## 1.什么是SpringAOP

什么是aop:Aspect Oriented Programming的缩写，面向切面编程，通过预编译和动态代理实现程序功能的  
统一维护的一种技术  
主要功能：日志记录，性能统计，安全控制，事务处理，异常处理等

## 2.SpringAOP框架的用途

提供了声明的企业服务，特别是EJB的替代服务的声明  
允许用户控制自己的方面，以完成OOP和AOP的互补使用  
  
OOP：模拟真实的世界，一切皆是对象

## 3.AOP的实现方式

下边这两种Spring都是支持的

3.1预编译  
-AspectJ  完整的面向切面编程解决方案--》spring不是完整的解决方案，不过spring提供比较好的实现方式,当然spring是同时也是支持这种方式的，这也是一种常用的方式  
  
3.2运行期间动态代理（JDK动态代理，CGLib动态代理）  
-SpringAop,JbossAop

Spring的AOP使用纯java实现，无需特殊的编译过程，不需要控制类的加载器层次，目前只支持方法的执行的连接点（通知Spring Bean某个方法执行）  
不是为了提供完整AOP实现；而是侧重于一种AOP于IOC容器之间的整合，SpringAOP不会AspectJ（完整的AOP解决方案）竞争

**Spring没有使用AspectJ的时候，也可以通过如下方式实现AOP**

Spring AOP 默认使用标准的JavaSE动态代理作为AOP代理，这使得任何接口（或者集合）都可以被代理  
Spring AOP 中也可以使用CGLIB代理（如果一个业务对象没有实现一个接口）  
有接口的：使用JDK的动态里  
无接口的：使用CGLIB代理

## 4.SpringAOP中的一些概念

在实际使用SpringAOP之前，了解他的概念是必不可少的一个过程,

#### SpringAOP主要围绕以下概念展开：

切面（Aspect）一个关注点的模块化，这个关注点可能会横切多个对象  
连接点（Joinpoint）程序执行过程中某个特定的连接点  
通知（Advice） 在切面的某个特的连接点上执行的动作  
切入点（Pointcut）匹配连接的断言，在Aop中通知和一个切入点表达式关联  
引入（Intruduction） 在不修改类代码的前提下，为类添加新的方法和属性  
目标对象（Target Object） 被一个或者多个切面所通知的对象  
Aop代理（AOP Proxy） AOP框架创建的对象，用来实现切面契约（aspect contract）（包括方法执行等）  
织入（Weaving）把切面连接到其他的应用程序类型或者对象上，并创建一个被通知的对象，氛围：编译时织入，类加载时织入，执行时织入

#### 通知类型Advice:

前置通知（before advice） 在某个连接点（jion point）之前执行的通知，但不能阻止连接点前的执行（除非抛出一个异常）  
返回后通知（after returning advice）在某个连接点（jion point）正常执行完后执行通知  
抛出异常通知（after throwing advice） 在方法异常退出时执行的通知  
后通知（after(finally) advice）在方法抛出异常退出时候的执行通知（不管正常返回还是异常退出）  
环绕通知（around advice) 包围一个连接点（jion point)的通知

#### 切入点Pointcut:SpringAOP占时仅仅支持方法的连接点

例如定义切入点表达式 execution(\* com.sample.service.impl..\*.\*(..))

execution()是最常用的切点函数，其语法如下所示：

 整个表达式可以分为五个部分：

 1、execution(): 表达式主体。

 2、第一个\*号：表示返回类型，\*号表示所有的类型。

 3、包名：表示需要拦截的包名，后面的两个句点表示当前包和当前包的所有子包，com.sample.service.impl包、子孙包下所有类的方法。

 4、第二个\*号：表示类名，\*号表示所有的类。

 5、\*(..):最后这个星号表示方法名，\*号表示所有的方法，后面括弧里面表示方法的参数，两个句点表示任何参数。

execution(public \* \*(..)) 切入点为执行所有的public方式时  
execution(\* set\*(..)) 切入点执行所有的set开始的方法时  
execution(\* com.xyz.service.Account.\*(..)) 切入点执行Account类的所有方法时  
execution(\* com.xyz.service.\*.\*(..))切入点执行com.xyz.service包下的所有方法时  
execution(\* com.xyz.service..\*.\*(..)) 切入点执行com.xyz.service包以及其子包的所有的方法时

上边这种方式aspectj和springaop通用的，其他方式可以自己查找资料

推荐：

<http://blog.csdn.net/abcd898989/article/details/50809321>

http://blog.csdn.net/peng658890/article/details/7223046

## 5.SpringAOP实现

上边的一些概念，看过后可能还是不懂，可以自行查阅资料，下边我也会说明

我们在Java项目开发中，AOP常见的配置有如下3种：

a.基于SpringAOP容器的实现，使用AspectJ(Spring封装了AspectJ的使用)，这是一种比较常用的方式，可以很容易看清代码结构

b.运行期间的动态代理实现，这是一种比较老的实现方式，比较繁琐

c.注解实现，这种方式开发起来简单，但是不利于后期维护，比如说很难找出你所有使用了SpringAOP注解的地方

这里我主要介绍第一种，和第二种的使用

### 5.1SpringAOP中的容器实现

### 5.1.1 前置通知 before

某个需要切入的方法之前执行切面中的方法

Spring所有的切面通知都必须放在一个<aop:config>内（可以配置多个<aop:config>元素），每一个<aop:config>可以包含pointcut,adviso  
和aspect元素（注意这些元素的出现是由顺序的）

