Springmvc基础知识

什么是springmvc?

Springmvc框架原理（掌握）

前端控制器、处理器映射器、处理器配置器、视图解析器

Springmvc入门程序

目的：对前端控制器、处理器映射器、处理器配置器、视图解析器进行学习。

非注解的处理器映射器、处理器适配器

注解的处理器映射器、处理器适配器

Springmvc和mybatis整合（掌握）

SpringMvc注解开发（掌握）

常用的注解学习

参数绑定（简单类型、POJO类型、集合类型）

自定义的参数绑定（掌握）

Springmvc和struts2的区别

Springmvc高级知识

参数绑定（集合类型）

数据回显、

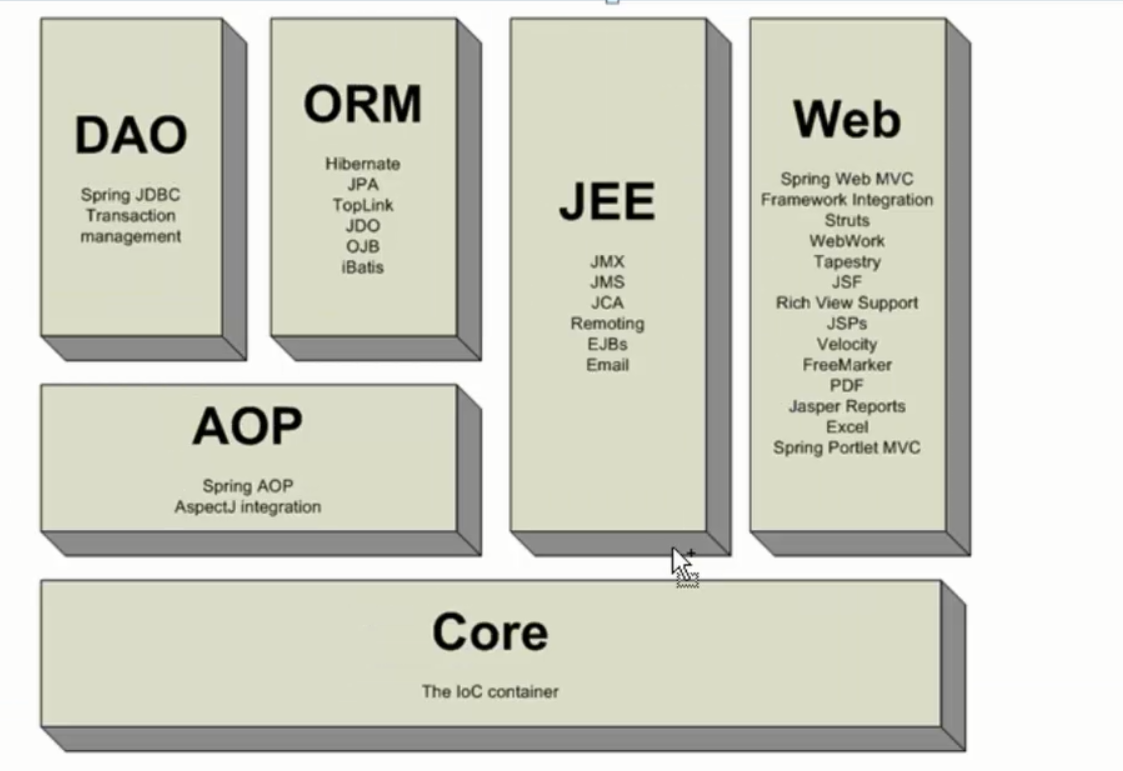
上传图片

JSON数据交互

RESTful支持

拦截器

# springmvc框架学习



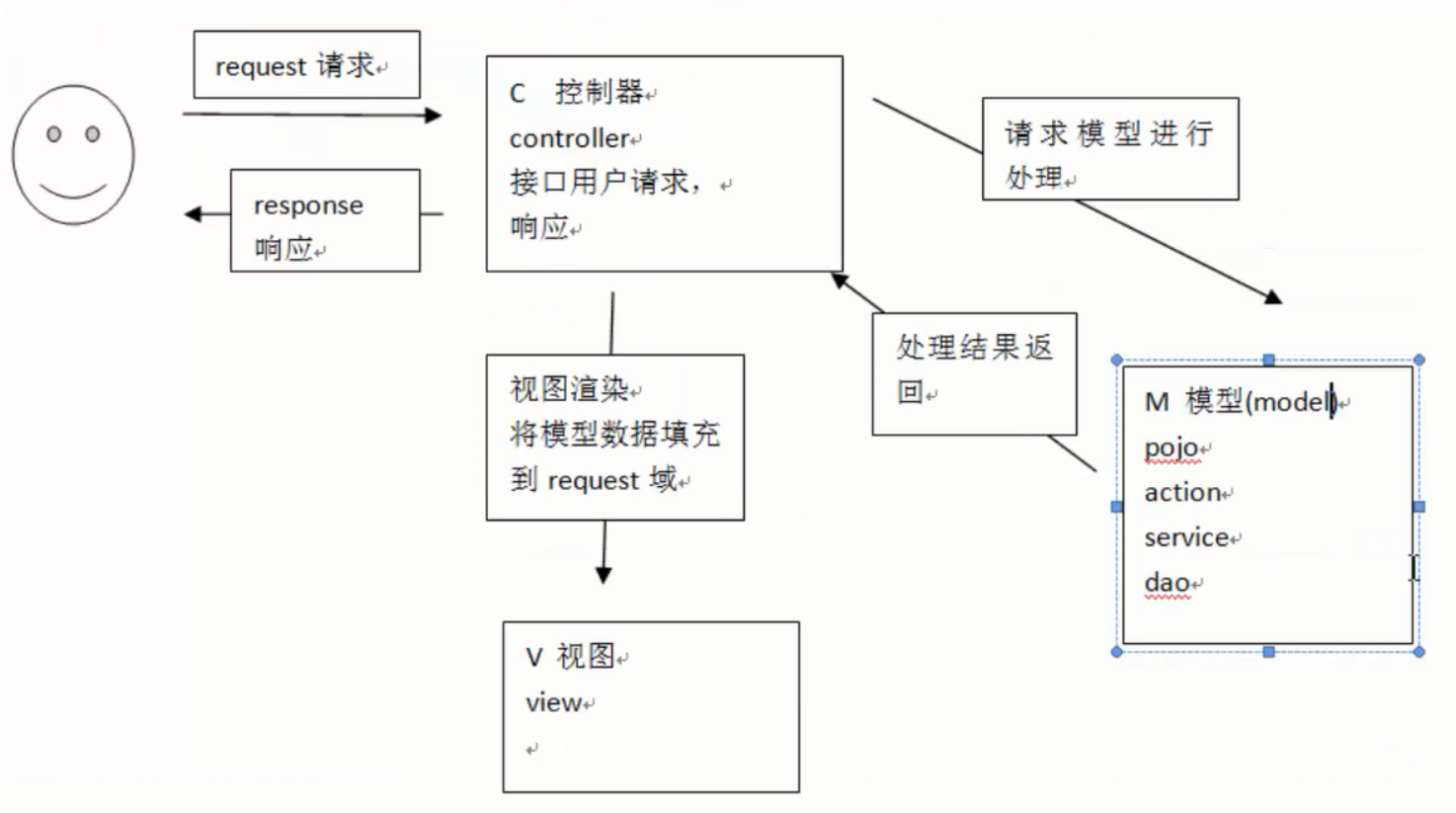
## 1.1什么是springmvc

Springmvc是spring框架的一部分，springmvc和spring无需通过中间层进行整合。

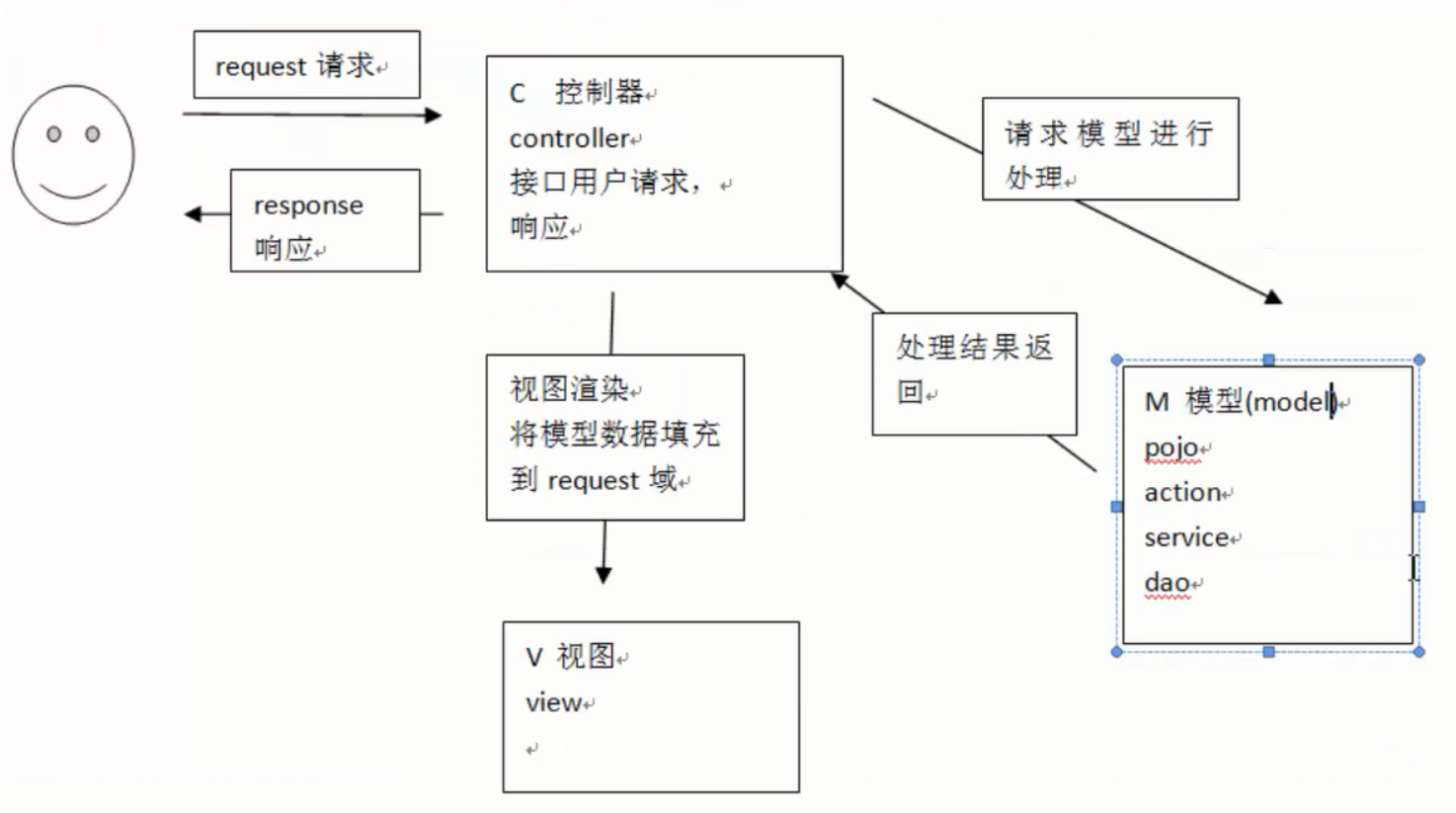
Springmvc是一个基于mvc的web的框架

## 1.2 mvc在B/S系统下的应用

mvc是一个设计模式,mvc在B/S系统下的应用：



## 1.3.Springmvc框架



11.响应

10.视图渲染，将模型数据填充到request域

9.返回view

8.请求进行视图解析

7.返回modeAndview

视图解析器

View resolve

JSP

freeMarker

Excel

pdf

4.请求执行适配器

6.返回modeAndview

5执行

Handler处理器

Controller

处理器适配器

HandlerAdapter

去执行Handler

1.Request请求 url

2.请求查找handler

3.返回一个执行链HandlerExecutionChain{

Interceptor1

Interceptor2

handler

}

处理器映射器

HandlerMapping

前端控制器

DispatcherServlet

第一步：发起请求到前端控制器（DispetcherServlet）

第二步：前端控制器请求HandlerMapping查找Handler

可以根据xml配置和注解进行查找

第三步：处理器映射器HandlerMapping向前端控制器返回Handler

第四步：前端控制器调用处理器适配器去执行Handler

第五步：处理器适配器去执行Handler

第六步： Handler执行完给适配器返回modelAndVIew

第七步：处理器适配器向前端控制器返回ModleAndVIew

ModelAndView是springmvc的底层对象，包括model和view

第八步：前端控制器请求视图解析器去进行视图解析

根据逻辑视图名解析成真正的视图（jsp）

第九步：视图解析器向前端控制器返回view

第十部：前端控制器进行前端视图渲染

视图渲染就是将模型数据（在modelAndView对象中）填充到request域

第十一步：前端控制器向用户响应结果

组件：

1. 前端控制器DispetcherServlet（不需要程序员开发）
   1. 作用：接受请求，响应结果，相当于一个转发器
   2. 有了DispetcherServlet减少了其他组件之间的耦合度，
2. 处理器映射器HandlerMapping（不需要程序员开发）
   1. 作用：根据请求的url查找Handler
3. 处理器Handler（需要程序员开发）
   1. 在开发Handler时候按照HandlerAdapter的要求去做，这样是适配器才可以正确去执行Handler
4. 处理器是配置HandlerAdapter
   1. 作用：按照特定的规则（HandlerAdapter要求的规则）执行Handler
5. 视图解析器view resolve（不需要程序员开发）
   1. 作用：进行视图解析，根据逻辑视图解析成真正的视图（jsp）
6. 视图View
   1. View是一个接口，实现类支持不同的实现类型（jsp/freemarker/pdf）

# 入门程序

## 2.1需求

功能需求：商品的列表查询

## 2.2配置

Springmvc和mybatis使用一个案例

Springmvc的jar包：spring-webmvc



## 2.3配置前端控制器

<display-name>Archetype Created Web Application</display-name>

<!-- 配置前端控制器 -->

<servlet>

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

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

<!-- contextConfigLocation配置spring加载的配置文件（配置处理器映射器、适配器等）

如果不配置contextConfigLocation，默认加载的是/WEB-INF/servlet名称-servlet.xml(springmvc-servlet.xml)

-->

<init-param>

<param-name></param-name>

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

</init-param>

</servlet>

<servlet-mapping>

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

