LOGO

Spring MVC 3.0实战指南

参考《Spring 3.x企业应用开发实战》

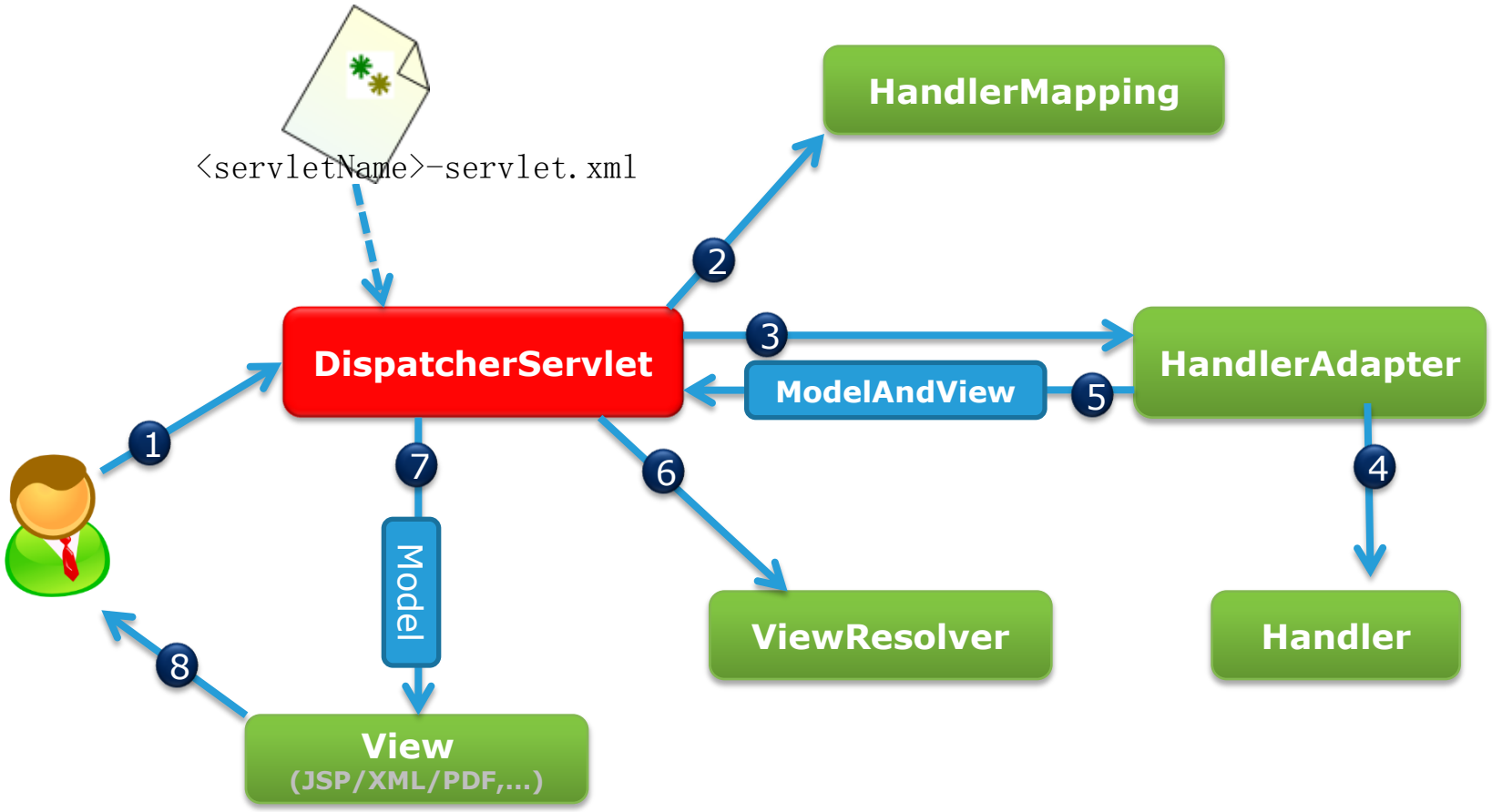
目录

Spring MVC 3.0新特性

- 支持REST风格的URL
- 添加更多注解，可完全注解驱动
- 引入HTTP输入输出转换器（HttpMessageConverter）
- 和数据转换、格式化、验证框架无缝集成
- 对静态资源处理提供特殊支持
- 更加灵活的控制器方法签名，可完全独立于Servlet API

参考《Spring 3.x企业应用开发实战》

Spring MVC框架结构



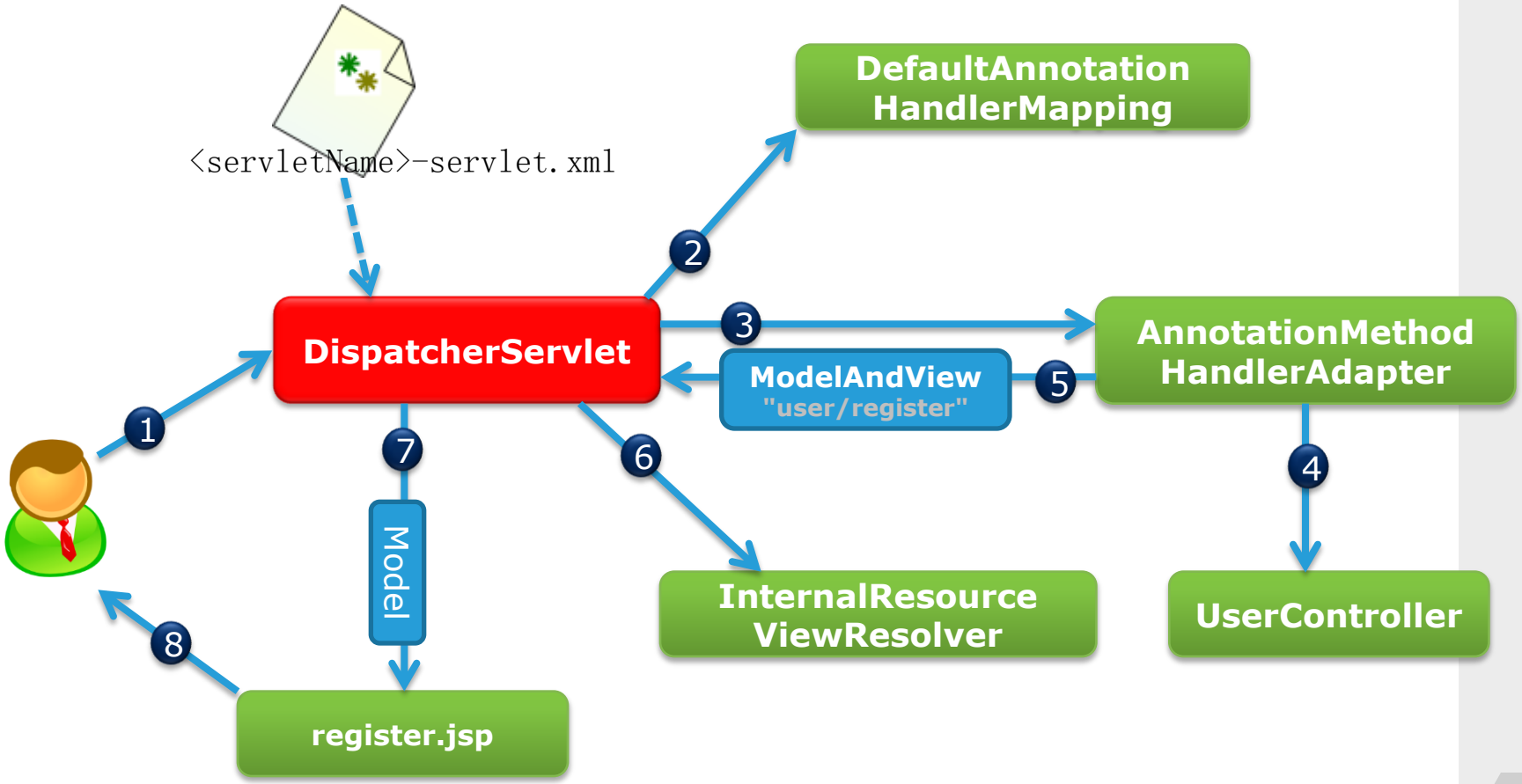
参考《Spring 3.x企业应用开发实战》

Spring MVC框架结构

```
package com.baobaotao.web;
...
@Controller                          ①← 将UserController变成一个Handler
@RequestMapping("/user")             ②←指定控制器映射的URL
public class UserController {

        @RequestMapping(value = "/register") ③←处理方法对应的URL，相对于
                                              ②处的URL
        public String register() {
                return "user/register"; ④←返回逻辑视图名
        }
}
```

