Spring中

## 1 ：AOP代理模式

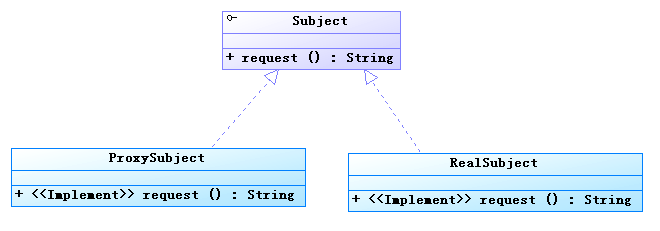
**代理模式:**代理模式的英文叫做Proxy或Surrogate，中文都可译为”代理“，所谓代理，就是一个人或者一个机构代表另一个人或者另一个机构采取行动。在一些情况下，一个客户不想或者不能够直接引用一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用

**抽象主题角色**：**声明了真实主题和代理主题的共同接口**，这样一来在任何可以使用真实主题的地方都可以是使用代理主题

**代理主题（Proxy）角色**：代理主题角色内部含有对真实主题的引用，从而可以在任何时候操作真实主题对象；代理主题角色提供一个与真实主题角色相同的接口，以便可以在任何时候都可以替代真实主题控制对真实主题的引用，负责在需要的时候创建真实主题对象（和删除真实主题对象）；代理角色通常在将客户端调用传递给真实的主题之前或之后，都要执行某个操作，而不是单纯地将调用传递给真实主题对象。

**真实主题角色**：定义了代理角色所代表地真实对象

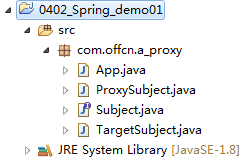




### 测试代码

**0402\_Spring\_demo01**

结构：



* Subject接口

**public** **interface** Subject {

**public** String request();

}

* TargetSubject类，目标对象

**public** **class** TargetSubject **implements** Subject {

**public** String request() {

**return** "该房屋出租3000元";

}

}

* ProxySubject类，代理对象

**public** **class** ProxySubject **implements** Subject {

//有代理来访问目标对象

**public** String request() {

Subject subject = **new** TargetSubject();

//修改对目标对象的某些业务逻辑

String str = subject.request();

**return** str.replace("3000", "4000");

}

}

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

//直接访问目标对象

// Subject subject = new TargetSubject();

// String str = subject.request();

// System.out.println("str:"+str);

//访问代理对象

Subject subject = **new** ProxySubject();

String str = subject.request();

System.*out*.println("str:"+str);

}

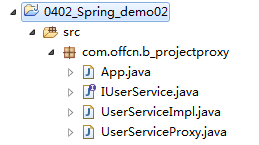
}

## 2：代理模式实际应用实例

### （1）测试代码

**0402\_Spring\_demo02**

结构：



* IuserService接口

**public** **interface** IUserService {

**public** **void** saveUser(String name,String password);

**public** **void** updateUser(String name,String password);

**public** **void** deleteUser(String name);

**public** String findUser();

}

* UserServiceImpl类

**public** **class** UserServiceImpl **implements** IUserService {

**public** **void** saveUser(String name, String password) {

System.*out*.println("【新增】用户名："+name+"，密码："+password);

}

**public** **void** updateUser(String name, String password) {

System.*out*.println("【修改】用户名："+name+"，密码："+password);

}

**public** **void** deleteUser(String name) {

System.*out*.println("【删除】用户名："+name);

}

**public** String findUser() {

System.*out*.println("【查询】用户");

**return** "小强";

}

}

* UserServiceProxy代理类

**public** **class** UserServiceProxy **implements** IUserService {

IUserService userService = **new** UserServiceImpl();

**public** **void** saveUser(String name, String password) {

security();

userService.saveUser(name, password);

}

**public** **void** updateUser(String name, String password) {

security();

userService.updateUser(name, password);

}

**public** **void** deleteUser(String name) {

userService.deleteUser(name);

}

**public** String findUser() {

**return** userService.findUser();

}

//安全校验的方法

**public** **void** security(){

System.*out*.println("开始进行安全校验！！！");

}

}

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

//直接访问目标对象

// IUserService userService = new UserServiceImpl();

// userService.saveUser("超级强", "123");

// userService.updateUser("超级强", "123");

// userService.deleteUser("超级强");

// String str = userService.findUser();

// System.out.println("str:"+str);

//使用代理对象访问目标对象（优势：目标对象专注于实现目标对象需要的业务操作，代理对象用来添加其他的操作）

IUserService userService = **new** UserServiceProxy();

userService.saveUser("超级强", "123");

userService.updateUser("超级强", "123");

userService.deleteUser("超级强");

String str = userService.findUser();

System.*out*.println("str:"+str);

}

}

## 3、AOP的底层实现

### 1：JDK动态代理

**import** java.lang.reflect.InvocationHandler;

**import** java.lang.reflect.Method;

**import** java.lang.reflect.Proxy;

**public** **class** JDKProxy **implements** InvocationHandler{

**private** Object targerObject;//代理的目标对象

/\*\*

\* 创建代理对象

\* **@param** targerObject 目标对象

\* **@return**

\*/

**public** Object createProxyInstance(Object targerObject){

**this**.targerObject=targerObject;

/\*

\* 第一个参数设置代码使用的类加载器,一般采用跟目标类相同的类加载器

\* 第二个参数设置代理类实现的接口,跟目标类使用相同的接口

\* 第三个参数设置回调对象,当代理对象的方法被调用时,会调用该参数指定对象的invoke方法

\*/

**return** **Proxy.*newProxyInstance*(**

**targerObject.getClass().getClassLoader(),**

**targerObject.getClass().getInterfaces(),**

**this);**

}

/\*\*

\* **@param** proxy 目标对象的代理类实例

\* **@param** method 对应于在代理实例上调用接口方法的Method实例

\* **@param** args 传入到代理实例上方法参数值的对象数组

\* **@return** 方法的返回值 没有返回值时是null

\*/

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

System.*out*.println("代理类实例 "+proxy.getClass());

System.*out*.println("方法名称 "+method.getName());

