Android单元测试参考

版本	说明	作者
v1.0.0	Android单元测试参考	李应锋
v1.1.0	Mockito、PowerMock单元测试实践	饶煦庭、董建慧、仵闯

单元测试是应用程序测试策略中的基本测试，通过对代码进行单元测试，可以轻松地验证单个单元的逻辑是否正确，在每次构建之后运行单元测试，可以帮助您快速捕获和修复因代码更改（重构、优化等）带来的回归问题。

一、单元测试的目的以及测试内容

为什么要进行单元测试？

- 提高稳定性，能够明确地了解是否正确的完成开发；
- 快速反馈bug，跑一遍单元测试用例，定位bug；
- 在开发周期中尽早通过单元测试检查bug，最小化技术债，越往后可能修复bug的代价会越大，严重的情况下会影响项目进度；
- 为代码重构提供安全保障，在优化代码时不用担心回归问题，在重构后跑一遍测试用例，没通过说明重构可能是有问题的，更加易于维护。

单元测试要测什么？

- 列出想要测试覆盖的正常、异常情况，进行测试验证;

- 性能测试，例如某个算法的耗时等等。

二、单元测试的分类

1. 本地测试(Local tests): 只在本地机器JVM上运行，以最小化执行时间，这种单元测试不依赖于Android框架，或者即使有依赖，也很方便使用模拟框架来模拟依赖，以达到隔离Android依赖的目的，模拟框架如google推荐的Mockito；
2. 仪器化测试(Instrumented tests): 在真机或模拟器上运行的单元测试，由于需要跑到设备上，比较慢，这些测试可以访问仪器（Android系统）信息，比如被测应用程序的上下文，一般地，依赖不太方便通过模拟框架模拟时采用这种方式。

三、JUnit 注解

了解一些JUnit注解，有助于更好理解后续的内容。

Annotation	**描述**
@Test public void method()	定义所在方法为单元测试方法
@Test (expected = Exception.class) public void method()	测试方法若没有抛出Annotation中的Exception类型(子类也可以)->失败
@Test(timeout=100) public void method()	性能测试，如果方法耗时超过100毫秒->失败
@Before public void method()	这个方法在每个测试之前执行，用于准备测试环境(如: 初始化类，读输入流等)，在一个测试类中，每个@Test方法的执行都会触发一次调用。

Annotation	描述
@After public void method()	这个方法在每个测试之后执行，用于清理测试环境数据，在一个测试类中，每个@Test方法的执行都会触发一次调用。
@BeforeClass public static void method()	这个方法在所有测试开始之前执行一次，用于做一些耗时的初始化工作(如: 连接数据库)，方法必须是static
@AfterClass public static void method()	这个方法在所有测试结束之后执行一次，用于清理数据(如: 断开数据连接)，方法必须是static
@Ignore或者@Ignore("太耗时") public void method()	忽略当前测试方法，一般用于测试方法还没有准备好，或者太耗时之类的
@FixMethodOrder(MethodSorters.NAME_ASCENDING) public class TestClass{}	使得该测试类中的所有测试方法都按照方法名的字母顺序执行，可以指定3个值，分别是DEFAULT、JVM、NAME_ASCENDING

四、本地测试

根据单元有没有外部依赖（如Android依赖、其他单元的依赖），将本地测试分为两类，首先看看没有依赖的情况：

- 添加依赖，google官方推荐

```
dependencies {
// Required -- JUnit 4 framework
testImplementation 'junit:junit:4.12'
// Optional -- Mockito framework(可选，用于模拟一些依赖对象，以达到隔离依赖的效果)
    testImplementation 'org.mockito:mockito-core:2.19.0'
}
```

- 单元测试代码存储位置
  事实上，AS已经帮我们创建好了测试代码存储目录。

```
app/src
├── androidTestjava（仪器化单元测试、UI测试） ├── main/java（业务代码）
└── test/java（本地单元测试）
```

- 创建测试类

可以自己手动在相应目录创建测试类，AS也提供了一种快捷方式：选择对应的类->将光标停留在类名上->按下ALT + ENTER->在弹出的弹窗中选择Create Test

> Note: 勾选setUp/@Before会生成一个带@Before注解的setUp()空方法，tearDown/@After则会生成一个带@After的空方法。

- 运行测试用例

1. 运行单个测试方法：选中@Test注解或者方法名，右键选择 Run ；
2. 运行一个测试类中的所有测试方法：打开类文件，在类的范围内右键选择 Run ，或者直接选择类文
   件直接右键 Run ；
3. 运行一个目录下的所有测试类：选择这个目录，右键 Run 。

