第3章 白盒测试用例设计方法

学习目标

- 语句覆盖
- 判定覆盖
- 条件覆盖
- 判定/条件覆盖
- 组合条件覆盖
- 路径覆盖
- 基本路径测试法
- 循环测试
- 代码审查
- 静态结构分析
- Rational Purify 应用

白盒测试以检查程序的内部结构和逻辑为根本，分为逻辑覆盖测试、路径测试以及静态代码审查等，白盒测试又可分为手工测试和应用工具测试，本章最后介绍了 Rational Purify 的简单应用。

3.1 逻辑覆盖测试

白盒测试方法是把测试对象看作一个打开的盒子，测试人员依据程序内部逻辑结构相关信息，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的状态，确定实际的状态是否与预期的状态一致。

逻辑覆盖测试是以程序内在逻辑结构为基础的测试，重点关注测试覆盖率。包括以下6种类型：语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、条件组合覆盖和路径覆盖。下面将以1995年软件设计师考试的一道考试题目为例进行讲解，程序流程图如图3-1所示。

3.1.1 语句覆盖

语句覆盖是指设计若干个测试用例，使程序中的每个可执行语句至少执行一次。在保证每条语句都运行的前提下，测试用例应尽量少。在语句覆盖的基础上可以实现程序段覆盖，进而是程序块的覆盖。

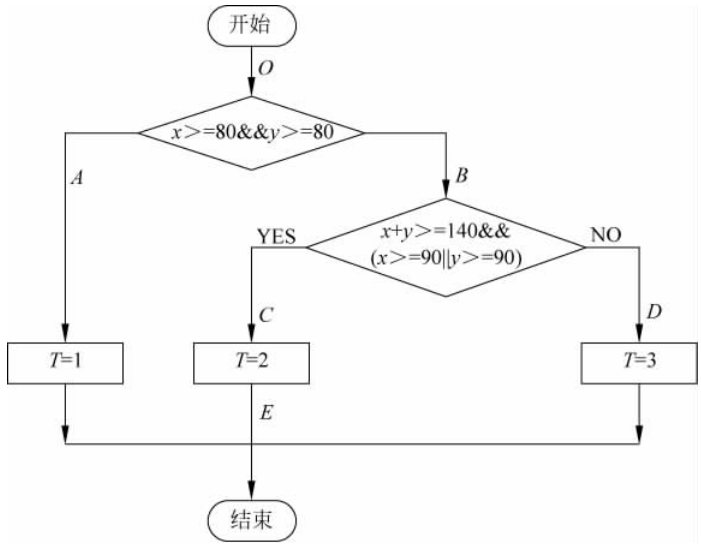

图 3-1 程序 1 流程图

以程序 1 为例,可执行语句有 3 条,能否设计一个测试用例使 3 条语句都得到运行? 答案是否定的,3 条语句位于 3 个分支上,所以应该设计 3 个测试用例,使程序运行 3 次才可能使得所有的可执行语句得到运行。语句覆盖的测试用例如表 3-1 所示。

表 3-1 语句覆盖的测试用例

序 号	X	Y	预 期 结 果	路 径
1	50	50	$T=3$	OBDE
2	90	70	$T=2$	OBCE
3	90	90	$T=1$	OAE

评价语句覆盖程度通常要借助语句覆盖率,即已执行的可执行语句占程序中可执行语句总数的百分比,表示为:已执行的可执行语句/程序中可执行语句总数。

$$\text{语句覆盖率} = \frac{\text{已执行的可执行语句}}{\text{程序中可执行语句总数}} \times 100\%$$

一般来讲,语句覆盖率越高越好,但它是最重要的衡量指标吗? 单纯的语句覆盖无法发现 && 与 || 用错的问题。另外,语句覆盖测试不能发现循环次数存在问题的程序,如:

```
while(i > 3 && i < 7)
{
    s = s + i;
    i++;
}
```

而程序本意:

```
while(i >= 3 && i <= 7)
{
    s = s + i;
    i++;
}
```

因此，语句覆盖测试是较弱的一种覆盖测试。

3.1.2　判定覆盖

比语句覆盖稍强的覆盖标准是判定覆盖，判定覆盖的含义是：设计足够多的测试用例，使程序中的每个判定都至少获得“真值”和“假值”。程序中的判定有：分支判定和循环判定。除了双值判定语句外，还有多值判定语句，如 case 语句，因此判定覆盖更一般的含义是：使得每一个判定获得每一种可能的结果至少一次。

为了表示方便、做到理解上的统一，本书做出如下约定：表示某个条件的真假使用 T*i* 和 F*i*（*i* 为该条件的序号），表示某个判定的真假使用 TD*i* 和 FD*i*（*i* 为该判定的序号）。如：

```
if (A<5 and B==5)
        x=x+2;
```

则表示条件如下：

A<5，记为 T1；*A*>=5，记为 F1

B==5，记为 T2；*B*!=5，记为 F2

表示判定如下：

A<5 and *B*==5 判定为真，记为 TD1；判定为假，记为 FD1

程序 1 中有两个判定

D1：*X*>=80 && *Y*>=80

D2：*X*+*Y*>=140 &&（*X*>=90||*Y*>=90）

根据判定覆盖的定义，应设计测试用例使 TD1、FD1 和 TD2、FD2 都出现，究竟要怎样组合取决于条件不能相互冲突。表 3-2 为可能的测试用例设计方案之一。

表 3-2　判定覆盖的测试用例

序号	状态	条　件	*X*	*Y*	预期结果	路径
1	TD1T1T2	*X*≥80，*Y*≥80	90	90	*T*=1	OAE
2	FD1F1T2	*X*<80，*Y*≥80，*X*+*Y*≥140	35	100	*T*=2	OBCE
	TD2T3F4T5	*X*<90，*Y*≥90				
3	FD1F1T2	*X*<80，*Y*≥80，*X*+*Y*≥140	75	75	*T*=3	OBDE
	FD2T3F4F5	*X*<90，*Y*<90				

表 3-2 中的状态包含了所有判定的真值与假值，因此该组测试用例满足判定覆盖的要求，同时也满足语句覆盖。但是，仍然无法发现程序段中存在的逻辑判定错误。

3.1.3　条件覆盖

条件覆盖的含义是：构造一组测试用例，使得每一个判定中每个逻辑条件的可能值至少被满足一次。程序中的判定分为单一条件的判定和多个条件的复合判定两种类型，如：

```
if(x>4)
        y=6;
```

这是单一条件构成的判定,其条件覆盖与判定覆盖的效果是一样的。而多条件的判定是指判定由 && 或者||连接起来的表达式,如:

```
if (A<5 and B==5)
    x=x+2;
```

