date: 2017-09-10 11:16:24 title: 解耦神器dagger2 description: dagger2入门以及深入理解 categories: # 这里写的分类会自动汇集到 categories 页面上，分类可以多级

- dagger2 tags: # 这里写的标签会自动汇集到 tags 页面上
- dagger2

一. Dagger2介绍

1.Dagger2是什么?

Dagger2是由Google接手开发，最早的版本Dagger1 是由Square公司开发的，大神JakeWharton最近也从 Square 公司跳槽到 Google。

```
A fast dependency injector for Android and Java
Android和Java的依赖快速注入器
```

2. Dagger2相较于Dagger1的优势是什么?

- **更好的性能**：相较于Dagger1，它使用的预编译期间生成代码来完成依赖注入，而不是用的反射。大家知道反射对手机应用开发影响是比较大的，因为反射是在程序运行时加载类来进行处理所以会比较耗时，而手机硬件资源有限，所以相对来说会对性能产生一定的影响。
- **容易跟踪调试**：因为dagger2是使用生成代码来实现完整依赖注入，所以完全可以在相关代码处下断点进行运行调试。

3. 使用依赖注入的最大好处是什么?

快速自动的构建出我们所需要的依赖对象，这里的依赖对象可以理解为某一个成员变量。例如在 `MVP` 中， `VP` 层就是互相关联的， `V` 要依赖对应的 `P` ，而 `P` 也要依赖对应的 `V` 。dagger2 能解决的就是这种依赖关系，通过注入的方式，将双方的耦合再次降低，在实际的使用中体现为一个注解想要的对象就创建好了，咱们不用再去管理所依赖对象的创建等情况了。

4. 举个例子

如果在 `MainActivity` 中，有 `Tinno` 的实例，则称 `MainActivity` 对 `Tinno` 有一个依赖。如果不用Dagger2的情况下我们应该这么写：

```
Tinno mTinno;

public MainActivity() {
    mTinno = new Tinno();
}
```

上面例子面临着一个问题，一旦某一天 `Tinno` 的创建方式（如构造参数）发生改变，那么你不但需要修改 `MainActivity` 中创建 `Tinno` 的代码，还要修改其他所有地方创建 `Tinno` 的代码。如果我们使用了Dagger2的话，就不需要管这些了，只需要在需要 `Tinno` 的地方写下：

```
@Inject
```


© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME

```
Tinno mTinno;
```

二. Dagger2使用

1. gradle配置

Android Studio 2.2以前的版本需要使用Gradle插件`android-apt`(Annotation Processing Tool)，协助Android Studio处理`annotation processors`；`annotationProcessor`就是APT工具中的一种，是google开发的内置框架，不需要引入，所以可以像下面这样直接使用。

```
// Add Dagger dependencies
dependencies {
  compile 'com.google.dagger:dagger:2.4'
  annotationProcessor 'com.google.dagger:dagger-compiler:2.4'
}
```

