

# Android单排上王者系列之Dagger2使用解析



study\_zhxu (/u/b43595e0aecb) [+ 关注](#)

2016.08.12 23:44\* 字数 2695 阅读 1213 评论 3 喜欢 20

(/u/b43595e0aecb)

本篇文章已授权微信公众号 guolin\_blog（郭霖）独家发布

## 前言###

现在Dagger2在项目中的使用越来越多，Dagger2是Dagger的升级版本，Dagger我没有用过，但是本篇说的是Dagger2，主要讲解的是Dagger2是如何使用的。对了，忘了说Dagger其实是一个依赖注入的框架。

## 什么是依赖注入###

依赖注入是一种面向对象的编程模式，它的出现是为了降低耦合性，所谓耦合就是类之间依赖关系，所谓降低耦合就是降低类和类之间依赖关系。可能有的人说自己之前并没有使用过依赖注入，其实真的没有使用过吗？当我们在一个类的构造函数中通过参数引入另一个类的对象，或者通过set方法设置一个类的对象其实就是使用的依赖注入。

## 通常依赖注入有以下几种方式###

- 通过接口注入

```
interface ClassBInterface {
    void setB(ClassB b);
}

public class ClassA implements ClassBInterface {
    ClassB classB;
    @Override
    void setB(ClassB b) {
        classB = b;
    }
}
```

- 通过set方法注入

```
public class ClassA {
    ClassB classB;
    public void setClassB(ClassB b) {
        classB = b;
    }
}
```

- 通过构造方法注入



```
public class ClassA {
    ClassB classB;
    public void ClassA(ClassB b) {
        classB = b;
    }
}
```



- 通过注解的方式注入

```
public class ClassA {
    //此时并不会完成注入，还需要依赖注入框架的支持，如Dagger2
    @inject
    ClassB classB;
    public ClassA() {
    }
}
```

下面我们来说如何通过Dagger2来实现依赖注入吧。

## 引入Dagger2

### 添加apt插件

```
dependencies {
    classpath 'com.android.tools.build:gradle:2.1.2' //添加apt插件
    classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
}
```

### 添加依赖(在build.gradle中添加如下代码)

```
apply plugin: 'com.android.application' //添加如下代码，应用apt插件
apply plugin: 'com.neenbedankt.android-apt'
...
dependencies {
    ...
    compile 'com.google.dagger:dagger:2.4' apt 'com.google.dagger:dagger-compiler:2.4'
    //java注解
    compile 'org.glassfish:javax.annotation:10.0-b28'
    ...
}
```

## 使用Dagger2

添加完Dagger的依赖后我们如何在项目中使用Dagger呢？

在项目中绝大多数使用都是Dagger结合MVP架构使用的，在MVP中使用是非常典型的降低耦合的使用。不懂MVP的可以看这里 (<https://www.jianshu.com/p/608017b7f42b>)。本篇文章中的示例是一个简单的登陆功能的示例，代码沿用上一篇讲解MVP的登陆代码，看这里 (<https://www.jianshu.com/p/608017b7f42b>)，该示例采用MVP架构设计通过Dagger2进行解耦合，下面就来看看如何使用吧。

在使用Dagger2前我们最好简单的了解一下MVP，主要是为了理解本篇中的代码。简单了解MVP即使不会写MVP也可以看的懂本篇的代码。

为什么要选择在MVP模式中使用Dagger2呢？

因为在MVP模式中Activity持有presenter的引用，同时presenter也持有view的引用，这样便于更新UI界面，这样Activity就和presenter仅仅的耦合在一起了，而Dagger2是依赖注入框架就是解耦合的，所以子MVP中使用Dagger2也就再好不过了。

在上篇文章讲解MVP时我们可以明显的看到如下代码

```

public class LoginActivity extends AppCompatActivity implements ILoginView,View.OnClickListener{
    private Button mLogin ;
    private Button mClear ;
    private EditText mName ;
    private EditText mPassWord ;
    ILoginPresenter loginPresenter ;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mLogin = (Button) findViewById(R.id.btn_login);
        mClear = (Button) findViewById(R.id.btn_clear);
        mName = (EditText) findViewById(R.id.et_name);
        mPassWord = (EditText) findViewById(R.id.et_password);
        mLogin.setOnClickListener(this);
        mClear.setOnClickListener(this); //持有presenter的引用并且创建对象
        loginPresenter = new LoginPresenterCompl(this) ;
    }
    .....
}