参考《Spring 3.x企业应用开发实战》

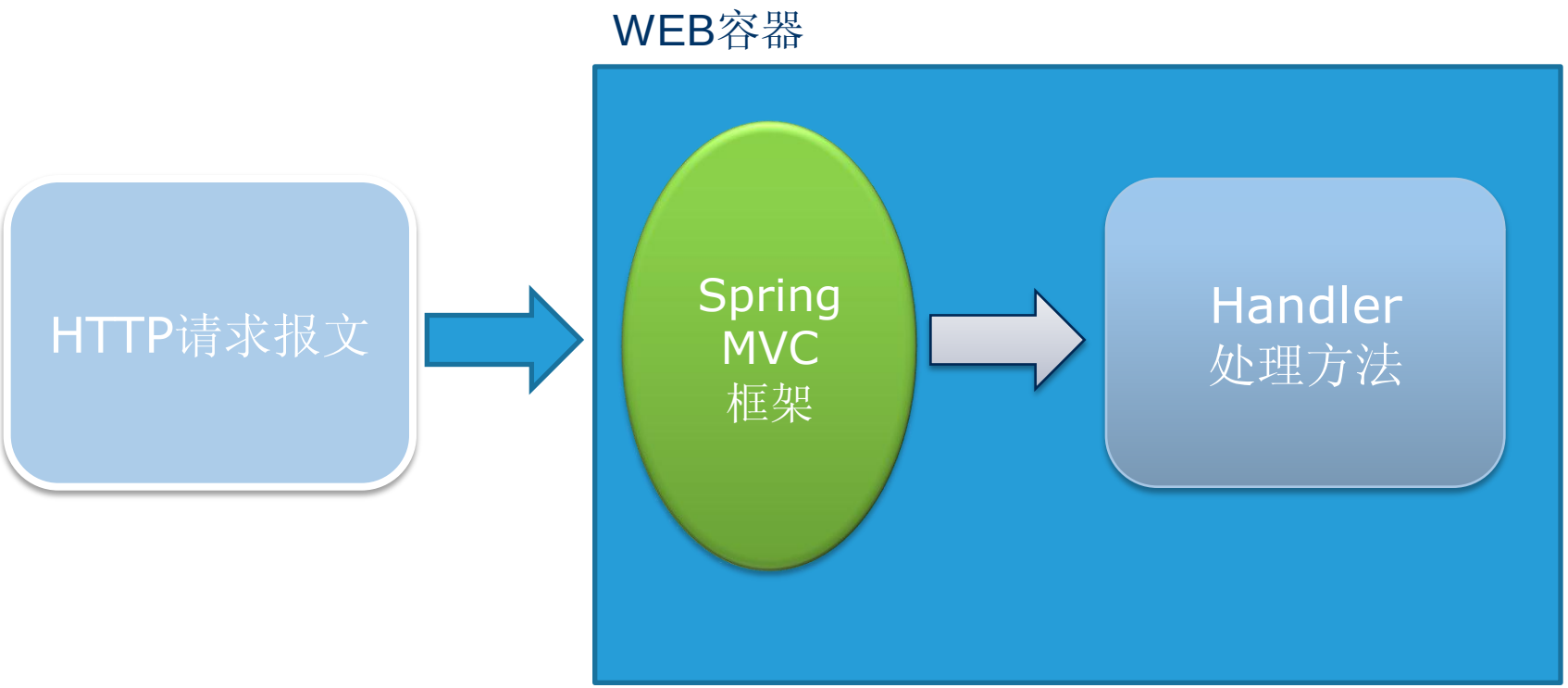
框架的实现者



参考《Spring 3.x企业应用开发实战》

目录

HTTP请求映射原理



参考《Spring 3.x企业应用开发实战》

Spring MVC进行映射的依据

参考《Spring 3.x企业应用开发实战》

通过URL限定:URL表达式

@RequestMapping不但支持标准的URL，还支持Ant风格（即?、*和**的字符，参见3.3.2节的内容）的和带{xxx}占位符的URL。以下URL都是合法的：

- /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/createUseraa、/user/createUserbb等URL。

- /user/{userId}
  匹配user/123、user/abc等URL。

- /user/**/{userId}
  匹配user/aaa/bbb/123、user/aaa/456等URL。

- company/{companyId}/user/{userId}/detail
  匹配company/123/user/456/detail等的URL。

参考《Spring 3.x企业应用开发实战》

通过URL限定:绑定{xxx}中的值

```
@RequestMapping("/{userId}")
public ModelAndView showDetail(@PathVariable("userId") String userId){
        ModelAndView mav = new ModelAnd
        mav.setViewName("user/showDe
        mav.addObject("user", userServi
        return mav;
}
```

URL中的{xxx}占位符可以通过@PathVariable("xxx")绑定到操作方法的入参中。

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

  @RequestMapping("/pets/{petId}")
  public void findPet(@PathVariable String ownerId,
                      @PathVariable String petId, Model model) {
    …
  }
}
```

如果@PathVariable不指定参数名，只有在编译时打开debug开关（javac -debug=no）时才可行！！（不建议）

参考《Spring 3.x企业应用开发实战》

通过请求方法限定:请求方法

请求方法，在HTTP中这被叫做动词（verb），除了两个大家熟知的（GET和POST）之外，标准方法集合中还包含PUT、DELETE、HEAD和OPTIONS。这些方法的含义连同行为许诺都一起定义在HTTP规范之中。一般浏览器只支持GET和POST方法。

序号	请求方法	说明
1	GET	使用GET方法检索一个表述（representation）——也就是对资源的描述。多次执行同一GET请求，不会对系统造成影响，GET方法具有幂等性[指多个相同请求返回相同的结果]。GET请求可以充分使用客户端的缓存。
2	POST	POST方法，通常表示“创建一个新资源”，但它既不安全也不具有幂等性（多次操作会产生多个新资源）。
3	DELETE	DELETE，表示删除一个资源，你也可以一遍又一遍地操作它，直到得出结果：删除不存在的东西没有任何问题
4	PUT	幂等性同样适用于PUT（基本的含义是“更新资源数据，如果资源不存在的话，则根据此URI创建一个新的资源”）

参考《Spring 3.x企业应用开发实战》

通过请求方法限定:代码示例

示例1：

```
@RequestMapping(value=“/delete”)
public String test1(@RequestParam("userId") String userId){
    return "user/test1";
}
```

→所有URL为<*controllerURI*>/delete的请求由test1处理(任何请求方法)

示例2：

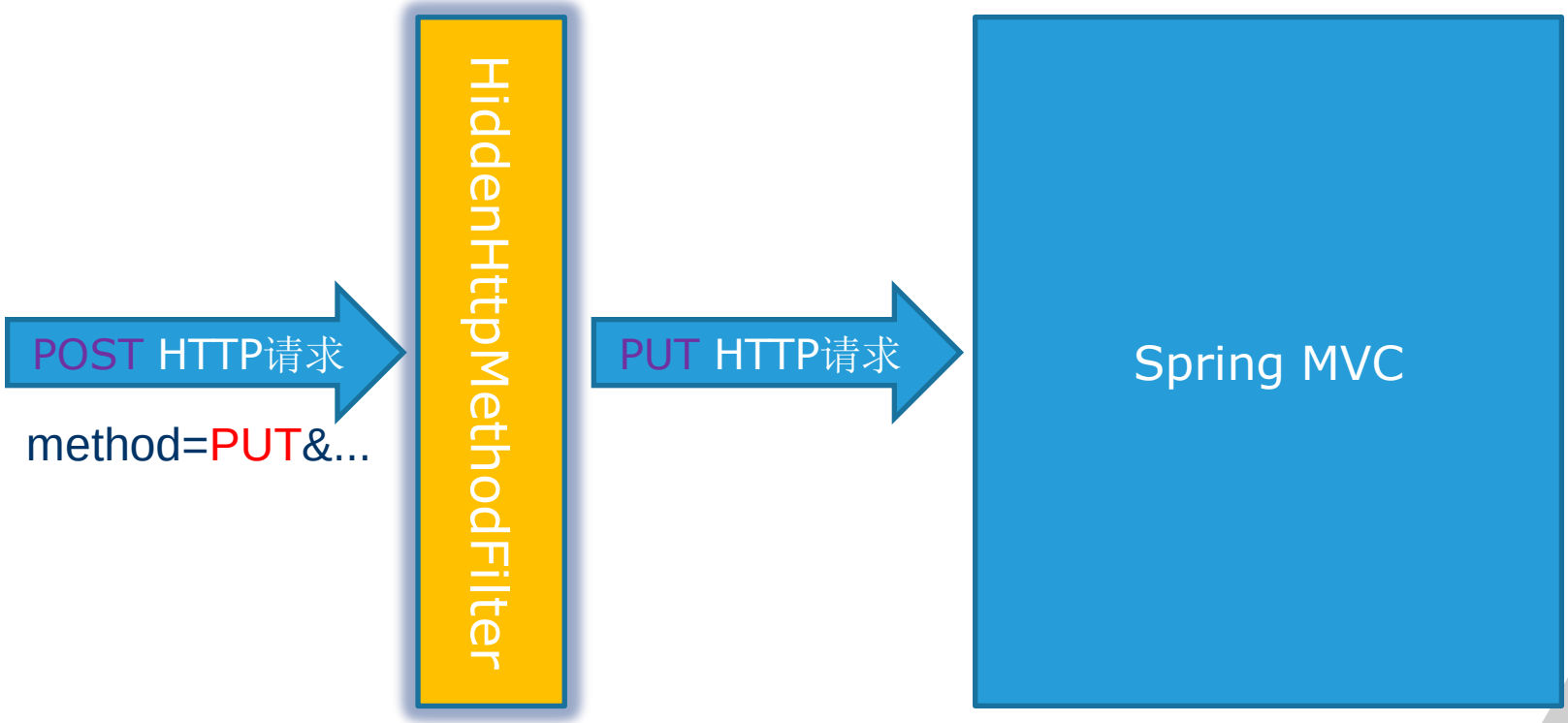
```
@RequestMapping(value="/delete",method=RequestMethod.POST)
public String test1(@RequestParam("userId") String userId){
    return "user/test1";
}
```

→所有URL为<*controllerURI*>/delete **且请求方法为POST** 的请求由test1处理

参考《Spring 3.x企业应用开发实战》

通过请求方法限定:模拟请求方法

通过在web.xml中配置一个
org.springframework.web.filter.HiddenHttpMethodFilter
通过POST请求的_method参数指定请求方法，HiddenHttpMethodFilter
动态更改HTTP头信息。



参考《Spring 3.x企业应用开发实战》

通过请求/请求头参数限定:示例

通过请求参数限定：

```
@RequestMapping(value="/delete", params="userId")
public String test1(@RequestParam("userId") String userId){
    ...
}
```

通过请求头参数限定：

