第 3 章

CHAPTER 3

Spring MVC 详解

3.1 Spring MVC 概述

3.1.1 什么是 MVC

MVC(Model-View-Controller)是一种软件架构模式,旨在将应用程序分成 3 个核心部分:模型(Model)、视图(View)和控制器(Controller)。这种模式的设计目标是实现各部分之间的松耦合,以便更轻松地进行开发、测试和维护。

1. M:Model(模型层)

模型层是应用程序的核心,它包含了应用程序的数据和业务逻辑。在 Java Web 应用中,模型通常是由 JavaBean 表示的。这些 JavaBean 可以是实体类,用于表示业务对象(例如用户、订单等),也可以是用于处理业务逻辑和数据访问的服务类或 DAO(Data Access Object,数据访问对象)。

模型层的主要职责包括以下几种。

(1) 封装应用程序的数据和状态。

(2) 提供访问和操作数据的方法。

(3) 执行业务逻辑。

2. V:View(视图层)

视图层是用户界面的呈现层,负责将模型的数据呈现给用户。在 Web 应用中,视图通常是由 HTML、JSP 或其他模板引擎生成的页面。视图的主要职责是将数据呈现为用户友好的界面,以便用户可以与之交互。

视图层的主要特点包括以下几点。

(1) 显示模型层的数据。

(2) 接收用户的输入。

(3) 提供用户友好的界面。

3. C：Controller(控制层)

控制器是应用程序的主要逻辑处理部分，负责处理用户的请求并根据需要更新模型和视图。在 Java Web 应用中，控制器通常是由 Servlet 或 Spring MVC 控制器表示的。控制器的主要职责包括以下几种。

(1) 接收来自用户的请求。

(2) 调用模型层处理业务逻辑。

(3) 选择合适的视图并将模型的数据传递给视图。

(4) 处理与用户交互相关的逻辑。

MVC 架构主要具有以下优势。

(1) 分离关注点(Separation of Concerns)：MVC 将应用程序分成 3 个独立的部分，每部分专注于不同的任务，从而使代码更易于理解、测试和维护。

(2) 模块化(Modularity)：每部分都是相互独立的模块，可以根据需要进行替换、修改或扩展。

(3) 可复用性(Reusability)：通过模型和控制器的复用，可以减少重复代码的编写，提高代码的复用性。

(4) 灵活性(Flexibility)：MVC 架构使应用程序的各部分可以独立开发、测试和部署，从而提高了系统的灵活性和可扩展性。

(5) 易于维护(Ease of Maintenance)：由于各部分之间的松耦合，使系统更容易进行修改、调试和优化。

3.1.2 MVC 大概流程

当用户将请求发送至 DispatcherServlet 时，DispatcherServlet 首先会根据请求信息调用 HandlerMapping 来确定处理该请求的 Controller。Controller 接受请求后，可能会进行一系列的业务逻辑处理，包括调用服务层、数据访问层等，最终将处理结果封装为 ModelAndView 对象。接着，DispatcherServlet 根据视图解析器将视图名称解析为实际的 View 对象，View 对象负责渲染模型数据以生成最终的 HTML 响应。最后，DispatcherServlet 将响应返给用户，完成一次请求处理流程。在这个流程中，各个组件相互配合，协同工作，使请求能够得到有效处理，并生成最终的响应，如图 3-1 所示。

3.1.3 MVC 的功能概述

Spring MVC 围绕 DispatcherServlet 设计。DispatcherServlet 的作用是将请求分发到不同的处理器。从 Spring 2.5 开始，使用 Java 5 或者以上版本的用户可以采用基于注解的 Controller 声明方式。Spring 的 Web 模块提供了大量独特的功能。

(1) 清晰的角色划分：控制器(Controller)、验证器(Validator)、命令对象(Command Object)、表单对象(Form Object)、模型对象(Model Object)、Servlet 分发器(Dispatcher

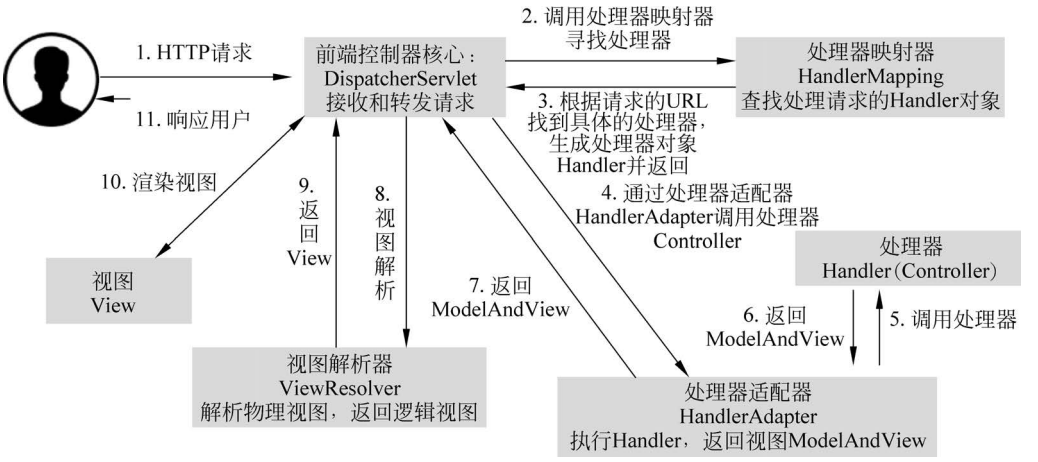

图 3-1 请求处理流程

Servlet)、处理器映射器(Handler Mapping)、视图解析器(View Resolver)等。每个角色都可以由一个专门的对象来实现。

(2) 强大而直接的配置方式：将框架类和应用程序类都作为 JavaBean 配置，支持跨多个 context 的引用，例如，在 Web 控制器中对业务对象和验证器进行引用。

(3) 可适配、非侵入：可以根据不同的应用场景，选择合适的控制器子类（simple 型、command 型、form 型、wizard 型、multi-action 型或者自定义），而不是从单一控制器（例如 Action/ActionForm)继承。

(4) 可重用的业务代码：可以使用现有的业务对象作为命令或表单对象，而不需要去扩展某个特定框架的基类。

(5) 可定制的绑定(Binding)和验证(Validation)：例如将类型不匹配作为应用级的验证错误，这可以保存错误的值。再例如本地化的日期和数字绑定等。在其他某些框架中，只能使用字符串表单对象，需要手动解析它并转换到业务对象。

(6) 可定制的 Handler Mapping 和 View Resolution：Spring 提供了从最简单的 URL 映射到复杂的、专用的定制策略。与某些 Web MVC 框架强制开发人员使用单一特定技术相比，Spring 显得更加灵活。

(7) 灵活的 Model 转换：在 Spring Web 框架中，使用基于 Map 的键-值对来达到轻易地与各种视图技术的集成。

(8) 可定制的本地化和主题(Theme)解析：支持在 JSP 中可选择地使用 Spring 标签库、支持 JSTL、支持 Velocity(不需要额外的中间层)等。

(9) 简单而强大的 JSP 标签库(Spring Tag Library)：支持包括诸如数据绑定和主题(Theme)之类的许多功能。它提供在标记方面的最大灵活性。

3.1.4 快速上手

创建 Maven 工程，JDK 1.8 项目选择 webapp，如图 3-2 所示。

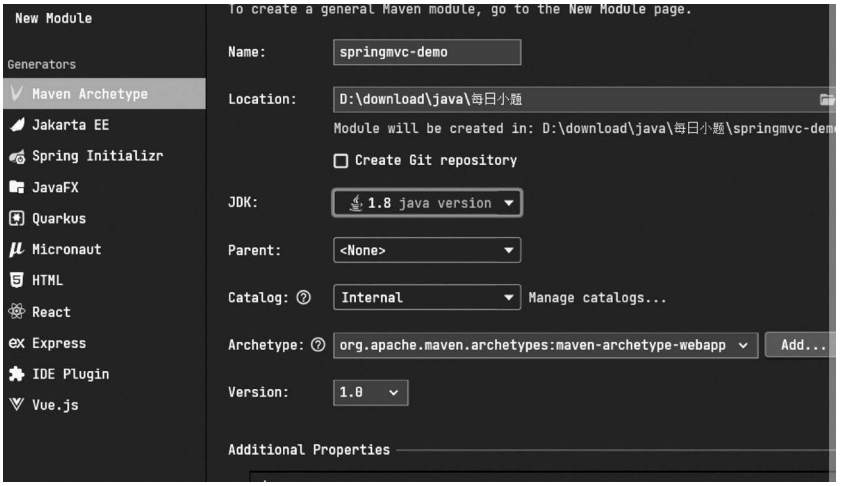

图 3-2　创建项目

引入依赖配置,代码如下:

```
//第 3 章 pom.xml
<dependencies>
    <!-- SpringMVC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.1</version>
    </dependency>
    <!-- 日志 -->
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
    </dependency>
    <!-- ServletAPI -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

在 webapp 中的 WEB-INF 文件夹下配置 web.xml,代码如下:

```
//第 3 章 web.xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
```

```
            xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee http://java.sun.com/
xml/ns/javaee/web-app_2_5.xsd"
            version = "2.5">

