SpringMVC

一 Spring MVC 概述

SpringMVC 是一个 WEB 层、控制层框架，主要用来与客户端交互，业务逻辑的调用

SpringMVC 是 Spring 大家族的一大组件， Spring 整合 SpringMVC 可以做到无缝集成。

为了么有了 Servlet 还要学 SpringMVC ？

Servlet 的开发配置相对麻烦，servlet 特别多的时候 web.xml 文件将会非常臃肿

每个 Servlet 都只能处理一个功能，如果需要多个功能就需要开发多个 Servlet，项目中存在大量 Servlet 显得臃肿。

获取请求参数 进行类型转换 封装数据到 bean 的 过程比较繁琐。

其他开发中不方便的地方，例如，乱码问题..数据格式处理..表单校验。

二 Spring MVC 的组件

1 前端控制器

本质上是一个 Servlet，相当于一个中转站，所有的访问都会走到该 Servlet，然

后通过配置中转到相应的 Handler(处理器)，获取到数据和视图后，再使用相应视图做出响应。

2 处理器映射器

本质上就是保存的处理器(Handler)和访问路径的对应关系，在需要时供前端控制器查阅

3 处理器适配器

本质上是一个适配器，可以根据要求找到对应的处理器来运行。前端控制器通过处理器映射器找到对应的处理器信息后，将请求响应对应的 Handler 信息交给处理器适配器处理，处理器适配器找到真正的 Handler 执行后，将结果 Model 和 view 返回前端控制器。

4 处理器(Handler)

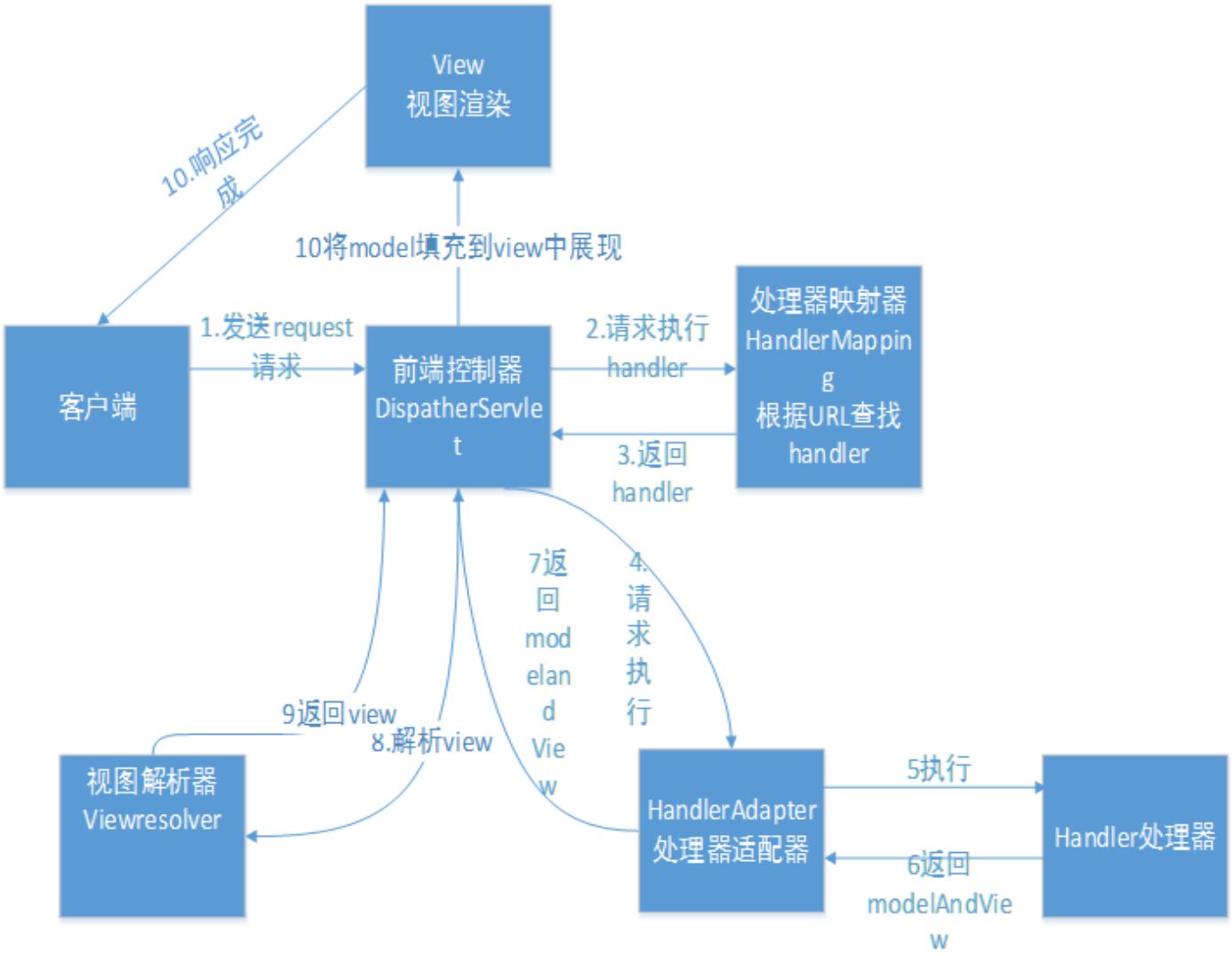
通过注解修饰的方法?

