MVC

model：Dao层

pojo：User (id, name, sex, age, ..., other)

用户完整的属性

vo：UserVo

视图中有时候不需要完整的User属性，比如只需要（id, name）

view：视图

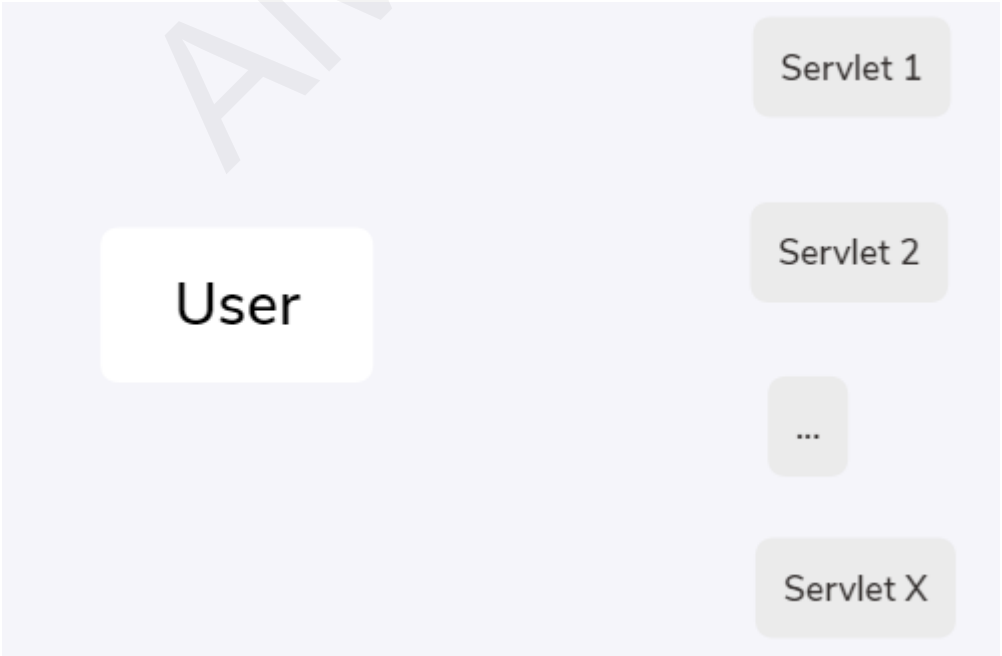
jsp

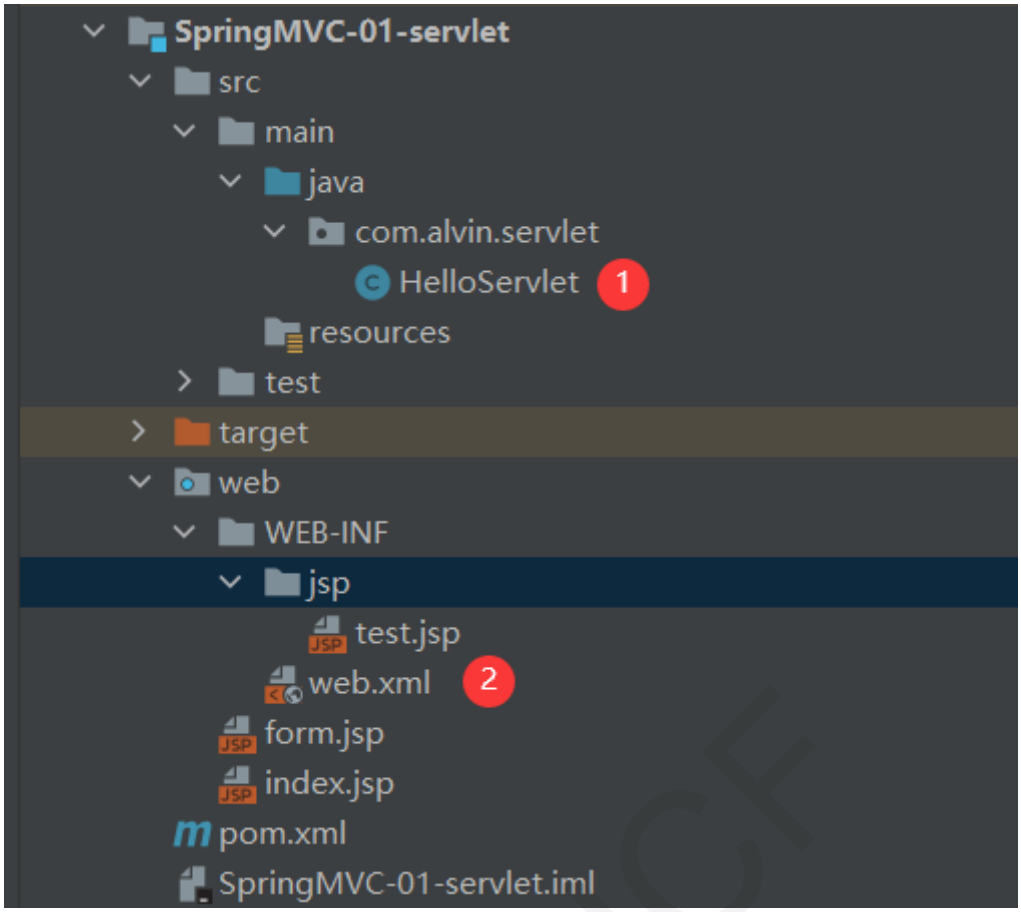
asp.net

controller：控制器

servlet

Servlet





1. 用户请求

```
http://localhost:8080/hello?method=add
http://localhost:8080/hello?method=del
```

2. url映射到java类或者类方法

```
    <!--
        url ->  逻辑地址 ->  class
        /hello ->  hello ->  com.alvin.servlet.HelloServlet
    -->
    <servlet>
        <servlet-name>hello</servlet-name>
        <servlet-class>com.alvin.servlet.HelloServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>hello</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
```

3. 封装用户提交的数据
4. 处理请求--调用相关的业务处理--封装响应数据

```
package com.alvin.servlet;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
```

```java
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class HelloServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        // 获取前端参数
        String method = req.getParameter("method");
        if(method.equals("add")) // http://localhost:8080/hello?method=add
            req.getSession().setAttribute("msg", "执行了add方法");

        if(method.equals("del")) // http://localhost:8080/hello?method=del
            req.getSession().setAttribute("msg", "执行了del方法");
        // 调用业务层
        // 视图转发 + 重定向
        req.getRequestDispatcher("/WEB-INF/jsp/test.jsp").forward(req, resp);
    }

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

5. 将相应的数据进行渲染(jsp/html)

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
${msg}
</body>
</html>
```

