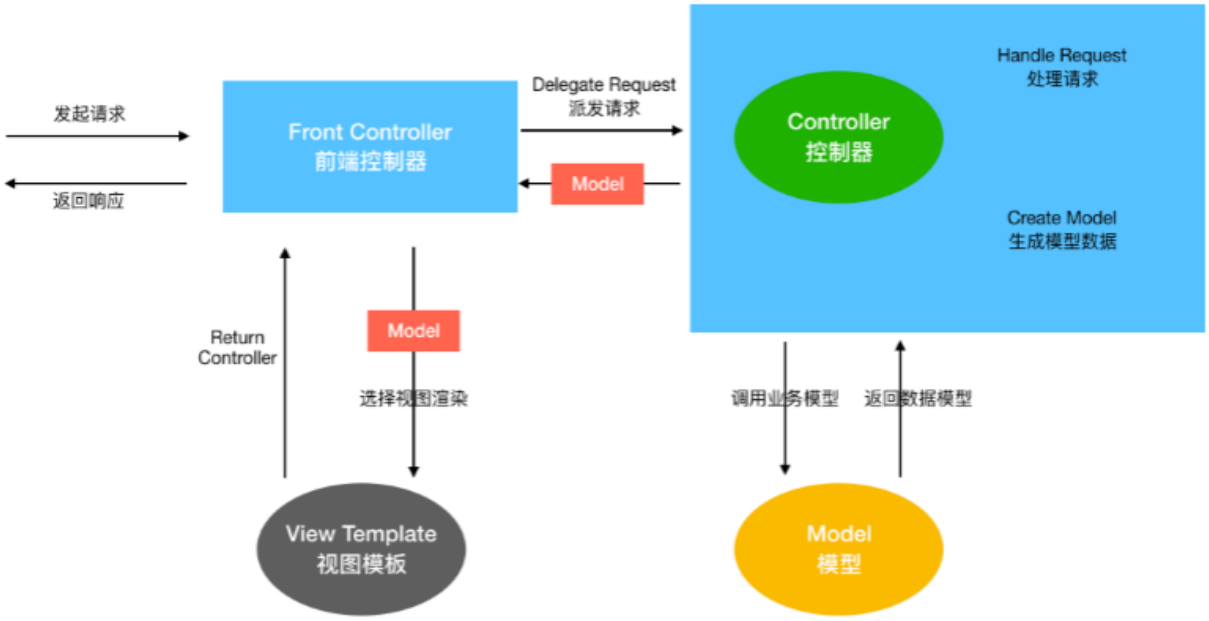
1、SpringMVC核心处理流程

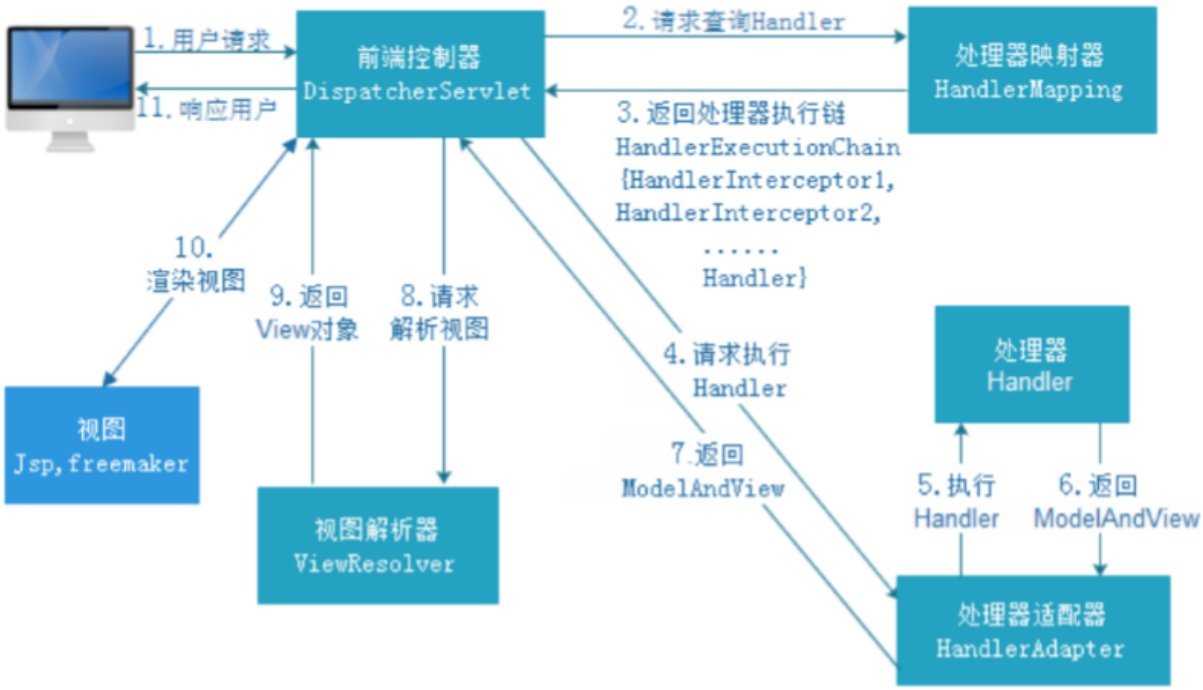
SpringMVC是对servlet的封装，它有一个前端控制器（DispatchServlet），负责调度各个组件的处理，处理流程如下：



