

1. 什么是MVC

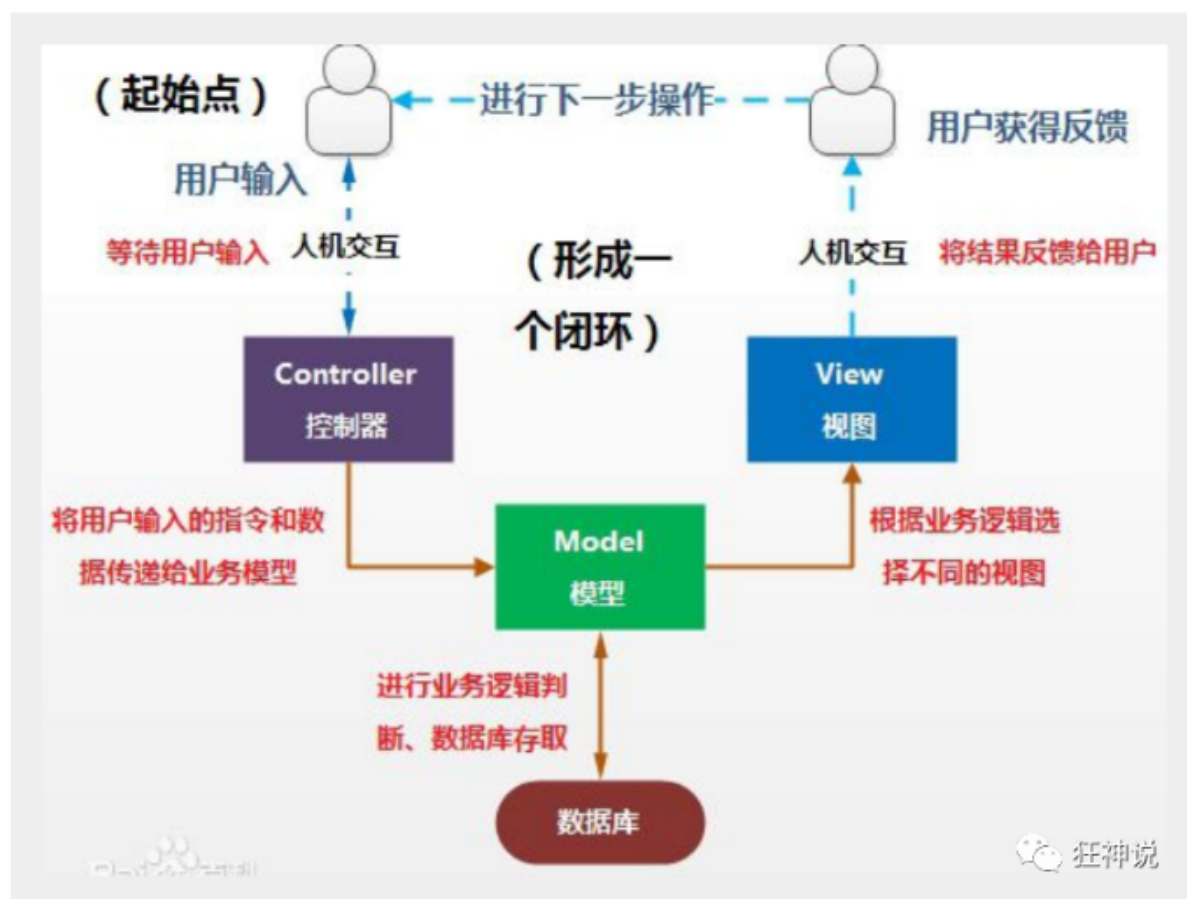
- MVC是模型(Model)、视图(View)、控制器(Controller)的简写，是一种软件设计规范。
- 是将业务逻辑、数据、显示分离的方法来组织代码。
- MVC主要作用是**降低了视图与业务逻辑间的双向耦合**。
- MVC不是一种设计模式，**MVC是一种架构模式**。当然不同的MVC存在差异。

Model (模型)：数据模型，提供要展示的数据，因此包含数据和行为，可以认为是领域模型或JavaBean组件（包含数据和行为），不过现在一般都分离开来：Value Object（数据Dao）和服务层（行为Service）。也就是模型提供了模型数据查询和模型数据的状态更新等功能，包括数据和业务。

View (视图)：负责进行模型的展示，一般就是我们见到的用户界面，客户想看到的东西。

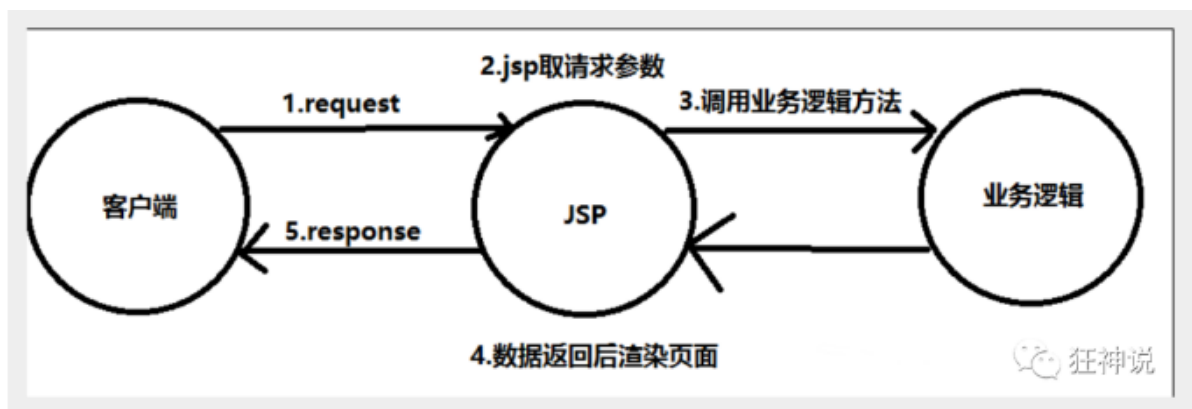
Controller (控制器)：接收用户请求，委托给模型进行处理（状态改变），处理完毕后把返回的模型数据返回给视图，由视图负责展示。也就是说控制器做了个调度员的工作。

最典型的MVC就是JSP + servlet + javabean的模式。



1.1 Model1时代

- 在web早期的开发中，通常采用的都是Model1。
- Model1中，主要分为两层，视图层和模型层。

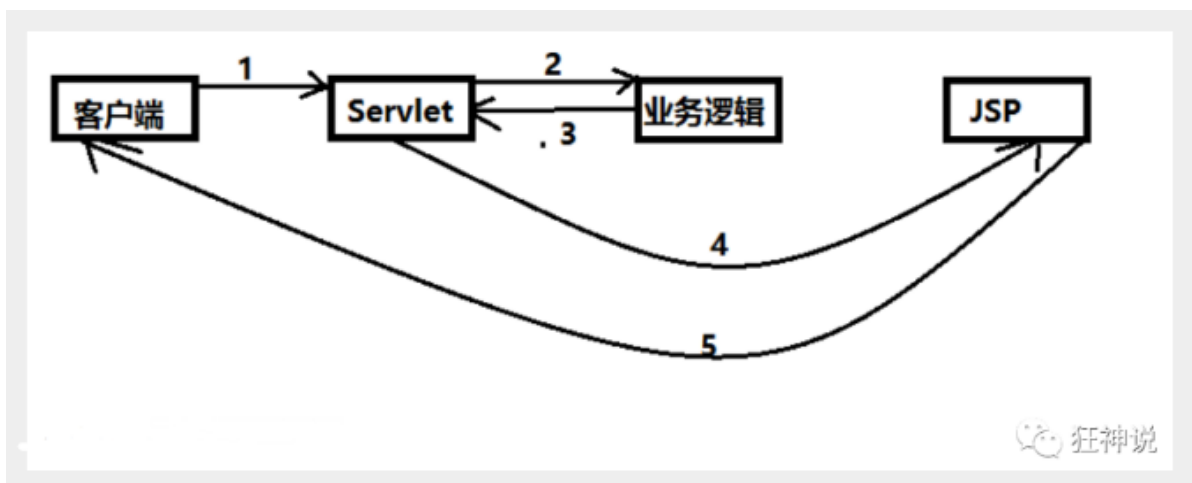


Model1优点：架构简单，比较适合小型项目开发；

Model1缺点：JSP职责不单一，职责过重，不便于维护；

1.2 Model2时代

Model2把一个项目分成三部分，包括视图、控制、模型。



用户发请求

1. Servlet接收请求数据，并调用对应的业务逻辑方法
2. 业务处理完毕，返回更新后的数据给servlet
3. servlet转向到JSP，由JSP来渲染页面
4. 响应给前端更新后的页面

职责分析：

Controller：控制器

1. 取得表单数据
2. 调用业务逻辑
3. 转向指定的页面

Model：模型

1. 业务逻辑
2. 保存数据的状态

View：视图

1. 显示页面

Model2这样不仅提高的代码的复用率与项目的扩展性，且大大降低了项目的维护成本。Model 1模式的实现比较简单，适用于快速开发小规模项目，Model1中JSP页面身兼View和Controller两种角色，将控制逻辑和表现逻辑混杂在一起，从而导致代码的重用性非常低，增加了应用的扩展性和维护的难度。Model2消除了Model1的缺点。

1.3 回顾Servlet

【Moudle: springmvc-01-servlet】

1. 新建一个Maven工程当做父工程！ pom依赖！

```
1  <dependencies>
2      <dependency>
3          <groupId>org.springframework</groupId>
4          <artifactId>spring-webmvc</artifactId>
5          <version>5.2.0.RELEASE</version>
6      </dependency>
7
8      <dependency>
9          <groupId>javax.servlet</groupId>
10         <artifactId>servlet-api</artifactId>
11         <version>2.5</version>
12     </dependency>
13
14     <dependency>
15         <groupId>javax.servlet.jsp</groupId>
16         <artifactId>jsp-api</artifactId>
17         <version>2.2</version>
18     </dependency>
19
20     <dependency>
21         <groupId>javax.servlet</groupId>
22         <artifactId>jstl</artifactId>
23         <version>1.2</version>
24     </dependency>
25
26     <dependency>
27         <groupId>junit</groupId>
28         <artifactId>junit</artifactId>
29         <version>4.12</version>
30     </dependency>
31 </dependencies>
```

2. 建立一个Moudle: springmvc-01-servlet， 添加Web app的支持！
3. 导入servlet 和 jsp 的 jar 依赖

```

1  <dependencies>
2      <dependency>
3          <groupId>javax.servlet</groupId>
4          <artifactId>servlet-api</artifactId>
5          <version>2.5</version>
6      </dependency>
7      <dependency>
8          <groupId>javax.servlet.jsp</groupId>
9          <artifactId>jsp-api</artifactId>
10         <version>2.2</version>
11     </dependency>
12 </dependencies>

```

4. 编写一个Servlet类，用来处理用户的请求

```

1  public class HelloServlet extends HttpServlet {
2
3      @Override
4      protected void doGet(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
5          // 1. 获取前端参数
6          String method = req.getParameter("method");
7          if (method.equals("add")){
8              req.getSession().setAttribute("msg", "执行了add方法");
9          }
10         if (method.equals("delete")){
11             req.getSession().setAttribute("msg", "执行了delete方法");
12         }
13         // 2. 调用业务层
14
15         // 3. 视图转发或者重定向
16         req.getRequestDispatcher("./WEB-INF/jsp/test.jsp").forward(req, resp);
17     }
18
19
20     @Override
21     protected void doPost(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
22         doGet(req, resp);
23     }
24 }

```

5. 编写Hello.jsp，在WEB-INF目录下新建一个jsp的文件夹，新建hello.jsp

```

1 <%@ page contentType="text/html;charset=UTF-8" language="java" %>
2 <html>
3 <head>
4     <title>Test</title>
5 </head>
6 <body>
7
8     ${msg}
9
10 </body>
11 </html>

```

6. 在web.xml中注册Servlet

```

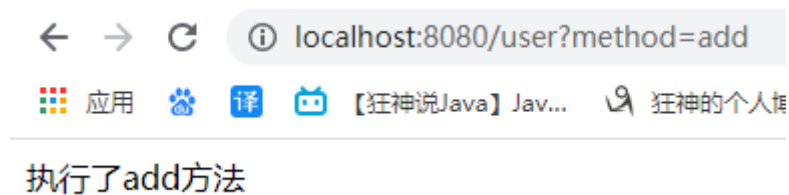
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
5     version="4.0">
6
7     <servlet>
8         <servlet-name>hello</servlet-name>
9         <servlet-class>com.longg.servlet.HelloServlet</servlet-class>
10    </servlet>
11    <servlet-mapping>
12        <servlet-name>hello</servlet-name>
13        <url-pattern>/hello</url-pattern>
14    </servlet-mapping>
15 </web-app>

```

7. 配置Tomcat, 并启动测试

- localhost:8080/hello?method=add
- localhost:8080/hello?method=delete

8. 访问结果



MVC框架要做哪些事情

1. 将 Url 映射到 Java 类或 Java 类的方法;
2. 封装用户提交的数据;
3. 处理请求--调用相关的业务处理--封装响应数据;
4. 将响应的数据进行渲染, Jsp / Html 等表示层数据;

说明:

常见的服务器端MVC框架有: Struts、Spring MVC、ASP.NET MVC、Zend Framework、JSF;

常见前端MVC框架: vue、angularjs、react、backbone;

由MVC演化出了另外一些模式如: MVP、MVVM 等等。

2. 什么是SpringMVC

2.1 概述



Spring MVC是Spring Framework的一部分，是基于Java实现MVC的轻量级Web框架。

查看官方文档: <https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#spring-web>

我们为什么要学习SpringMVC呢?

