SpringMVC学习笔记(一)

内容包括：

1. SpringMVC执行流程
2. 注解开发`@controller`，`@RequestMapping`
3. 请求参数处理
4. 数据输出处理
5. 视图解析

参考视频：
[B站 尚硅谷雷丰阳大神的Spring、Spring MVC、MyBatis课程]

1. SpringMVC概述

MVC：

- **Model（模型）：** 数据模型，提供要展示的数据，：Value Object（数据Dao） 和 服务层（行为Service），提供数据和业务。
- **View（视图）：** 负责进行模型的展示，即用户界面
- **Controller（控制器）：** 调度员，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。

SpringMVC的特点：

- Spring为展现层提供的基于MVC设计理念的Web框架
- SpirngMVC通过一套MVC注解，让POJO成为处理请求的控制器，而无须实现任何接口
- 支持REST风格的URL请求
- 采用了松散耦合可拔插组件结构，扩展性和灵活性

2. HelloWorld

1) 导入依赖

spring-webmvc的maven依赖

```
    <dependencies>
        <!-- SpringWeb基础包-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>

        <!--        核心包-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>

       <!--        日志包-->
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.3</version>
        </dependency>

        <!--         注解支持包-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>
    </dependencies>
```

2) 配置web.xml， 注册DispatcherServlet

`DispatcherServlet`：前端控制器，负责请求分发。

要绑定Spring的配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">

    <!--注册DispatcherServlet,请求分发器（前端控制器）-->
    <servlet>
        <servlet-name>springDispatcherServlet</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!--绑定Spring配置文件-->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc-config.xml</param-value>
        </init-param>
        <!--启动级别为1，即服务器启动后就启动-->
        <!--值越小优先级越高，越先创建对象-->
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- /  拦截所有的请求；（不包括.jsp，jsp由Tomcat来处理），
        覆盖了父类的DispatcherServlet的pattern，静态资源被拦截。-->
    <!-- *.jsp 拦截jsp请求，覆盖了父类的JspServlet-->
    <!-- /* 拦截所有的请求；（包括.jsp，一旦拦截jsp页面就不能显示了）-->
    <servlet-mapping>
        <servlet-name>springDispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

3) Spring配置文件

Spring的配置文件Springmvc-config.xml。

1. 开启了包扫描，让指定包下的注解生效，由IOC容器统一管理
2. 配置了视图解析器 `InternalResourceViewResolver`，这里可以设置前缀和后缀，拼接视图名字

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

    <!--开启包扫描,让指定包下的注解生效,由IOC容器统一管理-->
    <context:component-scan base-package="com.xiao.controller"/>

    <!--配置视图解析器，拼接视图名字，找到对应的视图-->
    <bean id="internalResourceViewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value="/WEB-INF/page/"/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

4) 编写controller层

HelloController类：

1. @Controller：告诉Spirng这是一个控制器，交给IOC容器管理
2. @RequestMapping("/hello01")：/ 表示项目地址，当请求项目中的hello01时，返回一个/WEB-INF/page/success.jsp页面给前端

```
@Controller
public class HelloController {

    @RequestMapping("/hello01")
    public String toSuccess(){
        System.out.println("请求成功页面");
        return "success";
    }
    @RequestMapping("/hello02")
    public String toError() {
        System.out.println("请求错误页面");
        return "error";
    }
}
```

5) 编写跳转的jsp页面

项目首页 index.jsp，两个超链接，分别发出hello01和hello02的请求

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

  <a href="hello01">点这里去成功页面</a>
  <a href="hello02">点这里去失败页面</a>

  </body>
</html>
```

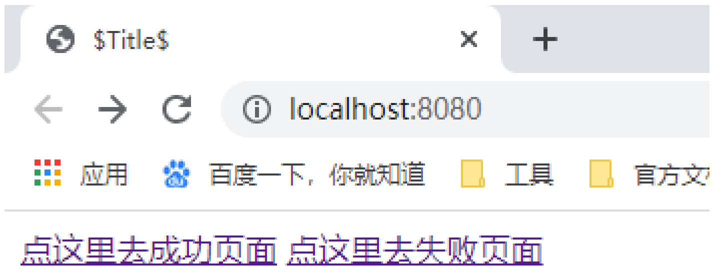
成功页面success.jsp和失败页面error.jsp，要注意文件的路径/WEB-INF/page/...jsp，与上面的保持一致

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>成功页面</title>
</head>
<body>
    <h1>这里是成功页面</h1>
</body>
</html>
```

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>错误页面</title>
</head>
<body>
    <h1>这里是错误页面</h1>
</body>
</html>
```

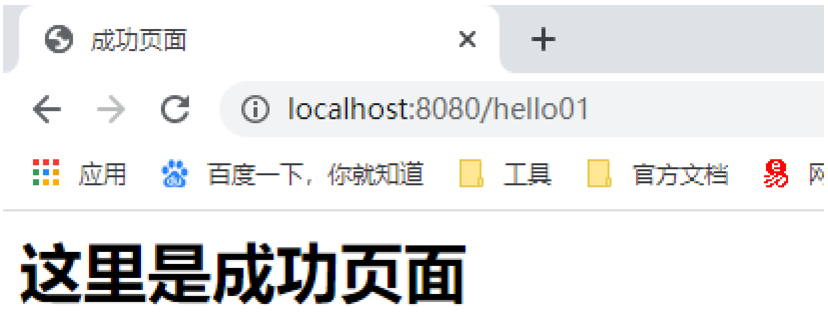
6) 访问

启动项目：



点这里去成功页面 点这里去失败页面

点击去成功页面，可以看到发出了/hello01请求，页面转发到/WEB-INF/page/success.jsp，控制台输出了请求成功页面。



这里是成功页面

3. 实现细节

3.1 运行流程

1. 客户端点击链接发送请求：http://localhost:8080/hello01；
2. 来到tomcat服务器；
3. SpringMVC的前端控制器收到所有请求；
4. 看请求地址和@RequestMapping标注的哪个匹配，来找到底使用哪个类的哪个方法来处理；
5. 前端控制器找到目标处理器类和目标方法，直接利用反射执行目标方法；
6. 方法执行完后有一个返回值，SpringMVC认为这个返回值就是要去的页面地址；
7. 拿到方法返回值后，视图解析器进行拼串得到完整的页面地址
8. 得到页面地址，前端控制器帮我们转发到页面

3.2 url映射

> RequestMapping

01 标注在方法上

告诉SpringMVC这个方法用来处理什么请求。

`@RequestMapping("/hello01")`中的`/`可以省略，就是默认从当前项目下开始。

02 标注在类上

表示为当前类中的所有方法的请求地址，指定一个基准路径。toSuccess()方法处理的请求路径是`/haha/hello01`。

```
@Controller
@RequestMapping("/haha")
public class HelloController {

    @RequestMapping(value = "/hello01")
    public String toSuccess(){
        System.out.println("请求成功页面");
        return "success";
    }
}
```

03 规定请求方式

method属性规定请求方式，默认是所求请求方式都行。method = RequestMethod.GET，method = RequestMethod.POST。

如果方法不匹配会报：**HTTP Status 405 错误 – 方法不被允许**

```
@RequestMapping(value = "/hello01",method = RequestMethod.GET)
public String toSuccess(){
    System.out.println("请求成功页面");
    return "success";
}
```

组合用法

- @GetMapping 等价于 @RequestMapping(method =RequestMethod.GET)
- @PostMapping
- @PutMapping
- @DeleteMapping
- @PatchMapping

04 规定请求参数

params属性规定请求参数。会造成错误：**HTTP Status 400 – 错误的请求**

不携带该参数，表示参数值为null；携带了不给值表示参数值是空串

```
//必须携带username参数
@RequestMapping(value = "/hello03",params ={"username"})
//必须不携带username参数
@RequestMapping(value = "/hello03",params ={"! username"})
//必须携带username参数，且值必须为123
@RequestMapping(value = "/hello03",params ={"username=123"})
//username参数值必须不为123，不携带或者携带了不是123都行
@RequestMapping(value = "/hello03",params ={"username=! 123"})
//username参数值必须不为123，不携带password，携带page
@RequestMapping(value = "/hello03",params ={"username=!
123","page","!password"})
```

05 规定请求头

headers属性规定请求头。其中User-Agent：浏览器信息

谷歌浏览器：User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.97 Safari/537.3

06 Ant风格URL

URL地址可以写模糊的通配符，模糊和精确多个匹配情况下精确优先。

? ：**替代任意一个字符**

```
@RequestMapping( "/hello0?") /
```

*：**替代任意多个字符或一层路径**

```
@RequestMapping( "/hello0*")   //任意多个字符
@RequestMapping( "/a/*/hello01")  //一层路径
```

```
    @RequestMapping(value = "/test/*/a")
    public String myMethodTest01() {
        System.out.println("post01");
        return "success";
    }
    // test/[^\/]+/b ->post01
    // /test/*/b ->post02
    @RequestMapping(value = "/test/**/a")
    public String myMethodTest02() {
        System.out.println("post02");
        return "success";
    }
```

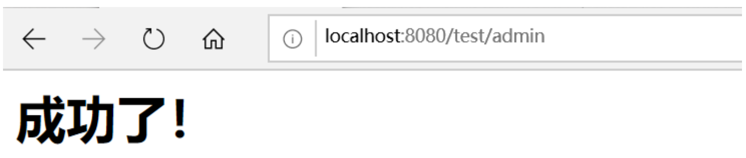
**：替代任意多层路径

```
@RequestMapping( "/a/**/hello01")  //任意多层路径
```

07 PathVariable

可以用/test/{paramsName1}/{paramsName2}来获取Url上传的参数值

```
@RequestMapping(value = "/test/{id}", method = RequestMethod.GET)
    public String myMethodTest03(@PathVariable("id") String id) {
        System.out.println(id);
        return "success";
    }
```


localhost:8080/test/admin

成功了！

```
admin  //打印
```

3.3 Spring配置文件的默认位置

默认位置是 /WEB-INF/xxx-servlet.xml，其中xxx是自己在web.xml文件中配置的servlet-name属性。

`dispatcherServlet-servlet.xml`

当然也可以手动指定文件位置。

```
<servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:applicationContext.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
</servlet>
```

3.5 url-pattern

/ 拦截所有的请求，不拦截jsp

/* **拦截所有的请求**，包括*.jsp，一旦拦截jsp页面就不能显示了。. jsp是tomcat处理的事情

看Tomcat的配置文件web.xml中，有DefaultServlet和JspServlet，

- DefaultServlet是Tomcat中处理静态资源的，Tomcat会在服务器下找到这个资源并返回。如果我们自己配置`url-pattern=/`，相当于禁用了Tomcat服务器中的DefaultServlet，这样如果请求静态资源，就会去找前端控制器找@RequestMapping，**这样静态资源就不能访问了。**解决办法：

  ```
  <!-- 告诉Spring MVC自己映射的请求就自己处理，不能处理的请求直接交给tomcat -->
  <mvc:default-servlet-handler />
  <!--开启MVC注解驱动模式，保证动态请求和静态请求都能访问-->
  <mvc:annotation-driven/>
  ```

- JspServlet，保证了jsp可以正常访问

```
<servlet>
    <servlet-name>default</servlet-name>
    <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>0</param-value>
    </init-param>
    <init-param>
        <param-name>listings</param-name>
        <param-value>false</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

    <servlet-mapping>
        <servlet-name>default</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>


<servlet>
```

```
     <servlet-name>jsp</servlet-name>
     <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
     <init-param>
         <param-name>fork</param-name>
         <param-value>false</param-value>
      </init-param>
      <init-param>
          <param-name>xpoweredBy</param-name>
          <param-value>false</param-value>
      </init-param>
      <load-on-startup>3</load-on-startup>
</servlet>

    <servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>*.jsp</url-pattern>
        <url-pattern>*.jspx</url-pattern>
    </servlet-mapping>
```

4. REST风格

4.1 概述

REST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。其强调HTTP应当以资源为中心，并且规范了URI的风格；规范了HTTP请求动作（GET/PUT/POST/DELETE/HEAD/OPTIONS）的使用，具有对应的语义。

- 资源（Resource）：网络上的一个实体，每种资源对应一个特定的URI，即URI为每个资源的独一无二的识别符；
- 表现层（Representation）：把资源具体呈现出来的形式，叫做它的表现层。比如txt、HTML、XML、JSON格式等；
- 状态转化（State Transfer）：每发出一个请求，就代表一次客户端和服务器的一次交互过程。GET用来获取资源，POST用来新建资源，PUT用来更新资源，DELETE用来删除资源。

在参数上使用 @PathVariable 注解，可以获取到请求路径上的值，也可以写多个

