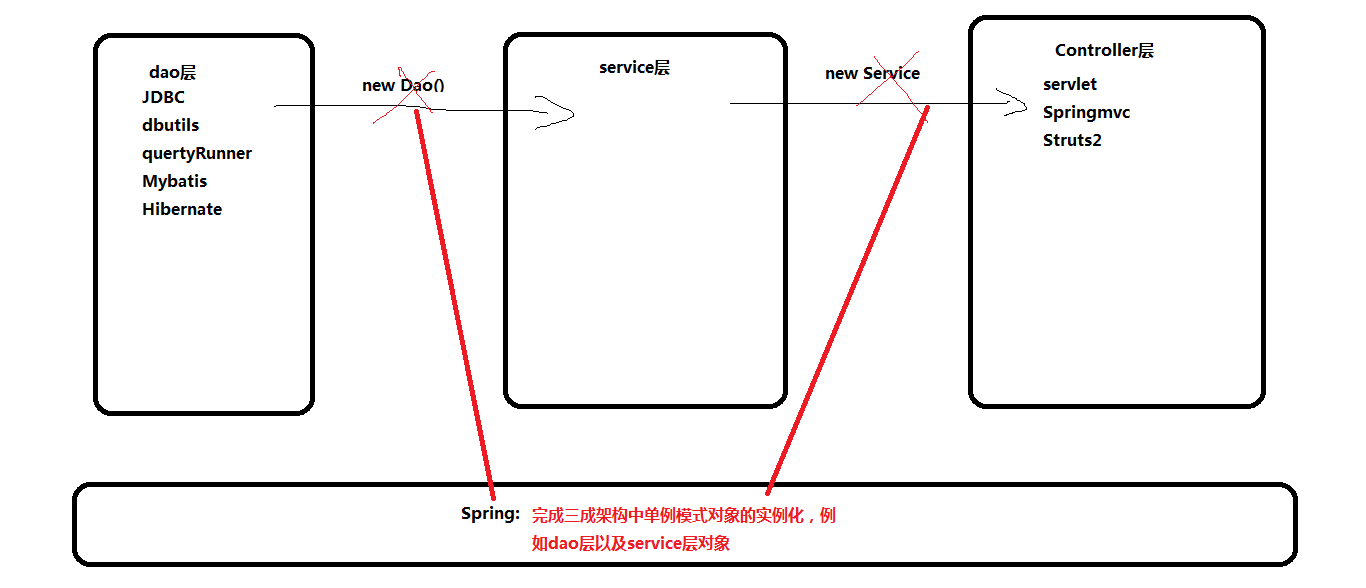
## 内容

1. Spring介绍
2. Spring框架编写入门程序
3. Spring配置文件中bean节点属性说明
4. Spring的核心思想IOC,DI
5. Spring 中操作JDBC的插件JdbcTemplate

## Spring介绍

1. 开源的一站式的开发框架。



1. 特点：

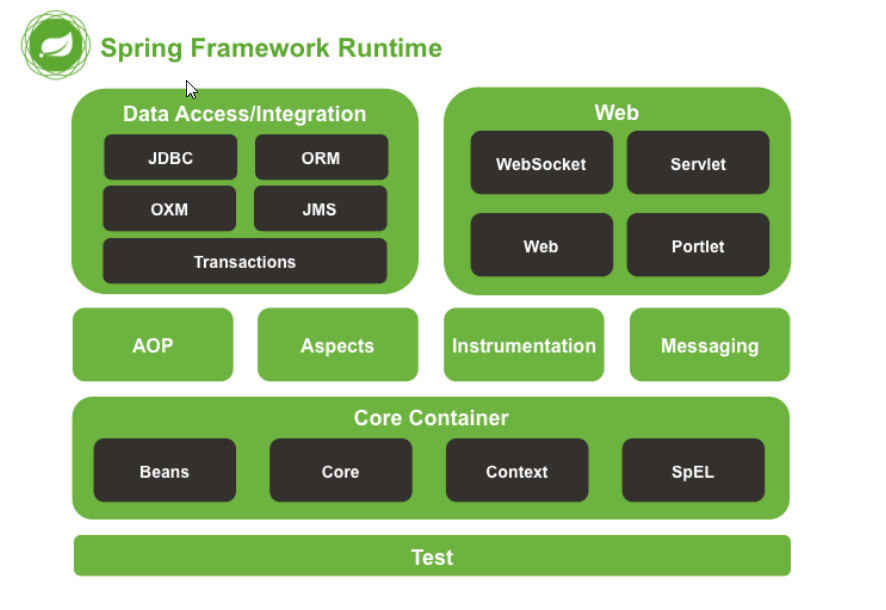
1.可以达到解耦的效果，降低三层架构直接的耦合度。

2.支持AOP编程，实现事务的声明式管理。

3.可以兼容集成其他优秀框架。

4.支持springboot,springclound等微服务开发。

3. Spring架构



## Spring框架编写入门程序

1. 创建项目加入依赖包

commons-logging-1.2.jar

spring-beans-4.3.6.RELEASE.jar

spring-context-4.3.6.RELEASE.jar

spring-core-4.3.6.RELEASE.jar

spring-expression-4.3.6.RELEASE.jar

1. 编写配置文件：可以任意命名，beas.xml,appliactionContext.xml

创建类UserDaoImpl

|  |
| --- |
| **public** **class** UserDaoImpl {  **public** **void** save() {  System.***out***.println("保存用户");  }  } |

在Spring配置文件中实例化类

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd"*>  <bean id=*"ud"* class=*"com.offcn.dao.UserDaoImpl"*></bean>      </beans> |

1. 解析配置文件

|  |
| --- |
| **public** **class** App {  **public** **static** **void** main(String[] args) {  //配置文件的解析  ApplicationContext ap=**new** ClassPathXmlApplicationContext("beans.xml");  UserDaoImpl udao=(UserDaoImpl) ap.getBean("ud");  udao.save();  }  } |

## Spring配置文件中bean节点属性说明

<!-- id:唯一

name:作用和id一样的，但是可以为一个节点命名多个名字

class：要实例化的对象的类型

lazy-init="true":设置延时加载实例化对象,针对某一个bean节点

default-lazy-init="true"：针对全部bean节点

scope="prototype":以多例（原型）模式实例化对象

init-method:初始化bean节点后执行的方法

destroy-method：销毁bean节点时执行的方法

-->

解析beans.xml的三种方式

1、配置文件相对src的路径

ApplicationContext ap=new ClassPathXmlApplicationContext("beans.xml");

