第 4 章　白盒测试

学习目标

1. 理解白盒测试的定义、静态白盒测试、动态白盒测试的定义、域测试、符号测试、路径覆盖和程序变异法。

2. 掌握静态白盒测试、动态白盒测试的常用方法、白盒测试方法的应用策略。

4.1　白盒测试定义

白盒测试与程序内部结构相关，因此也称结构测试或逻辑驱动测试。进行白盒测试时，测试者必须检查程序的内部结构，从程序的逻辑结构着手，得出测试数据。因此测试者需要了解程序结构的实现细节等知识，才能有效进行测试用例的设计工作。白盒测试并不是简单地按照代码设计用例，而是需要根据不同的测试需求，结合不同的测试对象，使用合适的方法进行测试。白盒测试的主要方法有逻辑覆盖测试、基本路径测试、程序插装等。

白盒测试又称为结构测试或逻辑驱动测试，是对软件的过程性细节做细致的检查，把测试对象看作一个打开的盒子，允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试，通过在不同点检查程序状态，确定实际状态是否与预期的状态一致。

白盒测试只测试软件产品的内部结构和处理过程，而不测试软件产品的功能，用于纠正软件系统在描述、表示和规格上的错误，是进一步测试的前提。白盒测试分为静态和动态两种：静态白盒测试是在不执行软件的条件下，有条理地仔细审查软件设计、体系结构和代码，从而找出软件缺陷的过程，有时也称为结构分析；动态白盒测试也称结构化测试，通过查看并使用代码的内部结构，设计和执行测试。

4.2　静态白盒测试

静态白盒测试也称为结构分析，是在不执行程序的条件下审查软件设计、体系结构和代码，从而找出软件缺陷的过程。测试对象是文档、代码等非计算机执行的部分。在项目中使用静态白盒测试是基于这样的原则：错误发现得越早，改正错误的成本越低，正确改正错误的可能性越大，改正错误时可能引发的其他错误的数量也越少。静态白盒测试方法包括代码检查法、静态结构分析法、静态质量度量法。常用的是代码检查法，这些方法在

程序开始编码之后、基于计算机的动态测试开始之前使用。

下面具体介绍三种常用的静态白盒测试方法。

1）代码检查法。

代码检查法主要检查代码和程序设计的一致性，代码结构的合理性，代码编写的标准性、可读性，代码逻辑表达的正确性等方面。主要参考文档为：程序设计文档、程序的源代码清单、编码规范、代码缺陷检查表等。

代码检查法能快速找到缺陷，一旦发现错误，能够在代码中对其进行精确定位，从而降低了错误修正的成本。代码检查看到的是问题本身而非问题的征兆。代码检查非常耗费时间，而且代码检查需要知识和经验的积累。

代码检查法包括代码审查、代码走查和桌面检查三种方式。

（1）代码审查和走查。

代码审查和走查这两种方法的形成、流程一样，规程、方法不一样。

代码审查和走查都是以小组为单位阅读代码，它是一系列规程和错误检查方法的集合。审查或走查小组通常由不需要对程序细节很了解的协调人员、程序的编码人员、程序的设计人员、测试专家四人组成，都是以会议的形式进行。会议理想时间为 90～120 分钟，按照每小时阅读 150 行代码的速度进行。对大型软件应安排多个会议同时进行，每个会议处理一个或几个模块或子程序。

代码审查规程和方法：在代码审查会议上，程序作者逐条语句讲述程序的逻辑结构，参与人根据“代码缺陷检查表”分析程序，检查内容包括编码标准规范和错误列表。编码规范是指团队根据自己的经验和风格进行设置的一些规范。错误列表一般是代码潜在的 Bug，由于某种代码写法虽然没有语法错误，但是可能存在其他错误，比如会导致线程死锁，这些都是错误列表应该检查的。程序员之间可以隔一定的时间抽取代码进行审查。结束会议后，把这些经验汇成列表，作为下次代码审查的依据，并针对错误修正进行跟踪。输出文档是“代码检查记录表”，此表主要内容有日期、主持人、参与人员、范围、发现的问题、问题处理、跟踪检查等。

代码走查规程和方法：在代码走查会议上，参与者参考“设计规格书”使用计算机来执行代码。测试人员准备一些简单的测试用例，它的作用是提供启动代码走查和质疑程序员逻辑思路及其他设想的手段。在会议期间，把测试数据沿程序的逻辑结构走一遍，程序的状态记录在纸或白板上以供监视。在大多数的代码走查中，很多问题是在向程序员提问的过程中发现的，而不是由测试用例本身直接发现的。