    <!-- 配置核心控制器 -->
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <!-- SPRING MVC 配置文件加载路径
            (1)在默认情况下,读取 WEB-INF 下面的文件
            (2)可以改为加载类路径下(resources 目录),加上 classpath:
        -->
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:springmvc.xml</param-value>
    </init-param>
    <!--
          DispatcherServlet 对象创建时间问题
          (1)在默认情况下,第 1 次访问该 Servlet 的创建对象,意味着在这段时间才去加载
springMVC.xml
          (2)可以改变为在项目启动的时候就创建该 Servlet,提高用户访问体验。
              <load-on-startup>1</load-on-startup>
                    数值越大,对象创建的优先级越低!(数值越小,越先创建)
        -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcherServlet</servlet-name>
<url-pattern>*.do</url-pattern>
<!-- 标签中使用/和/* 的区别:
/所匹配的请求可以是/login、.html、.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
-->
    </servlet-mapping>

</web-app>
```

在 src/main/resource 目录下创建 springmvc.xml 文件,写入配置,代码如下:

```
//第 3 章 springmvc.xml
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
        xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc = "http://www.springframework.org/schema/mvc"
```

```
        xmlns:context = "http://www.springframework.org/schema/context"
        xsi:schemaLocation = "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring - beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring - mvc.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring - context.xsd">

    <!-- 1.扫描 Controller 的包 -->
    <context:component - scan base - package = "com.panther.controller"/>

    <!-- 2.配置视图解析器 -->
    <bean
    class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
            <!-- 2.1 页面前缀 -->
            <property name = "prefix" value = "/WEB - INF/templates/"/>
            <!-- 2.2 页面后缀 -->
            <property name = "suffix" value = ".html"/>
        <property name = "templateMode" value = "HTML5"/>
        <property name = "characterEncoding" value = "UTF - 8" />
    </bean>
    <!-- 3.开启 MVC 注解驱动 -->
    <mvc:annotation - driven/>
</beans>
```

Controller 需要和上面配置文件配置的地址相同，代码如下：

```
//第 3 章 HelloController.java
@Controller
public class HelloController {

    //@RequestMapping 注解:处理请求和控制器方法之间的映射关系
    //http://localhost:8088/springmvc_war/
        @RequestMapping("/hello.do")
@ResponseBody
    public String index() {
        return "<p style = color: red;>hello world</p>";
    }
}
```

由于项目需要运行在 Tomcat 服务器上，因此需要下载 Tomcat 服务器，如图 3-3 所示。

在弹出界面后先选择左上角的加号添加 Tomcat Server，然后选择 Local，如图 3-4 所示。

然后单击 Deployment 部署构建好的 Artifacts，如图 3-5 所示。

如果没有 Artifacts，则需要在项目结构生成，如图 3-6 所示。

创建对应的 Artifacts 步骤，选择创建的模块进行生成即可，如图 3-7 所示。

在浏览器地址栏输入 localhost:8088/springmvc/hello.do，这样就可以看到输出的信息，结果如图 3-8 所示。

图 3-3　配置 Tomcat

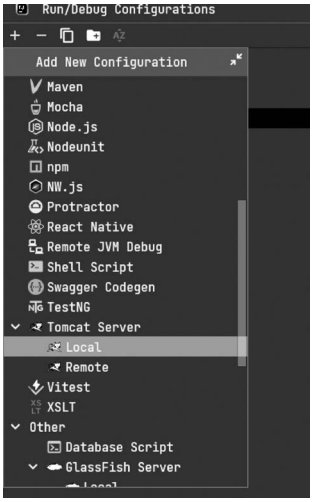

图 3-4　选择 Local

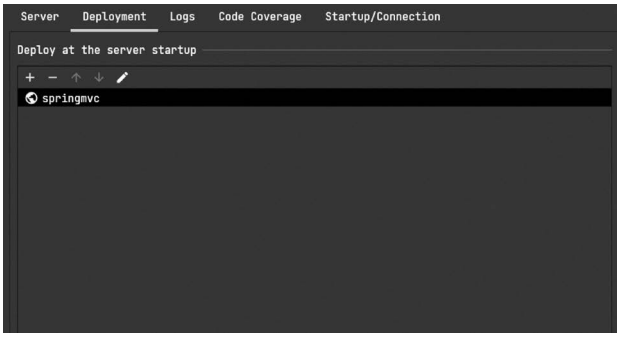

图 3-5　选择对应的 Artifacts

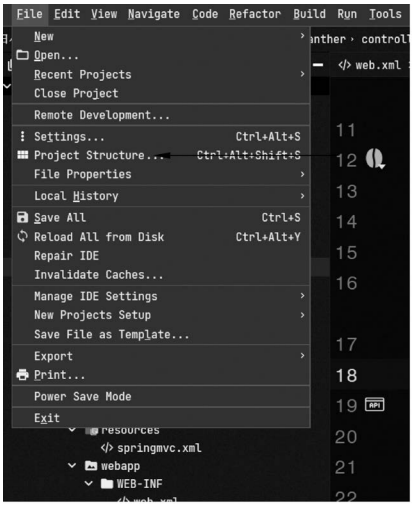

图 3-6　生成项目的 Artifacts（1）

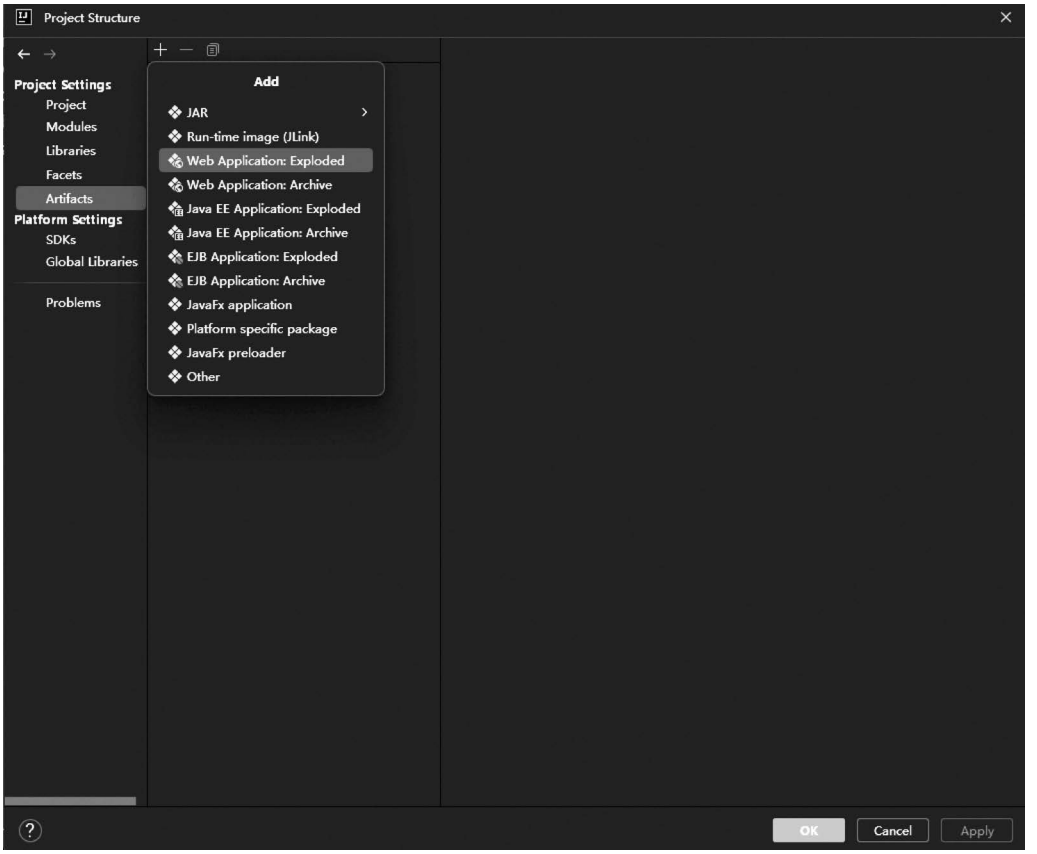

图 3-7 生成项目的 Artifacts(2)

图 3-8 页面请求后的结果

3.2 Spring MVC 核心组件

本节介绍 Spring MVC 的三大组件,分别是处理器映射器(HandlerMapper)、处理器适配器(HandlerAdapter)、视图解析器(ViewResolver)。这些组件在 Spring MVC 框架中扮演着至关重要的角色,它们负责协调请求的处理、调度适当的处理器及解析视图,为开发者

提供了强大而灵活的工具，使开发 Web 应用程序变得更加简单和高效。

1. HandlerMapper 的作用

处理器映射可以将 Web 请求映射到正确的处理器 Controller 上。当接收到请求时，DispatcherServlet 将请求交给 HandlerMapping 处理器映射，让它检查请求并找到一个合适的 HandlerExecutionChain，这个 HandlerExecutionChain 包含一个能处理该请求的处理器 Controller，然后 DispatcherServlet 执行在 HandlerExecutionChain 中的处理器 Controller。

Spring 内置了许多处理器映射策略，目前主要由 3 个实现：SimpleUrlHandlerMapping、BeanNameUrlHandlerMapping 和 RequestMappingHandlerMapping，所有实现类如图 3-9 所示。

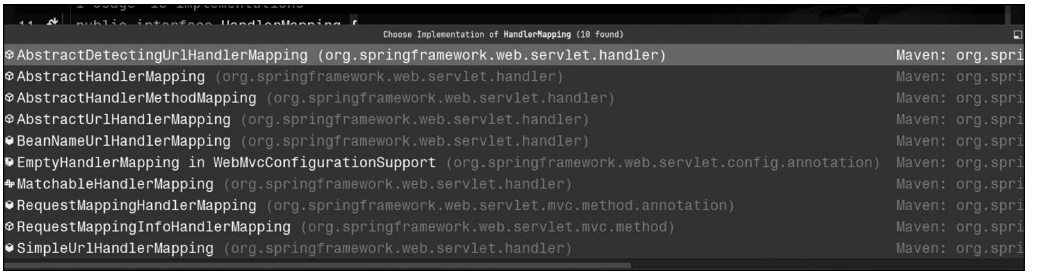

图 3-9 handlerMapper 实现类

(1) SimpleUrlHandlerMapper：在应用上下文中可以进行配置，并且有 Ant 风格的路径匹配功能。例如在 springmvc.xml 中配置一个 SimpleUrlHandlerMapping 处理器映射。

在 springmvc.xml 文件中添加以下信息，代码如下：