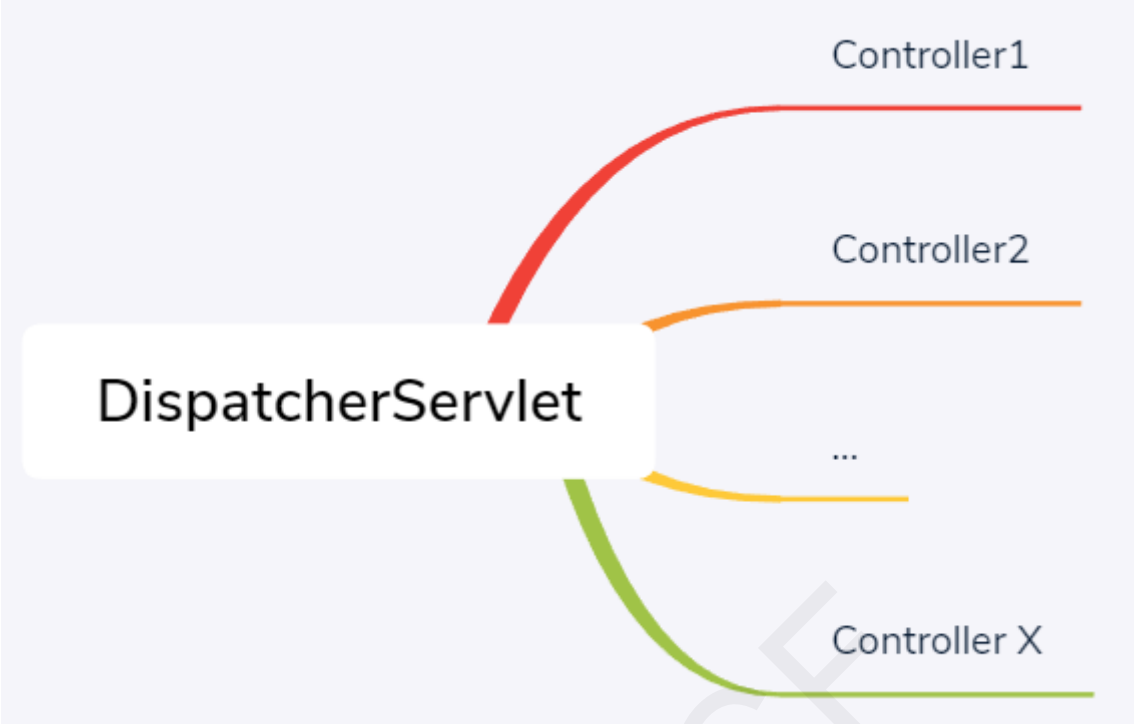

SpringMVC

- 轻量级、简单易学、灵活
- 高效，基于请求响应的MVC框架
- 与Spring无缝结合，兼容性好
- 约定由于配置
- 功能强大：RESTful、数据验证、格式化、本地化、主题等等

围绕`DispatcherServlet`调度设计，DispatcherServlet的作用是将请求发送到不同的处理器。

> 可能的问题：
>
> - 控制台看看错误

- jar是否存在，不存在则添加lib依赖
- 重启tomcat



1. 配置DispatcherServlet

```
    <!--
==========================================================================
============================-->
    <!--
        url ->  逻辑地址 ->  class
        / ->  springmvc ->  org.springframework.web.servlet.DispatcherServlet
    -->
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- 关联一个 SpringMVC 的配置文件 -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc-servlet.xml</param-value>
        </init-param>

        <!-- 启动级别 -->
        <load-on-startup>1</load-on-startup>
    </servlet>
    <!-- / 匹配所有的请求 不包括 .jsp -->
    <!-- /* 匹配所有的请求 包括 .jsp -->
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

```
    <!--
================================================================================
============================-->
```

2. 配置springmvc-servlet.xml

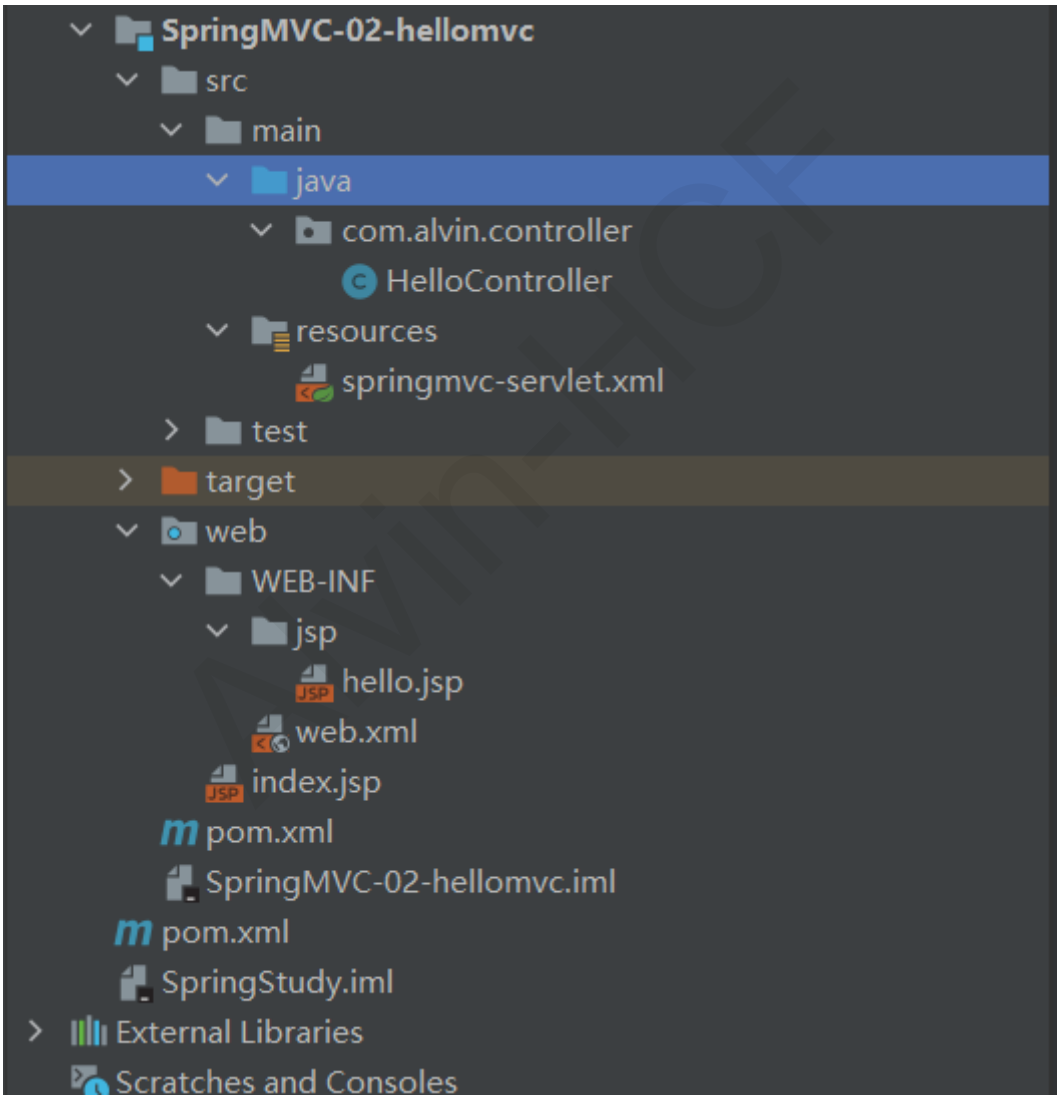
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- web.xml 配置
        url ->  逻辑地址 ->  class
        / ->  springmvc ->  org.springframework.web.servlet.DispatcherServlet

        在这之后，DispatcherServlet 会进行url和控制器的映射，如例子所示：
            http://localhost:8080/hello/handShake 分为三个部分
            http://localhost:8080: ip和端口找到服务器
            / : org.springframework.web.servlet.DispatcherServlet
            /hello/handShake : DispatcherServlet对其进行地址映射找到对应的 Controller

    -->

    <!-- HandlerMapping 映射器 -->
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

    <!-- HandlerAdapter 控制器适配器 -->
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>

    <!-- Handler 控制器 -->
    <bean id="/hello" class="com.alvin.controller.HelloController"/>

    <!-- viewResolver 视图解析器 -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="internalResourceViewResolver">
        <!-- 前缀 -->
        <property name="prefix" value="/WEB_INF/jsp/"/>
        <!-- 后缀 -->
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

3. 控制器

