# SringMVC

# 第一天springMVC

## SpringMVC介绍

SpirngMVC是什么：是一个常用的表现层的框架，用于我们在web开发中，常用的参数传递，请求接收，请求响应，页面跳转等常用功能。



## SpringMVC的处理流程

前端控制器

DispatcherServlet

Handler

处理器

请求业务处理

用户请求

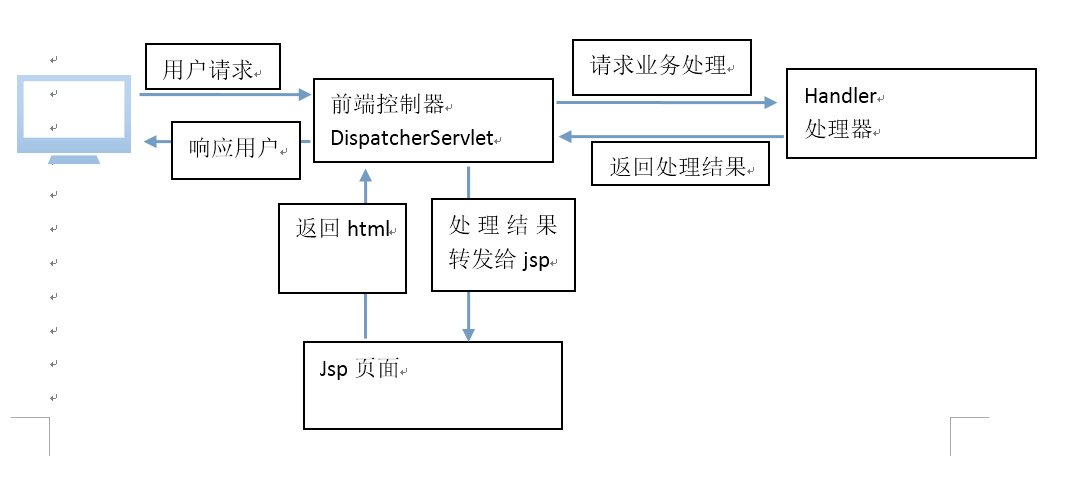
返回处理结果

Jsp页面

处理结果转发给jsp

返回html

响应用户

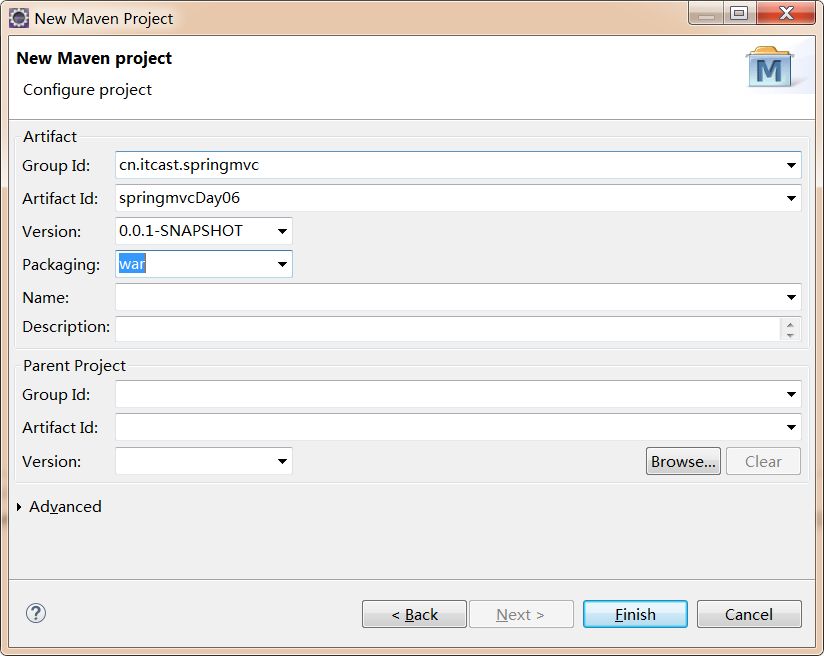


## springMVC的入门案例

### springMVC的入门案例环境搭建

环境准备：jdk1.8+eclipse-MARS+springMVC

#### 第一步：创建eclipseWEB工程



#### 第二步：解决工程创建之后的两个问题

第一个问题：jdk版本过低问题

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

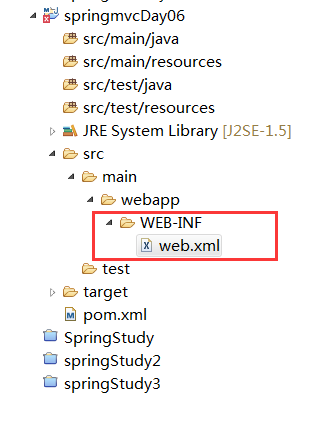
</configuration>

</plugin>

</plugins>

</build>

第二个问题：web.xml丢失问题



创建web.xml

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

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

xmlns="http://java.sun.com/xml/ns/javaee"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

version="2.5">

<display-name>01\_helloWorld</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

</web-app>

#### 第三步：导入springMVC必须依赖的jar包

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-expression</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>javax.servlet.jsp.jstl</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>3.0-alpha-1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.1</version>

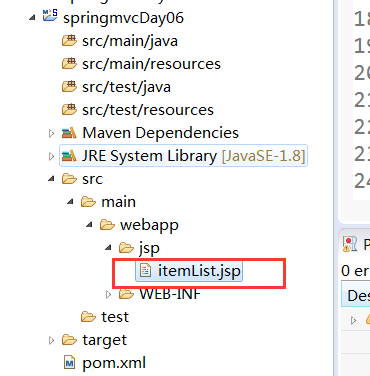
<scope>provided</scope>

</dependency>

</dependencies>

#### 第四步：需求定义

实现商品列表的静态数据展示



#### 第五步：创建springmvc.xml并且进行配置

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

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

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

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"*

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

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

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

xsi:schemaLocation=*"*

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

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

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

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

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

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

</beans>

#### 第六步：配置springMVC的前端控制器

<!-- The front controller of this Spring Web application, responsible for handling all application requests -->

<servlet>

<servlet-name>springDispatcherServlet</servlet-name>

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

<init-param>

<!-- 指定springMVC的配置文件所在的路径，如果不指定，默认找 ： /WEB-INF/${servlet-name}-servlet.xml -->

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

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

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- Map all requests to the DispatcherServlet for handling -->

<servlet-mapping>

<servlet-name>springDispatcherServlet</servlet-name>

<!-- 这个url-pattern表示你要拦截的哪些请求 -->

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

</servlet-mapping>

#### 第七步：创建我们的ItemController对象

**package** cn.itcast.springmvc.controller;

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

@Controller

**public** **class** ItemController {

}

定义pojo对象

**public** **class** Items {

**private** **int** id;

**private** String name;

**private** Double price;

**private** Date createtime;

**private** String detail;

**public** **int** getId() {

**return** id;

}

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

**this**.id = id;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** Double getPrice() {

**return** price;

}

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

**this**.price = price;

}

**public** Date getCreatetime() {

**return** createtime;

}

**public** **void** setCreatetime(Date createtime) {

**this**.createtime = createtime;

}

**public** String getDetail() {

**return** detail;

}

**public** **void** setDetail(String detail) {

**this**.detail = detail;

}

**public** Items(**int** id, String name, Double price, Date createtime, String detail) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.price = price;

**this**.createtime = createtime;

**this**.detail = detail;

}

}

#### 第八步：开发我们的springMVC的代码

/\*\*

\* 在springMVC当中，通过ModelAndView这个对象来表示模型和视图的一个映射关系

\* **@return**

\* **@RequestMapping** 这个注解的作用就是将我们的请求与我们执行的方法进行映射

\*/

@RequestMapping("/itemList.action")

**public** ModelAndView getItemList(){

ModelAndView view = **new** ModelAndView();

//addObject这个方法就相当于把我们的数据封装到我们的request域当中去了。

List<Items> list = **new** ArrayList<Items>();

list.add(**new** Items(1, "imac1", 20000d, **new** Date(), "买买买，毫不留情"));

list.add(**new** Items(2, "imac2", 20000d, **new** Date(), "买买买，毫不留情"));

list.add(**new** Items(3, "imac3", 20000d, **new** Date(), "买买买，毫不留情"));

list.add(**new** Items(4, "imac4", 20000d, **new** Date(), "买买买，毫不留情"));

view.addObject("itemList", list);

//通过setViewName来指定我们要跳转的jsp页面

view.setViewName("/WEB-INF/jsp/itemList.jsp");

**return** view;

}

#### 第九步：访问测试

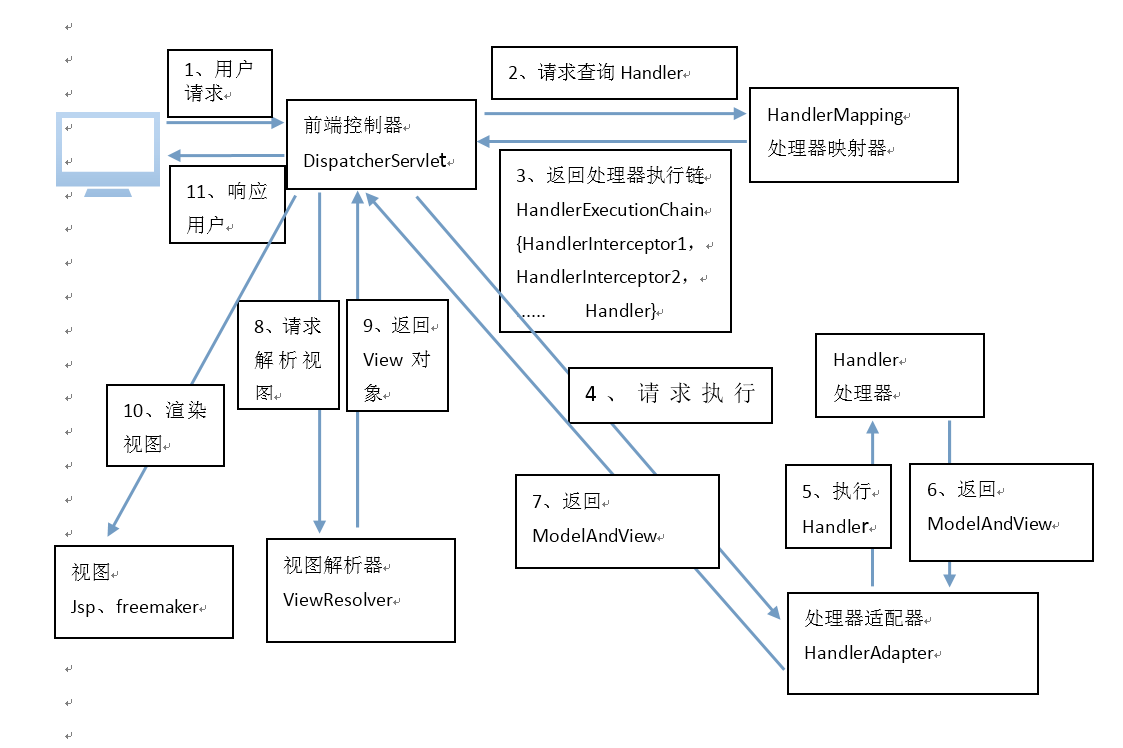
将我们的项目加入到我们的tomcat当中，并启动我们的tomcat

访问测试

<http://localhost:8080/springmvcDay06/itemList.action>

## springMVC的执行流程

### 框架结构



### 架构流程

1. 用户发送请求至前端控制器DispatcherServlet
2. DispatcherServlet收到请求调用HandlerMapping处理器映射器。
3. 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
4. DispatcherServlet通过HandlerAdapter处理器适配器调用处理器
5. 执行处理器(Controller，也叫后端控制器)。
6. Controller执行完成返回ModelAndView
7. HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet
8. DispatcherServlet将ModelAndView传给ViewReslover视图解析器
9. ViewReslover解析后返回具体View
10. DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。
11. DispatcherServlet响应用户

