其实我们在使用Dagger2的时候，发现还是比较繁琐的，要自己写Module、Component、Provides等等，于是Hilt的团队就和Dagger2的团队一起，设计了面向Android移动端的依赖注入框架 -- Hilt

Hilt配置

在项目级的build.gradle中，引入支持hilt的插件，注意官方文档中的2.28-alpha版本可能有文件，建议使用下面的版本

classpath "com.google.dagger:hilt-android-gradle-plugin:2.43.2"

app的build.gradle中引入插件

id 'dagger.hilt.android.plugin'

引入依赖

implementation "com.google.dagger:hilt-android:2.43.2"

kapt "com.google.dagger:hilt-android-compiler:2.43.2"

Hilt的使用

首先按照惯例，先写一个Module

@Module

class RecordModule {

@Provides

fun providerRecord():Record{

return Record()

}

}

如果是Dagger2的写法，需要再写一个Component，将RecordModule加载进去，那么Hilt就不要这一步，而是需要一个注解InstallIn来声明这个Module使用在哪个地方

@InstallIn(ApplicationComponent::class)

@Module

class RecordModule {

@Provides

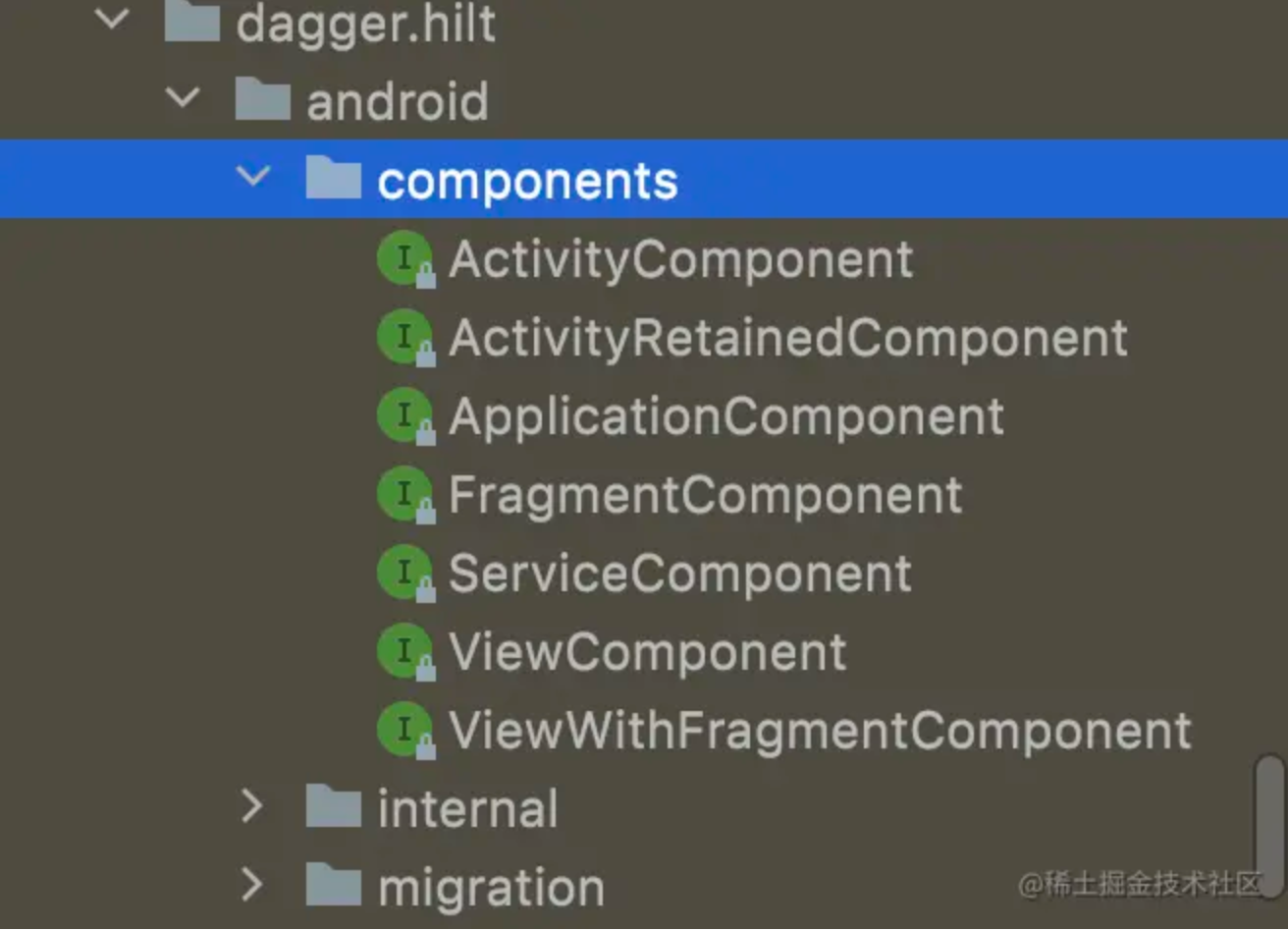
fun providerRecord(): Record {

return Record()

}

}

在Hilt中有以下几个Component，我这里拿几个典型说一下



首先ApplicationComponent，它会存在整个App生命周期中，随着App的销毁而销毁，也就意味着，在App的任何位置都可以使用这个Module

//A Hilt component that has the lifetime of the application

@Singleton

@DefineComponent

public interface ApplicationComponent {}

像ActivityComponent，肯定就是存在于整个Activity生命周期中，随着Activity的销毁而销毁

//A Hilt component that has the lifetime of the activity.

