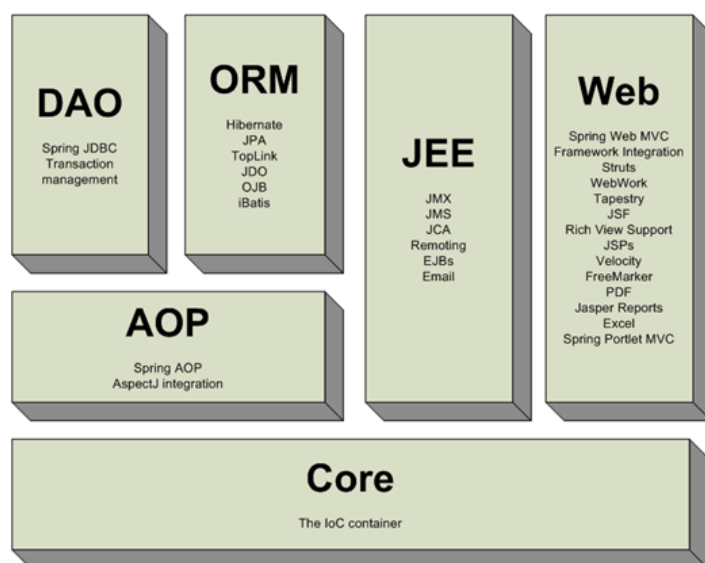


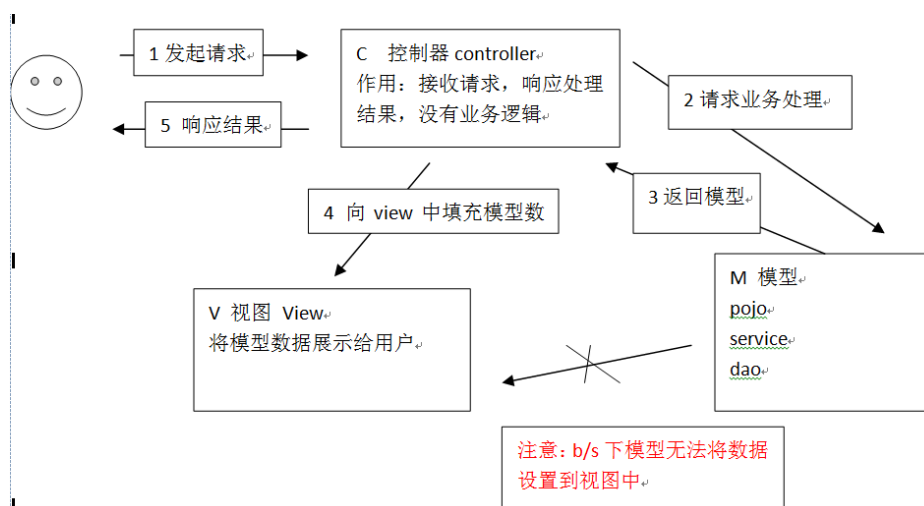
# 一、什么是 SpringMVC

## 1、SpringMVC 简介

spring web mvc 和 struts2 都属于表现层的框架，如下图 spring 的整体结构图中可以看出，SpringMVC 是 spring 框架的一个模块，SpringMVC 和 spring 无需通过中间整合层进行整合。



SpringMVC 是一个基于 web 的 mvc 框架。mvc 可以理解为是一种设计模式。如下图所示是 mvc 设计模式在 b/s 系统下应用：

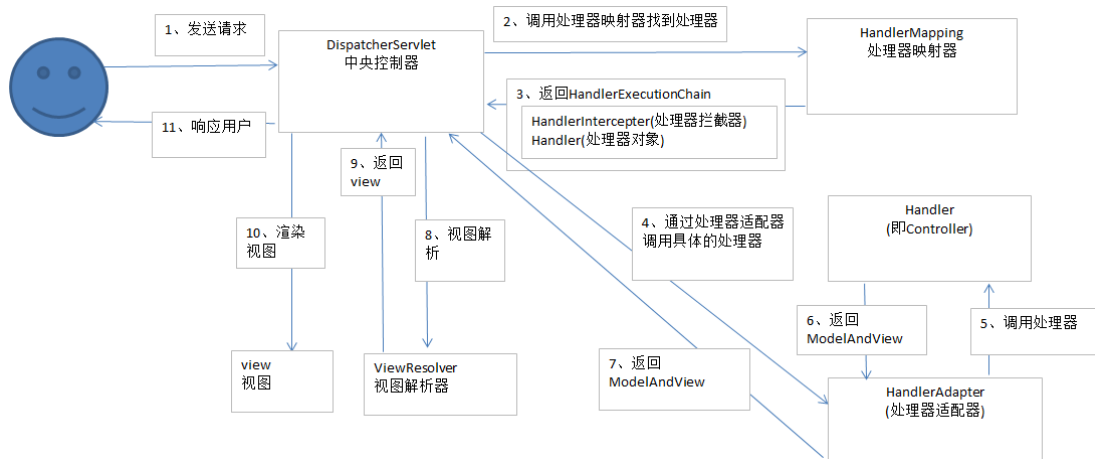


1、用户发起 request 请求至控制器(Controller)，控制接收用户请求的数据，委托给模型进行处理；

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 应用中模型是可以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。

## 2、SpringMVC 架构



- 1、用户发送请求至 **前端控制器 DispatcherServlet**。
- 2、DispatcherServlet 收到请求调用 **处理器映射器 HandlerMapping**。
- 3、处理器映射器根据请求 url 找到具体的 **处理器 Handler**，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 **处理器适配器 HandlerAdapter** 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**。ModelAndView 对象是包含视图和业务数据的混合对象。
- 7、HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。
- 8、DispatcherServlet 将 ModelAndView 传给 **视图解析器 ViewResolver**。
- 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。

需要注意的是，在实际开发中，我们主要关注的是 **处理器 Controller** 和 **视图 View** 的开发。

### 3、SpringMVC 核心组件

#### 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 的存在降低了各组件之间的耦合性。

#### 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。

## 二、SpringMVC 入门案例

### 1、基本需求

- 1、业务流程：管理员维护商品信息；用户挑选商品，购买，创建订单。
- 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、项目开发

- 1、首先导入项目所需 jar 包：

 commons-logging-1.2.jar  
 jstl-1.2.jar  
 log4j-1.2.17.jar  
 spring-aop-4.3.13.RELEASE.jar  
 spring-aspects-4.3.13.RELEASE.jar  
 spring-beans-4.3.13.RELEASE.jar  
 spring-context-4.3.13.RELEASE.jar  
 spring-core-4.3.13.RELEASE.jar  
 spring-expression-4.3.13.RELEASE.jar  
 spring-jdbc-4.3.13.RELEASE.jar  
 spring-orm-4.3.13.RELEASE.jar  
 spring-test-4.3.13.RELEASE.jar  
 spring-tx-4.3.13.RELEASE.jar  
 spring-web-4.3.13.RELEASE.jar  
 spring-webmvc-4.3.13.RELEASE.jar

- 2、前端控制器的配置，在 web.xml 文件中：

```
<!-- 配置前端控制器 -->
<servlet>
    <servlet-name>SpringMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
```

```

    <!-- contextConfigLocation配置SpringMVC加载的配置文件（配置文件里面配置处理器适配器，
    处理器等等） -->
    <!-- 如果不配置contextConfigLocation，则默认加载/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>
    <!-- 第一种： *.action，所有以*.action结尾的访问，都由DispatcherServlet解析
        第二种： / ，所有访问的地址都由DispatcherServlet解析，对于这种配置会出现一个问题，
        那就是静态文件的访问不需要DispatcherServlet进行解析（因为访问静态文件直接返回即可，不用再由
        处理器处理）。但是这种方式可以实现RESTful风格的url
        第三种： /* ，这种配置不对，使用这种配置，最终要转发到一个jsp页面时，仍然会由
        DispatcherServlet解析jsp的地址，不能根据jsp的页面找到Handler，会报错-->
    <url-pattern>*.action</url-pattern>
</servlet-mapping>

```

**url-pattern:** \*.action 的请求交给 DispatcherServlet 处理。除此之外还有其他一些配置方法：

(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。

**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。

最后还可以配置：load-on-startup: 1，表示 servlet 随服务启动。

### 3、配置处理器适配器，在 SpringMVC.xml 文件中：

```

<!-- 处理器适配器，所有处理器适配器都实现HandlerAdapter -->
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>

```

**SimpleControllerHandlerAdapter:** 即简单控制器处理适配器。通过查看源代码可以发现，SimpleControllerHandlerAdapter 实现了 HandlerAdapter 接口。

```

public class SimpleControllerHandlerAdapter implements HandlerAdapter {

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

    public interface Controller {

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

```

所有实现了 `org.springframework.web.servlet.mvc.Controller` 接口的 Bean 作为 SpringMVC 的后端控制器（处理器）。

4、编写 Handler，需要实现 Controller 接口，它才能由 SimpleControllerHandlerAdapter 处理器适配器执行。

```

package com.zxt.controller;

import java.util.ArrayList;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

import com.zxt.domain.Items;

/**
 * 实现Controller接口的处理器
 * @author zxt
 */
public class ItemsController implements Controller {

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

        // 商品列表
    }
}

```

```

        ArrayList<Items> itemList = new ArrayList<Items>();

        // 这些数据应该由service查询数据库得到，这里先用静态数据代替
        Items items1 = new Items();
        items1.setId(1);
        items1.setName("联想笔记本");
        items1.setPrice(6000f);
        items1.setDetail("ThinkPad T430 联想笔记本电脑! ");

        Items items2 = new Items();
        items2.setId(2);
        items2.setName("苹果手机");
        items2.setPrice(5000f);
        items2.setDetail("iphone6苹果手机! ");

        itemList.add(items1);
        itemList.add(items2);

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

        // 填充数据
        // 相当于request的setAttribute方法
        modelAndView.addObject("itemsList", itemList);

        // 设置视图
        modelAndView.setViewName("items/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。

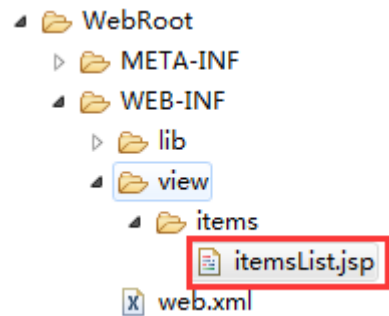
## 5、视图开发：

在还没有配置视图解析器的时候，`ModelAndView` 的视图设置如下：

```

// 设置视图
modelAndView.setViewName("/WEB-INF/view/items/itemsList.jsp");

```



```
<table>
  <thead>
    <tr>
      <td>商品编号</td>
      <td>商品名称</td>
      <td>商品价格</td>
      <td>商品详情</td>
    </tr>
  </thead>

  <tbody>
    <c:forEach items="${itemsList}" var="items" >
      <tr>
        <td>${items.id }</td>
        <td>${items.name }</td>
        <td>${items.price }</td>
        <td>${items.detail }</td>
      </tr>
    </c:forEach>
  </tbody>
</table>
```

## 6、配置处理器映射器。

