注解

一、注解简介

注解（Annotation）,也叫元数据。一种代码级别的说明。它是JDK1.5及以后版本引入的一个特性，与类、接口、枚举是在同一个层次。它可以声明在包、类、字段、方法、局部变量、方法参数等的前面，用来对这些元素进行说明 。

二、注解的作用

1.生成文档；是java最早提供的注解。常用的有@param @return 等

2.跟踪代码依赖性，实现替代配置文件功能；

3.编译时进行格式检查；如@Override

三、元注解（meta-annotation）

1、元注解定义

元注解就是用于定义自定义注解的注解。在JDK 1.5中提供了4个标准的用来对注解类型进行注解的注解类，它们分别是：

@Target : 描述注解的使用范围  
@Retention : 描述注解的生存期，主要有三种

@Documented : 描述在使用 javadoc 工具为类生成帮助文档时是否要保留其注解信息

@Inherited : 使被它修饰的注解具有继承性

2、@Target注解

可修饰的对象范围：注解可以用于修饰 packages、types（类、接口、枚举、注解类）、类成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数），在定义注解类时使用了@Target 能够更加清晰的知道它能够被用来修饰哪些对象，它的取值范围定义在ElementType 枚举中。如下：

**public enum** ElementType {

*TYPE*, *//类、接口、枚举类*  *FIELD*, *//成员变量（包括枚举常量）  
 METHOD*, *//成员方法  
 PARAMETER*, *//方法参数  
 CONSTRUCTOR*, *//构造方法  
 LOCAL\_VARIABLE*, *//局部变量  
 ANNOTATION\_TYPE*, *//注解类  
 PACKAGE*, *//包  
 TYPE\_PARAMETER*, *//类型参数* ***@since*** *1.8  
 TYPE\_USE //使用类型的任何地方* ***@since*** *1.8*}

3、@Retention注解

Reteniton注解用来限定那些被它所注解的注解类在注解到其他类上以后，可被保留到何时，一共有三种策略，定义在RetentionPolicy枚举类中。如下：

**public enum** RetentionPolicy {  
  *SOURCE*, *// 源文件保留  
 CLASS*, *// 编译期保留，默认值*

*RUNTIME // 运行期保留，可通过反射获取注解信息*}

定义分别使用三种策略之一的注解类：

@Retention(RetentionPolicy.*SOURCE*)  
**public** @**interface** SourcePolicy{  
}

@Retention(RetentionPolicy.*CLASS*)  
**public** @**interface** ClassPolicy{  
}

@Retention(RetentionPolicy.*RUNTIME*)  
**public** @**interface** RuntimePolicy{  
}

测试类：

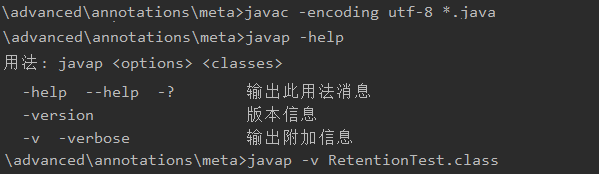
**public class** RetentionTest{  
 @SourcePolicy

**public void** sourcePolicy(){}

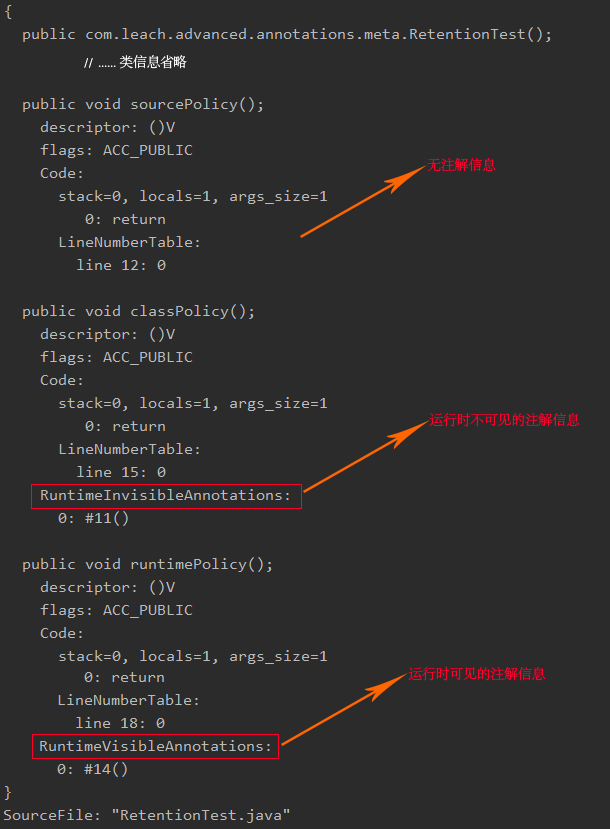
@ClassPolicy  
 **public void** classPolicy(){}

