**注解**

## 一、定义概述

**1 Annotation 中文译过来就是注解、标释的意思**，在 Java 中注解是一个很重要的知识点，但经常还是有点让新手不容易理解。我个人认为，比较糟糕的技术文档主要特征之一就是：用专业名词来介绍专业名词。

比如：Java 注解用于为 Java 代码提供元数据。作为元数据，注解不直接影响你的代码执行，但也有一些类型的注解实际上可以用于这一目的。Java 注解是从 Java5 开始添加到 Java 的。

## 二、 注解语法

因为平常开发少见，相信有不少的人员会认为注解的地位不高。其实同 classs 和 interface 一样，注解也属于一种类型。它是在 Java SE 5.0 版本中开始引入的概念。

**1 注解的定义**

注解通过 @interface 关键字进行定义。

public @interface TestAnnotation {

}

它的形式跟接口很类似，不过前面多了一个 @ 符号。上面的代码就创建了一个名字为 TestAnnotaion 的注解。你可以简单理解为创建了一张名字为 TestAnnotation 的标签。

**2 注解的应用**

上面创建了一个注解，那么注解的的使用方法是什么呢。

@TestAnnotation

public class Test {

}

创建一个类 Test,然后在类定义的地方加上 @TestAnnotation 就可以用 TestAnnotation 注解这个类了。你可以简单理解为将 TestAnnotation 这张标签贴到 Test 这个类上面。不过，要想注解能够正常工作，还需要介绍一下一个新的概念那就是元注解。

**3 元注解**

元注解是可以注解到注解上的注解，或者说元注解是一种基本注解，但是它能够应用到其它的注解上面。如果难于理解的话，你可以这样理解。元注解也是一张标签，但是它是一张特殊的标签，它的作用和目的就是给其他普通的标签进行解释说明的。

**元标签有 @Retention、@Documented、@Target、@Inherited、@Repeatable 5 种。**

**（1）@Retention**

Retention 的英文意为保留期的意思。当 @Retention 应用到一个注解上的时候，它解释说明了这个注解的的存活时间。

它的取值如下：

- RetentionPolicy.SOURCE 注解只在源码阶段保留，在编译器进行编译时它将被丢弃忽视。

- RetentionPolicy.CLASS 注解只被保留到编译进行的时候，它并不会被加载到 JVM 中。

- RetentionPolicy.RUNTIME 注解可以保留到程序运行的时候，它会被加载进入到 JVM 中，所以在程序运行时可以获取到它们。

我们可以这样的方式来加深理解，@Retention 去给一张标签解释的时候，它指定了这张标签张贴的时间。@Retention 相当于给一张标签上面盖了一张时间戳，时间戳指明了标签张贴的时间周期。

@Retention(RetentionPolicy.RUNTIME)

public @interface TestAnnotation {

}

上面的代码中，我们指定 TestAnnotation 可以在程序运行周期被获取到，因此它的生命周期非常的长。

**（2）@Documented**

顾名思义，这个元注解肯定是和文档有关。它的作用是能够将注解中的元素包含到 Javadoc 中去。

**（3）@Target**

Target 是目标的意思，@Target 指定了注解运用的地方。

你可以这样理解，当一个注解被 @Target 注解时，这个注解就被限定了运用的场景。

类比到标签，原本标签是你想张贴到哪个地方就到哪个地方，但是因为 @Target 的存在，它张贴的地方就非常具体了，比如只能张贴到方法上、类上、方法参数上等等。@Target 有下面的取值

ElementType.ANNOTATION\_TYPE 可以给一个注解进行注解

ElementType.CONSTRUCTOR 可以给构造方法进行注解

ElementType.FIELD 可以给属性进行注解

ElementType.LOCAL\_VARIABLE 可以给局部变量进行注解

ElementType.METHOD 可以给方法进行注解

ElementType.PACKAGE 可以给一个包进行注解

ElementType.PARAMETER 可以给一个方法内的参数进行注解

ElementType.TYPE 可以给一个类型进行注解，比如类、接口、枚举

**（4）@Inherited**

Inherited 是继承的意思，但是它并不是说注解本身可以继承，而是说如果一个超类被 @Inherited 注解过的注解进行注解的话，那么如果它的子类没有被任何注解应用的话，那么这个子类就继承了超类的注解。说的比较抽象。代码来解释。

