# 4.spring AOP深入

## 4.1.后置通知

目标方法执行之后执行，可以拿到目标方法的返回值。

步骤：配置文件中配置

|  |
| --- |
| <aop:after-returning method="afterReturn" pointcut-ref="pc" returning="name"/> |

在后置通知中获取返回值

|  |
| --- |
| //后置通知  public void afterReturn(String name){  System.out.println("我是一个后置通知");  System.out.println("后置通知拿到的返回值为："+name);  } |

注意：参数名称必须与配置文件中相同。

## 4.2.环绕通知

（当换做环绕通知时，无法拿到返回值。）

执行完环绕通知后，返回值也必须返回。

环绕通知中改造代码

|  |
| --- |
| //环绕通知,要求必须添加ProceedingJoinPoint  public Object around(ProceedingJoinPoint joinPoint) throws Throwable{  System.out.println("环绕通知开始");  //目标方法执行  Object obj = joinPoint.proceed();  System.out.println("环绕通知结束");  return obj;  } |

改造后，即可获取返回值。

## 4.3.AOP的注解形式

3.1.步骤：配置文件

|  |
| --- |
| <!-- 开启aop注解形式 -->  <aop:aspectj-autoproxy/> |

切面中写上注解

|  |
| --- |
| @Component  @Aspect  public class MyAspect {  @Before("execution(\* service..\*.\*(..))")  public void before(){  System.out.println("我是前置通知");  }  } |

最后测试

3.2.接下来，大家用这种方法自己将剩下的4大通知写一下，留一个思考题，在异常通知中，需要获取异常信息和目标方法。

通知的注解类型：

异常通知中获取异常信息和目标方法。

|  |
| --- |
| @AfterThrowing(value="execution(\* service..\*.\*(..))",throwing="throwable")  public void afterThrow(JoinPoint joinPoint,Throwable throwable){  System.out.println("我是异常通知");  System.out.println("异常信息为："+throwable.getMessage());  System.out.println("异常目标方法："+joinPoint.getSignature().getName());  } |

自己在后置通知中获取返回值

|  |
| --- |
| @AfterReturning(value="pointCut()",returning="result")  public void afterReturn(Object result){  System.out.println("我是后置通知");  System.out.println("后置通知中获取返回值:"+result);  } |

3.3.设置通用的切入点

写一个空方法，定义切入点

|  |
| --- |
| @Pointcut("execution(\* service..\*.\*(..))")  public void pointCut(){    } |

其他通知引用切入点，注意引用的时候要加”()”

|  |
| --- |
| @Before("pointCut()")  public void before(){  System.out.println("我是前置通知");  } |

3.4.多个切面的执行过程

再次配置一个切面

|  |
| --- |
| @Component  @Aspect  public class MyAspect1 {  @Around("execution(\* service..\*.\*(..))")  public Object around(ProceedingJoinPoint joinPoint) throws Throwable{  System.out.println("环绕通知测试开始");  Object object = joinPoint.proceed();  System.out.println("环绕通知测试结束");  return object;  }  } |

切入点方法只能在本类中使用

思考：目标方法执行几次？

测试发现，目标方法只执行一次（其实就像给快递包包裹，不管包几层，东西只有一个）

执行过程：当客户端调用目标方法的时候，会执行切面中的通知，当执行到环绕通知中proceed（）时，会再次检测是否还有通知，当检测到还有通知的时候，再次执行通知，以此类推，目标方法只执行一次，当执行以后，便不再执行。

## 4.4.AOP练习：

写出三层结构

4.1.异常:

定义切面类

|  |
| --- |
| @Component  @Aspect  public class MyAspect {  @Pointcut("execution(\* web..\*.\*(..))")  public void pointCut(){    }  @AfterThrowing(value="pointCut()",throwing="throwable")  public void afterThrow(JoinPoint joinPoint, Throwable throwable){  System.out.println("目标对象："+joinPoint.getTarget().getClass());  System.out.println("目标方法："+joinPoint.getSignature().getName());  System.out.println("异常类型："+throwable.getClass());  System.out.println("异常信息："+throwable.getMessage());  }  } |

