



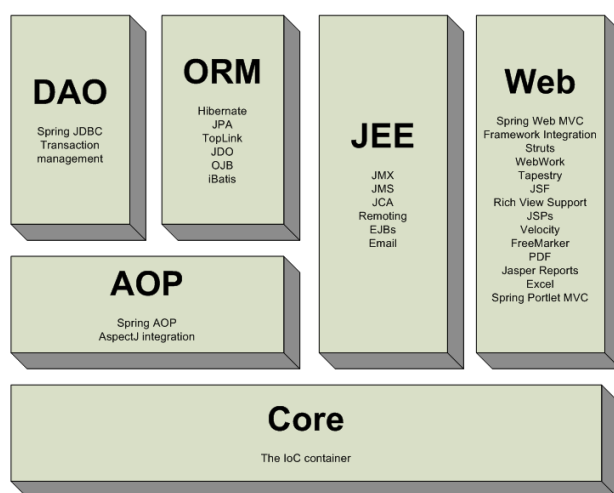
Spring web mvc 框架课程

讲师：传智.燕青

1 SpringMVC 架构

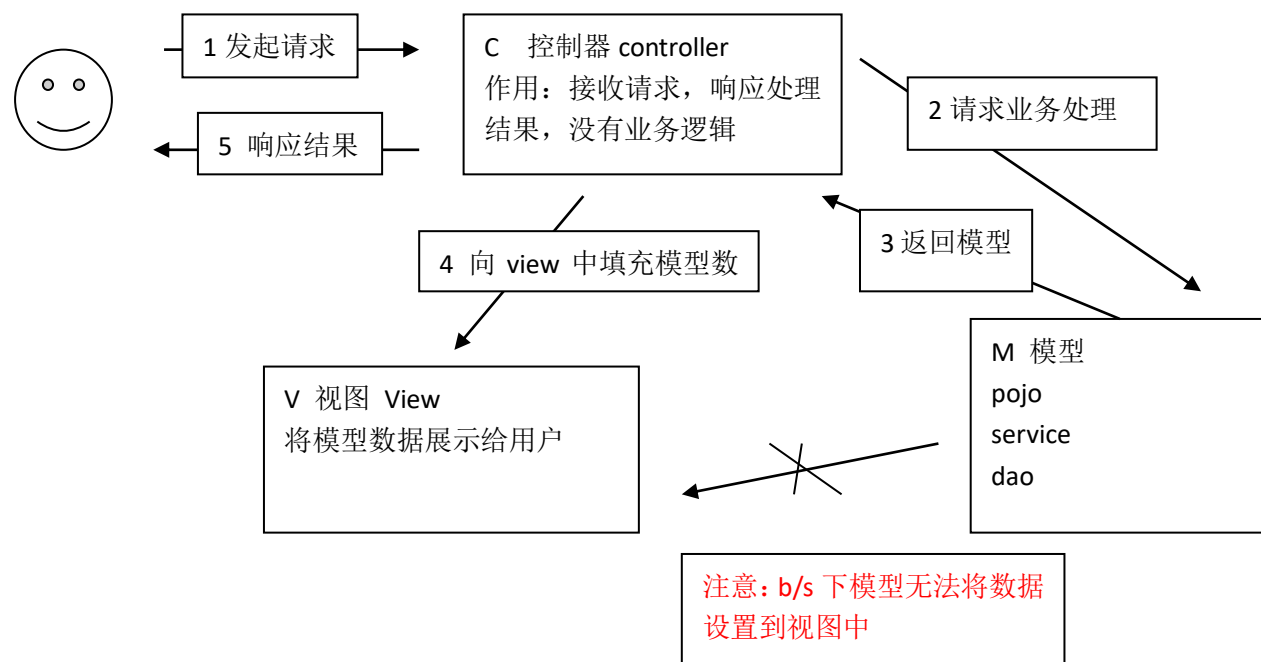
1.1 Spring web mvc 介绍

Spring web mvc 和 Struts2 都属于表现层的框架,它是 Spring 框架的一部分,我们可以从 Spring 的整体结构中看得出来:



1.2 Web MVC

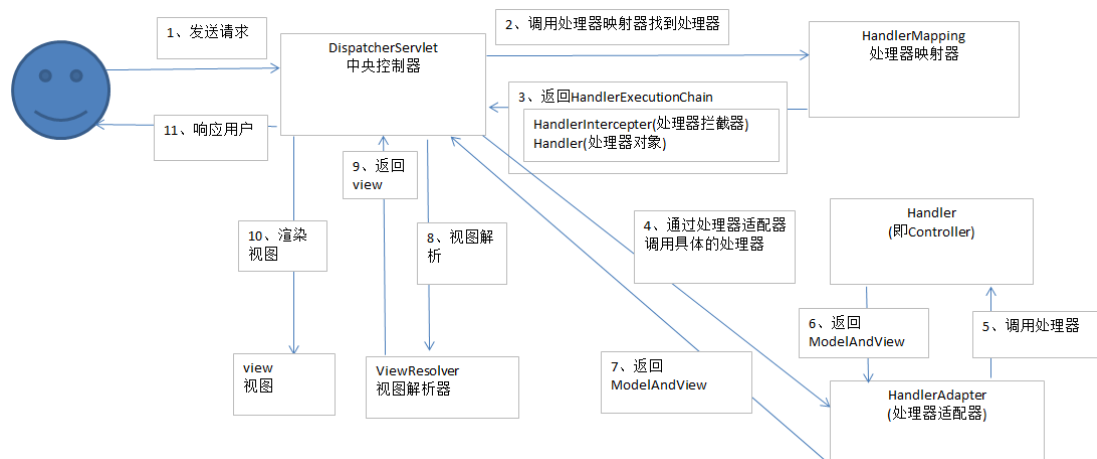
mvc 设计模式在 b/s 系统下应用:



- 1、用户发起 request 请求至控制器(Controller)
控制接收用户请求的数据, 委托给模型进行处理
- 2、控制器通过模型(Model)处理数据并得到处理结果
模型通常是指业务逻辑
- 3、模型处理结果返回给控制器
- 4、控制器将模型数据在视图(View)中展示
web 中模型无法将数据直接在视图上显示, 需要通过控制器完成。如果在 C/S 应用中模型是可将数据在视图中展示的。
- 5、控制器将视图 response 响应给用户
通过视图展示给用户要的数据或处理结果。

1.3 Spring web mvc 架构

1.3.1 架构图



1.3.2 架构流程

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

1.3.3 组件说明

以下组件通常使用框架提供实现：

◆ DispatcherServlet：前端控制器

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

◆ HandlerMapping：处理器映射器

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

◆ Handler：处理器

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

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

◆ HandlerAdapter：处理器适配器

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

◆ View Resolver：视图解析器

View Resolver负责将处理结果生成View视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。springmvc框架提供了很多的View视图类型，包括：jstlView、freemarkerView、pdfView等。