**if**(args!=**null**&&args.length>0){//方法的参数值

**for**(**int** i=0;i<args.length;i++){

System.*out*.println("方法参数值 "+args[i].toString()); }

}

Object returnvalue=**null**; //定义方法的返回值 没有返回值时是null

**returnvalue=method.invoke(this.targerObject, args); //调用目标对象的方法**

System.*out*.println("方法的返回值 "+returnvalue);

**return** returnvalue;

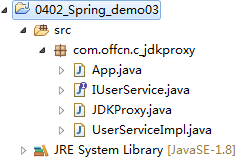
}

}

#### （1）测试代码

0402\_Spring\_demo03

结构：



* IUserService接口

**public** **interface** IUserService {

**public** **void** saveUser(String name,String password);

**public** **void** updateUser(String name,String password);

**public** **void** deleteUser(String name);

**public** String findUser();

}

* UserServiceImpl类

/\*\*Target(目标对象)：需要访问的真实主题就是目标对象，本例：UserServiceImpl类\*/

**public** **class** UserServiceImpl **implements** IUserService {

/\*\*joinpoint(连接点)：目标对象中定义的所有方法，该方法就是连接点，本例：saveUser(),updateUser(),deleteUser(),findUser()\*/

**public** **void** saveUser(String name, String password) {

System.*out*.println("【新增】用户名："+name+"，密码："+password);

}

**public** **void** updateUser(String name, String password) {

System.*out*.println("【修改】用户名："+name+"，密码："+password);

}

**public** **void** deleteUser(String name) {

System.*out*.println("【删除】用户名："+name);

}

**public** String findUser() {

System.*out*.println("【查询】用户");

**return** "小强";

}

}

* JDKProxy代理类（JDK代理模式）

//使用jdk代理的方式，创建代理对象

/\*\*Proxy（代理对象）：在访问目标对象方法之前或者之后，先要执行代理对象的方法，本例：JDKProxy类\*/

/\*\*Aspect(切面): 通知是一个方法，需要放置在类中，该类就是切面，本例：JDKProxy\*/

**public** **class** JDKProxy **implements InvocationHandler** {

//目标对象

**private** Object targetObject;

//使用目标对象创建一个代理对象

/\*\*Weaving(织入):是指把切面应用到目标对象来创建新的代理对象的过程.切面在指定的连接点织入到目标对象\*/

**public** Object createProxyWithTarget(Object targetObject) {

**this**.targetObject = targetObject;

//使用目标对象创建代理对象

/\*

\* 第一个参数设置代码使用的类加载器,一般采用跟目标类相同的类加载器

\* 第二个参数设置代理类实现的接口,跟目标类使用相同的接口

\* 第三个参数设置回调对象,当代理对象的方法被调用时,会调用该参数指定对象的invoke方法

\*/

**return** Proxy.*newProxyInstance*(

**this**.targetObject.getClass().getClassLoader(),

**this**.targetObject.getClass().getInterfaces(),

**this**);

}

//访问目标对象，要先执行invok的方法

/\*

\* 参数：

\* Object proxy：代理对象

\* Method method：通过Method对象可以访问到目标对象中的方法

\* Object[] args：传递给目标对象方法的参数

\* 返回值：

\* Object，表示目标对象方法的返回值

\*/

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

/\*\*Pointcut(切入点):只对目标对象中的某些方法进行拦截，被拦截的方法就是切入点，本例：saveUser(),updateUser()\*/

**if**(method.getName().contains("save") || method.getName().contains("update")){

security();

System.*out*.println("代理对象实例："+proxy.getClass());

System.*out*.println("目标对象方法的名称："+method.getName());

**if**(args!=**null** && args.length>0){

**for**(Object o :args){

System.*out*.println("参数："+o);

}

}

}

//访问目标对象(传递2个参数，参数1：目标对象，参数2：目标对象接收的参数)，返回值Object，指的是目标对象方法的返回值

Object returnValue = method.invoke(**this**.targetObject, args);

**return** returnValue;

}

//安全校验的方法

/\*\*Advice(通知):通知通常指的是方法，切入点需要做的事情，本例：security();\*/

**public** **void** security(){

System.*out*.println("开始进行安全校验！！！");

}

}

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

//直接访问目标对象

// IUserService userService = new UserServiceImpl();

// userService.saveUser("超级强", "123");

// userService.updateUser("超级强", "123");

// userService.deleteUser("超级强");

// String str = userService.findUser();

// System.out.println("str:"+str);

//使用代理对象访问目标对象

//需要目标对象

IUserService userService = **new** UserServiceImpl();

//使用目标对象，创建一个代理对象(代理对象与目标对象实现相同的接口)

JDKProxy jdkProxy = **new** JDKProxy();

IUserService proxyUserService = (IUserService) jdkProxy.createProxyWithTarget(userService);

proxyUserService.saveUser("超级强", "123");

proxyUserService.updateUser("超级强", "123");

proxyUserService.deleteUser("超级强");

String str = proxyUserService.findUser();

System.*out*.println("str:"+str);

}

}

### 2：CGLIB代理

**CGlib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。**

**import** java.lang.reflect.Method;

**import** net.sf.cglib.proxy.Enhancer;

**import** net.sf.cglib.proxy.MethodInterceptor;

**import** net.sf.cglib.proxy.MethodProxy;

**public** **class** UserManagerCglibProxy **implements MethodInterceptor** {

**private** Object targerObject; //代理的目标对象

/\*\*

\* 创建目标对象的代理对象

\* **@param** targerObject 目标对象

\* **@return**

\*/

**public** Object createProxyInstance(Object targerObject){

**this**.targerObject=targerObject;

Enhancer enhancer=**new** Enhancer(); //该类用于生成代理对象

enhancer.setSuperclass(**this**.targerObject.getClass()); //设置父类

enhancer.setCallback(**this**); //设置回调用对象为本身

**return** enhancer.create(); //创建代理对象

}

/\*\*

\* **@param** obj 目标对象代理类的实例

\* **@param** method 代理实例上调用父类方法的Method实例

\* **@param** args 传入到代理实例上方法参数值的对象数组

\* **@param** methodProxy 使用它调用父类的方法

\* **@return**

\* **@throws** Throwable

\*/

**public** Object intercept(Object obj, Method method, Object[] args,

MethodProxy methodProxy) **throws** Throwable {

System.*out*.println("代理类 "+obj.getClass());

