# SpringMVC介绍

## 三层结构介绍

\*\*\*我们的开发架构一般都是基于两种形式，一种是 C/S 架构，也就是客户端/服务器，另一种是 **B/S 架构**，也就是浏览器服务器。在 JavaEE 开发中，几乎全都是基于 B/S 架构的开发。那么在 B/S 架构中，系统标准的三层架构包括：**表现层、业务层、持久层**。三层架构在我们的实际开发中使用的非常多，所以我们课程中的案例也都是基于三层架构设计的。

\*\*\*三层架构中，每一层各司其职，接下来我们就说说每层都负责哪些方面：

\*\*\***表现层**：

\* 也就是我们常说的web 层。它负责接收客户端请求，向客户端响应结果，通常客户端使用http 协议请求web 层，web 需要接收 http 请求，完成 http 响应。

\* 表现层包括展示层和控制层：控制层负责接收请求，展示层负责结果的展示。

\* 表现层依赖业务层，接收到客户端请求一般会调用业务层进行业务处理，并将处理结果响应给客户端。

\* 表现层的设计一般都使用 MVC 模型。（MVC 是表现层的设计模型，和其他层没有关系）

\*\*\***业务层**：

\* 也就是我们常说的 service 层。它负责业务逻辑处理，和我们开发项目的需求息息相关。web 层依赖业务层，但是业务层不依赖 web 层。

\* 业务层在业务处理时可能会依赖持久层，如果要对数据持久化需要保证事务一致性。（也就是我们说的， 事务应该放到业务层来控制）

\*\*\***持久层**：

\* 也就是我们是常说的 dao 层。负责数据持久化，包括数据层即数据库和数据访问层，数据库是对数据进行持久化的载体，数据访问层是业务层和持久层交互的接口，业务层需要通过数据访问层将数据持久化到数据库中。通俗的讲，持久层就是和数据库交互，对数据库表进行曾删改查的。

## MVC设计模式介绍

\*\*\***MVC** 全名是 Model View Controller，是**模型(model)－视图(view)－控制器(controller)**的缩写， 是一种用于设计创建 Web 应用程序表现层的模式。MVC 中每个部分各司其职：

\*\*\*Model（模型）：

模型包含业务模型和数据模型，数据模型用于封装数据，业务模型用于处理业务。

\*\*\*View（视图）：

通常指的就是我们的 jsp 或者 html。作用一般就是展示数据的。

通常视图是依据模型数据创建的。

\*\*\*Controller（控制器）：

是应用程序中处理用户交互的部分。作用一般就是处理程序逻辑的。

## SpringMVC介绍

* **Spring MVC是什么？**

\*\*\*SpringMVC 是一种基于 Java 的实现 MVC 设计模型的请求驱动类型的轻量级 Web 框架，属于 SpringFrameWork 的后续产品，已经融合在 Spring Web Flow 里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 Spring 可插入的 MVC 架构，从而在使用 Spring 进行 WEB 开发时，可以选择使用 Spring 的 Spring MVC 框架或集成其他 MVC 开发框架，如 Struts1(现在一般不用)，Struts2 等。

\*\*\*SpringMVC 已经成为*目前最主流的 MVC 框架*之一，并且**随着 Spring3.0 的发布，全面超越 Struts2，成为最优秀的 MVC 框架。**

\*\*\*它通过一套注解，让一个简单的 Java 类成为处理请求的控制器，而无须实现任何接口。同时它还支持RESTful 编程风格的请求。

**总之：**

Spring MVC和Struts2一样，都是**为了解决表现层问题**的web框架，它们都是基于MCC设计模式的。而这些表现层框架的主要职责就是**处理前端HTTP请求**。

* **Spring MVC由来？**

Spring MVC全名叫Spring Web MVC，它是Spring家族Web模块的一个重要成员。这一点,我们可以从Spring的整体结构中看得出来：

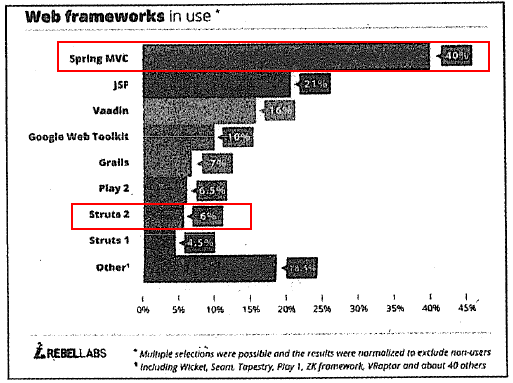


* **为什么学习SpringMVC？**

也许你要问，为什么要学习Spring MVC呢？struts2不才是主流吗？看SSH的概念有多火？

其实很多初学者混淆了一个概念，SSH实际上指的是Struts1.x+Spring+Hibernate。这个概念已经有十几年的历史了。在Struts1.x时代，它是当之无愧的霸主，但是在新的MVC框架涌现的时代，形式已经不是这样了，Struts2.x借助了Struts1.x的好名声，让国内开发人员认为Struts2.x是霸主继任者（其实两者在技术上无任何关系），导致国内程序员大多数学习基于Struts2.x的框架，又一个貌似很火的概念出来了S2SH（Struts2+Spring+Hibernate）整合开发。

不要再被蒙蔽了，看看下面的调查统计吧：



**SpringMVC的市场占有率是40%**，而Struts2只有可怜的6%。这已然说明了学习SpringMVC的必要性了，再说了，SpringMVC本身就是spring家族的一员，与整合spring时，SpringMVC根本无需中间整合包，而struts2得需要。

