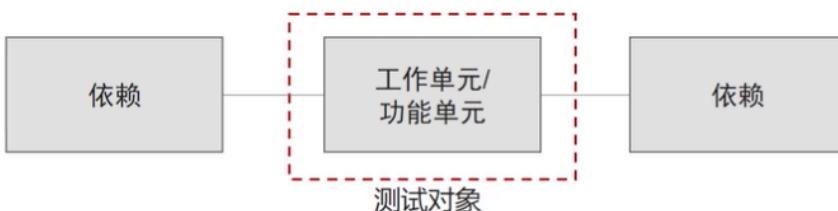


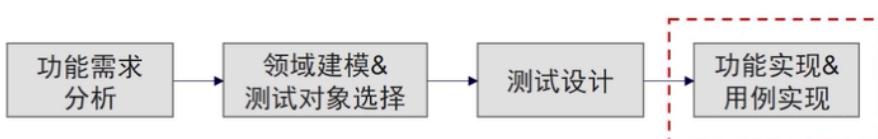
# 目录

|                   |
|-------------------|
| · 一：开发者测试基础       |
| · 单元测试&TDD        |
| · 软件测试术语          |
| · 开发者测试（DT）的定义和价值 |
| · 可测试性            |
| · 测试分层            |
| 二：测试设计方法          |
| · 测试设计概述          |
| · 常见的测试设计方法       |
| · 通过代码覆盖分析进行测试补充  |
| 三：C/C++测试实现与执行    |
| · 测试框架概述          |
| · gtest测试框架       |
| · gMock基础知识       |
| · GDB调试技能         |

# 单元测试 (UT, Unit Test)



- ✓ **单元**: 最小粒度的功能单元/工作单元。对于面向过程语言(如C语言), 功能单元是对外的功能接口; 对于面向对象的语言(如C++/Java) 功能单元是某个类或多个功能相关类的集合。功能单元通常在100~500行。
- ✓ **单元测试**: 一段自动化的代码, 先构造功能单元的输入, 然后调用功能单元, 最后对功能单元的输出进行校验。单元测试主要针对功能单元对外的接口进行测试, 功能单元内部调用的函数不需要测试, 因为功能单元对外的接口相对比较稳定, 这样测试用例才比较稳定。



**UTDD**: 先写一个测试用例运行失败, 再功能实现运行成功, 最后重构, 然后进入下一个迭代;  
**非UTDD**: 先完成功能实现, 再基于测试设计补充用例实现。

- ✓ 开展单元测试的工作流通常为: 基于功能需求进行领域建模, 并基于领域模型选择测试对象, 然后对测试对象进行测试设计生成测试用例, 最后再功能实现&用例实现。
- ✓ UTDD是UT实现的一个极限玩法, 不管是否使用TDD, 领域建模&测试对象选择、测试设计都是要做的。
- ✓ 单元测试的测试设计也要基于黑盒(基于功能需求设计, 而非代码内部结构)视角进行设计。

# 单元测试实例：1.功能需求分析

## ATM：

ATM控制器主要负责登录、退出登录、存款、取款、余额查询功能，交互信息显示通过ATM显示器，账户余额保存在数据库

### 存款功能：

- 1、登录（卡号、密码校验）后才能存款
- 2、存款金额必须是100的倍数，假设存款金额肯定不会负数
- 3、存款后余额=存款前余额+存款金额

### 取款功能：

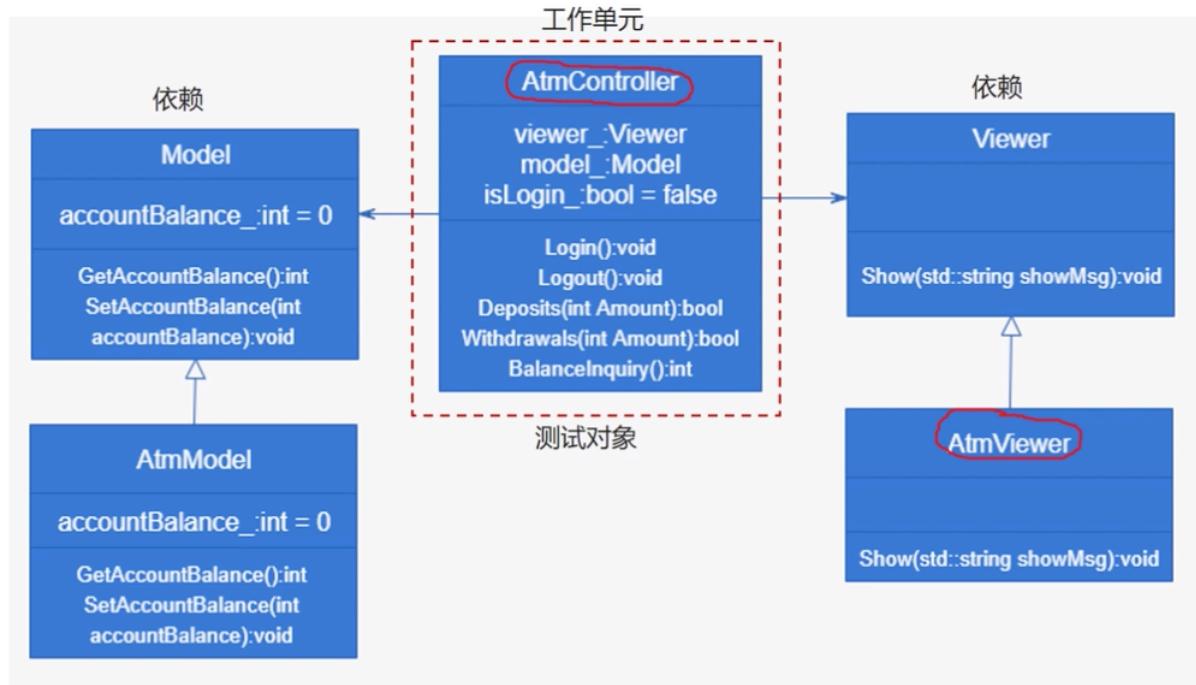
- 1、登录（卡号、密码校验）后才能取款
- 2、取款金额必须是100的倍数，假设取款金额肯定不会负数
- 3、取款金额必须 $\leq$ 余额
- 4、取款后余额=取款前余额-取款金额

### 余额查询功能：

- 1、登录（卡号、密码校验）后才能查询
- 2、返回并显示当前余额

## 单元测试&TDD

### 单元测试实例：2.领域建模&测试对象选择



## 单元测试实例：3. 测试设计



测试设计的目标：最少的测试用例，覆盖最多的测试场景

### 取款功能测试设计

#### 1. 识别因子

是否登录、是否是100整数、余额是否足够（余额、取款额）

#### 2. 确定因子取值

a. 是否登录，取值：是/否

b. 是否是100整数，取值：是/否

c. 余额是否足够（余额、取款额）：

假设“余额”是500，那么“取款额”可能是0~2147483647，不可能都测等价类：一个等价类中的多个取值相互等价（处理逻辑相同）

针对取款额，有效等价类：1~500，无效等价类：501~2147483647，根据等价类，只要选择2个取值即可，有效等价类：200，无效等价类：800

边界值：边界容易出问题，所以取款额的取值补充边界值499,500,501，但是取款额有100整数的限制，边界值保留500  
所以最终取款额的取值为200,500,800

#### 3. 确定因子组合

EC/BC/pair-wise/全组合等组合方法，也可以人工分析组合，见下表

#### 4. 确定预期结果

见下表

| 是否登录 | 是否是100整数 | 余额  | 取款额 | 测试用例描述                | 前置条件        | 操作步骤  | 预期结果       |
|------|----------|-----|-----|-----------------------|-------------|-------|------------|
| 否    | NA       | NA  | NA  | 用户不登录则不能取款            | 未登录         | 取款200 | 取款失败       |
| 是    | 否        | NA  | NA  | 取款额不是100倍数不能取款        | 成功登录，并存入500 | 取款150 | 取款失败，余额不变  |
| 是    | 是        | 500 | 200 | 成功登录，余额500，取款200，取款成功 | 成功登录，并存入500 | 取款200 | 取款成功，余额300 |
| 是    | 是        | 500 | 500 | 成功登录，余额500，取款500，取款成功 | 成功登录，并存入500 | 取款500 | 取款成功，余额为0  |
| 是    | 是        | 500 | 800 | 成功登录，余额500，取款800，取款失败 | 成功登录，并存入500 | 取款800 | 取款失败，余额不变  |

## 单元测试实例：4.功能实现（AtmController类定义）

```
class AtmController {
public:
    AtmController(Viewer &viewer, Model &model) : viewer_(viewer), model_(model), isLogin_(false)
    {};
    ~AtmController() {};

    // 登录
    void Login();

    // 退出登录
    void Logout();

    // 存款
    bool Deposits(int Amount);

    // 取款
    bool Withdrawals(int Amount);

    // 余额查询
    int BalanceInquiry();

private:
    Viewer &viewer_;
    Model &model_;
    bool isLogin_;
};
```

## 单元测试实例：4.功能实现（AtmController类实现）

```
// 登录
void AtmController::Login()
{
    isLogin_ = true; //此处简化实现
    return;
}
// 退出登录
void AtmController::Logout()
{
    isLogin_ = false; //此处简化实现
    return;
}
// 存款
bool AtmController::Deposits(int amount)
{
    if (!isLogin_) {
        viewer_.Show("Please Login first!!!!");
        return false;
    }
    model_.SetAccountBalance(model_.GetAccountBalance()+amount);
    return true;
}

// 取款
bool AtmController::Withdrawals(int amount)
{
    if (!isLogin_) {
        viewer_.Show("Please Login first!!!!");
        return false;
    }
    if (amount > model_.GetAccountBalance()) {
        viewer_.Show("Insufficient balance!!!!");
        return false;
    }
    if (amount % 100 != 0) {
        viewer_.Show("Please output a value that is a multiple of 100!!!!");
        return false;
    }
    model_.SetAccountBalance(model_.GetAccountBalance()-amount);
    return true;
}

// 余额查询
int AtmController::BalanceInquiry()
{
    viewer_.Show(std::string("balance: ") + std::to_string(model_.GetAccountBalance()));
    return model_.GetAccountBalance();
}
```

## 单元测试实例：4.功能实现（AtmViewer&AtmModel）

```
class Viewer {
public:
    Viewer();
    virtual ~Viewer();
    virtual void Show(std::string showMsg) = 0;
};

class AtmViewer : public Viewer {
public:
    AtmViewer();
    ~AtmViewer();
    void Show(std::string showMsg);
};

// 消息显示
void AtmViewer::Show(std::string showMsg)
{
    std::cout << "AtmViewer: " << showMsg << std::endl; //此处简化
    实现
}
```

```
class Model {
public:
    Model();
    virtual ~Model();
    virtual int GetAccountBalance() = 0;
    virtual void SetAccountBalance(int accountBalance) = 0;
};

class AtmModel : public Model {
public:
    AtmModel() : accountBalance_(0) {};
    ~AtmModel() {};
    int GetAccountBalance();
    void SetAccountBalance(int accountBalance);
private:
    int accountBalance_;
};

int AtmModel::GetAccountBalance()
{
    return accountBalance_; //此处简化实现
}
void AtmModel::SetAccountBalance(int accountBalance)
{
    accountBalance_ = accountBalance; //此处简化实现
}
```

## 单元测试实例：5. 测试实现（不使用测试框架）

```
void TestWithdrawalsFailWhenNotLogin()
{
    // Given: 成功登录，存入500元，然后退出登录
    AtmViewer atmViewer;
    AtmModel atmModel;
    AtmController atmController(atmViewer, atmModel);
    atmController.Login();
    atmController.Deposits(500);
    atmController.Logout();

    // When: 取款200元
    bool isWithdrawalsSucess = atmController.Withdrawals(200);

    // Then: 取款失败，余额500元
    if (isWithdrawalsSucess != false) {
        std::cout << "expect false ;" << "actual " << true << std::endl;
    }

    int balance = atmController.BalanceInquiry();
    if (balance != 500) {
        std::cout << "expect 500 ;" << "actual " << balance << std::endl;
    }
    return;
}

不用测试框架存在的问题：
1、只能运行所有测试用例，没法选择某个用例执行，除非重新编译；
2、结果检查繁琐，需要自己封装；
3、没有用例运行的统计信息；
4、同一种场景的测试的公共处理需要自己抽取并显式调用；
5、AtmViewer、AtmModel这2个依赖，需要自己模拟实现，才能隔离测试；.....
```

```
void TestWithdrawalsSucess()
{
    // Given: 成功登录，存入500元
    AtmViewer atmViewer;
    AtmModel atmModel;
    AtmController atmController(atmViewer, atmModel);
    atmController.Login();
    atmController.Deposits(500);

    // When: 取出200元
    atmController.Withdrawals(200);

    // Then: 余额300元
    int balance = atmController.BalanceInquiry();
    if (balance != 300) {
        std::cout << "expect 300 ;" << "actual " << balance << std::endl;
    }
    return;
}

int main()
{
    TestWithdrawalsFailWhenNotLogin();
    TestWithdrawalsSucess();
}
```

## 单元测试实例：5. 测试实现（使用测试框架gtest/gmock）

```
#include "gmock/gmock.h"
class MockViewer : public Viewer {
public:
    MOCK_METHOD1(Show, void(std::string));
};

void MockShow(std::string showMsg)
{
    std::cout << "MockViewer: " << showMsg << std::endl;
}

#include "gtest/gtest.h"
class TestAtm : public Test {
public:
    static void SetUpTestCase() {
        std::cout << "testsuite setup!!!" << std::endl;
    }

    static void TearDownTestCase() {
        std::cout << "testsuite teardown!!!" << std::endl;
    }

    void SetUp() {
        EXPECT_CALL(mockViewer, Show(_)).WillRepeatedly(Invoke(Mock
Show));
        atmModel.SetAccountBalance(0);
        std::cout << "testcase setup!!!" << std::endl;
    }

    void TearDown() {
        std::cout << "testcase teardown!!!" << std::endl;
    }
public:
    MockViewer mockViewer;
    AtmModel atmModel;
};

// ***** Test Case *****
// 测试用例描述: 未登录, 取款失败
// 预置条件: 成功登录, 存入500元, 退出登录
// 操作步骤: 取款200元
// 期望结果: 取款失败
// 修改历史: 1.2021-1-23 xxx
***** /
```

```
TEST_F(TestAtm, TestWithdrawalsFailWhenNotLogin) {
    // Given: 成功登录, 存入500元, 然后退出登录
    AtmController atmController(mockViewer, atmModel);
    atmController.Login();
    atmController.Deposits(500);
    atmController.Logout();

    // When: 取款200元
    bool isWithdrawalsSucess = atmController.Withdrawals(200);

    // Then: 取款失败, 余额500元
    EXPECT_EQ(isWithdrawalsSucess, false);
}
```