@Inherited

@Retention(RetentionPolicy.RUNTIME)

@interface Test {}

@Test

public class A {}

public class B extends A {}

注解 Test 被 @Inherited 修饰，之后类 A 被 Test 注解，类 B 继承 A,类 B 也拥有 Test 这个注解。可以这样理解：老子非常有钱，所以人们给他贴了一张标签叫做富豪。老子的儿子长大后，只要没有和老子断绝父子关系，虽然别人没有给他贴标签，但是他自然也是富豪。老子的孙子长大了，自然也是富豪。这就是人们口中戏称的富一代，富二代，富三代。虽然叫法不同，好像好多个标签，但其实事情的本质也就是他们有一张共同的标签，也就是老子身上的那张富豪的标签。

**（5）@Repeatable**

Repeatable 自然是可重复的意思。@Repeatable 是 Java 1.8 才加进来的，所以算是一个新的特性。什么样的注解会多次应用呢？通常是注解的值可以同时取多个。举个例子，一个人他既是程序员又是产品经理,同时他还是个画家。

@interface Persons {

Person[] value();

}

@Repeatable(Persons.class)

@interface Person{

String role default "";

}

@Person(role="artist")

@Person(role="coder")

@Person(role="PM")

public class SuperMan{

}

注意上面的代码，@Repeatable 注解了 Person。而 @Repeatable 后面括号中的类相当于一个容器注解。什么是容器注解呢？就是用来存放其它注解的地方。它本身也是一个注解。

我们再看看代码中的相关容器注解。

@interface Persons {

Person[] value();

}

按照规定，它里面必须要有一个 value 的属性，属性类型是一个被 @Repeatable 注解过的注解数组，注意它是数组。如果不好理解的话，可以这样理解。Persons 是一张总的标签，上面贴满了 Person 这种同类型但内容不一样的标签。把 Persons 给一个 SuperMan 贴上，相当于同时给他贴了程序员、产品经理、画家的标签。我们可能对于 @Person(role=”PM”) 括号里面的内容感兴趣，它其实就是给 Person 这个注解的 role 属性赋值为 PM ，大家不明白正常，马上就讲到注解的属性这一块。

**4 注解的属性**

注解的属性也叫做成员变量。注解只有成员变量，没有方法。注解的成员变量在注解的定义中以“**无形参的方法**”形式来声明，其方法名定义了该成员变量的名字，其返回值定义了该成员变量的类型。

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface TestAnnotation {

int id();

String msg();

}

上面代码定义了 TestAnnotation 这个注解中拥有 id 和 msg 两个属性。在使用的时候，我们应该给它们进行赋值。赋值的方式是在注解的括号内以 value=”” 形式，多个属性之前用 ，隔开。

@TestAnnotation(id=3,msg="hello annotation")

public class Test {

}

需要注意的是，在注解中定义属性时它的类型必须是 8 种基本数据类型外加 类、接口、注解及它们的数组。注解中属性可以有默认值，默认值需要用 default 关键值指定。比如：

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

public @interface TestAnnotation {

public int id() default -1;

public String msg() default "Hi";

}

TestAnnotation 中 id 属性默认值为 -1，msg 属性默认值为 Hi。

它可以这样应用。

@TestAnnotation()

public class Test {}

因为有默认值，所以无需要再在 @TestAnnotation 后面的括号里面进行赋值了，这一步可以省略。

另外，还有一种情况。如果一个注解内仅仅只有一个名字为 value 的属性时，应用这个注解时可以直接接属性值填写到括号内。

public @interface Check {

String value();

}

上面代码中，Check 这个注解只有 value 这个属性。所以可以这样应用。

@Check("hi")

int a;

这和下面的效果是一样的

@Check(value="hi")

int a;

最后，还需要注意的一种情况是一个注解没有任何属性。比如

public @interface Perform {}

那么在应用这个注解的时候，括号都可以省略。

@Perform

public void testMethod(){}

## 三、Java 预置的注解

**1 @Deprecated**

这个元素是用来标记过时的元素，想必大家在日常开发中经常碰到。编译器在编译阶段遇到这个注解时会发出提醒警告，告诉开发者正在调用一个过时的元素比如过时的方法、过时的类、过时的成员变量。

public class Hero {

@Deprecated

public void say(){

System.out.println("Noting has to say!");