2. 注解

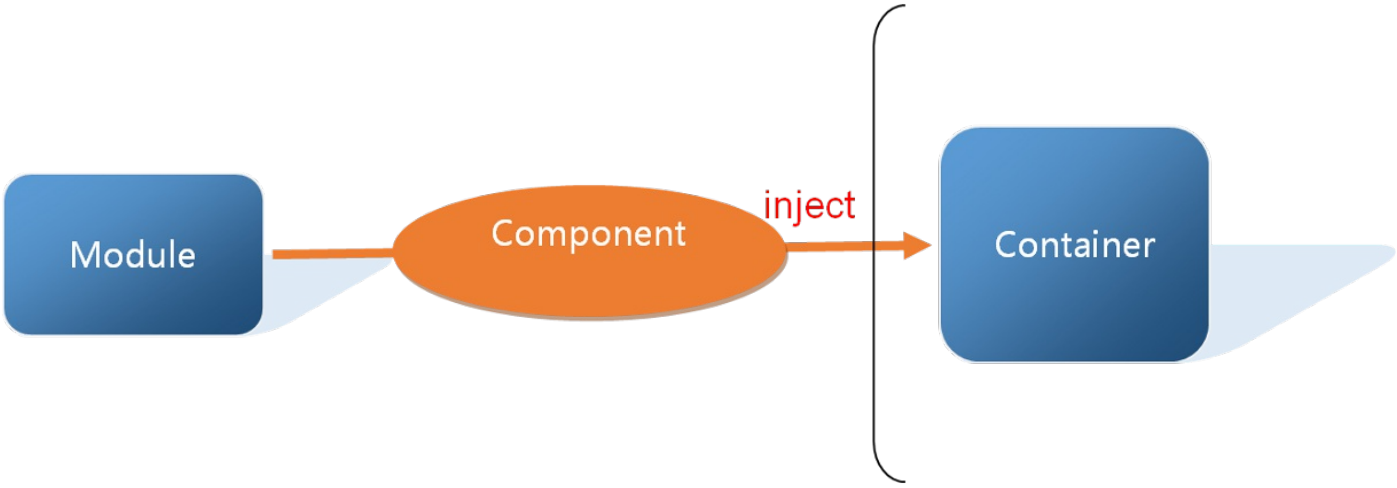
Dagger2 通过注解来生成代码，定义不同的角色，主要的注解如下：

- **@Module**: 用来标注类，Module类里面的方法专门提供依赖，所以我们定义一个类，用@Module注解，这样Dagger在构造类的实例的时候，就知道从哪里去找到需要的依赖。
- **@Provides**: 用来标注方法，在Module中，我们定义的方法是用这个注解，以此来告诉Dagger我们想要构造对象并提供这些依赖。
- **@Inject**: 用来标注对象变量或构造方法，通常在需要依赖的地方使用这个注解。换句话说，你用它告诉Dagger这个类或者字段需要依赖注入。这样，Dagger就会构造一个这个类的实例并满足他们的依赖。
- **@Component**: 通常用来标注接口，Component从根本上来说就是一个注入器，也可以说是@Inject和@Module的桥梁，它的主要作用就是连接这两个部分。将Module中产生的依赖对象自动注入到需要依赖实例的Container中。
- **@Scope**: 标注`Component`和`Module`提供对象的方法，Dagger2可以通过自定义注解限定注解作用域，来管理每个对象实例的生命周期。
- **@Qualifier**: 用来标注方法，当类的类型不足以鉴别一个依赖的时候，我们就可以使用这个注解标示。例如：在Android中，我们会需要不同类型的context，所以我们就可以定义 qualifier注解“@perApp”和“@perActivity”，这样当注入一个context的时候，我们就可以告诉 Dagger我们想要哪种类型的context。

3. 结构

Dagger2要实现一个完整的依赖注入，通常必不可少的元素有三种：**Module**，**Component**，**Container**。为了便于理解，其实可以把`component`想象成`针管`，`module`是`注射器`，里面的`依赖对象`是待`注入的药水`，`build方法`是插进`患者（Container）`，`inject方法`的调用是`推动活塞`。

© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME



4. 简单的例子

申明需要依赖的对象：使用了注解方式，还是以`Tinno`为例，使得Dagger2能找到它。

```
public class Tinno {
    @Inject //这里可以看到加入了注解方式
    public Tinno() {
    }
}
```

申明`Component`接口：申明完后rebuild一下工程，使其自动生成`Component`实现类`DaggerMainActivityComponent`。

```
//用这个标注标识是一个连接器
@Component
public interface MainActivityComponent {
    //这个连接器要注入的对象。这个inject标注的意思是，我后面的参数对象里面有标注为@Inject的属性，这个标注的属性是需要这个连接器注入进来的。
    void inject(MainActivity activity);
}
```