<!--方法1； \*.action 访问以.action结尾的由DispatcherServlet进行解析

方法2；/ 所有访问的action都由DispatcherServlet进行解析，对于静态的文件不让DispatcherServlet进行解析

使用此种方法可以实现RESTful风俗的url

方法3；/\* 这样配置不对，使用这种配置，最终要转发到jsp页面时候，仍然由DispatcherServlet解析jsp地址,

不能由jsp页面找到Handler,会 报错

-->

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

</servlet-mapping>

## 2.4配置处理器适配器

在classpath下的springmvc.xml中配置适配器

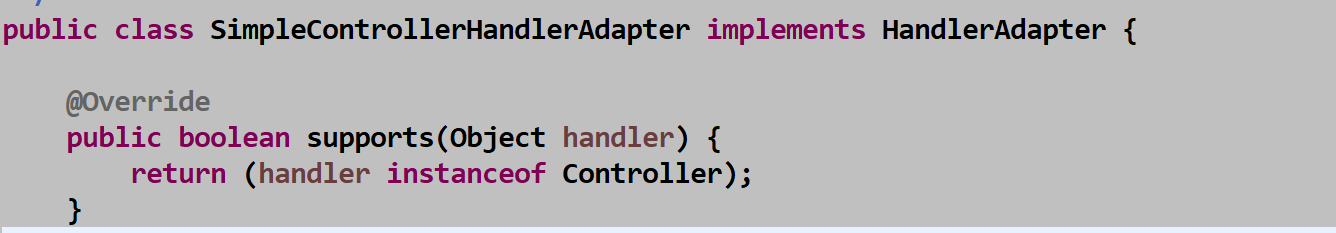
<!-- 配置处理器适配器

所有的适配器都实现了HandlerAdapter接口，

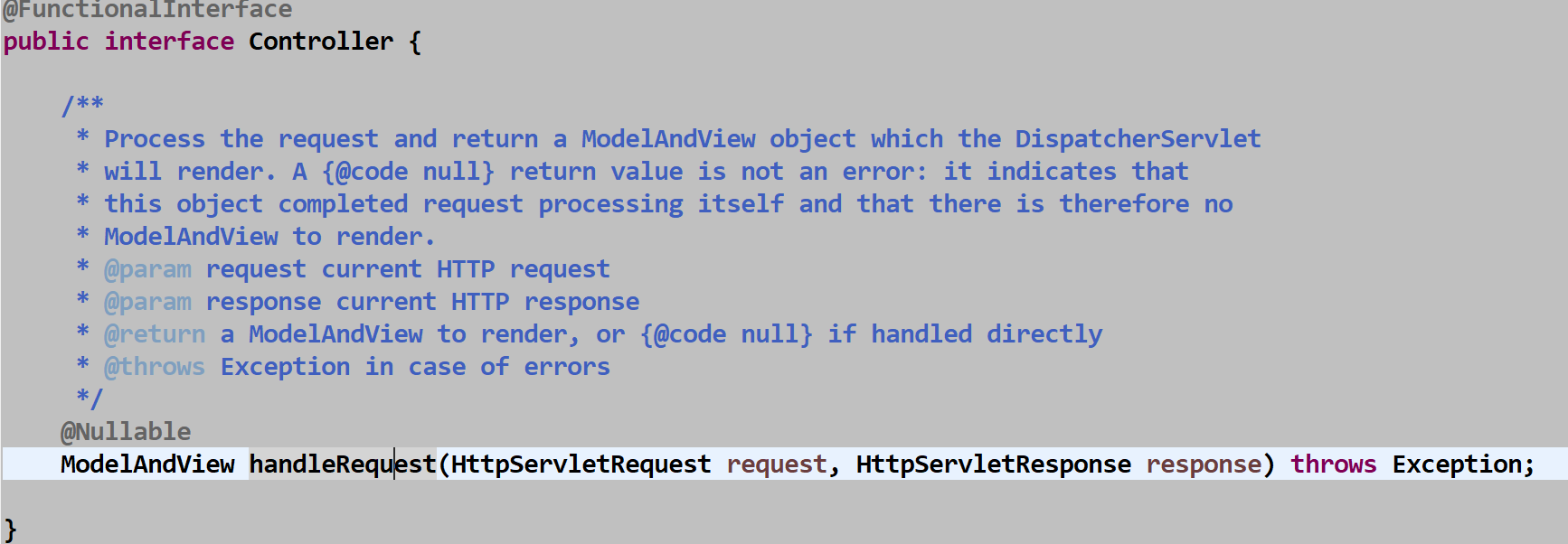
-->

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

源代码：



此适配器能够实现Controller接口的Handler



## 2.5开发Handler（Controller）

需要实现Controller接口，才能由

org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter这个适配器执行

**public** **class** ItemsController **implements** Controller {

@Override

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

//调用service查找数据库，查询商品列表，这里使用静态数据

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

//向list中添加数据

Items item1=**new** Items();

item1.setName("手机");

item1.setPrice("400");

item1.setDetail("手机11111");

Items item2=**new** Items();

item2.setName("电脑");

item2.setPrice("400");

item2.setDetail("iphone 6p");

itemslist.add(item2);

itemslist.add(item1);

//返回ModelAndView

ModelAndView mav=**new** ModelAndView();

//相当于request的setAttribute()方法，在jsp页面通过itemslist获取数据

mav.addObject("itemslist",itemslist);

//指定视图

mav.setViewName("/WEB-INF/jsp/items/itemslist.jsp");

**return** mav;

}

}

## 2.6视图编写

<%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"ISO-8859-1"*%>

<%

String path = request.getContextPath();

String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+path+"/";

%>

<%@ 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">

<html>

<head>

<base href=*"*<%=basePath%>*"*>

<title>商品查询页面</title>

<meta http-equiv=*"pragma"* content=*"no-cache"*>

<meta http-equiv=*"cache-control"* content=*"no-cache"*>

<meta http-equiv=*"expires"* content=*"0"*>

<meta http-equiv=*"keywords"* content=*"keyword1,keyword2,keyword3"*>

<meta http-equiv=*"description"* content=*"This is my page"*>

<!--

<link rel="stylesheet" type="text/css" href="styles.css">

-->

</head>

<body>

<form action=*"*${pageContext.request.contextPath}*/item/queryItem.action"*

method=*"post"*>

查询条件：

<table>

<tr>

<td><input type=*"submit"* value=*"查询"*></td>

</tr>

</table>

商品列表：

<table>

<tr>

<td>商品名称</td>

<td>商品价格</td>

<td>商品日期</td>

<td>商品描述</td>

<td>操作</td>

</tr>

<c:forEach items=*"*${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.details}</td>

