

# SpringMVC框架

---

## SpringMVC框架

### 一、SpringMVC简介

- 1、什么是MVC
- 2、什么是SpringMVC
- 3、SpringMVC的特点

### 二、HelloWorld

- 1、开发环境
- 2、创建Maven工程
  - ①添加Web模块
  - ②打包方式：war
  - ③引入依赖
- 3、配置web.xml
  - ①默认配置方式
  - ②扩展配置方式【推荐】
- 4、创建请求控制器
- 5、创建springMVC的配置文件
- 6、测试HelloWorld
  - ①通过对首页的访问
  - ②通过超链接跳转到指定页面
- 7、总结

### 三、@RequestMapping注解

- 1、@RequestMapping注解的功能
- 2、@RequestMapping注解的位置
- 3、@RequestMapping注解的value属性
- 4、@RequestMapping注解的method属性
- 5、@RequestMapping注解的params属性
- 6、@RequestMapping注解的headers属性
- 7、SpringMVC支持ant风格的路径
- 8、SpringMVC支持路径中的占位符（重点）

### 四、SpringMVC获取请求参数

- 1、通过ServletAPI获取
- 2、通过控制器方法的形参获取请求参数
- 3、@RequestParam
- 4、@RequestHeader
- 5、@CookieValue
- 6、通过POJO获取请求参数
- 7、解决获取请求参数的乱码问题

### 五、域对象共享数据

- 1、使用ServletAPI向request域对象共享数据
- 2、使用ModelAndView向request域对象共享数据
- 3、使用Model向request域对象共享数据
- 4、使用map向request域对象共享数据
- 5、使用ModelMap向request域对象共享数据
- 6、Model、ModelMap、Map的关系

- 7、向session域对象共享数据
- 8、向application域对象共享数据

## 六、SpringMVC的视图

- 1、ThymeleafView
- 2、转发视图
- 3、重定向视图
- 4、视图控制器view-controller

## 七、RESTful

- 1、RESTful简介
  - ①资源
  - ②资源的表述
  - ③状态转移
- 2、RESTful的实现
- 3、HiddenHttpMethodFilter

## 八、RESTful案例

- 1、准备工作
- 2、功能清单
- 3、具体功能：访问首页
  - ①创建view-controller
  - ②创建页面
- 4、具体功能：查询所有员工数据
  - ①控制器方法
  - ②创建employee\_list.html
- 5、具体功能：删除
  - ①创建处理delete请求方式的表单
  - ②删除超链接绑定事件点击
  - ③控制器方法
- 6、具体功能：跳转到添加数据页面
  - ①配置view-controller
  - ②创建employee\_add.html
- 7、具体功能：执行保存
  - ①控制器方法
- 8、具体功能：跳转到更新数据页面
  - ①修改超链接
  - ②控制器方法
  - ③创建employee\_update.html
- 9、具体功能：执行更新
  - ①控制器方法

## 九、HttpMessageConverter

- 1、@RequestBody
- 2、RequestEntity
- 3、@ResponseBody
- 4、SpringMVC处理json【@ResponseBody】
- 5、SpringMVC处理ajax
- 6、@RestController注解
- 7、ResponseEntity

## 十、文件上传和下载

- 1、文件下载
- 2、文件上传

## 十一、拦截器

- 1、拦截器的配置
- 2、拦截器的三个抽象方法
- 3、多个拦截器的执行顺序

## 十二、异常处理器

- 1、基于配置的异常处理
- 2、基于注解的异常处理

## 十三、注解配置SpringMVC

- 1、创建初始化类，代替web.xml
- 2、创建SpringConfig配置类，代替spring配置文件
- 3、创建WebConfig配置类，代替SpringMVC配置文件
- 4、测试功能

## 十四、SpringMVC执行流程

- 1、SpringMVC常用组件
- 2、DispatcherServlet初始化过程
  - ①初始化WebApplicationContext
  - ②创建WebApplicationContext
  - ③DispatcherServlet初始化策略
- 3、DispatcherServlet调用组件处理请求
  - ①processRequest()
  - ②doService()
  - ③doDispatch()
  - ④processDispatchResult()
- 4、SpringMVC的执行流程

### 附录：创建web工程标配

- 1、配置pom.xml
- 2、配置web.xml
- 3、配置SpringMVC.xml
- 4、注意

# 一、SpringMVC简介

---

## 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，指的是工程中的javaBean，作用的是处理数据

JavaBean分为两类

- 一类称为实体类Bean：专门存储业务数据的。
- 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，作用是接受请求和响应的数据

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

## 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，目前业界普遍选择了SpringMVC作为JavaEE项目表述层开发的**首选方案**

注：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

## 3、SpringMVC的特点

- **Spring家族原生产品**，与IOC容器等基础设施无缝衔接
- **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

# 二、HelloWorld

---

## 1、开发环境

IDEA, Maven, Tomcat

## 2、创建Maven工程

①添加Web模块

②打包方式：war

③引入依赖

```
<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>

    <!--spring5和Thymeleaf整合包-->
    <dependency>
        <groupId>org.thymeleaf</groupId>
        <artifactId>thymeleaf-spring5</artifactId>
        <version>3.0.12.RELEASE</version>
    </dependency>
</dependencies>

