# **Android 利用 APT 技术在编译期生成代码**

APT(Annotation Processing Tool 的简称)可以在代码编译期解析注解，并且成新的 Java 文件，减少手动的代码输入，现在有很多主流库都用上了APT，比如Dagger2, ButterKnife, EventBus3 ，我们要紧跟潮流，与时俱进呐！

下面通过一个简单的 View 注入项目 ViewFinder 来介绍 APT 相关内容，简单实现了类似于ButterKnife 中的两种注解 @BindView 和 @OnClick 。

项目地址：<https://github.com/brucezz/ViewFinder>

大概项目结构如下：

* viewFinder-annotation - 注解相关模块
* viewFinder-compiler - 注解处理器模块
* viewfinder - API 相关模块
* sample - 示例 Demo 模块

### **实现目标**

在通常的 Android 项目中，会写大量的界面，那么就会经常重复地写一些代码，比如：

TextView text = (TextView) findViewById(R.id.tv);text.setOnClickListener(**new** View.OnClickListener() {

**@Override**

**public** **void** **onClick**(View view) {

// on click

}});

天天写这么冗长又无脑的代码，还能不能愉快地玩耍啦。所以，我打算通过 ViewFinder 这个项目替代这重复的工作，只需要简单地标注上注解即可。通过控件 id 进行注解，并且@OnClick 可以对多个控件注解同一个方法。就像下面这样子咯：

**@BindView**(R.id.tv) TextView mTextView;**@OnClick**({R.id.tv, R.id.btn})**public** **void** **onSomethingClick**() {

// on click}

### **定义注解**

*创建 module viewFinder-annotation ，类型为 Java Library，定义项目所需要的注解。*

在 ViewFinder 中需要两个注解 @BindView 和 @OnClick 。实现如下：

**@Retention**(RetentionPolicy.CLASS)**@Target**(ElementType.FIELD)**public** **@interface** BindView {

**int** **value**();}

**@Retention**(RetentionPolicy.CLASS)**@Target**(ElementType.METHOD)**public** **@interface** OnClick {

**int**[] **value**();}

@BindView 需要对成员变量进行注解，并且接收一个 int 类型的参数； @OnClick 需要对方法进行注解，接收一组 int 类型参数，相当于给一组 View 指定点击响应事件。

### **编写 API**

*创建 module viewfinder，类型为 Android Library。在这个 module 中去定义 API，也就是去确定让别人如何来使用我们这个项目。*

首先需要一个 API 主入口，提供静态方法直接调用，就比如这样：

ViewFinder.inject(**this**);

同时，需要为不同的目标（比如 Activity、Fragment 和 View 等）提供重载的注入方法，最终都调用 inject() 方法。其中有三个参数：

* host 表示注解 View 变量所在的类，也就是注解类
* source 表示查找 View 的地方，Activity & View 自身就可以查找，Fragment 需要在自己的 itemView 中查找
* provider 是一个接口，定义了不同对象（比如 Activity、View 等）如何去查找目标 View，项目中分别为 Activity、View 实现了 Provider 接口（具体实现参考[项目代码](https://github.com/brucezz/ViewFinder)吧

**public** **class** **ViewFinder** {

**private** **static** **final** ActivityProvider PROVIDER\_ACTIVITY = **new** ActivityProvider();

**private** **static** **final** ViewProvider PROVIDER\_VIEW = **new** ViewProvider();

**public** **static** **void** **inject**(Activity activity) {

inject(activity, activity, PROVIDER\_ACTIVITY);

}

**public** **static** **void** **inject**(View view) {

// for view

inject(view, view);

}

**public** **static** **void** **inject**(Object host, View view) {

// for fragment

inject(host, view, PROVIDER\_VIEW);

}

**public** **static** **void** **inject**(Object host, Object source, Provider provider) {

// how to implement ?

}}

那么 inject() 方法中都写一些什么呢？

首先我们需要一个接口 Finder，然后为每一个注解类都生成一个对应的内部类并且实现这个接口，然后实现具体的注入逻辑。在 inject() 方法中首先找到调用者对应的 Finder 实现类，然后调用其内部的具体逻辑来达到注入的目的。

接口 Finder 设计如下 ：

**public** **interface** **Finder**<T> {

**void** **inject**(T host, Object source, Provider provider);}

举个 MainActivity 生成 MainActivity$$Finder，对其注解的 View 进行初始化和设置点击事件，这就跟我们平常所写的重复代码基本相同。