```
package com.alvin.controller;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

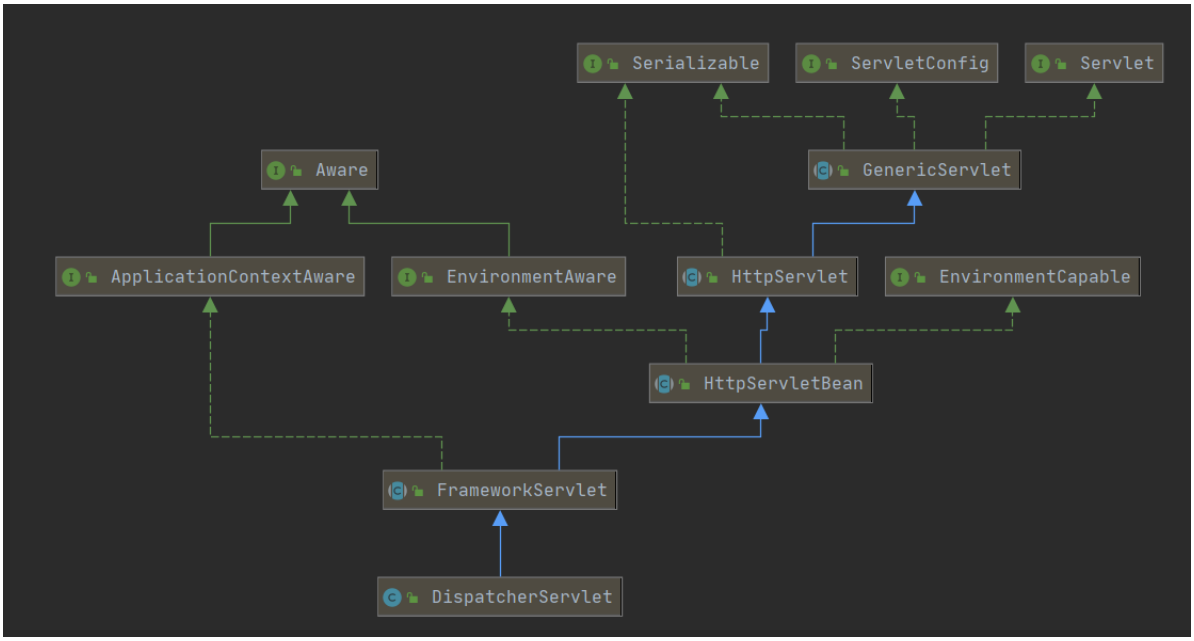
```java
// 实现 Controller 接口
public class HelloController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
        // ModelAndView 模型视图对象
        ModelAndView mv = new ModelAndView();
        // 封装对象，放置到 mv 中
        mv.addObject("msg", "HelloSpringMVC");
        // /WEB-INF/jsp/hello.jsp
        mv.setViewName("hello");
        return mv;
    }
}
```

