

7.6 白盒测试技术

设计测试方案是测试阶段的关键技术问题。所谓测试方案包括具体的测试目的(例如,预定要测试的具体功能),应该输入的测试数据和预期的结果。通常又把测试数据和预期的输出结果称为测试用例。其中最困难的问题是设计测试用的输入数据。

不同的测试数据发现程序错误的能力差别很大,为了提高测试效率降低测试成本,应该选用高效的测试数据。因为不可能进行穷尽的测试,所以选用少量“最有效的”测试数据,做到尽可能完备的测试就更重要了。

设计测试方案的基本目标是,确定一组最可能发现某个错误或某类错误的测试数据。已经研究出许多设计测试数据的技术,这些技术各有优缺点,没有哪一种是最好的,更没有哪一种可以代替其余所有技术;同一种技术在不同的应用场合效果可能相差很大,因此,通常需要联合使用多种设计测试数据的技术。

本节讲述在用白盒方法测试软件时设计测试数据的典型技术,下一节讲述在用黑盒方法测试软件时设计测试数据的典型技术。

7.6.1 逻辑覆盖

有选择地执行程序中某些最有代表性的通路是对穷尽测试的唯一可行的替代办法。所谓逻辑覆盖是对一系列测试过程的总称,这组测试过程逐渐进行越来越完整的通路测试。测试数据执行(或叫覆盖)程序逻辑的程度可以划分成哪些不同的等级呢?从覆盖源程序语句的详尽程度分析,大致有以下一些不同的覆盖标准。

1. 语句覆盖

为了暴露程序中的错误,至少每个语句应该执行一次。语句覆盖的含义是,选择足够的测试数据,使被测程序中每个语句至少执行一次。例如,图 7.5 所示的程序流程图描绘了一个被测模块的处理算法。

为了使每个语句都执行一次,程序的执行路径应该是 `sacbed`,为此只需要输入下面的测试数据(实际上 X 可以是任意实数):

$$A = 2, B = 0, X = 4$$

语句覆盖对程序的逻辑覆盖很少,在上面例子中两个判定条件都只测试了条件为真的情况,如果条件为假时处理有错误,显然不能发现。此外,语句覆盖只关心判定表达式的值,而没有分别测试判定表达式中每个条件取不同值时的情况。在上面的例子中,为了执行 `sacbed` 路径,以测试每个语句,只需两个判定表达式($A > 1$) AND ($B = 0$) 和 ($A = 2$) OR ($X > 1$) 都取真值,因此使用上述一组测试数据就够了。但是,如果程序中把第一个判定表达式中的逻辑运算符 AND 错写成 OR,或把第二个判定表达式中的条件 $X > 1$ 误写成 $X < 1$,使用上面的测试数据并不能查出这些错误。

综上所述,可以看出语句覆盖是很弱的逻辑覆盖标准,为了更充分地测试程序,可以采用下述的逻辑覆盖标准。

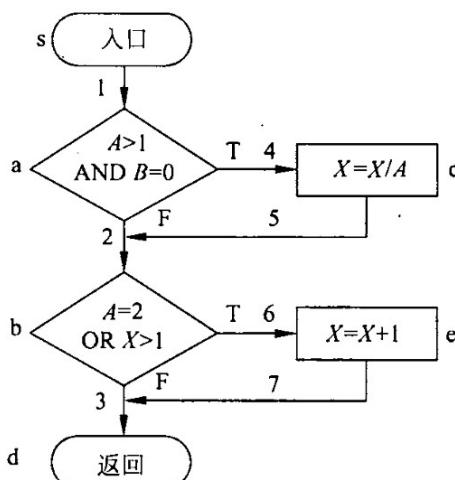


图 7.5 被测试模块的流程图

2. 判定覆盖

判定覆盖又叫分支覆盖,它的含义是,不仅每个语句必须至少执行一次,而且每个判定的每种可能的结果都应该至少执行一次,也就是每个判定的每个分支都至少执行一次。

对于上述例子来说,能够分别覆盖路径 sacbed 和 sabd 的两组测试数据,或者可以分别覆盖路径 sacbd 和 sabed 的两组测试数据,都满足判定覆盖标准。例如,用下面两组测试数据就可做到判定覆盖:

- I. $A=3, B=0, X=3$ (覆盖 sacbd)
- II. $A=2, B=1, X=1$ (覆盖 sabed)

判定覆盖比语句覆盖强,但是对程序逻辑的覆盖程度仍然不高,例如,上面的测试数据只覆盖了程序全部路径的一半。

3. 条件覆盖

条件覆盖的含义是,不仅每个语句至少执行一次,而且使判定表达式中的每个条件都取到各种可能的结果。

图 7.5 的例子中共有两个判定表达式,每个表达式中有两个条件,为了做到条件覆盖,应该选取测试数据使得在 a 点有下述各种结果出现:

$$A > 1, A \leq 1, B = 0, B \neq 0$$

在 b 点有下述各种结果出现:

$$A = 2, A \neq 2, X > 1, X \leq 1$$

只需要使用下面两组测试数据就可以达到上述覆盖标准:

- I. $A=2, B=0, X=4$

(满足 $A > 1, B = 0, A = 2$ 和 $X > 1$ 的条件,执行路径 sacbed)

- II. $A=1, B=1, X=1$

(满足 $A \leq 1, B \neq 0, A \neq 2$ 和 $X \leq 1$ 的条件,执行路径 sabd)



条件覆盖通常比判定覆盖强,因为它使判定表达式中每个条件都取到了两个不同的结果,判定覆盖却只关心整个判定表达式的值。例如,上面两组测试数据也同时满足判定覆盖标准。但是,也可能有相反的情况:虽然每个条件都取到了两个不同的结果,判定表达式却始终只取一个值。例如,如果使用下面两组测试数据,则只满足条件覆盖标准并不满足判定覆盖标准(第二个判定表达式的值总为真):

I. $A=2, B=0, X=1$

(满足 $A>1, B=0, A=2$ 和 $X \leq 1$ 的条件,执行路径 sacbed)

II. $A=1, B=1, X=2$

(满足 $A \leq 1, B \neq 0, A \neq 2$ 和 $X > 1$ 的条件,执行路径 sabed)

4. 判定/条件覆盖

既然判定覆盖不一定包含条件覆盖,条件覆盖也不一定包含判定覆盖,自然会提出一种能同时满足这两种覆盖标准的逻辑覆盖,这就是判定/条件覆盖。它的含义是,选取足够的测试数据,使得判定表达式中的每个条件都取到各种可能的值,而且每个判定表达式也都取到各种可能的结果。

对于图 7.5 的例子而言,下述两组测试数据满足判定/条件覆盖标准:

I. $A=2, B=0, X=4$

II. $A=1, B=1, X=1$

但是,这两组测试数据也就是为了满足条件覆盖标准最初选取的两组数据,因此,有时判定/条件覆盖也并不比条件覆盖更强。

5. 条件组合覆盖

条件组合覆盖是更强的逻辑覆盖标准,它要求选取足够的测试数据,使得每个判定表达式中条件的各种可能组合都至少出现一次。

对于图 7.5 的例子,共有 8 种可能的条件组合,它们分别是:

(1) $A>1, B=0$

(2) $A>1, B \neq 0$

(3) $A \leq 1, B=0$

(4) $A \leq 1, B \neq 0$

(5) $A=2, X>1$

(6) $A=2, X \leq 1$

(7) $A \neq 2, X>1$

(8) $A \neq 2, X \leq 1$

和其他逻辑覆盖标准中的测试数据一样,条件组合(5)~(8)中的 X 值是指在程序流程图第二个判定框(b 点)的 X 值。

下面的 4 组测试数据可以使上面列出的 8 种条件组合每种至少出现一次:

I. $A=2, B=0, X=4$

(针对(1),(5)两种组合,执行路径 sacbed)