```

**注：**由于Maven的传递性，我们不必将所有需要的jar包全部配置到依赖，而是配置最顶端的依赖，其他靠传递性导入

### 3、配置web.xml

注册SpringMVC的前端控制器DispatcherServlet

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

```

<!--配置SpringMVC的前端控制器，对浏览器发送的请求统一进行处理-->
<servlet>
    <servlet-name>springMVC</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
</servlet>
<servlet-mapping>
    <servlet-name>springMVC</servlet-name>
    <!--设置springMVC的核心控制器所能处理的请求的请求路径/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-->
    <url-pattern>/</url-pattern>
</servlet-mapping>

```

## ②扩展配置方式【推荐】

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

```

<!-- 配置SpringMVC的前端控制器，对浏览器发送的请求统一进行处理 -->
<servlet>
    <servlet-name>springMVC</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>

    <!--通过初始化参数指定springMVC配置文件的位置和名称-->
    <init-param>
        <!-- contextConfigLocation为固定值 -->
        <param-name>contextConfigLocation</param-name>
        <!-- 使用classpath:表示从类路径查找配置文件，例如maven工程中的
src/main/resources -->
        <param-value>classpath:springMVC.xml</param-value>
    </init-param>

    <!--作为框架的核心组件，在启动过程中有大量的初始化操作要做，而这些操作放在第一次请求时才会执行会严重影响访问速度，因此需要通过此标签将启动控制
DispatcherServlet的初始化时间提前到服务器启动时间-->
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>springMVC</servlet-name>
    <!--设置springMVC的核心控制器所能处理的请求的请求路径/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-->
    <url-pattern>/</url-pattern>
</servlet-mapping>

```

注：

<url-pattern>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

## 4、创建请求控制器

由于前端控制器对浏览器发送请求进行了统一管理，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的控制器有一个**POJO（普通的java类）**担任，因此需要通过**@Controller**注解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

```
//普通的java类
@Controller
public class HelloController{

}
```

## 5、创建springMVC的配置文件

```
<beans xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
</beans>
<!--自动扫描包-->
<context:component-scan base-package='com.atgigu.mvc.controller'/>

<!--配置Thymeleaf视图解析器-->
<bean id="viewResolver"
class="org.thymeleaf.spring5.view.ThymeleafViewResolver">
<property name="order" value="1"/>
<property name="characterEncoding" value="UTF-8"/>
<property name="templateEngine">
<bean class="org.thymeleaf.spring5.SpringTemplateEngine">
<property name="templateResolver">
<bean
class="org.thymeleaf.spring5.templatereolver.SpringResourceTemplateRe
solver">
```

```

        <!-- 视图前缀 -->
        <property name="prefix" value="/WEB-INF/templates/" />

        <!-- 视图后缀 -->
        <property name="suffix" value=".html" />
        <property name="templateMode" value="HTML5" />
        <property name="characterEncoding" value="UTF-8"
    />

    </bean>
</property>
</bean>
</property>
</bean>

<!--
处理静态资源，例如html、js、css、jpg
若只设置该标签，则只能访问静态资源，其他请求则无法访问
此时必须设置<mvc:annotation-driven/>解决问题
-->
<mvc:default-servlet-handler/>

<!-- 开启mvc注解驱动 -->
<mvc:annotation-driven>
    <mvc:message-converters>
        <!-- 处理响应中文内容乱码 -->
        <bean

            class="org.springframework.http.converter.StringHttpMessageConverter"
        >
            <property name="defaultCharset" value="UTF-8" />
            <property name="supportedMediaTypes">
                <list>
                    <value>text/html</value>
                    <value>application/json</value>
                </list>
            </property>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>

```



## 6、测试HelloWorld

### ①通过对首页的访问

在请求控制器中创建处理请求的方法

```
// @RequestMapping注解：处理请求和控制器方法之间的映射关系
// @RequestMapping注解的value属性可以通过请求地址匹配请求，/表示的当前工程的上下文路径
// localhost:8080/springMVC/
@RequestMapping("/")
public String index() {
    //设置视图名称
    return "index";
}
```

### ②通过超链接跳转到指定页面

在主页index.html中设置超链接

```
<!DOCTYPE html>
<!--配置命名空间-->
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>首页</title>
</head>
<body>
    <h1>首页</h1>
    <a th:href="@{/hello}">Hello world</a><br/>
</body>
</html>
```

在请求控制器中创建请求的方法

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

## 7、总结

浏览器发送请求，若请求地址符合控制器的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**【启动服务器访问为localhost:8080/项目名称，**也就是value值为"/"**】，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的页面。

## 三、@RequestMapping注解

### 1、@RequestMapping注解的功能

从注解名称上我们可以看出，@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接收到指定的请求之后，就会找到映射关系中对应的控制器方法来处理请求。**

### 2、@RequestMapping注解的位置

@RequestMapping表示一个类：设置映射请求的请求路径的**初始信息**

@RequestMapping表示一个方法：设置映射请求的请求路径的**具体信息**

也就是请求路径为：类+方法的路径组合

```
@Controller
@RequestMapping("/test")
public class RequestMappingController{
    //此时请求映射所映射的请求的请求路径为：/test/testRequestMapping
    @RequestMapping("/testRequestMapping")
    public String testRequestMapping(){
        return "success";
    }
}
```