### 组件说明

以下组件通常使用框架提供实现：

* DispatcherServlet：前端控制器

用户请求到达前端控制器，它就相当于mvc模式中的c，dispatcherServlet是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet的存在降低了组件之间的耦合性。

* HandlerMapping：处理器映射器

HandlerMapping负责根据用户请求找到Handler即处理器，springmvc提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

* Handler：处理器

Handler 是继DispatcherServlet前端控制器的后端控制器，在DispatcherServlet的控制下Handler对具体的用户请求进行处理。

由于Handler涉及到具体的用户业务请求，所以一般情况需要程序员根据业务需求开发Handler。

* HandlAdapter：处理器适配器

通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

* View Resolver：视图解析器

View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。

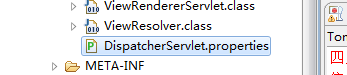
* View：视图

springmvc框架提供了很多的View视图类型的支持，包括：jstlView、freemarkerView、pdfView等。我们最常用的视图就是jsp。

一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面。

|  |
| --- |
| 说明：在springmvc的各个组件中，处理器映射器、处理器适配器、视图解析器称为springmvc的三大组件。  需要用户开放的组件有handler、view |

### 框架默认加载组件



## 注解映射器和适配器

### 组件扫描器

使用组件扫描器省去在spring容器配置每个controller类的繁琐。使用<context:component-scan>自动扫描标记@controller的控制器类，配置如下：

<!-- 扫描controller注解,多个包中间使用半角逗号分隔 -->

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

### RequestMappingHandlerMapping

注解式处理器映射器，对类中标记@ResquestMapping的方法进行映射，根据ResquestMapping定义的url匹配ResquestMapping标记的方法，匹配成功返回HandlerMethod对象给前端控制器，HandlerMethod对象中封装url对应的方法Method。

从spring3.1版本开始，废除了DefaultAnnotationHandlerMapping的使用，推荐使用RequestMappingHandlerMapping完成注解式处理器映射。

配置如下：

<!--注解映射器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"*/>

注解描述：

@RequestMapping：定义请求url到处理器功能方法的映射

### RequestMappingHandlerAdapter

注解式处理器适配器，对标记@ResquestMapping的方法进行适配。

从spring3.1版本开始，废除了AnnotationMethodHandlerAdapter的使用，推荐使用RequestMappingHandlerAdapter完成注解式处理器适配。

配置如下：

<!--注解适配器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"*/>

### <mvc:annotation-driven>

springmvc使用<mvc:annotation-driven>自动加载RequestMappingHandlerMapping和RequestMappingHandlerAdapter，可用在springmvc.xml配置文件中使用<mvc:annotation-driven>替代注解处理器和适配器的配置。

## 视图解析器

在springmvc.xml文件配置如下：

|  |
| --- |
| <bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"viewClass"*  value=*"org.springframework.web.servlet.view.JstlView"* />  <property name=*"prefix"* value=*"/WEB-INF/jsp/"* />  <property name=*"suffix"* value=*".jsp"* />  </bean> |

InternalResourceViewResolver：支持JSP视图解析

viewClass：JstlView表示JSP模板页面需要使用JSTL标签库，所以classpath中必须包含jstl的相关jar 包。此属性可以不设置，默认为JstlView。

prefix 和suffix：查找视图页面的前缀和后缀，最终视图的址为：

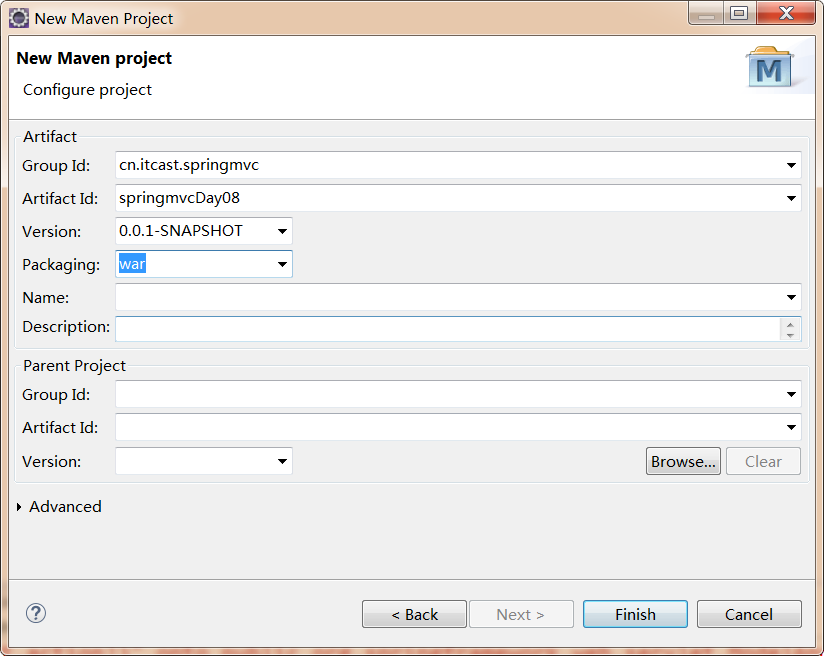
前缀+**逻辑视图名**+后缀，逻辑视图名需要在controller中返回ModelAndView指定，比如逻辑视图名为hello，则最终返回的jsp视图地址 “WEB-INF/jsp/hello.jsp”

## 四、三大框架整合

需求，实现从数据库当中查询出商品列表信息，展现到页面上

开发环境：srpingMVC 4.2.4 + spring 4.2.4+ mybatis 3.2.7+maven整合开发我们的商品列表页面和商品编辑页面

### 第一步：maven项目创建



### 第二步：解决jdk版本过低问题以及web.xml丢失问题

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

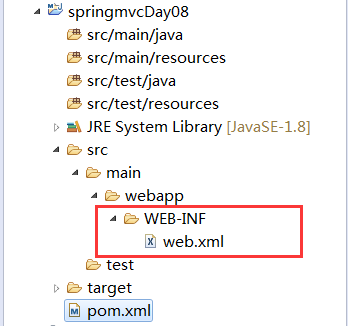
<encoding>utf-8</encoding>

</configuration>

</plugin>

</plugins>

</build>



创建web.xml

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

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

xmlns="http://java.sun.com/xml/ns/javaee"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

version="2.5">

<display-name>01\_helloWorld</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

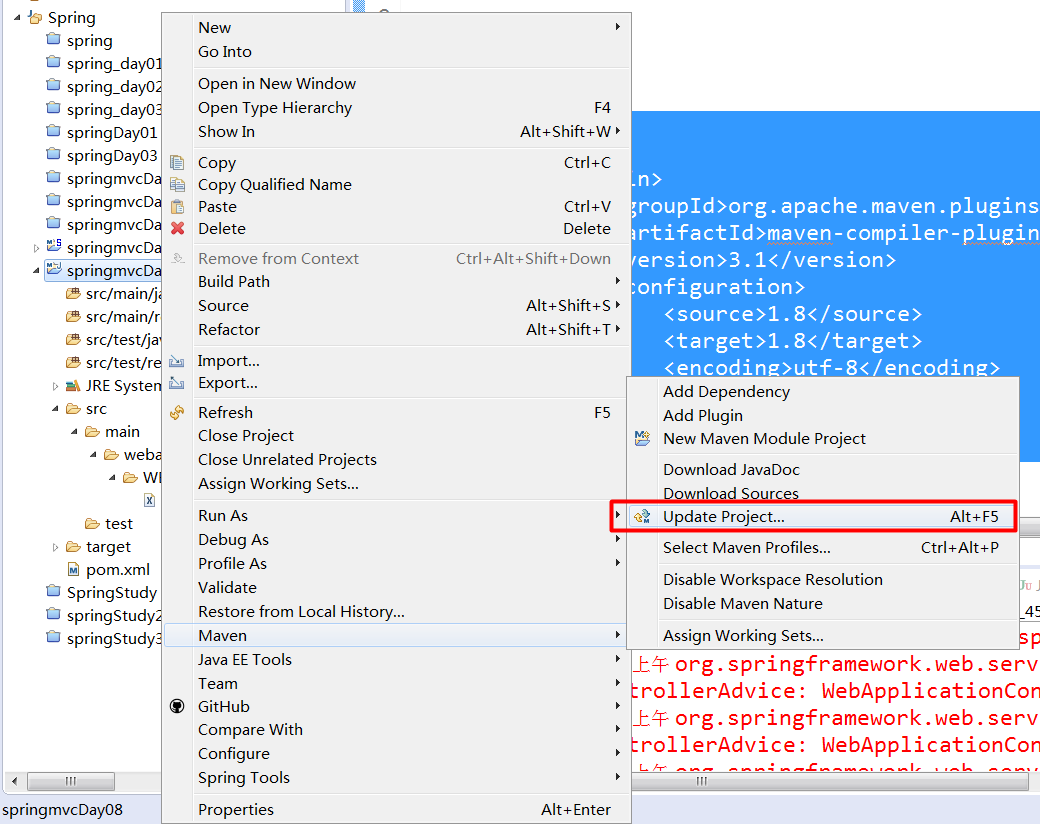
<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

</web-app>



### 第三步：导入相应的jar包

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-expression</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>aopalliance</groupId>

<artifactId>aopalliance</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.6.8</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>4.2.4.RELEASE</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

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

<version>5.1.38</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.2.2</version>

</dependency>

<dependency>

<groupId>commons-pool</groupId>

<artifactId>commons-pool</artifactId>

<version>1.5.3</version>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.0.18</version>

</dependency>

<dependency>

<groupId>c3p0</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.1.2</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>2.2.2</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.2.7</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-spring</artifactId>

<version>1.2.2</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.5</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.5</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>3.0-alpha-1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.1</version>

