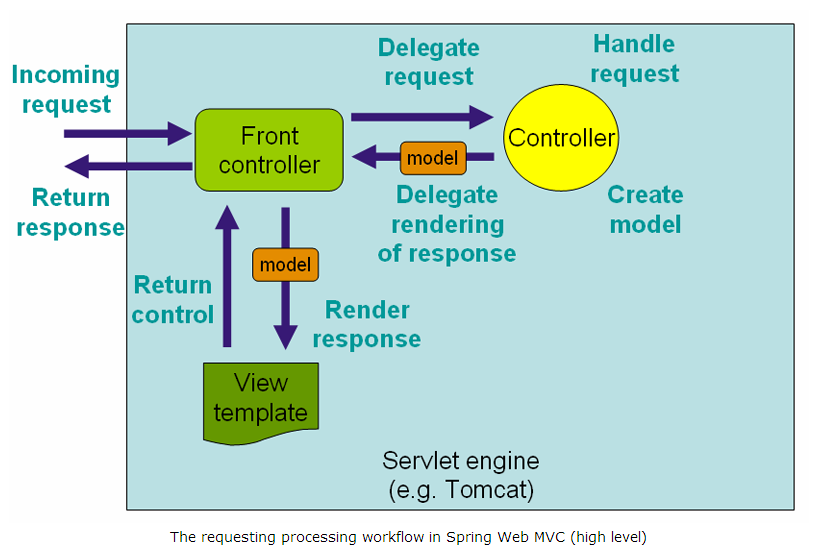
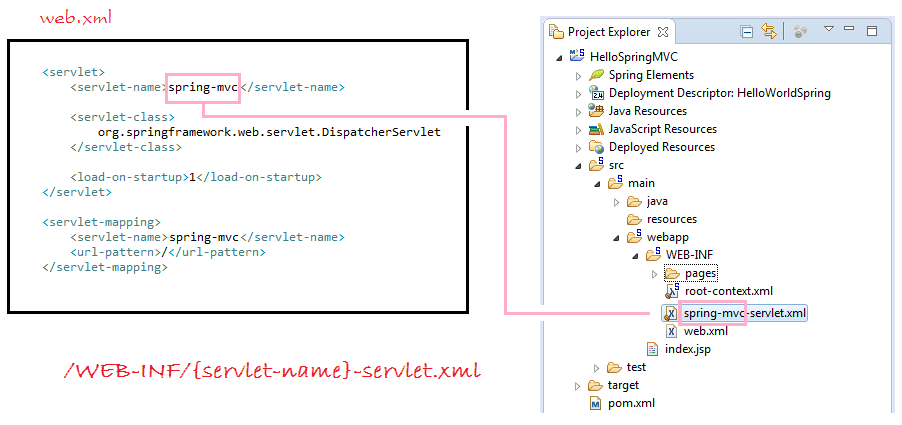
**一、基础入门**

[Spring MVC](http://docs.spring.io/spring/docs/4.0.x/spring-framework-reference/html/mvc.html) 是一个模型 - 视图 - 控制器（MVC）的Web框架建立在中央前端控制器servlet（DispatcherServlet），这是负责发送每个请求到合适的处理器，解决视图并最终返回响应的概念。它是 Spring 产品组合的一部分，它享有Spring IoC容器紧密结合Spring松耦合等特点，因此它有所有优点。



**1、Spring MVC** **DispatcherServlet**读取 xml 配置文件的原则：

{servlet-name} ==> /WEB-INF/{servlet-name}-servlet.xml



如果你不想用 SpringMVC 的使用原则，可以重新配置 SpringMVC  DispatcherServlet 在 web.xml 文件中：

<servlet>

<servlet-name>my-dispatcher-name</servlet-name>

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

<init-param>

<!-- override default name {servlet-name}-servlet.xml -->

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

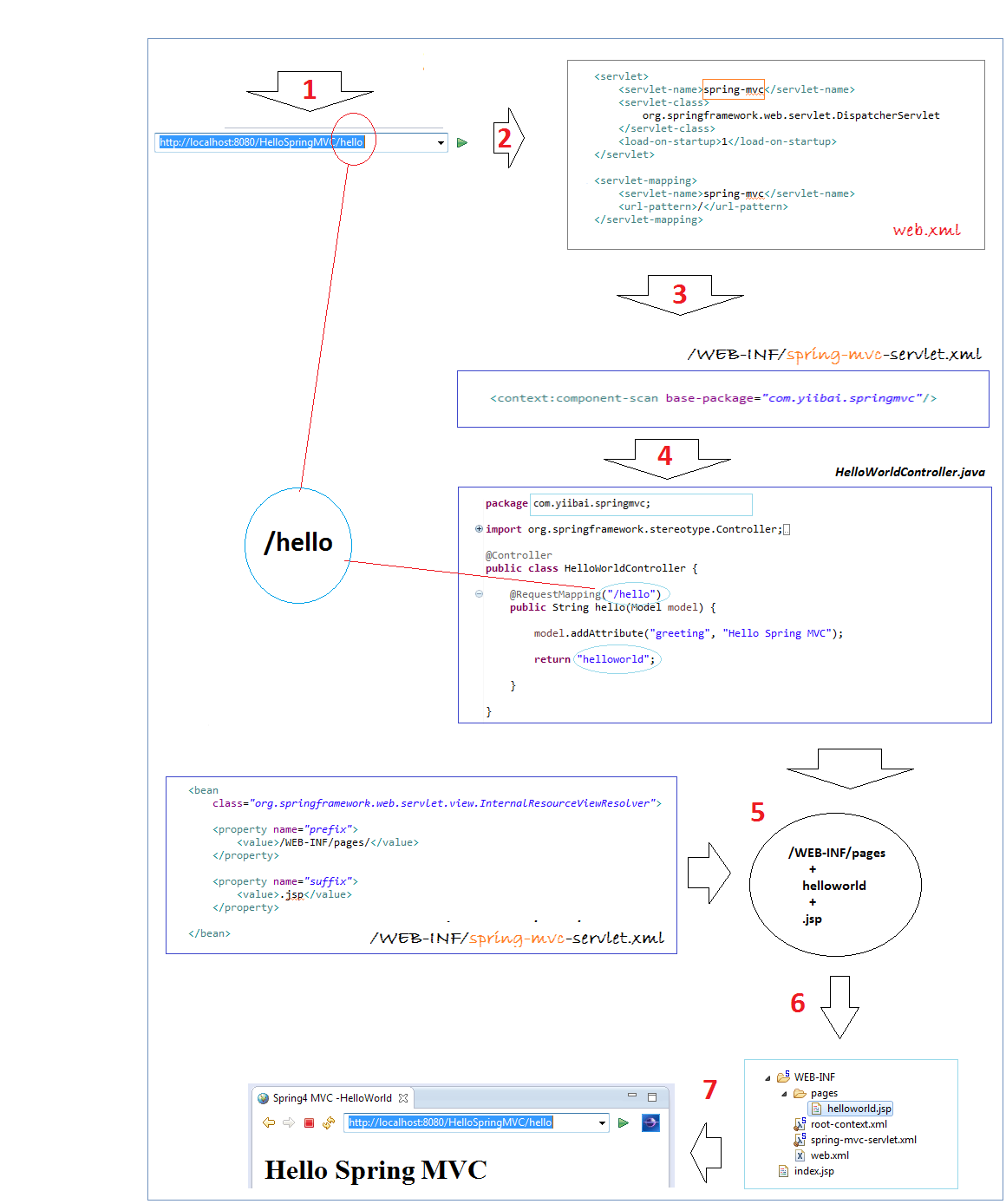
<param-value>/WEB-INF/springmvc-myconfig.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

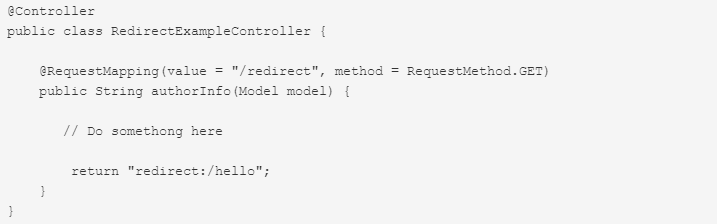
</servlet>

**2、应用程序的流程：**



3、使用@RequestParam 注解将请求参数绑定到你的控制器方法参数。

4、使用前缀 "redirect:" ，该方法返回字符串，可以重定向到另一页面。



## 5、@PathVariable示例

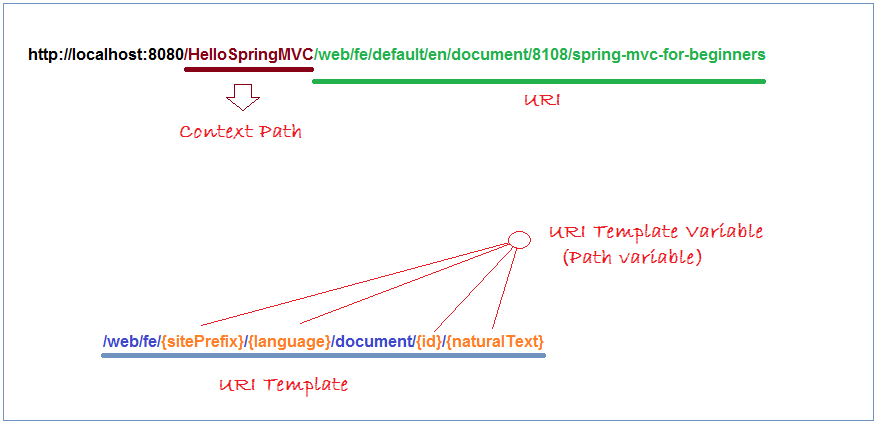
在Spring MVC中，可以使用@PathVariable注释将一个方法参数绑定到一个URI模板变量的值：

例如，这是一个模板的URI：

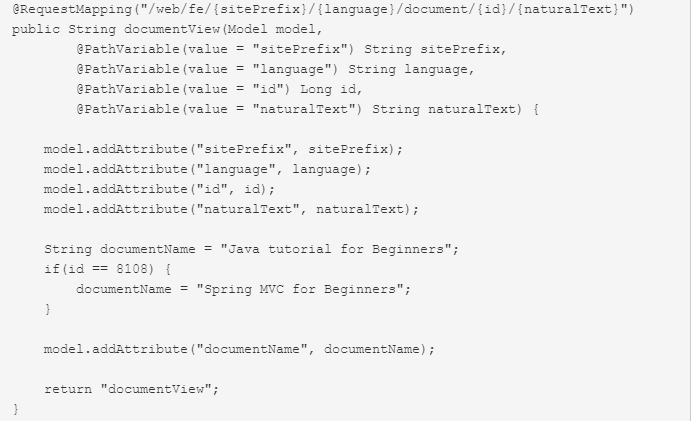
* /web/fe/{sitePrefix}/{language}/document/{id}/{naturalText}

而下面的 URI 模板匹配上面：

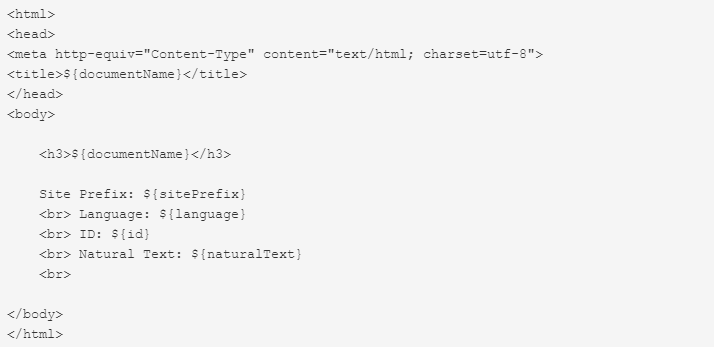
* /web/fe/default/en/document/8108/spring-mvc-for-beginners
* /web/fe/default/vi/document/8108/spring-mvc-cho-nguoi-moi-bat-dau
* .....



下面的代码片段显示了用法：



* /WEB-INF/pages/documentView.jsp



运行URL：

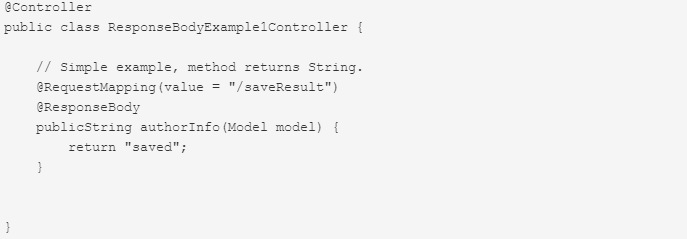
<http://localhost:8080/HelloSpringMVC/web/fe/default/en/document/8108/spring-mvc-for-beginners>

## 6、@ResponseBody示例

如果您使用 @ResponseBody 注释到方法， spring 将尝试转换它的返回值，并自动写入到HTTP响应。在这种情况下，并不需要一个特定的视图。

*注：方法不一定需要返回字符串类型。*

**使用@ResponseBody和方法返回字符串的简单例子。**



**二、Spring4 MVC HelloWorld 注解和JavaConfig实例**

使XML配置在两个地方：第一个是 spring-mvc-servlet.xml ，定义了视图解析程序识别真正的视图、位置搜索、通过组件扫描Bean。第二个是 web.xml, 我们定义前端控制器配置和URL模式将被寻找匹配。

再次创建一个Hello world的例子，但这个时候我们使用Java配置。 我们将删除上面提到的XML文件，并通过它们对应的Java替换这些XML配置。

**1、创建所需的目录结构项目**

使用Eclipse创建一个Maven web工程 包含使用Eclipse一步一步的向导来创建一个Maven项目。

以下将是最后的项目结构

**2、使用Spring和Servlet依赖更新pom.xml**

我们要讨论以 Spring Java为基础的配置取决于Servlet 3.0 的API， 因此，我们需要包含的依赖在 pom.xml 中。

<properties>

<springframework.version>4.0.6.RELEASE</springframework.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${springframework.version}</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>javax.servlet.jsp-api</artifactId>

<version>2.3.1</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

</dependencies>

<build>

<finalName>SpringMVCNoXML</finalName>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>2.4</version>

<configuration>

<warSourceDirectory>src/main/webapp</warSourceDirectory>

<warName>SpringMVCNoXML</warName>

<failOnMissingWebXml>false</failOnMissingWebXml>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

首先要注意这里maven-war-plugin 插件的声明。正如我们将完全删除web.xml ，我们需要配置这个插件，以避免Maven构建war包失败。第二个变化是加入了JSP/Servlet/Jstl 的依赖关系，这些我们可能需要，因为我们将要使用 servlet API和JSTL视图在我们的代码中。在一般情况下，容器已经包含这些库，从而在pom.xml中为他们提供了，我们可以设置作用范围。

**3、添加控制器类**

在src/main/java下添加一个控制器类，如下所示：com.why.springmvc.controller.HelloworldController

**package** com.why.springmvc.controller;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.ModelMap;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestMethod;

@Controller

@RequestMapping("/")

**public** **class** HelloWorldController {

@RequestMapping(method = RequestMethod.*GET*)

**public** String sayHello(ModelMap model){

model.addAttribute("greeting", "Hello World from spring 4 MVC");

**return** "welcome";

}

@RequestMapping(value = "/helloagain",method = RequestMethod.*GET*)

**public** String sayHelloAgain(ModelMap model){

model.addAttribute("greeting", "Hello World Again, from spring 4 MVC");

**return** "welcome";

}

}

在类名@Controller注解声明这个类的Spring bean 以及 @RequestMapping注解声明了这个类是默认处理程序键入“/”的所有请求。第一种方法没有声明因此任何映射，它将继承映射的映射声明是在类级别上，默认处理GET请求。方法二(由于额外的映射声明使用value属性)形式 /hello 将再次请求。属性方法说哪种类型的HTTP请求这种方法可以服务。

方法说哪种类型的HTTP请求这种方法可以服务。 ModelMap是一个Map实现，在这里作为替代[request.getAttribute()/request.setAttribute()] 设定值作为请求属性。请注意，我们从这个方法返回“welcome”字符串。此字符串将后缀和前缀后缀，在视图解析器定义的前缀(见上面的 spring-servlet.xml)，形成真正的视图文件名。

**4、添加视图**

创建一个新的文件夹命名为views在WEB-INF目录下，并添加一个简单的JSP页面welcome.jsp (WEB-INF/views/welcome.jsp)从控制器到简单的访问模式值。

<%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*

pageEncoding=*"utf-8"*%>

<!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>HelloWorld page</title>

</head>

<body>

Greeting : ${greeting}</body></html>

**5、添加配置类**

在src/main/java下添加下面提到的类指定的包，如下图所示。这种构造类可以被看作是一个替代 spring-servlet.xml，因为它包含了所有必需的组件的扫描和视图解析器的信息。

com.why.springmvc.configuration.HelloWorldConfiguration

**package** com.why.springmvc.configuration;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.servlet.ViewResolver;

**import** org.springframework.web.servlet.config.annotation.EnableWebMvc;

**import** org.springframework.web.servlet.view.InternalResourceViewResolver;

**import** org.springframework.web.servlet.view.JstlView;

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = "com.yiibai.springmvc")

**public** **class** HelloWorldConfiguration {

@Bean

**public** ViewResolver viewResolver() {

InternalResourceViewResolver viewResolver = **new** InternalResourceViewResolver();

viewResolver.setViewClass(JstlView.**class**);

viewResolver.setPrefix("/WEB-INF/views/");

viewResolver.setSuffix(".jsp");

**return** viewResolver;

}

}

@Configuration指明该类包含注解为@Bean 生产 bean管理是由Spring容器的一个或多个bean方法。 以上配置类对应等同于以下XML：

<!-- 自动扫描配置包 -->

<context:component-scan base-package=*"com.why.springmvc"*/>

<!-- 注解驱动配置 -->

<mvc:annotation-driven />

<context:annotation-config/>

<!-- 视图名称解析器：InternalResourceViewResolver -->

<bean

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

<!-- 前缀 -->

<property name=*"prefix"*>

<value>/WEB-INF/views/</value>

</property>

<!-- 后缀 -->

<property name=*"suffix"*>

<value>.jsp</value>

</property>

</bean>

@EnableWebMvc 等同于 mvc:annotation-driven 在XML中. 它能够为使用@RequestMapping向特定的方法传入的请求映射@Controller-annotated 类。

@ComponentScan 等同于 context:component-scan base-package="..." 提供 spring 在哪里寻找管理 beans/classes.

