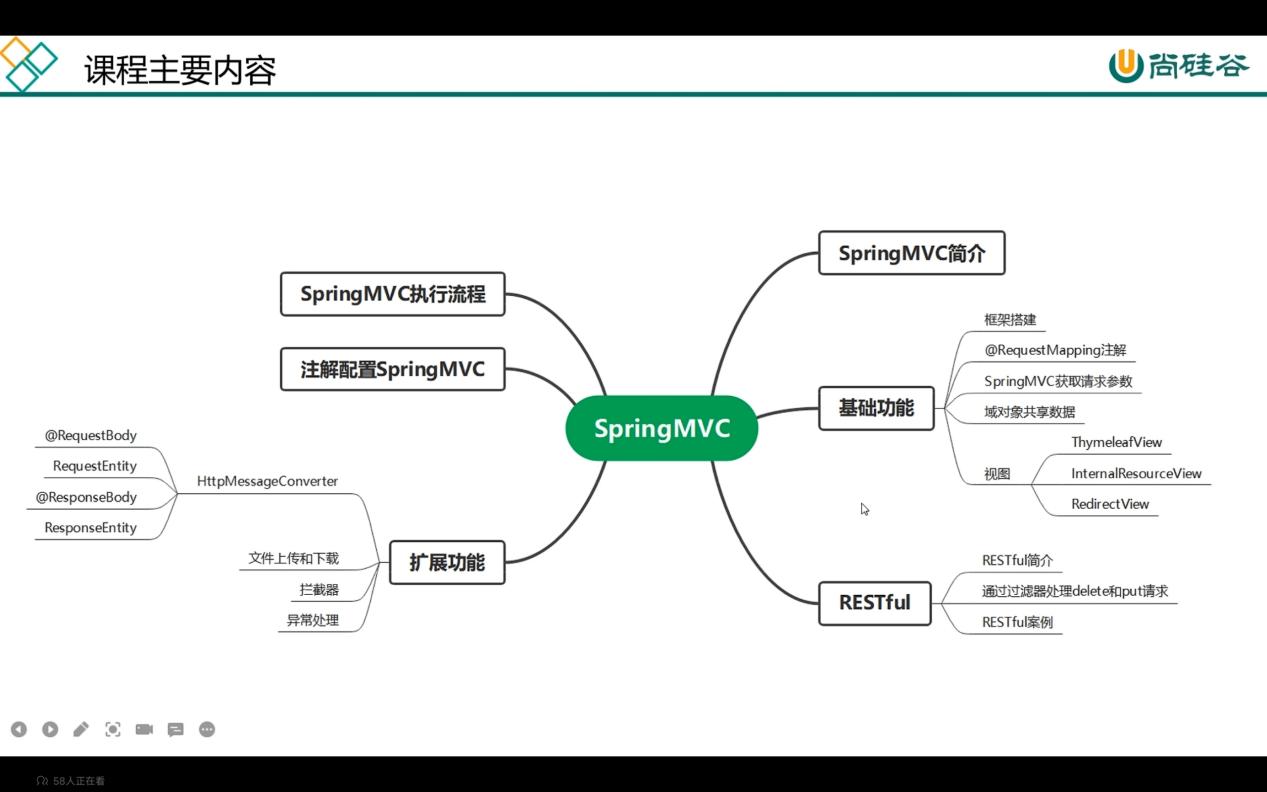
Spring MVC

知识结构：



Spring MVC简介：（Spring的web模块）

基本概念：

MVC是什么：一种软件架构，将软件按照模型，视图，控制器来划分

1. -模型层，指的是javabean（实体类Bean，业务处理Bean），作用是处理数据

V-view层，视图层，指工程中的html，jsp页面，作用是与用户进行交互，展示数据

C--Controller 控制层，指工程中的servlet，作用是请求和响应浏览器

MVC的工作流程：

用户通过view视图层将请求发送到服务器，在服务器中被Controller接收，调用相应的Model模型层处理请求，将处理结果返回给Controller，在将结果传到view层。

什么是Spring MVC

Spring为表示层开发提供的一套完备的解决方案

三层架构（表示层，业务逻辑层，数据访问层）

Spring MVC的优点：

能够与Spring在整合时，无缝衔接

代码清新简介，大幅度提升开发效率，性能卓越

基于原生Servlet，封装了Servlet，通过前端控制器DispatcherServlet对响应和请求处理

注意：Tomcat中内置了servlet api 和jsp，为不发生冲突，使用

<scope>provided</scope>标签

第一个Spring MVC项目建立过程：

先在pom.xml文件中设置打包方式：<packaging>war</packaging>（必要的，否则后边的配置web无法进行）

**一．先新建一个web项目，两种方式：**

然后配置pom.xml文件：

<dependencies>

<!-- SpringMVC -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.3.1</version> </dependency>

