Android 单元测试实践

什么是单元测试

在计算机编程中，单元测试（Unit Testing）又称为模块测试, 是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件。但是什么叫"程序单元"呢？是一个模块、还是一个类、还是一个方法（函数）呢？不同的人、不同的语言，都有不同的理解。一般的定义，尤其是是在OOP领域，是一个类的一个方法。在此，我们也这样理解：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，而写的测试代码。

单元测试的三个步骤：

- setup：即新建出待测试的类、设置一些前提条件等
- 执行动作：即调用被测类的被测方法，并获取返回结果
- 验证结果：验证获取的结果跟预期的结果是一样的

单元测试不是集成测试

这里需要强调一个观念，那就是单元测试只是测试一个方法单元，它不是测试一整个流程。举个例子来说，一个Login页面，上面有两个输入框和一个button。两个输入框分别用于输入用户名和密码。点击button以后，有一个UserManager会去执行performlogin操作，然后将结果返回，更新页面。那么我们给这个东西做单元测试的时候，不是测这一整个login流程。这种整个流程的测试：给两个输入框设置正确的用户名和密码，点击login button，最后页面得到更新。叫做集成测试，而不是单元测试。当然，集成测试也是有他的必要性的，然而这不是每个程序员应该花多少精力所在的地方。为什么是这样呢？因为集成测试设置起来很麻烦，运行起来很慢，在保证代码质量、改善代码设计方面更起不到任何作用，因此它的重要程度并不是那么高

为什么要做单元测试

- 也许是因为换了工作，也许是因为职位调动，或其他原因，需要接手新的项目，当拿到一个新项目的时候，会有一种诚惶诚恐的感觉，因为一时间比较难理清楚整个app的结构是怎么划分的，各部分各模块之间又是什么样的关系。会怕改了某一个地方，结果其他一个莫名其妙的地方的受到了影响，然后导致了一个

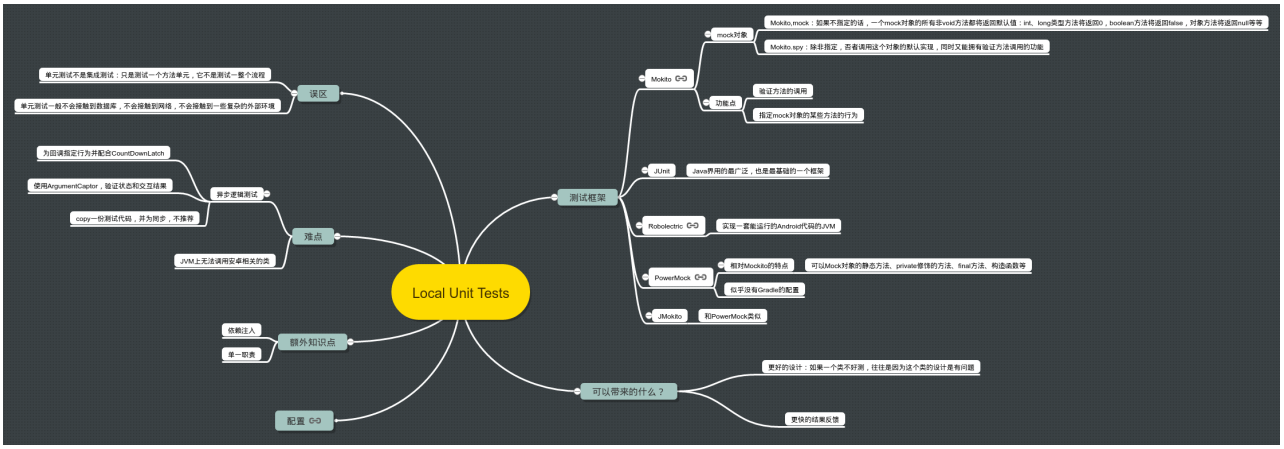
bug。所以，那种时候就会希望，如果我改了某个地方，能有个东西告诉我，这个改动影响到哪些地方，这样改是不是有问题的，会不会导致bug。虽然我可以把app启动起来，看看是不是能正常工作，然而一种case能工作，并不代表所有影响到的case都能工作，例如，一个密码设置功能，可能有不同的入口，首次启动的密码设置，设置界面的修改密码、设置界面的重置密码或者还有只有会员才有的增强密码设置，这样我就可能需要安装APP，进入不同的入口进行操作来验证这段代码是否正确，这也是一个很痛苦很费时间的过程，而且很多的外部条件也很难满足，比如说需要什么样的网络条件，需要用户是会员等等。尤其是在不知道有哪些地方用到了的情况下，我更加难以去遍历所有用到的地方，一个一个去验证这个改动有没有问题。在这种情况下，单元测试是才是最好的工具。首先，单元测试只是针对一个代码单元写的测试，保证一个代码单元的正确性总比保证整个app的正确性容易吧？遍历一个方法的所有参数和输出情况总比遍历一个app的所有用户场景容易吧？跑一次单元测试总比运行一次app快吧？

- 在不写单元测试的情况下，我们的开发流程可能是这样的，要把整个功能都做完整，从model到controller(或Presenter、ViewModel）到view到util等等，一整套流程做下来，到最后才可能运行起来看看是不是对的，有的时候哪怕所有代码都写完了，也不一定能验证是不是对的，比如说后台还没有ready等等。总之，在没有单元测试的情况下，我们需要等到最后一刻才能手动验证代码是不是对的，然后发现原来这里错了一点，那里少了一点，然后一遍一遍的把app运行起来，改一点运行一遍。。。当开始写单元测试之后，可以在写完一部分功能独立的代码后，就能立刻看到他们是不是正确的。如果不是的话，我可以立刻就改正，而不用等到所有代码都写完整

