CHAPTER

第3章 白盒测试

【本章要点】

- 白盒测试的概念；
- 逻辑覆盖测试；
- 常见的边界值类型；
- 基本路径的计算；
- 循环语句的处理；
- 程序插桩的方法。

【本章目标】

- 理解并掌握白盒测试的概念；
- 了解白盒测试和调试的异同；
- 理解并掌握各种白盒测试技术方法及其特点和适用情况。

3.1 白盒测试概述

在软件工程中，简单地说白盒测试就是一种用于检查代码是否按照预期工作的验证技术，又称结构测试(Structural Testing)、逻辑驱动测试(Logic-driven Testing)或基于程序的测试(Program-based Testing)。“白盒”是指可视的，“盒子”是指被测试的软件。所以说白盒测试是一种可视的测试软件的方法，即它把测试对象看作一个透明的盒子，测试人员要了解程序结构和处理过程，按照程序内部逻辑测试程序，检查程序中的每条通路是否按照预定要求工作。白盒测试的主要特点就是它主要针对被测程序的源代码，测试者可以完全不考虑程序的功能。白盒测试的过程如图 3-1 所示。

读者可能会问，用户在使用软件的过程中关心的只是软件的功能，为什么在软件测试的过程中要花费时间和精力来做白盒测试呢？

其中的一个原因就在于软件自身存在的缺陷：

(1) 逻辑错误和不正确假设与一条程序路径被运行的可能性成反比。当主要功能、条件或控制完成后，常常会在后续的工作中开始出现错误，设计者通常能够很好了解常用功能，但当处理特殊情况时则容易出现问题，并且难以被发现。

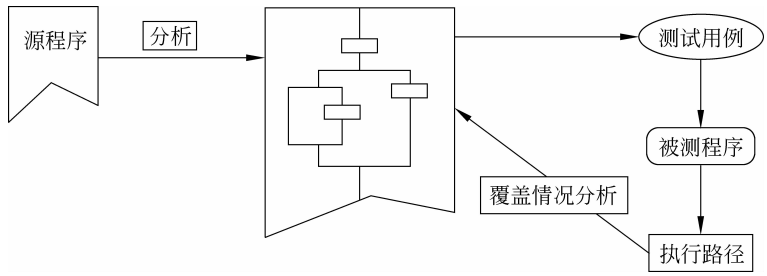

图 3-1　白盒测试过程示意图

(2) 很多读者经常认为某条逻辑路径不可能被执行,但程序的逻辑流有时是和直觉不一致的,也就是说关于控制流和数据流的一些无意识的假设可能导致设计错误,此时只有路径测试才能发现这些错误。

(3) 随机错误难以避免。把一个程序翻译为程序设计语言源代码后,有可能产生某些笔误,虽然语法检查机制能够发现很多错误。但是,还有一些错误只有在测试开始时才能发现。而错误在每个逻辑路径上出现的几率是一样的。

另外一个原因就在于功能测试本身的局限性。简单地说,如果程序实现了没有被描述的行为,功能测试是无法发现的(病毒就是这样一个例子),这将会给软件带来隐患,而白盒测试就能够发现这样的缺陷。正如 Beizer 所说:“错误潜伏在角落里,聚集在边界上。”相对而言白盒测试能够更容易地发现它。

白盒测试方法大体可分为静态分析和动态测试。但是,白盒测试的用例设计技术有多种,在稍后的章节中将会作具体介绍。那么,在对被测软件进行白盒测试时,主要对程序进行哪些方面的检查呢? 有如下几点:

(1) 保证一个模块中的所有独立执行路径至少测试一次;

(2) 对所有逻辑判定取值 true 和 false 的两种情况都至少测试一次;

(3) 在循环边界和运行界限内执行循环体;

(4) 测试内部数据结构的有效性。

在软件测试领域,有六种基本的测试类型:单元测试、集成测试、功能测试/系统测试、可接受性测试、回归测试和 Beta 测试。白盒测试可以用在其中的三种测试类型中。

1. 单元测试

单元指的就是一个不能够再分割成其他组件的组件。那么单元测试就是对一组相关组件或单元的独立测试。软件工程师书写白盒测试用例的目的就是用来检查单元编码是否正确。单元测试是十分重要的,因为在单元与其他代码集成之前,它能够确保代码的可靠性。代码一旦同底层基代码集成到一起,就难以对所发现的软件缺陷进行定位。而且,因为是软件工程师自己书写和运行单元测试,软件公司常常不对单元测试过程中所发现的缺陷进行跟踪,也就是不公开单元测试的缺陷。因此,最好自己先找出错误,在还没有提交给测试人员之前先修复它。研究表明,大约有 65%的缺陷可以在单元测试中发现。

2. 集成测试

集成测试就是对集成到一起的软件组件和硬件组件进行的测试,用于评估这些组件

之间能否进行正确的交互。书写集成测试用例的主要目的就是为了检查各种组件之间的接口。如果测试员能够很好地了解某个测试用例需要多个程序单元进行交互，那么在集成测试中可以使用黑盒测试用例。另外，测试员也可以书写白盒测试用例来检查他们所熟悉的各个单元接口。

