# 注解

# 理解Java注解

注解就相当于对源代码打的标签，给代码打上标签和删除标签对源代码没有任何影响。有的人要说了，你尽几把瞎扯，没有影响，打这些标签干毛线呢？其实不是这些标签自己起了什么作用，而且外部工具通过访问这些标签，然后根据不同的标签做出了相应的处理。这是注解的精髓，理解了这一点一切就变得不再那么神秘。

例如我们写代码用的IDE（例如 IntelliJ Idea）,它检查发现某一个方法上面有@Deprecated这个注解，它就会在所有调用这个方法的地方将这个方法标记为删除。访问和处理Annotation的工具统称为APT(Annotation Processing Tool)

# 获取类字段两种方式

关于获取类的字段有两种方式：getFields()和getDeclaredFields()。

getFields()：获得某个类的所有的公共（public）的字段，包括父类中的字段。

getDeclaredFields()：获得某个类的所有声明的字段，即包括public、private和proteced，但是不包括父类的申明字段。

同样类似的还有getConstructors()和getDeclaredConstructors()、getMethods()和getDeclaredMethods()，这两者分别表示获取某个类的方法、构造函数

# 注解可以分为以下3类

## 基本注解

Java内置的注解共有5个

@Override：让编译器检查被标记的方法，保证其重写了父类的某一个方法。此注解只能标记方法。源码如下：

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.SOURCE)  
public @interface Override {  
}

@Deprecated：标记某些程序元素已经过时，程序员请不要再使用了。源码如下：

@Documented

@Retention(RetentionPolicy.RUNTIME)  
@Target(value={CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})  
public @interface Deprecated {  
}

@SuppressWarnings ：告诉编译器不要给老子显示警告，老子不想看，老子清楚的知道自己在干什么。源码如下：

@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL\_VARIABLE})

@Retention(RetentionPolicy.SOURCE)  
public @interface SuppressWarnings {  
 String[] value();  
}

其内部有一个String数组，根据传入的值来取消相应的警告：

deprecation：使用了不赞成使用的类或方法时的警告；

unchecked：执行了未检查的转换时的警告，例如当使用集合时没有用泛型 (Generics) 来指定集合保存的类型;

fallthrough：当 Switch 程序块直接通往下一种情况而没有 Break 时的警告;

path：在类路径、源文件路径等中有不存在的路径时的警告;

serial：当在可序列化的类上缺少 serialVersionUID 定义时的警告;

finally：任何 finally 子句不能正常完成时的警告;

all：关于以上所有情况的警告。

@SafeVarargs(Java7 新增) ：@SuppressWarnings可以用在各种需要取消警告的地方，而 @SafeVarargs主要用在取消参数的警告。就是说编译器如果检查到你对方法参数的操作，有可能发生问题时会给出警告，但是你很自（任）性，老子不要警告，于是你就加上了这个标签。源码如下：

@Documented

@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})  
public @interface SafeVarargs {}

其实这个注解是专为取消堆污染警告设置的，因为Java7会对可能产生堆污染的代码提出警告，什么是堆污染？且看下面代码

@SafeVarargs

private static void method(List<String>... strLists) {  
 List[] array = strLists;  
 List<Integer> tmpList = Arrays.asList(42);  
 array[0] = tmpList; //非法操作，但是没有警告  
 String s = strLists[0].get(0); //ClassCastException at runtime!  
}

如果不使用 @SafeVarargs，这个方法在编译时候是会产生警告的 ： “…使用了未经检查或不安全的操作。”,用了就不会有警告，但是在运行时会抛异常。

@FunctionalInterface(Java8 新增)： 标记型注解，告诉编译器检查被标注的接口是否是一个函数接口，即检查这个接口是否只包含一个抽象方法，只有函数接口才可以使用Lambda表达式创建实例。源码如下：

@Documented

@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface FunctionalInterface {}

## 元注解

用来给其他注解打标签的注解，即用来注解其他注解的注解。元注解共有6个。从上面的基本注解的源代码中就会看到使用了元注解来注解自己。

@Retention：用于指定被此元注解标注的注解的保留时长，源代码如下：

@Documented

@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.ANNOTATION\_TYPE)  
public @interface Retention {  
 RetentionPolicy value();  
}

