# Spring

## Spring概念

1. Spring是开源的轻量级框架
2. Spring核心主要有两部分：
3. AOP

面向切面编程，扩展功能不需要改变源代码实现

1. IOC

控制反转，之前创建对象要用new语句，现在创建实体类对象只需要交给Spring来通过配置创建就可以了

1. Spring是一站式的框架
2. Spring在Javaee的三层结构中，都提供了不同的解决技术。

--Web层：SpringMVC，用在web层

--Service层：Spring的IOC层

--DAO层：Spring的jdbcTemplate

4.Spring版本

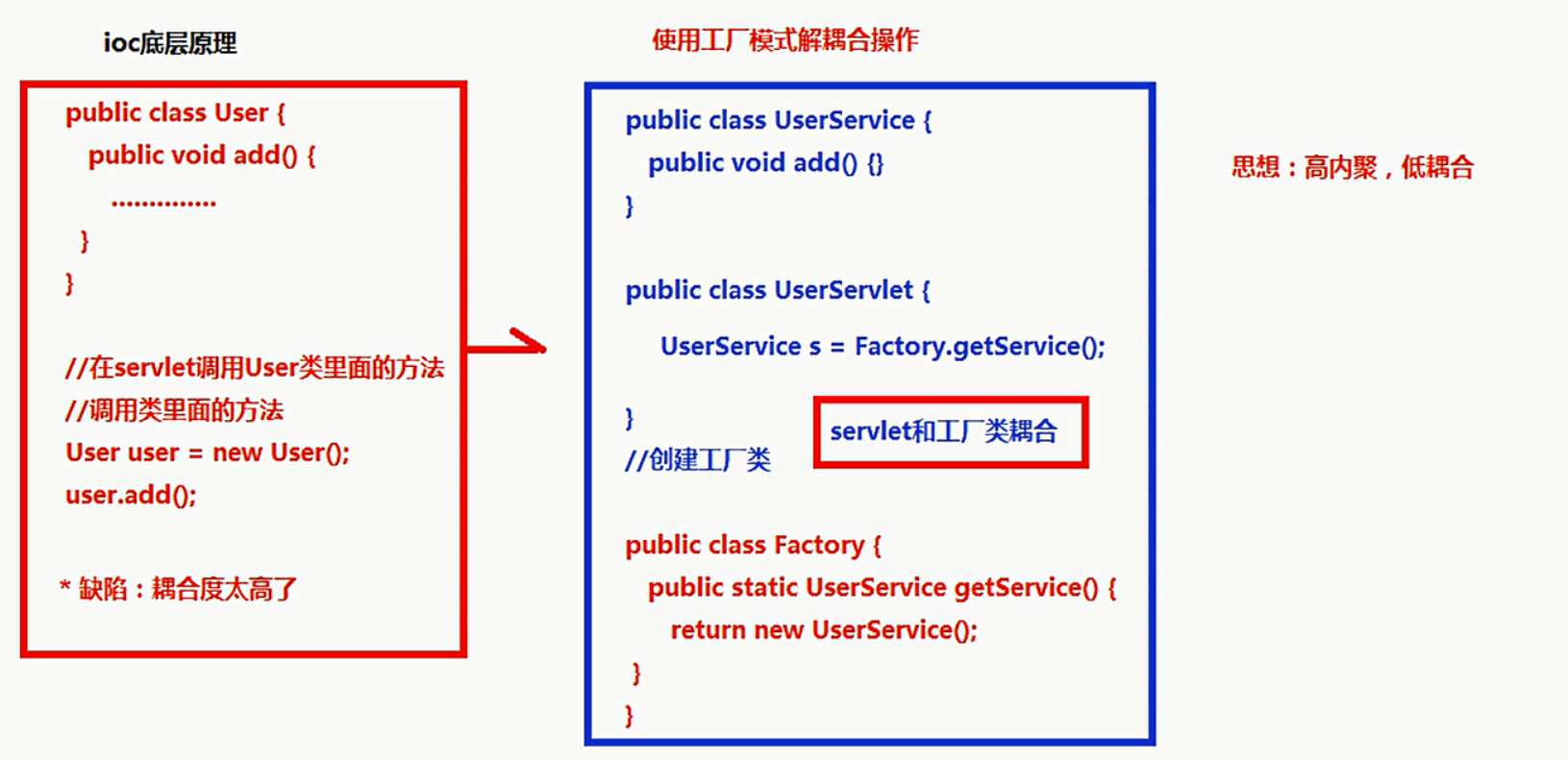
## Spring的IOC操作

1. 把对象的创建交给Spring进行管理
2. IOC有两部分操作
3. IOC的配置文件方式实现
4. IOC基于注解的方式实现

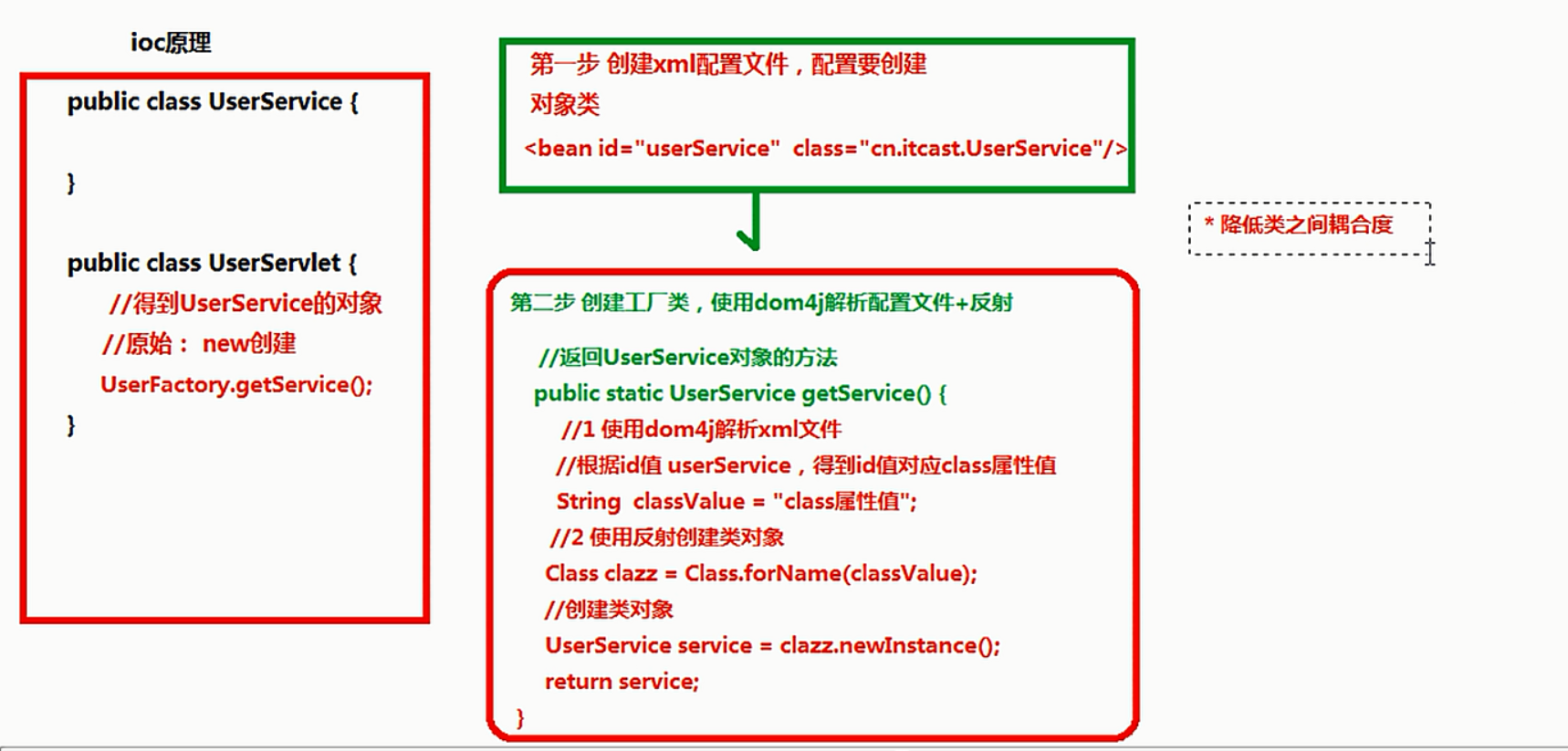
## IOC的底层原理

1. IOC的底层使用的技术
2. xml配置文件
3. dom4j解析xml
4. 工厂设计模式
5. 反射
6. 画图实现IOC的原理

正常的实现原理：



IOC的底层实现原理（重要，主要用的是反射机制，加工厂和配置文件）：



1. IOC入门案例
2. 导入JAR包

最基本功能的核心JAR包：beans、context、core、spEL

导入支持日志输出的JAR包

1. 创建一个类，在类中创建一个方法

**public** **class** User {

**public** **void** add(){

System.***out***.println("add....");

}

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

User user=**new** User();

user.add();

}

}

1. 创建Spring配置文件，配置创建类
2. Spring核心配置文件的位置不是固定的，建议放到src下边，官方建议applicationContext.xml
3. 引入约束（Schema约束）
4. 配置对象的创建

<!-- ioc入门 -->

<bean id=*"user"* class=*"com.wbs.domain.User"*>

1. 测试对象创建

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

//1加载Spring配置文件，根据配置创建对象

ApplicationContext context=**new** ClassPathXmlApplicationContext("bean1.xml ");

//2得到配置创建的对象

User user=(User)context.getBean("user");

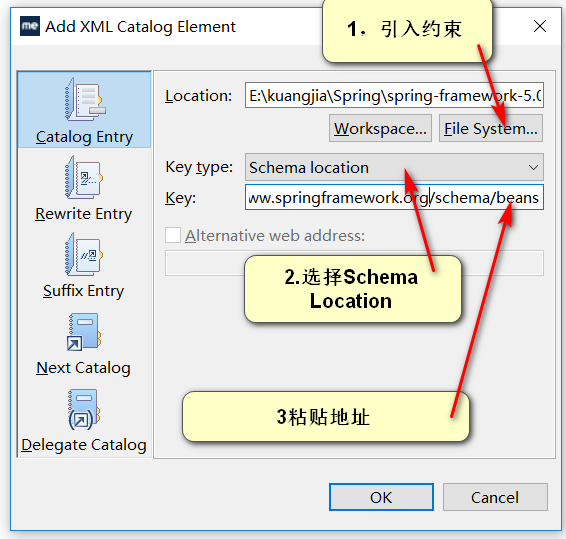
System.***out***.println(user);

user.add();

}

1. 配置文件没有提示
2. Spring引入的约束为Schema约束，把约束文件引入到MyEclipse中

复制约束的路径



# Spring的Bean管理（xml配置文件）

bean实例化的方式

1. 在Spring中通过配置文件创建对象
2. Bean实例化有三种方式

第一种：使用类中的参数构造创建（常用）

**public** User() {}

第二种：通过静态工程创建（不常用）

创建静态的方法，返回类的实例对象

<!-- 使用静态工程来创建对象 -->

<bean id=*"bean01"* class=*"com.wbs.bean.bean01factory"* factory-method=*"getBean01"*/>