- 运行前面测试验证邮箱格式的例子，测试结果会在 Run 窗口展示

也可以通过命令 gradlew test 来运行所有的测试用例，这种方式可以添加如下配置，输出单元测试过程中各类测试信息：

```
android { ...
    testOptions.unitTests.all {
        testLogging {
            events 'passed', 'skipped', 'failed', 'standardOut',
'standardError'
} }
```

在单元测试中通过System.out或者System.err打印的也会输出。

- 通过模拟框架模拟依赖，隔离依赖
  前面验证邮件格式的例子，本地JVM虚拟机就能提供足够的运行环境，但如果要测试的单元依赖了Android框架，比如用到了Android中的Context类的一些方法，本地JVM将无法提供这样的环境，这时候模拟框架Mockito就派上用场了。
- 一个Context#getString(int)的测试用例

```
import static org.hamcrest.core.Is.is;
import static org.junit.Assert.assertThat;
import static org.mockito.Mockito.when;
@RunWith(MockitoJUnitRunner.class)
public class MockUnitTest {
    private static final String FAKE_STRING = "AndroidUnitTest";
    @Mock
    Context mMockContext;
    @Test
    public void readStringFromContext_LocalizedString() {
//模拟方法调用的返回值，隔离对Android系统的依赖
 when(mMockContext.getString(R.string.app_name)).thenReturn(FAKE_STRING);
        assertThat(mMockContext.getString(R.string.app_name),
is(FAKE_STRING));
 when(mMockContext.getPackageName()).thenReturn("com.jdqm.androidunittest")
;
        System.out.println(mMockContext.getPackageName());
    }
}
```

read string from context

通过模拟框架Mockito，指定调用context.getString(int)方法的返回值，达到了隔离依赖的目的，其中Mockito使用的是cglib动态代理技术。

五、仪器化测试

在某些情况下，虽然可以通过模拟的手段来隔离Android依赖，但代价很大，这种情况下可以考虑仪器化的单元测试，有助于减少编写和维护模拟代码所需的工作量。

仪器化测试是在真机或模拟器上运行的测试，它们可以利用Android framework APIs 和 supporting APIs。如果测试用例需要访问仪器(instrumentation)信息(如应用程序的Context)，或者需要Android框架组件的真正实现(如Parcelable或SharedPreferences对象)，那么应该创建仪器化单元测试，由于要跑到真机或模拟器上，所以会慢一些。

- 配置

```
dependencies {
androidTestImplementation 'com.android.support:support-
annotations:27.1.1'
androidTestImplementation 'com.android.support.test:runner:1.0.2'
androidTestImplementation 'com.android.support.test:rules:1.0.2'
}
```

```
android {
...
defaultConfig {
...
testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
}
}
```

- Example
  这里举一个操作SharedPreference的例子，这个例子需要访问Context类以及SharedPreference的具体实现，采用模拟隔离依赖的话代价会比较大，所以采用仪器化测试比较合适。

这是业务代码中操作SharedPreference的实现

```
public class SharedPreferenceDao {
    private SharedPreferences sp;
    public SharedPreferenceDao(SharedPreferences sp) {
        this.sp = sp;
}
    public SharedPreferenceDao(Context context) {
        this(context.getSharedPreferences("config", Context.MODE_PRIVATE));
}
    public void put(String key, String value) {
        SharedPreferences.Editor editor = sp.edit();
        editor.putString(key, value);
        editor.apply();
}
    public String get(String key) {
        return sp.getString(key, null);
} }
```

创建仪器化测试类（app/src/androidTest/java）

```
// @RunWith 只在混合使用 JUnit3 和 JUnit4 需要，若只使用JUnit4，可省略
@RunWith(AndroidJUnit4.class)
public class SharedPreferenceDaoTest {
public static final String TEST_KEY = "instrumentedTest"; public static final
String TEST_STRING = "玉刚说";
    SharedPreferenceDao spDao;
@Before
    public void setUp() {
        spDao = new SharedPreferenceDao(App.getContext());
}
@Test
    public void sharedPreferenceDaoWriteRead() {
        spDao.put(TEST_KEY, TEST_STRING);
        Assert.assertEquals(TEST_STRING, spDao.get(TEST_KEY));
} }
```

运行方式和本地单元测试一样，这个过程会向连接的设备安装apk，测试结果将在Run窗口展示

仔细看打印的log，可以发现，这个过程向模拟器安装了两个apk文件，分别是app-debug.apk和app-debug-androidTest.apk，instrumented测试相关的逻辑在app-debug-androidTest.apk中。简单介绍一下安装apk命令pm install:

```
// 安装apk
//-t:允许安装测试 APK
//-r:重新安装现有应用，保留其数据，类似于替换安装
//更多请参考 https://developer.android.com/studio/command-line/adb?hl=zh-cn adb
shell pm install -t -r filePath
```