@RuntimePolicy  
 **public void** runtimePolicy(){}  
}

使用javac编译，使用javap查看字节码信息：



字节码信息如下：



4、@Documented注解

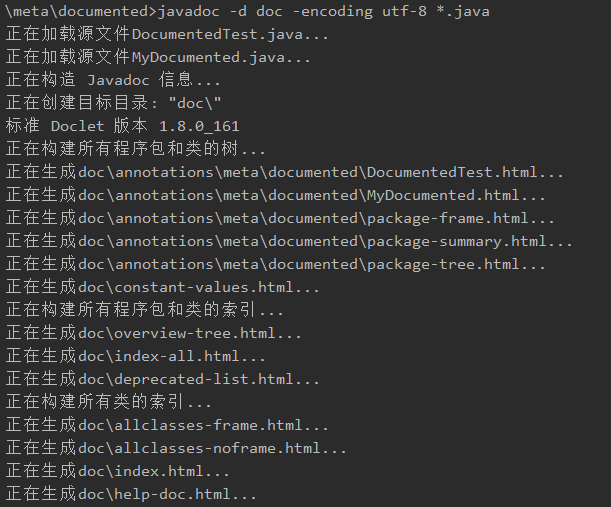
创建一个带有 @Documented 的自定义注解类：

@Documented  
@Target({ElementType.*TYPE*, ElementType.*METHOD*})  
**public** @**interface** MyDocumented{  
 String value() **default** "这是@Documented注解为文档添加的注释";  
}

创建测试类：

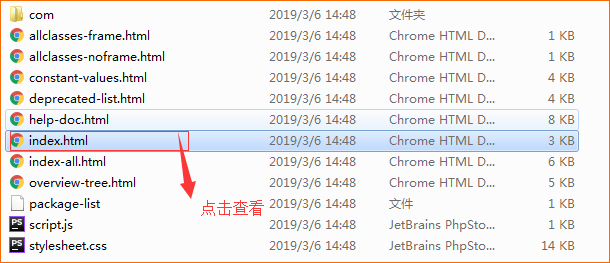
@MyDocumented  
**public class** documentedTest{  
 @Override  
 @MyDocumented  
 **public** String toString(){  
 **return this**.toString();  
 }  
}

接下来，使用以下命令为 MyDocumentedTest 类生成帮助文档：



这样在doc目录下就有我们需要的帮助文档了，通过index.html可以查看。

doc目录结构如下：



访问index.html页面,如下：



可以看到，DocumentTest使用了注解MyDocumented,可以点击DocumentedTest查看更详细内容。

5、@Inherited注解

创建一个被@Inherited修饰的注解类MyInherited：

@Inherited  
@Target(ElementType.*TYPE*)  
@Retention(RetentionPolicy.*RUNTIME*)  
**public** @**interface** MyInherited{  
 String name() **default** "inherited annotation";  
}

创建一个父类Animal使用MyInherited注解注释，和一个子类Cat不使用注解注释：

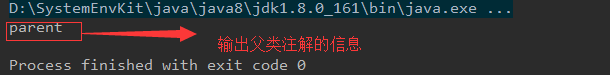
@MyInherited(name="parent")  
**public abstract class** Animal{  
 **public abstract void** sayMyself();  
}