**public** **class** **MainActivity$$Finder** **implements** Finder<MainActivity> {

**@Override**

**public** **void** **inject**(**final** MainActivity host, Object source, Provider provider) {

host.mTextView = (TextView) (provider.findView(source, **2131427414**));

host.mButton = (Button) (provider.findView(source, **2131427413**));

host.mEditText = (EditText) (provider.findView(source, **2131427412**));

View.OnClickListener listener;

listener = **new** View.OnClickListener() {

**@Override**

**public** **void** **onClick**(View view) {

host.onButtonClick();

}

};

provider.findView(source, **2131427413**).setOnClickListener(listener);

listener = **new** View.OnClickListener() {

**@Override**

**public** **void** **onClick**(View view) {

host.onTextClick();

}

};

provider.findView(source, **2131427414**).setOnClickListener(listener);

}}

好了，所有注解类都有了一个名为 xx$$Finder 的内部类。我们首先通过注解类的类名，得到其对应内部类的 Class 对象，然后实例化拿到具体对象，调用注入方法。

**public** **class** **ViewFinder** {

// same as above

**private** **static** **final** Map<String, Finder> FINDER\_MAP = **new** HashMap<>();

**public** **static** **void** **inject**(Object host, Object source, Provider provider) {

String className = host.getClass().getName();

**try** {

Finder finder = FINDER\_MAP.get(className);

**if** (finder == **null**) {

Class<?> finderClass = Class.forName(className + "$$Finder");

finder = (Finder) finderClass.newInstance();

FINDER\_MAP.put(className, finder);

}

finder.inject(host, source, provider);

} **catch** (Exception e) {

**throw** **new** RuntimeException("Unable to inject for " + className, e);

}

}}

另外代码中使用到了一点反射，所以为了提高效率，避免每次注入的时候都去找 Finder 对象，这里用一个 Map 将第一次找到的对象缓存起来，后面用的时候直接从 Map 里面取。

到此，API 模块的设计基本搞定了，接下来就是去通过注解处理器为每一个注解类生成 Finder内部类。

### **创建注解处理器**

*创建 module viewFinder-compiler，类型为 Java Library，实现一个注解处理器。*

这个模块需要添加一些依赖：

compile **project**(':viewfinder-annotation')compile 'com.squareup:javapoet:1.7.0'compile 'com.google.auto.service:auto-service:1.0-rc2'

* 因为要用到前面定义的注解，当然要依赖 viewFinder-annotation。
* javapoet 是****方块公司****出的又一个好用到爆炸的裤子，提供了各种 API 让你用各种姿势去生成 Java 代码文件，避免了徒手拼接字符串的尴尬。
* auto-service 是 Google 家的裤子，主要用于注解 Processor，对其生成 META-INF 配置信息。

下面就来创建我们的处理器 ViewFinderProcessor。

**@AutoService**(Processor.class)**public** **class** **ViewFinderProcesser** **extends** AbstractProcessor {

/\*\* \* 使用 Google 的 auto-service 库可以自动生成 META-INF/services/javax.annotation.processing.Processor 文件 \*/

**private** Filer mFiler; //文件相关的辅助类

**private** Elements mElementUtils; //元素相关的辅助类

**private** Messager mMessager; //日志相关的辅助类

**@Override**

**public** **synchronized** **void** **init**(ProcessingEnvironment processingEnv) {

**super**.init(processingEnv);

mFiler = processingEnv.getFiler();

mElementUtils = processingEnv.getElementUtils();

mMessager = processingEnv.getMessager();

}

/\*\* \* @return 指定哪些注解应该被注解处理器注册 \*/

**@Override**

**public** Set<String> **getSupportedAnnotationTypes**() {

Set<String> types = **new** LinkedHashSet<>();

types.add(BindView.class.getCanonicalName());

types.add(OnClick.class.getCanonicalName());