但是此种方法还需要写出静态方法来创建改类的对象，如下：

**public** **class** bean01factory {

//静态的方法，返回bean01对象

**public** **static** Bean01 getBean01(){

**return** **new** Bean01();

}

}

其调用的方法与第一种方法是一样的

**public** **void** test2(){

ApplicationContext ac=**new** ClassPathXmlApplicationContext("bean1.xml");

Bean01 bean01=(Bean01)ac.getBean("bean01");

System.***out***.println(bean01);

//bean01.add();

}

第三种：通过实例工厂创建

创建一个不是静态的方法，返回类对象

<!-- 实例工厂创建 -->

<!-- 先创建Bean02Factory对象 -->

<bean id=*"bean02factory"* class=*"com.wbs.bean.Bean02Factory"*></bean>

<bean id=*"bean02"* factory-bean=*"bean02factory"* factory-method=*"getBean02"*></bean>

其中实例工厂的方法中返回一个new 的对象，如下：

**public** **class** Bean02Factory {

//普通的方法

**public** bean02 getBean02(){

**return** **new** bean02();

}

}

测试代码；

public void test3(){

ApplicationContext ac=new ClassPathXmlApplicationContext("bean1.xml");

/\*Bean02Factory f=(Bean02Factory)ac.getBean("bean02factory");

System.out.println(f);\*/

bean02 b=(bean02)ac.getBean("bean02");

System.out.println(b);

b.add();

}

## Bean标签的常用属性

1. Id:起的名称，可以任意命名

-id 的属性值，不能包含特殊的符号

-可以根据id值得到配置对象

1. Class：创建对象所在类的全路径
2. Name：功能和id属性是一样的，但是name属性中可以包含特殊的符号
3. Scrop：范围

-singleton默认的,单例的，

-prototype 多例的

-request：创建对象，把对象放到request中

-session：创建对象，把对象放到session中

-globalSession：创建对象，把对象放到globalSession中

# 属性注入

1. 创建对象的时候，可以向类的属性中设置值
2. 属性注入的三种方式

在JAVA的方式功三种：

第一种方式：用set方式注入

public Class User {

private String name;

public void setName(String name){

this.name=name;

}

}

User user=new User();

User.setName(“xxx”);

第二种方式：有参构注入

public class User{

privaete String name;

public User(String name){

this.name=name;

}

}

User user=new User(“xxx”);