3. 回归测试

回归测试是一种具有选择性的对系统或组件的重复测试，用来验证对软件所做的修改没有带来不良的影响，系统或组件仍然符合特定的需求。正如集成测试一样，回归测试既可以使用黑盒测试，也可以使用白盒测试，或者把二者组合起来进行测试。白盒的单元测试和集成测试用例都可以保存起来，作为回归测试的一部分重新运行。

在白盒测试过程中，我们必须使用预先确定的输入来运行代码以便检查程序的正确性，确保代码能够产生预期的输出。因此，程序员通常要书写桩模块和驱动模块进行白盒测试。驱动模块就是用于触发被测模块的一个软件模块，一般要提供测试输入、控制和监测并报告测试结果。最简单的形式就是使用一行能够调用一个方法的代码。例如，如果想移动球场上的一个运动员，驱动代码可能是这样，即 movePlayer(Player, diceRoll)；这个驱动代码可能被主方法调用。而白盒测试用例将要执行这行驱动代码并且检查运动员的位置(如使用 player. getPosition()方法)，以确保运动员现在处于运动场上的相应位置。桩模块就是能够代替软件模块的计算机程序语句，可以模拟实际组件行为的组件或对象。例如，若 movePlayer 方法还没有完成，那么就可以暂时使用下面所示的代码，把运动员移动到标识为 1 的位置。

```
Public void movePlayer(Player player,int diceValue){
Player.setPosition(1);}
```

当然，最后要由正确的程序逻辑来代替这个方法。但是，开发桩模块的程序员可以调用正在开发的代码中的方法，甚至是一个还没有规定预期行为的方法。桩模块和驱动模块通常被看作是随时可以抛弃的代码，但是可以通过填写这些代码来实现真正的方法，也可以把驱动模块作为自动化测试用例。

3.1.1 白盒测试与调试的异同

白盒测试与调试的最终目的都是让被测应用(AUT)可以正常安全地运行，都是保证软件质量过程的一个环节。那么，白盒测试和调试有哪些不同呢？

从承担的任务来看，白盒测试同其他类型测试一样，它的任务是发现所开发的项目中的缺陷；但是，调试不属于测试，其任务是纠正软件中的缺陷。

从最终的结果来看，白盒测试有预知的结果，不可预知的只是程序是否通过测试，并且成功测试的结果是发现错误的症状，从而引起调试的进行；而调试的结果是消除项目中的错误。

从执行的过程来看，软件测试只是发现程序中有错误的迹象，没有错误定位，也不需要找到出错原因；软件调试是根据测试报告的记录，在软件测试后纠正错误的工作，包括确定错误位置和修改错误。测试是一个发现错误、改正错误、重新测试的过程；而调试是

一个推理过程。

从准备工作来看,测试从已知的条件开始,使用预先定义的程序;调试一般是以不可知的内部条件开始,做统一性调试。

从执行的计划性来看,测试是有计划的并要进行测试设计;而调试则不受时间约束。

测试的执行是有规程的;而调试的执行往往要求程序员进行必要推理以至知觉的“飞跃”。

从执行的人员来看,测试经常是由独立的测试组在不了解软件设计的条件下完成的,而调试必须由程序员来完成。

从所使用的工具来看,大多数白盒测试的执行和设计可由工具支持,而调试程序员能利用的工具主要是调试器。

3.1.2 白盒测试的分类

程序的结构形式是白盒测试的主要依据。程序结构主要用流程图 N-S 图来表示程序 的执行路径数目庞大,让程序的所有路径都执行一次是不可能的。对一个具有多重选择和循环嵌套的程序,有无数个不同的路径。如图 3-2 所示的一个小程序的流程图,它包括了一个执行 20 次的循环。包含的不同执行路径数达 5^{20} 条,对每一条路径进行测试需要 1 毫秒,假定一年工作 365×24 小时,要想把所有路径测试完,需 3170 年。

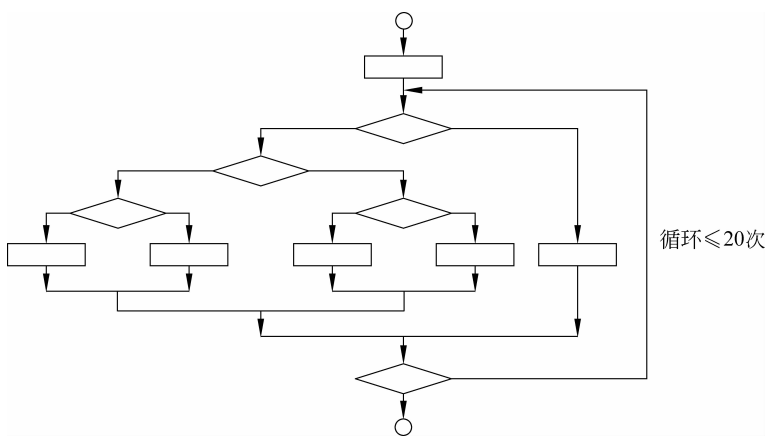

图 3-2 循环程序流程图

由此可见,彻底的测试(穷举测试)是无法实现的。但是为了检查程序的正确性,每完成一个代码模块时,却需要设计一定的测试用例。因此,为了节省时间和资源,提高测试效率,就必须采用一些方法和技巧有选择地设计测试用例,以达到最佳的测试效果。白盒测试用例设计技术就是研究如何用最少的测试用例来最大限度地发现软件中的错误。常用测试用例设计方法有:

(1) 逻辑覆盖测试;

(2) 边界值测试;

(3) 基本路径测试;

（4）循环语句测试；
（5）程序插桩测试；
（6）数据流测试；
（7）变异测试。

3.2 白盒测试用例设计技术

3.2.1 逻辑覆盖测试

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，弄清测试进行到什么程度才可以结束测试。可以根据实际情况并参考如下结构测试的覆盖准则进行决定。

1. 语句覆盖

例 3-1

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

上述程序段的流程图如图 3-3 所示。

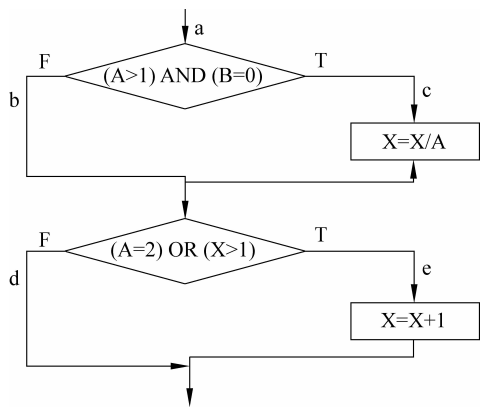

图 3-3 例 3-1 程序流程图

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。如果选用的测试用例是：

```
A=2
B=0        CASE 1
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，从而达到了语句覆盖。

如果选用的测试用例是：

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

则程序按路径 abe 执行。此时该程序段只执行了其中的 3 个语句，所以未达到语句覆盖。

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。但即使程序中每个语句都得到执行，也不能保证程序完全正确。

假如这一程序段中两个判断的逻辑运算有问题：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。这时仍使用测试用例 CASE 1，程序仍将按路径 ace 执行。这时虽然也达到了语句覆盖，却发现不了判断中逻辑运算的错误。

与其他几种逻辑覆盖比较起来，语句覆盖是比较弱的覆盖原则。达到了语句覆盖可能给人们一种心理的满足，以为每个语句都测试过，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。这是因为被测程序并非语句的无序堆积，语句之间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的情况至少经历一次，即判断的真假值均曾被满足。所以，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分别是：

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

若选用另外两组测试用例：