- 不熟悉你的代码的人人看你的代码的时候，如果有单元测试代码，对别人理解你的代码会有一定的促进作用，因为单元测试代码里面一般会包含了你当时设计这部分模块时的思路

Local Unit Tests

Local Unit Tests运行在本地JVM，不需要安装APP，所以运行时间很快。也因此不能依赖Android的API，所以大多数时候需要用Mock的形式来做替换（后面会提到）



配置

- 测试代码目录：(src/test/java)
- 一般使用到的测试框架
  - JUnit4
  - Mockito

使用

JUnit4

这是Java界用的最广泛，也是最基础的一个框架，其他的很多框架，包括我们后面会看到的Mockito，都是基于或兼容JUnit4的。使用比较简单，最多的是其`Assert`类提供的`assertXXX`方法。

假设这样的一个类需要测试

```
public class Calculator {

  public int add(int one, int another) {
    return one + another;
  }

  public int multiply(int one, int another) {
    return one * another;
  }
```

如果不使用单元测试框架，我们可能需要这样来验证这段代码的正确性：

```
public class CalculatorTest {
  public static void main(String[] args) {
```

```
        Calculator calculator = new Calculator();
        int sum = calculator.add(1, 2);
        if(sum == 3) {
            System.out.println("add() works!")
        } else {
            System.out.println("add() does not works!")
        }

        int product = calculator.multiply(2, 4);
        if (product == 8) {
            System.out.println("multiply() works!")
        } else {
            System.out.println("multiply() does not works!")
        }
    }
}
```

然后我们再通过某种方式，比如命令行或IDE，运行这个`CalculatorTest`的`main`方法，在看着terminal的输出，才知道测试是通过还是失败。想一下，如果我们有很多的类，每个类都有很多方法，那么就要写一堆这样 的代码，每个类对于一个含有main方法的test类，同时main方法里面会有一堆代码。这样既写起来痛苦，跑起来更痛苦，比如说，你怎么样一次性跑所有的测试类呢？所以，一个测试框架为我们做的最基本的事情，就是允许我们按照某种更简单的方式写测试代码，把每一个测试单元写在一个测试方法里面，然后它会自动找出所有的测试方法，并且根据你的需要，运行所有的测试方法，或者是运行单个测试方法，或者是运行部分测试方法等等。对于上面的例子，如果使用Junit的话，我们可以按照如下的方式写测试代码：

```
public class CalculatorTest {
    Calculator mCalculator;

    @Before
    public void setup() {
        mCalculator = new Calculator();
    }

    @Test
    public void testAdd() throws Exception {
        int sum = calculator.add(1, 2);
        Assert.assertEquals(3, sum);
    }

    @Test
    public void testMultiply() throws Exception {
        int product = calculator.multiply(2, 4);
        Assert.assertEquals(8, product);
    }

}
```

上面的`@Before`修饰的方法，会在测试开始前调用，这里是新建了一个`Calculator`对象，所以之后的一些测试单元都可以直接使用这个实例，`@Test`修饰的方法，是用来需要测试的单元，例如`testAdd`方法是用来测试`Calculator`类的加法操作，测试单元内使用`Assert`类提供的`assertXXX`方法来验证结果。如果还有其他的测试方法，则以此类推。另外还有一些可能需要用到的修饰符，如`@After`，`@BeforeClass`，`@AfterClass`等。

Mockito

Mockito的两个重要的功能是，验证Mock对象的方法的调用和可以指定mock对象的某些方法的行为。（对于不懂Mock概念的同学来说，第一次看到的确很可能很难理解）

为什么要使用Mockito？

这是项目中的一个例子：

```
/**
 * @param <T> 用于过滤的实体类型
 */
public interface BaseItemFilter <T> {
    /**
     * @param item
     * @return true：不过滤;false：需要过滤
     */
    boolean accept(T item);

}
```

`BaseItemFilter`是用来判断某种指定类型的实体是否需要过滤的，类似java中的`FileFilter`，目的是为了用了过滤不符合要求的实体。

以下是我们的关键服务过滤器的实现：

```
public class EssentialProcessFilter implements BaseItemFilter< RunningAppBean > {

    /**
     * 系统关键进程及用户主要的进程
     */
    private static HashSet< String > sCoreList = new HashSet< String >();

    /**
     * 加载系统核心进程列表
     * @param context
     */
    public static void loadCoreList(Context context) {
```

```
        if (sCoreList.isEmpty()) {
            final Resources r = context.getResources();
            String[] corePackages = r.getStringArray(R.array.default_core_list);
            Collections.addAll(sCoreList, corePackages);
        }
    }

    @Override
    public boolean accept(RunningAppBean appModle) {
        return appModle != null && !(isEssentialProcess(appModle.mPackageName) || isEs
    }

    /**
     * 判断进程是否属于重要进程
     * @param process
     * @return
     */
    public static boolean isEssentialProcess(String process) {
        return sCoreList.contains(process);
    }

    /**
     * 系统关键进程关键词模糊匹配
     * @param packageName
     * @param isSystemApp
     * @return
     */
    public static boolean isEssentialProcessMock(String packageName, boolean isSystemApp
        return 省略...额外的一些判断;
    }

}
```

