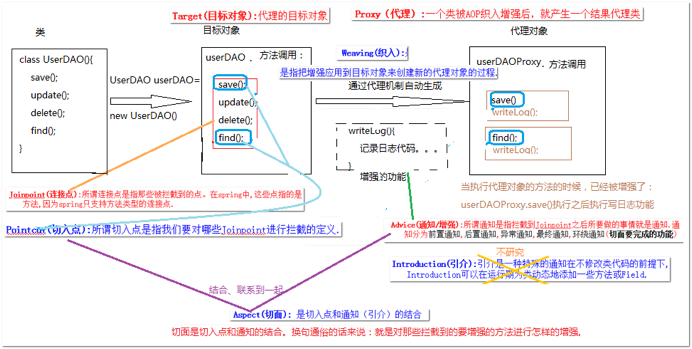
# Spring4第三天

内容回顾：

1. 常用注解
   1. 用于创建bean对象的注解
      1. @Component
      2. @Controller
      3. @Service
      4. @Repository
   2. 注入
      1. @Value
      2. @Autowired:按照类型注入
      3. @Qualifier
      4. @Resource:直接按照id注入对象类型
   3. 范围
      1. @Scope
   4. 生命周期
      1. @PostContruct
      2. @PreDestroy
2. spring整合junit单元测试
   1. 引入依赖
   2. 在单元测试类上加@RunWith
   3. 在单元测试类上加@ContextConfiguration指定spring配置文件路径
3. AOP：面向切面编程，在不修改源码的情况下实现功能的增强。
4. AOP原理：动态代理技术

# Spring的AOP中相关术语的介绍



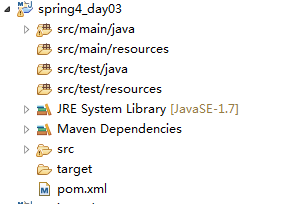


# Spring中AOP编程（xml方式）

## 基于xml的Spring AOP开发入门

### 创建工程并引入依赖

首先，创建spring4\_day03工程：



AOP开发，IOC的依赖不能少；同时引入AOP的依赖，坐标如下：

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

</dependency>

完整的pom.xml如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day03</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

</dependency>

**<dependency>**

**<groupId>org.springframework</groupId>**

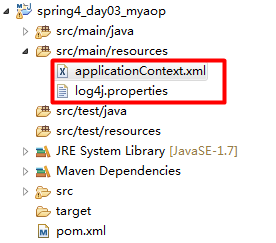
**<artifactId>spring-aspects</artifactId>**

**</dependency>**

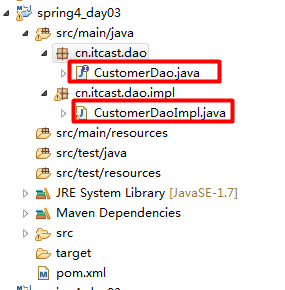
</dependencies>

</project>

* 引入applicationContext.xml和log4j.properties



### 编写接口和实现类



创建CustomerDao接口：

**public** **interface** CustomerDao {

**public** **void** save();

}

创建CustomerDao接口的实现类CustomerDaoImpl：

**public** **class** CustomerDaoImpl **implements** CustomerDao {

@Override

**public** **void** save() {

System.***out***.println("持久层：客户保存...");

}

}

### 配置相关类到Spring中

* 把CustomerDaoImpl配置在Spring中

<?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*

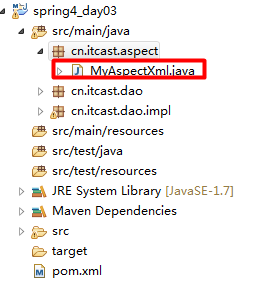
*http://www.springframework.org/schema/context/spring-context.xsd"*>

<bean id=*"customerDao"* class=*"cn.itcast.dao.impl.CustomerDaoImpl"*></bean>

</beans>

### 编写切面类

创建切面类MyAspectXml：



/\*\*

\* 自定义切面类

\* **@author** kevin

\*/

**public** **class** MyAspectXml {

**public** **void** writeLog(){

System.***out***.println("记录日志啦.....");

}

}

### 配置切面类

把MyAspectXml切面类在applicationContext.xml中配置：

<?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: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=*"customerDao"* class=*"cn.itcast.dao.impl.CustomerDaoImpl"*></bean>

**<bean id=*"myAspectXml"* class=*"cn.itcast.aspect.MyAspectXml"*></bean>**

</beans>

### 进行AOP配置

在applicationContext.xml配置AOP相关的信息：

<?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.xsd*

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

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

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

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

<bean id=*"customerDao"* class=*"cn.itcast.dao.impl.CustomerDaoImpl"*></bean>

<bean id=*"myAspectXml"* class=*"cn.itcast.aspect.MyAspectXml"*></bean>

**<!-- AOP配置 -->**

**<aop:config>**

**<!-- 配置切入点：告诉spring框架哪些方法需要被增强 -->**

**<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))"* id=*"pointcut1"*/>**

**<!-- 配置切面：告诉spring框架调用切面类中的哪个方法来增强 -->**

**<aop:aspect ref=*"myAspectXml"*>**

**<aop:before method=*"writeLog"* pointcut-ref=*"pointcut1"*/>**

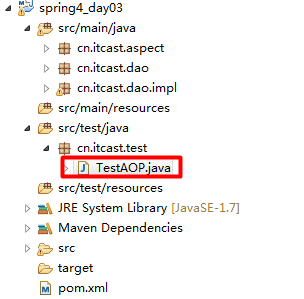
**</aop:aspect>**

**</aop:config>**

</beans>

### 编写测试类

创建单元测试类TestAOP，在其中创建单元测试方法test1:



@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestAOP {

@Autowired

**private** CustomerDao customerDao;

@Test

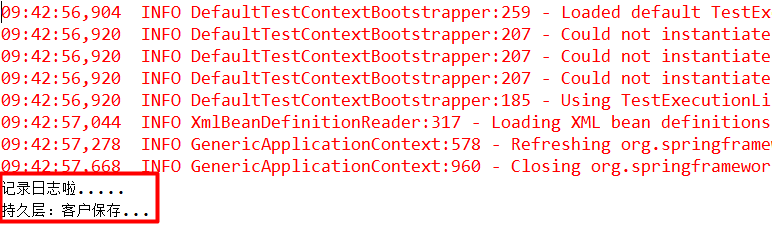
**public** **void** test1(){

customerDao.save();