```
    @RequestMapping(value = "/hello04/username/{id}")
    public String test2(@PathVariable("id") int id){
        System.out.println(id);
        return "success";
    }
```

4.2 页面上发出PUT请求

对一个资源的增删改查用请求方式来区分：

- /book/1 GET：查询1号图书
- /book/1 DELETE：删除1号图书
- /book/1 PUT：修改1号图书
- /book POST：新增图书

页面上只能发出GET请求和POST请求。将POST请求转化为put或者delete请求的步骤：

1. 把前端发送方式改为post 。

2. 在web.xml中配置一个filter：HiddenHttpMethodFilter过滤器
3. 必须携带一个键值对，key=_method, value=put或者delete

```
<!--这个过滤器的作用 ：就是将post请求转化为put或者delete请求-->
<filter>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-
class>
</filter>
<filter-mapping>
    <filter-name>HiddenHttpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<form action="hello03" method="post">
  <input type="hidden" name="_method" value="delete">
  <input type="submit" name="提交">
</form>
```

高版本Tomcat会出现问题：JSPs only permit GET POST or HEAD，在页面上加上异常处理即可

```
<%@ page contentType="text/html;charset=UTF-8" language="java"
 isErrorPage="true" %>
1
```

5 请求参数处理

5.1 传入参数

1. 如果提交的参数名称和处理方法的参数名一致，则无需处理，直接使用

提交数据 : http://localhost:8080/hello05?username=zhangsan，控制台会输出zhangsan

```
@RequestMapping("/hello05")
public String test03(String username) {
    System.out.println(username);
    return "success";
}
```

2. 提交的参数名称和处理方法的参数名不一致，使用@RequestParam注解

注解 `@RequestParam` 可以获取请求参数，默认必须携带该参数，也可以指定 `required=false`，和没携带情况下的默认值 `defaultValue`

```
@RequestMapping("/hello05")
public String test03(@RequestParam(value = "username",required = false,
defaultValue ="hehe" ) String name) {
    System.out.println(name);
    return "success";
}
```

还有另外两个注解：

- `@RequestHeader`：获取请求头中的信息，比如User-Agent：浏览器信息

```
@RequestMapping("/hello05")
public String test03(@RequestHeader("User-Agent" ) String name) {
    System.out.println(name);
    return "success";
}
```

- `@CookieValue`：获取某个cookie的值

```
@RequestMapping("/hello05")
public String test03(@CookieValue("JSESSIONID" ) String name) {
     System.out.println(name);
     return "success";
}
```

5.2 传入一个对象

传入POJO，SpringMVC会自动封装，**提交的表单域参数必须和对象的属性名一致，否则就是null，请求没有携带的字段，值也会是null。**同时也还可以级联封装。

新建两个对象User和Address：

```
public class User {
    private String username;
    private Integer age;
    private Address address;
    //....
}
123456
public class Address {
    private String name;
    private Integer num;
        //....
}
12345
```

前端请求：

```
<form action="hello06" method="post">
    姓名： <input type="text" name="username"> <br>
    年龄： <input type="text" name="age"><br>
    地址名：<input type="text" name="address.name"><br>
    地址编号：<input type="text" name="address.num"><br>
    <input type="submit" name="提交">
</form>
```

后端通过对象名也能拿到对象的值，没有对应的值则为null

```
@RequestMapping("/hello06")
public String test03(User user) {
    System.out.println(user);
    return "success";
}
```

5.3 传入原生ServletAPI

处理方法还可以传入原生的ServletAPI：

```
@RequestMapping("/hello07")
public String test04(HttpServletRequest request, HttpSession session) {
    session.setAttribute("sessionParam","我是session域中的值");
    request.setAttribute("reqParam","我是request域中的值");
    return "success";
}
```

通过EL表达式获取到值，`${requestScope.reqParam}`：

```
<%@ page contentType="text/html;charset=UTF-8" language="java"
 isErrorPage="true" %>
<html>
<head>
    <title>成功页面</title>
</head>

<body>

<h1>这里是成功页面</h1>
${requestScope.reqParam}
${sessionScope.sessionParam}
</body>
</html>
```

5.4 乱码问题

一定要放在在其他Filter前面。

```
<filter>
   <filter-name>encoding</filter-name>
   <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-
class>
    <!--解决请求乱码-->
   <init-param>
       <param-name>encoding</param-name>
       <param-value>utf-8</param-value>
   </init-param>
    <!--解决响应乱码-->
   <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>encoding</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<!--在Tomcat的server.xml中的8080处 URLEncoding="UTF-8"-->
```

6. 数据输出

6.1 Map、Model、ModelMap

实际上都是调用的 **BindingAwareModelMap**(隐含模型)，将数据放在**请求域(requestScope)中**进行转发，用EL表达式可以取出对应的值。

```
/**
 * SpringMVC除过在方法上传入原生的request和session外还能怎么样把数据带给页面
 *
 * 1）、可以在方法处传入Map、或者Model或者ModelMap。
 *      给这些参数里面保存的所有数据都会放在请求域中。可以在页面获取
 *   关系：
 *      Map，Model，ModelMap：最终都是BindingAwareModelMap在工作；
 *      相当于给BindingAwareModelMap中保存的东西都会被放在请求域中；
 *
 *      Map(interface(jdk))      Model(interface(spring))
 *          ||                          //
 *          ||                         //
 *          \/                        //
 *      ModelMap(class)               //
 *                  \\              //
 *                   \\            //
 *                  ExtendedModelMap
 *                          ||
 *                          \/
 *                  BindingAwareModelMap
 *
 * 2）、方法的返回值可以变为ModelAndView类型；
 *          既包含视图信息（页面地址）也包含模型数据（给页面带的数据）；
 *          而且数据是放在请求域中；
 *          request、session、application；
 *
 *
 * @author lfy
 *
 */
```

- Map

```
@RequestMapping("/Api2")
    public String api2(Map<String,Object> map){
        map.put("msg","hello");
        return "map";
    }
```

- Model

```
@RequestMapping("/Api3")
    public String api3(Model model){
        model.addAttribute("msg","hello2");
        return "map";
    }
```

- ModelMap

```
@RequestMapping("/Api4")
    public String api4(ModelMap modelMap){
        modelMap.addAttribute("msg","hello3");
        return "map";
    }
```

map页面:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>

<head>
    <title>Title</title>
</head>

<body>

pageScope:  ${pageScope.msg}

requestScope :   ${requestScope.msg}

sessionScope:     ${sessionScope.msg}

applicationScope:   ${applicationScope.msg}

</body>
</html>
```

【补充】jsp的4个作用域 pageScope、requestScope、sessionScope、applicationScope的区别:

- **page**指当前页面有效。在一个jsp页面里有效
- **request** 指在一次请求的全过程中有效，即从http请求到服务器处理结束，返回响应的整个过程，存放在**HttpServletRequest**对象中。在这个过程中可以使用forward方式跳转多个jsp。在这些页面里都可以使用这个变量。
- **Session**是用户全局变量，在整个会话期间都有效。只要页面不关闭就一直有效（或者直到用户一直未活动导致会话过期，默认session过期时间为30分钟，或调用HttpSession的invalidate()方法）。存放在HttpSession对象中
- **application**是程序全局变量，对每个用户每个页面都有效。存放在ServletContext对象中。它的存活时间是最长的，如果不进行手工删除，它们就一直可以使用

6.2 ModelAndView

返回一个模型视图对象ModerAndView， 既包含视图信息(页面地址)，也包含模型数据(给页面带的数据)

```
@RequestMapping("/hello04")
public ModelAndView test04 (){
   //新建一个模型视图对象，也可以直接传入名字
   ModelAndView mv = new ModelAndView();
   //封装要显示到视图中的数据
   //相当于req.setAttribute("msg",HelloWorld!);
   mv.addObject("msg","HelloWorld!");
   //设置视图的名字，相当于之前的return "success";
   mv.setViewName("success");
   return mv;
}
```

6.3 @SessionAttributes

给Session域中携带数据使用注解`@SessionAttributes`，只能标在类上，value属性指定key，type可以指定保存类型。这个注解会引发异常**一般不用，就用原生API**

`@SessionAttributes(value = "msg")`：表示给BindingAwareModelMap中保存key为msg的数据时，在session中也保存一份；

`@SessionAttributes(types = {String.class})`：表示只要保存String类型的数据时，给session中也放一份。

```
//表示给BindingAwareModelMap中保存key为msg的数据时，在session中也保存一份
@SessionAttributes(value = "msg")
@Controller
public class outputController {
    @RequestMapping("/hello01")
    public String test01 (Map<String,Object> map){
        map.put("msg","HelloWorld!");
        return "success";
    }
}
```

6.4 @ModelAttribute

```
ModelAttribute：
使用场景：
1）、页面：form提交更新
2）、dao：全字段更新。没带的字段会在数据库中更新为null；

/**
 * 测试ModelAttribute注解；
 * 使用场景：书城的图书修改为例；
 * 1）页面端；
 *      显示要修改的图书的信息，图书的所有字段都在
 * 2）servlet收到修改请求，调用dao；
 *      String sql="update bs_book set title=?,
 *                  author=?,price=?,
 *                  sales=?,stock=?,img_path=?
 *              where id=?";
 * 3）实际场景？
 *      并不是全字段修改；只会修改部分字段，以修改用户信息为例；
 *      username  password  address;
 *      1）、不修改的字段可以在页面进行展示但是不要提供修改输入框；
 *       2）、为了简单，Controller直接在参数位置来写Book对象
 *       3）、SpringMVC为我们自动封装book；（没有带的值是null）
 *       4）、如果接下来调用了一个全字段更新的dao操作；会将其他的字段可能变为null；
 *           sql = "update bs_book set"
 *           if(book.getBookName()){
 *               sql +="bookName=?,"
 *           }
 *           if(book.getPrice()){
 *               sql +="price=?"
 *           }
 *
 * 4）、如何能保证全字段更新的时候，只更新了页面携带的数据；
 *       1）、修改dao；代价大？
 *       2）、Book对象是如何封装的？
 *           1）、SpringMVC创建一个book对象，每个属性都有默认值，bookName就是null；
 *               1、让SpringMVC别创建book对象，直接从数据库中先取出一个id=100的book对象的
信息
 *               2、Book [id=100, bookName=西游记, author=张三, stock=12, sales=32,
price=98.98]
 *
 *           2）、将请求中所有与book对应的属性一一设置过来；
 *               3、使用刚才从数据库取出的book对象，给它 的里面设置值；（请求参数带了哪些值就
覆盖之前的值）
 *               4、带了的字段就改为携带的值，没带的字段就保持之前的值
 *           3）、调用全字段更新就有问题；
 *               5、将之前从数据库中查到的对象，并且封装了请求参数的对象。进行保存；
 *
 * @author lfy
 */
```

方法入参标注该注解后，入参的对象就会放到数据模型中，会提前于控制方法先执行，并发方法允许的结果放在隐含模型中。

处理这样的场景：

前端传来数据，SpringMVC自动封装成对象，实际上是创建了一个对象，每个属性都有默认值，然后将请求参数中对应是属性设置过来，但是如果没有的值将会是null，如果拿着这个数据去更新数据库，会造成其他字段也变为null。因此希望使用`@ModelAttribute`，会在目标方法执行前先做一些处理

```
@ModelAttribute
public void  myModelAttribute(ModelMap modelMap){
    System.out.println("modelAttribute方法执行了");
    //提前做一些处理
    User user = new User("zhangsan",20);
    //保存一个数据到BindingAwareModelMap中，目标方法可以从中取出来
    modelMap.addAttribute("user",user);
}

@RequestMapping("/hello05")
public void  test05(@ModelAttribute("user") User user){
    System.out.println("目标方法执行了");
    //在参数上加上@ModelAttribute注解，可以拿到提前存入的数据
    System.out.println(user);

}
```

6.5 @ResponseBody

在控制器类中，在方法上使用**@ResponseBody**注解可以不走视图解析器，如果返回值是字符串，那么直接将字符串写到客户端；如果是一个对象，会将对象转化为JSON串，然后写到客户端。