安装完这两个apk后，通过am instrument命令运行instrumented测试用例，该命令的一般格式：

```
am instrument [flags] <test_package>/<runner_class>
```

例如本例子中的实际执行命令：

```
adb shell am instrument -w -r -e debug false -e class
'com.jdqm.androidunittest.SharedPreferenceDaoTest#sharedPreferenceDaoWriteRea
d'
com.jdqm.androidunittest.test/android.support.test.runner.AndroidJUnitRunner
```

```
-w: 强制 am instrument 命令等待仪器化测试结束才结束自己(wait)，保证命令行窗口在测试期间 不
关闭，方便查看测试过程的log
-r: 以原始格式输出结果(raw format)
-e: 以键值对的形式提供测试选项，例如 -e debug false
关于这个命令的更多信息请参考
https://developer.android.com/studio/test/command-line?hl=zh-cn
```

如果你实在没法忍受instrumented test的耗时问题，业界也提供了一个现成的方案Robolectric

六、常用单元测试开源库

1. Mocktio

https://github.com/mockito/mockito

Mock对象，模拟控制其方法返回值，监控其方法的调用等。

- 添加依赖

```
testImplementation 'org.mockito:mockito-core:2.19.0'
```

Example

```
import static org.hamcrest.core.Is.is;
import static org.junit.Assert.*;
import static org.mockito.ArgumentMatchers.anyInt;
import static org.mockito.Mockito.*;
import static
org.mockito.internal.verification.VerificationModeFactory.atLeast;

@RunWith(MockitoJUnitRunner.class)
public class MyClassTest {

@Mock
MyClass test;

@Test
public void mockitoTestExample() throws Exception {

//可是使用注解@Mock替代
//MyClass test = mock(MyClass.class);



// 当调用test.getUniqueId()的时候返回 43
when(test.getUniqueId()).thenReturn( 18 );
// 当调用test.compareTo()传入任意的Int值都返回 43
when(test.compareTo(anyInt())).thenReturn( 18 );

// 当调用test.close()的时候，抛NullPointerException异常
doThrow(new NullPointerException()).when(test).close();
// 当调用test.execute()的时候，什么都不做
doNothing().when(test).execute();
assertThat(test.getUniqueId(), is(18));
// 验证是否调用了1次test.getUniqueId() verify(test, times(1)).getUniqueId();
// 验证是否没有调用过test.getUniqueId() verify(test, never()).getUniqueId();
// 验证是否至少调用过2次test.getUniqueId() verify(test, atLeast(2)).getUniqueId();
// 验证是否最多调用过3次test.getUniqueId() verify(test, atMost(3)).getUniqueId();
// 验证是否这样调用过:test.query("test string") verify(test).query("test string");
// 通过Mockito.spy() 封装List对象并返回将其mock的spy对象 List list = new
LinkedList();
List spy = spy(list);
//指定spy.get(0)返回"Jdqm" doReturn("Jdqm").when(spy).get(0);
assertEquals("Jdqm", spy.get(0));
}
}
```

-w: 强制 am instrument 命令等待仪器化测试结束才结束自己(wait)，保证命令行窗口在测试期间不关闭，方便查看测试过程的log
-r: 以原始格式输出结果(raw format)
-e: 以键值对的形式提供测试选项，例如 -e debug false
关于这个命令的更多信息请参考
https://developer.android.com/studio/test/command-line?hl=zh-cn

2. powermock

https://github.com/powermock/powermock

对于静态方法的mock

- 添加依赖

```
testImplementation 'org.powermock:powermock-api-mockito2:1.7.4'
testImplementation 'org.powermock:powermock-module-junit4:1.7.4'
```

> Note: 如果使用了Mockito，需要这两者使用兼容的版本，具体参考
> https://github.com/powermock/powermock/wiki/Mockito#supported-versions

- Example

```
@RunWith(PowerMockRunner.class)
@PrepareForTest({StaticClass1.class, StaticClass2.class})
public class StaticMockTest {

@Test
public void testSomething() throws Exception{
// mock完静态类以后，默认所有的方法都不做任何事情
mockStatic(StaticClass1.class);
when(StaticClass1.getStaticMethod()).thenReturn("Jdqm");
StaticClass1.getStaticMethod();
//验证是否StaticClass1.getStaticMethod()这个方法被调用了一次
verifyStatic(StaticClass1.class, times( 1 ));
}
}
```

或者是封装为非静态，然后用Mockito:

```
class StaticClass1Wraper{
  void someMethod() {
    StaticClass1.someStaticMethod();
  }
```

