# 基础知识

## **什么是单元测试？**

单元测试，是为了测试某一个类的某一个方法能否正常工作

public class NormalClassA {

public int testAdd(int a, int b) {

return a + b;

}}

public class NormalClassATest {

@Test

public void testAdd() {

NormalClassA classA = new NormalClassA();

Assert.assertEquals(5, classA.testAdd(1,4));

}

｝

#### **setup：**

一般是new出你要测试的那个类，以及其他一些前提条件的设置：NormalClassA classA = new NormalClassA();

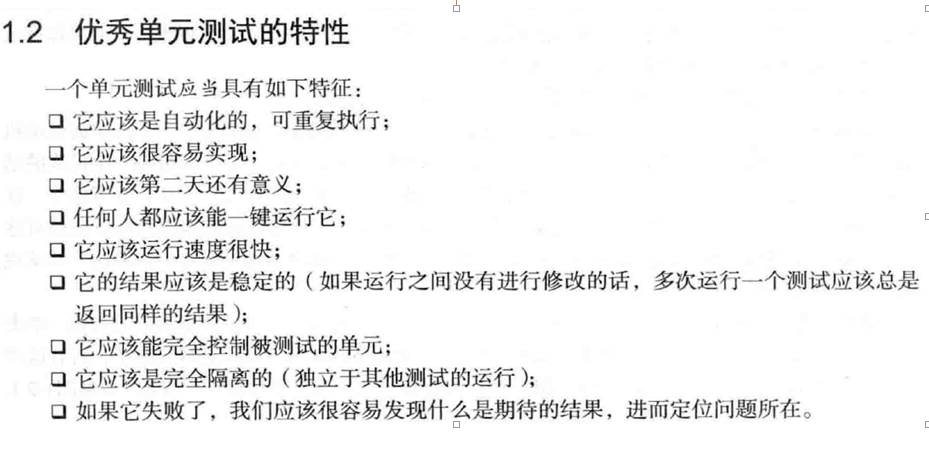
#### **执行操作：**

一般是调用你要测试的那个方法，获得运行结果：classA.testAdd(1,4)

#### **验证结果：**

验证得到的结果跟预期中是一样的：Assert.assertEquals(5, classA.testAdd(1,4));

#### **《单元测试的艺术》一书中的优秀的单元测试特性：**



## **2.单元测试目的？**

#### **1. 验收。提前发现代码中的bug。以后代码有改动或重构，跑下用例就可以了。**

#### **2. 帮助开发。开发过程中，可以快速验证自己的代码逻辑，不用一遍遍完整跑起来。提高效率。**

#### **3. 规范代码。为了让代码可测，会尽量避免些耦合特别多的代码。以及特别大的类和函数。**

## **3. 单元测试工具链**

#### **谷歌推荐的是：junit4 + mockito + robolectric**

#### **我们最终选择的工具链：junit4 + mockito + powermock + robolectric**

### **junit4**

现在另一个流行的是testng，这个框架更多用于服务器接口测试。用于安卓单测的比较少。  
再者junit4是谷歌官方推荐，跟robolectric以及其他安卓官方框架例如：AndroidJunitRunner等兼容性更好。  
所以选择junit4.

### **mock**

mockito功能有点弱，很多功能受限。  
powermock可以跟mockito兼容而且功能更强。与mockito比，支持修改和mock静态类或对象的私有方法/成员，还支持很多反射方法，非常实用。  
powermock目前使用人数非常多，github上的star很高 2450。

### **robolectric**

可以比较完整的模拟安卓接口。这样case就可以运行在jvm上。目前这个没有其他类似框架，无可替代，而且谷歌官方推荐。

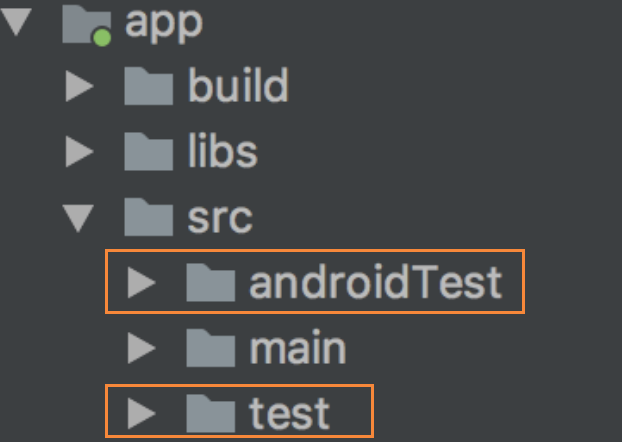
环境  
这套框架运行于JVM，不需要安卓模拟器和手机。

## **4. 如何在android project中运行单元测试**

#### **安卓自动化测试有两种：**

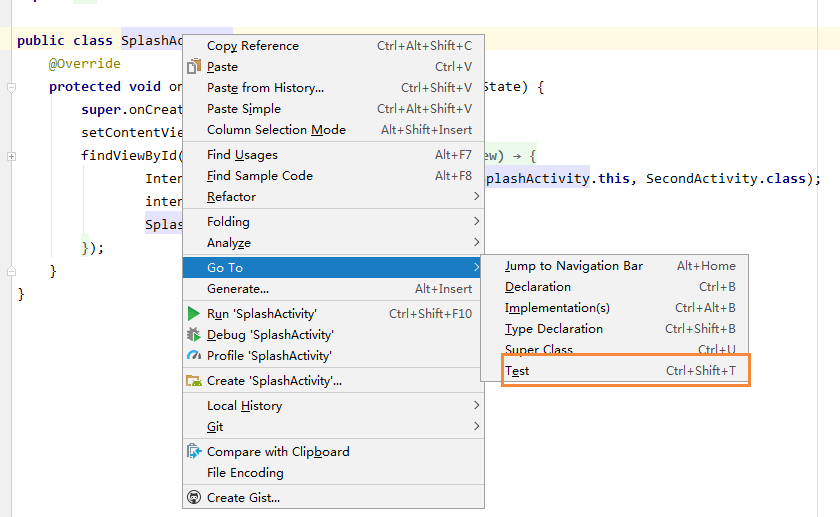
#### **Local tests:运行与JVM上**

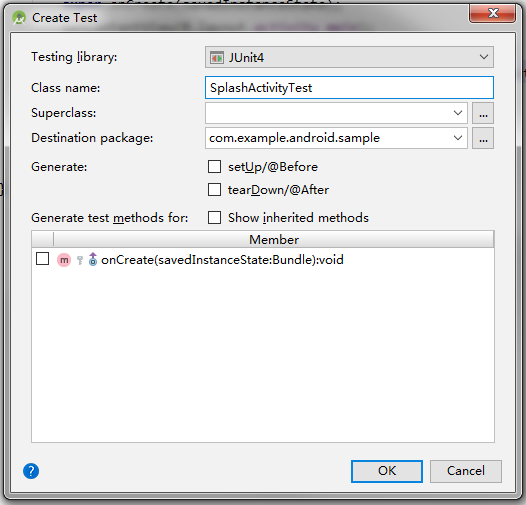
#### **Instrumented tests:运行于手机或模拟器，因为要依赖Android Framework，例如Context等。**

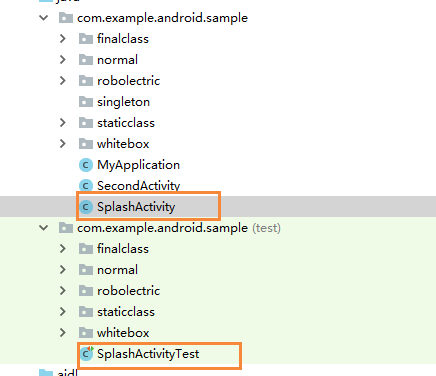


新建模块时会自动生成androidTest和test两个目录  
androidTest中的测试用例对应上面提到的 Instrumented tests  
test中的测试用例对应上面提到的Local tests

