[**Spring boot中使用aop详解**](https://www.cnblogs.com/bigben0123/p/7779357.html)

版权声明：本文为博主武伟峰原创文章，转载请注明地址http://blog.csdn.net/tianyaleixiaowu。

aop是spring的两大功能模块之一，功能非常强大，为解耦提供了非常优秀的解决方案。

现在就以springboot中aop的使用来了解一下aop。

**一：使用aop来完成全局请求日志处理**

创建一个springboot的web项目，勾选aop，pom如下：

**[html]** [view plain](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147) [copy](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

[print?](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4. <modelVersion>4.0.0</modelVersion>
6. <groupId>com.example</groupId>
7. <artifactId>testaop</artifactId>
8. <version>0.0.1-SNAPSHOT</version>
9. <packaging>jar</packaging>
11. <name>testaop</name>
12. <description>Demo project for Spring Boot</description>
14. <parent>
15. <groupId>org.springframework.boot</groupId>
16. <artifactId>spring-boot-starter-parent</artifactId>
17. <version>1.5.3.RELEASE</version>
18. <relativePath/> <!-- lookup parent from repository -->
19. </parent>
21. <properties>
22. <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
23. <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
24. <java.version>1.8</java.version>
25. </properties>
27. <dependencies>
28. <dependency>
29. <groupId>org.springframework.boot</groupId>
30. <artifactId>spring-boot-starter-aop</artifactId>
31. </dependency>
32. <dependency>
33. <groupId>org.springframework.boot</groupId>
34. <artifactId>spring-boot-starter-web</artifactId>
35. </dependency>
37. <dependency>
38. <groupId>org.springframework.boot</groupId>
39. <artifactId>spring-boot-starter-test</artifactId>
40. <scope>test</scope>
41. </dependency>
42. </dependencies>
44. <build>
45. <plugins>
46. <plugin>
47. <groupId>org.springframework.boot</groupId>
48. <artifactId>spring-boot-maven-plugin</artifactId>
49. </plugin>
50. </plugins>
51. </build>

54. </project>

创建个controller

**[java]** [view plain](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147) [copy](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

[print?](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

1. **package com.example.controller;**
3. **import org.springframework.web.bind.annotation.RequestMapping;**
4. **import org.springframework.web.bind.annotation.RestController;**
6. /\*\*
7. \* Created by wuwf on 17/4/27.
8. \*
9. \*/
10. @RestController
11. **public class FirstController {**
13. @RequestMapping("/first")
14. **public Object first() {**
15. **return "first controller";**
16. }
18. @RequestMapping("/doError")
19. **public Object error() {**
20. **return 1 / 0;**
21. }
22. }

创建一个aspect切面类

**[java]** [view plain](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147) [copy](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

[print?](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

1. **package com.example.aop;**
3. **import org.aspectj.lang.JoinPoint;**
4. **import org.aspectj.lang.ProceedingJoinPoint;**
5. **import org.aspectj.lang.annotation.\*;**
6. **import org.springframework.stereotype.Component;**
7. **import org.springframework.web.context.request.RequestContextHolder;**
8. **import org.springframework.web.context.request.ServletRequestAttributes;**
10. **import javax.servlet.http.HttpServletRequest;**
11. **import java.util.Arrays;**
13. /\*\*
14. \* Created by wuwf on 17/4/27.
15. \* 日志切面
16. \*/
17. @Aspect
18. @Component
19. **public class LogAspect {**
20. @Pointcut("execution(public \* com.example.controller.\*.\*(..))")
21. **public void webLog(){}**
23. @Before("webLog()")
24. **public void deBefore(JoinPoint joinPoint) throws Throwable {**
25. // 接收到请求，记录请求内容
26. ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();
27. HttpServletRequest request = attributes.getRequest();
28. // 记录下请求内容
29. System.out.println("URL : " + request.getRequestURL().toString());
30. System.out.println("HTTP\_METHOD : " + request.getMethod());
31. System.out.println("IP : " + request.getRemoteAddr());
32. System.out.println("CLASS\_METHOD : " + joinPoint.getSignature().getDeclaringTypeName() + "." + joinPoint.getSignature().getName());
33. System.out.println("ARGS : " + Arrays.toString(joinPoint.getArgs()));
35. }
37. @AfterReturning(returning = "ret", pointcut = "webLog()")
38. **public void doAfterReturning(Object ret) throws Throwable {**
39. // 处理完请求，返回内容
40. System.out.println("方法的返回值 : " + ret);
41. }
43. //后置异常通知
44. @AfterThrowing("webLog()")
45. **public void throwss(JoinPoint jp){**
46. System.out.println("方法异常时执行.....");
47. }
49. //后置最终通知,final增强，不管是抛出异常或者正常退出都会执行
50. @After("webLog()")
51. **public void after(JoinPoint jp){**
52. System.out.println("方法最后执行.....");
53. }
55. //环绕通知,环绕增强，相当于MethodInterceptor
56. @Around("webLog()")
57. **public Object arround(ProceedingJoinPoint pjp) {**
58. System.out.println("方法环绕start.....");
59. **try {**
60. Object o =  pjp.proceed();
61. System.out.println("方法环绕proceed，结果是 :" + o);
62. **return o;**
63. } **catch (Throwable e) {**
64. e.printStackTrace();
65. **return null;**
66. }
67. }
68. }

