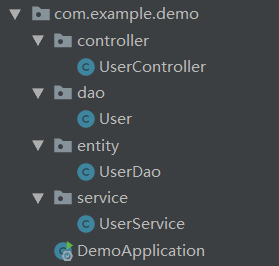
1. 准备工作

首先这里创建了一个简单的springboot项目：



各个类的内容如下所示：

@Data

@AllArgsConstructor

@NoArgsConstructor

public class User {

private Integer id;

private String name;

}

@Component

public class UserDao {

public User findUserById(Integer id) {

if(id > 10) {

return null;

}

return new User(id, "user-" + id);

}

}

@Service

public class UserService {

private final UserDao userDao;

public UserService(UserDao userDao) {

this.userDao = userDao;

}

public User findUserById(Integer id) {

return userDao.findUserById(id);

}

}

@RestController

public class UserController {

private final UserService userService;

public UserController(UserService userService) {

this.userService = userService;

}

@RequestMapping("user/{id}")

public User findUser(@PathVariable("id") Integer id) {

return userService.findUserById(id);

}

}

2. 使用注解执行固定的操作

现在我们已经有了这样的一个简单的web项目了，直接访问localhost:8080/user/6后，显然会得到一个如下的json串

{

"id": 6,

"name": "user-6"

}

但是我们不满足于此，这个项目也未免太简陋了，现在我们就来为它增加一个日志的功能（不要说使用log4j等日志框架，我们的目的是学习自定义注解）

假设我们现在的目的是，在调用controller中的findUser方法前，先在控制台输出一句话。好了那就开始做吧，我们先创建一个annotation包，里面创建我们自定义的注解类KthLog：

package com.example.demo.annotation;

import java.lang.annotation.\*;

@Documented

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface KthLog {

String value() default "";

}

这里注解类上的三个注解称为元注解，其分别代表的含义如下：

@Documented：注解信息会被添加到Java文档中

@Retention：注解的生命周期，表示注解会被保留到什么阶段，可以选择编译阶段、类加载阶段，或运行阶段

@Target：注解作用的位置，ElementType.METHOD表示该注解仅能作用于方法上

然后我们可以把注解添加到方法上：

@KthLog("这是日志内容")

@RequestMapping("user/{id}")

public User findUser(@PathVariable("id") Integer id) {

return userService.findUserById(id);

}

这个注解目前是没有任何作用的，因为我们仅仅是对注解进行了声明，并没有在任何地方来使用这个注解，注解的本质也是一种广义的语法糖，最终还是要利用Java的反射来进行操作

不过Java给我们提供了一个AOP机制，可以对类或方法进行动态的扩展，想较深入地了解这一机制的可以参考我的这篇文章：从源码解读Spring的AOP

我们创建切面类，如下：

@Component

@Aspect

public class KthLogAspect {

@Pointcut("@annotation(com.example.demo.annotation.KthLog)")

private void pointcut() {}

@Before("pointcut() && @annotation(logger)")

public void advice(KthLog logger) {

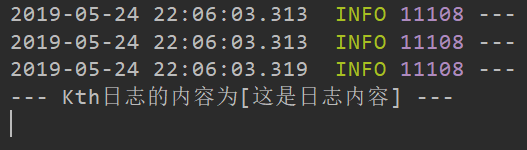
System.out.println("--- Kth日志的内容为[" + logger.value() + "] ---");

}

}

其中@Pointcut声明了切点（这里的切点是我们自定义的注解类），@Before声明了通知内容，在具体的通知中，我们通过@annotation(logger)拿到了自定义的注解对象，所以就能够获取我们在使用注解时赋予的值了。这里如果对于切点和通知等概念不了解的，建议先去查阅一些aop的知识再回来看本文较好，本文更注重于实践，而不是概念的讲解

然后我们现在再来启动web服务，在浏览器上输入localhost:8080/user/6（使用JUnit单元测试也可以），会发现控制台成功输出：



3. 使用注解获取更详细的信息

刚才我们使用自定义注解实现了在方法调用前输出一句日志，但是我们并不知道这是哪个方法、哪个类输出的，如果有两个方法都加上了这个注解，且value的值都一样，那我们该怎么区分这两个方法呢？比如现在我们给UserController类中添加了一个方法：

@RestController

public class UserController {

private final UserService userService;

public UserController(UserService userService) {

this.userService = userService;

}

@KthLog("这是日志内容")

@RequestMapping("user/{id}")

public User findUser(@PathVariable("id") Integer id) {

return userService.findUserById(id);

}

@KthLog("这是日志内容")

@RequestMapping("compared")

public void comparedMethod() { }

}

如果我们调用comparedMethod()方法，显然会得到和刚才一样的输出结果，这时候我们就需要对注解做进一步改造，其实很简单，只需要在切面类的advice()方法中添加一个JoinPoint参数即可，如下：

@Before("pointcut() && @annotation(logger)")

public void advice(JoinPoint joinPoint, KthLog logger) {

System.out.println("注解作用的方法名: " + joinPoint.getSignature().getName());

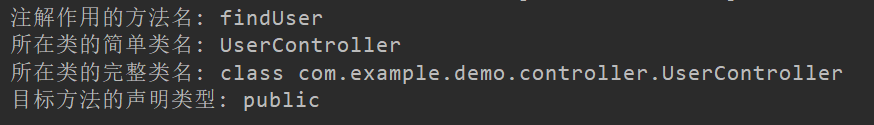
System.out.println("所在类的简单类名: " + joinPoint.getSignature().getDeclaringType().getSimpleName());

