1. 为啥要学 SpringMVC?

1.1 SpringMVC 简介

在学习 SpringMVC 之前我们先看看在使用 Servlet 的时候我们是如何处理用户请求的:

配置web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
version="4.0">
```

```
<!--将 Servlet接口实现类交给Tomcat管理-->
<servlet>
    <servlet-name>userServlet</servlet-name> <!--Servlet接口实现类名称-->
    <servlet-class>com.xxl.controller.UserServlet</servlet-class><!--声明servlet接口实现类类路径-->
</servlet>
<servlet-mapping>
    <servlet-name>userServlet</servlet-name>
    <url-pattern>/user</url-pattern>  <!--设置Servlet类的请求路径，必须以 / 开头-->
</servlet-mapping>

```

</web-app>

复制代码

继承 HttpServlet，实现 doGet 和 doPost 方法

```
public class UserServlet extends HttpServlet {
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
// 这里用来处理用户的 get 请求
System.out.println("哈哈哈哈哈哈我头上有注解");
}
```

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    // 这里用来处理用户的 post 请求
    System.out.println("接收到用户的 post 请求");
}

```

}
复制代码
获取请求参数
String name = request.getParameter("name");
String age = request.getParameter("age");
....
复制代码
一顿操作下来，我们发现用 Servlet 处理用户的请求也太麻烦了吧。
每个 Servlet 都要继承 HttpServlet、重写两个方法，我们需要写一堆 getParameter() 方法来获取请求参数，而且还要做数据类型的转换。
那有没有一个别人封装好的工具或者是框架让我少写这些重复性的代码呢?
SpringMVC闪亮登场。
SpringMVC 是一种轻量级的、基于 MVC 的 Web 层应用框架，它属于 Spring 框架的一部分。
SpringMVC 说白了就是对 Servlet 进行了封装，方便大家使用。
1.2 SpringMVC 优点
天生与 Spring 集成
支持 Restful 风格开发
便于与其他视图技术集成，例如 theamleaf、freemarker等
强大的异常处理
对静态资源的支持
总之就是好用!
2. HelloWorld
这里我们先来开发一个基于 SpringMVC 的程序，感受一下 SpringMVC 的迷人特性。
开发工具：IDEA
构建工具：Maven
2.1 新建基于 Maven 的 web 项目

2.2 加入依赖

```
<!-- servlet -->
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>javax.servlet-api</artifactId>
<version>4.0.1</version>
</dependency>
<!--springmvc -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>5.2.8.RELEASE</version>
</dependency>
<!-- junit -->
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.11</version>
<scope>test</scope>
</dependency>
```

复制代码

2.3 创建中央调度器

DispatcherServlet 是 SpringMVC 的中央调度器，它主要负责加载 SpringMVC 的配置。

从它的名字来看，他也属于一个 Servlet，遵守 Servlet 规范。所以我们需要在 web.xml 中创建 DispatcherServlet。

web.xml

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >
<web-app>
<!--servlet: DispatcherServlet-->
<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>
复制代码
2.4 创建 SpringMVC 的配置文件
这里我们在 src/resources 资源目录下创建 SpringMVC的配置文件 springmvc.xml，该文件名字可以任意命名。
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"
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
</beans>
复制代码
2.5 创建处理请求的处理器
TestController:
/**

- @Desc: 处理器
- @Author: 知否技术
- @date: 下午7:39 2022/4/29

*/
@Controller
public class TestController {
@RequestMapping("/hello")
public ModelAndView sayHello() {
ModelAndView mv = new ModelAndView();
mv.addObject("msg", "你好啊，李银河，我是王小波。");
mv.setViewName("/hello.jsp");
return mv;
}
}

