springmvc第一天 springmvc的基础知识

springmvc课程安排：

第一天： 基础知识

1. springmvc框架(重点)

mvc在b/s系统中应用方式

springmvc框架原理（DispatcherServlet前端控制器、处理器映射器、处理器适配器、视图解析器）

2. springmvc入门程序

需求：商品列表查询

常用的处理器映射器，处理器适配器

注解的处理器映射器，处理器适配器用于注解开发(重点)

3. 注解开发基础

常用的注解使用

参数绑定

自定义参数绑定

4. springmvc和struts的区别

第二天： 高级知识

使用springmvc上传图片

集合参数的绑定

validation校验器

异常处理器使用（用于系统异常处理）

RESTful支持

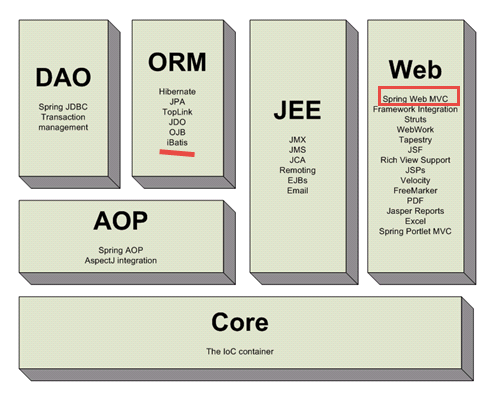
拦截器

学生练习

# springmvc框架

## springmvc是什么？

springmvc是spring的一个模块，提供web层解决方案（基于mvc设计架构）



## mvc在b/s系统的应用

mvc是一个设计模式，在b/s系统的应用：

request

**C （controller）控制器**

**作用：接收用户请求**

**及响应结果**

response

**M （model）**

**模型**

**pojo**

**action**

**service**

**dao**

请求模型进行处理

返回处理结果

**V (view)视图**

填充模型数据、渲染视图

注意：b/s系统模型无法将数据推送到视图

## springmvc的框架

DispatcherServlet

前端控制器

1 request请求

url

Handler处理器

HandlerMapping处理器映射器

2请求查找Handler

3 HandlerExecutionChain{

HandlerInterceptor

HandlerInterceptor

Handler

}

HandlerAdapter处理器适配器

4请求执行Handler

5执行

6ModelAndView

7向前端控制器返回

ModelAndView

ViewResolver视图解析器

8请求进行视图解析

9返回view

View视图

10渲染视图

向view填充模型数据

11 response

第一步：用户发起request请求，请求至DispatcherServlet前端控制器

第二步：DispatcherServlet前端控制器请求HandlerMapping处理器映射器查找Handler

DispatcherServlet：前端控制器，相当于中央调度器，各各组件都和前端控制器进行交互，降低了各各组 件之间耦合度。

第三步：HandlerMapping处理器映射器，根据url及一些配置规则（xml配置、注解配置）查找Handler，将Handler返回给DispatcherServlet前端控制器

第四步：DispatcherServlet前端控制器调用适配器执行Handler，有了适配器通过适配器去扩展对不同Handler执行方式（比如：原始servlet开发，注解开发）

第五步：适配器执行Handler

Handler是后端控制器，当成模型。

第六步：Handler执行完成返回ModelAndView

ModelAndView：springmvc的一个对象，对Model和view进行封装。

第七步：适配器将ModelAndView返回给DispatcherServlet

第八步：DispatcherServlet调用视图解析器进行视图解析，解析后生成view

视图解析器根据逻辑视图名解析出真正的视图。

View：springmvc视图封装对象，提供了很多view，jsp、freemarker、pdf、excel。。。

第九步：ViewResolver视图解析器给前端控制器返回view

第十步：DispatcherServlet调用view的渲染视图的方法，将模型数据填充到request域 。

第十一步：DispatcherServlet向用户响应结果(jsp页面、json数据。。。。)

DispatcherServlet：前端控制器，由springmvc提供

HandlerMappting：处理器映射器，由springmvc提供

HandlerAdapter：处理器适配器，由springmvc提供

Handler：处理器，需要程序员开发

ViewResolver：视图解析器，由springmvc提供

View：真正视图页面需要由程序编写

# 入门程序

## 需求

实现商品列表查询

## 需要的jar包

使用spring3.2.0（带springwebmvc模块）



## 前端控制器

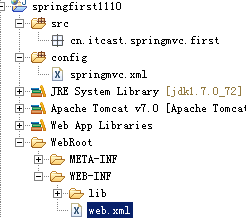
在web.xml中配置：

*<!-- 前端控制器 -->*<**servlet**>  
 <**servlet-name**>springmvc</**servlet-name**>  
 <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  
 *<!-- 加载springmvc配置 -->* <**init-param**>  
 <**param-name**>contextConfigLocation</**param-name**>  
 *<!-- 配置文件的地址  
 如果不配置contextConfigLocation，  
 默认查找的配置文件名称classpath下的：servlet名称+"-serlvet.xml"即：springmvc-serlvet.xml  
 -->* <**param-value**>classpath:springmvc.xml</**param-value**>  
 </**init-param**>  
</**servlet**>  
  