启动项目

**模拟正常执行的情况，访问http://localhost:8080/first，看控制台结果：**

方法环绕start.....  
URL : http://localhost:8080/first  
HTTP\_METHOD : GET  
IP : 0:0:0:0:0:0:0:1  
CLASS\_METHOD : com.example.controller.FirstController.first  
ARGS : []  
方法环绕proceed，结果是 :first controller  
方法最后执行.....  
方法的返回值 : first controller

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*分割线\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**模拟出现异常时的情况，访问http://localhost:8080/doError，看控制台结果：**  
方法环绕start.....  
URL : http://localhost:8080/doError  
HTTP\_METHOD : GET  
IP : 0:0:0:0:0:0:0:1  
CLASS\_METHOD : com.example.controller.FirstController.error  
ARGS : []  
java.lang.ArithmeticException: / by zero

......

方法最后执行.....  
方法的返回值 : null

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*分割线\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

通过上面的简单的例子，可以看到aop的执行顺序。知道了顺序后，就可以在相应的位置做切面处理了。

**二： 切面方法说明**

@Aspect

作用是把当前类标识为一个切面供容器读取

@Before  
标识一个前置增强方法，相当于BeforeAdvice的功能

@AfterReturning

后置增强，相当于AfterReturningAdvice，方法退出时执行

@AfterThrowing

异常抛出增强，相当于ThrowsAdvice

@After

final增强，不管是抛出异常或者正常退出都会执行

@Around

环绕增强，相当于MethodInterceptor

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*分割线\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

各方法参数说明：

**除了@Around外，每个方法里都可以加或者不加参数JoinPoint，如果有用JoinPoint的地方就加，不加也可以，JoinPoint里包含了类名、被切面的方法名，参数等属性，可供读取使用。@Around参数必须为ProceedingJoinPoint，pjp.proceed相应于执行被切面的方法。@AfterReturning方法里，可以加returning = “XXX”，XXX即为在controller里方法的返回值，本例中的返回值是“first controller”。@AfterThrowing方法里，可以加throwing = "XXX"，供读取异常信息，**如本例中可以改为：

**[java]** [view plain](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147) [copy](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

[print?](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

1. //后置异常通知
2. @AfterThrowing(throwing = "ex", pointcut = "webLog()")
3. **public void throwss(JoinPoint jp, Exception ex){**
4. System.out.println("方法异常时执行.....");
5. }

一般常用的有before和afterReturn组合，或者单独使用Around，即可获取方法开始前和结束后的切面。

**三：关于切面PointCut的切入点**

**execution切点函数**

execution函数用于匹配方法执行的连接点，语法为：

execution(方法修饰符(可选)  返回类型  方法名  参数  异常模式(可选))

参数部分允许使用通配符：

\*  匹配任意字符，但只能匹配一个元素

.. 匹配任意字符，可以匹配任意多个元素，表示类时，必须和\*联合使用

+  必须跟在类名后面，如Horseman+，表示类本身和继承或扩展指定类的所有类

参考：http://blog.csdn.net/autfish/article/details/51184405

除了execution()，Spring中还支持其他多个函数，这里列出名称和简单介绍，以方便根据需要进行更详细的查询

 @annotation()

表示标注了指定注解的目标类方法

例如 @annotation(org.springframework.transaction.annotation.Transactional) 表示标注了@Transactional的方法

args()

通过目标类方法的参数类型指定切点

例如 args(String) 表示有且仅有一个String型参数的方法

@args()

通过目标类参数的对象类型是否标注了指定注解指定切点

如 @args(org.springframework.stereotype.Service) 表示有且仅有一个标注了@Service的类参数的方法

within()

通过类名指定切点

如 with(examples.chap03.Horseman) 表示Horseman的所有方法

target()

通过类名指定，同时包含所有子类

如 target(examples.chap03.Horseman)  且Elephantman extends Horseman，则两个类的所有方法都匹配

@within()

匹配标注了指定注解的类及其所有子类

如 @within(org.springframework.stereotype.Service) 给Horseman加上@Service标注，则Horseman和Elephantman 的所有方法都匹配

@target()

所有标注了指定注解的类

如 @target(org.springframework.stereotype.Service) 表示所有标注了@Service的类的所有方法

 this()

大部分时候和target()相同，区别是this是在运行时生成代理类后，才判断代理类与指定的对象类型是否匹配

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*分割线\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**逻辑运算符**

表达式可由多个切点函数通过逻辑运算组成

 &&

与操作，求交集，也可以写成and

例如 execution(\* chop(..)) && target(Horseman)  表示Horseman及其子类的chop方法

 ||

或操作，求并集，也可以写成or

例如 execution(\* chop(..)) || args(String)  表示名称为chop的方法或者有一个String型参数的方法

!