复制代码
2.6 声明组件扫描器
我们在 springmvc.xml 中注册组件扫描器，
<!-- 扫描组件，将加上@Controller注解的类作为SpringMVC的控制层 -->
<context:component-scan base-package="com.zhifou"></context:component-scan>
复制代码
2.7 创建 jsp 页面
2.8 配置视图解析器
我们需要在 springmvc.xml 中配置请求文件的路径和文件后缀。
<!--
配置视图解析器
作用：将prefix + 视图名称 + suffix 确定最终要跳转的页面
-->
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="prefix" value="/WEB-INF/view/"></property>
<property name="suffix" value=".jsp"></property>
</bean>
复制代码
2.9 修改处理器请求文件路径
因为我们指定了请求文件的后缀是 .jsp，所以这里可以省略。
2.10 配置 tomcat，启动项目测试
3. 请求
3.1 @RequestMapping
@RequestMapping 注解用来指定处理哪些 URL 请求。
3.1.1 注解常用属性
value
value 用来表示请求的 url，可以省略不写
@RequestMapping(value = "/hello")
复制代码
简写
@RequestMapping("/hello")
复制代码
method
method 用来表示请求方式，不写的话默认是 GET 请求。常用的请求方式：
POST、GET、PUT、DELETE
复制代码

如果使用 method 属性，不能省略 value 属性。

```
@RequestMapping(value = "/hello",method = RequestMethod.GET)
public ModelAndView sayHello() {
ModelAndView mv = new ModelAndView();
mv.addObject("message", "你好啊，李银河，我是王小波。");
mv.setViewName("/hello");
return mv;
}
```

复制代码

3.1.2 标记位置

标记在类上面

一个系统包含很多模块，例如用户、商品、订单等模块。

我们需要为不同的模块定义不同的类，然后在类上面添加 @RequestMapping 注解，表示这个模块下面统一的请求路径：例如：

```
// 用户操作控制器
@Controller
@RequestMapping("/user")
public class UserController {
}
// 订单操作控制器
@Controller
@RequestMapping("/order")
public class OrderController {
}
...
```

复制代码

标记在方法上面

每个模块都有很多方法，例如增删改查等。所以我们一般会在相应的方法上面添加 @RequestMapping 注解，表示请求这个模块下的某个方法，例如：

```
@Controller
@RequestMapping("/user")
public class UserController {
@RequestMapping("/list")
public Object list() {
return null;
}
@RequestMapping(value = "/add",method = RequestMethod.POST)
```

```
public Object add() {
return null;
}
@RequestMapping(value = "/update",method = RequestMethod.POST)
public Object update() {
return null;
}
@RequestMapping(value = "/delete",method = RequestMethod.DELETE)
public Object delete() {
return null;
}
}
```

复制代码

所以当我们获取用户列表信息时，我们请求的后台接口的 url 就是：

```
ip地址:端口号/项目名/uset/list
//例如：
localhost:8080/ems/user/list
```

复制代码

3.1.3 @RequestMapping 的缩写注解

方法上关于不同请求方式的注解都比较长，例如:

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

复制代码

SpringMVC 为我们提供了简化写法：

GET请求：

```
@GetMapping("/list")
```

复制代码

POST 请求：

```
@PostMapping("/login")
```

复制代码

DELETE 请求：

```
@DeleteMapping("/delete/{id}")
```

复制代码

PUT 请求

```
@PutMapping("/update")
```

复制代码

3.2 接收请求参数

3.2.1 接收多个参数

```
@PostMapping("/login")
public Result<User> login(String username,String password) {
User user = userService.login(username, password);
return Result.success(user);
}
```
复制代码

3.2.2 实体类作为参数

Spring MVC 会按请求参数名和 POJO 属性名进行自动匹配，自动为该对象填充属性值。

```
@PostMapping("/login")
public Result<User> login(User user){
User user = userService.login(user.getUsername(), user.getPassword);
return Result.success(user);
}
```
复制代码

3.2.3 @RequestParam 注解

使用 @RequestParam 可以把请求参数传递给请求方法。

属性：

value：参数名

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

defaultValue: 默认值，当没有传递参数时使用该值

```
@PostMapping("/login")
public Result<User> login(@RequestParam String username, @RequestParam String password) {
User user = userService.login(username, password);
return Result.success(user);
}
```
复制代码

3.2.4 @PathVariable 注解

通过 @PathVariable 可以将 URL 中占位符参数绑定到控制器处理方法的入参中。

URL 中的 {xxx} 占位符可以通过 @PathVariable("xxx")绑定到操作方法的入参中。

```
@DeleteMapping(value = "/delete/{id}")
public Result<?> delete(@PathVariable("id") int id) {
userService.remove(id);
return Result.success();
}
```
复制代码

