Android单元测试框架介绍 -- 调研篇

Android Han 于 2021-08-29 14:46:40 发布

查阅了关于Android 单元测试 的一些资料后，发现网上对于现有的单元测试框架说法褒贬不一，各有优缺点，在框架的选择上也是需要很多因素来决定，因此我总结了一下最近的调研，对现有的单元测试框架做下简单介绍，列举下各个框架的优缺点，根据每个框架的优缺点，做下抉择。

单元测试一般分两类：

- 本地测试：运行在本地的计算机上，这些测试编译之后可以直接运行在本地的Java虚拟机上（JVM）。可以最大限度的缩短执行的时间。如果测试中用到了Android框架中的对象，那么谷歌推荐使用Robolectric来模拟对象。
- 插桩测试：在Android设备或者模拟器上运行的测试，这些测试可以访问插桩测试信息，比如被测设备的Context，使用此方法可以运行具有复杂Android依赖的单元测试。Espresso 和 UI Automator就是这类测试，Espresso一般用来测试单个界面，UI Automator一般用来测试多界面交互。它们运行的比本地测试慢很多，所以谷歌建议最好是必须针对设备测试的时候才使用。

1 框架调研

使用AS在新建工程时，可以看到src目录下有androitTest和test两个目录，二者都是Android测试代码的目录，但有所不同：

1. /src/androidTest的代码需要运行在真机/模拟器上，主要是测某个功能是否正常，类似于UI自动化测试。
2. /src/test的代码可以直接运行在JVM上，可以验证函数级别的逻辑，就是我们一般所说的单元测试代码。

所以说Android的测试代码分为 运行在真机和 JVM 上两类，下面介绍下相关的几个框架：

1. JUnit，Java单元测试的根基，基本上都是通过断言来验证函数返回值/对象的状态是否正确。
2. Espresso，谷歌官方提供的UI自动化测试框架，需要运行在手机/模拟器上，类似于Appium。
3. Robolectric，实现了一套可以在JVM上运行的Android代码。
4. Mockito，如果被测的业务依赖比较复杂的上下文，就可以使用Mock来模拟被测代码依赖的对象来保证单元测试的进行。

1.1 JUnit

JUnit是Java单元测试的根基，测试用例的运行和验证都依赖于它来进行。Android使用Java语言开发，**Android单元测试自然离不开JUnit。**

JUnit的用途主要是：

1. 提供了若干注解，轻松地组织和运行测试。
2. 提供了各种断言api，用于验证代码运行是否符合预期。

断言的api不做介绍了，自行查阅官方wiki。

简单介绍一下几个常用注解：

1.@Test

标记该方法为测试方法。测试方法必须是public void，可以抛出异常。

2.@RunWith

指定一个Runner来提供测试代码运行的上下文环境。（Runner的概念）

3.@Rule

定义测试类中的每个测试方法的行为，比如指定某个Acitivity作为测试运行的上下文。

4.@Before

初始化方法，通常进行测试前置条件的设置。

5.@After

释放资源，它会在每个测试方法执行完后都调用一次。

```
@RunWith(JUnit4.class)
public class JUnitSample {
    Object object;

    //初始化方法，通常进行用于测试的前置条件/依赖对象的初始化
    @Before
    public void setUp() throws Exception {
        object = new Object();
    }

    //测试方法，必须是public void
    @Test
    public void test() {
        Assert.assertNotNull(object);
    }
}
```

> ps: 一个测试类单元测试的执行顺序为： @BeforeClass –> @Before –> @Test –> @After –> @AfterClass

结论：JUnit是单元测试的根基。

1.2 Espresso

谷歌官方的UI自动化测试框架，用Espresso写的测试代码，必须跑在emulator或者是device上面，并且在测试代码的运行过程中，也会真正的拉起页面、发生UI交互、文件读写、网络请求等等，最后通过各种断言检查UI状态。

框架提供了以下三类api:

1. ViewMatchers，找出被测的View对象，相当于在测试代码中实现了findViewById。
2. ViewActions，发送交互事件，即在测试代码中模拟UI触摸交互。
3. ViewAssertions，验证UI状态，在测试代码运行完成后检查UI状态是否符合预期，可以看做是UI状态的断言。