### **（1）为类创建一个测试**



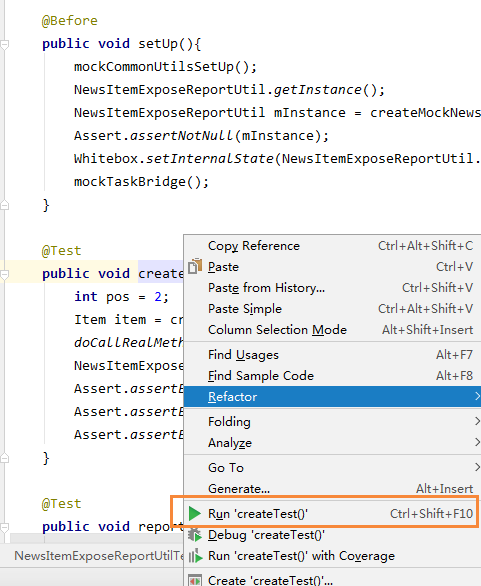




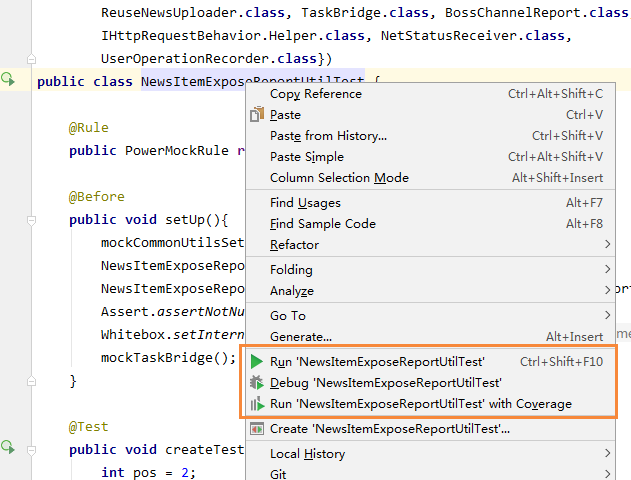
test的同级目录下就会生成测试类文件

### **（2）如何运行？**

#### **方法一：通过Android Studio运行**



函数内右键，运行单个测试用例



在类名上右键，运行该类的所有测试用例

#### **方法二：通过命令行**

在Terminal中运行：

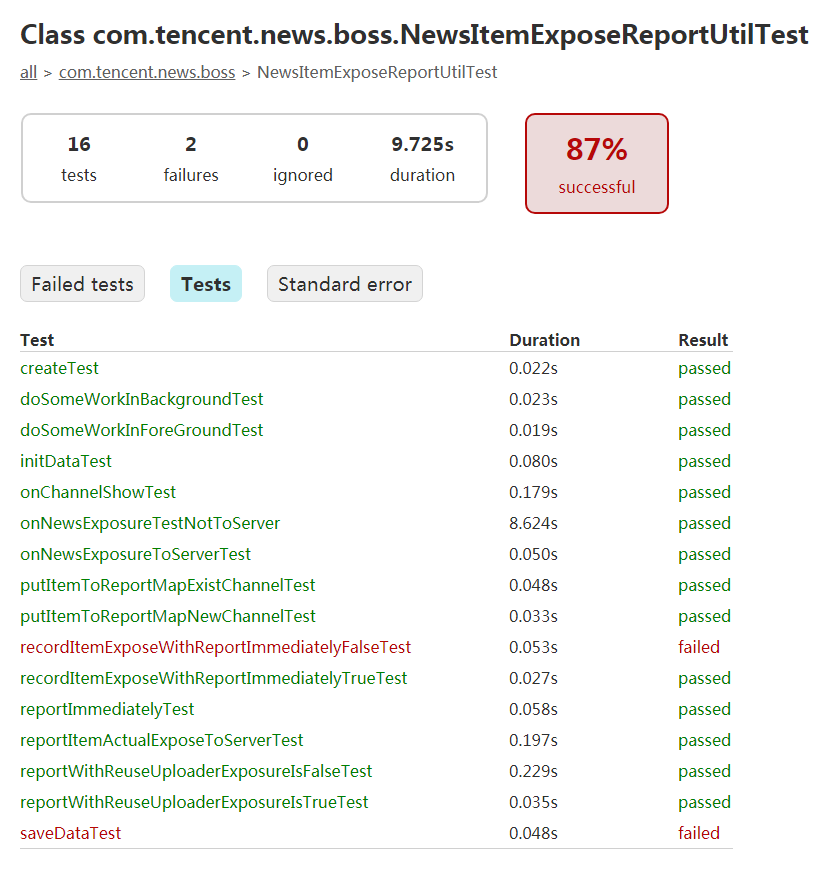
gradlew testDebug --continue

通过命令行运行后，会生成测试用例的报告，用Android Studio运行不会生成。

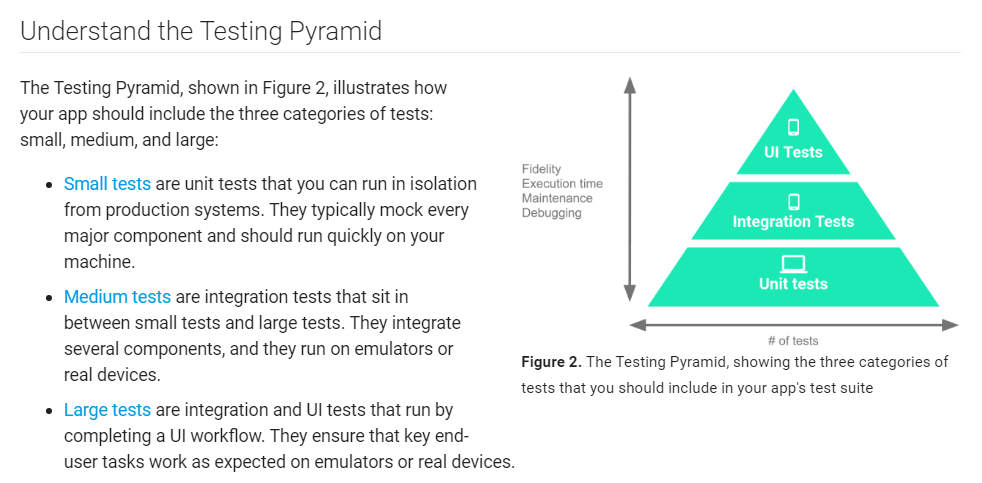
报告目录

对应模块的build目录下，build\reports\tests\testDebugUnitTest

以m\_news\_list模块为例：TencentNews\m\_news\_list\build\reports\tests\testDebugUnitTest



## **5. 测试分层**

  
<https://developer.android.com/training/testing/fundamentals>  
这个是安卓官方对测试金字塔的描述。

Small tests是单元测试，在local JVM就可以运行。

Medium tests是集成测试，测试集成的几个模块，运行于真机或模拟器。

Large tests是UI测试，测试整个UI工作流程，运行于真机或模拟器。

开发写的单元测试和测试写的集成测试区别在哪里？

单元测试测试的测试单元一定是远小于集成测试，主要原因是：单元测试是运行在JVM上，直接去测试某个类或某几个类，因为没有走整个App正常的启动和调用流程，导致非常多的模块是没有初始化的，为了测试用例不崩溃，必须把需要用到，但没初始化的类或方法mock，如果测试单元很大，需要mock的东西就非常多，很难保证不出错。而集成测试是运行在真机或模拟器，可以按照App正常的启动和调用流程，需要mock的东西不多，所以测试单元可以很大。