3.3 解决中文乱码

请求参数如果含有中文，会出现中文乱码问题。我们可以通过在 web.xml 中配置字符过滤器来解决中文乱码问题。

```
<filter>
<filter-name>encodingFilter</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>encodingFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

复制代码

注：filter 标签要在 servlet 标签之上

4. 响应

SpringMVC 提供了以下几种方式处理响应数据。

4.1 返回 String

SpringMVC 的视图解析器将返回的字符串转换为视图（页面）。

```
@GetMapping("/hello")
public String toHello() {
return "/hello";
}
```

复制代码

重定向：

使用 redirect 关键字可以重定向到其他页面。

```
@GetMapping("/hello")
public String toHello() {
return "redirect:/hello";
}
```

复制代码

4.2 返回 ModelAndView

控制器在处理完用户的请求之后，如果既想跳转页面，又想传递数据，可以使用 ModelAndView 对象。

```
@RequestMapping(value = "/hello",method = RequestMethod.GET)
public ModelAndView sayHello() {
ModelAndView mv = new ModelAndView();
mv.addObject("message", "你好啊，李银河，我是王小波。");
mv.setViewName("/hello");
return mv;
}
```

复制代码

4.3 返回 Model

Mode 对象也可以在跳转页面的时候传递数据。

```
@GetMapping("/print")
public String print(Model model) {
model.addAttribute("msg", "祝大家五一快乐！");
return "/hello";
}
```

复制代码

4.4 返回 json

控制器的方法也可以返回 Object 对象，但返回的对象不是作为视图出现的，而是作为页面上显示的数据。

返回对象，需要使用 @ResponseBody 注解将对象转换为 json 格式的数据响应给浏览器。

```
@GetMapping(value = "/print", produces = "application/json;charset=utf8")
@ResponseBody
public Object print() {
User user = new User(11111L, "张无忌", 12);
String json = JSONUtil.toJsonStr(user);
return json;
}
```

复制代码

如果遇到中文乱码的问题，我们可以在 @GetMapping 注解里面设置 produces 属性。

@RestController 注解

@RestController 注解是 @Controller 和 @ResponseBody 注解的组合注解，表示这个控制器中所有的方法全都返回 json 格式的数据。

4.5 返回 void

有时候我们不需要跳转页面，也不需要转发数据，我们只是希望浏览器能够渲染数据(例如后台向前台输出验证码)。这时候可以使用 HttpServletResponse 向浏览器输出数据。

```
@GetMapping("/print")
public void print(HttpServletResponse response) throws IOException {
/**解决响应数据中文乱码问题**/
response.setContentType("text/html;charset=utf8");
PrintWriter writer = response.getWriter();
writer.print("祝大家五一快乐");
writer.flush();
writer.close();
}
```
复制代码

5. 访问静态资源

我们在 web.xml 中配置 url-pattern 为 /，当我们请求项目的静态资源的时候，SpringMVC 的中央调度器会先根据处理器映射器寻找相应的处理器，结果没找到，所以请求静态资源会报 404。

我们可以使用 mvc:resources 标签来解决无法访问静态资源的问题。

但是 DispatcherServlet 的映射路径为 /，而 Tomcat 默认的 Servlet 的映射路径也为/，所以 DispatcherServlet 会覆盖 Tomcat 中默认的 Servlet，去处理除 jsp 之外的所有资源，导致静态资源无法被访问。

所以这里还需要结合另外一个标签使用：

```
<!-- static/**表示该目录下所有文件--><mvc:resources mapping="/static/**" location="/static/"/>
mvc:annotation-driven/
```
复制代码

这里我们测试访问项目中的照片：

```
@GetMapping("/hello")
public String toHello() {
return "/hello";
}
```
复制代码

6. 全局异常处理

在开发过程中我们会遇到各种各样的异常，我们会有两种处理方式：

try-catch 捕获

throws 抛出异常

但是用户不希望看到一堆异常信息：

即便是程序出错了，他们也希望看到一些看得懂的错误信息，例如：

```
{
code: 500,
message: '该商品信息不存在，请联系管理员'
}
```

复制代码
所以我们需要搞一个全局异常处理器先捕获这些错误信息，然后返回给用户统一的结果信息。
创建全局异常处理器
@RestControllerAdvice
public class GlobalExceptionHandler {
//自定义异常处理器
@ExceptionHandler(CustomException.class)
public Result CustomExceptionHandler(Exception e) {