**public class** Cat **extends** Animal{  
 @Override  
 **public void** sayMyself(){  
 System.*out*.println("大家好，我是猫咪^\_^！");  
 }  
}

最后创建一个测试类，测试通过Cat的类信息获取父类的注解信息：

**public class** InheritedTest{  
 **public static void** main(String[] args){  
 Class<Cat> child = Cat.**class**;  
 MyInherited annotation = child.getAnnotation(MyInherited.**class**);  
 System.*out*.println(annotation.name());  
 }  
}

测试结果如下：



四、注解的原理

1、注解的本质

「java.lang.annotation.Annotation」接口中有这么一句话，用来描述注解。

The common interface extended by all annotation types

所有的注解类型都继承自这个普通的接口（Annotation）

比如 @Override注解：

@Target(ElementType.*METHOD*)  
@Retention(RetentionPolicy.*SOURCE*)  
**public** @**interface** Override {  
}

通过反编译之后：

**public interface** Override **extends** Annotation{}

说明注解类的本质就是实现了Annotation接口的类对象。

一个注解准确意义上来说，只不过是一种特殊的注释而已，如果没有解析它的代码，它可能连注释都不如。

解析注解有两种形式：

1. 编译期直接扫描（RetentionPolicy为SORUCE和CLASS时）

2. 运行期反射（RetentionPolicy为RUNTIME时）

2、运行时注解的解析原理

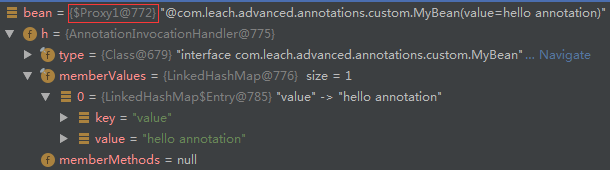
通过反射获取注解时，返回的是Java 运行时生成的动态代理对象$Proxy1，该代理对象就是当前注解类的具体实现类。通过代理对象调用自定义注解（接口）的方法，会最终调用AnnotationInvocationHandler 的invoke 方法。该方法会从memberValues 这个Map 中索引出对应的值。而memberValues 的来源是Java 常量池。

逐步分析：

自定义个注解类和测试类：

@Target({ElementType.*FIELD*,ElementType.*TYPE*,ElementType.*METHOD*})  
@Retention(RetentionPolicy.*RUNTIME*)  
**public** @**interface** MyBean{  
 String value() **default** "bean";  
}

@MyBean("hello annotation")  
**public class** MyBeanTest{  
 **public static void** main(String[] args){  
 System.*setProperty*("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");  
 Class<MyBeanTest> test = MyBeanTest.**class**;  
 MyBean bean = test.getAnnotation(MyBean.**class**);  
 System.*out*.println(bean.value());  
 }  
}

断点运行程序：  
 

可以看见MyBean注解的实例是jvm生成的动态代理类的对象$Proxy1。

这个运行时生成的动态代理对象是可以导出到文件的，方法有两种：

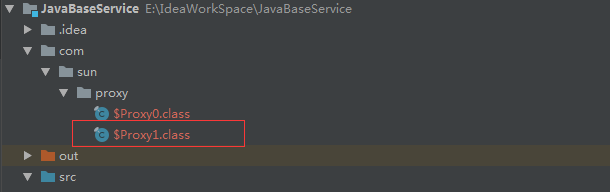
1. 在代码中加入

System.*setProperty*("sun.misc.ProxyGenerator.saveGeneratedFiles","true");

2. 在运行时加入jvm 参数

-Dsun.misc.ProxyGenerator.saveGeneratedFiles=true

我们使用第一种，然后运行程序，如图会在proxy目录中生成$Proxy1.class：



Intellij自带了反编译工具，直接双击打开，得到如下的Java代码:

**public final class** $Proxy1 **extends** Proxy **implements** MyBean {  
 **private static** Method m1;  
 **private static** Method m2;  
 **private static** Method m4;  
 **private static** Method m0;  
 **private static** Method m3;

