永和大王门店管理系统

# 永和大王门店管理系统

## 项目开发背景

永和大王是全国连锁快餐企业，自1995年12月在上海市开设第一家餐厅以来，经过24年的发展，已经在中国大陆开设了近300家直营连锁餐厅，所经营的品均是符合中国人的口味的豆浆、油条、各种稀饭、面食以及中式点心，并坚持以直营方式扩张获得了稳定、标准、一致的发展，深受大众喜爱。

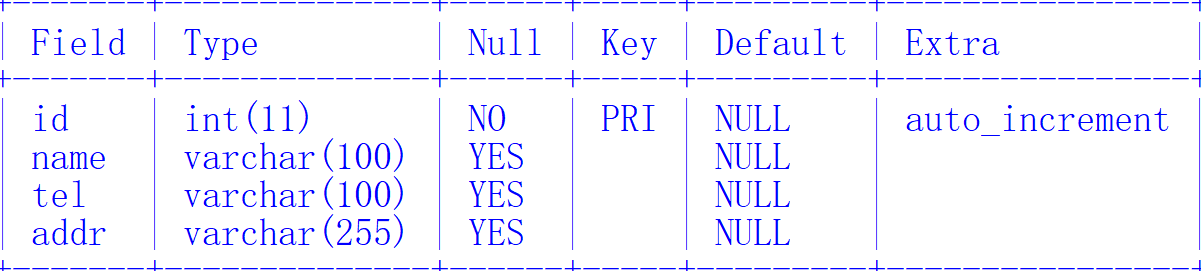
这里利用之前学过的MySQL、JSP、JSTL、html、css等技术，以及目前市面上最流行的企业级Java开发框架SSM实现一个《永和大王门店管理系统》，其中包括门店管理模块和订单管理模块，两个模块中，实现了对门店和订单信息的增删改查功能。



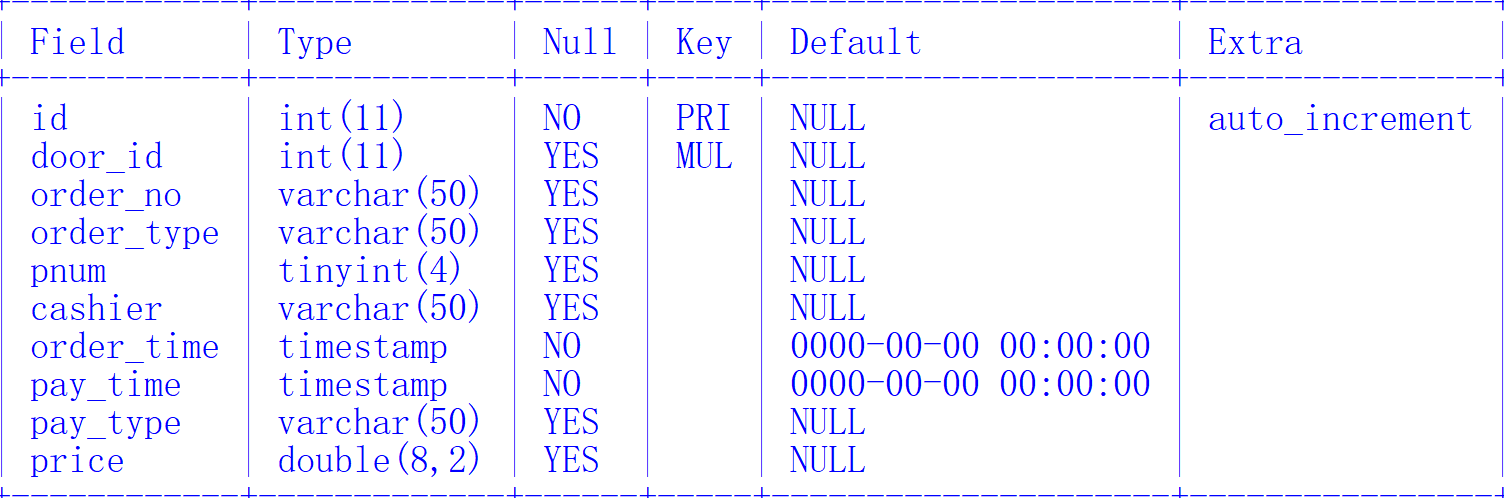


## 数据库及表设计

门店表结构：



订单表结构：



执行SQL脚本文件，创建库和所需表

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

-- 创建yonghedb库、tb\_door、tb\_order表并插入记录

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

-- 删除yonghedb库(如果存在)

-- drop database if exists yonghedb;

-- 重新创建yonghedb库

create database if not exists yonghedb charset utf8;

-- 选择yonghedb库

use yonghedb;

-- 删除门店表(需要先删除订单表)

drop table if exists tb\_order;

drop table if exists tb\_door;

-- 创建门店表

create table tb\_door(

id int primary key auto\_increment, -- 门店id

name varchar(100), -- 门店名称

tel varchar(100), -- 联系电话

addr varchar(255) -- 门店地址

);

-- 往门店表中插入记录

insert into tb\_door values ('1', '永和大王(北三环西路店)', '010-62112313', '北三环西路甲18号院-1号大钟寺中坤广场d座');

insert into tb\_door values ('2', '永和大王(知春路店)', '010-82356537', '知春路29号大运金都');

insert into tb\_door values ('3', '永和大王(东直门)', '010-84477746', '东直门外大街48号东方银座b2-08');

insert into tb\_door values ('4', '永和大王(北京站)', '010-65286602', '毛家湾胡同甲13号北京站候车大厅2层');

insert into tb\_door values ('5', '永和大王(学院路店)', '010-62152539', '学院南路37号超市发四道口店四道口西北角');

-- 删除订单表(如果存在)

drop table if exists tb\_order;

-- 创建订单表

create table tb\_order(

id int(11) primary key AUTO\_INCREMENT, -- 订单id

door\_id int(11), -- 门店id

order\_no varchar(50), -- 订单编号

order\_type varchar(50), -- 订单类型(堂食/打包/外卖..)

pnum int, -- 用餐人数

cashier varchar(50), -- 收银员

order\_time timestamp NOT NULL DEFAULT '0000-00-00 00:00:00', -- 下单时间

pay\_time timestamp NOT NULL DEFAULT '0000-00-00 00:00:00', -- 支付时间

pay\_type varchar(50), -- 支付类型(微信支付/支付宝支付)

price double, -- 支付金额

foreign key(door\_id) REFERENCES tb\_door(id) -- 关联外键

);

-- 往订单表中插入记录

INSERT INTO tb\_order VALUES ('1', '1', 'P001', '堂食', '1', '张三', '2018-04-26 14:49:07', '2018-04-26 14:50:38', '微支付', '16.00');

INSERT INTO tb\_order VALUES ('2', '1', 'P003', '外卖', '3', '张三', '2018-04-27 13:34:07', '2018-04-27 13:34:38', '微支付', '20.00');

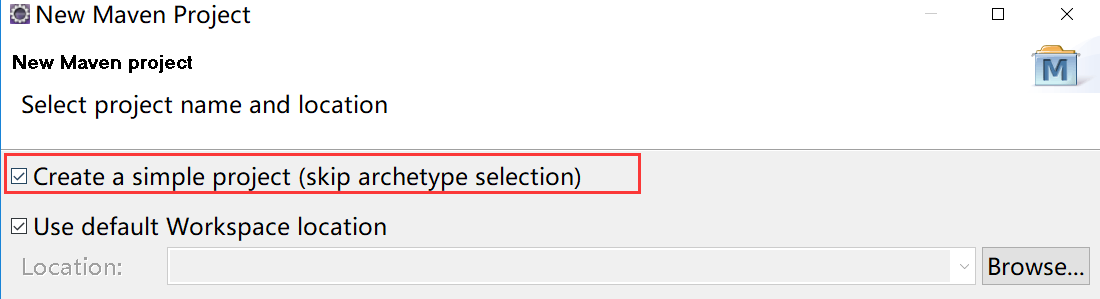
INSERT INTO tb\_order VALUES ('3', '1', 'P005', '打包', '1', '张三', '2019-01-22 11:59:22', '2019-01-22 11:59:22', '微支付', '28.00');

INSERT INTO tb\_order VALUES ('4', '1', 'P007', '堂食', '1', '李四', '2019-01-23 13:01:26', '2019-01-23 13:01:26', '微支付', '49.00');