或者在类上加 **@RestController**注解，可以让类中的所有方法都不走视图解析器，直接返回JSON字符串

7. SpringMVC执行流程源码

7.0 SpringMVC的九大组件

- multipartResolver：文件上传解析器
- localeResolver：区域信息解析器，和国际化有关
- themeResolver：主题解析器
- handlerMappings：handler的映射器
- handlerAdapters：handler的适配器
- handlerExceptionResolvers：异常解析功能
- viewNameTranslator：请求到视图名的转换器
- flashMapManager：SpringMVC中允许重定向携带数据的功能
- viewResolvers：视图解析器

```
	 /** 文件上传解析器*/
	private MultipartResolver multipartResolver;
	/** 区域信息解析器；和国际化有关 */
	private LocaleResolver localeResolver;
	/** 主题解析器；强大的主题效果更换 */
	private ThemeResolver themeResolver;
	/** Handler映射信息；HandlerMapping */
	private List<HandlerMapping> handlerMappings;
	/** Handler的适配器 */
	private List<HandlerAdapter> handlerAdapters;
	/** SpringMVC强大的异常解析功能；异常解析器 */
	private List<HandlerExceptionResolver> handlerExceptionResolvers;
	/**  */
	private RequestToViewNameTranslator viewNameTranslator;
	/** FlashMap+Manager：SpringMVC中运行重定向携带数据的功能 */
	private FlashMapManager flashMapManager;
	/** 视图解析器；  */
	private List<ViewResolver> viewResolvers;
```

onRefresh()->initStrategies() DispatcherServlet中：

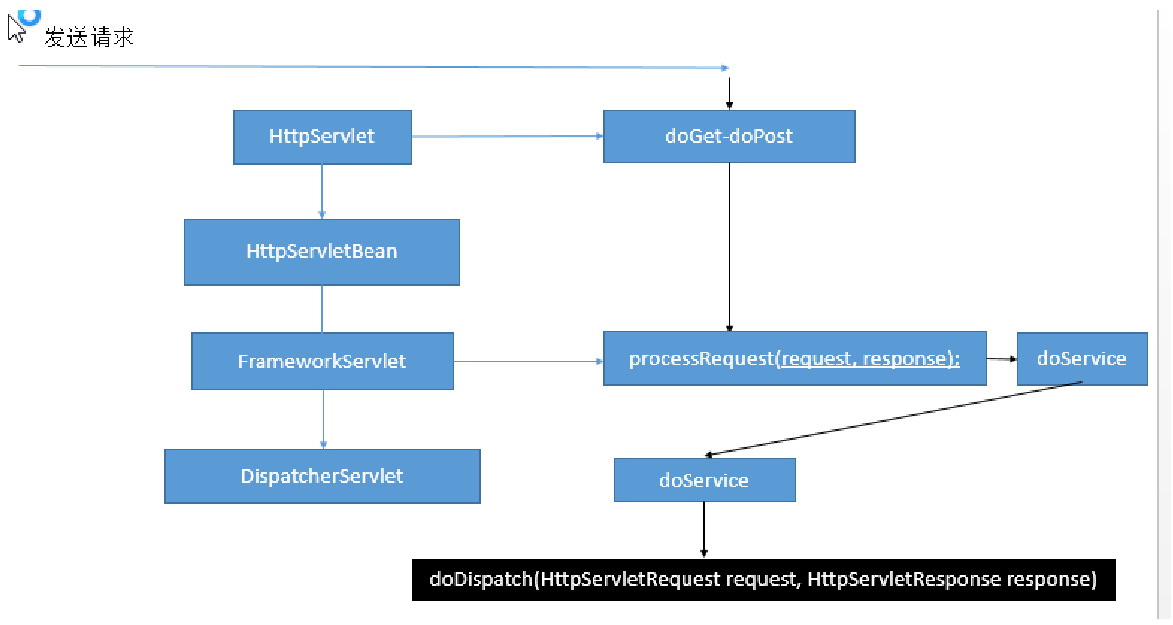
```java
protected void initStrategies(ApplicationContext context) {
        initMultipartResolver(context);
        initLocaleResolver(context);
        initThemeResolver(context);
        initHandlerMappings(context);
        initHandlerAdapters(context);
        initHandlerExceptionResolvers(context);
        initRequestToViewNameTranslator(context);
        initViewResolvers(context);
        initFlashMapManager(context);
    }
```

例：初始化HandlerMapping

```java
private void initHandlerMappings(ApplicationContext context) {
        this.handlerMappings = null;

        if (this.detectAllHandlerMappings) {
            // Find all HandlerMappings in the ApplicationContext, including
ancestor contexts.
            Map<String, HandlerMapping> matchingBeans =
                    BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
HandlerMapping.class, true, false);
            if (!matchingBeans.isEmpty()) {
                this.handlerMappings = new ArrayList<HandlerMapping>
(matchingBeans.values());
                // We keep HandlerMappings in sorted order.
                OrderComparator.sort(this.handlerMappings);
            }
        }
        else {
            try {
                HandlerMapping hm = context.getBean(HANDLER_MAPPING_BEAN_NAME,
HandlerMapping.class);
                this.handlerMappings = Collections.singletonList(hm);
            }
            catch (NoSuchBeanDefinitionException ex) {
                // Ignore, we'll add a default HandlerMapping later.
            }
        }

        // Ensure we have at least one HandlerMapping, by registering
        // a default HandlerMapping if no other mappings are found.
        if (this.handlerMappings == null) {
            this.handlerMappings = getDefaultStrategies(context,
HandlerMapping.class);
            if (logger.isDebugEnabled()) {
                logger.debug("No HandlerMappings found in servlet '" +
getServletName() + "': using default");
            }
        }
    }
```

组件的初始化：　有些组件在容器中是使用类型找的，有些组件是使用id找的；
去容器中找这个组件，如果没有找到就用默认的配置；

7.1 前端控制器DisatcherServlet


发送请求
HttpServlet
doGet-doPost
HttpServletBean
FrameworkServlet
processRequest(request, response);
doService
DispatcherServlet
doService
doDispatch(HttpServletRequest request, HttpServletResponse response)

7.2 SpringMVC执行流程

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        boolean multipartRequestParsed = false;
        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
        try {
            ModelAndView mv = null;
            Exception dispatchException = null;
            try {

                //1、检查是否文件上传请求

                processedRequest = checkMultipart(request);
                multipartRequestParsed = processedRequest != request;

                // Determine handler for the current request.
                //2、根据当前的请求地址找到那个类能来处理；

                mappedHandler = getHandler(processedRequest);

                //3、如果没有找到哪个处理器（控制器）能处理这个请求就404，或者抛异常

                if (mappedHandler == null || mappedHandler.getHandler() == null)
{
                    noHandlerFound(processedRequest, response);
                    return;
                }

                // Determine handler adapter for the current request.
                //4、拿到能执行这个类的所有方法的适配器；（反射工
AnnotationMethodHandlerAdapter）

                HandlerAdapter ha =
getHandlerAdapter(mappedHandler.getHandler());
```

```
                // Process last-modified header, if supported by the handler.

                String method = request.getMethod();
                boolean isGet = "GET".equals(method);
                if (isGet || "HEAD".equals(method)) {
                    long lastModified = ha.getLastModified(request,
mappedHandler.getHandler());
                    if (logger.isDebugEnabled()) {
                        String requestUri =
urlPathHelper.getRequestUri(request);
                        logger.debug("Last-Modified value for [" + requestUri +
"] is: " + lastModified);
                    }
                    if (new ServletWebRequest(request,
response).checkNotModified(lastModified) && isGet) {
                        return;
                    }
                }
                if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                    return;
                }
                try {

                    // Actually invoke the handler.处理（控制）器的方法被调用
                    //控制器（Controller），处理器（Handler）
                    //5、适配器来执行目标方法；
                    //将目标方法执行完成后的返回值作为视图名，设置保存到ModelAndView中
                    //目标方法无论怎么写，最终适配器执行完成以后都会将执行后的信息封装成
ModelAndView

                    mv =
ha.handle(processedRequest,response,mappedHandler.getHandler());
                } finally {
                    if (asyncManager.isConcurrentHandlingStarted()) {
                        return;
                    }
                }
                applyDefaultViewName(request, mv);//如果没有视图名设置一个默认的视图
名；
                mappedHandler.applyPostHandle(processedRequest, response, mv);
            } catch (Exception ex) {
                dispatchException = ex;
            }

            //转发到目标页面；
            //6、根据方法最终执行完成后封装的ModelAndView；
            //转发到对应页面，而且ModelAndView中的数据可以从请求域中获取
            processDispatchResult(processedRequest, response, mappedHandler,
                                  mv, dispatchException);
        } catch (Exception ex) {
            triggerAfterCompletion(processedRequest, response, mappedHandler,
ex);
        } catch (Error err) {
            triggerAfterCompletionWithError(processedRequest, response,
mappedHandler, err);
        } finally {
            if (asyncManager.isConcurrentHandlingStarted()) {
                // Instead of postHandle and afterCompletion

mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
                return;
            }
            // Clean up any resources used by a multipart request.
            if (multipartRequestParsed) {
                cleanupMultipart(processedRequest);
            }
        }
    }
```

总体概览

1. **用户发出请求，DispatcherServlet接收请求并拦截请求。**
2. **调用doDispatch()方法进行处理：**
    1. getHandler()：根据当前请求地址中找到能处理这个请求的目标处理器类(处理器)；
        - 根据当前请求在HandlerMapping中找到这个请求的映射信息，获取到目标处理器类
        - mappedHandler = getHandler(processedRequest);
    2. getHandlerAdapter()：根据当前处理器类找到能执行这个处理器方法的适配器；
        - 根据当前处理器类，找到当前类的HandlerAdapter（适配器）
        - HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
    3. 使用刚才获取到的适配器(AnnotationMethodHandlerAdapter)执行目标方法；
        - mv = ha.handle(processedRequest,response,mappedHandler.getHandler());
    4. 目标方法执行后，会返回一个ModerAndView对象
        - **mv** = ha.handle(processedRequest,response,mappedHandler.getHandler());
    5. 根据ModerAndView的信息转发到具体页面，并可以在请求域中取出ModerAndView中的模型数据
        - processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);

> HandlerMapping为处理器映射器，保存了每一个处理器能处理哪些请求的映射信息，handlerMap
>
> HandlerAdapter为处理器适配器，能解析注解方法的适配器，其按照特定的规则去执行Handler

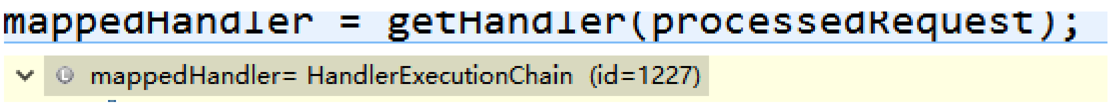
具体细节

步骤一：

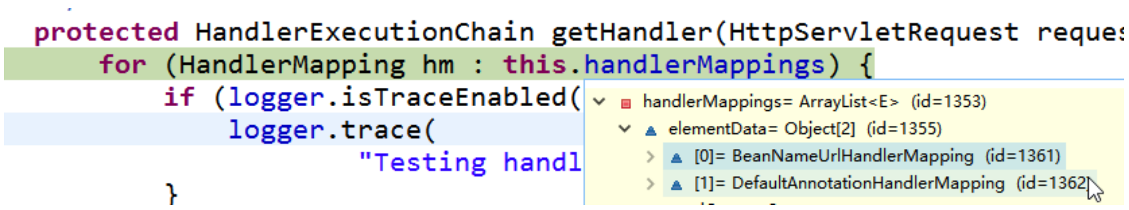
getHandler():

> **怎么根据当前请求就能找到哪个类能来处理？**

- getHandler()会返回目标处理器类的执行链


mappedHandler = getHandler(processedRequest);
mappedHandler= HandlerExecutionChain (id=1227)

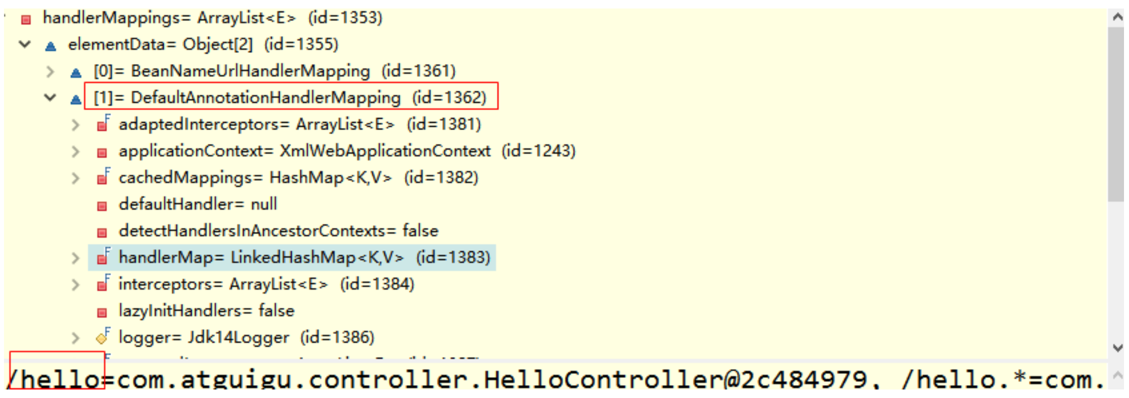
- HandlerMapping：处理器映射：他里面保存了每一个处理器能处理哪些请求的映射信息



