**https://www.cnblogs.com/dream-to-pku/p/5735293.html**

**[Spring学习进阶 （三） Spring AOP](https://www.cnblogs.com/dream-to-pku/p/5735293.html)**

一、是什么  
AOP是Aspect Oriented Programing的简称，最初被译为“面向方面编程”；  
AOP通过横向抽取机制为无法通过纵向继承体系进行抽象的重复性代码提供了解决方案。  
比如事务的控制我们就可以按照这种方式，但是横向抽取出来之后，如何将这些独立的逻辑融合到业务逻辑中完成和原来一样的业务操作才是关键，这也是AOP解决的主要问题。  
1.相关的术语  
连接点：程序执行的某个特定位置，如类开始初始化前，类初始化后，类某个方法调用前和调用后，方法抛出异常后。一个类或一段程序代码拥有一些具有边界性质的特定点，这些  
特定点就被称为连接点。Spring仅支持方法的连接点，即仅能在方法调用前，方法调用后，方法抛出异常时以及方法调用前后这些程序执行点织入增强。  
 连接点由两个信息确定：一是用方法表示程序的执行点，二是用相对点表示的方位。  
 切点：AOP通过切点定位特定连接点。通俗点说：连接点相当于数据库中的记录，而切点相当于查询条件。切点和连接点不是一对一的关系，一个切点可以匹配多个连接点。  
 增强：织入目标连接点上的一段程序代码。增强接口：BeforeAdvice AfterReturningAdvice ThrowsAdvice等。  
 目标对象：  
 引介：一种特殊的增强，为类添加一些属性和方法  
 织入：将增强添加到目标类具体连接点上的过程  
 三种方式：编译器织入 类装载器织入 动态代理织入  
 代理：  
 切面：由切点和增强组成，既包括了横切逻辑的定义，也包括了连接点的定义。  
二、.创建增强类  
Spring使用增强类定义横切逻辑，同时由于Spring只支持方法连接点，增强还包括了在方法的哪一点加入横切码的方位信息，所以增强既包含横向逻辑，又包含部分连接点的信息。  
1.增强类型  
AOP联盟为增强定义了org.aopalliance.aop.Advice接口，Spring支持5种类型的增强，安装增强在目标类方法的连接点设置  
①：前置增强：org.springframework.aop.BeforeAdvice代表前置增强，因为Spring只支持方法级的增强，所以MethodBeforeAdvice是目前可用的前置增强，表示杂目标方法执行前实施增强而。  
②：后置增强：org.springframework.aop.AfterReturningAdvice代表后增强，表示在目标方法执行后实施增强  
③：环绕增强：org.aopalliance.intercept.MethodInterceptor代表环绕增强，表示在目标方法前后执行增强  
④：异常抛出增强：org.springframework.aop.ThrowsAdvice代表抛出异常增强，表示在目标方法抛出异常后实施增强  
⑤：引介增强：org.springframework.aop.IntroductionInterceptor代表引介增强，表示在目标类中添加一些新的方法和属性  
2.前置增强  
示例：假设服务生只做两件事：欢迎顾客和对顾客服务  
①定义实体类Waiter.java

[复制代码](javascript:void(0);)

1 package com.smart.advice;

2

3 public interface Waiter {

4 void greetTo(String name);

5

6 void serveTo(String name);

7 }

[复制代码](javascript:void(0);)

②：普通的服务情况NaiveWaiter.java

[复制代码](javascript:void(0);)

1 package com.smart.advice;

2

3 public class NaiveWaiter implements Waiter {

4

5 public void greetTo(String name) {

6 System.out.println("greet to " + name + "...");

7 }

8

9 public void serveTo(String name) {

10 System.out.println("serving " + name + "...");

11 }

12 }

[复制代码](javascript:void(0);)

③：假设要在服务之前先友好的打招呼GreetingBeforeAdvice.java

[复制代码](javascript:void(0);)

1 package com.smart.advice;

2

3 import java.lang.reflect.Method;

4

5 import org.springframework.aop.MethodBeforeAdvice;

