**Dagger2 入门**

**依赖注入**

Dagger2是Android中比较热门的依赖注入框架,什么是依赖注入呢?维基百科上是这样描述的:

控制反转（Inversion of Control，缩写为IoC），是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。其中最常见的方式叫做依赖注入（Dependency Injection，简称DI），还有一种方式叫“依赖查找”（Dependency Lookup）。通过控制反转，对象在被创建的时候，由一个调控系统内所有对象的外界实体，将其所依赖的对象的引用传递给它。也可以说，依赖被注入到对象中.

通俗的来讲呢,就是一个类中需要依赖其他对象时,不需要你亲自为那些需要依赖的对象赋值,为那些对象赋值的操作交给了IOC框架.

**Dagger2介绍**

一般的IOC框架都是通过反射来实现的,但Dagger2作为Android端的IOC框架,为了不影响性能,它是通过apt动态生成代码来实现的.  
Dagger2主要分为三个模块:

1. 依赖提供方Module,负责提供依赖中所需要的对象,实际编码中类似于工厂类
2. 依赖需求方实例,它声明依赖对象,它在实际编码中对应业务类,例如Activity,当你在Activity中需要某个对象时,你只要在其中声明就行,声明的方法在下面会讲到.
3. 依赖注入组件Component,负责将对象注入到依赖需求方,它在实际编码中是一个接口,编译时Dagger2会自动为它生成一个实现类.

Dagger2的主要工作流程分为以下几步:

1. 将依赖需求方实例传入给Component实现类
2. Component实现类根据依赖需求方实例中依赖声明,来确定该实例需要依赖哪些对象
3. 确定依赖对象后,Component会在与自己关联的Module类中查找有没有提供这些依赖对象的方法,有的话就将Module类中提供的对象设置到依赖需求方实例中

通俗上来讲就好比你现在需要一件衣服,自己做太麻烦了,你就去商店买,你跟商店老板说明你想要购买的类型后,商店老板就会在自己的衣服供应商中查找有没有你所说的类型,有就将它卖给你.其中你就对应上面所说的依赖需求方实例,你只要说明你需要什么,商店老板则对应Component实现类,负责满足别人的需求,而衣服供应商则对应Module类,他负责生产衣服.也许这里有点绕,但经过下面的Demo,也许能够帮助你理解.

**书写Demo**

**引入Dagger2**

在项目下的build.gradle文件中添加apt插件:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | buildscript {  ...  dependencies {  classpath 'com.android.tools.build:gradle:2.2.2'  // NOTE: Do not place your application dependencies here; they belong  // in the individual module build.gradle files  //添加apt插件  classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'  }  }  ... |

在app目录的build.gradle文件中添加:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | //应用apt插件  apply plugin: 'com.neenbedankt.android-apt'  ...  dependencies {  ...  //引入dagger2  compile 'com.google.dagger:dagger:2.4'  apt 'com.google.dagger:dagger-compiler:2.4'  //java注解  provided 'org.glassfish:javax.annotation:10.0-b28'  } |

**编写布料类Cloth**

写一个Cloth类用作依赖对象,它包含一个color属性

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class Cloth {  private String color;  public String getColor() {  return color;  }  public void setColor(String color) {  this.color = color;  }  @Override  public String toString() {  return color + "布料";  }  } |

**书写Module类**

现在的需求是MainActivity中需要使用到Cloth对象,所以我们要为MainActivity书写一个Module类用来提供Cloth对象,相当于创建了一个提供商

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Module  public class MainModule {  @Provides  public Cloth getCloth() {  Cloth cloth = new Cloth();  cloth.setColor("红色");  return cloth;  }  } |

嗯?怎么多了两个注解?这两个注解有什么用呢?  
注解是Dagger2中的关键,编写Module类时要在该类上声明@Module以表明该类是Module类,这样Dagger2才能识别,那@Provides又是干嘛的呢?它的作用是声明Module类中哪些方法是用来提供依赖对象的,当Component类需要依赖对象时,他就会根据返回值的类型来在有@Provides注解的方法中选择调用哪个方法.在一个方法上声明@Provides注解,就相当于创建了一条生产线,这条生产线的产物就是方法的返回值类型.有了这条生产线,供应商就能提供这种类型的商品了,当商店老板发现有人需要这种类型的商品时,供应商就可以提供给他了