3. Robolectric

http://robolectric.org

主要是解决仪器化测试中耗时的缺陷，仪器化测试需要安装以及跑在Android系统上，也就是需要在Android虚拟机或真机上面，所以十分的耗时，基本上每次来来回回都需要几分钟时间。针对这类问题，业界其实已经有了一个现成的解决方案: Pivotal实验室推出的Robolectric，通过使用Robolectrict模拟Android系统核心库的Shadow Classes的方式，我们可以像写本地测试一样写这类测试，并且直接运行在工作环境的JVM上，十分方便。

- 添加配置

```
testImplementation "org.robolectric:robolectric:3.8"
android { ...
  testOptions {
    unitTests {
      includeAndroidResources = true
    }
} }
```

- Example
  模拟打开MainActivity，点击界面上面的Button，读取TextView的文本信息。

MainActivity.java

```
public class MainActivity extends AppCompatActivity {

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
final TextView tvResult = findViewById(R.id.tvResult);
Button button = findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
tvResult.setText("Robolectric Rocks!");
}
});
}
}
```

测试类(app/src/test/java/)

```
@RunWith(RobolectricTestRunner.class)
public class MyActivityTest {
@Test
    public void clickingButton_shouldChangeResultsViewText() throws
Exception {
        MainActivity activity =
Robolectric.setupActivity(MainActivity.class);
Button button = activity.findViewById(R.id.button); TextView results =
activity.findViewById(R.id.tvResult); //模拟点击按钮，调用OnClickListener#onClick
button.performClick();
        Assert.assertEquals("Robolectric Rocks!",
results.getText().toString());
} }
```

测试结果

Robolectric test passed

耗时 917 毫秒，是要比单纯的本地测试慢一些。这个例子非常类似于直接跑到真机或模拟器上，然而它只需要跑在本地JVM即可，这都是得益于Robolectric的Shadow。

> Note: 第一次跑需要下载一些依赖，可能时间会久一点，但后续的测试肯定比仪器化测试打包两个apk并安装的过程快。

在第六小节介绍了通过仪器化测试的方式跑到真机上进行测试SharedPreferences操作，可能吐槽的点都在于耗时太长，现在通过Robolectric改写为本地测试来尝试减少一些耗时。

在实际的项目中，Application可能创建时可能会初始化一些其他的依赖库，不太方便单元测试，这里额外创建一个Application类，不需要在清单文件注册，直接写在本地测试目录即可。

```
public class RoboApp extends Application {}
```

在编写测试类的时候需要通过@Config(application = RoboApp.class)来配置Application，当需要

传入Context的时候调用RuntimeEnvironment.application来获取：
app/src/test/java/

```
@RunWith(RobolectricTestRunner.class)
@Config(application = RoboApp.class)
public class SharedPreferenceDaoTest {

public static final String TEST_KEY = "instrumentedTest";
public static final String TEST_STRING = "玉刚说";


SharedPreferenceDao spDao;


@Before
public void setUp() {
//这里的Context采用RuntimeEnvironment.application来替代应用的Context
        spDao = new SharedPreferenceDao(RuntimeEnvironment.application);
    }
@Test
    public void sharedPreferenceDaoWriteRead() {
        spDao.put(TEST_KEY, TEST_STRING);
        Assert.assertEquals(TEST_STRING, spDao.get(TEST_KEY));
} }
```

像本地此时一样把它跑起来即可。

七、Mockito+PowerMock单元测试

1. 使用框架

JUnit+Mockito+PowerMock

由于Mockito不能测试private、static、final修饰的方法（kotlin方法、类默认有final修饰），故需要引入Mockito的增强版PowerMock来进行测试

2. 官方地址

Mockito: https://github.com/mockito/mockito

PowerMock: https://github.com/powermock/powermock

Mockito和PowerMock的版本对应关系，请参考：

https://github.com/powermock/powermock/wiki/Mockito

2. 优缺点

2.1 优点

- 可模拟测试绝大多数业务逻辑处理场景，覆盖面广
- 测试的类与其他类之间耦合，可通过模拟对象来解决，如Presenter与Model、View耦合
- 无需真机运行，测试效率高

2.2 缺点

- 不支持涉及到真实运行环境组件的方法测试，如Android相关组件、GreenDao、RxJava等

可通过引入Robolectric框架来模拟Android组件，或者用仪器化测试来作为单元测试的补充。

> Robolectric框架可用来模拟用户操作，测试具体的方法执行情况
>
> https://github.com/robolectric/robolectric

3. 常用打桩方法

方法名	描述
thenAnswer(Answer answer)	对结果进行拦截
thenReturn(T t)	设置返回结果
thenThrow(Throwable throwable)	设置要抛出的异常
thenCallRealMethod()	设置执行真实方法
doAnswer(Answer answer)	对结果进行拦截
doReturn(T t)	设置返回结果
doThrow(Throwable throwable)	设置要抛出的异常
doCallRealMethod()	设置执行真实方法
doNothing()	设置void方法内部逻辑不执行（空实现）