<!-- 日志 -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

<!-- ServletAPI -->

<dependency>

<groupId>javax.servlet</groupId>

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

<version>3.1.0</version>

<!--provided 已被服务器提供-->

<scope>provided</scope>

</dependency>

<!-- Spring5和Thymeleaf整合包 -->

<dependency>

<groupId>org.thymeleaf</groupId>

<artifactId>thymeleaf-spring5</artifactId>

<version>3.0.12.RELEASE</version>

</dependency>

</dependencies>

**二．然后配置web.xml:**

两种方式：

1.默认配置：SpringMVC的配置文件默认位于WEB-INF下，默认名称为<servlet-name>-

servlet.xml

<!--配置Spring MVC 的前端控制器，对浏览器发送的请求进行统一处理-->

<servlet>

<servlet-name>SpringMVC</servlet-name>

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

</servlet>

<servlet-mapping>

<servlet-name>SpringMVC</servlet-name>

<!-- / 表示所匹配的请求可以是/login或.html或.js或.css方式的请求路径 但是/不能匹配.jsp请求路径的请求

/\* 表示所匹配的请求可以是/login或.html或.js或.css方式或者也可以是.jsp请求

-->

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

</servlet-mapping>

2.扩展配置方式

通过两个标签 ：<init-param>来配置配置文件的路径和名字，然后将名字作为新建在resources文件夹下的配置文件的名字

<!-- 配置SpringMVC的前端控制器，对浏览器发送的请求统一进行处理 -->

<servlet>

<servlet-name>springMVC</servlet-name>

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

<!-- 通过初始化参数指定SpringMVC配置文件的位置和名称 -->

<init-param>

<!-- contextConfigLocation为固定值 -->

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

<!-- 使用classpath:表示从类路径查找配置文件，例如maven工程中的 src/main/resources -->

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

</init-param>

<!--作为框架的核心组件，在启动过程中有大量的初始化操作要做 而这些操作放在第一次请求时才执行会严重影响访问速度

因此需要通过此标签将启动控制DispatcherServlet的初始化时间提前到服务器启动时

-->

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

</servlet>

<servlet-mapping>

<servlet-name>springMVC</servlet-name>

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

</servlet-mapping>

1. **创建请求控制器**

前端控制器对浏览器发送的请求进行了统一的处理，但是具体的请求有不同的处理过程，因此需要创建处理具体请求的类，即请求控制器.

请求控制器中每一个处理请求的方法成为控制器方法

因为SpringMVC的控制器由一个POJO（普通的Java类）担任，因此需要通过@Controller注解将其标识

为一个控制层组件，交给Spring的IoC容器管理，此时SpringMVC才能够识别控制器的存在

新建一个类：

添加@Controller注解，标识该类为IOC容器内的一个组件，即控制器，其中的方法为控制方法



1. **创建Spring MVC 的配置文件**

视图解析器：控制页面跳转

这里的通过前缀后缀就是能够确定要解析的文件位置和属性，比如下面的配置文件就把定位在/WEB-INF/templates/下的.html文件进行解析。

然后要把.jsp文件解析，就要修改视图前缀前边的bean标签的class的内容，选择合适的class

<!-- 配置Thymeleaf视图解析器 -->

<bean id="viewResolver" class="org.thymeleaf.spring5.view.ThymeleafViewResolver">

<property name="order" value="1"/>

<property name="characterEncoding" value="UTF-8"/>

<property name="templateEngine">

<bean class="org.thymeleaf.spring5.SpringTemplateEngine">

<property name="templateResolver">

<bean class="org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver">

<!-- 视图前缀 -->

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

<!-- 视图后缀 -->

<property name="suffix" value=".html"/>

<property name="templateMode" value="HTML5"/>

<property name="characterEncoding" value="UTF-8" />

</bean>

</property>

</bean>

</property>

</bean>

<!--处理静态资源，例如html、js、css、jpg 若只设置该标签，则只能访问静态资源，

其他请求则无法访问 此时必须设置<mvc:annotation-driven/>解决问题 -->

<mvc:default-servlet-handler/>

<!-- 开启mvc注解驱动 -->

<mvc:annotation-driven>

<mvc:message-converters>

<!-- 处理响应中文内容乱码 -->

<bean class="org.springframework.http.converter.StringHttpMessageConverter">

<property name="defaultCharset" value="UTF-8" />

<property name="supportedMediaTypes">

<list>

<value>text/html</value>

<value>application/json</value>

</list>

</property>

</bean>

</mvc:message-converters>

</mvc:annotation-driven>

**然后在新建的.html文件中配置：thymeleaf的命名空间**

**<html lang="en" xmlns:th="http://www.thymeleaf.org">**

1. **在之前建立好的控制器类里面写**

**控制方法，返回要解析的文件名称给视图解析器，然后配置Tomcat服务器进行调试**

@Controller //通过注解标识该类为一个控制器

public class HelloController {

//请求路径是“/”,访问的页面是 /WEB-INF/templates/index.html

@RequestMapping(value="/") //请求映射，将请求和关系创建映射，当请求是 /时，执行下面这个方法，返回一个“index”给视图解析器

public String index(){

//返回视图名称,被视图解析器所解析

return "index";

}

}

**加深一下：在index中跳转到另一个路径**

**当要解析一个/target路径的文件时：**

**<!--th标签和后边的写法用来解析，当里面是绝对路径时，thymeleaf自动帮我们添加上下文路径-->**

**<a th:href="@{/target}">访问目标页面targe.html</a>**

**然后写控制器方法，映射请求为/target，然后部署Tomcat测试**

**总结：**

**Spring MVC创建第一个案例：**

**1.首先建一个web项目，然后配置pom.xml 引入MVC和thymeleaf视图解析器的路径。**

**2.然后配置web.xml文件，两种方法，一般用第二种-扩展配置，<init-param>来配置Spring MVC配置文件的路径和名字，然后将名字作为新建在resources文件夹下的Spring MVC配置文件的名字（实际上就是在web.xml配置文件中配置一个新的Spring MVC配置文件，设置他的路径在main下的resources下）**

**3.然后创建控制器，也就是一个类，使用注解方式标识@Controller（这个时候就要在第二步的Spring MVC配置文件下配置开启扫描注解）**

**4.配置第二步建立的Spring MVC配置文件，在其中配置thymeleaf视图解析器**

**5.新建.html /.jsp文件，然后部署Tomcat服务器，进行测试**

**关键是通过控制类里面的控制方法，如果返回一个.html文件的名称，这个名称就会通过方法前的注解@RequestMapping（“/”）将请求和关系创建映射，当请求是 /时，执行下面这个方法，返回一个“index”给视图解析器**

**Thymeleaf视图解析器的作用就是接收控制器的控制方法传来的名称，为它添加**

**上下文路径**

RequestMapping注解

请求映射，将请求和处理请求的控制器方法关联起来，建立映射关系

SpringMVC接受到指定的请求，就会找到映射关系中对应的控制器方法来处理这个请求

RequestMapping注解中的值必须是唯一的，否则会报错，两个控制器方法同时处理一个请求

@RequestMapping标识一个类：设置映射请求的请求路径的初始信息，不同模块添加一层路径

@RequestMapping标识一个方法：设置映射请求请求路径的具体信息

举例：

当控制器类和方法之前都使用RequestMapping注解时，在写当前路径时，必须把类中的value也写上

<a th:href="@{/hello/testRequestMapping}">测试RequestMapping注解的位置</a></br>

控制器类和方法：

@Controller

@RequestMapping("/hello")

public class ReauestMappingtest {

@RequestMapping("/testRequestMapping")

public String test1(){

return "success";