6

7 public class GreetingBeforeAdvice implements MethodBeforeAdvice {

8 public void before(Method method, Object[] args, Object obj) throws Throwable {

9 String clientName = (String) args[0];

10 System.out.println("How are you！Mr." + clientName + ".");

11 }

12 }

[复制代码](javascript:void(0);)

BeforeAdvice是前置增强的接口，方法前置增强的MethodBeforeAdvice接口是其子类，Spring目前只提供方法调用的前置增强。这正是定义BeforeAdvice接口存在的意义。MethodBeforeAdvice接口仅定义了唯一的方法  
before(Method method, Object[] args, Object obj)  
④：进行测试

[复制代码](javascript:void(0);)

1 package com.smart.advice;

2

3 import org.springframework.aop.BeforeAdvice;

4 import org.springframework.aop.framework.ProxyFactory;

5 import org.testng.annotations.\*;

6

7 public class BeforeAdviceTest {

8 private Waiter target;

9 private BeforeAdvice advice;

10 private ProxyFactory pf;

11

12 @BeforeTest

13 public void init() {

14 target = new NaiveWaiter();

15 advice = new GreetingBeforeAdvice();

16 //①Spring提供的代理工厂

17 pf = new ProxyFactory();

18 // ②设置代理目标

19 pf.setTarget(target);

20 //③为代理目标添加增强

21 pf.addAdvice(advice);

22 }

23

24 @Test

25 public void beforeAdvice() {

26 Waiter proxy = (Waiter) pf.getProxy();

27 proxy.greetTo("John");

28 proxy.serveTo("Tom");

29 }

30 }

[复制代码](javascript:void(0);)

⑤：在Spring中配置

[复制代码](javascript:void(0);)

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

4 xsi:schemaLocation="http://www.springframework.org/schema/beans

5 http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

6

7 <bean id="greetingAdvice" class="com.smart.advice.GreetingBeforeAdvice" />

8 <bean id="target" class="com.smart.advice.NaiveWaiter" />

9 <bean id="waiter"

10 class="org.springframework.aop.framework.ProxyFactoryBean"

11 p:proxyInterfaces="com.smart.advice.Waiter" p:target-ref="target"

12 p:interceptorNames="greetingAdvice"/>

13 </beans>

[复制代码](javascript:void(0);)

代码介绍：target:代理的目标对象  
 ProxyInterfaces:代理需要实现的接口，可以是多个接口，别名interfaces  
 interceptorNames:需要植入目标对象的Bean列表，采用bean的名称指定，这些Bean必须是实现了org.aopalliance.intercept.MethodInterceptor或org.springframeworkwork.aop.Advisor的Bean.配置的顺序对应调用的顺序  
 singleton:返回的代理是否是单实例，默认为单实例  
 optimize:设置为true时，强制使用CGLib代理  
 proxyTargetClass:是否对类进行代理，设置为true时，使用CGLib代理。  
增强测试代码：

[复制代码](javascript:void(0);)

package com.smart.advice;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.testng.annotations.\*;

public class SpringAdviceTest {

@Test

public void testAdvice() {

String configPath = "com/smart/advice/beans.xml";

ApplicationContext ctx = new ClassPathXmlApplicationContext(configPath);

Waiter waiter = (Waiter) ctx.getBean("waiter");

waiter.greetTo("John");

}

}

[复制代码](javascript:void(0);)

3.后置增强  
后置增强在目标类方法调用后执行，假设服务生在每次服务后，也需要使用规范的礼貌用语，则使用后置增强来实现。

[复制代码](javascript:void(0);)

1 package com.smart.advice;

2

3 import java.lang.reflect.Method;

4

5 import org.springframework.aop.AfterReturningAdvice;

6

7 public class GreetingAfterAdvice implements AfterReturningAdvice {

8 //在目标类方法调用后执行

9 public void afterReturning(Object returnObj, Method method, Object[] args,

10 Object obj) throws Throwable {

11 System.out.println("Please enjoy yourself!");

12 }

13 }

[复制代码](javascript:void(0);)