}

public void speak(){

System.out.println("I have a dream!");

}

}

定义了一个 Hero 类，它有两个方法 say() 和 speak() ，其中 say() 被 @Deprecated 注解。然后我们在 IDE 中分别调用它们。可以看到，say() 方法上面被一条直线划了一条，这其实就是编译器识别后的提醒效果。

**2 @Override**

这个大家应该很熟悉了，提示子类要复写父类中被 @Override 修饰的方法

**3 @SuppressWarnings**

阻止警告的意思。之前说过调用被 @Deprecated 注解的方法后，编译器会警告提醒，而有时候开发者会忽略这种警告，他们可以在调用的地方通过 @SuppressWarnings 达到目的。

@SuppressWarnings("deprecation")

public void test1(){

Hero hero = new Hero();

hero.say();

hero.speak();

}

**4 @SafeVarargs**

参数安全类型注解。它的目的是提醒开发者不要用参数做一些不安全的操作,它的存在会阻止编译器产生 unchecked 这样的警告。它是在 Java 1.7 的版本中加入的。

**5 @SafeVarargs // Not actually safe!**

static void m(List<String>... stringLists) {

Object[] array = stringLists;

List<Integer> tmpList = Arrays.asList(42);

array[0] = tmpList; // Semantically invalid, but compiles without warnings

String s = stringLists[0].get(0); // Oh no, ClassCastException at runtime!

}

上面的代码中，编译阶段不会报错，但是运行时会抛出 ClassCastException 这个异常，所以它虽然告诉开发者要妥善处理，但是开发者自己还是搞砸了。

Java 官方文档说，未来的版本会授权编译器对这种不安全的操作产生错误警告。

**6 @FunctionalInterface**

函数式接口注解，这个是 Java 1.8 版本引入的新特性。函数式编程很火，所以 Java 8 也及时添加了这个特性。函数式接口 (Functional Interface) 就是一个具有一个方法的普通接口。

@FunctionalInterface

public interface Runnable {

/\*\*

\* When an object implementing interface <code>Runnable</code> is used

\* to create a thread, starting the thread causes the object's

\* <code>run</code> method to be called in that separately executing

\* thread.

\* <p>

\* The general contract of the method <code>run</code> is that it may

\* take any action whatsoever.

\*

\* @see java.lang.Thread#run()

\*/

public abstract void run();

}

我们进行线程开发中常用的 Runnable 就是一个典型的函数式接口，上面源码可以看到它就被 @FunctionalInterface 注解。可能有人会疑惑，函数式接口标记有什么用，这个原因是函数式接口可以很容易转换为 Lambda 表达式。这是另外的主题了，有兴趣的同学请自己搜索相关知识点学习。

## 四、注解的提取

1 博文前面的部分讲了注解的基本语法，现在是时候检测我们所学的内容了。我通过用标签来比作注解，前面的内容是讲怎么写注解，然后贴到哪个地方去，而现在我们要做的工作就是检阅这些标签内容。 形象的比喻就是你把这些注解标签在合适的时候撕下来，然后检阅上面的内容信息。要想正确检阅注解，离不开一个手段，那就是反射。

**2 注解与反射。**

注解通过反射获取。首先可以通过 Class 对象的 isAnnotationPresent() 方法判断它是否应用了某个注解

public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass) {}

然后通过 getAnnotation() 方法来获取 Annotation 对象。

public <A extends Annotation> A getAnnotation(Class<A> annotationClass) {}

或者是 getAnnotations() 方法。

public Annotation[] getAnnotations() {}

前一种方法返回指定类型的注解，后一种方法返回注解到这个元素上的所有注解。

如果获取到的 Annotation 如果不为 null，则就可以调用它们的属性方法了。比如

@TestAnnotation()

public class Test {

public static void main(String[] args) {

boolean hasAnnotation = Test.class.isAnnotationPresent(TestAnnotation.class);

if ( hasAnnotation ) {

TestAnnotation testAnnotation = Test.class.getAnnotation(TestAnnotation.class);

