软件测试方法和技术实践

第1章 单元测试

赵钦佩

qinpeizhao@tongji.edu.cn

http://sse.tongji.edu.cn/zhaoqinpei/

1. 单元测试

- 1.1 我是个很棒的程序员，我的代码就不需要测了吗？
- 1.2 给你一段简单代码，知道如何测吗？
- 1.3 if判断语句再复杂一些，如何测？
- 1.4 “||”写错了怎么都没发现？
- 1.5 可以自动检查代码吗？
- 1.6 面对一个非独立的函数，如何测试？
- 1.7 绕过数据库，如何进行测试？
- 1.8 JUnit是单元测试神器吗？
- 1.9 还有哪些单元测试工具吗？

測试V. S. 打地鼠

单元测试的定义

- 定义

  单元测试（又称为模块测试，Unit Testing）是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。

- 时机

  单元测试和编码是同步进行，但在Test-Driven Development (TDD)中，强调测试在先，编码在后。单元测试一般由开发人员完成，QA人员辅助。

> *"Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead."*
>
> *Martin Fowler*

- 概念

  模块、组件、单元

单元测试的目标

- **目标**：单元模块被正确编码
- 信息能否正确地流入和流出单元
- 在单元工作过程中，其内部数据能否保持其完整性，包括内部数据的形式、内容及相互关系不发生错误，全局变量在单元中的处理和影响
- 为限制数据加工而设置的边界处，能否正确工作
- 单元的运行能否做到满足特定的逻辑覆盖

功能逻辑正确！

单元测试的目标

```
for(i=0; i < len; i++)
{
  point = nPoints[i].copyPoint();
  distance = calculateDistance(startPoint, endPoint, point);
  if(distance > 0)
  {
    pointsList.add(nPoints[i]);
    if(distance > maxDistance)
    {
      maxDistance = distance;
      maxIndex = i;
    }
  }
}
// no left points found
if(pointsList.size() == 0)
{
  maxDistancePoint = null;
  return null;
}

maxDistancePoint = nPoints[maxIndex].copyPoint(); // set the maxPoint
leftPoints = (Point[]) pointsList.toArray(new Point[pointsList.size()]);

return leftPoints;
}

/**
 * Description: execute Quick Hull algorithm on the given data points using
 * start-end line as delimiter and only left side of the line will be analyzed
 * @param nPoints
 * @param startPoint
 * @param endPoint
 * @return
 */
```