有两个条件,分别是:

$A<5$　　记为 T1,$A>=5$　　记为 F1

$B==5$　　记为 T2,$B!=5$　　记为 F2

程序 1 中有两个判定,共有 5 个条件,标记如下:

- D1:(1)$X>=80$ &&(2) $Y>=80$
- D2:(3)$X+Y>=140$ &&((4) $X>=90$||(5) $Y>=90$)

其中括弧中的数字为该条件的编号。

则需要构造一组测试用例,使得每个条件的真假值至少被满足一次。可能的测试用例如表 3-3 所示。

表 3-3　程序 1 的条件覆盖测试用例

序号	状态	条　件	X	Y	预期结果	路径
1	TD1T1T2	$X\geqslant80$,$Y\geqslant80$	90	90	$T=1$	OAE
2	FD1F1T2	$X<80$,$Y\geqslant80$,$X+Y\geqslant140$	70	100	$T=2$	OBCE
	TD2T3F4T5	$X<90$,$Y\geqslant90$				
3	TD1T1F2	$X\geqslant80$,$Y<80$,$X+Y<140$	100	10	$T=3$	OBDE
	FD2F3T4F5	$X\geqslant90$,$Y<90$				

表 3-3 中的状态包含了所有条件的真值与假值,可见,这组测试用例是满足条件覆盖的。

思考:满足条件覆盖就是满足判定覆盖吗?请看下面的例子。

判定 $A<5$ and $B==5$,其满足条件覆盖的状态有:

T1F2 和 F1T2　　===〉　　FD1 和 FD1

或者

T1T2 和 F1F2　　===〉　　TD1 和 FD1

这两种组合都是满足条件覆盖的测试用例状态,读者可任选其一。但是,第 2 组同时满足判定覆盖,而第 1 组仅仅覆盖判定的一种取值。

为解决这一矛盾,需要兼顾多条件和分支来进行测试。

3.1.4　判定/条件覆盖

设计足够的测试用例,使得判定中每个条件的所有可能(真/假)至少出现一次,并且每个判定本身的判定结果(真/假)也至少出现一次。即,满足判定/条件覆盖的测试用例应该同时满足条件覆盖和判定覆盖。

程序 1 的判定/条件覆盖测试用例设计如表 3-4 所示。

表 3-4　程序 1 的判定/条件覆盖测试用例

序号	状态	条　　件	X	Y	预期结果	路径
1	TD1T1T2	$X\geqslant 80, Y\geqslant 80$	90	90	$T=1$	OAE
2	FD1F1T2	$X<80, Y\geqslant 80, X+Y\geqslant 140$	70	100	$T=2$	OBCE
	TD2T3F4T5	$X<90, Y\geqslant 90$				
3	TD1T1F2	$X\geqslant 80, Y<80, X+Y<140$	100	10	$T=3$	OBDE
	FD2F3T4F5	$X\geqslant 90, Y<90$				

表 3-4 中的状态包含了所有条件的真值与假值，以及所有判定的真值与假值。可见，这组测试用例是满足判定/条件覆盖的。

同理，对于：

```
if (A < 5 and B == 5)
        x = x + 2;
```

为满足判定/条件覆盖，应选择的组合是 T1T2 和 F1F2＝＝＝〉TD1 和 FD1。但是，若把逻辑运算符“&&”错写成“||”或第 2 个运算符“||”错写成“&&”，该测试用例仍然无法发现上述逻辑错误。

3.1.5　组合条件覆盖

组合条件覆盖的含义是：设计足够的测试用例，使得每个判定中条件的各种可能组合都至少出现一次。显然满足组合条件覆盖的测试用例是一定满足判定覆盖、条件覆盖和判定/条件覆盖的。

程序 1 的设计过程如下所示。

(1) 列出各个判定内条件的完全组合。

程序 1 的第 1 个判定的组合条件如下：

- T1 T2
- T1 F2
- F1 T2
- F1 F2

程序 1 的第 2 个判定的组合条件如下：

- F3　F4　F5
- F3　F4　T5
- F3　T4　F5
- F3　T4　T5
- T3　F4　F5
- T3　F4　T5
- T3　T4　F5
- T3　T4　T5

(2) 分析条件间的制约关系。

在程序 1 中隐含着这样的关系,如表 3-5 中的序号 1 所示,如果 $X<80$(F1),则不可能存在 $X>=90$(T4),在设计测试用例时需要考虑这种制约关系。

表 3-5 条件间的互斥关系

序 号	互 斥 关 系	
1	F1	T4
2	F2	T5
3	F3	T4T5
4	T1T2	T4T5

(3) 考虑判定间的条件组合。

根据第(2)步获得的互斥关系,两个判定的条件间可能存在组合,如图 3-2 所示。

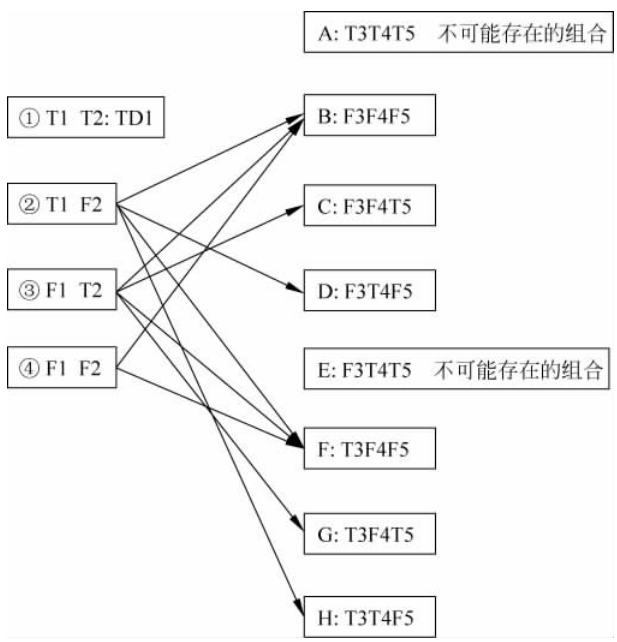

图 3-2 条件间可能的组合关系

具体的组合策略如下:

- 如果某个条件组合只有一种可能性,则优先确立关系。
- 不可能存在的组合不需要设计测试用例。
- 除了不可能的情况,所有的条件组合都必须至少出现一次。

因此,程序 1 的满足组合条件覆盖的测试用例的可能方案如表 3-6 所示。

组合条件覆盖的测试用例设计过程较复杂,在实际应用中难度较大。满足组合条件覆盖的测试用例一定是满足判定覆盖、条件覆盖和判定/条件覆盖的。

表 3-6　组合条件覆盖的测试用例

序号	状态	条　件	X	Y	预期结果	路径
1	TD1T1T2	$X\geqslant 80, Y\geqslant 80$	90	90	$T=1$	OAE
2	T1F2T3T4F5	$X\geqslant 80, Y<80, X+Y\geqslant 140$ $X\geqslant 90, Y<90$	100	70	$T=2$	OBCE
3	F1T2F3F4T5	$X<80, Y\geqslant 80, X+Y<140$ $X<90, Y\geqslant 90$	20	90	$T=2$	OBDE
4	F1T2T3F4T5	$X<80, Y\geqslant 80, X+Y\geqslant 140$ $X<90, Y\geqslant 90$	70	90	$T=2$	OBCE
5	T1F2F3F4F5	$X\geqslant 80, Y<80, X+Y<140$ $X<90, Y<90$	80	40	$T=3$	OBDE
6	F1T2T3F4F5	$X<80, Y\geqslant 80, X+Y\geqslant 140$ $X<90, Y<90$	40	120	$T=3$	OBDE

3.1.6　路径覆盖

所谓路径覆盖就是设计足够多的测试用例使每个路径都有可能被执行。程序 1 有 3 条可执行路径，设计测试用例如表 3-7 所示。

表 3-7　路径覆盖的测试用例

序号	状态	条　件	X	Y	预期结果	路径
1	TD1T1T2	$X\geqslant 80, Y\geqslant 80$	90	90	$T=1$	OAE
2	FD1F1T2 TD2T3F4T5	$X<80, Y\geqslant 80, X+Y\geqslant 140$ $X<90, Y\geqslant 90$	35	100	$T=2$	OBCE
3	FD1F1T2 FD2T3F4F5	$X<80, Y\geqslant 80, X+Y\geqslant 140$ $X<90, Y<90$	75	75	$T=3$	OBDE

3.2　基本路径测试

在测试实践中，一个不太复杂的程序，其可能的执行路径数都可能是一个庞大的数字，因此要在测试中实现路径覆盖是不现实的。为了解决这一难题，只有把覆盖的路径数压缩到一定数量范围内，例如，程序中的循环体只执行一次或较少次数。下面介绍的基本路径测试就是这样一种测试方法，它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，从而设计出测试用例的方法。设计出的测试用例要保证在测试中程序的每一个可执行语句至少被执行一次。

