SSM 框架整合教程：三、SpringMVC——尚硅谷学习笔记 2022 年

三、SpringMVC

1、SpringMVC 简介

1.1、什么是 MVC

MVC 是一种软件架构的思想，将软件按照模型、视图、控制器来划分。

M：Model，模型层，指工程中的 JavaBean，作用是处理数据。

JavaBean 分为两类：

- 一类称为实体类 Bean：专门存储业务数据的，如 Student、User 等。
- 一类称为业务处理 Bean：指 Service 或 Dao 对象，专门用于处理业务逻辑和数据访问。

V：View，视图层，指工程中的 html 或 jsp 等页面，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，作用是接收请求和响应浏览器。

MVC 的工作流程： 用户通过视图层发送请求到服务器，在服务器中请求被 Controller 接收，Controller 调用相应的 Model 层处理请求，处理完毕将结果返回到 Controller，Controller 再根据请求处理的结果找到相应的 View 视图，渲染数据后最终响应给浏览器。

1.2、什么是 SpringMVC

SpringMVC 是 Spring 的一个后续产品，是 Spring 的一个子项目。

SpringMVC 是 Spring 为表述层开发提供的一整套完备的解决方案。在表述层框架历经 Strust、WebWork、Strust2 等诸多产品的历代更迭之后，目前业界普遍选择 SpringMVC 作为 Java EE 项目表述层开发的**首选方案**。

> 注意：三层架构分为表述层（或表示层）、业务逻辑层、数据访问层，表述层表示前台页面和后台 servlet。

1.3、SpringMVC 的特点

- **Spring 家族原生产品**，与 IOC 容器等基础设施无缝对接。
- **基于原生的 Servlet**，通过了功能强大的**前端控制器 DispatcherServlet**，对请求和响应进行统一处理。
- 表述层各细分领域需要解决的问题**全方位覆盖**，提供**全面解决方案**。
- **代码清新简洁**，大幅度提升开发效率。
- 内部组件化程度高，可插拔式组件**即插即用**，想要什么功能配置相应组件即可。
- **性能卓著**，尤其适合现代大型、超大型互联网项目要求。

2、入门案例

2.1、开发环境

IDE：idea 2022.3

构建工具：maven 3.8.1

服务器：tomcat 10.1.13

Spring 版本：6.0.9

2.2、创建 maven 工程

2.2.1 ① 添加 web 模块

2.2.2 ② 打包方式：war

2.2.3 ③ 引入依赖

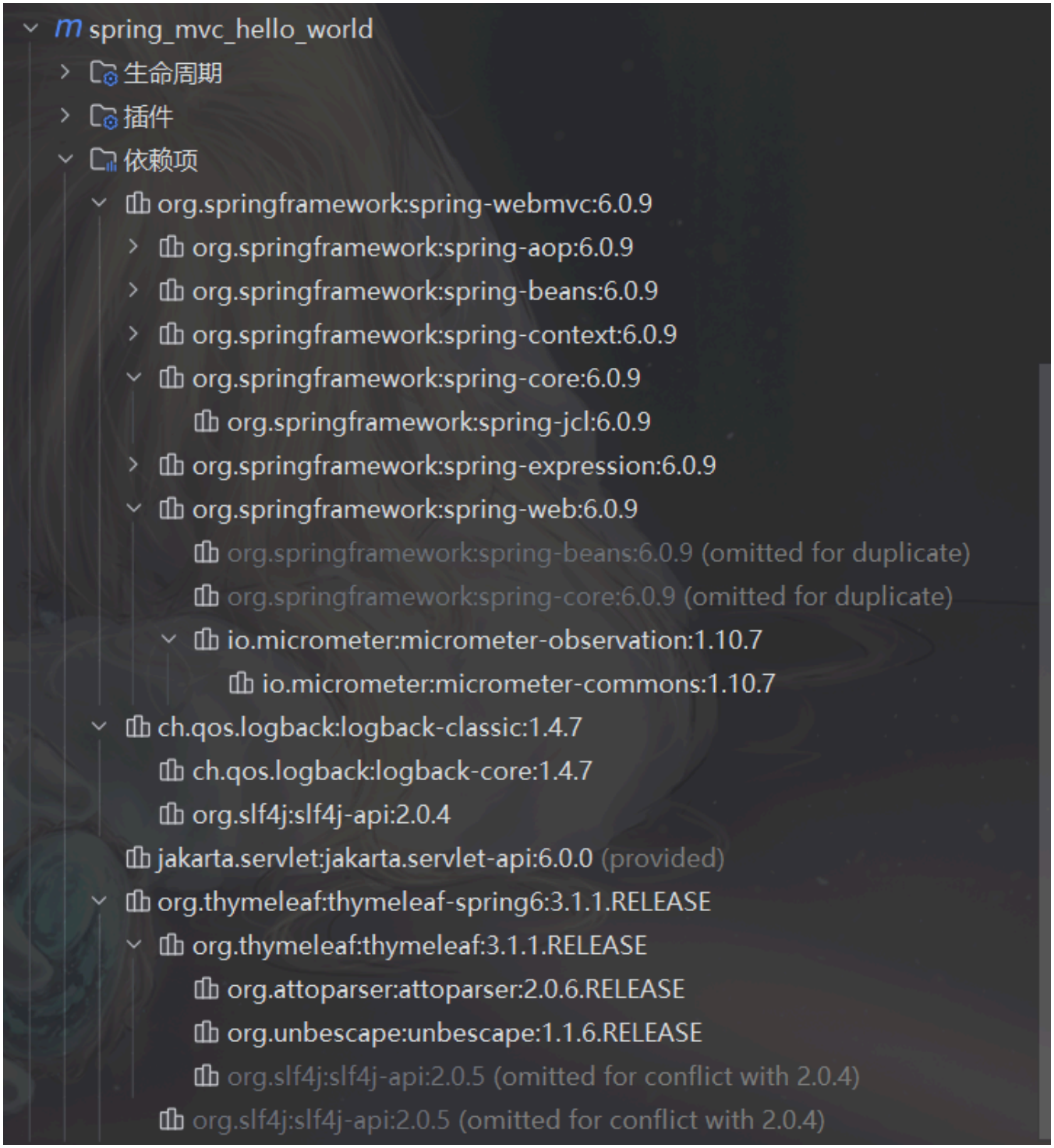
```
<dependencies>
    <!-- SpringMVC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>6.0.9</version>
    </dependency>

    <!-- 日志 -->
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.4.7</version>
    </dependency>

    <!-- ServletAPI -->
    <dependency>
        <groupId>jakarta.servlet</groupId>
        <artifactId>jakarta.servlet-api</artifactId>
        <version>6.0.0</version>
        <scope>provided</scope>
    </dependency>

    <!-- Spring6 和 Thymeleaf 整合包 -->
    <dependency>
        <groupId>org.thymeleaf</groupId>
        <artifactId>thymeleaf-spring6</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>
</dependencies>
```

> 注意：由于 Maven 的传递性，我们不必将所有需要的包全部配置依赖，而是配置最顶端的依赖，其他靠传递性导入。



2.3、配置 web.xml

注册 SpringMVC 的前端控制器 DispatcherServlet。

2.3.1 ① 默认配置方式

此配置作用下，SpringMVC 的配置文件默认位于 WEB-INF 下，默认名称为 <servlet-name>-

servlet.xml，例如，以下配置所对应 SpringMVC 的配置文件位于 WEB-INF 下，文件名为 springMVC-servlet.xml。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakarta
         version="6.0">
    <!-- 配置 SpringMVC 的前端控制器 DispatcherServlet -->
    <!--
        SpringMVC 的配置文件默认的位置和名称:
        位置: WEB-INF 下
        名称: <servlet-name>-servlet.xml, 当前配置下的配置文件名为 SpringMVC-servlet.xml
     -->
    <servlet>
        <servlet-name>springMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>springMVC</servlet-name>
        <!--
            url-pattern 中 / 和 /* 的区别:
            /: 匹配浏览器向服务器发送的所有请求（不包括 .jsp）
            /*: 匹配浏览器向服务器发送的所有请求（包括 .jsp）
         -->
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

2.3.2 ② 扩展配置方式

可通过 init-param 标签设置 SpringMVC 配置文件的位置和名称，通过 load-on-startup 标签设置 SpringMVC 前端控制器 DispatcherServlet 的初始化时间。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakarta
         version="6.0">
    <!-- 配置 SpringMVC 的前端控制器 DispatcherServlet -->
    <!--
        SpringMVC 的配置文件默认的位置和名称:
        位置: WEB-INF 下
        名称: <servlet-name>-servlet.xml, 当前配置下的配置文件名为 SpringMVC-servlet.xml
     -->
    <servlet>
        <servlet-name>springMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

        <!-- 设置 SpringMVC 配置文件的位置和名称 -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>

        <!-- 将 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 中 / 和 /* 的区别:
            /: 匹配浏览器向服务器发送的所有请求 (不包括 .jsp)
            /*: 匹配浏览器向服务器发送的所有请求 (包括 .jsp)
         -->
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

> 注意：
>
> 标签中使用 / 和 /* 的区别：
>
> / 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。
>
> 因此就可以避免在访问 jsp 页面时，该请求被 DispatcherServlet 处理，从而找不到相应的

> 页面。
>
> /* 则能够匹配所有请求，例如在使用过滤器时，若需要对所有请求进行过滤，就需要使用
> /* 的写法。

2.4、创建请求控制器

由于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。

```
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;

/**
 * @author MYXH
 * @date 2023/8/31
 */
@Controller
public class HelloController
{

}
```

2.5、创建 SpringMVC 的配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.
    <!-- 扫描控制层组件 -->
    <context:component-scan base-package="com.myxh.springmvc.controller"/>

    <!-- 配置 Thymeleaf 视图解析器 -->
    <bean id="viewResolver" class="org.thymeleaf.spring6.view.ThymeleafViewResolver">
        <property name="order" value="1"/>
        <property name="characterEncoding" value="UTF-8"/>
        <property name="templateEngine">
            <bean class="org.thymeleaf.spring6.SpringTemplateEngine">
                <property name="templateResolver">
                    <bean class="org.thymeleaf.spring6.templateresolver.SpringResourceTemplateReso
                        <!-- 视图前缀 -->
                        <property name="prefix" value="/WEB-INF/templates/"/>

                        <!-- 视图后缀 -->
                        <property name="suffix" value=".html"/>

                        <property name="templateMode" value="HTML"/>
                        <property name="characterEncoding" value="UTF-8"/>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>
</beans>
```

2.6、测试 HelloWorld

2.6.1 ① 实现对首页的访问

在请求控制器中创建处理请求的方法。

```java
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @author MYXH
 * @date 2023/8/31
 */
@Controller
public class HelloController
{
    @RequestMapping("/")
    public String portal()
    {
        // 将逻辑视图返回
        return "index";
    }
}
```

2.6.2 ② 通过超链接跳转到指定页面

在主页 index.html 中设置超链接。

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>首页</title>
  </head>

  <body>
    <h1>index.html</h1>
    <h2>1、通过超链接跳转到指定页面</h2>
    <a th:href="@{/hello}">测试 SpringMVC 相对路径</a>
    <br />
    <a href="/hello">测试绝对路径</a>
  </body>
</html>
```

创建 success.html 页面。

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>成功</title>
  </head>

  <body>
    <h1>success.html</h1>
  </body>
</html>
```

在请求控制器中创建处理请求的方法。

```java
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @author MYXH
 * @date 2023/8/31
 */
@Controller
public class HelloController
{
    @RequestMapping("/")
    public String portal()
    {
        // 将逻辑视图返回
        return "index";
    }

    @RequestMapping("/hello")
    public String hello()
    {
        return "success";
    }
}
```

2.7、总结

浏览器发送请求，若请求地址符合前端控制器的 url-pattern，该请求就会被前端控制器 DispatcherServlet 处理。前端控制器会读取 SpringMVC 的核心配置文件，通过扫描组件找到控制器，将请求地址和控制器中 @RequestMapping 注解的 value 属性值进行匹配，若匹配成功，该注解所标识的控制器方法就是处理请求的方法。处理请求的方法需要返回一个字符串类型的视图名称，该视图名称会被视图解析器解析，加上前缀和后缀组成视图的路径，通过 Thymeleaf 对视图进行渲染，最终转发到视图所对应页面。

3、@RequestMapping 注解

3.1、@RequestMapping 注解的功能

从注解名称上我们可以看到，@RequestMapping 注解的作用就是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 接收到指定的请求，就会来找到在映射关系中对应的控制器方法来处理这个请求。

3.2、@RequestMapping 注解的位置

@RequestMapping 标识一个类：设置映射请求的请求路径的初始信息。

@RequestMapping 标识一个方法：设置映射请求请求路径的具体信息。

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>首页</title>
  </head>

  <body>
    <h1>index.html</h1>

    <h2>1、@RequestMapping注解</h2>
    <h3>1.1、@RequestMapping 注解标识的位置</h3>
    <a th:href="@{/test/hello}">测试 @RequestMapping 注解所标识的位置</a>
  </body>
</html>
```

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>成功</title>
  </head>

  <body>
    <h1>success.html</h1>
  </body>
</html>
```

```java
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 1、@RequestMapping 注解标识的位置
 * {@code @RequestMapping} 标识一个类：设置映射请求的请求路径的初始信息
 * {@code @RequestMapping} 标识一个方法：设置映射请求请求路径的具体信息
 */
@Controller
@RequestMapping("/test")
public class TestRequestMappingController
{
    // 此时控制器方法所匹配的请求的请路径为 /test/hello
    @RequestMapping("/hello")
    public String hello()
    {
        return "success";
    }
}
```

3.3、@RequestMapping 注解的 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。

```
<h3>1.2、@RequestMapping 注解 value 属性</h3>
<a th:href="@{/test/hello2}">测试 @RequestMapping 注解的 value 属性</a>
```

```
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 1、@RequestMapping 注解标识的位置
 * {@code @RequestMapping} 标识一个类：设置映射请求的请求路径的初始信息
 * {@code @RequestMapping} 标识一个方法：设置映射请求请求路径的具体信息
 * <p>
 * 2、@RequestMapping 注解 value 属性
 * 作用：通过请求的请求路径匹配请求
 * value 属性是数组类型，即当前浏览器所发送请求的请求路径匹配 value 属性中的任何一个值
 * 则当前请求就会被注解所标识的方法进行处理
 */
@Controller
@RequestMapping("/test")
public class TestRequestMappingController
{
    // 此时控制器方法所匹配的请求的请路径为 /test/hello
    @RequestMapping(
            value = {"/hello", "/hello2"}
    )
    public String hello()
    {
        return "success";
    }
}
```

3.4、@RequestMapping 注解的 method 属性

@RequestMapping 注解的 method 属性通过请求的请求方式（get 或 post）匹配请求映射。

@RequestMapping 注解的 method 属性是一个 RequestMethod 类型的数组，表示该请求映射能够匹配多种请求方式的请求。

若当前请求的请求地址满足请求映射的 value 属性，但是请求方式不满足 method 属性，则浏览器报错 405：Request method ‘POST’ not supported。

```
<h3>1.3、@RequestMapping 注解的 method属性</h3>
<form th:action="@{/test/hello}" method="post">
  <input type="submit" value="测试 @RequestMapping 注解的 method 属性" />
</form>
```

```java
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 1、@RequestMapping 注解标识的位置
 * {@code @RequestMapping} 标识一个类：设置映射请求的请求路径的初始信息
 * {@code @RequestMapping} 标识一个方法：设置映射请求请求路径的具体信息
 * <p>
 * 2、@RequestMapping 注解 value 属性
 * 作用：通过请求的请求路径匹配请求
 * value 属性是数组类型，即当前浏览器所发送请求的请求路径匹配 value 属性中的任何一个值
 * 则当前请求就会被注解所标识的方法进行处理
 * <p>
 * 3、@RequestMapping 注解的 method属性
 * 作用：通过请求的请求方式匹配请求
 * method 属性是 RequestMethod 类型的数组，即当前浏览器所发送请求的请求方式匹配 method 属性中的任何一中请
 * 则当前请求就会被注解所标识的方法进行处理
 * 若浏览器所发送的请求的请求路径和 @RequestMapping 注解 value 属性匹配，但是请求方式不匹配
 * 此时页面报错：HTTP 状态 405 - Request method 'XXX' not supported
 * 在 @RequestMapping 的基础上，结合请求方式的一些派生注解：
 * {@code @GetMapping}, @PostMapping, @DeleteMapping, @PutMapping
 */
@Controller
@RequestMapping("/test")
public class TestRequestMappingController
{
    // 此时控制器方法所匹配的请求的请路径为 /test/hello
    @RequestMapping(
            value = {"/hello", "/hello2"},
            method = {RequestMethod.POST, RequestMethod.GET},
    )
    public String hello()
    {
        return "success";
    }
}
```

> 注意：

> 1、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。
>
> 2、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 HiddenHttpMethodFilter，在 RESTful 部分。

3.5、@RequestMapping 注解的 params 属性（了解）

@RequestMapping 注解的 params 属性通过请求的请求参数匹配请求映射。

@RequestMapping 注解的 params 属性是一个字符串类型的数组，可以通过四种表达式设置请求参数和请求映射的匹配关系。

“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。

```
<h3>1.4、@RequestMapping 注解的 params 属性</h3>
<form th:action="@{|/test/hello?username=${'MYXH'}|}" method="post">
  <input type="submit" value="测试 @RequestMapping 注解的 params 属性" />
</form>

<form th:action="@{/test/hello(username='MYXH')}" method="post">
  <input type="submit" value="测试 @RequestMapping 注解的 params 属性" />
</form>
```

```java
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 1、@RequestMapping 注解标识的位置
 * {@code @RequestMapping} 标识一个类：设置映射请求的请求路径的初始信息
 * {@code @RequestMapping} 标识一个方法：设置映射请求请求路径的具体信息
 * <p>
 * 2、@RequestMapping 注解 value 属性
 * 作用：通过请求的请求路径匹配请求
 * value 属性是数组类型，即当前浏览器所发送请求的请求路径匹配 value 属性中的任何一个值
 * 则当前请求就会被注解所标识的方法进行处理
 * <p>
 * 3、@RequestMapping 注解的 method属性
 * 作用：通过请求的请求方式匹配请求
 * method 属性是 RequestMethod 类型的数组，即当前浏览器所发送请求的请求方式匹配 method 属性中的任何一中请
 * 则当前请求就会被注解所标识的方法进行处理
 * 若浏览器所发送的请求的请求路径和 @RequestMapping 注解 value 属性匹配，但是请求方式不匹配
 * 此时页面报错：HTTP 状态 405 - Request method 'XXX' not supported
 * 在 @RequestMapping 的基础上，结合请求方式的一些派生注解：
 * {@code @GetMapping}, @PostMapping, @DeleteMapping, @PutMapping
 * <p>
 * 4、@RequestMapping 注解的 params 属性
 * 作用：通过请求的请求参数匹配请求，即浏览器发送的请求的请求参数必须满足 params 属性的设置
 * params 可以使用四种表达式：
 * "param"：表示当前所匹配请的请求参数中必须携带 param 参数
 * "!param"：表示当前所匹配请求的请求参数中一定不能携带 param 参数
 * "param=value"：表示当前所匹配请求的请求参数中必须携带 param 参数且值必须为 value
 * "param!=value"：表示当前所匹配请求的请求参数中可以不携带 param，若携带值一定不能是 value
 * 若浏览器所发送的请求的请求路径和 @RequestMapping 注解 value 属性匹配，但是请求参数不匹配
 * 此时页面报错：HTTP 状态 400 - Parameter conditions "username" not met for actual request paramete
 * <p>
 * 5、@RequestMapping 注解的 headers 属性
 * 作用：通过请来的请求头信息匹配请求，即浏览器发送的请求的请求头信息必须满足 headers 属性的设置
 * headers 可以使用四种表达式：
 * "header"：表示当前所匹配请的请求参数中必须携带 header 参数
 * "!header"：表示当前所匹配请求的请求参数中一定不能携带 header 参数
```