5 视图解析器

本质还是一种映射关系，可以将视图名映射到真正的视图地址。前端控制器调用处理器适配完成后得到 model 和 view，将 view 信息传给视图解析器得到真正的 view

6 视图

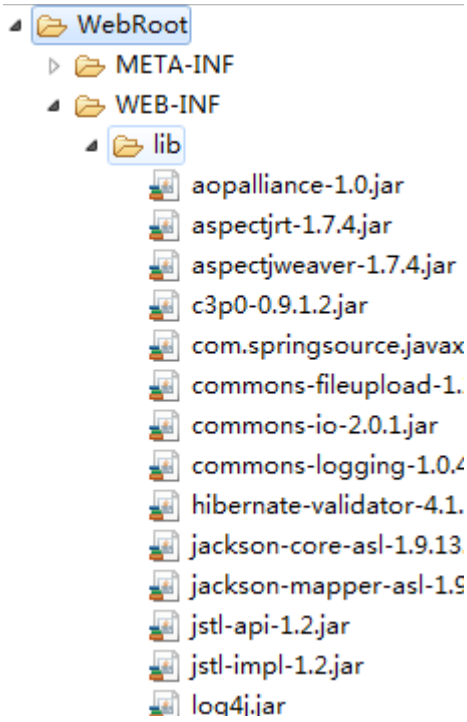
本质上就是将 handler 处理器中返回的 model 数据嵌入到视图解析器解析后得到的 jsp 页面中，向客户端做出响应


View
视图渲染
10.响应完成
10将model填充到view中展现
客户端
1.发送request请求
前端控制器
DispatherServlet
2.请求执行handler
处理器映射器
HandlerMapping
根据URL查找handler
3.返回handler
7返回modelandView
4.请求执行
9返回view
8.解析view
视图解析器
Viewresolver
HandlerAdapter
处理器适配器
5执行
6返回modelAndView
Handler处理器

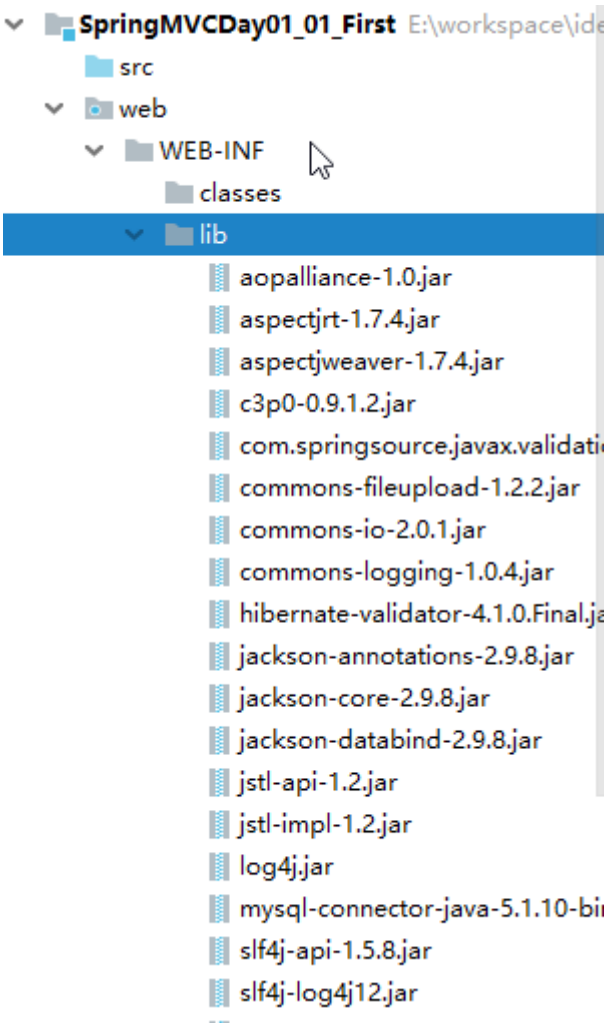
三　Spring MVC 组件的配置

1　导入 SpringMVC 相关开发包

1.A　配置方式



1.B　注解方式



2　创建配置文件

在 src 目录下创建 springMVC.xml

SpringMVC 默认会自动在 web 应用的 WEB-INF 目录下去寻找[前端控制器 ServletName]-servlet.xml 作为当前 SpringMVC 的核心配置文件。

或者在前端控制器中配置目录

创建这个文件，这个文件本身其实就是 Spring 的配置文件，所以导入 Spring 相关的约束信息，包括 beans、context、mvc

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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-3.2.
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd
    ">
```

3 配置前端控制器(WEB.xml 中配置)

前端控制器本质为 Servlet，需要在 web.xml 中配置 Servet。

```xml
<servlet>
                                    <servlet-name>springMvc</servlet-name>
    <!-- 如果报错 DispatcherServlet' is not assignable to javax.Servlet 则需要导入 tomcat-->
                                    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
            <!-- 如果不配置 会到默认目录找默认文件 -->
                                    <init-param>
                                        <param-name>contextConfigLocation</param-name>
                                        <param-value>classpath:springMVC.xml</param-value>
                                    </init-param>
</servlet>
<servlet-mapping>
                                    <servlet-name>springMvc</servlet-name>
            <url-pattern>*.action</url-pattern><!-- 随便写个路径就可以 -->
</servlet-mapping>
```

4 配置视图解析器(Spring MVC 配置文件中)

在 SpringMVC 配置文件中配置