System.*out*.println("方法名称 "+method.getName());

**if**(args!=**null**&&args.length>0){//方法的参数值

**for**(**int** i=0;i<args.length;i++){

System.*out*.println("方法参数 "+args[i]); }}

Object returnvalue =**null**; //方法的返回值,无返回类型时，为null

returnvalue=methodProxy.invoke(**this**.targerObject, args); //调用目标对象的方法

System.*out*.println("方法的返回值 "+returnvalue);

**return** returnvalue;

}

}

**这里要引入cglib的相关包：com.springsource.net.sf.cglib-2.1.3.jar，从spring提供的第三方包中获取。**

#### （1）测试代码

0402\_Spring\_demo04

结构：



* UserServiceImpl类

/\*\*Target(目标对象)：需要访问的真实主题就是目标对象，本例：UserServiceImpl类\*/

**public** **class** UserServiceImpl {

/\*\*joinpoint(连接点)：目标对象中定义的所有方法，该方法就是连接点，本例：saveUser(),updateUser(),deleteUser(),findUser()\*/

**public** **void** saveUser(String name, String password) {

System.*out*.println("【新增】用户名："+name+"，密码："+password);

}

**public** **void** updateUser(String name, String password) {

System.*out*.println("【修改】用户名："+name+"，密码："+password);

}

**public** **void** deleteUser(String name) {

System.*out*.println("【删除】用户名："+name);

}

**public** String findUser() {

System.*out*.println("【查询】用户");

**return** "小强";

}

}

* CglibProxy代理类（Cglib代理模式）

//使用cglib代理的方式，创建代理对象

/\*\*Proxy（代理对象）：在访问目标对象方法之前或者之后，先要执行代理对象的方法，本例：CglibProxy类\*/

/\*\*Aspect(切面): 通知是一个方法，需要放置在类中，该类就是切面，本例：CglibProxy\*/

**public** **class** CglibProxy **implements** MethodInterceptor {

//目标对象

**private** Object targetObject;

//使用目标对象创建一个代理对象

/\*\*Weaving(织入):是指把切面应用到目标对象来创建新的代理对象的过程.切面在指定的连接点织入到目标对象\*/

**public** Object createProxyWithTarget(Object targetObject) {

**this**.targetObject = targetObject;

//使用Cglib创建代理对象

Enhancer enhancer = **new** Enhancer();

//使用类加载器

enhancer.setClassLoader(**this**.targetObject.getClass().getClassLoader());

//使用目标对象调用的类

enhancer.setSuperclass(**this**.targetObject.getClass());

//调用回调对象，每次访问目标对象方法的时候，先会执行intercept的方法

enhancer.setCallback(**this**);

//创建代理对象

**return** enhancer.create();

}

//访问目标对象，要先执行intercept的方法

/\*

\* 参数：

\* Object proxy：代理对象

\* Method method：通过Method对象可以访问到目标对象中的方法

\* Object[] args：传递给目标对象方法的参数

\* MethodProxy methodProxy：访问代理对象的方法，获取签名对象用来获取

\* 返回值：

\* Object，表示目标对象方法的返回值

\*/

**public** Object intercept(Object proxy, Method method, Object[] args,

MethodProxy methodProxy) **throws** Throwable {

/\*\*Pointcut(切入点):只对目标对象中的某些方法进行拦截，被拦截的方法就是切入点，本例：saveUser(),updateUser()\*/

**if**(method.getName().contains("save") || method.getName().contains("update")){

security();

System.*out*.println("代理对象实例："+proxy.getClass());

System.*out*.println("目标对象方法的名称："+method.getName());

**if**(args!=**null** && args.length>0){

**for**(Object o :args){

System.*out*.println("参数："+o);

}

}

//使用methodProxy

Signature signature = methodProxy.getSignature();

System.*out*.println(signature.getName()+" "+signature.getArgumentTypes()+" "+signature.getReturnType());

}

//访问目标对象(传递2个参数，参数1：目标对象，参数2：目标对象接收的参数)，返回值Object，指的是目标对象方法的返回值

Object returnValue = method.invoke(**this**.targetObject, args);

**return** returnValue;

}

//安全校验的方法

/\*\*Advice(通知):通知通常指的是方法，切入点需要做的事情，本例：security();\*/

**public** **void** security(){

System.*out*.println("开始进行安全校验！！！");

}

}

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

//直接访问目标对象

// IUserService userService = new UserServiceImpl();

// userService.saveUser("超级强", "123");

// userService.updateUser("超级强", "123");

// userService.deleteUser("超级强");

// String str = userService.findUser();

// System.out.println("str:"+str);

//使用代理对象访问目标对象

//需要目标对象

UserServiceImpl userService = **new** UserServiceImpl();

//使用目标对象，创建一个代理对象(代理对象与目标对象实现相同的接口)

CglibProxy cglibProxy = **new** CglibProxy();

UserServiceImpl proxyUserService = (UserServiceImpl) cglibProxy.createProxyWithTarget(userService);

proxyUserService.saveUser("超级强", "123");

proxyUserService.updateUser("超级强", "123");

proxyUserService.deleteUser("超级强");

String str = proxyUserService.findUser();

System.*out*.println("str:"+str);

