Android单元测试在蘑菇街支付金融部门的实践

chriszou.com/2016/04/25/android-unit-testing-wechat-group-share

April 25, 2016

大家好，我是蘑菇街支付金融部门的邹勇，花名叫小创。今天很高兴跟大家分享一下安卓的单元测试在蘑菇街支付金融的实践。下面，我们从为什么开始。

为什么要写单元测试

首先要介绍为什么蘑菇街支付金融这边会采用单元测试的实践。说起来比较巧，刚开始的时候，只是我一个人会写单元测试。后来老板们知道了，觉得这是件很有价值的事情，于是就叫我负责我们组的单元测试这件事情。就这样慢慢的，单元测试这件事情就成了我们这边的正常实践了。再后来，在公司层面也开始有一定的推广。

要说为什么要写单元测试的话，我相信大部分人都能承认、也能理解单元测试在保证代码质量，防止bug或尽早发现bug这方面的作用，这可能是大家觉得单元测试最大的作用。然而我觉得，除了这方面的作用，单元测试还能在很大程度上改善代码的设计，同时还能节约时间，让人工作起来更自信、更开心，以及其他的一些好处。这些都是我的切身感受，我相信也是多数真正实践过单元测试的人的切身感受，而不是为了宣传这个东西而说的好听的大话。

说到节约时间，大家可能就会好奇了，写单元测试需要时间，维护单元测试代码也需要时间，应该更费时间才对啊？

这就是在开始分享之前，我想重点澄清的一点，那就是，单元测试本身其实不会占用多少时间，相反，还会节约时间。只是：1. 学习如何做单元测试需要时间；2. 在一个没有单元测试的项目中加入单元测试，需要一定的结构调整的时间，因为一个有单元测试跟没有单元测试的项目，结构上还是有较大不同的。

打个比方，开车这件事情，需要很多时间吗？我相信很少人会说开车这件事情需要很多时间，而是：1. 学习开车，需要一定的时间；2. 如果路面不平的话，那么修路需要一定的时间。单元测试也是类似的情况。

那为什么说单元测试可以节约时间呢？简单说几点：1. 如果没有单元测试的话，就只能把app运行起来测试，这比运行一次单元测试要慢多了。2. 尽早发现bug，减少了debug和fixbug的时间。3. 重构的时候，大大减少手动验证重构正确性的时间。

所以，我希望大家能去掉"没时间写单元测试"这个印象，如果工作上安排太紧，没有时间学习如何做单元测试的话，可以自己私底下学，然后在慢慢应用到项目中。

单元测试简单介绍，以及void方法怎么测

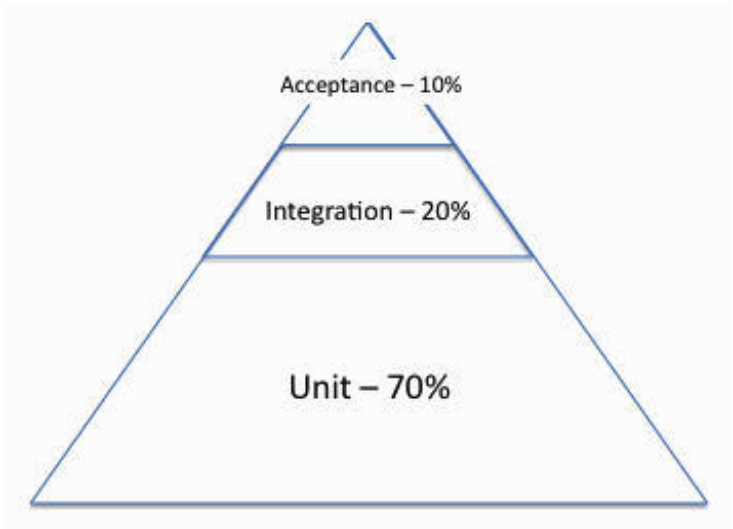
接下来介绍我们这边是怎么做安卓单元测试的。首先澄清一下概念，在安卓上面写测试，有很多技术方案。有JUnit、Instrumentation test、Espresso、UiAutomator等等，还有第三方的Appium、Robotium、Calabash等等。我们现在讲的是使用JUnit和其他的一些框架，写可以在我们开发环境的JVM上面直接运行的单元测试，其他的几种其实都不属于单元测试，而是集成测试或者叫Functional test等等。这两者明显的不同是，前者可以直接在开发

