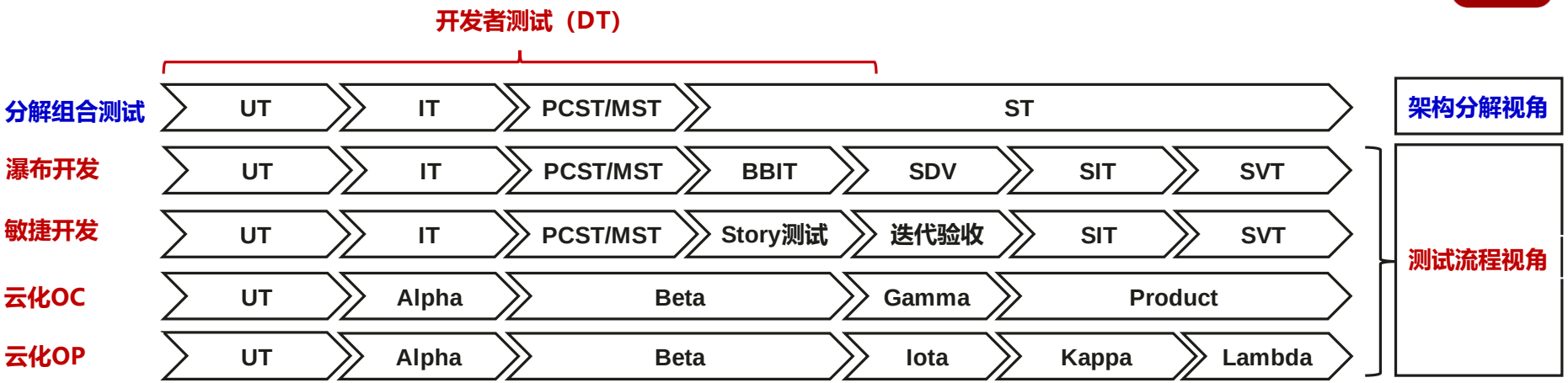
开发者测试赋能 (C/C++)

HUAWEI

目录

HUAWEI

开发者测试（DT）定义



- 开发者测试（Developer Test，DT），是指**开发者所做的测试**，有别于专职测试人员进行的测试活动。**主要是PC级仿真测试，少量的真实环境测试。**
- 除了少数联调用例，DT主要**关注点**是**自己开发的功能**是否OK，自己开发的功能可能是一小块，但是还是需要**UT、IT、PCST、少量真实环境测试**（业界UT:IT:ST比例是7:2:1，但需要结合业务特点选择合适比例，测试分层那门课有讲），目的是**除了验证新开发的单元功能ok外，还要通过集成验证与周边的接口是否ok，以及是否破坏其他功能。**

HUAWEI

开发者测试（DT）的价值

● 测试前移，提前发现问题

后端问题解决的成本远大于前端。

✓ **对开发者测试的要求：**有测试框架支撑，能够高效的编写/调试/运行测试代码，测试框架能够支撑85%及以上场景测试

● 持续集成（CI）门禁

多人协同开发，采用分支开发合并到主干集成的方式，分支时间越长，代码越多集成到主干越困难，可能会出现编译失败、功能冲突等问题，**敏捷推荐分支代码持续集成到主干**，为了保证主干健康，需要**门禁防护网**（静态检查、动态检查），动态检查主要就是自动化测试用例，DT测试用例不上门禁，本地测得再好，集成到主干还是各种问题，所以**DT测试用例必须上门禁**

✓ **对开发者测试的要求：**满足门禁要求，测试工程编译+运行5分钟左右，测试用例自动化校验，行覆盖率80%以上

● 重构防护网

提升可维护性的核心思想就是封装隔离变化，但对于变化的识别很难做到100%准确，如果将所有可能出现的变化都封装隔离又会导致过度设计。基于敏捷思想，我们要简单设计，然后对新出现的变化进行封装隔离，所以要持续重构，重构要求每一步的功能不发生变化，除了重构16字真言（旧的不变，新的创建，一步切换，旧的再见）的指导外，还需要每次修改完代码都运行一下测试用例，确保功能不变。

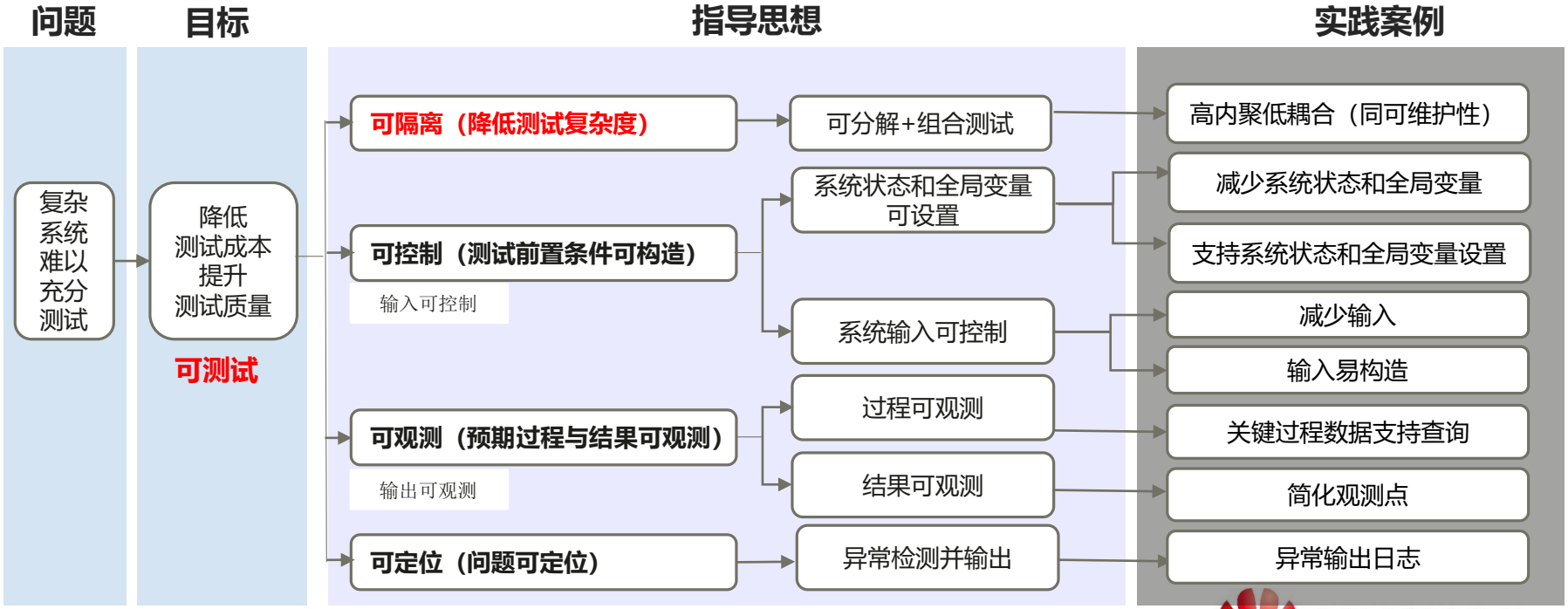
✓ **对开发者测试的要求：**本地测试工程编译+运行足够高效（UT 30s，IT 3分钟，PCST 5分钟），测试用例自动化校验

目录

HUAWEI

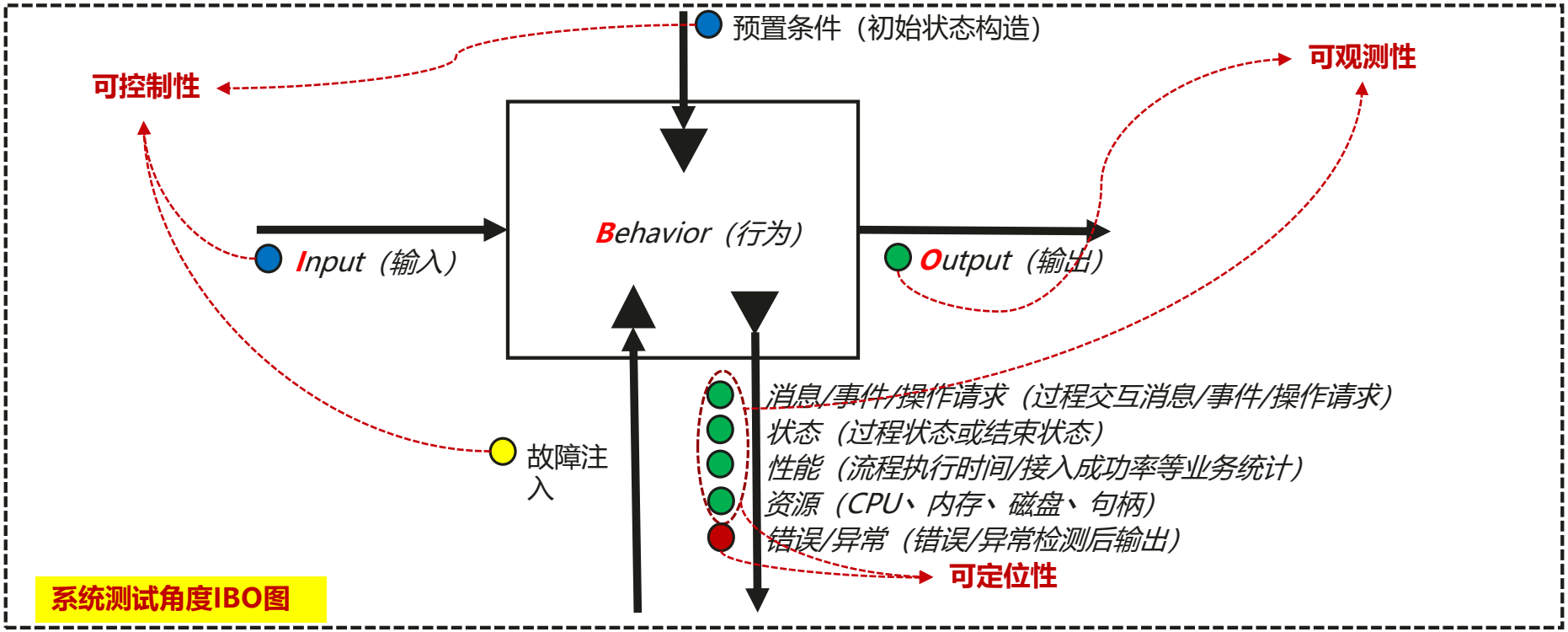
可测试性的定义、指导思想和实例

- 可测试性：软件发现故障并隔离、定位故障的能力，以及在一定的时间和成本前提下，进行测试设计、测试执行的能力
- 可测试性好的代码，更容易被测试，测试出来的问题更容易定位，可以降低测试成本，提升测试质量。
- 可测试性主要包括可隔离性、可控制性、可观测性、可定位性。



HUAWEI

从IBO模型导出可控制性、可观测性、可定位性的定义



- **可控制性**：对广义**输入**的控制能力；**可观测性**：对广义**输出**的观察能力；**可定位性**：**输出**信息**支撑问题分析**的能力
- **被测对象粒度**不一样，边界不一样，**IBO**的定义也有所**差异**，比如：

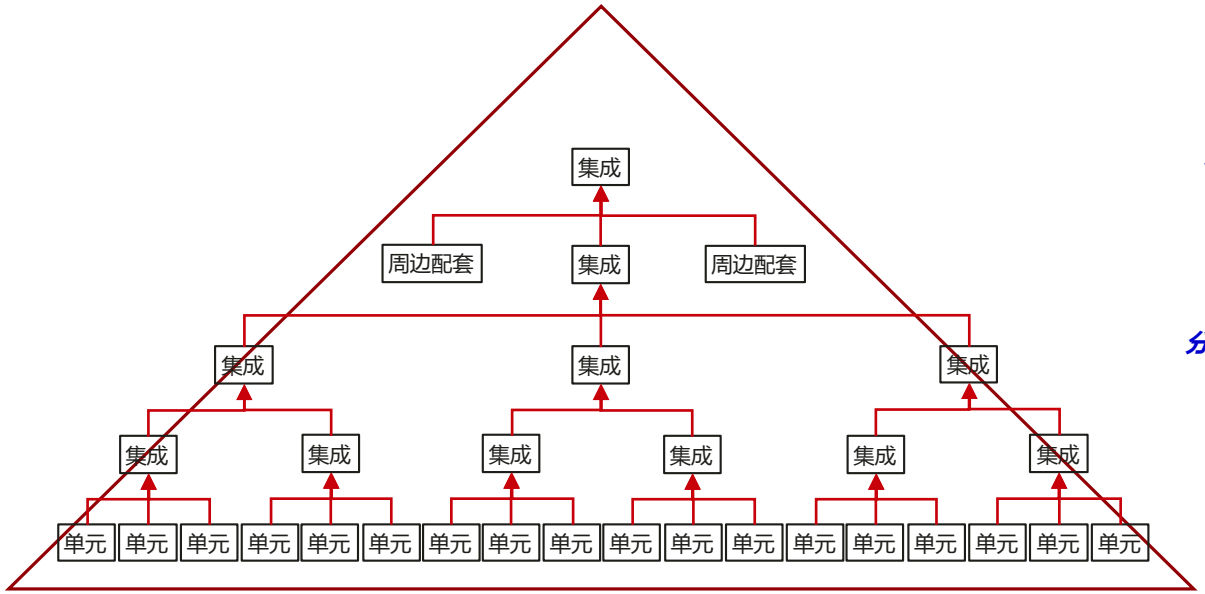
  组件IT测试，组件对外交互消息是**输入输出**；SDV测试，组件交互消息是**处理过程**。