```xml
<!-- 配置视图解析器 -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <!--前缀和后缀，处理器适配器返回视图后会默认在视图名加上前缀后缀-->
```

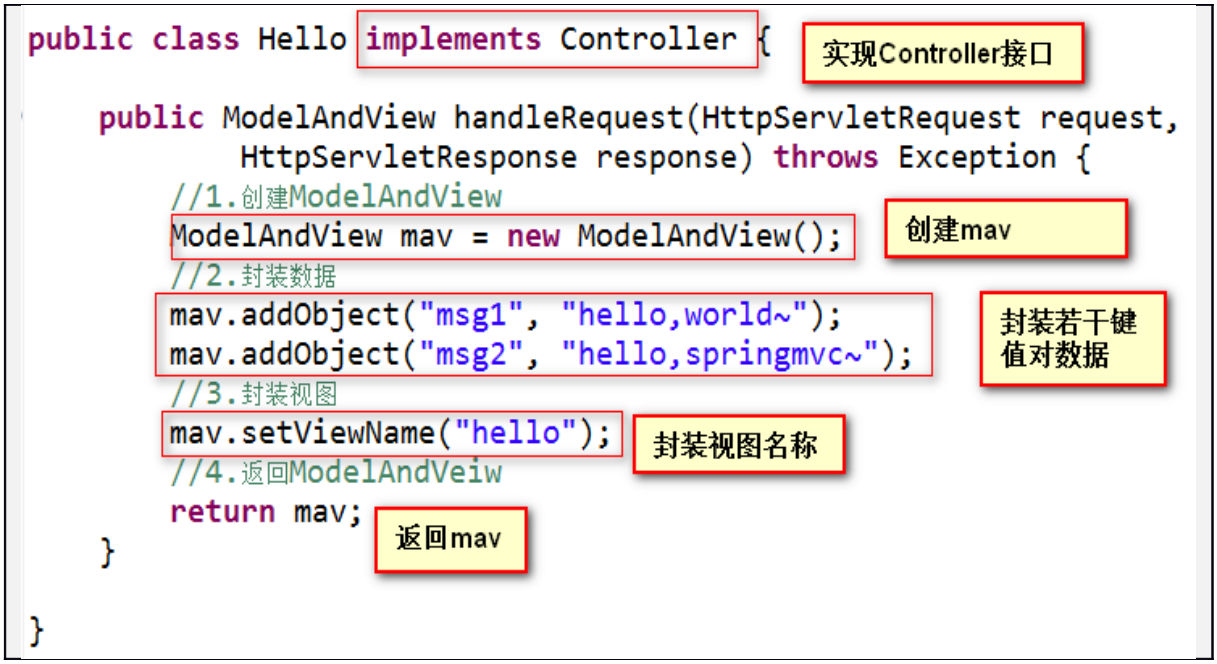
```
    <!--前缀和后缀，处理器如果返回的时资源跳转的字符串，则不会匹配前缀后缀-->
                                <property          name="prefix"          value="/jsp"/>
                                <property          name="suffix"          value=".jsp"/>
</bean>
```

5 创建处理器(类中、方法中配置注解)

5.A 配置方式创建

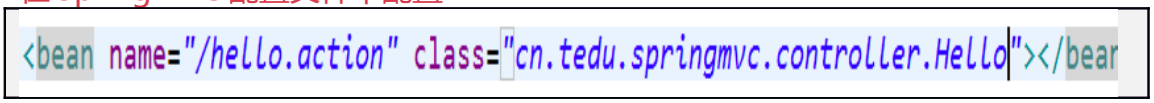
5.A.a 创建处理器

想要开发一个处理器，写一个类实现 Controller 接口，重写 handlerRequest 方法，编写代码处理请求，并将处理好的数据和目标视图封装到 ModelAndView 中返回



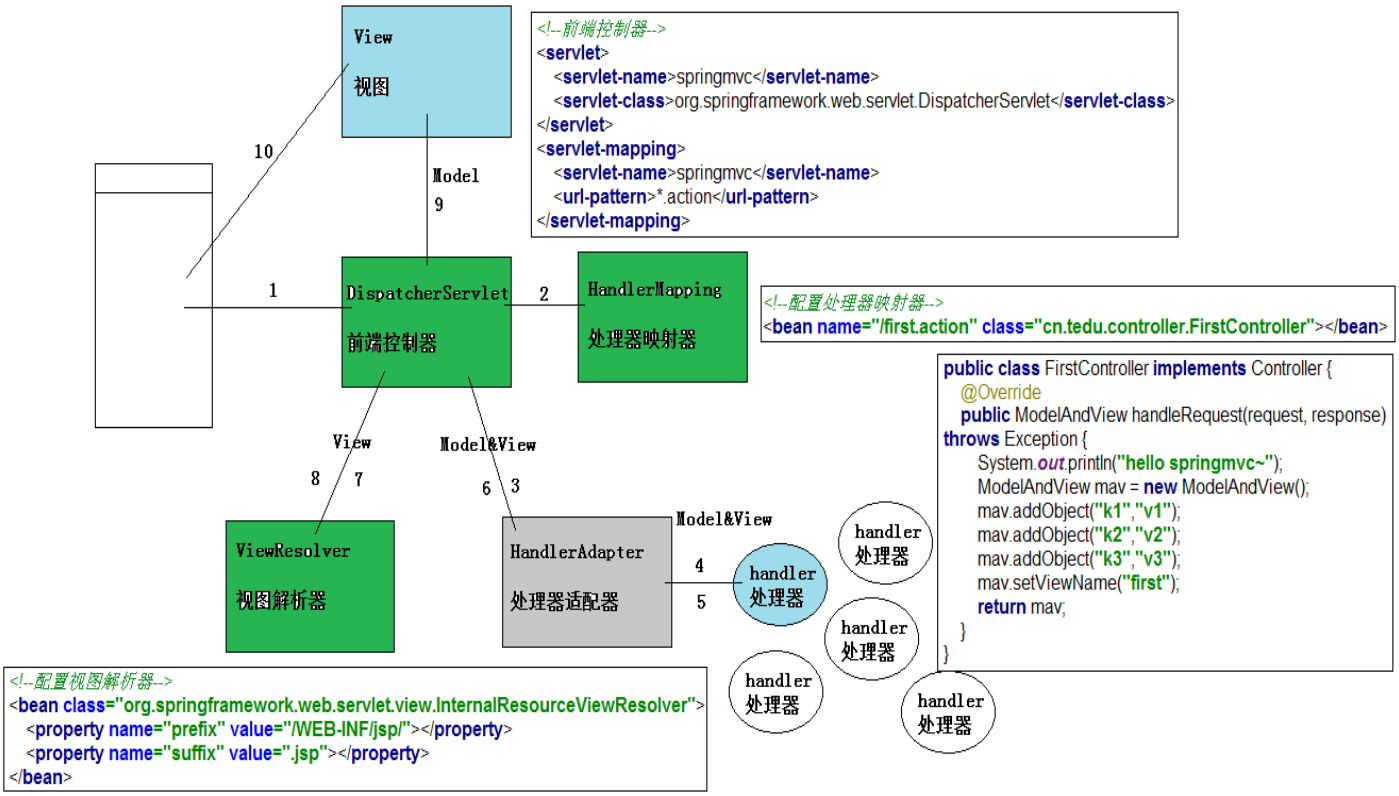
5.A.b 配置处理器映射器中的路径和处理器的映射关系

在 SpringMVC 配置文件中配置



5.A.c 处理流程

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
  <title>Title</title>
</head>
<body>
  first.jsp.... ${k1}${k2}${k3}
</body>
</html>
```




```xml
<!--前端控制器-->
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

