SpringMVC

尚硅谷 Java

版本 V1.0

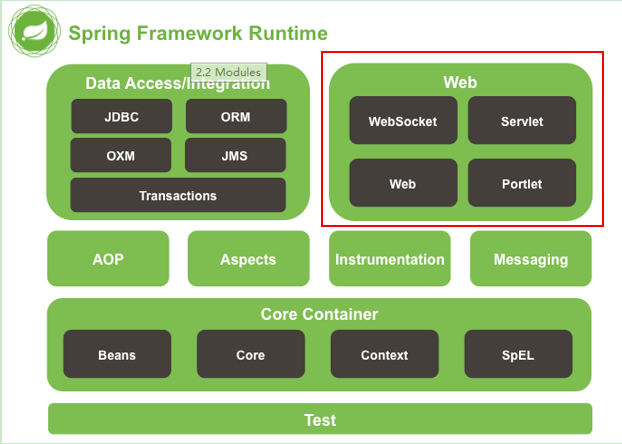
# 一、SpringMVC简介

## 1.1 SpringMVC概念

Spring: 是一个容器框架，用来简化开发！

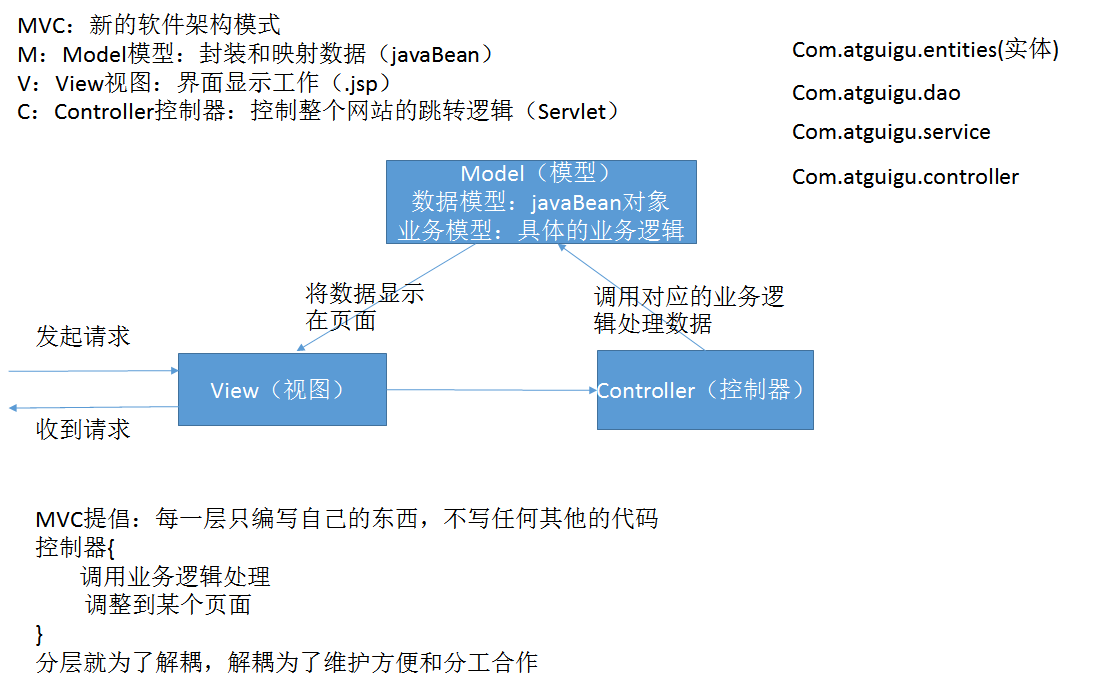
MVC： 是一种web开发的设计思想，分层解耦；

SpringMVC： Spring提供的符合MVC设计思想的，简化web开发的模块。



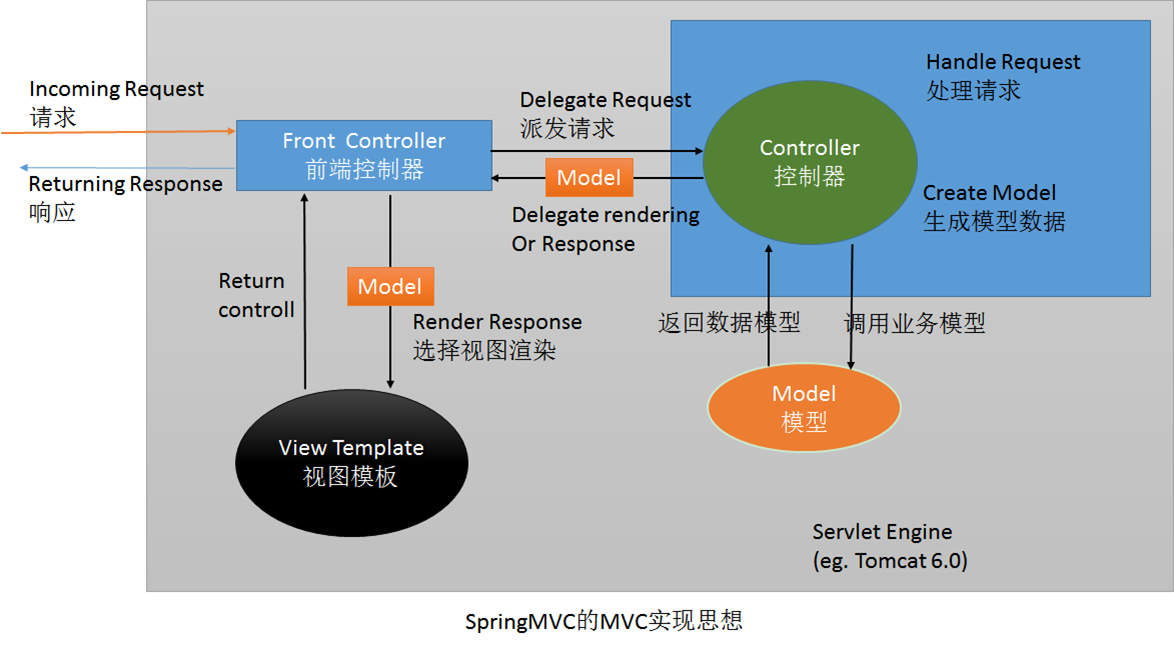
## 1.2 SpringMVC的核心

传统的MVC：



SpringMVC遵循了MVC的开发模式，但是做了一些改进：

在传统的MVC的基础上，增加了一个前端控制器来只能派发所有的请求！

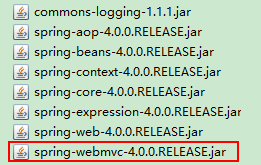


## 1.3 SpringMVC优点

* **Spring MVC 通过一套 MVC 注解**，让 POJO 成为处理请求的控制器，而无须实现任何接口。
* **支持 REST 风格的 URL 请求**
* 采用了松散耦合可插拔组件结构，比其他 MVC 框架更具扩展性和灵活性

# 二、Helloworld

## 1. 建工程，导jar包

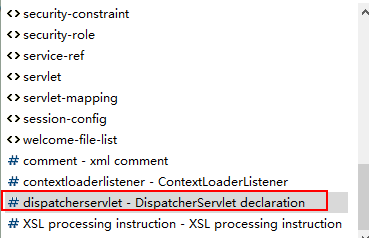


## 2. 配置

### 2.1配置前端控制器

SpringMVC的核心是前端控制器，所有的请求都需要用前端控制器来做智能派发。前端控制器实际上是一个Servlet！

使用alt+/，选择DispatcherServlet代码片段！



<servlet>

<servlet-name>springDispatcherServlet</servlet-name>

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

<init-param>

<!--配置文件的路径 -->

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

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

</init-param>

<!-- servlet被创建的时机： 请求第一次访问的时候创建

load-on-startup：配置servlet创建的策略，随着web应用的一创建就创建

后边配置一个Int型的参数，代表创建的优先级，数字越小，优先级越高！

-->

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

</servlet>

<!-- 处理哪些请求 -->

<servlet-mapping>

<servlet-name>springDispatcherServlet</servlet-name>

<!-- 配置拦截所有请求：

注意：只能写/

/\*:不能写

-->

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

</servlet-mapping>

### 2.2 配置Spring的配置文件



## 3.代码编写

### 1 页面、成功页面

### 2 处理器

@Controller

**public** **class** HelloHandler {

/\*

\* 1. 路径分为相对路径和绝对路径（/）

\* 浏览器端解析： html http://ip:端口号

\* 服务器端解析： http://ip:端口号：项目名

\*/

@RequestMapping("/hello")

**public** String hello(){

System.***out***.println("进入了hello请求！");

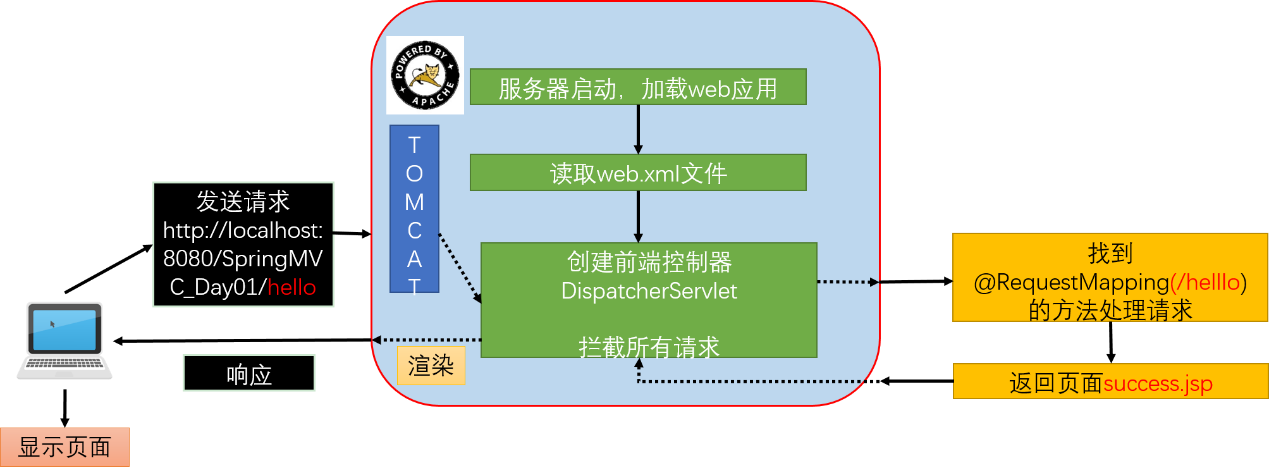
//request.getRequestDispatcher("xx").forward(request.response);

**return** "/WEB-INF/pages/success.jsp";

}

}

### 3.流程



## 4. 细节

### 4.1 默认的配置文件

DispatcherServlet的<init-param>参数可以忽略！如果忽略，容器会自动加载WEF-INF目录下，名称为 【xxx】-servlet.xml的配置文件！

### 4.2 <url-pattern>

只能写/。如果写/\*，会拦截所有的JSP页面！

原因：

每个工程的web.xml实际上是继承了Tomcat服务器的web.xml。

在Tomcat服务器的web.xml配置了一个JspServlet的Servlet，这个Servlet负责编译和处理所有的JSP页面；它默认拦截 /\*.jsp 和 /\*.jspx

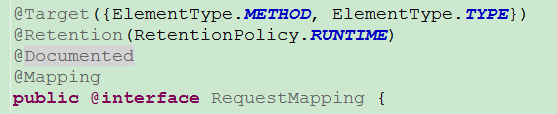
遗留问题：静态资源无法访问！

静态资源：除去Servlet、Jsp,其他的都是！

原因：默认Tomcat服务器的web.xml配置了DefaultServlet，是默认拦截/请求，我们现在的DispatcherServlet覆盖了之前的配置，所以会在DispatcherServlet，里面寻找处理静态资源的方法！我们并没有编写，因此所有静态资源的请求都会报错404！

# 三、@RequestMapping

## 1. 概述



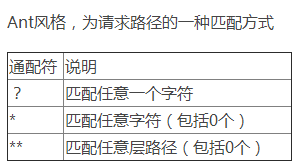
@RequestMapping注解负责处理请求，可以标注在类和方法上！

①标注在方法上： 可以处理哪些请求

②标注在类上：相当于为这个类的所有方法的@RequestMapping属性，添加了一个基准路径（父路径）

通常不同的模块，可以使用不同的基准路径！

## 2. Ant风格的路径映射



同时符合多个条件，精确优先！

## 3. 属性

### 3.1 value

value: String[ ] ,可以映射的请求，多个请求用,隔开

### 3.2 method

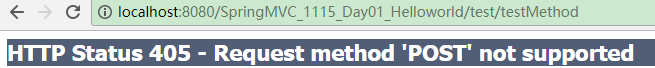
method: 限制这个请求的请求方式

请求方式：【GET】, HEAD, 【POST】, PUT, PATCH, DELETE, OPTIONS, TRACE

如果请求方式不匹配，报错405；

4xx: 客户的错误

5xx: 服务端错误



### 3.3 params

params： 限定请求携带的参数。支持！、=、！=等逻辑判断。

示例：

age:必须携带age参数

！user：不能携带user参数

gender=0:必须携带gender，且gender=0;

gender!=0:必须携带gender，且gender不等于0

params={"age","!user","gender!=0"}：以上三个条件必须同时满足

即：必须有age参数，不能有user参数，gender!=0

### 3.4 headers

headers:限定请求携带的请求头信息。示例：headers={"Host=localhost:8080"}:只有Host是localhost:8080的请求才能访问！否则：404