<td><a href=*"*${pageContext.request.contextPath}*/item/editItem.action?id=*${item.id}*"*>修改</a></td>

</tr>

</c:forEach>

</table>

</form>

</body>

</html>

## 2.7配置Handler

将编写的Handler在spring容器中加载

<!-- 配置handler -->

<bean name=*"/quetyItems.action"* class=*"com.wbs.controller.ItemsController"*></bean>

## 2.8配置处理器映射器

在classpath下的springmvc.xml中配置处理器映射器

<!-- 配置处理器映射器 :根据bean的name进行查找Handler，将action的url配置在bean的name中；

将bean的name作为url进行映射查找，需要在配置Handler时候指定beanname(url)

-->

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

## 2.9配置试图解析器

需要配置解析jsp的解析器

<!-- 配置试图解析器

解析jsp视图，默认使用jstl标签，所以classpath下边必须要有jstl的包

-->

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

## 2.10部署调试

访问地址：<http://localhost:8080/springmvc/queryItems.action>

# 非注解的处理器映射器和适配器

## 3.1非注解映射器

处理器映射器：

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

另外一种映射器：

*org.springframework.web.servlet.handler.SimpleUrlHandlerMapping*

<!-- 简单url映射 -->

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

<property name=*"mappings"*>

<props>

<!-- 对itemsController1进行url映射，url是/queryItems1.action -->

<prop key=*"/queryItems1.action"*>itemsController1</prop>

<prop key=*"/queryItems2.action"*>itemsController1</prop>

</props>

</property>

</bean>

结论：多个映射器可以并存，前端控制器会判断url能让那个映射器处理就让哪个映射器处理

## 3.2非注解处理器适配器

1.

<!-- 配置处理器映射器 :根据bean的name进行查找Handler，将action的url配置在bean的name中；

将bean的name作为url进行映射查找，需要在配置Handler时候指定beanname(url)

所有的适配器都实现了HandlerMapping接口

-->

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

*org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping要求编写的Handler实现Controller接口。*

2.

<!-- 另一个非注解适配器 -->

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

要求：编写的Handler实现HttpRequestHandler接口

Handler的编写：

**public** **class** ItemsController2 **implements** HttpRequestHandler {

@Override

**public** **void** handleRequest(HttpServletRequest arg0, HttpServletResponse arg1) **throws** ServletException, IOException {

//调用service查找数据库，查询商品列表，这里使用静态数据

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

//向list中添加数据

Items item1=**new** Items();

item1.setName("手机");

item1.setPrice(400f);

item1.setDetail("手机11111");

item1.setCreatetime(**new** Date());

Items item2=**new** Items();

item2.setName("电脑");

item2.setPrice(500f);

item2.setDetail("iphone 6p");

item2.setCreatetime(**new** Date());

itemslist.add(item2);

itemslist.add(item1);

//设置模型数据

arg0.setAttribute("list", itemslist);

//设置转发的视图 arg0.getRequestDispatcher("/WEB-INF/jsp/items/itemslist.jsp").forward(arg0, arg1);

/\* //使用此种方法可以通过修改response来设置响应的格式，比如响应json数据

response.setCharacterEncoding("utf-8");

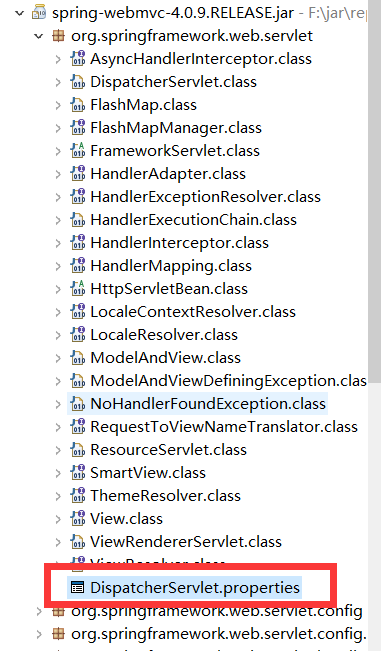
response.setContentType("application/json;charset=utf-8");

response.getWriter().write("json串");\*/

}

测试：localhost:8080/springmvc/queryItems3.action

## DispetcherServler.properties



前端控制器从上边的文件中加载处理映射器、适配器、视图解析器等组件，如果不在springmvc.xml中配置，使用默认加载的。

# 注解的处理器映射器和适配器

## 4.1 配置注解映射器和适配器

在spring3.1之前使用下边的注解映射器：

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

3.1之后用注解映射器:

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

在spring3.1之前使用下边的注解适配器：

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

3.1之后用注解适配器:

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

<!-- 注解映射器 -->

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

<!-- 注解适配器 -->

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

<!-- 使用下边的mvc:annotation-driven代替上边的注解映射器和注解适配器

mvc:annotation-driven默认加载很多的参数绑定方法，比如json转换解析器就默认加载，

如果使用mvc:annotation-driven，

就不需要配置上边的额两种RequestMappingHandlerMapping和RequestMappingHandlerAdapter

在实际开发中用mvc:annotation-driven

-->

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

## 4.2开发注解Handler

使用注解的映射器和注解的适配器。（注解的映射器适配器必须配对）

//使用Controller表示他是一个控制器

@Controller

**public** **class** ItemsController3{

//注解的Handler

//商品列表的查询,建议将器url写成和方法名一样，方便使用

//@RequestMapping实现queryItems方法和url的映射，一个方法对应一个url,.action可加可不加

@RequestMapping("/queryItems.action")

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

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

//向list中添加数据

Items item1=**new** Items();

item1.setName("手机");

item1.setPrice(400f);

item1.setDetail("手机11111");

item1.setCreatetime(**new** Date());

Items item2=**new** Items();

item2.setName("电脑");

item2.setPrice(500f);

item2.setDetail("iphone 6p");

item2.setCreatetime(**new** Date());

itemslist.add(item2);

itemslist.add(item1);

//返回ModelAndView

ModelAndView mav=**new** ModelAndView();

//相当于request的setAttribute()方法，在jsp页面通过itemslist获取数据

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

mav.addObject("list",itemslist);

//指定视图

mav.setViewName("/WEB-INF/jsp/items/itemslist.jsp");