可以看到，这里的关键服务的判断的判断规则可以分两部分，一个是从String.xml中预设的一段Arrays数组查找是否右符合的，这个需要在初始化或某个时机预先调用EssentialProcessFilter#loadCoreList（Context context）方法来加载，另外的一个判断是在EssentialProcessFilter#isEssentialProcessMock方法中定义，这个类中accept方法，定义了只要符合其中一种规则，那么我们就需要把它过滤。

这个时候我们来写单元测试，你一开始就会发现你没有办法新建一个Context对象来读取String.xml，即使你想尽任何方法新建一个ContextImpl实例，最后你还是会出错的，主要原因再在于Gradle运行**Local Unit Test**所使用的android.jar里面所有API都是空实现，并抛出异常的。现在想想，我们实际上并不需要真的读取String.xml，我们需要验证的是记录在我们的关键列表集合是否生效，既然这样，我们前面说过了，Mockito的两个重要的功能是，验证Mock对象的方法的调用和可以指定mock对象的某些方法的行为。我们是否可以Mock一个Context对象并且指定它读取String.xml的行为？答案

是可以的，如下就是使用Mockito的一段测试代码

```
public class TestListFilter2 {
    @Mock
    Context mContext;
    @Mock
    Resources mResources;

    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
        Mockito.when(mContext.getResources()).thenReturn(mResources);
        Mockito.when(mResources.getStringArray(R.array.default_core_list)).thenReturn(
        //模拟加载XML资源
        EssentialProcessFilter.loadCoreList(mContext);
    }

    /**
     * 测试关键服务的过滤器
     */
    @Test
    public void testEssentialFilter() {
        EssentialProcessFilter processFilter = new EssentialProcessFilter();
        ListFilter<RunningAppBean> listFilter = Mockito.spy(ListFilter.class);
        listFilter.addFilter(processFilter);
        List<RunningAppBean> list = new ArrayList<RunningAppBean>();
        list.addAll(getEssentialAppBean());
        list.addAll(getNormalRunningApp());
        List<RunningAppBean> result = Mockito.mock(ArrayList.class);
        for (RunningAppBean runningAppBean : list) {
            if (listFilter.accept(runningAppBean)) {
                result.add(runningAppBean);
            }
        }
        Mockito.verify(listFilter, Mockito.times(list.size())).accept(Mockito.any(Running
        Mockito.verify(result, Mockito.times(getNormalRunningApp().size())).add(Mock
    }
    /**
     * 关键服务应用包名
     */
    public String[] getEssentialProcessArray() {
        return new String[]{"android.process.acore", "android.process.media", "android.tts
    }
}
```

上面的代码，我们使用`@Mock`来Mock了Context和Resource对象，这需要我们在`setup`的时候使用`MockitoAnnotations.initMocks(this)`方法来使得这些注解生效，如果再不使用`@Mock`注解的时候，我们还可以使用`Mockito.mock`方法来Mock对象。这里我们

指定了Context对象在调用getResources方法的时候返回一个同样是Mock的Resources对象，这里的Resources对象，指定了在调用getStringArray(R.array.default_core_list)方法的时候返回的字符串数组的数据是通过我们的getEssentialProcessArray方法获得的，而不是真的是加载String.xml资源。最后调用EssentialProcessFilter.loadCoreList(mContext)方法使得EssentialProcessFilter内记录的关键服务集合的数据源就是我们指定的。目前，我们使用的就是改变Mock对象的行为的功能。

在测试单元testEssentialFilter方法中，使用Mockito.spy(ListFilter.class)来Mock一个ListFilter对象（这是一个BaseItemFilter的实现，里面记录了BaseItemFilter的集合，用了记录一系列的过滤规则），这里使用spy方法Mock出来的对象除非指定它的行为，否者调用这个对象的默认实现，而使用mock方法Mock出来的对象，如果不指定对象的行为的话，所有非void方法都将返回默认值：int、long类型方法将返回0，boolean方法将返回false，对象方法将返回null等等，我们也同样可以使用@spy注解来Mock对象。这里的listFilter对象使用spy是为了使用默认的行为，确保accept方法的逻辑正确执行，而result对象使用mock方式来Mock，是因为我们不需要真的把过滤后的数据添加到集合中，而只需要验证这个Mock对象的add方法调用的多少次即可。

最后就是对Mock对象的行为的验证，分别验证了listFilter#accept方法和result#add方法的执行次数，其中Mockito#any系列方法用来指定无论传入任何参数值。

依赖注入，写出更容易测试的代码

这里举一个新的例子，在使用MVP模式开发一个登录页

这是我们的LoginPresenter，Presenter的职责是作为View和Model之间的桥梁，UserManager就是这里的Model层，它用来处理用户登录的业务逻辑

```
public class LoginPresenter {
  private final LoginView mLoginView;
  private UserManager mUserManager = new UserManager();

  public LoginPresenter(LoginView loginView) {
      mLoginView = loginView;
  }

  public void login(String username, String password) {
    // ....
    mLoginView.showLoginHint();
    mUserManager.performLogin(username, password);
    // ...
  }
```

