Android单元测试（五）：依赖注入，将mock方便的用起来

chriszou.com/2016/05/05/android-unit-testing-di

May 5, 2016

在上一篇文章中，我们讲了要将mock出来的dependency真正使用起来，需要在测试环境下通过某种方式set 到用到它的那个对象里面进去，替换掉真实的实现。我们前面举的例子是：

```
public class LoginPresenter {
   private UserManager mUserManager = new UserManager();
   public void login(String username, String password) {
      //。。。some other code
      mUserManager.performLogin(username, password);
   }
}
```

在测试 `LoginPresenter#login()` 时，为了能够将mock出来的 `UserManager` set 到 `LoginPresenter` 里面，我们前面的做法是简单粗暴，给 `LoginPresenter` 加一个 `UserManager` 的setter。然而这种做法毕竟不是很优雅，一般来说，我们正式代码里面是不会去调用这个setter，修改 `UserManager` 这个对象的。因此这个setter存在的意义就纯粹是为了方便测试。这个虽然不是没有必要，却不是太好看，因此在有选择的情况下，我们不这么做。在这里，我们介绍依赖注入这种模式。

对于依赖注入（Dependency Injection，以下简称DI）的准确定义可以在这里找到。它的基本理念这边简单描述下，首先这是一种代码模式，这个模式里面有两个概念：Client和Dependency。假如你的代码里面，一个类用到了另外一个类，那么前者叫Client，后者叫Dependency。结合上面的例子，`LoginPresenter` 用到了 `UserManager` ，那么 `LoginPresenter` 叫Client， `UserManager` 叫Dependency。当然，这是个相对的概念，一个类可以是某个类的Dependency，却是另外一个类的Client。比如说如果 `UserManager` 里面用到了 `Retrofit` ，那么相对于 `Retrofit` ， `UserManager` 又是Client。DI的基本思想就是，对于Dependency的创建过程，并不在Client里面进行，而是由外部创建好，然后通过某种方式set到Client里面。这种模式，就叫做依赖注入。

是的，依赖注入就是这么简单的一个概念，这边需要澄清的一点是，这个概念本身跟dagger2啊，RoboGuice这些框架并没有什么关系。现在很多介绍DI的文章往往跟dagger2是在一起的，因为dagger2的使用相对来说不是很直观，所以导致很多人认为DI是多么复杂的东西，甚至认为只能用dagger等框架来实现依赖注入，其实不是这样的。实现依赖注入很简单，dagger这些框架只是让这种实现变得**更加**简单，简洁，优雅而已。

DI的常见实现方式

下面介绍DI的实现方式，通常来说，这里是大力介绍dagger2的地方。但是，虽然dagger2的确是非常好的东西，然而如果我直接介绍dagger2的话，会很容易导致一个误区，认为在测试的时候，也只能用dagger来做依赖注入或创建对应的测试类，因此，我这边刻意不介绍dagger。先让大家知道最基本的DI怎么实现，然后在测试的时候如何更方便高效的使用。

实现DI这种模式其实很简单，有多种方式，上一篇文章中提到的setter，其实就是实现DI的一种方式，叫做 *setter injection* 。此外，通过方法的参数传递进去（*argument injection*），也是实现DI的一种方式：

```
public class LoginPresenter {
   //这里，LoginPresenter不再持有UserManager的一个引用，而是作为方法参数直接传进去
   public void login(UserManager userManager, String username, String password) {
      //... some other code
      userManager.performLogin(username, password);
   }
}
```

然而更常用的方式，是将Dependency作为Client的构造方法的参数传递进去：

```
public class LoginPresenter {
   private final UserManager mUserManager;
   //将UserManager作为构造方法参数传进来
   public LoginPresenter(UserManager userManager) {
      this.mUserManager = userManager;
   }
   public void login(String username, String password) {
      //... some other code
      mUserManager.performLogin(username, password);
   }
}
```

