Hilt入门 看这一篇就够了！！

元浩875 LV.5
2021年09月03日 12:00 · 阅读 2375

关注

前文

关于依赖注入，很多小伙伴都没怎么使用过，觉得没有什么用，而且使用起来很麻烦，不过一旦学习完了，你会发现对平时开发有很大的作用。

相比于之前的Dagger，Android中建议使用Hilt来完成自动依赖注入，那这篇文章就来好好聊一下Hilt。

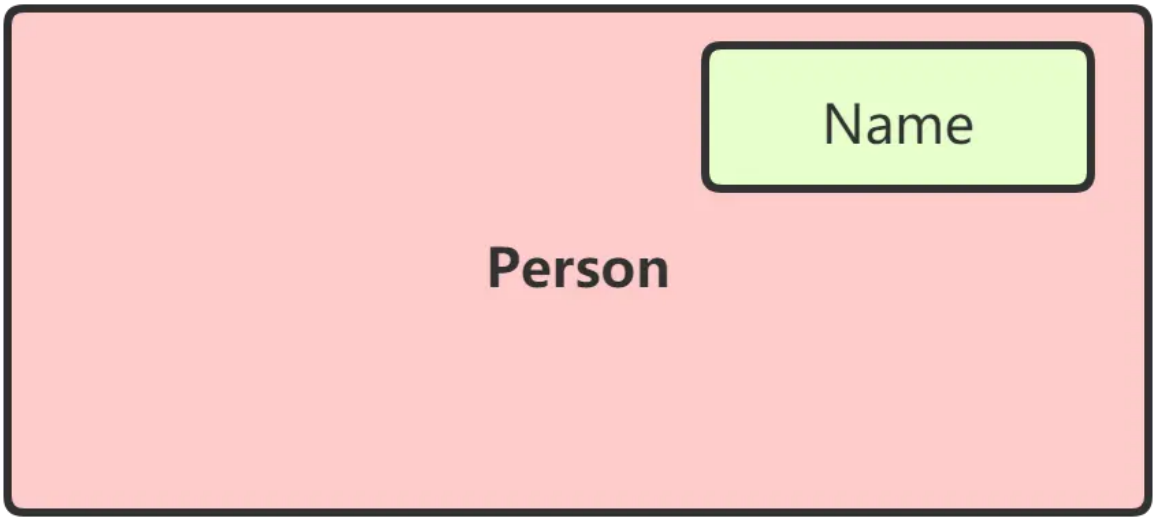
基础介绍

什么是依赖注入

就是类要引用其他类的实例。比如我定义一个类叫做Person，他有一个Name类型的字段name，所以这个Person类就依赖这个Name类型的实例。

```
data class Person(val name: Name)

data class Name(val firstName: String,
                val lastName: String)
```



我们需要Person实例的时候，一般有2种方法来获取Name的实例:

直接构造函数注入

比如我这里的Name类，它是可以通过构造函数来创建实例的，所以得到Name的实例很容易。

字段注入

这啥意思呢，有些类的对象是无法通过构造函数来创建的，比如系统里的Activity实例或者接口类型的实例，那如何做了，就是字段注入。

js 复制代码

```
class Person(){
    lateinit var name: Name
}

data class Name(val firstName: String,
                val lastName: String)

fun test(){
    val person = Person()
    val name = Name("张","三")
    person.name = name

}
```

比如这里name这个字段，就可以在person类创建后，再进行赋值。

自动依赖注入

从前面可以看出，通过代码自行创建依赖项实例并且提供依赖项实例给需要的类，这种方法是手动注入或者人工注入，虽然代码没啥问题，但是如果类的依赖项非常多，而且要严格执行顺序，比如上面例子中在获取打印person.name前必须要实例化好name，这种手动注入的方式就比较容易出错，所以推荐使用自动依赖注入的方式。自动依赖注入目前都只有2个方向：

