# 单元测试之道

提高系统稳定性、健壮性、为企业切实降低开发成本

尹洪亮 Kevin

系统架构师

# 单测理论

Unit Test Theory

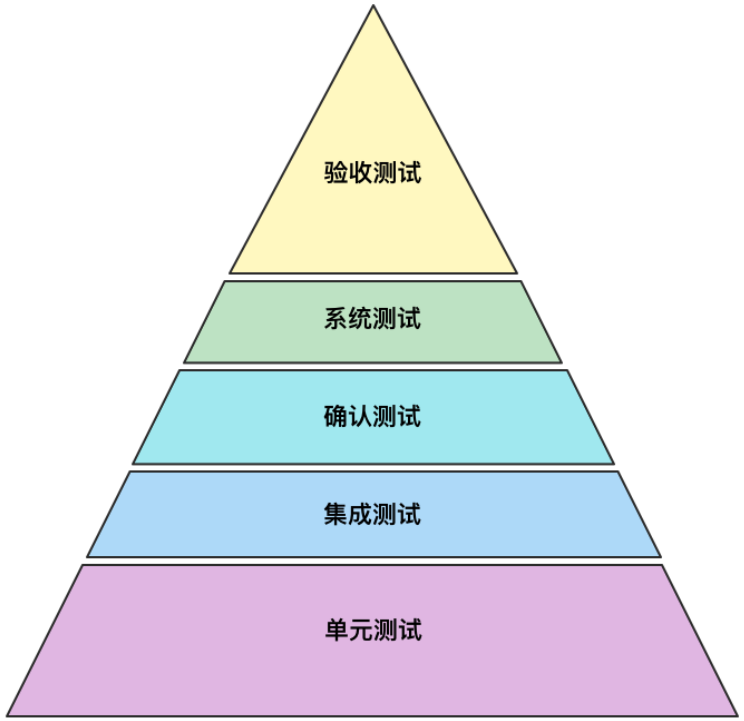
# 什么是单元测试

单元测试(unittesting)，是指对软件中的最小可测试单元进行检查和验证。

对于单元测试中单元的含义，一般来说，要根据实际情况去判定其具体含义，如C语言中单元指一个函数，Java里单元指一个类/函数，图形化的软件中可以指一个窗口或一个菜单等。

总的来说，单元就是人为规定的最小的被测功能模块。单元测试是在软件开发过程中要进行的最低级别的测试活动，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。

