



哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN



規格嚴格 功夫到家

1920—2025

面向对象的软件构造导论

第六章 软件测试及代码质量保障



抽象工厂模式概述

- 抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的方式。



抽象工厂模式

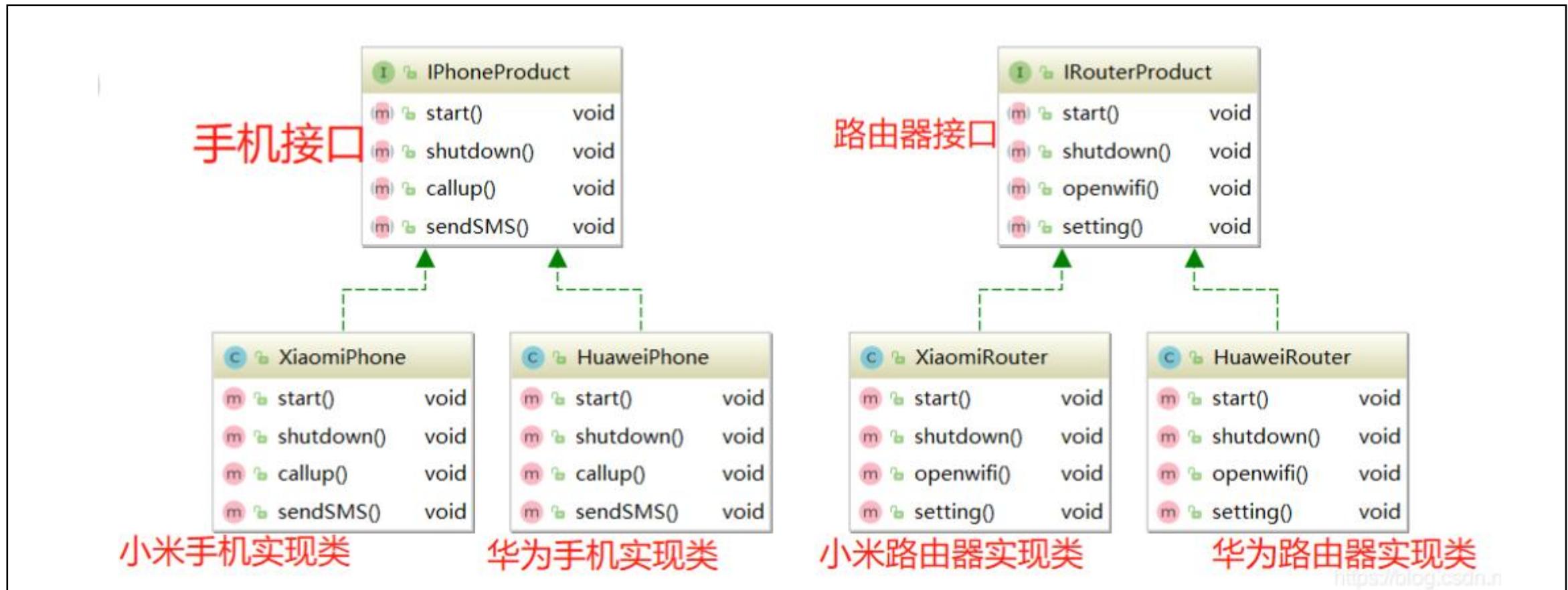
类图分析：





抽象工厂模式

类图分析：





抽象工厂模式

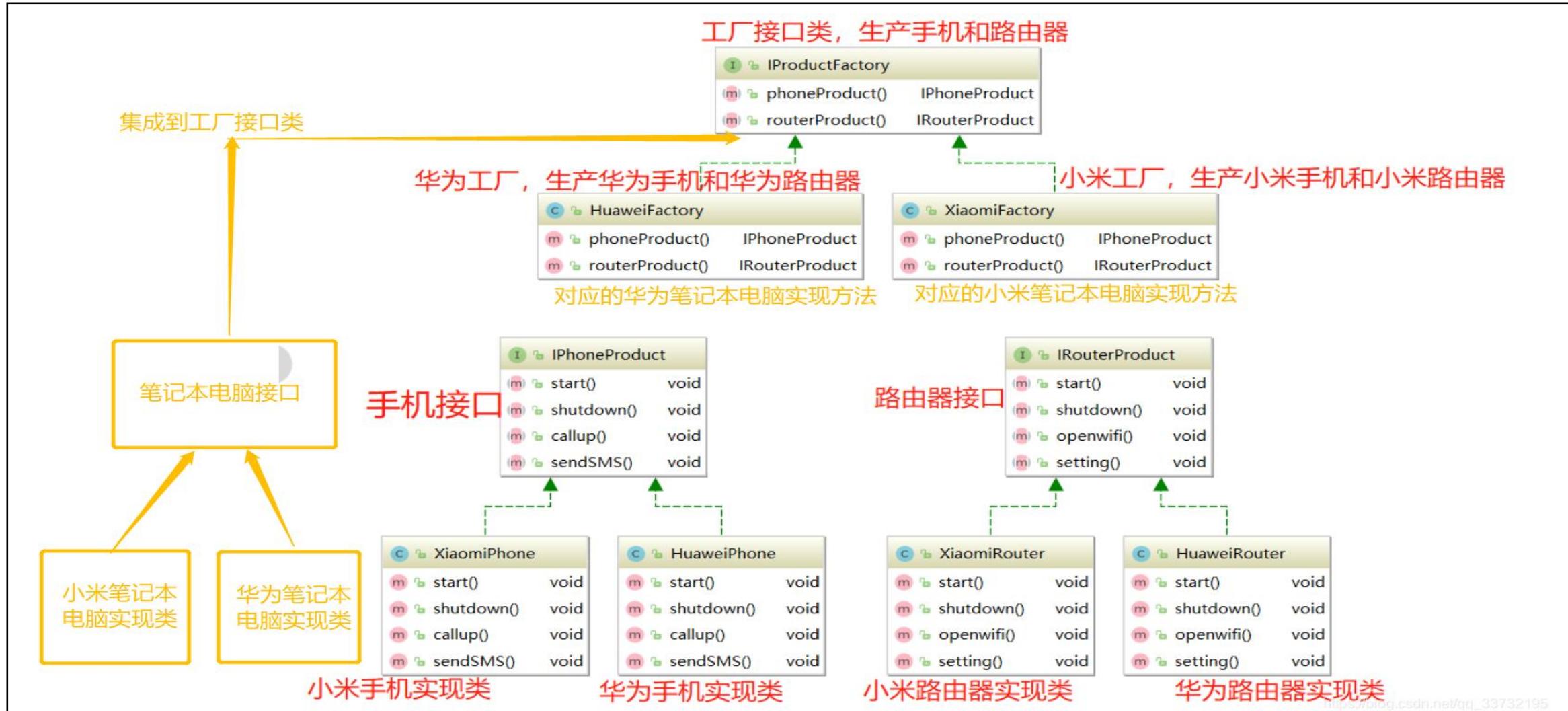
以上就是抽象工厂模式的实现，现在回看产品族和产品等级的概念，如果新增一个品牌产品族或产品类型会怎样？

拓展一个产品类型

- 我们会发现，拓展一个产品类型是非常困难的，例如产品类型中新增笔记本电脑，也就是说华为和小米现在可以生产电脑了，如下页图片所示（黄色字体为新增一个产品类型需要做的事），对顶层的工厂接口类也要修改，这是非常麻烦的。



抽象工厂模式





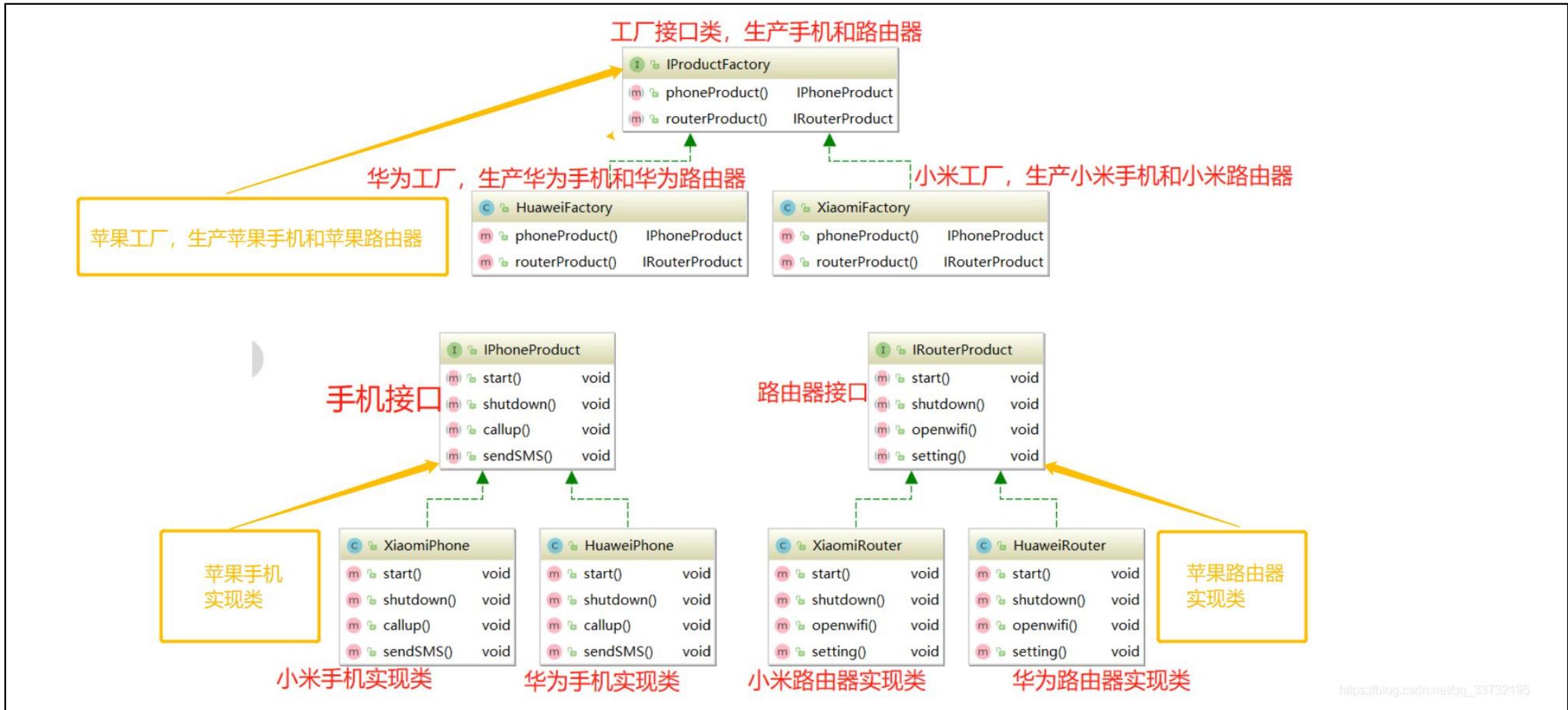
抽象工厂模式

拓展一个品牌（产品族）

- 如果新增一个品牌，也就是说新增一个厂商来生产手机和路由器，如下页图片所示（黄色字体为新增一个产品族需要做的事），新增一个产品族不用修改原来的代码，符合OCP原则，十分方便。



抽象工厂模式





抽象工厂模式优缺点

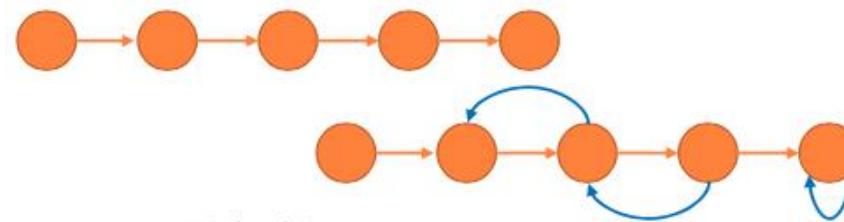
- 优点
 - 一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象（将一个品牌的产品统一一起创建）。
- 缺点
 - 产品类型扩展非常困难，要新增一个产品类，既要修改工厂抽象类里的代码，又修改具体的实现类里面的代码。
 - 增加了系统的抽象性和理解难度。