```
<!-- 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。

## 7、配置处理器

```
<!-- 配置处理器 -->
<bean name="/queryItems.action" class="com.zxt.controller.ItemsController"></bean>
```

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

## 8、配置视图解析器。这里需要配置解析 jsp 页面的视图解析器。



```
<!-- 视图映射器 -->
<!-- 解析jsp页面，默认使用JSTL标签，所以在classpath目录下得由jstl的jar包 -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/view/" />
    <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/view/hello.jsp”。

此时，ModelAndView 设置视图如下:

```
// 设置视图
modelAndView.setViewName("items/itemsList");
```

### 三、非注解的处理器映射器和适配器

#### 1、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。

##### 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。

```
<!-- 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" />
```

##### 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。

```
<!-- 配置处理器 -->
<bean id="ItemsController" name="/queryItems.action" class="com.zxt.controlle.
<!-- 简单url -->
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/queryItems1.action">ItemsController</prop>
            <prop key="/queryItems2.action">ItemsController</prop>
        </props>
    </property>
</bean>
```



多个映射器可以并存,前端控制器判断 url 能够让哪些映射器映射,就让正确的映射器处理。

#### 2、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。

##### 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。

```
<!-- 处理器适配器，所有处理器适配器都实现HandlerAdapter -->
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
```

## 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：

```
<!-- HttpRequestHandlerAdapter适配器 -->
<bean class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"/>
```

Controller 实现如下：

```
public class ItemsController1 implements HttpRequestHandler {
    @Override
    public void handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        // 商品列表
        ArrayList<Items> itemList = new ArrayList<Items>();

        // 这些数据应该由service查询数据库得到，这里先用静态数据代替
        Items items1 = new Items();
        items1.setId(1);
        items1.setName("联想笔记本");
        items1.setPrice(6000f);
        items1.setDetail("ThinkPad T430 联想笔记本电脑!");

        Items items2 = new Items();
        items2.setId(2);
        items2.setName("苹果手机");
        items2.setPrice(5000f);
        items2.setDetail("iphone6苹果手机!");

        itemList.add(items1);
        itemList.add(items2);

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

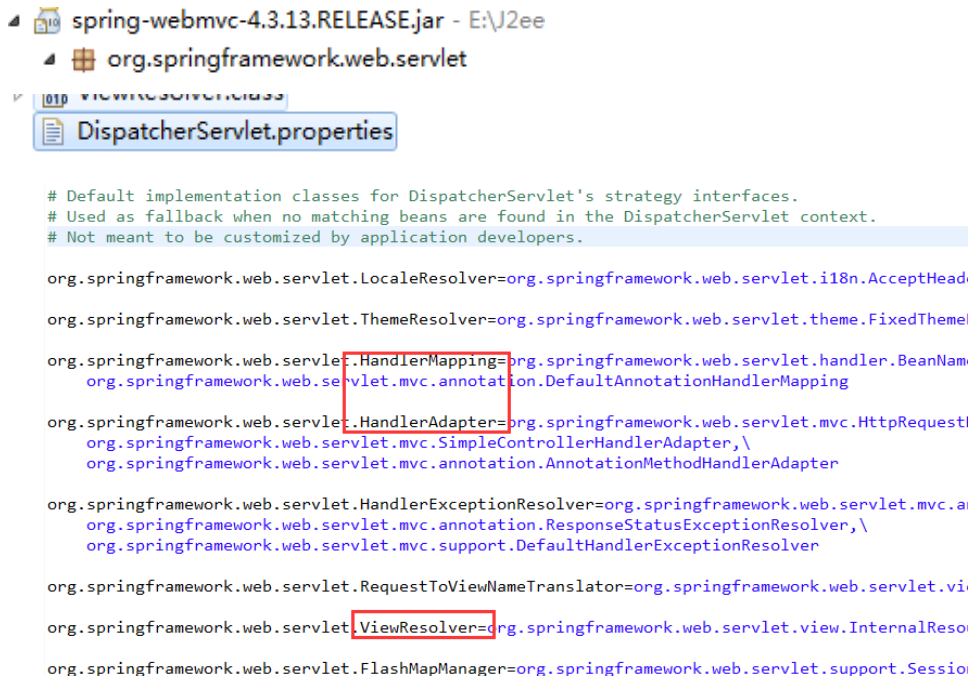
        // 设置视图
        request.getRequestDispatcher("/WEB-INF/view/items/itemsList.jsp").forward(request, response);
    }
}
```

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");
```

## 四、注解的处理器映射器和适配器

### 1、DispatcherServlet.properties



如果不在 SpringMVC.xml 文件中配置处理器映射器、适配器、视图解析器等组件，前端控制器会从上边的文件中加载，使用默认的组件。

### 2、注解的处理器映射器和适配器版本

在 spring3.1 之前，使用

org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping 注解映射器。

在 spring3.1 之后，使用

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping 注解映射器。

在 spring3.1 之前，使用

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

在 spring3.1 之后，使用

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

### 3、配置注解映射器和适配器

```
<!--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"/>
```

#### 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。

注解描述：**`@RequestMapping`**：定义请求 `url` 到处理器功能方法的映射。

#### RequestMappingHandlerAdapter

注解式处理器适配器，对标记 `@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。

需要注意的是：注解的映射器和注解的适配器需要配对使用。

#### <mvc:annotation-driven>

```
<!-- springmvc使用<mvc:annotation-driven>自动加载RequestMappingHandlerMapping和RequestMappingHandlerAdapter -->
<!-- <mvc:annotation-driven>还默认加载了很多参数绑定 -->
<mvc:annotation-driven></mvc:annotation-driven>
```

SpringMVC 使用 `<mvc:annotation-driven>` 自动加载 `RequestMappingHandlerMapping` 和 `RequestMappingHandlerAdapter`，可在 `SpringMVC.xml` 配置文件中使用 `<mvc:annotation-driven>` 替代上述注解处理器和适配器的配置。

### 4、注解处理器 Handler 的开发

首先需要注意：使用注解开发需要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="com.zxt.springmvc.controller"/>
```

Controller 的代码如下：

```
/**
 * @Controller标识该类是一个控制器
 * @author zxt
 */
@Controller
public class ItemsController {
    // @RequestMapping实现对itemsList方法和url进行映射，一个方法对应一个url
    // 一般建议url与方法名一致，方便维护
    @RequestMapping("/itemsList")
    public ModelAndView itemsList() {
        // 商品列表
        ArrayList<Items> itemsList = new ArrayList<Items>();

        // 这些数据应该由service查询数据库得到，这里先用静态数据代替
        Items items1 = new Items();
        items1.setId(1);
        items1.setName("联想笔记本");
        items1.setPrice(6000f);
        items1.setDetail("ThinkPad T430 联想笔记本电脑！");
        Items items2 = new Items();
        items2.setId(2);
        items2.setName("苹果手机");
        items2.setPrice(5000f);
        items2.setDetail("iphone6苹果手机！");

        itemsList.add(items1);
        itemsList.add(items2);

        // 创建ModelAndView 填充数据、设置视图
        ModelAndView modelAndView = new ModelAndView();
        // 填充数据
        modelAndView.addObject("itemsList", itemsList);
        // 设置视图
        modelAndView.setViewName("items/itemsList");

        return modelAndView;
    }
    // 这里可以定义其他方法
}
```

## 5、@RequestMapping 注解

通过 RequestMapping 注解可以定义不同的处理器映射规则。

@RequestMapping(value="/items") 或 @RequestMapping("/items"), value 的值是数组, 可以将多个 url 映射到同一个方法。

### 5.1、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。

```
@Controller
// 一个项目中往往会有多个 Controller, 为了对 url 进行分类管理, 可以在这里定义根路径, 最终访问的 url 即: 根路径+子路径
// 比如商品列表的 url: /items/itemsList.action (窄化请求映射)
@RequestMapping("/items")
public class ItemsController {
```

### 5.2、请求方法限定

1、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="/editItems", method=RequestMethod.GET)

2、限定 POST 方法: @RequestMapping(method = RequestMethod.POST)

如果通过 GET 访问则报错: HTTP Status 405 - Request method 'GET' not supported。

3、GET 和 POST 都可以:

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

### 5.3、RequestMapping 接口

```
public interface RequestMapping extends Annotation {
    // 指定映射的名称
    public abstract String name();
    // 指定请求路径的地址
    public abstract String[] value();
    // 指定请求的方式, 是一个 RequestMethod 数组, 可以配置多个方法
    public abstract RequestMethod[] method();
    // 指定参数的类型
```

```
public abstract String[] params();  
public abstract String[] headers();  
// 指定数据请求的格式  
public abstract String[] consumes();  
// 指定返回的内容类型  
public abstract String[] produces();  
}
```

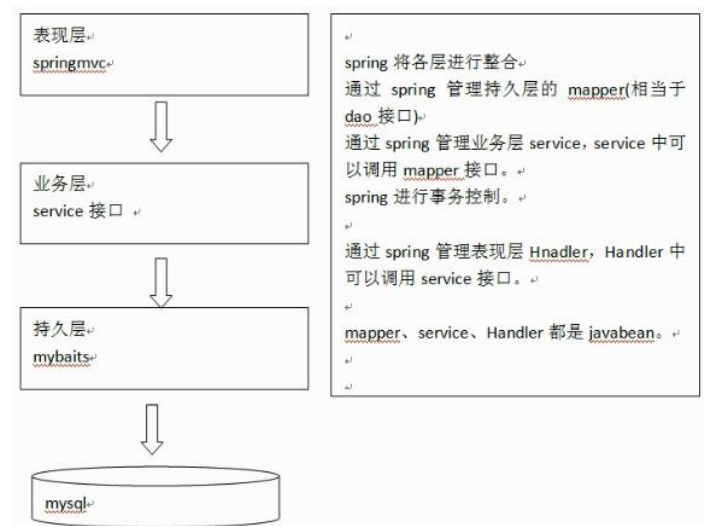
一个例子：

```
@RequestMapping(value = {"/modifyGet.do", "/modifyGet1.do"},  
method={RequestMethod.POST, RequestMethod.GET}, consumes={"application/json"},  
produces={"application/json"}, params={"name=mike", "pwd=123456"}, headers={"a=1"})
```



## 五、SpringMVC 与 mybatis 整合

### 1、整合原理

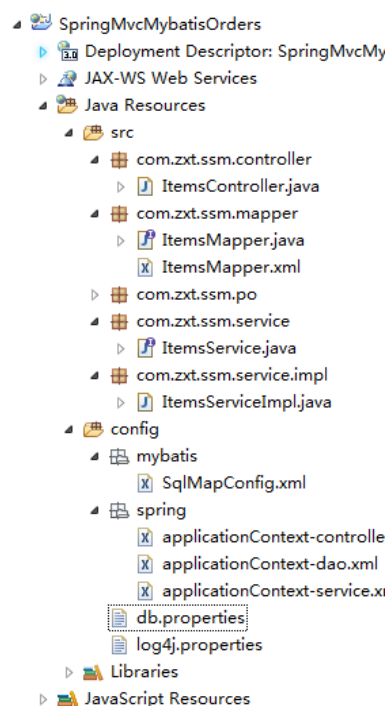


**第一步：整合 dao 层：**Mybatis 与 Spring 整合，通过 Spring 来管理 mapper 接口，使用 mapper 的扫描器自动扫描 mapper 接口在 spring 中进行注册；

**第二步：整合 service 层：**通过 spring 容器管理 service 接口，使用配置文件（或者注解的方式）将 service 接口配置在 spring 容器中，实现事务控制；

**第三步：整合 SpringMVC：**由于 SpringMVC 是 spring 的模块，所以无需整合。

工程目录结构如下：



## 2、整合 Dao 层

Dao 整合即 mybatis 与 spring 的整合，目标即通过 spring 管理 Mybatis 的 SqlSessionFactory、以及操作数据库的 mapper 接口。

1、首先 ItemsMapper.xml，编写商品查询的 sql；