<scope>provided</scope>

</dependency>

</dependencies>

### 第四步：配置MyBatis的xml文件

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

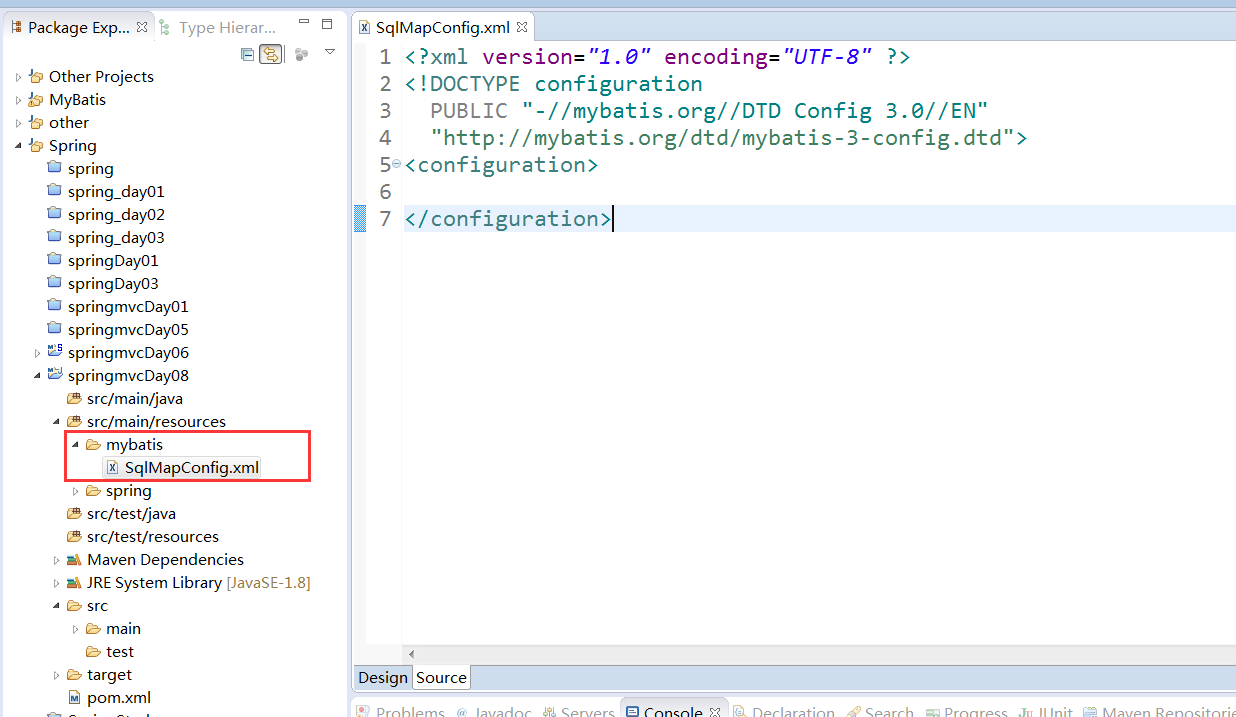
<!DOCTYPE configuration

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

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

<configuration>

</configuration>



### 第五步：配置applicationContext.xml

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

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

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

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"*

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

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

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

xsi:schemaLocation=*"*

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

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

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

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

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

<!-- 包扫描的配置 -->

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

<!-- 数据库连接池 -->

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

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

<property name=*"driverClassName"* value=*"${jdbc.driver}"*></property>

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

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

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

</bean>

<!-- sqlSessionFactoryBean -->

<bean class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<property name=*"configLocation"* value=*"classpath:mybatis/SqlMapConfig.xml"*></property>

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

</bean>

<!-- 设置我们的mapper文件的扫描路径 -->

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

<property name=*"basePackage"* value=*"cn.itcast.springmvc.mapper"*></property>

</bean>

<!-- 事物管理配置 -->

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

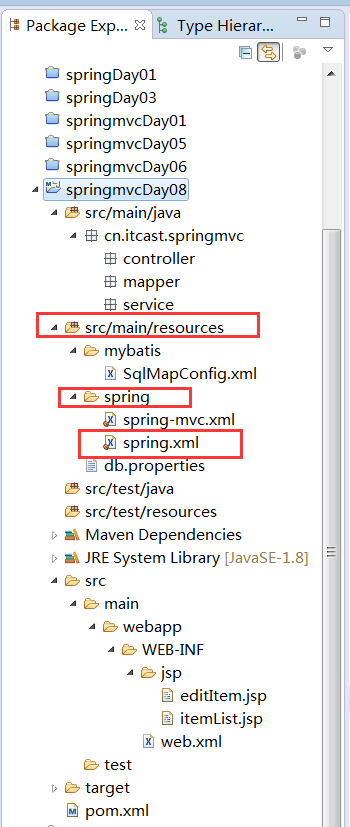
<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<!—使用注解的方式来开启事物 -->

<tx:annotation-driven/>

</beans>



### 第六步：配置spring-mvc.xml

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

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

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

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"*

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

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

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

xsi:schemaLocation=*"*

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

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

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

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

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

<!-- 配置包扫描的路径 -->

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

</context:component-scan>

<!-- 配置适配器和映射器 -->

<mvc:annotation-driven/>

<!-- 配置我们的视图解析器 -->

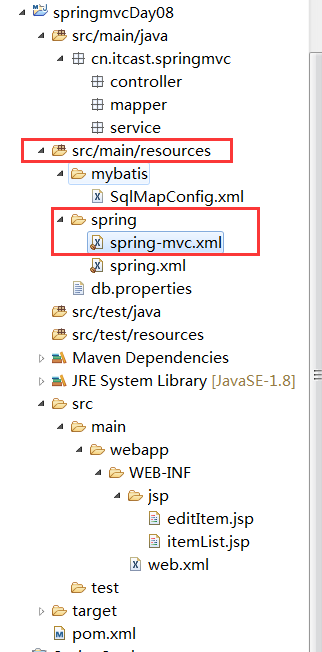
<bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

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

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

</bean>

</beans>



### 第七步：配置web.xml

<!-- The front controller of this Spring Web application, responsible for handling all application requests -->

<!-- 配置springMVC的转发器 -->

<servlet>

<servlet-name>springDispatcherServlet</servlet-name>

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

<init-param>

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

<param-value>classpath:spring/spring-mvc.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- Map all requests to the DispatcherServlet for handling -->

<servlet-mapping>

<servlet-name>springDispatcherServlet</servlet-name>

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

</servlet-mapping>

<!-- 配置spring的容器的加载监听 -->

<!-- needed for ContextLoaderListener -->

<context-param>

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

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

</context-param>

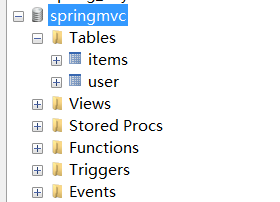
<!-- Bootstraps the root web application context before servlet initialization -->

<listener>

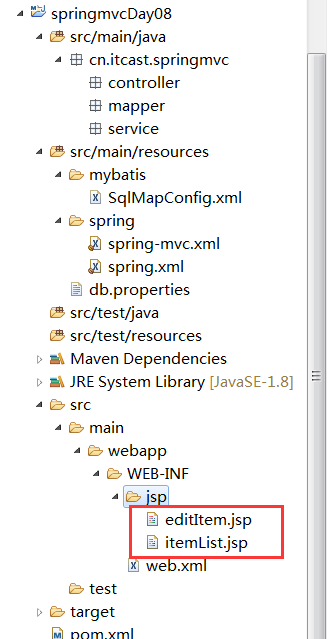
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

### 第八步：准备数据库



### 第九步：将jsp页面放入到我们的工程下面



### 第十步：controller实现

@Controller

**public** **class** ItemController {

@Autowired

**private** ItemService itemService;

@RequestMapping("/itemList.action")

**public** ModelAndView getItemList(){

ModelAndView view = **new** ModelAndView();

List<Items> itemList = itemService.getItemList();

view.addObject("itemList", itemList);

view.setViewName("itemList");

**return** view;

}

}

### 第十一步：service层实现

@Service

@Transactional

**public** **class** ItemServiceImpl **implements** ItemService {

@Autowired

**private** ItemsMapper itemsMapper;

@Override

**public** List<Items> getItemList() {

ItemsExample example = **new** ItemsExample();

List<Items> selectByExampleWithBLOBs = itemsMapper.selectByExampleWithBLOBs(example);

**return** selectByExampleWithBLOBs;

}

}

### 第十二步：dao层实现

通过mybatis的反向工程生成即可。

### 第十三步：加载项目，浏览器访问

<http://localhost:8080/springmvcDay08/itemList.action>

## 五、springMVC当中的参数绑定

### 需求：

打开商品编辑页面，展示商品信息。

### 需求分析

编辑商品信息，需要根据商品id查询商品信息，然后展示到页面。

请求的url：/itemEdit.action

参数：id（商品id）

响应结果：商品编辑页面，展示商品详细信息。

#### 第一步：定义ItemServiceImpl实现类中的方法

@Override

**public** Items getItemById(**int** id) {

**return** itemsMapper.selectByPrimaryKey(id);

}

#### 第二步：定义请求路径与对应的请求方法

@RequestMapping("/itemEdit.action")

**public** ModelAndView getItemById(HttpServletRequest request,HttpServletResponse response,HttpSession session){

ModelAndView view = **new** ModelAndView();

String parameter = request.getParameter("id");

Integer id = **null**;

**if**(**null** != parameter && "" != parameter){

id=Integer.*parseInt*(parameter);

}**else**{

**return** **null**;

}

Items itemById = itemService.getItemById(id);

view.addObject("item", itemById);

view.setViewName("editItem");

**return** view;

}

### 默认支持的参数类型

处理器形参中添加如下类型的参数处理适配器会默认识别并进行赋值。

#### HttpServletRequest

通过request对象获取请求信息

#### HttpServletResponse

通过response处理响应信息

#### HttpSession

通过session对象得到session中存放的对象

#### Model/ModelMap

ModelMap是Model接口的实现类，通过Model或ModelMap向页面传递数据，如下：

//调用service查询商品信息

Items item = itemService.findItemById(id);

model.addAttribute("item", item);

页面通过${item.XXXX}获取item对象的属性值。

使用Model和ModelMap的效果一样，如果直接使用Model，springmvc会实例化ModelMap。

|  |
| --- |
| 如果使用Model则可以不使用ModelAndView对象，Model对象可以向页面传递数据，View对象则可以使用String返回值替代。不管是Model还是ModelAndView，其本质都是使用Request对象向jsp传递数据。 |

如果使用Model则方法可以改造成：

|  |
| --- |
| @RequestMapping("/itemEdit")  **public** String itemEdit(HttpServletRequest request, Model model) {  //从Request中取id  String strId = request.getParameter("id");  Integer id = **null**;  //如果id有值则转换成int类型  **if** (strId != **null** && !"".equals(strId)) {  id = **new** Integer(strId);  } **else** {  //出错  **return** **null**;  }  Items items = itemService.getItemById(id);  //创建ModelAndView  //ModelAndView modelAndView = new ModelAndView();  //向jsp传递数据  //modelAndView.addObject("item", items);  model.addAttribute("item", items);  //设置跳转的jsp页面  //modelAndView.setViewName("editItem");  //return modelAndView;  **return** "editItem";  } |

### SpringMVC简单数据类型绑定

当请求的参数名称和处理器形参**名称一致**时会将请求参数与形参进行绑定。从Request取参数的方法可以进一步简化。

|  |
| --- |
| @RequestMapping("/itemEdit")  **public** String itemEdit(Integer id, Model model) {  Items items = itemService.getItemById(id);  //向jsp传递数据  model.addAttribute("item", items);  //设置跳转的jsp页面  **return** "editItem";  } |

#### 支持的数据类型

参数类型推荐使用包装数据类型，因为基础数据类型不可以为null

整形：Integer、int

字符串：String

单精度：Float、float

双精度：Double、double

布尔型：Boolean、boolean

说明：对于布尔类型的参数，请求的参数值为true或false。

处理器方法：

public String editItem(Model model,Integer id,Boolean status) throws Exception

请求url：

http://localhost:8080/xxx.action?id=2&status=false

### @RequestParam 的使用

使用@RequestParam常用于处理简单类型的绑定。

value：参数名字，即入参的请求参数名字，如value=“item\_id”表示请求的参数区中的名字为item\_id的参数的值将传入；

required：是否必须，默认是true，表示请求中一定要有相应的参数，否则将报；

TTP Status 400 - Required Integer parameter 'XXXX' is not present

defaultValue：默认值，表示如果请求中没有同名参数时的默认值

定义如下：

public String editItem(@RequestParam(value="item\_id",required=true) String id) {

}

形参名称为id，但是这里使用value=" item\_id"限定请求的参数名为item\_id，所以页面传递参数的名必须为item\_id。

注意：如果请求参数中没有item\_id将跑出异常：

HTTP Status 500 - Required Integer parameter 'item\_id' is not present

这里通过required=true限定item\_id参数为必需传递，如果不传递则报400错误，可以使用defaultvalue设置默认值，即使required=true也可以不传item\_id参数值

