# 1. 代理前情回顾

在之前的学习过程中，我们提到过代理和AOP。现在回顾一下，然后再重点讲AOP和在Spring中的使用。

在开发中，可能会遇到动态修改执行的方法，或者在方法前后执行一些重复代码的情况。例如控制事务提交、监控方法的执行、添加日志、控制权限等。

上述问题中基本上都是在方法执行前后，来执行一些重复的代码。可利用代理来解决问题，这样来避免编写重复代码，使开发者关注具体业务。

实际开发时不会使用静态代理（因为要书写较多的代理类和其中的许多代理方法，静态代理反而变得更加复杂），一般使用JDK、CGLIB或者javassist实现动态代理。但即便使用动态代理，还是有些细节需要完善。例如一个目标方法既要添加事务控制，也要添加日志记录，此时就要求能为目标对象执行多个代理，且能控制每个代理的执行顺序。下面学习Spring中的AOP就能很好地解决这些问题（Spring也是封装了JDK和CGLIB动态代理）。

# 2. AOP

## 2.1 AOP的概念

AOP（aspect object programming）即“面向切面编程”。虽然Spring官方文档中讲解了很多AOP概念，但直接看是不容易懂的，比较抽象，下面以案例来说明。

现在实现一个简单的“事务控制”AOP案例。

（1）UserDao接口：

|  |
| --- |
| **package** com.sd.dao;  **public interface** UserDao {  **void** save(); *// 保存* **void** update(); *// 更新* **void** delete(); *// 删除* **void** select(); *// 查询* } |

（2）UserDao的实现UserDaoImpl。注意，由于使用代理处理事务，这里是写“业务代码”：

|  |
| --- |
| **package** com.sd.dao;  **public class** UserDaoImpl **implements** UserDao {  @Override  **public void** save() {  System.***out***.println(**"save data"**);  }   @Override  **public void** update() {  System.***out***.println(**"update data"**);  }   @Override  **public void** delete() {  System.***out***.println(**"delete data"**);  }   @Override  **public void** select() {  System.***out***.println(**"select data"**);  } } |

（3）“事务管理器”中提供开启事务和提交事务的代码。实际中这部分代码中，一般先使用ThreadLocal与当前会话连接绑定，然后获取当前线程的连接。

|  |
| --- |
| **package** com.sd.dao;  **public class** TransactionManager {  **public void** beginTransaction() {  System.***out***.println(**"begin transaction"**);  }   **public void** commit() {  System.***out***.println(**"commit"**);  } } |

（4）最后，拿到代理对象并使用：

|  |
| --- |
| **package** com.sd;  **import** com.sd.dao.TransactionManager; **import** com.sd.dao.UserDao; **import** com.sd.dao.UserDaoImpl; **import** java.lang.reflect.InvocationHandler; **import** java.lang.reflect.Method; **import** java.lang.reflect.Proxy;  **public class** Main {  **public static void** main(String[] args) {  UserDao userDao = *getProxyObject*();  userDao.save();  userDao.update();  userDao.delete();  userDao.select();  }   *// 用于获取代理对象的方法* **public static** UserDao getProxyObject() {  UserDao userDao = **new** UserDaoImpl();  **return** (UserDao) Proxy.*newProxyInstance*(userDao.getClass().getClassLoader(),  userDao.getClass().getInterfaces(),  **new** InvocationHandler() {  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  Object obj = **null**;  *// 判断方法是否需要添加事务* **if** (**"save"**.equals(method.getName()) ||  **"update"**.equals(method.getName()) ||  **"delete"**.equals(method.getName()) ) {  TransactionManager tx = **new** TransactionManager();  tx.beginTransaction(); *// 开启事务* obj = method.invoke(userDao, args); *// 执行目标对象方法* tx.commit(); *// 提交事务* } **else** {  obj = method.invoke(userDao, args);  }  **return** obj;  }  });  } } |