Spring MVC的特点:

1. 轻量级，简单易学
2. 高效，基于请求响应的MVC框架
3. 与Spring兼容性好，无缝结合
4. 约定优于配置
5. 功能强大：RESTful、数据验证、格式化、本地化、主题等
6. 简洁灵活

Spring的web框架围绕**DispatcherServlet** [调度Servlet] 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形式进行开发，十分简洁；

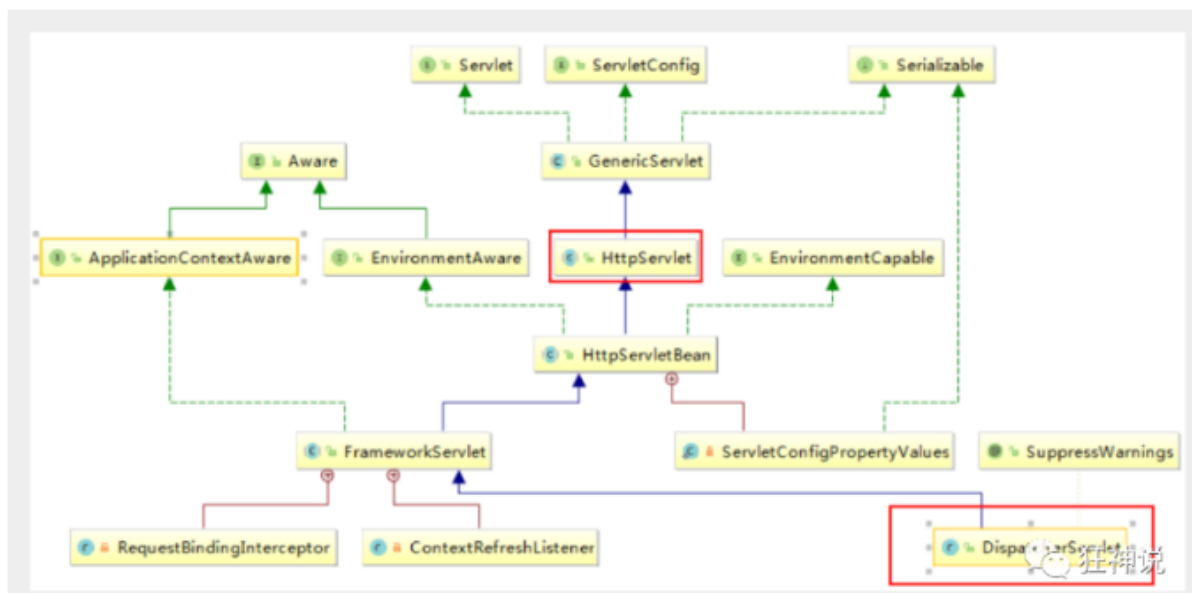
正因为SpringMVC好，简单，便捷，易学，天生和Spring无缝集成(使用SpringIoC和Aop)，使用约定优于配置，能够进行简单的junit测试，支持Restful风格，异常处理，本地化，国际化，数据验证，类型转换，拦截器 等等.....所以我们要学习。

最重要的一点还是用的人多，使用的公司多。

2.2 中心控制器

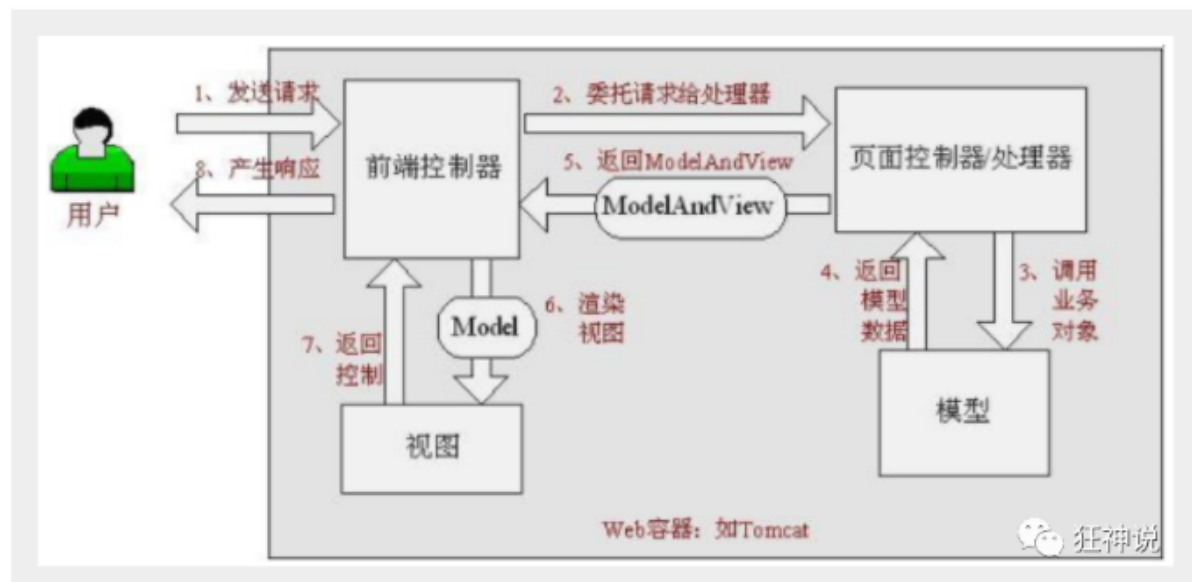
Spring的web框架围绕DispatcherServlet设计。DispatcherServlet的作用是将请求分发到不同的处理器。从Spring 2.5开始，使用Java 5或者以上版本的用户可以采用基于注解的controller声明方式。

Spring MVC框架像许多其他MVC框架一样，**以请求为驱动，围绕一个中心Servlet分派请求及提供其他功能，DispatcherServlet是一个实际的Servlet (它继承自HttpServlet 基类)。**

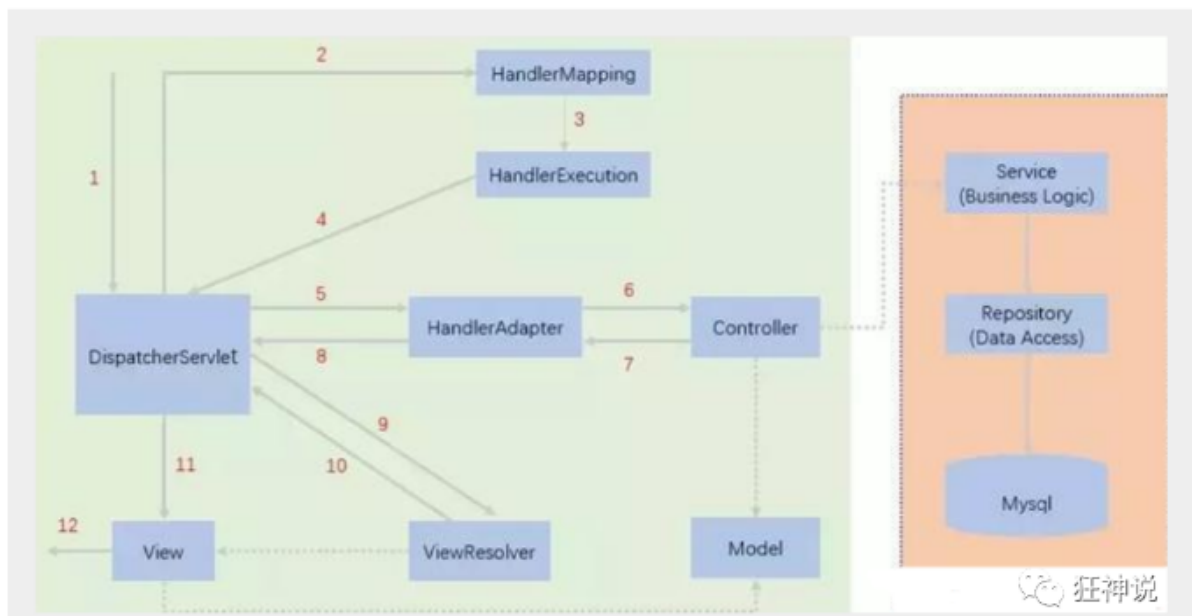


SpringMVC的原理如下图所示：

当发起请求时被前置的控制器拦截到请求，根据请求参数生成代理请求，找到请求对应的实际控制器，控制器处理请求，创建数据模型，访问数据库，将模型响应给中心控制器，控制器使用模型与视图渲染视图结果，将结果返回给中心控制器，再将结果返回给请求者。



2.3 SpringMVC执行原理



图为SpringMVC的一个较完整的流程图，实线表示SpringMVC框架提供的技术，不需要开发者实现，虚线表示需要开发者实现。

简要分析执行流程

1. DispatcherServlet表示前置控制器，是整个SpringMVC的控制中心。用户发出请求，DispatcherServlet接收请求并拦截请求。
 - 我们假设请求的url为：<http://localhost:8080/SpringMVC/hello>
 - **如上url拆分成三部分：**
 - <http://localhost:8080>服务器域名
 - SpringMVC部署在服务器上的web站点
 - hello表示控制器
 - 通过分析，如上url表示为：请求位于服务器localhost:8080上的SpringMVC站点的hello控制器。
2. HandlerMapping为处理器映射。DispatcherServlet调用HandlerMapping,HandlerMapping根据请求url查找Handler。
3. HandlerExecution表示具体的Handler,其主要作用是根据url查找控制器，如上url被查找控制器为：hello。
4. HandlerExecution将解析后的信息传递给DispatcherServlet,如解析控制器映射等。
5. HandlerAdapter表示处理器适配器，其按照特定的规则去执行Handler。
6. Handler让具体的Controller执行。
7. Controller将具体的执行信息返回给HandlerAdapter,如ModelAndView。
8. HandlerAdapter将视图逻辑名或模型传递给DispatcherServlet。
9. DispatcherServlet调用视图解析器(ViewResolver)来解析HandlerAdapter传递的逻辑视图名。
10. 视图解析器将解析的逻辑视图名传给DispatcherServlet。
11. DispatcherServlet根据视图解析器解析的视图结果，调用具体的视图。
12. 最终视图呈现给用户。

3. HelloSpringMVC(配置版)

【Moudle: springmvc-02-hellomvc】

1. 新建一个Moudle , 添加web的支持;
2. 确定导入了SpringMVC 的依赖;
3. 配置web.xml , 注册DispatcherServlet;

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
5                             http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
6         version="4.0">
7     <!--1.注册DispatcherServlet-->
8     <servlet>
9         <servlet-name>springmvc</servlet-name>
10        <servlet-
11class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
12        <!--关联一个springmvc的配置文件:【servlet-name】-servlet.xml-->
13        <init-param>
14            <param-name>contextConfigLocation</param-name>
15            <param-value>classpath:springmvc-servlet.xml</param-value>
16        </init-param>
17        <!--启动级别-1-->
18        <load-on-startup>1</load-on-startup>
19    </servlet>
20
21    <!-- / 匹配所有的请求; (不包括.jsp) -->
22    <!-- /* 匹配所有的请求; (包括.jsp) -->
23    <servlet-mapping>
24        <servlet-name>springmvc</servlet-name>
25        <url-pattern>/</url-pattern>
26    </servlet-mapping>
27 </web-app>
```

4. 编写SpringMVC 的配置文件! 名称: springmvc-servlet.xml : [servletname]-servlet.xml说明, 这里的名称要求是按照官方来的;

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://www.springframework.org/schema/beans
5                           http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7 </beans>
```

5. 在SpringMVC 的配置文件中添加**处理器映射器**;

```

1 <!--添加处理器映射器-->
2 <bean
  class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"
/>

```

6. 在SpringMVC 的配置文件中添加**处理器适配器**;

```

1 <!--添加处理器适配器-->
2 <bean
  class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"
/>

```

7. 在SpringMVC 的配置文件中添加**视图解析器**;

```

1 <!--视图解析器InternalResourceViewResolver: 解析DispatcherServlet给他的
  ModelAndView
2     1.获取ModelAndView的数据
3     2.解析ModelAndView的视图名字
4     3.拼接视图名字,找到对应的视图 /WEB-INF/jsp/hello.jsp
5     4.将数据渲染到视图中
6 -->
7 <bean
  class="org.springframework.web.servlet.view.InternalResourceViewResolver"
  id="InternalResourceViewResolver">
8     <!--前缀-->
9     <property name="prefix" value="/WEB-INF/jsp/" />
10    <!--后缀-->
11    <property name="suffix" value=".jsp" />
12 </bean>

```

8. 编写我们要操作业务Controller , 要么实现Controller接口, 要么增加注解; 需要返回一个 ModelAndView, 装数据, 封视图;

```

1 package com.longg.controller;
2
3 import org.springframework.web.servlet.ModelAndView;
4 import org.springframework.web.servlet.mvc.Controller;
5
6 import javax.servlet.http.HttpServletRequest;
7 import javax.servlet.http.HttpServletResponse;
8
9 public class HelloController implements Controller {
10
11     public ModelAndView handleRequest(HttpServletRequest
  httpServletRequest, HttpServletResponse httpServletResponse) throws
  Exception {
12         //ModelAndView 模型和视图
13         ModelAndView mv = new ModelAndView();
14
15         //封装对象,放在ModelAndView中
16         mv.addObject("msg","HelloSpringMVC!");
17
18         //封装要跳转的视图,放在ModelAndView中
19         mv.setViewName("hello"); // :/WEB-INF/jsp/hello.jsp
20         return mv;

```

```

21     }
22 }

```

9. 将自己的类交给SpringIOC容器，注册bean

```

1 <!--处理器Handler    id是请求路径: http://localhost/long-->
2 <bean id="/long" class="com.longg.controller.HelloController"/>

```

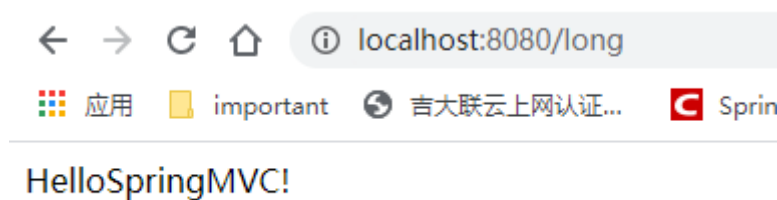
10. 写要跳转的jsp页面，显示ModelAndView存放的数据，以及我们的正常页面；

```

1 <%@ page contentType="text/html;charset=UTF-8" language="java" %>
2 <html>
3 <head>
4     <title>Long</title>
5 </head>
6 <body>
7
8     ${msg}
9
10 </body>
11 </html>

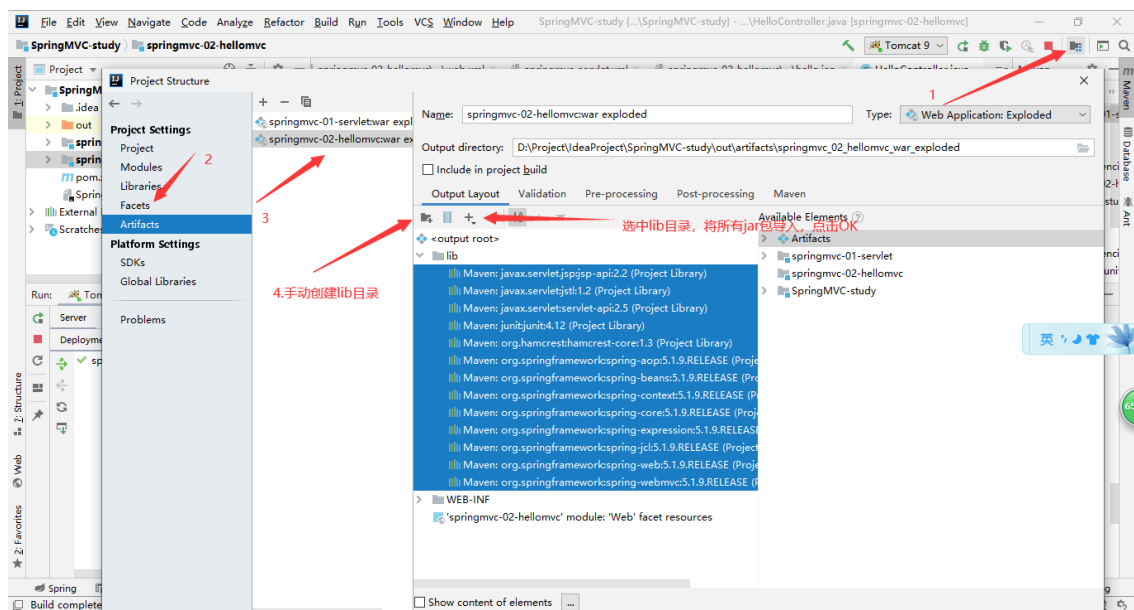
```

11. 配置Tomcat 启动测试！



可能遇到的问题：访问出现404，排查步骤：

1. 查看控制台输出，看一下是不是缺少了什么jar包。
2. 如果jar包存在，显示无法输出，就在IDEA的项目发布中，添加lib依赖！（与classes同级目录）



3. 重启Tomcat 即可解决！

小结：看这个估计大部分同学都能理解其中的原理了，但是我们实际开发才不会这么写，不然就疯了，还学这个玩意干嘛！我们来看个注解版实现，这才是SpringMVC的精髓，到底有多么简单，看这个图就知道了。



4. 使用注解开发SpringMVC（注解版）

【Moudle：springmvc-03-annotation】

4.1 使用注解开发的步骤

1. 新建一个Moudle，添加web支持！【Moudle：springmvc-03-annotation】
2. 由于Maven可能存在资源过滤的问题，我们将配置完善

```
1  <build>
2      <resources>
3          <resource>
4              <directory>src/main/java</directory>
5              <includes>
6                  <include>**/*.properties</include>
7                  <include>**/*.xml</include>
8              </includes>
9              <filtering>>false</filtering>
10         </resource>
11         <resource>
12             <directory>src/main/resources</directory>
13             <includes>
```

```

14         <include>**/*.properties</include>
15         <include>**/*.xml</include>
16     </includes>
17     <filtering>>false</filtering>
18 </resource>
19 </resources>
20 </build>

```

3. 在pom.xml文件引入相关的依赖:

主要有Spring框架核心库、Spring MVC、servlet,JSTL等。我们在父依赖中已经引入了!