**6、添加初始化类**

添加一个初始化类实现 WebApplicationInitializer 在src/main/java 中使用如下图所示指定包(在这种情况下，作为替代在 web.xml 中定义的任何 Spring 配置)。在Servlet 3.0的容器启动时，这个类将被加载并初始化，并在启动由servlet容器调用方法：com.why.springmvc.configuration.HelloWorldInitializer

**package** com.why.springmvc.configuration;

**import** javax.servlet.ServletContext;

**import** javax.servlet.ServletException;

**import** javax.servlet.ServletRegistration;

**import** org.springframework.web.WebApplicationInitializer;

**import** org.springframework.web.context.support.AnnotationConfigWebApplicationContext;

**import** org.springframework.web.servlet.DispatcherServlet;

**public** **class** HelloWorldInitializer **implements** WebApplicationInitializer {

**public** **void** onStartup(ServletContext container) **throws** ServletException {

AnnotationConfigWebApplicationContext ctx = **new** AnnotationConfigWebApplicationContext();

ctx.register(HelloWorldConfiguration.**class**);

ctx.setServletContext(container);

ServletRegistration.Dynamic servlet = container.addServlet("dispatcher", **new** DispatcherServlet(ctx));

servlet.setLoadOnStartup(1);

servlet.addMapping("/");

}

}

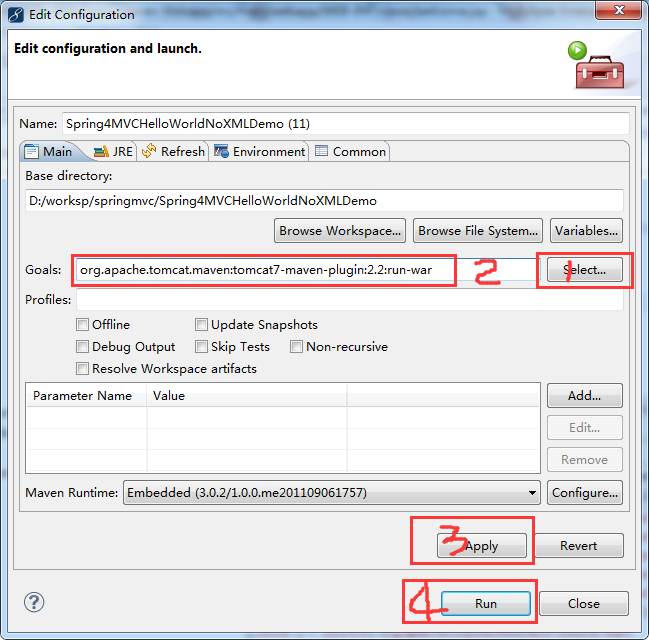
内容上面类似 web.xml 在之前的教程中的内容，因为我们使用的是前端控制器 DispatcherServlet，分配映射(URL模式的XML)和而不是提供给 Spring 配置文件(spring-servlet.xml)的路径，在这里，我们正在注册的配置类。总体而言，我们都在做同样的事情，只是方式有所不同。

**7、构建和部署应用程序**

有一点要记住，像WebApplicationInitializer，Spring 是基于Java 的配置API依赖于 Servlet3.0容器。确保你没有使用Servlet声明任何小于3.0。对于我们的情况，我们将从应用程序中删除 web.xml 文件。

现在构建war (无论是作为Eclipse中提到的最后一个教程)或通过Maven的命令行(mvn clean install)。部署war 到Servlet3.0容器。由于我在这里使用Tomcat，我就干脆把这个 war 文件放到 Tomcat 的 webapps 文件夹，然后在 tomcat 的bin 目录里面点击 start.bat 运行。

或者 右键工程 =>Run As => Maven install 完成后，再次 右键工程 =>Run As => Maven build，弹出选择：



**三、Spring4 MVC 文件下载实例**

下载文件是相当简单的，涉及以下步骤。

* 创建一个InputStream到文件用于下载。
* 查找MIME类型下载文件的内容：

–可以是application/pdf, text/html,application/xml,image/png等等。

* 将内容类型与上述发现的MIME类型响应（HttpServletResponse）：

response.setContentType(mimeType);

* 针对以上找到MIME类型设置内容长度：

response.setContentLength(file.getLength());//length in bytes

* 为响应设置内容处理标头：

response.setHeader(“Content-Disposition”, “attachment; filename=” + fileName); //随着“附件”文件将下载。可能会显示一个“另存为”基于浏览器的设置对话框。

response.setHeader(“Content-Disposition”, “inline; filename=” + fileName);//通过“内联”浏览器将尝试显示内容到浏览器中（图片，PDF，文本，...）。对于其他内容类型，文件将直接下载。

* 从InputStream中复制字节响应到 OutputStream。
* 一旦复制完成后，关闭输入输出流。

…………………….

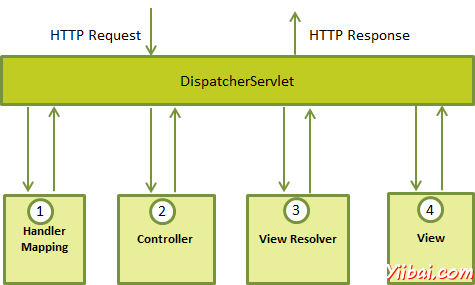
**四、Spring MVC总结**

MVC模式会导致分离的应用程序（输入逻辑，业务逻辑和UI逻辑）的不同方面，同时提供这些元素之间的松耦合。

* 模型（Model ）封装了应用程序的数据和一般他们会组成的POJO。
* 视图（View）是负责呈现模型数据和一般它生成的HTML输出，客户端的浏览器能够解释。
* 控制器（Controller ）负责处理用户的请求，并建立适当的模型，并把它传递给视图渲染。

**DispatcherServlet**

Spring的web模型 - 视图 - 控制器（MVC）框架是围绕着处理所有的HTTP请求和响应的DispatcherServlet的设计。Spring的Web MVC框架的DispatcherServlet的请求处理流程说明如下图：



下面是对应于传入的HTTP请求到DispatcherServlet的事件序列：

* 接收HTTP请求后，DispatcherServlet 咨询 HandlerMapping 来调用相应的控制器。
* 该控制器接受请求并调用基于使用GET或POST方法相应的服务方法。服务方法将基于定义的业务逻辑设置模型数据，并返回视图名到DispatcherServlet。
* DispatcherServlet将需要帮助的ViewResolver从拾取到该请求所定义的视图。
* 一旦视图被敲定，DispatcherServlet会传递模型数据是在浏览器上最终呈现的视图。

所有上述部件，即HandlerMapping，控制器和视图解析器WebApplicationContext 部分是纯的 ApplicationContext 必要的 Web应用程序的一些额外的功能扩展。

**所需配置：**

需要映射所需的DispatcherServlet处理，通过在web.xml文件中使用URL映射请求。下面是一个例子，说明声明和映射 HelloWeb DispatcherServlet 的例子：



web.xml 文件将被保存在您的Web应用程序的 WebContent/ WEB-INF 目录。 在 DispatcherServlet 的 HelloWeb 初始化，该框架将尝试从一个名为 [servlet-name]-servlet.xml位于应用程序 WebContent/WEB-INF 目录文件加载应用程序上下文。在这种情况下我们的文件将是HelloWeb-servlet.xml。

接下来，<servlet-mapping>标记指示URL会被DispatcherServlet处理。这里全部用。jsp结束HTTP请求将由DispatcherServlet的HelloWeb处理。

如果不想用默认文件名为[servlet-name]-servlet.xml和默认位置的WebContent/WEB-INF，可以通过添加servlet 的 listenerContextLoaderListener 在 web.xml 文件中定义该文件的名称和位置如下：



现在，让我们检查所需配置的HelloWeb-servlet.xml文件，放置在 Web应用程序的WebContent/WEB-INF目录：



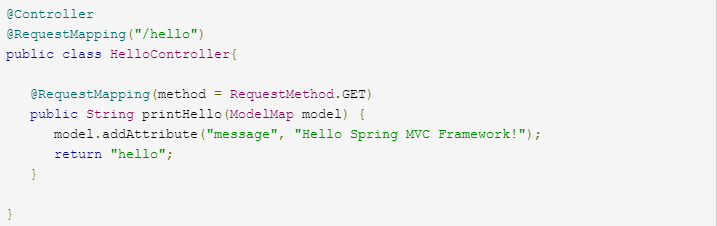
下面是有关 HelloWeb-servlet.xml 文件的要点：

* [servlet-name]-servlet.xml 文件将被用于创建定义的Bean，会覆盖在全局范围里名字相同的Bean的定义。
* <context:component-scan...>标签将使用**启动Spring MVC的注解扫描**功能，允许做出像 @Controller和使用@RequestMapping注解等使用。
* 使用InternalResourceViewResolver将有定义来**解析视图名**的规则。按照上述定义的规则，命名为你好的逻辑视图被委托给一个视图实现位于/WEB-INF/jsp/hello.jsp。

下一节将学习如何创建实际组件IE浏览器。控制器，模型和视图。

**定义控制器：**

DispatcherServlet 委托请求发送到控制器，以执行特定于它的功能。注解@Controller表示一个特定的类提供一个控制器的角色。注解@RequestMapping 用于将URL映射到任何一个类或者一个特定的处理方法。



注解@Controller类定义为一个Spring MVC控制器。在这里，使用@RequestMapping第一次使用表明，该控制器上的所有处理方法是相对于/hello 路径。下一步标注使用@RequestMapping（方法= RequestMethod.GET）用于声明printHello() 方法作为**控制器的默认服务方法**来**处理HTTP GET请求**。可以定义另一种方法来处理同一URL的POST请求。

可以写在上面的另一种形式，可以使用@RequestMapping添加额外的属性如下控制：



value属性指示该处理方法所映射到的URL和method属性定义了服务的方法来处理HTTP GET请求。有以下重要点要注意关于上述定义的控制器：

* 将定义所需的业务逻辑内部的服务方法。可以调用此方法在另一个方法按要求。
* 基于定义的业务逻辑，将在此方法中创建一个模型。您可以设定不同的模型的属性和这些属性将被视图访问提出的最终结果。此示例创建一个具有自己的属性“message”的模型。
* 定义的服务方法可以返回一个字符串，其中包含要用于渲染模型视图的名称。这个例子返回“hello”作为逻辑视图名。

**创建JSP视图：**

Spring MVC支持多种类型的视图不同的演示技术。这些措施包括：JSP，HTML，PDF，Excel，XML，Velocity，XSLT，JSON，Atom和RSS订阅，JasperReports等，但最常用JSTL JSP模板。因此，我们编写一个/WEB-INF/hello/hello.jsp简单的hello 视图：



在这里， ${message}是我们已经建立了控制器内部的属性。你可以在视图中显示多个属性。

<mvc:resources..../>标签被用来映射静态页面。映射属性必须是一个Ant模式，指定URL的HTTP请求的模式。位置属性必须指定具有静态页面，包括图片，样式表，JavaScript和其他静态内容的一个或多个有效的资源目录位置。多个资源位置可以使用逗号分隔的值的列表来指定。

<mvc:resources mapping="/pages/\*\*" location="/WEB-INF/pages/" />

<mvc:annotation-driven/>

**五、Spring MVC补充（重点）**

Spring Web MVC也是服务到工作者模式的实现，但进行可优化。**前端控制器**是**DispatcherServlet**；应用控制器其实拆为**处理器映射器**(**Handler Mapping**)进行处理器管理和**视图解析器**(**View Resolver**)进行视图管理；页面控制器/动作/处理器为**Controller接口**（仅包含ModelAndView handleRequest(request, response) 方法）的实现（也可以是任何的POJO类）；支持本地化（Locale）解析、主题（Theme）解析及文件上传等；提供了非常灵活的**数据验证、格式化和数据绑定机制**；提供了强大的约定大于配置（惯例优先原则）的契约式编程支持。

**1、Spring Web MVC架构**

Spring Web MVC框架也是一个基于请求驱动的Web框架，并且也使用了**前端控制器模式**来进行设计，再根据请求映射规则分发给相应的页面控制器（动作/处理器）进行处理。首先让我们整体看一下Spring Web MVC处理请求的流程：

**1）Spring Web MVC处理请求的流程**



具体执行步骤如下：

* 首先用户发送请求—>前端控制器，前端控制器根据请求信息（如URL）来决定选择哪一个页面控制器进行处理并把请求委托给它，即以前的控制器的控制逻辑部分；图中的1、2步骤；
* 页面控制器接收到请求后，进行功能处理，首先需要**收集和绑定请求参数到一个对象**，这个对象在Spring Web MVC中叫**命令对象**，并进行验证，然后将命令对象委托给业务对象进行处理；处理完毕后返回一个ModelAndView（模型数据和逻辑视图名）；图2-1中的3、4、5步骤；
* 前端控制器收回控制权，然后根据返回的逻辑视图名，选择相应的视图进行渲染，并把模型数据传入以便视图渲染；图2-1中的步骤6、7；
* 前端控制器再次收回控制权，将响应返回给用户，图2-1中的步骤8；至此整个结束。

**2）Spring Web MVC核心架构**

核心架构的具体流程步骤如下：

* 首先用户发送请求——>DispatcherServlet：前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为**统一访问点**，进行全局的流程控制；
* DispatcherServlet—>HandlerMapping：HandlerMapping将会把**请求映射**为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象，通过这种策略模式，很容易添加新的映射策略；
* DispatcherServlet——>HandlerAdapter：HandlerAdapter将会把**处理器**包装为**适配器**，从而支持多种类型的处理器，即**适配器设计模式**的应用，从而很容易**支持很多类型的处理器**；
* HandlerAdapter——>处理器功能处理方法的调用，HandlerAdapter将会根据适配的结果**调用**真正的**处理器的功能处理方法**，完成功能处理；并返回一个**ModelAndView**对象（包含模型数据、逻辑视图名）；
* ModelAndView的逻辑视图名——> ViewResolver： ViewResolver将把**逻辑视图名**解析为具体的View，通过这种策略模式，很容易更换其他视图技术；
* View——>渲染，View会根据传进来的Model模型数据进行渲染，此处的**Model**实际是一个**Map数据**结构，因此很容易支持其他视图技术；
* 返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

此处我们只是讲了核心流程，没有考虑拦截器、本地解析、文件上传解析等，后边再细述。



架构图对应的DispatcherServlet**核心代码**如下：

//前端控制器分派方法

**protected** **void** doDispatch(HttpServletRequest request, HttpServletResponse response) **throws** Exception {

HttpServletRequest processedRequest = request;

HandlerExecutionChain mappedHandler = **null**;

**int** interceptorIndex = -1;

**try** {

ModelAndView mv;

**boolean** errorView = **false**;

**try** {

//检查是否是请求是否是multipart（如文件上传），如果是将通过MultipartResolver解析

processedRequest = checkMultipart(request);

//步骤2、请求到处理器（页面控制器）的映射，通过HandlerMapping进行映射

mappedHandler = getHandler(processedRequest, **false**);

**if** (mappedHandler == **null** || mappedHandler.getHandler() == **null**) {

noHandlerFound(processedRequest, response);

**return**;

}

//步骤3、处理器适配，即将我们的处理器包装成相应的适配器（从而支持多种类型的处理器）

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

// 304 Not Modified缓存支持

//此处省略具体代码

// 执行处理器相关的拦截器的**预处理**（HandlerInterceptor.preHandle）

//此处省略具体代码

// 步骤4、由适配器执行处理器（调用处理器相应功能处理方法）

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

// Do we need view name translation?