执行上述代码，就会发现每执行save、update和delete方法前后，就会“自动”进行事务的开启和提交。这个简单的示例说明了，AOP的主要作用是实现了“业务代码”和“关注点代码”分离。在上述代码中，“业务代码”就是单纯的保存、更新等操作，而“关注点代码”是开发者需要关注的代码，例如事务、权限、日志等等。在开发中，希望开发者专注于业务代码，即花主要的精力来实现业务，而降低开发者对“关注点代码”的花费的时间和精力，这将提高开发效率。而利用AOP也实现这样的效果。

使用AOP实现“业务代码”和“关注点代码”分离的好处：

（1）因为关注点代码通常会被多次使用，因此分离后只需要写一次关注点代码；

（2）让开发者只关注核心的业务处理。

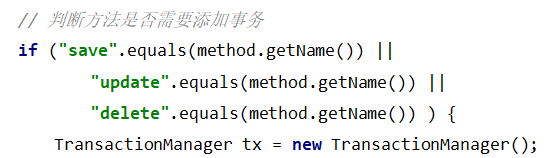
下面结合上述案例，说明一下AOP中提到的概念。

（1）什么是切面（Aspect）。切面就是编程中的日志、事务、权限、安全框架这些东西。例如上面的TransactionManager就是切面类。“切面”类就是指关注点代码形成的类，其中抽取了重复执行的代码。可使用切面类的例子有：事物、日志和权限等。

（2）什么是通知（Advice）。切面中的方法就是通知。例如上述的TransactionManager中的beginTransaction和commit方法就是通知。

（3）什么是连接点（Join point）。被业务开发者调用的方法叫做连接点。例如上述在主类main方法中调用的userDao对象的update、save等方法就是连接点。由于使用代理，连接点的方法体有可能是被代理对象更改过的方法体。

（4）什么是切入点（Cutpoint）。代理中设置的过滤目标对象方法的条件就相当于是设置了切入点。即下面这部分代码：



就相当于是“切入点”。

（5）什么叫“织入（或编织，Weaving）”：形成代理对象的过程就叫织入。

（6）还有针对目标方法而言的前置通知、后置通知、异常通知、最终通知和环绕通知：

前置通知（Before Advice）：在目标方法之前执行的通知；

后置通知（After returning advice）：在目标方法之后执行的通知；

异常通知（After throwing advice）：在目标方法抛出异常后执行的通知；

最终通知（After (finally) advice）：在目标方法执行结束或者异常后执行的通知；

环绕通知（Around advice）：会环绕目标方法执行的通知。

上述提到的这些通知在下面也会举例说明。

## 2.2 利用Spring实现AOP

AOP的实现实际上是使用了动态代理的技术。即在程序运行中，当执行到核心业务代码时，就在业务方法上动态的植入切面类代码（即关注点代码）。

程序怎么知道哪些业务方法需要进行AOP代理呢？因此我们需要使用“切入点”。可以通过切入点表达式来指定哪些类的哪些业务方法需要使用AOP，这样程序运行时就能为这些方法植入切面代码。

Spring中继承了AOP，这是一个重要的功能，因此我们下面结合Spring来实现AOP的示例。Spring中AOP使用下面方式实现动态代理：

（1）若目标对象实现了接口，则使用JDK代理；

（2）若目标对象没有实现接口，则使用cglib代理。

首先，要在项目中使用Spring中对AOP的支持，除了要在项目中引入之前Spring相关的包，还要进入AspectJ相关的jar包。AspectJ是一个实现AOP的Java组件，它比Spring出来还早，Spring利用AspectJ实现AOP并把AOP发扬光大。AspectJ由Eclipse基金会维护，它的下载地址：<http://www.eclipse.org/aspectj/downloads.php> 。解压下载的jar包，将lib目录下的“aspectjrt.jar”和“aspectjweaver.jar”两个包引入到项目中，这样才能真正地在Spring中引用AOP。