SpringMVC的三大核心组件，以及九大组件说明

组件	说明
HandlerMapping (处理器映射器，核心)	用来查找 Handler，表现形式可以是类，也 口, 但只能实现单业务, 不推荐)。 标注了@RequestMapping的每个方法都可 HandlerMapping作用是找到拦截器链(Inter
HandlerAdapter (处理器适配器，核心)	Handler的形参和返回是任意形式，所以需 因为DispatherServlet的方法只有一个 doSe
ViewResolver (视图解析器，核心)	将String类型的视图名和Locale解析为View resolveViewName()方法。 作用是拼接物理视图路径(jsp等)，并将返回 第一：找到物理位置的模版，比如jsp，free 第二：找到视图类型，是jsp还是ftl渲染引擎 默认实现了InternalResourceViewResolver
HandlerExceptionResolver	用于处理 Handler 产生的异常情况。作用是 之后交给渲染方法进行渲染成页面。
MultipartResolver	用于上传请求，通过将普通的请求包装成 M 现。MultipartHttpServletRequest 可以通过 果上传多个文件，还 可以调用 getFileMap 这样的结构，MultipartResolver 的作用就是 上传的功能。
RequestToViewNameTranslator	作用是从请求中获取 ViewName。ViewRes View，如果Handler并没有设置View或View 径为请求路径。 比如请求/a/b/c，没有设置View的逻辑路径
LocaleResolver	ViewResolver 组件的 resolveViewName 方 名，一个是 Locale。 LocaleResolver 用于从请求中解析出 Loca 用来表示一个区域。这 个组件也是 i18n 的
ThemeResolver	用来解析主题。主题是样式、图片及它们所 MVC 中一套主题对应一个 properties文件， 有资源，如图 片、CSS样式等。创建主题主 新建一个“主题名.properties”并将资 源设置 与主题相关的类有 ThemeResolver、Them ThemeResolver负责从请求中解析出主题名 ThemeSource根据主题名找到具体的主题， Theme来获取主题和 具体的资源。
FlashMapManager	用于重定向时的参数传递。 可以通过ServletRequestAttributes.getReq 其属性**OUTPUT_FLASH_MAP_ATTRIBU Handler中Spring就会自动将其设置到Mode

具体处理流程如下：



2、SpringMVC的配置

（1）、web.xml中的配置：

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>

  <!-- 设置form  post请求form中数据有中文时提交到服务端乱码问题 -->
  <filter>
    <filter-name>encodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
      <param-name>encoding</param-name>
      <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
      <param-name>forceEncoding</param-name>
      <param-value>true</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <!--配置springmvc请求方式转换过滤器，会检查请求参数中是否有_method参数，如果有就按照指定的请求
  <filter>
```

