单元测试

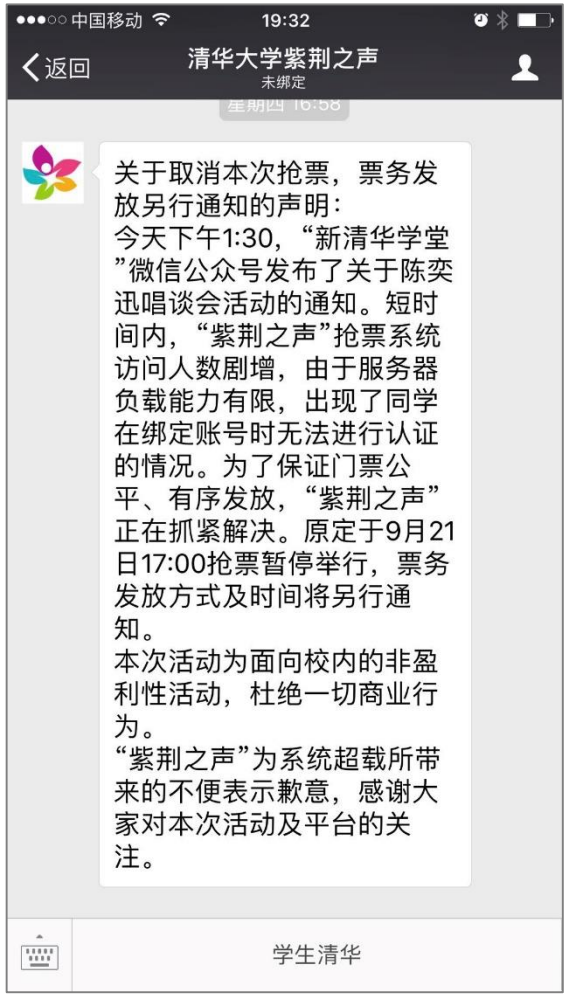
清华大学软件学院 刘强

教学提纲

1. 软件测试概述
2. 黑盒测试方法
3. 白盒测试方法
4. 代码性能分析

关于软件质量的问题

清华大学紫荆之声



各位校领导、有关部门的负责人：

您好，我是计算机系的一位学生。今天，校团委联合新清华学堂组织了 9 月 22 日"陈奕迅唱谈会"的强迫活动，在活动过程中，出现了大量严重的问题，在此予以投诉。

起初，门票的发放采用微信预约的形式。同学们需要填写学号、info 密码、身份证号等个人信息在"清华大学紫荆之声"公众号中绑定微信账号后方可进行抢票。这本是避免冒领的方法，但在实施中出现了如下问题：

1. 站点没有采用学校官方的统一认证形式，而是索要用户名密码，有私自记录的嫌疑。
2. 站点连接到了校外未知来源的服务器，并且使用了不安全的 HTTP 协议进行传输，有被中间人截获的风险，可能泄露同学敏感信息。
3. 站点所要的信息过多（如身份证号），用意不明，且违反相关法规。

在服务器无法承受负荷、微信页面被封、网络学堂禁止了对方服务器的访问后，"清华大学紫荆之声"公众号紧急停止了原定于 17：00 开始的线上抢票。但在 21:01，该公众号出人意料地发送了一条推送，通知需要今晚 22:00 在新清华学堂进行登记后，明天方可领票（可见附件 wechat.png）。这条推送立刻引发了骚乱，大量同学采用飙车、奔跑等方式立刻赶往新清华学堂，使得学堂路交通秩序一片混乱。短时间内新清华学堂前广场挤满了等待登记信息的同学，虽然同学们自发排队，但新清华学堂门前灯光昏暗，场面拥挤不堪，无法得到有效控制。同时领票的同学们将自行车、电动车等在学堂路上任意停放，阻碍了正常的行人、车辆通行。保卫部紧急前来维持秩序，才使得没有发生严重的事故。

这次事件中暴露出的问题有：

1. 团委未能对活动做好有效的计划和预案，在活动开始前一天进行门票分发，时间仓促，安排不周。
2. 选择"突击"推送通知领票，造成学校正常的秩序被扰乱。
3. 未能在进行排队领票等活动时事先通知保卫部等相关部门，也没有工作人员及时维护秩序，造成开始阶段的混乱场面，对排队者的人身安全构成威胁。一旦有人摔倒发生踩踏事故，后果不堪设想。

希望各位领导、负责人能够重视此事，彻查事件发生的原因、严肃处理相关责任人，避免此类事件的再次发生。

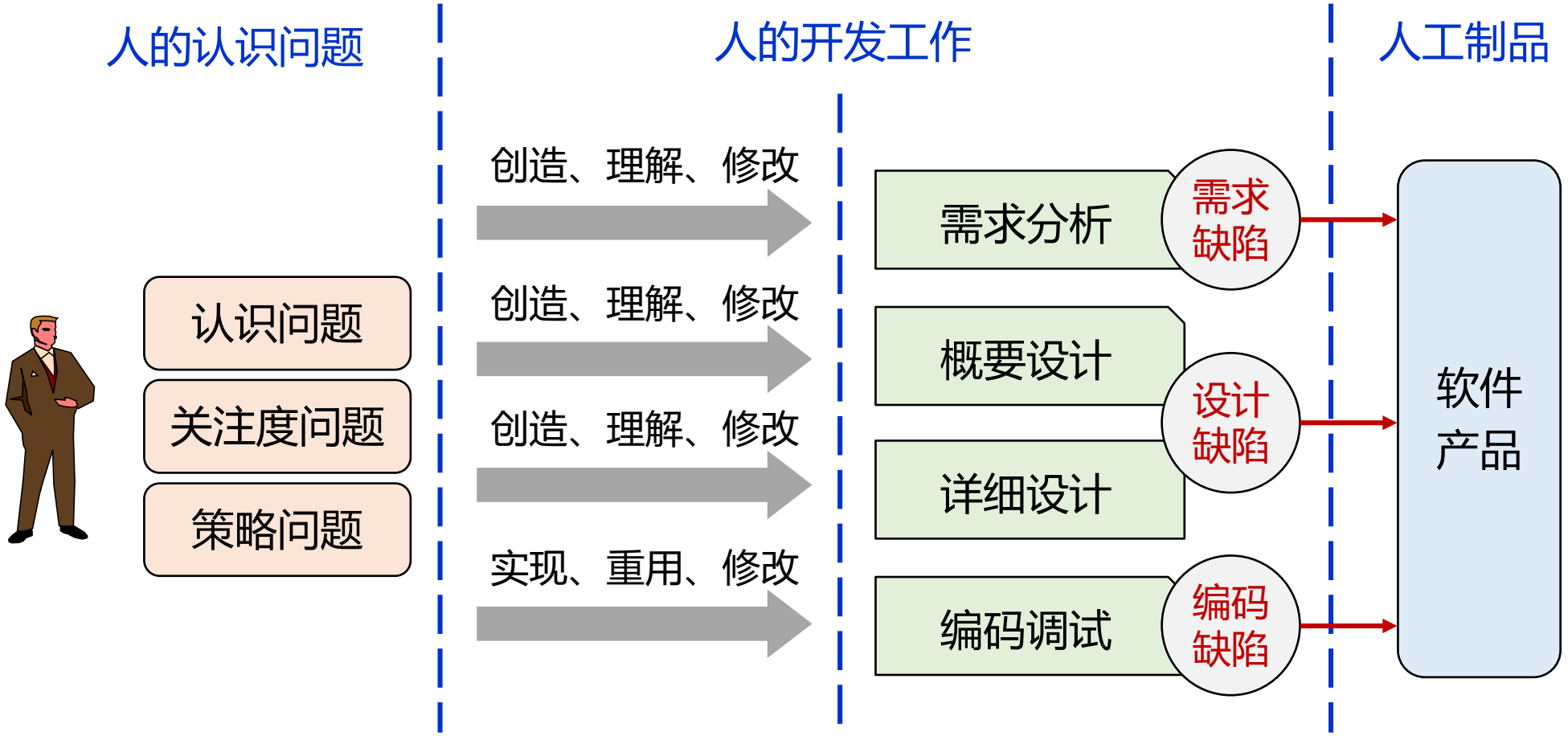
P.S. 明天下午的领票时间依旧可能会出现混乱情况，烦请密切关注此事的进展。

邱校长收到了你的来信，让我们转达对你的感谢。学校已经了解相关情况并进行了相应的处理，同时要求相关部门认真总结，防止类似事件再次发生。

祝：学习愉快！

清华大学校长办公室

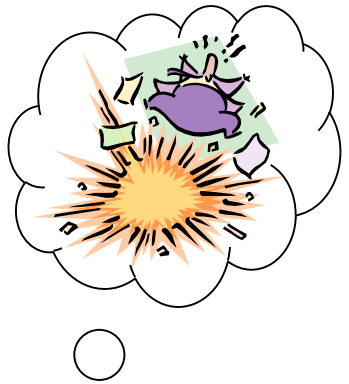
软件缺陷的产生


人的认识问题
人的开发工作
人工制品
认识问题
关注度问题
策略问题
创造、理解、修改
创造、理解、修改
创造、理解、修改
实现、重用、修改
需求分析
概要设计
详细设计
编码调试
需求缺陷
设计缺陷
编码缺陷
软件产品

软件缺陷的演化

举例：爱国者导弹

1991年2月，一次拦截失利
造成28名美国士兵丧生



软件
系统

计时用系统时钟（1/10秒）
并以整数表达

缺少防范措施

缺陷 —激活→ 故障 —演变→ 失效

寄存器存储导致误差
$(0.000000095)_{10}$

$0.000000095 \times 100h \times 60 \times 60 \times 10 = 0.34s$

软件测试

测试 —— 发现问题 —— 修复

软件测试的定义

软件测试的定义

正向思维：验证软件正常工作

- 评价一个程序或系统的特性或能力并确定是否达到预期的结果。
- 在设计规定的环境下运行软件的所有功能，直至全部通过。

逆向思维：假定软件有缺陷

- 测试是为了发现错误而针对某个程序或系统的执行过程。
- 寻找容易犯错地方和系统薄弱环节，试图破坏系统直至找不出问题。

软件测试的目的

直接目标：发现软件错误

- 以最少的人力、物力和时间找出软件中潜在的各种错误和缺陷，通过修正这些错误和缺陷提高软件质量，回避软件发布后由于潜在的软件错误和缺陷造成的隐患所带来的商业风险。

期望目标：检查系统是否满足需求

- 测试是对软件质量的度量和评估，以验证软件的质量满足客户需求的程度，为用户选择和接受软件提供有力的依据。

附带目标：改进软件过程

- 通过分析错误产生的原因，可以帮助发现当前开发所采用的软件过程缺陷，从而进行软件过程改进；通过分析整理测试结果，可以修正软件开发规则，为软件可靠性分析提供依据。

测试的局限性

测试的不彻底性

- 测试只能说明错误的存在，但不能说明错误不存在
- 经过测试后的软件不能保证没有缺陷和错误

测试的不完备性

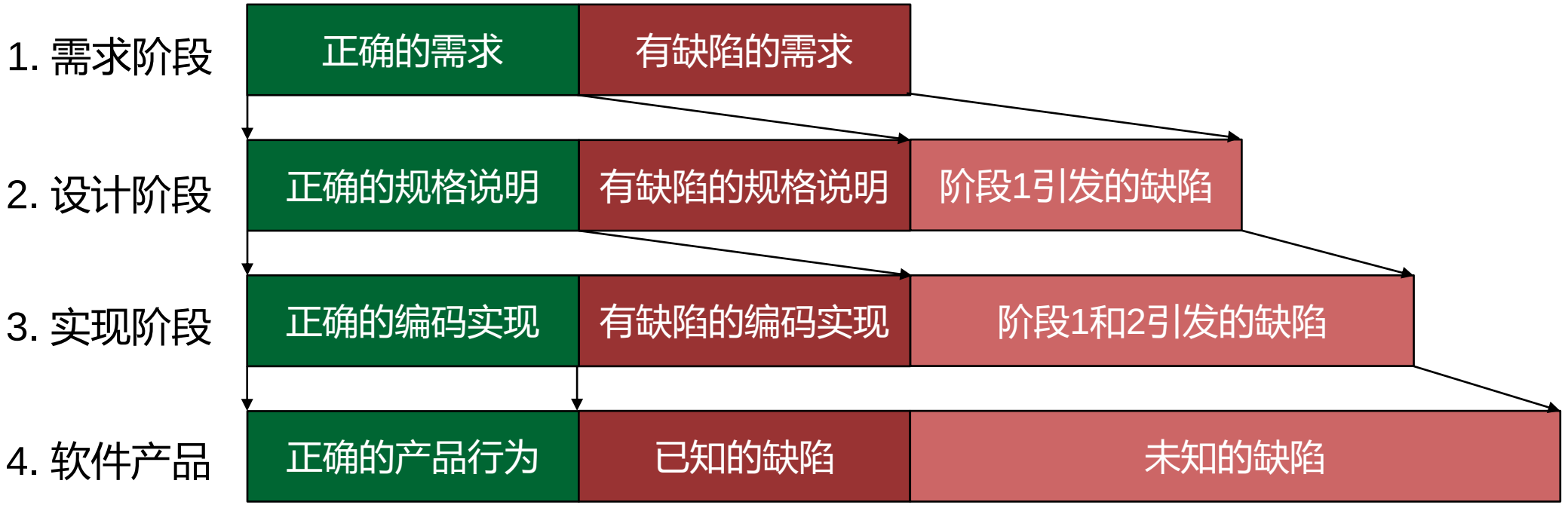
- 测试无法覆盖到每个应该测试的内容
- 不可能测试到软件的全部输入与响应
- 不可能测试到全部的程序分支的执行路径