**if** (mv != **null** && !mv.hasView()) {

mv.setViewName(getDefaultViewName(request));

}

// 执行处理器相关的拦截器的**后处理**（HandlerInterceptor.postHandle）

//此处省略具体代码

}**catch** (ModelAndViewDefiningException ex) {

logger.debug("ModelAndViewDefiningException encountered", ex);

mv = ex.getModelAndView();

}**catch** (Exception ex) {

Object handler = (mappedHandler != **null** ? mappedHandler.getHandler() : **null**);

mv = processHandlerException(processedRequest, response, handler, ex);

errorView = (mv != **null**);

}

//步骤5 步骤6、解析视图并进行视图的渲染

//步骤5 由ViewResolver解析View（viewResolver.resolveViewName(viewName, locale)）

//步骤6 视图在渲染时会把Model传入（view.render(mv.getModelInternal(), request, response);）

**if** (mv != **null** && !mv.wasCleared()) {

render(mv, processedRequest, response);

**if** (errorView) {

WebUtils.clearErrorRequestAttributes(request);

}

}**else** {

**if** (logger.isDebugEnabled()) {

logger.debug("Null ModelAndView returned to DispatcherServlet with name '" + getServletName() +

"': assuming HandlerAdapter completed request handling");

}

}

// 执行处理器相关的拦截器的完成后处理（HandlerInterceptor.afterCompletion）

//此处省略具体代码

**catch** (Exception ex) {

// Trigger after-completion for thrown exception.

triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, ex);

**throw** ex;

}**catch** (Error err) {

ServletException ex = **new** NestedServletException("Handler processing failed", err);

// Trigger after-completion for thrown exception.

triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, ex);

**throw** ex;

}**finally** {

// Clean up any resources used by a multipart request.

**if** (processedRequest != request) {

cleanupMultipart(processedRequest);

}

}

}

**3）问题及解答**

* 请求如何给前端控制器？这个应该在web.xml中进行部署描述，在HelloWorld中详细讲解。
* 前端控制器如何**根据请求**信息**选择页面**控制器进行功能处理？ 我们需要配置**HandlerMapping**进行映射
* 如何支持多种页面控制器呢？配置HandlerAdapter从而支持多种类型的页面控制器
* 如何页面控制器**如何使用业务对象**？可以预料到，肯定利用**Spring IoC**容器的**依赖注入**功能
* 页面控制器如何**返回模型数据**？使用ModelAndView返回
* 前端控制器如何根据页面控制器返回的逻辑视图名选择具体的**视图**进行**渲染**？ 使用ViewResolver进行解析
* 不同的视图技术如何使用相应的模型数据？ 因为Model是一个Map数据结构，很容易支持其他视图技术

**4）具体的核心开发步骤**

* DispatcherServlet在web.xml中的部署描述，从而拦截请求到Spring Web MVC
* HandlerMapping的配置，从而将请求映射到处理器
* HandlerAdapter的配置，从而支持多种类型的处理器
* ViewResolver的配置，从而将逻辑视图名解析为具体视图技术
* 处理器（页面控制器）的配置，从而进行功能处理

**2、Spring Web MVC优势**

* 清晰的**角色划分**：前端控制器（DispatcherServlet）、请求到处理器映射（HandlerMapping）、处理器适配器（HandlerAdapter）、视图解析器（ViewResolver）、处理器或页面控制器（Controller）、验证器（Validator）、命令对象（Command **请求参数绑定到的对象**就叫**命令对象**）、表单对象（Form Object **提供给表单展示和提交到的对象**就叫**表单对象**）。
* 分工明确，而且扩展点相当灵活，可以很容易扩展，虽然几乎不需要；
* 由于命令对象就是一个POJO，无需继承框架特定API，可以使用命令对象直接作为业务对象；
* 和Spring 其他框架无缝集成，是其它Web框架所不具备的；
* 可适配，通过HandlerAdapter可以支持任意的类作为处理器；
* 可定制性，HandlerMapping、ViewResolver等能够非常简单的定制；
* 功能强大的数据验证、格式化、绑定机制；
* 利用Spring提供的**Mock对象**能够非常简单的进行**Web层单元测试**；
* 本地化、主题的解析的支持，使我们更容易进行国际化和主题的切换。
* 强大的JSP标签库，使JSP编写更容易。

………………还有比如RESTful风格的支持、简单的文件上传、约定大于配置的契约式编程支持、基于注解的零配置支持等等。

**3、实例配置**

**1）前端控制器DispatcherServlet配置**

<!-- 入口配置(前端控制器配置)：DispatcherServlet -->

<servlet>

<servlet-name>spring-mvc</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>spring-mvc</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

**load-on-startup**：表示启动容器时初始化该Servlet；

**url-pattern**：表示哪些请求交给Spring Web MVC处理，“/”是用来定义默认servlet映射的。也可以如“\*.html”表示拦截所有以html为扩展名的请求。

**2）配置HandlerMapping、HandlerAdapter**

<!-- HandlerMapping -->

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

<!-- HandlerAdapter -->

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

**BeanNameUrlHandlerMapping**：表示**将请求的URL和Bean名字映射**，如URL为 “上下文/hello”，则Spring配置文件必须有一个名字为“/hello”的Bean，上下文默认忽略。

**SimpleControllerHandlerAdapter**：表示所有实现了org.springframework.web.servlet.mvc.**Controller**接口的Bean可以**作为Spring Web MVC中的处理器**。 如果需要其他类型的处理器可以通过实现HadlerAdapter来解决。

**3）配置ViewResolver**

<!-- ViewResolver -->

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

<property name=*"viewClass"* value=*"org.springframework.web.servlet.view.JstlView"*/>

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

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

</bean>

**InternalResourceViewResolver**：用于支持Servlet、JSP视图解析；

**viewClass**：JstlView表示JSP模板页面需要使用JSTL标签库，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（前缀[逻辑视图名]后缀），比如传进来的逻辑视图名为hello，则该该jsp视图页面应该存放在“WEB-INF/jsp/hello.jsp”；

**4）开发处理器/页面控制器**

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

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

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

**public** **class** HelloWorldController **implements** Controller {

@Override

**public** ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse resp) **throws** Exception {

//1、收集参数、验证参数

//2、绑定参数到命令对象

//3、将命令对象传入业务对象进行业务处理

//4、选择下一个页面

ModelAndView mv = **new** ModelAndView();

//添加**模型数据** 可以是任意的POJO对象

mv.addObject("message", "Hello World!");

//设置**逻辑视图名**，视图解析器会根据该名字解析到具体的视图页面

mv.setViewName("hello");

**return** mv;

}

}

**org.springframework.web.servlet.mvc.Controller**：页面控制器/处理器必须实现Controller接口，注意别选错了；后边我们会学习其他的处理器实现方式；

**public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse resp)** ：功能处理方法，实现相应的功能处理，比如收集参数、验证参数、绑定参数到命令对象、将命令对象传入业务对象进行业务处理、最后返回ModelAndView对象；

**ModelAndView**：包含了视图要实现的**模型数据和逻辑视图名**；“mv.addObject("message", "Hello World!");

”表示添加模型数据，此处可以是任意POJO对象；“mv.setViewName("hello");”表示设置逻辑视图名为“hello”，视图解析器会将其解析为具体的视图，如前边的视图解析器InternalResourceViewResolver会将其解析为“WEB-INF/jsp/hello.jsp”。

将其添加到Spring配置文件(WEB-INF/chapter2-servlet.xml)，让其接受Spring IoC容器管理:

<!-- 处理器 -->

<bean name=*"/hello"* class=*"cn.javass.chapter2.web.controller.HelloWorldController"*/>

name="/hello"：前边配置的BeanNameUrlHandlerMapping，表示如过请求的URL为 “上下文/hello”，则将会交给该Bean进行处理。

**5）开发视图界面**

此处省略……

**6）运行流程分析**



运行步骤：

* 首先用户发送请求http://localhost:9080/springmvc-chapter2/hello——>web容器，web容器根据“/hello”路径映射到DispatcherServlet（url-pattern为/）进行处理；
* DispatcherServlet——>BeanNameUrlHandlerMapping进行请求到处理的映射，BeanNameUrlHandlerMapping将“/hello”路径直接映射到名字为“/hello”的Bean进行处理，即HelloWorldController，BeanNameUrlHandlerMapping将其包装为HandlerExecutionChain（只包括HelloWorldController处理器，没有拦截器）；
* DispatcherServlet——> SimpleControllerHandlerAdapter，SimpleControllerHandlerAdapter将HandlerExecutionChain中的处理器（HelloWorldController）**适配**为SimpleControllerHandlerAdapter；
* SimpleControllerHandlerAdapter——> HelloWorldController处理器功能处理方法的调用，SimpleControllerHandlerAdapter将会调用处理器的handleRequest方法进行功能处理，该处理方法返回一个ModelAndView给DispatcherServlet；
* hello（ModelAndView的逻辑视图名）——>InternalResourceViewResolver， InternalResourceViewResolver使用JstlView，具体视图页面在/WEB-INF/jsp/hello.jsp；
* JstlView（/WEB-INF/jsp/hello.jsp）——>渲染，将在处理器传入的模型数据(message=HelloWorld！)在视图中展示出来；
* 返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。

**7）POST中文乱码解决方案**

spring Web MVC框架提供了org.springframework.web.filter.CharacterEncodingFilter用于解决POST方式造成的中文乱码问题，具体配置如下：

<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>

**4、Spring新特性**

Spring2.5引入注解式处理器支持，通过@**Controller** 和 @**RequestMapping**注解定义我们的处理器类。并且提供了一组强大的注解：需要通过处理器映射**DefaultAnnotationHandlerMapping**和处理器适配器**AnnotationMethodHandlerAdapter**来开启支持@Controller 和 @RequestMapping注解的处理器。

* **@Controller**：用于标识是处理器类；
* **@RequestMapping**：请求到处理器功能方法的映射规则；
* **@RequestParam**：请求参数到处理器功能处理方法的方法参数上的绑定；
* **@ModelAttribute**：请求参数到**命令对象**的绑定；
* **@SessionAttributes**：用于声明session级别存储的属性，放置在处理器类上，通常列出模型属性（如@ModelAttribute）对应的名称，则这些属性会透明的保存到session中；
* **@InitBinder**：自定义数据绑定注册支持，用于将请求参数转换到命令对象属性的对应类型；

Spring3.0引入RESTful架构风格支持(通过@PathVariable注解和一些其他特性支持),且又引入了更多的注解支持：

* **@CookieValue**：cookie**数据**到处理器**功能处理方法的方法参数**上的绑定；
* **@RequestHeader**：请求头（header）数据到处理器功能处理方法的方法参数上的绑定；
* **@RequestBody**：请求的body体的绑定（通过HttpMessageConverter进行类型转换）；
* **@ResponseBody**：处理器功能处理方法的**返回值**作为响应体（通过HttpMessageConverter进行**类型转换**）；
* **@ResponseStatus**：定义处理器功能处理方法/异常处理器返回的状态码和原因；
* **@ExceptionHandler**：注解式声明异常处理器；
* **@PathVariable**：请求URI中的模板变量部分到处理器功能处理方法的方法参数上的绑定，从而支持RESTful架构风格的URI；

还有比如：

* JSR-303验证框架的无缝支持（通过**@Valid**注解**定义验证元数据**）；
* 使用Spring 3开始的ConversionService进行类型转换（PropertyEditor依然有效），支持使用**@NumberFormat** 和 **@DateTimeFormat**来进行**数字和日期的格式化**；
* **HttpMessageConverter**（Http输入/输出**转换器**，比如JSON、XML等的数据输出转换器）；
* **ContentNegotiatingViewResolver**，**内容协商视图解析器**，它还是**视图解析器**，只是它**支持根据请求信息将同一模型数据以不同的视图方式展示**（如json、xml、html等），RESTful架构风格中很重要的概念（**同一资源，多种表现形式**）；

Spring 3 引入 一个 mvc XML的命名空间用于支持mvc配置，包括如：

* **<mvc:annotation-driven>**：
* 自动注册基于注解风格的处理器需要的DefaultAnnotationHandlerMapping、AnnotationMethodHandlerAdapter；
* 支持Spring3的ConversionService自动注册；
* 支持JSR-303验证框架的自动探测并注册（只需把JSR-303实现放置到classpath）；
* **自动注册相应的HttpMessageConverter**（用于支持@RequestBody 和 @ResponseBody）（如XML输入输出转换器（只需将JAXP**实现**放置到**classpath**）、JSON输入输出转换器（只需将Jackson实现放置到classpath））等。
* **<mvc:interceptors>**：注册自定义的处理器拦截器；
* **<mvc:view-controller>**：和ParameterizableViewController类似，收到相应请求后直接选择相应的视图；
* **<mvc:resources>**：逻辑静态资源路径到物理静态资源路径的支持；
* **<mvc:default-servlet-handler>**：当在web.xml 中DispatcherServlet使用<url-pattern>/</url-pattern> 映射时，能映射静态资源（当Spring Web MVC框架没有处理请求对应的控制器时（如一些静态资源），转交给默认的Servlet来响应静态文件，否则报404找不到资源错误，）。

对Servlet 3.0的全面支持：

* **@EnableWebMvc**：用于在基于Java类定义Bean配置中开启MVC支持，和XML中的<mvc:annotation-driven>功能一样；
* **新的@Contoller和@RequestMapping注解支持类**：处理器映射RequestMappingHandlerMapping 和 处理器适配器RequestMappingHandlerAdapter组合来代替Spring2.5开始的处理器映射DefaultAnnotationHandlerMapping和处理器适配器AnnotationMethodHandlerAdapter，提供更多的扩展点，它们之间的区别我们在处理器映射一章介绍。
* **新的@ExceptionHandler 注解支持类**：ExceptionHandlerExceptionResolver来代替Spring3.0的AnnotationMethodHandlerExceptionResolver，在异常处理器一章我们再详细讲解它们的区别。
* **@RequestMapping的"consumes" 和 "produces" 条件支持**：用于支持@RequestBody 和 @ResponseBody，
* consumes**指定请求的内容是什么类型的内容**，即本处理方法消费什么类型的数据，如consumes="application/json"表示JSON类型的内容，Spring会根据相应的HttpMessageConverter进行请求内容区数据到@RequestBody注解的命令对象的转换；
* produces**指定生产什么类型的内容**，如produces="application/json"表示JSON类型的内容，Spring的根据相应的HttpMessageConverter进行请求内容区数据到@RequestBody注解的命令对象的转换，Spring会根据相应的HttpMessageConverter进行模型数据（返回值）到JSON响应内容的转换
* 以上内容，本章第×××节详述。
* **URI模板变量增强**：URI模板变量可以直接绑定到**@ModelAttribute指定的命令对象**、**@PathVariable**方法参数在视图渲染之前被合并到模型数据中（除JSON序列化、XML混搭场景下）。
* **@Validated**：JSR-303的javax.validation.Valid一种变体（非JSR-303规范定义的，而是Spring自定义的），用于提供对**Spring的验证器**（org.springframework.validation.Validator）支持，需要hibernate Validator 4.2及更高版本支持；
* **@RequestPart：**提供对**“multipart/form-data”**请求的全面支持，支持Servlet 3.0文件上传（javax.servlet.http.Part）、支持内容的HttpMessageConverter（即根据请求头的Content-Type，来判断内容区数据是什么类型，如JSON、XML，能**自动转换为命令对象**），比@**RequestParam**更强大（只能对请求参数数据绑定，key-alue格式），而@RequestPart支持如JSON、XML内容区数据的绑定；详见本章的第×××节；
* **Flash 属性 和 RedirectAttribute**：通过FlashMap存储一个请求的输出，当进入另一个请求时作为该请求的输入，典型场景如重定向（POST-REDIRECT-GET模式，1、POST时将下一次需要的数据放在FlashMap；2、重定向；3、通过GET访问重定向的地址，此时FlashMap会把1放到FlashMap的数据取出放到请求中，并从FlashMap中删除；从而支持在两次请求之间保存数据并防止了重复表单提交）。

Spring Web MVC提供FlashMapManager用于管理FlashMap，默认使用SessionFlashMapManager，即数据默认存储在session中。

**六、DispatcherServlet详解**

**1、作用**

DispatcherServlet是前端控制器设计模式的实现，提供spring Web MVC的**集中访问点**，而且负责职责的分派，而且与Spring IoC容器无缝集成，从而可以获得Spring的所有好处。

DispatcherServlet主要用作职责调度工作，本身主要用于控制流程，主要职责如下：

* 文件上传解析，如果请求类型是multipart将通过MultipartResolver进行文件上传解析；
* 通过HandlerMapping，将请求映射到处理器（返回一个HandlerExecutionChain，它包括一个处理器、多个HandlerInterceptor拦截器）；
* 通过HandlerAdapter支持多种类型的处理器(HandlerExecutionChain中的处理器)；
* 通过ViewResolver解析逻辑视图名到具体视图实现；
* 本地化解析；
* 渲染具体的视图等；
* 如果执行过程中遇到异常将交给HandlerExceptionResolver来解析。