第三种方式：使用接口注入

public interface Dao{

Public void delete(String name);

}

public class DalTest implements Dao{

String name;

public void delete(String name){

this.name=name;

}

}

1. 在Spring框架里面，只支持前两种注入方式

-set方式

-有参构造函数注入方式

## 使用有参构造注入

配置文件：

<!-- 使用有参构造函数注入 -->

<bean id=*"demo"* class=*"com.wbs.property.PropertyDemo1"* >

<!-- 使用构造参数注入 name：属性值；value是要设置的值-->

<constructor-arg value=*"博盛"*></constructor-arg>

<constructor-arg value=*"15"*></constructor-arg>

</bean>

实体类：

**public** **class** PropertyDemo1 {

String username;

String age;

**public** PropertyDemo1(String username,String age) {

**this**.username = username;

**this**.age=age;

}

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

System.***out***.println("Demo1="+username+"age="+age);

}

}

测试类：

**public** **void** test4(){

ApplicationContext ac=**new** ClassPathXmlApplicationContext("bean1.xml");

PropertyDemo1 pd=(PropertyDemo1)ac.getBean("demo");

pd.test1();

}

## 使用set方式注入

配置：

<!-- 使用set方法注入属性 -->

<bean id=*"book"* class=*"com.wbs.property.Book"*>

<!-- 注入属性值

name:是属性的名称

value：设置具体的值

-->

<property name=*"bookname"* value=*"java编程思想"*></property>

实体类：

**public** **class** Book {

**private** String bookname;

**public** String getBookname() {

**return** bookname;

}

**public** **void** setBookname(String bookname) {

**this**.bookname = bookname;

}

**public** **void** demoBook(){

System.***out***.println("bookname="+bookname);

}

}

测试：

**public** **void** test5(){

ApplicationContext ac=**new** ClassPathXmlApplicationContext("bean1.xml");

Book book=(Book)ac.getBean("book");

book.demoBook();

}

## 注入对象类型的属性（重点）

1. 创建一个service和dao类

正常的方式是：

1. 在service中得到dao的对
2. 具体实现过程（Spring中的方法）
3. 在service里面把dao作为属性类型
4. 生成dao类型的set方法

**public** **class** UserService {

//定义一个dao类型的属性

**private** UserDao userdao;

**public** **void** setUserdao(UserDao userdao) {

**this**.userdao = userdao;

}

**public** **void** add(){

System.***out***.println("service.....");

//在service里面得到dao对象才可以调用dao里面的方法

/\*UserDao userdao=new UserDao();

userdao.add();\*/

}

}

1. 在配置文件中注入关系

<!-- 注入对象类型的属性 -->

<!-- 1配置service和dao对象 -->

<bean id=*"userdao"* class=*"com.wbs.dao.UserDao"*></bean>

<bean id=*"userservice"* class=*"com.wbs.dao.UserService"*>

<!-- 注入dao对象

name:是userservice中的dao对象的一个属性名称

现在不能写value，因为value是一个String类型，而现在的这个属性是一个dao对象的属性

应该使用ref属性

ref:写dao配置中bean标签中的id

-->

<property name=*"userdao"* ref=*"userdao"*></property>

</bean>

注意：上边的黄色标出的下边的ref中的是从上边的那儿来的，即将usedao注入到userservice中。上边的配置中值注入了一个属性，如果有很多的属性的话，可以注入很多的属性。

## P名称空间的注入（会用就可以）

1. 名称空间：

xmlns:p=[*http://www.springframework.org/schema/p*](http://www.springframework.org/schema/p)

*需要在使用的时候引入上边的这句话*

*才能使用p名称空间的注入*

<!-- p名称空间注入 -->

<bean id=*"person"* class=*"com.wbs.property.Person"* p:pname=*"博盛"*></bean>

这个效果和set注入是一样的，同样要在实体类中写上属性的set方法，如下：

**public** **class** Person {

**private** String pname;

**public** **void** setPname(String pname) {

**this**.pname = pname;

}

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

System.***out***.println("persion name is "+pname);

}

}

## Spring注入一些复杂类型属性

1. 数组
2. List集合
3. Map集合
4. Properties类型

配置示例：

<!-- Spring中注入复杂类型 -->

<bean id=*"person1"* class=*"com.wbs.property.Person"*>

<!-- 数组 -->

<property name=*"arrs"*>

<list>

<value>篮球</value>

<value>排球</value>

<value>足球</value>

</list>

</property>

<!-- List注入 -->

<property name=*"list"*>

<list>

<value>一月</value>

<value>二月</value>

<value>三月</value>

</list>

</property>

<!-- Map注入 -->

<property name=*"map"*>

<map>

<entry key=*"aa"* value=*"AAAA"*></entry>

<entry key=*"bb"* value=*"BBBB"*></entry>

<entry key=*"cc"* value=*"CCCC"*></entry>

</map>

</property>

<!-- properties注入 -->

<property name=*"properties"*>

<props>

<prop key=*"driverclass"*>com.mysql.jdbc.Driver</prop>

<prop key=*"username"*>root</prop>

</props>

</property>

</bean>

此种方法只要会用就可以了。

# IOC和DI的区别

1. IOC

控制反转，把对象的创建交给Spring进行配置

1. DI

依赖注入，向类里面的属性中设置值

3.IOC和DI之间的关系，依赖注入（DI）不能单独存在，需要在IOC的基础之上完成操作

# Spring整合web项目

1. 加载Spring核心配置文件

//1加载Spring配置文件，根据配置创建对象

ApplicationContext context=**new**ClassPathXmlApplicationContext("bean1.xml");

这个加载文件每次需要new一个对象，效率很低

1. 解决办法

把加载文件和创建对象的过程，交给服务器启动的时候完成

1. 实现原理
2. setvletContext
3. 监听器
4. 具体使用

-a.在服务器启动的时候，为每个项目创建一个ServletContext对象

-b.在ServletContext创建的时候，使用监听器可以具体监听到ServletContext对象在什么时候创建

---1使用监听器监听到ServletContext对象创建的时候，加载Spring的配置文件，把配置文件的配置的对象进行创建

---2把创建出来的对象放到ServletContext域对象中（setAttribute方法）

-c.获取对象，得到ServletContext对象（getAttribute()方法）

注意：这个Spring已经封装好了的，我们只需要配置就可以了，原理需要掌握。

具体做法：

1.Spring帮我们封装了一个监听器，只需要配置监听器就可以了。

2.在配置监听器之前，需要导入一个Spring整合web项目的JAR包



Web.xml配置：

<!-- 配置监听器 -->

<listener>

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

</listener>

3.指定spring加载配置文件的位置(web.xml)

