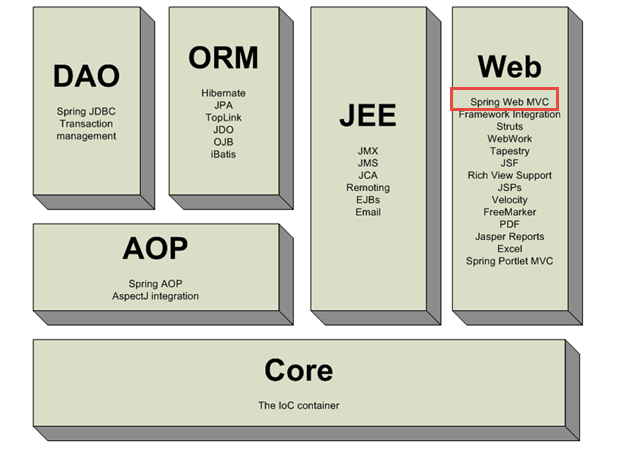
# SpringMVC

## springmvc框架



### MVC在B/S系统下的应用

C

request

M

请求模型进行处理

返回处理结果

V

视图渲染，将模型数据填充request域

response

B/S系统下，模型无法将数据填充到视图

### SpringMVC框架

SpringMVC是Spring框架的一个模块，是一个基于MVC的Web框架，其与Spring整合无需中间整合层，整合jar包，

Handler处理器（controller）

处理器映射器HandlerMapping

返回执行链

HandlerExecutionChain{

HandlerInterceptor1

HandlerInterceptor2

Handler

}

处理器适配器

HandlerAdapter

请求执行Handler

请求执行Handler

返回ModelAndView

视图解析器View resolver

请求进行视图解析

返回view

request

response

前端控制器DispatcherServlet

请求查找Handler

返回ModelAndView

V

视图渲染，将模型数据填充request域

执行流程： 1.发起请求到DispatcherServlet

2. DispatcherServlet请求HandlerMapping通过配置文件或注解方式查找 Handler

3. HandlerMapping向DispatcherServlet返回Handler

4. DispatcherServlet调用HandlerAdapter执行Handler

5. HandlerAdapter调用Handler处理器执行Handler

6.Handler处理器执行完成给HandlerAdapter返回ModelAndView

7. HandlerAdapter向DispatcherServlet返回ModelAndView（springmvc框架的底层对象，包括 Model和view）

8. DispatcherServlet请求View resolver进行视图解析

9. View resolver向DispatcherServlet返回View

10.DispatcherServlet进行视图渲染，将模型数据(在ModelAndView对象中)填充request域

11. DispatcherServlet响应结果

组件： 1.DispatcherServlet：用来接受请求并响应结果，相当于转发器，用来与多个组件交互，减少了其它组件之间的耦合度，相当于中央处理器

2. HandlerMapping：用来根据请求的url查找Handler

3.HandlerAdapter：按照特定要求去执行Handler

4. Handler处理器：按照HandlerAdapter的要求执行Handler(需要程序员开发)

5 View resolver：进行视图解析，根据逻辑视图名解析成真正的视图

6.View：是一个接口，其实现类支持不同的View类型(需要程序员开发jsp)

### 3.源码剖析：

1. 前端控制器接收请求

protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {

···

}

2.调用处理器映射器查找Handler，处理器映射器返回执行链

mappedHandler = ~~getHandler~~(processedRequest, false);

protected HandlerExecutionChain ~~getHandler~~(HttpServletRequest request, boolean cache) throws Exception {

return getHandler(request);

}

protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {

for (HandlerMapping hm : this.handlerMappings) {

if (logger.isTraceEnabled()) {

logger.trace(

"Testing handler map [" + hm + "] in DispatcherServlet with name '" + getServletName() + "'");

}

HandlerExecutionChain handler = hm.getHandler(request);

if (handler != null) {

return handler;

}

}

return null;

}

3. 调用处理器适配器执行Handler，处理器适配器调用Handler处理器执行Handler得到执行结果ModelAndView后依次返回

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

4. 请求视图解析器进行视图解析并返回视图，在进行视图渲染后将模型数据填充request域

processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);

render(mv, request, response);

view = resolveViewName(mv.getViewName(), mv.getModelInternal(), locale, request);

protected View resolveViewName(String viewName, Map<String, Object> model, Locale locale,