2.配置文件在硬盘中的绝对路径

ApplicationContext ap=new FileSystemXmlApplicationContext("D:\\myworksp\\spring001\\src\\beans.xml");

3.解析配置文件通过绝对路径，默认实例化对象是延时的

BeanFactory ap=new XmlBeanFactory(new FileSystemResource("D:\\myworksp\\spring001\\src\\beans.xml"));

## Spring的核心思想IOC,DI

### IOC

（Inversion of Control）控制反转，应用程序本身不负责对象的创建（实例化），对象的实例化由Spring 容器（配置文件）完成。

Spring实例化对象常用的三种方式：

1. 接口实现类方式

定义接口和接口对应的实现类

配置：

<!--1.接口，实现类方式，class:实现类 -->

<bean id="pd" class="com.offcn.dao.PerDaoImpl"></bean>

1. 静态工厂模式

即通过工厂类里的静态方法得到需要的对象

创建一个工厂类，在工厂类创建一个静态方法得到需要的对象

配置：

<!--2.静态工厂模式 -->

<bean id="pd1" class="com.offcn.utils.PerDaoFactory" factory-method="getPserDao"></bean>

1. 工厂模式

即通过工厂类里的非静态方法得到需要的对象

<!-- 3.工厂模式 -->

<bean id="sf" class="com.offcn.utils.PerDaoFactory"></bean>

<bean id="pd2" factory-bean="sf" factory-method="getPserDao1"></bean>

### DI

(Dependency Injection)依赖注入,为Spring容器实例化对象的属性赋值的过程称为依赖注入的过程。通俗的说就是在spring实例化某个类时，为这个类依赖的属性赋值的过程称为依赖注入。

set注入

解析property,根据name的值找到对应的set方法，将对应的值赋值给set方法的形参。

案例：

|  |
| --- |
| **public** **class** PerServiceImpl **implements** PerService {  PerDao pd;  **int** pid;  String name;  List<Object> list;  Set<Object> set;  Map<String,Object> map;  Object[] array;    **public** **void** setPd(PerDao pd) {  **this**.pd = pd;  }  **public** **void** setPid(**int** pid) {  **this**.pid = pid;  }  **public** **void** setName(String name) {  **this**.name = name;  }    **public** **void** setList(List<Object> list) {  **this**.list = list;  }  **public** **void** setSet(Set<Object> set) {  **this**.set = set;  }  **public** **void** setMap(Map<String, Object> map) {  **this**.map = map;  }  **public** **void** setArray(Object[] array) {  **this**.array = array;  }  @Override  **public** **void** save() {  // **TODO** Auto-generated method stub  pd.save();  System.***out***.println("pid:"+pid+"----"+name);  **for**(Object o:list) {  System.***out***.println("list:"+o);  }    **for**(Object o:set) {  System.***out***.println("set:"+o);  }    **for**(Object o:array) {  System.***out***.println("array:"+o);  }    Set<String> keySet = map.keySet();  **for**(Object k:keySet) {  System.***out***.println("map key:"+k+"----"+map.get(k));  }      }  } |

在配置文件beans.xml中完成注入：

|  |
| --- |
| <bean id=*"ps"* class=*"com.offcn.service.PerServiceImpl"*>  <property name=*"pd"* ref=*"pd"*></property>  <property name=*"pid"* value=*"12"*></property>  <property name=*"name"* value=*"弟弟"*></property>  <property name=*"list"*>  <list>  <value>14</value>  <value>14</value>  <value>kl</value>  <!-- <ref bean="pd"/> -->  <bean class=*"com.offcn.dao.PerDaoImpl"*></bean>  </list>  </property>  <property name=*"map"*>  <map>  <entry key=*"map001"*>  <value>jlkjlk</value>  </entry>  <entry key=*"map002"*>  <ref bean=*"ud"*/>  </entry>  </map>  </property>  <property name=*"set"*>  <set>  <value>14</value>  <value>14</value>  <value>kl</value>  <!-- <ref bean="pd"/> -->  <bean class=*"com.offcn.dao.PerDaoImpl"*></bean>  </set>  </property>  <property name=*"array"*>  <array>  <value>14</value>  <value>14</value>  <value>kl</value>  <!-- <ref bean="pd"/> -->  <bean class=*"com.offcn.dao.PerDaoImpl"*></bean>  </array>  </property>  </bean> |

构造注入

通过有参构造方法，为类中所依赖的属性赋值。

按照参数的索引注入(index)

|  |
| --- |
| <bean id=*"ss"* class=*"com.offcn.service.StuServiceImpl"*>  <constructor-arg index="0">  <value>12</value>  </constructor-arg>  <constructor-arg index="1">  <value>小牧</value>  </constructor-arg>  </bean> |

按照参数的类型注入(type)

|  |
| --- |
| <bean id=*"ss"* class=*"com.offcn.service.StuServiceImpl"*>  <constructor-arg type="String">  <value>12</value>  </constructor-arg>  <constructor-arg type="String">  <value>小牧</value>  </constructor-arg>  </bean> |

按照参数的名称注入(name)

|  |
| --- |
| <bean id=*"ss"* class=*"com.offcn.service.StuServiceImpl"*>  <constructor-arg name=*"sid"* index=*"0"*>  <value>12</value>  </constructor-arg>  <constructor-arg name=*"uname"* index=*"1"*>  <value>小牧</value>  </constructor-arg>  </bean> |

注意：三种方式可以合起来一起使用，唯一确定一个有参构造方法。

**练习**

**Spring方式完成数据源的实例化**

**以c3po的连接池为例。**

|  |
| --- |
| <bean id=*"ds"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>  <property name=*"driverClass"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/hibdata"*></property>  <property name=*"user"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  </bean> |

**以dbcp连接池为例。**

|  |
| --- |
| <bean id=*"ds1"* class=*"org.apache.commons.dbcp.BasicDataSource"*>  <property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"url"* value=*"jdbc:mysql://localhost:3306/hibdata"*></property>  <property name=*"username"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  </bean> |

## Spring 中操作JDBC的插件JdbcTemplate

1. **加入插件的依赖包**

****

1. 定义实体类

在com.offcn.bean定义News实体类

1. **dao层中通过**JdbcTemplate完成CURD操作

**接口NewsDao**

|  |
| --- |
| **package** com.offcn.dao;  **import** java.util.List;  **import** com.offcn.bean.News;  **public** **interface** NewsDao {  **public** List<News> getAll();  **public** News getByNid(**int** nid);  **public** **int** save(News news);  **public** **int** update(News news);  **public** **int** del(**int** nid);  } |

