白盒测试方法

白盒测试方法

在软件测试中，白盒测试是根据被测程序的内部结构设计测试用例的一种测试方法，具体的白盒测试方法有程序控制流分析、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。 本文主要介绍程序结构分析、逻辑覆盖和程序插装，对其他的白盒测试方法只作简单介绍。

第一章 程序结构分析

程序的结构形式是白盒测试的主要依据。本章将从控制流分析、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

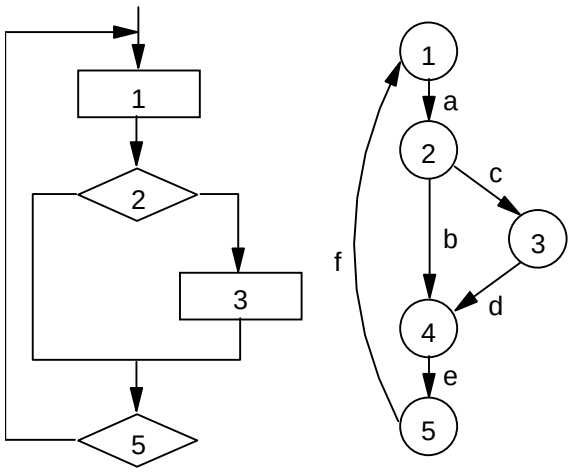
程序流程图（flowchart）又称框图，也许是人们最熟悉，也是最容易接受的一种程序控制结构的图形表示了。在这种图上的框内常常标明了处理要求或条件，这些在做路径分析时是不重要的。为了更加突出控制流的结构，需要对程序流程图做些简化。在图1中给出了简化的例子。其中（a）是一个含有两出口判断和循环的程序流程图，我们把它简化成（b）的形式，称这种简化了的流程图为控制流图（Control-flow graph）。

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，是由该控制图中含有5个节点决定的。矩阵中6个元素a、b、c、d、e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。



(a) 程序流程图 (b) 控制流图

图1 程序流程图和控制流图

	a			
		c	b	
			d	
				e
f				

图2 控制流图矩阵

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。目前对这四种情况的检测主要通过编译器和程序分析工具来实现。

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

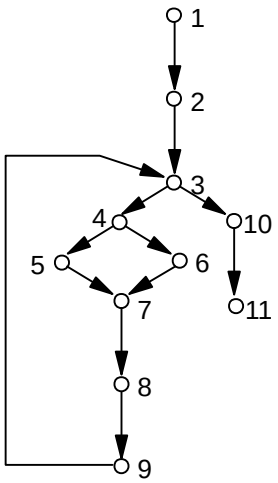
只引用了Y和Z。输入语句

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PORTRAN中，循环语句PO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3 控制流图及其定义和引用的变量

图3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）。

目前通过编译器或程序分析工具通过数据流分析可以查找出对未定义变量的使用和未曾使用的定义。

3、数据流分析应用的其它方面

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更，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，因为它会提醒他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4是这一算法的程序。图5是三个关系的表。其中第一个表（图5(a)）给出每一语句执行时所用到其输入值的变量。例如，从图4的算法很明显地看出，M的输入值在语句2中得到直接使用，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。

```
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4 整除算法

	M	N
1		
2	√	
3	√	√
4	√	√
5	√	√

(a)

	Q	R
1	√	
2	√	√
3	√	√
4	√	
5	√	√

(b)

	Q	R
M	√	√
N	√	√

(c)

图5 整除算法中输入值、语句与输出值的关系

第二个关系（图5(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。

第二章 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。

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：

```
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6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。