II. $A=2, B=1, X=1$

(针对(2),(6)两种组合,执行路径 sabed)

III. $A=1, B=0, X=2$

(针对(3),(7)两种组合,执行路径 sabed)

IV. $A=1, B=1, X=1$

(针对(4),(8)两种组合,执行路径 sabd)

显然,满足条件组合覆盖标准的测试数据,也一定满足判定覆盖、条件覆盖和判定/条件覆盖标准。因此,条件组合覆盖是前述几种覆盖标准中最强的。但是,满足条件组合覆盖标准的测试数据并不一定能使程序中的每条路径都执行到,例如,上述 4 组测试数据都没有测试到路径 sacbd。

以上根据测试数据对源程序语句检测的详尽程度,简单讨论了几种逻辑覆盖标准。在上面的分析过程中常常谈到测试数据执行的程序路径,显然,测试数据可以检测的程序路径的多少,也反映了对程序测试的详尽程度。从对程序路径的覆盖程度分析,能够提出下述一些主要的逻辑覆盖标准。

6. 点覆盖

图论中点覆盖的概念定义如下:如果连通图 G 的子图 G' 是连通的,而且包含 G 的所有结点,则称 G' 是 G 的点覆盖。

在第 6.5 节中已经讲述了从程序流程图导出流图的方法。在正常情况下流图是连通的有向图。满足点覆盖标准要求选取足够多的测试数据,使得程序执行路径至少经过流图的每个结点一次,由于流图的每个结点与一条或多条语句相对应,显然,点覆盖标准和语句覆盖标准是相同的。

7. 边覆盖

图论中边覆盖的定义是:如果连通图 G 的子图 G'' 是连通的,而且包含 G 的所有边,则称 G'' 是 G 的边覆盖。为了满足边覆盖的测试标准,要求选取足够多测试数据,使得程序执行路径至少经过流图中每条边一次。通常边覆盖和判定覆盖是一致的。

8. 路径覆盖

路径覆盖的含义是,选取足够多测试数据,使程序的每条可能路径都至少执行一次(如果程序图中有环,则要求每个环至少经过一次)。

作为一个练习,读者可以自己设计用路径覆盖标准测试图 7.5 所示模块的测试数据。

7.6.2 控制结构测试

现有的很多种白盒测试技术,是根据程序的控制结构设计测试数据的技术,下面介绍几种常用的控制结构测试技术。

1. 基本路径测试

基本路径测试是 Tom McCabe 提出的一种白盒测试技术。使用这种技术设计测试用例时,首先计算程序的环形复杂度,并用该复杂度为指南定义执行路径的基本集合,从该基本集合导出的测试用例可以保证程序中的每条语句至少执行一次,而且每个条件在执行时都将分别取真、假两种值。

使用基本路径测试技术设计测试用例的步骤如下。

第一步,根据过程设计结果画出相应的流图。

例如,为了用基本路径测试技术测试下列的用 PDL 描述的求平均值过程,首先画出图 7.6 所示的流图。注意,为了正确地画出流图,这里把被映射为流图结点的 PDL 语句编了序号。