```
CASE 4       CASE 5
 A=3           A=2
 B=0           B=1
 X=3           X=1
```

则可分别执行路径 acd 和 abe。同样使两个判断的 4 个分支 c、e 和 b、d 分别得到覆盖。

上述两组测试用例不仅满足了判定覆盖，同时还达到了语句覆盖的目的。但是，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 CASE 5，照样能按原路径执行(abe)，而不影响结果。所以，判定覆盖只能达到判定覆盖仍无法确定判断内部条件的错误。

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≤1 时，记为 $\overline{T}_1$；

B=0 取真值，记为 T_2；

B=0 取假值，即 B≠0 时，记为 $\overline{T}_2$

第二个判断应考虑到：

A=2 取真值，记为 T_3；

A=2 取假值，即 A≠2 时，记为 $\overline{T}_3$；

X>1 取真值，记为 T_4；

X>1 取假值，即 X≤1，记为 $\overline{T}_4$；

表 3-1 中给出了 3 个测试用例：CASE6、CASE7、CASE8，执行该程序段所走路径及覆盖条件。

表 3-1　条件覆盖测试用例 1

测试用例	A	B	X	路径	覆 盖 条 件
CASE6	2	0	3	ace	T_1, T_2, T_3, T_4
CASE7	1	0	1	abd	$\overline{T}_1, T_2, \overline{T}_3, \overline{T}_4$
CASE8	2	1	1	abe	$T_1, \overline{T}_2, T_3, \overline{T}_4$

从表 3-1 中可以看到，3 个测试用例把 4 个条件的 8 种情况均做了覆盖。

进一步分析后，可以发现这些测试用例在覆盖了 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、e 似乎也覆盖了。这样是否可以说达到了条件覆盖，也实现了判定覆盖呢？来分析另一种情况，假定选用两组测试用例是 CASE8 和 CASE9，执行程序段的覆盖情况如表 3-2 所示。

表 3-2　条件覆盖测试用例 2

测试用例	A	B	X	所走路径	覆盖分支	覆 盖 条 件
CASE8	1	0	3	abe	be	$\overline{T}_1, T_2, \overline{T}_3, T_4$
CASE9	2	1	1	abe	be	$T_1, \overline{T}_2, T_3, \overline{T}_4$

这一覆盖情况表明，覆盖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。为了解决这一矛盾，需要兼顾条件和分支覆盖的需求。

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。

例子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：

(1) A>1，B=0 记为 T_1T_2；

(2) A>1，B≠0 记为 $T_1\overline{T}_2$；

(3) A≤1，B=0 记为 $\overline{T}_1T_2$；

(4) A≤1，B≠0 记为 $\overline{T}_1\overline{T}_2$；

(5) A=2，X>1 记为 T_3T_4；

(6) A=2，X≤1 记为 $T_3\overline{T}_4$；

(7) A≠2，X>1 记为 $\overline{T}_3T_4$；

(8) A≠2，X≤1 记为 $\overline{T}_3\overline{T}_4$。

这里设计了 4 个测试用例，用以覆盖上述 8 种条件组合，如表 3-3 所示。

表 3-3　判定-条件覆盖

测试用例	A	B	X	覆盖组合号	所走路径	覆 盖 条 件
CASE1	2	0	3	(1)(5)	ace	T_1,T_2,T_3,T_4
CASE8	2	1	1	(2)(6)	abe	$T_1,\overline{T}_2,T_3,\overline{T}_4$
CASE9	1	0	3	(3)(7)	abe	$\overline{T}_1,T_2,\overline{T}_3,T_4$
CASE10	1	1	1	(4)(8)	abd	$\overline{T}_1,\overline{T}_2,\overline{T}_3,\overline{T}_4$

注意到，这一程序共有 4 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，有的虽然提到了所走路径问题，但尚未涉及路径的覆盖，而能否全面覆盖路径在软件测试中仍是个重要问题，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。只有程序中每一条路径都进行了检验，才能说对程序进行了全面检验。

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覆盖程序中所有可能的路径。

针对例 3-1 中的程序有 4 条可能路径

ace 记为 L1

abd 记为 L2

abe 记为 L3

acd 记为 L4

这里给出 4 个测试用例：CASE1、CASE7、CASE8 和 CASE11，使其分别覆盖这 4 条路径，见表 3-4。

表 3-4　路径覆盖

测试用例	A	B	X	覆盖路径
CASE1	2	0	3	ace
CASE7	1	0	1	abd
CASE8	2	1	1	abe
CASE11	3	0	1	acd

这里所用的程序很短，只有 4 条路径。在实际应用中一般不太复杂的程序，其路径数都是一个庞大的数字，要在测试中覆盖所有路径是不可能的。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，只执行一次程序中的循环体。但即使对于路径数很有限的程序已经达到了路径覆盖，仍然不能保证被测程序的正确性。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。想要在有限的时间内用有限的方法来发现所有的程序错误是不可能的，这涉及有关软件测试局限性的问题。

3.2.2 边界值分析

等价类划分和边界值分析为软件测试提供了一种设计白盒测试用例的策略。毫无疑问，在测试过程中应该经常考虑使用这两种方法。例如，如果某个人想买一座房子，但是它可能有也可能没有足够的资金。这时就可以考虑使用这两种方法，那么应该包括如下几个测试用例：

(1) 房子的价格是 100 万元，买主有 200 万元现金(有足够资金购买房子)；

(2) 房子的价格是 100 万元，买主有 50 万元现金(没有足够资金购买房子)；

(3) 房子的价格是 100 万元，买主有 99 万元现金(边界值)；

(4) 房子的价格是 100 万元，买主有 101 万元现金(边界值)。

在测试的过程中，对于程序中存在的循环，考虑使用等价类划分的方法测试时要使用正常值来执行循环操作。如使用边界值分析方法来测试，一定要给循环条件赋予低于正常值、正常值、高于正常值等边界值。

3.2.3 基本路径测试

基本路径测试是 Tom McCabe[MCC76]首先提出的一种白盒测试技术，允许测试用例设计者导出一个过程设计的逻辑复杂性测度，并使用该测度作为指南来定义执行路径的基本集。从该基本集导出的测试用例保证对程序中的每一条语句至少执行一次。

在使用基本路径方法设计时要使用到流图或程序图。流图使用符号来描述逻辑控制流，每一种结构化构成元素有一个相应的流图符号。其中，圆代表一个或多个语句，称为流图的节点；一个处理方框序列和一个菱形决策框可被映射为一个节点；流图中的箭头，称为边或连接，代表控制流，类似于流程图中的箭头；一条边必须终止于一个节点，即使该节点并不代表任何语句。由边和节点限定的范围称为区域。计算区域时应包括图外部的范围。任何过程设计表示法都可被翻译成流图。当程序设计中遇到复合条件时，生成的流图就会变得更为复杂。当条件语句中用到一个或多个布尔运算符(如逻辑 OR、AND、NAND、NOR)时，就出现了复合条件。包含条件的节点被称为判定节点，可以从每一个判定节点发出两条或多条边。

下面引入环形复杂度的概念。环形复杂度是一种为程序逻辑复杂性提供定量测度的软件度量，将该度量用于基本路径方法，计算所得的值定义了程序基本集的独立路径数量，并提供了确保所有语句至少执行一次的测试数量的上界。

独立路径是指程序中至少引进一个新的处理语句集合或一个新条件的任一路径。采用程序图的术语,即独立路径必须至少包含一条在定义路径之前不曾用到的边。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。如果能将测试设计为强迫运行这些路径(基本集),那么程序中的每一条语句将至少被执行一次,每一个条件执行时都将分别取 true 和 false。应该注意到基本集并不唯一,实际上,给定的过程设计可派生出任意数量的不同基本集。

那么,如何才能知道需要寻找多少条路径呢?由于环形复杂度是以图论为基础,能够提供非常有用的软件度量,因此通过对环形复杂度的计算可以得到这个问题的答案。可用如下 3 种方法之一来计算复杂性:

(1) 控制流图中区域的数量对应于环形的复杂性。

(2) 控制流图 G 的环形复杂度——$V(G)$,定义为 $V(G)=E-N+2$,E 表示控制流图中边的数量,N 表示程序图中节点数量。

(3) 控制流图 G 的环形复杂度——$V(G)$,也可定义为 $V(G)=P+1$,P 是控制流图 G 中判定节点的数量。

更重要的是,$V(G)$的值提供了组成基本集的独立路径的上界,并由此得出覆盖所有程序语句所需的测试设计数量的上界。下面根据环形复杂度的值讨论如何导出测试用例,步骤如下:

(1) 以设计或代码为基础,画出相应的控制流图。

(2) 确定所得程序图的环形复杂性。可采用上面给出的任意一种算法来计算环形复杂性——$V(G)$。

(3) 确定线性独立的路径的一个基本集。$V(G)$的值提供了程序控制结构中线性独立的路径的数量。

(4) 准备测试用例,强制执行基本集中每条路径。测试人员可选择数据以便在测试每条路径时适当设置判定节点的条件。

(5) 执行每个测试用例,并和期望值比较,一旦完成所有测试用例,测试者可以确定在程序中的所有语句至少被执行一次。

要注意的是,某些独立路径不能以独立的方式被测试,即穿越路径所需的数据组合不能形成程序的正常流,在这种情况下,这些路径必须作为另一个路径测试的一部分进行测试。具体过程请读者参考例 3-2。

例 3-2