}

}

Alt+7 查看属性

Value属性：

@RequestMapping注解的value属性通过请求的请求地址匹配请求映射

@RequestMapping注解的value属性是一个字符串类型的数组，表示该请求映射能够匹配多个请求地址所对应的请求

@RequestMapping注解的value属性必须设置，至少通过请求地址匹配请求映射

Method属性：

请求方式匹配请求（get，post）

可以设置多个值，String类型，有固定值

method={RequestMethod.GET,RequestMethod.POST}

对于处理指定请求方式的控制器方法，SpringMVC中提供了@RequestMapping的派生注解：

处理get请求的映射-->@GetMapping

处理post请求的映射-->@PostMapping

处理put请求的映射-->@PutMapping

处理delete请求的映射-->@DeleteMapping

常用的请求方式有get，post，put，delete，但是目前浏览器只支持get和post，若在form表单提交时，为method设置了其他请求方式的字符

串（put或delete），则按照默认的请求方式get处理

若要发送put和delete请求，则需要通过spring提供的过滤器HiddenHttpMethodFilter，在RESTful部分会讲到

Params属性：

参数请求，字符串数组类型

"param"：要求请求映射所匹配的请求必须携带param请求参数

"!param"：要求请求映射所匹配的请求必须不能携带param请求参数

"param=value"：要求请求映射所匹配的请求必须携带param请求参数且param=value

"param!=value"：要求请求映射所匹配的请求必须携带param请求参数但是param!=value

Headers属性：

http报头属性

Spring MVC支持路径中的占位符

原始方式：/deleteUesr?id=1

rest方式：/deleteUser/1

例子：

@RequestMapping("/testPath/{id}/{username}")

public String testPath(@PathVariable("id")Integer id,@PathVariable("username")String username ){ //这个注解表示将id的值传到形参中

System.out.println("id:"+id+",username:"+username);

return "success";

}

<a th:href="@{/testPath/1/1234}">测试Spring MVC 支持路径中的占位符</a>

**Spring MVC获取请求参数：**

1. 通过servlet API获取

请求被前端处理器处理，与控制器相匹配之后，然后经过一系列组件包括DispatchServlet调用控制器方法

@RequestMapping("/testServletAPI")

//形参的request表示当前请求

public String testServletAPI(HttpServletRequest request){

String username=request.getParameter("username");

String password=request.getParameter("password");

System.out.println("username:"+username+",password:"+password);

return "success";

}

<a th:href="@{/testServletAPI(username='admin',password=123456)}">测试ServletAPI获取请求参数</a>

1. 通过控制器方法的形参获取请求参数

上边的方法太麻烦，因为Spring MVC已经内部封装好了那些原始的Servlet方法获取到的参数。

通过本方法，直接将要获取的参数写在控制器方法的形参上，然后就能直接获取到

@RequestMapping("/testParam")

public String testParam(String username,String password){

System.out.println("username:"+username+",password:"+password);

return "success";

}

<a th:href="@{/testParam(username='admin',password=123456)}">测试控制器的形参获取请求参数</a>

**当出现多个参数同名时，在形参中写一个名字，然后就能获取所有的值**

@RequestMapping("/testParam")

public String testParam(String username,String password,String hobby){

System.out.println("username:"+username+",password:"+password+",hobby:"+hobby);

return "success";

}

<form th:action="@{/testParam}" method="post">

用户名：<input type="text" name="username"><br>

密码：<input type="password" name="password"><br>

爱好：<input type="checkbox" name="hobby" value="a">a<br>

<input type="checkbox" name="hobby" value="b">b<br>

<input type="checkbox" name="hobby" value="c">c<br>

<input type="submit" value="测试使用控制器形参获取请求参数">

</form>

当我们要获取的参数名字写错之后，导致无法获取，解决办法：

使用一个注解来将两个对应起来：

例子：

@RequestMapping("/testParam")

public String testParam(

@RequestParam("user\_name") String username, String password, String hobby){

// @RequestParam("user\_name") String username,这个注解表示将user\_name 和username两个相对应起来

System.out.println("username:"+username+",password:"+password+",hobby:"+hobby);

return "success";

}

<input type="text" name="user\_name"><br>

@RequestHeader注解

与@RequestParams用法相同，将请求头信息和控制器方法的形参创建映射

@CookieValue

将cookie数据和控制器方法的形参创建映射

通过POJO获取请求参数：

就是上边直接将相对应的参数写在控制器类的控制器方法形参时，能直接获取参数的值，但是如果太多就显得冗余，所以直接写一个实体类，实体类里面封装了要获取的参数，直接在控制器方法的形参中写实体类对象，最后直接输出实体类对象就可以了，但是这样获取的参数的值会出现乱码问题

解决乱码问题：

Get请求不会发生乱码，因为是由Tomcat的编码方式导致的，只需修改Tomcat的编码方式就能解决

<Connector port="80" URIEncoding="UTF-8" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" />

由于DIspcherServlet已经获取到了请求数据，所以通过Request.CharacterEncoding()修改不了编码方法，必须在DIspcherServlet获取请求之前设置编码方式