（2）桌面检查。

桌面检查是一种传统的检查方法，由程序员检查自己编写的程序。程序员在程序通过编译之后，对源程序代码进行分析、检验，并补充相关文档，由于程序员熟悉自己的程序及其程序设计风格，桌面检查由程序员自己进行可以节省时间，但应避免主观片面性。桌面检查的效果逊色于代码检查和走查，但桌面检查胜过没有检查。

2）静态结构分析法

主要是以图形的方式表现程序的内部结构。测试者通过使用测试工具分析程序源代码

的系统结构、数据结构、内部控制逻辑等内部结构，生成函数调用关系图、模块控制流图、函数内部控制流图等各种图形图表，清晰地标识整个软件的组成结构，便于理解，通过分析这些图表（包括控制流分析、数据流分析、接口分析、表达式分析），检查软件是否存在缺陷或错误。

静态结构分析法通常采用以下方法进行源程序的静态分析。

（1）通过生成各种图表，来帮助对源程序的静态分析，常用的各种引用表如下。

① 标号交叉引用表。列出在各模块中出现的全部标号，并标出标号属性，包括已说明、未说明、已使用、未使用等属性。表中还包括在模块以外的全局标号、计算标号等。

② 变量交叉引用表。变量交叉引用表即变量定义与引用表，在表中应标明各变量的属性，包括已说明、未说明、隐式说明以及类型及其使用情况等属性，进一步还可区分是否出现在赋值语句的右边，是否属于公共变量、全局变量或特权变量等属性。

③ 子程序（宏、函数）引用表。在表中列出各个子程序、宏和函数的属性，包括已定义、未定义、定义类型等属性，还要列出参数表，包括输入参数个数、顺序、类型，输出参数的个数、顺序、类型，已引用、未引用、引用次数等属性。

④ 等价表。等价表需要列出在等价语句或等值语句中出现的全局变量和标号。

⑤ 常数表。常数表需要列出全部数字常数和字符常数，并指出它们在哪些语句中首先被定义。

这些表可为源程序的静态分析提供辅助信息。例如，利用子程序（宏、函数）引用表、等价（变量、标号）表、常数表等，可以直接从表中查出说明/使用错误等；利用循环层次表、变量交叉引用表、标号交叉引用表等，可以做错误预测和程序复杂程度计算。

常用的各种关系图、控制流图主要有函数调用关系图和模块控制流图。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。用函数调用关系图可以检查函数的调用关系是否正确，是否存在孤立的函数而没有被调用，明确函数被调用的频繁度，对调用频繁的函数可以重点检查。通过查看函数调用关系图，可以发现系统是否存在结构缺陷，发现哪些函数是重要的，哪些是次要的，需要使用什么级别的覆盖要求等。

模块控制流图是由许多结点和连接结点的边组成的图形，其中每个结点代表一条或多条语句，边表示控制流向，模块控制流图可以直观地反映出一个函数的内部结构。

（2）静态错误分析。静态错误分析主要用于确定在源程序中是否有某类错误或“危险”结构。

① 类型和单位分析。类型和单位分析主要为了强化对源程序中数据类型的检查，发现在数据类型上的错误和单位上的不一致性。

② 引用分析。最广泛使用的静态错误分析方法就是发现引用异常。如果沿着程序的控制路径，变量在赋值以前被引用，或变量在赋值以后未被引用，这时就发生了引用异常。

为了检测引用异常，需要检查通过程序的每一条路径。通常采用类似深度优先的方法遍历程序流程图的每一条路径，也可以建立引用，引出探测工具，这种工具包括两个表，

定义表和未引用表。每张表中都包含一组变量表。未引用表中包含已被赋值但还未被引用的一些变量。

当扫描抵达一个长度大于 1 的结点 V 时，深度优先搜索算法要求先检查最左分支的那一部分程序流程图，然后再检查其他分支。在最左分支检查完之后，算法控制返回到结点 V，从栈中恢复该结点的定义表和未引用表的以前的副表，然后再去遍历该结点的下一个分支，这个过程要持续到全部分支检查完为止。

（3）表达式分析。对表达式进行分析，可以发现和纠正在表达式中出现的错误。表达式分析主要包括以下四个方面内容：

① 在表达式中不正确地使用了括号造成的错误；

② 数组下标越界造成错误；

③ 除数为 0 造成错误；

④ 对负数开平方，或对 π 求正切值造成错误。

最复杂的一类表达式分析是对浮点数计算的误差进行检查，由于使用二进制数不精确地表示十进制浮点数，常常使计算结果出乎意料。

（4）接口分析。接口分析可以检查模块之间接口的一致性和模块与外部数据库之间接口一致性。程序关于接口的静态错误分析主要检查过程、函数过程之间接口的一致性。因此，要检查形式参数和实际参数在类型、数量、维数、顺序、使用上的一致性，检查全局变量和公共数据区在使用上的一致性。