```
protected HandlerExecutionChain getHandler(HttpServletRequest reques
    for (HandlerMapping hm : this.handlerMappings) {
        if (logger.isTraceEnabled(
            logger.trace(
                    "Testing handl
        }
```

```
handlerMappings= ArrayList<E>  (id=1353)
  elementData= Object[2]  (id=1355)
    [0]= BeanNameUrlHandlerMapping  (id=1361)
    [1]= DefaultAnnotationHandlerMapping  (id=1362)
```

- handlerMap：ioc容器启动创建Controller对象的时候扫描每个处理器都能处理什么请求，保存在HandlerMapping的handlerMap属性中；下一次请求过来，就来看哪个HandlerMapping中有这个请求映射信息就行了



```
handlerMappings= ArrayList<E>  (id=1353)
  elementData= Object[2]  (id=1355)
    [0]= BeanNameUrlHandlerMapping  (id=1361)
    [1]= DefaultAnnotationHandlerMapping  (id=1362)
      adaptedInterceptors= ArrayList<E>  (id=1381)
      applicationContext= XmlWebApplicationContext  (id=1243)
      cachedMappings= HashMap<K,V>  (id=1382)
      defaultHandler= null
      detectHandlersInAncestorContexts= false
      handlerMap= LinkedHashMap<K,V>  (id=1383)
      interceptors= ArrayList<E>  (id=1384)
      lazyInitHandlers= false
      logger= Jdk14Logger  (id=1386)
/hello=com.atguigu.controller.HelloController@2c484979, /hello.*=com.
```

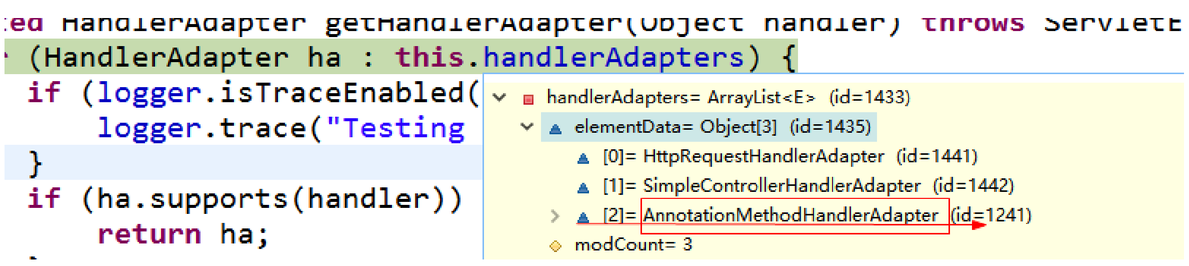
循环遍历拿到能处理url的类

```
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws
Exception {
        for (HandlerMapping hm : this.handlerMappings) {
            if (logger.isTraceEnabled()) {
                logger.trace(
                        "Testing handler map [" + hm + "] in DispatcherServlet
with name '" + getServletName() + "'");
            }
            HandlerExecutionChain handler = hm.getHandler(request);
            if (handler != null) {
                return handler;
            }
        }
        return null;
    }
```

步骤二：

getHandlerAdapter():

> **如何找到目标处理器类的适配器。要拿适配器才去执行目标方法**



```
ed HandlerAdapter getHandlerAdapter(Object handler) throws ServletE
 (HandlerAdapter ha : this.handlerAdapters) {
  if (logger.isTraceEnabled(
      logger.trace("Testing
  }
  if (ha.supports(handler))
      return ha;
```

```
handlerAdapters= ArrayList<E>  (id=1433)
  elementData= Object[3]  (id=1435)
    [0]= HttpRequestHandlerAdapter  (id=1441)
    [1]= SimpleControllerHandlerAdapter  (id=1442)
    [2]= AnnotationMethodHandlerAdapter  (id=1241)
  modCount= 3
```

AnnotationMethodHandlerAdapter：

- 能解析注解方法的适配器；
- 处理器类中只要有标了注解的这些方法就能用；

```java
protected HandlerAdapter getHandlerAdapter(Object handler) throws
ServletException {
        for (HandlerAdapter ha : this.handlerAdapters) {
            if (logger.isTraceEnabled()) {
                logger.trace("Testing handler adapter [" + ha + "]");
            }
            if (ha.supports(handler)) {
                return ha;
            }
        }
        throw new ServletException("No adapter for handler [" + handler +
                "]: The DispatcherServlet configuration needs to include a
HandlerAdapter that supports this handler");
    }
```

步骤三:

执行目标方法的细节;

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

↓

return invokeHandlerMethod(request, response, handler);

```java
protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
HttpServletResponse response, Object handler)
            throws Exception {
          //拿到方法的解析器
        ServletHandlerMethodResolver methodResolver =
getMethodResolver(handler);
          //方法解析器根据当前请求地址找到真正的目标方法
        Method handlerMethod = methodResolver.resolveHandlerMethod(request);
          //创建一个方法执行器;
        ServletHandlerMethodInvoker methodInvoker = new
ServletHandlerMethodInvoker(methodResolver);
          //包装原生的request, response,
        ServletWebRequest webRequest = new ServletWebRequest(request, response);
          //创建了一个,隐含模型

        ExtendedModelMap implicitModel = new BindingAwareModelMap();//**重点

         //真正执行目标方法;目标方法利用反射执行期间确定参数值,提前执行modelattribute等所
有的操作都在这个方法中;
        Object result = methodInvoker.invokeHandlerMethod(handlerMethod,
handler, webRequest, implicitModel);
        //=======================看后边补充的代码块===========================
        ModelAndView mav =
                methodInvoker.getModelAndView(handlerMethod, handler.getClass(),
result, implicitModel, webRequest);

        methodInvoker.updateModelAttributes(handler, (mav != null ?
mav.getModel() : null), implicitModel, webRequest);

        return mav;
    }
```

↓

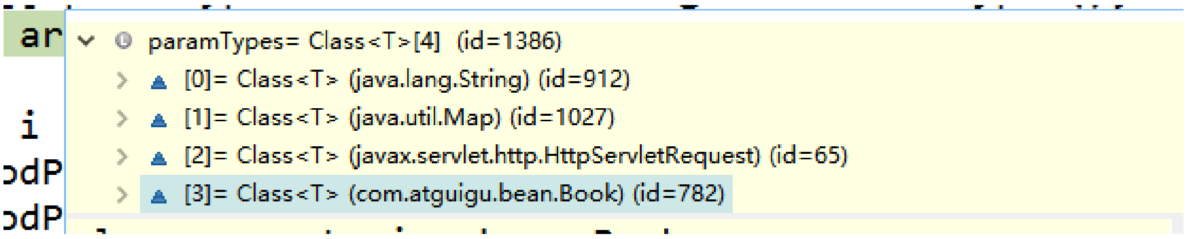
```
Object result = methodInvoker.invokeHandlerMethod(handlerMethod, handler,
webRequest, implicitModel);
```

```
publicfinal Object invokeHandlerMethod(Method handlerMethod, Object handler,
            NativeWebRequest webRequest, ExtendedModelMap implicitModel) throws
Exception {
        Method handlerMethodToInvoke =
BridgeMethodResolver.findBridgedMethod(handlerMethod);
        try {
            boolean debug = logger.isDebugEnabled();
            for (String attrName :
this.methodResolver.getActualSessionAttributeNames()) {
                Object attrValue =
this.sessionAttributeStore.retrieveAttribute(webRequest, attrName);
                if (attrValue != null) {
                    implicitModel.addAttribute(attrName, attrValue);
                }
            }

          //找到所有@ModelAttribute注解标注的方法;
            for (Method attributeMethod :
this.methodResolver.getModelAttributeMethods()) {
                Method attributeMethodToInvoke =
BridgeMethodResolver.findBridgedMethod(attributeMethod);
                //先确定modelattribute方法执行时要使用的每一个参数的值;
               Object[] args = resolveHandlerArguments(attributeMethodToInvoke,
handler, webRequest, implicitModel);
       //==========================看后边补充的代码块
=====================================
                if (debug) {
                    logger.debug("Invoking model attribute method: " +
attributeMethodToInvoke);
                }
                String attrName =
AnnotationUtils.findAnnotation(attributeMethod, ModelAttribute.class).value();

                if (!"".equals(attrName) &&
implicitModel.containsAttribute(attrName)) {
                    continue;
                }

                ReflectionUtils.makeAccessible(attributeMethodToInvoke);

               //提前运行ModelAttribute,
                Object attrValue = attributeMethodToInvoke.invoke(handler,
args);
                if ("".equals(attrName)) {
                    Class<?> resolvedType =
GenericTypeResolver.resolveReturnType(attributeMethodToInvoke,
handler.getClass());
                    attrName =
Conventions.getVariableNameForReturnType(attributeMethodToInvoke, resolvedType,
attrValue);
                }
                /*

                方法上标注的ModelAttribute注解如果有value值
                @ModelAttribute("abc")
                hahaMyModelAttribute()

                标了: attrName="abc"
                没标: attrName="";attrName就会变为返回值类型首字母小写,
                     比如void ,或者book;

                     【
                        @ModelAttribute标在方法上的另外一个作用;
                        可以把方法运行后的返回值按照方法上@ModelAttribute("abc")
                        指定的key放到隐含模型中;
                        如果没有指定这个key; 就用返回值类型的首字母小写
                     】

                        {
                            haha=Book [id=100, bookName=西游记, author=吴承恩,
stock=98,                                      sales=10, price=98.98],
                            void=null
                        }
                */
                //把提前运行的ModelAttribute方法的返回值也放在隐含模型中
                if (!implicitModel.containsAttribute(attrName)) {
                    implicitModel.addAttribute(attrName, attrValue);
                }
            }

               //再次解析目标方法参数是哪些值
            Object[] args = resolveHandlerArguments(handlerMethodToInvoke,
handler, webRequest, implicitModel);
            if (debug) {
                logger.debug("Invoking request handler method: " +
handlerMethodToInvoke);
            }
            ReflectionUtils.makeAccessible(handlerMethodToInvoke);


            //执行目标方法
            return handlerMethodToInvoke.invoke(handler, args);
        }
        catch (IllegalStateException ex) {
            // Internal assertion failed (e.g. invalid signature):
            // throw exception with full handler method context...
            throw new HandlerMethodInvocationException(handlerMethodToInvoke,
ex);
        }
        catch (InvocationTargetException ex) {
            // User-defined @ModelAttribute/@InitBinder/@RequestMapping method
threw an exception...
            ReflectionUtils.rethrowException(ex.getTargetException());
            return null;
        }
    }
```

确定方法运行时使用的每一个参数的值

Object[] args = resolveHandlerArguments(attributeMethodToInvoke, handler, webRequest, implicitModel);

```
@RequestMapping("/updateBook")
    public String updateBook
           (
                @RequestParam(value="author")String author,
                Map<String, Object> model,
                HttpServletRequest request,
                @ModelAttribute("haha")Book book
            )
```



```
paramTypes= Class<T>[4]  (id=1386)
  [0]= Class<T> (java.lang.String) (id=912)
  [1]= Class<T> (java.util.Map) (id=1027)
  [2]= Class<T> (javax.servlet.http.HttpServletRequest) (id=65)
  [3]= Class<T> (com.atguigu.bean.Book) (id=782)
```

