复习了一下Spring，发现忘了很多东西了。特别是AOP这一块儿，于是认真看了看，写篇文章总结总结。

**概念**

AOP（Aspect Oriented Programming），即面向切面编程（也叫面向方面编程，面向方法编程）。其主要作用是，在不修改源代码的情况下给某个或者一组操作添加额外的功能。像日志记录，事务处理，权限控制等功能，都可以用AOP来“优雅”地实现，使这些额外功能和真正的业务逻辑分离开来，软件的结构将更加清晰。AOP是OOP的一个强有力的补充。

**术语**

AOP的术语不太直观，Spring文档中也没有给一个确切的定义，所以重在理解。

* Join Point: Spring AOP中，join point就是一个方法。（通俗来讲就是起作用的那个方法）。
* Pointcut: 用来指定join point（通俗来讲就是描述的一组符合某个条件的join point）。通常使用pointcut表达式来限定joint point，Spring默认使用AspectJ pointcut expression language。
* Advice: 在join point上特定的时刻执行的操作，Advice有几种不同类型，下文将会讨论（通俗地来讲就是起作用的内容和时间点）。
* Introduction：给对象增加方法或者属性。
* Target object: Advice起作用的那个对象。
* AOP proxy: 为实现AOP所生成的代理。在Spring中有两种方式生成代理:JDK代理和CGLIB代理。
* Aspect: 组合了Pointcut与Advice，在Spring中有时候也称为Advisor。某些资料说Advisor是一种特殊的Aspect，其区别是Advisor只能包含一对pointcut和advice，但是aspect可以包含多对。AOP中的aspect可以类比于OOP中的class。
* Weaving：将Advice织入join point的这个过程。

**Advice的类型**

* Before advice:  执行在join point之前的advice,但是它不能阻止joint point的执行流程，除非抛出了一个异常（exception）。
* After returning advice: 执行在join point这个方法返回之后的advice。
* After throwing advice: 执行在join point抛出异常之后的advice。
* After(finally) advice: 执行在join point返回之后或者抛出异常之后的advice，通常用来释放所使用的资源。
* Around advice: 执行在join point这个方法执行之前与之后的advice。

**两种代理**

Spring AOP是基于代理机制的。上文说到，Spring AOP通过JDK Proxy和CGLIB Proxy两种方法实现代理。

如果target object没有实现任何接口，那么Spring将使用CGLIB来实现代理。CGLIB是一个开源项目,它是一个强大的,高性能,高质量的Code生成类库,它可以在运行期扩展Java类与实现Java接口。

如果target object实现了一个以上的接口，那么Spring将使用JDK Proxy来实现代理，因为Spring默认使用的就是JDK Proxy，并且JDK Proxy是基于接口的。这也是Spring提倡的面向接口编程。当然，你也可以强制使用CGLIB来进行代理，但是这样可能会造成性能上的下降。

**Pointcut expression**

Pointcut通过pointcut expression来描述，有若干种限定词。由于Pointcut的定义在[Spring文档7.2.3 Declaring a pointcut](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/aop.html" \l "aop-pointcuts" \t "_blank)中写得比较详细，所以在此不再赘述。

**Spring AOP的使用**

我们可以通过三种方式来使用Spring AOP，它们分别是：@Aspect-based（Annotation），Schema-based（XML），以及底层的Spring AOP API。

无论基于aspectj还是ｘｍｌ的ａｏｐ　三点信息组装起来:

1:aspect

2.pointcut

3;advice

**@Aspect-based (Annotation)**

Annotaion是最常用的方式。

**配置**

首先，我们应该在配置文件中增加对Annotation的支持。

假设我们的配置文件是classpath下的applicationContext.xml，添加如下片段:

<aop:aspectj-autoproxy />

**业务逻辑类**

假设我们有一个UserManager类，这个类负责处理业务逻辑。类的定义如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class UserManager {            /\*这个方法需要一个参数\*/            public void addUser(String user) {                      System.out.println("addUser(String str) method is executed!");            }              public void deleteUser() {                      System.out.println("deleteUser() method is executed!");            }            /\*这个方法返回一个字符串\*/            public String getUser() {                      System.out.println("getUser() method is executed!");                      return "Hello";            }            /\*这个方法抛出一个异常\*/            public void editUser() throws Exception {                      throw new Exception("something is wrong.");            }  } |

这是一个很普通的Java对象，看不出任何Spring AOP的痕迹，这也是Spring低侵入式设计的体现。

**切面（Aspect）类**

为了给业务逻辑增加额外功能，我们需要定义一个切面类，切面类里包含了pointcut和advice。假设我们的切面类是ExampleAspect，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | @Aspect  public class ExampleAspect {              @Pointcut("execution(\* com.psjay.example.spring.aop.\*.\*(..))")            public void aPointcut() {            }              @Before("aPointcut()")            public void beforeAdvice() {                      System.out.println("before advice is executed!");            }              @AfterReturning(pointcut = "aPointcut()", returning="r")            public void afterReturningAdvice(String r) {                      if (r != null)                                 System.out.println("after returning advice is executed! returning String is : " + r);            }              @After("aPointcut()")            public void AfterAdvice() {                      System.out.println("after advice is executed!");            }              @After("aPointcut() && args(str)")            public void AfterAdviceWithArg(String str) {                      System.out.println("after advice with arg is executed!arg is : " + str);            }              @AfterThrowing(pointcut="aPointcut()",throwing="e")            public void afterThrowingAdvice(Exception e) {                      System.out.println("after throwing advice is executed!exception msg is : " + e.getMessage());            }    } |