**3.5 consumes**

只接受内容类型是哪种的请求，规定请求头中的Content-Type

**3.6 produces**

告诉浏览器返回的内容类型是什么，给响应头中加上Content-Type:text/html;charset=utf-8

# 四、REST

## 0. @PathVariable

①@PathVariable:获取指定请求路径上的变量值！

②如果路径无法匹配，就报错！

@RequestMapping("/handle/{id}")

**public** String handle(@PathVariable("id")Integer id){

System.***out***.println("路径上的id:"+id);

**return** "success";

}

## 1. REST概念

REST：即 Representational State Transfer。**（资源）表现层状态转化。是目前最流行的一种互联网软件架构**。

* **资源（Resources）**：网络上的一个实体，或者说是网络上的一个具体信息。

它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的存在。

可以用一个URI（统一资源定位符）指向它，每种资源对应一个特定的 URI 。

获取这个资源，访问它的URI就可以，因此 URI 即为每一个资源的独一无二的识别符。

* **表现层（Representation）**：把资源具体呈现出来的形式，叫做它的表现层（Representation）。比如，文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式。
* **状态转化（State Transfer）**：每发出一个请求，就代表了客户端和服务器的一次交互过程。HTTP协议，是一个无状态协议，即所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生“状态转化”（State Transfer）。

而这种转化是建立在表现层之上的，所以就是 “表现层状态转化”。

* **具体说，就是 HTTP 协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。**

**它们分别对应四种基本操作：GET 用来获取资源，POST 用来新建资源，PUT 用来更新资源，DELETE 用来删除资源。**

总结：

①REST是一种思想。REST推崇简洁的URL表达！

②REST这种思想，认为万物皆资源，所有的请求都是为了获取资源，通过HTTP协议中请求方式的不同，表达对资源的不同状态的修改！即GET对应查询、PUT对应修改、DELETE对应删除、POST对应添加。

REST推荐的URL格式：

/操作的资源类型/资源标识符 以请求方式的不同，区别对资源的不同操作（CRUD）

|  |  |  |
| --- | --- | --- |
| 以对Book的增删改查为例 | 过去URL | 使用REST的URL |
| 增 | /addBook? bookName=xx | /book（发送POST请求） |
| 删 | /deleteBook?bookId=1 | /book/1 (发送DELETE请求) |
| 改 | /updateBook?bookId=1&bookName=xx | /book/1(发送PUT请求) |
| 查 | /getBook?bookId=1 | /book/1(发送GET请求) |

## 2. 简单的REST示例

问题：GET和POST请求很常见，而PUT和DELETE请求如何发？

解决：

①配置一个请求方式过滤器HiddenHttpMethodFilter，这个过滤器可以将POST请求转换为相应的PUT、DELETE

<filter>

<filter-name>HiddenHttpMethodFilter</filter-name> <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class></filter>

<filter-mapping>

<filter-name>HiddenHttpMethodFilter</filter-name>

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

</filter-mapping>

②创建一个method=post的表单

③表单中有名为\_method的属性，值为要发的请求方式（PUT,DELETE）不区分大小写

<a href=*"book/1"*>查询图书</a><br>

<form action=*"book"* method=*"post"*>

<input value=*"新增"* type=*"submit"*>

</form><br>

<form action=*"book/1"* method=*"post"*>

<input name=*"\_method"* value=*"delete"* type=*"hidden"*>

<input value=*"删除"* type=*"submit"*>

</form><br>

<form action=*"book/1"* method=*"post"*>

<input name=*"\_method"* value=*"put"* type=*"hidden"*>

<input value=*"修改"* type=*"submit"*>

</form><br>

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

**public** String getBook(@PathVariable("id")Integer id){

System.***out***.println("查询"+id+"号图书！");

**return** "success";

}

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

**public** String addBook(){

System.***out***.println("新增了一本图书！");

**return** "success";

}

@RequestMapping(value="/book/{id}",method=RequestMethod.***DELETE***)

**public** String deleteBook(@PathVariable("id")Integer id){

System.***out***.println("删除"+id+"号图书！");

**return** "success";

}

@RequestMapping(value="/book/{id}",method=RequestMethod.***PUT***)

**public** String updateBook(@PathVariable("id")Integer id){

System.***out***.println("修改"+id+"号图书！");

**return** "success";

}

## 3 HiddenHttpMethodFilter原理

HiddenHttpMethodFilter的doFilterInternal方法中，对请求方式进行了转换

@Override

**protected** **void** doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)

**throws** ServletException, IOException {

//从请求参数中获取名称为 \_method的属性，这个属性就是我们自己写的PUT，DELETE

String paramValue = request.getParameter(**this**.methodParam);

//判断，当前是POST请求，且有\_method属性

**if** ("POST".equals(request.getMethod()) && StringUtils.*hasLength*(paramValue)) {

//将\_method属性转换为大写

String method = paramValue.toUpperCase(Locale.***ENGLISH***);

//将原来的请求，再次包装。在HttpMethodRequestWrapper里面将请求设置为我们的PUT，DELETE，并重写了getMethod()

HttpServletRequest wrapper = **new** HttpMethodRequestWrapper(request, method);

//放行包装后的请求

filterChain.doFilter(wrapper, response);

}

**else** {

//GET请求，和POST请求，直接放行

filterChain.doFilter(request, response);

}

}

**private** **static** **class** HttpMethodRequestWrapper **extends** HttpServletRequestWrapper {

**private** **final** String method;

**public** HttpMethodRequestWrapper(HttpServletRequest request, String method) {

**super**(request);

**this**.method = method;

}

@Override

**public** String getMethod() {

**return** **this**.method;

}

}

# 五、处理请求

## 1. @RequestParam 获取请求参数

在处理方法入参处使用 @RequestParam 可以把请求参数传递给请求方法，功能类似request.getParameter("xx")。

**常见属性**：

value：参数名

required：是否必须。默认为 true, 表示请求参数中必须包含对应的参数，若不存在，将抛出异常

defaultValue:如果无法正确获取参数，就指定一个默认值。如果不指定，默认是null

**使用方法**：

① 默认情况下，只需要在处理方法的参数上，声明与请求参数名一致的形参即可。但是要注意方法参数名 与请求参数名要一致，否则无法自动赋值。

②如果遇到形参与请求参数不一致，可以使用@RequestParam("xx")明确指定要获取的参数名。

③一旦使用@RequestParam注解，如果无法找到指定的参数，就报错400。原因是默认@RequestParam(required=true)。可以通过修改@RequestParam(required=false)来解决。

## 2 .@RequestHeader

**作用**：从请求中获取请求头对应的信息。功能类似于request.getHeader("");

## 3. @CookieValue

之前获取Cookie的方式较为麻烦！

Cookie[] cookies=request.getCookies();

String value="";

for(Cookie c: cookies){

if(c.getName().equals("JESSIONID")){

value=c.getValue();

}

}

现在使用@CookieValue("xx")获取指定cookie的值

## 4. 封装POJO

当页面传入的参数，恰好对应一个POJO的属性时，可以将这个属性封装为POJO。

做法：如果我们要提交的参数正好对应POJO的属性，只需要在处理方法的形参位置，声明一个POJO，SpringMVC会自动地帮我们封装参数

规则：参数名与 POJO的属性(setter)一一对应

级联属性的封装： 属性.属性。例如：<input name="phone.brand">

## 5. 使用原生API

SpringMVC同时支持在处理方法的参数位置直接声明和使用原生的API。可用的原生API类型有HttpServletRequest、HttpServletResponse、HttpSession、java.security.Principal、Locale、InputStream、OutputStream

Reader、Writer。

实际上掌握前三种即可，后面的六大对象，可以通过前三个获取！

request.getLocale();

request.getUserPrincipal();

request.getInputStream();

request.getReader();

response.getWriter();

response.getOutputStream();

## 6.乱码问题的解决

乱码的分类：请求乱码和响应乱码。

请求乱码：

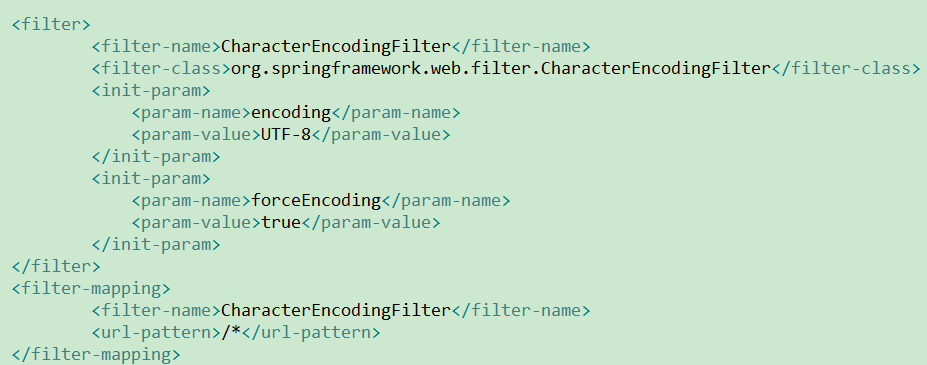
GET： 修改server.xml,在8080端口处添加属性 URIEncoding=UTF-8



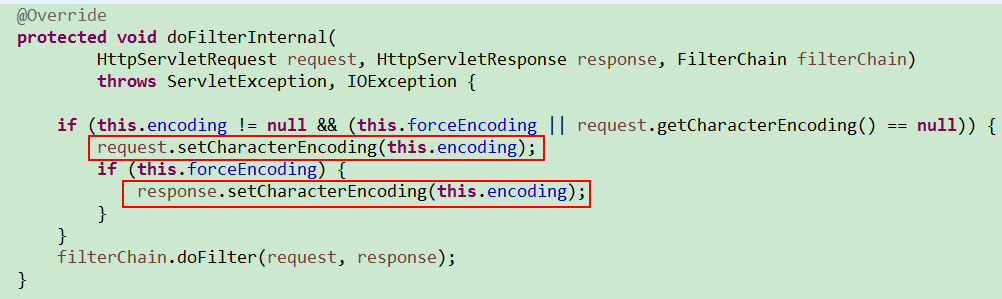
POST：（PUT，DELETE）也是POST。在第一次获取请求之前，设置request.setCharacterEncoding("UTF-8");

响应乱码：response.setContentType("text/html;charset=utf-8")

最终解决： SpringMVC提供了一个字符编码过滤器来解决；



原理：



我们在声明字符编码过滤器后，需要配置两个参数：encoding和forceEncoding。encoding代表设置请求和响应的编码。forceEncoding代表是否处理响应乱码。

注意：保证这个filter必须在其他filter之前配置在有效！

# 六、输出数据

## 1. web.xml标配

<DispatcherServlet> 前端控制器

< CharacterEncodingFilter> 字符编码过滤器，要配在首位

< HiddenHttpMethodFilter> 支持RESTFUL请求

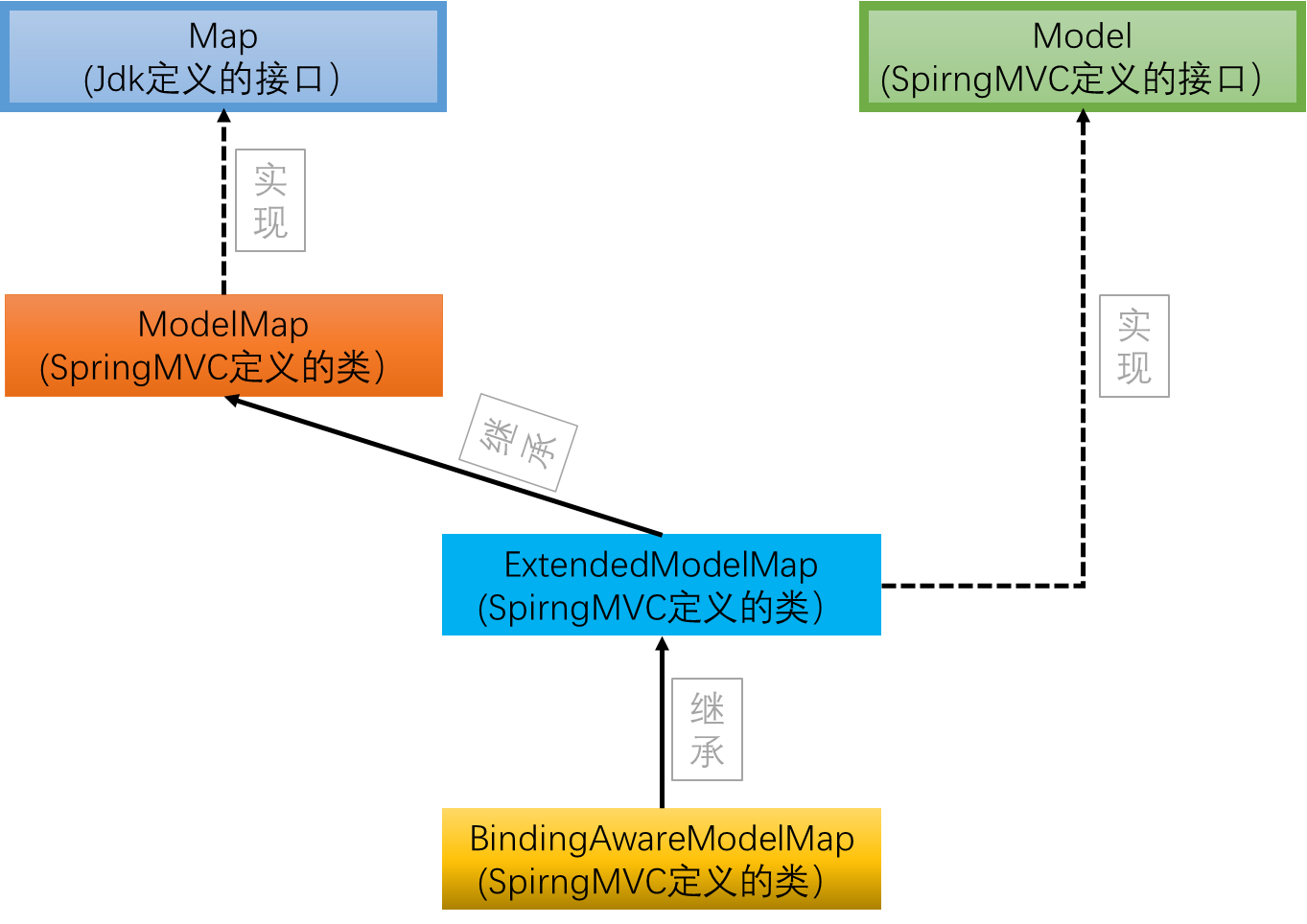
## 2. Map、ModelMap、Model

①SpringMVC提供了Model,Map,ModelMap来帮我们向页面传值

②这三种方式都可以向页面传值，且默认将数据，保存在请求域中！三种方式的最终实现类都是：

org.springframework.validation.support.BindingAwareModelMap

三种类型的关系：BindingAwareModelMap是Map、ModelMap、Model的实现类



## 3. ModelAndView

* 1. 控制器处理方法的返回值如果为 ModelAndView, 则其既包含视图信息，也包含模型数据信息。ModelAndView的数据也会放入到请求域中。
  2. 添加模型数据:

MoelAndView addObject(String attributeName, Object attributeValue)

ModelAndView addAllObject(Map<String, ?> modelMap)

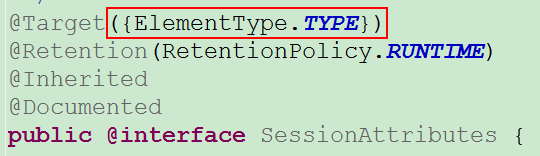
* 1. 设置视图:

void setView(View view)

void setViewName(String viewName)

## 4. @SesssionAttributes

@SesssionAttributes可以向Session域中保存数据！，这个注解，只能标注在类上面！





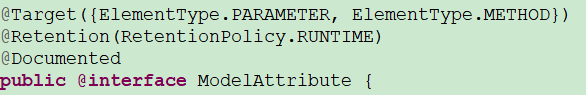
属性：

value: 保存指定Key的数据到session域中。

types: 保存指定类型的数据到session域中。

## 5. @ModelAttribute

①@ModelAttribute注解可以标注在方法上和参数上。



②@ModelAttribute注解标注在方法上，此方法会在这个类的任意请求方法执行之前先执行，并将方法的返回值保存到隐含模型中。

保存的规则：

key: @ModelAttribute（"key"）注解标注的key；

没有标，就是返回值类型字母小写。如：返回Book，就是book、返回void，就是void；

value：@ModelAttribute方法的返回值。如果返回void，就是null

③@ModelAttribute标注在参数上，可以给参数赋值！

当执行请求方法的时候，此时会遵循以下步骤来创建POJO：

确定key：

a):标注了@ModelAttribute("xx")注解，且有value属性，key就是xx；

b):否则就是类名首字母小写；

从隐含模型中根据key取出对应的值：

a如果在隐含模型中成功找到key对应的值，就赋值！

b)如果隐含模型中没有，判断当前类是否标注了@SessionAttributes注解

如果标了，尝试从session域中取出值，赋值！

有可能取出的值为null，就报错！(也就是虽然在@SessionAttributes的value属性中有key，但是在session中并没有，所以没有值)报错

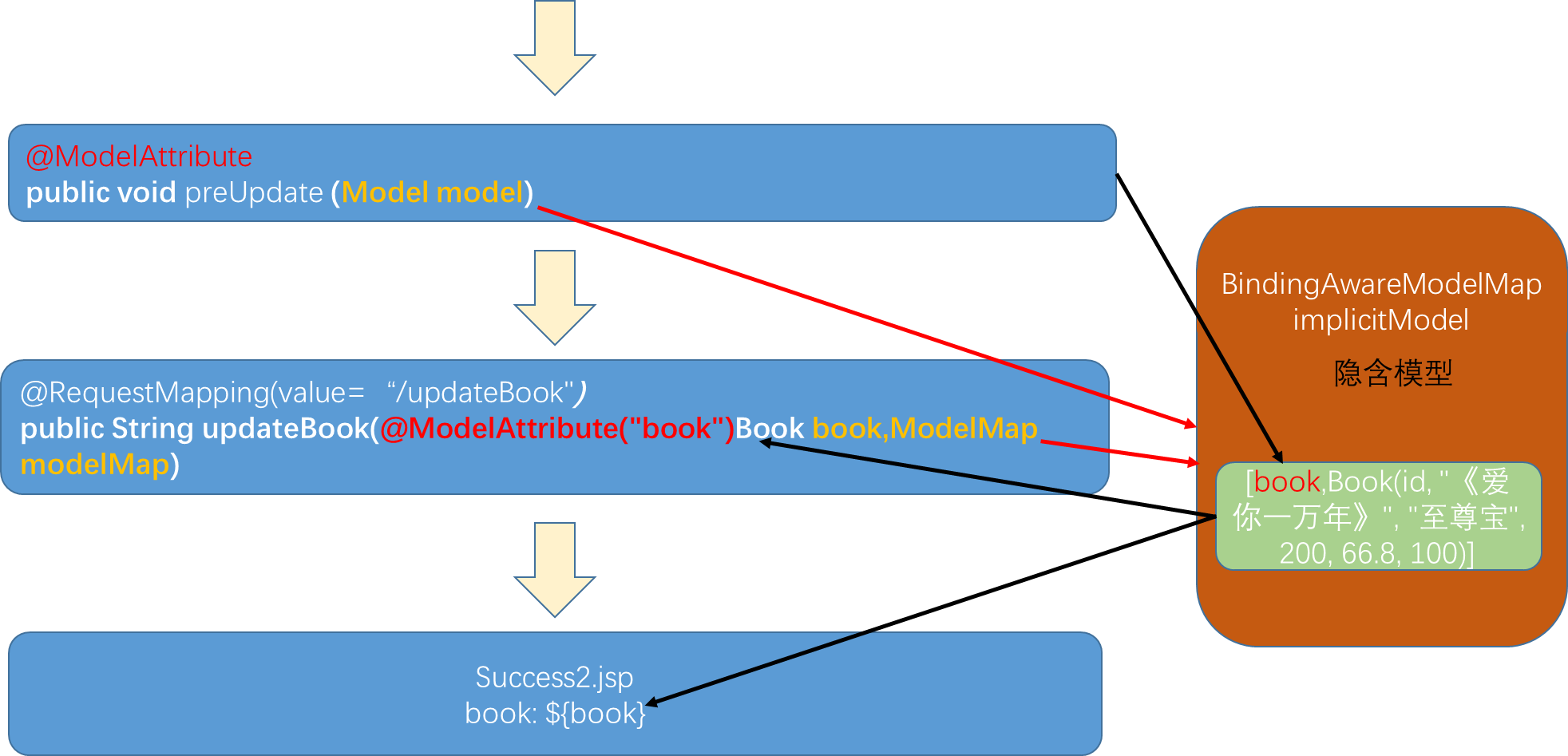
HTTP Status 500 - Session attribute 'book' required - not found in session

c)以上两步都无法确定，直接利用反射，创建一个新的对象！

## 6 . 隐含模型

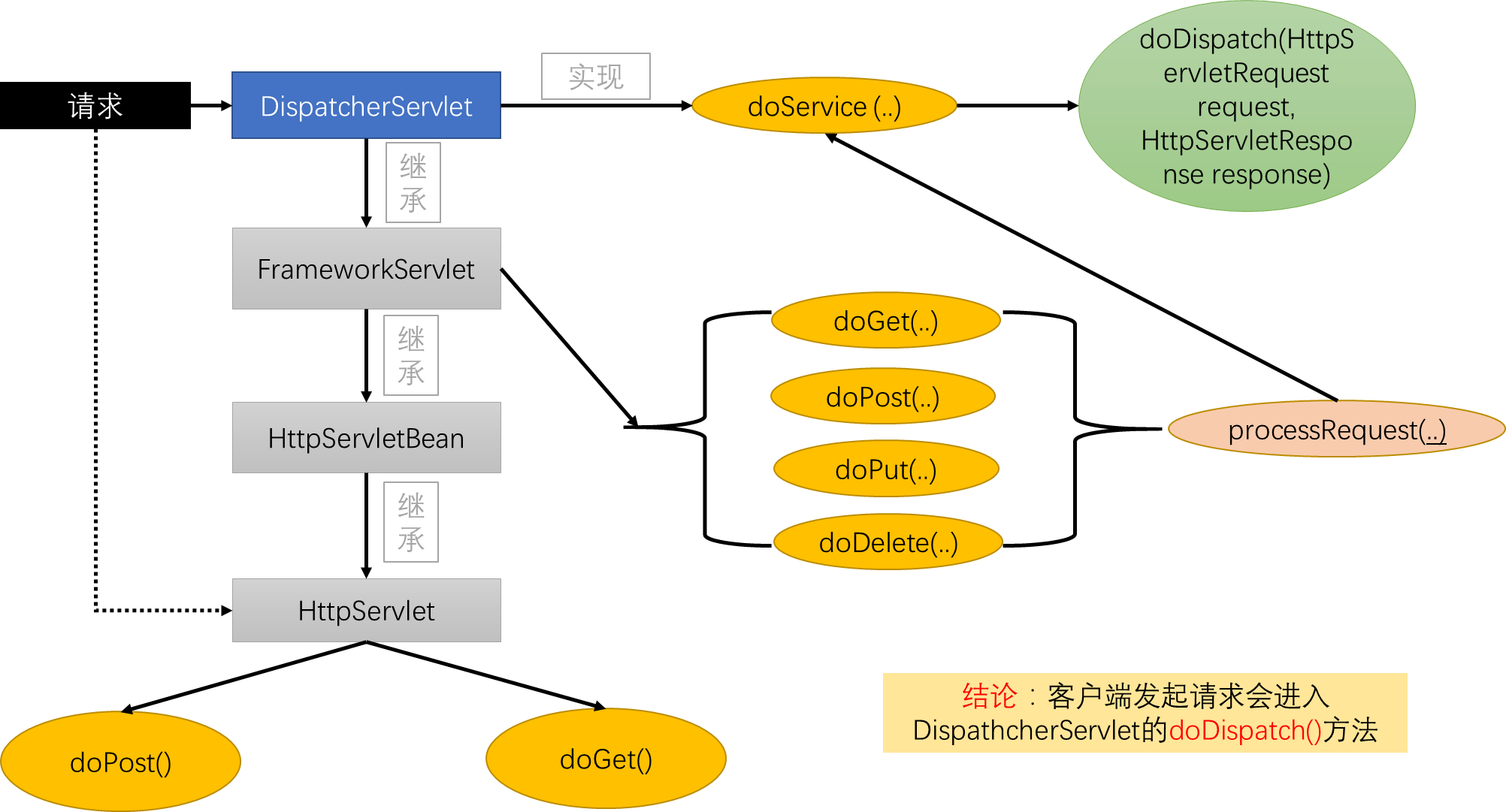
伴随着一次请求，不管在@ModelAttribute标注的方法中，声明的Model，或者在处理请求方法中标注的Model、Map、ModelMap，都指向一个贯穿始终的BindingAwareModelMap，这个BindingAwareModelMap的参数名在底层源码中为implicitModel，翻译过来，我们将其称为隐含模型。用来保存数据！

隐含模型中的数据，都会被放入到请求域中！



# 七、源码（了解）

## 1. 入口



真正处理方法的doDispathch()

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

HttpServletRequest processedRequest = request;

HandlerExecutionChain mappedHandler = **null**;

**boolean** multipartRequestParsed = **false**;

WebAsyncManager asyncManager = WebAsyncUtils.*getAsyncManager*(request);

**try** {

// ModelAndView可以封装数据和页面视图信息

ModelAndView mv = **null**;

Exception dispatchException = **null**;

**try** {

processedRequest = checkMultipart(request);

multipartRequestParsed = processedRequest != request;

//根据当前请求的url，找到匹配的Hander，包装为HandlerExcutionChain的对象

mappedHandler = getHandler(processedRequest);

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

//找不到可以处理当前请求的Handler，就报错404

noHandlerFound(processedRequest, response);

**return**;

}

//拿到当前Handler的适配器，相当于对Handler进行了包装，包装为一个反射工具 HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

// Process last-modified header, if supported by the handler.