从源代码中可以看出，其有一个属性value,返回一个枚举RetentionPolicy 类型，有3种类型：

RetentionPolicy.SOURCE: ：注解信息只保留在源代码中，编译器编译源码时会将其直接丢弃。

RetentionPolicy.CLASS:：注解信息保留在class文件中，但是虚拟机VM不会持有其信息。

RetentionPolicy.RUNTIME:：注解信息保留在class文件中，而且VM也会持有此注解信息，所以可以通过反射的方式获得注解信息。

@Target：用于指定被此元注解标注的注解可以标注的程序元素，源码如下：

@Documented

@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.ANNOTATION\_TYPE)  
public @interface Target {  
 ElementType[] value();  
}

从源码中可以看出，其有一个属性value,返回一个枚举ElementType类型的数组，这个数组的值就代表了可以使用的程序元素。

public enum ElementType {

/\*\*标明该注解可以用于类、接口（包括注解类型）或enum声明\*/  
 TYPE,  
  
 /\*\* 标明该注解可以用于字段(域)声明，包括enum实例 \*/  
 FIELD,  
  
 /\*\* 标明该注解可以用于方法声明 \*/  
 METHOD,  
  
 /\*\* 标明该注解可以用于参数声明 \*/  
 PARAMETER,  
  
 /\*\* 标明注解可以用于构造函数声明 \*/  
 CONSTRUCTOR,  
  
 /\*\* 标明注解可以用于局部变量声明 \*/  
 LOCAL\_VARIABLE,  
  
 /\*\* 标明注解可以用于注解声明(应用于另一个注解上)\*/  
 ANNOTATION\_TYPE,  
  
 /\*\* 标明注解可以用于包声明 \*/  
 PACKAGE,  
  
 /\*\*  
 \* 标明注解可以用于类型参数声明（1.8新加入）  
 \*/  
 TYPE\_PARAMETER,  
  
 /\*\*  
 \* 类型使用声明（1.8新加入)  
 \*/  
 TYPE\_USE  
}

例如@Override注解使用了 @Target(ElementType.METHOD)，那么就意味着，它只能注解方法，不能注解其他程序元素。

当注解未指定Target值时，则此注解可以用于任何元素之上，多个值使用{}包含并用逗号隔开，下面代码表示，此Annotation既可以注解构造函数、字段和方法：

@Target(value={CONSTRUCTOR, FIELD, METHOD})

值得注意的是，TYPE\_PARAMETER，TYPE\_USE是Java8 加入的新类型，在Java8之前，只能在声明各种程序元素时使用注解，而TYPE\_PARAMETER允许使用注解修饰参数类型，TYPE\_USE允许使用注解修饰任意类型。

//TYPE\_PARAMETER 修饰类型参数

class A<@Parameter T> { }  
  
//TYPE\_USE则可以用于标注任意类型(不包括class)  
  
//用于父类或者接口  
class Image implements @Rectangular Shape { }  
  
//用于构造函数  
new @Path String("/usr/bin")  
  
//用于强制转换和instanceof检查,注意这些注解中用于外部工具，它们不会对类型转换或者instanceof的检查行为带来任何影响。  
String path=(@Path String)input;  
if(input instanceof @Path String)  
  
//用于指定异常  
public Person read() throws @Localized IOException.  
  
//用于通配符绑定  
List<@ReadOnly ? extends Person>  
List<? extends @ReadOnly Person>  
  
@NotNull String.class //非法，不能标注class  
import java.lang.@NotNull String //非法，不能标注import

虽然Java8 提供了类型注解，但是没有提供APT,所以需要框架自己实现。

@Documented：将被标注的注解生成到javadoc中。

@Inherited：其让被修饰的注解拥有被继承的能力。如下，我们有一个用@Inherited修饰的注解@InAnnotation，那么这个注解就拥有了被继承的能力。

@Inherited

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface InAnnotation{  
}  
  
@InAnnotation  
class Base{}  
  
class Son extends Base{}

当使用此注解修饰一个基类Base, 其子类Son 并没有使用任何注解修饰，但是其已经拥有了@InAnnotation这个注解，相当于Son 已经被@InAnnotation修饰了

@Repeatable ：使被修饰的注解可以重复的注解某一个程序元素。例如下面的代码中@ShuSheng这个自定义注解使用了@Repeatable修饰，所以其可以按照下面的语法重复的注解一个类。

```Java

@ShuSheng(name="frank",age=18)

@ShuSheng(age = 20)

public class AnnotationDemo{}

