

3-2 白盒测试

2019年4月14日 10:27

[1 白盒测试概述](#)

[2 测试覆盖标准](#)

[3 基本路径法](#)

[4 循环测试](#)

[5 xUnit白盒测试](#)

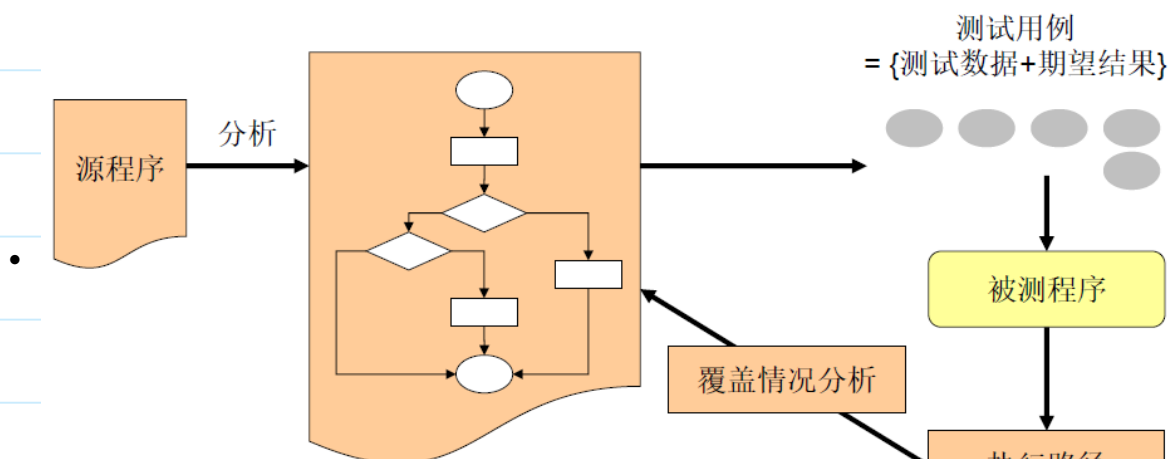
1 白盒测试概述

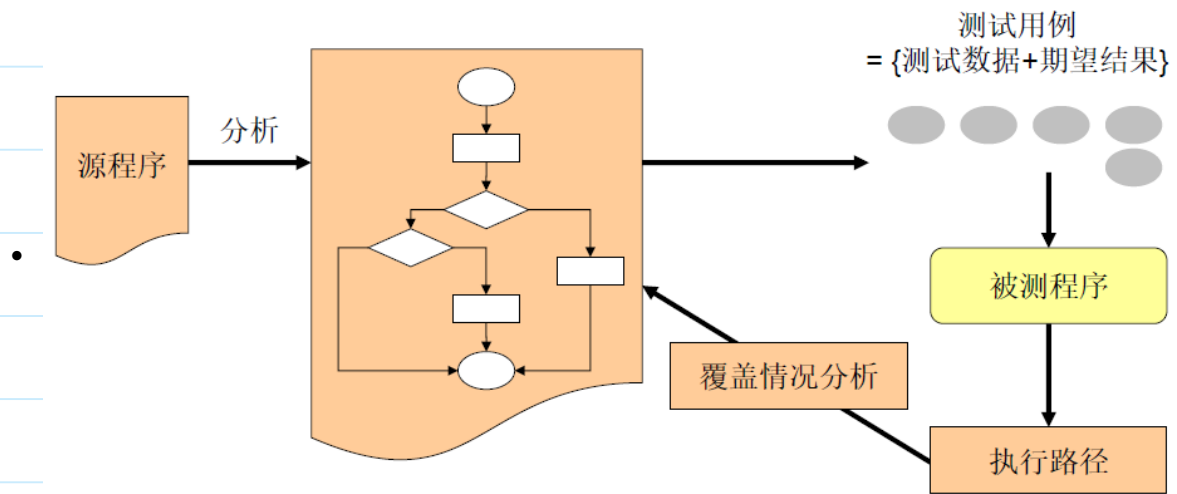
白盒测试的概念

- 白盒测试(又称为“结构测试”或“逻辑驱动测试”)
- 把测试对象看做一个透明的盒子，它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。

白盒测试的目的

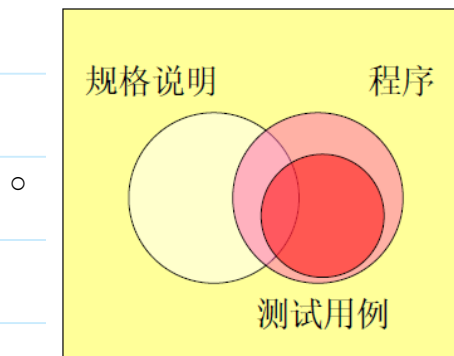
- 白盒测试主要对程序模块进行如下的检查：
 - 对模块的每一个独立的执行路径至少测试一次；
 - 对所有的逻辑判定的每一个分支(真与假)都至少测试一次；
 - 在循环的边界和运行界限内执行循环体；
 - 测试内部数据结构的有效性；
- 代码评审：靠人发现代码中不符合规范的地方、潜在的错误；
- 代码性能分析：发现代码中的性能缺陷；
- 白盒测试：发现代码中的错误！
- 测试用例中的输入数据从程序结构导出，但期望输出务必从需求规格中导出。



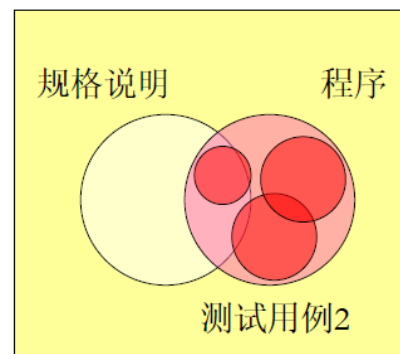
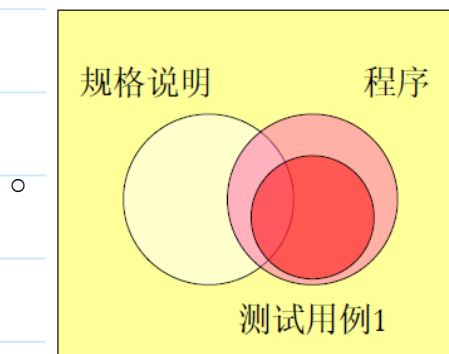


白盒测试的VennDiagram

- 注意：覆盖区域只能在程序所实现的部分



- 测试用例所覆盖的程序实现范围越大，就越优良.设计良好的测试用例，使之尽可能完全覆盖软件的内部实现



2 测试覆盖标准

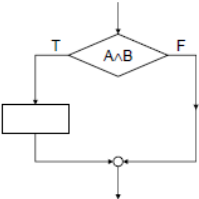
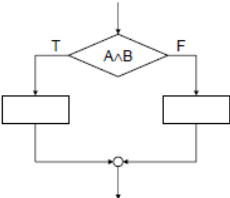
- 白盒测试的特点：
 - 以程序的内部逻辑为基础设计测试用例，又称逻辑覆盖法。
 - 应用白盒法时，手头必须有程序的规格说明以及程序清单。
- 白盒测试考虑测试用例对程序内部逻辑的覆盖程度：
 - 最彻底的白盒法是覆盖程序中的每一条路径，但是由于程序中一般含有循环，所以路径的数目极大，要执行每一条路径是不可能的，只能希望覆盖的程度尽可能高些。

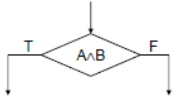

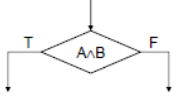
- 对一个具有多重选择和循环嵌套的程序，不同的路径数目可能是天文数字。
 - 举例：某个小程序的流程图，包括了一个执行20次的循环。

包含的不同执行路径数达5的20次方条，对每一条路径进行测试需要1ms，假定一年工作365×24小时，要把所有路径测试完，需3170年。
- 为了衡量测试的覆盖程度，需要建立一些标准，目前常用的一些覆盖标准从低到高分别是：
 - **逻辑覆盖：**
 - 语句覆盖
 - 判定覆盖(分支覆盖)
 - 条件覆盖
 - 判定/条件覆盖
 - 条件组合覆盖
 - **控制结构覆盖：**
 - 基本路径测试
 - 循环测试
 - 条件测试
 - 数据流测试

五种覆盖标准的对比

发现错误的能力	弱	语句覆盖	每条语句至少执行一次
		判定覆盖	每一判定的每个分支至少执行一次
		条件覆盖	每一判定中的每个条件，分别按“真”、“假”至少各执行一次
		判定/条件覆盖	同时满足判定覆盖和条件覆盖的要求
	强	条件组合覆盖	求出判定中所有条件的各种可能组合值，每一可能的条件组合至少执行一次

覆盖标准	程序结构举例	测试用例应满足的条件
语句覆盖		$A \wedge B = T$
判定覆盖		$A \wedge B = T$ $A \wedge B = F$

覆盖标准	程序结构举例	测试用例应满足的条件
条件覆盖		$A=T, A=F$ $B=T, B=F$
判定/条件覆盖		$A \wedge B=T, A \wedge B=F$ $A=T, A=F$ $B=T, B=F$
条件组合覆盖		$A=T \wedge B=T$ $A=T \wedge B=F$ $A=F \wedge B=T$ $A=F \wedge B=F$

