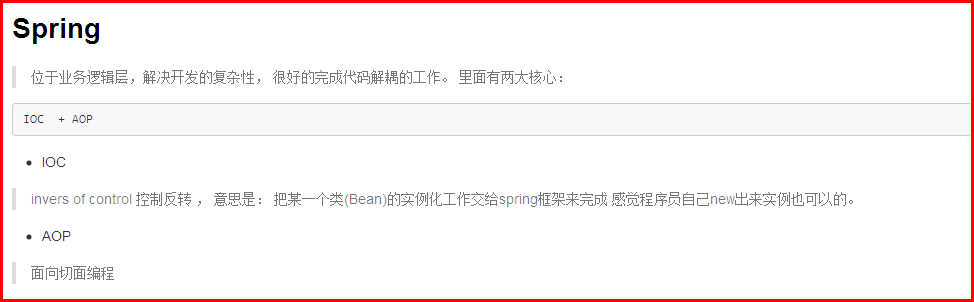
**Spring**

==========================================================================Spring\_note01

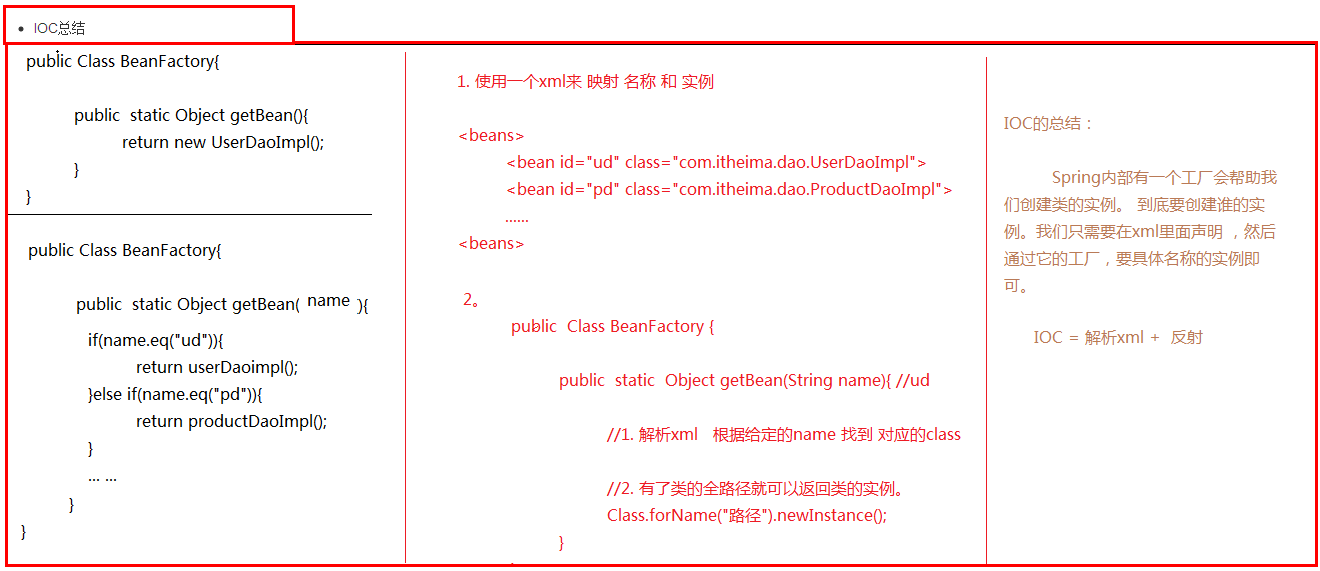








工厂类的缺点 : 针对工厂类 , 实现多种 对象调用的 不同Impl的实现,但只能有一种impl的实现





applicationContext.xml 的 标签解读





1. spring生成Bean

// 1. 创建 applicationContext实例

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

// 2. spring --> 工厂要实例 --> 拿到 Service --> 就可以对Service 层进行操作

UserService userService = (UserService) context.getBean("us");

生成Bean的3重方式 :

1. 无参构造生成 --> 建议使用 --> impl类一定要有一个无参构造,否则报错

xml:

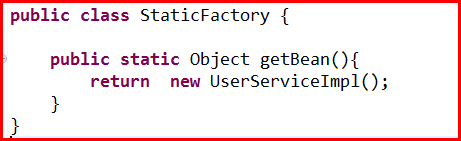
<bean id="us" class="com.sp.service.impl.UserServiceImpl">

2. 静态工厂 --> 了解

1. xml:

<bean id="us" class="com.sp.factory.StaticFactory" factory-method="getBean"></bean>

2. 建立一个 工厂类



3. 实例工厂 --> 了解

xml:

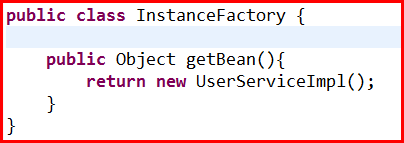
<!-- 声明了工厂 -->

<bean id="factory" class="com.sp.factory.InstanceFactory" ></bean>

<!-- 根据 在factory-bean="factory" 中去找 声明工厂的id="factory" ,然后找到那个类 -->

<bean id="us" factory-bean="factory" factory-method="getBean"></bean> -->

2. 建立一个 实例工厂类 -- 提供 一个获取 实例的方法



2. 注入 DI dependency Injection 依赖注入



给属性赋值

1. Init
2. setXXX

// 01 构造方法注入 --> 可以写有参构造(不需要写无参构造)

xml:

<!-- name="name" 是 Impl中的 属性名 value="aobama" 给属性名赋值 -->

<bean id="us" class="com.sp.service.impl.UserServiceImpl02">

<constructor-arg name="name" value="aobama"></constructor-arg>

</bean>

2. 提供一个 有参构造 , 和一个 属性

private String name;

public UserServiceImpl02(String name) {

super();

this.name = name;