3）静态质量度量法

根据 ISO/IEC 9126 质量模型作为基础，我们可以构造质量度量模型，用于评估软件的各个方面。该模型从上到下分为三层，分别是质量因素、分类标准和度量规则。度量规则使用了代码行数、注释频度等参数度量软件的各种行为属性和度量规则参数表。

分类标准软件的可维护性采用以下四个分类标准来评估，分别是可分析性、可修改性、稳定性、可测性。每个分类标准由一系列度量规则组成，各个规则分配一个权重，由规则的取值与权重值计算出每个分类标准的取值。质量因素的取值与分类标准的计算方式类似，依据各分类标准取值组合权重方法计算。

4.3 动态白盒测试

动态白盒测试也称为结构化测试，是在使用和运行程序的条件下，软件测试员查看代码内部结构和实现方式来确定哪些要测试，哪些不要测试，如何开展测试，怎样设计和执行测试用例。动态白盒测试常用的测试用例设计方法有逻辑覆盖测试法（逻辑驱动测试法）和基本路径测试法两种。

4.3.1 逻辑覆盖测试法

测试覆盖率用于确定测试所执行到的覆盖项的百分比。其中的覆盖项是指作为测试基础的一个入口或属性，如语句、分支、条件等。测试覆盖率可以表示出测试的充分性，在

测试分析报告中可以作为量化指标的依据，测试覆盖率越高效果越好。但覆盖率不是目标，只是一种手段。测试覆盖率包括功能点覆盖率和结构覆盖率：功能点覆盖率用于表示软件已经实现的功能与软件需要实现的功能之间的比例关系；结构覆盖率包括语句覆盖率、分支覆盖率、循环覆盖率、路径覆盖率等。

逻辑覆盖测试法：以程序内部的逻辑结构为基础的用例设计方法，它通过对程序逻辑结构的遍历实现程序的覆盖。根据覆盖目标的不同，逻辑覆盖分为语句覆盖、判定覆盖（分支覆盖）、条件覆盖、判定-条件覆盖（分支-条件覆盖）、条件组合覆盖、路径覆盖六种覆盖测试方法。下面举例说明六种覆盖测试方法。

例 4.1　示例程序源代码。

```
int logicExample（int x，int y）
{
    int magic=0;
    if（x>0&&y>0）
        magic=x+y+10;          //语句块 1
    else
        magic=x+y-10;          //语句块 2
    if（magic<0）
        magic=0;               //语句块 3
    return magic;              //语句块 4
}
```

一般逻辑覆盖测试不会直接根据源代码，而是根据流程图来设计测试用例，根据例 4.1 示例程序源代码画出流程图，如图 4.1 所示。

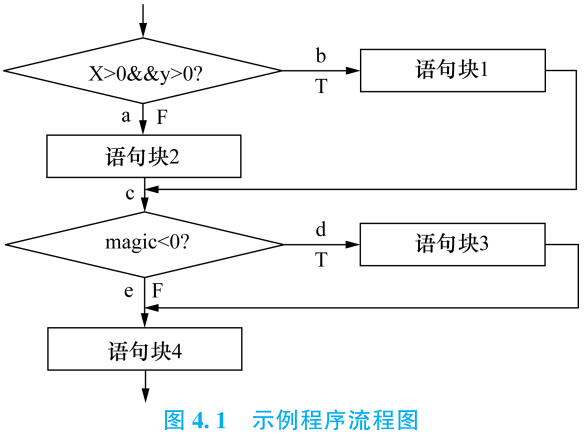


图 4.1　示例程序流程图

（1）语句覆盖。

语句覆盖要求设计足够多的测试用例，运行被测程序，使得程序中每条语句至少被执行一次。在本例中，可执行语句是指语句块 1 到语句块 4 中的语句。

例 4.1 的语句覆盖测试用例如表 4.1 所示。

表 4.1　语句覆盖测试用例

测试用例编号	输入数据	预期输出	覆盖语句
CASE1	x=3　y=3	magic=16	语句块 1、4
CASE2	x=-3　y=0	magic=0	语句块 2、3

通过这两个测试用例即达到语句覆盖的标准（测试用例组不是唯一的）。

我们可以直观地从流程图得到测试用例，测试所有的执行语句。但语句覆盖不能准确地判断运算中的逻辑关系错误。假设第一个判断语句 if（x>0&&y>0）中的“&&”被错误地写成了“‖”，即 if（x>0‖y>0），使用上面设计出来的一组测试用例来进行测试，仍然可以达到 100%的语句覆盖。在六种逻辑覆盖标准中，语句覆盖标准是最弱的。