```
    <filter-name>hiddenHttpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>hiddenHttpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

<!--启动springmvc-->
  <servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:springmvc.xml</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>

    <!--
      方式一：带后缀，比如*.action  *.do *.aaa
              该种方式比较精确、方便，在以前和现在企业中都有很大的使用比例
      方式二：/ 不会拦截 .jsp，但是会拦截.html等静态资源（静态资源：除了servlet和jsp之外的js、c
            为什么配置为/ 会拦截静态资源？？？
                因为tomcat容器中有一个web.xml（父），你的项目中也有一个web.xml（子），是一个继
                      父web.xml中有一个DefaultServlet， url-pattern 是一个 /
                      此时我们自己的web.xml中也配置了一个 / ,覆写了父web.xml的配置
            为什么不拦截.jsp呢？
                因为父web.xml中有一个JspServlet，这个servlet拦截.jsp文件，而我们并没有覆写这个
                所以springmvc此时不拦截jsp，jsp的处理交给了tomcat

            如何解决/拦截静态资源这件事？
                通过springmvc.xml中配置静态资源的两种方式：
                <!--静态资源配置，方案一-->
                原理：添加该标签配置之后，会在SpringMVC上下文中定义一个DefaultServletHttpRequ
                 这个对象如同一个检查人员，对进入DispatcherServlet的url请求进行过滤筛查，如果发
                 那么会把请求转由web应用服务器（tomcat）默认的DefaultServlet来处理，如果不是静
                 SpringMVC框架处理
                <!--<mvc:default-servlet-handler/>-->

                <!--静态资源配置，方案二，SpringMVC框架自己处理静态资源
                mapping:约定的静态资源的url规则
                location：指定的静态资源的存放位置
                -->
```

```
74                <mvc:resources location="classpath:/"  mapping="/resources/**"/>
75                <mvc:resources location="/WEB-INF/js/" mapping="/js/**"/>
76
77        方式三：/* 拦截所有，包括.jsp
78     -->
79     <!--拦截匹配规则的url请求，进入springmvc框架处理-->
80     <url-pattern>/</url-pattern>
81   </servlet-mapping>
82 </web-app>
```

如下配置是maven自动生成的web.xml头，会导致JSP中使用el表达式无法获取到ModelAndView中的数据，

```
1 <!DOCTYPE web-app PUBLIC
2 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
3 "http://java.sun.com/dtd/web-app_2_3.dtd" >
4
5 <web-app>
6
7 </web-app>
```

需要使用如下配置才可以：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <web-app version="2.5"
4 xmlns="http://java.sun.com/xml/ns/javaee"
5 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
7 http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
8
9 </web-app>
```

（2）、springmvc.xml的配置：

大致分为如下几类：

- 1.开启controller层注解扫描。注意只扫描controller层。
- 2.配置视图解析器。内置实现InternalResourceViewResolver支持jsp，若要支持freemaker，velocity，Thymeleaf(推荐) 则需要可插　　拔的引入相关解析器即可。
  可配置的参数就是前缀，后缀；从逻辑视图地址来拼接为物理视图地址。
- 3.配置静态资源。
- 4.配置拦截器。<mvc:interceptors>，和filter类似；
- 5.配置multipartResolver多元素解析器。注意id固定，用于上传(支持多文件)，spring有内置，当然也有第三方插件。
- 6.处理器，映射器的自动最优注册。本质是handler的形参，返回值的一些处理，比如枚举和日期格式化处理。

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

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!-- 注意xsd需要开启mvc的支持 -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         https://www.springframework.org/schema/context/spring-context.xsd
         http://www.springframework.org/schema/mvc
         https://www.springframework.org/schema/mvc/spring-mvc.xsd
">
    <!--开启controller扫描-->
    <context:component-scan base-package="com.zyc.controller"/>

    <!--配置springmvc的视图解析器-->
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

    <!--
        自动注册最合适的处理器映射器，处理器适配器(调用handler方法)
    -->
    <mvc:annotation-driven conversion-service="conversionServiceBean"/>