<**servlet-mapping**>  
 <**servlet-name**>springmvc</**servlet-name**>  
 *<!--  
 可以配置/ ，此工程 所有请求全部由springmvc解析，此种方式可以实现 RESTful方式，需要特殊处理对静态文件的解析不能由springmvc解析  
 可以配置\*.do或\*.action，所有请求的url扩展名为.do或.action由springmvc解析，此种方法常用  
 不可以/\*，如果配置/\*，返回jsp也由springmvc解析，这是不对的。  
 -->* <**url-pattern**>\*.action</**url-pattern**>  
</**servlet-mapping**>

## springmvc.xml

在springmvc.xml中配置springmvc架构三大组件（处理器映射器、适配器、视图解析器）

## 工程结构



## 处理器映射器

在springmvc.xml中配置：

BeanNameUrlHandlerMapping： 根据请求url（XXXX.action）匹配spring容器bean的 name

找到对应的bean（程序编写的Handler）

*<!-- 【配置处理器映射器】 springmvc框架根据HandlerMapping接口判断是否是处理器映射器 -->  
<!-- 根据bean的name进行查找Handler 将action的url配置在bean的name中 -->*<**bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"**/>

所有处理器映射器都实现HandlerMapping接口。

## 处理器适配器

在springmvc.xml配置：

*<!-- 【配置处理器适配器】 springmvc框架根据HandlerAdapter接口判断是否是处理器适配器 -->*<**bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"**/>

所有的适配器都是实现了HandlerAdapter接口。

程序编写Handler根据适配器的要求编写。

SimpleControllerHandlerAdapter适配器要求：

通过supports方法知道Handler必须要实现哪个接口：

@Override  
**public boolean** supports(Object handler) {  
 **return** (handler **instanceof** Controller);  
}

## Handler编写

需要实现Controller接口：

**public interface** Controller {  
  
 */\*\*  
 \* Process the request and return a ModelAndView object which the DispatcherServlet  
 \* will render. A {****@code*** *null} return value is not an error: it indicates that  
 \* this object completed request processing itself and that there is therefore no  
 \* ModelAndView to render.  
 \** ***@param request*** *current HTTP request  
 \** ***@param response*** *current HTTP response  
 \** ***@return*** *a ModelAndView to render, or {****@code*** *null} if handled directly  
 \** ***@throws*** *Exception in case of errors  
 \*/* ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) **throws** Exception;  
  
}

@Override  
**public** ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) **throws** Exception {  
 *// 使用静态数据将商品信息列表显示在jsp页面  
 // 商品列表* List<Product> productList = **new** ArrayList<Product>();  
  
 Product product\_1 = **new** Product();  
 product\_1.setName(**"联想笔记本"**);  
 product\_1.setPrice(6000f);  
 product\_1.setCreateTime(**new** Date());  
 product\_1.setDetail(**"ThinkPad T430 联想笔记本电脑！"**);  
  
 Product product\_2 = **new** Product();  
 product\_2.setName(**"苹果手机"**);  
 product\_2.setPrice(5000f);  
 product\_2.setDetail(**"iphone6苹果手机！"**);  
  
 productList.add(product\_1);  
 productList.add(product\_2);  
  
  
 ModelAndView modelAndView = **new** ModelAndView();  
 *// 将数据填充到request  
 //request.setAttribute("productList", productList);* modelAndView.addObject(**"productList"**, productList);  
 *// 指定转发jsp页面* modelAndView.setViewName(**"/WEB-INF/jsps/product/productList.jsp"**);  
 **return** modelAndView;  
}

## 配置Handler

在springmvc.xml配置Handler由spring管理Handler。

*<!-- 【配置Handler】 由于使用了BeanNameUrlHandlerMapping处理映射器，name配置为url -->*<**bean id="itemController1" name="/productList.action" class="cn.qdgxy.springmvc.first.ProductController1"**/>

## 配置视图解析器

配置视图解析，能够解析jsp视图：

*<!-- 【配置视图解析器】 要求将jstl的包加到classpath -->  
<!-- ViewResolver -->*<**bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"**></**bean**>

## 工程部署

访问：http://localhost:8080/springfrist1110/itemList.action



## 异常

### HandlerMapping没有找到Handler

HandlerMapping没有找到Handler，404后边不显示jsp路径



### jsp页面地址错误



# 其它非注解处理器映射器和适配器

## BeanNameUrlHandlerMapping(映射器)

根据请求url（XXXX.action）匹配spring容器bean的name

找到对应的bean（程序编写的Handler）

## SimpleUrlHandlerMapping(映射器)

*<!--【简单url映射】 集中配置bean的id对应 的url -->*<**bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"**>  
 <**property name="mappings"**>  
 <**props**>  
 <**prop key="/productList1.action"**>productController1</**prop**>  
 <**prop key="/productList2.action"**>productController1</**prop**>  
 </**props**>  
 </**property**>  
</**bean**>

注意：在springmvc.xml配置了多个处理器映射器，多个处理器映射器可以共存。

## SimpleControllerHandlerAdapter(适配器)

要求程序编写的Handler(Controller)需要实现 Controller接口。

## HttpRequestHandlerAdapter（适配器）

在springmvc.xml配置：HttpRequestHandlerAdapter

要求Handler 实现 HttpRequestHandler 接口

### 开发Handler

**public class** ProductController2 **implements** HttpRequestHandler {  
  
 @Override  
 **public void** handleRequest(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  
 List<Product> productList = **new** ArrayList<>();  
  