}

// 02 set方法注入 --> 需要无参构造

xml:

<!-- 注入简单字符串 -->

<!-- name="address" 是 Impl中的 属性名 value="北京" 给属性名赋值 -->

<bean id="us" class="com.sp.service.impl.UserServiceImpl02">

<property name="address" value="北京"></property>

</bean>

<!-- 注入数组 -->

<!-- <bean id="us" class="com.sp.service.impl.UserServiceImpl02">

<property name="address">

<array>

<value>北京</value>

<value>上海</value>

<value>深圳</value>

<value>广州</value>

</array>

</property>

</bean> -->

<!-- 注入list集合 -->

<!-- <bean id="us" class="com.sp.service.impl.UserServiceImpl02">

<property name="address">

<list>

<value>北京1</value>

<value>上海2</value>

<value>深圳3</value>

<value>广州4</value>

</list>

</property>

</bean> -->

<!-- 注入Map集合 -->

<!-- <bean id="us" class="com.sp.service.impl.UserServiceImpl02">

<property name="address">

<map>

<entry key="地址1" value="北京"></entry>

<entry key="地址2" value="上海"></entry>

<entry key="地址3" value="深圳"></entry>

<entry key="地址4" value="广州"></entry>

</map>

</property>

</bean> -->

2. (必须一个无参构造,如果写了有参构造) 提供一个相应类型的属性 和set方法

private Map<String , String> address;

//private List address;

//private Array address;

public void setAddress(Map<String, String> address) {

this.address = address;

}

// 2.1 注入引用对象 (set) --> 一般经常使用

xml:

<!-- new UserServiceImpl() -->

<bean id="us" class="com.sp.service.impl.UserServiceImpl02">

<!-- name这个其实就是类中的属性 名称 ， ref 就是告诉spring指向哪一个类的引用。 -->

<property name="userDao" ref="ud"></property>

</bean>

2. (必须一个无参构造,如果写了有参构造) 提供一个相应的 对象类型的属性,set方法

private UserDao userDao;

public void setUserDao(UserDao userDao) {

this.userDao = userDao;

}

// 3. 名称空间

// 3.1 p命名空间注入

1. xml

1. 引用约束文件

2. 指定的标签

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- p -->

<bean id="us" class="com.sp.service.impl.UserServiceImpl02" p:pStr="北京"> </bean>

</beans>

2. 提供一个想用类型的属性,set方法

private String pStr;

public void setPStr(String pStr) {

this.pStr = pStr;

}

// 3.2 c 命名空间注入

1. xml

1. 引用约束文件

2. 指定的标签

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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

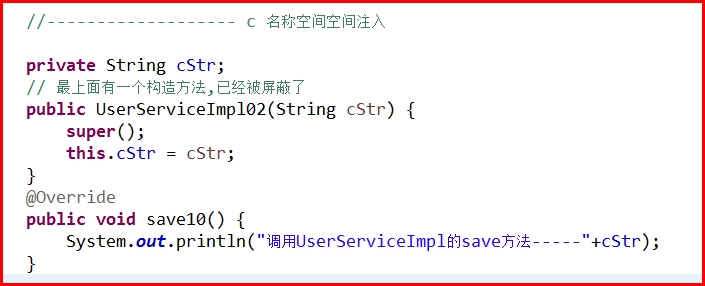
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<!-- c -->

<bean id="us" class="com.sp.service.impl.UserServiceImpl02" c:cStr="北京"> </bean>

</beans>

2. 提供一个想用类型的属性,和构造方法



// 4. EL 属性注入

1. xml 并不需要做什么 , 但是格式是 #{} 的格式,这个的强大之处是可以做运算

<!-- el 格式 #{} -->

<bean id="us" class="com.sp.service.impl.UserServiceImpl02">

<property name="elStr" value="#{'北京sss'}"> </property>

<!-- <property name="pStr" value="#{1+1}"></property>

<property name="pStr" value="#{'beijing'.length()}"></property>

<property name="pStr" value="#{1 > 2}"></property> -->

</bean>

2. 提供一个想用类型的属性,set方法

private String elStr;

public void setElStr(String elStr){

this.elStr = elStr;

}

4. Spring和Servlet整合

0. 创建 javaWeb\_project

1. 导入jar包 , 之前的8个

2. 导入日志文件

3. servlet 访问 service层

// 传统方法 1.

UserService userService = new UserServiceImpl();

userService.save();

// 2. 解析xml ， 创建工厂 --> 缺点 : 每次进来都会创建 一个工厂 --> 消耗性能

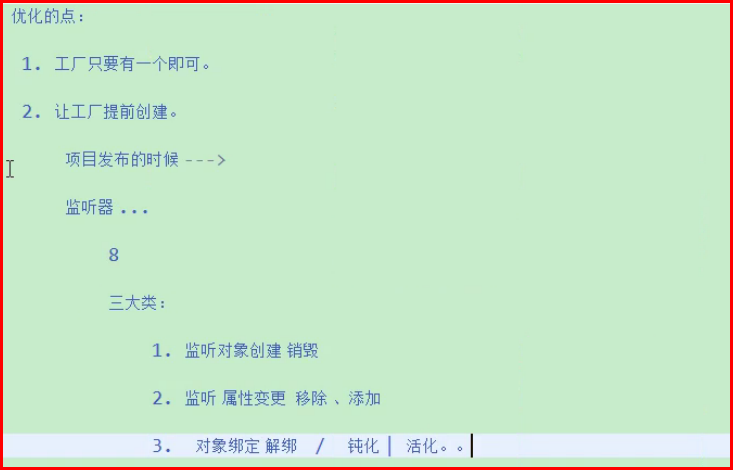
// ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