HttpServletRequest request) throws Exception {

for (ViewResolver viewResolver : this.viewResolvers) {

View view = viewResolver.resolveViewName(viewName, locale);

if (view != null) {

return view;

}

}

return null;

}

在render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response)方法中得到view之后渲染

view.render(mv.getModelInternal(), request, response);

View是一个接口

public abstract class AbstractView extends WebApplicationObjectSupport implements View, BeanNameAware {

protected void exposeModelAsRequestAttributes(Map<String, Object> model, HttpServletRequest request) throws Exception {

for (Map.Entry<String, Object> entry : model.entrySet()) {

String modelName = entry.getKey();

Object modelValue = entry.getValue();

if (modelValue != null) {

request.setAttribute(modelName, modelValue);

if (logger.isDebugEnabled()) {

logger.debug("Added model object '" + modelName + "' of type [" + modelValue.getClass().getName() +

"] to request in view with name '" + getBeanName() + "'");

}

}

else {

request.removeAttribute(modelName);

if (logger.isDebugEnabled()) {

logger.debug("Removed model object '" + modelName +

"' from request in view with name '" + getBeanName() + "'");

}

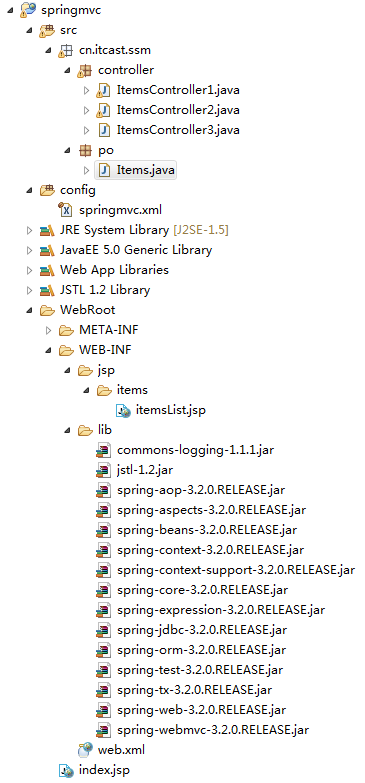
}

}

}

## 入门程序

### 1.导入Spring相关jar包和需要的配置文件，jar包一定要包括spring-webmvc



### 2.配置前端控制器

<!--配置springmvc前端控制器 -->

<servlet>

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

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

<init-param>

<!--contextConfigLocation中配置了springmvc需要加载的配置文件-->

<!--如果不配置contextConfigLocation，默认加载的是/WEB-INF/servlet名称-serlvet.xml-->

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

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

</init-param>

</servlet>

<servlet-mapping>

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

<!--1.\*.action:访问以.action结尾的请求路径由DispatcherServlet进行解析-->

<!--2./:所有访问地址都由DispatcherServlet进行解析，但对静态资源解析存在问题-->

<!--3./\*:最终转发到的jsp页面，也由DispatcherServlet解析，由于不能根据jsp页面找到Handler，会报错-->

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

</servlet-mapping>

### 3.配置springmvc.xml

<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"

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

http://www.springframework.org/schema/beans/spring-beans-3.2.xsd

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

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

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

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

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

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">

<!--配置Handler，name属性值为该Handler的url，注意要与前端控制器的url-pattern对应-->

<bean name="/queryItems.action" class="cn.itcast.ssm.controller.ItemsController1" />

<bean id="itemsController2" name="/queryItems1.action" class="cn.itcast.ssm.controller.ItemsController2" />

<!--配置处理器映射器，该处理器映射器将bean的name属性值作为url，所有的处理器映射器都实现 HandlerMapping接口-->

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />

<!--另一个处理器映射器，该处理器映射器可根据Handler的id来进行url配置，多个处理器映射器可以共存-->

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="mappings">

<props>

<!--对id为itemsController1的Handr进行url映射，url是/queryItems1.action-->

<prop key="/queryItems2.action">itemsController2</prop>

<!--一个Handler可进行多次url映射，现在该Handler有三个url-->

<prop key="/queryItems3.action">itemsController2</prop>

</props>

</property>

</bean>

<!--配置处理器适配器，所有处理器适配器都实现 HandlerAdapter接口，这是处理器适配器的标识，该处理器适配器能执行实现Controller接口的Handler-->