这种实现DI的模式叫 *Constructor Injection*。其实一般来说，提到DI，指的都是这种方式。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。因此，尽量使用 *Constructor injection*。

说到这里，你可能会有一个疑问，如果把依赖都声明在Constructor的参数里面，这会不会让这个类的Constructor参数变得非常多？如果真的发生这种情况了，那往往说明这个类的设计是有问题的，需要重构。为什么呢？我们代码里面的类，一般可以分为两种，一种是Data类，比如说UserInfo，OrderInfo等等。另外一种是Service类，比如 `UserManager`，AudioPlayer等等。所以这个问题就有两种情况了：

1. 如果Constructor里面传入的很多是基本类型的数据或数据类，那么或许你要做的，是创建一个（或者是另一个）数据类把这些数据封装一下，这个过程的价值可是大大滴！而不仅仅是封装一下参数的问题，有了一个类，很多的方法就可以放到这个类里面了。这点请参考Martin Fowler的《重构》第十章“Introduce Parameter Object”。

2. 如果传入的很多是service类，那么这说明这个类做的事情太多了，不符合单一职责的原则（Single Responsibility Principle，SRP），因此，需要重构。

接下来说回我们的初衷：DI在测试里面的应用。

DI在单元测试里面的应用

所谓DI在单元测试里面的应用，其实说白了就是使用DI模式，将mock出来的Dependency set到Client里面去。我相信这篇文章解释到这里，那么答案也就比较明显了，为了强调我们要尽量使用 *Constructor injection*，对于 *setter Injection* 和 *Argument injection* 这边就不做代码示例了。
如果你的代码使用的是 *Constructor injection*：

```
public class LoginPresenter {
    private final UserManager mUserManager;
    //将UserManager作为构造方法参数传进来
    public LoginPresenter(UserManager userManager) {
        this.mUserManager = userManager;
    }
    public void login(String username, String password) {
        //... some other code
        mUserManager.performLogin(username, password);
    }
}
```

其中我们要测的方法是 `login()`，要验证 `login()` 方法调用了 `mUserManager` 的 `performLigon()`。对应的测试方法如下：

```
public class LoginPresenterTest {
    @Test
    public void testLogin() {
        UserManager mockUserManager = Mockito.mock(UserManager.class);
        LoginPresenter presenter = new LoginPresenter(mockUserManager);  //创建的时候，讲mock传进去
        presenter.login("xiaochuang", "xiaochuang password");
        Mockito.verify(mockUserManager).performLogin("xiaochuang", "xiaochuang password");
    }
}
```

很简单，对吧。

小结

这篇文章介绍了DI的概念，以及在单元测试里面的应用，这里特意没有介绍dagger2的使用，目的是要强调：

1. 一个灵活的，易于测试的，符合SRP的，结构清晰的项目，关键在于要应用依赖注入这种模式，而不是用什么来做依赖注入。

2. 等你学会使用dagger以后，要记得在测试的时候，如果可以直接mock dependency并传给被测类，那就直接创建，不是一定要使用dagger来做DI

然而如果完全不使用框架来做DI，那么在正式代码里面就有一个问题了，那就是dependency的创建工作就交给上层client去处理了，这可不是件好事情。想想看，`LoginActivity`里面创建`LoginPresenter`的时候，还得知道`LoginPresenter`用了`UserManager`。然后创建一个`UserManager`对象给`LoginPresenter`。对于`LoginActivity`来说，它觉得我才懒得管你用什么样的`UserManager`呢，我只想告诉你login的时候，你给我老老实实的login就好了，你用什么Manager我不管。所以，直接在`LoginActivity`里面创建`UserManager`，可能不是个好的选择。那怎么样算是一个好的选择呢？dagger2给了我们答案。
于是下一篇文章我们介绍dagger2。

文中的代码在github这个项目里面。

最后，如果你对安卓单元测试感兴趣，欢迎加入我们的交流群，因为群成员超过100人，没办法扫码加入，请关注下方公众号获取加入方法。