// //问工厂要实例。

// UserService userService = (UserService) context.getBean("userService");

// userService.save();

// 3. 优化 --> 视频可以在深入一下



// 3.1 导入jar包 spring-web-4.2.9.RELEASE.jar

// 3.2 web.xml 配置

<!-- 一旦servletcontext创建， 那么就会执行这个ContextLoaderListener,接着就需要解析xml ， 然后创建工厂。

下面的context-param 的意义就是指定xml在哪一个地方

视频需要看一下

-->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

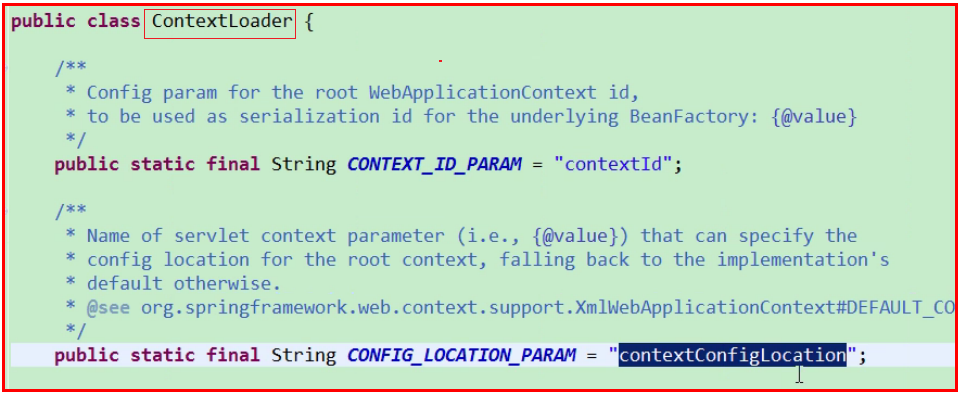
</context-param>

--------------------------------

ContextLoaderListener : 实现了 ServletContextListener



ContextLoader --> contextConfigLocation



// 3.3 servlet操作:

WebApplicationContext context = WebApplicationContextUtils.getWebApplicationContext(getServletContext());

UserService userService = (UserService) context.getBean("userService");

userService.save();

==========================================================================Spring\_note02

1. IOC注解

2. spring测试

3. AOP(面向切面编程) 开发

1. IOC注解

0. 使用直接 要额外的 添加 jar包 spring-aop-4.2.9.RELEASE.jar --> jar包是针对 aop的

1. XML

<!--

spring 的 doc 的40系列 的是schema

点击 搜索 the XXX schema

-->

<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 definitions here -->

<!--

打开注解的扫描开关 spring 会检测我们的类上面是否有注解，如果有，就形成映射。

userService === UserServiceImpl

component : 组件 , spring 将action,context...都称之为组件

base-package="com.sp" 会搜索填入的路径下全部的文件 (包括下一层路径)

-->

<context:component-scan base-package="com.sp"></context:component-scan>

<!-- 这个只会扫描，已经在这个applicationContext.xml 里面声明的bean 这些类，不声明的东西，不能扫描

这个就没有什么用

-->

<!-- <context:annotation-config/> -->

</beans>

2. UserServiceImpl 类添加注解

@Component("userService") // == <bean id=""> id的值

public class UserServiceImpl implements UserService {

@Override

public void save() {

System.out.println("调用UserServiceImpl的save方法");

}

}

3. 使用 和 一般的spring 没什么区别

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

UserService bean = (UserService)context.getBean("userService");

bean.save();

执行的顺序: 当读取到注解开关被打开的时候 <context:component-scan base-package="com.sp"></context:component-scan>

当传递 userService --> context.getBean("userService"); 的时候

就会扫描包 找到 有 @Component("userService") 的类 , 将 UserServiceImpl 实例传递过去 给 UserService bean = UserServiceImpl ;

4. ----------- 注解的 一些 关键字

xml中:

UserServiceImpl 类上:

//spring 把我们分层的三层结构中的类，都称之为 组件。

//@Component("userService") //这个引号里面的值，等同于 <bean id="">

//Spring -- 但是后面它也扩展了，对每一层组件的称呼 , 当然也可以全部用 component

//@Service("userService") : service

//@Repository : dao

//@Controller : action

@Scope("prototype")

同xml中bean标签的属性scope <bean scope=""/>，用来设置一个类作用范围

singleton , prototype , request , response

@PostConstrut

同xml中bean标签的属性init-method <bean init-method=""/>,

用来设置spring框架初始化此类实例时调用的初始化方法,

标注在此类的初始化方法上

@PreDestroy

同xml中bean标签的属性init-method <bean destroy-method=""/>,

用来设置spring框架销毁此类实例时调用的销毁方法,

标注在此类的销毁方法上

2.DI注入 UserServiceImpl 类中: --> 主要用于 注入

---> 一般属性 , 没什么意义 , 可以直接赋值之类的

@Value("碧海哦...")

private String name;

---> 注入对象 : UserDao

1.