### 测试框架关键字简介：

- 1、**TEST\_F宏**: 每个宏定义一个测试用例，宏的两个参数分别代表测试套名和测试用例名，可随意定义
- 2、**SetUp和TearDown**: 在每个测试用例调用前和调用后执行
- 3、**SetUpTestCase和TearDownTestCase**: 在测试套调用前和调用后执行，必须为static void类型
- 4、**EXPECT\_EQ**: 预期结果检查
- 5、**MOCK\_METHOD1, EXPECT\_CALL**: 对一个类的接口进行打桩/模拟

## 单元测试实例：5. 测试实现（使用测试框架gtest/gmock）

```
*****
测试用例描述: 从ATM成功取出200元
前置条件: 成功登录，存入500元
操作步骤: 取款200元
期望结果: 取款成功，余额300元
修改历史: 1.2021-1-23 xxx
*****
TEST_F(TestAtm, TestWithdrawalsSucess) {
    // Given: 成功登录，存入500元
    AtmController atmController(mockViewer, atmModel);
    atmController.Login();
    atmController.Deposits(500);

    // When: 取款200元
    bool isWithdrawalsSucess = atmController.Withdrawals(200);

    // Then: 取款成功，余额300元
    EXPECT_EQ(isWithdrawalsSucess, true);
    int balance = atmController.BalanceInquiry();
    EXPECT_EQ(balance, 300);
}
```

一、环境准备  
1、linux环境  
2、gcc7.2  
3、cmake 3.14.1

二、编译工程  
1、切到ut-demo目录  
2、执行以下命令  
mkdir build  
cd build/  
cmake ../  
make

三、执行用例  
1、运行所有用例  
.tests/atmTest  
2、还可以执行用例执行  
.tests/atmTest --gtest\_filter=TestAtm.TestWithdrawalsSucess



```
[=====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 2 tests from TestAtm
testsuite setup!!!
[ RUN   ] TestAtm.TestWithdrawalsFailWhenNotLogin
testcase setup!!!
MockViewer: Please Login first!!!
MockViewer: balance: 500
testcase teardown!!!
[ OK   ] TestAtm.TestWithdrawalsFailWhenNotLogin (0 ms)
[ RUN   ] TestAtm.TestWithdrawalsSucess
testcase setup!!!
MockViewer: balance: 300
testcase teardown!!!
[ OK   ] TestAtm.TestWithdrawalsSucess (0 ms)
testsuite teardown!!!
[-----] 2 tests from TestAtm (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test case ran. (0 ms total)
[  PASSED ] 2 tests.
```

# 优秀单元测试的特点

## ➤ 单元测试的原则：

1、在《代码整洁之道》一书的单元测试章节中，Robert C.Martin提出了单元测试代码应该遵循的**FIRST原则**：

**F-FAST(快速原则)**：单元测试应该是可以快速运行的，在各种测试方法中，单元测试的运行速度是最快的，全量用例运行通常要在1分钟内；

**I-Independent(独立原则)**：单元测试应该是可以独立运行的，单元测试用例互相无强依赖，无对外部资源的强依赖；

**R-Repeatable(可重复原则)**：单元测试应该可以稳定重复的运行，并且每次运行的结果都是相同的；

**S-Self Validating(自我验证原则)**：单元测试应该是用例自动进行验证的，不能依赖人工验证；

**T-Timely(及时原则)**：单元测试必须及时的进行编写，更新和维护，以保证用例可以随着业务代码的变化动态的保障质量。

2、编写单元测试用例时，为了保证被测模块的交付质量，需要符合**BCDE原则**：

**B: Border**,边界值测试,包括循环边界、特殊取值、特殊时间点、数据顺序等；

**C: Correct**,正确的输入,并得到预期的结果；

**D: Design**,与设计文档相结合,来编写单元测试；

**E: Error**,单元测试的目标是证明程序有错,而不是程序无错。为了发现代码中潜在的错误,我们需要在编写测试用例时有一些强制的错误输入(如非法数据、异常流程、非业务允许输入等)来得到预期的错误结果。

## ➤ 优秀单元测试的特点：

1、用例头注释包含测试用例描述、前置条件、操作步骤、预期结果，提升可读性；

2、用例名称体现测试意图；

3、用例满足GWT结构（Given/When/Then），并使用 `// Given: xxx // When: xxx // Then: xxx` 的方式注释清楚

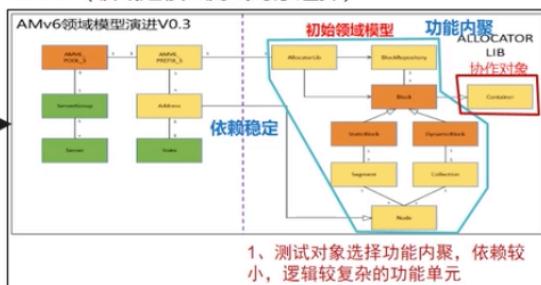
4、用例职责单一，一个用例只测一个场景，用例里禁止使用switch、if/else，测试代码是测试业务逻辑，而不应该去构造业务逻辑

5、尽量消除用例间的重复代码，适当的抽象封装，让单元测试用例尽量简洁，单元测试用例一般不超过30行；

# TDD

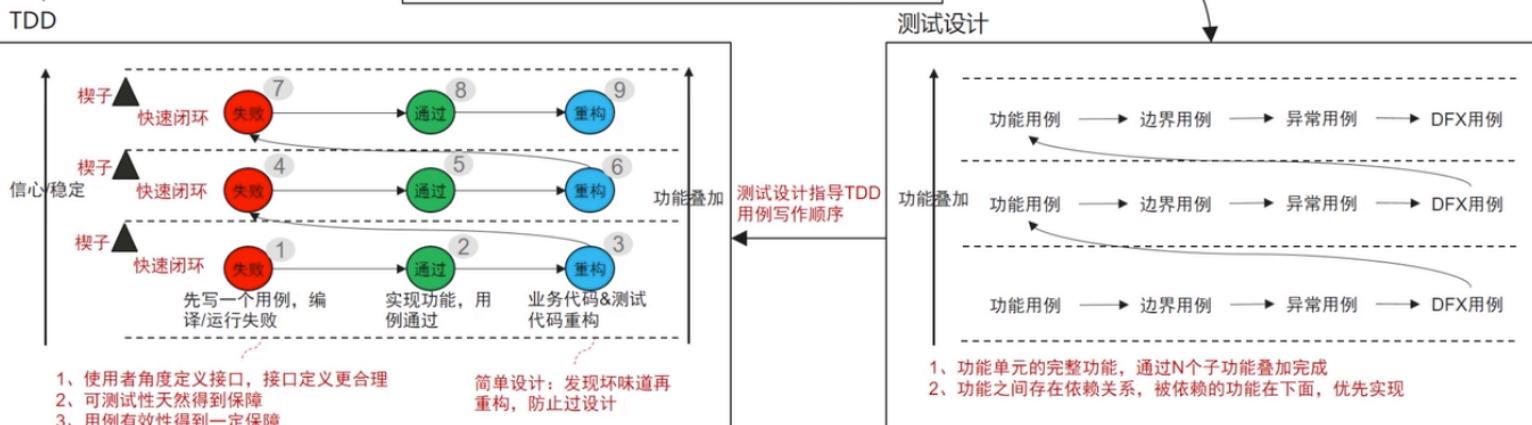
频繁的重构发掘出更多的协作对象，不断演进领域模型

DDD (领域建模&测试对象选择)



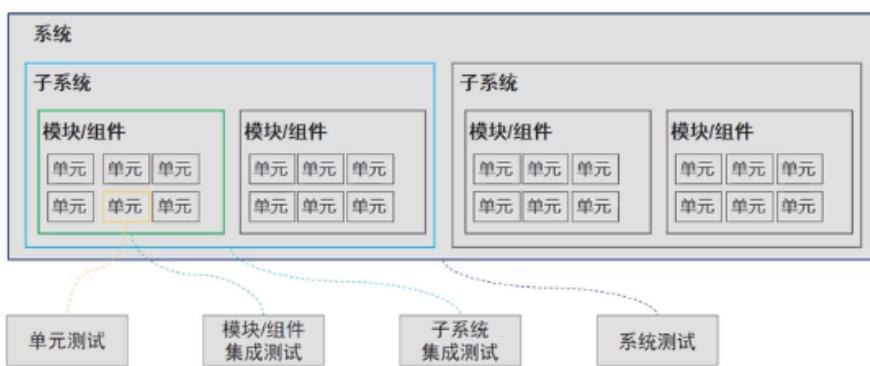
领域建模指导测试对象选择

测试设计



DDD指导TDD的测试对象选择，测试设计指导TDD用例写作顺序，TDD演进DDD的领域模型，使业务、架构、测试形成闭环

# 单元测试&集成测试&系统测试



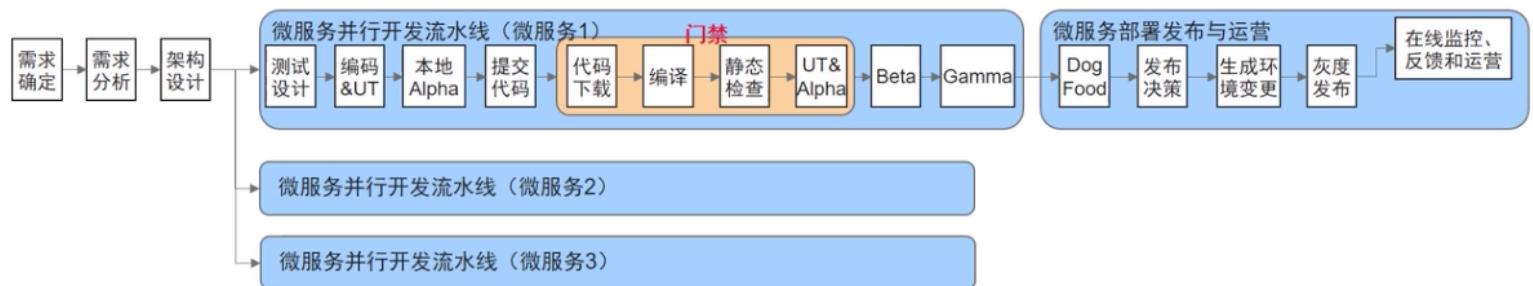
Integration Test (集成测试)，也叫组装测试或联合测试，按照设计要求将各个单元、组件或子系统集成到一起测试是否能正确运行；

System Test (系统测试)，是对整个系统的测试，将硬件、软件、操作人员看作一个整体，检验它是否有不符合系统说明书的地方。

## CMM流程（主要用于嵌入式）中测试相关术语

| 英文名                                        | 中文名              | 含义                                                                                                                                               |
|--------------------------------------------|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| Advanced Research                          | 预研               |                                                                                                                                                  |
| Offering Requirement Analysis              | 产品包需求分析          |                                                                                                                                                  |
| Design Requirement                         | 设计需求             |                                                                                                                                                  |
| Design Specification                       | 设计规格             |                                                                                                                                                  |
| Project-level Requirement Analysis         | 项目级需求分析          |                                                                                                                                                  |
| Project-level High-level Design            | 项目级概要设计          |                                                                                                                                                  |
| Project-level Low-level Design             | 项目级详细设计          |                                                                                                                                                  |
| Coding                                     | 编码               |                                                                                                                                                  |
| Project-level Unit Test (UT)               | 项目级单元测试 (UT)     | 对最小粒度的工作单元/功能单元进行验证的测试。                                                                                                                          |
| Project-level Integration Test (IT)        | 项目级集成测试 (IT)     | 组件/模块内部集成测试，即组件/模块对外接口测试。                                                                                                                        |
| Project-level System Test (ST)             | 项目级系统测试 (ST)     | 通常指子系统测试，也叫Module System Test (MST, 模块系统测试)，如果不与硬件强相关，可在PC上测试，所以也叫PCST。                                                                          |
| Building Block Integration and Test (BBIT) | 构件模块集成与测试 (BBIT) | 模块（子系统）间接口测试，验证模块之间的接口能不能配合。                                                                                                                     |
| System Design Verification (SDV)           | 系统设计验证(SDV)      | 验证各子系统的配合是否满足设计需求 (DR)，对内部的实现还是关注的，偏灰盒。                                                                                                          |
| System Integration Test (SIT)              | 系统集成测试(SIT)      | SIT也是验证设计需求是否得以满足，与SDV不同的是，SIT完全把系统当作一个黑盒来测试，不关心内部具体的实现。主要关注业务性能、话务量等测试。                                                                         |
| System Verification Test (SVT)             | 系统验证测试(SVT)      | SVT是验收测试，其测试对象是产品包需求OR。产品包需求给出了产品的范围，从产品可能的应用环境的角度刻画系统，SVT的目的就是确认（或验收）产品包需求给出的各种应用场景产品均能满足。产品包需求不考虑内部实现的差异，SVT也是从整个系统的角度考虑包需求的各种应用场景，属于“系统级”的测试。 |
| Beta Test                                  | 试验局 (Beta) 测试    |                                                                                                                                                  |
| Launch                                     | 发布               |                                                                                                                                                  |
| Life Cycle (Including ESP Maintenance)     | 生命周期（含ESP维护）     |                                                                                                                                                  |

## DevOps模式中测试相关术语



| 英文名            | 中文名  | 含义                         |
|----------------|------|----------------------------|
| Unit Test (UT) | 单元测试 | 对最小粒度的工作单元/功能单元进行验证的测试。    |
| Alpha          |      | 服务API功能测试、服务API性能测试。       |
| Beta           |      | 服务API集成测试，性能、安全、可靠性测试。     |
| Gamma          |      | 用户场景测试、探索测试、压力测试。          |
| Product        |      | 服务可用性测试、灰度测试、A/B测试、在线压力测试。 |