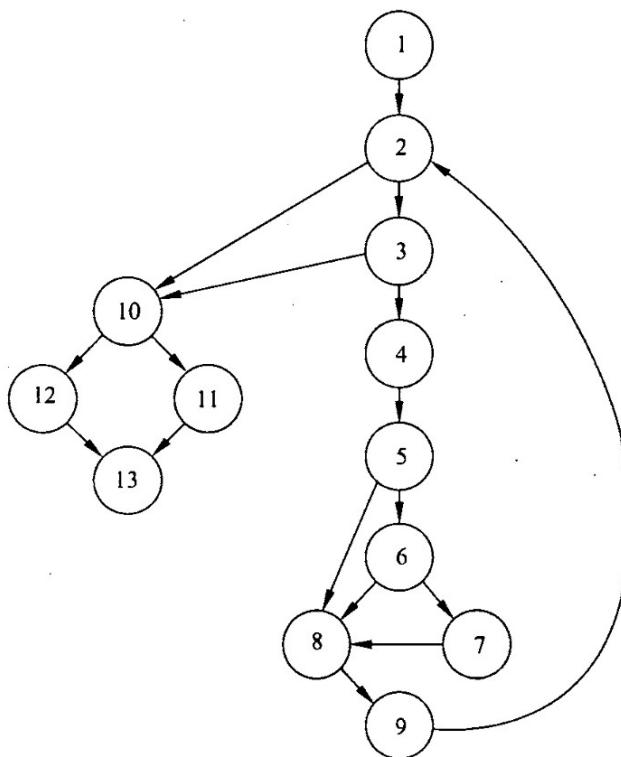


图 7.6 求平均值过程的流图

```

PROCEDURE average;
/* 这个过程计算不超过 100 个在规定值域内的有效数字的平均值;
同时计算有效数字的总和及个数。 */
INTERFACE RETURNS average, total, input, total, valid;
INTERFACE ACCEPTS value, minimum, maximum;

TYPE value[1..100] IS SCALAR ARRAY;
TYPE average, total, input, total, valid,
minimum, maximum, sum IS SCALAR;
  
```

```
TYPE i IS INTEGER;

1:    i=1;
      total. input=total. valid=0;
      sum=0;
2:    DO WHILE value[i] <> -999
3:        AND total. input<100
4:        increment total. input by1;
5:        IF value[i]>=minimum
6:            AND value[i]<=maximum
7:            THEN increment total. valid by 1;
               sum=sum+value[i];
8:        ENDIF
               increment i by 1;
9:    ENDDO
10:   IF total. valid>0
11:   THEN average=sum/total. valid;
12:   ELSE average=-999;
13:   ENDIF
END average
```

第二步,计算流图的环形复杂度。

环形复杂度定量度量程序的逻辑复杂性。有了描绘程序控制流的流图之后,可以用第 6.5.1 小节讲述的 3 种方法之一计算环形复杂度。经计算,图 7.6 所示流图的环形复杂度为 6。

第三步,确定线性独立路径的基本集合。

所谓独立路径是指至少引入程序的一个新处理语句集合或一个新条件的路径,用流图术语描述,独立路径至少包含一条在定义该路径之前不曾用过的边。

使用基本路径测试法设计测试用例时,程序的环形复杂度决定了程序中独立路径的数量,而且这个数是确保程序中所有语句至少被执行一次所需的测试数量的上界。

对于图 7.6 所描述的求平均值过程来说,由于环形复杂度为 6,因此共有 6 条独立路径。例如,下面列出了 6 条独立路径。

路径 1: 1—2—10—11—13

路径 2: 1—2—10—12—13

路径 3: 1—2—3—10—11—13

路径 4: 1—2—3—4—5—8—9—2—...

路径 5: 1—2—3—4—5—6—8—9—2—...

路径 6: 1—2—3—4—5—6—7—8—9—2—...

路径 4、5、6 后面的省略号(…表示,可以后接通过控制结构其余部分的任意路径(例如,10—11—13)。

通常在设计测试用例时,识别出判定结点是很有必要的。本例中结点 2、3、5、6 和 10

是判定结点。

第四步,设计可强制执行基本集合中每条路径的测试用例。

应该选取测试数据使得在测试每条路径时都适当地设置好了各个判定结点的条件。例如,可以测试上一步得出的基本集合的测试用例如下。

路径 1 的测试用例:

value[k]=有效输入值,其中 $k < i$ (i 的定义在下面)

value[i]=-999,其中 $2 \leq i \leq 100$

预期结果: 基于 k 的正确平均值和总数

注意,路径 1 无法独立测试,必须作为路径 4 或 5 或 6 的一部分来测试。

路径 2 的测试用例:

value[1]=-999

预期结果: average=-999, 其他都保持初始值

路径 3 的测试用例:

试图处理 101 个或更多个值

前 100 个数值应该是有效输入值

预期结果: 前 100 个数的平均值,总数为 100

注意,路径 3 也无法独立测试,必须作为路径 4 或 5 或 6 的一部分来测试。

路径 4 的测试用例:

value[i]=有效输入值,其中 $i < 100$

value[k] < minimum, 其中 $k < i$

预期结果: 基于 k 的正确平均值和总数

路径 5 的测试用例:

value[i]=有效输入值,其中 $i < 100$

value[k] > maximum, 其中 $k < i$

预期结果: 基于 k 的正确平均值和总数

路径 6 的测试用例:

value[i]=有效输入值,其中 $i < 100$

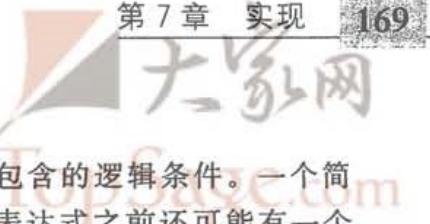
预期结果: 正确的平均值和总数

在测试过程中,执行每个测试用例并把实际输出结果与预期结果相比较。一旦执行完所有测试用例,就可以确保程序中所有语句都至少被执行了一次,而且每个条件都分别取过 true 值和 false 值。

应该注意,某些独立路径(例如,本例中的路径 1 和路径 3)不能以独立的方式测试,也就是说,程序的正常流程不能形成独立执行该路径所需要的数据组合(例如,为了执行本例中的路径 1,需要满足条件 total.valid>0)。在这种情况下,这些路径必须作为另一个路径的一部分来测试。

2. 条件测试

尽管基本路径测试技术简单而且高效,但是仅有这种技术还不够,还需要使用其他控



制结构测试技术,才能进一步提高白盒测试的质量。

用条件测试技术设计出的测试用例,能够检查程序模块中包含的逻辑条件。一个简单条件是一个布尔变量或一个关系表达式,在布尔变量或关系表达式之前还可能有一个 NOT(\neg)算符。关系表达式的形式如下:

$$E_1 < \text{关系算符} > E_2$$

其中, E_1 和 E_2 是算术表达式,而 $<\text{关系算符}>$ 是下列算符之一: $<$, \leq , $=$, \neq , $>$ 或 \geq 。复合条件由两个或多个简单条件、布尔算符和括弧组成。布尔算符有 OR(|), AND(&) 和 NOT(\neg)。不包含关系表达式的条件称为布尔表达式。

因此,条件成分的类型包括布尔算符、布尔变量、布尔括弧(括住简单条件或复合条件)、关系算符及算术表达式。

如果条件不正确,则至少条件的一个成分不正确。因此,条件错误的类型如下:

- 布尔算符错(布尔算符不正确,遗漏布尔算符或有多余的布尔算符)
- 布尔变量错
- 布尔括弧错
- 关系算符错
- 算术表达式错

条件测试方法着重测试程序中的每个条件。本节下面将讲述的条件测试策略有两个优点:①容易度量条件的测试覆盖率;②程序内条件的测试覆盖率可指导附加测试的设计。

条件测试的目的不仅是检测程序条件中的错误,而且是检测程序中的其他错误。如果程序 P 的测试集能有效地检测 P 中条件的错误,则它很可能也可以有效地检测 P 中的其他错误。此外,如果一个测试策略对检测条件错误是有效的,则很可能该策略对检测程序的其他错误也是有效的。

人们已经提出了许多条件测试策略。分支测试可能是最简单的条件测试策略:对于复合条件 C 来说,C 的真分支和假分支以及 C 中的每个简单条件,都应该至少执行一次。

域测试要求对一个关系表达式执行 3 个或 4 个测试。对于形式为

$$E_1 < \text{关系算符} > E_2$$

的关系表达式来说,需要 3 个测试分别使 E_1 的值大于、等于或小于 E_2 的值。如果 $<\text{关系算符}>$ 错误而 E_1 和 E_2 正确,则这 3 个测试能够发现关系算符的错误。为了发现 E_1 和 E_2 中的错误,让 E_1 值大于或小于 E_2 值的测试数据应该使这两个值之间的差别尽可能小。