### 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

```
<a th:href="@{/testRequestMapping}">测试1</a>
<a th:href="@{/test}">测试2</a>
```

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

### 4、@RequestMapping注解的method属性

没有设置的时候任何的请求方式都可以请求

@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

```
<a th:href="@{/test}">测试@RequestMapping的value属性-->/test</a>
<form th:action="@{/test}" method="post">
    <input type='submit'>
</form>
```

```
@RequestMapping(value={"/testRequestMapping","/test"},method=
{RequestMethod.GET,RequestMethod.POST})
public String testRequestMapping(){
    return "success";
}
```

注：

- 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【RequestMethod.GET】， post【RequestMethod.POST】， put， delete

但是目前浏览器只支持get和post，若在form表单提交时，为method设置了其他请求方式的字符串（put或delete），则按照默认的请求方式get处理 若要发送put和delete请求，则需要通过spring提供的过滤器HiddenHttpMethodFilter，在RESTful部分会讲到

## 5、@RequestMapping注解的params属性

必须同时满足params属性才得以匹配

@RequestMapping注解的params属性通过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**

```
<a th:href="@{/test(username='admin',password=123456)}">测试  
@RequestMapping的params属性-->/test</a><br>
```

```
@RequestMapping(  
    value={"/testRequestMapping", "/test"},  
    method={RequestMethod.POST, RequestMethod.GET},  
    params={"username", "password!=123456"}  
)  
public String testRequestMapping(){  
    return "success";  
}
```

注:

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!=123456" not met for actual request parameters: username={admin}, password={123456}

## 6、@RequestMapping注解的headers属性

**必须同时满足headers的属性才得以匹配，匹配请求头信息**

@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

## 7、SpringMVC支持ant风格的路径

模糊匹配的功能

?: 表示任意的单个字符

\*: 表示任意的0个或多个字符

\*\*: 表示任意的一层或多层目录

注意：在使用\*\*时，只能用 /\*\*/XXX的方式

## 8、SpringMVC支持路径中的占位符（重点）

原始方式：/deleteUser?id=1

**rest方式：/deleteUser/1**

SpringMVC路径中的占位符常用于RESTful风格，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，就可以相应的@RequestMapping注解的value属性中通过占位符{XXX}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的形参

```
<a th:href="@{/testRest/1/admin}">测试路径中的占位符</a>
```

```
@RequestMapping("/testRest/{id}/{username}")
public String testRest(@PathVariable("id") String
id,@PathVariable("username") String username){
    System.out.println("id:"+id+",username:"+username);
    return "success";
}
//最终输出结果--> id:1,username:admin
```

## 四、SpringMVC获取请求参数

### 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

```
@RequestMapping("/testParam")
//形参位置的request表示当前请求，参数没有name的不能用这个方式
public String testParam(HttpServletRequest request){
    String username = request.getParameter("username");
    return username;
}
```

### 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就会将请求参数赋值给相应的形参

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

```
@RequestMapping("/testParam")
public String testParam(String username,String password){
    System.out.println("username:"+username+",password:"+password);
    return "success";
}
```

注：

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，同名的结果用逗号隔开；也可以接受字符串数组

```

@RequestMapping("/testParam")
public String testParam(String hobby){
    System.out.println(hobby);
    //输出hobby:a,b,c
}

@RequestMapping("/testParam")
public String testParam(String[] hobby){
    System.out.println(hobby);
    //输出hobby:[a,b,c]
}

```

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

### 3、@RequestParam

@RequestParam**是将请求参数和控制器方法的形参创建映射关系，也就是要拿哪一个请求参数去对应形参**

@RequestParam注解一共有三个属性

1. value：指定形参赋值的请求参数的参数名【name的别名】
2. required：设置是否必须传输此参数，**默认为true**
  1. 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;
  2. 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**
3. 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""（文本框什么都不输入）时，则使用默认值为形参赋值（这个参数值就是默认值）**

```

public String testParam(@RequestParam("user_name") String username){
    //那么可以去匹配名为user_name的请求参数
}

```

## 4、@RequestHeader

通过形参获取当前请求头信息必须使用这个注解

@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

## 5、@CookieValue

通过形参获取Cookie信息必须使用这个注解

@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

## 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，那么请求参数就会以此为**属性赋值**。

```
<form th:action="@{/testpojo}" method="post">
    用户名: <input type="text" name="username">
    密码: <input type="password" name="password">
    性别: <input type="radio" name="sex" value="男">男<input
type="radio" name="sex" value="女">女
    年龄: <input type="text" name="age">
    邮箱: <input type="text" name="email">
    <input type="submit">
</form>
```

```
@RequestMapping("/testpojo")
public String testProjo(User user){
    System.out.println(user);
    return "success"
}
```

```
//最终结果-->User{id=null, username='张三',password='123',age=23,sex='男',email='123@qq.com'}
```



## 7、解决获取请求参数的乱码问题

web.xml注册顺序，监听器Listen、过滤器Filter、Servlet

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

```
<!--配置SpringMVC编码过滤器-->
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-
class>

    <init-param>
        <!--这个参数可以为空，也就是可以不设置-->
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
        <!--强迫编码-->
        <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>
```

注：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

## 五、域对象共享数据

### 1、使用ServletAPI向request域对象共享数据

```
@RequestMapping("/testServletAPI")
public String testServletAPI(HttpServletRequest request){
    request.setAttribute("testScope","hello,servletAPI");
    return "success";
}
```