与前置增强类似，通过实现AfterReturningAdvice来定义后置增强的逻辑  
配置：

[复制代码](javascript:void(0);)

1 <bean id="greetingAfter" class="com.smart.advice.GreetingAfterAdvice" />

2 <bean id="target" class="com.smart.advice.NaiveWaiter" />

3 <bean id="waiter"

4 class="org.springframework.aop.framework.ProxyFactoryBean"

5 p:proxyInterfaces="com.smart.advice.Waiter" p:target-ref="target"

6 p:interceptorNames="greetingBefore,greetingAfter"/>

[复制代码](javascript:void(0);)

4.环绕增强  
 环绕增强允许在目标类方法调用前后织入横切逻辑，综合实现了前置，后置增强两者的功能、

[复制代码](javascript:void(0);)

1 package com.smart.advice;

2

3 import org.aopalliance.intercept.MethodInterceptor;

4 import org.aopalliance.intercept.MethodInvocation;

5

6 public class GreetingInterceptor implements MethodInterceptor {

7

8 public Object invoke(MethodInvocation invocation) throws Throwable {

9 Object[] args = invocation.getArguments();//目标方法入参

10 String clientName = (String) args[0];

11 System.out.println("How are you！Mr." + clientName + ".");

12

13 Object obj = invocation.proceed(); //通过反射机制调用目标方法

14

15 System.out.println("Please enjoy yourself!");

16

17 return obj;

18 }

19 }

[复制代码](javascript:void(0);)

使用配置：

1 <bean id="greetingAround" class="com.smart.advice.GreetingInterceptor" />

2 <bean id="target" class="com.smart.advice.NaiveWaiter" />

3 <bean id="waiter" class="org.springframework.aop.framework.ProxyFactoryBean"

4 p:proxyInterfaces="com.smart.advice.Waiter" p:target-ref="target"

5 p:interceptorNames="greetingAround" />

5.异常抛出增强  
 异常抛出增强最适合的应用场景是事务管理，当参与事务的某个Dao发生异常时，事务管理器就必须回滚事务

[复制代码](javascript:void(0);)

1 package com.smart.advice;

2

3 import com.smart.domain.ViewSpace;

4

5 import java.sql.SQLException;

6

7 public class ViewSpaceService {

8

9 public boolean deleteViewSpace(int spaceId) {

10

11 throw new RuntimeException("运行异常。");

12 }

13

14 public void updateViewSpace(ViewSpace viewSpace) throws Exception {

15 // do sth...

16 throw new SQLException("数据更新操作异常。");

17

18 }

19 }

[复制代码](javascript:void(0);)

我们通过TransactionManager这个异常抛出增强对业务方法进行增强处理，同意捕捉抛出的异常并回滚事务

[复制代码](javascript:void(0);)

1 package com.smart.advice;

2

3 import java.lang.reflect.Method;

4

5 import org.springframework.aop.AfterReturningAdvice;

6 import org.springframework.aop.ThrowsAdvice;

7

8

9 public class TransactionManager implements ThrowsAdvice, AfterReturningAdvice {

10 public void afterReturning(Object returnObj, Method method, Object[] args,

11 Object obj) throws Throwable {

12 returnObj = false;

13 System.out.println("Please enjoy yourself!");

14 }

15

16 public void afterThrowing(Method method, Object[] args, Object target,

17 Exception ex) throws Throwable {

18 ex = null;

19 System.out.println("-----------");

20 System.out.println("method:" + method.getName());

21 //System.out.println("抛出异常:" + ex.getMessage());

22 System.out.println("成功回滚事务。");

23 }

24 }

[复制代码](javascript:void(0);)

使用配置：

[复制代码](javascript:void(0);)

1 <bean id="viewSpaceSer

2 viceTarget" class="com.smart.advice.ViewSpaceService" />

3 <bean id="transactionManager" class="com.smart.advice.TransactionManager" />

4 <bean id="viewSpaceService" class="org.springframework.aop.framework.ProxyFactoryBean"