示例：

- when(presenter.someMethod()).thenReturn(object)

> 用于有返回值的方法，推荐写法，可读性更好

- doCallRealMethod().when(presenter).someVoidMethod()

> 用于无返回值的方法

4. 常用行为验证方法

方法名	描述
after(long timemills)	在指定时间后验证
timeout(long timemills)	验证方法是否超时
atLeast(int times)	验证方法至少执行了n次
atMost(int times)	验证方法最多执行了n次
description(String desc)	验证失败时输出内容
times(int n)	验证方法执行了n次
never()	验证方法从未执行，等同于times(0)
only()	验证方法只执行了一次，等同于times(1)

示例：

- Mockito.verify(presenter, times(2)).someMethod()
- PowerMockito.verifyPrivate(presenter, times(2)).invoke("methodName", 参数类型(Mockito.anyString())...)

5. 实践

5.1 初始化设置

@PrepareForTest注解中把需要mock的类都声明出来，presenter、model、view等等

```
@RunWith(PowerMockRunner.class)
@PrepareForTest({CollectedWrongQuestionPresenter.class,
        CollectedWrongQuestionActivity.class, CollectedWrongQuestionModel.class})
public class CollectedQuestionPresenterTest {

    Class<CollectedWrongQuestionPresenter> presenterClass = CollectedWrongQuestionPresenter.class;
    CollectedWrongQuestionPresenter presenter = PowerMockito.mock(presenterClass);
    CollectedWrongQuestionModel model = PowerMockito.mock(CollectedWrongQuestionModel.class);
```

5.2 示例一

- 重点

  测试类与其他类耦合

  doAnswer: 可以验证调用方法的参数是否正确，非void的方法还可以用获取到参数重写方法的实现，然后返回结果

  doCallRealMethod

  thenReturn

- 被测试的方法

```
fun init() {
    val subjectName: String? = mModel.getSubjectName()
    mView?.setMainTitle(subjectName)
    isStaggerMode = mModel.getDisplayMode()
    mView?.setDisplayMode(isStaggerMode)
    mView?.initFilter( subjectId: subjectId ?: 0)
}
```

- 测试方法中用到的一些成员变量赋值

mModel: 用mock出来的model对象通过反射赋值

mView: public变量直接用mock出来的Activity对象赋值

subjectId: val修饰的成员变量，仍然可以使用反射的方式赋值

```
//变量赋值
Field mModelField = PowerMockito.field(presenterClass, fieldName:"mModel");
mModelField.set(presenter, model);
CollectedWrongQuestionActivity activity = PowerMockito.mock(CollectedWrongQuestionActivity.class);
presenter.mView = activity;
```

- 被测试方法中涉及到的方法调用的模拟设置语句声明

  注意：如果mock出来的对象的方法没有设置语句声明，则方法被调用时不会执行任何逻辑，等同于doNothing，有返回值的方法直接返回默认值

  - init()：被测试方法直接设置真实执行，如果该方法不是void方法，则推荐的写法应为：

```
when(presenter).init().thenCallRealMethod();
```

  - mModel.getSubjectName()、mModel.getDisplayMode(): 直接设置返回值，方法的真实执行逻辑涉及到GreenDao、Mmkv等不能mock的框架，所以直接设置返回值模拟
  - mView.setMainTitle(mainTitle: String)、mView.setDisplayMode(isStaggerMode: Boolean)、mView.initFilter(subjectId: Int): 通过doAnswer验证调用方法的参数是否正确

```
//方法模拟声明
PowerMockito.doCallRealMethod().when(presenter).init();
PowerMockito.when(model.getSubjectName()).thenReturn("数学");
PowerMockito.when(model.getDisplayMode()).thenReturn(true);
PowerMockito.doAnswer(new Answer<Object>() {
    @Override
    public Object answer(InvocationOnMock invocation) throws Throwable {
        String subjectName = invocation.getArgument( index: 0);
        Assert.assertEquals( expected: "数学", subjectName);
        return null;
    }
}).when(activity).setMainTitle(Mockito.anyString());
```

- 完整测试方法

```
@Test
public void testInit() throws Exception{
    //变量赋值
    Field mModelField = PowerMockito.field(presenterClass, fieldName:"mModel");
    mModelField.set(presenter, model);
    CollectedWrongQuestionActivity activity = PowerMockito.mock(CollectedWrongQuestionActivity.class);
    presenter.mView = activity;
    //方法模拟声明
    PowerMockito.doCallRealMethod().when(presenter).init();
    PowerMockito.when(model.getSubjectName()).thenReturn("数学");
    PowerMockito.when(model.getDisplayMode()).thenReturn(true);
    PowerMockito.doAnswer(new Answer<Object>() {
        @Override
        public Object answer(InvocationOnMock invocation) throws Throwable {
            String subjectName = invocation.getArgument( index: 0);
            Assert.assertEquals( expected: "数学", subjectName);
            return null;
        }
    }).when(activity).setMainTitle(Mockito.anyString());
    //被测试方法调用
    presenter.init();
}
```