}

}

运行test1方法，观察结果，发现save方法被增强了：



Aop代码编写流程

1. 编写目标类
2. 编写切面类以及通知方法
3. 把目标类和切面类交给spring管理
4. 指定切面类：用哪个类来增强
5. 配置切入点表达式：指定哪些方法需要增强
6. 配置通知：指定遇到满足切入点表达式的方式用切面类中的哪个方法增强

## 切入点表达式的语法



切入点表达式语法：[修饰符] 返回类型 包名.类名.方法名(形式参数)

常见写法：

* execution(public \* \*(..)) 所有的public方法
* execution(\* set\*(..)) 所有set开头的方法
* **execution(\* com.xyz.service.AccountService.\*(..)) AccountService类中的所有方法**
* **execution(\* com.xyz.service.\*.\*(..))**  **com.xyz.service包下所有的方法**
* execution(\* com.xyz.service..\*.\*(..)) · com.xyz.service包及其子包下所有的方法

## Spring中AOP的通知类型

在AOP开发入门中，我们写的是一个前置通知，表示在目标方法执行之前增强



那么，除了前置通知以外，Spring还提供了其它的通知类型：

1. 前置通知
2. 后置通知
3. 环绕通知
4. 异常通知
5. 最终通知

### 前置通知

应用： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志

CustomerDaoImpl接口：

**public** **interface** CustomerDao {

**public** **void** save();

}

CustomerDaoImpl实现类：

**public** **class** CustomerDaoImpl **implements** CustomerDao {

@Override

**public** **void** save() {

System.***out***.println("持久层：客户保存...");

}

}

现在需要对CustomerDaoImpl中的save方法前置增强，需要创建一个切面类MyAspectXml，在其中创建前置通知方法before:

/\*\*

\* 切面类

\*

\* **@author** xiaokaibo

\*

\*/

**public** **class** MyAspectXml {

/\*\*

\* 前置通知方法 应用场景： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志

\* joinPoint:连接点，指的是被增强的那个方法

\*/

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

String username = "rose";

**if** (!"admin".equals(username)) {

// 非admin用户，不具备权限，抛出异常

//joinPoint.getTarget().getClass().getName()获取目标类的名字

//joinPoint.getSignature().getName()获取被增强方法的名字

**throw** **new** RuntimeException("对不起！您没有对" + joinPoint.getTarget().getClass().getName() + "类中"

+ joinPoint.getSignature().getName() + "方法的访问权限");

}

}

}

在applicationContext.xml中配置前置通知，在执行save方法之前，执行before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: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"*>

<bean id=*"customerDao"* class=*"cn.itcast.dao.impl.CustomerDaoImpl"*></bean>

<!-- 配置切面类 -->

<bean id=*"myAspectXml"* class=*"cn.itcast.aspect.MyAspectXml"*></bean>

<aop:config>

<!-- 配置切入点表达式：告诉spring框架哪些方法需要被增强 -->

<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))"* id=*"pointcut1"*/>

<aop:aspect ref=*"myAspectXml"*>

<!-- 配置前置通知：告诉spring在目标方法执行之前增强 -->

<aop:before method=*"before"* pointcut-ref=*"pointcut1"*/>

</aop:aspect>

</aop:config>

</beans>

在TestAOP中创建单元测试方法test1：

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestAOP {

@Autowired

**private** CustomerDao customerDao;

@Test

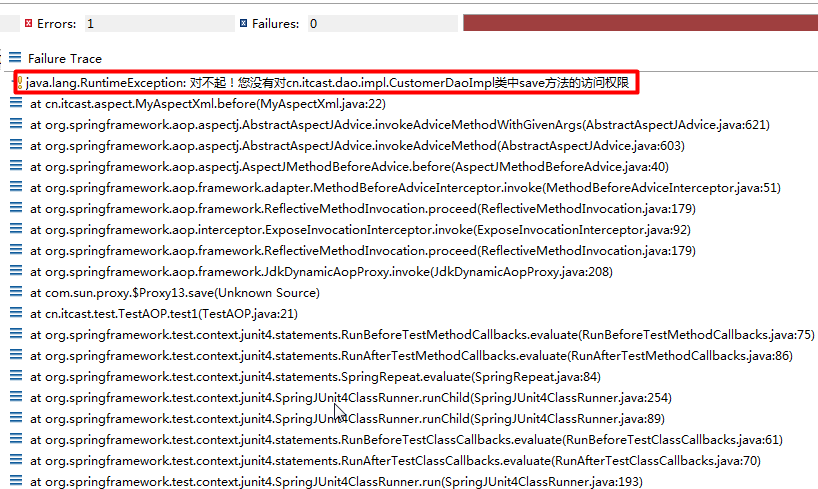
**public** **void** test1(){

customerDao.save();

}

}

测试结果如下：



### 后置通知

特点：在目标方法运行后，返回值后执行增强代码逻辑。

应用场景：与业务相关的，如ATM取款机取款后，自动下发短信。

* 在CustomerDao接口中增加delete方法，如下：

**public** **interface** CustomerDao {

**public** **void** save();

**public** Integer delete();

}

* 在目标类CustomerDaoImpl中编写目标方法delete方法，需要对该方法增强

**public** **class** CustomerDaoImpl **implements** CustomerDao {

@Override

**public** **void** save() {

System.***out***.println("持久层：客户保存...");

}

@Override

**public** Integer delete() {

System.***out***.println("持久层：客户删除...");

**return** 100;

}

}

* 在切面类MyAspectXml中定义一个后置通知方法afterReturning方法，并添加一个形参result,表示目标方法的返回值：

/\*\*

\* 切面类

\*

\* **@author** xiaokaibo

\*

\*/

**public** **class** MyAspectXml {

/\*\*

\* 前置通知方法 应用场景： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志

\* joinPoint:连接点，指的是被增强的那个方法

\*/

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

String username = "rose";

**if** (!"admin".equals(username)) {

// 非admin用户，不具备权限，抛出异常

//joinPoint.getTarget().getClass().getName()获取目标类的名字

//joinPoint.getSignature().getName()获取被增强方法的名字

**throw** **new** RuntimeException("对不起！您没有对" + joinPoint.getTarget().getClass().getName() + "类中"

+ joinPoint.getSignature().getName() + "方法的访问权限");