传统软件开发过程模型

□ 两种基本模型

- 线性模型
- 迭代过程



□ 传统软件开发过程模型

- 瀑布过程
- 增量过程
- 原型过程



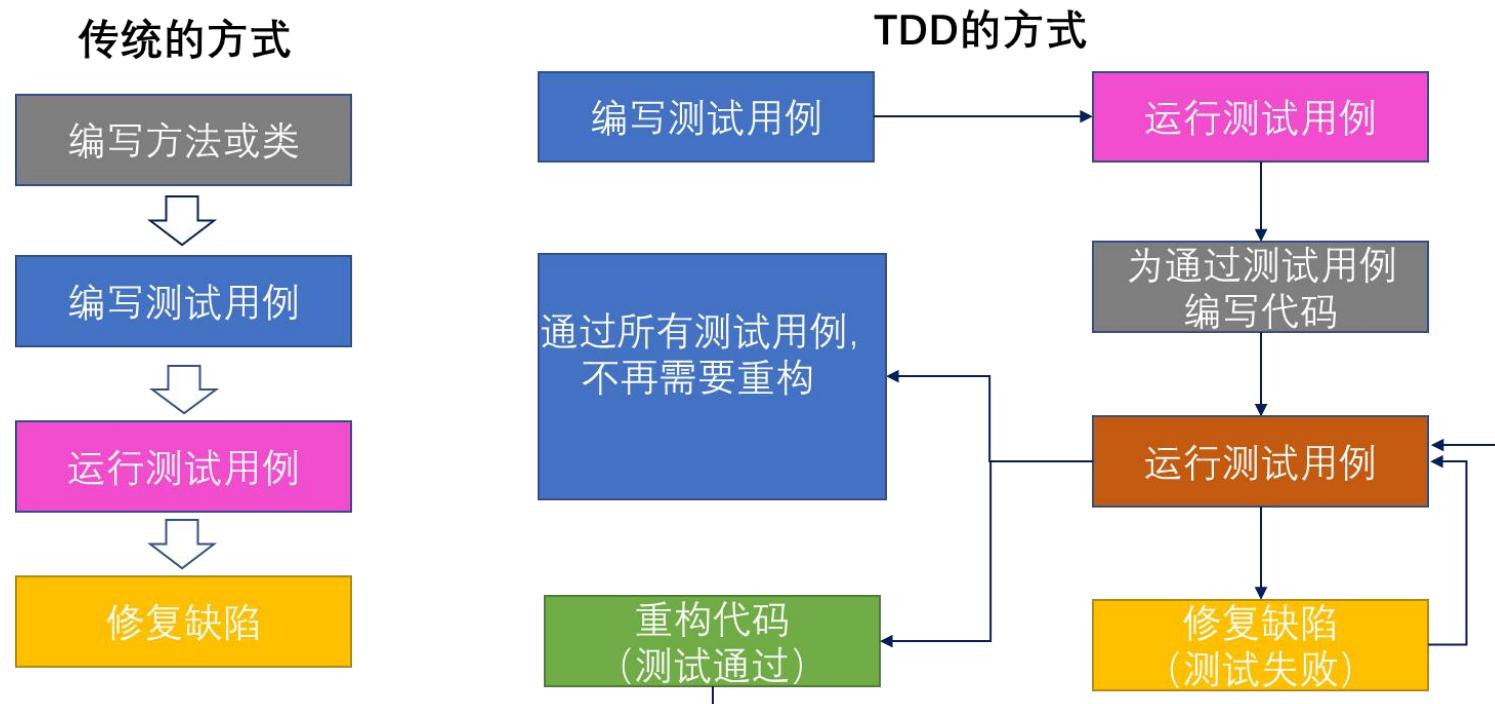
□ 流行的软件开发过程模型

- 敏捷开发
- 测试驱动开发



测试驱动的开发

测试驱动开发 (Test-Driven Development, 简称TDD) 是一种不同于传统软件开发流程的新型的开发方法。它要求在**编写某个功能的代码之前先编写测试代码**，然后只编写使测试通过的功能代码，通过测试来推动整个开发的进行。这有助于编写**简洁可用**和高质量的代码，并加速开发过程。





第六章：软件测试及代码质量保障

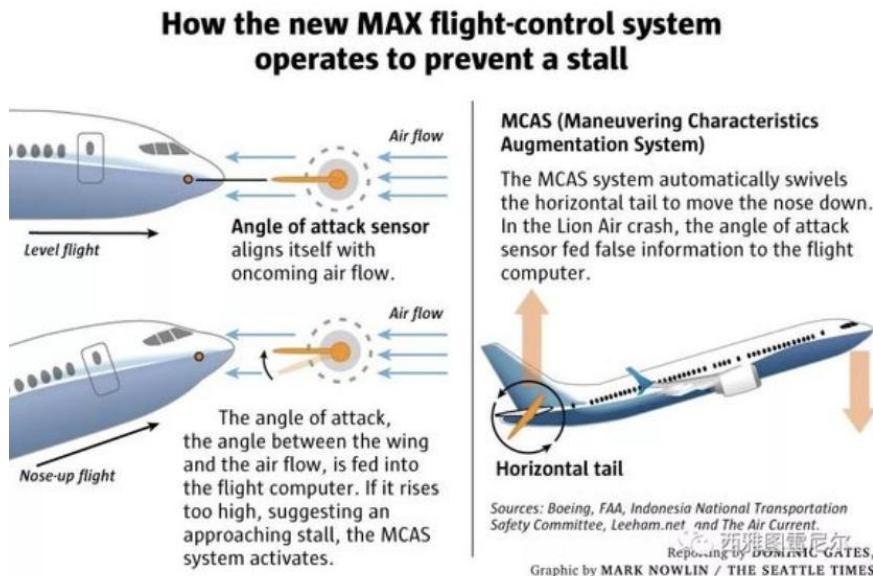
- 1. 软件测试的定义和分类
- 2. 测试用例的定义和其设计方法
- 3. 白盒测试（定义、控制流程图、逻辑覆盖方法、测试用例设计）
- 4. 黑盒测试（定义、等价类划分、边界值分析、场景法）
- 5. 压力测试，性能测试和代码覆盖率测试
- 6. 代码质量保证



软件缺陷的典型案例分析



波音737-MAX8，2018年10月



- if (AOA_sensor_1.value > threshold) { activate_MCAS();
if ((AOA_sensor_1.value > threshold) &&
(AOA_sensor_2.value agrees) && (no sensor fault flag){}
- 5秒后再次自动启动
- MCAS每次启动时，移动水平安定面的幅度非常大，最高可达2.5 度



软件测试的定义(IEEE)

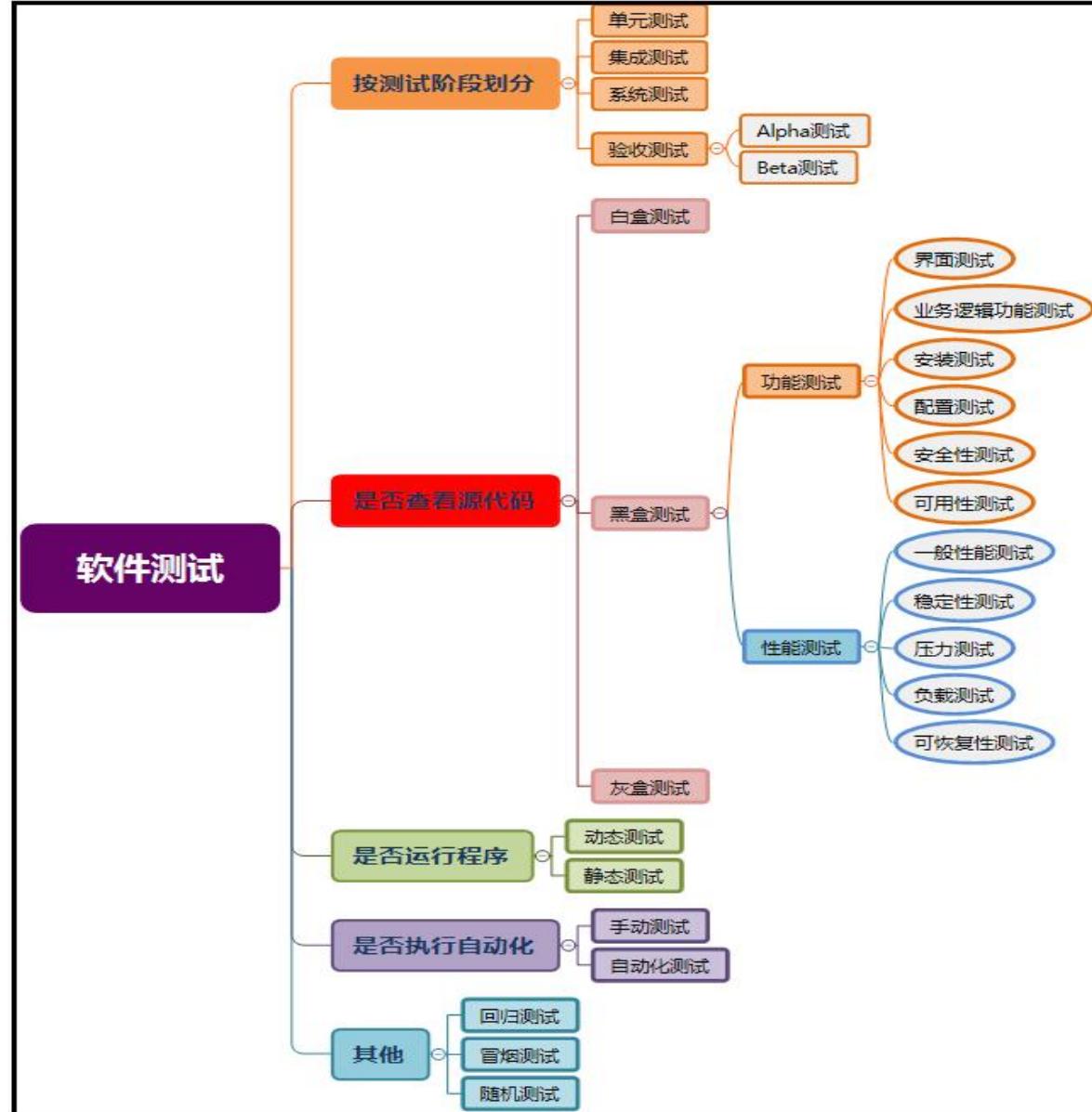
使用人工或自动的手段来运行或测量被测系统，或静态检查被测系统的过程，其目的在于校验被测系统**是否满足需求**，并找出实际系统输出和预期结果之间的差异。软件测试是以需求为中心，并非以缺陷为中心！

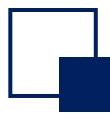




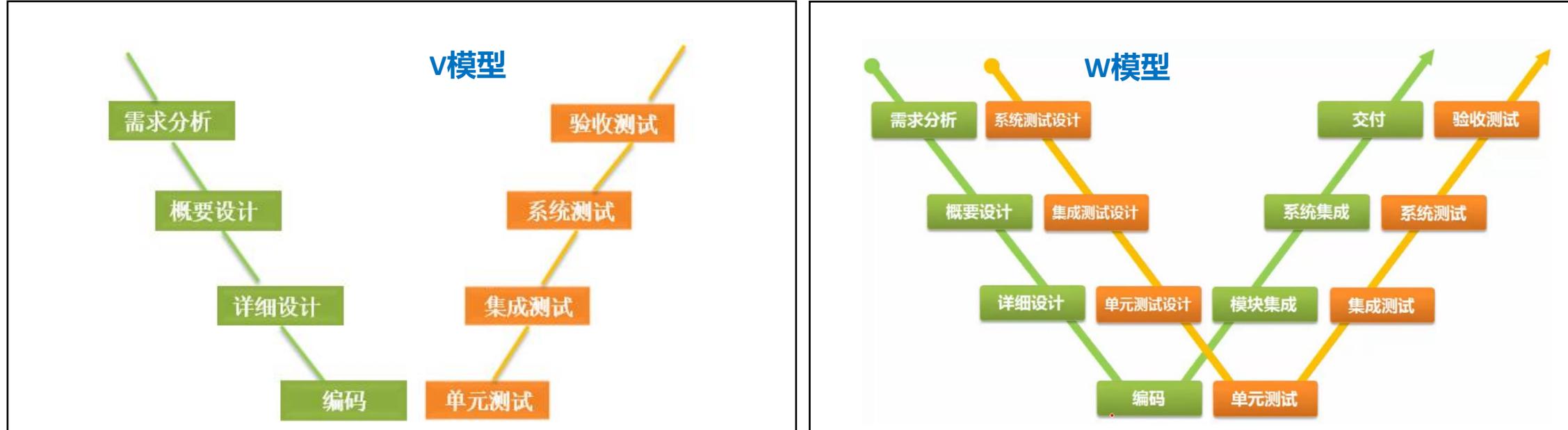
软件测试的分类

软件生命周期中将会执行不同类型的或级别的软件测试，组合应用这些**不同类型的**的测试才能确保软件系统功能和行为符合预期要求。

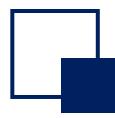




软件测试的分类



- 单元测试对应编码阶段，其测试对象是单个模块或组件；
- 集成测试对应详细设计，其测试对象是一组模块或组件；
- 系统测试和验收测试分别对应概要设计和需求阶段，测试对象是整个系统。



测试用例：定义

测试用例(Test Case)是指对一项特定的软件产品进行测试任务的描述，体现测试方案、方法、技术和策略。其内容包括测试目标、测试环境、输入数据、测试步骤、预期结果、测试脚本等，最终形成文档。简单地认为，测试用例是为某个特殊目标而编制的一组[测试输入](#)、[执行条件](#)以及[预期结果](#)，用于核实是否满足某个特定软件[需求](#)。

Problem and the solution to it arise simultaneously.