在使用的地方注入，这里是 MainActivity：

```
public class MainActivity extends AppCompatActivity {
    private static final String TAG = "MainActivity";
    @Inject
    Tinno mTinno;//加入注解，标注这个Tinno是需要注入的

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView dagger2TextView = (TextView) findViewById(R.id.dagger2_text_view);
        //使用组件进行构造，注入
        DaggerMainActivityComponent.builder().build().inject(this);
        Log.d(TAG, "onCreate: mTinno = " + mTinno);

        dagger2TextView.setText(mTinno.toString());
    }
}
```

这是最简单的一种使用了。首先我们看到，第一印象是我去 ，这个更复杂了啊 。我只能说确实，因为这个是它对的最基础的使用，看起来很笨拙，但是当它在大型项目里面，在依赖更多的情况下，则会发生质的飞跃，会发现

© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME

它非常好用，并且将你需要传递的参数都隐藏掉，来实现解耦。

5. 常规使用方法

细心的朋友发现了，我在结构中说 dagger2结构的时候提到通常必不可少的三元素，这个例子只用到了 **Component__和__Container**，而 **Module** 并未提及，通过以下这个例子，能更加深刻的理解 **Module** 的作用。实现一个 **MainModule**，提供一些实例构造，通过 **Component** 联系起来。

```
@Module //实现一个类，标注为 Module
public class MainModule {

    @Provides //实现一些提供方法，供外部使用
    public Tinno provideTinno(){
        return new Tinno();
    }
}
```

在 **MainComponent** 中，指明 **Component** 查找 **Module** 的位置

```
@Component(modules = MainModule.class)
public interface MainActivityComponent {// 通常定义为接口，Dagger2框架将自动生成Component的实现类，对应的类名是Dagger×××××，这里对应的实现类是DaggerMainActivityComponent
    void inject(MainActivity activity);// 注入到MainActivity(Container)的方法，方法名一般使用inject
}
```

最后我们的 `Tinno` 类中的 `@Inject` 和构造函数可以去掉了(亲测不去掉也是可以正常运行的，此时也是使用 `Module` 中提供的对象，具体可以通过后面分享的 `@Scope` 来验证，这样说明：**Component会首先从Module维度中查找类实例，若找到就用Module维度创建类实例，并停止查找Inject维度，否则才是从Inject维度查找类实例，所以创建类实例级别Module维度要高于Inject维度。**)，如下所示。

```
public class Tinno {
}
```

注入使用的地方完全不用修改，也能得到和之前例子一样的结果。

6. 更多用法

6.1 方法参数

上面的例子 `@Provides` 标注的方法是没有输入参数的， `Module` 中 `@Provides` 标注的方法是可以带输入参数的，其参数值可以由 `Module` 中的其他被 `@Provides` 标注的方法提供。

```
@Module //实现一个类，标注为 Module
public class MainModule {
    private Context mContext;

    public MainModule(Context context) {
        mContext = context;
    }

    @Provides //实现一些提供方法，供外部使用
    public Tinno provideTinno(Gson gson, CameraTeam cameraTeam) {
        return new Tinno(mContext, gson, cameraTeam);
    }

    @Provides
```


© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME

```
    public Gson provideGson() {
        return new GsonBuilder()
                .excludeFieldsWithModifiers(Modifier.PROTECTED)//忽略protected字段
                .setDateFormat("yyyy-MM-dd'T'HH:mm:ssZ")
                .create();

    }

//    @Provides
//    public CameraTeam provideCameraTeam() {
//        return new CameraTeam();
//    }
}
```

如果找不到被`@Provides`注释的方法提供对应参数对象的话，将会自动调用被`@Inject`注释的构造方法生成相应对象。

```
public class CameraTeam {
    @Inject
    public CameraTeam() {

    }
}
```

由于我们修改了`MainModule`，所以对应注入的地方要稍微修改一下：

```
//注意此处比之前多了.mainModule(new MainModule(getApplicationContext()))
DaggerMainActivityComponent.builder().mainModule(new
MainModule(getApplicationContext())).build().inject(this);
```

思考　　通过上面3个例子`@Provides`和`@Inject`两种方式提供对象的区别?

