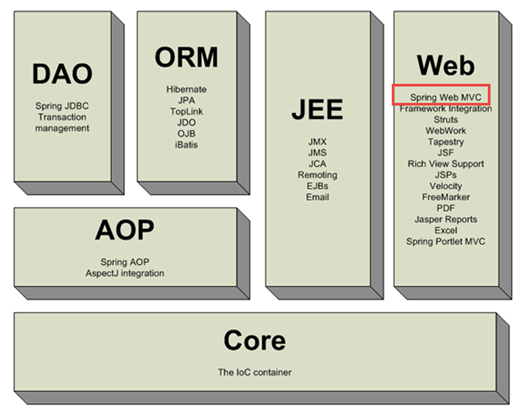
# 一、Spring MVC框架介绍

Spring MVC 是一个基于MVC的web框架，是Spring框架的一个模块,在Spring中的位置如下：



Spring MVC的执行流程如下图：

web容器(如Tomcat)

2.委托请求给处理器

1.http请求

Browser

页面控制器

前端控制器

8.响应请求

5.返回ModelAndView

3.调用业务对象

7返回视图

6.根据Model渲染视图

4.返回模型数据

数据模型

视图

1. 客户端发送一个HTTP请求，Web服务器接收到请求后，如果匹配前端控制器的请求映射路径(web.xml中配置)，Web容器会将请求转发给前端控制器；如上图的1。
2. 前端控制器接收到请求后，根据请求的信息(url、HTTP方法、请求报文头、请求参数、Cookie等)及HandlerMapping(相当于路由控制器)的配置找到处理请求的页面控制器(Controller)；如上图的2。
3. 页面控制器接收到请求后，进行功能处理；首先是收集和绑定请求参数到命令对象，并进行验证，然后将命令对象委托给业务对象进行处理；处理完成后返回相关数据给页面控制器，页面控制器封装成ModelAndView(包含视图名称和模型数据)对象返回给前端控制器，如图所示的3,4,5
4. 前端控制器会调用视图解析 器(ViewResolver)，根据视图名称和模型数据渲染出视图(可能是HTML页面、JSON数据或XML数据)，并返回给浏览器。如上图的6,7,8。

# 二、前端控制器：DispatcherServlet

DispatcherServlet是Spring MVC的核心，负责接收Http请求并协调Spring MVC各个组件以完成请求处理。

了解Spring MVC框架的工作机制，需要回答下面三个问题：

1）DispatcherServlet如何捕获特定http请求，交由Spring MVC框架处理？

2）位于web层的Spring容器(WebApplicationContext)如何与位于业务层的Spring容器(ApplicationContext)建立关联，使得web层的Bean可以调用业务层的Bean？

3）如何初始化Spring MVC的各个组件，并且装配到DispatcherServlet中？

## 2.1 DispatcherServlet配置

DispatcherServlet本质上是一个Servlet，所以可以在web.xml文件中配置处理的url模式，如：

<! -- 配置Spring配置文件路径 -->

<Context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*: spring-root.xml</param-value>

</context-param>

<! -- spring mvc 分发器 -->

<Servlet>

<servlet-name>spring-mvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

<! -- 配置文件，默認是：/WEB-INF/xxx-servlet.xml -->

<Init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*: spring-mvc.xml</param-value>

</init-param>

</servlet>

<! -- Spring MVC转发规则 -->

<Servlet-mapping>

<servlet-name>spring-mvc</servlet-name>

<url-pattern>\*.do</url-pattern>

</servlet-mapping>

1. 在①处，通过配置contextConfigLocation参数，指定业务层Spring容器的配置文件(多个用逗号分隔).
2. 在②处，配置DispatcherServlet，并指定web层Spring容器的配置文件(多个用逗号分隔).
3. 在③处配置DispatcherServlet匹配的url规则，此处所有.do结尾的url请求都会被捕获到。

**Remark**：①：Spring容器可以设置父子关系，以实现解耦；子容器可以访问父容器的Bean，而父容器不能访问子容器的Bean；在这里web层的Spring容器作为业务层Spring容器的子容器。②：web.xml可以配置多个DispatcherServlet，以处理不同的请求。