```
 * "header=value": 表示当前所匹配请求的请求参数中必须携带 header 参数且值必须为 value
 * "header!=value": 表示当前所匹配请求的请求参数中可以不携带 header，若携带值一定不能是 value
 * 若浏览器所发送的请求的请求路径和 @RequestMapping 注解 value 属性匹配，但是请求头信息不匹配
 * 此时页面报错：HTTP 状态 404 - The requested resource is not available.
 */
@Controller
@RequestMapping("/test")
public class TestRequestMappingController
{
    // 此时控制器方法所匹配的请求的请路径为 /test/hello
    @RequestMapping(
            value = {"/hello", "/hello2"},
            method = {RequestMethod.POST, RequestMethod.GET},
            params = {"username", "!password", "age=21", "gender!=女"}
            headers = {"Referer"}
    )
    public String hello()
    {
        return "success";
    }
}
```

> 注意：
>
> 若当前请求满足 @RequestMapping 注解的 value 和 method 属性，但是不满足 params 属性，此时页面回报错 400：Parameter conditions “username, password!=520.ILY！” not met for actual request parameters: username={MYXH}, password={520.ILY!}

3.6、@RequestMapping 注解的 headers 属性（了解）

@RequestMapping 注解的 headers 属性通过请求的请求头信息匹配请求映射。

@RequestMapping 注解的 headers 属性是一个字符串类型的数组，可以通过四种表达式设置请求头信息和请求映射的匹配关系。

“header”：要求请求映射所匹配的请求必须携带 header 请求头信息。

“!header”：要求请求映射所匹配的请求必须不能携带 header 请求头信息。

“header=value”：要求请求映射所匹配的请求必须携带 header 请求头信息且 header=value。

“header!=value”：要求请求映射所匹配的请求必须携带 header 请求头信息且 header!=value。

若当前请求满足@RequestMapping 注解的 value 和 method 属性，但是不满足 headers 属性，此时页面显示 404 错误，即资源未找到。

3.7、SpringMVC 支持 ant 风格的路径

？：表示任意的单个字符。

*：表示任意的 0 个或多个字符。

**：表示任意层数的任意目录。

注意：在使用**时，只能使用/**/xx 的方式。

3.8、SpringMVC 支持路径中的占位符（重点）

原始方式：/deleteUser?id=1

rest 方式：/user/delete/1

SpringMVC 路径中的占位符常用于 RESTful 风格中，当请求路径中将某些数据通过路径的方式传输到服务器中，就可以在相应的 @RequestMapping 注解的 value 属性中通过占位符{xx}表示传输的数据，在通过 @PathVariable 注解，将占位符所表示的数据赋值给控制器方法的形参。

```
<h3>1.5、SpringMVC 支持 ant 风格的路径</h3>
<a th:href="@{/test/ant/test}">测试 @RequestMapping 注解支持 ant 风格的路径</a>

<h3>1.6、SpringMVC 支持路径中的占位符</h3>
<form th:action="@{/test/rest/1/MYXH}" method="post">
  <input type="submit" value="测试 @RequestMapping 注解的 value 属性的占位符" />
</form>
```

```java
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 1、@RequestMapping 注解标识的位置
 * {@code @RequestMapping} 标识一个类：设置映射请求的请求路径的初始信息
 * {@code @RequestMapping} 标识一个方法：设置映射请求请求路径的具体信息
 * <p>
 * 2、@RequestMapping 注解 value 属性
 * 作用：通过请求的请求路径匹配请求
 * value 属性是数组类型，即当前浏览器所发送请求的请求路径匹配 value 属性中的任何一个值
 * 则当前请求就会被注解所标识的方法进行处理
 * <p>
 * 3、@RequestMapping 注解的 method属性
 * 作用：通过请求的请求方式匹配请求
 * method 属性是 RequestMethod 类型的数组，即当前浏览器所发送请求的请求方式匹配 method 属性中的任何一中请
 * 则当前请求就会被注解所标识的方法进行处理
 * 若浏览器所发送的请求的请求路径和 @RequestMapping 注解 value 属性匹配，但是请求方式不匹配
 * 此时页面报错：HTTP 状态 405 - Request method 'XXX' not supported
 * 在 @RequestMapping 的基础上，结合请求方式的一些派生注解：
 * {@code @GetMapping}, @PostMapping, @DeleteMapping, @PutMapping
 * <p>
 * 4、@RequestMapping 注解的 params 属性
 * 作用：通过请求的请求参数匹配请求，即浏览器发送的请求的请求参数必须满足 params 属性的设置
 * params 可以使用四种表达式：
 * "param"：表示当前所匹配请的请求参数中必须携带 param 参数
 * "!param"：表示当前所匹配请求的请求参数中一定不能携带 param 参数
 * "param=value"：表示当前所匹配请求的请求参数中必须携带 param 参数且值必须为 value
 * "param!=value"：表示当前所匹配请求的请求参数中可以不携带 param，若携带值一定不能是 value
 * 若浏览器所发送的请求的请求路径和 @RequestMapping 注解 value 属性匹配，但是请求参数不匹配
 * 此时页面报错：HTTP 状态 400 - Parameter conditions "username" not met for actual request paramete
 * <p>
 * 5、@RequestMapping 注解的 headers 属性
 * 作用：通过请来的请求头信息匹配请求，即浏览器发送的请求的请求头信息必须满足 headers 属性的设置
 * headers 可以使用四种表达式：
 * "header"：表示当前所匹配请的请求参数中必须携带 header 参数
 * "!header"：表示当前所匹配请求的请求参数中一定不能携带 header 参数
```

```
 * "header=value": 表示当前所匹配请求的请求参数中必须携带 header 参数且值必须为 value
 * "header!=value": 表示当前所匹配请求的请求参数中可以不携带 header，若携带值一定不能是 value
 * 若浏览器所发送的请求的请求路径和 @RequestMapping 注解 value 属性匹配，但是请求头信息不匹配
 * 此时页面报错：HTTP 状态 404 - The requested resource is not available.
 * <p>
 * 6、SpringMVC 支持 ant 风格的路径
 * 在 @RequestMapping 注解的 value 属性值中设置一些特殊字符
 * ?：任意的单个字符（不包括 ? 和 /）
 * *：任意个数的任意字符（不包括 ? 和 /）
 * **：任意层数的任意目录，注意使用方式只能 / ** /，** 写在双斜线中，前后不能有任何的其他字符
 * <p>
 * 7、@RequestMapping 注解使用路径中的占位符
 * 传统：/deleteUser?id=1
 * rest：/user/delete/1
 * 需要在 @RequestMapping 注解的 value 属性中所设置的路径中，使用{xx}的方式表示路径中的数据
 * 在通过 @PathVariable 注解，将位符所标识的值和控制器方法的形参进行绑定
 */
@Controller
@RequestMapping("/test")
public class TestRequestMappingController
{
    // 此时控制器方法所匹配的请求的请路径为 /test/hello
    @RequestMapping(
            value = {"/hello", "/hello2"},
            method = {RequestMethod.POST, RequestMethod.GET},
            // params = {"username", "!password", "age=21", "gender!=女"}
            headers = {"Referer"}
    )
    public String hello()
    {
        return "success";
    }

    @RequestMapping("/ant/**")
    public String testAnt()
    {
        return "success";
    }

    @RequestMapping("/rest/{id}/{username}")
    public String testRest(@PathVariable("id") Integer id, @PathVariable("username") String userna
    {
        System.out.println("id = " + id);
        System.out.println("username = " + username);
```

```
        return "success";
    }
}
```

4、SpringMVC 获取请求参数

4.1、通过 ServletAPI 获取

将 HttpServletRequest 作为控制器方法的形参，此时 HttpServletRequest 类型的参数表示封装了当前请求的请求报文的对象。

```
<h2>2、SpringMVC 获取请求参数</h2>
<h3>2.1、通过 ServletAPI 获取请求参数</h3>
<form th:action="@{/param/servlet/api}" method="post">
  <label>
    用户名:
    <input type="text" name="username" /><br />
    密 码: <input type="password" name="password" /><br />
    <input type="submit" value="登录" />
  </label>
</form>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpSession;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @author MYXH
 * @date 2023/9/1
 * @deprecated
 * 获取请求参数的方式:
 * 1、通过 ServletAPI 获获取
 * 只需要在控制器方法的形参位置设置 HttpServletRequest 类型的形参
 * 就可以在控制器方法中使用 request 对象获取请求参数
 */
@Controller
public class TestParamController
{
    @RequestMapping("/param/servlet/api")
    public String getParamByServletAPI(HttpServletRequest request)
    {
        HttpSession session = request.getSession();

        String username = request.getParameter("username");
        String password = request.getParameter("password");
        System.out.println("username = " + username);
        System.out.println("password = " + password);

        return "success";
    }
}
```

4.2、通过控制器方法的形参获取请求参数

在控制器方法的形参位置，设置和请求参数同名的形参，当浏览器发送请求，匹配到请求映射时，在 DispatcherServlet 中就会将请求参数赋值给相应的形参。

```
<h3>2.2、通过控制器方法的形参获取请求参数</h3>
<form th:action="@{/param}" method="post">
  <label>
    用户名:
    <input type="text" name="name" /><br />
    密 码: <input type="password" name="password" /><br />
    <input type="submit" value="登录" />
  </label>
</form>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpSession;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.CookieValue;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

/**
 * @author MYXH
 * @date 2023/9/1
 * @deprecated
 * 获取请求参数的方式：
 * 1、通过 ServletAPI 获获取
 * 只需要在控制器方法的形参位置设置 HttpServletRequest 类型的形参
 * 就可以在控制器方法中使用 request 对象获取请求参数
 * <p>
 * 2、通过控制器方法的形参获取
 * 只需要在控制器方法的形参位置，设置一个形参，形参的名字和请求参数的名字一致即可
 * <p>
 * 3、@RequestParam：将请求参数和控制器方法的形参绑定
 * {@code @RequestParam} 注解的三个属性：value，required，defaultValue
 * value：设置和形参绑定的请参数的名字
 * required：设置是否必须传输 vaLue 所对应的请求参数
 * 默认值为 true，表示 value 所对应的请求参数必须传输，否则页面报错：
 * HTTP 状态 400 - Required string parameter 'xxx' is not present
 * 若设置为 false，则表示 value 所对应的请求参数不是必须传输，若为传输，则形参值为 null
 * defaultValue：设置当没有传输 value 所对应的请求参数时，为形参设置的默认值，此时和 required 属性值无关
 * <p>
 * 4、@RequestHeader：请求头信息和控制器方法的形参绑定
 * {@code @RequestHeader} 注解的三个属性：value，required，defaultValue
 * <p>
 * 5、@CookieValue：cookie 数据和控制器方法的形参绑定
 * {@code @CookieValue} 注解的三个属性：value，required，defaultValue
 */
@Controller
public class TestParamController
{
    @RequestMapping("/param/servlet/api")
    public String getParamByServletAPI(HttpServletRequest request)
    {
        HttpSession session = request.getSession();
```

```
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        System.out.println("username = " + username);
        System.out.println("password = " + password);

        return "success";
    }

    @RequestMapping("/param")
    public String getParam(@RequestParam(value = "name", required = true, defaultValue = "MYXH") S
                           String password,
                           @RequestHeader("Referer") String referer,
                           @CookieValue("JSESSIONID") String jsessionId
    )
    {
        System.out.println("username = " + username);
        System.out.println("password = " + password);
        System.out.println("referer = " + referer);
        System.out.println("jsessionId = " + jsessionId);

        return "success";
    }
}
```

> 注意：
>
> 若请求所传输的请求参数中有多个同名的请求参数，此时可以在控制器方法的形参中设置字符串数组或者字符串类型的形参接收此请求参数。
>
> 若使用字符串数组类型的形参，此参数的数组中包含了每一个数据。
>
> 若使用字符串类型的形参，此参数的值为每个数据中间使用逗号拼接的结果。

4.3、@RequestParam

@RequestParam 是将请求参数和控制器方法的形参创建映射关系。

@RequestParam 注解一共有三个属性：

value：指定为形参赋值的请求参数的参数名。

required：设置是否必须传输此请求参数，默认值为 true。

若设置为 true 时，则当前请求必须传输 value 所指定的请求参数，若没有传输该请求参数，且没有设置 defaultValue 属性，则页面报错 400：Required String parameter ‘xxx’ is not present；若设置为 false，则当前请求不是必须传输 value 所指定的请求参数，若没有传输，则注解所标识的形参的值为 null。

defaultValue：不管 required 属性值为 true 或 false，当 value 所指定的请求参数没有传输或传输的值为""时，则使用默认值为形参赋值。

4.4、@RequestHeader

@RequestHeader 是将请求头信息和控制器方法的形参创建映射关系。

@RequestHeader 注解一共有三个属性：value、required、defaultValue，用法同 @RequestParam。

4.5、@CookieValue

@CookieValue 是将 cookie 数据和控制器方法的形参创建映射关系。

@CookieValue 注解一共有三个属性：value、required、defaultValue，用法同 @RequestParam。

4.6、通过 POJO 获取请求参数

可以在控制器方法的形参位置设置一个实体类类型的形参，此时若浏览器传输的请求参数的参数名和实体类中的属性名一致，那么请求参数就会为此属性赋值。

```
<h3>2.3、通过 pojo 取请求参数</h3>
<form th:action="@{/param/pojo}" method="post">
  <label>
    用户名:
    <input type="text" name="username" /><br />
    密 码: <input type="password" name="password" /><br />
    <input type="submit" value="登录" />
  </label>
</form>
```

```java
package com.myxh.springmvc.pojo;

import org.springframework.stereotype.Component;

/**
 * @author MYXH
 * @date 2023/9/1
 */
@Component
public class User
{
    private Integer id;
    private String username;
    private String password;
    private Integer age;
    private String gender;
    private String email;

    public User()
    {

    }

    public User(Integer id, String username, String password, Integer age, String gender, String e
    {
        this.id = id;
        this.username = username;
        this.password = password;
        this.age = age;
        this.gender = gender;
        this.email = email;
    }

    public Integer getId()
    {
        return id;
    }

    public void setId(Integer id)
    {
        this.id = id;
    }

    public String getUsername()
```

```
    {
        return username;
    }

    public void setUsername(String username)
    {
        this.username = username;
    }

    public String getPassword()
    {
        return password;
    }

    public void setPassword(String password)
    {
        this.password = password;
    }

    public Integer getAge()
    {
        return age;
    }

    public void setAge(Integer age)
    {
        this.age = age;
    }

    public String getGender()
    {
        return gender;
    }

    public void setGender(String gender)
    {
        this.gender = gender;
    }

    public String getEmail()
    {
        return email;
    }
```

```
    public void setEmail(String email)
    {
        this.email = email;
    }

    @Override
    public String toString()
    {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                ", email='" + email + '\'' +
                '}';
    }
}
```

```java
package com.myxh.springmvc.controller;

import com.myxh.springmvc.pojo.User;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpSession;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.CookieValue;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

/**
 * @author MYXH
 * @date 2023/9/1
 * @deprecated
 * 获取请求参数的方式:
 * 1、通过 ServletAPI 获获取
 * 只需要在控制器方法的形参位置设置 HttpServletRequest 类型的形参
 * 就可以在控制器方法中使用 request 对象获取请求参数
 * <p>
 * 2、通过控制器方法的形参获取
 * 只需要在控制器方法的形参位置，设置一个形参，形参的名字和请求参数的名字一致即可
 * <p>
 * 3、@RequestParam: 将请求参数和控制器方法的形参绑定
 * {@code @RequestParam} 注解的三个属性: value, required, defaultValue
 * value: 设置和形参绑定的请参数的名字
 * required: 设置是否必须传输 vaLue 所对应的请求参数
 * 默认值为 true，表示 value 所对应的请求参数必须传输，否则页面报错:
 * HTTP 状态 400 - Required string parameter 'xxx' is not present
 * 若设置为 false，则表示 value 所对应的请求参数不是必须传输，若为传输，则形参值为 null
 * defaultValue: 设置当没有传输 value 所对应的请求参数时，为形参设置的默认值，此时和 required 属性值无关
 * <p>
 * 4、@RequestHeader: 请求头信息和控制器方法的形参绑定
 * {@code @RequestHeader} 注解的三个属性: value, required, defaultValue
 * <p>
 * 5、@CookieValue: cookie 数据和控制器方法的形参绑定
 * {@code @CookieValue} 注解的三个属性: value, required, defaultValue
 * <p>
 * 6、通过控制器方法的实体类类型的形参获取请求参数
 * 需要在控制器方法的形参位置设置实体类类型的形参，要保证实体类中的属性的属性名和请求参数的名字一致
 * 可以通过实体类类型的形参获取请求参数
 * <p>
 * 7、解决获取请求此参数的乱码问题
 * 在 web.xml 中配置 Spring 的编码过滤器 characterEncodingFilter
```

```
 */
@Controller
public class TestParamController
{
    @RequestMapping("/param/servlet/api")
    public String getParamByServletAPI(HttpServletRequest request)
    {
        HttpSession session = request.getSession();

        String username = request.getParameter("username");
        String password = request.getParameter("password");
        System.out.println("username = " + username);
        System.out.println("password = " + password);

        return "success";
    }

    @RequestMapping("/param")
    public String getParam(@RequestParam(value = "name", required = true, defaultValue = "MYXH") S
                           String password,
                           @RequestHeader("Referer") String referer,
                           @CookieValue("JSESSIONID") String jsessionId
    )
    {
        System.out.println("username = " + username);
        System.out.println("password = " + password);
        System.out.println("referer = " + referer);
        System.out.println("jsessionId = " + jsessionId);

        return "success";
    }

    @RequestMapping("/param/pojo")
    public String getParamByPojo(User user)
    {
        System.out.println("user = " + user);

        return "success";
    }
}
```

4.7、解决获取请求参数的乱码问题

解决获取请求参数的乱码问题，可以使用 SpringMVC 提供的编码过滤器

CharacterEncodingFilter，但是必须在 web.xml 中进行注册。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakarta
         version="6.0">
    <!-- 配置 Spring 的编码过滤器 -->
    <filter>
        <filter-name>characterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>

        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>characterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- 设置 SpringMVC 的前端控制器 -->
    <servlet>
        <servlet-name>springMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>

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

    <servlet-mapping>
        <servlet-name>springMVC</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

> 注意：
>
> SpringMVC 中处理编码的过滤器一定要配置到其他过滤器之前，否则无效。

5、域对象共享数据

5.1、使用 ServletAPI 向 request 域对象共享数据

```
<h2>3、域对象共享数据</h2>
<h3>3.1、使用 ServletAPI 向 request 域对象共享数据</h3>
<a th:href="@{/test/servlet/api}">测试 ServletAPI 向 request 域对象共享数据</a>
```

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>成功</title>
  </head>

  <body>
    <h1>success.html</h1>
    <p th:text="${testRequestScope}" />
  </body>
</html>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;


import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 向域对象共享数据:
 * 1、使用 ServletAPI 向 request 域对象共享数据
 */
@Controller
public class TestScopeController
{
    @RequestMapping("/test/servlet/api")
    public String testServletAPI(HttpServletRequest request)
    {
        request.setAttribute("testRequestScope", "Hello, ServletAPI");
        Object testRequestScope = request.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }
}
```

5.2、使用 ModelAndView 向 request 域对象共享数据

```
<h3>3.2、使用 ModelAndView 向 request 域对象共享数据</h3>
<a th:href="@{|/test/model/${#strings.escapeXml('and')}/view|}"
  >测试 ModelAndView 向 request 域对象共享数据</a
>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 向域对象共享数据:
 * 1、使用 ServletAPI 向 request 域对象共享数据
 * <p>
 * 2、使用 ModelAndView 向 request 域对象共享数据
 * 使刚 ModelAndView 时，可以使用其 Model 功能向请求域共享数据
 * 使用 View 功能设置逻辑视图，但是控制器方法一定要 ModelAndView 作为方法的返回值
 */
@Controller
public class TestScopeController
{
    @RequestMapping("/test/servlet/api")
    public String testServletAPI(HttpServletRequest request)
    {
        request.setAttribute("testRequestScope", "Hello, ServletAPI");
        Object testRequestScope = request.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    /**
     * ModelAndView 包含 Model 和 view 的功能
     * Model: 向请求域中共享数据
     * view: 设置逻辑视图实现页面跳转
     *
     * @return 模型和视图
     */
    @RequestMapping("/test/model/and/view")
    public ModelAndView testModelAndView()
    {
        ModelAndView modelAndView = new ModelAndView();
```

```
        // 向请求域中共享数据
        modelAndView.addObject("testRequestScope", "Hello, ModelAndView");

        // 设置逻辑视图
        modelAndView.setViewName("success");

        return modelAndView;
    }
}
```