（2）判定覆盖。

判定覆盖，又称分支覆盖，要求设计足够多的测试用例，运行被测程序，使得程序中的每个判断的“真”和“假”分支都至少被执行一次。在本例中共有两个判断 if（x>0&&y>0）和 if（magic<0）。

例 4.1 的判定覆盖测试用例如表 4.2 所示。

表 4.2　判定覆盖测试用例

测试用例编号	输入数据	预期输出	执行路径
CASE3	x=3　y=3	magic=16	b→c→e
CASE4	x=-3　y=0	magic=0	a→c→d→e

通过这两个测试用例，两个判断的真、假分支都已经被执行过，所以满足了判断覆盖的标准。

由于可执行语句要不就在判定的真分支，要不就在假分支上，判定覆盖比语句覆盖要多几乎一倍的测试路径，所以，只要满足了判定覆盖标准就一定满足语句覆盖标准。因此，判定覆盖比语句覆盖强。但判定覆盖会忽略条件中取或的情况。假设第一个判断语句 if（x>0&&y>0）中的“&&”被程序员错误地写成了“‖”，使用上面设计出来的一组测试用例，仍然可以达到 100%的判定覆盖，所以判定覆盖也无法发现上述的逻辑错误。

（3）条件覆盖。

条件覆盖要求设计足够多的测试用例，运行被测程序，使得判定中的每个条件获得各种可能的结果，即每个条件至少有一次为真值，有一次为假值。在本例中有两个判断 if（x>0&&y>0）和 if（magic<0），共计三个条件 x>0、y>0 和 magic<0。

例 4.1 的条件覆盖测试用例如表 4.3 所示。

表 4.3　条件覆盖测试用例

测试用例编号	输入数据	预期输出	执行路径
CASE5	x=3　y=0	magic=0	a→c→d→e
CASE6	x=-3　y=15	magic=2	a→c→e

通过这两个测试用例，三个条件的各种可能取值都满足了一次，达到了 100%条件覆盖的标准。

显然条件覆盖相比判定覆盖，增加了对符合判定情况的测试，增加了测试路径。但是条件覆盖只能保证每个条件至少有一次为真，而不考虑所有的判定结果。因此，条件覆盖并不能保证判定覆盖。

（4）判定-条件覆盖。

判定-条件覆盖要求设计足够多的测试用例，运行被测程序，使得被测试程序中的每个判断本身的判定结果（真/假）至少满足一次，同时，每个逻辑条件的可能值也至少被满足一次。即同时满足 100%判定覆盖和 100%条件覆盖的标准。

例 4.1 的判定-条件覆盖测试用例如表 4.4 所示。

表 4.4　判定-条件覆盖测试用例

测试用例编号	输入数据	预期输出	执行路径
CASE7	x=3　y=3	magic=16	b→c→e
CASE8	x=-3　y=0	magic=0	a→c→d→e

通过这两个测试用例，所有条件的可能取值都满足了一次，而且所有的判断本身的判定结果也都满足了一次。

达到 100%判定-条件覆盖标准一定能够达到 100%条件覆盖、100%判定覆盖和 100%语句覆盖。判定-条件覆盖满足判定覆盖准则和条件覆盖准则，弥补了二者的不足。但未考虑条件的组合情况。

（5）条件组合覆盖。

条件组合覆盖要求设计足够多的测试用例，运行被测程序，使得被测试程序中每个判定中条件结果的所有可能组合至少执行一次。其测试用例应该注意如下三点：

① 条件组合只针对同一个判断语句内存在多个条件的情况，让这些条件的取值进行笛卡尔乘积组合；

② 不同的判断语句内的条件取值之间无须组合；

③ 对于单条件的判断语句，只需要满足自己的所有取值即可。

例 4.1 的条件组合覆盖测试用例如表 4.5 所示。

表 4.5　条件组合覆盖测试用例

测试用例编号	输入数据	预期输出	执行路径
CASE9	x=3　y=3	magic=16	b→c→e
CASE10	x=-3　y=0	magic=0	a→c→d→e
CASE11	x=3　y=0	magic=0	a→c→d→e
CASE12	x=-3　y=15	magic=2	a→c→e

通过这四个测试用例，程序中所有条件取值的组合都被满足了一次。

条件组合覆盖准则满足判定覆盖、条件覆盖、判定-条件覆盖准则，线性地增加了测

试用例的数量。本例的程序存在三条路径，条件组合覆盖不能保证所有的路径被执行。

（6）路径覆盖。

路径覆盖要求设计足够的测试用例，运行被测程序，覆盖程序中所有可能的路径。

例 4.1 的路径覆盖测试用例如表 4.6 所示。

表 4.6　路径覆盖测试用例

测试用例编号	输入数据	预期输出	执行路径
CASE13	x=3　y=3	magic=16	b→c→e
CASE14	x=-3　y=0	magic=0	a→c→d→e
CASE15	x=-3　y=15	magic=2	a→c→e