```

在上述代码中可以看到activity持有了presenter的引用并且创建了该对象，但是如果presenter的构造函数发生改变则这里也需要改变，其实所有和presenter构造函数相关的代码都要改变。

但是如果我们使用Dagger2依赖框架该如何使用呢？

请看下面代码activity中的代码

```

public class LoginActivity extends AppCompatActivity implements ILoginView,View.OnClickListener{
    .....
    //注意此处使用了注解
    @Inject LoginPresenterCompl loginPresenter ;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mLogin = (Button) findViewById(R.id.btn_login);
        mClear = (Button) findViewById(R.id.btn_clear);
        mName = (EditText) findViewById(R.id.et_name);
        mPassWord = (EditText) findViewById(R.id.et_password);
        mLogin.setOnClickListener(this);
        mClear.setOnClickListener(this);
        DaggerMainComponent.builder().mainModule(new MainModule(this)).build().inject(this);
    }
    .....
}

```

LoginPresenterCompl中的代码

```

public class LoginPresenterCompl implements ILoginPresenter {
    private ILoginView loginView ;
    private User user ;
    //注意此处使用了注解
    @Inject public LoginPresenterCompl(ILoginView view){
        loginView = view ;
        user = new User("张三","123456") ;
    }
    .....
}

```

只有上述两个注解还无法完成依赖注入，还需要如下两个新增类新增的MainModule类

```

@Modulepublic
class MainModule {
    private final ILoginView view ;
    public MainModule(ILoginView view){
        this.view = view ;
    }
    @Provides
    ILoginView provideILogView(){
        return view ;
    }
}

```

## 新增的MainComponent接口

```

@Component(modules = MainModule.class)
public interface MainComponent {
    public void inject(LoginActivity activity) ;
}

```

通过直接注解和上述两个接口类即可完成Dagger2的依赖注入。在LoginActivity中是通过

```

DaggerMainComponent.builder().mainModule(new MainModule(this)).build().inject(this)

```

完成依赖注入的。看完上面的代码后，一脸的懵逼，WTF(what the fuck)，这TM是什么，这么复杂，还不如之前的简单呢，新增了两个类还有这么多代码，得不偿失呀！同志们，如果你们第一眼看到后是这样想的话，说明和我想的一样，呵呵。每一个刚接触Dagger2的人可能都会这样想，因为我们只看到了表面。

不错，表面上我们是多了一个类和接口也多了很多代码，但是这样的组合其实是可以理解的。因为通常简单的代码具有耦合性，而要想降低这样的耦合就需要其他的辅助代码，其实少代码量和低耦合这两者并不能同时兼顾，古人云：鱼和熊掌不可兼得。我们作为堂堂聪明绝顶的程序猿怎么可能会输给古人呢。

好！下面来认真讲解Dagger2是如何完成依赖注入的。

首先我们来看看LoginActivity代码LoginActivity中有这么一段代码

```

@Inject
LoginPresenterCompl loginPresenter ;

```

同样在LoginPresenterCompl中也有这么一段代码

```

@Inject
public LoginPresenterCompl(ILoginView view){
    loginView = view ;
    user = new User("张三","123456") ;
}

```

之所以挑出这两段代码是因为它们都添加了@Inject注解。

在LoginActivity中其实只有这么一句提到loginPresenter，在接下来的代码中并没有对其进行初始化。那loginPresenter是如何进行初始化的呢（此处注意添加@Inject注解的变量不能被private修饰）？

直观上我们可以这样理解，被@Inject注解的代码存在某种联系，当代码执行到@Inject的时候程序会自动进入到这个类的构造方法中，如果正巧这个构造方法也被@Inject修饰了，那么系统就会帮我们自动创建对象。