**实现类NewsDaoImpl**

|  |
| --- |
| **package** com.offcn.dao;  **import** java.util.List;  **import** org.springframework.jdbc.core.BeanPropertyRowMapper;  **import** org.springframework.jdbc.core.JdbcTemplate;  **import** org.springframework.jdbc.core.RowMapper;  **import** com.offcn.bean.News;  **public** **class** NewsDaoImpl **implements** NewsDao {    JdbcTemplate jt;    **public** **void** setJt(JdbcTemplate jt) {  **this**.jt = jt;  }  @Override  **public** List<News> getAll() {  // **TODO** Auto-generated method stub  String sql="select \* from news";  RowMapper<News> rm=**new** BeanPropertyRowMapper<News>(News.**class**);  List<News> nlist = jt.query(sql, rm);  **return** nlist;  }  @Override  **public** News getByNid(**int** nid) {  // **TODO** Auto-generated method stub  String sql="select \* from news where nid="+nid;  RowMapper<News> rm=**new** BeanPropertyRowMapper<News>(News.**class**);  News news = jt.queryForObject(sql, rm);  **return** news;  }  @Override  **public** **int** save(News news) {  // **TODO** Auto-generated method stub  String sql="insert into news values(null,?,?,?)";  **return** jt.update(sql, news.getTitle(),news.getContent(),news.getPhoto());  }  @Override  **public** **int** update(News news) {  // **TODO** Auto-generated method stub  String sql="update news set title=?,content=? where nid=?";  **return** jt.update(sql, news.getTitle(),news.getContent(),news.getNid());  }  @Override  **public** **int** del(**int** nid) {  // **TODO** Auto-generated method stub  **return** jt.update("delete from news where nid="+nid);  }  } |

1. 在配置文件beans.xml中实例化dao层实现类，JdbcTemplate,连接池

|  |
| --- |
| <!-- 实例化dbcp连接池 -->  <bean id=*"ds1"* class=*"org.apache.commons.dbcp.BasicDataSource"*>  <property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>  <property name=*"url"* value=*"jdbc:mysql://localhost:3306/hibdata"*></property>  <property name=*"username"* value=*"root"*></property>  <property name=*"password"* value=*"root"*></property>  </bean>    <!-- 实例化JdbcTemplate-->  <bean id=*"jdtm"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>  <!-- 依赖连接池 -->  <property name=*"dataSource"* ref=*"ds1"*></property>  </bean>    <!-- 实现dao层实现类NewsDaoImpl -->  <bean id=*"nd"* class=*"com.offcn.dao.NewsDaoImpl"*>  <!--依赖Jdbctemplate -->  <property name=*"jt"* ref=*"jdtm"*></property>  </bean> |

1. 测试

|  |
| --- |
| **public** **class** App2 {  **public** **static** **void** main(String[] args) **throws** IllegalAccessException, IllegalArgumentException, InvocationTargetException, NoSuchMethodException, SecurityException {  ApplicationContext ap=**new** ClassPathXmlApplicationContext("beans.xml");  NewsDao nd=(NewsDao) ap.getBean("nd");    //查询所有数据  // List<News> nlist = nd.getAll();  // System.out.println(nlist);    //根据逐渐查询  // News news = nd.getByNid(4);  // System.out.println(news);    //保存  // News news=new News();  // news.setTitle("第三方");  // news.setContent("取消时序");  // news.setPhoto("no.gif");  // int row = nd.save(news);  // System.out.println(row);    //更新  // News news=new News();  // news.setNid(17);  // news.setTitle("第三方wdd");  // news.setContent("取消时序ddd");  // news.setPhoto("no.gif");  // int row = nd.update(news);  // System.out.println(row);      //删除  **int** row = nd.del(17);  System.***out***.println(row);    }  } |