# 项目环境搭建(SSM整合)

## 项目环境搭建

1、创建Maven的简单web工程

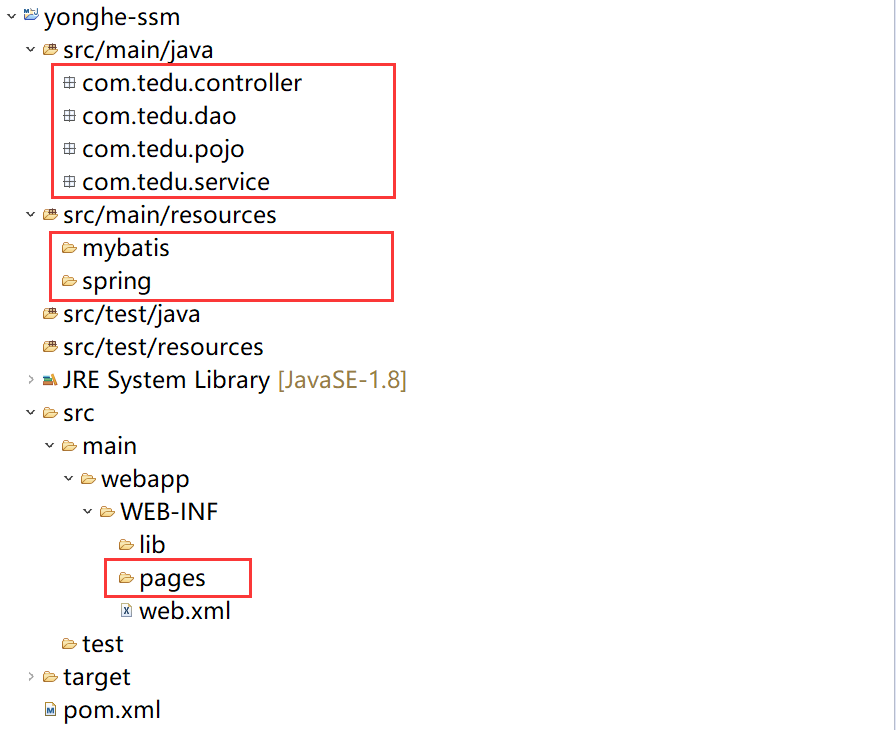




2、补全WEB-INF目录和web.xml文件



3、创建包路径和目录



4、在pom.xml文件中，引入junit、log4j、servlet等依赖包

<dependencies>

<!-- 单元测试 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.10</version>

<scope>test</scope>

</dependency>

<!-- 整合log4j -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.6.4</version>

</dependency>

<!-- Jackson Json处理工具包 -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.4.2</version>

</dependency>

<!-- Servlet/JSP/JSTL -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jsp-api</artifactId>

<version>2.0</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>jstl</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

</dependencies>

5、在resources目录下创建log4j.properties文件，配置内容如下:

# Global logging configuration

log4j.rootLogger=DEBUG, stdout

# Console output...

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

## 整合spring框架

1、在pom.xml文件中，引入spring的依赖包

<!-- 整合spring框架（包含springmvc）

这个jar文件包含springmvc开发时的核心类, 同时也会将依赖的相关jar文件引入进来(spring的核心jar文件也包含在内)

-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.1.3.RELEASE</version>

</dependency>

<!--这个jar文件包含对Spring对JDBC数据访问进行封装的所有类 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>4.1.3.RELEASE</version>

</dependency>

<dependency>

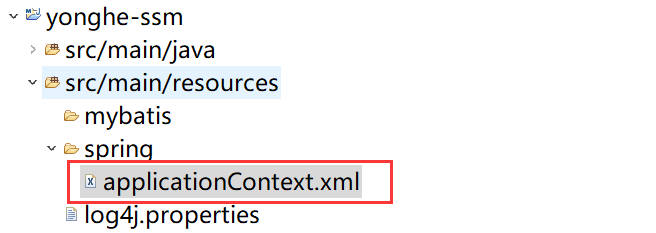
<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>4.1.3.RELEASE</version>

</dependency>

2、在resources/spring目录下，创建spring的核心配置文件：applicationContext.xml



配置内容如下：

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

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

xmlns:aop="http://www.springframework.org/schema/aop" xmlns:tx="http://www.springframework.org/schema/tx"

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-4.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd

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

</beans>

## 整合springmvc框架

前面在添加spring的jar包同时, 将springmvc的jar包也引入了，因此这里不再引入springmvc的jar包

1、在resources/spring目录下，创建springmvc的核心配置文件：springmvc-config.xml，内容配置如下：

<?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:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/mvc*

*http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*

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

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

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

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

<!-- 1.配置前端控制器放行静态资源(html/css/js等，否则静态资源将无法访问) -->

<mvc:default-servlet-handler/>

<!-- 2.配置注解驱动，用于识别注解（比如@Controller） -->

<mvc:annotation-driven></mvc:annotation-driven>

<!-- 3.配置需要扫描的包：spring自动去扫描 base-package 下的类，

如果扫描到的类上有 @Controller、@Service、@Component等注解，

将会自动将类注册为bean

-->

<context:component-scan base-package=*"com.tedu.controller"*>

</context:component-scan>

<!-- 4.配置内部资源视图解析器

prefix:配置路径前缀

suffix:配置文件后缀

-->

<bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<property name=*"prefix"* value=*"****/WEB-INF/pages/****"*/>

<property name=*"suffix"* value=*"****.jsp****"*/>

</bean>

</beans>

2、在web.xml中配置springmvc

<!-- 配置springmvc, 将所有请求交给springmvc来处理 -->

<servlet>

<servlet-name>springmvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!-- 配置springmvc核心配置文件的位置，默认Springmvc的配置文件是在WEB-INF目录下，默认的名字为springmvc-servlet.xml，如果要放在其他目录，则需要指定如下配置：

-->

<init-param>

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

<param-value>classpath:spring/\*.xml</param-value>

</init-param>

</servlet>

<!-- 其中的斜杠（/）表示拦截所有请求（除JSP以外）, 所有请求都要经过springmvc前端控制器 -->

<servlet-mapping>

<servlet-name>springmvc</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

3、配置springmvc乱码处理过滤器

<!-- 乱码处理过滤器 -->

<filter>

<filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<!-- 指定编码集 -->

<init-param>

<param-name>encoding</param-name>

<param-value>utf-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encodingFilter</filter-name>

<!-- 指定拦截方式为拦截所有请求 -->

<url-pattern>/\*</url-pattern>

</filter-mapping>

3、测试springmvc框架:

(1)创建test.jsp页面

在WEB-INF/pages/目录下，创建test.jsp页面。

<%@ page pageEncoding=*"utf-8"*%>

<!DOCTYPE html>

<html>

<head>

<meta charset=*"utf-8"*/>

</head>

<body>

<h1>test springmvc~~~</h1>

</body>

</html>

(2)创建测试Controller: TestController

/\*\* 测试类：测试springmvc开发环境 \*/

@Controller

**public** **class** TestController {

@RequestMapping("/hello")

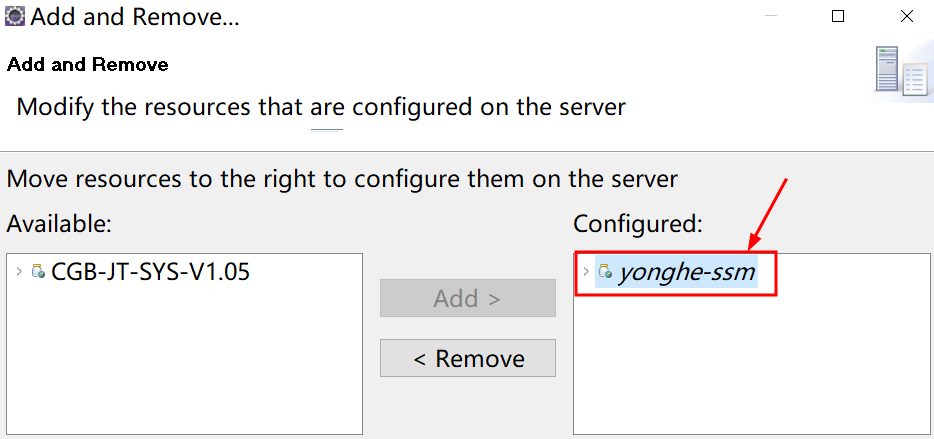
**public** String hello(){

**return** "test";

}

}

(3)将项目部署到服务器中，启动服务器：



访问效果如下：



## 整合mybatis框架

1、在pom.xml文件中，引入mybatis及相关依赖包

<!-- 整合mybatis框架 -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.2.8</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-spring</artifactId>

<version>1.2.2</version>

</dependency>

<!-- mysql驱动 -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.32</version>

</dependency>

<!-- druid连接池 -->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.1.6</version>

</dependency>

2、在resources/mybatis目录下，创建mybatis的核心配置文件：mybatis-config.xml

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

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<!-- MyBatis的全局配置文件 -->

<configuration >

<!-- 1.配置开发环境 -->

<environments default=*"develop"*>

<!-- 这里可以配置多个环境，比如develop，test等 -->

<environment id=*"develop"*>

<!-- 1.1.配置事务管理方式：JDBC：将事务交给JDBC管理（推荐） -->

<transactionManager type=*"JDBC"*></transactionManager>

<!-- 1.2.配置数据源，即连接池方式:JNDI/POOLED/UNPOOLED -->

<dataSource type=*"POOLED"*>

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

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/yonghedb?characterEncoding=utf-8"*/>