这只是表面的理解，这其中肯定还有很多我们没有看到的“猫腻”。这俩不会无缘无故的有联系，肯定还有第三者，通过这个第三者这两个被@Inject注解修饰的代码才会产生联系。

这个第三者是谁呢？

自然的我们会想到我们添加的这个类和接口。

首先我们来分析MainComponent接口代码如下

```

@Component(modules = MainModule.class)
public interface MainComponent {
    public void inject(LoginActivity activity) ;
}

```

MainComponent是一个接口（也可以是一个抽象类），在这个接口中我们定义了一个inject()方法，其中参数是LoginActivity对象，同时MainComponent还被@Component注解着，注解中modules的值是MainModule.class，这个内容会在接下来的地方进行说明，暂时先放一放。

此时在Android studio中，如果我们rebuild一下项目就会有新的发现。在项目的build/generated/source/apt/debug/项目包名/dagger目录下生成对应的包其中包含DaggerMainComponent类，这个类名其实不是固定的，是根据我们上面写的MainComponent，加了Dagger前缀生成的DaggerMainComponent。其实在这个时候我们就已经完成了present的依赖注入。但是在

```

DaggerMainComponent.builder().mainModule(new MainModule(this)).build().inject(this)

```

中我们看到还有一个MainModule，这个是我们自己创建的一个类MainModule代码如下

```

@Modulepublic
class MainModule {
    private final ILoginView view ;
    public MainModule(ILoginView view){
        this.view = view ;
    }
    @Provides
    ILoginView provideILogView(){
        return view ;
    }
}

```

我们可以看到这个类被@Module注解修饰，内部有一个ILoginView的变量和一个构造方法还有一个被@Provides修饰的provideILogView方法。

看到这还是一脸懵逼，这个类是干嘛的？

在MainComponent接口中我们看到这么一个注解@Component(modules = MainModule.class)，这里用到了MainModule，可见MainComponent需要MainModule一起才能完成工作。其实这个类我们可以理解成提供参数的，也就是提供参数依赖的，如何理解呢？

在MainModule中我们为什么要提供ILoginView类型的对象？为什么不是其他的呢？这是因为LoginPresenterCompl的构造函数需要这么一个参数，所以我们在这里提供这么一个相同的参数，并通过被@Provides注解修饰的方法将其返回出去，如果LoginPresenterCompl还需要其他的参数，同样我们也可以在这里添加对应类型的参数然后通过另一个被@Provides注解修饰的方法返回出去。在MainComponent接口中提供的inject()方法的参数是LoginActivity，这个参数的含义是LoginPresenter要在什么地方注入。

### 了解了各个类的功能后我们来总结一下

- @Inject 程序会将Dagger2会将带有此注解的变量或者构造方法参与到依赖注入当中，Dagger2会实例化这个对象- @Module 带有该注解的类需要对外提供依赖，其实就是提供实例化需要的参数，Dagger2在实例化的过程中发现一些参数，Dagger2就会到该类中寻找带有@Provides注解的以provide开头的需找对应的参数
- @Component 带有该注解的接口或抽象类起到一个关联桥梁的作用，作用就是将带有@Inject的方法或对象和带有@Module的类进行关联，只有通过该接口或抽象类才可以在实例化的时候到带有@Module类类中寻找需要的参数，也就是依赖注入。

OK，下面我们来捋捋思路。

- 1、在这个示例代码中，LoginActivity中需要LoginPresenterCompl，所以在LoginActivity中定义了该对象并且通过@Inject将其注解，同时到

LoginPresenterCompl的构造方法中也通过@Inject将其注解，表明这些是需要依赖注入的。

- 2、因为在LoginPresenterCompl的构造方法需要ILoginView类型的参数，所以需要通过依赖将获取这些参数，所以就需要带有@Module注解的类用于获取需要的参数，在@Module注解的类中通过被@Provides注解的以provide开头的方法对外提供需要的参数，一般而言有几个参数就需要有几个带有@Provides的方法。
- 3、此时还需要一个桥梁将两者联系到一起，带有@Component的接口或抽象类就起到这个桥梁的作用。注解中有一个module的值，这个值指向需要依赖的Module类，同时其中有一个抽象方法inject()，其中的参数就是我们需要在哪个类中实例化LoginPresenterCompl，因为我们需要在LoginActivity中实例化，所以参数类型就是LoginActivity类型。然后在Android studio中rebuild我们的项目，就会生成DaggerMainComponent类，通过

```
DaggerMainComponent.builder().mainModule(new MainModule(this)).build().inject(this);
```