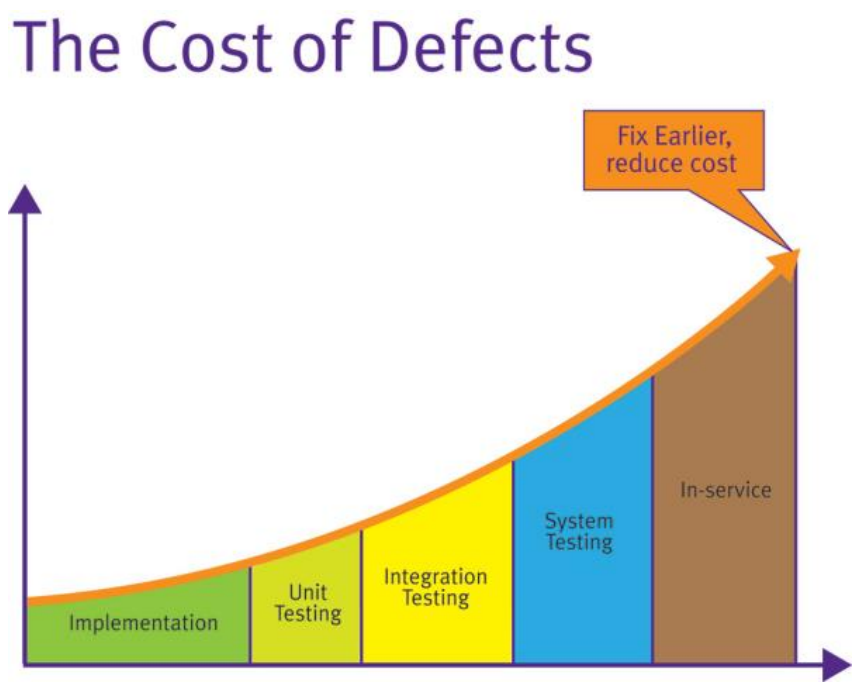
# 测试金字塔




单元测试是整个测试金字塔的基石，单元测试的质量高低对整个系统的健壮性影响越大。

- **单元测试:** 又称模块测试（也有称为方法测试），是针对软件设计的最小单位——程序模块进行正确性检验的测试工作。其目的在于检查每个程序单元能否正确实现详细设计说明中的模块功能、性能、接口和设计约束等要求，发现各模块内部可能存在的各种错误。单元测试需要从程序的内部结构出发设计测试用例。多个模块可以平行地独立进行单元测试
- **集成测试:** 也叫做组装测试（也有称为模块测试）。通常在单元测试的基础上，将所有的程序模块进行有序的、递增的测试。集成测试是检验程序单元或部件的接口关系，逐步集成为符合概要设计要求的程序部件或整个系统
- **确认测试:** 也叫有效性测试。是在模拟的环境下，验证软件的所有功能和性能及其他特性是否与用户的预期要求一致。通过了确认测试之后的软件，才具备了进入系统测试阶段的资质
- **系统测试:** 是在真实的系统运行的环境下，检查完整的程序系统能否和系统（包括硬件、外设、网络和系统软件、支持平台等）正确配置、连接，并最终满足用户的所有需求
- **验收测试:** 是软件产品检验的最后一个环节。按照项目任务书或合同、供需双方约定的验收依据文档进行的对整个系统的测试与评审，决定是否接收或拒收系统。

# 缺陷与成本




- **缺陷暴露越晚成本越高：**无论缺陷的严重程度如何，越晚发现则伤害越大，缺陷的成本与发现他们所花费的时间成正比。
- **消除缺陷和返工占用大量成本：**软件开发的很大一部分成本包括错误消除和项目返工。返工过程的成本高于初始过程，因此有必要在设计和需求阶段及早发现缺陷以避免这种额外费用。大量的缺陷通常发生在项目的初始阶段，早期的缺陷检测将降低项目的总体成本。
- **越晚解决缺陷成本越高：**即使是微小的缺陷，在后期进行根本原因分析和回归测试时，也会花费巨大的成本；后期发现的缺陷一般不是犹豫缺少代码或者录入错误造成的，而是更加的隐蔽。
- **缺陷都是在早期引入的：**大量的缺陷都是在需求阶段、设计阶段引入的，而在编码期如果有充分的单元测试，就很容易发现这些问题。

# 质量差的原因和解决方案

## 关注流程错误而不是代码错误

代码错误并不是引发缺陷的根本原因，代码错误只是一个表象，而是流程错误导致的，流程更需要改进。

- ✓ 是否任何测试用例都被阻塞到最后一分钟才得以解决。
- ✓ 是否单元测试覆盖率很低。
- ✓ 是否在即将要进行验收测试时还在进行代码的修改。
- ✓ 是否经常发生紧急错误

## 改善流程和机制

很多公司都遵循严格的时间表和敏捷开发流程，缺陷数量直线飙升，需要强有力的措施，很多预防机制。

- ✓ 提高代码评审覆盖率，加强代码审查、code review。
- ✓ 提高单元测试覆盖率，要覆盖到所有的核心代码行。
- ✓ 构建单元测试自动化，对于不断迭代的系统，能够保障新修改的代码不影响原有功能。
- ✓ 足够多的测试用例，测试用例要具有足够的覆盖度

# 单测规范与原则

Specifications and principles of unit testing

# 工程目录规范

- ✓ src/main/java　　应用程序Java Source目
- ✓ src/main/resource　　应用程序资源文件目录
- ✓ src/test/java　　应用程序单元测试Java Source目录
- ✓ src/test/resource　　应用程序单元测试资源文件目录(为自动化测试准备)

# 测试用例规范

- **包名一致性:** src/test/java下的包名 和 对应的src/main/java下的包名保持一致
- **类名命名规范:** 类名+Test
  - ✓ 例1：UserServiceTest
  - ✓ 例2：OrderControllerTest
- **测试方法命名规范：要求易读、易于理解，test方法名_测试条件**
  - ✓ 例1：testGetUserById　　测试getUserById方法
  - ✓ 例2：testGetAllUser　　测试getAllUser方法
  - ✓ 例3：testAddUserById_less_10　　测试addUserById方法，当id小于10的时候
  - ✓ 例4：testAddUserById_not_less_10　　测试addUserById方法，当id不小于10的时候

# AIR原则

**单元测试在线上运行时，应该想空气（AIR)一样感觉不到存在。**

- **A：Automatic，自动化**
  - ✓ 单元测试应该是全自动执行的，而非交互式的；可以在项目构建过程中执行，定期执行。
  - ✓ 单元测试的结果不需要人工检查，而是用assert断言来验证。
- **I：Independent，独立性**
  - ✓ 保持单元测试的独立性，是为了保证单元测试的稳定性与可靠性，单元测试用例之间决不能相互调用，也不能依赖于彼此的先后执行顺序。
- **R：Repeatable，可重复性**
  - ✓ 单元测试可重复执行的，不能受外界环境的限制。
  - ✓ 为了不受外界环境的影响，要求设计代码时就把sut的依赖改成注入，在测试时用spring这样的DI框架注入到一个本地实现或者Mock实现。

