


# Android编译期插桩，让程序自己写代码(一)



Android高级架构师 (/u/cff70524f5cf) [+ 关注](#)  
0.6 2019.04.03 22:29 字数 2253 阅读 1192 评论 0 喜欢 51  
(/u/cff70524f5cf)

## 前言

近些年，编译期插桩技术在Android圈越来越普遍。无论是可以生成JAVA源码的ButterKnife、Dagger，还是操作字节码的VirtualAPK，甚至是新兴的语言Kotlin都用到了编译期插桩技术。学习这门技术对我们理解这些框架的原理十分有帮助。另外，我们通过这种技术可以抽离出复杂、重复的代码，降低程序耦合性，提高代码的可复用性，提高开发效率。因此，了解编译期插桩技术十分必要。在介绍这项技术之前，我们先来了解一下Android代码的编译过程以及插桩位置。话不多说，直接上图。



image.png

## APT

APT(Annotation Processing Tool)是一种编译期注解处理器。它通过定义注解和处理器来实现编译期生成代码的功能，并且将生成的代码和源代码一起编译成.class文件。

**代表框架：**ButterKnife、Dagger、ARouter、EventBus3、DataBinding、AndroidAnnotation等。

在介绍如何应用APT技术之前，我们先来了解一些相关的知识。

## 一、Element

### 1.简介

Element 是一种在**编译期**描述.java文件静态结构的一种类型，它可能表示一个package、一个class、一个method或者一个field。Element 的比较应该使用 equals，因为编译期间同一个 Element 可能会用两个对象表示。JDK提供了以下5种 Element。

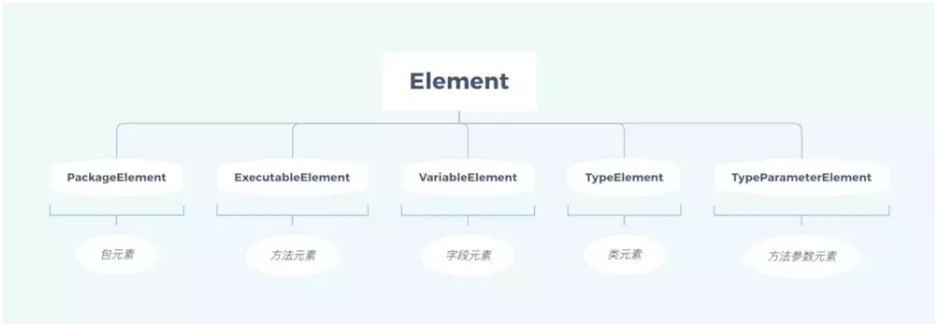


image.png

## 2.Element的存储结构

编译器采用类似Html的Dom树来存储Element。我们用下面的 Test.java 来具体说明。

```
//PackageElement
package me.zhangkuo.compile;

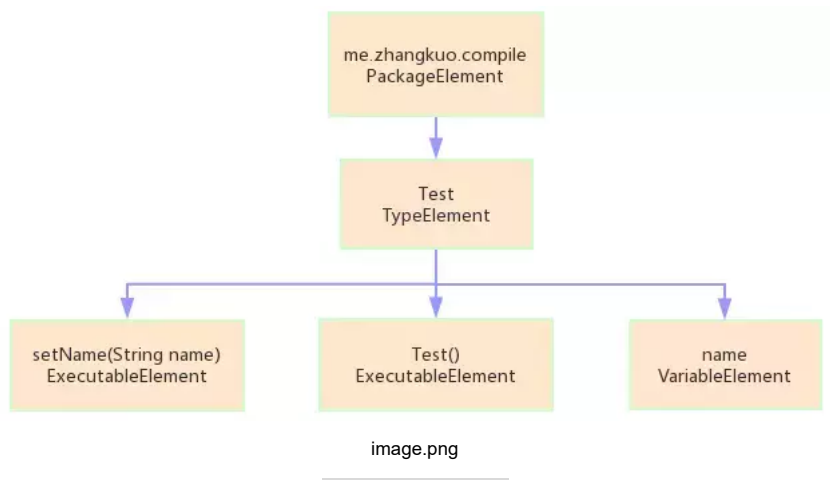
//TypeElement
public class Test {

    //VariableElement
    private String name;

    //ExecutableElement
    private Test(){
    }

    //ExecutableElement
    public void setName(/* TypeParameterElement */ String name) {
        this.name = name;
    }
}
```

Test.java 用Element树结构描述如下：



我们可以看到 setName(String name) 的 ExecutableElement 中并没有子节点 TypeParameterElement。这是因为 TypeParameterElement 没有被纳入到 Element 树中。不过我们可以通过 ExecutableElement 的 getTypeParameters() 方法来获取。