4. 配置web.xml

注意点:

- 注意web.xml版本问题, 要最新版!
- 注册DispatcherServlet
- 关联SpringMVC的配置文件
- 启动级别为1
- 映射路径为 / 【不要用/*, 会404】

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
5                             http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
6         version="4.0">
7     <!--1.注册servlet-->
8     <servlet>
9         <servlet-name>SpringMVC</servlet-name>
10        <servlet-
11class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
12        <!--通过初始化参数指定SpringMVC配置文件的位置, 进行关联-->
13        <init-param>
14            <param-name>contextConfigLocation</param-name>
15            <param-value>classpath:springmvc-servlet.xml</param-value>
16        </init-param>
17        <!-- 启动顺序, 数字越小, 启动越早 -->
18        <load-on-startup>1</load-on-startup>
19    </servlet>
20    <!--所有请求都会被springmvc拦截 -->
21    <servlet-mapping>
22        <servlet-name>SpringMVC</servlet-name>
23        <url-pattern>/</url-pattern>
24    </servlet-mapping>
25
26 </web-app>

```

```

1 / 和 /* 的区别:
2 < url-pattern > / </ url-pattern > 不会匹配到.jsp, 只针对我们编写的请求;
3 即: .jsp 不会进入spring的 DispatcherServlet类 。
4 < url-pattern > /* </ url-pattern > 会匹配 *.jsp,
5 会出现返回 jsp视图 时再次进入spring的DispatcherServlet 类, 导致找不到对应的
  controller所以报404错。

```

5. 添加Spring MVC配置文件

- 让IOC的注解生效
- 静态资源过滤：HTML .JS .CSS . 图片， 视频 ...
- MVC的注解驱动
- 配置视图解析器

在resource目录下添加springmvc-servlet.xml配置文件，配置的形式与Spring容器配置基本类似，为了支持基于注解的IOC，设置了自动扫描包的功能，具体配置信息如下：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xmlns:mvc="http://www.springframework.org/schema/mvc"
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans.xsd
8     http://www.springframework.org/schema/context
9     https://www.springframework.org/schema/context/spring-context.xsd
10    http://www.springframework.org/schema/mvc
11    https://www.springframework.org/schema/mvc/spring-mvc.xsd">
12
13     <!-- 自动扫描包，让指定包下的注解生效,由IOC容器统一管理 -->
14     <context:component-scan base-package="com.longg.controller"/>
15
16     <!-- 静态资源过滤，让Spring MVC不处理静态资源 -->
17     <mvc:default-servlet-handler />
18
19     <!--
20     支持mvc注解驱动
21         在spring中一般采用@RequestMapping注解来完成映射关系
22         要想使@RequestMapping注解生效
23         必须向上下文中注册DefaultAnnotationHandlerMapping和一个
24         AnnotationMethodHandlerAdapter实例
25         这两个实例分别在类级别和方法级别处理。
26         而annotation-driven配置帮助我们自动完成上述两个实例的注入。
27     -->
28     <mvc:annotation-driven />
29
30     <!-- 视图解析器 -->
31     <bean
32         class="org.springframework.web.servlet.view.InternalResourceViewResolver"
33         id="internalResourceViewResolver">
34         <!-- 前缀 -->
35         <property name="prefix" value="/WEB-INF/jsp/" />
36         <!-- 后缀 -->
37         <property name="suffix" value=".jsp" />
38     </bean>
39 </beans>

```

注：在视图解析器中我们把所有的视图都存放在/WEB-INF/目录下，这样可以保证视图安全，因为这个目录下的文件，客户端不能直接访问。

6. 创建Controller

```
1 package com.longg.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.RequestMapping;
6
7 @Controller
8 /**
9  * 如果添加在类上的请求路径则请求路径为多级: http://localhost/hello/long
10  * 如果不添加类上的请求路径则: http://localhost/long
11  */
12 //@RequestMapping("/hello")
13 public class HelloController {
14     @RequestMapping("/long")
15     public String hello(Model model){
16         // 封装数据
17         model.addAttribute("msg", "Hello,我是SpringMVC");
18         // 会自动被视图解析器处理
19         return "hello";
20     }
21 }
22 }
23 }
```

- @Controller是为了让Spring IOC容器初始化时自动扫描到;
- @RequestMapping是为了映射请求路径，这里因为类与方法上都有映射所以访问时应该是/HelloController/hello;
- 方法中声明Model类型的参数是为了把Action中的数据带到视图中;
- 方法返回的结果是视图的名称hello，加上配置文件中的前后缀变成WEB-INF/jsp/hello.jsp。

7. 创建视图层

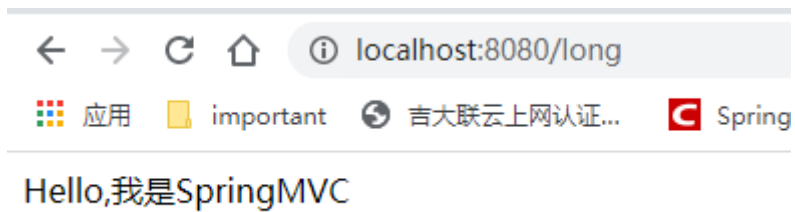
在WEB-INF/ jsp目录中创建hello.jsp，视图可以直接取出并展示从Controller带回的信息;

可以通过EL表示取出Model中存放的值，或者对象;

```
1 <%@ page contentType="text/html; charset=UTF-8" language="java" %>
2 <html>
3 <head>
4     <title>SpringMVC</title>
5 </head>
6 <body>
7     ${msg}
8 </body>
9 </html>
```

8. 配置Tomcat运行

配置Tomcat，开启服务器，访问对应的请求路径!



OK, 运行成功!

小结

实现步骤其实非常的简单:

1. 新建一个web项目;
2. 导入相关jar包;
3. 编写web.xml, 注册DispatcherServlet;
4. 编写springmvc配置文件;
5. 接下来就是去创建对应的控制类, controller;
6. 最后完善前端视图和controller之间的对应;
7. 测试运行调试;

使用springMVC必须配置的三大件:

处理器映射器、处理器适配器、视图解析器

通常, 我们只需要**手动配置视图解析器**, 而**处理器映射器**和**处理器适配器**只需要开启**注解驱动**即可, 而省去了大段的xml配置

5. Controller 配置总结

【Moudle: springmvc-04-controller】

5.1 控制器Controller

- 控制器复杂提供访问应用程序的行为, 通常通过接口定义或注解定义两种方法实现。
- 控制器负责解析用户的请求并将其转换为一个模型。
- 在Spring MVC中一个控制器类可以包含多个方法
- 在Spring MVC中, 对于Controller的配置方式有很多种

5.2 实现Controller接口

Controller是一个接口, 在org.springframework.web.servlet.mvc包下, 接口中只有一个方法;

```
1 //实现该接口的类获得控制器功能
2 public interface Controller {
3     //处理请求且返回一个模型与视图对象
4     ModelAndView handleRequest(HttpServletRequest var1, HttpServletResponse
5     var2) throws Exception;
6 }
```

测试

1. 新建一个Moudle, springmvc-04-controller。将刚才的03 拷贝一份, 我们进行操作!

- 删掉HelloController
- mvc的配置文件只留下 视图解析器!

2. 编写一个Controller类, ControllerTestInterface

```

1 // 只要实现了Controller接口的类, 说明这就是一个控制器了
2 public class ControllerTestInterface implements Controller {
3     public ModelAndView handleRequest(HttpServletRequest request,
4     HttpServletRequest response, HttpServletResponse response) throws
5     Exception {
6         ModelAndView mv = new ModelAndView();
7
8         mv.addObject("msg", "这是ControllerInterface的测试");
9         mv.setViewName("test");
10
11         return mv;
12     }
13 }

```

3. 编写完毕后, 去Spring配置文件中注册请求的bean; name对应请求路径, class对应处理请求的类

```

1 <bean name="/long1"
2   class="com.longg.controller.ControllerTestInterface"/>

```

配置文件:

```

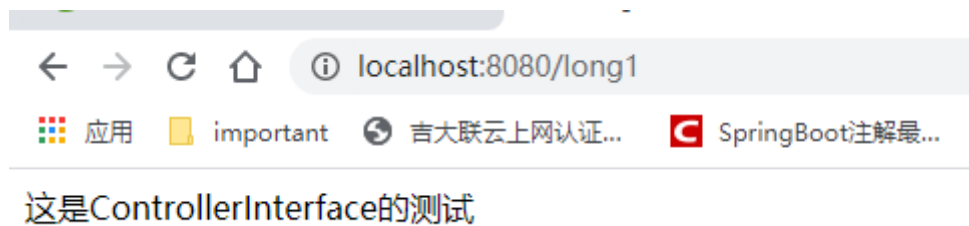
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:context="http://www.springframework.org/schema/context"
5   xmlns:mvc="http://www.springframework.org/schema/mvc"
6   xsi:schemaLocation="http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans.xsd
8     http://www.springframework.org/schema/context
9     https://www.springframework.org/schema/context/spring-context.xsd
10    http://www.springframework.org/schema/mvc
11    https://www.springframework.org/schema/mvc/spring-mvc.xsd">
12
13     <!-- 视图解析器 -->
14     <bean
15       class="org.springframework.web.servlet.view.InternalResourceViewResolver"
16       id="internalResourceViewResolver">
17       <!-- 前缀 -->
18       <property name="prefix" value="/WEB-INF/jsp/" />
19       <!-- 后缀 -->
20       <property name="suffix" value=".jsp" />
21     </bean>
22
23     <!--注册请求的bean; name对应请求路径, class对应处理请求的类-->
24     <bean name="/long1"
25       class="com.longg.controller.ControllerTestInterface"/>
26
27 </beans>

```

4. 编写前端test.jsp, 注意在WEB-INF/jsp目录下编写, 对应我们的视图解析器

```
1 <%@ page contentType="text/html;charset=UTF-8" language="java" %>
2 <html>
3 <head>
4     <title>Kuangshen</title>
5 </head>
6 <body>
7     ${msg}
8 </body>
9 </html>
```

5. 配置Tomcat运行测试, 我这里没有项目发布名配置的就是一个 / , 所以请求不用加项目名, OK!



说明:

- 实现接口Controller定义控制器是较老的办法
- 缺点是: 一个控制器中只有一个方法, 如果要多个方法则需要定义多个Controller; 定义的方式比较麻烦;

5.3 使用注解@Controller

- @Controller注解类型用于声明Spring类的实例是一个控制器 (在讲IOC时还提到了另外3个注解);
- Spring可以使用扫描机制来找到应用程序中所有基于注解的控制器类, 为了保证Spring能找到你的控制器, 需要在配置文件中声明组件扫描。

```
1 <!-- 自动扫描指定的包, 下面所有注解类交给IOC容器管理 -->
2 <context:component-scan base-package="com.longgg.controller"/>
```

- 增加一个ControllerTestAnnotation类, 使用注解实现;

```
1 @Controller // 代表这个类会被Spring接管
2 // 被这个注解的类, 里面所有的方法如果返回值为String, 并且有具体的页面可以跳转, 那么就会被视图解析器解析
3 public class ControllerTestAnnotation {
4
5     @RequestMapping(value = "/long2") // value/path/不写的效果是一样的
6     public String test(Model model){
7         model.addAttribute("msg", "测试ControllerTestAnnotation的类");
8         return "test";
9     }
10
11 }
```

- 运行tomcat测试

测试ControllerTestAnnotation的类

可以发现，我们的两个请求都可以指向一个视图，但是页面结果的结果是不一样的，从这里可以看出视图是被复用的，而控制器与视图之间是弱耦合关系。

5.4 RequestMapping

@RequestMapping

- @RequestMapping注解用于映射url到控制器类或一个特定的处理程序方法。可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。
- 只注解在方法上面

```
1 @Controller
2 public class TestController {
3     @RequestMapping("/long")
4     public String test(){
5         return "test";
6     }
7 }
```

访问路径: <http://localhost:8080/long>

- 同时注解类与方法

```
1 @Controller
2 @RequestMapping("/admin")
3 public class TestController {
4     @RequestMapping("/long")
5     public String test(){
6         return "test";
7     }
8 }
```

访问路径: <http://localhost:8080/admin/long>, 需要先指定类的路径再指定方法的路径;

6. RestFul 风格(简洁, 高效, 安全)

【Moudle: springmvc-04-controller】

6.1 概念

Restful就是一个资源定位及资源操作的风格。不是标准也不是协议，只是一种风格。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

- 功能
 - 资源：互联网所有的事物都可以被抽象为资源；
 - 资源操作：使用POST、DELETE、PUT、GET，使用不同方法对资源进行操作；

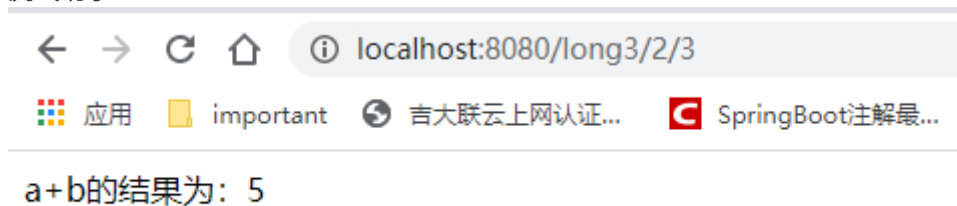
- 分别对应添加、删除、修改、查询;
- **传统方式操作资源**：通过不同的参数来实现不同的效果！方法单一，post 和 get
 - <http://127.0.0.1/item/queryItem.action?id=1> 查询,GET
 - <http://127.0.0.1/item/saveItem.action> 新增,POST
 - <http://127.0.0.1/item/updateItem.action> 更新,POST
 - <http://127.0.0.1/item/deleteItem.action?id=1> 删除,GET或POST
- **使用RESTful操作资源**：可以通过不同的请求方式来实现不同的效果！如下：请求地址一样，但是功能可以不同！
 - <http://127.0.0.1/item/1> 查询,GET
 - <http://127.0.0.1/item> 新增,POST
 - <http://127.0.0.1/item> 更新,PUT
 - <http://127.0.0.1/item/1> 删除,DELETE
- **学习测试**
 1. 新建一个类 RestFulController;
 2. 在Spring MVC中可以使用 @PathVariable 注解，让方法参数的值对应绑定到一个URI模板变量上;

```

1  @Controller
2  public class RestFulController {
3
4      /**
5       * 传统的url: http://localhost:8080/long3?a=2&b=3
6       * RestFul风格的url: http://localhost:8080/long3/a/b
7       */
8      @RequestMapping("/long3/{a}/{b}")
9      public String test(@PathVariable int a, @PathVariable int b, Model
model){
10         int res = a+b;
11         model.addAttribute("msg","a+b的结果为: " + res);
12         return "test";
13     }
14
15 }