关联代码

功能逻辑正确

函数无错

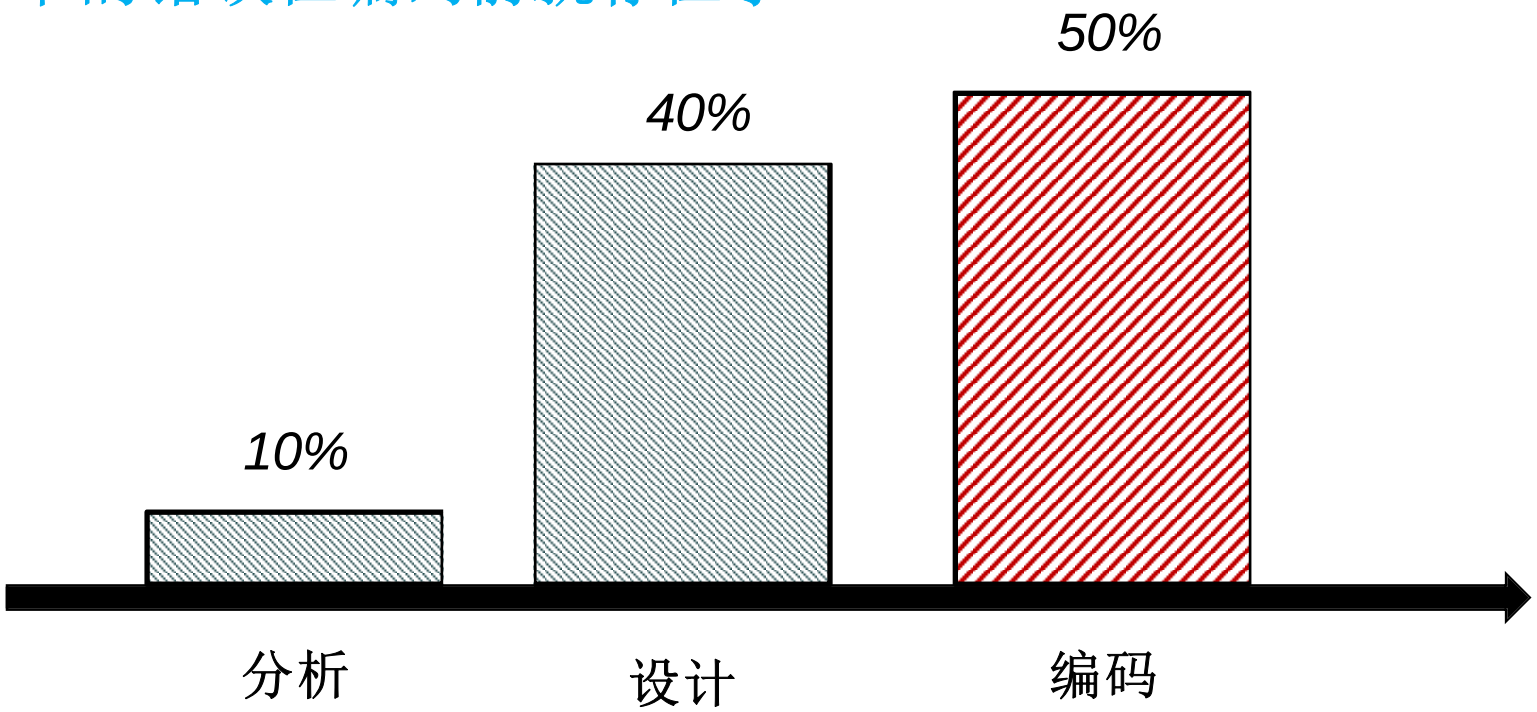
各层判定
处理正确

单元测试的必要性

- 尽早介入越好
  - 错误发现越早, 成本越低
  - 前期发现问题比较容易
  - 前期修正问题更容易

越早介入越好

一半的错误在编码前就存在了！



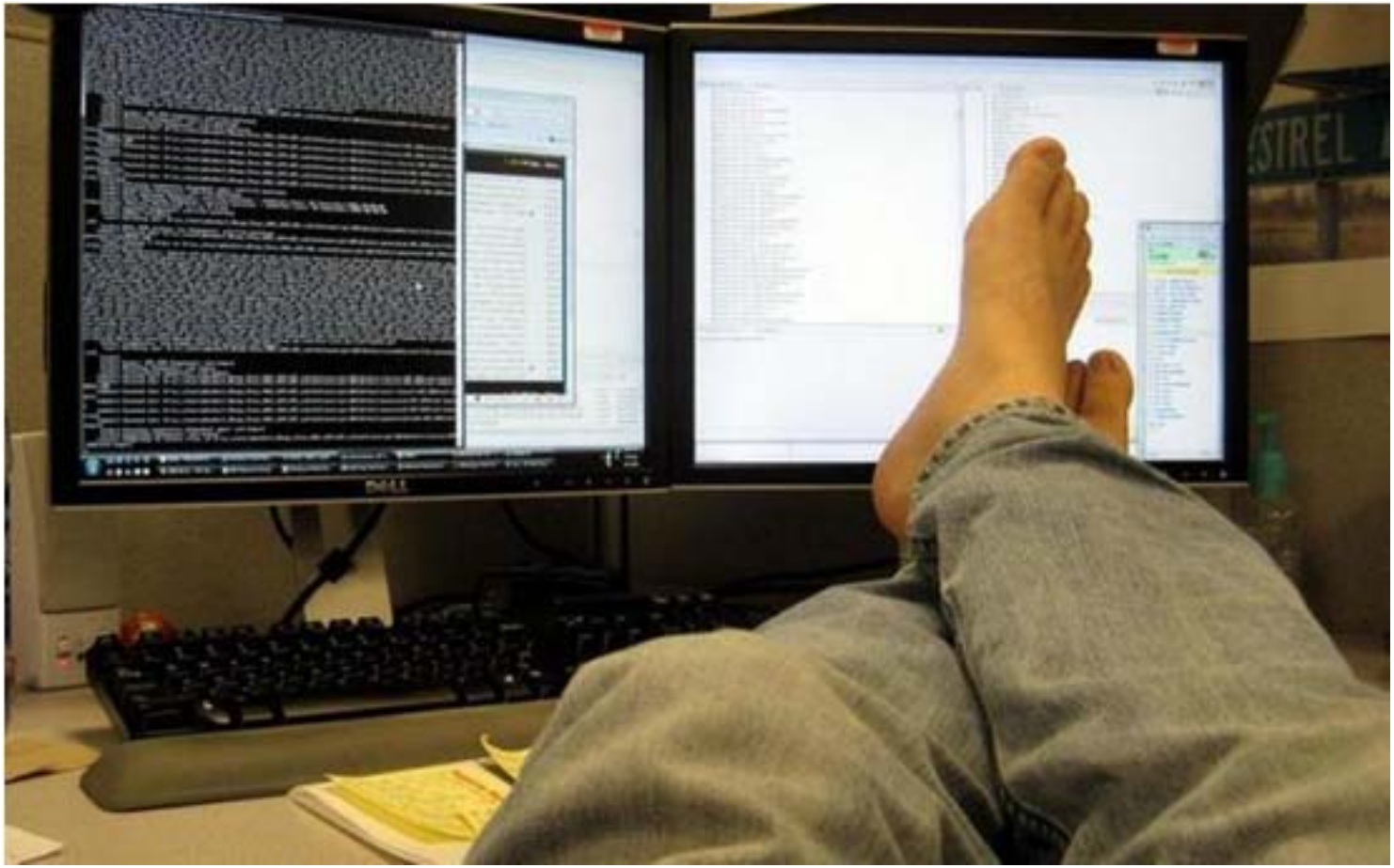
越早介入越好

开发中各个阶段因缺陷产生的成本

来自： http://blog.csdn.net/dellfox/article/details/7018181

1.1 我是一个很棒的程序员?

- 调试v.s. 测试

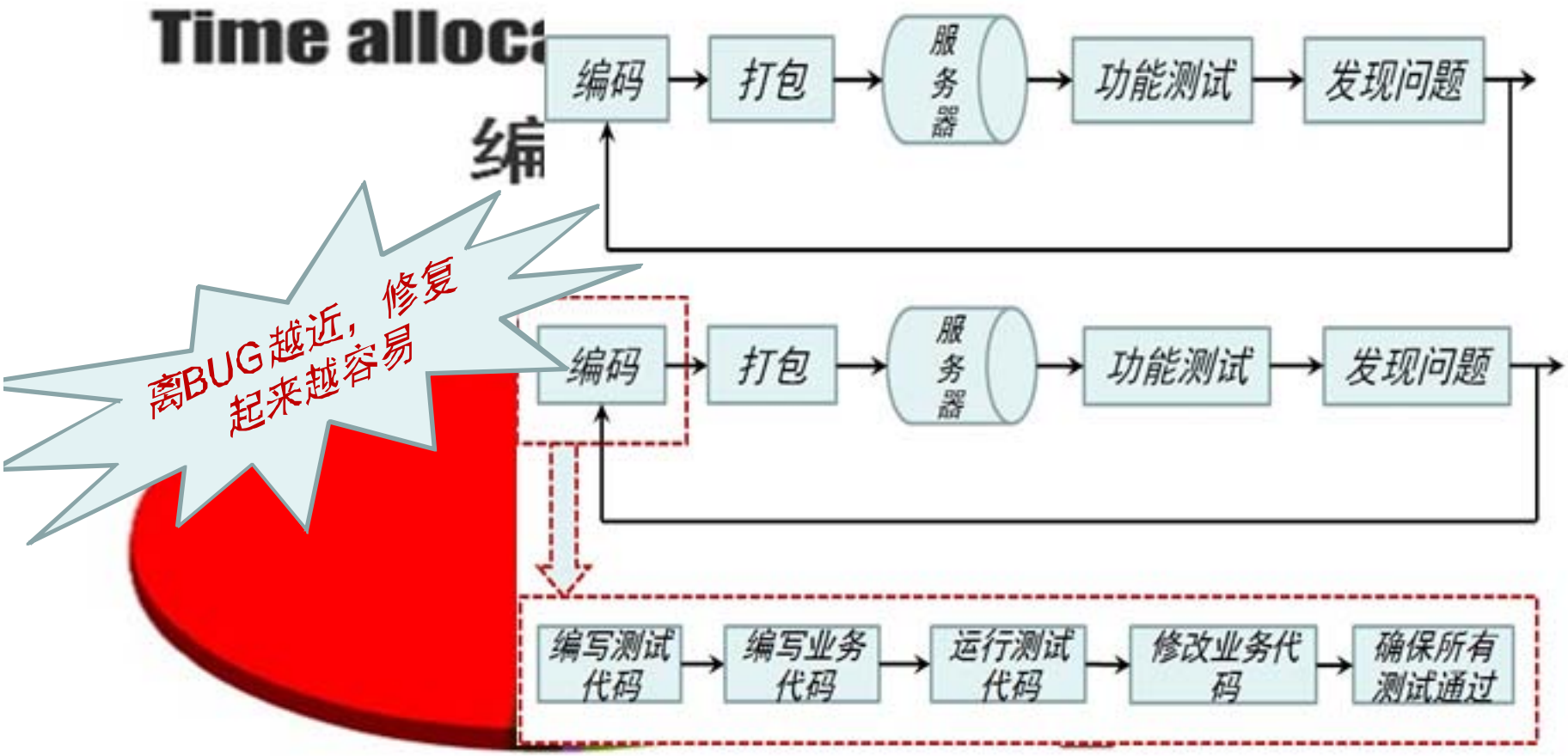


1.1 很棒的程序员

- 编程过程中，每写1000行代码会犯几十个错误
- 编程与编译运行结束后，每1000行代码中大约残留有
- 寻找与修改程序投资的30% -60

1.1 程序员心理

- 编写单元测试会导致不能按时完成编码任务,推迟项目进度
- 单元测试的价值不高，完全是浪费时间



1.1 程序员心理

- 编写单元测试会导致不能按时完成编码任务,推迟项目进度
- 单元测试的价值不高，完全是浪费时间

1.1 程序员心理

- 业务逻辑比较简单，不值得编写单元测试
  - 适应变更，需求在不断变化中（需求文档，注释）
  - 由代码的编写者去写的单元测试要比由其它人去编写的单元测试要更完善,更准确
  - 代码维护（修正BUG，增加功能）
  - 集成测试（print?）

1.1 程序员心理

- 不知道怎么编写单元测试
  - 为所有的类编写单元测试？
  - 学会使用断言（assertion）
  - 最大化测试覆盖率
  - 避免重复的测试代码
  - 不要依赖于测试方法的执行顺序
- 项目没有要求，所以不编写
- 在项目的前期还是尽量去编写单元测试，但是越到项目的后期就越失控