 Product product\_1 = **new** Product();  
 product\_1.setName(**"联想笔记本"**);  
 product\_1.setPrice(6000f);  
 product\_1.setCreateTime(**new** Date());  
 product\_1.setDetail(**"ThinkPad T430 联想笔记本电脑！"**);  
  
 Product product\_2 = **new** Product();  
 product\_2.setName(**"苹果手机"**);  
 product\_2.setPrice(5000f);  
 product\_2.setDetail(**"iphone6苹果手机！"**);  
  
 productList.add(product\_1);  
 productList.add(product\_2);  
  
 request.setAttribute(**"productList"**, productList);  
 *// 指定转发jsp页面* request.getRequestDispatcher(**"/WEB-INF/jsps/product/productList.jsp"**).forward(request, response);  
 }

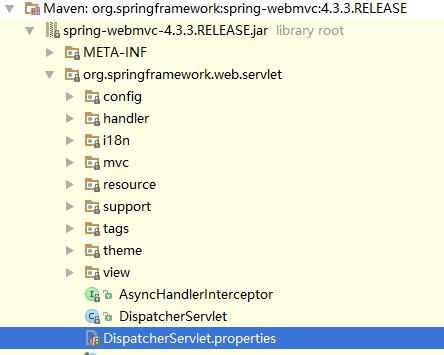
### 配置Handler

*<!-- handler2 -->*<**bean id="productController2" class="cn.qdgxy.springmvc.first.ProductController2"**></**bean**>

使用简单url映射器：

*<!--【简单url映射】 集中配置bean的id对应 的url -->*<**bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"**>  
 <**property name="mappings"**>  
 <**props**>  
 <**prop key="/productList1.action"**>productController1</**prop**>  
 <**prop key="/productList2.action"**>productController2</**prop**>  
 </**props**>  
 </**property**>  
</**bean**>

# DispatcherServlet.properoties



DispatcherServlet前端控制器加载 DispatcherServlet.properoties 配置文件，从而默认加载各各组件，

如果在springmvc.xml中配置了处理器映射器和适配器，以sprintmvc.xml中配置的为准

# 注解映射器和适配器

## 注解映射器

spring3.1之前默认加载映射器是

org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping，3.1之后要使用：

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping

在springmvc.xml中配置RequestMappingHandlerMapping：

*<!-- 注解处理器映射器 -->*<**bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"**/>

使用RequestMappingHandlerMapping需要在Handler 中使用@controller标识此类是一个控制器，使用@requestMapping指定Handler方法所对应的url。

## 注解适配器

spring3.1之前默认加载映射器是

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

3.1之后要使用：

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

*<!-- 注解适配器 -->*<**bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"**/>

RequestMappingHandlerAdapter，不要求Handler实现任何接口，它需要和RequestMappingHandlerMapping注解映射器配对使用，主要解析Handler方法中的形参。

## 注解开发Hanlder

## 配置Handler

*/\*\*  
 \* create:2016-10-18 8:40  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/*@Controller  
**public class** ProductController3 {  
  
 *//商品列表，@RequestMapping中url建议和方法名一致，方便开发维护* @RequestMapping(**"/queryProduct"**)  
 **public** ModelAndView queryProduct() {  
 List<Product> productList = **new** ArrayList<>();  
  
 Product product\_1 = **new** Product();  
 product\_1.setName(**"联想笔记本 ProductController2"**);  
 product\_1.setPrice(6000f);  
 product\_1.setCreateTime(**new** Date());  
 product\_1.setDetail(**"ThinkPad T430 联想笔记本电脑！"**);  
  
 Product product\_2 = **new** Product();  
 product\_2.setName(**"苹果手机"**);  
 product\_2.setPrice(5000f);  
 product\_2.setDetail(**"iphone6苹果手机！"**);  
  
 productList.add(product\_1);  
 productList.add(product\_2);  
  
 ModelAndView modelAndView = **new** ModelAndView();  
 modelAndView.addObject(**"productList"**, productList);  
 *// 指定转发jsp页面* modelAndView.setViewName(**"/WEB-INF/jsps/product/productList.jsp"**);  
 **return** modelAndView;  
 }  
   
}

*<!-- 注解的handler，单个配置 -->  
<!-- <bean class="cn.qdgxy.springmvc.first.ProductController3"/>-->  
<!-- 使用spring组件扫描 -->*<**context:component-scan base-package="cn.qdgxy.springmvc.first"**/>

**建议使用组件扫描**，组件扫描可以扫描@Controller、@Service、@component、@Repsitory

# 小结

DispatcherServlet：前端控制器，相当于中央调度器，可以降低组件之间的耦合度。

HandlerMapping：处理器映射器,负责根据url查找Handler。

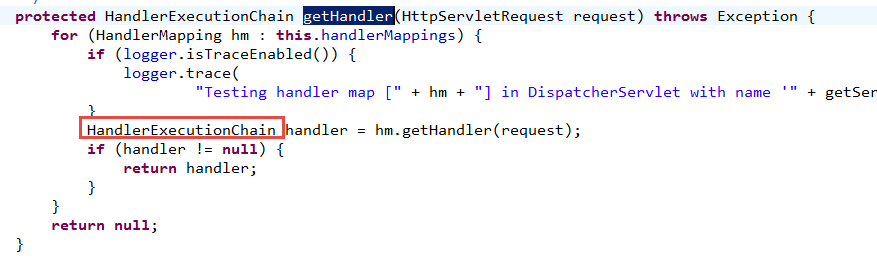
HandlerAdapter：处理器适配器，负责根据适配器要求的规则去执行处理器。可以通过扩展适配器支持不同类型的Handler。

viewResolver：视图解析器，根据逻辑视图名解析成真正的视图，

*<!-- 【配置视图解析器】 要求将jstl的包加到classpath -->  
<!-- ViewResolver -->*<**bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  
 *<!-- 前缀 -->* <**property name="prefix" value="/WEB-INF/jsp/product/"**/>  
 *<!-- 后缀 -->* <**property name="suffix" value=".jsp"**/>  
</**bean**>