HUAWEI

目录

HUAWEI

为什么要测试分层



后测

分层守护

先测

SVT：测试人员测试

SDV/SIT：测试人员测试

PCST/部分SDV：开发人员测试

IT：开发人员测试

UT：开发人员测试

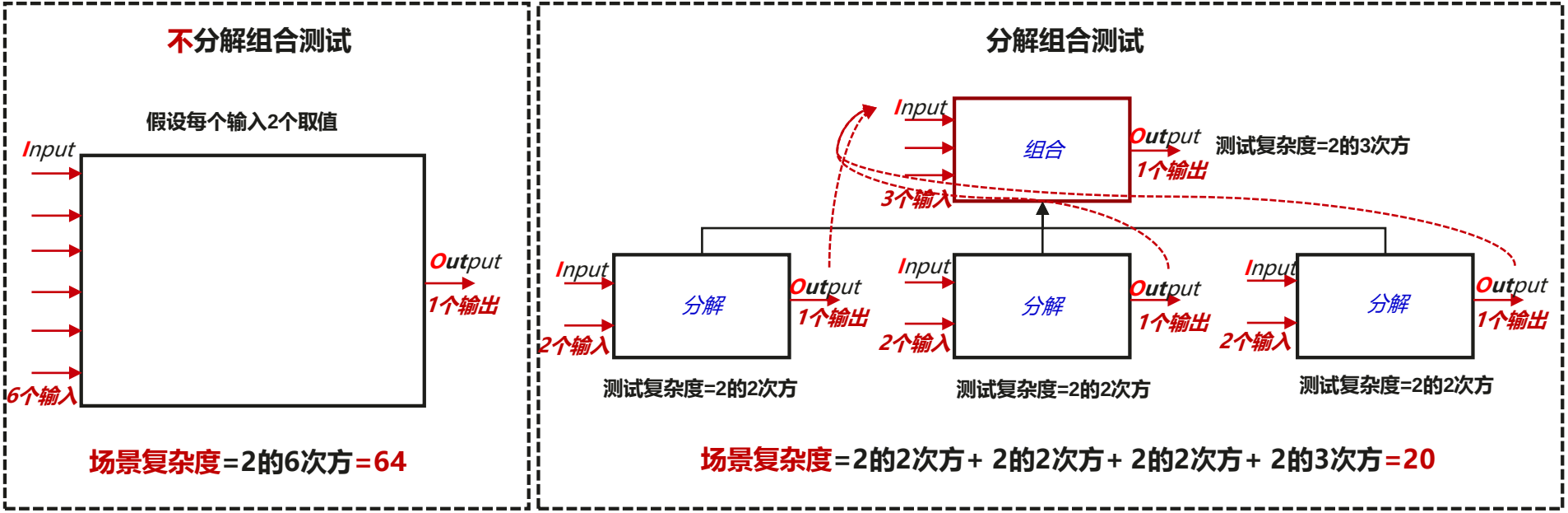
开发者测试（DT）

- **测试分层基于分解组合测试的思想（分治法），利用分治法，降低测试复杂度**
- **问题越遗留到后端解决成本越高，分层守护降低了问题解决成本**

理想的测试分层呈金字塔状（业界测试金字塔UT:IT:ST为7：2：1），与分治法的树形对应

HUAWEI

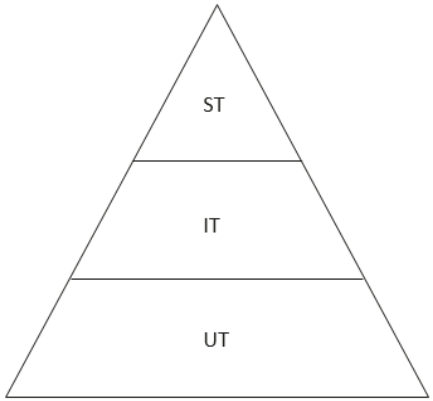
分解组合测试降低测试复杂度的证明



- 分解组合测试，利用分治法，降低测试复杂度，类似算法思想里面的分治法（n方->n*logn）
- 分解组合测试与系统架构分解组合对应

HUAWEI

测试分层（通用模型）



ST（系统测试）：系统/子系统内多个或所有组件/模块/微服务的集成测试（PCST/MST），也可能是更大粒度的全系统端到端测试（SDV/SIT/SVT）

IT（集成测试），基于组件/模块/微服务接口的测试，即组件/模块/微服务内的集成测试

UT（单元测试），组件/模块/微服务内粒度较小的功能接口测试，例如：C头文件对外暴露接口，C++类公共接口

这里给出的是抽象的测试分层，具体测试分层怎么分、分几层与产品的规模、形态有关

比如对于规模非常小的系统，分1~2层就够了，对于规模很大的系统，可能需要分5~6层，甚至更多。

- **UT（单元测试）**

  组件/模块/微服务内小粒度功能接口的测试，内部调用的函数不需要测试，因为功能接口相对稳定，针对功能接口的用例也相对稳定。面向对象语言，主要测试类对外暴露的公共接口；对于面向过程的语言（C语言等），主要测试头文件中对外暴露的接口。

  UT测试也需要做测试设计，且要从黑盒（功能）角度设计，从输入**(I)**、处理**(B)**、预期输出**(O)**的角度进行用例分析设计，测试覆盖率仅作为反馈，用于分析哪些功能场景没有覆盖到，从而指导测试设计优化并补充测试用例。

- **IT（集成测试）**

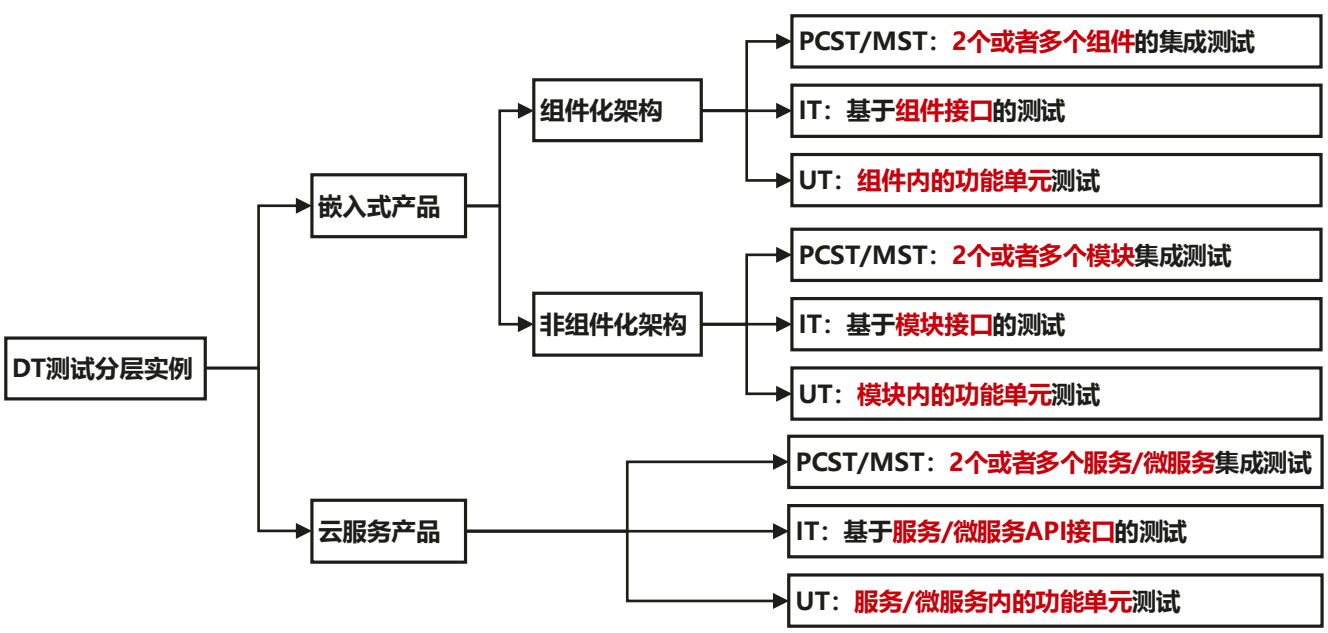
  基于组件/模块/微服务接口的测试，即组件/模块/微服务内的集成测试，一般基于个人级PC测试工程测试。

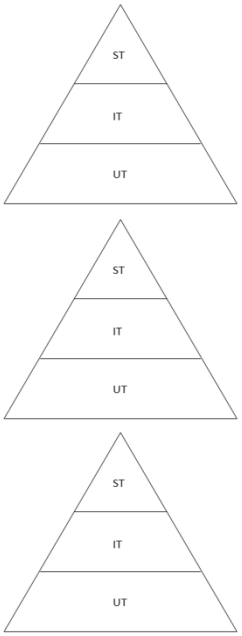
- **ST（系统测试）**

  子系统内多个或所有组件/模块/微服务的集成测试（PCST/MST），也可能是更大粒度的全系统端到端测试（SDV/SIT/SVT）。

HUAWEI

测试分层（实例，DT部分）

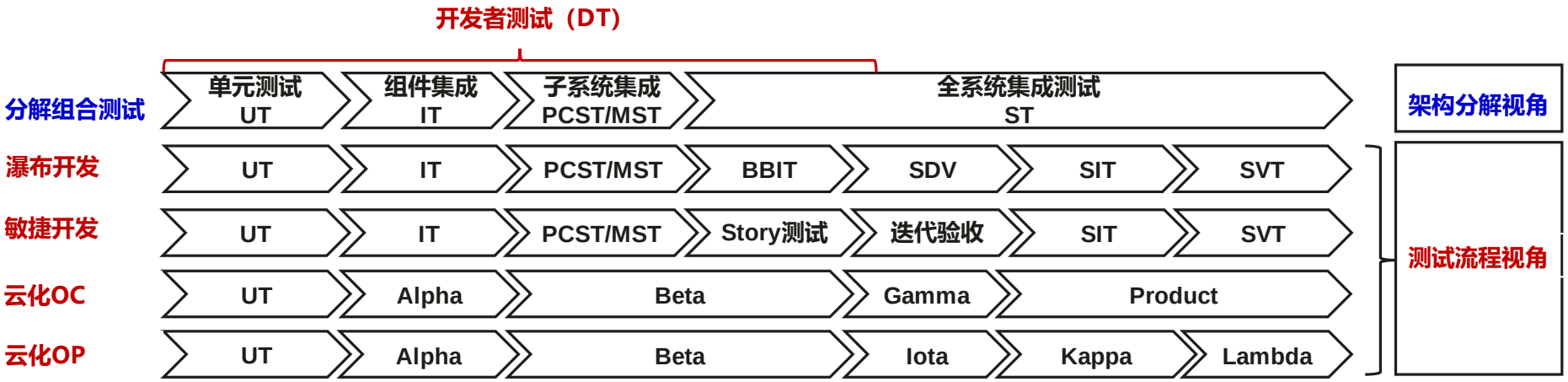




◆ 仅针对开发者测试（DT）涉及的分层实例，不包含后端的测试

HUAWEI

测试分层（实例，包含后端测试）

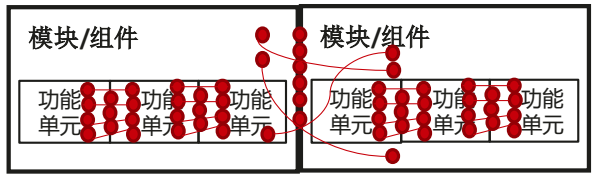


备注：
云化OP：On Premise 指服务软件部署于客户侧，含云和非云（以云为主）等情况；
云化OC：On Cloud 指服务软件部署于公有云上，为多家运营商提供服务。

- **架构分解视角**的测试分层，基于分解组合测试的思想，与架构分解组合对应
- **测试流程视角**的测试分层，基于测试流程先后顺序
- **架构分解视角**的测试分层与**测试流程视角**的测试分层通常可以对应起来

HUAWEI

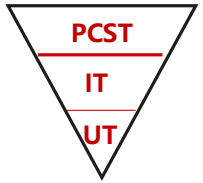
选择合适的测试分层比例

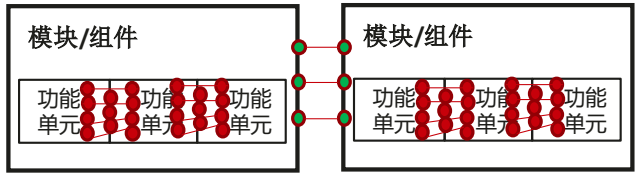


功能单元之间、模块/组件之间耦合大，边界不清晰，血肉模糊

成本导向

- 功能单元、模块/组件边界模糊，分解成本大
- 功能单元、模块/组件边界不稳定，维护成本大
- 强拆的成本大于分解组合（分治法）测试的收益

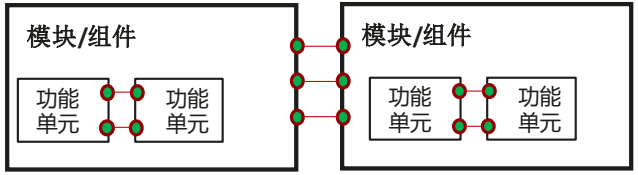




功能单元之间耦合大，边界不清晰，模块/组件之间耦合小，边界清晰

成本导向

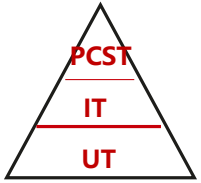
- 功能单元边界模糊，分解成本大
- 功能单元边界不稳定，维护成本大
- 功能单元强拆成本大