**return** mav;

}

//可以在这个类中定义其他的方法了，比如商品的添加、修改

}

## 4.3 在spring容器中加载Handler

使用组件扫描：

<!-- 对于注解的Handler可以单个配置

在实际开发中建议使用组建扫描

-->

<!-- <bean class="com.wbs.controller.ItemsController3"></bean> -->

<!-- 可以扫描controller、service、

这里让组建扫描controller，指定controller的包,这样的话上边的就可以屏蔽掉

-->

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

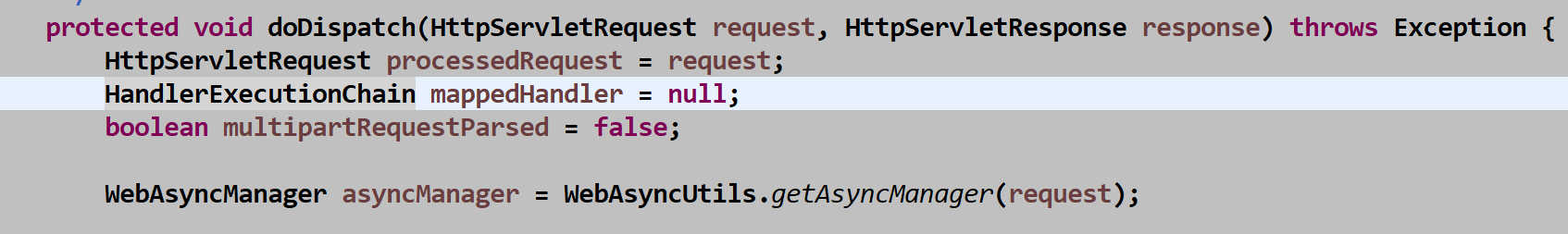
## 4.4 部署访问

http://localhost:8080/springmvc/queryItems4.action

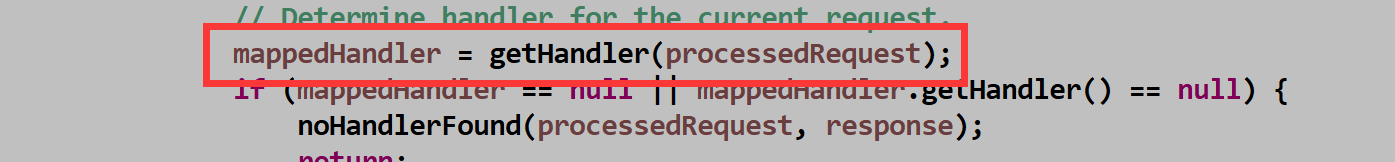
# 5 源码分析

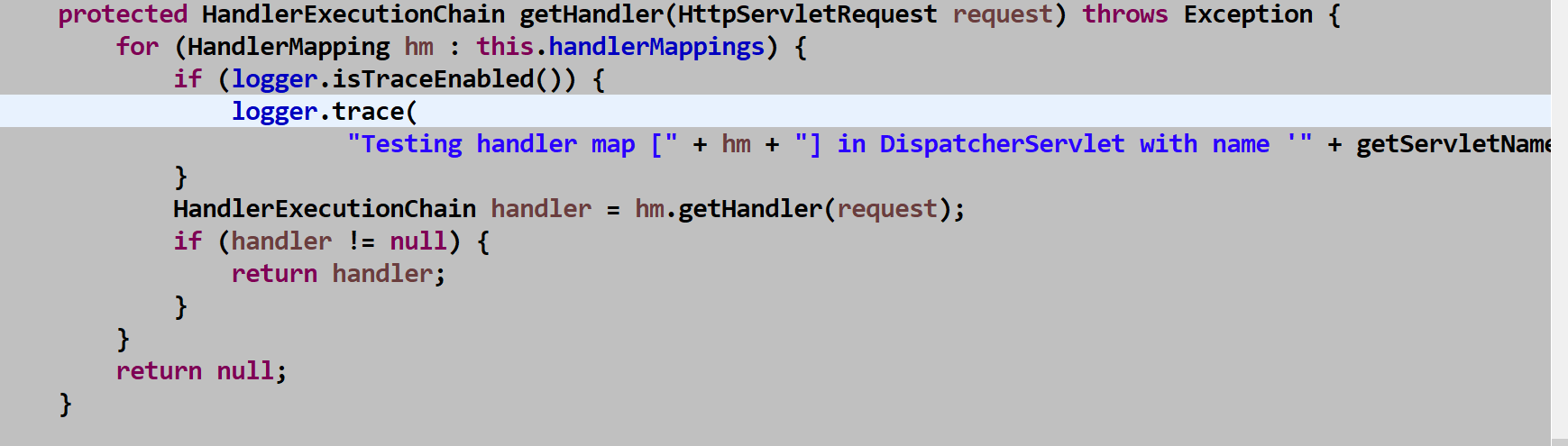
通过前端控制器源码分析springmvc的执行过程

第一步：前端控制器接收请求，调用doDispatcher()方法：

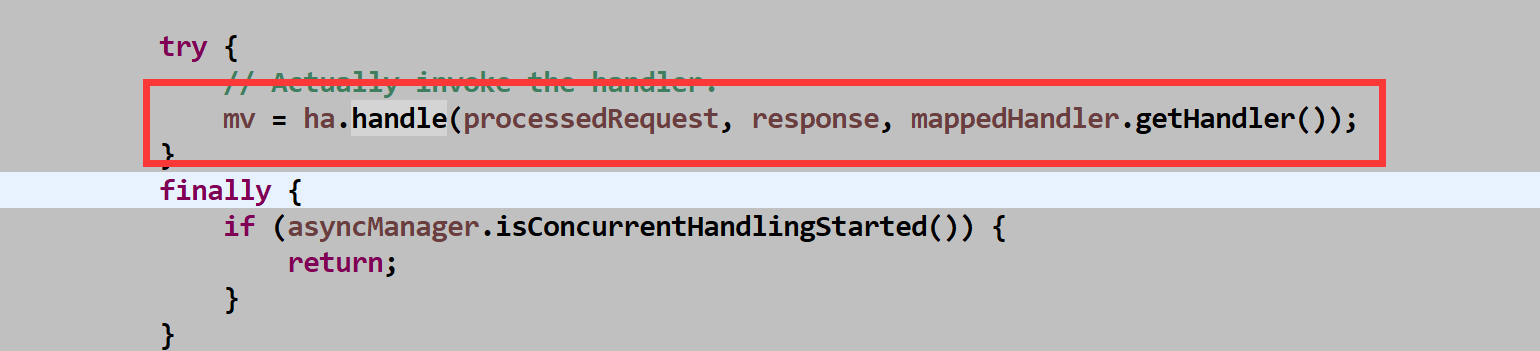


第二步：前端控制器调用处理器映射器查找Handler



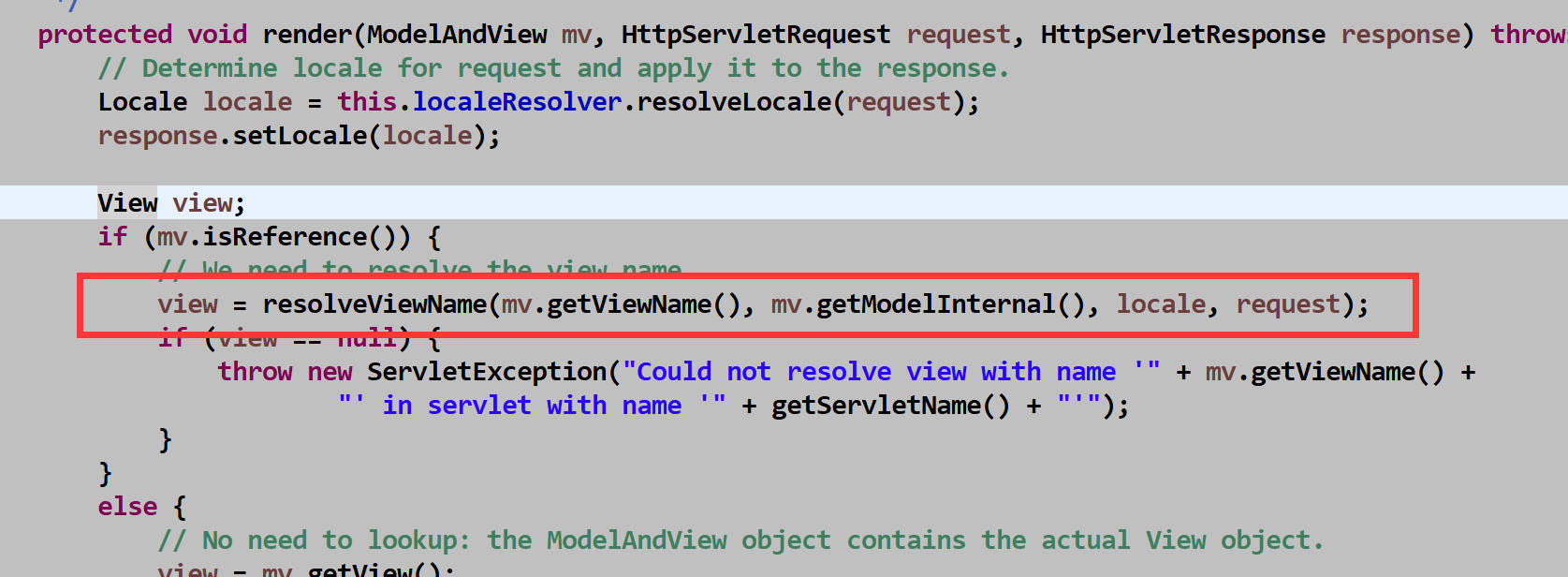


第三部：调用处理器适配器执行Handler，得到执行的结果ModelAndView

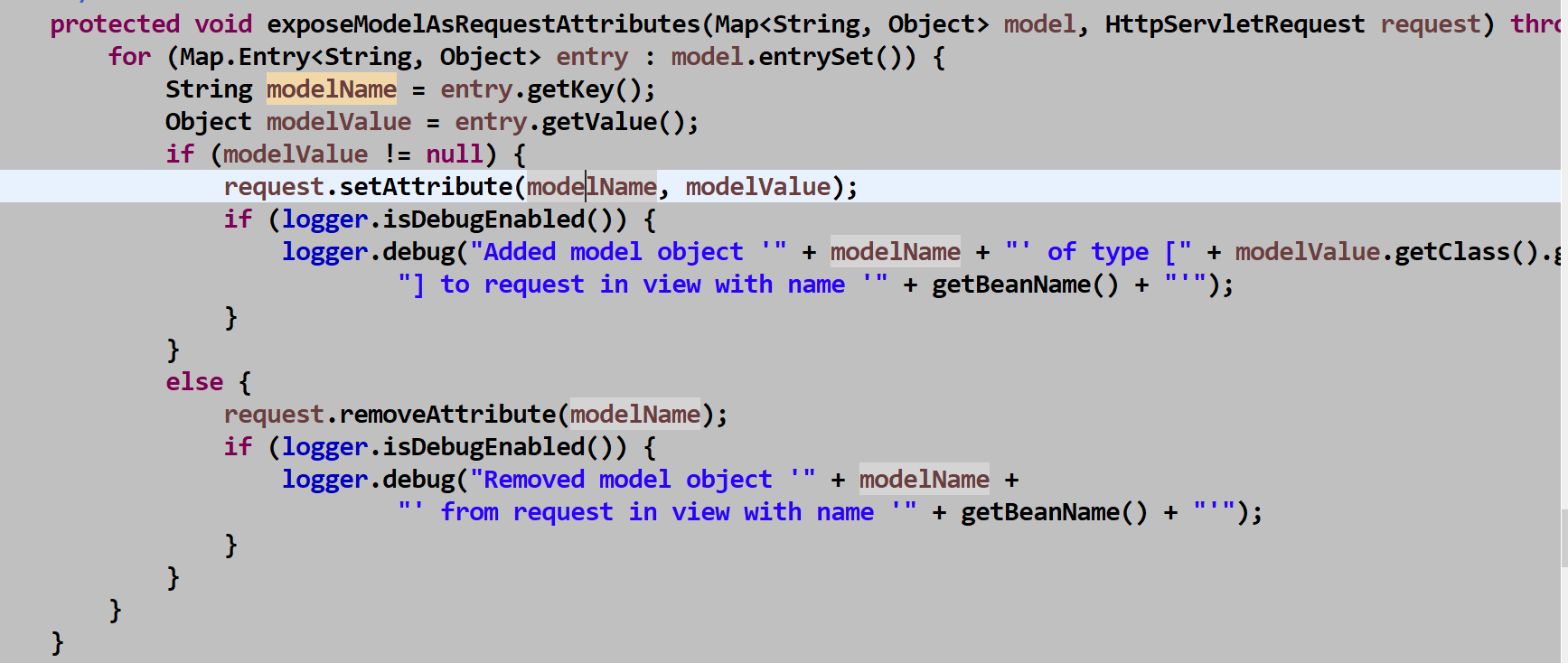


第四步：视图渲染，将model的结果填充到request

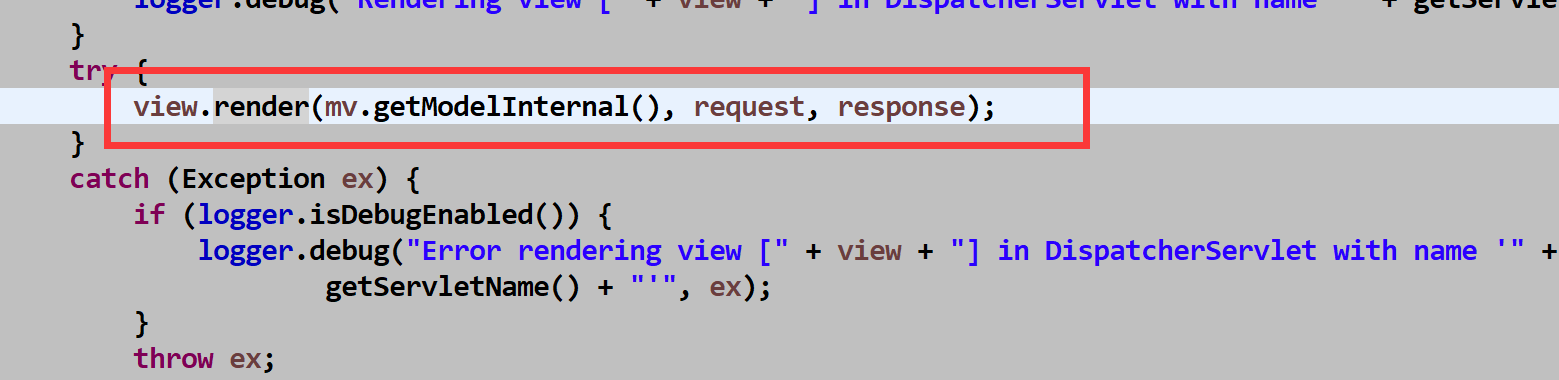
视图解析，得到view



调用view的渲染方法，将model的结果填充到request域



渲染的方法：DispetcherServlet.class的render()方法中的：



以上基本就是springmvc的整个执行流程。

# 6入门程序总结

通过入门程序的理解springmvc的前端控制器、处理器映射器、处理器适配器、视图解析器的用法。

前端控制器配置：

方法1； \*.action 访问以.action结尾的由DispatcherServlet进行解析

方法2；/ 所有访问的action都由DispatcherServlet进行解析，对于静态的文件不让DispatcherServlet进行解析

使用此种方法可以实现RESTful风俗的url

处理器映射器：

1. 非注解（了解）
2. 注解（掌握）
   1. 对标记@Controller的类的@RequestMapping方法进行映射。在@ResuestMapping定义了映射的url
   2. 使用注解的开发方式不用在.xml中配置url和Handler的映射关系

