springmvc-day02

第一章 RESTFul风格交互方式

第一节 RESTFul概述

1. REST的概念

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

- 定位：互联网软件架构风格
- 倡导者：Roy Thomas Fielding
- 文献：Roy Thomas Fielding的博士论文

2. REST要解决的问题: 将针对功能设计系统转变成针对资源设计系统

传统的软件系统仅在本地工作，但随着项目规模的扩大和复杂化，不但整个项目会拓展为分布式架构，很多功能也会通过网络访问第三方接口来实现。在通过网络访问一个功能的情况下，我们不能轻易假设网络状况稳定可靠。所以当一个请求发出后没有接收到对方的回应，那我们该如何判定本次操作成功与否?

下面以保存操作为例来说明一下针对功能和针对资源进行操作的区别：

- 针对功能设计系统

  保存一个 Employee 对象，没有接收到返回结果，判定操作失败，再保存一次。但是其实在服务器端保存操作已经成功了，只是返回结果在网络传输过程中丢失了。而第二次的补救行为则保存了重复、冗余但 id 不同的数据，这对整个系统数据来说是一种破坏。

- 针对资源设计系统

  针对 id 为 3278 的资源执行操作，服务器端会判断指定 id 的资源是否存在。如果不存在，则执行保存操作新建数据；如果存在，则执行更新操作。所以这个操作不论执行几次，对系统的影响都是一样的。在网络状态不可靠的情况下可以多次重试，不会破坏系统数据。

> 幂等性：如果一个操作执行一次和执行 N 次对系统的影响相同，那么我们就说这个操作满足幂等性。而幂等性正是 REST 规范所倡导的。

3. RESTFul风格的架构特点

3.1 通过URL就知道要操作什么资源

REST是针对资源设计系统，所以在REST中一个URL就对应一个资源， 为实现操作**幂等性**奠定基础。

3.2 通过Http请求的方式就知道要对资源进行何种操作

在REST中，针对同一资源的增删改查操作的URL是完全相同的，它是通过Http协议的不同请求方式来区分不同操作的

REST 风格**主张**在项目设计、开发过程中，具体的操作符合 HTTP 协议定义的请求方式的**语义**。

操作	请求方式
查询操作	GET
保存操作	POST
删除操作	DELETE
更新操作	PUT

> 另有一种说法：
>
> - POST 操作针对功能执行，没有锁定资源 id，是非幂等性操作。
> - PUT 操作锁定资源 id，即使操作失败仍然可以针对原 id 重新执行，对整个系统来说满足幂等性。
>   - id 对应的资源不存在：执行保存操作
>   - id 对应的资源存在：执行更新操作

3.3 URL更加简洁也更加隐晦

REST风格提倡 URL 地址使用统一的风格设计，从前到后各个单词使用斜杠分开，不使用问号键值对方式携带请求参数，而是将要发送给服务器的数据作为 URL 地址的一部分，以保证整体风格的一致性。还有一点是不要使用请求扩展名。

使用问号键值对的方式给服务器传递数据太明显，容易被人利用来对系统进行破坏。使用 REST 风格携带数据不再需要明显的暴露数据的名称。

操作	传统风格	REST 风格
保存	/CRUD/saveEmp	URL 地址：/CRUD/emp 请求方式：POST
删除	/CRUD/removeEmp?empId=2	URL 地址：/CRUD/emp/2 请求方式：DELETE
更新	/CRUD/updateEmp	URL 地址：/CRUD/emp 请求方式：PUT
查询（表单回显）	/CRUD/editEmp?empId=2	URL 地址：/CRUD/emp/2 请求方式：GET

第二节 四种请求方式的映射

1. 为什么要进行请求方式的映射

在 HTML 中，GET 和 POST 请求可以天然实现，但是 DELETE 和 PUT 请求无法直接做到。SpringMVC 提供了 **HiddenHttpMethodFilter** 帮助我们**将 POST 请求转换为 DELETE 或 PUT 请求。**

2. 具体执行映射操作

2.1 映射PUT 请求

2.1.1 修改web.xml文件

```
<!--一定要配置在解决乱码的Filter之后-->
<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>
```

2.1.2 页面的表单

- 要点1：原请求方式必须是 post
- 要点2：新的请求方式名称通过请求参数发送
- 要点3：请求参数名称必须是 _method
- 要点4：请求参数的值就是要改成的请求方式

```
<form th:action="@{/rest/movie}" method="post">
    <input type="hidden" name="_method" value="put"/>
    <button>发送请求</button>
</form>
```

