Android模块开发之APT技术

juexingzhe 关注
2 2017.06.04 13:02:59 字数 1,490 阅读 11,345

APT，就是Annotation Processing Tool 的简称，就是可以在代码编译期间对注解进行处理，并且生成Java文件，减少手动的代码输入。注解我们平时用到的比较多的可能会是运行时注解，比如大名鼎鼎的retrofit就是用运行时注解，通过动态代理来生成网络请求。编译时注解平时开发中可能会涉及的比较少，但并不是说不常用，比如我们经常用的轮子Dagger2, ButterKnife, EventBus3 都在用，所以要紧跟潮流来看看APT技术的来龙去脉。

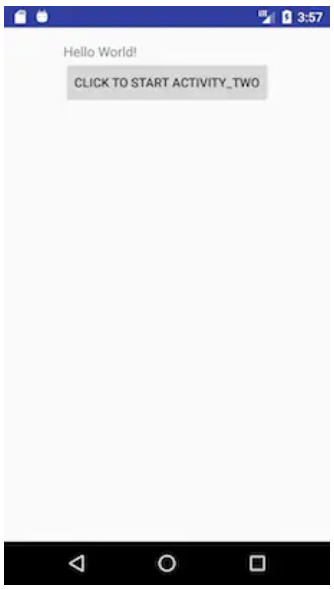
关于注解的大概分类，不清楚的同学可以参考我前面的博客，反射注解与动态代理综合使用。

1.实现目标

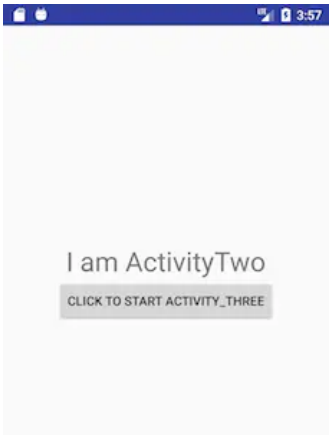
我们还是通过一个栗子来进行分析。平时一般启动Activity都是这样通过startActivity或者startActivityForResult等等balabala。今天我们通过给Activity添加一个注解@RouteAnnotation(name = "RouteName_ActivityTwo"),然后通过注解来启动Activity，AnnocationRouter.getSingleton().navigat("RouteName_ActivityTwo")。

```
1 mContext.startActivity(intent)或者startActivityForResult
```

下面是演示的图片：



1.png



2.png

2.自定义注解

首先就是定义注解RouteAnnotation，作用对象就是类，作用范围就是编译时。然后就是接受一个参数，也就是Activity的别名。这个不清楚请自行参考我上面提到的博客。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.CLASS)
public @interface RouteAnnotation {
    String name();
}
```

3.创建注解处理器

注解处理器一般会是一个项目比较底层的模块，因此我们创建一个Java Library，annotationprocess模块，要依赖前面的注解。自定义的处理器需要继承AbstractProcessor，需要自己实现process方法,一般我们会实现其中的4个方法：

```
public class AnnotationProcessor extends AbstractProcessor {
    @Override
    public synchronized void init(ProcessingEnvironment processingEnvironment) {
    }

    @Override
    public SourceVersion getSupportedSourceVersion() {
    }

    @Override
    public Set<String> getSupportedAnnotationTypes() {
    }

    @Override
    public boolean process(Set<? extends TypeElement> set, RoundEnvironment roundEnvironment)
    }
}
```

> init()方法可以初始化拿到一些使用的工具，比如文件相关的辅助类 Filer;元素相关的辅助类Elements;日志相关的辅助类Messager;
> getSupportedSourceVersion()方法返回 Java 版本;

> getSupportedAnnotationTypes()方法返回要处理的注解的结合;
> 上面几个方法写法基本都是固定的，重头戏是process()方法。

在看具体的代码之前需要先说下Elements这个类，因为后面生成Java文件会经常用到。通过Element可以拿到特定的元素类型。

> TypeElement :类或者接口类型
> VariableElement:成员变量
> ExecuteableElement:成员方法

下面看下我们这个栗子中AnnotationProcessor具体代码：

```
public class AnnotationProcessor extends AbstractProcessor {


    private Filer mFiler;

    @Override
    public synchronized void init(ProcessingEnvironment processingEnvironment) {
        super.init(processingEnvironment);
        mFiler = processingEnvironment.getFiler();
    }

    @Override
    public SourceVersion getSupportedSourceVersion() {
        return SourceVersion.latestSupported();
    }

    @Override
    public Set<String> getSupportedAnnotationTypes() {
        LinkedHashSet<String> types = new LinkedHashSet<>();
        types.add(RouteAnnotation.class.getCanonicalName());
        return types;
    }

    @Override
    public boolean process(Set<? extends TypeElement> set, RoundEnvironment roundEnvironment)
        HashMap<String, String> nameMap = new HashMap<>();

        Set<? extends Element> annotationElements = roundEnvironment.getElementsAnnotatedWith(

        for (Element element : annotationElements) {
            RouteAnnotation annotation = element.getAnnotation(RouteAnnotation.class);
            String name = annotation.name();
            nameMap.put(name, element.getSimpleName().toString());
            //nameMap.put(element.getSimpleName().toString(), name);//MainActiviy-RouteName_Ma
        }