这段代码存在了一些问题

- 如果现在要改变`UserManager`生成方式，如需要用`new UserManager(String config)`初始化，需要修改`LoginPresenter`代码
- 如果想测试不同`UserManager`对象对`LoginPresenter`的影响很困难，因为`UserManager`的初始化被写死在了`LoginPresenter`的构造函数中（现在的Mockito可以使用`@InjectMocks`来很大程度缓解这个问题）

为了把依赖解耦，我们一般可以作如下改变

```
public class LoginPresenter {
  private final LoginView mLoginView;
  private final UserManager mUserManager;

  //将UserManager作为构造方法参数传进来
  public LoginPresenter(LoginView loginView, UserManager userManager) {
    this.mLoginView = loginView;
    this.mUserManager = userManager;
  }

  public void login(String username, String password) {
    // ... some other code
    mUserManager.performLogin(username, password);
  }
}
```

这就是一种常见的依赖注入方式，这种方式的好处是，依赖关系非常明显。你必须在创建这个类的时候，就提供必要的dependency。这从某种程度上来说，也是在说明这个类所完成的功能。实现依赖注入很简单，比较有名的Dagger、Dragger2这些框架可以让这种实现变得更加简单，简洁，优雅，有兴趣可以自行了解。作出这种修改之后，我们就可以很容易的Mock出`UserManager`对象来对`LoginPresenter`做单元测试

对异步任务进行测试

很多时候，我们耗时的业务逻辑都是再异步线程中处理的，这样就会对我们的测试造成了一些困难，因为很可能我们的异步任务执行完之前，我们的Test单元已经执行完了，为了能够顺利验证，其中一个思路是，首先需要一个回调接口来告诉我们处理完成，之后再我们知道回调后，继续执行测试代码，这里的一个简单实现就是使用`CountDownLatch`，也可以使用`wait`和`notifyAll`，下面以`CountDownLatch`为例子

这里是我们的回调接口

```
public interface ScanListener<T> {
  /**
   * @param t 扫描结果
   */
```

```
    void onScannedCompleted(T t);
}
```

测试代码如下：

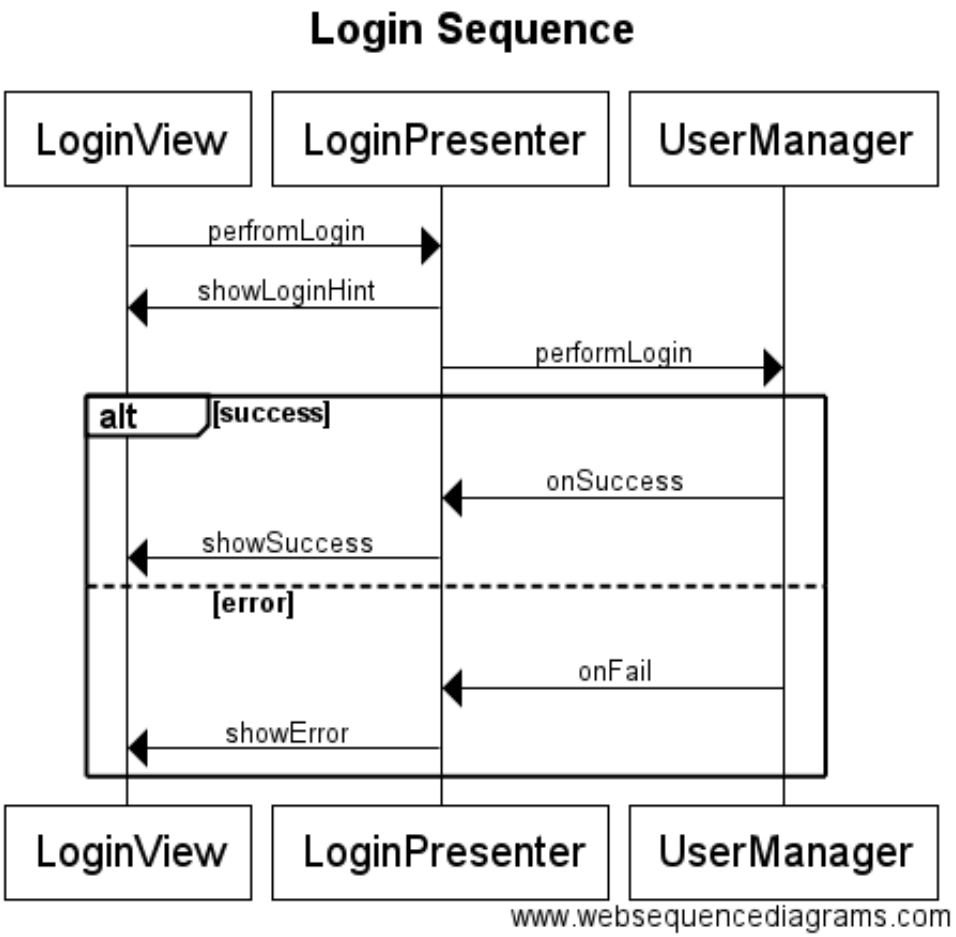
```
public class TestRunningAppScanner {
    CountDownLatch mSignal = null;
    RunningAppScanner mRunningAppScanner;
    @Mock
    BaseScanner.ScanListener<List<RunningAppBean>> mRunningAppBeanScanListener;
    @Captor
    ArgumentCaptor<List<RunningAppBean>> mListArgumentCaptor;

    @Before
    public void setUp() {
        mSignal = new CountDownLatch(1);
        MockitoAnnotations.initMocks(this);
        mRunningAppScanner = Mockito.spy(new RunningAppScanner(InstrumentationRegist
    }

    @After
    public void tearDown() {
        mSignal.countDown();
    }

    @Test
    public void testRunningApp() throws InterruptedException {
        Assert.assertFalse(mRunningAppScanner.isRunning());
        Mockito.doAnswer(new Answer() {
            @Override
            public Object answer(InvocationOnMock invocation) throws Throwable {
                mSignal.countDown();
                return invocation.getArguments();
            }
        }).when(mRunningAppBeanScanListener).onScannedCompleted((List<RunningAppBe
        mRunningAppScanner.setScanListener(mRunningAppBeanScanListener);
        mRunningAppScanner.startScanning();
        Assert.assertTrue(mRunningAppScanner.isRunning());
        mSignal.await();
        Assert.assertFalse(mRunningAppScanner.isRunning());
        Mockito.verify(mRunningAppBeanScanListener, Mockito.times(1)).onScannedComp
        //必定最少有1个进程再运行
        Assert.assertTrue(mListArgumentCaptor.getValue().size() > 0);
    }
}
```