```
//第 3 章 springmv.xml
<bean
    class = "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
                    <property name = "mappings">
                        <props>
                            <prop key = "/hello.do"> helloController </prop>
                        </props>
                    </property>
                </bean>
    <bean id = "helloController" class = "com.panther.controller.HelloController"/>
```

对应的 Controller 代码如下：

```
//第 3 章 HelloController.java
public class HelloController implements Controller {

        @Override
        public ModelAndView handleRequest(HttpServletRequest
    httpServletRequest,HttpServletResponse httpServletResponse) throws
    Exception {
```

```
        ModelAndView mv = new ModelAndView("success");
        return mv;
    }
}
```

(2) BeanNameUrlHandlerMapping。

spring.xml 的配置,代码如下:

```
//第 3 章 spring.xml
<!-- 1.创建 BeanNameUrlHandlerMapping -->
<bean
    class = "org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<!-- 2.创建 Controller 对象,这里的 id 必须是页面访问的路径(以斜杠开头) -->
<bean id = "/hello.do" class = "com.panther.controller.HelloController"/>
```

(3) RequestMappingHandlerMapping:这是 3 个中最常用的 HandlerMapping,因为注解方式比较通俗易懂,代码清晰,只需在代码前加上@RequestMapping()的相关注解就可以了。

代码如下:

```
//第 3 章 HelloController.java
@Controller
public class HelloController {
//如果没加 ResponseBody,则返回就是视图
@RequestMapping("/hello.do")
    public String index() {
        return "/hello";
    }
}
```

2. 处理适配器

HandlerAdapter 字面上的意思就是处理适配器,它的作用用一句话概括就是调用具体的方法对用户发来的请求进行处理。当 HandlerMapping 获取执行请求的 Controller 时,DispatcherServlet 会根据 Controller 对应的类型来调用相应的 HandlerAdapter 进行处理。

HandlerAdapter 的实现有 HttpRequestHandlerAdapter、SimpleServletHandlerAdapter、SimpleControllerHandlerAdapter、AnnotationMethodHandlerAdapter(Spring MVC 3.1 后已废弃)和 RequestMappingHandlerAdapter。实现类图如图 3-10 所示。

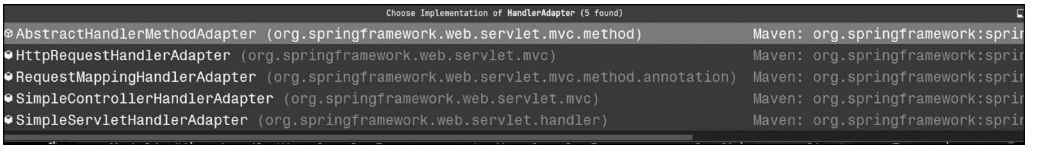

图 3-10 HandlerAdapter 实现类

（1）HttpRequestHandlerAdapter：处理类型为 HttpRequestHandler 的 handler，对 handler 的处理是调用 HttpRequestHandler 的 handleRequest()方法。

Controller 类实现的代码如下：

```
//第 3 章 HelloController.java
public class HelloController implements HttpRequestHandler {
    @Override
    public void handleRequest(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        response.getWriter().write("<p>hello Spring MVC!</p>");
    }
}
```

还需要在 Spring MVC 中写入 HttpRequestHandlerAdapter 才会使用当前的 Adapter，代码如下：

```
//第 3 章 springmvc.xml
<!-- 1.创建 BeanNameUrlHandlerMapping -->
<bean
    class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
<!-- 2.创建 HttpRequestHandlerAdapter -->
<bean
    class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"/>

<!-- 3.创建 Controller 对象,这里的 id 必须是页面访问的路径(以斜杠开头) -->
<bean id="/hello.do" class="com.panther.controller.HelloController"/>
```

（2）SimpleServletHandlerAdapter：处理类型为 Servlet，也就是把 Servlet 当作 Controller 来处理，使用 Servlet 的 service 方法处理用户请求。

Controller 类在 Spring MVC 中还是需要注入 SimpleServletHandlerAdapter 才能使用当前的适配器，代码如下：

```
//第 3 章 HelloController.java
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        resp.getWriter().write("<p style= color: red;>hello Spring MVC!</p>");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        super.doGet(req,resp);
    }
}
```

（3）SimpleControllerHandlerAdapter：处理类型为 Controller 的控制器，使用

Controller 的 handlerRequest 方法处理用户请求。

Controller 实现的代码如下：

```
//第 3 章 HelloController.java
public class hellocontroller implements Controller {

    @Override
      public ModelAndView handleRequest ( HttpServletRequest request, HttpServletResponse
response) throws Exception {
        response.getWriter().write("Hello - www.yiidian.com");
        return null;
    }
}
```

（4）RequestMappingHandlerAdapter：处理类型为 HandlerMethod 的控制器，通过 Java 反射调用 HandlerMethod 的方法来处理用户请求。

Controller 实现的代码如下：

```
//第 3 章 HelloController.java
@Controller
public class HelloController {
    @RequestMapping("/hello.do")
    public String index() {
        return "< p style = color: red;> hello Spring MVC! </p >";
    }
}
```

3. 视图解析器

Spring MVC 中的视图解析器的主要作用就是将逻辑视图转换成用户可以看到的物理视图。

当用户对 Spring MVC 应用程序发起请求时，这些请求都会被 Spring MVC 的 DispatcherServlet 处理，通过处理器找到最合适的 HandlerMapping 定义的请求映射中最合适的映射，然后通过 HandlerMapping 找到相对应的 Handler，再通过相对应的 HandlerAdapter 处理该 Handler。返回结果是一个 ModelAndView 对象，当该 ModelAndView 对象中不包含真正的视图而是一个逻辑视图路径时，ViewResolver 就会把该逻辑视图路径解析为真正的 View 视图对象，然后通过 View 的渲染，将最终结果返给用户。

Spring MVC 中处理视图最终要的两个接口就是 ViewResolver 和 View，ViewResolver 的作用是将逻辑视图解析成物理视图，View 的主要作用是调用其 render()方法对物理视图进行渲染。

Spring MVC 提供常见视图解析器，如表 3-1 所示。

表 3-1 视图解析器

视图类型	说明
BeanNameViewResolver	将逻辑视图名称解析为一个 Bean，Bean 的 ID 等于逻辑视图名
InternalResourceViewResolver	将视图名解析为一个 URL 文件，一般使用该解析器将视图名映射为一个保存在 WEB-INF 目录下的程序文件，如 JSP
JaperReportsViewResolver	JaperReports 是基于 Java 的开源报表工具，该解析器解析为报表文件对应的 URL
FreeMarkerViewResolver	解析为基于 FreeMarker 模板的模板文件
VelocityViewResolver	解析为 Velocity 模板技术的模板文件
VelocityLayoutViewResolver	解析为 Velocity 模板技术的模板文件

3.3 Spring MVC 的注解和配置

在 Spring MVC 的世界里，注解和配置是构建现代 Web 应用程序的核心元素。它们为开发者提供了简洁、灵活的方式来定义和处理 Web 请求，从而大大地提升了开发效率。在数字化浪潮的涌动下，Web 应用程序成为连接世界的桥梁。在这个桥梁的搭建过程中，Spring MVC 以其强大的注解和配置能力成为众多开发者的首选。

3.3.1 @RequestionMapping

从注解名称上可以看到，@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。Spring MVC 接收到指定的请求，就会在映射关系中找到对应的控制器方法来处理这个请求。

(1) 只要 value 是一个数组，就可以匹配多个值，代码如下：

```
//别名 path 和 value 是一样的
@AliasFor("path")
String[] value() default {};
```

请求/testRequestMapping 和/test 都是由 valueDemo 进行处理，代码如下：

```
@RequestMapping(value = {"/testRequestMapping", "/test"})
public String valueDemo(){
    return "success";
}
```

(2) Method：@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。

底层结构是数组，元素是 requestionMethod 枚举，代码如下：

```
RequestMethod[ ] method() default {};
//RequestMethod
public enum RequestMethod {
    GET,
    HEAD,
    POST,
    PUT,
    PATCH,
    DELETE,
    OPTIONS,
    TRACE;
    private RequestMethod() {
}
}
```

默认为 GET 请求，如果想改变请求方式，则有两种方式，实现代码如下：

```
//第 3 章 HelloController.java
//1. 在 RequestMapping 上指定 Method
@RequestMapping(value = "/test1" , method = RequestMethod.POST)
public String method1(){
    return "success";
}
//直接使用 MVC 提供的注解写法
@PostMapping( "/test2" )
public String method2(){
    return "success";
}
```

(3) Params：@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 4 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。

例如一个接口只允许超级管理员访问，代码如下：

```
//第 3 章 HelloController.java
@DeleteMapping(
        value = {"/test3"}
        ,params = {"username","role == 'super'"}
)
```

```
public String test3(){
    return "success";
}
```

(4) Head：@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 4 种表达式设置请求头信息和请求映射的匹配关系。使用方式和 params 一样。例如，当前接口必须经用户认证完才能访问，代码如下：