功能单元之间耦合小，边界清晰，模块/组件之间耦合小，边界清晰

成本导向

- 功能单元、模块/组件边界清晰，分解成本小
- 功能单元、模块/组件边界稳定，维护成本小



耦合比较大，不要强拆；提升可隔离性是关键。

HUAWEI

目录

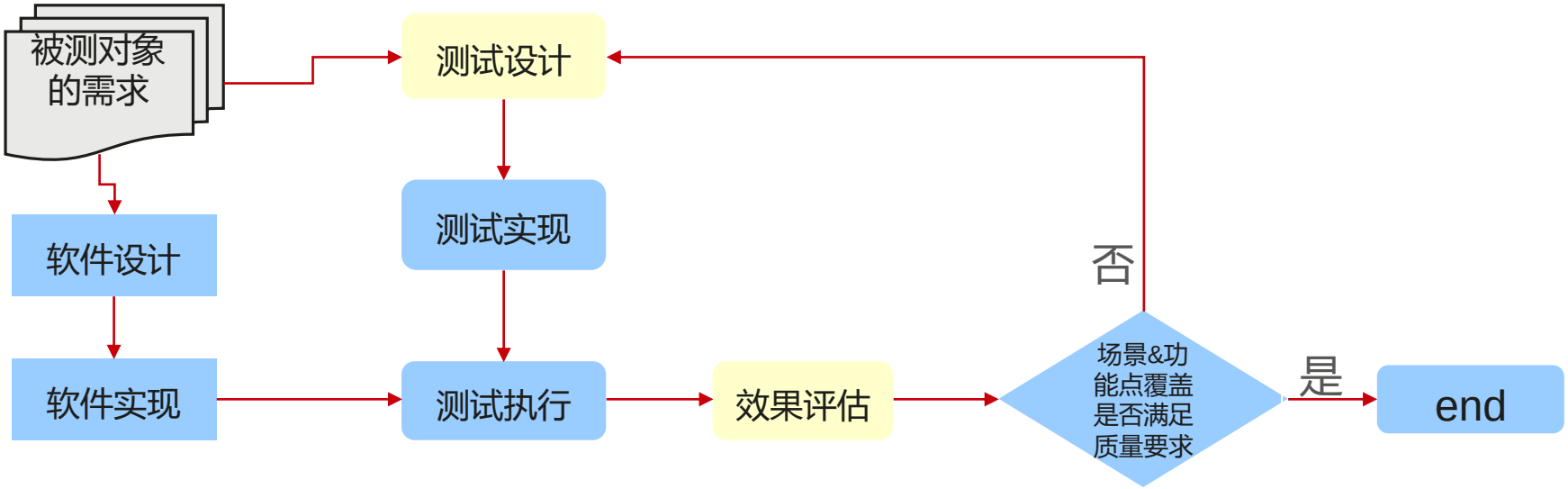
HUAWEI

测试设计的目的

1. 测试设计：基于被测对象的**测试依据，**通过一定的分析设计方法得到测试用例的活动。这些测试用例可以**实现特定（**与被测对象本身的风险程度相匹配**）的测试覆盖。**
2. 测试设计着眼于如下三方面的问题：
   - 识别影响被测对象行为的测试因子，避免遗漏
   - 通过合理的方法对测试因子的取值进行选取，并进行合理组合
   - 明确合理的预期结果
3. 好的测试设计可以获得更优的**测试性价比：在有限的时间内，高质量**测试设计输出的测试用例往往能够**更合理**对被测对象的功能点 & 场景进行覆盖，满足交付质量要求。
4. 虽然我们提倡测试性价比，但我们并不提倡在一个测试用例中测试过多的内容，每个用例应该有自己明确的测试目的。

HUAWEI

开发者测试设计所处的位置和大致过程

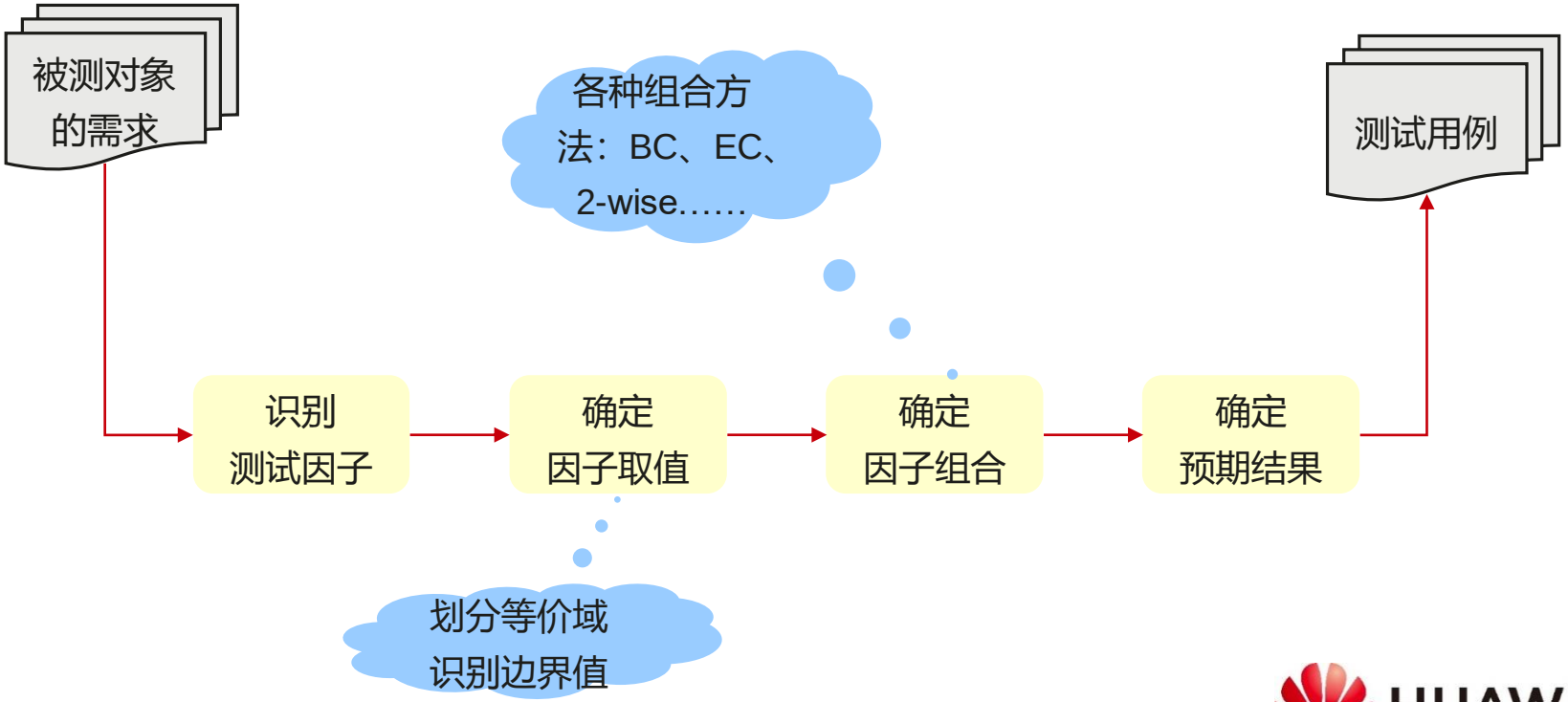


测试设计可看做一个**两阶段循环**的过程：

1. 阶段一：基于**被测对象的需求**进行测试设计。
2. 阶段二：基于**代码覆盖率**进行评估，根据重要程度，对遗漏的功能和异常测试点进行必要补充。

HUAWEI

测试设计的一般步骤

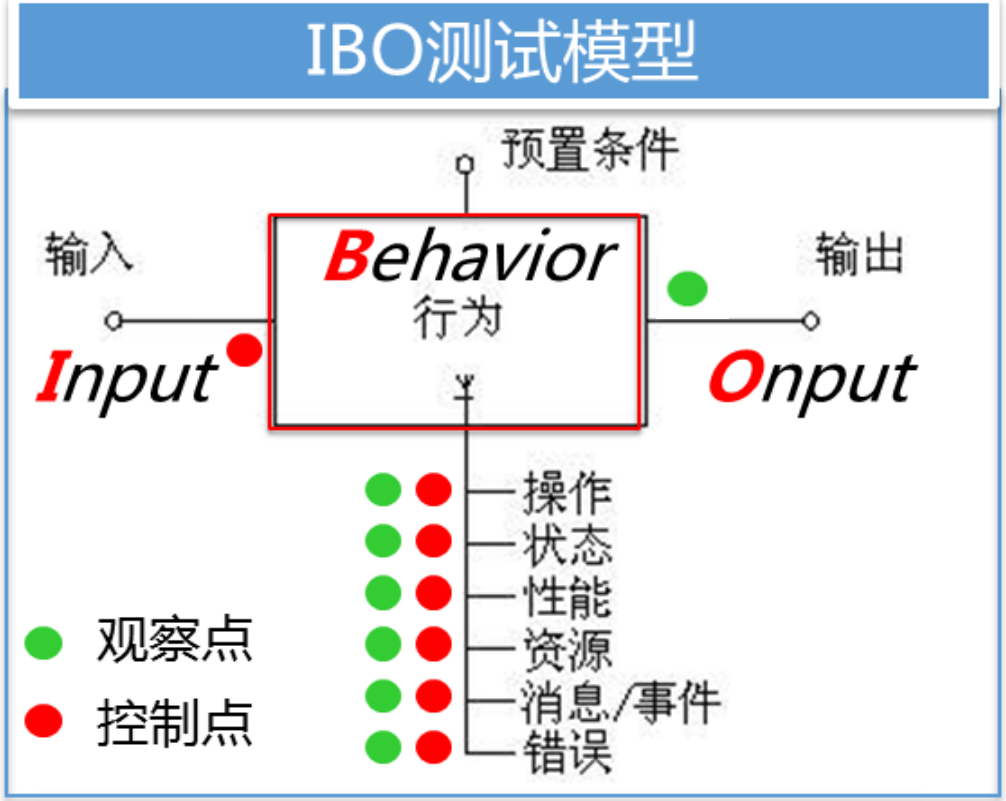


HUAWEI

目录

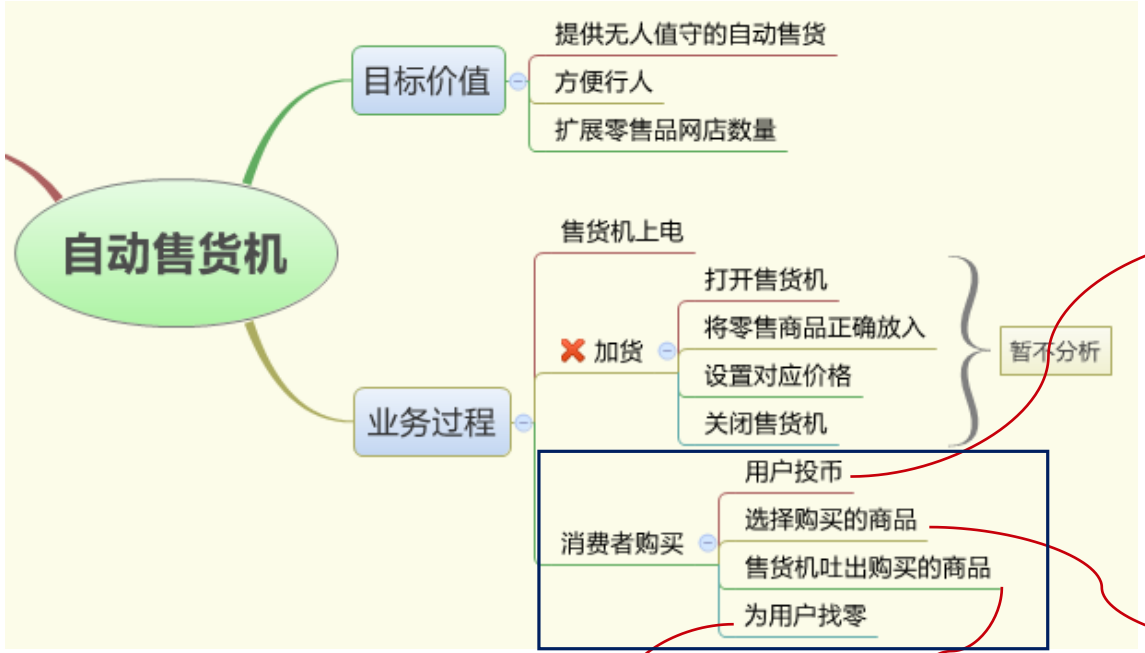
HUAWEI

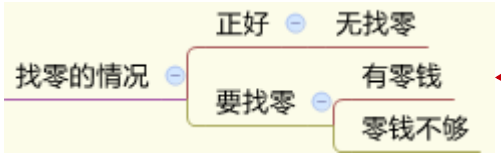
通过IBO模型理解被测特性

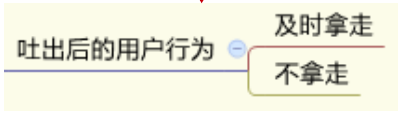


HUAWEI

通过思维导图打开操作步骤识别测试因子







参考以前的经验库识别有没有遗漏的因子

因子	因子取值	公共检查点
接口类型	VLANIF接口（SUPERVLAN）	预置条件，不涉及
	QINQ子接口	
	QINQ终结子接口	
	非对称QINQ终结子接口	
	ETH-TRUNK主接口	
	ETH-TRUNK QINQ子接口	
	ETH-TRUNK QINQ终结子接口	
	非对称ETH-TRUNK QINQ终结子接口	
	对称ETH-TRUNK QINQ终结子接口	
	POS主接口	
ETH TRUNK接口模式	手工ETH-TRUNK模式	
	静态ETH-TRUNK模式（LACP）	
POS接口模式	HDLC	
	PPP	

公共测试因子 / SR公共测试因子 / MR公共测试因子 / 配置表 / 测试因子标准化印

基本思路总结：

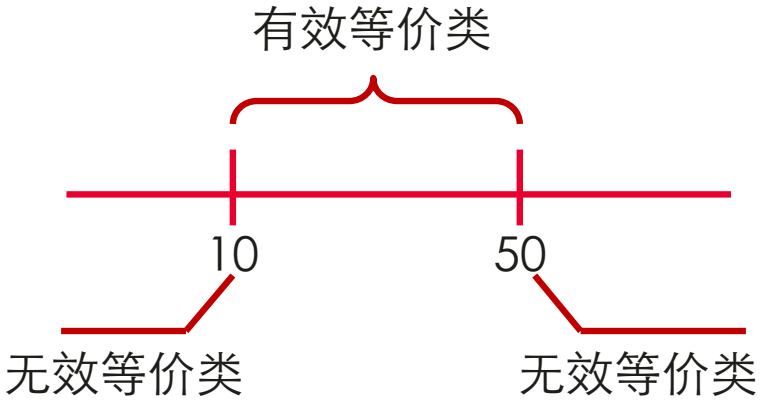
- **首先自己通过系统化的方法识别因子；**
- **在自己实在想不出来后，参考以前的经验库，或其他人的评审补充因子**

