安卓单元测试(十一)：异步代码怎么测试

chriszou.com/2016/08/06/android-unit-testing-async

August 6, 2016

这是被问得最多的问题之一。。。

问题

今天讲一个我们讨论群里面被问得最多的一个问题：怎么测试异步操作。问题很明显，测试方法跑完了的时候，被测代码可能还没跑完，这就有问题了。比如下面的类：

```
public class RepoModel {
    private Handler mUiHandler = new Handler(Looper.getMainLooper());
    public void loadRepos(final RepoCallback callback) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000);
                    final List<Repo> repos = new ArrayList<>();
                    repos.add(new Repo("android-unit-testing-tutorial",
                                       "A repo that demos how to do android unit testing"));
                    mUiHandler.post(new Runnable() {
                        @Override
                        public void run() {
                            callback.onSuccess(repos);
                        }
                    });
                } catch (final InterruptedException e) {
                    e.printStackTrace();
                    mUiHandler.post(new Runnable() {
                        @Override
                        public void run() {
                            callback.onFailure(500, e.getMessage());
                        }
                    });
                }
            }
        }).start();
    }
    interface RepoCallback {
        void onSuccess(List<Repo> repos);
        void onFailure(int code, String msg);
    }
}
```

在上面的例子中，`loadRepos()`方法里面new了一个线程来异步的加载repo。如果我们按正常的方式写对应的测试：

```
@RunWith(RobolectricGradleTestRunner.class)
@Config(constants = BuildConfig.class, sdk = 21)
public class RepoModelTest {
    @Test
    public void testLoadRepos() throws Exception {
        RepoModel model = new RepoModel();
        final List<Repo> result = new ArrayList<>();
        model.loadRepos(new RepoCallback() {
            @Override
            public void onSuccess(List<Repo> repos) {
                result.addAll(repos);
            }
            @Override
            public void onFailure(int code, String msg) {
                fail();
            }
        });
        assertEquals(1, result.size());
    }
}
```

你会发现上面的测试方法永远会fail，这是因为在执行`assertEquals(1, result.size());`的时候，`loadRepos()`里面启动的线程还没执行完毕呢，因此，callback里面的`result.addAll(repos);`也没有得到执行，所以`result.size()`返回永远是0。

要解决这个问题，或者更general的说，要测试异步代码，有两种思路，一是等异步代码执行完了再执行assert操作，二是将异步变成同步。
接下来讲讲，具体怎么样用这两种思路来测试异步代码。

思路1，等待异步代码执行完毕：快使用CountDownLatch！

在上面的例子中，我们要做的，其实是等待Callback里面的代码执行完毕。要达到这个目的，有一个非常好用的神器，那就是`CountDownLatch`。`CountDownLatch`是一个类，它有两对配套使用的方法，那就是`countDown()`和`await()`。`await()`方法会阻塞当前线程，直到`countDown()`被调用了一定的次数，这个次数就是在创建这个`CountDownLatch`对象时，传入的构造参数。比如：

```
CountDownLatch latch = new CountDownLatch(3);
//.....
//下面这行代码会让当前线程一直停在这里
//直到latch.countDown()被调用了3次（一般是在其它线程）
latch.await();
```

使用`CountDownLatch`来实现上面例子的单元测试，方法如下：

```
@RunWith(RobolectricGradleTestRunner.class)
@Config(constants = BuildConfig.class, sdk = 21)
public class RepoModelTest {
    @Test
    public void testLoadRepos() throws Exception {
        RepoModel model = new RepoModel();
        final List<Repo> result = new ArrayList<>();
        final CountDownLatch latch = new CountDownLatch(1); //创建CountDownLatch
        model.loadRepos(new RepoCallback() {
            @Override
            public void onSuccess(List<Repo> repos) {
                result.addAll(repos);
                latch.countDown();  //这里countDown，外面的await()才能结束
            }
            @Override
            public void onFailure(int code, String msg) {
                fail();
            }
        });
        latch.await();
        assertEquals(1, result.size());
    }
}
```

`CountDownLatch` 的工作原理类似于倒序计数，刚开始设定了一个数字，每次 `countDown()` 这个数字减一， `await()` 方法会一直等待，直到这个数字为0。 `await()` 还有一个重载方法，可以用来指定你要等待多久，因为很多时候你不想一直等下去。你想等待一会，如果没等到，那就做别的事情。这种时候你就可以使用这个重载方法：

```
//等待2秒钟，如果2秒以后，计数是0了，则返回True，否则返回False。
latch.await(2, TimeUnit.SECONDS);
```

`CountDownLatch` 的使用还是比较简单直观的。基本上，所有有Callback的异步，包括RxJava（Subscriber其实就相当于Callback的角色），都可以使用这种方式来做测试，不论内部是通过什么样的方式来实现异步的。不过，使用 `CountDownLatch` 来做单元测试，有一个很大的限制，那就是 `countDown()` 必须可以在测试代码里面写，换句话说，必需有Callback。如果被测的异步方法（比如上面的l `oadRepos()`)不是通过Callback的方式来通知结果，而是通过post EventBus的Event来通知外面方法运行的结果，那 `CountDownLatch` 是无法解决这个异步方法的单元测试问题的。

此外， `CountDownLatch` 还有一个缺点，那就是写起来有点罗嗦，创建对象、调用 `countDown()` 、调用 `await()` 都必须手动写，而且还没有通用性，你没有办法抽出一个类或方法来简化代码。