**书写Component接口**

|  |  |
| --- | --- |
| 1  2  3  4 | @Component(modules=MainModule.class)  public interface MainComponent {  void inject(MainActivity mainActivity);  } |

和Module类一样,Component类也是需要注解声明的,那个注解就是@Component,但是@Component注解的作用可不是单单用来声明Component类,他还有更强大的功能,@Component注解有modules和dependencies两个属性,这两个属性的类型都是Class数组,modules的作用就是声明该Component含有哪几个Module,当Component需要某个依赖对象时,就会通过这些Module类中对应的方法获取依赖对象,MainComponent中只包含MainModule,所以令modules=MainModule.class,相当于供应商和商店老板确定合作关系的合同.而dependencies属性则是声明Component类的依赖关系,这个下面再详讲.  
接口中那个方法又是干嘛用的呢?  
我们现在只是声明了Component类,但我们要怎么将Component类和依赖需求方对象联合起来呢?答案就是通过这个inject方法,这个方法可以将依赖需求方对象送到Component类中,Component类就会根据依赖需求方对象中声明的依赖关系来注入依赖需求方对象中所需要的对象,本Demo中MainActivity中需要Cloth对象,所以我们通过inject方法将MainActivity实例传入到MainComponent中,MainComponent就会从MainModule中的getCloth方法获取Cloth实例,并将该实例赋值给MainActivity中的cloth字段.相当于你去商店的道路,没有这条路,你就无法去商店和老板说明你所需要的东西.**但是这里需要注意的是,inject方法的参数不能用子类来接收,例如本Demo中,如果inject的参数是Activity,那么Dagger2就会报错.**

**在MainActivity中声明**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class MainActivity extends AppCompatActivity {  private TextView tv;  @Inject  Cloth cloth;  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  tv = (TextView) findViewById(R.id.tv);  MainComponent build = DaggerMainComponent.builder().mainModule(new MainModule()).build();  build.inject(this);  tv.setText("我现在有" + cloth);  }  } |

上面代码中有两处关键:

1. 声明依赖对象Cloth,就是在cloth字段上添加@Inject注解,Dagger2中声明依赖对象都是通过@Inject注解,**但是@Inject注解的字段不能是private和protected的.**
2. 通过Dagger2自动生成的类来创建Component的实现类,创建时需要传入该Component实现类所需要的Module类实例,传入方法就是调用Module类类名首字母小写对应的方法.这里我们通过Dagger2自动生成的DaggerMainComponent类创建了MainComponent的实例,相当于我们创建了一个实实在在的商店,不再是理论上的商店,但是创建商店一定也要创建真实的供应商嘛,所以创建Component实现类时一定要传入Module的实例.(注意编写完Component接口后Dagger2并不会自动创建对应的类,需要我们点击Android Studio中bulid菜单下的Rebulid Poject选项,或者直接书写代码,编译时Dagger2就会帮你自动生成).  
   再将MainActivity通过inject方法发送到MainComponent中,调用完inject方法后,你就会发现,MainActivity中的cloth字段已经被赋值,而且该cloth对应的就是我们在MainModule类getCloth方法中创建的Cloth对象.

**结果**

[](http://blog.izouxiang.cn/imgs/dagger2demo_1.jpeg)

**另一种方法**

前面的Demo可能给人最大的感受就是麻烦吧?就是为cloth赋个值,又要写什么Module类,又是要写什么Component接口.其实Dagger2还可以用注解来提供依赖对象.让我们来瞧瞧怎么使用.

**创建依赖类Shoe**

我们又创建一个依赖类Shoe

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class Shoe {  @Inject  public Shoe() {  }  @Override  public String toString() {  return "鞋子";  }  } |

但是这次我们创建的方式和Cloth不一样了,我们在构造函数上声明了@Inject注解,这个注解有什么用呢?作用可大了,当Component在所拥有的Module类中找不到依赖需求方需要类型的提供方法时,Dagger2就会检查该需要类型的有没有用@Inject声明的构造方法,有则用该构造方法创建一个.  
相当于你去商店购买东西,你需要的东西商店的供应商不生产,商店老板就只好帮你去网上看看有没有你需要的东西,有则帮你网购一个.(假设你不会网购,哈哈^ ^).