用的电脑，或者是CI上面的JVM上运行，而且可以只运行那么一小部分代码，速度非常快。而后者必须要有模拟器或真机，把整个project打包成一个app，然后上传到模拟器或真机上，再运行相关的代码，速度相对来说慢很多。
单元测试的定义相信大家都知道，就是为我们写的某一个代码单元（比如一个方法）写的测试代码。一个单元测试大概可以分为三个部分：

1. setup：即new 出待测试的类，设置一些前提条件
2. 执行动作：即调用被测类的被测方法，并获取返回结果
3. 验证结果：验证获取的结果跟预期的结果是一样的

然而一个类的方法分两种，一种是有返回值的方法。一种是没有返回值的方法，即void方法。对于有返回值的方法，固然测试起来是很容易的，但是对于没有返回值的方法，该怎么测试呢？这里的关键是，怎么样获取这个方法的“返回结果”?
这里举一个例子来说明一下，顺便澄清一个十分常见的误解。比如说有一个Activity，管他叫 `DataActivity` ，它有一个 `public void loadData()` 方法, 会去调用底层的 `DataModel` 类，异步的执行一些网络请求。当网络请求返回以后，更新用户界面。
这里的 `loadData()` 方法是void的，它该怎么测试呢？一个最直接的反应可能是，调用 `loadData()` 方法(当然，实际可能是通过其他事件触发)，然后一段时间后，验证界面得到了更新。然而这种方法是错的，这种测试叫集成测试，而不是单元测试。因为它涉及到很多个方面，它涉及到 `DataModel` 、网络服务器，以及网络返回正确时， `DataActivity` 内部的处理，等等。集成测试固然有它的必要性，但不是我们应该最关注的地方，也不是最有价值的地方。我们应该最关注的是单元测试。关于这一点，有一个Test Pyramid的理论：

Test Pyramid理论基本大意是，单元测试是基础，是我们应该花绝大多数时间去写的部分，而集成测试等应该是冰山上面能看见的那一小部分。
那么对于这个case，正确的单元测试方法，应该是去验证 `loadData()` 方法调用了 `DataModel` 的某个请求数据的方法，同时传递的参数是正确的。“调用了DataModel的方法，同时参数是。。。”这个才是 `loadData()` 这个方法的“返回结果”。



Mock的概念以及Mockito框架

要验证某个对象的某个方法得到调用了，就涉及到mock的使用。 这里对mock的概念做个简单介绍，以免很多同学不熟悉，mock就是创建一个虚假的、模拟的对象。在测试环境下，用来替换掉真实的对象。这样就能达到两个目的：1. 可以随时指定mock对象的某个方法返回什么样的值，或执行什么样的动作。 2. 可以验证mock对象的某个方法有没有得到调用，或者是调用了多少次，参数是什么等等。

要使用mock，一般需要使用mock框架，目前安卓最常用的有两个，Mockito和JMockit。两者的区别是，前者不能mock static method和final class、final method，后者可以。我们依然采用的是Mockito，原因说起来惭愧，是因为刚开始并不知道JMockit这个东西，后来查了一些资料，看过很多对比Mockito和JMockit的文章，貌似大部分还是很看好JMockit的，只是有一个问题，那就是跟robolectric的结合也有一些bug，同时使用姿势跟Mockito有较大的不同，因此一直没有抽时间去实践过。这个希望以后能够做进一步的调查，到时候在给大家分享一下使用感受。
但是使用Mockito，就有一个问题，那就是static method和final class、final method没有办法mock，对于这点如何解决，我们稍后会介绍到。