```
@RequestMapping(value="/show",headers="content-type=text/*")②
public String test2(@RequestParam("userId") String userId){
    ...
}
```

参考《Spring 3.x企业应用开发实战》

通过请求/请求头参数限定:更多

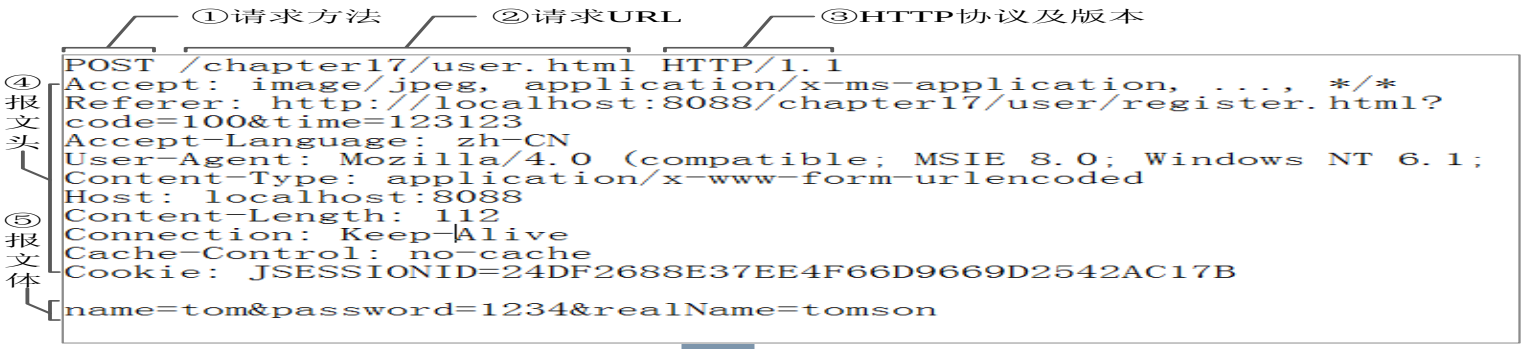
params和headers分别通过请求参数及报文头属性进行映射，它们支持简单的表达式，下面以params表达式为例说明，headers可以参照params进行理解之。

- **"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。

参考《Spring 3.x企业应用开发实战》

目录

通过注解绑定:示意图



@RequestParam→绑定请求参数

@RequestHeader→绑定请求头参数

@CookieValue→绑定Cookie的值

@PathVariable→绑定URL中的变量

public String handle1(...)

参考《Spring 3.x企业应用开发实战》

通过注解绑定:示例

```
@RequestMapping(value="/handle1")
public String handle1(@RequestParam("userName") String userName,
                      @RequestParam("password") String password,
                      @RequestParam("realName") String realName){
        ...
}
```

```
@RequestMapping(value="/handle2")
public String handle2(@CookieValue("JSESSIONID") String sessionId,
    @RequestHeader("Accept-Language") String accpetLanguage){
  ...
}
```

参考《Spring 3.x企业应用开发实战》

通过注解绑定:小心抛出异常

@RequestParam有以下三个参数。

- value：参数名。
- required：是否必需，默认为true，表示请求中必须包含对应的参数名，如果不存在将抛出异常。
- defaultValue：默认参数名，设置该参数时，自动将required设为false。极少情况需要使用该参数，也不推荐使用该参数。

```
@RequestMapping(value="/handle1")
public String handle1(@RequestParam("userName") String userName,){
        ...
}
```

上面的处理方法，如果HTTP请求不包含“userName”参数时，将产生异常！！

因此，如果不能保证存在”userName”的参数，必须使用：

@RequestParam(value = "userName", **required = false**)

参考《Spring 3.x企业应用开发实战》

使用命令/表单对象绑定

所谓命令/表单对象并不需要实现任何接口，仅是一个拥有若干属性的POJO。Spring MVC按：

“HTTP请求参数名 = 命令/表单对象的属性名”

的规则，自动绑定请求数据，支持“级联属性名”，自动进行基本类型数据转换。

```
@RequestMapping(value = "/handle14")
public String handle14(User user) {
        …
}
```

userName=xxx&password=yyy

```
class User{
    private String userName;
    private String password;
}
```

参考《Spring 3.x企业应用开发实战》

使用Servlet API对象作为入参

在Spring MVC中，控制器类可以不依赖任何Servlet API对象，但是Spring MVC并不阻止我们使用Servlet API的类作为处理方法的入参。值得注意的是，*如果处理方法自行使用HttpServletResponse返回响应，则处理方法的返回值设置成void即可*。

```
@RequestMapping(value = "/handle21")
public void handle21(HttpServletRequest request,HttpServletResponse response) {
                        String userName = WebUtils.findParameterValue(request, "userName");
                        response.addCookie(new Cookie("userName", userName));
}
```

```
public String handle23(HttpSession session) {
                        session.setAttribute("sessionId", 1234);
                        return "success";
}
```

```
public String handle24(HttpServletRequest request,
                        @RequestParam("userName")String userName) {
                        …
                        return "success";
}
```

使用Spring的Servlet API代理类

Spring MVC在org.springframework.web.context.request包中定义了若干个可代理Servlet原生API类的接口，如WebRequest和NativeWebRequest，它们也允许作为处理类的入参，通过这些代理类可访问请求对象的任何信息。

```
@RequestMapping(value = "/handle25")
public String handle25(WebRequest request) {
        String userName = request.getParameter("userName");
        return "success";
}
```

参考《Spring 3.x企业应用开发实战》

使用IO对象作为入参

Spring MVC允许控制器的处理方法使用java.io.InputStream/java.io.Reader及java.io.OutputStream/java.io.Writer作为方法的入参

```
@RequestMapping(value = "/handle31")
public void handle31(OutputStream os) throws IOException{
    Resource res = new ClassPathResource("/image.jpg");//读取类路径下的图片文件
    FileCopyUtils.copy(res.getInputStream(), os);//将图片写到输出流中
}
```

Spring MVC将获取ServletRequest的InputStream/Reader或ServletResponse的OutputStream/Writer，然后按类型匹配的方式，传递给控制器的处理方法入参。

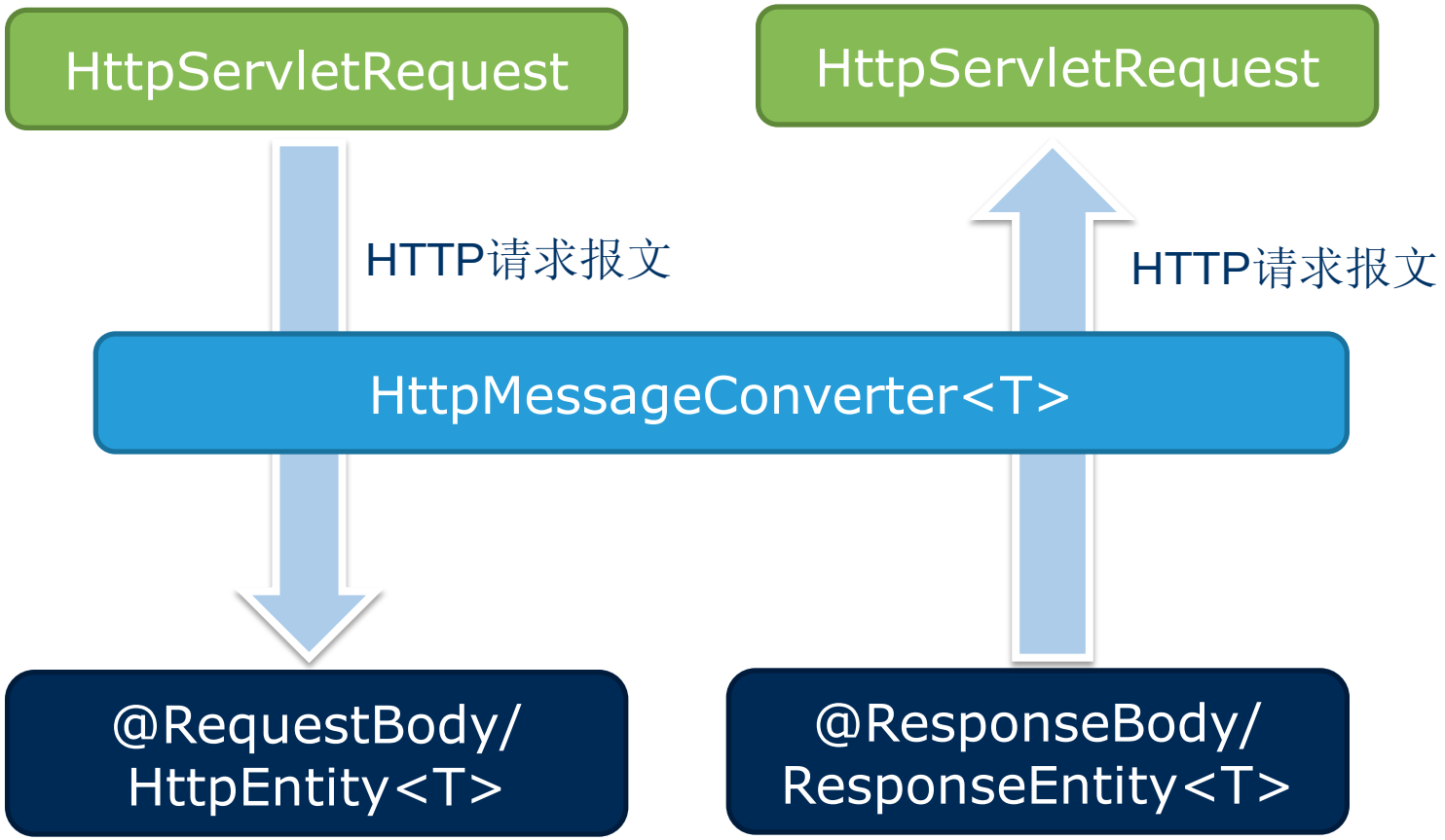
参考《Spring 3.x企业应用开发实战》

其他类型的参数

控制器处理方法的入参除支持以上类型的参数以外，还支持java.util.Locale、java.security.Principal，可以通过Servlet的HttpServletRequest 的getLocale()及getUserPrincipal()得到相应的值。如果处理方法的入参类型为Locale或Principal，Spring MVC自动从请求对象中获取相应的对象并传递给处理方法的入参。

```
@RequestMapping(value = "/handle32")
public void handle31(Locale locale) throws IOException{
    ...
}
```

HttpMessageConverter<T>



HttpMessageConverter<T>实现类

AnnotationMethodHandlerAdapter

注册到...

HttpMessageConverter<T>
接口方法→
T read(HttpInputMessage httpInputMessage)
void write(T t,HttpOutputMessage httpOutputMessage)

实现类：
StringHttpMessageConverter
FormHttpMessageConverter
XmlAwareFormHttpMessageConverter
ResourceHttpMessageConverter
BufferedImageHttpMessageConverter
ByteArrayHttpMessageConverter
SourceHttpMessageConverter
MarshallingHttpMessageConverter
Jaxb2RootElementHttpMessageConverter
MappingJacksonHttpMessageConverter
RssChannelHttpMessageConverter
AtomFeedHttpMessageConverter

使用@RequestBody/@ResponseBody

将HttpServletRequest的getInputStream()内容绑定到入参，将处理方法返回值写入到HttpServletResponse的getOutputStream()中。

```
@RequestMapping(value = "/handle41")
public String handle41(@RequestBody  String requestBody ) {
    System.out.println(requestBody);
    return "success";
}
```

```
@ResponseBody
@RequestMapping(value = "/handle42/{imageId}")
public byte[] handle42(@PathVariable("imageId") String imageId) throws IOException {
    System.out.println("load image of "+imageId);
    Resource res = new ClassPathResource("/image.jpg");
    byte[] fileData =FileCopyUtils.copyToByteArray(res.getInputStream());
    return fileData;
}
```

优点：处理方法签名灵活不受限
缺点：只能访问报文体，不能访问报文头

参考《Spring 3.x企业应用开发实战》

使用HttpEntity<T>/ResponseEntity<T>

```
@RequestMapping(value = "/handle43")
public String handle43(HttpEntity<String> httpEntity){
     long contentLen = httpEntity.getHeaders().getContentLength();
     System.out.println(httpEntity.getBody());
      return "success";
}
```

```
@RequestMapping(params = "method=login")
public ResponseEntity<String> doFirst(){
    HttpHeaders headers = new HttpHeaders();
    MediaType mt=new MediaType("text","html",Charset.forName("UTF-8"));
    headers.setContentType(mt);
    ResponseEntity<String> re=null;
    String return = new String("test");
    re=new ResponseEntity<String>(return,headers, HttpStatus.OK);
    return re;
}
```

优点：处理方法签名受限
缺点：不但可以访问报文体，还能访问报文头

参考《Spring 3.x企业应用开发实战》

输出XML和JSON

```
<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"
      p:messageConverters-ref="messageConverters"/>
<util:list id="messageConverters">
    <bean class="org.springframework.http.converter.BufferedImageHttpMessageConverter" />
    <bean class="org.springframework.http.converter.ByteArrayHttpMessageConverter" />
    <bean class="org.springframework.http.converter.StringHttpMessageConverter" />
    <bean class="org.springframework.http.converter.xml
                                    .XmlAwareFormHttpMessageConverter" />
    <bean class="org.springframework.http.converter.xml
                                    .MarshallingHttpMessageConverter"
          p:marshaller-ref="xmlMarshaller"
          p:unmarshaller-ref="xmlMarshaller">
    </bean>
    <bean class="org.springframework.http.converter.json
                                    .MappingJacksonHttpMessageConverter"/>
</util:list>

<!--①声明Marshaller，使用XStream技术-->
<bean id="xmlMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="streamDriver">
        <bean class="com.thoughtworks.xstream.io.xml.StaxDriver"/>②
    </property>
    <property name="annotatedClasses">
        <list>
            <value>com.baobaotao.domain.User</value>  ②
        </list>
    </property>
</bean>
…
```

处理XML转换

处理JSON转换

使用 STAX 对 XML 消息进行处理，STAX 占用内存少，速度也很快

我们将使用 XStream 的注解定义 XML 转换规则，使用到 XStream 注解的类在此声明

参考《Spring 3.x企业应用开发实战》

使用HttpEntity<T>/ResponseEntity<T>

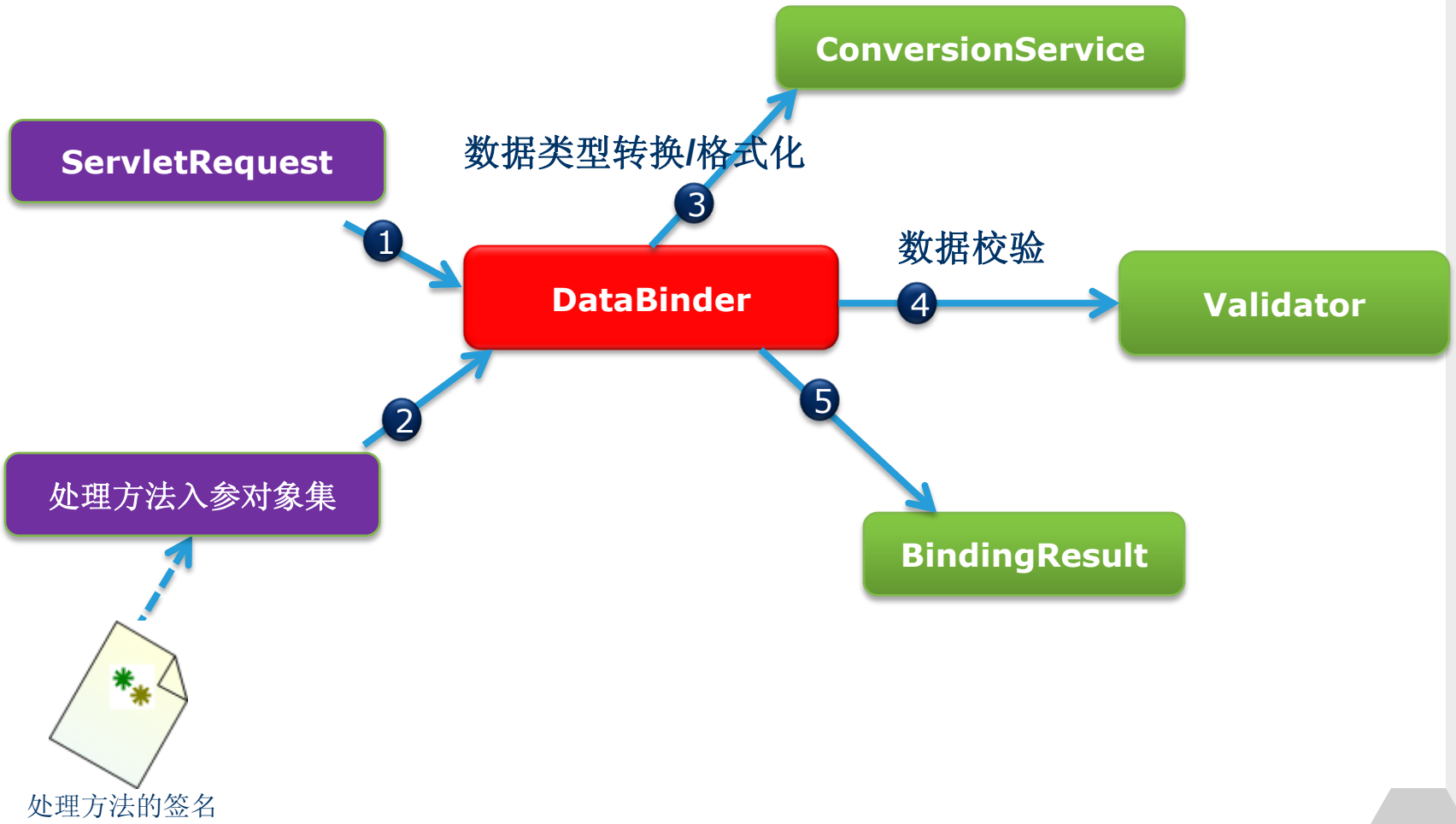
```
@RequestMapping(value = "/handle51")
public ResponseEntity<User>  handle51(HttpEntity<User> requestEntity){
    User user = requestEntity.getBody();
    user.setUserId("1000");
    return new ResponseEntity<User>(user,HttpStatus.OK);
}
```

对于服务端的处理方法而言，除使用@RequestBody/@ResponseBody或HttpEntity<T> /ResponseEntity<T>进行方法签名外，不需要进行任何额外的处理，借由Spring MVC中装配的HttpMessageConverter，它即拥有了处理XML及JSON的能力了。

参考《Spring 3.x企业应用开发实战》

目录

数据绑定机理


ConversionService
ServletRequest
数据类型转换/格式化
3
1
DataBinder
数据校验
4
Validator
2
5
处理方法入参对象集
BindingResult
处理方法的签名

数据类型转换

低版本的Spring 只支持标准的PropertyEditor类型体系，不过PropertyEditor存在以下缺陷：

- 只能用于字符串和Java对象的转换，不适用于任意两个Java类型之间的转换；
- 对源对象及目标对象所在的上下文信息（如注解、所在宿主类的结构等）不敏感，在类型转换时不能利用这些上下文信息实施高级转换逻辑。

有鉴于此，Spring 3.0在核心模型中添加了一个通用的类型转换模块，ConversionService是Spring类型转换体系的核心接口。

Spring 3.0同时支持PropertyEditor和ConversionService 进行类型转换，在Bean配置、Spring MVC处理方法入参绑定中使用类型转换体系进行工作。

参考《Spring 3.x企业应用开发实战》

PropertyEditor依然有效

对于简单的类型转换，依然建议使用PropertyEditor。按照PropertyEditor的协议，会自动查找Bean类相同类包是否存在<BeanName>Editor.class，如果存在会使用它作为Bean的编辑器。

```
com.book.core.cache.expired
  |_CacheSpace.java
  |_ CacheSpaceEditor.java
```

```
<bean id="expireManager"
        class="com.book.core.cache.expire.SimpleCacheLogicExpireManager">
    <property name="cacheSpaceList">
        <list>
            <value>comBookSpace:com/comBook/**</value>
            <value>bookSpace:com/book/**:100</value>
            <value>companySpace:com/company/**</value>
        </list>
    </property>
</bean>
```

参考《Spring 3.x企业应用开发实战》

强大的ConversionService，让很多梦想成真

由于ConversionService在进行类型转换时，可以使用到Bean所在宿主类的上下文信息（包括类结构，注解信息），所以可以实施更加高级的类型转换，如注解驱动的格式化等功能。

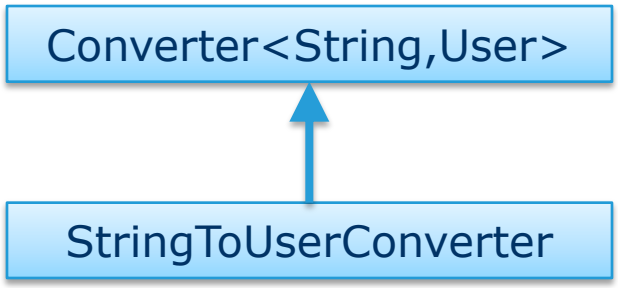
```
public class User {
    @DateTimeFormat(pattern="yyyy-MM-dd")
    private Date birthday;
}
```

以上User类，通过一个@DateTimeFormat注解，为类型转换提供了一些“额外”的信息，即代表日期的“源字符器”格式是“yyyy-MM-dd”

参考《Spring 3.x企业应用开发实战》

基于ConversionService体系，定义自定义的类型转换器

定义自定义转换器：



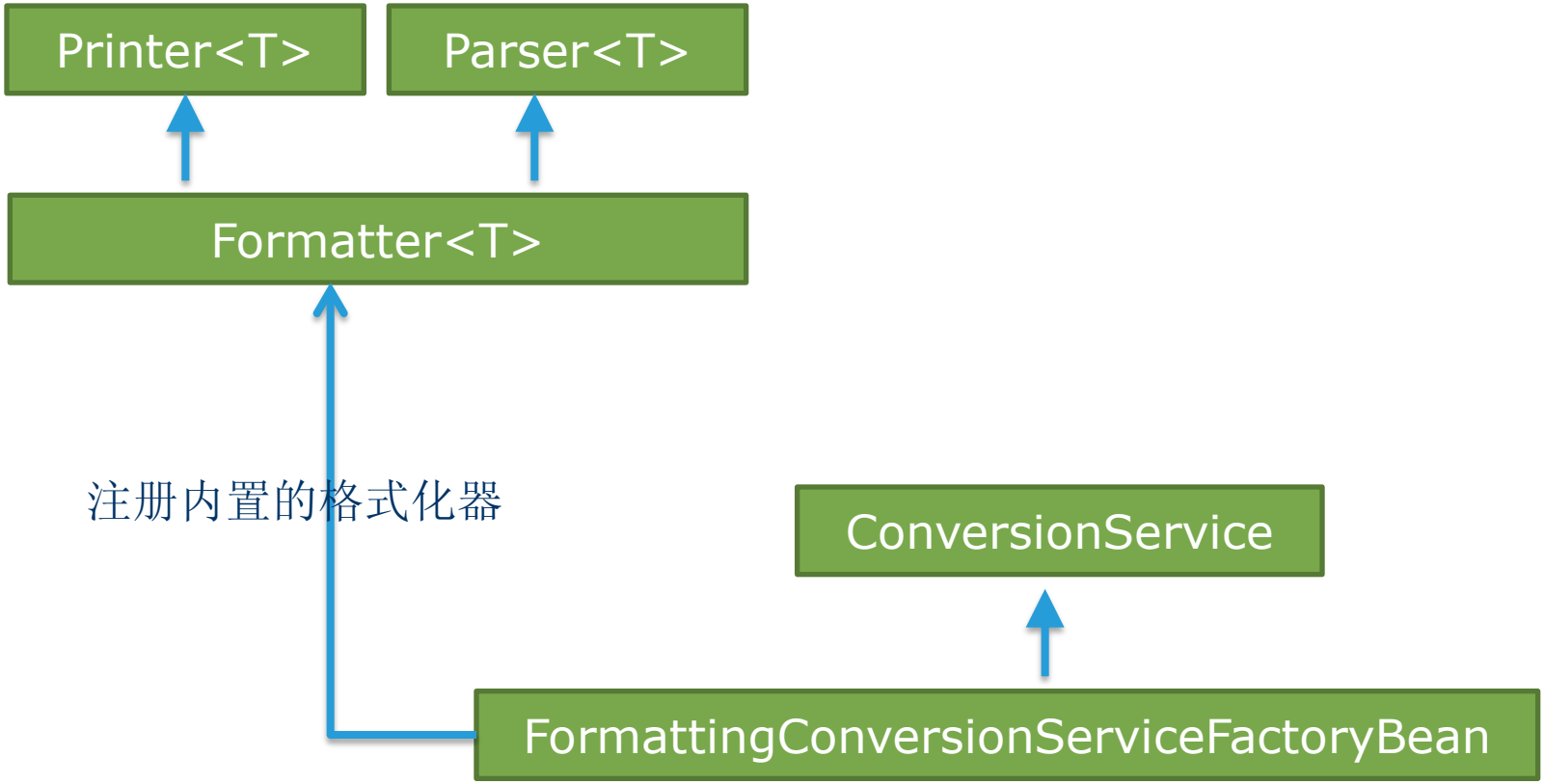
org.springframework.core.convert.converter.Converter

注册自定义转换器：

```
<mvc:annotation-driven conversion-service="conversionService"/>
<bean id="conversionService"
    class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
      <list>
          <bean class="com.baobaotao.domain.StringToUserConverter"/>
      </list>
    </property>
</bean>
```

参考《Spring 3.x企业应用开发实战》

格式化：带格式字符串←→内部对象 相互转换



使用支持格式化的转换器

```
<mvc:annotation-driven conversion-service="conversionService"/>
<bean id="conversionService"
    class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
    <property name="converters">
      <list>
         <bean class="com.baobaotao.domain.StringToUserConverter"/>
       </list>
    </property>
</bean>
```

值得注解的是，<mvc:annotation-driven/>标签内部默认创建的ConversionService实例就是一个FormattingConversionServiceFactoryBean，自动支持如下的格式化注解：

- @NumberFormatter：用于数字类型对象的格式化。
- @CurrencyFormatter：用于货币类型对象的格式化。
- @PercentFormatter：用于百分数数字类型对象的格式化。

>>参见testhandle82()

参考《Spring 3.x企业应用开发实战》

数据校验框架

Spring 3.0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 的DataBinder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，则可直接通过注解驱动的方式进行数据校验。

Spring的org.springframework.validation是校验框架所在的包

参考《Spring 3.x企业应用开发实战》

JSR 303

JSR 303是Java为Bean数据合法性校验所提供的标准框架，它已经包含在Java EE 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。

你可以通过http://jcp.org/en/jsr/detail?id=303了解JSR 303的详细内容。

注　解	功能说明
@Null	被注释的元素必须为 **null**
@NotNull	被注释的元素必须不为 **null**
@AssertTrue	被注释的元素必须为 **true**
@AssertFalse	被注释的元素必须为 **false**
@Min(value)	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
@Max(value)	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
@DecimalMin(value)	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
@DecimalMax(value)	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
@Size(max, min)	被注释的元素的大小必须在指定的范围内
@Digits (integer, fraction)	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
@Past	被注释的元素必须是一个过去的日期
@Future	被注释的元素必须是一个将来的日期

数据校验框架

<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。

```
public class User {
    @Pattern(regexp="w{4,30}")
    private String userName;

    @Length(min=2,max=100)
    private String realName;

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

  @DecimalMin(value="1000.00")
  @DecimalMax(value="100000.00")
  @NumberFormat(pattern="#,###.##")
  private long salary;
}
```

注意：Spring本身没有提供JSR 303的实现，所以必须将JSR 303的实现者（如Hibernate Validator）的jar文件放到类路径下，Spring将自动加载并装配好JSR 303的实现者。

参考《Spring 3.x企业应用开发实战》

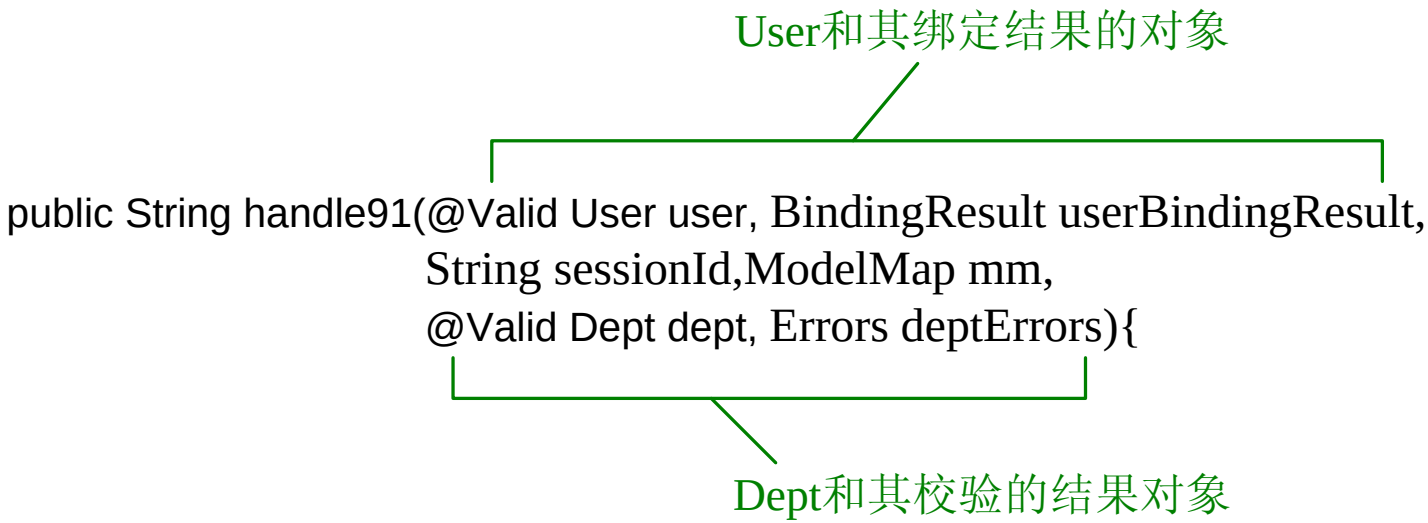
如何使用注解驱动的校验

```
@Controller
@RequestMapping("/user")
public class UserController {
    @RequestMapping(value = "/handle91")
    public String handle91(@Valid  User user,
                           BindingResult bindingResult){
        if(bindingResult.hasErrors()){
            return "/user/register3";
        }else{
            return "/user/showUser";
        }
    }
}
```

在已经标注了JSR 303注解的表单/命令对象前标注一个@Valid，Spring MVC框架在将请求数据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。

参考《Spring 3.x企业应用开发实战》

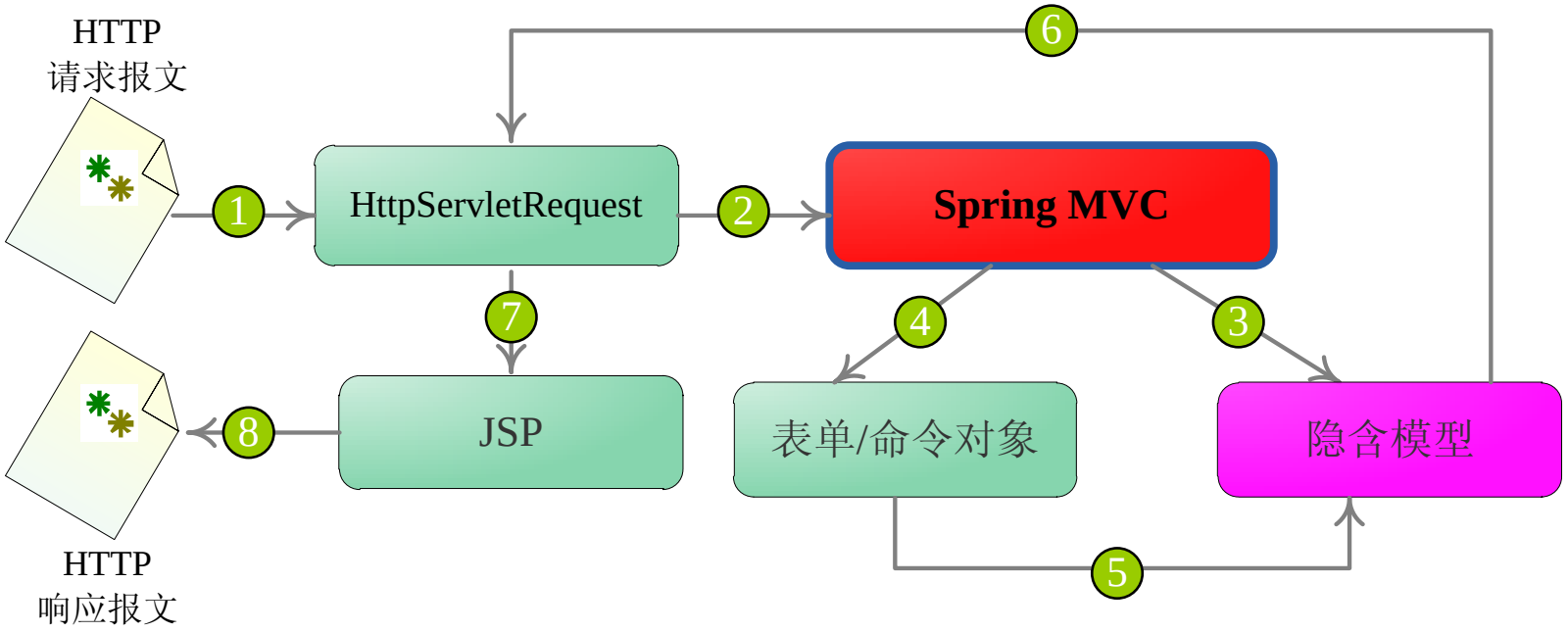
使用校验功能时，处理方法要如何签名？？



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在其后的入参中，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。

参考《Spring 3.x企业应用开发实战》

校验错误信息存放在什么地方？？



- 4.Spring MVC将HttpServletRequest对象数据绑定到处理方法的入参对象中（表单/命令对象）；
- 5.将绑定错误信息、检验错误信息都保存到隐含模型中；
- 6.本次请求的对应隐含模型数据存放到HttpServletRequest的属性列表中，暴露给视图对象。

参考《Spring 3.x企业应用开发实战》

页面如何显示错误信息

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c"      uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form"   uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<title>注册用户</title>
  <style>.errorClass{color:red}</style>
</head>
<body>
  <form:form modelAttribute="user"  action="user/handle91.html">
     <form:errors path="*"/>
     <table>
                  <tr>
                    <td>用户名：</td>
                    <td>
                      <form:errors path="userName" cssClass="errorClass"/>
                      <form:input path="userName" />
                    </td>
                  </tr>
          …
    </table>
  </form:form>
</body>
</html>
```

参考《Spring 3.x企业应用开发实战》

如何对错误信息进行国际化(1)

一个属性发生校验错误时，Spring MVC会产生一系列对应的错误码键：

```
public class User {
    @Pattern(regexp="w{4,30}")→假设发生错误
    private String userName;
}
```

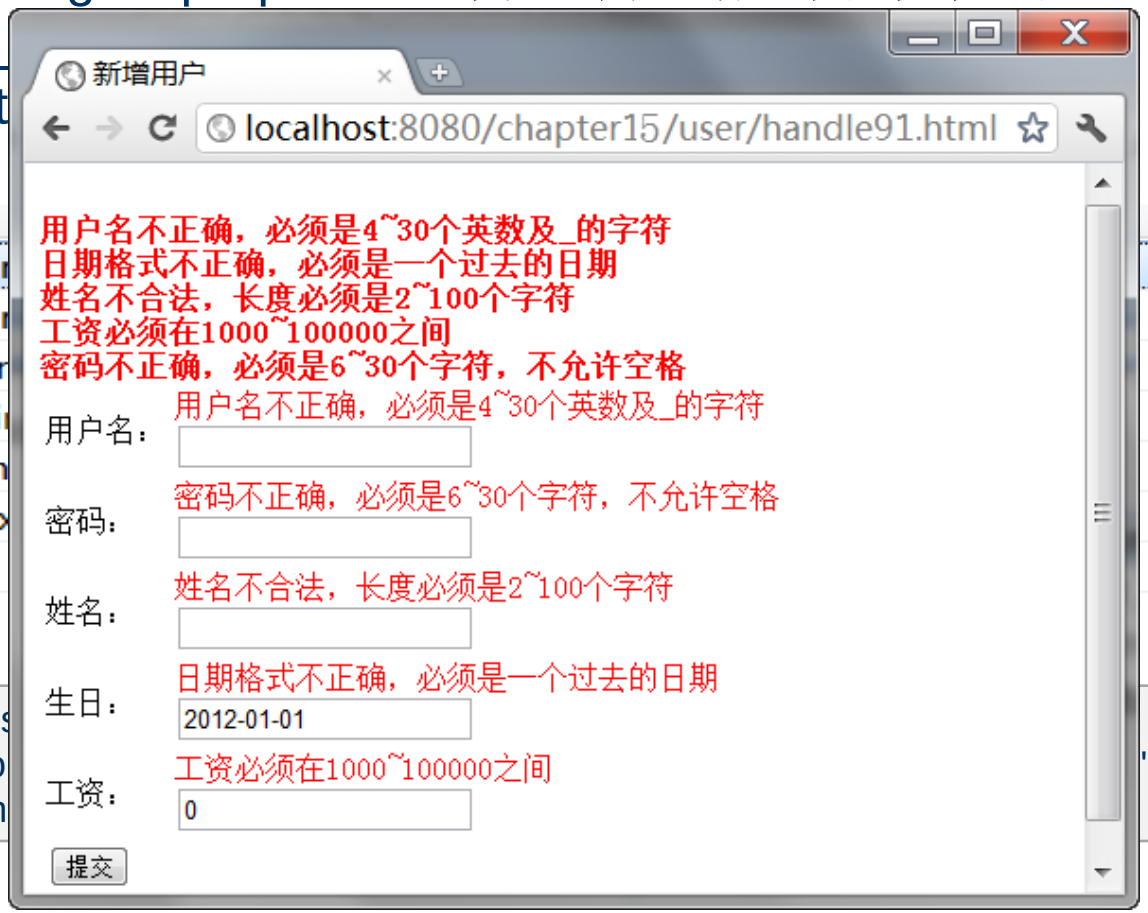
如果userName的@Pattern校验规则未通过，则会在“隐含模型”中产生如下的错误键，这些错误键可以作为“国际化消息”的属性键。

- Pattern.user.userName
- Pattern.userName
- Pattern.String
- Pattern

参考《Spring 3.x企业应用开发实战》

如何对错误信息进行国际化(2)

我们在conf/i18n/下添加基名为messages的国际化资源，一个是默认的messages. properties，另一个是对应中国大陆的messages_
CN.propert



name
Pattern.user
Pattern.user
Length.user
Past.user.bi
DecimalMin
DecimalMax

<bean id="mess
class="o
p:basen

参考《Spring 3.x企业应用开发实战》

目录

数据模型访问结构

接收请求 → 处理请求 → 请求响应

@ModelAttribute("user")

@SessionAttributes

ModelAndView，Map及Model

数据模型

key1=value1

key2=value2

...

暴露给...