测试作用的间接性

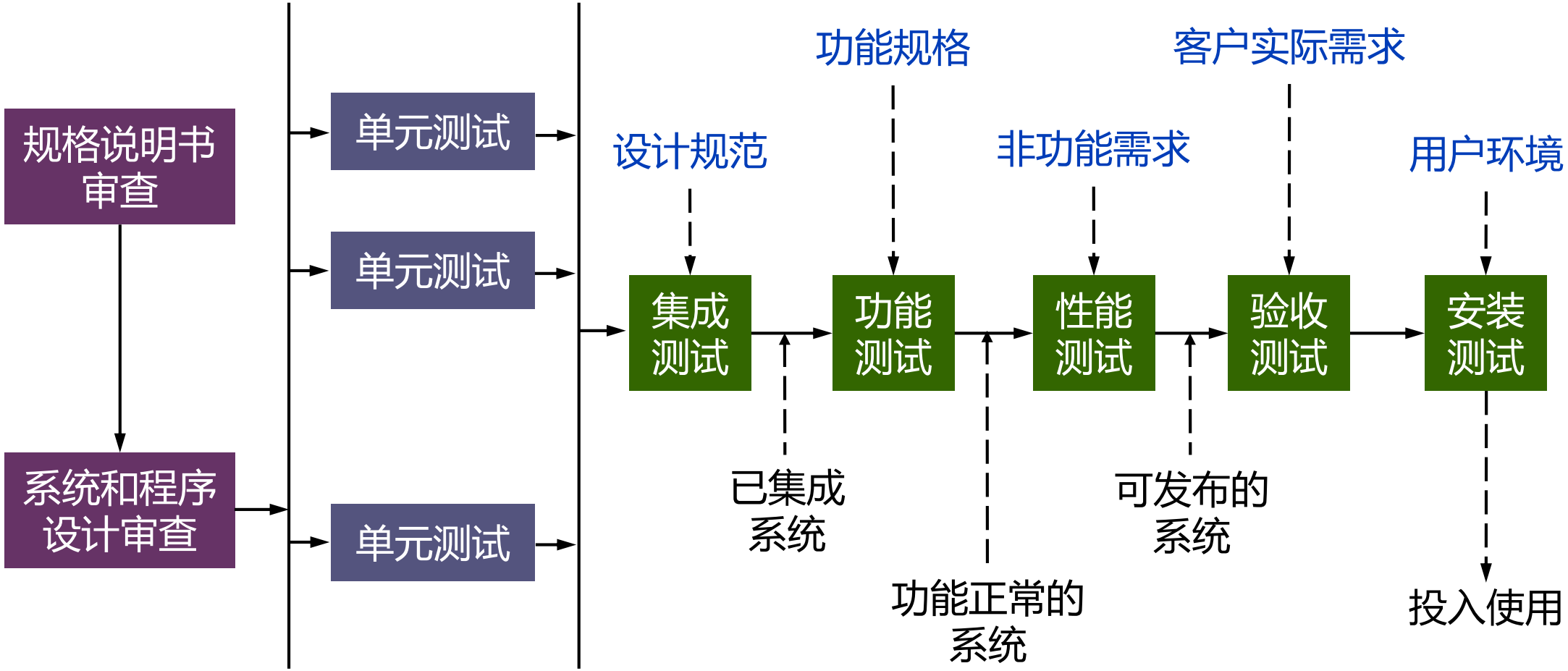
- 测试不能直接提高软件质量，软件质量的提高要依靠开发
- 测试通过早期发现缺陷并督促修正缺陷来间接地提高软件质量

测试应尽早介入

尽早地在缺陷刚引入时就发现和修复，可以有效地避免缺陷的雪崩效应。



软件测试活动


规格说明书审查
系统和程序设计审查
单元测试
单元测试
单元测试
设计规范
功能规格
非功能需求
客户实际需求
用户环境
集成测试
功能测试
性能测试
验收测试
安装测试
已集成系统
功能正常的系统
可发布的系统
投入使用

单元测试

单元测试（Unit Testing）是对软件基本组成单元进行的测试，其测试对象是软件设计的最小单位（模块或者类）。

单元测试一般由编写代码的开发人员执行，用于检测被测代码的功能是否正确。

单元测试

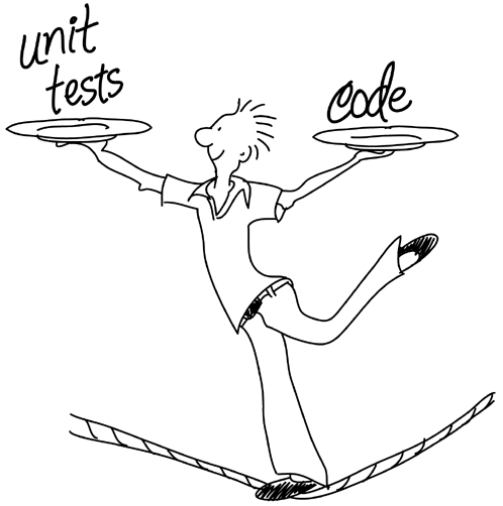
单元测试（Unit Testing）是对软件中的最小可测试单元进行检查和验证。

单元测试

程序员必须对自己的代码质量负责，单元测试是对自己代码质量的基本承诺。

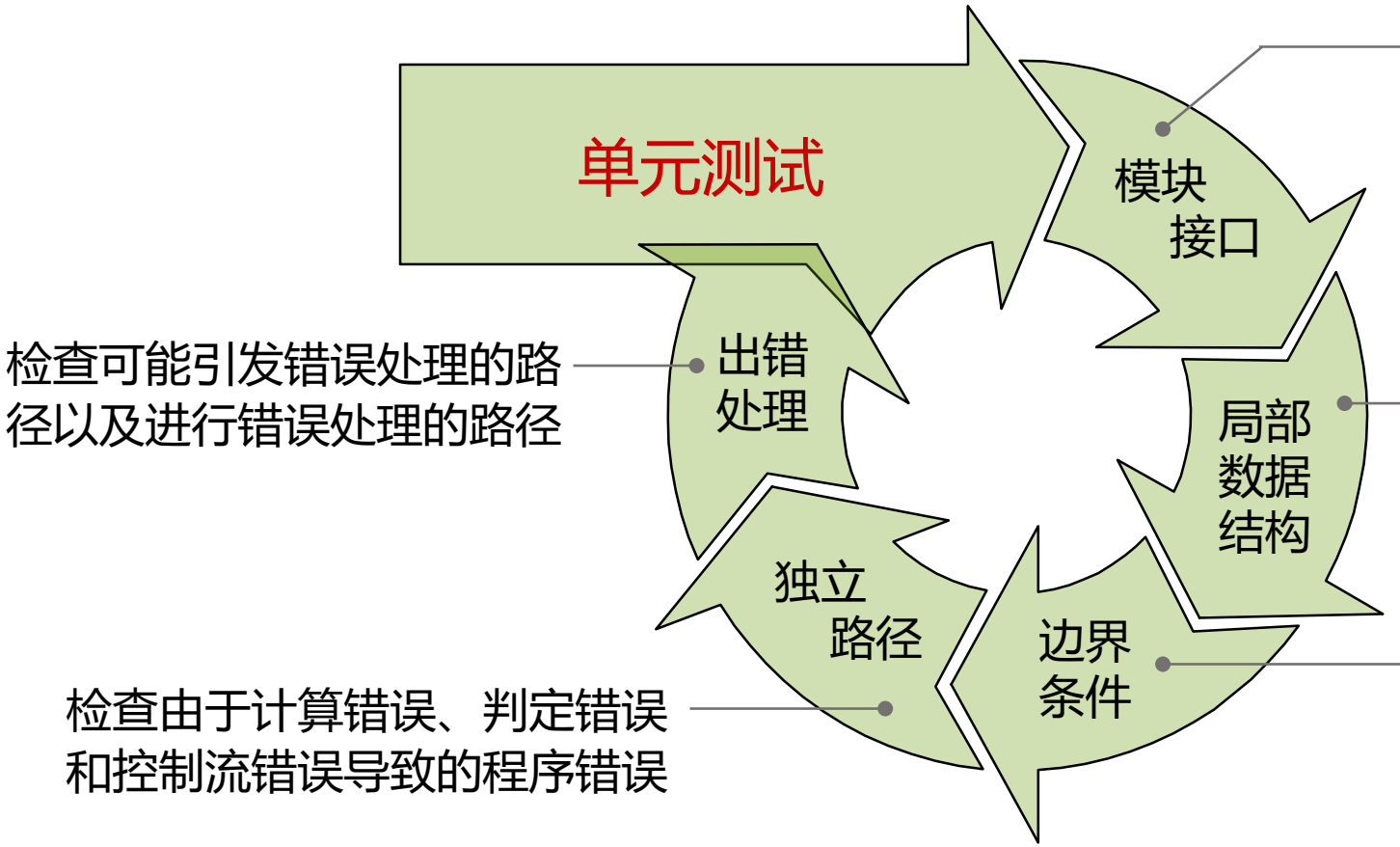
程序 = UT + CODE

测试人员有权利对没有做过UT的代码说No



在现实中，代码质量最好、开发速度最快的程序员是单元测试做得最好的。

单元测试内容



对通过所有被测模块的
数据流进行测试

检查模块中的数据结构
是否正确的定义和使用

检查数据流或控制流中条件或
数据处于边界时的出错可能性

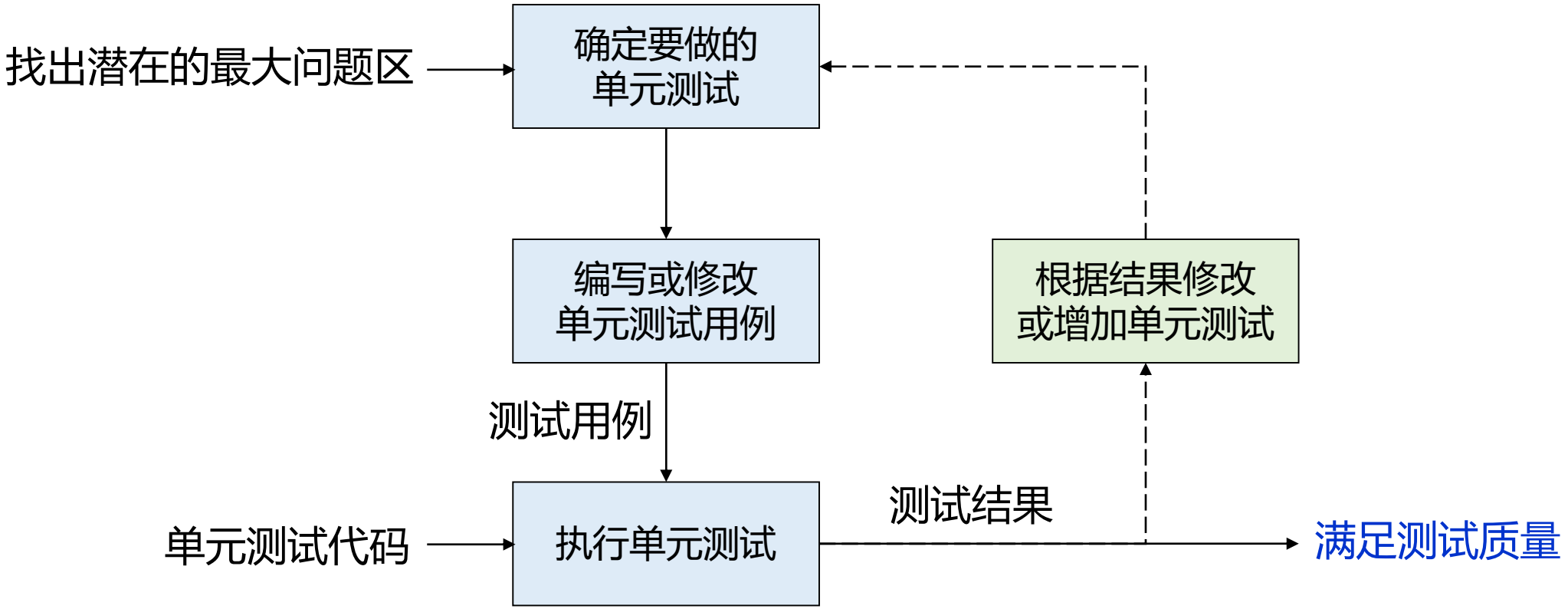
检查由于计算错误、判定错误
和控制流错误导致的程序错误

检查可能引发错误处理的路
径以及进行错误处理的路径

单元测试原则

快速的	单元测试应能快速运行，如果运行缓慢，就不会愿意频繁运行它。
独立的	单元测试应相互独立，某个测试不应为下一个测试设定条件。当测试相互依赖时，一个没通过就会导致一连串的失败，难以定位问题。
可重复的	单元测试应该是可以重复执行的，并且结果是可以重现的。
自我验证的	单元测试应该有布尔输出，无论是通过或失败，不应该查看日志文件或手工对比不同的文本文件来确认测试是否通过。
及时的	及时编写单元测试代码，应恰好在开发实际的单元代码之前。

单元测试过程


找出潜在的最大问题区
确定要做的
单元测试
编写或修改
单元测试用例
测试用例
单元测试代码
执行单元测试
测试结果
根据结果修改
或增加单元测试
满足测试质量

单元测试质量

测试通过率是指在测试过程中执行通过的测试用例所占比例，单元测试通常要求测试用例通过率达到100%。

测试覆盖率是用来度量测试完整性的一个手段，通过覆盖率数据，可以了解测试是否充分以及弱点在哪里。代码覆盖率是单元测试的一个衡量标准，但也不能一味地去追求覆盖率。

单元测试质量

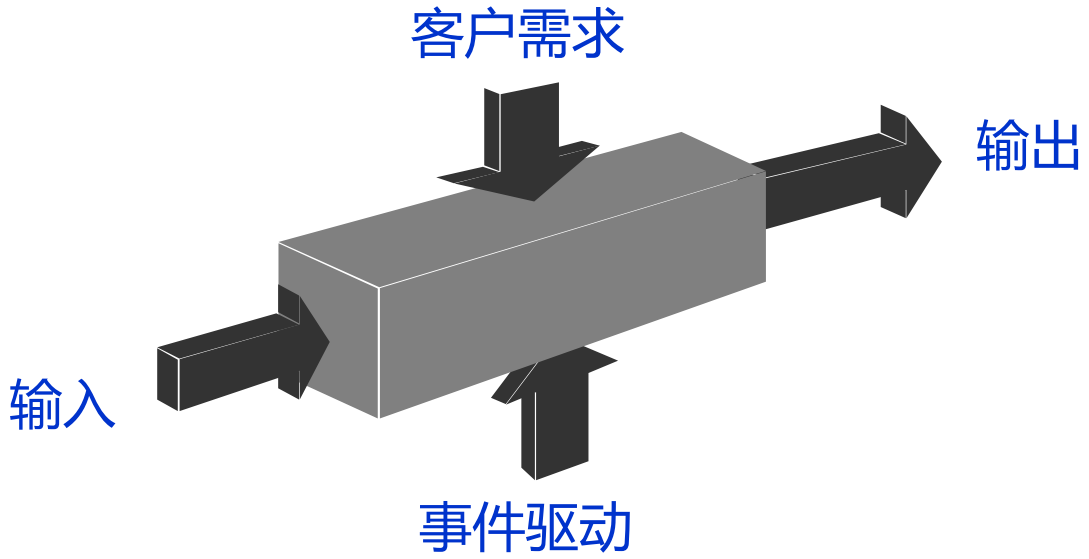
单元测试方法

静态测试：通过人工分析或程序正确性证明的方式来确认程序正确性。

动态测试：通过动态分析和程序测试等方法来检查和确认程序是否有问题。

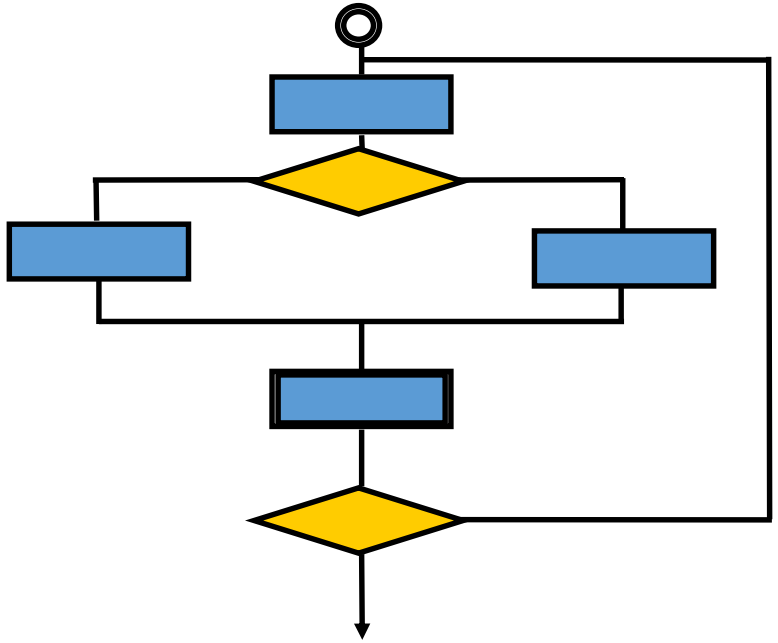
单元测试方法

黑盒测试（Black Box Testing）：又称功能测试，它将测试对象看做一个黑盒子，完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。