5.3 示例二：

- 重点

  private方法mock

  静态方法mock

- 被测试方法：

```
/**
 * 获取需下载答案的url列表
 */
private fun getNeedDownloadAnswerUrls(dataList: List<CollectedQuestionBean>) : List<String> {
    val needDownloadAnswerUrls = arrayListOf<String>()
    for (collectedQuestionBean in dataList) {
        //题目已有瀑布流模式缓存图片，无需下载答案
        if (!collectedQuestionBean.questionBitmapPath.isNullOrEmpty()) continue
        val collectQuestion = collectedQuestionBean.collectQuestion ?: continue
        val list = if (!CollectedQuestionUtils.parseWithYunzuoyeMode(collectQuestion.answerDto)) {
            //题舟
            getAnswerUrlListForTizhou(collectQuestion)
        } else {
            //云作业
            getAnswerUrlListForYunZuoYe(collectQuestion)
        }
        needDownloadAnswerUrls.addAll(list)
    }
    return needDownloadAnswerUrls
}
```

- PowerMockito.method(): 反射出private方法的Method对象

  PowerMockito.field(): 反射出private属性的Field对象

```
//测试方法调用
Method method = PowerMockito.method(modelClass, methodName: "getNeedDownloadAnswerUrls", List.class);
List<String> answerUrls = (List<String>) method.invoke(model, list);
```

- 私有方法mock声明

```
//模拟声明
PowerMockito.when(model, methodName: "getNeedDownloadAnswerUrls", Mockito.anyList()).thenCallRealMethod();
PowerMockito.when(model, methodName: "getAnswerUrlListForTizhou",
        Mockito.any(CollectQuestion.class)).thenCallRealMethod();
PowerMockito.when(model, methodName: "getAnswerUrlListForYunZuoYe",
        Mockito.any(CollectQuestion.class)).thenCallRealMethod();
```

- 静态方法mock声明

```
/**
 * 获取云作业需下载的答案url列表
 */
private fun getAnswerUrlListForYunZuoYe(collectQuestion: CollectQuestion): List<String> {
    val urlList = arrayListOf<String>()
    val answerDtoBean = collectQuestion.answerDto ?: return urlList
    if (!answerDtoBean.annotationUrl.isNullOrEmpty()) {...}
    if (!answerDtoBean.childrenUrl.isNullOrEmpty()) {...}
    if (!answerDtoBean.correctedUrl.isNullOrEmpty()) {...}
    if (!answerDtoBean.proofsUrl.isNullOrEmpty()) {
        val proofs = answerDtoBean.proofsUrl
        if (FileUtils.checkIsUrl(proofs)) {
            urlList.add(proofs)
        } else if (CompressUtil.isBase64(proofs)) {
            val url = CompressUtil.unGzip(proofs)
            if (FileUtils.checkIsUrl(url)) {
                urlList.add(url)
            }
        }
    }

    return urlList
}
```

```
/**
 * 检查是否是url
 *
 * @param url url
 * @return true or false
 */
public static boolean checkIsUrl(@android.support.annotation.Nullable String url) {
    if (TextUtils.isEmpty(url)) {
        return false;
    }
    return HTTP_PATTERN.matcher(url).matches();
}
```

getAnswerUrlListForYunZuoYe()方法中使用了FileUtils的checkIsUrl方法，该方法中使用了android包下的TextUtils工具类，所以需要mock该静态方法，直接设置返回值为true

```
@PrepareForTest({CollectedWrongQuestionModel.class, FileUtils.class, NetworkApiFactoryImpl.class, CommonApiImpl.class})
PowerMockito.mockStatic(FileUtils.class);
PowerMockito.when(FileUtils.checkIsUrl(Mockito.anyString())).thenReturn(true);
```

> 引申：如果需要mock的静态方法是void方法，则mock方式与mock私有方法相似，示例如下：

```
object QuestionUtils {
    /**
     * 设置题目为题库场景
     */
    @JvmStatic
    fun setQtQuestionToQuestionBankScene(qtQuestion: QTQuestion) {
        qtQuestion.sceneType = QTConstants.SceneType.SCENE_QUESTION_BANK
        qtQuestion.clientType = QTConstants.ClientType.TEACHER
    }
}
```

```
PowerMockito.mockStatic(QuestionUtils.class);
PowerMockito.doCallRealMethod().when(QuestionUtils.class,
                    "setQtQuestionToQuestionBankScene",
Mockito.any(QTQuestion.class));
```