<!-- 指定spring配置文件的位置 -->

<context-param>

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

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

</context-param>

# Spring的Bean管理

## 注解介绍

1. 代码里面特殊的标记，使用注解可以完成相关的功能
2. 写法：@注解名称（属性名=属性值）
3. 可以用在类、方法、属性上边

## Spring注解开发准备

1. **导入基本的JAR包**

****

**导入AOP的JAR包：**

****

**2．创建类，创建方法**

1. **创建配置文件，引入约束**

**做Spring的IOC的注解的开发，需要引入新的约束（Context约束）**

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

**开启注解的扫描**

<!-- 配置开启注解的扫描,base-package；是创建对象所在类的包的名字

1.到包里扫描类，方法，属性，上面是否有注解

-->

<!-- 方法1 -->

<!-- <context:component-scan base-package="com.wbs.domain,com.wbs.service"></context:component-scan> -->

<!-- 方法2，建议使用-->

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

<!-- 只扫描属性上边的注解 -->

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

## 注解创建对象

1.创建对象有四个注解：

@Component

@Cotroller web层

@Services 业务层

@Repository 持久层

这四个注解的功能是一样的，都是创建对象的。

@Component(value="user")//相当于<bean id="user" class=""/>

**public** **class** User {

**public** **void** add(){

System.***out***.println("user中的add()...........");

}

}

1. 创建对象是单实例还是多实例

@Scope(value="prototype")

在类上边加上这句话，来配置多实例或者单实例

Prototype:多实例

Singletoin:单实例

## 注解注入属性

1. 创建service类，创建dao类，在service中得到dao对象
2. 注入属性的注解

@Autowired 自动注入

@Autowired//自动注入，是根据类名来寻找的

**private** UserDao userdao;//使用注解的时候不需要set方法

@Resource name属性是要注入的dao对象的属性中的value值(常用)

@Resource(name="userdao")//name属性是注解方式创建的dao对象的value值

UserService对象：

@Component(value="userservice")//value可以不写

**public** **class** UserService {

//得到dao对象

//1在service中创建一个Dao类型的对象，在dao的属性上边使用注解完成对象注入

@Autowired//自动注入，是根据类名来寻找的

**private** UserDao userdao;//使用注解的时候不需要set方法

//@Resource(name="userdao")//name属性是注解方式创建的dao对象的value值

**private** UserDao userdao;//使用注解的时候不需要set方法

**public** **void** add(){

System.***out***.println("service 中的add().....");

userdao.add();

}

}

UserDao对象：

@Component(value="userdao")

**public** **class** UserDao {

**public** **void** add(){

System.***out***.println("userDao 中的add()...");

}

}

## 配置文件和注解混合使用

1. 创建对象使用配置文件方式实现

<bean id=*"bookservice"* class=*"com.wbs.xmlanoo.BookService"*></bean>

<bean id=*"bookdao"* class=*"com.wbs.xmlanoo.BookDao"*></bean>

<bean id=*"orderdao"* class=*"com.wbs.xmlanoo.OrderDao"*></bean>

1. 注入属性的操作使用注解方式实现（提前写了两个dao类）

**public** **class** BookService {

//得到Bookdao和OrderDao

@Resource(name="bookdao")

**private** BookDao bookdao;

@Resource(name="orderdao")

**private** OrderDao orderdao;

**public** **void** add(){

System.***out***.println("service");

bookdao.book();

orderdao.buy();