完成我们需要的依赖注入。###总结可能我们通过上面的讲解，知道了如何使用Dagger2了，也知道具体的流程了，但是可能还会有些疑惑，为什么？Dagger2是如何通过一些接口和类就完成依赖注入的？在此声明，别着急，知道如何使用这只是第一步，在下一篇文章中将会讲解Dagger2实现依赖注入的原理。敬请期待！！

小礼物走一走，来简书关注我

赞赏支持

Android单排上王者系列 (/nb/3077635)

举报文章 © 著作权归作者所有



study\_zhxu (/u/b43595e0aecb)

写了 4733 字，被 40 人关注，获得了 58 个喜欢  
(/u/b43595e0aecb)

+ 关注

CSDN博客地址[http://blog.csdn.net/study\\_zhxu](http://blog.csdn.net/study_zhxu)

喜欢 20



更多分享

(<http://cwb.assets.jianshu.io/notes/images/5247711>)



下载简书 App ▶  
随时随地发现和创作内容



(/apps/download?utm\_source=nbc)

被以下专题收入，发现更多相似内容



Android知识 (/c/3fde3b545a35?utm\_source=desktop&utm\_medium=notes-included-collection)



Android开发 (/c/d1591c322c89?utm\_source=desktop&utm\_medium=notes-included-collection)

# Android单排上王者系列之Dagger2注入原理解析



study\_zhxu (/u/b43595e0aecb) [+ 关注](#)

2016.08.13 00:11\* 字数 1925 阅读 1339 评论 2 喜欢 14 赞赏 1

(/u/b43595e0aecb)

本篇文章已授权微信公众号 **guolin\_blog**（郭霖）独家发布

MVP模式讲解 (<https://www.jianshu.com/p/608017b7f42b>)

在MVP中使用Dagger2 (<https://www.jianshu.com/p/98e344cadd8b>)

Dagger2的注入原理解析 (<https://www.jianshu.com/p/4a4008ac68ad>)

在上篇博客中我们介绍了Dagger2该如何在项目中使用，这篇博客将继续分析Dagger2实现的原理，代码依然采用上篇的代码，看这里

(<https://www.jianshu.com/p/98e344cadd8b>)。

## Dagger2的注入原理

原理的讲解我们通过小明来带我们学习。

小明在看了MVP的实战解析和Dagger2的使用后知道了Dagger2该如何在MVP模式中使用，但是小明是一个要求上进的好同学，小明并不满足于如何使用，小明想钻研钻研源码，看看如何实现的。小明在钻研Dagger2的时候突然意识到Dagger2是采用注解的形式完成任务的，使用注解其实是不明智的选择，会大大消耗性能，影响应用的运行速度。小明看到这里有点疑惑了，既然注解这么影响性能，那为什么Dagger2还要使用注解呢？为什么Dagger2还这么被广泛的使用呢？

于是小明到github上查看Dagger2的介绍

**官方介绍是**

A fast dependency injector for Android and Java.

Dagger2是一个Android和Java中的快速注射器。

小明又疑惑了注解反射怎么是快速的呢？小明没有灰心又继续查看代码，终于发现Dagger2是和其他依赖注入框架是有区别的，Dagger2是通过apt插件在编译阶段生成注入代码的，也就是说反射只是在编译阶段使用了，而在应用运行的时候其实运行的是真正的Java代码并没有涉及到注解反射，小明终于明白了，难怪Dagger2是快速注入框架。

小明有了这个重大发现后决定一鼓作气把Dagger2生成的代码给理清楚。

## 编译阶段生成代码

小明通过在Android studio中通过执行Build->Rebuild Project，在app/build/generated/source/apt目录下发现生成了

