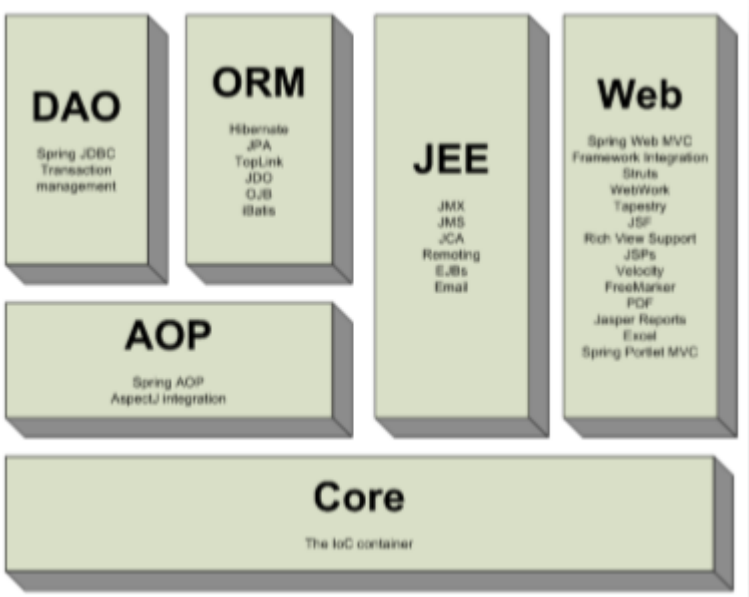
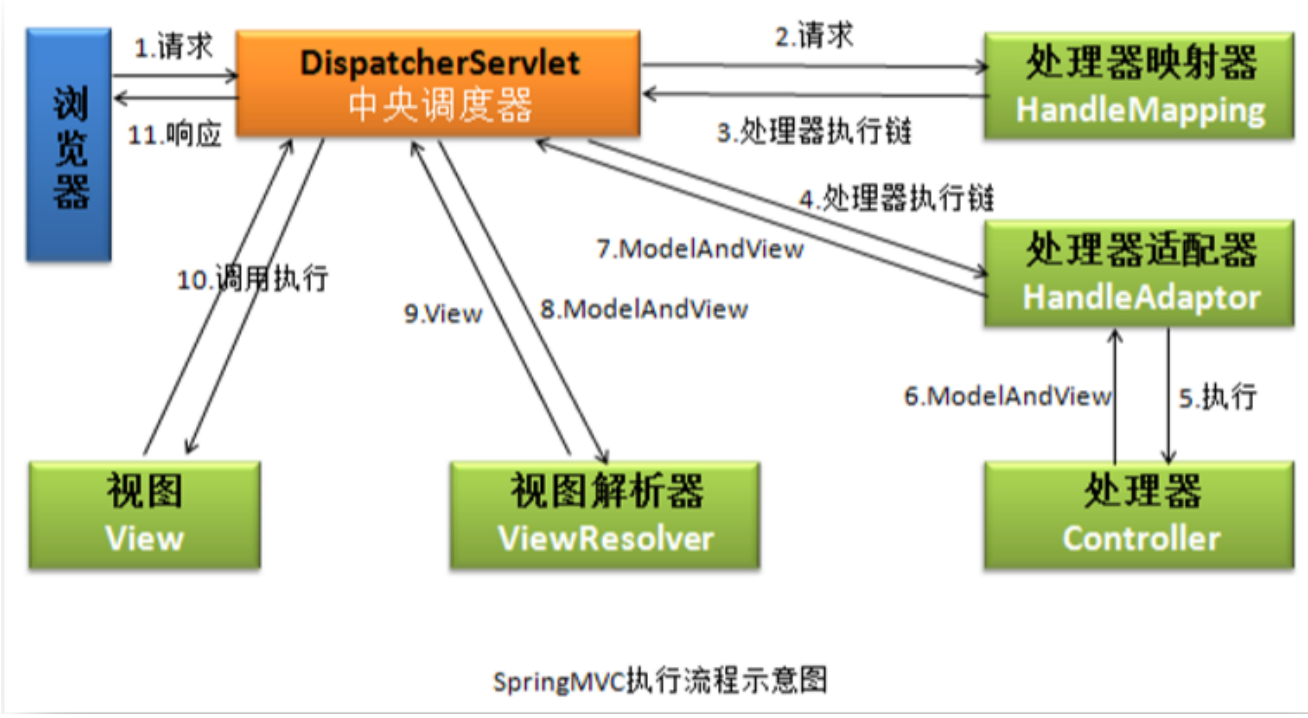
简介