5.3、使用 Model 向 request 域对象共享数据

```
<h3>3.3、使用 Model 向 request 域对象共享数据</h3>
<a th:href="@{/test/model}">测试 Model 向 request 域对象共享数据</a>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 向域对象共享数据：
 * 1、使用 ServletAPI 向 request 域对象共享数据
 * <p>
 * 2、使用 ModelAndView 向 request 域对象共享数据
 * 使刚 ModelAndView 时，可以使用其 Model 功能向请求域共享数据
 * 使用 View 功能设置逻辑视图，但是控制器方法一定要 ModelAndView 作为方法的返回值
 * <p>
 * 3、使用 Model 向 request 域对象共享数据
 */
@Controller
public class TestScopeController
{
    @RequestMapping("/test/servlet/api")
    public String testServletAPI(HttpServletRequest request)
    {
        request.setAttribute("testRequestScope", "Hello, ServletAPI");
        Object testRequestScope = request.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    /**
     * ModelAndView 包含 Model 和 view 的功能
     * Model：向请求域中共享数据
     * view：设置逻辑视图实现页面跳转
     *
     * @return 模型和视图
     */
    @RequestMapping("/test/model/and/view")
```

```java
    public ModelAndView testModelAndView()
    {
        ModelAndView modelAndView = new ModelAndView();

        // 向请求域中共享数据
        modelAndView.addObject("testRequestScope", "Hello, ModelAndView");

        // 设置逻辑视图
        modelAndView.setViewName("success");

        return modelAndView;
    }

    @RequestMapping("/test/model")
    public String testModel(Model model)
    {
        // org.springframework.validation.support.BindingAwareModelMap
        System.out.println("model.getClass().getName() = " + model.getClass().getName());

        // 向请求域中共享数据
        model.addAttribute("testRequestScope", "Hello, Model");
        Object testRequestScope = model.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }
}
```

5.4、使用 ModelMap 向 request 域对象共享数据

```html
<h3>3.4、使用 ModelMap 向 request 域对象共享数据</h3>
<a th:href="@{/test/model/map}">测试 ModelMap 向 request 域对象共享数据</a>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 向域对象共享数据：
 * 1、使用 ServletAPI 向 request 域对象共享数据
 * <p>
 * 2、使用 ModelAndView 向 request 域对象共享数据
 * 使刚 ModelAndView 时，可以使用其 Model 功能向请求域共享数据
 * 使用 View 功能设置逻辑视图，但是控制器方法一定要 ModelAndView 作为方法的返回值
 * <p>
 * 3、使用 Model 向 request 域对象共享数据
 * <p>
 * 4、使用 ModelMap 向 request 域对象共享数据
 */
@Controller
public class TestScopeController
{
    @RequestMapping("/test/servlet/api")
    public String testServletAPI(HttpServletRequest request)
    {
        request.setAttribute("testRequestScope", "Hello, ServletAPI");
        Object testRequestScope = request.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    /**
     * ModelAndView 包含 Model 和 view 的功能
     * Model：向请求域中共享数据
     * view：设置逻辑视图实现页面跳转
     *
```

```
     * @return 模型和视图
     */
    @RequestMapping("/test/model/and/view")
    public ModelAndView testModelAndView()
    {
        ModelAndView modelAndView = new ModelAndView();

        // 向请求域中共享数据
        modelAndView.addObject("testRequestScope", "Hello, ModelAndView");

        // 设置逻辑视图
        modelAndView.setViewName("success");

        return modelAndView;
    }

    @RequestMapping("/test/model")
    public String testModel(Model model)
    {
        // org.springframework.validation.support.BindingAwareModelMap
        System.out.println("model.getClass().getName() = " + model.getClass().getName());

        // 向请求域中共享数据
        model.addAttribute("testRequestScope", "Hello, Model");
        Object testRequestScope = model.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    @RequestMapping("/test/model/map")
    public String testModelMap(ModelMap modelMap)
    {
        // org.springframework.validation.support.BindingAwareModelMap
        System.out.println("modelMap.getClass().getName() = " + modelMap.getClass().getName());

        // 向请求域中共享数据
        modelMap.addAttribute("testRequestScope", "Hello, ModelMap");
        Object testRequestScope = modelMap.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }
}
```

5.5、使用 map 向 request 域对象共享数据

```
<h3>3.5、使用 map 向 request 域对象共享数据</h3>
<a th:href="@{/test/map}">测试 map 向 request 域对象共享数据</a>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 向域对象共享数据：
 * 1、使用 ServletAPI 向 request 域对象共享数据
 * <p>
 * 2、使用 ModelAndView 向 request 域对象共享数据
 * 使刚 ModelAndView 时，可以使用其 Model 功能向请求域共享数据
 * 使用 View 功能设置逻辑视图，但是控制器方法一定要 ModelAndView 作为方法的返回值
 * <p>
 * 3、使用 Model 向 request 域对象共享数据
 * <p>
 * 4、使用 ModelMap 向 request 域对象共享数据
 * <p>
 * 5、使用 map 向 request 域对象共享数据
 * 6、Model 和 ModelMap 和 map 的关系
 * 其实在底层中，这些类型的形参最终都是通过 BindingAwareModelMap 创建
 * public class BindingAwareModelMap extends ExtendedModelMap {}
 * public class ExtendedModelMap extends ModelMap implements Model {}
 * public class ModelMap extends LinkedHashMap<String, Object> {}
 * public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V> {}
 * public interface Model {}
 */
@Controller
public class TestScopeController
{
    @RequestMapping("/test/servlet/api")
    public String testServletAPI(HttpServletRequest request)
    {
        request.setAttribute("testRequestScope", "Hello, ServletAPI");
        Object testRequestScope = request.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);
```

```
        return "success";
    }

    /**
     * ModelAndView 包含 Model 和 view 的功能
     * Model：向请求域中共享数据
     * view：设置逻辑视图实现页面跳转
     *
     * @return 模型和视图
     */
    @RequestMapping("/test/model/and/view")
    public ModelAndView testModelAndView()
    {
        ModelAndView modelAndView = new ModelAndView();

        // 向请求域中共享数据
        modelAndView.addObject("testRequestScope", "Hello, ModelAndView");

        // 设置逻辑视图
        modelAndView.setViewName("success");

        return modelAndView;
    }

    @RequestMapping("/test/model")
    public String testModel(Model model)
    {
        // org.springframework.validation.support.BindingAwareModelMap
        System.out.println("model.getClass().getName() = " + model.getClass().getName());

        // 向请求域中共享数据
        model.addAttribute("testRequestScope", "Hello, Model");
        Object testRequestScope = model.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    @RequestMapping("/test/model/map")
    public String testModelMap(ModelMap modelMap)
    {
        // org.springframework.validation.support.BindingAwareModelMap
        System.out.println("modelMap.getClass().getName() = " + modelMap.getClass().getName());
```

```
        // 向请求域中共享数据
        modelMap.addAttribute("testRequestScope", "Hello, ModelMap");
        Object testRequestScope = modelMap.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    @RequestMapping("/test/map")
    public String testMap(Map<String, Object> map)
    {
        // org.springframework.validation.support.BindingAwareModelMap
        System.out.println("map.getClass().getName() = " + map.getClass().getName());

        // 向请求域中共享数据
        map.put("testRequestScope", "Hello, map");
        Object testRequestScope = map.get("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }
}
```

5.6、Model、ModelMap、Map 的关系

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

```
public interface Model {}
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V> {}
public class ModelMap extends LinkedHashMap<String, Object> {}
public class ExtendedModelMap extends ModelMap implements Model {}
public class BindingAwareModelMap extends ExtendedModelMap {}
```

5.7、向 session 域共享数据

```
<h3>3.6、使用 ServletAPI 向 session 域共享数据</h3>
<a th:href="@{/test/session}">测试 ServletAPI 向 session 域共享数据</a>
```

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>成功</title>
  </head>

  <body>
    <h1>success.html</h1>
    <p th:text="${testRequestScope}" />
    <p th:text="${session.testSessionScope}" />
  </body>
</html>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpSession;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 向域对象共享数据:
 * 1、使用 ServletAPI 向 request 域对象共享数据
 * <p>
 * 2、使用 ModelAndView 向 request 域对象共享数据
 * 使刚 ModelAndView 时，可以使用其 Model 功能向请求域共享数据
 * 使用 View 功能设置逻辑视图，但是控制器方法一定要 ModelAndView 作为方法的返回值
 * <p>
 * 3、使用 Model 向 request 域对象共享数据
 * <p>
 * 4、使用 ModelMap 向 request 域对象共享数据
 * <p>
 * 5、使用 map 向 request 域对象共享数据
 * <p>
 * 6、Model 和 ModelMap 和 map 的关系
 * 其实在底层中，这些类型的形参最终都是通过 BindingAwareModelMap 创建
 * public class BindingAwareModelMap extends ExtendedModelMap {}
 * public class ExtendedModelMap extends ModelMap implements Model {}
 * public class ModelMap extends LinkedHashMap<String, Object> {}
 * public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V> {}
 * public interface Model {}
 */
@Controller
public class TestScopeController
{
    @RequestMapping("/test/servlet/api")
    public String testServletAPI(HttpServletRequest request)
    {
        request.setAttribute("testRequestScope", "Hello, ServletAPI");
```

```
        Object testRequestScope = request.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    /**
     * ModelAndView 包含 Model 和 view 的功能
     * Model：向请求域中共享数据
     * view：设置逻辑视图实现页面跳转
     *
     * @return 模型和视图
     */
    @RequestMapping("/test/model/and/view")
    public ModelAndView testModelAndView()
    {
        ModelAndView modelAndView = new ModelAndView();

        // 向请求域中共享数据
        modelAndView.addObject("testRequestScope", "Hello, ModelAndView");

        // 设置逻辑视图
        modelAndView.setViewName("success");

        return modelAndView;
    }

    @RequestMapping("/test/model")
    public String testModel(Model model)
    {
        // org.springframework.validation.support.BindingAwareModelMap
        System.out.println("model.getClass().getName() = " + model.getClass().getName());

        // 向请求域中共享数据
        model.addAttribute("testRequestScope", "Hello, Model");
        Object testRequestScope = model.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    @RequestMapping("/test/model/map")
    public String testModelMap(ModelMap modelMap)
    {
```

```
        // org.springframework.validation.support.BindingAwareModelMap
        System.out.println("modelMap.getClass().getName() = " + modelMap.getClass().getName());

        // 向请求域中共享数据
        modelMap.addAttribute("testRequestScope", "Hello, ModelMap");
        Object testRequestScope = modelMap.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    @RequestMapping("/test/map")
    public String testMap(Map<String, Object> map)
    {
        // org.springframework.validation.support.BindingAwareModelMap
        System.out.println("map.getClass().getName() = " + map.getClass().getName());

        // 向请求域中共享数据
        map.put("testRequestScope", "Hello, map");
        Object testRequestScope = map.get("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    @RequestMapping("/test/session")
    public String testSession(HttpSession session)
    {
        session.setAttribute("testSessionScope", "Hello, Session");
        Object testSessionScope = session.getAttribute("testSessionScope");
        System.out.println("testSessionScope = " + testSessionScope);

        return "success";
    }
}
```

5.8、向 application 域共享数据

```
<h3>3.7、使用 ServletAPI 向 application 域共享数据</h3>
<a th:href="@{/test/application}">测试 map 向 application 域共享数据</a>
```

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>成功</title>
  </head>

  <body>
    <h1>success.html</h1>
    <p th:text="${testRequestScope}" />
    <p th:text="${session.testSessionScope}" />
    <p th:text="${application.testApplicationScope}" />
  </body>
</html>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.ServletContext;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpSession;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/1
 * @description
 * 向域对象共享数据:
 * 1、使用 ServletAPI 向 request 域对象共享数据
 * <p>
 * 2、使用 ModelAndView 向 request 域对象共享数据
 * 使刚 ModelAndView 时，可以使用其 Model 功能向请求域共享数据
 * 使用 View 功能设置逻辑视图，但是控制器方法一定要 ModelAndView 作为方法的返回值
 * <p>
 * 3、使用 Model 向 request 域对象共享数据
 * <p>
 * 4、使用 ModelMap 向 request 域对象共享数据
 * <p>
 * 5、使用 map 向 request 域对象共享数据
 * <p>
 * 6、Model 和 ModelMap 和 map 的关系
 * 其实在底层中，这些类型的形参最终都是通过 BindingAwareModelMap 创建
 * public class BindingAwareModelMap extends ExtendedModelMap {}
 * public class ExtendedModelMap extends ModelMap implements Model {}
 * public class ModelMap extends LinkedHashMap<String, Object> {}
 * public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V> {}
 * public interface Model {}
 */
@Controller
public class TestScopeController
{
    @RequestMapping("/test/servlet/api")
    public String testServletAPI(HttpServletRequest request)
    {
```

```
        request.setAttribute("testRequestScope", "Hello, ServletAPI");
        Object testRequestScope = request.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    /**
     * ModelAndView 包含 Model 和 view 的功能
     * Model：向请求域中共享数据
     * view：设置逻辑视图实现页面跳转
     *
     * @return 模型和视图
     */
    @RequestMapping("/test/model/and/view")
    public ModelAndView testModelAndView()
    {
        ModelAndView modelAndView = new ModelAndView();

        // 向请求域中共享数据
        modelAndView.addObject("testRequestScope", "Hello, ModelAndView");

        // 设置逻辑视图
        modelAndView.setViewName("success");

        return modelAndView;
    }

    @RequestMapping("/test/model")
    public String testModel(Model model)
    {
        // org.springframework.validation.support.BindingAwareModelMap
        System.out.println("model.getClass().getName() = " + model.getClass().getName());

        // 向请求域中共享数据
        model.addAttribute("testRequestScope", "Hello, Model");
        Object testRequestScope = model.getAttribute("testRequestScope");
        System.out.println("testRequestScope = " + testRequestScope);

        return "success";
    }

    @RequestMapping("/test/model/map")
    public String testModelMap(ModelMap modelMap)
```

```
{
    // org.springframework.validation.support.BindingAwareModelMap
    System.out.println("modelMap.getClass().getName() = " + modelMap.getClass().getName());

    // 向请求域中共享数据
    modelMap.addAttribute("testRequestScope", "Hello, ModelMap");
    Object testRequestScope = modelMap.getAttribute("testRequestScope");
    System.out.println("testRequestScope = " + testRequestScope);

    return "success";
}

@RequestMapping("/test/map")
public String testMap(Map<String, Object> map)
{
    // org.springframework.validation.support.BindingAwareModelMap
    System.out.println("map.getClass().getName() = " + map.getClass().getName());

    // 向请求域中共享数据
    map.put("testRequestScope", "Hello, map");
    Object testRequestScope = map.get("testRequestScope");
    System.out.println("testRequestScope = " + testRequestScope);

    return "success";
}

@RequestMapping("/test/session")
public String testSession(HttpSession session)
{
    session.setAttribute("testSessionScope", "Hello, Session");
    Object testSessionScope = session.getAttribute("testSessionScope");
    System.out.println("testSessionScope = " + testSessionScope);

    return "success";
}

@RequestMapping("/test/application")
public String testApplication(HttpSession session)
{
    ServletContext servletContext = session.getServletContext();
    servletContext.setAttribute("testApplicationScope", "Hello, Application");
    Object testApplicationScope = servletContext.getAttribute("testApplicationScope");
    System.out.println("testApplicationScope = " + testApplicationScope);
```

```
        return "success";
    }
}
```

6、SpringMVC 的视图

SpringMVC 中的视图是 View 接口，视图的作用渲染数据，将模型 Model 中的数据展示给用户。

SpringMVC 视图的种类很多，默认有转发视图和重定向视图。

当工程引入 jstl 的依赖，转发视图会自动转换为 JstlView。

若使用的视图技术为 Thymeleaf，在 SpringMVC 的配置文件中配置了 Thymeleaf 的视图解析器，由此视图解析器解析之后所得到的是 ThymeleafView。

6.1、ThymeleafView

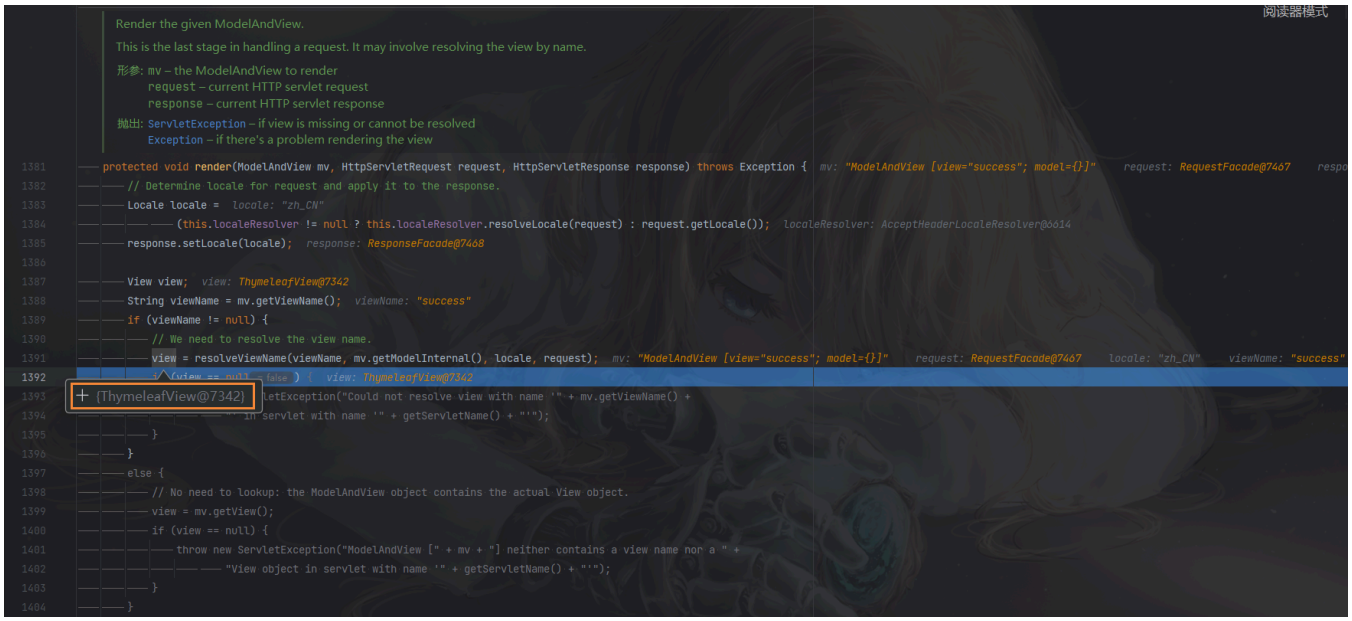
当控制器方法中所设置的视图名称没有任何前缀时，此时的视图名称会被 SpringMVC 配置文件中所配置的视图解析器解析，视图名称拼接视图前缀和视图后缀所得到的最终路径，会通过转发的方式实现跳转。

```
<h2>4、SpringMVC 的视图</h2>
<h3>4.1、ThymeleafView</h3>
<a th:href="@{/test/view/thymeleaf}">测试 SpringMVC 的视图 ThymeleafView</a>
```

```java
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @author MYXH
 * @date 2023/9/1
 */
@Controller
public class TestViewController
{
    @RequestMapping("/test/view/thymeleaf")
    public String testThymeleafView()
    {
        return "success";
    }
}
```



