Android 自定义注解(Annotation)_旧路-CSDN博客_android 自定义注解

笔记本:	程序员		
创建时间:	2020/8/20 星期四 16:06	**更新时间:**	2020/8/20 星期四 16:10
标签:	Android, 编程		
URL:	https://blog.csdn.net/wuyuxing24/article/details/81139846		

Android 自定义注解(Annotation)

现在市面上很多框架都有使用到注解，比如butterknife库、EventBus库、Retrofit库等等。也是一直好奇他们都是怎么做到的啥。咱们能不能自己去实现一个简单的注解呢。

注解(Annotation)是JDK1.5新增加功能，注解其实就是添加在类、变量、方法、参数等前面的一个修饰符一个标记而已(不要把如下面的代码里面@Override、@IdRes就是注解。

```
@Override
public <T extends View> T findViewById(@IdRes int id) {
return getDelegate().findViewById(id);
}
```

上面我们强调了注解就是一个修饰符一个标记而且。但是通过注解能做的事情确是无穷。在代码编译或者运行的过程中我们可找到这些注解之后咱们就可以做很多事情了，比如自动做一些代码处理(赋值、检测、调用等等)或者干脆生成一些额外的java文件等的实例来说明。

注解的作用:简化代码，提高开发效率。

> 注意哦，肯定是能提高代码开发效率，并不一定能提供程序运行效率。

接下来我们通过学习自定义注解(定义我们自己的注解)来让大家对注解有一个深刻的认识。

一、元注解

在我们自定义注解之前我们需要来先了解下元注解。元注解是用来定义其他注解的注解(在自定义注解的时候，需要使用到元注解)。java.lang.annotation提供了四种元注解：@Retention、 @Target、@Inherited、@Documented。

> 元注解是用来修饰注解的注解。在自定义注解的时候我们肯定都是要用到元注解的。因为我们需要定义我们注解的是方法还是变量，注解的存活时间等等。

元注解	说明
@Target	表明我们注解可以出现的地方。是一个ElementType枚举
@Retention	这个注解的的存活时间
@Document	表明注解可以被javadoc此类的工具文档化
@Inherited	是否允许子类继承该注解，默认为false

1.1、@Target

@Target元注解用来表明我们注解可以出现的地方，参数是一个ElementType类型的数组，所以@Target可以设置注解同时出现既可以出现来类的前面也可以出现在变量的前面。

@Target元注解ElementType枚举(用来指定注解可以出现的地方):

@Target-ElementType类型	说明
ElementType.TYPE	接口、类、枚举、注解
ElementType.FIELD	字段、枚举的常量
ElementType.METHOD	方法
ElementType.PARAMETER	方法参数
ElementType.CONSTRUCTOR	构造函数
ElementType.LOCAL_VARIABLE	局部变量
ElementType.ANNOTATION_TYPE	注解
ElementType.PACKAGE	包

1.2、@Retention

@Retention表示需要在什么级别保存该注释信息，用于描述注解的生命周期(即：被描述的注解在什么范围内有效)。参数是R对象。

RetentionPolicy的枚举类型有(默认值为CLASS.)：

@Retention-RetentionPolicy类型	说明
RetentionPolicy.SOURCE	注解只保留在源文件，当Java文件编译成class文件的时候，注解被遗弃
RetentionPolicy.CLASS	注解被保留到class文件，但jvm加载class文件时候被遗弃，这是默认的生命周期
RetentionPolicy.RUNTIME	注解不仅被保存到class文件中，jvm加载class文件之后，仍然存在

> SOURCE < CLASS < RUNTIME,前者能作用的地方后者一定也能作用.

1.3、@Document

@Document表明我们标记的注解可以被javadoc此类的工具文档化。

1.4、@Inherited

@Inherited表明我们标记的注解是被继承的。比如，如果一个父类使用了@Inherited修饰的注解，则允许子类继承该父类的注

二、自定义注解

2.1、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。

2.1.1、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();

}
```

2.1.2、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。

2.1.2.1、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。

2.1.2.2、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Annotat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();
```

2.1.2.3、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方法。下面我们也会用两个具体的实例来说明。