前端接受域对象，下同

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Title</title>
  </head>
  <body>
    <!--使用th:text获取域对象数据；如果是request域对象直接写键名，如果是
    session域对象写session.键，如果是ServletContext域对象使用application.键-->
    <p th:text="${testRequestScope}"></p>
  </body>
</html>

```

## 2、使用ModelAndView向request域对象共享数据

不管用什么方式进行域对象共享数据，最终都封装到ModelAndView中

```

@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("success");
    return mav;
}

```

## 3、使用Model向request域对象共享数据

```

@RequestMapping("/testModel")
public String testModel(Model model){
    model.addAttribute("testScope","hello,model");
    return "success";
}

```

## 4、使用map向request域对象共享数据

```
@RequestMapping("/testMap")
public String testMap(Map<String,Object> map){
    map.put("testScope","hello,map");
    return "success";
}
```

## 5、使用ModelMap向request域对象共享数据

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

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

## 7、向session域对象共享数据

```
@RequestMapping("/testSession")
public String testSession(HttpSession session){
    session.setAttribute("testSessionScope","hello,session");
    return "success";
}
```

## 8、向application域对象共享数据

```
@RequestMapping("/testApplication")
public String testApplication(HttpSession session){
    ServletContext application = session.getServletContext();
    application.setAttribute("testApplicationScope",
    "hello,application");
    return "success";
}
```

## 六、SpringMVC的视图

SpringMVC中的视图是View接口，视图的作用是渲染，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InternalResourceView和重定向视图RedirectView

**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事ThymeleafView

### 1、ThymeleafView

当控制器方法中所设置的视图名称**没有任何前缀**时，此时视图名称会被SpringMVC配置文件的视图解析器解析，**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**

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

### 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：

1. **当控制器方法所设置的视图名称前缀为 "forward:"** 创建转发视图
2. 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方式实现跳转
3. **去掉前缀之后剩下的路径找到控制器，再创建一次视图，转发到success**
4. 也即是**先转发到控制器方法，再由控制器方法跳转到对应的页面**
5. 例如 "forward:/" , "forward:/employee"

```
@RequestMapping("/testForward")
public String testForward(){
    return "forward:/testHello";
}
```

### 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:/employee"

```
@RequestMapping("/testRedirect")
public String testRedirect(){
    return "redirect:/testHello";
}
```

注：

重定向视图在解析时，会先将redirect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

### 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标签来表示

```
<!--
    path: 设置处理的请求地址
    view-name: 设置请求地址所对应的视图名称
-->
<mvc:view-controller path="/testview" view-name="success"></mvc:view-controller>
```

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注解驱动的标志：

<mvc:annotation-driven/>

## 七、RESTful

# 1、RESTful简介

**REST: Representational State Transfer，表现层资源状态转移。**

## 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。

## 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。

## ③状态转移

状态转移说的是：在客户端和服务端之间转移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。

# 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杆分开，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
保存操作	saveUser	user -->post请求
删除操作	deleteUser?id=1	user/1 -->delete请求
更新操作	updateUser	user -->请求

### 3、HiddenHttpMethodFilter

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

SpringMVC提供了HiddenHttpMethodFilter将POST请求转换为DELETE和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过滤器将会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` 的值才是最终请求方式**

在web.xml中注册HiddenHttpMethodFilter

```
<filter>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <filter-
class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-
class>
</filter>
<filter-mapping>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

目前为止，SpringMVC中提供两个过滤器：CharacterEncodingFilter和HiddenHttpMethodFilter

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

原因：

1. 在 CharacterEncodingFilter 中通过 `request.setCharacterEncoding(encoding)` 方法设置字符集的
2. `request.setCharacterEncoding(encoding)` 方法要求前面不能有任何获取请求参数的操作
3. 而 HiddenHttpMethodFilter 恰恰有一个获取请求方式的操作：

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

## 八、RESTful案例

## 1、准备工作

和传统的CRUD一样，实现对员工信息的增删改查

- 环境搭建
- 准备实体类【包括Integer id \ String lastName \ Integer gender \ String email】
- 准备dao模拟数据

```
@Repository
public class EmployeeDao{
    private static Map<Integer,Employee> employees = null;
    static{
        employees = new HashMap<Integer,Employee>();
        employees.put(1001, new Employee(1001, "E-AA",
"aa@163.com", 1));
        employees.put(1002, new Employee(1002, "E-BB",
"bb@163.com", 1));
        employees.put(1003, new Employee(1003, "E-CC",
"cc@163.com", 0));
        employees.put(1004, new Employee(1004, "E-DD",
"dd@163.com", 0));
        employees.put(1005, new Employee(1005, "E-EE",
"ee@163.com", 1));
    }
    private static Integer initId = 1006;
    public void save(Employee employee){
        if(employee.getId() == null){
            employee.setId(initId++);
        }
        employees.put(employee.getId(), 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);
    }
}
```



## 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/2	DELETE
跳转到添加数据页面	/toAdd	GET
执行保存	/employee	POST
跳转到更新数据页面	/employee/2	GET
执行更新	/employee	PUT

