在日常的开发中，通常我们在创建对象的时候，通过new关键字创建，这种创建方式并不优雅，通常来说，端上要拿到一个对象之后，是直接拿到用的，而不是说我需要自给自销，针对这种状况，依赖注入技术应运而生。

在Android中常用的注入框架有两种：Dagger2和Hilt，Hilt是JetPack组件中提供的依赖注入组件，是在Dagger2组件的基础上扩展，因此本章主要介绍Dagger2的使用及原理。

Dagger2的使用

如果要使用Dagger2，首先需要导入依赖

implementation 'com.google.dagger:dagger:2.6'

kapt 'com.google.dagger:dagger-compiler:2.6' //如果使用kotlin就需要使用kapt

看到annotationProcessor，如果之前看到编译时技术的文章就能知道，Dagger2一定是在编译期生成了一些类，运行的时候就可以直接调用。

buildscript {

dependencies {

classpath 'com.android.tools.build:gradle:3.3.2'

classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'

}

}

从Dagger2注解了解依赖注入思想

因为我们在使用某个类对象时，不需要自己再手动去创建，而是直接拿到用，那么就需要一个对象提供者。

@Module

public class BusinessModule {

@Provides

public static BusinessMode providerBusinessMode() {

return new BusinessMode();

}

}

例如BusinessModule是用于存储业务层需要的类对象，就可以使用@Module注解注释；  
如果我们需要一个BusinessMode对象，那么提供providerBusinessMode方法，采用@Provides注解注释，通过providerBusinessMode就可以得到我们想要的这个类

在创建了Module之后，需要Component组件将这个依赖注入到某个Activity或者Fragment中

@Component(modules = {BusinessModule.class})

public interface BusinessComponent {

void inject(MainActivity mainActivity);

}

注意这里不能使用多态，需要明确到底要注入到哪一个Activity或者Fragment中。

@Inject

var businessMode:BusinessMode? = null

这样，通过Inject注解修饰我们想要的类，通过BusinessComponent具体实现类DaggerBusinessComponent实现注入

DaggerBusinessComponent.create().inject(this)

Dagger2使用中遇到的错误总结

****错误1：****

Dagger does not support injection into private fields

这个错误是Dagger不支持通过@Inject注解生成私有成员变量，我们通过编译后的代码可以看到，之前声明的businessMode对象，是私有的

@javax.inject.Inject()

private com.lay.image\_process.dagger2.module.BusinessMode businessMode;

所以通过@JvmField注解修饰便可以解决这个问题

@Inject

@JvmField

var businessMode:BusinessMode? = null

****错误2：****

dagger.Provides missing element type

出现这个错误的原因就是dagger的版本和注解处理器的版本不一致导致的，即便是dagger有更新，只要compiler没有更新就不行，需要保持一致。

错误3：

java.lang.NoClassDefFoundError: javax/annotation/Generated

这是在编译时报错，需要我们添加javax annotation的依赖

implementation 'javax.annotation:javax.annotation-api:1.3.2'

kapt "javax.annotation:javax.annotation-api:1.3.2"

Dagger2的原理分析

Component组件实现注入原理

DaggerBusinessComponent.create().inject(this)

首先我们先看下create方法做了什么事

public static Builder builder() {

return new Builder();

}

public static BusinessComponent create() {

return builder().build();

}

create方法的主要作用就是创建了Builder对象，并调用了build方法，看下Builder类。

public static final class Builder {

private Builder() {}

public BusinessComponent build() {

return new DaggerBusinessComponent(this);

}

/\*\*

\* @deprecated This module is declared, but an instance is not used in the component. This method is a no-op. For more, see https://google.github.io/dagger/unused-modules.

\*/

@Deprecated

public Builder businessModule(BusinessModule businessModule) {

Preconditions.checkNotNull(businessModule);

return this;

}

}

我们可以看到build方法就是将DaggerBusinessComponent创建出来，并将Builder作为参数传递进去，看下DaggerBusinessComponent的构造方法

private DaggerBusinessComponent(Builder builder) {

assert builder != null;

initialize(builder);

}

在DaggerBusinessComponent的构造方法中，调用了initialize初始化方法

private void initialize(final Builder builder) {

this.mainActivityMembersInjector =

MainActivity\_MembersInjector.create(BusinessModule\_ProviderBusinessModeFactory.create());

