Java Annotation

Trinea

www.trinea.cn

微博：trinea

博客：http://www.trinea.cn/

GitHub：https://github.com/Trinea

目录

Annotation 示例

- Override

```
@Override
public void onCreate(Bundle savedInstanceState);
```

- Retrofit

```
@GET("/users/{username}")
User getUser(@Path("username") String username);
```

- Butter Knife

```
@InjectView(R.id.user) EditText username;
```

- ActiveAndroid

```
@Column(name = “Name") public String name;
```


Retrofit 为符合 RESTful 规范的网络请求框架

Butter Knife 为 View 及事件等依赖注入框架

Active Android 为 ORM 框架

更多见：https://github.com/Trinea/android-open-project

目录

Annotation 是什么

An annotation is a form of metadata, that can be added to Java source code. Classes, methods, variables, parameters and packages may be annotated. Annotations have no direct effect on the operation of the code they annotate.

能够添加到 Java 源代码的语法元数据。类、方法、变量、参数、包都可以被注解，可用来将信息元数据与程序元素进行关联

Annotation 中文常译为“注解”

Annotation 作用

- 标记，用于告诉编译器一些信息
- 编译时动态处理，如动态生成代码
- 运行时动态处理，如得到注解信息

这里的三个作用实际对应着后面自定义 Annotation 时说的 @Retention 三种值分别表示的 Annotation

```
public class Person {

    private int    id;
    private String name;

    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public boolean equals(Person person) {
        return person.id == id;
    }

    public int hashCode() {
        return id;
    }

    public static void main(String[] args) {

        Set<Person> set = new HashSet<Person>();
        for (int i = 0; i < 10; i++) {
            set.add(new Person(1, "Jim"));
        }
        System.out.println(set.size());
    }
}
```

运行结果如何?

```
public class Person {

    private int    id;
    private String name;

    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        return (obj instanceof Person) && (((Person)obj).id == id);
    }

    @Override
    public int hashCode() {
        return id;
    }
}
```


这个示例代码程序实际想说明的是标记型注解 Override 的作用。

注意红色加粗部分，前一页的重写 equals 方法实际是没成功的，因为入参不对，变成函数重载了，但如果记得加上 Override 注解，编译器就会提示函数重写错误。

所以上一页程序的输出值是 10 而不是 1

目录

Annotation 分类

- 标准 Annotation

  Override, Deprecated, SuppressWarnings

- 元 Annotation

  @Retention, @Target, @Inherited, @Documented

- 自定义 Annotation

1. 标准 Annotation 是指 Java 自带的几个 Annotation，上面三个分别表示重写函数，函数已经被禁止使用，忽略某项 Warning
2. 元 Annotation 是指用来定义 Annotation 的 Annotation，在后面 Annotation 自定义部分会详细介绍含义
3. 自定义 Annotation 表示自己根据需要定义的 Annotation，定义时需要用到上面的元 Annotation

这里只是一种分类而已，也可以根据作用域分为源码时、编译时、运行时 Annotation，后面在自定义 Annotation 时会具体介绍

目录

Annotation 自定义—调用

```
public class App {

  @MethodInfo(
    author = “trinea.cn+android@gmail.com”,
    date = "2014/02/14",
    version = 2)
  public String getAppName() {
    return "trinea";
  }
}
```


这里是调用自定义 Annotation——MethodInfo 的示例

MethodInfo Annotation 作用为给方法添加相关信息，包括 author、date、version

Annotation 自定义—定义

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Inherited
public @interface MethodInfo {

    String author() default "trinea@gmail.com";

    String date();