System.out.println("id:"+testAnnotation.id());

System.out.println("msg:"+testAnnotation.msg());

}

}

}

程序的运行结果是：

id:-1

msg:

这个正是 TestAnnotation 中 id 和 msg 的默认值。

上面的例子中，只是检阅出了注解在类上的注解，其实属性、方法上的注解照样是可以的。同样还是要假手于反射。

@TestAnnotation(msg="hello")

public class Test {

@Check(value="hi")

int a;

@Performq

public void testMethod(){}

@SuppressWarnings("deprecation")

public void test1(){

Hero hero = new Hero();

hero.say();

hero.speak();

}

public static void main(String[] args) {

boolean hasAnnotation = Test.class.isAnnotationPresent(TestAnnotation.class);

if ( hasAnnotation ) {

TestAnnotation testAnnotation = Test.class.getAnnotation(TestAnnotation.class);

//获取类的注解

System.out.println("id:"+testAnnotation.id());

System.out.println("msg:"+testAnnotation.msg());

}

try {

Field a = Test.class.getDeclaredField("a");

a.setAccessible(true);

//获取一个成员变量上的注解

Check check = a.getAnnotation(Check.class);

if ( check != null ) {

System.out.println("check value:"+check.value());

}

Method testMethod = Test.class.getDeclaredMethod("testMethod");

if ( testMethod != null ) {

// 获取方法中的注解

Annotation[] ans = testMethod.getAnnotations();

for( int i = 0;i < ans.length;i++) {

System.out.println("method testMethod annotation:"+ans[i].annotationType().getSimpleName());

}

}

} catch (NoSuchFieldException e) {

// TODO Auto-generated catch block

e.printStackTrace();

System.out.println(e.getMessage());

} catch (SecurityException e) {

// TODO Auto-generated catch block

e.printStackTrace();

System.out.println(e.getMessage());

} catch (NoSuchMethodException e) {

// TODO Auto-generated catch block

e.printStackTrace();

System.out.println(e.getMessage());

}

}

}

它们的结果如下：

id:-1

msg:hello

check value:hi

method testMethod annotation:Perform

需要注意的是，如果一个注解要在运行时被成功提取，那么 @Retention(RetentionPolicy.RUNTIME) 是必须的。

## 五、注解的使用场景

1 提供信息给编译器： 编译器可以利用注解来探测错误和警告信息

2 编译阶段时的处理： 软件工具可以用来利用注解信息来生成代码、Html文档或者做其它相应处理。

3 运行时的处理： 某些注解可以在程序运行的时候接受代码的提取

## 六、实际例子

**1 @Jiecha例子**

—— 程序员 A : 我写了一个类，它的名字叫做 NoBug，因为它所有的方法都没有错误。

—— 我：自信是好事，不过为了防止意外，让我测试一下如何？

—— 程序员 A: 怎么测试？

—— 我：把你写的代码的方法都加上 @Jiecha 这个注解就好了。

—— 程序员 A: 好的。

NoBug.java

package ceshi;

import ceshi.Jiecha;

public class NoBug {

@Jiecha

public void suanShu(){

System.out.println("1234567890");

}

@Jiecha

public void jiafa(){

System.out.println("1+1="+1+1);

}

@Jiecha

public void jiefa(){

System.out.println("1-1="+(1-1));

}

@Jiecha

public void chengfa(){

System.out.println("3 x 5="+ 3\*5);

}

@Jiecha

public void chufa(){

System.out.println("6 / 0="+ 6 / 0);

}

public void ziwojieshao(){

System.out.println("我写的程序没有 bug!");

}

}

上面的代码，有些方法上面运用了 @Jiecha 注解。

这个注解是我写的测试软件框架中定义的注解。

package ceshi;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)

public @interface Jiecha {

}

然后，我再编写一个测试类 TestTool 就可以测试 NoBug 相应的方法了。

package ceshi;

import java.lang.reflect.InvocationTargetException;

import java.lang.reflect.Method;

public class TestTool {

public static void main(String[] args) {

// TODO Auto-generated method stub

NoBug testobj = new NoBug();