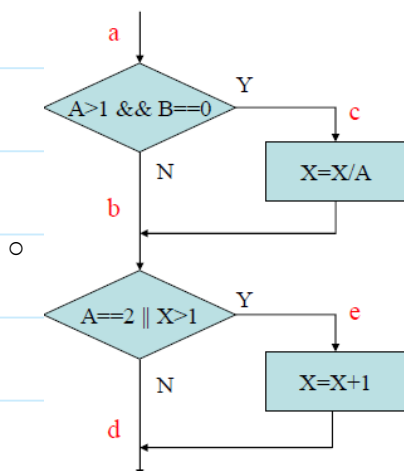
语句覆盖

- 语句覆盖(StatementCoverage):
 - 为了暴露程序中的错误，应选择足够多的测试数据，使被测程序中的每条语句至少应该执行一次。
 - 语句覆盖是最弱的测试标准。
- 例如：

```
float example (float A, float B,
float X)
{
    if (A>1 && B==0)
        X = X/A;

    if (A==2 || X>1)
        X = X+1;
}
```

测试用例：A=2, B=0, X=3



- 语句覆盖实际是很弱的：
 - 如果第一个条件语句中的“&&”错误的写成“||”，测试用例(A=2,B=0,X=3)无法发现这个错误；
 - 如果第三个条件语句中X>1误写成X>0，这个测试用例也不能暴露它

- 一般认为“语句覆盖”是很不充分的一种标准，是最弱的逻辑覆盖准则。

- 语句覆盖的另一个例子

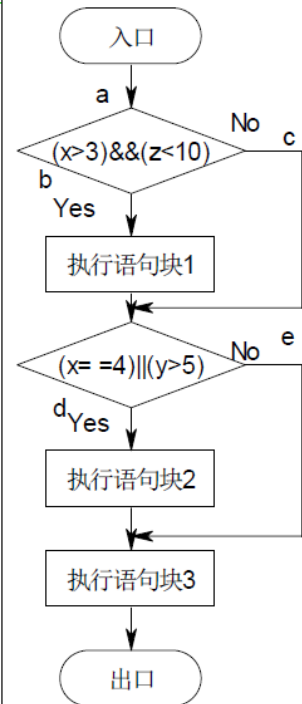
```
void DoWork (int x, int y, int z)
{
    int k = 0, j = 0;

    if ( (x>3) && (z<10) ) {
        k = x*y - 1;    //语句块1
        j = sqrt(k);
    }

    if( (x==4) || (y>5) ) {
        j = x*y + 10;    //语句块2
    }

    j=j%3;                //语句块3
}
```

测试用例：x=4, y=5, z=5



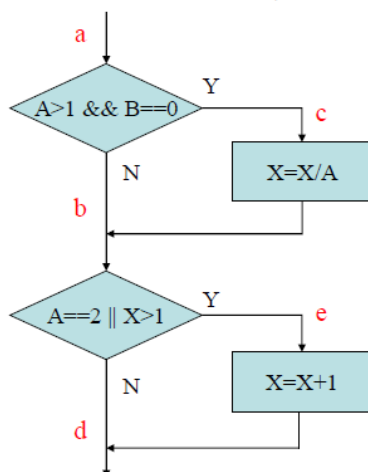
判定覆盖

- 比“语句覆盖”稍强的覆盖标准是“判定覆盖” (DecisionCoverage)标准，其含义是：
执行足够的测试用例，使得程序中的每一个分支至少都通过一次。

- 例1

- 测试用例：

- — A=3, B=0, X=1 (沿路径acd执行)
- — A=2, B=1, X=3 (沿路径abe执行)



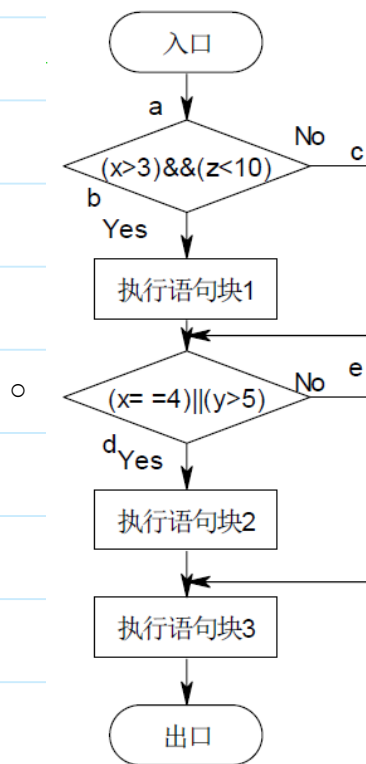
- 例2:

- 测试用例：

- — x=4, y=5, z=5 (沿路径abd)
- — x=2, y=5, z=5 (沿路径ace)

- 两个测试用例虽然能够满足判定覆盖的要求，但是也不能对判断条件进行全面检查

例如把第二个条件 $y>5$ 错误的写成 $y<5$ 、上面的测试用例同样满足了判定覆盖。



条件覆盖

- 一个判定中往往包含了若干个条件；
 - 例如：判定 $(A>1)\&\&(B==0)$ 包含了两个条件： $A>1$ 、 $B==0$
- 引进一个更强的覆盖标准——“条件覆盖” (ConditionCoverage)：执行足够的测试用例，使得判定中的每个条件获得各种可能的结果。
- 例子1:

- 该程序有四个条件：

$A>1$ 、 $B==0$ 、 $A==2$ 、 $X>1$

- 为了达到“条件覆盖”标准，需要执行足够的测试用例，使得在a点有：

$A>1$ 、 $A\leq 1$ 、 $B==0$ 、 $B!=0$

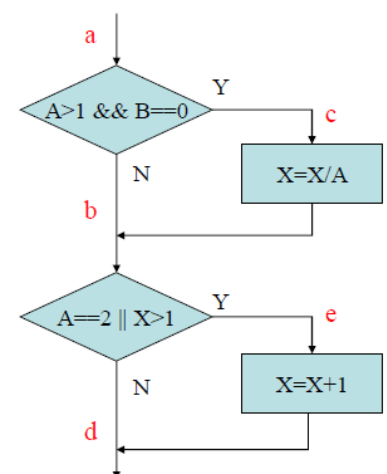
等各种结果出现，以及在b点有：

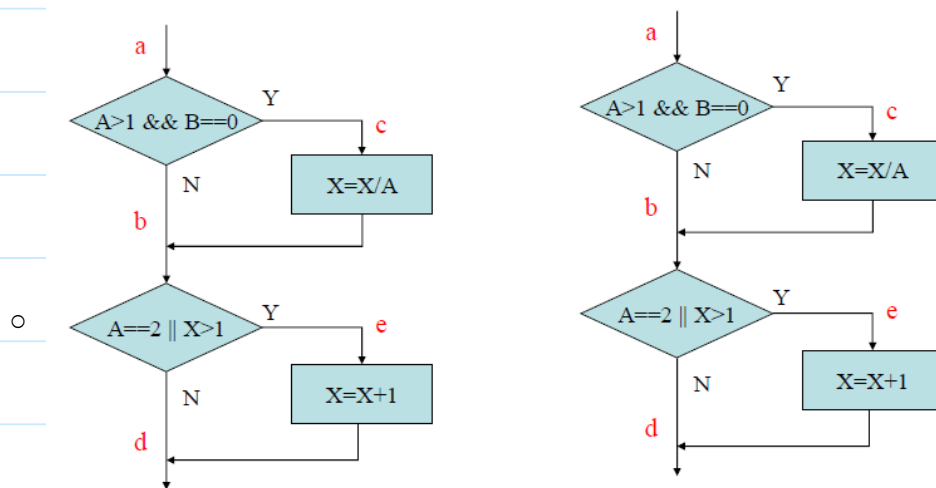
$A==2$ 、 $A!=2$ 、 $X>1$ 、 $X\leq 1$

等各种结果出现。

- 只需设计以下两个测试用例就可满足这一标准：

- $A=2$ ， $B=0$ ， $X=4$ (沿路径ace执行)；
- $A=1$ ， $B=1$ ， $X=1$ (沿路径abd执行)。

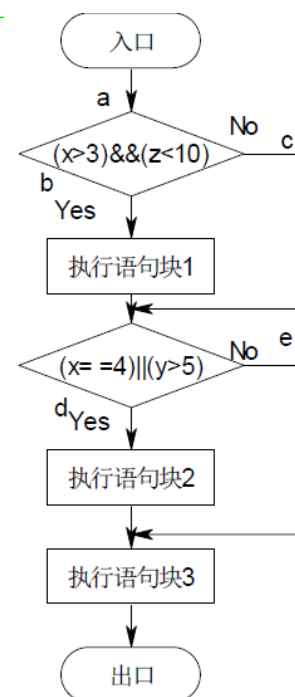




A=2, B=0, X=4 (沿路径ace执行)
A=1, B=1, X=1 (沿路径abd执行)

• 条件覆盖第二个例子

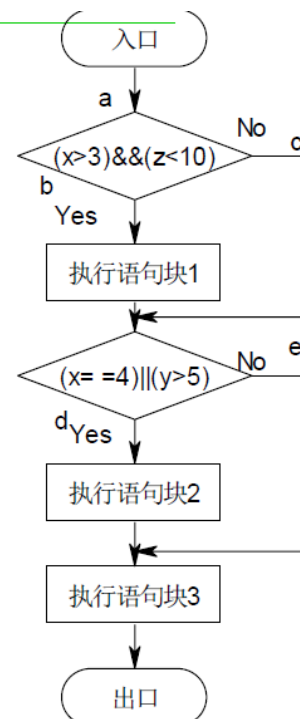
- 生成条件覆盖测试用例的方法：对所有条件的可能取值加以标记。
- 对于第一个判定语句：
 - 条件 $x>3$ 取真值为T1，取假值为-T1
 - 条件 $z<10$ 取真值为T2，取假值为-T2
- 对于第二个判定语句：
 - 条件 $x==4$ 取真值为T3，取假值为-T3
 - 条件 $y>5$ 取真值为T4，取假值为-T4