}

}

## 4：spring面向切面的编程

**Spring提供2个代理模式，一个是jdk代理，另一个cglib代理**

**1.若目标对象实现了若干接口，spring使用JDK的java.lang.reflect.Proxy类代理。**

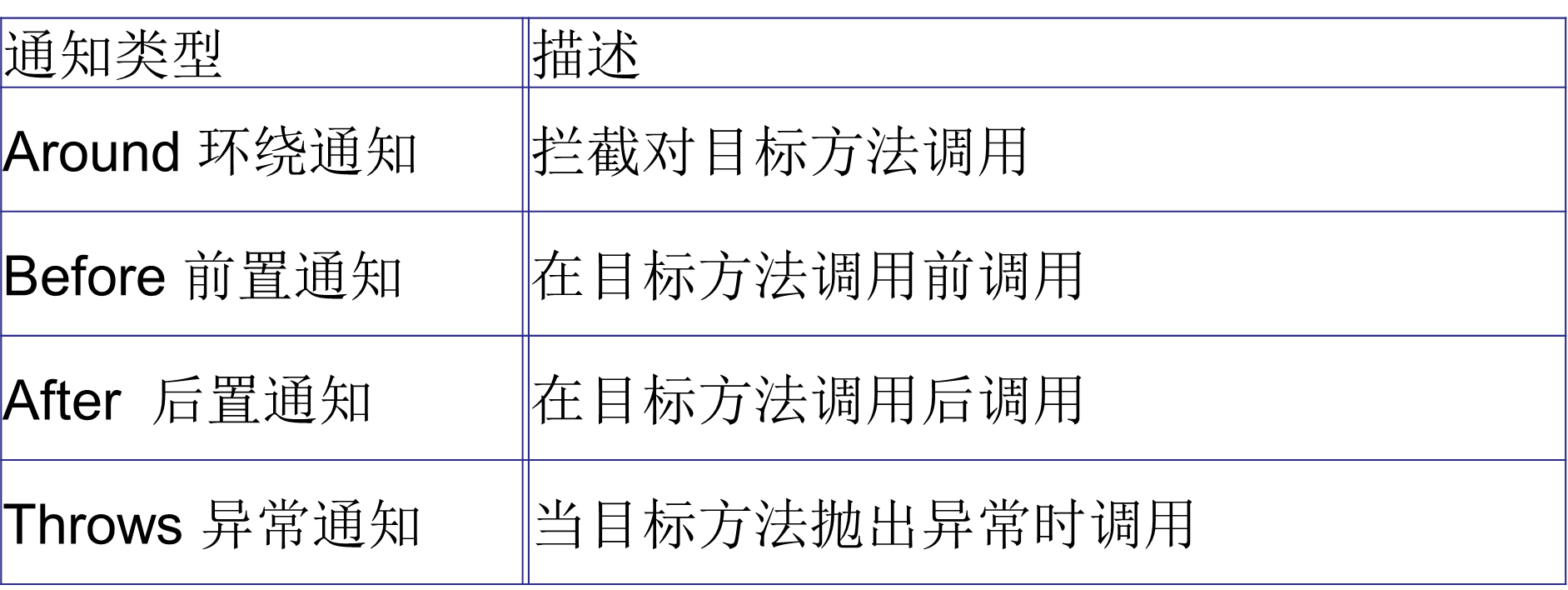
**2.若目标对象没有实现任何接口，spring使用CGLIB库生成目标对象的子类。**

注意：开发时尽量使用接口的编程，

**(1)对接口创建代理优于对类创建代理，因为会产生更加松耦合的系统。**

**(2)标记为final的方法不能够被通知。spring是为目标类产生子类。任何需要被通知的方法都被复写，将通知织入。final方法是不允许重写的。**

**(3) spring只支持方法连接点，不支持属性的连接点**



### （1）配置：

**要进行AOP编程，首先我们要在spring的配置文件中引入aop命名空间：**

<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-4.3.xsd

**http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd**">

</beans>

**Spring提供了两种切面使用方式，实际工作中我们可以选用其中一种：**

* 基于XML配置方式进行AOP开发。
* 基于注解方式进行AOP开发。

**SpringAOP编程，需要引入的jar包**

\* cglib-3.2.4.jar

cglib代理

\* aopalliance-1.0.jar

\* aspectjtools-1.8.4.jar

\* spring-aop-4.3.6.RELEASE.jar

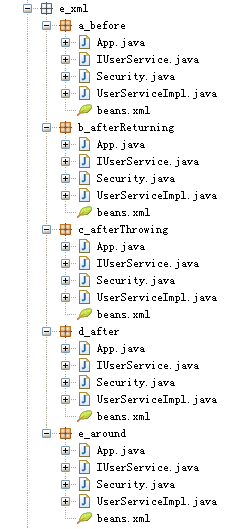
spring的面向切面编程,提供AOP(面向切面编程)实现

\* spring-aspects-4.3.6.RELEASE.jar

spring提供对AspectJ框架的整合

### （2）基于XML方式的配置：测试代码

结构：



* IuserService接口

**public** **interface** IUserService {

**public** **void** saveUser(String name,String password);

**public** **void** updateUser(String name,String password);

**public** **void** deleteUser(String name);

**public** String findUser();

}

* UserServiceImpl实现类

**public** **class** UserServiceImpl **implements** IUserService {

**public** **void** saveUser(String name, String password) {

System.*out*.println("【新增】用户名："+name+"，密码："+password);

}

**public** **void** updateUser(String name, String password) {

System.*out*.println("【修改】用户名："+name+"，密码："+password);

}

**public** **void** deleteUser(String name) {

System.*out*.println("【删除】用户名："+name);

}

**public** String findUser() {

System.*out*.println("【查询】用户");

**return** "小强";

}

}

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

//直接访问目标对象

//使用代理对象访问目标对象，当在spring的容器中添加<aop>

ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/offcn/e\_xml/a\_before/beans.xml");

IUserService userService = (IUserService)ac.getBean("userServiceImpl");

userService.saveUser("超级强", "123");

userService.updateUser("超级强", "123");

userService.deleteUser("超级强");

String str = userService.findUser();

System.*out*.println("str:"+str);

}

}

#### 【1】前置通知

* 切面类Security

/\*\*切面\*/

**public** **class** Security {

/\*\*通知\*/

/\*\*

\* 任何通知方法可以将第一个参数定义为org.aspectj.lang.JoinPoint类型 （

\* 环绕通知需要定义第一个参数为ProceedingJoinPoint类型， 它是 JoinPoint 的一个子类）。

\* 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）、

\* toString() （打印出正在被通知的方法的有用信息）。

\*/

**public** **void** checkSecurity(JoinPoint joinPoint){

System.*out*.println("正在执行验证...");

Object [] args = joinPoint.getArgs();

**if**(args!=**null** && args.length>0){

**for**(Object o:args){

System.*out*.println("参数："+o);

}

}

System.*out*.println("代理对象："+joinPoint.getThis().getClass());

System.*out*.println("目标对象："+joinPoint.getTarget().getClass());

System.*out*.println("访问目标对象方法的名称："+joinPoint.getSignature().getName());

System.*out*.println("spring容器中定义切入点的表达式："+joinPoint.toString());

}

}

* Spring容器（beans.xml）

<?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/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.3.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"*>