4. 目录结构

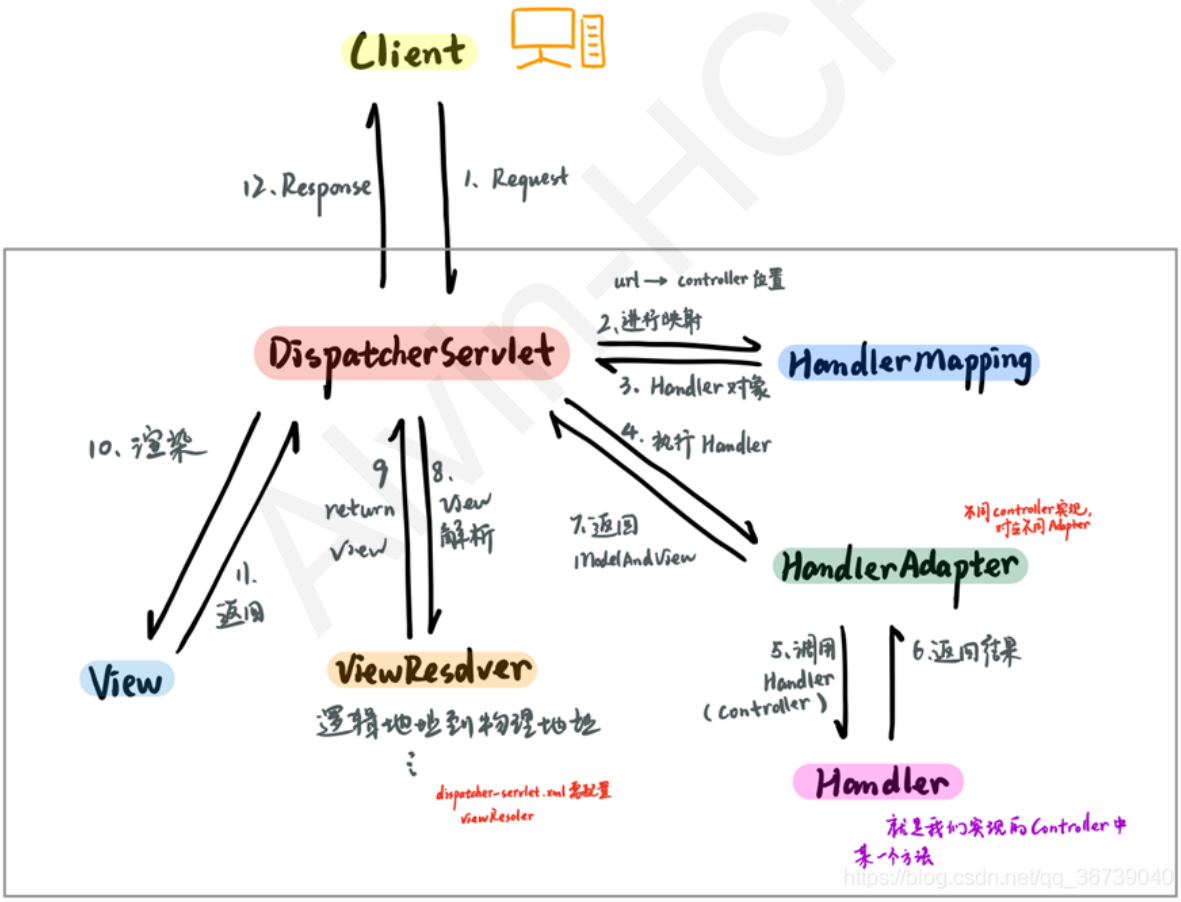


DispatcherServlet

继承关系



功能关系



Controller

实现

1. Controller接口实现
2. @Controller注解

- 接口实现

```
package org.springframework.web.servlet.mvc;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.lang.Nullable;
import org.springframework.web.servlet.ModelAndView;

@FunctionalInterface
public interface Controller {
    @Nullable
    ModelAndView handleRequest(HttpServletRequest var1, HttpServletResponse
var2) throws Exception;
}
```

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

    <!-- web.xml 配置
        url ->  逻辑地址 ->  class
        / ->  springmvc ->  org.springframework.web.servlet.DispatcherServlet

        在这之后，DispatcherServlet 会进行url和控制器的映射，如例子所示：
            http://localhost:8080/hello/handShake 分为三个部分
            http://localhost:8080: ip和端口找到服务器
            / : org.springframework.web.servlet.DispatcherServlet
            /hello/handShake : DispatcherServlet对其进行地址映射找到对应的 Controller

    -->

    <!-- %第 2 步% HandlerMapping 映射器 -->
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

    <!-- %第 3 步% HandlerAdapter 控制器适配器 -->
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>

    <!-- %第 4 步% Handler 控制器 -->
    <bean id="/hello" class="com.alvin.controller.HelloController"/>

    <!-- %第 5 步% viewResolver 视图解析器
        1. 获取 ModelAndView 的数据
        2. 解析 ModelAndView 的名字
        3. 拼接视图名字 /WEB-INF/jsp/hello.jsp
        4. 数据渲染到视图上
    -->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="internalResourceViewResolver">
        <!-- 前缀 -->
```

```
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <!-- 后缀 -->
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

```
package com.alvin.controller;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

// 实现 Controller 接口
public class HelloController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
        // ModelAndView 模型视图对象
        ModelAndView mv = new ModelAndView();
        // 封装对象，放置到 mv 中
        mv.addObject("msg", "HelloSpringMVC");
        // /WEB-INF/jsp/hello.jsp
        mv.setViewName("hello");
        return mv;
    }
}
```

- 注解实现

```
    <!-- 自动扫描包，让指定的包注解生效，由IOC容器统一管理; 但是需要使用@Conponent,
@Controller等等注解
            不在需要单独的配置，如下:
                <bean id="/hello" class="com.alvin.controller.HelloController"/>
    -->
    <context:component-scan base-package="com.alvin.controller"/>

    <!-- 支持注解驱动
        在Spring中一般使用 @RequestMapping 注解来完成映射关系，想要让@RequestMapping生效
        必须向上下文(context)中注册 DefaultAnnotationHandlerMapping 和
AnnotationMethodhandlerAdapter 实例,
        这两个实例会在类级别和方法级别进行处理。
        而 annotation-driven 配置帮助我们自动完成上述两个实例的注入

        <mvc:annotation-driven/>     : 完成了以下俩个配置的功能
            1. HandlerMapping 映射器
            <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
            2. HandlerAdapter 控制器适配器
            <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
```

```
    -->
    <mvc:annotation-driven/>
```

```
package com.alvin.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/hello")
public class HelloController {

    @RequestMapping("/h1")
    public String hello(Model model)
    {
        // 数据封装
        model.addAttribute("msg","Hello, Annotation");
        // /WEB-INF/jsp/hello.jsp
        return "hello";
    }
}
```

RequestMapping

```
// RequestMapping的源码
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

package org.springframework.web.bind.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.springframework.core.annotation.AliasFor;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Mapping
public @interface RequestMapping {
    String name() default "";

    @AliasFor("path")
    String[] value() default {};

    @AliasFor("value")
    String[] path() default {};

    RequestMethod[] method() default {};
```

```
    String[] params() default {};

    String[] headers() default {};

    String[] consumes() default {};

    String[] produces() default {};
}
```

```
@RequestMapping("/hello")
@RequestMapping(value="/hello")
@RequestMapping(path="/hello")
// 三者含义一致
```

> 请求的url映射到对应的业务处理方法。

```
@RequestMapping("/hello")
```

等同于bean配置：

```
<bean id="/hello" class="com.alvin.controller.HelloController"/>
```

RESTful风格

- 让路径更加简洁
- 获取参数更加方便
- 更加安全，不会暴露参数含义

1. 使用 / 分割

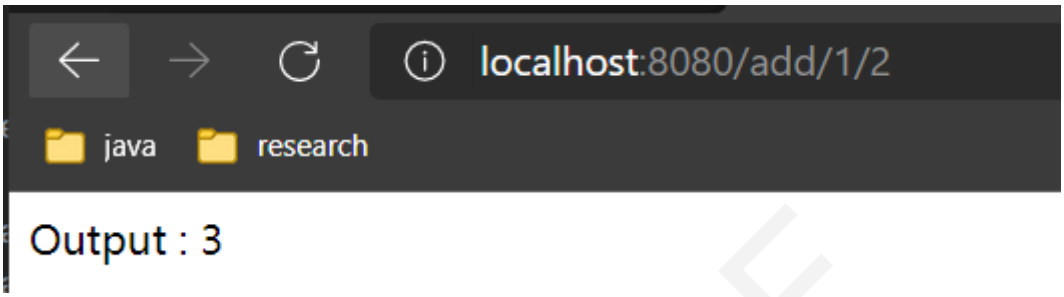
> `localhost:8080/method?add=1`
>
> `localhost:8080/method/1`