设计测试用例

测试用例	通过路径	条件取值	覆盖分支
x=4、y=6、z=5	abd	T1、T2、T3、T4	bd
x=2、y=5、z=5	ace	-T1、T2、-T3、-T4	ce
x=4、y=5、z=15	acd	T1、-T2、T3、-T4	cd

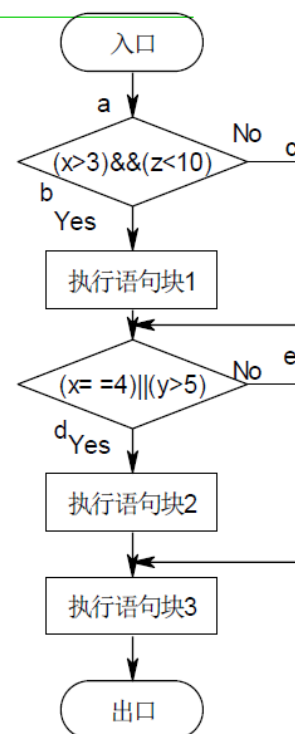
上面的测试用例不但覆盖了所有分支的真假两个分支，而且覆盖了判断中的所有条件的可能值。



- “条件覆盖”通常比“判定覆盖”强，因为它使一个判定中的每一个条件都取到了两个不同的结果，而判定覆盖则不保证这一点。
- 但在某些时候，“条件覆盖”并不能完全包含“判定覆盖”。
- “条件覆盖”也可能不包含“语句覆盖”。
 - 例如：对语句“if(A&&B)S”设计测试用例使其满足条件覆盖，即“使A为真并使B为假”、“使A为假而且B为真”，但是它们都未能使语句S得以执行。

- 对右图设计下面的测试用例，虽然满足条件覆盖，但只覆盖了第一个条件的取假分支和第二个条件的取真分支，不满足判定覆盖的要求。

测试用例	路径	条件取值	覆盖分支
x=2、y=6、z=5	acd	-T1、T2、-T3、T4	acd
x=4、y=5、z=15	acd	T1、-T2、T3、-T4	acd



判定/条件覆盖

- 针对上面的问题引出了另一种覆盖标准——“判定/条件覆盖” (Decision/Condition Coverage):
 - 执行足够的测试用例，使得判定中每个条件取到各种可能的值，并使每个分支取到各种可能的

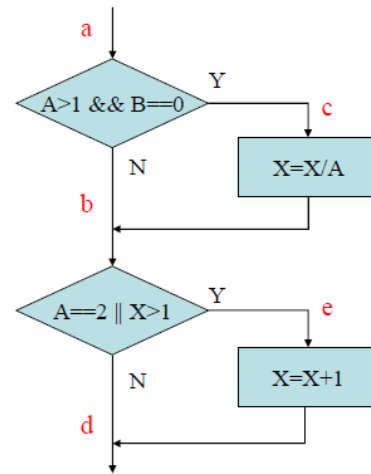
结果。

- 对右图的程序，以下两个测试用例

A=2, B=0, X=4 (沿ace路)

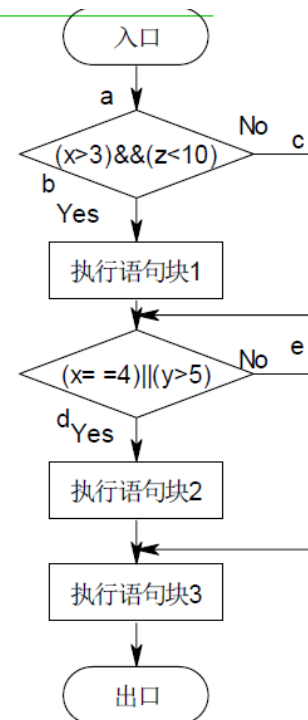
A=1, B=1, X=1 (沿abd路径)

- 是满足这一标准的。



- 设计两个测试用例便可以覆盖8个条件值以及4个判断分支。

测试用例	路径	条件取值	覆盖分支
x=4、y=6、z=5	abd	T1、T2、T3、T4	abd
x=2、y=5、z=11	ace	-T1、-T2、-T3、-T4	ace



- 判定/条件覆盖从表面来看，它测试了所有条件的取值，但是有可能某些条件掩盖了另一些条件。
- 例如：
 - 对于条件表达式 $(x>3)\&\&(z<10)$ 来说，必须两个条件都满足才能确定表达式为真。如果 $x>3$ 为假，则编译器一般不再判断 $z<10$ 是否满足了。
 - 对于表达式 $(x==4)\|\|(y>5)$ 来说，若 $x==4$ 测试结果为真，就认为表达式的结果为真，这时不再检查 $y>5$ 条件了。
- 因此，采用判定/条件覆盖，逻辑表达式中的错误不一定能够查出来。

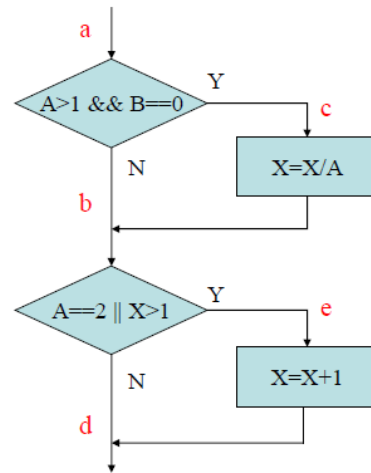
条件组合覆盖

- 针对上述问题又提出了另一种标准—“条件组合覆盖” (MultipleConditionCoverage):
 - 执行足够的测试用例，使得每个判定中多个条件的各种取值的可能组合都至少出现一次。
- 显然，满足“条件组合覆盖”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”。

- 针对该程序，需要设计测试用例使下面 8 种条件组合都能够出现：

- 1) $A > 1, B == 0$ 2) $A > 1, B != 0$
- 3) $A <= 1, B == 0$ 4) $A <= 1, B != 0$
- 5) $A == 2, X > 1$ 6) $A == 2, X <= 1$
- 7) $A != 2, X > 1$ 8) $A != 2, X <= 1$

- 5-8 四种情况是第二个 if 语句的条件组合，而 X 的值在该语句之前是要经过计算的，所以还必须根据程序的逻辑推算出在程序的入口点 X 的输入值应是什么。



- 下面四个测试用例可以使上述 8 种条件组合分别至少出现一次：

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

使 1)、5) 两种情况出现；

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

使 2)、6) 两种情况出现；

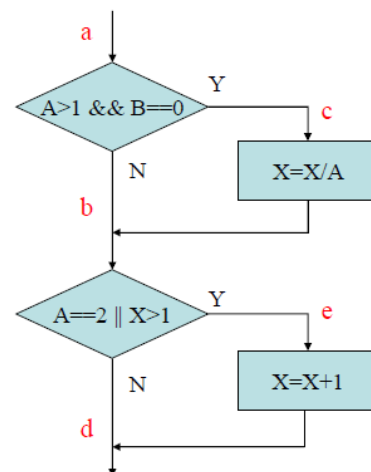
$A=1, B=0, X=2$

使 3)、7) 两种情况出现；

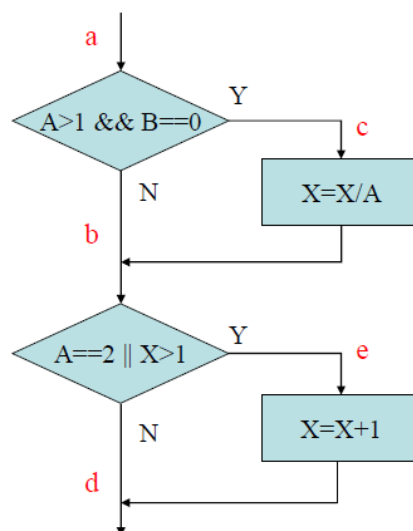
$A=1, B=1, X=1$

使 4)、8) 两种情况出现。

- | | |
|---------------------|---------------------|
| 1) $A > 1, B == 0$ | 2) $A > 1, B != 0$ |
| 3) $A <= 1, B == 0$ | 4) $A <= 1, B != 0$ |
| 5) $A == 2, X > 1$ | 6) $A == 2, X <= 1$ |
| 7) $A != 2, X > 1$ | 8) $A != 2, X <= 1$ |

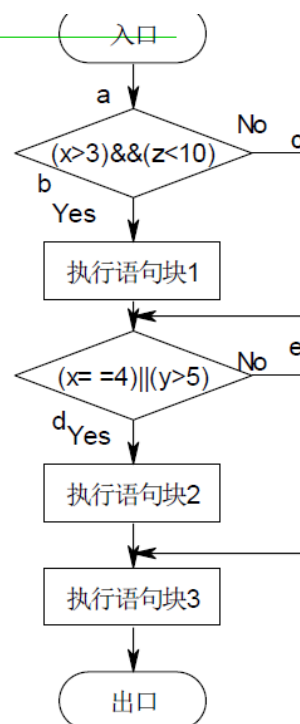


上面四个例子虽然满足条件组合覆盖，但并不能覆盖程序中的每一条路径，例如路径acd就没有执行，因此，条件组合覆盖标准仍然是不彻底。



对右图中各个判定的条件取值组合加以标记如下：

- 1) $x > 3, z < 10$ 记做T1 T2, 第一个判定的Yes分支
- 2) $x > 3, z \geq 10$ 记做T1 -T2, 第一个判定的No分支
- 3) $x \leq 3, z < 10$ 记做-T1 T2, 第一个判定的No分支
- 4) $x \leq 3, z \geq 10$ 记做-T1 -T2, 第一个判定的No分支
- 5) $x = 4, y > 5$ 记做T3 T4, 第二个判定的Yes分支
- 6) $x = 4, y \leq 5$ 记做T3 -T4, 第二个判定的Yes分支
- 7) $x \neq 4, y > 5$ 记做-T3 T4, 第二个判定的Yes分支
- 8) $x \neq 4, y \leq 5$ 记做-T3 -T4, 第二个判定的No分支