既然已经知道了SpringMVC的重要性了，那么下面就跟着我一起看看它的神奇之处吧！

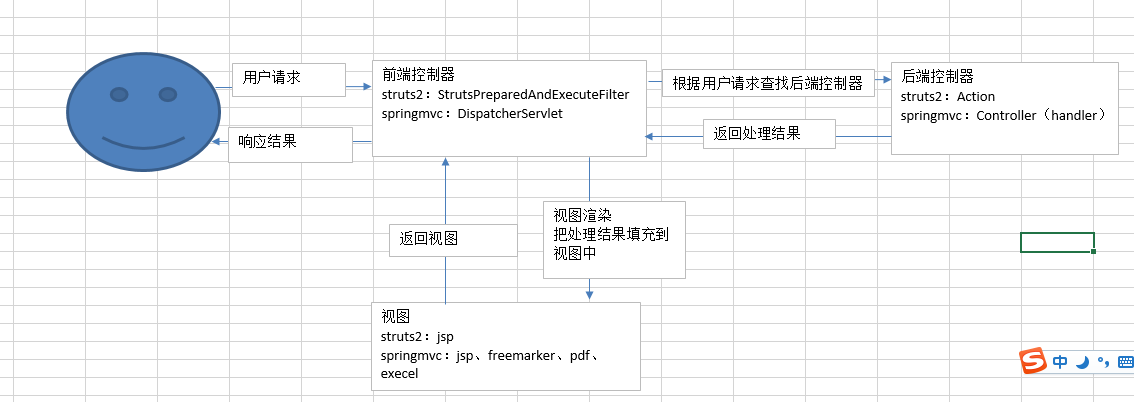
*接下来，让我们一起看看SpringMVC是如何处理HTTP请求的？*

* **SpringMVC如何处理请求？**

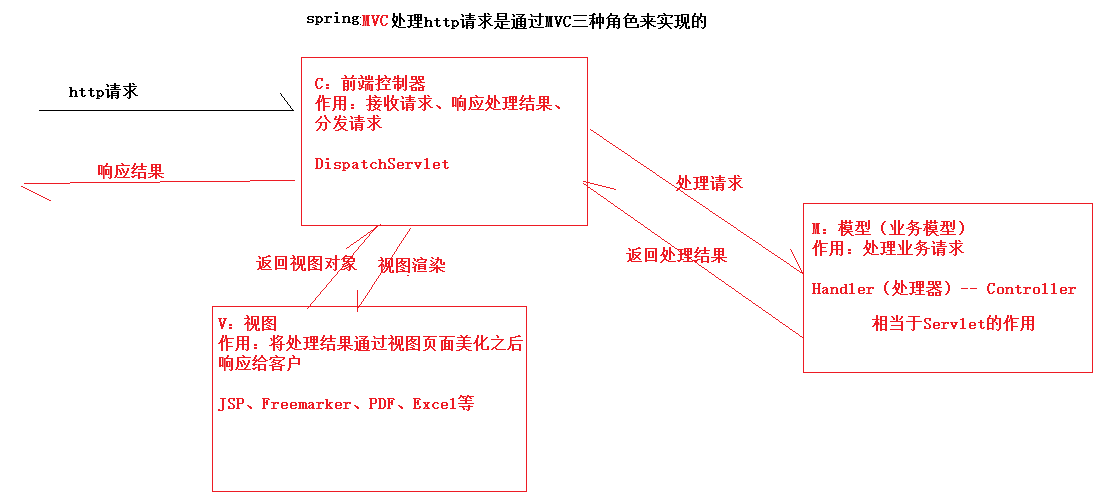
*SpringMVC是基于MVC设计模式的，MVC模式指的就是Model（业务模型）、View（视图）、Controller（控制器）*。SpringMVC处理请求就是通过MVC这三种角色来实现的。

不过千万不要把MVC设计模式和工程的三层结构混淆，三层结构指的是表现层、业务层、数据持久层。而MVC只针对表现层进行设计。

下面，让我们来看看具体的处理流程吧！



流程来学校已经解除



# 入门程序

## 学习目标

* 如何配置前端控制器
* 如何开发处理器（方式有多种）

## 需求

访问/queryItem,返回商品列表页面，商品数据暂时使用静态数据（不从数据库查询）。

## 开发环境

本案例使用开发环境：

Jdk：jdk1.7.0\_72

IDE：mars 2

Tomcat：apache-tomcat-7.0.53

Springmvc：4.1.3

## 开发步骤

### 第一步：创建一个web工程

工程名称为：springmvc

### 第二步：编写POM文件，添加依赖

### 第三步：配置前端控制器

在web.xml中添加DispatcherServlet的配置。

|  |
| --- |
| <!-- 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>  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <!-- 此处可以多种配置方式： 第一种：\*.do，以.do结尾的url进行访问时由DispatcherServlet解析 第二种：/，所有访问的URL都由DispatcherServlet解析，对于静态文件的解析需要配置不让DispatcherServlet进行解析  使用此种方式，可以实现RESTful风格的url 第三种：/\*，经测试，这种配置有问题。 如果请求或者通过Controller转发jsp页面时，也会被拦截，此时找不到handler，会报错 -->  <url-pattern>\*.do</url-pattern>  </servlet-mapping> |

### 第四步：创建springmvc.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*  *http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>    </beans> |

### 第五步：编写处理器

|  |
| --- |
| ***要点：***  ***处理器开发方式有多种：实现HttpRequestHandler接口、实现Controller接口、注解方式等。***  ***不过企业开发中，推荐使用注解方式开发处理器，本课程讲解的就是注解方式。*** |