Class clazz = testobj.getClass();

Method[] method = clazz.getDeclaredMethods();

//用来记录测试产生的 log 信息

StringBuilder log = new StringBuilder();

// 记录异常的次数

int errornum = 0;

for ( Method m: method ) {

// 只有被 @Jiecha 标注过的方法才进行测试

if ( m.isAnnotationPresent( Jiecha.class )) {

try {

m.setAccessible(true);

m.invoke(testobj, null);

} catch (Exception e) {

// TODO Auto-generated catch block

//e.printStackTrace();

errornum++;

log.append(m.getName());

log.append(" ");

log.append("has error:");

log.append("\n\r caused by ");

//记录测试过程中，发生的异常的名称

log.append(e.getCause().getClass().getSimpleName());

log.append("\n\r");

//记录测试过程中，发生的异常的具体信息

log.append(e.getCause().getMessage());

log.append("\n\r");

}

}

}

log.append(clazz.getSimpleName());

log.append(" has ");

log.append(errornum);

log.append(" error.");

// 生成测试报告

System.out.println(log.toString());

}

}

测试的结果是：

1234567890

1+1=11

1-1=0

3 x 5=15

chufa has error:

caused by ArithmeticException

/ by zero

NoBug has 1 error.

提示 NoBug 类中的 chufa() 这个方法有异常，这个异常名称叫做 ArithmeticException，原因是运算过程中进行了除 0 的操作。所以，NoBug 这个类有 Bug。这样，通过注解我完成了我自己的目的，那就是对别人的代码进行测试。所以，再问我注解什么时候用？我只能告诉你，这取决于你想利用它干什么用。

**2 注解应用实例**

注解运用的地方太多了，因为我是 Android 开发者，所以我接触到的具体例子有下：

**（1）JUnit**

JUnit 这个是一个测试框架，典型使用方法如下：

public class ExampleUnitTest {

@Test

public void addition\_isCorrect() throws Exception {

assertEquals(4, 2 + 2);

}

}

@Test 标记了要进行测试的方法 addition\_isCorrect().

**（2）ButterKnife**

ButterKnife 是 Android 开发中大名鼎鼎的 IOC 框架，它减少了大量重复的代码。

public class MainActivity extends AppCompatActivity {

@BindView(R.id.tv\_test)

TextView mTv;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

ButterKnife.bind(this);

}

}

## 六、最后总结

如果注解难于理解，你就把它类同于标签，标签为了解释事物，注解为了解释代码。

注解的基本语法，创建如同接口，但是多了个 @ 符号。

注解的元注解。

注解的属性。

注解主要给编译器及工具类型的软件用的。

注解的提取需要借助于 Java 的反射技术，反射比较慢，所以注解使用时也需要谨慎计较时间成本。

## 七、注解扫描类

**1 Reflections 通过扫描 classpath，索引元数**据，允许在运行时查询这些元数据，也可以保存收集项目中多个模块的元数据信息。使用Reflections快速扫描指定包下自定义的Controller和RequestMapping两个注解，先去扫描加了@Controller注解的类，接着获取这些类下面加了@RequestMapping注解的方法，然后通过Java的反射invoke方法去调用加了RequestMapping注解的方法并输出注解上的信息。

**2 Maven 项目导入**

<dependency>

<groupId>org.reflections</groupId>

<artifactId>reflections</artifactId>

<version>0.9.10</version>

</dependency>

**3 annotation包下面自定义了两个注解**

Controller.java：

package annotationTest.annotation;

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target(ElementType.TYPE)// 注解会在class字节码文件中存在，在运行时可以通过反射获取到

@Retention(RetentionPolicy.RUNTIME)//定义注解的作用目标\*\*作用范围字段、枚举的常量/方法

@Documented//说明该注解将被包含在javadoc中

public @interface Controller {

String value() default "";

}

**4 RequestMapping.java**

package annotationTest.annotation;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Target({ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

public @interface RequestMapping {

String value() default "";

/\*\*

\* 是否为序列号

\*

\* @return

\*/

boolean id() default false;

/\*\*

\* 字段名称

\*

\* @return

\*/

String name() default "";