<!-- 创建目标对象 -->

<bean id=*"userServiceImpl"* class=*"com.offcn.a\_before.UserServiceImpl"*></bean>

<!-- 声明切面 (无灵魂)-->

**<bean id=*"security"* class=*"com.offcn.a\_before.Security"*></bean>**

<!-- spring的aop编程，所有的操作（切面、通知、切入点）都要放置到aop:config -->

<aop:config>

<!--

定义个切面，此时切面的类具有了灵魂

id：惟一标识

ref：注入对象

-->

<aop:aspect id=*"aa"* ref=*"security"*>

<!--

声明切入点

id：切入点的惟一标识

expression：切入点的表达式语言，指定项目中哪个类哪个方法作为切入点

-->

<aop:pointcut id=*"save"* expression=*"execution(\* com.offcn.a\_before.UserServiceImpl.saveUser(..))"* />

<aop:pointcut id=*"update"* expression=*"execution(\* com.offcn.a\_before.UserServiceImpl.updateUser(..))"* />

<!--

定义通知（切入点要做的事情）

前置通知：在访问目标对象方法之前，先执行通知定义的方法

特点：如果代理对象（切面）中的方法（通知）抛出异常，此时不会执行目标对象

\* pointcut-ref：注入切入点，这样才能让切入点关联通知

\* method:指定切面中定义的通知的方法

-->

**<aop:before pointcut-ref=*"save"* method=*"checkSecurity"*/>**

**<aop:before pointcut-ref=*"update"* method=*"checkSecurity"*/>**

</aop:aspect>

</aop:config>

</beans>

#### 【2】后置通知

* 切面类Security

/\*\*切面\*/

**public** **class** Security {

/\*\*通知\*/

/\*\*

\* 任何通知方法可以将第一个参数定义为org.aspectj.lang.JoinPoint类型 （

\* 环绕通知需要定义第一个参数为ProceedingJoinPoint类型， 它是 JoinPoint 的一个子类）。

\* 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）、

\* toString() （打印出正在被通知的方法的有用信息）。

\*/

**public** **void** checkSecurity(JoinPoint joinPoint**,Object returnValue**){

System.*out*.println("正在执行验证...");

Object [] args = joinPoint.getArgs();

**if**(args!=**null** && args.length>0){

**for**(Object o:args){

System.*out*.println("参数："+o);

}

}

System.*out*.println("代理对象："+joinPoint.getThis().getClass());

System.*out*.println("目标对象："+joinPoint.getTarget().getClass());

System.*out*.println("访问目标对象方法的名称："+joinPoint.getSignature().getName());

System.*out*.println("spring容器中定义切入点的表达式："+joinPoint.toString());

System.*out*.println("目标对象方法的返回值："+returnValue);

}

}

* Spring容器（beans.xml）

<?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/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.3.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"*>

<!-- 创建目标对象 -->

<bean id=*"userServiceImpl"* class=*"com.offcn.b\_afterreturning.UserServiceImpl"*></bean>

<!-- 声明切面 (无灵魂)-->

<bean id=*"security"* class=*"com.offcn.b\_afterreturning.Security"*></bean>

<aop:config>

<aop:aspect id=*"aa"* ref=*"security"*>

<aop:pointcut id=*"save"* expression=*"execution(\* com.offcn.b\_afterreturning.UserServiceImpl.saveUser(..))"* />

<aop:pointcut id=*"find"* expression=*"execution(\* com.offcn.b\_afterreturning.UserServiceImpl.findUser(..))"* />

<!--

后置通知：在访问目标对象方法之后，再执行通知定义的方法

特点：1：如果在目标对象中抛出异常，此时不会执行通知

2：因为是先执行目标对象中的方法，再执行通知，所以能不能在通知中获取目标对象的方法的返回值？能

第一步：在spring容器中定义：returning="returnValue"

第二步：在通知的方法中的第二个参数，可以指定Object类型，

例如public void checkSecurity(JoinPoint joinPoint,Object returnValue){

总结： 参数一定要放在到第二个参数的位置

参数一定一个Object类型

参数的属性名称一定要与spring容器中定义的returning相一致

-->

**<aop:after-returning pointcut-ref=*"save"* method=*"checkSecurity"* returning=*"returnValue"*/>**

**<aop:after-returning pointcut-ref=*"find"* method=*"checkSecurity"* returning=*"returnValue"*/>**

</aop:aspect>

</aop:config>

</beans>

#### 【3】异常通知

* 切面类Security

/\*\*切面\*/

**public** **class** Security {

/\*\*通知\*/

/\*\*

\* 任何通知方法可以将第一个参数定义为org.aspectj.lang.JoinPoint类型 （

\* 环绕通知需要定义第一个参数为ProceedingJoinPoint类型， 它是 JoinPoint 的一个子类）。

\* 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）、

\* toString() （打印出正在被通知的方法的有用信息）。

\*/

**public** **void** checkSecurity(JoinPoint joinPoint**,Throwable** **throwingValue**){

System.*out*.println("正在执行验证...");

Object [] args = joinPoint.getArgs();

**if**(args!=**null** && args.length>0){

**for**(Object o:args){

System.*out*.println("参数："+o);

}

}

System.*out*.println("代理对象："+joinPoint.getThis().getClass());

System.*out*.println("目标对象："+joinPoint.getTarget().getClass());

System.*out*.println("访问目标对象方法的名称："+joinPoint.getSignature().getName());

System.*out*.println("spring容器中定义切入点的表达式："+joinPoint.toString());

System.*out*.println("目标对象方法抛出的异常是："+throwingValue);

}

}

* Spring容器（beans.xml）

<?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/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.3.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"*>

<!-- 创建目标对象 -->

<bean id=*"userServiceImpl"* class=*"com.offcn.c\_afterthrowing.UserServiceImpl"*></bean>

<!-- 声明切面 (无灵魂)-->

<bean id=*"security"* class=*"com.offcn.c\_afterthrowing.Security"*></bean>

<aop:config>

<aop:aspect id=*"aa"* ref=*"security"*>

<aop:pointcut id=*"save"* expression=*"execution(\* com.offcn.c\_afterthrowing.UserServiceImpl.saveUser(..))"* />