2.1.3 handler方法

```
@PutMapping("/movie")
public String updateMovie(){
    logger.debug("PUT请求....");
    return "target";
}
```

2.2 映射DELETE请求

2.2.1 web.xml中要维持之前映射PUT请求的配置

2.2.2 前端页面

通常删除超链接会出现在列表页面：

```
<h3>将XXX请求转换为DELETE请求</h3>
<div id="app">
    <table id="dataTable">
        <tr>
            <th>姓名</th>
            <th>年龄</th>
            <th>删除</th>
        </tr>
        <tr>
            <td>张三</td>
            <td>40</td>
            <td>
                <a th:href="@{/rest/movie}" @click="deleteMovie">删除</a>
            </td>
        </tr>
        <tr>
            <td>李四</td>
```

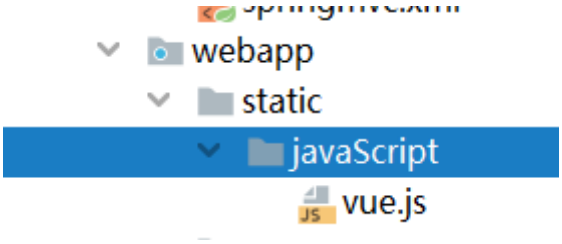
```
            <td>30</td>
            <td>
                <a th:href="@{/rest/movie}" @click="deleteMovie">删除</a>
            </td>
        </tr>
    </table>
</div>
```

创建负责转换的表单

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

使用vue给删除超链接绑定单击响应函数:

1. 引入vue




```
<!--引入vue-->
<script th:src="@{/static/javaScript/vue.js}"></script>
```

2. 绑定单击响应函数

```
var vue = new Vue({
    "el":"#app",
    "methods":{
        deleteMovie(){
            console.log("aaaaaaa")
            //真正发送删除请求:
            //1. 阻止标签的默认行为
            event.preventDefault()
            //2. 创建一个空表单:先动态设置表单的action
            var myForm = document.getElementById("myForm");
            myForm.action = event.target.href
            //并且使用js代码提交表单
            myForm.submit()
        }
    }
});
```

2.2.3 handler方法

```
@DeleteMapping("/movie")
public String deleteMovieById(){
    logger.debug("DELETE请求....");
    return "target";
}
```

第三节 PathVariable注解获取路径参数

1. REST 风格路径参数

请看下面链接：

> /emp/20
>
> /shop/product/iphone

如果我们想要获取链接地址中的某个部分的值，就可以使用 @PathVariable 注解，例如上面地址中的20、iphone部分。

2. 具体操作

2.1 单个路径参数

2.1.1 发送请求携带参数

```
<a th:href="@{/rest/movie/2}">携带参数movieId</a>
```

2.1.2 handler方法获取路径参数

```
//注意:{movieId是一个占位符}，表示获取该位置的值，@PathVariable("movieId")表示获取占位符为"movieId"的值
@GetMapping("/movie/{movieId}")
public String findMovieById(@PathVariable("movieId") Integer movieId){
    logger.debug("GET请求...."+movieId);
    return "target";
}
```

2.2 多个路径参数

2.2.1 发送请求携带参数

```
<a th:href="@{/rest/movie/2/22/123}">携带多个参数</a>
```

2.2.2 handler方法获取路径参数

```
@GetMapping("/movie/{categoryId}/{groupId}/{movieId}")
public String findMovieById(@PathVariable("categoryId") Integer
categoryId,@PathVariable("groupId")Integer groupId,@PathVariable("movieId")
Integer movieId){
    logger.debug("GET请求...."+categoryId+":"+groupId+":"+movieId);
    return "target";
}
```

第四节 RESTFul综合案例

1. 案例准备

将前面的传统CRUD案例复制并且重新导入

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	/movie	GET
删除	/movie/2	DELETE
跳转到添加数据的表单	/movie/add.html	GET
执行保存	/movie	POST
跳转到更新数据的表单	/movie/2	GET
执行更新	/movie	PUT