}

}

/\*\*

\* 后置通知方法

\* 应用场景： ATM取款机取款后，自动下发短信

\* 参数result:被增强那个方法的返回值

\*/

**public** **void** afterReturning(JoinPoint joinPoint,Object result){

Date date = **new** Date();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateStr = sdf.format(date);

System.***out***.println("尊敬的用户，您于" + dateStr + "取款" + result + "元");

}

}

* 在applicationContext.xml中配置切入点及后后置通知，并指定returning属性的值为result，通知方法的形参的名字要与returning的值一致：

<?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"*>

<bean id=*"customerDao"* class=*"cn.itcast.dao.impl.CustomerDaoImpl"*></bean>

<!-- 配置切面类 -->

<bean id=*"myAspectXml"* class=*"cn.itcast.aspect.MyAspectXml"*></bean>

<aop:config>

<!-- 配置切入点表达式：告诉spring框架哪些方法需要被增强 -->

<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))"* id=*"pointcut1"*/>

<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.delete(..))"* id=*"pointcut2"*/>

<aop:aspect ref=*"myAspectXml"*>

<!-- 配置前置通知：告诉spring在目标方法执行之前增强 -->

<aop:before method=*"before"* pointcut-ref=*"pointcut1"*/>

<!-- 配置后置通知 -->

<aop:after-returning method=*"afterReturning"* pointcut-ref=*"pointcut2"* **returning=*"result"***/>

</aop:aspect>

</aop:config>

</beans>

* 修改TestAOP中的test1方法，如下：

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestAOP {

@Autowired

**private** CustomerDao customerDao;

@Test

**public** **void** test1(){

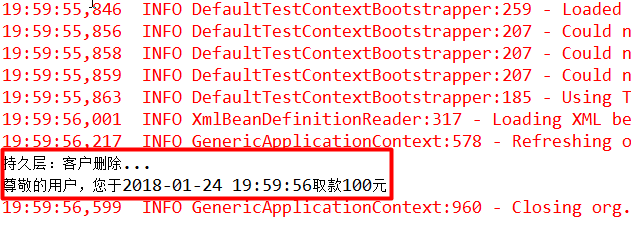
// customerDao.save();

customerDao.delete();

}

}

* 测试结果如下：



### 环绕通知

特点：目标执行前后，都进行增强（控制目标方法执行）

应用场景：日志、缓存、权限、性能监控、**事务管理**

**增强代码的方法要求：**

接受的参数：ProceedingJoinPoint（可执行的连接点）

返回值：Object返回值

抛出Throwable异常。

* 在CustomerDao接口中增加update方法

**public** **interface** CustomerDao {

**public** **void** save();

**public** Integer delete();

**public** **void** update();

}

* 在目标类CustomerDaoImpl中编写目标方法update方法，需要对该方法增强

**public** **class** CustomerDaoImpl **implements** CustomerDao {

@Override

**public** **void** save() {

System.***out***.println("持久层：客户保存...");

}

@Override

**public** Integer delete() {

System.***out***.println("持久层：客户删除...");

**return** 100;

}

@Override

**public** **void** update() {

System.***out***.println("持久层：客户更新...");

}

}

* 在切面类MyAspectXml中定义一个环绕通知方法around方法，需要给该方法增加一个形参ProceedingJoinPoint ，表示正在执行的连接点（目标）。

/\*\*

\* 切面类

\*

\* **@author** xiaokaibo

\*

\*/

**public** **class** MyAspectXml {

/\*\*

\* 前置通知方法 应用场景： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志

\* joinPoint:连接点，指的是被增强的那个方法

\*/

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

String username = "rose";

**if** (!"admin".equals(username)) {

// 非admin用户，不具备权限，抛出异常

//joinPoint.getTarget().getClass().getName()获取目标类的名字

//joinPoint.getSignature().getName()获取被增强方法的名字

**throw** **new** RuntimeException("对不起！您没有对" + joinPoint.getTarget().getClass().getName() + "类中"

+ joinPoint.getSignature().getName() + "方法的访问权限");

}

}

/\*\*

\* 后置通知方法

\* 应用场景： ATM取款机取款后，自动下发短信

\* 参数result:被增强那个方法的返回值

\*/

**public** **void** afterReturning(JoinPoint joinPoint,Object result){

Date date = **new** Date();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateStr = sdf.format(date);

System.***out***.println("尊敬的用户，您于" + dateStr + "取款" + result + "元");

}

/\*\*

\* 环绕通知方法

\* 应用场景：事务处理

\* **@param** proceedingJoinPoint 正在执行的连接点

\* **@return**

\*/

**public** Object around(ProceedingJoinPoint proceedingJoinPoint){

System.***out***.println("开启事务");

//获取目标方法的参数

Object[] args = proceedingJoinPoint.getArgs();

Object result = **null**;

**try** {

//调用目标方法，获取目标方法的返回值

result = proceedingJoinPoint.proceed(args);

System.***out***.println("提交事务");

} **catch** (Throwable e) {

System.***out***.println("回滚事务");

}

//返回目标方法的返回值

**return** result;

}

}

说明：

ProceedingJoinPoint：表示正在执行的连接点，也就是目标方法

joinpoint.proceed表示调用目标方法

* 在applicationContext.xml中配置切入点表达式及环绕通知

<?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"*>

<bean id=*"customerDao"* class=*"cn.itcast.dao.impl.CustomerDaoImpl"*></bean>

<!-- 配置切面类 -->

<bean id=*"myAspectXml"* class=*"cn.itcast.aspect.MyAspectXml"*></bean>

<aop:config>

<!-- 配置切入点表达式：告诉spring框架哪些方法需要被增强 -->

<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))"* id=*"pointcut1"*/>

<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.delete(..))"* id=*"pointcut2"*/>

<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.update(..))"* id=*"pointcut3"*/>

<aop:aspect ref=*"myAspectXml"*>

<!-- 配置前置通知：告诉spring在目标方法执行之前增强 -->

<aop:before method=*"before"* pointcut-ref=*"pointcut1"*/>

<!-- 配置后置通知 -->

<aop:after-returning method=*"afterReturning"* pointcut-ref=*"pointcut2"* returning=*"result"*/>