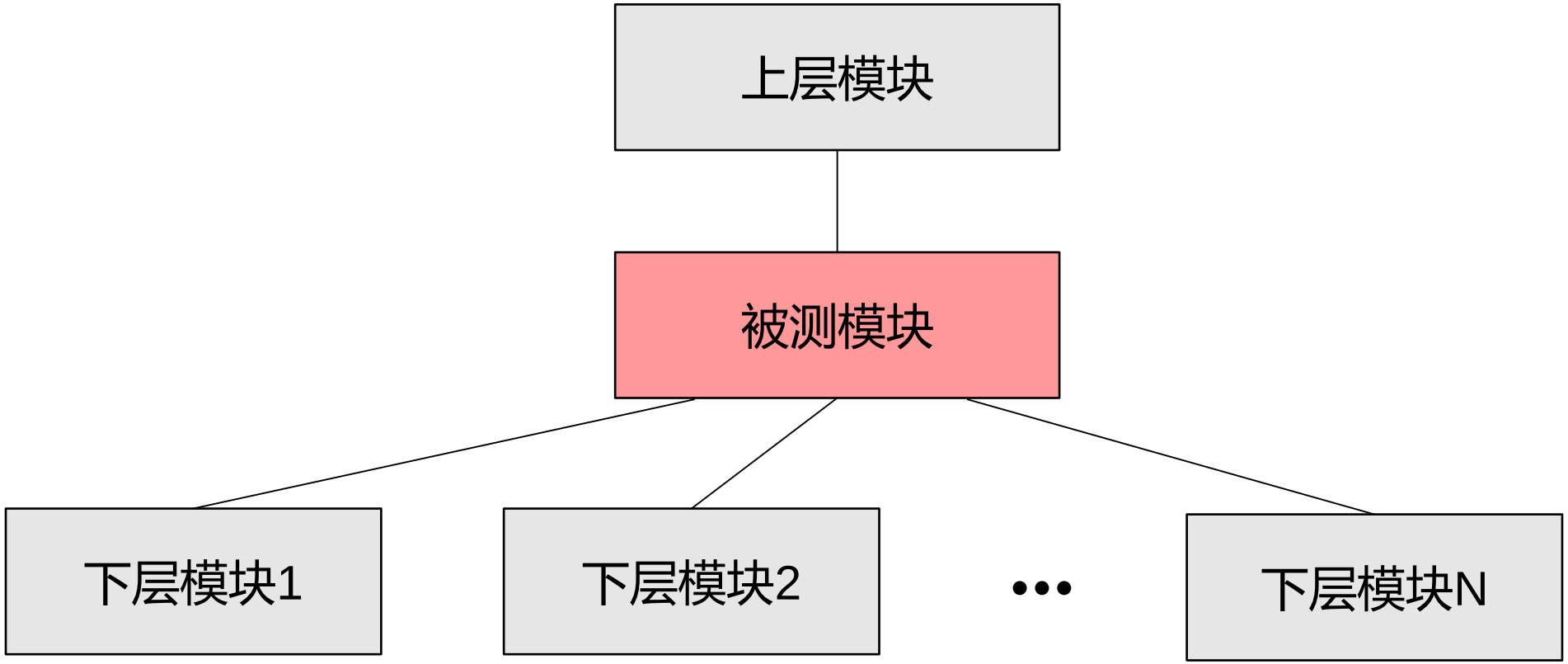


单元测试方法

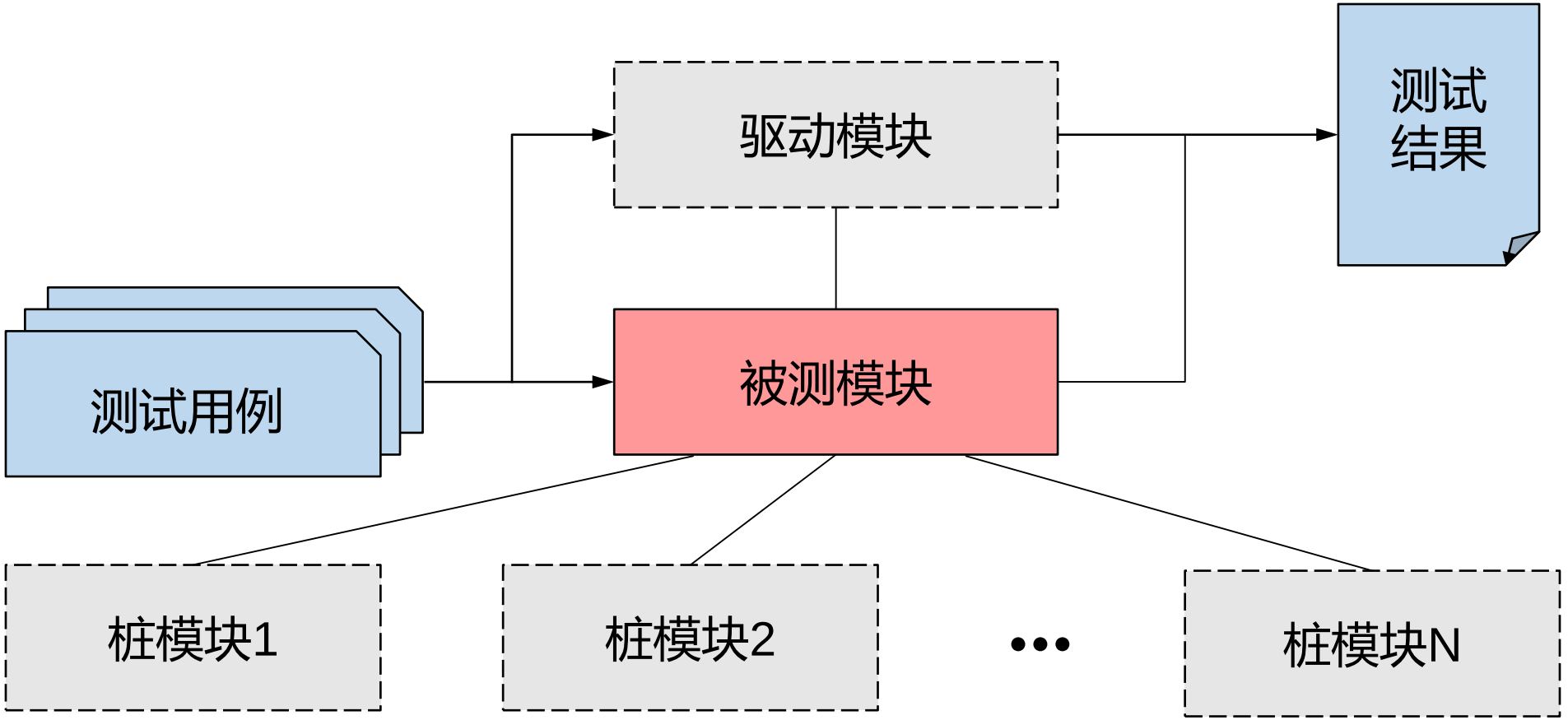
白盒测试（White Box Testing）：又称结构测试，它把测试对象看做一个透明的盒子，允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。



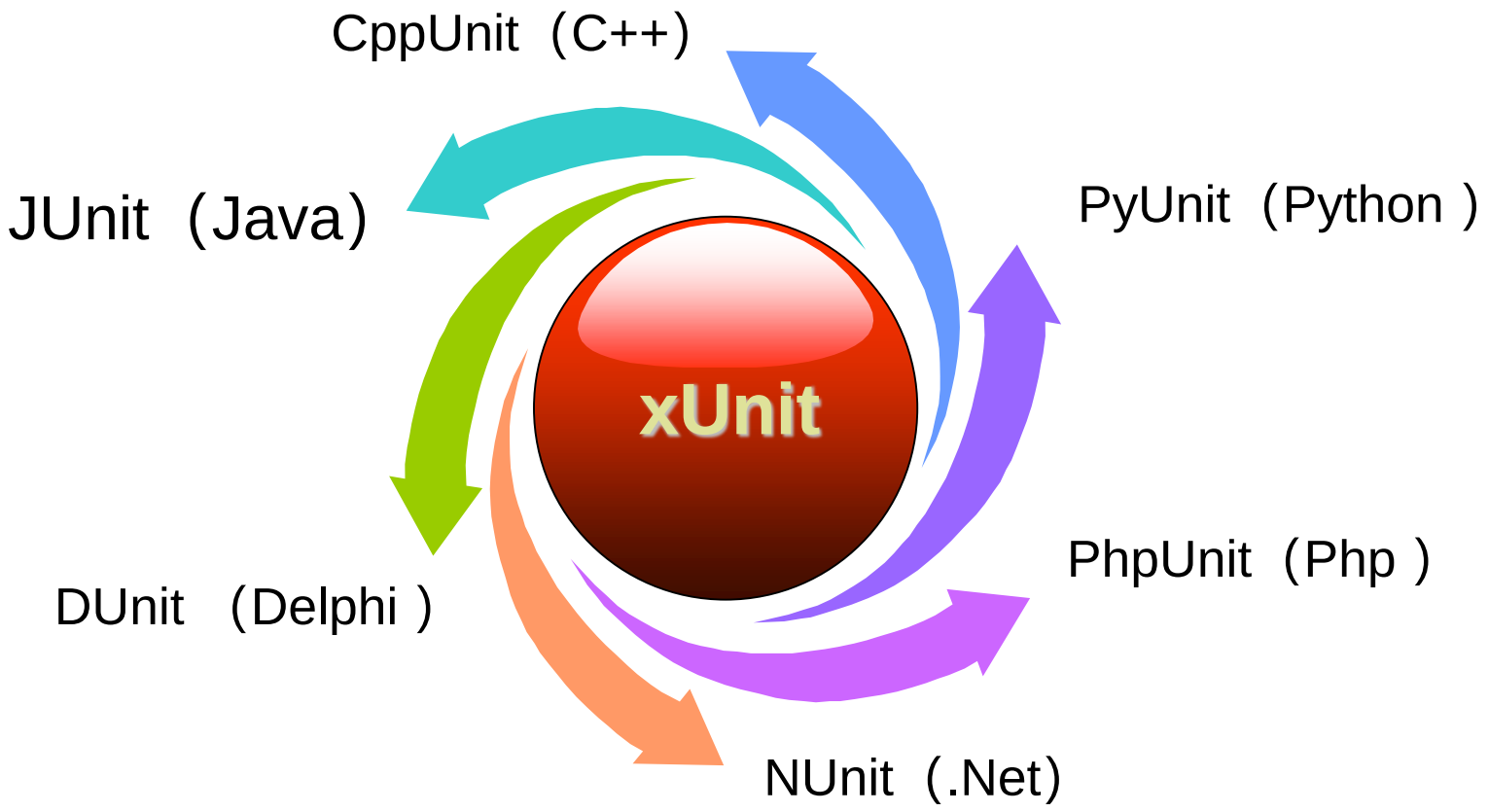
单元测试方法



单元测试方法



单元测试之xUnit


CppUnit（C++）
JUnit（Java）
PyUnit（Python ）
xUnit
PhpUnit（Php ）
DUnit （Delphi ）
NUnit（.Net）

单元测试之xUnit

xUnit 通常适用于以下场景的测试

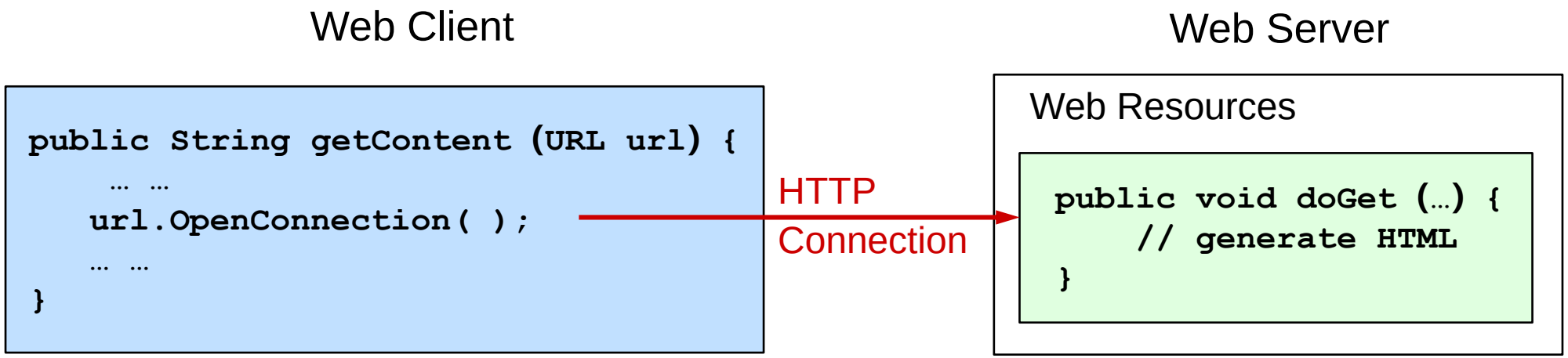
- 单个函数、一个类或者几个功能相关类的测试
- 尤其适用于纯函数测试或者接口级别的测试

xUnit 无法适用于复杂场景的测试

- 被测对象依赖关系复杂，甚至无法简单创建出这个对象
- 对于一些失败场景的测试
- 被测对象中涉及多线程合作
- 被测对象通过消息与外界交互的场景

思考与讨论

举例：WebClient 类通过调用自身的 String getContent（URL url）方法来获得远程Web服务器上对应的URL内容。



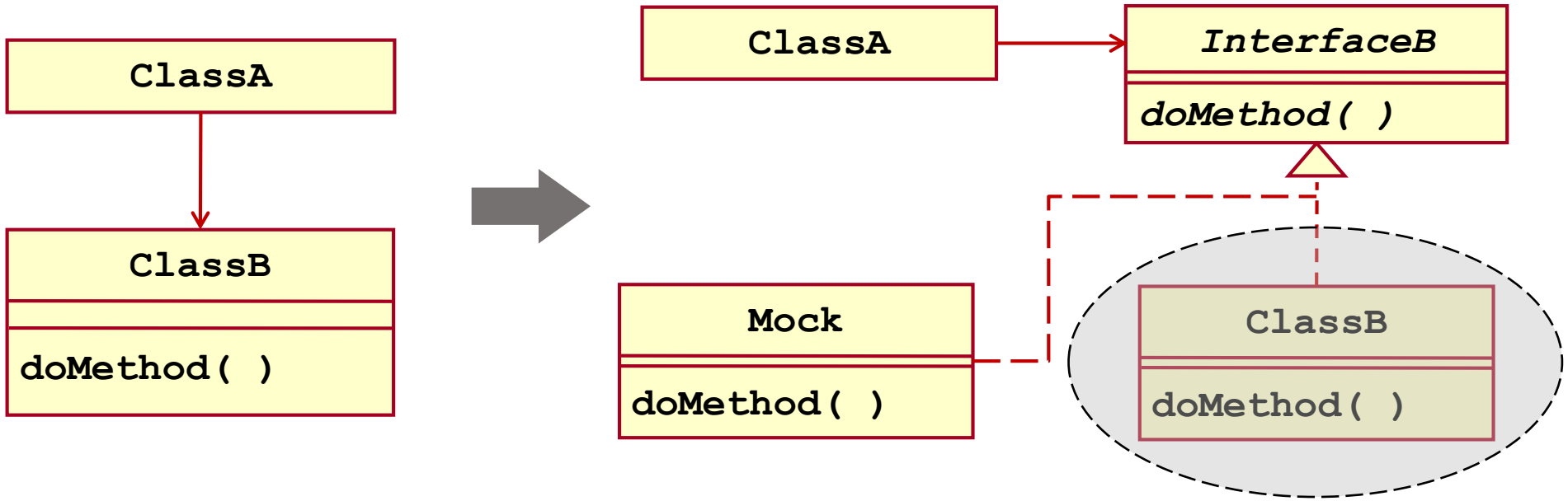
单元测试之Mock

Mock测试是在测试过程中对于某些不容易构造或者不容易获取的对象，用一个虚拟的对象（即Mock对象）来创建以便测试的方法。

- 真实对象具有不可确定的行为（产生不可预测的结果）
- 真实对象很难被创建（如具体的Web容器）
- 真实对象的某些行为很难触发（如网络错误）
- 真实情况令程序的运行速度很慢
- 真实对象有用户界面
- 测试需要询问真实对象它是如何被调用的
- 真实对象实际上并不存在

单元测试之Mock

关键：需要应用针对接口的编程技术，即被测试的代码通过接口来引用对象，再使用Mock对象模拟所引用的对象及其行为，因此被测试模块并不知道它所引用的究竟是真实对象还是Mock对象。



单元测试之Mock

思考：在微信抢票应用中，程序通过直接调用系统函数来获得当前的系统时间。如果使用Mock方法进行测试，应该如何重构程序?