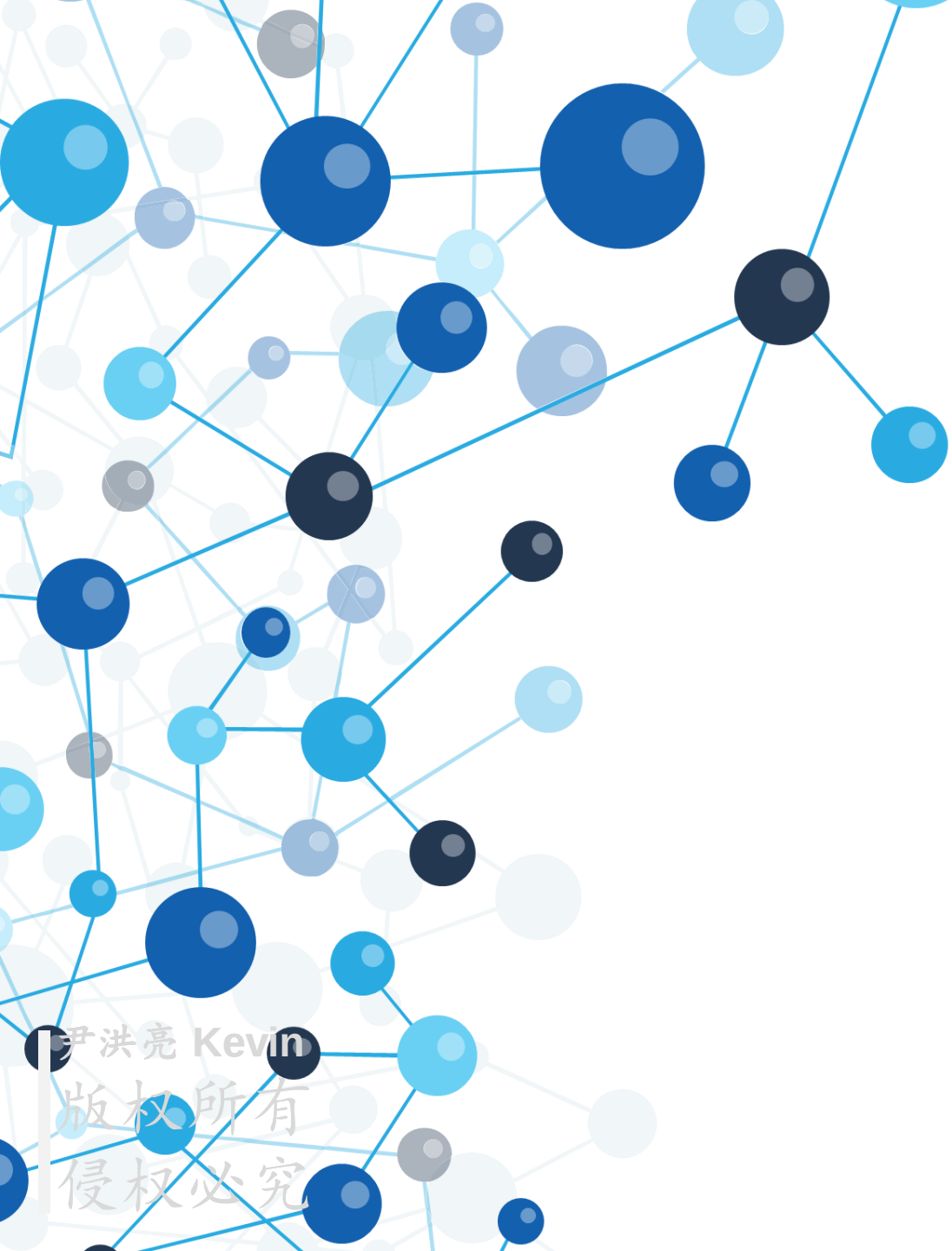
# BCDE原则

**BCDE原则：编写单元测试代码要遵循BCDE原则，保证模块的交付质量**

- 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。
- 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，单元测试的目的是证明程序有错，而不是证明程序无错。为了发现代码中潜在的错误，我们需奥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

# 其他原则

- ✓ **粒度小**：对于单元测试，要保证测试粒度足单一和小，能够精准定位问题，单测粒度要控制在方法级别。
- ✓ **核心通过率高**：核心业务、应用、模块、方法的单元测试要确保通过率为100%。
- ✓ **执行速度快**：单个测试用例执行时间不能超过10秒，整个项目的单测时间控制在20分钟以内。
- ✓ **覆盖率高**：整体语句、核心模块的语句覆盖率、分支覆盖率都要达到100%。
- ✓ **避免数据污染**：如果需要执行数据库相关测试，要设定回滚或打标机制。