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

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

</dataSource>

</environment>

</environments>

<!-- 2.加载Mapper配置文件,路径以斜杠间隔: xx/xx/../xx.xml -->

<mappers>

<mapper resource=*"mybatis/mapper/DoorMapper.xml"*/>

</mappers>

</configuration>

3、创建实体类Door，用于封装所有的门店信息

**package** com.tedu.pojo;

**public** **class** Door {

**private** Integer id; //门店编号

**private** String name; //门店名称

**private** String tel; //门店电话

**private** String addr; //门店地址

//getter和setter方法

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** String getTel() {

**return** tel;

}

**public** **void** setTel(String tel) {

**this**.tel = tel;

}

**public** String getAddr() {

**return** addr;

}

**public** **void** setAddr(String addr) {

**this**.addr = addr;

}

//重写toString方法

@Override

**public** String toString() {

**return** "Door [id=" + id + ", name=" + name + ", tel=" + tel + ", addr=" + addr + "]";

}

}

4、在src/main/resources/mybatis/mapper目录下，创建Door实体类的映射文件—DoorMapper.xml



配置内容如下：

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

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- 门店表的映射文件 namespace值为对应接口的全路径 -->

<mapper namespace="com.tedu.dao.DoorMapper">

<!-- 1.查询所有门店信息，id值为对应接口中方法的名字

resultType指定将查询的结果封装到哪个pojo对象中

-->

<select id="findAll" resultType="com.tedu.pojo.Door">

select \* from tb\_door

</select>

</mapper>

5、在mybatis的全局配置文件中引入DoorMapper.xml映射文件

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

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<!-- MyBatis的全局配置文件 -->

<configuration >

<!-- 1.配置开发环境 -->

...

<!-- 2.加载Mapper配置文件,路径以斜杠间隔: xx/xx/../xx.xml -->

<!-- 配置映射文件 -->

<mappers>

<mapper resource=*"mybatis/mapper/DoorMapper.xml"*/>

</mappers>

</configuration>

4、创建com.tedu.dao.DoorMapper接口，并根据EmpMapper.xml文件中的sql语句，提供findAll方法

**package** com.tedu.dao;

**import** java.util.List;

**import** com.tedu.pojo.Door;

/\*\*

\* DoorMapper接口

\* 声明增删改查方法,对门店信息进行操作

\*/

**public** **interface** DoorMapper {

/\*\*

\* 1.查询所有门店信息

\*/

**public** List<Door> findAll();

}

5、创建测试com.tedu.controller.TestMybatis类，对mybatis开发环境进行测试

**package** com.tedu.controller;

**import** java.io.InputStream;

**import** java.util.List;

**import** org.apache.ibatis.io.Resources;

**import** org.apache.ibatis.session.SqlSession;

**import** org.apache.ibatis.session.SqlSessionFactory;

**import** org.apache.ibatis.session.SqlSessionFactoryBuilder;

**import** com.tedu.dao.DoorMapper;

**import** com.tedu.pojo.Door;

/\*\* 测试类：测试mybatis开发环境 \*/

**public** **class** TestMybatis {

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

//1.读取mybatis-config.xml核心文件

InputStream in = Resources.*getResourceAsStream*(

"mybatis/mybatis-config.xml");

//2.获取SqlSessionFactory工厂

SqlSessionFactory factory =

**new** SqlSessionFactoryBuilder()

.build(in);

//3.获取SqlSession对象

SqlSession session = factory.openSession();

//4.获取DoorMapper接口的实例

DoorMapper mapper = session.getMapper(DoorMapper.**class**);

//5.调用findAll方法查询所有门店信息

List<Door> list = mapper.findAll();

//6.遍历所有门店信息

**for**(Door door : list){

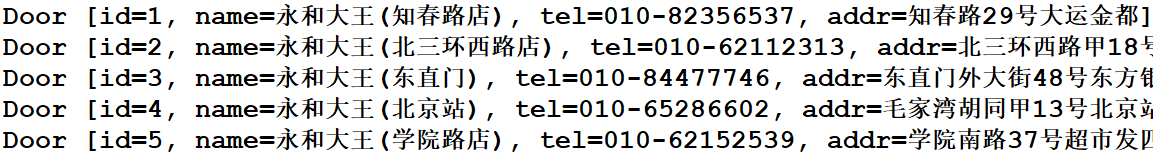
System.***out***.println(door);

}

}

}

执行结果：



## 整合spring和mybatis

1、修改mybatis-config.xml文件，将连接池等配置移除，在spring中配置

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

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<!-- MyBatis的全局配置文件 -->

<configuration >

<!-- 1.配置开发环境 -->

<!-- 1.1.配置事务管理方式：JDBC：将事务交给JDBC管理（推荐） -->

<!-- 1.2.配置数据源，即连接池方式:JNDI/POOLED/UNPOOLED -->

<!-- 2.加载Mapper配置文件,路径以斜杠间隔: xx/xx/../xx.xml -->

</configuration>

2、在applicationContext.xml中配置druid连接池

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

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

xmlns:aop="http://www.springframework.org/schema/aop" xmlns:tx="http://www.springframework.org/schema/tx"

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-4.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd

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

<!-- 1.加载jdbc.properties文件的位置 -->

<context:property-placeholder location="classpath:jdbc.properties"/>

<!-- 2.配置druid连接池 ，id是固定值，class是druid连接池类的全路径 -->

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">

<!-- 配置连接数据库的基本信息 -->

<property name="driverClassName" value="${db.driverClassName}"></property>

<property name="url" value="${db.url}"></property>

<property name="username" value="${db.username}"></property>

<property name="password" value="${db.password}"></property>

</bean>

<!-- 3.整合spring和mybatis框架

将SqlSession等对象的创建交给Spring容器

id值(sqlSessionFactory)是固定值

-->

<bean id="sqlSessionFactory"

class="org.mybatis.spring.SqlSessionFactoryBean">

<!-- 3.1.指定mybatis核心配置文件的位置 -->

<property name="configLocation"

value="classpath:mybatis/mybatis-config.xml"></property>

<!-- 3.2.配置连接池(数据源) ref指向连接池bean对象的id值 -->

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

<!-- 3.3、扫描所有的 XxxMapper.xml映射文件，读取其中配置的SQL语句 -->

<property name="mapperLocations" value="classpath:mybatis/mapper/\*.xml"/>

</bean>

<!-- 4、定义mapper接口扫描器 -->

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<!-- 扫描所有XxxMapper接口，将接口实例的创建交给spring容器 -->

<property name="basePackage"

value="com.tedu.dao"/>

</bean>

<!-- 5.配置需要扫描的包(service层)：spring自动去扫描 base-package下的类，

如果扫描到的类上有 @Controller、@Service、@Component等注解，

将会自动将类注册为bean（即由spring创建实例）

-->

<context:component-scan

base-package="com.tedu.service">

</context:component-scan>

</beans>

4、在resources目录下创建jdbc.properties文件，将连接数据库的基本信息提取到文件中

db.driverClassName=com.mysql.jdbc.Driver

db.url=jdbc:mysql:///yonghedb?characterEncoding=utf-8

db.username=root

db.password=root

5、复制TestController类，改名为TestSSM

**package** com.tedu.controller;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** com.tedu.dao.DoorMapper;

**import** com.tedu.pojo.Door;

/\*\* 测试类：测试SSM开发环境 \*/

@Controller /\* 这个注解表示当前类属于Controller层代码 \*/

**public** **class** TestSSM {

/\*\* 自动装配：由spring自动为属性赋值(对象) \*/

@Autowired

DoorMapper mapper;

@RequestMapping("/testssm")

**public** String testSSM(){

//1.调用findAll方法查询所有门店信息

List<Door> list = mapper.findAll();

//2.遍历所有门店信息

**for**(Door door : list){

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

}

**return** "test";

}

}

# 查询所有门店

查询所有门店开发步骤：

1、点击【门店管理】，访问DoorController中的查询所有门店信息的方法【doorList】

2、在【doorList】方法中调用service层，service层再调用dao层查询所有门店信息，返回所有门店信息的List<Door>集合

3、在【doorList】方法中，将返回的所有门店信息集合（List<Door>）存入Model中，并转向door\_list.jsp（即 return "door\_list"），将所有门店信息带到jsp页面进行显示

## 准备工作

1、创建Door实体类，用于封装所有的门店信息（若上面已创建，直接跳过即可！）

**package** com.tedu.pojo;

/\*\* 门店实体类 \*/

**public** **class** Door {

**private** Integer id; //门店编号

**private** String name; //门店名称

**private** String tel; //门店电话

**private** String addr; //门店地址

//getter和setter方法

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** String getTel() {

**return** tel;

}

**public** **void** setTel(String tel) {

**this**.tel = tel;

}

**public** String getAddr() {

**return** addr;

}

**public** **void** setAddr(String addr) {

**this**.addr = addr;

}

//重写toString方法

@Override

**public** String toString() {

**return** "Door [id=" + id + ", name=" + name + ", tel=" + tel + ", addr=" + addr + "]";

}

}

2、创建Door实体类的映射文件—DoorMapper.xml（若上面已创建，直接跳过即可！）



配置内容如下：

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

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- 门店表的映射文件 namespace值为对应接口的全路径 -->