# 不同视角的软件测试分类

✓ 从是否执行程序的角度来看，软件测试可以划分为：

**静态测试：**测试时不执行被测试软件

**动态测试：**测试时执行被测试软件

✓ 从是否关心软件内部结构和具体实现的角度来看，软件测试可以划分为：

**白盒测试：**需要了解内部结构和代码

**黑盒测试：**不关心内部结构和代码

**灰盒测试：**介于白盒测试和黑盒测试之间

✓ 按软件开发过程的阶段划分，软件测试可以划分为：

**单元测试：**针对工作单元/功能单元的测试（工作单元/功能单元指某个功能、某个/几个类等）

**集成测试：**将各个单元、组件或子系统集成到一起测试是否能正确运行

**系统测试：**测试软件是否符合系统中的各项需求

**验收测试：**类似系统测试，但由用户执行

✓ 按测试的具体目的进行划分，软件测试可以划分为：

**功能测试：**软件测试是否符合功能性需求，通常采用黑盒测试方法

**性能测试：**测试软件在各种状态下的性能，找出性能瓶颈

**安全测试：**测试该软件防止非法入侵的能力

**回归测试：**在软件被修改或运行环境发生变化后进行重新测试

**兼容性测试：**测试该软件与其他软件、硬件的兼容能力

**安装测试：**测试软件的安装、卸载、升级是否正常

✓ 根据测试手段，软件测试可以分为：

**手动测试：**测试过程中需要人工干预和判断

**自动化测试：**测试过程中基本无需人工干预

✓ 根据测试自由度，软件测试可以分为：

**预定义测试和探索性测试。**

## 不同级别测试比较

| 测试手段        | 依赖 | 用例开发效率 | 用例执行效率 | 问题定位效率 | 有效的测试覆盖率 | 对设计的反馈 |
|-------------|----|--------|--------|--------|----------|--------|
| UT          | 低  | 低      | 高      | 高      | 高        | 高      |
| IT (组件级)    | 低  | 中      | 高      | 中      | 中        | 高      |
| PCST (子系统级) | 中  | 中      | 中      | 中      | 中        | 中      |
| 仿真 (多个子系统)  | 高  | 中      | 低      | 低      | 低        | 低      |
| 真实环境 (全系统)  | 高  | 中      | 低      | 低      | 低        | 低      |

- ✓ 被测对象越小，执行效率、问题定位效率、有效测试覆盖率越高；
- ✓ 被测对象越小，对设计的反馈越高（被测对象小需要隔离测试，耦合/依赖越大，越难隔离）

## 开发者测试（DT）的定义和价值

### 开发者测试（DT）定义



- ✓ 开发者测试（Developer Testing, 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级测试（UT、IT、PCST），少量的仿真测试和真实环境测试。
- ✓ 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IT、PCST、少量真实环境测试，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。

## 开发者测试（DT）的定义和价值

### 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。

解决问题的人力投入(人天)



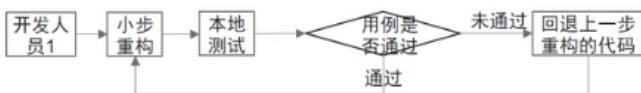
#### ➤ 持续集成（CI）门禁

每次提交代码上库，都必须跑一次持续集成（使用编译+运行较快的UT/IT/PCST作为CI门禁，5分钟内），否则就算开发人员本地测试过，上库后还可能存在逻辑冲突的问题（比如开发人员1修改了函数A的前半部分，开发人员2修改了函数A的后半部分，可以合并成功，但是功能逻辑可能不符合预期了）。



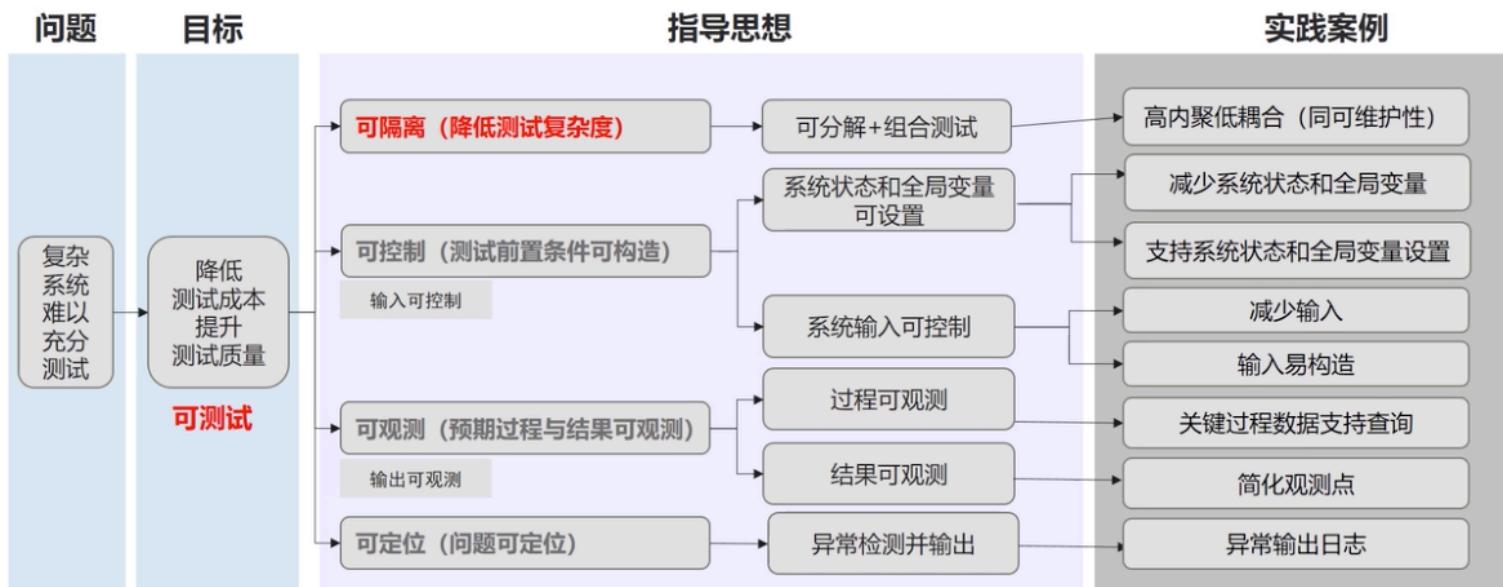
#### ➤ 重构防护网

重构需要小步快跑，快速反馈，使用编译+运行较快的UT/IT/PCST作为重构的快速反馈，UT1分钟内，IT3分钟内，PCST 5分钟内。



# 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。



# 什么是可隔离性，如何做到可隔离性

可隔离性指软件元素（功能单元、模块/组件等）被分解/隔离的能力，也有称可分解性

```
void ErrorStatProcess(.....)
{
    //加载路径归属项目组及Owner配置信息
    LoadOwnerConfigFile(.....);

    //根据错误所属路径与配置文件匹配错误归属项目组、责任人
    ErrorMatch(.....);

    //根据项目组进行分组统计
    ErrorStatByGroup(.....);

    //连接mongo db
    mongoClient = MongoClient(host, ip);
    db = mongoClient.dbName

    .....
    //向mongo db写入统计数据
    db[docName].insert_one(reportInfo);
}
```



```
int WriteToDb(.....)
{
    //连接mongo db
    mongoClient = MongoClient(host, ip);
    db = mongoClient.dbName

    .....
    //向mongo db写入统计数据
    db[docName].insert_one(reportInfo);
}

Void ErrorStatProcess(.....)
{
    //加载路径归属项目组及Owner配置信息
    LoadOwnerConfigFile(.....);

    //根据错误所属路径与配置文件匹配错误归属项目组、责任人
    ErrorMatch(.....);

    ErrorStatByGroup(.....); //根据项目组进行分组统计

    WriteToDb(.....); //将统计数据写入数据库
}
```

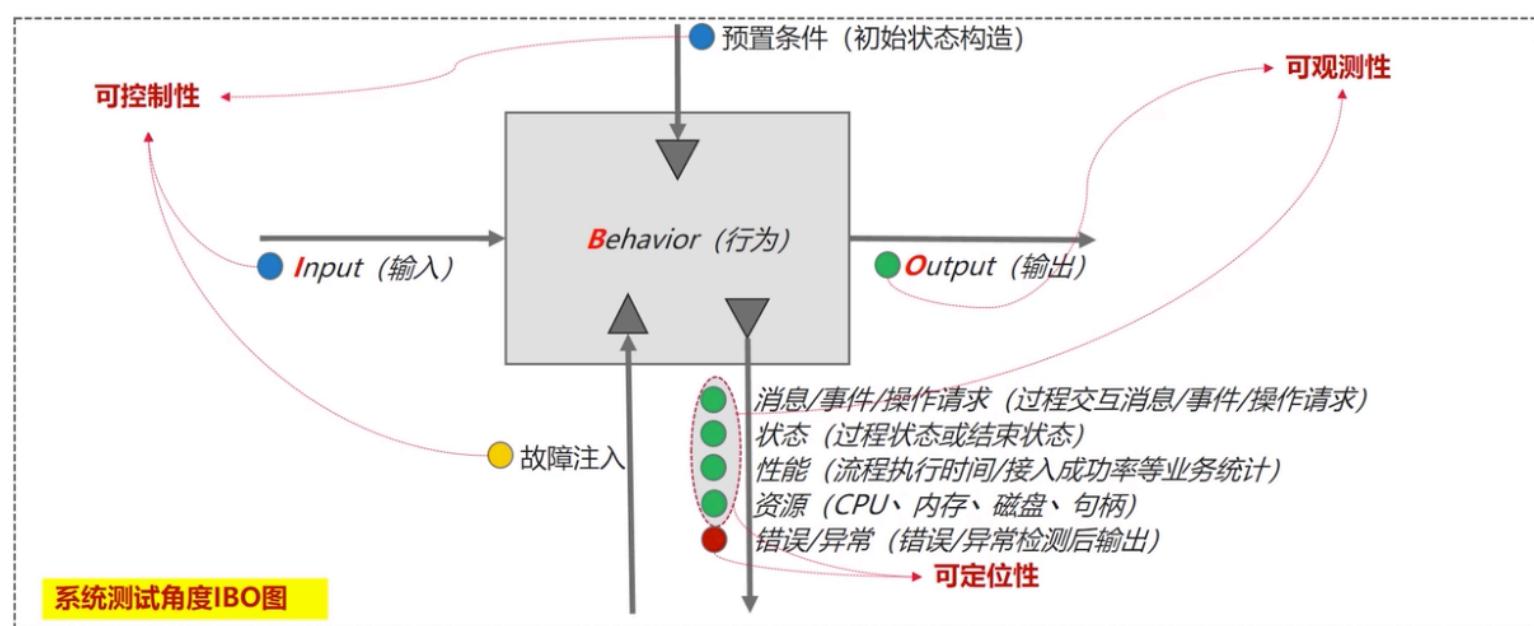
通过接口隔离

如果没有真实的DB，这个函数无法测试

将写DB功能抽象成接口，即可将WriteToDb打桩即可测试

- ✓ 可隔离性要求子系统/模块/组件/功能单元的边界清晰，设计目标同可维护性中的高内聚低耦合/SOLID原则
- ✓ 可测试性中隔离是指隔离测试，可维护性里的隔离是指隔离变化，但本质都是追求高内聚低耦合

## 从IBO模型导出可控制性、可观测性、可定位性的定义

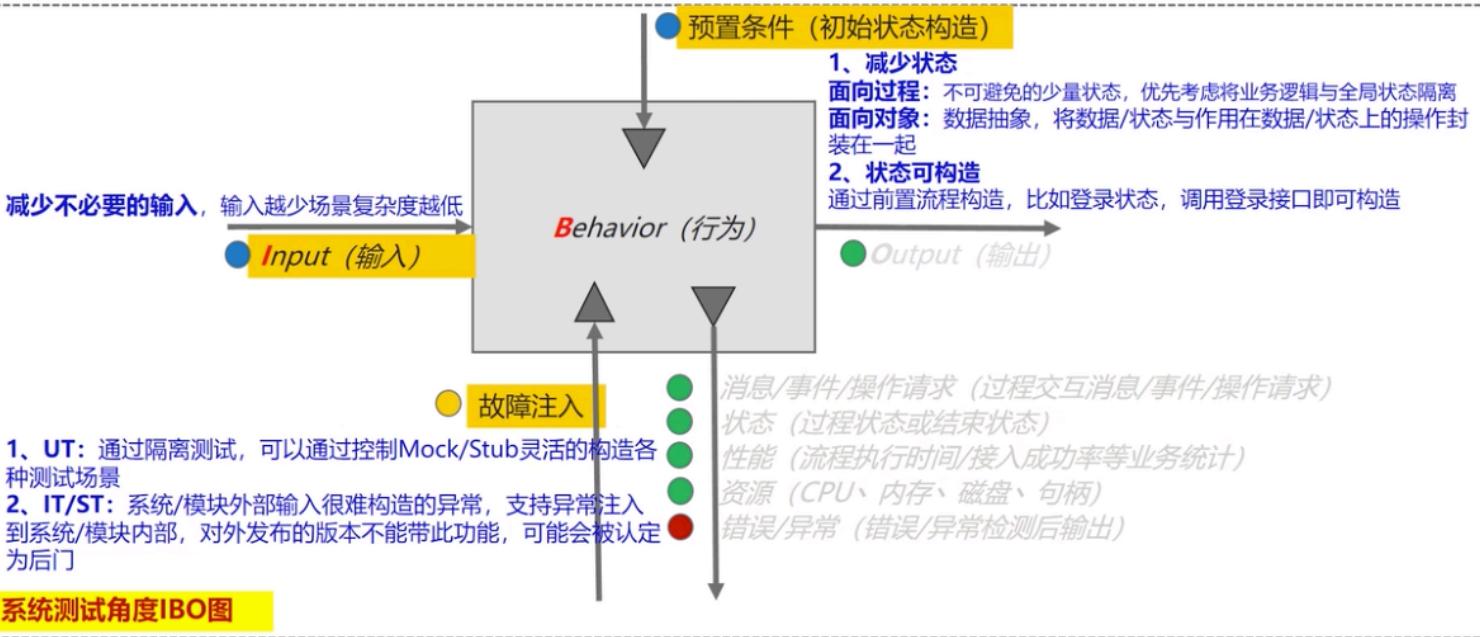


✓ 可控制性：对广义输入的控制能力；可观测性：对广义输出的观察能力；可定位性：输出信息支撑问题分析的能力

✓ 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。

# 提升可控制性



# 可控制性 (UT案例)

## 1、使用全局状态

```
int Withdrawals(unsigned int amount) {  
    if (!g_isLogin) { .....  
    }  
}
```

不推荐，全局状态不易跟踪、查找、构造



## 2、面向过程：业务逻辑与全局状态隔离

```
int Withdrawals(unsigned int amount, bool isLogin)  
{  
    if (!isLogin) { .....  
    }  
}
```