话不多说，直接看简单demo：

```
//使用Espresso提供的AndroidJUnit4运行测试代码
@RunWith(AndroidJUnit4.class)
public class EspressoSample {

    // 利用Espresso提供的ActivityTestRule拉起MainActivity
    @Rule
    public ActivityTestRule<MainActivity> mIntentsRule = new IntentsTestRule<>(MainActivity.class);

    @Test
    public void testNoContentView() throws Exception {
        //withId函数会返回一个ViewMatchers对象，用于查找id为R.id.btn_get的view
        onView(withId(R.id.btn_get))
                //click函数会返回一个ViewActions对象，用于发出点击事件
                .perform(click());

        //通过定时轮询loadingView是否展示中，来判断异步的网络请求是否完成
        View loadingView = mIntentsRule.getActivity().findViewById(R.id.loading_view);
        while (true) {
            Thread.sleep(1000);
            if (loadingView.getVisibility() == View.GONE) {
                break;
            }
        }

        //请求请求完成后，检查UI状态
        //找到R.id.img_result的view
        onView(withId(R.id.img_result))
                //matches函数会返回一个ViewAssertions对象，检查这个view的某个状态是否符合预期
                .check(matches(isDisplayed()));
    }
}
```


以上测试代码需要运行在真机/模拟器上，运行过程中可以看到自动拉起MainActivity，并且自动点击了id为btn_get的按钮，然后loading结束后，检查到id为img_result正在展示中，符合预期，整个测试用例就执行成功了。

可以感觉到Espresso的确比较强大，通过其提供的api，常用的UI逻辑基本都可以进行测试。但在复杂项目中，Espreeso的缺点也非常明显：

1. 粒度粗。

Espresso本质上就是一种UI自动化测试方案，很难去验证函数级别的逻辑，如果仅仅是想验证某个功能是否正常的话，又受限于网络状况、设备条件甚至用户账户等等因素，测试结果不可控。

2. 逻辑复杂。

一般页面UI元素庞大且复杂，不可能每个View的交互逻辑都去写测试代码验证，只能选择性验证一些关键交互。

3. 运行速度慢。

用Espresso写测试代码，必须跑在emulator或者是device上面。运行测试用例就变成了一个漫长的过程，因为要打包、上传到机器、然后再一个一个地运行UI界面，这样做的好处是手机上的表现很直观，但是调试和运行速度是真的慢，效率和便捷性上肯定是不如人工测试。

结论：Espresso用例的编写就像是在做业务代码的逆向实现，在实际工作中还不如直接运行项目代码进行人工自测，所以个人感觉Espresso是一个强大的UI自动化测试工具，而非单元测试的解决方案。

1.3 Robolectric

在查找的众多资料中，都建议不使用Robolectric框架，只有美团技术团队发表了一篇关于使用Robolectric的文章。

Robolectric有点倾向于UI的一个测试框架：查询一个控件，模拟点击，验证逻辑

Espresso的问题很明显，那么有没有可能让Android代码脱离手机/模拟器，直接运行在JVM上面呢?

我们需要一个能够隔离Android依赖，并且能够 直接在IDE里run一下就可以知道结果的单元测试方案。

这就牵涉到android.jar的问题，android.jar包含了Android Framework的所有类、函数、变量的声明，但它没有任何具体实现，android.jar仅仅用于JAVA代码编译时，并不会真正打包进APK，Android Framework的真正实现是在设备/模拟器上。在JVM上调用Android SDK里的函数会直接throw RuntimeException。

所以Android单元测试需要解决的一大痛点，就是如何隔离整个Android SDK的依赖。

谷歌官方推荐的开源测试框架 Robolectric就是这么一个工具，简单来说它实现了一套可以在JVM上运行的Android代码。

谷歌官方推荐的开源测试框架 Robolectric就是这么一个工具，它实现了一套可以在JVM上运行的Android代码。Shadow是Robolectric的核心，这个框架针对Android SDK中的对象，提供了很多影子对象（如Activity和ShadowActivity、TextView和ShadowTextView等），Robolectric的本质是在Java运行环境下，采用Shadow的方式对Android中的组件进行模拟测试，从而实现Android单元测试。对于一些Robolectirc暂不支持的组件，可以采用自定义Shadow的方式扩展Robolectric的功能。