再者，单元测试是白盒测试。集成测试是黑盒测试。

## **6. 两种函数不同的测试方法**

函数有两种：验证返回值和验证函数调用。

### **（1）有返回值函数**

既可以验证返回值，也可以验证函数调用

### **（2）无返回值**

验证函数调用

public class NormalClassA {

public NormalClassB mClassB = new NormalClassB();

public int testPlus(int a, int b) {

return mClassB.testPlus(a, b);

}}

public class NormalClassB {

public int testPlus(int a, int b) {

return a + b;

}}

public class NormalClassATest {

@Test

public void testPlusInvokeMock() {

NormalClassA classA = new NormalClassA();

classA.mClassB = Mockito.mock(NormalClassB.class);

//验证返回值

Assert.assertEquals(6, classA.testPlus(3,3));

//验证函数调用，并且参数是3，3

Mockito.verify(classA.mClassB).testPlus(3, 3);

}

## **7 单元测试基本规范：**

### **（1）测试用例存放目录**

单元测试文件放在对应模块的test目录下

### **（2）单元测试代码规范：**

✓ 测试类的包名：与被测类的包名保持一致  
✓ 测试类命名方式：被测类名+Test。示例：  
被测类名AppDownload，则测试类名为  
AppDownloadTest  
✓ 测试用例命名方式：test+首字母大写的被  
测方法名，如果同一个逻辑要测不同分支，产生多个同类测试用例的情况，用例前缀需保持一致，后面跟“\_xxxx\_xxxx”，xxxx明确分支意图。格式为大写开头驼峰。。示例：  
被测方法名sendRequest，则测试用例名称  
为testSendRequest，如有多个，则为  
testSendRequest\_HasLogin\_Success, testSendRequest\_HasLogin\_Fail,…  
。  
用例命名规范参考谷歌代码：  
<https://cs.android.com/android/platform/superproject/+/master:packages/apps/WallpaperPicker2/tests/src/com/android/wallpaper/picker/individual/IndividualPickerActivityTest.java>  
✓ 测试类必须有setup()、cleanup()方法，用  
于初始化和资源释放。  
✓ 测试用例必须有断言。

### **（3）单元测试验证标准：**

通过代码覆盖率判断  
代码覆盖率包含：  
行覆盖，函数覆盖，分支覆盖。

### **（4）单元测试验证标准：**

通过代码覆盖率判断  
代码覆盖率包含：  
行覆盖，函数覆盖，分支覆盖。

### **（5）那些需要做单元测试：**

非UI的逻辑代码，例如：工具方法、MVP中的P。

### **（6）代码架构规范：**

保持UI和逻辑的分析，推荐采用MVP模式，方便单元测试。

# JUnit4

## **1. 引入依赖**

dependencies {

testCompile 'junit:junit:4.12'}

## **2. 使用Junit**

###（1）常用注解

public class NormalClassATest {

@Before

public void setup() {

}

@Test

public void testAdd() {

NormalClassA classA = new NormalClassA();

Assert.assertEquals(5, classA.testAdd(1,4));

}

@After

public void cleanup() {

}}

@Test代表一条测试用例。  
@Before修饰的方法，在每个测试方法调用之前，这个方法都会得到调用。所以上面的例子中，testAdd()被运行之前，setup()会被调用一次  
@After 对应于@Before，每个测试方法运行结束之后，会运行的方法。比如一个测试文件操作的类，那么在它的测试类中，可能@Before里面需要去打开一个文件，而每个测试方法运行结束之后，都需要去close这个文件。这个时候就可以把文件close的操作放在@After里面，让它自动去执行。  
@BeforeClass  
@AfterClass。@BeforeClass的作用是，在跑一个测试类的所有测试方法之前，会执行一次被@BeforeClass修饰的方法，执行完所有测试方法之后，会执行一遍被@AfterClass修饰的方法。这两个方法可以用来setup和release一些公共的资源，需要注意的是，被这两个annotation修饰的方法必须是静态的。

####执行顺序：

@BeforeClass –>

@Before –> @Test –> @After //case 1

@Before –> @Test –> @After //case 2

@Before –> @Test –> @After //case n

–> @AfterClass

这是比较常用的注解，其他注解可以参考下表：

| **注解名** | **含义** |
| --- | --- |
| @Test | 表示此方法为测试方法 |
| @Before | 在每个测试方法前执行，可做初始化操作 |
| @After | 在每个测试方法后执行，可做释放资源操作 |
| @Ignore | 忽略的测试方法 |
| @BeforeClass | 在类中所有方法前运行。此注解修饰的方法必须是static void |
| @AfterClass | 在类中最后运行。此注解修饰的方法必须是static void |
| @RunWith | 指定该测试类使用某个运行器 |
| @Parameters | 指定测试类的测试数据集合 |
| @Rule | 重新制定测试类中方法的行为 |
| @FixMethodOrder | 指定测试类中方法的执行顺序 |

###（2）Assert类

| **方法名** | **方法描述** |
| --- | --- |
| assertNotEquals | 断言传入的预期值与实际值是不相等的 |
| assertArrayEquals | 断言传入的预期数组与实际数组是相等的 |
| assertNull | 断言传入的对象是为空 |
| assertNotNull | 断言传入的对象是不为空 |
| assertTrue | 断言条件为真 |
| assertFalse | 断言条件为假 |
| assertSame | 断言两个对象引用同一个对象，相当于“==” |
| assertNotSame | 断言两个对象引用不同的对象，相当于“!=” |
| assertThat | 断言实际值是否满足指定的条件 |

####通过Assert类验证结果是否符合预期

###（3）assertThat验证结果

Assert.assertEquals(4, classA.testAdd(1,4));出错时的提示：

junit.framework.AssertionFailedError: Expected :4Actual :5

<Click to see difference>

assertThat(4, equalTo(5));出错时的提示：

java.lang.AssertionError: Expected: <5>

but: was <4>Expected :<5>

Actual :<4>

<Click to see difference>

####assertThat的作用跟Assert类似，提供的方法更多一些，返回错误也更容易读。

| **匹配器** | **说明 例子** |
| --- | --- |
| is | 断言参数等于后面给出的匹配表达式 |
| not | 断言参数不等于后面给出的匹配表达式 |
| equalTo | 断言参数相等 |
| equalToIgnoringCase | 断言字符串相等忽略大小写 |
| containsString | 断言字符串包含某字符串 |
| startsWith | 断言字符串以某字符串开始 |
| endsWith | 断言字符串以某字符串结束 |
| nullValue | 断言参数的值为null |
| notNullValue | 断言参数的值不为null |
| greaterThan | 断言参数大于 |
| lessThan | 断言参数小于 |
| greaterThanOrEqualTo | 断言参数大于等于 |
| lessThanOrEqualTo | 断言参数小于等于 |
| closeTo | 断言浮点型数在某一范围内 |
| allOf | 断言符合所有条件，相当于&& |
| anyOf | 断言符合某一条件，相当于或 |
| hasKey | 断言Map集合含有此键 |
| hasValue | 断言Map集合含有此值 |
| hasItem | 断言迭代对象含有此元素 |