<aop:pointcut id=*"find"* expression=*"execution(\* com.offcn.c\_afterthrowing.UserServiceImpl.findUser(..))"* />

<!--

异常通知：在访问目标对象方法之后，前提是目标对象方法中抛出异常，此时才会执行通知定义的方法

特点：1：只有目标对象方法中抛出异常，此时才会执行通知

2：在通知的方法中捕获异常

第一步：在spring容器中定义

第二步：在通知的方法中的第二个参数的位置，可以指定，例如public void checkSecurity(JoinPoint joinPoint,Throwable throwingValue){

\* 要求一：获取目标对象抛出的异常的参数要放置在第二个参数的位置

\* 要求二：类型必须指定Throwable类型

\* 要求三：Throwable对应的属性值要和spring容器中定义的throwing="throwingValue"值要相匹配

-->

**<aop:after-throwing pointcut-ref=*"save"* method=*"checkSecurity"* throwing=*"throwingValue"*/>**

**<aop:after-throwing pointcut-ref=*"find"* method=*"checkSecurity"* throwing=*"throwingValue"*/>**

</aop:aspect>

</aop:config>

</beans>

#### 【4】环绕通知

* 切面类Security

/\*\*切面\*/

**public** **class** Security {

/\*\*通知\*/

/\*\*

\* 普通通知类型：

\* 任何通知方法可以将第一个参数定义为org.aspectj.lang.JoinPoint类型 （

\* 环绕通知需要定义第一个参数为ProceedingJoinPoint类型， 它是 JoinPoint 的一个子类）。

\* 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）、

\* toString() （打印出正在被通知的方法的有用信息）。

\*

\* 环绕通知类型

\* 通知的第一个参数必须是 ProceedingJoinPoint类型。

\* 在通知体内，调用 ProceedingJoinPoint的proceed()方法会导致 后台的连接点方法执行。

\* proceed 方法也可能会被调用并且传入一个 Object[]对象-该数组中的值将被作为方法执行时的参数。

\*

\* 环绕通知将通知的方法的返回值要定义成Object类型，只有这样才能将目标对象方法的返回值，传递给客户端

\*/

public Object checkSecurity(ProceedingJoinPoint joinPoint){

//如果调用joinPoint.proceed();方法放置在通知的最前面，此时就相当于后置通知

System.out.println("环绕通知：在目标执行前调用："+joinPoint.getTarget().getClass());

Object value = null;

try {

value = joinPoint.proceed();

} catch (Throwable e) {

e.printStackTrace();

}

/\*System.out.println("正在执行验证...");

Object [] args = joinPoint.getArgs();

if(args!=null && args.length>0){

for(Object o:args){

System.out.println("参数："+o);

}

}

System.out.println("代理对象："+joinPoint.getThis().getClass());

System.out.println("目标对象："+joinPoint.getTarget().getClass());

System.out.println("访问目标对象方法的名称："+joinPoint.getSignature().getName());

System.out.println("spring容器中定义切入点的表达式："+joinPoint.toString());\*/

// //如果调用joinPoint.proceed();方法放置在通知的最后，此时就相当于前置通知

// Object value = null;

// try {

// value = joinPoint.proceed();

// } catch (Throwable e) {

// e.printStackTrace();

// }

System.out.println("环绕通知：在目标执行后调用："+joinPoint.getTarget().getClass()+" 获取返回结果:"+value);

return value;

}

}

* Spring容器（beans.xml）

<?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/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.3.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"*>

<!-- 创建目标对象 -->

<bean id=*"userServiceImpl"* class=*"com.offcn.e\_around.UserServiceImpl"*></bean>

<!-- 声明切面 (无灵魂)-->

<bean id=*"security"* class=*"com.offcn.e\_around.Security"*></bean>

<aop:config>

<aop:aspect id=*"aa"* ref=*"security"*>

<aop:pointcut id=*"save"* expression=*"execution(\* com.offcn.e\_around.UserServiceImpl.saveUser(..))"* />

<aop:pointcut id=*"find"* expression=*"execution(\* com.offcn.e\_around.UserServiceImpl.findUser(..))"* />

<!--

环绕通知：

最后一种通知是环绕通知。环绕通知在一个目标对象方法执行之前和之后执行。

它使得通知有机会 在一个方法执行之前和执行之后运行。而且它可以决定这个方法在什么时候执行，如何执行，甚至是否执行。

环绕通知经常在某线程安全的环境下，你需要在一个方法执行之前和之后共享某种状态的时候使用。

请尽量使用最简单的满足你需求的通知。（比如如果简单的前置通知也可以适用的情况下不要使用环绕通知）。

-->

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

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

</aop:aspect>

</aop:config>

</beans>

### （3）基于注解方式的配置：测试代码

**首先为了在Spring配置中使用@AspectJ切面，你首先必须启用Spring对@AspectJ切面配置的支持，并确保*自动代理*** **(蓝色部分)：**

**<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-4.3.xsd

**http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd”>**

**<!--启用Spring对@AspectJ的支持 -->**

**<aop:aspectj-autoproxy/>**

**<!-- 声明切面对象 -->**

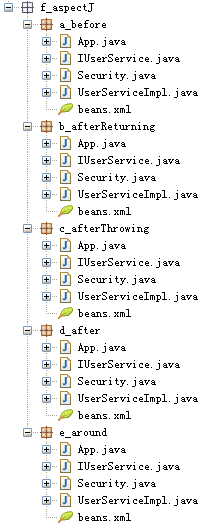
<bean id="security" class="com.offcn.service.Security" />

**<!-- 创建接口实现类对象 -->**

<bean id="userManager" class="com.offcn.service.UserManagerImpl" />

**</beans>**

结构：



* IuserService接口

**public** **interface** IUserService {

**public** **void** saveUser(String name,String password);

**public** **void** updateUser(String name,String password);

**public** **void** deleteUser(String name);

**public** String findUser();

}

* UserServiceImpl实现类

**public** **class** UserServiceImpl **implements** IUserService {

**public** **void** saveUser(String name, String password) {

System.*out*.println("【新增】用户名："+name+"，密码："+password);

}

**public** **void** updateUser(String name, String password) {

System.*out*.println("【修改】用户名："+name+"，密码："+password);