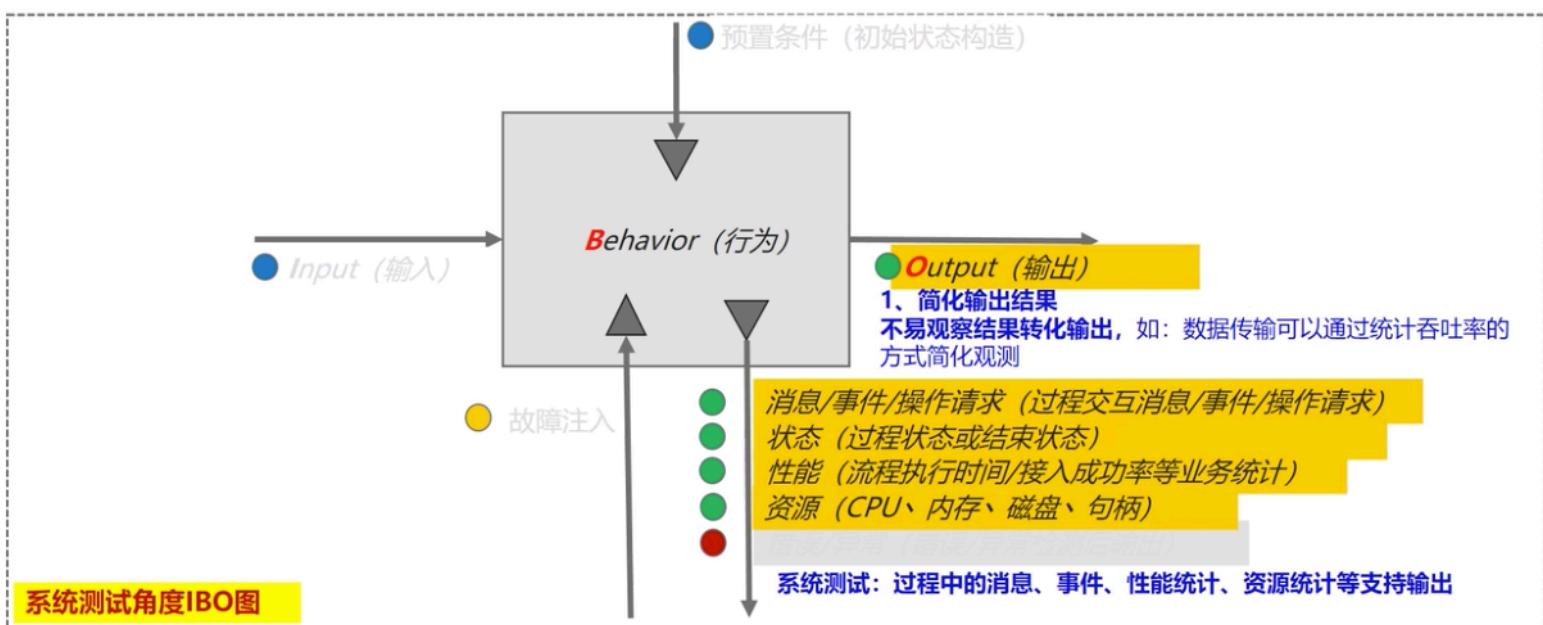
推荐，全局状态隔离后，输入明确，易使用、查找

## 3、面向对象：将状态与作用在状态上的操作封装在一起

```
class Atm {  
public:  
    int Login(.....){  
        if(...) {  
            is_login = true;  
        }  
    }  
    int Withdrawals(unsigned int amount) {  
        if (!isLogin_) { .....  
        }  
    }  
private:  
    bool isLogin_;
```

推荐，面向对象，状态封装在对象内部，通过Login接口构造状态

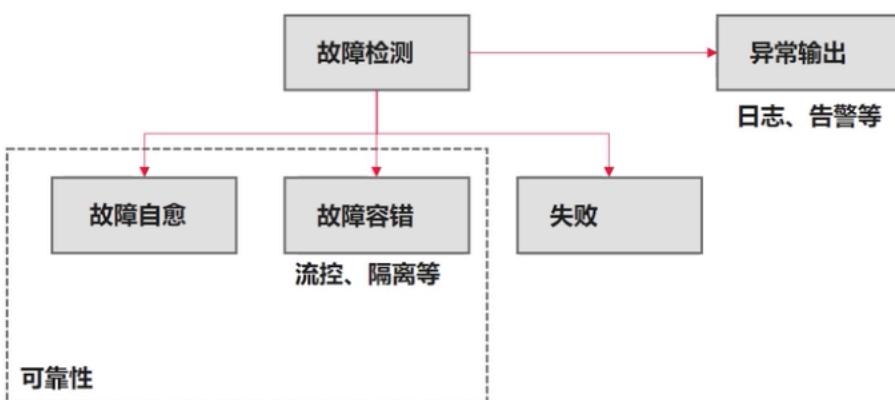
## 提升可观测性



可控制性=对广义输出的观察能力=预期输出校验的难易程度

- ✓ 可观测性，通常针对系统/子系统/模块等测试对象较大场景，单元测试一般观察函数输出参数即可
- ✓ 自动化测试尽量减少实现过程观测，因为实现过程可能会发生变化，一旦过程改变，对应的自动化用例就会失败

# 提升可定位性



- ✓ 不管故障被自愈、容错，还是最终失败，异常都需要输出，不能被内部逻辑完全消化
- ✓ 日志要分层（INFO/WARN/ERROR），日志量与可定位性要权衡，太少不足以定位问题，太多影响性能，浪费磁盘空间
- ✓ 有时过程中的交互消息/事件、状态、性能、资源等过程信息输出对问题定位也有非常关键的作用

## 可定位性（案例）

- ✓ 异常返回没有任何异常输出，很难支撑问题定位

```
void Func1(.....){  
    ...  
    rsIt = Proc1();  
    if(rsIt != OK) {  
        return; //异常返回，不能确定从哪里返回的  
    }  
  
    rsIt = Proc2();  
    if(rsIt != OK) {  
        return; //异常返回，不能确定从哪里返回的  
    }  
}
```



```
void Func1(.....){  
    ...  
    rsIt = Proc1();  
    if(rsIt != OK) {  
        LOG(ERROR, "error msg detail...");  
        return;  
    }  
  
    rsIt = Proc2();  
    if(rsIt != OK) {  
        LOG(ERROR, "error msg detail...");  
        return;  
    }  
}
```

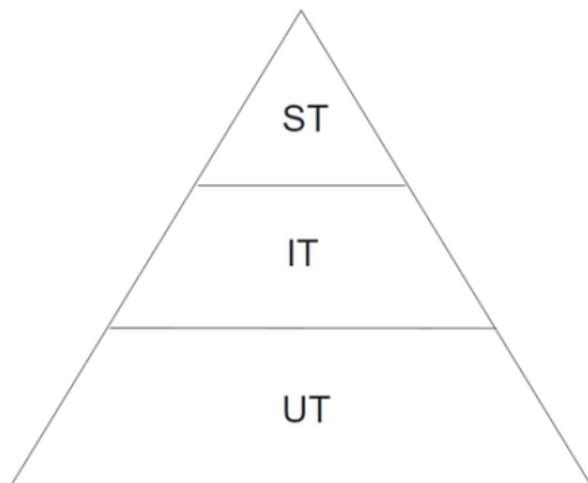
- ✓ 异常被兼容吞掉了，系统外面感知不到异常存在，但是可能不符合业务逻辑

```
void Func2(.....){  
    if(tti != TTI_2MS && tti != TTI_10_MS) {  
        tti = TTI_10_MS; //异常被兼容处理掉了  
    }  
}
```



```
void Func2(.....){  
    if(tti != TTI_2MS && tti != TTI_10_MS) {  
        LOG(ERROR, "error msg detail...");  
        tti = TTI_10_MS;  
    }  
}
```

# 测试分层通用模型



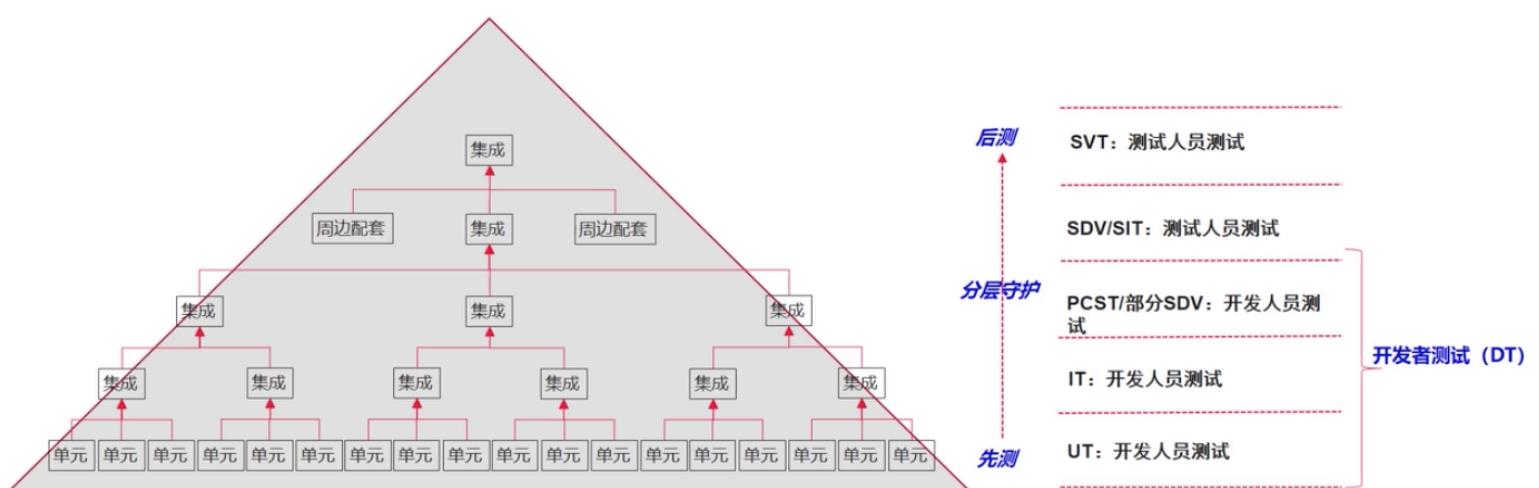
- ✓ 这里给出的是抽象的测试分层，具体测试分层怎么分、分几层与产品的规模、形态有关  
比如：对于规模非常小的系统，分1~2层就够了，对于规模很大的系统，可能需要分5~6层，甚至更多。
- ✓ 业界一般推荐 $UT:IT:ST=7:2:1$

## 测试分层实例



- ✓ 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

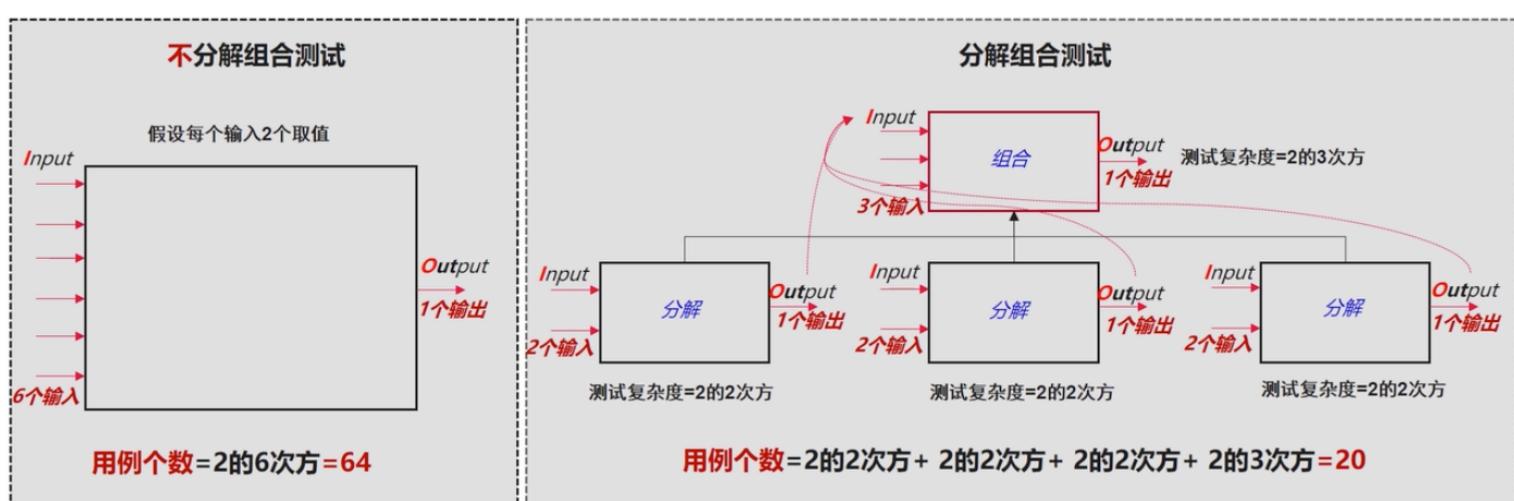
# 为什么要测试分层



- ✓ 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
- ✓ 分层守护尽量将问题在前端解决，问题越遗留到后端解决成本越高

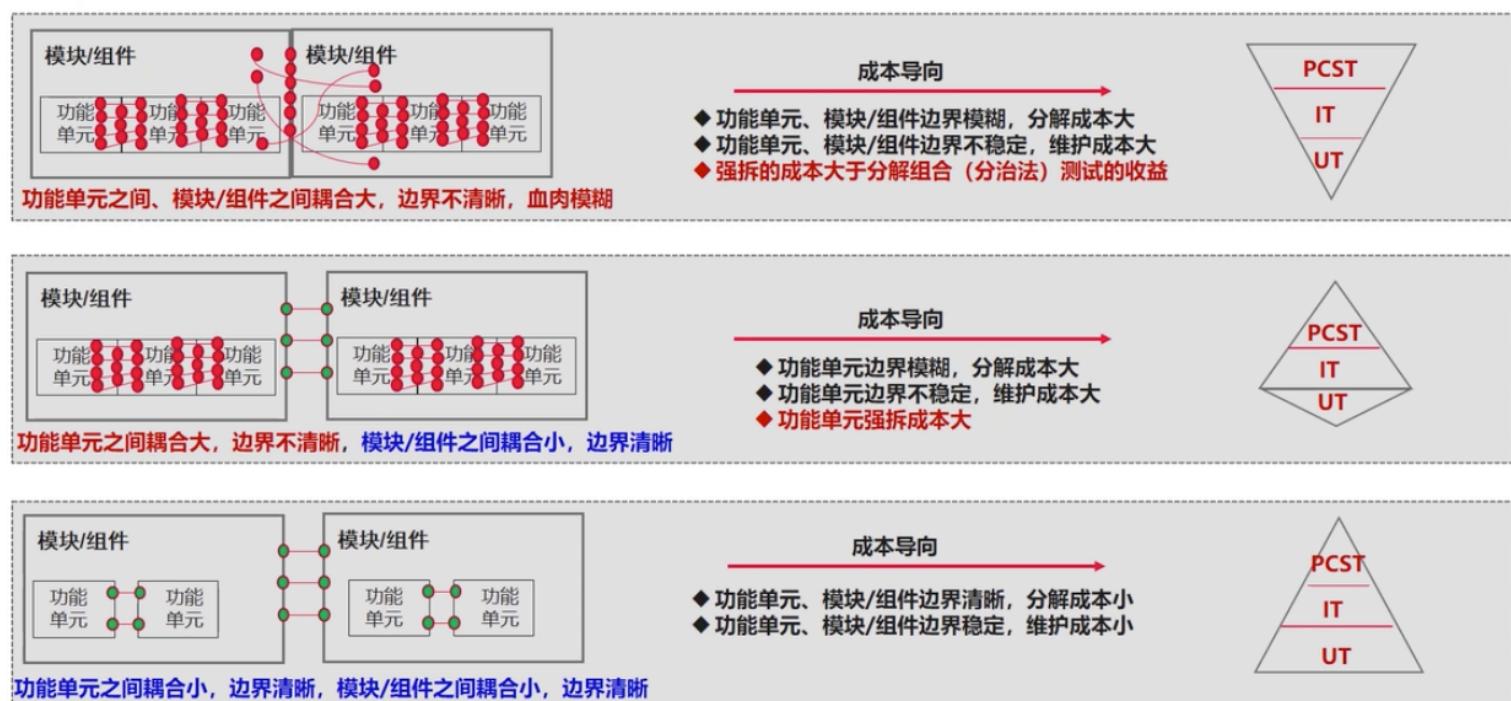
理想的测试分层呈金字塔状（业界测试金字塔UT:IT:ST为7: 2: 1），与分治法的树形对应