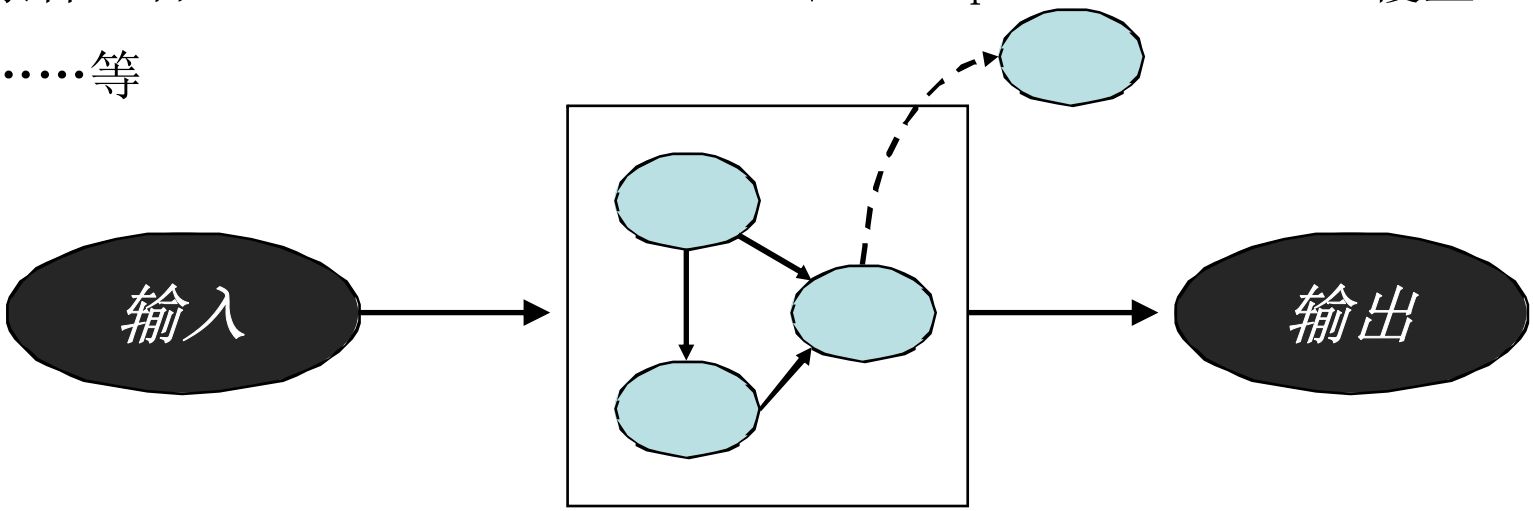
1.2 代码如何测？

- 评价标准：代码覆盖率
- 代码覆盖的步骤：
  - 第一步：源代码转换为流程图。流程图可以比较直观而详细地描述需要覆盖的语句
  - 第二步：分析流程图，根据测试对象的要求选择需要覆盖的代码
  - 第三步：根据第二步结果确定测试数据，生成*测试用例*
- 测试用例：一组测试输入、执行条件以及预期结果

1.2 白盒测试

白盒测试/结构化测试方法（基于代码的测试技术）

- 语句（statement）覆盖
- 分支（branch）覆盖/判定（decision）覆盖
- 条件（condition）覆盖
- 判定（decision）/条件（condition）覆盖
- 条件组合（condition combination/multiple condition）覆盖
- ……等



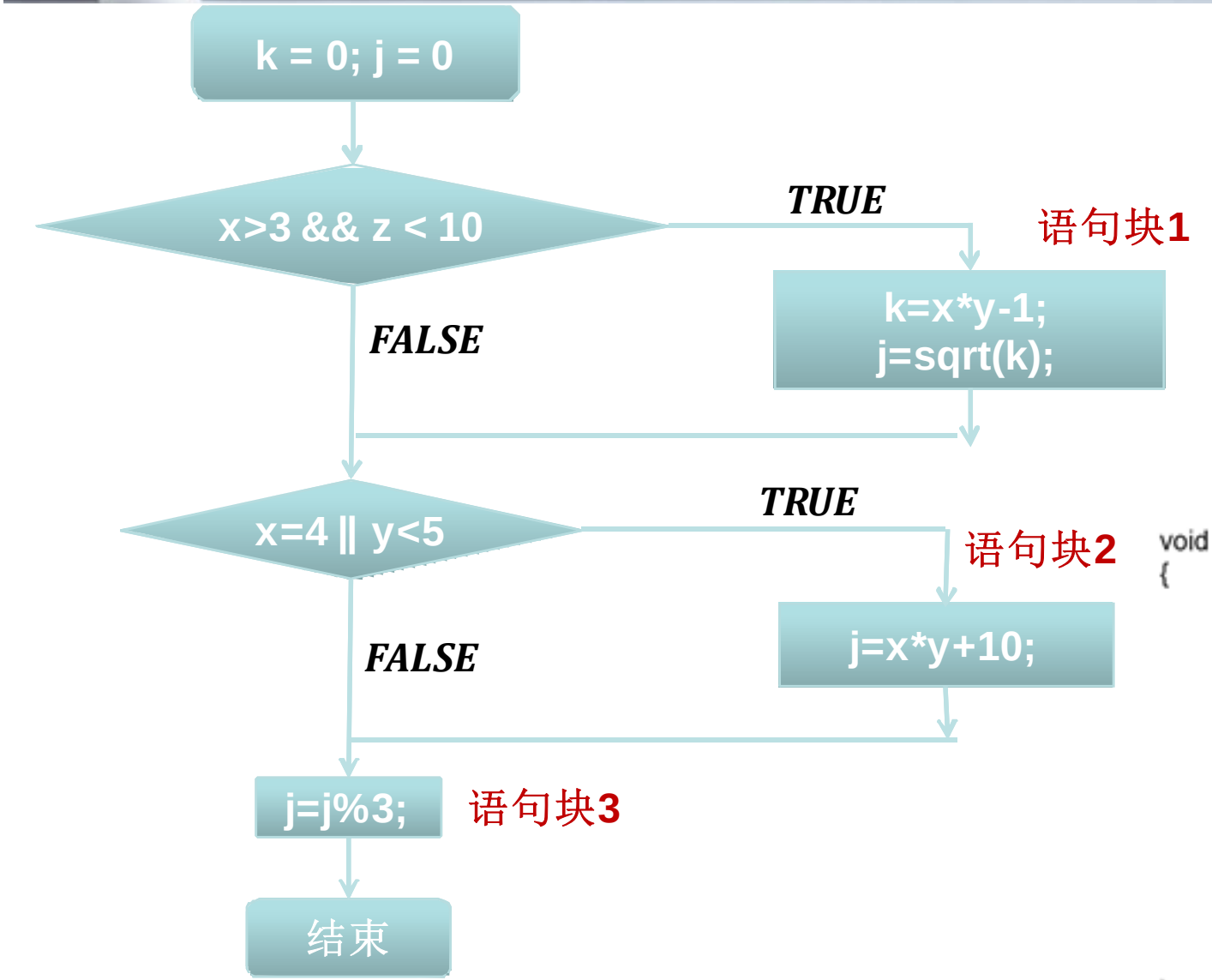
1.3 复杂判断语句如何测？

```
void Code(int x, int y, int z)
{
        int k = 0;
        int j = 0;
        if( (x > 3) && (z < 10) )
        {
                k = x * y - 1;
                j = sqrt(k);  //语句块1
        }
        if( (x==4)||(y > 5) )
        {
                j = x * y + 10;  //语句块2
        }
        j = j % 3;   //语句块3
}
```

