一、概述

在Android应用开发中，我们常常为了提升开发效率会选择使用一些基于注解的框架，但是由于反射造成一定运行效率的损耗，所以我们会更青睐于编译时注解的框架，例如：

- butterknife 免去我们编写View的初始化以及事件的注入的代码。
- EventBus3 方便我们实现组建间通讯。
- fragmentargs 轻松的为fragment添加参数信息，并提供创建方法。
- ParcelableGenerator 可实现自动将任意对象转换为Parcelable类型，方便对象传输。

类似的库还有非常多，大多这些的库都是为了自动帮我们完成日常编码中需要重复编写的部分（例如：每个Activity中的View都需要初始化，每个实现`Parcelable`接口的对象都需要编写很多固定写法的代码）。

这里并不是说上述框架就一定没有使用反射了，其实上述其中部分框架内部还是有部分实现是依赖于反射的，但是很少而且一般都做了缓存的处理，所以相对来说，效率影响很小。

但是在使用这类项目的时候，有时候出现错误会难以调试，主要原因还是很多用户并不了解这类框架其内部的原理，所以遇到问题时会消耗大量的时间去排查。

那么，于情于理，在编译时注解框架这么火的时刻，我们有理由去学习：

- 如何编写一个机遇编译时注解的项目

首先，是为了了解其原理，这样在我们使用类似框架遇到问题的时候，能够找到正确的途径去排查问题；其次，我们如果有好的想法，发现某些代码需要重复创建，我们也可以自己来写个框架方便自己日常的编码，提升编码效率；最后也算是自身技术的提升。

注：以下使用IDE为`Android Studio`.

本文将以编写一个View注入的框架为线索，详细介绍编写此类框架的步骤。

二、编写前的准备

在编写此类框架的时候，一般需要建立多个module，例如本文即将实现的例子：

- ioc-annotation 用于存放注解等，Java模块
- ioc-compiler 用于编写注解处理器，Java模块
- ioc-api 用于给用户提供使用的API，本例为Andriod模块
- ioc-sample 示例，本例为Andriod模块

那么除了示例以为，一般要建立3个module，module的名字你可以自己考虑，上述给出了一个简单的参考。当然如果条件允许的话，有的开发者喜欢将存放注解和API这两个module合并为一个module。

对于module间的依赖，因为编写注解处理器需要依赖相关注解，所以：

> ioc-compiler依赖ioc-annotation

我们在使用的过程中，会用到注解以及相关API

> 所以ioc-sample依赖ioc-api；ioc-api依赖ioc-annotation

三、注解模块的实现

注解模块，主要用于存放一些注解类，本例是模板butterknife实现View注入，所以本例只需要一个注解类：

```
@Retention(RetentionPolicy.CLASS)
@Target(ElementType.FIELD)
public @interface BindView
{
    int value();
}123456
```

我们设置的保留策略为Class，注解用于Field上。这里我们需要在使用时传入一个id，直接以value的形式进行设置即可。

你在编写的时候，分析自己需要几个注解类，并且正确的设置`@Target`以及`@Retention`即可。

四、注解处理器的实现

定义完成注解后，就可以去编写注解处理器了，这块有点复杂，但是也算是有章可循的。

该模块，我们一般会依赖注解模块，以及可以使用一个`auto-service`库

`build.gradle`的依赖情况如下：

```
dependencies {
    compile 'com.google.auto.service:auto-service:1.0-rc2'
    compile project (':ioc-annotation')
}1234
```

`auto-service`库可以帮我们去生成`META-INF`等信息。

（1）基本代码

注解处理器一般继承于`AbstractProcessor`，刚才我们说有章可循，是因为部分代码的写法基本是固定的，如下：

```
@AutoService(Processor.class)
public class IocProcessor extends AbstractProcessor{
    private Filer mFileUtils;
    private Elements mElementUtils;
    private Messager mMessager;
    @Override
    public synchronized void init(ProcessingEnvironment processingEnv){
        super.init(processingEnv);
        mFileUtils = processingEnv.getFiler();
        mElementUtils = processingEnv.getElementUtils();
        mMessager = processingEnv.getMessager();
    }
    @Override
    public Set<String> getSupportedAnnotationTypes(){
        Set<String> annotationTypes = new LinkedHashSet<String>();
        annotationTypes.add(BindView.class.getCanonicalName());
        return annotationTypes;
    }
```

```
    @Override
    public SourceVersion getSupportedSourceVersion(){
        return SourceVersion.latestSupported();
    }
    @Override
    public boolean process(Set<? extends TypeElement> annotations,
RoundEnvironment roundEnv){
    }12345678910111213141516171819202122232425
```

在实现`AbstractProcessor`后，`process()`方法是必须实现的，也是我们编写代码的核心部分，后面会介绍。

我们一般会实现`getSupportedAnnotationTypes()`和`getSupportedSourceVersion()`两个方法，这两个方法一个返回支持的注解类型，一个返回支持的源码版本，参考上面的代码，写法基本是固定的。