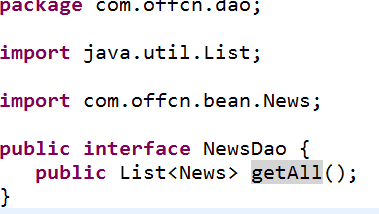
## 一站式开发中常用的注解

### XML方式一站式开发案例

**1.实体类News**,省略查看源码

**2.dao层NewsDao**

接口

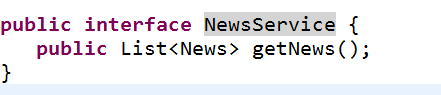


实现类

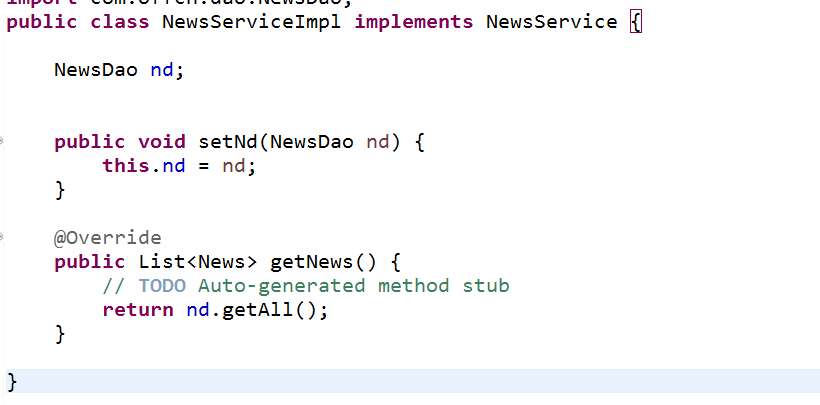


1. **Service层**

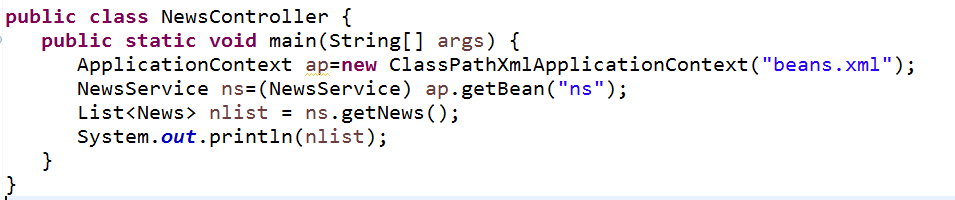
接口



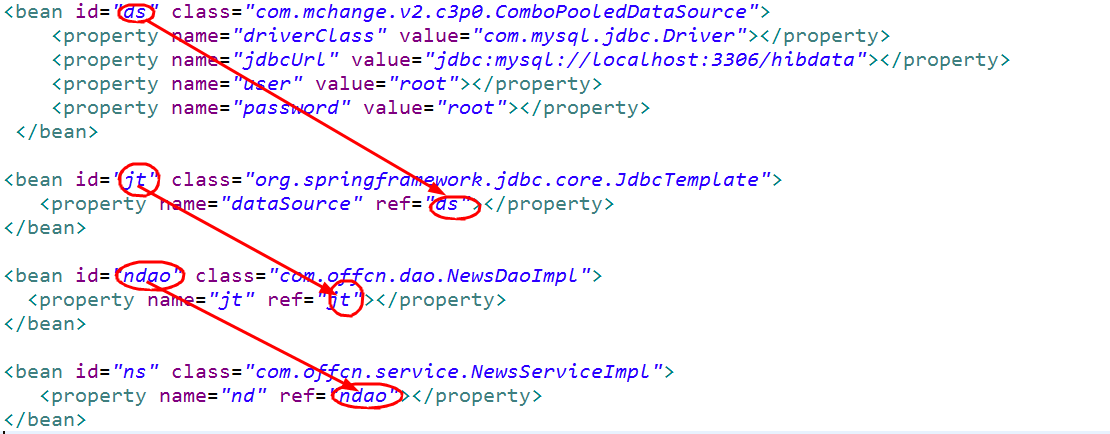
实现类



1. **控制层**



1. **配置文件**

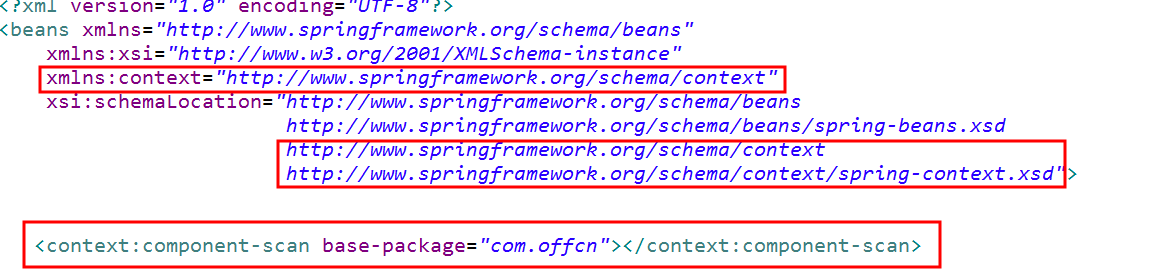


### 注解方式一站式开发案例

1. @Repository：作用在dao层的实现类上用来实例化dao层的对象。
2. @Service:作用在service层的实现类上用来实例化service层的对象。
3. @AutoWired/@Resource:作用在依赖的属性上用来给属性赋值。
4. @Controller，@Action:作用在控制层用来实例化控制层对象。
5. @Component:作用在三层架构以外其他类上用来实例化这些类。

注解：

1. 开启注解扫描

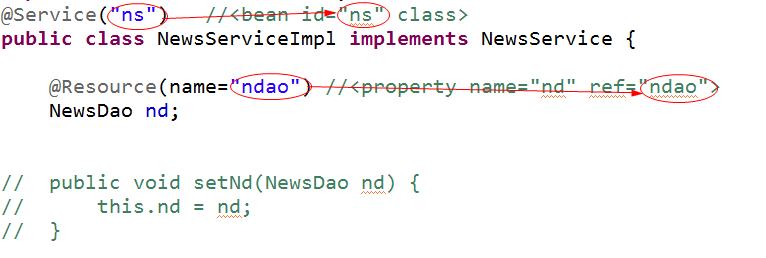


1. 加入依赖包spring-aop.jar
2. 在dao层，service层实现类上以及所依赖的属性上分别加注解。

dao层



Service层



1. 测试

|  |
| --- |
| **public** **class** NewsController {  **public** **static** **void** main(String[] args) {  ApplicationContext ap=**new** ClassPathXmlApplicationContext("beans.xml");  NewsService ns=(NewsService) ap.getBean("ns");  List<News> nlist = ns.getNews();  System.***out***.println(nlist);  }  } |

**细节**：

@Resource注解：

指定名称：先按照当前属性的名称name找到对应的bean节点，赋值给当前属性;

若没有找到对应的节点会按照属性的类型查找对应的bean节点，赋值给当前属性

没有指定名称：先按照当前属性的默认名称（属性类型的首字母小写）找到对应的bean节点，赋值给当前属性;若没有找到对应的节点会按照属性的类型查找对应的bean节点，赋值给当前属性。

@AutoWired注解：

按照当前属性（jt）的类型（JdbcTemplate）查找到系统中该类型的bean节点，赋值给当前属性。

**所以属性类型对应的bean节点只有一个，最好按照类型注入，否则按照名称注入**。

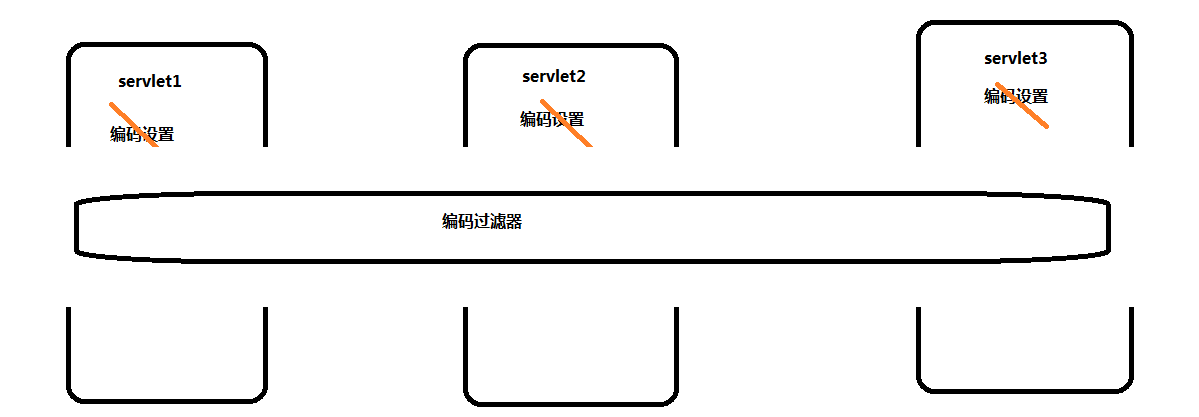
@Repository和@Service注解的value可以省略不写，默认value值为类名的首字母小写。