*注解方式使用到的注解主要有以下两个：*

**@Controller**注解：在类上添加该注解，指定该类为一个请求处理器，不需要实现任何接口或者继承任何类。

**@RequestMapping**注解：在方法上添加该注解，指定请求的url由该方法处理。其中“.do”可以加也可以不加。

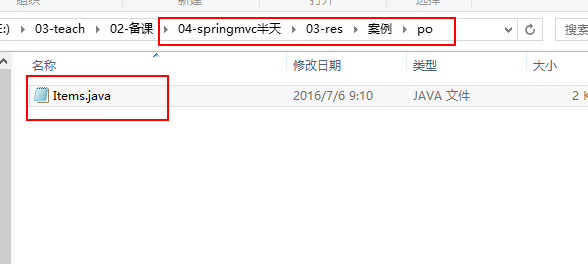
*处理器的返回值是ModelAndView对象，该对象的具体理解如下：*

**ModelAndView**：方法返回值对象，该对象包含两个功能：一个是将数据存储到Request域中，一个是设置响应视图，比如将视图设置为“/WEB-INF/jsp/itemList.jsp”

*处理器代码：*

|  |
| --- |
| @Controller  **public** **class** ItemController {  @RequestMapping("/queryItem")  **public** ModelAndView queryItem() **throws** Exception {    List<Item> itemList = **new** ArrayList<>();    //商品列表  Item Item\_1 = **new** Item();  item\_1.setName("联想笔记本\_3");  item\_1.setPrice(6000f);  item\_1.setDetail("ThinkPad T430 联想笔记本电脑！");    Item item\_2 = **new** Item();  item\_2.setName("苹果手机");  item\_2.setPrice(5000f);  item\_2.setDetail("iphone6苹果手机！");    itemList.add(item\_1);  itemList.add(item\_2);  //创建modelandView对象  ModelAndView modelAndView = **new** ModelAndView();  //添加model  modelAndView.addObject("itemList", itemList);  //添加视图  modelAndView.setViewName("/WEB-INF/jsp/itemList.jsp");  **return** modelAndView;  }  } |

**注意：商品数据使用Item类描述，可以使用参考资料中提供的pojo类。**



### 第六步：配置处理器

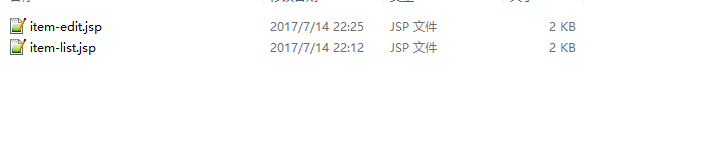
使用组件扫描器省去在spring容器配置每个controller类的繁琐。

使用<context:component-scan>自动扫描标记@controller的控制器类，在springmvc.xml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.datang.springmvc.controller"*/>

### 第七步：创建视图



把参考资料中的itemList.jsp复制到工程的/WEB-INF/jsp目录下。

### 第八步：访问测试

访问地址：<http://localhost:8080/day57_springmvc/queryItem.d>o

## 开发Handler的其他方式

### 实现HttpRequestHandler接口

|  |
| --- |
| //第二种Handler处理器的编写方式：实现HttpRequestHandler接口  **public** **class** ItemsController2 **implements** HttpRequestHandler {  @Override  **public** **void** handleRequest(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  List<Items> itemsList = **new** ArrayList<Items>();  Items items\_1 = **new** Items();  items\_1.setName("联想笔记本");  items\_1.setPrice(6000f);  items\_1.setDetail("ThinkPad T430 联想笔记本电脑！");  Items items\_2 = **new** Items();  items\_2.setName("苹果手机");  items\_2.setPrice(5000f);  items\_2.setDetail("iphone6苹果手机！");  itemsList.add(items\_1);  itemsList.add(items\_2);    //数据填充  request.setAttribute("itemsList", itemsList);    request.getRequestDispatcher("/WEB-INF/jsp/items/itemsList.jsp").forward(request, response);  }  } |

### 实现Controller接口

开发时需要实现Controller接口，该接口类只有一个接口方法。

|  |
| --- |
| //第一种方式：实现Controller接口  **public** **class** ItemsController **implements** Controller {  @Override  **public** ModelAndView handleRequest(HttpServletRequest req,  HttpServletResponse resp) **throws** Exception {  // **TODO** Auto-generated method stub  List<Items> itemsList = **new** ArrayList<Items>();  Items items\_1 = **new** Items();  items\_1.setName("联想笔记本");  items\_1.setPrice(6000f);  items\_1.setDetail("ThinkPad T430 联想笔记本电脑！");  Items items\_2 = **new** Items();  items\_2.setName("苹果手机");  items\_2.setPrice(5000f);  items\_2.setDetail("iphone6苹果手机！");  itemsList.add(items\_1);  itemsList.add(items\_2);  // 创建ModelAndView准备填充数据、设置视图  ModelAndView modelAndView = **new** ModelAndView();  // 相当于request.setAttribute(key,value)  modelAndView.addObject("itemsList", itemsList);  // 指定视图的地址  modelAndView.setViewName("WEB-INF/jsp/items/itemsList.jsp");  **return** modelAndView;  }  } |

## 非注解方式的处理器映射器和适配器

**如果在DispatcherServlet.properties文件中出现的处理器适配器和处理器映射器，如果没有特殊要求（比如在适配器中配置转换器、在映射器中配置拦截器），那么就不需要再单独进行配置了。**

### HttpRequestHandlerAdapter

HttpRequestHandlerAdapter，http请求处理器适配器，所有实现了

org.springframework.web.HttpRequestHandler接口的处理器，可以通过此适配器适配、执行。

#### Handler实现

该适配器只能适配执行实现了HttpRequestHandler接口的handler类

|  |
| --- |
| **public** **class** ItemsController1 **implements** HttpRequestHandler {  @Override  **public** **void** handleRequest(HttpServletRequest request,  HttpServletResponse response) **throws** ServletException, IOException {  // 查询数据库获取商品列表，此处用静态数据代替，与mybatis整合后再真正查询数据库  List<Items> itemsList = **new** ArrayList<Items>();  Items items\_1 = **new** Items();  items\_1.setName("联想笔记本");  items\_1.setPrice(6000f);  items\_1.setDetail("ThinkPad T430 联想笔记本电脑！");  Items items\_2 = **new** Items();  items\_2.setName("苹果手机");  items\_2.setPrice(5000f);  items\_2.setDetail("iphone6苹果手机！");  itemsList.add(items\_1);  itemsList.add(items\_2);    //填充数据到request域  request.setAttribute("itemsList", itemsList);    //转发视图  request.getRequestDispatcher("/WEB-INF/jsp/items/itemsList.jsp").forward(request, response);  }  } |

#### 适配器配置

1、采取默认配置（DispatcherServlet.properties）

2、可以在springmvc.xml中进行显式配置：

|  |
| --- |
| <!-- 配置http请求处理器适配器 -->  <bean class=*"org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"*></bean> |

