Android公共技术点之一-Java注解

2016-04-25 | publicTech | 1129

基础是学习任何技术的必须，接下来会介绍一些Android上用到的一些公共技术点。

看了Trinea在codekk上的一些公共技术点，这些点不管在Java还是Android上，都是重要的基础点,所以准备学习之。

Annotation概念

注解是 Java5的一个新特性。注解是插入代码中的一种注释或者是一种元数据(meta data)。
官方的解释:

> Annotations, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.

作用：

- 编写文档:通过代码里的元数据生成文档
- 代码分析:通过代码里标识的元数据对代码进行分析
- 编译检查:通过代码里标识的元数据让编译器实现基本的编译检查

元注解

元注解的作用就是注解其他注解。Java中定义了4个标准的meta-annotation类型，用以对其他的annotation类型做说明，分别是：

1. @Target
2. @Retention
3. @Documented
4. @Inherited

@Target

说明了Annotation所修饰的对象的作用：用户描述注解的使用范围
取值(ElementType):

- CONSTRUCTOR: 描述构造器
- FIELD：描述域
- LOCAL_VARIABLE:描述局部变量
- METHOD:描述方法
- PACKAGE:描述包
- PARAMETER:描述参数
- TYPE:描述类、接口(包括注解类型) 或enum声明

如果没有声明，可以修饰所有

@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
取值(RetentionPolicy):

- SOURCE(源码时)
- CLASS(编译时)
- RUNTIME(运行时)

默认为CLASS

@Documented

标记注解，没有成员

用于描述其它类型的annotation应该被作为标注的程序成员的公共api，可以文档化

@Inherited

标记注解

用该注解修饰的注解，会被子类继承

Annotation自定义

自定义注解使用@interface声明一个注解，每一个方法就是声明一个配置参数，方法的名称就是参数的名称

返回的类型就是参数的类型(返回值类型只能是基本类型、Class、String、enum)。可以通过default来声明参数的默认值。

下面举个例子:

```
1  @Documented
2  @Retention(CLASS)
3  @Target(FIELD)
4  @Inherited
5  public @interface MyAnnotation {
6      String name();
7      int id() default 1;
8      int[] value();
9  }
```

定义了MyAnnotation注解是编译时注解，用于修饰属性，可以被继承和文档化，有3个配置参数。

Annotation解析

主要是根据@Retention分类，下面主要介绍 CLASS 和 RUNTIME

Search this site

运行时Annotation解析

运行时Annotation是指@Retention为 RUNTIME 的Annotation,解析Annotation的API：

```
1  T getAnnotation(Class annotationClass) //返回改程序上存在、指定类型的注解
2  Annotation[] getAnnotations()  //返回改程序元素上存在的所有注解
3  boolean is AnnotationPresent(Annotation)  //判断该程序元素上是否包含指定类型的注解
4  Annotation[] getDeclaredAnnotations()  //返回直接存在在改元素上的所有注解,不包含继承的注解
```

获取注解的信息：

```
1  private void processAnnotation(Class<?> clazz) {
2      Field[] fields = clazz.getDeclaredFields();
3      for (Field field : fields) {
4          if (field.isAnnotationPresent(MyAnnotation.class)) {
5              MyAnnotation myAnnotation = field.getAnnotation(MyAnnotation.class);
6              Log.d("DEBUG", "### id:" + myAnnotation.id() + ", name:" + myAnnotation.name()
7                      + ", value: " + myAnnotation.value());
8          }
9      }
10 }
```

编译时Annotation解析

编译时Annotation指@Retention为 CLASS 的Annotation，由编译器自动解析，基于APT注解处理工具。

apt：Annotation Processing Tool，官方说明

> The command-line utility apt, annotation processing tool, finds and executes annotation processors based on the annotations present in the set of specified source files being examined. The annotation processors use a set of reflective APIs and supporting infrastructure to perform their processing of program annotations (JSR 175)

如何使用apt：

1. 自定义类集成自 AbstractProcessor

Search this site

2. 重写其中的 process 函数

上文定义的MyAnnotation，使用apt，该如何进行解析：(在android studio中直接使用AbstractProcessor，会找不到这个类，具体的解决方法，请看知识点Annotation Processing Tool)

```
@SupportedAnnotationTypes({ "MyAnnotation" })
public class MyAnnotationProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment env) {
        for (TypeElement te : annotations) {
            for (Element element : env.getElementsAnnotatedWith(te)) {
                MyAnnotation myAnnotation = element.getAnnotation(MyAnnotation.class);
                ... //具体的处理逻辑
            }
        }
        return false;
    }
}
```

SupportedAnnotationTypes 表示这个 Processor 要处理的 Annotation 名字。

process 函数中参数 annotations 表示待处理的 Annotations，参数 env 表示当前或是之前的运行环境

优点：

- 提高开发效率
- 减少代码量
- apt并不会影响性能

缺点：

- 可读性较差
- 生成一些辅助类，内存消耗变大
- android的65535方法数问题

开源库实例讲解

现在很多第三方库运用注解来实现具体功能，看看它们之间的区别

Retrofit

Retrofit是Restful的httpClient，目前版本2.0.2。

看官网的例子

```
1  public interface GitHubService {
2    @GET("users/{user}/repos")
3    Call<List<Repo>> listRepos(@Path("user") String user);
4  }
5  ....
6  Retrofit retrofit = new Retrofit.Builder()
7      .baseUrl("https://api.github.com/")
8      .build();
9
10 GitHubService service = retrofit.create(GitHubService.class);
```