判定条件 M1

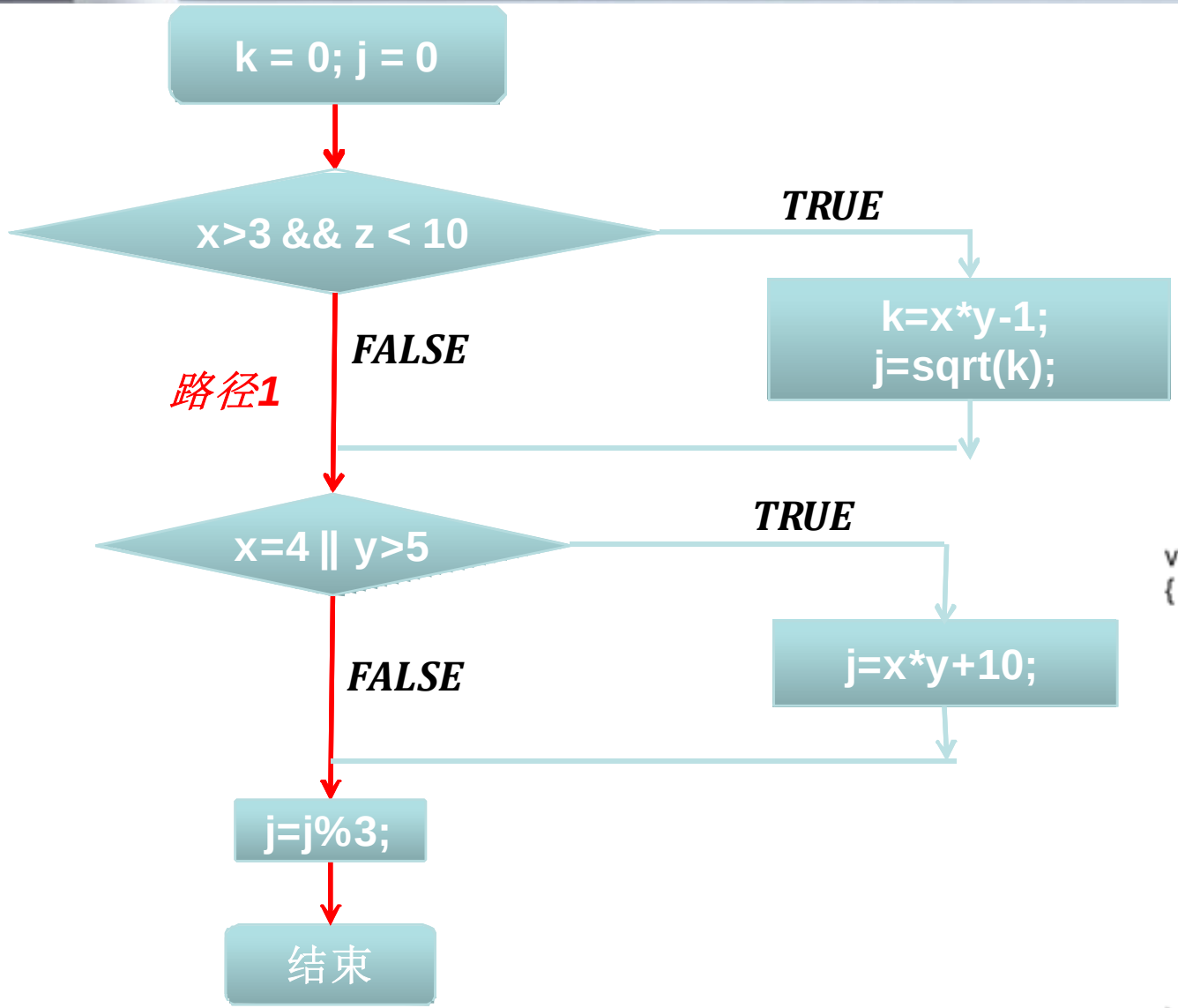
判定条件 M2

1.3 流程图



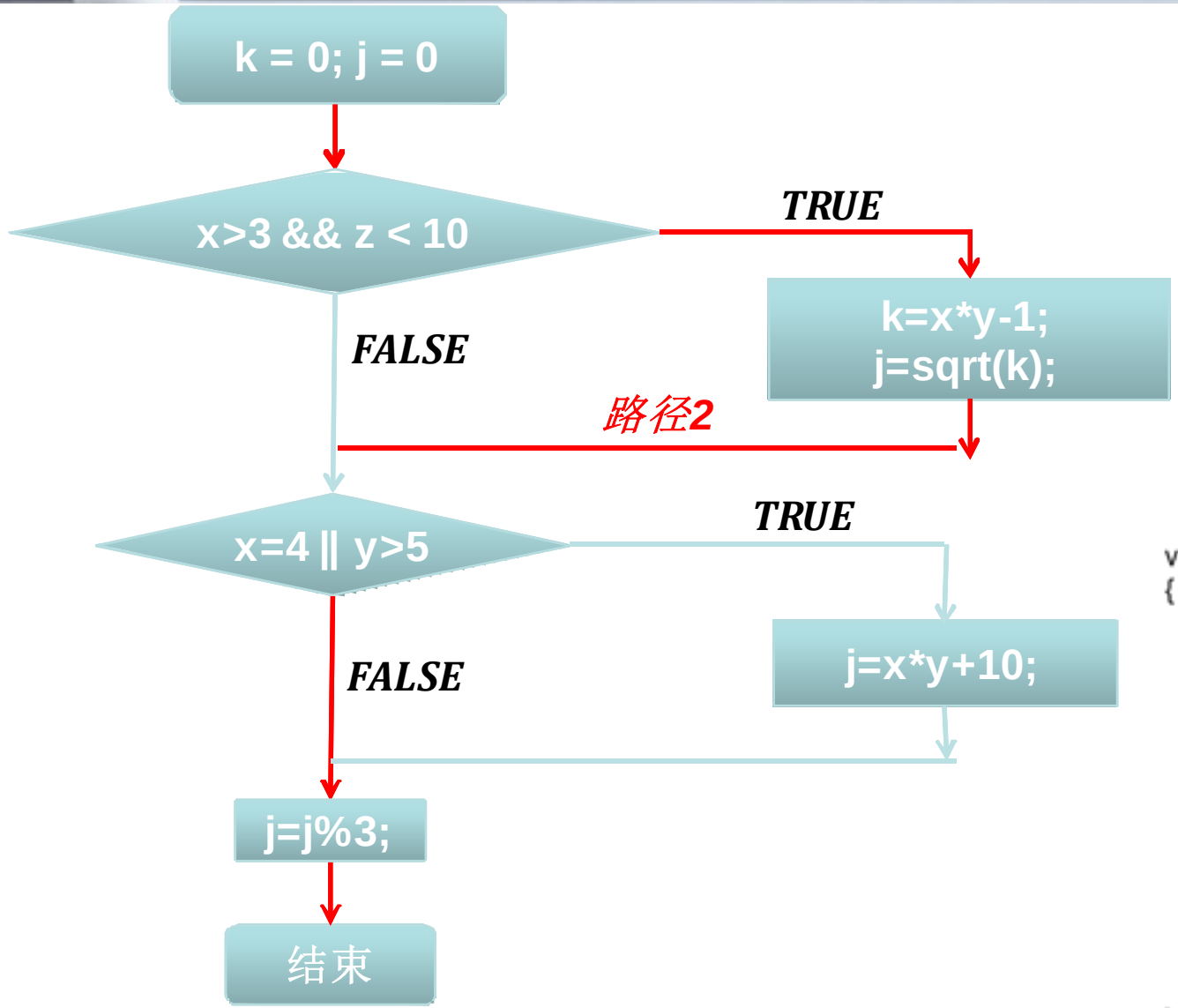

```
void Code(int x, int y, int z)
{
        int k = 0;
        int j = 0;
        if( (x > 3) && (z < 10) )
        {
                k = x * y - 1;
                j = sqrt(k);  //语句块1
        }
        if( (x==4)||(y > 5) )
        {
                j = x * y + 10;  //语句块2
        }
        j = j % 3;   //语句块3
}
```