**2、在web.xml中的配置**

<!-- 入口配置(前端控制器配置)：DispatcherServlet -->

<servlet>

<servlet-name>spring-mvc</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

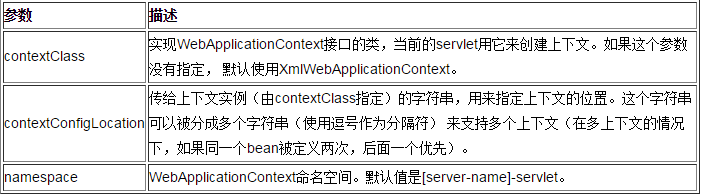
<servlet-name>spring-mvc</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

DispatcherServlet默认使用WebApplicationContext作为上下文，Spring默认配置文件为“/WEB-INF/[servlet名字]-servlet.xml”。

DispatcherServlet也可以配置自己的初始化参数，覆盖默认配置：



添加初始化参数：

<servlet>

<servlet-name>chapter2</servlet-name>

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

<load-on-startup>1</load-on-startup>

<init-param>

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

<param-value>classpath:spring-servlet-config.xml</param-value>

</init-param>

</servlet>

如果使用如上配置，Spring Web MVC框架将加载“classpath:spring-servlet-config.xml”来进行初始化上下文而不是“/WEB-INF/[servlet名字]-servlet.xml”。

**3、上下文关系**

集成Web环境的通用配置：

<context-param>

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

<param-value>

classpath:spring-common-config.xml,

classpath:spring-budget-config.xml

</param-value>

</context-param>

<listener>

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

ContextLoaderListener</listener-class>

</listener>

如上配置是Spring集成Web环境的通用配置；一般**用于加载除Web层的Bean（如DAO、Service等）**，以便于**与其他任何Web框架集成**。

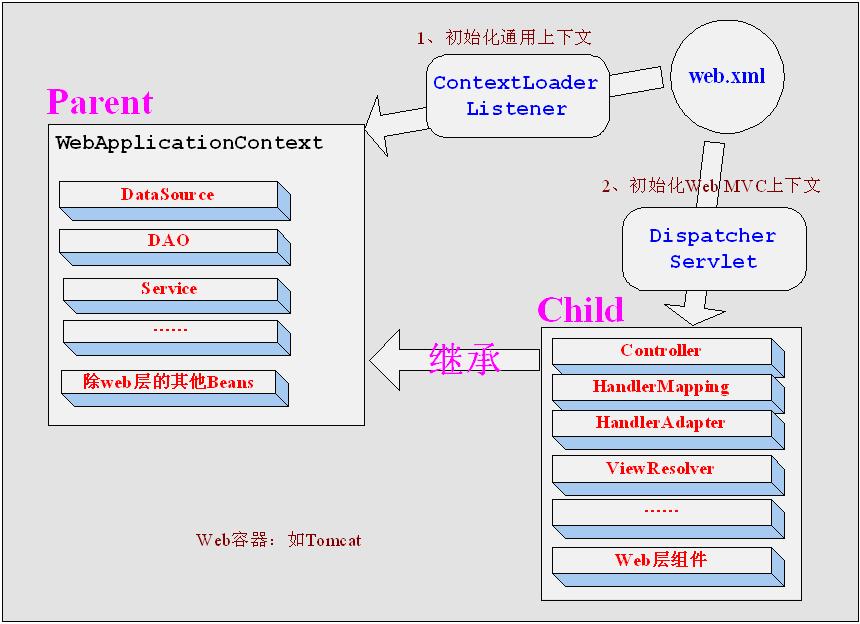
**contextConfigLocation**：表示用于加载Bean的配置文件；

**contextClass**：表示用于加载Bean的ApplicationContext实现类，默认**WebApplicationContext**。

创建完毕后会将该上下文放在ServletContext：

servletContext.setAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE,**this**.context);

**ContextLoaderListener**初始化的上下文和**DispatcherServlet**初始化的上下文关系，如图



从图中可以看出：

**ContextLoaderListener**初始化的上下文加载的Bean是**对于整个应用程序共享**的，不管是使用什么表现层技术，一般如DAO层、Service层Bean；

**DispatcherServlet**初始化的上下文加载的Bean是**只对Spring Web MVC有效的Bean**，如Controller、HandlerMapping、HandlerAdapter等等，该初始化上下文应该只加载Web相关组件。

整个DispatcherServlet初始化的过程，具体主要做了如下两件事情：

1、初始化Spring Web MVC使用的Web上下文，并且可能指定父容器为（ContextLoaderListener加载了根上下文）；

2、初始化DispatcherServlet使用的策略，如HandlerMapping、HandlerAdapter等。

**4、DispatcherServlet中使用的特殊的Bean**

DispatcherServlet默认使用WebApplicationContext作为上下文，因此我们来看一下该上下文中有哪些特殊的Bean：

* **Controller**：处理器/页面控制器，做的是MVC中的C的事情，但控制逻辑转移到前端控制器了，用于对请求进行处理；
* **HandlerMapping**：请求到处理器的映射，如果映射成功返回一个HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象；如BeanNameUrlHandlerMapping将URL与Bean名字映射，映射成功的Bean就是此处的处理器；
* **HandlerAdapter**：HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；如SimpleControllerHandlerAdapter将对实现了Controller接口的Bean进行适配，并且掉处理器的handleRequest方法进行功能处理；
* **ViewResolver**：ViewResolver将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；如**InternalResourceViewResolver**将**逻辑视图名**映射为**jsp视图**；
* **LocalResover**：本地化解析，因为Spring支持国际化，因此LocalResover解析客户端的Locale信息从而方便进行国际化；
* **ThemeResovler**：主题解析，通过它来实现一个页面多套风格，即常见的类似于软件皮肤效果；
* **MultipartResolver**：文件上传解析，用于支持文件上传；
* **HandlerExceptionResolver**：处理器异常解析，可以将异常映射到相应的统一错误界面，从而显示用户友好的界面（而不是给用户看到具体的错误信息）；
* **RequestToViewNameTranslator**：当处理器没有返回逻辑视图名等相关信息时，自动将请求URL映射为逻辑视图名；
* **FlashMapManager**：用于管理FlashMap的策略接口，FlashMap用于存储一个请求的输出，当进入另一个请求时作为该请求的输入，通常用于重定向场景，后边会细述。

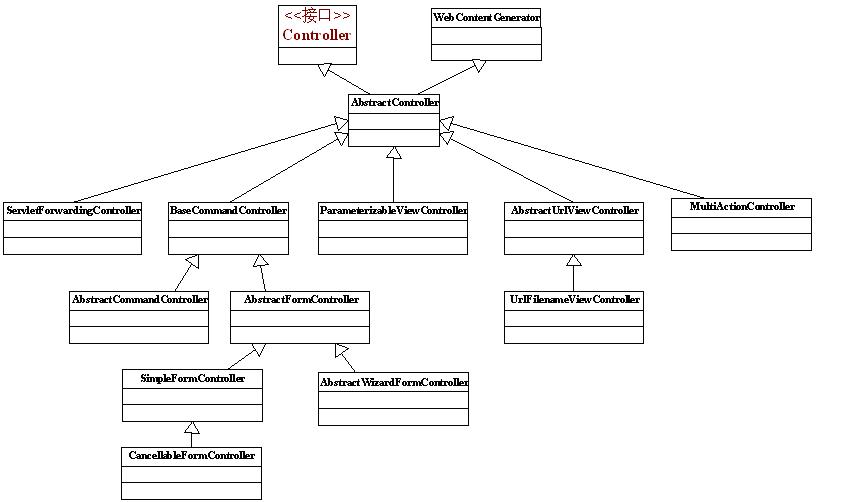
**七、Controller详解**

Controller控制器，是MVC中的部分C，为什么是部分呢？因为此处的控制器主要负责功能处理部分：

* 收集、验证请求参数并绑定到命令对象；
* 将命令对象交给业务对象，由业务对象处理并返回模型数据；
* 返回ModelAndView（Model部分是业务对象返回的模型数据，视图部分为逻辑视图名）。

**1、Controller接口**

Spring默认提供了一些Controller接口的实现以便使用，具体继承体系如图：



………

**八、处理器拦截器**

**1、简介**

spring Web MVC的处理器拦截器（如无特殊说明，下文所说的拦截器即处理器拦截器）。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

**常见应用场景：**

* 日志记录：记录请求信息的日志，以便进行信息监控、信息统计、计算PV（Page View）等。
* 权限检查：如登录检测，进入处理器检测检测是否登录，如果没有直接返回到登录页面；
* 性能监控：有时候系统在某段时间莫名其妙的慢，可以通过拦截器在进入处理器之前记录开始时间，在处理完后记录结束时间，从而得到该请求的处理时间（如果有反向代理，如apache可以自动记录）；
* 通用行为：读取cookie得到用户信息并将用户对象放入请求，从而方便后续流程使用，还有如提取Locale、Theme信息等，只要是多个处理器都需要的即可使用拦截器实现。
* OpenSessionInView：如hibernate，在进入处理器打开Session，在完成后关闭Session。

本质也是AOP（面向切面编程），也就是说符合横切关注点的所有功能都可以放入拦截器实现。

**2、拦截器接口**

**package** org.springframework.web.servlet;

**public** **interface** HandlerInterceptor {

**boolean** preHandle(

HttpServletRequest request, HttpServletResponse response,

Object handler)

**throws** Exception;

**void** postHandle(

HttpServletRequest request, HttpServletResponse response,

Object handler, ModelAndView modelAndView)

**throws** Exception;

**void** afterCompletion(

HttpServletRequest request, HttpServletResponse response,

Object handler, Exception ex)

**throws** Exception;

}

* **preHandle**：预处理回调方法，实现处理器的预处理（如登录检查），第三个参数为响应的处理器（如我们上一章的Controller实现）；

**返回值**：true表示继续流程（如调用下一个拦截器或处理器）；

false表示流程中断（如登录检查失败），不会继续调用其他的**拦截器或处理器**，此时我们需要通过response来产生响应；

* **postHandle**：后处理回调方法，实现处理器的后处理（但在渲染视图之前），此时我们可以通过**modelAndView**（模型和视图对象）对**模型数据**进行处理或对视图进行处理，modelAndView也可能为null。
* **afterCompletion**：整个请求处理完毕回调方法，即在视图渲染完毕时回调，如性能监控中我们可以在此记录结束时间并输出消耗时间，还可以进行一些资源清理，类似于try-catch-finally中的finally，但仅调用处理器执行链中preHandle返回true的拦截器的afterCompletion。

**3、拦截器适配器**

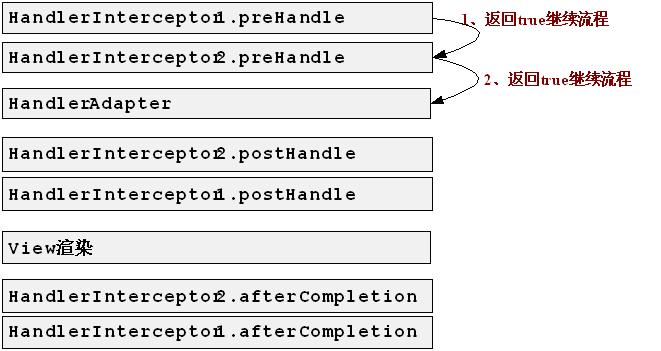
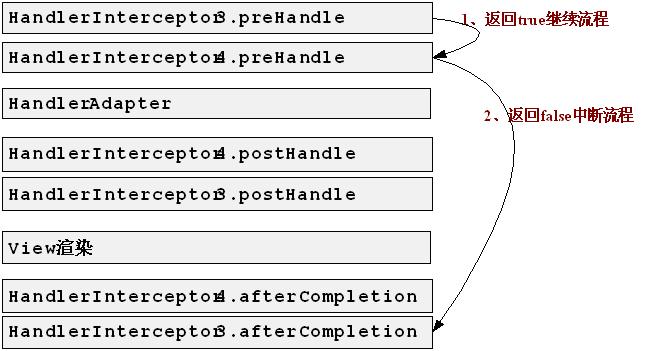
有时候我们可能只需要实现三个回调方法中的某一个，如果实现**HandlerInterceptor**接口的话，三个方法必须实现，不管你需不需要，此时spring提供了一个HandlerInterceptorAdapter适配器（一种适配器设计模式的实现），允许我们只实现需要的回调方法。

**public** **abstract** **class** HandlerInterceptorAdapter **implements** HandlerInterceptor {

//省略代码 此处所以三个回调方法都是空实现，preHandle返回true。

}

**4、运行流程图**

左图为正常流程图，右图为中断流程。

中断流程中，比如是HandlerInterceptor2中断的流程（preHandle返回false），此处仅调用它之前拦截器的preHandle返回true的afterCompletion方法。

**DispatcherServlet内部工作：**

//doDispatch方法

//1、处理器拦截器的预处理（正序执行）

HandlerInterceptor[] interceptors = mappedHandler.getInterceptors();

**if** (interceptors != **null**) {

**for** (**int** i = 0; i < interceptors.length; i++) {

HandlerInterceptor interceptor = interceptors[i];

**if** (!interceptor.preHandle(processedRequest, response, mappedHandler.getHandler())) {

//1.1、失败时触发afterCompletion的调用

triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, **null**);

**return**;

}

interceptorIndex = i;//1.2、记录当前预处理成功的索引

}

}

//2、处理器适配器调用我们的处理器

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

//当我们返回null或没有返回逻辑视图名时的默认视图名翻译（详解4.15.5 RequestToViewNameTranslator）

**if** (mv != **null** && !mv.hasView()) {

mv.setViewName(getDefaultViewName(request));

}

//3、处理器拦截器的后处理（逆序）

**if** (interceptors != **null**) {

**for** (**int** i = interceptors.length - 1; i >= 0; i--) {

HandlerInterceptor interceptor = interceptors[i];

interceptor.postHandle(processedRequest, response, mappedHandler.getHandler(), mv);

}

}

//4、视图的渲染

**if** (mv != **null** && !mv.wasCleared()) {

render(mv, processedRequest, response);

**if** (errorView) {

WebUtils.clearErrorRequestAttributes(request);

}

//5、触发整个请求处理完毕回调方法afterCompletion

triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, **null**);

注：以上是流程的简化代码，中间省略了部分代码，不完整。

//triggerAfterCompletion方法

**private** **void** triggerAfterCompletion(HandlerExecutionChain mappedHandler, **int** interceptorIndex,

HttpServletRequest request, HttpServletResponse response, Exception ex) **throws** Exception {

// 5、触发整个请求处理完毕回调方法afterCompletion （逆序从1.2中的预处理成功的索引处的拦截器执行）

**if** (mappedHandler != **null**) {

HandlerInterceptor[] interceptors = mappedHandler.getInterceptors();

**if** (interceptors != **null**) {

**for** (**int** i = interceptorIndex; i >= 0; i--) {

HandlerInterceptor interceptor = interceptors[i];

**try** {

interceptor.afterCompletion(request, response, mappedHandler.getHandler(), ex);

}**catch** (Throwable ex2) {

logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);

} } } } }

**5、应用**

**1）性能监控**

记录一下请求的处理时间。

实现分析：

1、在进入处理器之前记录开始时间，即在拦截器的preHandle记录开始时间；

2、在结束请求处理之后记录结束时间，即在拦截器的afterCompletion记录结束实现，并用结束时间-开始时间得到这次请求的处理时间。

**问题：**

我们的拦截器是单例，因此不管用户请求多少次都只有一个拦截器实现，即线程不安全，那我们应该怎么记录时间呢？

解决方案是使用**ThreadLocal**，它是**线程绑定的变量，提供线程局部变量**（一个线程一个ThreadLocal，A线程的ThreadLocal只能看到A线程的ThreadLocal，不能看到B线程的ThreadLocal）。

代码实现：

**package** cn.javass.chapter5.web.interceptor;

**public** **class** StopWatchHandlerInterceptor **extends** HandlerInterceptorAdapter {

**private** NamedThreadLocal<Long> startTimeThreadLocal =

**new** NamedThreadLocal<Long>("StopWatch-StartTime");

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response,

Object handler) **throws** Exception {

**long** beginTime = System.currentTimeMillis();//1、开始时间

startTimeThreadLocal.set(beginTime);//线程绑定变量（该数据只有当前请求的线程可见）

**return** **true**;//继续流程

}

@Override

**public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response,

Object handler, Exception ex) **throws** Exception {

**long** endTime = System.currentTimeMillis();//2、结束时间

**long** beginTime = startTimeThreadLocal.get();//得到线程绑定的局部变量（开始时间）

**long** consumeTime = endTime - beginTime;//3、消耗的时间

**if**(consumeTime > 500) {//此处认为处理时间超过500毫秒的请求为慢请求

//**TODO** 记录到日志文件

System.out.println(

String.format("%s consume %d millis", request.getRequestURI(), consumeTime));