真正视图地址==前缀+逻辑视图名+后缀

1、DispatcherServlet通过HandlerMapping查找Handler

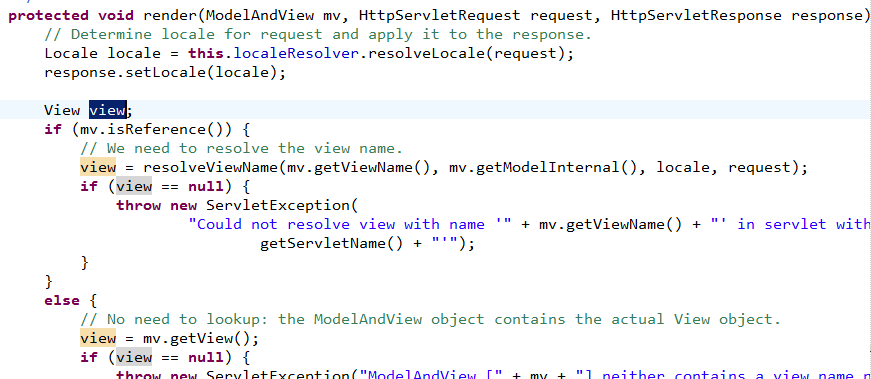


2、DispatcherServlet通过适配器去执行Handler，得到ModelAndview



3、视图解析

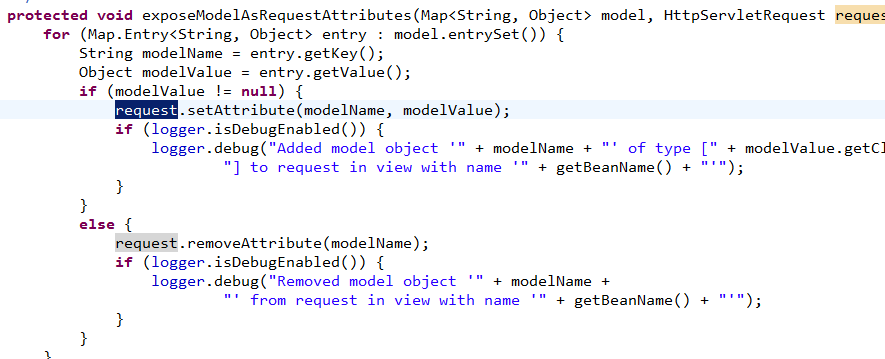
视图解析完成得到一个view：



4、进行视图渲染

将Model中的数据 填充到request域。





# springmvc和mybatis整合工程搭建

## 整合思路

在mybatis和spring整合的基础上 添加springmvc。

spring要管理springmvc编写的Handler（controller）、mybatis的SqlSessionFactory、mapper

第一步：整合dao，spring和mybatis整合

第二步：整合service，spring管理service接口，service中可以调用spring容器中dao(mapper)

第三步：整合controller，spring管理controller接口，在controller调用service

## jar包

mybatis：3.2.7

spring：3.2.0

mybatis的jar

mybatis和spring整合包

spring的所有jar包(包括 springmvc的包)

数据库驱动包

log4j日志



## 工程结构

### 配置文件

applicationContext-dao.xml --- 配置数据源、SqlSessionFactory、mapper扫描器

applicationContext-service.xml --- 配置service接口

applicationContext-transaction.xm l-- 事务管理

sprintmvc.xml --- springmvc的配置，配置处理器映射器、适配器、视图解析器

SqlMapConfig.xml --- mybatis的配置文件，配置别名、settings、mapper

### applicationContext-dao.xml

配置mybatis的数据源、sqlSessionFactory、mapper扫描器

*<!-- 加载配置文件 -->*<**context:property-placeholder location="classpath:dbconfig.properties"**/>  
*<!-- 数据库连接池 -->*<**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}"**/>  
  
 *<!-- 初始化连接:连接池启动时创建的初始化连接数量-->* <**property name="initialSize" value="${dhcp.initialSize}"**/>  
 *<!-- 最大活动连接:连接池在同一时间能够分配的最大活动连接的数量-->* <**property name="maxActive" value="${dhcp.maxActive}"**/>  
 *<!--最大空闲连接:连接池中容许保持空闲状态的最大连接数量,超过的空闲连接将被释放-->* <**property name="maxIdle" value="${dhcp.maxIdle}"**/>  
 *<!-- 最小空闲连接数:低于这个数量会被创建新的连接 -->* <**property name="minIdle" value="${dhcp.minIdle}"**/>  
 *<!--最大等待时间，当没有可用连接时，连接池等待连接释放的最大时间，超过该时间限制会抛出异常，如果设置-1表示无限等待（默认为无限，调整为60000ms，避免因线程池不够用，而导致请求被无限制挂起）-->* <**property name="maxWait" value="${dhcp.maxWait}"**/>  
</**bean**>  
  