**在MainActivity中声明Shoe依赖**

我们修改之前的MainActivity,添加一点东西

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class MainActivity extends AppCompatActivity {  ...  @Inject  Shoe shoe;  @Override  protected void onCreate(Bundle savedInstanceState) {  ...  tv.setText("我现在有" + cloth + "和" + shoe);  }  } |

**结果**

[](http://blog.izouxiang.cn/imgs/dagger2demo_2.jpeg)

**注意**

有些读者可能会这样想:为什么不都用这种方法来声明呢?为什么要用Module类?  
答案是这样的,项目中我们会用到别人的jar包,我们无法修改别人的源码,就更别说在人家的类上添加注解了,所以我们只能通过Module类来提供.

**复杂一点的情况**

我们创建的这些依赖类都不用依赖于其它依赖类,但是如果需要依赖于其它依赖类又要怎么弄呢?

**创建依赖类Clothes**

我们又来创建一个衣服类Clothes,制作衣服时需要布料,所以我们在创建Clothes的实例时需要用到Cloth实例

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Clothes {  private Cloth cloth;  public Clothes(Cloth cloth) {  this.cloth = cloth;  }  public Cloth getCloth() {  return cloth;  }  @Override  public String toString() {  return cloth.getColor() + "衣服";  }  } |

**在Module类中增加提供方法**

现在我们的MainActivity中需要依赖于Clothes对象,所以我们在MianModule中添加提供Clothes对象的方法,但是Clothes需要依赖于Cloth对象,这要怎么办呢?可能最先想到的办法就是这样:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Provides  public Clothes getClothes(){  Cloth cloth = new Cloth();  cloth.setColor("红色");  return new Clothes(cloth);  } |

直接在方法中创建一个Cloth不就得了,但是你有没有发现,创建Cloth的代码已经在getCloth方法中有了,我们能不能用getCloth方法中创建的Cloth实例来创建Clothes实例呢?  
Dagger2提供了这样的功能,我们只要在getClothes方法中添加Cloth参数,Dagger2就会像帮依赖需求方找依赖对象一样帮你找到该方法依赖的Cloth实例,所以我们代码可以这样改:

|  |  |
| --- | --- |
| 1  2  3  4 | @Provides  public Clothes getClothes(Cloth cloth){  return new Clothes(cloth);  } |

**在MainActivity中声明Clothes依赖**

我们修改之前的MainActivity,添加一点东西

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class MainActivity extends AppCompatActivity {  ...  @Inject  Clothes clothes;  @Override  protected void onCreate(Bundle savedInstanceState) {  ...  tv.setText("我现在有" + cloth + "和" + shoe + "和" + clothes);  }  } |

**结果**

[](http://blog.izouxiang.cn/imgs/dagger2demo_3.jpeg)

**依赖总结**

同理,在带有@Inject注解的构造函数要是依赖于其它对象,Dagger2也会帮你自动注入.笔者就不测试了,希望读者亲测一下.  
这里我们引用[依赖注入神器：Dagger2详解系列](https://dreamerhome.github.io/2016/07/07/dagger/?utm_source=tuicool&utm_medium=referral)中的一段话:

我们有两种方式可以提供依赖，一个是注解了@Inject的构造方法，一个是在Module里提供的依赖，那么Dagger2是怎么选择依赖提供的呢，规则是这样的：

* 步骤1：查找Module中是否存在创建该类的方法。
* 步骤2：若存在创建类方法，查看该方法是否存在参数
* 步骤2.1：若存在参数，则按从步骤1开始依次初始化每个参数
* 步骤2.2：若不存在参数，则直接初始化该类实例，一次依赖注入到此结束
* 步骤3：若不存在创建类方法，则查找Inject注解的构造函数，看构造函数是否存在参数
* 步骤3.1：若存在参数，则从步骤1开始依次初始化每个参数
* 步骤3.2：若不存在参数，则直接初始化该类实例，一次依赖注入到此结束

也就说Dagger2会递归的提供依赖.