创建Dao 接口 和 impl 类

2. UserDaoImpl (dao层 或 持久层)

@Repository("userDao") // @Repository() 不写其实也有一个默认的名字 --> 类名首字母小写 userDaoImpl,那么在service上面,@Resource(name="userDaoImpl")了

// 或者 @Component("userDao")

public class UserDaoImpl implements UserDao { }

3. UserServiceImpl(service层) 中

// 注入的资源是谁 。 要写一个name 在前面 = "标记的名称" 直接根据给定的名称去找。

@Resource(name="userDao")

private UserDao userDao ;

// ------- 或者

// @Autowired // 自动装配， 根据现在需要注入的类型 UserDao 找到这个userDao的实现，然后把实现类的对象给注入进来。

// 只适合实现类只有一个的情况下 , 如果有多个实现类 (UserDaoImpl , UserDaoImpl02) , 就报错

// private UserDao userDao ; 

3. spring测试

1. 导入jar包 spring-test-4.2.9.RELEASE.jar

2. xml 注入

<!-- 在xml里面声明bean , Spring测试类 , 可以不用打开注解开关 -->

<bean id="userService02" class="com.sp.service.impl.UserServiceImpl02"></bean>

<bean id="userDao03" class="com.sp.dao.impl.UserDaoImpl03"></bean>

3. 在类上

1. TestUser02.java

@RunWith(SpringJUnit4ClassRunner.class) //SpringJUnit4ClassRunner 测试使用的环境是什么， 类

@ContextConfiguration("classpath:applicationContext02.xml")

public class TestUser02 {

//---------------------------------------------------Spring 测试

@Autowired

private UserService userService02;

@Test

public void testSave(){

userService02.save01();

}

}

2. userService02

//--------------------------spring 测试

@Resource(name="userDao03")

private UserDao userDao03;

@Override

public void save01() {

System.out.println("调用UserServiceImpl02 的save01 方法");

System.out.println("1: userDao03: "+ userDao03.getClass());

userDao03.save01();

}

3. userDao03 --> 没什么特别的 , 就是一个方法调用

// spring test

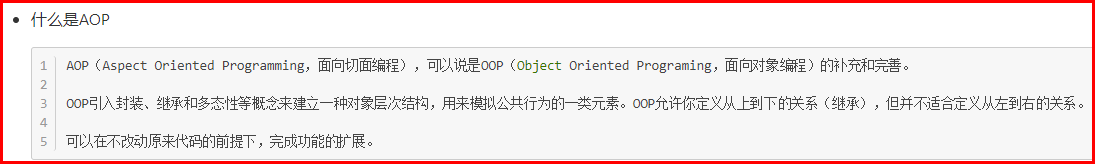
@Override

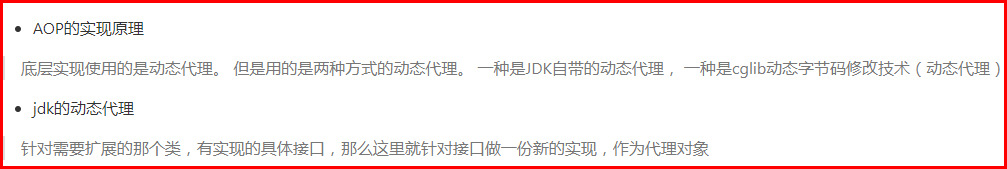
public void save01() {

System.out.println("userDaoImpl03 save01 invoke 11111~~~");

}

4. AOP(面向切面编程) 开发





1. 原理: 底层是 动态代理

有两个: 1. 代理类有接口 --> jdk 自带的动态代理

2. 代理类没有接口 --> cglib动态字节码修改技术

1. 有接口的

1. 创建 UserService接口

2. 创建UserServiceImpl 实现类

3. 创建 testUser.java 测试类,在测试类中:

@Test // 针对这个类有接口。

public void testJDKProxy() {

// 1. 创建真实对象

final UserService userService = new UserServiceImpl();

// 2. 创建代理

// 参数1: 类加载器 参数2: 真实对象 参数3: 匿名内部类

UserService proxyService = (UserService) Proxy.newProxyInstance(userService.getClass().getClassLoader(),

userService.getClass().getInterfaces(), new InvocationHandler() {

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

// 调用真实对象

return method.invoke(userService, args); // --- save();

}

});

// 3. 调用代理的方法

proxyService.save();

}

如果没有接口 , 那么参数2: userService.getClass().getInterfaces() 就无法获取到;

2. 没有接口

1. 没有接口用 enhancer 设置父类是谁。

这个会创建一个子类 , 让当前的实现类称为父类 (内部实现)

直接在 testUser.java 测试类中 :

@Test // 针对这个类没有接口。

public void testCglibProxy() {

// 1. 创建真实对象

final UserServiceImpl userService = new UserServiceImpl();

// 2. 创建代理对象

Enhancer enhancer = new Enhancer();

// 设置父类是谁。

enhancer.setSuperclass(UserServiceImpl.class);

// 这个方法等同于 InvocationHandler

enhancer.setCallback(new MethodInterceptor() {

@Override

public Object intercept(Object arg0, Method arg1, Object[] arg2, MethodProxy arg3) throws Throwable {

return arg1.invoke(userService, arg2);

}

});

// 创建代理对象

UserServiceImpl proxyService = (UserServiceImpl) enhancer.create();

proxyService.save();

}

Spring的aop 的底层原理 : 在查找实现类是否有接口,入果有,那么 就是jdk的动态代理 , 反之就是cglib的动态代理

2.术语:

连接点

切入点

切面

增强

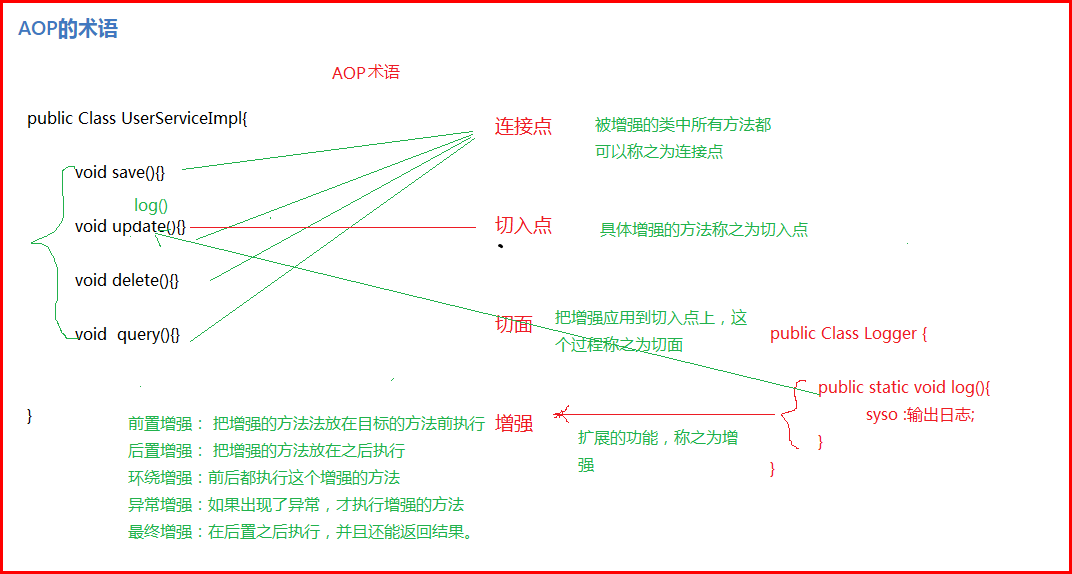
前置增强

后置增强

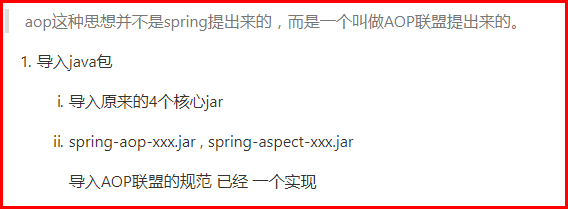
环绕增强

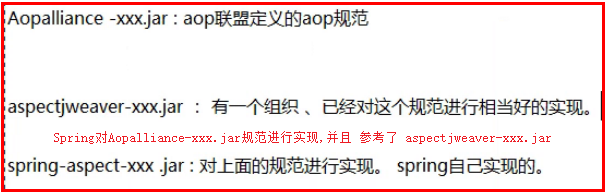
异常增强

最终增强



3. aop入门 --> 疑问: 不同包下的不同类的增强 , 需要扫描包的范围扩大吗? 还是可以有多个 <aop:pointcut 切入点 ? --> 估计只能缩小包扫描的范围





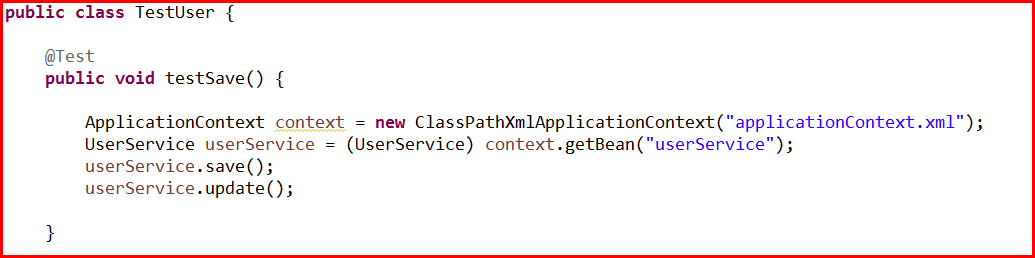
1. 增强方法



2. 接口和实现类



1. 测试类



4. applicationContext.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!--

spring 的 doc 的40系列 的是schema

点击 搜索 the XXX schema

约束导入AOP的约束 the aop schema

-->

<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"*>

<!-- 在xml里面声明bean -->

<bean id=*"userService"* class=*"com.sp.service.impl.UserServiceImpl"*></bean>

<!-- 切面增强类 -->

<bean id=*"logAspect"* class=*"com.sp.aspect.Logger"*></bean>

<!-- 把谁的方法增强到谁的方法上 loglogAspect 里面的一个log方法增强到userService里面的save方法上-->

<aop:config>

<!-- 切入点

execution(\* com.xyz.myapp.service.\*.\*(..))

第一个\* ： 表示任意返回值

com.xyz.myapp.service : 指定找这个包下

第二个\* 找上面这个包下的任意类

第三个\* 找到这些类中的任意方法

(..) : 匹配任意方法参数

-->

<aop:pointcut expression=*"execution(\* com.sp.service.impl.\*.save(..))"* id=*"savePointCut"*/>

<!-- 把logAspect这个类中的log 应用到切入点找到的那些方法的前面去。 执行前置增强 -->

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

<!-- 根据上面的表达式找到的方法，给这些方法应用前置增强，给他们增强一个log的功能 -->

<!-- <aop:before method="log" pointcut-ref="savePointCut"/> -->

<!-- <aop:after method="log" pointcut-ref="savePointCut"/> -->

<!-- 环绕增强 注意: 在方法增强的前面需要添加一个额外的方法aroundLog ,-->

<!-- <aop:around method="aroundLog" pointcut-ref="savePointCut"/> -->

<!-- 异常增强 , 有异常.才会打印log()方法 -->

<!-- <aop:after-throwing method="log" pointcut-ref="savePointCut"/> -->

<!-- 最终通知 : 如果save方法有返回值,那么可以拿到这个返回值,并没什么用-->

<!-- <aop:after-returning method="log" pointcut-ref="savePointCut"/> -->

</aop:aspect>

</aop:config>

</beans>

3. 入门 --> 手写版

1. 导入jar包

aopalliance-1.0.jar --> AOP联盟的声明的规范

spring-aop-4.2.9.RELEASE.jar --> 这个IOC 注解 也需要用到 (必须的包)

aspectjweaver-1.8.9.jar --> 其他大神/团队实现的(在Spring之前)

spring-aspects-4.2.9.RELEASE.jar -->Spring对上面的 AOP联盟的声明的规范 的实现 ,(有一部分包括aspectjweaver)

2. 创建增强类

Logger.java

public class Logger {

public static void log(){

System.out.println("输入日志~~");

}

}

3. xml

1. 导入aop约束

2.

