https://www.jianshu.com/p/1dd6a26c881b

**spring aop 细说advice,advisor**

[](https://www.jianshu.com/u/d3f1e12064a5)

[不迷失](https://www.jianshu.com/u/d3f1e12064a5) 关注

2016.05.18 15:02\* 字数 2188 阅读 7529评论 2喜欢 5

Spring支持五种类型的增强或通知(Advice):  
Before(方法执行前) org.apringframework.aop.MethodBeforeAdvice  
AfterReturning(方法返回后) org.springframework.aop.AfterReturningAdvice  
After-throwing(异常抛出后) org.springframework.aop.ThrowsAdviceArround  
环绕，即方法前后 org.aopaliance.intercept.MethodInterceptor  
引介，不常用 org.springframework.aop.IntroductionInterceptor

示例：

package spring4;

/\*\*

\* Created by weixin:javajidi\_com.

\* 业务逻辑接口

\*/

public interface Service {

void print(String str);

}

package spring4;

/\*\*

\* Created by weixin:javajidi\_com.

\*/

public class ServiceImpl implements Service {

public void print(String str) {

System.out.println("我是业务方法"+str);

}

}

package spring4.aop;

import org.springframework.aop.AfterReturningAdvice;

import java.lang.reflect.Method;

/\*\*

\* Created by weixin:javajidi\_com.

\*/

public class AfterAdvice implements AfterReturningAdvice {

public void afterReturning(Object o, Method method, Object[] objects, Object o1) throws Throwable {

System.out.println("AfterAdvice方法执行完成了");

System.out.println(method.getName()+";"+o1.getClass());

}

}

package spring4.aop;

import org.springframework.aop.MethodBeforeAdvice;

import java.lang.reflect.Method;

/\*\*

\* Created by weixin:javajidi\_com.

\* 方法执行前的逻辑，称前置通知

\*/

public class BeforeAdvice implements MethodBeforeAdvice{

public void before(Method method, Object[] objects, Object o) throws Throwable {

System.out.println("BeforeAdvice方法执行前");

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

}

}

package spring4.aop;

import org.aopalliance.intercept.MethodInterceptor;

import org.aopalliance.intercept.MethodInvocation;

/\*\*

\* Created by weixin:javajidi\_com.

\*/

public class RoundAdvice implements MethodInterceptor {

public Object invoke(MethodInvocation methodInvocation) throws Throwable {

System.out.println("Roundadvice方法执行前");

System.out.println(methodInvocation.getArguments()[0]);//可以获取目标方法的参数值

Object result=methodInvocation.proceed();//调用目标对象的方法

System.out.println("RoundAdvice方法执行完成了");

return result;

}

}

package spring4.aop;

import org.springframework.aop.framework.ProxyFactory;

import spring4.Service;

import spring4.ServiceImpl;

/\*\*

\* Created by weixin:javajidi\_com.

\*/

public class Test {

public static void main(String[] arg){

Service service=new ServiceImpl();//

//使用代理工厂为目标对象创建代理，并织入我们自己的advice逻辑

ProxyFactory proxyFactoryBean=new ProxyFactory();

proxyFactoryBean.setTarget(service);//设置目标对象

proxyFactoryBean.addAdvice(new BeforeAdvice());//为目标对象织入增强

proxyFactoryBean.addAdvice(new AfterAdvice());

proxyFactoryBean.addAdvice(new RoundAdvice());

Service proxy=(Service)proxyFactoryBean.getProxy();

proxy.print("test");

}

}

运行结果：

BeforeAdvice方法执行前

print;class spring4.ServiceImpl

Roundadvice方法执行前

test

我是业务方法test

RoundAdvice方法执行完成了

AfterAdvice方法执行完成了

print;class spring4.ServiceImpl

如果通过xml配置，我们可以使用ProxyFactoryBean

<?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"

xsi:schemaLocation="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.xsd">

<bean id="beforeAdvice" class="spring4.aop.BeforeAdvice"/>

<bean id="afterAdvice" class="spring4.aop.AfterAdvice"/>

<bean id="roundAdvice" class="spring4.aop.RoundAdvice"/>

<bean id="service" class="spring4.ServiceImpl"/>

<bean id="serviceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="interceptorNames">

<list>

<value>beforeAdvice</value>

<value>afterAdvice</value>

<value>roundAdvice</value>

</list>

</property>

<property name="target" ref="service"></property>

</bean>

</beans>

package spring4.aop;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import spring4.Service;

/\*\*

\* Created by weixin:javajidi\_com.