# 单元测试实战

Mokito Unit Test Framework

# 单测工具

- ✓ **Junit**：https://junit.org/junit5/
- ✓ **Spring Test**：https://docs.spring.io/spring-framework/docs/6.0.x/reference/html/testing.html
- ✓ **AssertJ**：断言库，https://joel-costigliola.github.io/assertj/
- ✓ **Hamcrest**：断言库/谓词库， http://hamcrest.org/JavaHamcrest/index
- ✓ **Mockito**：Mock框架，https://site.mockito.org/
- ✓ **JSONassert**：json断言库， https://github.com/skyscreamer/JSONassert
- ✓ JsonPath：json版本的XPath，https://github.com/json-path/JsonPath
- ✓ TestableMock: Mock框架, https://alibaba.github.io/testable-mock/#/

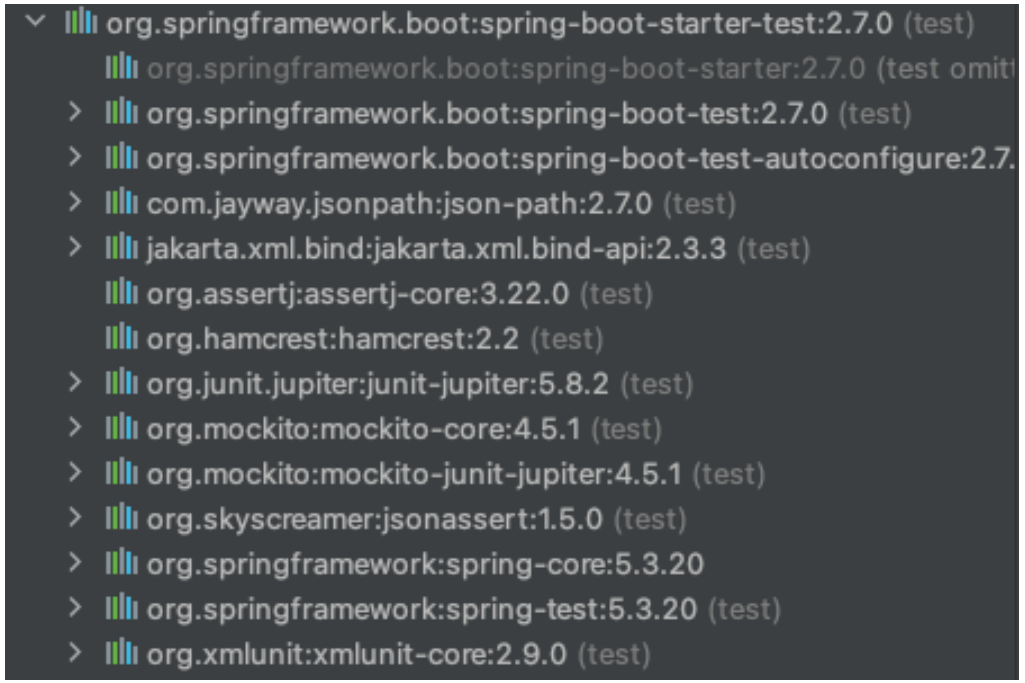
# Maven 引入 Mokito框架依赖

maven

```maven
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>3.3.3</version>
  <scope>test</scope>
</dependency>
```

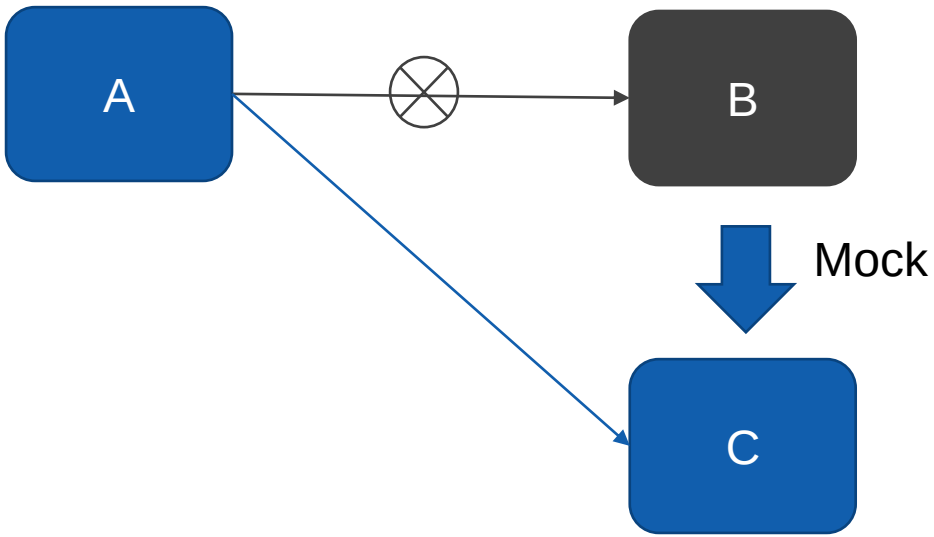
# Spring 集成 Mokito框架

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```



# 什么是Mock

模拟被测试对象的依赖（包括数据和行为）



# Mokito mock对象

1 使用Mockito.mock(clazz)静态方法来mock对象

示例 com.demo.unit.mokito.MockitoTest1

2 使用@Mock注解方式来mock对象，使用@ExtendWith(MockitoExtension.class) 注解来开启Mock注解

示例 com.demo.unit.mokito.MockitoTest2

3 使用MockitoAnnotations.initMocks(this); 和 MockitoAnnotations.openMocks(this); 方法来开启Mock注解。其中initMocks方法已经不推荐使用