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

2 商品订单业务说明



本教程在通过商品订单业务学习使用 springmvc 进行功能开发。

2.1 业务流程

- 1、管理员维护商品信息
- 2、用户挑选商品，购买，创建订单

2.2 数据库环境

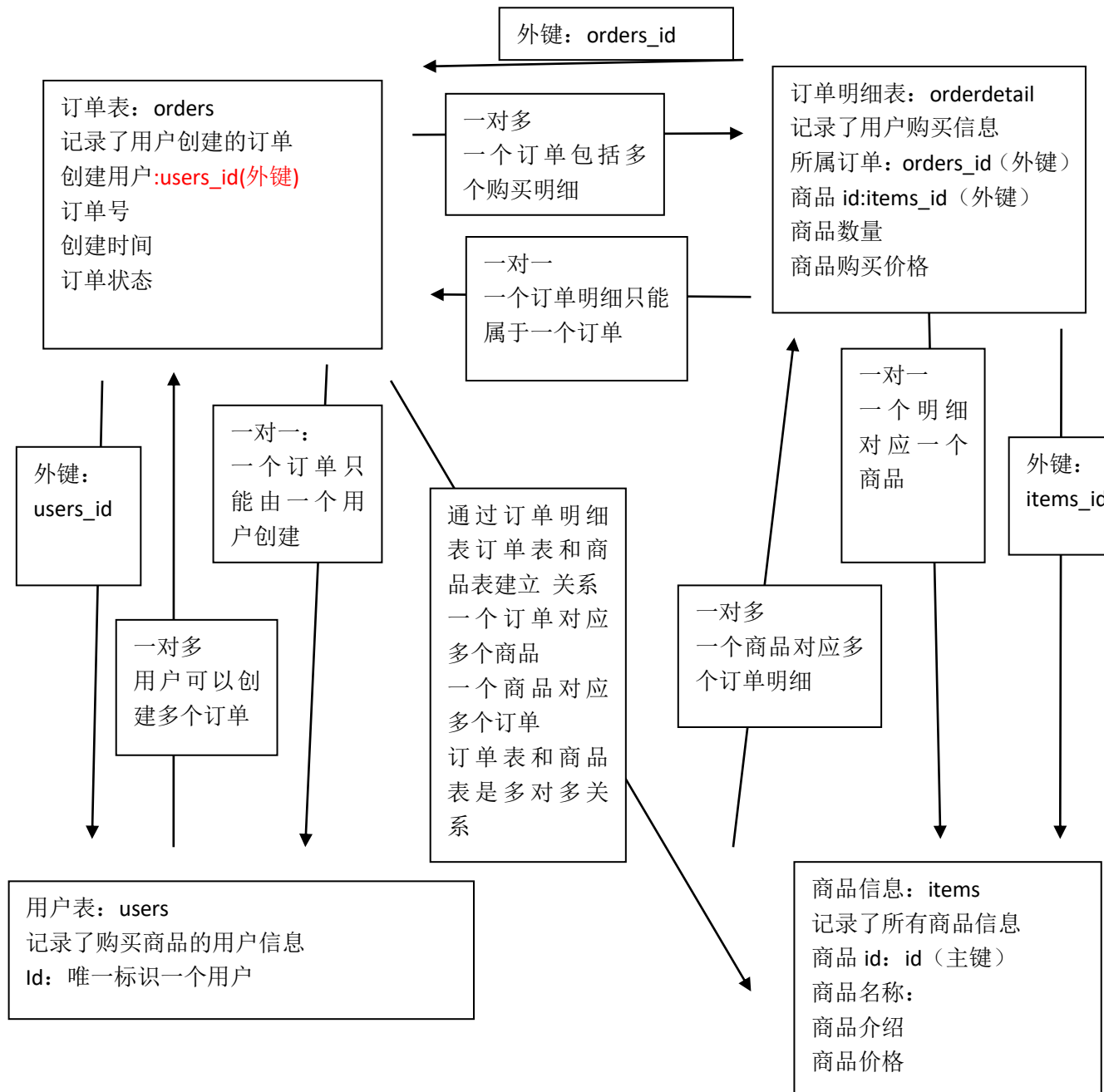
先导入 sql_table.sql，再导入 sql_data.sql 脚本：

 sql_data.sql
 sql_table.sql

如下：



2.3 商品订单数据模型



3 SpringMVC 入门

3.1 需求

实现商品查询列表功能。

3.2 开发环境准备

本教程使用 Eclipse+tomcat7 开发

详细参考“Eclipse 开发环境配置-indigo.docx”文档

3.3 第一步:建立一个 Web 项目

在 eclipse 下创建动态 web 工程 springmvc_first。

3.4 第二步:导入 spring3.2.0 的 jar 包



3.5 第三步：前端控制器配置

在 WEB-INF\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>
<init-param>
<param-name>contextConfigLocation</param-name>
<param-value>classpath:springmvc.xml</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>springmvc</servlet-name>
<url-pattern>*.action</url-pattern>
</servlet-mapping>
```

load-on-startup: 表示servlet随服务启动;

url-pattern: *.action的请求交给DispatcherServlet处理。

contextConfigLocation: 指定springmvc配置的加载位置，如果不指定则默认加载WEB-INF/[DispatcherServlet 的Servlet 名字]-servlet.xml。

3.5.1 Servlet 拦截方式

1、拦截固定后缀的url，比如设置为 *.do、*.action， 例如： /user/add.action
此方法最简单，不会导致静态资源（jpg, js, css）被拦截。

2、拦截所有，设置为/，例如： /user/add /user/add.action
此方法可以实现REST风格的url，很多互联网类型的应用使用这种方式。
但是此方法会导致静态文件（jpg, js, css）被拦截后不能正常显示。需要特殊处理。

3、拦截所有，设置为/*，此设置方法错误，因为请求到Action，当action转到jsp时再次被拦截，提示不能根据jsp路径mapping成功。

3.6 第四步：springmvc 配置文件

Springmvc默认加载WEB-INF/[前端控制器的名字]-servlet.xml，也可以在前端控制器定义处指定加载的配置文件，如下：

```
<init-param>
<param-name>contextConfigLocation</param-name>
<param-value>classpath:springmvc.xml</param-value>
</init-param>
```

如上代码，通过contextConfigLocation加载classpath下的springmvc.xml配置文件。

3.7 第五步：配置处理器适配器

在 springmvc.xml 文件配置如下：

```
<bean
    class="org.springframework.web.servlet.mvc.SimpleController
HandlerAdapter"/>
```

SimpleControllerHandlerAdapter：即简单控制器处理适配器，所有实现了org.springframework.web.servlet.mvc.Controller 接口的Bean作为Springmvc的后端控制器。

3.8 第六步：处理器开发

```
public class ItemList1 implements Controller {

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

        //商品列表
        List<Items> itemList = 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准备填充数据、设置视图
ModelAndView modelAndView = new ModelAndView();

//填充数据
modelAndView.addObject("itemsList", itemsList);
//视图
modelAndView.setViewName("order/itemsList");

return modelAndView;
}

}
```

org.springframework.web.servlet.mvc.Controller: 处理器必须实现Controller 接口。

ModelAndView: 包含了模型数据及逻辑视图名

3.9 第七步：配置处理器映射器

在 springmvc.xml 文件配置如下：

```
<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
">

<!-- 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" />
```

BeanNameUrlHandlerMapping: 表示将定义的Bean名字作为请求的url, 需要将编写的controller在spring容器中进行配置, 且指定bean的name为请求的url, 且必须以.action结尾。

3.10 第八步: 处理器配置

在 springmvc.xml 文件配置如下:

```
<!-- controller配置 -->
<bean name="/items1.action" id="itemList1"
class="cn.itcast.springmvc.controller.first.ItemList1"/>
```

name="/items1.action": 前边配置的处理器映射器为BeanNameUrlHandlerMapping, 如果请求的URL 为“上下文/items1.action”将会成功映射到ItemList1控制器。

3.11 第九步：配置视图解析器

在 `springmvc.xml` 文件配置如下：

```
<!-- ViewResolver -->
<bean
class="org.springframework.web.servlet.view.InternalResourceViewResol
ver">
<property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
<property name="prefix" value="/WEB-INF/jsp/" />
<property name="suffix" value=".jsp" />
</bean>
```

InternalResourceViewResolver：支持JSP视图解析

viewClass: JstlView表示JSP模板页面需要使用JSTL标签库，所以classpath中必须包含jstl的相关jar包；

prefix 和suffix：查找视图页面的前缀和后缀，最终视图的地址为：

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

3.12 第十步：视图开发

创建`/WEB-INF/jsp/order/itemsList.jsp` 视图页面：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>查询商品列表</title>
</head>
```

```
<body>
商品列表:
<table width="100%" border=1>
<tr>
    <td>商品名称</td>
    <td>商品价格</td>
    <td>商品描述</td>
</tr>
<c:forEach items="${itemsList }" var="item">
<tr>
    <td>${item.name }</td>
    <td>${item.price }</td>
    <td>${item.detail }</td>
</tr>
</c:forEach>

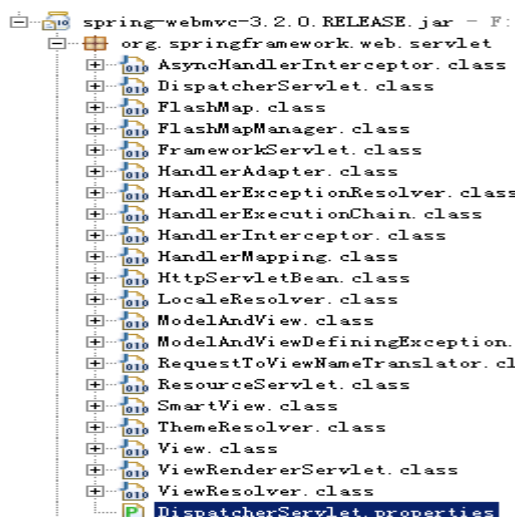
</table>
</body>
</html>
```