处理器适配器：

1. 非注解
2. 注解
   1. 注解的适配器和注解的映射器是配对使用的，即不能使用非注解的映射器进行映射

实际开发中的适配器和映射器的配置：

<mvc:annotation-driven/>可以代替下边的两行配置：

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

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

视图解析器前缀和后缀的配置：

具体的页面：其中不需要写前缀和后缀：

//指定视图

//如果在视图解析器中配置jsp路径的前缀和jsp路径的后缀，可修改为如下

//mav.setViewName("/WEB-INF/jsp/items/itemslist.jsp");

//上边的路径配置可以不在程序中指定jsp路径的前缀和jsp路径的后缀，如下：

mav.setViewName("items/itemslist");

在springmvc.xml中配置：

<!-- 配置试图解析器

解析jsp视图，默认使用jstl标签，所以classpath下边必须要有jstl的包

-->

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

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

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

</bean>

# springmvc和mybatis整合

## 8.1需求：

使用springmvc和mybatis完成商品列表的查询

## 8.2整合思路

Springmvc+mybatis系统架构：

持久层

mybatis

业务层：

service接口

Service接口

表现层

springmvc

mySql

Spring将各层进行整合，通过spring进行管理持久层的Mpaper(相当于Dao接口)

Spring管理业务层的service,service通过spring管理可以调用mapper接口

Spring进行管理表现层Handler，Handler可以调用service接口

Spring进行事务控制

Mapper、service、Handler、都是JavaBean

第一步：整合Dao层

mybatis和spring整合,通过spring管理mapper接口

使用mapper的扫描器自动扫描maper接口在spring中进行注册

第二步：整合service

通过spring管理service接口

使用配置的方式将service接口配置在spring的文佳中

实现事务控制

第三步：整合springmvc

由于springmvc是spring的模块，不需要整合

## 8.3准备环境

所需要的jar包：

数据库驱动包

mybatis的jar包

mybatis和spring的整合包

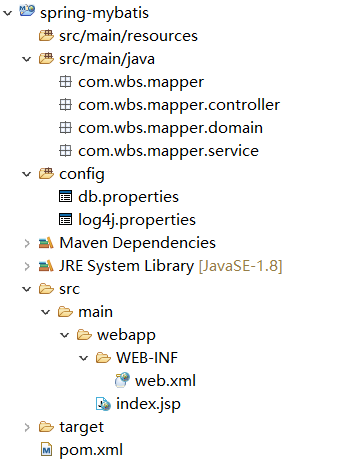
log4j包

数据库连接池包 dbcp

spring3.2所有jar包

jstl包

## 8.4工程结构



## 8.5整合Dao

mybatis和spring的整合包

### 8.5.1 sqlMapConfig.xml

mybatis自己的配置文件

<?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>

<!-- 全局的setting配置，根据需要添加 -->

<!-- <mappers>

<mapper resource=”sqlmap/User.xml”></mapper>

</mappers> -->

<!-- 配置mapper,由于使用spring和mybatis的整合包进行mapper扫描，这里不再配置，但是必须遵循一些规范： mapper.xml文件和mapper.java同名并且在一个目录 -->

<!-- 配置别名：别名可以在mapper.xml中作为resultType来使用，默认扫描的规则是PO类首字母变小写 -->

<typeAliases>

<!-- 批量扫描别名 -->

<package name=*"com.wbs.domain"* />

</typeAliases>

</configuration>

### 8.5.2 applicationContext-dao.xml

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

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

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

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

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

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

xsi:schemaLocation=*"*

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

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

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

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

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

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

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

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

<!-- 加载数据库的配置文件 -->

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

<!-- 数据源 dbcp -->

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

<!-- 注入里面的属性值 -->

<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>

<property name=*"maxActive"* value=*"10"*></property>

<property name=*"maxIdle"* value=*"5"*></property>

</bean>

<!-- sqlsessionFactory -->

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

<!-- 加载mybatis的配置文件-->

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

<!-- 数据源 -->

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

</bean>

<!-- mapper批量扫描，从mapper的包中扫描出mapper的接口，自动创建代理对象并且在spring容器中注册，

需要遵循规范：mapper.xml和mapper.java名称一样并且在同一目录

-->

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

<!-- 如果扫描多个包，每个包中间使用半角逗号分隔 -->

<property name=*"basePackage"* value=*"com.wbs.mapper"*></property>

<!-- 下边的name中只能使用sqlSessionFactoryBeanName，而不能引用sqlSessionFactory

因为如果那样写就不会扫描加载数据库的配置文件。因为MapperScannerConfigurer先执行，别的后执行。

自动扫描出来的bean的id为mapper类名（首字母小写）

-->

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

</bean>

</beans>

sqlSessionFactiory

数据源

mapper扫描器

### 8.5.3 逆向工程生成PO类（单表增删改查）

将生成的文件拷贝到工程

### 8.5.4手动定义商品查询mapper

针对综合查询mapeper，一般情况会有关联查询，建议自定义mapper

Sql语句：

select \* from items where name like '%手%'

#### 8.5.4.1 ItemsMapperCustomer.xml

<?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=*"com.wbs.mapper.ItemsMapperCustomer"*>

<!-- 定义商品查询的SQL片段 -->

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

<!-- 使用动态SQL：通过if判断，满足条件的进行拼接 -->

<!-- 商品的查询条件是通过ItemsQueryCustomer中的ItemsCustomer属性传递 -->

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

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

items.name like '%${itemsCustomer.name}%'

</if>

</if>

</sql>

<!-- 商品的列表查询 parameterType:建议传入包装对象（包装了商品信息）

resultType建议使用扩展对象ItemsCustomer

-->

<select id=*"findItemsList"* resultType=*"com.wbs.domain.ItemsCustomer"* parameterType=*"com.wbs.domain.ItemsQueryVo"*>

select \* from items

<where>

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

</where>

</select>

</mapper>

#### 8.5.4.12ItemsMapperCustomer.java

**public** **interface** ItemsMapperCustomer {

//查询商品的列表

**public** List<ItemsCustomer> findItemsList(ItemsQueryVo itemsQueryVo) **throws** Exception;

}

## 8.6整合service

让spring管理service

### 8.6.1定义service接口

//商品管理的service

**public** **interface** ItemsService {

//商品的查询列表

**public** List<ItemsCustomer> findItemsList(ItemsQueryVo itemsQueryVo) **throws** Exception;

}

接口的实现类：

**public** **class** ItemsServiceImpl **implements** ItemsService {

@Autowired

**private** ItemsMapperCustomer itemsMapperCustomer;

@Override

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

//通过ItemsMapperCustomer查询数据

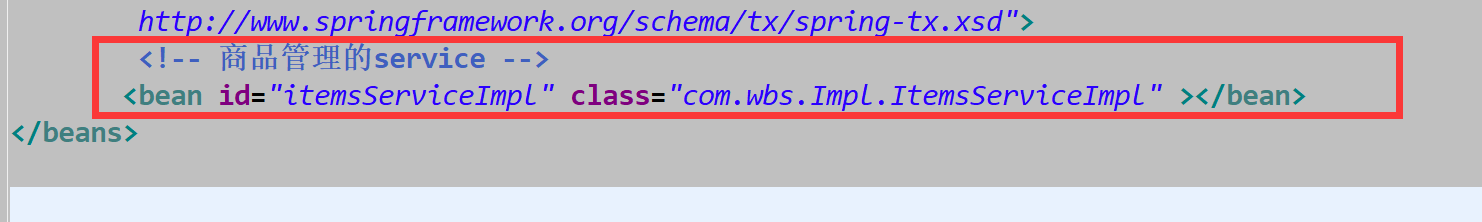
**return** itemsMapperCustomer.findItemsList(itemsQueryVo);

}

}

### 8.6.2在spring容器中配置service

创建applicationContext-service.xml,在这个文件中配置service接口



### 8.6.3事务控制

applicationContext-transaction.xml

<!-- 事务管理

对mybatis操作数据库的事务控制，spring使用jdbc的事务控制类

-->

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

<!-- 数据源：ref实在applicationContext-dao.xml中配置了 -->

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

</bean>

<!-- 通知 -->

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

<tx:attributes>

<!-- 传播行为 REQUIRED:必须的；-->

<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 -->

<aop:config>

<aop:advisor advice-ref=*"txAdvice"* pointcut=*"execution(\* com.wbs.Impl.\*.\*(..))"*/>

</aop:config>

</beans>

## 整合springmvc

### 8.1创建springmvc.xml文件

配置处理映射器、处理适配器、处理映射器、视图解析器

<!-- 可以扫描controller、service、

这里让组建扫描controller

-->

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

<!--mvc:annotation-driven默认加载很多的参数绑定方法，比如json转换解析器就默认加载，

如果使用mvc:annotation-driven，

就不需要配置上边的额两种RequestMappingHandlerMapping和RequestMappingHandlerAdapter

-->

<mvc:annotation-driven/>

<!-- 配置视图解析器

解析jsp视图，默认使用jstl标签，所以classpath下边必须要有jstl的包

-->

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

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

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

</bean>

</beans>

### 8.2 配置前端控制器

<!-- 配置前端控制器 -->

<servlet>

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

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

<!-- contextConfigLocation配置spring加载的配置文件（配置处理器映射器、适配器等）

如果不配置contextConfigLocation，默认加载的是/WEB-INF/servlet名称-servlet.xml(springmvc-servlet.xml)