### 2.2.1 XML配置实现AOP

这里用XML方式配置实现AOP。我们将2.1节的例子用Spring中的AOP实现。2.1节中的代码基本不需要改变，除了主类中不再需要提供获得代理对象的方法，因为使用Spring的AOP后，代理对象是从Spring中拿到的。

XML应该这样配置（需要引入AOP的名称空间）：

|  |
| --- |
| *<?***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:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"**>  *<!-- 1. 把目标对象的类UserDaoImpl加入到容器中 -->* <**bean id="userDao" class="com.sd.dao.UserDaoImpl"** />  *<!-- 2. 把切面类（即控制事务的切面类）加入到容器中， -->* <**bean id="tx" class="com.sd.dao.TransactionManager"** />  *<!-- 下面是AOP的配置 -->* <**aop:config**>  *<!-- 3. 定义切入点表达式。切入点表达式是为了找到要切入的类和方法（相当于判断某方法是否需要切入拦截）。id还是还切入点的唯一标识 -->* <**aop:pointcut id="pointcut" expression="execution(\* com.sd.dao.UserDaoImpl.\*(..))"** />  *<!-- 4. 定义各切面。ref要引用切面类的bean，这里就是引用上述的tx -->* <**aop:aspect ref="tx"**>  *<!-- 5. 定义各种通知。method表示通知执行的方法，pointcut-ref引用上述的切入点表达式，这样就能最终生成代理类 -->  <!-- aop:before是前置通知 -->* <**aop:before method="beginTransaction" pointcut-ref="pointcut"** />  *<!-- aop:after-returning是后置通知 -->* <**aop:after-returning method="commit" pointcut-ref="pointcut"** />  </**aop:aspect**>  </**aop:config**> </**beans**> |

上述就是先配置目标类bean、切面类bean，再配置AOP。AOP中先配置了切入点，用于让Spring找到要切入的目标方法，具体的下面还会讲。最后，XML中定义了通知Advice，这样，主类就可这样使用到代理对象，不用自己编写代码实现代理：

|  |
| --- |
| **package** com.sd;  **import** com.sd.dao.UserDao; **import** org.springframework.context.ApplicationContext; **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public class** Main {  **public static void** main(String[] args) {  ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  UserDao userDao = (UserDao) ac.getBean(**"userDao"**);  userDao.update();  userDao.select();  } } |

上述方法执行update和select方法时，都会执行“事务”，这是因为切入点表达式现在没有区分到判断是哪个方法（下节详细讲）。

下面是应用Spring实现AOP时，Spring中的执行流程（即原理，以上述为例）：

（1）Spring容器启动时，加载配置的目标类和切面类bean，并进行实例化；

（2）Spring容器解析到aop:config后，先解析切入点表达式，按照切入点表达式匹配Spring容器中的bean，本例中，就能匹配到UserDaoImpl这个bean。

（3）如果匹配成功，则为bean创建代理对象。也就是说，要想Spring创建代理对象，则目标对象的类要先加入到bean中，且切入点表达式能匹配到。那么Spring如何创建代理对象呢？其实是根据其中配置的具体通知Advice来的。例如本例中配置了前置通知和后置通知，则Spring创建的代理对象，会在目标方法前执行前置通知，再调用目标方法，最后调用后置通知方法。通知中指定了切入点。

说明：当然，如果Spring检测到一个类是切面类的话，就不会把他当做目标类。例子中切面类也在切入点表达式的匹配范围中，但是Spring并不会为切面类再加上“切面”。

（4）当使用者用ac.getBean()获取对象时，如果该对象有代理对象，则返回代理对象，否则返回对象本身。

### 2.2.2 切入点表达式

切入点表达式可以对指定的“方法”拦截，从而给方法所在的类生成代理对象。关于切入点表达式的学习，可以在Spring文档的“11.3 Schema-based AOP support”这章中详细学习（Declaring a pointcut一节）。