除此以外，我们还会选择复写`init()`方法，该方法传入一个参数`processingEnv`,可以帮助我们去初始化一些父类类：

- Filer mFileUtils; 跟文件相关的辅助类，生成JavaSourceCode.
- Elements mElementUtils;跟元素相关的辅助类，帮助我们去获取一些元素相关的信息。
- Messager mMessager;跟日志相关的辅助类。

这里简单提一下`Elemnet`，我们简单认识下它的几个子类，根据下面的注释，应该已经有了一个简单认知。

```
Element
  - VariableElement //一般代表成员变量
  - ExecutableElement //一般代表类中的方法
  - TypeElement //一般代表代表类
  - PackageElement //一般代表Package12345
```

（2）process的实现

process中的实现，相比较会比较复杂一点，一般你可以认为两个大步骤：

- 收集信息
- 生成代理类（本文把编译时生成的类叫代理类）

什么叫收集信息呢？就是根据你的注解声明，拿到对应的Element，然后获取到我们所需要的信息，这个信息肯定是为了后面生成`JavaFileObject`所准备的。

例如本例，我们会针对每一个类生成一个代理类，例如`MainActivity`我们会生成一个`MainActivity$$ViewInjector`。那么如果多个类中声明了注解，就对应了多个类，这里就需要：

- 一个类对象，代表具体某个类的代理类生成的全部信息，本例中为`ProxyInfo`
- 一个集合，存放上述类对象（到时候遍历生成代理类），本例中为`Map<String, ProxyInfo>`，key为类的全路径。

这里的描述有点模糊没关系，一会结合代码就好理解了。

a.收集信息

```
private Map<String, ProxyInfo> mProxyMap = new HashMap<String, ProxyInfo>();
@Override
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
roundEnv){
    mProxyMap.clear();
```

```
    Set<? extends Element> elements =
roundEnv.getElementsAnnotatedWith(BindView.class);
    //一、收集信息
    for (Element element : elements){
        //检查element类型
        if (!checkAnnotationUseValid(element)){
            return false;
        }
        //field type
        VariableElement variableElement = (VariableElement) element;
        //class type
        TypeElement typeElement = (TypeElement)
variableElement.getEnclosingElement();//TypeElement
        String qualifiedName = typeElement.getQualifiedName().toString();

        ProxyInfo proxyInfo = mProxyMap.get(qualifiedName);
        if (proxyInfo == null){
            proxyInfo = new ProxyInfo(mElementUtils, typeElement);
            mProxyMap.put(qualifiedName, proxyInfo);
        }
        BindView annotation = variableElement.getAnnotation(BindView.class);
        int id = annotation.value();
        proxyInfo.mInjectElements.put(id, variableElement);
    }
    return true;
}
12345678910111213141516171819202122232425262728 29
```

首先我们调用一下`mProxyMap.clear();`，因为process可能会多次调用，避免生成重复的代理类，避免生成类的类名已存在异常。

然后，通过`roundEnv.getElementsAnnotatedWith`拿到我们通过`@BindView`注解的元素，这里返回值，按照我们的预期应该是`VariableElement`集合，因为我们用于成员变量上。

接下来for循环我们的元素，首先检查类型是否是`VariableElement`.

然后拿到对应的类信息`TypeElement`，继而生成`ProxyInfo`对象，这里通过一个`mProxyMap`进行检查，key为`qualifiedName`即类的全路径，如果没有生成才会去生成一个新的，`ProxyInfo`与类是一一对应的。

接下来，会将与该类对应的且被`@BindView`声明的`VariableElement`加入到`ProxyInfo`中去，key为我们声明时填写的id，即View的id。

这样就完成了信息的收集，收集完成信息后，应该就可以去生成代理类了。

b.生成代理类

```
@Override
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
roundEnv){
    //...省略收集信息的代码，以及try,catch相关
    for(String key : mProxyMap.keySet()){
        ProxyInfo proxyInfo = mProxyMap.get(key);
        JavaFileObject sourceFile = mFileUtils.createSourceFile(
                proxyInfo.getProxyClassFullName(), proxyInfo.getTypeElement());
            Writer writer = sourceFile.openWriter();
            writer.write(proxyInfo.generateJavaCode());
            writer.flush();
```

```
            writer.close();
        }
        return true;
}1234567891011121314
```

可以看到生成代理类的代码非常的简短，主要就是遍历我们的`mProxyMap`，然后取得每一个`ProxyInfo`，最后通过`mFileUtils.createSourceFile`来创建文件对象，类名为`proxyInfo.getProxyClassFullName()`，写入的内容为`proxyInfo.generateJavaCode()`.