2. Get、Post...

> **传统方式处理资源：**
>
> http://127.0.0.1/item/queryUser.action?id=1 查询,GET
> http://127.0.0.1/item/saveUser.action 新增,POST
> http://127.0.0.1/item/updateUser.action 更新,POST
> http://127.0.0.1/item/deleteUser.action?id=1 删除,GET或POST

@RequestMapping("add/{a}/{b}") 和 @PathVariable

```java
@Controller
public class RestFulController {
    @RequestMapping("add/{a}/{b}")
    public String test1(@PathVariable int a, @PathVariable int b, Model model)
    {
        int ret = a + b;
        model.addAttribute("msg","Output : " + ret);
        return "test";
    }
}
```

> http://localhost:8080/add/1/2
>
> 
>

`@RequestMapping("add/{a}/{b}")` 中的 `{a} {b}` 和 `@PathVariable int a, @PathVariable int b` 相对应。

> http://localhost:8080/add/1/2
>
> 使用以前的url表示的话就是：
>
> http://localhost:8080/add?a=1&b=2

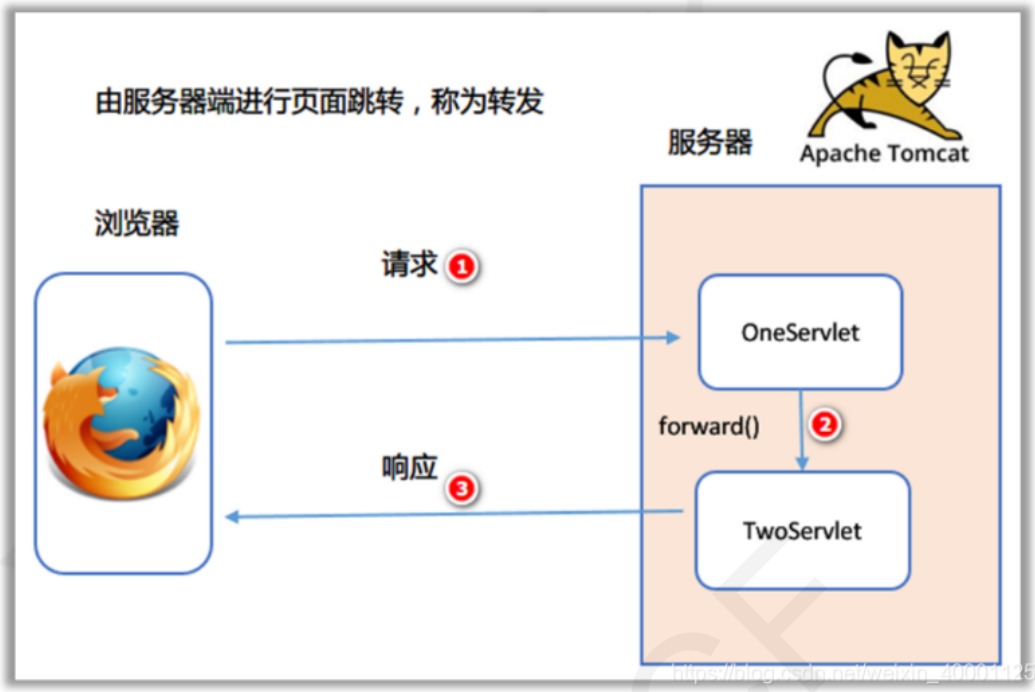
```java
@RequestMapping(name = "URL", method=RequestMethod.POST)
@GetMapping
@PostMapping
@PutMapping
@DeleteMapping
@PatchMapping
...

@PathVariable
```

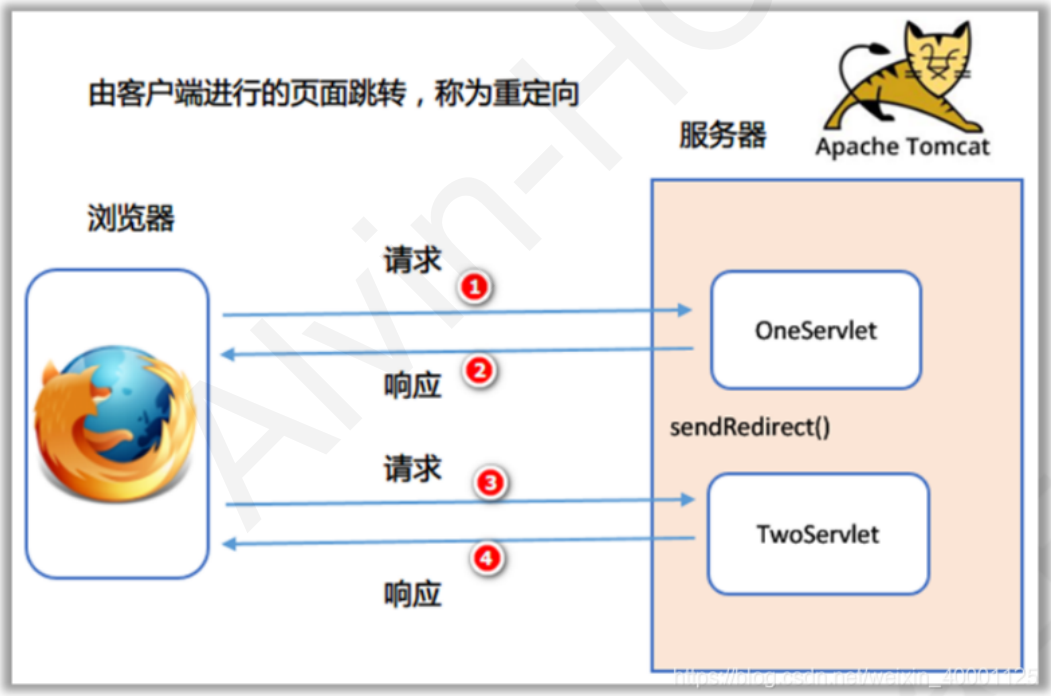
ModelAndView

```xml
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
id="internalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
```

> prefix + ViewName + suffix

转发和重定向问题





转发的特点：

1. 地址栏不发生变化，显示的是上一个页面的地址
2. 请求次数：只有1次请求
3. 根目录：http://localhost:8080/项目地址/，包含了项目的访问地址
4. 请求域中数据不会丢失