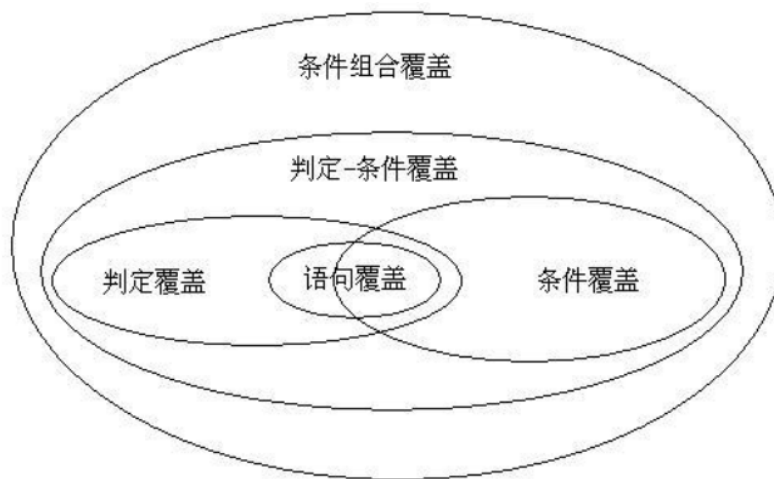


- 根据定义取4个测试用例，就可以覆盖上面8种条件取值的组合。
- 测试用例：

测试用例	通过路径	条件取值	覆盖组合号
x=4、y=6、z=5	abd	T1、T2、T3、T4	1和5
x=4、y=5、z=15	acd	T1、-T2、T3、-T4	2和6
x=2、y=6、z=5	acd	-T1、T2、-T3、T4	3和7
x=2、y=5、z=15	ace	-T1、-T2、-T3、-T4	4和8

- 上面的测试用例覆盖了所有条件的可能取值的组合，覆盖了所有判断的可取分支，但是却丢失了一条路径abe。

五种覆盖标准之间的关系

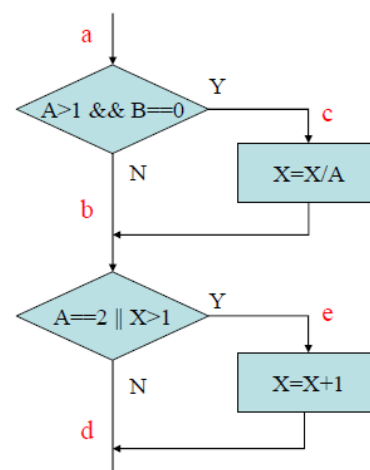


3 基本路径法

路径测试

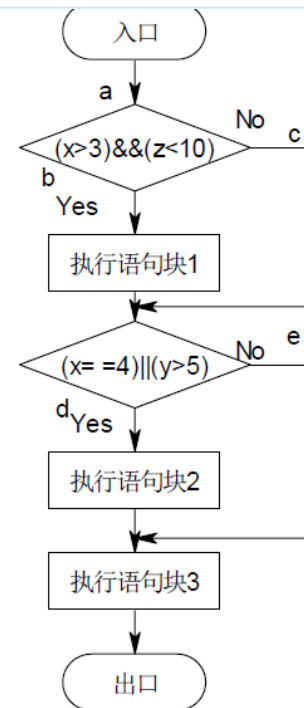
- **路径测试**：设计足够多的测试用例，覆盖被测试对象中的所有可能路径。
- 下面的测试用例则可对程序进行全部的路径覆盖。

测试用例	通过路径
A=2、B=0、X=3	ace
A=1、B=0、X=1	abd
A=2、B=1、X=1	abe
A=3、B=0、X=1	acd



- 下面的测试用例则可对程序进行全部的路径覆盖。

测试用例	通过路径	覆盖条件
x=4、y=6、z=5	abd	T1、T2、T3、T4
x=4、y=5、z=15	acd	T1、-T2、T3、-T4
x=2、y=5、z=15	ace	-T1、-T2、-T3、T4
x=5、y=5、z=5	abe	T1、T2、-T3、-T4

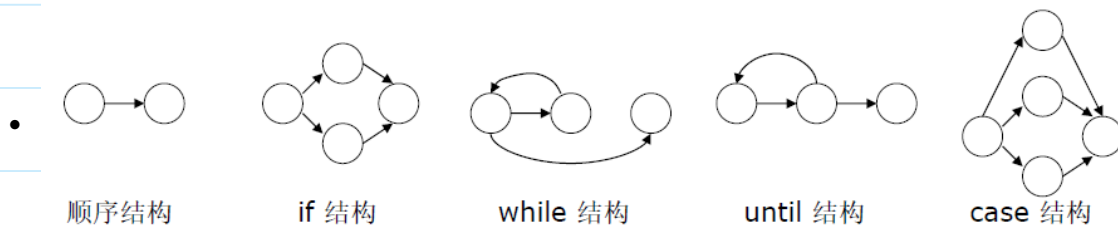


基本路径测试

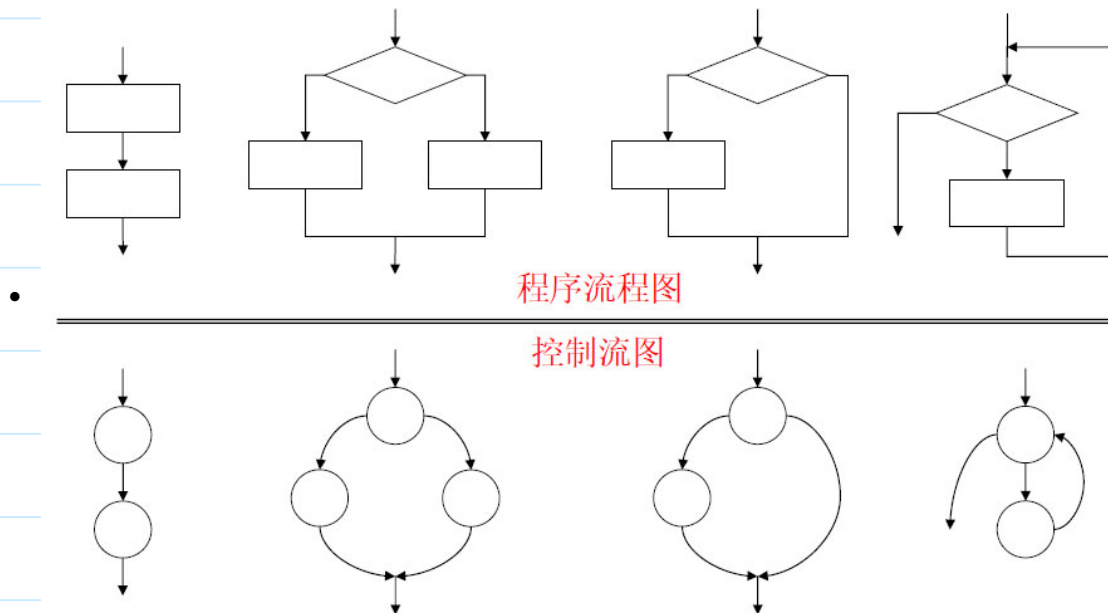
- 在实际中，即使一个不太复杂的程序，其路径都是一个庞大的数字，要在测试中覆盖所有的路径是不现实的。为了解决这一难题，只得把覆盖的路径数压缩到一定限度内，例如，程序中的循环体只执行一次
- 基本路径测试(Basicpathtest):
 - 在程序控制图的基础上，通过分析控制构造的环行复杂性，导出基本可执行路径集合，从而设计测试用例。
 - 设计出的测试用例要保证在测试中程序的每一个基本独立路径至少执行一次。
- 在程序控制流图的基础上，通过分析控制构造的环路复杂性，导出基本可执行路径集合，从而设计测试用例。包括以下4个步骤和一个工具方法：
 - 程序的控制流图：描述程序控制流的一种图示方法。
 - 计算程序圈复杂度：从程序的环路复杂性可导出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执行一次所必须的测试用例数目的上界。
 - 导出测试用例：根据环复杂度和程序结构设计用例数据输入和预期结果。
 - 准备测试用例：确保基本路径集中的每一条路径的执行。