教学提纲

1. 软件测试概述
2. 黑盒测试方法
3. 白盒测试方法
4. 代码性能分析

测试用例的重要性

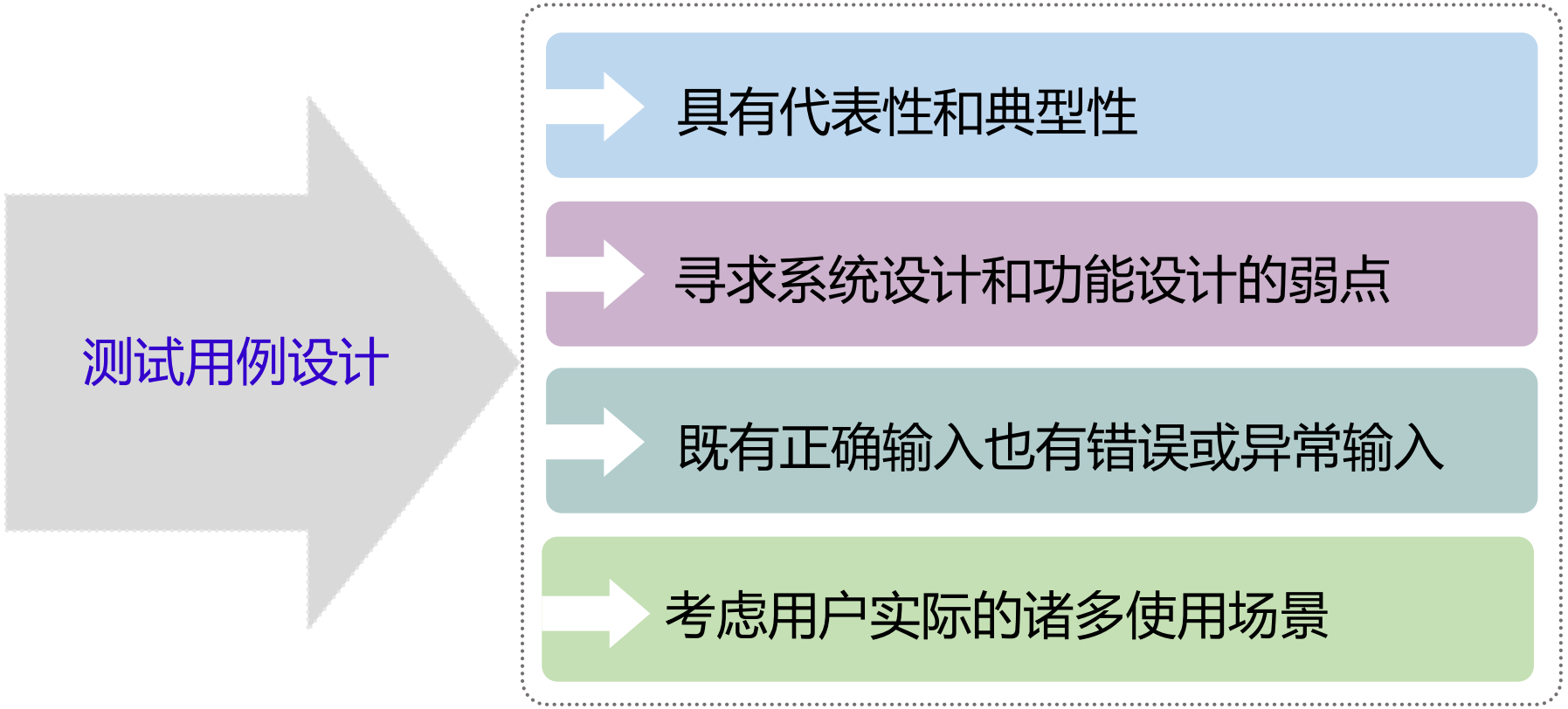
测试用例的概念

测试集
测试用例值的集合

- 测试用例值：完成被测软件的某个执行所需的输入值
- 期望结果：当且仅当程序满足其期望行为，执行测试时产生的结果
- 前缀值：将软件置于合适的状态来接受测试用例值的任何必要的输入
- 后缀值：测试用例值被发送以后，需要被发送到软件的任何输入

验证值：查看测试用例值结果所要用到的值
结束命令：终止程序或返回到稳定状态所要用到的值

测试用例设计的要求

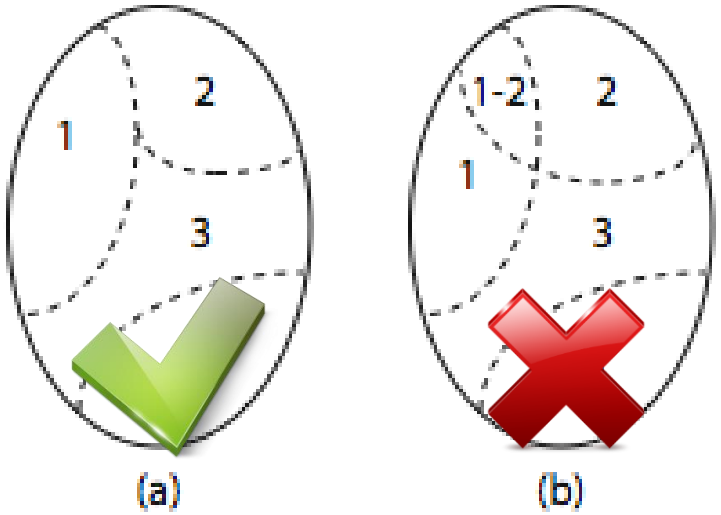

测试用例设计
具有代表性和典型性
寻求系统设计和功能设计的弱点
既有正确输入也有错误或异常输入
考虑用户实际的诸多使用场景

黑盒测试技术

黑盒测试是将测试对象看做一个黑盒子，完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。

等价类划分

等价类划分是将输入域划分成尽可能少的若干子域，在划分中要求每个子域两两互不相交，每个子域称为一个等价类。

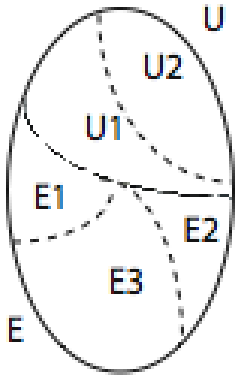


- 同一输入域的等价类划分可能不唯一
- 只需从每一个等价类中选取一个输入作为测试用例
- 对于相同的等价类划分，不同测试人员选取的测试用例集可能是不同的

等价类类型

有效等价类是对规格说明有意义、合理的输入数据构成的集合，能够检验程序是否实现了规格说明中预先规定的功能和性能。

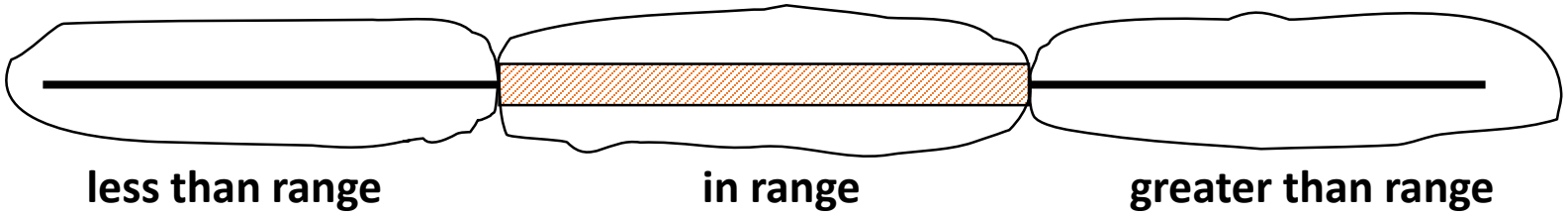
无效等价类是对规格说明无意义、不合理的输入数据构成的集合，以检查程序是否具有一定的容错性。



- E 表示所有正常和合法的输入
- U 表示所有异常和非法的输入

变量的等价类

取值范围：在输入条件规定了取值范围的情况下，可以确定一个有效等价类和两个无效等价类。



举例：程序的输入参数 x 是小于100大于10的整数。

1个有效等价类：$10 < x < 100$

2个无效等价类：$x \leq 10$ 和 $x \geq 100$

变量的等价类

字符串：在规定了输入数据必须遵守的规则情况下，可确定一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。

举例：姓名是长度不超过20的非空字符串，且只由字母组成，数字和其他字符都是非法的。

1个有效等价类：满足了上述所有条件的字符串

3个无效等价类：

- 空字符串
- 长度超过20的字符串
- 包含了数字或其它字符的字符串

变量的等价类

枚举：若规定输入数据是一组值（假定N个），并且程序要对每一个输入值分别处理，可确定N个有效等价类和一个无效等价类。

举例：某程序根据不同的学历分别计算岗位工资，其中学历可以是专科、本科、硕士、博士等四种类型。

4个有效等价类：专科、本科、硕士、博士

1个无效等价类：其他学历

如果将专科、本科、硕士、博士按一种方式计算岗位工资，这时应如何划分等价类？

变量的等价类

数组：数组是一组具有相同类型的元素的集合，数组长度及其类型都可以作为等价类划分的依据。

举例：假设某程序的输入是一个整数数组 `int oper[3]`

1个有效等价类：所有合法值的数组，如 `{-10,20}`

2个无效等价类：

- 空数组
- 所有大于期望长度的数组，如 `{-9,0,12,15}`

如果对数组元素有其他附加约束，例如数组`oper`元素的取值范围是`[-3,3]`，则需要增加相应的等价类。

变量的等价类

复合数据类型：复合数据类型是包含两个或两个以上相互独立的属性的输入数据，在进行等价类划分时需要考虑输入数据的每个属性的合法和非法取值。

举例：
```
struct student {
        string name;
        string course[100];
        int grade[100];
    }
```

对复合数据类型中的每个元素进行等价类划分，再将这些等价类进行组合，最终形成对软件整个输入域的划分。

等价类组合

测试用例生成：测试对象通常有多个输入参数，如何对这些参数等价类进行组合测试，来保证等价类的覆盖率，是测试用例设计首先需要考虑的问题。

所有有效等价类的代表值都集成到测试用例中，即覆盖有效等价类的所有组合。任何一个组合都将设计成一个有效的测试用例，也称正面测试用例。

无效等价类的代表值只能和其他有效等价类的代表值（随意）进行组合。因此，每个无效等价类将产生一个额外的无效测试用例，也称负面测试用例。

举例：判断三角形类型

输入三个整数a、b、c，分别作为三角形的三条边，现通过一个程序判断这三条边构成的三角形类型，包括等边三角形、等腰三角形、一般三角形（特殊的还包括直角三角形）以及构不成三角形。

现在要求输入的三个整数a、b、c必须满足以下条件：

条件1：1≤a≤100

条件2：1≤b≤100

条件3：1≤c≤100

条件4：a<b+c

条件5：b<a+c

条件6：c<a+b

请使用等价类划分方法，设计该程序的测试用例。

举例：判断三角形类型

等价类划分：

① 按输入取值划分

{0，>0，<0} 或 {0，1，>1，<0}

② 按输出的几何特性划分

{等腰且非等边三角形，等边三角形，一般三角形，非三角形}

你会选择哪一种方法划分等价类

举例：判断三角形类型

标准等价类测试用例

序号	测试用例描述	输入参数			期望输出
		a	b	c	
1	a>0，b>0，c>0 a+b>c，b+c>a，a+c>b a=b=c	10	10	10	等边三角形
2	a>0，b>0，c>0 a+b>c，b+c>a，a+c>b a=b≠c 或 b=c≠a 或 a=c≠b	10	10	5	等腰三角形
3	a>0，b>0，c>0 a+b>c，b+c>a，a+c>b a≠b≠c	3	4	5	一般三角形
4	a>0，b>0，c>0 a+b≤c 或 b+c≤a 或 a+c≤b	4	1	2	非三角形

举例：判断三角形类型

健壮等价类测试用例

序号	测试用例描述	输入参数			期望输出
		a	b	c	
1-4	有效等价类同前面（略）				
5	a<0，b>0，c>0	-1	5	5	a 值越界
6	a>0，b<0，c>0	5	-1	5	b 值越界
7	a>0，b>0，c<0	5	5	-1	c 值越界
8	a>100，b>0，c>0	101	5	5	a 值越界
9	a>0，b>100，c>0	5	101	5	b 值越界
10	a>0，b>0，c>100	5	5	101	c 值越界

边界值分析

边界值分析是对输入或输出的边界值进行测试的一种方法，它通常作为等价类划分法的补充，这种情况下的测试用例来自等价类的边界。

- 先确定边界：通常输入或输出等价类的边界就是应该着重测试的边界情况。
- 选取正好等于、刚刚大于或刚刚小于边界的值作为测试数据，而不是选取等价类中的典型值或任意值。