Post提交方式：

设置过滤器：在web.xml中

<filter>

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

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<!--设置编码-->

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

<!--设置响应的编码-->

<init-param>

<param-name>forceResponseEncoding</param-name>

<param-value>true</param-value>

</init-param>

</filter>

<filter-mapping>

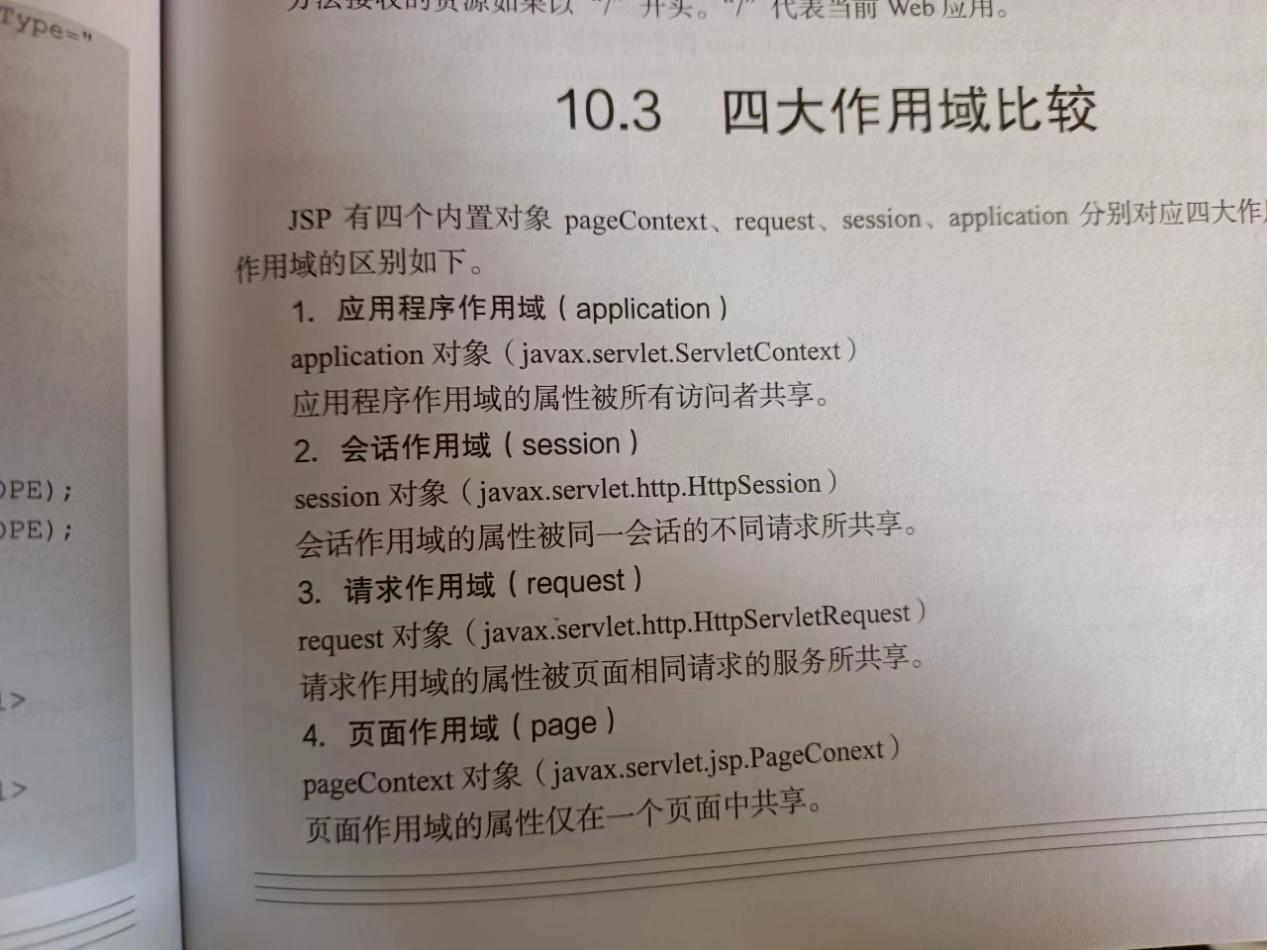
<filter-name>CharacterEncodingFilter</filter-name>

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

</filter-mapping>

域对象共享数据：

四个域对象：



Appication ,session,Request，PageContext

选择域最小的能实现功能的对象

向Request域对象获取数据：

1. 使用ServletAPI来共享Request域对象

request.setAttribute(键，值)方法来获取

//使用ServletAPI向Request域对象共享数据

@RequestMapping("/testRequestByServletAPI")

public String testRequestByServletAPI(HttpServletRequest request){

request.setAttribute("testRequestScope1","hello,servletAPI");

return "success";

}

2）使用ModelAndView向request域对象共享数据

ModelAndView两个作用：

向Request域共享数据

设置视图名称，实现跳转

@RequestMapping("/testModelAndView")

public ModelAndView testModelAndView(){

ModelAndView mav=new ModelAndView();

//处理模型数据，向Request域共享数据

mav.addObject("testRequestScope2","hello,ModelAndView");

//设置视图名称

mav.setViewName("success");

return mav;

}

3）使用Mode向Request域对象共享数据

@RequestMapping("/testModel")

public String testModel(Model model){

model.addAttribute("testRequestScope3","hello,model");

return "success";

}

1. 使用map向Request域对象共享数据

@RequestMapping("/testmap")

public String testMap(Map<String,Object> map){

map.put("testRequestScope","hello,map");

return "success";

}

1. ModeMap向Request域对象共享数据

@RequestMapping("/testModelMap")

public String testModelMap(ModelMap modelMap){

modelMap.addAttribute("testRequestScope","hello,ModelMap");

return "success";

}

1. Model、ModelMap、Map的关系：

也就是BindingAwareModelMap 类实现了上面三个

Model、ModelMap、Map类型的参数其实本质上都是 BindingAwareModelMap 类型的

看源码：

public interface Model{} public class ModelMap extends LinkedHashMap<String, Object> {} public class ExtendedModelMap extends ModelMap implements Model {} public class BindingAwareModelMap extends ExtendedModelMap {}

看源码，上边几种方式先执行控制器方法之后，将获得对象封装为ModeAndView对象，该对象内有view和Model

向session域共享数据

注意了，必须用session.键来获取值，这里与Request不同

<p th:text="${session.testSessionScope}"></p>

@RequestMapping("/testSession")

public String testSession(HttpSession session){

session.setAttribute("testSessionScope","hello,session");

return "success";

}

向application域共享数据

<p th:text="${application.testApplication}"></p>

@RequestMapping("/testApplication")

public String testApplication(HttpSession session){

ServletContext application=session.getServletContext();

application.setAttribute("testApplication","hello,Application");

return "success";