重定向的特点：

1. 地址栏：显示新的地址
2. 请求次数：2次
3. 根目录：http://localhost:8080/ 没有项目的名字
4. 请求域中的数据会丢失，因为是2次请求

区别	转发forward()	重定向sendRedirect()
根目录	包含项目访问地址	没有项目访问地址
地址栏	不会发生变化	会发生变化
哪里跳转	服务器端进行的跳转	浏览器端进行的跳转
请求域中数据	不会丢失	会丢失

数据处理

请求参数

请求参数，参数名一致：

```
// http://localhost:8080/hello?name=alvin
@RequestMapping("/hello")
public String hello(String name){
    System.out.println(name);
    return "hello";
}
```

请求参数，参数名不一致： 使用@RequestParam("...")

```
// http://localhost:8080/hello?username=alvin
@RequestMapping("/hello")
public String hello(@RequestParam("username")String name){
    System.out.println(name);
    return "hello";
}
```

请求参数，提交的是一个对象：

```
    /**
     * 属性名字需要一致，顺序没有关系
     * http://localhost:8080/user/t3?id=1&name=alvin&age=18
     * http://localhost:8080/user/t3?name=alvin&id=1&age=18
     *      User(id=1, name=alvin, age=18)
     *
     * http://localhost:8080/user/t3?nasasme=alvin&id=1&age=18
     *      User(id=1, name=null, age=18)
     * http://localhost:8080/user/t3?nme=alvin&ids=1&agse=18
     *      User(id=0, name=null, age=0)
     * 使用对象接收参数
     * @param user
     * @return
     */
    @GetMapping("/t3")
    public String t3(User user){
        System.out.println(user);
        System.out.println("使用对象接收参数");
```

```
        return "test";
    }
```

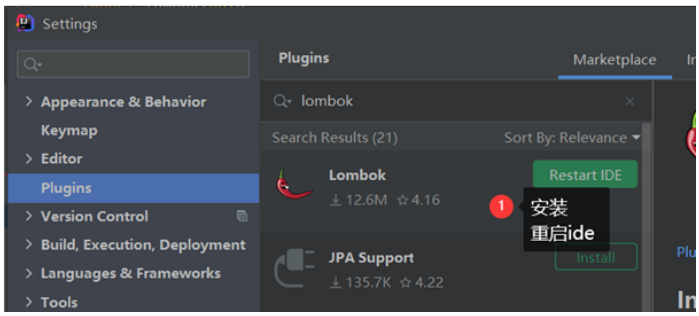
```
package com.alvin.pojo;

/***
 *  lombok ：  自动生成Getter/Setter/Constructor等等方法 的包
 *          <dependency>
 *              <groupId>org.projectlombok</groupId>
 *              <artifactId>lombok</artifactId>
 *              <version>1.18.20</version>
 *          </dependency>
 *
 * @Getter/@Setter ： 作用类上，生成所有成员变量的getter/setter方法；
 *                    作用于成员变量上，生成该成员变量的getter/setter方法。
 *                    可以设定访问权限及是否懒加载等。
 * @ToString ：  作用于类，覆盖默认的toString()方法，可以通过of属性限定显示某些字段，通过
exclude属性排除某些字段。
 * @EqualsAndHashCode ：  作用于类，覆盖默认的equals和hashCode
 * @NonNull ：  主要作用于成员变量和参数中，标识不能为空，否则抛出空指针异常。
 * @NoArgsConstructor ：  生成无参构造器；
 * @AllArgsConstructor ：  生成全参构造器
 * @Builder ：  作用于类上，将类转变为建造者模式
 * @Log ：作用于类上，生成日志变量。针对不同的日志实现产品，有不同的注解
 * ...
 */

import lombok.AllArgsConstructor; // 生成全参构造器
// 作用于类上，是以下注解的集合：@ToString @EqualsAndHashCode @Getter @Setter
@RequiredArgsConstructor
import lombok.Data;
import lombok.NoArgsConstructor; // 生成无参构造器；

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private int id;
    private String name;
    private int age;
}
```

```java
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private String name;
    private int age;
    private String sex;
}
```

```java
@ResponseBody   //不会经过视图解析器
@RequestMapping("/u1/t1")
public String t1()
{
    User user = new User("Alvin",18,"男");
    return_
}
```

提示错误

解决方案： 装 lombok插件



数据回显

数据回显：Model、ModelMap，ModelAndView

```java
    /***
     * @RequestParam("username") 必须使用 username 接收
     * @param name 接收前端传送的参数
     * @param model 发送给前端的参数
     * @return 页面
     */
    // http://localhost:8080/user/t1?username=hahah
    @GetMapping("/t2")
    public String t2(@RequestParam("username") String name, Model model){
        System.out.println("接受前端参数："+ name);
        model.addAttribute("msg", name);
        return "test";
    }
```

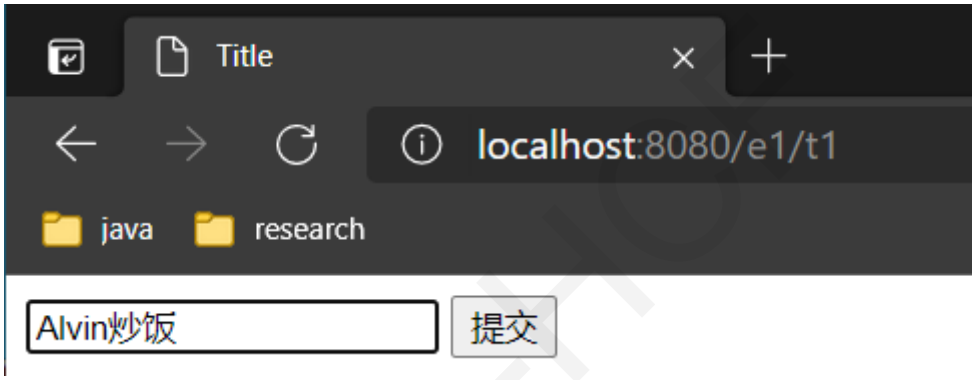
> LinkedHashMap
>
> ModelMap ： 继承了LinkedHashMap，功能更强一点。
>
> Model ： 简单，只有几个方法，简化了操作。
>
> ModelAndView：存储数据的同时，可以进行设置返回的逻辑视图。

```java
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

package org.springframework.ui;

import java.util.Collection;
import java.util.Map;
import org.springframework.lang.Nullable;

public interface Model {
```