3.13 第十一步：部署在 tomcat 测试

通过请求：http://localhost:8080/springmvc_first/items1.action，如果页面输出商品列表就表明我们成功了！

3.14 DispatcherServlet

DispatcherServlet 作为 springmvc 的中央调度器存在，DispatcherServlet 创建时会默认从 DispatcherServlet.properties 文件加载 springmvc 所用的各各组件，如果在 springmvc.xml 中配置了组件则以 springmvc.xml 中配置的为准，DispatcherServlet 的存在降低了 springmvc 各各组件之间的耦合度。



3.15 HandlerMapping 处理器映射器

HandlerMapping 负责根据 request 请求找到对应的 Handler 处理器及 Interceptor 拦截器，将它们封装在 HandlerExecutionChain 对象中给前端控制器返回。

3.15.1 BeanNameUrlHandlerMapping

BeanNameUrl 处理器映射器，根据请求的 url 与 spring 容器中定义的 bean 的 name 进行匹配，从而从 spring 容器中找到 bean 实例。

```
<!--beanName Url映射器 -->
<bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
```

3.15.2 SimpleUrlHandlerMapping

simpleUrlHandlerMapping 是 BeanNameUrlHandlerMapping 的增强版本，它可以将 url 和处理器 bean 的 id 进行统一映射配置。

```
<!--简单url映射 -->
<bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"
```

```
>
    <property name="mappings">
        <props>
            <prop key="/items1.action">controller的bean id</prop>
            <prop key="/items2.action">controller的bean id</prop>
        </props>
    </property>
</bean>
```

3.16 HandlerAdapter 处理器适配器

HandlerAdapter 会根据适配器接口对后端控制器进行包装（适配），包装后即可对处理器进行执行，通过扩展处理器适配器可以执行多种类型的处理器，这里使用了适配器设计模式。

3.16.1 SimpleControllerHandlerAdapter

SimpleControllerHandlerAdapter简单控制器处理器适配器，所有实现了org.springframework.web.servlet.mvc.Controller 接口的Bean通过此适配器进行适配、执行。

适配器配置如下：

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

3.16.2 HttpRequestHandlerAdapter

HttpRequestHandlerAdapter，http请求处理器适配器，所有实现了org.springframework.web.HttpRequestHandler 接口的Bean通过此适配器进行适配、执行。

适配器配置如下：

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

Controller实现如下：


```
public class ItemList2 implements HttpRequestHandler {

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

        // 商品列表
        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("iphone5 苹果手机!");

        itemsList.add(items_1);
        itemsList.add(items_2);

        // 填充数据
        request.setAttribute("itemsList", itemsList);

        // 视图

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

    }

}
```

从上边可以看出此适配器的handleRequest方法没有返回ModelAndView，可通过response修改定义响应内容，比如返回json数据：

```
response.setCharacterEncoding("utf-8");
response.setContentType("application/json;charset=utf-8");
response.getWriter().write("json串");
```

3.17 注解映射器和适配器

3.17.1 Controller 的代码

```
@Controller
public class ItemList3 {

    @RequestMapping("/queryItem.action")
    public ModelAndView queryItem() {
        // 商品列表
        List<Items> itemList = 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!");

        itemList.add(items_1);
        itemList.add(items_2);

        // 创建modelAndView准备填充数据、设置视图
        ModelAndView modelAndView = new ModelAndView();

        // 填充数据
        modelAndView.addObject("itemsList", itemList);
        // 视图
        modelAndView.setViewName("order/itemsList");

        return modelAndView;
    }
}
```

3.17.2 组件扫描器

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

```
<!-- 扫描controller注解,多个包中间使用半角逗号分隔 -->
<context:component-scan
base-package="cn.itcast.springmvc.controller.first"/>
```

3.17.3 RequestMappingHandlerMapping

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

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

配置如下：

```
<!--注解映射器 -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>
```

注解描述：

@RequestMapping：定义请求 url 到处理器功能方法的映射

3.17.4 RequestMappingHandlerAdapter

注解式处理器适配器，对标记 `@RequestMapping` 的方法进行适配。

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

配置如下：

```
<!--注解适配器 -->
```

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

3.17.5 <mvc:annotation-driven>

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

3.18 springmvc 处理流程源码分析

1. 用户发送请求到 DispatcherServlet 前端控制器
2. DispatcherServlet 调用 HandlerMapping（处理器映射器）根据 url 查找 Handler

```
// Determine handler for the current request.
mappedHandler = getHandler(processedRequest, false);

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. DispatcherServlet 调用 HandlerAdapter(处理器适配器)对 HandlerMapping 找到 Handler 进行包装、执行。HandlerAdapter 执行 Handler 完成后，返回了一个 ModelAndView(springmvc 封装对象)

DispatcherServlet 找一个合适的适配器：

```
// Determine handler adapter for the current request.
HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
```

适配器执行 Handler

```
// Actually invoke the handler.  
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
```

4. DispatcherServlet 拿着 ModelAndView 调用 ViewResolver（视图解析器）进行视图解析，解析完成后返回一个 View（很多不同视图类型的 View）

```
// Did the handler return a view to render?  
if (mv != null && !mv.wasCleared()) {  
    render(mv, request, response);  
}
```

视图解析：

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

5. DispatcherServlet 进行视图渲染，将 Model 中数据放到 request 域，在页面展示

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

将 model 数据放在 request 域：

```
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() + "'");  
            }  
        }  
    }  
}
```

4 整合 mybatis

为了更好的学习 springmvc 和 mybatis 整合开发的方法，需要将 springmvc 和 mybatis 进行整合。

整合目标：控制层采用 springmvc、持久层使用 mybatis 实现。

4.1 需求

实现商品查询列表，从 mysql 数据库查询商品信息。

4.2 jar 包

包括：spring（包括 springmvc）、mybatis、mybatis-spring 整合包、数据库驱动、第三方连接池。

参考：“mybatis 与 springmvc 整合全部 jar 包”目录

4.3 Dao

目标：

1、spring 管理 SqlSessionFactory、mapper

详细参考 mybatis 教程与 spring 整合章节。

4.3.1 db.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis
jdbc.username=XXXX
jdbc.password=XXXX
```

4.3.2 log4j.properties

```
# Global logging configuration, 建议开发环境中要用debug
log4j.rootLogger=DEBUG, stdout
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

4.3.3 sqlMapConfig.xml

在 classpath 下创建 *mybatis/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>

<!--使用自动扫描器时，mapper.xml文件如果和mapper.java接口在一个目录则此处不用
定义mappers -->
<mappers>
<package name="cn.itcast.ssm.mapper" />
</mappers>
</configuration>
```

4.3.4 applicationContext-dao.xml

配置数据源、事务管理，配置 SqlSessionFactory、mapper 扫描器。

```
<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 ">
<!-- 加载配置文件 -->
```

```
<context:property-placeholder location="classpath:db.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}"/>
    <property name="maxActive" value="30"/>
    <property name="maxIdle" value="5"/>
</bean>

<!-- 让spring管理sqlSessionFactory 使用mybatis和spring整合包中的 -->
<bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
    <!-- 数据库连接池 -->
    <property name="dataSource" ref="dataSource" />
    <!-- 加载mybatis的全局配置文件 -->
    <property name="configLocation"
value="classpath:mybatis/SqlMapConfig.xml" />
</bean>
<!-- mapper扫描器 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage"
value="cn.itcast.springmvc.mapper"></property>
<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
</bean>

</beans>
```

4.3.5 ItemsMapper.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="cn.itcast.ssm.mapper.ItemsMapper">

    <!-- sql片段 -->
    <!-- 商品查询条件 -->
```



```
<sql id="query_items_where">
    <if test="items!=null">
        <if test="items.name!=null and items.name!=''">
            and items.name like '%${items.name}%'
        </if>
    </if>
</sql>

<!-- 查询商品信息 -->
<select id="findItemsList" parameterType="queryVo"
resultType="items">
    select * from items
    <where>
        <include refid="query_items_where"/>
    </where>
</select>

</mapper>
```

4.3.6 ItemsMapper.java

```
public interface ItemsMapper {
    //商品列表
    public List<Items> findItemsList(QueryVo queryVo) throws Exception;
}
```

4.4 Service

目标:

- 1、Service 由 spring 管理
- 2、spring 对 Service 进行事务控制。

4.4.1 applicationContext-service.xml

配置 service 接口。

4.4.2 applicationContext-transaction.xml

配置事务管理器。