包含 n 个变量的布尔表达式需要 2^n 个(每个变量分别取真或假这两个可能值的组合数)测试。这个策略可以发现布尔算符、变量和括弧的错误,但是,该策略仅在 n 很小时才是实用的。

在上述种种条件测试技术的基础上,K. C. Tai 提出了一种被称为 BRO(branch and relational operator)测试的条件测试策略。如果在条件中所有布尔变量和关系算符都只出现一次而且没有公共变量,则 BRO 测试保证能发现该条件中的分支错和关系算符错。

BRO 测试利用条件 C 的条件约束来设计测试用例。包含 n 个简单条件的条件 C 的

条件约束定义为 (D_1, D_2, \dots, D_n) , 其中 $D_i (0 < i \leq n)$ 表示条件 C 中第 i 个简单条件的输出约束。如果在条件 C 的一次执行过程中, C 中每个简单条件的输出都满足 D 中对应的约束, 则称 C 的这次执行覆盖了 C 的条件约束 D 。

对于布尔变量 B 来说, B 的输出约束指出, B 必须是真(t)或假(f)。类似地, 对于关系表达式来说, 用符号 $>$, $=$ 和 $<$ 指定表达式的输出约束。

作为第一个例子, 考虑下列条件

$$C_1 : B_1 \& B_2$$

其中, B_1 和 B_2 是布尔变量。 C_1 的条件约束形式为 (D_1, D_2) , 其中 D_1 和 D_2 中的每一个都是 t 或 f 。值(t, f)是 C_1 的一个条件约束, 并由使 B_1 值为真 B_2 值为假的测试所覆盖。BRO 测试策略要求, 约束集 $\{(t, t), (f, t), (t, f)\}$ 被 C_1 的执行所覆盖。如果 C_1 因布尔算符错误而不正确, 则至少上述约束集中的一个约束将迫使 C_1 失败。

作为第二个例子, 考虑下列条件

$$C_2 : B_1 \& (E_3 = E_4)$$

其中, B_1 是布尔变量, E_3 和 E_4 是算术表达式。 C_2 的条件约束形式为 (D_1, D_2) , 其中 D_1 是 t 或 f , D_2 是 $>$, $=$ 或 $<$ 。除了 C_2 的第二个简单条件是关系表达式之外, C_2 和 C_1 相同, 因此, 可以通过修改 C_1 的约束集 $\{(t, t), (f, t), (t, f)\}$ 得出 C_2 的约束集。注意, 对于 $(E_3 = E_4)$ 来说, t 意味 $=$, 而 f 意味着 $<$ 或 $>$, 因此, 分别用 $(t, =)$ 和 $(f, =)$ 替换 (t, t) 和 (f, t) , 并用 $(t, <)$ 和 $(t, >)$ 替换 (t, f) , 就得到 C_2 的约束集 $\{(t, =), (f, =), (t, <), (t, >)\}$ 。覆盖上述条件约束集的测试, 保证可以发现 C_2 中布尔算符和关系算符的错误。

作为第三个例子, 考虑下列条件

$$C_3 : (E_1 > E_2) \& (E_3 = E_4)$$

其中, E_1 , E_2 , E_3 和 E_4 是算术表达式。 C_3 的条件约束形式为 (D_1, D_2) , 而 D_1 和 D_2 的每一个都是 $>$, $=$ 或 $<$ 。除了 C_3 的第一个简单条件是关系表达式之外, C_3 和 C_2 相同, 因此, 可以通过修改 C_2 的约束集得到 C_3 的约束集, 结果为:

$$\{(>, =), (=, =), (<, =), (>, <), (>, >)\}$$

覆盖上述条件约束集的测试, 保证可以发现 C_3 中关系算符的错误。

3. 循环测试

循环是绝大多数软件算法的基础, 但是, 在测试软件时却往往未对循环结构进行足够的测试。

循环测试是一种白盒测试技术, 它专注于测试循环结构的有效性。在结构化的程序中通常只有 3 种循环, 即简单循环、串接循环和嵌套循环, 如图 7.7 所示。下面分别讨论这 3 种循环的测试方法。