```
<!-- Mapper.xml文件中的namespace与mapper接口的类路径相同。 -->
<mapper namespace="com.zxt.ssm.mapper.ItemsMapper">

    <!-- 定义sql片段 -->
    <sql id="query_items_condition">
        <!-- 使用动态sql，使用if判断，满足条件时才进行sql拼接 -->
        <if test="itemsCustom != null">
            <if test="itemsCustom.name != null and itemsCustom.name != ''">
                items.name like '%${itemsCustom.name}%'
            </if>
        </if>
    </sql>

    <!-- 商品列表查询 -->
    <!-- parameterType: 传入包装类（包装商品的查信息） -->
    <select id="selectItems" parameterType="com.zxt.ssm.po.ItemsQueryVo"
        resultType="com.zxt.ssm.po.ItemsCustom">
        select * from items
        <where>
            <include refid="query_items_condition"></include>
        </where>
    </select>

</mapper>
```

2、ItemsMapper.java 接口；

```
public interface ItemsMapper {
    public List<ItemsCustom> selectItems(ItemsQueryVo itemsQueryVo) throws Exception;
}
```

3、配置 mybatis 的配置文件 SqlMapConfig.xml；

4、编写 applicationContext-dao.xml，在其中配置数据库的连接池、

SqlSessionFactory 以及 mapper 接口的自动扫描：

```
<?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"
xmlns:mvc="http://www.springframework.org/schema/mvc"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.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="10"/>
    <property name="maxIdle" value="5"/>
</bean>

<!-- SqlSessionFactory -->
<!-- 让spring管理sqlSessionFactory 使用mybatis和spring整合包中的sqlSession工厂 -->
<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批量扫描，在Mapper的包中扫描出mapper接口，自动创建代理对象，并且在spring容器中注册，扫描之后，包下的Mapper的代理类的bean对象的id和Mapper接口名一样，首字母小写 -->
<!-- 这里同样需要：要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中-->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 指定扫描的包名，如果有多个包，不能用通配符，可以用逗号或分号分隔定义多个包 -->
    <property name="basePackage" value="com.zxt.ssm.mapper"></property>
    <!-- 这里配置SqlSessionFactory需要使用sqlSessionFactoryBeanName而不是sqlSessionFactory，这是由于使用sqlSessionFactory之后，扫描会先于上面配置文件的加载，这样

```

会出现错误 -->

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

</beans>
```

### 3、整合 Service 层

目标：1、Service 由 spring 管理；2、spring 对 Service 进行事务控制。

首先编写 service 的接口和实现类：

```
public interface ItemsService {
    // 商品查询列表
    public List<ItemsCustom> selectItems(ItemsQueryVo itemsQueryVo) throws Exception;
}

public class ItemsServiceImpl implements ItemsService {
    // 由于在applicationContext-dao.xml中配置了对mapper接口的自动批量扫描，所以这里可以直接注入使用
    @Autowired
    private ItemsMapper itemsMapper;

    @Override
    public List<ItemsCustom> selectItems(ItemsQueryVo itemsQueryVo) throws Exception
    {
        // 通过ItemsMapper接口来操作数据库
        return itemsMapper.selectItems(itemsQueryVo);
    }
}
```

然后创建 applicationContext-service.xml，配置 service。

```
<!-- 配置service，商品管理的service -->
<bean id="itemsService" class="com.zxt.ssm.service.impl.ItemsServiceImpl"></bean>
```

同时创建 applicationContext-transaction.xml 进行事务管理的配置：

```
<!-- 事务管理器：对mybatis操作数据库进行事务控制，spring使用jdbc的事务控制类 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- 注入dataSource， dataSource在applicationContext-dao.xml中已经配置好 -->
    <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 通知 -->
<tx:advice id="txadvice" transaction-manager="transactionManager">
```

```

<!-- 作事务操作 -->
<tx:attributes>
    <!-- 设置进行事务操作的方法的匹配 -->
    <tx:method name="save*" propagation="REQUIRED"/>
    <tx:method name="select*" 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:attributes>
</tx:advice>

<!-- 切面 -->
<aop:config>
    <!-- 切入点 -->
    <aop:pointcut expression="execution(* com.zxt.ssm.service.impl.*(..))"
id="pointcut1"/>

    <!-- 切面 -->
    <aop:advisor advice-ref="txadvice" pointcut-ref="pointcut1"/>
</aop:config>

```

## 4、SpringMVC 配置

1、创建 SpringMVC.xml，进行 SpringMVC 的各项配置，处理器映射器，处理器适配器等：

```

<!-- SpringMVC使用<mvc:annotation-driven>自动加载RequestMappingHandlerMapping和
RequestMappingHandlerAdapter -->
<!-- <mvc:annotation-driven>还默认加载了很多参数绑定 -->
<!-- 自动会加载 处理器映射器 和 处理器适配器 -->
<mvc:annotation-driven></mvc:annotation-driven>

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

<!-- 视图映射器 -->
<!-- 解析jsp页面，默认使用JSTL标签，所以在classpath目录下得有jstl的jar包 -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/view/" />
    <property name="suffix" value=".jsp"/>
</bean>

```

## 2、前端控制器：DispatcherServlet 在 web.xml 文件中配置

```
<!-- SpringMVC配置前端控制器 -->
<servlet>
    <servlet-name>SpringMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

<!-- contextConfigLocation配置SpringMVC加载的配置文件（配置文件里面配置处理器适配器，
处理器等等） -->
<!-- 如果不配置contextConfigLocation，则默认加载/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>
    <!-- 第一种： *.action，所有以*.action结尾的访问，都由DispatcherServlet解析
        第二种： / ，所有访问的地址都由DispatcherServlet解析，对于这种配置会出现一个问题，
        那就是静态文件的访问不需要，DispatcherServlet进行解析（因为访问静态文件直接返回即可，不用再
        由处理器处理）。但是这种方式可以实现RESTful风格的url
        第三种： /* ，这种配置不对，使用这种配置，最终要转发到一个jsp页面时，仍然会由
        DispatcherServlet解析jsp的地址，不能根据jsp的页面找到Handler，会报错-->
    <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

## 3、编写 controller（即 Handler）

```
@Controller
public class ItemsController {
    @Autowired
    private ItemsService itemsService;

    // @RequestMapping实现对itemsList方法和url进行映射，一个方法对应一个url，一般建议url
    与方法名一致，方便维护
    @RequestMapping("/itemsList")
    public ModelAndView itemsList() throws Exception {
        // 商品列表，这些数据由service查询数据库得到
        List<ItemsCustom> itemsList = itemsService.selectItems(null);

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

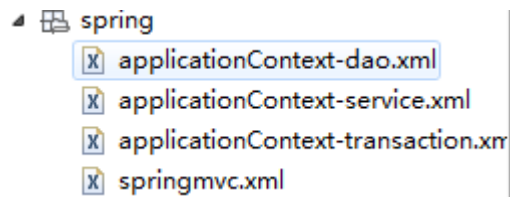


```

        <td>${items.price }</td>
        <td>${items.detail }</td>
    </tr>
</c:forEach>
</tbody>
</table>
</body>
</html>

```

5、加载 spring 容器，将 mapper、service、controller 加载到 spring 容器中。



建议使用通配符的形式加载上面的配置文件，在 web.xml 中，添加 spring 容器监听器，加载 spring 容器。

```

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

```

注意一个错误：一开始的时候，上面加载 spring 容器的时候，忘记<listener>了，导致 Service 的创建一直不成功（由于 Mapper 没有创建，所以 Service 中使用时无法注入，相应的 controller 中无法注入 service）。

## 5、商品信息修改功能

上述完成了商品信息的查询功能，现在需要完成商品信息的修改。

需求说明：

- 1、进入商品查询列表页面；
- 2、点击修改，进入商品修改页面，页面中显示要求修改的商品（从数据库中获取，根据 id）；
- 3、在商品修改页面修改商品信息，修改后，点击提交。



## 5.1、Mapper

1、根据 id 查询商品；2、根据 id 更新商品。

```
<!-- 根据商品id查询商品信息 -->
<select id="selectItemsById" parameterType="int" resultType="com.zxt.ssm.po.Items">
    select * from items where id=#{id}
</select>
<!-- 根据id更新商品信息 -->
<update id="updateItems" parameterType="com.zxt.ssm.po.ItemsCustom">
    update items
    <set>
        <if test="name != null">
            name=#{name},
        </if>
        <if test="pic != null">
            pic=#{pic},
        </if>
        <if test="createtime != null">
            createtime=#{createtime},
        </if>
        <if test="detail != null">
            detail=#{detail},
        </if>
        <if test="price != null">
            price=#{price},
        </if>
    </set>
    where id=#{id}
</update>
```

Mapper 接口

```
// 根据商品id获取商品信息
public Items selectItemsById(int id) throws Exception;
// 根据id修改商品信息
public void updateItems(ItemsCustom itemsCustom) throws Exception;
```

## 5.2、Service

service 接口功能：1、根据 id 查询商品信息；2、修改商品信息。

```
public interface ItemsService {
    // 商品查询列表
    public List<ItemsCustom> selectItems(ItemsQueryVo itemsQueryVo) throws Exception;

    /**
```

```

    * @Description: 根据商品id获取商品信息
    * @param id: 使用int无法判断是否为空，所以一般使用Integer
    * @throws Exception
    * @return: Items
    */
    public ItemsCustom selectItemsById(Integer id) throws Exception;

    /**
     * @Description: 根据id修改商品信息
     * @param id: 需要修改的商品信息的标记
     * @param itemsCustom
     * @throws Exception
     */
    public void updateItems(Integer id, ItemsCustom itemsCustom) throws Exception;
}

```

## Service 实现类

```

public class ItemsServiceImpl implements ItemsService {
    // 由于在applicationContext-dao.xml中配置了对mapper接口的自动批量扫描，所以这里可以直接注入使用
    @Autowired
    private ItemsMapper itemsMapper;

    @Override
    public List<ItemsCustom> selectItems(ItemsQueryVo itemsQueryVo) throws Exception
    {
        // 通过ItemsMapper接口来操作数据库
        return itemsMapper.selectItems(itemsQueryVo);
    }

    @Override
    public ItemsCustom selectItemsById(Integer id) throws Exception {
        Items items = itemsMapper.selectItemsById(id);
        /**
         * 中间对商品信息进行业务处理，返回Items的扩展类: ItemsCustom
         * 例如商品类中只有生产日期，而没有是否过期的信息，可以做一定的操作得到该信息，封装到ItemsCustom，显示给用户
         */
        ItemsCustom itemsCustom = new ItemsCustom();
        // 将Items中的属性复制到ItemsCustom中
        BeanUtils.copyProperties(items, itemsCustom);

        return itemsCustom;
    }
}