<!-- 配置环绕通知 -->

<aop:around method=*"around"* pointcut-ref=*"pointcut3"*/>

</aop:aspect>

</aop:config>

</beans>

* 修改TestAOP中的test1方法：

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestAOP {

@Autowired

**private** CustomerDao customerDao;

@Test

**public** **void** test1(){

// customerDao.save();

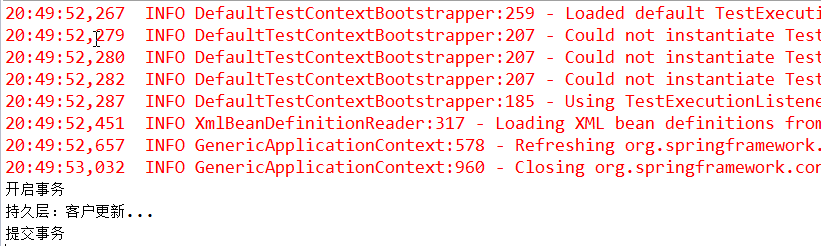
// customerDao.delete();

customerDao.update();

}

}

* 测试结果如下图：



### 异常抛出通知

作用：目标代码出现异常，通知执行。记录异常日志、通知管理员（短信、邮件）

应用场景：处理异常（一般不可预知），记录日志

* 在CustomerDao接口中增加list方法：

**public** **interface** CustomerDao {

**public** **void** save();

**public** Integer delete();

**public** **void** update();

**public** **void** list();

}

* 在目标类CustomerDaoImpl中编写目标方法list方法，需要对该方法增强

**public** **class** CustomerDaoImpl **implements** CustomerDao {

@Override

**public** **void** save() {

System.***out***.println("持久层：客户保存...");

}

@Override

**public** Integer delete() {

System.***out***.println("持久层：客户删除...");

**return** 100;

}

@Override

**public** **void** update() {

System.***out***.println("持久层：客户更新...");

}

@Override

**public** **void** list() {

System.***out***.println("持久层：客户查询...");

//自己造一个异常

**int** i = 10 / 0;

}

}

* 在切面类MyAspectXml中定义一个异常通知方法afterThrowing方法，在方法加一个异常参数，名字叫ex

/\*\*

\* 切面类

\*

\* **@author** xiaokaibo

\*

\*/

**public** **class** MyAspectXml {

/\*\*

\* 前置通知方法 应用场景： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志 joinPoint:连接点，指的是被增强的那个方法

\*/

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

String username = "rose";

**if** (!"admin".equals(username)) {

// 非admin用户，不具备权限，抛出异常

// joinPoint.getTarget().getClass().getName()获取目标类的名字

// joinPoint.getSignature().getName()获取被增强方法的名字

**throw** **new** RuntimeException("对不起！您没有对" + joinPoint.getTarget().getClass().getName() + "类中"

+ joinPoint.getSignature().getName() + "方法的访问权限");

}

}

/\*\*

\* 后置通知方法 应用场景： ATM取款机取款后，自动下发短信 参数result:被增强那个方法的返回值

\*/

**public** **void** afterReturning(JoinPoint joinPoint, Object result) {

Date date = **new** Date();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateStr = sdf.format(date);

System.***out***.println("尊敬的用户，您于" + dateStr + "取款" + result + "元");

}

/\*\*

\* 环绕通知方法 应用场景：事务处理

\*

\* **@param** proceedingJoinPoint

\* 正在执行的连接点

\* **@return**

\*/

**public** Object around(ProceedingJoinPoint proceedingJoinPoint) {

System.***out***.println("开启事务");

// 获取目标方法的参数

Object[] args = proceedingJoinPoint.getArgs();

Object result = **null**;

**try** {

// 调用目标方法，获取目标方法的返回值

result = proceedingJoinPoint.proceed(args);

System.***out***.println("提交事务");

} **catch** (Throwable e) {

System.***out***.println("回滚事务");

}

// 返回目标方法的返回值

**return** result;

}

/\*\*

\* 异常通知方法

\* 应用场景：处理异常

\* **@param** ex 目标方法抛出的异常，名字要与配置文件中配置的名字一致

\*/

**public** **void** afterThrowing(JoinPoint joinPoint, Throwable ex) {

System.***out***.println("注意了:在" + joinPoint.getTarget().getClass().getName() + "中的"

+ joinPoint.getSignature().getName() + "方法中发生了异常：" + ex.getMessage());

}

}

* 在applicationContext.xml中配置切入点表达式及异常通知，需要给异常通知指定throwing属性，表示异常的名字，异常通知方法的形参名字要与此处一致：

<?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"*>

<bean id=*"customerDao"* class=*"cn.itcast.dao.impl.CustomerDaoImpl"*></bean>

<!-- 配置切面类 -->

<bean id=*"myAspectXml"* class=*"cn.itcast.aspect.MyAspectXml"*></bean>

<aop:config>

<!-- 配置切入点表达式：告诉spring框架哪些方法需要被增强 -->

<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))"* id=*"pointcut1"*/>

<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.delete(..))"* id=*"pointcut2"*/>

<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.update(..))"* id=*"pointcut3"*/>

<aop:pointcut expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.list(..))"* id=*"pointcut4"*/>

<aop:aspect ref=*"myAspectXml"*>

<!-- 配置前置通知：告诉spring在目标方法执行之前增强 -->

<aop:before method=*"before"* pointcut-ref=*"pointcut1"*/>

<!-- 配置后置通知 -->

<aop:after-returning method=*"afterReturning"* pointcut-ref=*"pointcut2"* returning=*"result"*/>

<!-- 配置环绕通知 -->

<aop:around method=*"around"* pointcut-ref=*"pointcut3"*/>

<!-- 配置异常通知 -->

<aop:after-throwing method=*"afterThrowing"* pointcut-ref=*"pointcut4"* throwing=*"ex"*/>

</aop:aspect>

</aop:config>

</beans>

* 修改TestAOP中的test1方法：

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestAOP {

@Autowired

**private** CustomerDao customerDao;

@Test

**public** **void** test1(){

// customerDao.save();

// customerDao.delete();

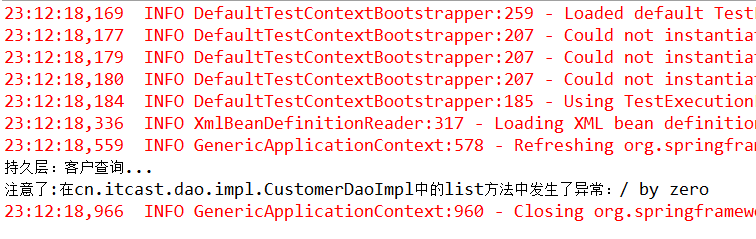
// customerDao.update();

customerDao.list();