/\*\*

\* 字段描述

\*

\* @return

\*/

String description() default "";

}

**5 在model包下面定义了一个存放RequestMapping注解方法的对象**

ExecutorBean.java

package annotationTest.model;

import java.lang.reflect.Method;

public class ExecutorBean {

private Object object;

private Method method;

public Object getObject() {

return object;

}

public void setObject(Object object) {

this.object = object;

}

public Method getMethod() {

return method;

}

public void setMethod(Method method) {

this.method = method;

}

}

**6 service包下面定义了几个类，其中有两个类使用了自定义的Controller注解**

SunService.java

package annotationTest.service;

import annotationTest.annotation.Controller;

import annotationTest.annotation.RequestMapping;

@Controller

public class SunService {

@RequestMapping(id = true, name = "test1", description = "sun测试1", value = "/test1")

public void test1() {

System.out.println("SunService->test1()");

}

@RequestMapping(id = true, name = "test2", description = "sun测试2", value = "/test2")

public void test2() {

System.out.println("SunService->test2()");

}

}

7 MoonService.java

package annotationTest.service;

import annotationTest.annotation.Controller;

import annotationTest.annotation.RequestMapping;

@Controller

public class MoonService {

@RequestMapping(id = true, name = "moon测试3", description = "/test3", value = "/test3")

public void test3() {

System.out.println("MoonService->test3()");

}

@RequestMapping(id = true, name = "moon测试4", description = "/test4", value = "/test4")

public void test4() {

System.out.println("MoonService->test4()");

}

}

**8 Stars.java**

package annotationTest.service;

import annotationTest.annotation.RequestMapping;

public class Stars {

@RequestMapping(id = true, name = "test1", description = "stars测试1", value = "/test1")

public void test1() {

System.out.println("Stars->test1()");

}

}

**9 util包下面定义了一个工具类，来对包进行扫描获取自定义注解的类和方法**

AnnoManageUtil.java

package annotationTest.util;

import java.lang.reflect.Method;

import java.util.HashMap;

import java.util.Map;

import java.util.Set;

import annotationTest.annotation.Controller;

import annotationTest.annotation.RequestMapping;

import annotationTest.model.ExecutorBean;

import org.reflections.Reflections;

public final class AnnoManageUtil {

/\*\*

\* 获取指定文件下面的RequestMapping方法保存在mapp中

\*

\* @param packageName

\* @return

\*/

public static Map<String, ExecutorBean> getRequestMappingMethod(String packageName) {

Reflections reflections = new Reflections(packageName);

Set<Class<?>> classesList = reflections.getTypesAnnotatedWith(Controller.class);

// 存放url和ExecutorBean的对应关系

Map<String, ExecutorBean> mapp = new HashMap<String, ExecutorBean>();

for (Class classes : classesList) {

//得到该类下面的所有方法

Method[] methods = classes.getDeclaredMethods();

for (Method method : methods) {

//得到该类下面的RequestMapping注解

RequestMapping requestMapping = method.getAnnotation(RequestMapping.class);

if (null != requestMapping) {

ExecutorBean executorBean = new ExecutorBean();

try {

executorBean.setObject(classes.newInstance());

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

}

executorBean.setMethod(method);

mapp.put(requestMapping.value(), executorBean);

}

}

}

return mapp;

}

}

**10 test包下面是一个测试的类**

package annotationTest.test;

import java.lang.reflect.InvocationTargetException;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import annotationTest.annotation.Controller;

import annotationTest.annotation.RequestMapping;

import annotationTest.model.ExecutorBean;

import annotationTest.util.AnnoManageUtil;

public class Test {

public static void main(String[] args) {

List<Class<?>> classesList = null;

classesList = AnnoManageUtil.getPackageController("annotationTest.service", Controller.class);

Map<String, ExecutorBean> mmap = new HashMap<String, ExecutorBean>();

AnnoManageUtil.getRequestMappingMethod(classesList, mmap);

ExecutorBean bean = mmap.get("/test1");

try {

bean.getMethod().invoke(bean.getObject());

RequestMapping annotation = bean.getMethod().getAnnotation(RequestMapping.class);

System.out.println("注解名称：" + annotation.name() + "\t注解描述：" + annotation.description());