\*/

public class Test {

public static void main(String[] arg){

// Service service=new ServiceImpl();

// //使用代理工厂为目标对象创建代理，并织入我们自己的advice逻辑

// ProxyFactory proxyFactoryBean=new ProxyFactory();

// proxyFactoryBean.setTarget(service);//设置目标对象

// proxyFactoryBean.addAdvice(new BeforeAdvice());//为目标对象织入增强

// proxyFactoryBean.addAdvice(new AfterAdvice());

// proxyFactoryBean.addAdvice(new RoundAdvice());

// Service proxy=(Service)proxyFactoryBean.getProxy();

// proxy.print("test");

ApplicationContext applicationContext=new ClassPathXmlApplicationContext("classpath:spring.xml");

Service service=applicationContext.getBean("serviceProxy",Service.class);

service.print("test");

}

}

在前面介绍各类增强时，大家可能没有注意到一个问题：增强被织入到目标类的所有方法中。

假设我们希望有选择地织入到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了。描述连接点是进行AOP编程最主要的一项工作。增强提供了连接点方位信息：如织入到方法前面、后面等，而切点进一步描述织入到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和MethodMatcher构成。它通过ClassFilter定位到某些特定类上，通过MethodMatcher定位到某些特定方法上，这样Pointcut就拥有了描述某些类的某些特定方法的能力。

Spring提供了六种类型切点：

·静态方法切点：org.springframework.aop.support.StaticMethodMatcherPointcut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签名，而后者使用正则表达式匹配方法签名；

·动态方法切点：org.springframework.aop.support.DynamicMethodMatcherPointcut是动态方法切点的抽象基类，默认情况下它匹配所有的类。 DynamicMethodMatcherPointcut类已经过时，可以使用DefaultPointcutAdvisor和DynamicMethodMatcherPointcut动态方法匹配器替代之；

·注解切点：org.springframework.aop.support.annotation.AnnotationMatchingPointcut实现类表示注解切点。使用AnnotationMatchingPointcut支持在Bean中直接通过JDK5.0注解标签定义的切点；

·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，我们就可以使用链接表达式对其进行操作，形如：  
Pointcut pc=new ComposablePointcut().union(classFilter).intersection(methodMatcher).intersection(pointcut)

前面已经解释过，切面就是由切点和增强组成。由于增强既包含横切代码，又包含部分的连接点信息（方法前、方法后等的方位信息），所以我们可以仅通过增强类生成一个切面。但切点仅代表目标类连接点的部分信息（类和方法的定位），所以仅有切点，我们无法制作出一个切面，必须结合增强才能制作出切面。

Spring使用org.springframework.aop.Advisor接口表示切面的概念，一个切面同时包含横切代码和连接点信息。

·Advisor:代表一般切面，它仅包含一个Advice，我们说过，因为Advice包含了横切代码和连接点的信息，所以Advice本身就是一个简单的切面，只不过它代表的横切的连接点是所有目标类的所有方法，因为这个横切面太宽泛，所以一般不会直接使用；

·PointcutAdvisor:代表具有切点的切面，它包含Advice和Pointcut两个类，这样我们就可以通过类、方法名以及方法方位等信息灵活地定义切面的连接点，提供更具适用性的切面。

PointcutAdvisor主要有6个具体的实现类，分别介绍如下：

·DefaultPointcutAdvisor:最常用的切面类型，它可以通过任意Pointcut和Advice定义一个切面，唯一不支持的是引介的切面类型，一般可以通过扩展该类实现自定义的切面；

·NameMatchMethodPointcutAdvisor:通过该类可以定义按方法名定义切点的切面：

·RegexpMethodPointcutAdvisor:对于按正则表达式匹配方法名进行切点定义的切面，可以通过扩展该实现类进行操作。 RegexpMethodPointcutAdvisor允许用户以正则表达式模式串定义方法匹配的切点，其内部通过JdkRegexpMethodPointcut构造出正则表达式方法名切点。

·StaticMethodMatcherPointcutAdvisor：静态方法匹配器切点定义的切面，默认情况下，匹配所有的目标类；

·AspecJExpressionPointcutAdvisor:用于Aspecj切点表达式定义切点的切面。

·AspecJPointcutAdvisor:用于AspecJ语法定义切点的切面。

这些Advisor的实现类，都可以在Pointcut中找到对应物，实际上，它们都是通过扩展对应Pointcut实现类并实现PointcutAdvisor接口进行定义。此外，Advisor都实现了org.springframework.core.Ordered接口，Spring将根据Advisor定义的顺序决定织入切面的顺序。

我们通过使用RegexpMethodPointcutAdvisor来理解如何使用。

RegexpMethodPointcutAdvisor表示通过正则表达式进行切点描述的切面，它有一个pattern属性用来指定增强要应用到哪些类的哪些方法，也可以通过patterns属性指定多个表达式进行匹配。有一个advice属性用来表示要应用的增强，这样就能表示一个完整的切面了。  
我们举几个例子用进一步认识正则表达式在配置匹配方法上的具体应用：  
示例1：.\*set.\*表示所有类中的以set前缀的方法，如com.baobaotao.Waiter.setSalary()，  
Person.setName()等；  
示例2：com.advisor.\*表示com.advisor包下所有类的所有方法；  
示例3：com.service.\*Service.\* 牢匹配com.service包下所有类名以Service结尾的类的所有方法，如com.service.UserService.save(User user)、com.service.ForumService.update(Forum forum)等方法；  
示例4：com.service.\*.save.+ 匹配com.service包中所有类中所以save为前缀的方法。如匹配com.service.UserService类的saveUser()和saveLoginLog()方法，但不匹配该类中的save()方法。