```
    return Result.fail(500,e.getMessage());
}

```

}
复制代码
@RestControllerAdvice：表示这个类是全局异常处理器，返回的格式是 json 格式。
@ExceptionHandler：捕获的异常类型
创建自定义异常
/**

- @Desc: 自定义异常
- @Author: 知否技术
- @date: 下午8:05 2022/5/3

*/
public class CustomException extends RuntimeException{
public CustomException(String message) {
super(message);
}
}
复制代码
因为大部分异常是运行时异常，所以这里自定义的异常继承了运行时异常。
封装返回的统一结果类
/**

- @Desc: 统一结果类

- @Author: 知否技术

- @date: 下午9:02 2022/5/3

*/

```
public class Result implements Serializable {
private int code;
private String message;
private Object data;
private Result(String message) {
this.code = 200;
this.message = message;
}
private Result(String message, Object data) {
this.code = 200;
this.message = message;
this.data = data;
}
private Result(int code, String message) {
this.code = code;
this.message = message;
}
public static Result success(String message) {
return new Result(message);
}
public static Result success(String message, Object data) {
return new Result(message, data);
}
public static Result fail(int code, String message) {
return new Result(code, message);
}
public int getCode() {
return code;
}
public void setCode(int code) {
this.code = code;
}
public String getMessage() {
```

return message;
}
public void setMessage(String message) {
this.message = message;
}
public Object getData() {
return data;
}
public void setData(Object data) {
this.data = data;
}
}
复制代码
测试
@GetMapping("/hello")
@ResponseBody
public Result toHello() {
throw new CustomException("卧槽！！出错了");
}
复制代码
1. 拦截器
SpringMVC 中的拦截器主要是用来拦截用户的请求，并进行相关的处理。
实现拦截器：
创建拦截器
/**

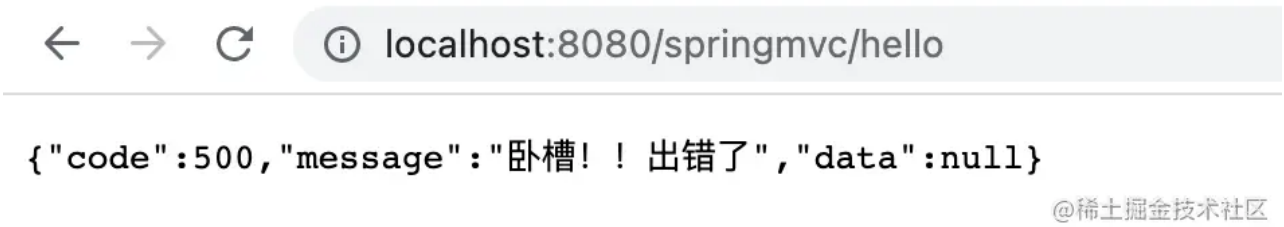
- @Desc:
- @Author: 知否技术
- @date: 下午2:09 2022/5/3

*/
public class MyInterceptor implements HandlerInterceptor {
// 在处理器中方法执行之前执行
@Override
public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

```
return false;

```

}
// 在处理器中方法执行之后执行
@Override
public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
ModelAndView modelAndView) throws Exception {
}
}
复制代码



7.配置拦截器
<!-- 配置拦截器 -->
mvc:interceptors
mvc:interceptor
<!--拦截所有请求-->
<mvc:mapping path="/**"/>
<!--排除请求-->
<mvc:exclude-mapping path="/user/login"/>
<bean class="com.zhifou.interceptor.MyInterceptor"></bean>
</mvc:interceptor>
</mvc:interceptors>
复制代码
测试：

```java
/**
 * @Desc:
 * @Author: 知否技术
 * @date: 下午2:09 2022/5/3
 */
public class MyInterceptor implements HandlerInterceptor {

    // 在处理器中方法执行之前执行
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) t
        /*这里处理用户的请求*/
        System.out.println("拦截器拦截用户请求。。。。。。。。");
        return true;
    }
    // 在处理器中方法执行之后执行
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, Mod