## 3、具体功能：访问首页

### ①创建view-controller

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

### ②创建页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8" >
    <title>Title</title>
  </head>
  <body>
    <h1>首页</h1>
    <a th:href="@{/employee}">访问员工信息</a>
  </body>
</html>
```

## 4、具体功能：查询所有员工数据

### ①控制器方法

```

@RequestMapping(value="/employee" ,method = RequestMethod.GET)
public String getEmployeeList(Model model){
    Collention<Employee> employeeList = employeeDao.getAll();
    model.addAttribute("employeeList",employeeList);
    return "employee_list";
}

```

## ②创建employee\_list.html

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
        <title>Employee Info</title>
        <script type="text/javascript" th:src="@{/static/js/vue.js}">
    </script>
    </head>
    <body>
        <table border="1" cellpadding="0" cellspacing="0" style="text-align:center;" id="dataTable">
            <tr>
                <th colspan="5">Employee Info</th>
            </tr>
            <tr>
                <th>id</th>
                <th>lastName</th>
                <th>email</th>
                <th>gender</th>
                <th>options(<a th:href="@{/toAdd}">add</a>)</th>
            </tr>
            <tr th:each="employee : ${employeeList}">
                <td th:text="${employee.id}"></td>
                <td th:text="${employee.lastName}"></td>
                <td th:text="${employee.email}"></td>
                <td th:text="${employee.gender}"></td>
                <td>
                    <a class="deleteA" @click="deleteEmployee"
th:href="@{'/employee/'+${employee.id}}">delete</a>
                    <a
th:href="@{'/employee/'+${employee.id}}">update</a>
                </td>
            </tr>
        </table>
    </body>
</html>

```

## 5、具体功能：删除

### ①创建处理delete请求方式的表单

```
<!--作用-->
<form id="delete_form" method="post">
    <!--HiddenHttpMethodFilter要求：必须传输 _method 请求参数，并且值作为最终的请求方式-->
    <input type="hidden" name="_method" value="delete">
</form>
```

### ②删除超链接绑定事件点击

引入vue.js

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

删除超链接

```
<a class="delete" @click="deleteEmployee"
th:href="@{'/employee'+${employee.id}}">delete</a>
```

通过vue处理点击事件

```
<script type="text/javascript">
    var vue = new Vue({
        el:"#dataTable",
        methods:{
            //event代表当前事件
            deleteEmployee:function(event){
                //通过id获取表单标签
                var delete_form =
document.getElementById("delete_form");
                //将触发事件的超链接的href属性作为表单的action属性值
                delete_form.action = event.target.href;
                //提交表单
                delete_form.submit();
                //阻止超链接的默认跳转行为
                event.preventDefault();
            }
        }
    })
</script>
```

### ③控制器方法

```
@RequestMapping(value="/employee/{id}",method=RequestMethod.DELETE)
public String deleteEmployee(@PathVariable("id") Integer id){
    employeeDao.delete(id);
    return "redirect:/employee";
}
```

## 6、具体功能：跳转到添加数据页面

### ①配置view-controller

```
<mvc:view-controller path="/toAdd" view-name="employee_add">
</mvc:view-controller>
```

### ②创建employee\_add.html

```
<form th:action="@{/employee}" method="post">
    lastName:<input type="text" name="lastName"><br>
    email:<input type="text" name="email"><br>
    gender:<input type="radio" name="gender" value="1">male
    <input type="radio" name="gender" value="0">female<br>
    <input type="submit" value="add"><br>
</form>
```

## 7、具体功能：执行保存

### ①控制器方法

```
@RequestMapping(value="/employee",method=RequestMethod.POST)
public String addEmployee(Employee employee){
    employeeDao.save(employee);
    return "redirect:/employee";
}
```

## 8、具体功能：跳转到更新数据页面

### ①修改超链接

```
<a th:href="@{'/employee/'+${employee.id}}"></a>
```

## ②控制器方法

```
@RequestMapping(value="/employee/{id}",method=RequestMethod.GET)
public String getEmployeeById(@PathVariable("id") Integer id ,Model
model){
    Employee employee = employeeDao.get(id);
    model.addAttribute("employee",employee);
    return "employee_update";
}
```

## ③创建employee\_update.html

```
<from th:action="@{/employee}" method="post">
    <input type="hidden" name="_method" value="put">
    <input type="hidden" name="id" value="${employee.id}">
    lastName:<input type="text" name="lastName"
th:value="${employee.lastName}"><br>
    email:<input type="text" name="email"
th:value="${employee.email}"><br>
    <!--
        th:field="${employee.gender}"可用于单选框或复选框的回显
        若单选框的value和employee.gender的值一致，则添加
checked="checked"属性
    -->
    gender:<input type="radio" name="gender" value="1"
th:field="${employee.gender}">male
    <input type="radio" name="gender" value="0"
th:field="${employee.gender}">female<br>
    <input type="submit" value="update"><br>
</from>
```

## 9、具体功能：执行更新

### ①控制器方法

```
@RequestMapping(value="/employee",method=RequestMethod.PUT)
public String updateEmployee(Employee employee){
    employeeDao.save(employee);
    return "redirect:/employee";
}
```

## 九、HttpMessageConverter

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

HttpMessageConverter提供了两个注解和两个类型：@RequestBody, @ResponseBody, RequestEntity, ResponseEntity