}

**public** **void** deleteUser(String name) {

System.*out*.println("【删除】用户名："+name);

}

**public** String findUser() {

System.*out*.println("【查询】用户");

**return** "小强";

}

}

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

//直接访问目标对象

//使用代理对象访问目标对象，当在spring的容器中添加<aop>

ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/offcn/a\_before/beans.xml");

IUserService userService = (IUserService)ac.getBean("userServiceImpl");

userService.saveUser("超级强", "123");

userService.updateUser("超级强", "123");

userService.deleteUser("超级强");

String str = userService.findUser();

System.*out*.println("str:"+str);

}

}

#### 【1】前置通知

* 切面类Security

/\*\*切面\*/

**@Aspect**  //表示在spring容器中定义：<aop:aspect id="aa" ref="security">

**public** **class** Security {

/\*\*

\* 声明定义目标对象方法的切入点（指的是方法）

\* 要求：1、方法的修饰符任意

\* 2、要求方法没有返回值

\* 3、方法的名称任意（一个类中，不允许出现同名方法，表明的切入点的id惟一）

\* 4、方法没有参数

\* 5、方法体为空

\*/

//相当于spring容器中定义：<aop:pointcut id="save" expression="execution(\* com.offcn.a\_before.UserServiceImpl.saveUser(..))" />

**@Pointcut(value="execution(\* com.offcn.a\_before.UserServiceImpl.saveUser(..))")**

**public** **void** save(){};

**@Pointcut(value="execution(\* com.offcn.a\_before.UserServiceImpl.updateUser(..))")**

**public** **void** update(){};

/\*\*通知\*/

/\*\*

\* 任何通知方法可以将第一个参数定义为org.aspectj.lang.JoinPoint类型 （

\* 环绕通知需要定义第一个参数为ProceedingJoinPoint类型， 它是 JoinPoint 的一个子类）。

\* 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）、

\* toString() （打印出正在被通知的方法的有用信息）。

\*/

//相当于spring容器中定义：<aop:before pointcut-ref="save" method="checkSecurity"/><aop:before pointcut-ref="update" method="checkSecurity"/>

**@Before(value="save() || update()")**

**public** **void** checkSecurity(JoinPoint joinPoint){

System.*out*.println("正在执行验证...");

Object [] args = joinPoint.getArgs();

**if**(args!=**null** && args.length>0){

**for**(Object o:args){

System.*out*.println("参数："+o);

}

}

System.*out*.println("代理对象："+joinPoint.getThis().getClass());

System.*out*.println("目标对象："+joinPoint.getTarget().getClass());

System.*out*.println("访问目标对象方法的名称："+joinPoint.getSignature().getName());

System.*out*.println("spring容器中定义切入点的表达式："+joinPoint.toString());

**if**(**true**){

**throw** **new** RuntimeException("抛出运行时异常！");

}

}

}

* Spring容器（beans.xml）

<?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/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.3.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"*>

<!-- 启用Spring对@AspectJ切面配置的支持 -->

**<aop:aspectj-autoproxy></aop:aspectj-autoproxy>**

<!-- 创建目标对象 -->

<bean id=*"userServiceImpl"* class=*"com.offcn.a\_before.UserServiceImpl"*></bean>

<!-- 声明切面 (无灵魂)-->

<bean id=*"security"* class=*"com.offcn.a\_before.Security"*></bean>

</beans>

#### 【2】后置通知

* 切面类Security

/\*\*切面\*/

**@Aspect** //表示在spring容器中定义：<aop:aspect id="aa" ref="security">

**public** **class** Security {

/\*\*

\* 声明定义目标对象方法的切入点（指的是方法）

\* 要求：1、方法的修饰符任意

\* 2、要求方法没有返回值

\* 3、方法的名称任意（一个类中，不允许出现同名方法，表明的切入点的id惟一）

\* 4、方法没有参数

\* 5、方法体为空

\*/

//相当于spring容器中定义：<aop:pointcut id="save" expression="execution(\* com.offcn.a\_before.UserServiceImpl.saveUser(..))" />

**@Pointcut(value="execution(\* com.offcn.b\_afterreturning.UserServiceImpl.saveUser(..))")**

**public** **void** save(){};

**@Pointcut(value="execution(\* com.offcn.b\_afterreturning.UserServiceImpl.findUser(..))")**

**public** **void** find(){};

/\*\*通知\*/

/\*\*

\* 任何通知方法可以将第一个参数定义为org.aspectj.lang.JoinPoint类型 （

\* 环绕通知需要定义第一个参数为ProceedingJoinPoint类型， 它是 JoinPoint 的一个子类）。

\* 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）、

\* toString() （打印出正在被通知的方法的有用信息）。

\*/

//相当于spring容器中定义：<aop:before pointcut-ref="save" method="checkSecurity"/><aop:before pointcut-ref="update" method="checkSecurity"/>

**@AfterReturning(value="save() || find()",returning="returnValue")**

**public** **void** checkSecurity(JoinPoint joinPoint,Object returnValue){

System.*out*.println("正在执行验证...");

Object [] args = joinPoint.getArgs();

**if**(args!=**null** && args.length>0){

**for**(Object o:args){

System.*out*.println("参数："+o);

}

}

System.*out*.println("代理对象："+joinPoint.getThis().getClass());

System.*out*.println("目标对象："+joinPoint.getTarget().getClass());

System.*out*.println("访问目标对象方法的名称："+joinPoint.getSignature().getName());

System.*out*.println("spring容器中定义切入点的表达式："+joinPoint.toString());

System.*out*.println("目标对象方法的返回值："+returnValue);

// if(true){

// throw new RuntimeException("抛出运行时异常！");

// }

}

}

* Spring容器（beans.xml）

<?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/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.3.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"*>

<!-- 启用Spring对@AspectJ切面配置的支持 -->

**<aop:aspectj-autoproxy></aop:aspectj-autoproxy>**

<!-- 创建目标对象 -->

<bean id=*"userServiceImpl"* class=*"com.offcn.b\_afterreturning.UserServiceImpl"*></bean>