非操作，求反集，也可以写成not

例如 execution(\* chop(..)) and !args(String)  表示名称为chop的方法但是不能是只有一个String型参数的方法

execution常用于匹配特定的方法，如update时怎么处理，或者匹配某些类，如所有的controller类，是一种范围较大的切面方式，多用于日志或者事务处理等。

其他的几个用法各有千秋，视情况而选择。

以上标红的比较常用。下面来看annotation的。

**四：自定义注解**

一般多用于某些特定的功能，比较零散的切面，譬如特定的某些方法需要处理，就可以单独在方法上加注解切面。

我们来自定义一个注解：

**[java]** [view plain](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147) [copy](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

[print?](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

1. **package com.example.aop;**
3. **import java.lang.annotation.ElementType;**
4. **import java.lang.annotation.Retention;**
5. **import java.lang.annotation.RetentionPolicy;**
6. **import java.lang.annotation.Target;**
8. /\*\*
9. \* Created by wuwf on 17/4/27.
10. \*/
11. @Target({ElementType.METHOD, ElementType.TYPE})
12. @Retention(RetentionPolicy.RUNTIME)
13. **public @interface UserAccess {**
14. String desc() **default "无信息";**
15. }

注解里提供了一个desc的方法，供被切面的地方传参，如果不需要传参可以不写。

在Controller里加个方法

**[java]** [view plain](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147) [copy](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

[print?](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

1. @RequestMapping("/second")
2. @UserAccess(desc = "second")
3. **public Object second() {**
4. **return "second controller";**
5. }

切面类：

**[java]** [view plain](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147) [copy](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

[print?](http://blog.csdn.net/tianyaleixiaowu/article/details/70853147)

1. **package com.example.aop;**
3. **import org.aspectj.lang.JoinPoint;**
4. **import org.aspectj.lang.ProceedingJoinPoint;**
5. **import org.aspectj.lang.annotation.\*;**
6. **import org.springframework.stereotype.Component;**
8. /\*\*
9. \* Created by wuwf on 17/4/27.
10. \*/
11. @Component
12. @Aspect
13. **public class UserAccessAspect {**
15. @Pointcut(value = "@annotation(com.example.aop.UserAccess)")
16. **public void access() {**
18. }
20. @Before("access()")
21. **public void deBefore(JoinPoint joinPoint) throws Throwable {**
22. System.out.println("second before");
23. }
25. @Around("@annotation(userAccess)")
26. **public Object around(ProceedingJoinPoint pjp, UserAccess userAccess) {**
27. //获取注解里的值
28. System.out.println("second around:" + userAccess.desc());
29. **try {**
30. **return pjp.proceed();**
31. } **catch (Throwable throwable) {**
32. throwable.printStackTrace();
33. **return null;**
34. }
35. }
36. }

主要看一下@Around注解这里，如果需要获取在controller注解中赋给UserAccess的desc里的值，就需要这种写法，这样UserAccess参数就有值了。

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*分割线\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

启动项目，访问http://localhost:8080/second，看控制台：

方法环绕start.....  
URL : http://localhost:8080/second  
HTTP\_METHOD : GET  
IP : 0:0:0:0:0:0:0:1  
CLASS\_METHOD : com.example.controller.FirstController.second  
ARGS : []  
second around:second  
second before  
方法环绕proceed，结果是 :second controller  
方法最后执行.....  
方法的返回值 : second controller

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*分割线\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

通知结果可以看到，两个aop切面类都工作了，顺序呢就是下面的

spring aop就是一个同心圆，要执行的方法为圆心，最外层的order最小。从最外层按照AOP1、AOP2的顺序依次执行doAround方法，doBefore方法。然后执行method方法，最后按照AOP2、AOP1的顺序依次执行doAfter、doAfterReturn方法。也就是说对多个AOP来说，先before的，一定后after。  
对于上面的例子就是，先外层的就是对所有controller的切面，内层就是自定义注解的。  
那不同的切面，顺序怎么决定呢，尤其是同格式的切面处理，譬如两个execution的情况，那spring就是随机决定哪个在外哪个在内了。  
所以大部分情况下，我们需要指定顺序，最简单的方式就是在Aspect切面类上加上@Order(1)注解即可，order越小最先执行，也就是位于最外层。像一些全局处理的就可以把order设小一点，具体到某个细节的就设大一点。