**return** types;

}

/\*\* \* @return 指定使用的 Java 版本。通常返回 SourceVersion.latestSupported()。 \*/

**@Override**

**public** SourceVersion **getSupportedSourceVersion**() {

**return** SourceVersion.latestSupported();

}

**@Override**

**public** **boolean** **process**(Set<? **extends** TypeElement> annotations, RoundEnvironment roundEnv) {

// to process annotations

**return** **false**;

}}

用 @AutoService 来注解这个处理器，可以自动生成配置信息。

在 init() 可以初始化拿到一些实用的工具类。

在 getSupportedAnnotationTypes() 方法中返回所要处理的注解的集合。

在 getSupportedSourceVersion() 方法中返回 Java 版本。

这几个方法写法基本上都是固定的，重头戏在 process() 方法。

*这里插播一下 Element 元素相关概念，后面会用到不少。*

Element 元素，源代码中的每一部分都是一个特定的元素类型，分别代表了包、类、方法等等，具体看 Demo。

**package** **com.example**;

**public** **class** **Foo** { // TypeElement

**private** **int** a; // VariableElement

**private** Foo other; // VariableElement

**public** **Foo**() {} // ExecuteableElement

**public** **void** **setA**( // ExecuteableElement

**int** newA // TypeElement

) {

}}

这些 Element 元素，相当于 XML 中的 DOM 树，可以通过一个元素去访问它的父元素或者子元素。

element.getEnclosingElement();// 获取父元素element.getEnclosedElements();// 获取子元素

注解处理器的整个处理过程跟普通的 Java 程序没什么区别，我们可以使用面向对象的思想和设计模式，将相关逻辑封装到 model 中，使得流程更清晰简洁。分别将注解的成员变量、点击方法和整个注解类封装成不同的 model。

**public** **class** **BindViewField** {

**private** VariableElement mFieldElement;

**private** **int** mResId;

**public** **BindViewField**(Element element) **throws** IllegalArgumentException {

**if** (element.getKind() != ElementKind.FIELD) {

**throw** **new** IllegalArgumentException(

String.format("Only fields can be annotated with @%s", BindView.class.getSimpleName()));

}

mFieldElement = (VariableElement) element;

BindView bindView = mFieldElement.getAnnotation(BindView.class);

mResId = bindView.value();

}

// some getter methods}

主要就是在初始化时校验了一下元素类型，然后获取注解的值，在提供几个 get 方法。OnClickMethod 封装类似。

**public** **class** **AnnotatedClass** {

**public** TypeElement mClassElement;

**public** List<BindViewField> mFields;

**public** List<OnClickMethod> mMethods;

**public** Elements mElementUtils;

// omit some easy methods

**public** JavaFile **generateFinder**() {

// method inject(final T host, Object source, Provider provider)

MethodSpec.Builder injectMethodBuilder = MethodSpec.methodBuilder("inject")

.addModifiers(Modifier.PUBLIC)

.addAnnotation(Override.class)

.addParameter(TypeName.get(mClassElement.asType()), "host", Modifier.FINAL)

.addParameter(TypeName.OBJECT, "source")

.addParameter(TypeUtil.PROVIDER, "provider");

**for** (BindViewField field : mFields) {

// find views

injectMethodBuilder.addStatement("host.$N = ($T)(provider.findView(source, $L))", field.getFieldName(),

ClassName.get(field.getFieldType()), field.getResId());

}

**if** (mMethods.size() > **0**) {

injectMethodBuilder.addStatement("$T listener", TypeUtil.ANDROID\_ON\_CLICK\_LISTENER);

}

**for** (OnClickMethod method : mMethods) {

// declare OnClickListener anonymous class

TypeSpec listener = TypeSpec.anonymousClassBuilder("")

.addSuperinterface(TypeUtil.ANDROID\_ON\_CLICK\_LISTENER)

.addMethod(MethodSpec.methodBuilder("onClick")

.addAnnotation(Override.class)

.addModifiers(Modifier.PUBLIC)

.returns(TypeName.VOID)

.addParameter(TypeUtil.ANDROID\_VIEW, "view")

.addStatement("host.$N()", method.getMethodName())

.build())

.build();

injectMethodBuilder.addStatement("listener = $L ", listener);

**for** (**int** id : method.ids) {

// set listeners

injectMethodBuilder.addStatement("provider.findView(source, $L).setOnClickListener(listener)", id);

}

}

// generate whole class

TypeSpec finderClass = TypeSpec.classBuilder(mClassElement.getSimpleName() + "$$Finder")

.addModifiers(Modifier.PUBLIC)

.addSuperinterface(ParameterizedTypeName.get(TypeUtil.FINDER, TypeName.get(mClassElement.asType())))

.addMethod(injectMethodBuilder.build())

.build();

String packageName = mElementUtils.getPackageOf(mClassElement).getQualifiedName().toString();

// generate file

**return** JavaFile.builder(packageName, finderClass).build();

}}

AnnotatedClass 表示一个注解类，里面放了两个列表，分别装着注解的成员变量和方法。在generateFinder() 方法中，按照上一节设计的模板，利用 JavaPoet 的 API 生成代码。这部分没啥特别的姿势，照着 [JavaPoet 文档](https://github.com/square/javapoet) 来就好了，文档写得很细致。

