说到编译时注解(RetentionPolicy.CLASS)都要和注解处理器(Annotation Processor) 扯上关系，因为这里是真正体现编译时注解价值的地方。需要注意的一点是，运行时注解(RetentionPolicy.RUNTIME)和源码注解(RetentionPolicy.SOURCE)也可以在注解处理器进行处理，不同的注解有各自的生命周期，根据你实际使用来确定。

注解处理器(Annotation Processor)

首先来了解下什么是注解处理器，注解处理器是 javac 的一个工具，它用来在编译时扫描和处理注解（Annotation）。你可以自定义注解，并注册到相应的注解处理器，由注解处理器来处理你的注解。一个注解的注解处理器，以 **Java** 代码（或者编译过的字节码）作为输入，生成文件（通常是.java 文件）作为输出。这些生成的 Java 代码是在生成的.java 文件中，所以你不能修改已经存在的 Java 类，例如向已有的类中添加方法。这些生成的 Java 文件，会同其他普通的手动编写的 Java 源代码一样被 javac 编译。

自定义注解(RetentionPolicy.CLASS)

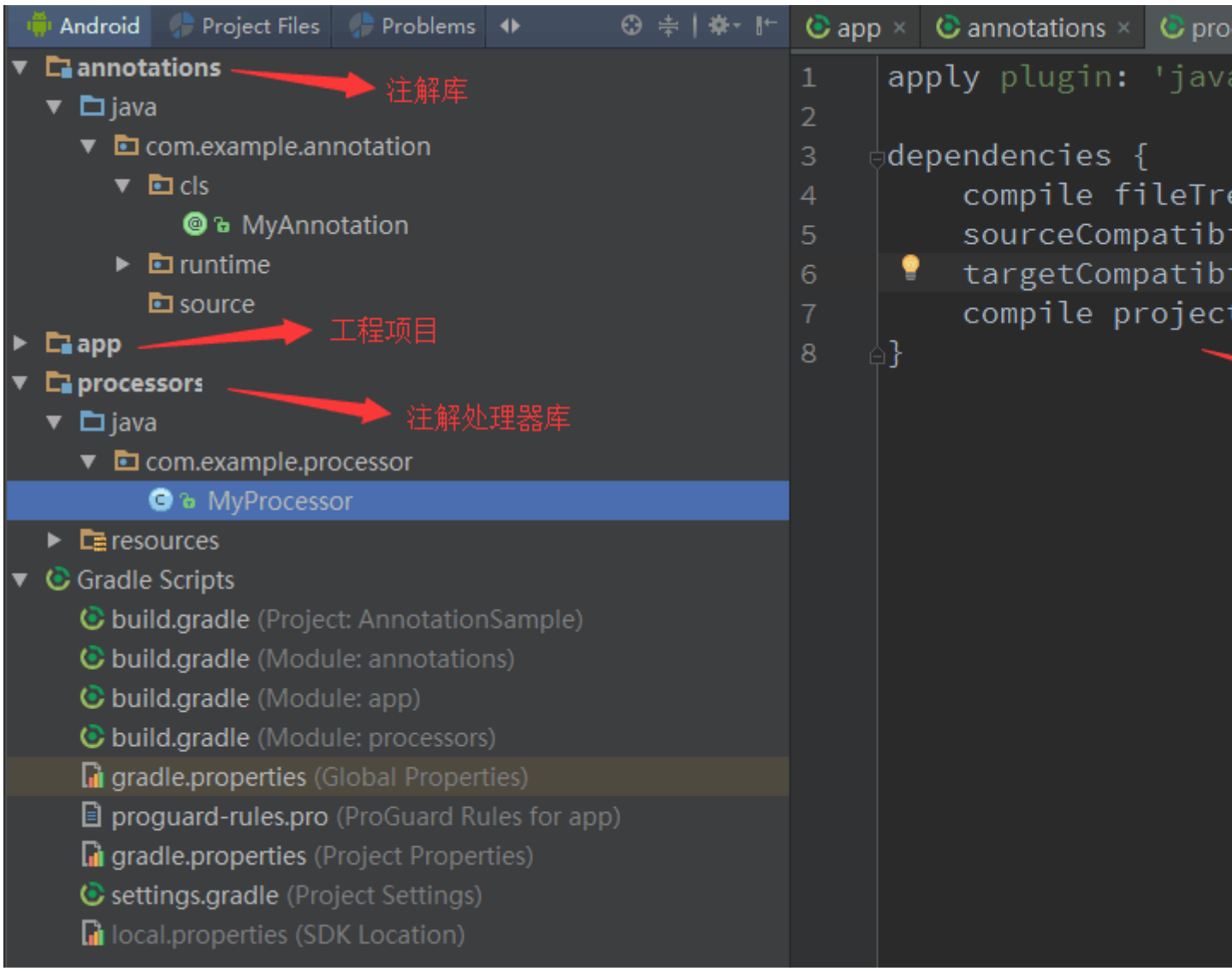
先来定义要使用的注解，这里建一个 Java 库来专门放注解，库名为：annotations，和下面要创建的注解处理器分开，至于为什么要分开创建后面再说。注解库指定 JDK 版本为 1.7，如何指定往下看。自定义注解如下：

```
/**
 * 编译时注解
 */
@Retention(RetentionPolicy.CLASS)
@Target(ElementType.TYPE)
public @interface MyAnnotation {
    String value();
}
```