}

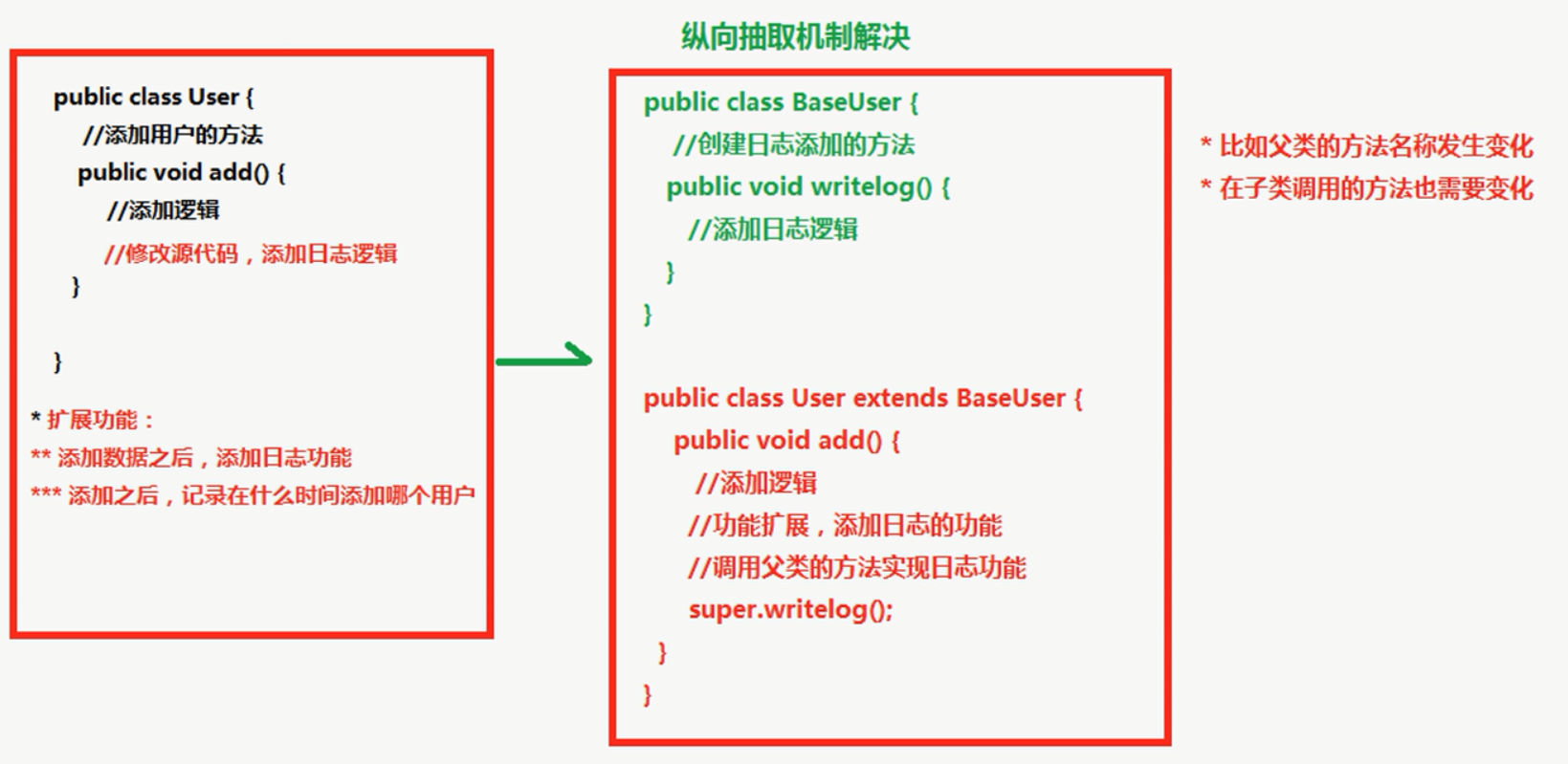
}

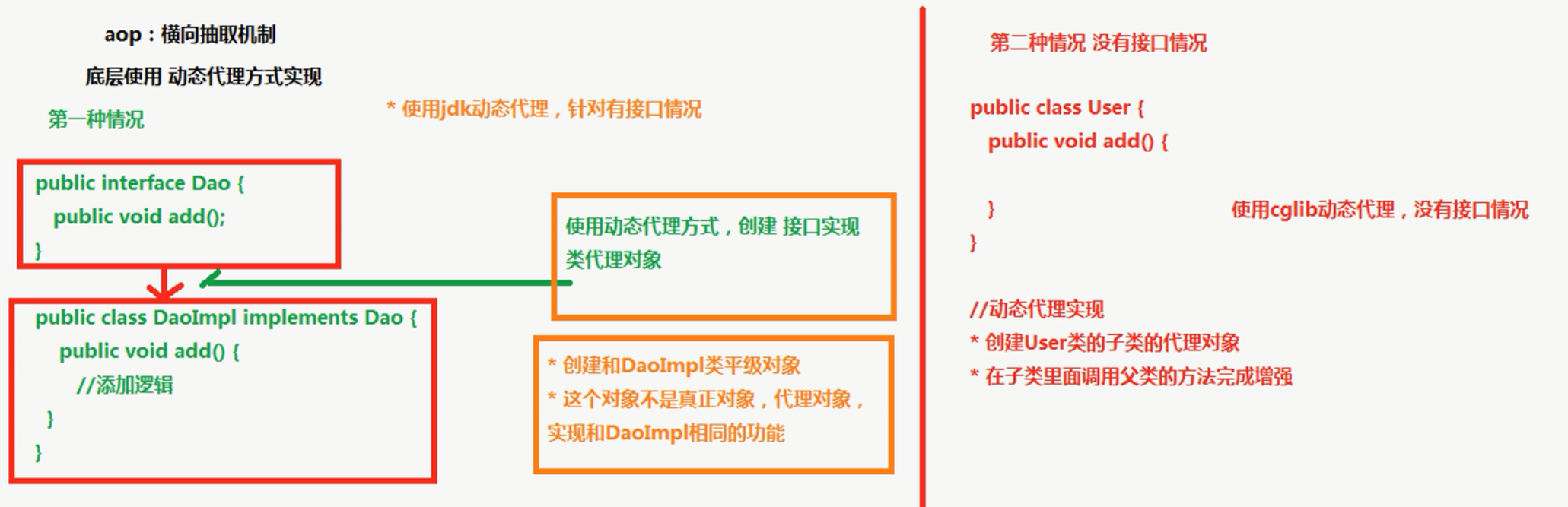
# AOP

## AOP概念

1. 面向切面编程，扩展一个功能不通过源代码实现
2. AOP采用横向抽取机制，取代了传统纵向继承体系重复性代码

## AOP原理



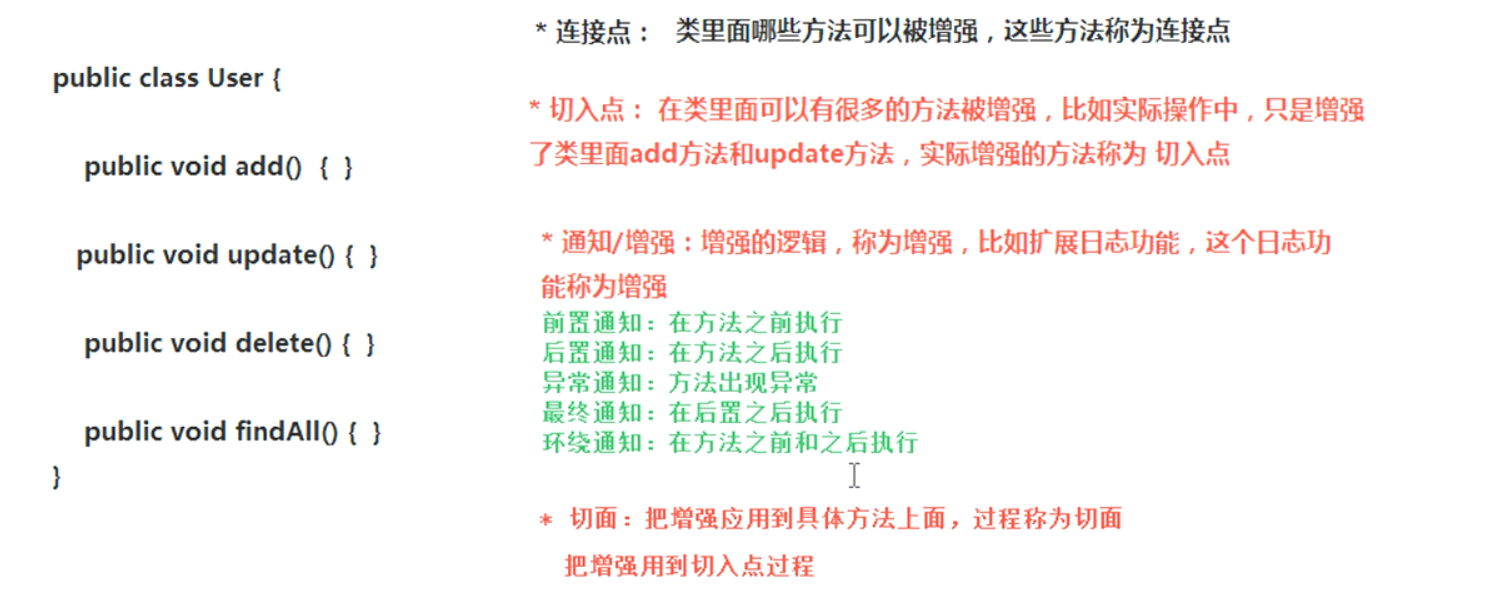


## AOP操作术语

Joinpoint 连接点，类里面可以被增强的方法，称为连接点

Pointcut 切入点，类中实际增强的方法

Advice 通知（增强）实际增强的逻辑，比如扩展日志功能，这个日志功能称为增强



Aspect 切面，把增强用到切入点的过程

Introduction 引介，在运行期间动态地添加一些方法

Target 目标对象，增强方法所在的类

Weaving 织入

Proxy 代理

## Spring的AOP操作

1. 在Spring里面，进行AOP操作，使用aspectJ实现

a.AspectJ不是Spring的一部分，和Spring一起进行AOP操作

b.Spring2.0之后增加了对aspectJ的支持

2.AOP的准备工作

a.导入基本JAR包，还需要导入AOP相关的JAR包





1. 创建基本配置文件，引入AOP约束

<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 definitions here -->

</beans>

## 使用表达式配置切入点

1. 切入点：实际增强的方法
2. 常用的表达式

Execute(<访问修饰符>?<返回类型><方法名>（<参数>）<异常>)

1).execution(\* com.wbs.aop.Book.add(..))

