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> viewProvid
er) {
                 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 retu
rn 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。咦！！！！
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_ProvideILoginViewFactory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.checkN
otNull(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cereate()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.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)

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

喜欢　14