此外，再给大家介绍两个Element中十分有用的方法。

```
public interface Element extends AnnotatedConstruct {
    //获取父Element
    Element getEnclosingElement();
    //获取子Element的集合
    List<? extends Element> getEnclosedElements();
}
```

二、TypeMirror

Element 有一个 asType() 方法用来返回 TypeMirror。TypeMirror 表示 Java 编程语言中的类型。这些类型包括基本类型、声明类型（类和接口类型）、数组类型、类型变量和 null 类型。还可以表示通配符类型参数、executable 的签名和返回类型，以及对应于包和关键字 void 的伪类型。我们一般用TypeMirror进行类型判断。如下段代码，用来比较元素所描述的类型是否是Activity的子类。

^

+

🔖

❤

🔗

```
/**
 * 类型相关工具类
 */
private Types typeUtils;
/**
 * 元素相关的工具类
 */
private Elements elementUtils;
private static final String ACTIVITY_TYPE = "android.app.Activity";

private boolean isSubActivity(Element element){
    //获取当前元素的TypeMirror
    TypeMirror elementTypeMirror = element.asType();
    //通过工具类Elements获取Activity的Element，并转换为TypeMirror
    TypeMirror viewTypeMirror = elementUtils.getTypeElement(ACTIVITY_TYPE).asType();
    //用工具类typeUtils判断两者间的关系
    return typeUtils.isSubtype(elementTypeMirror, viewTypeMirror)
}
```

### 三、一个简单的ButterKnife

这一节我们通过编写一个简单的 ButterKnife 来介绍一下如何编写一个APT框架。APT应该是编译期插桩最简单的一种技术，通过三步就可以完成。

#### 1. 定义编译期注解。

我们新增一个Java Library Module命名为 `apt_api`，编写注解类BindView。

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

这里简单介绍一下 `RetentionPolicy`。`RetentionPolicy` 是一个枚举,它的值有三种: `SOURCE`、`CLASS`、`RUNTIME`。

- `SOURCE`: 不参与编译，让开发者使用。
- `CLASS`: 参与编译，运行时不可见。给编译器使用。
- `RUNTIME`: 参与编译，运行时可见。给编译器和JVM使用。

#### 2. 定义注解处理器。

同样，我们需要新增一个Java Library Module命名为 `apt_processor`。

我们需要引入两个必要的依赖：一个是我们新增的module `apt_annotation`，另一个是google的 `com.google.auto.service:auto-service:1.0-rc3`（以下简称 `auto-service`）。

```
implementation project(':apt_api')
api 'com.google.auto.service:auto-service:1.0-rc3'
```

新增一个类 `ButterKnifeProcessor`，继承 `AbstractProcessor`。



```

@AutoService(Processor.class)
public class ButterKnifeProcessor extends AbstractProcessor {
    /**
     * 元素相关的工具类
     */
    private Elements elementUtils;
    /**
     * 文件相关的工具类
     */
    private File filer;
    /**
     * 日志相关的工具类
     */
    private Messenger messenger;
    /**
     * 类型相关工具类
     */
    private Types typeUtils;

    @Override
    public Set<String> getSupportedAnnotationTypes() {
        return Collections.singleton(BindView.class.getCanonicalName());
    }

    @Override
    public SourceVersion getSupportedSourceVersion() {
        return SourceVersion.RELEASE_7;
    }

    @Override
    public synchronized void init(ProcessingEnvironment processingEnvironment) {
        super.init(processingEnvironment);
        elementUtils = processingEnv.getElementUtils();
        filer = processingEnv.getFiler();
        messenger = processingEnv.getMessenger();
        typeUtils = processingEnv.getTypeUtils();
    }

    @Override
    public boolean process(Set<? extends TypeElement> set, RoundEnvironment roundEnv) {
        return false;
    }
}

```

auto-service 为我们简化了定义注解处理器的流程。@AutoService 就是由 auto-service 提供的，其作用是用来告诉编译器我们定义的 ButterKnifeProcessor 是一个编译期注解处理器。这样在编译时 ButterKnifeProcessor 才会被调用。

我们还重写了AbstractProcessor提供的四个方法：getSupportedAnnotationTypes、getSupportedSourceVersion、init、process。

- getSupportedAnnotationTypes 表示处理器可以处理哪些注解。这里返回的是我们之前定义的BindView。除了重写方法之外，还可用通过注解来实现。

```
@SupportedAnnotationTypes(value = {"me.zhangkuo.apt.annotation.BindView"})
```

- getSupportedSourceVersion 表示处理器可以处理的Java版本。这里我们采用最新的JDK版本就可以了。同样，我们也可以通过注解来实现。

```
@SupportedSourceVersion(value = SourceVersion.latestSupported())
```

