Android单元测试方案

> 最终方案：Junit + PowerMock + Robolectric + Espresso

Android单元测试简介

我们写程序的时候，其实大部分时间不是花在写代码上面，而是花在debug上面，是花在找出问题到底出在哪上面，而单元测试可以最快的发现你的新代码哪里出问题，也就可以快速的定位到问题所在，然后给以及时的解决，这也可以在很大程度上防止regression bug（回归缺陷）。

Android的单元测试分为两大类：

1. Unit Test

通过JUnit，以及第三方测试框架，我们可以编写测试代码，生成class文件，直接运行在JVM虚拟机中。

优点： 很快。使用简单，方便。

缺点： 不够逼真。比如有些硬件相关的问题，无法通过这些测试出来。

代表框架： `JUnit(标准)`, `Robolectric`, `mockito`, `powermock`

2. Instrumentation Test

通过Android系统的Instrumentation测试框架，我们可以编写测试代码，并且打包成APK，运行在Android手机上。

优点： 逼真

缺点： 很慢

代表框架： `JUnit` (Android自带), `espresso`, `Android`

单元测试方案简介

在本方案中同时采用两种单元测试方式，其中：

Unit Test方案：Junit + PowerMock + Robolectric

`Robolectric`通过实现一套`JVM`能运行的Android代码，从而做到脱离Android环境进行测试。在`unit test`运行的时候去截取android相关的代码调用，然后转到他们的他们实现的代码去执行这个调用的过程。因为不需要再真机上运行，所以测试速度相较于Instrumentation Test要快很多。

Robolectric是个非常强大好用的单元测试框架。虽然使用的过程中肯定也会遇到问题，我个人就遇到不少问题，尤其是跟第三方的library比如Retrofit、ActiveAndroid结合使用的时候，会有不少问题，但瑕不掩瑜，我们依然可以用它完成很大部分的unit testing工作。

Robolectric 3.1（目前最先版本为3.2.2）已支持针对非AndroidSdk的类做Shadow，但是不支持Powermock。如果使用3.0的robolectric 就可以支持Powermock, 如果选择3.1以上的版本的robolectric 就可以支持非AndroidSdk的类的Shadow。本方案中采用Robolectric3.2.2进行Android相关的测试。

使用场景：
纯Java类，使用JUnit和PowerMock进行测试;
与Android环境相关的类，使用Robolectric3.2.2测试。（也可以采用Robolectric3.0 + PowerMock）

Instrumentation Test方案：Espresso

`Espresso`作为Google推出的Instrumentation UI测试框架，在API支持方面有着天然的优势。相对于`Robotium`和`UIAutomator`，它的特点是规模更小、更简洁，API更加精确，编写测试代码简单，容易快速上手。但是Espresso测试必须运行`emulator`或者是真机上面，所以这是个很慢的过程，因为要打包、dexing、上传到机器、运行起来界面。以这个速度是没有办法用于快速开发的。

Espresso是Google官方推出的一款为Android开发人员提供的一款功能强大的UI测试框架。其为开发人员提供了一组API来构建UI测试，开发人员可以使用这些API编写简洁、运行可靠的自动化UI测试。Espresso适合用来编写白盒自动化测试，测试代码将利用测试应用的实现来完成测试流程。

Espresso 测试框架的主要功能：

- **灵活的 API** ：用于目标应用中的视图和适配器匹配；
- **丰富的操作 API** ：用于自动化 UI 交互；
- **UI 线程同步** ：用于提升测试可靠性。

其缺点是运行时需要真机支持，而且需要安装真实app（真实逻辑）以支持相关测试操作。所以，如果真实app有改动，就需要重新编译打包安装，导致测试时间漫长。
Espresso只支持API 8（Android 2.2）以上。

使用场景：测试UI，以及在提测后进行功能测试

Junit + PowerMock + Robolectric

Robolectric

添加依赖

```
dependencies {
    testCompile "org.robolectric:robolectric:3.2.2"
    //robolectric针对support-v4的shadows
    testCompile "org.robolectric:shadows-support-v4:3.0"
}
```