}

Spring MVC的视图

Spring MVC中的视图是View接口，作用是将Model中的数据展示给用户

Spring MVC中的视图种类有多种，默认是转发视图（InternalResourceView）和重定向视图（RedirectView）

这里用到了JSTL

当工程引入jstl的依赖，转发视图会自动转换为JstlView

在SpringMVC的配置文件中配置了Thymeleaf的视图解析器，因此视图解析器解析之后所得到的是ThymeleafView

1. ThymeleafView

当控制器方法中所设置的视图名称没有任何前缀时，此时的视图名称会被SpringMVC配置文件中所配置

的视图解析器解析，视图名称拼接视图前缀和视图后缀所得到的最终路径，会通过转发的方式实现跳转

1. InternalResourceView

当控制器方法中所设置的视图名称以"forward:"为前缀时，创建InternalResourceView视图，此时的视

图名称不会被SpringMVC配置文件中所配置的视图解析器解析，而是会将前缀"forward:"去掉，剩余部

分作为最终路径通过转发的方式实现跳转

@RequestMapping("/testForward")

public String testForward(){

return "forward:/testThymeleafView";

}

返回forward前缀，跳转到一个控制器方法

1. RedirectView

当控制器方法中所设置的视图名称以"redirect:"为前缀时，创建RedirectView视图，此时的视图名称不

会被SpringMVC配置文件中所配置的视图解析器解析，而是会将前缀"redirect:"去掉，剩余部分作为最终路径通过重定向的方式实现跳转

转发和重定向的区别：

1、请求次数

重定向是浏览器向服务器发送一个请求并收到响应后再次向一个新地址发出请求，转发是服务器收到请求后为了完成响应跳转到一个新的地址；重定向至少请求两次，转发请求一次；

2、地址栏不同

重定向地址栏会发生变化，转发地址栏不会发生变化；

3、是否共享数据

重定向两次请求不共享数据，转发一次请求共享数据（在request级别使用信息共享，使用重定向必然出错）；

4、跳转限制

重定向可以跳转到任意URL，转发只能跳转本站点资源；

5、发生行为不同

重定向是客户端行为，转发是服务器端行为

Web INfo下的内容只能通过转发访问到，重定向不行

@RequestMapping("/testRedirect")

public String testRedirect(){

return "redirect:/testThymeleafView";

}

}

视图控制器：

当控制器方法中，仅仅用来实现页面跳转，即只需要设置视图名称时，可以将处理器方法使用viewcontroller标签进行表示

<mvc:view-controller path="/" view-name="index"></mvc:view-controller>

这样之后，前边所有的跳转功能全部不能用，所以再加一个标签

<!--开启mvc注解驱动-->

<mvc:annotation-driven />

这样就可以重新实现请求映射了

配置InternalResourceViewResolve视图解析器来进行测试：（这样配置，没有前缀的当作转发，前缀为redirect的使用重定向）

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

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

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

</bean>

注意这里的跳转路径前边需要这样写，通过pageContext的方法获取路径

<a href="${pageContext.request.contextPath}/success">success.jsp</a>

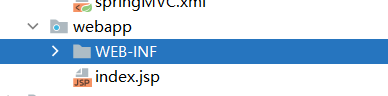
@RequestMapping("/success")

public String success(){

return "success";

}

注意：下面图中的index.jsp为默认打开的页面，放在webapp目录下面



**RESTFUL：**

表现层资源状态转移，资源就是传输的各种格式的文件，状态转移，也就是客户端和服务器端之间进行交互的转移（transfer）代表资源状态的表述。通过转移和操作资源的表述，来间接实现操作资源的目的

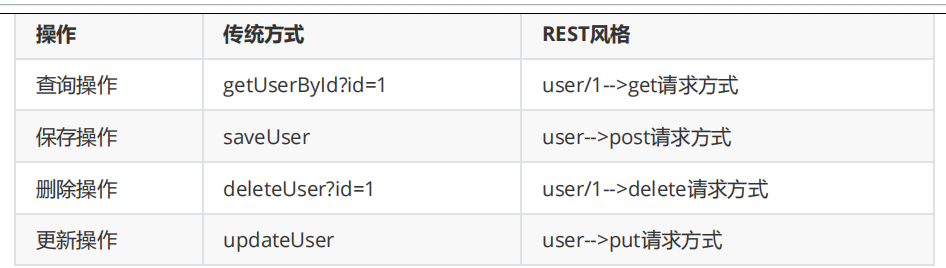
讲白了就是URI，代表这个资源在客户端和服务器端的转移，然后我们要做的就是在进行不同的操作时，将URI的格式特定化

实现：

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

REST 风格提倡 URL 地址使用统一的风格设计，从前到后各个单词使用斜杠分开，不使用问号键值对方式携带请求参数，而是将要发送给服务器的数据作为 URL 地址的一部分



模拟get，post请求：

@Controller

public class UserController {

/\*\*

\* 使用RESTFUL 模拟用的资源的增删改查

\* /user GET 查询所有用户信息

\* /user/1 GET 根据用户ID查询所有用户信息

\* /user post 添加用户信息

\* /user/1 DELETE 删除用户信息

\* /user PUT 更新用户信息

\*/

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

public String getAlluers(){

System.out.println("查询所有用户信息");

return "success";

}

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

public String getUserByid(){

System.out.println("查询id=1的用户信息");

return "success";

}

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

public String insertUser(String username,String password){

System.out.println("添加用户信息："+username+","+password);

return "success";

}

}

<a th:href="@{/user}">查询所有用户信息</a><br>

<a th:href="@{/user/1}">查询id为1用户信息</a><br>

<form th:action="@{/user}" method="post">

用户名： <input type="text" name="username"><br>

密码： <input type="password" name="password"><br>

<input type="submit" name="添加"><br>

</form>

由于浏览器只支持发送get和post方式的请求，那么该如何发送put和delete请求呢？

SpringMVC 提供了 HiddenHttpMethodFilter 帮助我们将 POST 请求转换为 DELETE 或 PUT 请求

首先需要在web.xml中配置HiddenHttpMethodFilter

<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>

打开源码查看，

HiddenHttpMethodFilter 处理put和delete请求的条件：

a>当前请求的请求方式必须为post

b>当前请求必须传输请求参数\_method

例子：

<form th:action="@{/user}" method="post">

<input type="hidden" name="\_method" value="PUT">

用户名： <input type="text" name="username"><br>

密码： <input type="password" name="password"><br>

<input type="submit" name="修改" value="修改"><br>

</form>

@RequestMapping(value="/user" ,method= RequestMethod.PUT)

public String updateUser(String username,String password){

System.out.println("修改用户信息："+username+","+password);

return "success";