1.3 流程图-路径1



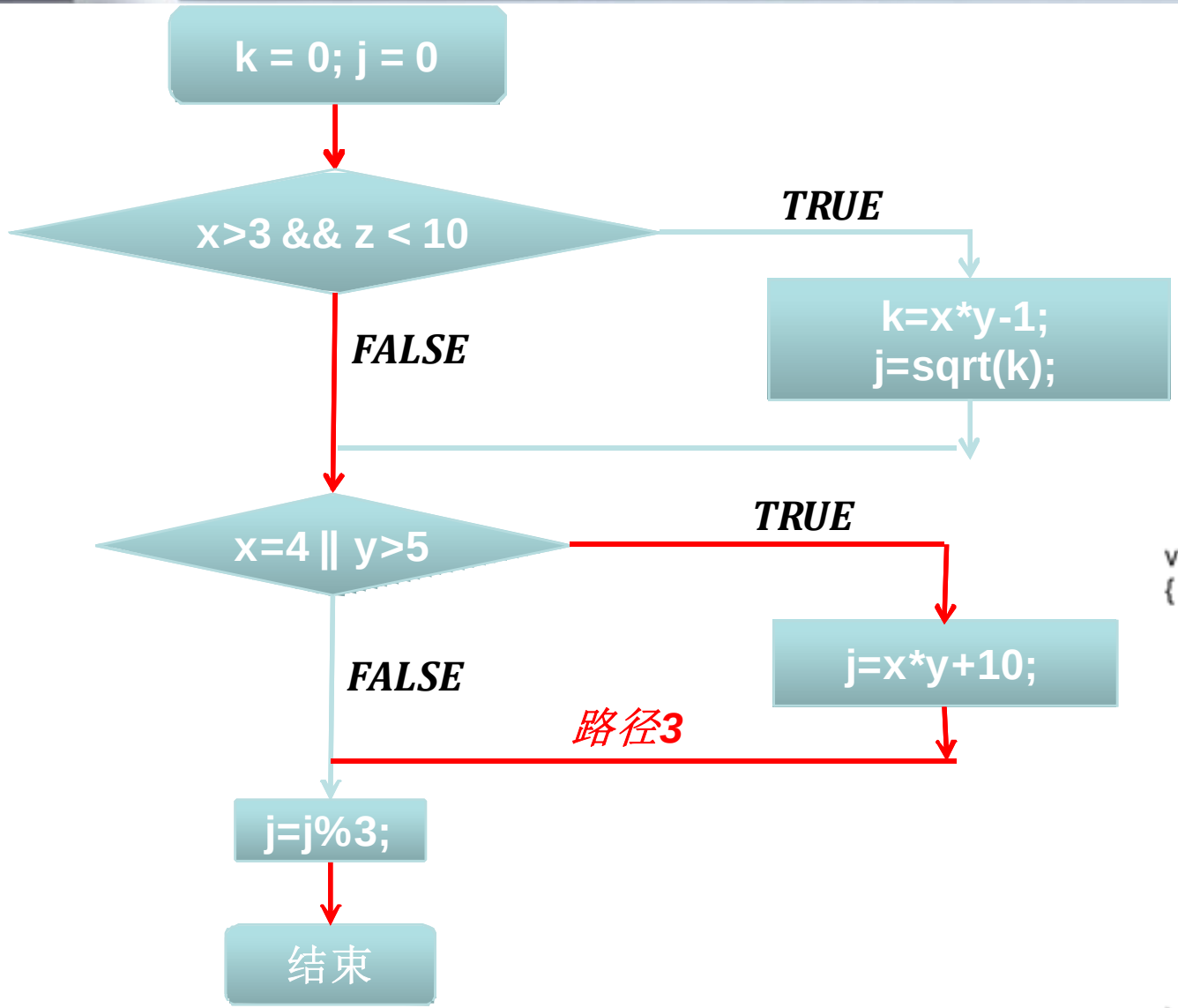

```
void Code(int x, int y, int z)
{
        int k = 0;
        int j = 0;
        if( (x > 3) && (z < 10) )
        {
                k = x * y - 1;
                j = sqrt(k);  //语句块1
        }
        if( (x==4)||(y > 5) )
        {
                j = x * y + 10;  //语句块2
        }
        j = j % 3;   //语句块3
}
```

1.3 流程图-路径2



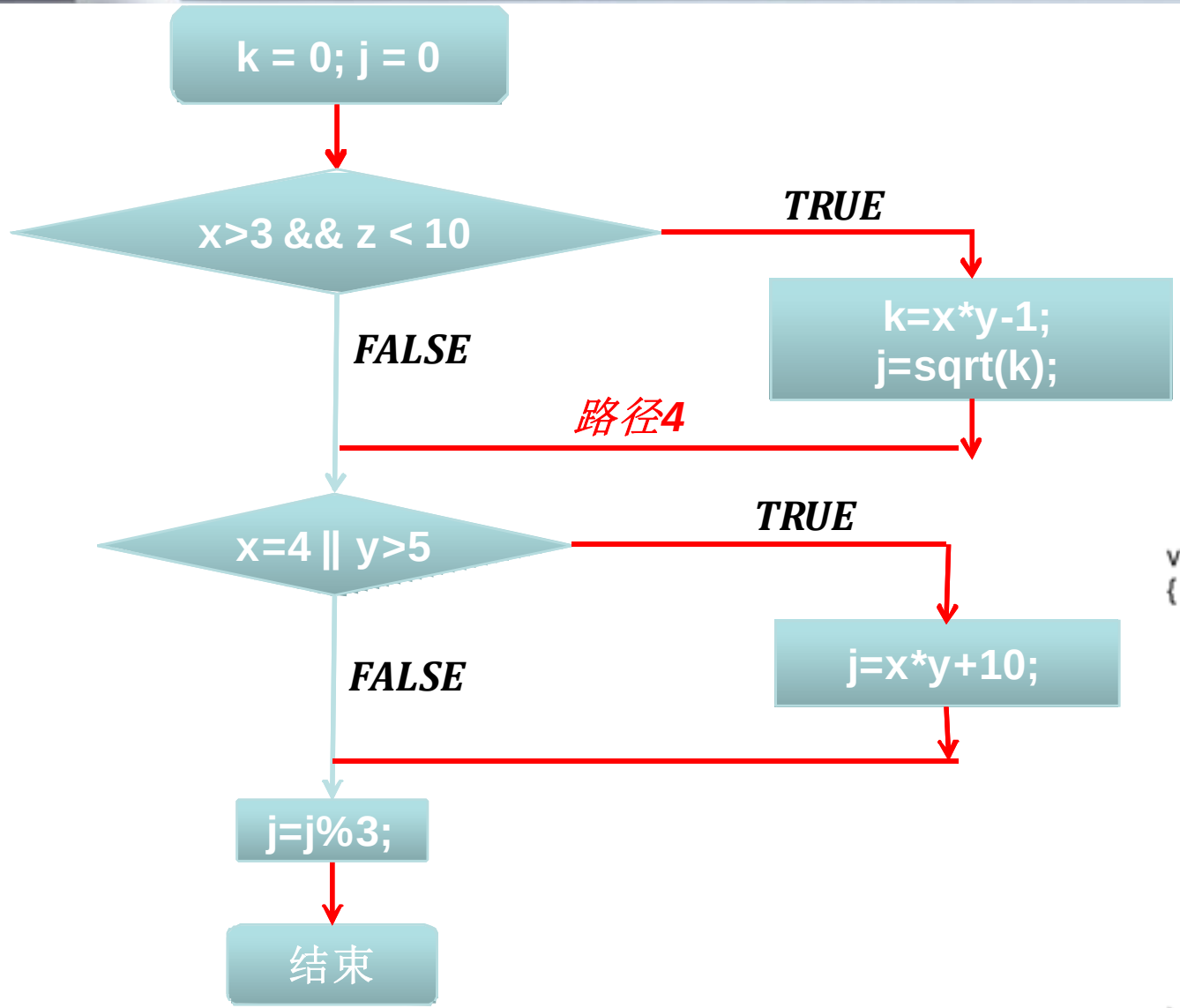

```
void Code(int x, int y, int z)
{
        int k = 0;
        int j = 0;
        if( (x > 3) && (z < 10) )
        {
                k = x * y - 1;
                j = sqrt(k);  //语句块1
        }
        if( (x==4)||(y > 5) )
        {
                j = x * y + 10;  //语句块2
        }
        j = j % 3;   //语句块3
}
```

1.3 流程图–路径3




```
void Code(int x, int y, int z)
{
        int k = 0;
        int j = 0;
        if( (x > 3) && (z < 10) )
        {
                k = x * y - 1;
                j = sqrt(k);  //语句块1
        }
        if( (x==4)||(y > 5) )
        {
                j = x * y + 10;  //语句块2
        }
        j = j % 3;   //语句块3
}
```

1.3 流程图-路径4

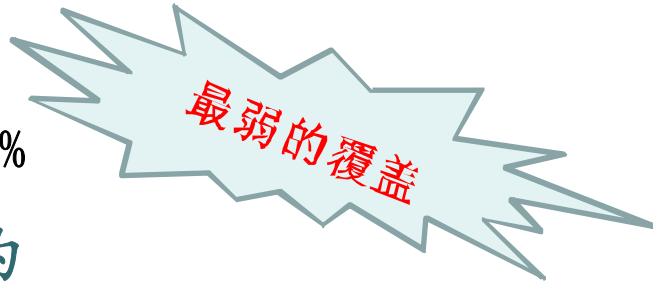



```
void Code(int x, int y, int z)
{
        int k = 0;
        int j = 0;
        if( (x > 3) && (z < 10) )
        {
                k = x * y - 1;
                j = sqrt(k);  //语句块1
        }
        if( (x==4)||(y > 5) )
        {
                j = x * y + 10;  //语句块2
        }
        j = j % 3;   //语句块3
}
```