为了清晰描述基本路径测试方法，需要首先对其中几个基本概念进行说明，包括程序控制流图的符号、计算环形复杂度的方法、独立路径的确定等。

3.2.1　控制流图

在设计程序时，为了更加突出控制流的结构，可对程序流程图进行简化，简化后的流程

图称为控制流图。

(1) 控制流图的构成。

简化后所涉及的图形符号只有两种,即结点和控制流线。如图 3-3 所示的程序流程图可转化为如图 3-4 所示的程序控制流图。

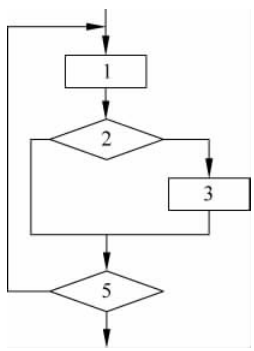

图 3-3 程序流程图

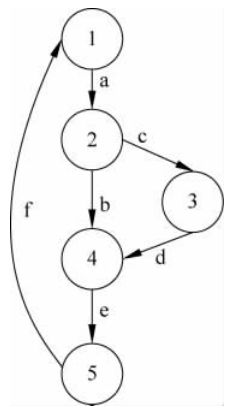

图 3-4 程序控制流图

结点是标有编号的圆圈,在转换控制流图时,下列情况必须用结点表示。

- 程序流程图中矩形框所表示的处理
- 菱形框表示的两个甚至多个出口判断
- 多条流线相交的汇合点

边是由带箭头的弧或线表示,与程序流程图中的流线一致,表明了控制的顺序,它代表程序中的控制流,通常标有名字。

(2) 常见控制结构的控制流图,如图 3-5 所示。

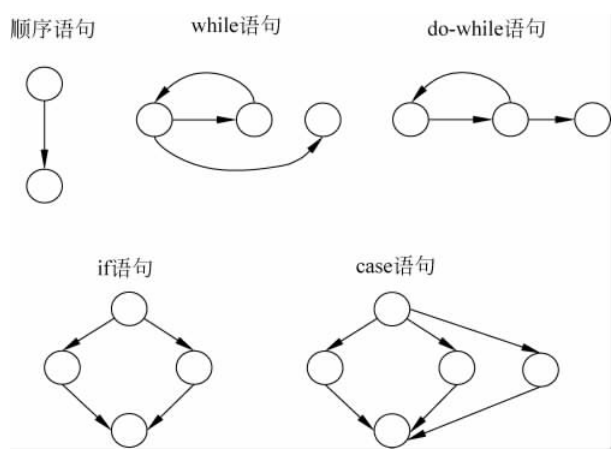

图 3-5 常见控制结构的控制流图

请读者仔细区分 while 语句和 do-while 语句的控制流图。

(3) 将程序流程图转成控制流图,转换时需要遵照如下规则。

包含条件的节点被称为判断结点(也叫谓词结点),由判断结点发出的边必须终止于某一个节点,由边和结点所限定的范围被称为区域。

这里我们假定在流程图中用菱形框表示的判定条件内没有复合条件(即单一条件判定),而一组顺序处理框可以映射为一个单一的结点。

控制流图中的箭头(边)表示了控制流的方向,类似于流程图中的流线,一条边必须终止于一个结点。

在选择或者是多分支结构中分支的汇聚处,即使汇聚处没有执行语句也应该添加一个汇聚结点。

图 3-6 对应的控制流图如图 3-7 所示。

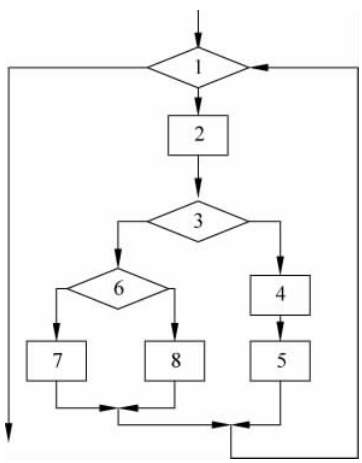

图 3-6　程序流程图

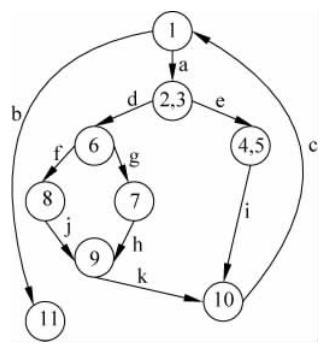

图 3-7　程序控制流图

图 3-8 对应的控制流图如图 3-9 所示。

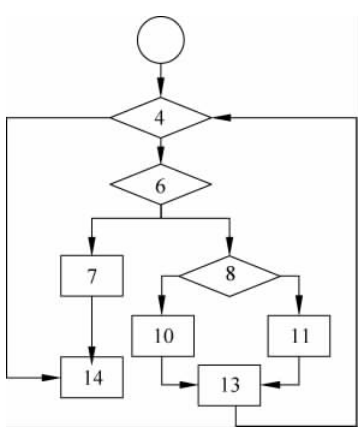

图 3-8　程序流程图

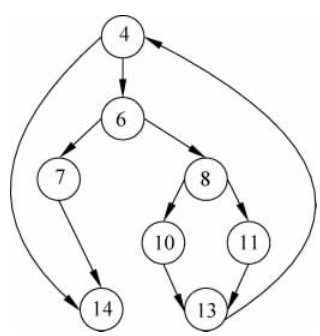

图 3-9　程序控制流图

(4) 复合条件的控制流图。

如果判定中的条件表达式是复合条件,即条件表达式是由一个或多个逻辑运算符连接的逻辑表达式,则需要将复合条件的判断改变为一系列只有单个条件的嵌套的判断。变换

过程如图 3-10 所示。

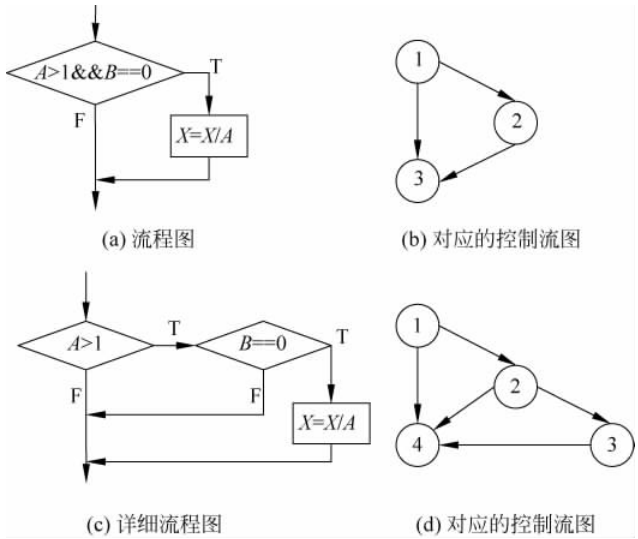

图 3-10 复合条件判定的控制流图

如果判定条件是 $A>1||B==0$,则转化过程如图 3-11 所示。

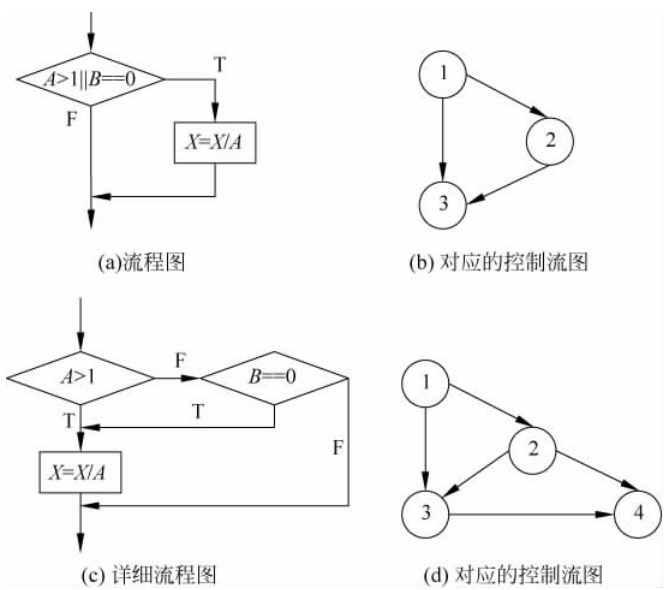

图 3-11 复合条件判定的控制流图

3.2.2 环形复杂度

环形复杂度(也称为圈复杂度)是一种为程序逻辑复杂度提供定量尺度的软件度量。也可将该度量用于基本路径测试方法,它可以提供程序基本集的独立路径数量和确保所有语句至少被执行一次的测试数量上界。计算环形复杂度的方法有 3 种。

(1) 流图中区域的数量对应于环形复杂度。

(2) 给定流图 G 的环形复杂度为 $V(G)$，定义为 $V(G)=E-N+2$，E 是流图中边的数量，N 是流图中节点的数量。

(3) 给定流图 G 的环形复杂度 $V(G)$，定义为 $V(G)=P+1$，P 是流图 G 中判定结点的数量。

对应图 3.10 的环形复杂度的计算方法如下所示。

- 流图中有 4 个区域。
- $V(G)=10$(条边)-8(结点)$+2=4$。
- $V(G)=3$(个判定结点)$+1=4$。

3.2.3　独立路径

独立路径是指程序中至少引入一个新的处理语句集合或一个新条件的程序通路，它必须至少包含一条在本次定义路径之前不曾用过的边。程序的环形复杂度是程序基本路径集合中的独立路径条数，这是确定程序中每个可执行语句至少被执行一次所必需的测试用例数目的上界。