}

}

}

**NamedThreadLocal**：Spring提供的一个**命名的ThreadLocal（线程本地对象）实现**。

在测试时需要把stopWatchHandlerInterceptor放在拦截器链的第一个，这样得到的时间才是比较准确的。

**2）登录检测**

在访问某些资源时（如订单页面），需要用户登录后才能查看，因此需要进行登录检测。

**流程：**

* 访问需要登录的资源时，由拦截器重定向到登录页面；
* 如果访问的是登录页面，拦截器不应该拦截；
* 用户登录成功后，往cookie/session添加登录成功的标识（如用户编号）；
* 下次请求时，拦截器通过判断cookie/session中是否有该标识来决定继续流程还是到登录页面；
* 在此拦截器还应该允许游客访问的资源。

拦截器代码如下所示：

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response,

Object handler) **throws** Exception {

//1、请求到登录页面 放行

**if**(request.getServletPath().startsWith(loginUrl)) {

**return** **true**;

}

//2、**TODO** 比如退出、首页等页面无需登录，即此处要放行 允许游客的请求

//3、如果用户已经登录 放行

**if**(request.getSession().getAttribute("username") != **null**) {

//更好的实现方式的使用cookie

**return** **true**;

}

//4、非法请求 即这些请求需要登录后才能访问

//重定向到登录页面

response.sendRedirect(request.getContextPath() + loginUrl);

**return** **false**;

}

**提示：**推荐能使用servlet规范中的过滤器Filter实现的功能就用Filter实现，因为HandlerInteceptor只有在Spring Web MVC环境下才能使用，因此**Filter是最通用的、最先应该使用的**。如登录这种拦截器最好使用Filter来实现

**九、注解式控制器详解**

**1、简介**

Spring注解控制器的新特性如上第五章（Spring MVC补充）第4小节所述…

**URL路径映射：**使用URL映射请求到处理器的功能处理方法；

**请求方法映射限定：**如限定功能处理方法只处理GET请求；

**请求参数映射限定：**如限定只处理包含“abc”请求参数的请求；

**请求头映射限定：**如限定只处理“Accept=application/json”的请求。

* **HandlerMapping实现**：使用**DefaultAnnotationHandlerMapping**（spring3.1之前）或**RequestMappingHandlerMapping**（spring3.1）替换之前的**BeanNameUrlHandlerMapping**。注解式处理器映射会扫描spring容器中的bean，发现bean实现类上拥有，@Controller或@RequestMapping注解的bean，并将它们作为**处理器**。
* **HandlerAdapter实现**：使用**AnnotationMethodHandlerAdapter**（spring3.1之前）或**RequestMappingHandlerAdapter**（spring3.1）替换之前的**SimpleControllerHandlerAdapter**。注解式处理器适配器会通过反射**调用相应的功能处理方法**（方法上拥有@RequestMapping注解）。

**2、请求映射**

处理器定义好了，那接下来我们应该**定义功能处理方法**，接收用户请求处理并选择视图进行渲染：



**http请求信息**包含六部分信息：

①请求方法，如GET或POST，表示提交的方式；

②URL，请求的地址信息；

③协议及版本；

④请求头信息（包括Cookie信息）；

⑤回车换行（CRLF）；

⑥请求内容区（即请求的内容或数据），如表单提交时的参数数据、URL请求参数（?abc=123 ？后边的）等。

那此处我们可以看到有①、②、④、⑥一般是可变的，因此我们可以这些信息进行请求到。处理器的功能处理方法的映射，因此**请求的映射**分为如下几种：

* URL路径映射：使用URL映射请求到处理器的功能处理方法；
* 请求方法映射限定：如限定功能处理方法只处理GET请求；
* 请求参数映射限定：如限定只处理包含“abc”请求参数的请求；
* 请求头映射限定：如限定只处理“**Accept=application/json**”的请求。

**1）URL路径映射**

**（1）普通URL路径映射**

@RequestMapping(value={"/test1", "/user/create"})：多个URL路径可以映射到同一个处理器的功能处理方法

**（2）URL模板模式映射**

* @RequestMapping(value="/users/{userId}")：{×××}占位符， 请求的URL可以是“/users/123456”或“/users/abcd”，通过6.6.5讲的通过**@PathVariable**可以提取URI模板模式中的{×××}中的×××变量。
* @RequestMapping(value="/users/{userId}/create")：这样也是可以的，请求的URL可以是“/users/123/create”。
* @RequestMapping(value="/users/{userId}/topics/{topicId}")：这样也是可以的，请求的URL可以是“/users/123/topics/123”。

**（3）Ant风格的URL路径映射**

* @RequestMapping(value="/users/\*\*")：可以匹配“/users/abc/abc”，但“/users/123”将会被【URI模板模式映射中的“/users/{userId}”模式优先映射到】【详见4.14的最长匹配优先】。
* @RequestMapping(value="/product?")：可匹配“/product1”或“/producta”，但不匹配“/product”或“/productaa”（注：?表示仅匹配一位数或字符）;
* @RequestMapping(value="/product\*")：可匹配“/productabc”或“/product”，但不匹配“/productabc/abc”;
* @RequestMapping(value="/product/\*")：可匹配“/product/abc”，但不匹配“/productabc”;
* @RequestMapping(value="/products/\*\*/{productId}")：可匹配“/products/abc/abc/123”或“/products/123”，也就是Ant风格和URI模板变量风格可**混用**;

**（4）正则表达式风格的URL路径映射**

从Spring3.0开始支持正则表达式风格的URL路径映射，格式为{变量名:正则表达式}，这样我们就可以通过6.6.5讲的通过@PathVariable提取模式中的{×××：正则表达式匹配的值}中的×××变量了。

@RequestMapping(value="/products/{categoryCode:\\d+}-{pageNumber:\\d+}")：可以匹配“/products/123-1”，但不能匹配“/products/abc-1”，这样可以设计更加严格的规则。

**正则表达式**风格的**URL路径映射**是一种**特殊的URI模板模式映射**：URI模板模式映射是{userId}，不能指定模板变量的数据类型，如是数字还是字符串；正则表达式风格的URL路径映射，可以指定模板变量的数据类型，可以将规则写的相当复杂。

**（5）组合使用是“或”的关系**

如@RequestMapping(value={"/test1", "/user/create"}) 组合使用**是或**的关系，即“/test1”或“/user/create”请求URL路径都可以映射到@RequestMapping指定的功能处理方法。

以上URL映射的测试类为：cn.javass.chapter6.web.controller.mapping.MappingController.Java。

到此，我们学习了spring Web MVC提供的**强大的URL路径映射**，而且可以实现非常复杂的URL规则。Spring Web MVC不仅仅提供URL路径映射，还提供了其他强大的映射规则。接下来我们看一下请求方法映射限定吧。

**2）请求方法映射限定：RequestMethod**

一般我们熟悉的表单一般分为两步：第一步**展示**，第二步**提交**，如4.9、SimpleFormController那样，那如何通过@RequestMapping来实现呢？

**（1）请求方法映射限定**

**展示表单**一般为**GET**请求方法；**提交表单**一般为**POST**请求方法。但6.5.1节讲的URL路径映射方式对任意请求方法是全盘接受的，因此我们需要某种方式来告诉相应的功能处理方法只处理如GET请求方法的请求或POST请求方法的请求。

接下来我们使用@RequestMapping来实现SimpleFormController的功能吧：

**package** cn.javass.chapter6.web.controller.method;

//省略import

@Controller

@RequestMapping("/customers/\*\*") //①处理器的通用映射前缀

**public** **class** RequestMethodController {

//②类级别的@RequestMapping窄化

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

**public** String showForm() {

System.out.println("===============GET");

**return** "customer/create";

}

//③类级别的@RequestMapping窄化

@RequestMapping(value="/create", method = RequestMethod.POST)

**public** String submit() {

System.out.println("================POST");

**return** "redirect:/success";

} }

**①**处理器的**通用映射前缀**（父路径）：表示该处理器只处理匹配“/customers/\*\*”的请求；

**②**对类级别的@RequestMapping进行**窄化**，表示showForm可处理匹配“/customers/\*\*/create”且请求方法为“GET”的请求；

**③**对类级别的@RequestMapping进行**窄化**，表示submit可处理匹配“/customers/\*\*/create”且请求方法为“POST”的请求。

**（2）组合使用是“或”的关系**

@RequestMapping(**value**="/methodOr", **method** = {RequestMethod.POST, RequestMethod.GET})：即请求方法可以是 GET 或 POST。

**提示：**

* 一般浏览器只支持GET、POST请求方法，如想浏览器支持PUT、DELETE等请求方法只能模拟，稍候章节介绍。
* 除了GET、POST，还有HEAD、OPTIONS、PUT、DELETE、TRACE。
* DispatcherServlet默认开启对 GET、POST、PUT、DELETE、HEAD的支持；
* 如果需要支持OPTIONS、TRACE，请添加DispatcherServlet在web.xml的初始化参数：dispatchOptionsRequest 和 dispatchTraceRequest 为true。

请求方法的详细使用请参考RESTful架构风格一章。

**3）请求参数数据映射限定：params**

**（1）请求数据中有指定参数名**

**package** cn.javass.chapter6.web.controller.parameter;

//省略import

@Controller

@RequestMapping("/parameter1") //①处理器的通用映射前缀

**public** **class** RequestParameterController1 {

//②进行类级别的@RequestMapping窄化

@RequestMapping(params="create", method=RequestMethod.GET)

**public** String showForm() {

System.out.println("===============showForm");

**return** "parameter/create";

}

//③进行类级别的@RequestMapping窄化

@RequestMapping(params="create", method=RequestMethod.POST)

**public** String submit() {

System.out.println("================submit");

**return** "redirect:/success";

} }

**②**@RequestMapping(params="create", method=RequestMethod.GET) ：表示请求中有**“create”的参数名**且请求方法为“GET”即可匹配，如可匹配的请求URL**“http://×××/parameter1?create”**；

**③**@RequestMapping(params="create", method=RequestMethod.POST)：表示请求中有“create”的参数名且请求方法为“POST”即可匹配；

此处的create请求参数名表示你**请求的动作**，即你想要的功能的一个标识，常见的CRUD(增删改查)我们可以使用如下请求参数名来表达：

* （create请求参数名 且 **GET**请求方法） 新增**页面展示**、（create请求参数名 且 POST请求方法）新增**提交**；
* （update请求参数名 且 GET请求方法） 新增页面展示、（update请求参数名 且 POST请求方法）新增提交；
* （delete请求参数名 且 GET请求方法） 新增页面展示、（delete请求参数名 且 POST请求方法）新增提交；
* （query请求参数名 且 GET请求方法） 新增页面展示、（query请求参数名 且 POST请求方法） 新增提交；
* （list请求参数名 且 GET请求方法） 列表页面展示；
* （view请求参数名 且 GET请求方法） 查看单条记录页面展示。

**（2）请求数据中没有指定参数名**

//请求参数不包含 create参数名

//进行类级别的@RequestMapping窄化

@RequestMapping(params="!create", method=RequestMethod.GET)

@RequestMapping(params="!create", method=RequestMethod.GET)：表示请求中**没有“create”参数名**且请求方法为“GET”即可匹配，如可匹配的请求URL“http://×××/parameter1?abc”。

**（3）请求数据中指定参数名=值**

**package** cn.javass.chapter6.web.controller.parameter;

//省略import

@Controller

@RequestMapping("/parameter2") //①处理器的通用映射前缀

**public** **class** RequestParameterController2 {

//②进行类级别的@RequestMapping窄化

@RequestMapping(params="submitFlag=create", method=RequestMethod.GET)

**public** String showForm() {

System.out.println("===============showForm");

**return** "parameter/create";

}

//③进行类级别的@RequestMapping窄化

@RequestMapping(params="submitFlag=create", method=RequestMethod.POST)

**public** String submit() {

System.out.println("===============submit");

**return** "redirect:/success";

}

}

**②**@RequestMapping(params="submitFlag=create", method=RequestMethod.GET)：表示请求中有**“submitFlag=create”请求参数**且请求方法为“GET”即可匹配，如请求URL为http://×××/parameter2?submitFlag=create；

**③**@RequestMapping(params="submitFlag=create", method=RequestMethod.POST)：表示请求中有“submitFlag=create”请求参数且请求方法为“POST”即可匹配；

此处的submitFlag=create请求参数**表示你请求的动作**，即你想要的功能的一个标识，常见的CRUD(增删改查)我们可以使用如下请求参数名来表达：

* （submitFlag=create请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=create请求参数名 且 POST请求方法） 新增提交；
* （submitFlag=update请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=update请求参数名 且 POST请求方法） 新增提交；
* （submitFlag=delete请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=delete请求参数名 且 POST请求方法） 新增提交；
* （submitFlag=query请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=query请求参数名 且 POST请求方法） 新增提交；
* （submitFlag=list请求参数名 且 GET请求方法） 列表页面展示；
* （submitFlag=view请求参数名 且 GET请求方法） 查看单条记录页面展示。

**（4）请求数据中指定参数名!=值**

//请求参数submitFlag 不等于 create

@RequestMapping(params="submitFlag!=create", method=RequestMethod.GET)

@RequestMapping(params="submitFlag!=create", method=RequestMethod.GET)：表示请求中的参数“submitFlag!=create”且请求方法为“GET”即可匹配，如可匹配的请求URL“http://×××/parameter1?submitFlag=abc”。

**（5）组合使用是“且”的关系**

@RequestMapping(params={"test1", "test2=create"}) //②进行类级别的@RequestMapping窄化

@RequestMapping(params={"test1", "test2=create"})：表示请求中的有“test1”参数名 且 **有**“test2=create”**参数即可匹配**，如可匹配的请求URL“http://×××/parameter3?test1&test2=create。

**4）请求头数据映射限定：headers**

**（1）请求头数据中有指定参数名**

**@RequestMapping**(value="/header/test1", headers = "Accept")：表示请求的URL必须为“/header/test1”且 请求头中**必须有Accept参数**才能匹配。

**@RequestMapping**(value="/header/test1", headers = "abc")：表示请求的URL必须为“/header/test1”且 请求头中必须有abc参数才能匹配。

**（2）请求头数据中没有指定参数名**

**@RequestMapping**(value="/header/test2", headers = "!abc")：表示请求的URL必须为“/header/test2” 且请求头中必须**没有abc参数才能匹配**。

**（3）请求头数据中指定参数名=值**

@RequestMapping(value="/header/test3", headers = "**Content-Type**=application/json")：表示请求的URL必须为“/header/test3” 且 请求头中必须有“Content-Type=application/json”参数即可匹配。

当你请求的URL为“/header/test3” 但 如果请求头中没有或不是“Content-Type=application/json”参数（如“text/html”其他参数），将返回“HTTP Status 415”状态码【表示不支持的媒体类型(Media Type)，也就是MIME类型】，即我们的功能处理方法只能处理application/json的媒体类型。

@RequestMapping(value="/header/test4", headers = "Accept=application/json")：表示请求的URL必须为“/header/test4” 且 请求头中必须有“Accept =application/json”参数即可匹配。（将Modify Header的Accept参数值改为“application/json”即可）；

当你请求的URL为“/header/test4” 但 如果请求头中没有“Accept=application/json”参数（如“text/html”其他参数），将返回“HTTP Status 406”状态码【不可接受，服务器无法根据Accept头的媒体类型为客户端生成响应】，即客户只接受“application/json”媒体类型的数据，即我们的功能处理方法的响应只能返回“application/json”媒体类型的数据。

@RequestMapping(value="/header/test5", headers = "Accept=text/\*") ：表示请求的URL必须为“/header/test5” 且 请求头中必须有如“Accept=text/plain”参数即可匹配。（将Modify Header的Accept参数值改为“text/plain”即可）；

**Accept=text/\*：**表示主类型为text，子类型任意，如“text/plain”、“text/html”等都可以匹配。

@RequestMapping(value="/header/test6", headers = "Accept=\*/\*") ：表示请求的URL必须为“/header/test6” 且 请求头中必须**有任意Accept参数即可匹配**。（将Modify Header的Accept参数值改为“text/html”或“application/xml”等都可以）。

**Accept=\*/\*：**表示主类型任意，子类型任意，如“text/plain”、“application/xml”等都可以匹配。

**（4）请求头数据中指定参数名!=值**

@RequestMapping(value="/header/test7", headers = "Accept!=text/vnd.wap.wml")：表示请求的URL必须为“/header/test7” 且 请求头中**必须有“Accept”参数**但**值不等于“text/vnd.wap.wml”**即可匹配。

**（5）组合使用是“且”的关系**

@RequestMapping(value="/header/test8", headers = {"Accept!=text/vnd.wap.wml","abc=123"})：表示请求的URL必须为“/header/test8” 且 请求头中必须有“Accept”参数但值不等于“text/vnd.wap.wml”且 请求中必须有**参数“abc=123”**即可匹配。