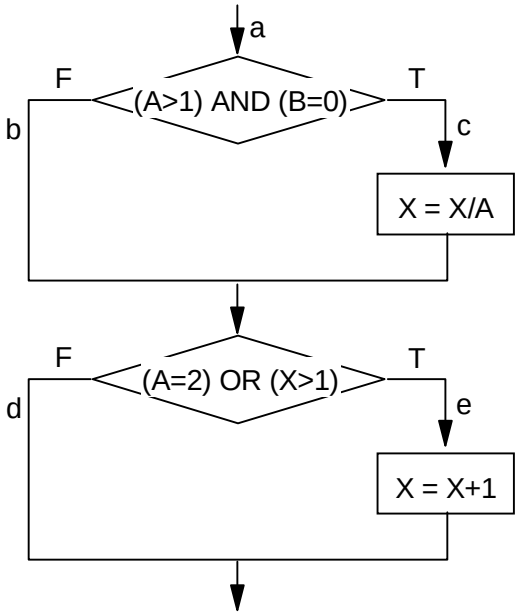


图6 被测程序段流程图

一、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。这时所谓“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
X = 3
} ……CASE1

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
X = 3
} ……CASE2

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推积，语句之间的确存在着许多有机的联系。

二、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，运行被侧程序，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 } ……CASE1
X = 3

A = 1
B = 0 } ……CASE3
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 } ……CASE4
X = 3

A = 2
B = 1 } ……CASE5
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。

三、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，记为F_1

B = 0，取真值，记为T_2

B = 0，取假值，即B≠0，记为F_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≠2，记为F_3

X＞1，取真值，记为T_4

X＞1，取假值，即X≤1，记为F_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	ABX	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	F_1， T_2， F_3， F_4
CASE 8	2 1 1	a b e	T_1， F_2， T_3， F_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	a b e	b e	F_1， T_2， F_3， T_4
CASE 8	2 1 1	a b e	b e	T_1， F_2， T_3， F_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。

四、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 ≠ 0 记为 T_1，F_2

③ A ≤1，B ＝ 0 记为 F_1，T_2

④ A ≤1，B ≠ 0 记为 F_1，F_2

⑤ A ＝2，X〉 1 记为 T_3，T_4

⑥ A ＝2，X ≤ 1 记为 T_3，F_4

⑦ A ≠2，X〉 1 记为 F_3，T_4

⑧ A ≠2，X ≤ 1 记为 F_3，F_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	ABX	覆盖组合号	所走路径	覆盖条件
CASE 1	2 0 3	① ⑤	a c e	T_1，T_2，T_3，T_4
CASE 8	2 1 1	② ⑥	a b c	T_1，F_2，T_3，F_4
CASE 9	1 0 3	③ ⑦	a b e	F_1，T_2，F_3，T_4
CASE 10	1 1 1	④ ⑧	a b d	F_1，F_2，F_3，F_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。

五、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

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	ABX	覆盖路径
CASE 1	2 0 3	a c e (L_1)
CASE 7	1 0 1	a b d (L_2)
CASE 8	2 1 1	a b e (L_3)
CASE 11	3 0 1	a c d (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，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，软件测试是有局限性的。

六、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：

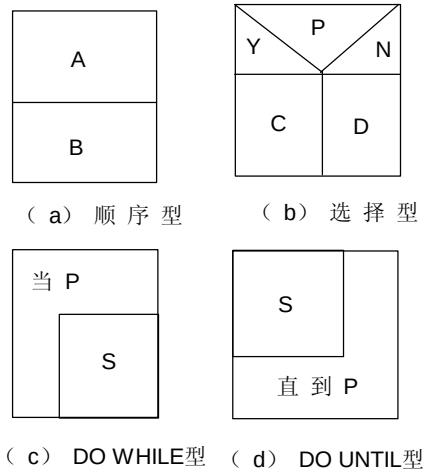
顺序型——构成串行操作；

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7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7（c）和图7（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。