```xml
<!--配置处理器映射器-->
<bean name="/first.action" class="cn.tedu.controller.FirstController"></bean>
```

```java
public class FirstController implements Controller {
  @Override
  public ModelAndView handleRequest(request, response)
throws Exception {
    System.out.println("hello springmvc~");
    ModelAndView mav = new ModelAndView();
    mav.addObject("k1","v1");
    mav.addObject("k2","v2");
    mav.addObject("k3","v3");
    mav.setViewName("first");
    return mav;
  }
}
```

```xml
<!--配置视图解析器-->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/jsp/"></property>
  <property name="suffix" value=".jsp"></property>
</bean>
```

5.B 注解方式创建

5.B.a 在 SpringMVC 配置文件中配置

```xml
<!-- 开启 Spring MVC 的包扫描 存放处理器的包 -->
<context:component-scan base-package="cn.shu.blog.web"/>
<!-- 开启注解方式 MVC-->
<mvc:annotation-driven/>
```

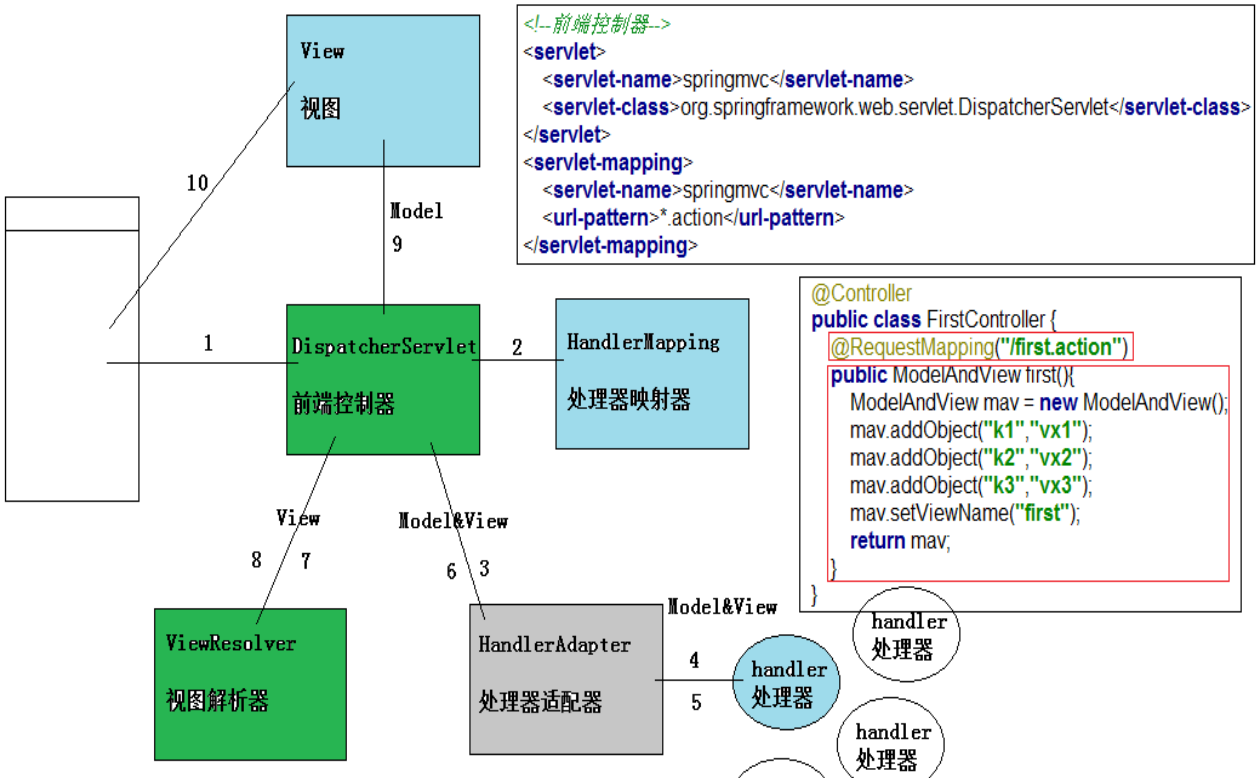
5.B.b 创建处理器

在 SpringMVC 包扫描目录下开发控制器，并配置为 Spring 的 bean
在其中开发控制器方法，并通过@RequestMapping 注解进行路径映射

```java
@Controller
public class FirstController {
    @RequestMapping("/test01.action")
    public ModelAndView test01(){
        //创建mav对象封装模型数据
        ModelAndView mav = new ModelAndView();
        mav.addObject( attributeName: "k1", attributeValue: "v1");
        mav.addObject( attributeName: "k2", attributeValue: "v2");
        //封装视图数据
        mav.setViewName("t1");
        //返回mav
        return mav;
    }
}
```

5.B.c 处理流程

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
  <title>Title</title>
</head>
<body>
  first.jsp.... ${k1}${k2}${k3}
</body>
</html>
```




```xml
<!--前端控制器-->
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