*<!-- sqlSessionFactory 整合Mybatis -->*<**bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"**>  
 *<!-- 数据源 -->* <**property name="dataSource" ref="dataSource"**/>  
 *<!-- mybatis配置文件 -->* <**property name="configLocation" value="classpath:mybatis/SqlMapConfig.xml"**/>  
</**bean**>  
  
*<!-- 配置DAO层的sqlSessionTemplate -->  
<!-- <bean id="sqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate">  
 <constructor-arg index="0" ref="sqlSessionFactory"/>  
 </bean>-->  
  
  
<!-- 扫描mapper：方便  
 MapperScannerConfigurer：mapper的扫描器，将包下边的mapper接口自动创建代理对象，  
 自动创建到spring容器中，bean的id是mapper的类名（首字母小写） -->*<**bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"**>  
 *<!-- 配置扫描包的路径  
 如果要扫描多个包，中间使用半角逗号分隔 -->* <**property name="basePackage" value="cn.qdgxy.ssm.mapper"**/>  
 *<!-- 使用sqlSessionFactoryBeanName -->* <**property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"**/>  
</**bean**>

### applicationContext-transation.xml

在此配置文件配置事务，声明式事务控制。

*<!-- 配置事务管理器 -->*<**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  
 *<!-- 注入sessionFactory -->* <**property name="dataSource" ref="dataSource"**/>  
</**bean**>  
  
*<!-- 开启注解的事务管理 -->*<**tx:annotation-driven transaction-manager="transactionManager"**/>  
  
*<!-- 通知 -->  
<!--<tx:advice id="txAdvice" transaction-manager="transactionManager">  
 <tx:attributes>  
 <tx:method name="save\*" propagation="REQUIRED"/>  
 <tx:method name="insert\*" propagation="REQUIRED"/>  
 <tx:method name="update\*" propagation="REQUIRED"/>  
 <tx:method name="delete\*" propagation="REQUIRED"/>  
 <tx:method name="find\*" propagation="SUPPORTS" read-only="true"/>  
 <tx:method name="select\*" propagation="SUPPORTS" read-only="true"/>  
 <tx:method name="get\*" propagation="SUPPORTS" read-only="true"/>  
 </tx:attributes>  
</tx:advice>-->  
  
<!-- aop -->  
<!--<aop:config>  
 <aop:advisor advice-ref="txAdvice"  
 pointcut="execution(\* cn.itcast.ssm.service.impl.\*.\*(..))"/>  
</aop:config>-->*

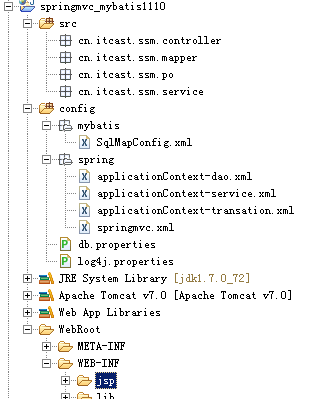
## 前端控制器配置

*<?***xml version="1.0" encoding="UTF-8"***?>*<**web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  
 version="3.1"**>  
  
 *<!-- 前端控制器 -->* <**servlet**>  
 <**servlet-name**>springmvc</**servlet-name**>  
 <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  
 *<!-- 加载springmvc配置 -->* <**init-param**>  
 <**param-name**>contextConfigLocation</**param-name**>  
 *<!-- 配置文件的地址  
 如果不配置contextConfigLocation，  
 默认查找的配置文件名称classpath下的：servlet名称+"-serlvet.xml"即：springmvc-serlvet.xml  
 -->* <**param-value**>classpath:spring/springmvc.xml</**param-value**>  
 </**init-param**>  
 </**servlet**>  
  
 <**servlet-mapping**>  
 <**servlet-name**>springmvc</**servlet-name**>  
 *<!--  
 可以配置/ ，此工程 所有请求全部由springmvc解析，此种方式可以实现 RESTful方式，需要特殊处理对静态文件的解析不能由springmvc解析  
 可以配置\*.do或\*.action，所有请求的url扩展名为.do或.action由springmvc解析，此种方法常用  
 不可以/\*，如果配置/\*，返回jsp也由springmvc解析，这是不对的。  
 -->  
 <!--<url-pattern>\*.action</url-pattern>-->* <**url-pattern**>/</**url-pattern**>  
 </**servlet-mapping**>  
</**web-app**>

## 配置springmvc.xml

*<!-- 【配置Handler】 -->  
<!-- 使用spring组件扫描 -->*<**context:component-scan base-package="cn.qdgxy.ssm.controller"**/>  
  
  
*<!-- 通过annotation-driven可以替代下边的处理器映射器和适配器 -->*<**mvc:annotation-driven conversion-service="conversionService"**/>  
  
*<!-- 【注解处理器映射器】 -->*<**bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"**/>  
  
*<!-- 【注解适配器】 -->*<**bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"**></**bean**>  
  
  
*<!-- 【视图解析器】 要求将jstl的包加到classpath -->  
<!-- ViewResolver -->*<**bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  
 *<!-- 这里的配置我的理解是自动给后面action的方法return的字符串加上前缀和后缀，变成一个 可用的url地址 -->  
 <!-- 前缀 -->* <**property name="prefix" value="/WEB-INF/jsp/product/"**/>  
 *<!-- 后缀 -->* <**property name="suffix" value=".jsp"**/>  
</**bean**>