Robolectric在第一次运行时，会下载一些sdk依赖包，每个sdk依赖包至少50M，而 https://oss.sonatype.org 服务器比较慢，导致下载速度非常慢。所以最好手动下载所需要的文件。例如,需要下载以下文件：

```
Downloading: org/robolectric/android-all/6.0.1_r3-robolectric-0/and
roid-all-6.0.1_r3-robolectric-0.jar from repository sonatype at htt
ps://oss.sonatype.org/content/groups/public/
Transferring 56874K from sonatype
```

解决方案：

1. 从http://repo1.maven.org/maven2/org/robolectric/android-all/6.0.1_r3-robolectric-0/android-all-6.0.1_r3-robolectric-0.jar中下载 `android-all-6.0.1_r3-robolectric-0.jar`
2. 将jar文件放置在本地maven仓库地址中，例如：`C:\Users\Administrator\.m2\repository\org\robolectric\android-all\6.0.1_r3-robolectric-0`

重写TestRunner

在非app的module中添加Robolectric测试框架时，很可能会出现以下异常：

> java.lang.RuntimeException: build\intermediates\bundles\debug\AndroidManifest.xml not found or not a file; it should point to your project's AndroidManifest.

发生此异常，是因为无法正常获取AndroidManifest文件。此时就需要重写TestRunner,可以从`RobolectricGradleTestRunner`继承，然后覆盖`getAppManifest`方法，构建自己想要的特殊的`AndroidManifest`对象来实现对AndroidManifest,assets,res资源的控制。

```
public class MyRobolectricTestRunner extends RobolectricTestRunner
{
    ... ...
    @Override
    protected AndroidManifest getAppManifest(Config config) {
        int nameLength = projectName.length();
        String rootPath = System.getProperty("user.dir", "./");
        int index  = rootPath.indexOf(projectName);
        if (index == -1) {
            throw new RuntimeException("project name not found in u
ser.dir");
        }
        //获取项目的根目录
        rootPath = rootPath.substring(0, index + nameLength);
        String manifestProperty = rootPath + "/newhousesdk/src/mai
n/AndroidManifest.xml";
        String resProperty = rootPath + "/newhousesdk/src/main/re
s";
        String assetsProperty = rootPath + "/newhousesdk/src/main/a
ssets";
        return new AndroidManifest(
            Fs.fileFromPath(manifestProperty),
            Fs.fileFromPath(resProperty),
            Fs.fileFromPath(assetsProperty)) {
            @Override
            public int getTargetSdkVersion() {
                return MAX_SDK_SUPPORTED_BY_ROBOLECTRIC;
            }
        };
    }
}
```

隔离原Application依赖

如果用 Robolectric 单元测试，不配置Application ，就会调用原来的项目的Application，而 App 有很多第三方库依赖。于是，执行 App 生命周期时， robolectric 就容易报错。
正确配置 Application 方式：

1. 自定义一个用于单元测试的 RoboApplication.class
2. 配置Application。有以下两种方式：

方式一：在单元测试 XXTest 加上 @Config(application = RoboApplication.class) 。

```
@Config(application = RoboApplication.class)
public class XXXTest { }
```

方式二：在TestRunner中设置Application

```
@Override
protected Config buildGlobalConfig() {
    return new Config.Builder()
        .setApplication(RoboApplication.class)
        .build();
}
```

日志输出

我们在写UT的过程，其实也是在调试代码，而日志输出对于代码调试起到极大的作用。而`Robolectric`对日志输出的支持其实非常简单。只需要在每个TestCase的setUp( )方法或者Application的onCreate()方法中中添加一句命令：

```
@Before
public void setUp() throws URISyntaxException {
    //输出日志
    ShadowLog.stream = System.out;
}
```

