Android单元测试（六）：使用dagger2来做依赖注入，以及在单元测试中的应用

chriszou.com/2016/05/09/android-unit-testing-di-dagger

May 9, 2016

> 注：
> 1. 代码中的 //<= 表示新加的、修改的等需要重点关注的代码
>
> 2. Class#method表示一个类的instance method，比如 LoginPresenter#login 表示LoginPresenter的login（非静态）方法。

问题

在前一篇文章中，我们讲述了依赖注入的概念，以及依赖注入对单元测试极其关键的重要性和必要性。在那篇文章的结尾，我们遇到了一个问题，那就是如果不使用DI框架，而全部采用手工来做DI的话，那么所有的Dependency都需要在最上层的client来生成，这可不是件好事情。继续用我们前面的例子来具体说明一下。

假设有一个登录界面，`LoginActivity`，他有一个`LoginPresenter`，`LoginPresenter`用到了`UserManager`和`PasswordValidator`，为了让问题变得更明显一点，我们假设`UserManager`用到`SharedPreference`（用来存储一些用户的基本设置等）和`UserApiService`，而`UserApiService`又需要由`Retrofit`创建，而`Retrofit`又用到`OkHttpClient`（比如说你要自己控制timeout、cache等东西）。

应用DI模式，UserManager的设计如下：

```
public class UserManager {
    private final SharedPreferences mPref;
    private final UserApiService mRestAdapter;
    public UserManager(SharedPreferences preferences, UserApiService userApiService) {
        this.mPref = preferences;
        this.mRestAdapter = userApiService;
    }
    /**Other code*/
}
```

LoginPresenter的设计如下：

```
public class LoginPresenter {
    private final UserManager mUserManager;
    private final PasswordValidator mPasswordValidator;
    public LoginPresenter(UserManager userManager, PasswordValidator passwordValidator) {
        this.mUserManager = userManager;
        this.mPasswordValidator = passwordValidator;
    }
    /**Other code*/
}
```

在这种情况下，最终的client LoginActivity里面要new一个presenter，需要做的事情如下：

```
public class LoginActivity extends AppCompatActivity {
    private LoginPresenter mLoginPresenter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        OkHttpClient okhttpClient = new OkHttpClient.Builder()
                .connectTimeout(30, TimeUnit.SECONDS)
                .build();
        Retrofit retrofit = new Retrofit.Builder()
                .client(okhttpClient)
                .baseUrl("https://api.github.com")
                .build();
        UserApiService userApiService = retrofit.create(UserApiService.class);
        SharedPreferences preferences = PreferenceManager.getDefaultSharedPreferences(this);
        UserManager userManager = new UserManager(preferences, userApiService);
        PasswordValidator passwordValidator = new PasswordValidator();
        mLoginPresenter = new LoginPresenter(userManager, passwordValidator);
    }
}
```

这个也太夸张了，`LoginActivity`所需要的，不过是一个`LoginPresenter`而已，然而它却需要知道`LoginPresenter`的Dependency是什么，`LoginPresenter`的Dependency的Dependency又是什么，然后new一堆东西出来。而且可以预见的是，这个app的其他地方也需要这里的`OkHttpClient`、`Retrofit`、`SharedPreference`、`UserManager`等等dependency，因此也需要new这些东西出来，造成大量的代码重复，和不必要的object instance生成。然而如前所述，我们又必须用到DI模式，这个怎么办呢？

想想，如果能达到这样的效果，那该有多好：我们只需要在一个类似于dependency工厂的地方统一生产这些dependency，以及这些dependency的dependency。所有需要用到这些Dependency的client都从这个工厂里面去获取。而且更妙的是，一个client（比如说`LoginActivity`）只需要知道它直接用到的Dependency（`LoginPresenter`），而不需要知道它的Dependency（`LoginPresenter`）又用到哪些Dependency（`UserManager`和`PasswordValidator`）。系统自动识别出这个依赖关系，从工厂里面把需要的Dependency找到，然后把这个client所需要的Dependency创建出来。

有这样一个东西，帮我们实现这个效果吗？相信聪明的你已经猜到了，回答是肯定的，它就是我们今天要介绍的dagger2。

解药：Dagger2

在dagger2里面，负责生产这些Dependency的统一工厂叫做 ***Module*** ，所有的client最终是要从module里面获取Dependency的，然而他们不是直接向module要的，而是有一个专门的“工厂管理员”，负责接收client的要求，然后到Module里面去找到相应的Dependency，提供给client们。这个“工厂管理员”叫做 ***Component***。基本上，这是dagger2里面最重要的两个概念。

