https://blog.csdn.net/SzeLH/article/details/78870476

**2017.12.22 自定义注解、AOP、拦截器**

2017年12月22日 10:27:30 [SzeLH](https://me.csdn.net/SzeLH" \t "_blank) 阅读数：1075

一、自定义注解(AOP切面位置)   
Annotation:注解   
创建Annotation类

1.@Target   
@Target说明了Annotation所修饰的对象范围，用@Target更准确地指出修饰的目标。   
取值(ElementType)有：   
　　　　1.CONSTRUCTOR:用于描述构造器   
　　　　2.FIELD:用于描述域   
　　　　3.LOCAL\_VARIABLE:用于描述局部变量   
　　　　4.METHOD:用于描述方法   
　　　　5.PACKAGE:用于描述包   
　　　　6.PARAMETER:用于描述参数   
　　　　7.TYPE:用于描述类、接口(包括注解类型) 或enum声明

2.@Retention   
@Retention定义了该Annotation被保留的时间长短。   
取值（RetentionPoicy）有：   
　　　　1.SOURCE:在源文件中有效（即源文件保留）   
　　　　2.CLASS:在class文件中有效（即class保留）   
　　　　3.RUNTIME:在运行时有效（即运行时保留）

自定义注解：

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

public @interface xxxxx{ //注解名

}

* 1
* 2
* 3
* 4

二、AOP   
1.依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

* 1
* 2
* 3
* 4

2.拦截类   
①

@Aspect

@Component

public class OperateAop {

private static final Logger logger = LoggerFactory.getLogger(OperateAop.class);

//定义切面、拦截规则

@Pointcut("@annotation(com.commit.server.common.aop.Authority)") //Authority是自定义的标签

public void authority(){

}

@Before("authority()")

public void doAuthority(JoinPoint Point){

Method method = ((MethodSignature) point.getSignature()).getMethod(); //获取切面的方法

Object[] args = point.getArgs(); //获取参数,参数没有类型，都是Object对象，要判断。

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

效果：在加上@Authority注解的地方会被拦截，在方法执行前调用doAuthority方法。

②当有两个Aspect类时，切口调用两个Aspect中方法的顺序是不确定的，可以通过@Order来标识先后顺序。

③在同一个 aspect 类中，针对同一个 pointcut，定义了两个相同的 advice(比如，定义了两个 @Before)，那么这两个 advice 的执行顺序是无法确定的，哪怕给这两个 advice 添加了 @Order 这个注解，也不行。

④对于@Around这个advice，不管它有没有返回值，但是必须要方法内部调用一下 point.proceed();否则，Controller 中的接口将没有机会被执行，从而也导致了 @Before这个advice不会被触发。通过 point.proceed()来区分方法执行Before和After

三、拦截器   
自定义拦截器：   
1、创建自己的拦截器类并实现 HandlerInterceptor 接口。 (@Component注解)

@Component

public class AuthorityInterceptor implements HandlerInterceptor {

private static final Logger logger = LoggerFactory.getLogger(AuthorityInterceptor.class);

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

throws Exception {

logger.info("{}",request.getAttribute("user"));

if(request.getAttribute("user") == null){

logger.info("未登录");

throw new AuthorityException(ExceptionConstant.UNLOGIN);

}

return true;// 只有返回true才会继续向下执行，返回false取消当前请求

}

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) throws Exception {

} //请求处理之后进行调用，但是在视图被渲染之前（Controller方法调用之后）

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

throws Exception {

} //在整个请求结束之后被调用，也就是在DispatcherServlet 渲染了对应的视图之后执行（主要是用于进行资源清理工作）

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

2、创建一个Java类继承WebMvcConfigurerAdapter，并重写 addInterceptors 方法。 (@Configuration注解)   
3、实例化自定义的拦截器，然后将对像手动添加到拦截器链中（在addInterceptors方法中添加）。

@Configuration

public class AuthorityConfigurer extends WebMvcConfigurerAdapter {

@Override

public void addInterceptors(InterceptorRegistry registry) {

//// 多个拦截器组成一个拦截器链

//// addPathPatterns 用于添加拦截规则

//// excludePathPatterns 用户排除拦截

registry.addInterceptor(new AuthorityInterceptor()).

addPathPatterns("/\*\*").

excludePathPatterns("/login/login").

excludePathPatterns("/login/change").

excludePathPatterns("/login/create");

super.addInterceptors(registry);

}

}