#### 如何适配

|  |
| --- |
| **public** **class** HttpRequestHandlerAdapter **implements** HandlerAdapter {  **public** **boolean** supports(Object handler) {  **return** (handler **instanceof** HttpRequestHandler);  }  **public** ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)  **throws** Exception {  ((HttpRequestHandler) handler).handleRequest(request, response);  **return** **null**;  }  **public** **long** getLastModified(HttpServletRequest request, Object handler) {  **if** (handler **instanceof** LastModified) {  **return** ((LastModified) handler).getLastModified(request);  }  **return** -1L;  }  } |

### SimpleControllerHandlerAdapter

SimpleControllerHandlerAdapter简单控制器处理器适配器，所有实现了

org.springframework.web.servlet.mvc.Controller接口的处理器，可以通过此适配器适配、执行。

**参考入门程序**，只是入门程序中，SimpleControllerHandlerAdapter没有在springmvc.xml中进行配置，默认读取的是DispatcherServlet.properties中的SimpleControllerHandlerAdapter配置。

#### Handler实现

开发Handler时需要**实现Controller接口**

|  |
| --- |
| //springmvc的handler开发要实现Controller接口（除了实现Controller接口，还可以实现其他接口）  **public** **class** ItemsController **implements** Controller {  @Override  **public** ModelAndView handleRequest(HttpServletRequest req,  HttpServletResponse resp) **throws** Exception {  // **TODO** Auto-generated method stub  List<Items> itemsList = **new** ArrayList<Items>();  Items items\_1 = **new** Items();  items\_1.setName("联想笔记本");  items\_1.setPrice(6000f);  items\_1.setDetail("ThinkPad T430 联想笔记本电脑！");  Items items\_2 = **new** Items();  items\_2.setName("苹果手机");  items\_2.setPrice(5000f);  items\_2.setDetail("iphone6苹果手机！");  itemsList.add(items\_1);  itemsList.add(items\_2);  // 创建ModelAndView准备填充数据、设置视图  ModelAndView modelAndView = **new** ModelAndView();  // 相当于request.setAttribute(key,value)  modelAndView.addObject("itemsList", itemsList);  // 指定视图的地址  modelAndView.setViewName("WEB-INF/jsp/items/itemsList.jsp");  **return** modelAndView;  }  } |

#### 适配器配置

1、采取默认配置（DispatcherServlet.properties）

2、可以在springmvc.xml中进行配置：

|  |
| --- |
| <!-- 配置处理器适配器 ，所有适配器都实现HandlerAdapter接口 -->  <bean  class=*"org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"* /> |

#### 如何适配

|  |
| --- |
| **public** **class** SimpleControllerHandlerAdapter **implements** HandlerAdapter {  **public** **boolean** supports(Object handler) {  **return** (handler **instanceof** Controller);  }  **public** ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)  **throws** Exception {  **return** ((Controller) handler).handleRequest(request, response);  }  **public** **long** getLastModified(HttpServletRequest request, Object handler) {  **if** (handler **instanceof** LastModified) {  **return** ((LastModified) handler).getLastModified(request);  }  **return** -1L;  }  } |

### BeanNameUrlHandlerMapping

BeanNameUrlHandlerMapping，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的实例。

**参考入门程序。**

**Spring容器中bean的name的值不能重复**

#### 映射器配置

1、采取默认配置（DispatcherServlet.properties）

2、可以在springmvc.xml中进行显式配置：

|  |
| --- |
| <!-- 配置处理器映射器 -->  <!-- 将bean的name与请求url进行映射查找，需要配置handler时指定handler的name -->  <bean class=*"org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"* /> |

#### 配置handler

根据映射器的映射规则，在springmvc.xml中**配置URL和处理器bean的name之间的映射关系**

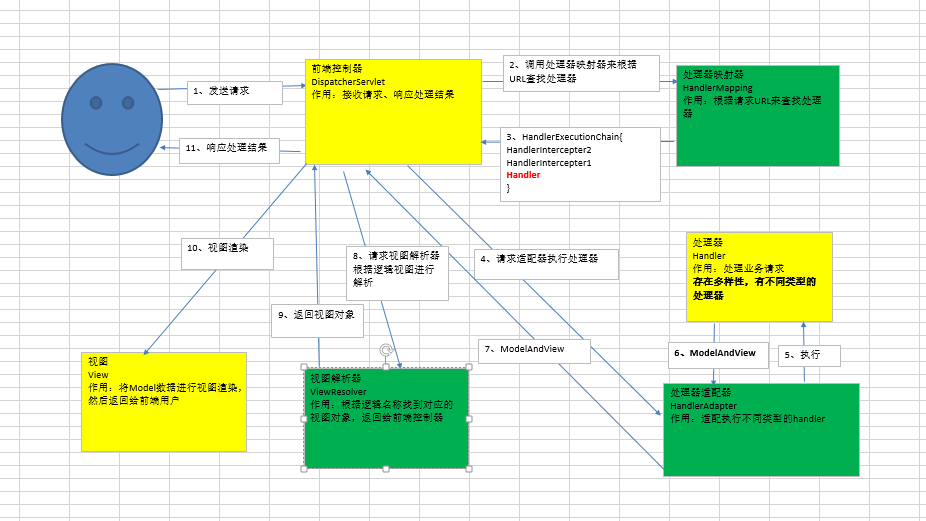
|  |
| --- |
| <!-- 配置处理器 -->  <bean name=*"/queryItems.do"*  class=*"com.kkb.springmvc.controller.ItemsController"*/> |

