**获取指定包下所有自定义注解并提取注解信息**

原创 2017年06月02日 15:55:06

* 标签：
* [java](http://so.csdn.net/so/search/s.do?q=java&t=blog)
* 1123

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。

**Maven 项目导入**

<dependency>

<groupId>org.reflections</groupId>

<artifactId>reflections</artifactId>

<version>0.9.10</version>

</dependency>

* 1

**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 "";

}

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 "";

}

* 1

**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;

}

}

* 1

**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()");

}

}

* 1

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()");

}

}

* 1

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()");

}

}

* 1

**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;

}

}

* 1

**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();

}

}

}

* 1

运行得到:

http://i.imgur.com/j25ZfYV.png

**其他**

1. 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。

1. 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).
2. 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。
3. 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.
4. 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。
5. 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。
6. 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
7. 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。

版权声明：本文为博主原创文章，未经博主允许不得转载。