```
//第 3 章 HelloController.java
@PutMapping(
        value = {"/test3"}
        ,headers = {"auth"})
public String test4(){
    return "success";
}
```

3.3.2 @PathVariable

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

获取 URL 的部分信息充当参数传递，代码如下：

```
//第 3 章 HelloController.java
@GetMapping("/testRest/{id}")
public String testRest(@PathVariable("id") String page){
    return "success";
}
```

3.3.3 @RequestParam

@RequestParam 可以为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 3 个属性。

(1) value：指定为形参赋值的请求参数的参数名。

(2) required：设置是否必须传输此请求参数，默认值为 true。

若设置为 true，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，并且没有设置 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 所指定的请求参数的，若没有传输，则注解所标识的形参的值为 null。

(3) defaultValue：不管 required 属性值为 true 还是 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。

代码如下：

```
//第 3 章 HelloController.java
@GetMapping("/test5")
public String test5(@RequestParam(value = "name",required =
    false,defaultValue = "admin") String name){
    return "success";
}
```

3.3.4 @CookieValue

Spring MVC 提供 @CookieValue 方便我们获取指定 Cookie 数据，代码如下：

```
//第 3 章 HelloController.java
@RequestMapping("/test7")
public String save(@CookieValue(value = "sessionId",required = false) String sessionId){
        return "success";
}
```

3.3.5 @RequestBody

@RequestBody 可以获取请求体，需要在控制器方法中设置一个形参，使用@RequestBody 进行标识，当前请求的请求体就会为当前注解所标识的形参赋值。当引用类型作为参数传递时，任何对该参数对象的修改都会直接反映到原始对象上，无须额外步骤“填充”或同步这些更改，代码如下：

```
//第 3 章 HelloController.java
@RequestMapping("/test8")
public String test8(@RequestBody User user){
    return "success";
}
```

3.3.6 @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，代码如下：

```
//第 3 章 HelloController.java
@RequestMapping("/test8")
@ResponseBody
public String test8(@RequestBody User user){
    return "success";
}
```

3.3.7 修复浏览器中文乱码问题

可以在 web.xml 文件中注册字符编码，MVC 配置代码如下：

```
//第 3 章 springmvc.xml
<!-- 配置 Spring MVC 的编码过滤器，需要配置在所有过滤器的前面，否则会失效 -->
  <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>forceResponseEncoding</param-name>
      <param-value>true</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

3.4 域共享数据

在一个请求中需要存储一些信息，供同一批连接的请求共享。例如，用户登录，可以先将用户脱敏后的信息存在 Session 中，再请求就可以获得登录用户信息。

3.4.1 使用 ServletAPI 向 request 域对象共享数据

在 Java Web 应用中，ServletAPI 提供了一系列用于处理 HTTP 请求和响应的类和接口，其中，HttpServletRequest 对象是一个重要的组成部分，它代表了客户端发送给服务器的 HTTP 请求。这个对象包含了请求的所有信息，如请求头、请求参数、请求方法(GET、POST 等)等。

HttpServletRequest 对象同时也作为一个域对象(Scope Object)使用，这意味着它可以用来存储和共享数据。这些数据可以在同一个请求的生命周期内被多个组件(如 Servlet、JSP 等)访问。这种机制使在不同组件之间传递数据变得非常方便。

实现数据共享，代码如下：

```
//第 3 章 HelloController.java
@RequestMapping("/testServletAPI")
public String testServletAPI(HttpServletRequest request){
```

```
        request.setAttribute("auth", "admin");
        return "success";
    }
    //相同连接的下次请求接口
    @RequestMapping("/test8")
    public String testServletAPI(HttpServletRequest request){
    //返回值为 Object,可以先取出上次接口存入的值,然后进行操作
    String role = (String)request.getAttribute("auth");
    If(role.length == 0 || !"admin".equals(role))
            return "404"
        return "success";
    }
```

3.4.2 使用 ServletAPI 向 session 域对象共享数据

在 Java Web 应用程序中,session 是一个用于存储与特定用户会话相关联的数据的对象。当用户首次访问 Web 应用程序时,服务器会为该用户创建一个新的会话(如果尚未存在),并分配一个唯一的会话 ID,这个 ID 通常通过 Cookie 或 URL 重写的方式发送到客户端。之后,用户的后续请求将包含这个会话 ID,以便服务器能够识别并加载与该用户会话相关联的数据。

ServletAPI 提供了与 session 对象进行交互的接口。HttpSession 是 javax.servlet.http 包中的一个接口,它表示一个与某个用户会话相关联的会话。通过 HttpSession 对象,可以在不同的 Servlet 或 JSP 页面之间共享数据。

实现数据共享,代码如下:

```
//第 3 章 HelloController.java
@RequestMapping("/testSession")
public String testSession(HttpServletRequest request){
    request.getSession().setAttribute("testScope", "hello,servletAPI");
return "success";
}
```

3.4.3 使用 ModelAndView 向 request 域对象共享数据

在 Spring MVC 框架中,ModelAndView 是一个非常重要的类,它用于封装模型数据(业务数据)和视图名称(要渲染的页面)。通过 ModelAndView,可以将数据从控制器(Controller)传递到视图(View),以便在视图页面中展示这些数据。

当使用 ModelAndView 来共享数据时,实际上是将数据添加到 Model 中,而 Model 本质上是一个 Map,它存储了键-值对形式的数据。这些数据在请求处理过程中会被存储在 HttpServletRequest 的属性(attribute)中,因此它们可以在视图中被访问。

实现数据共享,代码如下:

```
//第 3 章 HelloController.java
@RequestMapping("/testModelAndView")
public ModelAndView testModelAndView(){
        /**
         * ModelAndView 有 Model 和 View 的功能
         * Model 主要用于向请求域共享数据
         * View 主要用于设置视图,实现页面跳转
         */
        ModelAndView mav = new ModelAndView();
        //向请求域共享数据
        mav.addObject("testScope", "hello,ModelAndView");
        //设置视图,实现页面跳转
        mav.setViewName("index");
        return mav;
}
```

3.4.4 使用 Model 向 request 域对象共享数据

在 Java Web 开发中,特别是使用 Servlet 和 JSP 技术时,我们经常需要在多个组件之间共享数据。一种常见的场景是将数据从后端(如 Servlet)传递到前端(如 JSP 页面)。request 域对象是一个用于存储和共享数据的机制,它允许我们在处理 HTTP 请求的不同阶段(如从 Servlet 到 JSP)之间传递数据。

当使用 Model 向 request 域对象共享数据时,我们实际上是在利用某种框架(如 Spring MVC)提供的功能来简化这个过程。在 Spring MVC 中,Model 是一个接口,它通常被实现为 ModelMap 或 ModelAndView 中的一个组件,用于存储要在视图中显示的数据。

实现数据共享,代码如下:

```
//第 3 章 HelloController.java
@RequestMapping("/testModel")
public String testModel(Model model){
    model.addAttribute("testScope", "hello,Model");
    return "success";
}
```

3.4.5 使用 ModelMap 向 request 域对象共享数据

在 Spring MVC 中,ModelMap 是一个接口,它继承自 Map < String, Object >,通常用于在 Controller 和 View(通常是 JSP 页面)之间传递数据。当 Controller 处理一个 HTTP 请求时,它可以创建一个 ModelMap 实例,并将需要的数据作为属性添加到这个 Map 中。这些数据随后会被自动添加到 HTTP 请求的 request 域对象中,并可以在 View 层(如 JSP 页面)中通过 EL 表达式或其他方式访问。

实现数据共享,代码如下:

```
//第 3 章 HelloController.java
@RequestMapping("/testModelMap")
public String testModelMap(ModelMap modelMap){
    modelMap.addAttribute("testScope", "hello,ModelMap");
    return "success";
}
```

3.4.6 使用 Map 向 request 域对象共享数据

在 Java Web 开发中，使用 Map 向 request 域对象共享数据通常发生在不使用高级框架(如 Spring MVC)的情况下，而是直接使用 Servlet 和 JSP。request 域对象是一个存储与当前 HTTP 请求相关联的属性的对象，这些属性可以在处理请求的过程中被访问。

当在 Servlet 中处理数据并在随后的 JSP 页面中显示这些数据时，可以将数据放入 request 对象中。由于 request 对象实现了 javax.servlet.http.HttpServletRequest 接口，所以它提供了 setAttribute(String name,Object value)方法，允许将数据以键-值对的形式存储在 request 域对象中。

Map 是一个接口，它定义了存储键-值对的数据结构。在 Servlet 中，可以使用 Map 来收集和准备数据，然后使用 setAttribute()方法将这些数据添加到 request 对象中。

实现数据共享，代码如下：

```
//第 3 章 HelloController.java
private static Map<String, Object> map =
        new ConcurrentHashMap<>(16,0.75f,16);
@RequestMapping("/testMap")
public String testMap(){
    map.put("testScope", "hello,Map");
    return "success";
}
```

3.4.7 Model、ModelMap、Map 的关系

Model、ModelMap、Map 类型的参数其实本质上都是 BindingAwareModelMap 类型，具体如下：

(1) Model 是 Model 接口的顶级父接口。

(2) ModelMap 实现了 LikedMap，从而间接地实现了 Map。

(3) Map 是 Map 的顶级接口。

(4) BindingAwareModelMap 继承了 ExtendedModelMap，而 ExtendedModelMap 继承了 ModelMap 和实现了 Model 的接口，所以联系在一起。

3.4.8 向 application 域共享数据

在 Java Web 应用中，application 域是一个特殊的域对象，它用于在整个 Web 应用程序

的生命周期内存储和共享数据。与 request、session 和 page 域相比，application 域的数据在 Web 应用程序启动时被加载，并且会一直存在直到 Web 应用程序停止或重启。