下面，我们来看看这两个概念，对应到代码里面，是怎么样的。

生产Dependency的工厂：Module

首先是Module，一个Module对应到代码里面就是一个类，只不过这个类需要用dagger2里面的一个annotation `@Module` 来标注一下，来表示这是一个Module，而不是一个普通的类。我们说Module是生产Dependency的地方，对应到代码里面就是Module里面有很多方法，这些方法做的事情就是创建Dependency。用上面的例子中的Dependency来说明：

```
@Module
public class AppModule {
   public OkHttpClient provideOkHttpClient() {
      OkHttpClient okhttpClient = new OkHttpClient.Builder()
            .connectTimeout(30, TimeUnit.SECONDS)
            .build();
      return okhttpClient;
   }
   public Retrofit provideRetrofit(OkHttpClient okhttpClient) {
      Retrofit retrofit = new Retrofit.Builder()
            .client(okhttpClient)
            .baseUrl("https://api.github.com")
            .build();
      return retrofit;
   }
}
```

在上面的Module（`AppModule`）中，有两个方法 `provideOkHttpClient()` 和 `provideRetrofit(OkHttpClient okhttpClient)` ，分别创建了两个Dependency， `OkHttpClient` 和 `Retrofit` 。但是呢，我们也说了，一个Module就是一个类，这个类有一些生产Dependency的方法，但它也可以有一些正常的，不是用来生产Dependency的方法。那怎么样让管理员知道，一个Module里面哪些方法是用来生产Dependency的，哪些不是呢？为了方便做这个区分，dagger2规定，所有生产Dependency的方法必须用 `@Provides` 这个annotation标注一下。所以，上面的 `AppModule` 正确的写法应该是：

```
@Module
public class AppModule {
    @Provides
    public OkHttpClient provideOkHttpClient() {
        OkHttpClient okhttpClient = new OkHttpClient.Builder()
                .connectTimeout(30, TimeUnit.SECONDS)
                .build();
        return okhttpClient;
    }
    @Provides
    public Retrofit provideRetrofit(OkHttpClient okhttpClient) {
        Retrofit retrofit = new Retrofit.Builder()
                .client(okhttpClient)
                .baseUrl("https://api.github.com")
                .build();
        return retrofit;
    }
}
```

这种用来生产Dependency的、用 `@Provides` 修饰过的方法叫**Provider方法**。这里要注意第二个Provider方法 `provideRetrofit(OkHttpClient okhttpClient)` ，这个方法有一个参数，是 `OkHttpClient` 。这是因为创建一个 `Retrofit` 对象需要一个 `OkHttpClient` 的对象，这里通过参数传递进来。这样做的好处是，当Client向管理员（Component）索要一个 `Retrofit` 的时候，Component会自动找到Module里面找到生产Retrofit的这个 `provideRetrofit(OkHttpClient okhttpClient)` 方法，找到以后试图调用这个方法创建一个 `Retrofit` 对象，返回给Client。但是调用这个方法需要一个 `OkHttpClient` ，于是Component又会去找其他的provider方法，看看有没有哪个会生产 `OkHttpClient` 。于是就找到了上面的第一个provider方法： `provideOkHttpClient()` 。找到以后，调用这个方法，创建一个 `OkHttpClient` 对象，再调用 `provideRetrofit(OkHttpClient okhttpClient)` 方法，把刚刚创建的 `OkHttpClient` 对象传进去，创建出一个 `Retrofit` 对象，返回给Client。当然，如果最后找到的 `provideOkHttpClient()` 方法也需要其他参数，那么管理员还会继续递归的找下去，直到所有的Dependency都被满足了，再一个一个创建Dependency，然后把最终Client需要的Dependency呈递给Client。

很好，现在我们把文章开头的例子中的所有Dependency都用这种方式，在 `AppModule` 里面声明一个provider方法：

```
@Module
public class AppModule {
    @Provides
    public OkHttpClient provideOkHttpClient() {
        OkHttpClient okhttpClient = new OkHttpClient.Builder()
                .connectTimeout(30, TimeUnit.SECONDS)
                .build();
        return okhttpClient;
    }
    @Provides
    public Retrofit provideRetrofit(OkHttpClient okhttpClient) {
        Retrofit retrofit = new Retrofit.Builder()
                .client(okhttpClient)
                .baseUrl("https://api.github.com")
                .build();
        return retrofit;
    }
    @Provides
    public UserApiService provideUserApiService(Retrofit retrofit) {
        return retrofit.create(UserApiService.class);
    }
    @Provides
    public SharedPreferences provideSharedPreferences(Context context) {
        return PreferenceManager.getDefaultSharedPreferences(context);
    }
    @Provides
    public UserManager provideUserManager(SharedPreferences preferences, UserApiService
service) {
        return new UserManager(preferences, service);
    }
    @Provides
    public PasswordValidator providePasswordValidator() {
        return new PasswordValidator();
    }
    @Provides
    public LoginPresenter provideLoginPresenter(UserManager userManager, PasswordValidator
validator) {
        return new LoginPresenter(userManager, validator);
    }
}
```