上面代码中Mock了一个`ScanListener`类型的回调接口，并在`testRunningApp`方法中，指定了它`onScannedCompleted`方法的行为，`RunningAppScanner#startScanning`方法是用来启

动异步逻辑的，在开始异步逻辑之前设置了回调接口，在开始异步逻辑后，就使用了`mSignal.await()`方法来阻塞当前线程，直到`onScannedCompleted`方法的执行，会调用到在setup的时候新建的`CountDownLatch`对象的`countDown()`，表明异步任务的完成，以继续执行下面的测试代码

上面的测试代码实际上是属于`Instrumented Tests`，它的异步逻辑是去扫描当前设备运行的进程，需要运行再Android设备上，它的异步逻辑是实际运行的而且会扫描出真实结果。但有些时候，我们只是想**验证状态和交互结果**，而不需要它真的执行异步逻辑，我们可以使用其他的方式来进行测试，我们以一个登录流程为例子

这是登录流程的时序图



回调的定义

```
public interface LoginCallback {

    void onSuccess();

    void onFail(int code);
}
```

这是`LoginPresenter`

```
public class LoginPresenter {
    private final LoginView mLoginView;
    private final UserManager mUserManager;

    //将UserManager作为构造方法参数传进来
    public LoginPresenter(LoginView loginView, UserManager userManager) {
        this.mLoginView = loginView;
        this.mUserManager = userManager;
    }

    public void login(String username, String password) {
        mLoginView.showLoginHint();
        mUserManager.performLogin(username, password, new UserManager.LoginCallback
            @Override
            public void onSuccess() {
                mLoginView.showSuccess();
            }

            @Override
            public void onFail(int code) {
                mLoginView.showError();
            }
        });
    }
}
```

我们现在需要对`LoginPresenter`进行测试，我们的测试点在于接收到不同的回调结果的时候，对View进行不同的展示，至于`UserManager`的测试则应该是另外的一个测试单元的`LoginPresenter`的测试代码

```
public class TestDemo {
    @Mock
    LoginView mLoginView;
    @Mock
    UserManager mUserManager;

    LoginPresenter mLoginPresenter;

    @Captor
    ArgumentCaptor<LoginCallback> mCallbackArgumentCaptor;

    @Before
    public void before() {
        MockitoAnnotations.initMocks(this);
        mLoginPresenter = new LoginPresenter(mLoginView, mUserManager);
    }

    @Test
```

```
    public void showError() {
        mLoginPresenter.login("username", "password");
        Mockito.verify(mUserManager, Mockito.times(1)).performLogin(Mockito.anyStrin
        mCallbackArgumentCaptor.getValue().onFail(0);
        Mockito.verify(mLoginView, Mockito.times(1)).showError();
    }

    @Test
    public void showSuccess() {
        mLoginPresenter.login("username", "password");
        Mockito.verify(mUserManager, Mockito.times(1)).performLogin(Mockito.anyStrin
        mCallbackArgumentCaptor.getValue().onSuccess();
        Mockito.verify(mLoginView, Mockito.times(1)).showSuccess();
    }
}
```

这里的关键是`@Captor ArgumentCaptor<LoginCallback> mCallbackArgumentCaptor;`，ArgumentCaptor是作用是用来捕获所传入方法特定的参数，然后可以进一步对参数进行断言，而且是在方法调用之后使用。这里我们捕获到`UserManager#performLogin`的第三个`LoginCallback`的参数，然后直接使用它来 做相应回调来验证不同情况下的反馈结果是否正确

Mockito的一些替代或扩增库

使用Mockito并不可以Mock对象的静态方法、private修饰的方法、static方法、构造函数等，使用JMockit或PowerMock是可以解决这样的问题，有时间的话可以去实践下

单元测试的优点

- 不依赖Android的API，运行速度快，所以更快地得到结果反馈
- 引导更好的代码设计（单一职责、依赖注入），如果一个类不好测，往往是因为这个类的设计是有问题

Instrumented Tests

Instrumented Unit tests是需要运行再Android设备上的（物理/虚拟），通常我们使用Mock的方式不能很好解决对Android的API的依赖的这个问题，而使用这种测试方式可以依赖Android的API，使用Android提供的Instrumentation系统，将单元测试代码运行在模拟器或者是真机上，但很多情况来说，我们还是会需要和Mockito一起使用的。**这中方案速度相对慢，因为每次运行一次单元测试，都需要将整个项目打包成apk，上传到模拟器或真机上，就跟运行了一次app似得**，所以。