5 p:interceptorNames="transactionManager"

6 p:target-ref="viewSpaceServiceTarget"

7 p:proxyTargetClass="true"/><!--因为ViewSpaceService是类，所以使用CGLib代理-->

[复制代码](javascript:void(0);)

三、创建切面  
在增强中，增强被织入目标类的所有方法中，如果希望有选择地织入目标类某些特定的方法中，就需要使用切点进行，目标连接点的定位。  
增强提供了连接点方位信息：如织入方法前面、后面等。而切点进一步描述织入哪些类的哪些方法上。  
Spring通过org.springframework.aop.Pointcut接口描述切点。PointCut由ClassFilter（定位类）和MethodMatcher（定位方法）构成。  
在ClassFilter只定义了matches(Class clazz),其参数代表一个被检测类，该方法判别被检测的类是否匹配过滤条件。  
Spring支持两种方法匹配器：静态方法匹配器和动态方法匹配器。静态方法匹配器仅对方法名签名进行匹配，而动态方法匹配器会在运行期检查方法入参的值。  
1.切点类型  
静态方法切点：org.springframework.aop.support.StaticMethodMatcerPointcut是静态方法切点的抽象基类，默认情况下它匹配所有的类。StaticMethodMatcherPointcut包括两个主要的子类  
分别是NameMatchMethodPointcut和AbstractRegexpMethodPointcut，前者提供简单字符串匹配方法前面，而后者使用正则表达式匹配方法前面。  
动态方法切点：org.springframework.aop.support.DynamicMethodMatcerPointcut是动态方法切点的抽象基类，默认情况下它匹配所有的类，DynamicMethodMatcerPointcut已经过时，  
可以使用DefaultPointcutAdvisor和DynamicMethodMatcherPointcut动态方法代替。  
注解切点：org.springframework.aop.support.annotation.AnnotationMatchingPointcut实现类表示注解切点。  
表达式切点：org.springframework.aop.support.ExpressionPointcut接口主要是为了支持AspectJ切点表达式语法而定义的接口  
流程切点：org.springframework.aop.support.ControlFlowPointcut实现类表示控制流程切点。ControlFlowPointcut是一种特殊的切点，它根据程序执行堆栈的信息查看目标方法是否由某一个方法直接或间接发起调用，以此判断是否为匹配的连接点  
复合切点：org.springframework.aop.support.ComposablePointcut实现类是为了创建多个切点而提供的方便操作类，它所有的方法都返回ComposablePointcut类，这样就可以使用链接表达式对切点进行操作  
2.切面类型  
一般切面：Advisor代表一般切面，仅包含一个Advice，他本身就是一个简单的切面，只不过他代表的横切连接点是所有目标类的所有方法。  
切点切面：PointcutAdvisor具有切点的切面，包含Advice和Pointcut两个类，这样就可以通过类，方法名以及方法方位等信息灵活地定义切面的连接点  
引介切面：IntroductionAdvisor引介切面，引介切面是对应引介增强的特殊切面，它应用于类层面上，使用ClassFilter进行定义。  
3.静态普通方法名匹配切面  
StaticMethodMatcherPointcutAdvisor代表一个静态方法匹配切面，它通过StaticMethodMatcherPointcut定义切点，通过类过滤和方法名匹配定义切点。  
示例Waiter.java

[复制代码](javascript:void(0);)

1 package com.smart.advisor;

2

3 public class Waiter {

4

5 public void greetTo(String name) {

6 System.out.println("waiter greet to " + name + "...");

7 }

8

9 public void serveTo(String name) {

10 System.out.println("waiter serving " + name + "...");

11 }

12 }

[复制代码](javascript:void(0);)

Seller.java

[复制代码](javascript:void(0);)

1 package com.smart.advisor;

2

3 public class Seller {

4 public void greetTo(String name) {

5 System.out.println("seller greet to " + name + "...");

6 }

7 }

[复制代码](javascript:void(0);)