### 2.1.1 DispatcherServlet配置参数

DispatcherServlet遵循“约定大于配置”的原则，默认情况下可以不需要额外配置。下面是常用的配置参数：

|  |  |  |
| --- | --- | --- |
| **参数名称** | **说明** | **默认值** |
| namespace | DispatcherServlet 对应的命名空间，用来构造Spring配置文件的路径 | <servlet-name>-servlet |
| contextConfigLocation | DispatcherServlet上下文对应的Spring配置文件有多个，配置该属性，用逗号分隔 |  |
| publishContext | DispatcherServlet根据该属性觉得是否将WebApplicationContext发布到ServletContext的属性列表中，以便调用者可以访问到 | True |

## 2.2 Java Code配置Spring MVC

原理：在Servlet 3.0 环境中，容器会在类路径中查找实现了javax.servlet.ServletContainerInitializer的类，如果发现已有实现类，就会调用它来配置Servlet容器。在Spring中，org.springframework.web.SpringServletContainerInitializer 类实现了该接口，同时这个类又会查找实现了org.springframework.web.WebApplicationInitializer接口的类，并将配置任务交给这些实现类去完成，spring 提供一个便利的抽象类：AbstractAnnotationConfigDispatcherServletInitializer。

**public** **class** WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer

{

/\*\*

\* 配置root上下文

\*/

@Override

protected Class<?>[] getRootConfigClasses() {

return new Class<?>[] { RootConfig.class };

}

/\*\*

\* 配置Spring MVC

\*/

@Override

protected Class<?>[] getServletConfigClasses() {

return new Class<?>[] { WebAppConfig.class };

}

/\*\*

\* 指定被servlet处理的url

\*/

@Override

protected String[] getServletMappings() {

return new String[] { "/" };

}

}

@Configuration

@ComponentScan

**public** **class** RootConfig {

}

/\*\*

\* 配置Spring MVC

\*

\* **@author** wdhcxx

\*

\*/

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = { "com.wangdh.spring.mvc.controller" })

**public** **class** WebAppConfig **extends** WebMvcConfigurerAdapter {

@Bean

**public** ViewResolver viewResolver() {

InternalResourceViewResolver viewResolver = **new** InternalResourceViewResolver();

viewResolver.setViewClass(JstlView.**class**); // 设置视图类型

// viewResolver.setSuffix(".jsp");

**return** viewResolver;

}

/\*\*

\* 配置转换器：将请求信息转换为字符串，以String类型返回数据

\*

\* **@return**

\*/

@Bean

**public** StringHttpMessageConverter stringHttpMessageConverter() {

**return** **new** StringHttpMessageConverter(Charset.*forName*("UTF-8"));

}

/\*\*

\* 配置json转换器

\*

\* **@return**

\*/

@Bean

**public** MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter() {

// MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter = new MappingJackson2HttpMessageConverter();

// mappingJackson2HttpMessageConverter.setPrettyPrint(true);

//

// return mappingJackson2HttpMessageConverter;

**return** **new** MappingJackson2HttpMessageConverter () {

// 重写 writeInternal 方法，在返回内容前首先进行加密

@Override

**protected** **void** writeInternal (Object object, HttpOutputMessage outputMessage) **throws** IOException, HttpMessageNotWritableException

{

// 使用 Jackson 的 ObjectMapper 将 Java 对象转换成 Json String

ObjectMapper mapper = **new** ObjectMapper ();

String json = mapper.writeValueAsString (object);

// 加密

String result = json + "加密了！";

// 输出

outputMessage.getBody ().write (result.getBytes ());

}

};

}

/\*\*

\* 添加自定义类型转换器

\*/

@Override

**public** **void** addFormatters(FormatterRegistry registry) {

registry.addConverter(**new** StringToUserConverter());