<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />

<!--另一个处理器适配器，该处理器映射器能执行实现HandlerAdapter接口的Handler，多个处理器适配器也可以共存-->

<bean class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"/>

<!--配置视图解析器，解析jsp视图，默认使用jstl标签-->

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

<!--配置jsp路径的前缀和后缀-->

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

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

</bean>

</beans>

### 4.开发Handler

1.实现Controller接口的Handler，由SimpleControllerHandlerAdapter处理器适配器执行

public class ItemsController1 implements Controller {

public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception {

List<Items> itemsList = new ArrayList<Items>();

Items items\_1 = new Items();

items\_1.setName("联想笔记本");

items\_1.setPrice(6000f);

items\_1.setDetail("ThinkPad T430 联想笔记本电脑！");

Items items\_2 = new Items();

items\_2.setName("苹果手机");

items\_2.setPrice(5000f);

items\_2.setDetail("iphone6苹果手机！");

itemsList.add(items\_1);

itemsList.add(items\_2);

//创建ModelAndView

ModelAndView modelAndView = new ModelAndView();

//相当于request.setAttribut("itemsList", itemsList);

modelAndView.addObject("itemsList", itemsList);

//相当于request.getRequestDispatcher("/WEB-INF/jsp/items/itemsList.jsp").forward(request, response);

modelAndView.setViewName("items/itemsList");

//返回ModelAndView

return modelAndView;

}

}

2.实现HttpRequestHandler接口的Handler，由HttpRequestHandlerAdapter处理器适配器执行

public class ItemsController2 implements HttpRequestHandler {

public void handleRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

List<Items> itemsList = new ArrayList<Items>();

Items items\_1 = new Items();

items\_1.setName("联想笔记本");

items\_1.setPrice(6000f);

items\_1.setDetail("ThinkPad T430 联想笔记本电脑！");

Items items\_2 = new Items();

items\_2.setName("苹果手机");

items\_2.setPrice(5000f);

items\_2.setDetail("iphone6苹果手机！");

itemsList.add(items\_1);

itemsList.add(items\_2);

request.setAttribute("itemsList", itemsList);

request.getRequestDispatcher("items/itemsList ").forward(request, response);

}

}

### 5.测试

Jsp页面内容

<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>

</tr>

<c:forEach items="${requestScope.itemsList }" var="item">

<tr>

<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>

</tr>

</c:forEach>

</table>

响应结果

|  |  |  |  |
| --- | --- | --- | --- |
| 商品名称 | 商品价格 | 生产日期 | 商品描述 |
| 联想笔记本 | 6000.0 |  | ThinkPad T430 联想笔记本电脑！ |
| 苹果手机 | 5000.0 |  | iphone6苹果手机！ |

### 6.注意点

DispatcherSerlvet.properties

org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeResolver

org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\

org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\

org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\

org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\

org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\

org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator

org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver

org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFlashMapManager

如果不在springmvc.xml中配置组件，前端控制器会从该文件加载默认组件

## 注解处理器映射器和适配器

在spring3.1之前使用org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping注解映射器。

和org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter注解适配器

但在spring3.1之后需要使用org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping注解映射器

和org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter注解适配器

<!--必须配对使用-->

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

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

//标识该类为一个Handler（现在该类可以没有任何实现）

@Controller

public class ItemsController3 {

//配置url和Handler方法映射（现在一个Handler中可以有多个方法）

@RequestMapping("/queryItems")

public ModelAndView queryItems()throws Exception{

List<Items> itemsList = new ArrayList<Items>();

Items items\_1 = new Items();

items\_1.setName("联想笔记本");

items\_1.setPrice(6000f);

items\_1.setDetail("ThinkPad T430");

Items items\_2 = new Items();

items\_2.setName("苹果手机");

items\_2.setPrice(5000f);

items\_2.setDetail("iphone6苹果手机！");

itemsList.add(items\_1);

itemsList.add(items\_2);