<!-- 在xml里面声明bean -->

<bean id="userService" class="com.sp.service.impl.UserServiceImpl"></bean>

<!-- 切面增强类 -->

<bean id="logAspect" class="com.sp.aspect.Logger"></bean>

<!-- 把谁的方法增强到谁的方法上 loglogAspect 里面的一个log方法增强到userService里面的save方法上-->

<aop:config>

<!-- 切入点

execution(\* com.xyz.myapp.service.\*.\*(..))

第一个\* ： 表示任意返回值

com.xyz.myapp.service : 指定找这个包下

第二个\* 找上面这个包下的任意类

第三个\* 找到这些类中的任意方法

(..) : 匹配任意方法参数

-->

<aop:pointcut expression="execution(\* com.sp.service.impl.\*.\*(..))" id="savePointCut"/>

<!-- 把logAspect这个类中的log 应用到切入点找到的那些方法的前面去。 执行前置增强 -->

<aop:aspect ref="logAspect">

<!-- 根据上面的表达式找到的方法，给这些方法应用前置增强，给他们增强一个log的功能 -->

<aop:before method="log" pointcut-ref="savePointCut"/>

</aop:aspect>

</aop:config>

3. 在 TestUser.java 里面调用

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

UserService userService = (UserService) context.getBean("userService");

userService.save();

增强 ------------ (环绕 , 后置 , ...)

1. UserService 增加 void update(); 方法

并且在 TestUser 调用 userService.update();

那么

1. 可以调用 void update(); 方法

xml : --这个表达式 , 不用改变

<aop:pointcut expression="execution(\* com.sp.service.impl.\*.\*(..))" id="savePointCut"/>

2. 不可以调用 void update(); 方法

xml : -- 这个表达式的 找到的类的 方法 , 给写死 , 不用通配符 (向struts2的方法的破匹配 , 方法名可以是: save\_\* )

<aop:pointcut expression="execution(\* com.sp.service.impl.\*.save(..))" id="savePointCut"/>

2. 后置强 ( save 还是前置) -- 以下 都是

<aop:pointcut expression="execution(\* com.sp.service.impl.\*.save(..))" id="savePointCut"/>

1. xml : -- 表达式只看save方法

<aop:after method="log" pointcut-ref="savePointCut"/>

3. 环绕

1. xml : 同时写

<aop:before method="log" pointcut-ref="savePointCut"/>

<aop:after method="log" pointcut-ref="savePointCut"/>

2.

1. 在 增强的方法中,新添加一个 有参数的方法

public static void log() { // --> 之前原有的

System.out.println("输入日志~~");

}

// --> 新添加一个 有参数的方法 参数类型 ProceedingJoinPoint

// --> 这个参数 , 就是传递 真正执行的save方法

public static void aroundLog(ProceedingJoinPoint joinPoint) {

try {

log(); --> 调用增强方法

joinPoint.proceed(); // --> 真正执行的save方法

log(); --> 调用增强方法

} catch (Throwable e) {

e.printStackTrace();

}

}

2. xml: --> 注意方法的改变

<aop:around method="aroundLog" pointcut-ref="savePointCut"/>

4. 异常增强

1. xml

<aop:after-throwing method="log" pointcut-ref="savePointCut"/>

2. 执行的时间 --> 执行的save方法 , 要有一个异常

@Override

public void save() {

System.out.println("调用UserServiceImpl 的save方法22~~" );

// 异常增强测试

int i = 10 / 0;

}

5. 最终增强 :

1. xml

<aop:after-returning method="log" pointcut-ref="savePointCut"/>

2. 其他的可以都不变

最终增强 - 与其他的区别:

最终增强 可以在save方法上面 设置 一个 返回值 , 在增强的log方法中 , 可以拿到那个返回值

1. UserServiceImpl , 方法返回值

@Override // 最终增强 返回值测试

public int save() {

System.out.println("调用UserServiceImpl 的save方法22~~" );

return 10;

}

2. Logger , log 拿到 返回值 --> 问题 : log怎么拿到返回值 ????????

public static void log() {

System.out.println("输入日志~~");

}

6. 扩展: 多个类怎么找 (就多个service吧)

1. 必须在 xml中注册这个类:

<bean id="userService01" class="com.sp.service.impl.UserServiceImpl01"></bean>

2. 其他的 就一样的 用法

==================================================================

========day03

1. AOP 注解开发

1. xml

<!-- 在xml里面声明bean -->

<bean id="userService" class="com.sp.service.impl.UserServiceImpl"></bean>

<!-- 切面增强类 -->

<bean id="logAspect" class="com.sp.aspect.Logger"></bean>

<!-- 打开aop 切面的自动代理 如果使用了注解aop 就必须使用这个。 打开开关 -->

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

2. 在切面增强类中 : (两步: )

@Aspect // 1. --> 声明是切面增强类

public class Logger {

// 2. 告诉这个方法将要做的是 前置增强 (在扩展的功能方法上，使用@Before @Afte..r来指定走的是什么增强。)

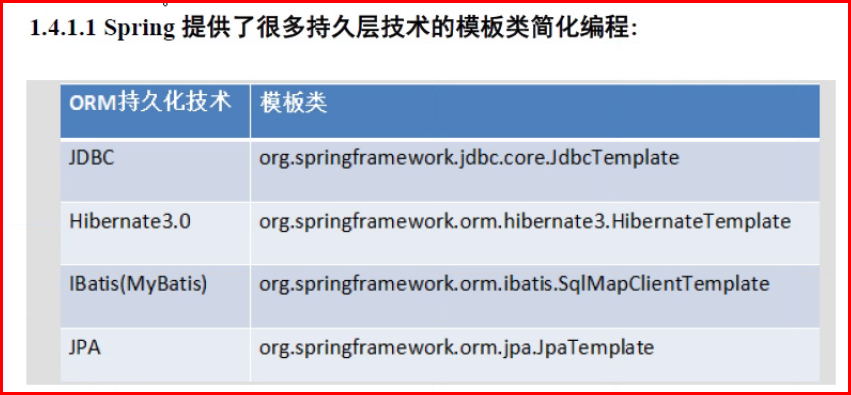
@Before(value = "execution(\* com.sp.service.\*.\*(..))")