常见Robolectric用法：

Robolectric支持单元测试范围从Activity的跳转、Activity展示View（包括菜单）和Fragment到View的点击触摸以及事件响应，同时Robolectric也能测试Toast和Dialog。对于需要网络请求数据的测试，Robolectric可以模拟网络请求的response。对于一些Robolectric不能测试的对象，比如ConcurrentTask，可以通过自定义Shadow的方式现实测试。下面将着重介绍Robolectric的常见用法。

Robolectric 2.4模拟网络请求

```
public void prepareHttpResponse(String filePath) throws IOException {

String netData = FileUtils.readFileToString(FileUtils.

toFile(getClass().getResource(filePath)), HTTP.UTF_8);

Robolectric.setDefaultHttpResponse(200, netData);

}//代码适用于Robolectric 2.4，3.0需要注意网络请求的包的位置
```

由于Robolectric 2.4并不会发送网络请求，因此需要本地创建网络请求所返回的数据，上述函数的filePath便是本地数据的文件的路径，setDefaultHttpResponse()则创建了该请求的Response。上述函数执行后，单元测试工程便拥有了与本地数据数据对应的网络请求，在这个函数执行后展示的Activity便是有数据的Activity。

在Robolectric 3.0环境下，单元测试可以发真的请求，并且能够请求到数据，本文依旧建议采用mock的办法构造网络请求，而不要依赖网络环境。

Activity展示测试与跳转测试

创建网络请求后，便可以测试Activity了。测试代码如下：

```
@Test

public void testSampleActivity(){

SampleActivity sampleActivity＝Robolectric.buildActivity(SampleActivity.class).
```



```
create().resume().get();

assertNotNull(sampleActivity);

assertEquals("Activity的标题", sampleActivity.getTitle());

}
```

Robolectric.buildActivity()用于构造Activity，create()函数执行后，该Activity会运行到onCreate周期，resume()则对应onResume周期。assertNotNull和assertEquals是JUnit中的断言，Robolectric只提供运行环境，逻辑判断还是需要依赖JUnit中的断言。

Activity跳转是Android开发的重要逻辑，其测试方法如下：

```
@Testpublic void testActivityTurn(ActionBarActivity firstActivity, Class secondActivity) {

Intent intent = new Intent(firstActivity.getApplicationContext(), secondActivity);

assertEquals(intent, Robolectric.shadowOf(firstActivity).getNextStartedActivity());//3.0的API与2.4不同

}
```

控件的点击以及可视验证

```
@Testpublic void testButtonClick(int buttonID){

Button submitButton = (Button) activity.findViewById(buttonID);

assertTrue(submitButton.isEnabled());

submitButton.performClick();

//验证控件的行为

}
```

Dialog和Toast测试

测试Dialog和Toast的方法如下：

```
public void testDialog(){
Dialog dialog = ShadowDialog.getLatestDialog();
assertNotNull(dialog);
}
public void testToast(String toastContent){
ShadowHandler.idleMainLooper();
assertEquals(toastContent, ShadowToast.getTextOfLatestToast());
}
```

网上对于Robolectric的弊端吐槽大于使用，下面列举下Robolectric的一些弊端：

1. Robolectric版本和Android SDK版本强依赖。Robolectric会shadow大部分Android的代码版本分散且缺少说明
1. 首次启动Robolectric会下载maven相关的依赖失败。这个依赖的文件较大，且下载逻辑是写在Robolectric框架里的，不能通过网络代理的方式解决，网上有一些解决方案，但在新版本的Robolectric里都已经失效了。
1. 不兼容第三方库。大量的第三方库并没有对应的shadow类，会在启动时/运行时报错。
1. 静态代码块易报错。我们经常在静态代码块里去加载so库或者执行一些初始化逻辑，基本上都会报错且很难解决。如果为了这个单元测试反过来去修改这些逻辑，就显得有点本末倒置、得不偿失了。
1. 在另外一篇博客中，博主说mock框架与Robolectric存在兼容问题。

国外关于Robolectri也有不少讨论：https://www.philosophicalhacker.com/post/why-i-dont-use-roboletric/

结论：当被测的代码（Presenter、Model层等）不可避免的依赖了Android SDK代码（比如TextUtils、Looper等），Robolectric可以轻松地让测试代码跑在JVM上，这应该是Robolectric的最大意义了。但是因为上述几点的情况，当连成功运行代码都成为了一种奢望，我不觉得这么一个单元测试框架能够在项目落地。

1.4 Mock

Junit已经能完成单元测试了，为啥要使用Mockito或者Robolectric?