1.3 语句覆盖

- 语句覆盖

  = （被执行的语句数量/总的语句数量）×100%

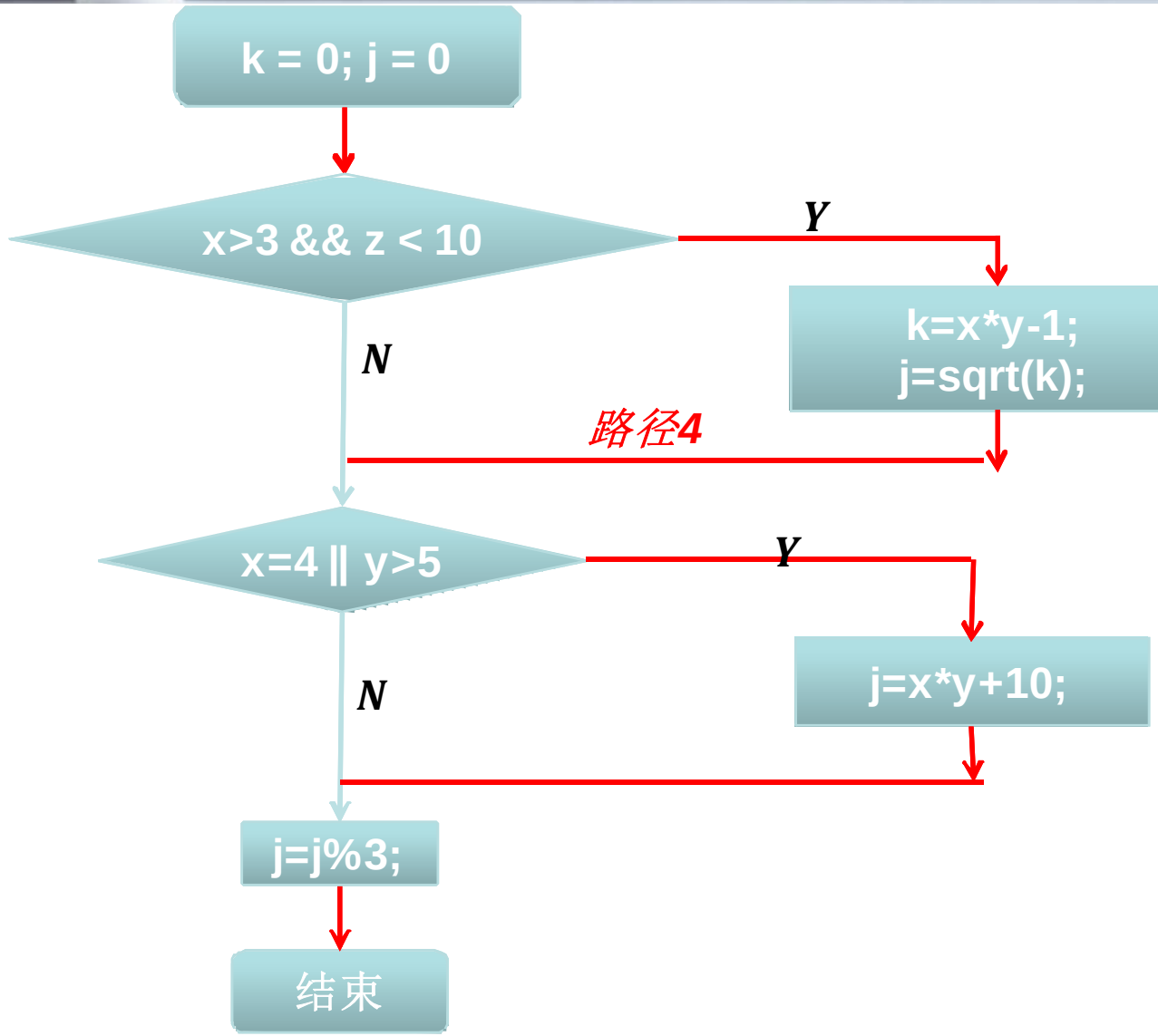


- 达到100%的语句覆盖可能很困难，因为
  - 不可达代码；
  - 处理错误代码（处理异常）；
  - 小概率事件

```
if( (x > y) && (y > z) && (z > x))
{
        cout<<“This should never happen!”;
}
```

测试用例	输出	M1	M2	覆盖路径
x=4; z=5; y=8	k=31, j=0	True	True	路径4

1.3 语句覆盖



测试用例：

x=4; z=5; y=8

100%语句覆盖

```
void Code(int x, int y, int z)
{
        int k = 0;
        int j = 0;
        if( (x > 3) && (z < 10) )
        {
                k = x * y - 1;
                j = sqrt(k);  //语句块1
        }
        if( (x==4)||(y > 5) )
        {
                j = x * y + 10;  //语句块2
        }
        j = j % 3;   //语句块3
}
```

1.3 分支/判定覆盖

分支/判定覆盖

= （被执行的分支数量/总的分支数量）×100%

要确保每个判定得到了TRUE和FALSE的结果，即保证每个判定条件取TRUE和取FALSE各至少一次

测试用例	输出	M1	M2	覆盖路径
x=4; z=5; y=8	k=31, j=0	TRUE	TRUE	路径4
x=2; z=11; y=5	k=0, j=0	FALSE	FALSE	路径1
x=13; z=5; y=2	k = 25, j = 2	TRUE	FALSE	路径2
x=4; z=11; y=6	k=0, j=1	FALSE	TRUE	路径3

1.3 分支/判定覆盖

分支/判定覆盖

= （被执行的分支数量/总的分支数量）×100%

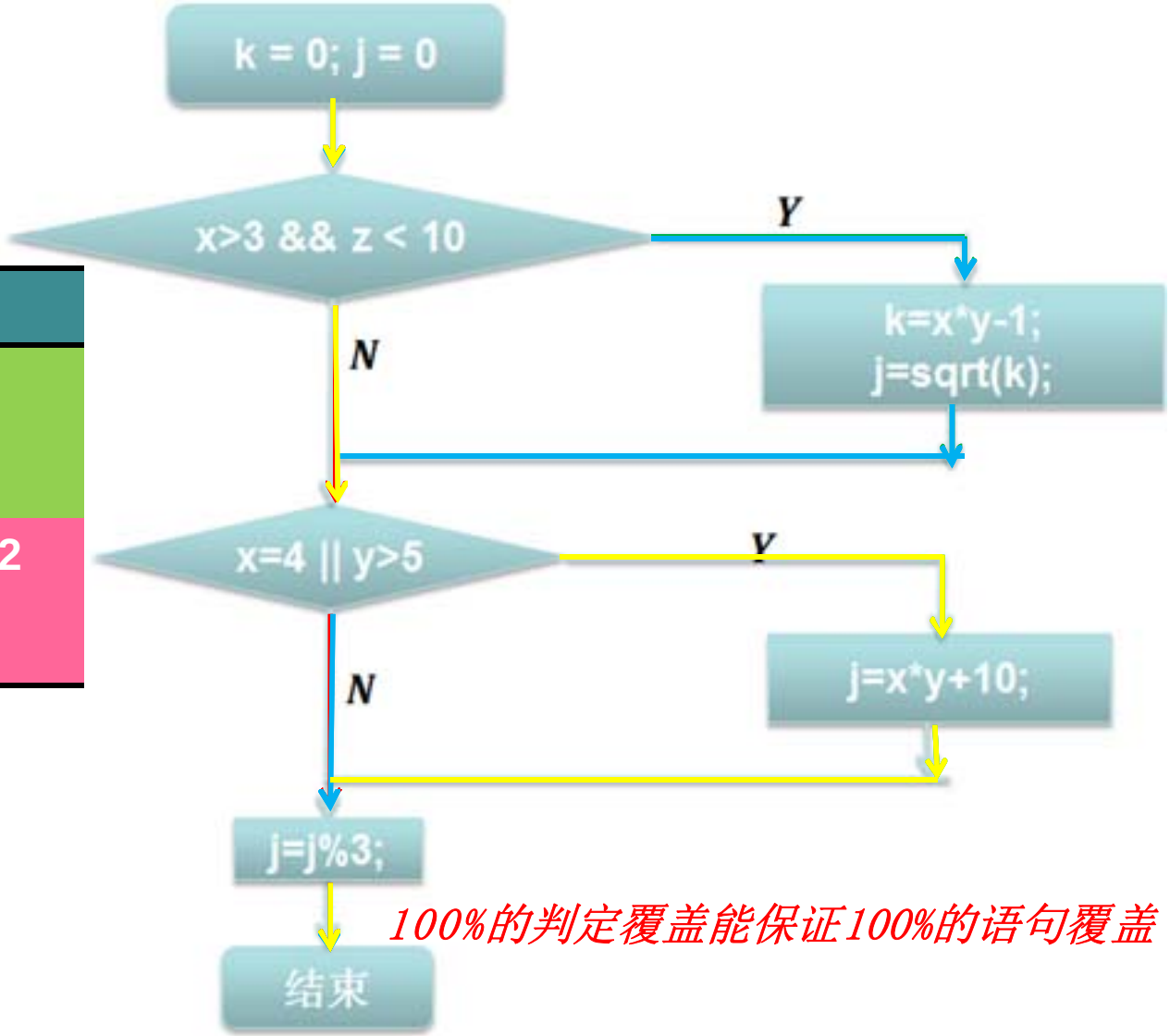
要确保每个判定得到了TRUE和FALSE的结果，即保证每个判定条件取TRUE和取FALSE各至少一次