本例中共有四条路径，其中路径 a→c→e 不可能实现，通过这三个测试用例，所有可能的路径都满足过一次。

路径覆盖测试方法可以对程序进行彻底的测试，比前面五种方法覆盖面都广。100%满足路径覆盖，一定能 100%满足判定覆盖标准，但并不一定能 100%满足条件覆盖，也就不能满足 100%条件组合覆盖。

从上例可知，单独采用任何一种逻辑覆盖方法都不能完全覆盖所有的测试用例，任何一个高效的测试用例，都是针对具体测试场景的。逻辑测试不是片面的测试正确的结果或是测试错误的结果，而是尽可能全面地覆盖每一个逻辑路径。所以在实际测试用例设计中，就要先从代码分析入手，根据不同的代码逻辑规则、语句执行情况，选用适合的覆盖方法。要根据不同需要和不同测试用例设计特征，将不同的设计方法组合起来，交叉使用，以实现最佳的测试用例输出。

4.3.2　基本路径测试法

基本路径测试法是在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例的方法。设计出的测试用例要保证被测程序的每个可执行语句至少被执行一次。

采用基本路径测试法设计测试用例，主要包括以下四个步骤。

（1）以详细设计或源代码作为基础，导出程序控制流图。

程序控制流图是描述程序控制流的一种图示方法，可以用图 4.2 的基本符号来描述程序结构。

控制流图由结点和控制流线（弧）两种图形符号组成。

结点以标有编号的圆圈表示，用于表示程序流程图中矩形框、菱形框的功能，是一个或多个分支的语句或源程序语句。

控制流线（弧）也称控制流图的边或链接，以箭头表示，与程序流程图中的流线功能一致，需注意以下两种情况：

① 分支的汇聚处应有一个汇聚结点，即使该结点并不代表任何语句；

② 由边和结点限定的范围称为区域，需要注意，图形外的区域也应记为一个区域。

例 4.2　程序的源代码如下。

```
void sort（int iRecordNum，int itype）
{
int x=0；int y=0；
while（iRecordNum>0）                    ①
    {
  if（itype==0）                         ②
        { x=y+2；break；}                ③
     else
        {
         if（itype==1）                  ④
           y=y+10；                      ⑤
         else
           y=y+20；                      ⑥
         iRecordNum=iRecordNum-1；       ⑦
    }
  }                                      ⑧
}
```