- init 方法主要用来做一些准备工作。我们一般在这里初始化几个工具类。上述代码我们初始了与元素相关的工具类 elementUtils、与日志相关的工具类 messenger、与文件相关的 filer 以及与类型相关工具类 typeUtils。我们接下来会看到 process 主要是通过这几个类来生成代码的。



- `process` 用来完成具体的程序写代码功能。在具体介绍 `process` 之前，请允许我先推荐一个库：javapoet (<https://links.jianshu.com/go?to=https%3A%2F%2Flink.juejin.im%3Ftarget%3Dhttps%253A%252F%252Fgithub.com%252Fsquare%252Fjavapoet>)。javapoet 是由神奇的 square 公司开源的，它提供了非常人性化的api，来帮助开发者生成.java源文件。它的 `README.md` 文件为我们提供了丰富的例子，是我们学习的主要工具。



```

private Map<TypeElement, List<Element>> elementPackage = new HashMap<>();
private static final String VIEW_TYPE = "android.view.View";
private static final String VIEW_BINDER = "me.zhangkuo.apt.ViewBinding";

@Override
public boolean process(Set<? extends TypeElement> set, RoundEnvironment roundEnvironment) {
    if (set == null || set.isEmpty()) {
        return false;
    }
    elementPackage.clear();
    Set<? extends Element> bindViewElement = roundEnvironment.getElementsAnnotatedWith(BindView.class);
    //收集数据放入elementPackage中
    collectData(bindViewElement);
    //根据elementPackage中的数据生成.java代码
    generateCode();
    return true;
}

private void collectData(Set<? extends Element> elements){
    Iterator<? extends Element> iterable = elements.iterator();
    while (iterable.hasNext()) {
        Element element = iterable.next();
        TypeMirror elementTypeMirror = element.asType();
        //判断元素的类型是否是View或者是View的子类型。
        TypeMirror viewTypeMirror = elementUtils.getTypeElement(VIEW_TYPE).asType();
        if (typeUtils.isSubtype(elementTypeMirror, viewTypeMirror) || typeUtils.isSubtype(elementTypeMirror, viewTypeMirror)) {
            //找到父元素，这里认为是@BindView标记字段所在的类。
            TypeElement parent = (TypeElement) element.getEnclosingElement();
            //根据parent不同存储的List中
            List<Element> parentElements = elementPackage.get(parent);
            if (parentElements == null) {
                parentElements = new ArrayList<>();
                elementPackage.put(parent, parentElements);
            }
            parentElements.add(element);
        } else {
            throw new RuntimeException("错误处理，BindView应该标注在类型是View的类上");
        }
    }
}

private void generateCode(){
    Set<Map.Entry<TypeElement, List<Element>>> entries = elementPackage.entrySet();
    Iterator<Map.Entry<TypeElement, List<Element>>> iterator = entries.iterator();
    while (iterator.hasNext()){
        Map.Entry<TypeElement, List<Element>> entry = iterator.next();
        //类元素
        TypeElement parent = entry.getKey();
        //当前类元素下，注解了BindView的元素
        List<Element> elements = entry.getValue();
        //通过JavaPoet生成BindView的MethodSpec
        MethodSpec methodSpec = generateBindViewMethod(parent, elements);

        String packageName = getPackage(parent).getQualifiedName().toString();
        ClassName viewBinderInterface = ClassName.get(elementUtils.getTypeElement(VIEW_BINDER));
        String className = parent.getQualifiedName().toString().substring(
            packageName.length() + 1).replace('.', '$');
        ClassName bindingClassName = ClassName.get(packageName, className + "_ViewBinding");

        try {
            //生成 className_ViewBinding.java文件
            JavaFile.builder(packageName, TypeSpec.classBuilder(bindingClassName)
                .addModifiers(PUBLIC)
                .addSuperinterface(viewBinderInterface)
                .addMethod(methodSpec)
                .build())
                .writeTo(filer);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

private MethodSpec generateBindViewMethod(TypeElement parent, List<Element> elements) {
    ParameterSpec.Builder parameter = ParameterSpec.builder(ClassName.get("int"), "id");
    MethodSpec.Builder bindViewMethod = MethodSpec.methodBuilder("findViewById");
    bindViewMethod.addParameter(parameter.build());
    bindViewMethod.addModifiers(Modifier.PUBLIC);
    bindViewMethod.addStatement("$T temp = ($T)target", parent, parent);
    for (Element element : elements) {
        int id = element.getAnnotation(BindView.class).value();
        bindViewMethod.addStatement("temp.$N = temp.findViewById($L)", element.getSimpleName(), id);
    }
}

```



```
        return bindViewMethod.build();
    }
}
```

process的代码比较长，但是它的逻辑非常简单看，主要分为收集数据和生成代码两部分。我为关键的地方都加了注释，就不再详细解释了。到这里我们基本上完成了注解器的编写工作。