String method = request.getMethod();

**boolean** isGet = "GET".equals(method);

**if** (isGet || "HEAD".equals(method)) {

**long** lastModified = ha.getLastModified(request, mappedHandler.getHandler());

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

String requestUri = ***urlPathHelper***.getRequestUri(request);

logger.debug("Last-Modified value for [" + requestUri + "] is: " + lastModified);

}

**if** (**new** ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {

**return**;

}

}

**if** (!mappedHandler.applyPreHandle(processedRequest, response)) {

**return**;

}

**try** {

//真正执行处理请求方法的地方

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

}

**finally** {

**if** (asyncManager.isConcurrentHandlingStarted()) {

**return**;

}

}

applyDefaultViewName(request, mv);

mappedHandler.applyPostHandle(processedRequest, response, mv);

}

**catch** (Exception ex) {

dispatchException = ex;

}

//将数据分发到页面渲染

processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);

}

**catch** (Exception ex) {

triggerAfterCompletion(processedRequest, response, mappedHandler, ex);

}

**catch** (Error err) {

triggerAfterCompletionWithError(processedRequest, response, mappedHandler, err);

}

**finally** {

**if** (asyncManager.isConcurrentHandlingStarted()) {

// Instead of postHandle and afterCompletion

mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);

**return**;

}

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

**if** (multipartRequestParsed) {

cleanupMultipart(processedRequest);

}

}

}

## 2. 获取处理当前请求的Handler



HanderMapping：处理器映射器，通过解析当前请求的url路径，找到对应可以处理当前url的Handler。

决定处理当前请求的handler

**protected** HandlerExecutionChain getHandler(HttpServletRequest request) **throws** Exception {

**for** (HandlerMapping hm : **this**.handlerMappings) {

**if** (logger.isTraceEnabled()) {

logger.trace(

"Testing handler map [" + hm + "] in DispatcherServlet with name '" + getServletName() + "'");

}

HandlerExecutionChain handler = hm.getHandler(request);

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

**return** handler;

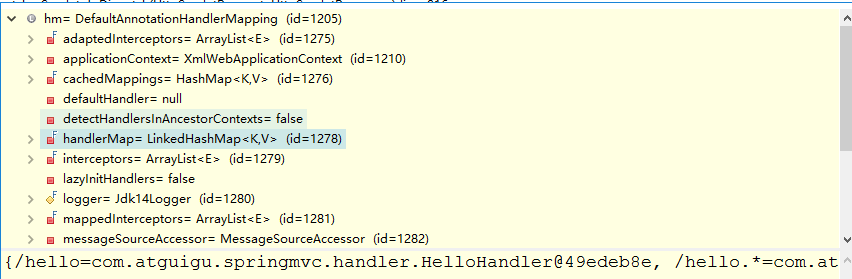
}

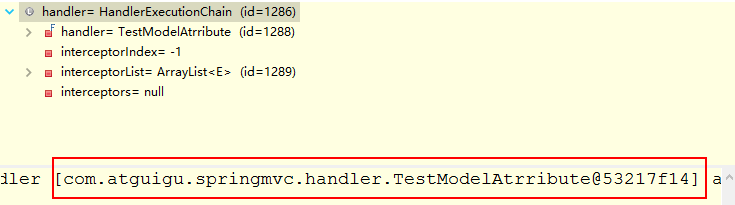
}

**return** **null**;

}

我们使用@RequestMapping注解，标识了方法可以处理的请求信息，于是默认使用DefaultAnnotaionHandlerMapping进行解析，解析完，将url与可以处理的类的映射信息，放入HandlerMap属性中。





## 3. 包装为适配器

根据当前处理器，包装为一个Handler的适配器（高级的反射工具）。最后调用适配器来执行Handler中的方法



## 4. 执行目标方法处理请求

### 4.1 进入handle方法进行处理

@Override

**public** ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

……

**return** invokeHandlerMethod(request, response, handler);

}

**protected** ModelAndView invokeHandlerMethod(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

//找到当前要使用的Handler

ServletHandlerMethodResolver methodResolver = getMethodResolver(handler);

//根据当前的url请求和@Requestmapping的信息，找到处理请求的方法

Method handlerMethod = methodResolver.resolveHandlerMethod(request);

ServletHandlerMethodInvoker methodInvoker = **new** ServletHandlerMethodInvoker(methodResolver);

ServletWebRequest webRequest = **new** ServletWebRequest(request, response);

//创建一个隐含模型

ExtendedModelMap implicitModel = **new** BindingAwareModelMap();

//指定请求的处理方法，详见4.2

Object result = methodInvoker.invokeHandlerMethod(handlerMethod, handler, webRequest, implicitModel);

//执行目标方法

ModelAndView mav =

methodInvoker.getModelAndView(handlerMethod, handler.getClass(), result, implicitModel, webRequest);

methodInvoker.updateModelAttributes(handler, (mav != **null** ? mav.getModel() : **null**), implicitModel, webRequest);

**return** mav;

}

在处理请求前，创建了一个BindingAwaerModelMap（1434），用来保存数据！我们称这个Map为隐含模型



### 4.2 真正执行目标方法的细节

**public** **final** Object invokeHandlerMethod(Method handlerMethod, Object handler,

NativeWebRequest webRequest, ExtendedModelMap implicitModel) **throws** Exception {

Method handlerMethodToInvoke = BridgeMethodResolver.*findBridgedMethod*(handlerMethod);

**try** {

**boolean** debug = ***logger***.isDebugEnabled();

**for** (String attrName : **this**.methodResolver.getActualSessionAttributeNames()) {

Object attrValue = **this**.sessionAttributeStore.retrieveAttribute(webRequest, attrName);

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

implicitModel.addAttribute(attrName, attrValue);

}

}

//通过方法解析器，找出所有标注了@ModelAttribute注解的方法

**for** (Method attributeMethod : **this**.methodResolver.getModelAttributeMethods()) {

Method attributeMethodToInvoke = BridgeMethodResolver.*findBridgedMethod*(attributeMethod);

//解析所有ModelAttibute方法的所有形参

Object[] args = resolveHandlerArguments(attributeMethodToInvoke, handler, webRequest, implicitModel);

**if** (debug) {

***logger***.debug("Invoking model attribute method: " + attributeMethodToInvoke);

}

//确定当前标注了@ModelAttibute(“xx”)的value属性，也就是xx

String attrName = AnnotationUtils.*findAnnotation*(attributeMethod, ModelAttribute.**class**).value();

**if** (!"".equals(attrName) && implicitModel.containsAttribute(attrName)) {

**continue**;

}

ReflectionUtils.*makeAccessible*(attributeMethodToInvoke);

//执行标注了@ModelAttibute的方法！先于目标方法执行

Object attrValue = attributeMethodToInvoke.invoke(handler, args);

**if** ("".equals(attrName)) {

//如果@Modelattibute注解的value属性是“”,就进入此方法

Class<?> resolvedType = GenericTypeResolver.*resolveReturnType*(attributeMethodToInvoke, handler.getClass());

//将返回值类型的首字母小写作为attrName

attrName = Conventions.*getVariableNameForReturnType*(attributeMethodToInvoke, resolvedType, attrValue);

}

//判断隐含模型中是否包含attrName

**if** (!implicitModel.containsAttribute(attrName)) {

//将ModelAttribute标注方法的返回值作为值，attrName为key，加入到隐含模型

implicitModel.addAttribute(attrName, attrValue);

}

}

//解析处理请求的目标方法的参数

Object[] args = resolveHandlerArguments(handlerMethodToInvoke, handler, webRequest, implicitModel);

**if** (debug) {

***logger***.debug("Invoking request handler method: " + handlerMethodToInvoke);

}

ReflectionUtils.*makeAccessible*(handlerMethodToInvoke);

//真正执行处理请求的目标方法

**return** handlerMethodToInvoke.invoke(handler, args);

}

**……**

}

### 4.3 解析方法参数

**private** Object[] resolveHandlerArguments(Method handlerMethod, Object handler,

NativeWebRequest webRequest, ExtendedModelMap implicitModel) **throws** Exception {

//获取当前方法所有的参数

Class<?>[] paramTypes = handlerMethod.getParameterTypes();

//创建一个数组，这个数组大小跟当前方法的参数个数一致

Object[] args = **new** Object[paramTypes.length];

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

MethodParameter methodParam = **new** MethodParameter(handlerMethod, i);

methodParam.initParameterNameDiscovery(**this**.parameterNameDiscoverer);

GenericTypeResolver.*resolveParameterType*(methodParam, handler.getClass());

String paramName = **null**;

String headerName = **null**;

**boolean** requestBodyFound = **false**;

String cookieName = **null**;

String pathVarName = **null**;

String attrName = **null**;

**boolean** required = **false**;

String defaultValue = **null**;

**boolean** validate = **false**;

Object[] validationHints = **null**;

**int** annotationsFound = 0;

Annotation[] paramAnns = methodParam.getParameterAnnotations();

//找到参数上标注的所有注解，并且确定注解的各个属性

**for** (Annotation paramAnn : paramAnns) {

**if** (RequestParam.**class**.isInstance(paramAnn)) {

RequestParam requestParam = (RequestParam) paramAnn;

paramName = requestParam.value();

required = requestParam.required();

defaultValue = parseDefaultValueAttribute(requestParam.defaultValue());

annotationsFound++;

}

**else** **if** (RequestHeader.**class**.isInstance(paramAnn)) {

RequestHeader requestHeader = (RequestHeader) paramAnn;

headerName = requestHeader.value();

required = requestHeader.required();

defaultValue = parseDefaultValueAttribute(requestHeader.defaultValue());

annotationsFound++;

}

**else** **if** (RequestBody.**class**.isInstance(paramAnn)) {

requestBodyFound = **true**;

annotationsFound++;

}

**else** **if** (CookieValue.**class**.isInstance(paramAnn)) {

CookieValue cookieValue = (CookieValue) paramAnn;

cookieName = cookieValue.value();

required = cookieValue.required();

defaultValue = parseDefaultValueAttribute(cookieValue.defaultValue());

annotationsFound++;

}

**else** **if** (PathVariable.**class**.isInstance(paramAnn)) {

PathVariable pathVar = (PathVariable) paramAnn;

pathVarName = pathVar.value();

annotationsFound++;

}

//如果参数上面标注了@ModelAttribute注解，就确定参数的key，也就是@ModelAttribute的value值

**else** **if** (ModelAttribute.**class**.isInstance(paramAnn)) {

ModelAttribute attr = (ModelAttribute) paramAnn;

attrName = attr.value();

annotationsFound++;

}

**else** **if** (Value.**class**.isInstance(paramAnn)) {

defaultValue = ((Value) paramAnn).value();

}

**else** **if** (paramAnn.annotationType().getSimpleName().startsWith("Valid")) {

validate = **true**;

Object value = AnnotationUtils.*getValue*(paramAnn);

validationHints = (value **instanceof** Object[] ? (Object[]) value : **new** Object[] {value});

}

}

//如果找到多个注解

**if** (annotationsFound > 1) {

**throw** **new** IllegalStateException("Handler parameter annotations are exclusive choices - " +

"do not specify more than one such annotation on the same parameter: " + handlerMethod);

}

//如果当前方法的参数上一个注解都没有标

**if** (annotationsFound == 0) {

//看是否是九个原生API，具体解析，详见4.3

Object argValue = resolveCommonArgument(methodParam, webRequest);

**if** (argValue != WebArgumentResolver.***UNRESOLVED***) {

args[i] = argValue;

}

**else** **if** (defaultValue != **null**) {

args[i] = resolveDefaultValue(defaultValue);

}

**else** {

//获取参数类型

Class<?> paramType = methodParam.getParameterType();

//如果是Model类型或Map类型，就把隐含模型赋值为这个参数

**if** (Model.**class**.isAssignableFrom(paramType) || Map.**class**.isAssignableFrom(paramType)) {

**if** (!paramType.isAssignableFrom(implicitModel.getClass())) {

**throw** **new** IllegalStateException("Argument [" + paramType.getSimpleName() + "] is of type " +

"Model or Map but is not assignable from the actual model. You may need to switch " +

"newer MVC infrastructure classes to use this argument.");

}

args[i] = implicitModel;

}

**else** **if** (SessionStatus.**class**.isAssignableFrom(paramType)) {

args[i] = **this**.sessionStatus;

}

**else** **if** (HttpEntity.**class**.isAssignableFrom(paramType)) {

args[i] = resolveHttpEntityRequest(methodParam, webRequest);

}

**else** **if** (Errors.**class**.isAssignableFrom(paramType)) {

**throw** **new** IllegalStateException("Errors/BindingResult argument declared " +

"without preceding model attribute. Check your handler method signature!");

}

**else** **if** (BeanUtils.*isSimpleProperty*(paramType)) {

paramName = "";

}

**else** {

attrName = "";

}

}

}

//为标注了@RequestParam、@CookieValue、@RequestHeader等参数赋值！

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

args[i] = resolveRequestParam(paramName, required, defaultValue, methodParam, webRequest, handler);

}

**else** **if** (headerName != **null**) {

args[i] = resolveRequestHeader(headerName, required, defaultValue, methodParam, webRequest, handler);

}

**else** **if** (requestBodyFound) {

args[i] = resolveRequestBody(methodParam, webRequest, handler);

}

**else** **if** (cookieName != **null**) {

args[i] = resolveCookieValue(cookieName, required, defaultValue, methodParam, webRequest, handler);

}

**else** **if** (pathVarName != **null**) {

args[i] = resolvePathVariable(pathVarName, methodParam, webRequest, handler);

}

**else** **if** (attrName != **null**) {

//创建POJO，并且包装为WebDataBinder

WebDataBinder binder =

resolveModelAttribute(attrName, methodParam, implicitModel, webRequest, handler);

**boolean** assignBindingResult = (args.length > i + 1 && Errors.**class**.isAssignableFrom(paramTypes[i + 1]));

**if** (binder.getTarget() != **null**) {

// 将WebDataBinder中的目标POJO和请求参数一一赋值

doBind(binder, webRequest, validate, validationHints, !assignBindingResult);

}

args[i] = binder.getTarget();

**if** (assignBindingResult) {

args[i + 1] = binder.getBindingResult();

i++;

}

implicitModel.putAll(binder.getBindingResult().getModel());

}

}

**return** args;

}

文字版：

①拿到方法上面标注的所有参数的类型

②穿一个和参数个数一样的数组，用来放解析后的参数

③解析参数：

a)拿到参数上标注的所有注解

获取注解上面的所有属性的值

b)看是否是标注参数（九个原生API）

是标准参数，就返回；

c)是否是Model，Map,ModelMap，如果是，就直接将隐含模型赋值！

创建POJO对象

**private** WebDataBinder resolveModelAttribute(String attrName, MethodParameter methodParam,

ExtendedModelMap implicitModel, NativeWebRequest webRequest, Object handler) **throws** Exception {

// Bind request parameter onto object...