<mapper namespace="com.tedu.dao.DoorMapper">

</mapper>

## dao层代码实现

**1、更新resources/mybatis/mapper/DoorMapper.xml文件，添加“查询所有门店信息”SQL（若上面已创建，直接跳过即可！）**

<!-- 1.查询所有门店信息

id值为对应接口中方法的名字

resultType指定将查询的结果封装到哪个pojo对象中

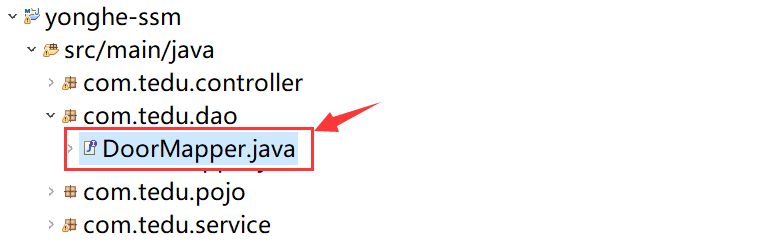
-->

<select id="findAll" resultType="com.tedu.pojo.Door">

select \* from tb\_door

</select>

2、创建DoorMapper接口, 添加查询所有门店方法（若上面已创建，直接跳过即可！）



DoorMapper代码实现如下：

**package** com.tedu.dao;

**import** java.util.List;

**import** com.tedu.pojo.Door;

**public** **interface** DoorMapper {

/\*\*

\* 查询所有门店信息

\* **@return** List<Door>

\*/

**public** List<Door> findAll();

}

## service层代码实现

1、创建DoorService接口，并添加查询所有门店信息的方法

**package** com.tedu.service;

**import** java.util.List;

**import** org.springframework.stereotype.Service;

**import** com.tedu.pojo.Door;

**public** **interface** DoorService {

/\*\*

\* 1.查询所有门店信息

\* **@return** List<Door>

\*/

**public** List<Door> findAll();

}

2、创建DoorService接口的实现类—DoorServiceImpl，实现父接口中的findAll方法：调用dao层的findAll方法查询所有门店信息

**package** com.tedu.service;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** com.tedu.dao.DoorMapper;

**import** com.tedu.pojo.Door;

/\* @Service作用1： 将当前对象的创建交给spring管理

\* 作用2：作为业务层代码的标识

\*/

**@Service**

**public** **class** DoorServiceImpl **implements** DoorService {

@Autowired //自动注入(由spring创建mapper对象并为属性赋值)

**private** DoorMapper mapper;

@Override

**public** List<Door> findAll() {

//1.调用DoorMapper的findAll方法，查询所有门店信息

List<Door> list = doorMapper.findAll();

//2.将所有门店信息的List集合返回

**return** list;

}

}

## controller层代码实现

1、创建DoorController类，提供doorList方法：调用service层的findAll方法，查询所有门店信息。

**package** com.tedu.controller;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** com.tedu.pojo.Door;

**import** com.tedu.service.DoorService;

@Controller

/\* @Controller作用1： 将当前对象的创建交给spring管理

\* 作用2：作为控制层代码的标识

\*/

**public** **class** DoorController {

@Autowired //自动注入(由spring创建当前对象并为属性赋值)

**private** DoorService doorService;

@RequestMapping("/doorList")

**public** String doorList(){

//调用service层的findAll方法，查询所有的门店信息

List<Door> list = doorService.findAll();

**for** (Door d:list) {

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

}

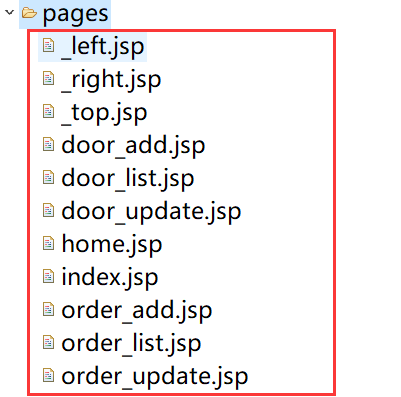
**return** "door\_list";

}

}

## 在页面上显示所有门店列表

1、导入页面（JSP）



2、创建通用的页面跳转方法

@PathVariable 映射 URL 绑定的占位符

URL 中的 {xxx} 占位符可以通过@PathVariable(“xxx“) 绑定到操作方法的入参中。

举例：现在要访问 /WEB-INF/pages/index.jsp页面，按照之前的讲解，我们会在Controller中添加一个方法，例如：

@RequestMapping("/index")

**public** String xxx(){

**return** "index";

}

如果有很多这样的jsp页面需要我们访问呢？每访问一个jsp都要在Controller中添加一个方法吗？毫无疑问，这样做太繁琐了。

通过上面的观察，我们可以发现，@RequestMapping("/index") 中的"/**index**" 和 **return** "index"中的"**index**" 名称是相同的，按照此规律，我们可以提供一个通用的页面跳转方法，访问路径中的名字是什么，就返回/WEB-INF/pages/下的什么JSP页面。

因此，创建PageController类，代码实现如下：

@Controller

**public** **class** PageController {

/\*\* 通用的页面跳转方法

\* 例如:访问路径为 "/index"，则方法最后返回的值为："index"

\* 即最终返回 /WEB-INF/pages/index.jsp

\*/

@RequestMapping("/{page}")

**public** String page(@PathVariable String page){

**return** page;

}

}

3、更新DoorController代码，将门店list集合存入model带到JSP显示

@RequestMapping("/doorList")

**public** String doorList(Model model){

//调用service层的findAll方法，查询所有的门店信息

List<Door> list = doorService.findAll();

//将所有门店信息的list集合存入model中，带到JSP显示

model.addAttribute("list", list);

**return** "door\_list";

}

4、实现door\_list.jsp

<%@ page pageEncoding=*"utf-8"*%>

<%-- 引入JSTL标签库 --%>

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>

...

<body>

...

...

<!-- 模版数据 -->

<c:forEach items=*"*${ list }*"* var=*"door"* varStatus=*"status"*>

<tr>

<td>${ status.count }</td>

<td>${ door.name }</td>

<td>${ door.tel }</td>

<td>${ door.addr }</td>

<td>

<a href=*"doorDelete?id=*${ door.id }*"*>删除</a>

&nbsp;|&nbsp;

<a href=*"doorUpdate?id=*${ door.id }*"*>修改</a>

</td>

</tr>

</c:forEach>

</table>

</body>

# 新增门店信息

新增门店信息开发步骤：

1、在门店信息列表页面中，点击【新增门店】，跳转到门店新增页面【door\_add.jsp】

2、用户在【door\_add.jsp】中输入新增的门店信息，点击【提交】将门店信息提交给DoorController的【doorAdd】方法

3、在【doorAdd】方法中，通过参数接收浏览器提交过来的门店信息，调用service层，service层再调用dao层将门店信息保存到数据库中，

4、新增成功后，重定向到门店列表页面，显示所有门店（即重定向到DoorController的【doorList】方法，先查询所有门店，再将所有门店信息带到JSP显示）

## dao层代码实现

1、更新resources/mybatis/mapper/DoorMapper.xml文件，添加“新增门店信息”SQL。

<!-- 2.新增门店信息 -->

<insert id=*"addDoor"*>

insert into tb\_door

values(#{id}, #{name}, #{tel}, #{addr})

</insert>

2、更新com.tedu.dao.DoorMapper接口，添加 addDoor方法，新增门店信息。

/\*\*

\* 2.新增门店信息

\* **@param** door

\*/

**public** **void** addDoor(Door door);

## service层代码实现

1、更新com.tedu.service.DoorService接口，添加 addDoor方法，新增门店信息

/\*\*

\* 2.新增门店信息

\* **@param** door

\*/

**public** **void** addDoor(Door door);

2、更新com.tedu.service.DoorServiceImpl类，实现父接口中的 addDoor方法：调用dao层的addDoor方法新增门店信息

@Override

**public** **void** addDoor(Door door) {

//调用DoorMapper的addDoor方法，新增门店信息

doorMapper.addDoor(door);

}

## Controller层代码实现

1、更新com.tedu.controller.DoorController类，添加并实现doorAdd方法：调用service层的addDoor方法新增门店信息

/\*\* 2.新增门店信息 \*/

@RequestMapping("/doorAdd")

**public** String doorAdd(Door door){

//1.调用service层的addDoor方法，新增门店信息

doorService.addDoor(door);

//2.重定向到门店列表页面, 显示所有门店信息

**return** "redirect:/doorList";

}

# 删除门店信息

删除门店信息开发步骤：

1、在门店信息列表页面中，点击门店信息后的【删除】按钮，访问DoorController中的【doorDelete】方法，访问的同时将所要删除的门店id一同提交给服务器。

2、在【doorDelete】方法中，通过参数接收门店id，调用service层，service层再调用dao层通过id删除指定的门店信息。

3、删除成功后，重定向到门店列表页面，显示所有门店（即重定向到DoorController的【doorList】方法，先查询所有门店，再将所有门店信息带到JSP显示）。