6.2、转发视图

SpringMVC 中默认的转发视图是 InternalResourceView。

SpringMVC 中创建转发视图的情况：

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。

例如"forward:/"，“forward:/test/model”

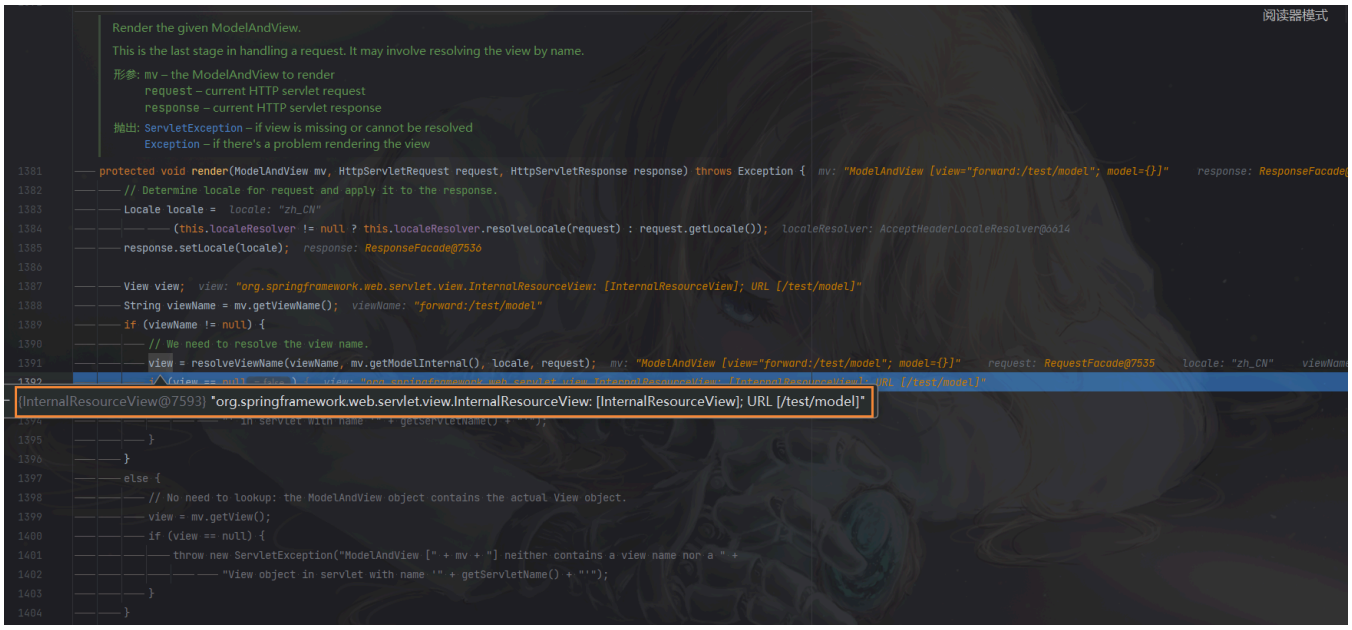
```
<h3>4.2、转发视图</h3>
<a th:href="@{/test/view/forward}"
  >测试 SpringMVC 的视图 InternalResonanceView</a
>
```

```
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @author MYXH
 * @date 2023/9/1
 */
@Controller
public class TestViewController
{
    @RequestMapping("/test/view/thymeleaf")
    public String testThymeleafView()
    {
        return "success";
    }

    @RequestMapping("/test/view/forward")
    public String testInternalResonanceView()
    {
        return "forward:/test/model";
    }
}
```



6.3、重定向视图

SpringMVC 中默认的重定向视图是 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。

例如"redirect:/"，“redirect:/test/model”

```
<h3>4.3、重定向视图</h3>
<a th:href="@{/test/view/redirect}">测试 SpringMVC 的视图 RedirectView</a>
```

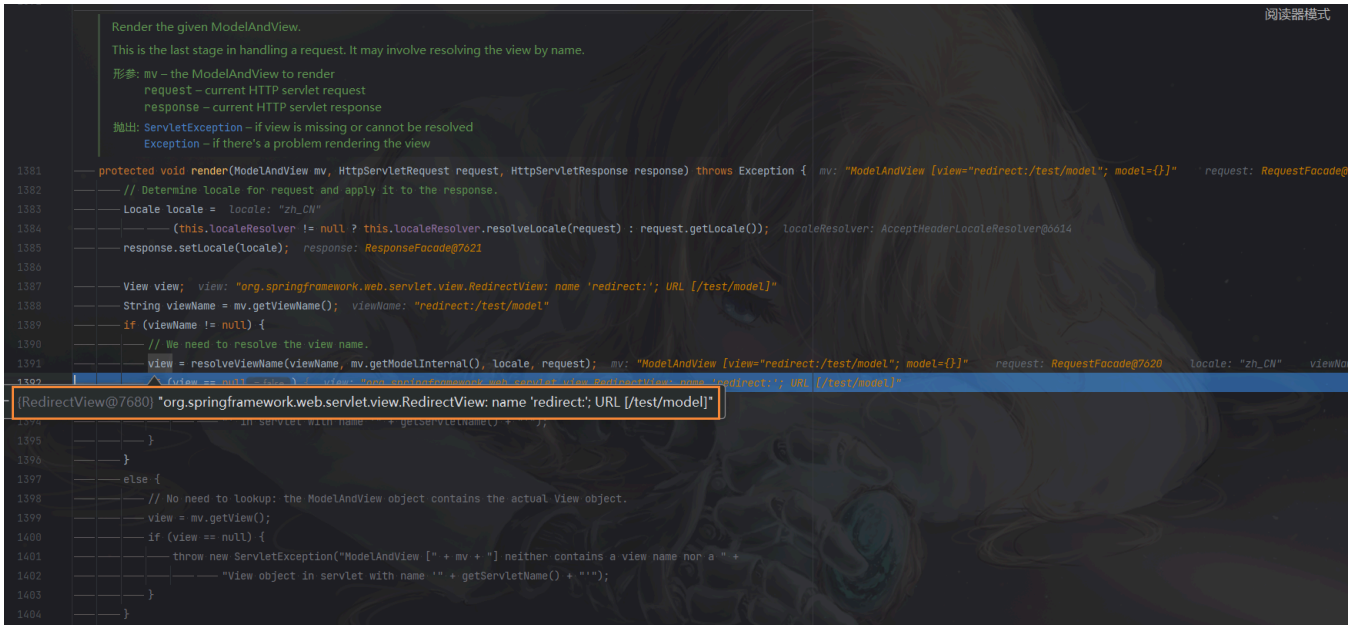
```java
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @author MYXH
 * @date 2023/9/1
 */
@Controller
public class TestViewController
{
    @RequestMapping("/test/view/thymeleaf")
    public String testThymeleafView()
    {
        return "success";
    }

    @RequestMapping("/test/view/forward")
    public String testInternalResonanceView()
    {
        return "forward:/test/model";
    }

    @RequestMapping("/test/view/redirect")
    public String testRedirectView()
    {
        return "redirect:/test/model";
    }
}
```



注意：

重定向视图在解析时，会先将 redirect:前缀去掉，然后会判断剩余部分是否以/开头，若是则会自动拼接上下文路径。

6.4、视图控制器 view-controller

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 view-controller 标签进行表示。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns="http://www.springframework.org
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.
    <!-- 扫描控制层组件 -->
    <context:component-scan base-package="com.myxh.springmvc.controller"/>

    <!-- 配置 Thymeleaf 视图解析器 -->
    <bean id="viewResolver" class="org.thymeleaf.spring6.view.ThymeleafViewResolver">
        <property name="order" value="1"/>
        <property name="characterEncoding" value="UTF-8"/>
        <property name="templateEngine">
            <bean class="org.thymeleaf.spring6.SpringTemplateEngine">
                <property name="templateResolver">
                    <bean class="org.thymeleaf.spring6.templateresolver.SpringResourceTemplateReso
                        <!-- 视图前缀 -->
                        <property name="prefix" value="/WEB-INF/templates/"/>

                        <!-- 视图后缀 -->
                        <property name="suffix" value=".html"/>

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

    <!-- 开启 mvc 的注解驱动 -->
    <mvc:annotation-driven/>

    <!--
        视图控制器：为当前的请求直接设置视图名称实现页面跳转
        若设置视图控制器，则只有视图控制器所设置的请求会被处理，其他的请求将全部 404
        此时必须在配置一个标签：<mvc:annotation-driven/>
     -->
    <mvc:view-controller path="/" view-name="index"/>
</beans>
```

> 注意：
>
> 当 SpringMVC 中设置任何一个 view-controller 时，其他控制器中的请求映射将全部失效，

> 此时需要在 SpringMVC 的核心配置文件中设置开启 mvc 注解驱动的标签：
>
> <mvc:annotation-driven/>

7、RESTful

7.1、RESTful 简介

REST：**Re**presentational **S**tate **T**ransfer，表现层资源状态转移。

7.1.1 ① 资源

资源是一种看待服务器的方式，即，将服务器看作是由很多离散的资源组成。每个资源是服务器上一个可命名的抽象概念。因为资源是一个抽象的概念，所以它不仅仅能代表服务器文件系统中的一个文件、数据库中的一张表等等具体的东西，可以将资源设计的要多抽象有多抽象，只要想象力允许而且客户端应用开发者能够理解。与面向对象设计类似，资源是以名词为核心来组织的，首先关注的是名词。一个资源可以由一个或多个 URI 来标识。URI 既是资源的名称，也是资源在 Web 上的地址。对某个资源感兴趣的客户端应用，可以通过资源的 URI 与其进行交互。

7.1.2 ② 资源的表述

资源的表述是一段对于资源在某个特定时刻的状态的描述。可以在客户端-服务器端之间转移（交换）。资源的表述可以有多种格式，例如 HTML/XML/JSON/纯文本/图片/视频/音频等等。资源的表述格式可以通过协商机制来确定。请求-响应方向的表述通常使用不同的格式。

7.1.3 ③ 状态转移

状态转移说的是：在客户端和服务器端之间转移（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。

7.2、RESTful 的实现

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。

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，以保证整体风格的一致性。

操作	传统方式	REST 风格
查询操作	getUserById?id=1	user/1 -> GET 请求方式
保存操作	saveUser	user -> POST 请求方式
删除操作	deleteUser?id=1	user/1 -> DELETE 请求方式
更新操作	updateUser	user -> PUT 请求方式

7.3、HiddenHttpMethodFilter

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**。

HiddenHttpMethodFilter 处理 put 和 delete 请求的条件：

① 当前请求的请求方式必须为 post

② 当前请求必须传输请求参数 _method

满足以上条件，**HiddenHttpMethodFilter** 过滤器就会将当前请求的请求方式转换为请求参数 _method 的值，因此请求参数_method 的值才是最终的请求方式。

在 web.xml 中注册**HiddenHttpMethodFilter**。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jakarta
         version="6.0">
    <!-- 配置 Spring 的编码过滤器 -->
    <filter>
        <filter-name>characterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>

        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>characterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- 配置处理请求方式的过滤器 -->
    <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>

    <!-- 设置 SpringMVC 的前端控制器 -->
    <servlet>
        <servlet-name>springMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

        <init-param>
            <param-name>contextConfigLocation</param-name>
```

```
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>

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

    <servlet-mapping>
        <servlet-name>springMVC</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

注意：

目前为止，SpringMVC 中提供了两个过滤器：CharacterEncodingFilter 和 HiddenHttpMethodFilter。

在 web.xml 中注册时，必须先注册 CharacterEncodingFilter，再注册 HiddenHttpMethodFilter。

原因：

- 在 CharacterEncodingFilter 中通过 request.setCharacterEncoding(encoding) 方法设置字符集的怕编码。
- request.setCharacterEncoding(encoding) 方法要求前面不能有任何获取请求参数的操作。
- 而 HiddenHttpMethodFilter 恰恰有一个获取请求方式的操作：
    - ```
      String paramValue = request.getParameter(this.methodParam);
      ```
      ```

7.4、创建页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>首页</title>
  </head>

  <body>
    <h1>index.html</h1>

    <h2>1、RESTful</h2>
    <h3>1.1、查询所有的用户信息</h3>
    <a th:href="@{/user}">查询所有的用户信息</a>

    <h3>1.2、根据 id 查询用户信息</h3>
    <form th:action="@{/user/1}">
      <input type="submit" value="查询 id 为 1 的用户信息" />
    </form>

    <h3>1.3、添加用户信息</h3>
    <form th:action="@{/user}" method="post">
      <input type="submit" value="添加用户信息" />
    </form>

    <h3>1.4、修改用户信息</h3>
    <form th:action="@{/user}" method="post">
      <input type="hidden" name="_method" value="put" />
      <input type="submit" value="修改用户信息" />
    </form>

    <h3>1.5、根据 id 删除用户信息</h3>
    <form th:action="@{/user/1}" method="post">
      <input type="hidden" name="_method" value="delete" />
      <input type="submit" value="删除 id 为 1 的用户信息" />
    </form>
  </body>
</html>
```

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>成功</title>
  </head>

  <body>
    <h1>success.html</h1>
  </body>
</html>
```

7.5、创建控制器

```java
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

/**
 * @author MYXH
 * @date 2023/9/2
 * @description
 * 查询所有的用户信息：/user -> GET
 * 根据 id 查询用户信息：/user/1 -> GET
 * 添加用户信息：/user -> POST
 * 修改用户信息：/user -> PUT
 * 根据 id 删除用户信息：/user/1 -> DELETE
 * <p>
 * 注意：浏览器目前只能发送 get 和 post 请求
 * 若要发送 put 和 delete 请求，需要在 web.xml 中配置一个过滤器 HiddenHttpMethodFilter
 * 配置了过滤器之后，发送的请求要满足两个件，才能将请求方式转换为 put 或 delete
 * 1、当前请求必须为 post
 * 2、当前请求必须传输请求参数 method，method 的值才是最终的请求方式
 */
@Controller
public class TestRestController
{
    // @RequestMapping(value = "/user", method = RequestMethod.GET)
    @GetMapping("/user")
    public String getAllUser()
    {
        System.out.println("查询所有的用户信息：/user -> GET");

        return "success";
    }

    // @RequestMapping(value = "/user/{id}", method = RequestMethod.GET)
    @GetMapping("/user/{id}")
    public String getUserById(@PathVariable("id") Integer id)
    {
        System.out.println("根据 id 查询用户信息：/user/" + id + "-> GET");

        return "success";
    }

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

```
    public String insertUser()
    {
        System.out.println("添加用户信息：/user -> POST");

        return "success";
    }

    // @RequestMapping(value = "/user", method = RequestMethod.PUT)
    @PutMapping("/user")
    public String updateUser()
    {
        System.out.println("修改用户信息：/user -> PUT");

        return "success";
    }

    // @RequestMapping(value = "/user/{id}", method = RequestMethod.DELETE)
    @DeleteMapping("/user/{id}")
    public String deleteUser(@PathVariable("id") Integer id)
    {
        System.out.println("根据 id 删除用户信息：/user/" + id + "-> DELETE");

        return "success";
    }
}
```

8、RESTful 案例

8.1、准备工作

和传统 CRUD 一样，实现对员工信息的增删改查。

- 搭建环境。
- 准备实体类。

```java
package com.myxh.springmvc.pojo;

/**
 * @author MYXH
 * @date 2023/9/2
 */
public class Employee
{
    private Integer employeeId;
    private String employeeName;
    private Integer age;
    private String gender;
    private String email;

    public Employee()
    {

    }

    public Employee(Integer employeeId, String employeeName, Integer age, String gender, String em
    {
        this.employeeId = employeeId;
        this.employeeName = employeeName;
        this.age = age;
        this.gender = gender;
        this.email = email;
    }

    public Integer getEmployeeId()
    {
        return employeeId;
    }

    public void setEmployeeId(Integer employeeId)
    {
        this.employeeId = employeeId;
    }

    public String getEmployeeName()
    {
        return employeeName;
    }

    public void setEmployeeName(String employeeName)
```

```
    {
        this.employeeName = employeeName;
    }

    public Integer getAge()
    {
        return age;
    }

    public void setAge(Integer age)
    {
        this.age = age;
    }

    public String getGender()
    {
        return gender;
    }

    public void setGender(String gender)
    {
        this.gender = gender;
    }

    public String getEmail()
    {
        return email;
    }

    public void setEmail(String email)
    {
        this.email = email;
    }

    @Override
    public String toString()
    {
        return "Employee{" +
                "employeeId=" + employeeId +
                ", employeeName='" + employeeName + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                ", email='" + email + '\'' +
                '}';
```

```
    }
}
```

- 准备 DAO 模拟数据。

```
package com.myxh.springmvc.dao;

import com.myxh.springmvc.pojo.Employee;

import java.util.Collection;

/**
 * @author MYXH
 * @date 2023/9/2
 */
public interface EmployeeDao
{
    void save(Employee employee);

    Collection<Employee> getAll();

    Employee get(Integer id);

    void delete(Integer id);
}
```

```java
package com.myxh.springmvc.dao.impl;

import com.myxh.springmvc.dao.EmployeeDao;
import com.myxh.springmvc.pojo.Employee;
import org.springframework.stereotype.Repository;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/2
 */
@Repository
public class EmployeeDaoImpl implements EmployeeDao
{
    private static final Map<Integer, Employee> employees;

    static
    {
        employees = new HashMap<>();
        employees.put(1, new Employee(1, "MYXH", 21, "男", "1735350920@qq.com"));
        employees.put(2, new Employee(2, "张三", 20, "男", "zhangsan@qq.com"));
        employees.put(3, new Employee(3, "李四", 22, "男", "lisi@qq.com"));
        employees.put(4, new Employee(4, "王五", 23, "男", "wangwu@qq.com"));
        employees.put(5, new Employee(5, "赵六", 24, "男", "zhaoliu@qq.com"));
    }

    private static Integer initId = 6;

    public void save(Employee employee)
    {
        if (employee.getEmployeeId() == null)
        {
            employee.setEmployeeId(initId++);
        }
        employees.put(employee.getEmployeeId(), employee);
    }

    public Collection<Employee> getAll()
    {
        return employees.values();
    }
```

```
    public Employee get(Integer id)
    {
        return employees.get(id);
    }

    public void delete(Integer id)
    {
        employees.remove(id);
    }
}
```

8.2、功能清单

功能	URL 地址	请求方式
访问首页 √	/	GET
查询全部数据 √	/employee	GET
删除 √	/employee/1	DELETE
跳转到添加数据页面 √	/to/add	GET
执行保存 √	/employee	POST
跳转到更新数据页面 √	/employee/1	GET
执行更新 √	/employee	PUT

8.3、具体功能：访问首页

8.3.1 ① 配置 view-controller

```
<!-- 开启 mvc 的注解驱动 -->
<mvc:annotation-driven/>

<!-- 配置视图控制器 -->
<mvc:view-controller path="/" view-name="index"/>
```

8.3.2 ② 配置 default-servlet-handler

```
<!--
    配置默认的 servlet 处理静态资源

    当前工程的 web.xml 配置的前端控制器 DispatcherServlet 的 url-pattern 是 /
    tomcat 的 web.xml 配置的 DefaultServlet 的 url-pattern 也是 /
    此时，浏览器发送的请求会优先被 DispatcherServlet 进行处理，但是 DispatcherServlet 无法处理静态资源
    若配置了 <mvc:default-servlet-handler/>，此时浏览器发送的所有请求都会被 DefaultServlet 处理
    若配置了 <mvc:default-servlet-handler/> 和 <mvc:annotation-driven/>
    浏览器发送的请求会先被 DispatcherServlet 处理，无法处理在交给 DefaultServlet 处理
    -->
<mvc:default-servlet-handler/>
```

8.3.3 ③ 创建页面

```
<h2>2、RESTful 案例</h2>
<h3>2.1、查询所有的员工信息</h3>
<a th:href="@{/employee}">查询所有的员工信息</a>
```