上面的代码如果你仔细看的话，会发现一个问题，那就是其中的SharedPreference provider方法 `provideSharedPreferences(Context context)` 需要一个context对象，但是 `AppModule` 里面并没有context 的Provider方法，这个怎么办呢？对于这个问题，你可以再创建一个context provider方法，但是context对象从哪来呢？我们可以自定义一个Application，里面提供一个静态方法返回一个context，这种做法相信大家都干过。Application类如下：

```
public class MyApplication extends Application {
    private static Context sContext;
    @Override
    public void onCreate() {
        super.onCreate();
        sContext = this;
    }
    public static Context getContext() {
        return sContext;
    }
}
```

provider方法如下：

```
    @Provides
    public Context provideContext() {
        return MyApplication.getContext();
    }
```

但是这种方法不是很好，为什么呢，因为context的获得相当于是写死了，只能从MyApplication.getContext()，如果测试环境下想把Context换成别的，还要给MyApplication定义一个setter，然后调用MyApplication.setContext(...)，这个就绕的有点远。更好的做法是，把Context作为 `AppModule` 的一个构造参数，从外面传进来（应用DI模式，还记得吗？）：

```
@Module
public class AppModule {
    private final Context mContext;
    public AppModule(Context context) {
        this.mContext = context;
    }
    @Provides
    public Context provideContext() {
        return mContext;
    }
    //其他的provider方法
}
```

是的，一个Module就是一个正常的类，它也可以有构造方法，以及其他正常类的特性。你可能会想那给构造函数的context对象从哪来呢？别急，这个问题马上解答。

Dependency工厂管理员：Component

前面我们讲了dagger2的一半，就是生产Dependency的工厂：Module。接下来我们讲另一半，工厂管理员：Component。跟Module不同的是，我们在实现Component时，不是定义一个类，而是定义一个接口（interface）：

```
public interface AppComponent {
}
```

名字可以随便取，跟Module需要用 `@Module` 修饰一下类似的，一个dagger2的Component需要用 `@Component` 修饰一下，来标注这是一个dagger2的Component，而不是一个普通的interface，所以正确的定义方式是：

```
@Component
public interface AppComponent {
}
```

在实际情况中，可能有多个Module，也可能有多个Component，那么当Component接收到一个Client的Dependency请求时，它怎么知道要从哪个Module里面去找这些Dependency呢？它不可能遍历我们的每一个类，然后找出所有的Module，再遍历所有Module的Provider方法，去找Dependency，这样先不说能不能做到，就算做得到，效率也太低了。因此dagger2规定，我们在定义Component的时候，必须指定这个管理员“管理”哪些工厂（Module）。指定的方法是，把需要这个Component管理的Module传给 `@Component` 这个注解的modules属性（或者叫方法？），如下：

```
@Component(modules = {AppModule.class}) //<=
public interface AppComponent {
}
```

modules属性接收一个数组，里面是这个Component管理的所有Module。在上面的例子中， `AppComponent` 只管理 `AppModule` 一个。

Component给Client提供Dependency的方法

前面我们讲了Module和Component的实现，接下来就是Component怎么给Client提供Dependency的问题了。一般来说，有两种，当然总共不止这两种，只不过这两种最常用，也最好理解，一般来说用这两种就够了，因此这里不赘述其他的方法。

方法一：在Component里面定义一个返回Dependency的方法

第一种是在Component里面定义一个返回Dependency的方法，比如LoginActivity需要LoginPresenter，那么我们可以在 `AppComponent` 里面定义一个返回 `LoginPresenter` 的方法：

```
@Component(modules = {AppModule.class})
public interface AppComponent {
   LoginPresenter loginPresenter();
}
```

你可能会好奇，为什么Component只需要定义成接口就行了，不是应该定义一个类，然后自己使用Module去做这件事吗？如果是这样的话，那就太low了。dagger2的工作原理是，在你的java代码编译成字节码的过程中，dagger2会对所有的Component（就是用 `@Component` 修饰过的interface）进行处理，自动生成一个实现了这个interface的类，生成的类名是Component的名字前面加上“Dagger”。比如我们定义的 `AppComponent` ，对应的自动生成的类叫做 `DaggerAppComponent` 。我们知道，实现一个interface需要实现里面

的所有方法，因此，`DaggerAppComponent`是实现了`loginPresenter();`这个方法的。实现的方式大致就是从`AppComponent`管理的`AppModule`里面去找`LoginPresenter`的Provider方法，然后调用这个方法，返回一个`LoginPresenter`。