6.2 添加多个Module

一个`Component`可以添加多个`Module`，这样`Component`获取依赖时候会自动从多个`Module`中查找获取。添加多个`Module`有两种方法，一种是在`Component`的注解`@Component(modules={××××, ×××})`中添加多个`modules`。

```
@Component(modules={MainModule.class,ModuleA.class,ModuleB.class,ModuleC.class}) //直接在Component引用多个 Module
public interface MainActivityComponent {
    ...
}
```

另外一种添加多个`Module`的方法可以使用`@Module`的`includes`的方法`(includes={××××, ×××})`。

```
@Module(includes={ModuleA.class,ModuleB.class,ModuleC.class})//先在一个 Module 中includes其他 Module
public class MainModule {
    ...
}
@Component(modules={MainModule.class}) //只有一个 Module 时可以不用{}
public interface MainActivityComponent {
    ...
}
```

6.3 区分返回类型相同@Provides方法

© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME

如果我们在 `Module` 中有重复的类型返回，例如我定义两个 `Context` 类型的 `provides` 在 `Module` 中的话，编译直接会报错：

```
Error:(16, 10) 错误: android.content.Context is bound multiple times:
@Provides android.content.Context
com.ape.dagger2.MainModule.provideApplicationContext()
@Provides android.content.Context
com.ape.dagger2.MainModule.provideActivityContext()
```

那如果我们真的需要注入同一类型多次呢，这个问题总会有解决方案的吧？要是真的这么坑估计也没人用 dagger 了吧！哈哈。。。 其实 dagger2 为我们提供了两种方式来解决这个问题：

- **可以使用 `@Qualifier` 的注解来区分**
- **`@Named("xx")` 的注解。**

@Named 方式

```
@Module //实现一个类，标注为 Module
public class MainModule {
    private Context mApplicationContext;
    private Context mActivityContext;

    public MainModule(Context context, Context activityContext) {
        mApplicationContext = context;
        mActivityContext = activityContext;
    }

    @Provides //实现一些提供方法，供外部使用
    public Tinno provideTinno(@Named("application")/*使用的是 application*/Context
context, Gson gson, CameraTeam cameraTeam) {
        return new Tinno(context, gson, cameraTeam);
    }

    @Named("application") //标注为 application
    @Provides
    public Context provideApplicationContext() {
        return mApplicationContext;
    }

    @Named("activity") //标注为 activity
    @Provides
    public Context provideActivityContext() {
        return mActivityContext;
    }
  ...
}
```

@Qualifier方式

```
@Qualifier
@Documented
@Retention(RUNTIME)
public @interface ApplicationQualifier {
}
```

```
@Qualifier
```


© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME

```
@Documented
@Retention(RUNTIME)
public @interface ActivityQualifier {
}
```

```
@Module //实现一个类，标注为 Module
public class MainModule {
    private Context mApplicationContext;
    private Context mActivityContext;

    public MainModule(Context context, Context activityContext) {
        mApplicationContext = context;
        mActivityContext = activityContext;
    }

    @Provides //实现一些提供方法，供外部使用
    public Tinno provideTinno(@ApplicationQualifier/*此处使用为 ApplicationQualifier*/
Context context, Gson gson, CameraTeam cameraTeam) {
        return new Tinno(context, gson, cameraTeam);
    }

    @ApplicationQualifier //标注为 ApplicationQualifier
    @Provides
    public Context provideApplicationContext() {
        return mApplicationContext;
    }

    @ActivityQualifier //标注为 ActivityQualifier
    @Provides
    public Context provideActivityContext() {
        return mActivityContext;
    }
    ...
}
```