2). execution (\* com.wbs.aop.Book.\*(..))

3). execution (\* \*.\*(..))

4). execution (\* save\*(..)) 匹配所有save开头的方法

注意：前面的\*与后边的类路径之间有一个空格，前面的第一个\*表示对所有的访问类型进行增强（如private，public ，protected）

# 使用aspectJ实现AOP有两种方式

## 基于aspectJ的AOP的xml配置

<!-- 1配置对象 -->

<bean id=*"book"* class=*"com.wbs.aop.Book"*></bean>

<bean id=*"mybook"* class=*"com.wbs.aop.MyBook"*></bean>

<!--2 配置AOP操作 -->

<aop:config>

<!-- 1，配置切入点 ,id:给切入点起名；expression:表达式的形式-->

<aop:pointcut expression=*"execution(\* com.wbs.aop.Book.\*(..))"* id=*"pointcut1"*/>

<!-- ,2，配置切面 ,把增强用到方法上边-->

<aop:aspect ref=*"myBook"*>//ref是增强的对象

<!-- 配置增强的类型

如下是前置增强

method；增强类里面哪个方法作为前置增强

pointcut-ref；切入点引用，指的是用的是哪个切入点

-->

<aop:before method=*"before1"* pointcut-ref=*"pointcut1"*/>

</aop:aspect>

<!-- 配置后置增强 -->

<aop:after-returning method=*"after1"* pointcut-ref=*"pointcut1"*/>

<!-- 环绕增强 -->

<aop:around method=*"around1"* pointcut-ref=*"pointcut1"*/>

</aop:config>

测试对象及其方法

//增强类

**public** **class** MyBook {

//前置方法

**public** **void** before1(){

System.***out***.println("前置增强。。。。。");

}

//后置方法

**public** **void** after1(){

System.***out***.println("后置。。。。。");

}

//环绕通知

**public** **void** around1(ProceedingJoinPoint proceedingJoinPoint) **throws** Throwable{

//ProceedingJoinPoint可以调用被增的方法（如add()）

//方法之前执行

System.***out***.println("方法之前执行");

//执行被增强的方法

proceedingJoinPoint.proceed();

//方法之后执行

System.***out***.println("方法之后执行");

}

}

被增强类

**public** **class** Book {

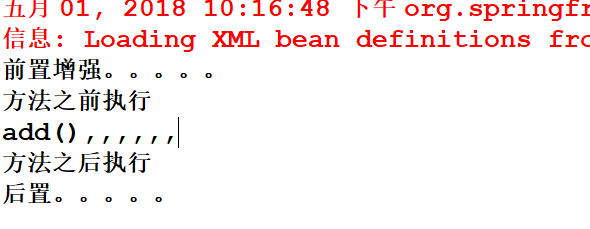
**public** **void** add(){

System.***out***.println("add(),,,,,,");

}

}

测试结果：



## 基于aspectJ的注解的方式实现

# Log4J介绍

1.可以看到程序运行中更详细的信息

经常使用log4J查看日志信息

2.使用；

（1）导入Log4J的JAR包

（2）复制Log4J的配置文件到src下边

3.设置日志级别

1.info:看到基本信息

2.debug:看到更详细的信息

# 基于aspectj的注解的 AOP

1使用注解实现AOP操作

第一步：创建对象

<!-- 创建文件 -->

<bean id=*"book"* class=*"com.wbs.aop.Book"*></bean>

<bean id=*"mybook"* class=*"com.wbs.aop.MyBook"*></bean>

第二步：在spring的核心配置文件中，开启AOP操作

<!-- 开启AOP操作 -->

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

第三步：注解

@Aspect

**public** **class** MyBook {

@Before(value="execution(\* com.wbs.aop.Book.\*(..))")

**public** **void** before1(){

System.***out***.println("before1......");

}

}

## Spring的JDBC模板操作

1. Spring是一站式框架
2. 针对javaee的三层，每一层都有解决技术
3. 在dao层，使用jdbcTemplate
4. Spring对不同的持久化作了不同的封装
5. jdbcTemplate对jdbc进行封装
6. jdbcTemplate模板的使用和dbUtils使用很相似，都是对数据库进行crud操作

## 具体的使用

1. 导入jar包



2创建对象，设置数据库的信息

1. 创建jdbcTemplate对象，设置数据源
2. 调用jdbcTemplate对象里的方法实现操作

### 增加

//增加

@Test

**public** **void** add(){

//1.设置数据库信息

DriverManagerDataSource datesource=**new** DriverManagerDataSource();

datesource.setDriverClassName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

datesource.setUrl("jdbc:sqlserver://localhost:1433;DatabaseName=jdbcTemplate");

datesource.setUsername("sa");

datesource.setPassword("wbs19950305");

//2.创建jdbcTemplate对象，设置数据源

jdbcTemplate=**new** JdbcTemplate(datesource);

//3.使用jdbcTemplate里面的方法实现操作

//创建sql语句

String sql="insert into users values(?,?)";

**int** row=jdbcTemplate.update(sql,"Lucy","520");//返回成功的行数

System.***out***.println("row="+row);

}

### 删除

//删除

@Test

**public** **void** delete(){

DriverManagerDataSource datesource=**new** DriverManagerDataSource();

datesource.setDriverClassName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

datesource.setUrl("jdbc:sqlserver://localhost:1433;DatabaseName=jdbcTemplate");

datesource.setUsername("sa");

datesource.setPassword("wbs19950305");

//2.创建jdbcTemplate对象，设置数据源

JdbcTemplate jdbcTemplate=**new** JdbcTemplate(datesource);

//3.使用jdbcTemplate里面的方法实现操作

//创建sql语句

String sql="delete from users where username=?";

**int** row=jdbcTemplate.update(sql,"Lucy");

System.***out***.println(row);

}

### 修改

//修改

@Test

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

DriverManagerDataSource datesource=**new** DriverManagerDataSource();

datesource.setDriverClassName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

datesource.setUrl("jdbc:sqlserver://localhost:1433;DatabaseName=jdbcTemplate");

datesource.setUsername("sa");

datesource.setPassword("wbs19950305");

//2.创建jdbcTemplate对象，设置数据源

JdbcTemplate jdbcTemplate=**new** JdbcTemplate(datesource);

//3.使用jdbcTemplate里面的方法实现操作

//创建sql语句

String sql="update users set password=? where username=?";

**int** row=jdbcTemplate.update(sql,"1314","Lucy");

System.***out***.println(row);