## 1、@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

```
<from th:action="@{/testRequestBody}" method="post">
  用户名: <input type="text" name="username"><br>
  密码: <input type="password" name="password"><br>
  <input type="submit">
</from>
```

```
@RequestMapping("/testRequestBody")
public String testRequestBody(@RequestBody String requestBody){
    System.out.println("requestBody:"+requestBody);
    return "success";
}
```

输出结果：requestBody:username=admin&password=123456

## 2、RequestEntity

RequestEntity封装请求报文的一种类型，需要在控制器方法的形参中设置该类型的形参，当前请求的请求报文就会赋值给该形参，可以通过getHeaders()获取请求头信息，通过getBody()获取请求体信息

```
@RequestMapping("/testRequestEntity")
public String testRequestEntity(RequestEntity<String> requestEntity){
    System.out.println("requestHeader:"+requestEntity.getHeaders());
    System.out.println("requestBody:"+requestEntity.getBody());
    return "success";
}
```

## 3、@ResponseBody

@ResponseBody标识一个控制器，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

```

@RequestMapping("/testResponseBody")
@ResponseBody
public String testResponseBody(){
    return "success";
}

```

结果：在浏览器页面显示success

## 4、SpringMVC处理json 【@ResponseBody】

**@ResponseBody处理json的步骤**

①导入jackson的依赖

```

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

```

②在SpringMVC的核心配置文件中开启mvc的注解驱动，此时在HandlerAdaptor中会自动装配一个消息转换器：MappingJackson2HttpMessageConverter，可以将响应到浏览器的java对象转换为json格式的字符串

```

<mvc:annotation-driven/>

```

③在处理方法上使用@ResponseBody注解进行标识

④将java对象直接作为控制器方法的返回值返回，将会自动转换为json格式的字符串

```

@RequestMapping("/testResopnseUser")
@ResponseBody
public User testResponseUser(){
    return new User(1001,"admin","123456",23,"男");
}

```

浏览器的页面中展示的结果

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

## 5、SpringMVC处理ajax

### ①请求超链接

```
<div id="app">
  <a th:href="@{/testAjax}" @click="testAjax">testAjax</a>
</div>
```

### ②通过vue和axios处理点击事件

```
<script type="text/javascript" th:src="@{/static/js/vue.js}"></script>
<script type="text/javascript" th:src="@{/static/js/axios.min.js}">
</script>
<script type="text/javascript">
  var vue = new Vue({
    el: "#app",
    methods: {
      testAjax: function(event) {
        axios({
          method: "post",
          url: event.target.href,
          params: {
            username: "admin",
            password: "123456"
          }
        }).then(function(response) {
          alter(response.data);
        });
        event.preventDefault();
      }
    }
  })
</script>
```

### ③控制器方法

```
@RequestMapping("/testAjax")
@ResponseBody
public String testAjax(String username, String password) {
  System.out.println("username:" + username + ", password:" + password);
  return "hello, ajax";
}
```



## 6、@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

## 7、ResponseEntity

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

# 十、文件上传和下载

文件上传与下载的底层原理均为**文件复制**，也就是浏览器和服务端之间文件的相互复制

### 1、文件下载

使用ResponseEntity实现下载文件的功能

```
@RequestMapping("/testDown")
public ResponseEntity<byte[]> testResponseEntity(HttpSession session)
throws IOException{
    //获取ServletContext对象
    ServletContext servletContext = session.getServletContext();
    //获取服务器中文件的真实路径
    String realPath = servletContext.getRealPath("/static/img/1.jpg");
    //创建输入流
    InputStream is = new FileInputStream(realPath);
    //将流读到字节数组中
    byte[] bytes = new byte[is.available()];
    is.read(bytes);
    //创建HttpHeaders对象设置响应头信息
    MultiValueMap<String, String> headers = new HttpHeaders();
    //设置要下载方式以及下载文件的名称，设置下载方式
    headers.add("Content-Disposition", "attachment;filename=1.jpg");
    //设置响应状态码
    HttpStatus statusCode = HttpStatus.OK;
    //创建ResponseEntity对象，参数分别为响应体、响应头、状态码
    ResponseEntity<byte[]> responseEntity = new ResponseEntity<>
(bytes, headers, statusCode);
    //关闭输入流
    is.close();
    return responseEntity;
}
```

## 2、文件上传

文件上传要求form表单的请求方式为post，并且添加属性  
enctry="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：

### ①添加依赖

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

### ②在SpringMVC的配置文件中添加配置

```
<!--必须通过文件解析器的解析才能将文件转换为MultipartFile对象-->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"></bean>
```

### ③控制器方法

```
@RequestMapping("/testUp")
public String testUp(MultipartFile photo,HttpMessgae session) throws
IOException{
    //获取上传的文件的文件名
    String fileName = photo.getOriginalFilename();
    //处理文件重名问题
    String bzName = filename.substring(filename.lastIndexOf(".");
    fileName = UUID.randomUUID().toString() + baName;
    //获取服务器中photo目录的路径
    ServletContext servletContext = session.getServletContext();
    String photoPath = servletContext.getRealPath("photo");
    File file = new File(photoPath);
    if(!file.exists()){
        file.mkdir();
    }
    String finalPath = photoPath + File.separator + fileName;
    //实现上传功能
    photo.transferTo(new File(finalPath));
}
```