    int version() default 1;
}
```


这里是 MethodInfo 的实现部分

1. 通过 @interface 定义，注解名即为自定义注解名
2. 注解配置参数名为注解类的方法名，且：

a. 所有方法没有方法体，没有参数没有修饰符，实际只允许 public & abstract 修饰符，默认为 public ，不允许抛异常

b. 方法返回值只能是基本类型，String, Class, annotation, enumeration 或者是他们的一维数组

c. 若只有一个默认属性，可直接用 value() 函数。一个属性都没有表示该 Annotation 为 Mark Annotation

3. 可以加 default 表示默认值

元 Annotation

- @Documented 是否会保存到 Javadoc 文档中
- @Retention 保留时间，可选值 SOURCE（源码时），CLASS（编译时），RUNTIME（运行时），默认为 CLASS
- @Target 可以用来修饰哪些程序元素，如 TYPE, METHOD, CONSTRUCTOR, FIELD, PARAMETER 等，未标注则表示可修饰所有
- @Inherited 是否可以被继承，默认为 false

@Retention 值为 SOURCE 大都为 Mark Annotation，这类 Annotation 大都用来校验，比如 Override, Deprecated, SuppressWarnings

Annotation 自定义—解释

- 通过 @interface 定义，注解名即为自定义注解名
- 注解配置参数名为注解类的方法名，且：

A. 所有方法没有方法体，没有参数没有修饰符，实际只允许public & abstract 修饰符，默认为 public ，不允许抛异常

B. 方法返回值只能是基本类型，String, Class, annotation, enumeration 或者是他们的一维数组

C. 只有一个默认属性，可直接用 value() 函数。 一个属性没有表示为 Mark Annotation

- 可以加 default 表示默认值

目录

运行时 Annotation 解析

运行时 Annotation 指 @Retention 为 RUNTIME 的 Annotation，可手动调用下面 API 解析

- 常用 API

a. method.getAnnotation(AnnotationName.class);
b. method.getAnnotations();
c. method.isAnnotationPresent(AnnotationName.class);

其他 @Target 如 Field，Class 方法类似

因为一个 Target 可以被多个 Annotation 修饰，getAnnotation(AnnotationName.class) 表示得到该 Target 某个 Annotation 的信息

getAnnotations() 则表示得到该 Target 所有 Annotation

isAnnotationPresent(AnnotationName.class) 表示该 Target 是否被某个 Annotation 修饰

运行时 Annotation 解析

```
public static void main(String[] args) {
	try {
		Class cls = Class.forName("cn.trinea.java.test.annotation.App");
		for (Method method : cls.getMethods()) {
			MethodInfo methodInfo = method.getAnnotation(
MethodInfo.class);
			if (methodInfo != null) {
				System.out.println("method name:" + method.getName());
				System.out.println("method author:" + methodInfo.author());
				System.out.println("method version:" + methodInfo.version());
				System.out.println("method date:" + methodInfo.date());
			}
		}
	} catch (ClassNotFoundException e) {
		e.printStackTrace();
	}
}
```


以之前自定义的 MethodInfo 为例，利用 Target（这里是 Method）getAnnotation 函数得到 Annotation 信息，然后就可以调用 Annotation 的方法得到响应属性值

编译时 Annotation 解析

编译时 Annotation 指 @Retention 为 CLASS 的 Annotation，由 apt 自动解析。

- 需要做的

a. 自定义类集成自 AbstractProcessor
b. 重写其中的 process 函数

这块很多同学不理解，实际是 apt(Annotation Processing Tool) 在编译时自动查找所有继承自 AbstractProcessor 的类，然后调用他们的 process 方法去处理

编译时 Annotation 解析

```
@SupportedAnnotationTypes({ "cn.trinea.java.test.annotation.MethodInfo" })
public class MethodInfoProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> annotations,
RoundEnvironment env) {

        HashMap<String, String> map = new HashMap<String, String>();
        for (TypeElement te : annotations) {
            for (Element element : env.getElementsAnnotatedWith(te)) {
                MethodInfo methodInfo = element.getAnnotation(MethodInfo.class);
                map.put(element.getEnclosingElement().toString(), methodInfo.author());
            }
        }
        return false;
    }
}
```

www.trinea.cn

SupportedAnnotationTypes 表示这个 Processor 要处理的 Annotation 名字。

参数 annotations 表示待处理的 Annotations

参数 env 表示当前或是之前的运行环境

process 函数返回值表示这组 annotations 是否被这个 Processor 接受，如果接受后续子的 Processor 不会再对这个 Annotations 进行处理

目录

Annotation — Retrofit

- 调用

```
@GET("/users/{username}")
User getUser(@Path("username") String username);
```

- 定义

```
@Documented
@Target(METHOD)
@Retention(RUNTIME)
@RestMethod("GET")
public @interface GET {
  String value();
}
```


从定义可看出 Retrofit 的 Get Annotation 是运行时 Annotation，并且只能用于修饰 Method

Annotation — Retrofit

- 原理