### 3. 使用注解

在build.gradle中引入我们定义的注解和注解处理器。

```
implementation project(':apt_api')
annotationProcessor project(":apt_processor")
```

#### 应用注解

```
public class MainActivity extends AppCompatActivity {

    @BindView(R.id.tv_content)
    TextView tvContent;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.inject(this);

        tvContent.setText("这就是ButterKnife的原理");
    }
}
```

到这里，这篇文件就结束了。什么？你还没说 ButterKnife 这个类呢。好吧，这个真的很简单，直接贴代码吧。

```
public class ButterKnife {
    static final Map<Class<?>, Constructor<? extends ViewBinding>> BINDINGS = new

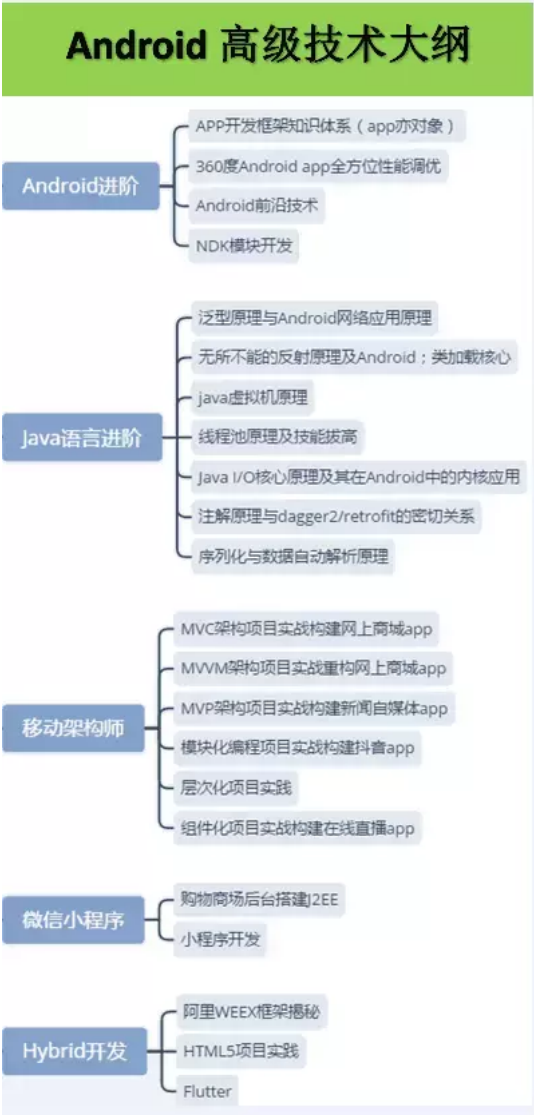
    public static void inject(Object object) {
        if (object == null) {
            return;
        }
        try {
            Class<?> cls = object.getClass();
            Constructor<? extends ViewBinding> constructor = findBindingConstructorForClass(cls);
            ViewBinding viewBinding = constructor.newInstance();
            viewBinding.bindView(object);
        } catch (Exception e) {
            // ignore
        }
    }

    private static Constructor<? extends ViewBinding> findBindingConstructorForClass(Class<?> cls) {
        Constructor<? extends ViewBinding> constructor = BINDINGS.get(cls);
        if (constructor == null) {
            String className = cls.getName();
            Class<?> bindingClass = cls.getClassLoader().loadClass(className + "_ViewBinding");
            constructor = (Constructor<? extends ViewBinding>) bindingClass.getConstructor(cls);
            BINDINGS.put(cls, constructor);
        }
        return constructor;
    }
}
```

### 【附】相关架构及资料



加群 Android IOC架构设计 ([https://links.jianshu.com/go?to=https%3A%2F%2Fjq.qq.com%2F%3F\\_wv%3D1027%26k%3D5gyv0JM](https://links.jianshu.com/go?to=https%3A%2F%2Fjq.qq.com%2F%3F_wv%3D1027%26k%3D5gyv0JM))领取获取往期Android高级架构资料、源码、笔记、视频。高级UI、性能优化、架构师课程、NDK、混合式开发（ReactNative+Weex）微信小程序、Flutter全方面的Android进阶实践技术，群内还有技术大牛一起讨论交流解决问题。



image

领取方式：

点赞+加群免费获取 Android IOC架构设计 ([https://links.jianshu.com/go?to=https%3A%2F%2Fjq.qq.com%2F%3F\\_wv%3D1027%26k%3D5gyv0JM](https://links.jianshu.com/go?to=https%3A%2F%2Fjq.qq.com%2F%3F_wv%3D1027%26k%3D5gyv0JM))

小礼物走一走，来简书关注我

赞赏支持