看来生成Java代码的方法都在ProxyInfo里面。

c.生成Java代码

这里我们主要关注其生成Java代码的方式。
下面主要看生成Java代码的方法：

```
#ProxyInfo
//key为id，value为对应的成员变量
public Map<Integer, VariableElement> mInjectElements = new HashMap<Integer,
VariableElement>();

public String generateJavaCode(){
    StringBuilder builder = new StringBuilder();
    builder.append("package " + mPackageName).append(";\n\n");
    builder.append("import com.zhy.ioc.*;\n");
    builder.append("public class ").append(mProxyClassName).append(" implements
" + SUFFIX + "<" + mTypeElement.getQualifiedName() + ">");
    builder.append("\n{\n");
    generateMethod(builder);
    builder.append("\n}\n");
    return builder.toString();
}
private void generateMethod(StringBuilder builder){
     builder.append("public void inject("+mTypeElement.getQualifiedName()+" host
, Object object )");
    builder.append("\n{\n");
    for(int id : mInjectElements.keySet()){
        VariableElement variableElement = mInjectElements.get(id);
        String name = variableElement.getSimpleName().toString();
        String type = variableElement.asType().toString() ;

        builder.append(" if(object instanceof android.app.Activity)");
        builder.append("\n{\n");
        builder.append("host."+name).append(" = ");
        builder.append("("+type+")
(((android.app.Activity)object).findViewById("+id+"));");
        builder.append("\n}\n").append("else").append("\n{\n");
        builder.append("host."+name).append(" = ");
        builder.append("("+type+")
(((android.view.View)object).findViewById("+id+"));");
        builder.append("\n}\n");
    }
    builder.append("\n}\n");
}123456789101112131415161718192021222324252627282930313233
```

这里主要就是靠收集到的信息，拼接完成的代理类对象了，看起来会比较头疼，不过我给出一个生成后的代码，对比着看会很多。

```
package com.zhy.ioc_sample;
import com.zhy.ioc.*;
public class MainActivity$$ViewInjector implements
ViewInjector<com.zhy.ioc_sample.MainActivity>{
    @Override
    public void inject(com.zhy.sample.MainActivity host , Object object ){
        if(object instanceof android.app.Activity){
            host.mTv = (android.widget.TextView)
(((android.app.Activity)object).findViewById(2131492945));
        }
        else{
            host.mTv = (android.widget.TextView)
(((android.view.View)object).findViewById(2131492945));
        }
    }
}12345678910111213
```

这样对着上面代码看会好很多，其实就死根据收集到的成员变量（通过`@BindView`声明的），然后根据我们具体要实现的需求去生成java代码。

这里注意下，生成的代码实现了一个接口`ViewInjector<T>`，该接口是为了统一所有的代理类对象的类型，到时候我们需要强转代理类对象为该接口类型，调用其方法；接口是泛型，主要就是传入实际类对象，例如`MainActivity`，因为我们在生成代理类中的代码，实际上就是`实际类.成员变量`的方式进行访问，所以，使用编译时注解的成员变量一般都不允许`private`修饰符修饰（有的允许，但是需要提供getter,setter访问方法）。

这里采用了完全拼接的方式编写Java代码，你也可以使用一些开源库，来通过Java api的方式来生成代码，例如：

- javapoet.

> A Java API for generating .java source files.

到这里我们就完成了代理类的生成，这里任何的注解处理器的编写方式基本都遵循着收集信息、生成代理类的步骤。

五、API模块的实现

有了代理类之后，我们一般还会提供API供用户去访问，例如本例的访问入口是

```
//Activity中
Ioc.inject(Activity);
//Fragment中，获取ViewHolder中
Ioc.inject(this, view);1234
```

模仿了butterknife，第一个参数为宿主对象，第二个参数为实际调用`findViewById`的对象；当然在Actiivty中，两个参数就一样了。

- API一般如何编写呢？

其实很简单，只要你了解了其原理，这个API就干两件事：

- 根据传入的host寻找我们生成的代理类：例如`MainActivity->MainActity$$ViewInjector`。
- 强转为统一的接口，调用接口提供的方法。

这两件事应该不复杂，第一件事是拼接代理类名，然后反射生成对象，第二件事强转调用。

```
public class Ioc{
    public static void inject(Activity activity){
        inject(activity , activity);
    }
    public static void inject(Object host , Object root){
        Class<?> clazz = host.getClass();
        String proxyClassFullName = clazz.getName()+"$$ViewInjector";
       //省略try,catch相关代码
        Class<?> proxyClazz = Class.forName(proxyClassFullName);
        ViewInjector viewInjector = (com.zhy.ioc.ViewInjector)
proxyClazz.newInstance();
        viewInjector.inject(host,root);
    }
}
public interface ViewInjector<T>{
    void inject(T t , Object object);
}
```

```
12345678910111213141516 17
```

代码很简单，拼接代理类的全路径，然后通过`newInstance`生成实例，然后强转，调用代理类的inject方法。

这里一般情况会对生成的代理类做一下缓存处理，比如使用`Map`存储下，没有再生成，这里我们就不去做了。

这样我们就完成了一个编译时注解框架的编写。

六、总结

本文通过具体的实例来描述了如何编写一个基于编译时注解的项目，主要步骤为：项目结构的划分、注解模块的实现、注解处理器的编写以及对外公布的API模块的编写。通过文本的学习应该能够了解基于编译时注解这类框架运行的原理，以及自己如何去编写这样一类框架。