Waiter有两个方法，分别是greetTo和serveTo，Seller拥有一个和Waiter相同名称的方法greetTo()。现在我们在Waiter#greetTo()方法调用前织入一个增强，即连接点为Waiter#greetTo()方法调用前的位置

[复制代码](javascript:void(0);)

1 package com.smart.advisor;

2

3 import java.lang.reflect.Method;

4

5 import org.springframework.aop.ClassFilter;

6 import org.springframework.aop.support.StaticMethodMatcherPointcutAdvisor;

7

8 public class GreetingAdvisor extends StaticMethodMatcherPointcutAdvisor {

9

10 public boolean matches(Method method, Class clazz) {

11 return "greetTo".equals(method.getName());//切点方法名匹配为greetTo

12 }

13

14 public ClassFilter getClassFilter() {//切点类匹配规则：为Waiter的类或子类

15 return new ClassFilter() {

16 public boolean matches(Class clazz) {

17 return Waiter.class.isAssignableFrom(clazz);

18 }

19 };

20 }

21 }

[复制代码](javascript:void(0);)

因为StaticMethodMatcherPointcutAdvisor抽象类唯一需要定义的是matches()方法。在默认情况下，该切面匹配所有的类，这里通过覆盖getClassFilter()方法，让他仅匹配Waiter类及其子类  
定义前置增强：

[复制代码](javascript:void(0);)

1 package com.smart.advisor;

2

3 import java.lang.reflect.Method;

4

5 import org.springframework.aop.MethodBeforeAdvice;

6

7 public class GreetingBeforeAdvice implements MethodBeforeAdvice {

8

9 public void before(Method method, Object[] args, Object obj) throws Throwable {

10 String clientName = (String) args[0];

11 System.out.println(obj.getClass().getName() + "." + method.getName());

12 System.out.println("How are you！Mr." + clientName + ".");

13 }

14 }

[复制代码](javascript:void(0);)

配置切面：静态方法匹配切面

[复制代码](javascript:void(0);)

1 <bean id="waiterTarget" class="com.smart.advisor.Waiter" />

2 <bean id="sellerTarget" class="com.smart.advisor.Seller" />

3 <bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />

4 <bean id="greetingAdvisor" class="com.smart.advisor.GreetingAdvisor"

5 p:advice-ref="greetingAdvice" /><!--向切面注入一个前置增强-->

6 <bean id="parent" abstract="true"<!--通过一个父<bean>定义公共的配置信息-->

7 class="org.springframework.aop.framework.ProxyFactoryBean"

8 p:interceptorNames="greetingAdvisor" p:proxyTargetClass="true" />

9 <bean id="waiter" parent="parent" p:target-ref="waiterTarget" /><!--waiter代理-->

10 <bean id="seller" parent="parent" p:target-ref="sellerTarget" /><!--seller代理-->

[复制代码](javascript:void(0);)

4.静态正则表达式匹配切面  
 在上面的配置中只能通过方法名定义切点，这种方式不够灵活。RegexpMethodPointcutAdvisor是正则表达式方法匹配的切面实现类。  
 使用正则表达式定义切面：

[复制代码](javascript:void(0);)

1 <!-- 正则表达式方法名匹配切面 -->

2 <bean id="regexpAdvisor"

3 class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"

4 p:advice-ref="greetingAdvice">

5 <property name="patterns"><!--用正则表达式定义目标类全限定方法名的匹配模式串-->

6 <list>

7 <value>.\*greet.\*</value><!--匹配模式串-->

8 </list>

9 </property>

10 </bean>

11 <bean id="waiter1" class="org.springframework.aop.framework.ProxyFactoryBean"

12 p:interceptorNames="regexpAdvisor" p:target-ref="waiterTarget"

13 p:proxyTargetClass="true" />

[复制代码](javascript:void(0);)