定义的是编译时注解，对象为类或接口等。

定义注解处理器

下面来定义注解处理器，另外建一个 Java 库工程，库名为：processors，记得是和存放注解的库分开的。注意，这里必须为 Java 库，不然会找不到 javax 包下的相关资源。来看下现在的目录结构：



这里定义一个注解处理器 **MyProcessor**，每一个处理器都是继承于 **AbstractProcessor**，并要求必须复写 **process()**方法，通常我们使用会去复写以下 4 个方法：

```
/**
 * 每一个注解处理器类都必须有一个空的构造函数，默认不写就行;
 */
public class MyProcessor extends AbstractProcessor {

    /**
     * init()方法会被注解处理工具调用，并输入 ProcessingEnviroment 参数。
     * ProcessingEnviroment 提供很多有用的工具类 Elements, Types 和 Filer
     * @param processingEnv 提供给 processor 用来访问工具框架的环境
     */
    @Override
    public synchronized void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
    }

    /**
     * 这相当于每个处理器的主函数 main()，你在这里写你的扫描、评估和处理注解的代码，以及生成 Java 文件。
```

```
     * 输入参数 RoundEnviroment，可以让你查询出包含特定注解的被注解元素
     * @param annotations  请求处理的注解类型
     * @param roundEnv  有关当前和以前的信息环境
     * @return  如果返回 true，则这些注解已声明并且不要求后续 Processor 处理它们；
     *          如果返回 false，则这些注解未声明并且可能要求后续 Processor 处理它们
     */
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        return false;
    }

    /**
     * 这里必须指定，这个注解处理器是注册给哪个注解的。注意，它的返回值是一个字符串的集合，包含本处理器想要处理的注解类型的合法全称
     * @return  注解器所支持的注解类型集合，如果没有这样的类型，则返回一个空集合
     */
    @Override
    public Set<String> getSupportedAnnotationTypes() {
        Set<String> annotataions = new LinkedHashSet<String>();
        annotataions.add(MyAnnotation.class.getCanonicalName());
        return annotataions;
    }

    /**
     * 指定使用的 Java 版本，通常这里返回 SourceVersion.latestSupported()，默认返回 SourceVersion.RELEASE_6
     * @return  使用的 Java 版本
     */
    @Override
    public SourceVersion getSupportedSourceVersion() {
        return SourceVersion.latestSupported();
    }
}
```