此时，无论是功能代码还是测试代码中的 Log.i()之类的相关日志都将输出在控制面板中。

单元测试示例

测试代码是放在`app/src/test`下面的，测试类的位置最好跟被测试类的位置对应，比如MainActivity放在`app/src/main/java/com/robo/test/MainActivity.java`那么对应的测试类MainActivityTest最好放在`app/src/test/java/com//robo/test/MainActivityTest.java`

下面以XfConsultantProfileActivity类的测试为例，进行简单的介绍：

新建Unit Test

通过注解配置TestRunner等基本信息

```
@RunWith(MyRobolectricTestRunner.class)
@Config(constants = BuildConfig.class)
public class XfConsultantProfileActivityUnitTest {
    private XfConsultantProfileActivity mActivity;

    @Before //在运行test之前，进行一些初始化操作
    public void setUp() throws Exception {
    }

    @Test //测试方法
    public void testFragment() throws InterruptedException {
    }
}
```

启动Activity

可以通过`setupActivity()`方法来直接启动Activity

```
mActivity = Robolectric.setupActivity(XfConsultantProfileActivity.c
lass);
```

如果要启动的Activity需要从Intent中获取额外的数据，那么就需要使用`buildActivity()`方法来获取`ActivityController`。通过`ActivityController`，不仅可以设置Activity的Intent，还可以创建Activity和控制Activity的生命周期。

```
Intent intent = new Intent();
intent.putExtra("consultant_id", (long)2176270);
intent.putExtra(ConstantsSetting.LAST_PAGE_PARAM, "");
intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
intent.setClass(RuntimeEnvironment.application, XFHouseDetailActivi
ty.class);
mActivity = Robolectric.buildActivity(XfConsultantProfileActivity.c
lass)
                .withIntent(intent).setup().get();
```

验证Fragment

通过Activity对象获取Fragment：

```
//获取Fragment
List<Fragment> fragmentList = mActivity.getSupportFragmentManage
r().getFragments();
XfConsultantProfileFragment mFragment = null;
if (fragmentList.get(0) instanceof XfConsultantProfileFragment) {
    mFragment = (XfConsultantProfileFragment) fragmentList.get(0);
}
assertNotNull(mFragment);
```

也可以脱离Activity，直接启动Fragment实体。对于`android.app.Fragment`直接使用`SupportFragmentTestUtil`的`startFragment`方法启动Fragment。

```
XfConsultantProfileFragment mFragment = XfConsultantProfileFragment
            .getInstance((long)2176270);
FragmentTestUtil.startFragment(mFragment);
assertNotNull(mFragment.getView());
```

而对于`support-v4`的Fragment需要使用`SupportFragmentTestUtil`的`startFragment`方法启动Fragment，且需要添加相应的依赖：

```
dependencies {
    //robolectric针对support-v4的shadows
    testCompile "org.robolectric:shadows-support-v4:3.0"
}
```

验证ListView

```
//执行ListView的Item点击事件
 ListView listView = (ListView) mFragment.findViewById(R.id.listvie
w);
 View view = listView.getAdapter().getView(1,null,null);
 listView.performItemClick(view,0,0);
```

处理网络回调

在后台线程中请求网络，请求完成后在UI线程里通过Listener接口通知请求完成，并传递请求回来的数据。这时需要如何处理呢？

在使用`Robolectric`框架测试需要在UI线程执行的逻辑时，在Android平台UI线程会轮询消息队列，然后从消息队列里取出消息，并将消息分发给Handler处理，UI线程执行的是轮询消息队列的死循环。但是在`Robolectric`框架中运行时，UI线程默认情况下并不会轮询消息队列，而需要在测试用例代码里主动驱动`UI线程`从消息队列里取出消息进行分发。测试用例执行时并不在UI线程，而是在单独的线程中，所以它可以主动驱动UI线程分发消息。