首先标注程序控制流图的结点，如图 4.2 图所示，注意分支的汇聚处应有一个汇聚结点。画出程序控制流图如图 4.3 所示。

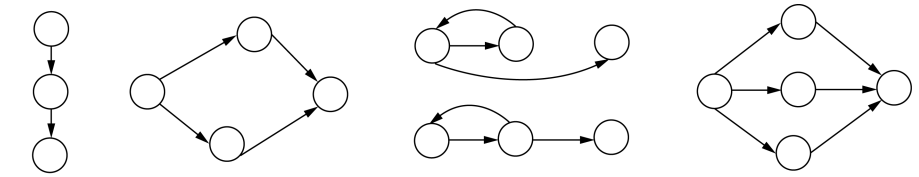


图 4.2　程序控制流图的基本符号示意图

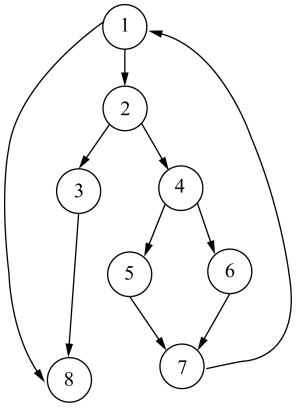


图 4.3　控制流图

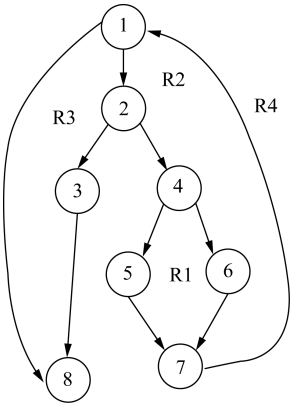


图 4.4　控制流图区域划分

（2）计算程序控制流图的环路复杂度。通过对程序控制流图的分析和判断，计算程序控制流图的环路复杂度。环路复杂度用 V（G）表示，有三种计算方法。

① 将环路复杂度定义为控制流图中的区域数。

如图 4.4 所示，程序控制流图将整个平面分成四个区域，R1、R2、R3 和 R4，因此，V(G)=4。

② 设定 E 为控制流图的边数，N 为图的结点数，则定义环路的复杂度为 V（G）=E-N+2。

如图 4.4 所示，E=10，N=8，V（G）=10-8+2=4。

③ 设定 P 为控制流图中的判定结点数，则有 V（G）=P+1。

如图 4.4 所示，图中的判定结点数 P=3，V（G）=3+1=4。

（3）确定独立路径集合。

从程序的环路复杂度可导出程序的独立路径条数。

独立路径，是指和其他的路径相比，至少引进一个新的处理语句集合或一个新判断条件的程序通路，即独立路径必须至少包含一条在定义之前不曾使用的边。如果只是已有路径的简单合并，并未包含任何新边，则不是独立路径。

独立路径集合中的每一条路径都是唯一的，但独立路径集合不是唯一的，可以有多组不同的独立路径集合。

如图 4.4 所示，可确定独立路径有如下四条。

第一条：①→⑧。

第二条：①→②→③→⑧。

第三条：①→②→④→⑤→⑦→①→⑧。

第四条：①→②→④→⑥→⑦→①→⑧。

设计测试用例，确保基本路径集合中每条路径的执行。

根据独立路径，来设计输入数据。为了确保独立路径集中的每一条路径的执行，根据判断结点给出的条件，选择适当的数据以保证每一条路径都被测试到。

依据步骤（3）中的独立路径，结合程序源代码，设计测试用例如表 4.7 所示。

表 4.7　基本路径测试用例

测试用例编号	输入数据	预期输出	覆盖路径
CASE1	iRecordNum=0　itype=0	x=0　y=0	①→⑧
CASE2	iRecordNum=1　itype=0	x=2　y=0	①→②→③→⑧
CASE3	iRecordNum=1　itype=1	x=0　y=10	①→②→④→⑤→⑦→①→⑧
CASE4	iRecordNum=1　itype=2	x=0　y=20	①→②→④→⑥→⑦→①→⑧

4.3.3　基本路径测试中的图形矩阵工具

图形矩阵是在基本路径测试中起辅助作用的软件工具，利用它可以实现自动地确定一

个基本路径集。

为了使导出程序控制流程图和决定基本测试路径的过程均自动化实现，科研人员开发了一个辅助基本路径测试的软件工具——图形矩阵（Graph Matrix），这个工具在进行基本路径测试时很有用。

利用图形矩阵可以实现自动地确定一个基本路径集。一个图形矩阵是一个方阵，其行/列数对应程序控制流程图中的结点数，每行和每列依次对应到一个被标识的结点，矩阵元素对应到结点间的连接。程序控制流程图的每一个结点都用数字加以标识，每一条边都用字母加以标识。如果在程序控制流程图中第 i 个结点到第 j 个结点有一个名为 x 的边相连接，则在对应的图形矩阵中第 i 行/第 j 列有一个非空的元素 x。

对每个矩阵项加入连接权值（Link Weight），图形矩阵就可以用于在测试中评估程序的控制结构，连接权值为控制流提供了额外的信息。在最简单情况下，连接权值是 1（存在连接）或 0（不存在连接），但是，连接权值也可以被赋予其他属性。比如：

（1）执行连接（边）的概率；

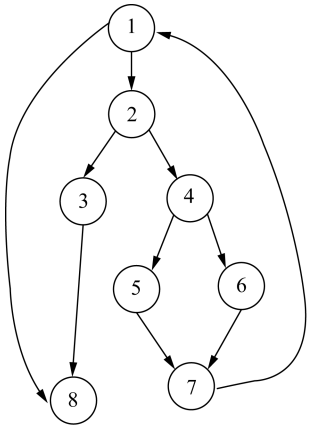
（2）穿越连接的处理时间；

（3）穿越连接时所需的内存；

（4）穿越连接时所需的资源。

根据上面的方法，画出图 4.5（a）所示的程序控制流程图对应的图形矩阵如图 4.5（b）所示。

如图 4.5（a）和 4.5（b）可知，连接权为“1”表示存在一个连接，在矩阵中如果一行有 2 个或更多的元素“1”，则这行所代表的结点一定是一个判定结点，通过矩阵中有 2 个以上（包括 2 个）元素为“1”的行的个数，就可以得到确定该图环路复杂性的另一种算法。