# Mockito

## **1. 为什么要Mock？**

### **（1）. 解决测试类对其他类的依赖**

在实际的单元测试中，我们要测试的类很可能对其他类有依赖，要测试的方法对其他方法或变量有依赖，如果直接运行这个要测试的方法，很大可能会因为某些变量没初始化NP。这样我们就无法顺利的进行测试。如果mock方法模拟一个类，最大的特点是，该类所有的方法都是空，所有变量都是初始值。这样这个类就对其他类几乎没有依赖了。然后只针对你要测试的方法调用doCallRealMethod()真正执行方法的实现。这样依赖就很少了，就比较可控了。再把依赖的方法mock，方法运行了。

public class JudgePresenter extends AbstractJudgePresenter {

public JudgePresenter(IJudgeContract.IView mView, QuestionInfo mCurrentQuestionInfo) {

super(mView, mCurrentQuestionInfo);

}

@Override

public void initUIData() {

mView.setBg();

updateLiveViewerCount();

｝

private void updateLiveViewerCount() {

DataLoader.getLiveViewerBuilder(GlobalLocalData.getInstance().getArticleId())

.response(new TNResponseCallBack<LiveViewersRet>()).build().submit();

}

｝

针对initUIData写测试用例：

@Test

public void initUIDataTest() {

IJudgeContract.IView mockView = Mockito.mock(JudgeDialog.class);

JudgePresenter presenter = new JudgePresenter(mockView, getFakeQuestionInfo());

presenter.initUIData();

Mockito.verify(mockView).setBg();

}

直接运行测试用例，会报错，提示网络库某个方法NP了。  
这是因为initUIDataTest()调到了updateLiveViewerCount()，该函数中调用到了网络请求库，而网络库模块是需要初始化的，所以有些值为null，导致NP。那只要把网络库初始化一下，不就可以了吗？然后如果尝试去初始话网络库，就会发现，初始化网络库又需要一堆依赖…，这就把问题搞复杂了。回到最初的目的，我们只是想测试initUIDataTest方法，我们只关心setBg()被调用了，至于updateLiveViewerCount方法的执行过程，不属于我们这个测试用例考虑的范畴，  
所以这里我们其实只需要mock一下updateLiveViewerCount方法，让他不执行就可以。用PowerMockito来实现，代码如下

PowerMockito.suppress(PowerMockito.method(JudgePresenter.class, "updateLiveViewerCount"));

所以完整的测试代码：

@Test

public void initUIDataTest() {

IJudgeContract.IView mockView = Mockito.mock(JudgeDialog.class);

JudgePresenter presenter = new JudgePresenter(mockView, getFakeQuestionInfo());

PowerMockito.suppress(PowerMockito.method(JudgePresenter.class, "updateLiveViewerCount"));

presenter.initUIData();

Mockito.verify(mockView).setBg();

}

### **（2）. 验证方法的调用**

只有对mock的对象或类，才能调用verify方法，验证方法的调用。

举个例子：

@Test

public void initUIDataTest() {

IJudgeContract.IView mockView = Mockito.mock(JudgeDialog.class);

JudgePresenter presenter = new JudgePresenter(mockView, getFakeQuestionInfo());

PowerMockito.suppress(PowerMockito.method(JudgePresenter.class, "updateLiveViewerCount"));

presenter.initUIData();

Mockito.verify(mockView).setBg();

}

为了验证JudgeDialog中setBg()方法被调用了，需要mock一个JudgeDialog。

## **2. Mockito**

Mockito是安卓官方推荐的，也是目前使用最广泛的。

### **（1）用法**

dependencies {

testCompile "org.mockito:mockito-core:2.8.9"

}

@RunWith(MockitoJUnitRunner.class)public class NormalClassAMockitoTest {

@Test

public void testAPlusInvokeMock() {

NormalClassA mockClassA = Mockito.mock(NormalClassA.class);

mockClassA.testPlus(3,3);

Mockito.verify(mockClassA).testPlus(3, 3);

}

｝

这个例子中，mock了NormalClassA用于验证该对象的testPlus()方法是否被调用了。  
注意类前面的注解[@RunWith](http://km.oa.com/user/RunWith" \o "@RunWith" \t "https://km.woa.com/group/35228/articles/show/_blank)(MockitoJUnitRunner.class)  
指明本类中的用例用MockitoJUnitRunner运行。

### **（2）mock和spy的不同**

#### **mock方法创建的对象，所有方法都是空方法。非void方法都将返回默认值，比如int方法将返回0，对象方法将返回null等，而void方法将什么都不做。**

#### **spy方法创建的对象，跟正常类对象一样，是正常对象的替身。**

#### **被测试代码：**

public class NormalClassB {

public int testPlus(int a, int b) {

return a + b;

}

｝

#### **用mock方式的代码：**

@Test

public void testPlusValueMock() {

NormalClassA classA = new NormalClassA();

classA.mClassB = Mockito.mock(NormalClassB.class);

Assert.assertEquals(0, classA.testPlus(3, 3));

}

这个例子中mock了NormalClassB，所以他的testPlus方法就会是空。调用 classA.testPlus(3, 3)后，将会返回int的默认值0，并不会返回6。

#### **用spy方式的代码：**

@Test

public void testPlusValueSpy() {

NormalClassA classA = new NormalClassA();

classA.mClassB = Mockito.spy(new NormalClassB());

Assert.assertEquals(6, classA.testPlus(3, 3));

}

这个例子中spy了NormalClassB，他的testPlus方法不为空，是具体实现的。调用 classA.testPlus(3, 3)后，将会返回6。

### **（3）mock和spy各自应用场景**

#### **Mock**

mock适合场景，类中的大多数方法，只想验证方法是否被调用，不关心方法内的执行结果。让类的所有方法都为空实现，这样可以减少对其他类的依赖。

如果想要mock对象中的某个函数执行具体的实现，该如何呢?  
使用doCallRealMethod()方法

@Test

public void testPlusValueMockWithCallRealMethod() {

NormalClassA classA = new NormalClassA();

classA.mClassB = Mockito.mock(NormalClassB.class);

doCallRealMethod().when(classA.mClassB).testPlus(Mockito.anyInt(), Mockito.anyInt());

Assert.assertEquals(6, classA.testPlus(3, 3));

}

#### **Spy**

spy跟mock相反，适合类中大部分函数需要关心具体实现。

如果想要spy对象中，某个函数不执行，该如何呢？  
分两种情况：有返回值和无返回值

#### **有返回值的函数：**

@Test

public void testPlusValueSpyWithDoReturn() {

NormalClassA classA = new NormalClassA();

classA.mClassB = Mockito.spy(NormalClassB.class);

doReturn(0).when(classA.mClassB).testPlus(Mockito.anyInt(), Mockito.anyInt());

Assert.assertEquals(0, classA.testPlus(3, 3));

}

#### **无返回值的函数：**

@Test

public void testPlusValueSpyWithDoThrow() throws Exception {

NormalClassA classA = new NormalClassA();

classA.mClassB = Mockito.spy(NormalClassB.class);

doNothing().when(classA.mClassB).voidMethod(Mockito.anyInt(), Mockito.anyInt());

}

这是Mockito的方法，后面介绍了Powemock后，还有其他方法可以让某个函数不执行。

### **（4）打桩方法**

doCallRealMethod()，doReturn()，doNothing()就是我们比较常用的打桩方法。

全部打桩方法如下：

| **方法名** | **方法描述** |
| --- | --- |
| thenReturn(T value) | 设置要返回的值 |
| thenThrow(Throwable… throwables) | 设置要抛出的异常 |
| thenAnswer(Answer<?> answer) | 对结果进行拦截 |
| doReturn(Object toBeReturned) | 提前设置要返回的值 |
| doThrow(Throwable… toBeThrown) | 提前设置要抛出的异常 |
| doAnswer(Answer answer) | 提前对结果进行拦截 |
| doCallRealMethod() | 调用某一个方法的真实实现 |
| doNothing() | 设置void方法什么也不做 |