因此，使用这种方式，当Client需要Dependency的时候，首先需要用`DaggerAppComponent`这个类创建一个对象，然后调用这个对象的`loginPresenter()`方法，这样Client就能获得一个`LoginPresenter`了，这个`DaggerAppComponent`对象的创建及使用方式如下：

```
public class LoginActivity extends AppCompatActivity {
    private LoginPresenter mLoginPresenter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        AppComponent appComponent = DaggerAppComponent.builder().appModule(new
AppModule(this)).build();  //<=
        mLoginPresenter = appComponent.loginPresenter();   //<=
    }
}
```

总结一下，我们到现在为止，做了什么：

1. 我们定义了一个`AppModule`类，里面定义了一些Provider方法
2. 定义了一个`AppComponent`，里面定义了一个返回`LoginPresenter`的方法`loginPresenter()`。

就这样，我们便可以使用`DaggerAppComponent.builder().appModule(new AppModule(this)).build().loginPresenter();`来获取一个`LoginPresenter`对象了。

这简直就是magic，不是吗？

如果不是dagger2，而是我们自己来实现这个`AppComponent` interface，想想我们需要做哪些事情：

1. 定义一个Constructor，接受一个`AppModule`对象，保存在field中（mAppModule)
2. 实现loginPresenter()方法，调用mAppModule的`provideLoginPresenter(UserManager userManager, PasswordValidator validator)`方法，这时候发现这个方法需要两个参数`UserManager`和`PasswordValidator`。
3. 调用`provideUserManager(SharedPreferences preferences, UserApiService service)`来获取一个`UserManager`，这时候发现这个方法又需要两个参数`SharedPreferences`和`UserApiService`。
4. 调用`provideSharedPreferences(Context context)`来获取一个SharedPreference，这时候发现先要有一个context
5. 。。。
6. 。。。

7. 。。。

说白了，就是把文章开头我们写的那段代码又实现了一遍，而使用dagger2，我们就做了前面描述的两件事而已，这里面错综复杂的Dependency关系dagger2帮我们自动理清了，生成相应的代码，去调用相应的Provider方法，满足这些依赖关系。

也许这里举得这个例子不足以让你觉得有什么大不了的，但是你要知道，一个正常的App，可不仅仅有一个Login page而已，稍微大点的App，Dependency都有几百甚至上千个，对于服务器程序来说，Dependency则更多。对于这点，大家可以去看Dagger2主要作者的这个视频，他里面提到了Google一个android app有3000行代码专门来管理Dependency，而一个Server app甚至有10万行这样的代码。这个时候要去手动new这些dependency、并且要以正确的顺序new出来，简直会要人命。而且让问题更加棘手的是，随着app的演进需求的变更，Dependency之间的关系也在动态的变化。比如说 `UserManager` 不再使用 `SharedPreference` ，而是使用database，这个时候 `UserManager` 的构造函数里面少了一个 `SharedPreferences` ，多了一个 `DatabaseHelper` 这样的东西，那么如果使用正常的方式管理Dependency，所有 `new UserManager` 的地方都要改，而是用dagger2，你只需要在 `AppModule` 里面添加一个DatabaseHelper Provider方法，同时把 `UserManager` 的provider方法第一参数从 `SharedPreferences` 改成 `DatabaseHelper` 就好了，所有用到 `UserManager` 的地方不需要做任何更改， `LoginPresenter` 不需要做任何更改， `LoginActivity` 不需要任何更改，这难道不是magic吗？

说点题外话，这种把问题(我们这里是依赖关系)描述出来，而不是把实现过程写出来的编程风格叫Declarative programming，跟它对应的叫Imperative Programming，相对于后者，前者的优势是：可读性更高，side effect更少，可扩展性更高等等。这是一种编程风格，跟语言、框架无关。当然，有的语言或框架天生就能让程序员更容易的使用这种style来编程。这方面最显著的当属Prolog，有兴趣的可以去了解下，绝对mind-blowing！

对于Java或Android开发者来说，想让我们的代码更加declarative，最好的方式是使用dagger2和RxJava。

方法二：Field Injection

话说回来，我们继续介绍dagger2，前面我们介绍了Component给Client提供Dependency的第一种方式，接下来继续介绍第二种方式，这种方式叫 *Field injection* 。这里我们继续用 `LoginActivity` 的例子来说明， `LoginActivity` 需要一个 `LoginPresenter` 。那么使用这种方式的做法是，我们就在 `LoginActivity` 里面定义一个 `LoginPresenter` 的field，这个field需要使用 `@Inject` 修饰一下：

```
public class LoginActivity extends AppCompatActivity {
    @Inject                         //<=
    LoginPresenter mLoginPresenter;     //<=
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

然后在onCreate()里面，我们把 `DaggerAppComponent` 对象创建出来，调用这个对象的inject方法，把 `LoginActivity` 传进去：

```
public class LoginActivity extends AppCompatActivity {
    @Inject
    LoginPresenter mLoginPresenter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        AppComponent appComponent = DaggerAppComponent.builder().appModule(new
AppModule(this)).build(); //<=
        appComponent.inject(this); //<=
        //从此之后，mLoginPresenter就被实例化了
        //mLoginPresenter.isLogin()
    }
}
```

当然，我们需要先在 `AppComponent` 里面定义一个 `inject(LoginActivity loginActivity)` 方法：

```
@Component(modules = {AppModule.class})
public interface AppComponent {
    void inject(LoginActivity loginActivity);  //<=
}
```

`DaggerAppComponent` 实现这个方法的方式是，去 `LoginActivity` 里面所有被 `@Inject` 修饰的field，然后调用 `AppModule` 相应的Provider方法，赋值给这个field。这里需要注意的是， `@Inject` field不能使private，不然dagger2找不到这个field。

通常来说，这种方式比第一种方式更简单，代码也更简洁。假设 `LoginActivity` 还需要其他的Dependency，比如需要一个统计打点的Dependency（ `StatManager` ），那么你只需要在 `AppModule` 里面定义一个Provider方法，然后在 `LoginActivity` 里面声明另外一个field就好了：

```
public class LoginActivity extends AppCompatActivity {
    @Inject
    LoginPresenter mLoginPresenter;
    @Inject
    StatManager mStatManager;   //<=
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        AppComponent appComponent = DaggerAppComponent.builder().appModule(new
AppModule(this)).build();
        appComponent.inject(this);
    }
}
```

无论有多少个@Inject field，都只需要调用一次`appComponent.inject(this);`。用过了你就会觉得，恩，好爽！

不过，需要注意的一点是，这种方式不支持继承，比如说`LoginActivity`继承自一个`BaseActivity`，而`@Inject StatManager mStatManager;`是放在`BaseActivity`里面的，那么在`LoginActivity`里面调用`appComponent.inject(this);`并不会让`BaseActivity`里面的`mStatManager`得到实例化，你必须在`BaseActivity`里面也调用一次`appComponent.inject(this);`。

@Singleton和Constructor Injection

到这里，Client从Component获取Dependency的两种方式就介绍完毕。但是这里有个问题，那就是每次Client向Component索要一个Dependency，Component都会创建一个新的出来，这可能会导致资源的浪费，或者说很多时候不是我们想要的，比如说，`SharedPreferences`、`UserManager`、`OkHttpClient`，`Retrofit`这些都只需要一份就好了，不需要每次都创建一个instance，这个时候我们可以给这些Dependency的Provider方法加上@Singleton就好了。如：

```
@Module
public class AppModule {
    @Provides
    @Singleton        //<=
    public OkHttpClient provideOkHttpClient() {
        OkHttpClient okhttpClient = new OkHttpClient.Builder()
                .connectTimeout(30, TimeUnit.SECONDS)
                .build();
        return okhttpClient;
    }
    //other method
}
```

这样，当Client第一次请求一个 `OkHttpClient` ，dagger2会创建一个instance，然后保存下来，下一次Client再次请求一个 `OkHttpClient` 是，dagger2会直接返回上次创建好的，而不用再次创建instance。这就相当于用一种更简便、而且DI-able的方式实现了singleton模式。

这里再给大家一个bonus，如果你不需要做单元测试，而只是使用dagger2来做DI，组织app的结构的话，其实 `AppModule` 里面的很多Provider方法是不需要定义的。比如说在这种情况下， `LoginPresenter` 的Provider方法 `provideLoginPresenter(UserManager userManager, PasswordValidator validator)` 就不需要定义，你只需要在定义 `LoginPresenter` 的时候，给它的Constructor加上 `@Inject` 修饰一下：