ModelAndView modelAndView = new ModelAndView();

modelAndView.addObject("itemsList", itemsList);

modelAndView.setViewName("items/itemsList ");

return modelAndView;

}

}

<!--现在无需再配置id属性或name属性-->

<bean class="cn.itcast.ssm.controller.ItemsController3"/>

<!--使用mvc:annotation-driven注解驱动可以代替上边的注解处理器映射器适配器的配置，并且还默认加载很多的参数的绑定方法-->

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

## SptingMVC和Mybatis整合

### 1.整合思路

表现层SpringMVC

Spring将各层整合：

管理持久层的mapper，使用mapper扫描器

管理业务层的service以及对其事务控制

管理表现层的handler

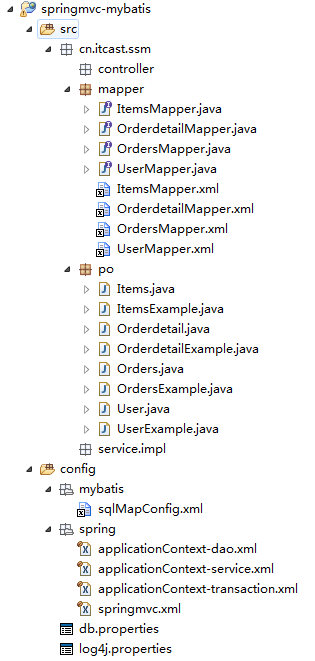
业务层Service接口

持久层Mybatis

MySql

### 2.环境搭建

导入所需jar包，所需配置文件，创建Mybatis、Spring三层、SpringMVC的配置文件，从逆向工程中引入文件，这些文件不要修改



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"

id="WebApp\_ID" version="2.5">

<display-name>springmvc-mybatis</display-name>

<context-param>

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

<param-value>/WEB-INF/classes/spring/applicationContext-\*.xml</param-value>

</context-param>

<listener>

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

</listener>

<servlet>

<servlet-name>springmvc</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-name>

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

</servlet-mapping>

<!--乱码过滤器，其可以处理POST请求乱码问题-->

<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>

<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>

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>

<typeAliases>

<package name="cn.itcast.ssm.po"/>

</typeAliases>

</configuration>

springmvc.xml

<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"

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

http://www.springframework.org/schema/beans/spring-beans-3.2.xsd

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

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

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

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

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

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">

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

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

<bean

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

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

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

</bean>

</beans>

applicationContext-dao.xml

<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"

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

http://www.springframework.org/schema/beans/spring-beans-3.2.xsd

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

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

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

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

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

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">

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

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

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

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

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

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

</bean>

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

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

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

</bean>

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

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

<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />

</bean>

</beans>

applicationContext-service.xml

<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"

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

http://www.springframework.org/schema/beans/spring-beans-3.2.xsd

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

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

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

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

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

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">

</beans>

applicationContext-transaction.xml

<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"

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

http://www.springframework.org/schema/beans/spring-beans-3.2.xsd

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

http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd

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

http://www.springframework.org/schema/context/spring-context-3.2.xsd

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

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd

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

http://www.springframework.org/schema/tx/spring-tx-3.2.xsd ">

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

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

</bean>

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

<tx:attributes>

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

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

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

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

<tx:method name="find\*" propagation="SUPPORTS" read-only="true"/>

<tx:method name="get\*" propagation="SUPPORTS" read-only="true"/>

<tx:method name="select\*" propagation="SUPPORTS" read-only="true"/>

</tx:attributes>

</tx:advice>

<aop:config>

<aop:advisor advice-ref="txAdvice" pointcut="execution(\* cn.itcast.ssm.service.impl.\*.\*(..))"/>

</aop:config>

</beans>

### 3.环境测试

1.定义扩展类

public class ItemsCustom extends Items{}

2.定义包装类

public class ItemsQueryVo {

private Items items;

private ItemsCustom itemsCustom;

public Items getItems() {

return items;

}

public void setItems(Items items) {

this.items = items;

}

public ItemsCustom getItemsCustom() {

return itemsCustom;

}

public void setItemsCustom(ItemsCustom itemsCustom) {

this.itemsCustom = itemsCustom;