```
标了注解：
          保存时哪个注解的详细信息；
          如果参数有ModelAttribute注解；
               拿到ModelAttribute注解的值让attrName保存
                    attrName="haha"

没标注解：
          1）、先看是否普通参数（是否原生API）
               再看是否Model或者Map，如果是就传入隐含模型；
          2）、自定义类型的参数没有ModelAttribute 注解
                    1）、先看是否原生API
                    2）、再看是否Model或者Map
                    3）、再看是否是其他类型的比如SessionStatus、HttpEntity、Errors
                    4）、再看是否简单类型的属性；比如是否Integer，String，基本类型
                            如果是paramName=“”
                    5)、attrName="";

如果是自定义类型对象，最终会产生两个效果；
     1）、如果这个参数标注了ModelAttribute注解就给attrName赋值为这个注解的value值
     2）、如果这个参数没有标注ModelAttribute注解就给attrName赋值"";
```

```
private Object[] resolveHandlerArguments(Method handlerMethod, Object handler,
            NativeWebRequest webRequest, ExtendedModelMap implicitModel) throws
Exception {
        Class<?>[] paramTypes = handlerMethod.getParameterTypes();
          //创建了一个和参数个数一样多的数组，会用来保存每一个参数的值
        Object[] args = new Object[paramTypes.length];


        for (int i = 0; i < args.length; i++) {
            MethodParameter methodParam = new MethodParameter(handlerMethod, i);

 methodParam.initParameterNameDiscovery(this.parameterNameDiscoverer);
            GenericTypeResolver.resolveParameterType(methodParam,
handler.getClass());
            String paramName = null;
            String headerName = null;
            boolean requestBodyFound = false;
            String cookieName = null;
            String pathVarName = null;
            String attrName = null;
            boolean required = false;
            String defaultValue = null;
            boolean validate = false;
            Object[] validationHints = null;
            int annotationsFound = 0;
            Annotation[] paramAnns = methodParam.getParameterAnnotations();

            //找到目标方法这个参数的所有注解，如果有注解就解析并保存注解的信息；
            for (Annotation paramAnn : paramAnns) {
                if (RequestParam.class.isInstance(paramAnn)) {
                    RequestParam requestParam = (RequestParam) paramAnn;
                    paramName = requestParam.value();
                    required = requestParam.required();
                    defaultValue =
parseDefaultValueAttribute(requestParam.defaultValue());
                    annotationsFound++;
                }
                else if (RequestHeader.class.isInstance(paramAnn)) {
                    RequestHeader requestHeader = (RequestHeader) paramAnn;
                    headerName = requestHeader.value();
                    required = requestHeader.required();
                    defaultValue =
parseDefaultValueAttribute(requestHeader.defaultValue());
                    annotationsFound++;
                }
                else if (RequestBody.class.isInstance(paramAnn)) {
                    requestBodyFound = true;
                    annotationsFound++;
                }
                else if (CookieValue.class.isInstance(paramAnn)) {
                    CookieValue cookieValue = (CookieValue) paramAnn;
                    cookieName = cookieValue.value();
                    required = cookieValue.required();
                    defaultValue =
parseDefaultValueAttribute(cookieValue.defaultValue());
                    annotationsFound++;
                }
                else if (PathVariable.class.isInstance(paramAnn)) {
                    PathVariable pathVar = (PathVariable) paramAnn;
                    pathVarName = pathVar.value();
                    annotationsFound++;
                }
                else if (ModelAttribute.class.isInstance(paramAnn)) {
                    ModelAttribute attr = (ModelAttribute) paramAnn;
                    attrName = attr.value();
                    annotationsFound++;
                }
                else if (Value.class.isInstance(paramAnn)) {
                    defaultValue = ((Value) paramAnn).value();
                }
                else if
(paramAnn.annotationType().getSimpleName().startsWith("Valid")) {
                    validate = true;
                    Object value = AnnotationUtils.getValue(paramAnn);
                    validationHints = (value instanceof Object[] ? (Object[])
value : new Object[] {value});
                }
            }
            if (annotationsFound > 1) {
                throw new IllegalStateException("Handler parameter annotations
are exclusive choices - " +
                        "do not specify more than one such annotation on the
same parameter: " + handlerMethod);
            }

             //没有找到注解的情况；
            if (annotationsFound == 0) {

                //解析普通参数
                Object argValue = resolveCommonArgument(methodParam,
webRequest);
                //====================看后边补充的代码块========================
                 //会进入resolveStandardArgument（解析标准参数）


                if (argValue != WebArgumentResolver.UNRESOLVED) {
                    args[i] = argValue;
                }
                else if (defaultValue != null) {
                    args[i] = resolveDefaultValue(defaultValue);
                }
                else {

               //判断是否是Model或者是Map旗下的，如果是将之前创建的隐含模型直接赋值给这个参
数
                    Class<?> paramType = methodParam.getParameterType();
                    if (Model.class.isAssignableFrom(paramType) ||
Map.class.isAssignableFrom(paramType)) {
                        if
(!paramType.isAssignableFrom(implicitModel.getClass())) {
                            throw new IllegalStateException("Argument [" +
paramType.getSimpleName() + "] is of type " +
                                    "Model or Map but is not assignable from the
actual model. You may need to switch " +
                                    "newer MVC infrastructure classes to use
this argument.");
                        }
                        args[i] = implicitModel;
                    }
                    else if (SessionStatus.class.isAssignableFrom(paramType)) {
                        args[i] = this.sessionStatus;
                    }
                    else if (HttpEntity.class.isAssignableFrom(paramType)) {
                        args[i] = resolveHttpEntityRequest(methodParam,
webRequest);
```

```
                    }
                    else if (Errors.class.isAssignableFrom(paramType)) {
                        throw new IllegalStateException("Errors/BindingResult
argument declared " +
                                "without preceding model attribute. Check your
handler method signature!");
                    }
                    else if (BeanUtils.isSimpleProperty(paramType)) {
                        paramName = "";
                    }
                    else {
                        attrName = "";
                    }
                }
            }


               //确定值的环节
            if (paramName != null) {
                args[i] = resolveRequestParam(paramName, required, defaultValue,
methodParam, webRequest, handler);
            }
            else if (headerName != null) {
                args[i] = resolveRequestHeader(headerName, required,
defaultValue, methodParam, webRequest, handler);
            }
            else if (requestBodyFound) {
                args[i] = resolveRequestBody(methodParam, webRequest, handler);
            }
            else if (cookieName != null) {
                args[i] = resolveCookieValue(cookieName, required, defaultValue,
methodParam, webRequest, handler);
            }
            else if (pathVarName != null) {
                args[i] = resolvePathVariable(pathVarName, methodParam,
webRequest, handler);
            }


            //确定自定义类型参数的值；还要将请求中的每一个参数赋值给这个对象
            else if (attrName != null) {
                WebDataBinder binder = resolveModelAttribute(attrName,
methodParam, implicitModel, webRequest, handler);
                //====================看后边代码补充============================
                boolean assignBindingResult = (args.length > i + 1 &&
Errors.class.isAssignableFrom(paramTypes[i + 1]));
                if (binder.getTarget() != null) {
                    doBind(binder, webRequest, validate, validationHints,
!assignBindingResult);
                }
                args[i] = binder.getTarget();
                if (assignBindingResult) {
                    args[i + 1] = binder.getBindingResult();
                    i++;
                }
                implicitModel.putAll(binder.getBindingResult().getModel());
            }
        }
        return args;
    }
```

如果没有注解：

resolveCommonArgument）**就是确定当前的参数是否是原生API**；

```
        @Override
        protected Object resolveStandardArgument(Class<?> parameterType,
NativeWebRequest webRequest) throws Exception {
            HttpServletRequest request =
webRequest.getNativeRequest(HttpServletRequest.class);
            HttpServletResponse response =
webRequest.getNativeResponse(HttpServletResponse.class);

            if (ServletRequest.class.isAssignableFrom(parameterType) ||
                    MultipartRequest.class.isAssignableFrom(parameterType)) {
                Object nativeRequest =
webRequest.getNativeRequest(parameterType);
                if (nativeRequest == null) {
                    throw new IllegalStateException(
                            "Current request is not of type [" +
parameterType.getName() + "]: " + request);
                }
                return nativeRequest;
            }
            else if (ServletResponse.class.isAssignableFrom(parameterType)) {
                this.responseArgumentUsed = true;
                Object nativeResponse =
webRequest.getNativeResponse(parameterType);
                if (nativeResponse == null) {
                    throw new IllegalStateException(
                            "Current response is not of type [" +
parameterType.getName() + "]: " + response);
                }
                return nativeResponse;
            }
            else if (HttpSession.class.isAssignableFrom(parameterType)) {
                return request.getSession();
            }
            else if (Principal.class.isAssignableFrom(parameterType)) {
                return request.getUserPrincipal();
            }
            else if (Locale.class.equals(parameterType)) {
                return RequestContextUtils.getLocale(request);
            }
            else if (InputStream.class.isAssignableFrom(parameterType)) {
                return request.getInputStream();
            }
            else if (Reader.class.isAssignableFrom(parameterType)) {
                return request.getReader();
            }
            else if (OutputStream.class.isAssignableFrom(parameterType)) {
                this.responseArgumentUsed = true;
                return response.getOutputStream();
            }
            else if (Writer.class.isAssignableFrom(parameterType)) {
                this.responseArgumentUsed = true;
                return response.getWriter();
            }
            return super.resolveStandardArgument(parameterType, webRequest);
        }
```

resolveModelAttribute

```
SpringMVC确定POJO值的三步；
1、如果隐含模型中有这个key（标了ModelAttribute注解就是注解指定的value，没标就是参数类型的首字母小写）指定的值；
        如果有将这个值赋值给bindObject；
2、如果是SessionAttributes标注的属性，就从session中拿；
3、如果都不是就利用反射创建对象；
```

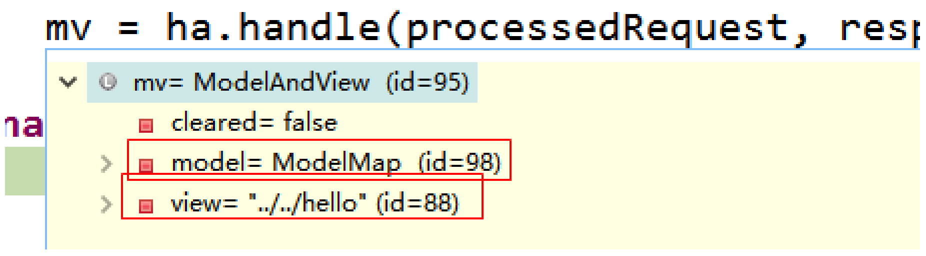
```
private WebDataBinder resolveModelAttribute(String attrName, MethodParameter methodParam,
            ExtendedModelMap implicitModel, NativeWebRequest webRequest, Object handler) throws Exception {

        // Bind request parameter onto object...
        String name = attrName;

        if ("".equals(name)) {
               //如果attrName是空串；就将参数类型的首字母小写作为值
                //Book book2121 -> name=book
            name = Conventions.getVariableNameForParameter(methodParam);
        }
        Class<?> paramType = methodParam.getParameterType();
        Object bindObject;

         //确定目标对象的值
        if (implicitModel.containsKey(name)) {
            bindObject = implicitModel.get(name);
        }
        else if (this.methodResolver.isSessionAttribute(name, paramType)) {
            bindObject =
this.sessionAttributeStore.retrieveAttribute(webRequest, name);
            if (bindObject == null) {
                raiseSessionRequiredException("Session attribute '" + name + "' required - not found in session");
            }
        }
        else {
            bindObject = BeanUtils.instantiateClass(paramType);
        }


        WebDataBinder binder = createBinder(webRequest, bindObject, name);
        initBinder(handler, name, binder, webRequest);
        return binder;
    }
```

总结：

1. 运行流程简单版；
2. 确定方法每个参数的值；
    1. 标注解：保存注解的信息；最终得到这个注解应该对应解析的值；
    2. 没标注解：
        1. 看是否是原生API；
        2. 看是否是Model或者是Map，SessionStatus、HttpEntity、Errors...
        3. 看是否是简单类型；paramName=""
        4. 给attrName赋值；attrName（参数标了@ModelAttribute("")就是指定的，没标就是""）
            1. attrName使用**参数的类型**首字母小写；或者使用之前@ModelAttribute("")的值
            2. 先看隐含模型中有每个这个attrName作为key对应的值；如果有就从隐含模型中获取并赋值
            3. 看是否是@SessionAttributes(value="haha")；标注的属性，如果是从session中拿；
            4. 不是@SessionAttributes标注的，利用反射创建一个对象；
        5. 不是@SessionAttributes标注的，利用反射创建一个对象；

步骤四：

1. 任何方法的返回值，最终都会被包装成ModelAndView对象