上面注释说的挺清楚了，我们需要处理的工作在 process() 方法中进行，等下给出例子。对于 getSupportedAnnotationTypes() 方法标明了这个注解处理器要处理哪些注解，返回的是一个 Set 值，说明一个注解处理器可以处理多个注解。除了在这个方法中指定要处理的注解外，还可以通过注解的方式来指定（SourceVersion 也一样）：

```
@SupportedSourceVersion(SourceVersion.RELEASE_8)
@SupportedAnnotationTypes("com.example.annotation.cls.MyAnnotation")
public class MyProcessor extends AbstractProcessor {
    // ...
}
```

因为兼容的原因，特别是针对 Android 平台，建议使用重载 **getSupportedAnnotationTypes()** 和 **getSupportedSourceVersion()**方法代替**@SupportedAnnotationTypes** 和 **@SupportedSourceVersion**

现在来添加对注解的处理，简单的输出一些信息即可，代码如下：

```
@Override
```

```
2.  public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
3.      // roundEnv.getElementsAnnotatedWith()返回使用给定注解类型的元素
4.      for (Element element : roundEnv.getElementsAnnotatedWith(MyAnnotation.class)) {
5.          System.out.println("------------------------------");
6.          // 判断元素的类型为 Class
7.          if (element.getKind() == ElementKind.CLASS) {
8.              // 显示转换元素类型
9.              TypeElement typeElement = (TypeElement) element;
10.             // 输出元素名称
11.             System.out.println(typeElement.getSimpleName());
12.             // 输出注解属性值
13.             System.out.println(typeElement.getAnnotation(MyAnnotation.class).value());
14.         }
15.         System.out.println("------------------------------");
16.     }
17.     return false;
18. }
```

到这里注解处理器也写好了，下面就看怎么运行它了。

运行注解处理器

在运行前，你需要在主项目工程中引入 annotations 和 processors 这两个库（引入 processors 库不是个好做法，后面介绍更适当的方法）。这时如果你直接编译或者运行工程的话，是看不到任何输出信息的，这里还要做的一步操作是指定注解处理器的所在，需要做如下操作：