}

}

3.配置sql语句

<?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" >

<mapper namespace="cn.itcast.ssm.mapper.ItemsMapperCustom" >

<sql id="query\_items\_where">

<if test="itemsCustom!=null">

<if test="itemsCustom.name!=null and itemsCustom.name!=''">

items.name LIKE '%${itemsCustom.name}%'

</if>

</if>

</sql>

<select id="findItemsList" parameterType="itemsQueryVo" resultType="itemsCustom">

SELECT items.\* FROM items

<where>

<include refid="query\_items\_where"/>

</where>

</select>

</mapper>

4.构造Mapper接口

public interface ItemsMapperCustom {

List<ItemsCustom> findItemsList(ItemsQueryVo itemsQueryVo) throws Exception;

}

5.创建Service

public interface ItemsService {

List<ItemsCustom> findItemsList(ItemsQueryVo itemsQueryVo) throws Exception;

}

6.创建Service实现类

public class ItemsServiceImpl implements ItemsService{

@Autowired

private ItemsMapperCustom itemsMapperCustom;

public List<ItemsCustom> findItemsList(ItemsQueryVo itemsQueryVo) throws Exception {

return itemsMapperCustom.findItemsList(itemsQueryVo);

}

}

7.将Service实现类放入Spring容器

<bean id="itemsService" class="cn.itcast.ssm.service.impl.ItemsServiceImpl"/>

8.开发Controller

@Controller

public class ItemsController {

private ItemsService itemService;

@Autowired

private ItemsService itemsService;

@RequestMapping("/queryItems")

public ModelAndView queryItems() throws Exception {

ModelAndView modelAndView = new ModelAndView();

modelAndView.addObject("itemsList", itemsService.findItemsList(null));

modelAndView.setViewName("items/itemsList");

return modelAndView;

}

}

## RequestMapping的其他两个作用

@Controller

//窄化请求映射，这个url片段相当于写在了每个方法的url片段之前

@RequestMapping("/items")

public class ItemsController {

//限制http请求方式，如果只限制GET，那么其它请求方式会报错，支持多限制

@RequestMapping(value="/editItems",method={RequestMethod.GET,RequestMethod.POST})

## controller方法返回值类型

controller方法的返回值类型有三种：ModelAndView、String、void，返回给HandlerAdapter，其最终会将返回内容封装成一个ModelAndView返回给DispatchServlet

### 1. ModelAndView，需要创建ModelAndView对象，它可以设置模型和视图

### 2. String，返回值指定视图，可以给返回值前缀添加 “redirect:” 或 “forward:” 指定重定向或转发

### 3.void，可以在参数列表中添加response，用来响应数据

## 参数绑定

### ***1.***参数绑定

在SpringMVC中提供了很多参数绑定的组件，可以将页面提交到controller的数据通过方法的参数来接收，不再是以往的注入成员变量值的方式了

在spirngmvc早期版本使用PropertyEditor组件，它只能将字符串转换成pojo，而后期版本中使用的converter组件可以满足任意类型的传换

### 2.参数绑定类型

#### 1.默认支持类型，即直接在controller方法的参数列表中添加即可使用

HttpServletRequest、HttpServletResponse、HttpSession、Model/ModelMap（ModelMap是Model接口的实现类，其都可以将model填充到request域）

#### 2.简单类型，要求形参和实参名称一致，如果不一致，可以@RequestParam注解

public String queryItems(@RequestParam(value="uname",required=true,defaultValue="李四")String username) throws Exception {

value对应实参名称，没有则报错、required指定是否必须存在，不存在则报错，默认为true、defaultValue存在时即使required=true也不报错

#### 3.pojo，需要实参名称和形参的属性名称一致，

#### 自定义类型，通常用来将日期格式字符串转换为日期类型，需要将定义转换器注入到自定义参数绑定组件，再将组件添加到注解驱动中

import org.springframework.core.convert.converter.Converter;

public class CustomDateConverter implements Converter<String, Date>{

public Date convert(String date) {

try {

return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").parse(date);

} catch (ParseException e) {

e.printStackTrace();

return null;