    }
}
```

@稀土掘金技术社区

```java
@GetMapping("/hello")
public String toHello() {
    System.out.println("控制器处理用户请求。。。。。");
    return "/hello";
}
```

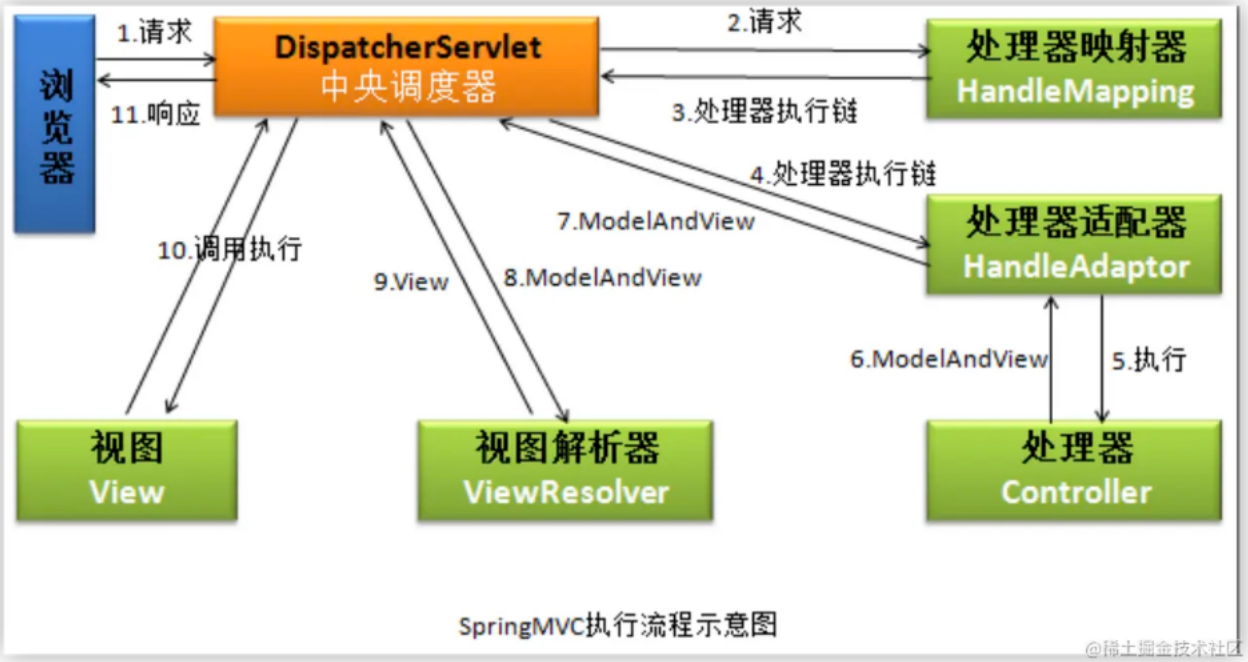
@稀土掘金技术社区

Output

```
拦截器拦截用户请求。。。。。。。。
控制器处理用户请求。。。。
```

@稀土掘金技术社区

1. SpringMVC 执行流程



1.Tomcat 服务器启动的时候会立即创建 DispatcherServlet(中央调度器)，同时会创建 SpringMVC 容器。

2.SpringMVC 容器初始化的时候会先根据配置文件中的组件扫描器先扫描一下哪些类上面有 @Controller 注解，并将这些类作为处理器类。

然后通过 @RequestMapping 注解生成对应的映射关系。这些对应关系由处理器映射器管理。

3.当收到用户的请求，中央调度器将请求转发给处理器映射器。

4.处理器映射器根据用户请求的 URL 从映射关系中找到处理该请求的处理器，然后封装成处理器执行链返回给中央处理器。

5.中央调度器根据处理器执行链中的处理器，找到能够执行该处理器的处理器适配器，处理器适配器调用执行处理器。

6.处理器将处理结果及要跳转的视图封装到 ModelAndView 中，并将其返回给处理器适配器。

7.处理器适配器直接将结果返回给中央调度器，中央调度器调用视图解析器，将 ModelAndView 中的视图名称封装为视图对象。

8.视图解析器将封装了的视图对象返回给中央调度器，中央调度器调用视图对象，填充数据，生成响应对象。

9.中央调度器将结果响应给浏览器。

————————————————

原文链接：https://blog.csdn.net/ma_xiao_qi/article/details/124689643