实践表明：大多数故障往往发生在输入定义域或输出值域的边界上，而不是内部。因此，针对各种边界情况设计测试用例，通常会取得很好的测试效果。

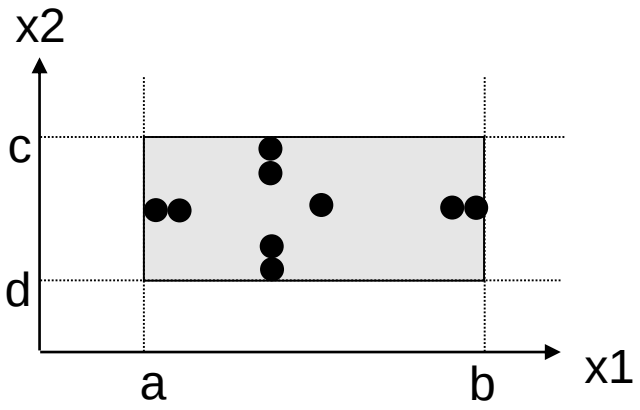
边界值分析

输入项	边界值	测试用例的设计思路
字符	起始 - 1个字符 结束 + 1个字符	假设一个文本输入区域允许输入1到255个字符，输入1个和255个字符作为有效等价类； 输入0个和256个字符作为无效等价类，这几个数值都属于边界条件值。
数值	最小值 - 1 最大值 + 1	假设某软件要求输入5位十进制整数值，可以使用10000作为最小值、99999作为最大值；然后使用刚好小于5位和大于5位的数值作为边界条件。
空间	小于空余空间一点 大于满空间一点	例如在用U盘存储数据时，使用比剩余磁盘空间大一点（几 KB）的文件作为边界条件。

边界值分析

基本思想：故障往往出现在程序输入变量的边界值附近

- 边界值分析法是基于可靠性理论中称为“单故障”的假设，即有两个或两个以上故障同时出现而导致失效的情况很少。
- 对程序中的每个变量重复：每次保留一个变量，让其余的变量取正常值，被保留的变量依次取 min、min+、nom、max- 和 max。



$<x1_{nom}, x2_{min}>$ $<x1_{nom}, x2_{min+}>$ $<x1_{nom}, x2_{nom}>$

$<x1_{nom}, x2_{max}>$ $<x1_{nom}, x2_{max-}>$ $<x1_{min}, x2_{nom}>$

$<x1_{min+}, x2_{nom}>$ $<x1_{max}, x2_{nom}>$ $<x1_{max-}, x2_{nom}>$

举例：判断三角形类型

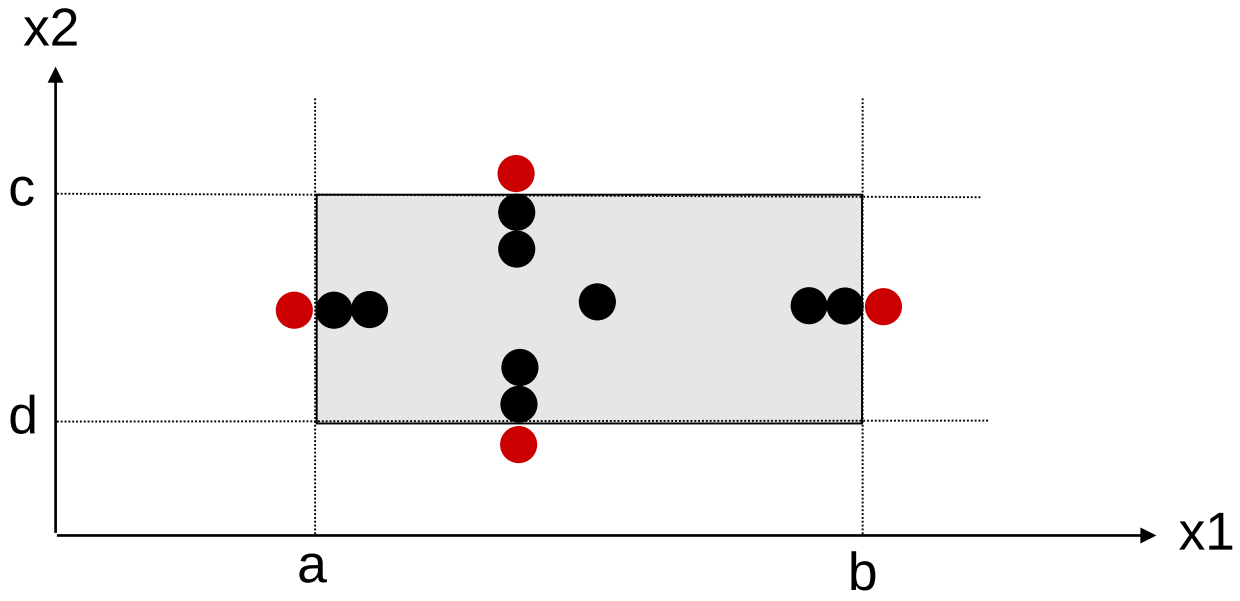
序号	测试用例描述	输入参数			期望输出
		a	b	c	
1	a>0，b>0，c>0 a+b>c，b+c>a，a+c>b a=b=c	60	60	60	等边三角形
2	a>0，b>0，c>0 a+b>c，b+c>a，a+c>b a=b≠c 或 b=c≠a 或 a=c≠b	60	60	1	等腰三角形
3		60	60	2	
4		50	50	99	
5		60	1	60	
6		60	2	60	

举例：判断三角形类型

序号	测试用例描述	输入参数			期望输出
		a	b	c	
7	a>0，b>0，c>0 a+b>c，b+c>a，a+c>b a=b≠c 或 b=c≠a 或 a=c≠b	50	99	50	等腰三角形
8		1	60	60	
9		2	60	60	
10		99	50	50	
11	a>0，b>0，c>0 a+b≤c 或 b+c≤a 或 a+c≤b	50	50	100	非三角形
12		50	100	50	
13		100	50	50	

健壮性测试

健壮性测试是作为边界值分析的一个简单的扩充，它除了对变量的5个边界值分析取值外，还要增加一个略大于最大值（max+）以及略小于最小值（min-）的取值，检查超过极限值时系统的情况。



错误推测法

错误推测法是人们根据经验或直觉推测程序中可能存在的各种错误，从而有针对性地编写检查这些错误的测试用例的方法。

- 软件缺陷具有空间聚集性，80%的缺陷常常存在于20%的代码中。因此，应当记住常常光临代码的高危多发“地段”，这样发现缺陷的可能性会大得多。
- 列举程序中所有可能的错误和容易发生错误的特殊情况，根据可能出现的错误情况选择测试用例。

教学提纲

1. 软件测试概述
2. 黑盒测试方法
3. 白盒测试方法
4. 代码性能分析

白盒测试技术

白盒测试是将测试对象看做一个透明的盒子，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。

黑盒测试与白盒测试

假设有一台的面包机，从上面倒入面粉与水，开动机器后下面出来的就是烤好的面包。如果这个面包机多年未用，我们用什么办法检查机器的情况？

黑盒测试与白盒测试

黑盒测试：

- 直接把水和面粉放进去看它是否可以工作

白盒测试：

- 把机器拆开，检查和清洗内部的部件，然后再组装起来，之后把水和面粉加进去试一试。

问题：

- 是否可以单纯用黑盒测试而不做白盒测试？

测试覆盖标准

- 测试需求：测试需求是软件制品的一个特定元素，测试用例必须满足或覆盖这个特定元素。
- 覆盖标准：一个覆盖标准是一条规则，或者是将测试需求施加在一个测试集上的一组规则。
- 测试覆盖：给定一个覆盖标准 *C* 和相关的测试需求集合 *TR*，欲使一个测试集合 *T* 满足*C*，当且仅当对于测试需求集合 *TR* 中的每一条测试需求*tr*，在 *T* 中至少存在一个测试 *t* 可以满足 *tr*。
- 覆盖程度：给定一个测试需求集合 *TR* 和一个测试集合 *T*，覆盖程度就是 *T* 满足的测试需求数占 *TR* 总数的比例。

测试覆盖标准

软心糖豆：6 种口味和 4 种颜色

柠檬味（黄色）、开心果味（绿色）、梨子味（白色）
哈密瓜味（橙色）、橘子味（橙色）、杏味（黄色）

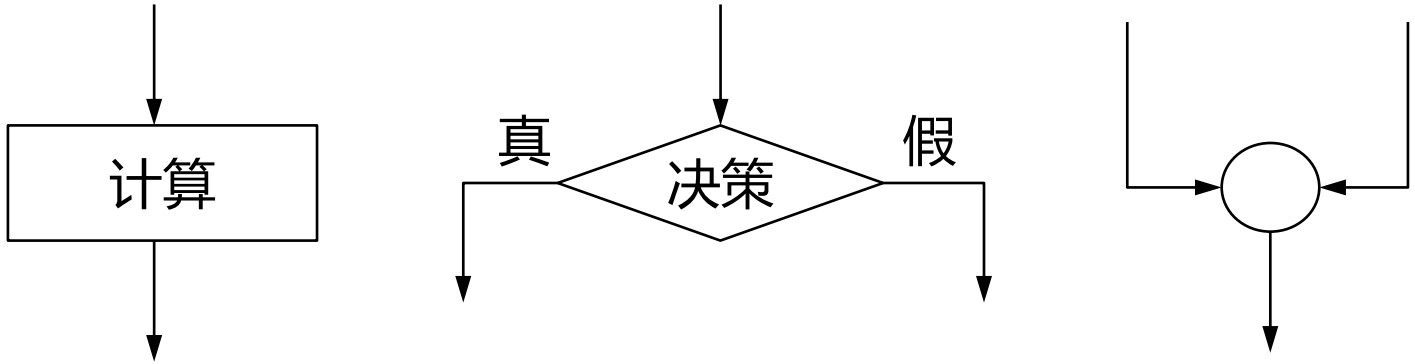
- 可以用什么覆盖标准来选择糖豆进行测试?
- 哪一种覆盖标准更好？为什么?
- 应该考虑哪些因素来选择覆盖标准?

控制流图

控制流图（CFG， Control Flow Graph）是一个过程或程序的抽象表示。

控制流图的基本符号：

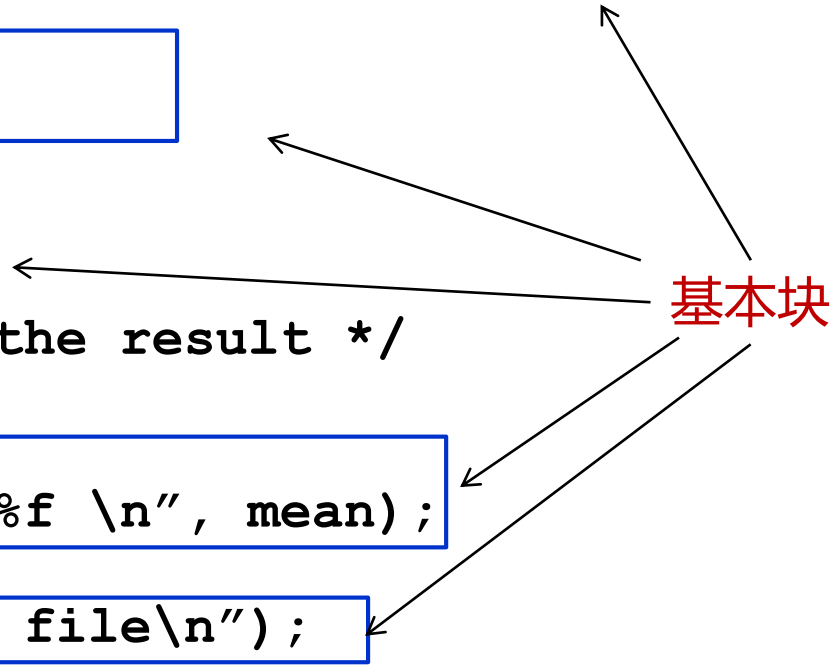
- 矩形代表了连续的顺序计算，也称基本块
- 节点是语句或语句的一部分，边表示语句的控制流

