*摘要: 基于注解的Spring AOP的配置和使用*

 AOP是OOP的延续，是Aspect Oriented Programming的缩写，意思是面向切面编程。可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。AOP实际是GoF设计模式的延续，设计模式孜孜不倦追求的是调用者和被调用者之间的解耦，AOP可以说也是这种目标的一种实现。

我们现在做的一些非业务，如：日志、事务、安全等都会写在业务代码中(也即是说，这些非业务类横切于业务类)，但这些代码往往是重复，复制——粘贴式的代码会给程序的维护带来不便，AOP就实现了把这些业务需求与系统需求分开来做。这种解决的方式也称代理机制。

先来了解一下AOP的相关概念，《Spring参考手册》中定义了以下几个AOP的重要概念，结合以上代码分析如下：

**切面（Aspect）：**官方的抽象定义为“一个关注点的模块化，这个关注点可能会横切多个对象”，在本例中，“切面”就是类TestAspect所关注的具体行为，例如，AServiceImpl.barA()的调用就是切面TestAspect所关注的行为之一。“切面”在ApplicationContext中<aop:aspect>来配置。

**连接点（Joinpoint）** ：程序执行过程中的某一行为，例如，UserService.get的调用或者UserService.delete抛出异常等行为。

**通知（Advice）** ：“切面”对于某个“连接点”所产生的动作，例如，TestAspect中对com.spring.service包下所有类的方法进行日志记录的动作就是一个Advice。其中，一个“切面”可以包含多个“Advice”，例如ServiceAspect。

**切入点（Pointcut）** ：匹配连接点的断言，在AOP中通知和一个切入点表达式关联。例如，TestAspect中的所有通知所关注的连接点，都由切入点表达式execution(\* com.spring.service.\*.\*(..))来决定。

**目标对象（Target Object）** ：被一个或者多个切面所通知的对象。例如，AServcieImpl和BServiceImpl，当然在实际运行时，Spring AOP采用代理实现，实际AOP操作的是TargetObject的代理对象。

**AOP代理（AOP Proxy）** ：在Spring AOP中有两种代理方式，JDK动态代理和CGLIB代理。默认情况下，TargetObject实现了接口时，则采用JDK动态代理，例如，AServiceImpl；反之，采用CGLIB代理，例如，BServiceImpl。强制使用CGLIB代理需要将 <aop:config>的 proxy-target-class属性设为true。

**通知（Advice）类型：**

**前置通知（Before advice）：**在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext中在<aop:aspect>里面使用<aop:before>元素进行声明。例如，TestAspect中的doBefore方法。

**后置通知（After advice）：**当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。ApplicationContext中在<aop:aspect>里面使用<aop:after>元素进行声明。例如，ServiceAspect中的returnAfter方法，所以Teser中调用UserService.delete抛出异常时，returnAfter方法仍然执行。

**返回后通知（After return advice）：**在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext中在<aop:aspect>里面使用<after-returning>元素进行声明。

**环绕通知（Around advice）：**包围一个连接点的通知，类似Web中Servlet规范中的Filter的doFilter方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext中在<aop:aspect>里面使用<aop:around>元素进行声明。例如，ServiceAspect中的around方法。

**抛出异常后通知（After throwing advice）：**在方法抛出异常退出时执行的通知。ApplicationContext中在<aop:aspect>里面使用<aop:after-throwing>元素进行声明。例如，ServiceAspect中的returnThrow方法。

**注：**可以将多个通知应用到一个目标对象上，即可以将多个切面织入到同一目标对象。

使用Spring AOP可以基于两种方式，一种是比较方便和强大的注解方式，另一种则是中规中矩的xml配置方式。

先说注解，使用注解配置Spring AOP总体分为两步，第一步是在xml文件中声明激活自动扫描组件功能，同时激活自动代理功能（同时在xml中添加一个UserService的普通服务层组件，来测试AOP的注解功能）：

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.1.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<!-- 激活组件扫描功能,在包cn.ysh.studio.spring.aop及其子包下面自动扫描通过注解配置的组件 -->

<context:component-scan base-package="cn.ysh.studio.spring.aop"/>

<!-- 激活自动代理功能 -->

<aop:aspectj-autoproxy proxy-target-class="true"/>

<!-- 用户服务对象 -->

<bean id="userService" class="cn.ysh.studio.spring.aop.service.UserService" />

</beans>

第二步是为Aspect切面类添加注解：

package cn.ysh.studio.spring.aop.aspect;

import org.apache.commons.logging.Log;import org.apache.commons.logging.LogFactory;import org.aspectj.lang.JoinPoint;import org.aspectj.lang.ProceedingJoinPoint;import org.aspectj.lang.annotation.After;import org.aspectj.lang.annotation.AfterReturning;import org.aspectj.lang.annotation.AfterThrowing;import org.aspectj.lang.annotation.Around;import org.aspectj.lang.annotation.Aspect;import org.aspectj.lang.annotation.Before;import org.aspectj.lang.annotation.Pointcut;import org.springframework.stereotype.Component;