## dao层代码实现

1、更新resources/mybatis/mapper/DoorMapper.xml文件，添加“删除门店信息”SQL。

<!-- 3.根据id删除指定门店信息 -->

<delete id=*"deleteById"*>

delete from tb\_door where id=#{id}

</delete>

2、更新com.tedu.dao.DoorMapper接口，添加 deleteById方法，根据id删除门店信息。

<!-- 3.根据id删除指定门店信息 -->

<delete id=*"deleteById"*>

delete from tb\_door where id=#{id}

</delete>

## service层代码实现

1、更新com.tedu.service.DoorService接口，添加 deleteById方法，根据id删除门店信息

/\*\*

\* 3.根据id删除指定的门店信息

\* **@param** id

\*/

**public** **void** deleteById(Integer id);

2、更新com.tedu.service.DoorServiceImpl类，实现父接口中的 deleteById方法：调用dao层的deleteById方法，根据id删除门店信息

@Override

**public** **void** deleteById(Integer id) {

//调用DoorMapper的deleteById方法，根据id删除指定的门店信息

doorMapper.deleteById(id);

}

## Controller层代码实现

1、更新com.tedu.controller.DoorController类，添加并实现doorDelete方法：调用service层的deleteById方法删除门店信息

/\*\* 3.删除门店信息 \*/

@RequestMapping("/doorDelete")

**public** String doorDelete(Integer id){

//1.调用service层的deleteById方法，删除指定id的门店信息

doorService.deleteById(id);

//2.重定向到门店列表页面, 显示所有门店信息

**return** "redirect:/doorList";

}

# 更新门店信息

更新门店信息开发步骤：

1、在门店信息列表页面中，点击门店信息后的【修改】按钮，访问DoorController中的【doorInfo】方法，访问的同时将所要查询的门店id一同提交给服务器。

2、在【doorInfo】方法中，通过参数接收门店id，调用service层，service层再调用dao层通过id查询指定的门店信息（Door）并返回

3、在【doorInfo】方法中，将返回的门店信息（Door）存入Model中，并转向door\_update.jsp（即 return "door\_update"），将门店信息带到jsp页面进行回显

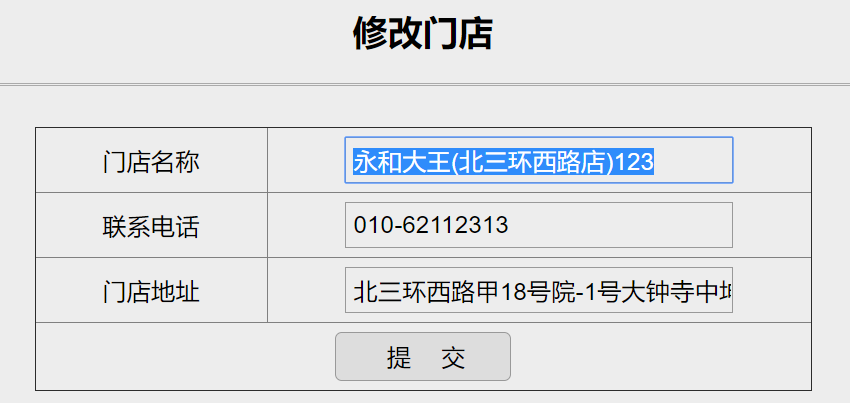
4、在原有门店信息的基础上，对门店信息进行修改，点击【提交】将修改后的门店信息提交给DoorController的【doorUpdate】方法

5、在【doorUpdate】方法中，通过参数接收浏览器提交过来的门店信息，调用service层，service层再调用dao层根据id修改门店信息。

6、修改成功后，重定向到门店列表页面，显示所有门店（即重定向到DoorController的【doorList】方法，先查询所有门店，再将所有门店信息带到JSP显示）

## dao层代码实现（查询门店）

修改门店信息需要在原有门店信息的基础上进行修改，因此在修改之前，需要先查询出原有门店信息，在原有门店信息的基础上修改后，再将修改后的门店信息提交给服务器端，服务器端将修改后的门店信息更新到数据库中



**1、更新resources/mybatis/mapper/DoorMapper.xml文件，添加“根据id查询门店信息”SQL。**

<!-- 4.根据id查询指定门店信息 -->

<select id=*"findById"* resultType=*"com.tedu.pojo.Door"*>

select \* from tb\_door where id=#{id}

</select>

2、更新com.tedu.dao.DoorMapper接口，添加 findById方法，根据id查询门店信息。

/\*\*

\* 4.根据id查询门店信息

\* **@param** id

\*/

**public** Door findById(Integer id);

## service层代码实现（查询门店）

1、更新com.tedu.service.DoorService接口，添加 findById方法，根据id查询门店信息

/\*\*

\* 4.根据id查询指定的门店信息

\* **@param** id

\* **@return**

\*/

**public** Door findById(Integer id);

2、更新com.tedu.service.DoorServiceImpl类，实现父接口中的 findById方法：调用dao层的findById方法，根据id查询门店信息

@Override

**public** Door findById(Integer id) {

//调用DoorMapper的findById方法，根据id查询门店信息

Door door = doorMapper.findById(id);

**return** door;

}

## Controller层代码实现（查询门店）

1、更新com.tedu.controller.DoorController类，添加并实现doorInfo方法：调用service层的findById方法根据id查询门店信息

/\*\* 4.根据id查询门店信息 \*/

@RequestMapping("/doorInfo")

**public** String doorInfo(Integer id, Model model){

//1.调用service层的findById方法，根据id查询门店信息

Door door = doorService.findById(id);

//2.将门店信息保存到Model中

model.addAttribute("door", door);

//3.将门店信息带到door\_update.jsp进行回显

**return** "door\_update";

}

## 数据回显

1、在door\_update.jsp中显示将要修改的门店信息:

<h2>修改门店</h2>

<hr/>

<form action=*"doorUpdate"* method=*"POST"*>

<!-- hidden隐藏域,在提交表单时连door.id一起提交 -->

<input type=*"hidden"* name=*"id"* value=*"*${door.id}*"*/>

<table border=*"1"*>

<tr>

<td width=*"30%"*>门店名称</td>

<td>

<input type=*"text"* name=*"name"* value=*"*${door.name}*"*/>

</td>

</tr>

<tr>

<td>联系电话</td>

<td>

<input type=*"text"* name=*"tel"* value=*"*${door.tel}*"*/>

</td>

</tr>

<tr>

<td>门店地址</td>

<td>

<input type=*"text"* name=*"addr"* value=*"*${door.addr}*"*/>

</td>

</tr>

<tr>

<td colspan=*"2"*>

<input type=*"submit"* value=*"提 交"* />

</td>

</tr>

</table>

</form>

将要修改的门店信息回显至修改页面后，下面实现修改门店信息

## dao层代码实现（更新门店）

1、更新resources/mybatis/mapper/DoorMapper.xml文件，添加“根据id更新门店信息”SQL：

<!-- 5.根据id更新指定门店信息 -->

<update id=*"updateById"*>

update tb\_door set name=#{name},tel=#{tel},addr=#{addr}

where id=#{id}

</update>

2、更新com.tedu.dao.DoorMapper接口，添加 updateById方法，根据id更新门店信息

/\*\*

\* 5.根据id更新门店信息

\* **@param** door

\*/

**public** **void** updateById(Door door);

## service层代码实现（更新门店）

1、更新com.tedu.service.DoorService接口，添加 updateById方法，根据id更新门店信息

/\*\*

\* 5.根据id更新指定门店信息

\* **@param** door

\*/

**public** **void** updateById(Door door);

2、更新com.tedu.service.DoorServiceImpl类，实现父接口中的 updateById方法：调用dao层的updateById方法，根据id更新门店信息

@Override

**public** **void** updateById(Door door) {

//调用DoorMapper的updateById方法，根据id更新门店信息

doorMapper.updateById(door);

}

## Controller层代码实现（更新门店）

1、更新com.tedu.controller.DoorController类，添加并实现doorUpdate方法：调用service层的updaeById方法删除门店信息

/\*\* 5.根据id更新门店信息 \*/

@RequestMapping("/doorUpdate")

**public** String doorUpdate(Door door){

//1.调用service层的updateById方法，根据id跟新门店信息

doorService.updateById(door);

//2.重定向到门店列表页面, 显示所有门店信息

**return** "redirect:/doorList";

}

# 查询所有订单

查询所有订单开发步骤：

1、点击【订单管理】，访问OrderController中的查询所有订单信息的方法【doorList】

2、在【orderList】方法中调用service层，service层再调用dao层查询所有订单信息，返回所有订单信息的List<Order>集合

3、在【orderList】方法中，将返回的所有订单信息集合（List<Order>）存入Model中，并转向order\_list.jsp（即 return "order\_list"），将所有订单信息带到jsp页面进行显示