视图对象

参考《Spring 3.x企业应用开发实战》

访问数据模型:ModelAndView

通过ModelAndView

```
@RequestMapping(method = RequestMethod.POST)
public ModelAndView createUser(User user) {
        userService.createUser(user);
        ModelAndView mav = new ModelAndView();
        mav.setViewName("user/createSuccess");
        mav.addObject("user", user);
        return mav;
}
```

参考《Spring 3.x企业应用开发实战》

访问数据模型：@ModelAttribute

1.使用方式一

```
@RequestMapping(value = "/handle61")
public String  handle61(@ModelAttribute("user") User user){
                user.setUserId("1000");
                return "/user/createSuccess";
}
```

→ Spring MVC将HTTP请求数据绑定到user入参中，然后再将user对象添加到数据模型中。

2.使用方式二

```
@ModelAttribute("user")
public User getUser(){
                User user = new User();
                user.setUserId("1001");
                return user;
}

@RequestMapping(value = "/handle62")
public String  handle62(@ModelAttribute("user") User user){
                user.setUserName("tom");
                return "/user/showUser";
}
```

访问UserController中任何一个请求处理方法前，Spring MVC先执行该方法，并将返回值以user为键添加到模型中

在此，模型数据会赋给User的入参，然后再根据HTTP请求消息进一步填充覆盖user对象

参考《Spring 3.x企业应用开发实战》

访问数据模型：Map及Model

org.springframework.ui.Model和java.util.Map:

```
@RequestMapping(value = "/handle63")
public String  handle63(ModelMap modelMap){
    modelMap.addAttribute("testAttr","value1");
    User user = (User)modelMap.get("user");
    user.setUserName("tom");
    return "/user/showUser";
}
```

Spring MVC一旦发现处理方法有Map或Model类型的入参，就会将请求内在的隐含模型对象的引用传给这些入参。

参考《Spring 3.x企业应用开发实战》

访问数据模型：@SessionAttributes

如果希望在多个请求之间共用某个模型属性数据，则可以在控制器类标注一个@SessionAttributes，Spring MVC会将模型中对应的属性暂存到HttpSession中：

```
@Controller
@RequestMapping("/user")
@SessionAttributes(“user”)①
public class UserController {

        @RequestMapping(value = "/handle71")
        public String  handle71(@ModelAttribute(“user”) User user){②
                user.setUserName("John");
                return "redirect:/user/handle72.html";
        }

        @RequestMapping(value = "/handle72")
        public String  handle72(ModelMap modelMap,SessionStatus sessionStatus){
                User user = (User)modelMap.get(“user”);③
                if(user != null){
                        user.setUserName("Jetty");
                        sessionStatus.setComplete();④
                }
                return "/user/showUser";
        }
}
```

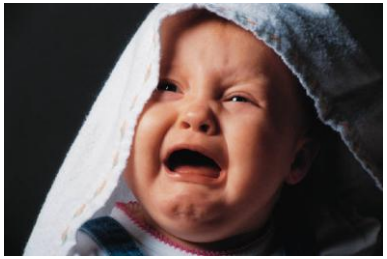
将②处的模型属性自动保存到HttpSession中

读取模型中的数据

让Spring MVC清除本处理器对应的会话属性

参考《Spring 3.x企业应用开发实战》

一场由@SessionAttributes引发的血案...



→

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

对入参标注@ModelAttribute(“xxx”)的处理方法，Spring MVC按如下流程处理（handle71(@ModelAttribute(“user”) User user)）：

1. 如果隐含模型拥有名为xxx的属性，将其赋给该入参，再用请求消息填充该入参对象直接返回，否则到2步 。
2. 如果xxx是会话属性，即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**。否则到 3。
3. 如果隐含模型不存在xxx属性，且xxx也不是会话属性，则创建入参的对象实例，再用请求消息填充该入参。

参考《Spring 3.x企业应用开发实战》

如何避免@SessionAttributes引发的血案

原来也是小Cakes一张...

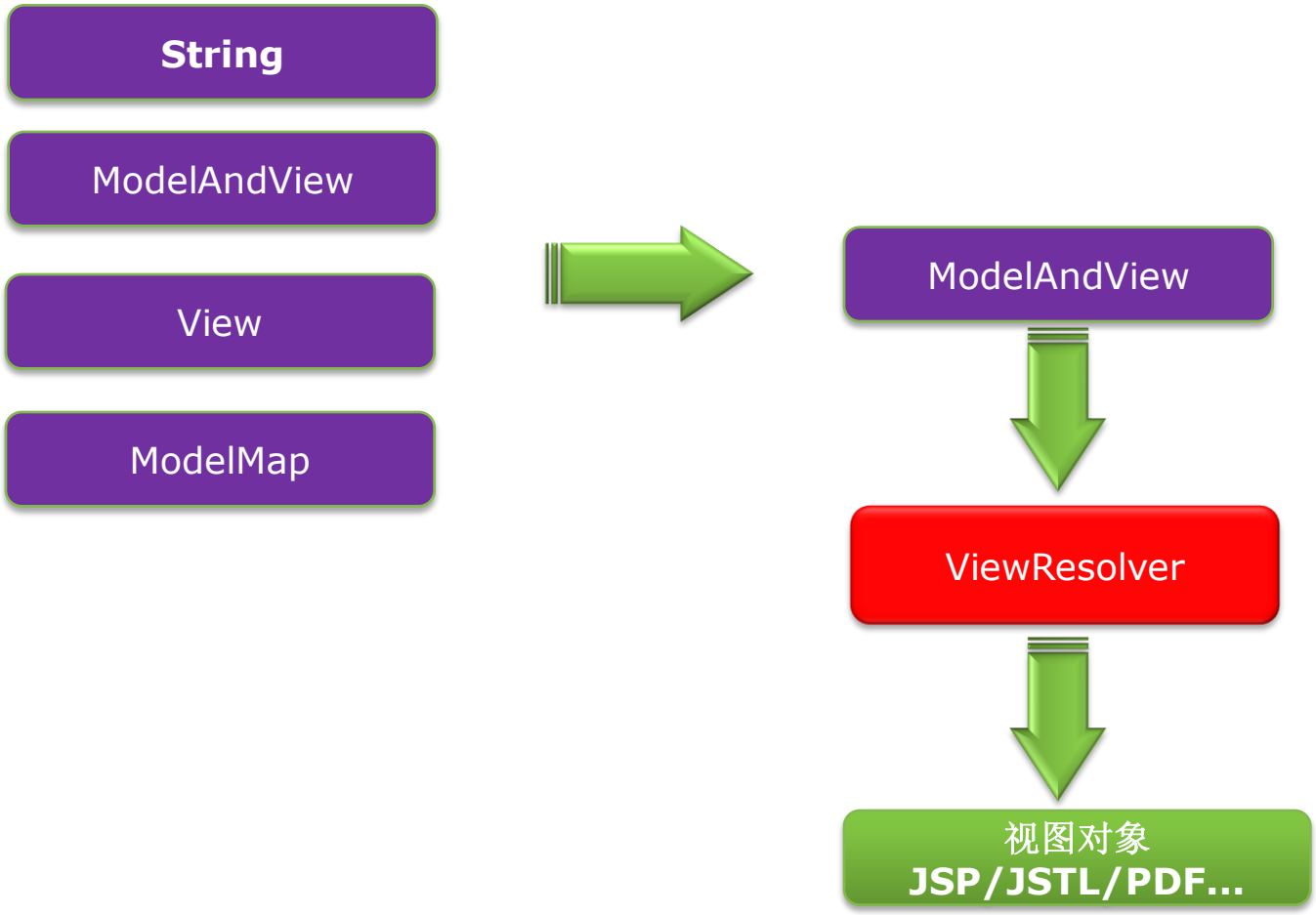
```
@Controller
@RequestMapping("/user")
@SessionAttributes(“user”)
public class UserController {
        @ModelAttribute("user")
        public User getUser(){
                User user = new User();
                return user;
        }

        @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

目录

Spring MVC如何解析视图

请求处理方法返回值类型



参考《Spring 3.x企业应用开发实战》

视图解析器类型

完成单一解析逻辑的视图解析器：

- InternalResourceViewResolver
- FreeMarkerViewResolver
- BeanNameViewResolver
- XmlViewResolver
- ...

基于协商的视图解析器：

- ContentNegotiatingViewResolver

该解析器是Spring 3.0新增的，它不负责具体的视图解析，而是作为一个中间人的角色根据请求所要求的MIME类型，从上下文中选择一个适合的视图解析器，再将视图解析工作委托其负责

基于协商的视图解析器

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver"
            p:order="0" p:defaultContentType="text/html" p:ignoreAcceptHeader="true"
   p:favorPathExtension="false" p:favorParameter="true" p:parameterName="content">
            <property name="mediaTypes">
                        <map>
                              <entry key="html" value="text/html" />
                              <entry key="xml" value="application/xml" />
                               <entry key="json" value="application/json" />
                        </map>
            </property>
            <property name="defaultViews">
              <list>
<bean class="org.springframework.web.servlet.view.json.MappingJacksonJsonView"
                                                p:renderedAttributes="userList" />
<bean class="org.springframework.web.servlet.view.xml.MarshallingView"
                                                p:modelKey="userList" p:marshaller-ref="xmlMarshaller" />
              </list>
            </property>
</bean>
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
            p:order="100" p:viewClass="org.springframework.web.servlet.view.JstlView"
            p:prefix="/WEB-INF/views/" p:suffix=".jsp" />
```

例子：http://localhost:9080/user/showUserListMix

目录

本地化:基础原理

一般情况下，Web应用根据客户端浏览器的设置判断客户端的本地化类型，用户可以通过IE菜单：工具→Internet选项...→语言...在打开的“语言首选项”对话框中选择本地化类型。

浏览器中设置的本地化类型会包含在HTML请求报文头中发送给Web服务器，确切地说是通过报文头的Accept-Language参数将“语言首选项”对话框中选择的语言发送到服务器，成为服务器判别客户端本地化类型的依据。

TcpTrace实例...