实现数据共享的代码如下：

```
//第 3 章 HelloController.java
@RequestMapping("/testApplication")
public String testApplication(){
        RequestAttributes requestAttributes =
    RequestContextHolder.currentRequestAttributes();
        HttpServletRequest httpServletRequest = (( ServletRequestAttributes )
requestAttributes).getRequest();
        ServletContext application = httpServletRequest.getServletContext();
        application.setAttribute("testApplicationScope",
    "hello,application");
        return "success";
}
```

3.5 Spring MVC 的参数绑定和数据转换

在 Spring MVC 的广阔世界中，参数绑定和数据转换是构建高效、灵活的 Web 应用程序的关键环节。想象一下，设计一个在线购物网站，当用户需要完成浏览商品、将商品添加到购物车、提交订单等操作时，这些操作的背后都需要一系列参数与后端代码的精准绑定和转换。这就是 Spring MVC 参数绑定和数据转换的魔力所在。

3.5.1 基本参数类型封装

1. 设计请求表单

后面的演示将不再展示 HTML 的代码，请求表单的代码如下：

```
//第 3 章 index.html
<h2>基本类型参数封装</h2>
<form action="http://localhost:8080/param.do">
    用户名:<input type="text" name="username"><br>
    年龄:<input type="text" name="age"><br>
    <input type="submit" value="提交">
</form>
```

2. 编写 Controller 接受参数

这里需要注意的是，控制器接收参数的形参名称必须和表单的 name 属性保持一致，否则会接收失败。接收的参数会被自动地映射到对应的字段中，代码如下：

```
//第 3 章 HelloController.java
@Controller
public class ParamController {
```

```
    @RequestMapping("/param.do")
    public String save(String username,Integer age){
        System.out.println("用户名:" + username);
        System.out.println("年龄:" + age);
        return "success";
    }
}
```

3. 运行测试

打开 HTML 页面，如图 3-11 所示。

单击“提交”按钮后控制台输出 username 和 age，如图 3-12 所示。

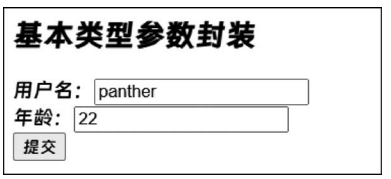

图 3-11　表单页面

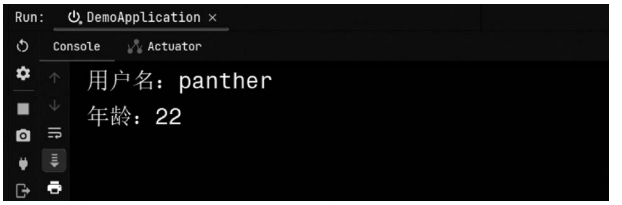

图 3-12　控制台输出

3.5.2　实体类型封装

将 username 和 age 封装成一个实体类，代码如下：

```
//第 3 章 User.java
public class User {
    private String username;
private Integer age;
//省略 get 和 set 方法
}
```

将 controller 的参数类型替换成 user，代码如下：

```
//第 3 章 HelloController.java
@Controller
public class ParamController {
    @RequestMapping("/param.do")
    public String save(User user){
        System.out.println("用户名:" + user.getUsername());
        System.out.println("年龄:" + user.getAge());
        return "success";
    }
}
```

运行测试，表单数据如图 3-13 所示。

控制台输出对应的数据，如图 3-14 所示。

图 3-13　表单页面

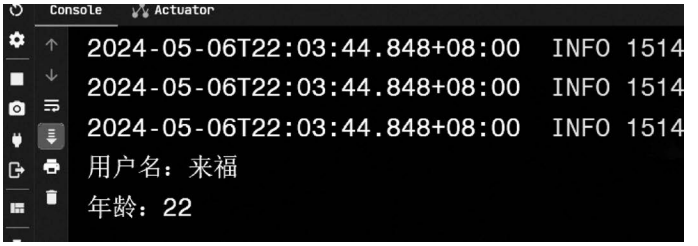

图 3-14　控制台输出

3.5.3　存在引用参数封装

在 Spring MVC 的应用过程中，在后端根据需要将表单数据封装在一个包装 Pojo 类型中，所谓包装 Pojo 类型，就是 Pojo 对象中包含另一个 Pojo 对象，代码如下：

```
//第 3 章 User.java
public class User {
    private String username;
    private Integer age;
private Address address;
//省略 set 和 get 方法
}
//address 类
public class Address {
    private String province;
    private String city;
}
```

这里封装用户的地址信息，name 为 address.province 这种写法，这代表把数据封装到 User 对象→Address 对象的 province 属性中，代码如下：

```
//第 3 章 index.html
<h2>基本类型参数封装</h2>
<form action="http://localhost:8080/param.do" method="post">
    用户名:<input type="text" name="username"><br>
    年龄:<input type="text" name="age"><br>
    省份:<input type="text" name="address.province"><br>
    城市:<input type="text" name="address.city"><br>
    <input type="submit" value="提交">
</form>
```

运行测试，表单数据如图 3-15 所示。

控制台输出数据，如图 3-16 所示。

3.5.4　List 集合封装

一个 Address 对象接收一个地址信息，如果有多个地址信息，则该怎么办呢？可以使用 List 集合来封装。

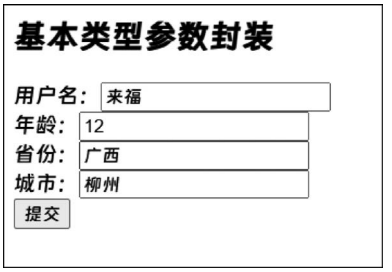

图 3-15 表单页面

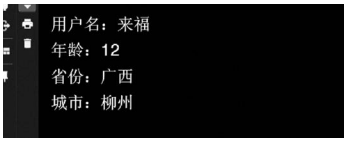

图 3-16 控制台输出

重新设计表单，代码如下：

```
//第 3 章 index.
< form action = "http://localhost:8080/param.do" method = "post">
    用户名:< input type = "text" name = "username">< br >
    年龄:< input type = "text" name = "age">< br >
    省份 1:< input type = "text" name = "address[0].province">< br >
    城市 1:< input type = "text" name = "address[0].city">< br >
    省份 2:< input type = "text" name = "address[1].province">< br >
    城市 2:< input type = "text" name = "address[1].city">< br >
    < input type = "submit" value = "提交">
</form >
```

将 user 实体类下的 address 改为 List，代码如下：

```
//第 3 章 User.java
public class User {
    private String username;
    private Integer age;
    private List < Address > address;
}
```

Controller 类中的输出也需要使用 for 循环来输出地址，代码如下：

```
//第 3 章 HelloController.java
@RequestMapping("/param.do")
@ResponseBody
public String save(User user){
    System.out.println("用户名:" + user.getUsername());
    System.out.println("年龄:" + user.getAge());
    //遍历所有地址信息
    for(Address addr:user.getAddress()){
        System.out.println(addr);
    }
    return "success";
}
```

测试运行，先填写表单数据，如图 3-17 所示。

控制台输出对应的数据，如图 3-18 所示。

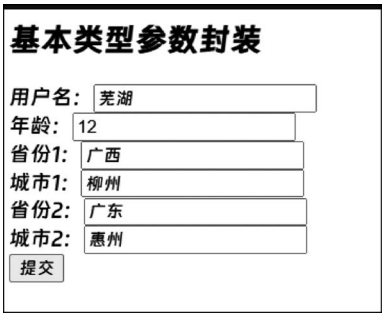

图 3-17 表单页面

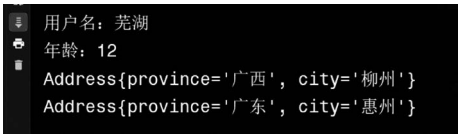

图 3-18 控制台输出

3.5.5 Map 集合封装

3.5.4 节利用 List 集合封装了多个地址信息，其实把 List 集合换成 Map 集合也是可以的。Spring MVC 如何使用 Map 集合类型封装表单参数呢？

重新改写表单，代码如下：

```
//第 3 章 index.html
<h2>基本类型参数封装</h2>
<form action="http://localhost:8080/param.do" method="post">
    用户名:<input type="text" name="username"><br>
    年龄:<input type="text" name="age"><br>
    省份 1:<input type="text" name="address['a1'].province"><br>
    城市 1:<input type="text" name="address['a1'].city"><br>
    省份 2:<input type="text" name="address['a2'].province"><br>
    城市 2:<input type="text" name="address['a2'].city"><br>
    <input type="submit" value="提交">
</form>
```

对于这里的 address['a1'].city，a1 是赋值给 Map 的 key，city 是赋值给 Address 的 city 属性，User 类中的 Address 字段需要转换成 Map 类型，代码如下：

```
//第 3 章 User.java
public class User {
    private String username;
    private Integer age;
private Map<String,Address> address;
//省略 get 和 set 方法
}
```

Controller 层遍历地址集合的方法需要修改成遍历 map 的方式，代码如下：

```
//第 3 章 HelloController.java
@RequestMapping("/param.do")
@ResponseBody
```

```
public String save(User user){
    System.out.println("用户名:" + user.getUsername());
    System.out.println("年龄:" + user.getAge());
    //遍历所有地址信息
    Map < String, Address > address = user.getAddress();
    for(Map.Entry < String,Address > entry : address.entrySet()){
        System.out.println(entry.getKey() + " -- " + entry.getValue());
    }
    return "success";
}
```