```

3. 测试请求



4. 思考：使用路径变量的好处？

- 使路径变得更加简洁;
- 获得参数更加方便，框架会自动进行类型转换。
- 通过路径变量的类型可以约束访问参数，如果类型不一样，则访问不到对应的请求方法，如这里访问的路径是/commit/1/a，则路径与方法不匹配，而不会是参数转换失败。

使用method属性指定请求类型

用于约束请求的类型，可以收窄请求范围。指定请求谓词的类型如GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE等

我们来测试一下：

- 增加一个方法

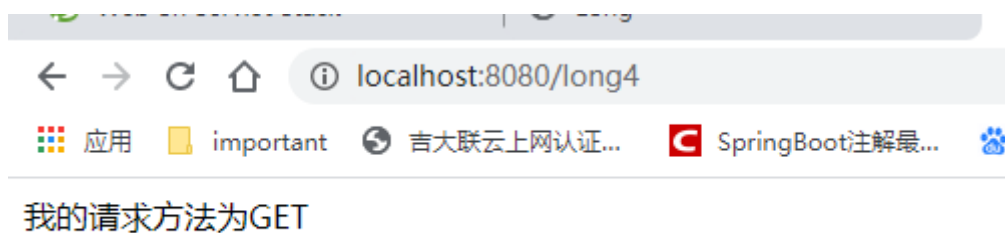
```
1 //映射访问路径,必须是POST请求
2 @RequestMapping(value = "/long4",method = RequestMethod.POST)
3 public String test2(Model model){
4     model.addAttribute("msg","我的请求方法为POST" );
5     return "test";
6 }
```

- 我们使用浏览器地址栏进行访问默认是Get请求，会报错405：



- 如果将POST修改为GET则正常了；

```
1 //映射访问路径,必须是Get请求
2 @RequestMapping(value = "/long4",method = RequestMethod.GET)
3 public String test2(Model model){
4     model.addAttribute("msg","我的请求方法为GET" );
5     return "test";
6 }
```



- 小结：

Spring MVC的 @RequestMapping 注解能够处理 HTTP 请求的方法, 比如 GET, PUT, POST, DELETE 以及 PATCH。

所有的地址栏请求默认都会是 HTTP GET 类型的。

方法级别的注解变体有如下几个：组合注解

```
1 @GetMapping
2 @PostMapping
3 @PutMapping
4 @DeleteMapping
5 @PatchMapping
```

7. SpringMVC：结果跳转三种方式(转发、重定向)

【Moudle: springmvc-04-controller】

7.1 ModelAndView

- 设置ModelAndView对象，根据view的名称，和视图解析器跳到指定的页面。

页面：{视图解析器前缀} + viewName + {视图解析器后缀}

```
1 <!-- 视图解析器 -->
2 <bean
3     class="org.springframework.web.servlet.view.InternalResourceViewResolver"
4     id="internalResourceViewResolver">
5     <!-- 前缀 -->
6     <property name="prefix" value="/WEB-INF/jsp/" />
7     <!-- 后缀 -->
8     <property name="suffix" value=".jsp" />
9 </bean>
```

对应的controller类

```
1 public class ControllerTestInterface implements Controller {
2     public ModelAndView handleRequest(HttpServletRequest httpServletRequest,
3     HttpServletResponse httpServletResponse) throws Exception {
4         ModelAndView mv = new ModelAndView();
5
6         mv.addObject("msg", "这是ControllerInterface的测试");
7         mv.setViewName("test");
8
9         return mv;
10    }
11 }
```

7.2 ServletAPI

- 通过设置ServletAPI，不需要视图解析器。

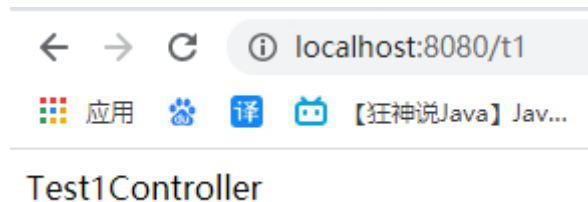
1. 通过HttpServletResponse进行输出
2. 通过HttpServletResponse实现重定向
3. 通过HttpServletResponse实现转发

```
1 @Controller
2 public class ResultGo {
3
4     @RequestMapping("/result/t1")
```

```

5     public void test1(HttpServletRequest req, HttpServletResponse rsp)
throws IOException {
6         rsp.getWriter().println("Hello, Spring BY servlet API");
7     }
8
9     @RequestMapping("/result/t2")
10    public void test2(HttpServletRequest req, HttpServletResponse rsp)
throws IOException {
11        rsp.sendRedirect("/index.jsp");
12    }
13
14    @RequestMapping("/result/t3")
15    public void test3(HttpServletRequest req, HttpServletResponse rsp)
throws Exception {
16        //转发
17        req.setAttribute("msg", "/result/t3");
18        req.getRequestDispatcher("/WEB-INF/jsp/test.jsp").forward(req, rsp);
19    }
20
21 }

```



7.3 SpringMVC

通过SpringMVC来实现转发和重定向 - 无需视图解析器;

测试前, 需要将视图解析器注释掉

```

1  @Controller
2  public class RedirectController {
3
4      /**
5       * 没有视图解析器的情况
6       */
7      @RequestMapping("/rc/t1")
8      public String test1(){
9          //转发一
10         return "/WEB-INF/jsp/hello.jsp";
11     }
12
13     @RequestMapping("/rc/t2")
14     public String test2(){
15         //转发二 (不建议使用)
16         return "forward:/WEB-INF/jsp/hello.jsp";
17     }
18
19     @RequestMapping("/rc/t3")
20     public String test3(){
21         //重定向
22         return "redirect:/index.jsp";
23     }
24 }

```

通过SpringMVC来实现转发和重定向 - 有视图解析器;

重定向, 不需要视图解析器, 本质就是重新请求一个新地方嘛, 所以注意路径问题.

可以重定向到另外一个请求实现.

```

1  @Controller
2  public class RedirectController {
3
4      /**
5       * 有视图解析器的情况
6       */
7      @RequestMapping("/rc/t4")
8      public String test4(){
9          //重定向 (重定向不能访问WEB-INF路径下的文件)
10         return "redirect:/index.jsp";
11     }
12 }

```

8. 数据处理（接收请求参数及数据回显）

8.1 处理提交数据

1、提交的域名称和处理方法的参数名一致

提交数据: <http://localhost:8080/hello?name=kuangshen>

处理方法:

```

1  @RequestMapping("/hello")
2  public String hello(String name){
3      System.out.println(name);
4      return "hello";
5  }

```

后台输出: kuangshen

2、提交的域名称和处理方法的参数名不一致

提交数据: <http://localhost:8080/hello?username=kuangshen>

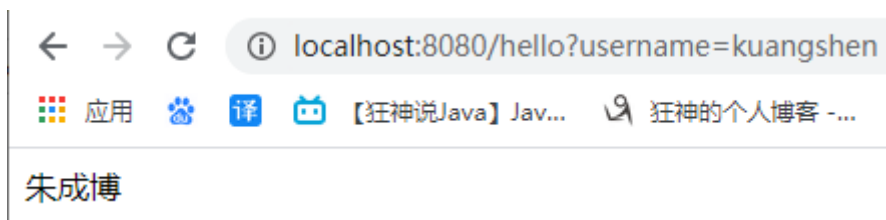
处理方法:

```

1  // @RequestParam("username") : username提交的域的名称 .
2  @RequestMapping("/hello")
3  public String hello(@RequestParam("username") String name){
4      System.out.println(name);
5      return "hello";
6  }

```

后台输出: kuangshen



3. 提交的是一个对象

要求提交的表单域和对象的属性名一致，参数使用对象即可

1. 实体类

```
1 public class User {
2     private int id;
3     private String name;
4     private int age;
5     //构造
6     //get/set
7     //toString()
8 }
```

2. 提交数据：<http://localhost:8080/user?name=kuangshen&id=1&age=15>

3. 处理方法：

```
1 @RequestMapping("/user")
2 public String user(User user){
3     System.out.println(user);
4     return "hello";
5 }
```

后台输出：User { id=1, name='kuangshen', age=15 }

说明：如果使用对象的话，前端传递的参数名和对象名必须一致，否则就是null。

8.2 数据显示到前端

第一种：通过ModelAndView

我们前面一直都是如此，就不过多解释

```
1 public class ControllerTest1 implements Controller {
2
3     public ModelAndView handleRequest(HttpServletRequest httpServletRequest,
4     HttpServletResponse httpServletResponse) throws Exception {
5         //返回一个模型视图对象
6         ModelAndView mv = new ModelAndView();
7         mv.addObject("msg", "ControllerTest1");
8         mv.setViewName("test");
9         return mv;
10    }
```

第二种：通过ModelMap

ModelMap

```

1 @RequestMapping("/hello")
2 public String hello(@RequestParam("username") String name, ModelMap model){
3     //封装要显示到视图中的数据
4     //相当于req.setAttribute("name",name);
5     model.addAttribute("name",name);
6     System.out.println(name);
7     return "hello";
8 }

```

第三种：通过Model

Model

```

1 @RequestMapping("/ct2/hello")
2 public String hello(@RequestParam("username") String name, Model model){
3     //封装要显示到视图中的数据
4     //相当于req.setAttribute("name",name);
5     model.addAttribute("msg",name);
6     System.out.println(name);
7     return "test";
8 }

```

8.3 对比

就对于新手而言简单来说使用区别就是：

- 1 Model 精简版，只有寥寥几个方法只适用于储存数据，简化了新手对于Model对象的操作和理解；
- 2
- 3 ModelMap 继承了 LinkedHashMap，除了实现了自身的一些方法，同样的继承 LinkedHashMap 的方法和特性；
- 4
- 5 ModelAndView 可以在储存数据的同时，可以进行设置返回的逻辑视图，进行控制展示层的跳转。

当然更多的以后开发考虑的更多的是性能和优化，就不能单单仅限于此的了解。

请使用80%的时间打好扎实的基础，剩下18%的时间研究框架，2%的时间去学点英文，框架的官方文档永远是最好的教程。

9. 数据处理（乱码问题）

【Moudle: springmvc-04-controller】

测试步骤：

1. 我们可以在首页编写一个提交的表单

```

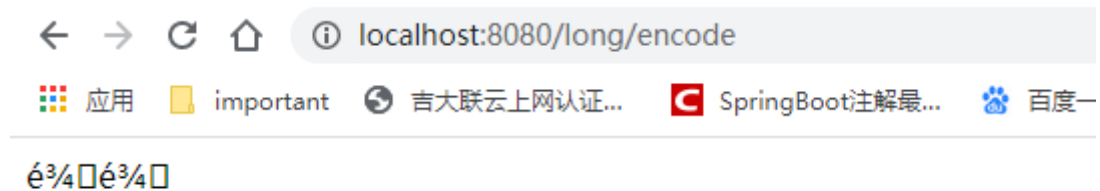
1 <form action="/long/encode" method="post">
2     <input type="text" name="name">
3     <input type="submit">
4 </form>

```

2. 后台编写对应的处理类

```
1 @Controller
2 public class EncodingController {
3
4     @RequestMapping("/long/encode")
5     public String test(Model model, String name){
6         System.out.println(name);
7         model.addAttribute("msg",name); //获取表单提交的值
8         return "test"; //跳转到test页面显示输入的值
9     }
10 }
```

3. 输入中文测试，发现乱码



不得不说，乱码问题是在我们开发中十分常见的问题，也是让我们程序猿比较头大的问题！

9.1 处理方法一：自定义过滤器

自定义一个过滤器:

```
1 public class EncodingFilter implements Filter {
2
3     public void init(FilterConfig filterConfig) throws ServletException {
4
5     }
6
7     public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException,
ServletException {
8         servletRequest.setCharacterEncoding("utf-8");
9         servletResponse.setCharacterEncoding("utf-8");
10
11         filterChain.doFilter(servletRequest,servletResponse);
12     }
13
14     public void destroy() {
15
16     }
17 }
```

在web.xml中配置我们的过滤器:

```

1  <!--自定义的乱码过滤器-->
2  <filter>
3      <filter-name>encoding</filter-name>
4      <filter-class>com.longg.filter.EncodingFilter</filter-class>
5  </filter>
6
7  <filter-mapping>
8      <filter-name>encoding</filter-name>
9      <url-pattern>/*</url-pattern>
10 </filter-mapping>

```

9.2 处理方法二：SpringMVC提供的过滤器

SpringMVC给我们提供了一个过滤器，可以在web.xml中配置。

修改了xml文件需要重启服务器！

```

1  <!--2.配置SpringMVC的乱码过滤-->
2  <filter>
3      <filter-name>encoding</filter-name>
4      <filter-
5      class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
6      <init-param>
7          <param-name>encoding</param-name>
8          <param-value>utf-8</param-value>
9      </init-param>
10 </filter>
11 <filter-mapping>
12     <filter-name>encoding</filter-name>
13     <url-pattern>/*</url-pattern>

```

9.3 处理方法三：终极过滤器

有些极端情况下.这个过滤器对get的支持不好。

处理方法：

1. 修改tomcat配置文件： 设置编码！

```

1  <Connector URIEncoding="utf-8" port="8080" protocol="HTTP/1.1"
2      connectionTimeout="20000"
3      redirectPort="8443" />

```

2. 自定义过滤器

```

1  package com.kuang.filter;
2
3  import javax.servlet.*;
4  import javax.servlet.http.HttpServletRequest;
5  import javax.servlet.http.HttpServletRequestWrapper;
6  import javax.servlet.http.HttpServletResponse;
7  import java.io.IOException;
8  import java.io.UnsupportedEncodingException;
9  import java.util.Map;
10

```

```

11  /**
12   * 解决get和post请求 全部乱码的过滤器
13   */
14  public class GenericEncodingFilter implements Filter {
15
16      @Override
17      public void destroy() {
18      }
19
20      @Override
21      public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
22          //处理response的字符编码
23          HttpServletResponse myResponse=(HttpServletResponse) response;
24          myResponse.setContentType("text/html;charset=UTF-8");
25
26          // 转型为与协议相关对象
27          HttpServletRequest httpRequest = (HttpServletRequest)
request;
28          // 对request包装增强
29          MyRequest myrequest = new
MyRequest(httpRequest);
30          chain.doFilter(myrequest, response);
31      }
32
33      @Override
34      public void init(FilterConfig filterConfig) throws
ServletException {
35      }
36
37  }
38
39  //自定义request对象, HttpServletRequest的包装类
40  class MyRequest extends HttpServletRequestWrapper {
41
42      private HttpServletRequest request;
43      //是否编码的标记
44      private boolean hasEncode;
45      //定义一个可以传入HttpServletRequest对象的构造函数, 以便对其进行装饰
46      public MyRequest(HttpServletRequest request) {
47          super(request); // super必须写
48          this.request = request;
49      }
50
51      // 对需要增强方法 进行覆盖
52      @Override
53      public Map getParameterMap() {
54          // 先获得请求方式
55          String method = request.getMethod();
56          if (method.equalsIgnoreCase("post")) {
57              // post请求
58              try {
59                  // 处理post乱码
60                  request.setCharacterEncoding("utf-8");
61                  return request.getParameterMap();
62              } catch (UnsupportedEncodingException e) {
63                  e.printStackTrace();
64              }