- 完整测试方法：

```
@Test
public void testGetNeedDownloadAnswerUrls() throws Exception {
    //模拟声明
    PowerMockito.when(model, methodName: "getNeedDownloadAnswerUrls", Mockito.anyList()).thenCallRealMethod();
    PowerMockito.when(model, methodName: "getAnswerUrlListForTizhou",
            Mockito.any(CollectQuestion.class)).thenCallRealMethod();
    PowerMockito.when(model, methodName: "getAnswerUrlListForYunZuoYe",
            Mockito.any(CollectQuestion.class)).thenCallRealMethod();
    PowerMockito.mockStatic(FileUtils.class);
    PowerMockito.when(FileUtils.checkIsUrl(Mockito.anyString())).thenReturn(true);
    //模拟数据
    String s = mockQuestionBeanJson;
    List<CollectedQuestionBean> list = new Gson().fromJson(s,
            new TypeToken<List<CollectedQuestionBean>>() {}.getType());
    //测试方法调用
    Method method = PowerMockito.method(modelClass, methodName: "getNeedDownloadAnswerUrls", List.class);
    List<String> answerUrls = (List<String>) method.invoke(model, list);
    //验证方法
    Assert.assertEquals( expected: 6, answerUrls.size());
}
```

5.4 示例三

- 重点

  异步回调方法测试

- 被测试方法

```
/**
 * 请求管控数据
 */
private fun requestControlData(staggerMode: Boolean, pageIndex: Int, selectionsBean: SelectionsBean?) {
    if (subjectId == null) {
        callback?.onError( failBean: null,  customError: "参数错误，科目id为空")
        return
    }
    NetworkApiFactoryImpl.getInstance().commonApi.getQuesitonControlData(subjectId,
            object : AbstractOnApiListener<QuestionControlData>() {
                override fun onSuccess(data: QuestionControlData?) {
                    if (data == null) {
                        callback?.onError( failBean: null,  customError: "数据解析异常")
                        return
                    }
                    controlData = hashMapOf()
                    data.collectQuestionCheckDtoList.forEach { it: CollectQuestionCheckDto
                        controlData!![it.questionId ?: ""] = it.shieldList ?: arrayListOf()
                    }
                    queryQuestions(staggerMode, pageIndex, selectionsBean, callback!!)
                }

                override fun onError(bean: RequestFailBean?) {
                    callback?.onError(bean,  customError: null)
                }

            })
}
```

- 声明一个ArgumentCaptor，泛型为回调类型

```
@Captor
private ArgumentCaptor<AbstractOnApiListener<QuestionControlData>> captor;
```

- ArgumentCaptor使用

```
Mockito.verify(commonApi).getQuesitonControlData(Mockito.anyInt(), captor.capture());
//模拟返回数据
QuestionControlData controlData = new QuestionControlData(new ArrayList<>(),  studentId: 151612,  subjectId: 2);
captor.getValue().onSuccess(controlData);
```

- 完整测试方法

```
@Test
public void testRequestControlData() throws Exception{
    //省略部分代码
    CommonApiImpl commonApi = PowerMockito.mock(CommonApiImpl.class);

    //mock声明
    PowerMockito.doCallRealMethod().when(model,  methodToExpect: "requestControlData", Mockito.anyBoolean(),
            Mockito.anyInt(), Mockito.any(SelectionsBean.class));
    //方法调用
    Method method = PowerMockito.method(modelClass,  methodName: "requestControlData",
            boolean.class, int.class, SelectionsBean.class);
    method.invoke(model,  ...args: true, 1, new SelectionsBean());
    Mockito.verify(commonApi).getQuesitonControlData(Mockito.anyInt(), captor.capture());
    //模拟返回数据
    QuestionControlData controlData = new QuestionControlData(new ArrayList<>(),  studentId: 151612,  subjectId: 2);
    captor.getValue().onSuccess(controlData);
    //验证
    Mockito.verify(model).queryQuestions(Mockito.anyBoolean(), Mockito.anyInt(),
            Mockito.any(SelectionsBean.class), Mockito.any(CollectedWrongQuestionModel.DataCallback.class));
}
```

- 使用doAnswer也可以模拟异步回调方法

AndroidJUnit相关

在某些情况下，虽然可以通过模拟的手段来隔离Android依赖，但代价很大，这种情况下可以考虑仪器化的单元测试，有助于减少编写和维护模拟代码所需的工作量。

仪器化测试是在真机或模拟器上运行的测试，它们可以利用Android framework APIs 和 supportingAPIs。如果测试用例需要访问仪器(instrumentation)信息(如应用程序的Context)，或者需要Android框架组件的真正实现(如SharedPreferences对象或者本地数据库)，那么应该创建仪器化单元测试，由于要跑到真机或模拟器上，所以会慢一些。