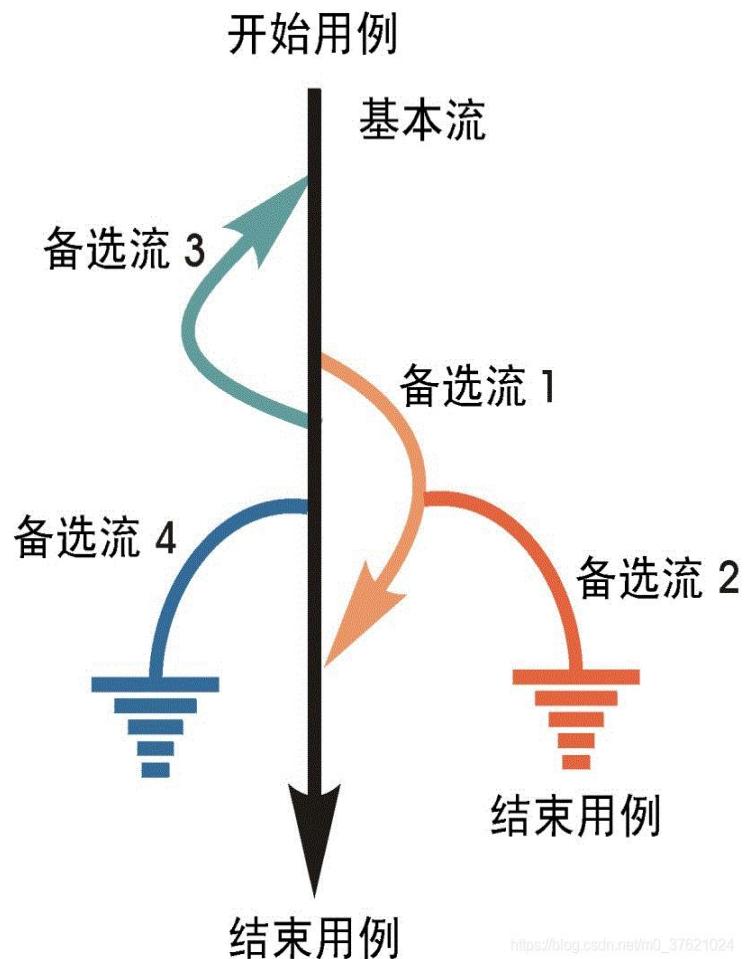
测试用例：设计方法

测试用例（Test Case）是将软件测试的行为活动做一个科学化的组织归纳，目的是能够将软件测试的行为转化成可管理的模式；同时测试用例也是将测试具体量化的方法之一，不同类别的软件，测试用例是不同的。

测试用例的设计方法主要有黑盒测试法和白盒测试法。

黑盒测试也称功能测试，黑盒测试着眼于程序外部结构，不考虑内部逻辑结构，主要针对软件界面和软件功能进行测试。

白盒测试又称结构测试、透明盒测试、逻辑驱动测试或基于代码的测试。白盒法全面了解程序内部逻辑结构、对所有逻辑路径进行测试。



https://blog.csdn.net/m0_37621024



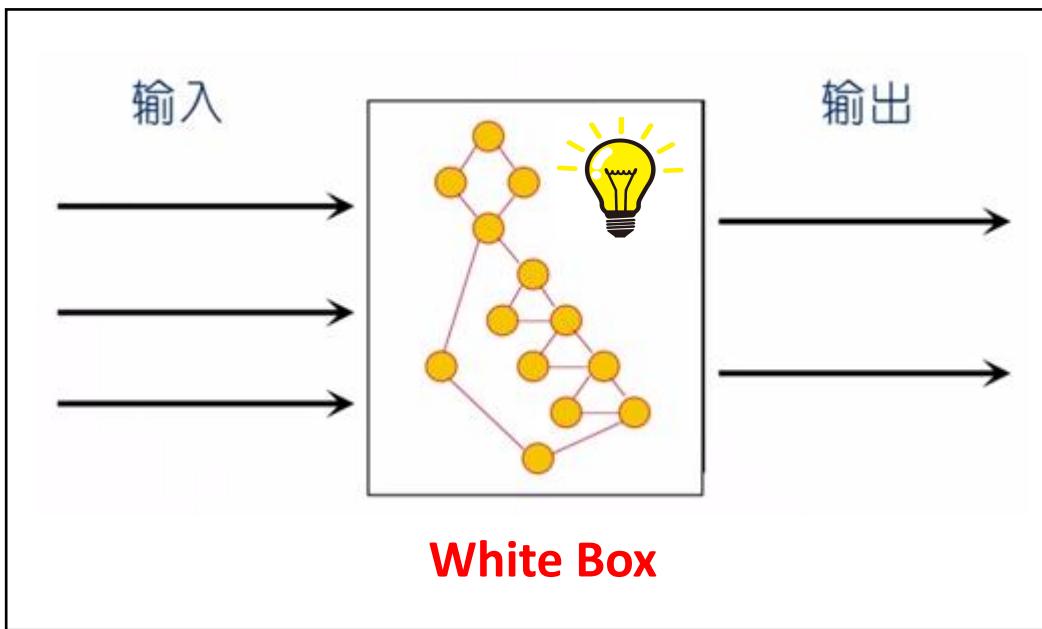
测试用例：设计原则

- 测试用例设计一般遵循以下原则：
 - **正确性。** 输入用户实际数据以验证系统是否满足需求规格说明书的要求;测试用例中的测试点应首先保证要至少覆盖需求规格说明书中的各项功能，并且正常。
 - **全面性。** 覆盖所有的需求功能项;设计的用例除对测试点本身的测试外，还需考虑用户实际使用的情况、与其他部分关联使用的情况、非正常情况(不合理、非法、越界以及极限输入数据)操作和环境设置等。
 - **连贯性。** 用例组织有条理、主次分明，尤其体现在业务测试用例上;用例执行粒度尽量保持每个用例都有测点，不能同时覆盖很多功能点，否则执行起来牵连太大，所以每个用例间保持连贯性很重要。
 - **可判定性。** 测试执行结果的正确性是可判定的，每一个测试用例都有相应的期望结果。
 - **可操作性。** 测试用例中要写清楚测试的操作步骤，以及与不同的操作步骤相对应的测试结果。



白盒测试：定义

也称作结构测试或逻辑驱动测试，它是基于程序的**源代码**，已知产品的内部工作过程，主要是对**程序内部结构**展开测试，关注程序实现细节，检验程序中的**每条通路**是否都有按照预定要求正确工作。



优势:

- 针对性强，可快速定位Bug
- 函数级别，Bug修复成本低
- 有助于了解测试的覆盖程度
- 有助于优化代码，预防缺陷

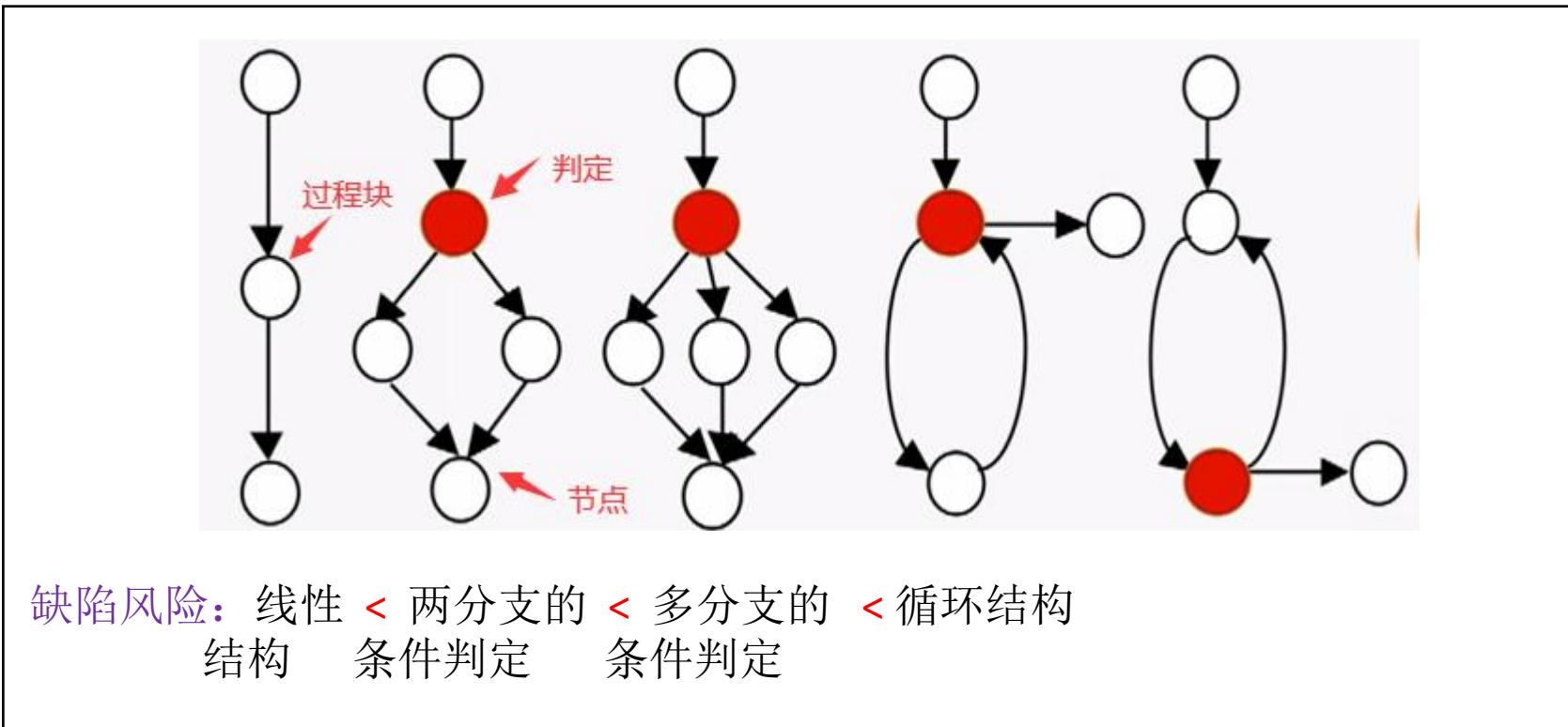
劣势:

- 对测试人员要求高
- 成本高



白盒测试：控制流程图

一种表示程序控制结构的图形工具，其基本元素是**节点**、**判定节点**、过程块。





白盒测试：建立被测对象模型

源代码

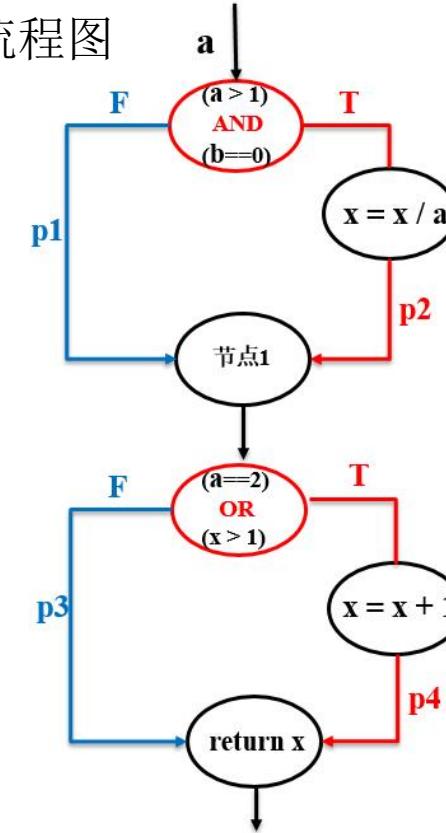
```
int Func1 (int a, int b, int x)
{
    if ((a > 1) && (b == 0))
        x = x / a;

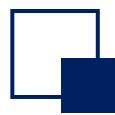
    if ((a == 2) || (x > 1))
        x = x + 1;

    return x;
}
```



控制流程图





白盒测试：逻辑覆盖方法

对判定的测试

- **关注点**: 判定表达式本身的复杂度
- **原理**: 考察源代码中的判定表达式, 通过对程序逻辑结构的遍历, 来实现测试对程序的覆盖。
- 常见覆盖指标:





白盒测试：逻辑覆盖方法

| 覆盖指标 | |
|--------|--|
| 语句覆盖 | 至少执行程序中 所有语句 一次。语句覆盖是 最弱的 逻辑覆盖准则。 |
| 判定覆盖 | 也称为分支覆盖，至少执行程序中 每个分支 一次，保证程序中每个判定节点取得每种可能的结果至少一次。 |
| 条件覆盖 | 保证程序中每个复合判定表达式中，每个简单判定条件的取真和取假情况至少执行一次。 |
| 判定条件覆盖 | 保证程序中每个复合判定表达式中，每个简单判定条件的取真和取假情况至少执行一次（条件覆盖），同时保证程序中每个判定节点取得每种可能的结果至少一次（判定覆盖）。 |
| 条件组合覆盖 | 保证程序每个判定节点中，所有简单判定条件的所有可能的取值组合情况至少执行一次。 |
| 路径覆盖 | 使程序中 每一条 可能的 路径 至少执行一次。该策略是 最强的 ，但一般是不可实现的。 |