```
mv = ha.handle(processedRequest, resp
```
mv= ModelAndView (id=95)
cleared= false
model= ModelMap (id=98)
view= "../../hello" (id=88)

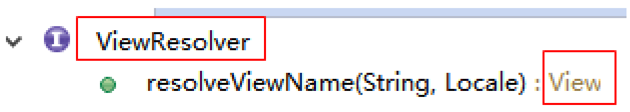
步骤五：

SpringMVC视图解析：

> 1、方法执行后的返回值会作为页面地址参考，转发或者重定向到页面
>
> 2、视图解析器可能会进行页面地址的拼串

```
processDispatchResult(processedRequest, response, mappedHandler,
    mv, dispatchException);
```

1. 调用processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException)
    - 来到页面的方法视图渲染流程
    - 将域中的数据在页面展示
    - 页面就是用来渲染模型数据的
2. 调用render(mv, request, response)
    - 渲染页面
3. View与ViewResolver
    - ViewResolver的作用是根据视图名（方法的返回值）得到View对象

- 

  ViewResolver

  resolveViewName(String, Locale) : View

4. 怎么能根据方法的返回值（视图名）得到View对象?

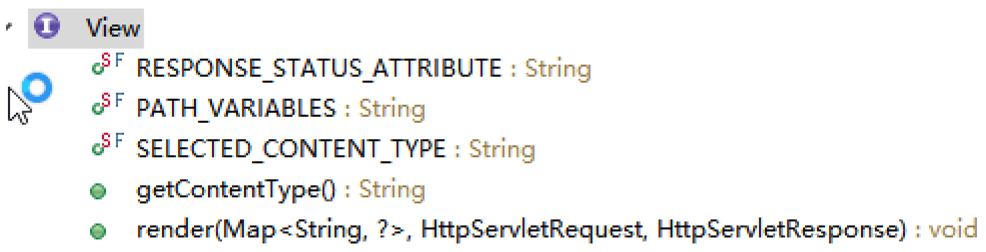
```
protected View resolveViewName(String viewName, Map<String, Object> model,
Locale locale,
            HttpServletRequest request) throws Exception {

          //遍历所有的ViewResolver；
        for (ViewResolver viewResolver : this.viewResolvers) {


          //viewResolver视图解析器根据方法的返回值，得到一个View对象；
            View view = viewResolver.resolveViewName(viewName, locale);


            if (view != null) {
                return view;
            }
        }
        return null;
    }
```

- resolveViewName实现

```
@Override
    public View resolveViewName(String viewName, Locale locale) throws
Exception {
        if (!isCache()) {
            return createView(viewName, locale);
        }
        else {
            Object cacheKey = getCacheKey(viewName, locale);
            View view = this.viewAccessCache.get(cacheKey);
            if (view == null) {
                synchronized (this.viewCreationCache) {
                    view = this.viewCreationCache.get(cacheKey);
                    if (view == null) {


                        // Ask the subclass to create the View object.
                         //根据方法的返回值创建出视图View对象；
                        view = createView(viewName, locale);


                        if (view == null && this.cacheUnresolved) {
                            view = UNRESOLVED_VIEW;
                        }
                        if (view != null) {
                            this.viewAccessCache.put(cacheKey, view);
                            this.viewCreationCache.put(cacheKey, view);
                            if (logger.isTraceEnabled()) {
                                logger.trace("Cached view [" + cacheKey
+ "]");
```

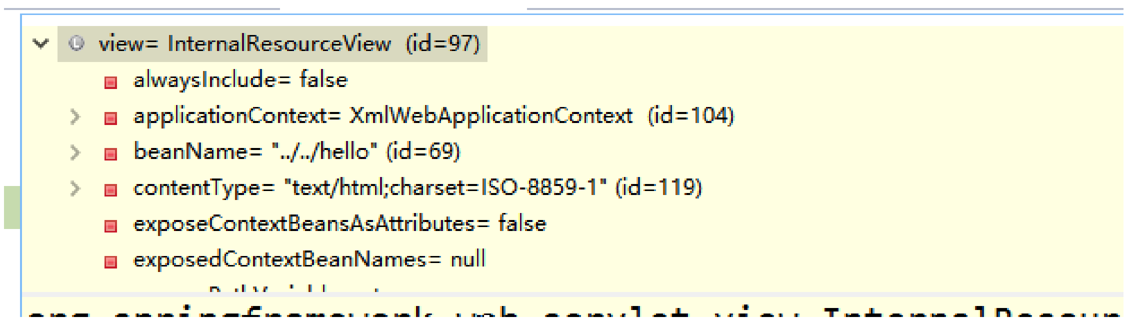
```
                        }
                    }
                }
            }
        }
        return (view != UNRESOLVED_VIEW ? view : null);
    }
}
```

- 创建View对象

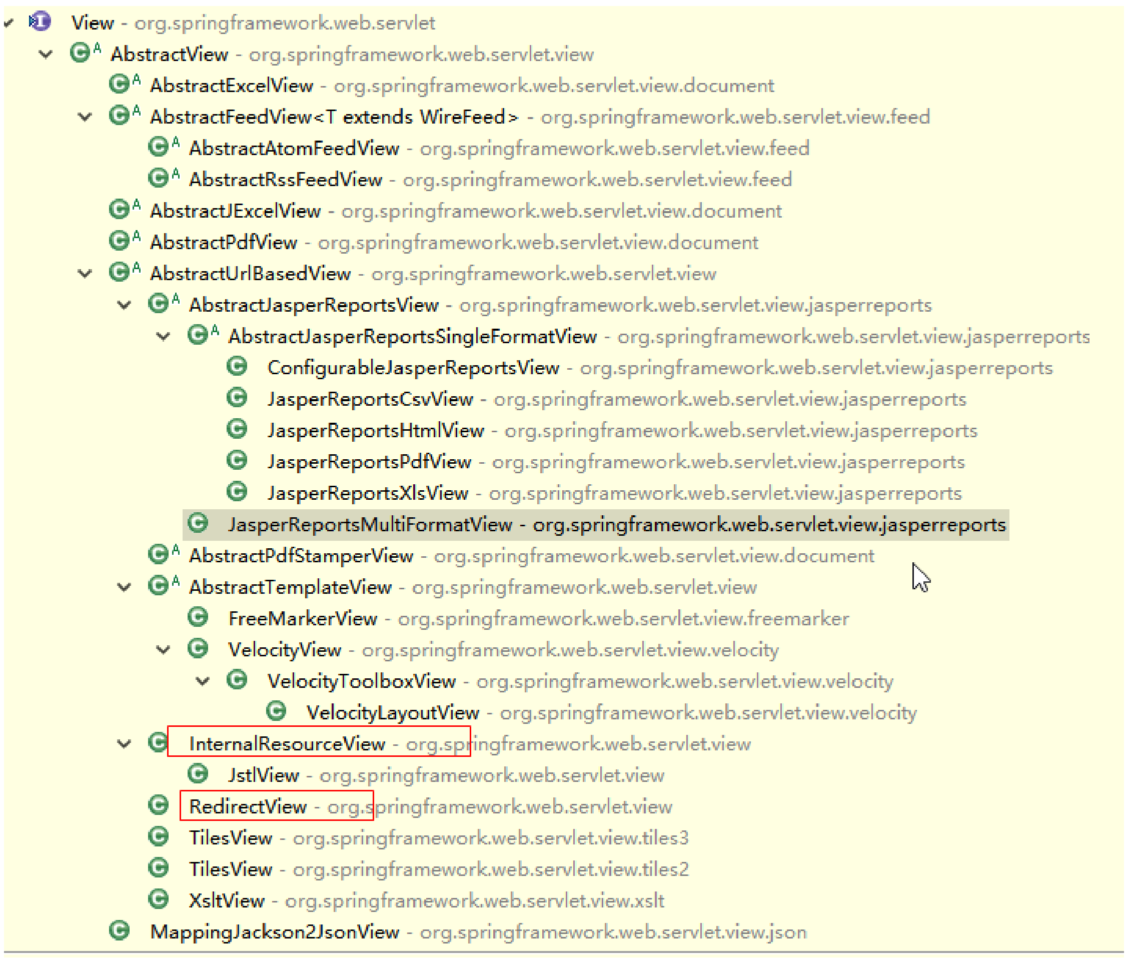


```
View
    RESPONSE_STATUS_ATTRIBUTE : String
    PATH_VARIABLES : String
    SELECTED_CONTENT_TYPE : String
    getContentType() : String
    render(Map<String, ?>, HttpServletRequest, HttpServletResponse) : void
```

```
  @Override
      protected View createView(String viewName, Locale locale) throws
Exception {
          // If this resolver is not supposed to handle the given view,
          // return null to pass on to the next resolver in the chain.
          if (!canHandle(viewName, locale)) {
              return null;
          }
          // Check for special "redirect:" prefix.
          if (viewName.startsWith(REDIRECT_URL_PREFIX)) {
              String redirectUrl =
viewName.substring(REDIRECT_URL_PREFIX.length());
              RedirectView view = new RedirectView(redirectUrl,
isRedirectContextRelative(), isRedirectHttp10Compatible());
              return applyLifecycleMethods(viewName, view);
          }
          // Check for special "forward:" prefix.
          if (viewName.startsWith(FORWARD_URL_PREFIX)) {
              String forwardUrl =
viewName.substring(FORWARD_URL_PREFIX.length());
              return new InternalResourceView(forwardUrl);
          }
          // Else fall back to superclass implementation: calling loadView.
          //如果没有前缀就使用父类默认创建一个View;
          return super.createView(viewName, locale);
      }
```



```
view= InternalResourceView  (id=97)
    alwaysInclude= false
    applicationContext= XmlWebApplicationContext  (id=104)
    beanName= "../../hello" (id=69)
    contentType= "text/html;charset=ISO-8859-1" (id=119)
    exposeContextBeansAsAttributes= false
    exposedContextBeanNames= null
```



```
View - org.springframework.web.servlet
  AbstractView - org.springframework.web.servlet.view
    AbstractExcelView - org.springframework.web.servlet.view.document
    AbstractFeedView<T extends WireFeed> - org.springframework.web.servlet.view.feed
      AbstractAtomFeedView - org.springframework.web.servlet.view.feed
      AbstractRssFeedView - org.springframework.web.servlet.view.feed
    AbstractJExcelView - org.springframework.web.servlet.view.document
    AbstractPdfView - org.springframework.web.servlet.view.document
    AbstractUrlBasedView - org.springframework.web.servlet.view
      AbstractJasperReportsView - org.springframework.web.servlet.view.jasperreports
        AbstractJasperReportsSingleFormatView - org.springframework.web.servlet.view.jasperreports
          ConfigurableJasperReportsView - org.springframework.web.servlet.view.jasperreports
          JasperReportsCsvView - org.springframework.web.servlet.view.jasperreports
          JasperReportsHtmlView - org.springframework.web.servlet.view.jasperreports
          JasperReportsPdfView - org.springframework.web.servlet.view.jasperreports
          JasperReportsXlsView - org.springframework.web.servlet.view.jasperreports
        JasperReportsMultiFormatView - org.springframework.web.servlet.view.jasperreports
      AbstractPdfStamperView - org.springframework.web.servlet.view.document
      AbstractTemplateView - org.springframework.web.servlet.view
        FreeMarkerView - org.springframework.web.servlet.view.freemarker
        VelocityView - org.springframework.web.servlet.view.velocity
          VelocityToolboxView - org.springframework.web.servlet.view.velocity
            VelocityLayoutView - org.springframework.web.servlet.view.velocity
      InternalResourceView - org.springframework.web.servlet.view
        JstlView - org.springframework.web.servlet.view
      RedirectView - org.springframework.web.servlet.view
      TilesView - org.springframework.web.servlet.view.tiles3
      TilesView - org.springframework.web.servlet.view.tiles2
      XsltView - org.springframework.web.servlet.view.xslt
    MappingJackson2JsonView - org.springframework.web.servlet.view.json
```

- 返回View对象
  - 视图解析器得到View对象的流程就是，所有配置的视图解析器都来尝试根据视图名（返回值）得到View（视图）对象；如果能得到就返回，得不到就换下一个视图解析器；
  - 调用View对象的render方法

```
@Override
    public void render(Map<String, ?> model, HttpServletRequest
request, HttpServletResponse response) throws Exception {
        if (logger.isTraceEnabled()) {
            logger.trace("Rendering view with name '" + this.beanName +
"' with model " + model +
                " and static attributes " + this.staticAttributes);
        }

        Map<String, Object> mergedModel =
createMergedOutputModel(model, request, response);

        prepareResponse(request, response);


        //渲染要给页面输出的所有数据
        renderMergedOutputModel(mergedModel, request, response);
    }
```

- InternalResourceView有这个方法renderMergedOutputModel；

```
@Override
```

```
    protected void renderMergedOutputModel(
            Map<String, Object> model, HttpServletRequest request,
HttpServletResponse response) throws Exception {

        // Determine which request handle to expose to the
RequestDispatcher.
        HttpServletRequest requestToExpose =
getRequestToExpose(request);

        // Expose the model object as request attributes.