配置

- 测试代码目录：(src/androidTest/java)
- 一般使用到的测试框架
    - Testing Support Library
        - AndroidJUnitRunner ： JUnit 4-compatible test runner for Android
        - Espresso ：UI testing framework; suitable for functional UI testing within an app
        - UI Automator ：UI testing framework; suitable for cross-app functional UI testing across system and installed apps

使用

AndroidJUnitRunner

AndroidJUnitRunner是JUnit4运行在Android平台上的兼容版本，使用也是很简单的且方法类似，以上面测试异步任务为例子

```
@RunWith ( AndroidJUnit4 . class )
@MediumTest
public class TestRunningAppScanner {
    CountDownLatch mSignal = null ;
    RunningAppScanner mRunningAppScanner ;
    @Mock
    BaseScanner . ScanListener< List< RunningAppBean > > mRunningAppBeanScanListener ;


    @Before
    public void setUp ( ) {
        mSignal = new CountDownLatch ( 1 ) ;
        MockitoAnnotations . initMocks ( this ) ;
        mRunningAppScanner = Mockito . spy ( new RunningAppScanner ( InstrumentationRegist
    }

    @After
    public void tearDown ( ) {
        mSignal . countDown ( ) ;
    }

    @Test
    public void testRunningApp ( ) throws InterruptedException {
        Assert . assertFalse ( mRunningAppScanner . isRunning ( ) ) ;
        Mockito . doAnswer ( new Answer ( ) {
```

```
            @Override
            public Object answer(InvocationOnMock invocation) throws Throwable {
                mSignal.countDown();
                return invocation.getArguments();
            }
        }).when(mRunningAppBeanScanListener).onScannedCompleted((List<RunningAppBe
        mRunningAppScanner.setScanListener(mRunningAppBeanScanListener);
        mRunningAppScanner.startScanning();
        Assert.assertTrue(mRunningAppScanner.isRunning());
        mSignal.await();
        Assert.assertFalse(mRunningAppScanner.isRunning());
        Mockito.verify(mRunningAppBeanScanListener, Mockito.times(1)).onScannedComp
    }
}
```

需要在测试类的定义上加上`@RunWith(AndroidJUnit4.class)`标注，另外`@MediumTest`标注是用来指定测试规模，有三种类型可以指定，限制如下，因为这里的异步任务的目的是扫描所有正在运行的App，所以这里使用`@MediumTest`，其他的`Assert#XXX`、`@Befor`，`@After`使用一致，另外这里还搭配Mockito使用。因为需要真正用到系统的一些服务（AMS，PKMS）或资源，这里的`InstrumentationRegistry`可以为我们提供`Instrumentation`对象，测试App的`Context`对象，目标App的`Context`对象和通过命令行启动测试所提供的参数信息。这里的`InstrumentationRegistry.getTargetContext()`就是用来获取目标App的`Context`对象。

Feature	SmallTest	MediumTest	LargeTest
Network	No	localhost only	Yes
Database	No	Yes	Yes
File system access	No	Yes	Yes
Use external systems	No	Discouraged	Yes
Multiple threads	No	Yes	Yes
Sleep statements	No	Yes	Yes
System properties	No	Yes	Yes
Execution Time (ms)	200<	1000<	>1000

Espresso

Espresso提供大量的方法用来进行UI测试，这些API可以让你写的简洁和可靠的自动化UI测试，站在用户的角度测试，所有逻辑是黑盒

Espresso的三个重要功能：

- 灵活的可扩展的视图（特别是AdapterView）匹配APIs.详情View matching .
- 大量的UI交互操作APIs.详情Action APIs .
- UI线程同步，提高测试可靠性,详情UI thread synchronization .

使用流程

- 元素定位

```
public static ViewInteraction onView ( final Matcher< View > viewMatcher ) {}
```

Espresso.onView方法接收一个Matcher<View>类型的入参，并返回一个ViewInteraction对象。ViewMatchers对象提供了大量的withXXX方法用来定位元素，常用的有withId，withText和一系列的isXXX方法用来判断元素的状态。如果单一的匹配条件无法精确地匹配出来唯一的控件，我们可能还需要额外的匹配条件，此时可以用AllOf#allOf()方法来进行复合匹配条件的构造(下面的AdapterView节有使用到)：

```
onView ( allOf ( withId ( id ), withText ( text ) ) )
```

- 操作元素

当定位到元素后，返回一个ViewInteraction对象，其perform方法可以接收一系列ViewAction用来进行模拟用户操作，ViewActions类提供了大量的操作实现，常用的有typeText，click等

```
public ViewInteraction perform ( final ViewAction . . . viewActions ) {}
```

- 验证结果

最后为了验证操作是否符合预期，我们还是需要定位到元素，获得一个ViewInteraction对象，其check方法接收了一个ViewAssertion的入参，该入参的作用就是检查结果是否符合我们的预期。

```
public ViewInteraction check ( final ViewAssertion viewAssert ) {}
```

ViewAssertion提供如下方法,这个方法接收了一个匹配规则，然后根据这个规则为我们生成了一个ViewAssertion对象,这个Matcher<View>的如参和定位元素的时候是一个用法的

```
public static ViewAssertion matches(final Matcher<? super View> viewMatcher) {}
```