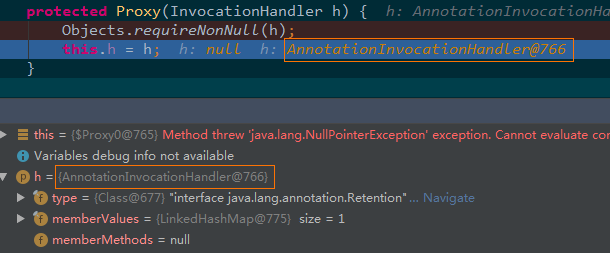
**public** $Proxy1(InvocationHandler var1) **throws** {  
 **super**(var1);  
 }

**public final boolean** equals(Object var1) **throws** { // 省略 … }  
 **public final** String toString() **throws** { // 省略 … }  
 **public final** Class annotationType() **throws** { // 省略 … }  
 **public final int** hashCode() **throws** { // 省略 … }  
  
 **public final** String value() **throws** {  
 **try** {  
 **return** (String)**super**.h.invoke(**this**, m3, (Object[])**null**);  
 } **catch** (RuntimeException | Error var2) {  
 **throw** var2;  
 } **catch** (Throwable var3) {  
 **throw new** UndeclaredThrowableException(var3);  
 }  
 }

}

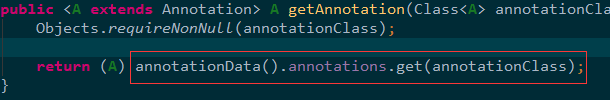
由于自定义注解的本质是继承了Annotation接口的接口，所以$Proxy1做为MyBean的代理实现类，实现了MyBean的value方法和Annotaton 的四个方法。  
 观察value实现方法，我们看到最终是通过h.invoke()方法来执行，而这个h通过构造函数可以追踪到Proxy类的InvocationHandler属性，所以h必定是InvocationHandler接口的实现类。

通过断点可以不难发现该实现类为AnnotationInvocationHandler，如图所示：



通过源码来确定h为AnnotationInvocationHandler:

通过测试代码定位：getAnnotation()，如下：



annotationData()会调用createAnnotationData();

createAnnotationData方法调用AnnotationParser.parseAnnontations();

parseAnnotations方法中调用parseAnnontations2();

parseAnnotations2方法中调用parseAnnotation2;

parseAnnotation2方法最终会调用annontationForMap方法返回结果;

annotationForMap的实现如下：

**public static** Annotation annotationForMap(**final** Class<? **extends** Annotation> var0, **final** Map<String, Object> var1) {

**return** (Annotation)AccessController.doPrivileged(

**new** PrivilegedAction<Annotation>() {  
 **public** Annotation run() {  
 **return** (Annotation)Proxy.newProxyInstance(

var0.getClassLoader(),

**new** Class[]{var0},

**new** AnnotationInvocationHandler(var0, var1));  
 }  
 });  
}

可以看到annotationForMap中使用动态代理AnnotationInvocationHandler生成了代理实例。

五、JavaAPI内置注解

|  |  |
| --- | --- |
| 内置注解 | 说明 |
| @Override | 标记方法为继承或实现的的方法 |
| @Deprecated | 标记当前的类或者方法或者字段等已经不再被推荐使用了，可能下一次的 JDK 版本就会删除。 |
| @SuppressWarnings | 主要用来压制 java 的警告 |

六、自定义注解

1、自定义注解的定义和使用

* 自定义注解使用@interface关键字定义，同时使用元注解注解；
* 自定义注解参数成员只能用public或默认（default）这两个访问权修饰；
* 参数成员类型可以是八种基本类型，String,Enum,Class,annotation以及这些类型的数组；
* 自定义注解可以没有定义成员，不过这样的注解没啥用。

示例：

@Target({ElementType.*FIELD*,ElementType.*TYPE*,ElementType.*METHOD*})  
@Retention(RetentionPolicy.*RUNTIME*)  
**public** @**interface** MyBean{  
 String value() **default** "bean";  
}

2、自定义注解的解析