（a） 顺 序 型　（b） 选 择 型

（c） DO WHILE型　（d） DO UNTIL型

图 7 N－ S图 表 示 的 基 本 控 制 结 构

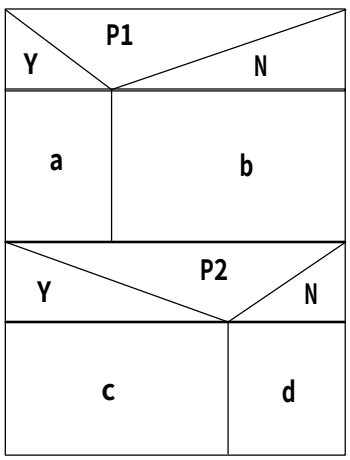


图8 两个串行的分支结构的N-S图

例如，图8表达了两个顺序执行的分支结构。两个分支谓词P1和P2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2×2 = 4。并且，这里的2是由于两个并列的操作，1 + 1 = 2 而得到的。

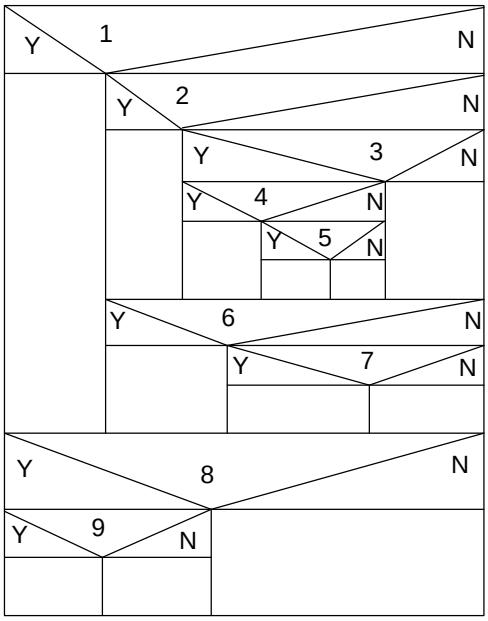


图 9 计 算 最 少 测 试 用 例 数 实 例

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9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P1和P2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10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5×3 = 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6×3 = 48。