<!-- 声明切面 (无灵魂)-->

<bean id=*"security"* class=*"com.offcn.b\_afterreturning.Security"*></bean>

</beans>

#### 【3】异常通知

* 切面类Security

/\*\*切面\*/

**@Aspect** //表示在spring容器中定义：<aop:aspect id="aa" ref="security">

**public** **class** Security {

/\*\*

\* 声明定义目标对象方法的切入点（指的是方法）

\* 要求：1、方法的修饰符任意

\* 2、要求方法没有返回值

\* 3、方法的名称任意（一个类中，不允许出现同名方法，表明的切入点的id惟一）

\* 4、方法没有参数

\* 5、方法体为空

\*/

//相当于spring容器中定义：<aop:pointcut id="save" expression="execution(\* com.offcn.a\_before.UserServiceImpl.saveUser(..))" />

**@Pointcut(value="execution(\* com.offcn.c\_afterthrowing.UserServiceImpl.saveUser(..))")**

**public** **void** save(){};

**@Pointcut(value="execution(\* com.offcn.c\_afterthrowing.UserServiceImpl.findUser(..))")**

**public** **void** find(){};

/\*\*通知\*/

/\*\*

\* 任何通知方法可以将第一个参数定义为org.aspectj.lang.JoinPoint类型 （

\* 环绕通知需要定义第一个参数为ProceedingJoinPoint类型， 它是 JoinPoint 的一个子类）。

\* 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）、

\* toString() （打印出正在被通知的方法的有用信息）。

\*/

//相当于spring容器中定义：<aop:before pointcut-ref="save" method="checkSecurity"/><aop:before pointcut-ref="update" method="checkSecurity"/>

**@AfterThrowing(value="save() || find()",throwing="throwingValue")**

**public** **void** checkSecurity(JoinPoint joinPoint,Throwable throwingValue){

System.*out*.println("正在执行验证...");

Object [] args = joinPoint.getArgs();

**if**(args!=**null** && args.length>0){

**for**(Object o:args){

System.*out*.println("参数："+o);

}

}

System.*out*.println("代理对象："+joinPoint.getThis().getClass());

System.*out*.println("目标对象："+joinPoint.getTarget().getClass());

System.*out*.println("访问目标对象方法的名称："+joinPoint.getSignature().getName());

System.*out*.println("spring容器中定义切入点的表达式："+joinPoint.toString());

System.*out*.println("目标对象抛出的异常："+throwingValue);

}

}

* Spring容器（beans.xml）

<?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/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.3.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"*>

<!-- 启用Spring对@AspectJ切面配置的支持 -->

**<aop:aspectj-autoproxy></aop:aspectj-autoproxy>**

<!-- 创建目标对象 -->

<bean id=*"userServiceImpl"* class=*"com.offcn.c\_afterthrowing.UserServiceImpl"*></bean>

<!-- 声明切面 (无灵魂)-->

<bean id=*"security"* class=*"com.offcn.c\_afterthrowing.Security"*></bean>

</beans>

#### 【4】环绕通知

* 切面类Security

/\*\*切面\*/

**@Aspect** //表示在spring容器中定义：<aop:aspect id="aa" ref="security">

**public** **class** Security {

/\*\*

\* 声明定义目标对象方法的切入点（指的是方法）

\* 要求：1、方法的修饰符任意

\* 2、要求方法没有返回值

\* 3、方法的名称任意（一个类中，不允许出现同名方法，表明的切入点的id惟一）

\* 4、方法没有参数

\* 5、方法体为空

\*/

//相当于spring容器中定义：<aop:pointcut id="save" expression="execution(\* com.offcn.a\_before.UserServiceImpl.saveUser(..))" />

**@Pointcut(value="execution(\* com.offcn.e\_around.UserServiceImpl.saveUser(..))")**

**public** **void** save(){};

**@Pointcut(value="execution(\* com.offcn.e\_around.UserServiceImpl.findUser(..))")**

**public** **void** find(){};

/\*\*通知\*/

/\*\*

\* 任何通知方法可以将第一个参数定义为org.aspectj.lang.JoinPoint类型 （

\* 环绕通知需要定义第一个参数为ProceedingJoinPoint类型， 它是 JoinPoint 的一个子类）。

\* 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）、

\* toString() （打印出正在被通知的方法的有用信息）。

\*/

//相当于spring容器中定义：<aop:before pointcut-ref="save" method="checkSecurity"/><aop:before pointcut-ref="update" method="checkSecurity"/>

**@Around(value="save() || find()")**

**public** Object checkSecurity(**ProceedingJoinPoint joinPoint**){

System.*out*.println("正在执行验证...");

Object [] args = joinPoint.getArgs();

**if**(args!=**null** && args.length>0){

**for**(Object o:args){

System.*out*.println("参数："+o);

}

}

System.*out*.println("代理对象："+joinPoint.getThis().getClass());

System.*out*.println("目标对象："+joinPoint.getTarget().getClass());

System.*out*.println("访问目标对象方法的名称："+joinPoint.getSignature().getName());

System.*out*.println("spring容器中定义切入点的表达式："+joinPoint.toString());

//调用proceed，放置到最后，此时相当于前置通知

Object value = **null**;

**try** {

**value = joinPoint.proceed();**

} **catch** (Throwable e) {

e.printStackTrace();

}

**return** value;

}

}

* Spring容器（beans.xml）

<?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/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.3.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"*>

<!-- 启用Spring对@AspectJ切面配置的支持 -->

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

<!-- 创建目标对象 -->

<bean id=*"userServiceImpl"* class=*"com.offcn.e\_around.UserServiceImpl"*></bean>

<!-- 声明切面 (无灵魂)-->

<bean id=*"security"* class=*"com.offcn.e\_around.Security"*></bean>

</beans>

简化版：

/\*简化版\*/  
@Before(value = "execution(\* com.offcn.zhujie.UserServiceImpl.deleteUser(..))")  
public void show(){  
 System.out.println("======我这个通知要加到com.offcn.zhujie.UserServiceImpl.deleteUser方法上去");  
}

总结：

1：spring的5种类型，目的在访问目标对象方法之前或者之后，先执行通知定义的方法，这种编程就是aop方式的编程思想，即面向切面的编程。

\* 应用（spring提供的声明式事务处理），后面讲

2：对于环绕通知来说：

Object value = joinPoint.proceed();就类似于jdk代理和cglib代理的Object returnValue = method.invoke(this.targetObject, args);，此时效果是一样的