所以在执行网络请求后，需要主要驱动UI线程轮询消息队列，获取返回的数据。从下面的代码可以看到我们可以通过获取Scheduler对象来判断消息队列中是否有消息，并调用Scheduler的runOneTask方法进行消息分发，这样就驱动了主线程进行消息轮询，

```
//获取主线程的消息队列的调度者，通过它可以知道消息队列的情况
//并驱动主线程主动轮询消息队列
Scheduler scheduler = Robolectric.getForegroundThreadScheduler();
//因为调用请求方法后 后台线程请求需要一段时间才能请求完毕，然后才会通知主线程
// 所以在这里进行等待，直到消息队列里存在消息
while (scheduler.size() == 0) {
    Thread.sleep(500);
}
//轮询消息队列，这样就会在主线程进行通知
scheduler.runOneTask();
```

Shadow

`Shadow`是Robolectric的立足之本。因此，框架针对Android SDK中的对象，提供了很多`Shadow`对象（如Activity和ShadowActivity、TextView和ShadowTextView等），这些`Shadow`对象，丰富了本尊的行为，能更方便的对Android相关的对象进行测试。从Robolectric 3.1开始已支持针对非Android SDK的类构建Shadow了。

1. 使用框架提供的Shadow对象

```
//通过Shadows.shadowOf()可以获取很多Android对象的Shadow对象
ShadowListView shadowListView = Shadows.shadowOf(listView);
shadowListView.performItemClick(0);
```

2. 如何自定义Shadow对象

以User类为例，创建自定义 Shadow对象。

```
@Implements(User.class) //原始对象
public class ShadowUser {
    @Implementation //重新实现原始对象中的方法
    public long getUserId() {
        return (long)70652;
    }
}
```

接下来，需将定义好的Shadow对象，在TestRunner进行设置

```
private static final Class<?> mShadowClass[] = {ShadowUser.class};

@Override
protected Config buildGlobalConfig() {
    return new Config.Builder()
        .setShadows(mShadowClass) //设置shadows
        .build();
}
```

PowerMock的使用

添加依赖：

```
 testCompile 'junit:junit:4.12'
 testCompile "org.powermock:powermock-module-junit4:1.6.4"
 testCompile "org.powermock:powermock-module-junit4-rule:1.6.4"
 testCompile "org.powermock:powermock-api-mockito:1.6.4"
 testCompile "org.powermock:powermock-classloading-xstream:1.6.4"
```

所谓的`mock`就是创建一个类的虚假的对象，在测试环境中，用来替换掉真实的对象，以达到两大目的：

- 验证这个对象的某些方法的调用情况，调用了多少次，参数是什么等等
- 指定这个对象的某些方法的行为，返回特定的值，或者是执行特定的动作

关于Mockito以及PowerMock的具体使用，网上有很多资料，在这里就不多介绍了。只强调一下在使用Mock时需要注意的两点是：

- Mockito.mock()并不是mock一整个类，而是根据传进去的一个类，mock出属于这个类的一个对象，并且返回这个mock对象；而传进去的这个类本身并没有改变，用这个类new出来的对象也没有受到任何改变！
- mock出来的对象并不会自动替换掉正式代码里面的对象，你必须要有某种方式把mock对象应用到正式代码里面