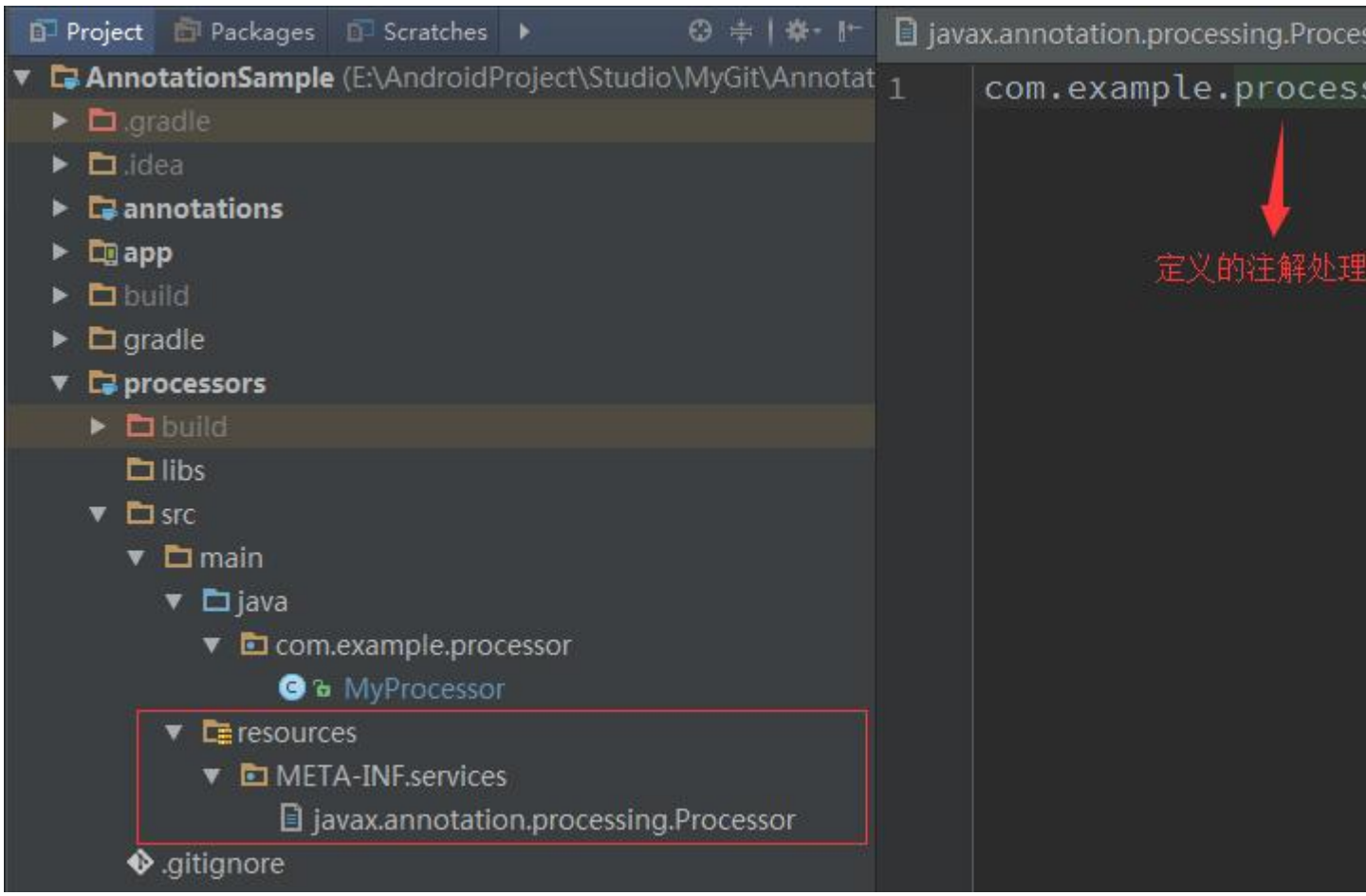
1、在 processors 库的 main 目录下新建 resources 资源文件夹；