```
public class LoginPresenter {
    private final UserManager mUserManager;
    private final PasswordValidator mPasswordValidator;
    @Inject
    public LoginPresenter(UserManager userManager, PasswordValidator passwordValidator) {
        this.mUserManager = userManager;
        this.mPasswordValidator = passwordValidator;
    }
    //other methods
}
```

dagger2会自动创建这个 `LoginPresenter` 所需要的Dependency是它能够提供的，所以会去Module里面找到这个 `LoginPresenter` 所需的Dependency，交给 `LoginPresenter` 的Constructor，创建好这Dependency，交给Client。这其实也是Client通过Component使用Dependency的一种方式，叫 *Constructor injection* （上一篇文章也提到*Constructor injection*，不过稍微有点不同，注意区分一下）同样的，在那种情况下， `UserManager` 的Provider方法也不需要定义，而只需要给 `UserManager` 的Constructor加上一个 `@Inject` 就好了。说白了，你只需要给那些不是通过Constructor来创建的Dependency（比如说SharedPreferences、UserApiService等）定义Provider方法。

有了 *Constructor injection* ，我们的代码又能得到进一步的简化，然而遗憾的是，这种方式将导致我们做单元测试的时候无法mock这中间的Dependency。说到单元测试，我们别忘了这个系列的主题T_T。。。那么接下来就介绍dagger2在单元测试里面的使用，以及为什么 *Constructor injection* 将导致单元测试里面无法mock这个Dependency。

dagger2在单元测试里面的使用

在介绍dagger2在单元测试里面的使用之前，我们先改进一下前面的代码，我们创建 `DaggerAppComponent` 的地方是在 `LoginActivity` ，其实这样不是很好，为什么呢？想想如果login以后其他地方也需要 `UserManager` ，那么我们又要创建一个 `DaggerAppComponent` ，这种地方是很多的，毕竟 `AppModule` 里面定义了一些整个app都要用到的Dependency，比如说Retrofit、SharedPreferences等等。如果每个需要用到的地方都创建一遍 `DaggerAppComponent` ，就导致了代码的重复和内存性能的浪费，理论上来说， `DaggerAppComponent` 对象整个app只需要一份就好了。所以我们在用dagger2的

时候，一般的做法是，我们会在app启动的时候创建好，放在某一个地方。比如，我们在自定义的Application#onCreate()里面创建好，然后放在某个地方，我个人习惯定义一个类叫ComponentHolder，然后放里面：

```
public class MyApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        AppComponent appComponent = DaggerAppComponent.builder()
                                .appModule(new AppModule(this))
                                .build();
        ComponentHolder.setAppComponent(appComponent);
    }
}
public class ComponentHolder {
    private static AppComponent sAppComponent;
    public static void setAppComponent(AppComponent appComponent) {
        sAppComponent = appComponent;
    }
    public static AppComponent getAppComponent() {
        return sAppComponent;
    }
}
```

然后在需要 `AppComponent` 的地方，使用 ComponentHolder.getAppComponent()来获取一个 `DaggerAppComponent` 对象：

```
public class LoginActivity extends AppCompatActivity {
    @Inject
    LoginPresenter mLoginPresenter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ComponentHolder.getAppComponent().inject(this);  //<=
    }
}
```

这样在用的地方，看起来代码也干净了很多。

到这里，我们就可以介绍在单元测试里面怎么来mock Dependency了。假设LoginActivity有两个EditText和一个login button，点击这个button，将从两个EditText里面获取用户名和密码，然后调用 `LoginPresenter` 的login方法：

```
public class LoginActivity extends AppCompatActivity {
    @Inject
    LoginPresenter mLoginPresenter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ComponentHolder.getAppComponent().inject(this);
        findViewById(R.id.login).setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                String username = ((EditText) findViewById(R.id.username)).getText().toString();
                String password = ((EditText) findViewById(R.id.password)).getText().toString();
                mLoginPresenter.login(username, password);
            }
        });
    }
}
```

我们现在要测的，就是当用户点击这个login button的时候，`mLoginPresenter`的login方法得到了调用，如果你看了这个系列的前面几篇文章，你就知道这里的`mLoginPresenter`需要mock掉。但是，这里的`mLoginPresenter`是从dagger2的component里面获取的，这里怎么把`mLoginPresenter`换成mock呢？

我们在回顾一下，其实`LoginActivity`只是向`DaggerAppComponent`索取了一个`LoginPresenter`，而`DaggerAppComponent`其实是调用了`AppModule`的`provideLoginPresenter()`方法来获得了一个`LoginPresenter`，返回给`LoginActivity`，也就是说，真正生产`LoginPresenter`的地方是在`AppModule`。还记得吗，我们创建`DaggerAppComponent`的时候，给它的builder传递了一个`AppModule`对象：

```
public class MyApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        AppComponent appComponent = DaggerAppComponent.builder()
                                .appModule(new AppModule(this)) //<= 这里这里
                                .build();
        ComponentHolder.setAppComponent(appComponent);
    }
}
```

其实`DaggerAppComponent`调用的`AppModule`对象，就是我们在创建它的时候传给那个builder的。那么，如果我们传给`DaggerAppComponent`的`AppModule`是一个mock对象，在这个mock对象的provideLoginPresenter()被调用的时候，返回一个mock的`LoginPresenter`，那么`LoginActivity`获得的，不就是一个mock的`LoginPresenter`了吗？

我们用代码来实现一下看看是什么样子，这里因为 `LoginActivity` 是android相关的类，因此需要用到robolectric这个framework，虽然这个我们还没有介绍到，但是代码应该看得懂，如下：