配置文件：spring-aop-schema-advice.xml

声明了切面类，当切入点中匹配到了类名包含BIZ字符串的类时，选取面类中的一个方法，在选取某种Adivice通知，切入该方法的执行过程

其中：aop:aspect 中id的值任意(表意性强)  ref的值为切面类的id值

    aop:pointcut 中expression写切入点表达式，说明那个方法可能需要做切入点

    aop:before 中method的值为切面中的方法名说明切入那个方法，pointcut-ref的值为切入点的id值

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
6. http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.2.xsd"**>**
8. <!-- 切面类 -->
9. **<bean** id="myAspect" class="com.briup.spring.aop.bean.annotation.aop.MyAspect"**></bean>**
11. <!-- 业务类 -->
12. **<bean** id="aspectBiz" class="com.briup.spring.aop.bean.annotation.aop.AspectBiz"**></bean>**
13. <!-- aop配置 可以配置多个-->
14. **<aop:config>**
15. <!-- 切面类 -->
16. **<aop:aspect** id="aspectTest" ref="myAspect"**>**
17. <!-- 切入点  标识切入点 aop包下类名包含Biz的类的所有方法-->
18. **<aop:pointcut** expression="execution(\* com.briup.spring.aop.bean.annotation.aop.\*Biz.\*(..))" id="myPointcut"**/>**
19. <!-- 通知 ,通过切入点切入切入切面类中的before方法-->
20. **<aop:before** method="before" pointcut-ref="myPointcut"**/>**
21. **</aop:aspect>**
22. **</aop:config>**
23. **</beans>**

切面类：

切面类中的某个方法，一般是用于切入业务类中的某个方法在某种状态时切入

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. /\*
2. \* 声明一个切面类
3. \* \*/
4. **public** **class** MyAspect {
6. **public** **void** before(){
7. System.out.println("aspect before");
8. }
10. }

业务类：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. //业务类
2. **public** **class** AspectBiz {
4. **public** **void** biz(){
5. System.out.println("Aspect biz");
6. }
8. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. @Test
2. **public** **void** aspectBefore(){
3. ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext("com/briup/spring/chap4/spring-aop-schema-advice.xml");
4. AspectBiz aspectBiz = (AspectBiz) ac.getBean("aspectBiz");
5. aspectBiz.biz();
6. }

结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. aspect before
2. Aspect biz

通过结果可以看出，前置通知在业务类AspectBiz的方法biz执行之前执行了before方法

#### 5.1.2后置通知after-returning

某个需要切入的方法执行完成之后执行切面中指定的方法

配置文件：spring-aop-schema-advice.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<aop:config>**
2. <!-- 切面类 -->
3. **<aop:aspect** id="aspectTest" ref="myAspect"**>**
4. <!-- 切入点  标识切入点 aop包下类名以Biz结尾的类的所有方法-->
5. **<aop:pointcut** expression="execution(\* com.briup.spring.aop.bean.annotation.aop.\*Biz.\*(..))" id="myPointcut"**/>**
6. <!-- 通知 ,通过切入点切入切入切面类中的before方法 在执行切入点指定的方法之前执行 -->
7. **<aop:before** method="before" pointcut-ref="myPointcut"**/>**
8. <!-- 返回之后的通知 -->
9. **<aop:after-returning** method="afterReturning" pointcut-ref="myPointcut"**/>**
10. **</aop:aspect>**
11. **</aop:config>**

切面类：

添加afterReturning方法

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. /\*
2. \* 声明一个切面类
3. \* \*/
4. **public** **class** MyAspect {
6. **public** **void** before(){
7. System.out.println("aspect before");
8. }
10. **public** **void** afterReturning(){
11. System.out.println("aspect afterReturning");
12. }
14. }

结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. aspect before
2. Aspect biz
3. aspect afterReturning

结果可以看出，后置通知会在业务方法的执行之后

#### 5.1.3异常通知after-throwing

在方法抛出异常后的通知

配置文件：spring-aop-schema-advice.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<aop:config>**
2. <!-- 切面类 -->
3. **<aop:aspect** id="aspectTest" ref="myAspect"**>**
4. <!-- 切入点  标识切入点 aop包下类名以Biz结尾的类的所有方法-->
5. **<aop:pointcut** expression="execution(\* com.briup.spring.aop.bean.annotation.aop.\*Biz.\*(..))" id="myPointcut"**/>**
6. <!-- 通知 ,通过切入点切入切入切面类中的before方法 在执行切入点指定的方法之前执行 -->
7. **<aop:before** method="before" pointcut-ref="myPointcut"**/>**
8. <!-- 返回之后的通知 -->
9. **<aop:after-returning** method="afterReturning" pointcut-ref="myPointcut"**/>**
10. <!--  -->
11. **<aop:after-throwing** method="afteThrowing" pointcut-ref="myPointcut"**/>**
12. **</aop:aspect>**
13. **</aop:config>**

切面类：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. /\*
2. \* 声明一个切面类
3. \* \*/
4. **public** **class** MyAspect {
6. **public** **void** before(){
7. System.out.println("aspect before");
8. }
10. **public** **void** afterReturning(){
11. System.out.println("aspect afterReturning");
12. }
14. **public** **void** afteThrowing(){
15. System.out.println("aspect afteThrowing");
16. }
17. }

业务类：

修改为如下：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **public** **void** biz(){
2. System.out.println("Aspect biz");
3. **throw** **new** RuntimeException(); //出现异常的时候afteThrowing才会执行
4. }

测试和结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. aspect before
2. Aspect biz
3. aspect afteThrowing
4. 上边的结果由于抛出异常后，不会正常的返回所有没有aspect afterReturning输出

