dagger2 使用教程第五节

九风特 关注
2020.09.18 17:19:15 字数 1,944 阅读 223

本节引言

前面我们分析了很多Dagger的注解，现在应该有个较全面的认识了。但dagger之所以比别的第三方模块难就是因为它比较抽象，包含的东西还挺多，接下来我们要讨论一些深一点的话题了

进一步讨论Component

既然是组件那就代表这东西可能不止一个，而且之间还可能存在联系。
dagger2的组件之间可能是这3中关系

- 完全独立
- 组件之间有依赖关系，但相对独立
- 成对出现，可说是父子关系

前边我们已经比较依赖我们的实例，现在仍然利用它来进行说明吧

完全独立

完全独立这个关系虽然好理解，但也说一说吧
我们还是来丰富我们原来的demo(只是为了偷懒实际上和之前的逻辑没关系)，但为了更清楚的理解Component的各种关系，我们对新增的一些类不提供@Inject构造函数。
假设我们要在主界面展示一台电脑和一台打印机的信息（界面弄个按钮，输出点信息模拟就是了）
先来创造两个类(非常简单的类,就一个名字),两个Module,两个Component

```
class Computer(private val name:String){
    override fun toString() = "Computer:$name"

}
class Printer(private val name:String){
    override fun toString() = "Printer:$name"
}
@Module
class ComputerModule{
    @Provides
    fun getComputer()=Computer("戴尔")
}

@Module
class PrinterModule{
    @Provides
    fun getPrinter()=Printer("惠普")
}

@Component(modules = [ComputerModule::class])
interface ComputerComponent{
    fun makeComputer():Computer
}

@Component(modules = [PrinterModule::class])
interface PrinterComponent{
    fun makePrinter():Printer
}
```

然后修改UI代码新增代码如下

```
    ...
    lateinit var myComputer:Computer
    lateinit var myPrinter:Printer
    ...

        myComputer = DaggerComputerComponent.create().makeComputer()
        myPrinter = DaggerPrinterComponent.builder().build().makePrinter()
```

```
        buttonShowInfo.setOnClickListener {
            textViewInfo.text = "$myComputer, $myPrinter"
        }
```

点击按钮，我们会看到一切都是按照预期

dell.png

好现在就是两个独立的Component，如果你看了前几节，到目前为止的代码，你都应该能轻松理解。

组件之间有依赖关系，但相对独立

现在我们发现打印机需要一个print功能，但我们发现这个print功能没有电脑不行，打印机无法自己打印，先来修改我们的Printer相关类：

```
class Printer(private val name:String, private val cpu:Computer){
    fun print() = "$cpu is working"
    override fun toString() = "Printer:$name"
}

@Module
class PrinterModule{
    @Provides
    fun getPrinter(cpu:Computer)=Printer("惠普", cpu)
}
```

现在挠头的事情来了，这个打印机需要一个computer参数，这我上哪给你弄去？Dagger又出新主意了，就是让一个Component可以依赖一个或多个其它Component。 先来改写代码

```
@Component(modules = [PrinterModule::class], dependencies = [ComputerComponent::class]
interface PrinterComponent{
    fun makePrinter():Printer
}
```

看到了吧PrinterComponent指定了依赖ComputerComponent, 还需要改写UI代码来明确这种依赖,自己理解下吧，我就不啰嗦了（可以看Dagger生成的源码）

```
        val cpuCom = DaggerComputerComponent.create()
        myComputer = cpuCom.makeComputer()
        myPrinter = DaggerPrinterComponent.builder().computerComponent(cpuCom).build()
```

这种依赖，两个component的也可以说是相互独立的，printer component也可以单独创建，不过没法构建我们希望的printer就是了

成对出现，可说是父子关系

现在我们还有另外一个类，叫做虚拟打印机，这个类和电脑的关系更密切，脱离电脑根本就不可能存在。鉴于这种紧密关系，我们称电脑和虚拟打印机是父子关系，鉴于我们现在已经很熟悉了，把相关的类一股脑贴出来吧:

```
class VirtualPrinter(private val name:String, private val cpu:Computer){
    override fun toString() = "VirtualPrinter:$name"
}

@Module
class VirtualPrinterModule{
    @Provides
    fun getVirtualPrinter(cpu:Computer)=VirtualPrinter("微软虚拟打印机", cpu)
}

@Component(modules = [ComputerModule::class])
interface ComputerComponent{
    fun makeComputer():Computer
    fun makeVirtualComponent():SubComponentVirtualPrinter
}

@Subcomponent(modules = [VirtualPrinterModule::class])
interface SubComponentVirtualPrinter{
    fun makeVirtualPrinter():VirtualPrinter
}
```

UI代码

```
...
lateinit var myVirtualPrinter:VirtualPrinter
...
        val cpuCom = DaggerComputerComponent.create()
        myComputer = cpuCom.makeComputer()
        myPrinter = DaggerPrinterComponent.builder().computerComponent(cpuCom).build()
        myVirtualPrinter = cpuCom.makeVirtualComponent().makeVirtualPrinter()
        buttonShowInfo.setOnClickListener {
            textViewInfo.text = "$myComputer, $myPrinter, $myVirtualPrinter, ${myPrint
        }
...
```

我们看到了一个新的关键字@Subcomponent这就表明这个component是一个子组件，在本例中这个子组件是通过其父组件ComputerComponent的makeVirtualComponent方法来提供的。在这种关系中子组件是不能单独存在的，必须先有父组件才能有子组件。 如果想深挖请看下源码吧，其实也没啥。

本着趁热打铁的精神，我们来继续引入一个概念(实际上我这个使用教程就是官方文档介绍关键字的顺序。。。)

惰性的注入

你可能会说：你这个demo不行，如果用户不点击"User"按钮，不是白白创建User实例了吗，浪费资源啊。我反驳说：大哥，我这是demo不是产品。不，你说的对，我改还不行吗！！！那我先抄一段官方文档的文字行不?