-->

<init-param>

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

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

</init-param>

</servlet>

<servlet-mapping>

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

<!--方法1； \*.action 访问以.action结尾的由DispatcherServlet进行解析

方法2；/ 所有访问的action都由DispatcherServlet进行解析，对于静态的文件不让DispatcherServlet进行解析

使用此种方法可以实现RESTful风俗的url

方法3；/\* 这样配置不对，使用这种配置，最终要转发到jsp页面时候，仍然由DispatcherServlet解析jsp地址,

不能由jsp页面找到Handler,会 报错

-->

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

</servlet-mapping>

### 8.3 编写Controller(Handler)

//商品的Controller

@Controller

**public** **class** ItemsController {

@Autowired

**private** ItemsService itemsService;

//商品的查询

@RequestMapping("/queryItems")

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

//调用service查找数据库，查询商品列表

List<ItemsCustomer> itemslist=itemsService.findItemsList(**null**);

//返回ModelAndView

ModelAndView mav=**new** ModelAndView();

//相当于request的setAttribute()方法，在jsp页面通过itemslist获取数据

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

mav.addObject("list",itemslist);

request.setAttribute("aa", "hello");

//指定视图

mav.setViewName("/WEB-INF/jsp/items/itemslist.jsp");

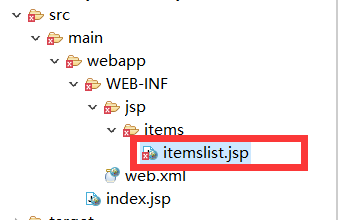
**return** mav;

}

//商品的修改

}

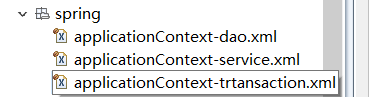
### 8.4 jsp



### 8.5 加载spring的容器

将mapper、service、controller加载到spring的容器中

将下边的文件加载到配置文件中：



建议使用通配符的方法，加载上边的文件，

在web.xml中添加spring容器的监听器

<web-app>

<!-- 加载spring的容器,使用通配符\* -->

<context-param>

<param-name>contextConfigLocation</param-name> <param-value>/WEB-INF/class/spring/applicationContext-\*.xml</param-value>

</context-param>

<listener>

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

</listener>

</web-app>

Pom.xml中统一配置spring的版本：

<properties> <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<spring.vision>4.0.2.RELEASE</spring.vision>

</properties>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${spring.vision}</version>

</dependency>

<dependency>

### 测试：

http://localhost:8080/spring-mybatis/queryItems.action

## 商品修改功能开发

需求：

操作流程：

1. 先进入商品查询列表页面
2. 点击修改，进入商品修改页面
   1. 页面中显示要显示的商品信息（从数据库查询）
   2. 根据商品的id来查询商品的信息
3. 在修改页面进行修改，修改后点击提交

### 9.1开发mapper

mapper:

根据id查询商品的信息

根据id更新items表的数据

不用开发了，因为逆向工程已经生成好了

### 9.1开发service

service接口:

根据id查询商品的信息

修改商品信息

//商品管理的service

**public** **interface** ItemsService {

//商品的查询列表

**public** List<ItemsCustomer> findItemsList(ItemsQueryVo itemsQueryVo) **throws** Exception;

//根据id查询商品信息

**public** ItemsCustomer findItemsById(Integer id) **throws** Exception;

//根据id修改商品信息(id:修改的商品的id，Itemscustomer:修改的商品的信息)

**public** **void** updateItems(**Integer** id,ItemsCustomer itemscustomer) **throws** Exception;

}

实现service接口的类：

**public** **class** ItemsServiceImpl **implements** ItemsService {

@Autowired

**private** ItemsMapperCustomer itemsMapperCustomer;

@Autowired

**private** ItemsMapper itemsMapper;

@Override

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

//通过ItemsMapperCustomer查询数据

**return** itemsMapperCustomer.findItemsList(itemsQueryVo);

}

@Override

**public** ItemsCustomer findItemsById(Integer id) **throws** Exception {

Items items=itemsMapper.selectByPrimaryKey(id);

//中间进行业务处理

//、、、、、中间做很多业务处理，比如判断是否过期等等

//返回ItemsCustomer

ItemsCustomer itemsCustomer=**new** ItemsCustomer();

//将items中的属性值拷贝到itemsCustomer

BeanUtils.*copyProperties*(items, itemsCustomer);

**return** itemsCustomer;

}

@Override

**public** **void** updateItems(Integer id, ItemsCustomer itemscustomer) **throws** Exception {

//修改商品

//添加业务的校验，通常在service中对关键的参数进行校验，在此校验id是否为空

//如果id为空，抛出异常

//更新商品信息,用updateByPrimaryKeyWithBLOBs根据id更新items表中的所有字段，包括大文本类型

//updateByPrimaryKeyWithBLOBs要求必须传入id，所以要set id

itemscustomer.setId(id);

itemsMapper.updateByPrimaryKeyWithBLOBs(itemscustomer);

}

}

### 9.1开发Controller

方法：

1. 商品信息修改展示页面
2. 商品信息修改提交方法

//商品的Controller

@Controller

**public** **class** ItemsController {

@Autowired

**private** ItemsService itemsService;

//商品的查询

@RequestMapping("/queryItems")

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

//调用service查找数据库，查询商品列表

List<ItemsCustomer> itemslist=itemsService.findItemsList(**null**);

//返回ModelAndView

ModelAndView mav=**new** ModelAndView();

//相当于request的setAttribute()方法，在jsp页面通过itemslist获取数据

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

mav.addObject("list",itemslist);

//指定视图

mav.setViewName("items/itemslist");

**return** mav;

}

//商品信息修改页面的展示

@RequestMapping("/editItems")

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

//调用service，根据id查询商品信息

ItemsCustomer itemsCustomer=itemsService.findItemsById(1);

//返回ModelAndView

ModelAndView mav=**new** ModelAndView();

//将商品信息放到Model中

mav.addObject("itemsCustomer",itemsCustomer);

//指定视图

mav.setViewName("items/editItems");

**return** mav;

}

//商品信息修改的提交

@RequestMapping("/editItemsSubmit")

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

//调用service更新商品信息，页面将商品信息传到此方法

//、、、、、、、、、

//返回ModelAndView

ModelAndView mav=**new** ModelAndView();

//指定视图

mav.setViewName("sucess");

**return** mav;

}

# 10 @requestMapping

* url映射

作用：定义Controller方法定义的Url，进行处理映射

* 窄化请求映射

//商品的Controller

@Controller

//为了对url进行分类管理，可以在这里定义根路径，最终访问路径为：根路径+子路径

//比如商品的查询变为：/items/queryItems.action

@RequestMapping("/imtes")

**public** **class** ItemsController {

* 限制http请求方法
  + 为了安全，对http的连接进行方法的限制

如果限制请求为POST，则用GET请求会报错：

 //商品的Controller

@Controller

//为了对url进行分类管理，可以在这里定义根路径，最终访问路径为：根路径+子路径

//比如商品的查询变为：/items/queryItems.action

//@RequestMapping("/items")

//限制http请求的方法,可以POST，可以GET

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

**public** **class** ItemsController {

# 11 Controller方法的返回值

## 1.

* 返回ModelAndView
  + 需要在方法结束的时候，定义ModelAndView，将model和view分别进行设置
* 返回String
  + 如果controller的方法返回时String，表示返回逻辑视图名
  + 真正的视图（jsp路径）=前缀+逻辑视图名+后缀

//商品的查询，修改不使用ModelAndView

@RequestMapping("/queryItems")

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

//调用service查找数据库，查询商品列表

List<ItemsCustomer> itemslist=itemsService.findItemsList(**null**);

//通过形参中的model将model数据传到页面,相当于mav.addObject("list",itemslist)方法

model.addAttribute("list",itemslist);

**return** "items/itemslist";

}

## 2 redirect重定向

商品修改提交后，重定向到商品的查询列表

Redirect特点：浏览器地址栏中的url会变化。修改提交的request数据无法传到重定向的地址中，因为redirect后重新进行request（requst无法进行共享）

当修改完信息之后，重定向到查询的页面：

//商品信息修改的提交

@RequestMapping("/editItemsSubmit")

**public** String editItemsSubmit() **throws** Exception{

//return "sucess";

//重定向到商品的查询列表

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

}

## forward

进行forward页面转发的时候，浏览器地址栏不变，request可以共享

//商品信息修改的提交

@RequestMapping("/editItemsSubmit")

**public** String editItemsSubmit( HttpServletRequest request) **throws** Exception{

//return "sucess";

//重定向到商品的查询列表

//return "redirect:queryItems.action";

//forward转发

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

}

## 4.返回void



@RequestMapping("/editItemsSubmit")

**public** String editItemsSubmit( HttpServletRequest request，HttpServletResponst response) **throws** Exception{}

下边的Controller是对上边的整体的总结：

//商品的Controller

@Controller

//为了对url进行分类管理，可以在这里定义根路径，最终访问路径为：根路径+子路径

//比如商品的查询变为：/items/queryItems.action

//@RequestMapping("/items")

//限制http请求的方法,可以POST，可以GET

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

**public** **class** ItemsController {

@Autowired

**private** ItemsService itemsService;

/\* //商品的查询

@RequestMapping("/queryItems")

public ModelAndView queryItems() throws Exception {

//调用service查找数据库，查询商品列表

List<ItemsCustomer> itemslist=itemsService.findItemsList(null);

//返回ModelAndView

ModelAndView mav=new ModelAndView();

//相当于request的setAttribute()方法，在jsp页面通过itemslist获取数据

System.out.println(itemslist);

mav.addObject("list",itemslist);

//指定视图

mav.setViewName("items/itemslist");

return mav;

}\*/

//商品的查询，修改不适用ModelAndView

@RequestMapping("/queryItems")

**public** ModelAndView queryItems(HttpServletRequest request) **throws** Exception {

//调用service查找数据库，查询商品列表

List<ItemsCustomer> itemslist=itemsService.findItemsList(**null**);

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

modelAndView.addObject("list",itemslist);

modelAndView.setViewName("items/itemslist");

**return** modelAndView;

}

//商品信息修改页面的展示

@RequestMapping("/editItems")

//@RequestParam里面指定request传入参数名称和形参进行绑定

//required指定参数是否必须要传入

//通过defaultValue可以设置默认值，如果id参数没有传入，将默认值和形参进行绑定

**public** String editItems(Model model,@RequestParam(value="id",required=**true**,defaultValue="") Integer items\_id) **throws** Exception{

//调用service，根据id查询商品信息

ItemsCustomer itemsCustomer=itemsService.findItemsById(items\_id);

//通过形参中的model将model数据传到页面,相当于mav.addObject("list",itemslist)方法

model.addAttribute("itemsCustomer",itemsCustomer);