### 工程结构



## 商品列表开发

### 需求

查询商品列表

### mapper

功能描述：根据条件查询商品信息，返回商品列表

一般情况下针对查询mapper需要自定义mapper。

首先针对单表进行逆向工程，生成代码。

#### mapper.xml

*<!-- 商品查询的sql片段  
 建议是以单表为单位定义查询条件  
 建议将常用的查询条件都写出来 -->*<**sql id="query\_product\_where"**>  
 <**if test="productCustom!=null"**>  
 <**if test="productCustom.name!=null and productCustom.name!=''"**>  
 and name like '%${productCustom.name}%'  
 </**if**>  
 <**if test="productCustom.id!=null"**>  
 and id = #{productCustom.id}  
 </**if**>  
 </**if**>  
</**sql**>  
  
*<!-- 商品查询  
 parameterType：输入 查询条件 -->*<**select id="findProductList" parameterType="cn.qdgxy.ssm.po.ProductQueryVo"  
 resultType="cn.qdgxy.ssm.po.ProductCustom"**>  
 SELECT *\** FROM product  
 <**where**>  
 <**include refid="query\_product\_where"**/>  
 </**where**>  
</**select**>

#### 包装类：

**public class** ProductQueryVo {  
  
 **private** ProductCustom **productCustom**;

#### mapper.java

*/\*\*  
 \* 商品查询  
 \*  
 \** ***@param productQueryVo*** *查询条件  
 \** ***@return*** *商品包装类  
 \*/*List<ProductCustom> findProductList(ProductQueryVo productQueryVo);

### service

*/\*\*  
 \* 商品Service实现  
 \* create:2016-10-30 20:33  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/*@Service  
@Transactional  
**public class** ProductServiceImpl **implements** ProductService {  
  
 @Resource  
 **private** ProductMapper **productMapper**;  
  
 */\*\*  
 \* 商品查询  
 \*  
 \** ***@param productQueryVo*** *查询条件  
 \** ***@return*** *商品包装类  
 \*/* @Override  
 **public** List<ProductCustom> findProductList(ProductQueryVo productQueryVo) **throws** Exception {  
 **return productMapper**.findProductList(productQueryVo);  
 }  
   
}

### 在applicationContext-service.xml中配置service

*<!-- 配置注解需要扫描的包 -->*<**context:component-scan base-package="cn.qdgxy.ssm.service"**/>

### controller

*/\*\*  
 \* 商品控制器  
 \* create:2016-10-30 20:51  
 \* <p>  
 \* author 李欣  
 \*  
 \** ***@version*** *1.0  
 \*/*@Controller**public class** ProductController {  
  
 @Resource  
 **private** ProductService **productService**;  
  
 */\*\*  
 \* 查找所有商品  
 \*  
 \** ***@return*** *ModelAndView  
 \** ***@throws*** *Exception 异常  
 \*/* @RequestMapping(**"findAllProduct"**)  
 **public** ModelAndView findAllProduct() **throws** Exception {  
 *//调用service查询商品列表* List<ProductCustom> productList = **productService**.findProductList(**null**);  
  
 ModelAndView modelAndView = **new** ModelAndView();  
 modelAndView.addObject(**"productList"**, productList);  
 *// 指定逻辑视图名* modelAndView.setViewName(**"productList"**);  
  
 **return** modelAndView;  
 }

}

### jsp



### 在web.xml配置spring监听器

*<!-- 配置spring容器监听器 -->*<**context-param**>  
 <**param-name**>contextConfigLocation</**param-name**>  
 <**param-value**>classpath:spring/applicationContext-\*.xml</**param-value**>  
</**context-param**>  
  
<**listener**>  
 <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**>  
</**listener**>

# 注解开发基础

## 商品修改

### 需求

功能描述：商品信息修改

操作流程：

1、在商品列表页面点击修改连接

2、打开商品修改页面，显示了当前商品的信息

根据商品id查询商品信息

3、修改商品信息，点击提交。

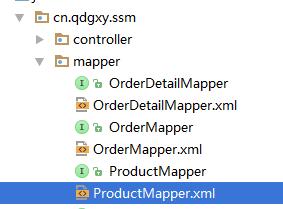
更新商品信息

### mapper

使用逆向工程生成代码：

根据商品id查询商品信息

更新商品信息



### service

*/\*\*  
 \* 商品Service接口  
 \* create:2016-10-30 20:31  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public interface** ProductService {  
  
 */\*\*  
 \** ***@param productQueryVo*** *查询条件  
 \** ***@return*** *商品包装类  
 \** ***@throws*** *Exception 异常  
 \*/* List<ProductCustom> findProductList(ProductQueryVo productQueryVo) **throws** Exception;  
  
 */\*\*  
 \* 根据商品id查询商品信息  
 \*  
 \** ***@param id*** *id  
 \** ***@return*** *商品包装类  
 \** ***@throws*** *Exception 异常  
 \*/* ProductCustom findProductById(Integer id) **throws** Exception;  
  
 */\*\*  
 \* 定义service接口，遵循单一职责，将业务参数细化 （不要使用包装类型，比如map）  
 \*  
 \** ***@param id*** *修改商品的id  
 \** ***@param productCustom*** *修改商品的信息  
 \** ***@throws*** *Exception 异常  
 \*/* **void** updateProduct(Integer id, ProductCustom productCustom) **throws** Exception;  
   