public static void log() {

System.out.println("输入日志~~");

}

}



2. jdbc 模板 --> JDBCtemplate

Spring位于业务逻辑层，前不着村，后不着店， 为了更好的推广自己，

所以它除了做好了自己本身的量大核心 （IOC + AOP）之外，还对Dao层的框架 提供了支持。

就算没有框架，也对原生的JDBC提供了支持。

jdbc的模板，其实就是spring对之前的jdbc 增删改查 进行了封装而已。

1. 模板入门 --> 最基本的

1. 导jar包

spring-beans-4.2.9.RELEASE.jar

spring-context-4.2.9.RELEASE.jar

spring-core-4.2.9.RELEASE.jar

spring-expression-4.2.9.RELEASE.jar --> 核心jar包

spring-tx-4.2.9.RELEASE.jar --> tx 事务

spring-jdbc-4.2.9.RELEASE.jar --> jdbc使用

spring-test-4.2.9.RELEASE.jar --> 测试 使用

mysql-connector-java-5.1.39-bin.jar --> jdbc 驱动包

2. 创建数据库

3. Jdbc\_入门.java 类中

只用记忆两个类: JdbcTemplate

DriverManagerDataSource

com.mysql.jdbc.Driver --> 打 Driver --> 然后导入包 , 拿包名 导入即可

// Driver

DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");// --> jdbc驱动

dataSource.setUrl("jdbc:mysql:///spring\_day03\_jdbc");

dataSource.setUsername("root");

dataSource.setPassword("19920317");

JdbcTemplate jdbcTemplate = new JdbcTemplate();

jdbcTemplate.setDataSource(dataSource);

String sql = "insert into t\_user values(null,?,?)";

jdbcTemplate.update(sql, "张三",19);

2. jdbc --> CRUD

jdbcTemplate 只有 update 和 query 两个方法 ,并没有 delete 和 insert方法

类似于 dbutils.

增:

String sql = "insert into t\_user values(null,?,?)";

jdbcTemplate.update(sql, "张三",19);

删:

String sql = "delete from t\_user where name = ?";

jdbcTemplate.update(sql, "张三");

改:

String sql = "update t\_user set age =? where name=?";

jdbcTemplate.update(sql, 49 ,"张三");

查:

--> 1. 查询有几条 记录 --> queryForObject()

String sql = "select count(\*) from t\_user";

//参数二： 表示返回的结果类型是什么。

Long result = jdbcTemplate.queryForObject(sql, Long.class);

--> 2. 查询单个对象 --> 需要自己封装对象到 bean这中 --> queryForObject()

// 1. 创建一个javabean --> User.java

// 2. Jdbc\_CRUD.java类的代码

// 1. 查询语句

String sql = "select \* from t\_user where name = ?";

// --> 需要实现一个RowMapper接口

User user = jdbcTemplate.queryForObject(sql , new MyRowMapper() ,"张三");

// 2. 实现 RowMapper接口 --> 用来封装数据(bean)

class MyRowMapper implements RowMapper<User>{

@Override

public User mapRow(ResultSet rs, int rowNum) throws SQLException {

User user = new User();

user.setId(rs.getInt("id"));

user.setName(rs.getString("name"));

user.setAge(rs.getInt("age"));

return user;

}

}

--> // 3. 查询多个对象 --> 需要自己封装对象到 bean这中 --> query();

// 1. 创建一个javabean --> User.java

// 2. Jdbc\_CRUD.java类的代码

// 1. 查询语句

String sql = "select \* from t\_user";

List<User> list = jdbcTemplate.query(sql, new MyRowMapper());

// 2. 实现 RowMapper接口

class MyRowMapper implements RowMapper<User>{

@Override

public User mapRow(ResultSet rs, int rowNum) throws SQLException {

User user = new User();

user.setId(rs.getInt("id"));

user.setName(rs.getString("name"));

user.setAge(rs.getInt("age"));

return user;

}

}

2. JDBCtemplate 的注入写法

1. 代码

UserService userService = new UserServiceImpl();

userService.save01();

.....

2. IOC + DI + JDBCtemplate --> (注入datasource) 不使用c3p0



xml:



-------------(注入datasource) 使用c3p0

<!-- c3p0 连接池 -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"></property>

<property name="jdbcUrl" value="jdbc:mysql:///spring\_day03\_jdbc"></property>

<property name="user" value="root"></property>

<property name="password" value="19920317"></property>

<property name="initialPoolSize" value="10"></property>

</bean>

Jdbc --> 是创建对象

c3p0 --> 是创建连接池 , 在连接池中获取对象

dao: --> 用法 和 以前的 一样了 --> 只要关注 xml的适配 , 即可

private JdbcTemplate jdbcTemplate;

public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate = jdbcTemplate;

}

@Override

public void save01() {

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

String sql = "insert into t\_user values(null,?,?)";

jdbcTemplate.update(sql, "李四4", 39);

}

3. 事务管理

事务可以用来包装一组操作，只要这一组操作里面有一个逻辑运行失败，那么就可以让这一组里面的所有操作都以失败结束。

就会执行事务的回滚操作， 如果都成功，那么就执行事务的提交操作。

spring事务管理

> 由于Spring对持久层的很多框架都有支持 ， Hibernate 、 jdbc 、 MyBatis 由于使用的框架不同，所以使用事务管理操作API 也不尽相同。