}

}

}

<bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="converters">

<list>

<bean class="cn.itcast.ssm.controller.CustomDateConverter"></bean>

</list>

</property>

</bean>

### 3.多个参数同时满足参数绑定条件时，甚至类型不同，都互不影响

## springmvc和struts2的区别

1、struts2基于类开发，而springmvc基于方法开发

2. struts2只能多例开发，而springmvc可以单例开发，因为springmvc是将url与controller的方法来进行映射，映射成功后会生成一个Handler对象，而对象中只有一个方法，方法执行结束，形参数据即销毁

3.struts2通过类的成员变量接收参数，而springmvc还可通过方法参数接收数据，更方便维护，springmvc的controller开发更类似service开发

3.经过实际测试，struts2速度较慢，是因为其使用了struts标签，所以使用struts建议使用jstl标签

## validation校验框架

在项目中，通常使用较多是前端请求参数的校验，对于安全要求较高点建议在服务端进行校验，其可以对三层进行校验

所需jar包：validation-api、jboss-logging、hibernate-validator

### 1.在spingmvc.xml中配置校验器，并将校验器其注入到处理器适配器中

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

<!--配置校验器-->

<bean id="validator" class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">

<property name="providerClass" value="org.hibernate.validator.HibernateValidator" />

<!--指定校验使用校验错误信息的配置文件，如果不指定则默认使用classpath下的ValidationMessages.properties-->

<property name="validationMessageSource" ref="messageSource" />

</bean>

<!--校验错误信息配置文件-->

<bean id="messageSource"

class="org.springframework.context.support.ReloadableResourceBundleMessageSource">

<!--资源文件名-->

<property name="basenames">

<list>

<value>classpath:CustomValidationMessages</value>

</list>

</property>

<property name="fileEncodings" value="utf-8" />

<!--对资源文件内容缓存时间，单位秒-->

<property name="cacheSeconds" value="120" />

</bean>

### 2. 创建CustomValidationMessages.properties配置文件并配置校验错误信息

items.name.length.error=商品名称字符必须在5~15之间！

items.createtime.isNUll=商品生产日期不能为空！

### 3. 在pojo中添加校验规则

//长度校验，要求名称在5到15字符之间

@Size(min=5,max=15,message="{items.name.length.error}",groups={ValidGroup1.class})

private String name;

//非空校验

@NotNull(message="{items.createtime.isNUll}")

private Date createtime;

### 4. 捕获校验错误信息

// 在需要校验的参数的前面添加@Validated，后边添加BindingResult参数

public ModelAndView editItemsSubmit(@Validated ItemsCustom itemsCustom,BindingResult bindingResult) throws Exception {

if (bindingResult.hasErrors()) {

//该方法返回List<ObjectError>

model.addAttribute("allErrors", bindingResult.getAllErrors());

model.addAttribute("items", itemsCustom);

return "items/editItems";

}

### 5.分组校验

不同时controller方法对同一个pojo进行校验时，要求可能是不同的，这时就可以使用分组校验将校验条件分隔开来

#### 1.定义不同的校验分组，它是一个接口，无需任何内容

package cn.itcast.ssm.controller.validation;

public interface ValidGroup1 {}

#### 2.在校验规则中添加分组

@Size(min=5,max=15,message="{items.name.length.error}",groups={ValidGroup1. class})

#### 3. 在controller方法的参数列表中为参数指定校验分组

@Validated(value = { ValidGroup1.class })

## SpringMVC的数据回显

SpringMVC默认会把pojo形参放到request域中，域数据的名称默认为pojo首字母小写，也可以使用@ModelAttribute注解指定名称

(@ModelAttribute("items") @Validated(value = { ValidGroup1.class }) ItemsCustom itemsCustom,BindingResult bindingResult)

该注解还可以用在自定义的方法上，就能够该方法的返回值放到request域中，需要指定域数据名称（该方法只是一个普通方法）

@ModelAttribute("types")

public Map<String, String> getItemTypes() {

Map<String, String> itemTypes = new HashMap<String, String>();

itemTypes.put("101", "数码");

itemTypes.put("102", "母婴");

return itemTypes;