**super**.addFormatters(registry);

}

/\*\*

\* 文件上传配置

\*

\* **@return**

\*/

@Bean

**public** MultipartResolver multipartResolver() {

CommonsMultipartResolver multipartResolver = **new** CommonsMultipartResolver();

multipartResolver.setDefaultEncoding("UTF-8");

multipartResolver.setMaxUploadSize(1024 \* 1024);

**return** multipartResolver;

}

/\*\*

\* 直接匹配到视图，不需要经过控制器

\*/

@Override

**public** **void** addViewControllers(ViewControllerRegistry registry) {

registry.addViewController("/login").setViewName("/go.jsp");

}

@Override

**public** **void** configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

**super**.configureDefaultServletHandling(configurer);

configurer.enable();

}

/\*\*

\* 静态资源处理器

\*/

@Override

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

**super**.addResourceHandlers(registry);

registry.addResourceHandler("/resource/\*\*").addResourceLocations("classpath:/META-INF/static");

}

/\*\*

\* 拦截器

\*/

@Override

**public** **void** addInterceptors(InterceptorRegistry registry) {

**super**.addInterceptors(registry);

}

/\*\*

\* 异常处理

\*/

@Override

**public** **void** configureHandlerExceptionResolvers(List<HandlerExceptionResolver> exceptionResolvers) {

SimpleMappingExceptionResolver simpleMappingExceptionResolver = **new** SimpleMappingExceptionResolver();

// 设置默认的错误视图

simpleMappingExceptionResolver.setDefaultErrorView("/defaultException.jsp");

// 设置异常在Model中保存的参数名，默认是exception

simpleMappingExceptionResolver.setExceptionAttribute("ex");

// 设置默认的响应码

simpleMappingExceptionResolver.setDefaultStatusCode(200);

// 配置解析出的viewName和statusCode的对应关系,key和value都是String类型

Properties statusCodes = **new** Properties();

statusCodes.setProperty("/aaa.jsp", "404");

simpleMappingExceptionResolver.setStatusCodes(statusCodes);

// 设置异常类(字符串)和viewName的对应关系，异常类可以是异常的一部分，也可以是异常父类的一部分

Properties mappings = **new** Properties();

mappings.put("ArithmeticException", "/aaa.jsp");

simpleMappingExceptionResolver.setExceptionMappings(mappings);

// 排除掉不处理的异常类型,精确匹配

simpleMappingExceptionResolver.setExcludedExceptions(Exception.**class**, ArithmeticException.**class**);

exceptionResolvers.add(simpleMappingExceptionResolver);

**super**.configureHandlerExceptionResolvers(exceptionResolvers);

}

}

## 2.3 DispatcherServlet装载Spring MVC组件

DispatcherServlet类有个initStrategies()方法：

**Protected** **void** initStrategies (ApplicationContext context) {

initMultipartResolver (context);

InitLocaleResolver (context);

InitThemeResolver (context);

InitHandlerMappings (context);

InitHandlerAdapters (context);

InitHandlerExceptionResolvers (context);

InitRequestToViewNameTranslator (context);

InitViewResolvers (context);

InitFlashMapManager (context);

}

会在WebApplicationContext初始化后执行，通过反射机制查找并装配Spring容器中用户显式定义的组件Bean，如果找不到，再装配默认的组件实例。

Spring默认的组件有：

# 本地化解析器

org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

# 主题解析器

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeResolver

# 处理器映射(2个)

org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\

org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

# 处理器适配器(3个)

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\

org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\

org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

# 异常处理器(3个)

org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\

org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\

org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

# 视图名称翻译器

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator

# 视图解析器

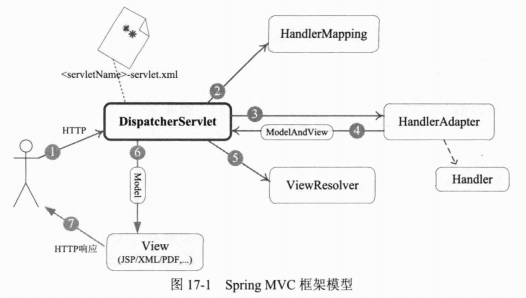
org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver

# flash管理器

org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFlashMapManager

DispatcherServlet装配各组件的逻辑如下：

|  |  |
| --- | --- |
| **组件类型** | **发现机制** |
| 文件上传解析器(1) | 1. 查找名为multipartResolver类型为MultipartResolver的Bean作为该类型组件； 2. 没有默认实现类。 |
| 本地化解析器(1) | 查找名称为localResolver类型为LocalResolver的Bean作为该类型的组件；没有找到则使用默认实现 |
| 主题解析器(1) |  |
| 处理器映射器(\*) |  |
| 处理器适配器(\*) |  |
| 异常处理器(\*) |  |
| 视图名称翻译器(1) |  |
| 视图解析器(\*) |  |
| Flash解析器(1) |  |



1.客户端发起一个http请求，web应用服务器接收到这个请求，如果匹配DispatcherServlet的请求映射规则，则web服务器将该请求转交给DispatcherServlet处理；

2. DispatcherServlet接收到请求后，根据请求信息(URL，http方法，请求报文头，请求参数，cookie等)及HandlerMapping配置找到处理请求的处理器(Handler)；

3.当DispatcherServlet根据HandlerMapping得到对应当前请求的Handler后，通过HandlerAdapter对Handler进行封装，再以同一的适配器接口调用Handler；

4.处理器Handler完成业务逻辑处理后将返回一个ModelAndView给DispatcherServlet，ModelAndView包含视图名称和模型数据信息；

5. DispatcherServlet借由ViewResolver完成视图名称到真正视图对象的解析工作；

6.得到真正的View后，DispatcherServlet使用ModelAndView中的模型数据进行视图渲染；

7.最终客户端得到的响应消息(html页面、xml或json数据等)

<?xml version="1.0" encoding="UTF-8"?>

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance http://www.springmodules.org/schema/cache/springmodules-cache.xsd http://www.springmodules.org/schema/cache/springmodules-ehcache.xsd"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_3\_0.xsd">

<display-name>Spring MVC</display-name>

<!-- 配置业务层的Spring容器 开始 -->

<!-- 通过指定context参数，让Spring使用 AnnotationConfigWebApplicationContext而非XmlWebApplicationContext -->

<context-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</context-param>

<!-- 指定标注了@Configuration的配置类，多个可以使用逗号分隔 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>

com.wangdh.spring.mvc.config.RootConfig

</param-value>

</context-param>

<!-- ContextLoaderListener监听器将根据上面的配置使用AnnotationConfigWebApplicationContext，根据contextConfigLocation指定的配置类启动Spring容器 -->

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

<!-- 配置业务层的Spring容器 结束 -->

<!-- 配置Spring MVC 开始 -->

<servlet>

<servlet-name>simpleMvc</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:simpleMvc-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>simpleMvc</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