        //generate Java File
        generateJavaFile(nameMap);

        return true;
    }

    private void generateJavaFile(Map<String, String> nameMap) {
        //generate constructor
        MethodSpec.Builder constructorBuidler = MethodSpec.constructorBuilder()
                .addModifiers(Modifier.PUBLIC)
                .addStatement("routeMap = new $T<>()", HashMap.class);
        for (String key : nameMap.keySet()) {
            String name = nameMap.get(key);
            constructorBuidler.addStatement("routeMap.put(\"$N\", \"$N\")", key, name);
        }
        MethodSpec constructorName = constructorBuidler.build();

        //generate getActivityRouteName method
```

```
        MethodSpec routeName = MethodSpec.methodBuilder("getActivityName")
                .addModifiers(Modifier.PUBLIC)
                .returns(String.class)
                .addParameter(String.class, "routeName")
                .beginControlFlow("if (null != routeMap && !routeMap.isEmpty())")
                .addStatement("return (String)routeMap.get(routeName)")
                .endControlFlow()
                .addStatement("return \"\"")
                .build();

        //generate class
        TypeSpec typeSpec = TypeSpec.classBuilder("AnnotationRoute$Finder")
                .addModifiers(Modifier.PUBLIC)
                .addMethod(constructorName)
                .addMethod(routeName)
                .addSuperinterface(Provider.class)
                .addField(HashMap.class, "routeMap", Modifier.PRIVATE)
                .build();


        JavaFile javaFile = JavaFile.builder("com.example.juexingzhe.annotaioncompiletest", ty
        try {
            javaFile.writeTo(mFiler);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

下面分别看下每个方法：

1.init方法中主要是拿到File文件工具对象，后面生成Java文件需要用到;

2.getSupportedSourceVersion就是返回Java版本，这个写法基本固定；

3.getSupportedAnnotationTypes就是返回需要处理的注解，我们这个栗子中就是RouteAnnotation；

4.重点就是process方法，方法里面主要工作就是生成Java文件。我们具体看下步骤：

> 1.roundEnvironment.getElementsAnnotatedWith(RouteAnnotation.class)拿到所有RouteAnnotation注解标注的类
> 2.循环取出注解的name属性和被标注的类名并缓存，其实就是：put("RouteName_ActivityTwo", "ActivityTwo");
> 3.通过javapoet库生成Java类，javapoet是square公司良心出品，让我们脱离手动凭借字符串来生成Java类的痛苦，可以通过各种姿势来生成Java类，这里不多做介绍，有需要的可以看官方文档，很详细。
> 4.最后通过JavaFile.builder("包名",TypeSpec)生成Java文件，包名可以随意取，最后生成的文件都是在主程序模块app.build.generated.source.apt目录下

最后生成的文件就是下面这样，对着生成的文件看上面构造类文件的过程会比较清晰。

```
public class AnnotationRoute$Finder implements Provider {
  private HashMap routeMap;

  public AnnotationRoute$Finder() {
    routeMap = new HashMap<>();
    routeMap.put("RouteName_ActivityTwo", "ActivityTwo");
    routeMap.put("RouteName_MainActivity", "MainActivity");
    routeMap.put("RouteName_ActivityThree", "ActivityThree");
  }

```

```
    public String getActivityName(String routeName) {
      if (null != routeMap && !routeMap.isEmpty()) {
        return (String)routeMap.get(routeName);
      }
      return "";
    }
}
```

4.注册注解处理器

编译的时候JVM怎么找到我们自定义的注解处理器？这个时候就要用到Java SPI机制，这个可以参考我们的上一篇博客Android模块开发之SPI。就是在annotationprocess模块的resources目录下新建META-INF/services，然后新建File，名称 `javax.annotation.processing.Processor` ，文件内容就是我们自定义注解处理器的全限定名 `com.example.AnnotationProcessor`
谷歌官方也出品了一个开源库Auto-Service，通过注解*@AutoService(Processor.class)*可以省略上面配置的步骤，这个后面有机会我们再专门扒一扒。到这里我们就可以使用自定义的注解了。

5.使用自定义注解

在ActivityTwo上面添加注解
`@RouteAnnotation(name = "RouteName_ActivityTwo")` ,然后在MainActivity中就可以通过name查找到ActivityTwo，navigat方法中其实也是封装了intent，然后通过startActivity启动。

```
AnnocationRouter.getSingleton().navigat("RouteName_ActivityTwo");
```

那么既然也是startActivity进行启动，为什么要自定义注解，然后通过navigat进行启动呢？设想一个场景，我们可以将所有Activity通过自定义注解设置别名name，然后外界就可以通过路径来启动Activity，比如构造url路径来启动。这方面也有很多的开源库这样做，比如阿里出品的路由框架ARouter等。

6.总结

APT技术其实就是自定义注解和注解处理器，在编译期间生成Java文件，类似于IOC控制反转，可以方便的进行解耦，在多模块开发时可以基于APT技术构造一套路由框架，去除startActivity等造成的类依赖，也可以通过url方式实现和H5的混合开发。
文字栗子的项目工程和具体跳转封装没有详细说明，本文只专注于APT技术的说明，后面会有专门的文章来说明。

谢谢！

欢迎关注公众号：JueCode