```java
@Controller
public class FirstController {
  @RequestMapping("/first.action")
  public ModelAndView first(){
    ModelAndView mav = new ModelAndView();
    mav.addObject("k1","vx1");
    mav.addObject("k2","vx2");
    mav.addObject("k3","vx3");
    mav.setViewName("first");
    return mav;
  }
}
```

```xml
<!--配置视图解析器-->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/jsp/"></property>
  <property name="suffix" value=".jsp"></property>
</bean>
```

5.B.d **SpringMVC 注解方式工作原理**

1、服务器启动时，先加载 WEB.xml，创建前端控制器 Servlet，前端控制器加载 SpringMVC 配置文件
2、当解析到包扫描时，扫描指定包，将有@Controler 注解的类解析为处理器
3、如果配置了注解方式，就会解析 Spring-MVC 注解
4、解析@RequestMapping(value="/hello.action")，将指定的地址和当前方法的映射关系进行保存
5、当用户发出请求访问一个地址时，SpringMVC 寻找该地址映射关系，如果存在，则找到响应处理器相应方法执行，如果找不到，则报 404

6 配置处理器映射器、处理器适配器

处理器映射器相当于在方法注解上 就已经配置了，处理器适配器不需要配置

四 @RequestMapping 注解详解

通过注解实现访问路径到处理器方法的映射

1 @RequestMapping 基本使用

可以用在方法上和类上
用在方法上表示将该方法变为一个处理器，且和指定路径做映射。
用在类上则配置的路径会作为这个类中所有处理器的路径的父路径使用。
在访问时要加上父路径和方法上的路径才能由该处理器处理

2 @RequestMapping 注解属性

```
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 {};
}
```

2.A String name() default "";

2.B String[] value() default {};

指定访问的路径 KEY 为 value，使用时如果注解只有一个参数，” value= “ 可省略，若 KEY 对应的只有一个值，则方括号 { } 可省略

```
@RequestMapping("/addComment.action")
public String addComment(HttpServletReque
```

访问：localhost/addComment.action

value 属性支持通配符匹配：

```
@RequestMapping("login/*")
public String addComment(HttpServletRequest r
```

访问：localhost/login/后接任意字符都可访问

2.C String[] path() default {};

与 value 同义，path(value)（path 和 value 互相引用，参见 RequestMapping 接口源码）path 属性，和 B 中的 value 属性使用一致，两者都是用来作为映射使用的。
也可使用通配符

2.D RequestMethod[] method() default {};

指定该处理器接收哪种方式的请求，*以下表示只接收 POST 请求*

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

public String addComment(HttpServletRequest request,Comment comment){
```

2.E String[] params() default {};

用来限定当前请求中必须包含指定名称的请求参数才会被当前处理器处理
通过 params 属性指定只处理请求参数符合指定要求的请求

格式 1:只指定名称，要求必须具有该名称的请求参数
格式 2:以"名称=值"或"名称!=值"的方式指定必须具有某个请求参数，且值必须等于或不等于给定值
格式 3:以"!名称"的方式指定必须不包含指定名称的请求参数

以下表示接收有 name 参数，且 gender 为 male，且没有!age...的请求

```
@RequestMapping(value = "/addComment.action",params= {"name","gender=male","!age","addr!=bj"})
```

2.F String[] headers() default {};

用来限定当前请求中必须包含指定名称的请求头才会被当前处理器处理

格式 1:只指定名称，要求必须具有该名称的请求头
格式 2:以"名称=值"或"名称!=值"的方式指定必须具有某个请求头，且值必须等于或不等于给定值
格式 3:以"!名称"的方式指定必须不包含指定名称的请求头

以下注解表示必须有请求头 host，且等于 localhost

```
@RequestMapping(value = "/addComment.action",headers= {"host=localhost"})
```

2.G String[] consumes() default {};

指定处理请求的提交内容类型（Content-Type），例如：application/json、text/html 时，才能够让该方法处理请求

例：只接收 json 数据类型的请求

```
@RequestMapping(value = "/addComment.action",consumes = "application/json")
```

2.H String[] produces() default {};

指定返回的内容类型，返回的内容类型必须是 request 请求头（Accept）中所包含的类型

例：返回 json 数据

```
@RequestMapping(value = "/addComment.action",produces = "application/json")
```

此外，produces 属性还可以指定返回值的编码

例：指定 utf-8 编码

```
@RequestMapping(value = "login",produces = "application/json, charset=utf-8")
```

*好像只对@ResponseBody 修饰的方法有效？**测试了，好像是***

五 获取请求参数

1 通过 request 对象获取

在处理器方法上传递 request 对象，SpringMVC 会自动传递

然后通过 request 的 getParameter 获取

```
@RequestMapping("/test")
public void test(HttpServletRequest request){
    String name = request.getParameter( s: "name");
}
```

2 形参直接接收参数

```
@RequestMapping("/test")
public void test(String name){
    System.out.println(name);
}
```

2.A 如果请求参数名和方法形参名不同（@RequestParam）

可通过@RequestParam 来指定该形参接收哪个请求参数

value	参数名字，即入参的请求参数名字，如 value= “delId” 表示请求的参数区中的名字为 delId 的参数的值将传入
required	是否必须，默认是 true，表示请求中一定要有相应的参数，否则将报 400 错误码；
defaultValue	默认值，表示如果请求中没有同名参数时的默认值

例：获取形参 myName 接收请求参数中的 name 字段

```
@RequestMapping("/test")
public void test(@RequestParam("name") String myName){
    System.out.println(myName);
}
```