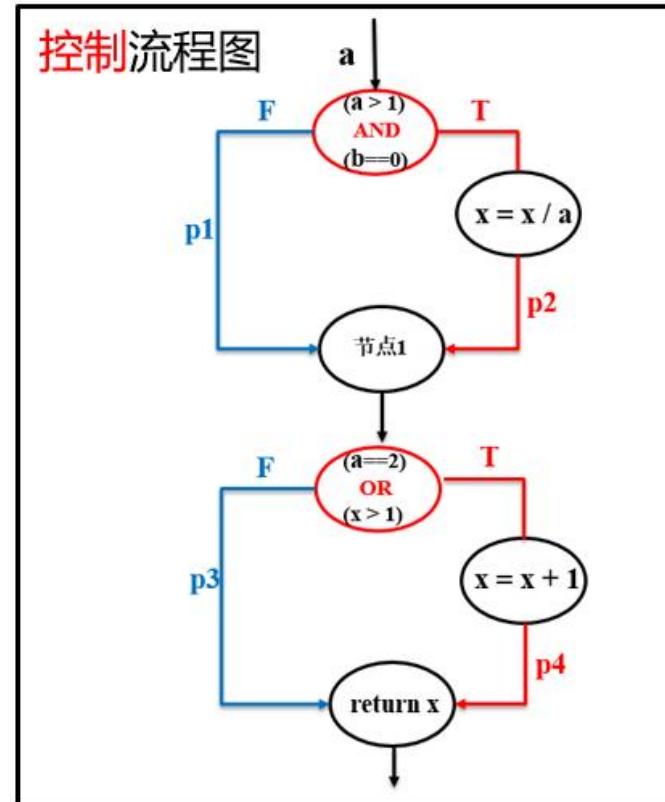
白盒测试：一个例子

- 4个基本逻辑判定条件

| | |
|----|----------|
| T1 | $a > 1$ |
| T2 | $b == 0$ |
| T3 | $a == 2$ |
| T4 | $x > 1$ |

- 4条执行路径

| | |
|-----|---------------------|
| L13 | $p1 \rightarrow p3$ |
| L14 | $p1 \rightarrow p4$ |
| L23 | $p2 \rightarrow p3$ |
| L24 | $p2 \rightarrow p4$ |





白盒测试：一个例子

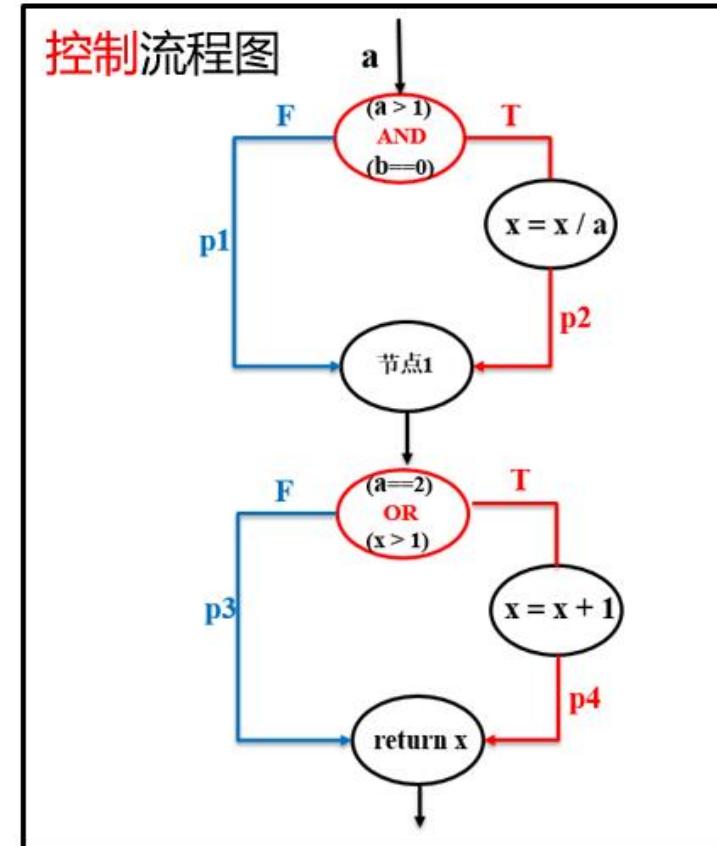
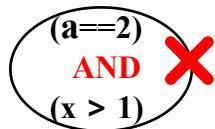
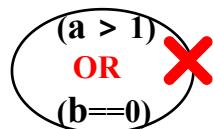
(一) 语句覆盖

至少执行程序中所有语句一次。语句覆盖是最弱的逻辑覆盖准则。

设计测试用例，实现语句覆盖：

| 测试用例ID | 输入 | | | 预期输出 | 覆盖路径 |
|--------|----|---|---|------|------|
| | a | b | x | | |
| TC1 | 2 | 0 | 4 | 3 | L24 |

测试漏洞：





白盒测试：一个例子

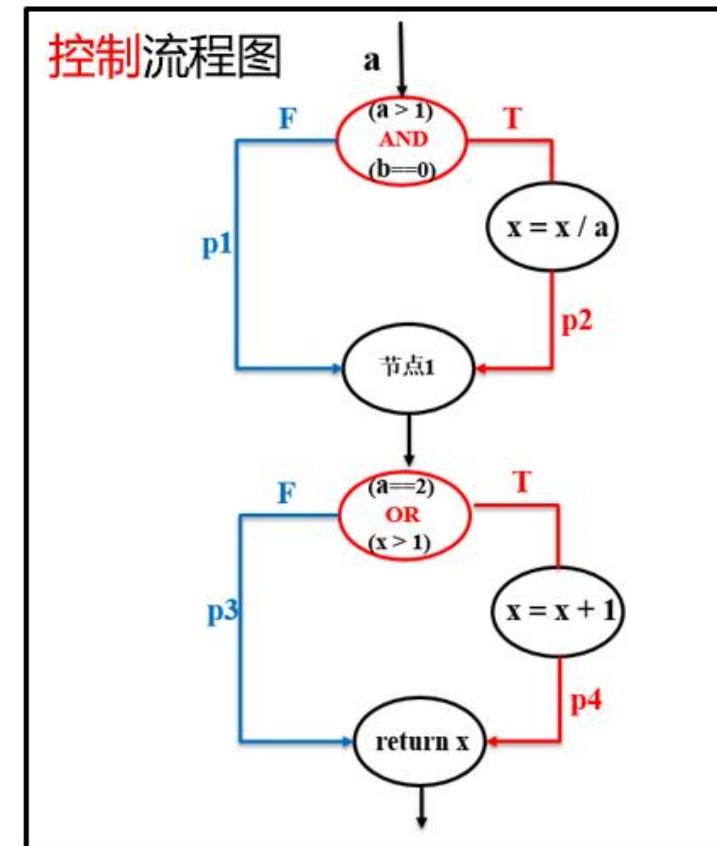
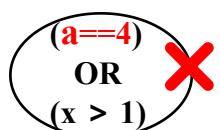
(二) 判定覆盖

也称为分支覆盖，至少执行程序中每个分支一次，保证程序中每个判定节点取得每种可能的结果至少一次。

设计测试用例，实现判定覆盖：

| 测试用例 ID | 输入 | | | 预期输出 | 覆盖路径 |
|---------|----|---|---|------|------|
| | a | b | x | | |
| TC1 | 1 | 1 | 2 | 3 | L14 |
| TC2 | 3 | 0 | 3 | 1 | L23 |

测试漏洞：



白盒测试：一个例子

(三) 条件覆盖

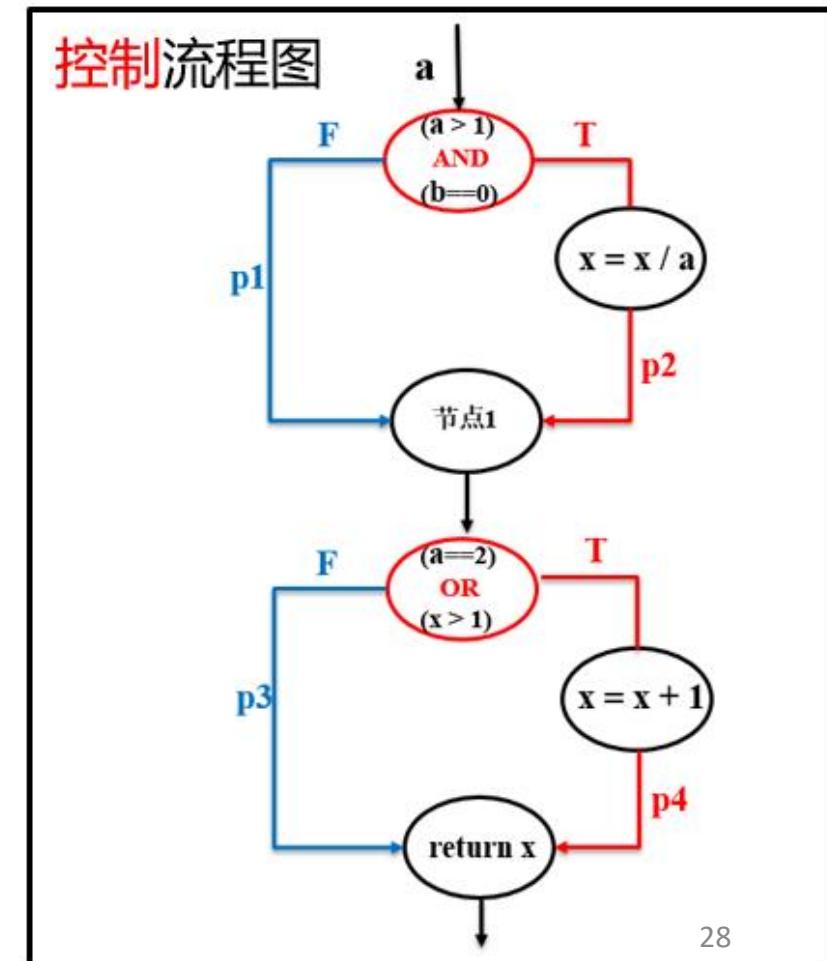
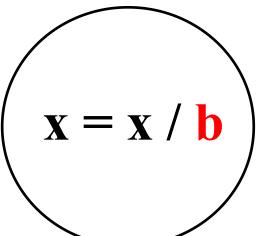
不仅每个语句至少执行一次，而且使判定表达式中的每个条件都取到各种可能的结果（真、假）。

| 测试用例ID | 输入 | | | 预期输出 | 覆盖路径 | 覆盖条件 |
|--------|----|---|---|------|------|----------------------------|
| | a | b | x | | | |
| TC1 | 2 | 1 | 0 | 1 | L14 | T1(T) T2(F) T3(T) T4(F) |
| TC2 | 1 | 0 | 2 | 3 | L14 | T1(F) T2(T) T3(F) T4(T) |

- 4个基本逻辑判定条件

| | | | |
|----|----------|----|----------|
| T1 | $a > 1$ | T3 | $a == 2$ |
| T2 | $b == 0$ | T4 | $x > 1$ |