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

## 十一、拦截器

### 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的配置文件中配置

```
<bean class="com.atguigu.interceptor.FirstInterceptor"></bean>
<ref bean="firstInterceptor"></ref>
<!-- 以上两种配置方式都是对DispatcherServlet所处理的所有的请求进行拦截 -->
<mvc:interceptor>
    <mvc:mapping path="/*"/>
    <mvc:exclude-mapping path="/testRequestEntity"/>
    <ref bean="firstInterceptor"></ref>
</mvc:interceptor>
<!--
    以上配置方式可以通过ref或bean标签设置拦截器，通过mvc:mapping设置需要拦截的
    请求，通过mvc:exclude-mapping设置需要排除的请求，即不需要拦截的请求
-->
```

### 2、拦截器的三个抽象方法

SpringMVC中的拦截器有三个抽象方法：

1. 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心，返回true表示放心，即调用控制器方法；返回false表示拦截，表示不调用控制器方法
2. postHandle：控制器方法执行之后执行postHandle()
3. afterComplation：处理完视图和模型数据，渲染视图完毕之后执行afterComplation()

### 3、多个拦截器的执行顺序

1. 若每个拦截器的preHandle()的返回值都为true
  1. 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

2. preHandle()会按照配置的顺序执行，而postHandle()和afterComplation()会按照配置的反序执行
2. 若某个拦截器的preHandle()返回值为false
  1. preHandle()返回false和它之前的拦截器的preHandle()都会执行，postHandle()都不执行
  2. 返回false的拦截器之前的拦截器的afterComplation()会执行

## 十二、异常处理器

### 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：

#### HandlerExceptionHandlerResolver

HandlerExceptionHandlerResolver接口的实现类有：DefaultHandlerExceptionHandlerResolver和SimpleMappingExceptionHandlerResolver

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

```
<bean
class="org.springframework.web.servlet.handler.SimpleMappingExceptionR
esolver">
    <protety name="exceptionMappings">
        <props>
            <!--
                properties的键表示处理器方法执行过程中出现的异常
                properties的值表示若出现指定的异常，设置一个新的视图名称，跳
转到指                                定页面
            -->
            <prop key="java.lang.ArithmeticException">error</prop>
        </props>
    </protety>
    <!--
        exceptionAttribute属性设置一个属性名，将出现的异常信息在请求域中进行
共享
    -->
    <property name="exceptionAttribute" value="ex"></property>
</bean>
```

## 2、基于注解的异常处理

```
//@ControllerAdvice表示标识为异常处理的组件
@ControllerAdvice
public class ExceptionController{
    //@ExceptionHandler用于设置所标识方法处理的异常
    @ExceptionHandler(ArithmeticException.class)
    //@ex表示当前请求处理中出现的异常对象
    public String handleArithmeticException(Exception ex ,Model model)
    {
        model.addAttribute("ex",ex);
        return "error";
    }
}
```

## 十三、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

### 1、创建初始化类，代替web.xml

在Servlet3.0的环境中，容器会在类路径中查找实现java.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他们来完成。

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。

```
public class webInit extends
AbstractAnnotationConfigDispatcherServletInitializer{
    /**
     * 指定spring的配置类，根配置
     * @return
     */
    @Override
    protected Class<?>[] getRootConfigClasses(){
        return new Class[]{SpringConfig.class};
    }

    /**
     * 指定SpringMVC的配置类
     * @return
     */
}
```

```

@Override
protected Class<?>[] getServletConfigClasses(){
    return new Class[]{WebConfig.class};
}

/**
 * 指定DispatcherServlet的映射规则，即url-pattern
 * @return
 */
@Override
protected String[] getServletMappings(){
    return new String[]{"/"};
}

/**
 * 添加过滤器
 * @return
 */
@Override
protected Filter[] getServletFilters(){
    CharacterEncodingFilter encodingFilter = new
CharacterEncodingFilter();
    encodingFilter.setEncoding("UTF-8");
    encodingFilter.setForceRequestEncoding(true);
    HiddenHttpMethodFilter hiddenHttpMethodFilter = new
HiddenHttpMethodFilter();
    return new Filter[]{encodingFilter,hiddenHttpMethodFilter};
}
}

```

## 2、创建SpringConfig配置类，代替spring配置文件

```

@Configuration
public class SpringConfig{
    //SSM整合之后，spring的配置信息写在此处
}

```

## 3、创建WebConfig配置类，代替SpringMVC配置文件

①扫描组件 ②视图解析器 ③view-controller ④默认Servlet处理静态资源default-servlet-handler ⑤mvc注解驱动 ⑥文件上传解析器 ⑦异常处理 ⑧拦截器

```

@Configuration
//扫描组件
@ComponentScan("com.atguigu.mvc.controller")

```

```

//开启mvc注解驱动
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer{
    //使用默认的servlet处理静态资源
    @Override
    public void
configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer){
    configurer.enable();
}

    //配置文件上传解析器
    @Bean
    public CommonsMultipartResolver multipartResolver(){
        return new CommonsMultipartResolver();
    }

    //配置拦截器
    @Override
    public void addInterceptors(InterceptorRegistry registry){
        FirstInterceptor firstInterceptor = new FirstInterceptor();

        registry.addInterceptor(firstInterceptor).addPathPatterns("/**");
    }

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

    //配置异常映射
    @Override
    public void
configureHandlerExceptionResolvers(List<HandlerExceptionResolver>
resolvers){
        SimpleMappingExceptionHandler exceptionResolver = new
SimpleMappingExceptionHandler();
        Properties prop = new Properties();
        prop.setProperty("java.lang.ArithmeticException", "error");
        //设置异常映射
        exceptionResolver.setExceptionMappings(prop);
        //设置共享异常信息的键
        exceptionResolver.setExceptionHandler("ex");
        resolvers.add(exceptionResolver);
    }

    //配置生成模块解析器
    @Bean