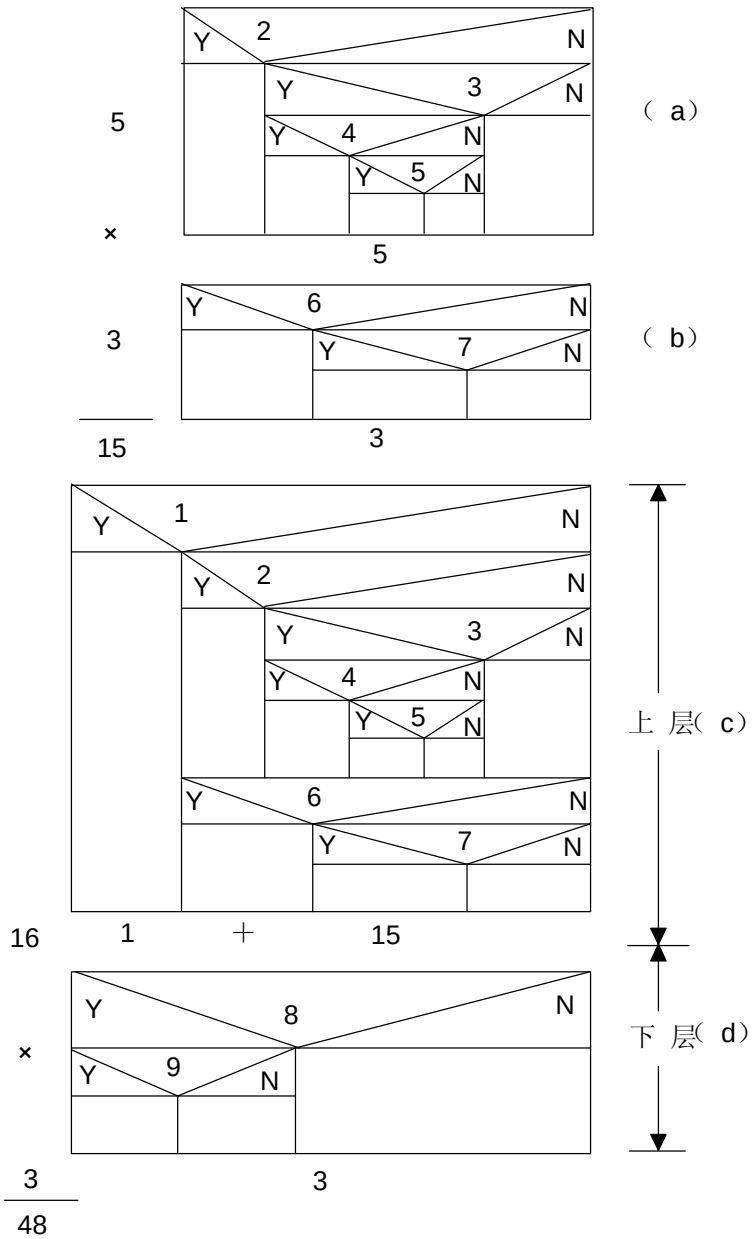


图 10 最 少 测 试 用 例 数 计 算

七、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 rel1 C（rel1可以是>或是<，A是变量，C是常量）型的分支谓词，当rel1为<时，应适当地选择A的值，使：

$$A = C - M$$

（M是距C最小的容器容许正数，若A和C均为整型时，M = 1）。同样，当rel1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合不一致。

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，应选择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。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。

第三章 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，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11给出了这一程序的流程图。图中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：

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

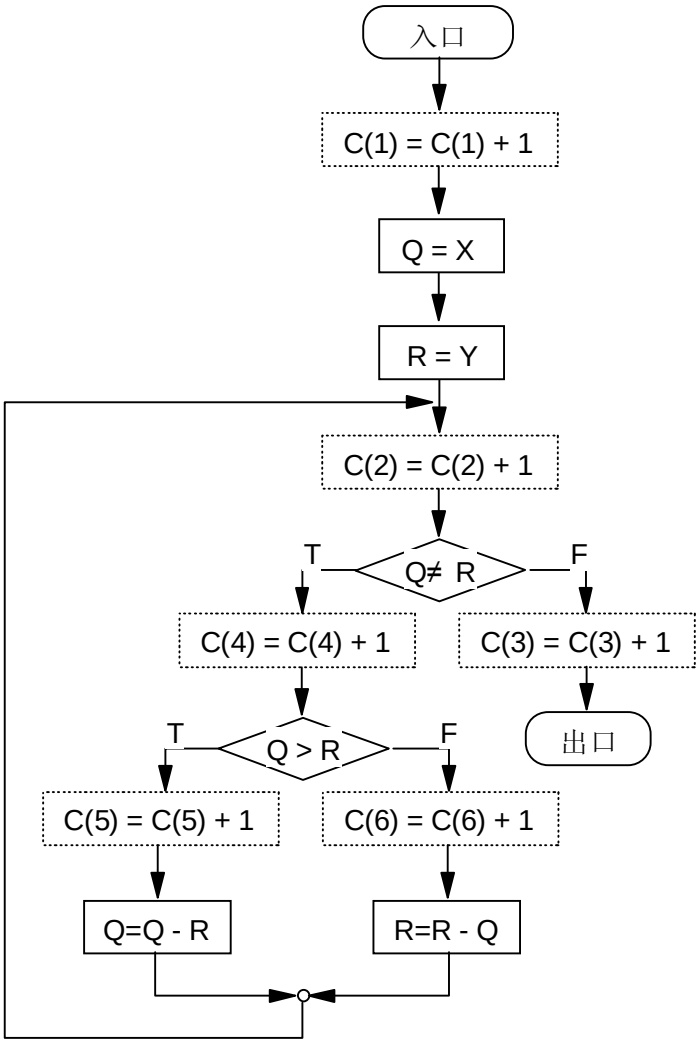


图11 插装后的求最大公约数程序流程图

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12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。