<!-- 配置Spring MVC 结束 -->

<!-- 首页列表 -->

<welcome-file-list>

<welcome-file>/index</welcome-file>

</welcome-file-list>

</web-app>

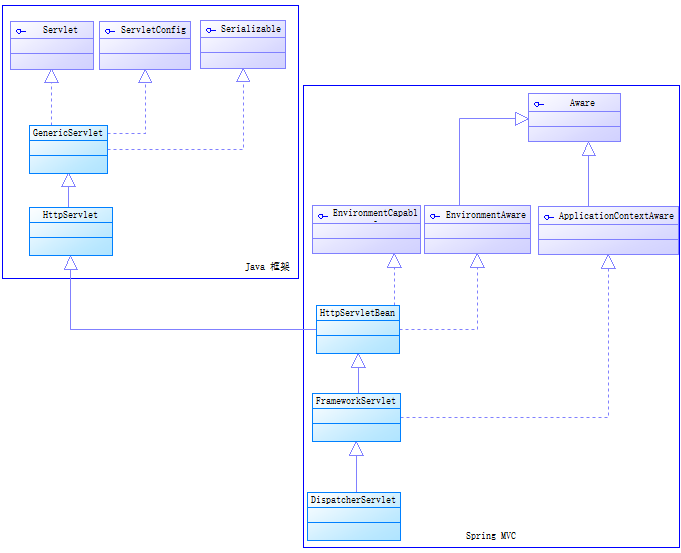
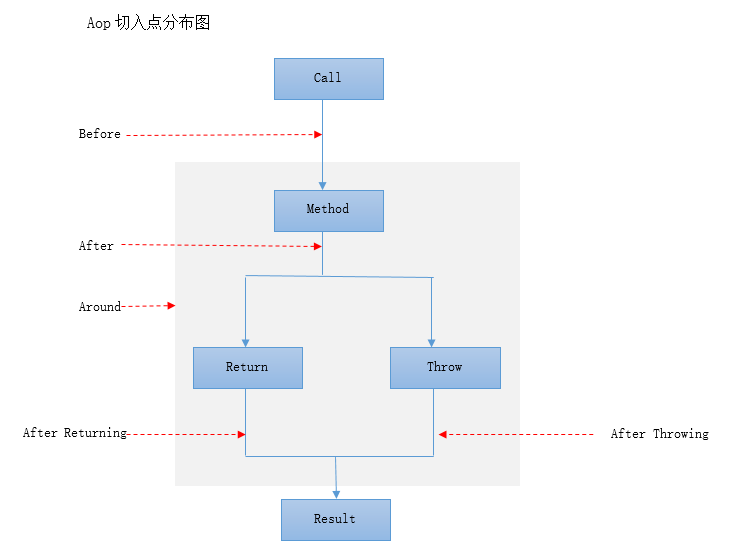


图1 Spring MVC Servlet继承关系



Aop切入点分布图

Call

Method

Return

Throw

Result

Before

After

Around

After Returning

After Throwing

# 三、Spring MVC常用注解

3.1 @Controller

注解在类上，表明这个类是Spring MVC里的Controller,将其声明为Spring的一个bean；Dispatcher Servlet会自动扫描注解了此注解的类，并将web请求映射到注解了@RequestMapping的方法上。

3.2 @RequestMapping

用来映射web请求(访问路径和参数)，可注解在类或方法上，注解在方法上的@RequestMapping路径会继承注解在类上的路径。

**@RequestMapping的value、method、params、headers分别表示请求URL、请求方法、请求参数和报文头的映射条件**。

3.3 @ResponseBody

支持将返回值放在response体内，而不是返回一个页面。

3.4 @RequestBody

支持request的参数在request体中，而不是直接链接在地址后面，此注解放置在参数前。

3.5 @PathVariable，@RequestParam，@RequestHeader，@CookieValue

用来接收路径参数，次注解放置在参数前。

3.6 @RestController

是一个组合注解，组合了@Controller和@ResponseBody

request.getParameter 🡺 参数存在URL中或form表单提交可获取到

数据流方式获取数据： 🡪 json/text方式提交的数据

ServletInputStream ris = request.getInputStream();

StringBuilder content = new StringBuilder();

byte[] b = new byte[1024];

int lens = -1;

while ((lens = ris.read(b)) > 0) {

content.append(new String(b, 0, lens));

}

String strcont = content.toString();

**文件类型提交，流数据获取**：

----------------------------083539673244341741611103

Content-Disposition: form-data; name="userName"