```
public void Sort(int iRecordNum,int iType)
0  {
1  int x=0;
2  int y=0;
3  while (iRecordNum--)
4  {
5  if(iType==0)
6  x=y+2;
7  else
```

```
8  if(iType==1)
9  x=y+10;
10  else
11  x=y+20;
12  }
13  }
```

画出其对应的控制流图，如图 3-4 所示。

如果在程序中遇到复合条件，例如，条件语句中有多个布尔运算符（逻辑 OR、AND）时，为每一个条件创建一个独立的节点，其中包含条件的节点称为判定节点，从每一个判定节点发出两条或多条边。

例 3-3

```
1  if(a‖b)
2  x=5;
3  else
4  y=10;
5  …
```

对应的逻辑如图 3-5 所示。

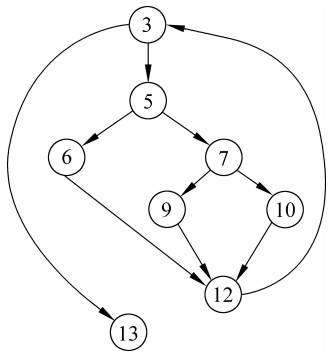

图 3-4　控制流图

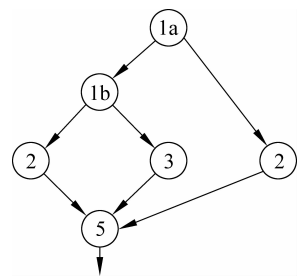

图 3-5　逻辑图

（1）计算圈复杂度。

根据前面公式，对应图 3-5 中代码的圈复杂度，计算如下：程序图中有 4 个区域；$V(G)$=11 条边－9 节点＋2＝4；或 $V(G)$=3 个判定节点＋1＝4。

（2）导出测试用例。

根据上面的计算方法，可得出 4 个独立的路径：

路径 1　3-13(iRecordNum＝0)

路径 2　3-5-6-12-3-13(iRecordNum ≧ 0,iType＝0)

路径 3　3-5-7-9-12-3-13(iRecordNum ≧ 0,iType＝1)

路径 4　3-5-7-10-12-3-13(iRecordNum ≧ 0,iType≠1,iType≠0)

最后，就可以根据上面的独立路径，去设计输入数据，使程序分别执行到上面 4 条路径。如果取 iRecordNum＝3;iType＝1，那么将遍历路径 4，预期的结果为 x＝10。

3.2.4 循环语句测试

循环测试是一种白盒测试技术,注重于循环构造的有效性。n 循环结构测试用例的设计循环可以划分为以下几种模式,如图 3-6 所示。

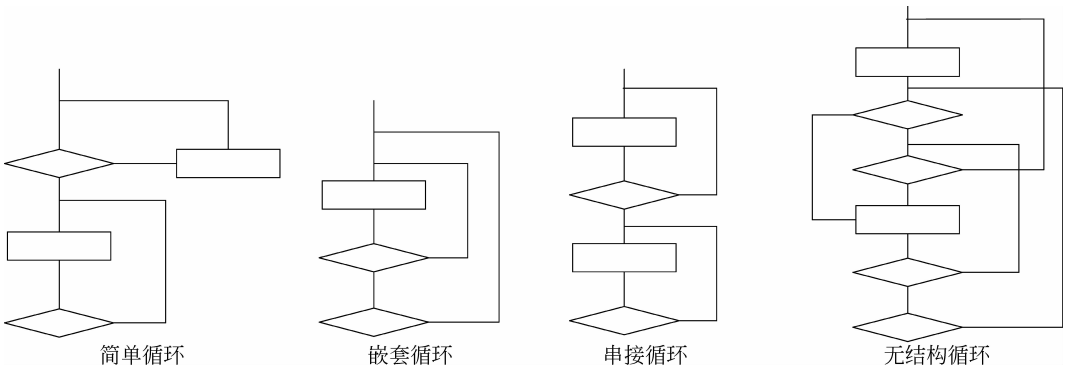

图 3-6 各种循环图

可以使用如下方法设计循环测试用例。

1. 简单循环

下列测试集用于简单循环,其中 n 是允许通过循环的最大次数。

1) 简单循环

(1) 零次循环:从循环入口到出口。

(2) 一次循环:检查循环初始值。

(3) 二次循环:两次通过循环。

(4) m 次循环:检查多次循环。

(5) 最大次数循环 n、比最大次数多一次 $n+1$、少一次的循环 $n-1$。

例 3-4 求最小值,如图 3-7 所示。

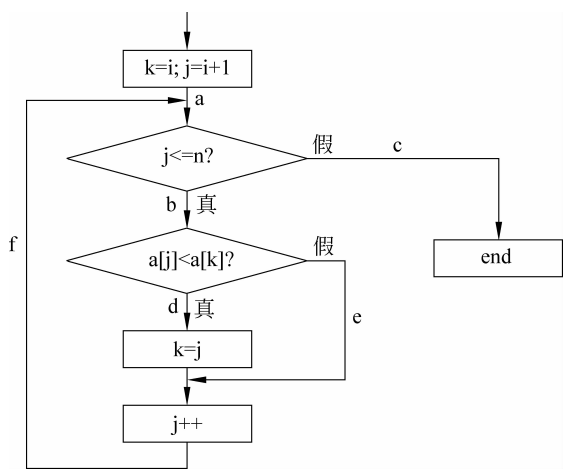

图 3-7 例 3-4 程序流程图