/\*\*

 \* 系统服务组件Aspect切面Bean

 \* @author Shenghany

 \* @date 2013-5-28

 \*///声明这是一个组件@Component//声明这是一个切面Bean@Aspectpublic class ServiceAspect {

private final static Log log = LogFactory.getLog(ServiceAspect.class);

//配置切入点,该方法无方法体,主要为方便同类中其他方法使用此处配置的切入点

@Pointcut("execution(\* cn.ysh.studio.spring.aop.service..\*(..))")

public void aspect(){ }

/\*

 \* 配置前置通知,使用在方法aspect()上注册的切入点

 \* 同时接受JoinPoint切入点对象,可以没有该参数

 \*/

@Before("aspect()")

public void before(JoinPoint joinPoint){

if(log.isInfoEnabled()){

log.info("before " + joinPoint);

}

}

//配置后置通知,使用在方法aspect()上注册的切入点

@After("aspect()")

public void after(JoinPoint joinPoint){

if(log.isInfoEnabled()){

log.info("after " + joinPoint);

}

}

//配置环绕通知,使用在方法aspect()上注册的切入点

@Around("aspect()")

public void around(JoinPoint joinPoint){

long start = System.currentTimeMillis();

try {

((ProceedingJoinPoint) joinPoint).proceed();

long end = System.currentTimeMillis();

if(log.isInfoEnabled()){

log.info("around " + joinPoint + "\tUse time : " + (end - start) + " ms!");

}

} catch (Throwable e) {

long end = System.currentTimeMillis();

if(log.isInfoEnabled()){

log.info("around " + joinPoint + "\tUse time : " + (end - start) + " ms with exception : " + e.getMessage());

}

}

}

//配置后置返回通知,使用在方法aspect()上注册的切入点

@AfterReturning("aspect()")

public void afterReturn(JoinPoint joinPoint){

if(log.isInfoEnabled()){

log.info("afterReturn " + joinPoint);

}

}

//配置抛出异常后通知,使用在方法aspect()上注册的切入点

@AfterThrowing(pointcut="aspect()", throwing="ex")

public void afterThrow(JoinPoint joinPoint, Exception ex){

if(log.isInfoEnabled()){

log.info("afterThrow " + joinPoint + "\t" + ex.getMessage());

}

}

}

测试代码：

package cn.ysh.studio.spring.aop;

import org.apache.commons.logging.Log;import org.apache.commons.logging.LogFactory;import org.springframework.context.ApplicationContext;import org.springframework.context.support.ClassPathXmlApplicationContext;

import cn.ysh.studio.spring.aop.service.UserService;import cn.ysh.studio.spring.mvc.bean.User;

/\*\*

 \* Spring AOP测试

 \* @author Shenghany

 \* @date 2013-5-28

 \*/public class Tester {

private final static Log log = LogFactory.getLog(Tester.class);

public static void main(String[] args) {

//启动Spring容器

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

//获取service组件

UserService service = (UserService) context.getBean("userService");

//以普通的方式调用UserService对象的三个方法

User user = service.get(1L);

service.save(user);

try {

service.delete(1L);

} catch (Exception e) {

if(log.isWarnEnabled()){

log.warn("Delete user : " + e.getMessage());

}

}

}

}

控制台输出如下：

 INFO [spring.aop.aspect.ServiceAspect:40] before execution(User cn.ysh.studio.spring.aop.service.UserService.get(long))

 INFO [spring.aop.service.UserService:19] getUser method . . .

 INFO [spring.aop.aspect.ServiceAspect:60] around execution(User cn.ysh.studio.spring.aop.service.UserService.get(long)) Use time : 42 ms!

 INFO [spring.aop.aspect.ServiceAspect:48] after execution(User cn.ysh.studio.spring.aop.service.UserService.get(long))

 INFO [spring.aop.aspect.ServiceAspect:74] afterReturn execution(User cn.ysh.studio.spring.aop.service.UserService.get(long))

 INFO [spring.aop.aspect.ServiceAspect:40] before execution(void cn.ysh.studio.spring.aop.service.UserService.save(User))

 INFO [spring.aop.service.UserService:26] saveUser method . . .