```

如何定义一个重复注解呢,如下所示，我们需要先定义一个容器，例如ShuShengs ，然后将其作为参数传入@Repeatable中。

@Repeatable(ShuShengs.class)

@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface ShuSheng {  
 String name() default "ben";  
 int age();  
}  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface ShuShengs {  
 ShuSheng[] value();  
}

# 运行时注解

## 自定义运行时注解

运行时注解：在代码运行的过程中通过反射机制找到我们自定义的注解，然后做相应的事情。

反射：对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性。

自定义运行是注解大的方面分为两步：一个是申明注解、第二个是解析注解。

## 申明注解

申明注解步骤：

通过@Retention(RetentionPolicy.RUNTIME)元注解确定我们注解是在运行的时候使用。

通过@Target确定我们注解是作用在什么上面的(变量、函数、类等)。

确定我们注解需要的参数。

比如下面一段代码我们声明了一个作用在变量上的BindString运行时注解。

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.FIELD)  
public @interface BindString {  
  
 int value();  
  
}

## 注解解析

运行时注解的解析我们简单的分为三个步骤：

找到类对应的所有属性或者方法(至于是找类的属性还是方法就要看我自定义的注解是定义方法上还是属性上了)。

找到添加了我们注解的属性或者方法。

做我们注解需要自定义的一些操作。

1. 获取类的属性和方法

既然注解是我们自定义的，我肯定事先会确定我们注解是加在属性上的还是加在方法上的。

通过Class对象我们就可以很容易的获取到当前类里面所有的方法和属性了：

Class类里面常用方法介绍(这里我们不仅仅介绍了获取属性和方法的，还介绍了一些其他Class里面常用的方法)

/\*\*

\* 包名加类名  
 \*/  
public String getName();  
  
/\*\*  
 \* 类名  
 \*/  
public String getSimpleName();  
  
/\*\*  
 \* 返回当前类和父类层次的public构造方法  
 \*/  
public Constructor<?>[] getConstructors();  
  
/\*\*  
 \* 返回当前类所有的构造方法(public、private和protected)  
 \* 不包括父类  
 \*/  
public Constructor<?>[] getDeclaredConstructors();  
  
/\*\*  
 \* 返回当前类所有public的字段，包括父类  
 \*/  
public Field[] getFields();  
  
/\*\*  
 \* 返回当前类所有申明的字段，即包括public、private和protected，  
 \* 不包括父类  
 \*/  
public native Field[] getDeclaredFields();  
  
/\*\*  
 \* 返回当前类所有public的方法，包括父类  
 \*/  
public Method[] getMethods();  
  
/\*\*  
 \* 返回当前类所有的方法，即包括public、private和protected，  
 \* 不包括父类  
 \*/  
public Method[] getDeclaredMethods();  
  
/\*\*  
 \* 获取局部或匿名内部类在定义时所在的方法  
 \*/  
public Method getEnclosingMethod();  
  
/\*\*  
 \* 获取当前类的包  
 \*/  
public Package getPackage();  
  
/\*\*  
 \* 获取当前类的包名  
 \*/  
public String getPackageName$();  
  
/\*\*  
 \* 获取当前类的直接超类的 Type  
 \*/  
public Type getGenericSuperclass();  
  
/\*\*  
 \* 返回当前类直接实现的接口.不包含泛型参数信息  
 \*/  
public Class<?>[] getInterfaces();  
  
/\*\*  
 \* 返回当前类的修饰符，public,private,protected  
 \*/  
public int getModifiers();

类里面每个属性对应一个对象Field，每个方法对应一个对象Method。

2. 找到添加注解的属性或者方法

上面说道每个属性对应Field，每个方法对应Method。而且Field和Method都实现了AnnotatedElement接口。都有AnnotatedElement接了我们就可以很容易的找到添加了我们指定注解的方法或者属性了。

AnnotatedElement接口常用方法如下:

/\*\*

\* 指定类型的注释是否存在于此元素上  
 \*/  
default boolean isAnnotationPresent(Class<? extends Annotation> annotationClass) {  
 return getAnnotation(annotationClass) != null;  
}  
  
/\*\*  
 \* 返回该元素上存在的指定类型的注解  
 \*/  
<T extends Annotation> T getAnnotation(Class<T> annotationClass);  
  
/\*\*  
 \* 返回该元素上存在的所有注解  
 \*/  
Annotation[] getAnnotations();  
  