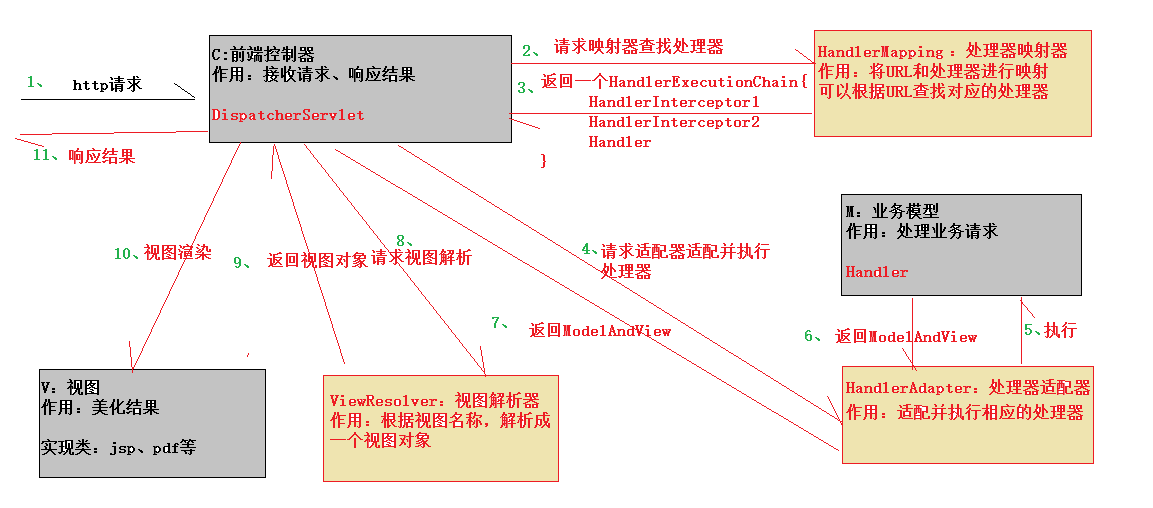
### 总结

* 非注解方式的处理器映射器和处理器适配器可以同时存在多个。
* 非注解方式的处理器映射器和处理器适配器之间可以混用。

# springmvc架构（重点）

## 框架结构





## 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. HandlerAdapter执行处理器(handler，也叫后端控制器)。
6. Controller执行完成返回ModelAndView
7. HandlerAdapter将handler执行结果ModelAndView返回给DispatcherServlet
8. DispatcherServlet将ModelAndView传给ViewReslover视图解析器
9. ViewReslover解析后返回具体View对象
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

## 组件说明

以下组件通常使用框架提供的实现：

* **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。

* **HandlAdapter**：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。

* **View**：视图

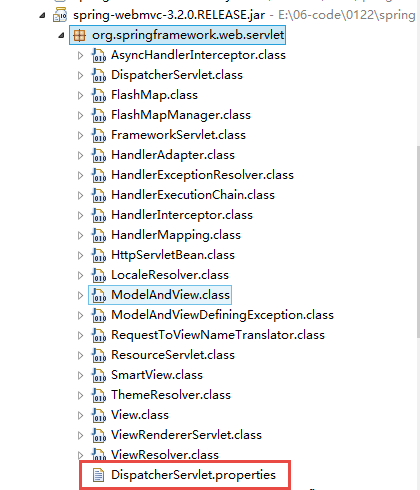
springmvc框架提供了很多的View视图类型的支持，包括：jstlView、freemarkerView、pdfView等。我们最常用的视图就是jsp。

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。

|  |
| --- |
| 说明：在springmvc的各个组件中，**处理器映射器、处理器适配器、视图解析器**称为springmvc的**三大组件**。  需要用户开发的组件有：**处理器、视图** |

## 默认配置文件

spring-webmvc-xxx.jar包中有一个DispatcherServlet.properties文件，该配置中默认加载了一些springmvc默认的其他组件，其中就包括三大组件。



|  |
| --- |
| # Default implementation classes for DispatcherServlet's strategy interfaces.  # Used as fallback when no matching beans are found in the DispatcherServlet context.  # Not meant to be customized by application developers.  org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver  org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeResolver  org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\  org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping  org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\  org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\  org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter  org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\  org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\  org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver  org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator  org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver  org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFlashMapManager |

## 注解映射器和适配器

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

**org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping**是在spring3.1之后使用的注解映射器。

org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter 是在spring3.1之前使用的注解适配器。

**org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter**是在spring3.1之后使用的注解配置器。

**注意：**

**1、注解方式的映射器和适配器在3.1版本前后是不一样的，主讲3.1之后。**

**2、注解方式的处理器映射器和处理器适配器必须配对使用。**

### 通过bean标签配置

**RequestMappingHandlerMapping**：注解式处理器映射器，对类中标记@ResquestMapping的方法进行映射，根据ResquestMapping定义的url匹配ResquestMapping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。

**RequestMappingHandlerAdapter**：注解式处理器适配器，对标记@ResquestMapping的方法进行适配。

配置如下：

<!--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"*/>

### 通过mvc标签配置（推荐）

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

springmvc.xml配置文件中**使用<mvc:annotation-driven/>标签替代bean标签配置的注解处理器和适配器**。

## 视图解析器

在springmvc.xml文件配置如下：

|  |
| --- |
| <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> |

**InternalResourceViewResolver**：默认支持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。*此属性可以不设置，默认为JstlView*。

**prefix** 和**suffix**：查找视图页面的前缀和后缀，最终视图的址为：前缀+**逻辑视图名**+后缀，逻辑视图名需要在controller中返回的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”

# ssm框架整合

## 整合思路

将工程的三层结构中的javabean分别使用spring配置文件进行管理。

1. 整合持久层mapper，包括数据源、会话工程及mapper代理对象的整合；
2. 整合业务层Service，包括事务及service的bean的配置；
3. 整合表现层Controller，直接使用springmvc的配置。
4. Web.xml加载spring容器

**Spring核心配置文件：**

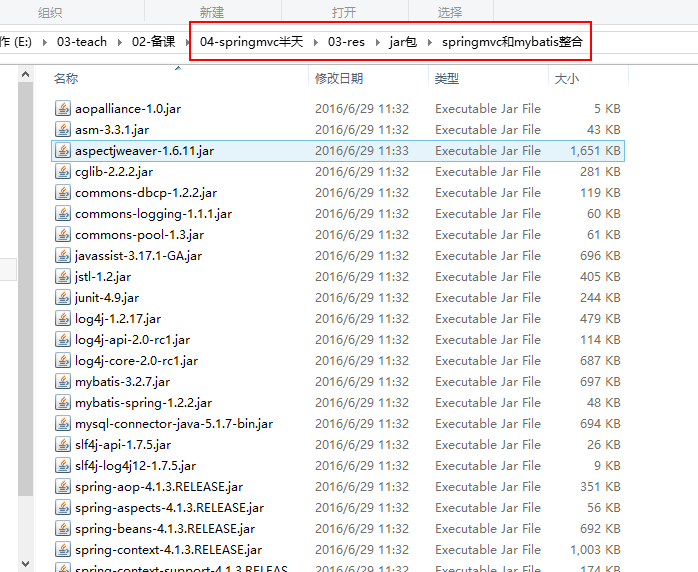
* applicationContext-dao.xml
* applicationContext-service.xml
* springmvc.xml