@GET定义：

```
1  @Documented
2  @Target(METHOD)
3  @Retention(RUNTIME)
4  public @interface GET {
5    String value() default "";
6  }
```

GET的Annotation定义是运行时的注解，只能修饰方法，有一个String属性。

在Retrofit初始化中可以看到原理，具体的实现在ServiceMethod

```
1  public Builder(Retrofit retrofit, Method method) {
2    this.retrofit = retrofit;
3    this.method = method;
4    this.methodAnnotations = method.getAnnotations();
5    this.parameterTypes = method.getGenericParameterTypes();
6    this.parameterAnnotationsArray = method.getParameterAnnotations();
7  }
8  ...
9  for (Annotation annotation : methodAnnotations) {
```

```
10      parseMethodAnnotation(annotation);
11    }
12    ...
13    private void parseMethodAnnotation(Annotation annotation) {
14      if (annotation instanceof DELETE) {
15        parseHttpMethodAndPath("DELETE", ((DELETE) annotation).value(), false);
16      } else if (annotation instanceof GET) {
17        parseHttpMethodAndPath("GET", ((GET) annotation).value(), false);
18    ...
19    }}}
```

上述代码会检查每个Annotation，看是否被rest method注解修饰，然后得到Annotation信息，在对接口进行动态代理时调用这些信息，完成具体的调用。

在Refrofit初始化create的时候，有动态代理行为。

```
1     public <T> T create(final Class<T> service) {
2     Utils.validateServiceInterface(service);
3     if (validateEagerly) {
4       eagerlyValidateMethods(service);
5     }
6     return (T) Proxy.newProxyInstance(service.getClassLoader(), new Class<?>[] { service },
7         new InvocationHandler() {
8           private final Platform platform = Platform.get();
9
10          @Override public Object invoke(Object proxy, Method method, Object... args)
11              throws Throwable {
12            // If the method is a method from Object then defer to normal invocation.
13            if (method.getDeclaringClass() == Object.class) {
14              return method.invoke(this, args);
15            }
16            if (platform.isDefaultMethod(method)) {
17              return platform.invokeDefaultMethod(method, service, proxy, args);
18            }
19            ServiceMethod serviceMethod = loadServiceMethod(method);
20            OkHttpCall okHttpCall = new OkHttpCall<>(serviceMethod, args);
21            return serviceMethod.callAdapter.adapt(okHttpCall);
22          }
23        });
24    }
```

Search this site

Butterknife

Butterknife,使用的是apt技术。

目前稳定版本8.0.0，与7.0相比，主要runtime和compiler分离成了两个，支持更多的配置属性。下面的例子基于7.0.1

```
1  @Bind(R.id.toolbar)
2  Toolbar toolbar;
```

@Bind定义：

```
1  @Retention(CLASS) @Target(FIELD)
2  public @interface Bind {
3    /** View ID to which the field will be bound. */
4    int[] value();
5  }
```

可以看出Bind注解是编译时注解，只能修饰属性，有个int数组属性。

具体的原理实现在ButterKnifeProcessor

```
1   @Override
2   public boolean process(Set<? extends TypeElement> elements, RoundEnvironment env) {
3     Map<TypeElement, BindingClass> targetClassMap = findAndParseTargets(env);
4
5     for (Map.Entry<TypeElement, BindingClass> entry : targetClassMap.entrySet()) {
6       TypeElement typeElement = entry.getKey();
7       BindingClass bindingClass = entry.getValue();
8
9       try {
10        JavaFileObject jfo = filer.createSourceFile(bindingClass.getFqcn(), typeElement);
11        Writer writer = jfo.openWriter();
12        writer.write(bindingClass.brewJava());
13        writer.flush();
14        writer.close();
15      } catch (IOException e) {
16        error(typeElement, "Unable to write view binder for type %s: %s", typeElement,
17            e.getMessage());
```

```
18          }
19        }
20
21        return true;
22    }
```

process方法，编译时，过滤Binding注解到targetClassMap，会根据 targetClassMap 中元素生成不同的 class 文件到最终的 APK 中，
运行时调用 ButterKnife.bind方法会到之前编译生成的类中去找。

本来还要分析下Dagger的注解，不过Dagger这块目前还不是很熟悉，它主要也是依赖注入框架，后面会和依赖注入知识一起介绍。

参考

- 公共技术点之 Java 注解 Annotation
- 深入理解Java：注解（Annotation）自定义注解入门
- 最新ButterKnife框架原理

欢迎您扫一扫上面的微信公众号，订阅我的博客！

曾经有一份打赏放在我面前,我没有珍惜.如果上天给我再来一次的机会，我会说三个字:赏死我.

赏

Search this site

android # annotation # ButterKnife

‹ Android公共技术点之二-Annotation Processing Tool

React Native Android开发之一: 环境配置 ›

1