/\*\*  
 \* 返回该元素指定类型的注解  
 \*/  
default <T extends Annotation> T[] getAnnotationsByType(Class<T> annotationClass) {  
 return AnnotatedElements.getDirectOrIndirectAnnotationsByType(this, annotationClass);  
}  
  
/\*\*  
 \* 返回直接存在与该元素上的所有注释(父类里面的不算)  
 \*/  
default <T extends Annotation> T getDeclaredAnnotation(Class<T> annotationClass) {  
 Objects.requireNonNull(annotationClass);  
 // Loop over all directly-present annotations looking for a matching one  
 for (Annotation annotation : getDeclaredAnnotations()) {  
 if (annotationClass.equals(annotation.annotationType())) {  
 // More robust to do a dynamic cast at runtime instead  
 // of compile-time only.  
 return annotationClass.cast(annotation);  
 }  
 }  
 return null;  
}  
  
/\*\*  
 \* 返回直接存在该元素岸上某类型的注释  
 \*/  
default <T extends Annotation> T[] getDeclaredAnnotationsByType(Class<T> annotationClass) {  
 return AnnotatedElements.getDirectOrIndirectAnnotationsByType(this, annotationClass);  
}  
  
/\*\*  
 \* 返回直接存在与该元素上的所有注释  
 \*/  
Annotation[] getDeclaredAnnotations();

3. 做自定义注解需要做的事情

添加了我们注解的属性或者方法已经拿到了，之后要做的就是自定义注解自定义的一些事情了。比如在某些特定条件下自动去执行我们添加注解的方法。下面我们也会用两个具体的实例来说明。

# 编译时注解

## 1.定义注解

在工程中添加一个java library类型的module，取名annotation

Android Studio -> file -> new module -> java library

定义三个注解

BindView：其实不用做过多的解释，用过ButterKnife都知道这是干啥的。其作用就是将XML layout 文件中的view映射到代码中

@Retention(RetentionPolicy.SOURCE)

@Target(ElementType.FIELD)  
public @interface BindView {  
 @IdRes int value();  
}

OnClick : 映射view的一个click事件，当一个view被点击后，此注解标记的对应方法就会触发。

@Retention(RetentionPolicy.SOURCE)

@Target(ElementType.METHOD)  
public @interface OnClick {  
 @IdRes int value();  
}

Keep ：这个就比较有意思了，我们准备使用此注解告诉编译器，不要混淆被此注解标记的类。由于混淆的存在程序中使用反射的类都不应该被混淆，不然运行时就找不到了。

@Retention(RetentionPolicy.CLASS)

@Target(ElementType.TYPE)  
public @interface Keep {  
}

## 创建Processor项目

新建一个module，取名为compiler，类型必须为java library，如下所示

Android Studio -> file -> new module -> java library

## 注册Processor

### 手动注册

新建一个类Processor，这类必须继承至 AbstractProcessor

必须严格按如下图所示的命名方式创建一个文件javax.annotation.processing.Processor，文件的内容为你定义的注解处理器的全类名，例如此处就是Processor的全类名：top.ss007.compiler.Processor。

### 自动注册

可以使用google的 AutoService，使用@AutoService标记你的注解处理器，这个库就会为你产生这个注册目录了。

Processor 一般会重写父类的4个方法：

init： 初始化工作，我们可以得到一些有用的工具，例如 Filer，我们需要它将生成的代码写入文件中

process： 最重要的方法，所有的注解处理都是在此完成

getSupportedAnnotationTypes： 返回我们所要处理的注解的一个集合

getSupportedSourceVersion： 要支持的java版本

下面是processor的代码片段

public class Processor extends AbstractProcessor {

private Filer filer;  
 private Messager messager;  
 private Elements elementUtils;  
  
 //每个存在注解的类整理出来，key:package\_classname value:被注解的类型元素  
 private Map<String, List<Element>> annotationClassMap = new HashMap<>();  
  
 @Override  
 public synchronized void init(ProcessingEnvironment processingEnv) {  
 super.init(processingEnv);  
 filer = processingEnv.getFiler();  
 messager = processingEnv.getMessager();  
 elementUtils = processingEnv.getElementUtils();  
 }  
  
 @Override  
 public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {   
 if (!roundEnv.processingOver()) {  
 ...  
 for (Map.Entry<String, List<Element>> entry : annotationClassMap.entrySet()) {  
 /\*  
 创建要生成的类，如下所示  
 @Keep  
 public class MainActivity$Binding {}\*/  
 TypeSpec.Builder classBuilder = TypeSpec.classBuilder(generatedClassName)  
 .addModifiers(Modifier.PUBLIC)  
 .addAnnotation(Keep.class);  
 ...  
  