```

```

@Override
public void updateItems(Integer id, ItemsCustom itemsCustom) throws Exception {
    // 由于修改商品的信息必须要提供商品的id，所以这里首先得保证需要修改的商品具有id号
    itemsCustom.setId(id);
    itemsMapper.updateItems(itemsCustom);
}
}

```

### 5.3、Controller

方法：1、商品信息修改页面显示；2、商品信息修改提交。

```

// 商品修改页面的展示
@RequestMapping("/editItems")
public ModelAndView editItems(Integer id) throws Exception {
    // 根据Service获取商品信息
    ItemsCustom itemsCustom = itemsService.selectItemsById(id);

    // 创建ModelAndView 填充数据、设置视图
    ModelAndView modelAndView = new ModelAndView();
    // 填充数据
    modelAndView.addObject("itemsCustom", itemsCustom);
    // 设置视图
    modelAndView.setViewName("items/editItem");

    return modelAndView;
}

// 商品信息修改提交
@RequestMapping("/editItemsSubmit")
public ModelAndView editItemsSubmit(Integer id, ItemsCustom itemsCustom, String name)
throws Exception {
    itemsService.updateItems(id, itemsCustom);
    // 商品列表，这些数据由service查询数据库得到
    List<ItemsCustom> itemsList = itemsService.selectItems(null);

    // 创建ModelAndView 填充数据、设置视图
    ModelAndView modelAndView = new ModelAndView();
    // 填充数据
    modelAndView.addObject("itemsList", itemsList);
    // 设置视图
    modelAndView.setViewName("items/itemsList");

    return modelAndView;
}

```

## 六、Controller 方法的返回值

### 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。

### 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");
```

### 3、返回字符串

#### 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/view/items/editItem.jsp
```

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

#### 2、Redirect 重定向

Controller 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?...&....`。

### 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` 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。

## 七、参数绑定

Spring 的参数绑定过程：从客户端请求 key/value，经过参数绑定，将 key/value 数据绑定到 controller 方法的形参上。key/value 数据举例：  
queryItems.action?id=001&type=T01。

SpringMVC 中，接收页面提交的数据是通过方法的形参来接收的，而不是在 Controller 中定义成员变量来接收。

SpringMVC 的注解适配器对 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。在特殊情况下需要自定义 converter。

### 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，在 Controller 方法的形参中添加即可以使用。

1、HttpServletRequest：通过 request 对象获取请求信息

```
@RequestMapping("/itemsList")  
public ModelAndView itemsList(HttpServletRequest request) throws Exception {
```

2、HttpServletResponse：通过 response 处理响应信息

3、HttpSession：通过 session 对象得到 session 中存放的对象

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.findItemById(id);
```

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

页面通过\${item.XXXX}获取 item 对象的属性值。使用 Model 和 ModelMap 的效果一样，如果直接使用 Model，SpringMVC 会自动实例化 ModelMap。

### 2、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。

#### 1、整型

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

#### 2、字符串

#### 3、单精度/双精度

#### 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`。

### 3、@RequestParam

使用 `@RequestParam` 常用于处理简单类型的绑定。当形参名称与页面提交的参数的名称不一致时就要使用这个注解。

**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。

```
// 商品修改页面的展示
@RequestMapping("/editItems")
public ModelAndView editItems(@RequestParam(value = "id", required = true) Integer item_id) throws Exception {
}
}
```

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

注意: 如果请求参数中没有 `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 `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 `id` 参数值。

## 4、简单 pojo

将 pojo 对象中的属性名与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：

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>修改商品信息</title>
  </head>
  <body>
    <form id="itemForm" action="<%=basePath%>items/editItemsSubmit.action"
method="post">
      <input type="hidden" name="id" value="${itemsCustom.id }"/>
      修改商品信息：
      <table width="100%" border="1">
        <tr>
          <td>商品名称</td>
          <td><input type="text" name="name"
value="${itemsCustom.name }"/></td>
        </tr>
        <tr>
          <td>商品价格</td>
          <td><input type="text" name="price"
value="${itemsCustom.price }"/></td>
        </tr>
        <tr>
          <td>商品生产日期</td>
          <td><input type="text" name="createtime"
value="${itemsCustom.createtime}"/></td>
        </tr>
        <tr>
          <td>商品简介</td>
          <td><textarea rows="3" cols="30"
name="detail">${itemsCustom.detail }</textarea></td>
        </tr>
        <tr>
          <td colspan="2" align="center"><input type="submit" value="提交"
"/></td>
        </tr>
      </table>
    </form>
```



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

Controller 方法定义如下:

```
// 商品信息修改提交
@RequestMapping("/editItemsSubmit")
public ModelAndView editItemsSubmit(Integer id, ItemsCustom itemsCustom) throws
Exception {
    // 由页面提交的数据修改商品信息
    itemsService.updateItems(id, itemsCustom);
}
```

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、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:

```
/**
 * @ClassName: UserQueryVo.java
 * @Description: 这里对商品的复杂查询的查询条件进行封装
 * @author zxt
 * @Date: 2018年1月9日 上午9:09:47
 */
public class ItemsQueryVo {
    private ItemsCustom itemsCustom;

    public ItemsCustom getItemsCustom() {
        return itemsCustom;
    }

    public void setItemsCustom(ItemsCustom itemsCustom) {
        this.itemsCustom = itemsCustom;
    }
}
```

页面定义:

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

Controller 方法定义如下:

```
public String addItemsSubmit(Model model, ItemsQueryVo itemsQueryVo) throws Exception{
    System.out.println(itemsQueryVo.getItemsCustom());
}
```

## 6、参数绑定的问题

1、POST 提交中文乱码，在 web.xml 中加入：

```
<!-- 解决post提交的中文乱码问题 -->
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-cl
ass>
    <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>
```

2、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：

方法一：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"/>
```

方法二：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。

3、form 表单文件上传是，需要设置 **enctype=" multipart/form-data"**，将数据以二进制格式传输，此时会导致参数绑定失败。

解决方法：需要在 mvc 配置文件中如下配置

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

4、tomcat 服务器配置：在 tomcat 服务器 service.xml 配置文件中配置 maxPostSize="0" 会导致参数绑定失败。

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           maxPostSize="0"
           redirectPort="8443" />
```

解决方法：maxPostSize="0" 会导致 post 可传递的大小限制为 0。在某些版本中会有此问题，如果想要设置可以设置为 maxPostSize="-1"，表示不限制 post 参数。

## 7、自定义参数绑定

需求：根据业务需求自定义日期格式进行参数绑定。请求的日期是字符串格式，需要将其转换使之与 pojo 类中的日期类型保持一致。

```
public class Items {
    private Integer id;
    private String name;
    private Float price;
    private String pic;
    private Date createtime;
```

Converter: 自定义 Converter

```
/**
 * @ClassName: ItemsCustomDateConverter.java
 * @Description: 日期转换器
 * @author zxt
 * @Date: 2018年3月11日 下午8:50:52
 */
public class ItemsCustomDateConverter implements Converter<String, Date> {

    @Override
    public Date convert(String source) {
        // 将日期字符串转成日期类型
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("EEE MMM dd HH:mm:ss z yyyy", java.util.Locale.US);

        try {
            // 转换成功
            return simpleDateFormat.parse(source);
        } catch (ParseException e) {
```

```

        e.printStackTrace();
    }

    // 转换失败, 返回null
    return null;
}
}

```

## 7.1、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>
            <!-- 自定义的参数绑定转换器 -->
            <bean
class="com.zxt.ssm.controller.converter.ItemsCustomDateConverter"/>
        </list>
    </property>
</bean>

```

## 7.2、配置方式 2

```

<!--注解适配器 -->
<bean
    class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHand
lerAdapter">
    <property name="webBindingInitializer" ref="customBinder"></property>
</bean>

<!-- 自定义webBinder -->
<bean id="customBinder"
    class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer">
    <property name="conversionService" ref="conversionService" />
</bean>

<!-- conversionService -->
<bean id="conversionService"
    class="org.springframework.format.support.FormattingConversionServiceFactoryBea
n">

```

```
<!-- 转换器 -->
<property name="converters">
    <list>
        <bean class=" com.zxt.ssm.controller.converter.ItemsCustomDateConverter
"/>
    </list>
</property>
</bean>
```

## 八、高级参数绑定

### 1、包装类型的 pojo 绑定

1、需求：商品查询条件的传入

2、实现方法

第一种方法：在形参中，添加 `HttpServletRequest request` 参数，通过 `request` 接收查询条件；

第二种方法：在形参中，让包装类型的 `pojo` 接收查询条件。

分析：页面传参的特点，复杂，多样化。例如：用户账号、用户名称、订单信息等，如果将这些查询条件放在简单的 `pojo` 类型中，则 `pojo` 中的属性较多，比较乱。建议使用包装的 `pojo`（即 `pojo` 类的属性中含有 `pojo`）。

3、具体实现：

```
public class ItemsQueryVo {  
    private ItemsCustom itemsCustom;  
  
    public ItemsCustom getItemsCustom() {  
        return itemsCustom;  
    }  
  
    public void setItemsCustom(ItemsCustom itemsCustom) {  
        this.itemsCustom = itemsCustom;  
    }  
}
```

页面参数

```
<form name="itemsForm" action="<%=basePath%>items/itemsList.action" method="post">  
    查询条件: <input type="text" name="itemsCustom.name" />  
    <input type="submit" value="提交" />  
</form>
```

Controller

```
@RequestMapping("/itemsList")  
public ModelAndView itemsList(HttpServletRequest request, ItemsQueryVo itemsQueryVo)  
throws Exception {  
    // 商品列表，这些数据由service查询数据库得到  
    List<ItemsCustom> itemsList = itemsService.selectItems(itemsQueryVo);  
  
    // 创建ModelAndView 填充数据、设置视图  
    ModelAndView modelAndView = new ModelAndView();  
    // 填充数据
```

```

modelAndView.addObject("itemsList", itemsList);
// 设置视图
modelAndView.setViewName("items/itemsList");

return modelAndView;
}

```

## 2、集合类型的绑定

### 1、数组绑定

1、需求：商品的批量删除，页面中选择多个商品的 id，传入到 controller 的形参上，controller 通过数组来接收。

2、页面定义：

```

<:forEach items="${itemsList}" var="item" >
  <tr>
    <td><input type="checkbox" name="item_ids" value="${item.id }"/></td>
    <td>${item.id }</td>
    <td>${item.name }</td>
    <td>${item.price }</td>
    <td>${item.detail }</td>
    <td><a href="%basePath%items/editItems.action?id=${item.id }">修改</a></td>
  </tr>
</:forEach>

```

3、controller 方法形参接收

```

// 批量删除商品信息
@RequestMapping("/deleteItems")
public ModelAndView deleteItems(Integer[] item_ids) throws Exception {
    // 批量删除商品
    itemsService.deleteItems(item_ids);
}

```

4、相应的 Mapper 的批量删除实现