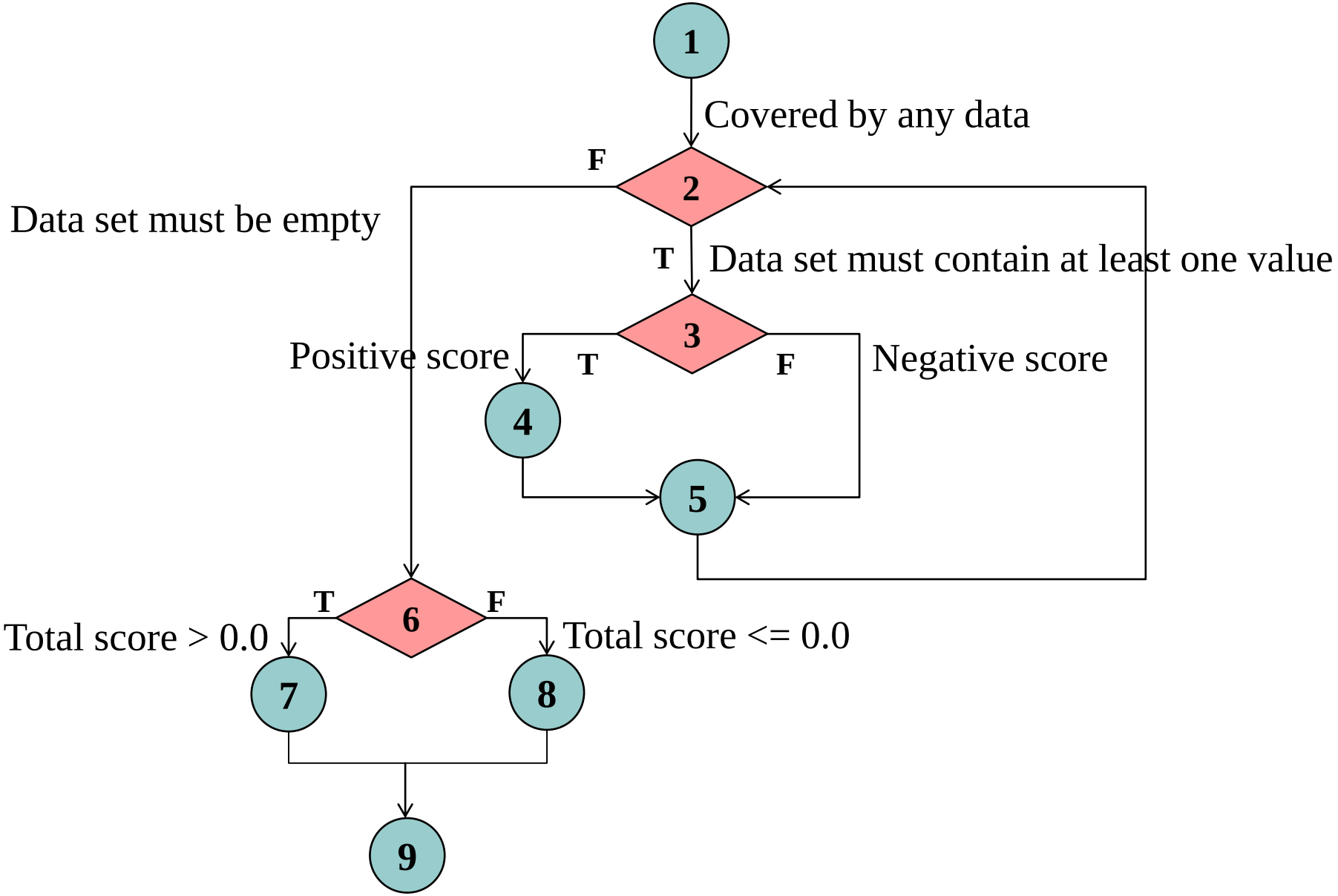


顺序计算　　判断节点　　合并节点

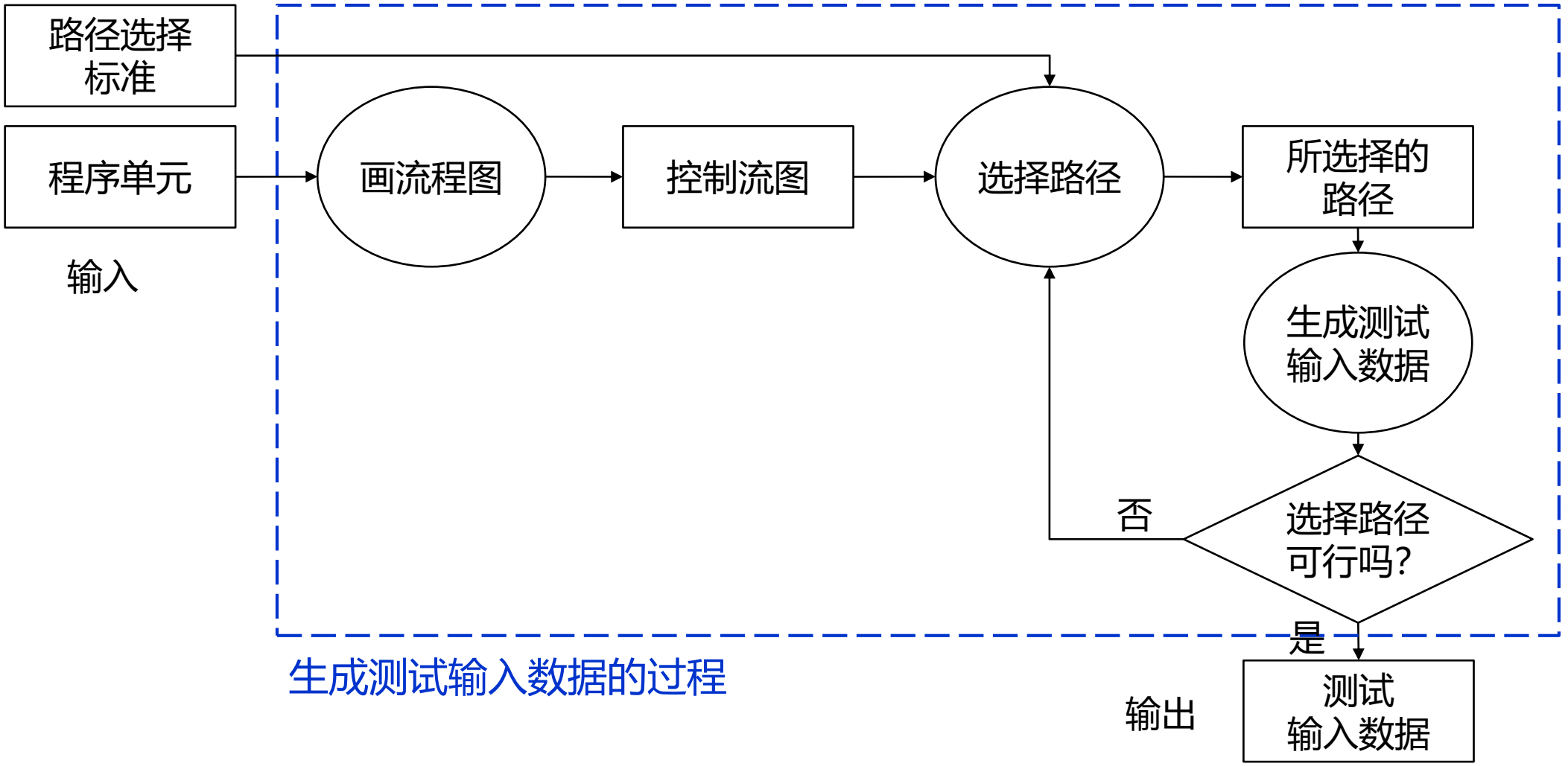
```
FindMean(float *mean, FILE *fp)
{
    float sum = 0.0, score = 0.0;
    int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
    while (!EOF(fp)) {
        if (score > 0.0) {
            sum += score;
            num++;
        }
        fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
    if (num > 0) {
        *mean = sum/num;
        printf("The mean score is %f \n", mean);
    } else
        printf("No scores found in file\n");
}
```

```
FindMean(float *mean, FILE *fp)
{
①   float sum = 0.0, score = 0.0;
    int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
②   while (!EOF(fp)) {
③     if (score > 0.0) {
④       sum += score;
        num++;
      }
⑤     fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
⑥   if (num > 0) {
⑦     *mean = sum/num;
      printf("The mean score is %f \n", mean);
    } else
⑧     printf("No scores found in file\n");
}
```




1
Covered by any data
F
2
Data set must be empty
T
Data set must contain at least one value
3
Positive score
T
F
Negative score
4
5
T
6
F
Total score > 0.0
Total score <= 0.0
7
8
9

基于控制流的测试

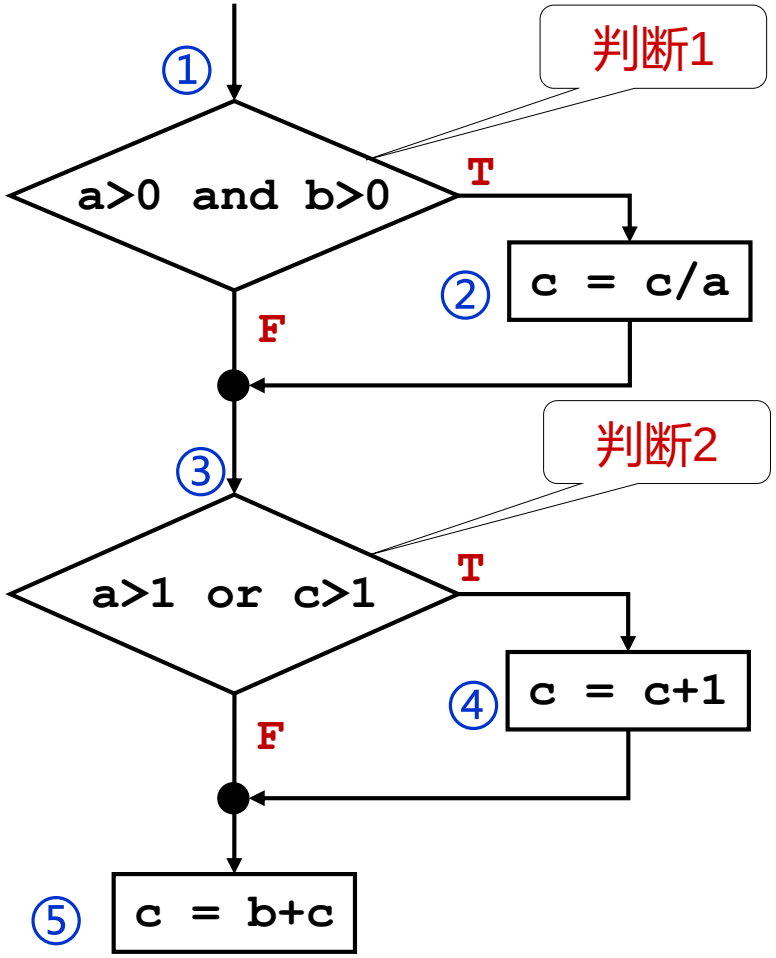

路径选择
标准
程序单元
输入
画流程图
控制流图
选择路径
所选择的
路径
生成测试
输入数据
选择路径
可行吗?
否
是
测试
输入数据
输出
生成测试输入数据的过程

代码覆盖标准

代码覆盖率描述的是代码被测试的比例和程度，通过代码覆盖率可以得知哪些代码没有被覆盖，从而进一步补足测试用例。

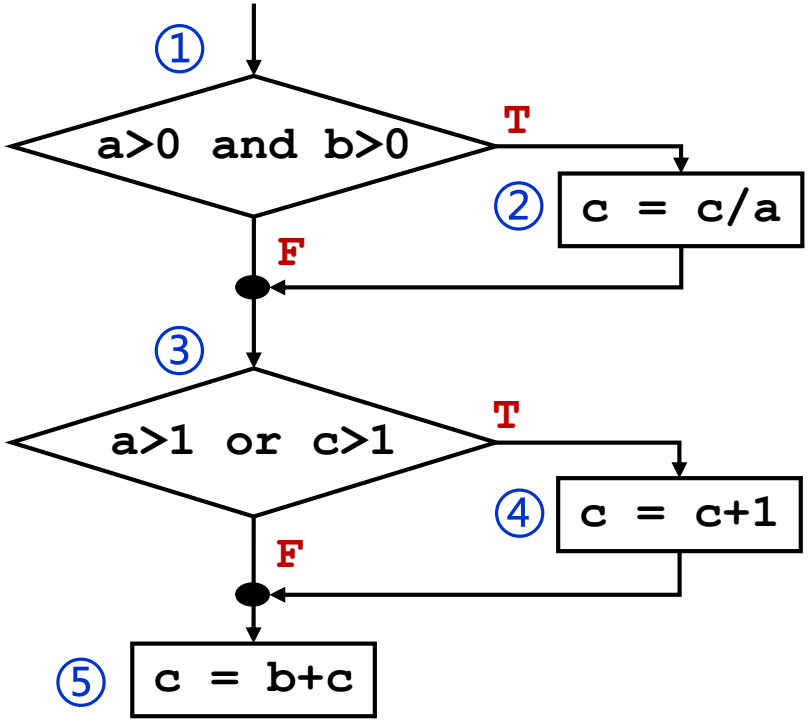
示例描述

```
double func1(int a, int b, double c)
{
    if (a>0 && b>0) {
        c = c/a;
    }

    if (a>1 || c>1) {
        c = c+1;
    }

    c = b+c;

    return c;
}
```



语句覆盖

程序中的每个可执行语句至少被执行一次。



输入：a=2, b=1, c=6

语句覆盖

程序中的每个可执行语句至少被执行一次。

测试用例：

输入：`a=2, b=1, c=6`

程序中三个可执行语句均被执行一次，满足语句覆盖标准。

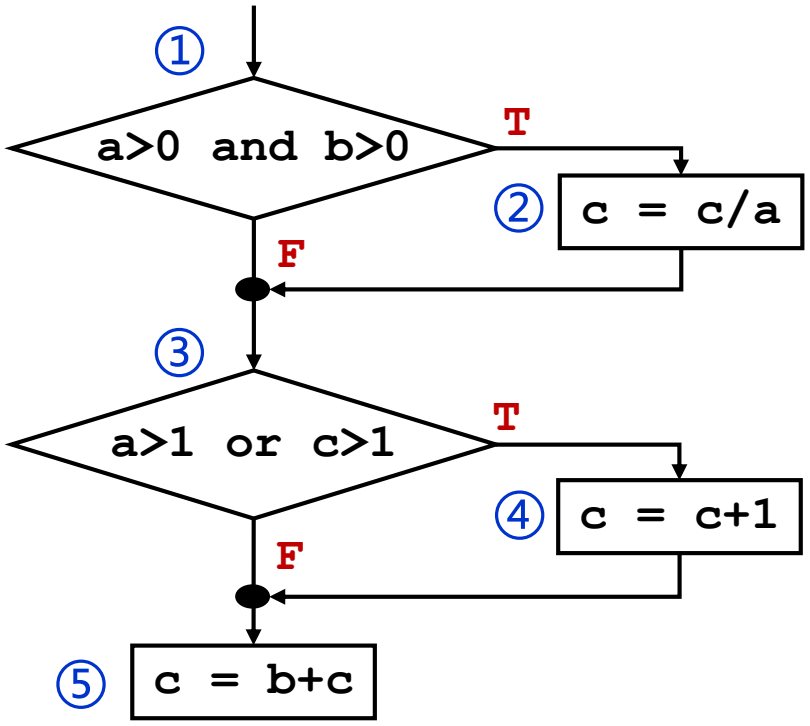
问题分析：

测试用例虽然覆盖可执行语句，但无法检查判断逻辑是否存在问题，例如第一个条件判断中“`&&`”被错误地写成“`||`”。

语句覆盖是最弱的逻辑覆盖准则。

判定覆盖（分支覆盖）

程序中每个判断的取真和取假分支至少经历一次，即判断真假值均被满足。



输入：a=2,b=1,c=6

输入：a=-2,b=-1,c=-3

判定覆盖（分支覆盖）

程序中每个判断的取真和取假分支至少经历一次，即判断真假值均被满足。

测试用例：

输入：`a=2,b=1,c=6`，覆盖判断1的 T分支和判断2的 T分支

输入：`a=-2,b=-1,c=-3`，覆盖判断1的 F分支和判断2的 F分支

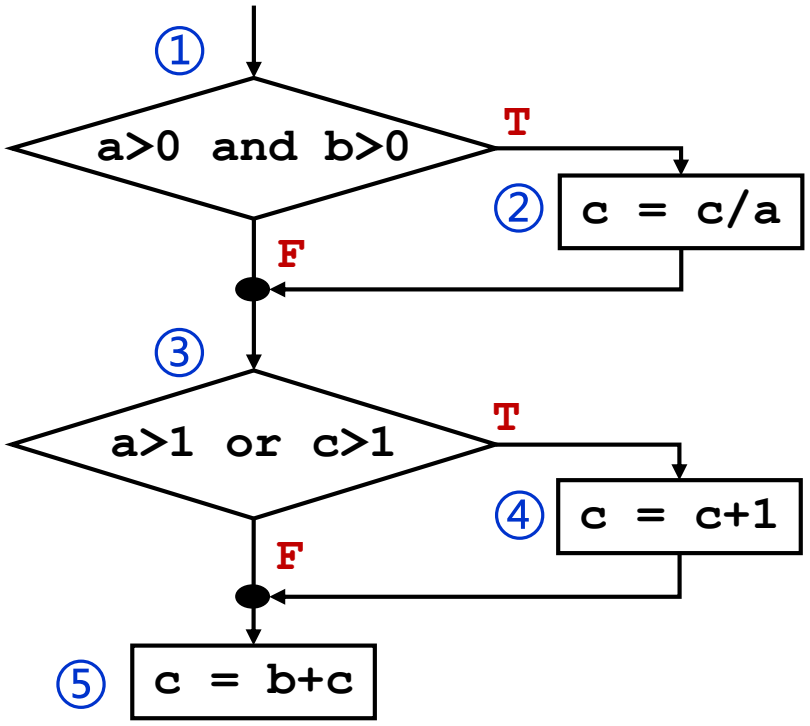
问题分析：

由于大部分判定语句是由多个逻辑条件组合而成，若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。

判定覆盖具有比语句覆盖更强的测试能力，但仍是弱的逻辑覆盖。

条件覆盖

每个判断中每个条件的可能取值至少满足一次。



输入： a=2,b=-1,c=-2

覆盖 a>0,b<=0,a>1,c<=1 条件

输入：a=-1,b=2,c=3

覆盖 a<=0,b>0,a<=1,c>1 条件

条件覆盖

每个判断中每个条件的可能取值至少满足一次。

测试用例：

输入： a=2,b=-1,c=-2，覆盖 a>0,b<=0,a>1,c<=1 条件

输入：a=-1,b=2,c=3，覆盖 a<=0,b>0,a<=1,c>1 条件

问题分析：

条件覆盖不一定包含判定覆盖，例如上面测试用例就没有覆盖判断1的T分支和判断2的 F分支。

条件覆盖只能保证每个条件至少有一次为真，而没有考虑整个判定结果。

判定条件覆盖

判断中所有条件的可能取值至少执行一次，且所有判断的可能结果至少执行一次。

测试用例：

输入：`a=2,b=1,c=6`，覆盖 `a>0,b>0,a>1,c>1` 且判断均为T

输入：`a=-1,b=-2,c=-3`，覆盖 `a<=0,b<=0,a<=1,c<=1`且判断均为F

问题分析：

判定条件覆盖能够同时满足判定、条件两种覆盖标准。

没有考虑条件的各种组合情况。

条件组合覆盖

判断中每个条件的所有可能取值组合至少执行一次，并且每个判断本身的结果也至少执行一次。

组合编号	覆盖条件取值	判定条件取值	判定-条件组合
1	a>0，b>0	判断1取T	a>0，b>0，判断1取T
2	a>0，b<=0	判断1取F	a>0，b<=0，判断1取F
3	a<=0，b>0	判断1取F	a<=0，b>0，判断1取F
4	a<=0，b<=0	判断1取F	a<=0，b<=0，判断1取F
5	a>1，c>1	判断2取T	a>1，c>1，判断2取T
6	a>1，c<=1	判断2取T	a>1，c<=1，判断2取T
7	a<=1，c>1	判断2取T	a<=1，c>1，判断2取T
8	a<=1，c<=1	判断2取F	a<=1，c<=1，判断2取F