```
<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 ">

    <!-- 事务管理器 -->
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        >
        <!-- 数据源 -->
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- 通知 -->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <!-- 传播行为 -->
            <tx:method name="save*" propagation="REQUIRED"/>
            <tx:method name="insert*" propagation="REQUIRED"/>
            <tx:method name="delete*" 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:attributes>
    </tx:advice>
```

```
<!-- 切面 -->
<aop:config>
    <aop:advisor advice-ref="txAdvice"
        pointcut="execution(* cn.itcast.springmvc.service.impl.*(..))"/>
</aop:config>

</beans>
```

4.4.3 OrderService

```
public interface OrderService {

    //商品查询列表
    public List<Items> findItemsList(QueryVo queryVo) throws Exception;
}
```

```
@Autowired
private ItemsMapper itemsMapper;

@Override
public List<Items> findItemsList(QueryVo queryVo) throws Exception {
    //查询商品信息
    return itemsMapper.findItemsList(queryVo);
}
}
```

4.5 Action

4.5.1 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 ">

<!-- 扫描controller注解,多个包中间使用半角逗号分隔 -->
<context:component-scan base-package="cn.itcast.ssm.controller"/>

<!--注解映射器 -->
<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"/>

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

```

4.5.2 web.xml

加载 spring 容器，配置 springmvc 前置控制器。

```
<?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"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    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</display-name>

    <!-- 加载spring容器 -->
    <context-param>
        <param-name>contextConfigLocation</param-name>

        <param-value>/WEB-INF/classes/spring/applicationContext.xml,/WEB-INF
/classes/spring/applicationContext-*.xml</param-value>
    </context-param>
    <listener>

    <listener-class>org.springframework.web.context.ContextLoaderListene
r</listener-class>
    </listener>

    <!-- 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>

    <!-- springmvc的前端控制器 -->
    <servlet>
        <servlet-name>springmvc</servlet-name>

        <servlet-class>org.springframework.web.servlet.DispatcherServlet</se
rvlet-class>
```

```
    <!-- contextConfigLocation不是必须的， 如果不配置
contextConfigLocation, springmvc的配置文件默认在：WEB-INF/servlet的
name+"-servlet.xml" -->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring/springmvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>*.action</url-pattern>
</servlet-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>
```

4.5.3 OrderController

```
@Controller
public class OrderController {

    @Autowired
    private OrderService orderService;

    @RequestMapping("/queryItem.action")
    public ModelAndView queryItem() throws Exception {
        // 商品列表
        List<Items> itemList = orderService.findItemsList(null);

        // 创建modelAndView准备填充数据、设置视图
        ModelAndView modelAndView = new ModelAndView();

        // 填充数据
        modelAndView.addObject("itemsList", itemList);
    }
}
```

```
// 视图
modelAndView.setViewName("order/itemsList");

return modelAndView;
}
}
```

4.6 测试

http://localhost:8080/springmvc_mybatis/queryItem.action

5 注解开发-基础

5.1 需求

使用 springmvc+mybatis 架构实现商品信息维护。

5.2 商品修改

5.2.1 dao

使用逆向工程自动生成的代码：

ItemsMapper.java

ItemsMapper.xml

5.2.2 service

```
//根据id查询商品信息
public Items findItemById(int id) throws Exception;

//修改商品信息
public void saveItem(Items items) throws Exception;
```

5.2.3 controller

修改商品信息显示页面:

```
@RequestMapping(value="/editItem")
public String editItem(Model model, Integer id) throws Exception{

    //调用service查询商品信息
    Items item = itemService.findById(id);

    model.addAttribute("item", item);

    return "item/editItem";
}
```

修改商品信息提交:

```
//商品修改提交
@RequestMapping("/editItemSubmit")
public String editItemSubmit(Items items) throws Exception{

    System.out.println(items);

    itemService.saveItem(items);

    return "success";
}
```

5.2.4 页面

/WEB-INF/jsp/item/itemsList.jsp

/WEB-INF/jsp/item/editItem.jsp

5.3 @RequestMapping

通过 RequestMapping 注解可以定义不同的处理器映射规则。

5.3.1 URL 路径映射

`@RequestMapping(value="/item")`或`@RequestMapping("/item")`

value 的值是数组，可以将多个 url 映射到同一个方法

5.3.2 窄化请求映射

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

如下：

`@RequestMapping` 放在类名上边，设置请求前缀

`@Controller`

`@RequestMapping("/item")`

方法名上边设置请求映射 url：

`@RequestMapping` 放在方法名上边，如下：

`@RequestMapping("/queryItem")`

访问地址为：/item/queryItem

5.3.3 请求方法限定

◆ 限定 GET 方法

`@RequestMapping(method = RequestMethod.GET)`

如果通过 Post 访问则报错：

HTTP Status 405 - Request method 'POST' not supported

例如：

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

◆ 限定 POST 方法

`@RequestMapping(method = RequestMethod.POST)`

如果通过 Post 访问则报错：

HTTP Status 405 - Request method 'GET' not supported

◆ GET 和 POST 都可以

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

5.4 controller 方法返回值

5.4.1 返回 ModelAndView

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

5.4.2 返回 void

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

1、使用 request 转向页面，如下：

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

2、也可以通过 response 页面重定向：

```
response.sendRedirect("url")
```

3、也可以通过 response 指定响应结果，例如响应 json 数据如下：

```
response.setCharacterEncoding("utf-8");  
response.setContentType("application/json;charset=utf-8");  
response.getWriter().write("json 串");
```

5.4.3 返回字符串

5.4.3.1 逻辑视图名

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

//指定逻辑视图名，经过视图解析器解析为jsp物理路径：

```
/WEB-INF/jsp/item/editItem.jsp
```

```
return "item/editItem";
```

5.4.3.2 Redirect 重定向

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

```
//重定向到queryItem.action地址,request无法带过去  
return "redirect:queryItem.action";
```

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

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

5.4.3.3 forward 转发

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

```
//结果转发到editItem.action, request可以带过去  
return "forward:editItem.action";
```

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

5.5 参数绑定

处理器适配器在执行 Handler 之前需要把 http 请求的 key/value 数据绑定到 Handler 方法形参数上。

5.5.1 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。

5.5.1.1 HttpServletRequest

通过 request 对象获取请求信息

5.5.1.2 HttpServletResponse

通过 response 处理响应信息

5.5.1.3 HttpSession

通过 session 对象得到 session 中存放的对象

5.5.1.4 Model/ModelMap

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

```
//调用service查询商品信息
Items item = itemService.findById(id);
model.addAttribute("item", item);
```

页面通过\${item.XXXX}获取 item 对象的属性值。

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

5.5.2 参数绑定介绍

注解适配器对 RequestMapping 标记的方法进行适配，对方法中的形参会进行参数绑定，早期 springmvc 采用 PropertyEditor（属性编辑器）进行参数绑定将 request 请求的参数绑定到方法形参上，3.X 之后 springmvc 就开始使用 Converter 进行参数绑定。

5.5.3 简单类型

当请求的参数名称和处理器形参名称一致时会将请求参数与形参进行绑定。

5.5.3.1 整型

```
public String editItem(Model model, Integer id) throws Exception {  
}
```

5.5.3.2 字符串

例子略

5.5.3.3 单精度/双精度

例子略

5.5.3.4 布尔型

处理器方法:

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

请求 url:

```
http://localhost:8080/springmvc_mybatis/item/editItem.action?id=2&status=false
```

说明: 对于布尔类型的参数, 请求的参数值为 true 或 false。

5.5.3.5 @RequestParam

使用 @RequestParam 常用于处理简单类型的绑定。

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

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

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

defaultValue: 默认值，表示如果请求中没有同名参数时的默认值

定义如下：

```
public String editItem(@RequestParam(value="item_id",required=true) String id) {  
  
}
```

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

注意：如果请求参数中没有 item_id 将跑出异常：

HTTP Status 500 - Required Integer parameter 'item_id' is not present

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

5.5.4 pojo

5.5.4.1简单 pojo

将 pojo 对象中的属性名于传递进来的属性名对应，如果传进来的参数名称和对象中的属性名称一致则将参数值设置在 pojo 对象中

页面定义如下；

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

Contrller 方法定义如下：

```
@RequestMapping("/editItemSubmit")  
public String editItemSubmit(Items items)throws Exception{  
    System.out.println(items);  
}
```