```

<!-- 批量删除商品 -->
<delete id="deleteItems" parameterType="Integer[]">
  delete from items
  <where>
    <!-- collection: 指定输入对象的集合属性名称, item: 每次遍历的中间变量
    open: 开始遍历时拼接的sql串, close: 结束遍历时拼接的串, separator: 遍历的两个对象中间需要拼接的串-->
    <!-- id in (.....) -->
    <foreach collection="array" item="id" open="id in (" separator="," close=")">
      #{id}
    </foreach>
  </where>
</delete>

```

## 2、List 集合绑定

1、通常在需要批量提交数据时，将批量提交的数据绑定到 `List<pojo>` 中。使用 `List` 接收页面信息时，必须将定义的 `List` 放在包装类中。

```
public class ItemsQueryVo {  
  
    private ItemsCustom itemsCustom;  
  
    // 批量商品信息  
    private List<ItemsCustom> itemsList;  
}
```

2、Controller 方法定义：进入批量商品修改页面，批量商品提交。

```
// 批量商品修改页面的展示  
@RequestMapping("/editQueryItems")  
public ModelAndView editQueryItems(ItemsQueryVo itemsQueryVo) throws Exception {  
    // 商品列表，这些数据由service查询数据库得到  
    List<ItemsCustom> itemsList = itemsService.selectItems(itemsQueryVo);  
  
    // 创建ModelAndView 填充数据、设置视图  
    ModelAndView modelAndView = new ModelAndView();  
    // 填充数据  
    modelAndView.addObject("itemsList", itemsList);  
    // 设置视图  
    modelAndView.setViewName("items/editItems");  
  
    return modelAndView;  
}  
  
// 批量商品修改页面的提交  
@RequestMapping("/editQueryItemsSubmit")  
// 通过itemsQueryVo接收批量提交的商品信息，将接收到的商品信息保存到itemsList属性中  
public ModelAndView editQueryItemsSubmit(ItemsQueryVo itemsQueryVo) throws Exception  
{  
  
}
```

3、页面定义

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



```

        <td><input type="text" name="itemsList[${status.index }].price"
value="${item.price }"/></td>
        <td><input type="text" name="itemsList[${status.index }].detail"
value="${item.detail }"/></td>
    </tr>
</c:forEach>

```

### 3、Map 绑定

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']"/></td>
    <td>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());
}

```

## 九、SpringMVC 拦截器

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。

拦截器定义，实现 HandlerInterceptor 接口，如下：

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

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 arg0, HttpServletResponse arg1, Object
arg2) throws Exception {
        // TODO Auto-generated method stub
        System.out.println("HandlerInterceptor1.preHandle..");
        return true;
    }

    /**
     * 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 arg0, HttpServletResponse arg1, Object
arg2, ModelAndView arg3)
        throws Exception {
        // TODO Auto-generated method stub
        System.out.println("HandlerInterceptor1.postHandle..");
    }

    /**
     * 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 arg0, HttpServletResponse arg1,
Object arg2, Exception arg3)
        throws Exception {
        // TODO Auto-generated method stub
        System.out.println("HandlerInterceptor1..afterCompletion..");
    }
}

```

## 1、拦截器配置

1、如果在 HandlerMapping 中配置拦截, 经过该 HandlerMapping 映射成功的 Handler 最终使用该拦截器。

```

<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="com.zxt.ssm.Interceptor.HandlerInterceptor1"/>
<bean id="handlerInterceptor2"
class="SpringMVC.intercapter.HandlerInterceptor2"/>

```

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>
        <!-- /**: 表示拦截所有url及其子url -->
        <mvc:mapping path="/**"/>
        <bean class="com.zxt.ssm.Interceptor.HandlerInterceptor1"/>
    </mvc:interceptor>
</mvc:interceptors>

```

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都返回成功调用, afterCompletion 只有 preHandle 返回 true 才调用。

## 十、SpringMVC 图片上传

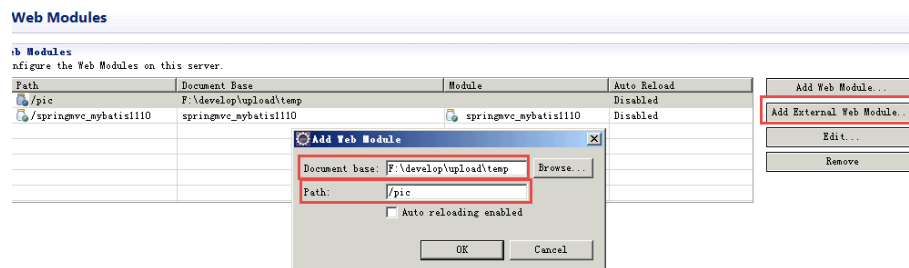
### 1、配置虚拟目录

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：



注意：在图片虚拟目录中，一定要将图片分级创建（提高 io 性能），例如按时间（年、月、日）分级等。

### 2、配置解析器

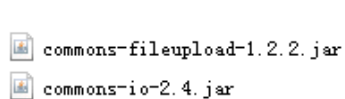
图片或者文件上传时，应该使用二进制格式，所以在页面表单提交中需要设置：enctype = “multipart / form-data”，需要 SpringMVC 对多部分类型的数据进行解析。此时需要在 SpringMVC.xml 中添加解析器（如果表单的类型是 multipart，一定要配置这个解析器，否则参数绑定失败，传参失败）如下：

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

### 3、jar 包

CommonsMultipartResolver 解析器依赖 commons-fileupload 和 commons-io，加

入如下 jar 包:



## 4、图片上传实例（使用虚拟目录）

### 4.1、controller

```
//商品修改提交
@RequestMapping("/editItemSubmit")
public String editItemSubmit(Items items, MultipartFile pictureFile)throws Exception{
    //原始文件名称
    String pictureFile_name = pictureFile.getOriginalFilename();
    //新文件名称
    String newFileName =
        UUID.randomUUID().toString()+pictureFile_name.substring(pictureFile_name.lastIndexOf("."));

    //上传图片
    File uploadPic = new java.io.File("F:/develop/upload/temp/"+newFileName);
    if(!uploadPic.exists()){
        uploadPic.mkdirs();
    }

    //向磁盘写文件
    pictureFile.transferTo(uploadPic);

    .....
}
```

### 4.2、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>

```

## 5、图片上传实例（提交至项目当前路径）

### 5.1、controller

```

// 商品信息修改提交
@RequestMapping("/editItemsSubmit")
public ModelAndView editItemsSubmit(Integer id, HttpServletRequest request,
ItemsCustom itemsCustom, MultipartFile picture) throws Exception {
    // 上传商品图片
    // 原始图片名称
    String originalFilename = picture.getOriginalFilename();
    if(picture != null && originalFilename != null && originalFilename.length() > 0)
    {
        // 存储图片的目录
        String realPath = request.getSession().getServletContext().getRealPath("/");

        // 上传文件的类型（后缀名）
        String type = originalFilename.substring(originalFilename.lastIndexOf("."));
        // 新的图片名称（使用UUID生成，避免重复）
        String newName = UUID.randomUUID() + type;

        File newFile = new File(realPath + "\\upload\\" + newName);
        // 将内存中的数据写入磁盘
        picture.transferTo(newFile);

        // 将图片名称写入到数据库中
        itemsCustom.setPic(newName);
    }

    // 由页面提交的数据修改商品信息
    itemsService.updateItems(id, itemsCustom);

    // 商品列表，这些数据由service查询数据库得到
    List<ItemsCustom> itemsList = itemsService.selectItems(null);

    // 创建ModelAndView 填充数据、设置视图

```

```
ModelAndView modelAndView = new ModelAndView();  
// 填充数据  
modelAndView.addObject("itemsList", itemsList);  
// 设置视图  
modelAndView.setViewName("items/itemsList");  
  
return modelAndView;  
}
```

## 十一、SpringMVC 开发 validation 校验


### 1、validation 校验

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。

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。

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。

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

### 3、配置 validator

```
<!-- 校验器 -->
<bean id="validator"
      class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"
      >
    <!-- 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.ReloadableResourceBundleMessageSource">
    <!-- 资源文件名-->
    <property name="basenames">
        <list>
            <value>classpath:CustomValidationMessages</value>
        </list>
    </property>
</bean>
```



```

        </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>

```

#### 4、将 validator 加到处理器适配器

##### 4.1、配置方式 1

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

##### 4.2、配置方式 2

```

<!-- 自定义webBinder -->
<bean id="customBinder"
    class="org.springframework.web.bind.support.ConfigurableWebBindingInitializer">
    <property name="validator" ref="validator" />
</bean>

```

```

<!-- 注解适配器 -->
<bean
    class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
    <property name="webBindingInitializer" ref="customBinder"></property>
</bean>

```

#### 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、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。

#### 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 BindingResult，一个 BindingResult 对应一个 pojo，且 BindingResult 放在 pojo 的后边。

#### 8、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"
    prefix="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>
```

## 2、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。

1、需求：商品修改提交只校验商品名称长度

2、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 {

}
```

3、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 })
```

## 3、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

## 4、数据回显

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。

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);
}
```

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：

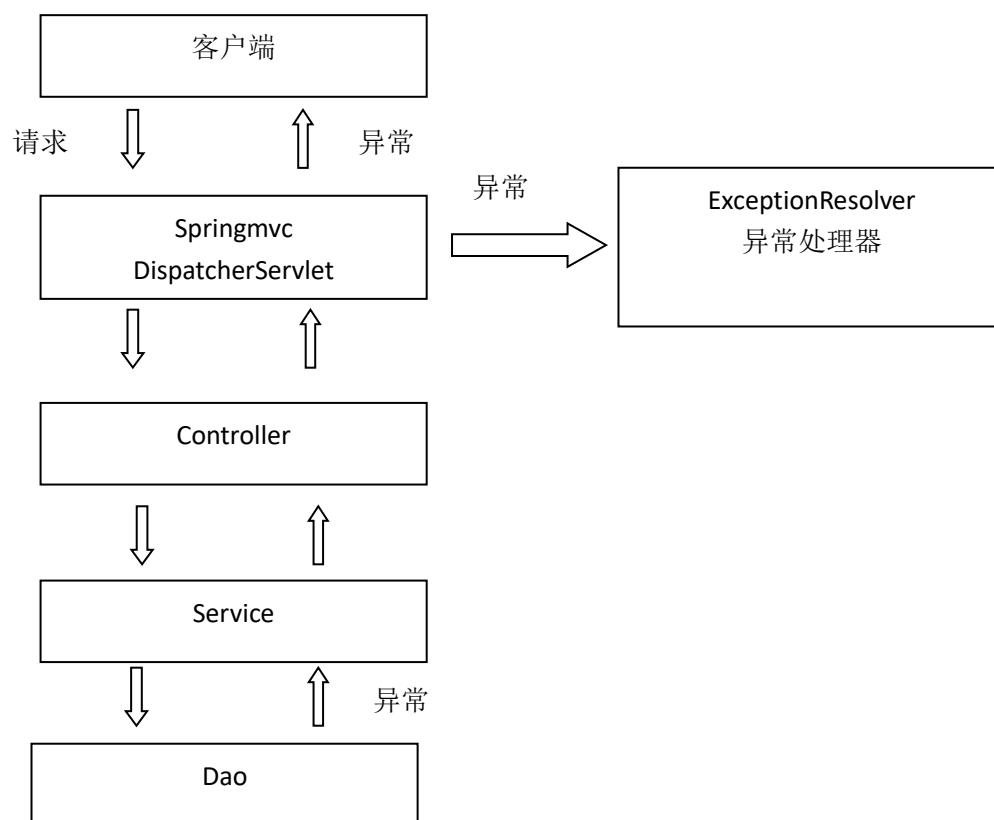
```
<select name="itemtype">
    <c:forEach items="${itemtypes }" var="itemtype">
        <option value="${itemtype.key }">${itemtype.value }</option>
    </c:forEach>
</select>
```

## 十二、SpringMVC 异常处理

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。

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：



### 1、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 {  
    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;
}
}