四、自动创建代理  
 在之前的示例中，都是通过ProxtFactoryBean创建织入切面的代理，每一个需要被代理的Bean都需要使用一个ProxyFactoryBean进行配置。  
 Spring提供了自动代理机制，让容器自动生成代理。  
 1.实现类介绍  
 这些机遇BeanPostProcessor的自动代理创建器的实现类，将根据一些规则自动在容器实例化Bean时为匹配的Bean生成代理实例可以分为3类。  
 ①：基于Bean配置名规则的自动代理创建器：允许为一组特定配置名的Bean自动创建代理实例的创建器，实现类为BeanNameAutoProxyCreator  
 ②：基于Advisor匹配机制的自动代理创建器：会对容器中所有的Advisor进行扫描，自动将这些切面应用到匹配的Bean中，实现类为DefaultAdvisorAutoProxyCreator  
 ③：基于Bean中AspectJ注解标签的自动代理创建器：为包含AspectJ注解的Bean自动创建代理实例，她的实现类是AnnotationAwareAspectJAutoProxyCreator.  
 2.BeanNameAutoProxyCreator

[复制代码](javascript:void(0);)

1 <bean id="waiter" class="com.smart.advisor.Waiter" />

2 <bean id="seller" class="com.smart.advisor.Seller" />

3 <bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />

4 <!-- 通过Bean名称自动创建代理 -->

5 <bean

6 class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator"

7 p:beanNames="\*er" p:interceptorNames="greetingAdvice"

8 p:optimize="true"/>

[复制代码](javascript:void(0);)

3.DefaultAdvisorAutoProxyCreator

[复制代码](javascript:void(0);)

1 <bean id="waiter" class="com.smart.advisor.Waiter" />

2 <bean id="seller" class="com.smart.advisor.Seller" />

3 <bean id="greetingAdvice" class="com.smart.advisor.GreetingBeforeAdvice" />

4 <!--通过Advisor自动创建代理-->

5 <bean id="regexpAdvisor"

6 class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"

7 p:patterns=".\*greet.\*" p:advice-ref="greetingAdvice" />

8 <bean

9 class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" />

[复制代码](javascript:void(0);)

五、基于@AspectJ配置切面  
之前是使用Pointcut和Advice接口描述切点和增强，并用Advisor整合两者描述切面，@AspectJ则采用注解来描述切点，增强，两者只是表达方式不同。  
1、语法基础  
①方法切点函数：通过描述目标类方法信息定义连接点  
②：方法入参切点函数：通过描述目标类方法入参的信息定义连接点  
③：目标类切点函数：通过描述目标类型信息定义连接点  
④：代理类切点函数：通过描述目标类的代理类的信息定义连接点  
2、使用准备  
在使用之前，要将Spring中的asm模块添加到类路径中。  
3、编程式织入示例：

[复制代码](javascript:void(0);)

1 package com.smart;

2

3 import com.smart.anno.Monitorable;

4 import com.smart.anno.NeedTest;

5

6 @Monitorable

7 public class NaiveWaiter implements Waiter {

8 public void greetTo(String clientName) {

9 System.out.println("NaiveWaiter:greet to "+clientName+"...");

10 }

11 @NeedTest

12 public void serveTo(String clientName){

13 System.out.println("NaiveWaiter:serving "+clientName+"...");

14 }

15 public void smile(String clientName,int times){

16 System.out.println("NaiveWaiter:smile to "+clientName+ times+"times...");

17 }

18 }

[复制代码](javascript:void(0);)

PreGreetingAspect.java

[复制代码](javascript:void(0);)

1 package com.smart.aspectj.example;

2

3 import org.aspectj.lang.annotation.Aspect;

4 import org.aspectj.lang.annotation.Before;

5 @Aspect//通过该注解将PreGreetingAspect标识为一个切面

6 public class PreGreetingAspect{

7 @Before("execution(\* greetTo(..))")//定义切点和增强类型

8 public void beforeGreeting(){//增强的横切逻辑

9 System.out.println("How are you");

10 }

11 }

[复制代码](javascript:void(0);)