3. 访问首页

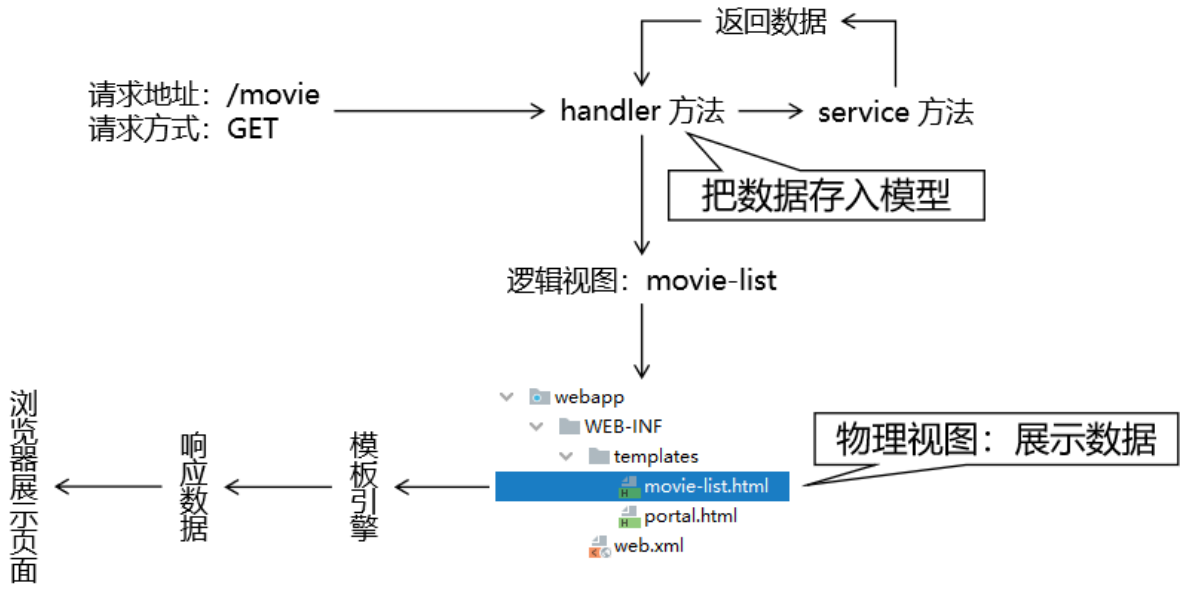
3.1 配置view-controller

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

其它不用做任何修改

4. 查询全部

4.1 案例流程



4.2 具体实现

4.2.1 handler方法

```
//因为类上的RequestMapping注解的值已经是"/movie"了
@GetMapping
public String showList(Model model){
    //1. 调用业务层的方法查询所有的movie
    List<Movie> movieList = movieService.getAll();
    //2. 将查询到的所有movie存储到请求域
    model.addAttribute("movieList",movieList);
    //3. 解析Thymeleaf模板显示所有movie
    return "list";
}
```

4.2.2 portal.html页面a标签的路径

```
<a th:href="@{/movie}">显示电影列表</a>
```

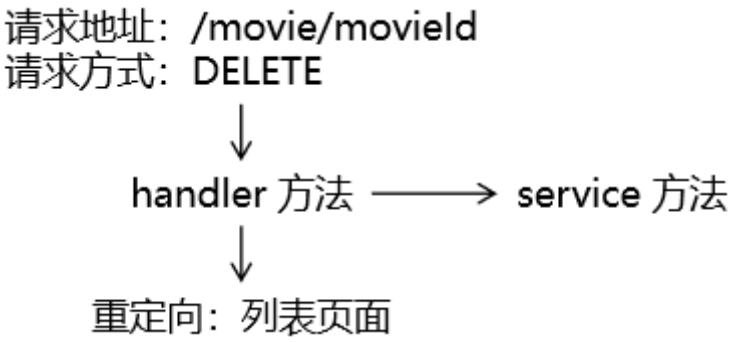
4.2.3 list页面展示(不用修改)