示例 com.demo.unit.mokito.MockitoTest3

# Mokito 验证性测试

场景 验证某方法是否被调用过 验证方法调用指定次数 验证方法调用顺序

示例 com.demo.unit.mokito.MockitoTest4

- 核心方法：
  - ✓ org.mockito.Mockito#verify(T, VerificationMode)
  - ✓ org.mockito.InOrder#verify(T)
- 参数说明：
  - ✓ T 代表被mock的对象
  - ✓ org.mockito.verification.VerificationMode代表验证模式（调用次数）
  - ✓ VerificationModeFactory工厂类提供了很多返回验证模式的方法，并且在Mockito类中进行了封装。

N次、最多1次、最少1次、最多N次、最少N次

# Mokito mock方法返回值

场景

mock方法返回值

mock方法抛出异常

mock方法返回变化性返回值

示例

com.demo.unit.mokito.MockitoTest5

# 使用断言判断

作用

**断言(assertion)**是一种在程序中的一阶逻辑(如：一个结果为真或假的逻辑判断式)，目的为了表示与验证软件开发者预期的结果——当程序执行到断言的位置时，对应的断言应该为真。若断言不为真时，程序会中止执行，并给出错误信息。

示例

com.demo.unit.mokito.MockitoTest6

**断言**

方案1、org.junit.jupiter.api.Assertions

方案2、org.hamcrest.MatcherAssert

Assertions 一般以静态化方式引入: import static org.junit.jupiter.api.Assertions.*;

# Mockito @Spy

作用 @Spy注解或者spy函数修饰的对象，其上的方法都是真实调用的。 而Mock的对象都是假调用

示例 com.demo.unit.mokito.MockitoTest7

# Unnecessary stubbings detected 检测到不必要的存根

示例 com.demo.unit.mokito.MockitoTest8

异常 有时候会遇到Unnecessary stubbings detected.报错是怎么回事？

org.mockito.exceptions.misusing.UnnecessaryStubbingException:

Unnecessary stubbings detected.

Clean & maintainable test code requires zero unnecessary code.

Following stubbings are unnecessary (click to navigate to relevant line of code):

1. -> at com.demo.unit.mokito.MockitoTest8.testSpy(MockitoTest8.java:29)

Please remove unnecessary stubbings or use 'lenient' strictness. More info: javadoc for UnnecessaryStubbingException class.

方案 在@Mock注解上使用宽松模式： @Mock(lenient = true)