测试用例	输出	M1	M2	覆盖路径
x=4; z=5; y=8	k=31, j=0	TRUE	TRUE	路径4
x=2; z=11; y=5	k=0, j=0	FALSE	FALSE	路径1
x=13; z=5; y=2	k = 25, j = 2	TRUE	FALSE	路径2
x=4; z=11; y=6	k=0, j=1	FALSE	TRUE	路径3

1.3 100%分支覆盖

100%分支覆盖

测试用例	输出
x=4; z=5; y=8	k=31, j=0
x=2; z=11; y=5	k=0, j=0
x=13; z=5; y=2	k = 25, j = 2
x=4; z=11; y=6	k=0, j=1

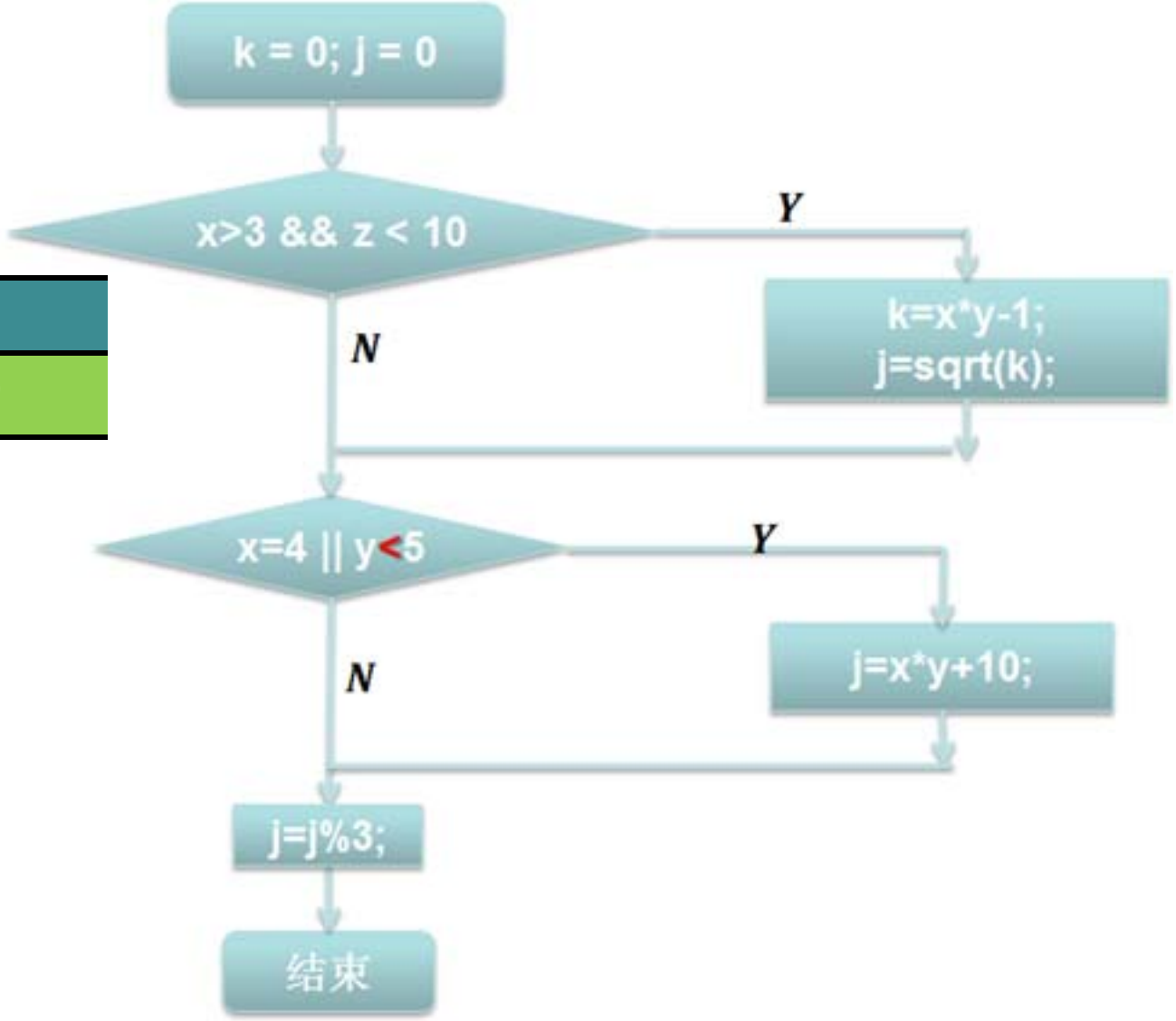


100%的判定覆盖能保证100%的语句覆盖

1.3 分支/判定覆盖缺点

测试用例	输出
x=4; z=5; y=8	k=31, j=0

忽略了表达式内的条件，不能发现每个条件的错误



1.3 条件覆盖

条件覆盖

要确保每个条件取TRUE和取FALSE各至少一次，可检查每个原子条件，不能保证所有判断分支都覆盖

原子条件：T1(x>3)，T2(z<10)，T3(x=4)，T4(y>5)