## AOP的介绍以及动态代理

### AOP介绍

AOP：（Aspect Oriented Programming）面向切面编程。通过[预编译](https://baike.baidu.com/item/%E9%A2%84%E7%BC%96%E8%AF%91/3191547)方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是[OOP](https://baike.baidu.com/item/OOP)的延续，是软件开发中的一个热点，也是[Spring](https://baike.baidu.com/item/Spring)框架中的一个重要内容，是[函数式编程](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0%E5%BC%8F%E7%BC%96%E7%A8%8B/4035031)的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的[耦合度](https://baike.baidu.com/item/%E8%80%A6%E5%90%88%E5%BA%A6/2603938)降低，提高程序的可重用性，同时提高了开发的效率。

1. 依赖动态代理实现功能的统一维护；
2. AOP是面向的对象编程的补充，将系统中和主要业务无关的代码提取出来封装到一个类中（代码块中），将封装后的类或者代码块成为切面。
3. 纵向重复，横向抽取。
4. 降低主要业务和辅助业务的耦合度，同时可以提高代码的重用性。



动态代理

**代理**：一个机构或者人代理另外一个人或者另外一个机构完成一部分工作。一个方法或者类代理另外一个方法或者另外一个类完成一部分其他辅助业务，降低主要业务和辅助业务的耦合度，同时通过辅助业务可以对主要业务增强。

**静态代理**：为每一个目标类（要被代理的类）创建一个代理类，静态代理中代理类在编译期就已经确定，静态代理的效率相对动态代理来说相对高一些，但是静态代理代码冗余大。

**动态代理**：根据不同的目标类（要被代理的类）通过反射在内存中动态生成一个代理类，动态代理则是JVM运行时动态生成，可以最大限度的降低代码的冗余。

常用的动态代理有两种JDK代理和CGLib代理。

### JDK代理

代理的目标类实现了接口（代理类和目标类实现相同的接口）

|  |
| --- |
| **public** **class** JDKProxy {  /\*  \* 为目标类生产代理类的方法  \*/  **public** Object getProxy(Object tarObj) {  /\*  \* 参数一：目标类的类加载器  \* 参数二：目标类实现的接口  \*参数三：InvocationHandler: 代理类对象执行方法时内存中执行的内容  \*/  **return** Proxy.*newProxyInstance*(tarObj.getClass().getClassLoader(), tarObj.getClass().getInterfaces(), **new** InvocationHandler() {    /\*  \* 参数一：代理类对象  \* 参数二：目标类的方法Method对象  \*参数三：目标类的方法参数  \*/  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  // **TODO** Auto-generated method stub  //执行目标类的主要业务  Object res = method.invoke(tarObj, args);  //辅助业务  System.***out***.println("提交事务");  **return** res;  }  });  }    } |

### CGLib代理

对代理目标类可以实现接口（代理类和目标类实现相同的接口）也可以是没有实现接口的目标类（代理类继承目标类）。

加入依赖包：



|  |
| --- |
| **public** **class** CGLibProxy {  /\*  \* 为目标类生产代理类的方法  \*/  **public** Object getProxy(Object tarObj) {  Enhancer en=**new** Enhancer();  en.setClassLoader(tarObj.getClass().getClassLoader());//目标类的类加载器  //en.setInterfaces(tarObj.getClass().getInterfaces());//目标类实现的接口  en.setSuperclass(tarObj.getClass());  en.setCallback(**new** MethodInterceptor() {    @Override  **public** Object intercept(Object proxy, Method method, Object[] args, MethodProxy arg3) **throws** Throwable {  // **TODO** Auto-generated method stub  //执行目标类的主要业务  Object res = method.invoke(tarObj, args);  //辅助业务  System.***out***.println("提交事务");  **return** res;  }  });//代理类对象执行方法时内存中执行的内容  **return** en.create();  }    } |

## SpringAOP配置

1. 连接点：需要增强的方法
2. 切点：方法被增强以后成为切点。
3. 通知：增强的内容成为通知。
4. 切面：把通知作用在切点上形成切面（通知+切点）。
5. 织入：把通知作用在切点上过程称为织入的过程。
6. 目标类：要被代理（增强）的类
7. 代理类：含有增强的内容的类

### 以XML方式完成SpringAOP的配置

SpringAOP配置步骤：

1. 依赖包



1. 准备目标类(UserService,UserServiceImpl)
2. 准备代理类以及增强的内容(辅助业务)
3. 织入的完成：把代理类中的辅助业务作用在目标类中方法上

在Spring的配置文件中配置如下



指定切点的语法

expression=execution(访问修饰 返回值类型 方法名（参数）)

访问修饰:（非必须）,可以省略不写支持任何访问修饰符

返回值类型：\*，任意返回值类型

方法名：包名(\*任意一级包,..包中所有内容).类名(\*).方法名（\*）

参数：（..任意参数）

案例：

execution( \* com.offcn.service2.\*.\*(..))：com.offcn.service2包中的任意类任意方法

execution( \* com.offcn.service2..\*(..))：com.offcn.service2包中的任意类或者任意包中的任意方法。

通知：

前置通知：辅助业务在主要业务之前执行。

后置通知：辅助业务在主要业务之后执行，当主要业务中出现异常，辅助业务不执行,在辅助业务中可以得到主要业务执行后的返回值。

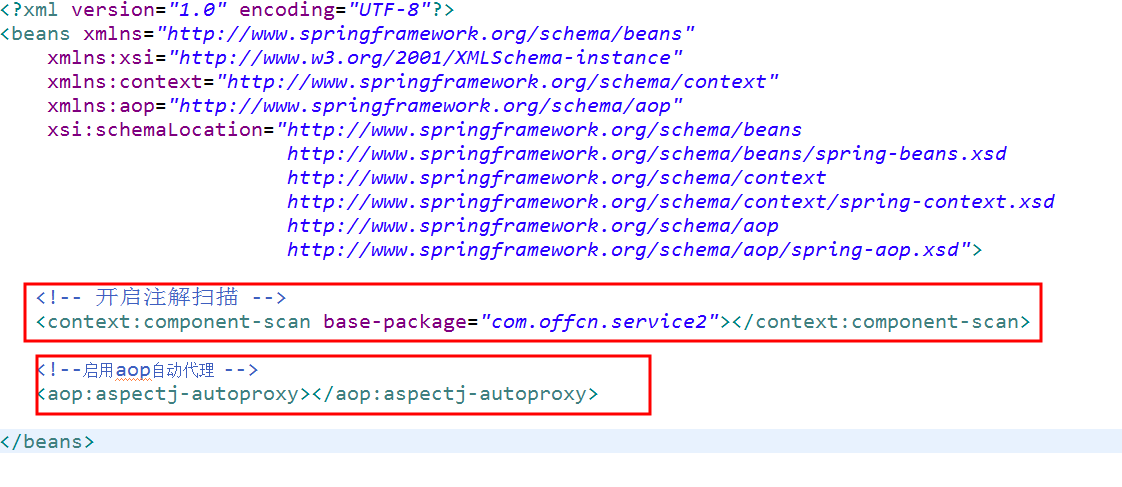
最终通知：辅助业务在主要业务之后执行，不管主要业务中是否出现异常，辅助业务都会执行