运行测试,在表单中输入对应的数据,数据如图 3-19 所示。

控制台输出对应的数据,如图 3-20 所示。

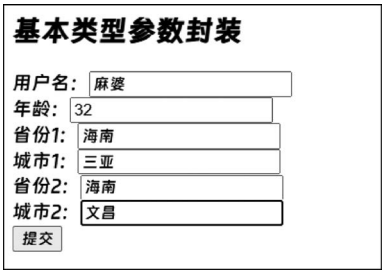

图 3-19 表单页面

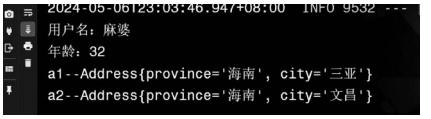

图 3-20 控制台输出

3.5.6 自定义类型转换器

在 Spring MVC 中,自定义类型转换器(Type Converter)或格式化器(Formatter)可以帮助开发者将请求中的参数转换为控制器方法中所需的特定类型。这在需要处理非标准的字符串,在对象的转换时特别有用。

Spring MVC 在默认情况下可以对基本类型进行类型转换,例如可以将 String 转换为 Integer、Double、Float 等,但是 Spring MVC 并不能转换日期类型(java.util.Date),如果希望把字符串参数转换为日期类型,则必须自定义类型转换器。接下来讲解如何自定义类型转换器。

先将表单重新设计为两个字段,代码如下:

```
//第 3 章 index.html
< form action = "http://localhost:8080/param.do" method = "post">
    用户名:< input type = "text" name = "username">< br >
    生日:< input type = "text" name = "birthday">< br >
    < input type = "submit" value = "提交">
</form >
```

重新编写一个实体类来接受这两个字段,代码如下:

```
//第 3 章 master.java
public class master {
private String username;
//这里接收的是 java.util.Date 类型
private Date birthday;
//get 和 set
}
```

Spring MVC 的自定义类型转换器必须实现 Converter 接口，自己编写一个 String 参数转化成 Date 类型的转换器，代码如下：

```
//第 3 章 StringToDateConverter.java
public class StringToDateConverter implements Converter < String, Date > {
    @Override
    public Date convert(String source) {
        Date date = null;
        try {
            //使用 SimpleDateFormat 将页面字符串日期转换为 java.util.Date 类型
            date = new SimpleDateFormat("yyyy - MM - dd").parse(source);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        return date;
    }
}
```

编写完自定义 Converter 后需要将它设置进 ConversionServiceFactoryBean 中才能生效，具体的代码如下：

```
//第 3 章 WebMvcConfig.java
@Configuration
public class WebMvcConfig {
    @Bean
    public WebMvcConfigurer webMvcConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addFormatters(FormatterRegistry registry) {
                registry.addConverter(new StringToDateConverter());
            }
        };
    }
}
```

修改 Controller，将接受参数类型改成 Master，代码如下：

```
//第 3 章 HelloController.java
@RequestMapping("/param.do")
@ResponseBody
public String save(master master){
    System.out.println("用户名:" + master.getUsername());
```

```
        System.out.println("生日:" + master.getBirthday());
        return "success";
    }
```

运行测试,输入表单数据,如图 3-21 所示。

控制台输出的数据如图 3-22 所示。

图 3-21 表单数据

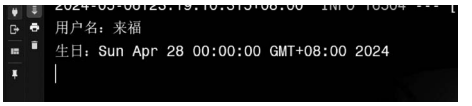

图 3-22 控制台数据

3.6 拦截器

Spring MVC 拦截器(Interceptor)是 Spring 框架提供的一种机制,用于在请求处理之前、之后或请求处理发生异常时执行一些操作。这些操作包括日志记录、身份验证、授权、性能监控、数据转换等。拦截器对于 AOP 的实现非常有用,因为它允许在不修改现有代码的情况下添加额外的功能。

系统中除了登录方法,其他所有方法都需要先验证用户是否登录了,若未登录,则让用户先跳转到登录页面,最笨的方法是在所有需要验证的方法内部都加上验证的代码,那么有没有更好的方法呢?

Spring MVC 确实为我们考虑到了这种需求,Spring MVC 在处理流程中提供了 3 个扩展点可以对整个处理流程进行干预,这个就是 Spring MVC 中拦截器提供的功能,代码如下:

```
//第 3 章 HandlerInterceptor.java
public interface HandlerInterceptor {
     default boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler){
//在调用自定义的 controller 之前会调用这种方法,若返回值为 false,则跳过 controller 方法的调
//用,否则将进入 controller 的方法中
         return true;
    }
default void postHandle(HttpServletRequest request, HttpServletResponse response, Object
handler, @Nullable ModelAndView modelAndView) {
//调用自定义 controller 中的方法之后会调用这种方法,此时还没有渲染视图,也就是还没有将结
//果输出到客户端
    }
```

```
default void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object
handler,@Nullable Exception ex) {
//整个请求处理完毕后,在将结果输出到客户端之后调用这种方法,此时可以做一些清理的工作,注意
//这种方法的最后一个参数是 Exception 类型的,说明这种方法不管整个过程是否有异常都会被调用
    }
}
```

书写拦截器的拦截处理逻辑,代码如下:

```
//第 3 章 LoginInterceptor.java
@Component
public class LoginInterceptor implements HandlerInterceptor {
    @Override
     public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        System.out.println(this.getClass().getSimpleName() + ".preHandle");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object
handler, ModelAndView modelAndView) throws Exception {
        System.out.println(this.getClass().getSimpleName() + ".postHandle");
    }

    @Override
     public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
Object handler, Exception ex) throws Exception {
        System.out.println(this.getClass().getSimpleName() + ".afterCompletion");
    }
}
```

在 MVCConfig 配置类中注册拦截器,然后配置拦截规则,具体的代码如下:

```
//第 3 章 WebMvcConfig.java
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //拦截所有请求,决定是否需要登录

    registry.addInterceptor(LoginInterceptor()).addPathPatterns("/**");
    }
    @Bean
    public LoginInterceptor LoginInterceptor() {
        return new LoginInterceptor();
    }
}
```

运行测试,测试结果如图 3-23 所示。

声明一个注解,代码如下:

```
LoginInterceptor.preHandle
test
LoginInterceptor.postHandle
LoginInterceptor.afterCompletion
```

图 3-23　控制台数据

```
//第 3 章 LoginCheck.java
@Target({ElementType.METHOD})                    //可用在方法名上
@Retention(RetentionPolicy.RUNTIME)              //运行时有效
public @interface LoginCheck {
}
```

获取注解并实现拦截逻辑，代码如下：

```
//第 3 章 AuthorityInterceptor.java
public class AuthorityInterceptor implements HandlerInterceptor {
    @Override
     public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        //如果不是映射到方法，则直接通过
        if (!(handler instanceof HandlerMethod)) {
            return true;
        }
        HandlerMethod handlerMethod = (HandlerMethod) handler;
        Method method = handlerMethod.getMethod();
        //判断接口是否需要登录
        LoginCheck LoginCheck = method.getAnnotation(LoginCheck.class);
        //有 @LoginRequired 注解，需要认证
        if (LoginCheck != null) {
            //这里写拦截后需要做什么，例如取缓存、Session、权限判断等
            System.out.println("LoginCheck...");
            return true;
        }
        return true;
    }
}
```

在 MVCConfig 配置类中注册拦截器，然后配置拦截规则，代码如下：

```
//第 3 章 WebMvcConfig.java
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //拦截所有请求，通过判断是否有 @LoginRequired 注解来决定是否需要登录

    registry.addInterceptor(AuthorityInterceptor()).addPathPatterns("/**");
    }
    @Bean
    public AuthorityInterceptor AuthorityInterceptor() {
        return new AuthorityInterceptor();
    }
}
```

运行测试，测试结果如图 3-24 所示。

```
LoginCheck...
test
```

图 3-24 控制台数据

3.7 文件上传和下载

在 Web 开发中，文件上传和下载是常见的功能需求。使用 Spring MVC 框架可以轻松地实现这些功能，为用户提供灵活的文件操作体验。本节将引入 Spring MVC 文件上传和下载的主题。

3.7.1 文件上传

文件上传是表现层常见的需求，在 Spring MVC 中底层使用 Apache 的 Commons FileUpload 工具来完成文件上传，对其进行封装，让开发者使用起来更加方便。接下来讲解如何开发。

引入依赖，代码如下：

```
//第 3 章 pom.xml
<!-- commons - fileUpload -->
<dependency>
    <groupId>commons - fileupload</groupId>
    <artifactId>commons - fileupload</artifactId>
    <version>1.3.1</version>
</dependency>
```

接下来将文件解析器配置到 IoC 容器中，该解析器的 id 必须为 multipartResolver，否则无法成功接收文件，代码如下：

```
//第 3 章 WebMvcConfig.java
@Configuration
public class WebMvcConfig {
    @Bean
    public MultipartConfigElement multipartConfigElement() {
        MultipartConfigFactory factory = new MultipartConfigFactory();
        //单个文件最大
        factory.setMaxFileSize(DataSize.ofMegabytes(10)); //MB
        //设置上传数据总大小
        factory.setMaxRequestSize(DataSize.ofMegabytes(10)); //MB
        return factory.createMultipartConfig();
    }
}
```

将表单设置为 multipart/form-data 结束参数，并且表单的请求方式只能是 POST，代

码如下：

```
//第 3 章 idnex.html
< h3 >以 Spring MVC 方式上传文件</h3 >
< form action = " http://localhost: 8080/upload" method = " post" enctype = " multipart/form - data">
    选择文件 :< input type = "file" name = "imgFile"> < br/>
    文件描述 :< input type = "text" name = "memo"> < br/>
    < input type = "submit" value = "上传">
</form >
```