切入点表达式的作用就是确定目标类的目标方法，否则使用程序判断哪些方法需要代理，比较麻烦。切入点可确定到方法级别，那么先看一个方法定义的最全信息是什么样子的。最详细的方法定义应该示例如下（方法名前使用全限定类型名）：

|  |
| --- |
| **public void** com.sd.AopTest.setStr(String str) **throws** Exception {...} |

而切入点表达式的格式是：

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

它们是一一对应的关系：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 描述 | 无 | 修饰符 | 返回值类型 | 所在包和类 | 方法名 | 参数 | 异常 |
| 方法签名 |  | public | void | com.sd.AopTest | setStr | (String str) | throws Exception |
| 切入点表达式 | execution( | modifiers-pattern? | ret-type-pattern | declaring-type-pattern? | name-pattern | (param-pattern) | throws-pattern?) |

其中，切入点表达式中，加问号的部分是可以省略的，其余的部分是必须有的。另外，切入点表达式中可以使用“\*”作为通配符，在方法参数中，可以使用“..”表示匹配任意多个、任意类型的参数。

切入点表达式的几个例子：

|  |  |
| --- | --- |
| 切入点表达式 | 匹配的方法 |
| execution(public \* \*(..)) | 匹配所有的public方法 |
| execution(\* save\*(..)) | 匹配所有的以save开头的方法 |
| execution(\* com.zhang.UserTest.save(..)) | 匹配UserTest类中的save方法 |
| execution(\* com.zhang.UserTest.\*(..)) | 匹配UserTest类的所有方法 |
| execution(\* com..\*.\*(..)) | 匹配com包及其子包下所有类的方法 |
| execute(\* com.aop.\*.service.\*.\*(..) | 匹配com.aop包下任意一个包下的service包下的任意类的方法 |
| execute(\* com.aop..service..\*.\*(..)) | com.aop包及其子包一直到service包，然后service包下的所有类的所有方法。这种适合于多模块的时候使用（每个模块都有service包）。 |

另外在Spring写切入点表达式时，还可使用逻辑表达式。例如：

|  |
| --- |
| <**aop:pointcut id="pt1" expression="execution(\* com.zhang.UserTest.save()) || execution(\* com.zhang.UserTest.delete())"** /> *<!-- 上述中也能使用 or。除此之外，也能使用and，但是&&必须要转义：&amp;&amp; --> <!-- 取非使用not或者! -->* <**aop:pointcut id="pt2" expression="not execution(\* com.zhang.UserTest.save())"** /> |

其实，并不一定先要在XML中配置好切入点表达式，然后在通知中引用。实际上，可直接在通知中书写一个特定的表达式，这样会按照通知自己的表达式来匹配。例如（使用pointcut属性即可）：

|  |
| --- |
| <**aop:before method="beginTransaction" pointcut="execution(\* com.sd.dao.\*.\*(..))"** /> |

### 2.2.3 SpringAOP中的通知

前面示例中使用了前置通知和后置通知，分别使用aop:before标签和aop:returning标签。

其实在Spring AOP中，所有的通知（即切面类方法）中都可拿到一个JoinPoint类型的参数，这个参数中封装了连接点（目标方法）信息。

只要在通知中的方法接收该类型参数即可，例如：

|  |
| --- |
| **public void** beginTransaction(JoinPoint joinPoint) {  *// 得到连接点的方法名（客户端调用的代码）* System.***out***.println(joinPoint.getSignature().getName());  *// 得到目标类的字节码* System.***out***.println(joinPoint.getTarget().getClass());  *// 也可获得其他信息等，自行研究* System.***out***.println(**"begin transaction"**); } |

这样再运行程序就会有信息在“begin transaction”前输出。

（1）关于后置通知

后置通知能得到目标对象的返回值。先要在后置通知的节点中配置returning属性，指明传递的返回值的参数名。例如：

|  |
| --- |
| <**aop:after-returning method="commit" pointcut-ref="pointcut" returning="ret"** /> |

