dagger2 使用教程第四节

九风特 关注
2020.09.17 17:51:19 字数 1,751 阅读 226

本节引言

通过前三章的学习， dagger2的主要注解@Inject, @Component, @Module, @Provides我们都进行了解读和检验。本节我们将引出一些新的注解。

给上节小纠结一个交代

上一节我们最后遇到一个问题，就是当Dog的owner需要38岁的实例，MainActivity的User需要28岁，我们怎么办？ 首先，你得让Dagger能提供两种User（age=28 and 38)。其实呢Dagger有一种叫做限定符的注释类型(可以自己定义也可以用Dagger定义好的一个@Named)，那么我们先用@Named来修改一下我们的Module下的Proveider,添加一个age28和age38

```
@Module
class UserModule
{
    @Provides
    fun provideUserCommon():User{
        return User(18)
    }

    @Provides
    @Named("age28")
    fun provideUserYoung():User{
        return  User(28)
    }

    @Provides
    @Named("age38")
    fun provideUserOld():User{
        return User(38)
    }

}
```

此时我们运行程序，会发现两个User实例age都是18 走了第一个Provider,要想他们分别走下面两个就要加@Named限定符

```
class Dog @Inject constructor(@Named("age38") var owner: User)
{
    lateinit var name: String
    override fun toString(): String {
        return "Dog:$name, User:${owner.name} User Age:${owner.age}"
    }
}

class MainActivity : AppCompatActivity() {
    @Inject
    @Named("age28")
    lateinit var user:User
    ...
}
```

现在我们运行程序，发现达到了预期，两个User确实一个28一个38岁，个人认为这种注解只适合构造类型比较少比较固定的时候，像这种岁数的字段还是不要放入构造参数了吧，想想都很闹心的。当然了你可能为了帮助记忆想自己创造注解，那也是可以的，比如我们创建一个YoungUser注解 一个OldUser注解来替代上面的@Named

```
@Qualifier
@Retention(AnnotationRetention.RUNTIME)
annotation class YoungUser

```

```
@Qualifier
@Retention(AnnotationRetention.RUNTIME)
annotation class OldUser
```

替换掉原来所有的@Named即可,我们这里都是为了说明一些知识点，用这种注释来解决这个问题，并不好，后面我们会讲到@BindsInstance，这个东西才是解决这种问题的正途

在此我们只是为了demo。那么至此对全部代码进行一个截图然后在继续吧

```
@Qualifier
@Retention(AnnotationRetention.RUNTIME)
annotation class YoungUser

@Qualifier
@Retention(AnnotationRetention.RUNTIME)
annotation class OldUser

class User @Inject constructor(val age: Int)
{
    lateinit var name:String
    override fun toString(): String {
        return "Name:$name, Age:$age"
    }
}

class Dog @Inject constructor(@OldUser var owner: User)
{
    lateinit var name: String
    override fun toString(): String {
        return "Dog:$name, User:${owner.name} User Age:${owner.age}"
    }
}
@Module
class UserModule
{
    @Provides
    fun provideUserCommon():User{
        return User(18)
    }

    @Provides
    @YoungUser
    fun provideUserYoung():User{
        return  User(28)
    }

    @Provides
    @OldUser
    fun provideUserOld():User{
        return User(38)
    }

}
@Component(modules = [UserModule::class])
interface MainComponent{
    fun inject(activity: MainActivity)
}

class MainActivity : AppCompatActivity() {
    @Inject
    @YoungUser
    lateinit var user:User

    @Inject lateinit var dog:Dog

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        DaggerMainComponent.builder().build().inject(this)
        user.name="Hero"
        dog.name="大黄"
        dog.owner.name="小明"
        buttonUser.setOnClickListener {
            textViewInfo.text = user.toString()
        }
        buttonDog.setOnClickListener {
            textViewInfo.text = dog.toString()
        }

```

```
    }
}
```

哈哈，讲了这么多，最终的demo代码也没几行，的确如此，可能说的有点啰嗦了，见谅吧。继续通过丰富demo来引出其它的知识点吧。

@Binds

接着我们前边的实例，我们假设社会发达了，每个user都有一个车。我们先来建立车这个类，车都是有引擎的，先看引擎的定义：

```
interface Engine{
    fun on()
    fun off()
}
class ChinaEngine @Inject constructor():Engine{
    override fun on() {
        Log.i("zrm", "ChinaEngine on")
    }

    override fun off() {
        Log.i("zrm", "ChinaEngine off")
    }
}
class Car @Inject constructor(val engine: Engine){
    lateinit var name:String
}

```

代码也很好理解，我们定义了一个引擎接口Engine, 一个真实引擎ChinaEngine ,然后Car类需要一个引擎，提供一个名称字段。 很简单

接下来改造User类,让他拥有一辆汽车,为了凸显有车还添加了一个类方法

```
class User @Inject constructor(val age: Int, val car:Car)
{
    lateinit var name:String
    fun gotoCompany()
    {
        car.engine.on()
        Log.i("zrm", "$name goto company")
        car.engine.off()
    }
    override fun toString(): String {
        return "User Name:$name, Age:$age, Car:${car.name}"
    }
}
```

看起来没啥问题吗？ 编译发现 产生一堆错误,我们稍微分析下就会明白，dagger2虽然知道car的构造函数，也知道ChinaEngine的构造函数，但它却不知道你这个car需要的引擎是ChinaEngine。所以我们得让Dagger2知道这件事情，告诉它怎么来提供Engine，那么改造Module

```
@Module
class UserModule
{
    @Provides
    fun provideEngine():Engine
    {
        return ChinaEngine()
    }

    @Provides
    fun provideUserCommon(car:Car):User{
        return User(18, car)
    }

    @Provides
    @YoungUser
    fun provideUserYoung(car:Car):User{
        return  User(28, car)
    }
```

```
20
21      @Provides
22      @OldUser
23      fun provideUserOld(car:Car):User{
24          return User(38, car)
25      }
26
27  }
```