- 基于反射的解决方案，可以在运行时连接依赖项
- 静态解决方案，可生成在编译时连接依赖项的代码

依赖注入的替代方法

对于依赖注入，除了自动依赖注入，我们平时开发中用的最多的就是服务定位器，也就是搞一个单例类，这个类是容器类或者xxManager之类的，这种类的作用就是提供实例。

```
class Person{
    val name = PersonManager.getName()
}

data class Name(val firstName: String,
                val lastName: String)

object PersonManager{

    fun getName() = Name("张","三")
}

fun test(){
    val person = Person()
    println(person.name)
}
```

js 复制代码

这种方式看似不错，但是也有很多致命缺陷：

- 作用域或者叫做实例的生命周期难以管理，因为很多实例不需要存在在整个生命周期范围内，如果指定一个特定生命周期内，这样又要加很多判断。
- 依赖项在调用类的地方使用，依赖项实现的地方在统一的类中，这个类可能会很大、很多东西，这样不利于修改和测试。

不论是手动依赖注入，还是服务定位器这种方法，都不太好，所以自动依赖库Hilt就很有必要使用了。

Hilt使用

话不多说，直接开整。

添加依赖

先在项目跟目录添加：

```js 复制代码
buildscript {
    ...
    dependencies {
        ...
        classpath 'com.google.dagger:hilt-android-gradle-plugin:2.28-alpha'
    }
}
```

然后在**app的gradle目录和需要用Hilt的moudule都**需要添加以下依赖:

```js 复制代码
...
apply plugin: 'kotlin-kapt'
apply plugin: 'dagger.hilt.android.plugin'

android {
    ...
}

dependencies {
    implementation "com.google.dagger:hilt-android:2.28-alpha"
    kapt "com.google.dagger:hilt-android-compiler:2.28-alpha"
}
```

最后就是项目要求Java 8，在app的gradle中添加：

```js 复制代码
android {
  ...
  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
}
```

Hilt Application

所有使用Hilt框架的应用都必须先使用@HiltAndroidApp来注解应用的Application，

```
@HiltAndroidApp
class App : Application()
```


这个操作是必不可少的，它主要有以下作用：

- 触发Hilt生成代码。
- 会生成一个类叫做Hilt_App，该类相当于一个容器，提供应用级依赖。
- 同时它也是应用的父组件，其他组件可以访问它提供的依赖项。

从这里就可以看出端倪，有一些概念，比如容器，依赖范围，父依赖等。

Hilt Android类

上面说了，我使用@HiltAndroidApp来给Application添加注解后，就可以给Application提供依赖项了，那如何对其他Android类做同样的事呢，这里就需要使用@AndroidEntryPoint这个注解了。

到目前，@AndroidEntryPoint注解能修饰的Android类有以下几种：

- Activity
- Fragment
- View
- Service
- BroadcastReceiver

也就是我使用@AndroidEntryPoint对上面几种Android类添加了注解后，就可以向它里面的字段注入依赖了。

这里注意一点，比如我AActivity依赖于BFragment，所以BFragment使用了@AndroidEntryPoint注解后，AActivity也必须使用该注解。

ok，前面说了2个注解可以让这个类生成一个容器，容器提供依赖，那我定义一个字段，如何标记这个字段的值需要进行依赖注入呢 这个就是@Inject注解执行字段注入。

```
@AndroidEntryPoint
class LoginMVVMActivity : BaseVMActivity<LoginViewModel>(true) {

    //这个字段的值就是注入的
    @Inject lateinit var programmer: Programmer

    private val loginViewModel : LoginViewModel by viewModels {
        InjectorUtils.provideLoginViewModelFactory(this)
    }

    override fun getLayoutResId(): Int = R.layout.activity_login_mvvmactivity

    override fun initView() {
        mBinding.setVariable(BR.viewModel,mViewModel)
    }

    override fun initData() {
        Log.i(TAG, "initData: programmer info = ${programmer.toString()}")
    }

}

//提供该依赖的地方，后面说
class Programmer @Inject constructor(){

    var name: String = "张三"
    var age: Int = 20

    override fun toString() : String{
        return "姓名:$name  年龄:$age"
    }
}
```

比如上面这个例子，我有个字段是programmer，但是它没有进行初始化，按理说会报错，但是这里通过依赖注入，会注入一个programmer的实例，在initData方法中并不会报未初始化异常。

Hilt绑定

前面代码我们进行了Hilt Android类，使用@Inject来注解标记哪些字段需要依赖注入，但是如何向Hilt提供这些依赖项的实例呢，这就需要定义Hilt绑定。