图 3.10 的环形复杂度是 4，可能写出如下的独立路径：

(1) 4－14。

(2) 4－6－7－14。

(3) 4－6－8－10－13－4－14。

(4) 4－6－8－11－13－4－14。

3.2.4　基本路径法的应用

以图 3-12 所示的程序为例，应用基本路径法设计测试用例。

第一步：画出控制流图，如图 3-13 所示。

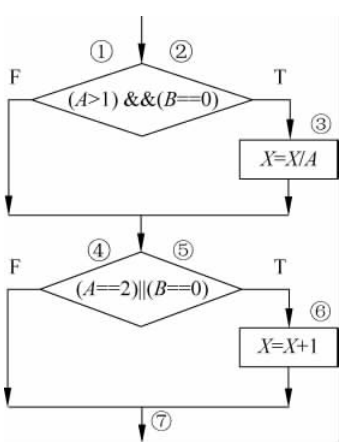

图 3-12　程序流程图

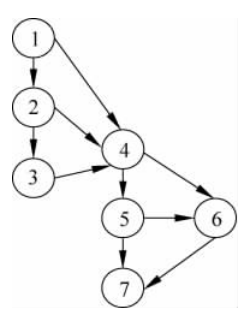

图 3-13　程序控制流图

第二步：计算圈图复杂度。

$$V(G) = E - N + 2 = 11 - 8 + 2 = 5$$

或者

$$V(G) = P + 1 = 4 + 1 = 5$$

第三步：导出独立路径。

- 路径 1：1—2—3—4—5—6—7
- 路径 2：1—2—3—4—5—7
- 路径 3：1—4—5—6—7
- 路径 4：1—2—4—5—7

补充说明：可能存在的路径如下所示。

(1) 1,4,5,7。

(2) 1,2,3,4,5,6,7。

(3) 1,2,4,6,7。

(4) 1,4,5,6,7。

(5) 1,2,3,4,5,6,7。

(6) 1,2,4,5,6,7。

(7) ……

这里不再一一列举，仅就怎样确认一条路径为独立路径加以说明。

至第 1 条路径包括的边有{(1,4)、(4,5)、(5,7)}

至第 2 条路径包括的边有{(1,4)、(4,5)、(5,7)、(1,2)、(2,3)、(3,4)、(5,6)、(6,7)}

注：其中的(4,5)已经在前面路径中出现过，在此不认做新的边。

至第 3 条路径包括的边有{(1,4)、(4,5)、(5,7)、(1,2)、(2,3)、(3,4)、(5,6)、(6,7)、(2,4)、(4,6)}

注：其中的(1,2)和(6,7)已经在前面路径中出现过，在此不认做新的边。

至第 4 条路径包括的边有{(1,4)、(4,5)、(5,7)、(1,2)、(2,3)、(3,4)、(5,6)、(6,7)、(2,4)、(4,6)}

注：所有的边都已经在前面路径中出现过，此路径不是新的独立路径。

第四步：设计测试用例，如表 3-8 所示。

表 3-8　基本路径法设计的测试用例

编　　号	输入数据			输出数据		覆盖路径
	A	B	X	$X=X/A$	$X=X+1$	
1	2	0	2	1	?	1-2-3-4-5-6-7
2	3	0	2	2/3	5/3	1-2-3-4-5-7
4	−1	0	1	?	?	1-4-5-6-7
5	3	2	6	?	7	1-2-4-5-7

注："?"表示不会执行的判定条件。

3.3　循环测试

循环是代码中很重要的部分，通常程序中有 4 种循环结构：简单循环、嵌套循环、串接循环和不规则循环，如图 3-14 所示。下面介绍如何设计循环的测试。

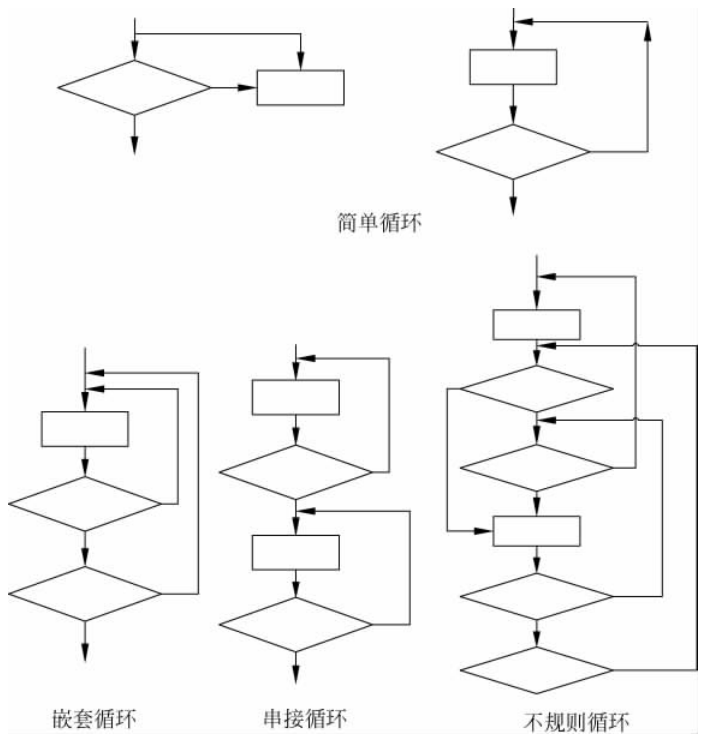

图 3-14　循环结构类型

(1) 简单循环的测试(假设有 n 次循环)。

① 跳过整个循环→ 0 次。

② 只有一次通过循环→1 次。

③ 某个 m 次通过循环，$m<n$→多次。

④ $n-1$，n，$n+1$ 次通过循环→上限边界。

(2) 嵌套循环的测试集。

① 把外循环设置为最小值并运行内循环所有可能的情况。

② 把外循环设置为最大值并运行内循环所有可能的情况。

③ 把内循环设置为最小值并运行内循环所有可能的情况。

④ 把内循环设置为最大值并运行内循环所有可能的情况。

⑤ 把所有循环变量都设置为最小值运行。

⑥ 把所有循环变量都设置为最大值运行。

(3) 串接循环的测试集。

① 如果两个循环相互独立，分别采用简单循环方式进行测试。

② 如果互相不独立,比如上一个循环的计数是下一个循环计数的初始值,推荐使用嵌套循环方式进行测试。

若是不规则循环,则尽量把这些循环重新设计为结构化的程序设计再进行测试。

3.4 代码检查

在实际使用中,静态代码检查比动态测试更有效率,更能快速找到缺陷。按经验估算,一般能发现30%~70%的逻辑设计和编码错误的缺陷。但是静态代码检查非常耗费时间,而且代码检查需要丰富的知识和经验积累。

静态测试包括代码检查和静态分析两种途径。它可以由人工进行,充分发挥人的逻辑思维优势,也可以借助软件工具自动进行。代码检查包括桌面检查、代码审查、代码走查、技术评审等。主要检查代码的设计是否一致、代码是否遵循标准性和可读性、代码的逻辑表达的正确性、以及代码结构的合理性等。

代码评审也称代码复查,是指通过阅读代码来检查源代码与编码标准的符合性以及代码质量的活动。代码评审有两个目的:第一个目的是确保要发布质量可靠的代码,能非常有效地发现所有类型的错误;第二个目的是作为教学工具帮助开发人员学会何时并且如何应用技术来提高代码的质量、一致性和维护性。