### 绑定pojo类型

#### 需求

将页面修改后的商品信息保存到数据库中

#### 需求分析

请求的url：/updateitem.action

参数：表单中的数据。

响应内容：更新成功页面

#### 使用pojo接收表单数据

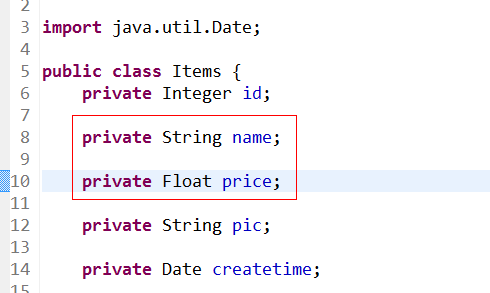
如果提交的参数很多，或者提交的表单中的内容很多的时候可以使用pojo接收数据。要求pojo对象中的属性名和表单中input的name属性一致。

页面定义如下;

<input type="text" name="name"/>

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

Pojo定义：



请求的参数名称和pojo的属性名称一致，会自动将请求参数赋值给pojo的属性。

|  |
| --- |
| @RequestMapping("/updateitem")  **public** String updateItem(Items items) {  itemService.updateItem(items);  **return** "success";  } |

|  |
| --- |
| 注意：提交的表单中不要有日期类型的数据，否则会报400错误。如果想提交日期类型的数据需要用到后面的自定义参数绑定的内容。 |

#### 解决post乱码问题

在web.xml中加入：

|  |
| --- |
| <filter>  <filter-name>CharacterEncodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>utf-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>CharacterEncodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

以上可以解决post请求乱码问题。

对于get请求中文参数出现乱码解决方法有两个：

修改tomcat配置文件添加编码与工程编码一致，如下：

<Connector URIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>

另外一种方法对参数进行重新编码：

String userName = new

String(request.getParamter("userName").getBytes("ISO8859-1"),"utf-8")

ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码

### 绑定包装pojo

#### 需求

使用包装的pojo接收商品信息的查询条件。

#### 需求分析

包装对象定义如下：

**Public class** QueryVo {

**private** Items items;

}

页面定义：

<input type="text" name="items.name" />

<input type="text" name="items.price" />

Controller方法定义如下：

public String useraddsubmit(Model model,QueryVo queryVo)throws Exception{

System.out.println(queryVo.getItems());

#### 接收查询条件

|  |
| --- |
| @RequestMapping("/queryitem")  **public** String queryItem(QueryVo queryVo) {  System.***out***.println(queryVo.getItems().getName());  System.***out***.println(queryVo.getItems().getPrice());  **return** **null**;  } |

### 自定义参数绑定

#### 需求

在商品修改页面可以修改商品的生产日期，并且根据业务需求自定义日期格式。

#### 需求分析

由于日期数据有很多种格式，所以springmvc没办法把字符串转换成日期类型。所以需要自定义参数绑定。前端控制器接收到请求后，找到注解形式的处理器适配器，对RequestMapping标记的方法进行适配，并对方法中的形参进行参数绑定。在springmvc这可以在处理器适配器上自定义Converter进行参数绑定。如果使用<mvc:annotation-driven/>可以在此标签上进行扩展。

#### 自定义Converter

**public** **class** DateConverter **implements** Converter<String, Date> {

@Override

**public** Date convert(String source) {

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

**try** {

**return** simpleDateFormat.parse(source);

} **catch** (ParseException e) {

e.printStackTrace();

}

**return** **null**;

}

}

#### 配置Converter



<!-- 加载注解驱动 -->

<mvc:annotation-driven conversion-service=*"conversionService"*/>

<!-- 转换器配置 -->

<bean id=*"conversionService"*

class=*"org.springframework.format.support.FormattingConversionServiceFactoryBean"*>

<property name=*"converters"*>

<set>

<bean class=*"cn.itcast.springmvc.convert.DateConverter"*/>

</set>

</property>

</bean>

#### 配置方式2（了解）

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

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

xmlns:dubbo=*"http://code.alibabatech.com/schema/dubbo"* xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

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

*http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd*

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

<!-- 扫描带Controller注解的类 -->

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

<!-- 转换器配置 -->

<bean id=*"conversionService"*

class=*"org.springframework.format.support.FormattingConversionServiceFactoryBean"*>

<property name=*"converters"*>

<set>

<bean class=*"cn.itcast.springmvc.convert.DateConverter"*/>

</set>

</property>

</bean>

<!-- 自定义webBinder -->

<bean id=*"customBinder"* class=*"org.springframework.web.bind.support.ConfigurableWebBindingInitializer"*>

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

</bean>

<!--注解适配器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"*>

<property name=*"webBindingInitializer"* ref=*"customBinder"*></property>

</bean>

<!-- 注解处理器映射器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"*/>

<!-- 加载注解驱动 -->

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

<!-- 视图解析器 -->

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

<property name=*"viewClass"*

value=*"org.springframework.web.servlet.view.JstlView"* />

<!-- jsp前缀 -->

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

<!-- jsp后缀 -->

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

</bean>

</beans>

注意：此方法需要独立配置处理器映射器、适配器，不再使用<mvc:annotation-driven/>

# 第二天springMVC

## 一、高级参数绑定

### 绑定数组

#### 需求

在商品列表页面选中多个商品，然后删除。

#### 需求分析

此功能要求商品列表页面中的每个商品前有一个checkbook，选中多个商品后点击删除按钮把商品id传递给Controller，根据商品id删除商品信息。

#### 第一步：Jsp中实现：

|  |
| --- |
| <c:forEach items=*"*${itemList }*"* var=*"item"*>  <tr>  <td><input name=*"ids"* value=*"*${item.id}*"* type=*"checkbox"*></td>  <td>${item.name }</td>  <td>${item.price }</td>  <td><fmt:formatDate value=*"*${item.createtime}*"* pattern=*"yyyy-MM-dd HH:mm:ss"*/></td>  <td>${item.detail }</td>  <td><a href=*"*${pageContext.request.contextPath }*/itemEdit.action?id=*${item.id}*"*>修改</a></td>  </tr>  </c:forEach> |

生成html代码如下：

页面选中多个checkbox向controller方法传递

|  |
| --- |
| <table width="100%" border=1>  <tr>  <td>商品名称</td>  <td>商品价格</td>  <td>生产日期</td>  <td>商品描述</td>  <td>操作</td>  </tr>  <tr>  <td><input name="ids" value="1" type="checkbox"></td>  <td>台式机</td>  <td>3000.0</td>  <td>2016-02-03 13:22:53</td>  <td></td>  <td><a href="/springmvc-web/itemEdit.action?id=1">修改</a></td>  </tr>  <tr>  <td><input name="ids" value="2" type="checkbox"></td>  <td>笔记本</td>  <td>6000.0</td>  <td>2015-02-09 13:22:57</td>  <td></td>  <td><a href="/springmvc-web/itemEdit.action?id=2">修改</a></td>  </tr>  <tr>  <td><input name="ids" value="3" type="checkbox"></td>  <td>背包</td>  <td>200.0</td>  <td>2015-02-06 13:23:02</td>  <td></td>  <td><a href="/springmvc-web/itemEdit.action?id=3">修改</a></td>  </tr>  </table> |

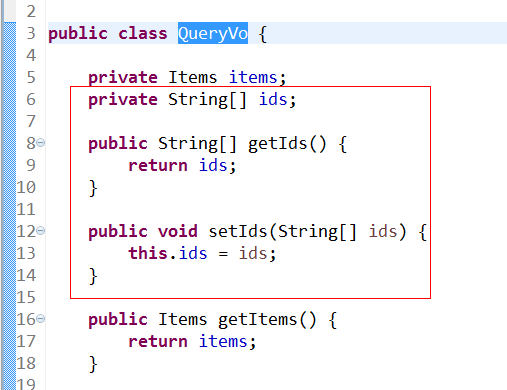
#### 第二步：Controller中接收参数

Controller方法中可以用String[]接收，或者pojo的String[]属性接收。两种方式任选其一即可。

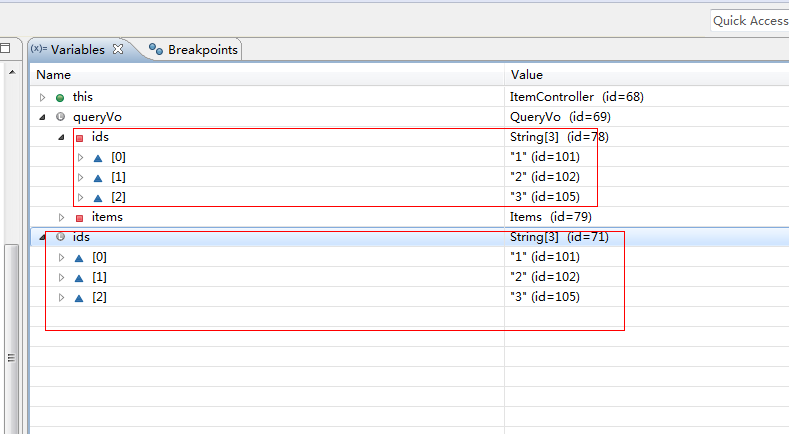
定义如下：

|  |
| --- |
| @RequestMapping("/queryitem")  **public** String queryItem(QueryVo queryVo, String[] ids) {  System.***out***.println(queryVo.getItems().getName());  System.***out***.println(queryVo.getItems().getPrice());  System.***out***.println(ids.toString());  **return** **null**;  } |

或者：



查看结果：



### 将表单的数据绑定到List

#### 需求

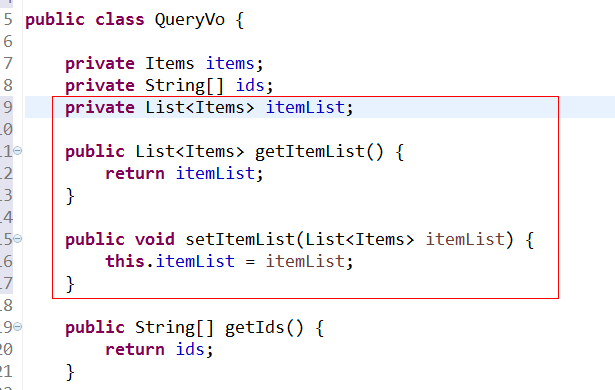
实现商品数据的批量修改。

#### 需求分析

要想实现商品数据的批量修改，需要在商品列表中可以对商品信息进行修改，并且可以批量提交修改后的商品数据。

#### 接收商品列表的pojo

List中存放对象，并将定义的List放在包装类中，使用包装pojo对象接收。



#### Jsp改造

页面定义如下：

<tr>

<td>

<input type="text" name=" itemsList[0].id" value="${item.id}"/>

</td>

<td>

<input type="text" name=" itemsList[0].name" value="${item.name }"/>

</td>

<td>

<input type="text" name=" itemsList[0].price" value="${item.price}"/>

</td>

</tr>

<tr>

<td>

<input type="text" name=" itemsList[1].id" value="${item.id}"/>

</td>

<td>

<input type="text" name=" itemsList[1].name" value="${item.name }"/>

</td>

<td>

<input type="text" name=" itemsList[1].price" value="${item.price}"/>

</td>

</tr>

Name属性必须是包装pojo的list属性+下标+元素属性。Jsp做如下改造：

|  |
| --- |
| <c:forEach items=*"*${itemList}*"* var=*"item"* varStatus=*"itemIndex"*>  <tr>  <td><input type=*"checkbox"* name=*"ids"* value=*${item.id*}/></td>  <td><input type=*"hidden"* name=*"itemList[*${itemIndex.index}*].id"* value=*"*${item.id}*"*/></td>  <td><input type=*"text"* name=*"itemList[*${itemIndex.index}*].name"* value=*"*${item.name }*"*/> </td>  <td><input type=*"text"* name = *"itemList[*${itemIndex.index}*].price"* value=*"*${item.price }*"*/> </td>  <td> <input type=*"text"* name=*"itemList[*${itemIndex.index}*].createtime"* value=*"* <fmt:formatDate value='${item.createtime}' pattern=*'yyyy-MM-dd HH:mm:ss'*/>*"*/> </td>  <td><input type=*"text"* name=*"itemList[*${itemIndex.index}*].detail"* value=*"*${item.detail }*"*/> </td>  <!-- ${item.id} -->  <td><a href=*"*${pageContext.request.contextPath }*/itemEdit.action?id=*${item.id}*"*>修改</a></td> <!-- -->  </tr>  </c:forEach> |

varStatus属性常用参数总结下：

${status.index}      输出行号，从0开始。

${status.count}      输出行号，从1开始。

${status.current}   当前这次迭代的（集合中的）项

${status.first}  判断当前项是否为集合中的第一项，返回值为true或false

${status.last}   判断当前项是否为集合中的最后一项，返回值为true或false

begin、end、step分别表示：起始序号，结束序号，跳跃步伐。

#### Contrller

|  |
| --- |
| @RequestMapping("/queryitem")  **public** String queryItem(QueryVo queryVo, String[] ids) {  System.***out***.println(queryVo.getItems().getName());  System.***out***.println(queryVo.getItems().getPrice());  System.***out***.println(ids.toString());  **return** **null**;  } |