异常通知：辅助业务在主要业务之后执行，只有在主要业务出现异常时，辅助业务才会执行。

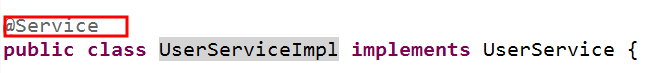
环绕通知：辅助业务可以在主要业务之前执行，辅助业务也可以在主要业务之厚执行，辅助业务也可以在主要业务之前，之后都执行。

### 以注解的方式完成aop的配置

1. spring配置文件：

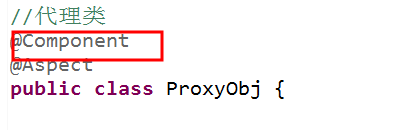


1. 目标类只用注解实例化

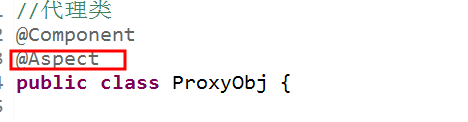


1. 代理类

3.1使用注解实例化代理类



3.2同时通过注解设置为切面类



3.3在辅助业务的方法上通过注解指定切点的范围



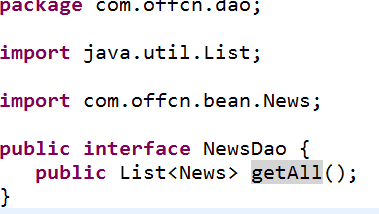
## 一站式开发中常用的注解

### XML方式一站式开发案例

**1.实体类News**,省略查看源码

**2.dao层NewsDao**

接口

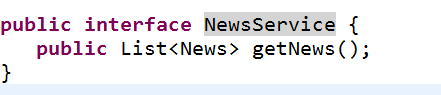


实现类

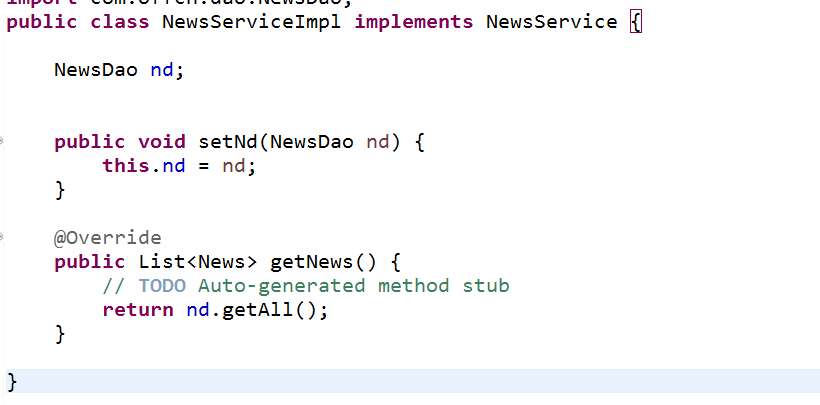


1. **Service层**

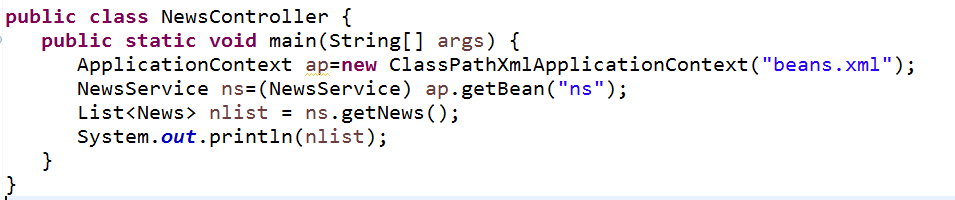
接口



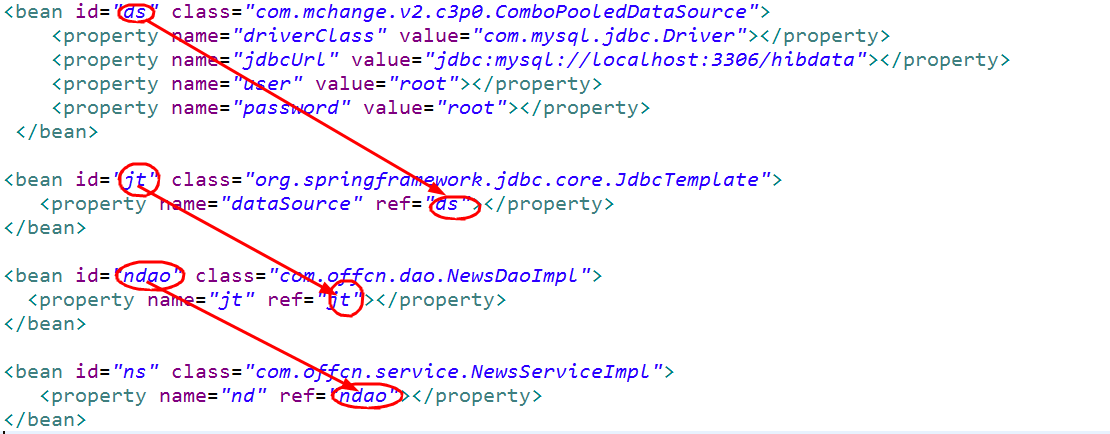
实现类



1. **控制层**



1. **配置文件**

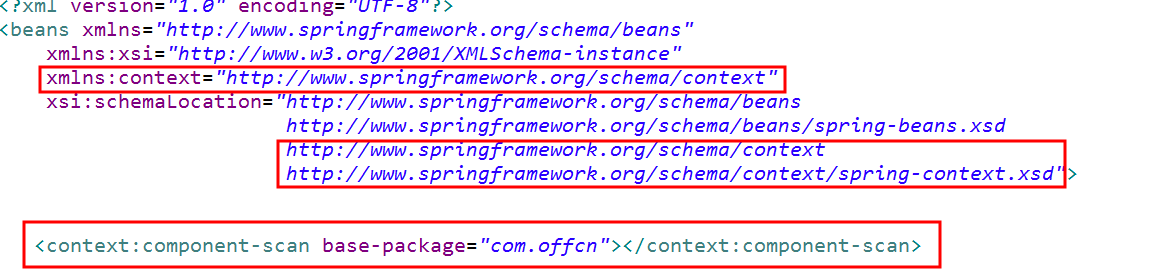


### 注解方式一站式开发案例

1. @Repository：作用在dao层的实现类上用来实例化dao层的对象。
2. @Service:作用在service层的实现类上用来实例化service层的对象。
3. @AutoWired/@Resource:作用在依赖的属性上用来给属性赋值。
4. @Controller，@Action:作用在控制层用来实例化控制层对象。
5. @Component:作用在三层架构以外其他类上用来实例化这些类。

注解：

1. 开启注解扫描

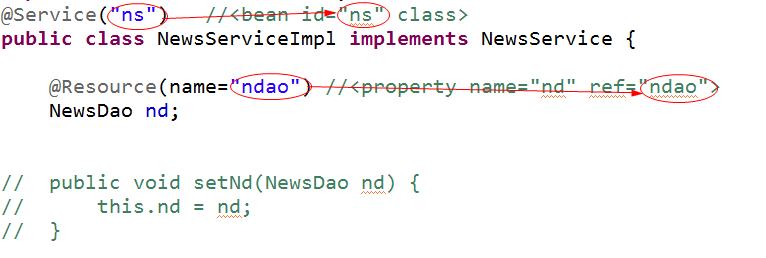


1. 加入依赖包spring-aop.jar
2. 在dao层，service层实现类上以及所依赖的属性上分别加注解。

dao层



Service层



1. 测试

|  |
| --- |