## 准备工作

1、创建com.tedu.pojo.Order实体类，用于封装所有的订单信息。

**package** com.tedu.pojo;

**import** java.util.Date;

/\*\* 订单实体类 \*/

**public** **class** Order {

**private** Integer id; //订单id

**private** Integer doorId; //门店编号

**private** String orderNo; //订单编号

**private** String orderType; //订单类型

**private** Integer pnum; //用餐人数

**private** String cashier; //收银员

**private** Date orderTime; //下单时间

**private** Date payTime; //结账时间

**private** String payType; //支付方式

**private** **double** price; //支付金额

//getter和setter方法

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** Integer getDoorId() {

**return** doorId;

}

**public** **void** setDoorId(Integer doorId) {

**this**.doorId = doorId;

}

**public** String getOrderNo() {

**return** orderNo;

}

**public** **void** setOrderNo(String orderNo) {

**this**.orderNo = orderNo;

}

**public** String getOrderType() {

**return** orderType;

}

**public** **void** setOrderType(String orderType) {

**this**.orderType = orderType;

}

**public** Integer getPnum() {

**return** pnum;

}

**public** **void** setPnum(Integer pnum) {

**this**.pnum = pnum;

}

**public** String getCashier() {

**return** cashier;

}

**public** **void** setCashier(String cashier) {

**this**.cashier = cashier;

}

**public** Date getOrderTime() {

**return** orderTime;

}

**public** **void** setOrderTime(Date orderTime) {

**this**.orderTime = orderTime;

}

**public** Date getPayTime() {

**return** payTime;

}

**public** **void** setPayTime(Date payTime) {

**this**.payTime = payTime;

}

**public** String getPayType() {

**return** payType;

}

**public** **void** setPayType(String payType) {

**this**.payType = payType;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

}

//重写toString方法

@Override

**public** String toString() {

**return** "Order [id=" + id + ", doorId=" + doorId + ", orderNo=" + orderNo + ", orderType=" + orderType

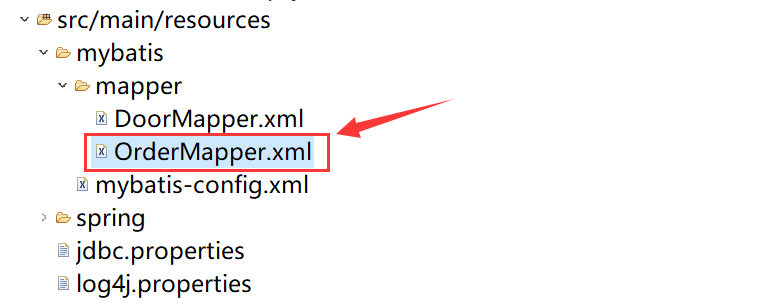
+ ", pnum=" + pnum + ", cashier=" + cashier + ", orderTime=" + orderTime + ", payTime=" + payTime

+ ", payType=" + payType + ", price=" + price + "]";

}

}

2、在resources/mybatis/mapper目录下创建Order实体类的映射文件—OrderMapper.xml。



配置内容如下：

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

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- 订单表映射文件 namespace值为对应接口的全路径 -->

<mapper namespace=*"com.tedu.dao.OrderMapper"*>

</mapper>

## dao层代码实现

**1、更新resources/mybatis/mapper/OrderMapper.xml文件，添加“查询所有订单信息”SQL。**

<!--

数据表列字段与pojo类属性映射关系

type:指定将查询的结果封装到哪个类pojo对象中

id:指定一个唯一表示resultMap的值

-->

<resultMap type=*"com.tedu.pojo.Order"* id=*"orderRM"*>

<id column=*"id"* property=*"id"*/>

<result column=*"door\_id"* property=*"doorId"*/>

<result column=*"order\_no"* property=*"orderNo"*/>

<result column=*"order\_type"* property=*"orderType"*/>

<result column=*"pnum"* property=*"pnum"*/>

<result column=*"cashier"* property=*"cashier"*/>

<result column=*"order\_time"* property=*"orderTime"*/>

<result column=*"pay\_time"* property=*"payTime"*/>

<result column=*"pay\_type"* property=*"payType"*/>

<result column=*"price"* property=*"price"*/>

</resultMap>

<!-- 1.查询所有订单信息 id值为对应接口中方法的名字

resultMap:指定为resultMap标签的id值

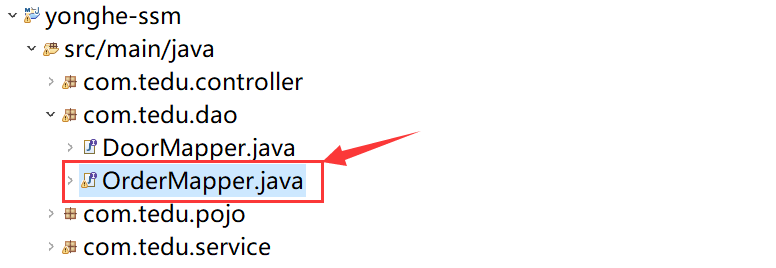
-->

<select id=*"findAll"* resultMap=*"orderRM"*>

select \* from tb\_order

</select>

2、创建OrderMapper接口, 添加查询所有订单方法。



OrderMapper代码实现如下：

**package** com.tedu.dao;

**import** java.util.List;

**import** com.tedu.pojo.Order;

**public** **interface** OrderMapper {

/\*\*

\* 1.查询所有订单信息

\* **@return** List<Order>

\*/

**public** List<Order> findAll();

}

## service层代码实现

1、创建OrderService接口，添加查询所有订单信息的findAll方法

**package** com.tedu.service;

**import** java.util.List;

**import** com.tedu.pojo.Order;

/\*\* OrderService接口 \*/

**public** **interface** OrderService {

/\*\*

\* 1.查询所有订单信息

\* **@return** List<Order>

\*/

**public** List<Order> findAll();

}

2、创建OrderService接口的实现类—OrderServiceImpl，实现父接口中的findAll方法：调用dao层的findAll方法查询所有订单信息

**package** com.tedu.service;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** com.tedu.dao.OrderMapper;

**import** com.tedu.pojo.Order;

@Service

**public** **class** OrderServiceImpl **implements** OrderService {

@Autowired //自动注入(由spring创建mapper对象并为属性赋值)

**private** OrderMapper orderMapper;

@Override

**public** List<Order> findAll() {

//1.调用OrderMapper的findAll方法，查询所有订单信息

List<Order> list = orderMapper.findAll();

//2.将所有订单信息的List集合返回

**return** list;

}

}

## controller层代码实现

1、创建OrderController类，提供orderList方法：调用service层的findAll方法，查询所有订单信息。

**package** com.tedu.controller;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** com.tedu.pojo.Door;

**import** com.tedu.pojo.Order;

**import** com.tedu.service.DoorService;

**import** com.tedu.service.OrderService;

/\* @Controller作用1： 将当前对象的创建交给spring管理

\* 作用2：作为控制层代码的标识

\*/

@Controller

**public** **class** OrderController {

@Autowired //自动注入(由spring创建对象并为属性赋值)

**private** OrderService orderService;

@Autowired //自动注入(由spring创建当前对象)

**private** DoorService doorService;

...

/\*\* 1.查询所有订单信息 \*/

@RequestMapping("/orderList")

**public** String orderList(Model model){

//1.调用OrderService层的findAll方法查询所有订单

List<Order> list = orderService.findAll();

//2.将所有订单的list集合存入Model中(存入request域中)

model.addAttribute("list", list);

//3.查询所有门店

List<Door> doorlist = doorService.findAll();

//4.将所有订单的list集合存入Model中(存入request域中)

model.addAttribute("doorList", doorlist);

//5.转向订单列表页面

**return** "order\_list";

}

}

## 在页面上显示所有订单列表

1、实现order\_list.jsp

<%@ page pageEncoding=*"utf-8"*%>

<%-- 引入JSTL标签库 --%>

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>

...

<body>

...

<!-- 模版数据 -->

<c:forEach items=*"*${ list }*"* var=*"order"*

varStatus=*"status"*>

<tr>

<td>${ status.count }</td>

<!-- 显示订单所属性门店名称 -->

<c:forEach items=*"*${doorList}*"* var=*"door"*>

<c:if test=*"*${ door.id==order.doorId }*"*>

<td class=*"1111"*>${ door.name }</td>

</c:if>

</c:forEach>

<td>${ order.orderNo }</td>

<td>${ order.orderType }</td>

<td>${ order.pnum }</td>

<td>${ order.cashier }</td>

<td>

<fmt:formatDate value=*"*${ order.orderTime }*"*

pattern=*"yyyy-MM-dd HH:mm:ss"*/>

</td>

<td>

<fmt:formatDate value=*"*${ order.payTime }*"*

pattern=*"yyyy-MM-dd HH:mm:ss"*/>

</td>

<td>${ order.payType }</td>

<td>${ order.price }</td>

<td>

<a href=*"orderDelete?id=*${ order.id }*"*>删除</a>

|

<a href=*"orderInfo?id=*${ order.id }*"*>修改</a>

</td>

</tr>

</c:forEach>

</table>

</body>

# 新增订单信息

新增订单信息开发步骤：

1、在订单信息列表页面中，点击【新增订单】，查询所有门店，将所有门店信息带到订单新增页面【order\_add.jsp】

2、在【order\_add.jsp】中输入新增的订单信息，点击【提交】将订单信息提交给OrderController的【orderAdd】方法

3、在【orderAdd】方法中，通过参数接收浏览器提交过来的订单信息，调用service层，service层再调用dao层将订单信息保存到数据库中，

4、新增成功后，重定向到订单列表页面，显示所有订单（即重定向到OrderController的【orderList】方法，先查询所有订单，再将所有订单信息带到JSP显示）