@ActivityScoped

@DefineComponent(parent = ActivityRetainedComponent.class)

public interface ActivityComponent {}

其他的都类似，都是随着组件的生命周期结束而消逝。

例如我们需要在MainActivity中注入某个类，那么需要使用@AndroidEntryPoint修饰，代表当前依赖注入的切入点

@AndroidEntryPoint

class MainActivity : AppCompatActivity()

同时，需要将当前app定义为Hilt App

@HiltAndroidApp

class MyApp : Application() {

override fun onCreate() {

super.onCreate()

}

}

而且，在MainActivity中注入这个对象之后，也不需要像Dagger那样，去创建具体的Component对象

@Inject

@JvmField

var record:Record? = null

局部单例

@InstallIn(ActivityComponent::class)

@Module

class RecordModule {

@Provides

@ActivityScoped

fun providerRecord(): Record {

return Record()

}

}

如果我们希望Record是一个单例对象，可以使用@ActivityScoped注解修饰，我们先看下效果

@Inject

@JvmField

var record: Record? = null

@Inject

@JvmField

var record2: Record? = null

在MainActivity中声明了两个对象，我们发现两个对象的hashcode是一致的，说明在当前Activity中这个对象就是单例，但是跳转到下一个Activity的时候，发现拿到的对象就是一个新的对象

2022-09-11 20:52:19.583 1860-1860/com.lay.image\_process E/TAG: record 83544912record2 83544912

2022-09-11 20:53:11.071 1860-1860/com.lay.image\_process E/TAG: record 163680212

也就是说，@ActivityScoped修饰的对象只是局部单例，并不是全局的；那么如何才能拿到一个全局的单例呢？其实在之前的版本中，有一个ApplicationComponent，其对应的作用域@Singleton拿到的对象就是全局单例，后来Google给移除了，我觉得Google之所以移除，可能就是推动大家采用数据共享设计模式。



为接口注入实现类

首先创建一个接口

interface MyCallback {

fun execute()

}

然后创建一个实现类，这里需要****注意，构造方法中如果需要传入上下文，那么需要使用@ApplicationContext修饰****，其他参数则不需要

class MyCallBackImpl : MyCallback {

private var context: Context? = null

@Inject

constructor(@ApplicationContext context: Context) {

this.context = context

}

override fun execute() {

Toast.makeText(context, "实现了", Toast.LENGTH\_SHORT).show()

}

fun clear() {

if (context != null) {

context = null

}

}

}

那么在创建module的时候，方法需要定义为抽象方法，而且需要使用@Binds来获取实现类，方法同样是抽象方法，参数为具体实现类，返回值为接口。

@Module

@InstallIn(ActivityComponent::class)

abstract class ImplModule {

@Binds

abstract fun getImpl(impl: MyCallBackImpl): MyCallback

}

那么在使用时，就非常简单了，这里获取到的就是MyCallBackImpl实现类

@Inject

@JvmField

var callback: MyCallback? = null

那么大家想一个问题，如果我有多个实现类，那么如何区分这个MyCallback到底是哪个实现类呢？同样可以使用注解来区分

class MyCallbackImpl2 : MyCallback {

private var context: Context? = null

@Inject

constructor(@ApplicationContext context: Context) {

this.context = context

}

override fun execute() {

Toast.makeText(context, "实现2", Toast.LENGTH\_SHORT).show()

}

}

这样的话，就有两个抽象方法，分别返回MyCallbackImpl2和MyCallBackImpl两个实现类，那么在区分的时候，就可以通过@BindImpl和@BindImpl2两个注解区分

@Module

@InstallIn(ActivityComponent::class)