# Mockito 框架的工作原理 & 实现一个微型Mockito

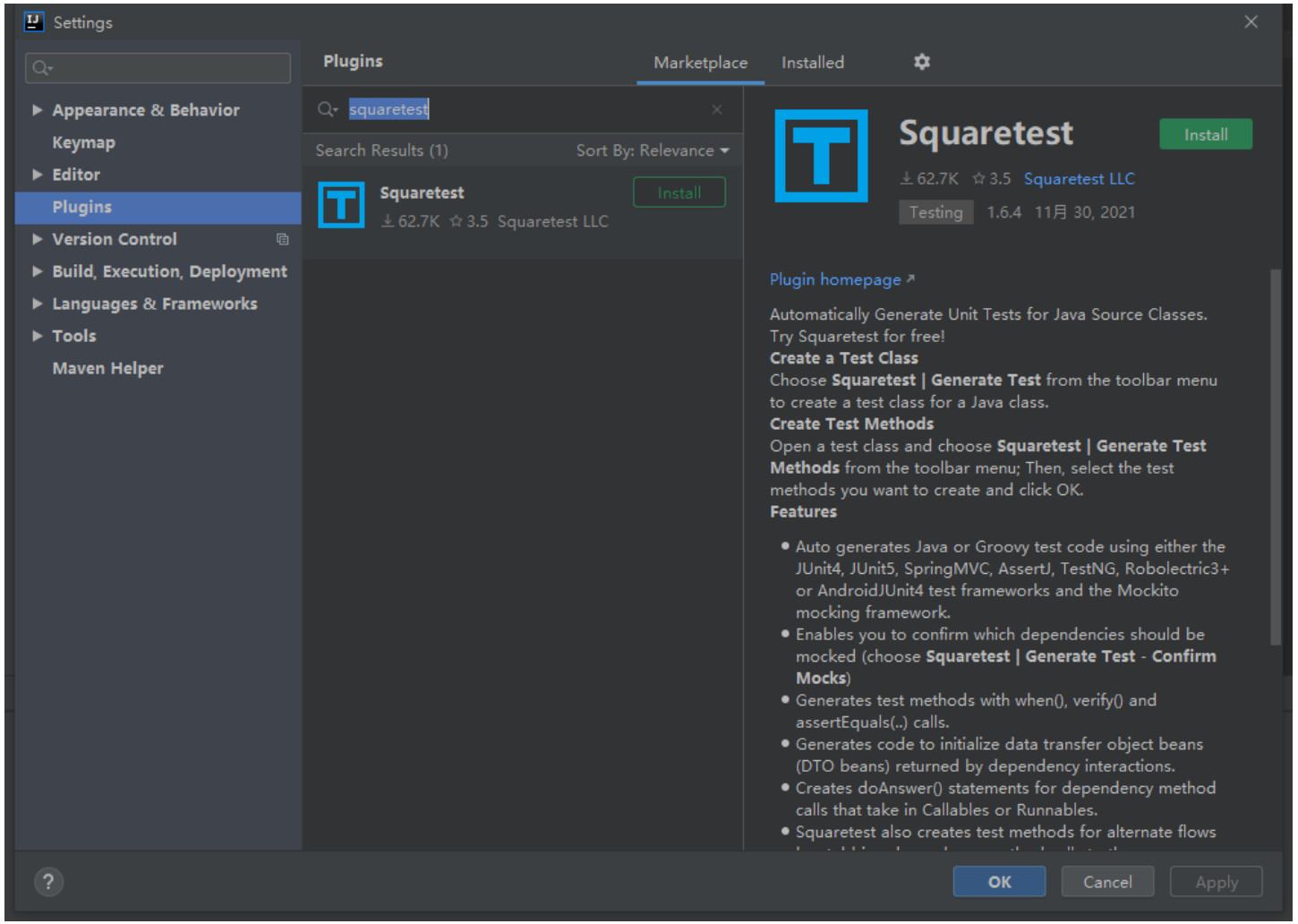
实现 com.demo.unit.simple

测试 com.demo.unit.simple.SimpleMockitoTest

# 单元测试插件（提效）

搜索并安装插件：**Squaretest**

可一键快速生成Mokito测试用例



# JSONassert

JSONassert Library

# JSONassert

10、JSONassert断言

参考：com.demo.unit.jsonassert.JSONAssertDemo

官方：http://jsonassert.skyscreamer.org/

官方： http://jsonassert.skyscreamer.org/quickstart.html

官方：http://jsonassert.skyscreamer.org/cookbook.html

# 单测覆盖率

Unit test coverage

# Jacoco

代码覆盖（Code coverage）是软件测试中的一种度量，描述程序中源代码被测试的比例和程度，所得比例称为代码覆盖率。

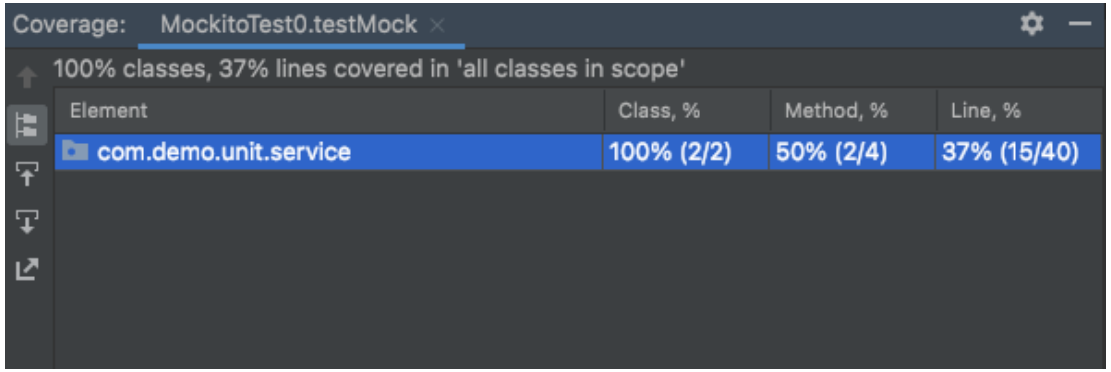
单元测试覆盖率常用的框架有JaCoCo、EMMA和Cobertura

- ✓ Jacoco工具： https://www.eclemma.org/jacoco/
- ✓ Emma工具：http://emma.sourceforge.net/
- ✓ Cobertra工具：http://cobertura.github.io/cobertura/

# IDEA 单测覆盖率

IDEA 已经集成了覆盖率插件


1 执行单元测试的时候




2 执行完毕查看覆盖率

在这里看看: https://youtrack.jetbrains.com/issue/IDEABKL-5941
您可以将-Djava.io.tmpdir参数添加到idea的启动脚本中，或替换TMP系统属性.
也许您还必须在安装目录中的idea.properties中更改idea.config.path和idea.system.path.