2、在 resources 文件夹下建立 META-INF/services 目录文件夹；

3、在 META-INF/services 目录文件夹下创建 javax.annotation.processing.Processor 文件；

4、在 javax.annotation.processing.Processor 文件写入注解处理器的全称，包括包路径；

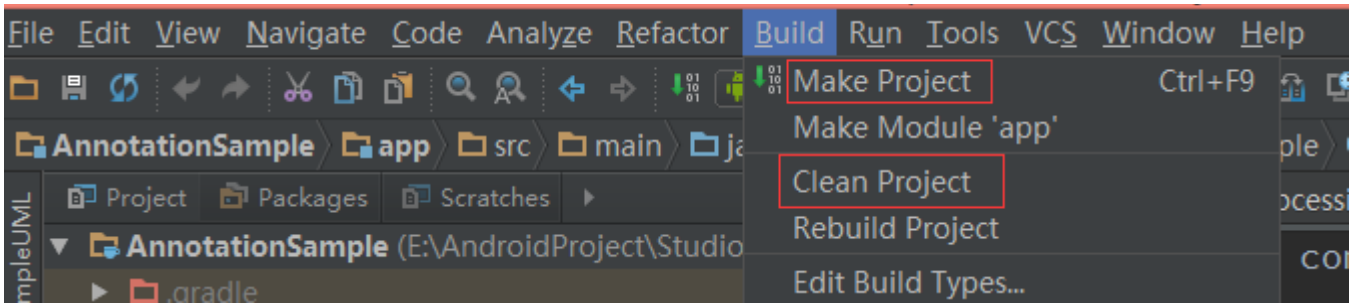
来看下整个目录结构：



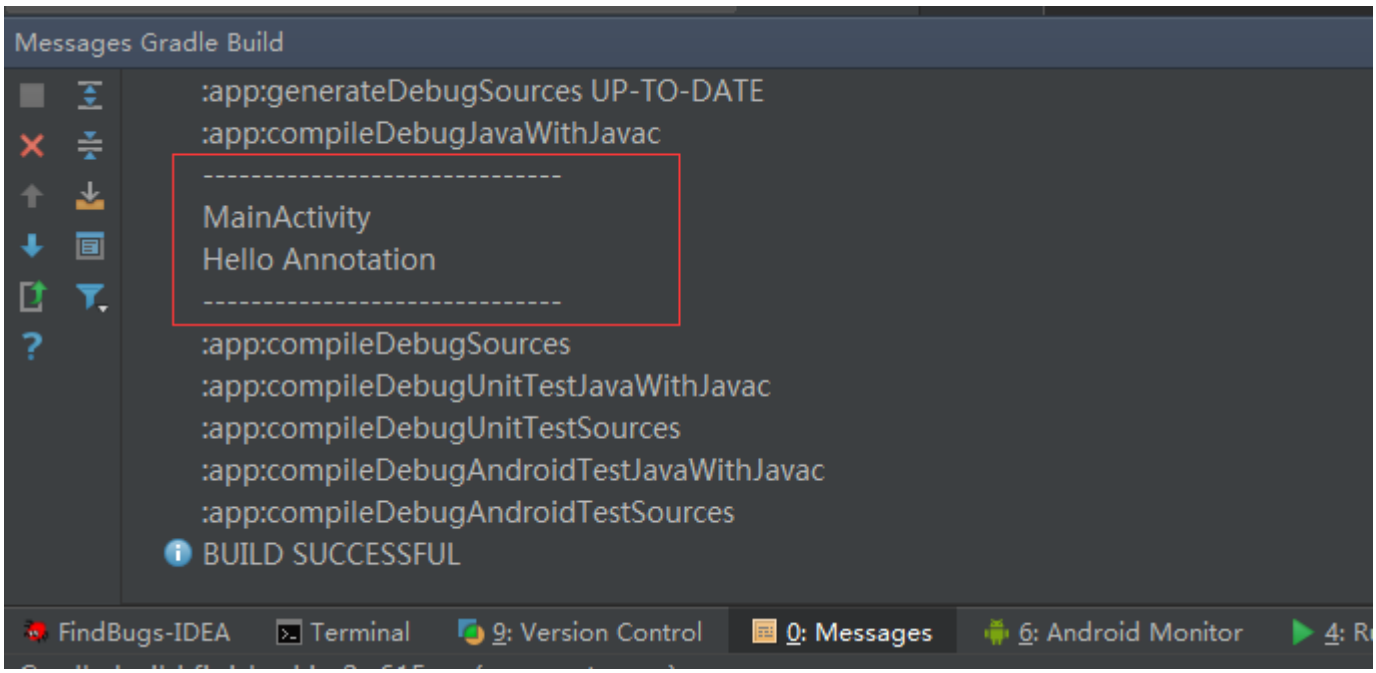
处理完就可以使用了，我们在项目中使用 @MyAnnotation 注解：

```
@MyAnnotation("Hello Annotation")
public class MainActivity extends AppCompatActivity {
    // ...
}
```

到这里我们重新编译下工程就应该有输出了，如果没看到输出则先清理下工程在编译，如下两个操作：



输出信息如下：



现在注解处理器已经可以正常工作了~

当然了，上面还遗留着一个问题，我们的主项目中引用了 processors 库，但注解处理器只在编译处理期间需要用到，编译处理完后就没有实际作用了，而主项目添加了这个库会引入很多不必要的文件，为了处理这个问题我们需要引入个插件 android-apt，它能很好地处理这个问题。