}

### 查询

1.查询的思想

/\*\*

\* DBUtils

\* QueryRunner runner=new QuetyRunner("datesource");

\*若返回的是一个对象

\*runner.query(sql,new Handle<User>(User.Class));

\*

\*若返回List集合

\*runner.query(sql,new BeanListHandler<>());

\*

\*1.在DBUtils有接口ResultSetHandler

\*dbutils提供了针对不同的结果实现类

\*

\*2，jdbcTemplate实现查询，有接口RowMapper

\*jdbcTemplate针对这个模板没有提供实现类，得到不同的数据类型需要自己封装

\*

\*/

2，具体操作

1. 查询返回某一个值

注意：queryForObject()中的两个参数，第一个是sql语句，第二个是要返回的类型的Class

//1查询有多少条记录

@Test

**public** **void** testCount(){

DriverManagerDataSource datesource=**new** DriverManagerDataSource();

datesource.setDriverClassName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

datesource.setUrl("jdbc:sqlserver://localhost:1433;DatabaseName=jdbcTemplate");

datesource.setUsername("sa");

datesource.setPassword("wbs19950305");

//2.创建jdbcTemplate对象，设置数据源

JdbcTemplate jdbcTemplate=**new** JdbcTemplate(datesource);

//3.使用jdbcTemplate里面的方法实现操作

//创建sql语句

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

//调用jdbcTemplate模板中的方法实现

**int** count=jdbcTemplate.queryForObject(sql, Integer.**class**);

System.***out***.println("count="+count);

}

1. 查询返回一个对象

1，，JDBC的原始实现代码：

//jdbc底层实现代码

@Test

**public** **void** testJDBC(){

Connection conn = **null**;

PreparedStatement ps = **null**;

ResultSet rs = **null**;

//加载驱动

**try** {

Class.*forName*("com.microsoft.sqlserver.jdbc.SQLServerDriver");

//创建链接

conn=DriverManager.*getConnection*("jdbc:sqlserver://localhost:1433;

DatabaseName=jdbcTemplate;","sa","wbs19950305");

//编写sql语句

String sql="select \* from users where username=?";

//预编译sql

ps=conn.prepareStatement(sql);

ps.setString(1, "Lucy");

//执行sql

rs=ps.executeQuery();

//遍历结果集

**while**(rs.next()){

//得到返回的结果的值

String username=rs.getString("username");

String pwd=rs.getString("password");

//将这两个值放入user中

User user= **new** User();

user.setPassword(pwd);

user.setUsername(username);

System.***out***.println(user);

}

} **catch** (Exception e) {

e.printStackTrace();

}**finally** {

**try** {

rs.close();

conn.close();

ps.close();

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

2，，使用模板完成

//查询返回对象

@Test

**public** **void** testObject(){

DriverManagerDataSource datesource=**new** DriverManagerDataSource();

datesource.setDriverClassName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

datesource.setUrl("jdbc:sqlserver://localhost:1433;DatabaseName=jdbcTemplate");

datesource.setUsername("sa");

datesource.setPassword("wbs19950305");

//2.创建jdbcTemplate对象，设置数据源

JdbcTemplate jdbcTemplate=**new** JdbcTemplate(datesource);

//3.使用jdbcTemplate里面的方法实现操作

//创建sql语句

String sql ="select \* from users where username=?";

//调用jdbcTemplate模板中的方法来实现,第二个参数是接口RowMapper，需要自己写一个类，实现该接口

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

System.***out***.println(user);

}

注意：queryForObject(sql, **new** MyRowMapper(), "Mac")这个方法中有三个参数，第一个是sql 语句，第二个是一个实现了RowMapper接口的一个类（Spring帮我们在这个接口中实现了一部分的封装），第三个参数是给查询的sql语句赋值，特别要注意的是第二个参数的写法，其具体的类（MyRowMapper）如下：

**public** **class** MyRowMapper **implements** RowMapper<User> {

//手动封装，spring帮我们封装了一部分

@Override

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

//1.从结果集把数据拿到

String username=rs.getString("username");

String pwd=rs.getString("password");

//2.把结果集封装到对象里

User user=**new** User();

user.setPassword(pwd);

user.setUsername(username);

**return** user;

}

}

1. 查询返回一个List集合

//查询返回List

@Test

**public** **void** testList(){

DriverManagerDataSource datesource=**new** DriverManagerDataSource();

datesource.setDriverClassName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

datesource.setUrl("jdbc:sqlserver://localhost:1433;DatabaseName=jdbcTemplate");

datesource.setUsername("sa");

datesource.setPassword("wbs19950305");

//2.创建jdbcTemplate对象，设置数据源

JdbcTemplate jdbcTemplate=**new** JdbcTemplate(datesource);

//3.使用jdbcTemplate里面的方法实现操作

//创建sql语句

String sql ="select \* from users ";

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

System.***out***.println(list);

}

## Spring配置c3p0连接池和dao使用jdbcTemplate

1. Spring配置c3p0连接池

第一步：导入JAR包



其中c3p0代码的事项方式如下：

ComboPooledDataSource datasource=**new** ComboPooledDataSource();

datasource.setDriverClass("com.microsoft.sqlserver.jdbc.SQLServerDriver");

datasource.setJdbcUrl("jdbc:sqlserver://localhost:1433;DatabaseName=jdbcTemplate");

datasource.setUser("sa");

datasource.setPassword("wbs19950305");

第二步：把代码的c3p0实现用配置文件的方式实现

创建Spring的配置文件

<!-- 配置c3p0连接池 -->

<bean id=*"DataSource"* class=*"om.mchange.v2.c3p0.ComboPooledDataSource"*>

<!-- 注入里面的属性值 -->

<property name=*"driverClass"* value=*"com.microsoft.sqlserver.jdbc.SQLServerDriver"*></property>

<property name=*"jdbcUrl"* value=*"jdbc:sqlserver://localhost:1433;DatabaseName=jdbcTemplate"*></property>

<property name=*"user"* value=*"sa"*></property>

<property name=*"password"* value=*"wbs19950305"*></property>

</bean>

2. 在dao里使用jdbcTemplate

1. 创建service和dao，配置service和dao对象，在service中注入dao对象
2. 创建jdbcTemplate对象，把模板注入到dao对象里面
3. 在jdbcTemplate对象那个中注入DataSource，因为在源代码中有”dataSource”这个属性，并且生成了它的set方法，所以直接注入这个”dataSource”就可以了。

<!-- 1.创建service和dao对象，在service中注入dao对象 -->

<bean id=*"userservice"* class=*"com.wbs.c3p0.UserService"*>

<property name=*"userdao"* ref=*"userdao"*></property>

</bean>

<bean id=*"userdao"* class=*"com.wbs.c3p0.UserDao"*>

<!-- 注入jdbcTemplate模板对象 -->

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

</bean>

<!-- 创建jdbcTemplate对象 -->

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

<!-- 把DataSource传递到模板的对象里 -->

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

</bean>

其中测试代码如下：

@Test

**public** **void** test(){

ApplicationContext context=**new** ClassPathXmlApplicationContext("bean1.xml");