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

5.5.4.2包装 pojo

如果采用类似 struts 中对象. 属性的方式命名，需要将 pojo 对象作为一个包装对象的属性，action 中以该包装对象作为形参。

包装对象定义如下：

```
Public class QueryVo {  
private Items items;  
  
}
```

页面定义：

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

Controller 方法定义如下：

```
public String useraddsubmit(Model model, QueryVo queryVo) throws Exception {  
System.out.println(queryVo.getItems());  
}
```

5.5.5 自定义参数绑定

5.5.5.1 需求

根据业务需求自定义日期格式进行参数绑定。

5.5.5.2 Converter

5.5.5.2.1 自定义 Converter

```
public class CustomDateConverter implements Converter<String, Date> {  
  
    @Override  
    public Date convert(String source) {  
        try {  
            SimpleDateFormat simpleDateFormat = new  
SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
}
```

```
        return simpleDateFormat.parse(source);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
}
```

5.5.5.2.2 配置方式 1

```
<mvc:annotation-driven conversion-service="conversionService">
</mvc:annotation-driven>
<!-- conversionService -->
    <bean id="conversionService"

        class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <!-- 转换器 -->
        <property name="converters">
            <list>
                <bean
class="cn.itcast.ssm.controller.converter.CustomDateConverter"/>
            </list>
        </property>
    </bean>
```

5.5.5.2.3 配置方式 2(自学)

```
<!--注解适配器 -->
    <bean

        class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
        <property name="webBindingInitializer"
ref="customBinder"></property>
    </bean>
```



```
<!-- 自定义webBinder -->
<bean id="customBinder"

    class="org.springframework.web.bind.support.ConfigurableWebBindingIn
itializer">
    <property name="conversionService" ref="conversionService" />
</bean>
<!-- conversionService -->
<bean id="conversionService"

    class="org.springframework.format.support.FormattingConversionServic
eFactoryBean">
    <!-- 转换器 -->
    <property name="converters">
        <list>
            <bean
class="cn.itcast.ssm.controller.converter.CustomDateConverter"/>
        </list>
    </property>
</bean>
```

5.5.6 集合类

5.5.6.1 字符串数组

页面定义如下:

页面选中多个 checkbox 向 controller 方法传递

```
<input type="checkbox" name="item_id" value="001"/>
<input type="checkbox" name="item_id" value="002"/>
<input type="checkbox" name="item_id" value="002"/>
```

传递到 controller 方法中的格式是: 001,002,003

Controller 方法中可以用 String[] 接收, 定义如下:

```
public String deleteitem(String[] item_id)throws Exception{
    System.out.println(item_id);
}
```

5.5.6.2 List

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

List 中对象：

成绩对象

```
Public class QueryVo {  
Private List<Items> itemList;//商品列表  
  
    //get/set 方法..  
}
```

包装类中定义 List 对象，并添加 get/set 方法如下：

页面定义如下：

```
<tr>  
<td>  
<input type="text" name=" itemList[0].id" value="${item.id}"/>  
</td>  
<td>  
<input type="text" name=" itemList[0].name" value="${item.name }"/>  
</td>  
<td>  
<input type="text" name=" itemList[0].price" value="${item.price}"/>  
</td>  
</tr>  
<tr>  
<td>  
<input type="text" name=" itemList[1].id" value="${item.id}"/>  
</td>  
<td>  
<input type="text" name=" itemList[1].name" value="${item.name }"/>  
</td>  
<td>  
<input type="text" name=" itemList[1].price" value="${item.price}"/>  
</td>  
</tr>
```

上边的静态代码改为动态 jsp 代码如下：

```
<c:forEach items="${itemsList }" var="item" varStatus="s">
<tr>
    <td><input type="text" name="itemsList[${s.index }].name"
value="${item.name }"/></td>
    <td><input type="text" name="itemsList[${s.index }].price"
value="${item.price }"/></td>
    .....
    .....
</tr>
</c:forEach>
```

Contrller 方法定义如下:

```
public String useraddsubmit(Model model, QueryVo queryVo) throws Exception {
    System.out.println(queryVo.getItemList());
}
```

5.5.6.3 Map

在包装类中定义 Map 对象，并添加 get/set 方法，action 使用包装对象接收。

包装类中定义 Map 对象如下:

```
Public class QueryVo {
private Map<String, Object> itemInfo = new HashMap<String, Object>();
    //get/set 方法..
}
```

页面定义如下:

```
<tr>
<td>学生信息: </td>
<td>
姓名: <input type="text" name="itemInfo['name']"/>
年龄: <input type="text" name="itemInfo['price']"/>
. . . . .
</td>
</tr>
```

Contrller 方法定义如下:

```
public String useraddsubmit(Model model, QueryVo queryVo) throws Exception {  
    System.out.println(queryVo.getStudentinfo());  
}
```

5.6 问题总结

5.6.1 404

页面找不到，视图找不到。



HandlerMapping 根据 url 没有找到 Handler。



5.6.2 Post 时中文乱码

在 web.xml 中加入：

```
<filter>  
<filter-name>CharacterEncodingFilter</filter-name>
```

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

以上可以解决 post 请求乱码问题。

对于 get 请求中文参数出现乱码解决方法有两个：

修改 tomcat 配置文件添加编码与工程编码一致，如下：

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

另外一种方法对参数进行重新编码：

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

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

5.7 与 struts2 不同

- 1、springmvc 的入口是一个 servlet 即前端控制器，而 struts2 入口是一个 filter 过滤器。
- 2、springmvc 是基于方法开发(一个 url 对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2 是基于类开发，传递参数是通过类的属性，只能设计为多例。
- 3、Struts 采用值栈存储请求和响应的数据，通过 OGNL 存取数据，springmvc 通过参数解析器是将 request 请求内容解析，并给方法形参赋值，将数据和视图封装成 ModelAndView 对象，最后又将 ModelAndView 中的模型数据通过 reques 域传输到页面。Jsp 视图解析器默认使用 jstl。

6 注解开发-高级

6.1 Validation（了解）


b/s 系统中对 http 请求数据的校验多数在客户端进行，这也是出于简单及用户体验性上考虑，但是在一些安全性要求高的系统中服务端校验是不可缺少的，本节主要学习 springmvc 实现控制层添加校验。


Spring3 支持 JSR-303 验证框架，JSR-303 是 JAVA EE 6 中的一项子规范，叫做 Bean Validation，官方参考实现是 Hibernate Validator（与 Hibernate ORM 没有关系），JSR 303 用于对 Java Bean 中的字段的值进行验证。


6.1.1 需求

在商品信息修改提交时对商品信息内容进行校验，例如商品名称必须输入，价格合法性校验。

6.1.2 加入 jar 包

 hibernate-validator-4.3.0.Final.jar

 jboss-logging-3.1.0.CR2.jar

 validation-api-1.0.0.GA.jar

6.1.3 配置 validator

```
<!-- 校验器 -->
<bean id="validator"

    class="org.springframework.validation.beanvalidation.LocalValidatorF
```

```
actoryBean">
    <!-- 校验器-->
    <property name="providerClass"
value="org.hibernate.validator.HibernateValidator" />
    <!-- 指定校验使用的资源文件，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.ReloadableResourceBundleM
essageSource">
    <!-- 资源文件名-->
    <property name="basenames">
        <list>
            <value>classpath:CustomValidationMessages</value>
        </list>
    </property>
    <!-- 资源文件编码格式 -->
    <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>
```

6.1.4 将 validator 加到处理器适配器

6.1.4.1 配置方式 1

```
<mvc:annotation-driven validator="validator">
</mvc:annotation-driven>
```

6.1.4.2 配置方式 2(自学)

```
<!-- 自定义webBinder -->
```

```
<bean id="customBinder"

    class="org.springframework.web.bind.support.ConfigurableWebBindingIn
itializer">
    <property name="validator" ref="validator" />
</bean>
```

```
<!-- 注解适配器 -->
<bean

    class="org.springframework.web.servlet.mvc.method.annotation.Request
MappingHandlerAdapter">
    <property name="webBindingInitializer"
ref="customBinder"></property>
</bean>
```

6.1.5 添加验证规则

```
public class Items {
    private Integer id;
    @Size(min=1,max=30,message="{item.name.length.error}")
    private String name;

    @NotEmpty(message="{pic.is.null}")
    private String pic;
```