        //将模型中的数据放在请求域中
        exposeModelAsRequestAttributes(model, requestToExpose);



        // Expose helpers as request attributes, if any.
        exposeHelpers(requestToExpose);

        // Determine the path for the request dispatcher.
        String dispatcherPath = prepareForRendering(requestToExpose,
response);

        // Obtain a RequestDispatcher for the target resource
(typically a JSP).
        RequestDispatcher rd = getRequestDispatcher(requestToExpose,
dispatcherPath);
        if (rd == null) {
            throw new ServletException("Could not get RequestDispatcher
for [" + getUrl() +
                    "]: Check that the corresponding file exists within
your web application archive!");
        }

        // If already included or response already committed, perform
include, else forward.
        if (useInclude(requestToExpose, response)) {
            response.setContentType(getContentType());
            if (logger.isDebugEnabled()) {
                logger.debug("Including resource [" + getUrl() + "] in
InternalResourceView '" + getBeanName() + "'");
            }
            rd.include(requestToExpose, response);
        }

        else {
            // Note: The forwarded resource is supposed to determine
the content type itself.
            if (logger.isDebugEnabled()) {
                logger.debug("Forwarding to resource [" + getUrl() + "]
in InternalResourceView '" + getBeanName() + "'");
            }

            //转发页面
            rd.forward(requestToExpose, response);
        }
    }
```

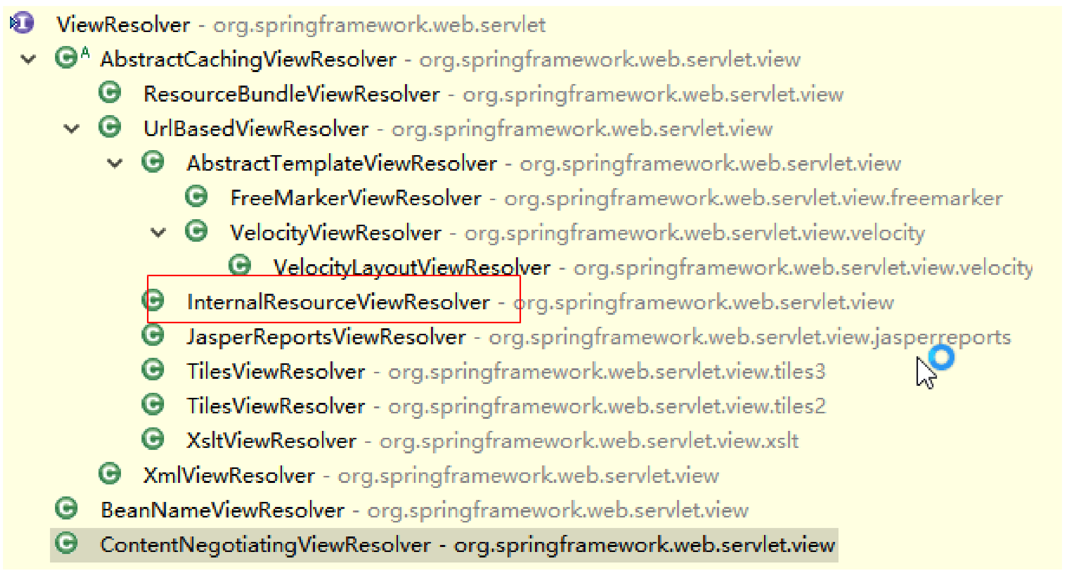
- 将模型中的所有数据取出来全放在request域中

```
protected void exposeModelAsRequestAttributes(Map<String, Object>
model, HttpServletRequest request) throws Exception {
        for (Map.Entry<String, Object> entry : model.entrySet()) {
            String modelName = entry.getKey();
            Object modelValue = entry.getValue();
            if (modelValue != null) {

                //将ModelMap中的数据放到请求域中
                request.setAttribute(modelName, modelValue);


                if (logger.isDebugEnabled()) {
                    logger.debug("Added model object '" + modelName +
"' of type [" + modelValue.getClass().getName() +
                            "] to request in view with name '" +
getBeanName() + "'");
                }
            }
            else {
                request.removeAttribute(modelName);
                if (logger.isDebugEnabled()) {
                    logger.debug("Removed model object '" + modelName +
                            "' from request in view with name '" +
getBeanName() + "'");
                }
            }
        }
    }
```

总结：

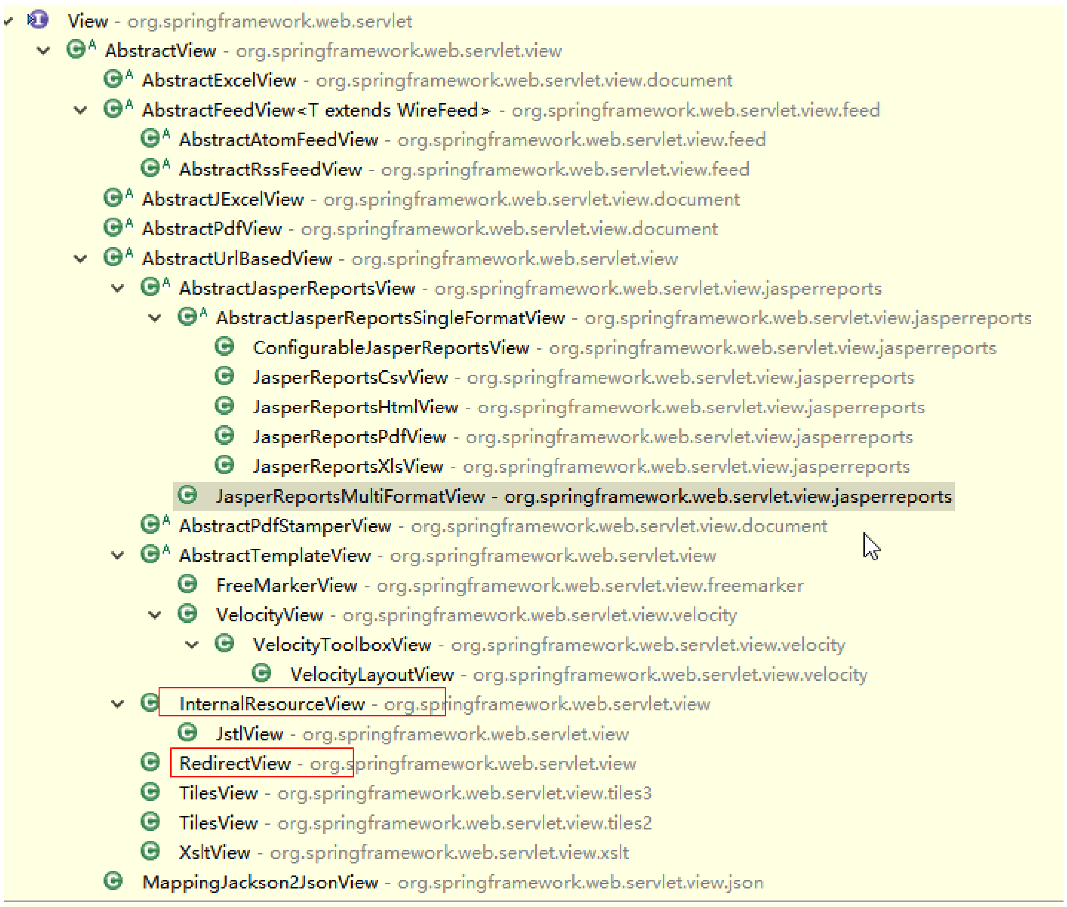
  - 视图解析器只是为了得到视图对象
  - 视图对象才能真正的转发（将模型数据全部放在请求域中）或者重定向到页面视图对象才能真正的渲染视图

- ViewResolver



```
ViewResolver - org.springframework.web.servlet
  AbstractCachingViewResolver - org.springframework.web.servlet.view
    ResourceBundleViewResolver - org.springframework.web.servlet.view
    UrlBasedViewResolver - org.springframework.web.servlet.view
      AbstractTemplateViewResolver - org.springframework.web.servlet.view
        FreeMarkerViewResolver - org.springframework.web.servlet.view.freemarker
        VelocityViewResolver - org.springframework.web.servlet.view.velocity
          VelocityLayoutViewResolver - org.springframework.web.servlet.view.velocity
      InternalResourceViewResolver - org.springframework.web.servlet.view
      JasperReportsViewResolver - org.springframework.web.servlet.view.jasperreports
      TilesViewResolver - org.springframework.web.servlet.view.tiles3
      TilesViewResolver - org.springframework.web.servlet.view.tiles2
      XsltViewResolver - org.springframework.web.servlet.view.xslt
    XmlViewResolver - org.springframework.web.servlet.view
  BeanNameViewResolver - org.springframework.web.servlet.view
  ContentNegotiatingViewResolver - org.springframework.web.servlet.view
```

- View:



```
View - org.springframework.web.servlet
  AbstractView - org.springframework.web.servlet.view
    AbstractExcelView - org.springframework.web.servlet.view.document
    AbstractFeedView<T extends WireFeed> - org.springframework.web.servlet.view.feed
      AbstractAtomFeedView - org.springframework.web.servlet.view.feed
      AbstractRssFeedView - org.springframework.web.servlet.view.feed
    AbstractJExcelView - org.springframework.web.servlet.view.document
    AbstractPdfView - org.springframework.web.servlet.view.document
    AbstractUrlBasedView - org.springframework.web.servlet.view
      AbstractJasperReportsView - org.springframework.web.servlet.view.jasperreports
        AbstractJasperReportsSingleFormatView - org.springframework.web.servlet.view.jasperreports
          ConfigurableJasperReportsView - org.springframework.web.servlet.view.jasperreports
          JasperReportsCsvView - org.springframework.web.servlet.view.jasperreports
          JasperReportsHtmlView - org.springframework.web.servlet.view.jasperreports
          JasperReportsPdfView - org.springframework.web.servlet.view.jasperreports
          JasperReportsXlsView - org.springframework.web.servlet.view.jasperreports
        JasperReportsMultiFormatView - org.springframework.web.servlet.view.jasperreports
      AbstractPdfStamperView - org.springframework.web.servlet.view.document
      AbstractTemplateView - org.springframework.web.servlet.view
        FreeMarkerView - org.springframework.web.servlet.view.freemarker
        VelocityView - org.springframework.web.servlet.view.velocity
          VelocityToolboxView - org.springframework.web.servlet.view.velocity
            VelocityLayoutView - org.springframework.web.servlet.view.velocity
      InternalResourceView - org.springframework.web.servlet.view
        JstlView - org.springframework.web.servlet.view
      RedirectView - org.springframework.web.servlet.view
      TilesView - org.springframework.web.servlet.view.tiles3
      TilesView - org.springframework.web.servlet.view.tiles2
      XsltView - org.springframework.web.servlet.view.xslt
    MappingJackson2JsonView - org.springframework.web.servlet.view.json
```

8. 视图解析

8.1 forward和redirect前缀

通过SpringMVC来实现转发和重定向。

- 直接 return “success”，会走视图解析器进行拼串
- 转发：return “forward:/succes.jsp”；直接写绝对路径，/表示当前项目下，不走视图解析器
- 重定向：return “redirect:/success.jsp”；不走视图解析器