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
        <title>电影列表页面</title>
        <style type="text/css">
            table {
                border-collapse: collapse;
                margin: 0px auto 0px auto;
            }
            table th,td {
                border: 1px solid black;
                text-align: center;
            }
        </style>
    </head>
    <body>
        <table>
            <thead>
                <tr>
                    <th>电影ID</th>
                    <th>电影名称</th>
                    <th>电影票价格</th>
                    <th>删除</th>
                    <th>更新</th>
                </tr>
            </thead>
            <tbody th:if="${#lists.isEmpty(movieList)}">
                <tr>
                    <td colspan="5">抱歉！没有查询到数据！</td>
                </tr>
            </tbody>
            <tbody th:unless="${#lists.isEmpty(movieList)}">
                <tr th:each="movie : ${movieList}">
                    <td th:text="${movie.movieId}">电影ID</td>
                    <td th:text="${movie.movieName}">电影名称</td>
                    <td th:text="${movie.moviePrice}">电影票价格</td>
                    <td>
```

```
                        <a
th:href="@{/movie/removeMovie(movieId=${movie.movieId})}">删除</a>
                    </td>
                    <td><a
th:href="@{/movie/toUpdatePage(movieId=${movie.movieId})}">更新</a></td>
                </tr>
            </tbody>
            <tfoot>
                <tr>
                    <td colspan="5"><a th:href="@{/add.html}">添加</a></td>
                </tr>
            </tfoot>
        </table>
    </body>
</html>
```

5. 删除一行数据

5.1 案例流程



重点在于将 GET 请求转换为 DELETE。基本思路是：通过一个**通用表单**，使用 **Vue** 代码先把 GET 请求转换为 POST，然后再借助 **hiddenHttpMethodFilter** 在服务器端把 POST 请求转为 DELETE。

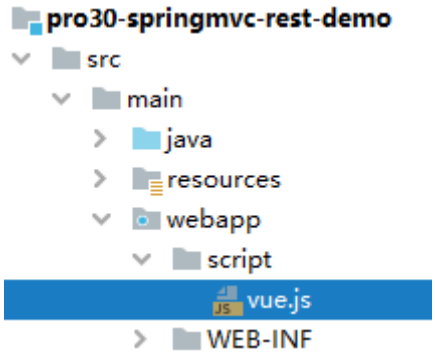
5.2 具体实现

5.2.1 创建通用表单

```
<!-- 组件名称：通用表单 -->
<!-- 组件作用：把删除超链接的 GET 请求转换为 POST，并携带 _method 请求参数 -->
<form id="convertForm" method="post">

    <!-- 请求参数作用：告诉服务器端 hiddenHttpMethodFilter 要转换的目标请求方式 -->
    <!-- 请求参数名：_method，这是 hiddenHttpMethodFilter 中规定的 -->
    <!-- 请求参数值：delete，这是因为我们希望服务器端将请求方式最终转换为 delete -->
    <input type="hidden" name="_method" value="delete"/>
</form>
```

5.2.2 删除超链接绑定单击响应函数

1. 引入vue




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

2. 删除超链接绑定单击事件

```
<a th:href="@{/movie/}+${movie.movieId}" @click="deleteMovie()">删除</a>
```

3. vue代码

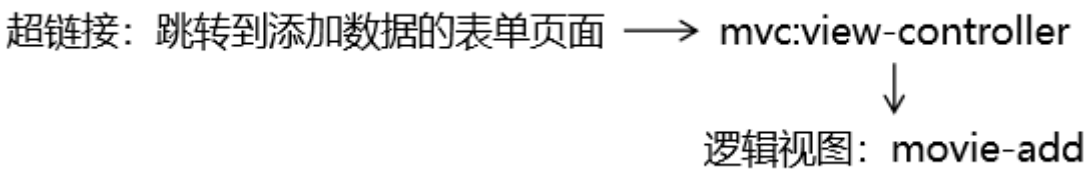
```
var vue = new Vue({
    "el":"#app",
    "methods":{
        deleteMovie(){
            //1. 阻止默认事件
            event.preventDefault()
            //2. 创建一个空表单:设置表单的action的值和当前a标签的路径一致
            var deleteForm = document.getElementById("deleteForm");
            deleteForm.action = event.target.href
            //3. 提交表单
            deleteForm.submit()
        }
    }
});
```

5.2.3 handler方法