测试漏洞：





白盒测试：一个例子

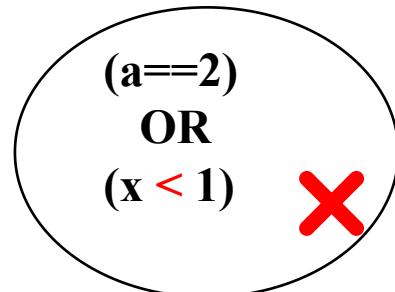
(四) 判定/条件覆盖

同时满足判定覆盖和条件覆盖。

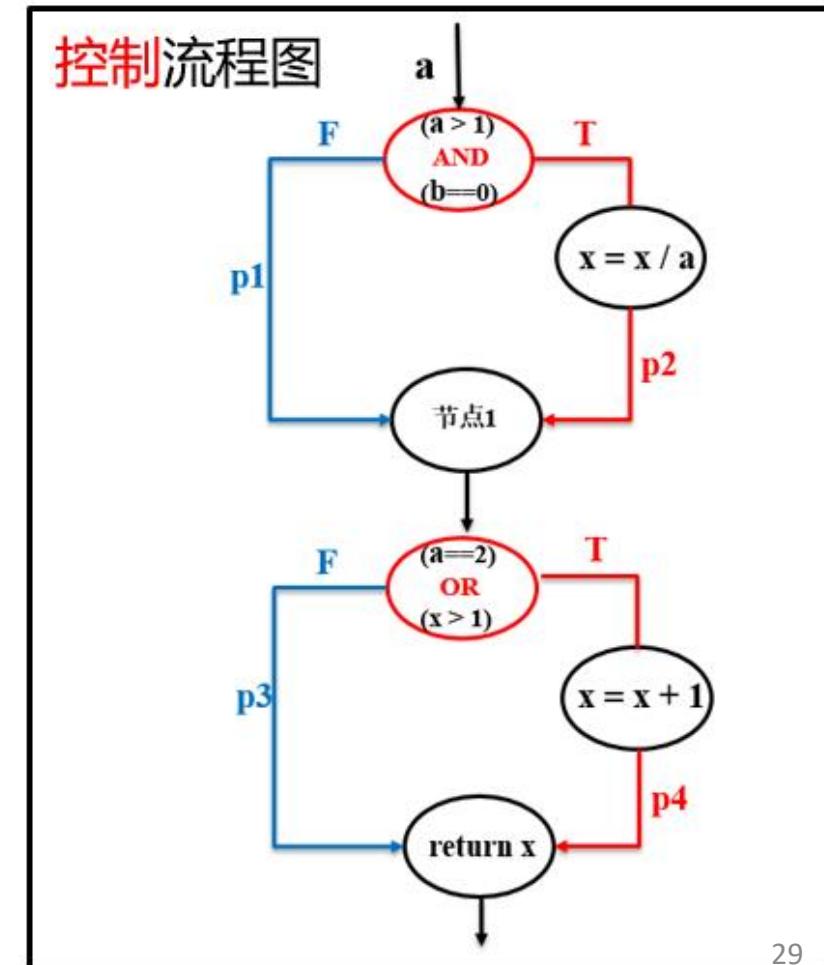
| 测试用例ID | 输入 | | | 预期输出 | 覆盖路径 | 覆盖条件 |
|--------|----|---|---|------|------|----------------------------|
| | a | b | x | | | |
| TC1 | 2 | 0 | 4 | 3 | L24 | T1(T) T2(T) T3(T) T4(T) |
| TC2 | 1 | 1 | 1 | 1 | L13 | T1(F) T2(F) T3(F) T4(F) |

- 4个基本逻辑判定条件

测试漏洞：



| | | | |
|----|--------|----|--------|
| T1 | a > 1 | T3 | a == 2 |
| T2 | b == 0 | T4 | x > 1 |





白盒测试：一个例子

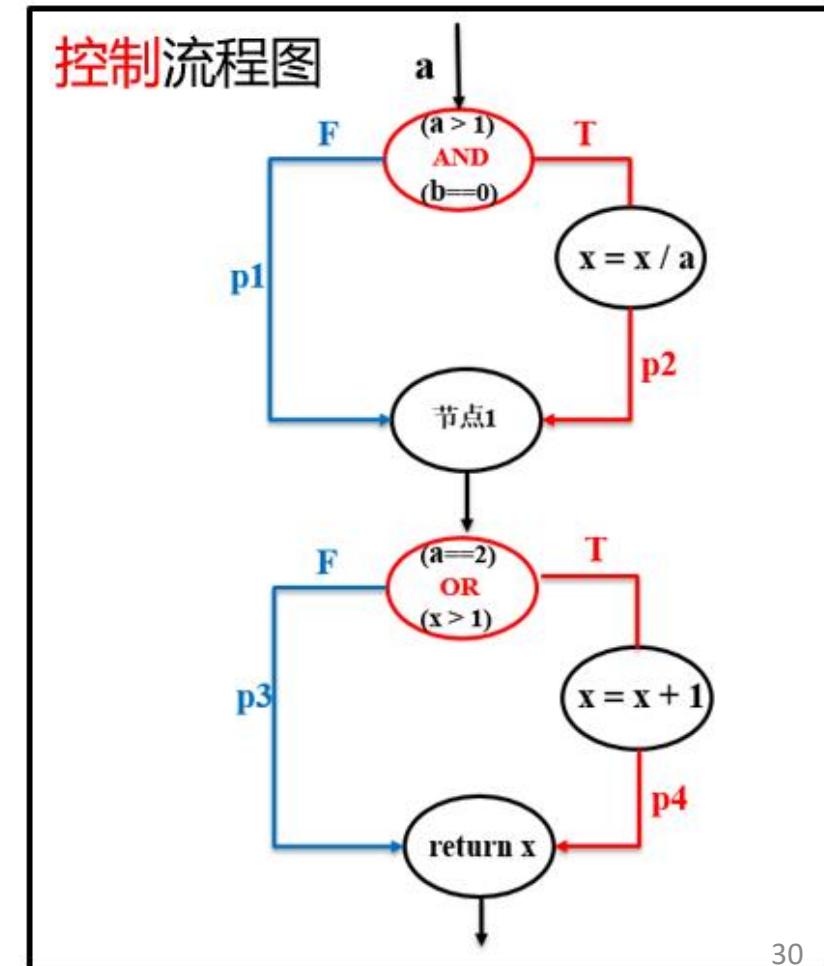
(五) 条件组合覆盖

保证程序每个判定节点中，所有简单判定条件的所有可能的取值组合情况至少执行一次。

总共8种可能的组合情况：

| 简单判定条件 | |
|----------|---------|
| T1:a > 1 | T2:b==0 |
| T | T |
| T | F |
| F | T |
| F | F |

| 简单判定条件 | |
|---------|----------|
| T3:a==2 | T4:x > 1 |
| T | T |
| T | F |
| F | T |
| F | F |





白盒测试：一个例子

(五) 条件组合覆盖

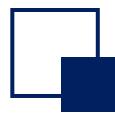
设计用例，实现条件组合覆盖：

可以选择 TC1、TC6、TC9、TC12

缺点：

- 可能存在冗余。如红框内的用例，有相同的覆盖路径。
- 组合数量大，花费时间多

| 测试用例ID | 输入 | | | 预期输出 | 判定条件 | | | | 覆盖路径 |
|--------|----|---|---|------|------|----|----|----|------|
| | a | b | x | | x | T1 | T2 | T3 | |
| TC1 | 2 | 0 | 4 | 3 | T | T | T | T | L24 |
| TC2 | 2 | 0 | 2 | 2 | T | T | T | F | L24 |
| TC3 | 3 | 0 | 6 | 3 | T | T | F | T | L24 |
| TC4 | 3 | 0 | 3 | 1 | T | T | F | F | L23 |
| TC5 | 2 | 1 | 2 | 3 | T | F | T | T | L14 |
| TC6 | 2 | 1 | 1 | 2 | T | F | T | F | L14 |
| TC7 | 3 | 1 | 2 | 3 | T | F | F | T | L14 |
| TC8 | 3 | 1 | 2 | 2 | T | F | F | F | L13 |
| | | | | | F | T | T | T | 不存在 |
| | | | | | F | T | T | F | 不存在 |
| TC9 | 1 | 0 | 2 | 3 | F | T | F | T | L14 |
| TC10 | 1 | 0 | 1 | 1 | F | T | F | F | L13 |
| | | | | | F | F | T | T | 不存在 |
| | | | | | F | F | T | F | 不存在 |
| TC11 | 1 | 1 | 2 | 3 | F | F | F | T | L14 |
| TC12 | 1 | 1 | 1 | 1 | F | F | F | F | L13 |



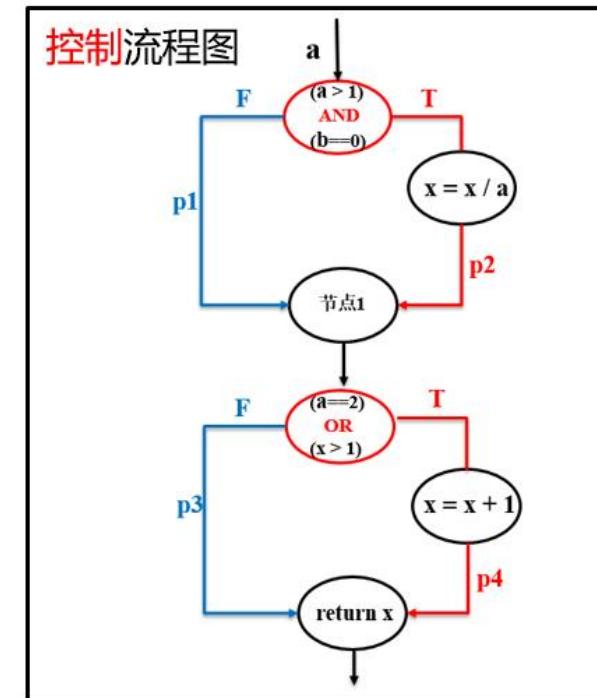
白盒测试：一个例子

(六) 路径覆盖

使程序中每一条可能的路径至少执行一次。该策略是**最强的**，但一般是不可实现的。

设计测试用例，实现路径覆盖：

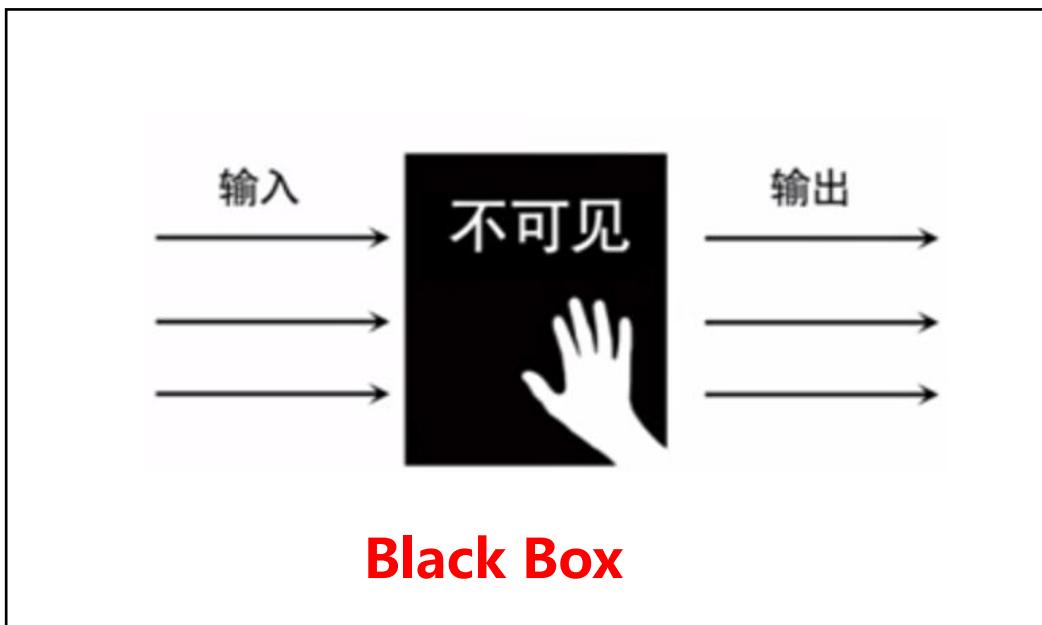
| 测试用例ID | 输入 | | | 预期输出 | 覆盖路径 |
|--------|----|---|---|------|------|
| | a | b | x | | |
| TC1 | 2 | 0 | 4 | 3 | L24 |
| TC2 | 1 | 1 | 1 | 1 | L13 |
| TC3 | 1 | 1 | 2 | 3 | L14 |
| TC4 | 3 | 0 | 3 | 1 | L23 |





黑盒测试：定义

也称作[功能测试](#)或[数据驱动](#)测试，它着眼于程序外部结构，在完全不考虑程序的内部逻辑结构和内部特性的情况下，测试者在[程序接口](#)进行测试，它只检查程序功能是否按照[需求规格说明书](#)的规定正常使用，程序是否能适当的接收[输入](#)数据而产生正确的[输出](#)信息。