 INFO [spring.aop.aspect.ServiceAspect:60] around execution(void cn.ysh.studio.spring.aop.service.UserService.save(User)) Use time : 2 ms!

 INFO [spring.aop.aspect.ServiceAspect:48] after execution(void cn.ysh.studio.spring.aop.service.UserService.save(User))

 INFO [spring.aop.aspect.ServiceAspect:74] afterReturn execution(void cn.ysh.studio.spring.aop.service.UserService.save(User))

 INFO [spring.aop.aspect.ServiceAspect:40] before execution(boolean cn.ysh.studio.spring.aop.service.UserService.delete(long))

 INFO [spring.aop.service.UserService:32] delete method . . .

 INFO [spring.aop.aspect.ServiceAspect:65] around execution(boolean cn.ysh.studio.spring.aop.service.UserService.delete(long)) Use time : 5 ms with exception : spring aop ThrowAdvice演示

 INFO [spring.aop.aspect.ServiceAspect:48] after execution(boolean cn.ysh.studio.spring.aop.service.UserService.delete(long))

 INFO [spring.aop.aspect.ServiceAspect:74] afterReturn execution(boolean cn.ysh.studio.spring.aop.service.UserService.delete(long))

 WARN [studio.spring.aop.Tester:32] Delete user : Null return value from advice does not match primitive return type for: public boolean cn.ysh.studio.spring.aop.service.UserService.delete(long) throws java.lang.Exception

可以看到，正如我们预期的那样，虽然我们并没有对UserSerivce类包括其调用方式做任何改变，但是Spring仍然拦截到了其中方法的调用，或许这正是AOP的魔力所在。

再简单说一下xml配置方式，其实也一样简单：

<?xml version="1.0" encoding="UTF-8"?><beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.1.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<!-- 系统服务组件的切面Bean -->

<bean id="serviceAspect" class="cn.ysh.studio.spring.aop.aspect.ServiceAspect"/>

<!-- AOP配置 -->

<aop:config>

<!-- 声明一个切面,并注入切面Bean,相当于@Aspect -->

<aop:aspect id="simpleAspect" ref="serviceAspect">

<!-- 配置一个切入点,相当于@Pointcut -->

<aop:pointcut expression="execution(\* cn.ysh.studio.spring.aop.service..\*(..))" id="simplePointcut"/>

<!-- 配置通知,相当于@Before、@After、@AfterReturn、@Around、@AfterThrowing -->

<aop:before pointcut-ref="simplePointcut" method="before"/>

<aop:after pointcut-ref="simplePointcut" method="after"/>

<aop:after-returning pointcut-ref="simplePointcut" method="afterReturn"/>

<aop:after-throwing pointcut-ref="simplePointcut" method="afterThrow" throwing="ex"/>

</aop:aspect>

</aop:config>

</beans>

个人觉得不如注解灵活和强大，你可以不同意这个观点，但是不知道如下的代码会不会让你的想法有所改善：

//配置前置通知,拦截返回值为cn.ysh.studio.spring.mvc.bean.User的方法@Before("execution(cn.ysh.studio.spring.mvc.bean.User cn.ysh.studio.spring.aop.service..\*(..))")

public void beforeReturnUser(JoinPoint joinPoint){

if(log.isInfoEnabled()){

log.info("beforeReturnUser " + joinPoint);

}

}

//配置前置通知,拦截参数为cn.ysh.studio.spring.mvc.bean.User的方法@Before("execution(\* cn.ysh.studio.spring.aop.service..\*(cn.ysh.studio.spring.mvc.bean.User))")

public void beforeArgUser(JoinPoint joinPoint){

if(log.isInfoEnabled()){

log.info("beforeArgUser " + joinPoint);

}

}

//配置前置通知,拦截含有long类型参数的方法,并将参数值注入到当前方法的形参id中@Before("aspect()&&args(id)")

public void beforeArgId(JoinPoint joinPoint, long id){

if(log.isInfoEnabled()){

log.info("beforeArgId " + joinPoint + "\tID:" + id);

}

}

附上UserService的代码(其实很简单)：

package cn.ysh.studio.spring.aop.service;

import org.apache.commons.logging.Log;import org.apache.commons.logging.LogFactory;

import cn.ysh.studio.spring.mvc.bean.User;

/\*\*

 \* 用户服务模型

 \* @author Shenghany

 \* @date 2013-5-28

 \*/public class UserService {

private final static Log log = LogFactory.getLog(UserService.class);

public User get(long id){

if(log.isInfoEnabled()){

log.info("getUser method . . .");

}

return new User();