```
@RunWith(RobolectricGradleTestRunner.class) //Robolectric相关，看不懂的话忽略
@Config(constants = BuildConfig.class, sdk = 21) //同上
public class LoginActivityTest {
    @Test
    public void testActivityStart() {
        @Test
        public void testActivityStart() {
            AppModule mockAppModule = spy(new AppModule(RuntimeEnvironment.application)); //创建一个mockAppModule，这里不能spy(AppModule.class)，因为`AppModule`没有默认无参数的Constructor，也不能mock(AppModule.class),原因是dagger2的约束，Provider方法不能返回null，除非用@Nullable修饰
            LoginPresenter mockLoginPresenter = mock(LoginPresenter.class);  //创建一个mockLoginPresenter
            Mockito.when(mockAppModule.provideLoginPresenter(any(UserManager.class), any(PasswordValidator.class))).thenReturn(mockLoginPresenter);  //当mockAppModule的provideLoginPresenter()方法被调用时，让它返回mockLoginPresenter
            AppComponent appComponent = DaggerAppComponent.builder().appModule(mockAppModule).build();  //用mockAppModule来创建DaggerAppComponent
            ComponentHolder.setAppComponent(appComponent);  //记得放到ComponentHolder里面，这样LoginActivity#onCreate()里面通过ComponentHolder.getAppComponent()获得的就是这里创建的appComponent
            LoginActivity loginActivity = Robolectric.setupActivity(LoginActivity.class); //启动LoginActivity，onCreate方法会得到调用，里面的mLoginPresenter通过dagger2获得的，将是mockLoginPresenter
            ((EditText) loginActivity.findViewById(R.id.username)).setText("xiaochuang");
            ((EditText) loginActivity.findViewById(R.id.password)).setText("xiaochuang is handsome");
            loginActivity.findViewById(R.id.login).performClick();
            verify(mockLoginPresenter).login("xiaochuang", "xiaochuang is handsome");  //pass!
        }
    }
}
```

这就是dagger2在单元测试里面的应用。基本上就是mock Module的Provider方法，让它返回你想要的mock对象。这也解释了为什么说只用 *Constructor injection* 的话，会导致Dependency无法mock，因为没有对应的Provider方法来让我们mock啊。上面的代码看起来也许你会觉得有点多，然而实际开发中，上面测试方法里的第1、4、5行都是通用的，我们可以把他们抽到一个辅助类里面：

```
public class TestUtils {
  public static final AppModule appModule = spy(new
AppModule(RuntimeEnvironment.application));
  public static void setupDagger() {
    AppComponent appComponent =
DaggerAppComponent.builder().appModule(appModule).build();
    ComponentHolder.setAppComponent(appComponent);
  }
}
```

这样我们前面的测试方法就可以简化了：

```
public class LoginActivityTest {
  @Test
  public void testActivityStart() {
    TestUtils.setupDagger();
    LoginPresenter mockLoginPresenter = mock(LoginPresenter.class);
    Mockito.when(TestUtils.appModule.provideLoginPresenter(any(UserManager.class),
any(PasswordValidator.class))).thenReturn(mockLoginPresenter);
    LoginActivity loginActivity = Robolectric.setupActivity(LoginActivity.class);
    ((EditText) loginActivity.findViewById(R.id.username)).setText("xiaochuang");
    ((EditText) loginActivity.findViewById(R.id.password)).setText("xiaochuang is handsome");
    loginActivity.findViewById(R.id.login).performClick();
    verify(mockLoginPresenter).login("xiaochuang", "xiaochuang is handsome");
  }
}
```

当然，上面的代码还可以用很多种方法作进一步简化，比如把 `TestUtils.setupDagger();` 放到 `@Before` 里面，或者是自定义一个基础测试类，把 `TestUtils.setupDagger();` 放这个基础测试类的 `@Before` 里面，然后 `LoginActivityTest` 继承这个基础测试类就可以了，or even better，自定义一个JUnit Rule，在每个测试方法被调用之前自动调用 `TestUtils.setupDagger();`。只是这些与当前的主题无关，就不具体展开叙述了。后面会讲到DaggerMock的使用，这个东西可真的是神器啊！简直不要太神器！

###单元测试里面，不要滥用dagger2
这里再重复一下上一篇文章的话，单元测试的时候不要滥用dagger2，虽然现在我们的app是用dagger2架构起来的，所有的Dependency都是在Module里面生产，但并不代表我们在做单元测试的时候，这些Dependency也只能在Module里面生产。比如说，`LoginPresenter`：