**注：**Accept:text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

如果您的请求中含有Accept：“\*/\*”，则可以匹配功能处理方法上的如“text/html”、“text/\*”，“application/xml”等。

**3、生产者、消费者限定**

**1）基本概念**

**（1）Media Type：媒体类型**

互联网媒体类型，一般就是我们所说的MIME类型，用来**确定请求的内容类型或响应的内容类型**。

* + 媒体类型格式：type/subtype(;parameter)?
  + type主类型，任意的字符串，如text，如果是\*号代表所有；
  + subtype 子类型，任意的字符串，如html，如果是\*号代表所有；
  + parameter 可选，一些参数，如Accept请求头的q参数， Content-Type的 charset参数。

**常见媒体类型：**

text/html ： HTML格式 text/plain ：纯文本格式 text/xml ：XML格式

image/gif ：gif图片格式 image/jpeg ：jpg图片格式 image/png：png图片格式

application/x-www-form-urlencoded ： <form encType=””>中默认的encType，form表单数据被编码为**key/value格式**发送到服务器（表单默认的**提交数据的格式**）。

**multipart/form-data** ： 当你需要在表单中进行**文件上传**时，就需要使用该格式；

application/xhtml+xml ：XHTML格式 application/xml ： XML数据格式

application/atom+xml ：Atom XML聚合格式 application/json ： JSON数据格式

application/pdf ：pdf格式 application/msword ： Word文档格式

application/octet-stream ： 二进制流数据（如常见的文件下载）。

在如tomcat服务器的 “conf/web.xml”中指定了扩展名到媒体类型的映射，在此我们可以看到服务器支持的媒体类型。

**（2）Content-Type：内容类型**

内容类型，即请求/响应的内容区数据的媒体类型；

**a.请求头的内容类型，表示发送到服务器的内容数据的媒体类型**

request中设置请求头“Content-Type: application/x-www-form-urlencoded”表示请求的数据为key/value数据。

**a）控制器1**

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

**public** String showForm() **throws** IOException {

//form表单，使用application/x-www-form-urlencoded编码方式提交表单

**return** "consumesproduces/Content-Type";

}

@RequestMapping(value = "/ContentType", method = RequestMethod.POST,

headers = "Content-Type=application/x-www-form-urlencoded")

**public** String request1(HttpServletRequest request) **throws** IOException {

//①得到请求的内容区数据的类型

String contentType = request.getContentType();

System.out.println("========ContentType:" + contentType);

//②得到请求的内容区数据的编码方式，如果请求中没有指定则为null

//注意，我们的CharacterEncodingFilter这个过滤器设置了编码(UTF-8)

//编码只能被指定一次，即如果客户端设置了编码，则过滤器不会再设置

String characterEncoding = request.getCharacterEncoding();

System.out.println("========CharacterEncoding:" + characterEncoding);

//③表示请求的内容区数据为form表单提交的参数，此时我们可以通过request.getParameter得到数据（key=value）

System.out.println(request.getParameter("realname"));

System.out.println(request.getParameter("username"));

**return** "success";

}

**showForm功能处理方式**：展示表单，且form的enctype="application/x-www-form-urlencoded"，在提交时请求的内容类型头为“Content-Type:application/x-www-form-urlencoded”；

**request1功能处理方法**：只对请求头为“Content-Type:application/x-www-form-urlencoded”的请求进行处理（即消费请求内容区数据）;

* + request.getContentType()：可以得到请求头的**内容区数据类型**（即Content-Type头的值）
  + request.getCharacterEncoding()：如“Content-Type:application/json;charset=GBK”,则得到的**编码**为“GBK”，否则如果你设置**过滤器**（CharacterEncodingFilter）则得到它设置的编码，否则返回null。
  + request.getParameter()：因为**请求的内容区数据**为application/x-www-form-urlencoded格式的数据，因此我们可以通过request.getParameter()得到相应参数数据。

request中设置请求头“Content-Type:application/json;charset=GBK”表示请求的内容区数据为json类型数据，且内容区的数据以GBK进行编码；

**b）控制器2**

@RequestMapping(value = "/request/ContentType", method = RequestMethod.POST,

headers = "Content-Type=application/json")

**public** String request2(HttpServletRequest request) **throws** IOException {

//①表示**请求的内容区数据为json数据**

InputStream is = request.getInputStream();

**byte** bytes[] = **new** **byte**[request.getContentLength()];

is.read(bytes);

//②得到请求中的内容区数据（以CharacterEncoding解码）

//此处得到数据后你可以通过如json-lib转换为其他对象

String jsonStr = **new** String(bytes, request.getCharacterEncoding());

System.out.println("json data：" + jsonStr);

**return** "success";

}

**request2功能处理方法**：只对请求头为“Content-Type:application/json”的进行请求处理（即消费请求内容区数据）;

* + request.getContentLength()：可以得到请求头的**内容区数据的长度**；
  + request.getCharacterEncoding()：如“Content-Type:application/json;charset=GBK”,则得到的编码为“GBK”，否则如果你设置过滤器（CharacterEncodingFilter）则得到它设置的编码，否则返回null。

我们得到json的字符串形式后就能很简单的转换为JSON相关的对象。

**c）客户端发送json数据请求**

//请求的地址

String url = "http://localhost:9080/springmvc-chapter6/request/ContentType";

//①创建Http Request(内部使用HttpURLConnection)

ClientHttpRequest request = **new** SimpleClientHttpRequestFactory().

createRequest(**new** URI(url), HttpMethod.POST);

//②设置请求头的内容类型头和内容编码（GBK）

request.getHeaders().set("Content-Type", "application/json;charset=gbk");

//③以GBK编码写出请求内容体

String jsonData = "{\"username\":\"zhang\", \"password\":\"123\"}";

request.getBody().write(jsonData.getBytes("gbk"));

//④发送请求并得到响应

ClientHttpResponse response = request.execute();

System.out.println(response.getStatusCode());

使用spring提供的Http客户端API **SimpleClientHttpRequestFactory**创建了请求并设置了请求的Content-Type和编码并在响应体中写回了json数据（即生产json类型的数据），此处是硬编码，实际工作可以使用json-lib等工具进行转换。

**b.响应头的内容类型，表示发送到客户端的内容数据类型，和请求头的内容类型类似，只是方向相反**

@RequestMapping("/response/ContentType")

**public** **void** response1(HttpServletResponse response) **throws** IOException {

//①表示响应的内容区数据的媒体类型为html格式，且编码为utf-8(客户端应该以utf-8解码)

response.setContentType("text/html;charset=utf-8");

//②写出响应体内容

response.getWriter().write("<font style='color:red'>hello</font>");

}

通过response.setContentType("text/html;charset=utf-8") 告诉客户端响应体媒体类型为html，编码为utf-8，大家可以通过chrome工具查看响应头为“Content-Type:text/html;charset=utf-8”，还一个“Content-Length:36”表示响应体大小

①客户端—发送请求—服务器：客户端通过请求头Content-Type指定内容体的媒体类型（即客户端此时是生产者），服务器根据Content-Type消费内容体数据（即服务器此时是消费者）；

**②服务器—发送请求—客户端**：服务器生产**响应头**Content-Type指定的响应体数据（即服务器此时是生产者），客户端根据Content-Type消费内容体数据（即客户端此时是消费者）。

**问题：**

**①**服务器端可以通过指定【headers = "Content-Type=application/json"】来声明可处理（可消费）的媒体类型，即只消费Content-Type指定的请求内容体数据；

**②**客户端如何告诉服务器端它只消费什么媒体类型的数据呢？即客户端接受（需要）什么类型的数据呢？服务器应该生产什么类型的数据？此时我们可以请求的Accept请求头来实现这个功能。

**（3）Accept**

用来指定什么媒体类型的响应是**可接受的**，即告诉服务器我需要什么媒体类型的数据，此时服务器应该根据Accept请求头生产指定媒体类型的数据。

**a.json数据**

**a）服务器端控制器**

@RequestMapping(value = "/response/ContentType", headers = "Accept=application/json")

**public** **void** response2(HttpServletResponse response) **throws** IOException {

//①表示响应的内容区数据的媒体类型为json格式，且编码为utf-8(客户端应该以utf-8解码)

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

//②写出响应体内容

String jsonData = "{\"username\":\"zhang\", \"password\":\"123\"}";

response.getWriter().write(jsonData);

}

服务器根据请求头“Accept=application/json”生产json数据。

**b）客户端端接收服务器端json数据响应**

**private** **static** **void** jsonRequest() **throws** IOException, URISyntaxException {

//请求的地址

String url = "http://localhost:9080/springmvc-chapter6/response/ContentType";

//①创建Http Request(内部使用HttpURLConnection)

ClientHttpRequest request =

**new** SimpleClientHttpRequestFactory().

createRequest(**new** URI(url), HttpMethod.POST);

//②设置客户端可接受的媒体类型（即需要什么类型的响应体数据）

request.getHeaders().set("Accept", "application/json");

//③发送请求并得到响应

ClientHttpResponse response = request.execute();

//④得到响应体的编码方式

Charset charset = response.getHeaders().getContentType().getCharSet();

//⑤得到响应体的内容

InputStream is = response.getBody();

**byte** bytes[] = **new** **byte**[(**int**)response.getHeaders().getContentLength()];

is.read(bytes);

String jsonData = **new** String(bytes, charset);

System.out.println("charset : " + charset + ", json data : " + jsonData);

}

**request.getHeaders().set("Accept", "application/json")**：表示客户端只接受（即只消费）json格式的响应数据；

**response.getHeaders()**：可以得到响应头，从而可以得到响应体的内容类型和编码、内容长度。

**b.xml数据**

**a）服务器端控制器**

@RequestMapping(value = "/response/ContentType", headers = "Accept=application/xml")

**public** **void** response3(HttpServletResponse response) **throws** IOException {

//①表示响应的内容区数据的媒体类型为xml格式，且编码为utf-8(客户端应该以utf-8解码)

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

//②写出响应体内容

String xmlData = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>";

xmlData += "<user><username>zhang</username><password>123</password></user>";

response.getWriter().write(xmlData);

}

和生产json数据唯一不同的两点：请求头为“Accept=application/xml”，响应体数据为xml。

**b）客户端端接收服务器端xml数据响应**

**private** **static** **void** xmlRequest() **throws** IOException, URISyntaxException {

//请求的地址

String url = "http://localhost:9080/springmvc-chapter6/response/ContentType";

//①创建Http Request(内部使用HttpURLConnection)

ClientHttpRequest request = **new** SimpleClientHttpRequestFactory().

createRequest(**new** URI(url), HttpMethod.POST);

//②设置客户端可接受的媒体类型（即需要什么类型的响应体数据）

request.getHeaders().set("Accept", "application/xml");

//③发送请求并得到响应

ClientHttpResponse response = request.execute();

//④得到响应体的编码方式

Charset charset = response.getHeaders().getContentType().getCharSet();

//⑤得到响应体的内容

InputStream is = response.getBody();

**byte** bytes[] = **new** **byte**[(**int**)response.getHeaders().getContentLength()];

is.read(bytes);

String xmlData = **new** String(bytes, charset);

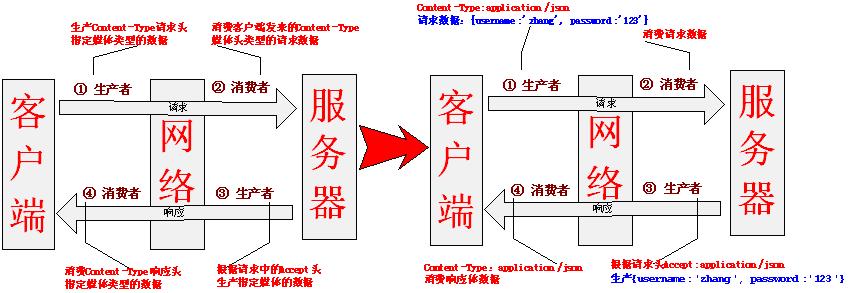
System.out.println("charset : " + charset + ", xml data : " + xmlData);

}

**request.getHeaders().set("Accept", "application/xml")**：表示客户端只接受（即只消费）xml格式的响应数据；

**response.getHeaders()**：可以得到响应头，从而可以得到响应体的内容类型和编码、内容长度。

**2）生产者消费者流程图**



从图看出：

**请求阶段：**客户端是生产者【生产Content-Type媒体类型的请求内容区数据】，服务器是消费者【消费客户端生产的Content-Type媒体类型的请求内容区数据】；

**响应阶段：**服务器是生产者【生产客户端请求头参数Accept指定的响应体数据】，客户端是消费者【消费服务器根据Accept请求头生产的响应体数据】。

**3）生产者、消费者限定**

Spring3.1开始支持**消费者、生产者限定**，而且必须使用如下**HandlerMapping**和**HandlerAdapter**才支持：

<!--Spring3.1开始的注解 HandlerMapping -->

<bean

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

<!--Spring3.1开始的注解 HandlerAdapter -->

<bean

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

**（1）功能处理方法是消费者**

@RequestMapping(value = "/consumes", **consumes** = {"application/json"})：此处使用consumes来**指定功能处理方法能消费的媒体类型**，其通过请求头的“Content-Type”来判断。此种方式相对使用@RequestMapping的“**headers = "Content-Type=application/json"**”更能表明你的目的。

服务器控制器代码详解cn.javass.chapter6.web.controller.consumesproduces.ConsumesController；

客户端代码类似于之前的Content-Type中的客户端，详见ConsumesClient.Java代码。

**（2）功能处理方法是生产者**

@RequestMapping(value = "/produces", **produces** = "application/json")：表示**功能处理方法将生产json格式的数据**，此时根据请求头中的**Accept**进行匹配，如请求头“Accept:application/json”时即可匹配;

@RequestMapping(value = "/produces", **produces** = "application/xml")：表示**功能处理方法将生产xml格式的数据**，此时根据请求头中的Accept进行匹配，如请求头“Accept:application/xml”时即可匹配。此种方式相对使用@RequestMapping的“**headers = "Accept=application/json"**”更能表明你的目的。

服务器控制器代码详解cn.javass.chapter6.web.controller.consumesproduces.ProducesController；

客户端代码类似于之前的Content-Type中的客户端，详见ProducesController.java代码。

当有如下Accept头：

* **Accept：**text/html,application/xml,application/json

将按照如下顺序进行**produces**的匹配 ①text/html ②application/xml ③application/json

* **Accept**：application/xml;q=0.5,application/json;q=0.9,text/html

将按照如下顺序进行produces的匹配 ①text/html ②application/json ③application/xml

q参数为**媒体类型的质量因子**，**越大则优先权越高**(从0到1)

* **Accept：**\*/\*,text/\*,text/html

将按照如下顺序进行produces的匹配 ①text/html ②text/\* ③\*/\*

即匹配规则为：**最明确的优先匹配**。

代码详见ProducesPrecedenceController1、ProducesPrecedenceController2、ProducesPrecedenceController3。

Accept详细信息，请参考http://tools.ietf.org/html/rfc2616#section-14.1。

**（3）窄化时是覆盖而非继承**

如类级别的映射为@RequestMapping(value="/narrow", produces="text/html")，方法级别的为@RequestMapping(produces="application/xml")，此时方法级别的映射**将覆盖类级别**的，因此请求头“Accept:application/xml”是成功的，而“text/html”将报406错误码，表示不支持的请求媒体类型。

详见cn.javass.chapter6.web.controller.consumesproduces.NarrowController。

**只有生产者/消费者 模式 是 覆盖，其他的使用方法是继承，如headers、params等都是继承**。

**（4）组合使用是“或”的关系**

@RequestMapping(produces={"text/html", "application/json"}) ：将匹配“Accept:text/html”或“Accept:application/json”。

**（5）问题**

**消费的数据**，如JSON数据、XML数据都是由我们读取请求的**InputStream**并根据需要自己转换为相应的模型数据，比较麻烦；

**生产的数据**，如JSON数据、XML数据都是由我们自己先**把模型数据**转换为json/xml等数据，然后输出响应流，也是比较麻烦的。

**4、数据绑定**

从请求中收集到哪些数据：



* @RequestParam绑定单个请求参数值；
* @PathVariable绑定URI模板变量值；
* @CookieValue绑定Cookie数据值
* @RequestHeader绑定请求头数据；
* @ModelAttribute绑定参数到命令对象；
* @SessionAttributes绑定命令对象到session；
* @RequestBody绑定请求的内容区数据并能进行**自动类型转换**等。
* @RequestPart绑定“multipart/data”数据，除了能绑定@RequestParam能做到的请求参数外，还能绑定上传的文件等。

除了上边提到的注解，我们还可以通过如HttpServletRequest等API得到请求数据，但推荐使用注解方式，因为使用起来更简单。

**1）功能处理方法支持的参数类型**

在继续学习之前，需要首先看看功能处理方法支持哪些类型的形式参数，以及他们的具体含义：

**（1）ServletRequest/HttpServletRequest 和 ServletResponse/HttpServletResponse**

**public** String requestOrResponse (

ServletRequest servletRequest, HttpServletRequest httpServletRequest,

ServletResponse servletResponse, HttpServletResponse httpServletResponse

)