6.1.6 错误消息文件 CustomValidationMessages

item.name.length.error=商品名称在1到30个字符之间
pic.is.null=请上传图片

如果在 eclipse 中编辑 properties 文件无法看到中文则参考“Eclipse 开发环境配置-indigo.docx”添加 propedit 插件。

6.1.7 捕获错误

修改 Controller 方法:

```
// 商品修改提交
@RequestMapping("/editItemSubmit")
public String editItemSubmit(@Validated @ModelAttribute("item") Items
items, BindingResult result,
    @RequestParam("pictureFile") MultipartFile[] pictureFile, Model
model)
    throws Exception {
    //如果存在校验错误则转到商品修改页面
    if (result.hasErrors()) {
        List<ObjectError> errors = result.getAllErrors();
        for(ObjectError objectError:errors){
            System.out.println(objectError.getCode());
            System.out.println(objectError.getDefaultMessage());
        }
        return "item/editItem";
    }
}
```

注意: 添加@Validated 表示在对 items 参数绑定时进行校验, 校验信息写入 BindingResult 中, 在要校验的 pojo 后边添加 BingdingResult, 一个 BindingResult 对应一个 pojo, 且 BingdingResult 放在 pojo 的后边。

商品修改页面显示错误信息:

页头:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt"
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

在需要显示错误信息地方:

```
<spring:hasBindErrors name="item">
<c:forEach items="${errors.allErrors}" var="error">
    ${error.defaultMessage }<br/>
</c:forEach>
</spring:hasBindErrors>
```

说明:

<spring:hasBindErrors name="item">表示如果 item 参数绑定校验错误下边显示错误信息。

上边的方法也可以改为:

在 controller 方法中将 error 通过 model 放在 request 域, 在页面上显示错误信息:

controller 方法:

```
if(bindingResult.hasErrors()){  
    model.addAttribute("errors", bindingResult);  
}
```

页面:

```
<c:forEach items="${errors.allErrors}" var="error">  
    ${error.defaultMessage }<br/>  
</c:forEach>
```

6.1.8 分组校验

如果两处校验使用同一个 Items 类则可以设定校验分组,通过分组校验可以对每处的校验个性化。

需求: 商品修改提交只校验商品名称长度

定义分组:

分组就是一个标识,这里定义一个接口:

```
public interface ValidGroup1 {  
  
}  
  
public interface ValidGroup2 {  
  
}
```

指定分组校验:

```
public class Items {  
    private Integer id;  
    //这里指定分组ValidGroup1, 此@Size校验只适用ValidGroup1校验  
  
    @Size(min=1,max=30,message="{item.name.length.error}",groups={ValidGroup1.class})  
    private String name;
```

```
// 商品修改提交  
@RequestMapping("/editItemSubmit")  
public String editItemSubmit(@Validated(value={ValidGroup1.class})  
@ModelAttribute("item") Items items, BindingResult result,  
@RequestParam("pictureFile") MultipartFile[] pictureFile, Model
```

```
model)
    throws Exception {
```

在@Validated 中添加 value={ValidGroup1.class}表示商品修改使用了 ValidGroup1 分组校验规则，也可以指定多个分组中间用逗号分隔，

```
@Validated(value={ValidGroup1.class, ValidGroup2.class })
```

6.1.9 校验注解

@Null 被注释的元素必须为 null

@NotNull 被注释的元素必须不为 null

@AssertTrue 被注释的元素必须为 true

@AssertFalse 被注释的元素必须为 false

@Min(value) 被注释的元素必须是一个数字，其值必须大于等于指定的最小值

@Max(value) 被注释的元素必须是一个数字，其值必须小于等于指定的最大值

@DecimalMin(value) 被注释的元素必须是一个数字，其值必须大于等于指定的最小值

@DecimalMax(value) 被注释的元素必须是一个数字，其值必须小于等于指定的最大值

@Size(max=, min=) 被注释的元素的大小必须在指定的范围内

@Digits(integer, fraction) 被注释的元素必须是一个数字，其值必须在可接受的范围内

@Past 被注释的元素必须是一个过去的日期

@Future 被注释的元素必须是一个将来的日期

@Pattern(regex=,flag=) 被注释的元素必须符合指定的正则表达式

Hibernate Validator 附加的 constraint

@NotBlank(message=) 验证字符串非 null，且长度必须大于 0

@Email 被注释的元素必须是电子邮箱地址

@Length(min=,max=) 被注释的字符串的大小必须在指定的范围内

@NotEmpty 被注释的字符串的必须非空

@Range(min=,max=,message=) 被注释的元素必须在合适的范围内

6.2 数据回显

6.2.1 需求

表单提交失败需要再回到表单页面重新填写，原来提交的数据需要重新在页面上显示。

6.2.2 简单数据类型

对于简单数据类型，如：Integer、String、Float 等使用 Model 将传入的参数再放到 request 域实现显示。

如下:

```
@RequestMapping(value="/editItems",method={RequestMethod.GET})
public String editItems(Model model,Integer id)throws Exception{

    //传入的id重新放到request域
    model.addAttribute("id", id);
}
```

6.2.3 pojo 类型

springmvc 默认支持 pojo 数据回显, springmvc 自动将形参中的 pojo 重新放回 request 域中, request 的 key 为 pojo 的类名 (首字母小写), 如下:

controller 方法:

```
@RequestMapping("/editItemSubmit")
public String editItemSubmit(Integer id,ItemsCustom itemsCustom)throws
Exception{
```

springmvc 自动将 itemsCustom 放回 request, 相当于调用下边的代码:

```
model.addAttribute("itemsCustom", itemsCustom);
```

jsp 页面:

```
<tr>
    <td>商品名称</td>
    <td><input type="text" name="name" value="${itemsCustom.name }"/></td>
</tr>
<tr>
    <td>商品价格</td>
    <td><input type="text" name="price" value="${itemsCustom.price }"/></td>
</tr>
```

页面中的从 “itemsCustom” 中取数据。

如果 key 不是 pojo 的类名(首字母小写), 可以使用@ModelAttribute 完成数据回显。

@ModelAttribute 作用如下:

1、绑定请求参数到 pojo 并且暴露为模型数据传到视图页面

此方法可实现数据回显效果。

```
// 商品修改提交
@RequestMapping("/editItemSubmit")
public String editItemSubmit(Model model,@ModelAttribute("item")
ItemsCustom itemsCustom)
```

页面:

```
<tr>
    <td>商品名称</td>
    <td><input type="text" name="name" value="${item.name }"/></td>
```

```
</tr>
<tr>
    <td>商品价格</td>
    <td><input type="text" name="price" value="${item.price }"/></td>
</tr>
```

如果不用@ModelAttribute 也可以使用 model.addAttribute("item", itemsCustom) 完成数据回显。

2、将方法返回值暴露为模型数据传到视图页面

```
//商品分类
@ModelAttribute("itemtypes")
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;
}
```

页面：

商品类型：

```
<select name="itemtype">
    <c:forEach items="${itemtypes }" var="itemtype">
        <option value="${itemtype.key }">${itemtype.value }</option>

    </c:forEach>
</select>
```