接下来就是 Controller 层代码 Spring MVC 提供了 MultipartFile 类来接受文件，代码如下：

```
//第 3 章 HelloController.java
@RequestMapping("/upload")
@ResponseBody
public String upload(MultipartFile imgFile, String memo){
        String upload = "D:\\download\\java\\file";
        //判断该目录是否存在，如果不存在，则自动创建
        File uploadFile = new File(upload);
        if(!uploadFile.exists()){
            uploadFile.mkdir();
        }
        //原来的文件名
        String oldName = imgFile.getOriginalFilename();
        //时间为文件名
        LocalDate currentDate = LocalDate.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy - MM - dd");
        //格式化当前日期
        String formattedDate = currentDate.format(formatter);
        //获取文件后缀
        String extName = oldName.substring(oldName.lastIndexOf(".")); //.jpg
        String fileName = formattedDate + extName;
        //保存
        try {
            imgFile.transferTo(new File(upload + "\\" + fileName));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return "< script > alert('文件上传成功 文件描述 :" + memo + "')</script >";
}
```

运行测试，前端数据如图 3-25 所示。

单击“上传”按钮查看返回信息，页面数据如图 3-26 所示。

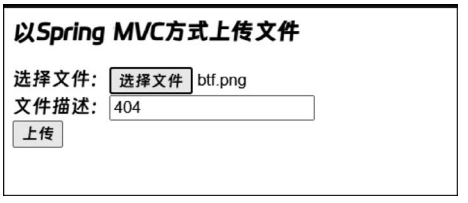

图 3-25 表单数据

图 3-26 页面数据

进入目标文件夹查看文件是否存在，如图 3-27 所示。

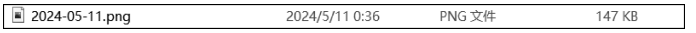

图 3-27　查看文件

3.7.2　文件下载

在 Spring MVC 中，文件下载是一个常见的功能，它允许用户从服务器获取文件并保存到本地计算机。

设计下载链接，代码如下：

```
//第 3 章 index.html
<h3>Spring MVC 文件下载</h3>
<a href="http://localhost:8080/down">下载</a>
```

编写 Controller 代码，代码如下：

```
//第 3 章 HelloController.java
@RequestMapping("/down")
public void upload(HttpServletResponse response) throws Exception {
    InputStream inputStream = null;
    OutputStream outputStream = null;
    try {
        inputStream = new FileInputStream("D:\\download\\java\\file\\404.png");
        response.setHeader("Content-Disposition", "attachment;filename=404.png");
        outputStream = response.getOutputStream();
        byte[] buf = new byte[1024];
        int len = 0;
        while ((len = inputStream.read(buf)) != -1) {
            outputStream.write(buf, 0, len);
        }
    } finally {
        //关闭资源
        if(outputStream != null)
            outputStream.close();
        if(inputStream != null)
            inputStream.close();
    }
}
```

运行测试，下载链接如图 3-28 所示。

单击“下载”按钮，页面会弹出如图 3-29 所示的下载界面。

图 3-28　下载链接

图 3-29　下载界面

3.8 MVC 一次请求的详细过程分析

了解一个请求是如何被 Spring MVC 处理的，由于整个流程涉及的代码非常多，所以本文的重点在于解析整体的流程，主要讲解 DispatcherServlet 这个核心类，弄懂这个流程，才能更好地理解具体的源码，回过头再来看则会豁然开朗。

MVC 的请求流程，如图 3-30 所示。

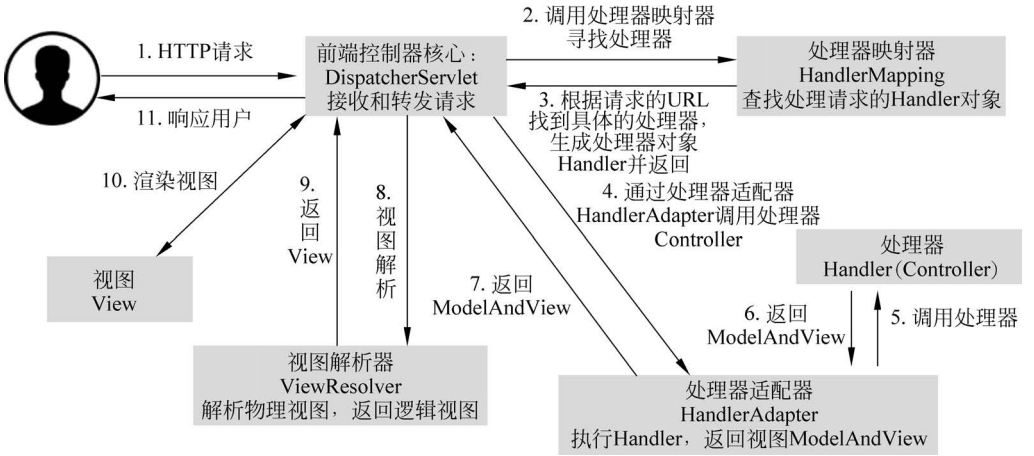

图 3-30 MVC 的请求流程

最后返给用户。

3.8.1 认识组件

Spring MVC 的关键组件包括 DispatcherServlet、MultipartResolver、HandlerMapping、HandlerAdapter、HandlerExceptionResolver、RequestToViewNameTranslator 、LocaleResolver、ThemeResolver、ViewResolver 和 FlashMapManager。它们协作处理 Web 请求、异常、视图解析和国际化，构建了一个灵活、可扩展的 MVC 框架，如表 3-2 所示。

表 3-2 MVC 常用组件介绍

组　　件	说　　明
DispatcherServlet	Spring MVC 的核心组件是请求的入口，负责协调各个组件工作
MultipartResolver	内容类型(Content-Type)为 multipart/ * 的请求的解析器，例如解析处理文件上传的请求，便于获取参数信息及上传的文件
HandlerMapping	请求的处理器匹配器，负责为请求找到合适的 HandlerExecutionChain 处理器执行链，包含处理器(handler)和拦截器(interceptors)

续表

组　　件	说　　明
HandlerAdapter	处理器的适配器。因为处理器 handler 的类型是 Object 类型，需要有一个调用者来实现 handler 是怎么被执行的。Spring 中的处理器的实现多变，例如用户处理器可以实现 Controller 接口、HttpRequestHandler 接口，也可以用@RequestMapping 注解将方法作为一个处理器等，这就导致 Spring MVC 无法直接执行这个处理器，所以这里需要一个处理器适配器，由它去执行处理器
HandlerExceptionResolver	处理器异常解析器，将处理器(handler)执行时发生的异常解析(转换)成对应的 ModelAndView 结果
RequestToViewNameTranslator	视图名称转换器，用于解析出请求的默认视图名
LocaleResolver	本地化(国际化)解析器，提供国际化支持
ThemeResolver	主题解析器，提供可设置应用整体样式风格的支持
ViewResolver	视图解析器，根据视图名和国际化，获得最终的视图 View 对象
FlashMapManager	FlashMap 管理器，负责在重定向时将参数保存至临时存储(默认 Session)

3.8.2 DispatcherServlet

通过观察 DispatcherServlet 的继承关系，就能发现它最终继承的是 JavaWeb 的 Servlet。DispatcherServlet 的继承类如图 3-31 所示。

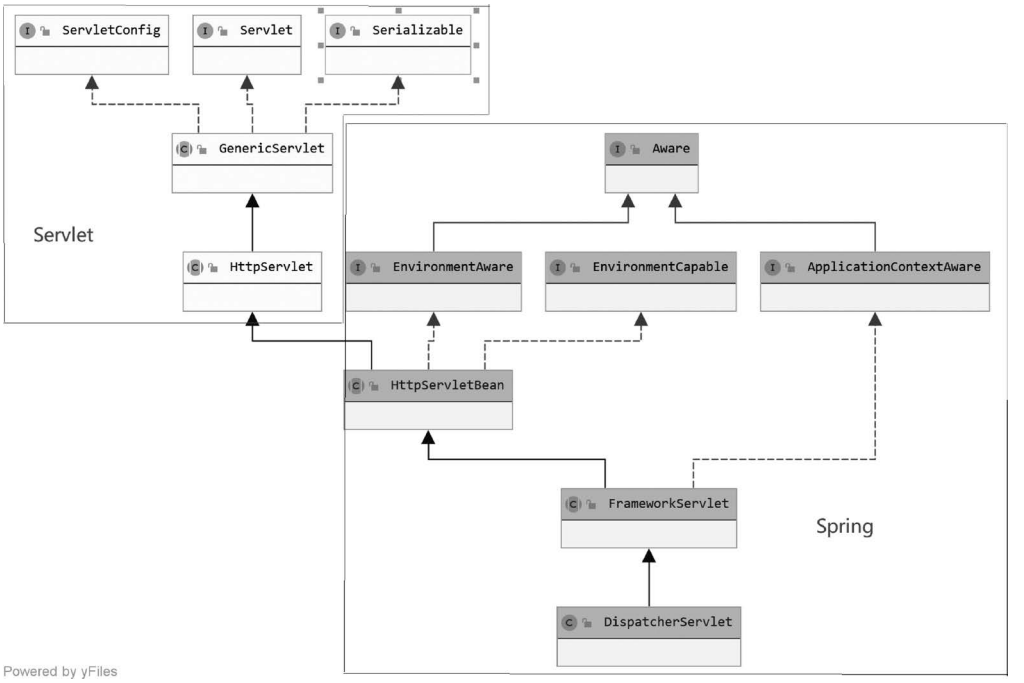

图 3-31 DispatcherServlet 的继承类

学习 Java Web 的时候可以了解到 Servlet 最终会实现两种方法 doGet 和 doPost，而这两种方法最终又会调用 processRequest()这种方法。