```

```

65         } else if (method.equalsIgnoreCase("get")) {
66             // get请求
67             Map<String, String[]> parameterMap =
request.getParameterMap();
68             if (!hasEncode) { // 确保get手动编码逻辑只运行一次
69                 for (String parameterName : parameterMap.keySet()) {
70                     String[] values = parameterMap.get(parameterName);
71                     if (values != null) {
72                         for (int i = 0; i < values.length; i++) {
73                             try {
74                                 // 处理get乱码
75                                 values[i] = new String(values[i]
76                                     .getBytes("ISO-8859-1"), "utf-
8");
77                             } catch (UnsupportedEncodingException e) {
78                                 e.printStackTrace();
79                             }
80                         }
81                     }
82                 }
83                 hasEncode = true;
84             }
85             return parameterMap;
86         }
87         return super.getParameterMap();
88     }
89
90     //取一个值
91     @Override
92     public String getParameter(String name) {
93         Map<String, String[]> parameterMap = getParameterMap();
94         String[] values = parameterMap.get(name);
95         if (values == null) {
96             return null;
97         }
98         return values[0]; // 取回参数的第一个值
99     }
100
101     //取所有值
102     @Override
103     public String[] getParameterValues(String name) {
104         Map<String, String[]> parameterMap = getParameterMap();
105         String[] values = parameterMap.get(name);
106         return values;
107     }
108 }

```

这个也是我在网上找的一些大神写的，一般情况下，SpringMVC默认的乱码处理就已经能够很好的解决了！

然后在web.xml中配置这个过滤器即可！

乱码问题，需要平时多注意，在尽可能能设置编码的地方，都设置为统一编码 UTF-8！

10. JSON讲解

【Moudle: springmvc-05-json】

10.1 什么是JSON

- JSON (JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式，目前使用特别广泛。
- 采用完全独立于编程语言的**文本格式**来存储和表示数据。
- 简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。
- 易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。

在 JavaScript 语言中，一切都是对象。因此，任何JavaScript 支持的类型都可以通过 JSON 来表示，例如字符串、数字、对象、数组等。看看他的要求和语法格式：

- 对象表示为键值对，数据由逗号分隔
- 花括号保存对象
- 方括号保存数组

JSON 键值对是用来保存 JavaScript 对象的一种方式，和 JavaScript 对象的写法也大同小异，键/值对组合中的键名写在前面并用双引号 "" 包裹，使用冒号 : 分隔，然后紧接着值：

```
1 {"name": "long"}
2 {"age": "16"}
3 {"sex": "男"}
```

很多人搞不清楚 JSON 和 JavaScript 对象的关系，甚至连谁是谁都不清楚。其实，可以这么理解：

- JSON 是 JavaScript 对象的字符串表示法，它使用文本表示一个 JS 对象的信息，本质是一个字符串。

```
1 var obj = {a: 'Hello', b: 'world'};           //这是一个对象，注意键名也是可以使
    用引号包裹的
2 var json = '{"a": "Hello", "b": "world"}';    //这是一个 JSON 字符串，本质是一
    个字符串
```

JSON 和 JavaScript 对象互转

- 要实现从JSON字符串转换为JavaScript 对象，使用 JSON.parse() 方法：

```
1 var obj = JSON.parse('{"a": "Hello", "b": "world"}');
2 //结果是 {a: 'Hello', b: 'world'}
```

- 要实现从JavaScript 对象转换为JSON字符串，使用 JSON.stringify() 方法：

```
1 var json = JSON.stringify({a: 'Hello', b: 'world'});
2 //结果是 '{"a": "Hello", "b": "world"}'
```

代码测试

1. 新建一个module，springmvc-05-json，添加web的支持
2. 在web目录下新建一个 jsonTest.html，编写测试内容

```
1 <!DOCTYPE html>
```

```

2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Json</title>
6
7      <script type="text/javascript">
8          //编写一个js的对象
9          var user = {
10              name: "long",
11              age: 16,
12              sex: "girl"
13          };
14
15          //将js对象转换成json字符串
16          var str = JSON.stringify(user);
17          console.log(str);
18
19          //将json字符串转换为js对象
20          var obj = JSON.parse(str);
21          console.log(obj);
22
23      </script>
24
25  </head>
26  <body>
27
28  </body>
29  </html>

```

3. 在IDEA中使用浏览器打开，查看控制台输出！



10.2 Controller返回JSON数据

10.2.1 Jackson

- 测试步骤

1. Jackson应该是目前比较好的 json 解析工具了。
2. 当然工具不止这一个，比如还有阿里巴巴的 fastjson 等等。
3. 我们这里使用Jackson，使用它需要导入它的jar包；


```

1 <!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-
  core -->
2 <dependency>
3     <groupId>com.fasterxml.jackson.core</groupId>
4     <artifactId>jackson-databind</artifactId>
5     <version>2.10.0</version>
6 </dependency>

```

4. 配置SpringMVC需要的配置

web.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
5     version="4.0">
6
7     <!--1.注册servlet-->
8     <servlet>
9         <servlet-name>SpringMVC</servlet-name>
10        <servlet-
  class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
11        <!--通过初始化参数指定SpringMVC配置文件的位置，进行关联-->
12        <init-param>
13            <param-name>contextConfigLocation</param-name>
14            <param-value>classpath:springmvc-servlet.xml</param-value>
15        </init-param>
16        <!-- 启动顺序，数字越小，启动越早 -->
17        <load-on-startup>1</load-on-startup>
18    </servlet>
19
20    <!--所有请求都会被springmvc拦截 -->
21    <servlet-mapping>
22        <servlet-name>SpringMVC</servlet-name>
23        <url-pattern>/</url-pattern>
24    </servlet-mapping>
25
26    <filter>
27        <filter-name>encoding</filter-name>
28        <filter-
  class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
29        <init-param>
30            <param-name>encoding</param-name>
31            <param-value>utf-8</param-value>
32        </init-param>
33    </filter>
34    <filter-mapping>
35        <filter-name>encoding</filter-name>
36        <url-pattern>/*</url-pattern>
37    </filter-mapping>
38
39 </web-app>

```

springmvc-servlet.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xmlns:mvc="http://www.springframework.org/schema/mvc"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7                           http://www.springframework.org/schema/beans/spring-beans.xsd
8                           http://www.springframework.org/schema/context
9                           https://www.springframework.org/schema/context/spring-context.xsd
10                          http://www.springframework.org/schema/mvc
11                          https://www.springframework.org/schema/mvc/spring-mvc.xsd">
12
13     <!-- 自动扫描指定的包，下面所有注解类交给IOC容器管理 -->
14     <context:component-scan base-package="com.kuang.controller"/>
15
16     <!-- 视图解析器 -->
17     <bean
18 class="org.springframework.web.servlet.view.InternalResourceViewResolver"
19       id="internalResourceViewResolver">
20       <!-- 前缀 -->
21       <property name="prefix" value="/WEB-INF/jsp/" />
22       <!-- 后缀 -->
23       <property name="suffix" value=".jsp" />
24     </bean>
25 </beans>

```

5. 我们随便编写一个User的实体类，然后我们去编写我们的测试Controller;

```

1 @Data
2 @AllArgsConstructor
3 @NoArgsConstructor
4 public class User {
5
6     private String name;
7     private int age;
8     private String sex;
9
10 }

```

6. 这里我们需要两个新东西，一个是@ResponseBody，一个是ObjectMapper对象，我们看下具体的用法，编写一个Controller;

```

1 @Controller
2 // @RestController // 该注解下的类中的所有方法都不走视图解析器，只返回字符串，等同于
3 // @ResponseBody
4 public class JacksonController {
5
6     @RequestMapping("/json1")
7     @ResponseBody // 使用该注解使该方法不走视图解析器，直接返回一个字符串
8     public String json1() throws JSONException {
9         // 创建一个jackson的对象映射器，用来解析数据
10        ObjectMapper mapper = new ObjectMapper();
11        // 创建一个对象
12        User user = new User("long", 16, "girl");
13        // 将我们的对象解析成为json格式

```

```

13     String str = mapper.writeValueAsString(user);
14     //由于@ResponseBody注解，这里会将str转成json格式返回；十分方便
15     return str;
16 }
17 }

```

7. 配置Tomcat，启动测试一下！



8. 发现出现了乱码问题，我们需要设置一下他的编码格式为utf-8，以及它返回的类型；通过 @RequestMapping的produces属性来实现，修改下代码：

```

1 //produces:指定响应体返回类型和编码
2 @RequestMapping(value = "/json1", produces = "application/json;charset=utf-8")

```

9. 再次测试， <http://localhost:8080/json1>，乱码问题OK！

• 返回json字符串统一解决

在类上直接使用 @RestController，这样子，里面所有的方法都只会返回 json 字符串了，不用再每一个都添加@ResponseBody！我们在前后端分离开发中，一般都使用 @RestController，十分便捷！

```

1 @RestController // 该注解下的类中的所有方法都不走视图解析器，只返回字符串，等同于
  @ResponseBody
2 public class JacksonController {
3
4     @RequestMapping("/json1")
5     public String json1() throws JsonProcessingException {
6         //创建一个jackson的对象映射器，用来解析数据
7         ObjectMapper mapper = new ObjectMapper();
8         //创建一个对象
9         User user = new User("long", 16, "girl");
10        //将我们的对象解析成为json格式
11        String str = mapper.writeValueAsString(user);
12        //由于@ResponseBody注解，这里会将str转成json格式返回；十分方便
13        return str;
14    }
15 }

```

启动tomcat测试，结果都正常输出！

• 测试集合输出

增加一个新的方法

```

1 @RequestMapping("/json2")
2 @ResponseBody
3 public String json2() throws JsonProcessingException {
4

```

```

5 //创建一个jackson的对象映射器，用来解析数据
6 ObjectMapper mapper = new ObjectMapper();
7 //创建一个对象
8 User user1 = new User("long1", 16, "girl");
9 User user2 = new User("long2", 16, "girl");
10 User user3 = new User("long3", 16, "girl");
11 User user4 = new User("long4", 16, "girl");
12 List<User> list = new ArrayList<User>();
13 list.add(user1);
14 list.add(user2);
15 list.add(user3);
16 list.add(user4);
17
18 //将我们的对象解析成为json格式
19 String str = mapper.writeValueAsString(list);
20 return str;
21 }

```

运行结果：



```

[{"name":"long1","age":16,"sex":"girl"},{"name":"long2","age":16,"sex":"girl"},{"name":"long3","age":16,"sex":"girl"},{"name":"long4","age":16,"sex":"girl"}]

```

• 输出时间对象

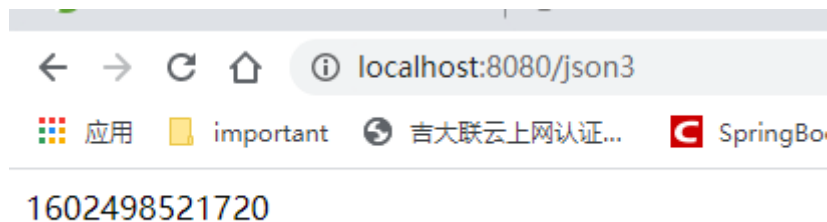
增加一个新的方法

```

1 @RequestMapping("/json3")
2 public String json3() throws JsonProcessingException {
3
4     ObjectMapper mapper = new ObjectMapper();
5
6     //创建时间一个对象， java.util.Date
7     Date date = new Date();
8
9     //将我们的对象解析成为json格式
10    String str = mapper.writeValueAsString(date);
11    return str;
12 }

```

运行结果：默认日期格式会变成一个数字，是1970年1月1日到当前日期的毫秒数！Jackson 默认是会把时间转成timestamps形式



```

1602498521720

```

解决方案一：取消timestamps形式， 自定义时间格式

```

1 @RequestMapping("/json4")
2 public String json4() throws JsonProcessingException {
3

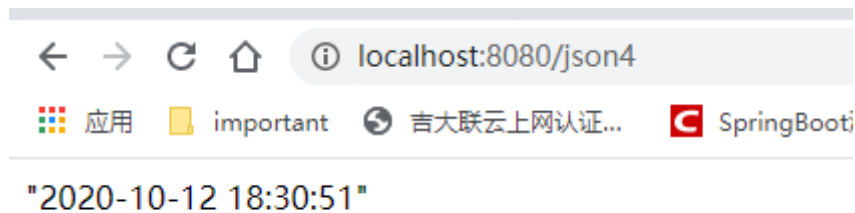
```

```

4      ObjectMapper mapper = new ObjectMapper();
5
6      //不使用时间戳的方式
7      mapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);
8      //自定义日期格式对象
9      SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
10     //指定日期格式
11     mapper.setDateFormat(sdf);
12
13     Date date = new Date();
14     String str = mapper.writeValueAsString(date);
15
16     return str;
17 }

```

运行结果：



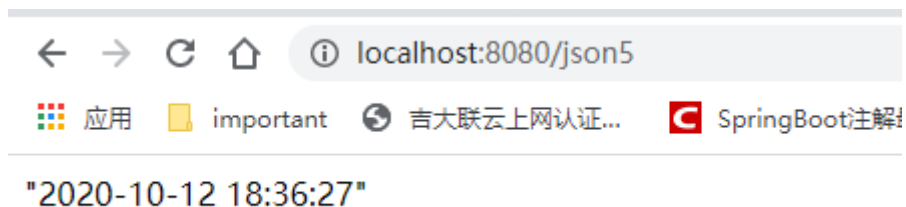
解决方案一：利用纯 Java 的形式自定义时间格式

```

1  @RequestMapping("/json5")
2  @ResponseBody
3  public String json5() throws JsonProcessingException {
4
5      ObjectMapper mapper = new ObjectMapper();
6
7      Date date = new Date();
8      //自定义日期的格式
9      SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
10
11     return mapper.writeValueAsString(sdf.format(date));
12 }

```

运行结果：



• 抽取为工具类

如果要经常使用的话，这样是比较麻烦的，我们可以将这些代码封装到一个工具类中；我们去编写下

```

1  public class JsonUtils {
2
3      public static String getJson(Object object) {

```

```

4         return getJson(object,"yyyy-MM-dd HH:mm:ss");
5     }
6
7     public static String getJson(Object object,String dateFormat) {
8         ObjectMapper mapper = new ObjectMapper();
9
10        //不使用时间差的方式
11        mapper.configure(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS,
12        false);
13        //自定义日期格式对象
14        SimpleDateFormat sdf = new SimpleDateFormat(dateFormat);
15        //指定日期格式
16        mapper.setDateFormat(sdf);
17        try {
18            return mapper.writeValueAsString(object);
19        } catch (JsonProcessingException e) {
20            e.printStackTrace();
21        }
22        return null;
23    }

```

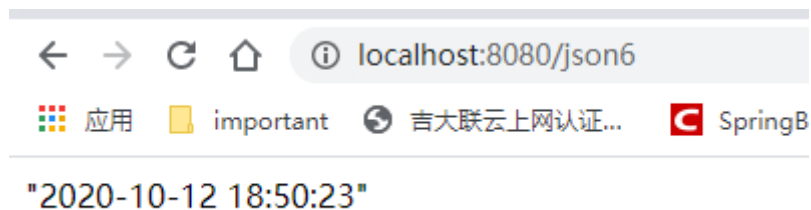
使用工具类:

```

1 @RequestMapping("/json6")
2 @ResponseBody
3 public String json6() throws JsonProcessingException {
4     Date date = new Date();
5     //return JsonUtils.getJson(date);
6     return JsonUtils.getJson(date,"yyyy-MM-dd HH:mm:ss");
7 }

```

运行结果:



```

1 @RequestMapping("/json7")
2 @ResponseBody // 使用该注解使该方法不走视图解析器，直接返回一个字符串，等同于
3 @RestController
4 public String json7() throws JsonProcessingException {
5     User user = new User("long", 16, "girl");
6     return JsonUtils.getJson(user);
7 }
8
9 @RequestMapping("/json8")
10 @ResponseBody
11 public String json8(){
12     User user1 = new User("long1", 16, "girl");
13     User user2 = new User("long2", 16, "girl");
14     User user3 = new User("long3", 16, "girl");
15     User user4 = new User("long4", 16, "girl");

```

```

15     List<User> list = new ArrayList<User>();
16     list.add(user1);
17     list.add(user2);
18     list.add(user3);
19     list.add(user4);
20     return JsonUtils.getJson(list);
21 }

```

← → ↺ 🏠 ⓘ localhost:8080/json7

应用 important 吉大联云上网认证... Sprir

{"name":"long","age":16,"sex":"girl"}

← → ↺ 🏠 ⓘ localhost:8080/json8

应用 important 吉大联云上网认证... SpringBoot注解最... 百度一下，你就知道 格力董明珠店-格力...