#### 5.1.4最终通知after(finally) advice

不管方法是否会抛出异常都会通知，就像try...catch..finallly

配置文件:spring-aop-schema-advice.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<aop:config>**
2. <!-- 切面类 -->
3. **<aop:aspect** id="aspectTest" ref="myAspect"**>**
4. <!-- 切入点  标识切入点 aop包下类名以Biz结尾的类的所有方法-->
5. **<aop:pointcut** expression="execution(\* com.briup.spring.aop.bean.annotation.aop.\*Biz.\*(..))" id="myPointcut"**/>**
6. <!-- 通知 ,通过切入点切入切入切面类中的before方法 在执行切入点指定的方法之前执行 -->
7. **<aop:before** method="before" pointcut-ref="myPointcut"**/>**
8. <!-- 返回之后的通知 -->
9. **<aop:after-returning** method="afterReturning" pointcut-ref="myPointcut"**/>**
10. <!--  -->
11. **<aop:after-throwing** method="afteThrowing" pointcut-ref="myPointcut"**/>**
12. <!-- after(finally) advice 不管是否抛出异常，最后都会执行的方法 -->
13. **<aop:after** method="after" pointcut-ref="myPointcut"**/>**
14. **</aop:aspect>**
15. **</aop:config>**

切面类：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. /\*
2. \* 声明一个切面类
3. \* \*/
4. **public** **class** MyAspect {
6. **public** **void** before(){
7. System.out.println("aspect before");
8. }
10. **public** **void** afterReturning(){
11. System.out.println("aspect afterReturning");
12. }
14. **public** **void** afteThrowing(){
15. System.out.println("aspect afteThrowing");
16. }
18. **public** **void** after(){
19. System.out.println("aspect after(finally)");
20. }
21. }

业务类：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. //业务类
2. public class AspectBiz {
4. public void biz(){
5. System.out.println("Aspect biz");
6. throw new RuntimeException();
7. }
8. }

测试结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. aspect before
2. Aspect biz
3. aspect afteThrowing
4. aspect after(finally)

从结果中可以看出，抛出异常后，切面类中的after方法还是可以执行

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. 如果业务类如下：没有异常，
2. **public** **class** AspectBiz {
3. **public** **void** biz(){
4. System.out.println("Aspect biz");
6. }
8. }

结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. aspect before
2. Aspect biz
3. aspect after(finally)

由以上结果可以看到，不管怎么样after方法还是会执行，很像try...catch..finally

### 5.1.5环绕通知around

在方法的执行前后执行通知

配置文件:spring-aop-schema-advice.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<aop:config>**
2. <!-- 切面类 -->
3. **<aop:aspect** id="aspectTest" ref="myAspect"**>**
4. <!-- 切入点  标识切入点 aop包下类名以Biz结尾的类的所有方法-->
5. **<aop:pointcut** expression="execution(\* com.briup.spring.aop.bean.annotation.aop.\*Biz.\*(..))" id="myPointcut"**/>**
6. <!-- 通知 ,通过切入点切入切入切面类中的before方法 在执行切入点指定的方法之前执行 -->
7. **<aop:before** method="before" pointcut-ref="myPointcut"**/>**
8. <!-- 返回之后的通知 -->
9. **<aop:after-returning** method="afterReturning" pointcut-ref="myPointcut"**/>**
10. <!--  -->
11. **<aop:after-throwing** method="afteThrowing" pointcut-ref="myPointcut"**/>**
12. <!-- after(finally) advice 不管是否抛出异常，最后都会执行的方法 -->
13. **<aop:after** method="after" pointcut-ref="myPointcut"**/>**
14. <!-- adroun advice 环绕通知，方法的第一参数必须是ProceedingJoinPoint类型 -->
15. **<aop:around** method="around" pointcut-ref="myPointcut"**/>**
16. **</aop:aspect>**
17. **</aop:config>**

切面类：

可以看出，环绕方法的第一参数必须四ProceedingJoinPoint类型

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. /\*
2. \* 声明一个切面类
3. \* \*/
4. **public** **class** MyAspect {
6. **public** **void** before(){
7. System.out.println("aspect before");
8. }
10. **public** **void** afterReturning(){
11. System.out.println("aspect afterReturning");
12. }
14. **public** **void** afteThrowing(){
15. System.out.println("aspect afteThrowing");
16. }
18. **public** **void** after(){
19. System.out.println("aspect after(finally)");
20. }
22. **public** **void** around(ProceedingJoinPoint joinPoint){
23. Object object = **null**;
24. **try**{
25. System.out.println("aspect around 1"); //方法执行前
26. object = joinPoint.proceed();  //代表业务方法的执行
27. System.out.println("aspect around 1"); //方法执行后
28. }**catch**(Throwable e){
29. e.printStackTrace();
30. }
31. }
32. }

测试结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. aspect around 1
2. Aspect biz
3. aspect around 1