SpringMVC也叫Spring web mvc，属于表现层的框架。SpringMVC是Spring框架的一部分，是在Spring3.0后发布的。



由以上Spring的结构图可以看出，Spring由四大部分组成：Dao部分（Dao与ORM）、AOP部分、Web部分（JEE与Web）、及IoC容器部分（Core）。

SpringMVC执行流程



SpringMVC执行流程示意图

执行流程简单分析

1. 浏览器提交请求到中央调度器
2. 中央调度器直接将请求转给处理器映射器
3. 处理器映射器会根据请求，找到处理该请求的处理器，并将其封装为**处理器执行链**后返回给中央调度器
4. 中央调度器根据处理器执行链中的处理器，找到能够执行该处理器的处理器适配器。
5. 处理器适配器调用执行处理器
6. 处理器将处理结果及要跳转的视图封装到一个对象ModelAndView中，并将其返回给处理器适配器。
7. 处理器适配器直接将结果返回给中央调度器
8. 中央调度器调用试图解析器，将ModelAndView中的视图名称封装为视图对象。
9. 视图解析器将封装了的视图对象返回给中央调度器
10. 中央调度器调用试视图对象，让其自己进行渲染，即进行数据填充，形成响应对象
11. 中央调度器响应浏览器

API简要说明

- **DispatcherServlet**

  中央调度器，也成为前端控制器，在MVC架构模式中充当控制器，DispatcherServlet是整个流程的控制中心，由它调用诸如处理器映射器、处理器适配器、视图解析器等其它组件处理用户请求。中央调度器的存在降低了组件之间的耦合度。

- **HandlerMapping**

  处理器映射器，负责根据用户请求找到相应的将要执行的Handler，即处理器，并将处理器封装为处理器执行链传给中央调度器。

- **HandlerAdapter**

  处理器适配器，通过HandlerAdapter对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。中央调度器会根据不同的处理器自动适配器，以执行处理器。

- **Handler**

处理器，也称为后端控制器，在DispatcherServlet的控制下Handler调用Service层对具体的用户请求进行处理。由于Handler涉及到具体用于业务请求，所以一般情况下需要程序员根据业务需求自己开发Handler。

- **ViewResolver**

  试图解析器，负责将处理结果生成View视图，ViewResolver首先将逻辑视图名解析为物理视图名，在生成View视图对象。最后将处理结果通过页面形式展示给用户。

  SpringMVC框架提供了很多的View视图类型，包括：JstlView、RedirectView等。一般情况下需要通过页面标签或页面模板技术将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面。

配置式开发

前言

所谓配置式开发是指，“处理器类是程序员手工定义的”、实现了特定接口的类，然后在SpringMVC配置文件进行显式的、明确的注册的开发方式。

在SpringMVC配置文件中如未明确的定义SpringMVC中的组件，如：处理器映射器、处理器适配器、视图解析器。将会自动采用DispatcherServlet默认的组件。在SpringMVC中查看DispatcherServlet.properties可知：

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

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

org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\
    org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\
    org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
    org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.annotation.AnnotationMethodHand
    org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
    org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

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

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

从上面的内容可知，这些组件的默认配置如下：

组件名称	默认值
处理器映射器	BeanNameUrlHandlerMapping 、DefaultAnnotationHandlerMapping
处理器适配器	HttpRequestHandlerAdapter、HttpRequestHandlerAdapt、AnnotationMethodHandlerAdapter
视图解析器	InternalResourceViewResolver