**@Named和@Qulifier注解的使用**

**@Named**

假设我们现在又有了新的需求,MainActivity中需要两种布料,分别是红布料和蓝布料,但我们的MainModule类中只能提供红布料,怎么办呢?  
读者可能会想:在MainModule类中再添加一个提供蓝布料的方法不就行了:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Provides  public Cloth getRedCloth() {  Cloth cloth = new Cloth();  cloth.setColor("红色");  return cloth;  }  @Provides  public Cloth getBlueCloth() {  Cloth cloth = new Cloth();  cloth.setColor("蓝色");  return cloth;  } |

可问题就来了,Dagger2是通过返回值类型来确定的,当你需要红布料时,它又怎么知道哪个是红布料呢?所以Dagger2为我们提供@Named注解,它怎么使用呢?它有一个value值,用来标识这个方法是给谁用的.修改我们的代码:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Provides  @Named("red")  public Cloth getRedCloth() {  Cloth cloth = new Cloth();  cloth.setColor("红色");  return cloth;  }  @Provides  @Named("blue")  public Cloth getBlueCloth() {  Cloth cloth = new Cloth();  cloth.setColor("蓝色");  return cloth;  } |

我们在getRedCloth方法上使用@Named("red")表明此方法返回的是红布料,同理,在getBlueCloth方法上使用@Named("blue")表明此方法返回的是蓝布料,接下我们只要在MainActivity中的布料字段上同样使用@Named注解,就可以一一配对了.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class MainActivity extends AppCompatActivity {  ...  @Inject  @Named("red")  Cloth redCloth;  @Inject  @Named("blue")  Cloth blueCloth;  @Override  protected void onCreate(Bundle savedInstanceState) {  ...  tv.setText("我现在有" + redCloth + "和" + blueCloth );  }  } |

在redCloth上用@Named("red")标记后,他就会对应Module中对应的方法.

**结果**

[](http://blog.izouxiang.cn/imgs/dagger2demo_4.png)

**@Qulifier**

@Qulifier功能和@Named一样,并且@Named就是继承@Qulifier的,我们要怎么使用@Qulifier注解呢?答案就是自定义一个注解:

|  |  |
| --- | --- |
| 1  2  3  4 | @Qualifier  @Retention(RetentionPolicy.RUNTIME)  public @interface RedCloth {  } |

有了这个注解,我们就可以用它在替换掉上面的@Named("red"),效果是一样的.读者可以亲自试一试.  
而且这两个注解还能使用在依赖参数上,比如这个:

|  |  |
| --- | --- |
| 1  2  3  4 | @Provides  public Clothes getClothes(@Named("blue") Cloth cloth){  return new Clothes(cloth);  } |

效果和上面说明的一样,进入这个方法的cloth由上面有@Named("blue")的方法提供

**@Singleton和@Scope的使用**

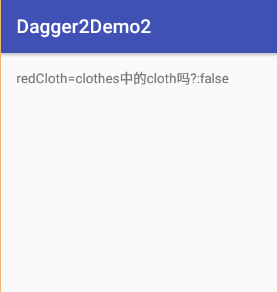
**@Singleton**

假设现在MainActivity中需要依赖Clothes和Cloth,我们在MainModule中提供这两个类的提供方法:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Module  public class MainModule {  @Provides  public Cloth getRedCloth() {  Cloth cloth = new Cloth();  cloth.setColor("红色");  return cloth;  }    @Provides  public Clothes getClothes(Cloth cloth){  return new Clothes(cloth);  }  } |

接着在MainActivity中声明:

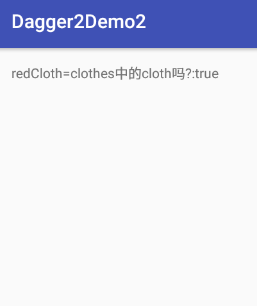
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class MainActivity extends AppCompatActivity {  private TextView tv;  @Inject  Cloth redCloth;  @Inject  Clothes clothes;  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  tv = (TextView) findViewById(R.id.tv);  MainComponent build = DaggerMainComponent.builder().mainModule(new MainModule()).build();  build.inject(this);  tv.setText("redCloth=clothes中的cloth吗?:" + (redCloth == clothes.getCloth()));  }  } |

运行结果:  
[](http://blog.izouxiang.cn/imgs/dagger2demo_5.png)  
你会发现,MainActivity中的Cloth对象和Clothes中的Cloth对象并不是同一个对象,注入过程中,对cloth注入时会调用一次getRedCloth方法,创建了一个Cloth对象;注入Clothes时又会调用一次getRedCloth方法,这时又会创建一个Cloth对象,所以才会出现上面的结果.但是如果需要MainActivity中的Cloth对象和Clothes中的Cloth对象是同一个对象又要怎么办呢?Dagger2为我们提供了@Singleton注解,和名字一样,这个注解的作用就是声明单例模式,我们先看看它怎么使用,下面再讲原理.  
首先,在getRedCloth方法上添加该注解:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Singleton  @Provides  public Cloth getRedCloth() {  Cloth cloth = new Cloth();  cloth.setColor("红色");  return cloth;  } |