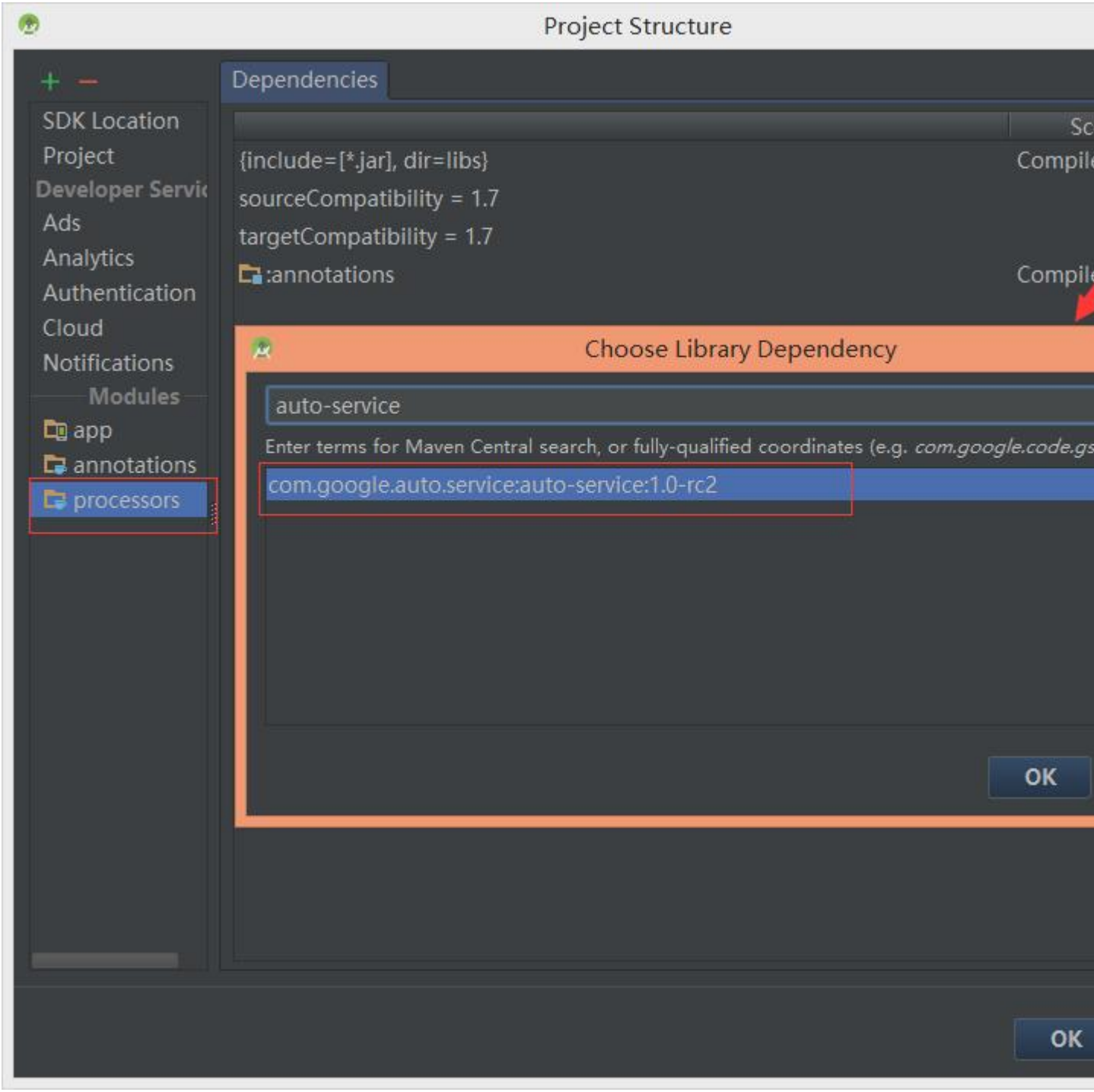
在介绍这个插件前，我想先介绍个好用的库 AutoService，这里有个坑。

AutoService

前面在指定注解处理器的时候你会不会觉得很麻烦？那么多步骤就为添加一个注解处理器，不过没关系，AutoService 可以帮你解决这个问题。

AutoService 注解处理器是 Google 开发的，用来生成 META-INF/services/javax.annotation.processing.Processor 文件的，你只需要在你定义的注解处理器上添加 @AutoService(Processor.class) 就可以了，简直不能再方便了。