| **public** **class** NewsController {  **public** **static** **void** main(String[] args) {  ApplicationContext ap=**new** ClassPathXmlApplicationContext("beans.xml");  NewsService ns=(NewsService) ap.getBean("ns");  List<News> nlist = ns.getNews();  System.***out***.println(nlist);  }  } |

**细节**：

@Resource注解：

指定名称：先按照当前属性的名称name找到对应的bean节点，赋值给当前属性;

若没有找到对应的节点会按照属性的类型查找对应的bean节点，赋值给当前属性

没有指定名称：先按照当前属性的默认名称（属性类型的首字母小写）找到对应的bean节点，赋值给当前属性;若没有找到对应的节点会按照属性的类型查找对应的bean节点，赋值给当前属性。

@AutoWired注解：

按照当前属性（jt）的类型（JdbcTemplate）查找到系统中该类型的bean节点，赋值给当前属性。

**所以属性类型对应的bean节点只有一个，最好按照类型注入，否则按照名称注入**。

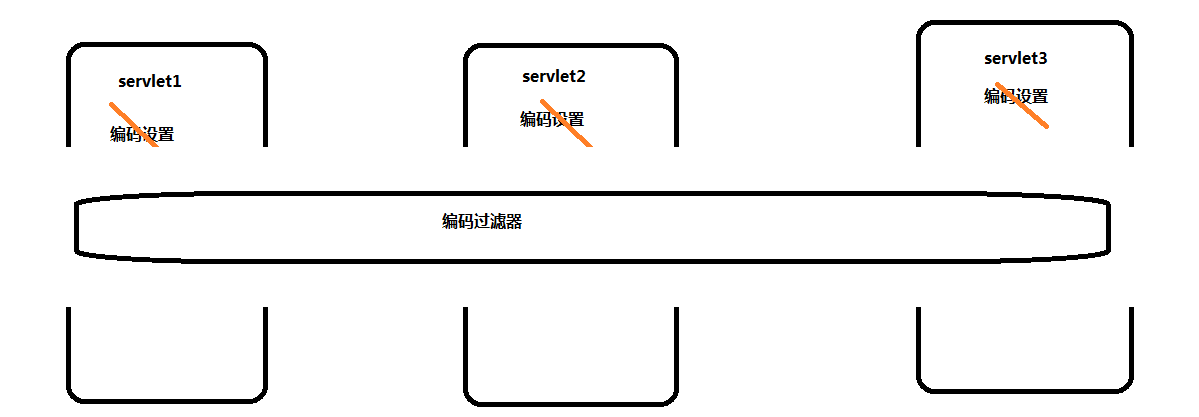
@Repository和@Service注解的value可以省略不写，默认value值为类名的首字母小写。

## AOP的介绍以及动态代理

### AOP介绍

AOP：（Aspect Oriented Programming）面向切面编程。通过[预编译](https://baike.baidu.com/item/%E9%A2%84%E7%BC%96%E8%AF%91/3191547)方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是[OOP](https://baike.baidu.com/item/OOP)的延续，是软件开发中的一个热点，也是[Spring](https://baike.baidu.com/item/Spring)框架中的一个重要内容，是[函数式编程](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0%E5%BC%8F%E7%BC%96%E7%A8%8B/4035031)的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的[耦合度](https://baike.baidu.com/item/%E8%80%A6%E5%90%88%E5%BA%A6/2603938)降低，提高程序的可重用性，同时提高了开发的效率。

1. 依赖动态代理实现功能的统一维护；
2. AOP是面向的对象编程的补充，将系统中和主要业务无关的代码提取出来封装到一个类中（代码块中），将封装后的类或者代码块成为切面。
3. 纵向重复，横向抽取。
4. 降低主要业务和辅助业务的耦合度，同时可以提高代码的重用性。