"绑定"这个词翻译感觉有点奇怪，它也就是提供一些方式，就是将某个类型的实例作为依赖项的方式。

构造函数注入

向Hilt提供依赖项绑定信息的最直接办法就是构造函数注入，在类的构造函数前加上@Inject注解，以告知Hilt如何提供该类的对象实例，比如：

```
//构造函数前加上@Inject注解
class Programmer @Inject constructor(){

    var name: String = "张三"
    var age: Int = 20

    override fun toString() : String{
        return "姓名:$name  年龄:$age"
    }
```


这里注意的构造函数是没有参数的，如果构造函数有参数，**那参数就是该类的依赖项，需要提供参数实例的依赖注入**，比如某类是：

```
class Programmer @Inject construcotr(
    val company: Company){
    ...
    }
```


这种情况下，company字段就默认也是@Inject标记注入了，需要Hilt提供Company实例。

Hilt模块

上面我们通过构造函数注入向Hilt提供该类型的实例方法是非常简单，但是有些类型的实例是无法通过构造函数来得到实例的，或者这个类是外部库里的类，也无法为这个类的构造函数添加注解。

这时就需要通过Hilt模块向Hilt框架提供绑定信息。先注意以下几点概念：

- Hilt模块是一个带有@Module注释的类，和Dagger模块一样，它会告知Hilt如何提供某些类型的实例。
- 与Dagger模块不同，Hilt模块必须使用@InstallIn为Hilt模块添加注释，已告知每个模块用在或安装在哪个Android类中。

- Hilt模块中提供的依赖项，可以在生成的所有与Hilt模块安装到的Android类关联的组件中使用。

这里第三点有点绕，简单捋一下，前面Hilt Android类会生成组件，组件里可以使用通过@Inject来注入的实例，而Hilt模块必须安装到Android类中，所以这里也就形成了一个闭环。

使用@Binds注入接口实例

我们来看一下第一种方式，在模块中如何提供依赖项，示例代码如下：

```
//构造函数注入
class Programmer @Inject constructor(){

    var name: String = "张三"
    var age: Int = 20
    //需要提供Work实例
    @Inject lateinit var work: Work

    override fun toString() : String{
        return "姓名:$name  年龄:$age 工作:${work.getWork()}"
    }
}
```

这里多了一个字段是work，这里需要注入，但是看一下Work类型：

```
interface Work {
    fun getWork(): String
}
```

这里是一个接口，就无法通过构造函数注入来实现，这里可以使用@Binds来注入接口实例：

```
//Work接口的一个实现
class Code @Inject constructor() : Work{
    override fun getWork(): String {
        return "coding"
    }
}

//定义一个模块
@Module
@InstallIn(ActivityComponent::class)  //模块安装在Activity组件中
abstract class WorkMdule{
```

```
    //使用@Binds修饰一个函数，用来向Hilt提供实例
    @Binds
    abstract fun bindWork(code: Code): Work
}
```

在上面代码中，定义了一个模块，同时这个模块可以安装到Activity组件中，其实也就是一个范围，也就是这个模块提供的依赖项可以在Activity中使用。

接着便是里面的关键步骤，使用@Binds注入接口实例，所以可以总结以下几点：

- @Binds注解修饰的函数的返回值，就是一个接口，也就是告诉Hilt要提供哪种类型的实例。
- @Binds注解修饰的函数的参数，是接口的一个实现，也就是真正提供的实例，而这个实现也需要依赖注入。

使用@Provides注入实例

对于不能使用构造函数创建的实例，除了上面说的需要提供的类型是接口外，还有一种情况，那就是这个类是来自外部库，比如Retrofit、OkHttpClient等类型，如果要提供这些类型的实例，就需要另一种方式了，我们简单看一下示例代码：

```
js 复制代码
//还是一样的模式，定义一个模块
@Module
@InstallIn(ActivityComponent::class)
object CompanyModule{

    //使用@Provides注解
    @Provides
    fun provideCompany():Company{
        //不是抽象函数了，有具体的函数体
        return CompanyInfo.getCompany()
    }
}

//接口，假如是外部库的
interface Company{
    fun getCompany(): String
}

//接口实现
class HighTech() : Company{
    override fun getCompany(): String {
        return "腾讯"
    }
}
```

```
//假设自己项目中的代码
object CompanyInfo{
    fun getCompany(): Company{
       val highTech = HighTech()
        return highTech
    }
}
```