}

}

* 测试结果如下：



### 最终通知

作用：不管目标方法是否发生异常，最终通知都会执行（类似于finally代码功能）

应用场景：释放资源 （关闭文件、 关闭数据库连接、 网络连接、 释放内存对象 ）

* 在切面类MyAspect中定义一个最终通知方法after方法

/\*\*

\* 切面类

\*

\* **@author** xiaokaibo

\*

\*/

**public** **class** MyAspectXml {

/\*\*

\* 前置通知方法 应用场景： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志 joinPoint:连接点，指的是被增强的那个方法

\*/

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

String username = "rose";

**if** (!"admin".equals(username)) {

// 非admin用户，不具备权限，抛出异常

// joinPoint.getTarget().getClass().getName()获取目标类的名字

// joinPoint.getSignature().getName()获取被增强方法的名字

**throw** **new** RuntimeException("对不起！您没有对" + joinPoint.getTarget().getClass().getName() + "类中"

+ joinPoint.getSignature().getName() + "方法的访问权限");

}

}

/\*\*

\* 后置通知方法 应用场景： ATM取款机取款后，自动下发短信 参数result:被增强那个方法的返回值

\*/

**public** **void** afterReturning(JoinPoint joinPoint, Object result) {

Date date = **new** Date();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateStr = sdf.format(date);

System.***out***.println("尊敬的用户，您于" + dateStr + "取款" + result + "元");

}

/\*\*

\* 环绕通知方法 应用场景：事务处理

\*

\* **@param** proceedingJoinPoint

\* 正在执行的连接点

\* **@return**

\*/

**public** Object around(ProceedingJoinPoint proceedingJoinPoint) {

System.***out***.println("开启事务");

// 获取目标方法的参数

Object[] args = proceedingJoinPoint.getArgs();

Object result = **null**;

**try** {

// 调用目标方法，获取目标方法的返回值

result = proceedingJoinPoint.proceed(args);

System.***out***.println("提交事务");

} **catch** (Throwable e) {

System.***out***.println("回滚事务");

}

// 返回目标方法的返回值

**return** result;

}

/\*\*

\* 异常通知方法 应用场景：处理异常

\*

\* **@param** ex

\* 目标方法抛出的异常

\*/

**public** **void** afterThrowing(JoinPoint joinPoint, Throwable ex) {

System.***out***.println("注意了:在" + joinPoint.getTarget().getClass().getName() + "中的"

+ joinPoint.getSignature().getName() + "方法中发生了异常：" + ex.getMessage());

}

/\*\*

\* 最终通知方法 应用场景：释放资源 （关闭文件、 关闭数据库连接、 网络连接、 释放内存对象 ）

\*

\* **@param** joinPoint

\* 被增强的那个方法

\*/

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

System.***out***.println("开始释放资源，对应的连接点信息为：" + joinPoint.getTarget().getClass().getName() + "的"

+ joinPoint.getSignature().getName() + "方法");

}

}

* 在applicationContext.xml中配置最终通知

<?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"*>

<bean id=*"customerDao"* class=*"cn.itcast.dao.impl.CustomerDaoImpl"*></bean>

<bean id=*"myAspectXml"* class=*"cn.itcast.aspect.MyAspectXml"*></bean>

<!-- AOP配置 -->

<aop:config>

<!-- 配置切入点 -->

<aop:pointcut

expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))"*

id=*"pointcut1"* />

<aop:pointcut

expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.delete(..))"*

id=*"pointcut2"* />

<aop:pointcut

expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.update(..))"*

id=*"pointcut3"* />

<aop:pointcut

expression=*"execution(\* cn.itcast.dao.impl.CustomerDaoImpl.list(..))"*

id=*"pointcut4"* />

<!-- 配置切面 -->

<aop:aspect ref=*"myAspectXml"*>

<!-- 前置通知 -->

<aop:before method=*"checkPrivilege"* pointcut-ref=*"pointcut1"* />

<!-- 后置通知 -->

<aop:after-returning method=*"afterReturning"*

pointcut-ref=*"pointcut2"* returning=*"result"* />

<!-- 环绕通知 -->

<aop:around method=*"around"* pointcut-ref=*"pointcut3"* />

<!-- 抛出异常通知 -->

<aop:after-throwing method=*"afterThrowing"*

pointcut-ref=*"pointcut4"* throwing=*"ex"* />

<!-- 最终通知 -->

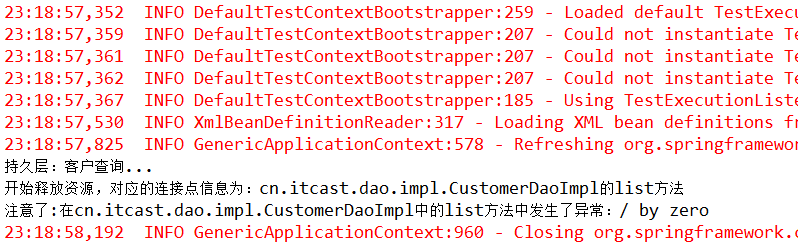
<aop:after method=*"after"* pointcut-ref=*"pointcut4"* />

</aop:aspect>

</aop:config>

</beans>

再次运行TestAOP中的test1方法，结果如下：



* 把CustomerDaoImpl中的list方法中的异常注释掉，再次运行TestAOP中的test1方法：

**public** **class** CustomerDaoImpl **implements** CustomerDao {

@Override

**public** **void** save() {

System.***out***.println("持久层：客户保存...");

}

@Override

**public** Integer delete() {

System.***out***.println("持久层：客户删除...");

**return** 100;

}

@Override

**public** **void** update() {

System.***out***.println("持久层：客户更新...");

}

@Override

**public** **void** list() {

System.***out***.println("持久层：客户查询...");

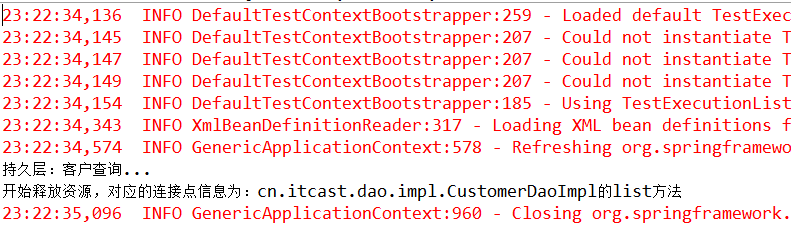
//自己造一个异常

//int i = 10 / 0;