3 请求参数封装到 bean

3.A 普通属性处理

前提是 bean 中必须提供相应的 set 方法，且传递的参数名需与 bean 中 set 方法对应

Bean:

```java
public class User {
    //初始值为访客
    @Value("1")
    private int id;
    private String userName;

    public void setUserName(String userName) {
        this.userName = userName;
    }
```

处理器：可以把请求参数封装到 User bean 中的 userName 属性

```java
    @RequestMapping("/test")
    public void test(User user){
        System.out.println(user);
    }
```

参数:

```
localhost:8080/test?userName=shuxinsheng
```

3.B Bean 中属性有 Bean 的处理

Comment Bean:此时 Comment 中有 User 类型

```java
public class Comment {
    private int id;
    private Date createDate;
    private String location;
    private String comment;

    private User user;
    public void setUser(User user) { this.user = user; }
```

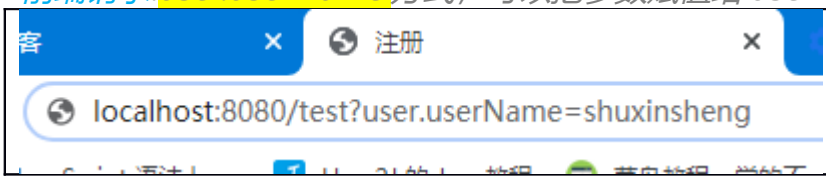
```java
public class User {
    //初始值为访客
    @Value("1")
    private int id;
    private String userName;

    public void setUserName(String userName)
        this.userName = userName;
    }
    public String getUserName() {
        return userName;
    }
```

处理器：

```java
@RequestMapping("/test")
public void test(Comment comment){
    System.out.println(comment);
}
```

前端请求:user.userName 方式，可以把参数赋值给 user bean 中的 userName 属性



4 多个同名参数

当传递的参数有多个同名参数时

```html
<tr>
    <td>爱好</td>
    <td>
    <input type="checkbox" name="like" value="zq"/>足球
    <input type="checkbox" name="like" value="lq"/>篮球
    <input type="checkbox" name="like" value="pq"/>排球
    <input type="checkbox" name="like" value="qq"/>铅球
    </td>
</tr>
```

如果形参中直接指定 String 类型，则 SpringMVC 会以逗号分隔保存到该形参中
如果形参中指定 String[] 数组类型，则 SpringMVC 会保存到该数组中

5 请求参数中文乱码问题

5.A SpringMVC 提供了过滤器用来解决全站 POST 提交乱码

这种方式只能解决 POST 提交的乱码，对 GET 方式提交的乱码无效！

```
<!-- 配置SpringMVC乱码解决过滤器 -->
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</
    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

5.B Get 提交乱码(手动编解码)

```
username = new String(username.getBytes("iso8859-1"),"utf-8");
```

5.C 也可以直接修改 Tomcat 中连接器的配置来使 tomcat 默认采用指定编码处理请求参数

但这种方式不建议大家使用，因为生产环境下不一定允许修改此项

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" URIEncoding="UTF-8"/>
```

6 日期数据处理

在处理器方法上，如果指定 Date 类型的形参时，SpringMVC 无法自动封装，需要手动注册适配器来指定转换方式

```java
@InitBinder
public void binder(ServletRequestDataBinder dataBinder){
    dataBinder.registerCustomEditor(Date.class
            ,new CustomDateEditor(
                    //表示将请求参数的字符串按该格式转为Date
                    new SimpleDateFormat( pattern: "yyyy-MM-dd")
                    , allowEmpty: true));
}
@RequestMapping("/test")
public void test(Date createDate){
    System.out.println(createDate);
}
```

7 SpringMVC 文件上传

7.A 准备文件上传表单

文件上传表单必须满足如下三个条件

a. 文件上传项必须有 name 属性
b. 表单必须是 Post 提交
c. 表单必须是 enctype="multipart/form-data"

表单：

```jsp
<form action="${pageContext.request.contextPath }/hello8.action"
    method="POST" enctype="multipart/form-data">

 <tr>
     <td>文件</td>
     <td><input type="file" name="fx"/></td>
 </tr>
```

7.B SpringMVC 配置文件中配置文件上传工具

```xml
<!-- 配置文件上传处理器bean -->
<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolve
    <property name="maxUploadSize" value="1024"></property>
</bean>
```

必须是这个id必须是这个类

配置上传文件时的参数

7.C 在处理器中实现文件上传

处理器可以接收一个 MutilpartFile 形参，形参名与表单字段 name 一致，或者通过 @RequestParam 指定

该参数可以获取文件的输入流、相关信息，或者直接调用 transferTo(File file)保存到本地

```
@RequestMapping("/test")
public void test(MultipartFile file) throws IOException {
    System.out.println(file.getSize());
    System.out.println(file.getOriginalFilename());
    file.transferTo(new File( pathname: "D:/"+file.getOriginalFilename()));
}
```

8 RESTFul 风格的请求参数处理

8.A 可以将参数放到请求的路径中，然后处理器获取路径的值作为参数

普通 get 请求：

```
Url:localhost/XXXX/addUser.action?name=tom&age=18
```

RESTFul 风格的请求：

```
Url:localhost/XXXX/addUser/tom/18.action
```

8.B 在路径中用{ }指定参数名 处理器形参用@PathVariable 指定

```
@RequestMapping("/test/{userName}.action")
public void test(@PathVariable String userName) {
    System.out.println(userName);
}
```

8.C 访问：最终获取到的就是值 shuxinsheng

六 SpringMVC 资源跳转

SpringMVC 的资源跳转不会匹配视图解析器的前缀和后缀

1 请求转发

1.A 传统方式实现请求转发

```
@RequestMapping("/test01.action")
public void test01(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

request.getRequestDispatcher("/index.jsp").forward(request,res
ponse);
}
```