控制流图的符号只有2种

- 图中的每一个圆称为流图的结点，代表一条或多条语句。
- 流图中的箭头称为边或连接，代表控制流。



程序流程图-控制流图



控制流图

- 如果判断中的条件表达式是由一个或多个逻辑运算符(OR,AND,NAND,NOR)连接的复合条件表达式，则需要改为一系列只有单条件的嵌套的判断。
- 例如：

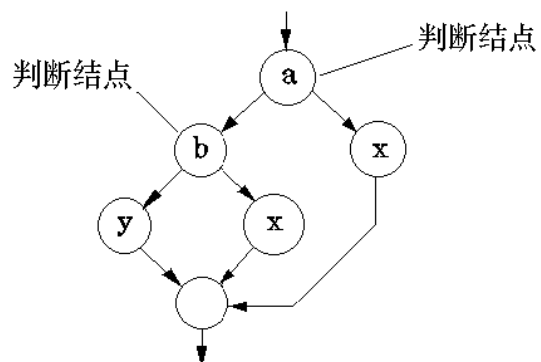
```

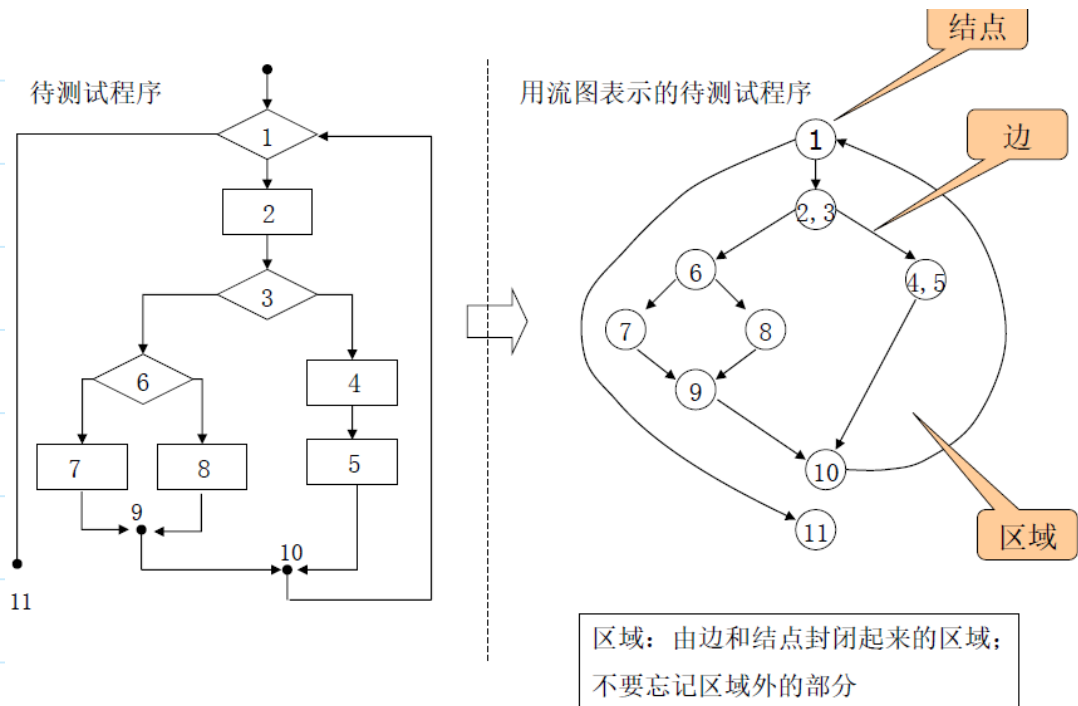
1 if a or b
2   x
3 else
4   y

```

对应的逻辑为：

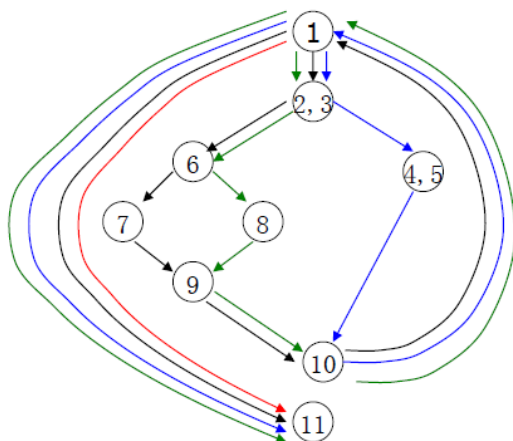
○





独立路径

- 独立路径(基本路径)：一条程序执行的路径，至少包含一条在定义该路径之前的其他基本路径中所不曾用过的边(即：至少引入程序的一个新处理语句集合或一个新条件)
- 注：独立路径的基本集合是不唯一的，可能存在多个。



路径1：1-11

路径2：1-2-3-4-5-10-1-11

路径3：1-2-3-6-8-9-10-1-11

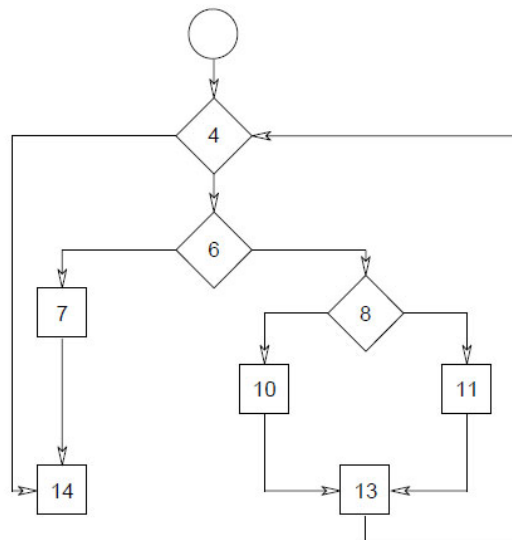
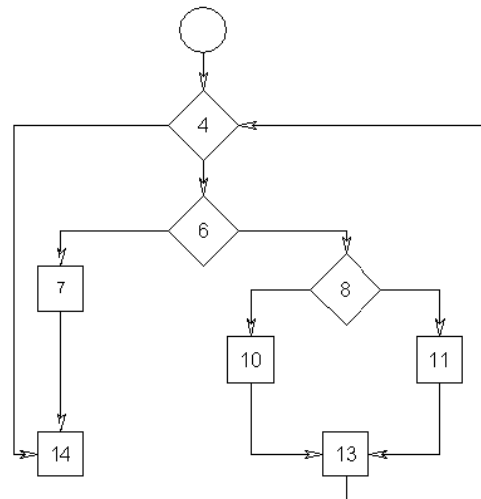
路径4：1-2-3-6-7-9-10-1-11

对以上路径的遍历，就是至少一次地执行了程序中的所有语句。

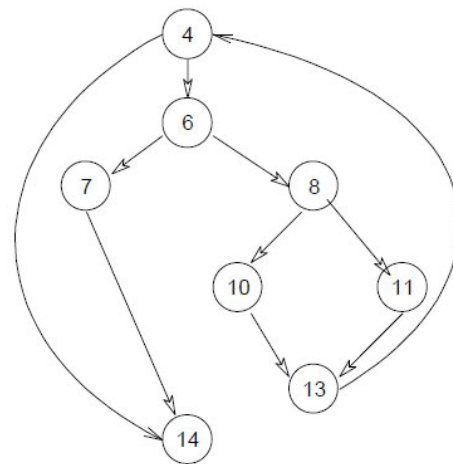
第一步：画出控制流图

```
void Sort(int iRecordNum,int iType)
```

```
1. {
2.   int x=0;
3.   int y=0;
4.   while (iRecordNum-- > 0)
5.   {
6.     if(0==iType)
7.     { x=y+2; break;}
8.     else
9.       if (1==iType)
10.        x=y+10;
11.      else
12.        x=y+20;
13.   }
14. }
```



程序流程图



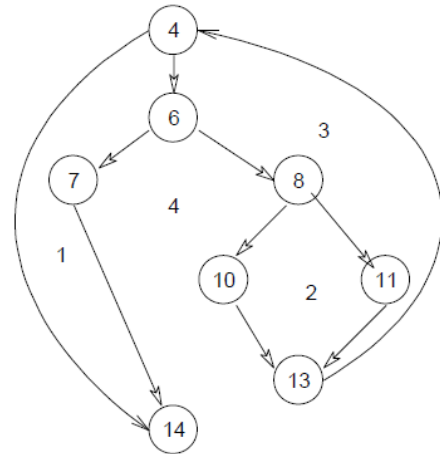
控制流图

第二步：计算圈复杂度

- 圈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，将该度量用于计算程序的基本的独立路径数目，为确保所有语句至少执行一次的测试数量的上界。
- 独立路径必须包含一条在定义之前不曾用到的边。
- 有以下三种方法计算圈复杂度：
 - 流图中区域的数量；
 - 给定流图G的圈复杂度V(G)，定义为 $V(G)=E-N+2$ ，E是流图中边的数量，N是流图中结点的数量；
 - 给定流图G的圈复杂度V(G)，定义为 $V(G)=P+1$ ，P是流图G中判定结点的数量。
- 对应上图中的圈复杂度，计算如下：
 - 流图中有4个区域：

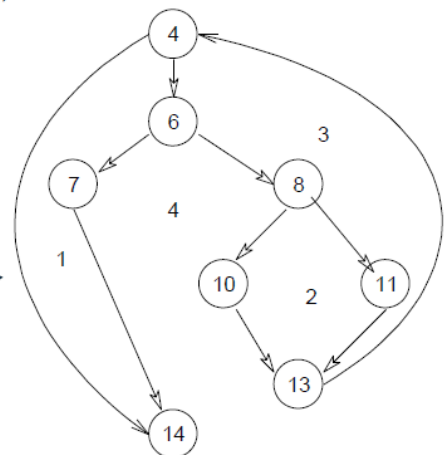
■ 对应上图中的圈复杂度，计算如下：

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



第三步：导出测试用例

- 根据上面的计算方法，可得出四个独立的路径。(一条独立路径是指，和其他的独立路径相比，至少引入一个新处理语句或一个新判断的程序通路。 $V(G)$ 值正好等于该程序的独立路径的条数。)
- 路径1: 4-14
- 路径2: 4-6-7-14
- 路径3: 4-6-8-10-13-4-14
- 路径4: 4-6-8-11-13-4-14
- 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四条路径。



第四步：准备测试用例

- 为了确保基本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某一条路径可以被测试到。
- 测试用例={测试数据+期望结果}
 - 测试数据是由路径和程序推论出来的；
 - 预期结果是从函数说明中导出，不能根据程序结构中导出！