[{"name":"long1","age":16,"sex":"girl"}, {"name":"long2","age":16,"sex":"girl"}, {"name":"long3","age":16,"sex":"girl"}, {"name":"long4","age":16,"sex":"girl"}]

10.2.2 Fastjson

fastjson.jar是阿里开发的一款专门用于Java开发的包，可以方便的实现json对象与JavaBean对象的转换，实现JavaBean对象与json字符串的转换，实现json对象与json字符串的转换。实现json的转换方法很多，最后的实现结果都是一样的。

fastjson 的 pom依赖！

```

1 <dependency>
2     <groupId>com.alibaba</groupId>
3     <artifactId>fastjson</artifactId>
4     <version>1.2.60</version>
5 </dependency>

```

fastjson 三个主要的类：

- 【JSONObject 代表 json 对象】
 - JSONObject实现了Map接口, 猜想 JSONObject底层操作是由Map实现的。
 - JSONObject对应json对象，通过各种形式的get()方法可以获取json对象中的数据，也可利用诸如size(), isEmpty()等方法获取"键：值"对的个数和判断是否为空。其本质是通过实现Map接口并调用接口中的方法完成的。
- 【JSONArray 代表 json 对象数组】
 - 内部是有List接口中的方法来完成操作的。
- 【JSON 代表 JSONObject和JSONArray的转化】
 - JSON类源码分析与使用
 - 仔细观察这些方法，主要是实现json对象，json对象数组，javabean对象，json字符串之间的相互转化。

代码测试，我们新建一个FastJsonDemo 类

```

1 public class FastJsonDemo {
2     public static void main(String[] args) {
3         //创建一个对象
4         User user1 = new User("long1", 16, "girl");
5         User user2 = new User("long2", 16, "girl");
6         User user3 = new User("long3", 16, "girl");

```

```

7      User user4 = new User("long4", 16, "girl");
8      List<User> list = new ArrayList<User>();
9      list.add(user1);
10     list.add(user2);
11     list.add(user3);
12     list.add(user4);
13
14     System.out.println("*****Java对象 转 JSON字符串*****");
15     String str1 = JSON.toJSONString(list);
16     System.out.println("JSON.toJSONString(list)==> "+str1);
17     String str2 = JSON.toJSONString(user1);
18     System.out.println("JSON.toJSONString(user1)==> "+str2);
19
20     System.out.println("\n***** JSON字符串 转 Java对象*****");
21     User jp_user1=JSON.parseObject(str2,User.class);
22     System.out.println("JSON.parseObject(str2,User.class)==>
"+jp_user1);
23
24     System.out.println("\n***** Java对象 转 JSON对象 *****");
25     JSONObject jsonObject = (JSONObject) JSON.toJSON(user2);
26     System.out.println("JSON.toJSON(user2)==> "+jsonObject);
27     System.out.println("(JSONObject) JSON.toJSON(user2)==>
"+jsonObject.getString("sex"));
28
29     System.out.println("\n***** JSON对象 转 Java对象 *****");
30     User to_java_user = JSON.toJavaObject(jsonObject, User.class);
31     System.out.println("JSON.toJavaObject(jsonObject1, User.class)==>
"+to_java_user);
32 }
33 }

```

例题:

```

1  @Controller
2  public class FastJsonController {
3      @RequestMapping("/fjson1")
4      @ResponseBody    // 使用该注解使该方法不走视图解析器，直接返回一个字符串，等同于
@RestController
5      public String fjson1(){
6          User user = new User("long", 16, "girl");
7          return JSON.toJSONString(user);
8      }
9
10     @RequestMapping("/fjson2")
11     @ResponseBody
12     public String fjson2() throws JsonProcessingException {
13         User user1 = new User("long1", 16, "girl");
14         User user2 = new User("long2", 16, "girl");
15         User user3 = new User("long3", 16, "girl");
16         User user4 = new User("long4", 16, "girl");
17         List<User> list = new ArrayList<User>();
18         list.add(user1);
19         list.add(user2);
20         list.add(user3);
21         list.add(user4);
22         return JSON.toJSONString(list);
23     }

```



```
24 }
25
```

测试:



这种工具类，我们只需要掌握使用就好了，在使用的时候在根据具体的业务去找对应的实现。

11. Ajax技术

【Module: springmvc-06-ajax】

11.1 AJAX初体验

异步可理解为局部刷新，同步指需要按部就班地完成一整套流程

- **AJAX = Asynchronous JavaScript and XML (异步的 JavaScript 和 XML) 。**
- AJAX 是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。
- **Ajax 不是一种新的编程语言，而是一种用于创建更好更快以及交互性更强的Web应用程序的技术。**
- 在 2005 年，Google 通过其 Google Suggest 使 AJAX 变得流行起来。Google Suggest能够自动帮你完成搜索单词。
- Google Suggest 使用 AJAX 创造出动态性极强的 web 界面：当您在谷歌的搜索框输入关键字时，JavaScript 会把这些字符发送到服务器，然后服务器会返回一个搜索建议的列表。
- 传统的网页(即不用ajax技术的网页)，想要更新内容或者提交一个表单，都需要重新加载整个网页。使用ajax技术的网页，通过在后台服务器进行少量的数据交换，就可以实现异步局部更新。
- 使用Ajax，用户可以创建接近本地桌面应用的直接、高可用、更丰富、更动态的Web用户界面。
- **伪造Ajax**

我们可以使用前端的一个标签来伪造一个ajax的样子。iframe标签

1. 新建一个module：springmvc-06-ajax，导入web支持！
2. 编写一个test.html 使用 iframe 测试，感受下效果

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>iFrame体验页面无刷新</title>
6 </head>
```

```

7   <body>
8       <script type="text/javascript">
9           function LoadPage(){
10               var targetUrl = document.getElementById('url').value;
11               console.log(targetUrl);
12               document.getElementById("iframePosition").src = targetUrl;
13           }
14       </script>
15
16       <div>
17           <p>请输入要加载的地址: </p>
18           <p>
19               <input id="url" type="text" value="https://www.baidu.com/" />
20               <input type="button" value="提交" onclick="LoadPage()">
21           </p>
22       </div>
23
24       <div>
25           <h3>加载页面位置: </h3>
26           <iframe id="iframePosition" style="width:100%; height:500px;">
27       </div>
28 </body>
29 </html>

```

3. 使用IDEA开浏览器测试一下!

利用AJAX可以做:

- 注册时, 输入用户名自动检测用户是否已经存在。
- 登陆时, 提示用户名密码错误
- 删除数据行时, 将行ID发送到后台, 后台在数据库中删除, 数据库删除成功后, 在页面DOM中将数据行也删除。
-等等

11.2 jQuery.ajax

- 纯JS原生实现Ajax我们不去讲解, 直接使用jquery提供的, 方便学习和使用, 避免重复造轮子, 有兴趣的同学可以去了解下JS原生XMLHttpRequest !
- Ajax的核心是XMLHttpRequest对象(XHR)。XHR为向服务器发送请求和解析服务器响应提供了接口。能够以异步方式从服务器获取新数据。
- jQuery 提供多个与 AJAX 有关的方法。
- 通过 jQuery AJAX 方法, 您能够使用 HTTP Get 和 HTTP Post 从远程服务器上请求文本、HTML、XML 或 JSON 同时您能够把这些外部数据直接载入网页的被选元素中。
- jQuery 不是生产者, 而是大自然搬运工。
- jQuery Ajax本质就是 XMLHttpRequest, 对他进行了封装, 方便调用!

```

1   jQuery.ajax(...)
2       部分参数:
3           url: 请求地址【重要】
4           type: 请求方式, GET、POST (1.9.0之后用method)【重要】
5           headers: 请求头
6           data: 要发送的数据【重要】
7           contentType: 即将发送信息至服务器的内容编码类型(默认: "application/x-www-
8                       form-urlencoded; charset=UTF-8")
9           async: 是否异步

```

```

9         timeout: 设置请求超时时间（毫秒）
10    beforeSend: 发送请求前执行的函数(全局)
11    complete: 完成之后执行的回调函数(全局)
12    success: 成功之后执行的回调函数(全局)【重要】
13    error: 失败之后执行的回调函数(全局)【重要】
14    accepts: 通过请求头发送给服务器，告诉服务器当前客户端课接受的数据类型
15    dataType: 将服务器端返回的数据转换成指定类型
16        "xml": 将服务器端返回的内容转换成xml格式
17        "text": 将服务器端返回的内容转换成普通文本格式
18        "html": 将服务器端返回的内容转换成普通文本格式，在插入DOM中时，如果包含
JavaScript标签，则会尝试去执行。
19    "script": 尝试将返回值当作JavaScript去执行，然后再将服务器端返回的内容转换成
普通文本格式
20    "json": 将服务器端返回的内容转换成相应的JavaScript对象
21    "jsonp": JSONP 格式使用 JSONP 形式调用函数时，如 "myurl?callback=?"
jQuery 将自动替换 ? 为正确的函数名，以执行回调函数

```

我们来个简单的测试，使用最原始的HttpServletResponse处理

1. 配置web.xml 和 springmvc的配置文件，复制上面案例的即可【记得静态资源过滤和注解驱动配置上】

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:context="http://www.springframework.org/schema/context"
5         xmlns:mvc="http://www.springframework.org/schema/mvc"
6         xsi:schemaLocation="http://www.springframework.org/schema/beans
7                             http://www.springframework.org/schema/beans/spring-beans.xsd
8                             http://www.springframework.org/schema/context
9                             https://www.springframework.org/schema/context/spring-
context.xsd
10                            http://www.springframework.org/schema/mvc
11                            https://www.springframework.org/schema/mvc/spring-mvc.xsd">
12
13     <!-- 自动扫描指定的包，下面所有注解类交给IOC容器管理 -->
14     <context:component-scan base-package="com.longg.controller"/>
15     <mvc:default-servlet-handler />
16     <mvc:annotation-driven />
17
18     <!-- 视图解析器 -->
19     <bean
20         class="org.springframework.web.servlet.view.InternalResourceViewResolver"
21         id="internalResourceViewResolver">
22         <!-- 前缀 -->
23         <property name="prefix" value="/WEB-INF/jsp/" />
24         <!-- 后缀 -->
25         <property name="suffix" value=".jsp" />
26     </bean>
27 </beans>

```

2. 写一个AjaxController

```

1 @RequestMapping("/ajax1")
2 public void ajax1(String name, HttpServletResponse response) throws
  IOException {
3     System.out.println(name);
4     if ("long".equals(name)){
5         response.getWriter().print("true");
6     }else{
7         response.getWriter().print("false");
8     }
9 }

```

3. 导入jquery，可以使用在线的CDN，也可以下载导入

```

1 <script src="https://code.jquery.com/jquery-3.5.1.js"></script>
2 <script src="${pageContext.request.contextPath}/statics/js/jquery-
  3.5.1.js"></script>

```

4. 编写index.jsp测试

```

1 <%@ page contentType="text/html; charset=UTF-8" language="java" %>
2 <html>
3     <head>
4         <title>首页</title>
5         <script src="${pageContext.request.contextPath}/statics/js/jquery-
  3.5.1.js"></script>
6         <script>
7             function a() {
8                 $.post({
9                     url: "${pageContext.request.contextPath}/ajax1",
10                    data: {"name" : $("#userName").val()},
11                    success: function (data) {
12                        alert(data);
13                    }
14                })
15            }
16        </script>
17    </head>
18    <body>
19        <!--失去焦点的时候利用ajax异步发起一个请求（携带信息）到后台-->
20        用户名: <input type="text" id="userName" onblur="a()">
21    </body>
22 </html>
23

```

5. 启动tomcat测试！打开浏览器的控制台，当我们鼠标离开输入框的时候，可以看到发出了一个ajax的请求！是后台返回给我们的结果！测试成功！

11.3 AJAX异步加载数据

• Springmvc实现

1. 实体类user

```

1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  public class User {
5      private String name;
6      private int age;
7      private String sex;
8  }

```

2. 获取一个集合对象，展示到前端页面

```

1  @RequestMapping("/ajax2")
2  public List<User> ajax2(){
3      List<User> list = new ArrayList<User>();
4      list.add(new User("long",18,"girl"));
5      list.add(new User("abraham",16,"girl"));
6      list.add(new User("abraham",16,"girl"));
7      return list;    //由于@RestController注解，将list转成json格式返回
8  }

```

3. 前端页面

```

1  <%@ page contentType="text/html;charset=UTF-8" language="java" %>
2  <html>
3  <head>
4      <title>Ajax测试</title>
5      <script src="${pageContext.request.contextPath}/statics/js/jquery-
6  3.5.1.js"></script>
7  </head>
8  <body>
9      <input type="button" id="btn" value="获取数据"/>
10     <table width="80%" align="center">
11         <tr>
12             <td>姓名</td>
13             <td>年龄</td>
14             <td>性别</td>
15         </tr>
16         <tbody id="content">
17             </tbody>
18     </table>
19
20     <script>
21         $(function () {
22             $("#btn").click(function () {
23                 $.post("${pageContext.request.contextPath}/ajax2",function
24 (data) {
25                     console.log(data)
26                     var html="";
27                     for (var i = 0; i <data.length ; i++) {
28                         html+= "<tr>" +
29                             "<td>" + data[i].name + "</td>" +
30                             "<td>" + data[i].age + "</td>" +
31                             "<td>" + data[i].sex + "</td>" +
32                             "</tr>"

```

```

33         $("#content").html(html);
34     });
35 }
36 })
37 </script>
38 </body>
39 </html>

```

4. 成功实现了数据回显!