在测试环境中使用mock：依赖注入

接下来的一个问题就是，如何在测试环境下，把 `DataModel` 换成mock的对象，而正式代码中， `DataModel` 又是正常的对象呢？
这个问题也有两种解决方案，一是使用专门的testing product flavor；二是使用依赖注入。
第一种方案就是用一个专门的product flavor来做testing，在这个testing flavor里面，里面把需要mock的类写一份mock的implementation，然后通过factory提供给client，这个factory的接口在testing flavor和正式的flavor里面是一样的，在跑testing的时候，专门使用这个testing flavor，这样通过factory得到的就是mock的类。这种情况看起来很简单，但其实很不灵活，因为只有一种mock实现；此外，代码会变得很丑陋，因为你需要为每一个dependency提供一个factory，会觉得很刻意；再者，多了一个flavor，很多gradle任务都会变得很慢。关于这种方案，可以参考这个视频。
因此，我们用的是第二种，依赖注入。先简单介绍一下依赖注入这个模式，他的基本理念是，某一个类（比如说 `DataActivity` ），用到的内部对象（比如说 `DataModel` ）的创建过程不在 `DataActivity` 内部去new，而是由外部去创建好 `DataModel` 的实例，然后通过某种方式set给 `DataActivity` 。这种模式应用是非常广泛的，尤其是在测试的时候。为了更方便的做依赖注入，如今有很多框架专门做这件事情，比如RoboGuice、Dagger、Dagger2等等。我们用的是Dagger2，理由很简单，这是目前最好用的DI框架。
关于Dagger2的文章，之前我们群里也分享了不少，但是好像我并没有看到讲述没有关于如何在测试环境下使用Dagger2的文章，这个还是略感遗憾的。离开单元测试，使用依赖注入就少了很有说服力的一个理由。
那么这里我就介绍一下，怎么样把Dagger2应用到单元测试中。熟悉dagger2的童靴可能知道，Dagger2里面最关键的有两个概念，**Module** 和**Component**。**Module**是负责生成诸如 `DataModel` 这样被别人（比如 `DataActivity` ）使用的类的地方。用术语的话，被别人使用的类 `DataModel` 叫*Dependency*，使用到了别的类的类 `DataActivity` 叫*Client*。
而**Component**则是供Client使用Dependency的统一接口。也就是说， `DataActivity` 通过Component，来得到一份 `DataModel` 的实例。
现在，关键的地方来了，Component本身是不生产dependency的，它只是搬运工而已，真正生产dependency的地方在Module。所以，创建Component需要用到Module，不同的Module生产出不同的dependency。在正式代码里面，我们使用正常的Module，生产正常的 `DataModel` 。而在测试环境中，我们写一个 `TestingModule` ，让它继承正常的Module，然后override掉生产 `DataModel` 的方法，让它生产mock的 `DataModel` 。在跑单元测试的时

候，使用这个 `TestingModule` 来创建Component，这样的话， `DataActivity` 通过Component得到的 `DataModel` 对象就是mock出来的 `DataModel` 对象。
使用这种方式，所有production code都不用专门为testing增加任何多余的代码，同时还能得到依赖注入的其他好处。

Robolectric：解决Android单元测试最大的痛点

接下来讲讲Android单元测试最大的痛点，那就是JVM上面运行纯JUnit单元测试时是不能使用Android相关的类的，因为我们开发用到的安卓环境是没有实现的，里面只定义了一些接口，所有方法的实现都是 `throw new RuntimeException("stub");` ，如果我们单元测试代码里面用到了安卓相关的代码的话，那么运行时就会遇到 `RuntimeException("Stub")` 。
要解决这个问题，一般来说有三种方案：

1. 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。
2. 用一定的架构，比如MVP等等，将安卓相关的代码隔离开了，中间的Presenter或Model是存java实现的，可以在JVM上面测试。View或其他android相关的代码则不测。
3. 使用Robolectric框架，这个框架基本可以理解为在JVM上面实现了一套安卓的模拟环境，同时给安卓相关的类增加了其他一些增强的功能，以方便做单元测试，使用这个框架，我们就可以在JVM上面跑单元测试的时候，就可以使用安卓相关的类了。

第一种方案能work，但是速度非常慢，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，这个显然不是单元测试该有的速度，更无法做TDD。这种方案首先被否决。
刚开始，我们采用的是Robolectric，原因有两个：1. 我们项目当时还没有比较清楚的架构，android跟纯java代码的隔离没有做好；2. 很多安卓相关的代码，还是需要测试的，比如说自定义View等等。然而慢慢的，我们的态度从拥抱Robolectric，到尽量不用它，尽量使用纯java代码去实现。可能大家觉得安卓相关的代码会很多，而纯java的很少，然而慢慢的你会发现，其实不是这样的，纯java的代码其实真不少，而且往往是核心的逻辑所在。之所以尽量不用Robolectric，是因为Robolectric虽然相对于Instrumentation testing来说快多了。但毕竟他也需要merge一些资源，build出来一个模拟的app，因此相对于纯java和JUnit来说，这个速度依然是很慢的。
用具体的数字来对比说明：