*有很多地方需要用到对象的类型，普通类型可以用*

*ClassName get(String packageName, String simpleName, String... simpleNames)*

*传入包名、类名、内部类名，就可以拿到想要的类型了（可以参考 项目中TypeUtil类）。*

*用到泛型的话，可以用*

*ParameterizedTypeName get(ClassName rawType, TypeName... typeArguments)*

*传入具体类和泛型类型就好了。*

这些 model 都确定好了之后，process() 方法就很清爽啦。使用 RoundEnvironment 参数来查询被特定注解标注的元素，然后解析成具体的 model，最后生成代码输出到文件中。

**@AutoService**(Processor.class)**public** **class** **ViewFinderProcesser** **extends** AbstractProcessor {

**private** Map<String, AnnotatedClass> mAnnotatedClassMap = **new** HashMap<>();

**@Override**

**public** **boolean** **process**(Set<? **extends** TypeElement> annotations, RoundEnvironment roundEnv) {

// process() will be called several times

mAnnotatedClassMap.clear();

**try** {

processBindView(roundEnv);

processOnClick(roundEnv);

} **catch** (IllegalArgumentException e) {

error(e.getMessage());

**return** **true**; // stop process

}

**for** (AnnotatedClass annotatedClass : mAnnotatedClassMap.values()) {

**try** {

info("Generating file for %s", annotatedClass.getFullClassName());

annotatedClass.generateFinder().writeTo(mFiler);

} **catch** (IOException e) {

error("Generate file failed, reason: %s", e.getMessage());

**return** **true**;

}

}

**return** **true**;

}

**private** **void** **processBindView**(RoundEnvironment roundEnv) **throws** IllegalArgumentException {

**for** (Element element : roundEnv.getElementsAnnotatedWith(BindView.class)) {

AnnotatedClass annotatedClass = getAnnotatedClass(element);

BindViewField field = **new** BindViewField(element);

annotatedClass.addField(field);

}

}

**private** **void** **processOnClick**(RoundEnvironment roundEnv) {

// same as processBindView()

}

**private** AnnotatedClass **getAnnotatedClass**(Element element) {

TypeElement classElement = (TypeElement) element.getEnclosingElement();

String fullClassName = classElement.getQualifiedName().toString();

AnnotatedClass annotatedClass = mAnnotatedClassMap.get(fullClassName);

**if** (annotatedClass == **null**) {

annotatedClass = **new** AnnotatedClass(classElement, mElementUtils);

mAnnotatedClassMap.put(fullClassName, annotatedClass);

}

**return** annotatedClass;

}}

首先解析注解元素，并放到对应的注解类对象中，最后调用方法生成文件。model 的代码中还会加入一些校验代码，来判断注解元素是否合理，数据是否正常，然后抛出异常，处理器接收到之后可以打印出错误提示，然后直接返回 true 来结束处理。

至此，注解处理器也基本完成了，具体细节参考项目代码。

### **实际项目使用**

*创建 module sample，普通的 Android module，来演示 ViewFinder 的使用。*

在整个项目下的 build.gradle 中添加

classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'

然后在 sample module 下的 build.gradle 中添加

apply plugin: 'com.neenbedankt.android-apt'

同时添加依赖：

compile **project**(':viewfinder-annotation')compile **project**(':viewfinder')apt **project**(':viewfinder-compiler')

然后随便创建个布局，随便添加几个控件，就能体验注解啦。

**public** **class** **MainActivity** **extends** AppCompatActivity {

**@BindView**(R.id.tv) TextView mTextView;

**@BindView**(R.id.btn) Button mButton;

**@BindView**(R.id.et) EditText mEditText;

**@OnClick**(R.id.btn)

**public** **void** **onButtonClick**() {

Toast.makeText(**this**, "onButtonClick", Toast.LENGTH\_SHORT).show();

}

**@OnClick**(R.id.tv)

**public** **void** **onTextClick**() {

Toast.makeText(**this**, "onTextClick", Toast.LENGTH\_SHORT).show();

}

**@Override**

**protected** **void** **onCreate**(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

ViewFinder.inject(**this**);

}}

这个时候 build 一下项目，就能看到生成的 MainActivity$$Finder 类了，再运行项目就跑起来了。每次改变注解之后，build 一下项目就好啦。

all done ~

这个项目也就是个玩具级的 APT 项目，借此来学习如何编写 APT 项目。感觉 APT 项目更多地是考虑如何去设计架构，类之间如何调用，需要生成什么样的代码，提供怎样的 API 去调用。最后才是利用注解处理器去解析注解，然后用 JavaPoet 去生成具体的代码。

思路比实现更重要，设计比代码更巧妙。