<?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"

xsi:schemaLocation="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.xsd">

<bean id="beforeAdvice" class="spring4.aop.BeforeAdvice"/>

<bean id="afterAdvice" class="spring4.aop.AfterAdvice"/>

<bean id="roundAdvice" class="spring4.aop.RoundAdvice"/>

<bean id="service" class="spring4.ServiceImpl"/>

<bean id="advisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<property name="advice" ref="beforeAdvice"/>

<!--正则表达式用来表示增加添加到哪些类的哪些方法-->

<property name="pattern" value="spring4\..\*.print.\*"/><!--表示应用到spring4包下所有类中的所有print开头的方法上-->

</bean>

<bean id="serviceProxy" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="interceptorNames">

<list>

<value>advisor</value><!--这里可以是advice,也可以是advisor-->

</list>

</property>

<property name="target" ref="service"></property>

</bean>

</beans>

在原service加多一个方法便于观察

package spring4.aop;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import spring4.Service;

/\*\*

\* Created by weixin:javajidi\_com.

\*/

public class Test {

public static void main(String[] arg){

// Service service=new ServiceImpl();

// //使用代理工厂为目标对象创建代理，并织入我们自己的advice逻辑

// ProxyFactory proxyFactoryBean=new ProxyFactory();

// proxyFactoryBean.setTarget(service);//设置目标对象

// proxyFactoryBean.addAdvice(new BeforeAdvice());//为目标对象织入增强

// proxyFactoryBean.addAdvice(new AfterAdvice());

// proxyFactoryBean.addAdvice(new RoundAdvice());

// Service proxy=(Service)proxyFactoryBean.getProxy();

// proxy.print("test");

ApplicationContext applicationContext=new ClassPathXmlApplicationContext("classpath:spring.xml");

Service service=applicationContext.getBean("serviceProxy",Service.class);

service.print("print");

service.say("say");

}

}

运行结果

BeforeAdvice方法执行前

print;class spring4.ServiceImpl

我是业务方法print

say:say

可见只有print方法被加入了增强

在前面的例子中，我们都通过ProxyFactoryBean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。对由拥有众多需要代理Bean的系统，这将非常麻烦。

幸运的是，Spring为我们提供了自动代理机制，让容器为我们自动生成代理，把我们从烦琐的配置工作中解放出来。在内部，Spring使用BeanPostProcessor自动地完成这项工作。这些基于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。这些代理创建器可以分为以下三类：

.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代理创建器，实现类为BeanNameAutoProxyCreator;

.基于Advisor匹配机制的自动代理创建器：它会对容器中所有的Advisor进行扫描，自动将这些切面应用到匹配的Bean中（即为目标Bean创建代理实例），实现类为DefaultAdvisorAutoProxyCreator;

.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，它的实现类是AnnotationAwareAspectjAutoProxyCreator。

所有的自动代理创建器类都实现了BeanPostProcessor，在容器实例化Bean时，BeanPostProcessor将对它进行加工处理，所以，自动代理创建器有机会对满足匹配规则的Bcan自动创建代理对象。  
使用时，我们只需在容器中注册相应bean即可生效。

<?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"

xsi:schemaLocation="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.xsd">

<bean id="beforeAdvice" class="spring4.aop.BeforeAdvice"/>

<bean id="afterAdvice" class="spring4.aop.AfterAdvice"/>

<bean id="roundAdvice" class="spring4.aop.RoundAdvice"/>

<bean id="service" class="spring4.ServiceImpl"/>

<bean id="advisor" class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<property name="advice" ref="beforeAdvice"/>

<!--正则表达式用来表示增加添加到哪些类的哪些方法-->

<property name="pattern" value="spring4\..\*.print.\*"/><!--表示应用到spring4包下所有类中的所有print开头的方法上-->

</bean>

<!--会扫描所有容器中的advisor,然后自动为这些advisor要应用的bean生成代理-->

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

</beans>

package spring4.aop;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import spring4.Service;

/\*\*

\* Created by weixin:javajidi\_com.

\*/

public class Test {

public static void main(String[] arg){

// Service service=new ServiceImpl();

// //使用代理工厂为目标对象创建代理，并织入我们自己的advice逻辑

// ProxyFactory proxyFactoryBean=new ProxyFactory();

// proxyFactoryBean.setTarget(service);//设置目标对象

// proxyFactoryBean.addAdvice(new BeforeAdvice());//为目标对象织入增强

// proxyFactoryBean.addAdvice(new AfterAdvice());

// proxyFactoryBean.addAdvice(new RoundAdvice());

// Service proxy=(Service)proxyFactoryBean.getProxy();

// proxy.print("test");

ApplicationContext applicationContext=new ClassPathXmlApplicationContext("classpath:spring.xml");

Service service=applicationContext.getBean("service",Service.class);

service.print("print");

service.say("say");

}

}

运行结果一致。