```
@DeleteMapping("/{movieId}")
public String removeMovie(@PathVariable("movieId") String movieId){
    //调用业务层的方法根据id删除movie
    movieService.removeMovieById(movieId);
    //重新查询所有
    return "redirect:/movie";
}
```

6. 跳转到添加数据的表单(不用修改)

6.1 案例流程

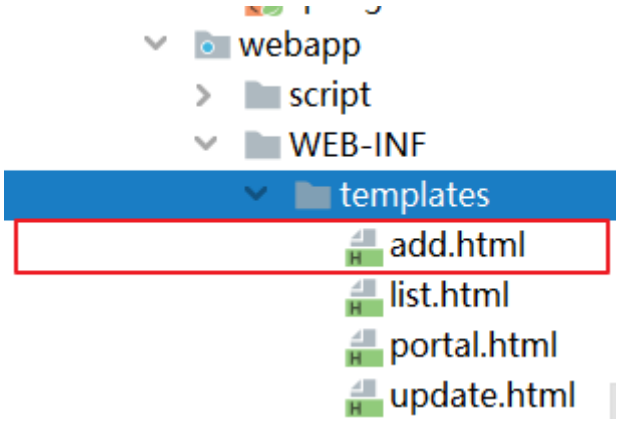


6.2 具体实现

6.2.1 配置view-controller

```
<!--8. 使用view-controller访问添加页面-->
<mvc:view-controller path="/add.html" view-name="add"/>
```

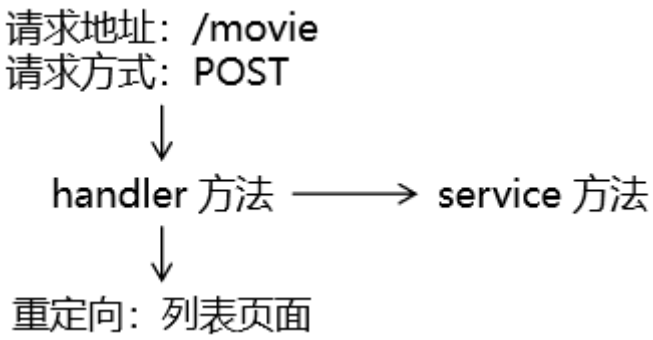
6.2.2 创建页面




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

    电影名称：<input type="text" name="movieName" /><br/>
    电影票价格：<input type="text" name="moviePrice" /><br/>

    <button type="submit">保存</button>

</form>
```

7. 执行添加

7.1 案例流程



7.2 具体实现

7.2.1 handler方法

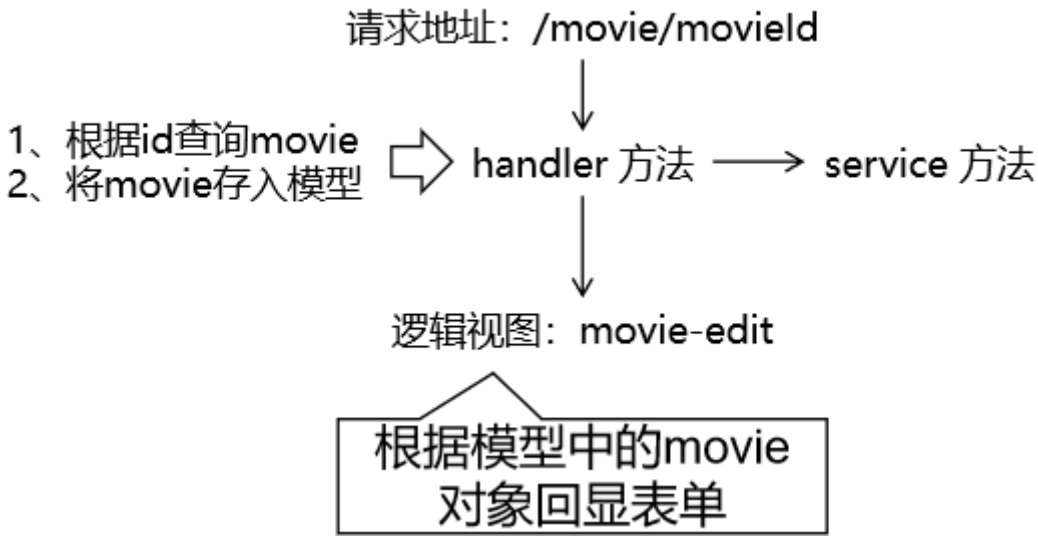
```
@PostMapping
public String addMovie(Movie movie){
    //调用业务层的方法添加Movie
    movieService.saveMovie(movie);
    //重新查询所有
    return "redirect:/movie";
}
```

7.2.2 添加页面

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>添加电影页面</title>
</head>
<body>
    <form th:action="@{/movie}" method="post">