```
k=i;
for(j=i+1;j<=n;j++)
if(A[j]<A[k])k=j;
```

测试用例的选择，如表 3-5 所示。

表 3-5　简单循环测试用例

循环	i　n	A[i]	A[i+1]	A[i+2]	k	路径
0	1　1				i	ac
1	1　2	1 2	2 1		i i+1	abefc abdfc
2	1　3	1 2 3 3	2 3 2 1	3 1 1 2	i i+2 i+2 i+1	abefefc abefdfc abdfdfc abdfefc

说明：d 改 k 的值，e 不改 k 的值。

2. 嵌套循环

如果将简单循环的测试方法用于嵌套循环，可能的测试数目就会随嵌套层数成几何级增加，下面是一种减少测试数目的方法：

(1) 从最内层循环开始，将其他循环设置为最小值；

(2) 对最内层循环使用简单循环，而使外层循环的迭代参数(即循环计数)最小，并为范围外或排除的值增加其他测试；

(3) 逐步外推，对其外面一层循环进行测试，测试时保持其他外层循环为最小值，并使其他的嵌套循环变量为“典型”值；

(4) 重复上述过程，直到测试所有的循环。

3. 串接循环

如果串接循环的循环都彼此独立，可以使用嵌套的策略测试。但是如果两个循环串接起来，而第一个循环是第二个循环的初始值，则这两个循环并不是独立的。如果循环不独立，则推荐使用嵌套循环的方法进行测试。

4. 无结构循环

不能测试，重新设计出结构化的程序后再进行测试。

3.2.5　程序插装

在软件动态测试中，程序插桩(Program Instrumentation)是一种基本的测试手段。

方法简介：往被测程序中插入操作，实现测试目的的方法。

最简单的插桩：在程序中插入打印语句 printf(“…”)语句。这样就可以在运行程序以后，一方面检验测试结果数据，另一方面借助插入语句给出的信息了解程序的动态执行特性。

如果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,或是每个语句实际执行次数,最好的办法就是利用程序插桩技术。

例 3-5 求取个整数 X 和 Y 的最大公约数。

相应的程序如下:

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

可以根据程序绘制出其流程图,为了记录该程序中语句的执行次数,使用插桩技术插入如下语句:

```
C(i)=C(i)+1, i=1,2,…,6
```

插桩之后的流程图如图 3-8 所示。

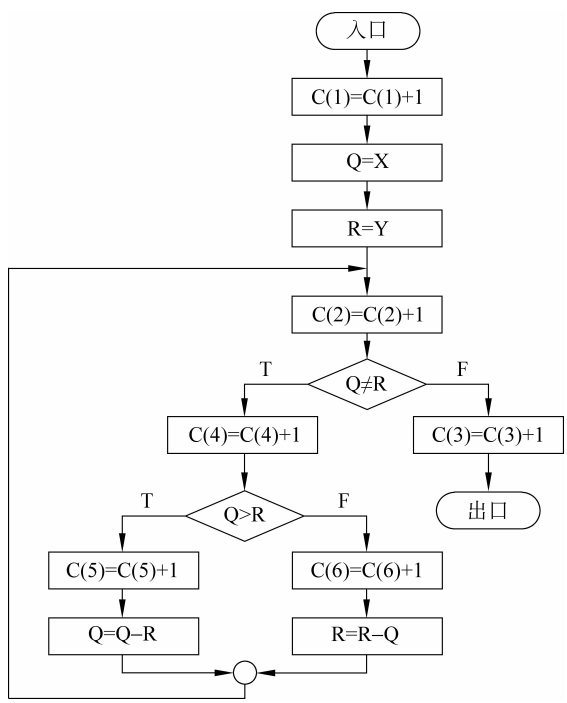

图 3-8 例 3-5 程序流程图

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来该程序的执行次数。如果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，就构成了完整的插桩程序。它就能记录并输出在各程序点上语句的实际执行次数。图 3-9 为插桩之后的程序，箭头所指为插入的语句。

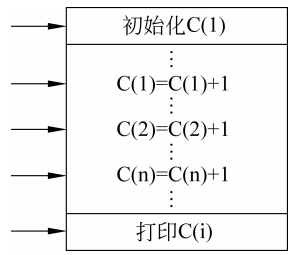

图 3-9　插桩程序示意图

设计插桩程序时需要考虑的问题包括：

(1) 需要探测哪些信息；

(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。

前两个问题需要结合具体的问题解决，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的探测点。

3.2.6　其他白盒测试方法

1. 数据流测试

数据流测试只关注变量接收值的点和使用(引用)这些值的点的结构性测试形式。可以用作路径测试的“真实性检查”。基于数据流覆盖的测试有助于填补边覆盖和路径覆盖之间的空缺。程序的数据流通常是在其控制流的基础上进行描述的，主要是指程序中变量的定义-使用关系，涉及如下几个概念：

(1) 如果程序中的变量出现在赋值号左边，称为对变量的定义(definition)；

(2) 如果变量出现在赋值号的右边，则称为对变量的计算使用(c-use：computation use)；

(3) 如果变量出现在谓词表达式中，则称为对变量的判定使用(p-use：predicate use)，并根据所在谓词表达式的值分为真(t)和假(f)；

(4) 对于程序的一条路径($i, n_i, n_2, \cdots, n_m, n_j$)，如果从 n_1 到 n_m 的节点中不包含对变量 x 的定义，则称为关于该变量的定义清除(definition-clear)的路径；

(5) 如果节点 n_d 包含对 x 的一个定义，而节点 $n_{c\text{-}use}$ 包含对 x 的一个计算使用，并且由 n_d 到 $n_{c\text{-}use}$ 是一条对 x 的定义清除的路径，那么 n_d 和 $n_{c\text{-}use}$ 就是 x 的一个定义-计算使用关系(definition-c-use)，表示为($n_d, n_{c\text{-}use}, x$)；

(6) 如果节点 n_d 包含对 x 的一个定义，而节点 $n_{p\text{-}use}$ 包含对 x 的一个判定使用，那么 n_d 和 $n_{p\text{-}use}$ 就是 x 的一个定义-判定使用关系(definition-p-use)，根据判定式的真假相应地表示为(n_d,($n_{p\text{-}use}$,t),x)和(n_d,($n_{p\text{-}use}$,f),x)；

(7) 经过定义-使用对(包括 c-use 和 p-use)的路径称作定义-使用路径(du-paths)。

给定一个测试用例集，假设 P 是运行这个测试用例集所经过的程序的完整路径集，如果 P 满足如下条件，则可定义相应的数据流测试覆盖标准如下。

- 定义覆盖：如果 P 覆盖了程序中所有的定义，也就是 P 包含从每一个定义到其某一相应使用(c-use 或 p-use)的定义-清除路径。
- 使用覆盖：如果 P 覆盖了程序中从每一个定义出发到所有与之相对应的使用(包括 c-use 和 p-use)的定义-清除路径。

- 定义-使用覆盖：如果 P 覆盖了与程序中每一个定义相对应的所有定义-使用路径，也就是说，如果一个定义-使用对之间存在多个路径，则这些路径都应被覆盖。

上述覆盖标准的强度是递增的，不过它们的强度都介于边覆盖和路径覆盖之间。数据流策略既考虑了数据运算方面，又考虑了程序的控制流方面，从而更有利于发现代码级的错误，不失为一种较好的测试方法，尤其是在路径覆盖无法实现的情况下。

2. 变异测试

变异测试(mutation testing)的提出始于 20 世纪 70 年代末期，是一种错误驱动测试，即针对某类特定程序错误而进行的测试，也是一种比较成熟的排错性测试方法(排错性测试方法的基本思想是通过检验测试数据集的排错能力来判断软件测试的充分性)。

假设 P 在测试集 T 上是正确的，可以找出 P 的变异体的某一集合：$M=\{M(P)\mid M(P)$ 是 P 的变异体$\}$，若变异体 M 中每一个元素在 T 上都存在错误，则可以认为源程序 P 的正确程度较高；否则若 M 中某些元素在 T 上不存在错误，则可能存在以下 3 种情况：

(1) 这些变异体与源程序 P 在功能上是等价的；

(2) 现有的测试数据不足以找出源程序 P 与其变异体的差别；

(3) 源程序 P 可能含有错误，而某些变异体却可能是正确的。

变异测试方法的理论基础来源于以下两个基本假设：

(1) 程序员的能力假设，即假设被测程序是由具有足够程序设计能力的程序员编写，因此所编写的程序是接近正确的；

(2) 组合效应假设，它假设简单的程序设计错误和复杂的程序设计错误之间具有组合效应，即一个测试数据如果能够发现简单的错误，那么也可以发现复杂的错误。正是这两个基本假设才确定了变异测试的基本特征，通过变异算子对程序作一个较小的语法变动来产生一个变异体。

本章小结

在白盒测试中，可以使用各种测试方法的综合测试。在测试中，应尽量先用工具进行静态结构分析。测试中可采取先静态后动态的方式：先进行静态结构分析、代码检查和静态质量度量，再进行覆盖率测试。利用静态分析的结果作为引导，通过代码检查和动态测试的方法对静态分析结果进行进一步的确认，使测试工作更为有效。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；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。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、静态质量度量；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。

习题

1. 选择题

(1) 以下关于白盒测试的叙述中,不正确的是(　　)。

A. 白盒测试仅与程序的内部结构有关,完全可以不考虑程序的功能要求

B. 逻辑覆盖法是一种常用的白盒测试方法

C. 程序中存在很多判定和条件,不可能实现 100%的条件覆盖

D. 测试基于代码,无法求额定设计正确与否

(2) 对于逻辑表达式((a&b)||c),需要(　　)个测试用例才能完成条件组合覆盖。

A. 2　　B. 3　　C. 4　　D. 5

(3) 逻辑覆盖法不包括(　　)。

A. 分支覆盖　　B. 语句覆盖

C. 需求覆盖　　D. 修正条件判定覆盖

(4) 如果某测试用例集实现了某软件的路径覆盖,那么它一定同时实现了该软件的(　　)。

A. 判定覆盖　　B. 条件覆盖

C. 判定/条件覆盖　　D. 组合覆盖

(5) 使用白盒测试方法时,确定测试数据的依据是指定的覆盖标准和(　　)。

A. 程序的注释　　B. 程序的内部逻辑

C. 用户使用说明书　　D. 程序的需求说明

2. 综合题

(1) 简述白盒测试用例的设计方法,并进行分析总结。

(2) 分析归纳逻辑覆盖的各种策略,并比较每种覆盖的特点,分析在怎样的情况下采用何种覆盖方式。

(3) 请按照各种覆盖方法为下述语句设计测试用例。