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (InvocationTargetException e) {

e.printStackTrace();

}

}

}

**11 运行得到**

其他使用 Reflections 可以查询以下元数据信息：

获得某个类型的所有子类型

获得标记了某个注解的所有类型／成员变量，支持注解参数匹配。

使用正则表达式获得所有匹配的资源文件

获得所有特定签名（包括参数，参数注解，返回值）的方法

Reflections 依赖 Google 的 Guava 库和 Javassist 库。

使用注解修饰了类/方法/成员变量等之后,这些注解不会自己生效,必须由这些注解的开发者提供相应的工具来提取并处理注解信息(当然,只有当定义注解时使用了@Retention(RetentionPolicy.RUNTIME)修饰,JVM才会在装载class文件时提取保存在class文件中的注解,该注解才会在运行时可见,这样我们才能够解析).Java使用Annotation接口来代表程序元素前面的注解,该接口是所有注解的父接口。 java5在java.lang.reflect包下新增了 用AnnotatedElement接口代表程序中可以接受注解的程序元素.

AnnotatedElement接口的实现类有：Class(类元素)、Field（类的成员变量元素）、Method(类的方法元素)、Package(包元素)，每一个实现类代表了一个可以接受注解的程序元素类型。

这样, 我们只需要获取到Class、 Method、 Filed等这些实现了AnnotatedElement接口的类的实例,通过该实例对象调用该类中的方法（AnnotatedElement接口中抽象方法的重写） 就可以获取到我们想要的注解信息了。

**获得Class类的实例有三种方法：**

利用对象调用getClass()方法获得Class实例

利用Class类的静态的forName()方法，使用类名获得Class实例

运用.class的方式获得Class实例，如：类名.class

AnnotatedElement接口提供的抽象方法（在该接口的实现类中重写了这些方法）：

<T extends Annotation> T getAnnotation(Class< T> annotationClass)&lt T extends Annotation>为泛型参数声明，表明A的类型只能是Annotation类型或者是Annotation的子类。

功能：返回该程序元素上存在的、指定类型的注解，如果该类型的注解不存在，则返回null

Annotation[] getAnnotations()

功能：返回此元素上存在的所有注解，包括没有显示定义在该元素上的注解（继承得到的）。（如果此元素没有注释，则返回长度为零的数组。）

< T extends Annotation> T getDeclaredAnnotation(Class < T> annotationClass)

功能：这是Java8新增的方法，该方法返回直接修饰该程序元素、指定类型的注解（忽略继承的注解）。如果该类型的注解不存在，返回null.

Annotation[] getDeclaredAnnotations()

功能：返回直接存在于此元素上的所有注解，该方法将忽略继承的注释。（如果没有注释直接存在于此元素上，则返回长度为零的一个数组。）

boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)

功能：判断该程序元素上是否存在指定类型的注解，如果存在则返回true,否则返回false。

&ltT extends Annotation> T[] getAnnotationsByTpye(Class<T> annotationClass)

功能： 因为java8增加了重复注解功能，因此需要使用该方法获得修饰该程序元素、指定类型的多个注解。

<T extends Annotation> T[] getDeclaredAnnotationsByTpye(Class<T>annotationClass)

功能： 因为java8增加了重复注解功能，因此需要使用该方法获得直接修饰该程序元素、指定类型的多个注解。

**Class提供了getMethod()、getField()以及getConstructor()方法(还有其他方法)，这些方法分别获取与方法、域变量以及构造函数相关的信息，这些方法返回Method、Field 以及Constructor类型的对象。**

**12 lombok-@Accessors注解**

1. fluent 一个布尔值。如果为真，pepper的getter就是 pepper()，setter方法就是pepper(T newValue)。并且，除非特别说明，chain默认为真。

2. chain 一个布尔值。如果为真，产生的setter返回的this而不是void。默认是假。

@Accessors(prefix = {"first"}, fluent = true, chain = false) private String firstName; 与生存的get和set方法有关，prefix与定义属性前缀相同时且接下来的字符大写才生效，可以看源码注释或自行尝试；

fluent是决定生成的get/set方法要不要set/get前缀，chain决定set方法是void类型还是返回this