行的程序就一定比一个100行的程序复杂吗？答案是否定的。这就好比100道100以内加减法题并不比做一道二元积分题复杂是一样的道理。例如，一个由1000行顺序执行的赋值语句、输出语句组成的程序，并不比一个100行的排序算法程序复杂。用程序的大小来度量程序的复杂度是片面和不准确的，而环路复杂度是程序复杂度度量的方法之一。程序中的控制路径越复杂，环路越多，则环路复杂度越高，环路复杂度用来定量度量程序的逻辑复杂度。根据程序的控制流图，可以计算程序的环路复杂度。

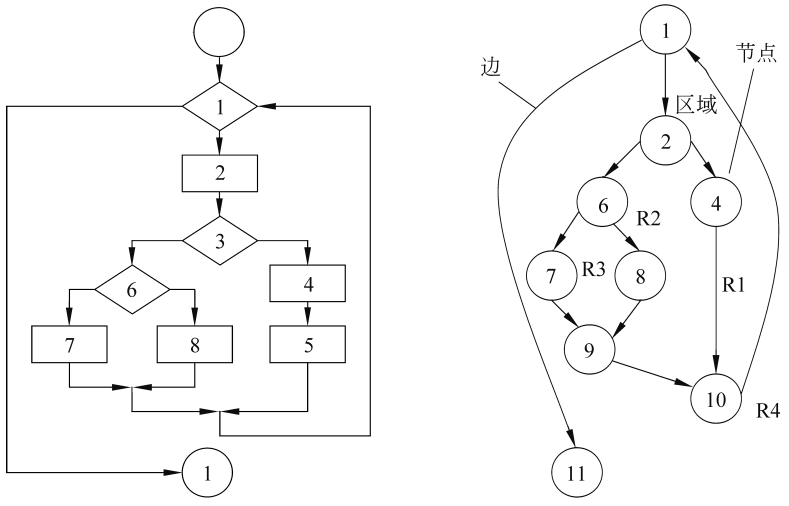
在画出控制流图的基础上，程序的环路复杂度可用以下3种方法求得。

(1) 环路复杂度为控制流图中的区域数。边和节点圈定的区域叫作区域，当对区域计数时，图形外的区域也应记为一个区域。

(2) 设 E 为控制流图的边数，N 为图的节点数，则环路复杂度为 $V(G)=E-N+2$。

(3) 若设 P 为控制流图中的判定节点数，则有 $V(G)=P+1$。

对于同一个控制流图，3种方法算出的结果是一样的。下面来看一个例子。图3-15(a)为一个程序的流程图，图3-15(b)为其对应的控制流图。



(a) 程序流程图　　(b) 控制流图

图3-15　程序流程图及其对应的控制流图

分别用3种方法来计算环路复杂度如下。

(1) 图中的区域数为4，故环路复杂度 $V(G)=4$。

(2) 边数 $E=11$，节点数 $N=9$，环路复杂度 $V(G)=E-N+2=4$。

(3) 图中的判定节点数 $P=3$，则有 $V(G)=3+1=4$。

3种方法算出的结果相等，环路复杂度为4。

3.4.3 基本路径覆盖

1. 程序中的路径

在把程序抽象为有向图之后，从程序入口到出口经过的各个节点的有序排列被称为路径，可以用路径表达式表示这样的一条路径。路径表达式可以是节点序列，也可以是弧序列，例如，如图 3-16 所示程序控制流图，其可能的程序执行路径见表 3-8。

表 3-8 可能的程序执行路径

路径编号	弧序列表示	节点序列表示
1	acde	1-2-3-4-5
2	abe	1-2-4-5
3	abefabe	1-2-4-5-1-2-4-5
4	abefabefabe	1-2-4-5-1-2-4-5-1-2-4-5
5	abefacde	1-2-4-5-1-2-3-4-5
…	…	…

需要注意的是，在程序中存在循环时，如果程序执行的循环次数不同，那么对应的执行路径就不同，例如表 3-8 中，路径 3 和路径 4 就是如此。为增强表达能力，可以在路径表达式中引入加法和乘方表达方式，加法可以表达分支结构，乘方可以表达循环结构。设有程序控制流图如图 3-17 所示，则它所有可能的路径可表达为：$(ac+bd)e(fe)^n$，其中 n 为循环的次数。

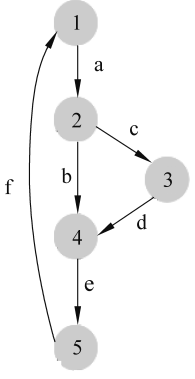


图 3-16 程序控制流图

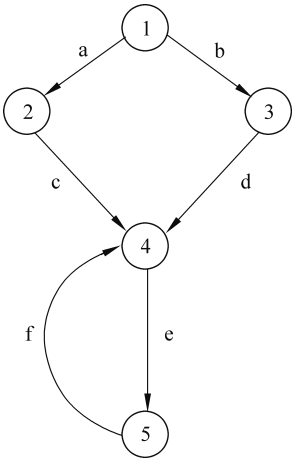


图 3-17 程序控制流图

2. 路径穷举测试不可行

一条 IF 语句就会有两条路径。两条 IF 语句的串联就会有四条路径，在实际问题中，