实践证明,代码评审是发现缺陷的有效方法。代码评审包括代码审查、代码走查、桌面检查等。

3.4.1 代码审查

代码审查作为质量保证的一部分,是静态测试的主要手段之一。代码审查包括以下内容。

(1) 编码规范问题:命名不规范、注释的质量等。

(2) 代码结构问题:重复代码、分层不当、紧耦合。

(3) 工具、框架使用不当:Spring、Hibernate、AJAX。

(4) 实现问题:错误验证、异常处理、线程、性能、安全问题。

(5) 测试问题:测试覆盖度、可测试性。

下面对代码审查的具体操作技术做简要说明。

(1) 审查的内容。评审代码选择的一般循序:最近一次迭代开发的代码;系统关键模块;业务较复杂的模块;缺陷率较高的模块;对于能力不足的成员所完成的代码,也要重点进行评审。

(2) 审查的流程,如表3-9所示。

(3) 评审的参与者:参与的人数可按项目来分,同一个项目中包含开发人员、设计人员、测试人员以及经验丰富的程序员,这样就能从不同角度进行评审代码。一般由经验丰富的程序员作为主评审员。

(4) 评审的时间和程序量:一般为1小时~2小时,基本上不会超过2小时,代码量在200行~400行。

表 3-9　审查的流程

代码审查	工作内容
会前准备	• 组织者应通知各参与者本次评审的范围，需要给出问题详细描述以及相关代码在SVN上的URL地址等 • 参与者应在会议前阅读源代码，列出发现的问题和亮点，并汇总给组织者 • 要指定评审的记录人
会议议程	• 如果是第一次会议，应先由该项目开发组长做整体介绍，参加者依次发言，应结合代码讲解发现的问题；每讲完一个问题，针对其展开讨论，将每个问题的讨论时间控制在10分钟以内 • 记录人应准确记录会上提出的所有问题、亮点及最终结论，供团队借鉴和跟踪 • 评审会议以确认问题为主，而不是讨论解决方案 • 每个角色也应明白各自的评审重点，如将QA重点放在编程规范、测试人员侧重于可测性、系统专家侧重于整体（如对其他功能的影响、性能等）
问题确认与追踪	那些大家达成一致认可的问题可由代码完成人提出解决方案，方案要得到问题发现者的同意，然后编程人员编码实现该方案，并进行测试和验证，并将验证结果提交给问题发现人，问题发现人确认无误后，该问题就可关闭

（5）几点建议：

① 作为项目成员，在代码编写完成后，首先要自检，不计算这时发现的缺陷；然后是项目组内的评审，应计算这时发现的缺陷；最后才是外部评审。

② 对大型软件的检查应安排多个代码检查会议同时进行，每个代码检查会议处理一个或几个模块或子程序。

③ 提出的建议应针对程序本身，而不应针对程序员；程序员必须怀着非自我本位的态度来对待错误检查，对整个过程采取积极和建设性的态度。

④ 项目经理要能够争取到足够的、合适的领域专家来参与评审，要提前协调。

3.4.2　代码走查

代码走查是以小组为单元进行代码阅读的，同样也是一系列规程和错误检查技术的集合。且代码走查也采用了持续1小时～2小时的不间断会议的形式。

成员组成：一般是由3人～5人组成，其中一人扮演“协调人”；一人担任秘书角色，负责记录所有查出的错误；还有一人担任测试人员。建议最佳的组合应该是：一位极富经验的程序员；一位程序设计语言专家；一位程序员新手（可以给出新颖、不带偏见的观点）；最终将维护程序的人员；一位来自其他不同项目的人员；一位来自该软件编程小组的程序员。

代码走查的流程与代码检查很类似，这里仅列出不同之处。即代码走查的任务：就是参与者“使用了计算机”。被指定为测试人员的那个人会带着一些书面的测试用例（程序或模块具有代表性的输入集及预期的输出集）来参加会议。且在会议期间，每个测试用例都在人们的头脑中进行推演。即把测试数据沿程序的逻辑结构走一遍，并把程序的状态（如变量的值）记录在纸张或白板上以供监视。

这些书面的测试用例必须结构简单、数量较少，因为人脑执行程序的速度比计算机执行

程序的速度慢上若干个量级。之所以提供这些测试用例,目的不在于其本身对测试起了多关键的作用,而是其提供了启动代码走查和质疑程序员逻辑思路及其设想的手段。因为,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。

3.4.3 桌面检查

桌面检查可被视为由单人进行的代码检查或代码走查;由一个人阅读程序,对照错误列表检查程序(详见本书附录部分),对程序推演测试数据。但是,桌面检查的效果不是很理想,原因是:单人检查完全没有约束;开发人员测试或检查自己程序的效果很不理想;检查者没有展示自己能力的机会,缺乏良好的效应。其结论是桌面检查胜于没有检查,但其效果远远逊于代码审查和代码走查。

3.5 Rational Purify 应用

自动化测试工具 Rational Purify 是 Rational PurifyPlus 工具中的一种。Rational Purify 适合查找典型的 Visual C/C++程序中的传统内存访问错误,以及 Java 代码中与垃圾内存收集相关的错误。将 Rational Robot 的回归测试与 Rational Purify 结合使用可完成可靠性测试。

Purify 提供了一套功能强大的内存使用状况分析工具,可以找出消耗了过量内存或者保留了不必要对象指针的函数调用。Rational Purify 可以运行 Java applet、类文件或 JAR 文件,支持 JVM 阅读器或 Microsoft Internet Explorer 等容器程序。

本节简要介绍 Rational Purify 的应用。

3.5.1 Purify 概述

Purify 是主要针对开发阶段的白盒测试,是综合性检测运行时错误的工具,并可以和其他复合应用程序(包括多线程和多进程程序)一起工作。Purify 将检查每一个内存操作,定位错误发生的地点并提供尽可能详细的信息帮助程序员分析错误发生的原因。

它可以发现的主要错误有:

(1) Reading or writing beyond the bounds of an array (数组读写越界)

(2) Using uninitialized memory (使用未初始化的内存)

(3) Reading or writing freed memory (读写未分配的内存)

(4) Reading or writing beyond the stack pointer (栈指针读写越界)

(5) Reading or writing through null pointers (读写空指针)

(6) Leaking memory and file descriptors (内存和文件描述符泄漏)

Purify 还可检查一些其他错误,如调用函数参数错误等。

由于 Purify 对内存的分析和记录是在程序运行完成以后才显示,如果需要在程序运行时观测就很不方便,所以 Purify 也提供外接 API 函数帮助在运行时显示内存状况以调试程序。