}

在这个方法中，主要就是将MainActivity\_MembersInjector类初始化，调用了这个类的create方法

public static MembersInjector<MainActivity> create(Provider<BusinessMode> businessModeProvider) {

return new MainActivity\_MembersInjector(businessModeProvider);

}

在MainActivity\_MembersInjector的create方法中，同样也是new出来一个对象，businessModeProvider就是我们需要注入的某个类对象的提供者，对应@Provides注解这个方法。

所以整个create方法做了2件事：  
（1）创建具体的DaggerBusinessComponent对象；  
（2）创建具体的注入类对象，并初始化Module提供者

然后调用了inject方法

@Override

public void inject(MainActivity mainActivity) {

mainActivityMembersInjector.injectMembers(mainActivity);

}

在这个方法中，调用了mainActivityMembersInjector的injectMembers方法

@Override

public void injectMembers(MainActivity instance) {

if (instance == null) {

throw new NullPointerException("Cannot inject members into a null reference");

}

instance.businessMode = businessModeProvider.get();

}

在这个方法中，就是将MainActivity中的businessMode变量（这里就是为啥不能声明为私有变量的原因，没法调用）赋值，businessModeProvider其实就是数据提供者，调用的get方法就是调用BusinessModule中的providerBusinessMode方法

@Override

public BusinessMode get() {

return Preconditions.checkNotNull(

BusinessModule.providerBusinessMode(),

"Cannot return null from a non-@Nullable @Provides method");

}

在这个方法中，就是将MainActivity中的businessMode变量（这里就是为啥不能声明为私有变量的原因，没法调用）赋值，businessModeProvider其实就是数据提供者，调用的get方法就是调用BusinessModule中的providerBusinessMode方法

@Override

public BusinessMode get() {

return Preconditions.checkNotNull(

BusinessModule.providerBusinessMode(),

"Cannot return null from a non-@Nullable @Provides method");

}

也就是说

instance.businessMode = businessModeProvider.get();

等价于

instance.businessMode = new BusinessMode();

****所以我们可以这样理解：@Component是@Inject和@Module建立沟通的桥梁****

多Component注入

在之前，我们是将BusinessComponent注入到了MainActivity中，假设我们再写一个组件，也注入MainActivity中，这样可以吗？

@Module

class CarModule {

@Provides

fun provideCar(): Car {

return Car()

}

}

@Component(modules = [CarModule::class])

interface CarComponent {

fun inject(activity: MainActivity)

}

错误: com.lay.image\_process.dagger2.module.BusinessMode cannot be provided without an @Inject constructor or from an @Provides- or @Produces-annotated method.

public abstract void [inject(@org.jetbrains.annotations.NotNull()](mailto:inject(@org.jetbrains.annotations.NotNull())

我们编译工程之后，发现已经报错了，也就是说，****同一个Activity或者Fragment只能注入一个Component****

那如果我们想要注入多个Component该怎么做呢？

@Component(modules = [CarModule::class])

interface CarComponent {

fun getCar():Car

}

我们看到，在Component注解中，除了modules之外，还可以配置dependencies

Class<?>[] modules() default {};

/\*\*

\* A list of types that are to be used as <a href="#component-dependencies">component

\* dependencies</a>.

\*/

Class<?>[] dependencies() default {};

通过dependencies将CarComponent引进来，就相当于将两个组件组合起来一起给MainActivity使用

@Component(modules = {BusinessModule.class}, dependencies = {CarComponent.class})

public interface BusinessComponent {

void inject(MainActivity mainActivity);

}

那么在使用的时候就不能直接使用create创建DaggerBusinessComponent，这个时候会多出一个carComponent接口

DaggerBusinessComponent.builder()

.carComponent(DaggerCarComponent.create())

.build().inject(this)

还有一种方式，@SubComponent注解，但是这种方式不是很灵活

类之间相互依赖

@Module

class CarModule {

@Provides

fun provideCar(): Car {

return Car()

}

@Provides

fun providerCarFactory(car: Car): CarFactory {

return CarFactory(car)

}

}

像CarFactory类中需要传入Car作为参数，那么只需要同样声明一个提供Car对象的方法，这样就能直接注入到CarFactory的构造方法中。

@Override

public CarFactory get() {

return Preconditions.checkNotNull(

module.providerCarFactory(carProvider.get()),

"Cannot return null from a non-@Nullable @Provides method");

}

通过源码我们可以看到就是，通过carProvider获取到car实例，然后传入了providerCarFactory方法中