}

## 异常处理

### 1.异常处理思路

系统中异常分为两类：预期异常和运行时异常，前者可通过捕获异常获取信息，后者则需要规范代码开发、进行运行测试等手段减少其出现几率

在系统的三层主要出现异常，都需要上抛，最后通过SpringMVC前端控制器交给全局异常处理器处理

### 2.全局异常处理器

#### 1.需要先自定义一个异常类

2.定义全局异常处理器

package cn.itcast.ssm.exception;

//要求该类实现HandlerExceptionResolver

public class CustomExceptionResolver implements HandlerExceptionResolver {

@Override

//在实现方法中需要判断异常是否为自定义异常，Object类型参数就是处理器适配器要执行Handler对象，它只有一个方法

public ModelAndView resolveException(HttpServletRequest request,HttpServletResponse response, Object handler, Exception ex) {

CustomException customException = null;

if(ex instanceof CustomException){

customException = (CustomException)ex;

}else{

customException = new CustomException("未知错误");

}

String message = customException.getMessage();

ModelAndView modelAndView = new ModelAndView();

modelAndView.addObject("message", message);

modelAndView.setViewName("error");

return modelAndView;

}

}

3. 在springmvc.xml中配置全局异常处理器

<!--只要实现HandlerExceptionResolver接口就是全局异常处理器，全局异常处理器一个系统只能有一个-->

<bean class="cn.itcast.ssm.exception.CustomExceptionResolver"></bean>

## SptingMVC上传

1.需要在springmvc.xml中配置multipart类型解析器（对于类型为multipart/form-data的表单，不配置会导致参数绑定失败）

<bean id="multipartResolver"

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

<property name="maxUploadSize">

<value>5242880</value>

</property>

</bean>

2.创建文件虚拟目录存储图片，并将其在Tomcat的server.xml中配置，这时通过虚拟目录的url路径即可访问到其内容：http://localhost:8080/pic/133.jpg



3.导入jar包：commons-fileupload、commons-io-1.4

4.测试代码

@RequestMapping("/editItemsSubmit")

public String editItemsSubmit(Model model,ItemsCustom itemsCustom,MultipartFile items\_pic) throws Exception {

if(items\_pic != null){

String originalFilename = items\_pic.getOriginalFilename();

if(originalFilename != null && originalFilename.length() > 0){

String pic\_path = "F:/temp";

String newFileName = UUID.randomUUID() + originalFilename.substring(originalFilename.lastIndexOf("."));

File newFile = new File(pic\_path,newFileName);

items\_pic.transferTo(newFile);

itemsCustom.setPic(newFileName);

}

}

itemsService.updateItems(itemsCustom.getId(), itemsCustom);

return queryItems(model);

}

## json数据交互

springmvc进行json交互有两种方式，但两种最终都需要使用@ResponseBody注解将java对象转成json串输出

1.请求为json串，需要指定contentType= application/json，需要使用@RequestBody注解将json串转为java对象

2.请求为key/value，需要指定contentType=application/x-www-form，不需要转换

### 1.导入jar包：jackson-core-asl、jackson-mapper-asl

### 2. 在注解处理器适配器中配置json转换器，如果使用的是注解驱动，则会自动配置

<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>

### 3.测试

function keyValue(){

$.ajax({

type:'post',

url:'${pageContext.request.contextPath }/keyVale.action',

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

data:'{"name":"手机","price":999}',

success:function(data){

alert(data.name);

}

});

}

function json(){

$.ajax({

type:'post',

url:'${pageContext.request.contextPath }/json.action',

//定contentType默认为是key/value类型，不需要指定

data:'name=手机&price=999',

success:function(data){

alert(data.name);

}

});

}

</script>

<body>

<input type="button" onclick="keyValue()" value="请求json，输出是json" />

<input type="button" onclick="json()" value="请求key/value，输出是json" />

public class JsonTest {

@RequestMapping("/keyValue")

public @ResponseBody ItemsCustom keyValue(@RequestBody ItemsCustom itemsCustom){

return itemsCustom;

}

@RequestMapping("/json")

public @ResponseBody ItemsCustom json(ItemsCustom itemsCustom){

return itemsCustom;