# Jacoco

1 Maven引入jacoco-maven-plugin

```xml
<build>
    <plugins>
        <!-- 生成JaCoCo覆盖率数据插件 -->
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.8</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                </execution>
                <execution>
                    <id>report</id>
                    <phase>test</phase>
                    <goals>
                        <goal>report</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

2

```
# 本地通过命令行执行命令，运行单元测试
mvn clean test -Dmaven.test.failure.ignore=true
```

3 生产jacoco报告

# Jacoco 报告

1

包级别覆盖率

unit-test-demo

## unit-test-demo

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| com.demo.unit.po | | 20% | | 20% | 202 | 237 | 319 | 362 | 157 | 189 | 4 | 10 |
| com.demo.unit.dto | | 77% | | 36% | 10 | 21 | 0 | 5 | 0 | 10 | 0 | 1 |
| com.demo.unit.controller | | 0% | | n/a | 2 | 2 | 3 | 3 | 2 | 2 | 1 | 1 |
| com.demo.unit | | 0% | | n/a | 2 | 2 | 3 | 3 | 2 | 2 | 1 | 1 |
| com.demo.unit.utils | | 70% | | 50% | 2 | 4 | 2 | 7 | 0 | 2 | 0 | 1 |
| com.demo.unit.simple | | 95% | | 58% | 6 | 17 | 1 | 29 | 1 | 11 | 0 | 4 |
| com.demo.unit.service | | 99% | | 87% | 1 | 10 | 0 | 40 | 0 | 6 | 0 | 2 |
| Total | 1,385 of 2,163 | 35% | 98 of 142 | 30% | 225 | 293 | 328 | 449 | 162 | 222 | 6 | 20 |

2

类级别覆盖率

unit-test-demo > com.demo.unit.service

## com.demo.unit.service

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AddressService | | 100% | | 100% | 0 | 3 | 0 | 13 | 0 | 2 | 0 | 1 |
| UserService | | 99% | | 83% | 1 | 7 | 0 | 27 | 0 | 4 | 0 | 1 |
| Total | 1 of 202 | 99% | 1 of 8 | 87% | 1 | 10 | 0 | 40 | 0 | 6 | 0 | 2 |

3

方法级覆盖率

unit-test-demo > com.demo.unit.service > AddressService

## AddressService

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods |
|---|---|---|---|---|---|---|---|---|---|---|
| AddressService() | | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| queryUsersByCity(String) | | 100% | | 100% | 0 | 2 | 0 | 12 | 0 | 1 |
| Total | 0 of 53 | 100% | 0 of 2 | 100% | 0 | 3 | 0 | 13 | 0 | 2 |

# Jacoco 报告

4

代码级覆盖率