```
if(a>2 && b<3 &&(c>4 || d<5))
{
statement;
}
else
{
statement;
}
```

(4) 针对 test 函数按照基本路径测试方法设计测试用例。

```
int Test(int i_count, int i_flag)
  {
```

```
    int i_temp=0;
    while(i_count>0)
    {
      if(0==i_flag)
      {
        i_temp=i_count+100;
        break;
      }
      else
      {
        if(1==i_flag)
        {
          i_temp=i_temp+10;
        }
        else
        {
          i_temp=i_temp+20;
        }
      }
      i_count--;
    }
    return i_temp;
  }
```

(5) 用逻辑覆盖法对下面的 Java 代码段进行测试(画出程序流程图及控制流图,并写出每种覆盖准则对应的测试用例及执行路径)。

```
public char function(int x, int y){
    char t;
    if((x>=90)&&(y>=90)){
        t='A';
    } else {
            if((x+y)>=165){
            t='B';
            }else {
                      t='C';
                  }
          }
    return t;
}
```

(6) 下面是选择排序的程序,将数组中的数据按从小到大的顺序进行排序。

```
public void select_sort(int a[])
{
    int i,j,k,t,n;
```

```
    n=a.length;
    for(i=0;i<n-1;i++)
    {
        k=i;
        for(j=i+1;j<n;j++)
        {
            if(a[j]<a[k])
            {
                k=j;
            }
        }
        if(i!=k)
        {
            t=a[k];
            a[k]=a[i];
            a[i]=t;
        }
    }
}
```

要求：

① 计算此程序段的环形复杂度 $V(G)$；

② 用基路径测试法给出测试路径；

③ 为各测试路径设计测试用例。

CHAPTER

第4章 黑盒测试

【本章要点】

- 黑盒测试的概念；
- 使用黑盒测试的原则和策略；
- 等价类的划分；
- 边界值的划分；
- 因果图的构成；
- 决策表和错误推测法的应用。

【本章目标】

- 理解并掌握黑盒测试的概念；
- 掌握应用黑盒测试的原则；
- 理解并掌握如何进行等价类的划分；
- 理解并掌握如何进行边界值分析；
- 理解并掌握如何使用因果图设计测试用例；
- 理解如何使用决策表进行测试用例的设计；
- 理解各种黑盒测试技术的特点及其适用的情况。

黑盒测试也称作功能测试和行为测试，主要是根据功能需求来测试程序是否按照预期工作。黑盒测试的目的是尽量发现代码所表现的外部行为的错误，主要有以下几类：

（1）功能不正确或不完整；

（2）接口错误；

（3）接口所使用的数据结构错误；

（4）行为或性能错误；

（5）初始化和终止错误。

通过这些测试，就可以确定软件所实现的功能是否符合规格说明。但是，也可以用来证明软件代码中是否有错误以及缺陷存在，这一点也是很重要的。

最好不要让编写被测模块代码的程序员以及了解程序代码结构的人员来做黑盒测试计划或运行黑盒测试。因为程序员容易按照编写代码的思路来做测试，而我们做测试的目的是为了能够满足用户的需求。总之，大部分组织都是由独立的测试团队来完成黑盒