Spring Web MVC框架会自动帮助我们把相应的Servlet请求/响应（Servlet API）作为参数传递过来。

**（2）InputStream/OutputStream 和 Reader/Writer**

**public** **void** inputOrOutBody(InputStream requestBodyIn, OutputStream responseBodyOut)

**throws** IOException {

responseBodyOut.write("success".getBytes());

}

**requestBodyIn**：获取**请求的内容区字节流**，等价于**request.getInputStream()**;

**responseBodyOut**：获取**相应的内容区字节流**，等价于**response.getOutputStream()**。

**public** **void** readerOrWriteBody(Reader reader, Writer writer)

**throws** IOException {

writer.write("hello");

}

**reader：**获取**请求的内容区字符流**，等价于**request.getReader();**

**writer：**获取**相应的内容区字符流**，等价于**response.getWriter()。**

InputStream/OutputStream 和 Reader/Writer两组不能同时使用，只能使用其中的一组。

**（3）WebRequest/NativeWebRequest**

**WebRequest**是spring Web MVC提供的**统一请求访问接口**，不仅仅可以**访问请求相关数据**（如参数区数据、请求头数据，但访问不到Cookie区数据），还可以**访问会话和上下文中的数据**；**NativeWebRequest继承了WebRequest**，并提供访问本地Servlet API的方法。

**public** String webRequest(WebRequest webRequest, NativeWebRequest nativeWebRequest) {

System.out.println(webRequest.getParameter("test"));//①得到请求参数test的值

webRequest.setAttribute("name", "value", WebRequest.SCOPE\_REQUEST);//②

System.out.println(webRequest.getAttribute("name", WebRequest.SCOPE\_REQUEST));

HttpServletRequest request =

nativeWebRequest.getNativeRequest(HttpServletRequest.**class**);//③

HttpServletResponse response =

nativeWebRequest.getNativeResponse(HttpServletResponse.**class**);

**return** "success";

}

* **webRequest.getParameter**：访问请求参数区的数据，可以通过**getHeader()**访问请求头数据；
* **webRequest.setAttribute/getAttribute**：到指定的作用范围内取/放属性数据，Servlet定义的三个作用范围分别使用如下常量代表：

**SCOPE\_REQUEST** ：代表请求作用范围；

**SCOPE\_SESSION** ：代表会话作用范围；

**SCOPE\_GLOBAL\_SESSION** ：代表**全局会话作用范围**，即ServletContext上下文作用范围。

* **nativeWebRequest.getNativeRequest/nativeWebRequest.getNativeResponse**：得到本地的Servlet API。

**（4）HttpSession**

**public** String session(HttpSession session) {

System.out.println(session);

**return** "success";

}

此处的session永远不为null。

**注意：**session访问不是线程安全的，如果需要线程安全，需要设置**AnnotationMethodHandlerAdapter**或**RequestMappingHandlerAdapter**的synchronizeOnSession属性为true，即可**线程安全**的**访问session**。

**（5）命令/表单对象**

Spring Web MVC能够**自动将请求参数绑定到功能处理方法的命令/表单对象**上。

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

**public** String toCreateUser(HttpServletRequest request, UserModel user) {

**return** "customer/create";

}

@RequestMapping(value = "/commandObject", method = RequestMethod.POST)

**public** String createUser(HttpServletRequest request, UserModel user) {

System.out.println(user);

**return** "success";

}

如果提交的表单（包含username和password文本域），将自动将**请求参数**绑定到命令对象user中去。

**（6）Model、Map、ModelMap**

Spring Web MVC 提供Model、Map或ModelMap让我们能去**暴露渲染视图需要的模型数据**。

@RequestMapping(value = "/model")

**public** String createUser(Model model, Map model2, ModelMap model3) {

model.addAttribute("a", "a");

model2.put("b", "b");

model3.put("c", "c");

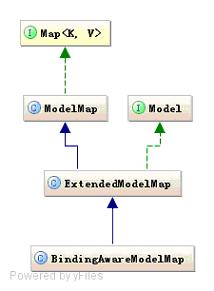
System.out.println(model == model2);

System.out.println(model2 == model3);

**return** "success";

}

虽然此处**注入的是三个不同的类型**（Model model, Map model2, ModelMap model3），但三者是**同一个对象**，如图所示：



**AnnotationMethodHandlerAdapter**和**RequestMappingHandlerAdapter**将使用**BindingAwareModelMap作为模型对象的实现**，即此处我们的形参（Model model, Map model2, ModelMap model3）**都是同一个BindingAwareModelMap实例**。

此处还有一点需要我们**注意：**

@RequestMapping(value = "/mergeModel")

**public** ModelAndView mergeModel(Model model) {

model.addAttribute("a", "a");//①添加模型数据

ModelAndView mv = **new** ModelAndView("success");

mv.addObject("a", "update");//②在视图渲染之前更新③处同名模型数据

model.addAttribute("a", "new");//③修改①处同名模型数据

//视图页面的a将显示为"update" 而不是"new"

**return** mv;

}

从代码中我们可以总结出功能处理方法的**返回值中的模型数据**（如ModelAndView）会 **合并** 功能处理方法**形式参数中的模型数据**（如Model），但如果两者之间有同名的，返回值中的模型数据会覆盖形式参数中的模型数据。

**（7）Errors/BindingResult**

@RequestMapping(value = "/error1")

**public** String error1(UserModel user, BindingResult result)

@RequestMapping(value = "/error2")

**public** String error2(UserModel user, BindingResult result, Model model)

@RequestMapping(value = "/error3")

**public** String error3(UserModel user, Errors errors)

以上代码都能获取错误对象。Spring3.1之前（使用AnnotationMethodHandlerAdapter）**错误对象**必须**紧跟在命令对象/表单对象之后**，如下定义是错误的：

@RequestMapping(value = "/error4")

**public** String error4(UserModel user, Model model, Errors errors)

如上代码从Spring3.1开始（使用**RequestMappingHandlerAdapter**）将能正常工作，但还是推荐“错误对象紧跟在命令对象/表单对象之后”，这样是万无一失的。

**（8）其他杂项**

**public** String other(Locale locale, Principal principal)

* **java.util.Locale：**得到**当前请求的本地化信息**，默认等价于ServletRequest.getLocale()，如果配置LocaleResolver解析器则由它决定Locale，后续介绍；
* **Java.security.Principal**：该主体对象**包含了验证通过的用户信息**，等价于HttpServletRequest.getUserPrincipal()。

**2）注解式控制器**

**（1）@RequestParam绑定单个请求参数值**

@RequestParam用于将请求参数区数据映射到功能处理方法的参数上。

**public** String requestparam1(@RequestParam(“param-name”) String username)

请求中**包含username参数**（如/requestparam1?username=zhang），则自动传入

@RequestParam注解主要有**哪些参数**：

* + **value**：参数名字，即入参的**请求参数名字**，如username表示请求的参数区中的名字为username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404错误码；
  + defaultValue：默认值，表示**如果请求中没有同名参数时的默认值**，默认值可以是SpEL表达式，如“#{systemProperties['java.vm.version']}”。

**public** String requestparam4(@RequestParam(value="username",required=**false**) String username)

表示请求中可以没有名字为username的参数，如果没有默认为null，此处需要**注意**如下几点：

* **原子类型**：必须有值，否则抛出异常，如果允许空值请使用包装类代替。
* **Boolean包装类型类型**：默认Boolean.FALSE，其他引用类型默认为null。

**public** String requestparam5(

@RequestParam(value="username", required=**true**, defaultValue="zhang") String username)

表示如果请求中没有名字为username的参数，默认值为“zhang”。

如果请求中有多个同名的应该如何接收呢？如给用户授权时，可能授予多个权限，首先看下如下代码：

**public** String requestparam7(@RequestParam(value="role") String roleList)

如果请求参数类似于url?role=admin&role=user，则实际roleList参数入参的数据为“admin,user”，即多个数据之间使用“，”分割；我们应该使用如下方式来接收多个请求参数：

**public** String requestparam7(@RequestParam(value="role") String[] roleList)

或

**public** String requestparam8(@RequestParam(value="list") List<String> list)

**（2）@PathVariable绑定URI模板变量值**

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

@RequestMapping(value="/users/{userId}/topics/{topicId}")

**public** String test(

@PathVariable(value="userId") **int** userId,

@PathVariable(value="topicId") **int** topicId)

如请求的URL为“控制器URL/users/123/topics/456”，则**自动将URL中模板变量**{userId}和{topicId}绑定到通过@PathVariable注解的同名参数上，即入参后userId=123、topicId=456。代码在PathVariableTypeController中。

**（3）@CookieValue绑定Cookie数据值**

@CookieValue用于**将请求的Cookie数据映射到功能处理方法的参数上。**

**public** String test(@CookieValue(value="JSESSIONID", defaultValue="") String sessionId)

如上配置将自动将**JSESSIONID值入参到sessionId参数**上，defaultValue表示Cookie中没有JSESSIONID时默认为空。

**public** String test2(@CookieValue(value="JSESSIONID", defaultValue="") Cookie sessionId)

传入参数类型也可以是javax.servlet.http.Cookie类型。

**（4）@RequestHeader绑定请求头数据**

@RequestHeader用于**将请求的头信息区数据映射到功能处理方法的参数上**。

@RequestMapping(value="/header")

**public** String test(

@RequestHeader("User-Agent") String userAgent,

@RequestHeader(value="Accept") String[] accepts)

如上配置将自动将请求头“User-Agent”值入参到userAgent参数上，并将“Accept”请求头值入参到accepts参数上。测试代码在HeaderValueTypeController中。

**（5）@ModelAttribute绑定请求参数到命令对象**

@ModelAttribute一个具有如下三个作用：

* **绑定请求参数到命令对象：**放在功能处理方法的入参上时，用于**将多个请求参数绑定到一个命令对象**，从而简化绑定流程，而且自动暴露为模型数据用于视图页面展示时使用；
* **暴露表单引用对象为模型数据：**放在处理器的**一般方法**（非功能处理方法）上时，是**为表单准备要展示的表单引用对象**，如注册时需要选择的所在城市等，而且在执行功能处理方法（@RequestMapping注解的方法）之前，自动添加到模型对象中，用于视图页面展示时使用；
* **暴露@RequestMapping方法返回值为模型数据：**放在功能处理方法的返回值上时，是暴露功能处理方法的返回值为模型数据，用于视图页面展示时使用。

**a.绑定请求参数到命令对象**

如用户登录，我们需要捕获用户登录的**请求参数（用户名、密码）**并**封装为用户对象**，此时我们可以**使用@ModelAttribute绑定多个请求参数到我们的命令对象**。

**public** String test1(@ModelAttribute("user") UserModel user)

和命令/表单对象功能一样。只是此处多了一个注解@ModelAttribute("user")，它的**作用**是**将该绑定的命令对象以“user”为名称添加到模型对象中供视图页面展示使用**。我们此时可以在**视图页面**使用**${user.username}来获取绑定的命令对象的属性**。

绑定请求参数到命令对象支持对象图导航式的绑定，如请求参数包含“?username=zhang&password=123&workInfo.city=bj”自动绑定到user中的workInfo属性的city属性中。

@RequestMapping(value="/model2/{username}")

**public** String test2(@ModelAttribute("model") DataBinderTestModel model)

DataBinderTestModel相关模型请从第三章拷贝过来，请求参数到命令对象的绑定规则详见【4.16.1、数据绑定】一节，**URI模板变量也能自动绑定到命令对象**中，当你请求的URL中包含“bool=yes&schooInfo.specialty=computer&hobbyList[0]=program&hobbyList[1]=music&map[key1]=value1&map[key2]=value2&state=blocked”会自动绑定到命令对象上。

**当URI模板变量和请求参数同名时，URI模板变量具有高优先权**。

**b.暴露表单引用对象为模型数据**

@ModelAttribute("cityList")

**public** List<String> cityList() {

**return** Arrays.asList("北京", "山东");

}

如上代码会在**执行功能处理方法之前执行**，并将其**自动添加到模型对象**中，在功能处理方法中调用Model 入参的containsAttribute("cityList")将会返回true。

@ModelAttribute("user") //①

**public** UserModel getUser(@RequestParam(value="username", defaultValue="") String username) {

//**TODO** 去数据库根据用户名查找用户对象

UserModel user = **new** UserModel();

user.setRealname("zhang");

**return** user;

}

如你要修改用户资料时一般需要根据用户的编号/用户名查找用户来进行编辑，此时可以通过如上代码查找要编辑的用户。也可以进行一些默认值的处理。

@RequestMapping(value="/model1") //②

**public** String test1(@ModelAttribute("user") UserModel user, Model model)

此处我们看到①和②有同名的命令对象，那Spring Web MVC内部如何处理的呢：

* 首先执行@ModelAttribute注解的方法，**准备视图展示时所需要的模型数据**；@ModelAttribute注解方法形式参数规则和@RequestMapping规则一样，如可以有@RequestParam等；
* 执行@RequestMapping注解方法，进行模型绑定时首先查找模型数据中是否含有同名对象，如果有直接使用，如果没有通过反射创建一个，因此②处的user将使用①处返回的命令对象。即②处的user等于①处的user。

**c.暴露@RequestMapping方法返回值为模型数据**

**public** @ModelAttribute("user2") UserModel test3(@ModelAttribute("user2") UserModel user)

大家可以看到**返回值类型**是**命令对象类型**，而且通过@ModelAttribute("user2")注解，此时会暴露返回值到模型数据（名字为user2）中供视图展示使用。那哪个视图应该展示呢？此时Spring Web MVC会根据RequestToViewNameTranslator进行**逻辑视图名的翻译**，详见【4.15.5、RequestToViewNameTranslator】一节。此时又有问题了，@RequestMapping注解方法的入参user暴露到模型数据中的名字也是user2，其实我们能猜到：@ModelAttribute注解的**返回值会覆盖**@RequestMapping注解方法中的@ModelAttribute注解的**同名命令对象**。

**d.** **匿名绑定命令参数**

**public** String test4(@ModelAttribute UserModel user, Model model)

或

**public** String test5(UserModel user, Model model)

此时我们没有为命令对象提供暴露到模型数据中的名字，此时的名字是什么呢？Spring Web MVC自动将简单类名（首字母小写）作为名字暴露，如“cn.javass.chapter6.model.UserModel”暴露的名字为“userModel”。

**public** @ModelAttribute List<String> test6()

或

**public** @ModelAttribute List<UserModel> test7()

对于**集合类型**（Collection接口的实现者们，包括数组），生成的**模型对象属性名**为“简单类名（首字母小写）”+“List”，如List<String>生成的模型对象属性名为“stringList”，List<UserModel>生成的模型对象属性名为“userModelList”。

其他情况一律都是使用简单类名（首字母小写）作为模型对象属性名，如Map<String, UserModel>类型的模型对象属性名为“map”。

**（6）@SessionAttributes绑定命令对象到session**

有时候需要在**多次请求之间保持数据**，一般情况需要我们明确的调用**HttpSession**的API来存取会话数据，如多步骤提交的表单。Spring Web MVC提供了**@SessionAttributes**进行**请求间透明的存取会话数据**。

//1、在控制器类头上添加@SessionAttributes注解

@SessionAttributes(value = {"user"}) //①

**public** **class** SessionAttributeController

//2、@ModelAttribute注解的方法进行表单引用对象的创建

@ModelAttribute("user") //②

**public** UserModel initUser()

//3、@RequestMapping注解方法的@ModelAttribute注解的参数进行命令对象的绑定

@RequestMapping("/session1") //③

**public** String session1(@ModelAttribute("user") UserModel user)

//4、通过SessionStatus的setComplete()方法清除@SessionAttributes指定的会话数据

@RequestMapping("/session2") //③

**public** String session(@ModelAttribute("user") UserModel user, SessionStatus status) {

**if**(**true**) { //④

status.setComplete();

}

**return** "success";

}

**@SessionAttributes(value = {"user"})含义**：@SessionAttributes(value = {"user"}) 标识将模型数据中的名字为“user” 的对象存储到会话中（默认HttpSession），此处value指定将模型数据中的哪些数据（名字进行匹配）存储到会话中，此外还有一个types属性表示模型数据中的哪些类型的对象存储到会话范围内，如果同时指定**value和types属性**则那些**名字和类型都匹配的对象**才能**存储到会话范围**内。

包含@SessionAttributes的**执行流程如下所示**：

* ①首先根据@SessionAttributes注解信息**查找**会话内的**对象**放入到**模型数据**中；
* ②执行@ModelAttribute注解的方法：如果模型数据中包含同名的数据，则不执行@ModelAttribute注解方法进行准备表单引用数据，而是使用①步骤中的会话数据；如果模型数据中不包含同名的数据，执行@ModelAttribute注解的方法并将返回值添加到模型数据中；
* ③执行@RequestMapping方法，绑定@ModelAttribute注解的参数：查找模型数据中是否有@ModelAttribute注解的同名对象，如果有直接使用，否则通过反射创建一个；并将请求参数绑定到该命令对象；