- 运行Instrumentation testing：几十秒，取决于app的大小
- Robolectric：10秒左右
- JUnit：几秒钟之内

当然，虽然运行一次Robolectric在10秒左右，但是对比运行一次app，还是要快太多。因此，刚开始的时候，从Robolectric开始完全是OK的。
以上就是现在我们这边单元测试用到的几个基本技术：JUnit4 + Mockito + Dagger2 + Robolectric。基本来说，并没有什么黑科技，都是业界标准。

一个具体的案例

接下来，我通过一个具体的案例，跟大家介绍一下，我们这边的一个app，具体是怎么单测的。
这里是我们收银台界面的样子：

假设Activity名字为 `CheckoutActivity` ，当它启动的时候， `CheckoutActivity` 会去调一个 `CheckoutModel` 的 `loadCheckoutData()` 方法，这个方法又会去调更底层的一个封装了用户认证等信息的网络请求Api类(`mApi`)的get方法，同时传给这个Api类一个callback。这个callback的做的事情是将结果通过Otto Bus(`mBus`) post出去。 `CheckoutActivity` 里面Subscribe了这个Event（方法名是 `onCheckoutDataLoaded()` ），然后根据Event的值相应的显示数据或错误信息。
代码简写如下：

```
public class CheckoutActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // other code, like setContentView, get data from Intent, etc.
        mCheckoutModel.loadCheckoutData(paymentId);
    }
    @Subscribe
    public void onCheckoutDataLoaded(DataLoadedEvent event) {
        if (event.successful()) {
            //Get data from event and update UI
        } else {
            //show error message
        }
    }
}
public class CheckoutModel {
    public void loadCheckoutData(String paymentId) {
        //Other code, like composing params
        mApi.get(someUrl, someParams, new NetworkCallback() {
            @Override
            public void onSuccess(Object data) {
                mBus.post(new DataLoadedEvent(data));
            }
            @Override
            public void onFailure(int code, String msg) {
                mBus.post(new DataLoadedEvent(code, msg));
            }
        });
    }
}
```

这里， `CheckoutActivity` 里面的 `mCheckoutModel` 、CheckoutModel里面的 `mApi` 、CheckoutModel里面的 `mBus` ，都是通过Dagger2注入进去的。在做单元测试的时候，这些都是mock。
对于这个流程，我们做了如下的单元测试：

- `CheckoutActivity` 启动单元测试：通过Robolectric提供的方法，启动一个 `Activity` 。验证里面的 `mCheckoutModel` 的 `loadCheckoutData()` 方法得到了调用，同时参数（订单ID等）是对的。

- `CheckoutModel` 的 `loadCheckoutData` 单元测试1：调用 `CheckoutModel` 的 `loadCheckoutData()` 方法，验证里面的 `mApi` 对应的get方法得到了调用，同时参数是对的。

- `CheckoutModel` 的 `loadCheckoutData` 单元测试2：mock Api类，指定当它的get方法在收到某些调用的时候，直接调用传入的callback的onSuccess方法，然后调用 `CheckoutModel` 的 `loadCheckoutData()` 方法，验证Otto bus的post方法得到了调用，并且参数是对的。

- `CheckoutModel` 的 `loadCheckoutData` 单元测试3：mock api类，指定当它的get方法在收到某些调用的时候，直接调用传入的callback的onFailure方法，然后调用 `CheckoutModel` 的 `loadCheckoutData()` 方法，验证Otto bus的post方法得到了调用，并且参数是对的。

- `CheckoutActivity` 的 `onCheckoutDataLoaded` 单元测试1：启动一个 `CheckoutActivity` ，调用他的 `onCheckoutDataLoaded()` ，传入含有正确数据的Event，验证相应的数据view显示出来了

- `CheckoutActivity` 的 `onCheckoutDataLoaded` 单元测试2：启动一个 `CheckoutActivity` ，调用他的 `onCheckoutDataLoaded()` ，传入含有错误信息的Event，验证相应的错误提示view显示出来了。

这里需要说明的一点是，上面的每一个测试，都是独立进行的，不是说下面的单元测试依赖于上面的。或者说必须先做上面的，再做下面的。
这部分较为详细的代码放在github上，groupshare这个package里面。