String name = attrName;

//判断pojo的key，如果key是空串就将类名首字母小写返回作为key

**if** ("".equals(name)) {

name = Conventions.*getVariableNameForParameter*(methodParam);

}

Class<?> paramType = methodParam.getParameterType();

Object bindObject;

//判断隐含模型中有没有这个Key

**if** (implicitModel.containsKey(name)) {

//如果有，就直接赋值

bindObject = implicitModel.get(name);

}

//如果隐含模型中找不到对应的key，就看是否这个类标注了@SesssionAttibute注解

**else** **if** (**this**.methodResolver.isSessionAttribute(name, paramType)) {

//从session域中，取出当前的key

bindObject = **this**.sessionAttributeStore.retrieveAttribute(webRequest, name);

//取出的值是null

**if** (bindObject == **null**) {

//抛出500异常

raiseSessionRequiredException("Session attribute '" + name + "' required - not found in session");

}

}

//最后如果都无法找到，就使用反射创建一个新的对象

**else** {

bindObject = BeanUtils.*instantiateClass*(paramType);

}

WebDataBinder binder = createBinder(webRequest, bindObject, name);

initBinder(handler, name, binder, webRequest);

**return** binder;

}

### 4.4 判断是否是原生的九个API：

@Override

**protected** Object resolveStandardArgument(Class<?> parameterType, NativeWebRequest webRequest) **throws** Exception {

HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.**class**);

HttpServletResponse response = webRequest.getNativeResponse(HttpServletResponse.**class**);

**if** (ServletRequest.**class**.isAssignableFrom(parameterType) ||

MultipartRequest.**class**.isAssignableFrom(parameterType)) {

Object nativeRequest = webRequest.getNativeRequest(parameterType);

**if** (nativeRequest == **null**) {

**throw** **new** IllegalStateException(

"Current request is not of type [" + parameterType.getName() + "]: " + request);

}

**return** nativeRequest;

}

**else** **if** (ServletResponse.**class**.isAssignableFrom(parameterType)) {

**this**.responseArgumentUsed = **true**;

Object nativeResponse = webRequest.getNativeResponse(parameterType);

**if** (nativeResponse == **null**) {

**throw** **new** IllegalStateException(

"Current response is not of type [" + parameterType.getName() + "]: " + response);

}

**return** nativeResponse;

}

**else** **if** (HttpSession.**class**.isAssignableFrom(parameterType)) {

**return** request.getSession();

}

**else** **if** (Principal.**class**.isAssignableFrom(parameterType)) {

**return** request.getUserPrincipal();

}

**else** **if** (Locale.**class**.equals(parameterType)) {

**return** RequestContextUtils.*getLocale*(request);

}

**else** **if** (InputStream.**class**.isAssignableFrom(parameterType)) {

**return** request.getInputStream();

}

**else** **if** (Reader.**class**.isAssignableFrom(parameterType)) {

**return** request.getReader();

}

**else** **if** (OutputStream.**class**.isAssignableFrom(parameterType)) {

**this**.responseArgumentUsed = **true**;

**return** response.getOutputStream();

}

**else** **if** (Writer.**class**.isAssignableFrom(parameterType)) {

**this**.responseArgumentUsed = **true**;

**return** response.getWriter();

}

**return** **super**.resolveStandardArgument(parameterType, webRequest);

}

## 5.结论

### 5.1. ModelAndView

目标方法执行后，不管返回值是什么样，都会包装为一个ModelAndView



详见3.1

### 5.2. implicitModel

在目标方法执行之前，会创建一个implicitModel隐含模型，并且会在目标方法执行期间，将创建的隐含模型作为参数传入到目标方法。

AnnotationMethodHandlerAdapter.class 的444行：



隐含模型伴随着目标方法，以及@ModelAttribute标注的方法

执行目标方法 ：AnnotationMethodHandlerAdapter.class 的446行：



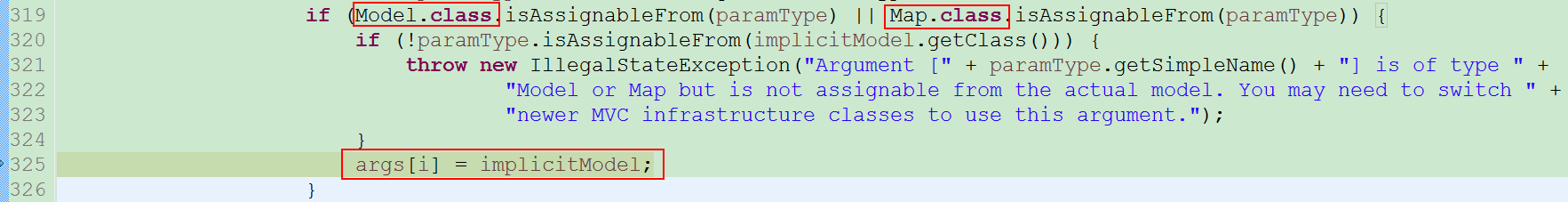
### 5.3. 原生的Servlet API

如果方法入参声明了原生的Servlet API参数，如何解析？详见4.4

### 5.4. Model、Map、ModelMap

方法参数上面声明的Model、Map、ModelMap类型，都会指向原先创建的隐含模型

HandlerMethodInvoker的319行：



### 5.5. @ModelAttribute

标注了@ModelAttribute的方法会提前运行！且提前将数据保存在了隐含模型中！并且将这个隐含模型传入到了目标方法的执行中！



# 八、视图

## 1. forward:

使用forward:可以完成一个转发操作；

格式：　forward:/; 转发的路径我们推荐使用绝对路径，即加上/

使用forward:后，转发的路径不会再经过我们配置的视图解析器来解析！

## 2. redirect:

使用redirect:完成重定向，SpringMVC会自动为我们拼上项目名

格式： redirect:/，重定向的路径也必须使用绝对路径，即加上/

使用redirect:后同样不会经过我们默认配置的视图解析器！

## 3.源码（了解）

### 3.1 入口



### 3.2 处理

#### 3.2.1 processDispatchResult（）

**private** **void** processDispatchResult(HttpServletRequest request, HttpServletResponse response,

HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception) **throws** Exception {

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

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

**if** (exception **instanceof** ModelAndViewDefiningException) {

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

mv = ((ModelAndViewDefiningException) exception).getModelAndView();

}

**else** {

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

mv = processHandlerException(request, response, handler, exception);

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

}

}

// Did the handler return a view to render?

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

render(mv, request, 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");

}

}

**if** (WebAsyncUtils.*getAsyncManager*(request).isConcurrentHandlingStarted()) {

// Concurrent handling started during a forward

**return**;

}

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

mappedHandler.triggerAfterCompletion(request, response, **null**);

}

}

#### 3.2.2 render()是渲染视图操作

①调用 resoleViewName()，根据ModelAndView 中保存的视图名，解析得到对应的视图对象！

**protected** **void** render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response) **throws** Exception {

// Determine locale for request and apply it to the response.

Locale locale = **this**.localeResolver.resolveLocale(request);

response.setLocale(locale);

View view;

**if** (mv.isReference()) {

//调用视图解析器，通过对视图名的解析，得到View对象

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

**if** (view == **null**) {

**throw** **new** ServletException(

"Could not resolve view with name '" + mv.getViewName() + "' in servlet with name '" +

getServletName() + "'");

}

}

**else** {

// No need to lookup: the ModelAndView object contains the actual View object.

view = mv.getView();

**if** (view == **null**) {

**throw** **new** ServletException("ModelAndView [" + mv + "] neither contains a view name nor a " +

"View object in servlet with name '" + getServletName() + "'");

}

}

// Delegate to the View object for rendering.

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

logger.debug("Rendering view [" + view + "] in DispatcherServlet with name '" + getServletName() + "'");

}

**try** {

//调用View的render方法，觉得是重定向还是转发包括，将数据传输到页面

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

}

**catch** (Exception ex) {

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

logger.debug("Error rendering view [" + view + "] in DispatcherServlet with name '"

+ getServletName() + "'", ex);

}

**throw** ex;

}

}

#### 3.2.3 解析视图

**protected** View resolveViewName(String viewName, Map<String, Object> model, Locale locale,

HttpServletRequest request) **throws** Exception {

//调用容器中所有的视图解析器，轮流解析，如果有一个可以解析成功就返回view对象

**for** (ViewResolver viewResolver : **this**.viewResolvers) {

View view = viewResolver.resolveViewName(viewName, locale);

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

**return** view;

}

}

**return** **null**;

}

#### 3.2.4 render() 渲染视图

@Override

**public** **void** render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response) **throws** Exception {

**……**

renderMergedOutputModel(mergedModel, request, response);

}

@Override

**protected** **void** renderMergedOutputModel(

Map<String, Object> model, HttpServletRequest request, HttpServletResponse response) **throws** Exception {

// Determine which request handle to expose to the RequestDispatcher.

HttpServletRequest requestToExpose = getRequestToExpose(request);

//将隐含模型中的数据，暴露到请求域中

exposeModelAsRequestAttributes(model, requestToExpose);

// Expose helpers as request attributes, if any.

exposeHelpers(requestToExpose);

// Determine the path for the request dispatcher.

String dispatcherPath = prepareForRendering(requestToExpose, response);

// Obtain a RequestDispatcher for the target resource (typically a JSP).

RequestDispatcher rd = getRequestDispatcher(requestToExpose, dispatcherPath);

**……**

rd.forward(requestToExpose, response);

}

}

**protected** **void** exposeModelAsRequestAttributes(Map<String, Object> model, HttpServletRequest request) **throws** Exception {

//将隐含模型中的数据遍历，且放入请求域！

**for** (Map.Entry<String, Object> entry : model.entrySet()) {

String modelName = entry.getKey();

Object modelValue = entry.getValue();

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

request.setAttribute(modelName, modelValue);

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

logger.debug("Added model object '" + modelName + "' of type [" + modelValue.getClass().getName() +

"] to request in view with name '" + getBeanName() + "'");

}

}

**else** {

request.removeAttribute(modelName);

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

logger.debug("Removed model object '" + modelName +

"' from request in view with name '" + getBeanName() + "'");

}

}

}

}

#### 3.2.5 总结

①处理请求的方法，最终会返回一个ModelAndView,既包含了视图（视图名），又包含了数据模型；

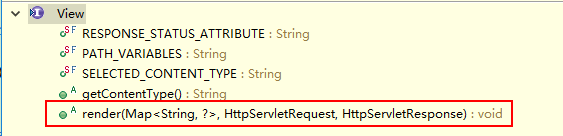
②进入processDispatchResult(),执行了关键代码render()

a) 拿到容器中所有的ViewResolver(视图解析器),轮番对我们视图名进行解析，解析调用的是resolveViewName(),返回一个View对象

b)调用View的render()，核心方法是renderMergedOutputModel()方法，进行转发或重定向，渲染数据.