并将目标方法改成带有返回值的，这样切面类的后置通知方法可这样拿到值：

|  |
| --- |
| *// commit这个后置通知的方法中，可以直接使用“String ret”接收参数。* **public void** commit(String ret) {  System.***out***.println(**"目标方法返回值："** + ret);  System.***out***.println(**"commit"**); } |

另外需要注意的是，如果目标方法中执行出现了异常，那么后置通知将不会执行。

（2）最终通知。最终通知使用“aop:after”标签，无论目标方法是否抛出异常，都会执行最终通知。注意最终通知中不能使用returning属性，通知方法也不能接受除了JoinPoint类型的参数。

（3）异常通知。异常通知方法可以接受目标方法抛出的异常并执行通知。异常通知使用“aop:after-throwing”标签配置。先在XML中配置异常传递到通知中的参数名：

|  |
| --- |
| <**aop:after-throwing method="afterEx" pointcut-ref="pointcut" throwing="ex"** /> |

然后异常通知中接收这个参数名就能拿到抛出的异常对象：

|  |
| --- |
| *// 方法中可接收joinPoint和ex，ex就是异常* **public void** afterEx(JoinPoint joinPoint, Throwable ex) {  System.***out***.println(**"出现了异常"**);  ex.printStackTrace(); } |

（4）环绕通知。环绕通知可以控制目标方法的执行。环绕通知使用“aop:around”标签来配置，例如配置了一个执行around方法的环绕通知：

|  |
| --- |
| <**aop:around method="around" pointcut-ref="pointcut"** /> |

为什么说环绕通知能控制目标方法呢？因为在调用目标方法之前先进入环绕通知，环绕通知可接收一个类型为“ProceedingJoinPoint”的参数。“ProceedingJoinPoint”是JoinPoint的子接口类型，在环绕方法中，只有调用了该类型参数的proceed()方法，目标方法才会执行，否则目标方法不会执行，当然其配置的后置方法也不会执行。因此环绕方法的作用是控制目标方法是否执行。例如：

|  |
| --- |
| **public void** around(ProceedingJoinPoint pjp) **throws** Throwable {  System.***out***.println(**"目标方法执行前"**);  pjp.proceed();  System.***out***.println(**"目标方法执行后"**); } |

如果不调用pjp.proceed()方法，则目标方法不会执行。环绕方法是比较常用的，例如进行权限控制时，没有权限就不允许进入目标方法。

像Shiro等主流框架都使用Spring的AOP实现了框架的一些基础功能，开发者使用时大概知道原理即可，大概知道是怎么实现的即可。

### 2.2.4 Spring AOP的小总结

从上述中AOP的例子可感受到，目标类和切面类（通知）是完全可以实现低耦合的，Spring帮我们做了很多工作。可以说，“代理方法 = 目标方法 + 通知”，因此我们只要在Spring中配置好，就能生成代理对象。

借助Spring AOP实现目标类和切面类的低耦合是非常有意义的，这样在开发中，关注业务的开发者和负责各种切面功能的开发者可以各自独立地进行开发，而最终运行时，Spring会自动生成代理类，把切面类和目标类“结合”在一起了。也就是说开发时彼此透明，但最终需要的功能都完整地整合在一起了，提高了开发效率，各司其职。并且，切面类既可以加上，也可以取消，而不影响主要的业务代码，即可拔插式的，不会影响别的东西。

Spring用这种方式将AOP发扬光大了。为什么使用AOP？这和Struts中使用拦截器是一样的道理。Struts中，许多的拦截器就相当是一些切面类，拦截器执行完后，会执行开发者编写的Action，Action就相当于是目标类，最后再返回到拦截器中。

使用AOP的作用就是把一些共用的、不变的、非业务逻辑的代码抽取出来，形成切面提供给程序使用，这样开发者就不需要再编写这部分代码了，减少了工作，因为自动就使用了这些AOP的功能。