}

}

测试结果如下：发现没有执行异常通知，但是依然执行了最终通知。

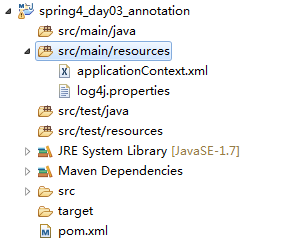


**注意，最终通知和后置通知的区别：最终通知，不管异常与否，都执行；而后置通知在异常时不执行。**

# Spring中AOP编程（注解方式）

## 基于注解的Spring AOP开发入门

### 创建工程，引入依赖，创建核心配置文件



pom.xml内容如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day03\_annotation</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

</dependency>

</dependencies>

</project>

在applicationContext.xml中开启注解扫描、开启aspectj自动代理

<?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"*>

<!-- 开启spring注解扫描 -->

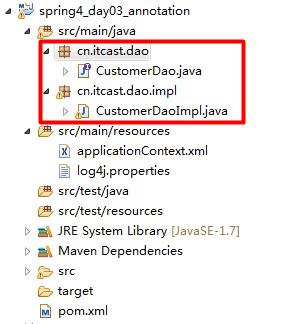
**<context:component-scan base-package=*"cn.itcast"*></context:component-scan>**

<!-- 开启aspectj自动代理 -->

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

</beans>

### 创建接口和实现类



**public** **interface** CustomerDao {

**public** **void** save();

**public** Integer delete();

**public** **void** update();

**public** **void** list();

}

**@Repository("customerDao")**

**public** **class** CustomerDaoImpl **implements** CustomerDao {

@Override

**public** **void** save() {

System.***out***.println("持久层：客户保存...");

}

@Override

**public** Integer delete() {

System.***out***.println("持久层：客户删除...");

**return** 100;

}

@Override

**public** **void** update() {

System.***out***.println("持久层：客户更新...");

}

@Override

**public** **void** list() {

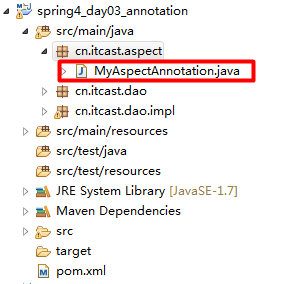
System.***out***.println("持久层：客户查询...");

**int** i = 10 / 0;

}

}

### 编写切面类



在MyAspectAnnotation类上添加@Component注解，把该类配置在Spring中；同时，还得添加@Aspect注解，表示该类是一个切面类；

在checkPrivilege方法上添加@Before注解，并指定切入点表达式

/\*\*

\* 切面类

\* **@author** kevin

\*/

@Component

**@Aspect**//表示该类是一个切面类

**public** **class** MyAspectAnnotation {

@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

**public** **void** writeLog(){

System.***out***.println("记录日志啦.....");

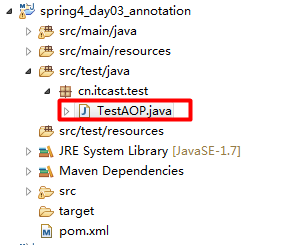
}

}

**提示：此处的切面类可以不取id.**

### 编写测试类

创建TestAOP测试类，在其中创建test1方法



@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestAOP {

@Autowired

**private** CustomerDao customerDao;

@Test

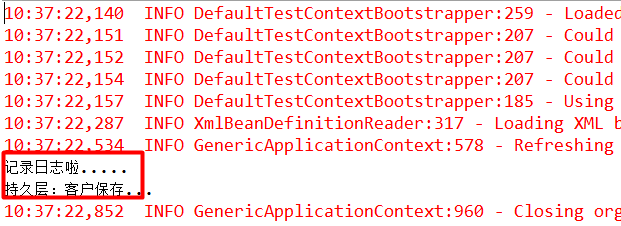
**public** **void** test1(){

customerDao.save();

}

}

观察结果，发现save方法被增强了：



## 用注解来配置Spring AOP中的各种通知

### 前置通知

/\*\*

\* 切面类

\* **@author** xiaokaibo

\*

\*/

@Component

@Aspect//表示该类是一个切面类

**public** **class** MyAspectAnnotation {

/\*@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

public void writeLog(){

System.out.println("记录日志啦.....");

}\*/

/\*\*

\* 前置通知方法 应用场景： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志 joinPoint:连接点，指的是被增强的那个方法

\*/

@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

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

String username = "rose";

**if** (!"admin".equals(username)) {

// 非admin用户，不具备权限，抛出异常

// joinPoint.getTarget().getClass().getName()获取目标类的名字

// joinPoint.getSignature().getName()获取被增强方法的名字

**throw** **new** RuntimeException("对不起！您没有对" + joinPoint.getTarget().getClass().getName() + "类中"

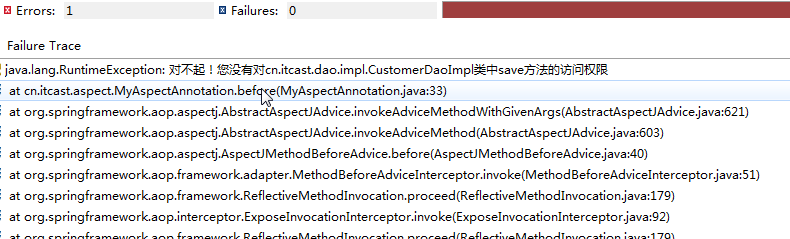
+ joinPoint.getSignature().getName() + "方法的访问权限");

}

}

}

运行TestAOP中的test1方法，测试结果如下（同xml配置的效果一下）：



### 后置通知

* 修改MyAspectAnnotation，在其中创建后置通知方法afterReturning，在该方法上添加@AfterReturning注解，通过value属性指定切入点表达式，returning属性指定返回值的名字，该方法形参的名字要与returning的值一致：

/\*\*

\* 切面类

\* **@author** xiaokaibo

\*

\*/

@Component

@Aspect//表示该类是一个切面类

**public** **class** MyAspectAnnotation {

/\*@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

public void writeLog(){

System.out.println("记录日志啦.....");