8.4、具体功能：查询所有员工数据

8.4.1 ① 控制器方法

```java
package com.myxh.springmvc.controller;

import com.myxh.springmvc.dao.EmployeeDao;
import com.myxh.springmvc.pojo.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import java.util.Collection;

/**
 * @author MYXH
 * @date 2023/9/2
 * @description
 * 查询所有的员工信息：/employee -> GET
 * 跳转到添加页面：/to/add -> GET
 * 添加员工信息：/employee -> POST
 * 跳转到修改页面：/employee/1 -> GET
 * 修改员工信息：/employee -> PUT
 * 根据 id 删除员工信息：/employee/1 -> DELETE
 */
@Controller
public class EmployeeController
{
    @Autowired
    private EmployeeDao employeeDao;

    @RequestMapping(value = "/employee", method = RequestMethod.GET)
    public String getAllEmployee(Model model)
    {
        // 查询所有的员工信息
        Collection<Employee> allEmployee = employeeDao.getAll();

        // 将所有的员工信息在请求域中共享
        model.addAttribute("allEmployee", allEmployee);

        // 跳转到列表页面
        return "employee_list";
    }
}
```

8.4.2 ② 创建 employee_list.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>员工列表</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}" />
  </head>

  <body>
    <table>
      <tr>
        <th colspan="6">员工列表</th>
      </tr>

      <tr>
        <th>员工 id</th>
        <th>员工姓名</th>
        <th>年龄</th>
        <th>性别</th>
        <th>电子邮件</th>
        <th>选项 (<a th:href="@{/to/add}">添加</a>) </th>
      </tr>

      <tr th:each="employee : ${allEmployee}">
        <td th:text="${employee.employeeId}"></td>
        <td th:text="${employee.employeeName}"></td>
        <td th:text="${employee.age}"></td>
        <td th:text="${employee.gender}"></td>
        <td th:text="${employee.email}"></td>

        <td>
          <a th:href="@{|/employee/${employee.employeeId}|}">修改</a>
        </td>
      </tr>
    </table>
  </body>
</html>
```

8.5、具体功能：删除

8.5.1 ① 创建处理 delete 请求方式的表单

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>员工列表</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}" />
  </head>

  <body>
    <table>
      <tr>
        <th colspan="6">员工列表</th>
      </tr>

      <tr>
        <th>员工 id</th>
        <th>员工姓名</th>
        <th>年龄</th>
        <th>性别</th>
        <th>电子邮件</th>
        <th>选项 (<a th:href="@{/to/add}">添加</a>) </th>
      </tr>

      <tr th:each="employee : ${allEmployee}">
        <td th:text="${employee.employeeId}"></td>
        <td th:text="${employee.employeeName}"></td>
        <td th:text="${employee.age}"></td>
        <td th:text="${employee.gender}"></td>
        <td th:text="${employee.email}"></td>

        <td>
          <a th:href="@{|/employee/${employee.employeeId}|}">修改</a>
        </td>
      </tr>
    </table>

    <form method="post">
      <input type="hidden" name="_method" value="delete" />
    </form>
  </body>
</html>
```

8.5.2 ② 删除超链接绑定点击事件

引入 vue.js。

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>员工列表</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}" />
  </head>

  <body>
    <table>
      <tr>
        <th colspan="6">员工列表</th>
      </tr>

      <tr>
        <th>员工 id</th>
        <th>员工姓名</th>
        <th>年龄</th>
        <th>性别</th>
        <th>电子邮件</th>
        <th>选项 (<a th:href="@{/to/add}">添加</a>) </th>
      </tr>

      <tr th:each="employee : ${allEmployee}">
        <td th:text="${employee.employeeId}"></td>
        <td th:text="${employee.employeeName}"></td>
        <td th:text="${employee.age}"></td>
        <td th:text="${employee.gender}"></td>
        <td th:text="${employee.email}"></td>

        <td>
          <a th:href="@{|/employee/${employee.employeeId}|}">修改</a>
        </td>
      </tr>
    </table>

    <form method="post">
      <input type="hidden" name="_method" value="delete" />
    </form>

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

删除超链接。

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>员工列表</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}" />
  </head>

  <body>
    <table>
      <tr>
        <th colspan="6">员工列表</th>
      </tr>

      <tr>
        <th>员工 id</th>
        <th>员工姓名</th>
        <th>年龄</th>
        <th>性别</th>
        <th>电子邮件</th>
        <th>选项 (<a th:href="@{/to/add}">添加</a>) </th>
      </tr>

      <tr th:each="employee : ${allEmployee}">
        <td th:text="${employee.employeeId}"></td>
        <td th:text="${employee.employeeName}"></td>
        <td th:text="${employee.age}"></td>
        <td th:text="${employee.gender}"></td>
        <td th:text="${employee.email}"></td>

        <td>
          <a th:href="@{|/employee/${employee.employeeId}|}">修改</a>
          <a
            @click="deleteEmployee"
            th:href="@{|/employee/${employee.employeeId}|}"
            >删除</a
          >
        </td>
      </tr>
    </table>

    <form method="post">
      <input type="hidden" name="_method" value="delete" />
    </form>
```

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

通过 vue 处理点击事件。

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>员工列表</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}" />
  </head>

  <body>
    <div id="app">
      <table>
        <tr>
          <th colspan="6">员工列表</th>
        </tr>

        <tr>
          <th>员工 id</th>
          <th>员工姓名</th>
          <th>年龄</th>
          <th>性别</th>
          <th>电子邮件</th>
          <th>选项 (<a th:href="@{/to/add}">添加</a>) </th>
        </tr>

        <tr th:each="employee : ${allEmployee}">
          <td th:text="${employee.employeeId}"></td>
          <td th:text="${employee.employeeName}"></td>
          <td th:text="${employee.age}"></td>
          <td th:text="${employee.gender}"></td>
          <td th:text="${employee.email}"></td>

          <td>
            <a th:href="@{|/employee/${employee.employeeId}|}">修改</a>
            <a
              @click="deleteEmployee"
              th:href="@{|/employee/${employee.employeeId}|}"
              >删除</a
            >
          </td>
        </tr>
      </table>

      <form method="post">
        <input type="hidden" name="_method" value="delete" />
```

```
      </form>
    </div>

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

    <script type="text/javascript">
      let vue = new Vue({
        el: "#app",

        methods: {
          deleteEmployee() {
            // 获取 form 表单
            let form = document.getElementsByTagName("form")[0];

            // 将超链接的 href 属性值赋值给 form 表单的 action 属性
            // event.target 表示当前触发事件的标签
            form.action = event.target.href;

            // 表单提交
            form.submit();

            // 阻止超链接的默认行为
            event.preventDefault();
          },
        },
      });
    </script>
  </body>
</html>
```

8.5.3 ③ 控制器方法

```java
package com.myxh.springmvc.controller;

import com.myxh.springmvc.dao.EmployeeDao;
import com.myxh.springmvc.pojo.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import java.util.Collection;

/**
 * @author MYXH
 * @date 2023/9/2
 * @description 查询所有的员工信息：/employee -> GET
 * 跳转到添加页面：/to/add -> GET
 * 添加员工信息：/employee -> POST
 * 跳转到修改页面：/employee/1 -> GET
 * 修改员工信息：/employee -> PUT
 * 根据 id 删除员工信息：/employee/1 -> DELETE
 */
@Controller
public class EmployeeController
{
    @Autowired
    private EmployeeDao employeeDao;

    @RequestMapping(value = "/employee", method = RequestMethod.GET)
    public String getAllEmployee(Model model)
    {
        // 查询所有的员工信息
        Collection<Employee> allEmployee = employeeDao.getAll();

        // 将所有的员工信息在请求域中共享
        model.addAttribute("allEmployee", allEmployee);

        // 跳转到列表页面
        return "employee_list";
    }

    @RequestMapping(value = "employee/{employeeId}", method = RequestMethod.DELETE)
    public String deleteEmployee(@PathVariable("employeeId") Integer employeeId)
```

```
    {
        // 根据 id 删除员工信息
        employeeDao.delete(employeeId);

        // 重定向到列表功能: /employee
        return "redirect:/employee";
    }
}
```

8.6、具体功能：跳转到添加数据页面

8.6.1 ① 配置 view-controller

```
<!-- 开启 mvc 的注解驱动 -->
<mvc:annotation-driven/>

<!-- 配置视图控制器 -->
<mvc:view-controller path="/" view-name="index"/>
<mvc:view-controller path="/to/add" view-name="employee_add"/>
```

8.6.2 ② 创建 employee_add.html

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>添加员工</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}" />
  </head>

  <body>
    <form th:action="@{/employee}" method="post">
      <table>
        <tr>
          <th colspan="2">添加员工</th>
        </tr>

        <tr>
          <td>员工姓名</td>
          <td>
            <label>
              <input type="text" name="employeeName" />
            </label>
          </td>
        </tr>

        <tr>
          <td>年龄</td>
          <td>
            <label>
              <input type="text" name="age" />
            </label>
          </td>
        </tr>

        <tr>
          <td>性别</td>
          <td>
            <label>
              <input type="radio" name="gender" value="男" />
              男
            </label>
            <label>
              <input type="radio" name="gender" value="女" />
              女
            </label>
```

```
          </td>
        </tr>

        <tr>
          <td>电子邮件</td>
          <td>
            <label>
              <input type="text" name="email" />
            </label>
          </td>
        </tr>

        <tr>
          <td colspan="2">
            <input type="submit" value="添加" />
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

8.7、具体功能：执行保存

8.7.1 ① 控制器方法

```java
package com.myxh.springmvc.controller;

import com.myxh.springmvc.dao.EmployeeDao;
import com.myxh.springmvc.pojo.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import java.util.Collection;

/**
 * @author MYXH
 * @date 2023/9/2
 * @description 查询所有的员工信息：/employee -> GET
 * 跳转到添加页面：/to/add -> GET
 * 添加员工信息：/employee -> POST
 * 跳转到修改页面：/employee/1 -> GET
 * 修改员工信息：/employee -> PUT
 * 根据 id 删除员工信息：/employee/1 -> DELETE
 */
@Controller
public class EmployeeController
{
    @Autowired
    private EmployeeDao employeeDao;

    @RequestMapping(value = "/employee", method = RequestMethod.GET)
    public String getAllEmployee(Model model)
    {
        // 查询所有的员工信息
        Collection<Employee> allEmployee = employeeDao.getAll();

        // 将所有的员工信息在请求域中共享
        model.addAttribute("allEmployee", allEmployee);

        // 跳转到列表页面
        return "employee_list";
    }

    @RequestMapping(value = "/employee", method = RequestMethod.POST)
    public String addEmployee(Employee employee)
```

```
    {
        // 保存员工信息
        employeeDao.save(employee);

        // 重定向到列表功能：/employee
        return "redirect:/employee";
    }

    @RequestMapping(value = "employee/{employeeId}", method = RequestMethod.DELETE)
    public String deleteEmployee(@PathVariable("employeeId") Integer employeeId)
    {
        // 根据 id 删除员工信息
        employeeDao.delete(employeeId);

        // 重定向到列表功能：/employee
        return "redirect:/employee";
    }
}
```

8.8、具体功能：跳转到更新数据页面

8.8.1 ① 修改超链接

```
<a th:href="@{|/employee/${employee.employeeId}|}">修改</a>
```

8.8.2 ② 控制器方法

```java
package com.myxh.springmvc.controller;

import com.myxh.springmvc.dao.EmployeeDao;
import com.myxh.springmvc.pojo.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import java.util.Collection;

/**
 * @author MYXH
 * @date 2023/9/2
 * @description 查询所有的员工信息：/employee -> GET
 * 跳转到添加页面：/to/add -> GET
 * 添加员工信息：/employee -> POST
 * 跳转到修改页面：/employee/1 -> GET
 * 修改员工信息：/employee -> PUT
 * 根据 id 删除员工信息：/employee/1 -> DELETE
 */
@Controller
public class EmployeeController
{
    @Autowired
    private EmployeeDao employeeDao;

    @RequestMapping(value = "/employee", method = RequestMethod.GET)
    public String getAllEmployee(Model model)
    {
        // 查询所有的员工信息
        Collection<Employee> allEmployee = employeeDao.getAll();

        // 将所有的员工信息在请求域中共享
        model.addAttribute("allEmployee", allEmployee);

        // 跳转到列表页面
        return "employee_list";
    }

    @RequestMapping(value = "/employee", method = RequestMethod.POST)
    public String addEmployee(Employee employee)
```

```
    {
        // 保存员工信息
        employeeDao.save(employee);

        // 重定向到列表功能: /employee
        return "redirect:/employee";
    }

    @RequestMapping(value = "employee/{employeeId}", method = RequestMethod.GET)
    public String toUpdate(Model model,@PathVariable("employeeId") Integer employeeId)
    {
        // 根据 id 查询员工信息
        Employee employee = employeeDao.get(employeeId);

        // 将员工信息在请求域中共享
        model.addAttribute("employee", employee);

        // 跳转到修改页面
        return "employee_update";
    }

    @RequestMapping(value = "employee/{employeeId}", method = RequestMethod.DELETE)
    public String deleteEmployee(@PathVariable("employeeId") Integer employeeId)
    {
        // 根据 id 删除员工信息
        employeeDao.delete(employeeId);

        // 重定向到列表功能: /employee
        return "redirect:/employee";
    }
}
```

8.8.3 ③ 创建 employee_update.html

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>修改员工</title>
    <link rel="stylesheet" th:href="@{/static/css/index_work.css}" />
  </head>

  <body>
    <form th:action="@{/employee}" method="post">
      <input type="hidden" name="_method" value="put" />
      <input
        type="hidden"
        name="employeeId"
        th:value="${employee.employeeId}"
      />

      <table>
        <tr>
          <th colspan="2">修改员工</th>
        </tr>

        <tr>
          <td>员工姓名</td>
          <td>
            <label>
              <input
                type="text"
                name="employeeName"
                th:value="${employee.employeeName}"
              />
            </label>
          </td>
        </tr>

        <tr>
          <td>年龄</td>
          <td>
            <label>
              <input type="text" name="age" th:value="${employee.age}" />
            </label>
          </td>
        </tr>
```

```
        <tr>
          <td>性别</td>
          <td>
            <label>
              <input
                type="radio"
                name="gender"
                value="男"
                th:field="${employee.gender}"
              />
              男
            </label>
            <label>
              <input
                type="radio"
                name="gender"
                value="女"
                th:field="${employee.gender}"
              />
              女
            </label>
          </td>
        </tr>

        <tr>
          <td>电子邮件</td>
          <td>
            <label>
              <input type="text" name="email" th:value="${employee.email}" />
            </label>
          </td>
        </tr>

        <tr>
          <td colspan="2">
            <input type="submit" value="修改" />
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

8.9、具体功能：执行更新

8.9.1 ① 控制器方法

```java
package com.myxh.springmvc.controller;

import com.myxh.springmvc.dao.EmployeeDao;
import com.myxh.springmvc.pojo.Employee;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import java.util.Collection;

/**
 * @author MYXH
 * @date 2023/9/2
 * @description 查询所有的员工信息：/employee -> GET
 * 跳转到添加页面：/to/add -> GET
 * 添加员工信息：/employee -> POST
 * 跳转到修改页面：/employee/1 -> GET
 * 修改员工信息：/employee -> PUT
 * 根据 id 删除员工信息：/employee/1 -> DELETE
 */
@Controller
public class EmployeeController
{
    @Autowired
    private EmployeeDao employeeDao;

    @RequestMapping(value = "/employee", method = RequestMethod.GET)
    public String getAllEmployee(Model model)
    {
        // 查询所有的员工信息
        Collection<Employee> allEmployee = employeeDao.getAll();

        // 将所有的员工信息在请求域中共享
        model.addAttribute("allEmployee", allEmployee);

        // 跳转到列表页面
        return "employee_list";
    }

    @RequestMapping(value = "/employee", method = RequestMethod.POST)
    public String addEmployee(Employee employee)
```

```
    {
        // 保存员工信息
        employeeDao.save(employee);

        // 重定向到列表功能: /employee
        return "redirect:/employee";
    }

    @RequestMapping(value = "employee/{employeeId}", method = RequestMethod.GET)
    public String toUpdate(Model model,@PathVariable("employeeId") Integer employeeId)
    {
        // 根据 id 查询员工信息
        Employee employee = employeeDao.get(employeeId);

        // 将员工信息在请求域中共享
        model.addAttribute("employee", employee);

        // 跳转到修改页面
        return "employee_update";
    }

    @RequestMapping(value = "/employee", method = RequestMethod.PUT)
    public String updateEmployee(Employee employee)
    {
        // 修改员工信息
        employeeDao.save(employee);

        // 重定向到列表功能: /employee
        return "redirect:/employee";
    }

    @RequestMapping(value = "employee/{employeeId}", method = RequestMethod.DELETE)
    public String deleteEmployee(@PathVariable("employeeId") Integer employeeId)
    {
        // 根据 id 删除员工信息
        employeeDao.delete(employeeId);

        // 重定向到列表功能: /employee
        return "redirect:/employee";
    }
}
```

9、SpringMVC 处理 ajax 请求

9.1、@RequestBody

@RequestBody 可以获取请求体信息，使用@RequestBody 注解标识控制器方法的形参，当前请求的请求体就会为当前注解所标识的形参赋值。

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>首页</title>
  </head>

  <body>
    <div id="app">
      <h1>index.html</h1>

      <h2>1、SpringMVC 处理 ajax 请求</h2>
      <h3>1.1、SpringMVC 处理 ajax 请求</h3>
      <input
        type="button"
        value="测试 SpringMVC 处理 ajax 请求"
        @click="testAjax"
      />
    </div>

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

    <script type="text/javascript">
      /**
       *     axios({
       *         // 请求路径
       *         url:"",
       *
       *         // 请求方式
       *         method:"",
       *
       *         // 以 name=value&name=value 的方式发送的请求参数
       *         // 不管使用的请方式是 get 或 post，请参数都会被拼接到请求地址后
       *         // 此种方式的请求参数可以通过 request.getParameter() 获取
       *         params:{},
       *
       *         // 以 json 格式发送的请求参数
       *         // 请求参数会被保存到请求报文的请求体传输到服务器
       *         // 此种方式的请求参数可以不通过 request.getParameter() 获取
       *         data:{}
       *     }).then(response=>{
       *         console.log(response.data);
       *     });
```

```
      */
      let vue = new Vue({
        el: "#app",

        methods: {
          testAjax() {
            axios
              .post("/spring_mvc_ajax/test/ajax?id=1", {
                username: "MYXH",
                password: "520.ILY!",
              })
              .then((response) => {
                console.log(response.data);
              });
          },
        },
      });
    </script>
  </body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>成功</title>
  </head>

  <body>
    <h1>success.html</h1>
  </body>
</html>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

import java.io.IOException;

/**
 * @author MYXH
 * @date 2023/9/6
 * @description
 * 1、@RequestBody：将请求体中的内容和控制器方法的形参进行绑定
 */
@Controller
public class TestAjaxController
{
    @RequestMapping("/test/ajax")
    public void testAjax(@RequestBody String requestBody, HttpServletResponse response, Integer id
    {
        System.out.println("requestBody = " + requestBody);
        System.out.println("id = " + id);
        response.getWriter().write("Hello, axios");
    }
}
```

输出结果：

requestBody = {“username”:“MYXH”,“password”:“520.lLY!”}
id = 1

9.2、@RequestBody 获取 json 格式的请求参数

在使用了 axios 发送 ajax 请求之后，浏览器发送到服务器的请求参数有两种格式：

1、name=value&name=value …，此时的请求参数可以通过 request.getParameter() 获取，对应 SpringMVC 中，可以直接通过控制器方法的形参获取此类请求参数。

2、{key:value,key:value, …}，此时无法通过 request.getParameter()获取，之前我们使用操作 json 的相关 jar 包 gson 或 jackson 处理此类请求参数，可以将其转换为指定的实体类对象或 map 集合。在 SpringMVC 中，直接使用 @RequestBody 注解标识控制器方法的形

> 参即可将此类请求参数转换为 Java 对象。

使用@RequestBody 获取 json 格式的请求参数的条件：

1、导入 jackson 的依赖。

```
<!-- jackson -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.15.1</version>
</dependency>
```

2、SpringMVC 的配置文件中设置开启 mvc 的注解驱动。