在基于annotation的Spring AOP中，@Aspect用来标注切面类。@Pointcut标注一个空的方法，用来代表一个pointcut，这个方法必须是public的。@Pointcut注解括号内是pointcut expression，例子中的表达式表示com.psjay.example.spring.aop的所有方法都是join point。而@Before,@After等注解对应着几种不同类型的Advice。被标注的方法就是一个Advice。@Advice注解括号内是一个pointcut。例子中的@afterReturningAdvice()，AfterAdviceWithArg()和afterThrowingAdvice()分别演示了Advice得到join point的返回值，Advice使用join point的参数，Advice使用join point抛出的异常对象几种操作。

**不要忘了在Spring配置文件中配置以上两个类的“Bean”，这里就不贴出具体代码了。**

**测试类**

测试类相对简单，就是从Spring中拿出bean演示AOP的结果。测试类代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class Test {              public static void main(String[] args) {                      ApplicationContext ctx  = new ClassPathXmlApplicationContext("applicationContext.xml");                      UserManager um =  ctx.getBean("userManager",UserManager.class);                      System.out.println("------ Case 1 --------");                      um.addUser("hey");                      System.out.println("------ Case 2 --------");                      try {                                 um.editUser();                      } catch (Exception e) {                        }                      System.out.println("------ Case 3 --------");                      um.getUser();              }  } |

**测试结果：**

—— Case 1 ——–  
before advice is executed!  
addUser(String str) method is executed!  
after advice is executed!  
after advice with arg is executed!arg is : hey  
—— Case 2 ——–  
before advice is executed!  
after advice is executed!  
after throwing advice is executed!exception msg is : something is wrong.  
—— Case 3 ——–  
before advice is executed!  
getUser() method is executed!  
after returning advice is executed! returning String is : Hello  
after advice is executed!

可以看到，Advice已经在对应的join point上起作用了。

**Schema-based（XML）**

除了使用Annotation，我们还可以使用XML来实现Spring AOP。使用XML来实现AOP只是将AOP的配置信息移到XML配置文件里，其他地方与annotation实现AOP并无太大区别。所以这里就不贴出相关代码了。具体配置方法，

 9http://www.blogjava.net/Images/OutliningIndicators/None.gif    <bean id="securityHandler" class="com.bjsxt.spring.SecurityHandler"/>             
10http://www.blogjava.net/Images/OutliningIndicators/None.gif      
11http://www.blogjava.net/Images/OutliningIndicators/None.gif    <bean id="userManager" class="com.bjsxt.spring.UserManagerImpl"/>  
12http://www.blogjava.net/Images/OutliningIndicators/None.gif      
13http://www.blogjava.net/Images/OutliningIndicators/None.gif    <aop:config>  
14http://www.blogjava.net/Images/OutliningIndicators/None.gif        <!-- 定义安全管理的方面 -->  
15http://www.blogjava.net/Images/OutliningIndicators/None.gif        <aop:aspect id="security" ref="securityHandler">  
16http://www.blogjava.net/Images/OutliningIndicators/None.gif            <!-- 定义切入点 -->  
17http://www.blogjava.net/Images/OutliningIndicators/None.gif            <aop:pointcut id="allAddMethod" expression="execution(\* com.bjsxt.spring.UserManagerImpl.add\*(..)))"/>  
18http://www.blogjava.net/Images/OutliningIndicators/None.gif            <!-- 定义通知：before -->  
19http://www.blogjava.net/Images/OutliningIndicators/None.gif            <aop:before method="checkSecurity" pointcut-ref="allAddMethod"/>  
20http://www.blogjava.net/Images/OutliningIndicators/None.gif        </aop:aspect>  
21http://www.blogjava.net/Images/OutliningIndicators/None.gif    </aop:config>      
22http://www.blogjava.net/Images/OutliningIndicators/None.gif</beans>

**Spring AOP API**

在Spring1.2中使用底层的Spring AOP API来实现AOP。当然，Spring3也是完全与其兼容的。我们可以借其窥探一下底层实现。由于比较复杂，我将单独写一篇文章来描述这种实现方式。

**总结**

Spring AOP是基于代理的，是运行时绑定的。合理的运用AOP，将使软件的开发更加便捷，清晰。

对比Spring AOP　annotation实现　和ｘｍｌ实现：ｘｍｌ实现更通用些，可以应用于ｊｄｋ的任何版本

而annotation 只能用于jdk5以上