HUAWEI

确定因子取值——划分等价类

等价类

按因子的约束对其取值范围进行等价类划分，等价类中任意选取的数据对系统功能的影响是相同的，通过对多个等价类中有代表性的数据的覆盖达到对该测试输入的覆盖。



一个大于等于10，小于等于50的因子的等价类

有效等价类

对于程序的规格而言是有意义、合理的数据输入的集合，用于测试程序是否实现了规格说明中约定的功能和性能要求。

无效等价类

对于程序的规格说明而言是不合理的数据输入，用于测试程序是否能经受得起非法输入的考验。

HUAWEI

等价类划分的一些常见原则

原则1：输入条件是一个布尔量，可获得一个有效等价类和一个无效等价类。

信号处理，只处理高电平输入，不处理低电平输入

1：高电平

0：低电平

有效等价类	无效等价类
1：高电平	0：低电平

原则2：输入条件规定的取值范围是一个闭合区间，可获得一个有效等价类和两个无效等价类；如果取值范围为开区间，则可获得一个有效等价类和一个无效等价类

控制块索引合法性检查，合法的控制块索引需要满足如下条件：

索引范围大于等于**0**，小于等于**99**

有效等价类	无效等价类
0 <= 索引范围 <=99	索引范围 < 0
	索引范围 >99

HUAWEI

等价类划分的一些常见原则

原则3：输入条件规定了输入数据必须遵守的规则，可获得一个符合所有规则的有效等价类和若干个从不同角度违反规则的无效等价类

银行卡密码合法性检查，银行卡密码需要满足两个条件：

- 全数字组合
- 密码长度为**6**位

有效等价类	无效等价类
长度为6位的全数字组合	长度大于6位的全数字组合
	长度小于6位的全数字组合
	非全数字组合

（注：实际中，建议更优的方式，是按照密码字符集、密码长度两个因子分别取值，再组合）

原则4：输入条件规定了输入数据的一组n个值且**分别处理**，可获得n个有效等价类和一个无效等价类

公司招聘录用条件为：

- 硕士生
- 博士生

有效等价类	无效等价类
硕士生	其他人员
博士生	

说明：如果我们知道已经划分的等价类里面的不同元素，在系统内部实际的处理方式是不同的，则应该进一步细分为不同的等价类

HUAWEI

确定因子取值——分析边界值

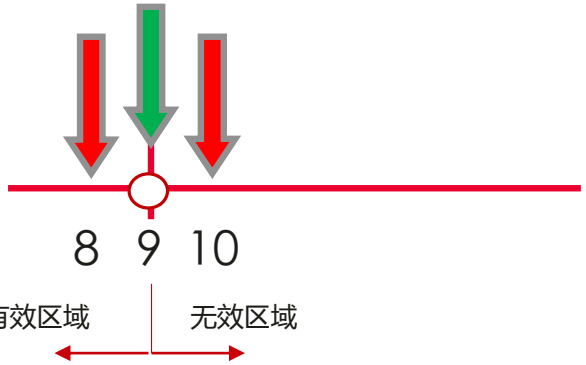
经验告诉我们，**大量的错误是发生在输入或输出范围的边界上**，边界值分析就是在划分的等价类区域的边界及其附近进行测试数据的选取。是对等价类划分的一种补充。

几条经验：

1. 如果输入条件规定了值的范围，则应取刚达到这个范围的边界的值，以及刚刚超越这个范围边界的值作为测试输入数据。
2. 如果输入是有序集合，则应选取集合的第一个元素和最后一个元素作为测试输入数据。

HUAWEI

从实例看边界值的重要性



测试输入条件：int X < 9

正确的代码	If (x < 9) {
	  return T;
	}
	else {
	  return F;
	}
		预期：
		8-T 9-F 10-F

错误情况1	If (x <= 9) {
	  return T;
	}
	else {
	  return F;
	}
		8-T 9-T 10-F
		取值为9的用例会失败

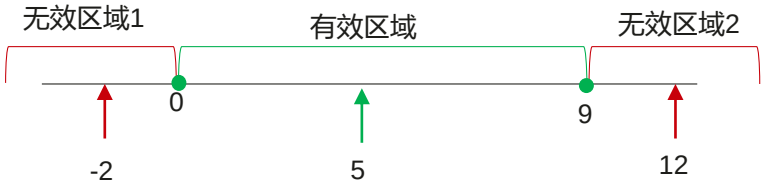
错误情况2	If (x > 9) {
	  return T;
	}
	else {
	  return F;
	}
		8-F 9-F 10-T
		取值为8的用例会失败

错误情况3
 	
注：有些语言，不等于的表达是 <>
	If (x != 9) {
	  return T;
	}
	else {
	  return F;
	}
		8-T 9-F 10-T
		取值为10的用例会失败

HUAWEI

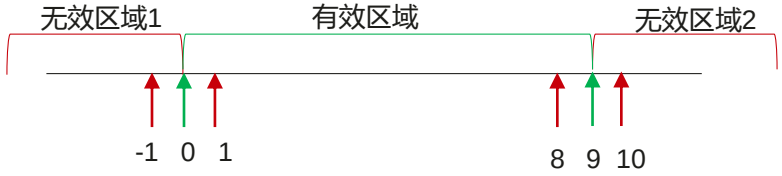
确定因子取值——综合运用

应用举例：
参数的合法范围为： [0,9] 类型为int



基于边界值分析对取值进行补充：
根据经验：输入条件规定了值的范围，应取刚达到这个范围的边界的值，以及刚刚超越这个范围边界的值作为测试输入数据。
因子范围为[0,9]，如下图所示，则边界取值可以补充如下6个边界点：

1. -1：下边界外
2. 0：下边界
3. 1：下边界内
4. 8：上边界内
5. 9：上边界
6. 10：上边界外



仅基于等价类划分的思路输出等价类取值列表：

有效等价类		无效等价类	
编号	取值	编号	取值
1	5	2	-2
		3	12

有效等价类		无效等价类	
编号	取值	编号	取值
1	0	6	-2
2	1	7	-1
3	5	8	10
4	8	9	12
5	9		

因子取值部分总结

1. 等价类划分是通过划分等价域减少因子的取值，边界值分析是为了找到更容易出错的取值。两者往往结合使用，针对一个单因子获取合理的取值。
2. 被测对象有多个测试因子时，需要在对每个因子合理取值的基础上，进一步采用因子组合技术确定测试输入，因子组合技术将在下节详细介绍。

复习题

请使用之前学习过的等价类划分 & 边界值分析方法，为下面例子输出等价类列表：

登录系统的账号需要满足如下规则：

1. 小写字母的组合
2. 账号字符长度大于等于6位，小于等于12位

HUAWEI

因子取值分析（总结 & 复习）

参考答案

有效等价类			无效等价类		
编号	取值	备注	编号	取值	备注
1	abcdefg	非边界等价类	6	abcd	非边界无效等价类，违反规则 x >= 6
2	abcdef	下边界有效等价类	7	abcdefghijkmnopq	非边界无效等价类,违反规则x <= 9
3	abcdefghijkm	上边界有效等价类	8	Abcdefg	带大写字母的无效等价类
4	abcdefg	下内边界的有效等价类	9	abcdef12	带数字的无效等价类
5	abcdefghijk	上内边界的有效等价类	10	abcdef_#	带特殊字符的无效等价类
			11	abcde	下外边界的无效等价类
			12	abcdefghijkmn	上外边界的无效等价类

思路解读：

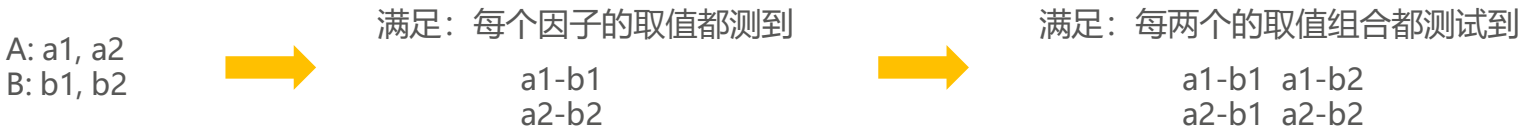
1、首先根据等价类的规则得到一个有效等价类和N个从不同角度违反规则的无效等价类：有效等价类1，无效等价类（6，7，8,9,10）

2、基于边界值分析补充：补充了有效等价类（2，3，4，5），和无效等价类（11，12）

HUAWEI

因子组合技术

- 因子组合技术是解决多个测试因子如何组合成测试用例的技术。
- 因子组合的目的是为了满足一定的覆盖，例如



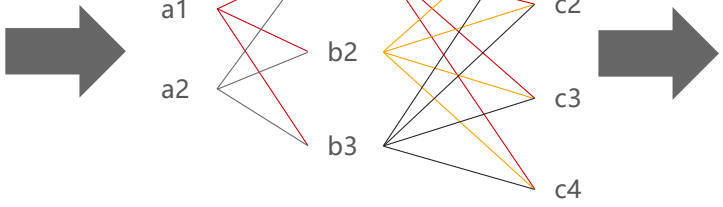
- 本节主要介绍4种常用的组合技术：
  1. AC **（All Combinations）**
  2. EC（ Each Choice ）
  3. BC（Basic Choice）
  4. N-wise（主要介绍pair-wise）
- 选择哪种组合技术依赖于被测对象的风险（对于开发者测试，可以同时考虑实现成本和执行成本）

HUAWEI

AC因子组合技术

- 定义：该方法要求将每个测试因子所有取值进行全组合。AC是覆盖最全面的覆盖方式。
- 优点：覆盖全面
- 缺点：对于复杂的测试对象，测试因子多，且因子取值也较多时，组合数量会太多

因子编号	取值个数	取值1	取值2	取值3	取值4
A	2	a1	a2		
B	2	b1	b2	b3	
C	3	c1	c2	c3	c4



a1-b1-c1 a2-b1-c1
a1-b1-c2 a2-b1-c2
a1-b1-c3 a2-b1-c3
a1-b1-c4 a2-b1-c4
a1-b2-c1 a2-b2-c1
a1-b2-c2 a2-b2-c2
a1-b2-c3 a2-b2-c3
a1-b2-c4 a2-b2-c4
a1-b3-c1 a2-b3-c1
a1-b3-c2 a2-b3-c2
a1-b3-c3 a2-b3-c3
a1-b3-c4 a2-b3-c4

组合个数 = 2 × 3 ×4 = 24

（每个因子取值个数的乘积）

HUAWEI

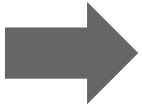
BC因子组合技术

- 定义：BC要求先确定一个基本的测试因子组合作为基本用例，然后以基本组合为基础，每次只改变一个测试因子来构造新的测试组合，直到遍历完每个因子的每个组合。

Setp1：首先确立一个基本用例，如左图示意的基本用例。
Step2：以基本用例为基础，每次改变一个测试因子的取值，输出一个新的测试用例
step3：重复step2直到所有因子取值都被覆盖为止

因子编号	取值个数	取值1	取值2	取值3	取值4
A	2	a1	a2		
B	3	b1	b2	b3	
C	4	c1	c2	c3	c4

基本用例



① a1, b1, c1
② a2, b1, c1
③ a1, b2, c1
④ a1, b3, c1
⑤ a1, b1, c2
⑥ a1, b1, c3
⑦ a1, b1, c4

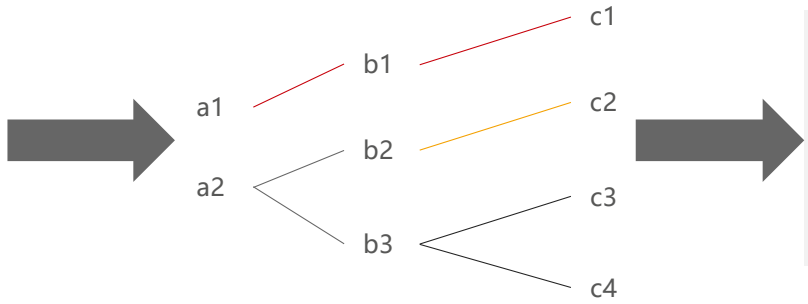
组合个数 = 所有因子取值个数 - 因子个数 + 1

HUAWEI

EC因子组合技术

- 定义：每一个测试因子的每一个取值在所有测试因子组合中至少出现一次。

因子编号	取值个数	取值1	取值2	取值3	取值4
A	2	a1	a2		
B	3	b1	b2	b3	
C	4	c1	c2	c3	c4



① a1，b1，c1
② a2，b2，c2
③ a2，b3，c3
④ a2，b3，c4

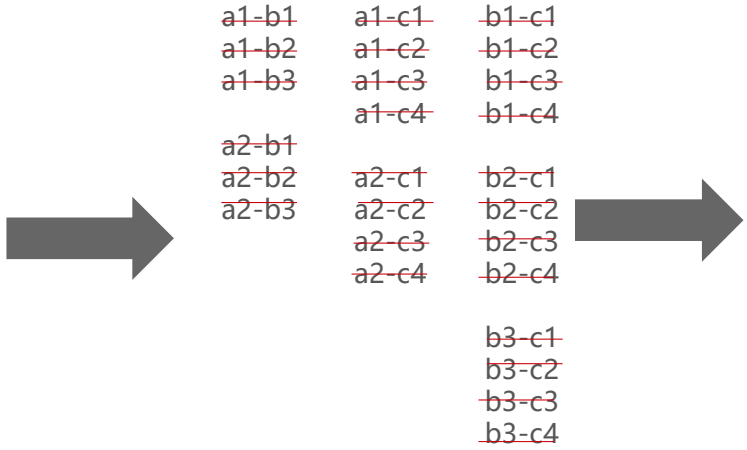
组合个数 = 所有因子中取值最多的那个因子的取值个数

HUAWEI

pair-Wise（2-wise）组合

- 定义：每两个测试因子的取值组合均至少覆盖一次
- 业界研究：2-wise是目前综合性价比最高的组合算法

因子编号	取值个数	取值1	取值2	取值3	取值4
A	2	a1	a2		
B	3	b1	b2	b3	
C	4	c1	c2	c3	c4



a1，b1，c1
a2，b2，c2
a1，b2，c4
a1，b3，c3
a2，b1，c4
a2，b3，c1
a1，b1，c2
a2，b2，c3
a2，b1，c3
a1，b2，c1
a2，b3，c2
a1，b3，c4