动态代理

**代理**：一个机构或者人代理另外一个人或者另外一个机构完成一部分工作。一个方法或者类代理另外一个方法或者另外一个类完成一部分其他辅助业务，降低主要业务和辅助业务的耦合度，同时通过辅助业务可以对主要业务增强。

**静态代理**：为每一个目标类（要被代理的类）创建一个代理类，静态代理中代理类在编译期就已经确定，静态代理的效率相对动态代理来说相对高一些，但是静态代理代码冗余大。

**动态代理**：根据不同的目标类（要被代理的类）通过反射在内存中动态生成一个代理类，动态代理则是JVM运行时动态生成，可以最大限度的降低代码的冗余。

常用的动态代理有两种JDK代理和CGLib代理。

### JDK代理

代理的目标类实现了接口（代理类和目标类实现相同的接口）

|  |
| --- |
| **public** **class** JDKProxy {  /\*  \* 为目标类生产代理类的方法  \*/  **public** Object getProxy(Object tarObj) {  /\*  \* 参数一：目标类的类加载器  \* 参数二：目标类实现的接口  \*参数三：InvocationHandler: 代理类对象执行方法时内存中执行的内容  \*/  **return** Proxy.*newProxyInstance*(tarObj.getClass().getClassLoader(), tarObj.getClass().getInterfaces(), **new** InvocationHandler() {    /\*  \* 参数一：代理类对象  \* 参数二：目标类的方法Method对象  \*参数三：目标类的方法参数  \*/  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  // **TODO** Auto-generated method stub  //执行目标类的主要业务  Object res = method.invoke(tarObj, args);  //辅助业务  System.***out***.println("提交事务");  **return** res;  }  });  }    } |

### CGLib代理

对代理目标类可以实现接口（代理类和目标类实现相同的接口）也可以是没有实现接口的目标类（代理类继承目标类）。

加入依赖包：



|  |
| --- |
| **public** **class** CGLibProxy {  /\*  \* 为目标类生产代理类的方法  \*/  **public** Object getProxy(Object tarObj) {  Enhancer en=**new** Enhancer();  en.setClassLoader(tarObj.getClass().getClassLoader());//目标类的类加载器  //en.setInterfaces(tarObj.getClass().getInterfaces());//目标类实现的接口  en.setSuperclass(tarObj.getClass());  en.setCallback(**new** MethodInterceptor() {    @Override  **public** Object intercept(Object proxy, Method method, Object[] args, MethodProxy arg3) **throws** Throwable {  // **TODO** Auto-generated method stub  //执行目标类的主要业务  Object res = method.invoke(tarObj, args);  //辅助业务  System.***out***.println("提交事务");  **return** res;  }  });//代理类对象执行方法时内存中执行的内容  **return** en.create();  }    } |

## SpringAOP配置

1. 连接点：需要增强的方法
2. 切点：方法被增强以后成为切点。
3. 通知：增强的内容成为通知。
4. 切面：把通知作用在切点上形成切面（通知+切点）。
5. 织入：把通知作用在切点上过程称为织入的过程。
6. 目标类：要被代理（增强）的类
7. 代理类：含有增强的内容的类

### 以XML方式完成SpringAOP的配置

SpringAOP配置步骤：

1. 依赖包



1. 准备目标类(UserService,UserServiceImpl)
2. 准备代理类以及增强的内容(辅助业务)
3. 织入的完成：把代理类中的辅助业务作用在目标类中方法上

在Spring的配置文件中配置如下



指定切点的语法

expression=execution(访问修饰 返回值类型 方法名（参数）)

访问修饰:（非必须）,可以省略不写支持任何访问修饰符

返回值类型：\*，任意返回值类型

方法名：包名(\*任意一级包,..包中所有内容).类名(\*).方法名（\*）

参数：（..任意参数）

案例：

execution( \* com.offcn.service2.\*.\*(..))：com.offcn.service2包中的任意类任意方法

execution( \* com.offcn.service2..\*(..))：com.offcn.service2包中的任意类或者任意包中的任意方法。

通知：

前置通知：辅助业务在主要业务之前执行。

后置通知：辅助业务在主要业务之后执行，当主要业务中出现异常，辅助业务不执行,在辅助业务中可以得到主要业务执行后的返回值。

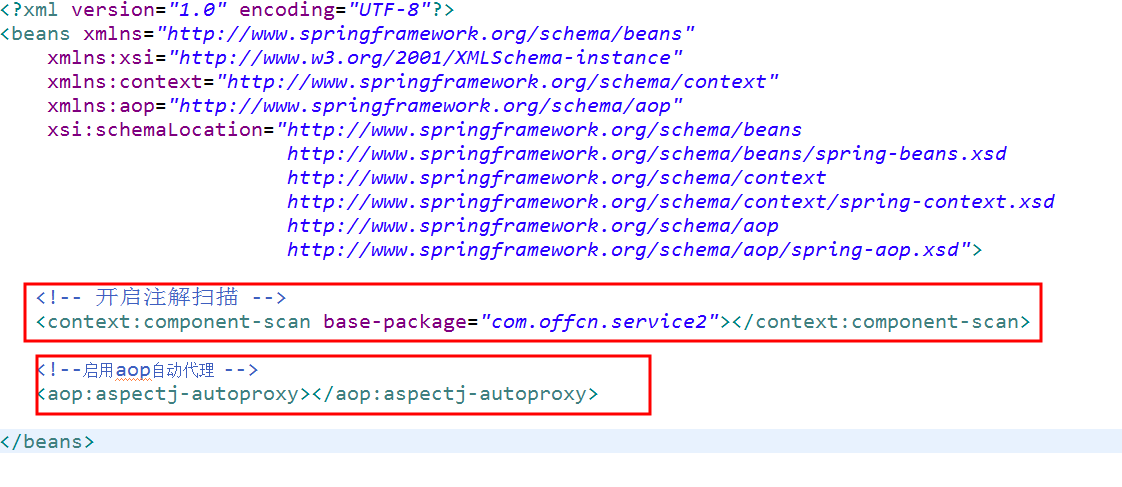
最终通知：辅助业务在主要业务之后执行，不管主要业务中是否出现异常，辅助业务都会执行

异常通知：辅助业务在主要业务之后执行，只有在主要业务出现异常时，辅助业务才会执行。

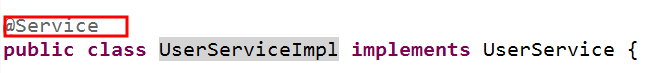
环绕通知：辅助业务可以在主要业务之前执行，辅助业务也可以在主要业务之厚执行，辅助业务也可以在主要业务之前，之后都执行。

### 以注解的方式完成aop的配置

1. spring配置文件：

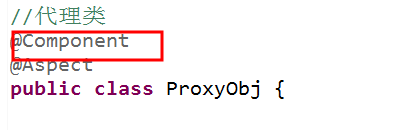


1. 目标类只用注解实例化

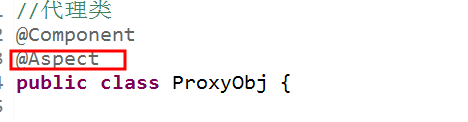


1. 代理类

3.1使用注解实例化代理类



3.2同时通过注解设置为切面类



3.3在辅助业务的方法上通过注解指定切点的范围