1.B springmvc 方式实现请求转发

可以通过返回 forward:/xxxx.xxx 格式的字符串表明要转发到指定地址

```
@RequestMapping("/test02.action")
public String test02(){
    return "forward:/index.jsp";
}
```

2 请求重定向

2.A 传统方式实现请求重定向

```
@RequestMapping("/test01.action")
public void test01(HttpServletRequest request,HttpServletResponse
response) throws IOException {
    response.sendRedirect(request.getContextPath()+"/index.jsp");
}
```

2.B SpringMVC 方式实现请求重定向

可以通过返回 redirect:/xxxx.xxx 格式的字符串表明要重定向到指定地址

通过这种方式实现请求重定向时不用在路径前写应用名，SpringMVC 会自动拼接应用名

```
@RequestMapping("/test02.action")
public String test02(){
    return "redirect:/index.jsp";
}
```

3 定时刷新

SpringMVC 中没有提供实现定时刷新的便捷方式，只能用传统方式实现定时刷新

```
@RequestMapping("/test01.action")
public void test01(HttpServletRequest request,HttpServletResponse response) throws
IOException {
    response.setContentType("text/html;charset=utf-8");
    response.getWriter().write("注册成功，3 秒后回到主页..");
response.setHeader("refresh","3;url="+request.getContextPath()+"/index.jsp");
}
```

七 域的使用

1 ServletRequest(request)域

1.A 传统方式

处理器方法上传递 *HttpServletRequest* 类型的形参，然后通过该参数写入或读取数据

1.B SpringMVC 方式

写入：通过在处理器方法上声明形参 Model，通过该参数，写入数据，则默认写入 reques 域中。注意：model 不能读取

```
public void test(Model model) {
    model.addAttribute( s: "name", o: "shuxinsheng");
}
```

读取:形参添加注解@RequestAttribute，自动注入

```
@RequestMapping("/test/{userName}.action")
public void test(@RequestAttribute  String name) {
    System.out.println(name);
}
```

2 HttpSession(session)域

2.A 传统方式

形参获取 HttpSession 参数，SpringMVC 自动注入，通过该参数操作 Session

2.B SpringMVC 方式

写入：传递 Model 参数，然后在类上使用@SessionAttributes 注解，可将写入到 Model 的参数自动写入到 session，这个注解有个 require 参数，要求是否必须拿到，如果为 true，则请求中没有该参数会报 404

注意：后面有个 s，并且该注解只能用于类上

```
@Controller
@RequestMapping("/my02")
@SessionAttributes("attr1x")
public class MyController02 {
    /**
     * 向session域中写入数据
     * 方式二：通过model+@SessionAttributes实现将数据写入session
     */
    @RequestMapping("/test04")
    public String test04(Model model) {
        model.addAttribute("attr1x",System.currentTimeMillis());
        return "my02test04";
    }
}
```

读取：可在方法参数上添加注解，可从 Session 域中获取数据

```
@RequestMapping("/test/{userName}.action")
public void test(@SessionAttribute String name) {
    System.out.println(name);
}
```

3 ServletContext 域

只能通过传统方式，在参数中传递 request 域，然后获取该域

```
@RequestMapping("/test/{userName}.action")
public void test(HttpServletRequest request) {
    ServletContext context = request.getServletContext();
    context.setAttribute( s: "attr1", System.currentTimeMillis());
}
```

八　其它注解

1　@ModelAttribute

1.A　使用在方法上

则被修饰的方法将会在当前类的任意 handler(处理器)方法执行之前执行，该方法的返回值会自动存入 mode 中供后续使用

```
@ModelAttribute("uname")
public String ma01(){
    System.out.println("ma01");
    return "zs";
}
```

自动将uname=zs
存入model中

1.B　使用在方法参数之前

则会从 model 中获取属性值赋值到被修饰的方法参数上

```
@RequestMapping("/test02.action")
public void test02(@ModelAttribute("uname") String uname){
    System.out.println(uname);
}
```

自动从model中获取uname对应的值
赋值到当前方法参数上

2　@CookieValue

用在控制器方法参数上

用来从 Cookie 中获取指定名称的 Cookie 值

```
@RequestMapping("/test01.action")
public void test01(@CookieValue("JSESSIONID") String v){
    System.out.println(v);
}
```

3 @RequestHeader

用在控制器方法参数上

用来获取指定名称请求头的值

```
@RequestMapping("/test01.action")
public void test01(@RequestHeader("User-Agent") String v){
    System.out.println(v);
}
```

九 异常处理

1 为当前 Controller 配置错误处理

只需在当前 Controller 方法上加上@ExceptionHandler 注解,方法可接收 Exception 参数

```
@ExceptionHandler
public ModelAndView exceptionHandler(Exception e){
    System.out.println("出异常了。。。"+e);
    ModelAndView mav = new ModelAndView();
    mav.addObject("exception", e);
    mav.setViewName("err");
    return mav;
}
```

2 注解方式配置全局的错误处理

专门定义一个类处理全局异常，类上需加上@ControllerAdvice 注解
方法上加上@ExceptionHandler 注解,方法可接收 Exception 参数

```java
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler
    public ModelAndView exceptionHandler(Exception e){
        System.out.println("出异常了。。。"+e);
        ModelAndView mav = new ModelAndView();
        mav.addObject("exception", e);
        mav.setViewName("err2");
        return mav;
    }
}
```

3 配置文件方式配置全局错误处理 (了解)

在 SpringMVC 配置文件中配置

```xml
<!-- 配置错误处理 -->
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <prop key="java.lang.Throwable">404</prop>
        </props>
    </property>
</bean>
```

十 返回数据

若方法返回值为 String，则默认返回的是视图名并匹配前缀后缀。

若想返回一段数据给客户端，则需要用注解@ResponseBody 修饰，或者在类上用@RestController 替代@ResponseBody 和@Controller