从结果可以看出，joinPoint.proceed其实就是执行业务方法，我们可以在其之前做和之后做一些操作

5.1.6通知中的参数传递

这里我们列举环绕通知中的参数参数传递，其他通知也是同样的方式。

配置文件：spring-aop-schema-advice.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:aop="http://www.springframework.org/schema/aop"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
6. http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.2.xsd"**>**
8. <!-- 切面类 -->
9. **<bean** id="myAspect" class="com.briup.spring.aop.bean.annotation.aop.MyAspect"**></bean>**
11. <!-- 业务类 -->
12. **<bean** id="aspectBiz" class="com.briup.spring.aop.bean.annotation.aop.AspectBiz"**></bean>**
13. <!-- aop配置 可以配置多个-->
14. **<aop:config>**
15. <!-- 切面类 -->
16. **<aop:aspect** id="aspectTest" ref="myAspect"**>**
17. <!-- 切入点  标识切入点 aop包下类名以Biz结尾的类的所有方法-->
18. **<aop:pointcut** expression="execution(\* com.briup.spring.aop.bean.annotation.aop.\*Biz.\*(..))" id="myPointcut"**/>**
19. <!-- 通知 ,通过切入点切入切入切面类中的before方法 在执行切入点指定的方法之前执行 -->
20. **<aop:before** method="before" pointcut-ref="myPointcut"**/>**
21. <!-- 返回之后的通知 -->
22. **<aop:after-returning** method="afterReturning" pointcut-ref="myPointcut"**/>**
23. <!--  -->
24. **<aop:after-throwing** method="afteThrowing" pointcut-ref="myPointcut"**/>**
25. <!-- after(finally) advice 不管是否抛出异常，最后都会执行的方法 -->
26. **<aop:after** method="after" pointcut-ref="myPointcut"**/>**
27. <!-- adroun advice 环绕通知，方法的第一参数必须是ProceedingJjoinPoint类型 -->
28. <!-- <aop:around method="around" pointcut-ref="myPointcut"/> -->
30. <!-- 参数传递  -->
31. **<aop:around** method="aroundInit" pointcut="execution(\* com.briup.spring.aop.bean.annotation.aop.AspectBiz.init(String,int))
32. and args(bizName,times)"**/>**
33. **</aop:aspect>**
34. **</aop:config>**
35. **</beans>**

其中，通知中也可以单独指定的pointcut，切入点中，我们单独指定了init方法，且指定了方法的参数类型，和方法参数的名字，其中方法参数名字，可以更实际的方法参数名字不同，比如说init(String biz,int t) 都是可以匹配到的，不过为了维护方便，最好还是都一样。

切面类：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. /\*
2. \* 声明一个切面类
3. \* \*/
4. **public** **class** MyAspect {
6. **public** **void** before(){
7. System.out.println("aspect before");
8. }
10. **public** **void** afterReturning(){
11. System.out.println("aspect afterReturning");
12. }
14. **public** **void** afteThrowing(){
15. System.out.println("aspect afteThrowing");
16. }
18. **public** **void** after(){
19. System.out.println("aspect after(finally)");
20. }
22. **public** **void** around(ProceedingJoinPoint joinPoint){
23. Object object = **null**;
24. **try**{
25. System.out.println("aspect around 1"); //方法执行前
26. object = joinPoint.proceed();  //代表业务方法的执行
27. System.out.println("aspect around 2"); //方法执行后
28. }**catch**(Throwable e){
29. e.printStackTrace();
30. }
31. }
33. //AOP中参数的传递
34. **public** **void** aroundInit(ProceedingJoinPoint joinPoint,String bizName,**int** times){
35. System.out.println(bizName+"--"+times);
36. Object object = **null**;
37. **try**{
38. System.out.println("aspect around 1"); //方法执行前
39. object = joinPoint.proceed();  //代表业务方法的执行
40. System.out.println("aspect around 1"); //方法执行后
41. }**catch**(Throwable e){
42. e.printStackTrace();
43. }
44. }
45. }

业务类：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **public** **class** AspectBiz {
3. **public** **void** biz(){
4. System.out.println("Aspect biz");
5. }
7. **public** **void** init(String bizName,**int** times){
8. System.out.println("aspectBiz init:"+bizName+"  "+times);
9. }
11. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. @Test
2. **public** **void** aspectAround(){
3. ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext("com/briup/spring/chap4/spring-aop-schema-advice.xml");
4. AspectBiz aspectBiz = (AspectBiz) ac.getBean("aspectBiz");
5. aspectBiz.init("init", 3);
6. }

测试结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. aspect before
2. init--3
3. aspect around 1
4. aspectBiz init:init  3
5. aspect around 1
6. aspect after(finally)
7. aspect afterReturning
8. AfterClass 标注的方法 会最后执行

可以看到，参数比顺利的传送过去

### 6.Advisor

Advisor就像一个小的自包含，只有一个advice切面通过一个bean标识，并且必须实现一个advice接口，同时advisor也可以很好的利用aspectJ的切入点表达式Spring通过配置文件中 <aop:advisor>元素支持advisor实际使用中，大多数情况下它会和transactional advice配合使用，用于事务控制

配置文件使用案例：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:context="http://www.springframework.org/schema/context"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:tx="http://www.springframework.org/schema/tx"
5. xmlns:aop="http://www.springframework.org/schema/aop"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
8. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
9. http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.2.xsd"**>**
11. **<bean** name="sessionFactory"
12. class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"**>**
13. **<property** name="configLocation"**>**
14. <!-- 路径位于src下 -->
15. **<value>**classpath:hibernate.cfg.xml**</value>**
16. **</property>**
17. **</bean>**
19. <!-- 事物配置 -->
20. **<bean** id="txManager"
21. class="org.springframework.orm.hibernate3.HibernateTransactionManager"**>**
22. **<property** name="sessionFactory"**>**
23. **<ref** local="sessionFactory" **/>**
24. **</property>**
25. **</bean>**
27. <!-- advisor配置 配置事务处理的Bean,定义切面（advice）
28. 由于getxxx,queryxxx,findxxx不涉及事务，可以设置read-only为true
29. --**>**
30. **<tx:advice** id="txAdvice" transaction-manager="txManager"**>**
31. **<tx:attributes>**
32. **<tx:method** name="get\*" read-only="true" **/>**
33. **<tx:method** name="query\*" read-only="true" **/>**
34. **<tx:method** name="find\*" read-only="true" **/>**
35. **<tx:method** name="save\*" propagation="REQUIRED" rollback-for="java.lang.Exception" **/>**
36. **<tx:method** name="add\*" propagation="REQUIRED" rollback-for="java.lang.Exception" **/>**
37. **<tx:method** name="del\*" propagation="REQUIRED" rollback-for="java.lang.Exception" **/>**
38. **<tx:method** name="update\*" propagation="REQUIRED" rollback-for="java.lang.Exception" **/>**
39. **<tx:method** name="\*" propagation="REQUIRED" rollback-for="java.lang.Exception" **/>**
40. **</tx:attributes>**
41. **</tx:advice>**
43. <!-- aop配置 -->
44. **<aop:config>**
45. <!-- 给impl包下的所有的方法配置adivce -->
46. **<aop:pointcut** expression="excution(\* com.briup.spring.aop.bean.annotation.service.impl..(..))" id="serviceMethod" **/>**
47. **<aop:advisor** advice-ref="txAdvice" pointcut-ref="serviceMethod"  **/>**
48. **</aop:config>**
50. **</beans>**