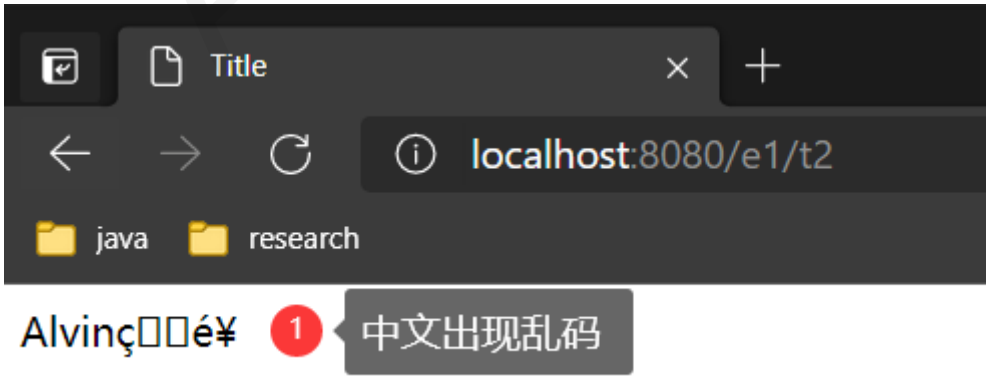
```
    Model addAttribute(String var1, @Nullable Object var2);
    Model addAttribute(Object var1);
    Model addAllAttributes(Collection<?> var1);
    Model addAllAttributes(Map<String, ?> var1);
    Model mergeAttributes(Map<String, ?> var1);
    boolean containsAttribute(String var1);
    @Nullable
    Object getAttribute(String var1);
    Map<String, Object> asMap();
}
```

乱码问题

> 请求时候出现的乱码问题!!!
>
> Client ---> Server




```
<form action="/e1/t2" method="post">
    <input type="text" name="name"/>
    <input type="submit">
</form>
```




```
    @RequestMapping("/e1/t1")
    public String t1()
    {
        return "form";
    }

    // 前端输入：Alvin炒饭
    @RequestMapping(value = "/e1/t2", method = RequestMethod.GET)
```

```
    public String t1(Model model, String name)
    {
        System.out.println(name); // 打印结果： Alvin炒饭
        model.addAttribute("msg", name);
        return "test";
        // 前端页面显示结果： Alvin炒饭
    }

    // 前端输入：Alvin炒饭
    @RequestMapping(value = "/e1/t2", method = RequestMethod.POST)
    public String t2(Model model, String name)
    {
        System.out.println(name); // 打印结果： Alvin???é??
        model.addAttribute("msg", name);
        return "test";
        // 前端页面显示结果： Alvinç••é¥•
    }
```

`get`方式提交的包含中文的字符串一般不会出现乱码，但是`post`方式会出现乱码！！！

在web.xml中配置如下语句，解决问题：

```
<!-- SpringMVC 提供的请求时候的乱码过滤：
            注意这儿解决的是：Client -> Server;  请求时候出现的乱码问题!!! -->
<filter>
        <filter-name>encoding</filter-name>
        <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>utf-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>encoding</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
<!-- / 和 /* 区别：
    <url-pattern> / </url-pattern>
        不会匹配到*.jsp，即：*.jsp不会进入spring的 DispatcherServlet类 。
    <url-pattern> /* </url-pattern>
        会匹配*.jsp，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。
    所以配置过滤器：需要使用 /*
-->
```

> org.springframework.web.filter.CharacterEncodingFilter 可以替换为自己写的过滤类。

```
public class EncodeFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        // 这是很简单的写法
```

```
        servletRequest.setCharacterEncoding("utf-8");
        servletResponse.setCharacterEncoding("utf-8");
        filterChain.doFilter(servletRequest,servletResponse);
    }

    @Override
    public void destroy() {    }
}
```

JSON

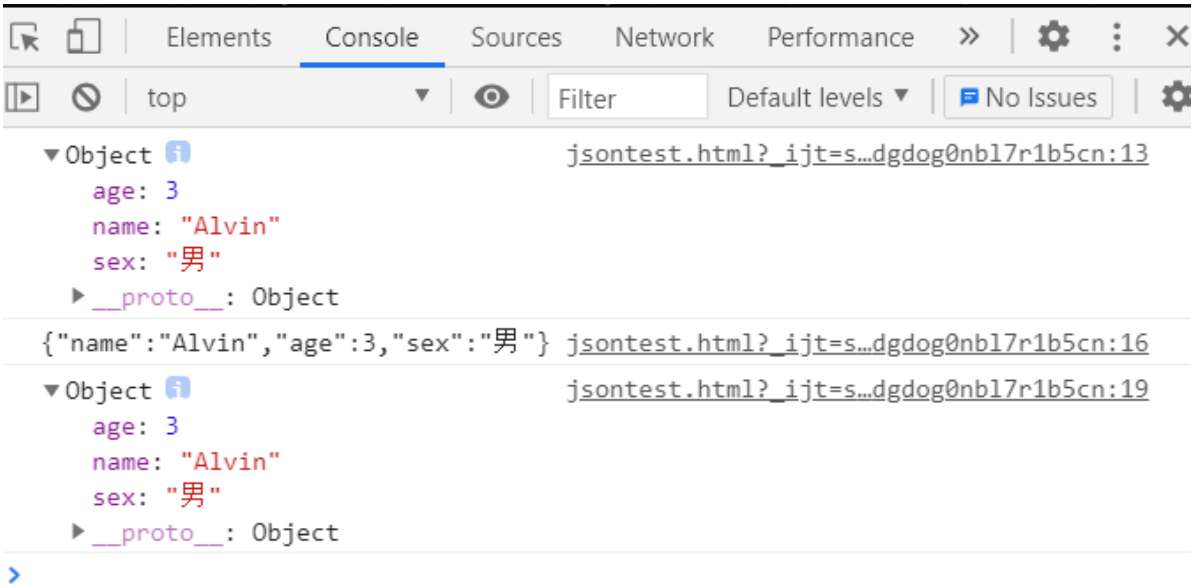
> **前后端分离时代：**
>
> - 后端提供接口，提供数据；
> - 前端独立部署，负责渲染后端数据。
>
> 前后端进行数据交换格式需要统一；JSON是一种很好的格式。

特点：

- 独立于编程语言的文本格式来存储和表示数据
- 简介、清晰的层次结构
- 易于人的读写，机器的生成和解析
- 提升传输效率

json键值对：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <script type="text/javascript">
        // 1. 构建一个 js 对象
        var user = {
            name:"Alvin",
            age:3,
            sex:"男"
        }
        console.log(user)
        // 2. js 对象转换为 json 对象
        var json = JSON.stringify(user)
        console.log(json)
        // 3. json对象转换为 js 对象
        var obj = JSON.parse(json)
        console.log(obj)