下面是测试主页部分UI测试的代码

```
@RunWith(AndroidJUnit4.class)
@LargeTest
public class MainActivityUITest {
    /**
     * {@link ActivityTestRule} is a JUnit {@link Rule @Rule} to launch your activity under test.
     * <p/>
     * Rules are interceptors which are executed for each test method and are important building
     * blocks of Junit tests.
     */
    @Rule
    public ActivityTestRule<MainActivity> mActivityRule = new ActivityTestRule<MainActivity>

    //默认在测试之前启动该Activity
//    public ActivityTestRule<MainActivity> mActivityRule = new ActivityTestRule<MainActivity>(M


    @Test
    public void testNavigateToAdvanceFragment() {
        Intent intent = new Intent();
        //携带信息的方式启动Activity
        intent.putExtra("EXTRA", "Test");
        mActivityRule.launchActivity(intent);
        // Open Drawer to click on navigation.
        Espresso.onView(ViewMatchers.withId(R.id.drawer_layout))
                .check(ViewAssertions.matches(DrawerMatchers.isClosed(Gravity.LEFT)))
                .perform(DrawerActions.open()); // Open Drawer
        // navigate to advance fragment.
        Espresso.onView(ViewMatchers.withText(R.string.advanced_settings_group_title_adv
                .perform(ViewActions.click());
        // Left Drawer should be closed.
        Espresso.onView(ViewMatchers.withId(R.id.drawer_layout))
                .check(ViewAssertions.matches(DrawerMatchers.isClosed(Gravity.LEFT)));
        Espresso.onView(ViewMatchers.withText(R.string.fingerprint_settings_title)).check
    }

}
```

`@Rule` + `ActivityTestRule`用来标记需要测试的Activity，测试框架会再运行@Test标注的测试用例之前会启动这个指定的Activity（类似还有`IntentsTestRule`，`ServiceTestRule`分别用于测试Intent和Service），有些情况我们测试的`Actvity`需要再`Intent`那里携带一些信息，这里也是可以通过`ActivityTestRule`的不同构造函数+ `ActivityTestRule#launchActivity`方法来完成

它的作用是找到R.id.drawer_layout的View，这是一个DrawerLayout，DrawerMatchers提供了用于判断DrawerLayout的状态，DrawerActions类提供了用于操作DrawerLayout的操作。首先是判断DrawerLayout是隐藏的，然后打开抽屉，并找到R.string.advanced_settings_group_title_advanced_features标记的Item，进行一次点击操作，最后验证当前显示的是高级界面

处理AdapterView

不同于之前提到的静态的控件，AdapterView在加载数据时，可能只有一部分显示在了屏幕上，对于没有显示在屏幕上的那部分数据，我们通过Espresso.onView()是没有办法找到的。所以需要使用Espresso.onData() ,为了精确定位，通常还需要我们自定义Matcher

基本流程的区别在于元素定位上：

- 元素定位

```
public static DataInteraction onData ( Matcher< ? extends Object > dataMatcher ) { }
```

这个方法接收一个Matcher<? extends Object>类型的入参，并返回一个DataInteraction对象。DataInteraction关注于AdapterView的数据。由于AdapterView的数据源可能很长，很多时候无法一次性将所有数据源显示在屏幕上，因此我们主要先关注AdapterView中包含的数据，而非一次性就进行视图元素的匹配，入参Matcher<? extends Object>构造一个针对于Object(数据)匹配的匹配规则。从以上对比可以看出，我们在使用onData()方法对AdapterView进行测试的时候，我们的思路就转变成了首先关注这个AdapterView的具体数据，而不是UI上呈现的内容。当然，我们最终的目标还是要找到目标的UI元素，但是我们是通过其数据源来进行入手的

- 自定义Matcher

  对于针对于Object(数据)的匹配器一般都需要我们自定义的，如下是针对AdapterView的数据是LockerItem类型的匹配器，用于LockerItem对象的title字段是否为输入的字符串，复写的matchesSafely()方法是真正执行匹配的地方了：

```
/* *
* 查找指定关键字的搜索条件
* @param name 需要搜索的关键字
*/
public static Matcher< Object > lockItemMatcher ( final String name ) {
  return new BoundedMatcher< Object , LockerItem > ( LockerItem . class ) {
    @Override
    protected boolean matchesSafely ( LockerItem item ) {
      return item != null && item . getTitle ( ) . equals ( name ) ;
```

```
        }

        @Override
        public void describeTo(Description description) {
          description.appendText("SearchItem has Name: " + name);
        }
      };
    }
```

- 指定AdapterView

  通常还需要我们指定AdapterView，避免存在多个AdapterView的时候所带来的麻烦,通过`inAdapterView`方法来匹配，匹配规则和之前的视图匹配一样，可以同id来进行匹配

```
public DataInteraction inAdapterView(Matcher<View> adapterMatcher){}
```

- Child View

  有时候我们需要对一个View中的某个子控件进行操作（比如点击一个ListView条目中的某个指定Button），这时我们可以通过`onChildView``方法指定相应的子控件

```
DataInteraction onChildView(Matcher<View> childMatcher){}
```

- 操作元素

同处理普通元素差不多

- 验证结果

同处理普通元素差不多

下面是一个找到id为`R.id.main_list`和数据类型为`LockerItem`的AdapterView中标题为`WeChat`的元素，并对该元素中id为`R.id.item_ripple`的子元素模拟一次点击事件