2.1.3、运行时注解实例

我们通过两个简单的实例来看下自定义运行时注解是怎么操作的。

2.1.3.1、通过注解自动创建对象

代码过程中，我们可能经常会犯这样的错误，定义了一个对象，但是经常忘了创建对象。跑出空指针异常。接下来我们通过自注解来自动去帮我们创建对象。

> AutoWired注解的声，指定注解是在变量上使用，并且在运行时有效。

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface AutoWired {

}
```

> AutoWired注解的解析，找到AutoWired注解的变量，创建对象，在吧对象赋值给AutoWired指定的那个变量。

```
public class AutoWiredProcess {

public static void bind(final Object object) {
Class parentClass = object.getClass();
Field[] fields = parentClass.getFields();
for (final Field field : fields) {
AutoWired autoWiredAnnotation = field.getAnnotation(AutoWired.class);
```

```
if (autoWiredAnnotation != null) {
field.setAccessible(true);
try {
Class<?> autoCreateClass = field.getType();
Constructor autoCreateConstructor = autoCreateClass.getConstructor();
field.set(object, autoCreateConstructor.newInstance());
} catch (NoSuchMethodException e) {
e.printStackTrace();
} catch (InstantiationException e) {
e.printStackTrace();
} catch (IllegalAccessException e) {
e.printStackTrace();
} catch (InvocationTargetException e) {
e.printStackTrace();
}

}
}
}

}
```

> AutoWired注解的使用，在onCrate()方法里面调用了AutoWiredProcess.bind(this);来解析注解。这样在运行的时候就会自动去创建UserInfo对象。

```
public class MainActivity extends AppCompatActivity {

//自动创建对象，不用我们去new UserInfo()了
@AutoWired
UserInfo mUserInfo;

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
AutoWiredProcess.bind(this);
}

}
```

2.1.3.2、通过注解自动findViewById()

我们也来简单的来实现一个类似Butterknife 库里面自动绑定View的一个功能。不用在每个View都要去写findViewById来找至

> 声明BindView注解，而且规定需要一个int参数。int参数代表View对应的id

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

int value();
}
```

> 解析BindView注解，通过findViewById找到VIew，在把View赋值给BindView注解指向的变量。

```
public class ButterKnifeProcess {

/**
* 绑定Activity
*/
public static void bind(final Activity activity) {
```

```
Class annotationParent = activity.getClass();
Field[] fields = annotationParent.getDeclaredFields();
Method[] methods = annotationParent.getDeclaredMethods();
// OnClick
// 找到类里面所有的方法
for (final Method method : methods) {
//找到添加了OnClick注解的方法
OnClick clickMethod = method.getAnnotation(OnClick.class);
if (clickMethod != null && clickMethod.value().length != 0) {
for (int id : clickMethod.value()) {
final View view = activity.findViewById(id);
view.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
try {
method.invoke(activity, view);
} catch (IllegalAccessException e) {
e.printStackTrace();
} catch (InvocationTargetException e) {
e.printStackTrace();
}
}
});
}
}
}

}

}
```

> 使用BindView注解，onCreate里面调用了ButterKnifeProcess.bind(this);来解析注解。

```
public class MainActivity extends AppCompatActivity {

//自动绑定view
@BindView(R.id.text_abstract_processor)
TextView mTextView;

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
ButterKnifeProcess.bind(this);
}
}
```

2.2、自定义编译时注解

编译时注解就是在编译的过程中用一个javac注解处理器来扫描到我们自定义的注解，生成我们需要的一些文件(通常是java文件

自定义编译时注解的步骤：

1. 声明注解。
2. 编写注解处理器。
3. 生成文件(通常是JAVA文件)。

> 第二步和第三步其实是柔和在一起的。我这里为了清晰一点就把他们独立开来了。

2.2.1、声明注解

编译时注解的声明和运行时注解的声明一样也是三步:

通过@Retention(RetentionPolicy.TYPE)元注解确定我们注解是在编译的时候使用。

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。

比如下面的代码我们自定义了一个作用在类上的编译时注解Factory，并且这个注解是需要两个参数的，一个是Class类型，一

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.CLASS)
public @interface Factory {
Class type();

String id();
}
```

2.2.2、编写注解处理器

和运行时注解的解析不一样，编译时注解的解析需要我们自己去实现一个注解处理器。

> 注解处理器(Annotation Processor)是javac的一个工具，它用来在编译时扫描和处理注解(Annotation)。一个注解的注解处理器，以Java代码（或者编译过的字节
> (通常是.java文件)作为输出。而且这些生成的Java文件同咱们手动编写的Java源代码一样可以调用。(注意：不能修改已经存在的java文件代码)。

注解处理器所做的工作，就是在代码编译的过程中，找到我们指定的注解。然后让我们更加自己特定的逻辑做出相应的处理(通
件)。

注解处理器的写法有固定套路的，两步：

注册注解处理器(这个注解器就是我们第二步自定义的类)。

自定义注解处理器类继承AbstractProcessor。

2.2.2.1、注册注解处理器

打包注解处理器的时候需要一个特殊的文件 javax.annotation.processing.Processor 在 META-INF/services 路径下。在
javax.annotation.processing.Processor文件里面写上我们自定义注解处理器的全称(包加类的名字)如果有多个注解处理器换行写

伟大的google为了方便我们注册注解处理器。给提供了一个注册处理器的库
@AutoService(Processor.class)的注解来简化我们的操作。我们只需要在我们自定义的注解处理器类前面加上google的这个注解
自动生成javax.annotation.processing.Processor文件，写入相的信息。不需要我们手动去创建。当然了如果你想使用google的这
必须加上下面的依赖。