## 分解组合测试降低测试复杂度的证明



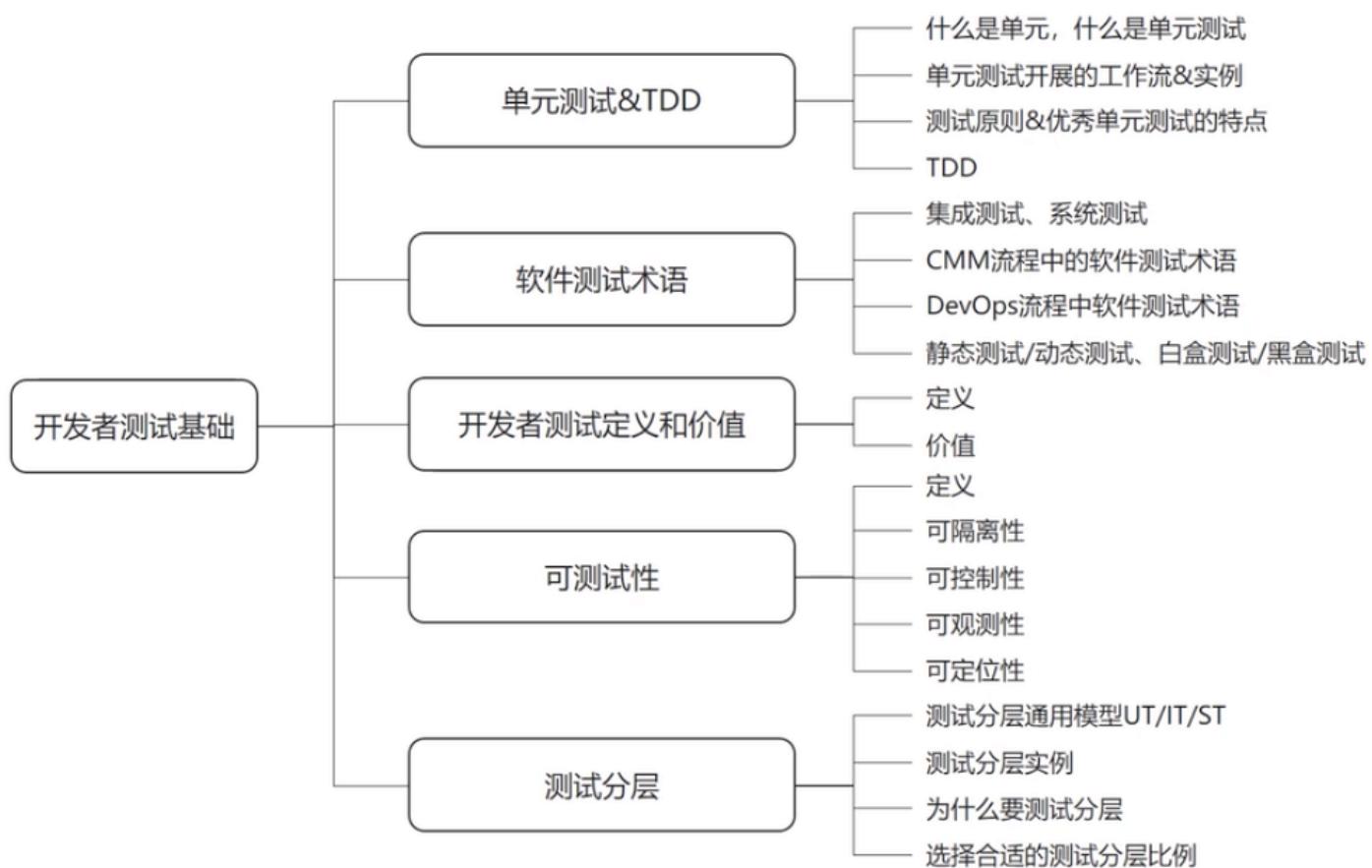
- ✓ 分解组合测试，利用分治法，降低测试复杂度，类似算法思想里面的分治法 ( $n^{\text{方}} \rightarrow n * \log n$ )
- ✓ 分解组合测试与系统架构分解组合对应

# 选择合适的测试分层比例



- ✓ **耦合/依赖比较大，不要强行隔离测试，强行隔离成本可能会很大；**
- ✓ **不要以可测试性不好作为不做DT的理由，基于当前现状选择合适的测试分层，先把DT做起来，然后通过架构优化，降低功能单元间、组件级的耦合/依赖，提升可隔离性逐步达成金字塔分层；**