定义完上面模块后，Company实例就可以提供了。

对于这种情况，我们使用的非常多，这里简单总结一下：

- 和使用@Binds注入接口实例不同，这里的模块类必须是object
- @Provides注解修饰的函数的返回值便是该模块提供的实例类型，而其函数体就是实现该类型实例的方法。

其实，上面便介绍完了所有提供依赖的方式，转而我们更需要了解的便是其中的细节，比如这个依赖实例的范围、要提供同一个类型的多种不同实现的实例该怎么办等问题，我们接着分析。

为同一类型提供多个绑定

前面说了提供依赖的几种方式，假如我现在需要提供一个全局APP范围内单例的OkHttpClient实例，然后特殊需求又要提供一个该类型不同实现的实例，由于返回类型是一样的，按照之前提供绑定的方式都是依据函数返回值来判定提供哪种类型的依赖实例，所以就有问题了。

为了解决问题，Hilt可以为同一个类型提供多个绑定，而用来区分不同实现的就是限定符。

其中限定符就是自定义的注解，还是上面的例子，现在对Company有2种实现，需要区分现在公司和上一家公司，那就定义2种注解：

```
//修饰注解的注解
//这里就定义了2种限定符
@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class CurrentCompany

@Qualifier
@Retention(AnnotationRetention.BINARY)
annotation class LastCompany
```

其实可以看出，限定符的定义和你准备修饰的类型是毫无关系的，只是命名有关系。

然后在在注入依赖和使用依赖的地方都加上注解：

```
@Module
@InstallIn(ActivityComponent::class)
object CompanyModule{

    //这里Company区分开，提供@CurrentCompany类型的实例
    @CurrentCompany
    @Provides
    fun provideCompany():Company{
        return CompanyInfo.getCompany()
    }

    @LastCompany
    @Provides
    fun provideLastCompany(): Company{
        return Shopping()
    }
}
```

这里只是展示了通过@Provides方式提供绑定的方式，假如是@Binds来提供接口绑定的话，直接在传入的参数加上注解即可，这里有需求可以查看官方文档。

在使用的地方：

```
@Inject
@CurrentCompany
lateinit var currentCompany: Company

@Inject
@LastCompany
lateinit var lastCompany: Company
```

最后打印的结果是：

```
programmer info = 姓名:张三  年龄:20 工作:coding 目前公司:腾讯  之前公司:阿里巴巴
```

可以发现这里使用非常简单，也就是通过限定符来做到提供依赖和使用依赖之间做到统一。

Hilt中预定义的限定符

前面说了，限定符就是一种注解，它来区分一种类型，它下面不同的实现，限定这个对象是哪种类型，在Hilt中预定义了2个限定符，分别是@ApplicationContext和@ActivityContext，顾名思义就是对Context这个类型做的区分，是Activity还是Application。

```
//示例代码
class Programmer @Inject constructor(
    //这里的Context就限定是App
    @ApplicationContext val context: Context
)
```

```
@ApplicationContext
@Inject
lateinit var appContext: Context
```

当然有了预定义的限定符，提供该Context的2种绑定也是Hilt帮我们做好了。

Android中Hilt生成的组件

上面我们了解了给Android组件注入依赖以及告诉Hilt提供依赖等，那给一个Android类使用@AndroidEntryPoint后会发生什么呢，它是如何做到的去找依赖，以及把依赖注入到Android类中的呢，这里就涉及一个类叫做组件。

当注入器注入Android类时，会生成Hilt组件类，也就是注入一个类都会有一个关联的Hilt组件类生成，然后Hilt的模块会通过@InstallIn把模块装载到组件中，组件便可以为Android类提供依赖了。

下面是注入器面向的Android对象以及生成的Hilt组件：

- Application -> ApplicationComponent
- ViewModel -> ActivityRetainedComponent
- Activity -> ActivityComponent
- Fragment -> FragmentComponent
- View -> ViewComponent
- Service -> ServiceComponent