        电影名称：<input type="text" name="movieName" /><br/>
        电影票价格：<input type="text" name="moviePrice" /><br/>

        <button type="submit">保存</button>

    </form>
</body>
</html>
```

8. 跳转到更新数据页面

8.1 案例流程



8.2 具体实现

8.2.1 handler方法

```
@GetMapping("/{movieId}")
public String toUpdatePage(@PathVariable("movieId") String movieId,Model model){
    //1. 根据要修改的电影的id查询到当前电影的信息
    Movie movie = movieService.getMovieById(movieId);
    //2. 将当前电影的信息存储到请求域
    model.addAttribute("movie",movie);
    //3. 解析update页面
    return "update";
}
```

8.2.2 修改更新的超链接

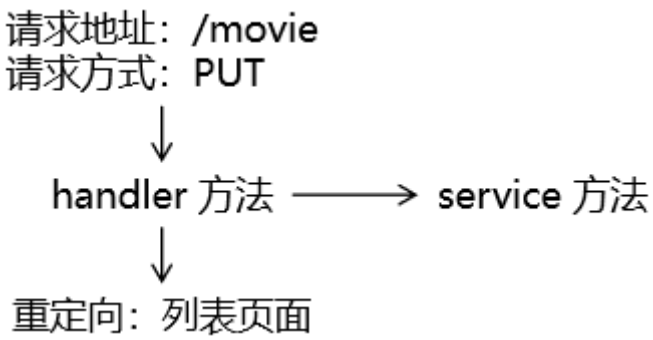
```
<a th:href="@{/movie/}+${movie.movieId}">更新</a>
```

8.2.3 页面回显数据

```
<form th:action="@{/movie}" method="post">
    <!--映射成put请求-->
    <input type="hidden" name="_method" value="put"/>
    <!--使用隐藏域绑定movieId-->
    <input type="hidden" name="movieId" th:value="${movie.movieId}" />
    电影名称：<input type="text" name="movieName" th:value="${movie.movieName}" />
<br/>
    电影票价格：<input type="text" name="moviePrice" th:value="${movie.moviePrice}"
/><br/>
    <button type="submit">更新</button>
</form>
```

9. 执行更新

9.1 案例流程



9.2 具体实现

9.2.1 handler方法

```
@PutMapping
public String updateMovie(Movie movie){
    //1. 调用业务层的方法修改电影信息
    movieService.updateMovie(movie);
    //2. 重新查询所有
    return "redirect:/movie";
}
```

第五节 REST的小结

1. 解决什么问题: 从根据功能设计url变成根据资源设计url
2. 是什么:资源状态转移**Re**presentational **S**tate **T**ransfer
3. 特征:
    1. 一个url就代表一个资源，也就是说通过url我们可以知道当前请求操作的是什么资源
    2. 请求方式代表操作,也就是说通过请求方式我们知道当前请求对当前资源做的是何种操作
4. 效果:
    1. url更加简洁
    2. url更加隐晦

3. 操作的幂等性

5. 怎么实现:

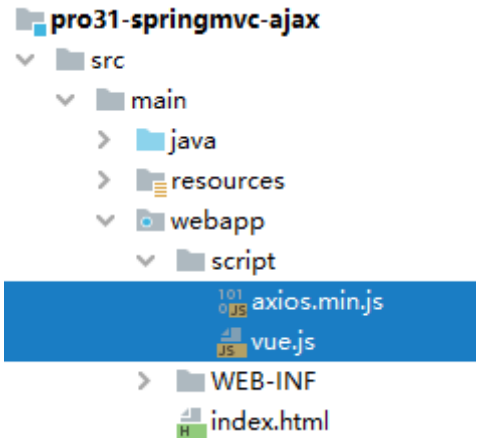
    1. 请求方式映射:HTML默认只能发送GET和POST请求，SpringMVC在某些时候需要将POST请求映射成PUT或者DELETE请求
    2. GetMapping、PostMapping、DeleteMapping、PutMapping要选择情况使用
    3. 合理设计url
    4. 使用Pathvariable注解获取REST风格的url上的路径参数

第二章 Ajax交互(重要)

第一节 获取请求参数

1. 获取普通类型参数(与之前获取请求参数的方式一致)

1.1 引入JavaScript库




```
<script type="text/javascript" src="script/vue.js"></script>
<script type="text/javascript" src="script/axios.min.js"></script>
```

1.2 前端代码