}

}

## RESTful支持

#### 1.RESTful（Representational State Transfer）是目前最流行的一种互联网软件架构，是一个开发理念，对http有很好的诠释，其结构清晰、符合标准、易于理解、扩展方便

RESTful的特点：1.对url进行了规范：<http://...../queryItems.action?id=001> –> http://.... /queryItems.action /001

2.对http方法进行了规范，不管是添加、更新、删除···使用的url都是一致的，但如果需要删除，需要设置http的方法为DELETE，controller则需要对 http方法进行判断，然后执行对应的方法

3 对contentType进行了规范，请求时指定contentType，可以根据需要类型得到返回数据

#### 2.实现RESTful的url规范

<!-- springmvc前端控制器，rest配置 -->

<servlet>

<servlet-name>springmvc\_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\_rest</servlet-name>

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

</servlet-mapping>

//通过@PathVariable注解获取占位符中的参数，如果占位符中的名称和形参名一致，可以不指定value，最终输出json数据

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

public @ResponseBody ItemsCustom itemsView(@PathVariable("id") Integer id)throws Exception{

return itemsService.findItemsById(id);

}

根据前面所提到的，如果url-patten使用 “/” ，则对静态数据解析存在问题，这时需要在springmvc中配置静态数据的解析方法

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

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

## SpringMVC拦截器

1.拦截器的定义

//springmvc定义拦截器需要实现HandlerInterceptor接口

public class HandlerInterceptor3 implements HandlerInterceptor {

@Override

//在Handler方法之前执行，return false表示拦截、return true表示放行，多用来进行身份校验

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

return false;

}

@Override

//在Handler方法开始执行执行之后，返回ModelAndView之前执行，多在这里将共用的模型数据传到视图，也可以在这统一指定视图

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView mv)throws Exception {}

@Override

//在Handler方法之后执行，多用来统一异常处理、统一日志处理

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)throws Exception {}

}

2.拦截器的配置

SpringMVC拦截器不是全局拦截器，是针对HandlerMapper进行拦截配置，结果该HandlerMapper映射成功的Handler才能使用拦截器

<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" />

还可以配置类似全局拦截器的拦截器，springmvc会将该拦截器注入到每个HandlerMapping中

<mvc:interceptors>

<mvc:interceptor>

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

<bean class="cn.itcast.ssm.interceptor.LoginInterceptor"></bean>

</mvc:interceptor>

<mvc:interceptor>

<!--/\*\*表示所有url，包括子url路径，而/\*只表示下一层-->

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

<bean class="cn.itcast.ssm.interceptor.HandlerInterceptor1"></bean>

</mvc:interceptor>

<mvc:interceptor>

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

<bean class="cn.itcast.ssm.interceptor.HandlerInterceptor2"></bean>

</mvc:interceptor>

</mvc:interceptors>

多拦截器执行流程：

1. 两个拦截器都放行

HandlerInterceptor1...preHandle

HandlerInterceptor2...preHandle

HandlerInterceptor2...postHandle

HandlerInterceptor1...postHandle

HandlerInterceptor2...afterCompletion

HandlerInterceptor1...afterCompletion

2.第一个放行，第二个不放行

HandlerInterceptor1...preHandle

HandlerInterceptor2...preHandle

HandlerInterceptor1...afterCompletion

3.都不放行

HandlerInterceptor1...preHandle

总结：1.preHandle方法按拦截器配置顺序执行，其余两个方法与之相反

2.只要有一个拦截器不放行，postHandle不会执行

3.哪个拦截器不放行，哪个拦截器的afterCompletion也不会执行

对于不同的url是否需要放行，一般写一个配置文件来配置所有公开url，在拦截器的preHandle方法中进行判断即可

public ItemsCustom findItemsById(Integer id) throws Exception {

Items items = itemsMapper.selectByPrimaryKey(id);

//业务操作····

//属性拷贝

ItemsCustom itemsCustom = new ItemsCustom();

BeanUtils.copyProperties(items, itemsCustom);

return itemsCustom;

}

public void updateItems(Integer id, ItemsCustom itemsCustom) throws Exception {

//业务操作···

itemsCustom.setId(id);

//该方法按实参的id进行数据更新，更新所有数据，包括大文本类型字段

itemsMapper.updateByPrimaryKeyWithBLOBs(itemsCustom);

}