### **（5）验证方法**

Mockito.verify()用于验证方法是否调用，调用次数等。

@Test

public void testAPlusInvokeMock() {

NormalClassA mockClassA = Mockito.mock(NormalClassA.class);

mockClassA.testPlus(3,3);

Mockito.verify(mockClassA).testPlus(3, 3);

Mockito.verify(mockClassA, times(1)).testPlus(3, 3);

Mockito.verify(mockClassA, atLeast(1)).testPlus(3, 3);

}

| **方法名** | **方法描述** |
| --- | --- |
| after(long millis) | 在给定的时间后进行验证 |
| timeout(long millis) | 验证方法执行是否超时 |
| atLeast(int minNumberOfInvocations) | 至少进行n次验证 |
| atMost(int maxNumberOfInvocations) | 至多进行n次验证 |
| description(String description) | 验证失败时输出的内容 |
| times(int wantedNumberOfInvocations) | 验证调用方法的次数 |
| never() | 验证交互没有发生,相当于times(0) |
| only() | 验证方法只被调用一次，相当于times(1) |

### **（6）参数匹配器**

@RunWith(MockitoJUnitRunner.class)public class NormalClassAMockitoTest {

@Test

public void testAPlusInvokeMock() {

NormalClassA mockClassA = Mockito.mock(NormalClassA.class);

mockClassA.testPlus(3,3);

Mockito.verify(mockClassA).testPlus(3, 3);

}

｝

Mockito.verify(mockClassA).testPlus(3, 3);  
这行验证成立条件有2个：

1. testPlus被调用。
2. testPlus的参数是3，3。

如果调用  
mockClassA.testPlus(3,4);  
验证是失败，因为参数不一致。

那么如何忽略具体的参数呢。这就需要用到参数匹配器。

Mockito.verify(mockClassA).testPlus(Mockito.anyInt(),Mockito.anyInt());

这样写，只要参数是整型就能验证成功。

#### **需要注意的特殊case**

一个参数确定，另一个参数是匹配器

Mockito.verify(mockClassA).testPlus(3 ,Mockito.anyInt());

如果这样写，能配置成功吗？  
答案是不能。  
正确写法：

Mockito.verify(mockClassA).testPlus(Mockito.eq(3) ,Mockito.anyInt());

匹配非基础类型  
如果函数原型是这样：

void putItemToReportMap(String channel, Properties prop, HashMap<String, HashSet<Properties>> map)

正确的验证方法：

Mockito.verify(mockClassA).putItemToReportMap(Mockito.anyString(), Mockito.any(Properties.class), Mockito.anyMap());

| **方法名** | **方法描述** |
| --- | --- |
| anyObject() | 匹配任何对象 |
| any(Class<T> type) | 与anyObject()一样 |
| any() | 与anyObject()一样 |
| anyBoolean() | 匹配任何boolean和非空Boolean |
| anyByte() | 匹配任何byte和非空Byte |
| anyCollection() | 匹配任何非空Collection |
| anyDouble() | 匹配任何double和非空Double |
| anyFloat() | 匹配任何float和非空Float |
| anyInt() | 匹配任何int和非空Integer |
| anyList() | 匹配任何非空List |
| anyLong() | 匹配任何long和非空Long |
| anyMap() | 匹配任何非空Map |
| anyString() | 匹配任何非空String |
| contains(String substring) | 参数包含给定的substring字符串 |
| argThat(ArgumentMatcher <T> matcher) | 创建自定义的参数匹配模式 |

### **（7）其他方法**

| **方法名** | **方法描述** |
| --- | --- |
| reset(T … mocks) | 重置Mock |
| spy(Class<T> classToSpy) | 实现调用真实对象的实现 |
| inOrder(Object… mocks) | 验证执行顺序 |
| [@InjectMocks](http://km.oa.com/user/InjectMocks" \o "@InjectMocks" \t "https://km.woa.com/group/35228/articles/show/_blank)注解 | 自动将模拟对象注入到被测试对象中 |

## **3. Mockito局限性**

public class JudgePresenter extends AbstractJudgePresenter {

public JudgePresenter(IJudgeContract.IView mView, QuestionInfo mCurrentQuestionInfo) {

super(mView, mCurrentQuestionInfo);

}

@Override

public void initUIData() {

mView.setBg();

updateLiveViewerCount();

｝

private void updateLiveViewerCount() {

DataLoader.getLiveViewerBuilder(GlobalLocalData.getInstance().getArticleId())

.response(new TNResponseCallBack<LiveViewersRet>()).build().submit();

}

｝

针对上面代码，测试initUIData()方法。  
需要验证mView.setBg()、updateLiveViewerCount被调用

验证updateLiveViewerCount的代码可能会这样写：

JudgePresenter mockPresenter = Mockito.mock(JudgePresenter.class);

Mockito.verfiy(mockPresenter).updateLiveViewerCount();

编辑器会提示第二行代码错误。updateLiveViewerCount是private，无法直接调用到。

这是Mockito的局限性之一，不能访问private方法。  
这个影响是很大的，因为我们非常多函数都是private的。

除此之外，Mockito框架不支持mock匿名类、final类、static方法。

#### **当你开始针对真实工程代码写单元测试时会发现，在Mockito的这些限制下，几乎没法写测试用例。**

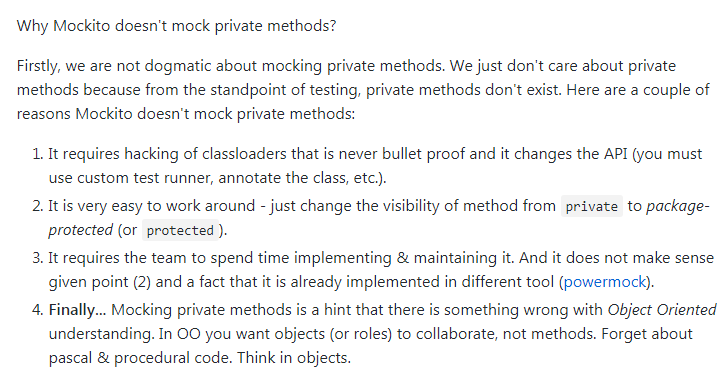
PowerMock是跟Mockito兼容的，功能更前大的mock框架，下篇文章会介绍~

# PowerMock

## **1. PowerMock介绍**

PowerMock是Mockito的扩展增强版，支持mock private、static、final方法和类，还增加了很多反射方法可以方便修改静态和非静态成员等。功能比Mockito增加很多。

为什么 Mockito 不自己支持 private 方法？



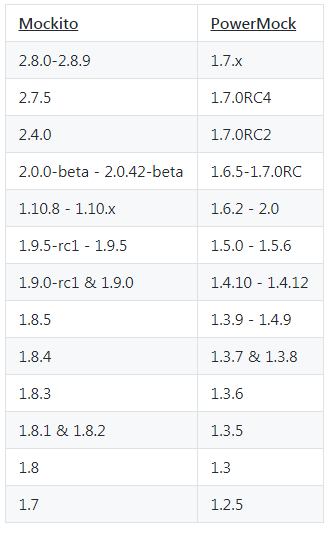
[ 请在这里填写图片描述 ]

[https://github.com/mockito/mockito/wiki/Mockito-And-Private-Methods](https://github.com/mockito/mockito/wiki/Mockito-And-Private-Methods" \t "https://km.woa.com/group/35228/articles/show/_blank)