再此为需要Engine的car提供了ChinaEngine, 由于ChinaEngine也是可注入的所以也可以不用new，尝试改成这样的格式:

```
1  class UserModule
2  {
3      @Provides
4      fun provideEngine(engine: ChinaEngine):Engine
5      {
6          return engine
7      }
8      ...
9  }
```

你应该这样理解上述代码 provideEngine的传入参数engine会由Dagger2帮你创建，所以你直接返回它就可以了。

官方文档关于此话题最后说
注意:使用@Binds是定义别名的首选方法，因为Dagger只在编译时需要该模块，并且可以避免在运行时装入该模块
那么我们来尝试使用@Binds，由于@Binds需要修饰一个纯虚接口，我们的其它Provider不是纯虚的，所以新建一个module（本来也是分成两个module更清晰一些)

```
1  @Module
2  interface EngineModule{
3      @Binds
4      fun  bindEngine(impl: ChinaEngine):Engine
5  }
```

在稍微改下Component

```
1  @Component(modules = [UserModule::class,EngineModule::class])
2  interface MainComponent{
3      fun inject(activity: MainActivity)
4  }
```

编译运行 一切okay,好了我们也学习了@Binds的使用，更详细的信息还是要看文档的

本节我们多说一些东西吧，我们来看另外一种需求，比如区公安局给每个用户都发了一本安全手册(SafeNotepad)。那么这些SafeNotepad我们应该怎么创建他们呢，通常来说一个用户创建一个SafeNotepad. 很自然也没啥吐槽的。但是dagger说你这些pad都长的一模一样，而且也没法变化，创建多个其实没啥鸟用。我听了后表示：你说的有点道理，但是。。。 好吧，谁让你是谷歌的亲儿子呢，我相信你说的对。先来看看官方文档的阐述，有点绕口：
----------官方文档开始----------

可重用的范围
有时您希望限制实例化@Inject构造的类或调用@Provides方法的次数，但是您不需要保证在任何特定组件或子组件的生命周期内使用完全相同的实例。这在诸如Android这样的环境中非常有用，因为在这些环境中，分配是很昂贵的。

对于这些绑定，您可以应用 `@Reusable` 作用域。@Reusable作用域绑定与其他作用域不同，它不与任何单个组件关联。相反，实际使用绑定的每个组件将缓存返回的或实例化的对象。

这意味着，如果在组件中安装带有@Reusable绑定的模块，但是只有子组件实际使用该绑定，那么只有该子组件将缓存绑定的对象。如果不共享一个祖先的两个子组件都使用这个绑定，那么它们都将缓存自己的对象。如果组件的祖先已经缓存了对象，子组件将重用它。

不能保证组件只调用绑定一次，因此将@Reusable应用到返回可变对象的绑定，或引用相同实例很重要的对象，是危险的。对于不可变对象使用@Reusable是安全的，如果您不关心它们被分配了多少次，那么您将不考虑它们的作用域。
(注：上边这一大段文字有点不好理解，结合下面代码来理解吧)

```
@Reusable // 用多少勺子没关系，但不要浪费。
class CoffeeScooper {
  @Inject CoffeeScooper() {}
}

@Module
class CashRegisterModule {
  @Provides
  @Reusable // 不要这样做!你在乎你把钱放在哪个收银机里。
            // 应该使用特定的范围代替.
  static CashRegister badIdeaCashRegister() {
    return new CashRegister();
  }
}

@Reusable // 不要这样做!您确实需要每次都使用一个新的过滤器，因此应该取消其作用域。
class CoffeeFilter {
  @Inject CoffeeFilter() {}
}

```

----------官方文档结束----------

看完上述文字更蒙圈的请举手，我的理解是:可重用就相当于是局部单例,我想不出在本例中如何体现它，就弄了个安全手册。。。
好了，老规矩，先建立SafeNotepad类

```
data class SafeNotePad @Inject constructor( val name:String,
                                            val content:String,
                                            val author:String)
```

修改下User类,输出pad

```
class User @Inject constructor(val age: Int, val car:Car, val pad:SafeNotePad)
{
    lateinit var name:String
    fun gotoCompany()
    {
        car.engine.on()
        Log.i("zrm", "$name goto company")
        car.engine.off()
    }
    override fun toString(): String {
        return "User Name:$name, Age:$age, Car:${car.name}, $pad"
    }
}
```

在修改下Module的Provider, 然后跟踪调试程序，发现Dog->Usser下的pad和 MainActivity->User下的pad不是同一个，这是显然的
那我们加上@Reusable会如何?

```
@Module
class UserModule
{
    @Reusable
    @Provides
    fun provideSafeNotepad()=SafeNotePad("安全手册", "不能喝酒", "朝阳分局")
```

```
7
8       @Provides
9       fun provideUserCommon(car:Car, pad:SafeNotePad):User{
10          return User(18, car, pad)
11      }
12
13      @Provides
14      @YoungUser
15      fun provideUserYoung(car:Car, pad:SafeNotePad):User{
16          return  User(28, car, pad)
17      }
18
19      @Provides
20      @OldUser
21      fun provideUserOld(car:Car, pad:SafeNotePad):User{
22          return User(38, car, pad)
23      }
24
25  }
```

现在跟踪发现， 两个pad成员是同一个地址，被重用了。

好了，本节就先到这里吧