(1) 简单循环。应该使用下列测试集来测试简单循环, 其中 n 是允许通过循环的最大次数。

- 跳过循环。
- 只通过循环一次。
- 通过循环两次。

- 通过循环 m 次, 其中 $m < n - 1$ 。
- 通过循环 $n - 1, n, n + 1$ 次。

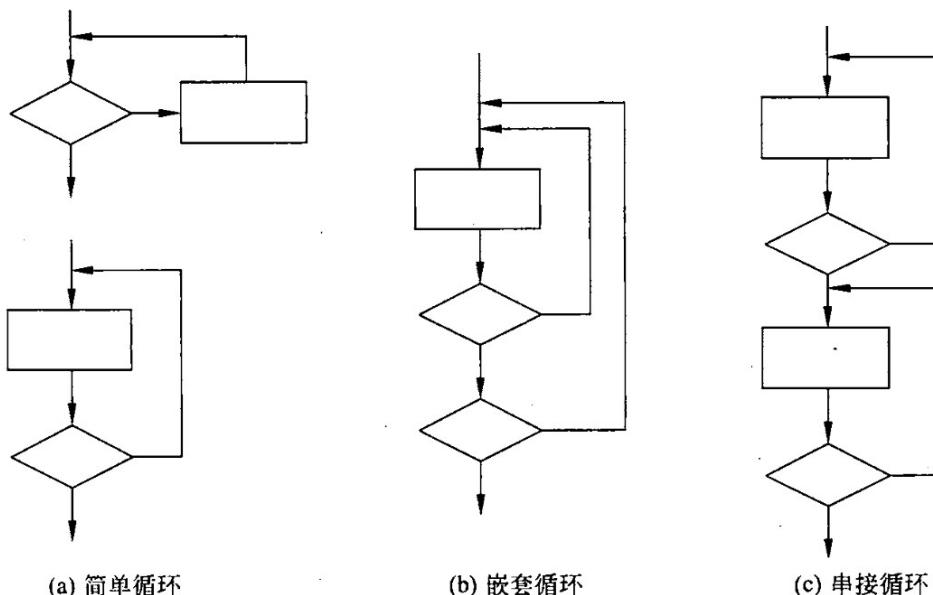


图 7.7 3 种循环

(2) 嵌套循环。如果把简单循环的测试方法直接应用到嵌套循环, 可能的测试数就会随嵌套层数的增加按几何级数增长, 这会导致不切实际的测试数目。B. Beizer 提出了一种能减少测试数的方法。

- 从最内层循环开始测试, 把所有其他循环都设置为最小值。
- 对最内层循环使用简单循环测试方法, 而使外层循环的迭代参数(例如, 循环计数器)取最小值, 并为越界值或非法值增加一些额外的测试。
- 由内向外, 对下一个循环进行测试, 但保持所有其他外层循环为最小值, 其他嵌套循环为“典型”值。
- 继续进行下去, 直到测试完所有循环。

(3) 串接循环。如果串接循环的各个循环都彼此独立, 则可以使用前述的测试简单循环的方法来测试串接循环。但是, 如果两个循环串接, 而且第一个循环的循环计数器值是第二个循环的初始值, 则这两个循环并不是独立的。当循环不独立时, 建议使用测试嵌套循环的方法来测试串接循环。

7.7 黑盒测试技术

黑盒测试着重测试软件功能。黑盒测试并不能取代白盒测试, 它是与白盒测试互补的测试方法, 它很可能发现白盒测试不易发现的其他类型的错误。

黑盒测试力图发现下述类型的错误:

- (1) 功能不正确或遗漏了功能。
- (2) 界面错误。