思路2，将异步变成同步

将异步变成同步也是解决异步代码测试问题的一种比较直观的思路。使用这种思路的主要手段是依赖注入，但是根据实现异步的方式不同，也有一些其它的手段。下面介绍几种常见的

异步实现，以及相应的单元测试的方法。

直接new Thread的情况

呃，如果你直接在正式代码里面 `new Thread()` 来做异步，那么你的代码是没有办法变成同步的，换成 `Executor` 这种方式来做吧。

Executor或ExecutorService的情况

如果你的代码是通过 `Executor` 或 `ExecutorService` 来做异步的，那在测试中把异步变成同步的做法，跟在测试中使用mock对象的方法是一样的，那就是使用依赖注入。在测试代码里面将同步的 `Executor` 注入进去。创建同步的 `Executor` 对象很简单，以下就是一个同步的 `Executor` ：

```
Executor executor = new Executor() {
   @Override
   public void execute(Runnable command) {
      command.run();
   }
};
```

当然，你可以使用一个辅助的factory方法来做这件事情。至于怎么样将这个同步的 `Executor` 在测试里面替换掉真实异步的那个 `Executor` ，就是依赖注入的问题了。具体的做法请参见系列第5篇：依赖注入，将mock方便的用起来，如果你使用了Dagger2的话，请看第六篇：使用dagger2来做依赖注入，以及在单元测试中的应用。

AsyncTask

笔者建议是不要使用 `AsyncTask` ，这个东西有很多问题，其中之一是它的行为是很难预测的，之二是如果你在 `Activity` 里面使用的话，其实这部分代码往往是不应该放在 `Activity` 里面的。

不过，如果你实在需要使用 `AsyncTask` ，同时又想对这些代码作单元测试的话，建议是使用 `AsyncTask#executeOnExecutor()` 而不是直接使用 `AsyncTask#execute()` ，然后通过依赖注入的方式，在测试环境下将同步的 `Executor` 注入进去。

RxJava

这个是不得不提的一种方法，随着越来越多的人使用RxJava来做异步操作，RxJava代码的单元测试也是经常被问到的一个问题。通常，我们是用下面的方式来使用RxJava的。

```
someMethodsThatReturnsAnObservable().subscribeOn(Schedulers.io()).observeOn(AndroidSchedu
```

这里的问题是，`Schedulers.io()`会让`Observable`的某些操作运行在另外一个线程中，从而导致本文开头说的那个问题。在这种情况下，要把RxJava的操作变成同步的，也有2种方式，第一种方式是使用依赖注入，将`subscribeOn`（也许还有`observeOn`）的`scheduler`从外面注入进来。第二种方式是使用RxJava提供的Util hook：`RxJavaPlugins#registerSchedulersHook()`，让`Schedulers.io()`返回当前测试运行所在的个线程，而不是另外的一个线程。具体做法请看一个例子：

```
public class RepoModel {
    private Handler mUiHandler = new Handler(Looper.getMainLooper());
    public RepoModel() {
    }
    //待测方法
    public Observable<List<Repo>> loadRepos() {
        return Observable.create(new OnSubscribe<List<Repo>>() {
            @Override
            public void call(Subscriber<? super List<Repo>> subscriber) {
                try {
                    //Imagine you're getting repos from network or database
                    Thread.sleep(2000);
                    final List<Repo> repos = new ArrayList<>();
                    repos.add(new Repo("android-unit-testing-tutorial",
                    "A repo that demos how to do android unit testing"));
                    if (!subscriber.isUnsubscribed()) {
                        subscriber.onNext(repos);
                        subscriber.onCompleted();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    if (!subscriber.isUnsubscribed()) {
                        subscriber.onError(e);
                    }
                }
            }
        }).subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread());
    }
}
@RunWith(RobolectricGradleTestRunner.class)
@Config(constants = BuildConfig.class, sdk = 21)
public class RepoModelTest {
    @Test
    public void testLoadReposInRx() {
        // 让Schedulers.io()返回当前线程
        RxJavaPlugins.getInstance().registerSchedulersHook(new RxJavaSchedulersHook() {
            @Override
            public Scheduler getIOScheduler() {
                return Schedulers.immediate();
            }
        });
        RepoModel model = new RepoModel();
        final List<Repo> result = new ArrayList<>();
```

```
        model.loadRepos().subscribe(new Action1<List<Repo>>() {
            @Override
            public void call(List<Repo> repos) {
                result.addAll(repos);
            }
        });
        assertEquals(1, result.size());
    }
}
```

怎么样，很简单吧？实事上，我们还可以使用

```
RxAndroidPlugins.getInstance().registerSchedulersHook(new RxAndroidSchedulersHook() {
            @Override
            public Scheduler getMainThreadScheduler() {
                return Schedulers.immediate();
            }
        });
```

来让 `AndroidSchedulers.mainThread()` 返回当前线程，这样，如果其它地方没有用到Android的类，我们就可以摆脱Robolectric了。这种方式的好处是你可以不用对你的正式代码作依赖注入处理，同时是通用的，你可以在 `@Before` 里面或其它地方作一次性的初始化，然后这个测试类的所有测试方法都可以使用相同的效果。

小结

本文介绍了几种异步代码的单元测试方法，实际上，在Android上实现异步当然不止这几种方式，还有 `ThreadHandler` 、 `IntentService` 、 `Loader` 等方式，但是笔者对于这些方式使用得较少，因此一时想不出很好的解释方式，但是思想应该都是一样的，那就是要么想办法等待异步线程结束，要么把异步变成同步。

文中的代码在github的这个repo。

希望本文能帮助到你。