    <!--注册自定义类型转换器-->
    <bean id="conversionServiceBean" class="org.springframework.format.support.Formatting
        <property name="converters">
            <set>
                <bean class="com.zyc.converter.DateConverter"></bean>
            </set>
        </property>
    </bean>

    <!--静态资源配置，方案一-->
    <!--
        原理：添加该标签配置之后，会在SpringMVC上下文中定义一个DefaultServletHttpRequestHandl
              这个对象如同一个检查人员，对进入DispatcherServlet的url请求进行过滤筛查，如果发现是
              那么会把请求转由web应用服务器（tomcat）默认的DefaultServlet来处理，如果不是静态资
              SpringMVC框架处理
    -->
    <!--<mvc:default-servlet-handler/>-->

```

```
    <!--静态资源配置，方案二，SpringMVC框架自己处理静态资源
        mapping:约定的静态资源的url规则
        location：指定的静态资源的存放位置
    -->
    <mvc:resources location="classpath:/"  mapping="/resources/**"/>
    <mvc:resources location="/WEB-INF/js/" mapping="/js/**"/>

    <mvc:interceptors>
        <!--拦截所有handler-->
        <!--<bean class="com.zyc.interceptor.LoginInterceptor"/>-->

        <mvc:interceptor>
            <!--配置当前拦截器的url拦截规则，**代表当前目录下及其子目录下的所有url-->
            <mvc:mapping path="/**"/>
            <!--exclude-mapping可以在mapping的基础上排除一些url拦截-->
            <!--<mvc:exclude-mapping path="/demo/**"/>-->
            <bean class="com.zyc.interceptor.LoginInterceptor"/>
        </mvc:interceptor>
        <mvc:interceptor>
            <mvc:mapping path="/**"/>
            <bean class="com.zyc.interceptor.MyIntercepter02"/>
        </mvc:interceptor>
    </mvc:interceptors>

    <!--多元素解析器
        id固定为multipartResolver
    -->
    <bean id="multipartResolver" class="org.springframework.web.multipart.commons.Commons
        <!--设置上传文件大小上限，单位是字节，-1代表没有限制也是默认的-->
        <property name="maxUploadSize" value="5000000"/>
    </bean>
</beans>
```

(3)、静态资源的处理

针对静态资源拦截的情况，介绍以下三种方式：

首先需要知道各个容器的默认Servlet：

容器	默认servlet名字
Tomcat, Jetty, JBoss, GlassFish	default
Google App Engine	_ah_default
Resin	resin-file
WebLogic	FileServlet
WebSphere	SimpleFileServlet

方式一：交由外部容器处理(如tomcat)

上面说过/会覆盖tomcat容器的defaultServlet导致静态资源被springmvc拦截，那么可以将<servlet-mapping>写在 **DispatcherServlet **的前面。

```
<!-- 注意写在 DispatcherServlet 前面，比如让tomcat的 defaultServlet 优先拦截-->
<servlet-mapping>
    <!-- 注意servlet名字需要和容器的匹配，参照上面列表！！！ -->
    <servlet-name>default</servlet-name>
    <url-pattern>*.jpg</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.js</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>*.css</url-pattern>
</servlet-mapping>
```

方式二：mvc:default-servlet-handler/ 先由springmvc筛查，静态资源交由外部容器

DispatcherServlet通过内部对象 DefaultServletHttpRequestHandler 对url请求进行过滤筛查，是静态资源就交由外部容器处理，如果非静态就由springmvc自己处理。

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

方式三：spring3.0.4以后版本提供了mvc:resources由springmvc自己处理，而且静态资源位置可以任意指定

```
<!--
 mapping: 代表url拦截规则，只要是/images 或 /js 等路由，则会去location对应位置找资源
 location: 代表资源位置，可以是webapp下，即 /; 也可以是classpath下，甚至可以在src包下