姓名	年龄	性别
long	18	girl
abraham	16	girl
abraham	16	girl

11.4 AJAX验证用户名体验

• 注册提示效果

测试一个小Demo，思考一下我们平时注册时候，输入框后面的实时提示怎么做到的；如何优化

1. 写一个Controller

```

1  @RequestMapping("/ajax3")
2  public String ajax3(String name,String pwd){
3      String msg = "";
4      if (name != null){
5          if ("long".equals(name)){
6              msg = "OK";
7          }else {
8              msg = "error";
9          }
10     }
11     if (pwd!=null){
12         if ("123456".equals(pwd)){
13             msg = "OK";
14         }else {
15             msg = "error";
16         }
17     }
18     return msg;
19 }

```

2. 前端页面 login.jsp

```

1  <%@ page contentType="text/html; charset=UTF-8" language="java" %>
2  <html>
3  <head>
4      <title>ajax登录测试</title>
5      <script src="${pageContext.request.contextPath}/statics/js/jquery-3.5.1.js"></script>
6      <script>
7
8          function ajax1(){
9              $.post({

```

```

10         url:"${pageContext.request.contextPath}/ajax3",
11         data:{"name" : $("#name").val()},
12         success:function (data) {
13             if (data.toString()=='OK'){
14                 $("#userInfo").css("color","green");
15             }else {
16                 $("#userInfo").css("color","red");
17             }
18             $("#userInfo").html(data);
19         }
20     });
21 }
22 function ajax2(){
23     $.post({
24         url:"${pageContext.request.contextPath}/ajax3",
25         data:{"pwd" : $("#pwd").val()},
26         success:function (data) {
27             if (data.toString()=='OK'){
28                 $("#pwdInfo").css("color","green");
29             }else {
30                 $("#pwdInfo").css("color","red");
31             }
32             $("#pwdInfo").html(data);
33         }
34     });
35 }
36
37 </script>
38 </head>
39 <body>
40     <p>
41         用户名:<input type="text" id="name" onblur="ajax1()"/>
42         <span id="userInfo"></span>
43     </p>
44     <p>
45         密码:<input type="text" id="pwd" onblur="ajax2()"/>
46         <span id="pwdInfo"></span>
47     </p>
48 </body>
49 </html>
50

```

3. 测试一下效果，动态请求响应，局部刷新，就是如此！

11.5 补充

• 获取baidu接口Demo

```

1  <!DOCTYPE HTML>
2  <html>
3  <head>
4      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
5      <title>JSONP百度搜索</title>
6      <style>
7          #q{
8              width: 500px;
9              height: 30px;

```

```

10         border:1px solid #ddd;
11         line-height: 30px;
12         display: block;
13         margin: 0 auto;
14         padding: 0 10px;
15         font-size: 14px;
16     }
17     #ul{
18         width: 520px;
19         list-style: none;
20         margin: 0 auto;
21         padding: 0;
22         border:1px solid #ddd;
23         margin-top: -1px;
24         display: none;
25     }
26     #ul li{
27         line-height: 30px;
28         padding: 0 10px;
29     }
30     #ul li:hover{
31         background-color: #f60;
32         color: #fff;
33     }
34 </style>
35 <script>
36
37     // 2.步骤二
38     // 定义demo函数（分析接口、数据）
39     function demo(data){
40         var ul = document.getElementById('ul');
41         var html = '';
42         // 如果搜索数据存在 把内容添加进去
43         if (data.s.length) {
44             // 隐藏掉的ul显示出来
45             ul.style.display = 'block';
46             // 搜索到的数据循环追加到li里
47             for(var i = 0;i<data.s.length;i++){
48                 html += '<li>'+data.s[i]+'</li>';
49             }
50             // 循环的li写入ul
51             ul.innerHTML = html;
52         }
53     }
54
55     // 1.步骤一
56     window.onload = function(){
57         // 获取输入框和ul
58         var q = document.getElementById('q');
59         var ul = document.getElementById('ul');
60
61         // 事件鼠标抬起时候
62         q.onkeyup = function(){
63             // 如果输入框不等于空
64             if (this.value != '') {
65                 // ☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆JSONPz重点
66                 ☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆
67                 // 创建标签

```



```

67         var script = document.createElement('script');
68         //给定要跨域的地址 赋值给src
69         //这里是要请求的跨域的地址 我写的是百度搜索的跨域地址
70         script.src =
        'https://sp0.baidu.com/5a1Fazu8AA54nxGko9WTAnF6hhy/su?
        wd='+this.value+'&cb=demo';
71         // 将组合好的带src的script标签追加到body里
72         document.body.appendChild(script);
73     }
74 }
75 }
76 </script>
77 </head>
78
79 <body>
80 <input type="text" id="q" />
81 <ul id="ul">
82
83 </ul>
84 </body>
85 </html>

```

12. 拦截器

【Module: springmvc-07-Interceptor】

12.1 概述

SpringMVC的处理器拦截器类似于Servlet开发中的过滤器Filter,用于对处理器进行预处理和后处理。开发者可以自己定义一些拦截器来实现特定的功能。

过滤器与拦截器的区别：拦截器是AOP思想的具体应用。

过滤器

- servlet规范中的一部分，任何java web工程都可以使用
- 在url-pattern中配置了/*之后，可以对所有要访问的资源进行拦截

拦截器

- 拦截器是SpringMVC框架自己的，只有使用了SpringMVC框架的工程才能使用
- 拦截器只会拦截访问的控制器方法，如果访问的是jsp/html/css/image/js是不会进行拦截的

12.2 自定义拦截器

想要自定义拦截器，必须实现 HandlerInterceptor 接口。

1. 新建一个Module，springmvc-07-Interceptor，添加web支持
2. 配置web.xml 和 applicationContext.xml文件
3. 编写一个拦截器

```

1 public class MyInterceptor implements HandlerInterceptor {
2
3     /*

```

```

4      在请求处理的方法之前执行
5      如果 return true; 执行下一个拦截器
6      如果 return false; 就不执行下一个拦截器
7      */
8      public boolean preHandle(HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse, Object o) throws Exception {
9          System.out.println("-----处理前-----");
10         return true;
11     }
12
13     /*
14     在请求处理方法执行之后执行
15     */
16     public void postHandle(HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse, Object o, ModelAndView
    modelAndView) throws Exception {
17         System.out.println("-----处理后-----");
18     }
19
20     /*
21     在dispatcherServlet处理后执行,做清理工作.
22     */
23     public void afterCompletion(HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse, Object o, Exception e) throws
    Exception {
24         System.out.println("-----清理-----");
25     }
26 }
27

```

4. 在springmvc的配置文件中配置拦截器

```

1  <!--关于拦截器的配置-->
2  <mvc:interceptors>
3      <mvc:interceptor>
4          <!--/** 包括路径及其子路径-->
5          <!--/admin/* 拦截的是/admin/add等等这种 , /admin/add/user不会被拦截-
->
6          <!--/admin/** 拦截的是/admin/下的所有-->
7          <mvc:mapping path="/**"/>
8          <!--bean配置的就是拦截器-->
9          <bean class="com.longg.config.MyInterceptor"/>
10     </mvc:interceptor>
11 </mvc:interceptors>

```

5. 编写一个Controller, 接收请求

```

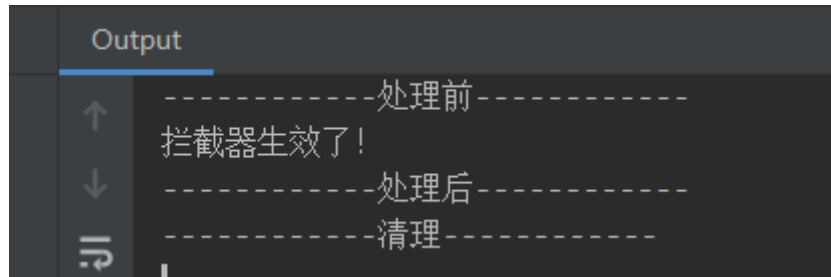
1  @RestController
2  public class TestController {
3
4      @RequestMapping("/interceptor")
5      public String testFunction() {
6          System.out.println("拦截器生效了!");
7          return "hello";
8      }
9  }

```

6. 前端 index.jsp

```
1 <a href="${pageContext.request.contextPath}/interceptor">拦截器测试</a>
```

7. 启动tomcat 测试一下!



12.3 验证用户是否登录 (认证用户)

实现思路

1. 有一个登陆页面，需要写一个controller访问页面。
2. 登陆页面有一提交表单的动作。需要在controller中处理。判断用户名密码是否正确。如果正确，向session中写入用户信息。返回登陆成功。
3. 拦截用户请求，判断用户是否登陆。如果用户已经登陆。放行， 如果用户未登陆，跳转到登陆页面

代码编写

1. 编写一个登陆页面 login.jsp

```
1 <%@ page contentType="text/html; charset=UTF-8" language="java" %>
2 <html>
3 <head>
4     <title>登录</title>
5 </head>
6 <body>
7     <form action="${pageContext.request.contextPath}/user/login">
8         用户名: <input type="text" name="username"> <br>
9         密码: <input type="password" name="pwd"> <br>
10        <input type="submit" value="提交">
11    </form>
12    <hr/>
13    <a href="${pageContext.request.contextPath}/user/index">回到首页</a>
14 </body>
15 </html>
```

2. 编写一个Controller处理请求

```
1 @Controller
2 @RequestMapping("/user")
3 public class LoginController {
4
5     /**
6      * 跳转到登陆页面
7      */
8     @RequestMapping("/enterLogin")
9     public String enterLogin() {
10         return "login";
11     }
12 }
```

```

12
13     /**
14     * 跳转到登录成功的main页面
15     */
16     @RequestMapping("/success")
17     public String success() {
18         return "main";
19     }
20
21     /**
22     * 登陆提交
23     */
24     @RequestMapping("/login")
25     public String login(HttpSession session, String username, String
26     pwd) throws Exception {
27         // 向session存入用户身份信息
28         System.out.println(username);
29         session.setAttribute("username", username);
30         return "main";
31     }
32
33     /**
34     * 退出登陆
35     */
36     @RequestMapping("/logout")
37     public String logout(HttpSession session) throws Exception {
38         // session.invalidate(); // session销毁
39         session.removeAttribute("username"); // 移除session中的单个
40         username节点（推荐使用）
41         return "login";
42     }
43
44     /**
45     * 回首页
46     */
47     @RequestMapping("/index")
48     public String index() throws Exception {
49         return "redirect:/index.jsp";
50     }
51 }

```

3. 编写一个登陆成功的页面 main.jsp

```

1  <%@ page contentType="text/html; charset=UTF-8" language="java" %>
2  <html>
3  <head>
4      <title>main</title>
5  </head>
6  <body>
7      <h2>登录成功页面</h2>
8      <hr/>
9      ${username}
10     <hr>
11     <a href="${pageContext.request.contextPath}/user/logout">注销登录</a>
12     <hr/>
13     <a href="${pageContext.request.contextPath}/user/index">回到首页</a>

```

```
14 </body>
15 </html>
```

4. 在 index 页面上测试跳转！启动Tomcat 测试，未登录也可以进入主页！

```
1 <%@ page contentType="text/html;charset=UTF-8" language="java" %>
2 <html>
3   <head>
4     <title>首页</title>
5   </head>
6   <body>
7     <a href="${pageContext.request.contextPath}/interceptor">拦截器测试
</a>
8     <hr/>
9     <a href="${pageContext.request.contextPath}/user/enterLogin">进入登录
页面</a>
10    <hr/>
11    <a href="${pageContext.request.contextPath}/user/success">进入main页
面</a>
12    <hr>
13    <a href="${pageContext.request.contextPath}/user/logout">注销登录</a>
14  </body>
15 </html>
```

5. 编写用户登录拦截器

方法一：拦截所有/user下的请求，根据条件判断给特定的请求放行

```
1 public class LoginInterceptor implements HandlerInterceptor {
2     public boolean preHandle(HttpServletRequest request,
3                               HttpServletResponse response, Object handler) throws Exception {
4         System.out.println("当前请求的URL: " + request.getRequestURI());
5         // 放行：在登陆页面则放行
6         if (request.getRequestURI().contains("login")) {
7             System.out.println("我在登录页面进入了main页面");
8             return true;
9         }
10
11        // 放行：回首页则放行
12        if (request.getRequestURI().contains("index")) {
13            System.out.println("我在登录页面进入了main页面");
14            return true;
15        }
16
17        HttpSession session = request.getSession();
18        // 放行：用户已登陆则放行
19        if(session.getAttribute("username") != null) {
20            System.out.println("我已经登录成功，直接进入了main页面");
21            return true;
22        }
23
24        // 判断用户没有登陆，则跳转到登陆页面
25        request.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward(request, response);
26        System.out.println("没有登录，请登录");
27        return false;
28    }
29 }
```

```

28     }
29 }

```

方法二：只拦截/user/success请求，其他请求全部不拦截，如果/user/success符合登录条件则放行，否则跳转到登录页面

```

1  public class LoginInterceptor implements HandlerInterceptor {
2      public boolean preHandle(HttpServletRequest request,
3      HttpServletResponse response, Object handler) throws Exception {
4      HttpSession session = request.getSession();
5      // 放行：用户已登陆则放行
6      if(session.getAttribute("username") != null) {
7          System.out.println("我已经登录成功，直接进入了main页面");
8          return true;
9      }
10
11     // 判断用户没有登陆，则跳转到登陆页面
12     request.getRequestDispatcher("/WEB-INF/jsp/login.jsp").forward(request, response);
13     System.out.println("没有登录，请登录");
14     return false;
15 }

```

6. 在Springmvc的配置文件中注册拦截器

方法一：拦截所有/user下的请求，根据条件判断给特定的请求放行

```

1  <!--关于拦截器的配置-->
2  <mvc:interceptors>
3      <mvc:interceptor>
4          <mvc:mapping path="/user/**"/>
5          <bean class="com.longg.config.LoginInterceptor"/>
6      </mvc:interceptor>
7  </mvc:interceptors>

```

方法二：只拦截/user/success请求，其他请求全部不拦截，如果/user/success符合登录条件则放行，否则跳转到登录页面

```

1  <!--关于拦截器的配置-->
2  <mvc:interceptors>
3      <mvc:interceptor>
4          <mvc:mapping path="/user/success"/>
5          <bean class="com.longg.config.LoginInterceptor"/>
6      </mvc:interceptor>
7  </mvc:interceptors>

```

7. 再次重启Tomcat测试！

13. SpringMVC：文件上传和下载

【Module: springmvc-08-file】

• 准备工作

文件上传是项目开发中最常见的功能之一，springMVC 可以很好的支持文件上传，但是SpringMVC上下文中默认没有装配MultipartResolver，因此默认情况下其不能处理文件上传工作。如果想使用Spring的文件上传功能，则需要在上下文中配置MultipartResolver。

前端表单要求：为了能上传文件，必须将表单的method设置为POST，并将enctype设置为multipart/form-data。只有在这样的情况下，浏览器才会把用户选择的文件以二进制数据发送给服务器；

对表单中的 enctype 属性做个详细的说明：

- application/x-www-form-urlencoded：默认方式，只处理表单域中的 value 属性值，采用这种编码方式的表单会将表单域中的值处理成 URL 编码方式。
- multipart/form-data：这种编码方式会以二进制流的方式来处理表单数据，这种编码方式会把文件域指定文件的内容也封装到请求参数中，不会对字符编码。
- text/plain：除了把空格转换为 "+" 号外，其他字符都不做编码处理，这种方式适用直接通过表单发送邮件。

```
1 <form action="" enctype="multipart/form-data" method="post">
2   <input type="file" name="file"/>
3   <input type="submit">
4 </form>
```