## 工程搭建

**jar包：**

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

参考：“mybatis与springmvc整合全部jar包”目录



## 工程整合

### 整合Mapper

也可以参考mybatis课件中【mybatis与spring的整合】。

#### applicationContext-dao.xml（核心）

在classpath下，创建spring目录，然后创建SqlMapConfig.xml：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.0.xsd"*>  <!-- 加载db.properties -->  <context:property-placeholder location=*"classpath:****db.properties****"* />  <!-- 配置数据源 -->  <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>  <!-- 配置SqlSessionFacotory -->  <bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>  <!-- 加载mybatis的配置文件（如果配置文件中没有配置项，可以忽略该文件） -->  <property name=*"configLocation"* value=*"classpath:****mybatis/SqlMapConfig.xml****"* />  <!-- 配置数据源 -->  <property name=*"dataSource"* ref=*"dataSource"* />  </bean>  <!-- 配置mapper扫描器，SqlSessionConfig.xml中的mapper配置去掉 -->  <bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>  <!-- 指定扫描的包 -->  <property name=*"basePackage"* value=*"****cn.datang.ssm.mapper****"* />  </bean>  </beans> |

#### SqlMapConfig.xml

在classpath下，创建mybatis目录，然后创建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>  </configuration> |

#### db.properties

在classpath下，创建db.properties：

|  |
| --- |
| jdbc.driver=com.mysql.jdbc.Driver  jdbc.url=jdbc:mysql://localhost:3306/ssm?characterEncoding=utf8  jdbc.username=root  jdbc.password=root |

#### log4j.properties

在classpath下，创建log4j.properties：

|  |
| --- |
| #dev env [debug] product env [info]  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 |

### 整合Service

#### applicationContext-service.xml

在spring文件夹下创建applicationContext-service.xml，文件中配置service。在这个配置文件中，需要配置service的bean和事务管理。

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.0.xsd"*>  <!-- 扫描Service -->  <context:component-scan base-package=*"cn.datang.ssm.service"* />  <!-- 配置事务 -->  <!-- 事务管理器，对mybatis操作数据库进行事务控制，此处使用jdbc的事务控制 -->  <bean id=*"transactionManager"*  class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>  <!-- 指定要进行事务管理的数据源 -->  <property name=*"dataSource"* ref=*"dataSource"*></property>  </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=*"add\*"* propagation=*"REQUIRED"* />  <tx:method name=*"insert\*"* propagation=*"REQUIRED"* />  <tx:method name=*"delete\*"* propagation=*"REQUIRED"* />  <tx:method name=*"del\*"* propagation=*"REQUIRED"* />  <tx:method name=*"remove\*"* propagation=*"REQUIRED"* />  <tx:method name=*"update\*"* propagation=*"REQUIRED"* />  <tx:method name=*"modify\*"* propagation=*"REQUIRED"* />  <tx:method name=*"find\*"* read-only=*"true"* />  <tx:method name=*"query\*"* read-only=*"true"* />  <tx:method name=*"select\*"* read-only=*"true"* />  <tx:method name=*"get\*"* read-only=*"true"* />  </tx:attributes>  </tx:advice>  <!-- aop -->  <aop:config>  <aop:advisor advice-ref=*"txAdvice"*  pointcut=*"execution(\* cn.datang.ssm.service.impl.\*.\*(..))"* />  </aop:config>  </beans> |

### 整合Controller

Spring和springmvc之间无需整合，直接用springmvc的配置

#### web.xml

在web.xml中添加springmvc的配置：

|  |
| --- |
| <?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>ssm</display-name>  <!-- 配置springmvc的前端控制器 -->  <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <!-- contextConfigLocation配置springmvc加载的配置文件（处理器、映射器等等） 如果不配置contextConfigLocation，默认加载的是/WEB-INF/servlet名称-servlet.xml（springmvc-servlet.xml） -->  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:spring/springmvc.xml</param-value>  </init-param>  <!-- 表示servlet随服务启动，里面的值要大于等于0，越小级别越高，一般设置为1、2 -->  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>\*.do</url-pattern>  </servlet-mapping>  </web-app> |

#### springmvc.xml

在spring包下创建springmvc.xml文件，内容如下：

|  |
| --- |
| <?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:p=*"http://www.springframework.org/schema/p"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*  *http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd"*>  <!-- 配置处理器映射器和处理器适配器 -->  <mvc:annotation-driven />  <!-- 配置视图解析器 -->  <bean  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"* value=*"/WEB-INF/jsp/"* />  <property name=*"suffix"* value=*".jsp"* />  </bean>  <!-- 使用注解的handler可以使用组件扫描器，加载handler -->  <context:component-scan base-package=*"cn.datang.ssm.controller"* />  </beans> |

### web.xml加载spring容器

在web.xml中，使用监听器来对spring的配置文件进行加载：

|  |
| --- |
| <!-- 加载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> |

## 整合测试

### 需求

实现商品查询列表，从mysql数据库查询商品信息。

### 持久层代码

根据需求分析，持久层需要查询item表中的记录，使用逆向工程的代码即可。

通过逆向工程，把po类、mapper.xml和mapper.java类生成出来并拷贝到项目中。

### 业务层代码

根据需求开发service的接口以及实现类，**注意：使用注解 @Service开发service**。

|  |
| --- |
| @Service  **public** **class** ItemServiceImpl **implements** ItemService {  @Autowired  **private** ItemMapper mapper;  **public** List<Item> queryItemList() {  ItemExample example = **new** ItemExample();  **return** mapper.selectByExample(example);  }  } |