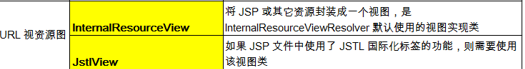
## 4. View

View就是视图，视图的形式有很多，可以是一个html页面，也就是一个PDF，也可以是一个程序！



render()；就是将隐含模型中的数据，放入请求域中！还可以觉得是采取转发操作还是重定向操作！

常见的视图：



RedirectView如何完成重定向操作：

@Override

**protected** **void** renderMergedOutputModel(Map<String, Object> model, HttpServletRequest request,

HttpServletResponse response) **throws** IOException {

//找到/，自动拼接上项目名

String targetUrl = createTargetUrl(model, request);

……

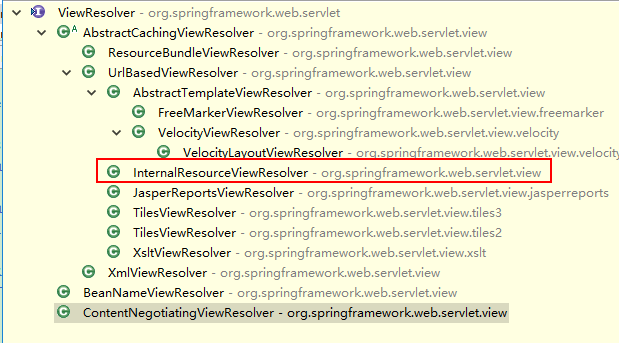
//重定向

sendRedirect(request, response, targetUrl, **this**.http10Compatible);

}

## 5. ViewResolver





视图解析器，作用就是根据视图名，解析得到对应的View视图对象。

InteranResourceViewResouvler,解析后，返回的是InteranResourceView！

### 1. BeanNameViewResolver



作用： 将视图名，作为id，去容器中寻找对应的视图对象！

### 2. InternalResourceViewResolver

如何解析视图的：

**public** View resolveViewName(String viewName, Locale locale) **throws** Exception {

**if** (!isCache()) {

**return** createView(viewName, locale);

}

**else** {

Object cacheKey = getCacheKey(viewName, locale);

View view = **this**.viewAccessCache.get(cacheKey);

**if** (view == **null**) {

**synchronized** (**this**.viewCreationCache) {

view = **this**.viewCreationCache.get(cacheKey);

**if** (view == **null**) {

// Ask the subclass to create the View object.

view = createView(viewName, locale);

**if** (view == **null** && **this**.cacheUnresolved) {

view = ***UNRESOLVED\_VIEW***;

}

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

**this**.viewAccessCache.put(cacheKey, view);

**this**.viewCreationCache.put(cacheKey, view);

**if** (logger.isTraceEnabled()) {

logger.trace("Cached view [" + cacheKey + "]");

}

}

}

}

}

**return** (view != ***UNRESOLVED\_VIEW*** ? view : **null**);

}

}

InternalResourceViewResolver如何创建一个视图

@Override

**protected** View createView(String viewName, Locale locale) **throws** Exception {

//根据视图名，判断当前的视图解析器，能否解析，如果不能，就返回null，交给其他视图解析器解析

**if** (!canHandle(viewName, locale)) {

**return** **null**;

}

//检查视图名是否包含 "redirect:" 前缀

**if** (viewName.startsWith(***REDIRECT\_URL\_PREFIX***)) {

String redirectUrl = viewName.substring(***REDIRECT\_URL\_PREFIX***.length());

//如果我们带了redirect:，就返回一个RedirectView

RedirectView view = **new** RedirectView(redirectUrl, isRedirectContextRelative(), isRedirectHttp10Compatible());

**return** applyLifecycleMethods(viewName, view);

}

//检查视图名是否包含 "forward:"前缀

**if** (viewName.startsWith(***FORWARD\_URL\_PREFIX***)) {

String forwardUrl = viewName.substring(***FORWARD\_URL\_PREFIX***.length());

//创建一个新的InternalResourceView()对象，所以就不会再给你拼串

**return** **new** InternalResourceView(forwardUrl);

}

// 使用当前的视图解析器，创建一个视图对象

**return** **super**.createView(viewName, locale);

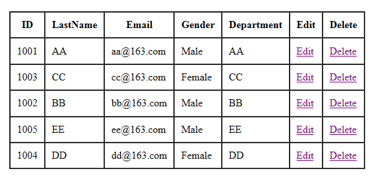
}

# 九、RestfulCRUD

## 1. 查询

### 1.1 要求

发送url: emps , GET,得到所有员工信息：



### 1.2 页面

<table border=*"0"* cellpadding=*"3"* cellspacing=*"1"* bgcolor=*"#c1c1c1"* style="text-align:*center*">

<tr bgcolor=*"#FFFFFF"*>

<td>ID</td>

<td>LASTNAME</td>

<td>GENDER</td>

<td>EMAIL</td>

<td>DEPARTMENT</td>

<td>EDIT</td>

<td>DELETE</td>

</tr>

<c:forEach items=*"*${emps}*"* var=*"emp"*>

<tr bgcolor=*"#FFFF00"*>

<td>${emp.id}</td>

<td>${emp.lastName}</td>

<td>${emp.gender==0?'小仙女':'小鲜肉'}</td>

<td>${emp.email}</td>

<td>${emp.department.departmentName}</td>

<td>EDIT</td>

<td>DELETE</td>

</tr>

</c:forEach>

</table> <br>

<a href=*"*${ctp}*/emp"*>添加员工</a>

### 1.3 后台

//查询所有员工

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

**public** String getAll(Model model){

//从数据库查询所有的员工

Collection<Employee> all = empDao.getAll();

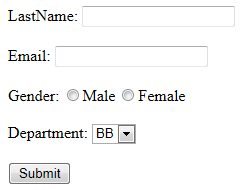
//将查出的结果放入请求域，送到页面回显

model.addAttribute("emps", all);

**return** "list";

}

## 2. 添加



### 2.1 思路

①在页面点击添加连接，发送 请求/emp,GET方式

②后台处理请求，转发到添加员工的输入页面

③在输入页面编辑完成后，发送 /emp ,POST方式

④后台处理POST请求，调用Dao，保存数据，重定向来到员工显示的页面

### 2.2 页面

<form action=*"*${ctp}*/emp"* method=*"post"*>

LASTNAME:<input name=*"lastName"*><br>

EMAIL:<input name=*"email"*><br>

GENDER:<c:forEach items=*"*${genders}*"* var=*"gender"*>

<input type=*"radio"* name=*"*${gender.key}*"*>${gender.value}&nbsp;

</c:forEach><br>

DEPARTMENT:<select name=*"department.id"*>

<c:forEach items=*"*${depts}*"* var=*"dept"*>

<option value=*"*${dept.id}*"*>${dept.departmentName}&nbsp;

</c:forEach>

</select><br>

<input value=*"添加员工"* type=*"submit"*>

</form>

### 2.3 后台

①处理请求，将部门和性别信息查询出，跳转到添加页面回显

//跳转到添加员工的输入页面

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

**public** String toAddPage(Model model){

//把部门和性别信息从数据库中查询出来，回显到页面

Collection<Department> departments = deptDao.getDepartments();

Map<Integer, String> genders = deptDao.getGenders();

model.addAttribute("depts", departments);

model.addAttribute("genders", genders);

**return** "add";

}

②页面提交数据后，真正完成添加操作

//真正添加员工保存到数据库的方法

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

**public** String addEmployee(Employee employee){

System.***out***.println("即将添加的员工信息："+employee);

empDao.save(employee);

**return** "redirect:/emps";

}

## 3.使用表单标签改造添加页面

①导入表单标签库



②编辑表单标签

<form:form action=*"*${ctp}*/emp"* method=*"post"* modelAttribute=*"emp"*>

LASTNAME:<form:input path=*"lastName"*/><br>

EMAIL:<form:input path=*"email"*/><br>

GENDER:<form:radiobuttons path=*"gender"* items=*"*${genders}*"*/><br>

DEPARTMENT:<form:select path=*"department.id"*

items=*"*${depts}*"* itemValue=*"id"* itemLabel=*"departmentName"*/><br>

<input value=*"添加员工"* type=*"submit"*>

</form:form>

③常见错误：



原因：

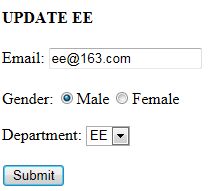
表单标签是用来快速回显请求域中的数据的！

每一个<form:input path="lastName"/>，如果你指定path属性，就一定要在请求域中，找到对应回显的数据！

默认去找key为command的一个POJO，看这个POJO里面有没有我们要回显的属性！

解决：将请求域中，放入要回显的对象，然后通过modelattribute属性，指定要回显对象的key

## 4. 修改



①在员工显示的页面，点击修改的超链接，发送 /emp/｛id｝ ，GET方式到后台

②后台根据指定的id，查询出要修改的员工，返回到修改页面，回显

③修改页面，重新编辑，发送 /emp/{id} ,PUT方式到后台

④处理请求，调用Dao，修改并保存，重定向到员工展示页面

问题：lastName不能被修改，因此在页面没有输入框，也就是请求参数中不会带有lastName属性。

解决：通过ModelAttribute注解，提前运行方法进行查询！

问题：报错如下



原因：标注了@ModelAttribute的方法会提前运行，因此在发送/emps请求时，也会运行，而此时通过@pathVariable注解获取id时，无法获取！

解决：将id作为请求参数来获取！

//提前将要修改的员工查询出来，放入到隐含模型中

@ModelAttribute("emp")

**public** Employee preUpdate(Integer id){

//需要一个id，有不能从路径上获取，当作请求参数获取

Employee employee=**new** Employee();

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

employee = empDao.get(id);

}

**return** employee;

}

## 5. 删除

发送 /emp/{id} ，DELETE请求，到后台！后台删除对应id的员工，重定向到员工展示页面！

问题：如何加载静态资源，如jquery文件



处理完之后，动态请求又无法访问了，如何解决：



页面：通过获取删除标签上面的href属性，赋值给form表单，然后通过js代码，提交表单的提交！

<script type=*"text/javascript"*>

$(**function**(){

//为删除的a标签绑定点击事件

$(".deleteClass").click(**function**(){

//获取发送的url

**var** url=$(**this**).attr("href");

//把url赋值表单的action属性，执行表单的提交

$("#form").attr("action",url).submit();

//取消自动跳转

**return** **false**;

});

});

</script>

# 十、组件化思想（了解）

## 1.九大组件

引入：

①根据当前请求，获取处理这个请求的Handler，这个Handler是通过HandlerMapping的对象来解析得到的！

HandlerMapping：处理器映射器；

②得到Handler，我们将其包装为一个HandlerAdapter的对象，最终是调用HandlerAdapter的handle()来具体处理请求！

③解析视图的对象叫ViewResolver，视图解析器！

总结：SpringMVC帮助我们完成各项功能是通过各种各样的组件来完成的！SpringMVC中有多少这种组件？九大组件！

**protected** **void** initStrategies(ApplicationContext context) {

initMultipartResolver(context);

initLocaleResolver(context);

initThemeResolver(context);

initHandlerMappings(context);

initHandlerAdapters(context);

initHandlerExceptionResolvers(context);

initRequestToViewNameTranslator(context);

initViewResolvers(context);

initFlashMapManager(context);

}

## 2.九大组件如何初始化：以ViewResolver为例子

**private** **void** initViewResolvers(ApplicationContext context) {

//创建一个viewResolver的集合，保存所有可以找到的viewResolver

**this**.viewResolvers = **null**;

//寻找容器中，所有的ViewResolver.class类型的对象，找到后，装配到原来的集合中

**if** (**this**.detectAllViewResolvers) {

// Find all ViewResolvers in the ApplicationContext, including ancestor contexts.

Map<String, ViewResolver> matchingBeans =

BeanFactoryUtils.*beansOfTypeIncludingAncestors*(context, ViewResolver.**class**, **true**, **false**);

**if** (!matchingBeans.isEmpty()) {

**this**.viewResolvers = **new** ArrayList<ViewResolver>(matchingBeans.values());

// We keep ViewResolvers in sorted order.

OrderComparator.*sort*(**this**.viewResolvers);

}

}

//如果容器中找不到视图解析器，就去找固定id为viewResolver的视图解析器

**else** {

**try** {

ViewResolver vr = context.getBean(***VIEW\_RESOLVER\_BEAN\_NAME***, ViewResolver.**class**);

**this**.viewResolvers = Collections.*singletonList*(vr);

}

**catch** (NoSuchBeanDefinitionException ex) {

// Ignore, we'll add a default ViewResolver later.

}

}

// Ensure we have at least one ViewResolver, by registering

//如果一个视图解析器都找不到，就执行默认策略，就是根据视图解析器的类名为key，从DispatcherSerlvet.properties文件中读取对应的值，也就是默认视图解析器的类型，给你创建一个默认的视图解析器！

**if** (**this**.viewResolvers == **null**) {

**this**.viewResolvers = getDefaultStrategies(context, ViewResolver.**class**);

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

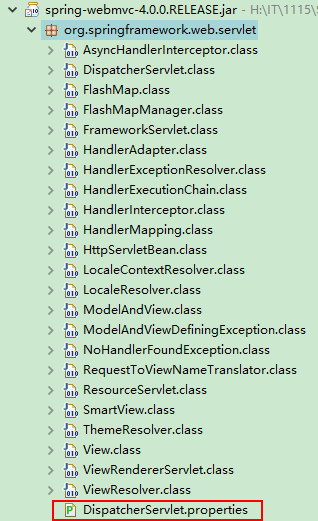
logger.debug("No ViewResolvers found in servlet '" + getServletName() + "': using default");

}

}

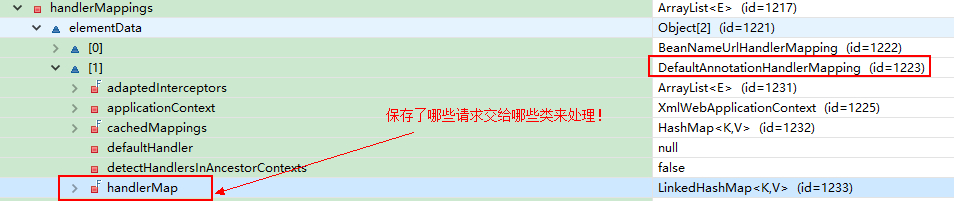
}

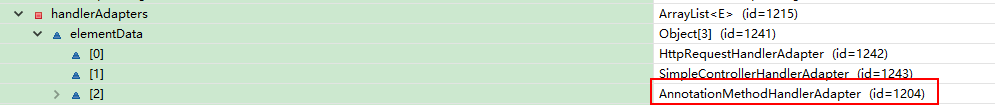
## 3.默认的初始化策略文件的位置



## 4. <mvc:annotationDriven>的作用（了解）

### 4.1. 什么都没有配：

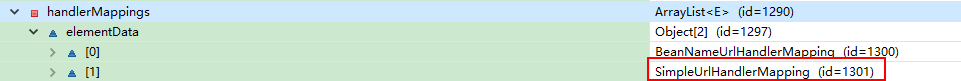




使用的是DefaultAnnotationHandlerMapping来保存处理器的映射信息；使用AnnotationMethodHandlerAdatper来处理请求方法！这个时候，静态资源无法访问，但是动态请求可以访问！