注意：接收List类型的数据必须是pojo的属性，方法的形参为List类型无法正确接收到数据。

## 二、@RequestMapping

通过RequestMapping注解可以定义不同的处理器映射规则。

### 窄化请求映射

在class上添加@RequestMapping(url)指定通用请求前缀， 限制此类下的所有方法请求url必须以请求前缀开头，通过此方法对url进行分类管理。

如下：

@RequestMapping放在类名上边，设置请求前缀

@Controller

@RequestMapping("/item")

方法名上边设置请求映射url：

@RequestMapping放在方法名上边，如下：

@RequestMapping("/queryItem ")

访问地址为：/item/queryItem

### 请求方法限定

* 限定GET方法

@RequestMapping(method = RequestMethod.*GET*)

如果通过Post访问则报错：

HTTP Status 405 - Request method 'POST' not supported

例如：

@RequestMapping(value="/editItem",method=RequestMethod.GET)

* 限定POST方法

@RequestMapping(method = RequestMethod.*POST*)

如果通过Post访问则报错：

HTTP Status 405 - Request method 'GET' not supported

* GET和POST都可以

@RequestMapping(method={RequestMethod.GET,RequestMethod.POST})

## 三、controller方法返回值

### 返回ModelAndView

controller方法中定义ModelAndView对象并返回，对象中可添加model数据、指定view。

### 返回void

在controller方法形参上可以定义request和response，使用request或response指定响应结果：

1、使用request转向页面，如下：

request.getRequestDispatcher("页面路径").forward(request, response);

2、也可以通过response页面重定向：

response.sendRedirect("url")

3、也可以通过response指定响应结果，例如响应json数据如下：

response.setCharacterEncoding("utf-8");

response.setContentType("application/json;charset=utf-8");

response.getWriter().write("json串");

### 返回字符串

#### 逻辑视图名

controller方法返回字符串可以指定逻辑视图名，通过视图解析器解析为物理视图地址。

//指定逻辑视图名，经过视图解析器解析为jsp物理路径：/WEB-INF/jsp/item/editItem.jsp

**return** "item/editItem";

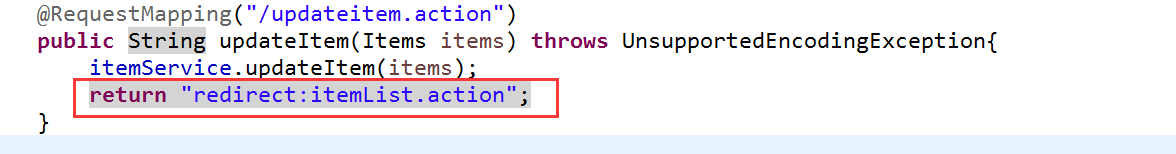
#### 请求重定向与请求转发的区别：

请求转发forward：浏览器URL地址不会发生变化，在服务器端做了一个跳转，通一个request域中

请求重定向redirect：浏览器地址会发生变化，服务器端向浏览器响应302状态码，浏览器收到302之后，重新请求服务器一个新的地址，不同的request域了

#### Redirect重定向

实现商品修改完成之后跳转到商品列表页面



Contrller方法返回结果重定向到一个url地址，如下商品修改提交后重定向到商品查询方法，参数无法带到商品查询方法中。

//重定向到queryItem.action地址,request无法带过去

**return** "redirect:queryItem.action";

redirect方式相当于“response.sendRedirect()”，转发后浏览器的地址栏变为转发后的地址，因为转发即执行了一个新的request和response。

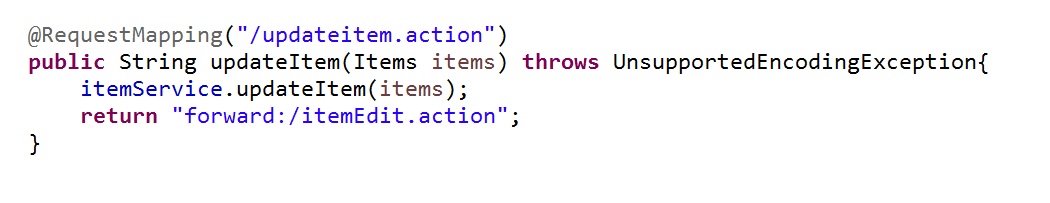
由于新发起一个request原来的参数在转发时就不能传递到下一个url，如果要传参数可以/item/queryItem.action后边加参数，如下：

/item/queryItem?...&…..

#### forward转发

实现商品修改完成之后，通过请求转发，继续跳转到修改页面

controller方法执行后继续执行另一个controller方法，如下商品修改提交后转向到商品修改页面，修改商品的id参数可以带到商品修改方法中。



//结果转发到editItem.action，request可以带过去

**return** "forward:editItem.action";

forward方式相当于“request.getRequestDispatcher().forward(request,response)”，转发后浏览器地址栏还是原来的地址。转发并没有执行新的request和response，而是和转发前的请求共用一个request和response。所以转发前请求的参数在转发后仍然可以读取到。

## 四、异常处理器

springmvc在处理请求过程中出现异常信息交由异常处理器进行处理，自定义异常处理器可以实现一个系统的异常处理逻辑。

### 异常处理思路

系统中异常包括两类：预期异常和运行时异常RuntimeException，前者通过捕获异常从而获取异常信息，后者主要通过规范代码开发、测试通过手段减少运行时异常的发生。

系统的dao、service、controller出现都通过throws Exception向上抛出，最后由springmvc前端控制器交由异常处理器进行异常处理，如下图：

Controller

客户端

Service

Dao

Springmvc

DispatcherServlet

请求

异常

HandlerExceptionResolver

异常处理器

异常

异常

### 自定义异常类

为了区别不同的异常通常根据异常类型自定义异常类，这里我们创建一个自定义系统异常，如果controller、service、dao抛出此类异常说明是系统预期处理的异常信息。

**public** **class** CustomException **extends** Exception {

/\*\* serialVersionUID\*/

**private** **static** **final** **long** *serialVersionUID* = -5212079010855161498L;

**public** CustomException(String message){

**super**(message);

**this**.message = message;

}

//异常信息

**private** String message;

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String message) {

**this**.message = message;

}

}

### 自定义异常处理器

**public** **class** CustomExceptionResolver **implements** HandlerExceptionResolver {

@Override

**public** ModelAndView resolveException(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex) {

String msg = null;

if(ex instanceof CustomerException){

CustomerException exception = (CustomerException) ex;

msg = exception.getExceptionMessage();

}else{

StringWriter s = new StringWriter();

PrintWriter printWriter = new PrintWriter(s);

ex.printStackTrace(printWriter);

msg = s.toString();

}

ModelAndView view = new ModelAndView();

view.setViewName("error");

view.addObject("message", msg);

**return** modelAndView;

}

}

|  |
| --- |
| 取异常堆栈：  **try** {    } **catch** (Exception e) {  StringWriter s = **new** StringWriter();  PrintWriter printWriter = **new** PrintWriter(s);  e.printStackTrace(printWriter);  s.toString();  } |

### 错误页面

<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*

pageEncoding=*"UTF-8"*%>

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

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

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>

<title>错误页面</title>

</head>

<body>

您的操作出现错误如下：<br/>

${message}

</body>

</html>

### 异常处理器配置

在springmvc.xml中添加：

<!-- 异常处理器 -->

<bean id=*"handlerExceptionResolver"* class=*"cn.itcast.ssm.controller.exceptionResolver.CustomExceptionResolver"*/>

### 异常测试

修改商品信息，id输入错误提示商品信息不存在。

修改controller方法“editItem”，调用service查询商品信息，如果商品信息为空则抛出异常：

// 调用service查询商品信息

Items item = itemService.findItemById(id);

**if**(item == **null**){

**throw** **new** CustomException("商品信息不存在!");

}

在service中抛出异常方法同上。

## 五、上传图片

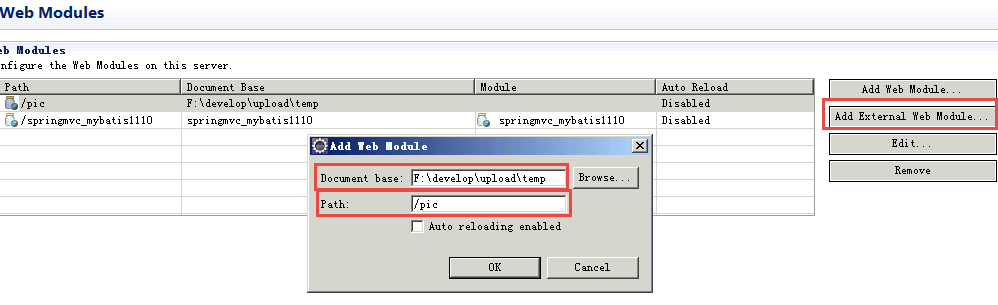
### 配置tomcat代理目录

在tomcat上配置图片虚拟目录，在tomcat下conf/server.xml中添加：

<Context docBase="F:\develop\upload\temp" path="/pic" reloadable="false"/>

访问http://localhost:8080/pic即可访问F:\develop\upload\temp下的图片。

也可以通过eclipse配置：



### 添加必须依赖jar包

*CommonsMultipartResolver*解析器依赖commons-fileupload和commons-io，加入如下jar包：

<!-- https://mvnrepository.com/artifact/commons-io/commons-io -->

<dependency>

<groupId>commons-io</groupId>

<artifactId>commons-io</artifactId>

<version>2.4</version>

</dependency>

<!-- https://mvnrepository.com/artifact/commons-fileupload/commons-fileupload -->

<dependency>

<groupId>commons-fileupload</groupId>

<artifactId>commons-fileupload</artifactId>

<version>1.3.1</version>

</dependency>

### 配置解析器

配置多媒体文件解析器

<!-- 文件上传 -->

<bean id=*"multipartResolver"*

class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>

<!-- 设置上传文件的最大尺寸为5MB -->

<property name="maxUploadSize" value="5242880"></property>

</bean>

### 图片上传

* controller：

//商品修改提交

@RequestMapping("/editItemSubmit")

**public** String editItemSubmit(Items items, MultipartFile pictureFile)**throws** Exception{

//原始文件名称

String pictureFile\_name = pictureFile.getOriginalFilename();

//新文件名称

String newFileName = UUID.*randomUUID*().toString()+pictureFile\_name.substring(pictureFile\_name.lastIndexOf("."));

//上传图片

File uploadPic = **new** java.io.File("F:/develop/upload/temp/"+newFileName);

**if**(!uploadPic.exists()){

uploadPic.mkdirs();

}

//向磁盘写文件

pictureFile.transferTo(uploadPic);

.....