```
sendCommon(){
    axios({
        "method":"POST",
        "url":"ajax/commonParameter",
        //params表示携带普通类型的参数
        "params":{
            "userName":"tom",
            "password":"123456"
        }
    }).then(response => {

    })
}
```

1.3 后端代码

```
@RequestMapping("/commonParameter")
public String commonParameter(User user){
    //获取异步请求携带的普通类型参数，和以前获取同步请求携带的参数是一样的
    logger.debug(user.toString());
    return "hello";
}
```

2. 获取JSON请求体参数

2.1 前端代码

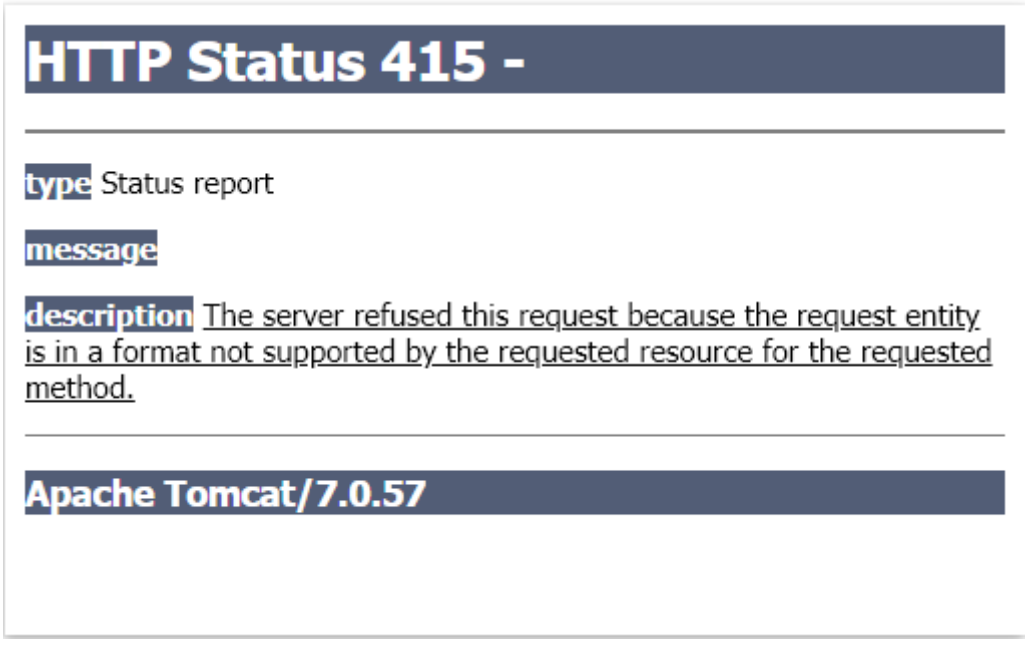
```
sendJsonBody(){
    axios({
        "method":"POST",
        "url":"ajax/jsonBodyParameter",
        //data表示携带json请求体类型的参数
        "data":{
            "userName":"tom",
            "password":"123456"
        }
    }).then(response => {

    })
}
```

2.2 后端代码

2.2.1 引入jackson依赖

```
<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.12.1</version>
</dependency>
```

如果忘记导入这个依赖，会看到下面的错误页面



关于 SpringMVC 和 Jackson jar包之间的关系，需要注意：当 SpringMVC 需要解析 JSON 数据时就需要使用 Jackson 的支持。但是 SpringMVC 的 jar 包并没有依赖 Jackson，所以需要我们自己导入。

我们自己导入时需要注意：SpringMVC 和 Jackson 配合使用有版本的要求。二者中任何一个版本太高或太低都不行。

SpringMVC 解析 JSON 数据包括两个方向：

- 从 JSON 字符串到 Java 实体类。
- 从 Java 实体类到 JSON 字符串。

另外，如果导入了 Jackson 依赖，但是没有开启 mvc:annotation-driven 功能，那么仍然会返回上面的错误页面。

也就是说，我们可以这么总结 SpringMVC 想要解析 JSON 数据需要两方面支持：

- mvc:annotation-driven
- 引入 Jackson 依赖

还有一点，如果运行环境是 Tomcat7，那么在 Web 应用启动时会抛出下面异常：

> org.apache.tomcat.util.bcel.classfile.ClassFormatException: Invalid byte tag in constant pool: 19

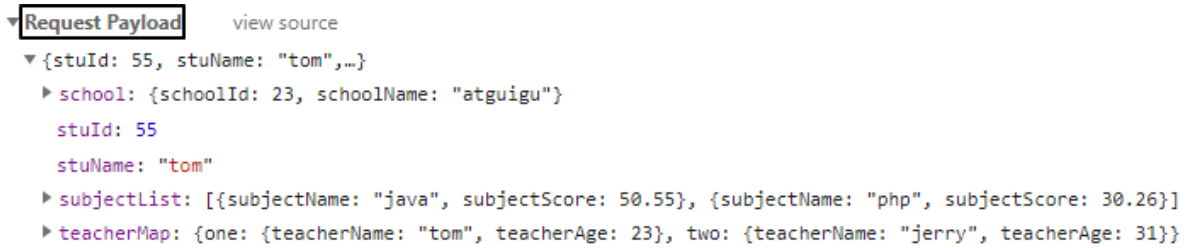
解决办法是使用 Tomcat8 或更高版本。

2.2.2 handler方法