参考《Spring 3.x企业应用开发实战》

本地化:Spring MVC的本地化解析器

- AcceptHeaderLocaleResolver：根据HTTP报文头的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，Spring MVC默认采用AcceptHeader- LocaleResolver。
- CookieLocaleResolver：根据指定Cookie值确定本地化类型。
- SessionLocaleResolver：根据Session中特定的属性值确定本地化类型。
- LocaleChangeInterceptor：从请求参数中获取本次请求对应的本地化类型。

参考《Spring 3.x企业应用开发实战》

本地化:Spring MVC的本地化解析器

- AcceptHeaderLocaleResolver：根据HTTP报文头的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，Spring MVC默认采用AcceptHeader- LocaleResolver。
- CookieLocaleResolver：根据指定Cookie值确定本地化类型。
- SessionLocaleResolver：根据Session中特定的属性值确定本地化类型。
- LocaleChangeInterceptor：从请求参数中获取本次请求对应的本地化类型。

参考《Spring 3.x企业应用开发实战》

LocaleChangeInterceptor：通过URL参数指定

很多国际型的网站都允许通过一个请求参数控制网站的本地化，如www.xxx.com? locale=zh_CN返回对应中国大陆的本地化网页，而www.xxx.com?locale=en返回本地化为英语的网页。这样，网站使用者可以通过URL的控制返回不同本地化的页面，非常灵活。

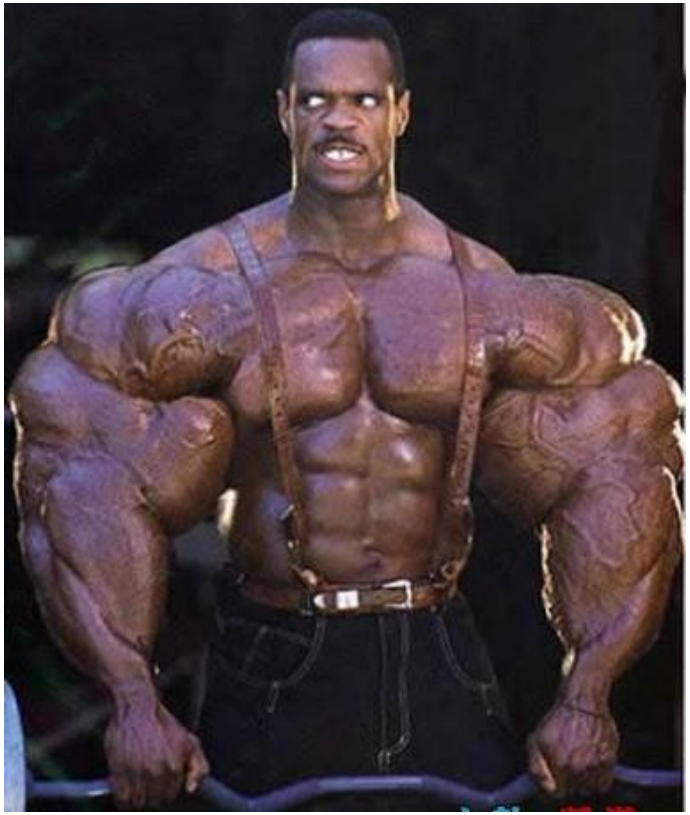
```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver"
                p:cookieName="clientLanguage"
                p:cookieMaxAge="100000"
                p:cookiePath="/"
                p:defaultLocale="zh_CN"/>
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor" />
</mvc:interceptors>
```

例子：

http://localhost:9080/user/handle91?locale=en_US

参考《Spring 3.x企业应用开发实战》

静态资源处理



Spring MVC 3.0提供的最强大的功能之一！！！
1.静态资源处理方式
2.静态资源映射

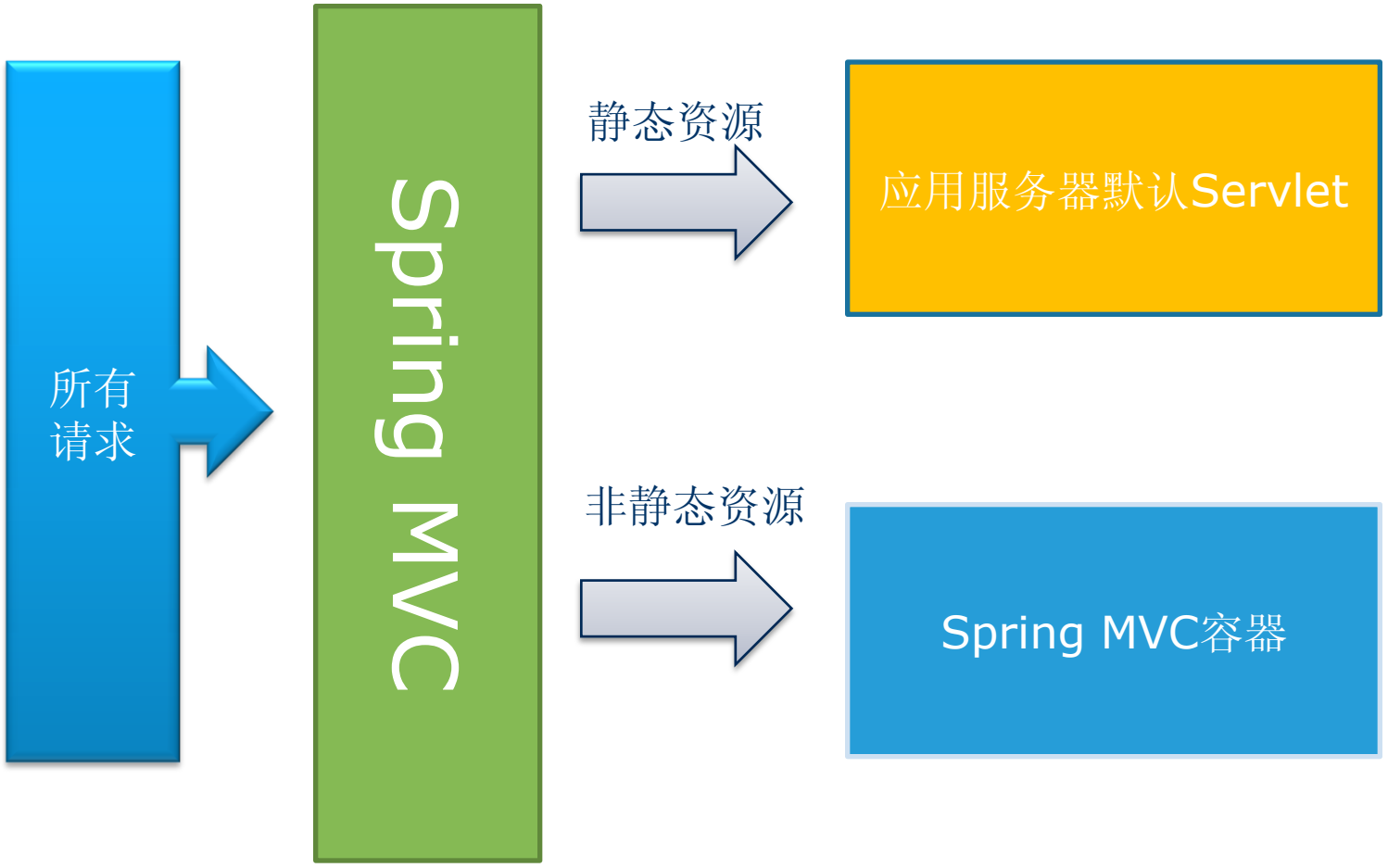
参考《Spring 3.x企业应用开发实战》

静态资源处理:使REST风格的URL成为实现

优雅REST风格的资源URL不希望带.html或.do等后缀，以下是几个优雅的URL。

- /blog/tom：用户tom的blog资源。
- /forum/java：java论坛板块资源。
- /order/4321：订单号为4321的订单资源；

静态资源处理:原理



静态资源处理:如何配置?

第一步：web.xml让所有请求都由Spring MVC处理

```
<servlet>
        <servlet-name>springServlet</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>springServlet</servlet-name>
        <url-pattern>/</url-pattern>
</servlet-mapping>
```

参考《Spring 3.x企业应用开发实战》

静态资源处理:如何配置?

第二步：springServlet-servlet.xml 让Web应用服务器处理静态资源

```
<mvc:default-servlet-handler/>
```

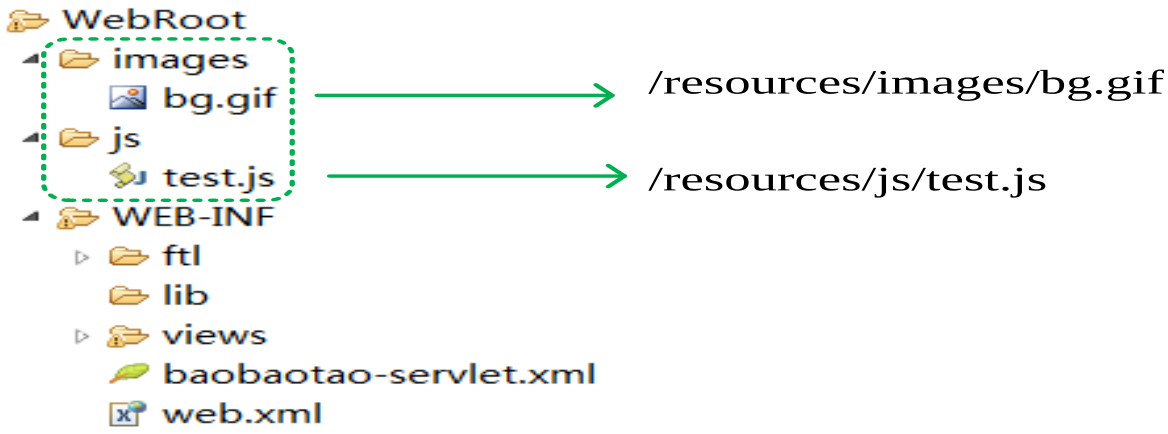
获取应用服务器的默认Servlet,大多数应用服务器的Servlet的名称都是“default”，如果默认不是“default”则使用

```
<mvc:default-servlet-handler
        default-servlet-name="<defaultServletName>"/>
```

参考《Spring 3.x企业应用开发实战》

物理静态资源路径映射逻辑资源路径

```
<mvc:resources mapping="/resources/**"
               location="/,classpath:/META-INF/publicResources/"/>
```




```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c"   uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>静态资源测试页面</title>
    <script src="<c:url value="/resources/js/test.js"/>" type="text/javascript"> </script>
  </head>
  <body>
    <script>test();</script>
  </body>
</html>
```

参考《Spring 3.x企业应用开发实战》

允许利用浏览器的缓存且不当心不同步

```
<mvc:default-servlet-handler/>
<bean id="rpe" class="com.baobaotao.web.ResourcePathExposer"
                    init-method="init"/>
<mvc:resources mapping="#{rpe.resourceRoot}/**"
                                          location="/" cache-period="31536000"/>
```

```
public class ResourcePathExposer implements ServletContextAware {
            public void init() {
                        String version = "1.2.1";
                        resourceRoot = "/resources-" + version;
                        getServletContext().setAttribute("resourceRoot",
                                    getServletContext().getContextPath()+resourceRoot);
            }
}
```

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-
8"%>
<%@ taglib prefix="c"      uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>静态资源测试</title>
    <script src="${resourceRoot}/js/test.js" type="text/javascript"> </script>
  </head>
   ...
</html>
```

AQ?

问题？？？

参考《Spring 3.x企业应用开发实战》

LOGO

谢 谢 !