```
compile 'com.google.auto.service:auto-service:1.0-rc3'
 1
```

比如下面的这段代码就使用上了google提供的这个注解器处理库，会自动注册注解处理器。

```
@AutoService(Processor.class)
public class FactoryProcessor extends AbstractProcessor {
...

}
```

2.2.2.2、自定义注解处理器类

自定义的注解处理器类一定要继承AbstractProcessor，否则找不到我们需要的注解。在这个类里面找到我们需要的注解。做比

关于AbstractProcessor里面的一些函数我们也做一个简单的介绍。

```
/**
* 每个Annotation Processor必须有一个空的构造函数。
* 编译期间，init()会自动被注解处理工具调用，并传入ProcessingEnvironment参数，
* 通过该参数可以获取到很多有用的工具类（Element，Filer，Messager等）
*/
@Override
public synchronized void init(ProcessingEnvironment processingEnvironment) {
```

```
super.init(processingEnvironment);
}

/**
* 用于指定自定义注解处理器(Annotation Processor)是注册给哪些注解的(Annotation),
* 注解(Annotation)指定必须是完整的包名+类名
*/
@Override
public Set<String> getSupportedAnnotationTypes() {
return super.getSupportedAnnotationTypes();
}

/**
* 用于指定你的java版本，一般返回：SourceVersion.latestSupported()
*/
@Override
public SourceVersion getSupportedSourceVersion() {
return SourceVersion.latestSupported();
}

/**
* Annotation Processor扫描出的结果会存储进roundEnvironment中，可以在这里获取到注解内容，编写你的操作逻辑。
* 注意:process()函数中不能直接进行异常抛出,否则程序会异常崩溃
*/
@Override
public boolean process(Set<? extends TypeElement> set, RoundEnvironment roundEnvironment) {
return false;
}
```

注解处理器的核心是process()方法(需要重写AbstractProcessor类的该方法)，而process()方法的核心是Element元素。Elem
素，在注解处理过程中，编译器会扫描所有的Java源文件，并将源码中的每一个部分都看作特定类型的Element。它可以代表包、
段等多种元素种类。所有Element肯定是有好几个子类。如下所示。

Element子类	解释
TypeElement	类或接口元素
VariableElement	字段、enum常量、方法或构造方法参数、局部变量或异常参数元素
ExecutableElement	类或接口的方法、构造方法，或者注解类型元素
PackageElement	包元素
TypeParameterElement	类、接口、方法或构造方法元素的泛型参数

关于Element类里面的方法我们也做一个简单的介绍：

```
/**
* 返回此元素定义的类型，int,long这些
*/
TypeMirror asType();

/**
* 返回此元素的种类:包、类、接口、方法、字段
*/
ElementKind getKind();

/**
* 返回此元素的修饰符:public、private、protected
*/
Set<Modifier> getModifiers();

/**
* 返回此元素的简单名称(类名)
*/
```

```
    Name getSimpleName();

    /**
    * 返回封装此元素的最里层元素。
    * 如果此元素的声明在词法上直接封装在另一个元素的声明中，则返回那个封装元素；
    * 如果此元素是顶层类型，则返回它的包；
    * 如果此元素是一个包，则返回 null；
    * 如果此元素是一个泛型参数，则返回 null.
    */
    Element getEnclosingElement();

    /**
    * 返回此元素直接封装的子元素
    */
    List<? extends Element> getEnclosedElements();

    /**
    * 返回直接存在于此元素上的注解
    * 要获得继承的注解，可使用 getAllAnnotationMirrors
    */
    List<? extends AnnotationMirror> getAnnotationMirrors();