使用哪种方式就仁者见仁智者见智了，但个人推荐使用 `@Qualifier`，毕竟输入太多字符串容易出错。

6.4 组件间依赖和子组件

有时我们需要依赖一个组件，这个最常见的用法是，如果我们定义了 `MainActivity` 的 `MainComponent`，并且它依赖咱们的 `AppComponent` 里面的 `IRepositoryManager` 的话就要这样定义了：

```
@Component(dependencies = AppComponent.class, modules = MainPresenterModule.class)
public interface MainComponent {
    void inject(MainActivity activity);
}
```

在 `AppComponent` 中需要将获取 `IRepositoryManager` 的方法暴露出来，不然还是无法注入成功的。

```
@Component(modules = {AppModule.class})
public interface AppComponent {
    //用于管理网络请求层,以及数据缓存层,对外开放的接口
    IRepositoryManager repositoryManager();
}
```


© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME

那如果我觉得暴露这些方法太麻烦了，那需要怎么办呢？最简单就是使用 `@SubComponent` ,在所属的父 `Component` 中定义一个 `SubComponent`，该 `SubComponent` 中将会包含父 `Component` 的所有方法，父 `Component` 不显示声明都可以。

```
@Subcomponent(modules = MainPresenterModule.class)
public interface MainComponent {
    void inject(MainActivity activity);
}
```

```
@Component(modules = {AppModule.class})
public interface AppComponent {
    // 提供 MainComponent 对象的获取方法
    MainComponent mainComponent(MainPresenterModule module);
}
```

在注入的时候直接使用父组件的 `mainComponent(MainPresenterModule module)` 包含子组件的 `module`：

```
appComponent.mainComponent(new MainPresenterModule(this)).inject(this);
//DaggerMainComponent.builder().appComponent(appComponent)
//                .mainPresenterModule(new
MainPresenterModule(this)).build().inject(this);
```

组件依赖和子组件的区别：

组件依赖	子组件
1. 保持两个 Component 都独立，没有任何关联 2. 明确的告诉别人这个 Component 所依赖的 Component 3. 两个拥有依赖关系的 Component 是不能有相同 @Scope 注解的 4. 依赖的组件会生成Dagger...Component	1. 保持两个 Component 内聚 2. 不关心这个 Component 依赖哪个 Component 3. 可以使用相同的@Scope注解 4. 子组件的组件不会生成 Dagger...Component

6.5 懒加载和强加载模式

在上面的比喻中，一针扎进去，是啥都给你打进去了，那么如果有些我想要在调用的时候才加载呢？这里 dagger 提供了 Lazy 的方式来注入，对应的获取就是：

```
public class Container{
    @Inject Lazy<Tinno> mTinnoLazy; //延迟加载
    @Inject Provider<Tinno> mTinnoProvider;//实现强制加载，每次调用get都会调用Module的
Provides方法一次，和懒加载模式正好相反,比如我们需要一次性创建出10个Tinno 对象
    public void init(){
        DaggerComponent.create().inject(this);
        Tinno tinno = mTinnoLazy.get();  //调用get时才创建b
    }
}
```

6.6 `@Scope` 详解

© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME

@Scope 是什么　　`Scope` 翻译过来就是辖域，再结合到计算机上，其实就是作用域的意思，学过高级语言的应该都知道设计模式中一个模式叫做单例模式，单例即为全局中该对象的实例只存在一个，而在 dagger2 中，`@scope` 的一个默认实现就是 `@Singleton`，也是Dagger2唯一自带的Scope注解，下面是 `@Singleton` 的源码，乍一看，很神奇啊，仅仅使用一个注解就可以实现单例！

```
@Scope
@Documented
@Retention(RUNTIME)
public @interface Singleton{}
```

可以看到定义一个 `Scope` 注解，通常需要添加以下三部分：

- **@Scope**：注明是Scope
- **@Documented**：标记文档提示，可以不用
- **@Retention(RUNTIME)**：运行时级别