}

## @RequestMapping

### 设置方法对应的url（完成url映射）

一个方法对应一个url

@RequestMapping(**"/findAllProduct"**)  
**public** ModelAndView findAllProduct(HttpServletRequest request) **throws** Exception {

### 窄化请求映射

在class上定义根路径

@Controller  
@RequestMapping(**"/product"**) *//定义url的根路径，访问时根路径+方法的url***public class** ProductController {

好处：更新规范系统 的url，避免 url冲突。

### 限制http请求的方法

通过requestMapping限制url请求的http方法，

如果限制请求必须是post，如果get请求就抛出异常：



商品修改方法，限制为http的get：

*// method = RequestMethod.GET 限制使用get方法*@RequestMapping(value=**"/editProduct"**, method = RequestMethod.***GET***)

## controller方法返回值

### 返回ModelAndView

*/\*\*  
 \* 修改商品  
 \*  
 \** ***@return*** *ModelAndView  
 \** ***@throws*** *Exception 异常  
 \*/  
// method = RequestMethod.GET 限制使用get方法*@RequestMapping(value = **"/editProduct"**, method = RequestMethod.***GET***)  
**public** ModelAndView editProduct() **throws** Exception {  
 ModelAndView modelAndView = **new** ModelAndView();  
  
 *// 调用service查询商品信息* ProductCustom productCustom = **productService**.findProductById(1);  
 *// 将模型数据传递到jsp* modelAndView.addObject(**"product"**, productCustom);  
 *// 指定逻辑视图名* modelAndView.setViewName(**"editProduct"**);  
  
 **return** modelAndView;  
}

### 返回字符串

如果controller方法返回jsp页面，可以简单将方法返回值类型定义 为字符串，最终返回逻辑视图名。

*//方法返回 字符串，字符串就是逻辑视图名，Model作用是将数据填充到request域，在页面展示*@RequestMapping(value = **"/editProduct"**)  
**public** String editProduct(Model model) **throws** Exception {  
 ModelAndView modelAndView = **new** ModelAndView();  
  
 *// 调用service查询商品信息* ProductCustom productCustom = **productService**.findProductById(1);  
  
 *// 将模型数据传递到jsp* model.addAttribute(**"product"**, productCustom);  
  
 **return "editProduct"**;  
}

### 返回void

@RequestMapping(value = **"/editProduct"**)  
**public void** editProduct(HttpServletRequest request, HttpServletResponse response) **throws** Exception {  
 *// 调用service查询商品信息* ProductCustom productCustom = **productService**.findProductById(1);  
  
 *// 将模型数据传递到jsp* request.setAttribute(**"product"**, productCustom);  
  
 request.getRequestDispatcher(**"/WEB-INF/jsps/product/editProduct.jsp"**).forward(request, response);  
}

使用此方法，容易输出json、xml格式的数据：

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"**);

### redirect重定向

如果方法重定向到另一个url，方法返回值为“redirect:url路径”

使用redirect进行重定向，request数据无法共享，url地址栏会发生变化的。

### forward转发

使用forward进行请求转发，request数据可以共享，url地址栏不会。

方法返回值为“forward:url路径”

@RequestMapping(**"/editProductSubmit"**)  
**public** String editProductSubmit() **throws** Exception {  
 *//调用service接口更新商品信息  
 // productService.updateProduct();  
  
 //请求重定向  
 //return "redirect:findAllProduct";  
 //转发* **return "forward:findAllProduct"**;  
}

## 参数绑定

### 参数绑定过程

http请求

HanlderAdapter(适配器)

Handler

以方法为单位进行编写

方法形参

key、value数据

早期springmvc是使用

PropertyEditor属性编辑器进行参数绑定(仅支持由字符串传为其它类型)

后期springmvc是使用

converter转换器进行参数绑定（支持任意类型转换）

将key/value数据绑定到方法形参

### 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

通过model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。

### @RequestParam

如果request请求的参数名和controller方法的形参数名称一致，适配器自动进行参数绑定。如果不一致可以通过

@RequestParam 指定request请求的参数名绑定到哪个方法形参上。

对于必须要传的参数，通过@RequestParam中属性**required**设置为true，如果不传此参数则报错。

对于有些参数如果不传入，还需要设置默认值，使用@RequestParam中属性**defaultvalue**设置默认值。

### 可以绑定简单类型

可以绑定整型、字符串、单精/双精度、日期、布尔型。

### 可以绑定简单pojo类型

简单pojo类型只包括简单类型的属性。

绑定过程：

request请求的参数名称和pojo的属性名一致，就可以绑定成功。

问题：

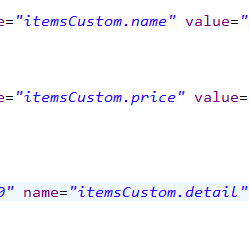
如果controller方法形参中有多个pojo且pojo中有重复的属性，使用简单pojo绑定无法有针对性的绑定，