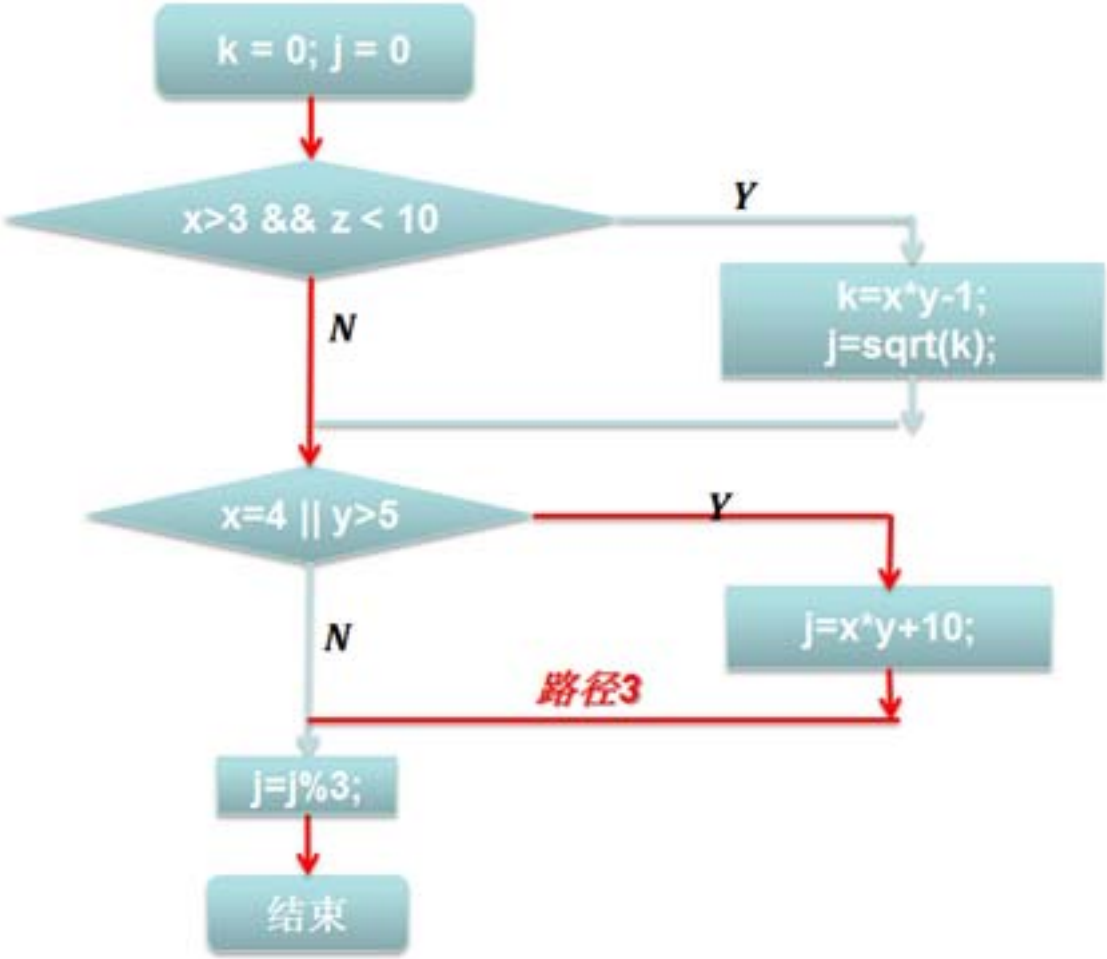
F1(x≤3)，F2(z≥10)，F3(x≠4)，F4(y≤5)

测试用例	输出	原子条件	覆盖路径
x=4; z=11; y=2	k = 0, j = 0	T1, F2, T3, F4	路径3
x=2; z=5; y=6	k=0, j=1	F1, T2, F3, T4	路径3

1.3 条件覆盖

条件覆盖 < 语句覆盖 < 判定覆盖

分析： 测试用例不一定能满足条件覆盖要求，同样这样可能会造成程序逻辑错误被遗漏。



1.3 判定/条件覆盖

判定/条件覆盖

判定条件中的所有**条件**可能至少执行一次取值，同时，所有**判定**的可能结果至少执行一次。判定覆盖与条件覆盖的结合。

原子条件：T1(x>3)，T2(z<10)，T3(x=4)，T4(y>5)

F1(x≤3)，F2(z≥10)，F3(x≠4)，F4(y≤5)

判定条件：M1 **(x > 3) && (z < 10)**

M2 **(x==4)||(y > 5)**

测试用例	输出	原子条件	M1	M2	覆盖路径
x=4; z=5; y=8	k=31, j=0	T1, T2, T3, T4	TRUE	TRUE	路径4
x=2; z=11; y=5	k=0, j=0	F1, F2, F3, F4	FALSE	FALSE	路径1

1.3 条件组合覆盖

条件组合覆盖

使得每个判定的所有可能的**条件取值组合**至少执行一次。

可满足分支覆盖，也同时满足语句覆盖

测试用例	原子条件 x>3; z<10	M1 (x>3&&z<10)	原子条件 x=4; y>5	M2 (x=4||y>5)	覆盖路径
x=4; y=6; z=9	T1, T2	TRUE	T3, T4	TRUE	路径4
x=4; y=5; z=10	T1, F2	FALSE	T3, F4	TRUE	路径3
x=2; y=6; z=9	F1, T2	FALSE	F3, T4	TRUE	路径2
x=4; y=5; z=10	F1, F2	FALSE	F3, F4	FALSE	路径1

1.3 覆盖率

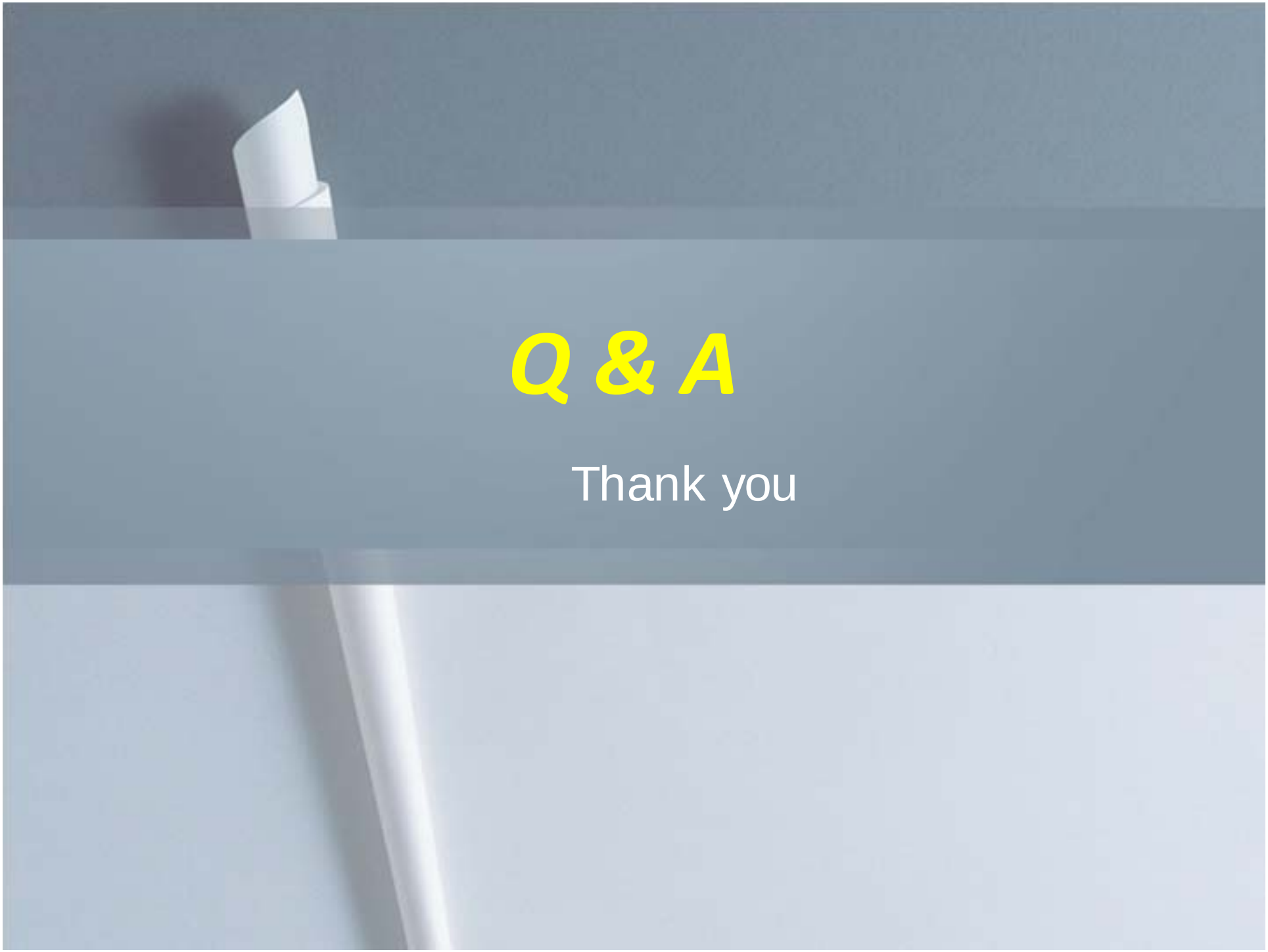
覆盖率使用的基本准则

- 覆盖率不是目的，只是一种手段
- 不可能针对所有的覆盖率去测试
- 只考虑一种覆盖标准不恰当
- 不要追求绝对100%的覆盖率

其它准则

- 使用最少测试来达到最大覆盖
- 要考虑测试质量，兼顾效益/成本分析
- 软件规格变化则需要更新相应的覆盖

Absence of proof is not proof of absence.
- William Cowper


Q & A
Thank you