## dao层代码实现

1、更新resources/mybatis/mapper/OrderrMapper.xml文件，添加“新增订单信息”SQL。

<!-- 2.新增订单信息 -->

<insert id=*"addOrder"*>

insert into tb\_order

values(#{id}, #{doorId}, #{orderNo}, #{orderType}, #{pnum},

#{cashier}, #{orderTime}, #{payTime}, #{payType}, #{price})

</insert>

2、更新com.tedu.dao.OrderMapper接口，添加 addOrder方法，新增订单信息。

/\*\*

\* 2.新增订单信息

\* **@param** Order

\*/

**public** **void** addOrder(Order Order);

## service层代码实现

**1、更新com.tedu.service.OrderService接口，添加 addOrder方法，新增订单信息**

/\*\*

\* 2.新增订单信息

\* **@param** order

\*/

**public** **void** addOrder(Order order);

**2、更新com.tedu.service.OrderServiceImpl类，实现父接口中的 addOrder方法：调用dao层的addOrder方法新增订单信息**

@Override

**public** **void** addOrder(Order order) {

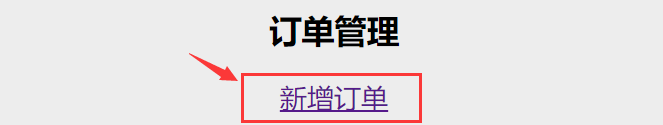
//调用OrderMapper的addOrder方法，新增订单信息

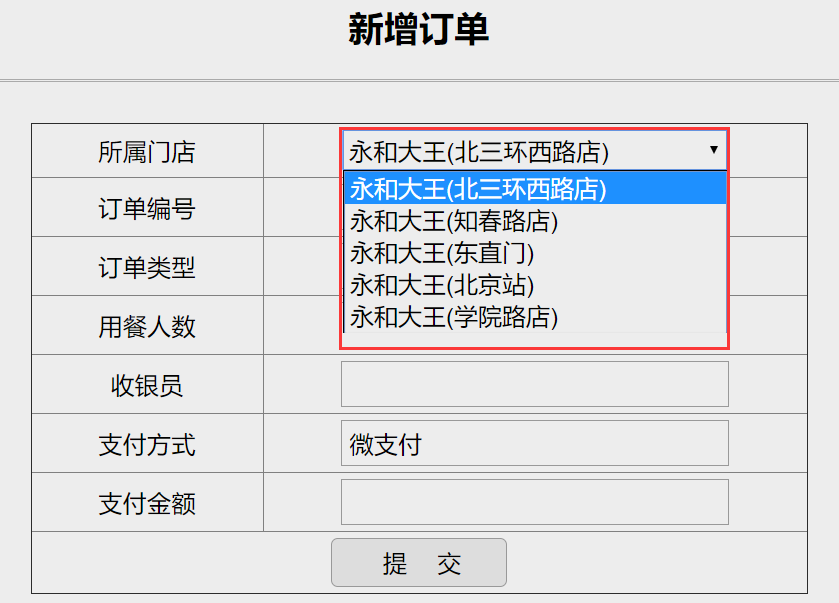
orderMapper.addOrder(order);

}

## Controller层代码实现

**1、修改"新增订单"链接地址，点击链接访问 /findAllDoorToOrderAdd 地址，查询所有门店信息，将所有门店信息转发带到order\_add.jsp页面中，并在下拉框中进行回显。**





2、更新com.tedu.controller.OrderController类，添加并实现toDoorAdd方法：调用DoorService层的findAll方法查询所有门店信息。

@Controller

**public** **class** OrderController {

@Autowired //自动注入(由spring创建当前对象)

**private** OrderService orderService;

...

/\*\* 2.1.添加订单信息 -- 查询所有门店并跳转到添加订单页面 \*/

@RequestMapping("/findAllDoorToOrderAdd")

**public** String toDoorAdd(Model model){

//1.调用doorService的findAll方法, 查询所有门店信息

List<Door> doorList = doorService.findAll();

//2.将所有门店List存入model中

model.addAttribute("doorList", doorList);

**return** "order\_add";

}

}

3、更新com.tedu.controller.OrderController类，添加并实现OrderAdd方法：调用service层的addOrder方法新增订单信息。

/\*\* 2.添加订单信息 \*/

@RequestMapping("/orderAdd")

**public** String orderAdd(Order order){

//1.调用service层的addOrder方法，新增订单信息

orderService.addOrder(order);

//2.重定向到订单列表页面, 显示所有订单信息

**return** "redirect:/orderList";

}

# 删除订单信息

删除订单信息开发步骤：

1、在订单信息列表页面中，点击订单信息后的【删除】按钮，访问OrderController中的【orderDelete】方法，访问的同时将所要删除的订单id一同提交给服务器。

2、在【orderDelete】方法中，通过参数接收订单id，调用service层，service层再调用dao层通过id删除指定的订单信息。

3、删除成功后，重定向到订单列表页面，显示所有订单（即重定向到OrderController的【orderList】方法，先查询所有订单，再将所有订单信息带到JSP显示）。

## dao层代码实现

1、更新resources/mybatis/mapper/OrderMapper.xml文件，添加“删除订单信息”SQL。

<!-- 3.根据id删除指定订单信息 -->

<insert id=*"deleteById"*>

delete from tb\_order where id=#{id}

</insert>

2、更新com.tedu.dao.OrderMapper接口，添加 deleteById方法，根据id删除订单信息。

/\*\*

\* 3.根据id删除指定的订单信息

\* **@param** id

\*/

**public** **void** deleteById(Integer id);

## service层代码实现

1、更新com.tedu.service.OrderService接口，添加 deleteById方法，根据id删除订单信息

/\*\*

\* 3.根据id删除指定的订单信息

\* **@param** id

\*/

**public** **void** deleteById(Integer id);

2、更新com.tedu.service.OrderServiceImpl类，实现父接口中的 deleteById方法：调用dao层的deleteById方法，根据id删除订单信息

@Override

**public** **void** deleteById(Integer id) {

//调用OrderMapper的deleteById方法，根据id删除指定的订单信息

orderMapper.deleteById(id);

}

## Controller层代码实现

1、更新com.tedu.controller.OrderController类，添加并实现orderDelete方法：调用service层的deleteById方法删除订单信息

/\*\* 3.根据id删除订单信息 \*/

@RequestMapping("/orderDelete")

**public** String orderDelete(Integer id){

//1.调用service层的deleteById方法，删除指定id的订单信息

orderService.deleteById(id);

//2.重定向到订单列表页面, 显示所有订单信息

**return** "redirect:/orderList";

}

# 更新订单信息

更新订单信息开发步骤：

1、在订单信息列表页面中，点击订单信息后的【修改】按钮，访问OrderController中的【orderInfo】方法，访问的同时将所要查询的订单id一同提交给服务器。

2、在【orderInfo】方法中，通过参数接收订单id，调用service层，service层再调用dao层通过id查询指定的订单信息（Order）并返回

3、在【orderInfo】方法中，将返回的订单信息（Order）存入Model中，并转向order\_update.jsp（即 return "order\_update"），将订单信息带到jsp页面进行回显

4、在原有订单信息的基础上，对订单信息进行修改，点击【提交】将修改后的订单信息提交给OrderController的【orderUpdate】方法

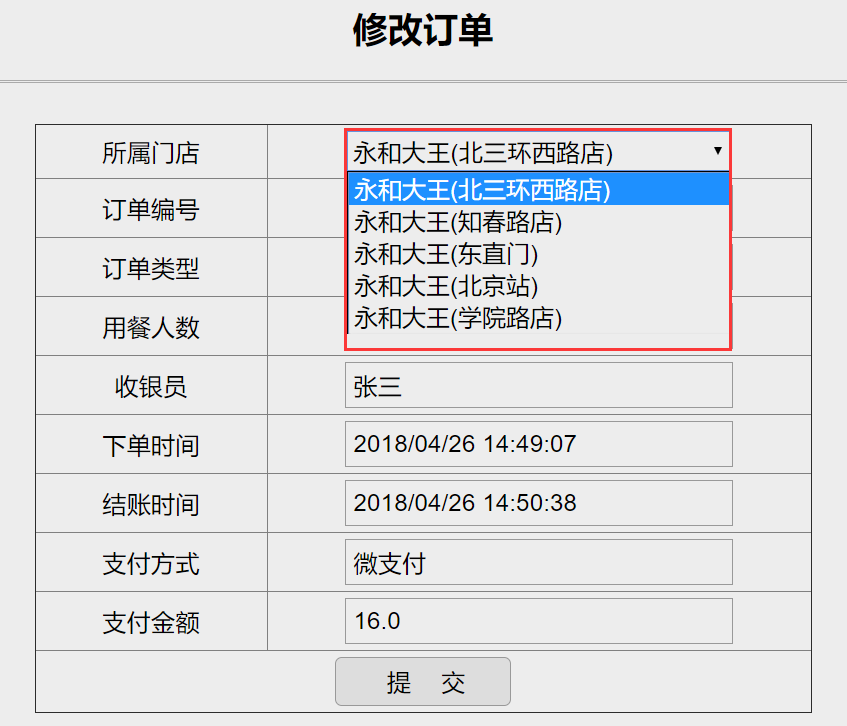
5、在【orderUpdate】方法中，通过参数接收浏览器提交过来的订单信息，调用service层，service层再调用dao层根据id修改订单信息。