**return** "items/editItems";

}

/\*//商品信息修改的提交

@RequestMapping("/editItemsSubmit")

public ModelAndView editItemsSubmit() throws Exception{

//调用service更新商品信息，页面将商品信息传到此方法

//、、、、、、、、、

//返回ModelAndView

ModelAndView mav=new ModelAndView();

//指定视图

mav.setViewName("sucess");

return mav;

}\*/

/\* //商品信息修改的提交

@RequestMapping("/editItemsSubmit")

public String editItemsSubmit() throws Exception{

//return "sucess";

//重定向到商品的查询列表

//return "redirect:queryItems.action";

//forward转发

return "forward:queryItems.action";

}\*/

//商品信息修改的提交

@RequestMapping("/editItemsSubmit")

**public** String editItemsSubmit( HttpServletRequest request) **throws** Exception{

//return "sucess";

//重定向到商品的查询列表

//return "redirect:queryItems.action";

//forward转发

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

}

}

# 14 参数绑定 14.1 spring的参数绑定过程

从客户端获取key/value数据，经过参数绑定，将key/value绑定到controller方法的形参上。

Springmvc接受页面提交的数据是通过方法形参来接受，而不是在controller类定义成员变量接收。

客户端请求

Key/value

处理器适配器调用springmvc提供的参数绑定组件将key/value转换成controller的方法形参。

参数绑定组件：springmvc在早期的版本使用PropertyEdit(只能将字符串转成Java对象)，**后期使用convertr(进行任意类型的转换)**

**Springmvc提供了很多converter(转换器)，在特殊情况下需要自定义converter；**

**比如对日期数据绑定需要自定义controller；**

Controller（形参）

## 14.2 默认支持的类型

直接在Controller方法的形参上定义下边类型的对象，就可以使用这些对象，在参数绑定的过程中，如果遇到下边的类型，直接进行绑定。

### 14.2.1 HttpServletRequest

通过request对象获取请求信息

### 14.2.2 HttpServletResponse

通过response获取响应信息

### 14.2.3 HttpSession

通过session对象得到session中存放的对象

### 14.2.4 Model/ModelMap

model是一个接口，modelMap是一个接口实现。

作用:将model数据填充到request域。

## 14.3 简单类型

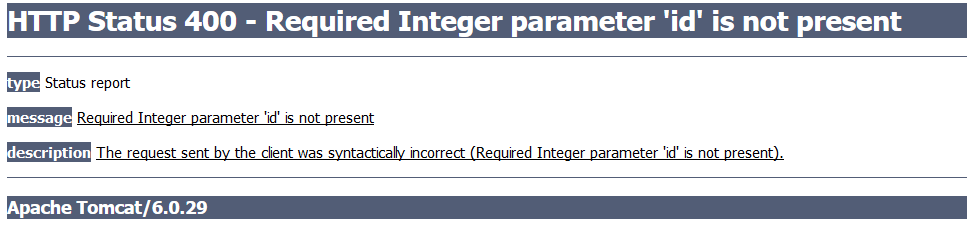
* 整形Integer
  + public String editItems(Model model,Integer id) throws Exception{}
* 字符串
* 单精度/双精度
* 布尔型
  + public String editItems(Model model,Integer id,Boolean status) throws Exception{}
  + 请求url：<http://localhost:8080/spring-mybatis/items/queryItems.action?id=1&status=false>
* 绑定的几种方法：

1. editItems(Modelmodel, Integer id)
2. editItems(Modelmodel,@RequestParam(value="id") Integer items\_id)
3. editItems(Modelmodel,@RequestParam(value="id",required=**true**) Integer items\_id)
4. editItems(Modelmodel,@RequestParam(value="id",required=**true**,defaultValue="") Integer items\_id)

使用@RequestParam对简单类型的参数进行绑定

如果不适用@RequestParam，要求request传入的参数名称和Controller方法的形参名一致，方可绑定成功。

如果使用@RequestParam，不用限制request传入的参数名称和controller方法的形参名称一致。



如果遇到上边的情况，肯定是id参数没有指定，即在@RequestParam中设置了required=true,而没有给此id传入任何值。

//商品信息修改页面的展示

@RequestMapping("/editItems")

//@RequestParam里面指定request传入参数名称和形参进行绑定

//required指定参数是否必须要传入

//通过defaultValue可以设置默认值，如果id参数没有传入，将默认值和形参进行绑定

**public** String editItems(Model model,@RequestParam(value="id",required=**true**,defaultValue="") Integer items\_id) **throws** Exception{

//调用service，根据id查询商品信息

ItemsCustomer itemsCustomer=itemsService.findItemsById(items\_id);

//通过形参中的model将model数据传到页面,相当于mav.addObject("list",itemslist)方法

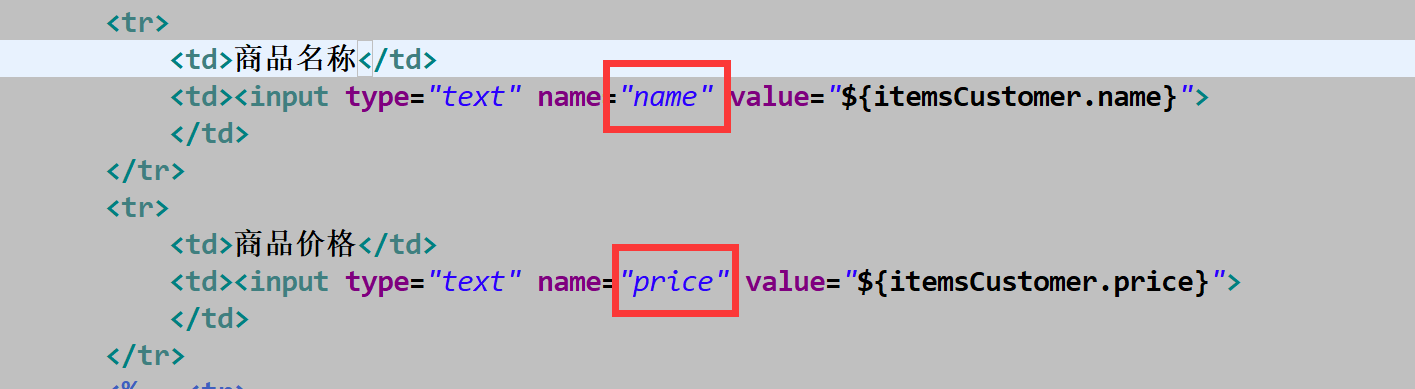
model.addAttribute("itemsCustomer",itemsCustomer);

**return** "items/editItems";

}

## 14.4 绑定POJO

* 页面中input的name和Controller的POJO形参中的属性名称一致，将页面中的数据绑定到POJO。
* POJO的绑定和简单类型的绑定互不影响，当形参中有editItems(Modelmodel, Integer id,String details,ItemsCustoemr itemsCustomer)的时候，String的deteils和POJO（ItemsCustomer）中的details属性是互不影响的，即只要页面传来有<input name=”details”这个属性,就会把details与形参进行绑定。
* 页面定义：