**此处需要注意**：如果使用@SessionAttributes注解控制器类之后，③步骤一定是从模型对象中取得同名的命令对象，如果模型数据中不存在将抛出HttpSessionRequiredException Expected session attribute ‘user’(Spring3.1)或HttpSessionRequiredException Session attribute ‘user’ required - not found in session(Spring3.0)异常。

* 如果会话可以销毁了，如多步骤提交表单的最后一步，此时可以调用**SessionStatus对象**的**setComplete()标识当前会话的@SessionAttributes指定的数据可以清理**了，此时当@RequestMapping功能处理方法执行完毕会进行清理会话数据。

通过Spring Web MVC的源代码验证一下吧，此处我们分析的是Spring3.1的**RequestMappingHandlerAdapter**，读者可以自行验证Spring3.0的AnnotationMethodHandlerAdapter，流程一样：

**a.** **RequestMappingHandlerAdapter.invokeHandlerMethod**

//1、RequestMappingHandlerAdapter首先调用ModelFactory的initModel方法**准备模型数据**：

modelFactory.initModel(webRequest, mavContainer, requestMappingMethod);

//2、调用@RequestMapping注解的**功能处理方法**

requestMappingMethod.invokeAndHandle(webRequest, mavContainer);

//3、**更新/合并**模型数据

modelFactory.updateModel(webRequest, mavContainer);

**b.** **ModelFactory.initModel**

Map<String, ?> attributesInSession = **this**.sessionAttributesHandler.retrieveAttributes(request);

//1.1、将与@SessionAttributes注解相关的会话对象放入模型数据中

mavContainer.mergeAttributes(attributesInSession);

//1.2、调用@ModelAttribute方法添加表单引用对象

invokeModelAttributeMethods(request, mavContainer);

//1.3、验证模型数据中是否包含@SessionAttributes注解相关的会话对象，不包含抛出异常

**for** (String name : findSessionAttributeArguments(handlerMethod)) {

**if** (!mavContainer.containsAttribute(name)) {

//1.4、此处防止在@ModelAttribute注解方法又添加了会话对象

//如在@ModelAttribute注解方法调用session.setAttribute("user", new UserModel());

Object value = **this**.sessionAttributesHandler.retrieveAttribute(request, name);

**if** (value == **null**) {

**throw** **new** HttpSessionRequiredException("Expected session attribute '" + name + "'");

}

mavContainer.addAttribute(name, value);

}

**c.** **ModelFactory.invokeModelAttributeMethods**

**for** (InvocableHandlerMethod attrMethod : **this**.attributeMethods) {

String modelName = attrMethod.getMethodAnnotation(ModelAttribute.**class**).value();

//1.2.1、如果模型数据中包含同名数据则不再添加

**if** (mavContainer.containsAttribute(modelName)) {

**continue**;

}

//1.2.2、调用@ModelAttribute注解方法并将返回值添加到模型数据中，此处省略实现代码

}

**d.** **requestMappingMethod.invokeAndHandle**

调用功能处理方法，此处省略

**e.** **ModelFactory.updateMode 更新模型数据**

//3.1、如果会话被标识为完成，此时从会话中清除@SessionAttributes注解相关的会话对象

**if** (mavContainer.getSessionStatus().isComplete()){

**this**.sessionAttributesHandler.cleanupAttributes(request);

}

//3.2、如果会话没有完成，将模型数据中的@SessionAttributes注解相关的对象添加到会话中

**else** {

**this**.sessionAttributesHandler.storeAttributes(request, mavContainer.getModel());

}

//省略部分代码

**（7）@Value绑定SpEL表示式**

@Value用于**将一个SpEL表达式结果映射到到功能处理方法的参数**上。

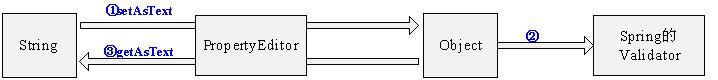
**public** String test(@Value("#{systemProperties['java.vm.version']}") String jvmVersion)

**3）数据类型转换**

**（1）简介**

编写可视化界面项目时，通常需要对数据进行类型转换、验证及格式化。

**a.** **Spring3之前，使用如下架构进行类型转换、验证及格式化**



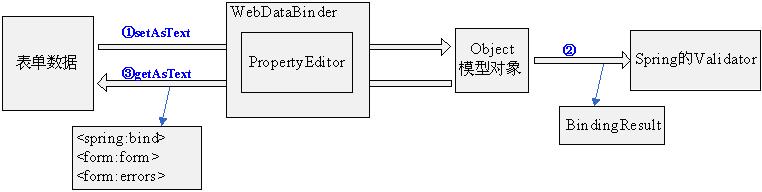
**流程：**

* **①类型转换**：首先调用PropertyEditor的**setAsText**（String），内部根据需要调用**setValue**(Object)方法进行设置转换后的值；
* **②数据验证**：需要显示调用spring的Validator接口实现进行数据验证；
* **③格式化显示**：需要调用PropertyEditor的getText进行格式化显示。

**缺点：**

* PropertyEditor被设计为只能String<——>Object之间转换，不能任意对象类型<——>任意类型，如我们常见的Long时间戳到Date类型的转换是办不到的；
* PropertyEditor是**线程不安全**的，也就是**有状态**的，因此每次使用时都需要创建一个，不可重用；
* PropertyEditor不是强类型的，setValue（Object）可以接受任意类型，因此需要我们自己判断类型是否兼容；
* 需要自己编程实现验证，Spring3支持更棒的**注解验证支持**；
* 在使用SpEL表达式语言或DataBinder时，只能进行String<--->Object之间的类型转换；
* 不支持细粒度的类型转换/格式化，如UserModel的registerDate需要转换/格式化类似“2012-05-01”的数据，而OrderModel的orderDate需要转换/格式化类似“2012-05-01 15：11：13”的数据，因为大家都为Java.util.Date类型，因此不太容易进行细粒度转换/格式化。

**Spring Web MVC环境中，数据类型转换、验证及格式化通常是这样使用的：**



**流程：**

**①类型转换**：首先表单数据（全部是字符串）通过**WebDataBinder进行绑定到命令对象**，内部通过PropertyEditor实现；

**②数据验证**：在控制器中的功能处理方法中，需要显示的调用Spring的Validator实现并将错误信息添加到BindingResult对象中；

**③格式化显示**：在表单页面可以通过如下方式展示通过PropertyEditor格式化的数据和错误信息：

<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>

<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

首先需要通过如上**taglib指令引入spring的两个标签库**。

//1、格式化单个命令/表单对象的值（好像比较麻烦，真心没有好办法）

<spring:bind path=*"dataBinderTest.phoneNumber"*>${status.value}</spring:bind>

//2、通过form标签，**内部的表单标签**会自动调用命令/表单对象属性对应的**PropertyEditor**进行**格式化**显示

<form:form commandName=*"dataBinderTest"*>

<form:input path=*"phoneNumber"*/><!-- 如果出错会显示错误之前的数据而不是空 -->

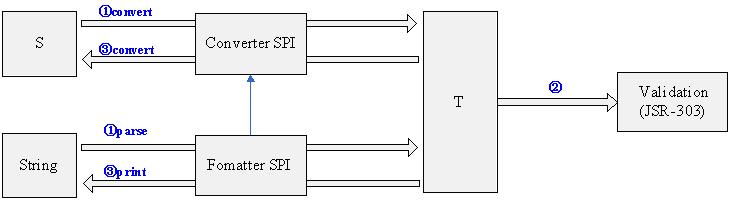
</form:form>

//3、显示验证失败后的错误信息

<form:errors></form:errors>

如上PropertyEditor和验证API使用起来比较麻烦，而且有许多缺点，因此Spring3提供了更强大的**类型转换**（Type Conversion）支持，它可以**在任意对象之间进行类型转换**，不仅仅是String<——>Object；也提供了强大的数据验证支持；同时提供了强大的数据格式化支持。

**b.** **从Spring3开始，可使用如下架构进行类型转换、验证及格式化：**



**流程：**

* **①类型转换：**内部的**ConversionService**会根据S源类型/T目标类型自动选择相应的Converter SPI进行**类型转换**，而且是强类型的，能在任意类型数据之间进行转换；
* **②数据验证：**支持JSR-303验证框架，如将**@Valid**放在需要验证的目标类型上即可；
* **③格式化显示：**其实就是任意目标类型---->String的转换，完全可以使用Converter SPI完成。

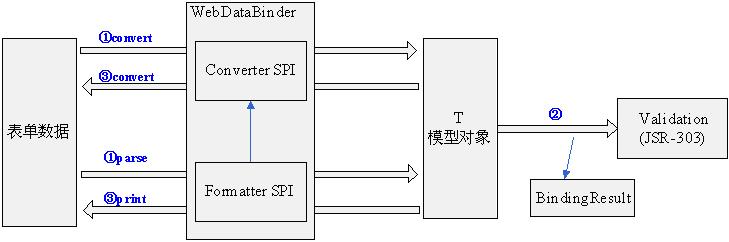
Spring为了更好的诠释**格式化/解析功能**提供了Formatter SPI，支持根据**Locale信息**进行格式化/解析，而且该套SPI可以**支持字段/参数级别的细粒度格式化/解析**，流程如下：

* **①类型解析（转换）：**String---->T类型目标对象的解析，和PropertyEditor类似；
* **③格式化显示：**任意目标类型---->String的转换，和PropertyEditor类似。

Formatter SPI最大特点是能进行字段/参数级别的细粒度解析/格式化控制，即使是Converter SPI也是粗粒度的（到某个具体类型，而不是其中的某个字段单独控制），目前Formatter SPI还不是很完善，如果您有好的想法可以到Spring官网提建议。

Formatter SPI内部实现实际委托给Converter SPI进行转换，即约束为解析/格式化String<---->任意目标类型。

**Spring Web MVC环境中，数据类型转换、验证及格式化通常是这样使用的：**



* **①类型转换：**首先表单数据（全部是字符串）通过WebDataBinder进行绑定到命令对象，内部通过Converter SPI实现；
* **②数据验证：**使用JSR-303验证框架进行验证；
* **③格式化显示：**在表单页面可以通过如下方式展示通过内部通过Converter SPI格式化的数据和错误信息：

<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>

<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

首先需要通过如上**taglib指令引入spring的两个标签库**。

//1、格式化单个命令/表单对象的值（好像比较麻烦，真心没有好办法）

<spring:bind path=*"dataBinderTest.phoneNumber"*>${status.value}</spring:bind>

//2、<spring:eval>标签，自动调用**ConversionService**并选择相应的Converter SPI进行**格式化展示**

<spring:eval expression=*"dataBinderTest.phoneNumber"*></spring:eval>

如上代码能工作的前提是在**RequestMappingHandlerMapping**配置了**ConversionServiceExposingInterceptor**，它的作用是暴露conversionService到请求中以便如<spring:eval>标签使用。

//3、通过form标签，**内部的表单标签**会自动调用命令/表单对象属性对应的**PropertyEditor**进行**格式化**显示

<form:form commandName=*"dataBinderTest"*>

<form:input path=*"phoneNumber"*/><!-- 如果出错会显示错误之前的数据而不是空 -->

</form:form>

//4、显示验证失败后的错误信息

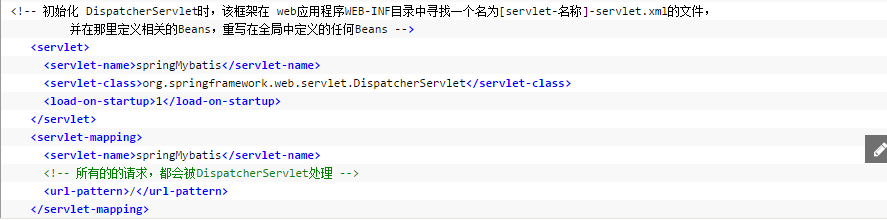
<form:errors></form:errors>

**5、Spring MVC 拦截器（资源和权限管理）**

**1）DispatcherServlet**

SpringMVC具有**统一的入口**DispatcherServlet，所有的请求都通过DispatcherServlet。

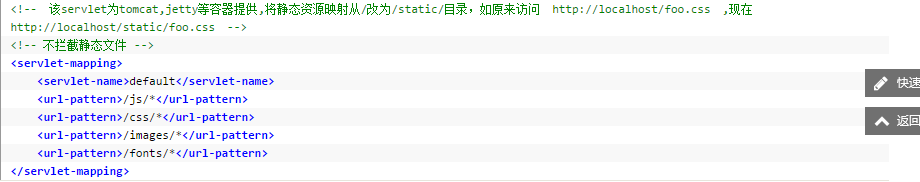
DispatcherServlet是前置控制器，配置在web.xml文件中的。拦截匹配的请求，Servlet拦截匹配规则要自已定义，把拦截下来的请求，依据某某规则分发到目标Controller来处理。 所以我们现在web.xml中加入以下配置：



**2）静态资源不拦截**

如果只配置拦截类似于\*.do格式的url，则对**静态资源的访问**是没有问题的，但是如果配置拦截了所有的请求（如我们上面配置的“/”），就会**造成**js文件、css文件、图片文件等**静态资源无法访问**。

一般实现拦截器主要是为了**权限管理**，主要是**拦截一些url请求**，所以**不对静态资源进行拦截**。要过滤掉静态资源一般有两种方式，第一种是采用**<mvc:default-servlet-handler />**（一般Web应用服务器默认的Servlet名称是"**default**"，所以这里我们激活Tomcat的defaultServlet来处理静态文件，在web.xml里配置如下代码即可）：



Tomcat, Jetty, JBoss, and GlassFish 默认 Servlet的名字 -- "default"

Resin 默认 Servlet的名字 -- "resin-file"

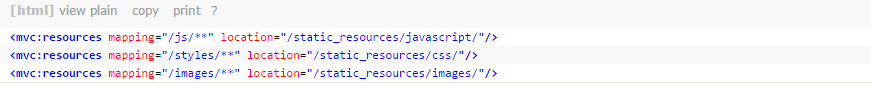
WebLogic 默认 Servlet的名字 -- "FileServlet"

WebSphere 默认 Servlet的名字 -- "SimpleFileServlet"

如果你所有的Web应用服务器的默认Servlet名称不是"default"，则需要通过**default-servlet-name**属性显示指定：



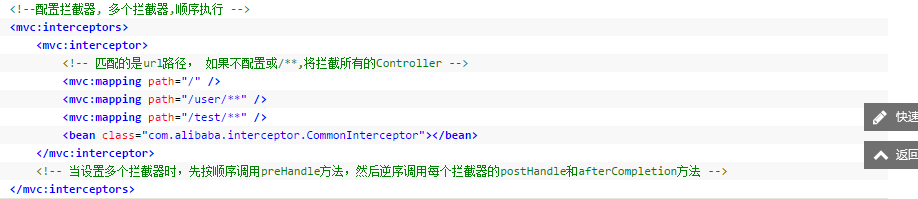
第二种是采用**<mvc:resources />**，在springmvc的配置文件中加入以下代码：



**3）自定义拦截器**

Spring MVC的拦截器**HandlerInterceptorAdapter**对应提供了三个**preHandle**，**postHandle**，**afterCompletion**方法。preHandle在**业务处理器处理请求之前**被调用，postHandle在业务处理器处理请求执行完成后,生成视图之前执行，afterCompletion在DispatcherServlet完全处理完请求后被调用,可用于**清理资源**等 。所以要想实现自己的权限管理逻辑，需要继承**HandlerInterceptorAdapter**并重写其三个方法。

首先在springmvc.xml中加入自己定义的拦截器实现逻辑CommonInterceptor，



**6、Spring MVC 缓存**

**步骤：**

1、在springmvc的**配置文件**中加入缓存配置，代码如下：



**（注意不要忘记引入对应的命名空间）**

2、在配置路径下（这里是默认的src下）建立ehcache.xml文件，并配置程序的相关cache策略，代码如下：

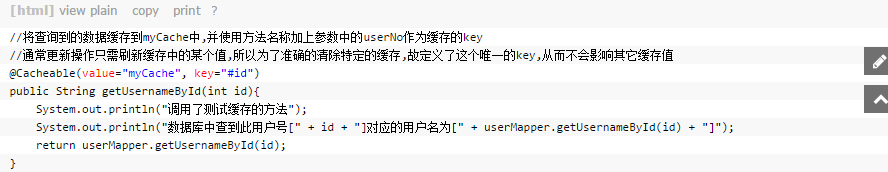


3、既然是ehcache，肯定要引入ehcache的jar：**ehcache-2.8.3**，至于还需要什么jar，运行后就会发现

4、运行后报错，nested exception is java.lang.NoClassDefFoundError:org/aopalliance/intercept

/MethodInterceptor是因为缺少aopaliance-1.0 jar包，加入即可。

5、**后台service代码**（注解是加在service的方法上的）：



**注意：springmvc有关缓存的注解主要是@Cacheable（执行缓存）、@CachePut、@CacheEvict（清除缓存）**