### 4.2. 单配<mvc:default-servlet-handler>

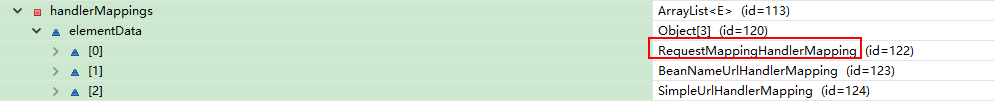


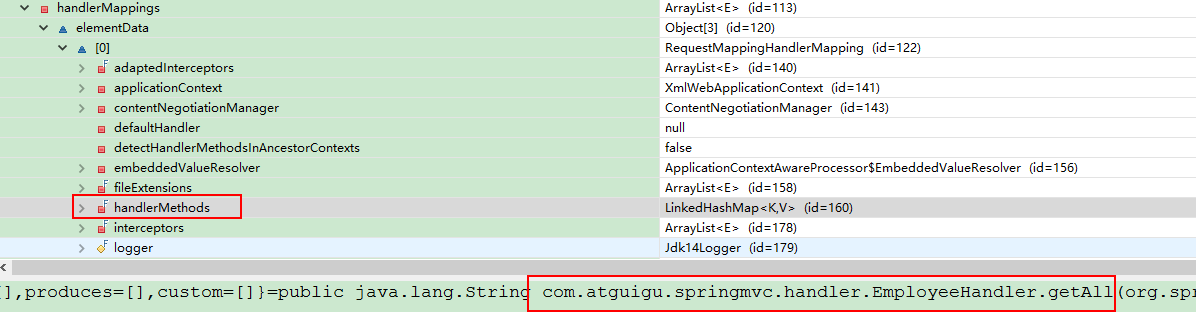


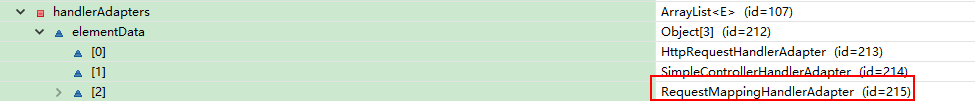


动态资源无法访问！

### 4.3. 加上<mvc:annotationDriven>



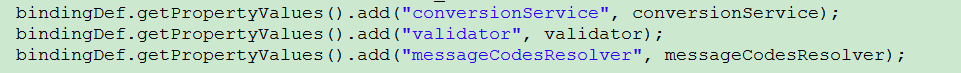




参考类：AnnotationDrivenBeanDefinationParser类：

标上<mvc:annotationDriven>后，往容器中加了各种组件！单车变摩托！

如：进行数据绑定的三个组件：



# 十一、数据绑定

## 1. 概念

我们在JSP或者是HTML页面表单中的数据，都是文本类型。而在处理请求的方法中，声明POJO的话，会将表单中的String自动赋值给POJO的属性。可是POJO的属性，有不同的类型，比如age是Integer类型，那么这个转换的过程统称为数据绑定。

数据绑定包括数据转换+数据格式化+数据校验。

①数据转换：页面传入的都是文本，也就是String。而我们POJO定义的属性有各种各样的类型；

数据转换就是将页面传入的文件，转为POJO定义的指定类型；

age:String---🡪Integer

②数据格式化： 将一些特殊的信息，如日期、数字转为指定的格式；

③数据校验： 对页面传入的信息，进行指定规则的验证，如非空、必须是邮箱格式；

## 2.流程

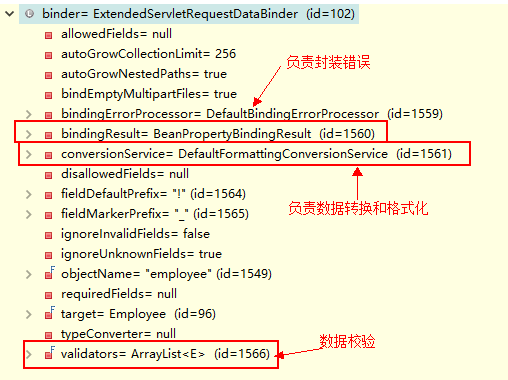
SpringMVC将请求对象Request(请求参数)和请求处理方法入参实例（POJO），传递给WebDataBinderFactory用来创建出一个WebDataBinder对象。

DataBinder对象，可以进行数据的数据类型转换+数据格式化+数据校验。

①调用装配在容器中的ConversionService组件，进行数据类型转换+数据格式化。通过converters进行数据转换！通过formatters进行数据格式化！

②调用Validator组件进行数据校验，校验出错误后，将错误抽取到BindingReuslt对象中；我们可以在请求方法的形参中声明该类型的参数，SpringMVC会把具体的BindingReuslt对象传递到对应的请求处理方法。

在请求处理方法中，就可以通过BindingReuslt获取具体的错误信息，还可以将信息返回到客户端显示。



@Override

**public** **final** Object resolveArgument(

MethodParameter parameter, ModelAndViewContainer mavContainer,

NativeWebRequest request, WebDataBinderFactory binderFactory)

**throws** Exception {

String name = ModelFactory.*getNameForParameter*(parameter);

Object attribute = (mavContainer.containsAttribute(name)) ?

mavContainer.getModel().get(name) : createAttribute(name, parameter, binderFactory, request);

//创建数据绑定器，数据绑定器中有完成数据校验+数据转换+数据格式化的组件

WebDataBinder binder = binderFactory.createBinder(request, attribute, name);

**if** (binder.getTarget() != **null**) {

bindRequestParameters(binder, request);

validateIfApplicable(binder, parameter);

**if** (binder.getBindingResult().hasErrors()) {

**if** (isBindExceptionRequired(binder, parameter)) {

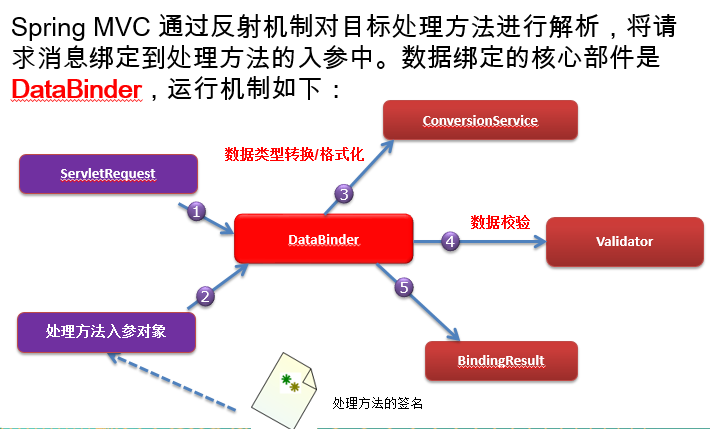
**throw** **new** BindException(binder.getBindingResult());

}

}

}

流程图解：



## 3. 自定义类型转换器

### 3.1 创建converter



实现convert接口来完成数据的转换！



示例：

**public** **class** MyConverter **implements** Converter<String, List<Employee>>{

@Autowired

**private** DepartmentDao deptDao;

@Override

**public** List<Employee> convert(String source) {

List<Employee> employees=**new** ArrayList<Employee>();

//Mayun-mayun@alibaba.com-1-101;LeiJun-leijun@Mi.com-1-102;DongMingZhu-LaydDong@Green.com-0-103;

**if** (source !=**null** && source !="") {

String[] emps = source.split(";");

**for** (String emp : emps) {

String[] props = emp.split("-");

**int** gender = Integer.*parseInt*(props[2]);

**int** deptId = Integer.*parseInt*(props[3]);

Employee employee = **new** Employee(**null**, props[0], props[1], gender, deptDao.getDepartment(deptId));

employees.add(employee);

}

}

**return** employees;

}

### 3.2 创建conversionService

FormattingConversionServiceFactoryBean即可以实现数据格式化又可以实现数据转换

<!-- 配置一个conversionService -->

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

<property name=*"converters"*>

<set>

<bean class=*"com.atguigu.springmvc.converter.MyConverter"*/>

</set>

</property>

</bean>

### 3.3 注册conversionService



## 4.数据格式化

### 4.1 日期格式化



规定页面提交的日期格式！

### 4.2 数字格式化



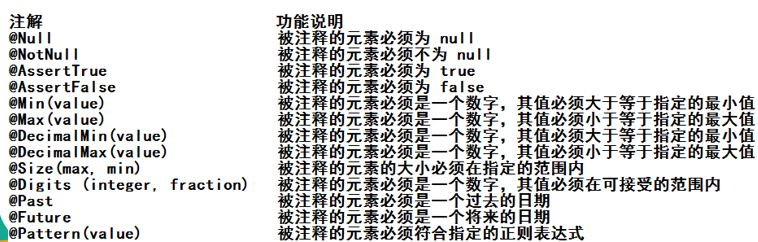
支持页面提交的数字格式！

## 5.数据校验

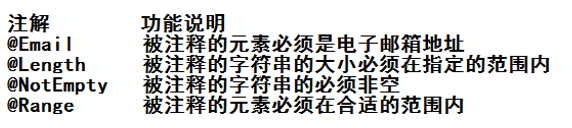
### 5.1 概念

数据校验：我们通过一种标准来校验，JSR303，代表Java的一种标准，校验标准，规范！

JSR303为我们提供了很多注解，来完成数据的校验



Hibernate Validator是一个校验框架，是一个对JSR303的参考实现，而且，除了实现之外，还扩展了以下注解:



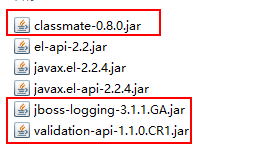
### 5.2 使用

①导包

核心包：

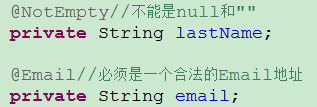


依赖的包：除去带el的！



②需要校验的字段标上注解

如：



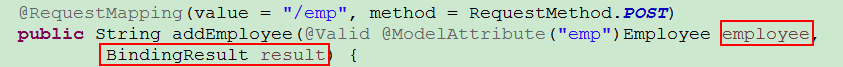
③在需要校验的POJO上，标上@Valid注解



## 6. 错误消息的获取

### 6.1 错误消息的后台获取

错误消息被封装在BindingResult对象中，注意，只会封装紧跟着的POJO的错误信息！



遍历错误：

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

**public** String addEmployee(@Valid @ModelAttribute("emp")Employee employee,

BindingResult result) {

**if** (result.hasErrors()) {

System.***out***.println("出错了！");

List<FieldError> errors = result.getFieldErrors();

**for** (FieldError fieldError : errors) {

System.***out***.println("出错的字段是：" + fieldError.getField()

+ "出现的错误信息：" + fieldError.getDefaultMessage());

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

}

//BindingResult里面保存的错误信息，会全部放入到request域中

**return** "add";

} **else** {

System.***out***.println("即将添加的员工信息：" + employee);

empDao.save(employee);

**return** "redirect:/emps";

}

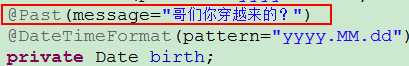
}

### 6.2 错误消息的前台回显

使用表单标签进行回显：<form:errors path="lastName"/>

### 6.3 自定义错误消息

还可以使用校验规则上面的message属性来自定义错误提示消息，但是不能国际化！



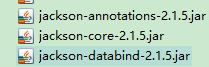
# 十二、处理Json

前台客户端，web客户端，安卓APP，IOSAPP；各种各样，但是与后台的通信方式都是Json！

## 1.使用@ResponseBody返回Json

将方法的返回值直接作为响应体返回！

①导包：



②后台：

@ResponseBody

@RequestMapping("/testJson")

**public** Collection<Employee> handleJson(){

Collection<Employee> all = empDao.getAll();

**return** all;

}

③前台提交ajax请求

$("#testJson").click(**function**(){

**var** url=$(**this**).attr("href");

$.ajax({

url:url,

type:"GET",

success:**function**(data){

console.log(data);

}

});

//取消跳转

**return** **false**;

});

## 2.使用@RequestBody将Json转换为POJO

①后台

@RequestMapping(value="/testJson4")

**public** String handleJson3(@RequestBody()Employee employee){

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

**return** "success";

}

②前台

$("#testSendJson").click(**function**(){

**var** url=$(**this**).attr("href");

//准备一个Json对象

**var** emp={lastName:"Jack",email:"JackMa@Alibaba.com",gender:1};

alert(**typeof** emp);

**var** empStr=JSON.stringify(emp);

alert(**typeof** empStr)

//data数据必须是一个JSON字符串

$.ajax({

url:url,

type:"POST",

data:empStr,

contentType:"application/json",

success:**function**(){

alert("success");

}

});

**return** **false**;

});

# 十三、文件上传和下载

## 1.ResponseEntity实现下载

ResponseEntity：响应报文的实体类，包含响应头和响应体！

原来的@ResponseBody只包含响应体！

思路：①将要下载的内容的流当作响应体返回给浏览器；

②设置响应头信息Content-disposition", "attachment;filename=xxx "告诉浏览器这是一个附件，需要下载！

@RequestMapping("/testDownLoad")

**public** ResponseEntity<**byte** []> handle2(HttpSession session) **throws** IOException{

ServletContext sc = session.getServletContext();

InputStream is = sc.getResourceAsStream("/image/helloworld.png");

**byte** [] result=**new** **byte**[is.available()];

is.read(result);

//下载的要点是在响应头上设置Content-disposition: attachment;filename=xxx

HttpHeaders headers = **new** HttpHeaders();

headers.add("Content-disposition", "attachment;filename=helloworld.png");

ResponseEntity<**byte**[]> responseEntity = **new** ResponseEntity<**byte** []>(result, headers, HttpStatus.***OK***);

**return** responseEntity;

}

## 2.文件上传

Spring MVC 为文件上传提供了直接的支持，这种支持是通过即插即用的 **MultipartResolver** 实现的。Spring 用 **Jakarta Commons FileUpload** 技术实现了一个 MultipartResolver 实现类：**CommonsMultipartResovler**

Spring MVC 上下文中默认没有装配 MultipartResovler，因此默认情况下不能处理文件的上传工作，如果想使用 Spring 的文件上传功能，需现在上下文中配置 MultipartResolver。

①页面的表单中enctype属性必须为 multipart/form-data

<form action=*"*${ctp}*/testFileUpload"* method=*"post"* enctype=*"multipart/form-data"*>

文件1：<input name=*"file"* type=*"file"*><br>

文件2：<input name=*"file"* type=*"file"*><br>

文件3：<input name=*"file"* type=*"file"*><br>

<input type=*"submit"* value=*"testFileUpload"*>

</form>

②导包：



③配置多部件解析器：CommonsMultiPartResolver,id必须是multipartResolver

<!--配置多部件解析器，id是固定的 -->

<bean id=*"multipartResolver"* class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>

<!-- 编码，要和jsp页面的编码一致！ -->

<property name=*"defaultEncoding"* value=*"UTF-8"*/>

<!-- #{}:SPEL表达式 -->

<property name=*"maxUploadSize"* value=*"#{1024\*1024\*10}"*/>

</bean>

注意：多部件解析器的id必须为multipartResolver,否则会报错java.lang.IllegalArgumentException: Expected MultipartHttpServletRequest: is a MultipartResolver configured?