```
<!-- 开启 mvc 的注解驱动 -->
<mvc:annotation-driven/>
```

3、在控制器方法的形参位置，设置 json 格式的请求参数要转换成的 Java 类型（实体类或 map）的参数，并使用 @RequestBody 注解标识。

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>首页</title>
  </head>

  <body>
    <div id="app">
      <h1>index.html</h1>

      <h2>1、SpringMVC 处理 ajax 请求</h2>
      <h3>1.1、SpringMVC 处理 ajax 请求</h3>
      <input
        type="button"
        value="测试 SpringMVC 处理 ajax 请求"
        @click="testAjax"
      />

      <h3>1.2、@RequestBody 获取 json 格式的请求参数</h3>
      <input
        type="button"
        value="测试 @RequestBody 注解处理 json 格式的请求参数"
        @click="testRequestBody"
      />
    </div>

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

    <script type="text/javascript">
      /**
       *     axios({
       *         // 请求路径
       *         url:"",
       *
       *         // 请求方式
       *         method:"",
       *
       *         // 以 name=value&name=value 的方式发送的请求参数
       *         // 不管使用的请方式是 get 或 post，请参数都会被拼接到请求地址后
       *         // 此种方式的请求参数可以通过 request.getParameter() 获取
       *         params:{},
       *
```

```
         *          // 以 json 格式发送的请求参数
         *          // 请求参数会被保存到请求报文的请求体传输到服务器
         *          // 此种方式的请求参数可以不通过 request.getParameter() 获取
         *          data:{}
         *      }).then(response=>{
         *          console.log(response.data);
         *      });
         */
        let vue = new Vue({
          el: "#app",

          methods: {
            testAjax() {
              axios
                .post("/spring_mvc_ajax/test/ajax?id=1", {
                  username: "MYXH",
                  password: "520.ILY!",
                })
                .then((response) => {
                  console.log(response.data);
                });
            },

            testRequestBody() {
              axios
                .post("/spring_mvc_ajax/test/requestBody/json", {
                  id: "1",
                  username: "MYXH",
                  password: "520.ILY!",
                  age: "21",
                  gender: "男",
                  email: "1735350920@qq.com",
                })
                .then((response) => {
                  console.log(response.data);
                });
            },
          },
        });
      </script>
    </body>
</html>
```

```java
package com.myxh.springmvc.controller;

import com.myxh.springmvc.pojo.User;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;

import java.io.IOException;
import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/6
 * @description
 * 1、@RequestBody：将请求体中的内容和控制器方法的形参进行绑定
 * 2、使用 @RequestBody 注解将 json 格式的请求参数转换为 Java 对象
 * ① 导入 jackson 的依赖
 * ② 在 SpringMVC 的配置文件中设置 <mvc:annotation-driven/>
 * ③ 在处理请求的控制器方法的形参位置，直接设置 json 格式的请求参数要转换的 Java 类型的形参，使用 @Reques
 */
@Controller
public class TestAjaxController
{
    @RequestMapping("/test/ajax")
    public void testAjax(@RequestBody String requestBody, HttpServletResponse response, Integer id
    {
        System.out.println("requestBody = " + requestBody);
        System.out.println("id = " + id);
        response.getWriter().write("Hello, axios");
    }

    @RequestMapping("/test/requestBody/json")
    public void testRequestBody(@RequestBody User user, HttpServletResponse response) throws IOExc
    {
        System.out.println("user = " + user);
        response.getWriter().write("Hello, requestBody");
    }

    // @RequestMapping("/test/requestBody/json")
    public void testRequestBody(@RequestBody Map<String, Object> map, HttpServletResponse response
    {
        System.out.println("map = " + map);
        response.getWriter().write("Hello, requestBody");
```

```
    }
}
```

9.3、@ResponseBody

@ResponseBody 用于标识一个控制器方法，可以将该方法的返回值直接作为响应报文的响应体响应到浏览器。

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>首页</title>
  </head>

  <body>
    <div id="app">
      <h1>index.html</h1>

      <h2>1、SpringMVC 处理 ajax 请求</h2>
      <h3>1.1、SpringMVC 处理 ajax 请求</h3>
      <input
        type="button"
        value="测试 SpringMVC 处理 ajax 请求"
        @click="testAjax"
      />

      <h3>1.2、@RequestBody 获取 json 格式的请求参数</h3>
      <input
        type="button"
        value="测试 @RequestBody 注解处理 json 格式的请求参数"
        @click="testRequestBody"
      />

      <h3>1.3、@ResponseBody 响应浏览器数据</h3>
      <a th:href="@{/test/responseBody}"
        >测试 @ResponseBody 注解响应浏览器数据</a
      >
    </div>

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

    <script type="text/javascript">
      /**
       *     axios({
       *         // 请求路径
       *         url:"",
       *
       *         // 请求方式
       *         method:"",
       *
```

```
 *          // 以 name=value&name=value 的方式发送的请求参数
 *          // 不管使用的请方式是 get 或 post，请参数都会被拼接到请求地址后
 *          // 此种方式的请求参数可以通过 request.getParameter() 获取
 *          params:{},
 *
 *          // 以 json 格式发送的请求参数
 *          // 请求参数会被保存到请求报文的请求体传输到服务器
 *          // 此种方式的请求参数可以不通过 request.getParameter() 获取
 *          data:{}
 *      }).then(response=>{
 *          console.log(response.data);
 *      });
 */
let vue = new Vue({
  el: "#app",

  methods: {
    testAjax() {
      axios
        .post("/spring_mvc_ajax/test/ajax?id=1", {
          username: "MYXH",
          password: "520.ILY!",
        })
        .then((response) => {
          console.log(response.data);
        });
    },

    testRequestBody() {
      axios
        .post("/spring_mvc_ajax/test/requestBody/json", {
          id: "1",
          username: "MYXH",
          password: "520.ILY!",
          age: "21",
          gender: "男",
          email: "1735350920@qq.com",
        })
        .then((response) => {
          console.log(response.data);
        });
    },
  },
});
```

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

```java
package com.myxh.springmvc.controller;

import com.myxh.springmvc.pojo.User;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import java.io.IOException;
import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/6
 * @description
 * 1、@RequestBody：将请求体中的内容和控制器方法的形参进行绑定
 * 2、使用 @RequestBody 注解将 json 格式的请求参数转换为 Java 对象
 * ① 导入 jackson 的依赖
 * ② 在 SpringMVC 的配置文件中设置 <mvc:annotation-driven/>
 * ③ 在处理请求的控制器方法的形参位置，直接设置 json 格式的请求参数要转换的 Java 类型的形参，使用 @Reques
 * 3、@ResponseBody：将所标识的控制器方法的返回值作为响应报文的响应体响应到浏览器
 */
@Controller
public class TestAjaxController
{
    @RequestMapping("/test/ajax")
    public void testAjax(@RequestBody String requestBody, HttpServletResponse response, Integer id
    {
        System.out.println("requestBody = " + requestBody);
        System.out.println("id = " + id);
        response.getWriter().write("Hello, axios");
    }

    @RequestMapping("/test/requestBody/json")
    public void testRequestBody(@RequestBody User user, HttpServletResponse response) throws IOExc
    {
        System.out.println("user = " + user);
        response.getWriter().write("Hello, requestBody");
    }

    // @RequestMapping("/test/requestBody/json")
    public void testRequestBody(@RequestBody Map<String, Object> map, HttpServletResponse response
    {
```

```
        System.out.println("map = " + map);
        response.getWriter().write("Hello, requestBody");
    }

    @RequestMapping("/test/responseBody")
    @ResponseBody
    public String testResponseBody()
    {
        return "Hello, responseBody";
    }
}
```

9.4、@ResponseBody 响应浏览器 json 数据

服务器处理 ajax 请求之后，大多数情况都需要向浏览器响应一个 Java 对象，此时必须将 Java 对象转换为 json 字符串才可以响应到浏览器，之前我们使用操作 json 数据的 jar 包 gson 或 jackson 将 java 对象转换为 json 字符串。在 SpringMVC 中，我们可以直接使用 @ResponseBody 注解实现此功能。

@ResponseBody 响应浏览器 json 数据的条件：

1、导入 jackson 的依赖。

```
<!-- jackson -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.15.1</version>
</dependency>
```

2、SpringMVC 的配置文件中设置开启 mvc 的注解驱动。

```
<!-- 开启 mvc 的注解驱动 -->
<mvc:annotation-driven/>
```

3、使用@ResponseBody 注解标识控制器方法，在方法中，将需要转换为 json 字符串并响应到浏览器的 Java 对象作为控制器方法的返回值，此时 SpringMVC 就可以将此对象直接转换为 json 字符串并响应到浏览器。

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>首页</title>
  </head>

  <body>
    <div id="app">
      <h1>index.html</h1>

      <h2>1、SpringMVC 处理 ajax 请求</h2>
      <h3>1.1、SpringMVC 处理 ajax 请求</h3>
      <input
        type="button"
        value="测试 SpringMVC 处理 ajax 请求"
        @click="testAjax"
      />

      <h3>1.2、@RequestBody 获取 json 格式的请求参数</h3>
      <input
        type="button"
        value="测试 @RequestBody 注解处理 json 格式的请求参数"
        @click="testRequestBody"
      />

      <h3>1.3、@ResponseBody 响应浏览器数据</h3>
      <a th:href="@{/test/responseBody}"
        >测试 @ResponseBody 注解响应浏览器数据</a
      >

      <h3>1.4、@ResponseBody 响应浏览器 json 数据</h3>
      <input
        type="button"
        value="测试 @ResponseBody 注解响应浏览器 json 数据"
        @click="testResponseBody"
      />
    </div>

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

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

```
 *     axios({
 *         // 请求路径
 *         url:"",
 *
 *         // 请求方式
 *         method:"",
 *
 *         // 以 name=value&name=value 的方式发送的请求参数
 *         // 不管使用的请方式是 get 或 post, 请参数都会被拼接到请求地址后
 *         // 此种方式的请求参数可以通过 request.getParameter() 获取
 *         params:{},
 *
 *         // 以 json 格式发送的请求参数
 *         // 请求参数会被保存到请求报文的请求体传输到服务器
 *         // 此种方式的请求参数可以不通过 request.getParameter() 获取
 *         data:{}
 *     }).then(response=>{
 *         console.log(response.data);
 *     });
 */
let vue = new Vue({
  el: "#app",

  methods: {
    testAjax() {
      axios
        .post("/spring_mvc_ajax/test/ajax?id=1", {
          username: "MYXH",
          password: "520.ILY!",
        })
        .then((response) => {
          console.log(response.data);
        });
    },

    testRequestBody() {
      axios
        .post("/spring_mvc_ajax/test/requestBody/json", {
          id: "1",
          username: "MYXH",
          password: "520.ILY!",
          age: "21",
          gender: "男",
          email: "1735350920@qq.com",
```

```
            })
            .then((response) => {
              console.log(response.data);
            });
        },

        testResponseBody() {
          axios
            .post("/spring_mvc_ajax/test/responseBody/json")
            .then((response) => {
              console.log(response.data);
            });
        },
      },
    });
  </script>
</body>
</html>
```

```java
package com.myxh.springmvc.controller;

import com.myxh.springmvc.pojo.User;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import java.io.IOException;
import java.util.Map;

/**
 * @author MYXH
 * @date 2023/9/6
 * @description
 * 1、@RequestBody：将请求体中的内容和控制器方法的形参进行绑定
 * 2、使用 @RequestBody 注解将 json 格式的请求参数转换为 Java 对象
 * ① 导入 jackson 的依赖
 * ② 在 SpringMVC 的配置文件中设置 <mvc:annotation-driven/>
 * ③ 在处理请求的控制器方法的形参位置，直接设置 json 格式的请求参数要转换的 Java 类型的形参，使用 @Reques
 * 3、@ResponseBody：将所标识的控制器方法的返回值作为响应报文的响应体响应到浏览器
 * 4、使用 @ResponseBody 注解响应浏览器 json 格式的数据
 * ① 导入 jackson 的依赖
 * ② 在 SpringMVC 的配置文件中设置 <mvc:annotation-driven/>
 * ③ 将需要转换为 json 字符串的 Java 对象直接作为控制器方法的返回值，使用 @ResponseBody 注解标识控制器方
 * 就可以将 Java 对象直接转换为 json 字符串，并响应到浏览器
 * <p>
 * 常用的 Java 对象转换为 json 的结果：
 * 实体类 -> json 对象
 * Map -> json 对象
 * List -> json 数组
 */
@Controller
// @RestController 相当于 @Controller + @ResponseBody
// @RestController
public class TestAjaxController
{
    @RequestMapping("/test/ajax")
    public void testAjax(@RequestBody String requestBody, HttpServletResponse response, Integer id
    {
        System.out.println("requestBody = " + requestBody);
        System.out.println("id = " + id);
        response.getWriter().write("Hello, axios");
```

```java
    }

    @RequestMapping("/test/requestBody/json")
    public void testRequestBody(@RequestBody User user, HttpServletResponse response) throws IOExc
    {
        System.out.println("user = " + user);
        response.getWriter().write("Hello, requestBody");
    }

    // @RequestMapping("/test/requestBody/json")
    public void testRequestBody(@RequestBody Map<String, Object> map, HttpServletResponse response
    {
        System.out.println("map = " + map);
        response.getWriter().write("Hello, requestBody");
    }

    @RequestMapping("/test/responseBody")
    @ResponseBody
    public String testResponseBody()
    {
        return "Hello, responseBody";
    }

    @RequestMapping("/test/responseBody/json")
    @ResponseBody
    public User testResponseBodyJson()
    {
        User user = new User(1, "MYXH", "520.ILY!", 21, "男", "1735350920@qq.com");

        return user;
    }

    @RequestMapping("/test/responseBody/json")
    @ResponseBody
    public List<User> testResponseBodyJson()
    {
        User user1 = new User(1, "MYXH", "520.ILY!", 21, "男", "1735350920@qq.com");
        User user2 = new User(2, "root", "000000", 21, "男", "root@qq.com");
        User user3 = new User(3, "admin", "123456", 21, "男", "admin@qq.com");
        User user4 = new User(4, "test", "test", 18, "男", "test@qq.com");
        List<User> list = Arrays.asList(user1, user2, user3, user4);

        return list;
    }
```

```
    @RequestMapping("/test/responseBody/json")
    @ResponseBody
    public Map<String , Object> testResponseBodyJson()
    {
        User user1 = new User(1, "MYXH", "520.ILY!", 21, "男", "1735350920@qq.com");
        User user2 = new User(2, "root", "000000", 21, "男", "root@qq.com");
        User user3 = new User(3, "admin", "123456", 21, "男", "admin@qq.com");
        User user4 = new User(4, "test", "test", 18, "男", "test@qq.com");
        Map<String, Object> map = new HashMap<>();
        map.put("1", user1);
        map.put("2", user2);
        map.put("3", user3);
        map.put("4", user4);

        return map;
    }
}
```

9.5、@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。

10、文件上传和下载

10.1、文件下载

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。

使用 ResponseEntity 实现下载文件的功能。

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>首页</title>
  </head>

  <body>
    <div id="app">
      <h1>index.html</h1>

      <h2>1、SpringMVC 处理 ajax 请求</h2>
      <h3>1.1、SpringMVC 处理 ajax 请求</h3>
      <input
        type="button"
        value="测试 SpringMVC 处理 ajax 请求"
        @click="testAjax"
      />

      <h3>1.2、@RequestBody 获取 json 格式的请求参数</h3>
      <input
        type="button"
        value="测试 @RequestBody 注解处理 json 格式的请求参数"
        @click="testRequestBody"
      />

      <h3>1.3、@ResponseBody 响应浏览器数据</h3>
      <a th:href="@{/test/responseBody}"
        >测试 @ResponseBody 注解响应浏览器数据</a
      >

      <h3>1.4、@ResponseBody 响应浏览器 json 数据</h3>
      <input
        type="button"
        value="测试 @ResponseBody 注解响应浏览器 json 数据"
        @click="testResponseBody"
      />

      <hr />

      <h2>2、文件上传和下载</h2>
      <h3>2.1、下载图片</h3>
      <a th:href="@{/test/download}">下载图片</a>
    </div>
```

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

<script type="text/javascript">
  /**
   *     axios({
   *         // 请求路径
   *         url:"",
   *
   *         // 请求方式
   *         method:"",
   *
   *         // 以 name=value&name=value 的方式发送的请求参数
   *         // 不管使用的请方式是 get 或 post, 请参数都会被拼接到请求地址后
   *         // 此种方式的请求参数可以通过 request.getParameter() 获取
   *         params:{},
   *
   *         // 以 json 格式发送的请求参数
   *         // 请求参数会被保存到请求报文的请求体传输到服务器
   *         // 此种方式的请求参数可以不通过 request.getParameter() 获取
   *         data:{}
   *     }).then(response=>{
   *         console.log(response.data);
   *     });
   */
  let vue = new Vue({
    el: "#app",

    methods: {
      testAjax() {
        axios
          .post("/spring_mvc_ajax/test/ajax?id=1", {
            username: "MYXH",
            password: "520.ILY!",
          })
          .then((response) => {
            console.log(response.data);
          });
      },

      testRequestBody() {
        axios
          .post("/spring_mvc_ajax/test/requestBody/json", {
```

```
              id: "1",
              username: "MYXH",
              password: "520.ILY!",
              age: "21",
              gender: "男",
              email: "1735350920@qq.com",
            })
            .then((response) => {
              console.log(response.data);
            });
        },

        testResponseBody() {
          axios
            .post("/spring_mvc_ajax/test/responseBody/json")
            .then((response) => {
              console.log(response.data);
            });
        },
      },
    });
  </script>
</body>
</html>
```

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.ServletContext;
import jakarta.servlet.http.HttpSession;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URLEncoder;
import java.nio.charset.StandardCharsets;

/**
 * @author MYXH
 * @date 2023/9/6
 * @description
 * ResponseEntity：可以作为控制器方法的返回值，表示响应到浏览器的完整的响应报文
 * <p>
 * 文件上传的要求：
 * 1、form 表单的请求方式必须为 post
 * 2、form 表单必须设置属性 enctype="multipart/form-data"
 */
@Controller
public class FileUploadAndDownloadController
{
    @RequestMapping("/test/download")
    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/大户爱.png");
        String realPath = servletContext.getRealPath(File.separator + "static" + File.separator +

        // 创建输入流
        InputStream inputStream = new FileInputStream(realPath);

        // 创建字节数组，is.available() 获取输入流所对应文件的字节数
```

```
        byte[] bytes = new byte[inputStream.available()];

        // 将流读到字节数组中
        inputStream.read(bytes);

        // 创建 HttpHeaders 对象设置响应头信息
        MultiValueMap<String, String> headers = new HttpHeaders();

        // 对文件名进行 URL 编码
        String filename = "大户爱.png";
        String encodedFileName = URLEncoder.encode(filename, StandardCharsets.UTF_8);

        // 设置要下载方式以及下载文件的名字
        headers.add("Content-Disposition", "attachment; filename*=UTF-8''" + encodedFileName);

        // 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
        inputStream.close();

        return responseEntity;
    }
}
```

10.2、文件上传

文件上传要求 form 表单的请求方式必须为 post，并且添加属性 enctype=“multipart/form-data”。

SpringMVC 中将上传的文件封装到 MultipartFile 对象中，通过此对象可以获取文件相关信息。

上传步骤：

10.2.1 ① 在 SpringMVC 的配置文件中添加配置：

```
<!-- 开启 mvc 的注解驱动 -->
<mvc:annotation-driven/>

<!-- 配置文件上传解析器 -->
<!-- 配置 StandardServletMultipartResolver 为 MultipartResolver -->
<!-- 该类实现了 Servlet3.0 规范的文件上传支持 -->
<bean id="multipartResolver" class="org.springframework.web.multipart.support.StandardServletMulti
```

10.2.2 ② 创建页面：

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" />
    <title>首页</title>
  </head>