}

当下我们的配置文件中有两个过滤器，Filter，我们考虑一下两个过滤器的配置顺序，因为设置编码之前不能获取任何请求参数，否则编码没作用，因此将编码过滤器放在前边

RESTFUL案例实现：

增删改查操作的实现

Spring MVC处理静态资源的过程

首先我们已经配置了一个前端控制器，每次请求都会处理之后，找Controller方法，返回结果，但是springMVC控制器方法中没有对静态资源的访问，所以采用默认的servlet来进行访问处理

两个注解：

<!--开放对静态资源的访问，默认的Servlet来处理-->

<mvc:default-servlet-handler />

<!--开启mvc注解驱动，由前端控制器处理-->

<mvc:annotation-driven />

HttpMessageConverter：

报文信息转换器，将请求报文转换为Java对象，或将Java对象转换为响应报文

HttpMessageConverter报文转换器提供了两个注解和两个类型：

@RequestBody，获取请求体

@ResponseBody，

RequestEntity，请求实体，包含请求头，请求体等

ResponseEntity，响应实体

@RequestBody注解：

@RequestMapping ("/testRequestBody")

public String testRequestBody(@RequestBody String requestBody){

System.out.println(requestBody);

return "success";

}

RequestEntity：

@RequestMapping ("/testRequestEntity")

public String testRequestEntity(RequestEntity<String> requestEntity){

//当前requestEntity表示整个请求报文的信息

System.out.println("请求头："+requestEntity.getHeaders());

System.out.println("请求体："+requestEntity.getBody());

return "success";

}

@ResponseBody：

在测试@ResponseBody之前，我们先使用ServletAPI进行一下测试：

@RequestMapping("/testResponse")

public void testResponse(HttpServletResponse response) throws IOException {

response.getWriter().print("hello response"); //展示浏览器数据

}

这样最后跳转的结果就是展示 hello response

然后我们使用注解来测试：

@RequestMapping("/testResponseBody")

@ResponseBody

//加注解之后，跳转的结果是展示浏览器的数据

public String testResponseBody(){

return "success";

}

能实现相同的结果，但简化很多

测试浏览器能不能接受一个对象：

@RequestMapping("/testResponseUser")

@ResponseBody

public User testResponseUser(){

return new User(1001,"admin","123456",23,"男");

}

结果报错。。。。

这里就不得不提出下面问题：

SpringMVC处理json

引入依赖：

<dependency> <groupId>com.fasterxml.jackson.core</groupId> <artifactId>jackson-databind</artifactId> <version>2.12.1</version> </dependency>

引入依赖之后，浏览器的页面中展示的结果：

{"id":1001,"username":"admin","password":"123456","age":23,"sex":"男"}

处理步骤：

@ResponseBody处理json的步骤：

a>导入jackson的依赖

b>在SpringMVC的核心配置文件中开启mvc的注解驱动，此时在HandlerAdaptor中会自动装配一个消息转换器：MappingJackson2HttpMessageConverter，可以将响应到浏览器的Java对象转换为Json格式的字符串

c>在处理器方法上使用@ResponseBody注解进行标识

d>将Java对象直接作为控制器方法的返回值返回，就会自动转换为Json格式的字符串

首先我们要知道json是什么：（我们这里是将java对象解析成json字符串）

JSON: JavaScript Object Notation(JavaScript 对象表示法)，是一种轻量级的数据交换格式，采用完全独立于编程语言的文本格式来存储和表示数据

Json类型：json对象，json数组

处理Ajax：

@RequestMapping("/testAxios")

@ResponseBody

public String testAxios(String username,String password){

System.out.println("username:"+username+",password:"+password);

return "hello,axios";

}

<div id="app">

<a @click="testAxios" th:href="@{/testAxios}">SpringMVC处理ajax</a>

</div>

<script type="text/javascript" th:src="@{/static/js/vue.js}"></script>

<script type="text/javascript" th:src="@{/static/js/axios.min.js}"></script>

<script type="text/javascript">

new Vue({

el:"#app",

methods:{

testAxios:function (event) {

axios({

method:"post",

url:event.target.href,

params:{

username:"admin",

password:"123456"

}

}).then(function (response) {

alert(response.data);

});

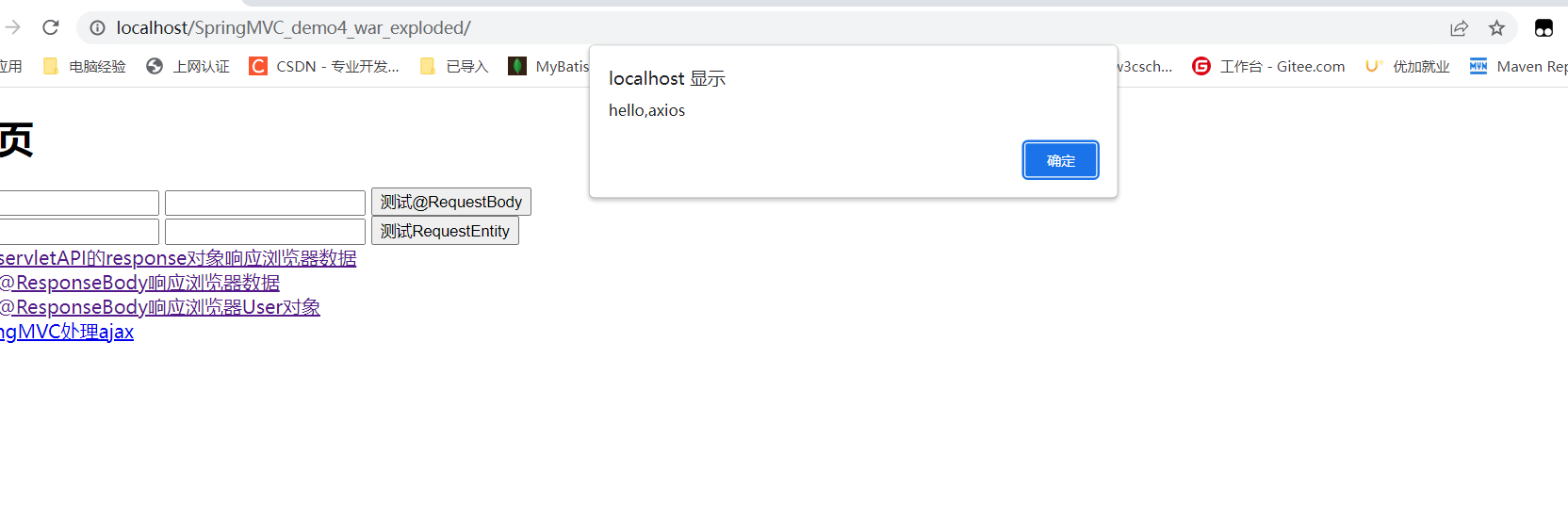
event.preventDefault();