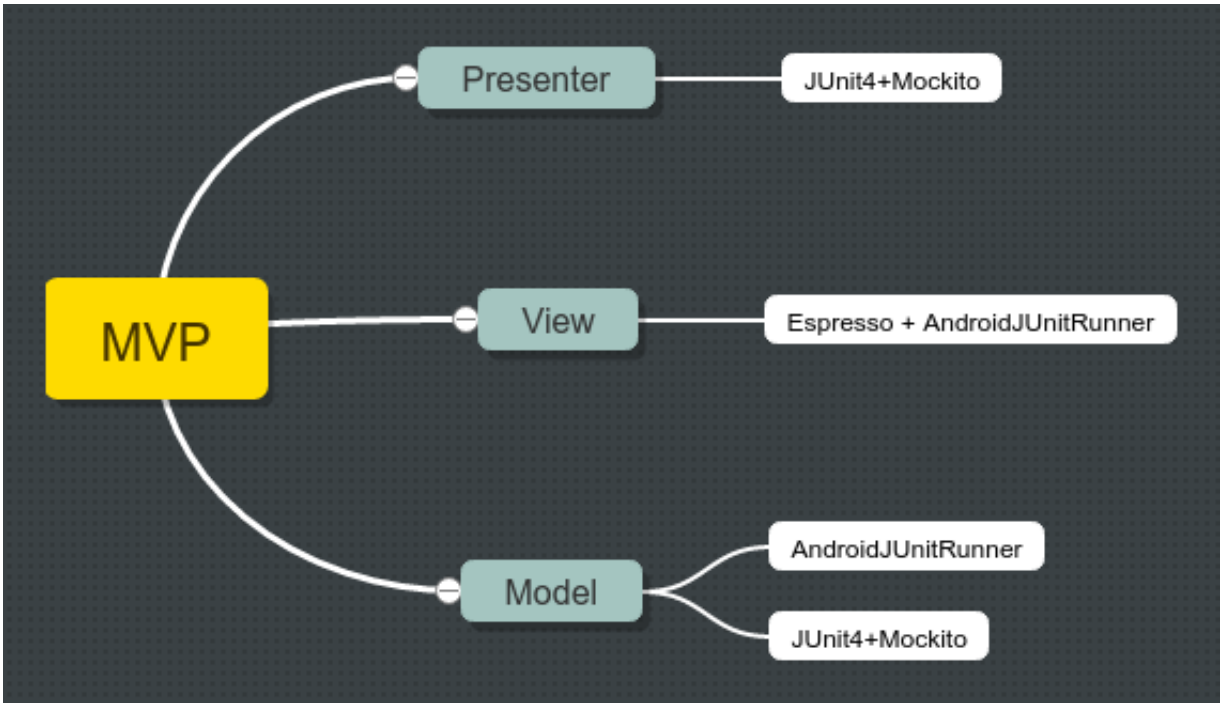
```
@RunWith(AndroidJUnit4.class)
@LargeTest
public class MainActivityUITest {
    /**
     * {@link ActivityTestRule} is a JUnit {@link Rule @Rule} to launch your activity under test.
     * <p/>
     * Rules are interceptors which are executed for each test method and are important building
     * blocks of Junit tests.
```

```
     */
    @Rule
    public ActivityTestRule< MainActivity > mActivityRule = new ActivityTestRule< MainActivity >

    @Test
    public void testLockFirstItem ( ) throws InterruptedException {
      Intent intent = new Intent ( ) ;
      intent . putExtra ( " EXTRA " , " Test " ) ;
      mActivityRule . launchActivity ( intent ) ;
      onData ( allOf ( is ( instanceOf ( LockerItem . class ) ) , lockItemMatcher ( " WeChat " ) ) ) . in
      // ...
    }

    /* *
     * 查找指定关键字的搜索条件
     * @param name 需要搜索的关键字
     */
    public static Matcher< Object > lockItemMatcher ( final String name ) {
      return new BoundedMatcher< Object , LockerItem > ( LockerItem . class ) {
        @Override
        protected boolean matchesSafely ( LockerItem item ) {
          return item != null && item . getTitle ( ) . equals ( name ) ;
        }

        @Override
        public void describeTo ( Description description ) {
          description . appendText ( " SearchItem has Name: " + name ) ;
        }
      } ;
    }

}
```

MVP各层的单元测试框架的使用

Google官方的Android Architecture Blueprints这个项目除了是Google利用不同的流行的开源库组合来搭建MVP框架的实践，同时还有完整的测试用例，所以学习价值很高，所以如果没了解过，现在就去看看吧，必定受益匪浅

这是这个项目对于MVP模式的各层做测试的各种测试框架使用



- View层:涉及到UI且需要再设备上运行，所以需要Espresso和AndroidJUnitRunner
- Preseneter层：Preseneter层应该设计成纯JAVA层的，所以使用JUnit+Mockito
- Model层：需要依赖Android环境

参考文档

- Best Practices for Testing
- Android 单元测试: 首先，从是什么开始
- Android单元测试在蘑菇街支付金融部门的实践
- 测试技巧– 你所不知道的测试黑科技
- Testing Asynchronous Methods
- Unit testing asynchronous methods with Mockito
- Android user interface testing with Espresso - Tutorial
- Robolectric3.0的介绍和实战
- 公共技术点之依赖注入
- Android Architecture Blueprints
- 解读Android官方MVP项目单元测试