发现当异常在web层的时候，可以抛出异常，如果把切入点web改为service，则无法抛出异常，但是如果把异常放在service层，依然会抛出异常

因为web层调用service层， Service层将异常抛给了web层，所以依然可以捕获到。

4.2.方法执行的时间

写一个环绕通知

|  |
| --- |
| @Around("pointCut()")  public Object around(ProceedingJoinPoint joinPoint) throws Throwable{  Long beginTime = System.currentTimeMillis();  Object result = joinPoint.proceed();  Long endTime = System.currentTimeMillis();  System.out.println("目标对象："+joinPoint.getTarget().getClass());  System.out.println("目标方法："+joinPoint.getSignature().getName());  System.out.println("目标方法执行时间："+(endTime-beginTime)+"毫秒");  return result;  } |

如果出现时间为0毫秒，说明你的系统非常快，本身方法就非常简单，对程序而言可能运行时间都不到1毫秒。可以人为制造运行时间

在service中加入如下代码

|  |
| --- |
| try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  } |

让程序睡一秒。

4.3.事务

步骤：

创建注解类

|  |
| --- |
| @Target(ElementType.METHOD)  @Retention(RetentionPolicy.RUNTIME)  public @interface Transaction {    } |

在方法上加上注解

|  |
| --- |
| @Override  @Transaction  public void addPerson() { |

在切面中配置环绕通知

|  |
| --- |
| @Around(value="execution(\* service..\*.\*(..))")  public Object around(ProceedingJoinPoint joinPoint) throws Throwable{  //获取方法名称  String methodName = joinPoint.getSignature().getName();  //获取方法,这里getMethod中，参数类型可以不写，这样就会按照方法名来匹配  Method method = joinPoint.getTarget().getClass().getMethod(methodName);  //获取方法上的注解  Transaction transaction = method.getAnnotation(Transaction.class);  //判断注解是否存在  if(transaction!=null){  System.out.println("事务开始");  Object result = joinPoint.proceed();  System.out.println("事务结束");  return result;  }else{  Object result = joinPoint.proceed();  return result;  }  } |

切面中快速获取注解：

|  |
| --- |
| /\*  \* 在切入点表达式后加上&& @annotation(transaction)，表示切入点还要满足带有注解  \*/  @Around("execution(\* service..\*.\*(..)) && @annotation(tx)")  public Object around(ProceedingJoinPoint joinPoint,Transaction tx) throws Throwable{  System.out.println("事务开始");  Object result = joinPoint.proceed();  System.out.println("事务结束");  return result;  } |

说明：@annotation(transaction)可以匹配到方法上的所有注解，@annotation(tx)中tx只是形参，代表的是任意的注解，在参数中会指定注解的类型，如果方法上的注解中有次类型的注解，则执行通知

4.4.权限控制

首先，创建权限注解类

|  |
| --- |
| @Target(ElementType.METHOD)  @Retention(RetentionPolicy.RUNTIME)  public @interface privilegeInfo {  public String name();  } |

在ServiceImpl的方法上加上注解

|  |
| --- |
| @Override  //@Transaction  //@Hello  @privilegeInfo(name="add")  public void addPerson() { |

准备用户权限，为了方便测试，将用户权限列表放在测试中，本来应该在数据库中获取

|  |
| --- |
| @Test  public void Test01(){  ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");  PersonServlet personServlet = (PersonServlet) context.getBean("personServlet");  List<String> list = new ArrayList<String>();  list.add("add");  list.add("update");  list.add("delete");  PriList.setPriList(list);  personServlet.addPerson();  } |

但是如何在切面中获取用户权限呢？创建一个工具类，将属性与方法设置为静态的。

|  |
| --- |
| public class PriList {  private static List<String> priList;  public static List<String> getPriList() {  return priList;  }  public static void setPriList(List<String> priList) {  PriList.priList = priList;  }  } |

在切面中配置通知

|  |
| --- |
| /\*  \* 只有有权限注解的类才会被拦截，没有权限注解的类说明不需要权限，直接放行  \*/  @Around("execution(\* service..\*.\*(..)) && @annotation(pri)")  public Object around1(ProceedingJoinPoint joinPoint,privilegeInfo pri) throws Throwable{  //获取注解中的name属性值  String pname = pri.name();  //判断用户列表中是否含有该权限值  if(PriList.getPriList().contains(pname)){//有权限，则进该方法  Object result = joinPoint.proceed();  return result;  }else{//无权限，则不能执行目标方法  System.out.println("对不起，权限不足");  return null;  }  } |

最后测试即可。