组合个数 = 最多因子取值个数 × 次多因子取值个数

HUAWEI

N-Wise组合技术

- 定义：每N个测试因子的取值组合均至少覆盖一次
- EC实际是1-wise，pair-wise实际是2-wise，业界常用的还有3-wise，其他很少使用

因子编号	取值个数	取值1	取值2	取值3	取值4
A	2	a1	a2		
B	2	b1	b2	b3	
C	3	c1	c2	c3	c4

HUAWEI

因子组合技术（总结 & 复习）

总结：

1. 等价类划分+边界值分析用于确定单因子的数据取值；
2. 当被测对象因子有多个因子时，通过因子组合技术确定测试输入；
3. 常见因子组合技术有：**EC**、**BC**、**AC**、**2-wise**等；
4. 业界研究：**pair-wise**是目前综合性价比最高的组合方法；
5. 手工输出测试因子组合容易出错，公司内有多种**用例设计辅助工具**可供选用。如：**PICT**（微软公司的一个小工具），**MTG**（我司自研的可视化基于模型辅助设计工具），**iCase**（轻量的基于**excel**的辅助设计工具）。可通过下面链接做进一步了解：

HUAWEI

因子组合技术（总结 & 复习）

- **复习题**：被测对象拥有4个测试因子，每个测试因子分别拥有3个等价类取值（包括有效等价类和无效等价类），请分别描述EC、BC、AC、Pair-wise方法输出的用例集合各有几个用例？并尝试使用pair-wise方法输出用例集合。

因子	取值1	取值2	取值3
A	a1	a2	a3
B	b1	b2	b3
C	c1	c2	c3
D	d1	d2	d3

HUAWEI

因子组合技术（总结 & 复习）

参考答案

1. AC方法对应组合个数 = 3*3*3*3 = 81
2. EC方法对应组合个数 = MAX（A，B，C，D）= 3
3. BC方法对应组合个数 = SUM(A, B,C,D) – 4 + 1 = 12 – 4 +1 = 9
4. Pair-wise对应组合= 3 * 3 = 9
5. Pair-wise组合列表

因子组合	A	B	C	d
1	a1	b1	c1	d1
2	a1	b2	c2	d2
3	a1	b3	c3	d3
4	a2	b1	c2	d3
5	a2	b2	c3	d1
6	a2	b3	c1	d2
7	a3	b1	c3	d2
8	a3	b2	c1	d3
9	a3	b3	c2	d1

HUAWEI

目录

HUAWEI

如何理解代码覆盖评估方法

- 我们常说的代码覆盖率，是一种对代码覆盖进行评估和分析的方法。
- 在测试执行后，通过对没有覆盖的部分进行评估，可以用来查找先前设计的用例还有哪些地方需要完善。
- 本节将介绍几种常见的代码覆盖评估方法：

1. 语句覆盖（statement coverage）
2. 判定覆盖（Decision coverage）
3. 条件覆盖（Condition coverage）
4. 判定/条件覆盖（Decision/Condition coverage）
5. 条件组合覆盖（branch condition combination）
6. 路径覆盖（Path coverage）

HUAWEI

语句覆盖

定义：程序中的每条可执行语句都能被执行到

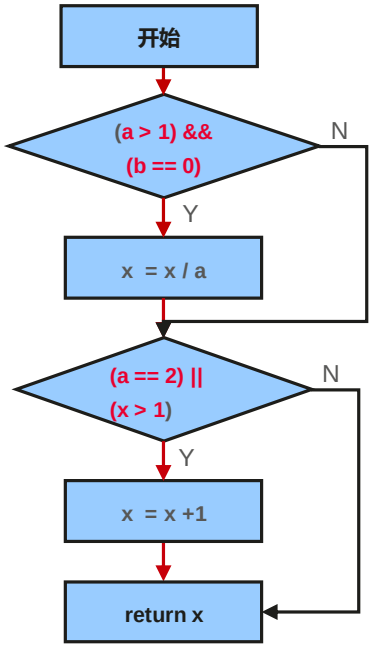
被测代码

```
int func(int a, int b, int x)
{
    if ((a > 1) && (b == 0)) {
        x  = x / a;
    }
    if ((a == 2) || (x > 1)) {
        x = x + 1;
    }
    return x;
}
```

测试输入： {a=2, b=0, x=1}

```
1:    4:int func(int a, int b, int x)
-:    5:{
1:    6:    if((a > 1) && ( b == 0)){
1:    7:        x = x / a;
-:    8:    }
-:    9:
1:   10:    if((a == 2) || ( x > 1)){
1:   11:        x = x + 1;
-:   12:    }
1:   13:    return x;
-:   14:}
```

使用gcov生成的结果



即使是100%语句覆盖，也不能说明被测代码没有问题，而且也不能对代码的修改100%看护。

比如，本页的示例中，我们将(a > 1) && (b == 0)修改为(a > 1) || (b == 0), 用例依然会执行成功！

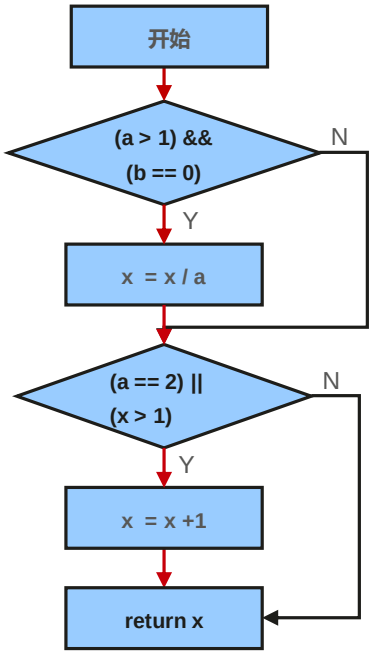
HUAWEI

判定覆盖

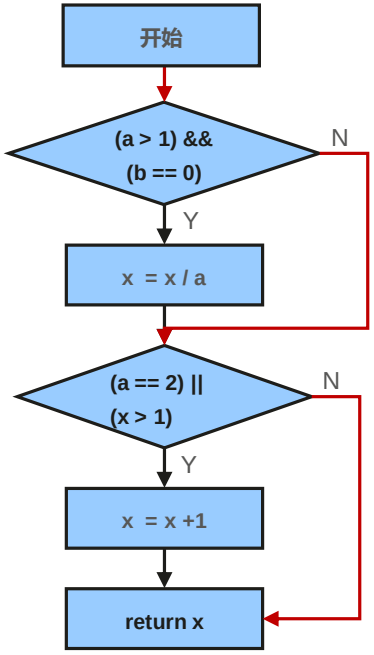
定义：程序中每个判断语句的每个分支都能得到至少一次覆盖。也称之为分支覆盖（branch coverage）

测试输入	判定1： **(a > 1) && (b == 0)**	判定2： **(a == 2) || (x > 1)**
{a=2, b=0, x=2}	T	T
{a=1, b=1, x=1}	F	F

- 和语句覆盖类似，100%判定覆盖也不能说明代码没问题，也不能看护所有的修改。比如，我们将二个判断语句修改成(a != 2) || (x > 1)，当前的两个用例仍然可以执行成功。
- 满足判定覆盖的一定可以满足语句覆盖。



用例1



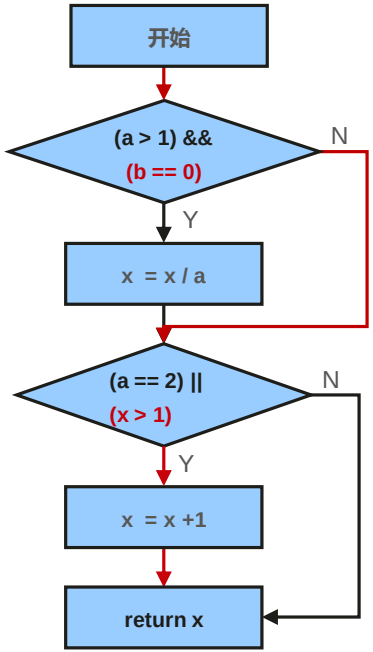
用例2

HUAWEI

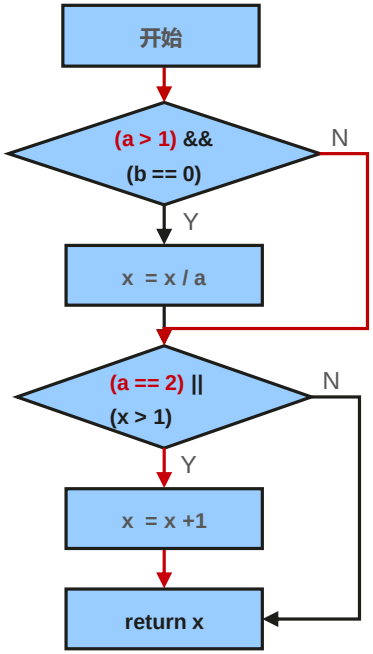
条件覆盖

定义：判断语句中的每个条件的“真”“假”取值都能至少被覆盖一次。条件覆盖不能保证判定覆盖。

	{a=1, b=0, x=3}	{a=2, b=1, x=1}
条件1 : **a > 1**	F	T
条件2: **b == 0**	T	F
条件3: **a == 2**	F	T
条件4: **x > 1**	T	F
判定1: **(a > 1) && (b == 0)**	F	F
判定2: **(a == 2) || (x > 1)**	T	T



用例1

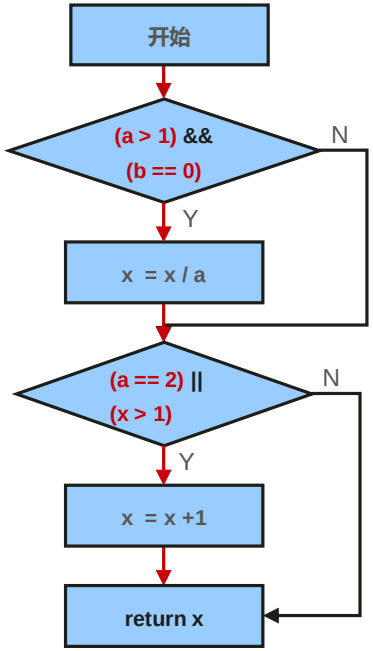


用例2

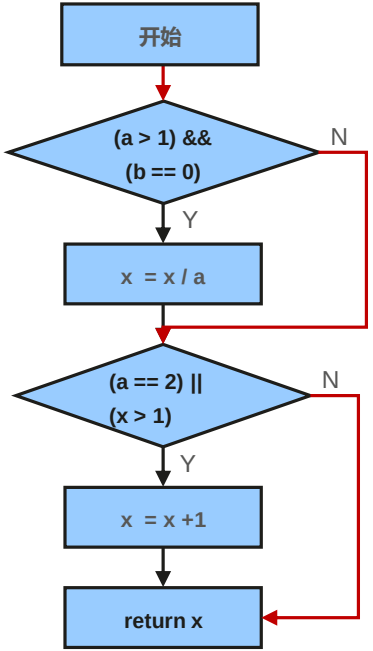
判定/条件覆盖

定义：判断语句中的每个条件的“真” “假”取值都能至少被覆盖一次，且每个判定语句的可能结果也能至少覆盖 一次，即：同时满足条件覆盖与判定覆盖。

		{a=2, b=0, x=4}	{a=1, b=1, x=1}
判定1: **(a > 1) && (b == 0)**	条件1 : **a > 1**	T	F
	条件2: **b == 0**	T	F
	判定1的结果	T	F
判定2: **(a == 2) || (x > 1)**	条件3: **a == 2**	T	F
	条件4: **x > 1**	T	F
	判定2的结果	T	F



用例1



用例2

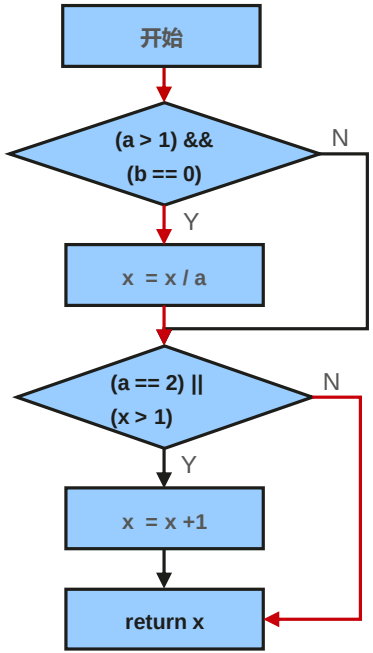
缺陷：对于判断语句(a == 2) || (x > 1)，如不小心修改了x >1这个条件，当前用例是没法发现的。

条件组合覆盖

定义：每个判定的条件取值组合至少能被覆盖一次

		{a=2, b=0, x=2}	{a=2, b=1, x=1}	{a=1, b=0, x=2}	{a=1, b=1, x=1}
判定1: **(a > 1) && (b == 0)**	条件1 : **a > 1**	T	T	F	F
	条件2: **b == 0**	T	F	T	F
	判定1的结果取值	T	F	F	F
判定2: **(a == 2) || (x > 1)**	条件3: **a == 2**	T	T	F	F
	条件4: **x > 1**	T	F	T	F
	判定2的结果取值	T	T	T	F