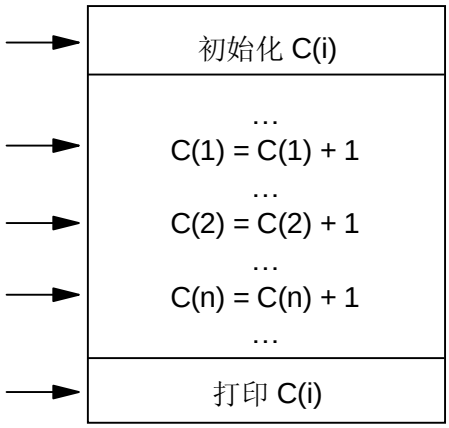


图12 插装程序中插入的语句

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。又如本文第二章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11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。为了在程序中设计最少的计数语句，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：

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。

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M（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13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$K \geq 0$；

② A = DEN * Q;

③ B = DEN * W/2;

④ $NUM/DEN - W < Q \leq NUM/DEN$。

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：

$$NUM/DEN - E < Q \leq NUM/DEN$$

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13 计算两非负数之商的迭代程序

图14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。

```
  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14 插入断言后的迭代程序

首先来检验在初始化以后循环后的断言。

① 由于 K = 0，所以$W = 2^{-K} = 1$是初值。

② 由于Q = 0，A的初始 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 ≤ 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ir-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：

$$A^{\hat{\imath}} = A$$

$$B^{\hat{\imath}} = B/2$$

$$W^{\hat{\imath}} = W/2$$

$$Q^{\hat{\imath}} = Q$$

$$K^{\hat{\imath}} = K + 1$$

再来检验4个断言：

① $W^{\hat{\imath}} = W/2 = 1/2 \,\&\&\, K^{\hat{\imath}}$

② $A^{\hat{\imath}} = A = DEN \,\&\, Q = DEN \,\&\, Q^{\hat{\imath}}$

③ $B^{\hat{\imath}} = B/2 = DEN \,\&\, W/4 = DEN \,\&\, W^{\hat{\imath}}/2$

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$NUM/DEN - W^{\hat{\imath}} < Q^{\hat{\imath}}$，且$Q^{\hat{\imath}}$ [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≥0），并作同前的变换，得到NUM/DEN - W″/2＜Q″ + W″/2或：

NUM/DEN - W″ ＜Q″，并且Q″≤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 ≤NUM/DEN必定为真，也就必定满足输出断言。

第四章 其他白盒测试方法简介

一、域测试

域测试（Domain Testing）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，他对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。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。在考察被测试程序的结构以后，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。

域测试有两个致命的弱点，一是为进行域测试对程序提出的限制过多，二是当程序存在很多路径时，所需的测试点也就很多。

二、符号测试

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。

符号测试可以看作是程序测试和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。

目前符号测试存在一些未得到圆满解决的问题，分别是：

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

符号测试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。

三、Z路径覆盖

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。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，达到天文数字，以至实现路径覆盖不可能做到。

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15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。

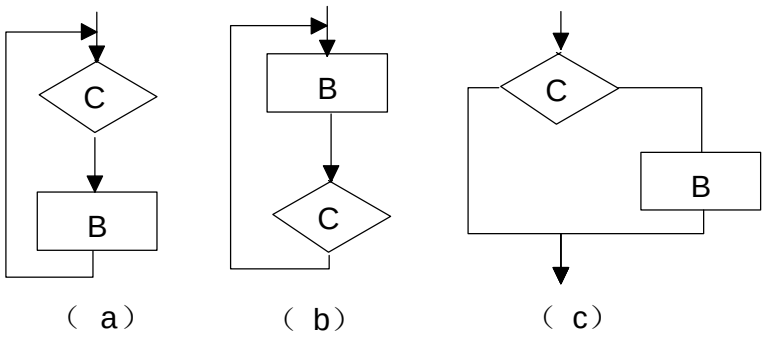


图 15 循 环 结 构 简 化 成 选 择 结 构

对于程序中的所有路径可以用路径树来表示，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。

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。

四、程序变异

程序变异方法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，一些变异测试工具被开发出来。关于程序变异测试方法，请参见清华大学郑人杰教授编写的《软件测试技术》一书。