如果在SpringMVC的配置文件中自定义配置DispatcherServlet的组件的话，将采用自定义的组件进行运作。

DispatchServlet映射路径问题

```
<servlet>
    <servlet-name>spring_dispathcer</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring_dispathcer</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

需要注意的是，在映射路径上最好使用后缀匹配方式，如写成*.do

- **不能写成/***

  这里的url-pattern不能写成/*，因为DispatcherServlet会将动态页面的跳转请求，及向JSP页面的跳转也当做是一个普通的Controller请求。将被中央调度器拦截并会调用处理器映射器为其查找相应的处理器。当然是找不到的，所以在这种情况下，所有的JSP页面跳转均会报404错误

- **最好也不要写成/**

  最好也不要写成 / ，因为DispatcherServlet会将向静态资源的获取请求，例如：css、js、jpg、png等资源的获取，当作是一个普通的Controller请求。中央调度器同样将会进行拦截并调用处理器映射器为其进行查找相应的处理器。

> 静态资源访问

<url-pattern/>的值并不是说写为/后，静态资源就无法访问了。经过一些配置，该问题也是可以解决的。

- **使用Tomcat中的default的Servlet**

  在Tomcat中，有一个专门用于处理器静态资源访问的Servlet-DefaultServlet。其<servlet-name/>为defalut。可以处理各种静态资源访问请求，该Servlet注册在Tomcat服务器的web.xml中。

我们需要做的就是直接使用它即可，即直接在项目的web.xml中注册<servlet-mapping/>即可。

```
 <servlet-mapping>
     <servlet-name>default</servlet-name>
     <url-pattern>*.jpg</url-pattern>
 </servlet-mapping>
```

- **使用<mvc:default-servlet-handler/>**

  <mvc:default-servlet-handler/>会将对静态资源的访问请求添加到SimpleUrlHandlerMapping的urlMap中，key就是请求的URI，而value则为默认Servlet请求处理器DefaultServletHttpRequestHandler对象。而该处理器调用了Tomcat的DefaultServlet来处理静态资源的访问请求

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

- **使用<mvc:resources/>**

  在Spring3.0.4版本后，Spring中定义了专门用于处理静态资源访问请求的处理器ResourceHttpRequestHandler。并且添加了<mvc:resource/>标签，专门用于解决静态资源无法访问问题。