System.out.println("所在类的完整类名: " + joinPoint.getSignature().getDeclaringType());

System.out.println("目标方法的声明类型: " + Modifier.toString(joinPoint.getSignature().getModifiers()));

}

然后我们再来执行一遍刚才的流程，看看会输出什么结果：



现在我们再将这些内容放到日志中，顺便修改一下日志的格式，如下：

@Before("pointcut() && @annotation(logger)")

public void advice(JoinPoint joinPoint, KthLog logger) {

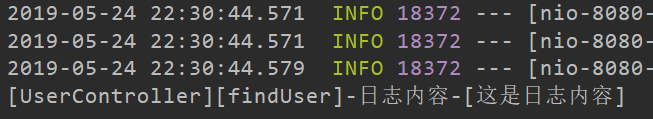
System.out.println("["

+ joinPoint.getSignature().getDeclaringType().getSimpleName()

+ "][" + joinPoint.getSignature().getName()

+ "]-日志内容-[" + logger.value() + "]");

}



4. 使用注解修改参数和返回值

我们把之前添加的compare()方法删去，现在我们的注解需要对方法的参数作出修改，以findUser()方法为例，假设我们传入的用户id是从1开始计数，后端则是从0开始计数，我们的@KthLog注解的开发者喜欢“多管闲事”，想要帮助其他人减轻一点压力，那该怎么做呢？

在这个应用场景中，我们需要做的有两件事：将传入的id减1，给返回的user类中的id加1。这就涉及到如何拿到参数的问题。因为我们需要管理方法执行前和执行后的操作，所以我们使用@Around环绕注解，如下：

@Around("pointcut() && @annotation(logger)")

public Object advice(ProceedingJoinPoint joinPoint, KthLog logger) {

System.out.println("["

+ joinPoint.getSignature().getDeclaringType().getSimpleName()

+ "][" + joinPoint.getSignature().getName()

+ "]-日志内容-[" + logger.value() + "]");

Object result = null;

try {

result = joinPoint.proceed();

} catch (Throwable throwable) {

throwable.printStackTrace();

}

return result;

}

这里除了将@Before改为@Around之外，还将参数中的JoinPoint改为了ProceedingJoinPoint，不过不用担心，JoinPoint能做的ProceedingJoinPoint都能做。这里通过调用proceed()方法，执行了实际的操作，并获取到了返回值，那么接下来对于返回值的操作相信就不用我再多说了，现在问题就是如何获取到参数

ProceedingJoinPoint继承了JoinPoint接口，在JoinPoint中，存在一个getArgs()方法，用于获取方法参数，返回的是一个Object数组，与之匹配的则是proceed(args)方法，这两个方法结合起来，就能够实现我们的目的：

@Around("pointcut() && @annotation(logger)")

public Object advice(ProceedingJoinPoint joinPoint, KthLog logger) {

System.out.println("["

+ joinPoint.getSignature().getDeclaringType().getSimpleName()

+ "][" + joinPoint.getSignature().getName()

+ "]-日志内容-[" + logger.value() + "]");

Object result = null;

Object[] args = joinPoint.getArgs();

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

if(args[i] instanceof Integer) {

args[i] = (Integer)args[i] - 1;

break;

}

}

try {

result = joinPoint.proceed(args);

} catch (Throwable throwable) {

throwable.printStackTrace();

}

if(result instanceof User) {

User user = (User) result;

user.setId(user.getId() + 1);

return user;

}

return result;

}

这里为了代码的鲁棒性做了两次参数类型校验，接着我们重新执行之前的测试，这里为了让结果更明显，我们在UserDao处添加一些输出，来显示实际执行的参数和返回的值各自是什么：

@Component

public class UserDao {

public User findUserById(Integer id) {

System.out.println("查询id为[" + id + "]的用户");

if(id > 10) {

return null;

}

User user = new User(id, "user-" + id);

System.out.println("返回的用户为[" + user.toString() + "]");

return user;

}

}

现在我们访问http://localhost:8080/user/6，来看控制台打印的结果：

我们发现在url上输入的6，在后端被转换成了5，最终查询的用户也是id为5的用户，说明我们参数转换成功了，然后我们来看浏览器得到的响应结果：

返回的用户id是6，而不是后端查询的5，说明我们对返回值的修改也成功了

5. 总结

在Web项目（这里特指Spring项目）中使用自定义注解开发，其原理还是依赖于Spring的AOP机制，这一点就与我们普通的Java项目有所区别。当然，如果是开发其他框架而需要使用自定义注解时，则需要自己实现一套机制，不过原理本质上都是大同小异，无非是将一些模板操作进行了封装

通过自定义的注解，我们不仅能够在方法执行前后进行扩展，同时还可以获取到作用方法的方法名，所在类等信息，更重要的是还能够修改参数和返回值，这几点应用下来基本就囊括了绝大部分自定义注解的功能。了解到这里，完全就能够自己动手来写一个自定义注解来简化我们的项目

————————————————

版权声明：本文为CSDN博主「E\_Mlls」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/qq\_37435078/article/details/90523309