```

```

    public ITemplateResolver templateResolver(){
        WebApplicationContext webApplicationContext =
ContextLoader.getCurrentWebApplicationContext();
        // ServletContextTemplateResolver需要一个ServletContext作为构造
参数，可通过WebApplicationContext 的方法获得
        ServletContextTemplateResolver templateResolver = new
ServletContextTemplateResolver(
webApplicationContext.getServletContext());
        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
    @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
    @Bean
    public ViewResolver viewResolver(SpringTemplateEngine
templateEngine){
        ThymeleafViewResolver viewResolver = new
ThymeleafViewResolver();
        viewResolver.setCharacterEncoding("UTF-8");
        viewResolver.setTemplateEngine(templateEngine);
        return viewResolver;
    }
}

```

## 4、测试功能

```

@RequestMapping("/")
public String index(){
    return "index";
}

```

# 十四、SpringMVC执行流程



## 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, 由它调用其他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【也就是Controller】
  - 在DispatcherServlet的控制下Handler对具体的用户请求进行处理
- **HandlerAdapter**: 处理适配器, 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**: 视图控制器, 不需要工程师开发, 由框架提供
  - 作用: 进行视图解析, 得到响应的视图例如: ThymeleafView、InternalResourceView、RedirectView
- **View**: 视图
  - 作用: 将模型数据通过页面展示给用户

## 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

### ①初始化WebApplicationContext

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

### ②创建WebApplicationContext

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

### ③DispatcherServlet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

## 3、DispatcherServlet调用组件处理请求

### ①processRequest()

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

## ②doService()

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

## ③doDispatch()

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

## ④processDispatchResult()

# 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

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

1. 如果没有配置，则控制台报映射找不到，客户端报404错误
2. 如果有配置，则访问目标资源（一般为静态资源，如：JS、CSS、HTML等），找不到客户端也会展示404错误

### 2. 存在则执行下面的流程

1. 根据该URI，调用HandlerMapping获得带Handler配置的所有相关的对象（包括Handler对象以及Handler对象对应的拦截器），最后以HandlerExecutionChain执行链对象的形式返回
2. DispatcherServlet 根据获得的Handler，选择一个合适的HandlerAdapter
3. 如果成功获得HandlerAdapter，此时将开始执行拦截器的preHandler(...)方法【正向】
4. 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
5. ) Handler执行完成后，向DispatcherServlet 返回一个 ModelAndView对象。
6. 此时将开始执行拦截器的postHandle(...)方法【逆向】。
7. 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。
8. 渲染视图完毕执行拦截器的afterCompletion(...)方法【逆向】。
9. 将渲染结果返回给客户端。

## 附录：创建web工程标配

### 1、配置pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.atguigu.mvc</groupId>
    <artifactId>Spring_mvc_rest</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <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>

    <!--spring5和Thymeleaf整合包-->
    <dependency>
        <groupId>org.thymeleaf</groupId>
        <artifactId>thymeleaf-spring5</artifactId>
        <version>3.0.12.RELEASE</version>
    </dependency>
</dependencies>
</project>

```

## 2、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">

    <!-- 配置过滤器-->
    <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>forceResponseEncoding</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PUT和DELETE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>

<!-- 配置SpringMVC前端控制器-->
    <servlet>
        <servlet-name>DispatcherServlet</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>DispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>

```

### 3、配置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: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/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-
package="com.atguigu.rest.controller"/>

```

```

<!-- 配置Thymeleaf视图解析器-->
<bean id="viewResolver"
      class="org.thymeleaf.spring5.view.ThymeleafViewResolver">
    <property name="order" value="1"/>
    <property name="characterEncoding" value="UTF-8"/>
    <property name="templateEngine">
        <bean class="org.thymeleaf.spring5.SpringTemplateEngine">
            <property name="templateResolver">
                <bean
class="org.thymeleaf.spring5.templateresolver.SpringResourceTemplateRe
solver">

                    <!-- 视图前缀 -->
                    <property name="prefix" value="/WEB-
INF/templates/" />

                    <!-- 视图后缀 -->
                    <property name="suffix" value=".html"/>
                    <property name="templateMode" value="HTML5"/>
                    <property name="characterEncoding" value="UTF-
8"/>

                </bean>
            </property>
        </bean>
    </property>
</bean>

<mvc:view-controller path="/" view-name="首页视图名称"></mvc:view-
controller>

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

<!-- 开放静态资源的访问-->
<mvc:default-servlet-handler/>
</beans>

```

## 4、注意

### 1. 资源找不到的情况下

1. 检查服务器有无（target目录下）——可以用maven进行重新打包
2. 开放对静态资源的访问或者写配置类实现  
configureDefaultServletHandling()

```
<mvc:default-servlet-handler/>
```