  <body>
    <div id="app">
      <h1>index.html</h1>

      <h2>1、SpringMVC 处理 ajax 请求</h2>
      <h3>1.1、SpringMVC 处理 ajax 请求</h3>
      <input
        type="button"
        value="测试 SpringMVC 处理 ajax 请求"
        @click="testAjax"
      />

      <h3>1.2、@RequestBody 获取 json 格式的请求参数</h3>
      <input
        type="button"
        value="测试 @RequestBody 注解处理 json 格式的请求参数"
        @click="testRequestBody"
      />

      <h3>1.3、@ResponseBody 响应浏览器数据</h3>
      <a th:href="@{/test/responseBody}"
        >测试 @ResponseBody 注解响应浏览器数据</a
      >

      <h3>1.4、@ResponseBody 响应浏览器 json 数据</h3>
      <input
        type="button"
        value="测试 @ResponseBody 注解响应浏览器 json 数据"
        @click="testResponseBody"
      />

      <hr />

      <h2>2、文件上传和下载</h2>
      <h3>2.1、下载图片</h3>
      <a th:href="@{/test/download}">下载图片</a>
```

```
  <h2>2.2、上传图片</h2>
  <form
    th:action="@{/test/upload}"
    method="post"
    enctype="multipart/form-data"
  >
    头像<input
      type="file"
      id="picture"
      name="picture"
      required
      accept="image/*"
    /><br />
    <input type="submit" value="上传" />
  </form>
</div>

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

<script type="text/javascript">
  /**
   *     axios({
   *         // 请求路径
   *         url:"",
   *
   *         // 请求方式
   *         method:"",
   *
   *         // 以 name=value&name=value 的方式发送的请求参数
   *         // 不管使用的请方式是 get 或 post, 请参数都会被拼接到请求地址后
   *         // 此种方式的请求参数可以通过 request.getParameter() 获取
   *         params:{},
   *
   *         // 以 json 格式发送的请求参数
   *         // 请求参数会被保存到请求报文的请求体传输到服务器
   *         // 此种方式的请求参数可以不通过 request.getParameter() 获取
   *         data:{}
   *     }).then(response=>{
   *         console.log(response.data);
   *     });
   */
  let vue = new Vue({
    el: "#app",
```

```
        methods: {
          testAjax() {
            axios
              .post("/spring_mvc_ajax/test/ajax?id=1", {
                username: "MYXH",
                password: "520.ILY!",
              })
              .then((response) => {
                console.log(response.data);
              });
          },

          testRequestBody() {
            axios
              .post("/spring_mvc_ajax/test/requestBody/json", {
                id: "1",
                username: "MYXH",
                password: "520.ILY!",
                age: "21",
                gender: "男",
                email: "1735350920@qq.com",
              })
              .then((response) => {
                console.log(response.data);
              });
          },

          testResponseBody() {
            axios
              .post("/spring_mvc_ajax/test/responseBody/json")
              .then((response) => {
                console.log(response.data);
              });
          },
        },
      });
    </script>
  </body>
</html>
```

10.2.3 ③ 控制器方法：

```java
package com.myxh.springmvc.controller;

import jakarta.servlet.ServletContext;
import jakarta.servlet.http.HttpSession;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.util.MultiValueMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.MultipartFile;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URLEncoder;
import java.nio.charset.StandardCharsets;
import java.util.UUID;

/**
 * @author MYXH
 * @date 2023/9/6
 * @description
 * ResponseEntity：可以作为控制器方法的返回值，表示响应到浏览器的完整的响应报文
 * <p>
 * 文件上传的要求：
 * 1、form 表单的请求方式必须为 post
 * 2、form 表单必须设置属性 enctype="multipart/form-data"
 */
@Controller
public class FileUploadAndDownloadController
{
    @RequestMapping("/test/download")
    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/大户爱.png");
        String realPath = servletContext.getRealPath(File.separator + "static" + File.separator +
```

```
        // 创建输入流
        InputStream inputStream = new FileInputStream(realPath);

        // 创建字节数组，is.available() 获取输入流所对应文件的字节数
        byte[] bytes = new byte[inputStream.available()];

        // 将流读到字节数组中
        inputStream.read(bytes);

        // 创建 HttpHeaders 对象设置响应头信息
        MultiValueMap<String, String> headers = new HttpHeaders();

        // 对文件名进行 URL 编码
        String filename = "大户爱.png";
        String encodedFileName = URLEncoder.encode(filename, StandardCharsets.UTF_8);

        // 设置要下载方式以及下载文件的名字
        headers.add("Content-Disposition", "attachment; filename*=UTF-8''" + encodedFileName);

        // 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
        inputStream.close();

        return responseEntity;
    }

    @RequestMapping("/test/upload")
    public String testUpload(@RequestParam("picture") MultipartFile picture, HttpSession session)
    {

        // 获取上传的文件的文件名
        String filename = picture.getOriginalFilename();

        // 获取上传的文件的后缀名
        String suffixName = filename.substring(filename.lastIndexOf("."));

        // 获取 UUID
        String uuid = UUID.randomUUID().toString();
```

```
        // 拼接一个新的文件名
        filename = uuid + suffixName;

        // 获取 ServLetContext 对象
        ServletContext servletContext = session.getServletContext();

        // 获取当前工程下 img 目录的真实路径
        String imgPath = servletContext.getRealPath(File.separator + "static" + File.separator + "

        // 创建 imgPath 所对应的 FiLe 对象
        File file = new File(imgPath);

        // 判断 fiLe 所对应目录是否存在
        if (!file.exists())
        {
            file.mkdir();
        }

        String finalPath = imgPath + File.separator + filename;

        // 上传文件
        picture.transferTo(new File(finalPath));

        return "success";
    }
}
```

11、拦截器

11.1、拦截器的配置

SpringMVC 中的拦截器用于拦截控制器方法的执行。

SpringMVC 中的拦截器需要实现 HandlerInterceptor。

SpringMVC 的拦截器必须在 SpringMVC 的配置文件中进行配置：

```
<!-- 开启 mvc 的注解驱动 -->
<mvc:annotation-driven/>

<!-- <bean id="firstInterceptor" class="com.myxh.springmvc.interceptor.FirstInterceptor"/> -->

<mvc:interceptors>
    <!-- bean 和 ref 标签所配置的拦截器默认对 DispatcherServlet 处理的所有的请求进行拦截 -->
    <!-- <bean class="com.myxh.springmvc.interceptor.FirstInterceptor"/> -->
    <ref bean="firstInterceptor"/>
    <ref bean="secondInterceptor"/>
    <!--
    <mvc:interceptor>
        &lt;!&ndash; 配置需要拦截的请求的请求路径，/** 表示所有请求 &ndash;&gt;
        <mvc:mapping path="/**"/>

        &lt;!&ndash; 配置需要排除拦截的请求的请求路径 &ndash;&gt;
        <mvc:exclude-mapping path="/test/exclude"/>

        &lt;!&ndash; 配置拦截器 &ndash;&gt;
        <ref bean="firstInterceptor"/>
    </mvc:interceptor>
        -->
</mvc:interceptors>
```

11.2、拦截器的三个抽象方法

SpringMVC 中的拦截器有三个抽象方法：

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()。

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

11.3、多个拦截器的执行顺序

① 若每个拦截器的 preHandle()都返回 true。

此时多个拦截器的执行顺序和拦截器在 SpringMVC 的配置文件的配置顺序有关：

preHandle()会按照配置的顺序执行，而 postHandle()和 afterCompletion()会按照配置的反序执

行。

② 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 afterCompletion()会执行。

```java
package com.myxh.springmvc.interceptor;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

/**
 * @author MYXH
 * @date 2023/9/6
 * @description
 * 拦截器的三个方法：
 * preHandle()：在控制器方法执行之前执行，其返回值表示对控制器方法的拦截（false）或放行（true）
 * postHandle()：在控制器方法执行之后执行
 * afterCompletion()：在控制器方法执行之后，且渲染视图完毕之后执行
 * <p>
 * 多个拦截器的执行顺序和在 SpringMVC 的配置文件中配置的顺序有关
 * preHandle() 按照配置的顺序执行，而 postHandle()和 afterCompletion() 按照配置的反序执行
 * <p>
 * 若拦截器中有某个拦截器的 preHandle() 返回了 false
 * 拦截器的 preHandle() 返回 false 和它之前的拦截器的 preHandle() 都会执行
 * 所有的拦截器的 postHandle() 都不执行
 * 拦截器的 preHandle() 返回 false 之前的拦截器的 afterCompletion() 会执行
 */
@Component
public class FirstInterceptor implements HandlerInterceptor
{
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object hand
    {
        System.out.println("FirstInterceptor -> preHandle");

        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handle
    {
        System.out.println("FirstInterceptor -> postHandle");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object h
```

```
    {
        System.out.println("FirstInterceptor -> afterCompletion");
    }
}
```

```java
package com.myxh.springmvc.interceptor;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

/**
 * @author MYXH
 * @date 2023/9/6
 * @description
 * 拦截器的三个方法：
 * preHandle()：在控制器方法执行之前执行，其返回值表示对控制器方法的拦截（false）或放行（true）
 * postHandle()：在控制器方法执行之后执行
 * afterCompletion()：在控制器方法执行之后，且渲染视图完毕之后执行
 * <p>
 * 多个拦截器的执行顺序和在 SpringMVC 的配置文件中配置的顺序有关
 * preHandle() 按照配置的顺序执行，而 postHandle()和 afterCompletion() 按照配置的反序执行
 * <p>
 * 若拦截器中有某个拦截器的 preHandle() 返回了 false
 * 拦截器的 preHandle() 返回 false 和它之前的拦截器的 preHandle() 都会执行
 * 所有的拦截器的 postHandle() 都不执行
 * 拦截器的 preHandle() 返回 false 之前的拦截器的 afterCompletion() 会执行
 */
@Component
public class SecondInterceptor implements HandlerInterceptor
{
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object hand
    {
        System.out.println("SecondInterceptor -> preHandle");

        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handle
    {
        System.out.println("SecondInterceptor -> postHandle");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object h
```

```
    {
        System.out.println("SecondInterceptor -> afterCompletion");
    }
}
```

12、异常处理器

12.1、基于配置的异常处理

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 和 SimpleMappingExceptionResolver。

SpringMVC 提供了自定义的异常处理器 SimpleMappingExceptionResolver，使用方式：

```
<!-- 开启 mvc 的注解驱动 -->
<mvc:annotation-driven/>

<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <!-- key 设置要处理的异常，value 设置出现该异常时要跳转的页面所对应的逻辑视图 -->
            <prop key="java.lang.ArithmeticException">error</prop>
        </props>
    </property>

    <!-- 设置共享在请求域中的异常信息的属性名 -->
    <property name="exceptionAttribute" value="ex"/>
</bean>
```

12.2、基于注解的异常处理

```java
package com.myxh.springmvc.controller;

import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

/**
 * @author MYXH
 * @date 2023/9/6
 */
// 将当前类标识为异常处理的组件
@ControllerAdvice
public class ExceptionController
{
    // 设置要处理的异常信息
    @ExceptionHandler(ArithmeticException.class)
    public String handleException(Throwable ex, Model model)
    {
        // ex 表示控制器方法所出现的异常
        model.addAttribute("ex", ex);

        return "error";
    }
}
```

13、注解配置 SpringMVC

使用配置类和注解代替 web.xml 和 SpringMVC 配置文件的功能。

13.1、创建初始化类，代替 web.xml

在 Servlet3.0 环境中，容器会在类路径中查找实现 javax.servlet.ServletContainerInitializer 接口的类，如果找到的话就用它来配置 Servlet 容器。 Spring 提供了这个接口的实现，名为 SpringServletContainerInitializer，这个类反过来又会查找实现 WebApplicationInitializer 的类并将配置的任务交给它们来完成。Spring3.2 引入了一个便利的 WebApplicationInitializer 基础实现，名为 AbstractAnnotationConfigDispatcherServletInitializer，当我们的类扩展了 AbstractAnnotationConfigDispatcherServletInitializer 并将其部署到 Servlet3.0 容器的时候，容器会自动发现它，并用它来配置 Servlet 上下文。

在 Servlet5.0 之后 javax 包名更改为了 jakarta。

```java
package com.myxh.springmvc.config;

import jakarta.servlet.Filter;
import org.springframework.web.filter.CharacterEncodingFilter;
import org.springframework.web.filter.HiddenHttpMethodFilter;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitialize

/**
 * @author MYXH
 * @date 2023/9/6
 * @description 代替 web.xml
 */
public class WebInit extends AbstractAnnotationConfigDispatcherServletInitializer
{
    /**
     * 设置一个配置类代替 Spring 的配置文件
     *
     * @return 根应用程序上下文的配置，如果不需要创建和注册根上下文，则为 null
     */
    @Override
    protected Class<?>[] getRootConfigClasses()
    {
        return new Class[]{SpringConfig.class};
    }

    /**
     * 设置一个配置类代替 SpringMVC 的配置文件
     * @return Servlet 应用程序上下文的配置，如果所有配置都是通过根配置类指定的，则为 null
     */
    @Override
    protected Class<?>[] getServletConfigClasses()
    {
        return new Class[]{WebConfig.class};
    }

    /**
     * 设置 SpringMVC 的前端控制器 DispatcherServlet 的 url-pattern
     */
    @Override
    protected String[] getServletMappings()
    {
        return new String[]{"/"};
    }
```

```
    /**
     * 设置当前的过滤器
     * @return 筛选器数组或 null
     */
    @Override
    protected Filter[] getServletFilters()
    {
        // 创建编码过滤器
        CharacterEncodingFilter characterEncodingFilter = new CharacterEncodingFilter();
        characterEncodingFilter.setEncoding("UTF-8");
        characterEncodingFilter.setForceEncoding(true);

        // 创建处理请求方式的过滤器
        HiddenHttpMethodFilter hiddenHttpMethodFilter = new HiddenHttpMethodFilter();

        return new Filter[]{characterEncodingFilter, hiddenHttpMethodFilter};
    }
}
```

13.2、创建 SpringConfig 配置类，代替 spring 的配置文件

```
package com.myxh.springmvc.config;

import org.springframework.context.annotation.Configuration;

/**
 * @author MYXH
 * @date 2023/9/6
 * @description 代替 Spring 的配置文件
 */
// 将类标识为配置类
@Configuration
public class SpringConfig
{

}
```

13.3、创建 WebConfig 配置类，代替 SpringMVC 的配置文件

```java
package com.myxh.springmvc.config;

import com.myxh.springmvc.interceptor.FirstInterceptor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.context.ContextLoader;
import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.multipart.support.StandardServletMultipartResolver;
import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.*;
import org.springframework.web.servlet.handler.SimpleMappingExceptionResolver;
import org.thymeleaf.spring6.SpringTemplateEngine;
import org.thymeleaf.spring6.templateresolver.SpringResourceTemplateResolver;
import org.thymeleaf.spring6.view.ThymeleafViewResolver;
import org.thymeleaf.templatemode.TemplateMode;
import org.thymeleaf.templateresolver.ITemplateResolver;

import java.util.List;
import java.util.Properties;

/**
 * @author MYXH
 * @date 2023/9/6
 * @description 代替 SpringMVC 的配置文件
 * 扫描组件、视图解析器、默认的 servlet、mvc 的注解驱动
 * 视图控制器、文件上传解析器、拦截器、异常解析器
 */
// 将类标识为配置类
@Configuration
// 扫描组件
@ComponentScan("com.myxh.springmvc.controller")
// 开启 mvc 的注解驱动
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer
{
    /**
     * 默认的 servlet 处理静态资源
     */
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer)
    {
        configurer.enable();
```

```
    }

    /**
     * 配置视图控制器
     */
    @Override
    public void addViewControllers(ViewControllerRegistry registry)
    {
        registry.addViewController("/").setViewName("index");
    }

    /**
     * {@code @Bean} 注解可以将标识的方法的返回值作为 bean 进行管理，bean 的 id 为方法的方法名
     */
    @Bean
    public StandardServletMultipartResolver multipartResolver()
    {
        return new StandardServletMultipartResolver();
    }

    /**
     * 配置拦截器
     */
    @Override
    public void addInterceptors(InterceptorRegistry registry)
    {
        FirstInterceptor firstInterceptor = new FirstInterceptor();
        registry.addInterceptor(firstInterceptor).addPathPatterns("/**");
    }

    /**
     * 配置异常解析器
     *
     * @param resolvers 解析器-最初是一个空列表
     */
    @Override
    public void configureHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers)
    {
        SimpleMappingExceptionResolver exceptionResolver = new SimpleMappingExceptionResolver();
        Properties prop = new Properties();
        prop.setProperty("java.lang.ArithmeticException", "error");
        exceptionResolver.setExceptionMappings(prop);
        exceptionResolver.setExceptionAttribute("ex");
        resolvers.add(exceptionResolver);
```

```
    }

    /**
     * 配置生成模板解析器
     *
     * @return templateResolver 模板解析程序
     */
    @Bean
    public ITemplateResolver templateResolver()
    {
        WebApplicationContext webApplicationContext = ContextLoader.getCurrentWebApplicationContex
        SpringResourceTemplateResolver templateResolver = new SpringResourceTemplateResolver();
        templateResolver.setPrefix("/WEB-INF/templates/");
        templateResolver.setSuffix(".html");
        templateResolver.setCharacterEncoding("UTF-8");
        templateResolver.setTemplateMode(TemplateMode.HTML);

        return templateResolver;
    }

    /**
     * 生成模板引擎并为模板引擎注入模板解析器
     *
     * @param templateResolver 模板解析程序
     * @return templateEngine 模板引擎
     */
    @Bean
    public SpringTemplateEngine templateEngine(ITemplateResolver templateResolver)
    {
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(templateResolver);

        return templateEngine;
    }

    /**
     * 生成视图解析器并未解析器注入模板引擎
     *
     * @param templateEngine 模板引擎
     * @return viewResolver 视图解析程序
     */
    @Bean
    public ViewResolver viewResolver(SpringTemplateEngine templateEngine)
    {
```

```
        ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
        viewResolver.setCharacterEncoding("UTF-8");
        viewResolver.setTemplateEngine(templateEngine);

        return viewResolver;
    }
}
```

13.4、测试功能

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>首页</title>
  </head>

  <body>
    <h1>index.html</h1>
  </body>
</html>
```

```
package com.myxh.springmvc.controller;

import org.springframework.stereotype.Controller;

/**
 * @author MYXH
 * @date 2023/9/6
 */
@Controller
public class TestController
{

}
```

```java
package com.myxh.springmvc.interceptor;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

/**
 * @author MYXH
 * @date 2023/9/6
 */
public class FirstInterceptor implements HandlerInterceptor
{
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object hand
    {
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handle
    {

    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object h
    {

    }
}
```