## LoginPresenterComp\_Factory类

代码如下

```

public final class LoginPresenterCompl_Factory implements Factory<LoginPresenterCompl> {
    private final Provider<ILoginView> viewProvider;
    public LoginPresenterCompl_Factory(Provider<ILoginView> viewProvider) {
        assert viewProvider != null;
        this.viewProvider = viewProvider;
    }
    @Override
    public LoginPresenterCompl get() {
        return new LoginPresenterCompl(viewProvider.get());
    }
    public static Factory<LoginPresenterCompl> create(Provider<ILoginView> viewProvider) {
        return new LoginPresenterCompl_Factory(viewProvider);
    }
}

```

为了对比小明又把LoginPresenterCompl的代码也找了出来代码如下

```

public class LoginPresenterCompl implements ILoginPresenter {
    @Inject
    public LoginPresenterCompl(ILoginView view){
        loginView = view ;
        user = new User("张三","123456") ;
    }
    .....
}

```

仔细看看LoginPresenterCompl\_Factory这个类，发现其中有三个方法

- 构造方法

构造方法中的参数viewProvider一个Provider类型的，而Provider的泛型参数是ILoginView，这个参数就是我们实例化LoginPresenterCompl需要的参数，在上篇中我们知道了该参数是一个依赖，是由MainModule提供的。

- get()方法

在该方法中初始化了我们正在需要的LoginPresenterCompl对象

- create()方法

在该方法中实例化了LoginPresenterCompl\_Factory本类对象

小明想既然上面的viewProvider是由MainModule提供的，那么就来看看MainModule对应的注入类吧

## MainModule\_ProvideILogViewFactory代码如下

```

public final class MainModule_ProvideILogViewFactory implements Factory<ILoginView> {
    private final MainModule module;
    public MainModule_ProvideILogViewFactory(MainModule module) {
        assert module != null; this.module = module;
    }
    @Override
    public ILoginView get() {
        return Preconditions.checkNotNull( module.provideILogView(), "Cannot return null from a non-@Nullable @Provides method");
    }
    public static Factory<ILoginView> create(MainModule module) {
        return new MainModule_ProvideILogViewFactory(module);
    }
}

```

对应的MainModule代码如下



```

@Modulepublic
class MainModule {
    private final ILoginView view ;
    public MainModule(ILoginView view){
        this.view = view ;
    }
    @Provides
    ILoginView provideILogView(){
        return view ;
    }
}

```

从结构中不难看出被@Provider注解修饰的方法会对应的生成Factory类，这个类中最主要的方法是get()方法，在该方法中调用了MainModule的provideILogView方法，而该方法是为了我们提供LoginPresenterCompl实例化参数的，LoginPresenterCompl的实例化是在LoginPresenterCompl\_Factory的get()方法中完成的。实例化代码如下

```

@Override
public LoginPresenterCompl get() {
    return new LoginPresenterCompl(viewProvider.get());
}

```

在代码中可以看出实例化过程中参数是由viewProvider.get()提供的。噢！！！！  
 在MainModule\_ProvideILogViewFactory中的get()方法其实返回了我们实例化的参数。  
 那么这个viewProvider是不是我们的MainModule\_ProvideILogViewFactory呢？  
 viewProvider是一个Provider类型，而MainModule\_ProvideILogViewFactory实现了Factory接口，那Provider和Factory有没有联系呢？  
 看这段代码

```

public interface Factory<T> extends Provider<T> {
}

```

发现Factory接口继承了Provider接口，所以其实viewProvider就是MainModule\_ProvideILogViewFactory类型。看到这里小明终于明白了LoginPresenterCompl\_Factory类和MainModule\_ProvideILogViewFactory类的关系了，也明白了实例化过程了。但是这些类的初始化和相关方法是如何被调用的，在哪里被调用的呢？  
 还有两个重要的类小明没有看到。MainComponent和对应的DaggerMainComponent。  
 代码如下  
 MainComponent代码

```

@Component(modules = MainModule.class)
public interface MainComponent {
    public void inject(LoginActivity activity) ;
}