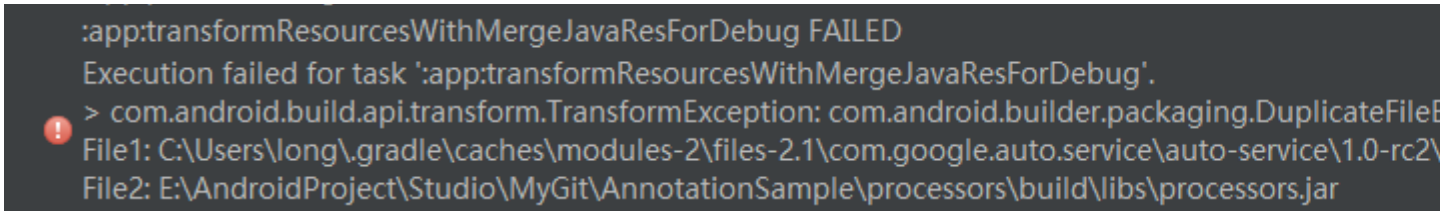
先给 processors 库依赖上 AutoService，你可以直接在 AndroidStudio 工具上搜索添加，如下：



添加好以后就可以直接用了，在我们之前定义的注解处理器上使用：

```
@AutoService(Processor.class)
public class MyProcessor extends AbstractProcessor {
    // ...
}
```

一句话完全搞定！这时重新 Make 下工程也能看到同样的输出信息了。但是如果你编译生成 APK 时，你会发现出现错误了，如下：



发现文件重复了！这里有个解决办法是在主项目的 **build.gradle** 加上这么一段：

```
apply plugin: 'com.android.application'

android {
    // ...
    packagingOptions {
        exclude 'META-INF/services/javax.annotation.processing.Processor'
    }
}
```

这样就不会报错了，这是其中的一个解决方法，还有个更好的解决方法就是用上上面提到的 android-apt 了，下面正式登场

Android-apt

那么什么是 **android**-apt 呢？官网有这么一段描述：

The android-apt plugin assists in working with annotation processors in combination with Android Studio. It has two purposes:

1、Allow to configure a compile time only annotation processor as a dependency, not including the artifact in the final APK or library
2、Set up the source paths so that code that is generated from the annotation processor is correctly picked up by Android Studio

大体来讲它有两个作用：

- 能在编译时期去依赖注解处理器并进行工作，但在生成 APK 时不会包含任何遗留的东西
- 能够辅助 Android Studio 在项目的对应目录中存放注解处理器在编译期间生成的文件

这个就可以很好地解决上面我们遇到的问题了，来看下怎么用。

首先在整个工程的 build.gradle 中添加如下两段语句：

```
buildscript {
    repositories {
        jcenter()
        mavenCentral()  // add
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:2.1.2'
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'  // add
    }
}
```

在主项目(app)的 build.gradle 中也添加两段语句：

```
apply plugin: 'com.android.application'
apply plugin: 'com.neenbedankt.android-apt' // add
```

```
// ...
dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:appcompat-v7:23.4.0'
    compile project(':annotations')
//    compile project(':processors')  替换为下面
    apt project(':processors')
}
```

这样就 OK 了，重新运行可以很好地工作了~

上面提到 android-apt 的作用有对编译时期生成的文件处理，关于生成文件的功能就不得不提 JavaPoet 了，关于这方面的内容放在下个文章里讲~

Demo :https://github.com/Rukey7/AnnotationSample