}\*/

/\*\*

\* 前置通知方法 应用场景： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志 joinPoint:连接点，指的是被增强的那个方法

\*/

@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

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

String username = "rose";

**if** (!"admin".equals(username)) {

// 非admin用户，不具备权限，抛出异常

// joinPoint.getTarget().getClass().getName()获取目标类的名字

// joinPoint.getSignature().getName()获取被增强方法的名字

**throw** **new** RuntimeException("对不起！您没有对" + joinPoint.getTarget().getClass().getName() + "类中"

+ joinPoint.getSignature().getName() + "方法的访问权限");

}

}

/\*\*

\* 后置通知方法 应用场景： ATM取款机取款后，自动下发短信 参数result:被增强那个方法的返回值

\* returning属性指定目标方法返回值的名字

\*/

@AfterReturning(value="execution(\* cn.itcast.dao.impl.CustomerDaoImpl.delete(..))",returning="result")

**public** **void** afterReturning(JoinPoint joinPoint, Object result) {

Date date = **new** Date();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateStr = sdf.format(date);

System.***out***.println("尊敬的用户，您于" + dateStr + "取款" + result + "元");

}

}

* 修改TestAOP中的test1方法：

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestAOP {

@Autowired

**private** CustomerDao customerDao;

@Test

**public** **void** test1(){

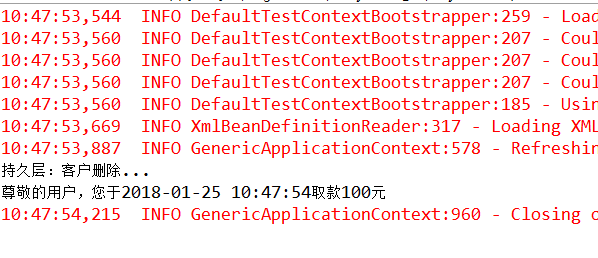
// customerDao.save();

customerDao.delete();

}

}

运行结果如下：



### 环绕通知

* 修改MyAspectAnnotation，在其中创建环绕通知方法around，在该方法上添加@Around注解，通过value属性指定切入点表达式:

/\*\*

\* 切面类

\* **@author** xiaokaibo

\*

\*/

@Component

@Aspect//表示该类是一个切面类

**public** **class** MyAspectAnnotation {

/\*@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

public void writeLog(){

System.out.println("记录日志啦.....");

}\*/

/\*\*

\* 前置通知方法 应用场景： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志 joinPoint:连接点，指的是被增强的那个方法

\*/

@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

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

String username = "rose";

**if** (!"admin".equals(username)) {

// 非admin用户，不具备权限，抛出异常

// joinPoint.getTarget().getClass().getName()获取目标类的名字

// joinPoint.getSignature().getName()获取被增强方法的名字

**throw** **new** RuntimeException("对不起！您没有对" + joinPoint.getTarget().getClass().getName() + "类中"

+ joinPoint.getSignature().getName() + "方法的访问权限");

}

}

/\*\*

\* 后置通知方法 应用场景： ATM取款机取款后，自动下发短信 参数result:被增强那个方法的返回值

\* returning属性指定目标方法返回值的名字

\*/

@AfterReturning(value="execution(\* cn.itcast.dao.impl.CustomerDaoImpl.delete(..))",returning="result")

**public** **void** afterReturning(JoinPoint joinPoint, Object result) {

Date date = **new** Date();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateStr = sdf.format(date);

System.***out***.println("尊敬的用户，您于" + dateStr + "取款" + result + "元");

}

/\*\*

\* 环绕通知方法 应用场景：事务处理

\*

\* **@param** proceedingJoinPoint

\* 正在执行的连接点

\* **@return**

\*/

@Around("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.update(..))")

**public** Object around(ProceedingJoinPoint proceedingJoinPoint) {

System.***out***.println("开启事务");

// 获取目标方法的参数

Object[] args = proceedingJoinPoint.getArgs();

Object result = **null**;

**try** {

// 调用目标方法，获取目标方法的返回值

result = proceedingJoinPoint.proceed(args);

System.***out***.println("提交事务");

} **catch** (Throwable e) {

System.***out***.println("回滚事务");

}

// 返回目标方法的返回值

**return** result;

}

}

* 修改TestAOP中的test1方法：

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestAOP {

@Autowired

**private** CustomerDao customerDao;

@Test

**public** **void** test1(){

//customerDao.save();

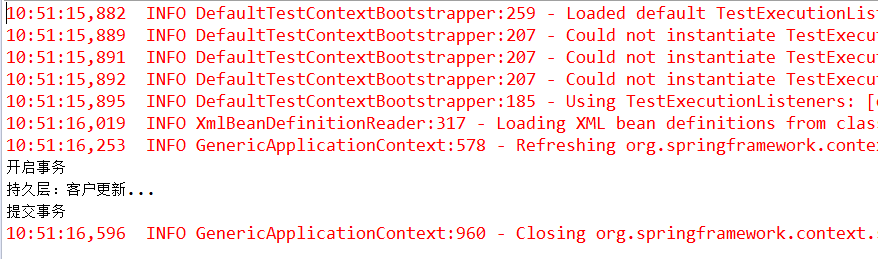
// customerDao.delete();

customerDao.update();

}

}

运行结果如下：



### 异常通知

* 修改MyAspectAnnotation，在其中创建异常通知方法afterThrowing，在该方法上添加@AfterThrowing注解，通过value属性指定切入点表达式,通过throwing属性指定异常的名字。在该方法中增加一个形参，表示发生的异常，形参的名字与throwing属性的值一致：

/\*\*

\* 切面类

\* **@author** xiaokaibo

\*

\*/

@Component

@Aspect//表示该类是一个切面类

**public** **class** MyAspectAnnotation {

/\*@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

public void writeLog(){

System.out.println("记录日志啦.....");

}\*/

/\*\*

\* 前置通知方法 应用场景： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志 joinPoint:连接点，指的是被增强的那个方法

\*/

@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

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

String username = "rose";

**if** (!"admin".equals(username)) {

// 非admin用户，不具备权限，抛出异常

// joinPoint.getTarget().getClass().getName()获取目标类的名字

// joinPoint.getSignature().getName()获取被增强方法的名字

**throw** **new** RuntimeException("对不起！您没有对" + joinPoint.getTarget().getClass().getName() + "类中"

+ joinPoint.getSignature().getName() + "方法的访问权限");

}

}

/\*\*

\* 后置通知方法 应用场景： ATM取款机取款后，自动下发短信 参数result:被增强那个方法的返回值

\* returning属性指定目标方法返回值的名字

\*/

@AfterReturning(value="execution(\* cn.itcast.dao.impl.CustomerDaoImpl.delete(..))",returning="result")

**public** **void** afterReturning(JoinPoint joinPoint, Object result) {

Date date = **new** Date();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateStr = sdf.format(date);

System.***out***.println("尊敬的用户，您于" + dateStr + "取款" + result + "元");

}

/\*\*

\* 环绕通知方法 应用场景：事务处理

\*

\* **@param** proceedingJoinPoint

\* 正在执行的连接点

\* **@return**

\*/

@Around("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.update(..))")