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

us.add();

}

成功添加一条数据到表中。

# Spring的事务管理

## 事务概念

1. 什么是事务？
2. 事务特性

原子性、一致性、隔离性、持久性

1. 不考虑隔离性产生读问题
2. 解决读问题

设置隔离级别

## Spring的事务管理API

1. Spring事务管理的两种方式

第一种：编程是事务管理（不用）

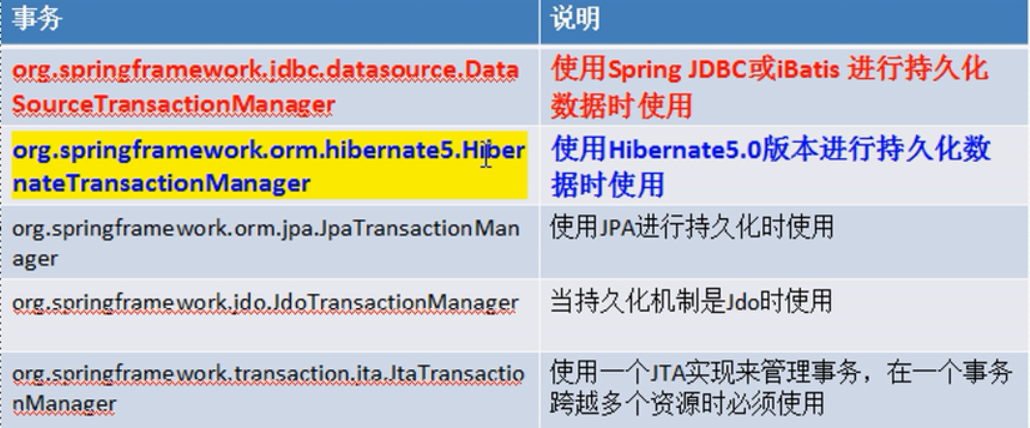
第二种：声明式事务管理

1. xml配置文件事项
2. 注解方式实现

2.Spring事务管理的API

接口：platformTransactionManager 事务管理器

1，spring针对不同的dao层框架，提供了不同的实现类



2，首先配置事务管理器

### 搭建转账的环境：

1. 创建数据库表，添加数据
2. 创建service和dao 完成注入关系
3. Service层（业务逻辑层）
4. Dao层（单独对数据库操作层），dao层不添加业务
5. 需求，小王转账50给小李

先导入约束，建立xml文件

<?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"*

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/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*

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

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

<!-- 配置c3p0连接池 -->

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

<!-- 注入里面的属性值 -->

<property name=*"driverClass"* value=*"com.microsoft.sqlserver.jdbc.SQLServerDriver"*></property>

<property name=*"jdbcUrl"* value=*"jdbc:sqlserver://localhost:1433;DatabaseName=jdbcTemplate"*></property>

<property name=*"user"* value=*"sa"*></property>

<property name=*"password"* value=*"wbs19950305"*></property>

</bean>

<bean id=*"orderservice"* class=*"com.wbs.service.OrderService"*>

<property name=*"orderdao"* ref=*"orderdao"*></property>

</bean>

<bean id=*"orderdao"* class=*"com.wbs.dao.OrderDao"*>

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

</bean>

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

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

</bean>

</beans>

创建dao

**public** **class** OrderDao {

//注入jdbcTemplate

**private** JdbcTemplate jdbcTemplate;

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

/\*\*

\* 在dao中不写业务，只写对数据操作的方法

\*/

//小王少钱的方法

**public** **void** lessMoney(){

String sql="update acount set salary=salary-? where username=?";

jdbcTemplate.update(sql,50,"小王");

}

//多钱的方法

**public** **void** moreMoney(){

String sql="update acount set salary=salary-? where username=?";

jdbcTemplate.update(sql,-50,"小李");

}

}

创建service

**public** **class** OrderService {

**private** OrderDao orderdao;

**public** **void** setOrderdao(OrderDao orderdao) {

**this**.orderdao = orderdao;

}

//调用dao的方法

//业务逻辑，写转账方法

**public** **void** accountMoney(){

//小王少50

orderdao.lessMoney();

//小李多50

orderdao.moreMoney();

}

}

测试：

@Test

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

ApplicationContext context=**new** ClassPathXmlApplicationContext("bean1.xml");

OrderService orderservice=(OrderService)context.getBean("orderservice");

orderservice.accountMoney();

}

}

注意：在写代码的过程中没有加事务，会产生一些问题

3.产生的问题

//调用dao的方法

//业务逻辑，写转账方法

**public** **void** accountMoney(){

//小王少50

orderdao.lessMoney();

//抛出异常

**int** i=3/0;

//小李多50

orderdao.moreMoney();

}

当小王少了50之后，产生了异常之后，小马不会再多50，改怎么办?

4.解决办法：添加事务解决，当出现异常时候，进行事务的回滚操作。

事务的添加方式

### 声明式事务管理（Xml配置）

1. 配置文件使用AOP思想

第一步：配置事务管理器

<!-- 配置事务管理器 -->

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

<!-- 注入dataSource ,ref是连接池的数据源 -->

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

</bean>

第二步：配置事务增强

<!-- 配置事务的增强,transaction-manager的值必须和事务管理器的id值一样 -->

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

<!-- 做什么事务操作 -->

<tx:attributes>

<!-- 设置进行事务操作的方法匹配规则 ，name是对哪些方法进行增强，比如这个配置是对service中的account\*方法进行事务增强propagation是事务隔离级别，不写也可以-->

<tx:method name=*"account\*"* propagation=*"REQUIRED"*/>

<!-- <tx:method name="insert\*"/> -->

</tx:attributes>

</tx:advice>

### 

第三步：配置切面

<!-- 3.配置切面 -->

<aop:config>

<!-- 切入点 -->

<aop:pointcut expression=*"execution(\* com.wbs.service.OrderService.\*(..))"* id=*"pointcur1"*/>

<!-- 切面,advice-ref是增强来自，必须和增强配置的id一样;pointcut-ref是切面来自，必须和切面的id一致 -->

<aop:advisor advice-ref=*"txadvice"* pointcut-ref=*"pointcur1"*/>

</aop:config>

### 声明式事务管理《注解》

第一步：配置事务管理器

<!-- 第一步：配置事务管理器 -->

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

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

</bean>

第二步：配置事务注解

<!-- 第二步：开启事务的注解 -->

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

第三步：要在使用事务的方法所在的类的上边添加注解

@Transactional

**public** **class** OrderService {

原理：开启事务之后，会在类中找@transactional注解，然后在这个类的所有方法都加上事务。