其他的问题

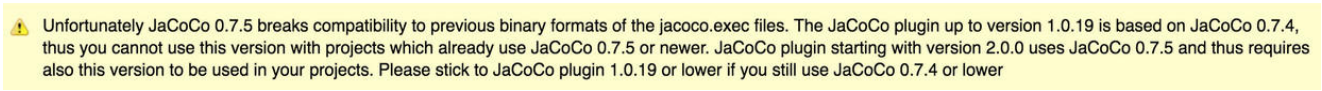
以上就是我们这边做单元测试用到的技术，以及一个基本流程，下面聊聊其他的几个问题。

哪些东西需要测试呢？

1. 所有的Model、Presenter/ViewModel、Api、Utils等类的public方法
2. Data类除了getter、setter、toString、hashCode等一般自动生成的方法之外的逻辑部分
3. 自定义View的功能：比如set data以后，text有没有显示出来等等，简单的交互，比如click事件，负责的交互一般不测，比如touch、滑动事件等等。
4. Activity的主要功能：比如view是不是存在、显示数据、错误信息、简单的点击事件等。比较复杂的用户交互比如onTouch，以及view的样式、位置等等可以不测。因为不好测。

CI和code coverage： Jacoco

要把单元测试正式化，CI是非常重要的一步，我们有一个运行Jenkins的CI server，每次开发者push代码到master branch的时候，会运行一次单元测试的gradle task，同时使用Jacoco做code coverage。
这里有个坑要特别注意，那就是项目里面的gradle Jacoco插件和Jenkins的Jacoco插件的兼容性问题。我们用的gradle Jacoco插件是7.1，更高版本的好像有问题。然后对应的Jenkins的Jacoco插件需要1.0.19或更低版本的，更高版本的jenkins plugin不支持低版本的gradle Jacoco项目版本。实际上，这点在Jenkins的Jacoco插件首页就有说明：



但是我当时没注意，所以覆盖率数据一直出不来，折腾了好一会，最后还是在同事的帮助下找到问题了。

遇到的坑，以及好的practice建议

接下来讲讲我们遇到的一些坑，以及一些好的practice建议。

1. Native libary

无论是纯JUnit还是Robolectric，都不支持load native library，会报UnsatisfiedLinkError的错。所以如果你的被测代码里面用到了native lib，那么可能需要给System.loadLibrary加上try catch。
如果是被测代码用到的第三方lib，而里面用到了native lib的话，一般有两种解决办法，一种是将用到native lib的第三方类外面自己在包一层，然后在测试的情况下mock掉。第二种是用Robolectric，给那个类创建一个shadow class。
第一种方法的好处是可以在测试的时候随时改变这个类的返回值或行为，缺点是需要另外创建一个wrapper类，会有点繁琐。第二种方式不能随时改变这个类的行为，但是写起来非常简单。所以，看自己的需要，选择相应的方法。
这两种方法，也是解决static method, final class/method不能mock的主要方式。

2. 尽量写出易于测试的代码

static method、直接new object、singleton、Global state等等这些都是一些不利于测试的代码方式，应该尽量避免，用依赖注入来代替这些方式。

3. 不要重复你的unit test

比如说你使用了一个builder模式来创建了一个类，这个builder有一个validator，来validate一些参数情况。那么这种情况，builder跟validator分开测，用各种正确的错误的参数情况去测试validator，然后测builder的时候，就不用遍历各种有效的跟无效的参数去测试了。
因为如果这样的话，到时候Validator的逻辑改了，那么针对Validator的测试跟针对Builder的测试都要修改，这个其实是重复的。这里只需要测试这个builder里面有一个Validator就好了。

4. 公共的单元测试library

如果你们公司也是组件化开发的话，抽出一个公共的单元测试类库来做单元测试，里面可以放一些公共的helper、utils、rules等等，这个可以极大的提高写单元测试的速度。

5. 把安卓里面的“纯java”代码copy一份到自己的项目里面

安卓里面有些类其实跟安卓没太大关系的，比如说TextUtils、Color等等，这些类完全可以把代码copy出来，放到自己的项目里面，然后其他地方就用这个类，这样也能部分摆脱android的依赖，使用JUnit而不是Robolectric，提高运行test的速度。