在项目开发测试中适用 Purify 的领域有以下两个。

(1) 使用 Purify 提供的 API 函数,在程序运行的必要环节,在观察器中显示需要获得的内存状况或打印消息。

(2) 对于运行环境要求简单的程序,如可以在自己虚拟机上运行的单机程序,可以使用 Purify 进行白盒测试,查找内存泄漏等运行时错误。

而对硬件有要求的程序,则不大可能使用 Purify。

3.5.2　Purify 实际运用

用 Visual C++ 6.0 新建一个空的 Win32 Console Application 工程,工程名为 HelloWorld,然后新建 C++源文件,并将其命名为 HelloWorld.cpp,编辑源文件,输入如下代码。

```
int main(){
    char * str1 = "hello";
    char * str2 = new char[5];

    char * str3 = str2;
    cout << str2 << endl;

    strcpy(str2,str1);
    cout << str2 << endl;

    delete str2;
    str2[0] + = 2;

    delete str3;
    return 0;
}
```

编写完成后,编译连接,生成 hello.exe 文件,在 Debug 目录下可以找到该文件。该程序虽然能够编译通过,但很明显有很多内存错误,使用 Purify 可以检测到这些内存错误。

接下来,启动 Purify,在 Purify 窗口中选择 File|Run 命令,或者按下 F5 键,可以打开如图 3-15 所示的窗口。

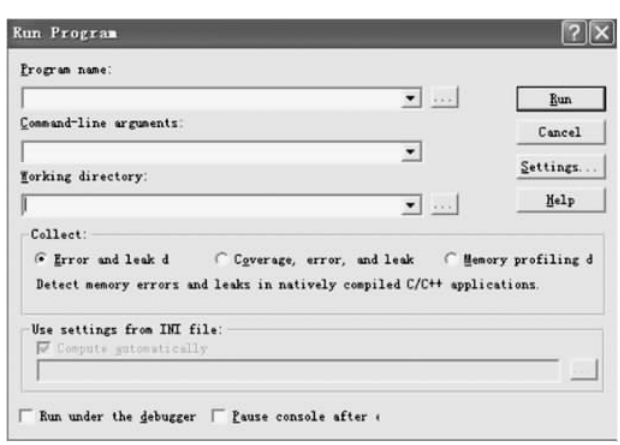

图 3-15　运行程序窗口

在 Program name 对应的文本框中,单击 ... 按钮找到 Hello.exe 所在路径,并选择该文件。Working directory 被自动设置为 Hello.exe 所在的目录。

如果要分析代码覆盖率等,可以选择 Coverage,error,and leak 选项。

单击 Run 按钮,程序开始运行。运行结束后,Purify 窗口中出现检测结果。

(1) 在窗口右侧选择 Error View 选项卡,如图 3-16 所示。

图 3-16 检测结果

在图 3-16 中黄色叹号标注的错误为 UMR,即未初始化内存读(Uninitialized Memory Read);红色叹号标注的错误有 ABR(数组越界读)、IPR(非法指针读)、ABW(数组越界写)、FMR(对已释放内存读)、FMW(对已释放内存写)、FFM(释放已经被释放的内存)。

单击第一个 UMR 前的加号,可以查看该错误的详细信息,如图 3-17 所示。

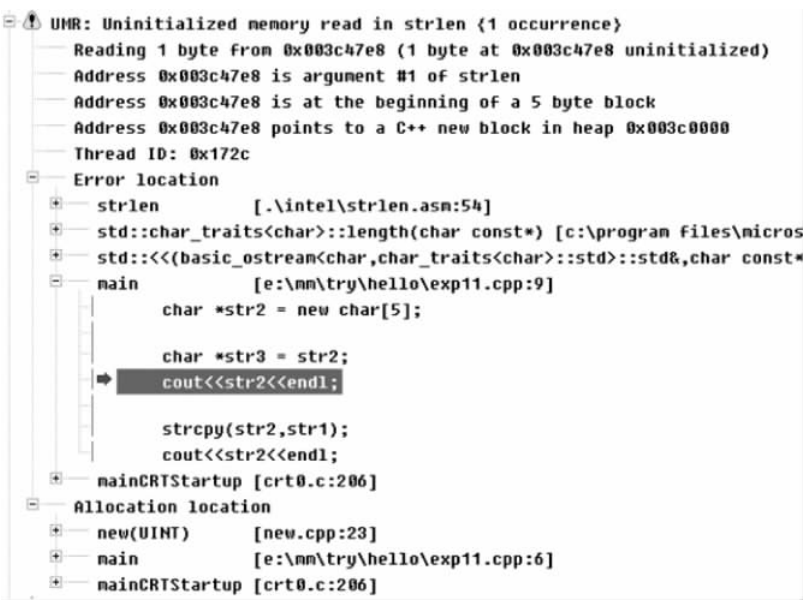

图 3-17 查看详细的错误信息

如果被测程序包含源代码,该错误的详细信息中会列出错误的代码行并解释造成该内存错误的原因。在图 3-17 中,Error location 部分指出错误在源代码中的位置,在 main 部分告诉测试者,错误在第 9 行,箭头所指部分即为源代码存在错误的行。很明显,该行试图输出 str2 指向的字符串,但实际上 str2 没有被初始化,因此产生了 UMR 错误。Allocation