有时需要延迟实例化对象。对于任何绑定T，都可以创建一个 `Lazy<T>` ，它将实例化延迟到第一次调用Lazy<T>的get()方法时。如果T是一个单例，那么Lazy<T>将是ObjectGraph中所有注入的相同实例。否则，每个注入站点将获得自己的惰性<T>实例。无论如何，对Lazy<T>的任何给定实例的后续调用都将返回相同的T的底层实例。

```
class GrindingCoffeeMaker {
  @Inject Lazy<Grinder> lazyGrinder;

```

```
  public void brew() {
    while (needsGrinding()) {
      //Grinder在第一次调用.get()时创建了一次并缓存了它。
      lazyGrinder.get().grind();
    }
  }
}
```

那我们现在就把我们的User对象改成Lazy的

```
lateinit var user:Lazy<User>
```

去掉原来初始化User的代码，在按钮点击中:

```
        buttonUser.setOnClickListener {
            user.get().apply {
                name = "Hero"
                car.name = "大众"
                gotoCompany()
                textViewInfo.text = toString()
            }
        }
```

运行一切okay，没啥问题。这个Lazy还不错，毕竟用的时候在创建类是个不错的事情(computer实例，可以用kotlin自己的lazy体制，就不介绍了)。

提供者注入

有时您需要返回多个实例，而不是只注入一个值。当您有几个选项(工厂、构建器等)时，一个选项是注入 `Provider<T>` 而不仅仅是T。Provider<T>每次调用.get()时都会为T调用绑定逻辑。如果该绑定逻辑是@Inject构造函数，那么将创建一个新的实例，但是@Provides方法没有这样的保证。

```
class BigCoffeeMaker {
  @Inject Provider<Filter> filterProvider;

  public void brew(int numberOfPots) {
  ...
    for (int p = 0; p < numberOfPots; p++) {
      maker.addFilter(filterProvider.get()); //每次换一个新filter。
      maker.addCoffee(...);
      maker.percolate();
      ...
    }
  }
}
```

这玩意我实在想不出啥时候有用,就不分析了, 在来分析一个不太常用的吧

可选的绑定

这个好像有点用吧，我们还是用我们的实例来进行点发挥和说明吧，比如我们的Car类，可以有引擎，也可以没有引擎，当然没引擎的话自然无法执行gotoCompany方法咯，那怎么办呢？这就需要使用@BindsOptionalOf注解了，来看一下官方对此注解的说明吧

如果你想要绑定可以工作，即使某些依赖没有绑定到组件中，你可以添加一个 `@BindsOptionalOf` 方法到一个模块:

```
@BindsOptionalOf abstract CoffeeCozy optionalCozy();

```

这意味着@Inject构造函数和成员以及@Provide方法可以依赖于一个Optional<CoffeeCozy>对象。如果组件中有对CoffeeCozy的绑定，则会出现这个"Optional", 如果没有绑定的CoffeeCozy，这个"Optional"将缺席。

请认真理解吧，让我组织更自然的语言来解释它，我也不会，改造我们的实例，贴代码看看

```
class Car @Inject constructor(val engine:Optional<Engine>){//改造car类添加Optional,说明此
    lateinit var name:String
}

@Module
interface EngineModule{//改造Module来说明这个Engine是个可选注入
    @BindsOptionalOf
    fun  optionalEngine():Engine
}

class User @Inject constructor(val age: Int, val car: Car, val pad: SafeNotePad)
{
    lateinit var name:String
    @RequiresApi(Build.VERSION_CODES.N)
    fun gotoCompany()//改造此方法，以适应car可能没引擎的情况,没引擎则什么也不做
    {
            if(car.engine.isPresent)
            {
                car.engine.get().on()
                Log.i("zrm", "$name goto company")
                car.engine.get().off()
            }
    }
    override fun toString(): String {
        return "User Name:$name, Age:$age, Car:${car.name}, $pad"
    }
}
```

那么现在运行程序，会发现，gotoCompany什么也没做，因为引擎为空了，如果不做@BindsOptionalOf的注释，engine如果不提供Provider的话，是编译不过的，自己可以试试。既然是可选，那就代表也可以有引擎，那我们来写个提供引擎的Module并加入到Component中

```
@Module
class EngineModuleChina{
    @Provides
    fun getEngine(): Engine {
        return ChinaEngine()
    }
}
@Component(modules = [UserModule::class, EngineModule::class, EngineModuleChina::class
interface MainComponent{
    fun inject(activity: MainActivity)
}

```

此时我们运行程序，发现gotoCompany又可以正常工作了。一切到达了预期
思考：这玩意(@BindsOptionalOf)啥时候有用呢，可能是你设计一个类，有很多字段都是需要注入的，但有的字段由于工作进度等等还没设计好提供方法，就可以用这玩意。 等你有了Provider就不用改很多代码了，就添加个Provider就行了。还有别的用途吗？？ 不想了。。。

还想继续，但到目前为止感觉已经基本够用了，等有时间在来个终章，把剩下的一些不太常用的东西继续说说，无论如何本节到此为止吧。