一旦设置了enctype为multipart/form-data，浏览器即会采用二进制流的方式来处理表单数据，而对于文件上传的处理则涉及在服务器端解析原始的HTTP响应。在2003年，Apache Software Foundation发布了开源的Commons FileUpload组件，其很快成为Servlet/JSP程序员上传文件的最佳选择。

- Servlet3.0规范已经提供方法来处理文件上传，但这种上传需要在Servlet中完成。
- 而Spring MVC则提供了更简单的封装。
- Spring MVC为文件上传提供了直接的支持，这种支持是用即插即用的MultipartResolver实现的。
- Spring MVC使用Apache Commons FileUpload技术实现了一个MultipartResolver实现类：CommonsMultipartResolver。因此，SpringMVC的文件上传还需要依赖Apache Commons FileUpload的组件。

13.1 文件上传

一、导入文件上传的jar包，commons-fileupload，Maven会自动帮我们导入他的依赖包 commons-io包；

```

1  <!--文件上传-->
2  <dependency>
3      <groupId>commons-fileupload</groupId>
4      <artifactId>commons-fileupload</artifactId>
5      <version>1.3.3</version>
6  </dependency>
7  <!--servlet-api 导入高版本的-->
8  <dependency>
9      <groupId>javax.servlet</groupId>
10     <artifactId>javax.servlet-api</artifactId>
11     <version>4.0.1</version>
12 </dependency>

```

二、配置bean: multipartResolver

【注意！！这个bean的id必须为：multipartResolver，否则上传文件会报400的错误！在这里栽过坑,教训！】

```

1  <!--文件上传配置-->
2  <bean id="multipartResolver"
3      class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
4      <!-- 请求的编码格式，必须和jsp的pageEncoding属性一致，以便正确读取表单的内容，默认为
ISO-8859-1 -->
5      <property name="defaultEncoding" value="utf-8"/>
6      <!-- 上传文件大小上限，单位为字节（10485760=10M） -->
7      <property name="maxUploadSize" value="10485760"/>
8      <property name="maxInMemorySize" value="40960"/>
9  </bean>

```

CommonsMultipartFile 的 常用方法:

- **String getOriginalFilename():** 获取上传文件的原名
- **InputStream getInputStream():** 获取文件流
- **void transferTo(File dest):** 将上传文件保存到一个目录文件中

我们去实际测试一下

三、编写前端页面

```

1  <form action="/upload" enctype="multipart/form-data" method="post">
2      <input type="file" name="file"/>
3      <input type="submit" value="upload">
4  </form>

```

四、Controller

```

1  package com.kuang.controller;
2
3  import org.springframework.stereotype.Controller;
4  import org.springframework.web.bind.annotation.RequestMapping;
5  import org.springframework.web.bind.annotation.RequestParam;
6  import org.springframework.web.multipart.commons.CommonsMultipartFile;
7
8  import javax.servlet.http.HttpServletRequest;
9  import java.io.*;
10
11 @Controller

```



```

12 public class FileController {
13     //RequestParam("file") 将name=file控件得到的文件封装成CommonsMultipartFile
    对象
14     //批量上传CommonsMultipartFile则为数组即可
15     @RequestMapping("/upload")
16     public String fileUpload(@RequestParam("file") CommonsMultipartFile file
    , HttpServletRequest request) throws IOException {
17
18         //获取文件名 : file.getOriginalFilename();
19         String uploadFileName = file.getOriginalFilename();
20
21         //如果文件名为空, 直接回到首页!
22         if ("".equals(uploadFileName)){
23             return "redirect:/index.jsp";
24         }
25         System.out.println("上传文件名 : "+uploadFileName);
26
27         //上传路径保存设置
28         String path = request.getServletContext().getRealPath("/upload");
29         //如果路径不存在, 创建一个
30         File realPath = new File(path);
31         if (!realPath.exists()){
32             realPath.mkdir();
33         }
34         System.out.println("上传文件保存地址: "+realPath);
35
36         InputStream is = file.getInputStream(); //文件输入流
37         OutputStream os = new FileOutputStream(new
    File(realPath,uploadFileName)); //文件输出流
38
39         //读取写出
40         int len=0;
41         byte[] buffer = new byte[1024];
42         while ((len=is.read(buffer))!=-1){
43             os.write(buffer,0,len);
44             os.flush();
45         }
46         os.close();
47         is.close();
48         return "redirect:/index.jsp";
49     }
50 }

```

五、测试上传文件, OK!

- 采用file.Transtio 来保存上传的文件

1. 编写Controller

```

1  /*
2   * 采用file.Transtio 来保存上传的文件
3   */
4  @RequestMapping("/upload2")
5  public String fileUpload2(@RequestParam("file") CommonsMultipartFile
    file, HttpServletRequest request) throws IOException {
6
7      //上传路径保存设置

```

```

8      String path = request.getServletContext().getRealPath("/upload");
9      File realPath = new File(path);
10     if (!realPath.exists()){
11         realPath.mkdir();
12     }
13     //上传文件地址
14     System.out.println("上传文件保存地址: "+realPath);
15
16     //通过CommonsMultipartFile的方法直接写文件（注意这个时候）
17     file.transferTo(new File(realPath + "/" +
18         file.getOriginalFilename()));
19
20     return "redirect:/index.jsp";
21 }

```

2. 前端表单提交地址修改

3. 访问提交测试，OK!

13.2 文件下载

文件下载步骤：

1. 设置 response 响应头
2. 读取文件 -- InputStream
3. 写出文件 -- OutputStream
4. 执行操作
5. 关闭流（先开后关）

代码实现：

```

1  @RequestMapping(value="/download")
2  public String downloads(HttpServletResponse response ,HttpServletRequest
3      request) throws Exception{
4      //要下载的图片地址
5      String path = request.getServletContext().getRealPath("/upload");
6      String fileName = "基础语法.jpg";
7
8      //1、设置response 响应头
9      response.reset(); //设置页面不缓存,清空buffer
10     response.setCharacterEncoding("UTF-8"); //字符编码
11     response.setContentType("multipart/form-data"); //二进制传输数据
12     //设置响应头
13     response.setHeader("Content-Disposition",
14         "attachment;fileName="+URLCoder.encode(fileName, "UTF-8"));
15
16     File file = new File(path,fileName);
17     //2、 读取文件--输入流
18     InputStream input=new FileInputStream(file);
19     //3、 写出文件--输出流
20     OutputStream out = response.getOutputStream();
21
22     byte[] buff =new byte[1024];
23     int index=0;
24     //4、执行 写出操作
25     while((index= input.read(buff))!= -1){
26         out.write(buff, 0, index);
27         out.flush();
28     }
29 }

```

```
27     }
28     out.close();
29     input.close();
30     return null;
31 }
```

前端

```
1 <a href="/download">点击下载</a>
```

测试，文件下载OK，大家可以和我们之前学习的JavaWeb原生的方式对比一下，就可以知道这个便捷多了！

14. 通用文件

14.1 web.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
5                             http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
6         version="4.0">
7     <!--DispatcherServlet-->
8     <servlet>
9         <servlet-name>DispatcherServlet</servlet-name>
10        <servlet-
11class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
12        <init-param>
13            <param-name>contextConfigLocation</param-name>
14            <param-value>classpath:applicationContext.xml</param-value>
15        </init-param>
16        <load-on-startup>1</load-on-startup>
17    </servlet>
18    <servlet-mapping>
19        <servlet-name>DispatcherServlet</servlet-name>
20        <url-pattern>/</url-pattern>
21    </servlet-mapping>
22    <!--encodingFilter-->
23    <filter>
24        <filter-name>encodingFilter</filter-name>
25        <filter-class>
26            org.springframework.web.filter.CharacterEncodingFilter
27        </filter-class>
28        <init-param>
29            <param-name>encoding</param-name>
30            <param-value>utf-8</param-value>
31        </init-param>
32    </filter>
33    <filter-mapping>
```

```

34         <filter-name>encodingFilter</filter-name>
35         <url-pattern>/*</url-pattern>
36     </filter-mapping>
37
38     <!--Session过期时间-->
39     <session-config>
40         <session-timeout>15</session-timeout>
41     </session-config>
42
43 </web-app>

```

14.2 springmvc-servlet.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:context="http://www.springframework.org/schema/context"
5      xmlns:mvc="http://www.springframework.org/schema/mvc"
6      xsi:schemaLocation="http://www.springframework.org/schema/beans
7          http://www.springframework.org/schema/beans/spring-beans.xsd
8          http://www.springframework.org/schema/context
9          https://www.springframework.org/schema/context/spring-context.xsd
10         http://www.springframework.org/schema/mvc
11         https://www.springframework.org/schema/mvc/spring-mvc.xsd">
12
13
14     <context:component-scan base-package="com.kuang.controller"/>
15     <mvc:default-servlet-handler />
16     <!--JSON乱码问题配置-->
17     <mvc:annotation-driven>
18         <mvc:message-converters register-defaults="true">
19             <bean
20 class="org.springframework.http.converter.StringHttpMessageConverter">
21                 <constructor-arg value="UTF-8"/>
22             </bean>
23             <bean
24 class="org.springframework.http.converter.json.MappingJackson2HttpMessageCon
25 verter">
26                 <property name="objectMapper">
27                     <bean
28 class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBe
29 an">
30                         <property name="failOnEmptyBeans" value="false"/>
31                     </bean>
32                 </property>
33             </bean>
34         </mvc:message-converters>
35     </mvc:annotation-driven>
36     <!-- 视图解析器 -->
37     <bean
38 class="org.springframework.web.servlet.view.InternalResourceViewResolver"
39         id="internalResourceViewResolver">
40         <!-- 前缀 -->
41         <property name="prefix" value="/WEB-INF/jsp/" />
42         <!-- 后缀 -->
43         <property name="suffix" value=".jsp" />
44     </bean>

```

14.3 使用到的注解

```

1  @Controller注解类型用于声明Spring类的实例是一个控制器
2  @RequestMapping("/HelloController")
3  @PathVariable int p1
4      组合注解  @GetMapping
5                  @PostMapping
6                  @PutMapping
7                  @DeleteMapping
8                  @PatchMapping
9  @RequestParam("username") String name
10 @ResponseBody
11 组合注解  @RestController

```

14.4 使用到的jar包

```

1  <dependencies>
2      <dependency>
3          <groupId>junit</groupId>
4          <artifactId>junit</artifactId>
5          <version>4.12</version>
6      </dependency>
7
8      <dependency>
9          <groupId>org.springframework</groupId>
10         <artifactId>spring-webmvc</artifactId>
11         <version>5.1.9.RELEASE</version>
12     </dependency>
13
14     <dependency>
15         <groupId>javax.servlet</groupId>
16         <artifactId>servlet-api</artifactId>
17         <version>2.5</version>
18     </dependency>
19
20     <dependency>
21         <groupId>javax.servlet.jsp</groupId>
22         <artifactId>jsp-api</artifactId>
23         <version>2.2</version>
24     </dependency>
25
26     <dependency>
27         <groupId>javax.servlet</groupId>
28         <artifactId>jstl</artifactId>
29         <version>1.2</version>
30     </dependency>
31
32     <!--
33     https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core -
34     ->
35     <dependency>
36         <groupId>com.fasterxml.jackson.core</groupId>
37         <artifactId>jackson-databind</artifactId>
38         <version>2.10.0</version>
39     </dependency>

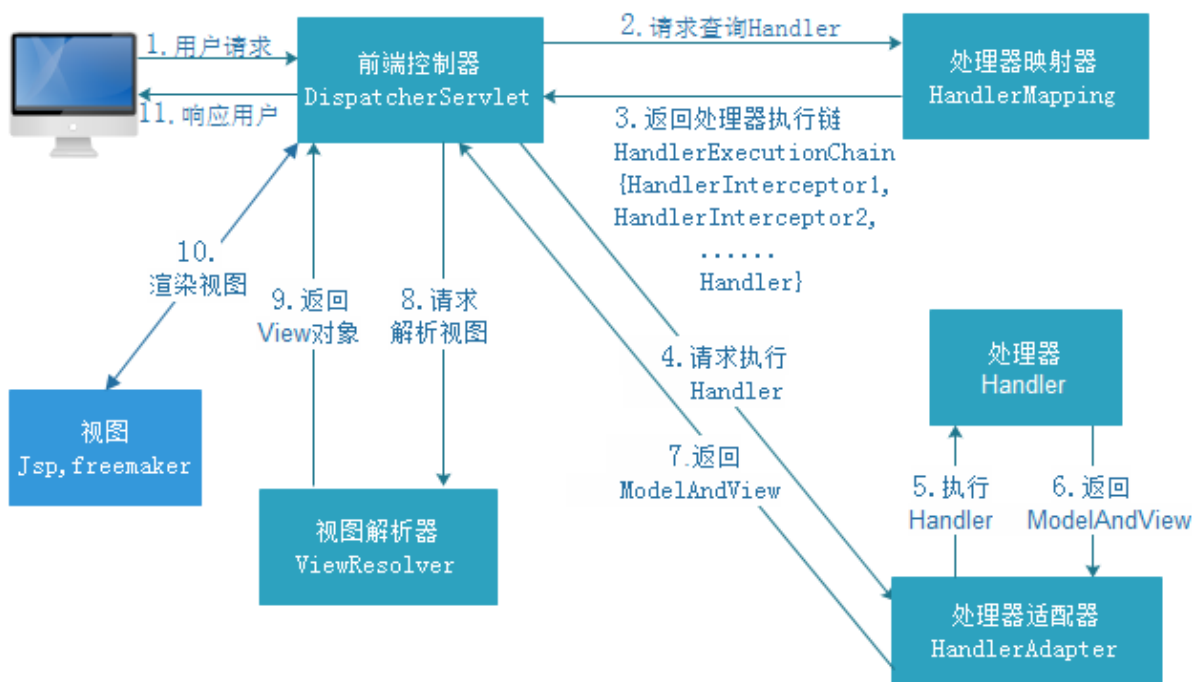
```

```

37     </dependency>
38
39     <!-- fastjson -->
40     <dependency>
41         <groupId>com.alibaba</groupId>
42         <artifactId>fastjson</artifactId>
43         <version>1.2.60</version>
44     </dependency>
45
46     <dependency>
47         <groupId>org.projectlombok</groupId>
48         <artifactId>lombok</artifactId>
49         <version>1.18.10</version>
50         <scope>compile</scope>
51     </dependency>
52 </dependencies>

```

15. SpringMVC的执行流程



- 一个请求匹配前端控制器 DispatcherServlet 的请求映射路径(在 web.xml中指定), WEB 容器将该请求转交给 DispatcherServlet 处理
- DispatcherServlet 接收到请求后, 将根据 请求信息 交给 处理器映射器 (HandlerMapping)

- **HandlerMapping** 根据用户的url请求 查找匹配该url的 **Handler**，并返回一个执行链
- **DispatcherServlet** 再请求 处理器适配器(**HandlerAdapter**) 调用相应的 **Handler** 进行处理并返回 **ModelAndView** 给 **DispatcherServlet**
- **DispatcherServlet** 将 **ModelAndView** 请求 **ViewResolver**（视图解析器）解析，返回具体 **View**
- **DispatcherServlet** 对 **View** 进行渲染视图（即将模型数据填充至视图中）
- **DispatcherServlet** 将页面响应给用户