在 FrameworkServlet 类中有一段集合，代码如下：

```
private static final Set < String > HTTP_SERVLET_METHODS = Set.of("DELETE", "HEAD", "GET",
"OPTIONS","POST", "PUT", "TRACE");
```

表示可以处理的请求和剩下的请求方式直接由 processRequest()处理，如图 3-32 所示。

图 3-32 processRequest()处理结果

对于 OPTIONS 和 TRACE 方法 Servlet 采取了不同的处理方式，Optional 处理代码如下：

```
//第 3 章 FrameworkServlet.java
@Override
protected void doOptions(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    //如果 dispatchOptionsRequest 为 true,则处理该请求,默认为 true
    if (this.dispatchOptionsRequest || CorsUtils.isPreFlightRequest(request)) {
        //处理请求
        processRequest(request, response);
        //如果响应 Header 包含 "Allow",则不需要交给父方法处理
        if (response.containsHeader("Allow")) {
            return;
        }
}
//调用父方法,并在响应 Header 的 Allow 增加 PATCH 的值
    super.doOptions(request, new HttpServletResponseWrapper(response) {
        @Override
        public void setHeader(String name, String value) {
            if ("Allow".equals(name)) {
                value = (StringUtils.hasLength(value) ? value + ", " : "") + HttpMethod.
PATCH.name();
            }
            super.setHeader(name, value);
        }
    });
}
```

OPTIONS 请求通常会在发送 POST 请求前进行发送，以此来判断当前服务器能否处理当前请求。Trace 的代码如下：

```
//第 3 章 FrameworkServlet.java
protected void doTrace(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    //如果 dispatchTraceRequest 为 true,则处理该请求,默认值为 false
    if (this.dispatchTraceRequest) {
        //处理请求
        processRequest(request, response);
        //如果响应的内容类型为 message/http,则不需要交给父方法处理
        if ("message/http".equals(response.getContentType())) {
            //Proper TRACE response coming from a handler - we're done
            return;
        }
    }
    //调用父方法
    super.doTrace(request, response);
}
```

Trace 请求方法主要用于在请求的路径上追踪客户端与服务器端之间的通信，用于调试和诊断网络问题，但出于安全性考虑，它通常在生产环境中被禁用，因此使用场景相对较少。

3.8.3 DoDispatch

在 Spring MVC 中，DispatcherServlet 是前端控制器的核心组件，负责接收所有的 HTTP 请求，并根据配置和逻辑来决定如何处理这些请求，而 doDispatch 方法是 DispatcherServlet 中的一个关键方法，它实现了请求的分发和处理逻辑。

分发请求的核心方法就是 DoDispatch 方法，感兴趣的读者可以自行阅读详细源码，这里以伪代码写出核心实现流程，代码如下：

```
//第 3 章 DoDispatch.java
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception {
    //获取异步管理器
    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
    try {
        ModelAndView mv = null;
        try {
            //检测请求是否为上传请求,如果是,则通过 multipartResolver 将其封装成
// MultipartHttpServletRequest 对象
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);
```

```
                //获得请求对应的 HandlerExecutionChain 对象(HandlerMethod 和
                //HandlerInterceptor 拦截器)
                mappedHandler = getHandler(processedRequest);
                if (mappedHandler == null) {
    //如果无法获取,则根据配置抛出异常或返回 404 错误
                    noHandlerFound(processedRequest, response);
                    return;
                }
                //获得当前 handler 对应的 HandlerAdapter 对象
                HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
                //处理有 Last-Modified 请求头的场景
                String method = request.getMethod();
                boolean isGet = "GET".equals(method);
                if (isGet || "HEAD".equals(method)) {
                    //获取请求中服务器端最后被修改的时间
                    long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
                    if (new ServletWebRequest ( request, response ). checkNotModified
    (lastModified) && isGet) {
                        return;
                    }
                }
                //前置处理拦截器
                //注意:该方法如果有一个拦截器的前置处理返回值 false,则开始倒序触发所有的
                //拦截器已完成处理
                if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                    return;
                }
                //真正地调用 handler 方法,也就执行对应的方法,并返回视图
                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
                //异步
                if (asyncManager.isConcurrentHandlingStarted()) {
                    return;
                }
                //无视图的情况下设置默认视图名称
                applyDefaultViewName(processedRequest, mv);
                //后置处理拦截器
                mappedHandler.applyPostHandle(processedRequest, response, mv);
            }
            //处理正常和异常的请求调用结果
            processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
        }
    }
```

DispatcherServlet 的 getHandler 方法是 Spring MVC 框架中的一个关键方法,它用于获取处理请求的处理器(Handler)。在这种方法中,DispatcherServlet 首先会从已注册的处理器映射器(HandlerMapper)中获取处理当前请求的处理器对象。处理器映射器会根据请求的 URL、请求方法等信息,匹配到相应的处理器。如果找到了匹配的处理器,则 getHandler 方法将返回该处理器对象;如果未找到匹配的处理器,则返回 null,表示没有找

到适合处理该请求的处理器。这种方法是 Spring MVC 请求处理流程中的关键步骤之一，通过它，DispatcherServlet 能够根据请求信息选择合适的处理器进行处理，源码如图 3-33 所示。

```
@Nullable
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    if (this.handlerMappings != null) {
        for (HandlerMapping mapping : this.handlerMappings) {
            HandlerExecutionChain handler = mapping.getHandler(request);
            if (handler != null) {
                return handler;
            }
        }
    }
    return null;
}
```

图 3-33　getHandler 源码

DispatcherServlet 的 getHandlerAdapter 方法是 Spring MVC 框架中的重要方法之一，它用于获取适配当前请求的处理器适配器（HandlerAdapter）。在这种方法中，DispatcherServlet 首先会从已注册的处理器适配器列表中选择适合当前请求处理器的适配器。处理器适配器负责调用处理器并执行请求处理逻辑，它会根据处理器的类型和执行方式来选择最合适的适配器。如果找到了匹配的适配器，则 getHandlerAdapter 方法将返回该适配器对象；如果未找到匹配的适配器，则返回 null。通过这种方法，DispatcherServlet 能够动态地选择合适的适配器来执行请求处理，实现了处理器和处理器适配器的解耦和灵活配置，源码如图 3-34 所示。

```
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
    if (this.handlerAdapters != null) {
        for (HandlerAdapter adapter : this.handlerAdapters) {
            if (adapter.supports(handler)) {
                return adapter;
            }
        }
    }
    throw new ServletException("No adapter for handler [" + handler +
            "]: The DispatcherServlet configuration needs to include a HandlerAdapter that supports
}
```

图 3-34　getHandlerAdapter 源码

3.8.4　processRequest

本节将详细地讲解核心代码的处理过程。在处理 HTTP 请求时，大多数请求会通过 processRequest 方法进行处理，这是 Spring MVC 框架中的一个核心方法。

在 processRequest 方法中，DispatcherServlet 首先会调用 getHandler 方法来获取与当前请求匹配的处理器（Handler）。一旦找到处理器，DispatcherServlet 会接着调用 getHandlerAdapter 方法以获取与处理器兼容的适配器。随后，使用此适配器来执行处理器中的请求处理逻辑。

在处理器处理请求的过程中，可能涉及一系列操作，如调用服务层、数据访问层等。最终，处理器会将处理结果封装成一个 ModelAndView 对象。

之后，DispatcherServlet 会根据 ViewResolver（视图解析器）来解析 ModelAndView 中的视图名称，并将模型数据传递给相应的视图以进行渲染。

最后，DispatcherServlet 会将渲染后的视图作为响应发送给客户端，从而完成整个请求处理流程。

这种方法不仅是 Spring MVC 框架中请求处理的核心流程，而且通过它，DispatcherServlet 能够智能地将请求分发给合适的处理器，并将处理结果呈现给用户。它实现了 MVC 模式中的控制器（Controller）角色，负责接收用户请求并返回响应，是整个框架的关键组成部分。对于想要深入了解其实现细节的读者，可以自行阅读框架的详细源码，代码如下：

```
//第 3 章 FrameworkServlet.java
protected final void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    //记录当前时间,用于计算处理请求花费的时间
    long startTime = System.currentTimeMillis();
    //记录异常,用于保存处理请求过程中发送的异常
    Throwable failureCause = null;

    //获取上下文信息,进行本地化和国际化处理
    LocaleContext previousLocaleContext = LocaleContextHolder.getLocaleContext();
    LocaleContext localeContext = buildLocaleContext(request);

    //获取当前线程的 Request 参数信息,构建 ServletRequestAttributes
    RequestAttributes previousAttributes = RequestContextHolder.getRequestAttributes();
    ServletRequestAttributes requestAttributes = buildRequestAttributes(request, response,
previousAttributes);

    //注册拦截器
    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    asyncManager.registerCallableInterceptor(FrameworkServlet.class.getName(), new
RequestBindingInterceptor());

    //将 localeContext 和 requestAttributes 放入当前线程中
    initContextHolders(request, localeContext, requestAttributes);

    try {
        //执行真正的逻辑,用户自己实现的逻辑
        doService(request, response);
    }
catch (ServletException | IOException ex) {
    //记录抛出的异常
```

```
            failureCause = ex;
            throw ex;
        }

        finally {
            //如果日志级别为 Debug,则打印请求日志
            logResult(request, response, failureCause, asyncManager);
            //发布 ServletRequestHandledEvent 请求处理完成事件
            publishRequestHandledEvent(request, response, startTime, failureCause);
        }
    }
```