简单点说：1. 改动较大，不是特别必要。2. powermock已经实现了。3.被测代码不够规范

## **2. PowerMock版本**

PowerMock是依赖Mockito的，所以使用时要一同引入。  
PowerMock跟Mockito的版本必须对应：  
[https://github.com/powermock/powermock/wiki/Mockito](https://github.com/powermock/powermock/wiki/Mockito" \t "https://km.woa.com/group/35228/articles/show/_blank)



[ 请在这里填写图片描述 ]

目前我们使用的版本：

testCompile 'junit:junit:4.12'

testCompile "org.mockito:mockito-core:2.8.9"

testCompile "org.powermock:powermock-module-junit4:1.7.1"

testCompile "org.powermock:powermock-api-mockito2:1.7.1" //注意，这里要用mockito2，否则报错

testCompile 'org.powermock:powermock-core:1.7.1'

testCompile "org.powermock:powermock-module-junit4-rule:1.7.1"

testCompile "org.powermock:powermock-classloading-xstream:1.7.1"

## **3. 基本使用**

@RunWith(PowerMockRunner.class)@PrepareForTest({NormalClassA.class})public class NormalClassAPowerMockTest {

@Test

public void testPrivateMethod() throws Exception {

NormalClassA classA = new NormalClassA();

NormalClassA mockClassA = PowerMockito.spy(classA);

mockClassA.publicAdd(2, 3);

PowerMockito.verifyPrivate(mockClassA, times(1)).invoke("privateAdd", Mockito.anyInt(), Mockito.anyInt());

}

｝

#### **1. [@RunWith](http://km.oa.com/user/RunWith" \o "@RunWith" \t "https://km.woa.com/group/35228/articles/show/_blank)(PowerMockRunner.class)，使用PowerMockRunner，使用这个Runner时，跟Mockito是兼容的，Mockito中的各种方法也是可以正常使用的。**

#### **2. [@PrepareForTest](http://km.oa.com/user/PrepareForTest" \o "@PrepareForTest" \t "https://km.woa.com/group/35228/articles/show/_blank)({NormalClassA.class})， PowerMock要mock的类一定要写在这个注解里，切记切记！**

#### **写的时候实在太容易忘了，而且，就算不写也不会报错，只是verify永远成功，所以特别隐蔽，忘记写了也很难发现。**

#### **3. 通过PowerMockito.verifyPrivate验证私有方法的调用。Mocktio的次数验证times(1)和参数匹配符也都是可以用的。**

## **4. 针对对象的操作**

### **（1）读取对象私有成员**

public class ObjectWhiteBoxClass {

private final List<String> mObjectList = new ArrayList<>();

public void publicObjectAdd(String str) {

mObjectList.add(str);

}

public List<String> getObjectList() {

return mObjectList;

}

}

@Test

public void testGetInternalState() {

ObjectWhiteBoxClass objectWhiteBoxClass = new ObjectWhiteBoxClass();

objectWhiteBoxClass.publicObjectAdd("test1str");

objectWhiteBoxClass.publicObjectAdd("test2str");

objectWhiteBoxClass.publicObjectAdd("test3str");

List<String> list = Whitebox.getInternalState(objectWhiteBoxClass, "mObjectList");

Assert.assertEquals(3, list.size());

Assert.assertEquals("test3str", list.get(list.size()-1));

}

### **（2）修改对象私有成员**

public class MockPrivateObjectClass {

private InnerClass innerClass;

public MockPrivateObjectClass() {

}

public int doPlus(int a, int b) {

return innerClass.plus(a, b);

}

public static class InnerClass {

public int plus(int a, int b) {

return a + b;

}

}

}

#### **第一种方式：Whitebox.setInternalState**

@RunWith(PowerMockRunner.class)@PrepareForTest({MockPrivateObjectClass.class})public class MockPrivateObjectClassTest {

@Test

public void doPlusTest() {

MockPrivateObjectClass objectClass = new MockPrivateObjectClass();

MockPrivateObjectClass.InnerClass mockInnerClass = Mockito.spy(MockPrivateObjectClass.InnerClass.class);

Whitebox.setInternalState(objectClass, "innerClass", mockInnerClass);

int ret = objectClass.doPlus(2, 3);

Assert.assertEquals(5, ret);

Mockito.verify(mockInnerClass).plus(2, 3);

}

｝

#### **第二种方式：MemberModifier.field**

@RunWith(PowerMockRunner.class)@PrepareForTest({MockPrivateObjectClass.class})public class MockPrivateObjectClassTest {

@Test

public void doPlusMethod2Test() throws Exception{

MockPrivateObjectClass objectClass = new MockPrivateObjectClass();

MockPrivateObjectClass.InnerClass mockInnerClass = Mockito.spy(MockPrivateObjectClass.InnerClass.class);

MemberModifier.field(MockPrivateObjectClass.class, "innerClass").set(objectClass, mockInnerClass);

int ret = objectClass.doPlus(2, 3);

Assert.assertEquals(5, ret);

Mockito.verify(mockInnerClass).plus(2, 3);

}

｝

### **（3）Verify对象私有方法**

public class NormalClassA {

public int publicAdd(int a, int b) {

return privateAdd(a, b);

}

private int privateAdd(int a, int b) {

return a + b;

}

}

@RunWith(PowerMockRunner.class)@PrepareForTest({NormalClassA.class})public class NormalClassAPowerMockTest {

@Test

public void testPrivateMethod() throws Exception {

NormalClassA classA = new NormalClassA();

NormalClassA mockClassA = PowerMockito.spy(classA);

mockClassA.publicAdd(2, 3);

PowerMockito.verifyPrivate(mockClassA, times(1)).invoke("privateAdd", Mockito.anyInt(), Mockito.anyInt());

}

｝

### **（4）Invoke对象私有方法**

public class ObjectWhiteBoxClass {

private final List<String> mObjectList = new ArrayList<>();

private void privateObjectAdd(String str) {

mObjectList.add(str);

}

private void privateObjectRemove(String str) {

mObjectList.remove(str);

}

public List<String> getObjectList() {

return mObjectList;

}

}

@Test

public void testInvokeMethod() throws Exception{

ObjectWhiteBoxClass objectWhiteBoxClass = new ObjectWhiteBoxClass();

Whitebox.invokeMethod(objectWhiteBoxClass, "privateObjectAdd", "test1");

Whitebox.invokeMethod(objectWhiteBoxClass, "privateObjectAdd", "test2");

Assert.assertEquals(2, objectWhiteBoxClass.getObjectList().size());

Whitebox.invokeMethod(objectWhiteBoxClass, "privateObjectRemove", "test1");

Assert.assertEquals(1, objectWhiteBoxClass.getObjectList().size());

}

### **（5）修改对象私有函数**

public class MockPrivateObjectClass {

public String stepName;

public MockPrivateObjectClass() {

}

private void setStepName() {

System.out.print("enter setStepName");

stepName = "has set name";

}

public void testStepName() {

setStepName();

System.out.print(stepName);

}

}

@Test

public void replacePrivateMethodTest() {

MockPrivateObjectClass objectClass = new MockPrivateObjectClass();

PowerMockito.replace(PowerMockito.method(MockPrivateObjectClass.class, "setStepName")).with(new InvocationHandler() {

@Override

public Object invoke(Object o, Method method, Object[] objects) throws Throwable {

Whitebox.setInternalState(o, "stepName", "modify step name");

return null;