}

public void save(User user){

if(log.isInfoEnabled()){

log.info("saveUser method . . .");

}

}

public boolean delete(long id) throws Exception{

if(log.isInfoEnabled()){

log.info("delete method . . .");

throw new Exception("spring aop ThrowAdvice演示");

}

return false;

}

}

应该说学习Spring AOP有两个难点，第一点在于理解AOP的理念和相关概念，第二点在于灵活掌握和使用切入点表达式。概念的理解通常不在一朝一夕，慢慢浸泡的时间长了，自然就明白了，下面我们简单地介绍一下切入点表达式的配置规则吧。

通常情况下，表达式中使用”execution“就可以满足大部分的要求。表达式格式如下：

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)

**modifiers-pattern**：方法的操作权限

**ret-type-pattern**：返回值

**declaring-type-pattern**：方法所在的包

**name-pattern**：方法名

**parm-pattern**：参数名

**throws-pattern**：异常

其中，除ret-type-pattern和name-pattern之外，其他都是可选的。上例中，execution(\* com.spring.service.\*.\*(..))表示com.spring.service包下，返回值为任意类型；方法名任意；参数不作限制的所有方法。

**最后说一下通知参数**

可以通过args来绑定参数，这样就可以在通知（Advice）中访问具体参数了。例如，<aop:aspect>配置如下：

<aop:config>

  <aop:aspect id="TestAspect" ref="aspectBean">

   <aop:pointcut id="businessService"

    expression="execution(\* com.spring.service.\*.\*(String,..)) and args(msg,..)" />

    <aop:after pointcut-ref="businessService" method="doAfter"/>

  </aop:aspect></aop:config>

上面的代码args(msg,..)是指将切入点方法上的第一个String类型参数添加到参数名为msg的通知的入参上，这样就可以直接使用该参数啦。

**访问当前的连接点**

在上面的Aspect切面Bean中已经看到了，每个通知方法第一个参数都是JoinPoint。其实，在Spring中，任何通知（Advice）方法都可以将第一个参数定义为**org.aspectj.lang.JoinPoint**类型用以接受当前连接点对象。JoinPoint接口提供了一系列有用的方法， 比如 getArgs() （返回方法参数）、getThis() （返回代理对象）、getTarget() （返回目标）、getSignature() （返回正在被通知的方法相关信息）和 toString() （打印出正在被通知的方法的有用信息）。

来源：http://ntzrj513.blog.163.com/blog/static/27945612201362232315/

appendix:

## Interface Signature

**All Known Subinterfaces:**

[AdviceSignature](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/reflect/AdviceSignature.html), [CatchClauseSignature](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/reflect/CatchClauseSignature.html), [CodeSignature](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/reflect/CodeSignature.html), [ConstructorSignature](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/reflect/ConstructorSignature.html), [FieldSignature](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/reflect/FieldSignature.html), [InitializerSignature](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/reflect/InitializerSignature.html), [LockSignature](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/reflect/LockSignature.html), [MemberSignature](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/reflect/MemberSignature.html),[MethodSignature](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/reflect/MethodSignature.html), [UnlockSignature](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/reflect/UnlockSignature.html)

**All Known Implementing Classes:**

[FieldSignatureImpl](http://www.eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/runtime/reflect/FieldSignatureImpl.html)

public interface

Represents the signature at a join point. This interface parallels java.lang.reflect.Member.

This interface is typically used for tracing or logging applications to obtain reflective information about the join point, i.e. using the j2se 1.4 java.util.logging API

 aspect Logging {

     Logger logger = Logger.getLogger("MethodEntries");

     before(): within(com.bigboxco..\*) && execution(public \* \*(..)) {

         Signature sig = thisJoinPoint.getSignature();

         logger.entering(sig.getDeclaringType().getName(),

                         sig.getName());

     }

 }

More detailed information about a specific kind of signature can be obtained by casting this Signature object into one of its more specific sub-types available inorg.aspectj.lang.reflect.

**See Also:**

Member, java.util.logging.Logger

### getDeclaringTypeName

java.lang.String ()

Returns the fully-qualified name of the declaring type. This is equivalent to calling getDeclaringType().getName(), but caches the result for greater efficiency.

### getName

java.lang.String ()

Returns the identifier part of this signature. For methods this will return the method name.

**See Also:**

Member.getName()

原文地址：[http://www.cnblogs.com/davidwang456/p/4013631.html](http://www.cnblogs.com/davidwang456/p/4013631.html" \t "https://my.oschina.net/sniperLi/blog/_blank)

* 分类：[spring](https://my.oschina.net/sniperLi/blog?catalog=3286476)

* 字数：2988