6、修改成功后，重定向到订单列表页面，显示所有订单（即重定向到OrderController的【orderList】方法，先查询所有订单，再将所有订单信息带到JSP显示）

## dao层代码实现（查询订单）

修改订单信息需要在原有订单信息的基础上进行修改，因此在修改之前，需要先查询出原有订单信息，在原有订单信息的基础上修改后，再将修改后的订单信息提交给服务器端，服务器端将修改后的订单信息更新到数据库中





**1、更新resources/mybatis/mapper/OrderMapper.xml文件，添加“根据id查询订单信息”SQL。**

<!-- 4.根据id查询指定订单信息 -->

<select id=*"findById"* resultMap=*"orderRM"*>

select \* from tb\_order where id=#{id}

</select>

2、更新com.tedu.dao.OrderMapper接口，添加 findById方法，根据id查询订单信息。

/\*\*

\* 4.根据id查询订单信息

\* **@param** id

\*/

**public** Order findById(Integer id);

## service层代码实现（查询门店）

1、更新com.tedu.service.OrderService接口，添加 findById方法，根据id查询订单信息

/\*\*

\* 4.根据id查询指定的订单信息

\* **@param** id

\* **@return**

\*/

**public** Order findById(Integer id);

**2、更新com.tedu.service.OrderServiceImpl类，实现父接口中的 updateById方法：调用dao层的updateById方法，根据id更新订单信息**

@Override

**public** Order findById(Integer id) {

//调用OrderMapper的findById方法，根据id查询订单信息

Order order = orderMapper.findById(id);

**return** order;

}

## Controller层代码实现（查询订单）

1、更新com.tedu.controller.OrderController类，添加并实现OrderInfo方法：

（1）调用DoorService的findAll方法查询所有门店信息。

（2）调用service层的findById方法根据id查询订单信息。

/\*\* 4.根据id查询订单信息 \*/

@RequestMapping("/orderInfo")

**public** String orderInfo(Integer id, Model model){

//1.调用doorService的findAll方法, 查询所有门店信息

List<Door> doorList = doorService.findAll();

//2.将所有门店List存入model中

model.addAttribute("doorList", doorList);

//3.调用service层的findyId方法，根据id查询订单信息

Order order = orderService.findById(id);

//4.将订单信息保存到Model中

model.addAttribute("order", order);

//5.将订单信息带到order\_update.jsp进行回显

**return** "order\_update";

}

## 修改订单之订单数据回显

1、在order\_update.jsp中显示将要修改的订单信息:

<%@ page pageEncoding=*"utf-8"*%>

<%-- 引入JSTL标签库 --%>

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>

<%@ taglib prefix=*"fmt"* uri=*"http://java.sun.com/jsp/jstl/fmt"* %>

...

...

<form action="orderUpdate" method="POST">

<!-- hidden隐藏域,在提交表单时连order.id一起提交 -->

<input type="hidden" name="id" value="${ order.id }"/>

<table border="1">

<tr>

<td width="30%">所属门店</td>

<td>

<select name="doorId">

<!-- 遍历所有门店, 显示所有门店列表 -->

<c:forEach items="${doorList}" var="door">

<option value="${door.id}">${door.name}</option>

</c:forEach>

</select>

</td>

</tr>

<tr>

<td>订单编号</td>

<td>

<input type="text" name="orderNo"

value="${ order.orderNo }"/>

</td>

</tr>

<tr>

<td>订单类型</td>

<td>

<input type="text" name="orderType"

value="${ order.orderType }"/>

</td>

</tr>

<tr>

<td>用餐人数</td>

<td>

<input type="text" name="pnum"

value="${ order.pnum }"/>

</td>

</tr>

<tr>

<td>收银员</td>

<td>

<input type="text" name="cashier"

value="${ order.cashier }"/>

</td>

</tr>

<tr>

<td>下单时间</td>

<td>

<input type="text" name="orderTime"

value='<fmt:formatDate

value="${ order.orderTime }"

pattern="yyyy/MM/dd HH:mm:ss"/>'/>

</td>

</tr>

<tr>

<td>结账时间</td>

<td>

<input type="text" name="payTime"

value='<fmt:formatDate

value="${ order.payTime }"

pattern="yyyy/MM/dd HH:mm:ss"/>'/>

</td>

</tr>

<tr>

<td>支付方式</td>

<td>

<input type="text" name="payType"

value="${ order.payType }"/>

</td>

</tr>

<tr>

<td>支付金额</td>

<td>

<input type="text" name="price"

value="${ order.price }"/>

</td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="提 交"/>

</td>

</tr>

</table>

</form>

## dao层代码实现（更新订单）

1、更新resources/mybatis/mapper/OrderMapper.xml文件，添加“根据id更新订单信息”SQL：

<!-- 5.根据id更新指定订单信息 -->

<update id=*"updaterById"*>

update tb\_order set door\_id=#{doorId}, order\_no=#{orderNo},

order\_type=#{orderType}, pnum=#{pnum}, cashier=#{cashier},

order\_time=#{orderTime}, pay\_time=#{payTime},

pay\_type=#{payType}, price=#{price} where id=#{id}

</update>

2、更新com.tedu.dao.OrderMapper接口，添加 updateById方法，根据id更新订单信息

/\*\*

\* 5.根据id更新订单信息

\* **@param** order

\*/

**public** **void** updateById(Order order);

## service层代码实现（更新订单）

1、更新com.tedu.service.OrderService接口，添加 updateById方法，根据id更新订单信息

/\*\*

\* 5.根据id更新指定订单信息

\* **@param** order

\*/

**public** **void** updateById(Order order);

2、更新com.tedu.service.OrderServiceImpl类，实现父接口中的 updateById方法：调用dao层的updateById方法，根据id更新订单信息

@Override

**public** **void** updateById(Order order) {

//调用OrderMapper的updateById方法，根据id更新订单信息

orderMapper.updateById(order);

}

## Controller层代码实现（更新订单）

1、更新com.tedu.controller.OrderController类，添加并实现orderUpdate方法：调用service层的updaeById方法删除订单信息

/\*\* 5.根据id更新订单信息 \*/

@RequestMapping("/orderUpdate")

**public** String orderUpdate(Order order){

//1.调用service层的updateById方法，根据id跟新订单信息

orderService.updateById(order);

//2.重定向到订单列表页面, 显示所有订单信息

**return** "redirect:/orderList";

}

# 扩展内容

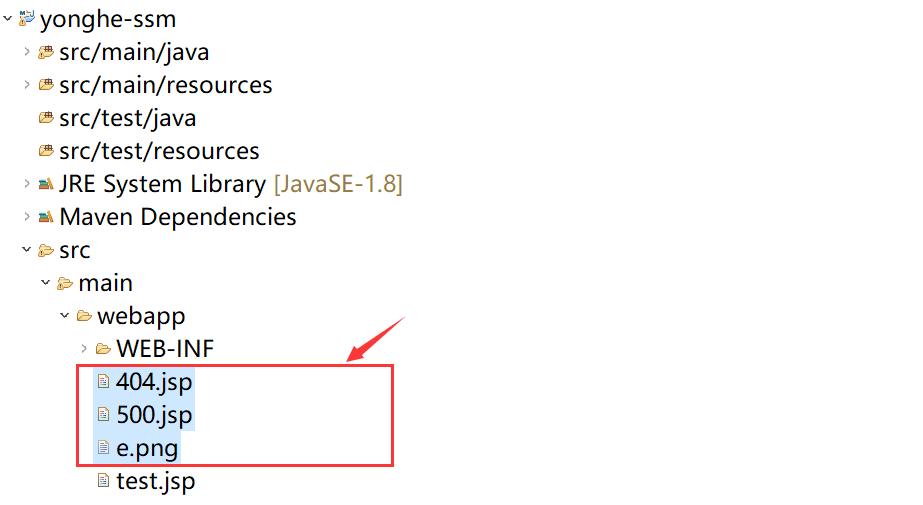
## 全站500/404提示

当通过浏览器访问服务器中不存在的资源时，浏览器的页面上会显示404；

当服务器端程序出现错误时，浏览器的页面上会显示500异常信息；

可以自己写更友好的404和500页面，代替默认的404和500页面。

1、将404.jsp和500.jsp以及e.png拷贝到webapp目录下



2、在web.xml文件中配置全站404和500跳转页面

<web-app>

...

<!-- 配置全站500异常跳转页面 -->

<error-page>

<error-code>500</error-code>

<location>/500.jsp</location>

</error-page>

<!-- 配置全站404跳转页面 -->

<error-page>

<error-code>404</error-code>

<location>/404.jsp</location>

</error-page>

...

</web-app>