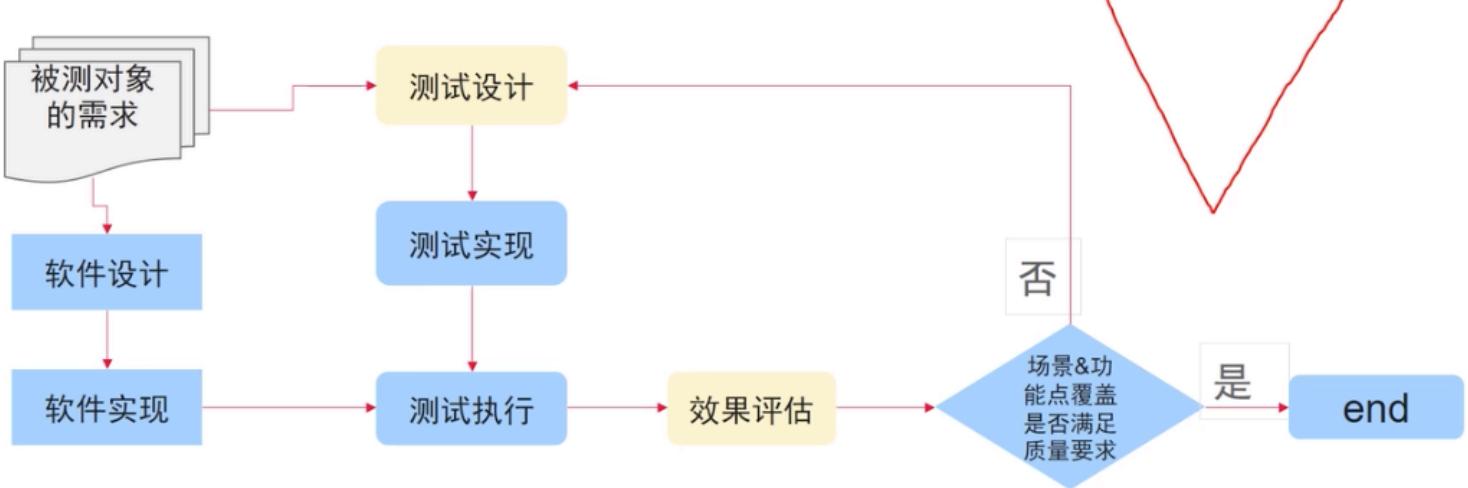
# 开发者测试基础部分主要内容回顾



# 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**。

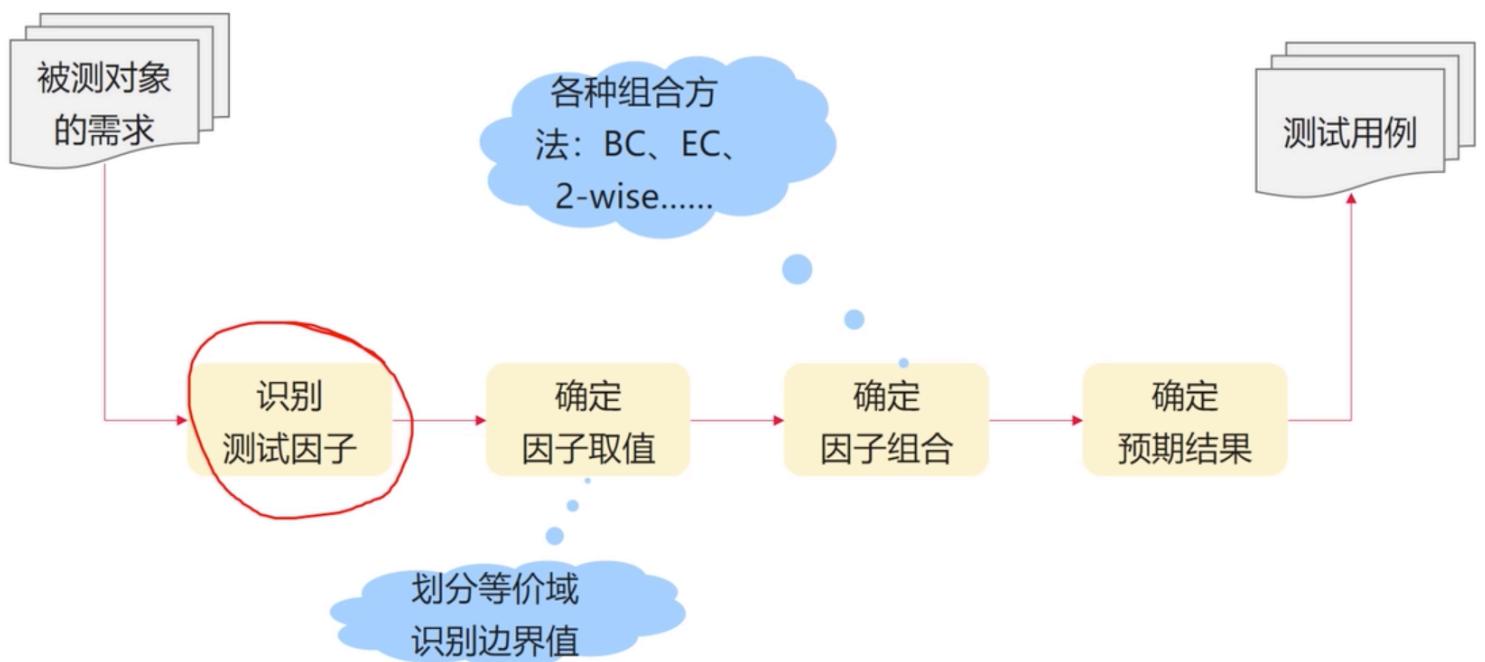
## 开发者测试设计所处的位置和大致过程



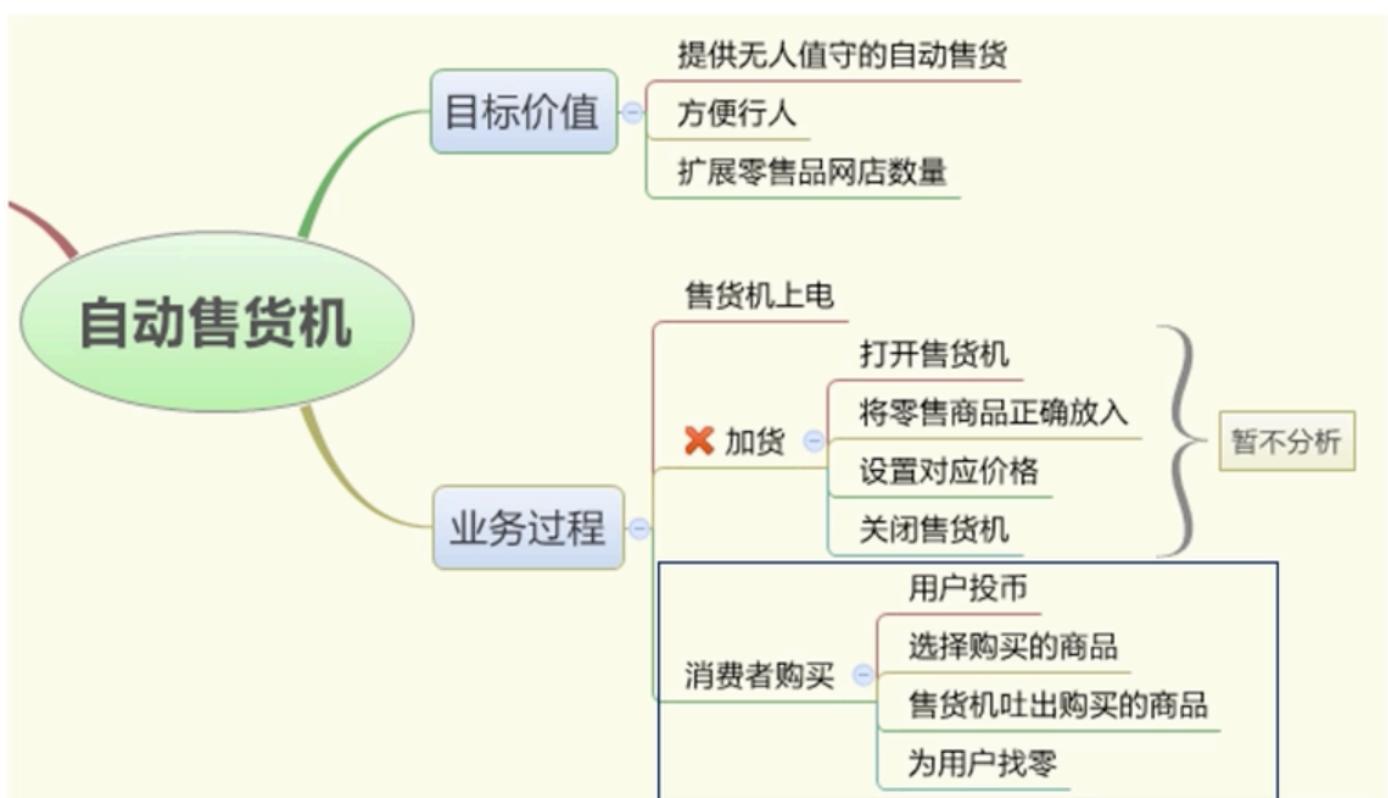
测试设计可看做一个**两阶段循环**的过程：

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。

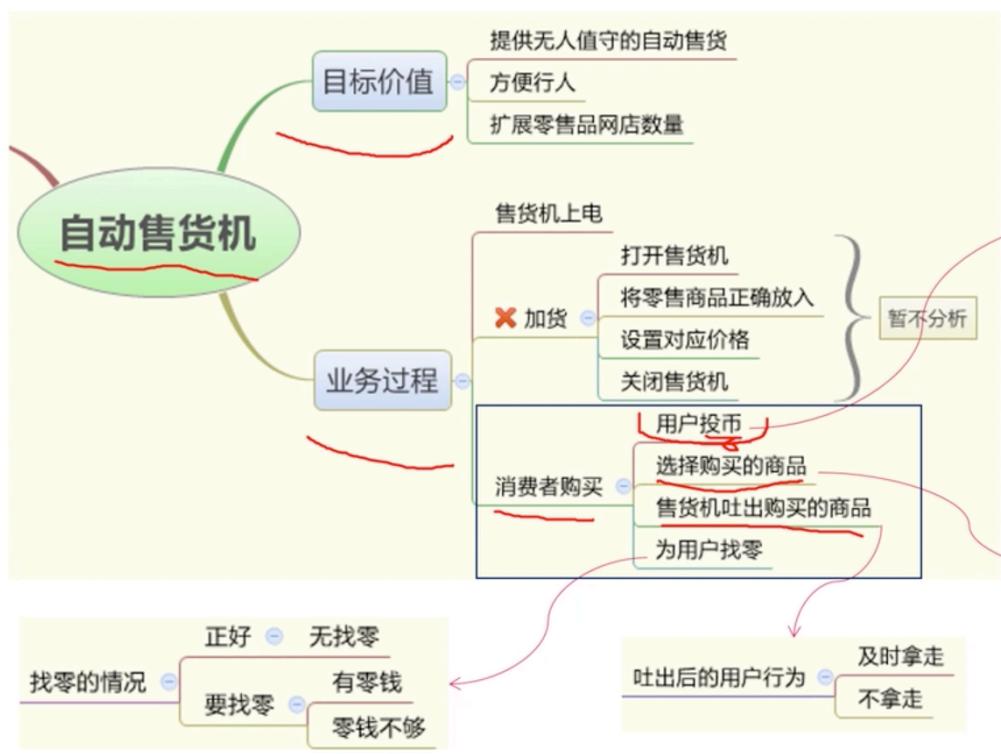
## 测试设计的一般步骤



# 通过思维导图打开操作步骤识别测试因子



## 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



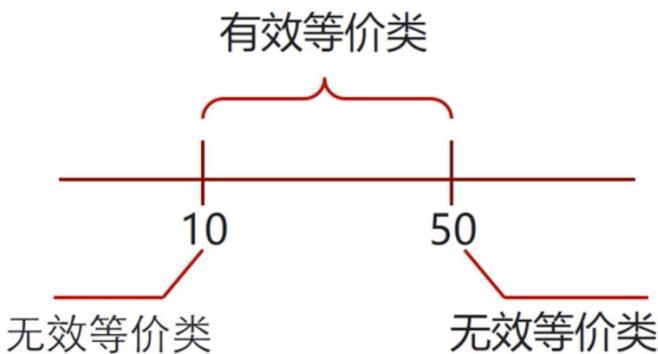
## 基本思路总结：

- 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

## 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。



#### 有效等价类

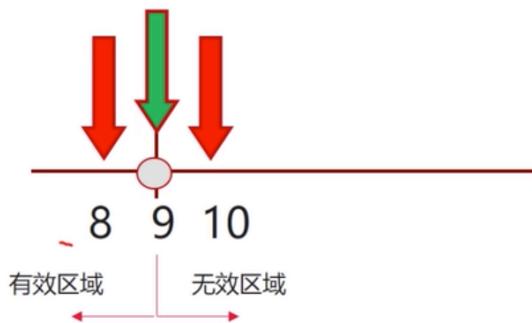
对于程序的规格而言是有意义、合理的数据输入的集合，用于测试程序是否实现了规格说明中约定的功能和性能要求。

#### 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。

一个大于等于10，小于等于50的因子的等价类

## 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

# 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



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 |

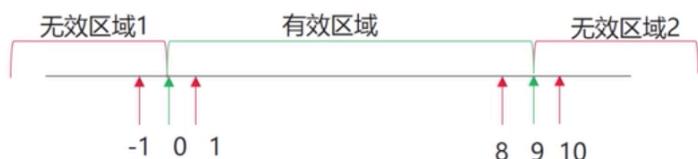


基于边界值分析对取值进行补充：

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



| 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

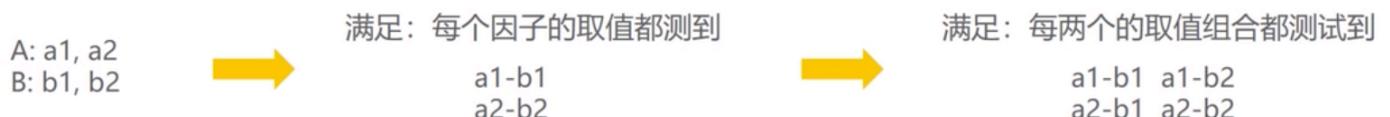
请使用之前学习过的等价类划分 & 边界值分析方法，为下面例子输出等价类列表：

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

# 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

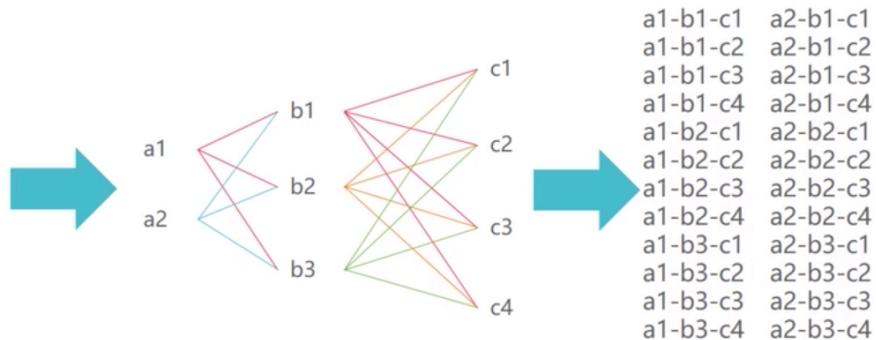


- 本节主要介绍4种常用的组合技术：
  1. AC (**All Combinations**)
  2. EC ( Each Choice )
  3. BC (Basic Choice)
  4. N-wise (主要介绍pair-wise)
- 选择哪种组合技术依赖于被测对象的风险（对于开发者测试，可以同时考虑实现成本和执行成本）

## 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<br>编号 | 取值<br>个数 | 取<br>值<br>1 | 取<br>值<br>2 | 取<br>值<br>3 | 取<br>值<br>4 |
|----------|----------|-------------|-------------|-------------|-------------|
| A        | 2        | a1          | a2          |             |             |
| B        | 2        | b1          | b2          | b3          |             |
| C        | 3        | c1          | c2          | c3          | c4          |



组合个数 =  $2 \times 3 \times 4 = 24$   
(每个因子取值个数的乘积)

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

## 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。

Step1：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



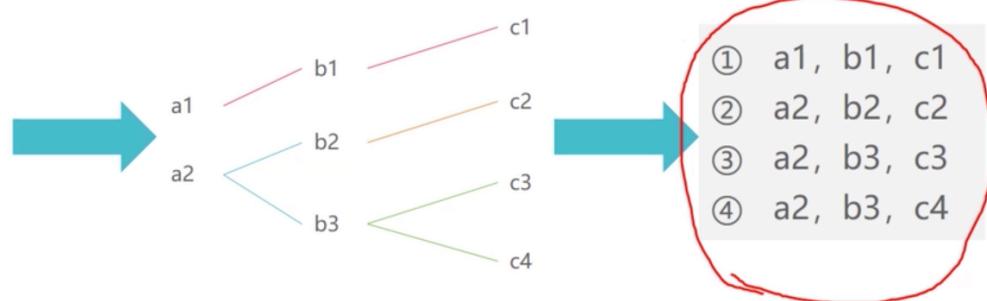
- ① a1, b1, c1
- ② a2, b1, c1
- ③ a1, b2, c1
- ④ a1, b3, c1
- ⑤ a1, b1, c2
- ⑥ a1, b1, c3
- ⑦ a1, b1, c4

$$\text{组合个数} = \text{所有因子取值个数} - \text{因子个数} + 1$$

## 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  |

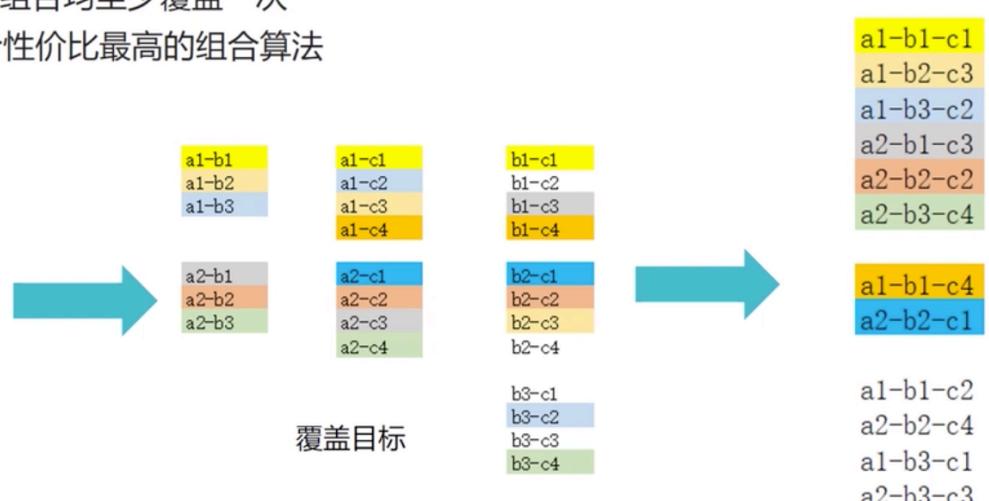


组合个数 = 所有因子中取值最多的那个因子的取值个数

## 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  |



组合个数 = 最多因子取值个数 × 次多因子取值个数

## 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<br>编号 | 取值<br>个数 | 取值<br>1 | 取值<br>2 | 取值<br>3 | 取值<br>4 |
|----------|----------|---------|---------|---------|---------|
| A        | 2        | a1      | a2      |         |         |
| B        | 3        | b1      | b2      | b3      | ✓       |
| C        | 4        | c1      | c2      | c3      | c4      |



✓

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

组合个数 = 最多因子取值个数 × 次多因子取值个数

## 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<br>编号 | 取值<br>个数 | 取值<br>1 | 取值<br>2 | 取值<br>3 | 取值<br>4 |
|----------|----------|---------|---------|---------|---------|
| A        | 2        | a1      | a2      |         |         |
| B        | 2        | b1      | b2      | b3      |         |
| C        | 3        | c1      | c2      | c3      | c4      |

AC: 24个用例

Pair-wise: 12个用例

BC: 7个用例

EC: 4个用例

# 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，实际中有很多用例设计辅助工具可供选用

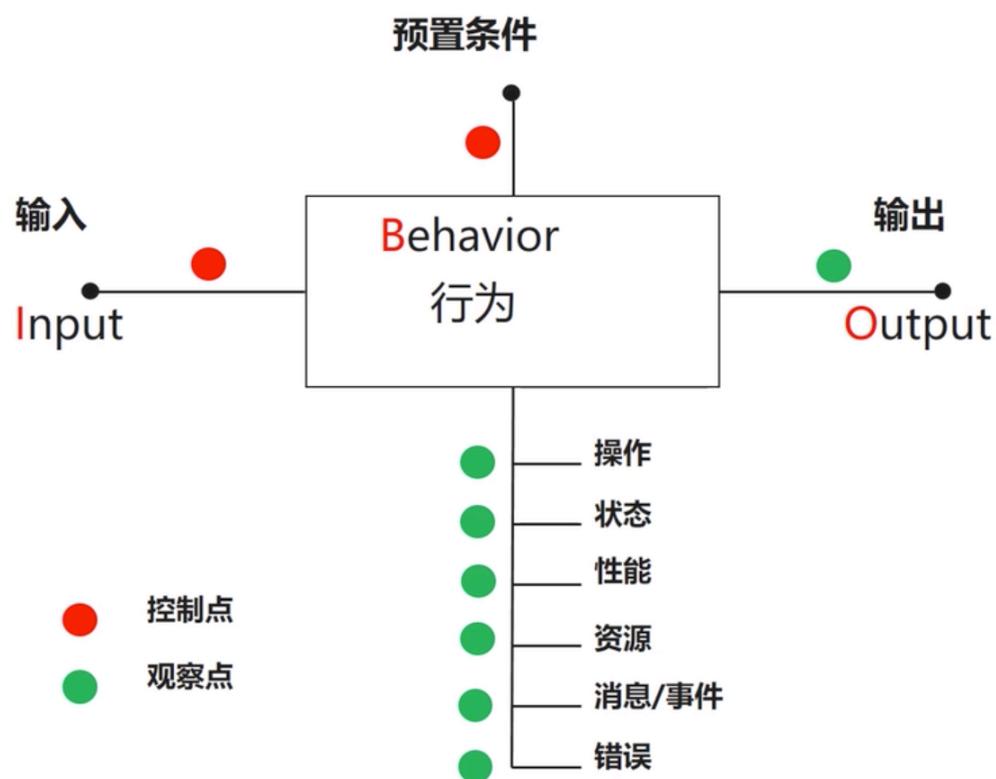


## 因子组合技术（总结 & 复习）

- 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。

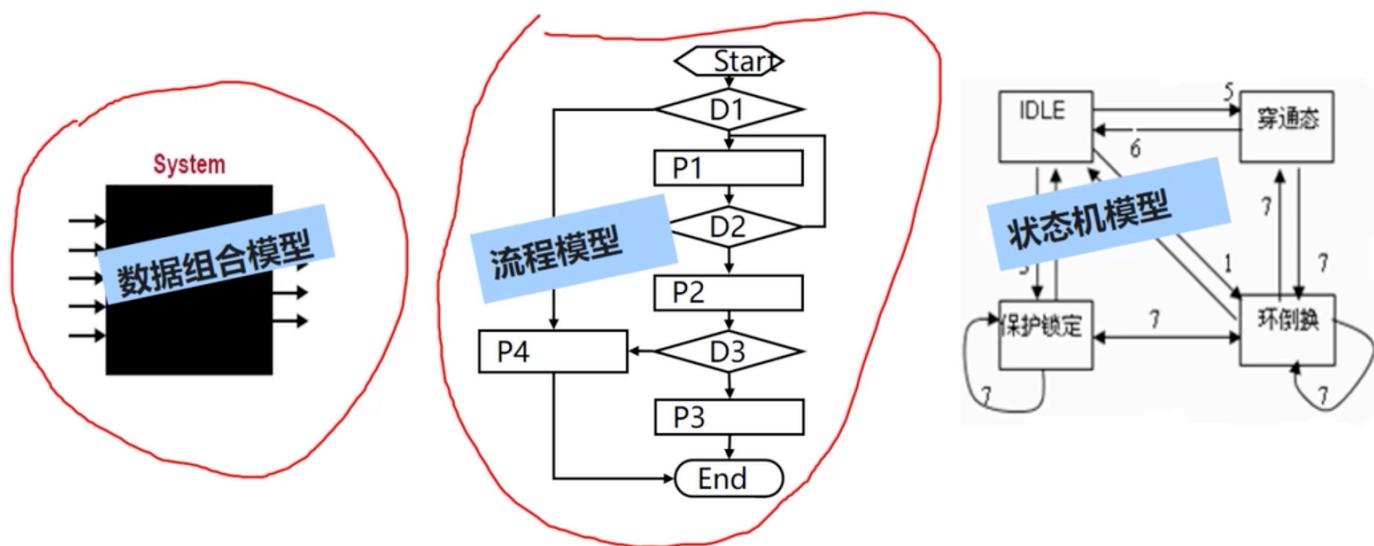
| 因子 | 取值1 | 取值2 | 取值3 |
|----|-----|-----|-----|
| A  | a1  | a2  | a3  |
| B  | b1  | b2  | b3  |
| C  | c1  | c2  | c3  |
| D  | d1  | d2  | d3  |