为了规范这些操作， Spring统一定义一个事务的规范 ，这其实是一个接口 。这个接口的名称 ： PlatformTrasactionManager

> 并且它对已经做好的框架都有支持， 如果dao层使用的是JDBC 或者 mybatis ，那么 可以使用DataSourceTransactionManager 来处理事务

> 如果dao层使用的是 Hibernate ， 那么可以使用HibernateTransactionManager 来处理事务

1. 代码 --> 事务的 代码 写法

在 TestUser.java 类中

@Test // 事务的 代码 写法

public void testSave01() {

// 步骤5 : 创建 jdbcTemplate

final JdbcTemplate jdbcTemplate = new JdbcTemplate();

jdbcTemplate.setDataSource(dataSource);

// 步骤3

DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///spring\_day03\_jdbc");

dataSource.setUsername("root");

dataSource.setPassword("19920317");

// 步骤2

// PlatformTransactionManager transactionManager;

DataSourceTransactionManager transactionManager = new DataSourceTransactionManager();

transactionManager.setDataSource(dataSource);

// 步骤1

TransactionTemplate transactionTemplate = new TransactionTemplate();

transactionTemplate.setTransactionManager(transactionManager);

// 步骤4 执行事务

transactionTemplate.execute(new TransactionCallback<Object>() {

@Override // 参数,事务的状态对象 可以设置回滚

public Object doInTransaction(TransactionStatus arg0) {

try {

// 这个方法里面的代码，最终会在事务里面执行。

String sql = "insert into t\_user values( null , ? , ?)";

// 步骤 5 : 创建 jdbcTemplate

jdbcTemplate.update(sql, "王老五2", 7);

int a = 10 / 0; --> 异常

jdbcTemplate.update(sql, "王老五3", 27);

} catch (DataAccessException e) {

e.printStackTrace();

// 设置回滚

arg0.setRollbackOnly();

}

return null;

}

});

}

2. 事务的xml写法 --> 原理Aop 对方法 增强事务方法 (执行方法之前打开事务, 执行方法之后,关闭事务)

1. xml

<!--

约束:

the tx (Transaction...) schema

需要 AOP 联盟的jar包

aopalliance-1.0.jar

aspectjweaver-1.8.9.jar

spring-aspects-4.2.9.RELEASE.jar

-->

<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:tx="http://www.springframework.org/schema/tx" xsi:schemaLocation="

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

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd

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

<bean id="userService" class="com.sp.service.impl.UserServiceImpl">

<property name="userDao" ref="userDao"></property>

</bean>

<!-- 必须要注入jdbctemplate -->

<bean id="userDao" class="com.sp.dao.impl.UserDaoImpl">

<property name="jdbcTemplate" ref="jdbcTemplate"></property>

</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

<property name="dataSource" ref="dataSource"></property>

</bean>

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"></property>

<property name="url" value="jdbc:mysql:///spring\_day03\_jdbc"></property>

<property name="username" value="root"></property>

<property name="password" value="19920317"></property>

</bean>

<!-- 以下是事务管理的配置 -->

<!-- 步骤2. 已经设置好了事务的管理者 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"></property>

</bean>

<!-- 步骤1. 事务的建议 一定要有 -->

<tx:advice id="advice" transaction-manager="transactionManager">

<!-- 综合前面找到的那些方法，这里再进行一次过滤，就是给谁，哪一个方法用事务

必须要有

-->

<tx:attributes>

<tx:method name="save01"/>

<!-- <tx:method name="\*"/> --> <!-- \* 代表 多个方法 -->

</tx:attributes>

</tx:advice>

<!-- 步骤3. 要给谁，给哪个方法使用事务呢？ 其实是对 某个 方法的增强 -->

<aop:config>

<aop:pointcut expression="execution(\* com.sp.service.impl.\*.\*(..))" id="transactionPointCut"/>

<!-- 使用上面叫做advice的事务建议，来对后面这个切面找到的那些方法，给他们加上事务的操作 -->

<aop:advisor advice-ref="advice" pointcut-ref="transactionPointCut"/>

</aop:config>

</beans>

3. 注解方式 --> 这个使用的比较多 , 比上面的简单

1. xml

<!--

约束:

the tx (Transaction...) schema

需要 AOP 联盟的jar包

aopalliance-1.0.jar

aspectjweaver-1.8.9.jar

spring-aspects-4.2.9.RELEASE.jar

-->

<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:tx="http://www.springframework.org/schema/tx" xsi:schemaLocation="

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

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd

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

<bean id="userService\_anno" class="com.sp.service.impl.UserServiceImpl\_anno">

<property name="userDao" ref="userDao"></property>

</bean>

<!-- 必须要注入jdbctemplate -->

<bean id="userDao" class="com.sp.dao.impl.UserDaoImpl">

<property name="jdbcTemplate" ref="jdbcTemplate"></property>

</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

<property name="dataSource" ref="dataSource"></property>

</bean>

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"></property>

<property name="url" value="jdbc:mysql:///spring\_day03\_jdbc"></property>

<property name="username" value="root"></property>

<property name="password" value="19920317"></property>

</bean>

<!-- 以下是事务管理的配置 -->

<!-- 已经设置好了事务的管理者 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"></property>

</bean>

<!-- 使用事务注解，需要提供一个驱动 -->

<tx:annotation-driven transaction-manager="transactionManager"/>

</beans>

2. UserServiceImpl\_anno.java 类中添加注解

@Transactional // 可用于 类 或者 对象 --> 再类上,那么这个类的所有方法都会添加注解 , 如果只针对某个对象,那么就在该方法上添加注解@Transactional

public class UserServiceImpl\_anno implements UserService { }

// --> 一般是在 类上 添加注解@Transactional