```

## DaggerMainComponent代码

```

public final class DaggerMainComponent implements MainComponent {
    private Provider<ILoginView> provideILogViewProvider;
    private Provider<LoginPresenterCompl> loginPresenterComplProvider;
    private MembersInjector<LoginActivity> loginActivityMembersInjector;
    private DaggerMainComponent(Builder builder) {
        assert builder != null;
        initialize(builder);
    }
    public static Builder builder() {
        return new Builder();
    }
    @SuppressWarnings("unchecked")
    private void initialize(final Builder builder) {
        this.provideILogViewProvider = MainModule_ProvideILogViewFactory.create(b
uilder.mainModule);
        this.loginPresenterComplProvider = LoginPresenterCompl_Factory.create(pr
ovideILogViewProvider);
        this.loginActivityMembersInjector = LoginActivity_MembersInjector.create(
loginPresenterComplProvider);
    }
    @Override
    public void inject(LoginActivity activity) {
        loginActivityMembersInjector.injectMembers(activity);
    }
    public static final class Builder {
        private MainModule mainModule;
        private Builder() {}
        public MainComponent build() {
            if (mainModule == null) {
                throw new IllegalStateException(MainModule.class.getCanon
icalName() + " must be set");
            }
            return new DaggerMainComponent(this);
        }
        public Builder mainModule(MainModule mainModule) {
            this.mainModule = Preconditions.checkNotNull(mainModule); return
this;
        }
    }
}

```

通过上面代码可以看出DaggerMainComponent实现了MainComponent接口并实现了其中的inject()方法。同时也提供了其他的辅助方法。小明决定从方法调用顺序开始入手查看

```
DaggerMainComponent.builder().mainModule(new MainModule(this)).build().inject(this);
```

## 注入过程

还记得在LoginActivity中添加的这个方法吗，分析DaggerMainComponent就从这段代码入手。

- 1、DaggerMainComponent调用了builder方法  
小明找到builder()方法看看这个方法到底做了什么事

```
public static Builder builder() { return new Builder();}
```

发现这个方法创建并返回了Builder对象Builder是什么东东呢？仔细看代码Build是DaggerMainCompone的内部类。

- 2、DaggerMainComponent.builder().mainModule(new MainModule(this))  
紧接着又调用了mainModule并将MainModule的对象传了进来。mainModule()方法是Builder中的一个方法，代码如下

```
public Builder mainModule(MainModule mainModule) { this.mainModule = Preconditions.checkNotNull(mainModule); return this;}
```

其中做了什么事呢？就是将传进来的MainModule对象赋值给本类的mainModule对象，并返回本类对象

- 3、DaggerMainComponent.builder().mainModule(new MainModule(this)).build()  
紧接着又调用了Builder的build()方法

```
public MainComponent build() { if (mainModule == null) {  
    throw new IllegalStateException(MainModule.class.getCanonicalName() + " must be set");  
}  
    return new DaggerMainComponent(this);  
}
```

该方法通过new DaggerMainComponent(this)创建了DaggerMainComponent对象并将其返回。那么new DaggerMainComponent(this)做了什么事呢？

```
private DaggerMainComponent(Builder builder) { assert builder != null; initialize(builder  
);}
```

其中调用了initialize方法并将builder对象传入。initialize()方法如下

```
private void initialize(final Builder builder) {  
    this.provideILogViewProvider = MainModule_ProvideILogViewFactory.create(builder.m  
ainModule);  
    this.loginPresenterComplProvider = LoginPresenterCompl_Factory.create(provideILog  
ViewProvider);  
    this.loginActivityMembersInjector = LoginActivity_MembersInjector.create(loginPre  
senterComplProvider);  
}
```

在initialize方法中小明终于看到了MainModule\_ProvideILogViewFactory和LoginPresenterCompl\_Factory的create方法被调用了。

首先通过传入的MainModule对象创建MainModule\_ProvideILogViewFactory对象provideILogViewProvider，然后将provideILogViewProvider对象作为参数来创建LoginPresenterCompl\_Factory对象。

前面已经讲过，MainModule\_ProvideILogViewFactory是一个Factory对象，而LoginPresenterCompl\_Factory创建对象需要一个Provider对象，同时Factory继承了Provider，所以可以将其传入。所以LoginPresenterComp\_Factory的viewProvider对象是一个MainModule\_ProvideILogViewFactory对象，这个概念前面也讲过，这里得到认证。