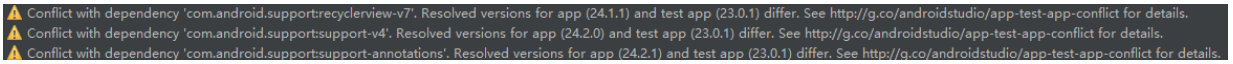
Espresso

开始使用Espresso

在使用Espresso构建相关UI测试之前，我们需要引入相关依赖包。在app项目**app\src**目录下建立**androidTest\java**目录,作为UI测试代码的独立目录（默认自动建立）,同时，我们也可以将测试所需的资源放入到该目录的对应目录下。然后再build.gradle中添加如下依赖：

```
androidTestCompile 'com.android.support.test:runner:0.5'
// Set this dependency to use JUnit 4 rules
androidTestCompile 'com.android.support.test:rules:0.5'
// Set this dependency to build and run Espresso tests
androidTestCompile 'com.android.support.test.espresso:espresso-cor
e:2.2.2'
// Set this dependency to build and run UI Automator tests
androidTestCompile 'com.android.support.test.uiautomator:uiautomato
r-v18:2.1.2'
// set this dependency to test recyclerView
androidTestCompile 'com.android.support.test.espresso:espresso-cont
rib:2.2.2'
// set this dependency to test intent
androidTestCompile 'com.android.support.test.espresso:espresso-inte
nts:2.2.2'
```

点击同步后，可能会发生依赖冲突，如下图：



这是因为引入的Epresso依赖包中使用到的某些library包和我们主项目中版本冲突，这时候就需要添加**强制依赖**来解决，在build.gradle中根据上述冲突所提示的版本加入如下代码：

```
configurations.all {
    resolutionStrategy.force 'com.android.support:recyclerview-v7:2
3.1.1'
    resolutionStrategy.force 'com.android.support:design:23.1.1'
    resolutionStrategy.force 'com.android.support:appcompat-v7:2
3.1.1'
}
```

再次点击同步后，还有可能发生问题，具体问题如下：

这个问题的意思是引入的架包**com.android.support.test.uiautomator:uiautomator-v18:2.1.2**最低支持的API版本是18，而实际项目的最低支持是15。
此时我们需要在**androidTest**目录下建立一个独立的测试AndroidMainfest文件，然后使用**tools:overrideLibrary="android.support.test.uiautomator.v18"**去override最低可支持的API版本。具体如下：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/androi
d"
          xmlns:tools="http://schemas.android.com/tools"
          package="your package name">

    <uses-sdk android:targetSdkVersion="24"
              android:minSdkVersion="14"/>

    <application
            android:allowBackup="true"
            android:supportsRtl="true">

    </application>
</manifest>
```

如果建立了对应的AndroidMainfest文件后还是报同样问题，**clean一下工程**。

Espresso两个重要的类

在进行测试时，需要配合与**JUnit**一起使用。
在测试类中都需要使用其`IntentsTestRule`或`ActivityTestRule`与`Unit`的`@Rule`注解来使用。它会在每一个被 `@Test` 注解的测试执行前初始化`Espresso-Intents/Espresso-Activity`，然后在测试执行完后释放`Espresso-Intents/Espresso-Activity`。被启动的 activity 会在每个测试执行完后被终止掉。

ActivityTestRule

从名字中可以看出来，是用来启动一个Activity的。其具体使用如下：

```
@Rule
public ActivityTestRule mActivityTestRule = new ActivityTestRule<>
(UITestActivity.class);
```

通过上述代码，我们可以在每个测试方法开始执行之前，启动指定的activity.同时，我们也可以通过上面的`mActivityTestRule`拿到启动的activity实例，如下代码：

```
Activity activity = mActivityTestRule.getActivity();
```

如果是要启动Intent中带有额外数据的Activity，这是我们要重写**ActivityTestRule**的`getActivityIntent()`方法，具体实现如下：

```
 @Rule
 public ActivityTestRule mIntentsTestRule = new ActivityTestRule<UI
TestActivity3>(UITestActivity3.class) {
        @Override
        protected Intent getActivityIntent() {
            Context targetContext = InstrumentationRegistry.getInst
rumentation().getTargetContext();
            Intent result = new Intent(targetContext, MainActivit
y.class);
            result.putExtra("data", "测试数据");
            return result;
        }
    };
```

IntentsTestRule

用一个Espresso-Intent来启动一个activity，可以用来测试Activity中的`onActivityResult()`方法的。其具体使用如下：