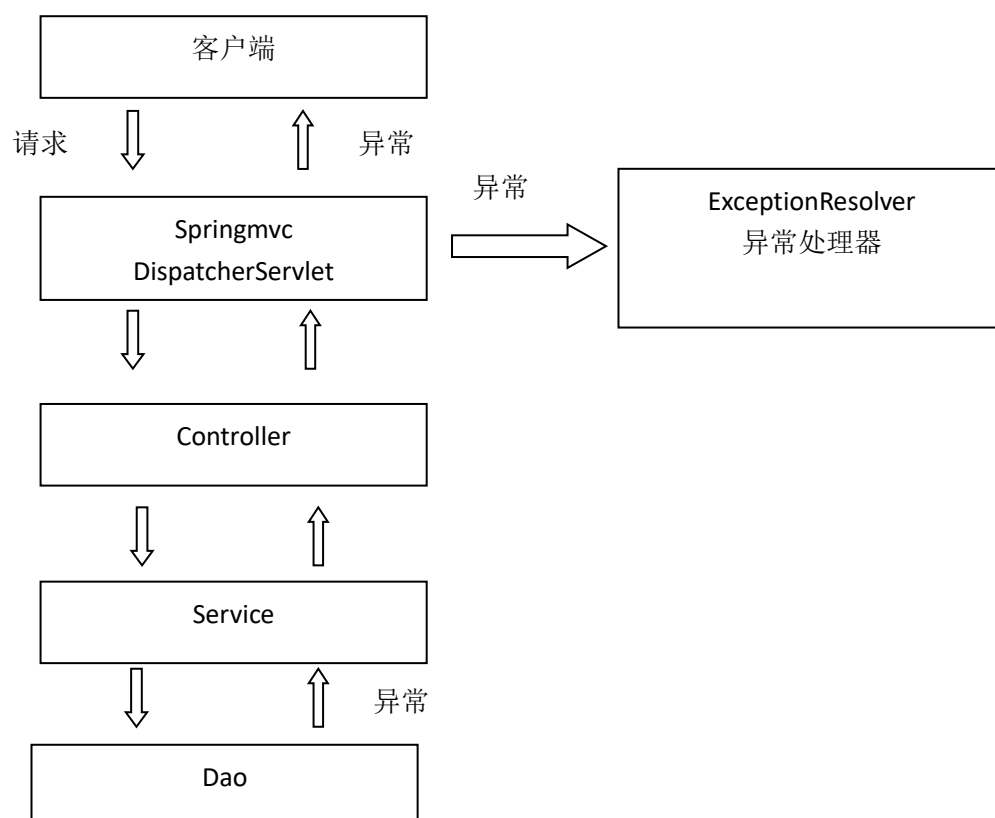
6.3 异常处理器

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

6.3.1 异常处理思路

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

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



6.3.2 自定义异常类

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

```
public class CustomException extends Exception {  
  
    /** serialVersionUID*/  
    private static final long serialVersionUID = -5212079010855161498L;  
  
    public CustomException(String message){  
        super(message);  
    }  
}
```

```
        this.message = message;
    }

    //异常信息
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

6.3.3 自定义异常处理器

```
public class CustomExceptionHandler implements HandlerExceptionResolver
{

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

        ex.printStackTrace();

        CustomException customException = null;

        //如果抛出的是系统自定义异常则直接转换
        if(ex instanceof CustomException){
            customException = (CustomException)ex;
        }else{
            //如果抛出的不是系统自定义异常则重新构造一个未知错误异常。
            customException = new CustomException("未知错误, 请与系统管理 员联系!");
        }

        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("message", customException.getMessage());
        modelAndView.setViewName("error");
    }
}
```

```
        return modelAndView;  
    }  
  
}
```

6.3.4 错误页面

```
<%@ page language="java" contentType="text/html; charset=UTF-8"  
    pageEncoding="UTF-8"%>  
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>  
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
    "http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
<title>错误页面</title>  
  
</head>  
<body>  
您的操作出现错误如下: <br/>  
${message }  
</body>  
  
</html>
```

6.3.5 异常处理器配置

在 springmvc.xml 中添加:

```
<!-- 异常处理器 -->  
    <bean id="handlerExceptionResolver"  
        class="cn.itcast.ssm.controller.exceptionResolver.CustomExceptionResolv  
er"/>
```


6.3.6 异常测试

修改商品信息，id 输入错误提示商品信息不存在。

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

```
// 调用service查询商品信息
Items item = itemService.findById(id);

if(item == null){
    throw new CustomException("商品信息不存在!");
}
```

在 service 中抛出异常方法同上。

6.4 上传图片

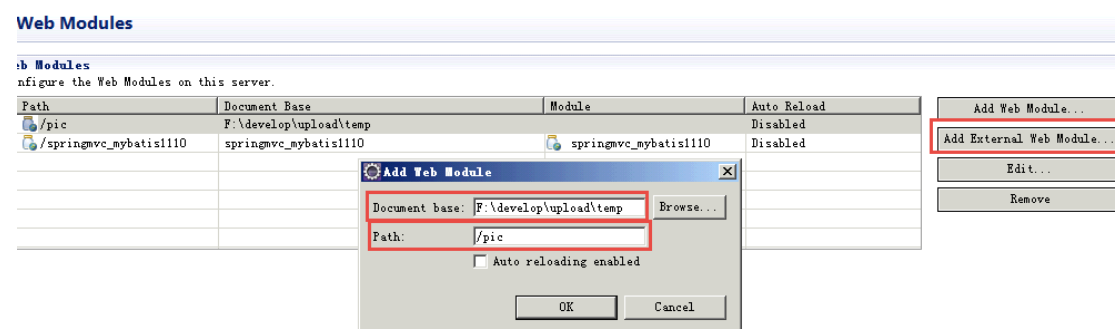
6.4.1 配置虚拟目录

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

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

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

也可以通过 eclipse 配置：





6.4.2 配置解析器

```
<!-- 文件上传 -->
<bean id="multipartResolver"

    class="org.springframework.web.multipart.commons.CommonsMultipartRes
olver">
    <!-- 设置上传文件的最大尺寸为5MB -->
    <property name="maxUploadSize">
        <value>5242880</value>
    </property>
</bean>
```

6.4.3 jar 包

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

 commons-fileupload-1.2.2.jar
 commons-io-2.4.jar

6.4.4 图片上传

◆ controller:

```
//商品修改提交
@RequestMapping("/editItemSubmit")
public String editItemSubmit(Items items, MultipartFile
pictureFile)throws Exception{

    //原始文件名称
    String pictureFile_name = pictureFile.getOriginalFilename();
    //新文件名称
    String newFileName =
UUID.randomUUID().toString()+pictureFile_name.substring(pictureFile_nam
e.lastIndexOf("."));

    //上传图片
    File uploadPic = new
java.io.File("F:/develop/upload/temp/"+newFileName);
```

```
        if(!uploadPic.exists()){
            uploadPic.mkdirs();
        }
        //向磁盘写文件
        pictureFile.transferTo(uploadPic);

        .....
    }
```

◆ 页面:

form 添加 enctype="multipart/form-data":

```
<form id="itemForm"

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

    method="post" enctype="multipart/form-data">
    <input type="hidden" name="pic" value="${item.pic }" />
```

file 的 name 与 controller 形参一致:

```
        <tr>
            <td>商品图片</td>
            <td><c:if test="${item.pic !=null}">
                
                <br />
            </c:if> <input type="file" name="pictureFile" /></td>
        </tr>
```

6.5 json 数据交互

6.5.1 @RequestBody

作用:

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

本例子应用：

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

6.5.2 @ResponseBody

作用：

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

本例子应用：

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

6.5.3 请求 json，响应 json 实现：

6.5.3.1环境准备

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

```
010 jackson-core-asl-1.9.11.jar -
010 jackson-mapper-asl-1.9.11.jar
```

6.5.3.2配置 json 转换器

在注解适配器中加入 `messageConverters`

```
<!--注解适配器 -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
    <property name="messageConverters">
        <list>
            <bean
class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"></bean>
        </list>
    </property>
</bean>
```

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

6.5.3.3controller 编写

```
// 商品修改提交json信息，响应json信息
@RequestMapping("/editItemSubmit_RequestJson")
public @ResponseBody Items editItemSubmit_RequestJson(@RequestBody
Items items) throws Exception {
    System.out.println(items);
    //itemService.saveItem(items);
    return items;
}
```

6.5.3.4页面 js 方法编写:

引入 js:

```
<script type="text/javascript"
```

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

```
//请求json响应json
function request_json(){
    $.ajax({
        type:"post",

        url:"${pageContext.request.contextPath }/item/editItemSubmit_Request
Json.action",
        contentType:"application/json;charset=utf-8",
        data:'{"name":"测试商品","price":99.9}',
        success:function(data){
            alert(data);
        }
    });
}
```

6.5.3.5测试结果:



从上图可以看出请求的数据是 json 格式

6.5.4 请 key/value，响应 json 实现：

表单默认请求 application/x-www-form-urlencoded 格式的数据即 key/value，通常有 post 和 get 两种方法，响应 json 数据是为了方便客户端处理，实现如下：

6.5.4.1 环境准备

同第一个例子

6.5.4.2 controller 编写

```
// 商品修改提交，提交普通form表单数据，响应json
@RequestMapping("/editItemSubmit_ResponseJson")
public @ResponseBody Items editItemSubmit_ResponseJson(Items items)
throws Exception {

    System.out.println(items);