location 指出错误的内存分配位置，main 部分表明是在第 6 行，展开可以查看具体的代码位置，如图 3-18 所示。

图 3-18　错误的内存分配在代码中的位置

根据检测结果，判断出错位置之后，可以有针对性地对代码进行修改，并修正该错误。

保存该检测结果信息，在工作目录中生成一个 Hello. pfy 文件。在工作目录中会生成一个日志文件，默认为 Hello_pure. log 文件。

以上就检测结果中的其中一个错误进行了分析，读者可以参照分析其他错误，找到错误原因及位置。

(2) 选择 Module View 选项卡，如图 3-19 所示。

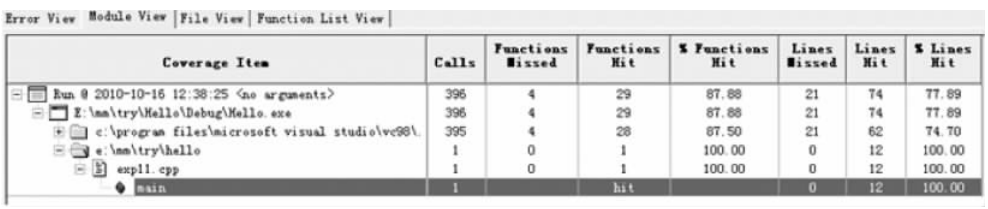

图 3-19　Module View 选项卡

双击 main，可以查看 main 函数的代码覆盖情况，如图 3-20 所示。

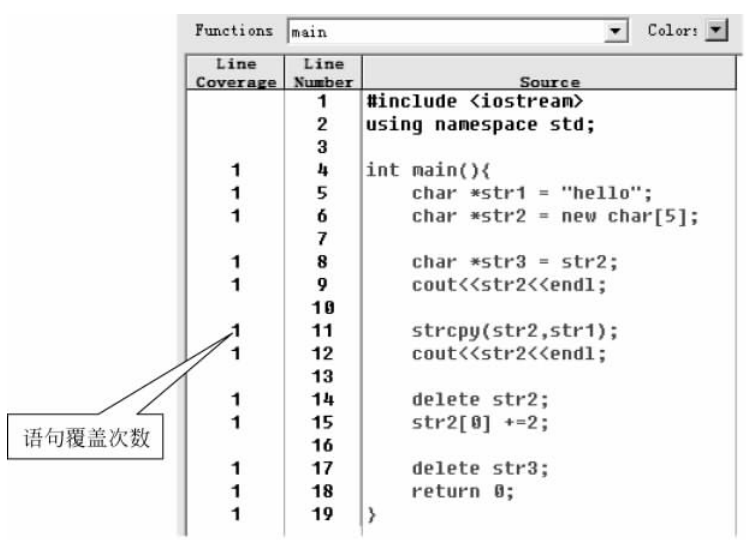

图 3-20　main 函数的代码覆盖情况

在 File View 选项卡中可以查看此处运行过程中覆盖到的源文件,包括一些库函数所在的源文件,如图 3-21 所示。

Coverage Item	Calls	Functions Missed	Functions Hit	% Functions Hit	Lines Missed	Lines Hit	% Lines Hit
Run @ 2010-10-16 12:38:25 <no arguments>	396	4	29	87.88	21	74	77.89
c:\program files\microsoft visual studio\vc98\include	395	4	28	87.50	21	62	74.70
ios	22	2	3	60.00	3	4	57.14
iosfwd	204	0	4	100.00	0	4	100.00
ostream	33	0	10	100.00	15	40	72.73
streambuf	89	2	4	66.67	3	5	62.50
xiosbase	42	0	6	100.00	0	8	100.00
xlocale	5	0	1	100.00	0	1	100.00
e:\mm\try\hello	1	0	1	100.00	0	12	100.00
exp11.cpp	1	0	1	100.00	0	12	100.00

图 3-21 File View 选项卡

在 Function List View 选项卡中可以查看本次运行过程中覆盖到的函数,包括库函数。

本章小结

本章主要讲解了白盒测试方法中的代码检查法、逻辑覆盖法和基本路径测试。其中代码检查的方法有代码审查、代码走查和桌面检查 3 种。

逻辑覆盖包含 6 种,简单总结如下所示。

1. 语句覆盖

主要特点:语句覆盖是最起码的结构覆盖要求,语句覆盖要求设计足够多的测试用例,使得程序中每条语句至少被执行一次。

优点:可以很直观地从源代码得到测试用例,无须细分每条判定表达式。

缺点:由于这种测试方法仅仅针对程序逻辑中显式存在的语句,但对于隐藏的条件和可能到达的隐式逻辑分支,是无法测试的。在 if 结构中若源代码没有给出 else 后面的执行分支,那么语句覆盖测试就不会考虑这种情况。但是我们不能排除这种分支以外的分支不会被执行,而往往这种错误会经常出现。再如,在 do-while 结构中,语句覆盖执行其中某一个条件分支。那么显然,语句覆盖对于多分支的逻辑运算是无法全面反映的,它只在乎运行一次,而不考虑其他情况。

2. 判定覆盖

主要特点:判定覆盖又称为分支覆盖,它要求设计足够多的测试用例,使得程序中每个判定至少有一次为真值,有一次为假值,即:程序中的每个分支至少被执行一次。每个判断的取真、取假至少被执行一次。

优点:判定覆盖比语句覆盖要多几乎一倍的测试路径,当然也就具有比语句覆盖更强的测试能力。同样判定覆盖也具有和语句覆盖一样的简单性,无须细分每个判定就可以得到测试用例。

缺点:往往大部分的判定语句是由多个逻辑条件组合而成(如,判定语句中包含 AND、OR、CASE),若仅仅判断其整个最终结果,而忽略每个条件的取值情况,必然会遗漏部分测试路径。

3. 条件覆盖

主要特点:条件覆盖要求设计足够多的测试用例,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。

优点：显然条件覆盖比判定覆盖，增加了对符合判定情况的测试，增加了测试路径。

缺点：要达到条件覆盖，需要足够多的测试用例，但条件覆盖并不能保证判定覆盖。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。

4. 判定/条件覆盖

主要特点：设计足够多的测试用例，使得判定中每个条件的所有可能结果至少出现一次，每个判定本身所有可能结果也至少出现一次。

优点：判定/条件覆盖满足判定覆盖准则和条件覆盖准则，弥补了两者的不足。

缺点：判定/条件覆盖准则的缺点是未考虑条件的组合情况。

5. 组合覆盖

主要特点：要求设计足够多的测试用例，使得每个判定中条件结果的所有可能组合至少出现一次。

优点：多重条件覆盖准则满足判定覆盖、条件覆盖和判定/条件覆盖准则。更改的判定/条件覆盖要求设计足够多的测试用例，使得判定中每个条件的所有可能结果至少出现一次，每个判定本身的所有可能结果也至少出现一次。并且每个条件都显示能单独影响判定结果。

缺点：线性地增加了测试用例的数量。

6. 路径覆盖

主要特点：设计足够多的测试用例，覆盖程序中所有可能的路径。

优点：这种测试方法可以对程序进行彻底测试，比前面5种的覆盖面都广。

缺点：由于路径覆盖需要对所有可能的路径进行测试(包括循环、条件组合、分支选择等)，那么需要设计大量、复杂的测试用例，使得工作量呈指数级增长。而在有些情况下，一些执行路径是不可能被执行的，如：

```
if(!A)B++;
if(!A)D--;
```

这两个语句实际只包括了两条执行路径，即A为真或假的时候对B和D的处理，真或假不可能都存在，而路径覆盖测试则认为是包含了真与假的4条执行路径。这样不仅降低了测试效率，而且大量的测试结果的累积，也为排错带来麻烦。

基本路径法需要首先画出程序控制流图，计算圈复杂度，进而写出独立路径，最后为每条独立路径设计一个测试用例。

循环测试需要根据循环的具体类型选择不同的测试策略。