## 预期结果不仅仅是返回值，还需要考虑内部状态和数据变化



## 测试设计技术补充说明

- 等价类划分，边界值分析以及本节介绍的因子组合技术，主要针对的是可以表达为多个因子组合的被测对象，一般称为数据组合模型
- 但实际工作中，对象的行为往往更复杂，有些可能适合用流程模型表达，有些适合用状态机模型表达。针对不同的对象模型，还有一些更针对性的测试设计方法。



# 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)

# 语句覆盖

定义：程序中的每条可执行语句都能被执行到

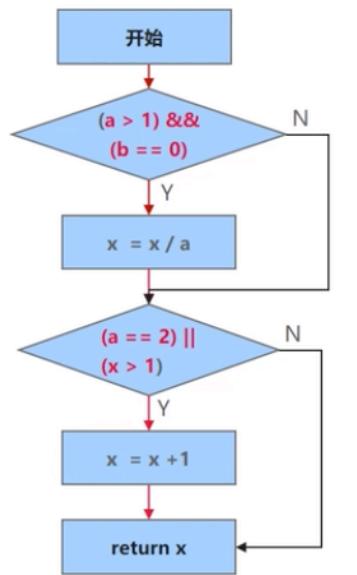
被测代码

```
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;  
-:  
-:  
1: 10:   if((a == 2) || ( x > 1)){  
1: 11:       x = x + 1;  
-:  
1: 12:   }  
1: 13:   return x;  
-:  
1: 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。

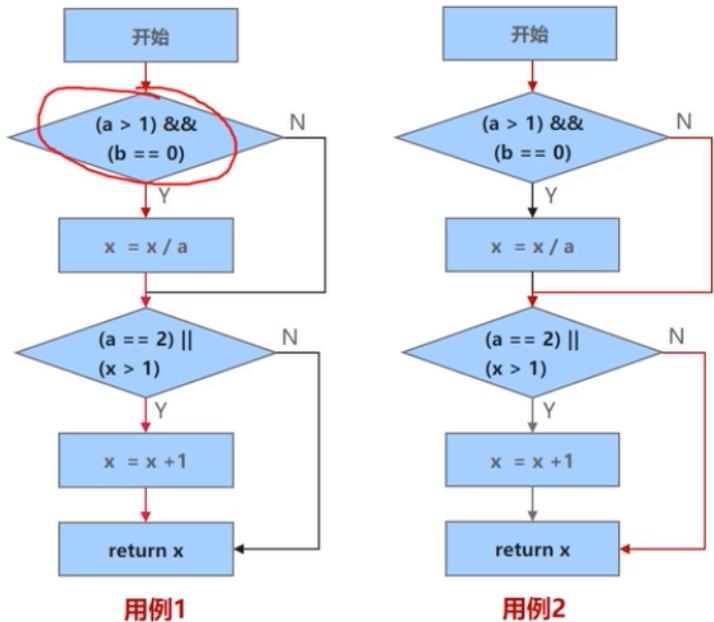
比如，本页的示例中，我们将`(a > 1) && (b == 0)`修改为`(a > 1) || (b == 0)`，用例依然会执行成功！

# 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：<br>$(a > 1) \&\& (b == 0)$ | 判定2：<br>$(a == 2) \parallel (x > 1)$ |
|-----------------|---------------------------------|--------------------------------------|
| {a=2, b=0, x=2} | T                               | T                                    |
| {a=1, b=1, x=1} | F                               | F                                    |

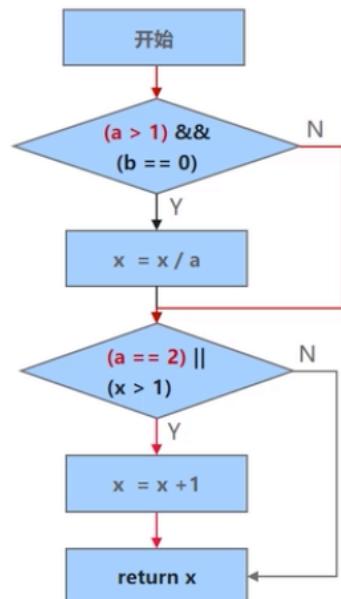
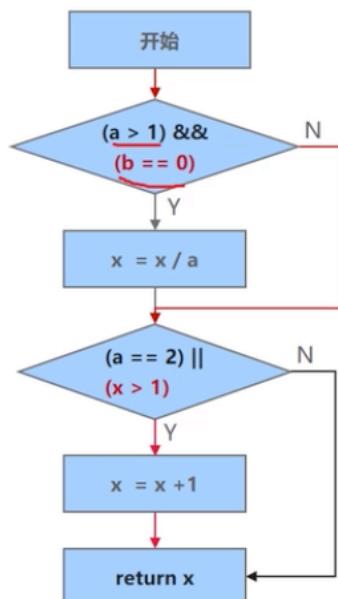
- 和语句覆盖类似，100%判定覆盖也不能说明代码没问题，也不能看护所有的修改。  
比如，我们将二个判断语句修改成 $(a != 2) \parallel (x > 1)$ ，当前的两个用例仍然可以执行成功。
- 满足判定覆盖的一定可以满足语句覆盖。



# 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,<br>b=0,<br>x=3} | {a=2,<br>b=1,<br>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:<br>$(a > 1) \&\& (b == 0)$ | F                     | F                     |
| 判定2:<br>$(a == 2)    (x > 1)$   | T                     | T                     |



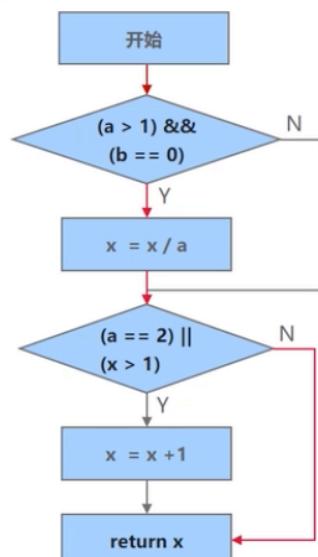
## 条件组合覆盖

定义：每个判定的条件取值组合至少能被覆盖一次

|                                |               | {a=2,<br>b=0,<br>x=2} | {a=2,<br>b=1,<br>x=1} | {a=1,<br>b=0,<br>x=2} | {a=1,<br>b=1,<br>x=1} |
|--------------------------------|---------------|-----------------------|-----------------------|-----------------------|-----------------------|
| 判定1：<br>$(a > 1) \&& (b == 0)$ | 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：<br>$(a == 2)    (x > 1)$  | 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                     |

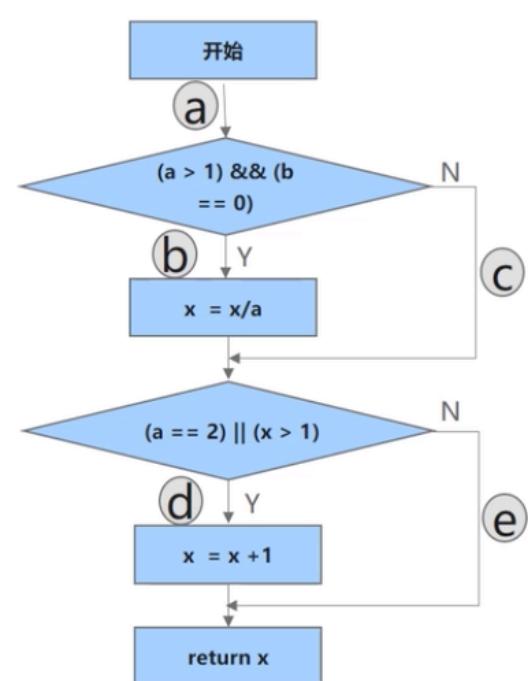
注意：并非所有条件组合都能找到方法进行覆盖，假定有一个分支判断语句形式为：if( $c == 'a' || c == 'b'$ )，则我们是找不到任何一个c的取值能同时满足两个条件都为true的

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)。



# 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,<br>b=0,<br>x=2} | {a=2,<br>b=1,<br>x=1} | {a=1,<br>b=0,<br>x=1} | {a=3,<br>b=0,<br>x=1} |
|-----------------------------|-------------|-----------------------|-----------------------|-----------------------|-----------------------|
| 判定1:<br>(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:<br>(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                   |

## 总结和练习参考答案：

① 语句覆盖:

| 用例<br>编号 | str   | 用例<br>预期 |
|----------|-------|----------|
| 1        | "rar" | 2        |

⑤ 条件组合覆盖:

| 用例<br>编号 | Str   | 用例<br>预期 |
|----------|-------|----------|
| 1        | "ras" | 2        |
| 2        | "sar" | 2        |
| 3        | "ra"  | 1        |

② 判定覆盖:

| 用例<br>编号 | Str    | 用例<br>预期 |
|----------|--------|----------|
| 1        | "rara" | 2        |
| 2        | "r"    | 1        |

⑥ 路径覆盖:

| 用例<br>编号 | Str  | 用例<br>预期 |
|----------|------|----------|
| 1        | "rs" | 2        |
| 2        | "sa" | 1        |

③ 条件覆盖:

| 用例<br>编号 | Str  | 用例<br>预期 |
|----------|------|----------|
| 1        | "ra" | 1        |
| 2        | "ar" | 1        |
| 3        | "sa" | 1        |
| 4        | "as" | 1        |

④ 判定/条件覆盖:

| 用例<br>编号 | Str  | 用例<br>预期 |
|----------|------|----------|
| 1        | "ra" | 1        |
| 2        | "ar" | 1        |
| 3        | "sa" | 1        |
| 4        | "as" | 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

|                 |  |
|-----------------|--|
| 一：开发者测试基础       |  |
| 单元测试&TDD        |  |
| 软件测试术语          |  |
| 开发者测试（DT）的定义和价值 |  |
| 可测试性            |  |
| 测试分层            |  |
| 二：测试设计方法        |  |
| 测试设计概述          |  |
| 常见的测试设计方法       |  |
| 通过代码覆盖分析进行测试补充  |  |
| 三：C/C++测试实现与执行  |  |
| · 测试框架概述        |  |
| gtest测试框架       |  |
| gMock基础知识       |  |
| GDB调试技能         |  |

# 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

## 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 |       |          |       |                                                                                  |
| 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 | ★★☆☆☆ | 可移植性好，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。

# 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>

# 目录

|                |                 |
|----------------|-----------------|
| 一：开发者测试基础      |                 |
|                | 单元测试&TDD        |
|                | 软件测试术语          |
|                | 开发者测试（DT）的定义和价值 |
|                | 可测试性            |
|                | 测试分层            |
| 二：测试设计方法       |                 |
|                | 测试设计概述          |
|                | 常见的测试设计方法       |
|                | 通过代码覆盖分析进行测试补充  |
| 三：C/C++测试实现与执行 |                 |
|                | 测试框架概述          |
|                | · gtest测试框架     |
|                | gMock基础知识       |
|                | GDB调试技能         |

# 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();
}
```

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

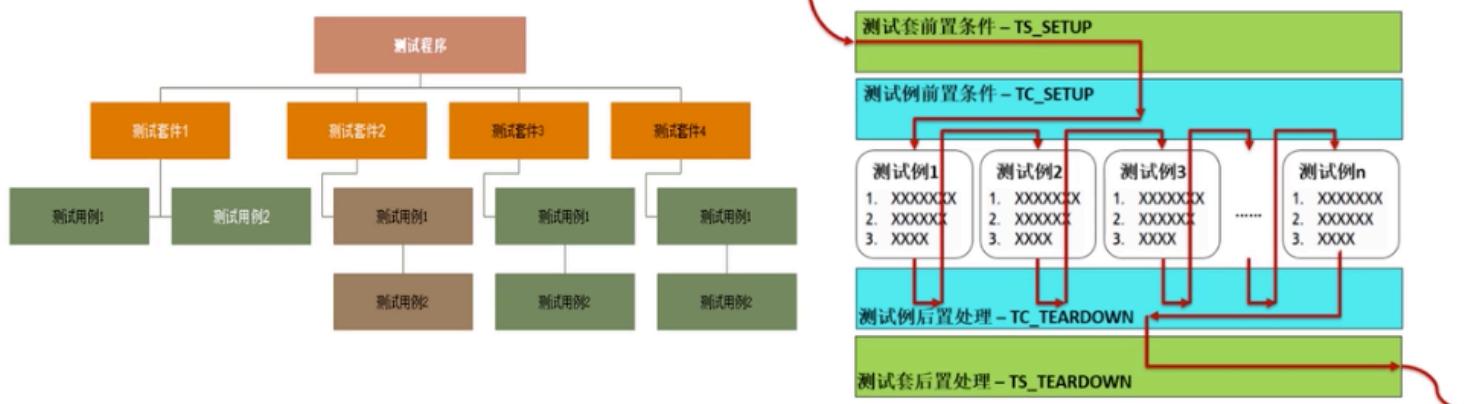
```
// 测试套与测试用例事件
class TestSuite : public Test
{
public:
    static void SetUpTestCase()
    {
        cout << "TestSuite测试套事件：在第一个testcase之前执行" << endl;
    }
    static void TearDownTestCase()
    {
        cout << "TestSuite测试套事件：在最后一个testcase之后执行" << endl;
    }
    virtual void SetUp()
    {
 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

```
// 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
```

# 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

# 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();
}
```