}

}

});

</script>



@RestController注解

@RestController注解是springMVC提供的一个复合注解，标识在控制器的类上，就相当于为类添加了@Controller注解，并且为其中的每个方法添加了@ResponseBody注解

ResponseEntity:

ResponseEntity用于控制器方法的返回值类型，该控制器方法的返回值就是响应到浏览器的响应报文

作用是实现文件下载功能。。

下面开始实现文件的下载实现：

@RequestMapping("/testDown")

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

//获取ServletContext对象

ServletContext servletContext = session.getServletContext();

//获取服务器中文件的真实路径

String realPath = servletContext.getRealPath("/static/img/1.jpg");

System.out.println(realPath);

//创建输入流

InputStream is = new FileInputStream(realPath);

//创建字节数组

byte[] bytes = new byte[is.available()];

//将流读到字节数组中

is.read(bytes);

//创建HttpHeaders对象设置响应头信息

MultiValueMap<String, String> headers = new HttpHeaders();

//设置要下载方式以及下载文件的名字

headers.add("Content-Disposition", "attachment;filename=1.jpg");

//设置响应状态码

HttpStatus statusCode = HttpStatus.OK;

//创建ResponseEntity对象

ResponseEntity<byte[]> responseEntity = new ResponseEntity<>(bytes, headers, statusCode);

//关闭输入流

is.close();

return responseEntity;

}

<a th:href="@{/testDown}">下载1.jpg</a>

将准备好的图片放在webapp下的一个目录里，然后照着上边部署之后，点击链接就能实现下载功能。

文件上传功能：

添加依赖：

<!-- https://mvnrepository.com/artifact/commons-fileupload/commons-fileupload -- ><dependency> <groupId>commons-fileupload</groupId> <artifactId>commons-fileupload</artifactId> <version>1.3.1</version> </dependency>

配置文件上传解析器：

下面的id必须写上，不然找不到封装之后的对象，而且id必须写下面的

<!--配置文件上传解析器，将上传的文件封装为MultiPartFile对象-->

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

写上传方式：

<!--首先上传必须用post方式，然后enctype="multipart/form-data" 标签表示将表单的提交以二进制形式上传-->

<form th:action="@{/testUp}" method="post" enctype="multipart/form-data">

头像：<input type="file" name="photo"><br>

<input type="submit" value="上传">

</form>

写控制类：

@RequestMapping("/testUp")

public String testUp(MultipartFile photo,HttpSession session) throws IOException{

String fileName=photo.getOriginalFilename();

ServletContext servletContext=session.getServletContext();

String photoPath=servletContext.getRealPath("photo");

File file=new File(photoPath);

//判断photoPath路径是否存在

if(!file.exists()){

//不存在，创建目录

file.mkdir();

}

String finalPath=photoPath+File.separator+fileName;

photo.transferTo(new File(finalPath)); //转移浏览器上的文件到服务器

return "success";

}

解决文件重名问题：

文件重名，后边上传的覆盖前边的、

解决办法，

UUID 是 通用唯一识别码（Universally Unique Identifier）的缩写，是一种软件建构的标准，亦为开放软件基金会组织在分布式计算环境领域的一部分。其目的，是让分布式系统中的所有元素，都能有唯一的辨识信息，而不需要通过中央控制端来做辨识信息的指定。如此一来，每个人都可以创建不与其它人冲突的UUID。在这样的情况下，就不需考虑数据库创建时的名称重复问题

@RequestMapping("/testUp")

public String testUp(MultipartFile photo,HttpSession session) throws IOException{

//获取上传的文件的文件名

String fileName=photo.getOriginalFilename();

//获取上传文件的后缀名

String suffixName=fileName.substring(fileName.lastIndexOf("."));

//将UUID作为文件名

String uuid= UUID.randomUUID().toString();

//将UUID和后缀名拼接后的结果作为文件名

fileName=uuid+suffixName;

//通过ServletContext获取服务器中photo目录的路径

ServletContext servletContext=session.getServletContext();

String photoPath=servletContext.getRealPath("photo");

File file=new File(photoPath);

//判断photoPath路径是否存在

if(!file.exists()){

//不存在，创建目录

file.mkdir();

}

String finalPath=photoPath+File.separator+fileName;

photo.transferTo(new File(finalPath)); //转移浏览器上的文件到服务器

return "success";

}

拦截器：对控制器方法进行拦截

三个抽象方法：

preHandle：控制器方法执行之前执行preHandle()，其boolean类型的返回值表示是否拦截或放行，返

回true为放行，即调用控制器方法；返回false表示拦截，即不调用控制器方法

postHandle：控制器方法执行之后执行postHandle()

afterComplation：处理完视图和模型数据，渲染视图完毕之后执行afterComplation()

拦截器操作：

1.写拦截器：

//或者extends HandlerInterceptor

public class FirstInterceptor implements HandlerInterceptor {

//重写默认方法，ctrl+o

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

System.out.println("First");

return true;

//返回false表示拦截，true表示放行，看源码

}

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {

System.out.println("post");

}

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {

System.out.println("after");