```
/** Loads {@link #requestMethod} and {@link #requestType}. */
private void parseMethodAnnotations() {
  for (Annotation methodAnnotation : method.getAnnotations()) {
    Class<? extends Annotation> annotationType = methodAnnotation.annotationType();
    RestMethod methodInfo = null;

    // Look for a @RestMethod annotation on the parameter annotation indicating request
    for (Annotation innerAnnotation : annotationType.getAnnotations()) {
      if (RestMethod.class == innerAnnotation.annotationType()) {
        methodInfo = (RestMethod) innerAnnotation;
        break;
      }
    }
```


RestMethodInfo.java 的 parseMethodAnnotations 方法如上，会检查每个方法的每个 Annotation， 看是否被 RestMethod 这个 Annotation 修饰的 Annotation 修饰，这个有点绕，就是是否被 GET、DELETE、POST、PUT、HEAD、PATCH 这些 Annotation 修饰，然后得到 Annotation 信息，在对接口进行动态代理时会掉用到这些 Annotation 信息从而完成调用。

因为 Retrofit 原理设计到动态代理，这里只介绍 Annotation，具体原理分析请期待 Android 优秀开源项目实现原理解析 项目的完成

Annotation — Butter Knife

- 调用

```
@InjectView(R.id.user)
EditText username;
```

- 定义

```
@Retention(CLASS)
@Target(FIELD)
public @interface InjectView {
  int value();
}
```


可看出 Butter Knife 的 InjectView Annotation 是编译时 Annotation，并且只能用于修饰属性

Annotation — Butter Knife

- 原理

```
@Override public boolean process(Set<? extends TypeElement> elements, RoundEnvironment env) {
  Map<TypeElement, ViewInjector> targetClassMap = findAndParseTargets(env);

  for (Map.Entry<TypeElement, ViewInjector> entry : targetClassMap.entrySet()) {
    TypeElement typeElement = entry.getKey();
    ViewInjector viewInjector = entry.getValue();

    try {
      JavaFileObject jfo = filer.createSourceFile(viewInjector.getFqcn(), typeElement);
      Writer writer = jfo.openWriter();
      writer.write(viewInjector.brewJava());
      writer.flush();
      writer.close();
    } catch (IOException e) {
      error(typeElement, "Unable to write injector for type %s: %s", typeElement, e.getMessage
    }
  }

  return true;
}
```

ButterKnifeProcessor.java 的 process 方法如上，编译时，在此方法中过滤 InjectView 这个 Annotation 到 targetClassMap 后，会根据 targetClassMap 中元素生成不同的 class 文件到最终的 APK 中，然后在运行时调用 ButterKnife.inject(x) 函数时会到之前编译时生成的类中去找。

这里只介绍 Annotation，具体原理分析请期待 Android 优秀开源项目实现原理解析 项目的完成

Annotation — ActiveAndroid

- 调用

```
@Column(name = “Name")
public String name;
```

- 定义

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Column {
    ……
}
```


可看出 ActiveAndroid 的 Column Annotation 是运行时 Annotation，并且只能用于修饰属性

Annotation — ActiveAndroid

- 原理

```
List<Field> fields = new LinkedList<Field>(ReflectionUtils.getDeclaredColumnFields(type));
Collections.reverse(fields);

for (Field field : fields) {
    if (field.isAnnotationPresent(Column.class)) {
        final Column columnAnnotation = field.getAnnotation(Column.class);
        String columnName = columnAnnotation.name();
        if (TextUtils.isEmpty(columnName)) {
            columnName = field.getName();
        }

        mColumnNames.put(field, columnName);
    }
}
```


TableInfo.java 的构造函数如上，运行时，得到所有行信息并存储起来用来构件表信息。

这里只介绍 Annotation，具体原理分析请期待 Android 优秀开源项目实现原理解析 项目的完成

回顾

一. Annotation 示例

二. Annotation 概念及作用

三. Annotation 分类

四. Annotation 自定义

五. Annotation 解析

六. 几个 Android 开源库 Annotation 原理简析

问题

- 如何判断一个 Annotation 是运行时还是编译时生效以及如何快速找到它的解析代码

QA

- 微博：trinea
- 博客：http://www.trinea.cn/
- GitHub：https://github.com/Trinea

关于 PPT 中或是 Android 的问题欢迎博客或是邮件（trinea.cn+android@gmail.com）和我交流。不过伸手党请远离，不回复。