```
@Rule
public IntentsTestRule mIntentsTestRule = new IntentsTestRule<>(UIT
estActivity.class);
```

然后在对应的`@Test`注解修饰的方法中使用如下代码添加

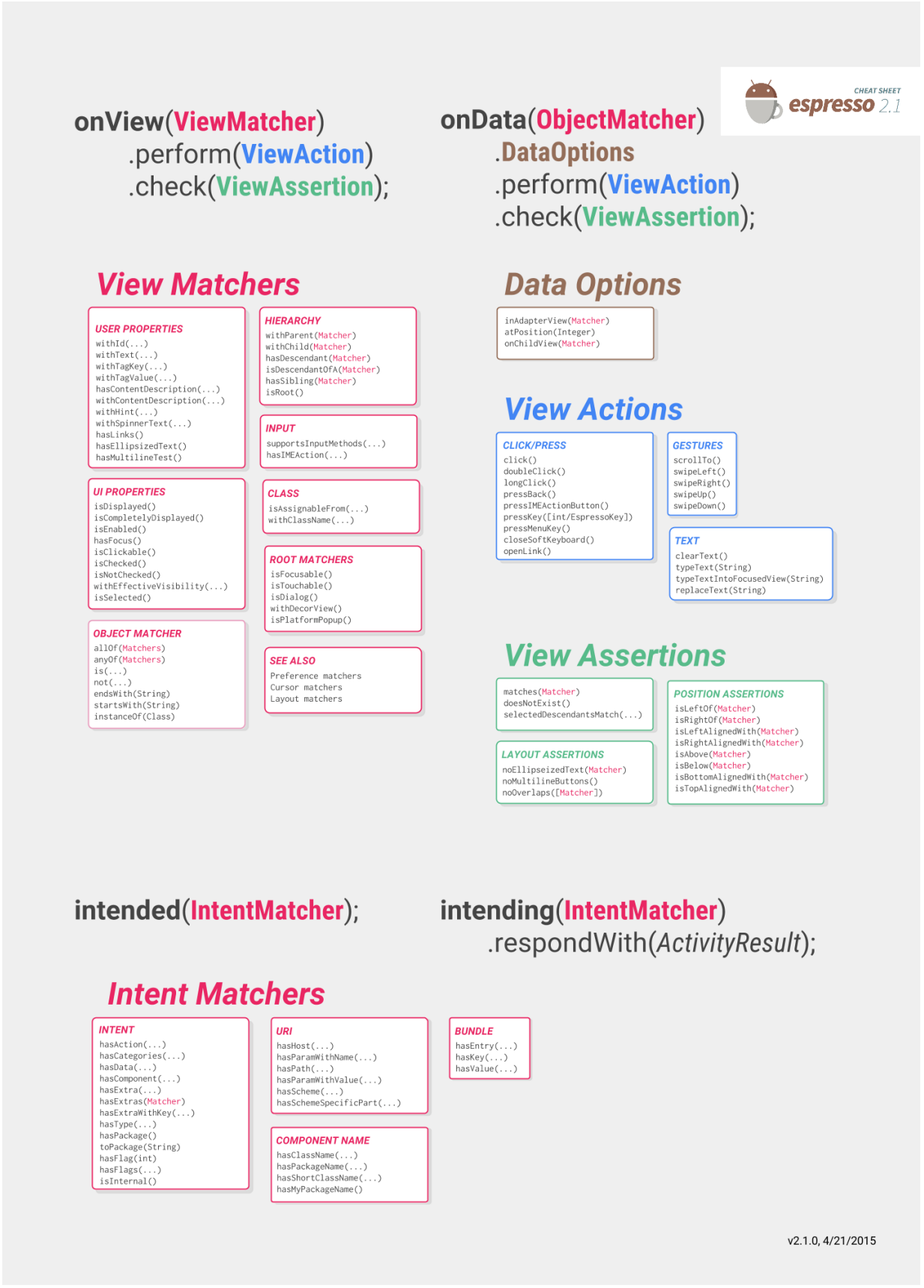
```
 @Test
 public void setTextTest1() throws Exception {
        Intent resultData = new Intent();
        String data = "123-345-6789";
        resultData.putExtra("data", data);
        Instrumentation.ActivityResult result = new Instrumentatio
n.ActivityResult(
            Activity.RESULT_OK, resultData);