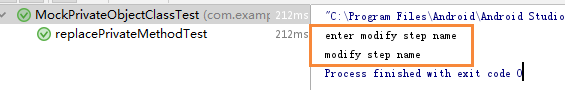
}

});

objectClass.testStepName();

}

运行结果：



[ 请在这里填写图片描述 ]

替换了setStepName方法的实现。

#### **这在写单测时，特别有用，譬如，将异步执行的代码，替换为同步，可以方便测试。**

### **（6） 调用私有构造方法**

public class PrivateConstructorClass {

private final List<String> mObjectList;

private PrivateConstructorClass() {

mObjectList = new ArrayList<>();

}

public void add(String str) {

mObjectList.add(str);

}

public List<String> getObjectList() {

return mObjectList;

}

}

@Test

public void testInvokePrivateConstructor() throws Exception{

PrivateConstructorClass object = Whitebox.invokeConstructor(PrivateConstructorClass.class);

object.add("test1");

Assert.assertEquals(1, object.getObjectList().size());

}

调用了私有的构造函数PrivateConstructorClass()

#### **这个对单例等特别有用**

### **（7） mock对象的创建**

public class NormalClassA {

public String getClassBName() {

NormalClassB b = new NormalClassB();

return b.getName();

}

}

public class NormalClassB {

public String getName(){

return "NormalClassB";

}

}

public static class MockNormalClassB extends NormalClassB {

@Override

public String getName() {

return "MockNormalClassB";

}

}

@Test

public void test

ClassB() throws Exception {

NormalClassA classA = new NormalClassA();

String name = classA.getClassBName();

System.out.println("before mock:" + name);

Assert.assertEquals("NormalClassB", name);

NormalClassB b = new MockNormalClassB();

NormalClassB mockB = Mockito.spy(b);

PowerMockito.whenNew(NormalClassB.class).withNoArguments().thenReturn(mockB);

name = classA.getClassBName();

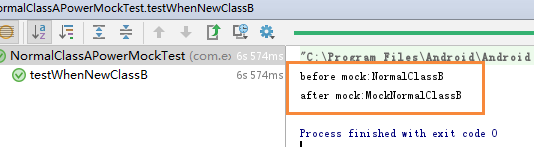
System.out.println("after mock:" + name);

Assert.assertEquals("MockNormalClassB", name);

Mockito.verify(mockB).getName();

}

运行结果：



[ 请在这里填写图片描述 ]

#### **new NormalClassB()动作是在函数内部，并没有赋给某个类成员，我们没法访问到NormalClassB对象，也就没法mock，这种情况特别适合用WhenNew方法。**

#### **注意：whenNew跟匿名类一起使用时，很可能会有问题，本文最后的总结中有介绍。**

### **（8）跳过某个方法**

Mockito中介绍过，通过doReutrn、doNothing，不执行某个函数。

PowerMock中有另一种方式：

public class MockPrivateObjectClass {

public String stepName;

private void setStepName() {

System.out.print("enter setStepName");

stepName = "has set name";

}

public void testStepName() {

setStepName();

System.out.print(stepName);

}

}

@Test

public void skipPrivateMethodTest() {

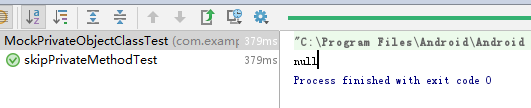
MockPrivateObjectClass objectClass = new MockPrivateObjectClass();

PowerMockito.suppress(PowerMockito.method(MockPrivateObjectClass.class, "setStepName"));

objectClass.testStepName();

}

执行结果



[ 请在这里填写图片描述 ]

enter setStepName log没有被打印，说明方法被跳过了。

### **（9）打桩对象私有方法**

public class H5JsApiScriptInterface {

private boolean shouldTriggerForAccount(int account) {

return !isAccountValid(account) || !isMainAvailable();

}

}

测试用例

@Test//注意 要增加PrepareForTest@PrepareForTest({LoginManager.class, H5JsApiScriptInterface.class})public class H5JsApiScriptInterfaceTest extends MainTNUnitTest {

//qq connect

@Test

public void testShowLoginWithType\_1() throws Exception{

PowerMockito.mockStatic(LoginManager.class);

H5JsApiScriptInterface mockScriptInterface = PowerMockito.mock(H5JsApiScriptInterface.class);

Mockito.doCallRealMethod().when(mockScriptInterface).showLoginWithType(Mockito.anyString(), Mockito.anyString());// 打桩对象私有方法，mock对象私有方法//注意 要增加PrepareForTest

PowerMockito.when(mockScriptInterface, PowerMockito.method(H5JsApiScriptInterface.class, "shouldTriggerForAccount", int.class)).withArguments(Mockito.anyInt()).thenReturn(true);

mockScriptInterface.showLoginWithType("qqconnect", "callback");

PowerMockito.verifyStatic(LoginManager.class, Mockito.times(1));

LoginManager.triggerLogin(Mockito.eq(LOGIN\_FROM\_WEB\_ONLY\_QQCONNECT), Mockito.any(Subscriber.class));

}

}

## **5. 针对静态类**

### **（1）访问静态私有成员**

public class StaticWhiteBoxClass {

private final static List<String> mStaticList = new ArrayList<>();

public static void publicStaticAdd(String str) {

mStaticList.add(str);

}

public static List<String> getStaticList() {

return mStaticList;

}

}

@Test

public void testGetInternalState() {

StaticWhiteBoxClass.publicStaticAdd("test1str");

StaticWhiteBoxClass.publicStaticAdd("test2str");

StaticWhiteBoxClass.publicStaticAdd("test3str");

List<String> list = Whitebox.getInternalState(StaticWhiteBoxClass.class, "mStaticList");

Assert.assertEquals(3, list.size());

Assert.assertEquals("test3str", list.get(list.size()-1));

}

### **（2）修改静态私有成员**

public class StaticWhiteBoxClass {

private final static List<String> mStaticList = new ArrayList<>();

public static void publicStaticAdd(String str) {

mStaticList.add(str);

}

public static List<String> getStaticList() {

return mStaticList;

}

}

@Test

public void testSetInternalState() {

List<String> list = new ArrayList<>();

list.add("outer1");

list.add("outer2");

list.add("outer3");

Whitebox.setInternalState(StaticWhiteBoxClass.class, "mStaticList", list);

List<String> objList = StaticWhiteBoxClass.getStaticList();

Assert.assertEquals(3, objList.size());

Assert.assertEquals("outer3", list.get(objList.size()-1));

}

### **（3）调用静态私有方法**

public class StaticWhiteBoxClass {

private final static List<String> mStaticList = new ArrayList<>();

private static void privateStaticAdd(String str) {

mStaticList.add(str);

}

public static List<String> getStaticList() {

return mStaticList;

}

}

@Test

public void testInvokeMethod() throws Exception{

Whitebox.invokeMethod(StaticWhiteBoxClass.class, "privateStaticAdd", "test1");

Whitebox.invokeMethod(StaticWhiteBoxClass.class, "privateStaticAdd", "test2");

Assert.assertEquals(2, StaticWhiteBoxClass.getStaticList().size());

}

### **（4）替换静态函数**

public class StaticWhiteBoxClass {

public static void originalMethod(String name) {

System.out.println("enter originalMethod name:" + name);

}

}

@Test

public void replacePrivateMethodTest() {

PowerMockito.replace(PowerMockito.method(StaticWhiteBoxClass.class, "originalMethod")).with(new InvocationHandler() {

@Override

public Object invoke(Object o, Method method, Object[] objects) throws Throwable {

System.out.println("enter modify method param:" + objects[0]);

return null;