条件组合覆盖

测试用例	覆盖条件	覆盖路径	覆盖组合
输入：a=2, b=1, c=6	a>0,b>0 a>1,c>1	1-2-4	1, 5
输入：a=2, b=-1, c=-2	a>0,b<=0 a>1,c<=1	1-3-4	2, 6
输入：a=-1, b=2, c=3	a<=0,b>0 a<=1,c>1	1-3-4	3, 7
输入：a=-1, b=-2, c=-3	a<=0,b<=0 a<=1,c<=1	1-3-5	4, 8

问题分析：

条件组合覆盖准则满足判定覆盖、条件覆盖和判定条件覆盖准则。

覆盖了所有组合，但覆盖路径有限，上面示例中1-2-5没覆盖。

路径覆盖

覆盖程序中的所有可能的执行路径。

测试用例	覆盖条件	覆盖路径	覆盖组合
输入：a=2, b=1, c=6	a>0,b>0 a>1,c>1	1-2-4	1, 5
输入：a=1, b=1, c=-3	a>0,b>0 a<=1,c<=1	1-2-5	1, 8
输入：a=-1, b=2, c=3	a<=0,b>0 a<=1,c>1	1-3-4	3, 7
输入：a=-1, b=-2, c=-3	a<=0,b<=0 a<=1,c<=1	1-3-5	4, 8

路径覆盖

问题分析：前面的测试用例完全覆盖所有路径，但没有覆盖所有条件组合。

下面结合条件组合和路径覆盖两种方法重新设计测试用例：

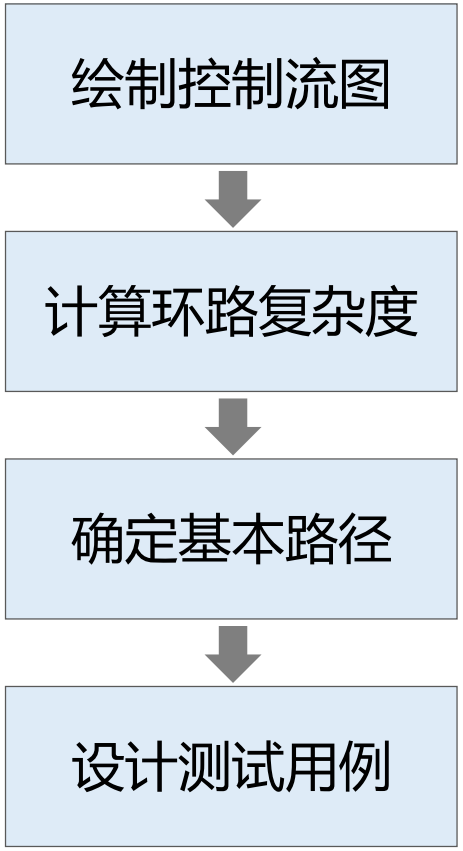
测试用例	覆盖条件	覆盖路径	覆盖组合
输入：a=2, b=1, c=6	a>0,b>0 a>1,c>1	1-2-4	1, 5
输入：a=1, b=1, c=-3	a>0,b>0 a<=1,c<=1	1-2-5	1, 8
输入：a=2, b=-1, c=-2	a>0,b<=0 a>1,c<=1	1-3-4	2, 6
输入：a=-1, b=2, c=3	a<=0,b>0 a<=1,c>1	1-3-4	3, 7
输入：a=-1, b=-2, c=-3	a<=0,b<=0 a<=1,c<=1	1-3-5	4, 8

如何看待测试覆盖率

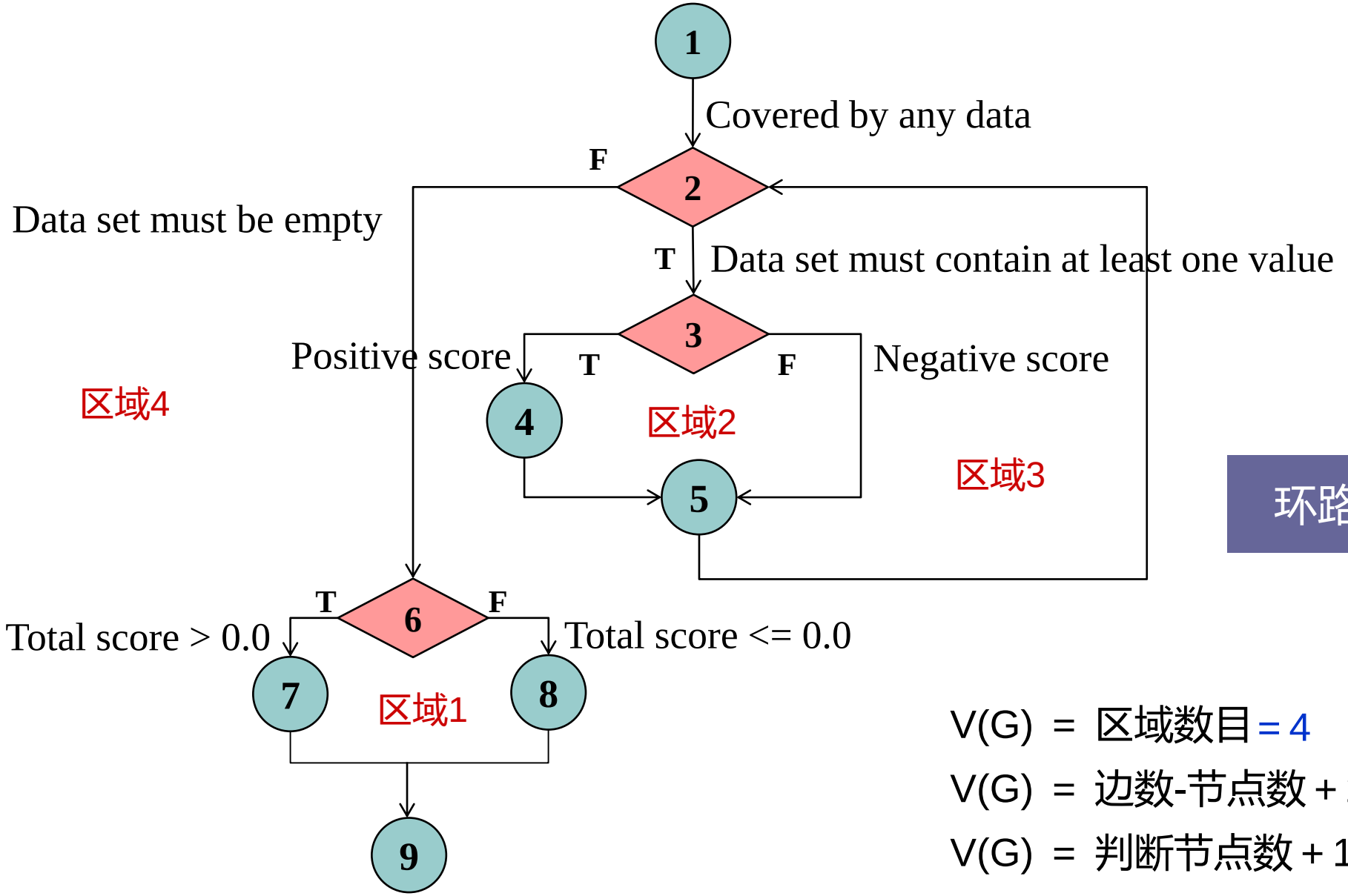
- 覆盖率数据只能代表测试过哪些代码，不能代表是否测试好这些代码。
- 较低的测试覆盖率能说明所做的测试还不够，但反之不成立。
- 路径覆盖 > 判定覆盖 > 语句覆盖
- 测试人员不能盲目追求代码覆盖率，而应该想办法设计更好的测试用例。
- 测试覆盖率应达到多少需要考虑软件整体的覆盖率情况以及测试成本。

基本路径测试

基本路径测试是在程序控制流图基础上，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，从而设计测试用例的方法。



```
FindMean(float *mean, FILE *fp)
{
    float sum = 0.0, score = 0.0;
    int num = 0;

    fscanf(fp, "%f", &score); /* Read and parse into score */
    while (!EOF(fp)) {
        if (score > 0.0) {
            sum += score;
            num++;
        }
        fscanf(fp, "%f", &score);
    }
    /* Compute the mean and print the result */
    if (num > 0) {
        *mean = sum/num;
        printf("The mean score is %f \n", mean);
    } else
        printf("No scores found in file\n");
}
```

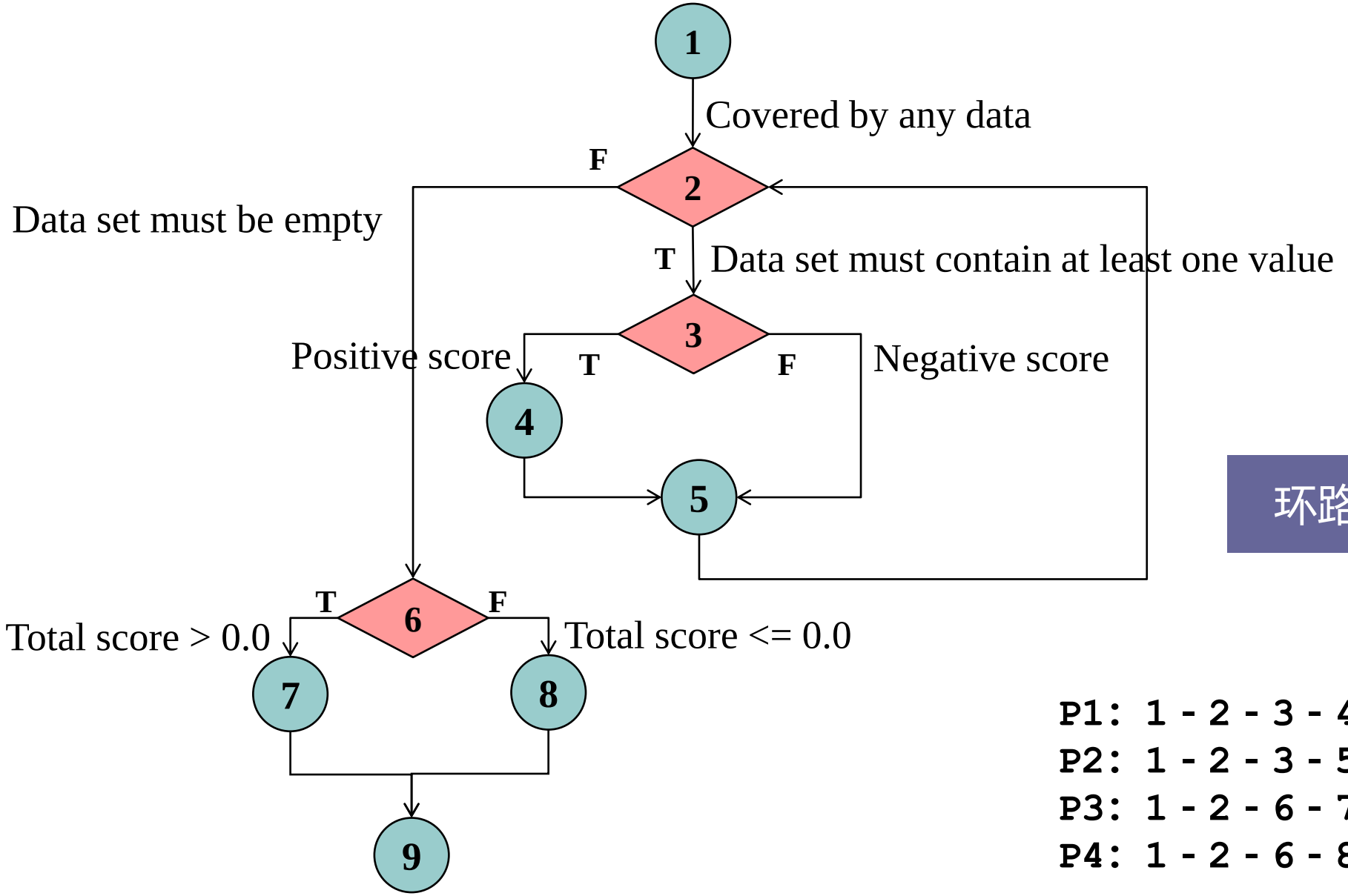


环路复杂性：4

V(G) = 区域数目 = 4

V(G) = 边数-节点数 + 2 = 11-9+2 = 4

V(G) = 判断节点数 + 1 = 3+1 = 4



环路复杂性：4

```
P1: 1 - 2 - 3 - 4 - 5 - 2 - •••
P2: 1 - 2 - 3 - 5 - 2 - •••
P3: 1 - 2 - 6 - 7 - 9
P4: 1 - 2 - 6 - 8 - 9
```

示例：基本路径测试

① 输入：**fp≠NULL**，文件有数据**{60.0,100.0,-1.0,110.0}**，覆盖路径**P1**

输出：打印信息“**The mean score is 90.0**”

② 输入：**fp≠NULL**，文件有数据**{0.0,-1.0,75.0}**，覆盖路径**P2**

输出：打印信息“**The mean score is 75.0**”

③ 输入：**fp≠NULL**，文件无任何数据，覆盖路径**P3**

输出：该路径不可达

④ 输入：**fp≠NULL**，文件无任何数据，覆盖路径**P4**

输出：打印信息“**No scores found in file**”