@Scpoe 怎么用　　那么接下来我们就看一下它的使用。代码如下：

```
//普通的对象
public class Tinno {}

@Module//申明Module
public class MainModule {
    @Provides
    @Singleton
    Tinno provideTinno() {
        return new Tinno();
    }
}

@Singleton
@Component(modules = UserModule.class)
public interface MainActivityComponent {//同一个Component可以申明多个注入Container
    void inject(MainActivity activity);
    void inject(SecondActivity activity);
}
```

我们创建一个普通的 `Tinno` 类，然后创建它的 `Module`，并且用 `@Singleton` 标记该 `Tinno` 返回对象，最后我们再创建它的 `Component`，然后用 `@Singleton` 标记这个 `Component`。这是一个标准的套路流程。接下来我们创建一个 `MainActivity` 和一个 `SecondActivity`，代码如下：

```
public class MainActivity extends AppCompatActivity {
    @Inject
    Tinno mTinno1;
    @Inject
    Tinno mTinno2;
    private TextView mContentTextView;
    private Button mContentButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mContentTextView = (TextView) findViewById(R.id.tv_content);
        mContentButton = (Button) findViewById(R.id.btn_content);

        // [1]
        MainActivityComponent component = DaggerMainActivityComponent.create();
        component.inject(this);
```


© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME

```
        // 第一行为 mTinno1 的信息，第二行为 mTinno2 的信息，第三行为该类中
MainActivityComponent 的信息
        mContentTextView.setText(mTinno1.toString() + "\n" + mTinno2.toString()+"\n"+
component.toString());
        mContentButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startActivity(new Intent(MainActivity.this, SecondActivity.class));
            }
        });
    }
}

public class SecondActivity extends AppCompatActivity {
    @Inject
    Tinno mTinno;
    private TextView mContentTextView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
        mContentTextView = (TextView) findViewById(R.id.tv_content);
        // [2]
        MainActivityComponent component = DaggerMainActivityComponent.create();
        component.inject(this);
        // 第一行为 mTinno 的信息，第二行为该类中 MainActivityComponent 的信息
        mContentTextView.setText(mTinno.toString() + "\n" + component.toString());
    }
}
```

运行结果如下图所示，没有问题的，单例实现成功了，发现两个 `Tinno` 的地址是一样的。

```
com.ape.dagger2.Tinno@1fe726f
com.ape.dagger2.Tinno@1fe726f
com.ape.dagger2.DaggerMainActivityComponent@10ebb7c
```

我们仅仅通过一个 `@Singleton` 标记就使得对象实现了单例模式，接下来我们点一下按钮跳转到 `SecondActivity` 中，如下图所示：

```
com.ape.dagger2.Tinno@b45d44f
com.ape.dagger2.DaggerMainActivityComponent@785a7dc
```

但是此时我们发现，不对啊， `SecondActivity` 的 `Tinno` 对象的地址和 `MainActivity` 中的 `Tinno` 对象地址并不一样啊，这个单例好像失效了啊！事实上并不是这样，那么为什么这个单例“失效”了呢？细心的小伙伴们已经看到了，两个 `Activity` 中的 `Component` 对象的地址是并不一样的，这样就好理解了 ——— 由于 `Component` 对象不是同一个，当然它们注入的对象也不会是同一个。那么我们如何解决这个问题呢？ 我们在 `Application` 层初始化 `MainActivityComponent` ，然后在 `Activity` 中直接获取这个 `MainActivityComponent` 对象，由于 `Application` 在全局中只会初始化一次， 所以 `Application` 中的 `MainActivityComponent` 对象只初始化一次，我们每次在 `Activity` 中获取 `Application` 中的这个 `MainActivityComponent` 当然就是同一个的啦。
`Application` 代码如下：

```
public class App extends Application {
    MainActivityComponent mComponent;
```


© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME

```
    @Override
    public void onCreate() {
        super.onCreate();
        mComponent = DaggerMainActivityComponent.create();
    }

    public MainActivityComponent getComponent() {
        return mComponent;
    }
}
```

我们只需要将 [1] 和 [2] 处的代码更改成 `MainActivityComponent component = ((App)getApplication()).getComponent();` 这样我们不就能将我们的 `MainActivityComponent` “单例”化了吗？截图这里就不再贴出了。

自定义@Scpoe

Dagger2中 `@Singleton` 和自己定义的 `@ActivityScope` 、 `@ApplicationScope` 等代码上并没有什么区别，区别是在那种 `Component` 依赖的 `Component` 的情况下，两个 `Component` 的 `@Scope` 不能相同，既然没什么区别，那为什么还要这么做呢？是因为这样标示可以清晰的区分 `Component` 依赖的层次，方便理清我们的代码逻辑层次，如下为自定义的 `ActivityScope` ：

```
@Scope
@Documented
@Retention(RUNTIME)
public @interface ActivityScope {
}
```

有 `@Scope` 注解和没 `@Scope` 注解的编译时生成代码的区别，在编译生成的 `DaggerMainActivityComponent` 的 `initialize` 函数代码中我们可以看到如下：

```
private void initialize(final Builder builder) {
    //标记了`@Singleton`中`DaggerMainActivityComponent`中实例化provideTinnoProvider方式
    this.provideTinnoProvider =

DoubleCheck.provider(MainModule_ProvideTinnoFactory.create(builder.mainModule));
    //未标记
    this.provideTinnoProvider =
MainModule_ProvideTinnoFactory.create(builder.mainModule);
}
```

有 `@Scope` 类注解的 `@Provider` 生成的代码，外层多了一层 `DoubleCheck.provider(…);` 没有 `@Scope` 类注解的则是直接create一个新的实例。关于 `DoubleCheck` ，简单来说就是加了 `@Scope` 的 `Provider` ， `Dagger` 会缓存一个实例在 `DaggerMainComponent` 中，在 `DaggerMainComponent` 中保持单例，缓存的 `provide` 跟随 `DaggerMainComponent` 的生命周期， `DaggerMainComponent` 被销毁时， `provider` 也被销毁，这就是局部单例的概念，假如你的 `DaggerMainComponent` 是在你应用的 `application` 中，则就形成了全局单例。

三. 小结

1. Dagger2到底有哪些好处?

- **增加开发效率、省去重复的简单体力劳动** 首先new一个实例的过程是一个重复的简单体力劳动，dagger2完全可以把new一个实例的工作做了，因此我们把主要精力集中在关键业务上、同时也能增加开发效率上。省去写单例的方法，并且也不需要担心自己写的单例方法是否线程安全，自己写的单例是懒汉模式还是饿汉模式。因为dagger2都可以把这些工作做了。

© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME

- **更好的管理类实例** 每个app中的ApplicationComponent管理整个app的全局类实例，所有的全局类实例都统一交给ApplicationComponent管理，并且它们的生命周期与app的生命周期一样。每个页面对应自己的Component，页面Component管理着自己页面所依赖的所有类实例。因为Component，Module，整个app的类实例结构变的很清晰。
- **解耦** 假如不用dagger2的话，一个类的new代码是非常可能充斥在app的多个类中的，假如该类的构造函数发生变化，那这些涉及到的类都得进行修改。设计模式中提倡把容易变化的部分封装起来。

2. Dagger2在Camera中使用思考！

- **CameraScheduler** 中依赖的各个模块实例可以通过dagger2注入，回调接口也可以在注入的时候传递过去，类似与MVP模式将V传递给P。
- 各个子模块也可以使用dagger2来注入相关实例。

本文所演示的代码在此下载：Dagger2Sample

MVP使用 Dagger2的例子在此下载：MaterialWeather

© Copyright Tuesday, Sep 12, 2017, 8:23 PM by COMPANYNAME