组件生命周期

既然Hilt了一个Android类，会生成一个Hilt组件，同时Android类都有生命周期，所以Hilt组件也要有生命周期，不然会造成内存溢出。

组件的生命周期如下图：

生成的组件	创建时机	销毁时机
ApplicationComponent	Application#onCreate()	Application#onDestroy()
ActivityRetainedComponent	Activity#onCreate()	Activity#onDestroy()
ActivityComponent	Activity#onCreate()	Activity#onDestroy()
FragmentComponent	Fragment#onAttach()	Fragment#onDestroy()
ViewComponent	View#super()	视图销毁时
ViewWithFragmentComponent	View#super()	视图销毁时
ServiceComponent	Service#onCreate()	Service#onDestroy()

通过上图可以看出生成的Hilt组件实例的生命周期也是遵循着Android组件生命周期。

组件作用域

前面说了组件生命周期，现在来说一下组件作用域，其实我感觉这个翻译有一点问题，总感觉它俩是一个东西，其实不然。

在默认情况下，Hilt中的所有绑定都没有限定作用域，也就是说每次在代码中请求使用这个绑定时，都会创建所需类型的一个新实例。

所以这就不太符合一些需求了，比如我们想在Activity共享一个实例，这就要给绑定限定一个作用域，而Hilt的做法是把绑定的作用域限定为组件的作用域，比如我一个绑定的作用域是整个Activity，即Activity单例，那就限定生成的组件的作用域即可，所以下面列出的是每个组件的作用域注解：

Android 类	生成的组件	作用域
Application	ApplicationComponent	@Singleton
View Model	ActivityRetainedComponent	@ActivityRetainedScope
Activity	ActivityComponent	@ActivityScoped
Fragment	FragmentComponent	@FragmentScoped
View	ViewComponent	@ViewScoped
带有 @WithFragmentBindings 注释的 View	ViewWithFragmentComponent	@ViewScoped
Service	ServiceComponent	@ServiceScoped

那既然有了作用域，对提供的依赖就可以指定其作用域，比如是单例的话就用@Singleton，如果是Activity内单例的话就使用@ActivityScoped：

```
@Module
@InstallIn(ActivityComponent::class)
abstract class WorkModule{

    @ActivityScoped
    @Binds
    abstract fun bindWork(code: Code): Work
}
```

```
//在Activity中的2个实例将是同一个对象
@Inject lateinit var work: Work
@Inject lateinit var work1: Work
```

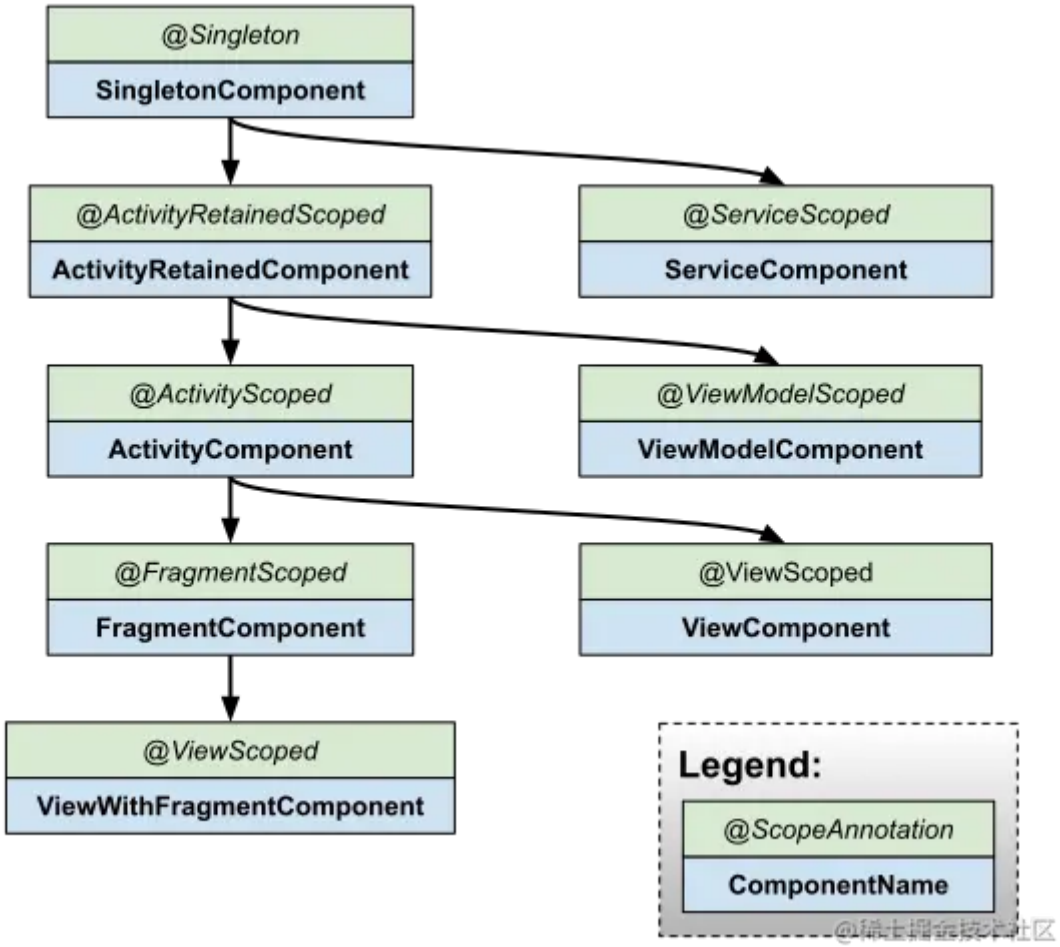
在这个例子中，把模块装载到Activity组件中，然后限定提供的依赖作用域也是Activity，这样在这个Activity范围内就只有一个实例。