        intending(allOf(toPackage("com.example.hanxu.weixindemo"), h
asExtra(any(String.class), any(String.class)))).respondWith(resul
t);
        onView(withId(R.id.tv_test)).check(matches(isClickable()));
    }
```

这里的`intending`和另外一个未使用的`intended`都是用来断言的。但其中的区别是`intended`只会按照你所设置的断言条件去验证intent是否`launched`，而`intending`则除此之外还会返回一个`result`。
其也可以通过重写`getActivityIntent()`方法来给启动的Activity塞入额外的数据。

Espresso的具体使用

所以Espresso提供的API都如下图所示：



onView

`onView(final Matcher<View> viewMatcher)`:根据指定的条件来找到指定的view。

```
onView(withId(R.id.tv_test));
onView(withText("测试数据"));
// 还可以使用allOf进行对复数条件进行验证
onView(allOf(withText("测试数据"), withId(R.id.et_name)));
```

perform

`perform(final ViewAction... viewActions)`：给指定View进行指定的操作。通常是放在`onView()`或`onData()`方法之后配合使用。

```
// 点击事件 如果是不可点击控件，将会抛出异常，导致测试失败
onView(withId(R.id.tv_test)).perform(click());
```

check

`check(final ViewAssertion viewAssert)`：对指定的view进行指定断言的check操作。通常是放在`onView()`或`onData()`或`perform()`方法之后配合使用。

```
onView(withId(R.id.et_name)).check(matches(withText(name)));
onView(withId(R.id.tv_test2)).check(matches(isClickable()));
// 同样还可以使用allOf进行对复数条件进行验证
onView(withId(R.id.tv_test)).perform(click()).check(matches(allOf(i
sClickable(), hasFocus())));
```

对于最基础的控件，如`TextView、EditView、Button`等，使用上述三个方法就能完成对其几乎所有操作的测试。但是对于`AdapterView`和`RecyclerView`等控件来说，这几个方法还不足以完成对其的测试，甚至根本不能找到这些控件。这里就需要使用到`Espresso`的其他方法。

onData

对与`AdapterView`来说，由于其是使用`Adapter`来动态加载数据，并且大部分时间都是只有其中一部分显示在屏幕上，对于那些没有显示的view，我们无法通过onView找到他。
所以这个时候就需要使用到`onData(Matcher<? extends Object> dataMatcher)`来完成通过`data`来完成对特定`View`的锁定。在这里的使用，对自定义`Matcher`的使用格外重要。其具体使用如下：

```
onData(allOf(is(instanceOf(MyUiTestEntity.class)), myUiTestMatche
r("0")))
        .inAdapterView(withId(R.id.rv_content))
        .atPosition(99)
        .onChildView(withId(R.id.tv_item))
        .perform(click());
```

这里是对布局中一个指定id的ListView进行找到**指定数据和指定位置**的Item，然后拿到其ItemView的**指定id控件**进行一个click操作。其中使用到了一个自定义Matcher,其定义如下：

```
public static Matcher<Object> myUiTestMatcher(final String key) {
        return new BoundedMatcher<Object, MyUiTestEntity>(MyUiTestEn
tity.class) {
            // 匹配条件设置
            @Override
            protected boolean matchesSafely(MyUiTestEntity item) {
                return item != null
                        && !TextUtils.isEmpty(item.data)
                        && item.data.equals(key);
            }