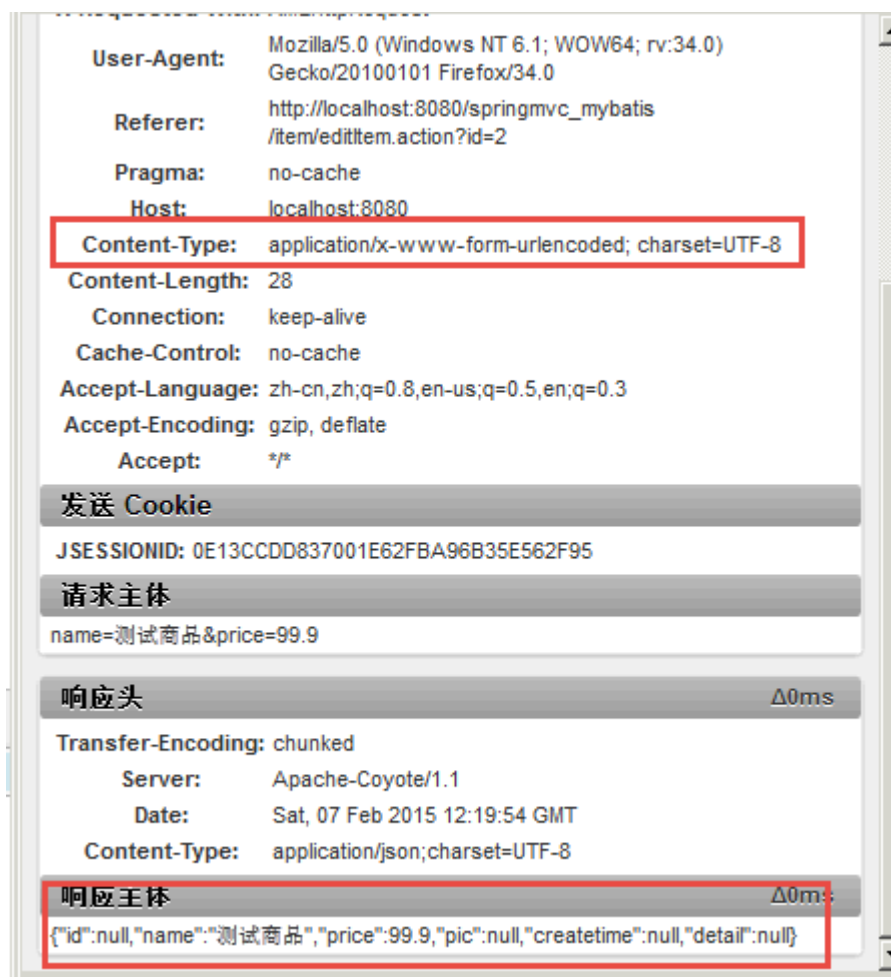
//    itemService.saveItem(items);
    return items;
}
```

6.5.4.3 页面 js 方法编写:

```
function formsubmit(){
    var user = "name=测试商品&price=99.9";
    alert(user);
    $.ajax(
        {
            type: 'post', //这里改为get也可以正常执行
            url: '${pageContext.request.contextPath}/item/
editItemSubmit_RequestJson.action',
//ContentType没指定将默认为: application/x-www-form-urlencoded
            data: user,
            success: function(data){
                alert(data.name);
            }
        }
    )
}
```

从上边的 js 代码看出，已去掉 ContentType 的定义，ContentType 默认为：application/x-www-form-urlencoded 格式。

6.5.4.4测试结果



从上图可以看出请求的数据是标准的 key/value 格式。

6.5.5 小结

实际开发中常用第二种方法，请求 key/value 数据，响应 json 结果，方便客户端对结果进行解析。

6.6 RESTful 支持

6.6.1 需求

RESTful 方式实现商品信息查询，返回 json 数据

6.6.2 添加 DispatcherServlet 的 rest 配置

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

    <servlet-class>org.springframework.web.servlet.DispatcherServlet</se
rvclet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring/springmvc.xml</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>springmvc-servlet-rest</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

6.6.3 URL 模板模式映射

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

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

```
@RequestMapping("/viewItems/{id}")
public @ResponseBody viewItems(@PathVariable("id") String id,Model
model) throws Exception{
    //方法中使用@PathVariable获取useried的值，使用model传回页面
    //调用 service查询商品信息
    ItemsCustom itemsCustom = itemsService.findItemsById(id);
    return itemsCustom;
}
```

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

6.6.4 静态资源访问<mvc:resources>

如果在 `DispatcherServlet` 中设置 `url-pattern` 为 `/` 则必须对静态资源进行访问处理。
`spring mvc` 的 `<mvc:resources mapping="" location="">` 实现对静态资源进行映射访问。
如下是对 `js` 文件访问配置：

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

7 拦截器

7.1 定义

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

7.2 拦截器定义

实现 `HandlerInterceptor` 接口，如下：

```
Public class HandlerInterceptor1 implements HandlerInterceptor{

    /**
     * controller执行前调用此方法
     * 返回true表示继续执行，返回false中止执行
     * 这里可以加入登录校验、权限拦截等
     */
    @Override
    Public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception
    {
        // TODO Auto-generated method stub
        Return false;
    }
}
```

```

    * controller执行后但未返回视图前调用此方法
    * 这里可在返回用户前对模型数据进行加工处理，比如这里加入公用信息以便页面显示
    */
    @Override
    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        // TODO Auto-generated method stub
    }
    /**
    * controller执行后且视图返回后调用此方法
    * 这里可得到执行controller时的异常信息
    * 这里可记录操作日志，资源清理等
    */
    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        // TODO Auto-generated method stub
    }
}
}
```

7.3 拦截器配置

7.3.1 针对某种 mapping 配置拦截器

```
<bean
    class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="handlerInterceptor1"/>
            <ref bean="handlerInterceptor2"/>
        </list>
    </property>
</bean>
<bean id="handlerInterceptor1"
```

```
class="springmvc.intercapter.HandlerInterceptor1"/>
    <bean                                id="handlerInterceptor2"
class="springmvc.intercapter.HandlerInterceptor2"/>
```

7.3.2 针对所有 mapping 配置全局拦截器

```
<!--拦截器 -->
<mvc:interceptors>
    <!--多个拦截器,顺序执行 -->
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean
class="cn.itcast.springmvc.filter.HandlerInterceptor1"></bean>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <bean
class="cn.itcast.springmvc.filter.HandlerInterceptor2"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

7.4 正常流程测试

7.4.1 代码:

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

```
@Override
public boolean preHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler) throws Exception {
    System.out.println("HandlerInterceptor1.preHandle..");
    return true;
}
```

7.4.2 运行流程

HandlerInterceptor1..preHandle..

HandlerInterceptor2..preHandle..

HandlerInterceptor2..postHandle..

HandlerInterceptor1..postHandle..

HandlerInterceptor2..afterCompletion..

HandlerInterceptor1..afterCompletion..

7.5 中断流程测试

7.5.1 代码：

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

7.5.2 运行流程

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

HandlerInterceptor1..preHandle..

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

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

HandlerInterceptor1..preHandle..

HandlerInterceptor2..preHandle..

HandlerInterceptor1..afterCompletion..

从日志看出第二个拦截器的 preHandler 方法返回 false 后第一个拦截器的

postHandler 没有执行，第二个拦截器的 postHandler 和 afterCompletion 没有执行，且 controller 也不执行了。

总结：

preHandle 按拦截器定义顺序调用

postHandler 按拦截器定义逆序调用

afterCompletion 按拦截器定义逆序调用

postHandler 在拦截器链内所有拦截器返回成功调用

afterCompletion 只有 preHandle 返回 true 才调用

7.6 拦截器应用

7.6.1 用户身份认证

```
Public class LoginInterceptor implements HandlerInterceptor{

    @Override
    Public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception
    {

        //如果是登录页面则放行
        if(request.getRequestURI().indexOf("login.action")>=0){
            return true;
        }
        HttpSession session = request.getSession();
        //如果用户已登录也放行
        if(session.getAttribute("user")!=null){
            return true;
        }
        //用户没有登录挑战到登录页面

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

        return false;
    }
}
```

```
}
```

7.6.2 用户登陆 controller

```
//登陆提交
//userid: 用户账号, pwd: 密码
@RequestMapping("/login")
public String loginsubmit(HttpSession session,String userid,String
pwd)throws Exception{

    //向session记录用户身份信息
    session.setAttribute("activeUser", userid);

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

//退出
@RequestMapping("/logout")
public String logout(HttpSession session)throws Exception{

    //session过期
    session.invalidate();

    return "redirect:item/queryItem.action";
}
```

8 学生练习

商品查询添加查询条件：商品名称、价格范围

添加商品功能开发

删除商品功能开发