 //将类写入文件中  
 try {  
 JavaFile.builder(packageName,  
 classBuilder.build())  
 .build()  
 .writeTo(filer);  
 } catch (IOException e) {  
 messager.printMessage(Diagnostic.Kind.ERROR, e.toString());  
 }  
 }  
 }  
 return true;  
 }  
  
  
 @Override  
 public Set<String> getSupportedAnnotationTypes() {  
 return new TreeSet<>(Arrays.asList(  
 BindView.class.getCanonicalName(),  
 OnClick.class.getCanonicalName(),  
 Keep.class.getCanonicalName()));  
 }  
  
 @Override  
 public SourceVersion getSupportedSourceVersion() {  
 return SourceVersion.latestSupported();  
 }  
...  
}

上面的代码也没有什么神奇的，就是在process方法里面处理我们定义的注解，然后使用javapoet 生成相应的代码，而这些代码本应该是我们程序员自己手写的，现在自动生成了,具体的看源码吧，文章最后贴出了源代码的地址。

## 定义初始化模块

代码是生成了，那我们怎么使用呢，注解生成器的代码需要我们主动去触发，还记得ButterKnife中那句ButterKnife.bind(this); 吗，就是干这个事情的

新建一个module，取名为binder，类型必须为android library，如下所示

Android Studio -> file -> new module -> android library

定义一个Binding类

public class Binding {

private Binding(){}  
  
 private static <T extends Activity> void instantiateBinder(T target,String suffix){  
 Class<?> targetClass=target.getClass();  
 String className=targetClass.getName();  
 try {  
 Class<?>bindingClass =targetClass  
 .getClassLoader()  
 .loadClass(className+suffix);  
 Constructor<?> classConstructor=bindingClass.getConstructor(targetClass);  
 try {  
 classConstructor.newInstance(target);  
 } catch (IllegalAccessException e) {  
 throw new RuntimeException("Unable to invoke " + classConstructor, e);  
 }   
 ...  
 }  
  
 public static <T extends Activity> void bind(T activity) {  
 instantiateBinder(activity, BindingSuffix.GENERATED\_CLASS\_SUFFIX);  
 }  
}

我们通过反射调用生成类的构造方法，而所有的绑定逻辑都在那个类的构造方法里面，如下所示

public MainActivity$Binding(MainActivity activity) {

bindViews(activity);  
 bindOnClicks(activity);  
}

是否还记得前面我们使用@Keep标记了生成的类，它的作用就是防止此类被混淆，确保反射调用可以顺利进行。那keep是怎么起作用的呢？

在此module的 proguard-rules.pro 文件中添加如下代码。

-keep class top.ss007.annotation.Keep

-keep @top.ss007.annotation.Keep public class \*  
-keepclassmembers @top.ss007.annotation.Keep class \*\* { \*;}

然后在其build.gradle 文件中添加如下代码

apply plugin: 'com.android.library'

android {  
 defaultConfig {  
 ...  
 consumerProguardFiles 'proguard-rules.pro'  
 }

上面的代码表示使用此module的 android application 都要使用 proguard-rules.pro 文件中定义的规则，我们就是在这个文件中申明不要混淆带有@Keep标记的类的。

# 5.使用

这个就比较简单了，毕竟我们使用ButterKnife那么多年了。在app的build.gradle 文件中添加

dependencies {

...  
 implementation project(':binder')  
 annotationProcessor project(':compiler')  
 ...

在Activity中绑定即可

# APT生成文件路径

高版本的gradle生成文件路径（gradle-6.6.1-all):

build/generated/ap\_generated\_sources/debug/out/packagename/

较低版本的gradle生成文件路径（gradle-5.1.1-all):

build/generated/source/apt/

# 遇到问题

1.apt生成文件失败

在调试的时候发现生成文件失败，程序已运行就报找不到对应的文件，把gradle版本和gradle插件版本都修改和demo一样，还是有问题，

调试发现自定义AbstractProcessor的init()方法会执行，但是process()方法不会执行，后来发现是定义的注解和运行时注解重名了，

CompileActivity用到的是运行时注解，AbstractProcessor使用的是编译时注解，改成编译时注解就成功生成文件了。

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5