```
 <mvc:resources mapping="/photo/**" location="/photo/"></mvc:resources>
```

调度组件

处理器映射器

- **BeanNameUrlHandlerMapping**

  根据注册处理器id进行适配

- **SimpleUrlHandlerMapping**

  可通过设置属性对不同请求Url请求同一个处理器

处理器

- **继承AbstractControler**

  可设置访问该控制器的请求方式

- **继承MultiActionControler**

  不仅可以设置访问该控制器的请求方式，而且同个控制器可有多个请求方法。通过注入不同的方法名解析器，可实现Url请求资源名匹配请求方法，Url请求携带参数匹配请求资源。

处理器适配器

- **SimpleControllerHandlerAdapter**

  所有实现了Controller接口的处理器Bean，均是通过此适配器进行适配，执行的。

- **HttpRequestHandlerAdapter**

  所有实现了HttpRequestHandler接口的处理器Bean，均是通过此适配器进行适配、执行的。

视图解析器

- **InternalResourceViewResolver**

  该视图解析器可设置解析视图名的前缀和后置，那么ModleAndView指定待转发的视图的逻辑路径时可对其进行简写，但缺点就是不可访问外部资源、存在程序冗余

- **BeanNameViewResolver**

  可解析外部资源View，在ModleAndView中通过指定注册的View的ID确定跳转资源的路径

- **XmlViewResolver**

  注册的View可存在到另外的Beans注册XML文件中，随后可通过设置该解析器的属性指定文件路径
- **ResourceBundleViewResolver**

  注册的View可存在到另外的Beans注册properties文件中，随后可通过设置该解析器的属性指定文件路径

视图

- **JstlView**

  定义内部资源视图对象
- **RedirectView**

  定义外部资源视图对象

ModelAndView

处理器在处理请求之后可将处理的数据存到到ModleAndView中的Model中，在通过View指定跳转显示的页面，将数据传递到页面中。

注解式开发

前言

注解式开发是SpringMVC开发Web应用最重要的方式，相比配置式开发，注解式开发简单许多，并写相对灵活很多。

请求方法映射

在每个请求方法都必须添加@RequestMapping注解，告知SpringMVC该方法为接收请求处理的方法。

若将@RequestMapping注解到Class上，那么该类中的所有请求方法都共享该注解上的参数值。

在该注解上可以设置**请求方法的请求方式**、**矫正请求参数**、**接收路径的变量**、**指定请求Url的参数**

在请求Url上通配符的应用

- ***与**的区别**

  在请求路径上使用通配符，*表示必须存在随意名一级的路径，**表示可存在零级或以上随意名的路径

请求方法形参

注解式开发中的请求方法可以存在五种类型的参数，并且可以有不同的重载方式，这也就是注解式开发相对配置式开发的灵活之处。

1.
2.
3.
4.
5.
6.
7.
8. 除了使用ModelAndView方式外。还可以使用Map、Model和ModelMap来向前台页面创造

使用后面3种方式，都是在方法参数中，指定一个该类型的参数.

请求方法返回值

根据处理请求的需求，请求方法返回值可以为；

SpringMVC核心技术

重定向和转载

转载

默认情况下，当处理器方法返回ModelAndView时，跳转到指定的View，使用的是请求转发。但也可显示的进行指出。此时，需在setViewName()指定的视图前添加forward，且此时的视图不会再与InternalResourceViewResolver视图解析器中的前缀和后缀进行拼接。即必须写出相对于项目根的路径。如：

```
modelAndView.setViewName("forward:/jsp/transpond.jsp");
```

需要注意的是，此时如果DispatcherServlet默认的视图解析器组件InternalResourceViewResolver被其他自定的视图解析器所替代的话，将会报无法解析该视图的错误，此时应该再配置InternalResourceViewResolver解析器。

转载的方式

1. 返回ModelAndView
2. 返回String
3. 返回Void，通过request

数据传递

向下传递数据的方式有以下：

1.
2.
3.
4.

重定向

重定向时需指定的视图前添加redirect，且此时的视图不会再与InternalResourceViewResolver视图解析器中的前缀和后缀进行拼接。即必须写出相对于项目根的路径。如：

```
modelAndView.setViewName("redirect:/jsp/transpond.jsp");
```

重定向的方式

1. 返回ModelAndView
2. 返回String
3. 返回Void，通过Respond

数据传递

再重定向时，请求参数是无法通过request的属性向下级资源中传递的，但可以通过以下方式将请求参数向下传递

1.
2.
3.
4.

异常处理

自己看讲义吧！！！

类型转换器

类型处理器的配置

类型转换器定义完毕后，需要在SpringMVC的配置文件中对类型转换进行配置。首先要注册类型转换器，然后再注册一个转换服务Bean，将类型转换服务注入给该转换服务Bean，最后由处理器适配器来使用该转换服务Bean。

1. **注册类型转换器**

```
<bean id="myDateConverter" class="com.jr.springmvc.core.datetransform.MyDateConverter">
</bean>
```

2. **创建转换服务Bean**

   由于SpringMVC对于类型转换的完成，不是直接使用类型转换器，而是通过创建具有多种转换功能的转换服务Bean来完成的。