关于spring的事务控制将会在另外一篇文章中介绍。

### 7.SpringAOP代理实现

这种实现方式，主要使用动态代理技术。 有接口的：使用JDK的动态里，无接口的：使用CGLIB代理

这种方式不是很常用

#### 7.1.1ProxyFactoryBean

1.使用SpringAOP代理最关键的是使用org.springframework.aop.framework.ProxyFactoryBean,可以完全控制切入点和通知advice以及他们的顺序

2.使用ProxyFacotryBean或者其他的IOC相关类来创建AOP代理的最重要的好处就是通知切入点可以由IOC来管理

3.被代理的类没有实现任何接口，使用CGLIB代理，否者使用JDK代理

4.通过设置ProxyTargetClass为true可以强制使用CGLIB，（无论是否实现接口）

5.如果目标类实现了一个或者多个接口，那么创建代理的类型将依赖于ProxyFactoryBean的配置

6.如果ProxyFactoryBean的proxyInterfaces属性被设置为一个或者多个全限定接口名，基于JDK的代理被创建

7.如果ProxyFactoryBean的proxyInterfaces属性没有被设置，但是目标类实现类一个或多个接口，那么ProxyFactoryBean将自动检测到这个目标类已经实现了至少一个接口，创建一个基于JDK的代理

#### 7.1.2使用代理实现环绕通知around

配置文件：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. <!-- 配置委托类 -->
7. **<bean** name="target" class="com.xxx.spring.aop.bean.UserDaoImpl"**></bean>**
8. <!-- 配置代理对象将来执行的时候，所要执行的处理程序 -->
9. **<bean** name="advice" class="com.xx.spring.aop.bean.AdviceTest"**></bean>**<!-- advice在方法执行前后添加我们的操作 -->
10. <!-- 配置代理对象 -->
11. **<bean** name="proxy" class="org.springframework.aop.framework.ProxyFactoryBean"**>**
12. <!--proxyInterfaces 如果ProxyFactoryBean的proxyInterfaces属性没有被设置，但是目标类实现类一个或多个接口，
13. 那么ProxyFactoryBean将自动检测到这个目标类已经实现了至少一个接口，创建一个基于JDK的代理 --**>**
14. **<property** name="proxyInterfaces" value="com.xxx.spring.aop.bean.UserDao"**></property>**
15. <!-- 注入目标对象(注入被代理的对象) -->
16. **<property** name="target" ref="target"**></property>**
17. <!-- 注入代理对象所要执行的处理程序,通知配置 -->
18. **<property** name="interceptorNames"**>**
19. **<list>**
20. **<value>**advice**</value>**
21. **</list>**
22. **</property>**
23. **</bean>**
24. **</beans>**

注意：

上边的<property name="proxyInterfaces" value="com.xxx.spring.aop.bean.UserDao">可以不用写，Spring的ProxyFactoryBean将会自动检测到

如果被代理的对象target实现了多个接口可以按照如下配置：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<property** name="interfaces"**>**
2. **<array>**
3. **<value>**com.xxx.spring.aop.bean.UserDao**</value>**
4. **</array>**
5. **</property>**

value中可以写多个接口

同样：通知interceptorNames如果只用一个通知可以写成<property name="interceptorNames" name="advice">  
    如果有多个通知可以写成：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<property** name="interceptorNames"**>**
2. **<list>**
3. **<value>**advice**</value>**
4. **</list>**
5. **</property>**

但是 通知是不可以省略的，同时推荐结合的写法

上边的委托类即使普通的实现类：即ProxyFactoryBean被代理的对象

<bean name="target" class="com.xxx.spring.aop.bean.UserDaoImpl"></bean>

上边的通知配置

<bean name="advice" class="com.briup.spring.aop.bean.AdviceTest"></bean>

ProxyFactoryBean的配置

需要配置被代理的目标对象，通知，目标类实现的接口（可以省略，被ProxyFactoryBean自动识别）

切面通知实现类

实现MethodInterceptor就可以实现环绕通知，我们可以在方法的目标对象类的方法执行前后加入处理逻辑

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **import** java.util.Date;
2. **import** org.aopalliance.intercept.MethodInterceptor;
3. **import** org.aopalliance.intercept.MethodInvocation;
5. **public** **class** AdviceTest **implements** MethodInterceptor{
7. @Override
8. **public** Object invoke(MethodInvocation method) **throws** Throwable {
9. System.out.println("方法开始执行"+**new** Date());
10. method.proceed();
11. System.out.println("方法执行完毕"+**new** Date());
12. **return** **null**;
13. }
14. }