14、SpringMVC 执行流程

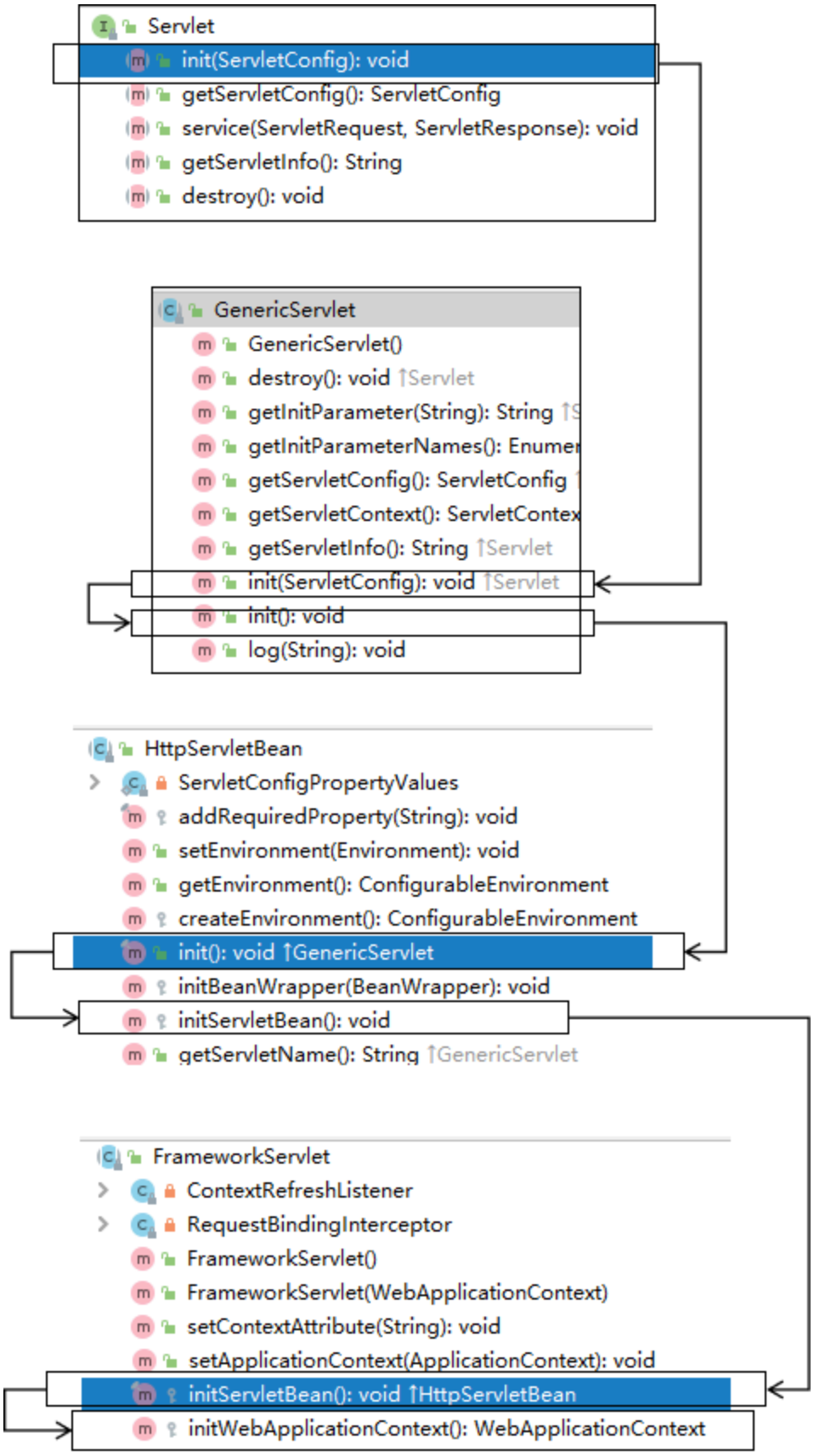
14.1、SpringMVC 常用组件

- DispatcherServlet：**前端控制器**，不需要工程师开发，由框架提供。
  作用：统一处理请求和响应，整个流程控制的中心，由它调用其它组件处理用户的请求。
- HandlerMapping：**处理器映射器**，不需要工程师开发，由框架提供。
  作用：根据请求的 url、method 等信息查找 Handler，即控制器方法。

- Handler：**处理器**，需要工程师开发。
  作用：在 DispatcherServlet 的控制下 Handler 对具体的用户请求进行处理。
- HandlerAdapter：**处理器适配器**，不需要工程师开发，由框架提供。
  作用：通过 HandlerAdapter 对处理器（控制器方法）进行执行。
- ViewResolver：**视图解析器**，不需要工程师开发，由框架提供。
  作用：进行视图解析，得到相应的视图，例如：ThymeleafView、InternalResourceView、RedirectView。
- View：**视图**。
  作用：将模型数据通过页面展示给用户。

14.2、DispatcherServlet 初始化过程

DispatcherServlet 本质上是一个 Servlet，所以天然的遵循 Servlet 的生命周期。所以宏观上是 Servlet 生命周期来进行调度。


Servlet
init(ServletConfig): void
getServletConfig(): ServletConfig
service(ServletRequest, ServletResponse): void
getServletInfo(): String
destroy(): void
GenericServlet
GenericServlet()
destroy(): void ↑Servlet
getInitParameter(String): String ↑S
getInitParameterNames(): Enumer
getServletConfig(): ServletConfig
getServletContext(): ServletContex
getServletInfo(): String ↑Servlet
init(ServletConfig): void ↑Servlet
init(): void
log(String): void
HttpServletBean
ServletConfigPropertyValues
addRequiredProperty(String): void
setEnvironment(Environment): void
getEnvironment(): ConfigurableEnvironment
createEnvironment(): ConfigurableEnvironment
init(): void ↑GenericServlet
initBeanWrapper(BeanWrapper): void
initServletBean(): void
getServletName(): String ↑GenericServlet
FrameworkServlet
ContextRefreshListener
RequestBindingInterceptor
FrameworkServlet()
FrameworkServlet(WebApplicationContext)
setContextAttribute(String): void
setApplicationContext(ApplicationContext): void
initServletBean(): void ↑HttpServletBean
initWebApplicationContext(): WebApplicationContext

14.2.1 ① 初始化 WebApplicationContext

所在类：org.springframework.web.servlet.FrameworkServlet。

```java
/**
 * Initialize and publish the WebApplicationContext for this servlet.
 * <p>Delegates to {@link #createWebApplicationContext} for actual creation
 * of the context. Can be overridden in subclasses.
 * @return the WebApplicationContext instance
 * @see #FrameworkServlet(WebApplicationContext)
 * @see #setContextClass
 * @see #setContextConfigLocation
 */
protected WebApplicationContext initWebApplicationContext() {
    WebApplicationContext rootContext =
            WebApplicationContextUtils.getWebApplicationContext(getServletContext());
    WebApplicationContext wac = null;

    if (this.webApplicationContext != null) {
        // A context instance was injected at construction time -> use it
        wac = this.webApplicationContext;
        if (wac instanceof ConfigurableWebApplicationContext cwac && !cwac.isActive()) {
            // The context has not yet been refreshed -> provide services such as
            // setting the parent context, setting the application context id, etc
            if (cwac.getParent() == null) {
                // The context instance was injected without an explicit parent -> set
                // the root application context (if any; may be null) as the parent
                cwac.setParent(rootContext);
            }
            configureAndRefreshWebApplicationContext(cwac);
        }
    }

    if (wac == null) {
        // No context instance was injected at construction time -> see if one
        // has been registered in the servlet context. If one exists, it is assumed
        // that the parent context (if any) has already been set and that the
        // user has performed any initialization such as setting the context id
        wac = findWebApplicationContext();
    }

    if (wac == null) {
        // No context instance is defined for this servlet -> create a local one
        // 创建 WebApplicationContext
        wac = createWebApplicationContext(rootContext);
    }

    if (!this.refreshEventReceived) {
```

```
            // Either the context is not a ConfigurableApplicationContext with refresh
            // support or the context injected at construction time had already been
            // refreshed -> trigger initial onRefresh manually here.
            synchronized (this.onRefreshMonitor) {
                onRefresh(wac);
            }
        }

        if (this.publishContext) {
            // Publish the context as a servlet context attribute.
            // 将 IOC 容器在应用域共享
            String attrName = getServletContextAttributeName();
            getServletContext().setAttribute(attrName, wac);
        }

        return wac;
    }
```

14.2.2 ② 创建 WebApplicationContext

所在类：org.springframework.web.servlet.FrameworkServlet。

```java
/**
 * Instantiate the WebApplicationContext for this servlet, either a default
 * {@link org.springframework.web.context.support.XmlWebApplicationContext}
 * or a {@link #setContextClass custom context class}, if set.
 * <p>This implementation expects custom contexts to implement the
 * {@link org.springframework.web.context.ConfigurableWebApplicationContext}
 * interface. Can be overridden in subclasses.
 * <p>Do not forget to register this servlet instance as application listener on the
 * created context (for triggering its {@link #onRefresh callback}), and to call
 * {@link org.springframework.context.ConfigurableApplicationContext#refresh()}
 * before returning the context instance.
 * @param parent the parent ApplicationContext to use, or {@code null} if none
 * @return the WebApplicationContext for this servlet
 * @see org.springframework.web.context.support.XmlWebApplicationContext
 */
protected WebApplicationContext createWebApplicationContext(@Nullable ApplicationContext parent) {
    Class<?> contextClass = getContextClass();
    if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {
        throw new ApplicationContextException(
                "Fatal initialization error in servlet with name '" + getServletName() +
                "': custom WebApplicationContext class [" + contextClass.getName() +
                "] is not of type ConfigurableWebApplicationContext");
    }

     // 通过反射创建 IOC 容器对象
    ConfigurableWebApplicationContext wac =
            (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);

    wac.setEnvironment(getEnvironment());

     // 设置父容器
    wac.setParent(parent);
    String configLocation = getContextConfigLocation();
    if (configLocation != null) {
        wac.setConfigLocation(configLocation);
    }
    configureAndRefreshWebApplicationContext(wac);

    return wac;
}
```

14.2.3 ③ DispatcherServlet 初始化策略

FrameworkServlet 创建 WebApplicationContext 后，刷新容器，调用 onRefresh(wac)，此方法

在 DispatcherServlet 中进行了重写，调用了 initStrategies(context)方法，初始化策略，即初始化 DispatcherServlet 的各个组件。

所在类：org.springframework.web.servlet.DispatcherServlet。

```
/**
 * Initialize the strategy objects that this servlet uses.
 * <p>May be overridden in subclasses in order to initialize further strategy objects.
 */
protected void initStrategies(ApplicationContext context) {
    initMultipartResolver(context);
    initLocaleResolver(context);
    initThemeResolver(context);
    initHandlerMappings(context);
    initHandlerAdapters(context);
    initHandlerExceptionResolvers(context);
    initRequestToViewNameTranslator(context);
    initViewResolvers(context);
    initFlashMapManager(context);
}
```

14.3、DispatcherServlet 调用组件处理请求

14.3.1 ① processRequest()

FrameworkServlet 重写 HttpServlet 中的 service()和 doXxx()，这些方法中调用了 processRequest(request, response)。

所在类：org.springframework.web.servlet.FrameworkServlet。

```java
/**
 * Process this request, publishing an event regardless of the outcome.
 * <p>The actual event handling is performed by the abstract
 * {@link #doService} template method.
 */
protected final void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    long startTime = System.currentTimeMillis();
    Throwable failureCause = null;

    LocaleContext previousLocaleContext = LocaleContextHolder.getLocaleContext();
    LocaleContext localeContext = buildLocaleContext(request);

    RequestAttributes previousAttributes = RequestContextHolder.getRequestAttributes();
    ServletRequestAttributes requestAttributes = buildRequestAttributes(request, response, previou

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
    asyncManager.registerCallableInterceptor(FrameworkServlet.class.getName(), new RequestBindingI

    initContextHolders(request, localeContext, requestAttributes);

    try {
         // 执行服务，doService() 是一个抽象方法，在 DispatcherServlet 中进行了重写
        doService(request, response);
    }
    catch (ServletException | IOException ex) {
        failureCause = ex;
        throw ex;
    }
    catch (Throwable ex) {
        failureCause = ex;
        throw new ServletException("Request processing failed: " + ex, ex);
    }

    finally {
        resetContextHolders(request, previousLocaleContext, previousAttributes);
        if (requestAttributes != null) {
            requestAttributes.requestCompleted();
        }
        logResult(request, response, failureCause, asyncManager);
        publishRequestHandledEvent(request, response, startTime, failureCause);
    }
}
```

14.3.2 ② doService()

所在类：org.springframework.web.servlet.DispatcherServlet。

```java
/**
 * Exposes the DispatcherServlet-specific request attributes and delegates to {@link #doDispatch}
 * for the actual dispatching.
 */
@Override
protected void doService(HttpServletRequest request, HttpServletResponse response) throws Exceptic
    logRequest(request);

    // Keep a snapshot of the request attributes in case of an include,
    // to be able to restore the original attributes after the include.
    Map<String, Object> attributesSnapshot = null;
    if (WebUtils.isIncludeRequest(request)) {
        attributesSnapshot = new HashMap<>();
        Enumeration<?> attrNames = request.getAttributeNames();
        while (attrNames.hasMoreElements()) {
            String attrName = (String) attrNames.nextElement();
            if (this.cleanupAfterInclude || attrName.startsWith(DEFAULT_STRATEGIES_PREFIX)) {
                attributesSnapshot.put(attrName, request.getAttribute(attrName));
            }
        }
    }

    // Make framework objects available to handlers and view objects.
    request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE, getWebApplicationContext());
    request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);
    request.setAttribute(THEME_RESOLVER_ATTRIBUTE, this.themeResolver);
    request.setAttribute(THEME_SOURCE_ATTRIBUTE, getThemeSource());

    if (this.flashMapManager != null) {
        FlashMap inputFlashMap = this.flashMapManager.retrieveAndUpdate(request, response);
        if (inputFlashMap != null) {
            request.setAttribute(INPUT_FLASH_MAP_ATTRIBUTE, Collections.unmodifiableMap(inputFlash
        }
        request.setAttribute(OUTPUT_FLASH_MAP_ATTRIBUTE, new FlashMap());
        request.setAttribute(FLASH_MAP_MANAGER_ATTRIBUTE, this.flashMapManager);
    }

    RequestPath previousRequestPath = null;
    if (this.parseRequestPath) {
        previousRequestPath = (RequestPath) request.getAttribute(ServletRequestPathUtils.PATH_ATTR
        ServletRequestPathUtils.parseAndCache(request);
    }

    try {
```

```
            // 处理请求和响应
            doDispatch(request, response);
        }
        finally {
            if (!WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
                // Restore the original attribute snapshot, in case of an include.
                if (attributesSnapshot != null) {
                    restoreAttributesAfterInclude(request, attributesSnapshot);
                }
            }
            if (this.parseRequestPath) {
                ServletRequestPathUtils.setParsedRequestPath(previousRequestPath, request);
            }
        }
    }
```

14.3.3 ③ doDispatch()

所在类：org.springframework.web.servlet.DispatcherServlet。

```java
/**
 * Process the actual dispatching to the handler.
 * <p>The handler will be obtained by applying the servlet's HandlerMappings in order.
 * The HandlerAdapter will be obtained by querying the servlet's installed HandlerAdapters
 * to find the first that supports the handler class.
 * <p>All HTTP methods are handled by this method. It's up to HandlerAdapters or handlers
 * themselves to decide which methods are acceptable.
 * @param request current HTTP request
 * @param response current HTTP response
 * @throws Exception in case of any kind of processing failure
 */
@SuppressWarnings("deprecation")
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Excepti
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    try {
        ModelAndView mv = null;
        Exception dispatchException = null;

        try {
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

            // Determine handler for the current request.
            /*
                mappedHandler：调用链
                包含handler、interceptorList、interceptorIndex
                handler：浏览器发送的请求所匹配的控制器方法
                interceptorList：处理控制器方法的所有拦截器集合
                interceptorIndex：拦截器索引，控制拦截器 afterCompletion() 的执行
             */
            mappedHandler = getHandler(processedRequest);
            if (mappedHandler == null) {
                noHandlerFound(processedRequest, response);
                return;
            }

            // Determine handler adapter for the current request.
            // 通过控制器方法创建相应的处理器适配器，调用所对应的控制器方法
            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
```

```
            // Process last-modified header, if supported by the handler.
            String method = request.getMethod();
            boolean isGet = HttpMethod.GET.matches(method);
            if (isGet || HttpMethod.HEAD.matches(method)) {
                long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
                if (new ServletWebRequest(request, response).checkNotModified(lastModified) && isG
                    return;
                }
            }

            // 调用拦截器的 preHandle()
            if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                return;
            }

            // Actually invoke the handler.
            // 由处理器适配器调用具体的控制器方法，最终获得 ModelAndView 对象
            mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

            if (asyncManager.isConcurrentHandlingStarted()) {
                return;
            }

            applyDefaultViewName(processedRequest, mv);

            // 调用拦截器的 postHandle()
            mappedHandler.applyPostHandle(processedRequest, response, mv);
        }
        catch (Exception ex) {
            dispatchException = ex;
        }
        catch (Throwable err) {
            // As of 4.3, we're processing Errors thrown from handler methods as well,
            // making them available for @ExceptionHandler methods and other scenarios.
            dispatchException = new ServletException("Handler dispatch failed: " + err, err);
        }

        // 后续处理：处理模型数据和渲染视图
        processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
    }
    catch (Exception ex) {
        triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
    }
```

```
    catch (Throwable err) {
        triggerAfterCompletion(processedRequest, response, mappedHandler,
                new ServletException("Handler processing failed: " + err, err));
    }
    finally {
        if (asyncManager.isConcurrentHandlingStarted()) {
            // Instead of postHandle and afterCompletion
            if (mappedHandler != null) {
                mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
            }
        }
        else {
            // Clean up any resources used by a multipart request.
            if (multipartRequestParsed) {
                cleanupMultipart(processedRequest);
            }
        }
    }
}
```

14.3.4 ④ processDispatchResult()

```java
/**
 * Handle the result of handler selection and handler invocation, which is
 * either a ModelAndView or an Exception to be resolved to a ModelAndView.
 */
private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
        @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv,
        @Nullable Exception exception) throws Exception {

    boolean errorView = false;

    if (exception != null) {
        if (exception instanceof ModelAndViewDefiningException mavDefiningException) {
            logger.debug("ModelAndViewDefiningException encountered", exception);
            mv = mavDefiningException.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()) {
        // 处理模型数据和渲染视图
        render(mv, request, response);
        if (errorView) {
            WebUtils.clearErrorRequestAttributes(request);
        }
    }
    else {
        if (logger.isTraceEnabled()) {
            logger.trace("No view rendering, null ModelAndView returned.");
        }
    }

    if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
        // Concurrent handling started during a forward
        return;
    }

    if (mappedHandler != null) {
        // Exception (if any) is already handled..
        // 调用拦截器的afterCompletion()
```

```
            mappedHandler.triggerAfterCompletion(request, response, null);
        }
    }
```

14.4、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：

① 不存在。

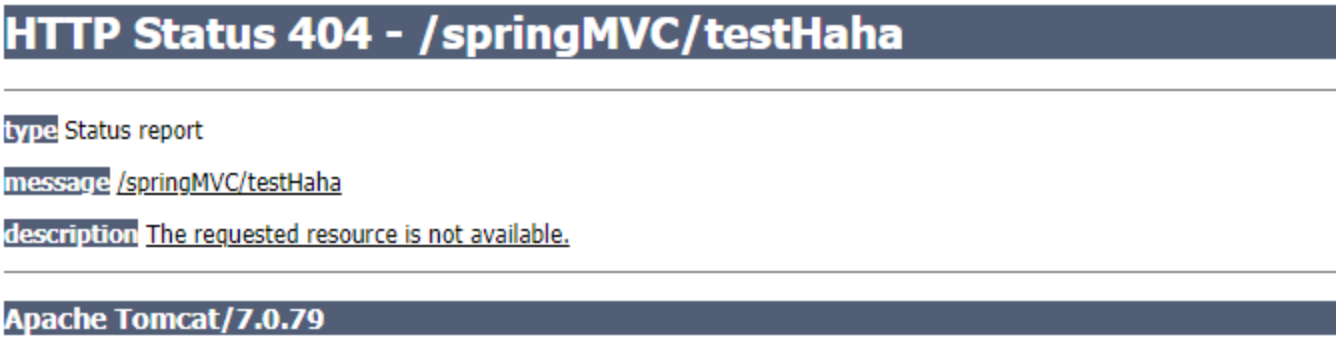
i. 再判断是否配置了 mvc:default-servlet-handler。

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。

```
DEBUG org.springframework.web.servlet.DispatcherServlet - GET "/springMVC/testHaha", parameters={}
WARN org.springframework.web.servlet.PageNotFound - No mapping for GET /springMVC/testHaha
DEBUG org.springframework.web.servlet.DispatcherServlet - Completed 404 NOT_FOUND
```

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。

```
DispatcherServlet - GET "/springMVC/testHaha", parameters={}
handler.SimpleUrlHandlerMapping - Mapped to org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler
DispatcherServlet - Completed 404 NOT_FOUND
```



② 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 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【正向】。

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：

① 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。

② 数据转换：对请求消息进行数据转换。如 String 转换成 Integer、Double 等。

③ 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。

④ 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。