路径1: 4-14

输入数据: iRecordNum=0, 或者取
iRecordNum<0的某一个值

预期结果: x=0

路径2: 4-6-7-14

输入数据: iRecordNum=1,iType=0

预期结果: x=2

路径3: 4-6-8-10-13-4-14

输入数据: iRecordNum=1,iType=1

预期结果: x=10

路径4: 4-6-8-11-13-4-14

输入数据: iRecordNum=1,iType=2

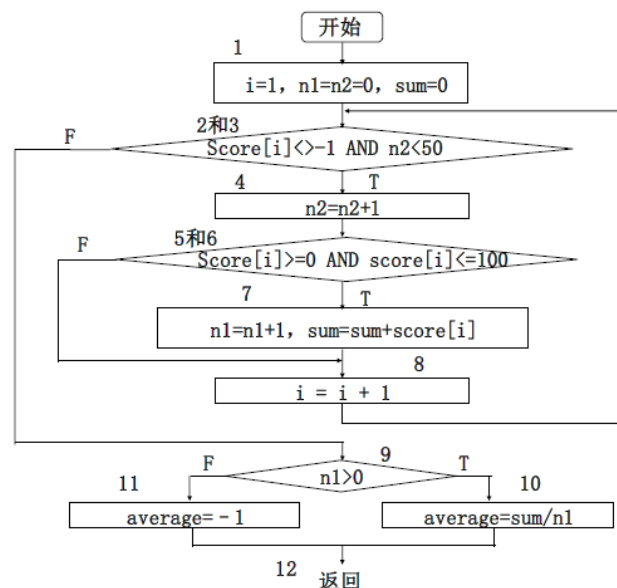
预期结果: x=20

```
void Sort(int iRecordNum,int iType)
```

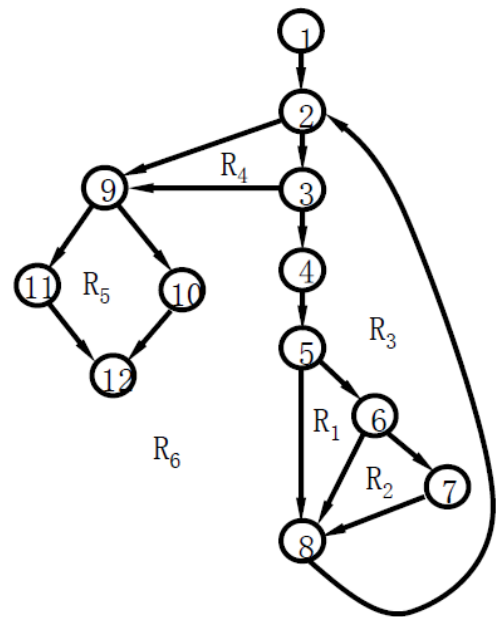
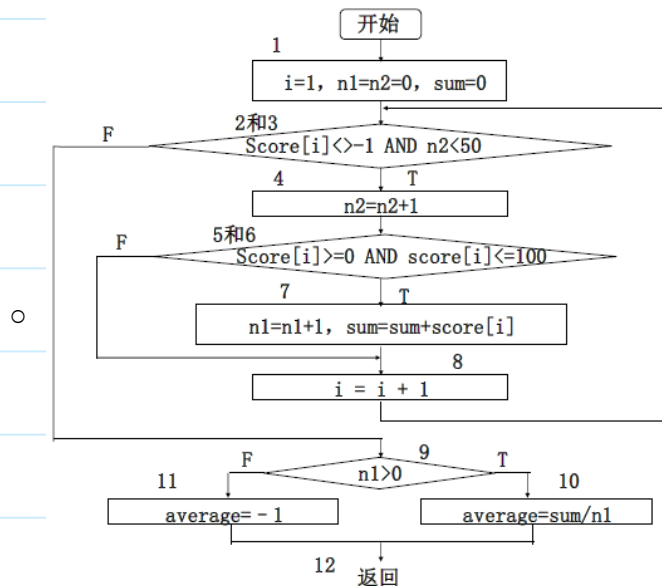
```
1. {  
2.   int x=0;  
3.   int y=0;  
4.   while (iRecordNum-- > 0)  
5.   {  
6.     if(0==iType)  
7.       {x=y+2; break;}  
8.     else  
9.       if(1==iType)  
10.        x=y+10;  
11.      else  
12.        x=y+20;  
13.    }  
14. }
```

示例1: 学生成绩统计

- 下例程序流程图描述了最多输入50个值(以-1作为输入结束标志), 计算其中有效的学生分数的个数、总分数和平均值。



- 第一步: 画出控制流图



- 第二步: 计算图复杂度

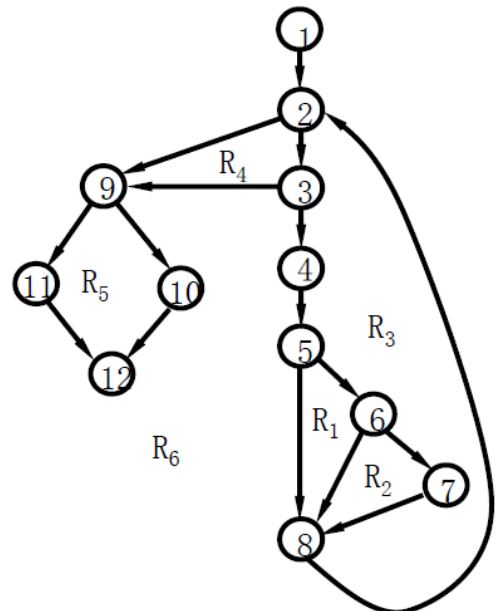
$$V(G) = 6 \text{ (个区域)}$$

$$V(G) = E - N + 2 = 16 - 12 + 2 = 6$$

其中E为流图中的边数，N为结点数；

○ $V(G) = P + 1 = 5 + 1 = 6$

其中P为谓词结点的个数。在流图中，结点2、3、5、6、9是谓词结点。



- 第三步: 确定基本路径集合

路径1: 1-2-9-10-12

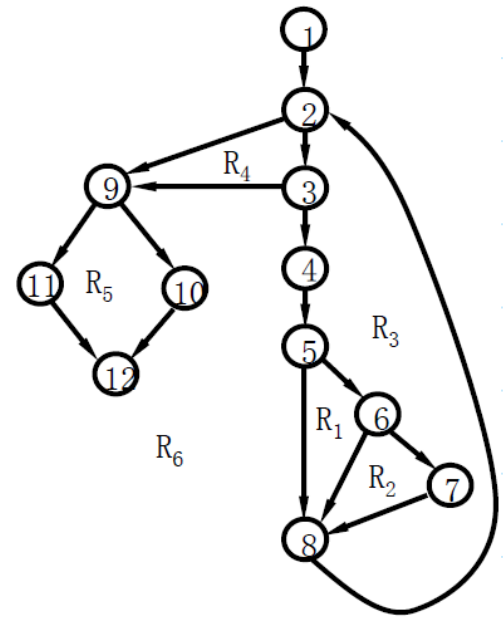
路径2: 1-2-9-11-12

路径3: 1-2-3-9-10-12

路径4: 1-2-3-4-5-8-2...

路径5: 1-2-3-4-5-6-8-2...

○ 路径6: 1-2-3-4-5-6-7-8-2...



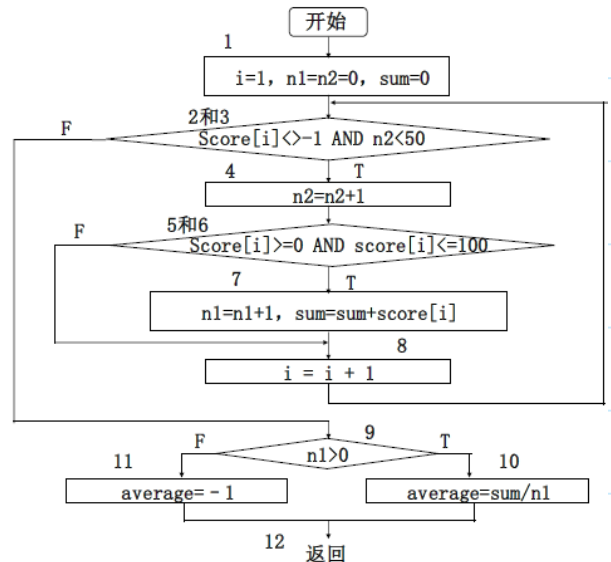
• 第四步：导出测试用例

▪ 路径1(1-2-9-10-12)的测试用例：

score[k]=有效分数值，当 $k < i$ ；

score[i]=-1, $2 \leq i \leq 50$ ；

▪ 期望结果：根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average。



▪ 路径2(1-2-...-2-9-11-12)的测试用例：

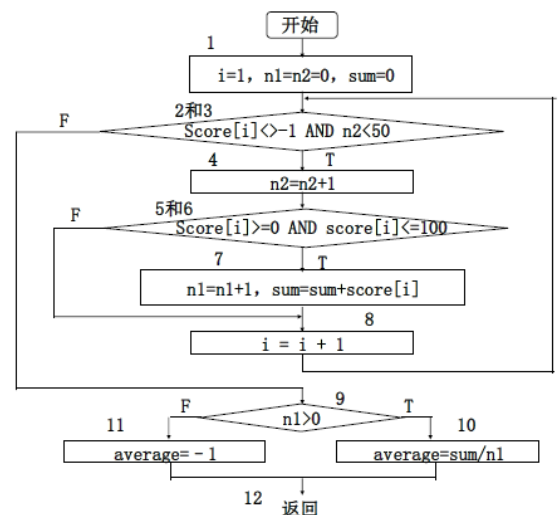
score[1] = -1；

期望的结果：average = -1，其他量保持初值。

○ ▪ 路径3(1-2-...-2-3-9-10-12)测试用例：

输入多于50个有效分数，即试图处理51个分数，要求前50个为有效分数；

期望结果：n1=50、且算出正确的总分和平均分。



- 路径4(1-2-3-4-5-8-2...)的测试用例:

score[i]=有效分数, 当 $i < 50$;

score[k]<0, $i < k$;