```
14. @Service
15. public class UserService {
16. 
17.     @Resource
18.     private TUserDao userDao;
19.     @Resource
20.     private TAddressDao addressDao;
21. 
22.     public TUser getUserById(Integer id){
23.         return userDao.selectByPrimaryKey(id);
24.     }
25. 
26.     public Boolean addUserById(Integer id){
27.         TUser tUser = new TUser();
28.         if(id>10){
29.             tUser.setUserid(id);
30.             tUser.setUsername("1username"+id);
31.             tUser.setPassword("1password"+id);
32.             tUser.setPhone("1phone"+id);
33.         }else{
34.             tUser.setUserid(id);
35.             tUser.setUsername("2username"+id);
36.             tUser.setPassword("2password"+id);
37.             tUser.setPhone("2phone"+id);
38.         }
39.         int result = this.userDao.insertSelective(tUser);
40.         return result == 1;
41.     }
42. 
43.     public List<UserDetailDTO> getAllUser(){
44. 
45.         List<UserDetailDTO> list = new ArrayList<UserDetailDTO>();
46. 
47.         List<TUser> tUsers = this.userDao.selectByExample(null);
48.         for (TUser tUser : tUsers) {
49. 
50.             UserDetailDTO userDetailDTO = new UserDetailDTO();
51.             userDetailDTO.setUser(tUser);
52. 
53.             Integer userid = tUser.getUserid();
54. 
55.             TAddressExample tAddressExample = new TAddressExample();
56.             tAddressExample.createCriteria().andUserIdEqualTo(userid);
57. 
58.             List<TAddress> tAddresses = this.addressDao.selectByExample(tAddressExample);
59.             userDetailDTO.setAddressList(tAddresses);
60. 
61.             list.add(userDetailDTO);
62.         }
63. 
64.         return list;
65. 
66.     }
```

# Jacoco 覆盖率报告说明

圈复杂度（Cyclomatic Complexity）

指令（C0 Coverage）

行（Lines）

unit-test-demo

## unit-test-demo

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| com.demo.unit.po | | 20% | | 20% | 202 | 237 | 319 | 362 | 157 | 189 | 4 | 10 |
| com.demo.unit.dto | | 77% | | 36% | 10 | 21 | 0 | 5 | 0 | 10 | 0 | 1 |
| com.demo.unit.controller | | 0% | | n/a | 2 | 2 | 3 | 3 | 2 | 2 | 1 | 1 |
| com.demo.unit | | 0% | | n/a | 2 | 2 | 3 | 3 | 2 | 2 | 1 | 1 |
| com.demo.unit.utils | | 70% | | 50% | 2 | 4 | 2 | 7 | 0 | 2 | 0 | 1 |
| com.demo.unit.simple | | 95% | | 58% | 6 | 17 | 1 | 29 | 1 | 11 | 0 | 4 |
| com.demo.unit.service | | 99% | | 87% | 1 | 10 | 0 | 40 | 0 | 6 | 0 | 2 |
| Total | 1,385 of 2,163 | 35% | 98 of 142 | 30% | 225 | 293 | 328 | 449 | 162 | 222 | 6 | 20 |

分支（C1 Coverage）

方法（Methods）

类（Classes）

# Jacoco 统计指标

- ✓ **指令（C0 Coverage）：** JaCoCo计数的最小单元是单一的Java字节码指令。指令覆盖率提供了关于字节码执行数量、未执行数量的信息。
- ✓ **分支（C1 Coverage）：** 对所有的if和switch语句计算分支覆盖率。统计在方法中分支执行数量、未执行数量的信息。但要注意，异常处理不在此计算范围内。
- ✓ **圈复杂度（Cyclomatic Complexity）：** 对非抽象方法计算圈复杂度，并汇总类、包和组的（圈）复杂度。这个值可以做为单元测试用例是否完全覆盖的参考。
- ✓ **行（Lines）：** 一行可能包含一条或多条指令，如果至少有一条指令被执行了，那么该行就算作是被执行了。
- ✓ **方法（Methods）：** 每个非抽象方法至少包含一条指令。如果至少有一条指令被执行了，那么该方法就算作是被执行了。
- ✓ **类（Classes）：** 如果类中至少有一个方法被执行了，那么该类就算作是被执行了。

# 总结

Summary

# 总结

## 01

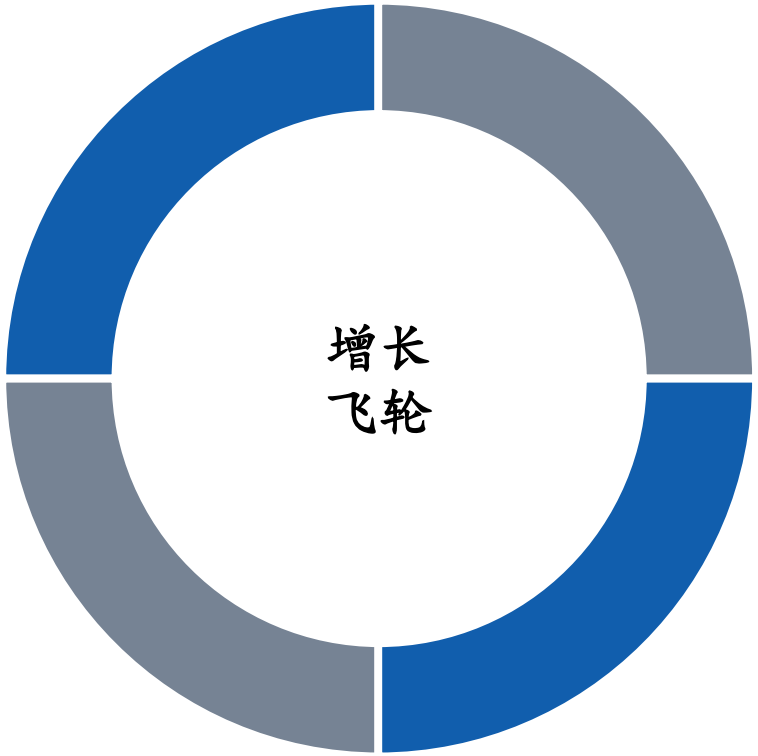
**改进流程**

改进研发流程，坚决加入单元测试的环节，改变观念

## 02

**建立规范**

建立良好的单元测试编写、执行规范和原则




## 03

**执行单测**

坚持执行和检查单测用例，提升单测通过率

## 04

**覆盖率检查**

通过不断的提高单测覆盖率，提高系统质量

联系作者，和我成为好友

# Thanks
# And Your Slogan Here.

尹洪亮 Kevin