    /**
    * 返回此元素上存在的指定类型的注解
    */
    <A extends Annotation> A getAnnotation(Class<A> var1);
```

> 关于TypeElement、VariableElement、ExecutableElement、PackageElement、TypeParameterElement每个类特有的方法我们这里就没有介绍了，大家可以到看一看。

自定义处理器的过程中我们除了要了解Element类和他的子类的用法，还有四个帮助类也是需要我们了解的。Elements、Typ Messager。

注解解析器帮助类	解释
Elements	一个用来处理Element的工具类
Types	一个用来处理TypeMirror的工具类
Filer	用于创建文件(比如创建class文件)
Messager	用于输出，类似printf函数

这四个帮助类都可以在init()函数里面通过ProcessingEnvironment获取到。类似如下的代码获取

```
    @AutoService(Processor.class)
    public class FactoryProcessor extends AbstractProcessor {

    /**
    * 用来处理TypeMirror的工具类
    */
    private Types mTypeUtils;
    /**
    * 用于创建文件
    */
    private Filer mFiler;
    /**
    * 用于打印信息
    */
    private Messager mMessager;
    ...

    /**
```

```
* 获取到Types、Filer、Messager、Elements
*/
@Override
public synchronized void init(ProcessingEnvironment processingEnvironment) {
super.init(processingEnvironment);
mTypeUtils = processingEnvironment.getTypeUtils();
mFiler = processingEnvironment.getFiler();
mMessager = processingEnvironment.getMessager();
...
}

...


}
```

2.2.3、生成文件

生成文件，通常是生成一个java文件。直接调用帮助类Filer的createSourceFile()函数就可以创建一个java文件。之后就是在这
我们需要的内容了。为了提高大家的开发效率推荐两个写java源文件的开源库FileWriter和JavaPoet。两个库用起来也很简单，这里
了。生成文件这一部分的内容非常的简答。具体可以参考我们下编译时注解实例。

> JavaWrite是JavaPoet增强版。

2.2.4、编译时注解实例

从网上找了一个非常全面自定义编译时注解的例子。例子来源于 https://blog.csdn.net/github_35180164/article/details/5
注解实现工厂模式。每个工厂模式通常都会有一个相应的Factory的帮助类来选择具体的工厂类，我们现在就想通过编译时注解来自
的帮助类，不用我们去手动编写了。

> Peple抽象类

```
public abstract class People {

public abstract String getName();

public abstract int getAge();

public abstract int getSex();

}
```

> Male类实现了People类，并且添加了@Factory注解

```
@Factory(id = "Male", type = People.class)
public class Male extends People{

@Override
public String getName() {
return "男生";
}

@Override
public int getAge() {
return 28;
}

@Override
public int getSex() {
return 0;
}
```

```
}
```

> Female类实现了People类，并且添加了@Factory注解

```
@Factory(id = "Female", type = People.class)
public class Female extends People {

@Override
public String getName() {
return "女生";
}

@Override
public int getAge() {
return 27;
}

@Override
public int getSex() {
return 1;
}
}
```

根据上面添加的注解，我们会去自动生成一个PeopleFactory类，而且里面的内容也编译的时候自动生成的，内容如下。

```
public class PeopleFactory {

public People create(String id) {
if (id == null) {
throw new IllegalArgumentException("id is null!");
}
if ("Female".equals(id)) {
return new com.tuacy.annotationlearning.annotation.abstractprocessor.Female();
}

if ("Male".equals(id)) {
return new com.tuacy.annotationlearning.annotation.abstractprocessor.Male();
}

throw new IllegalArgumentException("Unknown id = " + id);
}
}
```

为了实现上述功能，我们在Android Studio里面新建一个project。然后再新建一个annotationprocess的module，新建moc Library。在annotationprocess里面写我们注解的申明和注解的处理。

先申明一个Factory的注解

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.CLASS)
public @interface Factory {

/**
* 工厂的名字
*/
Class type();

/**
* 用来表示生成哪个对象的唯一id
*/
String id();


}
```

在自定义一个FactoryProcessor注解处理器继承AbstractProcessor。FactoryProcessor代码里面的内容比较多这里我就不粘
找到我们自定义的注解，然后做一些相应的判断，最后生成java文件代码。相应的代码大家可以在下面给出的DEMO里面看到，DE
写的也非常详细。生成JAVA文件使用的是JavaWriter库。

最后我们把annotationprocess module里面的代码打成jar包放到我们需要的工程里面去(同时把javawriter-2.5.1.jar也拷贝进
面说的People工厂一样使用就OK了。

本文DEMO下载地址

关于自定义注解的内容，我们就说的就这么多，希望能给大家起到一个抛砖引玉的作用，如果大家对DEMO里面的代码有什么