```java
@Controller
public class ResultSpringMVC {
    @RequestMapping("/hello01")
    public String test1(){
        //转发
        //会走视图解析器
        return "success";
    }

    @RequestMapping("/hello02")
    public String test2(){
        //转发二
        //不走视图解析器
        return "forward:/success.jsp";
    }

    @RequestMapping("/hello03")
    public String test3(){
        //重定向
        //不走视图解析器
        return "redirect:/success.jsp";
    }
  }
```

使用原生的ServletAPI时要注意，**/路径需要加上项目名才能成功**

```
    @RequestMapping("/result/t2")
    public void test2(HttpServletRequest req, HttpServletResponse resp)
  throwsIOException {
        //重定向
        resp.sendRedirect("/index.jsp");
    }

    @RequestMapping("/result/t3")
    public void test3(HttpServletRequest req, HttpServletResponse resp)
  throwsException {
        //转发
        req.setAttribute("msg","/result/t3");
        req.getRequestDispatcher("/WEB-INF/jsp/test.jsp").forward(req,resp);
    }
```

8.2 jstlView

导包导入了jstl的时候会自动创建为一个jstlView；可以快速方便的支持国际化功能；

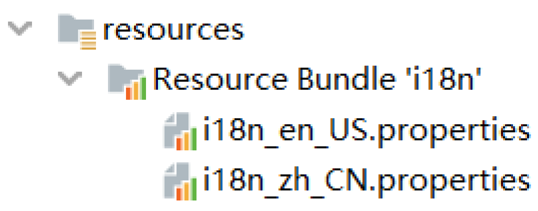
可以支持快速国际化；

javaWeb国际化步骤；

1. 得得到一个Locale对象；
2. 使用ResourceBundle绑定国际化资源文件
3. 使用ResourceBundle.getString("key")；获取到国际化配置文件中的值
4. web页面的国际化，fmt标签库来做
    - `<fmt:setLocale>`
    - `<fmt:setBundle>`
    - `<fmt:message>`

有了JstlView以后

1. 让Spring管理国际化资源就行



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

<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="i18n"></property>
</bean>
```

2. 直接在页面使用`<fmt:message>`

```jsp
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>@%>
    ...
<h1>
    <fmt:message key="welcomeinfo"/>
</h1>
<form action="">
    <fmt:message key="username"/>:<input /><br/>
    <fmt:message key="password"/>:<input /><br/>
    <input type="submit" value='<fmt:message key="loginBtn"/>'/>
</form>
    ...
```

注意:

一定要过SpringMVC的视图解析流程，人家会创建一个jstlView帮你快速国际化;

- 不能写redirect:
- 不能写forward:

```java
if (viewName.startsWith(FORWARD_URL_PREFIX)) {
            String forwardUrl = viewName.substring(FORWARD_URL_PREFIX.length());
            return new InternalResourceView(forwardUrl);
        }
```

8.3 mvc:view-controller

`mvc:view-controller`:

直接将请求映射到某个页面，不需要写方法了:

走了视图解析的功能

```xml
<mvc:view-controller path="/toLogin" view-name="login"/>
<!--开启MVC注解驱动模式-->
<mvc:annotation-driven/>
```

8.4 自定义视图解析器

扩展：加深视图解析器和视图对象；

- 视图解析器根据方法的返回值得到视图对象
- 多个视图解析器都会尝试能否得到视图对象；
- 视图对象不同就可以具有不同功能

```
for (ViewResolver viewResolver : this.viewResolvers) {
         //viewResolver视图解析器根据方法的返回值，得到一个View对象；
           View view = viewResolver.resolveViewName(viewName, locale);
           if (view != null) {
               return view;
           }
       }
```

- 让我们的视图解析器工作
- 得到我们的视图对象
- 我们的视图对象自定义渲染逻辑

自定义视图和视图解析器的步骤

1. 编写自定义的视图解析器，和视图实现类

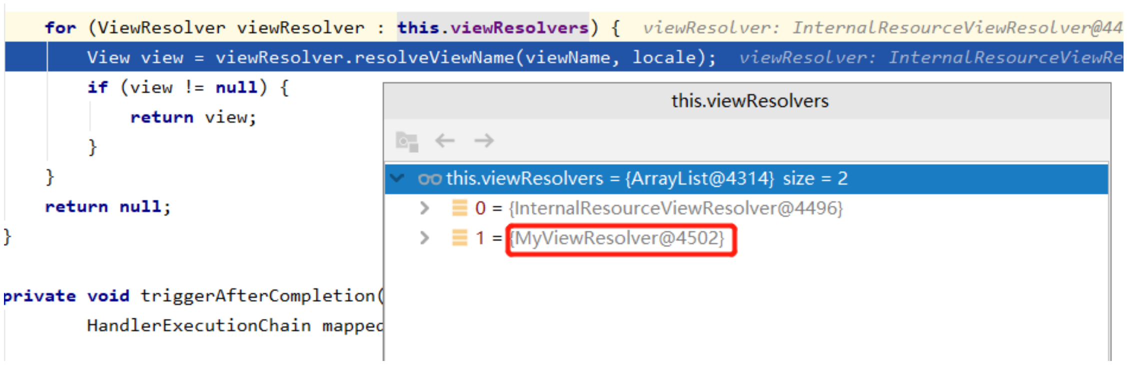
```
public class MyViewResolver implements ViewResolver {
    public View resolveViewName(String viewName, Locale locale) throws
Exception {
        if (viewName.startsWith("myView:")){
           return new MyView();
        }else{
            return null;
        }
    }
}
```

```
public class MyView implements View {
    public String getContentType() {
        return "text/html";
    }

    public void render(Map<String, ?> model, HttpServletRequest request,
HttpServletResponse response) throws Exception {
        System.out.println("保存的数据："+model);
        response.getWriter().write("即将展现内容:");
    }
}
```

2. 视图解析器必须放在ioc容器中，让其工作，能创建出我们的自定义视图对象

```
<bean class="com.chenhui.view.MyViewResolver"></bean>
```

在源码中看到我们的编写的解析器

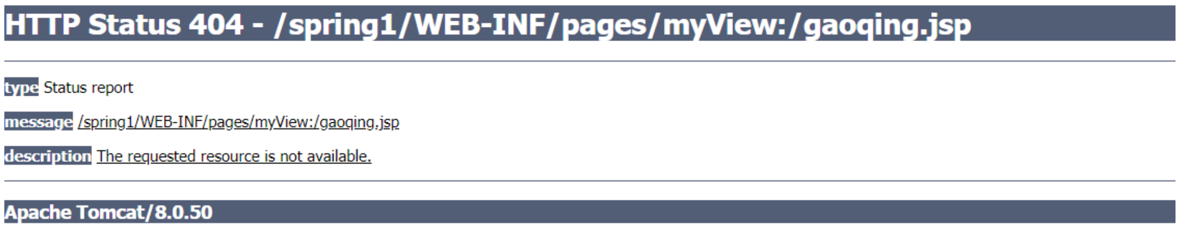


```
    for (ViewResolver viewResolver : this.viewResolvers) {  viewResolver: InternalResourceViewResolver@44
        View view = viewResolver.resolveViewName(viewName, locale);  viewResolver: InternalResourceViewRe
        if (view != null) {
            return view;
        }
    }
    return null;
}

private void triggerAfterCompletion(
        HandlerExecutionChain mappe
```

this.viewResolvers

```
this.viewResolvers = {ArrayList@4314}  size = 2
  0 = {InternalResourceViewResolver@4496}
  1 = {MyViewResolver@4502}
```

但是被InternalResourceViewResolver先拦截了执行了render



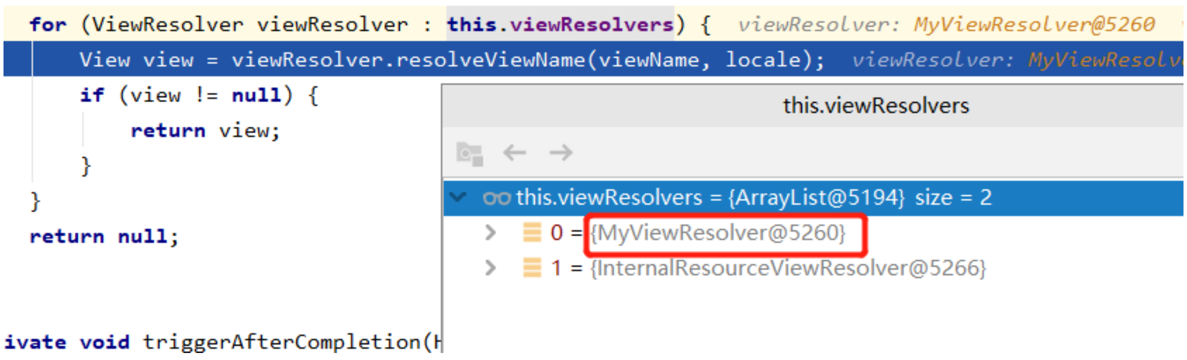
```
HTTP Status 404 - /spring1/WEB-INF/pages/myView:/gaoqing.jsp

type Status report

message /spring1/WEB-INF/pages/myView:/gaoqing.jsp

description The requested resource is not available.

Apache Tomcat/8.0.50
```

MyViewResolver要实现Ordered接口

```
public class MyViewResolver implements ViewResolver, Ordered {

    private Integer order = 0;

    public View resolveViewName(String viewName, Locale locale) throws Exception
{
        if (viewName.startsWith("myView:")) {
            return new MyView();
        } else {
            return null;
        }
    }

    public int getOrder() {
        return this.order;
    }

    public void setOrder(Integer order) {
        this.order = order;
    }
}
```

```
<bean class="com.chenhui.view.MyViewResolver">
        <property name="order" value="1"></property>
    </bean>
```

发现顺序已经改变

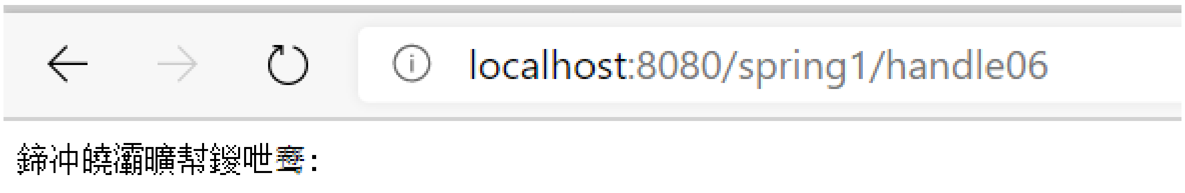


```
for (ViewResolver viewResolver : this.viewResolvers) {   viewResolver: MyViewResolver@5260
    View view = viewResolver.resolveViewName(viewName, locale);   viewResolver: MyViewResolv
    if (view != null) {
        return view;
    }
}
return null;

ivate void triggerAfterCompletion(H
```

```
this.viewResolvers
this.viewResolvers = {ArrayList@5194} size = 2
  0 = {MyViewResolver@5260}
  1 = {InternalResourceViewResolver@5266}
```

到了我们的页面（虽然乱码），需要设置ContentType

response.setContentType("text/html ");

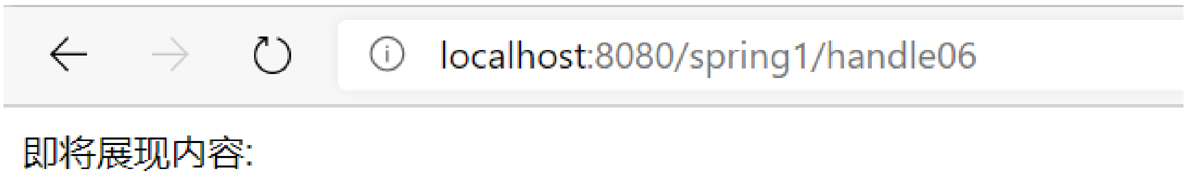


localhost:8080/spring1/handle06

鍗冲皢灞曠幇鍐呭:

```
public void render(Map<String, ?> model, HttpServletRequest request,
HttpServletResponse response) throws Exception {
        System.out.println("保存的数据: "+model);
        response.setContentType("text/html ");
        response.getWriter().write("即将展现内容:");
    }
```

成功!



localhost:8080/spring1/handle06

即将展现内容:

9. ResetCRUD