* 页面：

form添加enctype="multipart/form-data"：

<form id=*"itemForm"*

action=*"*${pageContext.request.contextPath }*/item/editItemSubmit.action"*

method=*"post"* enctype=*"multipart/form-data"*>

<input type=*"hidden"* name=*"pic"* value=*"*${item.pic }*"* />

file的name与controller形参一致：

<tr>

<td>商品图片</td>

<td><c:if test=*"*${item.pic !=null}*"*>

<img src=*"/pic/*${item.pic}*"* width=*100* height=*100* />

<br />

</c:if> <input type=*"file"* name=*"pictureFile"* /></td>

</tr>

## 六、json数据交互

### @RequestBody

作用：

@RequestBody注解用于读取http请求的内容(字符串)，通过springmvc提供的HttpMessageConverter接口将读到的内容转换为json、xml等格式的数据并绑定到controller方法的参数上。

List.action?id=1&name=zhangsan&age=12

本例子应用：

@RequestBody注解实现接收http请求的json数据，将json数据转换为java对象

### @ResponseBody

作用：

该注解用于将Controller的方法返回的对象，通过HttpMessageConverter接口转换为指定格式的数据如：json,xml等，通过Response响应给客户端

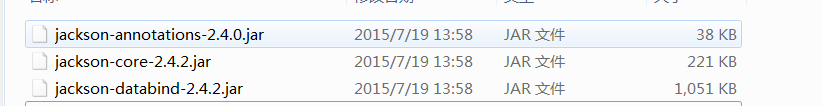
本例子应用：

@ResponseBody注解实现将controller方法返回对象转换为json响应给客户端

### 请求json，响应json实现：

#### 环境准备

Springmvc默认用MappingJacksonHttpMessageConverter对json数据进行转换，需要加入jackson的包，如下：



<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core -->

<dependency>

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

<artifactId>jackson-core</artifactId>

<version>2.8.8</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->

<dependency>

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

<artifactId>jackson-databind</artifactId>

<version>2.8.8</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-annotations -->

<dependency>

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

<artifactId>jackson-annotations</artifactId>

<version>2.8.8</version>

</dependency>

#### 配置json转换器

在注解适配器中加入messageConverters

<!--注解适配器 -->

<bean class=*"org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"*>

<property name=*"messageConverters"*>

<list>

<bean class=*"org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"*></bean>

</list>

</property>

</bean>

**注意：如果使用<mvc:annotation-driven /> 则不用定义上边的内容。**

#### controller编写

// 商品修改提交json信息，响应json信息

@RequestMapping("/editItemSubmit\_RequestJson")

@ResponseBody

**public** Items editItemSubmit\_RequestJson(@RequestBody Items items) **throws** Exception {

System.*out*.println(items);

//itemService.saveItem(items);

**return** items;

}

#### 页面js方法编写：

引入 js：

<script type="text/javascript"

src="${pageContext.request.contextPath }/js/jquery-1.4.4.min.js"></script>

//请求json响应json

**function** request\_json(){

$.ajax({

type:"post",

url:"${pageContext.request.contextPath }/item/editItemSubmit\_RequestJson.action",

contentType:"application/json;charset=utf-8",

data:'{"name":"测试商品","price":99.9}',

success:**function**(data){

alert(data);

}

});

}

#### 测试结果：



从上图可以看出请求的数据是json格式

## 七、RESTful支持

### 什么是restful？

Restful就是一个资源定位及资源操作的风格。不是标准也不是协议，只是一种风格，是对http协议的诠释。

资源定位：互联网所有的事物都是资源，要求url中没有动词，只有名词。没有参数

Url格式：http://blog.csdn.net/briblue/article/details/73824058

资源操作：使用put、delete、**post、get**，使用不同方法对资源进行操作。分别对应添加、删除、修改、查询。一般使用时还是post和get。Put和Delete几乎不使用。

### 需求

RESTful方式实现商品信息查询，返回json数据

### 添加DispatcherServlet的rest配置

<servlet>

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

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

<init-param>

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

<param-value>classpath:spring/springmvc.xml</param-value>

</init-param>

</servlet>

<servlet-mapping>

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

<!-- / ：表示拦截所有请求，但是不拦截jsp

/\* : 表示所有的请求都会拦截，包括jsp

一般都是推荐只使用 /

-->

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

</servlet-mapping>

### URL 模板模式映射

@RequestMapping(value="/ viewItems/{id}")：{×××}占位符，请求的URL可以是“/viewItems/1”或“/viewItems/2”，通过在方法中使用@PathVariable获取{×××}中的×××变量。

@PathVariable用于将请求URL中的模板变量映射到功能处理方法的参数上。

@RequestMapping("/itemEdit/{id}")

**public** String getItemEdit(Model model,@PathVariable("id")Integer id) **throws** CustomerException{

Items items = itemService.getItemById(id);

//view.addObject("item", items);

model.addAttribute("item", items);

**return** "editItem";

}

如果RequestMapping中表示为"/viewItems/{id}"，id和形参名称一致，@PathVariable不用指定名称。

商品查询的controller方法也改为rest实现：

// 查询商品列表

@RequestMapping("/queryItem")

**public** ModelAndView queryItem() **throws** Exception {

// 商品列表

List<Items> itemsList = itemService.findItemsList(**null**);

// 创建modelAndView准备填充数据、设置视图

ModelAndView modelAndView = **new** ModelAndView();

// 填充数据

modelAndView.addObject("itemsList", itemsList);

// 视图

modelAndView.setViewName("item/itemsList");

**return** modelAndView;

}

### 静态资源访问<mvc:resources>

如果在DispatcherServlet中设置url-pattern为 /则必须对静态资源进行访问处理。

spring mvc 的<mvc:resources mapping="" location="">实现对静态资源进行映射访问。

如下是对js文件访问配置：

<mvc:resources location="/js/" mapping="/js/\*\*"/>

## 八、拦截器

### 定义

Spring Web MVC 的处理器拦截器类似于Servlet 开发中的过滤器Filter，用于对处理器进行预处理和后处理。

区别： 拦截器不需要依赖servlet容器，过滤器需要依赖于javaWeb的环境

拦截器是spring提供的一种机制，过滤器是javaWeb提供的一种支持

### 拦截器定义

实现HandlerInterceptor接口，如下：

**Public class** HandlerInterceptor1 **implements** HandlerInterceptor{

/\*\*

\* controller执行前调用此方法

\* 返回true表示继续执行，返回false中止执行

\* 这里可以加入登录校验、权限拦截等

\*/

@Override

**Public boolean** preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) **throws** Exception {

// **TODO** Auto-generated method stub

**Return false**;

}

/\*\*

\* controller执行后但未返回视图前调用此方法

\* 这里可在返回用户前对模型数据进行加工处理，比如这里加入公用信息以便页面显示

\*/

@Override

**Public void** postHandle(HttpServletRequest request,

HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

// **TODO** Auto-generated method stub

}

/\*\*

\* controller执行后且视图返回后调用此方法

\* 这里可得到执行controller时的异常信息

\* 这里可记录操作日志，资源清理等

\*/

@Override

**Public void** afterCompletion(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

// **TODO** Auto-generated method stub

}

}

### 拦截器配置

#### 针对某种mapping配置拦截器

<bean

class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"*>

<property name=*"interceptors"*>

<list>

<ref bean=*"handlerInterceptor1"*/>

<ref bean=*"handlerInterceptor2"*/>

</list>

</property>

</bean>

<bean id=*"handlerInterceptor1"* class=*"springmvc.intercapter.HandlerInterceptor1"*/>

<bean id=*"handlerInterceptor2"* class=*"springmvc.intercapter.HandlerInterceptor2"*/>

#### 针对所有mapping配置全局拦截器

<!--拦截器 -->

<mvc:interceptors>

<!--多个拦截器,顺序执行 -->

<mvc:interceptor>

<mvc:mapping path=*"/\*\*"*/>

<bean class=*"cn.itcast.springmvc.filter.HandlerInterceptor1"*></bean>

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path=*"/\*\*"*/>

<bean class=*"cn.itcast.springmvc.filter.HandlerInterceptor2"*></bean>

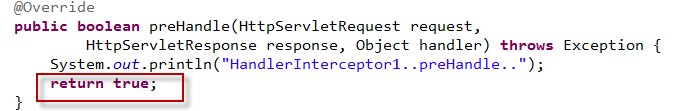
</mvc:interceptor>

</mvc:interceptors>

### 正常流程测试

#### 代码：

定义两个拦截器分别为：HandlerInterceptor1和HandlerInteptor2，每个拦截器的preHandler方法都返回true。



#### 运行流程

HandlerInterceptor1..preHandle..

HandlerInterceptor2..preHandle..

HandlerInterceptor2..postHandle..

HandlerInterceptor1..postHandle..

HandlerInterceptor2..afterCompletion..

HandlerInterceptor1..afterCompletion..

### 中断流程测试

#### 代码：

定义两个拦截器分别为：HandlerInterceptor1和HandlerInteptor2。

#### 运行流程

HandlerInterceptor1的preHandler方法返回false，HandlerInterceptor2返回true，运行流程如下：

HandlerInterceptor1..preHandle..

从日志看出第一个拦截器的preHandler方法返回false后第一个拦截器只执行了preHandler方法，其它两个方法没有执行，第二个拦截器的所有方法不执行，且controller也不执行了。

HandlerInterceptor1的preHandler方法返回true，HandlerInterceptor2返回false，运行流程如下：

HandlerInterceptor1..preHandle..

HandlerInterceptor2..preHandle..

HandlerInterceptor1..afterCompletion..

从日志看出第二个拦截器的preHandler方法返回false后第一个拦截器的postHandler没有执行，第二个拦截器的postHandler和afterCompletion没有执行，且controller也不执行了。

总结：

preHandle按拦截器定义顺序调用

postHandler按拦截器定义逆序调用

afterCompletion按拦截器定义逆序调用

postHandler在拦截器链内所有拦截器返成功调用

afterCompletion只有preHandle返回true才调用

### 拦截器应用

#### 处理流程

1. 有一个登录页面，需要写一个controller访问页面
2. 登录页面有一提交表单的动作。需要在controller中处理。
   1. 判断用户名密码是否正确
   2. 如果正确 向session中写入用户信息
   3. 返回登录成功，或者跳转到商品列表
3. 拦截器。
   1. 拦截用户请求，判断用户是否登录
   2. 如果用户已经登录。放行
   3. 如果用户未登录，跳转到登录页面。

#### 用户身份认证

**Public class** LoginInterceptor **implements** HandlerInterceptor{

@Override

**Public boolean** preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) **throws** Exception {

//如果是登录页面则放行

**if**(request.getRequestURI().indexOf("login.action")>=0){

**return true**;

}

HttpSession session = request.getSession();

//如果用户已登录也放行

**if**(session.getAttribute("user")!=**null**){

**return true**;

}

//用户没有登录挑战到登录页面

request.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward(request, response);

**return false**;

}

}

#### 用户登陆controller

//登陆页面

@RequestMapping("/login")

**public** String login(Model model)**throws** Exception{

**return** "login";

}

//登陆提交

//userid：用户账号，pwd：密码

@RequestMapping("/loginsubmit")

**public** String loginsubmit(HttpSession session,String userid,String pwd)**throws** Exception{

//向session记录用户身份信息

session.setAttribute("activeUser", userid);

**return** "redirect:item/queryItem.action";

}

//退出

@RequestMapping("/logout")

**public** String logout(HttpSession session)**throws** Exception{

//session过期

session.invalidate();