```

## 2、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;
    }
}

```



### 3、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>
```

### 4、异常处理器配置

在 springmvc.xml 中添加:

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

### 5、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。

### 十三、SpringMVC 与 struts2 的不同

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 `request` 域传输到页面。`Jsp` 视图解析器默认使用 `jstl`。

## 十四、Mybatis 分布式缓存机制

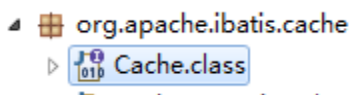
Redis 是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API。

不使用分布式缓存，缓存数据在各自的服务器中存储，不方便系统开发，所以要使用分布式缓存对缓存数据进行集中式管理。

mybatis 本身无法实现分布式缓存，需要和其他的分布式缓存进行整合。整合方法：mybatis 提供了一个 Cache 接口，如果要实现自己的缓存逻辑，只需要实现 Cache 接口即可。

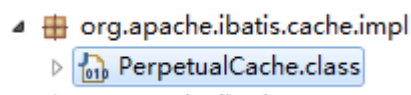
### 1、Mybatis 默认缓存机制

mybatis 提供二级缓存 Cache 接口，如下：



```
public interface Cache {  
  
    /**  
     * @return The identifier of this cache  
     */  
    String getId();  
  
    /**  
     * @param key Can be any object but usually it is a {@link CacheKey}  
     * @param value The result of a select.  
     */  
    void putObject(Object key, Object value);  
  
    /**  
     * @param key The key  
     * @return The object stored in the cache.  
     */  
    Object getObject(Object key);
```

它的默认实现类：



```

public class PerpetualCache implements Cache {

    private final String id;

    private Map<Object, Object> cache = new HashMap<Object, Object>();

    public PerpetualCache(String id) {
        this.id = id;
    }

    @Override
    public String getId() {
        return id;
    }
}

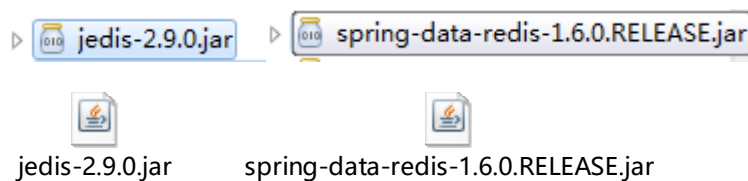
```

通过实现 Cache 接口可以实现 mybatis 缓存数据通过其它缓存数据库整合，mybatis 的特长是 sql 操作，缓存数据的管理不是 mybatis 的特长，为了提高缓存的性能可以将 mybatis 和第三方的缓存数据库整合，比如 ehcache、memcache、redis 等。

## 2、Mybatis 与 Redis 的整合

首先需要配置好 SSM 的开发环境，然后：

1、引入 jar 包。



2、在 Mybatis 的配置文件中开启缓存设置。

```

<settings>
    <!-- 全局映射器启用缓存 *主要将此属性设置完成即可 -->
    <setting name="cacheEnabled" value="true" />

    <!-- 查询时，关闭关联对象即时加载以提高性能 -->
    <setting name="lazyLoadingEnabled" value="false" />

    <!-- 对于未知的SQL查询，允许返回不同的结果集以达到通用的效果 -->
    <setting name="multipleResultSetsEnabled" value="true" />

    <!-- 设置关联对象加载的形态，此处为按需加载字段(加载字段由SQL指定)，不会加载关联表的所有
    字段，以提高性能 -->
    <setting name="aggressiveLazyLoading" value="true" />
</settings>

```

3、redis.properties。

```

# Redis settings

```

```

redis.host=127.0.0.1
redis.port=6379
redis.pass=

redis.maxIdle=300
redis.maxActive=600
redis.maxWait=1000
redis.testOnBorrow=true

```

#### 4、applicationContext-redis.xml。

```

<!-- 加载redis.properties文件中的内容，redis.properties文件中key命名要有一定的特殊规则 -->
-->
<context:property-placeholder location="classpath:db/redis.properties"
ignore-unresolvable="true"/>

<!-- redis数据源 -->
<bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">
    <property name="maxIdle" value="${redis.maxIdle}" />
    <property name="maxTotal" value="${redis.maxActive}" />
    <property name="maxWaitMillis" value="${redis.maxWait}" />
    <property name="testOnBorrow" value="${redis.testOnBorrow}" />
</bean>

<!-- Spring-redis连接池管理工厂 -->
<bean id="jedisConnectionFactory"
class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">
    <property name="hostName" value="${redis.host}" />
    <property name="port" value="${redis.port}" />
    <property name="password" value="${redis.pass}" />
    <property name="poolConfig" ref="poolConfig" />
</bean>

<!-- 使用中间类解决RedisCache.jedisConnectionFactory的静态注入，从而使MyBatis实现第三方缓存 -->
<!-- <bean id="redisCacheTransfer" class="com.redis.RedisCacheTransfer">
    <property name="jedisConnectionFactory" ref="jedisConnectionFactory" />
</bean> -->

```

#### 5、第三方内存数据库 Redis，定义一个 RedisCache 实现 org.apache.ibatis.cache.Cache 接口。

```

package com.redis;

import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

```

```

import org.apache.ibatis.cache.Cache;
import org.springframework.data.redis.connection.RedisConnection;
import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.serializer.JdkSerializationRedisSerializer;
import org.springframework.data.redis.serializer.RedisSerializer;

import redis.clients.jedis.exceptions.JedisConnectionException;

/**
 * @ClassName: RedisCache.java
 * @Description: 使用第三方内存数据库Redis作为二级缓存
 * @author zxt
 * @Date: 2018年3月29日 上午9:45:41
 */
public class RedisCache implements Cache {
    /**
     * 静态成员变量无法直接使用spring注解注入
     */
    private static JedisConnectionFactory jedisConnectionFactory;

    private final String id;

    /**
     * The {@code ReadWriteLock}.
     */
    private final ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

    public RedisCache(final String id) {
        if (id == null) {
            throw new IllegalArgumentException("Cache instances require an ID");
        }
        this.id = id;
    }

    public static void setJedisConnectionFactory(JedisConnectionFactory
jedisConnectionFactory) {
        RedisCache.jedisConnectionFactory = jedisConnectionFactory;
    }

    @Override
    public void clear() {
        RedisConnection connection = null;
        try {
            connection = jedisConnectionFactory.getConnection();

```

```

        connection.flushDb();
        connection.flushAll();

    } catch (JedisConnectionException e) {
        e.printStackTrace();
    } finally {
        if (connection != null) {
            connection.close();
        }
    }
}

@Override
public String getId() {
    return this.id;
}

@Override
public Object getObject(Object key) {
    Object result = null;
    RedisConnection connection = null;

    try {
        connection = jedisConnectionFactory.getConnection();
        RedisSerializer<Object> serializer = new
JdkSerializationRedisSerializer();
        result =
serializer.deserialize(connection.get(serializer.serialize(key)));

    } catch (JedisConnectionException e) {
        e.printStackTrace();
    } finally {
        if (connection != null) {
            connection.close();
        }
    }

    return result;
}

@Override
public void putObject(Object key, Object value) {

```

```

        RedisConnection connection = null;
        try {
            connection = jedisConnectionFactory.getConnection();
            RedisSerializer<Object> serializer = new
JdkSerializationRedisSerializer();
            connection.set(serializer.serialize(key), serializer.serialize(value));
        } catch (JedisConnectionException e) {
            e.printStackTrace();
        } finally {
            if (connection != null) {
                connection.close();
            }
        }
    }

    @Override
    public Object removeObject(Object key) {
        RedisConnection connection = null;
        Object result = null;
        try {
            connection = jedisConnectionFactory.getConnection();
            RedisSerializer<Object> serializer = new
JdkSerializationRedisSerializer();
            result = connection.expire(serializer.serialize(key), 0);
        } catch (JedisConnectionException e) {
            e.printStackTrace();
        } finally {
            if (connection != null) {
                connection.close();
            }
        }
        return result;
    }

    @Override
    public ReadWriteLock getReadWriteLock() {
        // TODO Auto-generated method stub
        return this.readWriteLock;
    }

    @Override
    public int getSize() {
        int result = 0;
        RedisConnection connection = null;

```



```

        try {
            connection = jedisConnectionFactory.getConnection();
            result = Integer.valueOf(connection.dbSize().toString());

        } catch (JedisConnectionException e) {
            e.printStackTrace();

        } finally {
            if (connection != null) {
                connection.close();
            }
        }
        return result;
    }
}

```

6、由于 Spring 无法注入静态成员变量，所以定义一个静态注入中间类。

```

package com.redis;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.stereotype.Component;

/**
 * @ClassName: RedisCacheTransfer.java
 * @Description: 静态注入中间类
 * @author zxt
 * @Date: 2018年3月29日 上午10:14:35
 */
@Component
public class RedisCacheTransfer {

    @Autowired
    public void setJedisConnectionFactory(JedisConnectionFactory
jedisConnectionFactory) {
        RedisCache.setJedisConnectionFactory(jedisConnectionFactory);
    }
}

```

7、UserMapper.xml。

```

<!-- Mapper.xml文件中的namespace与mapper接口的类路径相同。 -->
<mapper namespace="com.redis.mapper.UserMapper">
    <!-- 开启本mapper的namespace的缓存 -->
    <!-- type: 指定Cache接口的实现类 -->

```

```

<cache eviction="LRU" type="com.redis.RedisCache" />

<!-- 注册用户 -->
<insert id="registerUser" parameterType="com.redis.po.User">
    insert into user(username, password, tel) values(#{username}, #{password},
#{tel});
</insert>

<!-- 根据电话号码或者用户名查询用户 -->
<select id="checkUserUnique" parameterType="String"
resultType="com.redis.po.User">
    select * from user where ( tel = #{value} or username = #{value} )
</select>

<!-- 根据用户名（或者电话）和密码查找用户 -->
<select id="checkUser" parameterType="Map" resultType="com.redis.po.User">
    select * from user where (tel = #{username} or username = #{username}) and
password = #{password}
</select>

</mapper>

```

其中最重要的就是: `<cache eviction="LRU" type="com.redis.RedisCache" />`

## 8、UserController.java。

```

@Controller
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;

    @RequestMapping("/checkUser/{username}")
    public @ResponseBody User checkUser(@PathVariable("username") String username)
    throws Exception {
        return userService.checkUserUnique(username);
    }
}