注释：在类定义处标记了@AspectJ注解，第三方处理程序就可以通过类是否拥有@AspectJ注解判断是否为一个切面  
在beforeGreeting()方法处标记了@Before注解，并为该注解提供了成员值exection(\*greetTo(..))、@Before注解表示该增强是一个前置增强，而成员值是一个@AspectJ切点表达式，意思是在目标类greetTo()方法织入增强，greetTo()方法可以带任意的入参和任意的返回值  
beforeGreeting()是增强所使用的横切逻辑，该横切逻辑在目标方法前调用  
PreGreetingAspect类通过注解和代码，将切点，增强类型和增强的横切逻辑整合到一个类中，时切面的定义浑然天成。PreGreetingAspect一个类就相当于之前的BeforeAdvice、N俺么MatchMethodPointcut以及DefaultPointcutAdvisor三者联合表达的信息  
4、通过配置使用@AspectJ切面  
① 普通方式

1 <bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"/>

2 <bean id="waiter" class="com.smart.NaiveWaiter" />

3 <bean class="com.smart.aspectj.example.PreGreetingAspect" />

② 使用Schema的aop命名空间

1 <aop:aspectj-autoproxy/>

2 <bean id="waiter" class="com.smart.NaiveWaiter" />

3 <bean class="com.smart.aspectj.example.PreGreetingAspect" />

5、不同增强类型  
 @Before表示前置增强，相当于BeforeAdvice的功能  
 value:该成员用于定义切点  
 argName:由于无法通过Java反射机制获取方法入参名，所以如果在java编译时未启用调试信息或者需要在运行期解析切点，就必须通过这个成员指定注解所标注增强方法的参数名，，多个参数名用逗号分隔  
 @AfterReturning表示后置增强 相当于AfterReturningAdvice  
 value:该成员用于定义切点  
 pointcut:表示切点的信息，如果显示指定pointcut值，他将覆盖value的设置值，可以讲pointcut成员看成是value的同义词  
 returning:将目标对象方法的返回值绑定给增强的方法  
 argName:同上  
 @Around:表示环绕增强，相当于MethodInterceptor  
 value：定义切点  
 argNames:同上  
 @AfterThrowing:表示抛出增强相当于ThrowsAdvice  
 value:定义切点  
 pointcut:表示切点的信息，如果显示指定pointcut值，他将覆盖value的设置值，可以将pointcut成员看成是value的同义词  
 throwing:将抛出的异常绑定到增强方法中  
 argNames:同上  
 @After 表示final增强，都会执行  
 value定义切点  
 argNames同上  
 @DeclareParents  
 表示引介增强，相当于IntroductionInterceptor  
 value:该成员用于定义切点，他表示在哪个目标类上添加引介增强  
 defaultImpl：默认的接口实现类  
总结：AOP是OOP的有益补充，它为程序开发提供了一个崭新的思考角度，可以将任意性的横切逻辑抽取到同意的模块中，只有通过OOP的纵向抽象和AOP的横向抽取，程序才可以真正解决重复性代码的问题  
 Spring采用JDK动态代理和CGLib动态代理的技术在运行期织入增强，左移用户不需要装备特殊的编译器或类装载器就可以使用AOP的功能，要使用JDK动态代理，目标类必须事先接口，而CGLib不对类做任何限制，通过动态生成目标子类的方式提供代理。  
 JDK在创建代理对象时性能高于CGLib,而生成的代理对象的运行性能却比CGLib低，如果是singleton的代理，推荐使用CGLib动态代理  
 Spring只能在方法级别上织入增强，Spring提供了4中类型的方法增强，分别是前置增强后置增强环绕增强和异常抛出增强此外还有引介增强，引介增强是类级别的  
 它为目标类织入新的接口实现。广义上来看，增强其实就是一种最简单的切面，他既包括横切代码也包括切点信息，只不过他的切点只是简单地方法相对位置的信息，所以增强一般需要和切点联合才可以表示一个更具有实用性的切面

分类: [Spring](http://www.cnblogs.com/dream-to-pku/category/850772.html)

标签: [Spring AOP](http://www.cnblogs.com/dream-to-pku/tag/Spring%20AOP/)