```
<bean id="myConversionService"
class="org.springframework.context.support.ConversionServiceFactoryBean">
    <!--设置单个类型转化器-->
    <!--<property name="converters" ref="myDateConverter"></property>-->
    <!--设置多个类型转换器-->
    <property name="converters">
        <set>
            <ref bean="myDateConverter"></ref>
        </set>
    </property>
</bean>
```

3. **使用转换服务Bean**

   采用MVC的注解驱动注册方式，可以将转换服务直接诸如给处理器适配器。

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

数据验证

配置环境

1. **引入Jar**

除了SpringMVC的Jar外，我们还需要导入Hibernate Validator的Jar包。这些Jar包，可以从Hibernate官网下载或直接添加Maven依赖：

```
<!--Hinernate验证器-->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.4.2.Final</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.validation/validation-api -->
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.1.Final</version>
</dependency>
```

2. **配置SpringMVC文件**

验证器由SpringMVC框架的LocalValidatorFactoryBean类生成，而真正验证器的提供者则是HibenateValidator。

在SpringMVC配置文件中将验证器注册后，需要将其注入给注解驱动。

```
<!--注册验证器-->
<bean id="myValidator"
class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
    <property name="providerClass" value="org.hibernate.validator.HibernateValidator">
</property>
</bean>

<!--注册MVC注解驱动,开启验证器-->
<mvc:annotation-driven  validator="myValidator"></mvc:annotation-driven>
```

处理器方法的验证

1. 在封装数据的对象上添加验证添加注解

```
public class Person {
    @NotEmpty(message = "姓名不能为空")
    @Size(min = 3, max = 6, message = "姓名长度必须在{min}~{max}之间")
    String name;

    @NotEmpty(message = "年龄不能为空")
    @Min(value = 0, message = "不能小于零岁")
    @Max(value = 200, message = "不能大于200岁")
    String age;

    @NotEmpty(message = "号码不能为空")
    @Pattern(regexp = "^1[13578]\\d{9}$",message ="手机号码格式不正确")
    String mobile;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAge() {
        return age;
    }

    public void setAge(String age) {
        this.age = age;
    }

    public String getMobile() {
        return mobile;
    }

    public void setMobile(String mobile) {
        this.mobile = mobile;
    }
```

```
    }
```

2. 注意在处理器方法上封装数据的对象上添加@Validated注解
3. 在处理器方法的形参上添加BindingResult对象用于接收判断验证错误信息。

```
@RequestMapping("/validator/handledRequest")
public void handledRequest(@Validated Person person, BindingResult br) {


}
```

文件上传

配置环境

SpringMVC实现文件上传，需要在添加两个Jar包，一个是文件上传的Jar包，一个是其所依赖的IO包。

```
<!-- 上传组件包 -->
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.3.1</version>
</dependency>
<!--上传依赖IO包-->
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.4</version>
</dependency>
```

配置上传处理器

处理器方法中的形参MultipartFile对象需要由上传处理器生成。

在SpringMVC的配置文件中注册MultipartFile接口的实现类CommonsMultipartResolver的Bean，要求该Bean的**id必须为multipartResolver**。因为中央调度器会自动在SpringMVC的组件配置中自动查找才id的对象，并装配到自身的属性中。

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!--设置编码集-->
    <property name="defaultEncoding" value="utf-8"></property>
    <!--限制总上传大小-->
    <!--<property name="maxUploadSize" value="102400"></property>-->
</bean>
```

上传多个文件问题

用于接收表单元素所提交的处理器方法的形参类型为MultipartFile[]数组，且必须使用@RequestParam修饰。

为什么上传单个文件时MultipartFile参数不用使用@RequestParams修饰，而上传多个文件时MutipartFile[]需要使用@RequestParam修饰呢?