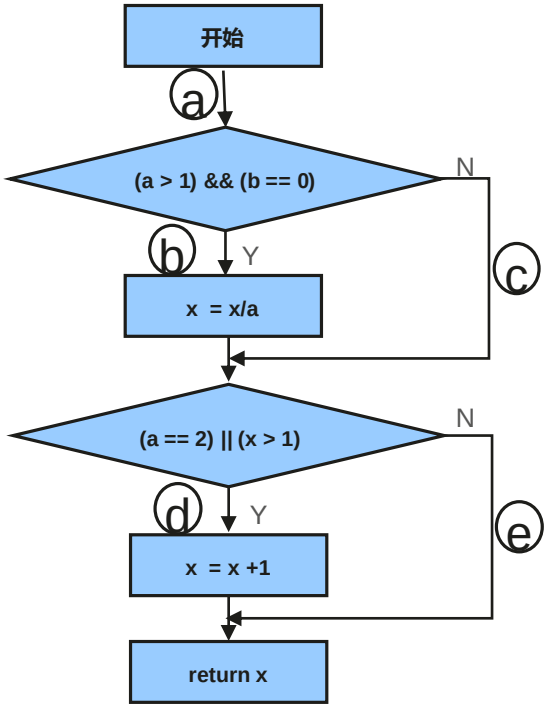
缺陷：虽然所有的条件都能被覆盖一次，所有的判定条件也被覆盖了至少一次，但程序执行路径上却没有覆盖下面这条路径(参考红色路径)。



注意：并非所有条件组合都能找到方法进行覆盖，假定有一个分支判断语句形式为:if(c == ‘a’|| c == ‘b’)，则我们是找不到任何一个c的取值能同事满足两个条件都为true的

路径覆盖

定义：程序中每条路径都能尽量被覆盖至少一次。路径覆盖不一定满足条件组合覆盖。



		{a=2, b=0, x=2}	{a=2, b=1, x=1}	{a=1, b=0, x=1}	{a=3, b=0, x=1}
判定1: **(a > 1) && (b == 0)**	条件1 : **a > 1**	T	T	F	T
	条件2: **b == 0**	T	F	T	T
	判定1的结果取值	**T**	**F**	**F**	**T**
判定2: **(a == 2) || (x > 1)**	条件3: **a == 2**	T	T	F	F
	条件4: **x > 1**	T	F	F	F
	判定2的结果取值	**T**	**T**	**F**	**F**
覆盖路径		abd	acd	ace	abe

HUAWEI

总结和练习

- 没有一种覆盖技术是100%能防止任何缺陷引入的。
- 代码覆盖率的意义仅仅在于分析出程序中未被覆盖的代码逻辑，从而对当前的测试用例进行评估和补充。

复习题：使用前面介绍的6种白盒测试设计技术为下面的代码片段设计测试输入，使得这六种判定覆盖率尽可能的高。

```
int Count(string str) {
    int words = 0;
    char last = ' ';
    for (string::size_type i = 0; i < str.length(); i++) {
        if (last == 'r' || last == 's') {
            words++;
        }
        last = str[i];
    }
    if (last == 'r' || last == 's') {
        words++;
    }
    return words;
}
```

Count函数接收一个string对象str，计算以‘s’或者‘r’结尾的子串数量。

比如：

1、“rcrc”，输出2

2、“123s“,输出1

3、“123”， 输出0

建议：可以尝试着先从**黑盒角度**设计一下用例，然后使用不同的**白盒测试设计**的方法分析覆盖情况后进行用例补充。

总结和练习参考答案：

① 语句覆盖：

用例编号	str	用例预期
1	“rar”	2

② 判定覆盖：

用例编号	Str	用例预期
1	“rara”	2
2	“r”	1

③ 条件覆盖：

用例编号	Str	用例预期
1	“ra”	1
2	“ar”	1
3	“sa”	1
4	“as”	1

④ 判定/条件覆盖：

用例编号	Str	用例预期
1	“ra”	1
2	“ar”	1
3	“sa”	1
4	“as”	1

⑤ 条件组合覆盖：

用例编号	Str	用例预期
1	“ras”	2
2	“sar”	2
3	“ra”	1

⑥ 路径覆盖：

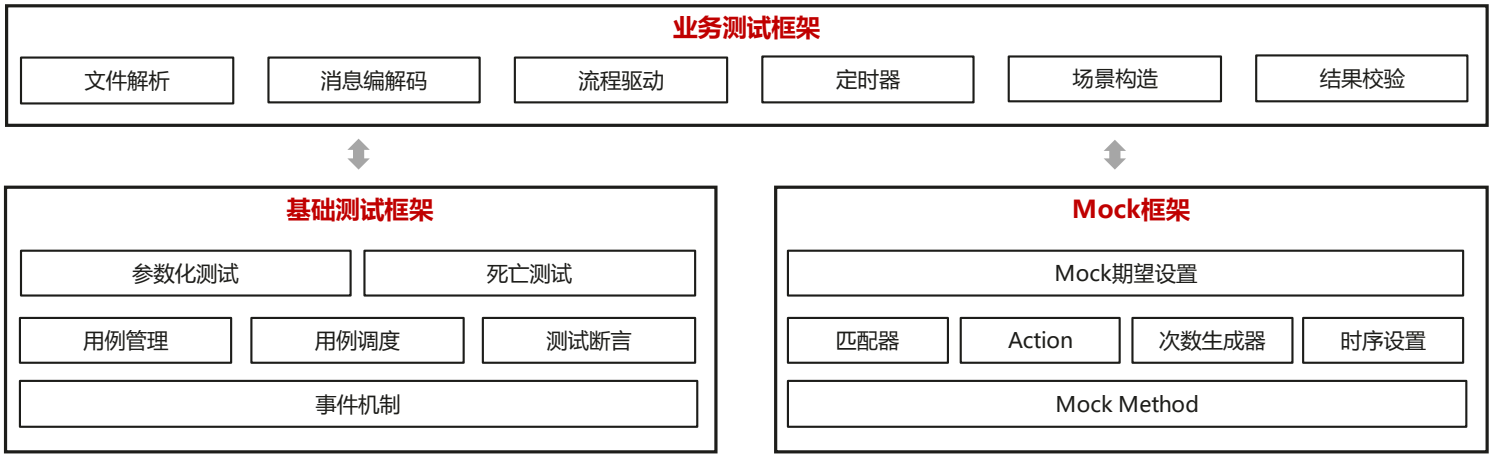
用例编号	Str	用例预期
1	“rs”	2
2	“sa”	1

注意！对于当前例题，实际上我们没法找到一个用例满足100%的条件组合覆盖，因为我们没法找到一个字符同时满足last == ‘r’|| last == ‘s’

目录

HUAWEI

测试框架简介



- 基础测试框架：提供测试例执行及验证机制
- Mock框架：提供屏蔽周边依赖对象的测试模拟机制
- 业务测试框架：提供适合业务产品特点的测试例验证机制

HUAWEI

常用的C/C++测试框架

工具种类	工具名称	操作环境		支持语言	License	社区活跃度	点评
		Win	Linux				
自研工具	CodeTest	★	★	C/C++	自研	2012工具部维护	以前叫HUTAF LLT，集成了cppunit、gtest测试框架，可以与HUTAF配合使用，支持覆盖率统计，支持真实设备运行，支持动态打桩和动态加载，但断言、测试宏功能不丰富。
	DTCenter	★	★	C/C++	自研	无线工具部维护	集成mockcpp（同时支持C与C++的mock）、Amocker(仅支持C的mock)支持C与C++打桩，但目前主要在无线产品线使用
开源工具	GoogleTest	★	★	C/C++	BSD	★★★★★	支持多种平台，支持用例自发现，丰富的断言功能，支持类型参数化、死亡测试等。C++领域知名度高，可以和gmock配合，对C++代码打桩，对C的打桩支持不友好。
	CppUnit	★	★	C/C++	GNU LGPL	★★★☆☆	不支持Mock功能，可通过集成Gmock使用，增加了太多的功能导致框架复杂，已被作者放弃，并重新编写了CppUnitLite。
	CUnit	★	★	C	GNU LGPL	★★☆☆☆	可移值性好，无用例管理功能，用例为C语言，不支持mock功能。
	CxxTest	★	★	C/C++	/	★★★☆☆	可移植性好，控制异常的能力及断言功能都不错，但需要自己注册测试用例，且需要用到perl/python对测试用例进行扫描，仅支持致命断言，仅支持全局函数mock。

✓ **优先**选择业界广泛使用的开源工具和商业工具。

目录

HUAWEI

gtest简介

Google的开源C++单元测试框架Google Test，简称gtest，它是一套用于编写C/C++测试用例的框架，可以运行在很多平台上（包括Linux、Mac OS X、Windows、Cygwin等等）。

gtest开源项目中包含两套框架，一个是gtest测试框架，一个是gmock框架，用来做单元测试，gmock则主要用来mock(模拟)待测试模块依赖的对象，以便去除测试中不必要的依赖。

gtest相比其他工具的优势：

1. 可移植性好，不需要Exception或RTTI，可以在多种操作系统上运行。
2. 拥有强大的断言系统，支持非致命断言，因此可以在同一个测试用例中支持多次失败检测。
3. 支持测试用例自动检测，并自动注册到测试框架中。
4. 使用程度高，社区活跃度较高，有非常详细的帮助指导文档，很容易获取到外部资源。

gtest测试框架提供了事件机制、断言、运行参数、参数化、及死亡测试等主要功能。

官方源码： https://github.com/google/googletest/blob/master/googletest

gtest事件机制演示

```
// 全局事件定义
class FooEnvironment : public testing::Environment
{
  public:
    virtual void SetUp() {
        cout << "Foo FooEnvironment SetUp" << endl;
    }
    virtual void TearDown() {
        cout << "Foo FooEnvironment TearDown" << endl;
    }
};

int main(int argc, char* argv[])
{
    testing::AddGlobalTestEnvironment(new
FooEnvironment);
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

```
// 测试套与测试用例事件
class TestSuite : public Test
{
public:
    static void SetUpTestCase()
    {
        cout << "TestSuite测试套事件：在第一个testcase之前执行" << endl;
    }
    static void TearDownTestCase()
    {
        cout << "TestSuite测试套事件：在最后一个testcase之后执行" << endl;
    }
    virtual void SetUp()
    {
        cout << "TestSuite测试用例事件：在每个testcase之前执行" << endl;
    }
    virtual void TearDown()
    {
        cout << "TestSuite测试用例事件：在每个testcase之后执行" << endl;
    }
};
```

```
// 测试用例
TEST_F(TestSuite, Test1_OK)
{
    cout << "第一个用例" << endl;
}
```

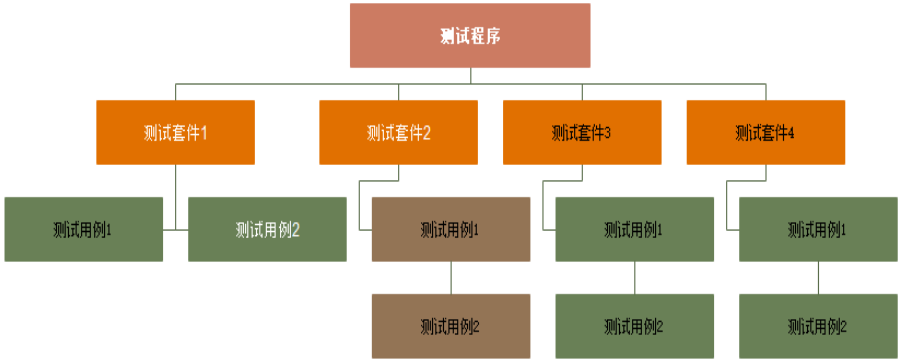
```
// 输出结果
Foo FooEnvironment SetUp
TestSuite测试套事件：在第一个testcase之前执行
TestSuite测试用例事件：在每个testcase之前执行
"第一个用例"
TestSuite测试用例事件：在每个testcase之后执行
TestSuite测试套事件：在最后一个testcase之后执行
Foo FooEnvironment TearDown
```

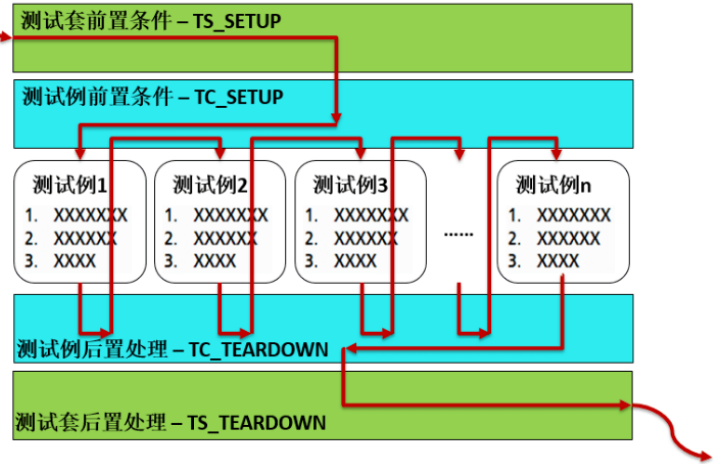
HUAWEI

gtest事件机制

gtest事件机制给框架使用者提供了在用例执行前后执行用户自定义操作的机会，gtest共提供了3种事件机制：

1. 全局级别(测试程序级)：在所有用例执行前，及所有用例执行完成后。
2. TestSuite级别(套件级)：在测试套件第一个用例执行前，最后一个用例执行后。
3. TestCase级别(用例级)：在每个用例执行前与执行后。





左图描述了测试程序、测试套件、测试用例之间的关系，**右图**描述了测试套件事件与测试用例事件的执行顺序

HUAWEI

gtest事件机制之全局事件

1. 实现全局事件必须写一个类，继承testing::Environment类，实现里面的SetUp和TearDown方法
2. SetUp()方法在所有用例执行前执行，TearDown()方法在所有用例执行后执行

```
class FooEnvironment : public testing::Environment
{
  public:
    virtual void SetUp() {
        std::cout << "Foo FooEnvironment SetUp" << std::endl;
    }
    virtual void TearDown() {
        std::cout << "Foo FooEnvironment TearDown" << std::endl;
    }
};

int main(int argc, char* argv[])
{
    testing::AddGlobalTestEnvironment(new FooEnvironment); //全局事件，还要在main函数里加入这句话
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