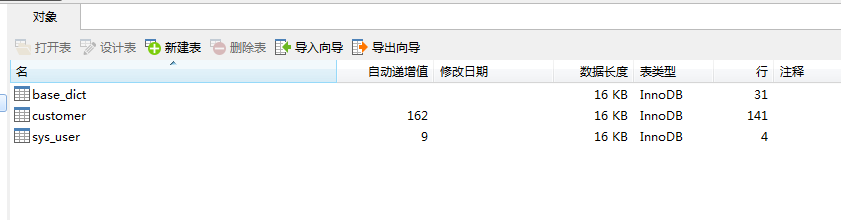
**return** "redirect:item/queryItem.action";

}

# 第三天：综合练习

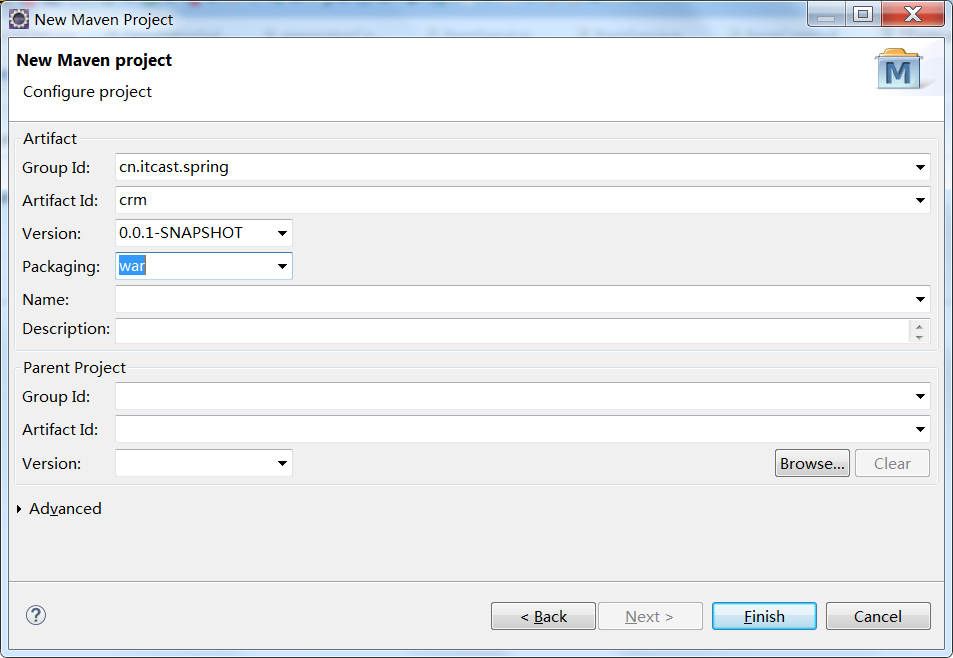
## 第一步：创建数据库

数据库使用mysql 数据库。



1. 创建crm数据库
2. 将参考资料中的sql脚本导入到数据库中

## 第二步：创建maven工程



### 解决web.xml丢失问题

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

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns=*"http://java.sun.com/xml/ns/javaee"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*

version=*"2.5"*>

<display-name>01\_helloWorld</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

</web-app>

### 使用插件解决jdk版本过低问题

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

## 第三步：导入必须的jar包

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-expression</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>aopalliance</groupId>

<artifactId>aopalliance</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.6.8</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>4.2.4.RELEASE</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

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

<version>5.1.38</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.2.2</version>

</dependency>

<dependency>

<groupId>commons-pool</groupId>

<artifactId>commons-pool</artifactId>

<version>1.5.3</version>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.0.18</version>

</dependency>

<dependency>

<groupId>c3p0</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.1.2</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>2.2.2</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.2.7</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-spring</artifactId>

<version>1.2.2</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.5</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.5</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>4.2.4.RELEASE</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>3.0-alpha-1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.1</version>

<scope>provided</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/commons-io/commons-io -->

<dependency>

<groupId>commons-io</groupId>

<artifactId>commons-io</artifactId>

<version>2.4</version>

</dependency>

<!-- https://mvnrepository.com/artifact/commons-fileupload/commons-fileupload -->

<dependency>

<groupId>commons-fileupload</groupId>

<artifactId>commons-fileupload</artifactId>

<version>1.3.1</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core -->

<dependency>

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

<artifactId>jackson-core</artifactId>

<version>2.8.8</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->

<dependency>

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

<artifactId>jackson-databind</artifactId>

<version>2.8.8</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-annotations -->

<dependency>

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

<artifactId>jackson-annotations</artifactId>

<version>2.8.8</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.commons/commons-lang3 -->

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.6</version>

</dependency>

</dependencies>

## 第四步：开发配置文件

### 1：mybatis的配置文件SqlMapConfig.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">

<configuration>

<settings>

<!-- 打印查询语句 -->

<setting name="logImpl" value="STDOUT\_LOGGING" />

</settings>

</configuration>

### 2：spring的配置文件applicationContext.xml

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

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

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

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"*

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

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

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

xsi:schemaLocation=*"*

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

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

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

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

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

<!-- 包扫描的配置 -->

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

<!-- 数据库连接池 -->

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

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

<property name=*"driverClassName"* value=*"${jdbc.driver}"*></property>

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

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

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

</bean>

<!-- sqlSessionFactoryBean -->

<bean class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<property name=*"configLocation"* value=*"classpath:mybatis/SqlMapConfig.xml"*></property>

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

</bean>

<!-- 设置我们的mapper文件的扫描路径 -->

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

<property name=*"basePackage"* value=*"cn.itcast.springmvc.mapper"*></property>

</bean>

<!-- 事物管理配置 -->

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

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

</bean>

<!-- 试用注解的方式来开启事物 -->

<tx:annotation-driven/>

</beans>

### 3：springMVC的配置

<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"

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

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

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

xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"

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

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

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-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/task

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

http://code.alibabatech.com/schema/dubbo

http://code.alibabatech.com/schema/dubbo/dubbo.xsd">

<!-- 加载属性文件 -->

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

<!-- 配置扫描 器 -->

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

<!-- 配置处理器映射器 适配器 -->

<mvc:annotation-driven/>

<!-- 配置视图解释器 jsp -->

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

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

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

</bean>

</beans>

配置resource.properties

customer.source.code=002

customer.industory.code=001

customer.level.code=006

配置db.properties

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/crmsys?characterEncoding=utf-8

jdbc.username=root

jdbc.password=admin

### 4:web.xml配置

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

<web-app version=*"2.5"* xmlns=*"http://java.sun.com/xml/ns/javaee"*

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

xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee*

*http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*>

<welcome-file-list>

<welcome-file>customer.action</welcome-file>

</welcome-file-list>

<!-- 上下文的位置 -->

<context-param>

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

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

</context-param>

<!-- Spring的监听器 -->

<listener>

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

</listener>

<!-- POST提交过滤器 UTF-8 -->

<filter>

<filter-name>encoding</filter-name>

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

<init-param>

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

<param-value>UTF-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encoding</filter-name>

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

</filter-mapping>

<!-- 前端控制器 -->

<servlet>

<servlet-name>crm</servlet-name>

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

<init-param>

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

<!-- 此处不配置 默认找 /WEB-INF/[servlet-name]-servlet.xml -->

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

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>crm</servlet-name>

<!-- 1:\*.do \*.action 拦截以.do结尾的请求 (不拦截 jsp png jpg .js .css) 2:/ 拦截所有请求

（不拦截.jsp) 建议使用此种 方式 （拦截 .js.css .png) (放行静态资源） 3:/\* 拦截所有请求（包括.jsp) 此种方式 不建议使用 -->

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

</servlet-mapping>

</web-app>

5．添加log4j.properties

log4j.rootLogger=INFO,A

log4j.appender.A=org.apache.log4j.ConsoleAppender

log4j.appender.A.Target=System.out

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

log4j.appender.A.layout.ConversionPattern=[%p][%d{yyyy-MM-dd HH\:mm\:ss,SSS}][%c]%m%n

#log4j.rootLogger=debug , stdout , D , E

log4j.appender.B=org.apache.log4j.RollingFileAppender

log4j.appender.B.File=project-jmall-log.log

log4j.appender.B.MaxFileSize=10MB

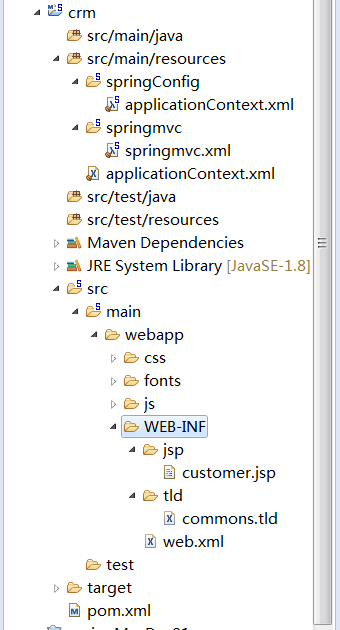
log4j.appender.B.Threshold=ALL

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

log4j.appender.B.layout.ConversionPattern=[%p][%d{yyyy-MM-dd HH\:mm\:ss,SSS}][%c]%m%n