1 返回字符串数据

1.A 方式一：通过 *HttpServletResponse* 对象返回

```
@RequestMapping("/test01.action")
public void test01(HttpServletResponse resp) throws Exception {
    resp.setContentType("text/html;charset=utf-8");
    resp.getWriter().write("abcde");
    resp.getWriter().write("中国");
}
```

1.B 方式二：参数直接获取 PrintWriter 对象返回

```
@RequestMapping("/test.action")
public void test02(PrintWriter writer) throws Exception {
    writer.write("shuxinsheng");
}
```

1.C 方式三：通过@ResponseBody 返回

```
/**  向客户端直接返回数据 - 字符串
*    方式二：通过@ResponseBody 将返回值直接写入响应
*  其中可以通过@RequestMapping 的 produces 属性控制输出时的类型编码*/
@ResponseBody
@RequestMapping(value="/test.action",produces="text/html;charset=utf-8")
public String test () {
    return "舒新胜";
}
```

2 返回 JSON 数据

2.A 方式一：手动拼接 json 字符串

```
@ResponseBody
@RequestMapping(value="/test.action")
public String test04() {
```

```
    String json = "{id:'123',name:'shuxinsheng'}";
    return json;
}
```

2.B 方式二：通过配置@ResponseBody 利用内置的 jackson 将对象处理为 json 返回

需要导入处理 JSON 的 jar 包

```
@ResponseBody
@RequestMapping(value="/test.action",produces="application/json;charset=utf-8")
public User test05()  {
    User user = new User();
    return user;
}
```

十一　处理器方法支持的参数类型和返回值类型总结

```
/**
 * SPringMVC可以接收的参数类型
 */
@RequestMapping("/test01.action")
public void test01(
        HttpServletRequest request,//请求对象
        HttpServletResponse response,//响应对象
        HttpSession session,//会话对象
        WebRequest webRequest,//request对象和session对象的合体

        InputStream in,//request.getInputStream()
        OutputStream out,//response.getOutputStream()
        Reader reader,//request.getReader()
        Writer writer,//response.getWriter()

        @RequestParam("xx") String param,//从请求参数中获取数据
        @PathVariable("xx") String pv,//从路径中获取数据
        @CookieValue("xx") String cv,//从Cookie获取数据
        @RequestHeader("xx") String hv,//从请求头中获取数据
        @RequestAttribute("xx") String ra,//从request域中获取数据
        @SessionAttribute("xx") String sa,//从Session域中获取数据
        @ModelAttribute("xx") String ma,//从Model中获取数据

        Model model,//通过Model接收模型数据
        Map map,//通过Map接收模型数据
        ModelMap mm,//通过ModelMap接收模型数据

        User user,//自定义bean类型，直接封装请求参数到自定义bean

        MultipartFile fx,//实现文件上传时，接收上传文件

        Errors err,//数据校验时使用
        BindingResult br//数据校验时使用
){
}
```

```java
/**
 * SpringMVC可以返回的数据类型
 */
@RequestMapping("/test02.action")
//public ModelAndView test02(){//返回ModelAndView，其中封装模型和视图相关信息
//public View test02(){//返回View，其中视图相关信息
//public String test03(){//返回字符串作为视图名称
//public void test04(){//返回值为void，则使用默认视图名(当前控制器方法路径去除后缀后的名称)
//@ResponseBody
//public String test05(){//无论返回什么，都直接发送给浏览器，如果是对象，转为json后发送
public User test06(){
    return null;
}
//除以上情况之外的返回值，
//默认将会把返回的内容存入Model中，
//使用默认视图名(当前控制器方法路径去除后缀后的名称)
/**
```

1 支持的方法参数类型

1.a.i. HttpServletRequest

代表当前请求的对象

1. HttpServletResponse

代表当前响应的对象

i. HttpSession

代表当前会话的对象

i. WebRequest

SpringMVC 提供的对象，相当于是 request 和 session 的合体，可以操作这两个域中的属性。

i. InputStream OutputStream Reader Writer

代表 request 中获取的输入流和 response 中获取的输出流

i. 通过@PathVariable @RequestParam 声明的方法参数

@PathVariable 可以将请求路径的指定部分获取赋值给指定方法参数

@RequestParam 可以将指定请求参数赋值给指定方法参数

如果不写此注解，则默认会将同名的请求参数赋值给方法参数

i. 通过@RequestAttribute、@SessionAttribute 和@ModelAttribute 声明的方法参数

@RequestAttribute 从 request 域中获取数据

@SessionAttribute 从 Session 域中获取数据

@ModelAttribute 从 Model 中获取数据

i. 通过@CookieValue 和@RequestHeader 声明的方法参数

@CookieValue 可以将请求中的指定名称的 cookie 赋值给指定方法参数

@RequestHeader 可以将请求参数中的指定名称的头赋值给指定方法参数

i. Model 和 ModelMap 和 java.util.Map

向这些 Model ModelMap Map 中存入属性，相当于向模型中存入数据

i. Bean 类

SpringMVC 自动将请求参数封装到 bean

i. MultipartFile

实现文件上传功能时，接收上传的文件对象

i. Errors BindingResult

实现数据验证的参数

2 支持的返回值类型

a.i. ModelAndView

可以返回一个 ModelAndView 对象，在其中封装 Model 和 View 信息

a. *View

可以直接返回一个代表视图的 View 对象

i. 字符串

直接返回视图的名称

i. void

如果返回值类型是 void，则会自动返回和当前处理器路径名相同的视图名

i. 方法被@ResponseBody 修饰

当方法被@ResponseBody 修饰时，默认将返回的对象转为 json 写入输出

i. 除以上之外返回的任何内容都会被当做模型中的数据来处理，值为返回的数据，键为返回类型名首字母转小写，而返回的视图名等同于返回值为 void 的时的视图名