(a) 程序控制流图

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

(b) 图形矩阵

图 4.5　程序控制流图及其对应的图形

4.3.4 程序插桩法

在软件动态测试中，程序插桩是一种基本的测试手段，有着广泛的应用。

程序插桩方法是借助往被测程序中插入操作，来实现测试目的的方法，即向源程序中添加一些语句，实现对程序语句的执行、变量的变化等情况进行检查。

程序插装方法简单地说是通过往被测程序中插入操作来实现测试目的的方法。

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 4.6 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，其形式如下。

$$C(i) = C(i) + 1 \qquad i=1, 2, \cdots, 6$$

如图 4.6 所示，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。

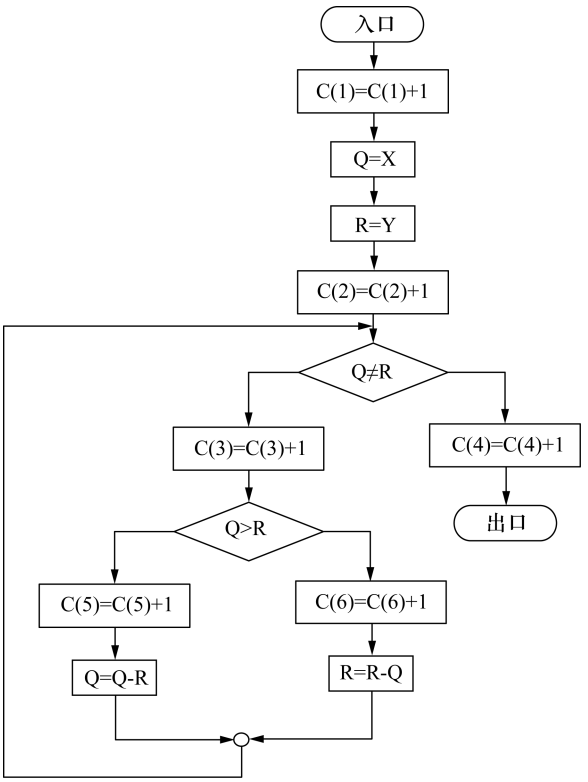


图 4.6 插桩后的求最大公约数程序流程图

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，等等。

设计程序插装程序时需要考虑的问题包括如下三个：

（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题目解决，并不能给出笼统的回答。第三个问题，需要考虑如何设置最少探测点的方案。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 Fotran 程序为例，列举至少应在如下部位设置技术语句：

（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、DOWHILE、DOUNTIL 及 DO 终端语句之后；

（5）BLOCK-IF、ELSEIF、ELSE 及 ENDIF 语句之后；

（6）LOGICALIF 语句处；

（7）输入/输出语句之后；

（8）CALL 语句之后；

（9）计算 GOTO 语句之后。

4.4　其他白盒测试方法

（1）域测试。

域测试是一种基于程序结构的测试方法。Howden 曾对程序中出现的错误进行分类，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，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。若程序的控制流有错误，对应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，也叫作域错误。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。还有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。

域测试的“域”是指程序的输入空间，其测试方法是基于对输入空间的分析。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通过被测程序产生正确的结果。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。子空间的划分是由程序中分支语句的谓词决定的。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，也可能出现无限循环而无出口的情况，输入空间中的元素都满足这些特定语句被执行所要求的条件。

基本路径测试法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。

域测试有两个致命的弱点，一个是为进行域测试而对程序提出的限制过多，另一个是当程序存在很多路径时，所需的测试也就很多。

（2）符号测试。

符号测试的基本思想是允许程序不仅仅输入具体的数值数据，也可以输入符号值，这一方法也因此而得名。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，符号测试可以被认为是普通测试的一个自然扩充。

符号测试可以看作是程序测试和程序验证的一个折中。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验证它的可靠性；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，实际上证明了程序接受此类输入所得到的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，就能较有把握地确认程序的正确性了。

从符号测试方法的使用来看，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和解释器。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；

④ 用解线性不等式方法求解路径条件，以求得满足各个限制谓词的测试数据；

⑤ 若上述不等式无解，则相应的路径不可执行；

⑥ 对可行路径进行测试。

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。而符号测试也存在以下三个未得到圆满解决的问题。

① 分支问题。

当采用符号测试进行到某一分支点处，分支谓词是符号表达式时，在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，需要测试人员做人工干预，或是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，这种情况通常出现在带有数组的程序中。比如下