优势：

- 方法简单有效
- 可以[整体](#)测试系统行为
- 开发与测试可以[并行](#)
- 对测试人员技术要求相对较低

劣势：

- 覆盖率较低
- 直接依赖于需求规格说明书
- 入门门槛低



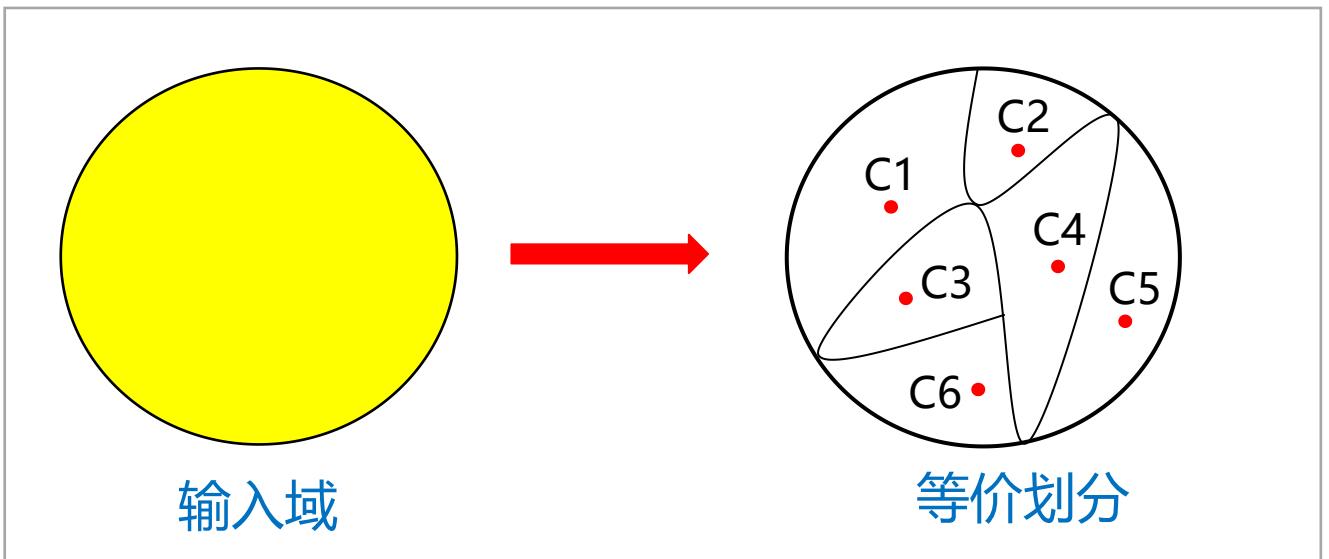
黑盒测试：等价类划分

等价类

输入域的一个子集，在该子集中，各个输入数据对揭示程序中错误是等效的。

等价类测试

将无穷多的数据缩减到有限个等价区域中，通过测试等价区域完成穷尽测试。



- 分而不交
- 合而不变
- 类内等价



黑盒测试：等价类划分

有效等价类

- 指对于软件的规格说明书而言，是**合理的**，**有意义**的输入数据所构成的集合。
- **用于检验被测系统是否能够完成指定功能。**

无效等价类

- 指对于软件的规格说明书而言，是**不合理的**，**没有意义**的输入数据所构成的集合。
- **用于考察被测系统的容错性。**



黑盒测试：等价类划分

划分原则：

- ① 如果输入条件规定了一个取值范围或取值个数，则可确定一个有效等价类和两个无效等价类。（如：数量从1到999）
- ② 如果输入条件规定了一个输入值的集合（假定n个），而且软件要对每个输入值进行不同处理，则可确定n个有效等价类和一个无效等价类。
- ③ 如果存在输入条件规定了“必须是”的情况，则可确定一个有效等价类和一个无效等价类。（如：标识符第一个字符必须是字母）
- ④ 如果输入的是布尔表达式，则可确定一个有效等价类和一个无效等价类。



黑盒测试：一个例子

系统某查询条件是 1990 年 1 月到 2019 年 12 月，由 6 位数字表示，前 4 位表示年份，后 2 位表示月份。

- 确定有效等价类和无效等价类：

| 输入条件 | 有效等价类 | | 无效等价类 | |
|--------------|-------|-----------|-------|----------|
| 日期的类型 和长度 | C1 | 6位数字字符 | C2 | 有非数字字符 |
| | | | C3 | 少于6位数字字符 |
| | | | C4 | 多于6位数字字符 |
| 年份范围 | C5 | 1990~2019 | C6 | 小于1990 |
| | | | C7 | 大于2019 |
| 月份范围 | C8 | 01~12 | C9 | 等于00 |
| | | | C10 | 大于12 |



黑盒测试：一个例子

- 编写测试用例覆盖所有等价类

➤ 覆盖**有效**等价类

| 测试数据 | 期望结果 | 覆盖的有效等价类 |
|--------|------|----------|
| 201408 | 输入有效 | C1、C5、C8 |

➤ 覆盖**无效**等价类（单缺陷原则）

| 测试数据 | 期望结果 | 覆盖的无效等价类 |
|----------|------|----------|
| aaa123 | 无效输入 | C2 |
| 2011 | 无效输入 | C3 |
| 20140408 | 无效输入 | C4 |
| 187905 | 无效输入 | C6 |
| 209811 | 无效输入 | C7 |
| 200100 | 无效输入 | C9 |
| 200156 | 无效输入 | C10 |



黑盒测试：边界值分析

是一种**常用的**黑盒测试技术。它倾向于选择**系统边界或边界附近**的数据来设计测试用例，考虑了边界条件的测试用例具有更高的测试回报率。

基本原理

经过长期的测试工作经验表明，大量的缺陷经常发生在**输入域或输出域的边界上**。因此设计一些测试用例，使程序运行在**边界情况附近**，这样揭露错误的可能性比较大。



黑盒测试：边界值分析

边界点：

指等价类中那些恰好处于边界的点，可能导致被测系统内部处理机制发生变化的点。

- 举个例子

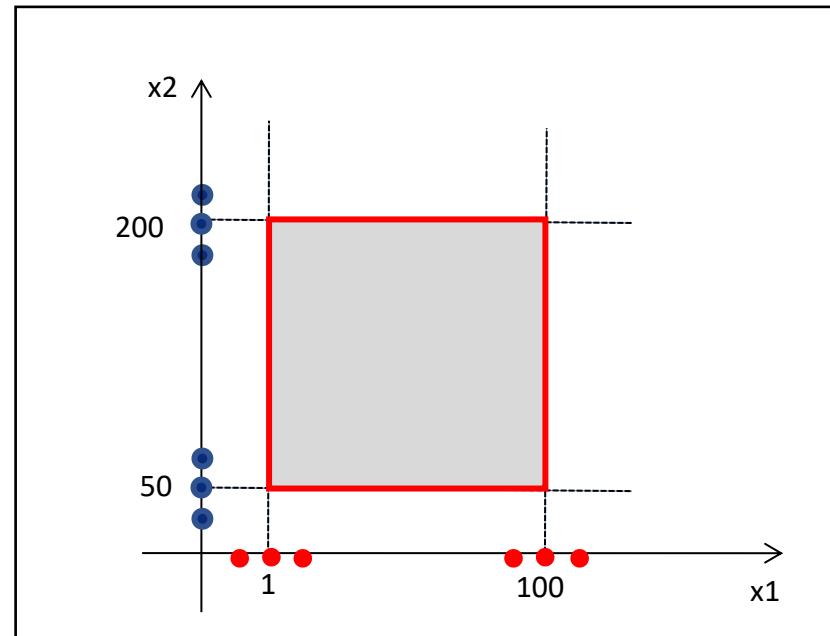
```
int Add (int x1, int x2)
```

$1 \leq x1 \leq 100$

$50 \leq x2 \leq 200$

- 需求说明：

- 对于有效输入，函数返回 $x1$ 与 $x2$ 的和；
- 对于无效输入，函数返回-1。



找到每个输入条件的边界点，即可找到系统边界。

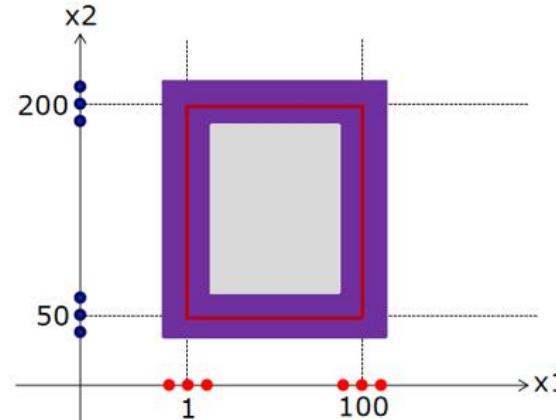


黑盒测试：边界值分析

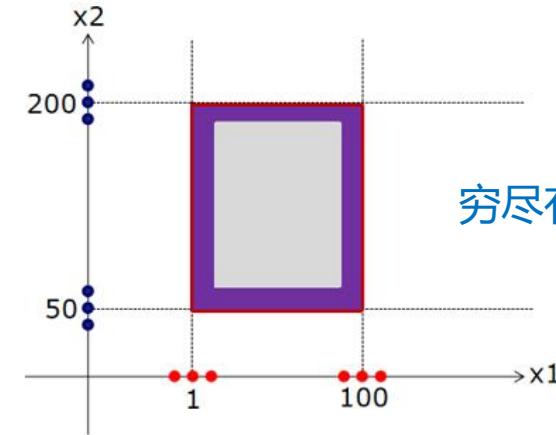
组织测试数据设计测试用例：

用例评价标准：数量、覆盖度、冗余度、缺陷定位能力、复杂度。

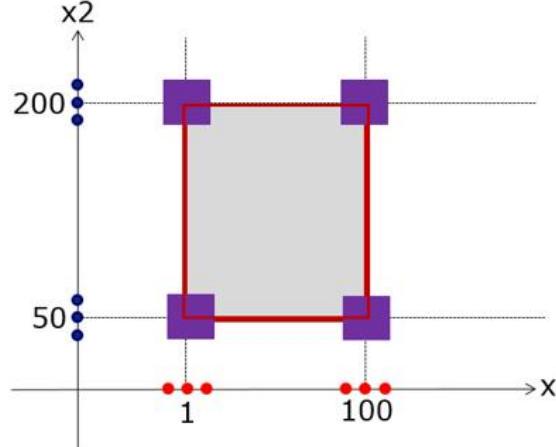
穷尽测试



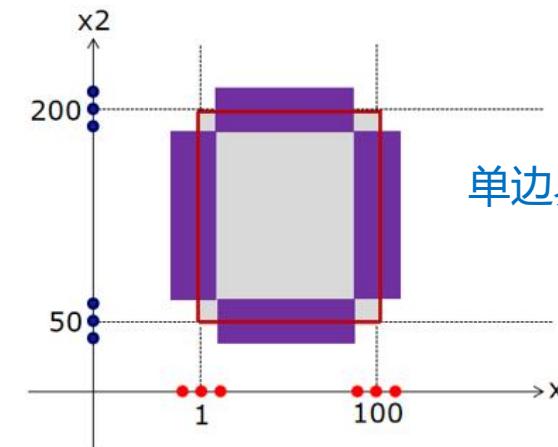
穷尽有效域测试



多边界测试



单边界测试





黑盒测试：边界值分析

组织测试数据设计测试用例：

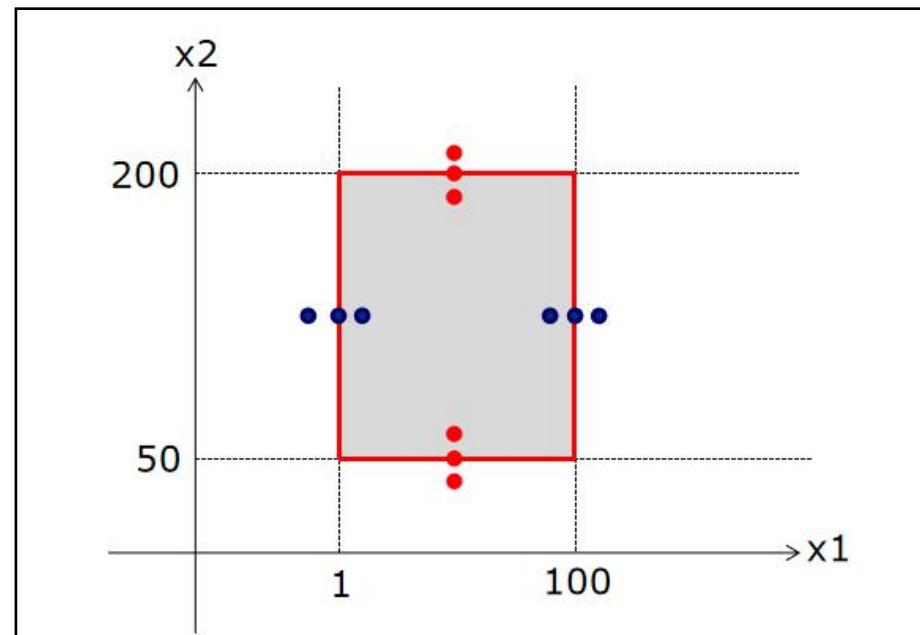
- 最终方案：优化后的单边界

```
int Add ( int x1, int x2 )
```