            // 相关描述设置
            @Override
            public void describeTo(Description description) {
                description.appendText("RecyclerView has item :" + k
ey);
            }
        };
    }
```

RecyclerView的测试

而对于`RecyclerView`来说，由于他使用的是一个`RecycledViewPooll`来进行`ViewHolder`的复用，不属于`AdapterView`，所以无法使用`onData`去操作他。

我们可以使用以下两种办法去对其进行测试：

1. 导入`android.support.test.espresso.contrib`包，使用其中的`RecyclerViewActions`来对其执行指定操作。
2. 自定义RecyclerViewMatcher以及相应的ViewActions来完成与上述方法一致的功能。

对于第一种方法,我们需要在`perform()`中调用`RecyclerViewActions`的相关方法来进行操作。其具体使用如下：

```
onView(withId(R.id.rv_content)).perform(RecyclerViewActions.scrollT
oPosition(80));
onView(withId(R.id.rv_content)).perform(RecyclerViewActions.scrollT
o(itemMatcher));
// 其他操作请自行查阅相关API
```

对于第二种方法，我们不需要额外架包的引用，通过完成自定义Matcher也能完成上述操作。自定义一个`RecyclerViewMatcher`来找到一个RecyclerView，并返回根据设置的`position`对应的ItemView或其ItemView中的某个指定id的控件。但这只实现了对指定位置item进行操作,没有进行滚动操作，所以当此Item并没有在屏幕中显示出来时，将会抛出异常，导致测试失败。所以针对滚动，我们再定义一个滚动操作`actionOnItemViewAtPosition`。

对于异步操作的支持

Espresso也是能够测试网络请求等异步耗时操作，其需要使用到叫**IdlingResource**的接口来判断初始化的Activty是否还有还有耗时操作。如果**无耗时操作**，再去执行测试代码。
首先自定义一个**IdlingResource**，其具体实现如下：

```
public class MyIdlingResource implements IdlingResource {

    private ResourceCallback mCallback = null;
    private UITestActivity mActivity;

    public MyIdlingResource(UITestActivity activity) {
        mActivity = activity;
    }

    @Override
    public String getName() {
        return "MyIdlingResource";
    }

    // 是否还有超时操作的判断 这里的mActivity.isSyncFinished()需要在异步操
作的回调中使用
    @Override
    public boolean isIdleNow() {
        boolean isIdle = mActivity != null && mActivity.isSyncFinis
hed();
        if (isIdle && mCallback != null) {
            mCallback.onTransitionToIdle();
        }
        return isIdle;
    }

    @Override
    public void registerIdleTransitionCallback(ResourceCallback cal
lback) {
        mCallback = callback;
    }
}
```

然后在测试代码中使用：

```
@RunWith(AndroidJUnit4.class)
@LargeTest
public class UITestActivityTest {

    String name;
    MyIdlingResource idlingResource;
    Activity activity;

    @Rule
    public ActivityTestRule mActivityTestRule = new ActivityTestRul
e<>(UITestActivity.class);

    @Before
    public void setUp() throws Exception {
        name = "my ui test";
        activity = mActivityTestRule.getActivity();
        idlingResource = new MyIdlingResource((UITestActivity) acti
vity);
        // 注册idlingResource
        Espresso.registerIdlingResources(idlingResource);
    }

    @After
    public void unregisterIntentServiceIdlingResource() throws Exce
ption{
        // 反注册idlingResource
        Espresso.unregisterIdlingResources(idlingResource);
    }

    @Test
    public void setTextTest1() throws Exception {
        // 只有在异步操作结束后才回去执行测试代码
        onView(withId(R.id.tv_test)).check(matches(hasFocus()));
    }
}
```

方案总结

在开发过程中采用Unit Test对程序进行快速的测试，

- 对于纯Java类，使用JUnit和PowerMock进行测试（例如接口调用）；
- 测试Android相关(Activity, Context, View等)的类使用Robolectric3.2.2测试。

在完成基本功能以后，可以使用Espresso进行UI测试和功能测试。

参考文档

Android单元测试框架Robolectric3.0介绍
Junit + Mockito + Powermock
Mockito的使用
Espresso浅析和使用
Android单元测试研究与实践