### 表现层代码

#### Controller类

在Controller类上添加 **@Controller** 注解

在Controller方法上添加 **@RequestMapping**注解进行url请求映射

|  |
| --- |
| @Controller  **public** **class** ItemController {  @Autowired  **private** ItemService service;  @RequestMapping("/queryItem")  **public** ModelAndView queryItem() {  // 根据查询条件去数据库中查询商品列表  List<Item> itemList = service.queryItemList();  // 创建ModelAndView准备填充数据、设置视图  ModelAndView modelAndView = **new** ModelAndView();  // 相当于request.setAttribute(key,value)  modelAndView.addObject("itemList", itemList);  // 设置视图（配置逻辑视图）  modelAndView.setViewName("item/itemList");  **return** modelAndView;  }  } |

#### 开发JSP



把参考资料中的itemList.jsp复制到工程的/WEB-INF/jsp目录下。

### 部署测试

[http://localhost:8080/ssm/queryItem.do](http://localhost:8080/ssm/queryItems.do)

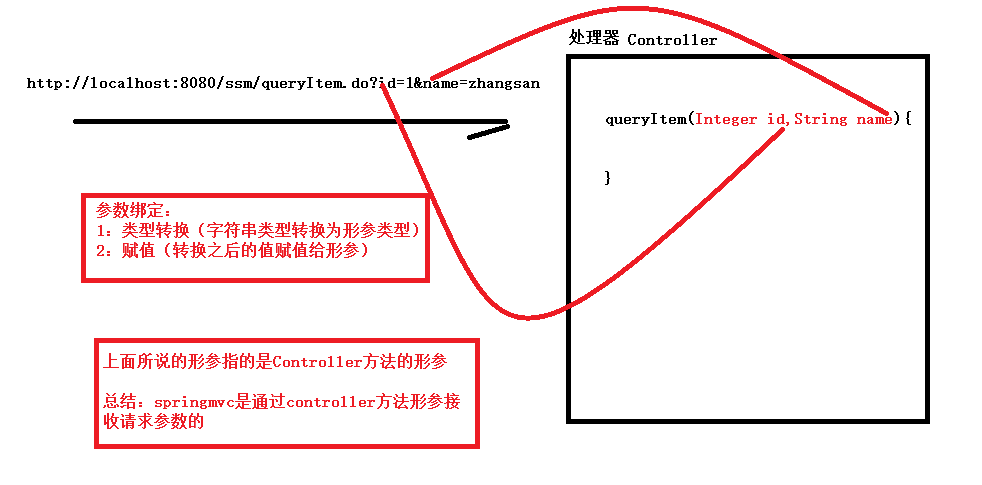
# 参数绑定

## 什么是参数绑定

客户端通过http请求发送的参数，默认是key/value格式（http://XXXXX?id=1&type=301）的***字符串***。

springmvc的**参数绑定组件**，就是将请求参数串中的**value值**进行**类型转换**，然后将转换后的值赋值给controller类中**方法的形参**，这个过程就是***参数绑定***。

总结：**springmvc使用controller方法形参接收请求的参数，这一点是和struts2不一样的**。



## 默认支持的参数类型

***controller方法形参中可以随时添加如下类型的参数，处理适配器会自动识别并进行赋值***。

### HttpServletRequest

通过request对象获取请求信息

### HttpServletResponse

通过response处理响应信息

### HttpSession

通过session对象得到session中存放的对象

### Model/ModelMap

ModelMap是Model接口的实现类，通过Model或ModelMap向页面传递数据，相当于Request的作用，如下：

//调用service查询商品信息

Item item = service.queryItemById(id);

model.addAttribute("item", item);

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

## 绑定简单数据类型

### 需求

打开商品编辑页面，展示商品信息。

### 需求分析

* 表现层分析
* 请求URL：itemEdit.do
* 请求参数：id=1
* 请求返回值：ModelAndView，view名称：item-edit.jsp
* Request域数据：key为item，value为Item对象
* 业务层分析
* 返回值：Item对象
* 方法参数：商品ID
* 业务逻辑：根据商品ID获取商品数据
* 持久层分析

### Dao

持久层使用逆向工程代码即可。

### Service

在ItemServiceImpl实现类中，添加以下方法：

|  |
| --- |
| @Autowired  **private** ItemMapper mapper;  @Override  **public** Item queryItemById(Integer id) {  **if** (id == **null**) {  **return** **null**;  }  **return** mapper.selectByPrimaryKey(id);  } |

### Controller参数绑定

controller中要根据id查询商品数据，需要从请求的参数中把请求的id取出来。最原始的方式就是直接从Request中或者该参数的值。

在springmvc中，对于java简单类型的参数，推荐的参数绑定方式有两种：

1. 直接绑定
2. 注解绑定

#### 直接绑定

|  |
| --- |
| **要点：**  **如果http请求的key的名称和controller方法的形参名称一致，就能直接绑定成功。** |

##### http请求

http://localhost:8080/xxx/showItemEdit.do?id=1，请求参数的key为**id**

##### Controller方法

Controller的形参为Integer id,它和请求的参数的key一致，所以直接绑定成功

|  |
| --- |
| @RequestMapping(value = "/showItemEdit")  **public** ModelAndView showItemEdit (**Integer id**) {  // 查询要显示的商品内容  Item item = service.queryItemById(id);  // 创建ModelAndView准备填充数据、设置视图  ModelAndView modelAndView = **new** ModelAndView();  // 相当于request.setAttribute(key,value)  modelAndView.addObject("item", item);  // 由于配置了ViewResolver，所以此处只写逻辑视图名称即可  modelAndView.setViewName("item/editItem");  **return** modelAndView;  } |

#### 通过注解绑定

|  |
| --- |
| **要点：**  **如果http请求参数的key和controller方法的形参名称不一致，则需要使用@RequestParam注解才能将请求参数绑定成功。** |

##### http请求

[http://localhost:8080/ssm/showItemEdit.do?itemid=1](http://localhost:8080/ssm/showEditItems.do?itemsid=1)，请求参数的key为itemid