**public** Object around(ProceedingJoinPoint proceedingJoinPoint) {

System.***out***.println("开启事务");

// 获取目标方法的参数

Object[] args = proceedingJoinPoint.getArgs();

Object result = **null**;

**try** {

// 调用目标方法，获取目标方法的返回值

result = proceedingJoinPoint.proceed(args);

System.***out***.println("提交事务");

} **catch** (Throwable e) {

System.***out***.println("回滚事务");

}

// 返回目标方法的返回值

**return** result;

}

/\*\*

\* 异常通知方法 应用场景：处理异常

\*

\* **@param** ex

\* 目标方法抛出的异常

\*/

@AfterThrowing(value="execution(\* cn.itcast.dao.impl.CustomerDaoImpl.list(..))",throwing="ex")

**public** **void** afterThrowing(JoinPoint joinPoint, Throwable ex) {

System.***out***.println("注意了:在" + joinPoint.getTarget().getClass().getName() + "中的"

+ joinPoint.getSignature().getName() + "方法中发生了异常：" + ex.getMessage());

}

}

* 修改TestAOP中的test1方法：

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestAOP {

@Autowired

**private** CustomerDao customerDao;

@Test

**public** **void** test1(){

// customerDao.save();

// customerDao.delete();

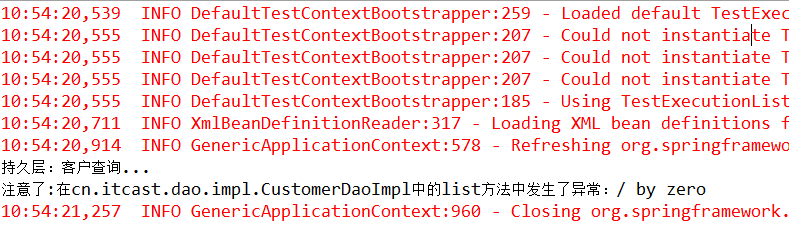
// customerDao.update();

customerDao.list();

}

}

运行结果如下：



### 最终通知

* 修改MyAspectAnnotation，在其中创建最终通知方法after，在该方法上添加@After注解，通过value属性指定切入点表达式：

/\*\*

\* 切面类

\* **@author** xiaokaibo

\*

\*/

@Component

@Aspect//表示该类是一个切面类

**public** **class** MyAspectAnnotation {

/\*@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

public void writeLog(){

System.out.println("记录日志啦.....");

}\*/

/\*\*

\* 前置通知方法 应用场景： 权限控制 （权限不足，抛出异常）、 记录方法调用信息日志 joinPoint:连接点，指的是被增强的那个方法

\*/

@Before("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.save(..))")

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

String username = "rose";

**if** (!"admin".equals(username)) {

// 非admin用户，不具备权限，抛出异常

// joinPoint.getTarget().getClass().getName()获取目标类的名字

// joinPoint.getSignature().getName()获取被增强方法的名字

**throw** **new** RuntimeException("对不起！您没有对" + joinPoint.getTarget().getClass().getName() + "类中"

+ joinPoint.getSignature().getName() + "方法的访问权限");

}

}

/\*\*

\* 后置通知方法 应用场景： ATM取款机取款后，自动下发短信 参数result:被增强那个方法的返回值

\* returning属性指定目标方法返回值的名字

\*/

@AfterReturning(value="execution(\* cn.itcast.dao.impl.CustomerDaoImpl.delete(..))",returning="result")

**public** **void** afterReturning(JoinPoint joinPoint, Object result) {

Date date = **new** Date();

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

String dateStr = sdf.format(date);

System.***out***.println("尊敬的用户，您于" + dateStr + "取款" + result + "元");

}

/\*\*

\* 环绕通知方法 应用场景：事务处理

\*

\* **@param** proceedingJoinPoint

\* 正在执行的连接点

\* **@return**

\*/

@Around("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.update(..))")

**public** Object around(ProceedingJoinPoint proceedingJoinPoint) {

System.***out***.println("开启事务");

// 获取目标方法的参数

Object[] args = proceedingJoinPoint.getArgs();

Object result = **null**;

**try** {

// 调用目标方法，获取目标方法的返回值

result = proceedingJoinPoint.proceed(args);

System.***out***.println("提交事务");

} **catch** (Throwable e) {

System.***out***.println("回滚事务");

}

// 返回目标方法的返回值

**return** result;

}

/\*\*

\* 异常通知方法 应用场景：处理异常

\*

\* **@param** ex

\* 目标方法抛出的异常

\*/

@AfterThrowing(value="execution(\* cn.itcast.dao.impl.CustomerDaoImpl.list(..))",throwing="ex")

**public** **void** afterThrowing(JoinPoint joinPoint, Throwable ex) {

System.***out***.println("注意了:在" + joinPoint.getTarget().getClass().getName() + "中的"

+ joinPoint.getSignature().getName() + "方法中发生了异常：" + ex.getMessage());

}

/\*\*

\* 后置通知方法 应用场景：释放资源 （关闭文件、 关闭数据库连接、 网络连接、 释放内存对象 ）

\*

\* **@param** joinPoint

\* 被增强的那个方法

\*/

@After("execution(\* cn.itcast.dao.impl.CustomerDaoImpl.list(..))")

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

System.***out***.println("开始释放资源，对应的连接点信息为：" + joinPoint.getTarget().getClass().getName() + "的"

+ joinPoint.getSignature().getName() + "方法");

}

}

再次运行TestAOP中的test1方法，结果如下：