# 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

## 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<,>。

# gtest断言之字符串比较

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

## 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\_FORMAT2(testing::FloatLE, val1, val2);

EXPECT\_PRED\_FORMAT2(testing::DoubleLE, val1, val2);

# 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。

# gtest运行参数之测试用例集合

| 命令行参数                           | 说明                                                                                                                                                                                                                                                                                        |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| --gtest_list_tests              | 使用这个参数时，将不会执行里面的测试用例，而是输出一个用例的列表                                                                                                                                                                                                                                                          |
| --gtest_filter                  | 对执行的测试用例进行过滤，支持通配符<br>? 单个字符<br>* 任意字符<br>- 排除，如， -a 表示除了a<br>: 取或，如， a:b 表示a或b                                                                                                                                                                                                           |
| --gtest_also_run_disabled_tests | 执行用例时，同时也执行被置为无效的测试用例。关于设置测试用例无效的方法为：<br>在测试用例名称或测试名称中添加DISABLED前缀，比如：<br><pre>// Tests that Foo does Abc.<br/>TEST(FooTest, DISABLED_DoesAbc) { }<br/>class DISABLED_BarTest : public testing::Test { };<br/>// Tests that Bar does Xyz.<br/>TEST_F(DISABLED_BarTest, DoesXyz) { }</pre> |
| --gtest_repeat=[COUNT]          | 设置用例重复运行次数，非常棒的功能！比如：<br>--gtest_repeat=1000 重复执行1000次，即使中途出现错误。<br>--gtest_repeat=-1 无限次数执行<br>--gtest_repeat=1000 --gtest_break_on_failure 重复执行1000次，并且在第一个错误发生时立即停止。这个功能对调试非常有用。                                                                                                       |

## gtest运行参数之用例异常处理

| 命令行参数                    | 说明                                                                                                                                                                   |
|--------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| --gtest_break_on_failure | 调试模式下，当用例失败时停止，方便调试                                                                                                                                                  |
| --gtest_throw_on_failure | 当用例失败时以C++异常的方式抛出                                                                                                                                                    |
| --gtest_catch_exceptions | 是否捕捉异常。gtest 默认是不捕捉异常的，因此，假如你的测试用例抛了一个异常，很可能会弹出一个对话框，这非常的不友好，同时也阻碍了测试用例的运行。如果想不弹这个框，可以通过设置这个参数来实现。如将--gtest_catch_exceptions 设置为一个非零的数。<br><br>注意：这个参数只在Windows 下有效。 |

## gtest运行参数示例

--gtest\_filter是一个比较常用的命令行参数，用于进行用例过滤，可以指定满足过滤条件的用例执行，在对执行失败用例进行调试时非常有用：

- 在Visual Studio中，直接在调试的命令参数中添加过滤参数：

```
--gtest_filter=BbhSuite.ul_cell_comp_reconfig_switch_assist_serving
```

- 在Linux中GDB调试中可以通过set args进行设置：

```
set args --gtest_filter=BbhSuite.ul_cell_comp_reconfig_switch_assist_serving
```

- 执行用例时，可直接在用例后添加参数进行过滤：

```
./ULCTRL --gtest_filter=BbhSuite.ul_cell_comp_reconfig_switch_assist_serving
```

上述参数指定执行BbhSuite测试套中的ul\_cell\_comp\_reconfig\_switch\_assist\_serving测试用例。

# gtest参数化

在设计测试用例时，经常需要考虑给被测函数传入不同参数值的情况。我们之前的做法通常是写一个通用方法，然后再编写测试用例调用它。gtest提供了一个灵活的参数化测试方案，使用时的步骤如下：

## 1. 指定参数类型

添加一个类，继承自`testing::TestWithParam<T>`，其中T就是你需要参数化的参数类型。

## 2. 使用参数进行用例测试

使用`TEST_P`宏，用`GetParam()`获取当前的参数的具体值。

## 3. 定义参数范围

使用`INSTANTIATE_TEST_CASE_P`宏定义参数测试参数范围

```
// 参数化测试
class IsPrimeParamTest : public::testing::TestWithParam<int> {};

TEST_P(IsPrimeParamTest, HandleTrueReturn)
{
    int n = GetParam();
    EXPECT_TRUE(IsPrime(n));
}

INSTANTIATE_TEST_CASE_P(TrueReturn, IsPrimeParamTest,
    testing::Values(3, 5, 11, 23, 17));
```

**第一个参数**是测试用例的前缀，可以任意取名。**第二个参数**是测试用例的名称，需要与之前定义参数化的类名称相同。

**第三个参数**可以理解为参数生成器，`testing::Values`表示使用括号内的参数。

## gtest参数化之参数生成函数

gtest提供了一系列的参数生成的函数，供参数化使用：

| 函数名称                                            | 说明                                                                                     |
|-------------------------------------------------|----------------------------------------------------------------------------------------|
| Range(begin, end[, step])                       | 范围在begin~end之间，步长为step，不包括end                                                          |
| Values(v1, v2, ..., vN)                         | v1,v2到vN的值                                                                             |
| ValuesIn(container) and<br>ValuesIn(begin, end) | 从一个C类型的数组或是STL容器，或是迭代器中取值                                                              |
| Bool()                                          | 取false 和 true 两个值                                                                      |
| Combine(g1, g2, ..., gN)                        | 将g1,g2,...gN进行排列组合，g1,g2,...gN本身是一个参数生成器，每次分别从g1,g2,...gN中各取出一个值，组合成一个元组(Tuple)作为一个参数。 |

# gtest死亡测试之DEATH

1. statement是被测试的代码语句
2. regex是一个正则表达式，用来匹配异常时stderr中输出的内容。

```
void Foo()
{
    int *pInt = 0;
    *pInt = 42;
}

TEST(FooDeathTest, Demo)
{
    EXPECT_DEATH(Foo(), "");
```

**重要：**编写死亡测试用例时，TEST 的第一个参数，即 testcase\_name，请使用 DeathTest 后缀。原因是 gtest 会优先运行死亡测试用例，应该是为线程安全考虑。

## gtest死亡测试之DEBUG\_DEATH

在Debug版和Release版本下，\*\_DEBUG\_DEATH的定义不一样。因为很多异常只会在Debug版本下抛出，而在Release版本下不会抛出，所以针对Debug和Release分别做了不同的处理，下面是gtest中的定义和自带示例：

```
#ifdef NDEBUG

#define EXPECT_DEBUG_DEATH(statement, regex) \
do { statement; } while (false)

#define ASSERT_DEBUG_DEATH(statement, regex) \
do { statement; } while (false)

#else

#define EXPECT_DEBUG_DEATH(statement, regex) \
EXPECT_DEATH(statement, regex)

#define ASSERT_DEBUG_DEATH(statement, regex) \
ASSERT_DEATH(statement, regex)

#endif // DEBUG

int DieInDebugElse12(int* sideeffect) {
    if (sideeffect) *sideeffect = 12;
#endif // NDEBUG
    GTEST_LOG_(FATAL, "debug death inside DieInDebugElse12()");
#endif // NDEBUG
    return 12;
}

TEST(TestCase, TestDieOr12WorksInDgbAndOpt)
{
    int sideeffect = 0;
    // Only asserts in dbg.
    EXPECT_DEBUG_DEATH(DieInDebugElse12(&sideeffect), "death");

    #ifdef NDEBUG
    // opt-mode has sideeffect visible.
    EXPECT_EQ(12, sideeffect);
    #else
    // dbg-mode no visible sideeffect.
    EXPECT_EQ(0, sideeffect);
    #endif
}
```

## gtest死亡测试之EXIT

1. statement是被测试的代码语句
2. predicate在这里必须是一个委托，接收int型参数，并返回bool。

只有返回值为true时，死亡测试用例才算通过

gtest提供了一些常用的predicate：

testing::ExitedWithCode(exit\_code)

testing::KilledBySignal(signal\_number) //windows下不支持

3. regex 是一个正则表达式，用来匹配异常时在stderr 中输出的内容

```
TEST(ExitDeathTest, Demo)
{
    EXPECT_EXIT(_exit(1), testing::ExitedWithCode(1), "");
}
```

**重要：**这里，要说明的是，\*\_DEATH 其实是对\*\_EXIT 进行的一次包装，\*\_DEATH的predicate为判断进程是否以非0退出码退出或被一个信号杀死。

## 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

目录

## 一：开发者测试基础

单元测试&TDD

软件测试术语

## 开发者测试（DT）的定义和价值

## 可测试性

测试分层

## 二：测试设计方法

测试设计概述

常见的测试设计方法

#### 通过代码覆盖分析进行测试补充

### • 三：C/C++测试实现与执行

测试框架概述

QTest 测试框架

## • gMock基础知识

GDB调试技能

## 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

## 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。

# Mock和Stub有什么区别?

1. Mock: 以任何形式参与测试并产生某种结果
2. Stub: 无需验证它是何时何地以何种方式被调用

## 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

# gMock例子

典型的gMock的测试代码可能如下：

```
1 // 定义需要模拟的接口类
2 class FooInterface {
3     public:
4         virtual ~FooInterface() {}
5         virtual std::string getArbitraryString() = 0;
6
7         virtual int getPosition() = 0;
8     };
9
10
11 // 构造类
12 class MockFoo : public FooInterface {
13     public:
14         MOCK_METHOD0(getArbitraryString, std::string());
15         MOCK_METHOD0(getPosition, int());
16     };
17
18
19 #include "stdafx.h"
20
21 using namespace ::seamless;
22 using namespace std;
23
24 using ::testing::Return;
25
26
27 int main(int argc, char** argv) {
28     // Since Google Mock depends on Google Test, InitGoogleMock() is also responsible for initializing
29     // therefore there's no need for calling testing::InitGoogleTest() separately.
30     ::testing::InitGoogleMock(&argc, argv);
31     int n = 100;
32     string value = "Hello World!";
33     MockFoo mockFoo;
34     EXPECT_CALL(mockFoo, getArbitraryString())
35         .Times(1)
36         .WillOnce(Return(value));
37     string returnValue = mockFoo.getArbitraryString();
38     cout << "Returned Value: " << returnValue << endl;
39     // 在这里Times(2)意思是调用两次，但是下边只调用了一次，所以会报出异常
40     EXPECT_CALL(mockFoo, getPosition())
41         .Times(2)
42         .WillRepeatedly(Return(n++));
43     int val = mockFoo.getPosition(); // 100
44     cout << "Returned Value: " << val << endl;
45     // getPosition指定了调用两次，这里只调用了一次，所以运行结果显示出错
46     return EXIT_SUCCESS;
47 }
```

引自：[https://www.cnblogs.com/jycboy/p/gmock\\_summary.html#autoid-7-4-0](https://www.cnblogs.com/jycboy/p/gmock_summary.html#autoid-7-4-0)

## 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

## 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));**

# 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

# gMock: 匹配器

| 类型    | 关键字                                                                                                                                                                            | 用法说明                                                                                                                                                                                    |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 通配匹配  | <code>A&lt;type&gt;()</code> 或者 <code>An&lt;type&gt;()</code>                                                                                                                  | <ul style="list-style-type: none"><li>• 类型匹配任意匹配</li><li>• 任何类型为type的参数都可以匹配</li></ul>                                                                                                  |
| 一般匹配  | <code>Ge(value)</code> 、 <code>Gt(value)</code> 、 <code>Le(value)</code> 、<br><code>Lt(value)</code> 、 <code>Ne(value)</code> 、 <code>IsNull()</code> 、 <code>NotNull()</code> | <code>参数&gt; =value</code> 、 <code>参数&gt; value</code> 、 <code>参数&lt;=value</code> 、 <code>参数&lt;value</code> 、<br><code>参数!=value</code> 、 <code>参数是个空指针</code> 、 <code>参数为非空指针</code> |
| 浮点数匹配 | <code>DoubleEq(a_double)</code><br><code>FloatEq(a_float)</code>                                                                                                               | <ul style="list-style-type: none"><li>• 参数= <code>a_double</code></li><li>• 参数= <code>a_float</code></li></ul>                                                                          |
| 字符串匹配 | <code>EndsWith(suffix)</code><br><code>HasSubstr(string)</code><br><code>StartsWith(prefix)</code><br><code>StrEq(string)</code><br><code>StrNe(string)</code>                 | <ul style="list-style-type: none"><li>• 参数以后缀suffix结尾</li><li>• 参数包含string子字符</li><li>• 参数以前缀prefix开始</li><li>• 参数与string相同，大小写敏感</li><li>• 参数与string不同，大小写敏感</li></ul>                 |
| 容器匹配  | <code>ContainerEq(container)</code><br><code>Contains(e)</code><br><code>Each(e)</code><br><code>Sizes(m)</code><br><code>IsEmpty</code>                                       | <ul style="list-style-type: none"><li>• 参数和container有同样内容</li><li>• 参数包含满足e的元素，e可以是值</li><li>• 参数中每个元素都满足e，e可以是值</li><li>• 参数的长度符合m的要求</li><li>• 参数为空容器</li></ul>                       |
| 指针匹配  | <code>Pointee(m)</code>                                                                                                                                                        | 参数(指针)指向的内容符合m                                                                                                                                                                          |

## 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

## 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设置的断点处停住。
- ◆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



# 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。

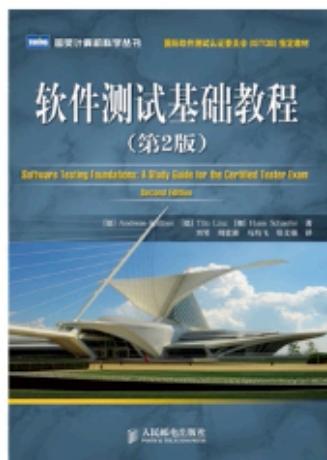
## 方式三：调试已在运行的文件 (通过pid ) #gdb [program] [pid]

- 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)。

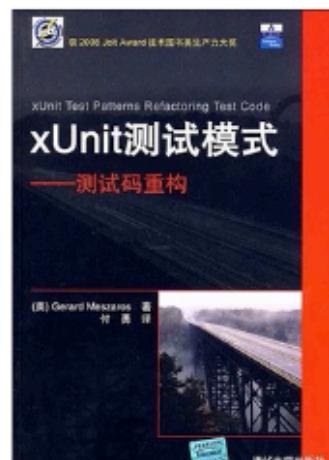
# GDB指令介绍

| 关键字              | 用法说明                                        |
|------------------|---------------------------------------------|
| <b>set</b>       | 用于设置gdb内部的一些环境与运行时的参数                       |
| <b>print</b>     | 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    |
| <b>show</b>      | 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     |
| <b>info</b>      | 可以描述程序的状态                                   |
| <b>help</b>      | 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      |
| <b>backtrace</b> | 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 |
| <b>break</b>     | 用于设置程序断点                                    |
| <b>list</b>      | 可缩写为l，用于列出源码                                |

## 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》