原因是SpringMVC在初始化时，会默认从容器中读取指定id的多部件解析器！



④上传示例：

@RequestMapping("/testFileUpload")

**public** String hello3(@RequestParam("file")MultipartFile [] files,HttpSession session) **throws** IllegalStateException, IOException{

//获取上传位置的真实路径

ServletContext sc = session.getServletContext();

String realPath = sc.getRealPath("/file");

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

**for** (MultipartFile multipartFile : files) {

**if** (!multipartFile.isEmpty()) {

//File.separator:根据系统环境动态生成分隔符

multipartFile.transferTo(**new** File(realPath+File.***separator***+multipartFile.getOriginalFilename()))

}

}

}

**return** "success";

}

# 十四、拦截器

## 1. 概述

拦截器是一个接口，类似 Filter，功能比Filter更加灵活。可以在我们请求方法执行的前后进行拦截，做一些额外的操作！



方法说明：

preHandle():在处理请求的方法之前执行！返回boolean值，意思是是否放行！

postHandle()：在处理请求的方法之后执行！

afterCompletion():在整个请求完成之后，也就是来到目标页面后，执行！

拦截器和Filter功能类似，但是有个本质的区别！即Filter 是WEB三大组件，由Tomcat创建！而拦截器是由Spring容器创建和管理，可以自动装配容器中的所有资源。

## 2. 实现

实现HandlerInterceptor接口

在配置文件中注册：

<bean id=*"msi"* class=*"com.atguigu.springmvc.utils.MySecondInterceptor"*/>

<!-- 配置拦截器 -->

<mvc:interceptors>

<!--bean:配置拦截器，如果直接放在 mvc:interceptors中，意味着拦截所有请求-->

<!-- <bean class="com.atguigu.springmvc.utils.MyFirstInterceptor"/> -->

<!--拦截所有请求 -->

<bean class=*"org.springframework.web.servlet.i18n.LocaleChangeInterceptor"*>

<property name=*"paramName"* value=*"lo"*/>

</bean>

<mvc:interceptor>

<!-- 拦截指定请求 -->

<mvc:mapping path=*"/testInterceptor"*/>

<bean class=*"com.atguigu.springmvc.utils.MyFirstInterceptor"*/>

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path=*"/testInterceptor"*/>

<ref bean=*"msi"*/>

</mvc:interceptor>

</mvc:interceptors>

## 3.拦截器的执行顺序

正常流程： (preHandle返回true)preHandle()---handle()---postHandle()---页面--- afterCompletion

特殊情况：preHandle返回false，后续流程都没有！

结论：

①只要preHandle()方法返回true,那么这个拦截器的afterCompletion都会执行！

②方法的执行顺序：

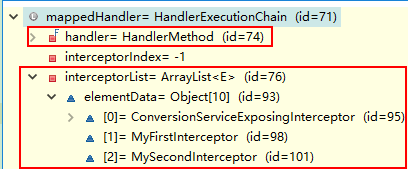
preHandle() :顺序执行；

postHandle():逆序执行；

afterCompletion():逆序执行；

## 4. 源码（了解）

### 4.1 HandlerExcutionChain



HandlerExcutionChain既包含当前请求的处理器，也包含拦截当前请求的拦截器！

### 4.2 preHandle()

**boolean** applyPreHandle(HttpServletRequest request, HttpServletResponse response) **throws** Exception {

//拿到所有的拦截器

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

//按照顺序循环遍历

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

HandlerInterceptor interceptor = getInterceptors()[i];

//执行preHandle()，只要返回false，就进入if语句块

**if** (!interceptor.preHandle(request, response, **this**.handler)) {

triggerAfterCompletion(request, response, **null**);

**return** **false**;

}

//等记下 preHandle（）放回true情况的一个索引信息

**this**.interceptorIndex = i;

}

}

**return** **true**;

}

一旦有一个拦截器，返回false,直接跳出方法！就不会有以下的流程！

**if** (!mappedHandler.applyPreHandle(processedRequest, response)) {

**return**;

}

### 4.3 postHandle()

**void** applyPostHandle(HttpServletRequest request, HttpServletResponse response, ModelAndView mv) **throws** Exception {

**if** (getInterceptors() == **null**) {

**return**;

}

//postHandle()逆序执行！

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

HandlerInterceptor interceptor = getInterceptors()[i];

interceptor.postHandle(request, response, **this**.handler, mv);

}

}

### 4.4 afterCompletion()

**void** triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, Exception ex)

**throws** Exception {

**if** (getInterceptors() == **null**) {

**return**;

}

//只有preHandle()方法返回true, interceptorIndex才会大于0，才会逆序执行afterCompletion

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

HandlerInterceptor interceptor = getInterceptors()[i];

**try** {

interceptor.afterCompletion(request, response, **this**.handler, ex);

}

**catch** (Throwable ex2) {

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

}

}

}

# 十五、运行流程

1. 核心是DispatcherSevlet,负责拦截所有的请求！

2. HandlerMapping,负责根据当前的请求，解析到可以处理当前请求的Handler处理器,拦截器。将处理器和拦截器包装为HandlerExcutionChain(处理器执行链)

3. 根据当前的处理器，得到一个执行处理器方法的适配器HandlerAdapter

4. 如果配置了拦截器，则调用拦截器的preHandle()

5. 执行处理请求的方法

①先执行标注了@ModelAttibute注解的方法

②执行目标方法

如何解析POJO：

a)会从隐含模型中，根据Key取出POJO的值；

key: 如果POJO前面标注了@ModelAttibute注解，且这个注解的value属性有值，key就是value;

否则，key都是类名首字母小写；

b)如果key在隐含模型中找不到，就判断是否标注了@SessionAttibute注解；

如果标注了，从这个注解的value属性中找；

如果有，就尝试从session域取。

取出的值不为空，就赋值。否则报错！

c)前两步都无法获取，就反射创建一个新的对象！

6. 执行拦截器的postHandle()

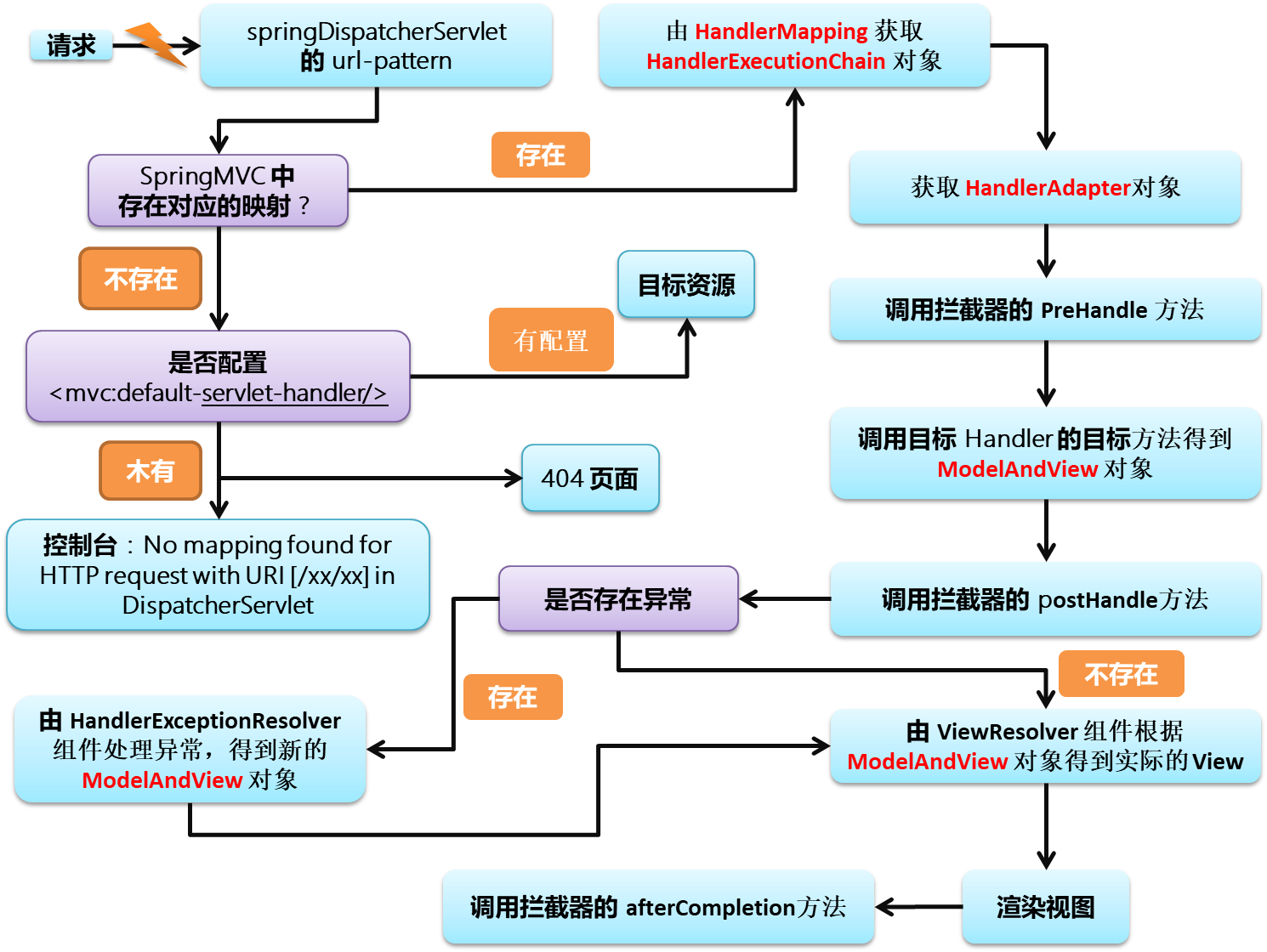
7. 渲染页面

①在渲染页面之前，解析异常

②调用视图解析器，解析得到视图对象；

③调用视图对象的render()

8. 执行拦截器的afterCompletion();



# 十六、整合Spring

1. why

首先在功能上，SpringMVC的配置文件也具有容器的作用。因此不需要分容器。其次，即便是分容器后，也可以在配置文件中使用<import>标签，来导入配置文件。

但是，从项目管理及分层解耦的角度来看。我们一般在SpringMVC的配置文件中配置的是web开发中控制器的相关内容，而在Spring的容器中，会配置数据源、事务管理器等服务层和数据访问层的组件。

所以在SpringMVC中整合Spring的核心就是将资源分容器管理！

原则：

①SpringMVC的容器只负责创建Handler;

②Spring的容器，负责创建其余的资源！

两个配置文件，会导致扫描到的bean被重复创建，因此我们可以通过自定义过滤规则来实现：

|  |
| --- |
| **springmvc.xml** |
| <context:component-scan base-package="com.atguigu.springmvc" use-default-filters="false">  <**context:include-filter** type="annotation"  expression="org.springframework.stereotype.Controller"/>  <**context:include-filter** type="annotation"  expression="org.springframework.web.bind.annotation.ControllerAdvice"/>  </context:component-scan> |

|  |
| --- |
| **beans.xml** |
| <context:component-scan base-package="com.atguigu.springmvc">  <**context:exclude-filter** type="annotation"  expression="org.springframework.stereotype.Controller"/>  <**context:exclude-filter** type="annotation"  expression="org.springframework.web.bind.annotation.ControllerAdvice"/>  </context:component-scan>  <!-- 配置数据源, 整合其他框架, 事务等. --> |

SpringMVC 的 IOC 容器中的 bean 可以来引用 Spring IOC 容器中的 bean. 返回来呢 ? 反之则不行. Spring IOC 容器中的 bean 却不能来引用 SpringMVC IOC 容器中的 bean

* 在 Spring MVC 配置文件中引用业务层的 Bean
* 多个 Spring IOC 容器之间可以设置为父子关系，以实现良好的解耦。
* Spring MVC WEB 层容器可作为 “业务层” Spring 容器的子容器：

即 WEB 层容器可以引用业务层容器的 Bean，而业务层容器却访问不到 WEB 层容器的 Bean

计算机生成了可选文字:
