



Spring MVC

基于 Spring4.x

讲师：佟刚
新浪微博：@尚硅谷-佟刚



内容概要

- 1.SpringMVC 概述
- 2.SpringMVC 的 HelloWorld
- 3.使用 @RequestMapping 映射请求
- 4.映射请求参数 & 请求头
- 5.处理模型数据
- 6.视图和视图解析器
- 7.RESTful CRUD
- 8.SpringMVC 表单标签 & 处理静态资源
- 9.数据转换 & 数据格式化 & 数据校验
- 10.处理 JSON : 使用 HttpMessageConverter
- 11.国际化
- 12.文件的上传
- 13.使用拦截器
- 14.异常处理
- 15.SpringMVC 运行流程
- 16.在 Spring 的环境下使用 SpringMVC
- 17.SpringMVC 对比 Struts2

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

SpringMVC 概述

- **Spring** 为展现层提供的基于 **MVC** 设计理念的优秀的 Web 框架，是目前最主流的 **MVC** 框架之一
- Spring3.0 后全面超越 Struts2，成为最优秀的 **MVC** 框架
- **Spring MVC** 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
- 采用了松散耦合可插拔组件结构，比其他 **MVC** 框架更具扩展性和灵活性

HelloWorld

- 步骤：
 - 加入 jar 包
 - 在 web.xml 中配置 **DispatcherServlet**
 - 加入 Spring MVC 的配置文件
 - 编写处理请求的处理器，并标识为处理器
 - 编写视图

HelloWorld : 加入 jar 包

- jar 包：
 - **commons-logging-1.1.3.jar**
 - spring-aop-4.0.0.RELEASE.jar
 - spring-beans-4.0.0.RELEASE.jar
 - spring-context-4.0.0.RELEASE.jar
 - spring-core-4.0.0.RELEASE.jar
 - spring-expression-4.0.0.RELEASE.jar
 - **spring-web-4.0.0.RELEASE.jar**
 - **spring-webmvc-4.0.0.RELEASE.jar**

HelloWorld：配置 web.xml

- 配置 **DispatcherServlet**：DispatcherServlet 默认加载 /WEB-INF/<servletName-servlet>.xml 的 Spring 配置文件，启动 WEB 层的 Spring 容器。可以通过 **contextConfigLocation** 初始化参数自定义配置文件的位置和名称

```
<servlet>
    <servlet-name>helloworld</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext-mvc.xml</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>helloworld</servlet-name>
    <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

HelloWorld：创建 Spring MVC 配置文件

- 配置自动扫描的包

```
<context:component-scan base-package= "com.atguigu.springmvc">
</context:component-scan>
```

- 配置视图解析器：视图名称解析器：将视图逻辑名解析为：/WEB-INF/pages/<viewName>.jsp

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

HelloWorld : 创建请求处理器类

```
@Controller
public class HelloWorld {

    @RequestMapping("/helloSpringMVC")
    public String helloworld(){
        System.out.println("helloworld...");
        return "success";
    }

}
```



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

web.xml

springmvc-1/helloWorld.action

url

```
@Controller
public class HelloWorldController {
    @RequestMapping("/helloWorld")
    public String helloWorld(){
        System.out.println("HelloWorld SpringMVC");
        return "success";
    }
}
```

Handler

/WEB-INF/view/success.jsp

实际的物理视图

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

SpringMVC 配置文件

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 **@RequestMapping** 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

使用 @RequestMapping 映射请求

- Spring MVC 使用 **@RequestMapping** 注解为控制器指定可以处理哪些 URL 请求
- 在控制器的类定义及方法定义处都可标注 **@RequestMapping**
 - **类定义处**：提供初步的请求映射信息。相对于 WEB 应用的根目录
 - **方法处**：提供进一步的细分映射信息。相对于类定义处的 URL。若类定义处未标注 @RequestMapping，则方法处标记的 URL 相对于 WEB 应用的根目录
- DispatcherServlet 截获请求后，就通过控制器上 @RequestMapping 提供的映射信息确定请求所对应的处理方法。

使用 @RequestMapping 映射请求示例

```
@Controller  
@RequestMapping("/hello")  
public class HelloWorld {
```

类定义处标记的 @RequestMapping 限定了处理器类可以处理所有 URI 为 /hello 的请求，它相对于 WEB 容器部署的根路径

```
    @RequestMapping("/SpringMVC")  
    public String helloworld(){  
        System.out.println("helloworld...");  
        return "success";  
    }
```

处理器类可以定义多个处理方法，处理来自/hello 下的请求

```
}
```

映射请求参数、请求方法或请求头

- 标准的 HTTP 请求报头

① 请求方法 ② 请求URL ③ HTTP协议及版本

④ 报文头

```
POST /chapter17/user.html HTTP/1.1
Accept: image/jpeg, application/x-ms-application, ..., */
Referer: http://localhost:8088/chapter17/user/register.html?
code=100&time=123123
Accept-Language: zh-CN
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1;
Content-Type: application/x-www-form-urlencoded
Host: localhost:8088
Content-Length: 112
Connection: Keep-Alive
Cache-Control: no-cache
Cookie: JSESSIONID=24DF2688E37EE4F66D9669D2542AC17B
name=tom&password=1234&realName=tomson
```

⑤ 报文体

映射请求参数、请求方法或请求头

- `@RequestMapping` 除了可以使用 **请求 URL** 映射请求外，还可以使用 **请求方法**、**请求参数**及**请求头**映射请求
- `@RequestMapping` 的 `value`、`method`、`params` 及 `heads` 分别表示**请求 URL**、**请求方法**、**请求参数**及**请求头**的映射条件，他们之间是**与**的关系，联合使用多个条件可让请求映射**更加精确化**。
- `params` 和 `headers`**支持简单的表达式**：
 - `param1`: 表示请求必须包含名为 `param1` 的请求参数
 - `!param1`: 表示请求不能包含名为 `param1` 的请求参数
 - `param1 != value1`: 表示请求包含名为 `param1` 的请求参数，但其值不能为 `value1`
 - `{"param1=value1", "param2"}`: 请求必须包含名为 `param1` 和 `param2` 的两个请求参数，且 `param1` 参数的值必须为 `value1`

映射请求参数、请求方法或请求头

```
@RequestMapping(value="/delete", method=RequestMethod.POST, params="userId")
public String test1(){
    //...
    return "user/test1";
}

@RequestMapping(value="/show", headers="contentType=text/*")
public String test2(){
    //...
    return "user/test2";
}
```

使用 @RequestMapping 映射请求

- **Ant 风格资源地址支持 3 种匹配符：**
 - ? : 匹配文件名中的一个字符
 - * : 匹配文件名中的任意字符
 - ** : ** 匹配多层路径
- **@RequestMapping 还支持 Ant 风格的 URL :**
 - /user/*/createUser: 匹配
/user/**aaa**/createUser、/user/**bbb**/createUser 等 URL
 - /user/**/createUser: 匹配
/user/createUser、/user/**aaa/bbb**/createUser 等 URL
 - /user/createUser???: 匹配
/user/createUser**aa**、/user/createUser**bb** 等 URL

@PathVariable 映射 URL 绑定的占位符

- 带占位符的 **URL** 是 **Spring3.0** 新增的功能，该功能在 SpringMVC 向 **REST** 目标挺进发展过程中具有里程碑的意义
- 通过 **@PathVariable** 可以将 **URL** 中占位符参数绑定到控制器处理方法的入参中：URL 中的 **{xxx}** 占位符可以通过 **@PathVariable("xxx")** 绑定到操作方法的入参中。

```
@RequestMapping("/delete/{id}")
public String delete(@PathVariable("id") Integer id){
    UserDao.delete(id);
    return "redirect:/user/list.action";
}
```

REST

- REST：即 **Representational State Transfer**。（资源）表现层状态转化。是目前最流行的一种互联网软件架构。它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用
- **资源（Resources）**：网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的存在。可以用一个**URI**（统一资源定位符）指向它，**每种资源对应一个特定的 URI**。要获取这个资源，访问它的**URI**就可以，因此 **URI 即为每一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（Representation）**。比如，文本可以用 **txt** 格式表现，也可以用 **HTML** 格式、**XML** 格式、**JSON** 格式表现，甚至可以采用二进制格式。
- **状态转化（State Transfer）**：每发出一个请求，就代表了客户端和服务器的一次交互过程。**HTTP**协议，是一个无状态协议，即所有的状态都保存在服务器端。因此，如果客户端想要操作服务器，必须通过某种手段，让服务器端发生“**状态转化**”（**State Transfer**）。而这种转化是建立在表现层之上的，所以就是“**表现层状态转化**”。具体说，就是 **HTTP 协议里面，四个表示操作方式的动词**：**GET、POST、PUT、DELETE**。它们分别对应四种基本操作：**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。

REST

- 示例：
 - /order/1 HTTP **GET**：得到 id = 1 的 order
 - /order/1 HTTP **DELETE**：删除 id = 1 的 order
 - /order/1 HTTP **PUT**：更新 id = 1 的 order
 - /order HTTP **POST**：新增 order
- **HiddenHttpMethodFilter**：浏览器 form 表单只支持 GET 与 POST 请求，而 DELETE、PUT 等 method 并不支持，Spring3.0 添加了一个过滤器，可以将这些请求转换为标准的 http 方法，使得支持 GET、POST、PUT 与 DELETE 请求。

@PathVariable 绑定 URL 占位符到入参

- 带占位符的 **URL** 是 **Spring3.0** 新增的功能，该功能在 SpringMVC 向 **REST** 目标挺进发展过程中具有里程碑的意义
- 通过 **@PathVariable** 可以将 **URL** 中占位符参数绑定到控制器处理方法的入参中：URL 中的 {xxx} 占位符可以通过 **@PathVariable("xxx")** 绑定到操作方法的入参中。

```
@RequestMapping("/delete/{id}")
public String delete(@PathVariable("id") Integer id){
    UserDao.delete(id);
    return "redirect:/user/list.action";
}
```

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求参数
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

请求处理方法签名

- Spring MVC 通过分析处理方法的签名，将 HTTP 请求信息绑定到处理方法的相应入参中。
- Spring MVC 对控制器处理方法签名的限制是很宽松的，几乎可以按喜欢的任何方式对方法进行签名。
- 必要时可以对方法及方法入参标注相应的注解（**@PathVariable**、**@RequestParam**、**@RequestHeader** 等）、Spring MVC 框架会将 HTTP 请求的信息绑定到相应的方法入参中，并根据方法的返回值类型做出相应的后续处理。

使用 @RequestParam 绑定请求参数值

- 在处理方法入参处使用 **@RequestParam** 可以把请求参数传递给请求方法
 - value : 参数名
 - required : 是否必须。默认为 true, 表示请求参数中必须包含对应的参数, 若不存在, 将抛出异常

```
@RequestMapping("/handle5")
public String handle5(@RequestParam(value="userName", required=false) String userName,
                      @RequestParam("age") int age){
    return "success";
}
```

使用 @RequestHeader 绑定请求报头的属性值

- 请求头包含了若干个属性，服务器可据此获知客户端的信息，通过 **@RequestHeader** 即可将请求头中的属性值绑定到处理方法的入参中

```
@RequestMapping("/handle7")
public String handle7(@RequestHeader("Accept-Encoding") String encoding,
                      @RequestHeader("Keep-Alive") long keppAlive){
    return "success";
}
```

使用 @CookieValue 绑定请求中的 Cookie 值

- **@CookieValue** 可让处理方法入参绑定某个 Cookie 值

```
@RequestMapping("/handle6")
public String handle6(@CookieValue(value="sessionId", required=false) String sessionId,
                      @RequestParam("age") int age){
    return "success";
}
```

使用 POJO 对象绑定请求参数值

- Spring MVC 会按请求参数名和 **POJO** 属性名进行自动匹配，自动为该对象填充属性值。支持级联属性。
如：dept.deptId、dept.address.tel 等

```
@RequestMapping("/handle8")
public String handle8(User user){
    return "success";
}
```

```
/handle8.action?userName=atguigu&dept.deptId=1&dept.address.tel=52571522
```

使用 Servlet API 作为入参

```
@RequestMapping("/handle9")
public void handle9(HttpServletRequest request, HttpServletResponse response){
    ...
}

@RequestMapping("/handle10")
public ModelAndView handle10(HttpServletRequest request){
    ModelAndView mav = new ModelAndView();
    ...
    return mav;
}

@RequestMapping("/handle11")
public String handle11(HttpSession session){
    ...
    return "success";
}

@RequestMapping("/handle12")
public String handle12(HttpServletRequest request,
    @RequestParam("userName") String userName){
    ...
    return "success";
}
```

MVC 的 Handler 方法可以接受哪些 ServletAPI 类型的参数

- HttpServletRequest
- HttpServletResponse
- HttpSession
- **java.security.Principal**
- **Locale**
- **InputStream**
- **OutputStream**
- **Reader**
- **Writer**

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- RESTful CRUD
- 视图和视图解析器
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

处理模型数据

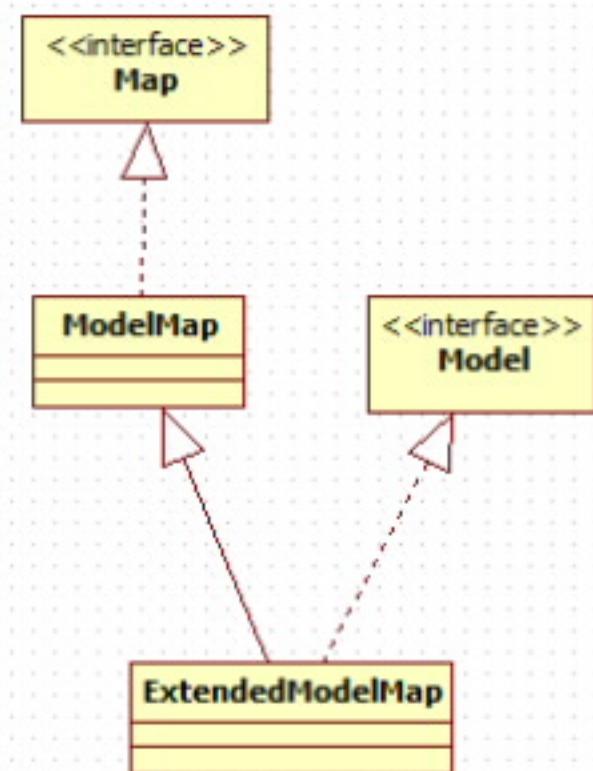
- Spring MVC 提供了以下几种途径输出模型数据：
 - **ModelAndView**: 处理方法返回值类型为 ModelAndView 时, 方法体即可通过该对象添加模型数据
 - **Map 及 Model**: 入参为 org.springframework.ui.Model、org.springframework.ui.ModelMap 或 java.util.Map 时, 处理方法返回时, Map 中的数据会自动添加到模型中。
 - **@SessionAttributes**: 将模型中的某个属性暂存到 HttpSession 中, 以便多个请求之间可以共享这个属性
 - **@ModelAttribute**: 方法入参标注该注解后, 入参的对象就会放到数据模型中

ModelAndView

- 控制器处理方法的返回值如果为 **ModelAndView**, 则其既包含视图信息，也包含模型数据信息。
- 添加模型数据：
 - ModelAndView addObject(String attributeName, Object attributeValue)
 - ModelAndView addAllObject(Map<String, ?> modelMap)
- 设置视图：
 - void setView(View view)
 - void setViewName(String viewName)

Map 及 Model

- Spring MVC 在内部使用了一个 org.springframework.ui.Model 接口存储模型数据
- 具体步骤
 - Spring MVC 在调用方法前会创建一个隐含的模型对象作为模型数据的存储容器。
 - 如果方法的入参为 Map 或 Model 类型，Spring MVC 会将隐含模型的引用传递给这些入参。在方法体内，开发者可以通过这个入参对象访问到模型中的所有数据，也可以向模型中添加新的属性数据



Map 及 Model 示例

```
@ModelAttribute("user")
public User getUser(){
    User user = new User();
    user.setAge(10);

    return user;
}
```

```
@RequestMapping("/handle20")
public String handle20(Map<String, Object> map){
    map.put("time", new Date());
    User user = (User) map.get("user");
    user.setEmail("TongGang@atguigu.com");
    return "success";
}
```

email: \${requestScope.user.email }

time: \${requestScope.time }

@SessionAttributes

- 若希望在多个请求之间共用某个模型属性数据，则可以在控制器类上标注一个 **@SessionAttributes**, Spring MVC 将在模型中对应的属性暂存到 HttpSession 中。
- @SessionAttributes 除了可以通过**属性名**指定需要放到会话中的属性外，还可以通过模型属性的**对象类型**指定哪些模型属性需要放到会话中
 - @SessionAttributes(types=User.class) 会将隐含模型中所有类型为 User.class 的属性添加到会话中。
 - @SessionAttributes(value={"user1", "user2"})
 - @SessionAttributes(types={User.class, Dept.class})
 - @SessionAttributes(value={"user1", "user2"}, types={Dept.class})

@SessionAttributes 示例

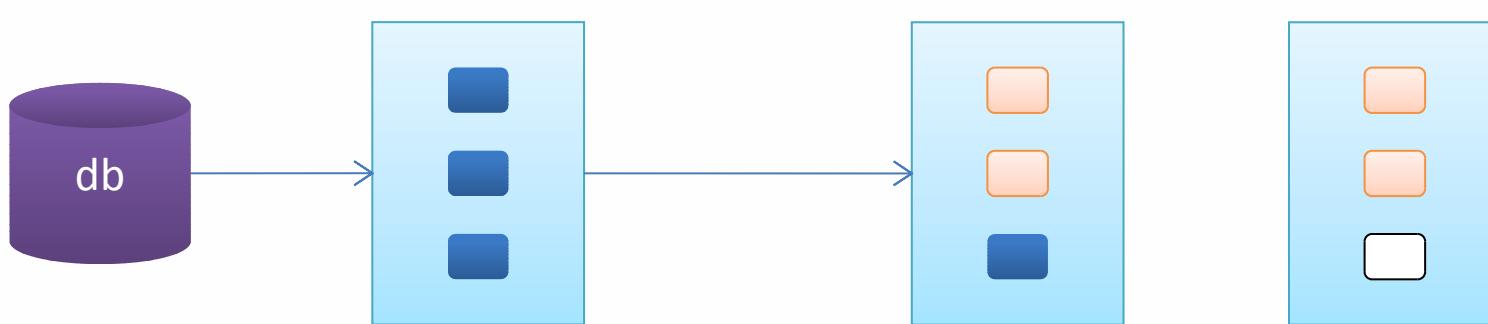
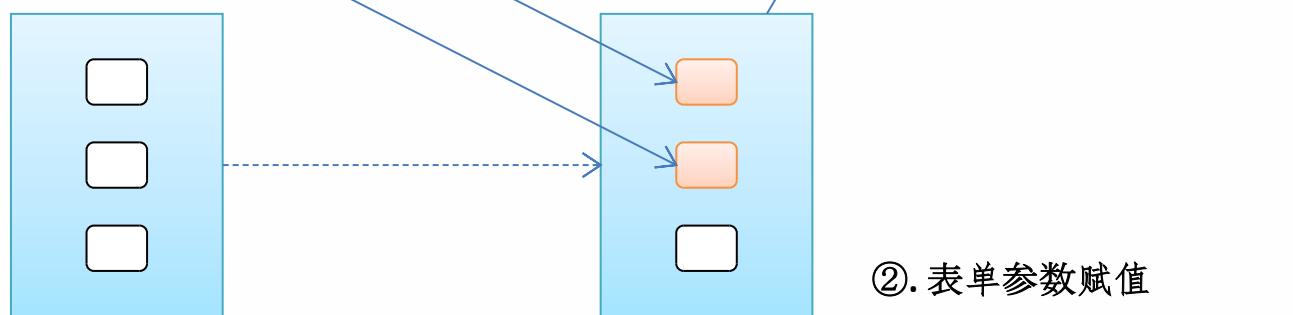
```
@SessionAttributes("user")
@Controller
@RequestMapping("/hello")
public class HelloWorld {

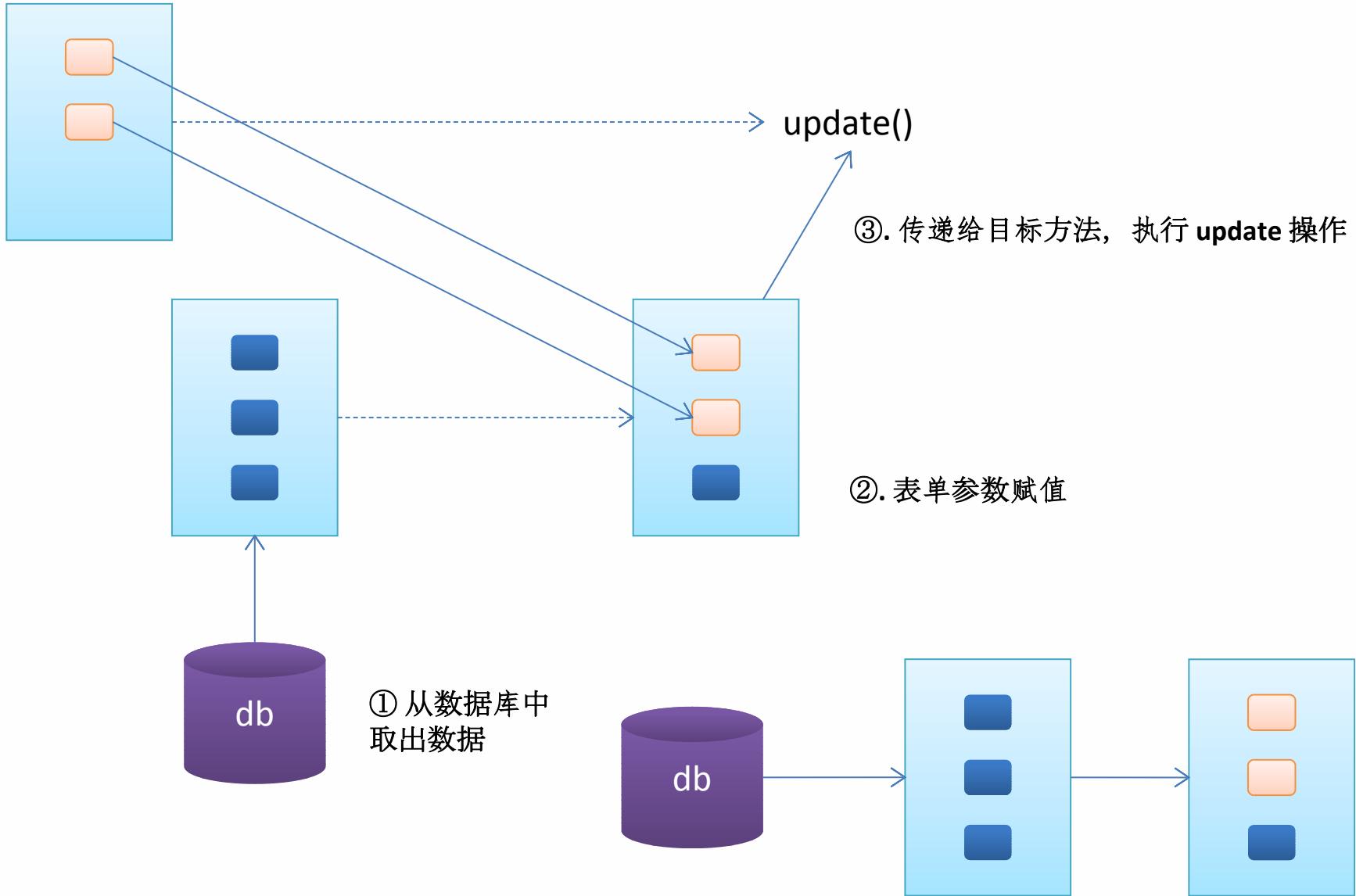
    @ModelAttribute("user")
    public User getUser(){
        User user = new User();
        user.setAge(10);

        return user;
    }
}
```

```
@RequestMapping("/handle21")
public String handle21(@ModelAttribute("user") User user){
    user.setAge(22);
    return "redirect:/hello/handle22.action";
}

@RequestMapping("/handle22")
public String handle22(Map<String, Object> map,
                      SessionStatus sessionStatus){
    User user = (User) map.get("user");
    user.setId(200);
    return "success";
}
```





@ModelAttribute

- 在方法定义上使用 **@ModelAttribute** 注解：**Spring MVC** 在调用目标处理方法前，会先逐个调用在方法级上标注了 **@ModelAttribute** 的方法。
- 在方法的入参前使用 **@ModelAttribute** 注解：
 - 可以从隐含对象中获取隐含的模型数据中获取对象，再将请求参数绑定到对象中，再传入入参
 - 将方法入参对象添加到模型中

由 @SessionAttributes 引发的异常

org.springframework.web.HttpSessionRequiredException:
Session attribute 'user' required - not found in session

- 如果在处理类定义处标注了 `@SessionAttributes("xxx")`, 则尝试从会话中获取该属性, 并将其赋给该入参, 然后再用请求消息填充该入参对象。如果在会话中找不到对应的属性, 则抛出 `HttpSessionRequiredException` 异常

```
bindObject = this.sessionAttributeStore.retrieveAttribute(webRequest, name);
if (bindObject == null) {
    raiseSessionRequiredException("Session attribute '" + name + "' required - not found")
}
```

如何避免@SessionAttributes引发的异常

```
@Controller
@RequestMapping("/user")
@SessionAttributes("user")
public class UserController {
    @ModelAttribute("user")
    public User getUser(){
        User user = new User();
        return user;
    }

    @RequestMapping(value = "/handle71")
    public String handle71(@ModelAttribute("user") User user){
        ...
    }

    @RequestMapping(value = "/handle72")
    public String handle72(ModelMap modelMap, SessionStatus sessionStatus){
        ...
    }
}
```

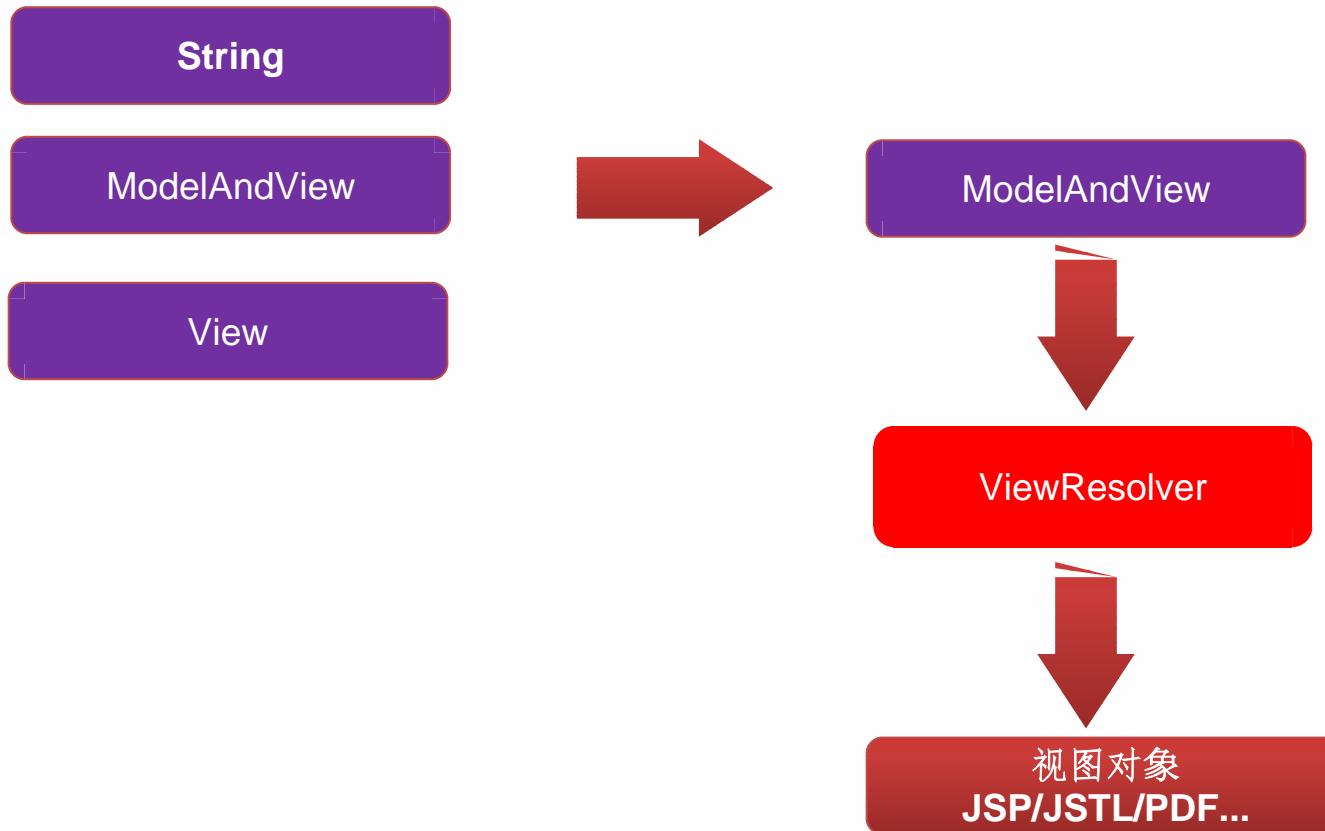
该方法会往隐含模型中添加一个名为**user**的模型属性

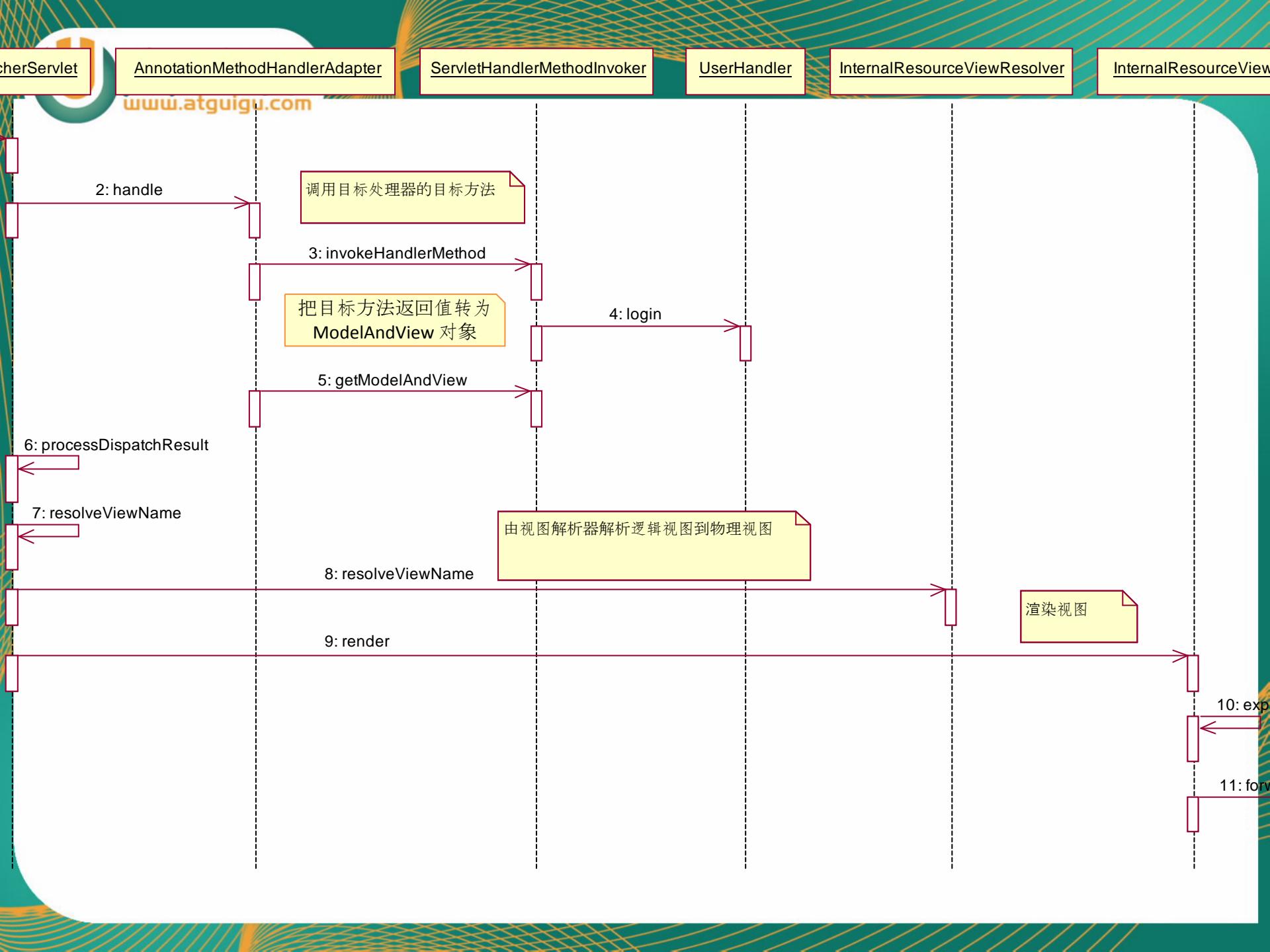
内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

Spring MVC如何解析视图

请求处理方法返回值类型





视图和视图解析器

- 请求处理方法执行完成后，最终返回一个 ModelAndView 对象。对于那些返回 String, View 或 ModeMap 等类型的处理方法，**Spring MVC 也会在内部将它们装配成一个 ModelAndView 对象**，它包含了逻辑名和模型对象的视图
- Spring MVC 借助**视图解析器（ViewResolver）**得到最终的视图对象（View），最终的视图可以是 JSP，也可能是 Excel、JFreeChart 等各种表现形式的视图
- 对于最终究竟采取何种视图对象对模型数据进行渲染，处理器并不关心，处理器工作重点聚焦在生产模型数据的工作上，从而实现 MVC 的充分解耦

视图

- 视图的作用是渲染模型数据，将模型里的数据以某种形式呈现给客户。
- 为了实现视图模型和具体实现技术的解耦，Spring 在 `org.springframework.web.servlet` 包中定义了一个高度抽象的 View 接口。

 View

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

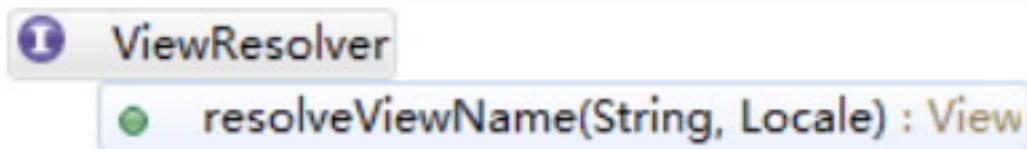
- 视图对象由视图解析器负责实例化。由于视图是无状态的，所以他们不会有线程安全的问题

常用的视图实现类

大类	视图类型	说明
URL 视资源图	InternalResourceView	将 JSP 或其它资源封装成一个视图，是 InternalResourceViewResolver 默认使用的视图实现类
	JstlView	如果 JSP 文件中使用了 JSTL 国际化标签的功能，则需要使用该视图类
文档视图	AbstractExcelView	Excel 文档视图的抽象类。该视图类基于 POI 构造 Excel 文档
	AbstractPdfView	PDF 文档视图的抽象类，该视图类基于 iText 处理 PDF 文档
报表视图	ConfigurableJasperReportsView	几个使用 JasperReports 报表技术的视图
	JasperReportsCsvView	
	JasperReportsMultiFormatView	
	JasperReportsHtmlView	
	JasperReportsPdfView	
	JasperReportsXlsView	
JSON 视图	MappingJacksonJsonView	将模型数据通过 Jackson 开源框架的 ObjectMapper 以 JSON 方式输出。

视图解析器

- SpringMVC 为逻辑视图名的解析提供了不同的策略，可以在 Spring WEB 上下文中**配置一种或多种解析策略，并指定他们之间的先后顺序**。每一种映射策略对应一个具体的视图解析器实现类。
- 视图解析器的作用比较单一：将逻辑视图解析为一个具体的视图对象。
- 所有的视图解析器都必须实现 ViewResolver 接口：



常用的视图解析器实现类

大类	视图类型	说明
解析为 Bean 的名字	BeanNameViewResolver	将逻辑视图名解析为一个 Bean，Bean 的 id 等于逻辑视图名
解析为 URL 文件	InternalResourceViewResolver	将视图名解析为一个 URL 文件，一般使用该解析器将视图名映射为一个保存在 WEB-INF 目录下的程序文件（如 JSP）
	JasperReportsViewResolver	JasperReports 是一个基于 Java 的开源报表工具，该解析器将视图名解析为报表文件对应的 URL
模版文件视图	FreeMarkerViewResoler	解析为基于 FreeMarker 模版技术的模版文件
	VelocityViewResolver	解析为基于 Velocity 模版技术的模版文件
	VelocityLayoutViewResolver	

- 程序员可以选择一种视图解析器或混用多种视图解析器
- 每个视图解析器都实现了 Ordered 接口并开放出一个 order 属性，**可以通过 order 属性指定解析器的优先顺序，order 越小优先级越高。**
- SpringMVC 会按视图解析器顺序的优先顺序对逻辑视图名进行解析，直到解析成功并返回视图对象，否则将抛出 ServletException 异常

InternalResourceViewResolver

- JSP 是最常见的视图技术，可以使用 InternalResourceViewResolver 作为视图解析器：

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

/WEB-INF/pages/ /user/createSuccess.jsp

InternalResourceViewResolver

- 若项目中使用了 JSTL，则 SpringMVC 会自动把视图由 InternalResourceView 转为 **JstlView**
- 若使用 JSTL 的 fmt 标签则需要在 SpringMVC 的配置文件中**配置国际化资源文件**

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

- 若希望直接响应通过 SpringMVC 渲染的页面，可以使用 **mvc:view-controller** 标签实现

```
<mvc:view-controller path="springmvc/testJstlView" view-name="success"/>
```

Excel 视图

- 若希望使用 Excel 展示数据列表，仅需要扩展 SpringMVC 提供的 **AbstractExcelView** 或 **AbstractJExcel View** 即可。实现 **buildExcelDocument()** 方法，在方法中使用模型数据对象构建 Excel 文档就可以了。
- **AbstractExcelView** 基于 **POI API**，而 **AbstractJExcelView** 是基于 **JExcelAPI** 的。
- 视图对象需要配置 **IOC 容器**中的一个 **Bean**，使用 **BeanNameViewResolver** 作为视图解析器即可
- 若希望直接在浏览器中直接下载 Excel 文档，则可以设置响应头 **Content-Disposition** 的值为 **attachment;filename=xxx.xls**

关于重定向

- 一般情况下，控制器方法返回字符串类型的值会被当成逻辑视图名处理
- 如果返回的字符串中带 **forward:** 或 **redirect:** 前缀时，SpringMVC 会对他们进行特殊处理：将 **forward:** 和 **redirect:** 当成指示符，其后的字符串作为 URL 来处理
 - redirect:success.jsp**：会完成一个到 **success.jsp** 的重定向的操作
 - forward:success.jsp**：会完成一个到 **success.jsp** 的转发操作

```
if (viewName.startsWith(REDIRECT_URL_PREFIX)) {  
    String redirectUrl = viewName.substring(REDI  
    RedirectView view = new RedirectView(redirec  
    return applyLifecycleMethods(viewName, view)  
}  
// Check for special "forward:" prefix.  
if (viewName.startsWith(FORWARD_URL_PREFIX)) {  
    String forwardUrl = viewName.substring(FORWA  
    return new InternalResourceView(forwardUrl);  
}
```

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- **RESTful CRUD**
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

RESTful SpringMVC CRUD

- 1. 显示所有员工信息
 - URI : **emps**
 - 请求方式 : **GET**
 - 显示效果

ID	LastName	Email	Gender	Department	Edit	Delete
1001	AA	aa@163.com	Male	AA	Edit	Delete
1003	CC	cc@163.com	Female	CC	Edit	Delete
1002	BB	bb@163.com	Male	BB	Edit	Delete
1005	EE	ee@163.com	Male	EE	Edit	Delete
1004	DD	dd@163.com	Female	DD	Edit	Delete

RESTful SpringMVC CRUD

- 2. 添加所有员工信息

- 显示添加页面：

- URI : **emp**
- 请求方式 : **GET**
- 显示效果

LastName:

Email:

Gender: Male Female

Department: BB

- 添加员工信息：

- URI : **emp**
- 请求方式 : **POST**
- 显示效果 : 完成添加, 重定向到 list 页面。

ID	LastName	Email	Gender	Department	Edit	Delete
1001	AA	aa@163.com	Male	AA	Edit	Delete
1003	CC	cc@163.com	Female	CC	Edit	Delete
1002	BB	bb@163.com	Male	BB	Edit	Delete
1005	EE	ee@163.com	Male	EE	Edit	Delete
1004	DD	dd@163.com	Female	DD	Edit	Delete
1006	FF	ff@163.com	Male	CC	Edit	Delete

RESTful SpringMVC CRUD

- 3. 删除操作
 - URL : **emp/{id}**
 - 请求方式 : **DELETE**
 - 删除后效果 : 对应记录从数据表中删除
- 4. 修改操作 : **lastName** 不可修改 !
 - 显示修改页面 :
 - URI : **emp/{id}**
 - 请求方式 : **GET**
 - 显示效果 : 回显表单。
 - 修改员工信息 :
 - URI : **emp**
 - 请求方式 : **PUT**
 - 显示效果 : 完成修改, 重定向到 **list** 页面。

UPDATE EE

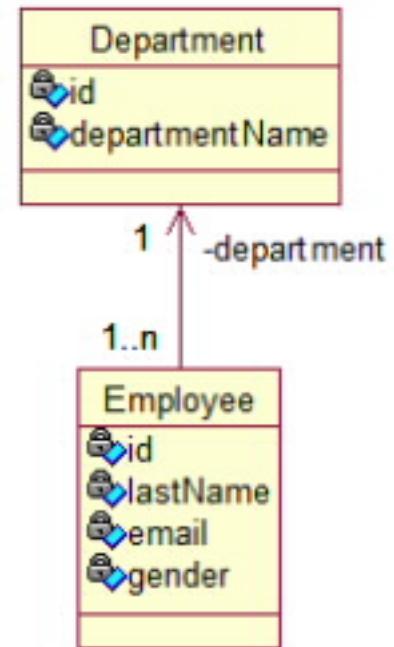
Email:

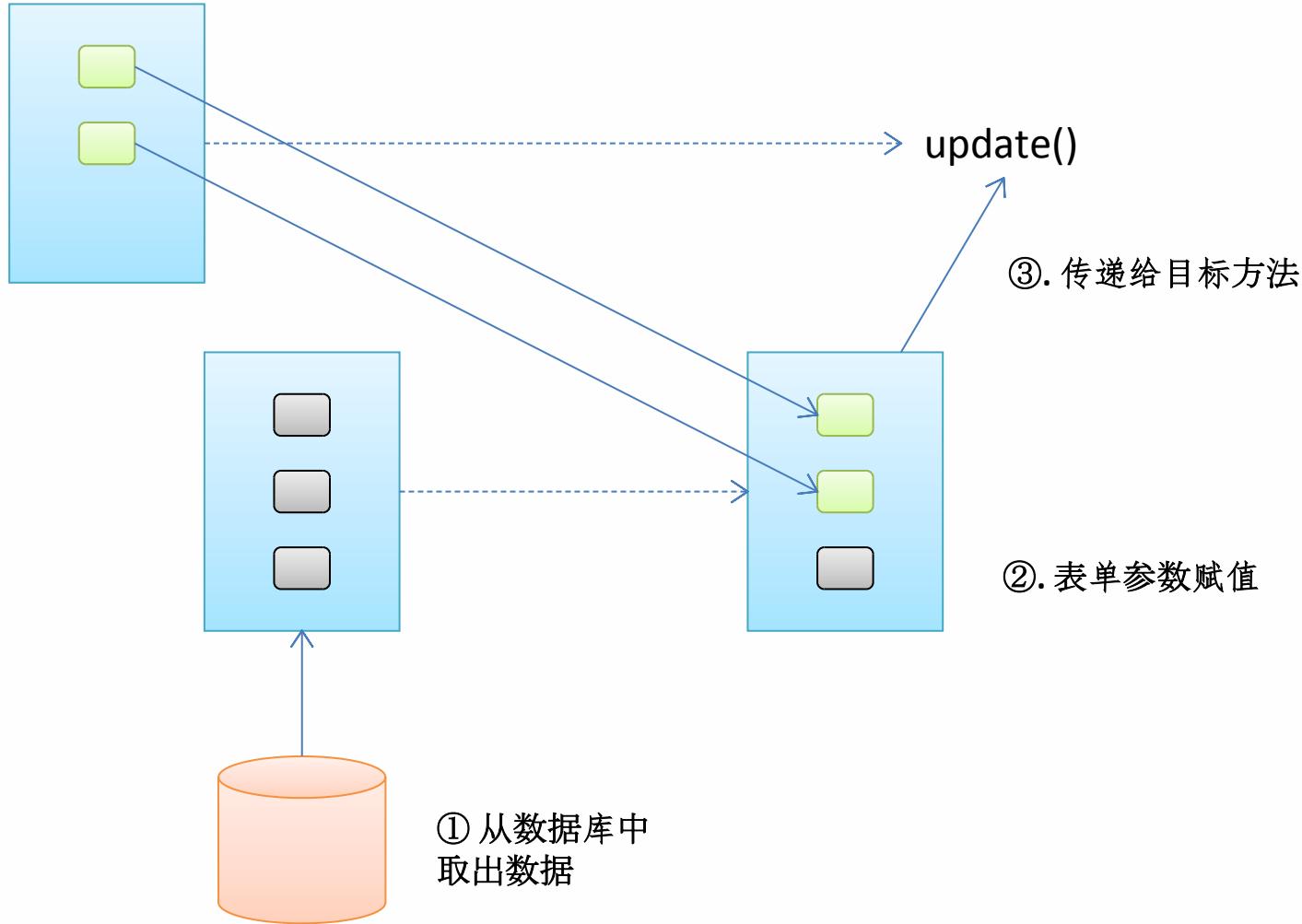
Gender: Male Female

Department:

RESTful SpringMVC CRUD

- 相关的类：
 - 实体类：Employee、Department
 - Handler : **EmployeeHandler**
 - Dao : EmployeeDao、DepartmentDao
- 相关的页面
 - **list.jsp**
 - **input.jsp**
 - **edit.jsp**





内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- **SpringMVC 表单标签 & 处理静态资源**
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

使用 Spring 的表单标签

- 通过 SpringMVC 的**表单标签**可以实现将模型数据中的属性和 HTML 表单元素相绑定，以实现表单数据**更便捷编辑和表单值的回显**

form 标签

- 一般情况下，通过 **GET** 请求获取表单页面，而通过 **POST** 请求提交表单页面，因此获取表单页面和提交表单页面的 **URL** 是相同的。只要满足该最佳条件的契约，**<form:form>** 标签就无需通过 **action** 属性指定表单提交的 **URL**
- 可以通过 **modelAttribute** 属性指定绑定的模型属性，若没有指定该属性，则默认从 **request** 域对象中读取 **command** 的表单 **bean**，如果该属性值也不存在，则会发生错误。

表单标签

- SpringMVC 提供了多个表单组件标签，如 `<form:input/>`、`<form:select/>` 等，用以绑定表单字段的属性值，它们的共有属性如下：
 - **path**：表单字段，对应 `html` 元素的 `name` 属性，支持级联属性
 - `htmlEscape`：是否对表单值的 HTML 特殊字符进行转换，默认值为 `true`
 - `cssClass`：表单组件对应的 CSS 样式类名
 - `cssErrorClass`：表单组件的数据存在错误时，采取的 CSS 样式

表单标签

- **form:input**、**form:password**、**form:hidden**、**form:textarea** : 对应 HTML 表单的 **text**、**password**、**hidden**、**textarea** 标签
- **form:radio** : 单选框组件标签，当表单 bean 对应的属性值和 value 值相等时，单选框被选中
- **form:radio** : 单选框组标签，用于构造多个单选框
 - **items** : 可以是一个 List、String[] 或 Map
 - **itemValue** : 指定 radio 的 value 值。可以是集合中 bean 的一个属性值
 - **itemLabel** : 指定 radio 的 label 值
 - **delimiter** : 多个单选框可以通过 delimiter 指定分隔符

表单标签

- **form:checkbox** : 复选框组件。用于构造单个复选框
- **form:checkboxs** : 用于构造多个复选框。使用方式同 **form:radioButtons** 标签
- **form:select** : 用于构造下拉框组件。使用方式同 **form:radioButtons** 标签
- **form:option** : 下拉框选项组件标签。使用方式同 **form:radioButtons** 标签
- **form:errors** : 显示表单组件或数据校验所对应的错误
 - <form:errors path=“ *” /> : 显示表单所有的错误
 - <form:errors path=“ user*” /> : 显示所有以 user 为前缀的属性对应的错误
 - <form:errors path=“ username” /> : 显示特定表单对象属性的错误

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

处理静态资源

- 优雅的 REST 风格的资源 URL 不希望带 .html 或 .do 等后缀
- 若将 DispatcherServlet 请求映射配置为 /，则 Spring MVC 将捕获 WEB 容器的所有请求，**包括静态资源的请求**，**SpringMVC** 会将它们当成一个普通请求处理，因找不到对应处理器将导致错误。
- 可以在 SpringMVC 的配置文件中配置 **<mvc:default-servlet-handler/>** 的方式解决静态资源的问题：
 - `<mvc:default-servlet-handler/>` 将在 SpringMVC 上下文中定义一个 **DefaultServletHttpRequestHandler**，它会对进入 DispatcherServlet 的请求进行筛查，如果发现是没有经过映射的请求，就将该请求交由 WEB 应用服务器默认的 **Servlet** 处理，如果不是静态资源的请求，才由 DispatcherServlet 继续处理
 - 一般 WEB 应用服务器默认的 **Servlet** 的名称都是 **default**。若所使用的 WEB 服务器的默认 **Servlet** 名称不是 **default**，则需要通过 **default-servlet-name** 属性显式指定

内容概要

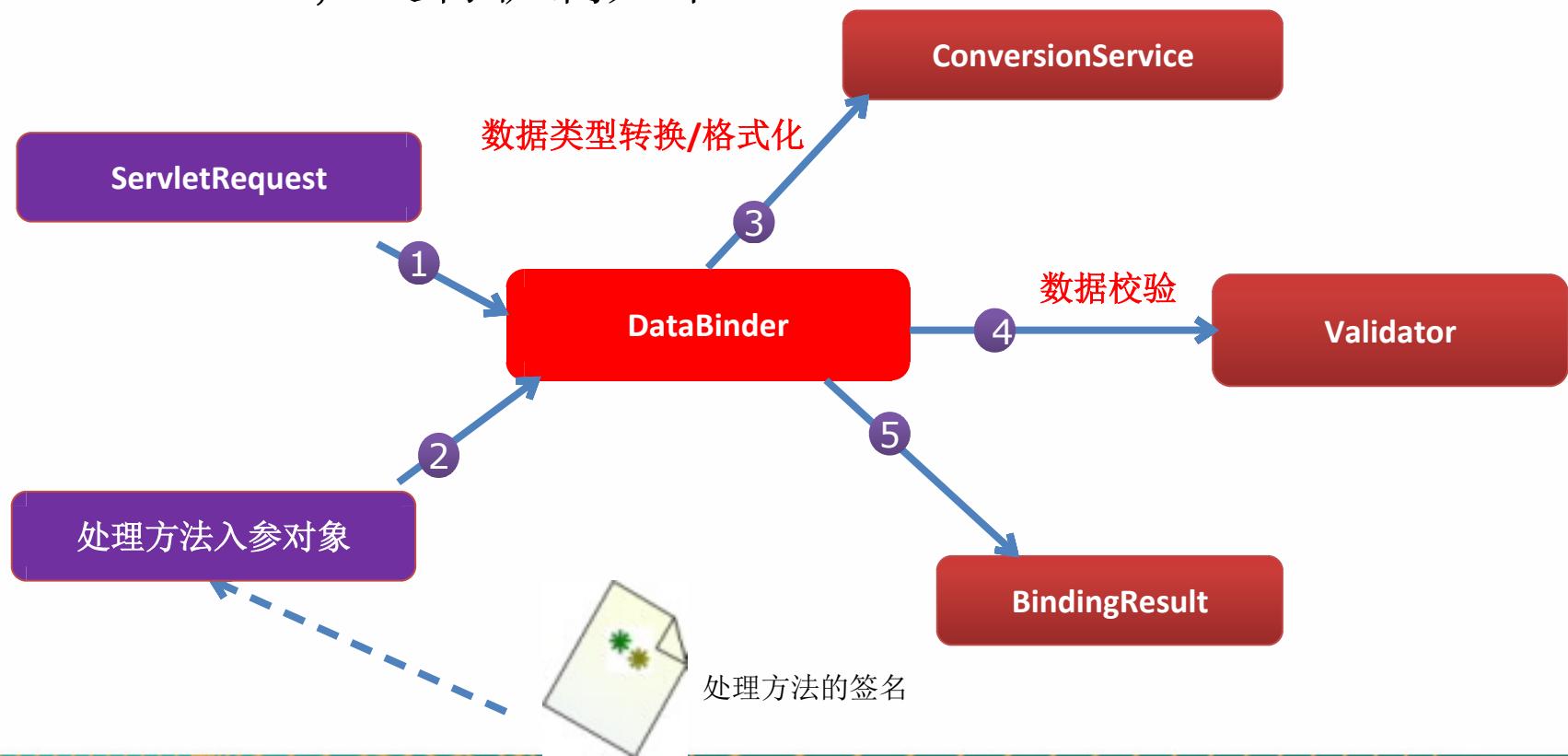
- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- **数据转换 & 数据格式化 & 数据校验**
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

数据绑定流程

- 1. Spring MVC 主框架将 `ServletRequest` 对象及目标方法的入参实例传递给 `WebDataBinderFactory` 实例，以创建 **DataBinder** 实例对象
- 2. `DataBinder` 调用装配在 Spring MVC 上下文中的 **ConversionService** 组件进行**数据类型转换、数据格式化**工作。将 `Servlet` 中的请求信息填充到入参对象中
- 3. 调用 **Validator** 组件对已经绑定了请求消息的入参对象进行数据合法性校验，并最终生成数据绑定结果 **BindingData** 对象
- 4. Spring MVC 抽取 **BindingResult** 中的入参对象和校验错误对象，将它们赋给处理方法的响应入参

数据绑定流程

- Spring MVC 通过反射机制对目标处理方法进行解析，将请求消息绑定到处理方法的入参中。数据绑定的核心部件是 **DataBinder**，运行机制如下：



数据绑定流程

```
WebDataBinder binder = binderFactory.createBinder(request, attribute, name);
if (binder.getTarget() != null) {
    bindRequestParameters(binder, request);
    validateIfApplicable(binder, parameter);
    if (binder.getBindingResult().hasErrors()) {
        if (isBindExceptionRequired(binder, parameter)) {
            throw new BindException(binder.getBindingResult());
        }
    }
}
```

数据绑定流程

▲ ● this	ExtendedServletRequestDataBinder (id=138)
■ allowedFields	null
■ autoGrowCollectionLimit	256
■ autoGrowNestedPaths	true
■ bindEmptyMultipartFiles	true
▷ ■ bindingErrorProcessor	DefaultBindingErrorProcessor (id=206)
▷ ■ bindingResult	BeanPropertyBindingResult (id=209)
▷ ■ conversionService	DefaultConversionService (id=134)
■ disallowedFields	null
▷ ■ fieldDefaultPrefix	"!" (id=216)
▷ ■ fieldMarkerPrefix	"_" (id=218)
■ ignoreInvalidFields	false
■ ignoreUnknownFields	true
▷ ■ objectName	"user" (id=194)
■ requiredFields	null
▷ ■ target	User (id=195)
■ typeConverter	null
▷ ■ validators	ArrayList<E> (id=226)

数据转换

- Spring MVC 上下文中内建了很多转换器，可完成大多数 Java 类型的转换工作。
- ConversionService converters =
 - **java.lang.Boolean -> java.lang.String** : org.springframework.core.convert.support.ObjectToStringConverter@f874ca
 - **java.lang.Character -> java.lang.Number** : CharacterToNumberFactory@f004c9
 - **java.lang.Character -> java.lang.String** : ObjectToStringConverter@68a961
 - **java.lang.Enum -> java.lang.String** : EnumToStringConverter@12f060a
 - **java.lang.Number -> java.lang.Character** : NumberToCharacterConverter@1482ac5
 - **java.lang.Number -> java.lang.Number** : NumberToNumberConverterFactory@126c6f
 - **java.lang.Number -> java.lang.String** : ObjectToStringConverter@14888e8
 - **java.lang.String -> java.lang.Boolean** : StringToBooleanConverter@1ca6626
 - **java.lang.String -> java.lang.Character** : StringToCharacterConverter@1143800
 - **java.lang.String -> java.lang.Enum** : StringToEnumConverterFactory@1bba86e
 - **java.lang.String -> java.lang.Number** : StringToNumberConverterFactory@18d2c12
 - **java.lang.String -> java.util.Locale** : StringToLocaleConverter@3598e1
 - **java.lang.String -> java.util.Properties** : StringToPropertiesConverter@c90828
 - **java.lang.String -> java.util.UUID** : StringToUUIDConverter@a42f23
 - **java.util.Locale -> java.lang.String** : ObjectToStringConverter@c7e20a
 - **java.util.Properties -> java.lang.String** : PropertiesToStringConverter@367a7f
 - **java.util.UUID -> java.lang.String** : ObjectToStringConverter@112b07f

自定义类型转换器

- **ConversionService** 是 Spring 类型转换体系的核心接口。
- 可以利用 **ConversionServiceFactoryBean** 在 Spring 的 IOC 容器中定义一个 **ConversionService**. **Spring** 将自动识别出 IOC 容器中的 **ConversionService**, 并在 **Bean** 属性配置及 **Spring MVC** 处理方法入参绑定等场合使用它进行数据的转换
- 可通过 **ConversionServiceFactoryBean** 的 **converters** 属性注册自定义的类型转换器

```
<bean id="conversionService" class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <list>
            <bean class="com.atguigu.springmvc.UserConverter"></bean>
        </list>
    </property>
</bean>
```

Spring 支持的转换器

- Spring 定义了 3 种类型的转换器接口， 实现任意一个转换器接口都可以作为自定义转换器注册到 **ConversionServiceFactroyBean** 中：
 - **Converter<S,T>**：将 S 类型对象转为 T 类型对象
 - **ConverterFactory**：将相同系列多个“同质” Converter 封装在一起。如果希望将一种类型的对象转换为另一种类型及其子类的对象（例如将 String 转换为 Number 及 Number 子类（Integer、Long、Double 等）对象）可使用该转换器工厂类
 - **GenericConverter**：会根据源类对象及目标类对象所在的宿主类中的上下文信息进行类型转换

自定义转换器示例

- **<mvc:annotation-driven conversion-service=“conversionService”/>** 会将自定义的 ConversionService 注册到 Spring MVC 的上下文中

```
<mvc:annotation-driven conversion-service="conversionService"></mvc:annotation-driven>

<bean id="conversionService" class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <list>
            <bean class="com.atguigu.springmvc.UserConverter"></bean>
        </list>
    </property>
</bean>
```

```
@RequestMapping("/handle24")
public String handle24(@RequestParam("user") User user){
    System.out.println(user);
    return "success";
}
```

关于 mvc:annotation-driven

- <mvc:annotation-driven /> 会自动注册**RequestMappingHandlerMapping**、**RequestMappingHandlerAdapter** 与 **ExceptionHandlerExceptionResolver** 三个bean。
- 还将提供以下支持：
 - 支持使用 **ConversionService** 实例对表单参数进行类型转换
 - 支持使用 **@NumberFormat annotation**、**@DateTimeFormat** 注解完成数据类型的格式化
 - 支持使用 **@Valid** 注解对 JavaBean 实例进行 JSR 303 验证
 - 支持使用 **@RequestBody** 和 **@ResponseBody** 注解

既没有配置 <mvc:default-servlet-handler/> 也没有配置 <mvc:annotation-driven/>

handlerAdapters	ArrayList<E> (id=199)
elementData	Object[3] (id=215)
[0]	HttpRequestHandlerAdapter (id=222)
[1]	SimpleControllerHandlerAdapter (id=224)
[2]	AnnotationMethodHandlerAdapter (id=79)

配置了 <mvc:default-servlet-handler/> 但没有配置 <mvc:annotation-driven/>

handlerAdapters	ArrayList<E> (id=121)
elementData	Object[2] (id=133)
[0]	HttpRequestHandlerAdapter (id=69)
[1]	SimpleControllerHandlerAdapter (id=140)

既配置了 <mvc:default-servlet-handler/> 又配置 <mvc:annotation-driven/>

handlerAdapters	ArrayList<E> (id=121)
elementData	Object[3] (id=135)
[0]	HttpRequestHandlerAdapter (id=142)
[1]	SimpleControllerHandlerAdapter (id=144)
[2]	RequestMappingHandlerAdapter (id=70)

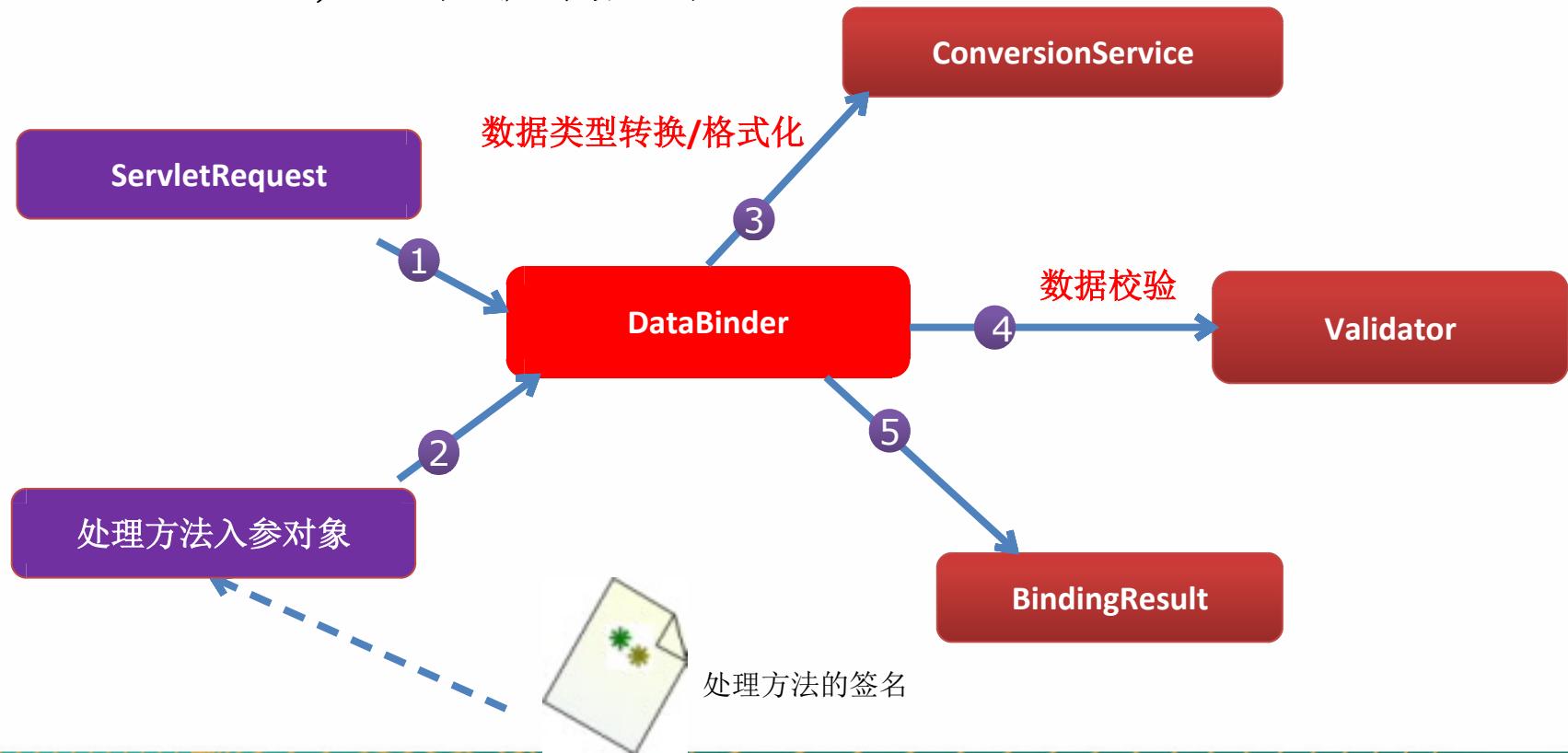
@InitBinder

- 由 **@InitBinder** 标识的方法，可以对 **WebDataBinder** 对象进行初始化。**WebDataBinder** 是 **DataBinder** 的子类，用于完成由表单字段到 **JavaBean** 属性的绑定
- @InitBinder**方法不能有返回值，它必须声明为**void**。
- @InitBinder**方法的参数通常是是 **WebDataBinder**

```
/**
 * 不自动绑定对象中的 roleSet 属性，另行处理。
 */
@InitBinder
public void initBinder(WebDataBinder dataBinder){
    dataBinder.setDisallowedFields("roleSet");
}
```

数据绑定流程

- Spring MVC 通过反射机制对目标处理方法进行解析，将请求消息绑定到处理方法的入参中。数据绑定的核心部件是 **DataBinder**，运行机制如下：



内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & **数据格式化** & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

数据格式化

- 对属性对象的输入/输出进行格式化，从其本质上讲依然属于“类型转换”的范畴。
- Spring 在格式化模块中定义了一个实现 `ConversionService` 接口的 **FormattingConversionService** 实现类，该实现类扩展了 `GenericConversionService`，因此它既具有类型转换的功能，又具有格式化的功能
- `FormattingConversionService` 拥有一个 **FormattingConversionServiceFactoryBean** 工厂类，后者用于在 Spring 上下文中构造前者

数据格式化

- FormattingConversionServiceFactroyBean 内部已经注册了：
 - NumberFormatAnnotationFormatterFactroy : 支持对数字类型的属性使用 **@NumberFormat** 注解
 - JodaDateTimeFormatAnnotationFormatterFactroy : 支持对日期类型的属性使用 **@DateTimeFormat** 注解
- 装配了 FormattingConversionServiceFactroyBean 后，就可以在 Spring MVC 入参绑定及模型数据输出时使用注解驱动了。 **<mvc:annotation-driven/>** 默认创建的 **ConversionService** 实例即为 **FormattingConversionServiceFactroyBean**

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

日期格式化

- **@DateTimeFormat** 注解可对 **java.util.Date**、**java.util.Calendar**、**java.long.Long** 时间类型进行标注：
 - **pattern** 属性：类型为字符串。指定解析/格式化字段数据的模式，如：“**yyyy-MM-dd hh:mm:ss**”
 - **iso** 属性：类型为 **DateTimeFormat.ISO**。指定解析/格式化字段数据的**ISO**模式，包括四种：**ISO.NONE**（不使用） -- 默认、**ISO.DATE**(**yyyy-MM-dd**)、**ISO.TIME**(**hh:mm:ss.SSSZ**)、**ISO.DATE_TIME**(**yyyy-MM-dd hh:mm:ss.SSSZ**)
 - **style** 属性：字符串类型。通过样式指定日期时间的格式，由两位字符组成，第一位表示日期的格式，第二位表示时间的格式：**S**：短日期/时间格式、**M**：中日期/时间格式、**L**：长日期/时间格式、**F**：完整日期/时间格式、**-**：忽略日期或时间格式

数值格式化

- **@NumberFormat** 可对类似数字类型的属性进行标注，它拥有两个互斥的属性：
 - **style** : 类型为 NumberFormat.Style。用于指定样式类型，包括三种：**Style.NUMBER**（正常数字类型）、**Style.CURRENCY**（货币类型）、**Style.PERCENT**（百分数类型）
 - **pattern** : 类型为 String, 自定义样式，如 pattern="#,###" ;

格式化示例

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

public class User {

    @DateTimeFormat(pattern="yyyy/MM/dd")
    private Date birthday;

    @NumberFormat(pattern="#,###.##")

    @RequestMapping("/handle19")
    public String handle19(@ModelAttribute("user") User user){
        user.setId(1000);
        System.out.println(user);
        return "success";
    }
}
```

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & **数据校验**
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

JSR 303

- **JSR 303** 是 Java 为 Bean 数据合法性校验提供的标准框架，它已经包含在 JavaEE 6.0 中。
- JSR 303 通过**在 Bean 属性上标注**类似于 @NotNull、@Max 等标准的注解指定校验规则，并通过标准的验证接口对 Bean 进行验证

注解

@Null
@NotNull
@AssertTrue
@AssertFalse
@Min(value)
@Max(value)
@DecimalMin(value)
@DecimalMax(value)
@Size(max, min)
@Digits(integer, fraction)
@Past
@Future
@Pattern(value)

功能说明

被注释的元素必须为 null
被注释的元素必须不为 null
被注释的元素必须为 true
被注释的元素必须为 false
被注释的元素必须是一个数字，其值必须大于等于指定的最小值
被注释的元素必须是一个数字，其值必须小于等于指定的最大值
被注释的元素必须是一个数字，其值必须大于等于指定的最小值
被注释的元素必须是一个数字，其值必须小于等于指定的最大值
被注释的元素的大小必须在指定的范围内
被注释的元素必须是一个数字，其值必须在可接受的范围内
被注释的元素必须是一个过去的日期
被注释的元素必须是一个将来的日期
被注释的元素必须符合指定的正则表达式

Hibernate Validator 扩展注解

- **Hibernate Validator** 是 JSR 303 的一个参考实现，除支持所有标准的校验注解外，它还支持以下的扩展注解

注解

@Email
@Length
@NotEmpty
@Range

功能说明

被注释的元素必须是电子邮箱地址
被注释的字符串的大小必须在指定的范围内
被注释的字符串的必须非空
被注释的元素必须在合适的范围内

Spring MVC 数据校验

- Spring 4.0 拥有自己独立的数据校验框架，同时支持 JSR 303 标准的校验框架。
- Spring 在进行数据绑定时，可同时调用校验框架完成数据校验工作。在 **Spring MVC** 中，可直接通过注解驱动的方式进行数据校验
- Spring 的 LocalValidatorFactoryBean 既实现了 Spring 的 Validator 接口，也实现了 JSR 303 的 Validator 接口。只要在 Spring 容器中定义了一个 **LocalValidatorFactoryBean**，即可将其注入到需要数据校验的 Bean 中。
- Spring 本身并没有提供 **JSR303** 的实现，所以**必须将 JSR303** 的实现者的 jar 包放到类路径下。

Spring MVC 数据校验

- <**mvc:annotation-driven/**> 会默认装配好一个 **LocalValidatorFactoryBean**, 通过在处理方法的入参上标注 **@valid** 注解即可让 Spring MVC 在完成数据绑定后执行数据校验的工作
- 在已经标注了 JSR303 注解的表单/命令对象前标注一个 **@Valid**, Spring MVC 框架在将请求参数绑定到该入参对象后, 就会调用校验框架根据注解声明的校验规则实施校验
- Spring MVC 是通过对处理方法签名的规约来保存校验结果的: 前一个表单/命令对象的校验结果保存到随后的入参中, 这个保存校验结果的入参必须是 **BindingResult** 或 **Errors** 类型, 这两个类都位于 `org.springframework.validation` 包中

Spring MVC 数据校验

- 需校验的 **Bean** 对象和其绑定结果对象或错误对象时成对出现的，它们之间不允许声明其他的入参
- Errors 接口提供了获取错误信息的方法，如 getErrorCount() 或 getFieldErrors(String field)
- BindingResult 扩展了 Errors 接口

User和其绑定结果的对象

```
public String handle 91(@Valid User user, BindingResult userBindingResult,  
String sessionId, ModelMap mm,  
@Valid Dept dept, Errors deptErrors){
```

Dept和其校验的结果对象

在目标方法中获取校验结果

- 在表单/命令对象类的属性中标注校验注解，在处理方法对应的入参前添加 `@Valid`, Spring MVC 就会实施校验并将校验结果保存在被校验入参对象之后的 `BindingResult` 或 `Errors` 入参中。
- 常用方法：
 - `FieldError getFieldError(String field)`
 - `List<FieldError> getFieldErrors()`
 - `Object getFieldValue(String field)`
 - `Int getErrorCount()`

在页面上显示错误

- Spring MVC 除了会将表单/命令对象的校验结果保存到对应的 BindingResult 或 Errors 对象中外，**还会将所有校验结果保存到“隐含模型”**
- 即使处理方法的签名中没有对应于表单/命令对象的结果入参，校验结果也会保存在“隐含对象”中。
- 隐含模型中的所有数据最终将通过 HttpServletRequest 的属性列表暴露给 JSP 视图对象，因此在 JSP 中可以获取错误信息
- 在 JSP 页面上可通过 **<form:errors path="userName">** 显示错误消息

示例

```
<form:form action= "hello/handle19.action" modelAttribute= "user">
    <form:errors path= "*"></form:errors>

    name: <input type= "text" name= "userName"/>
    <form:errors path= "userName"></form:errors>

    email: <input type= "text" name= "email"/>
    <form:errors path= "email"></form:errors>

    <input type= "submit" value= "Submit"/>
</form:form>
```

index.jsp

示例

```
public class User {  
  
    @DateTimeFormat(pattern="yyyy/MM/dd")  
    @Past  
    private Date birthday;  
  
    @NumberFormat(pattern="#,###.##")  
    @DecimalMax("9999")  
    @DecimalMin("1000")  
    private long salary;  
  
    @Pattern(regexp="\w{4,30}")  
    private String userName;  
  
    @Email  
    private String email;  
}
```

User

示例

```
@RequestMapping("/handle19")
public String handle19(@Valid @ModelAttribute("user") User user,
                      BindingResult bindingResult){

    if(bindingResult.hasErrors()){
        return "forward:/index.jsp";
    }else{
        System.out.println("验证通过....");
    }

    user.setId(1000);
    System.out.println(user);
    return "success";
}
```

目标方法

提示消息的国际化

- 每个属性在数据绑定和数据校验发生错误时，都会生成一个对应的 **FieldError** 对象。
- 当一个属性校验失败后，校验框架会为该属性生成 **4** 个消息代码，这些代码以校验注解类名为**前缀**，结合 **modelAttribute**、**属性名及属性类型名**生成多个对应的消息代码：例如 User 类中的 password 属性标准了一个 @Pattern 注解，当该属性值不满足 @Pattern 所定义的规则时，就会产生以下 4 个错误代码：
 - Pattern.user.password
 - Pattern.password
 - Pattern.java.lang.String
 - Pattern
- 当使用 **Spring MVC** 标签显示错误消息时，**Spring MVC** 会查看 **WEB** 上下文是否装配了对应的国际化消息，如果没有，则显示默认的错误消息，否则使用国际化消息。

提示消息的国际化

- 若数据**类型转换**或**数据格式转换**时发生错误，或**该有的参数不存在**，或**调用处理方法时发生错误**，都会在隐含模型中创建错误消息。其错误代码前缀说明如下：
 - **required**：必要的参数不存在。如 @RequiredParam("param1") 标注了一个入参，但是该参数不存在
 - **typeMismatch**：在数据绑定时，发生数据类型不匹配的问题
 - **methodInvocation**：Spring MVC 在调用处理方法时发生了错误
- 注册国际化资源文件

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

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- **处理 JSON : 使用 HttpMessageConverter**
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

处理 JSON

- 1. 加入 jar 包：

-  jackson-annotations-2.2.2.jar
-  jackson-core-2.2.2.jar
-  jackson-databind-2.2.2.jar

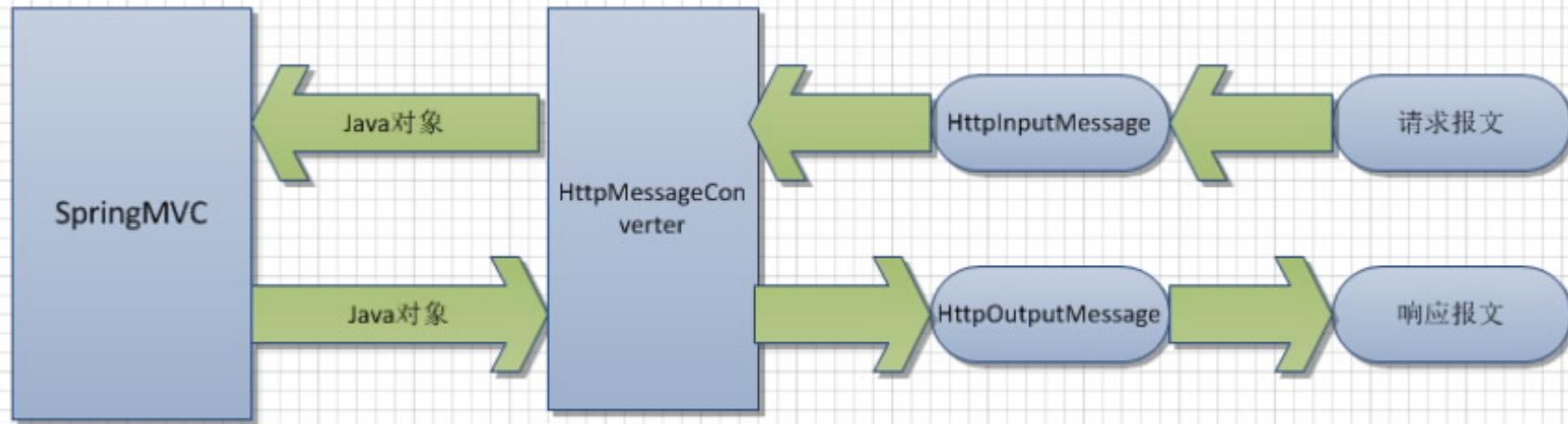
- 2. 编写目标方法，使其返回 JSON 对应的对象或集合
- 3. 在方法上添加 @ResponseBody 注解

```
@ResponseBody
@RequestMapping("/getUsers")
public List<User> testAjax(){
    List<User> users = new ArrayList<>();
    users.add(new User(1, "a", new Date(), 1000));
    users.add(new User(2, "b", new Date(), 2000));
    return users;
}
```

HttpMessageConverter<T>

- **HttpMessageConverter<T>** 是 Spring3.0 新添加的一个接口，负责将请求信息转换为一个对象（类型为 T），将对象（类型为 T）输出为响应信息
- **HttpMessageConverter<T>** 接口定义的方法：
 - Boolean canRead(Class<?> clazz, MediaType mediaType): 指定转换器可以读取的对象类型，即转换器是否可将请求信息转换为 clazz 类型的对象，同时指定支持 MIME 类型(text/html,application/json等)
 - Boolean canWrite(Class<?> clazz, MediaType mediaType): 指定转换器是否可将 clazz 类型的对象写到响应流中，响应流支持的媒体类型在 MediaType 中定义。
 - List<MediaType> getSupportMediaTypes(): 该转换器支持的媒体类型。
 - T read(Class<? extends T> clazz, **HttpInputMessage** inputMessage): 将请求信息流转换为 T 类型的对象。
 - void write(T t, MediaType contnetType, **HttpOutputMessgae** outputMessage): 将 T 类型的对象写到响应流中，同时指定相应的媒体类型为 contnetType。

HttpMessageConverter<T>



HttpMessageConverter<T> 的实现类

实现类	功能说明
StringHttpMessageConverter	将请求信息转换为字符串
FormHttpMessageConverter	将表单数据读取到 MultiValueMap 中
XmIAwareFormHttpMessageConverter	扩展于 FormHttpMessageConverter, 如果部分表单属性是 XML 数据 , 可用该转换器进行读取
ResourceHttpMessageConverter	读写 org.springframework.core.io.Resource 对象
BufferedImageHttpMessageConverter	读写 BufferedImage 对象
ByteArrayHttpMessageConverter	读写二进制数据
SourceHttpMessageConverter	读写 javax.xml.transform.Source 类型的数据
MarshallingHttpMessageConverter	通过 Spring 的 org.springframework.xml.Marshaller 和 Unmarshaller 读写 XML 消息
Jaxb2RootElementHttpMessageConverter	通过 JAXB2 读写 XML 消息, 将请求消息转换到标注 XmlRootElement 和 XmlType 直接的类中
MappingJacksonHttpMessageConverter	利用 Jackson 开源包的 ObjectMapper 读写 JSON 数据
RssChannelHttpMessageConverter	能够读写 RSS 种子消息
AtomFeedHttpMessageConverter	和 RssChannelHttpMessageConverter 能够读写 RSS 种子消息

HttpMessageConverter<T>

- DispatcherServlet 默认装配
RequestMappingHandlerAdapter , 而
RequestMappingHandlerAdapter 默认装配如下
HttpMessageConverter :

messageConverters	ArrayList<E> (id=255)
elementData	Object[6] (id=260)
[0]	ByteArrayHttpMessageConverter (id=263)
[1]	StringHttpMessageConverter (id=266)
[2]	ResourceHttpMessageConverter (id=269)
[3]	SourceHttpMessageConverter<T> (id=271)
[4]	AllEncompassingFormHttpMessageConverter (id=274)
[5]	Jaxb2RootElementHttpMessageConverter (id=277)

HttpMessageConverter<T>

- 加入 jackson jar 包后，`RequestMappingHandlerAdapter` 装配的 `HttpMessageConverter` 如下：

messageConverters	ArrayList<E> (id=261)
elementData	Object[7] (id=269)
[0]	ByteArrayHttpMessageConverter (id=280)
[1]	StringHttpMessageConverter (id=284)
[2]	ResourceHttpMessageConverter (id=286)
[3]	SourceHttpMessageConverter<T> (id=288)
[4]	AllEncompassingFormHttpMessageConverter (id=290)
[5]	Jaxb2RootElementHttpMessageConverter (id=292)
[6]	MappingJackson2HttpMessageConverter (id=295)

使用 `HttpMessageConverter<T>`

- 使用 `HttpMessageConverter<T>` 将请求信息转化并绑定到处理方法的入参中或将响应结果转为对应类型的响应信息，**Spring** 提供了两种途径：
 - 使用 `@RequestBody / @ResponseBody` 对处理方法进行标注
 - 使用 `HttpEntity<T> / ResponseEntity<T>` 作为处理方法的入参或返回值
- 当控制器处理方法使用到 `@RequestBody/@ResponseBody` 或 `HttpEntity<T>/ResponseEntity<T>` 时，**Spring** 首先根据请求头或响应头的 **Accept** 属性选择匹配的 `HttpMessageConverter`，进而根据参数类型或泛型类型的过滤得到匹配的 `HttpMessageConverter`，若找不到可用的 `HttpMessageConverter` 将报错
- `@RequestBody` 和 `@ResponseBody` 不需要成对出现

@RequestBody、@ResponseBody 示例

```
@ResponseBody  
@RequestMapping("/handle15")  
public byte[] handle15() throws IOException{  
    Resource resource = new ClassPathResource("/Lighthouse.jpg");  
    byte[] fileData = FileCopyUtils.copyToByteArray(resource.getInputStream());  
    return fileData;  
}
```

```
@RequestMapping(value="/handle14", method=RequestMethod.POST)  
public String handle14(@RequestBody String requestBody){  
    System.out.println(requestBody);  
    return "success";  
}
```

messageConverters	ArrayList<E> (id=261)
elementData	Object[7] (id=269)
[0]	ByteArrayHttpMessageConverter (id=280)
[1]	StringHttpMessageConverter (id=284)
[2]	ResourceHttpMessageConverter (id=286)
[3]	SourceHttpMessageConverter<T> (id=288)
[4]	AllEncompassingFormHttpMessageConverter (id=290)
[5]	Jaxb2RootElementHttpMessageConverter (id=292)
[6]	MappingJackson2HttpMessageConverter (id=295)

HttpEntity、 ResponseEntity 示例

```
@RequestMapping("/handle16")
public String handle16(HttpEntity<String> entity){
    System.out.println(entity.getHeaders().getContentLength());
    return "success";
}

@RequestMapping("/handle17")
public ResponseEntity<byte[]> handle17() throws IOException{
    Resource resource = new ClassPathResource("/Lighthouse.jpg");
    byte[] fileData = FileCopyUtils.copyToByteArray(resource.getInputStream());
    ResponseEntity<byte[]> responseEntity =
        new ResponseEntity<byte[]>(fileData, HttpStatus.OK);
    return responseEntity;
}
```

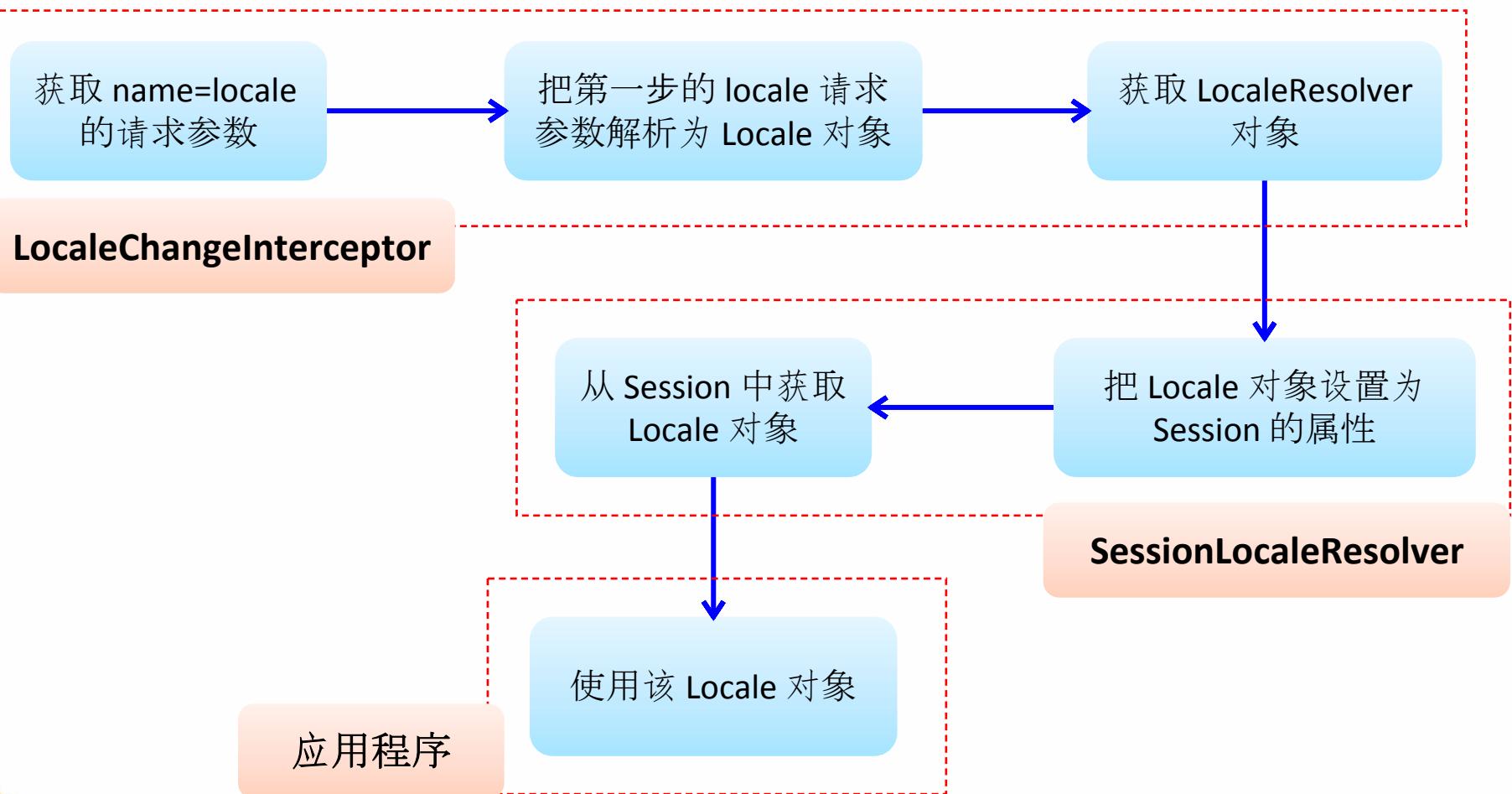
内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

国际化概述

- 默认情况下，SpringMVC 根据 **Accept-Language** 参数判断客户端的本地化类型。
- 当接受到请求时，SpringMVC 会在上下文中查找一个本地化解析器（**LocalResolver**），找到后使用它获取请求所对应的本地化类型信息。
- SpringMVC 还允许装配一个**动态更改本地化类型的拦截器**，这样通过指定一个请求参数就可以控制单个请求的本地化类型。

SessionLocaleResolver & LocaleChangeInterceptor 工作原理



本地化解析器和本地化拦截器

- **AcceptHeaderLocaleResolver** : 根据 **HTTP** 请求头的 **Accept-Language** 参数确定本地化类型，如果没有显式定义本地化解析器， SpringMVC 使用该解析器。
- **CookieLocaleResolver** : 根据指定的 **Cookie** 值确定本地化类型
- **SessionLocaleResolver** : 根据 **Session** 中特定的属性确定本地化类型
- **LocaleChangeInterceptor** : 从请求参数中获取本次请求对应的本地化类型。

```
<bean id="LocaleResolver"  
      class="org.springframework.web.servlet.i18n.SessionLocaleResolver"></bean>  
<mvc:interceptors>  
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"></bean>  
</mvc:interceptors>
```

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

文件上传

- Spring MVC 为文件上传提供了直接的支持，这种支持是通过即插即用的 **MultipartResolver** 实现的。Spring 用 **Jakarta Commons FileUpload** 技术实现了一个 MultipartResolver 实现类：**CommonsMultipartResovler**
- Spring MVC 上下文中默认没有装配 MultipartResovler，因此默认情况下不能处理文件的上传工作，如果想使用 Spring 的文件上传功能，需现在上下文中配置 MultipartResolver

配置 MultipartResolver

- defaultEncoding: 必须和用户 JSP 的 pageEncoding 属性一致，以便正确解析表单的内容
- 为了让 **CommonsMultipartResolver** 正确工作，必须先将 Jakarta Commons FileUpload 及 Jakarta Commons io 的类包添加到类路径下。

```
<bean id= "multipartResolver"
  class= "org.springframework.web.multipart.commons.CommonsMultipartResolver">
  <property name= "defaultEncoding" value= "UTF-8"></property>
  <property name= "maxUploadSize" value= "5242880"></property>
</bean>
```

文件上传示例

```
<form method="post" action="hello/upload.action" enctype="multipart/form-data">

    Desc: <input type="text" name="desc"/>
    File: <input type="file" name="file"/>
    <input type="submit" value="Submit"/>

</form>
```

```
@RequestMapping("/upload")
public String upload(@RequestParam("desc") String desc,
                     @RequestParam("file") MultipartFile file) throws IllegalStateException, IOException{
    if(!file.isEmpty()){
        System.out.println("desc");
        file.transferTo(new File("d:\\temp\\\" + file.getOriginalFilename()));
    }

    return "success";
}
```

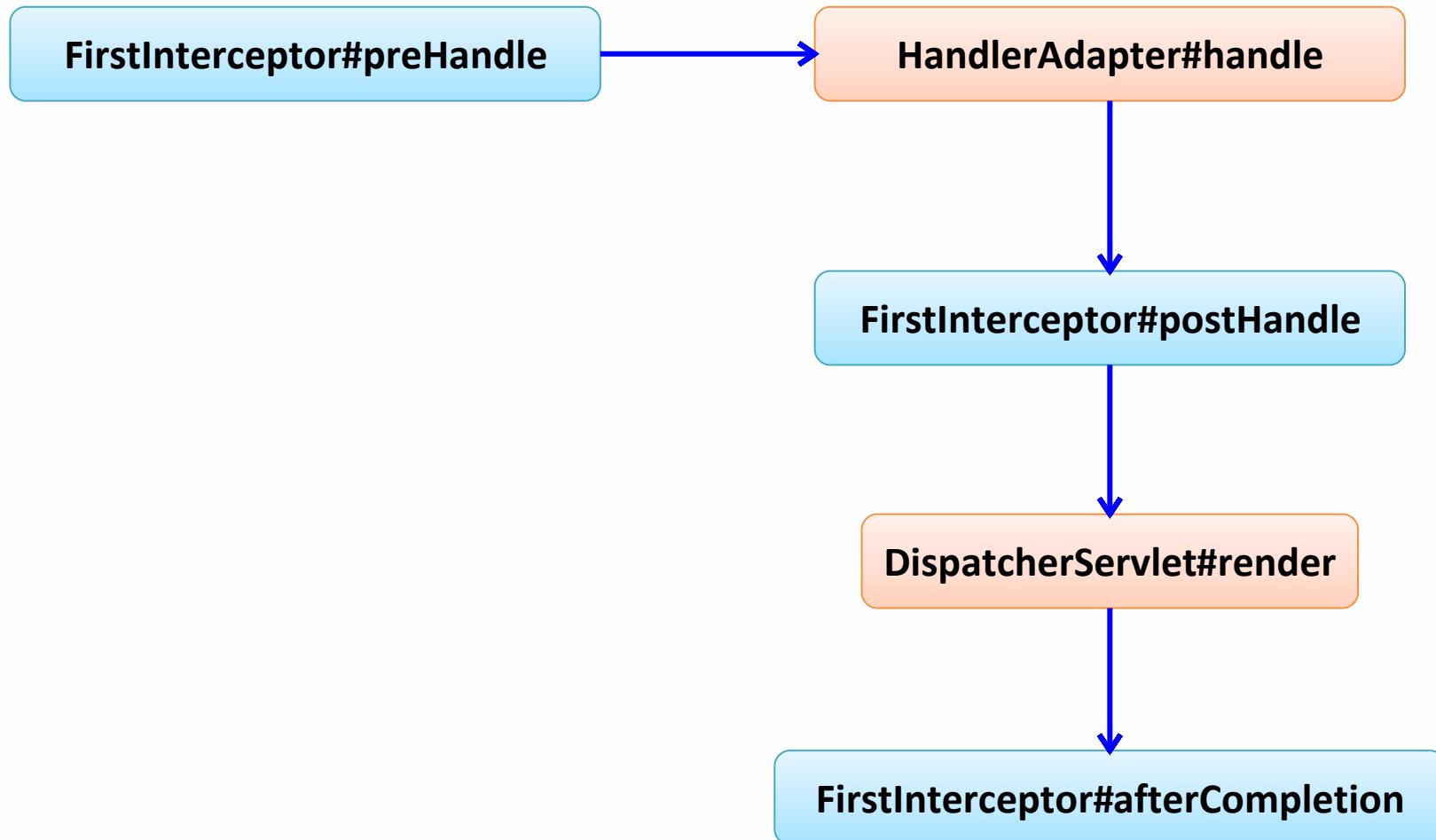
内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

自定义拦截器

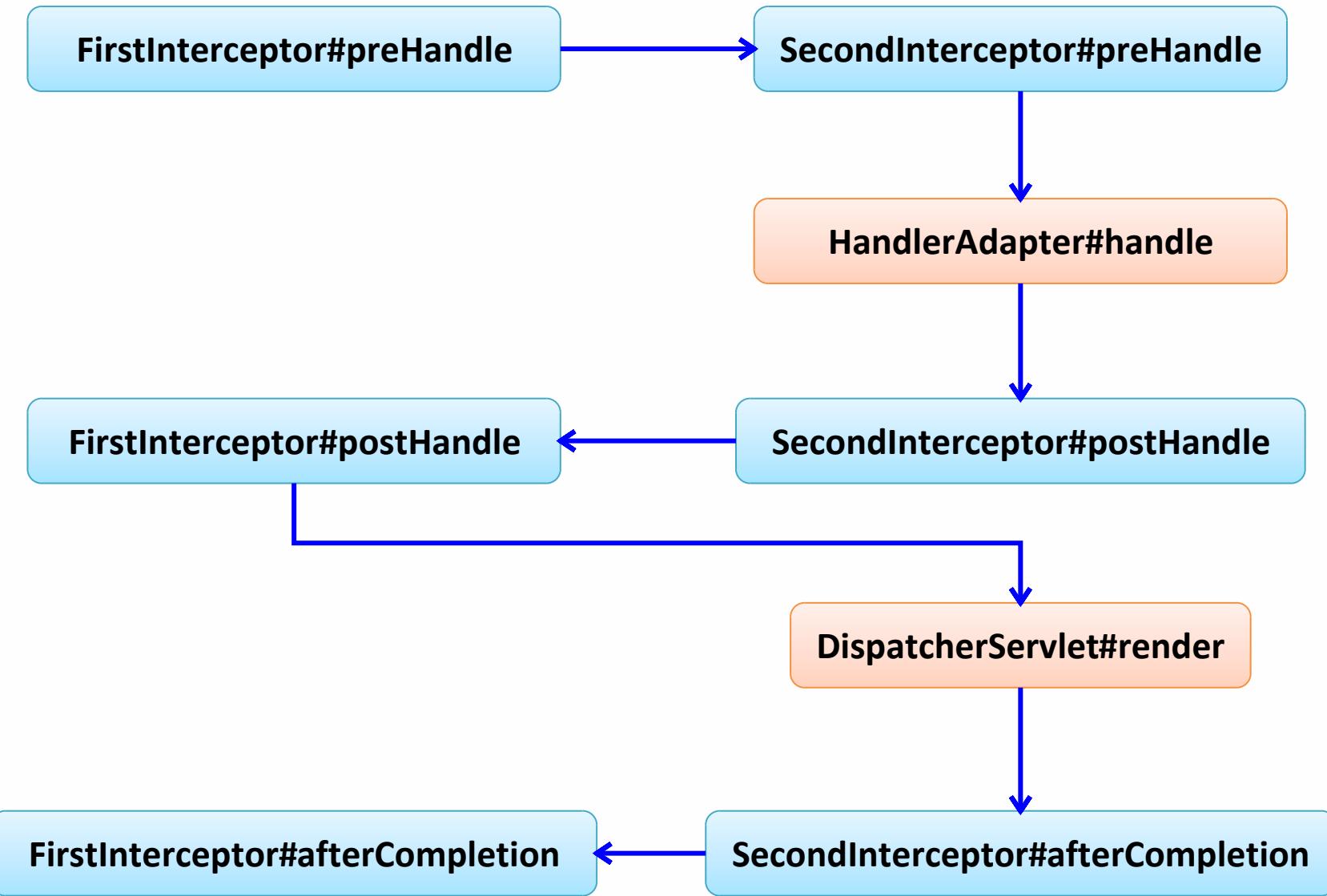
- Spring MVC也可以使用拦截器对请求进行拦截处理，用户可以自定义拦截器来实现特定的功能，**自定义的拦截器必须实现HandlerInterceptor接口**
 - **preHandle()**：这个方法在业务处理器处理请求之前被调用，在该方法中对用户请求 request 进行处理。如果程序员决定该拦截器对请求进行拦截处理后还要调用其他的拦截器，或者是业务处理器去进行处理，则返回**true**；如果程序员决定不再调用其他的组件去处理请求，则返回**false**。
 - **postHandle()**：这个方法在业务处理器处理完请求后，但是**DispatcherServlet** 向客户端返回响应前被调用，在该方法中对用户请求request进行处理。
 - **afterCompletion()**：这个方法在 **DispatcherServlet** 完全处理完请求后被调用，可以在该方法中进行一些资源清理的操作。

拦截器方法执行顺序



配置自定义拦截器

```
<mvc:interceptors>
    <!-- 拦截所有资源 -->
    <bean class="com.atguigu.springmvc.interceptors.HelloInterceptor"></bean>
    <!-- 拦截指定资源 -->
    <mvc:interceptor>
        <mvc:mapping path="/emps"/>
        <bean class="com.atguigu.springmvc.interceptors.HelloInterceptor2"></bean>
    </mvc:interceptor>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"></bean>
</mvc:interceptors>
```



return false

FirstInterceptor#preHandle

SecondInterceptor#preHandle

HandlerAdapter#handle

FirstInterceptor#postHandle

SecondInterceptor#postHandle

DispatcherServlet#render

FirstInterceptor#afterCompletion

SecondInterceptor#afterCompletion

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2

异常处理

- Spring MVC 通过 **HandlerExceptionResolver** 处理程序的异常，包括 Handler 映射、数据绑定以及目标方法执行时发生的异常。
- SpringMVC 提供的 HandlerExceptionResolver 的实现类

- ➊ HandlerExceptionResolver - org.springframework.web
- ➋ **G A AbstractHandlerExceptionResolver - org.springfra**
 - ➌ G A AbstractHandlerMethodExceptionResolver - c
 - ➍ G ExceptionHandlerExceptionResolver - org
 - ➎ G AnnotationMethodHandlerExceptionResolver
 - ➏ G DefaultHandlerExceptionResolver - org.spring
 - ➐ G ResponseStatusExceptionResolver - org.spring
 - ➑ G SimpleMappingExceptionResolver - org.spring
- ➒ G HandlerExceptionResolverComposite - org.spring

HandlerExceptionResolver

- DispatcherServlet 默认装配的 **HandlerExceptionResolver**：
 - 没有使用 `<mvc:annotation-driven/>` 配置：

Deprecated. as of Spring 3.2, in favor of [ExceptionHandlerExceptionResolver](#)

AnnotationMethodHandlerExceptionResolver (id=141)

ResponseStatusExceptionResolver (id=144)

DefaultHandlerExceptionResolver (id=148)

- 使用了 `<mvc:annotation-driven/>` 配置：

ExceptionHandlerExceptionResolver (id=140)

ResponseStatusExceptionResolver (id=143)

DefaultHandlerExceptionResolver (id=146)

ExceptionHandlerExceptionResolver

- 主要处理 Handler 中用 **@ExceptionHandler** 注解定义的方法。
- **@ExceptionHandler** 注解定义的方法**优先级问题**：例如发生的是`NullPointerException`, 但是声明的异常有`RuntimeException` 和 `Exception`, 此候会根据异常的最近继承关系找到继承深度最浅的那个 **@ExceptionHandler** 注解方法, 即标记了 `RuntimeException` 的方法
- `ExceptionHandlerMethodResolver` 内部若找不到`@ExceptionHandler` 注解的话, 会找 **@ControllerAdvice** 中的 **@ExceptionHandler** 注解方法

ResponseStatusExceptionResolver

- 在异常及异常父类中找到 **@ResponseStatus** 注解，然后使用这个注解的属性进行处理。
- 定义一个 **@ResponseStatus** 注解修饰的异常类

```
@ResponseStatus(HttpStatus.UNAUTHORIZED)
public class UnauthorizedException extends RuntimeException {}
```

- 若在处理器方法中抛出了上述异常：
若**ExceptionHandlerExceptionResolver** 不解析此异常。由于触发的异常 **UnauthorizedException** 带有**@ResponseStatus** 注解。因此会被**ResponseStatusExceptionResolver** 解析到。最后响应 **HttpStatus.UNAUTHORIZED** 代码给客户端。**HttpStatus.UNAUTHORIZED** 代表响应码 401，无权限。
关于其他的响应码请参考 **HttpStatus** 枚举类型源码。

DefaultHandlerExceptionResolver

- 对一些特殊的异常进行处理，比如
NoSuchRequestHandlingMethodException、
HttpRequestMethodNotSupportedException、
HttpMediaTypeNotSupportedException、
HttpMediaTypeNotAcceptableException
等。

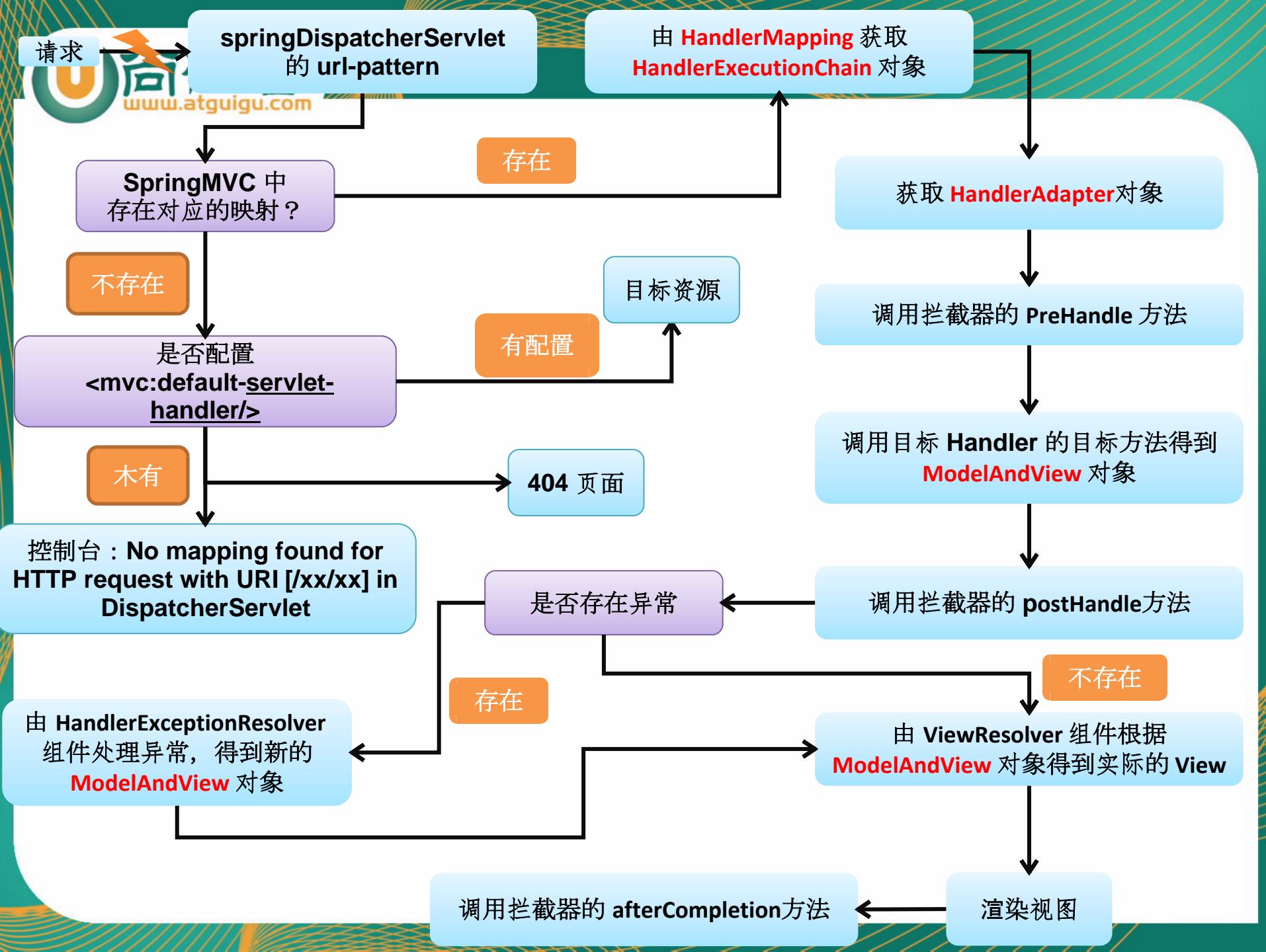
SimpleMappingExceptionResolver

- 如果希望对所有异常进行统一处理，可以使用 **SimpleMappingExceptionResolver**，它将异常类名映射为视图名，即发生异常时使用对应的视图报告异常

```
<bean id="simpleMappingExceptionResolver"
      class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <prop key="java.lang.ArithmetricException">error</prop>
        </props>
    </property>
</bean>
```

内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- **SpringMVC 运行流程**
- 在 Spring 的环境下使用 SpringMVC
- SpringMVC 对比 Struts2



内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 **Spring** 的环境下使用 **SpringMVC**
- SpringMVC 对比 Struts2

Bean 被创建两次 ?

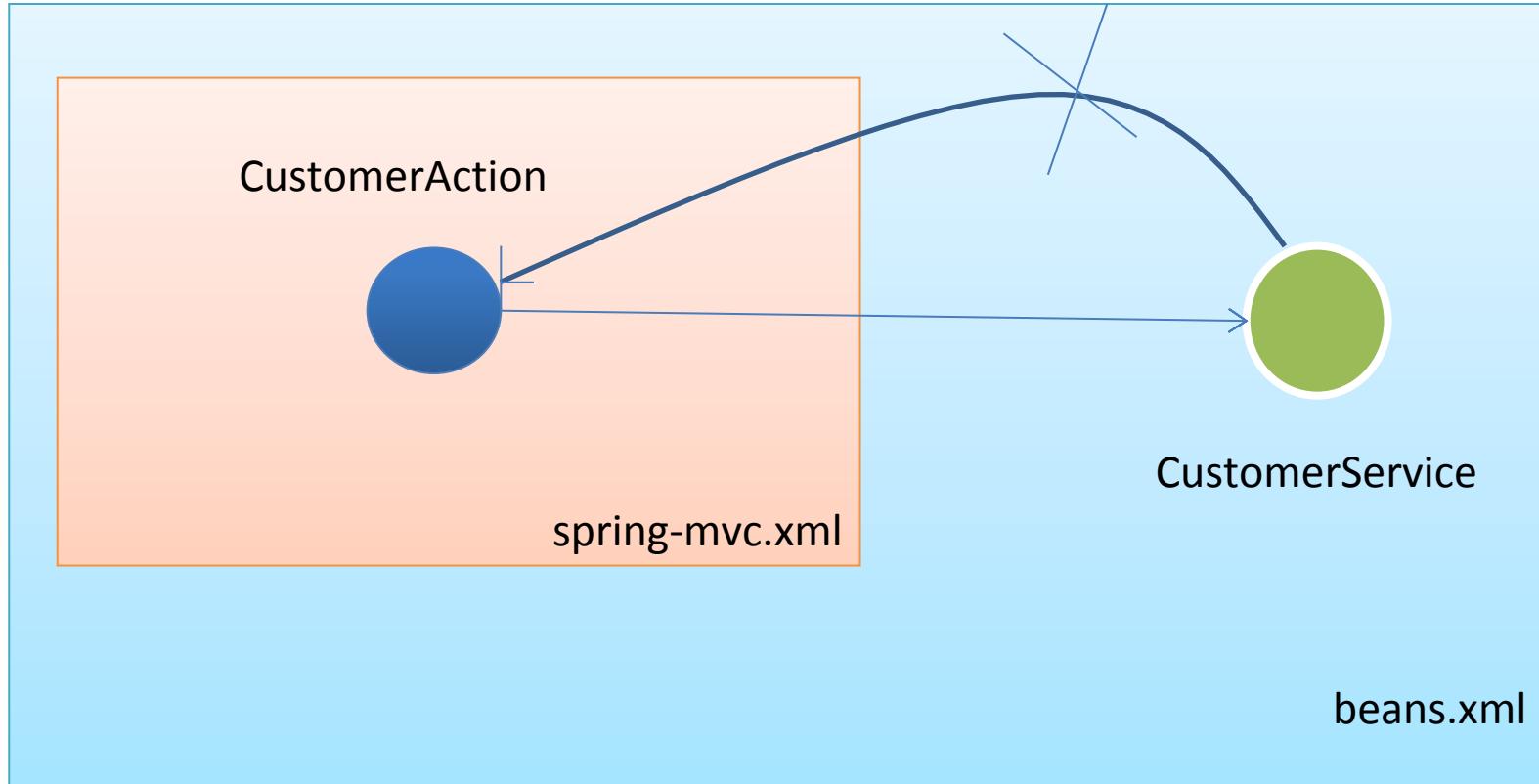
- Spring 的 IOC 容器不应该扫描 SpringMVC 中的 bean, 对应的 SpringMVC 的 IOC 容器不应该扫描 Spring 中的 bean

```
<context:component-scan base-package="com.atguigu.springmvc">
    <context:exclude-filter type="annotation"
        expression="org.springframework.stereotype.Controller"/>
    <context:exclude-filter type="annotation"
        expression="org.springframework.web.bind.annotation.ControllerAdvice"/>
</context:component-scan>

<context:component-scan base-package="com.atguigu.springmvc"
    use-default-filters="false">
    <context:include-filter type="annotation"
        expression="org.springframework.stereotype.Controller"/>
    <context:include-filter type="annotation"
        expression="org.springframework.web.bind.annotation.ControllerAdvice"/>
</context:component-scan>
```

在 Spring MVC 配置文件中引用业务层的 Bean

- 多个 Spring IOC 容器之间可以设置为父子关系，以实现良好的解耦。
- Spring MVC WEB 层容器可作为“业务层” Spring 容器的子容器：即 WEB 层容器可以引用业务层容器的 Bean，而业务层容器却访问不到 WEB 层容器的 Bean



内容概要

- SpringMVC 概述
- SpringMVC 的 HelloWorld
- 使用 @RequestMapping 映射请求
- 映射请求参数 & 请求头
- 处理模型数据
- 视图和视图解析器
- RESTful CRUD
- SpringMVC 表单标签 & 处理静态资源
- 数据转换 & 数据格式化 & 数据校验
- 处理 JSON : 使用 HttpMessageConverter
- 国际化
- 文件的上传
- 使用拦截器
- 异常处理
- SpringMVC 运行流程
- 在 Spring 的环境下使用 SpringMVC
- **SpringMVC 对比 Struts2**

SpringMVC 对比 Struts2

- ①. Spring MVC 的入口是 Servlet, 而 Struts2 是 Filter
- ②. Spring MVC 会稍微比 Struts2 快些. Spring MVC 是基于方法设计, 而 Sturts2 是基于类, 每次发一次请求都会实例一个 Action.
- ③. Spring MVC 使用更加简洁, 开发效率 Spring MVC 确实比 struts2 高: 支持 JSR303, 处理 ajax 的请求更方便
- ④. Struts2 的 OGNL 表达式使页面的开发效率相比 Spring MVC 更高些.