在这段代码中小明发现了LoginActivity\_MembersInjector类，于是小明又将这个类找了出来代码如下

```
public final class LoginActivity_MembersInjector implements MembersInjector<LoginActivity  
> {  
    private final Provider<LoginPresenterCompl> loginPresenterProvider;  
    public LoginActivity_MembersInjector(Provider<LoginPresenterCompl> loginPresenter  
Provider) {  
        assert loginPresenterProvider != null;  
        this.loginPresenterProvider = loginPresenterProvider;  
    }  
    public static MembersInjector<LoginActivity> create( Provider<LoginPresenterCompl  
> loginPresenterProvider) {  
        return new LoginActivity_MembersInjector(loginPresenterProvider);  
    }  
    @Override  
    public void injectMembers(LoginActivity instance) {  
        if (instance == null) {  
            throw new NullPointerException("Cannot inject members into a null referen  
ce");  
        }  
        instance.loginPresenter = loginPresenterProvider.get();  
    }  
    public static void injectLoginPresenter( LoginActivity instance, Provider<LoginPr  
esenterCompl> loginPresenterProvider) {  
        instance.loginPresenter = loginPresenterProvider.get();  
    }  
}
```

该类的create()方法需要一个Provider泛型是LoginPresenterCompl类型的参数，通过构造函数将其传入赋值给loginPresenterProvider变量。就这么简单。

- 4、DaggerMainComponent.builder().mainModule(new MainModule(this)).build().inject(this)  
最后调用了inject()方法

```
@Override
public void inject(LoginActivity activity) {
    loginActivityMembersInjector.injectMembers(activity);
}
```

在该方法中调用了LoginActivityMembersInjector中的injectMembers()方法。  
injectMembers()方法内容如下

```
@Override
public void injectMembers(LoginActivity instance) {
    if (instance == null) {
        throw new NullPointerException("Cannot inject members into a null reference");
    }
    instance.loginPresenter = loginPresenterProvider.get();
}
```

终于!!!!!! 在这个方法中实现了对LoginPresenterCompl对象的初始化。  
至此，小明终于弄清楚了Dagger2的注入原理了，小明表示清楚原理后妈妈再也不用担心Dagger2写错了。  
如果没有正确的分析这个生成的注入类可能很难理解Dagger2实现注入的框架，可能看原理代码让有些同学不知所措，相信我，多分析几遍就OK了。  
其实也不用纠结到底该如何使用Dagger2，只要我们理解了其实现的原理，具体如何使用看个人，能够做到灵活使用就OK了。  
至此Dagger2的原理分析就完成了。

## 总结

回顾一下该系列的文章

MVP模式讲解 (<https://www.jianshu.com/p/608017b7f42b>)

在MVP中使用Dagger2 (<https://www.jianshu.com/p/98e344cadd8b>)

Dagger2的注入原理解析 (<https://www.jianshu.com/p/4a4008ac68ad>)因为这三篇是连续的，代码都是在前一篇的基础上做的扩展，所以最好将三篇博客通读。希望这三篇文章能够帮到需要的同学，共同进步!!!

最后全部代码点击[这里](https://link.jianshu.com?t=https://github.com/studyzhxu/Dagger2MVP) ([https://link.jianshu.com?](https://link.jianshu.com?t=https://github.com/studyzhxu/Dagger2MVP)

[t=https://github.com/studyzhxu/Dagger2MVP](https://github.com/studyzhxu/Dagger2MVP))

小礼物走一走，来简书关注我

赞赏支持



(/u/98f411525c9b)

Android单排上王者系列 (/nb/3077635)

举报文章 © 著作权归作者所有



study\_zhxu (/u/b43595e0aecb)

写了 4733 字，被 40 人关注，获得了 58 个喜欢

(/u/b43595e0aecb)

+ 关注

CSDN博客地址[http://blog.csdn.net/study\\_zhxu](http://blog.csdn.net/study_zhxu)

喜欢 14