```
@RequestMapping("/jsonBodyParameter")
public String jsonBodyParameter(@RequestBody User user){
    //1. 获取Json请求体类型的参数必须要封装到POJO对象或者是Map中
    //2. POJO参数或者Map参数的前面一定要加入RequestBody注解
    //3. 你的项目中一定要引入jackson的依赖(因为SpringMVC默认支持jackson)
    //获取Json请求体类型的请求参数和获取普通类型的参数不一样
    logger.debug(user.toString());
    return "success";
}
```

2.2.3 RequestBody注解

适用 @RequestBody 注解的场景：请求体整个是一个 JSON 数据



Request Payload 翻译成中文大致可以说：请求负载。

第二节 响应JSON类型数据

1. 前端代码

```
sendJsonBody(){
    axios({
        "method":"POST",
        "url":"ajax/jsonBodyParameter",
        //data表示携带json请求体类型的参数
        "data":{
            "userName":"tom",
            "password":"123456"
        }
    }).then(response => {
        console.log(response.data.movieName)
    })
}
```

2. 后端代码

前提是项目中引入了jackson的依赖

```
@ResponseBody
@RequestMapping("/jsonBodyParameter")
public Movie jsonBodyParameter(@RequestBody User user){
    logger.debug(user.toString());

    Movie movie = new Movie(1, "西游记", 40.0);
    //目标:将movie转成json字符串响应给客户端
    //1. handler方法的返回值就是你要转成json的那个对象
    //2. handler方法上必须添加ResponseBody注解
    //3. 你的项目中一定要引入了jackson的依赖
    return movie;
}
```

3. 常见错误

3.1 500 错误

出现上面的错误页面，表示SpringMVC 为了将 实体类对象转换为 JSON 数据, 需要转换器。但是现在找不到转换器。它想要成功完成转换需要两方面支持：

- mvc:annotation-driven
- 引入Jackson依赖

3.2 406 错误(了解)

问题出现的原因：

- 请求地址扩展名：html
- 服务器端打算返回的数据格式：JSON

上面二者不一致。SpringMVC 要坚守一个商人的良心，不能干『挂羊头，卖狗肉』的事儿。解决办法有三种思路：

- 第一种方法：不使用请求扩展名
- 第二种方法：使用和实际返回的数据格式一致的扩展名

```
<servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>*.html</url-pattern>
    <url-pattern>*.json</url-pattern>
</servlet-mapping>
```

- 第三种方法：使用一个 HTTP 协议中没有被定义的扩展名，例如：*.do

4. RestController注解

4.1 提取ResponseBody注解

如果类中每个方法上都标记了 @ResponseBody 注解，那么这些注解就可以提取到类上。

4.2 合并注解

类上的ResponseBody 注解可以和Controller 注解合并为RestController 注解。所以使用了 RestController 注解就相当于给类中的每个方法都加了ResponseBody 注解。

4.3 RestController源码

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Controller
@ResponseBody
public @interface RestController {

    /**
     * The value may indicate a suggestion for a logical component name,
     * to be turned into a Spring bean in case of an autodetected component.
     * @return the suggested component name, if any (or empty String otherwise)
     * @since 4.0.1
     */
    @AliasFor(annotation = Controller.class)
    String value() default "";
}
```