abstract class ImplModule {

@Binds

@BindImpl

abstract fun getImpl(impl: MyCallBackImpl): MyCallback

@Binds

@BindImpl2

abstract fun getImpl2(impl2: MyCallbackImpl2): MyCallback

}

在调用时，同样需要使用@BindImpl2或者@BindImpl来区分获取哪个实现类

@Inject

@JvmField

@BindImpl2

var callback: MyCallback? = null

其实@BindImpl注解很简单，就是通过@Qualifier注解来区分

@Qualifier

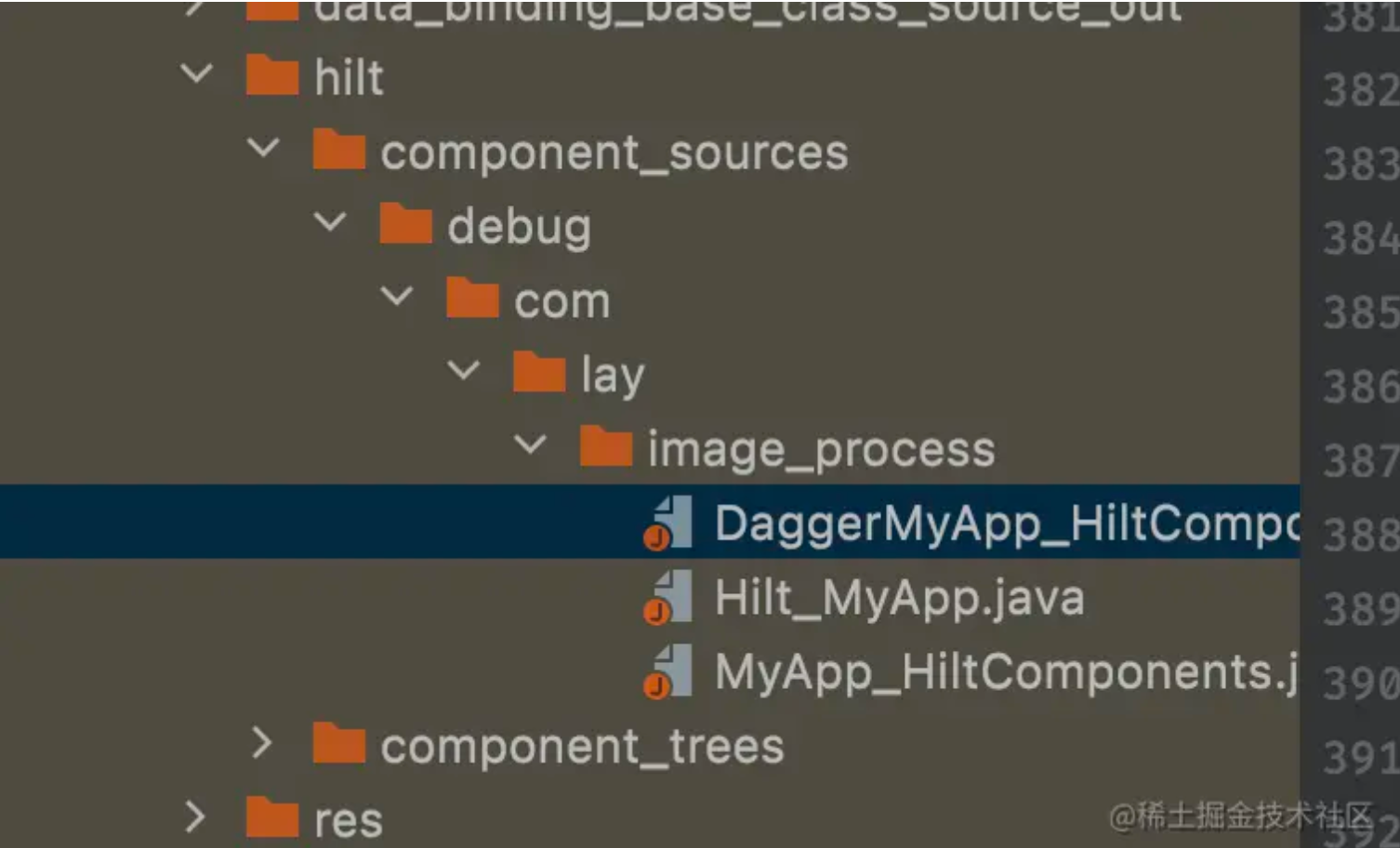
@Retention(AnnotationRetention.RUNTIME)

annotation class BindImpl2 {

}

Hilt原理

大家看到在调用这个注入对象的时候，发现没有看到对应的入口，并没有DaggerComponent那么显眼，其实编译时技术很多都是这样，是要从打包编译文件夹去找



我们可以看到，hilt是自己单独的一个文件夹，其中就有生成的资源文件

private MyCallbackImpl2 myCallbackImpl2() {

return new MyCallbackImpl2(ApplicationContextModule\_ProvideContextFactory.provideContext(singletonCImpl.applicationContextModule));

}

private MainActivity injectMainActivity3(MainActivity instance) {

MainActivity\_MembersInjector.injectRecord(instance, providerRecordProvider.get());

MainActivity\_MembersInjector.injectRecord2(instance, providerRecordProvider.get());

MainActivity\_MembersInjector.injectCallback(instance, myCallbackImpl2());

return instance;

}

其实我们可以看到，这种实现方式跟Dagger2其实是一样的，同样都是在内部初始化了某个类，例如MyCallbackImpl2，其context是由ApplicationContextModule提供的

@InjectedFieldSignature("com.lay.image\_process.MainActivity.callback")

@BindImpl2

public static void injectCallback(MainActivity instance, MyCallback callback) {

instance.callback = callback;

}

调用injectCallback就是将MainActivity中的callback赋值，获取的就是MyCallbackImpl2实现类。