实现类后，还需要告诉gtest添加这个全局事件，我们需要在main函数中通过testing::AddGlobalEnvironment方法将事件挂到框架中，也就是说，我们可以写很多个这样的类，然后将他们的事件都挂上去。

HUAWEI

gtest事件机制之TestSuite事件

- 写一个类，继承testing::Test，然后实现两个静态方法
- SetUpTestCase() 方法在第一个TestCase之前执行，gtest新版本已更名为SetUpTestSuite()
- TearDownTestCase() 方法在最后一个TestCase之后执行，gtest新版本已更名为TearDownTestSuite()

```
class FooTest : public testing::Test
{
 protected:
   static void SetUpTestCase()
   {
      shared_resource_ = new xxx;
   }
   static void TearDownTestCase()
   {
      delete shared_resource_;
      shared_resource_ = NULL;
   }
   static T* shared_resource_;
};
TEST_F(FooTest,  CaseOne)
{
   // you can refer to shared_resource here
}
TEST_F(FooTest,  CaseTwo)
{
   // you can refer to shared_resource here
}
```

gtest事件机制之TestCase事件

- TestCase事件是挂在每个用例执行前后，需要SetUp方法和TearDown方法
- SetUp()在每个TestCase之前执行 TearDown()在每个TestCase之后执行

```
class FooCalcTest: public testing::Test
{
 protected:
   virtual void SetUp()
   {
      m_foo.Init();
   }
   virtual void TearDown()
   {
      m_foo.Finalize();
   }
    FooCalc m_foo;
};

TEST_F(FooCalcTest, HandleNoneZeroInput)
{
   EXPECT_EQ(4, m_foo.Calc(12, 16));
}
TEST_F(FooCalcTest, HandleNoneZeroInput_Error)
{
   EXPECT_EQ(5, m_foo.Calc(12, 16));
}
```

gtest断言

gtest中的断言是一些与函数调用相似的宏,要测试一个类或函数，我们需要对其行为做出断言，当一个断言失败时，会在屏幕上输出该代码所在的源文件及其所在的位置行号。gtest提供二类断言，ASSERT_*版本的断言失败时会产生致命失败，并结束当前函数。EXPECT_*版本的断言产生非致命失败，而不会中止当前函数。通常更推荐使用EXPECT_*断言，这样运行一个测试用例时可以有不止一个的错误被报告出来。但如果在编写断言失败，就没有必要继续往下执行的测试时，就应该使用ASSERT_*断言。

gtest断言主要有以下功能：

1. 基本断言
2. 二进制比较
3. 字符串比较
4. 浮点检查
5. 显式返回成功或失败
6. 异常检查
7. Predicate Assertions
8. 类型检查
9. Windows HRESULT assertions

HUAWEI

gtest断言之基本断言

基本断言实现了基本的true/false条件测试：

致命断言	非致命断言	验证条件
ASSERT_TRUE(condition)	EXPECT_TRUE(condition)	condition is true
ASSERT_FALSE(condition)	EXPECT_FALSE(condition)	condition is false

记住，**当它们失败时，ASSERT_*产生一个致命失败并从当前函数返回，而EXPECT_*产生一个非致命失败，允许函数继续运行。**在两种情况下，一个断言失败都意味着它所包含的测试失败。

HUAWEI

gtest断言之二进制比较

致命断言	非致命断言	验证条件
ASSERT_EQ(expected,actual)	EXPECT_EQ(expected,actual)	expected == actual
ASSERT_NE(val1, val2)	EXPECT_NE(val1, val2)	val1 != val2
ASSERT_LT(val1, val2)	EXPECT_LT(val1, val2)	val1 < val2
ASSERT_LE(val1, val2)	EXPECT_LE(val1, val2)	val1 <= val2
ASSERT_GT(val1, val2)	EXPECT_GT(val1, val2)	val1 > val2
ASSERT_GE(val1, val2)	EXPECT_GE(val1, val2)	val1 >= val2

在出现失败事件时，gtest会将两个值（Val1和Val2）都打印出来。在ASSERT_EQ和EXPECT_EQ断言（以及后面介绍的其他断言）中，你应该把你希望测试的表达式放在actual（实际值）的位置上，将其期望值放在expected（期望值）的位置上，因为gtest的测试消息为这种惯例做了一些优化。

另外，该断言支持自定义的数据类型，但是必须自己重载类型相应的操作符例如<,>。

HUAWEI

gtest断言之字符串比较

致命断言	非致命断言	验证条件
ASSERT_STREQ(*expected_str*,*actual_str*)	EXPECT_STREQ(*expected_str*,*actual_str*)	the two C strings have the same content
ASSERT_STRNE(*str1*, *str2*)	EXPECT_STRNE(*str1*, *str2*)	the two C strings have different content
ASSERT_STRCASEEQ(*expected_str*,*actual_str*)	EXPECT_STRCASEEQ(*expected_str*,*actual_str*)	the two C strings have the same content, ignoring case
ASSERT_STRCASENE(*str1*, *str2*)	EXPECT_STRCASENE(*str1*, *str2*)	the two C strings have different content, ignoring case

- 注意断言名称中出现的“**CASE**”意味着大小写被忽略了。
- *STREQ*和*STRNE*也接受宽字符串（wchar_t*）。如果两个宽字符串比较失败，它们的值会做为UTF-8窄字符串被输出。

  一个NULL空指针和一个空字符串会被认为是不一样的。

HUAWEI

gtest断言之浮点检查

致命断言	非致命断言	判断条件
ASSERT_FLOAT_EQ(expected, actual)	EXPECT_FLOAT_EQ(expected, actual)	The two float values are almost equal
ASSERT_DOUBLE_EQ(expected, actual)	EXPECT_DOUBLE_EQ(expected, actual)	the two double values are almost equal
ASSERT_NEAR(val1, val2,abs_error)	EXPECT_NEAR(val1, val2,abs_error)	the difference between val1 and val2 doesn't exceed the given absolute error

对相近的两个数比较：

致命断言	非致命断言	判断条件
ASSERT_NEAR(val1, val2,abs_error)	EXPECT_NEAR(val1, val2,abs_error)	the difference between val1 and val2 doesn't exceed the given absolute error

同时还可以使用如下表达式进行float和double的比较：

EXPECT_PRED_FORMART2(testing::FloatLE, val1, val2);

EXPECT_PRED_FORMART2(testing::DoubleLE, val1, val2);

HUAWEI

gtest断言之高级功能

断言官方文档页：https://github.com/google/googletest/blob/master/googletest/docs/advanced.md

断言功能	参考节
显式返回成功或失败	Explicit Success and Failure
异常检查	Exception Assertions
Predicate Assertions	Predicate Assertions for Better Error Messages
类型检查	Type Assertions
Windows HRESULT assertions	Windows HRESULT assertions

gtest运行参数

使用gtest 编写的测试用例通常本身就是一个可执行文件，因此运行起来非常方便。同时，gtest也为我们提供了一系列的运行参数（环境变量、命令行参数或代码里指定），使得我们可以对用例的执行进行一些有效的控制。gtest提供了三种设置的途径：命令行参数、代码中指定FLAG、系统环境变量：

1、系统环境变量

系统环境变量全大写，比如对于--gtest_output，相应的系统环境变量为：GTEST_OUTPUT，有一个命令行参数例外，那就是--gtest_list_tests，它是不接受系统环境变量的（只是用来罗列测试用例名称）。

2、命令行参数

框架在main函数中已经将命令行参数带入gtest的宏中，这样框架就拥有了接收和响应gtest命令行参数的能力。

```
testing::InitGoogleTest(&argc, argv);
```

3、代码中指定FLAG

可以使用testing::GTEST_FLAG这个宏来设置。比如相对于命令行参数--gtest_output，可以使用testing::GTEST_FLAG(output) = “xml:”;来设置。注意，参数不需要加--gtest前缀。

注意：优先级是后设置的生效，一般规则是命令行参数 > 代码中指定FLAG > 系统环境变量。

目录

HUAWEI

什么是MOCK?

便捷的模拟对象的方法

- NiceMock：Uninteresting函数调用不会产生警告
- StrictMock: Uninteresting 函数调用会产生错误导致测试失败
- **NaggyMock**: Uninteresting函数调用只会产生警告，测试不会受影响

为什么要Mock测试?

1. 解决不同单元之间由于耦合而难于测试的问题
2. 通过模拟依赖以分解单元测试耦合的部分
3. 验证所调用的依赖的行为

HUAWEI

Mock对象适用的场景?

1. 独立被测单元和其依赖模块
2. 被测单元需要依赖尚未完成开发模块的返回值而进行后续处理
3. 被测单元依赖的对象难以模拟或者构造比较复杂

HUAWEI

Mock工具对比

工具类型	支持语言类别	支持操作系统	License	社区活跃度
gMock	C++	Windows/Linux	BSD	★★★★★
MockCpp	C/C++	Windows/Linux	Apache	★★★★☆
Opmock	C/C++	Windows/Linux	GPL	★★★☆☆
mockpp	C++	Linux	GPL	★★☆☆☆
FFF	C	Windows	MIT	★★★☆☆

- gMock在**开源选型**及**活跃度**上相对更为友好

HUAWEI

gMock支持的特性

1. 轻松创建mock类
2. 丰富的匹配器(Matcher)和行为(Action)
3. 有序、无序、部分有序的期望行为的定义
4. 支持多平台

HUAWEI

gMock典型应用流程

1. 引入用到的gMock名称
2. 建立模拟对象 — Mock Objects
3. 设置模拟对象默认动作
4. 在模拟对象上设置预期

HUAWEI

gMock典型应用流程：Mock类方法

```
class User {
public:
    User() {};
    ~User() {};
public:
    // 登录
    virtual bool Login(const std::string& username, const std::string& password) = 0;
    // 支付
    virtual bool Pay(int money) = 0;
    // 是否登录
    virtual bool Online() = 0;
};
```

```
class TestUser : public User {
public:
    MOCK_METHOD2(Login, bool(const std::string&, const std::string&));
    MOCK_METHOD1(Pay, bool(int));
    MOCK_METHOD0(Online, bool());
};
```

MOCK_METHODx(func, param)

- 用于模拟（mock）函数方法
- 仅限制于模拟类成员方法
- 宏接收两个参数
  func：函数名
  param：参数类型，返回值(形参)
- 函数为const类型则应使用MOCK_CONST_METHODx系列宏
- 使用该宏声明的类成员函数，可使用EXPECT_CALL等一系列宏指定函数在被调用时的Action

约束：

- 只能Mock类的虚函数
- 对静态函数支持不友好
- 在调用Mock函数前需设置好预期，否则其行为“未定义”

HUAWEI

gMock典型应用流程：设计测试场景

```
EXPECT_CALL(mock_object, Method(argument-matchers))
             .With(multi-argument-matchers)
             .Times(cardinality)
             .InSequence(sequences)
             .After(expectations)
             .WillOnce(action)
             .WillRepeatedly(action)
             .RetiresOnSaturation();
```

EXPECT_CALL声明一个调用期待，说明对象方法的执行逻辑
Method是mock对象中的mock方法，参数可通过matchers规则匹配

With指定多个参数的匹配方式

Times表示这个方法可以被执行的次数

InSequence用于指定函数执行的顺序

After方法用于指定某个方法只能在另一个方法之后执行

WillOnce表示执行一次方法时，将执行其参数action的方法

WillRepeatedly表示一直调用一个方法时，将执行其参数action的方法

RetiresOnSaturation用于保证期待调用不会被相同函数的期待所覆盖

HUAWEI

gMock典型应用流程：编写测试用例

```
TestUser testUser;
// 调用期待一
EXPECT_CALL(testUser, Online()).WillOnce(testing::Return(true));
// 调用期待二
EXPECT_CALL(testUser, Login("admin",_).WillRepeatedly(testing::Return(true));
// 调用期待三
EXPECT_CALL(testUser, Pay(_)).Times(5). WillOnce(testing::Return(true)).WillRepeatedly(testing::Return(false));
```

1. **调用期待一：**Online在调用一次后返回true，之后的调用返回默认的false
2. **调用期待二：**Login的第一个参数是admin，则总是返回true
3. **调用期待三：**Times(5)表示Pay函数期待被调用5次，后续调用返回默认值。WillOnce表征第一次调用返回true。WillRepeatedly表示之后的对Pay的调用都返回false。

gMock基础概念练习

给定如下类方法，应如何定义Mock接口?

virtual int OpenFile(const char* path, int openFlag) const = 0;

答案： MOCK_CONST_METHOD2(OpenFile, int(const char*, int);

virtual void CloseFile(const char* path, int closeFlag, int needForce) = 0;

答案： MOCK_METHOD3(CloseFile, void(const char*, int, int);

HUAWEI

gMock应用case初探

思考：该case使用到了何种gMock功能?

```
class MockCSubscriber: public CSubscriber
{
public:
    MockCSubscriber(int fd): CSubscriber(fd) {}
    MOCK_METHOD1(ReadBuf, int(int));
    MOCK_METHOD1(WriteBuf, int(int));
    MOCK_METHOD0(CloseSock, void());
};

TEST(SubPubHandler, sub_pub_read_and_write_buffer)
{
    int fd = 5;
    int readLen = 1000;
    int writeLen = 50;
    MockCSubscriber subObj(fd);
    ON_CALL(subObj, ReadBuf(readLen)).WillByDefault(Return(0));
    ON_CALL(subObj, WriteBuf(writeLen)).WillByDefault(Return(0));
    EXPECT_CALL(subObj, ReadBuf(readLen)).Times(1);
    EXPECT_CALL(subObj, WriteBuf(writeLen)).Times(1);
    EXPECT_CALL(subObj, CloseSock()).Times(1);
    CSubEventHandler subHandler(NULL);
    EXPECT_TRUE(subHandler.handleRead(&subObj));
}
```

Mock类CSubscriber方法

gMock应用方法:

- 通过构造函数生成Mock实例subObj
- 调用ON_CALL设置默认期望
- 调用EXPECT_CALL设置次数生成器

HUAWEI

gMock：匹配器

作用：对一个对象进行验证

匹配器可用在ON_CALL()及EXPECT_CALL()的**matcher**位置上，也可单独使用

- **EXPECT_THAT(value, matcher)**

用以判断value是否符合matcher定义的规则

- **ASSERT_THAT(value, matcher)**

同EXPECT_THAT，但是会产生一个最终错误导致测试失败退出

gMock：匹配器

类型	关键字	用法说明
通配匹配	_ A<type>() 或者An<type>()	• 类型匹配任意匹配 • 任何类型为type的参数都可以匹配
一般匹配	Ge(value)、Gt(value)、Le(value)、 Lt(value)、Ne(value)、IsNull() 、NotNull()	参数> =value、参数> value、参数<=value、参数<value、参数!=value、参数是个空指针、参数为非空指针
浮点数匹配	DoubleEq(a_double) FloatEq(a_float)	• 参数= a_double • 参数= a_float
字符串匹配	EndsWith(suffix) HasSubstr(string) StartWith(prefix) StrEq(string) StrNe(string)	• 参数以后缀suffix结尾 • 参数包含string子字符 • 参数以前缀prefix开始 • 参数与string相同，大小写敏感 • 参数与string不同，大小写敏感
容器匹配	ContainerEq(container) Contains(e) Each(e) SizeIs(m) IsEmpty	• 参数和container有同样内容 • 参数包含满足e的元素，e可以是值 • 参数中每个元素都满足e，e可以是值 • 参数的长度符合m的要求 • 参数为空容器
指针匹配	Pointee(m)	参数(指针)指向的内容符合m

HUAWEI

gMock：Action

gMock可选择使用系统或者用户自定义的行为

类型	关键字	用法说明
默认行为	DoDefault()	调用系统或者用户使用ON_CALL定义的默认行为
返回值	Return() Return(value) ReturnNull() ReturnPointee(ptr) ReturnRef(variable)	• 没有返回值的返回，适用于void函数 • 返回一个value值，必要时会做类型转换 • 返回一个空指针 • 返回指针ptr指向的值 • 返回variable的引用
副行为	Assign(&variable,value) Throw(exception)	• 将value赋给variable • 抛出异常exception
组合行为	IgnoreResult(a)	调用a,但是忽略它的结果，a必须有返回值

HUAWEI

gMock：次数生成器

次数生成器用于生成Times()中的次数参数

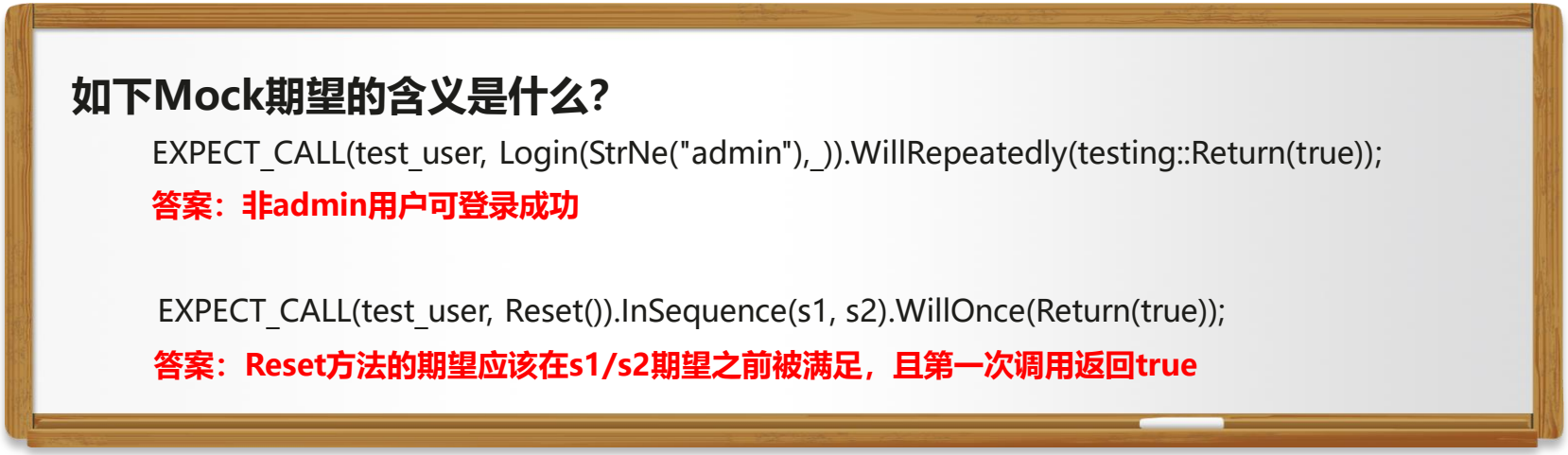
关键字	用法说明
AnyNumber()	任意次数
AtLeast(n)	期望最少N次
AtMost(n)	期望最多N次
Between(m,n)	期望介于m和n之间的次数，包括m和n次
Exactly(n)	期望n次

gMock：期望时序设置

Mock期望存在时序关系，可指定满足期望的次序，形成有时序的期望链条

类型	说明	例子
After从句	指定期望被调用后才允许调用其他期望	Expectation firstExp = EXPECT_CALL(car1,run(1)); Expectation secondExp = EXPECT_CALL(car1,run(2)); EXPECT_CALL(car1.run(5)).After(firstExp, secondExp);
InSequence对象	InSequence对象的作用域范围内的期望必须以定义的顺序被调用	Sequence s1, s2; MockFoo mockFoo; EXPECT_CALL(mockFoo, GetSize()).InSequence(s1, s2).WillOnce(Return(1)); EXPECT_CALL(mockFoo, GetValue()).InSequence(s1).WillOnce(Return(string("Hello World!")));
Sequence 对象	通过InSequence定义在同一个Sequence对象上的所有期望，必须以与定义时相同的顺序被调用。	

gMock进阶概念练习



如下Mock期望的含义是什么?

```
EXPECT_CALL(test_user, Login(StrNe("admin"),_)).WillRepeatedly(testing::Return(true));
```

答案：非admin用户可登录成功

```
EXPECT_CALL(test_user, Reset()).InSequence(s1, s2).WillOnce(Return(true));
```

答案：Reset方法的期望应该在s1/s2期望之前被满足，且第一次调用返回true

HUAWEI

目录

HUAWEI

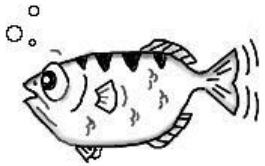
GDB简介

GDB是GNU开源组织发布的一个强大的UNIX下的程序调试工具。或许，各位比较喜欢那种图形界面方式的，像VC、BCB等IDE的调试，但如果你是在UNIX平台下做软件，你会发现GDB这个调试工具有比VC、BCB的图形化调试器更强大的功能。所谓“寸有所长，尺有所短”就是这个道理

一般来说，GDB主要帮助你完成下面四个方面的功能：

- ◆启动你的程序，可以按照你的自定义的要求随心所欲的运行程序。
- ◆可让被调试的程序在你所指定的调置的断点处停住。
- ◆当程序被停住时，可以检查此时你的程序中所发生的事。
- ◆动态的改变你程序的执行环境。

编译时增加-g选项，然后利用程序本身作为符号表调试，适用于联调阶段



HUAWEI

GDB调试的三种方式

方式一：启动程序并调试 #gdb [program]

- 这是使用GDB调试程序的最常用的方式。当使用GDB启动一个程序的时候，会自动默认第一个参数使用-se选项，第二个参数使用-p(attach pid的方式)或-c(读core文件的选项)选项。
- 在-se选项中-s表示-symbols，会从指定的文件中读取符号表，-e表示-exec，就是将指定的文件作为二进制来执行，这会在启动GDB之后再启动指定的二进制程序，然后直接关联GDB与相应程序，进入调试模式(可在top中看到为t状态，即trace stop)。

方式二：调试core文件 #gdb [program] [core]

- 对GDB来说，第二个参数是默认使用-p或者-c选项的，GDB会自动识别十进制数字为pid，尝试打开失败后才会尝试作为core来读取，所以如果有一个名字为数字的core文件，就要使用./1234来让GDB识别为core，或使用-c选项指定为core，而有时候可以不调用二进制文件直接调试core，也需要使用-c来指定后面的参数是core文件。
- 在这种模式下，GDB调试的其实是core文件，所以是可以直接看到core文件产生时的信息，如使用bt看到调用栈。

方式三：调试已在运行的文件（通过pid ）#gdb [program] [pid]

- 在产品环境下，很多时候当需要调试一个业务进程时，这个程序可能已经在运行了，这时候就需要使用GDB的attach功能。
- attach有三种方式
  - gdb [program] [pid]会自动识别第二个参数为pid(如果是十进制的数字的话)。
  - 运行gdb之后，在gdb中使用指令(gdb) attach [pid]。
  - 如果不使用源文件调试，可以直接#gdb –p [pid]来指定pid进入attach状态。
- 在attach状态下，被调试程序也会进入t状态，如果系统支持多线程调试，那么所有的子线程也会进入t状态(多线程调试后面会介绍)。

HUAWEI

GDB指令介绍

关键字	用法说明
set	用于设置gdb内部的一些环境与运行时的参数
print	用于打印的一个指令，可以搭配可选参数使用类似print/x(或p/x)的方式使用
show	描述的GDB本身的状态。你可以用set命令改变大多数你可以 用show显示的内容
info	可以描述程序的状态
help	一个查询指令，可以使用help来查看其他指令的用法，例如help print
backtrace	常直接使用缩写bt，可以查看调用栈，在调试core时一般首先就是使用bt查看调用栈信息
break	用于设置程序断点
list	可缩写为l，用于列出源码

HUAWEI

练习时间

GuessNumber

实现猜数字的游戏。游戏有四个格子，每个格子有一个0到9的数字，任意两个格子的数字都不一样。你有6次猜测的机会，如果猜对则获胜，否则失败。每次猜测时需依序输入4个数字，程序会根据猜测的情况给出xAxB的反馈，A前面的数字代表位置和数字都对的个数，B前面的数字代表数字对但是位置不对的个数。

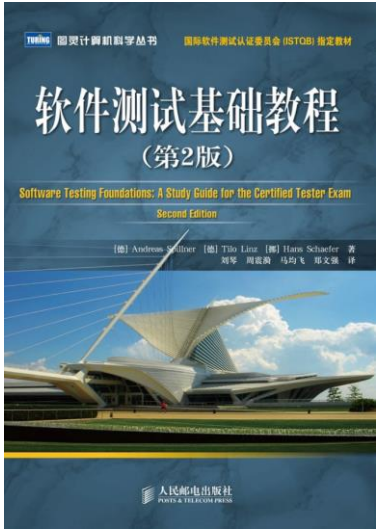
例如：答案是1 2 3 4， 那么对于不同的输入，有如下的输出

Input	Output	Instruction
1 5 6 7	1A0B	1 correct
2 4 7 8	0A2B	2 and 4 wrong position
0 3 2 4	1A2B	4 correct，2 and 3 wrong position
5 6 7 8	0A0B	all wrong
4 3 2 1	0A4B	4 numbers position wrong
1 2 3 4	4A0B	win, all correct
1 1 2 3	Wrong Input，Input again	
1 2	Wrong Input，Input again	

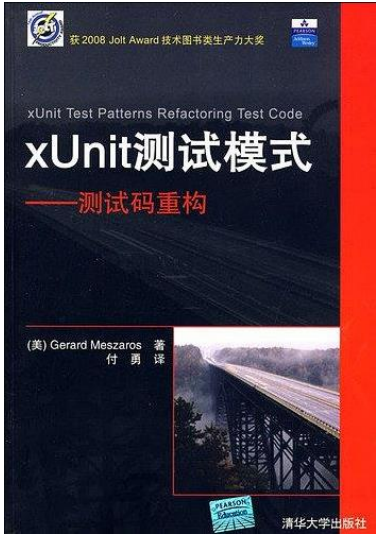
答案在游戏开始时随机生成。输入只有6次机会，在每次猜测时，程序应给出当前猜测的结果，以及之前所有猜测的数字和结果以供玩家参考。

HUAWEI

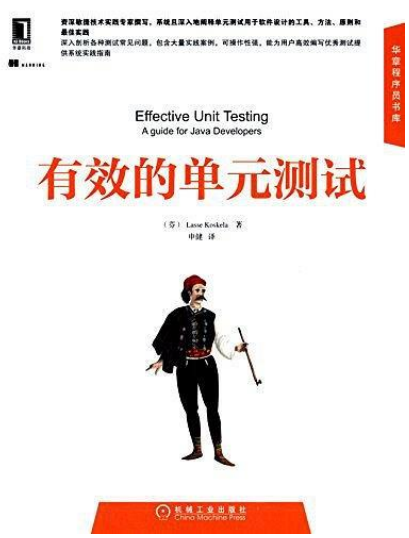
推荐学习材料



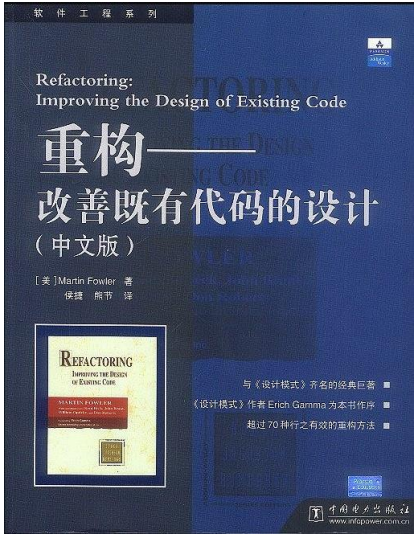
《软件测试基础教程》



《xUnit测试模式--测试码重构 》



《有效的单元测试》



《重构-改善既有代码的设计》

HUAWEI

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home and organization for a fully connected, intelligent world.

Copyright©2018 Huawei Technologies Co., Ltd.
All Rights Reserved.

The information in this document may contain predictive statements including, without limitation, statements regarding the future financial and operating results, future product portfolio, new technology, etc. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purpose only and constitutes neither an offer nor an acceptance. Huawei may change the information at any time without notice.

HUAWEI