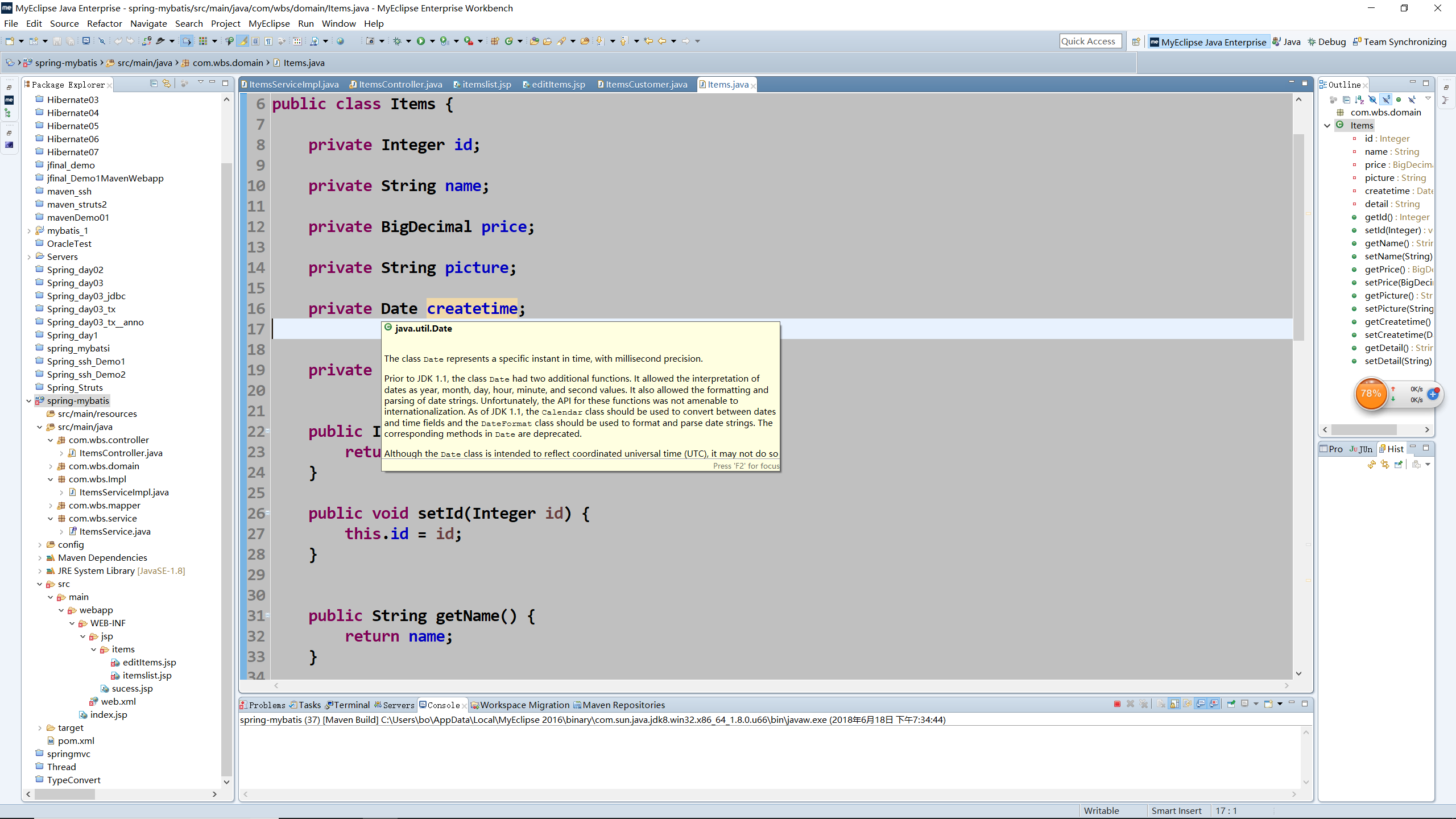
Controller的POJO的定义：



## 14.5 自定义参数绑定实现日期类型的绑定

对于Controller中形参中POJO对象，如果属性中有日期类型，需要自定义参数绑定。

将请求的日期的数据串转换成日期类型，要转换的日期类型和POJO中的属性的类型保持一致，所以自定义参数绑定将日期传转换成java.utils.Data类型



需要向Springmvc.xml中配置，在处理器适配器中注入自定义的参数绑定 组件

<!--mvc:annotation-driven默认加载很多的参数绑定方法，比如json转换解析器就默认加载，

如果使用mvc:annotation-driven，

就不需要配置上边的额两种RequestMappingHandlerMapping和RequestMappingHandlerAdapter

-->

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

<!-- 自定义参数绑定 -->

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

<!-- 转换器，自定义的，在下面写出了日期转换类 -->

<property name=*"converters"*>

<list>

<!-- 日期类型的转换 -->

<bean class=*"com.wbs.converter.DateConverter"*></bean>

</list>

</property>

</bean>

//日期转换类

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

@Override

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

//将日期串转换成日期类型 （格式是：yyyy-MM-dd HH:mm:ss）

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

**try** {

//转换成功，直接返回

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

} **catch** (ParseException e) {

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

e.printStackTrace();

}

//如果参数绑定失败，返回空

**return** **null**;

}

}



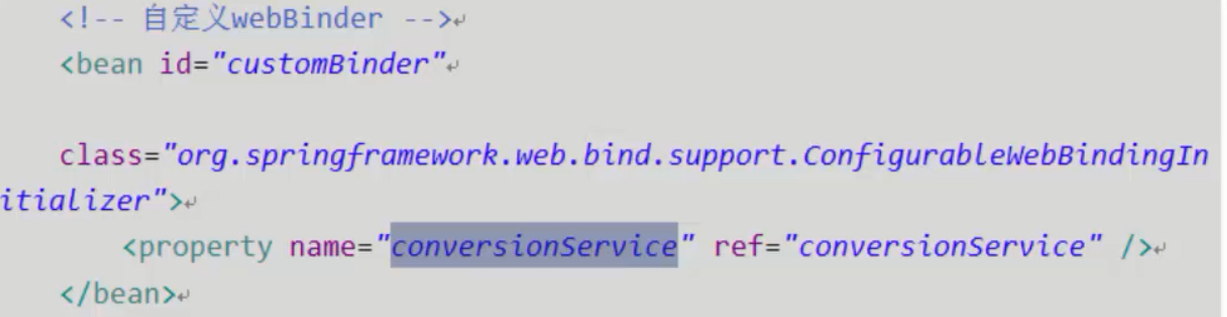
## 14.6配置方式2

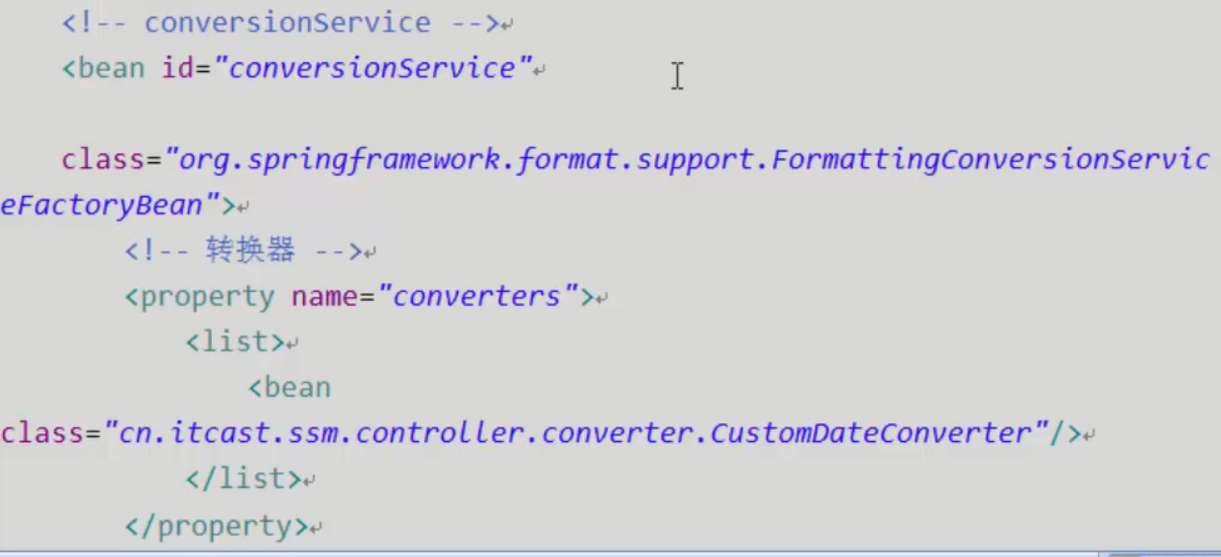
向下边的适配器中注入customBinder:



webBindingInitializer是对外提供的参数绑定的接口。

向下边的注入自定义的conversionService:





# 15乱码问题

## 15.1 POST乱码

在web.xml中添加post乱码的过滤器

<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>

## 15.1 get乱码

Get乱码有两种解决方法：

1. 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=”200000” port=”8080” protocol=”HTTP/1.1” redirectPort=”8443”/>

1. 另一种方法对参数重新进行编码

String Username= new

String(request.getParameter(“username”).getBytes(“ISO-8859-1”),”utf-8”)

Iso-8859-1 是tomcat默认编码，需要将tomcat编码后的内容安urf-8进行编码。

# 16 springmvc和struts2的区别

1. springmvvc是基于方法开发的，而struts2是基于类开发的
2. springmvc将url和controller方法进行映射，映射成功之后，springmvc会生成一个Handler对象，对象中只包含一个method。
3. 方法执行结束，形参数据销毁。
4. Springmvc可以进行单例开发，并且建议单例开发，但是struts2不行，因为struts2通过类的成员变量接受参数，无法使用单例，只能使用多例。
5. Springmvc的开发类似service的开发，更符合编码习惯。
6. Struts2经过实际测试，速度慢在于使用了struts标签，如果使用struts2，建议使用jstl。

# Springmvc高级知识

复习：

* springmvc框架：

DispetcherServlet：前端控制器，接受请求（request），进行响应（response）

Handlermapping:处理器映射器，根据url查找Handler，（可以通过xml配置、注解方式）

HandlerAdapter：处理器适配器，根据特定规则去执行Handler，编写Handler按照适配器的要求编写。

Handler：处理器（后端控制器），需要程序员编写，**常用注解的方式编写。**

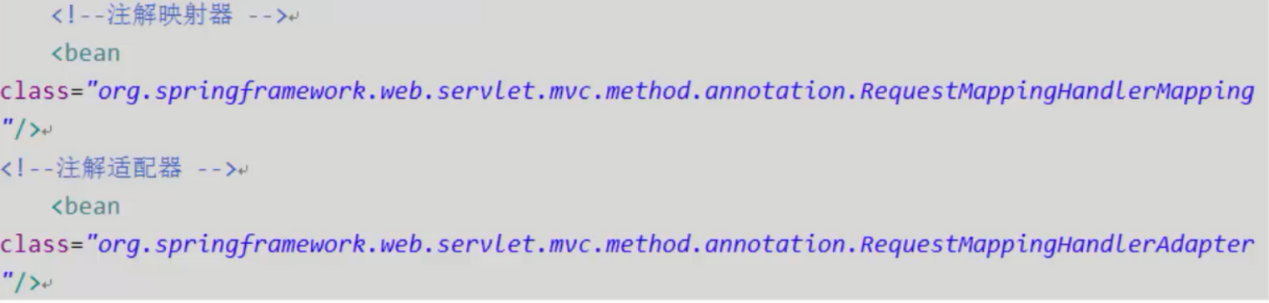
handler处理器执行后是一个**ModelAndView，具体开发时候，Handler返回的类型包括**：ModelAndView、String（逻辑视图名）、void（在Handler的形参中添加request、response，类似原始的servlet，但是注意：此种方式可以通过指定response响应的结果类型，事项ＪＳＯＮ数据的输出）

view resolver 视图解析器：根据逻辑视图名生成真正的视图名（在springmvc中使用view对象来表示）

view视图：jsp页面，仅仅是视图的展示，没有业务逻辑。

* 注解开发

使用注解的方式的处理器映射器和处理器适配器



在实际开发中，使用<mvc:annotation-driven>代替上边处理器映射器和处理器适配器。

@Controller：必须要加。作用：标识此类是一个Handler处理器。

@RequestMapping :必须加，

作用：

1. url和Handler的方法进行映射
2. 可以窄化请求映射，设置Handler类的根路径，url=根路径+子路径
3. 可以限制http请求的方法

映射成功后，springmvc会生成一个Handler对象，对象中只包含一个映射成功的method。

注解开发中参数绑定：

将request请求过来的key/value的数据（理解为一个串），通过转换（参数绑定的一部分），将key/value串转换成形参，将转换后的结果传递给形参（整个参数绑定过程）。

Springmvc所支持的参数绑定：

1. 默认支持很多类型：HttpServletRequest、response、session、model/modelMap(将模型数据填充到request域)。
2. 支持简单的数据类型：Integer、String、Date、BooLean
   1. 只要保持request请求的参数名和形参名称一致，就会自动绑定成功，如果request请求的参数名和形参名称不一致，可以使用@Requestparam来指定请求参数名，@Requestparam加在形参的前边。
3. 支持POJO类型
   1. 只要保证request提交的参数名和POJO中的属性名称一致，就会自动将request请求的参数设置到POJO的属性中。
   2. 形参中既有POJO，又有简单类型，参数绑定是互不影响的。
4. 自定义类型绑定：
   1. 日期类型绑定自定义
      1. 定义converter<原类型,目标类型>接口的实现类，比如：Converter<String,Date>是将日期数据串转换成java中的日期类型

注意：要转换的目标类型一定要和接收的POJO的属性的类型一致

* 1. 将定义的Converter实现类注入到处理器适配器中

springmvc 、struts2区别：

springmvc是单例的，struts2是多例开发。

springmvc是面向方法开发（更接近service接口的开发方式），struts2是面向类开发。