```

第一次执行，从数据库查询数据，缓存命中率为0。

```

DEBUG [http-bio-8089-exec-2] - Cache Hit Ratio [com.redis.mapper.UserMapper]: 0.0
DEBUG [http-bio-8089-exec-2] - Fetching JDBC Connection from DataSource
DEBUG [http-bio-8089-exec-2] - JDBC Connection [jdbc:mysql://localhost:3306/ndimensions?characterEnc
DEBUG [http-bio-8089-exec-2] - ==> Preparing: select * from user where ( tel = ? or username = ? )
DEBUG [http-bio-8089-exec-2] - ==> Parameters: 18770705756(String), 18770705756(String)
DEBUG [http-bio-8089-exec-2] - <==          Total: 1
DEBUG [http-bio-8089-exec-2] - Closing non transactional SqlSession [org.apache.ibatis.session.defau
DEBUG [http-bio-8089-exec-2] - Returning JDBC Connection to DataSource
DEBUG [http-bio-8089-exec-2] - Written [com.redis.po.User@2a34d1e8] as "application/json;charset=UTF

```

后续以相同的参数访问，则从缓存中取数据。

```

DEBUG [http-bio-8089-exec-6] - Cache Hit Ratio [com.redis.mapper.UserMapper]: 0.5
DEBUG [http-bio-8089-exec-6] - Closing non transactional SqlSession [org.apache.ibatis.session.default]
DEBUG [http-bio-8089-exec-6] - Written [com.redis.po.User@5f30a037] as "application/json;charset=UTF-8"
DEBUG [http-bio-8089-exec-6] - Null ModelAndView returned to DispatcherServlet with name 'springmvc'
DEBUG [http-bio-8089-exec-6] - Successfully completed request
DEBUG [http-bio-8089-exec-6] - DispatcherServlet with name 'springmvc' processing GET request for [/user/checkUser/18770705756]
DEBUG [http-bio-8089-exec-6] - Looking up handler method for path /user/checkUser/18770705756
DEBUG [http-bio-8089-exec-6] - Returning handler method [public com.redis.po.User com.redis.controller.UserController.checkUser(java.lang.String,java.lang.String)]
DEBUG [http-bio-8089-exec-6] - Returning cached instance of singleton bean 'userController'
DEBUG [http-bio-8089-exec-6] - Last-Modified value for [/SSMRedis/user/checkUser/18770705756] is: -1
DEBUG [http-bio-8089-exec-6] - Creating a new SqlSession
DEBUG [http-bio-8089-exec-6] - SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@3be8b1d1]
DEBUG [http-bio-8089-exec-6] - Cache Hit Ratio [com.redis.mapper.UserMapper]: 0.6666666666666666
DEBUG [http-bio-8089-exec-6] - Closing non transactional SqlSession [org.apache.ibatis.session.default]
DEBUG [http-bio-8089-exec-6] - Written [com.redis.po.User@6a07004e] as "application/json;charset=UTF-8"
DEBUG [http-bio-8089-exec-6] - Null ModelAndView returned to DispatcherServlet with name 'springmvc'
DEBUG [http-bio-8089-exec-6] - Successfully completed request

```

### 3、Could not resolve placeholder 异常

```

at java.lang.IllegalArgumentException: Could not resolve placeholder 'redis.maxIdle' in value "${redis.maxIdle}"
by: java.lang.IllegalArgumentException: Could not resolve placeholder 'redis.maxIdle' in value "${redis.maxIdle}"
at org.springframework.util.PropertyPlaceholderHelper.parseStringValue(PropertyPlaceholderHelper.java:172)
at org.springframework.util.PropertyPlaceholderHelper.replacePlaceholders(PropertyPlaceholderHelper.java:124)
at org.springframework.core.env.AbstractPropertyResolver.doResolvePlaceholders(AbstractPropertyResolver.java:237)
at org.springframework.core.env.AbstractPropertyResolver.resolveRequiredPlaceholders(AbstractPropertyResolver.java:217)
at org.springframework.context.support.PropertySourcesPlaceholderConfigurer.lambda$processProperties$0(PropertySourcesPlaceholderConfigurer.java:115)
at org.springframework.beans.factory.config.BeanDefinitionVisitor.resolveStringValue(BeanDefinitionVisitor.java:36)
at org.springframework.beans.factory.config.BeanDefinitionVisitor.resolveValue(BeanDefinitionVisitor.java:217)
at org.springframework.beans.factory.config.BeanDefinitionVisitor.visitPropertyValues(BeanDefinitionVisitor.java:115)
at org.springframework.beans.factory.config.BeanDefinitionVisitor.visitBeanDefinition(BeanDefinitionVisitor.java:115)
at org.springframework.beans.factory.config.PlaceholderConfigurerSupport.doProcessProperties(PlaceholderConfigurerSupport.java:115)
... 18 more

```

那么出现异常信息的可能性有三种：

- 1、location 中的属性文件配置错误；
- 2、location 中定义的配置文件里面没有对应的 placeholder 值；
- 3、第三种就比较麻烦点，可能是 Spring 容器的配置问题。

Spring 容器采用反射扫描的发现机制，在探测到 Spring 容器中有一个 `org.springframework.beans.factory.config.PropertyPlaceholderConfigurer` 的 Bean 就会停止对剩余 `PropertyPlaceholderConfigurer` 的扫描（Spring 3.1 已经使用 `PropertySourcesPlaceholderConfigurer` 替代 `PropertyPlaceholderConfigurer` 了）。

而这个基于命名空间的配置，其实内部就是创建一个 `PropertyPlaceholderConfigurer` Bean 而已。换句话说，即 Spring 容器仅允许最多定义一个 `PropertyPlaceholderConfigurer`，其余的会被 Spring 忽略掉。

所以除去 properties 文件路径错误、拼写错误外，出现 "Could not resolve placeholder" 很有可能是使用了多个 `PropertyPlaceholderConfigurer` 或者多个 `<context:property-placeholder>` 的原因。

比如我有一个 `dao.xml` 读取 `db.properties`，还有一个 `redis.xml` 读取

redis.properties，然后 web.xml 统一 load 这两个 xml 文件，如果这两个 xml 文件中分别有：

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

那么，一定会出 "Could not resolve placeholder"。

一定要记住，不管是在一个 Spring 文件还是在多个 Spring 文件被统一 load 的情况下，直接写：

```
<context:property-placeholder location="" />
```

```
<context:property-placeholder location="" />
```

是不允许的。

解决方案：在 Spring 3.0 中，可以写：

```
<context:property-placeholder      location="classpath:db/db.properties"
ignore-unresolvable="true"/>
<context:property-placeholder      location="classpath:db/redis.properties"
ignore-unresolvable="true"/>
```

注意两个都要加上 ignore-unresolvable="true"，一个加另一个不加也是不行的。

## 4、mybatis 整合 ehcache

ehCache 是一个纯 Java 的进程内缓存框架，是一种广泛使用的开源 Java 分布式缓存，具有快速、精干等特点，是 Hibernate 中默认的 CacheProvider。

### 4.1、mybatis 整合 ehcache 方法

1、引入 jar 包。



2、配置 cache 中 type 为 ehcache 实现 cache 接口的类型。

```
<!-- 开启本mapper的namespace的二级缓存 -->
<!-- type: 指定Cache接口的实现类的类型，mybatis默认使用PerpetualCache
要和ehcache整合，需要配置type为ehcache实现的cache接口的类型 -->
<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>
```

### 3、classpath 下添加配置文件：ehcache.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">
  <!-- <diskStore path="F:\develop\ehcache" /> -->
  <defaultCache
    maxElementsInMemory="1000"
    maxElementsOnDisk="10000000"
    eternal="false"
    overflowToDisk="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    diskExpiryThreadIntervalSeconds="120"
    memoryStoreEvictionPolicy="LRU">
  </defaultCache>
</ehcache>
```

属性说明：

**diskStore**：指定数据在磁盘中的存储位置。

**defaultCache**：当借助 `CacheManager.add("demoCache")` 创建 Cache 时，EhCache 便会采用 `<defaultCache/>` 指定的管理策略。

以下属性是必须的：

**maxElementsInMemory** - 在内存中缓存的 **element** 的最大数目；

**maxElementsOnDisk** - 在磁盘上缓存的 **element** 的最大数目，若是 0 表示无穷大；

**eternal** - 设定缓存的 **elements** 是否永远不过期。如果为 **true**，则缓存的数据始终有效，如果为 **false** 那么还要根据 **timeToIdleSeconds**，**timeToLiveSeconds** 判断；

**overflowToDisk** - 设定当内存缓存溢出的时候是否将过期的 **element** 缓存到磁盘上。

以下属性是可选的：

**timeToIdleSeconds** - 当缓存在 EhCache 中的数据前后两次访问的时间超过 **timeToIdleSeconds** 的属性取值时，这些数据便会删除，默认值是 0，也就是可闲置时间无穷大；

**timeToLiveSeconds** - 缓存 **element** 的有效生命期，默认是 0，也就是 **element** 存活时间无穷大；

**diskSpoolBufferSizeMB** 这个参数设置 **DiskStore**(磁盘缓存)的缓存区大小，默认是 30MB，每个 Cache 都应该有自己的一个缓冲区。

**diskPersistent** - 在 VM 重启的时候是否启用磁盘保存 EhCache 中的数据，默认是 **false**。

**diskExpiryThreadIntervalSeconds** - 磁盘缓存的清理线程运行间隔，默认是 120 秒。每个 120s，相应的线程会进行一次 EhCache 中数据的清理工作

**memoryStoreEvictionPolicy** - 当内存缓存达到最大，有新的 **element** 加入的时候，移除缓存中 **element** 的策略。默认是 LRU（最近最少使用），可选的有 LFU（最不常使用）和 FIFO（先进先出）。

4、还可以根据需求调整如下参数：

```
<cache type="org.mybatis.caches.ehcache.EhcacheCache" >
  <property name="timeToIdleSeconds" value="3600"/>
  <property name="timeToLiveSeconds" value="3600"/>
  <!-- 同ehcache参数maxElementsInMemory -->
  <property name="maxEntriesLocalHeap" value="1000"/>
  <!-- 同ehcache参数maxElementsOnDisk -->
  <property name="maxEntriesLocalDisk" value="10000000"/>
  <property name="memoryStoreEvictionPolicy" value="LRU"/>
</cache>
```

## 十五、SSM+RESTful 开发

### 1、什么是 RESTful?

RESTful 架构，就是目前最流行的一种互联网软件架构。它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用。RESTful（即 Representational State Transfer 的缩写）是一种开发理念，是对于 http 的很好的诠释。

对 url 进行规范，写成 RESTful 格式的 url：

非 REST 的 url: `http://.../queryItems.action?id=001&type=T01`

REST 的 url 风格: `http://....../items/001`

特点: url 简洁，将参数通过 url 传入服务端进行处理。

RESTful 对 http 的方法进行规范，不管是删除、添加、更新，使用的 url 是一致的，若要进行删除，需要设置 http 的方法为 delete, put, post 等；

后台 Controller 方法: 判断 http 方法，若为 delete 则执行删除，post 执行添加等操作；

对 http 的 contentType 进行规范，请求时指定 contentType，要 json 数据，设置成 json 格式的 type 等。

### 2、SSM 框架 + RESTful 开发

#### 2.1、搭建 SSM 环境

1、需要在项目中添加将结果转为 json 数据返回的 jar 包。

jackson-annotations-2.8.4.jar

jackson-core-2.8.4.jar

jackson-databind-2.8.4.jar

如果不添加这 3 个 jar 包，会输出错误如下：

`java.lang.IllegalArgumentException: No converter found for return value of type。`



jackson-2.9.4.jar

jackson-2.9.4.jar 包将上述的 3 个 jar 包整合在一起。

## 2、SpringMVC 前端控制器的配置。