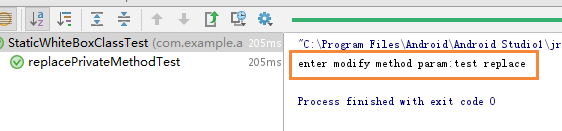
}

});

StaticWhiteBoxClass.originalMethod("test replace");

}

运行结果：



[ 请在这里填写图片描述 ]

### **（5）打桩静态Public方法**

public class StaticClass {

public static String ask(){

return "I'm Static class";

}

}

@Test

public void should\_mock\_static\_method() {

final String value = "10";

mockStatic(StaticClass.class);

when(StaticClass.ask()).thenReturn(value);

Assert.assertNotEquals("I'm Static class", StaticClass.ask());

}

#### **静态方法mock，一定要调用mockStatic(StaticClass.class);**

#### **when(StaticClass.ask()).thenReturn(value);PowerMock的这个语句跟Mockito是很像的，基本上Mockito中的的那些打桩方法在PowerMock都有对应的。**

### **（6）打桩静态private方法**

这种public或private的都可以用

public class H5JsApiScriptInterface {

// 静态私有方法

private static boolean isQQorWxAvailable() {

return true;

}

}

@PrepareForTest({LoginManager.class, H5JsApiScriptInterface.class})public class H5JsApiScriptInterfaceTest extends MainTNUnitTest { //qq and wx

@Test

public void testShowLoginWithType\_4() throws Exception{

PowerMockito.mockStatic(LoginManager.class);

PowerMockito.mockStatic(H5JsApiScriptInterface.class);

H5JsApiScriptInterface mockScriptInterface = PowerMockito.mock(H5JsApiScriptInterface.class);

Mockito.doCallRealMethod().when(mockScriptInterface).showLoginWithType(Mockito.anyString(), Mockito.anyString());

//打桩静态私有方法，mock静态私有方法

//注意增加@PrepareForTest

PowerMockito.doReturn(false).when(H5JsApiScriptInterface.class, "isQQorWxAvailable");

mockScriptInterface.showLoginWithType("qqorweixin", "callback");

PowerMockito.verifyStatic(LoginManager.class, Mockito.times(1));

LoginManager.triggerLogin(Mockito.eq(LOGIN\_FROM\_WEB\_BOTH\_QQ\_AND\_WX), Mockito.any(Subscriber.class));

}

}

### **（7）thenCallRealMethod方法**

public class StaticClass {

public static int add(int a, int b) {

return a + b;

}

public static int sum() {

int ret = 0;

for (int i = 0; i < 10; i++) {

ret = add(i, i+1);

}

return ret;

}

｝

@Test

public void call\_mock\_sum\_static\_method() {

mockStatic(StaticClass.class);

//如果不用spy，就用thenCallRealMethod也可以

when(StaticClass.sum()).thenCallRealMethod();

when(StaticClass.add(Mockito.anyInt(),Mockito.anyInt())).thenCallRealMethod();

int sum = StaticClass.sum();

Assert.assertEquals(19, sum);

}

### **（8）spy静态方法**

public class StaticClass {

public static int add(int a, int b) {

return a + b;

}

public static int sum() {

int ret = 0;

for (int i = 0; i < 10; i++) {

ret = add(i, i+1);

}

return ret;

}

｝

@Test

public void call\_spy\_sum\_static\_method() {

mockStatic(StaticClass.class);

//mock的话，函数都是空的切记，如果需要用到函数逻辑，要用spy

spy(StaticClass.class);

int sum = StaticClass.sum();

Assert.assertEquals(19, sum);

}

### **（9）Verify Public 静态方法**

public class StaticClass {

public static int add(int a, int b) {

return a + b;

}

public static int sum() {

int ret = 0;

for (int i = 0; i < 10; i++) {

ret = add(i, i+1);

}

return ret;

}

｝

@Test

public void call\_spy\_sum\_static\_method() {

mockStatic(StaticClass.class);

//mock的话，函数都是空的切记，如果需要用到函数逻辑，要用spy

spy(StaticClass.class);

int sum = StaticClass.sum();

Assert.assertEquals(19, sum);

verifyStatic(StaticClass.class, Mockito.times(10)); //验证调用了10次

StaticClass.add(Mockito.anyInt(),Mockito.anyInt()); //验证的是add方法

}

### **（10）Verify Private 静态方法**

public class StaticClass {

public static int dec(int total) {

int ret = total;

for (int i = 0; i < 10; i++) {

ret = sub(ret, i);

}

return ret;

}

private static int sub(int a, int b) {

return a - b;

}

}

@Test

public void call\_private\_dec\_static\_method() throws Exception {

mockStatic(StaticClass.class);

spy(StaticClass.class);

int ret = StaticClass.dec(100);

Assert.assertEquals(55, ret);

verifyPrivate(StaticClass.class, Mockito.atLeastOnce()).invoke("sub", Mockito.anyInt(), Mockito.anyInt());

}

### **（11）跳过静态方法**

public class StaticClass {

public static String mStepName;

public static void testStepName() {

setStepName();

}

private static void setStepName() {

System.out.println("enter setStepName");

mStepName = "setStepName";

}

}

@Test

public void skipStaticMethodTest() {

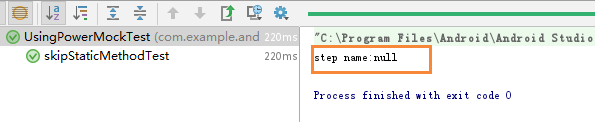
PowerMockito.suppress(PowerMockito.method(StaticClass.class, "setStepName"));

StaticClass.testStepName();

System.out.println("step name:" + StaticClass.mStepName);

}

运行结果：



[ 请在这里填写图片描述 ]

enter setStepName log没有打印，说明已经跳过了。  
无论函数是否有返回值，都可以用这中方法跳过该函数。

## **6. Final类**

public final class FinalClassA {

public final String finalMethod() {

return "finalMethod";

}

}

@Test

public final void mockPowerFinalClassTest() {

FinalClassA instance = new FinalClassA();

FinalClassA mock = PowerMockito.mock(FinalClassA.class);

Mockito.when(mock.finalMethod()).thenReturn("mock method");

Assert.assertNotEquals(instance.finalMethod(), mock.finalMethod());

}

上里面例子演示了，通过PowerMock可以mock Final类。

## **总结**

上面列出了PowerMoc差不多所有会使用到的场景下的使用范例，通过这些场景范例基本上可以解决我们单元测试中所有的需要mock的情况。

### **目前能想到的，PowerMock解决不了的是匿名类的场景：**

public class NormalClassA {

public void setAbsCallback() {

AbsCallback callback = new AbsCallback() {

@Override

void call() {

System.out.println("impl AbsCallback");

}

};

callback.call();

}

public static class AbsCallback {

public AbsCallback() {

}

void call(){

System.out.println("impl default AbsCallback");

};

}

public static class MockAbsCallback extends NormalClassA.AbsCallback {

@Override

void call() {

System.out.println("mock impl AbsCallback");

}

}

}

如果想mock setAbsCallback函数中的new AbsCallback()，这种匿名类，目前是无法用WhenNew mock的。

还有在匿名类中new 对象A，用whenNew来mock对象A也是不行的。

见这里的例子，同时解释了具体原因  
[https://stackoverflow.com/questions/7523389/how-do-i-use-powermockito-to-mock-the-construction-of-new-objects-when-testing-a](https://stackoverflow.com/questions/7523389/how-do-i-use-powermockito-to-mock-the-construction-of-new-objects-when-testing-a" \t "https://km.woa.com/group/35228/articles/show/_blank)