我们需要明确单元测试的目的：单元测试的目的是为了测试我们自己写的代码的正确性，它不需要测试外部的各种依赖，所以当我们遇到一个方法中有很多别的对象的依赖的时候，比如操作数据库，连接网络，读写文件等等，需要给它解依赖。

怎么解依赖呢？其实就是弄一些假对象，比如代码中是我们从网络获取一段json数据，转化成一个对象传入到我们的测试方法中。那么就可以直接new一个假的对象，并给它设置我们期望的返回值传给要测试的方法就好了，不需要再去请求网络获取数据。这个过程称之为mock

直接手动去new一个对象，然后去设置各种数据是比较麻烦的，而Mockito这类的框架就是用来简化我们手动mock的。使用他们来创建一个虚拟对象设置返回值等操作会变得非常简单。

Mock框架基本上是以下2个：

1. Mockito：

模拟对象并使其按照我们预期执行/返回（类似代码打桩）

验证模拟对象是否按照预期执行/返回

1. PowerMockito：

基于Mockito的扩展，二者的api都非常相似

支持模拟静态函数、构造函数、私有函数、final 函数以及系统函数

mockito

下面开始练习，测试代码写在 src/main/test/java文件夹下面

先练习使用mockito，引入依赖库

```
testImplementation 'org.mockito:mockito-core:3.0.0'
```

新建一个MockitoTest类，在类上添加注解@RunWith(MockitoJUnitRunner.class)表示Junit要把测试方法运行在MockitoJUnitRunner上

```
@RunWith(MockitoJUnitRunner.class)
public class MockitoTest {......}
```

例子1： 结果验证，测试某些结果是否正确，使用when和thenReturn表示当调用某个方法的时候指定返回值。最后通过assertEquals判断返回值是否正确

```
@Test
public void testMockitoResult() {
Person person = mock(Person.class);
//当调用person.getAge()方法的时候，给它返回一个18
when(person.getAge()).thenReturn(18);
//当调用person.getName()方法的时候，给它返回一个Lily
when(person.getName()).thenReturn("Lily");
//判断返回跟预期是否一样
assertEquals(18, person.getAge());
assertEquals("Lily", person.getName());
}
```

例子2： 验证行为，有时候会测试某些行为是否被执行过，通过verify方法可以验证某个方法是否执行过，执行的次数

```
@Test
public void testMockitoBehavior() {
Person person = mock(Person.class);
int age = person.getAge();
//验证getAge动作有没有发生
verify(person).getAge();
//验证person.getName()是不是没有调用
```



```
verify(person, never()).getName();
//验证是否最少调用过一次person.getAge
verify(person, atLeast(1)).getAge();
//验证getAge动作是否被调用了2次，前面只用了一次所以这里会报错
verify(person, times(2)).getAge();
}
```

例子3： 可以使用@Mock注解来mock一个对象比如

```
@Mock
List<Integer> mList;
@Test
public void testAnnotationMock(){
mList.add(0);
verify(mList).add(0);
}
```

Mockito虽然好用但是也有些不足，比如不能mock static、final、private等对象，使用PowerMock就可以实现了

PowerMockito

因为PowerMockito是基于Mockito的扩展，所以二者的api都非常相似，常用api是以下两类：

1. 模拟对象并指定某些函数的执行/返回值

```
when(...).thenReturn(...)
```

1. 验证模拟对象是否按照预期执行/返回

```
verify(...).invoke(...)
```

添加依赖:

```
testImplementation 'org.powermock:powermock-module-junit4:2.0.2'
testImplementation 'org.powermock:powermock-api-mockito2:2.0.2'
```

创建一个PowerMockTest类，在类上添加注解@RunWith(PowerMockRunner.class)，通知Junit该类的测试方法运行在PowerMockRunner中。在添加注解@PrepareForTest(Utils.class)表示要测试的方法所在的类，这里是一个自定义的Utils.class