```
<!-- SpringMVC配置前端控制器 -->
<servlet>
    <servlet-name>SpringMVC</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

<!-- contextConfigLocation配置SpringMVC加载的配置文件（配置文件里面配置处理器适配器，
处理器等等），如果不配置contextConfigLocation，则默认加载/WEB-INF/servlet名称-servlet.xml
（SpringMVC-servlet.xml）-->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring/SpringMVC.xml</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>SpringMVC</servlet-name>
    <!-- 第一种： *.action，所有以*.action结尾的访问，都由DispatcherServlet解析；
        第二种： / ，所有访问的地址都由DispatcherServlet解析，对于这种配置会出现一个问题，
        那就是静态文件的访问不需要DispatcherServlet进行解析（因为访问静态文件直接返回即可，不用再由
        处理器处理）。但是这种方式可以实现RESTful风格的url；
        第三种： /* ，这种配置不对，使用这种配置，最终要转发到一个jsp页面时，仍然会由
        DispatcherServlet解析jsp的地址，不能根据jsp的页面找到Handler，会报错-->
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

注意：要实现 RESTful 风格的 url 则需要配置： `<url-pattern>/</url-pattern>`。

同时需要解决静态资源直接返回，不需要解析的问题，需要在 SpringMVC.xml 中添加资源映射：

```
<!-- 静态资源映射 -->

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

或者也可以在 SpringMVC.xml 配置文件中使用 `<mvc:default-servlet-handler/>` 配置该指令放行默认的静态资源。

## 2.2、应用实例

1、需求：查询商品信息，返回 json 数据。



编写 mapper 和 service 接口以及实现类

ItemsMapper.xml

```
<!-- 根据商品id查询商品信息 -->
<select id="selectItemsById" parameterType="int" resultType="com.zxt.ssm.po.Items">
    select * from items where id=#{id}
</select>
```

ItemsMapper.java

```
// 根据商品id获取商品信息
public Items selectItemsById(int id) throws Exception;
```

ItemsService 接口

```
/**
 * @Description: 根据商品id获取商品信息
 * @param id: 使用int无法判断是否为空，所以一般使用Integer
 */
public ItemsCustom selectItemsById(Integer id) throws Exception;
```

ItemsServiceImpl 添加接口实现

```
@Override
public ItemsCustom selectItemsById(Integer id) throws Exception {
    Items items = itemsMapper.selectItemsById(id);

    /**
     * 中间对商品信息进行业务处理，返回Items的扩展类: ItemsCustom
     * 例如商品类中只有生产日期，而没有是否过期的信息，可以做一定的操作得到该信息，封装到
     ItemsCustom，显示给用户
     */
    ItemsCustom itemsCustom = new ItemsCustom();
    // 将Items中的属性复制到ItemsCustom中
    BeanUtils.copyProperties(items, itemsCustom);

    return itemsCustom;
}
```

## 2、controller

定义一个方法：使用 url 模板映射，进行 url 映射，使用 REST 风格的 url，能将查询商品信息的 id 传入 controller，输出 json，使用@ResponseBody 将 java 对象输出为 json 至前端页面。

```
@Controller
public class TestController {
    @Autowired
    private ItemsService itemsService;
```

```

@RequestMapping("/itemsview/{id}")
public @ResponseBody ItemsCustom findItemsById(@PathVariable("id") Integer id)
throws Exception {
    ItemsCustom itemsCustom = itemsService.selectItemsById(id);
    return itemsCustom;
}
}

```

"/itemsview/{id}"这里的 {id} 传入到(@PathVariable("id") Integer id) 的 id 里面。

如果有多个参数，例如增加 name 字段：

```

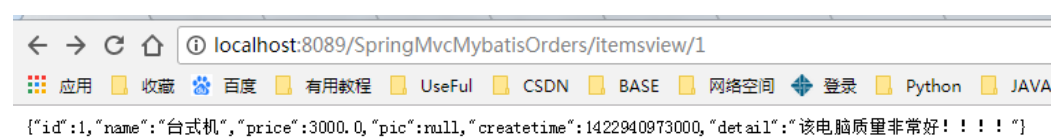
@RequestMapping("/itemsview/{id}/{name}")

(@PathVariable("id") Integer id , @PathVariable("name") String
diffname ), return itemsCustom 会经过@ResponseBody 注解转换为 json 数据格式。

```

3、测试，浏览器输入：

http://localhost:8080/SpringMvcMybatisOrders/itemsview/1



## 3、常见问题总结

### 3.1、数据转换

Java 中没有将数据转换成 JSON 格式的 jar 包，所以需要依赖外部 jackson 包，否则会出现：**java.lang.IllegalArgumentException** 错误。

### 3.2、静态资源访问处理

采用 RESTful 架构后，必须将 web.xml 中控制器拦截的请求设置为 ‘ / ’；但是这样会产生一个问题，就是会将 css, js, 图片等静态资源拦截，发生 404 错误。

解决方案如下：

方法一：配置<mvc:resources/>

SpringMVC 配置文件中这样使用:<mvc:resources mapping="请求 URI" location="资源位置">。

方法二：在 SpringMVC 配置文件中<use>使用<mvc:default-servlet-handler/> 配置该

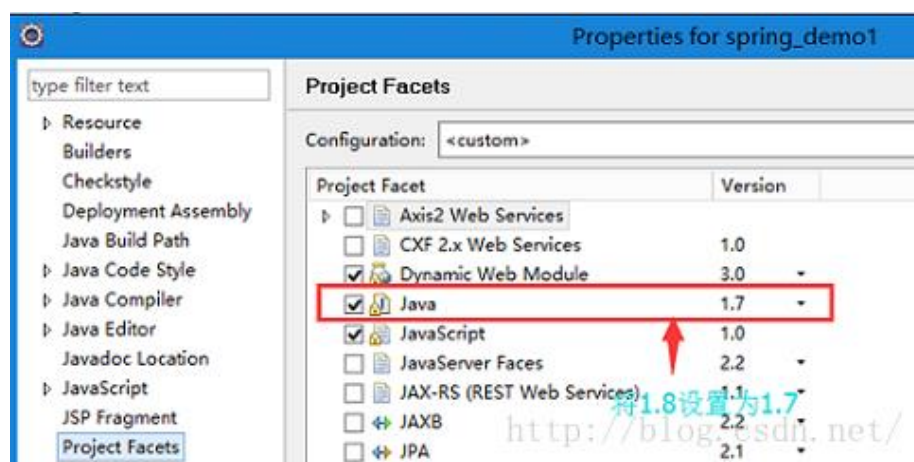
指令放行默认的静态资源。

### 3.3、版本不兼容问题

当出现 spring 版本和 java jdk 版本不兼容时也会出现：  
`java.lang.IllegalArgumentException` 异常。

通常来说 spring3.x 版本对应 JDK7.0 及以下版本，使用 spring4.x 以后版本，可以使用 JDK8.0。

右击项目 ---> properties --->project facets：修改 JDK 版本，需要将 1.8 降为 1.7 版本。



## 4、SSM 中常用注解

### 4.1、@RequestMapping（见四 5）

### 4.2、@PathVariable

@PathVariable 映射 URL 绑定的占位符，带占位符的 URL 是 Spring3.0 新增的功能，该功能在 SpringMVC 向 REST 目标挺进发展过程中具有里程碑的意义。通过 @PathVariable 可以将 URL 中占位符参数绑定到控制器处理方法的入参中：URL 中的 {xxx} 占位符可以通过 @PathVariable(“xxx”) 绑定到操作方法的入参中。

```
/**
 * @RequestMapping(value = "user/login/{id}/{name}/{status}") 中的 {id}/{name}/{status}
 * 与 @PathVariable int id、@PathVariable String name、@PathVariable boolean status
 * 一一对应，按名匹配。
 */
@RequestMapping(value = "user/login/{id}/{name}/{status}")
```

```
@ResponseBody
// @PathVariable 注解下的数据类型均可用
public User login(@PathVariable int id, @PathVariable String name, @PathVariable boolean
status) {
    // 返回一个 User 对象响应 ajax 的请求
    return new User(id, name, status);
}
```

### 4.3、@ResponseBody

`@ResponseBody` 注解表示该方法的返回的结果直接写入 HTTP 响应正文（`ResponseBody`）中，一般在异步获取数据时使用，通常是在使用 `@RequestMapping` 后，返回值通常解析为跳转路径，加上 `@ResponseBody` 后返回结果不会被解析为跳转路径，而是直接写入 HTTP 响应正文中。

作用：该注解用于将 `Controller` 的方法返回的对象，通过适当的 `HttpMessageConverter` 转换为指定格式后，写入到 `Response` 对象的 `body` 数据区。

使用时机：返回的数据不是 `html` 标签的页面，而是其他某种格式的数据时（如 `json`、`xml` 等）使用。

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。

### 4.4、@RequestBody

`@RequestBody` 注解则是将 HTTP 请求正文插入方法中，使用适合的 `HttpMessageConverter` 将请求体写入某个对象。

`@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 `Java` 对象，并绑定到 `controller` 方法的参数上。

作用：1)、该注解用于读取 `Request` 请求的 `body` 部分数据，使用系统默认配置的 `HttpMessageConverter` 进行解析，然后把相应的数据绑定到要返回的对象上；

2)、再把 `HttpMessageConverter` 返回的对象数据绑定到 `controller` 中方法的参数上。

使用时机：

A)、GET、POST 方式提时，根据 request header Content-Type 的值来判断：

application/x-www-form-urlencoded，可选（即非必须，因为这种情况的数据 @RequestParam，@ModelAttribute 也可以处理，当然 @RequestBody 也能处理）；

multipart/form-data，不能处理（即使用 @RequestBody 不能处理这种格式的数据）；

其他格式，必须（其他格式包括 application/json，application/xml 等。这些格式的数据，必须使用 @RequestBody 来处理）；

B)、PUT 方式提交时，根据 request header Content-Type 的值来判断：

application/x-www-form-urlencoded，必须；multipart/form-data，不能处理；其他格式，必须；

说明：request 的 body 部分的数据编码格式由 header 部分的 Content-Type 指定。

```
@RequestMapping(value = "user/login")
@ResponseBody
// 将 ajax (datas) 发出的请求写入 User 对象中
public User login(@RequestBody User user) {
    // 这样就不会再被解析为跳转路径，而是直接将 user 对象写入 HTTP 响应正文中
    return user;
}
```

#### 4.5、@RequestParam

使用 @RequestParam 常用于处理简单类型的绑定。当形参名称与页面提交的参数的名称不一致时就要使用这个注解。

**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。

```
// 商品修改页面的展示
@RequestMapping("/editItems")
public ModelAndView editItems(@RequestParam(value = "id", required = true) Integer item_id) throws Exception {
}
}
```

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

注意：如果请求参数中没有 `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 `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 `id` 参数值。