因为在上传单个文件时，每个表单中的文件对象，框架均会将其转换为MultipartFile类型这是与上传单个文件是相同的，不需要@ReqeustParam修饰。但上传多个文件时，处理器方法需要的不是MultipartFile类型参数，而是MultipartFile[]数组类型。默认情况下框架只会将一个一个的表单元素转换为一个一个的对象，但并不会将这多个对象创建为一个数组对象。

此时，需要使用@RequestParam来修饰这个数据参数，向框架表明，表单传来的参数与处理器方法接收的参数名称与类型相同，需要框架调用相应的转换器将请求参数转换为方法参数类型。所以，对于上传多个文件，处理器方法的MultipartFile[]数组参数必须使用注解@RequestParam修饰。

拦截器

生命周期

- **preHandle**

  该方法在处理器方法执行之前执行，其返回值为boollean，若为true，则紧接着会执行处理器方法，且会将afterCompletion()方法放入到一个专门的方法栈中等待执行。若为false将终止接下来的请求。

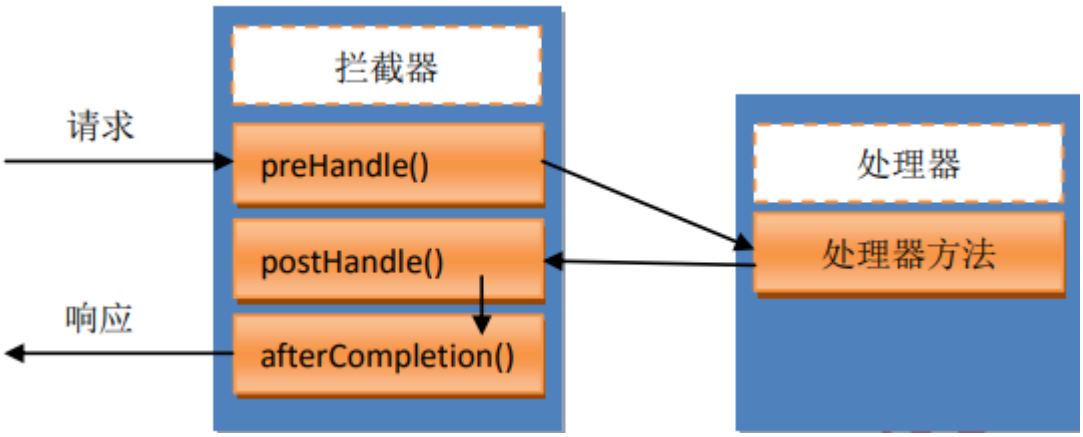
- **postHandle**

  该方法在处理器方法执行之后执行。处理器方法若最终未被执行，则该方法不会执行。由于该方法是在处理器方法执行完后执行，且该方法参数中包含ModelAndView，所以该方法可以修改处理器方法的处理结果数据，且可以修改跳转方向。

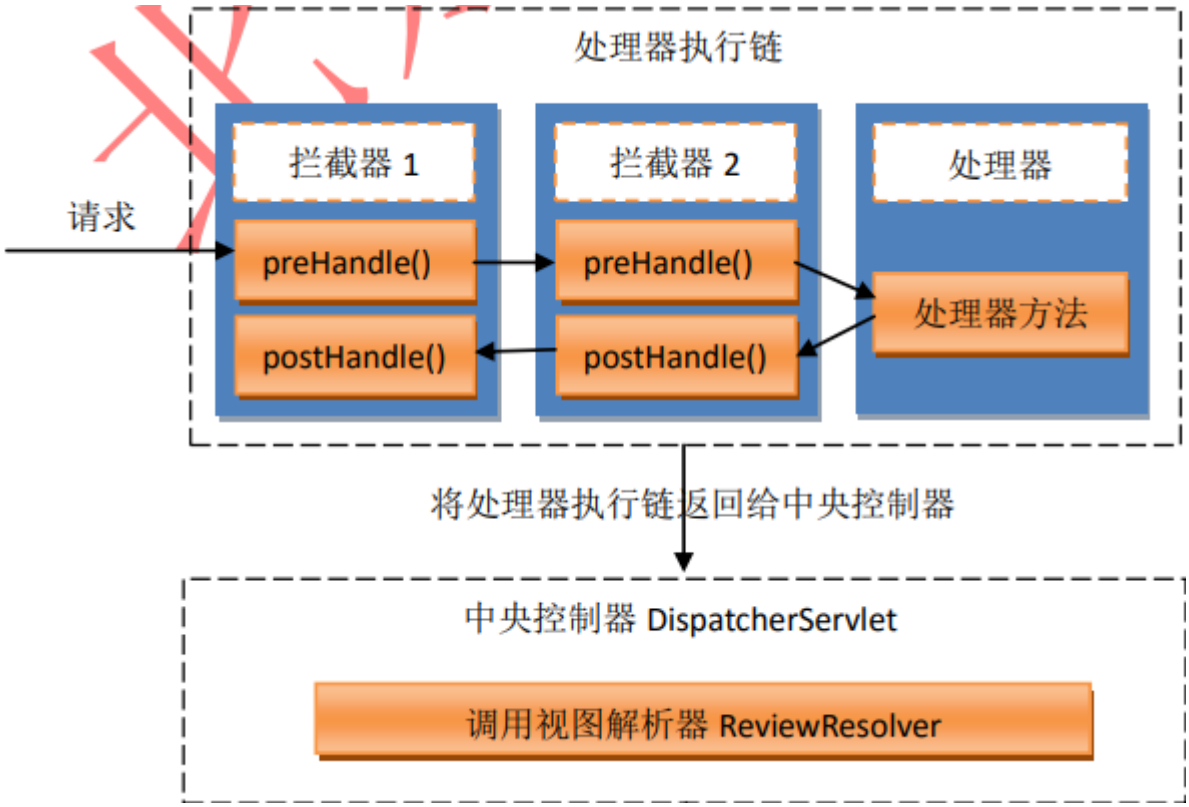
- **afterCompletion**

  当preHandle()方法返回true时，会将该方法放到专门的方法栈中，等到对请求进行响应的所有工作完成之后才执行该方法。即该方法是在中央调度器渲染（数据填充）了响应页面之后执行，也就是即将将处理数据响应给客户端之前，此时对ModleAndView在操作也对响应无济于事。

  可进行资源释放等操作



多个拦截器的执行



从上图可以看出，只要有一个preHandle()方法返回false，则上部的执行链将会断开。其后续的处理器方法与postHandle()方法将无法执行。但，无论执行链执行情况怎样，只要方法栈中有方法，即执行链中只要有preHandle()方法返回true，就会执行方法栈中的afterCompletion()方法。最终都会给出响应。

配置总结

类与接口

> 中央处理器

DispatcherServlet

> 处理静态资源问题

- **使用Servlet资源映射，映射到Tomaat的DefalutServelt去处理静态资源**

```
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.jpg</url-pattern>
</servlet-mapping>
```

- **资源请求先到DefaultServletHttpRequestHandler，而该处理器随后在调用DefalutServlet去处理**

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

- **使用Spring专门用于处理静态资源访问请求的处理器ResourceHttpRequestHandler**

```
<mvc:resources location="静态资源路径" mapping="映射路径"></mvc:resources>
```

> 处理中文乱码

CharacterEncodingFilter

> 处理器映射器

- **BeanNameUrlHandlerMapping**

  通过处理器注册ID来映射处理器
- **SimpleUrlHandlerMapping**

  自定义配置处理器映射

> 处理器适配器

- **SimpleControllerHandlerAdapter**

  适配并执行所有实现了Controller接口的处理器Bean
- **HttpRequestHandlerAdapter**

  适配并执行所有实现了HttpRequestHandler接口的处理器Bean

> 处理器

- **Controller**

  最原始的Controller，继承该类则表示该类是处理请求的类
- **AbstractController**

继承自Controller，多了一个可以限定请求方式的方法

- **MultiActionController**

  继承自AbstractController，它不仅可以限定请求方式，而且可以定义多个处理请求方法，通过请求URL中写上方法名就可以访问到指定方法上。它的实现是通过内部的MethodNameResolver对象做到的。