比如：方法形参有items和User，pojo同时存在name属性，从http请求过程的name无法有针对性的绑定到items或user。

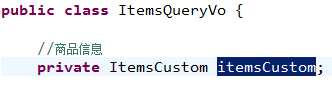
### 可以绑定包装的pojo

包装的pojo里边包括了pojo。

页面参数定义：



包装类型的属性也是itemsCustom：



按照上边的规则进行包装类型的绑定。

### 自定义绑定使用属性编辑器

springmvc没有提供默认的对日期类型的绑定，需要自定义日期类型的绑定。

#### 使用WebDataBinder（了解）

在controller类中定义：

*//自定义属性编辑器*@InitBinder  
**public void** initBinder(WebDataBinder binder) **throws** Exception {  
 *// Date.class必须是与controller方法形参pojo属性一致的date类型，这里是java.util.Date* binder.registerCustomEditor(Date.**class**, **new** CustomDateEditor(  
 **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**), **true**));  
}

使用这种方法问题是无法在多个controller共用。

#### 使用WebBindingInitializer（了解）

使用WebBindingInitializer让多个controller共用 属性编辑器。

自定义WebBindingInitializer，注入到处理器适配器中。

如果想多个controller需要共同注册相同的属性编辑器，可以实现PropertyEditorRegistrar接口，并注入webBindingInitializer中。

如下：

编写CustomPropertyEditor：

*/\*\*  
 \* 自定义属性编辑器  
 \* create:2016-11-05 20:22  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public class** CustomPropertyEditor **implements** PropertyEditorRegistrar {  
  
 @Override  
 **public void** registerCustomEditors(PropertyEditorRegistry propertyEditorRegistry) {  
 *// Date.class必须是与controller方法形参pojo属性一致的date类型，这里是java.util.Date* propertyEditorRegistry.registerCustomEditor(Date.**class**, **new** CustomDateEditor(  
 **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**), **true**));  
 }  
  
}

配置如下：

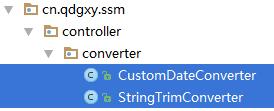
*<!-- 注册属性编辑器 -->*<**bean id="customPropertyEditor" class="cn.qdgxy.ssm.controller.propertyeditor.CustomPropertyEditor"**/>  
  
*<!-- 自定义webBinder -->*<**bean id="customBinder" class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer"**> *<!--propertyEditorRegistrars用于属性编辑器-->* <**property name="propertyEditorRegistrars"**>  
 <**list**>  
 <**ref bean="customPropertyEditor"**/>  
 </**list**>  
 </**property**>  
</**bean**>

*<!-- 【注解适配器】 -->*<**bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"**>  
 *<!-- 在webBindingInitializer中注入自定义属性编辑器、自定义转换器 -->* <**property name="webBindingInitializer" ref="customBinder"**/>  
</**bean**>

### 自定义参数绑定使用转换器(架构师掌握)

#### 实现Converter接口：

定义日期类型转换器和字符串去除前后空格转换器。



*/\*\*  
 \* 自定义日期转换器  
 \* create:2016-11-05 22:06  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public class** CustomDateConverter **implements** Converter<String, Date> {  
  
 @Override  
 **public** Date convert(String source) {  
 **try** {  
 *//进行日期转换* **return new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**).parse(source);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
  
}

*/\*\*  
 \* 字符串去除前后空格转换器  
 \* create:2016-11-05 22:11  
 \*  
 \** ***@author*** *李欣  
 \** ***@version*** *1.0  
 \*/***public class** StringTrimConverter **implements** Converter<String, String> {  
  
 @Override  
 **public** String convert(String source) {  
 **try** {  
 *//去掉字符串两边空格，如果去除后为空设置为null* **if** (source != **null**) {  
 source = source.trim();  
 **if** (source.equals(**""**)) {  
 **return null**;  
 }  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
  
 **return** source;  
 }  
  
}

#### 配置转换器

参考教案：

配置方式1针对不使用<mvc:annotation-driven>（**使用这种**）

*<!-- 【注解适配器】 -->*<**bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"**>  
 *<!-- 在webBindingInitializer中注入自定义属性编辑器、自定义转换器 -->* <**property name="webBindingInitializer" ref="customBinder"**/>  
</**bean**>

*<!-- 自定义webBinder -->*<**bean id="customBinder" class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer"**>  
 *<!-- 使用converter进行参数转 -->* <**property name="conversionService" ref="conversionService"**/>  
</**bean**>  
  
*<!-- conversionService -->*<**bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean"**>  
 *<!-- 转换器 -->* <**property name="converters"**>  
 <**set**>  
 <**bean class="cn.qdgxy.ssm.controller.converter.CustomDateConverter"**/>  
 <**bean class="cn.qdgxy.ssm.controller.converter.StringTrimConverter"**/>  
 </**set**>  
 </**property**>  
</**bean**>

配置方式2针对使用<mvc:annotation-driven>的配置

*<!-- 【配置Handler】 -->  
<!-- 使用spring组件扫描 -->*<**context:component-scan base-package="cn.qdgxy.ssm.controller"**/>  
  
*<!-- 通过annotation-driven可以替代下边的处理器映射器和适配器 -->*<**mvc:annotation-driven conversion-service="conversionService"**/>  
  
*<!-- conversionService -->*<**bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean"**>  
 *<!-- 转换器 -->* <**property name="converters"**>  
 <**set**>  
 <**bean class="cn.qdgxy.ssm.controller.converter.CustomDateConverter"**/>  
 <**bean class="cn.qdgxy.ssm.controller.converter.StringTrimConverter"**/>  
 </**set**>  
 </**property**>  
</**bean**>

## 问题处理

### post乱码

在web.xml中加入：

*<!-- 解决post乱码 -->*<**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