### 2.2.5 注解配置实现AOP

除了用XML配置实现AOP外，也能使用注解的方式实现AOP。这里还是以2.1节的例子作为示例来实现注解版本的AOP。

（1）首先在在XML中开启注解扫描和自动创建AOP代理方式（需要引入context和aop名称空间）：

|  |
| --- |
| *<?***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:aop="http://www.springframework.org/schema/aop"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd"**>  *<!-- 指定注解扫描的包。AOP相关的包必须在扫描之中 -->* <**context:component-scan base-package="com.sd.dao"** />  *<!-- 自动创建AOP代理 -->* <**aop:aspectj-autoproxy** /> </**beans**> |

（2）在类中加入注解。UserDao接口是不需要变的。为了把目标类和切面类加入到bean的管理，因此UserDaoImpl和TransactionManager需要用@Component注解加入到bean中。

除此之外就是AOP注解的配置的，AOP注解的配置全部写在切面类中。切面类会使用如下注解：

@Aspect：在切面类上使用，指定该类为一个切面类。

@PointCut("切入点表达式")：用于声明一个切入点表达式。在一个方法上使用，方法名就可来标识这个切入点表达式。由于该注解只是声明一个表达式，因此对应的方法只要空实现即可。例如：

|  |
| --- |
| @Pointcut(value = **"execution(\* com.sd.dao.\*.\*(..))"**) **public void** pointcut() {} |

相当于XML的配置：

|  |
| --- |
| <**aop:pointcut id="pointcut()" expression="execution(\* com.sd.dao.UserDaoImpl.\*(..))"** /> |

注意上述的id是带有方法括号的“pointcut()”，因此下面通知注解中引用时，也要加上括号。下面就是几个对应的通知注解：

（1）@Before("切入点表达式或引用上述@PointCut中对应方法")：用在方法上，指定一个前置通知。

（2）@AfterReturning("同上")：用在方法上，指定一个后置通知。

（3）@After("同上")：用在方法上，指定一个最终通知。

（4）@AfterThrowing("同上")：用在方法上，指定一个异常通知。

（5）@Around("同上")：用在方法上，指定一个环绕通知。

因此，UserDaoImpl中代码为：

|  |
| --- |
| **package** com.sd.dao;  **import** org.springframework.stereotype.Component;  @Component(**"userDao"**) **public class** UserDaoImpl **implements** UserDao {  // 只是用了一个@Component注解，其余的代码和之前都一样 } |

TransactionManager中的代码为：

|  |
| --- |
| **package** com.sd.dao;  **import** org.aspectj.lang.annotation.AfterReturning; **import** org.aspectj.lang.annotation.Aspect; **import** org.aspectj.lang.annotation.Before; **import** org.aspectj.lang.annotation.Pointcut; **import** org.springframework.stereotype.Component;  @Component(**"tx"**) @Aspect *// 切面类使用@Aspect注解* **public class** TransactionManager {  *// 定义切入点表达式* @Pointcut(value = **"execution(\* com.sd.dao.\*.\*(..))"**)  **public void** pointcut() {}   *// 前置通知，引用pointcut()* @Before(**"pointcut()"**)  **public void** beginTransaction() {  System.***out***.println(**"begin transaction"**);  }   *// 后置通知，引用pointcut()* @AfterReturning(**"pointcut()"**)  **public void** commit() {  System.***out***.println(**"commit"**);  } } |

主类测试运行的代码是不变的。这就是使用注解进行AOP的配置，程序运行正常。

其实AOP的注解用起来还是比较麻烦和不直观的，实际中还是XML用得比较多点。

### 2.2.6 Spring AOP配置多个切面和注意事项

可利用SpringAOP给一个目标类配置多个切面。例如现在增加一个Logger（日志）切面类：

|  |
| --- |
| **package** com.sd.dao;  **import** org.aspectj.lang.JoinPoint;  **public class** Logger {  **public void** startLog(JoinPoint joinPoint) {  String methodName = joinPoint.getSignature().getName();  System.***out***.println(**"开始执行方法："** + methodName);  }   **public void** endLog(JoinPoint joinPoint) {  String methodName = joinPoint.getSignature().getName();  System.***out***.println(**"方法执行结束："** + methodName);  } } |