再在MainComponent接口上添加该注解:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Singleton  @Component(modules=MainModule.class)  public interface MainComponent {  void inject(MainActivity mainActivity);  } |

我们看看运行结果:  
[](http://blog.izouxiang.cn/imgs/dagger2demo_6.png)  
有没有发现,MainActivity中的Cloth对象和Clothes中的Cloth对象是同一个对象了,是不是很神奇!

**@Scope**

@Singleton是怎么实现的呢?我们先看看@Scope注解,弄懂它,@Singleton你也就会明白了,下面我们就来分析分析  
顾名思义,@Scope就是用来声明作用范围的.@Scope和@Qulifier一样,需要我们自定义注解才能使用,我们先自定义一个注解:

|  |  |
| --- | --- |
| 1  2  3  4 | @Scope  @Retention(RetentionPolicy.RUNTIME)  public @interface PerActivity {  } |

这个注解有什么用呢?答案就是声明作用范围,当我们将这个注解使用在Module类中的Provide方法上时,就是声明这个Provide方法是在PerActivity作用范围内的,并且当一个Component要引用这个Module时,必须也要声明这个Component是PerActivity作用范围内的,否则就会报错,声明方法也很简单,就是在Component接口上使用这个注解.但是我们声明这个作用范围又有什么用呢?原来Dagger2有这样一个机制:在同一个作用范围内,Provide方法提供的依赖对象就会变成单例,也就是说依赖需求方不管依赖几次Provide方法提供的依赖对象,Dagger2都只会调用一次这个方法.就和上面那个例子一样,正常情况下,在注入MainActivity中的Cloth对象时会调用一次getRedCloth方法,注入Clothes对象时因为依赖Cloth对象,所以又会调用一次getRedCloth方法,导致这两个Cloth对象并不是同一个实例.但是我们给它声明作用范围后,这两次对Cloth的依赖只会调用一次getRedCloth方法,这样这两个Cloth对象就是同一实例了,这样就保证了在给MainActivity注入时,所有声明的Cloth依赖都是指向同一个实例.(注意:只有Module类中声明了作用范围的Provide方法才能实现单例,没声明的方法就不是单例的)  
查看源码你会发现Singleton其实是继承@Scope注解的,所以你知道了Singleton是怎么实现单例模式的吧.  
可能有些读者可能会问,Dagger2既然有了Singleton为什么还要我们自定义PerActivity注解?这就涉及到代码可读性了,当依赖需求方是Activity时,我们可以自定义一个PerActivity注解,当依赖需求方是Fragment时,我们又可以自定义一个PerFragment注解,这样我们就能清楚的区分依赖对象的提供目标了  
那我们通过构造函数提供依赖的方式又要怎么声明作用范围呢?答案就是在类名上使用注解标明,**切记不要在构造函数上用注解标明,这样是无效的**.  
读者可以试试用PerActivity注解代替上面例子中的Singleton注解,你会发现效果是一样的

**注意注意注意**:单例是在同一个Component实例提供依赖的前提下才有效的,不同的Component实例只能通过Component依赖才能实现单例.也就是说,你虽然在两个Component接口上都添加了PerActivity注解,但是这两个Component提供依赖时是没有联系的,他们只能在各自的范围内实现单例.(下一个例子会体现到)

**组件依赖dependencies的使用**

在实际开发中,我们经常会使用到工具类,工具类一般在整个App的生命周期内都是单例的,我们现在给我们的Demo添加一个工具类ClothHandler:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class ClothHandler {  public Clothes handle(Cloth cloth){  return new Clothes(cloth);  }  } |

它的功能就是将cloth加工成clothes,假设我们现在有两个Activity中都要使用该工具类,我们要怎么使用Dagger2帮我们注入呢?  
我们先用上面所学的方法试试,先在MainModule中添加提供方法:

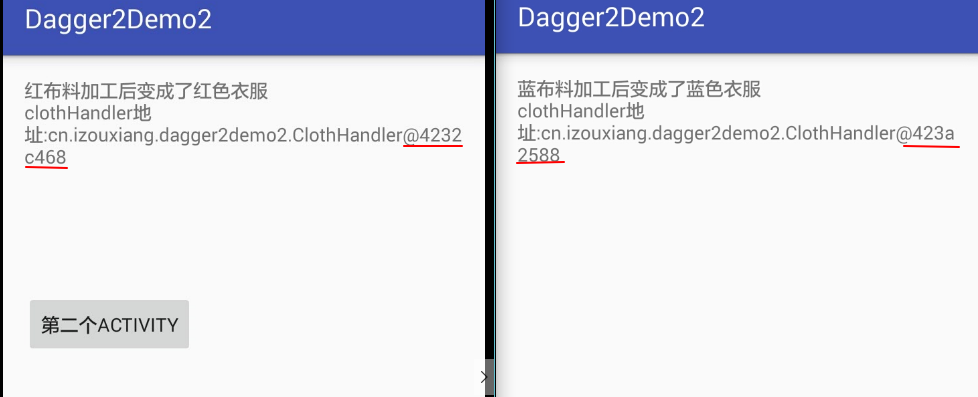
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @Module  public class MainModule {  @PerActivity  @Provides  public Cloth getRedCloth() {  Cloth cloth = new Cloth();  cloth.setColor("红色");  return cloth;  }    @PerActivity  @Provides  public ClothHandler getClothHandler(){  return new ClothHandler();  }  } |

再在MainActivity中声明依赖:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class MainActivity extends AppCompatActivity {  private TextView tv;  @Inject  Cloth redCloth;  @Inject  ClothHandler clothHandler;  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  tv = (TextView) findViewById(R.id.tv);  MainComponent build = DaggerMainComponent.builder().mainModule(new MainModule()).build();  build.inject(this);  tv.setText("红布料加工后变成了" + clothHandler.handle(redCloth) + "\nclothHandler地址:" + clothHandler);  }  //在布局文件中声明的点击方法  public void onClick(View v){  Intent intent = new Intent(this,SecondActivity.class);  startActivity(intent);  }  } |

同理在书写第二个Activity,并为它书写Module类Component接口:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | public class SecondActivity extends AppCompatActivity {  private TextView tv;  @Inject  Cloth blueCloth;  @Inject  ClothHandler clothHandler;  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_second);  tv = (TextView) findViewById(R.id.tv);  SecondComponent component = DaggerSecondComponent.builder().secondModule(new SecondModule()).build();  component.inject(this);  tv.setText("蓝布料加工后变成了" + clothHandler.handle(blueCloth) + "\nclothHandler地址:" + clothHandler);  }  }  @Module  public class SecondModule {  @PerActivity  @Provides  public Cloth getBlueCloth(){  Cloth cloth = new Cloth();  cloth.setColor("蓝色");  return cloth;  }  @PerActivity  @Provides  public ClothHandler getClothHandler(){  return new ClothHandler();  }  }  @PerActivity  @Component(modules = SecondModule.class)  public interface SecondComponent {  void inject(SecondActivity secondActivity);  } |

我们来看看结果:  
[](http://blog.izouxiang.cn/imgs/dagger2demo_7.png)  
你会发现,虽然我们成功的将ClothHandler注入到了这两个Activity中,但是你会发现,这两个Activity中的ClothHandler实例不是一样的(验证了上面那个结论),并且我们发现这种注入方式要在每一个Module中都要提供getClothHandler方法, 假如有20个Activity都需要用到ClothHandler,那我们都这样写,不就代码重复了吗.并且我们还要实现单例,怎么办呢?Dagger2很贴心的为我们提供了Component依赖,就能完美的解决这个问题.  
在面向对象的思想中,我们碰到这种情况一般都要抽取父类,Dagger2也是用的这种思想,我们先创建一个BaseModule,用来提供工具类:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Module  public class BaseModule {    @Singleton //单例  @Provides  public ClothHandler getClothHandler(){  return new ClothHandler();  }  } |

在创建一个BaseComponent接口:

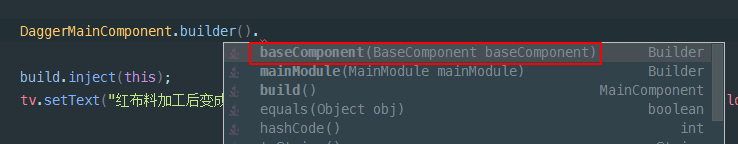
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Singleton //对应Module中声明的单例  @Component(modules = BaseModule.class)  public interface BaseComponent {  ClothHandler getClothHandler();  } |

嗯?这个Component怎么有点不一样,怎么没有inject方法呢?上面讲过,我们通过inject方法依赖需求方实例送到Component中,从而帮助依赖需求方实现依赖,但是我们这个BaseComponent是给其他Component提供依赖的,所以我们就可以不用inject方法,但是BaseComponent中多了一个getClothHandler方法,它的返回值是ClothHandler对象,这个方法有什么用呢?它的作用就是告诉依赖于BaseComponent的Component,BaseComponent能为你们提供ClothHandler对象,如果没有这个方法,BaseComponent就不能提供ClothHandler对象(这个提供规则和上面的依赖规则相同,可以实现单例).既然有了BaseComponent,那我们就可在其它Component中依赖它了.我们删除MainModule和SecondModule中的getClothHandler方法:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | @Module  public class MainModule {  @PerActivity  @Provides  public Cloth getRedCloth() {  Cloth cloth = new Cloth();  cloth.setColor("红色");  return cloth;  }  }  @Module  public class SecondModule {  @PerActivity  @Provides  public Cloth getBlueCloth(){  Cloth cloth = new Cloth();  cloth.setColor("蓝色");  return cloth;  }  } |

接下来在MainComponent和SecondComponent中声明依赖,就要用到@Component中的dependencies属性了:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @PerActivity  @Component(modules=MainModule.class,dependencies = BaseComponent.class)  public interface MainComponent {  void inject(MainActivity mainActivity);  }  @PerActivity  @Component(modules = SecondModule.class,dependencies = BaseComponent.class)  public interface SecondComponent {  void inject(SecondActivity secondActivity);  } |

下面我们用Android Studio中build菜单下的Rebuild Object选项后,你会发现创建MainComponent和SecondComponent实例时多了一个baseComponent方法:  
[](http://blog.izouxiang.cn/imgs/dagger2demo_8.png)  
这个方法需要我们传入一个BaseComponent实例,原因很简单,MainComponent和SecondComponent既然依赖BaseComponent,肯定需要你传入一个BaseComponent实例给它,它才能从BaseComponent实例中获取到它需要的对象嘛.但是需要注意的是,如果要MainComponent和SecondComponent依赖到的对象是同一个的话(也就是单例),创建它们是传入的BaseComponent实例也必须是同一个,上面说过,不同的Component实例是无法提供相同的依赖实例的,因为它们之间是没有联系的.这样的话,我们就需要在MainActivity和SecondActivity中能获取到同一个BaseComponent实例,怎么样能实现呢?很多人一开始都会想到用静态工厂,这种方法可行,但是我们一般都会自定义一个Application类,用它来提供BaseComponent实例,因为在整个App生命周期内都只有一个Application实例,所以其中的BaseComponent实例也不会变.我们自定义一个MyApplication类

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class MyApplication extends Application {  private BaseComponent baseComponent;  @Override  public void onCreate() {  super.onCreate();  baseComponent = DaggerBaseComponent.builder().baseModule(new BaseModule()).build();  }  public BaseComponent getBaseComponent() {  return baseComponent;  }  } |

我们在onCreate方法中创建BaseComponent实例,并对外提供获取方法.  
这种方式还有一种好处,就是当我们在BaseModule中需要用到Application实例时,我们就可以在创建BaseModule时传入this.  
接下来在AndroidManifest.xml中声明我们新建的MyApplication:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | <?xml version="1.0" encoding="utf-8"?>  <manifest package="cn.izouxiang.dagger2demo2"  xmlns:android="http://schemas.android.com/apk/res/android">  <application  android:name=".MyApplication"  android:allowBackup="true"  android:icon="@mipmap/ic\_launcher"  android:label="@string/app\_name"  android:supportsRtl="true"  android:theme="@style/AppTheme">  <activity android:name=".MainActivity">  <intent-filter>  <action android:name="android.intent.action.MAIN"/>  <category android:name="android.intent.category.LAUNCHER"/>  </intent-filter>  </activity>  <activity android:name=".SecondActivity">  </activity>  </application>  </manifest> |