- 配置

```
dependencies {
androidTestImplementation 'com.android.support.test:runner:1.0.2'
androidTestImplementation 'com.android.support.test:rules:1.0.2'
}
```

```
android {
...
defaultConfig {
...
testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
}
}
```

- 这里举一个错题本首页校验数据的例子，这个例子需要操作本地数据库，采用模拟隔离依赖的话代价会比较大，所以采用仪器化测试比较合适。

```
NetworkApiFactoryImpl.getInstance().commonApi.getUserInfo(object :
AbstractOnApiListener<UserInfoBean?>() {

            override fun onSuccess(data: UserInfoBean?) {
                if (data != null) {
                        //保存到数据库
                    UserInfoHelper.insertOrUpdate(data)
                    MmkvUtils.updateSubjectIdList(data.subjectIdList)
                }
                getStatisticResponse(abstractOnApiListener)
            }

            override fun onError(bean: RequestFailBean) {
                getStatisticResponse(abstractOnApiListener)
            }
        })


  private fun getStatisticResponse(abstractOnApiListener:
AbstractOnApiListener<StudentStatisticsResponse>) {
        HomeApiImpl.getHomeApi().getHomeStatistic(object :
AbstractOnApiListener<StudentStatisticsResponse>() {
            /**
             * 请求成功回调
             *
             * @param data data
             */
            override fun onSuccess(data: StudentStatisticsResponse?) {
                val subjectList = MmkvUtils.getSubjectIdList()?: mutableListOf()
                var statisticsResponse = data
                if (statisticsResponse == null) {
                    statisticsResponse = StudentStatisticsResponse()
                }
                //缓存=><科目,题目数量>map
                val map = mutableMapOf<Int, Int>()
                for (subjectBean in statisticsResponse.subjectCountDtoList) {
                    map[subjectBean.subjectId] =
subjectBean.thisWeekQuestionCount
                }
                val subjectCountList = mutableListOf<SubjectCountDto>()
```

```
                var subjectCountDto: SubjectCountDto?
                //词典表查询对应的科目词典列表
                val subjectDictBeanList =
DictionaryHelper.getDictionaryList(subjectList, DictionaryBean.SUBJECT)
                for (subjectDictBean in subjectDictBeanList) {
                    //重新生成科目统计数据对象
                    subjectCountDto = SubjectCountDto(subjectDictBean.id,
                            map[subjectDictBean.id] ?: 0,
                            subjectDictBean.name)
                    subjectCountList.add(subjectCountDto)
                }
                statisticsResponse.subjectCountDtoList = subjectCountList
                if (statisticsResponse.weekQuestionDtoList.size == 4) {

statisticsResponse.weekQuestionDtoList[statisticsResponse.weekQuestionDtoList.si
ze - 1].weekStr = "本周"
                }
                //处理完数据回调
                abstractOnApiListener.onSuccess(statisticsResponse)
            }

            /**
             * 请求失败回调
             *
             * @param bean bean
             */
            override fun onError(bean: RequestFailBean?) {
                abstractOnApiListener.onError(bean)
            }

        })
    }
```

- 创建仪器化测试类

```
@RunWith(AndroidJUnit4::class)
class HomePresenterTest {

    private lateinit var homePresenter: HomePresenter
    private lateinit var homeModel: HomeModel
    private var context =
InstrumentationRegistry.getInstrumentation().targetContext

    //是否获取用户信息接口
    private var getUserInfo = true
    private var subjectId = 1
    private var studentName = "张子钰"
    private var studentId: Long = 36364

    @Before
    fun initData() {
        homePresenter = HomePresenter()
        homeModel = HomeModel()
    }

    @Test
    fun onRefresh() {
```

```
        homeModel.getHomeData(getUserInfo, object :
AbstractOnApiListener<StudentStatisticsResponse>() {

            override fun onSuccess(data: StudentStatisticsResponse?) {
                Assert.assertEquals(4, data?.weekQuestionDtoList?.size)
                Assert.assertEquals(8, data?.subjectCountDtoList?.size)
                Assert.assertEquals(studentId,
homePresenter.getUserInfo()?.studentId)
                Assert.assertEquals(studentName,
homePresenter.getUserInfo()?.studentName)
            }

            override fun onError(bean: RequestFailBean?) {
                Assert.assertTrue(false)
            }
        })

    }
.......
}
```

运行方式和本地单元测试一样，这个过程会向连接的设备安装apk，测试结果将在Run窗口展示