例子1： 测试static方法

目标方法:

```
public static boolean isEmpty(@Nullable CharSequence str) {
return str == null || str.length() == 0;
}
```

测试方法:

```
@Test
public void testStatic(){
PowerMockito.mockStatic(Utils.class);
PowerMockito.when(Utils.isEmpty("abc")).thenReturn(false);
assertFalse(Utils.isEmpty("abc"));
}
```


例子2： 测试private方法 替换私有变量

目标方法:

```
private String name;
private String changeName(String name) {
return "ABC" + name;
}
public String getName() {
return name;
}
```

测试方法:

```
@Test
public void testPrivate() throws Exception {
Utils util = new Utils();
//调用私有方法
String res = Whitebox.invokeMethod(util, "changeName", "Lily");
assertEquals("ABCLily",res);
//替换私有变量 也可以使用MemberModifier来修改
Whitebox.setInternalState(util,"name","Lily");
assertEquals("Lily",util.getName());
}
```

例子3：

1. Mock被依赖的复杂对象
2. 执行被测代码
3. 验证逻辑是否按照预期执行/返回

```
public class PowerMockitoSample {

private MainActivity activity;

private ImageView mockImg;

private TextView mockTv;

@Before

public void setUp() {

activity = new MainActivity();

// 1. Mock被依赖的复杂对象。

// MainActivity依赖了一些View，下面就是Mock出被依赖的复杂对象，并使之成为MainActivity的私有变量

mockImg = PowerMockito.mock(ImageView.class);

Whitebox.setInternalState(activity, "resultImg", mockImg);

mockTv = PowerMockito.mock(TextView.class);

Whitebox.setInternalState(activity, "resultTv", mockTv);

Whitebox.setInternalState(activity, "loadingView", PowerMockito.mock(ProgressBar.class));

}

@Test

public void test_onFail() throws Exception {

// 2. 执行被测代码。
```



```
// 这里要验证activity.onFail()函数

String errorMessage = "test";

activity.onFail(errorMessage);

// 3. 验证逻辑是否按照预期执行/返回。

// 这里需要验证resultImg 和 resultTv有没有按照预期进行UI状态的改变

verify(mockImg).setImageResource(R.drawable.ic_error);

verify(mockTv).setText(errorMessage);

}

}
```

上面代码我们把MainActivity所依赖的各种View对象通过mock实现后，剩下的基本都是工作量的问题了。

可以看到，借助Mock框架可以很好的隔离复杂的依赖对象（比如View），从而保证被测的独立单元可以与程序的其他部分相隔离的情况下进行测试，然后专注于验证某个函数/模块的逻辑是否正确且健壮。

必须注意的是，在实际项目中会有很多常用但不影响业务逻辑的代码（Log以及其他统计代码等），部分静态代码块也直接调用Android SDK api。因为单元测试代码运行在JVM上，需要抑制/隔离这些代码的执行，PowerMockito都提供了不错的支持（下篇细说）。

结论：通过PowerMockito这种强大的Mock框架，将被测类所依赖的复杂对象直接代理掉，既不会要求侵入式地修改业务代码 也能够保证单元测试代码 快速有效地运行在JVM上，

2 结论

1. JUnit是基础。
2. Espresso需要跑在真机上，可用于依赖Android平台的功能测试而非单元测试。
3. Roboelctric问题太多在复杂项目中寸步难行，弃了。
4. Android单元测试主要是通过PowerMockito来隔离整个Android SDK以及项目业务的依赖，将单元测试的重心放在较细粒度（函数级别）的代码逻辑上。

PowerMockito非常强大，但PowerMock使用的越多，表示被测试的代码抽象层次越低，代码质量和结构也越差，有点历史的大型项目都是类似的情况。

参考文章

Android自动化测试入门(四)单元测试

美团技术团队单元测试介绍 --- Roboelctric

腾讯TMQ单元测试介绍及使用 --- mockito、PowerMockito --- 可以配置增加Jacoco覆盖率

Android单元测试在复杂项目里的落地姿势（PowerMock实践篇）

Android 单元测试实战（1）—— 调研与选型

单元测试的使用 --- 查看测试报告