}

}

2.配置拦截器

1）拦截器方法1

<!--配置拦截器-->

<mvc:interceptors>

<bean class="com.libaba.mvc.controller.interceptors.FirstInterceptor"></bean>

</mvc:interceptors>

这样配置之后，preHandle拦截，preHandle方法返回值为true时，可以放行

然后是对我们当前的所有请求都进行拦截

1. 拦截器配置2

首先用注解标识FirstInterceptor，为一个bean对象，

@Component

//或者extends HandlerInterceptor

public class FirstInterceptor implements HandlerInterceptor {

//重写默认方法，ctrl+o

然后在配置文件中配置：

<!-- <ref bean="firstInterceptor"></ref>-->

<mvc:interceptors>

<ref bean="firstInterceptor"></ref>

</mvc:interceptors>

1. 拦截器配置3

<!--配置拦截器-->

<mvc:interceptors>

<!-- <bean class="com.libaba.mvc.controller.interceptors.FirstInterceptor"></bean>-->

<!-- <ref bean="firstInterceptor"></ref>-->

<mvc:interceptor>

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

<mvc:exclude-mapping path="/"/>

<ref bean="firstInterceptor"></ref>

</mvc:interceptor>

</mvc:interceptors>

解释一下标签：

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

拦截路径，/\*表示拦截上下文路径下一层，/\*\*表示拦截上下文路径下所有

<mvc:exclude-mapping path="/"/>

排除 访问控制器映射 / ，其他请求被拦截

<ref bean="firstInterceptor"></ref>

选拦截器

多个拦截器的执行顺序：

若每个拦截器的preHandle()都返回true

preHandle方法按序执行，

postHandle方法倒序执行

afterCompetion方法倒序执行

若某个拦截器的preHandle()返回了false:

preHandle()返回false和它之前的拦截器的preHandle()都会执行，postHandle()都不执行，返回false

的拦截器之前的拦截器的afterComplation()会执行

异常处理器：

SpringMVC提供了一个处理控制器方法执行过程中所出现的异常的接口：HandlerExceptionResolver

HandlerExceptionResolver接口的实现类有：DefaultHandlerExceptionResolver（默认）和

SimpleMappingExceptionResolver（自定义）

基于配置的异常处理：

使用方式：

<bean

class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<property name="exceptionMappings">

<props>

<!--

properties的键表示处理器方法执行过程中出现的异常

properties的值表示若出现指定异常时，设置一个新的视图名称，跳转到指定页面

-->

<prop key="java.lang.ArithmeticException">error</prop>

</props>

</property>

<!--

exceptionAttribute属性设置一个属性名，将出现的异常信息在请求域中进行共享

-->

<property name="exceptionAttribute" value="ex"></property>

</bean>

测试数学运算异常：

<!--配置异常处理-->

<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<property name="exceptionMappings" >

<props>

<prop key="java.lang.ArithmeticException" >error</prop>

</props>

</property>

</bean>

注意：这里的error指的是我们在Templates文件夹下面建的新文件error.html，必须写对了。

@RequestMapping("/testExceptionHandler")

public String testExceptionHandler(){

System.out.println(1/0);

return "success";

}

如果要把当前异常信息显示在页面里：

<!--配置异常处理-->

<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<property name="exceptionMappings" >

<props>

<prop key="java.lang.ArithmeticException" >error</prop>

</props>

</property>

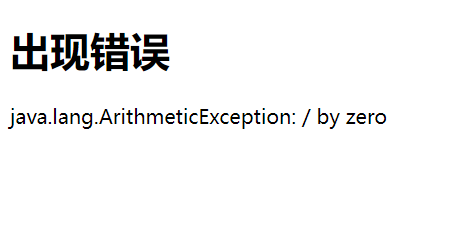
<property name="exceptionAttribute" value="ex"></property>

</bean>

蓝色标注的表示，设置一个键ex，他的值就是当前异常信息

在error.html中打印异常：

<p th:text="${ex}"></p>



基于注解的异常处理：

写异常控制器方法：

@ControllerAdvice

public class ExceptionController {

@ExceptionHandler(value={ArithmeticException.class,NullPointerException.class})

public String testException(Exception e, Model model){

model.addAttribute("e",e);//共享异常信息

return "error";

}

}

用上一个控制器方法的算数异常过程测试，：  
就能得到错误结果。。

注解配置SpringMVC

1. 创建初始化类，代替web.xml

2、创建SpringConfig配置类，代替spring的配置文件

3、创建WebConfig配置类，代替SpringMVC的配置文件

代码见springMVC-annotation

SpringMVC的执行流程

1) 用户向服务器发送请求，请求被SpringMVC 前端控制器 DispatcherServlet捕获。

2) DispatcherServlet对请求URL进行解析，得到请求资源标识符（URI），判断请求URI对应的映射：

a) 不存在

i. 再判断是否配置了mvc:default-servlet-handler

ii. 如果没配置，则控制台报映射查找不到，客户端展示404错误

iii. 如果有配置，则访问目标资源（一般为静态资源，如：JS,CSS,HTML），找不到客户端也会展示404

错误

b) 存在则执行下面的流程

3) 根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain执行链对象的形式返回。

4) DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter。

5) 如果成功获得HandlerAdapter，此时将开始执行拦截器的preHandler(…)方法

6) 提取Request中的模型数据，填充Handler入参，开始执行Handler（Controller)方法，处理请求。

在填充Handler的入参过程中，根据你的配置，Spring将帮你做一些额外的工作：

a) HttpMessageConveter： 将请求消息（如Json、xml等数据）转换成一个对象，将对象转换为指定

的响应信息

b) 数据转换：对请求消息进行数据转换。如String转换成Integer、Double等

c) 数据格式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等

d) 数据验证： 验证数据的有效性（长度、格式等），验证结果存储到BindingResult或Error中

7) Handler执行完成后，向DispatcherServlet 返回一个ModelAndView对象。

8) 此时将开始执行拦截器的postHandle(...)方法【逆向】。

9) 根据返回的ModelAndView（此时会判断是否存在异常：如果存在异常，则执行

HandlerExceptionResolver进行异常处理）选择一个适合的ViewResolver进行视图解析，根据Model

和View，来渲染视图。

10) 渲染视图完毕执行拦截器的afterCompletion(…)方法【逆向】。

11) 将渲染结果返回给客户端。