目标类：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. //委托类
2. **public** **class** UserDaoImpl **implements** UserDao {
4. @Override
5. **public** **void** saveUser() {
6. System.out.println("保存用户");
7. }
9. @Override
10. **public** **void** deleteUser() {
11. System.out.println("删除用户");
12. }
14. @Override
15. **public** **void** updateUser() {
16. System.out.println("更新用户");
17. }
19. }

接口：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **public** **interface** UserDao {
2. **public** **abstract** **void** saveUser();
3. **public** **abstract** **void** deleteUser();
4. **public** **abstract** **void** updateUser();
5. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. @Test
2. **public** **void** advice(){
3. BeanFactory factory = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap2/advice.xml");
4. UserDao userDao = factory.getBean("proxy", UserDao.**class**);
5. userDao.saveUser();
6. userDao.deleteUser();
7. userDao.updateUser();
8. }

测试结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. 方法开始执行Sun Sep 11 21:02:12 CST 2016
2. 保存用户
3. 方法执行完毕Sun Sep 11 21:02:12 CST 2016
4. 方法开始执行Sun Sep 11 21:02:12 CST 2016
5. 删除用户
6. 方法执行完毕Sun Sep 11 21:02:12 CST 2016
7. 方法开始执行Sun Sep 11 21:02:12 CST 2016
8. 更新用户
9. 方法执行完毕Sun Sep 11 21:02:12 CST 2016

#### 7.1.3使用动态代理实现前置通知before

配置文件：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. <!-- 配置委托类 -->
7. **<bean** name="target" class="com.xxx.spring.aop.bean.UserDaoImpl"**></bean>**
8. <!-- 配置代理对象将来执行的时候，所要执行的处理程序 -->
9. **<bean** name="before" class="com.xxx.spring.aop.bean.BeforeTest"**></bean>**
10. <!-- 配置代理对象 -->
11. **<bean** name="proxy" class="org.springframework.aop.framework.ProxyFactoryBean"**>**
12. <!-- 注入目标对象(注入被代理的对象) -->
13. **<property** name="target" ref="target"**></property>**
14. <!-- 注入代理对象需要实现的所有接口 -->
15. **<property** name="interfaces"**>**
16. **<array>**
17. **<value>**com.xxx.spring.aop.bean.UserDao**</value>**
18. **</array>**
19. **</property>**
20. <!-- 注入代理对象所要执行的处理程序 -->
21. **<property** name="interceptorNames"**>**
22. **<array>**
23. **<value>**before**</value>**
24. **</array>**
25. **</property>**
26. **</bean>**
27. **</beans>**

切面前置通知实现类：

实现MethodBeforeAdvice

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **import** java.lang.reflect.Method;
2. **import** org.springframework.aop.MethodBeforeAdvice;
3. **public** **class** BeforeTest **implements** MethodBeforeAdvice{
5. @Override
6. **public** **void** before(Method method, Object[] obj, Object object)
7. **throws** Throwable {
8. System.out.println("version 1.0 author tom "+method.getName()+" is execute");
9. }
11. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. @Test
2. **public** **void** before(){
3. BeanFactory factory = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap2/before.xml");
4. UserDao userDao = factory.getBean("proxy", UserDao.**class**);
5. userDao.saveUser();
6. userDao.deleteUser();
7. userDao.updateUser();
8. }

结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. version 1.0 author tom saveUser is execute
2. 保存用户
3. version 1.0 author tom deleteUser is execute
4. 删除用户
5. version 1.0 author tom updateUser is execute
6. 更新用户

#### 7.1.4后置通知afterReturning

配置文件：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. <!-- 配置委托类 -->
7. **<bean** name="target" class="com.xxx.spring.aop.bean.UserDaoImpl"**></bean>**
8. <!-- 配置代理对象将来执行的时候，所要执行的处理程序 -->
9. **<bean** name="after" class="com.xxx.spring.aop.bean.AfterTest"**></bean>**
10. <!-- 配置代理对象 -->
11. **<bean** name="proxy" class="org.springframework.aop.framework.ProxyFactoryBean"**>**
12. <!-- 注入目标对象(注入被代理的对象) -->
13. **<property** name="target" ref="target"**></property>**
14. <!-- 注入代理对象需要实现的所有接口 -->
15. **<property** name="interfaces"**>**
16. **<array>**
17. **<value>**com.xxx.spring.aop.bean.UserDao**</value>**
18. **</array>**
19. **</property>**
20. <!-- 注入代理对象所要执行的处理程序 -->
21. **<property** name="interceptorNames"**>**
22. **<array>**
23. **<value>**after**</value>**
24. **</array>**
25. **</property>**
26. **</bean>**
27. **</beans>**

切面后置通知：

实现 AfterReturningAdivce接口

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **import** java.lang.reflect.Method;
2. **import** org.springframework.aop.AfterReturningAdvice;
3. **public** **class** AfterTest  **implements** AfterReturningAdvice {
5. @Override
6. **public** **void** afterReturning(Object arg0, Method arg1, Object[] arg2,
7. Object arg3) **throws** Throwable {
8. System.out.println(arg1.getName()+" is over!");
9. }
10. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. @Test
2. **public** **void** after(){
3. BeanFactory factory = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap2/after.xml");
4. UserDao userDao = factory.getBean("proxy", UserDao.**class**);
5. userDao.saveUser();
6. userDao.deleteUser();
7. userDao.updateUser();
8. }

测试结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. 保存用户
2. saveUser is over!
3. 删除用户
4. deleteUser is over!
5. 更新用户
6. updateUser is over!

#### 7.1.5异常通知throw

配置文件：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. <!-- 配置委托类 -->
7. **<bean** name="target" class="com.briup.spring.aop.bean.UserDaoImpl"**></bean>**
8. <!-- 配置代理对象将来执行的时候，所要执行的处理程序 -->
9. **<bean** name="throws" class="com.briup.spring.aop.bean.ThrowsAdiviceTest"**></bean>**
10. <!-- 配置代理对象 -->
11. **<bean** name="proxy" class="org.springframework.aop.framework.ProxyFactoryBean"**>**
12. <!-- 注入目标对象(注入被代理的对象) -->
13. **<property** name="target" ref="target"**></property>**
14. <!-- 注入代理对象需要实现的所有接口 -->
15. **<property** name="interfaces"**>**
16. **<array>**
17. **<value>**com.briup.spring.aop.bean.UserDao**</value>**
18. **</array>**
19. **</property>**
20. <!-- 注入代理对象所要执行的处理程序 -->
21. **<property** name="interceptorNames"**>**
22. **<list>**
23. **<value>**throws**</value>**
24. **</list>**
25. **</property>**
26. **</bean>**
27. **</beans>**

异常通知切面：

参数中必须有Throwable的子类，前边的参数afterThrowing([Method, args, target], subclassOfThrowable)[]号中的参数可选

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **import** java.lang.reflect.Method;
3. **import** org.springframework.aop.ThrowsAdvice;
5. **public** **class** ThrowsAdiviceTest **implements** ThrowsAdvice{
7. **public** **void** afterThrowing(Method method,Object[] args,Object target,Exception ex)**throws** Throwable{//Throwable subclass
8. System.out.println("afterThrowing 2 ...."+method.getName()+"   "+ target.getClass().getName());
9. }
11. }

实现类：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. @Override
2. **public** **void** saveUser() **throws** RuntimeException{
3. System.out.println("保存用户");
4. **throw** **new** RuntimeException();
5. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. @Test
2. **public** **void** throwTest(){
3. BeanFactory factory = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap2/throw.xml");
4. UserDao userDao = factory.getBean("proxy", UserDao.**class**);
5. userDao.saveUser();
6. userDao.deleteUser();
7. userDao.updateUser();
8. }

结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. 保存用户
2. afterThrowing 2 ....saveUser   com.xxx.spring.aop.bean.UserDaoImpl

#### 7.1.6切入点配置pointcut

使用代理方式也可以配置切入点

NameMatchMethodPointcut，根据方法的名字进行匹配

mappedNames匹配的方法名集合

配置文件：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. <!-- 配置委托类 -->
7. **<bean** name="target" class="com.briup.spring.aop.bean.UserDaoImpl"**></bean>**
8. <!-- 配置代理对象将来执行的时候，所要执行的处理程序 -->
9. **<bean** name="advice" class="com.briup.spring.aop.bean.AdviceTest"**></bean>**
10. **<bean** name="throws" class="com.briup.spring.aop.bean.ThrowsAdiviceTest"**></bean>**
11. <!-- 切入点 -->
12. **<bean** name="pointcutBean" class="org.springframework.aop.support.NameMatchMethodPointcut"**>**
13. **<property** name="mappedNames"**>**
14. **<list>**
15. **<value>**sa\***</value>**
16. **</list>**
17. **</property>**
18. **</bean>**
19. <!-- advisor -->
20. **<bean** id="defaultAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor"**>**
21. **<property** name="advice" ref="throws"**></property>**
22. **<property** name="pointcut" ref="pointcutBean"**></property>**
23. **</bean>**
24. <!-- 配置代理对象 -->
25. **<bean** name="proxy" class="org.springframework.aop.framework.ProxyFactoryBean"**>**
26. <!-- 注入目标对象(注入被代理的对象) -->
27. **<property** name="target" ref="target"**></property>**
28. <!-- 注入代理对象需要实现的所有接口 -->
29. **<property** name="interfaces"**>**
30. **<array>**
31. **<value>**com.briup.spring.aop.bean.UserDao**</value>**
32. **</array>**
33. **</property>**
34. <!-- 注入代理对象所要执行的处理程序 -->
35. **<property** name="interceptorNames"**>**
36. **<list>**
37. **<value>**defaultAdvisor**</value>**
38. **<value>**throws**</value>**
39. **</list>**
40. **</property>**
41. **</bean>**
42. **</beans>**

如上边切入点配置：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. <!-- 切入点 -->
2. **<bean** name="pointcutBean" class="org.springframework.aop.support.NameMatchMethodPointcut"**>**
3. **<property** name="mappedNames"**>**
4. **<list>**
5. **<value>**sa\***</value>**
6. **</list>**
7. **</property>**
8. **</bean>**

切面：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. <!-- advisor -->
2. **<bean** id="defaultAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor"**>**
3. **<property** name="advice" ref="advice"**></property>**
4. **<property** name="pointcut" ref="pointcutBean"**></property>**
5. **</bean>**

#### 7.1.7使用匿名的代理对象

使用匿名的代理对象可以将bean的配置到代理里边，这样就不用为target目标对象配置单独的对象，这样可以直接避免目标对象

配置文件：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. <!-- 配置代理对象将来执行的时候，所要执行的处理程序 -->
7. **<bean** name="advice" class="com.briup.spring.aop.bean.AdviceTest"**></bean>**
8. **<bean** name="throws" class="com.briup.spring.aop.bean.ThrowsAdiviceTest"**></bean>**
9. <!-- 切入点 -->
10. **<bean** name="pointcutBean" class="org.springframework.aop.support.NameMatchMethodPointcut"**>**
11. **<property** name="mappedNames"**>**
12. **<list>**
13. **<value>**sa\***</value>**
14. **</list>**
15. **</property>**
16. **</bean>**
17. <!-- advisor -->
18. **<bean** id="defaultAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor"**>**
19. **<property** name="advice" ref="advice"**></property>**
20. **<property** name="pointcut" ref="pointcutBean"**></property>**
21. **</bean>**
22. <!-- 配置代理对象 -->
23. **<bean** name="proxy" class="org.springframework.aop.framework.ProxyFactoryBean"**>**
24. <!-- 注入目标对象(注入被代理的对象)，使用匿名的方式 -->
25. **<property** name="target"**>**
26. **<span** style="white-space:pre"**>**    **</span><bean** name="target" class="com.xxx.spring.aop.bean.UserDaoImpl"**></bean>**
27. **</property>**
28. <!-- 注入代理对象需要实现的所有接口 -->
29. **<property** name="interfaces"**>**
30. **<array>**
31. **<value>**com.xxx.spring.aop.bean.UserDao**</value>**
32. **</array>**
33. **</property>**
34. <!-- 注入代理对象所要执行的处理程序 -->
35. **<property** name="interceptorNames"**>**
36. **<list>**
37. **<value>**defaultAdvisor**</value>**
38. **<value>**throws**</value>**
39. **</list>**
40. **</property>**
41. **</bean>**
42. **</beans>**

### 7.1.8IntroductionInterceptor

Introduction是个特别的Advice，可以在不修改代码的基础上添加一些方法，可以参见

http://www.iteedu.com/webtech/j2ee/springdiary/35.php

http://go12345.iteye.com/blog/352745

### 8.自动代理实现

##### 8.1.BeanNameAutoProxyCreator

Spring允许使用自动代理的bean定义，他可以自动代理选定bean,这样我么就不用为代理对象声明接口，或者没有实现接口的时候，使用CGLIB代理

配置文件：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. <!-- 配置委托类 -->
7. **<bean** name="target" class="com.xxx.spring.aop.bean.UserDaoImpl"**></bean>**
8. <!-- 配置代理对象将来执行的时候，所要执行的处理程序 -->
9. **<bean** name="before" class="com.xxx.spring.aop.bean.BeforeTest"**></bean>**
10. <!-- 配置代理对象 -->
11. **<bean** name="proxy" class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator"**>**
12. <!-- 注入需要被代理的对象名字,会代理所有以targ开始的bean -->
13. **<property** name="beanNames"**>**
14. **<list>**
15. **<value>**targ\***</value>**
16. **</list>**
17. **</property>**
18. <!-- 注入advice或者advisor -->
19. **<property** name="interceptorNames"**>**
20. **<list>**
21. **<value>**before**</value>**
22. **</list>**
23. **</property>**
24. **</bean>**
25. **</beans>**

通过，自动代理，proxy会自动帮我们代理所有以targ开头的目标委托类

实现类：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. //委托类
2. **public** **class** UserDaoImpl **implements** UserDao {
3. @Override
4. **public** **void** saveUser(){
5. System.out.println("保存用户");
6. }
8. @Override
9. **public** **void** deleteUser() {
10. System.out.println("删除用户");
11. }
13. @Override
14. **public** **void** updateUser() {
15. System.out.println("更新用户");
16. }
18. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. @Test
2. **public** **void** autoAdvisor(){
3. BeanFactory factory = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap2/autoAdvisor.xml");
4. UserDao userDao = factory.getBean("proxy", UserDao.**class**);//autoAdvisor只能通过委托类的名字来拿
5. userDao.saveUser();
6. userDao.deleteUser();
7. userDao.updateUser();
8. }

结果：

**[plain]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. version 1.0 author tom saveUser is execute
2. 保存用户
3. version 1.0 author tom deleteUser is execute
4. 删除用户
5. version 1.0 author tom updateUser is execute
6. 更新用户

##### 8.2DefaultAdvisorAutoProxyCreator

使用DefaultAdvisorAutoProxyCreator我们可以不用显示的指定advisor的bean定义

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. <!-- 配置委托类 -->
7. **<bean** name="target" class="com.xxx.spring.aop.bean.UserDaoImpl"**></bean>**
8. <!-- 配置代理对象将来执行的时候，所要执行的处理程序 -->
9. **<bean** name="before" class="com.xxx.spring.aop.bean.BeforeTest"**></bean>**
10. <!-- 配置advisor -->
11. <!-- 作用:筛选要拦截的方法 -->
12. **<bean** name="advisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"**>**
13. <!-- 注入advice -->
14. **<property** name="advice" ref="before"**></property>**
15. <!-- 注入需要被拦截的目标对象中的方法 -->
16. **<property** name="patterns"**>**
17. **<list>**
18. **<value>**.\*deleteUser**</value>**
19. **</list>**
20. **</property>**
21. **</bean>**
23. <!-- 配置代理对象,当前IoC容器中自动应用，不用显示应用advisor的bean定义 -->
24. **<bean** name="proxy"  class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"**></bean>**
25. **</beans>**

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. @Test
2. **public** **void** autoProxyByName(){
3. BeanFactory factory = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap2/autoProxyByName.xml");
4. UserDao userDao = factory.getBean("target", UserDao.**class**);  //自动代理只能通过委托类的id值来拿
5. userDao.saveUser();
6. userDao.deleteUser();
7. userDao.updateUser();
8. }

结果：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52504633" \o "view plain) [copy](http://blog.csdn.net/u014292162/article/details/52504633" \o "copy)

1. 保存用户
2. version 1.0 author tom deleteUser is execute
3. 删除用户
4. 更新用户

参考文章：

<http://blog.csdn.net/abcd898989/article/details/50809321>

http://blog.csdn.net/peng658890/article/details/7223046