想给UserDaoImpl既加上Logger切面，又加上TransactionManager切面，则这样配置：

|  |
| --- |
| *<?***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:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"**>  *<!-- UserDaoImpl和两个切面类加入到bean -->* <**bean id="userDao" class="com.sd.dao.UserDaoImpl"** />  <**bean id="tx" class="com.sd.dao.TransactionManager"** />  <**bean id="logger" class="com.sd.dao.Logger"** />  <**aop:config**>  *<!-- 切入点表达式 -->* <**aop:pointcut id="pt" expression="execution(\* com.sd.dao.\*.\*(..))"** />  *<!-- 日志切面的相关通知 -->* <**aop:aspect ref="logger"**>  <**aop:before method="startLog" pointcut-ref="pt"** />  <**aop:after-returning method="endLog" pointcut-ref="pt"** />  </**aop:aspect**>  *<!-- 事务切面的相关通知 -->* <**aop:aspect ref="tx"**>  <**aop:before method="beginTransaction" pointcut-ref="pt"** />  <**aop:after-returning method="commit" pointcut-ref="pt"** />  </**aop:aspect**>  </**aop:config**> </**beans**> |

上述就是配置了两个切面，都有前置通知和后置通知。主类测试代码是一样的。最后运行结果是这样的：

|  |
| --- |
| 开始执行方法：delete  begin transaction  delete data  commit  方法执行结束：delete |

由于都有前置通知和后置通知，且logger切面的配置靠前，因此先执行logger的前置通知，再执行tx的前置通知，然后执行目标方法，最后执行tx的后置通知和logger的后置通知。整个执行的过程和Struts拦截器的执行过程十分相似。当然，如果是tx切面的配置靠前，则先执行tx的前置通知，即执行的顺序就是“tx前置通知 - logger前置通知 - 目标方法 - logger后置通知 - tx后置通知”。

最后，再将Spring AOP中关于代理的两个注意点：

（1）如果目标类实现了接口，则SpringAOP生成代理对象时，默认则采用JDK代理的方式生成代理类；如果目标类没有实现接口，则采用CGLIB方式生成代理类（子类方式）。这样的选择是在Spring内部完成的。可通过断点调试验证（关于CGLIB方式的验证，只要把例子中的UserDao接口删除，UserDaoImpl不实现UserDao接口，得到的bean直接用UserDaoImpl类型接收即可验证，程序会正常运行，AOP依然有效，这时就是使用了CGLIB子类代理）。

说明：输出代理对象的字节码（System.out.println(userDao.getClass())），若显示的结果为“class com.sun.proxy.$Proxy12”这样的字样，则其是JDK代理类型；如果显示的结果为“class com.sd.UserDao$$EnhancerBySpringCGLIB$$a270ced9”，其中有“CGLIB”字样的。说明Spring对于没有实现接口的类，确实是使用了cglib代理。但是我们并不需要在Spring中引入cglib的jar包，因为Spring中已经集成了cglib，只是包名和cglib有所差别。

（2）在一些情况下，需要强制使用CGLIB代理，那么需要在aop:config标签中配置proxy-target-class为true（默认为false，当目标类实现接口时就用JDK代理），这样就是强制使用CGLIB作为代理：

|  |
| --- |
| <**aop:config proxy-target-class="true"**>  *<!-- AOP的配置...... -->* </**aop:config**> |

这样，不管目标类是否实现了接口，Spring AOP都会使用CGLIB进行子类代理。

那么什么情况下必须要强制使用CGLIB代理呢？例如，当你的目标类已经实现了序列化等接口时，就不能再使用默认的JDK代理了。因为序列化接口中根本没有方法，只是一个标记接口，无法实现代理。同理，当目标类实现的接口并不是你想要的代理接口时，也应该强制使用CGLIB子类代理。

从案例中可以感受到Spring AOP提供的功能强大之处。以后实现代理可直接用Spring及AOP。