```
public class LoginPresenter {
    private final UserManager mUserManager;
    private final PasswordValidator mPasswordValidator;
    @Inject
    public LoginPresenter(UserManager userManager, PasswordValidator passwordValidator) {
        this.mUserManager = userManager;
        this.mPasswordValidator = passwordValidator;
    }
    public void login(String username, String password) {
        if (username == null || username.length() == 0) return;
        if (mPasswordValidator.verifyPassword(password)) return;
        mUserManager.performLogin(username, password);
    }
}
```

我们要测的是，`LoginPresenter#login()` 调用了 `mUserManager.performLogin()` 。在这里，我们可以按照上面的思路，使用dagger2来mock `UserManager` ，做法是mock module的 `provideUserManager()` 方法，让它返回一个mock的 `UserManager` ，然后去verify这个mock `UserManager` 的 `performLogin()` 方法得到了调用，代码大致如下：

```
@RunWith(RobolectricGradleTestRunner.class)
@Config(constants = BuildConfig.class, sdk = 21)
public class LoginPresenterTest {
    @Test
    public void testLogin() throws Exception {
        TestUtils.setupDagger();
        UserManager mockUserManager = mock(UserManager.class);
        Mockito.when(TestUtils.appModule.provideUserManager(any(SharedPreferences.class),
any(UserApiService.class))).thenReturn(mockUserManager);
        LoginPresenter presenter = ComponentHolder.getAppComponent().loginPresenter();
        presenter.login("xiaochuang", "xiaochuang is handsome");
        verify(mockUserManager).performLogin("xiaochuang", "xiaochuang is handsome");
    }
}
```

这样虽然可以，而且也不难，但毕竟路绕的有点远，而且你可能要做额外的一些工作，比如在 `AppComponent` 里面加一个正式代码不一定会用的 `loginPresenter();` 方法，另外因为 `AppModule` 里面有安卓相关的代码，我们还必须使用Robolectric，导致测试跑起来慢了很多。其实我们完全可以不用dagger2，有更好的办法，那就是直接new `LoginPresenter` ，传入mock `UserManager` ：

```
public class LoginPresenterTest {
    @Test
    public void testLogin() {
        UserManager mockUserManager = mock(UserManager.class);
        LoginPresenter presenter = new LoginPresenter(mockUserManager, new
PasswordValidator()); //因为这里我们不verify PasswordValidator，所以不需要mock这个。
        presenter.login("xiaochuang", "xiaochuang is handsome");
        verify(mockUserManager).performLogin("xiaochuang", "xiaochuang is handsome");
    }
}
```

程序是不是简单多了，也容易理解多了？

那么现在问题来了，如果这样的话，单元测试的时候，哪些情况应该用dagger2，那些情况不用呢？答案是，能不用dagger2，就不用dagger2，不得已用dagger2，才用dagger2。当然，这是一句废话，前面我们已经明显感受到了，在单元测试里面用dagger2比不用dagger2要麻烦多了，能不用当然不用。那么问题就变成了，什么情况下必须用dagger2、而什么时候可以不用呢？答案是，如果被测类（比如说`LoginActivity`）的Dependency（`LoginPresenter`）是通过*field injection* inject进去的，那么再测这个类（`LoginActivity`）的时候，就必须用dagger2，不然很难优雅的把mock传进去。相反，如果被测类有Constructor（比如说`LoginPresenter`），Dependency是通过Constructor传进去的，那么就可以不使用dagger2，而是直接new对象出来测。这也是为什么我在前一篇文章里面强烈的推荐 Constructor Injection的原因。

小结

这篇文章介绍了dagger2的使用，以及在单元测试里面的应用。哦好像忘了介绍把dagger2加到项目里面的方法，其实很简单，把以下代码加入build.gradle：

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}
apply plugin: 'com.android.application'  //这个已经有了，这里只是想说明要把android-apt这个plugin放到这个的后面。
apply plugin: 'com.neenbedankt.android-apt'
dependencies {
    //other dependencies
    //Dagger2
    compile 'com.google.dagger:dagger:2.0.2'
    compile 'javax.annotation:jsr250-api:1.0'
    apt 'com.google.dagger:dagger-compiler:2.0.2'
}
```

应该说，DI是一种很好的模式，哪怕不做单元测试，DI也会让我们的app的架构变得干净很多，可读性、维护性和可拓展性强很多，只不过单元测试让DI的必要性变得更加显著和迫切而已。而dagger2的作用，或者说角色，在于它让我们写正式代码的时候使用DI变得易如反掌，程序及其简洁优雅可读性高。同时，它在某些情况下让原来很难测的代码变得用以测试。

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。

参考：https://www.youtube.com/watch?v=oK_XtfXPkqw