    </script>
</head>
<body>

</body>
</html>
```



> json 解析工具
>
> - jackson
>
>  ```
>  <dependency>
>      <groupId>com.fasterxml.jackson.core</groupId>
>      <artifactId>jackson-databind</artifactId>
>      <version>2.12.2</version>
>  </dependency>
>  ```
>
> - 阿里的 fastjson
> - ...

返回json + jackson

@ResponseBody

一般是使用在单独的方法上的，需要哪个方法返回`json`数据格式，就在哪个方法上使用，具有针对性。

@RestController

一般是使用在类上的，它表示的意思其实就是结合了`@Controller`和`@ResponseBody`两个注解。

如果哪个类下的所有方法需要返回`json`数据格式的，就在哪个类上使用该注解，具有统一性；需要注意的是，使用了`@RestController`注解之后，其本质相当于在该类的所有方法上都统一使用了`@ResponseBody`注解，所以该类下的所有方法都会返回`json`数据格式，输出在页面上，而不会再返回视图。

响应时候乱码：

```
package com.alvin.controller;

import com.alvin.pojo.User;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class UserController {

    @ResponseBody   //不会经过视图解析器
    @RequestMapping("/u1/t1")
    public String t1()
    {
        User user = new User("Alvin炒饭",18,"男");
        return user.toString();
        // 前端显示的结果：User(name=Alvin??, age=18, sex=?)
    }

    @ResponseBody   //不会经过视图解析器
    @RequestMapping("/u1/t2")
    public String t2() throws JsonProcessingException {
        // jackson 的 ObjectMapper
        ObjectMapper mapper = new ObjectMapper();
        // 创建一个对象
        User user = new User("Alvin炒饭",18,"男");
        // 转换为字符串
        String str = mapper.writeValueAsString(user);
        return str;
/*
    前端显示的结果：{"name":"Alvin??","age":18,"sex":"?"}
    出现了乱码？？？
        过滤器解决的是：Client ---> Server;  请求时候出现的乱码问题!!!
        而这儿出现的是  Server ---> Client;  响应时候出现的乱码问题!!!

 */
    }

    /***
     * 解决响应时候的乱码问题：
     * 1. 单一方法 @RequestMapping(value = "/u1/t3", produces =
"application/json;charset=utf-8")
     * 2. 统一解决 springmvc的配置文件中添加消息转换配置 <mvc:annotation-driven> 中增加
信息转换配置
     *         <mvc:annotation-driven>
     *         <!-- 响应时期的 JSON 乱码问题解决 -->
     *         <mvc:message-converters>
     *             <bean
class="org.springframework.http.converter.StringHttpMessageConverter">
     *                 <constructor-arg value="utf-8"/>
     *             </bean>
     *             <bean
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConvert
er">
     *                 <property name="objectMapper">
```

```
     *                      <bean
class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean">
     *                          <property name="failOnEmptyBeans" value="false"/>
     *                      </bean>
     *                  </property>
     *              </bean>
     *          </mvc:message-converters>
     *      </mvc:annotation-driven>
     */

    @ResponseBody   //不会经过视图解析器
    @RequestMapping(value = "/u1/t3")
    public String t3() throws JsonProcessingException {
        // jackson 的 ObjectMapper
        ObjectMapper mapper = new ObjectMapper();
        // 创建一个对象
        User user = new User("Alvin炒饭",18,"男");
        // 转换为字符串
        String str = mapper.writeValueAsString(user);

        return str;
        /* 前端显示的结果：{"name":"Alvin炒饭","age":18,"sex":"男"}
        *  无乱码；
        * */
    }
}
```

返回json + fastjson

Fastjson 是一个由阿里开发的 Java 库，可以将 Java 对象转换为 JSON 格式，当然它也可以将 JSON 字符串转换为 Java 对象。

```
<!-- 阿里开发的 fastjson -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.60</version>
</dependency>
```

三个重要的类

- JSONObject 代表 json对象
- JSONArray 代表 json对象数组
- JSON 代表 json对象和json对象数组的转换

```
package com.alvin.controller;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONObject;
import com.alvin.pojo.User;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
```

```java
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.ArrayList;
import java.util.List;

@Controller
public class FastjsonController {
    @ResponseBody   //不会经过视图解析器
    @RequestMapping(value = "/f1/t1")
    public String t1()  {
        // 创建对象
        User user1 = new User("Alvin炒饭",18,"男");
        User user2 = new User("Alvin炒饭",18,"男");
        List<User> list =new ArrayList();
        list.add(user1);
        list.add(user2);
        // 转换为字符串
        System.out.println(" ============== java 转 String ============== ");
        String str = JSON.toJSONString(list);
        System.out.println(" JSON.toJSONString(list) : " + str);
        String str1 = JSON.toJSONString(user1);
        System.out.println(" JSON.toJSONString(user1) : "+ str1);


        System.out.println(" ============== String 转 java ============== ");
        User user = JSON.parseObject(str1, User.class);
        System.out.println(" JSON.parseObject(str1, User.class) : "+ user);


        System.out.println(" ============== java 转 JSON对象 ============== ");
        JSONObject jsonObject1 = (JSONObject) JSON.toJSON(user2);
        System.out.println(" JSON.parseObject(str1, User.class) : "+
jsonObject1.getString("name"));

        System.out.println(" ============== JSON对象 转 java ============== ");
        User to_java_user = JSON.toJavaObject(jsonObject1, User.class);
        System.out.println(" JSON.toJavaObject(jsonObject1, User.class) : "+
to_java_user);
        return str;
/**
 * ============== java 转 String ==============
 *  JSON.toJSONString(list) : [{"age":18,"name":"Alvin炒饭","sex":"男"},
{"age":18,"name":"Alvin炒饭","sex":"男"}]
 *  JSON.toJSONString(user1) : {"age":18,"name":"Alvin炒饭","sex":"男"}
 *  ============== String 转 java ==============
 *  JSON.parseObject(str1, User.class) : User(name=Alvin炒饭, age=18, sex=男)
 *  ============== java 转 JSON对象 ==============
 *  JSON.parseObject(str1, User.class) : Alvin炒饭
 *  ============== JSON对象 转 java ==============
 *  JSON.toJavaObject(jsonObject1, User.class) : User(name=Alvin炒饭, age=18,
sex=男)
 *
 *  前端显示:
 *      [{"age":18,"name":"Alvin炒饭","sex":"男"},{"age":18,"name":"Alvin炒
饭","sex":"男"}]
 *
 **/
    }
```

```
}
```

SSM

```
mysql> select version();
+------------+
| version()  |
+------------+
| 5.7.17-log |
+------------+
1 row in set (0.02 sec)
```

end