面这段程序

```
…
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。

符号测试中总要处理符号表达式。随着符号测试的执行，一些变量的符号表达式越来越大。特别是如果当符号执行树很大，分支点很多时，路径条件本身将变成一个非常长的合取式。如果能够有办法将其简化，自然会带来很大好处。但如果找不到简化的办法，那会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。

（3）Z 路径覆盖。

分析程序中的路径是指：从入口开始检验程序，执行过程中经历的各个语句，直到出口为止。这是白盒测试最为典型的分析方法，着眼于路径分析的测试被称为路径测试，完成路径测试的理想情况是做到路径覆盖。对于比较简单的小程序实现路径覆盖是可能做到的。但是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，甚至达到天文数字，以至不可能实现路径覆盖。

为了解决这一问题，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。一般称简化循环意义下的路径覆盖为 Z 路径覆盖。

所谓的循环简化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是多少，只考虑循环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。如图 4.7（a）和图 4.7（b）所示为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（假定只执行一次），再经判断转出，其效果也与图 4.7（c）中给出的条件选择结构只执行右分支的效果一样。

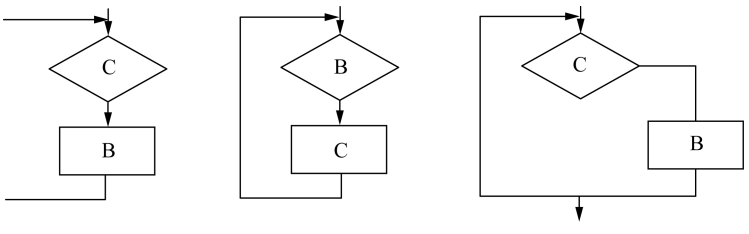


图 4.7　将循环结构简化成选择结构

对于程序中的所有路径可以用路径树来表示，其具体表示方法本文略。当得到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，就得到了所有的路径。

当得到所有的路径后，生成每个路径的测试用例，就可以做到 Z 路径覆盖测试。

（4）程序变异。

程序变异是一种错误驱动测试，错误驱动测试方法是针对某类特定程序错误的。经过多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搜索错误的范围尽可能地缩小，以利于专门测试某类错误是否存在。这样做便于将目标集中到对软件危害最大的可能错误，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。为便于测试人员使用变异方法，一些变异测试工具陆续被开发出来。

4.5 白盒测试方法应用策略

静态测试包括对软件产品的设计规格说明书的审查，对程序代码的阅读、审查等。静态分析的查错和分析功能是其他方法所不能替代的，已被当作一种自动化的代码校验方法。

动态测试是通过观察代码运行时的动作，来提供执行跟踪、时间分析，以及测试覆盖率方面的信息。动态测试通过真正运行程序发现错误。通过有效的测试用例，对应的输入/输出关系来分析被测程序的运行情况。

不同的测试方法各自的目标和侧重点不一样，在实际工作中，应将这两类方法结合起来运用，以达到更完美的效果。

以下是各种白盒测试方法的综合应用策略，可供在实际测试应用过程中参考。

（1）在测试中，应尽量先使用工具进行静态结构分析。

（2）测试中可采取先静态后动态的组合方式：先进行静态结构分析、代码检查，再进行覆盖率测试。

（3）利用静态分析的结果作为导引，通过代码检查和动态测试的方式对静态发现结果进行进一步的确认，使测试工作更为有效。

（4）覆盖率测试是白盒测试的重点，一般可使用基本路径测试法达到语句覆盖标准；对于软件的重点模块，应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，测试的侧重点不同：在单元测试阶段，以代码检查、逻辑覆盖为主；在集成测试阶段，需要增加静态结构分析等；在系统测试阶段，应根据黑盒测试的结果，采取相应的白盒测试。

以上的测试方法各有所长，每种方法都可设计出一组有用的测试用例，用这组测试用例可以比较容易地发现某种类型的错误，却不易发现另一种类型的错误。因此在实际测试中，应结合各种测试方法，形成综合策略。

4.6　本章小结

本章介绍了白盒测试的常用方法，在实践中发现，静态白盒测试和动态测试是互补的，缺少任何一种，都会降低错误检查的效率。项目组可根据项目的实际情况，来决定哪种白盒测试方法更适合。

习　题

1. 白盒测试的定义？白盒测试和黑盒测试的区别？
2. 逻辑覆盖测试方法包括哪些？哪一种覆盖率最高？为什么？
3. 请把下面的程序流程图转换成控制流图。

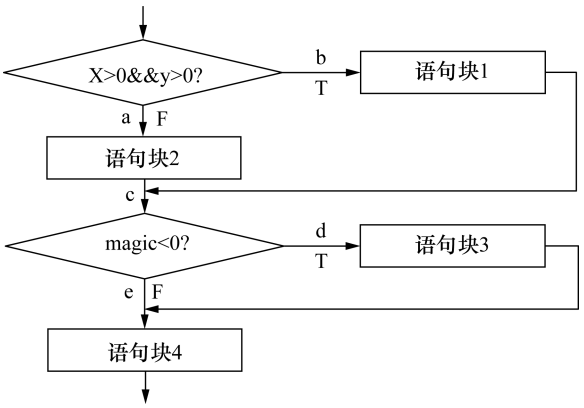


图 4.8　程序流程图