期望结果: 根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average。

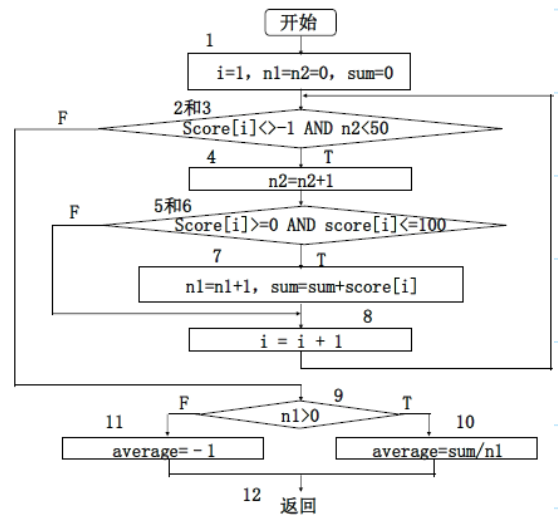
○

- 路径5(1-2-3-4-5-6-8-2...)的测试用例:

score[i]=有效分数, 当 $i < 50$;

score[k]>100, $i < k$;

期望结果: 根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average。

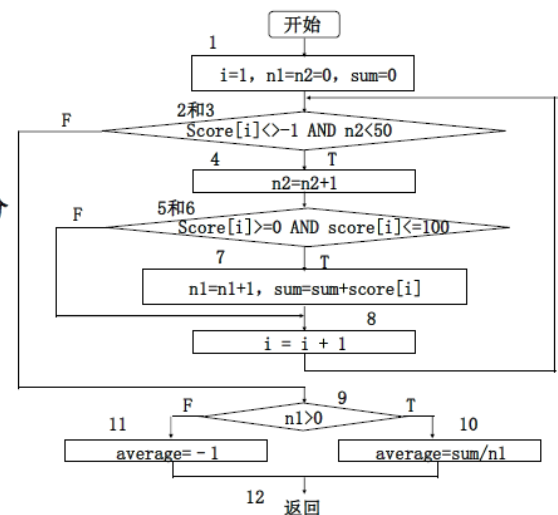


- 路径6(1-2-3-4-5-6-7-8-2...)的测试用例:

score[i]=有效分数, 当 $i < 50$;

期望结果: 根据输入的有效分数算出正确的分数个数n1、总分sum和平均分average。

○



示例: 使用二分搜索法进行数组排序

某算法的程序伪代码如下所示，它完成的基本功能是：

- 输入一个自小到大顺序排列的整型数组elemArray和一个整数key，算法通过二分搜索法查询key是否在elemArray中出现；
- 若找到，则在index中记录key在elemArray中出现的位置；
- 若找不到，则为index赋值-1。算法将index作为返回值。

```

int search (int key, int [] elemArray)
{
1   int bottom = 0;
2   int top = elemArray.length - 1;
3   int mid = 0;
4   int index = -1;
5   while (bottom <= top)
   {
6       mid = (top + bottom) / 2;
7       if (elemArray [mid] == key)
       {
8           index = mid;
9           break;
       }
       else
       {
10          if (elemArray [mid] < key)
11              bottom = mid + 1;
12          else
13              top = mid - 1;
       }
   }
   return index;
}

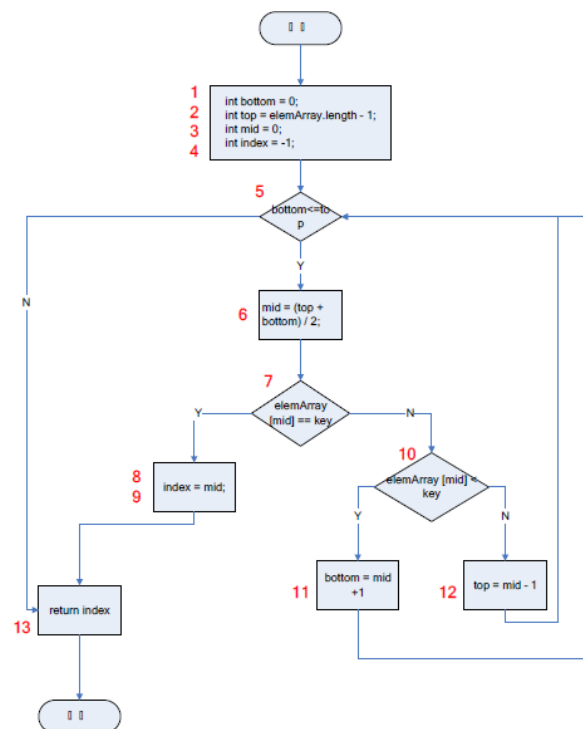
```

模块程序流程图

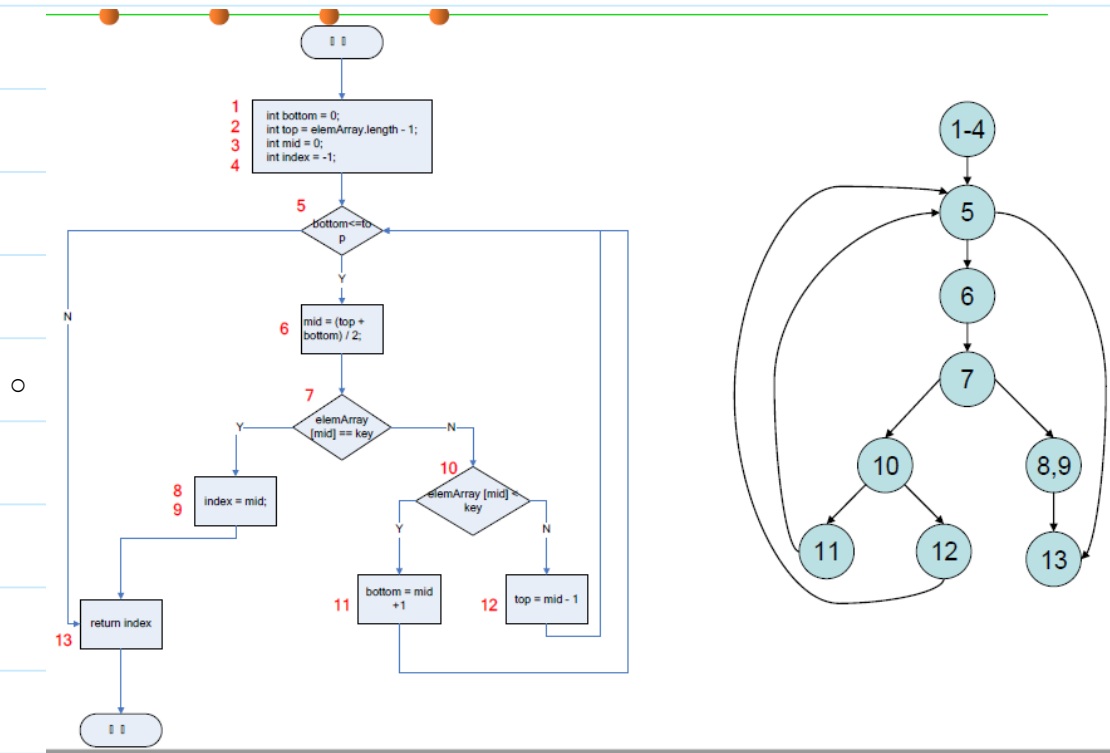
```

int search (int key, int [] elemArray)
{
1   int bottom = 0;
2   int top = elemArray.length - 1;
3   int mid = 0;
4   int index = -1;
5   while (bottom <= top)
   {
6       mid = (top + bottom) / 2;
7       if (elemArray [mid] == key)
       {
8           index = mid;
9           break;
       }
       else
       {
10          if (elemArray [mid] < key)
11              bottom = mid + 1;
12          else
13              top = mid - 1;
       }
   }
   return index;
}

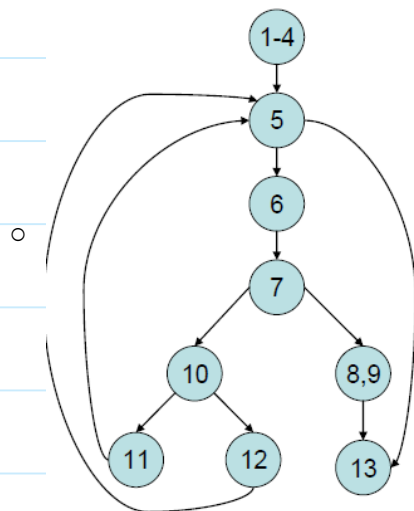
```