$1 \leq x1 \leq 100$

$50 \leq x2 \leq 200$

- 用例数量：12
- 覆盖度：100%
- 冗余度：无
- 缺陷定位能力：容易
- 复杂度：低



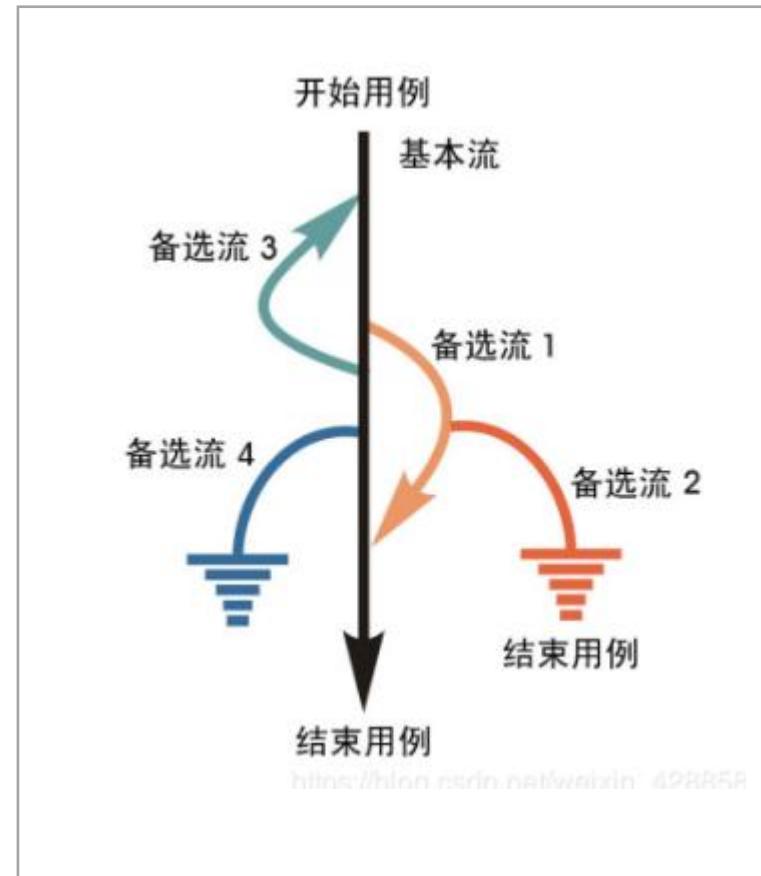


黑盒测试：场景法

以事件流为核心，主要目的是测试软件的主要业务流程、主要功能的正确性和异常处理能力。

测试步骤：

- 定义基本流和备选流；
- 定义场景；
- 从场景设计测试用例；
- 输入测试数据，完善测试用例。





黑盒测试：一个例子 – 定义基本流和备选流

举个例子：**ATM取款**

| 基本流 (正确取款) | |
|------------|-----------|
| 1 | 插入银行卡 |
| 2 | 验证银行卡 |
| 3 | 输入密码 |
| 4 | 验证密码 |
| 5 | 进入ATM机主界面 |
| 6 | 取款并选择金额 |
| 7 | ATM机验证 |
| 8 | 更新账户余额并出钞 |
| 9 | 返回主界面 |

基本流

| 备选流 (出错环节) | |
|------------|----------|
| 1 | 银行卡错误 |
| 2 | 密码错误 |
| 3 | 密码3次错误 |
| 4 | 卡内余额不足 |
| 5 | 超出当日可取 |
| 6 | ATM机余额不足 |
| | |
| | |
| | |

备选流

黑盒测试：一个例子 – 定义场景

➤ 定义场景并设计测试用例

| 用例编号 | 场景 | 银行卡号 | 密码 | 账户余额 | 当日可取 | ATM余额 | 预期结果 |
|------|-----------------------|------|-----|------|------|-------|---------|
| 1 | 场景1：成功取款 | V | V | V | V | V | 成功取款 |
| 2 | 场景2：银行卡无效 | I | N/A | N/A | N/A | N/A | 提示错误、退卡 |
| 3 | 场景3：密码错误 | V | I | N/A | N/A | N/A | 提示错误 |
| 4 | 场景4：密码3次错误 | V | I | N/A | N/A | N/A | 提示错误、吞卡 |
| 5 | 场景5：账户余额不足 | V | V | I | N/A | N/A | 提示错误、退卡 |
| 6 | 场景6：总取款金额超出 当日可取限额 | V | V | V | I | N/A | 提示错误、退卡 |
| 7 | 场景7：ATM机余额不足 | V | V | V | V | I | 提示错误、退卡 |

注：V 表示有效的数值；I 表示无效的数值；N/A表示不适用。



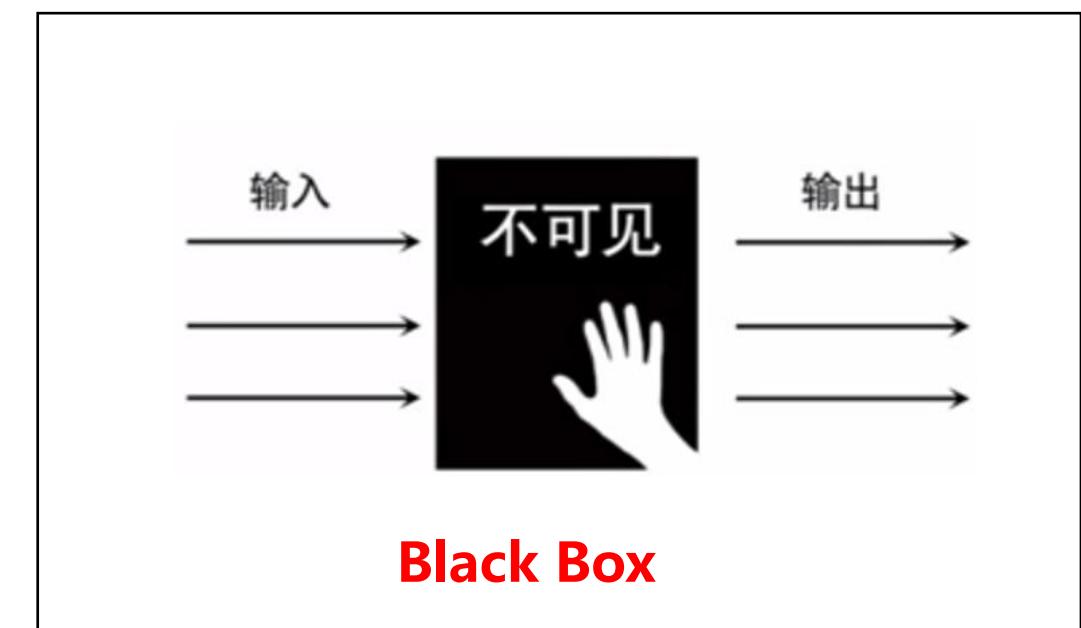
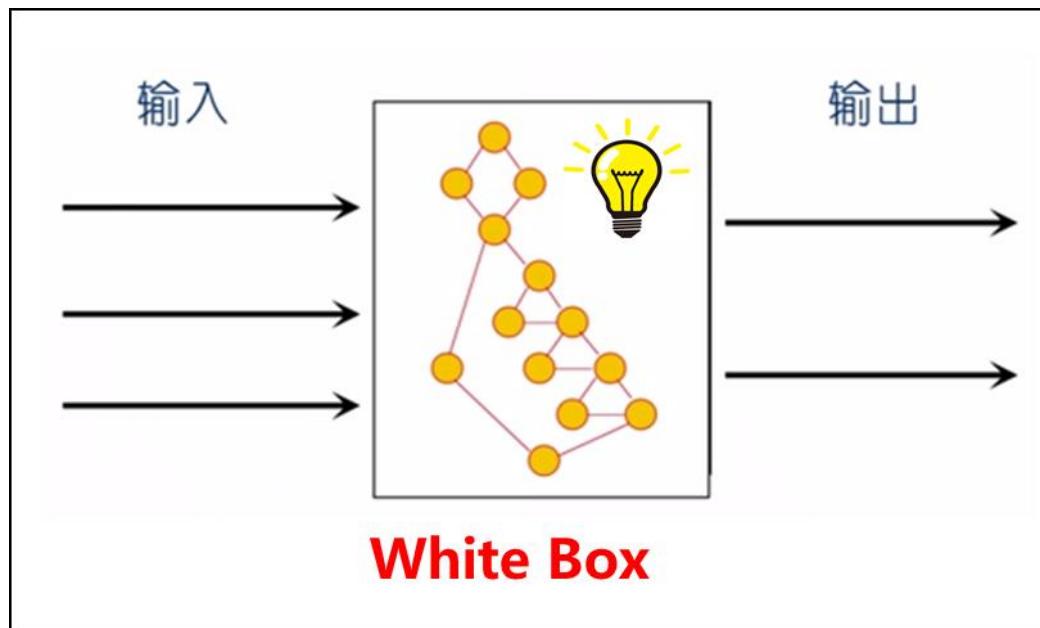
黑盒测试：一个例子 – 设计测试用例

➤ 输入测试数据，完善测试用例

| 用例编号 | 对应场景 | 前提条件 | 操作步骤 | 预期结果 |
|------|------|---|---|---|
| C1 | 场景1 | ① ATM机余额5000元 ② 银行卡号：1234567812345678 ③ 密码：123456 ④ 卡内余额：2000元 | ① 插入银行卡 ② 输入正确密码 ③ 进入主页后选择取款1000元 | ① ATM机输出1000元，提示用户取走并返回主界面 ② ATM余额4000元 ③ 用户卡内余额1000元 |
| C2 | 场景2 | ATM机余额4000元 | 插入无效银行卡 | 提示银行卡无效并退卡 |
| C3 | 场景3 | ① ATM机余额4000元 ② 银行卡号：1234567812345678 ③ 密码：123456 ④ 卡内余额：1000元 | ① 插入银行卡 ② 输入错误密码：654321 | 提示密码错误，并清空密码 |
| C4 | 场景3 | ① ATM机余额4000元 ② 银行卡号：1234567812345678 ③ 密码：123456 ④ 卡内余额：1000元 | 在C3基础上再次输入错误密码： 987654 | 提示密码错误，并清空密码 |
| C5 | 场景4 | ① ATM机余额4000元 ② 银行卡号：1234567812345678 ③ 密码：123456 ④ 卡内余额：1000元 | 在C4基础上再次输入错误密码： 876543 | 提示密码错误，并没收该卡 |

对比总结

- 白盒测试（结构测试）：主要检测软件编码过程中的错误。
- 黑盒测试（功能测试）：主要检测功能是否能够按照规格说明书正常使用。



- 对测试人员要求高
- 发现代码中的问题

- 相对简单
- 从用户角度测试功能



单元测试 (Unit Testing)

是代码层面的测试，是对软件中的**最小可测试单元**进行检查和验证。单元测试，顾名思义是测试一个“单元”，有别于集成测试，这个“单元”一般是类或函数，而不是模块或者系统。

单元测试可看作是编码工作的一部分，应该由**程序员**完成，经过了单元测试的代码才是已完成的代码，提交产品代码时也要同时提交测试代码。



单元测试和JUnit

JUnit：一个Java语言的单元测试框架

- 由Kent Beck（极限编程）和 Erich Gamma（设计模式）建立的。
- 是xUnit家族中最成功的一个。
- 大部分的Java IDE都集成了JUnit作为单元测试工具。
- 官网：<http://www.junit.org>



工欲善其事必先利其器！



单元测试和JUnit

传统测试方法：使用main函数

```
1 package com.junit.main;
2
3 public class Calculator {
4
5     public int add(int a, int b) {
6
7         return a + b;
8     }
9
10    public int subtract(int a, int b) {
11
12        return a - 1; // Red box and arrow here
13    }
14
15    public int multiply(int a, int b) {
16
17        return a * b;
18    }
19
20    public int divide(int a, int b) {
21
22        return a / b;
23    }
24
25
26 }
```

待测方法

```
1 package com.junit.main;
2
3 public class Calculator {
4     public int add(int a, int b) {
5         return a + b;
6     }
7
8     public int subtract(int a, int b) {
9         return a - 1;
10    }
11
12    public int multiply(int a, int b) {
13        return a * b;
14    }
15
16    public int divide(int a, int b) {
17        return a / b;
18    }
19
20    public static void main(String[] args)
21    {
22        Calculator calculator = new Calculator();
23        System.out.println(3 == calculator.add(1, 2));
24        System.out.println(1 == calculator.subtract(3, 2));
25        System.out.println(2 == calculator.multiply(1, 2));
26        System.out.println(1 == calculator.divide(2, 2));
27    }
28
29
30 }
```

Console output:

```
Problems @ Javadoc Declaration Console
<terminated> Calculator [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\javaw
true
false
true
true
```



单元测试和JUnit

The screenshot shows a Java development environment with two main windows:

Left Window (Code Editor): Displays the `CalculatorTest.java` file. The code defines a test class `CalculatorTest` with four test methods: `testAdd()`, `testSubtract()`, `testMultiply()`, and `testDivide()`. The `testDivide()` method is annotated with `//@Test(expected=ArithmeticException.class)`. The code editor has red boxes highlighting the `testSubtract()` and `testDivide()` methods.

```
1 package com.junit.main;
2
3 import static org.junit.Assert.*;
4
5 import org.junit.Before;
6 import org.junit.Test;
7
8 public class CalculatorTest {
9     private Calculator calculator;
10
11     @Before
12     public void setUp() throws Exception {
13         calculator = new Calculator();
14     }
15
16     @Test
17     public void testAdd() {
18         assertEquals(3,calculator.add(1, 2));
19     }
20
21     @Test
22     public void testSubtract() {
23         assertEquals(1,calculator.subtract(3, 2));
24     }
25
26     @Test
27     public void testMultiply() {
28         assertEquals(2,calculator.multiply(1, 2));
29     }
30
31     //@Test(expected=ArithmeticException.class)
32     @Test
33     public void testDivide() {
34         assertEquals(1,calculator.divide(2, 0));
35     }
36 }
```

Right Window (Package Explorer): Shows the execution results of the tests. It indicates 4 runs, 1 error, and 1 failure. The error is in the `testDivide()` method of the `CalculatorTest` class. The failure is also in the same method. The execution took 0.02 seconds.