接下来修改MainActivity和SecondActivity中的代码:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class MainActivity extends AppCompatActivity {  ...  @Override  protected void onCreate(Bundle savedInstanceState) {  ...  MainComponent build = DaggerMainComponent  .builder()  .baseComponent(((MyApplication)getApplication()).getBaseComponent())  .mainModule(new MainModule())  .build();  build.inject(this);  tv.setText("红布料加工后变成了" + clothHandler.handle(redCloth) + "\nclothHandler地址:" + clothHandler);  }  ...  }  public class SecondActivity extends AppCompatActivity {  ...  @Override  protected void onCreate(Bundle savedInstanceState) {  ...  SecondComponent component = DaggerSecondComponent  .builder()  .baseComponent(((MyApplication)getApplication()).getBaseComponent())  .secondModule(new SecondModule())  .build();  component.inject(this);  tv.setText("蓝布料加工后变成了" + clothHandler.handle(blueCloth) + "\nclothHandler地址:" + clothHandler);  }  } |

运行结果:  
[](http://blog.izouxiang.cn/imgs/dagger2demo_9.png)  
我们成功的将ClothHandler注入到了这两个Activity中,并且还实现了单例.(注意:这里能实现单例跟BaseComponent中声明了@Singleton有很大关系,因为BaseComponent都没有单例的话,外部依赖它的Component就更不可能单例了).

**Component的生命周期**

一般情况下我们都是在Activity的onCreate方法中创建Component实例,再调用inject方法完成依赖.所以Component依赖可以分为三个过程:

1. 创建Component实例

|  |  |
| --- | --- |
| 1  2  3  4  5 | MainComponent component = DaggerMainComponent  .builder()  .baseComponent(((MyApplication)getApplication()).getBaseComponent())  .mainModule(new MainModule())  .build(); |

1. 调用inject方法

|  |  |
| --- | --- |
| 1 | component.inject(this); |

1. 调用完这个方法整个依赖就完成了.
2. Component实例被销毁  
   onCreate()方法调用完成后,Component实例就会因为没有被引用而被垃圾回收器回收.其中传入给Component实例的Module实例也会一同被回收,这也就能说明不同的Component实例之间是没有联系的(Component依赖除外).

**总结:**

* 至此,Dagger2基础已讲完,对于Dagger2在项目中的使用方法,可以参考github上的开源项目.希望此篇文章能够对你有所帮助!
* 本篇文章是笔者用来记录自己对Dagger2的理解的,如果当中有错误,还请赐教,以便笔者纠正.
* 能够书写本篇文章,还得多亏了各位大神的blog,正因为各位大神的分享精神,才让我们这种小菜鸟能够成长.此篇文章分享出来的目的也就是为了传承这种精神

最后我们引用一下[Dagger2 Scope 注解能保证依赖在 component 生命周期内的单例性吗？](http://blog.piasy.com/2016/04/11/Dagger2-Scope-Instance/)中的注意事项:

* component 的 inject 函数不要声明基类参数；
* Scope 注解必须用在 module 的 provide 方法上，否则并不能达到局部单例的效果；
* 如果 module 的 provide 方法使用了 scope 注解，那么 component 就必须使用同一个注解，否则编译会失败；
* 如果 module 的 provide 方法没有使用 scope 注解，那么 component 和 module 是否加注解都无关紧要，可以通过编译，但是没有局部单例效果；
* 对于直接使用 @Inject 构造函数的依赖，如果把 scope 注解放到它的类上，而不是构造函数上，就能达到局部单例的效果了；

笔者再总结

* 被依赖的Component能提供某个对象时,一定要在接口中声明以该对象为返回值的方法(也就是暴露接口).这样依赖它的Component才能获取到这种对象.

**声明:**

**本文为笔者原创文章,转载需说明!**

**参考文章:**

[依赖注入神器：Dagger2详解系列](https://dreamerhome.github.io/2016/07/07/dagger/?utm_source=tuicool&utm_medium=referral)  
[Dagger2 Scope 注解能保证依赖在 component 生命周期内的单例性吗？](http://blog.piasy.com/2016/04/11/Dagger2-Scope-Instance/)  
[GeekNews](https://github.com/codeestX/GeekNews)  
[Dagger2图文完全教程](https://github.com/luxiaoming/dagger2Demo)