控制流图



- V(G)及独立路径

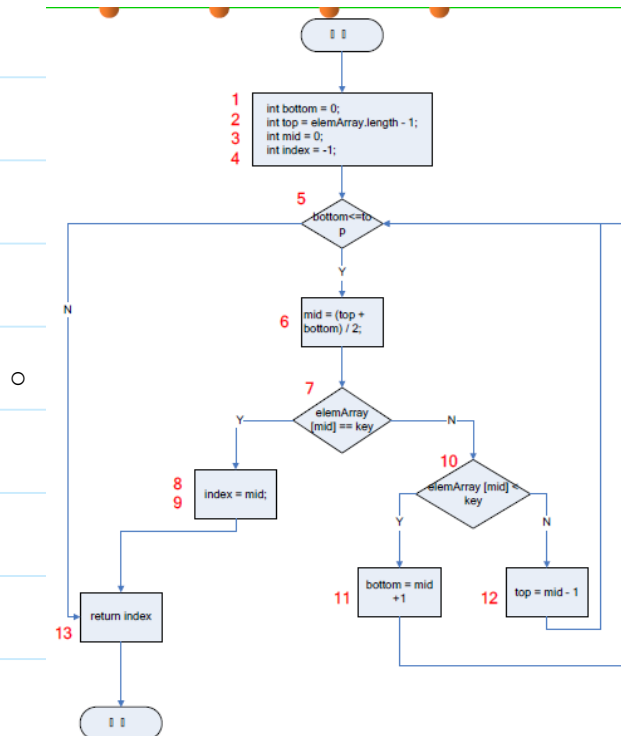


- 测试用例

- $V(G)=4$ 个区域
- $V(G)=3$ 个判断节点+1
- $V(G)=11$ 条边-9个节点+2

- 独立路径:

- 路径1: 1-4, 5, 13
- 路径2: 1-4, 5, 6, 7, 8, 9, 13
- 路径3: 1-4, 5, 6, 7, 10, 11, 5, 13
- 路径4: 1-4, 5, 6, 7, 10, 12, 5, 13



独立路径:

- 路径1: 1-4, 5, 13
- 路径2: 1-4, 5, 6, 7, 8, 9, 13
- 路径3: 1-4, 5, 6, 7, 10, 11, 5, 13
- 路径4: 1-4, 5, 6, 7, 10, 12, 5, 13

测试用例:

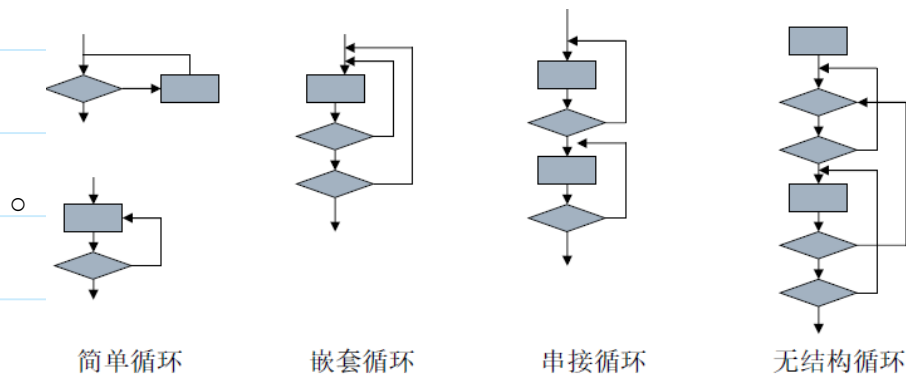
序号	输入: elemArray	输入: Key	期望输出: index
1		20	-1
2	1,2,3,4,5	3	2
3	1,2,3,4,5	4	3
4	1,2,3,4,5	2	1

小结

- 使用路径测试技术设计测试用例的步骤如下:
 - 根据过程设计结果画出相应的流程图
 - 计算流图的环形复杂度
 - 确定现行独立路径的基本集合
 - 设计可强制执行基本集合中每条路径的测试用例
- 注意:
 - 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来进行测试。
 - “圈复杂度”表示: 只要最多V(G)个测试用例就可以达到基本路径覆盖, 但并非一定要设计V(G)个用例。
 - 但是: 测试用例越简化, 代表测试越少、可能发现的错误就越少。

4 循环测试

- 循环测试是一种白盒测试技术, 注重于循环构造的有效性。
- 四种循环: 简单循环, 串接(连锁)循环, 嵌套循环和不规则循环



- 简单循环，测试应包括以下几种，其中的n表示循环允许的最大次数。
 - 零次循环：从循环入口直接跳到循环出口。
 - 一次循环：查找循环初始值方面的错误。
 - 二次循环：检查在多次循环时才能暴露的错误。
 - m次循环：此时的 $m < n$ ，也是检查在多次循环时才能暴露的错误
 - n(最大)次数循环、n+1(比最大次数多一)次的循环、n-1(比最大次数少一)次的循环。
- 嵌套循环
 - 从最内层循环开始，设置所有其他层的循环为最小值；
 - 对最内层循环做简单循环的全部测试。测试时保持所有外层循环的循环变量为最小值。另外，对越界值和非法值做类似的测试。
 - 逐步外推，对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值，所有其它嵌套内层循环的循环变量取“典型”值
 - 反复进行，直到所有各层循环测试完毕。
 - 对全部各层循环同时取最小循环次数，或者同时取最大循环次数。对于后一种测试，由于测试量太大，需人为指定最大循环次数
- 串接循环,要区别两种情况
 - 如果各个循环互相独立，则串接循环可以用与简单循环相同的方法进行测试。
 - 如果有两个循环处于串接状态，而前一个循环的循环变量的值是后一个循环的初值。则这几个循环不是互相独立的，则需要使用测试嵌套循环的办法来处理。
- 非结构循环
 - 对于非结构循环，不能测试，应重新设计循环结构，使之成为其它循环方式，然后再进行测试。

5 xUnit白盒测试

xUnit概述

xUnit——x代表不同的编程语言，xUnit是相应的单元测试工具；

使用JUnit进行Java白盒测试

- JUnit: 由ErichGamma(《设计模式》的作者)和KentBeck(ExtremeProgramming的创始人)编写的一个回归测试框架, 由程序员主导的白盒测试, 在写完模块的代码之后, 马上设计并运行测试用例, 尽快消除错误。
- 使用JUnit的好处
 - 可以使测试代码与产品代码分开。
 - 针对某一个类的测试代码通过较少的改动便可以应用于另一个类的测试(复用)。
 - 易于集成到测试程序的构建过程中, JUnit和Ant/Maven的结合可以实施增量开发。
 - JUnit是开源代码的, 可以进行二次开发。
 - 可以方便地对JUnit进行扩展。
- JUnit的扩展应用
 - JUnit+HttpUnit=WEB功能测试工具
 - JUnit+hansel=代码覆盖测试工具
 - JUnit+abbot=界面自动回放测试工具
 - JUnit+dbunit=数据库测试工具
 - JUnit+junitperf=性能测试工具
- 使用JUnit进行白盒测试
 - JUnit框架继承TestCase类, 用java来编写自动执行、自动验证的测试。这些测试在JUnit中称作“测试用例”。
 - JUnit提供一个机制能够把相关测试用例组合到一起, 称之为“测试套件(testsuite)”。
 - JUnit还提供了一个“运行器”来执行一个测试套件。如果有测试失败了, 这个测试运行器就报告出来; 如果没有失败, 就会显示“OK”。
- JUnit4使用Java5中的注解(annotation), 以下是JUnit4常用的几个annotation:
 - @Before: 初始化方法, 对于每一个测试方法都要执行一次(注意与BeforeClass区别, 后者是对于所有方法只执行一次, 早期版本采用setup())
 - @After: 释放资源, 对于每一个测试方法都要执行一次(注意与AfterClass区别, 后者是对于所有方法只执行一次, 早期版本采用teardown())
 - @Test: 测试方法, 在这里可以测试期望异常和超时时间(早期方法使用“test+被测函数名字”的方法声明)
 - @Test(expected=ArithmeticException)检查被测方法是否抛出ArithmeticException异常
 - @Ignore: 忽略的测试方法
 - @BeforeClass: 针对所有测试, 只执行一次, 且必须为staticvoid
 - @AfterClass: 针对所有测试, 只执行一次, 且必须为staticvoid
- JUnit单元测试步骤
 - a. 准备测试环境:
 - @BeforeClass、@AfterClass: 标识需要在每个测试方法前后需要进行的操作;

- @BeforeClass、@AfterClass：标识需要在测试类前后进行的操作。
 - 一个JUnit4的单元测试用例执行顺序为：
@BeforeClass@Before@Test@After@AfterClass;
 - 每一个测试方法的调用顺序为：@Before@Test@After;
- b. 设计测试方法，用@Test标识测试方法。设计测试用例，在测试方法中使用断言方法如 assertEquals(), assertTrue()等判断执行结果。
- c. 设计测试套件，或使用缺省的测试套件，调用Runner执行测试脚本，生成测试结果
- JUnit举例

```
public class Calculator {  
    private static int result; // 静态变量，用于存储运行结果  
    public void add(int n) {  
        result = result + n;  
    }  
    public void subtract(int n) {  
        result = result - 1; //Bug: 正确的应该是 result =result-n  
    }  
    public void multiply(int n) {  
        // 此方法尚未写好  
    }  
    public void divide(int n) {  
        result = result / n;  
    }  
    public void square(int n) {  
        result = n * n;  
    }  
    public void squareRoot(int n) {  
        for (; ); //Bug : 死循环  
    }  
    public void clear() { // 将结果清零  
        result = 0;  
    }  
    public int getResult() {  
        return result;  
    }  
}
```

```

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;

public class CalculatorTest {

    private static Calculator calculator = new Calculator();

    @Before
    public void setUp() throws Exception {
        calculator.clear();
    }

    @Test
    public void testAdd() {
        calculator.add(2);
        calculator.add(3);
        assertEquals(5, calculator.getResult());
    }

    @Test
    public void testSubtract() {
        calculator.add(10);
        calculator.subtract(2);
        assertEquals(8, calculator.getResult());
    }

    @Ignore("Multiply() Not yet implemented")
    @Test
    public void testMultiply() {
    }

    @Test
    public void testDivide() {
        calculator.add(8);
        calculator.divide(2);
        assertEquals(4, calculator.getResult());
    }
}

```

单元测试类