##### Controller方法

Controller的形参为Integer id,它和请求的参数不一致，要使用**@RequestParam**注解才能绑定成功

|  |
| --- |
| @RequestMapping(value = "/showItemEdit")  // 通过@RequestParam注解绑定简单类型  **public** ModelAndView showItemEdit(@RequestParam("itemid") Integer id) {  // 查询要显示的商品内容  Item item = service.queryItemById(id);  // 创建ModelAndView准备填充数据、设置视图  ModelAndView modelAndView = **new** ModelAndView();  // 相当于request.setAttribute(key,value)  modelAndView.addObject("item", item);  // 由于配置了ViewResolver，所以此处只写逻辑视图名称即可  modelAndView.setViewName("item/editItem");  **return** modelAndView;  } |

##### RequestParam注解介绍

* **value：**参数名字，即入参的请求参数名字，如value=“itemid”表示请求的参数中的名字为itemid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；

TTP 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

|  |
| --- |
| @RequestMapping(value = "/showItemEdit")  // 通过@RequestParam注解绑定简单类型  // 学习@RequestParam注解中的value、required、defaultValue属性  **public** ModelAndView showItemEdit (  @RequestParam(**value = "itemid", required = true, defaultValue = "2"**) Integer id) {  // 查询要显示的商品内容（此时还没有讲参数绑定，id先写死为1）  Item item = service.queryItemById(id);  // 创建ModelAndView准备填充数据、设置视图  ModelAndView modelAndView = **new** ModelAndView();  // 相当于request.setAttribute(key,value)  modelAndView.addObject("item", item);  // 由于配置了ViewResolver，所以此处只写逻辑视图名称即可  modelAndView.setViewName("item/editItem");  **return** modelAndView;  } |

## 绑定POJO类型

### 需求

将编辑页面修改后的商品信息保存到数据库中。

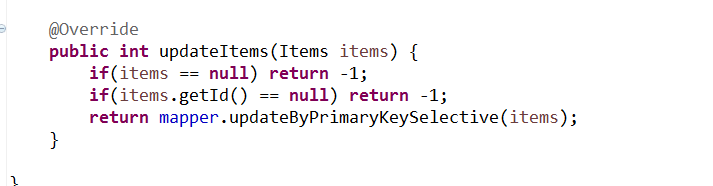
### 需求分析

* 表现层分析
* 请求URL：updateItem.do
* 请求参数：id=1&name=台式机12& price=3000.0&detail=台式机就是好
* 请求返回值：ModelAndView，view名称：success.jsp
* Request域数据：无
* 业务层分析
* 返回值：void
* 方法参数：Item item
* 业务逻辑：根据商品ID修改商品信息
* 持久层分析

### Dao

使用逆向工程代码

### Service



### Controller参数绑定

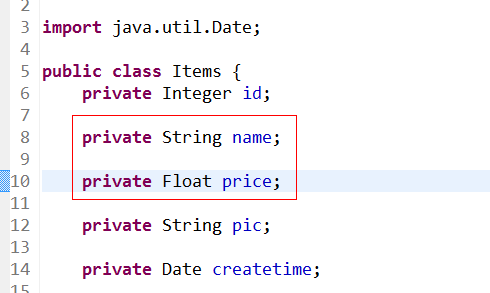
如果提交的参数很多，或者提交的表单中的内容很多的时候可以使用pojo接收数据。***要求pojo对象中的属性名和表单中input的name属性一致***。

Jsp页面定义如下;

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

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

Pojo定义：



***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Controller中pojo类型参数的对应属性上****。*

|  |
| --- |
| @RequestMapping("/updateItem")  **public** ModelAndView updateItem(Integer id,Item item) {  service.updateItem(item);  // 创建ModelAndView准备填充数据、设置视图  ModelAndView modelAndView = **new** ModelAndView();  // 设置视图（配置逻辑视图）  modelAndView.setViewName("success");  **return** modelAndView;  } |

### 解决post乱码问题

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

## 绑定包装POJO

### 需求

查询商品列表时，查询条件比较复杂（既包括商品信息，也包括其他信息），需要定义包装pojo接收参数。

### 参数绑定

包装对象定义如下：

|  |
| --- |
| **public** **class** ItemQueryVO {  //商品信息  **private** Item item;  } |

页面定义（itemList.jsp）：

|  |
| --- |
| 查询条件：  <table width=*"100%"* border=*1*>  <tr>  <td>商品名称：<input type=*"text"* name=*"item.name"* /></td>  <td><input type=*"submit"* value=*"查询"* /></td>  </tr>  </table> |

Controller方法定义如下：



**测试方法：断点跟踪，查看vo中的item对象是否有值。**

## 自定义参数绑定

### 需求

在商品修改页面可以修改商品的生产日期，并且根据业务需求自定义日期格式。

### 需求分析

由于日期数据有很多种格式，所以springmvc没办法把字符串转换成统一的日期类型。所以需要自定义参数绑定。

前端控制器接收到请求后，找到处理器适配器，适配器在执行处理器时，对方法中的形参进行参数绑定。所以如果自定义参数绑定组件的话，需要将该参数组件绑定到处理器适配器上。如果使用<mvc:annotation-driven/>可以在此标签上进行扩展。

### 自定义Converter

|  |
| --- |
| **public** **class** DateConverter **implements** Converter<String, Date> {  @Override  **public** Date convert(String source) {  SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  **try** {  **return** simpleDateFormat.parse(source);  } **catch** (ParseException e) {  e.printStackTrace();  }  **return** **null**;  }  } |

### 配置Converter

在springmvc.xml中，进行以下配置：

|  |
| --- |
| <!-- 加载注解驱动 -->  <mvc:annotation-driven conversion-service=*"conversionService"*/>  <!-- 转换器配置 -->  <bean id=*"conversionService"*  class=*"org.springframework.format.support.FormattingConversionServiceFactoryBean"*>  <property name=*"converters"*>  <set>  <bean class=*"cn.datang.ssm.controller.converter.DateConverter"*/>  </set>  </property>  </bean> |

# springmvc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过虑器。
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。
4. 市场占有率，决定springmvc和struts2的使用频率都很高。