wdh

----------------------------083539673244341741611103—

**form表单提交，流数据获取**

userName=wdh&age=10

**json或text方式获取**

{"userName":"hahaha"}

简单参数绑定： 🡪 form表单方式能绑定到值

@RequestParam 🡪 form表单参数绑定

@RequestBody 🡪 json方式提交，从请求体中获取参数,得到String或转成相应实体

# 四、消息转换器：HttpMessageConverter<T>

负责将请求消息转换为一个对象，把对象输出为响应消息。

|  |  |
| --- | --- |
| 实现类 | 功能说明 |
| StringHttpMessageConverter | 用途：将请求消息转为字符串   1. T为String类型 2. 可读取所有媒体类型(\*/\*)的请求信息，可通过设置supportedMediaTypes属性指定媒体类型 3. 响应消息的媒体类型为text/plain |
| FormHttpMessageConverter | 用途：将表单数据读取到MultiValueMap中   1. T为MultiValueMap<Stirng,?> 2. 支持读写application/x-www-form-urlencoded媒体类型 3. 支持响应multipart/form-data类型 |
| AllEncompassingFormHttpMessageConverter | 扩展自FormHttpMessageConverter，如果部分表单属性是xml数据，可使用该转换器 |
| ResourceHttpMessageConverter | 用途：读写Resource  1. 可读取所有媒体类型(\*/\*)的请求信息 |
| MarshallingHttpMessageConverter |  |
| MappingJackson2HttpMessageConverter |  |

HttpMessageConverter使用的两种方式：

1. [使用 @RequestBody](mailto:1.使用@RequestBody) 和@ResponseBody对处理方法进行标注
2. 使用HttpEntity<T>/ResponseEntity<T>作为方法的入参或返回值

Spring首先根据请求头或响应头的Accept属性选择匹配的HttpMessageConverter，然后根据参数类型或泛型类型的过滤得到匹配的HttpMessageConverter，如果找不到可用的Converter则报错。

**处理XML和Json**

MarshallingHttpMessageConverter：处理Xml格式的请求或响应消息

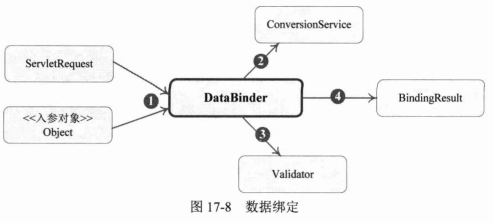
Jaxb2RootElementHttpMessageConverter:同上，底层使用JAXB

MappingJackson2HttpMessageConverter：处理json格式的请求或响应消息

**将模型数据暴露给视图的几种方式**

1. ModelAndView：返回值类型为ModelAndView时，方法体可通过该对象添加数据；
2. @ModelAttribute：方法入参标注该属性，入参的对象会放到数据模型中；
3. Map及Model：方法入参是org.springframework.ui.Model、org.springframework.ui.ModelMap、java.util.Map则返回时，Map中的数据会自动的添加到模型中
4. @SessionAttribute：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

# 五、数据绑定



类型转换器接口：

1.Converter<S,T>

2.GenericConverter

3.ConverterFactory

数据格式化器：

Formatter<T> -- 格式化器接口

NumberFormatter – 数字类型格式化

CurrencyFormatter – 货币类型格式化

PercentFormatter – 百分数格式化

@NumberFormat,@DateTimeFormat

数据校验

JSR-303 ： bean数据校验标准

Hibernate Validator ： JSR-303的一个参考实现

Validator接口

@Valid注解

BindingResult ： 绑定结果，内含验证错误

# 六、视图：View

InternalResourceView：将jsp或其他资源封装成一个视图

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；

MarshallingView ： 通过oxm的Marshall技术将数据模型以xml方式输出

MappingJackson2JsonView ： 将模型数据通过Jackson开源框架的ObjectMapper以Json方式输出

RedirectView ： 重定向视图

**视图解析器**

BeanNameViewResolver ： 将逻辑视图解析成一个bean，bean的ID等于逻辑视图名；

XmlViewResolver ：

ResourceBundleViewResolver：国际化资源解析器

InternalResourceViewResolver ： 将视图名解析为URL文件

XsltViewResolver：将视图名解析为一个指定的XSLT样式表的URL文件

FreeMarkerViewResolver：解析为基于FreeMarker模板技术的模板文件

VelocityViewResolver、VelocityLayoutViewResolver：解析为基于Velocity模板技术的模板文件

ContentNegotiatingViewResolver：不负责具体的视图解析，而是作为中间人角色根据请求的MIME类型，从上下文选择一个合适的视图解析器，委托其进行视图解析。

# 七、文件上传：MultipartResolver