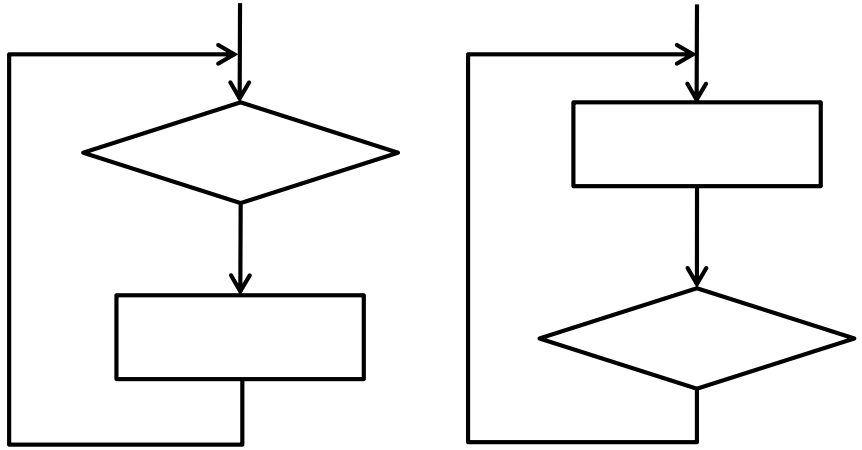
循环测试

循环测试

- 目的：检查循环结构的有效性
- 类型：简单循环、嵌套循环、串接循环和非结构循环

简单循环（次数为 n）

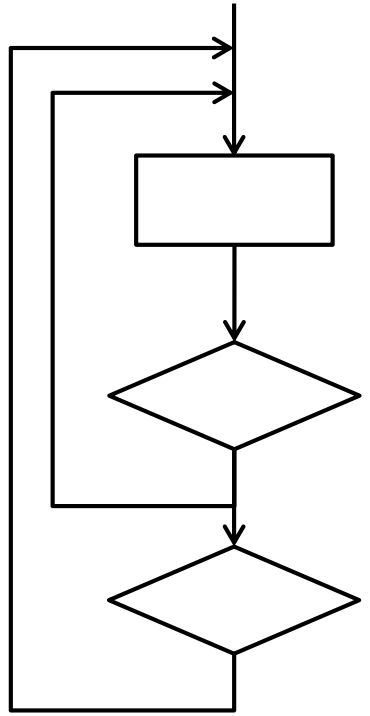
- 完全跳过循环
- 只循环 1 次
- 只循环 2 次
- 循环 m （ m < n ）次
- 分别循环 n-1, n, n+1 次



循环测试

嵌套循环

- 从最内层循环开始，所有外层循环次数设为最小值；
- 对最内层循环按照简单循环方法进行测试；
- 由内向外进行下一个循环的测试，本层循环的所有外层循环仍取最小值，而由本层循环嵌套的循环取某些“典型”值；
- 重复上一步的过程，直到测试完所有循环。



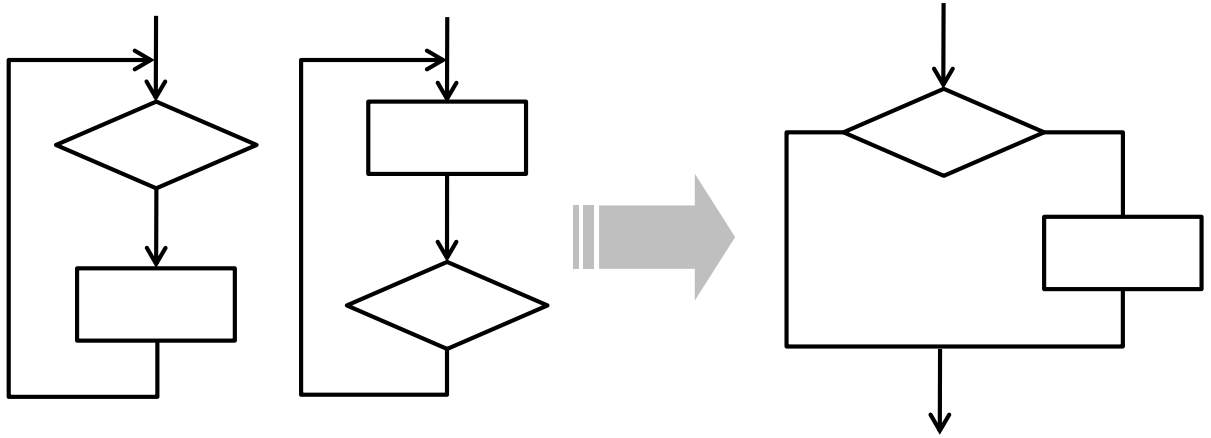
串接循环

- 独立循环：分别采用简单循环的测试方法；
- 依赖性循环：采用嵌套循环的测试方法。

循环测试

Z路径覆盖下的循环测试

- 这是路径覆盖的一种变体，将程序中的循环结构简化为选择结构的一种路径覆盖。
- 循环简化的目的是限制循环的次数，无论循环的形式和循环体实际执行的次数，简化后的循环测试只考虑执行循环体一次和零次（不执行）两种情况。



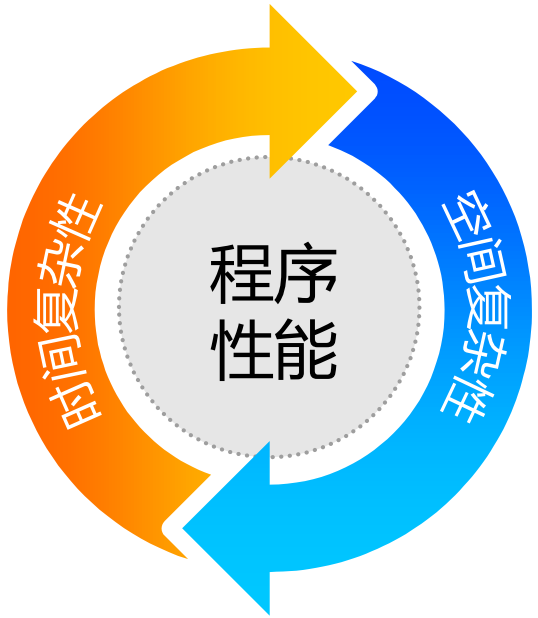
在循环简化的思路下，循环与判定分支的效果是一样的，即循环要么执行、要么跳过。

教学提纲

1. 软件测试概述
2. 黑盒测试方法
3. 白盒测试方法
4. 代码性能分析

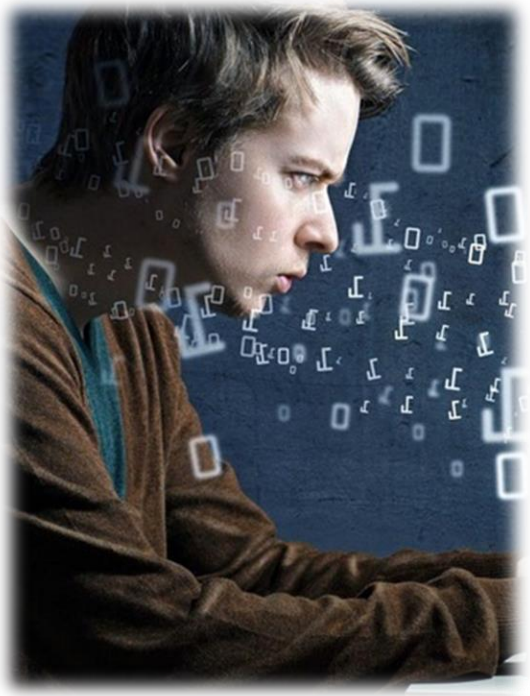
代码性能优化

优化是对代码进行等价变换，使得变换后的代码运行结果与变换前的代码运行结果相同，但执行速度加快或存储开销减少。



- 代码性能优化是一门复杂的学问。
- 根据 80/20 原则，实现程序的重构、优化、扩展以及文档相关的事情通常需要消耗 80% 的工作量。

代码性能优化



- 在满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高程序的效率
- 以提高程序的全局效率为主，提高局部效率为辅
- 在优化程序效率时，应先找出限制效率的“瓶颈”
- 先优化数据结构和算法，再优化执行代码
- 时间效率和空间效率可能是对立的，应当分析哪一个因素更重要，再做出适当的折衷

代码性能优化

- 从一开始就要考虑程序性能，不要期待在开发结束后再做一些快速调整
- 正确的代码要比速度快的代码重要，任何优化都不能破坏代码的正确性

- 认真选择测试数据，使其能够代表实际的使用状况
- 永远不要在没有执行前后性能评估的情况下尝试对代码进行优化

案例分析

编写程序：读入一个文本文件，统计在该文本文件中每个英文单词出现的频率，并输出单词频率最高的100个单词。其中，单词的定义是连续的若干个小写英文字母。

如何有效地提高代码的执行效率？

案例分析

编写程序：读入一个文本文件，统计在该文本文件中每个英文单词出现的频率，并输出单词频率最高的100个单词。其中，单词的定义是连续的若干个小写英文字母。

单词示例

- 1个单词： as
- 2个单词： as,asd
- 4个单词： sa,fdf.fdf fdfdf

```python
# -*- coding: utf-8 -*-
# __author__ = 'Pan Tianxiang'
# __email__ = 'ptx9363@gmail.com'
```

文件头注释，说明了该文件的编码格式，作者，联系方式等

```python
import re
```

包引用，导入文件需要的各种包文件

```python
def SplitWords(InputFile):

    #读入文件
    fileobject = open(InputFile)
    try:
        alltext = fileobject.read()
    finally:
        fileobject.close()

    #分割单词
    words = re.split('[^a-zA-Z]+', alltext)

    #统计单词词频
    dic = {}
    for word in words:
        if word in dic.keys():
            dic[word]+= 1
        else:
            dic[word] = 1

    #排序
    result = sorted(dic.items(), key=lambda dic:dic[1], reverse=True)

    print(result[1:100])
```

分词函数，用于实现我们实际的程序功能

```python
if __name__ == '__main__':
    SplitWords('input.txt')
```

主函数，程序的入口，相当于c++中的main函数

```python
def SplitWords(InputFile):

    #读入文件
    fileobject = open(InputFile)
    try:
        alltext = fileobject.read()
    finally:
        fileobject.close()

    #分割单词
    words = re.split('[^a-zA-Z]+', alltext)

    #统计单词词频
    dic = {}
    for word in words:
        if word in dic.keys():
            dic[word]+= 1
        else:
            dic[word] = 1

    #排序
    result = sorted(dic.items(), key=lambda dic:dic[1], reverse=True)

    print(result[1:100])
```

文件读入

分割单词，使用正则表达式

词频统计，利用词典类型进行判重的操作

排序，这里用python内置的排序算法来实现

性能测试工具

Profile是Python语言内置的性能分析工具，它能够有效地描述程序运行的性能状况，提供各种统计数据帮助程序员找出程序中的性能瓶颈。

```
import profile

def profileTest():
    Total =1;
    for i in range(10):
        Total=Total*(i+1)
        print(Total)
    return Total

if __name__ == "__main__":
    profile.run("profileTest()")
```

左图是一个求阶乘的小程序，Profile工具仅需要一行代码就可以对所测试函数进行代码性能测试。

```
E:\Python\python.exe E:/venv/Profilet
1
2
6
24
120
720
5040
40320
362880
3628800
         15 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(exec)
       10    0.000    0.000    0.000    0.000 :0(print)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 Profiletest.py:7(profileTest)
        1    0.000    0.000    0.000    0.000 profile:0(profileTest())
        0    0.000             0.000          profile:0(profiler)
```

程序输出结果

程序执行时间

详细的函数性能数据报表

ncalls 函数的被调用次数
tottime 函数总计运行时间，这里除去函数中调用的其他函数运行时间

percall 函数运行一次的平均时间，等于tottime/ncalls
cumtime 函数总计运行时间，这里包含调用的其他函数运行时间

percall 函数运行一次的平均时间，等于cumtime/ncalls
filename:lineno(function) 函数所在的文件名，函数的行号，函数名

```
       902806 function calls (902803 primitive calls) in 4.562 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 :0(_getdefaultlocale)
    31    0.000    0.000    0.000    0.000 :0(append)
                        ......
500000    1.141    0.000    1.141    0.000 :0(keys)
 41/40    0.000    0.000    0.000    0.000 :0(len)
     4    0.000    0.000    0.000    0.000 :0(min)
     1    0.000    0.000    0.000    0.000 :0(open)
     4    0.000    0.000    0.000    0.000 :0(ord)
     1    0.000    0.000    0.000    0.000 :0(print)
     1    0.094    0.094    0.094    0.094 :0(read)
                        ......
     1    0.750    0.750    1.203    1.203 :0(sorted)
     1    0.312    0.312    0.312    0.312 :0(split)
                        ......
     1    0.000    0.000    0.000    0.000 sre_parse.py:750(parse)
     2    0.000    0.000    0.000    0.000 sre_parse.py:90(__init__)
```

程序总耗时 4.562s

Keys()函数执行500000次，耗时 1.141s

输出几乎不耗时

输入耗时 0.094s

排序函数，耗时1.203s

分词函数，耗时 0.312s

案例：原因分析

为什么 keys() 函数的调用复杂度过高?

> keys(): Return a new view of the dictionary's keys (docs.python.org)

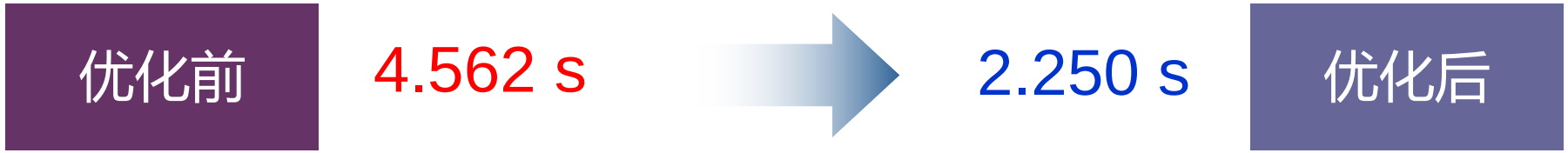
- 原因：每调用一次keys()函数，系统就会生成一个新的字典迭代器，如果这个生成过程重复50万次，……
- 优化：使用 in 操作符直接代替keys()，不再每次生成新的迭代器。

```
8  def SplitWords(InputFile):
9
10     #读入文件
11     fileobject = open(InputFile)
12     try:
13         alltext = fileobject.read()
14     finally:
15         fileobject.close()
16
17     #分割单词
18     words = re.split('[^a-zA-Z]+', alltext)
19
20     #统计单词词频
21     dic = {}
22     for word in words:
23         #if word in dic.keys():
24         if word in dic:
25             dic[word]+= 1
26         else:
27             dic[word] = 1
28
29     result = sorted(dic.items(), key=lambda dic:dic[1], reverse=True)
30
31     print(result[1:100])
```

```
402806 function calls (402803 primitive calls) in 2.250 seconds
```

测试发现：程序的总执行时间降低到 2.250s

案例：优化结果



- 性能优化的关键是如何发现问题，寻找解决问题的方法。
- 有效的测试是不可缺少的，通过测试找出真正的瓶颈，并分析优化结果
- 要避免不必要的优化，避免不成熟的优化，不成熟的优化是错误的来源

Python 代码性能优化

改进算法，选择合适的数据结构

- 良好的算法对性能起到关键作用，因此性能改进的首要点是对算法改进
- 算法时间复杂性的排序依次是

  O(1) ➔ O(lg n) ➔ O(n lg n) ➔ O(n^2) ➔ O(n^3) ➔ O(n^k) ➔ O(k^n) ➔ O(n!)

- 对成员的查找访问等操作，字典（dictionary）要比列表（list）更快
- 集合（set）的并、交、差的操作比列表（list）的迭代要快

Python 代码性能优化

循环优化的基本原则：尽量减少循环过程中的计算量，在多重循环的时候，尽量将内层的计算提到上一层。

字符串的优化：　Python的字符串对象是不可改变的。字符串连接的使用尽量使用 join() 而不是 +。当对字符串可以使用正则表达式或者内置函数处理时，选择内置函数。

使用列表解析和生成器表达式：列表解析要比在循环中重新构建一个新的 list 更为高效，因此可以利用这一特性来提高运行的效率。

谢谢大家！

THANKS