Finished after 0.02 seconds

Runs: 4/4 Errors: 1 Failures: 1

com.junit.main.CalculatorTest [Runner: JUnit 4] (0.000 s)

- testAdd (0.000 s)
- testSubtract (0.000 s) **Failed**
- testDivide (0.000 s) **Failed**
- testMultiply (0.000 s)

Below this, another execution result is shown where all tests pass successfully (0 errors, 0 failures) in 0.019 seconds.

Finished after 0.019 seconds

Runs: 4/4 Errors: 0 Failures: 0

com.junit.main.CalculatorTest [Runner: JUnit 4] (0.000 s)

- testAdd (0.000 s)
- testSubtract (0.000 s)
- testDivide (0.000 s)
- testMultiply (0.000 s)

Keeps the bar green to keeps the code clean !

—JUnit格言



性能测试 (Performance testing)

通常收集所有和测试有关的性能，被不同人在不同场合下进行使用。 性能测试是一种“正常”测试，主要测试使用时系统是否满足要求，同时可能为了保留系统的扩展空间而进行的一些稍稍超过“正常”范围的测试。

负载测试，压力测试

性能测试工具:Load impact

<http://loadimpact.com/>
<https://k6.io/>



压力测试 (Stress Testing)

压力测试是性能测试的一种，它的目标是测试在一定的负载下系统长时间运行的稳定性，但是这个负载不一定是应用系统本身造成的。压力测试尤其关注大业务量情况下长时间运行系统性能的变化（例如是否反应变慢、是否会内存泄漏导致系统逐渐崩溃、是否能恢复）。

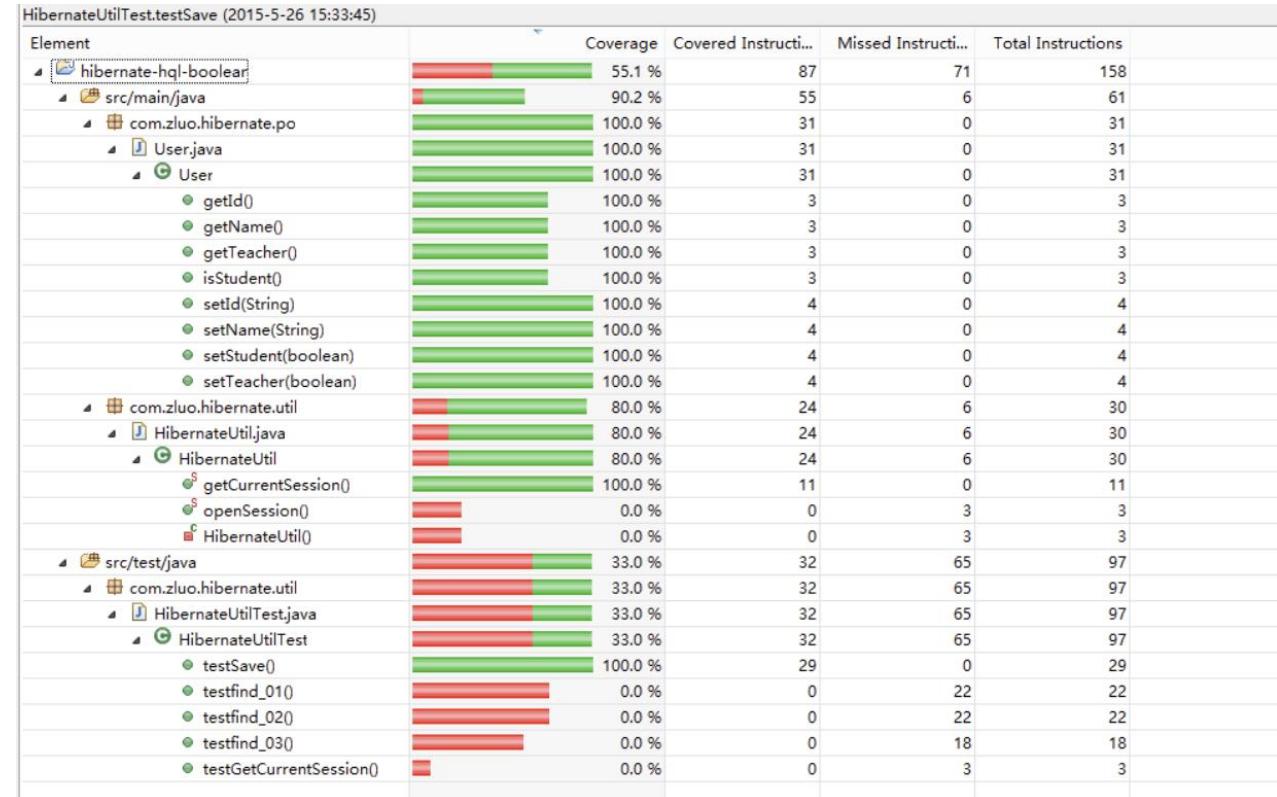
压力测试工具:JMeter
官网: <https://jmeter.apache.org>





代码覆盖率测试

- 代码覆盖率 = 代码的覆盖程度，一种度量方式
 - 语句覆盖(StatementCoverage)
 - 判定覆盖(DecisionCoverage)
 - 条件覆盖(ConditionCoverage)
 - 路径覆盖(PathCoverage)



代码覆盖率测试工具:JaCoCo 官网: <https://www.jacoco.org/jacoco/trunk/index.html>



代码质量

软件质量，不但依赖于架构及项目管理，而且与**代码质量**紧密相关。代码质量是软件开发不可或缺的一部分。



Bad Code



Good Code



如何评价代码质量

1. 可维护性 (Maintainability)

- 在不破坏原有代码设计、不引入新的 bug 的情况下，是否能够快速地修改或者添加代码。

2. 可读性 (Readability)

- 代码是否符合编码规范、命名是否达意、注释是否详尽、函数是否长短合适、模块划分是否清晰、是否符合高内聚低耦合等。

3. 可扩展性 (Extensibility)

- 在不修改或少量修改原有代码的情况下，是否可以通过扩展的方式添加新的功能代码。

如何评价代码质量

4. 可复用性 (Reusability)

- 代码中排名第一的坏味道就是代码重复。 ——Kent Benk 和 Martin Fowler
- 应该编写可重用的、通用的代码，复用已有的代码。

5. 可测试性 (Testability)

- 是否易于针对代码编写单元测试；
- 代码可测试性差、单元测试覆盖率低通常意味着代码设计得不够合理。

6. 简洁性 (Simplicity)

- KISS 原则：“Keep It Simple, Stupid”；
- 尽量保持代码简单、逻辑清晰。



如何评价代码质量

1. 可维护性 (Maintainability)

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class OrderProcessor {
    // 使用日志框架，配置集中（通常在外部文件）
    private static final Logger logger = LoggerFactory.getLogger(OrderProcessor.class);

    public void processOrder(Order order) {
        // ... 业务逻辑 ...
        logger.info("Order processed: {}", order.getId()); // 统一使用logger
        // ... 更多业务逻辑 ...
        logger.warn("Low stock for item {}", order.getItemId()); // 统一使用logger
    }
}
```

```
public class OrderProcessor {

    public void processOrder(Order order) {
        // ... 业务逻辑 ...
        // 日志记录硬编码在业务逻辑中
        System.out.println("Order processed: " + order.getId() + " at " + new Date());
        // ... 更多业务逻辑 ...
        // 另一个地方的日志格式不同
        System.err.println("WARNING: Low stock for item " + order.getItemId());
    }
}
```





如何评价代码质量

2. 可读性 (Readability)

```
public boolean isValidUser(User user) {  
    if (user == null) {  
        return false;  
    }  
    // 使用卫语句提前返回，减少嵌套  
    if (user.getName() == null || user.getName().isEmpty()) {  
        return false;  
    }  
    final int MIN_AGE = 0; // 常量替代魔法数字  
    final int MAX_AGE = 150;  
    if (user.getAge() <= MIN_AGE || user.getAge() >= MAX_AGE) {  
        return false;  
    }  
    return true;  
}
```

```
public boolean vld(Usr u) {  
    if (u != null) {  
        if (u.getNm() != null && !u.getNm().isEmpty()) {  
            if (u.getAge() > 0 && u.getAge() < 150) { // 魔法数字 0, 150  
                return true;  
            } else {  
                return false;  
            }  
        } else {  
            return false;  
        }  
    } else {  
        return false;  
    }  
}
```





如何评价代码质量

3. 可扩展性 (Extensibility) 4. 可复用性 (Reusability)

```
public class PaymentProcessor {  
    public void processPayment(Payment payment) {  
        if ("CREDIT_CARD".equals(payment.getType())) {  
            // 处理信用卡支付的详细逻辑...  
        } else if ("PAYPAL".equals(payment.getType())) {  
            // 处理PayPal支付的详细逻辑...  
        } else if ("BANK_TRANSFER".equals(payment.getType())) {  
            // 处理银行转账的详细逻辑...  
        } else {  
            throw new IllegalArgumentException("Unsupported payment type");  
        }  
    }  
}
```

```
public interface PaymentStrategy {  
    void processPayment(double amount);  
}  
  
public class CreditCardStrategy implements PaymentStrategy {  
    @Override  
    public void processPayment(double amount) {  
        // 信用卡支付具体实现  
    }  
}  
  
public class PayPalStrategy implements PaymentStrategy {  
    @Override  
    public void processPayment(double amount) {  
        // PayPal支付具体实现  
    }  
}  
  
public class BankTransferStrategy implements PaymentStrategy {  
    @Override  
    public void processPayment(double amount) {  
        // 银行转账具体实现  
    }  
}  
  
public class PaymentProcessor {  
    private PaymentStrategy strategy;  
  
    public PaymentProcessor(PaymentStrategy strategy) {  
        this.strategy = strategy;  
    }  
  
    public void processPayment(double amount) {  
        strategy.processPayment(amount);  
    }  
}
```





如何评价代码质量

5. 可测试性 (Testability)

```
public class OrderService {  
    private final InventoryService inventoryService; // 依赖接口  
  
    // 通过构造器注入依赖  
    public OrderService(InventoryService inventoryService) {  
        this.inventoryService = inventoryService;  
    }  
  
    public boolean placeOrder(Order order) {  
        // ... 其他逻辑  
        boolean inStock = inventoryService.checkStock(order.getItemId(), order.getQuantity());  
        if (!inStock) {  
            return false;  
        }  
        // ... 扣减库存等操作  
        return true;  
    }  
}
```

```
public class OrderService {  
    private InventoryService inventoryService = new InventoryServiceImpl(); // 紧耦合, 直接new  
  
    public boolean placeOrder(Order order) {  
        // ... 其他逻辑  
        boolean inStock = inventoryService.checkStock(order.getItemId(), order.getQuantity());  
        if (!inStock) {  
            return false;  
        }  
        // ... 扣减库存等操作  
        return true;  
    }  
}
```





如何评价代码质量

6. 简洁性 (Simplicity)

```
public String getFullName(User user) {
    if (user == null) {
        return "";
    }
    return String.join(" ",
        Optional.ofNullable(user.getFirstName()).orElse(""),
        Optional.ofNullable(user.getLastName()).orElse(""))
        .trim();
}
```

// 或者更简洁 (如果User的get方法保证不返回null, 或者用Objects.requireNonNullElse)

```
public String getFullNameSimpler(User user) {
    if (user == null) return "";
    return (user.getFirstName() + " " + user.getLastName()).trim();
    // 注意: 如果firstName为null, 会变成 "null LastName", 需要确保get方法不返回null或用Objects.requireNonNullElse
}
```

```
public String getFullName(User user) {
    String fullName = "";
    if (user != null) {
        String firstName = user.getFirstName();
        String lastName = user.getLastName();
        if (firstName != null) {
            fullName = firstName;
        }
        if (lastName != null) {
            if (!fullName.isEmpty()) {
                fullName = fullName + " " + lastName;
            } else {
                fullName = lastName;
            }
        }
    }
    return fullName;
}
```





如何提高代码质量

1. 严格遵循编码规范

阿里巴巴Java开发手册插件

2. 编写高质量的单元测试

3. 代码审查 (Code Review)

4. 开发未动文档先行

5. 持续重构，重构，重构

下面哪一项关于白盒测试和黑盒测试的描述是错误的？

- A 白盒测试侧重于软件编码中的错误，而黑盒测试侧重于功能错误
- B 白盒测试对测试人员要求高，而黑盒测试相对要求较低
- C 白盒测试的代码覆盖率较低，而黑盒测试的代码覆盖率非常高
- D 白盒测试侧重于开发者角度，而黑盒测试侧重于用户角度

 提交



本讲小结

- 1. 软件测试的定义和分类
- 2. 测试用例的定义和其设计方法
- 3. 白盒测试（定义、控制流程图、逻辑覆盖方法、测试用例设计）
- 4. 黑盒测试（定义、等价类划分、边界值分析、场景法）
- 5. 压力测试，性能测试和代码覆盖率测试
- 6. 代码质量保证