log4j.logger.cn.itcast.springmvc.mapper=DEBUGER

## 第五步：加入jsp以及分页标签



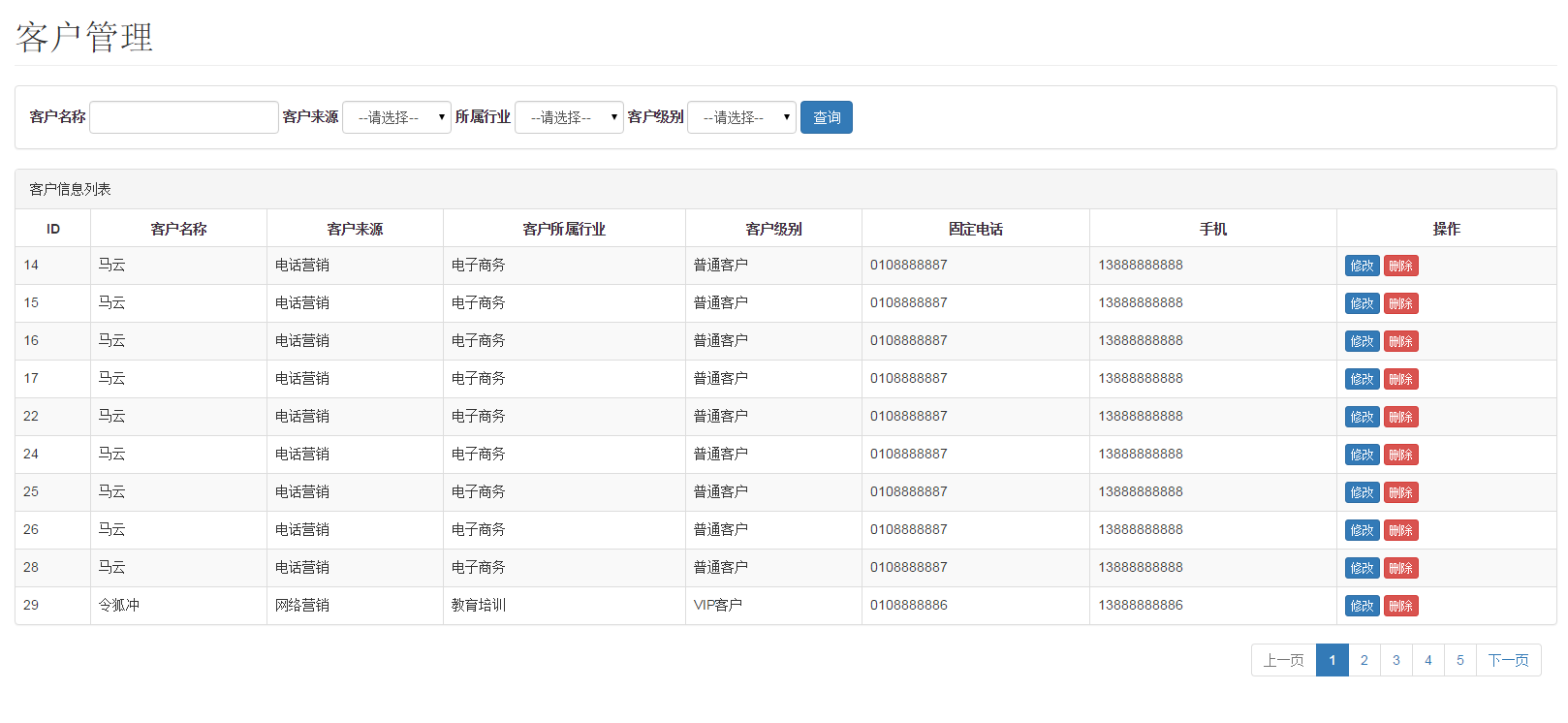
## 第六步：查询条件初始化

### 需求：初始化下拉框查询条件



## 第七步：客户列表展示

### 需求：实现客户列表展示，并且带分页



展示商品列表，并且可以根据查询条件过滤查询结果，并且实现分页处理。

### 第一步：定义pojo

POJO定义

**public** **class** QueryVo {

**private** String custName;

**private** String custSource;

**private** String custIndustry;

**private** String custLevel;

//当前页

**private** Integer page = 1;

//起始行

**private** Integer start;

//每页记录数

**private** Integer size =10 ;

**public** Integer getStart() {

**return** start;

}

**public** **void** setStart(Integer start) {

**this**.start = start;

}

**public** Integer getPage() {

**return** page;

}

**public** **void** setPage(Integer page) {

**this**.page = page;

}

**public** Integer getSize() {

**return** size;

}

**public** **void** setSize(Integer size) {

**this**.size = size;

}

**public** String getCustName() {

**return** custName;

}

**public** **void** setCustName(String custName) {

**this**.custName = custName;

}

**public** String getCustSource() {

**return** custSource;

}

**public** **void** setCustSource(String custSource) {

**this**.custSource = custSource;

}

**public** String getCustIndustry() {

**return** custIndustry;

}

**public** **void** setCustIndustry(String custIndustry) {

**this**.custIndustry = custIndustry;

}

**public** String getCustLevel() {

**return** custLevel;

}

**public** **void** setCustLevel(String custLevel) {

**this**.custLevel = custLevel;

}

}

### 第二步：customerController定义

CustomerController定义：

//客户来源

@Value("${customer\_source}")

private String customerSource;

//客户行业

@Value("${customer\_industry}")

private String customer\_industry;

//客户级别

@Value("${customer\_level}")

private String customer\_level;

@Autowired

private BaseDictService baseDictService;

@Autowired

private CustomerService customerService;

@RequestMapping("/customer/list.action")

**public** String getCustomerList(Model model,QueryVo queryVo) **throws** UnsupportedEncodingException{

**if**(StringUtils.*isNotBlank*(queryVo.getCustName())){

String custName = **new** String(queryVo.getCustName().getBytes("iso-8859-1"), "utf-8");

queryVo.setCustName(custName);

}

List<BaseDict> sourceCodeList = baseDictService.getBaseDictByCode(sourceCode);

List<BaseDict> industryList = baseDictService.getBaseDictByCode(industry);

List<BaseDict> levelList = baseDictService.getBaseDictByCode(level);

model.addAttribute("fromType", sourceCodeList);

model.addAttribute("industryType", industryList);

model.addAttribute("levelType", levelList);

Page<Customer> customerList = customerService.getCustomerList(queryVo);

model.addAttribute("page", customerList);

System.***out***.println(sourceCodeList.size());

model.addAttribute("custName", queryVo.getCustName());

model.addAttribute("custSource", queryVo.getCustSource());

model.addAttribute("custIndustry", queryVo.getCustIndustry());

model.addAttribute("custLevel", queryVo.getCustLevel());

**return** "customer";

}

### 第三步：customerService定义

customerService定义

@Service

public class CustomerServiceImpl implements CustomerService {

@Autowired

private CustomerDao customerDao;

@Override

public Page<Customer> getCustomerList(QueryVo queryVo) {

Page<Customer> page = new Page<Customer>();

if(queryVo.getPage() != null){

queryVo.setStart((queryVo.getPage()-1)\*queryVo.getSize());

}

List<Customer> customerList = customerDao.getCustomerList(queryVo);

page.setRows(customerList);

int totalRecord = customerDao.getCustomerTotal(queryVo);

page.setTotal(totalRecord);

page.setSize(queryVo.getSize());

page.setPage(queryVo.getPage());

return page;

}

}

### 第四步：sql语句定义

Sql语句定义：

<select id=*"getCustomerList"* parameterType=*"cn.itcast.springmvc.pojo.QueryVo"* resultType=*"cn.itcast.springmvc.pojo.Customer"*>

select a.cust\_id,a.cust\_name ,a.cust\_user\_id ,a.cust\_create\_id ,b.dict\_item\_name as cust\_source,c.dict\_item\_name as cust\_industry,d.dict\_item\_name as cust\_level ,a.cust\_linkman,a.cust\_phone,a.cust\_mobile,a.cust\_zipcode,a.cust\_address,a.cust\_createtime

from customer a

left join base\_dict b

on a.cust\_source = b.dict\_id

left join base\_dict c

on a.cust\_industry = c.dict\_id

left join base\_dict d on a.cust\_level = d.dict\_id

<where>

<if test=*"custName !=null and custName != '' "*>

and cust\_name like '%${custName}%'

</if>

<if test=*"custSource !=null and custSource != '' "*>

and cust\_source = #{custSource}

</if>

<if test=*"custIndustry !=null and custIndustry != '' "*>

and cust\_industry = #{custIndustry}

</if>

<if test=*"custLevel !=null and custLevel != '' "*>

and cust\_level = #{custLevel}

</if>

</where>

<if test=*"start !=null"*>

limit #{start},#{size}

</if>

</select>

<select id=*"getCustomerTotal"* parameterType=*"cn.itcast.springmvc.pojo.QueryVo"* resultType=*"int"*>

select count(1)

from customer a

left join base\_dict b

on a.cust\_source = b.dict\_id

left join base\_dict c

on a.cust\_industry = c.dict\_id

left join base\_dict d on a.cust\_level = d.dict\_id

<where>

<if test=*"custName !=null and custName != '' "*>

and cust\_name like '%${custName}%'

</if>

<if test=*"custSource !=null and custSource != '' "*>

and cust\_source = #{custSource}

</if>

<if test=*"custIndustry !=null and custIndustry != '' "*>

and cust\_industry = #{custIndustry}

</if>

<if test=*"custLevel !=null and custLevel != '' "*>

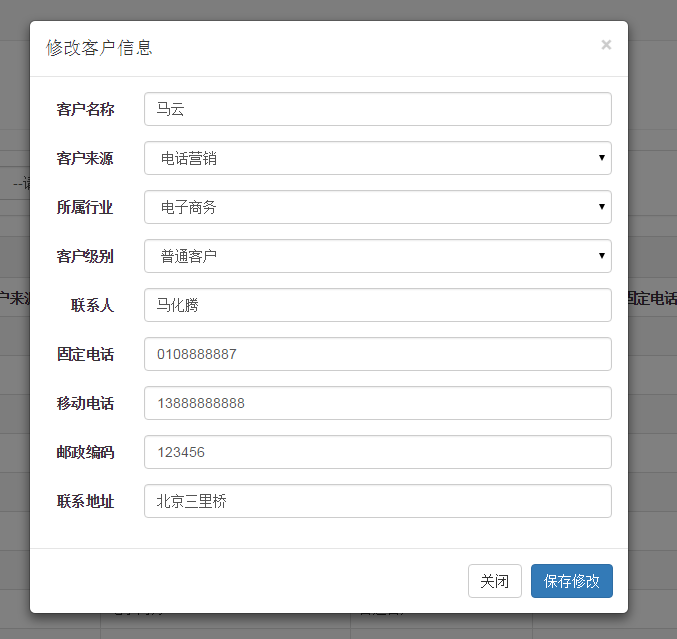
and cust\_level = #{custLevel}

</if>

</where>

</select>

## 第八步：修改客户信息



1. 点击客户列表中的“修改”按钮弹出客户信息修改对话框，并初始化客户信息

第一步：controll层定义

@RequestMapping("/customer/edit.action")

@ResponseBody

**public** Customer getCustomerById(Long id){

Customer customerById = customerService.getCustomerById(id);

**return** customerById;

}

第二步：service层定义

@Override

**public** Customer getCustomerById(**long** custId) {

Customer customer = customerDao.getCustomerById(custId);

**return** customer;

}

第三步：sql语句定义

<select id="getCustomerById" resultType="cn.itcast.springmvc.pojo.Customer" parameterType="long">

select \* from customer where cust\_id = #{custId}

</select>

1. 点击“保存修改”按钮将修改后的结果保存到数据库中

第一步：controller中的方法定义

@RequestMapping("/customer/update.action")

@ResponseBody

**public** String updateCustomer(Customer customer){

customerService.updateCustomer(customer);

**return** "success";

}

第二步：service中的方法定义

@Override

**public** **void** updateCustomer(Customer customer) {

customerDao.updateCustomer(customer);

}

第三步:sql语句定义

<update id=*"updateCustomer"* parameterType=*"cn.itcast.springmvc.pojo.Customer"* >

update customer

<set>

<if test=*"cust\_name !=null and cust\_name !='' "*>

cust\_name = #{cust\_name},

</if>

<if test=*"cust\_user\_id !=null and cust\_user\_id !='' "*>

cust\_user\_id = #{cust\_user\_id},

</if>

<if test=*"cust\_create\_id !=null and cust\_create\_id !='' "*>

cust\_create\_id = #{cust\_create\_id},

</if>

<if test=*"cust\_source !=null and cust\_source !='' "*>

cust\_source = #{cust\_source},

</if>

<if test=*"cust\_industry !=null and cust\_industry !='' "*>

cust\_industry = #{cust\_industry},

</if>

<if test=*"cust\_level !=null and cust\_level !='' "*>

cust\_level = #{cust\_level},

</if>

<if test=*"cust\_linkman !=null and cust\_linkman !='' "*>

cust\_linkman = #{cust\_linkman},

</if>

<if test=*"cust\_phone !=null and cust\_phone !='' "*>

cust\_phone = #{cust\_phone},

</if>

<if test=*"cust\_mobile !=null and cust\_mobile !='' "*>

cust\_mobile = #{cust\_mobile},

</if>

<if test=*"cust\_zipcode !=null and cust\_zipcode !='' "*>

cust\_zipcode = #{cust\_zipcode},

</if>

<if test=*"cust\_address !=null and cust\_address !='' "*>

cust\_address = #{cust\_address},

</if>

<if test=*"cust\_zipcode !=null and cust\_zipcode !='' "*>

cust\_zipcode = #{cust\_zipcode},

</if>

<if test=*"cust\_createtime !=null and cust\_createtime !='' "*>

cust\_createtime = #{cust\_createtime},

</if>

</set>

<where>

<if test=*"cust\_id !=null and cust\_id != '' "*>

and cust\_id = #{cust\_id}

</if>

</where>

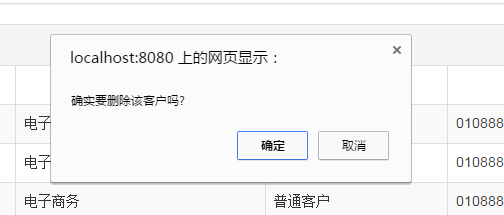
</update>

## 第九步：删除客户

### 需求



点击客户列表中的删除按钮，提示“警告信息”



点击确定后删除用户信息，并刷新页面。