同时注意，这里**提供的依赖的作用域必须和模块装载的组件作用域一致**，比如上面例子WorkModule被装载到Activity组件中，那提供的依赖也必须在一样的范围中，如果这里使用别的比如@Singleton来注解bindWork，则会报错。

组件层次结构

上面有了作用域的概念，那肯定就有包含关系，比如一个对象实例它的作用域是整个APP，那我在Activity当然可以访问到，所以这里就有了组件层次结构。

将模块安装到一个组件后，这个模块提供的绑定可以用作该组件其他绑定的依赖项，也可以用于组件层次结构中该组件下的任何子组件中其他绑定的依赖项：



这个也非常容易理解，如果提供的依赖是全APP生命周期，则使用单例注解，它子组件都可以访问到这个实例：

```
//全局作用域
@Singleton
class Skill @Inject constructor(){
    fun getSkill(): String{
        return "Android"
    }
}
```

```
//没有定义作用域，虽然每次创建都会有个Code对象，但是这个skill可以从
    //全局组件中获取
class Code @Inject constructor(val skill: Skill) : Work{
```

```
    override fun getWork(): String {
        return "coding in ${skill.getSkill()}"
    }
}
```

组件默认绑定

这啥意思呢，也非常好理解，每个Hilt组件都有默认绑定，也就是默认得依赖项，比如我Activity得默认依赖项就有Application，这不是废话吗，所以还是看一下每个组件都有哪些默认绑定：

Android 组件	默认绑定
ApplicationComponent	Application
ActivityRetainedComponent	Application
ActivityComponent	Application 和 Activity
FragmentComponent	Application、Activity 和 Fragment
ViewComponent	Application、Activity 和 View
ViewWithFragmentComponent	Application、Activity、Fragment 和 View
ServiceComponent	Application 和 Service

此外，前面还说了2个默认限定符来区分Context，所以再加上有默认绑定，所以可以在任何位置直接使用@ApplicationContext来注入App对象，在Activity作用域中使用@ActivityContext来获取Activity上下文。

总结

到这里这一篇的内容先告一段落，主要说了Hilt的使用，我们大概来总结一下：

- 依赖注入DI的概念，方便提供实例，方便测试等。

- Hilt了Android类后，会生成相应的组件，该组件有生命周期，该组件负责找实例和提供实例。
- 提供Hilt绑定也就是提供实例，可以通过构造函数和Hilt模块来实现，其中Hilt模块需要安装到组件上。
- 绑定可以限定作用域和限定符，以实现在特定组件上该实例是否是单例等。
- 每个组件都有默认的依赖项，比如ActiviyComponent组件，它就有App和Activity这2个默认的依赖项。

关于Hilt的内容还不止这些，关于更高级的用法，我们下一篇文章介绍。

具体可以查看： juejin.cn/post/700446...

分类： Android 标签： Android