6. 充分发挥JUnit Rule的作用

JUnit Rule是个很强大的工具，然而知道的人却不多。它的基本作用是，让你在执行某个测试方法前后，可以做一些事情。如果你的好几个测试类里面有很多的共同的setup、teardown工作，你可能会倾向于使用继承，结合@Before、@After来减少duplication，这里更建议大家使用JUnit Rule来实现这个目的，而不是用继承，这样可以有更大的灵活性。
比如，为了方便测试Activity的method，我们有一个ActivityRule，在跑一个测试方法之会启动target Activity，然后跑完以后自动finish这个activity。
其中一个比较有趣的用JUnit Rule实现的功能，是实现类似于BDD测试框架的命名方式。做单元测试的时候，你经常需要为同一个方法写好几个测试方法，每个测试方法测试不同的点。为了让命名更具可读性，我们往往会把名字写的很长，在这种情况下，如果用驼峰命名的话，需要不断切换大小写，写起来麻烦，可读性也不高。如果用下划线的话，写起来也很麻烦。如果你使用过BDD的一些框架（比如RSpec、Cucumber、Jasmine等），你就会异常怀念那种“命名”方式。如果你没用过的话，那种“命名”方式大概是这样的：

```
describe Hash do
  # 一下是一个测试方法，it后面的字符串就是这个测试方法的“命名”
  it "hashes the correct information in a key" do
     expect(hash[:hello]).to eq('world')
  end
end
```

这里的关键是，当测试方法失败的时候，这个字符串是要能被加到错误信息里面的。我们做了个JUnit Rule来达到这个效果。做法是结合一个自定义的annotation，这个annotation接收一个String，来描述这个测试方法的测试目的。在Rule里面将这个annotation读出来，如果测试没通过的话，把这个描述性的String加到输出的error message里面。这样在批量运行的时候，一看就知道没通过的测试是测什么东西的。而测试方法的命名则可以比较随意。
达到的效果如下：

```java
@Test
@JSpec(desc = "Should current user not null")
public void testcurrusernotnull() {
    Assert.assertNotNull(userModel.currentUser());
}
```

如果运行失败，得到如下的结果

```
java.lang.Throwable: UserModelTest#testcurrusernotnull: Should current user not null...Failed!

    at junit.framework.Assert.fail(Assert.java:55)
    at junit.framework.Assert.assertTrue(Assert.java:22)
    at junit.framework.Assert.assertFalse(Assert.java:39)
    at junit.framework.Assert.assertFalse(Assert.java:47)
    at com.mogujie.natasha.TestBase.af(TestBase.java:98)
    at com.mogujie.lib.UserModelTest.testcurrusernotnull(UserModelTest.java:28)
```

关于JUnit Rule的使用，大家可以自行google一下，也不难。

7. 善于利用AndroidStudio来加快你写测试的速度

AndroidStudio有很多feature可以帮助我们更快的写代码，比如code generation和live template。这点对于写正式代码也适用，不过对于写测试代码来说，效果更为突出。因为大部分测试代码的结构、风格都是类似的，在这里live template能起非常大的作用。此外，如果你先写测试，可以直接写一些还不存在的Class或method，然后alt+enter让AndroidStudio自动帮你生成。

8. 不要最求完美

刚开始的时候，不用追求测试代码的质量，也不用追求完美，如果有些地方不好写测试，可以先放放，以后再来补，有部分测试总比没有测试好。Martin Fowler说过

> Imperfect tests, run frequently, are much better than perfect tests that are never written at all.

然而等你熟悉写测试的方法以后，强烈建议先写测试！因为如果你先写了正式代码，那你对这写代码是如何work的已经有一个印象了，因此你往往会写出能顺利通过的测试，而忽略一些会让测试不通过的情况。如果先写测试，则能考虑得更全面。

9. 未来的打算

使用Groovy和RoboSpock或者是Kotlin和Spek，真正实现BDD，这是很可能的事情，只是目前我们这边还没太多那方面的实践，因此就不说太多了。以后有一定实践了，到时候可以再更大家交流。
文中部分代码：github

参考链接：
http://stackoverflow.com/questions/4105592/comparison-between-mockito-vs-jmockit-why-is-mockito-voted-better-than-jmockit
http://dontpanic.42.nl/2013/04/mockito-powermock-vs-jmockit.html
http://endran.nl/blog/mockito-vs-jmockit/
http://martinfowler.com/articles/continuousIntegration.html