  三种MethodNameResolver

```
public class MyController extends MultiActionController {

    public ModelAndView doFirst(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        ModelAndView mv = new ModelAndView();

        mv.addObject("method", "doFirst()");

        mv.setViewName("show");
        return mv;
    }

    public ModelAndView doSecond(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        ModelAndView mv = new ModelAndView();

        mv.addObject("method", "doSecond()");

        mv.setViewName("show");
        return mv;
    }
}
```

  - **InternalPathMethodNameResolver**

    解析请求URL中的资源名称去映射方法名

  - **ParameterMethodNameResolver**

    注册ParameterMethodNameResolver时配置指定URL去映射请求方法

```
配置方法名解析器
<bean id="propertiesMethodNameResolver" class="PropertiesMethodNameResolver">
    <Property name="mappings">
        <prop key="doFirst.do">doFirst</prop>
        <prop key="doSecond.do">doSecond</prop>
    </Property>
</bean>
```

  - **ParameterMethodNameResolver**

    解析指定URL参数内的值映射方法名

```
配置方法名解析器
<bean id"parameterMethodNameResolver" class="ParameterMethodNameResolver">
    <Property name="paraName" value="method"></Property>
</bean>
```

那么对应的doFirst()方法请求是

```
http://127.0.0.1:8080/controller/my.do?method=doFirst
```

> 视图解析器

- **InternalResourceViewResolver**

  内部资源视图解析

- **BeanNameViewResolver**

  视图对象名视图解析器

- **XmlViewResolver**

  解析指定XML文件内注册的视图对象的视图解析器

- **ResourceBundleViewResolver**

  解析指定Properties文件内注册的视图对象的视图解析器

> 视图和数据

ModelAndView

> 视图

- **RedirectView**

  外部资源视图

- **JstlView**

  内部资源视图

> 数据模型

Model

> 专门用于携带重定向参数

RedirectAttributes

> 异常解析器接口

HandlerExceptionResolver

> 类型转换器接口

Converter<Object source,Object target>

> 数据验证信息封装对象

BindingResult

> 客户端文件上传数据封装对象

MultipartFile

> 自定义拦截器实现接口

HandlerInterceptor

注解

> 声明处理器注解

```
@Controller
```

> 指定请求URL映射处理器

```
@RequestMapping
```

> 校正请求参数名

```
@RequestParam
```

> 处理器方法形参映射路径变量

```
@PathVariable
```

> 声明处理方法返回值封装成JSON数据返回给客户端

```
@ResponseBody
```

> 将请求传递中Content-Type指定格式的数据封装成使用该注解的Bean

```
@RequestBody
```

> 异常处理注解

```
@ExcetionHandler
```

> 声明进行数据验证

```
@Validated
```

配置组件

> 注册MVC注解驱动

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

- 属性

> 自定义配置异常映射解析器

```
<bean class="SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <prop key="拦截异常类路径">异常处理页面路径</prop>
        </props>
    </property>
    <property name="defaluErrorView" value="默认异常处理页面路径" />
    <property name="exception" value="异常对象引用名"/>
</bean>
```

> 注册转换服务

```
<bean id="conversionService" class="ConversionServiceFactoryBean">
    <property name="converters" ref="注册类型转换器ID"></property>
</bean>
```

> 注册验证器

```
<bean id="myValidate"
class="org.Springframework.validation.beanvalidation.LocalValidatorFactoryBean">
    <property name="providerClass" value="数据验证产品实现类"></property><!--可以使用
Hibernate的数据验证产品org.hibernate.validator.HibernateValidator-->
</bean>
```

> 文件上传处理器

```
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <property name="defalutEncoding" value="编码格式"></property>
    <property name="maxUploadSiez" value="最大上传数据大小"></property>
</bean>
```

> 注册拦截器

```
<mvc:interceptors>
    <mvc:interceptor>
        <mvc:mapping path="拦截请求URL"/><!--/**表示拦截所有请求-->
        <bean class="拦截器类路径"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```