 注意：如果使用了springmvc内部处理的静态资源的方式，那么html引用静态资源就必须使用 src="/js/..."
      (默认只从webapp下加载资源文件，WEB-INF/下是不支持加载的)
-->
<mvc:resources location="/images/,/WEB-INF/images/,classpath:/images/" mapping="/images/**
<mvc:resources location="/js/,/WEB-INF/js/,classpath:/js/" mapping="/js/**" />
```

（4）、乱码问题

GET乱码：需要在Tomcat等web容器里设置server.xml的配置

```
<Connector URIEncoding="utf-8" connectionTimeout="20000" port="8080"
protocol="HTTP/1.1" redirectPort="8443"/>
```

POST乱码：web.xml中如下配置

```
<filter>
  <filter-name>encodingFilter</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
  <init-param>
    <param-name>forceEncoding</param-name>
     <param-value>true</param-value>
   </init-param>
</filter>
<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

（5）、form提交请求方法的配置

form中method并没有提供 PUT 和 DELETE，只能写GET和POST，springmvc提供了 HiddenHttpMethodFilter 隐藏域请求方式转换过滤器。会检查请求参数中是否有_method参数，如果有就按照指定的请求方式进行转换。

前端：

```
<form method="post" action="/demo/handle/15/lisi">
  <input type="hidden" name="_method" value="put"/>
  <input type="submit" value="提交rest_put请求"/>
</form>

<form method="post" action="/demo/handle/15">
  <input type="hidden" name="_method" value="delete"/>
  <input type="submit" value="提交rest_delete请求"/>
</form>
```

web.xml:

```
<!--配置springmvc请求方式转换过滤器，会检查请求参数中是否有_method参数，如果有就按照指定的请求方
<filter>
  <filter-name>hiddenHttpMethodFilter</filter-name>
  <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>
```

```
<filter-mapping>
  <filter-name>hiddenHttpMethodFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

(6)、日期Date类型转换器

Date是有多种格式化方式的，springmvc并不知道你需要按哪种方式转换，所以提供了一个转换器接口。

实现接口后需要注册到**处理器适配器**

第一种是全局处理:

springmvc.xml:

```
<!--注册自定义类型转换器-->
<bean id="conversionServiceBean" class="org.springframework.format.support.FormattingConve
  <property name="converters">
    <!-- 这里是一个Set集合，意味着你可以定义很多转换器 -->
    <set>
      <bean class="com.zyc.converter.DateConverter"></bean>
    </set>
  </property>
</bean>
```

DateConverter:

```
/**
 * 自定义类型转换器
 * S：source，源类型
 * T：target：目标类型
 */
public class DateConverter implements Converter<String, Date> {
    @Override
    public Date convert(String source) {
        // 完成字符串向日期的转换
        SimpleDateFormat sdf1 = new SimpleDateFormat("yyyy-MM-dd");
        SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy/MM/dd");

        try {
            return sdf1.parse(source);
        } catch (ParseException e) {
           try{
                   return sdf2.parse(source);
            }catch(ParseException e1){
              e1.printStackTrace();
            }
        }
        return null;
    }
}
```

第二种是Controller单独局部设置，即通过@InitBinder

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    dateFormat.setLenient(false);
    binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, true));    //t
}
```

第三种是实体类中加日期格式化注解

```
@DateTimeFormat(pattern = "yyyy-MM-dd")
private Date createTime;
getter方法指定@JsonFormat格式
@DateTimeFormat(pattern="yyyy-MM-dd HH:mm:ss")
@JsonFormat(pattern="yyyy-MM-dd HH:mm:ss",timezone = "GMT+8")
public Date getCreateTime() {
    return this.createTime;
}
```

第四种是通过MappingJackson2HttpMessageConverter，即springmvc对jackson的JSON格式支持，也可重写objectMapper 类进行自定义实现

配置如下:

```
<mvc:annotation-driven>
  <mvc:message-converters>
    <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConvert
      <property name="objectMapper">
        <bean class="com.fasterxml.jackson.databind.ObjectMapper">
          <!-- 处理responseBody 里面日期类型 -->
          <!--   <property name="dateFormat">
                              <bean class="java.text.SimpleDateFormat">
                                  <constructor-arg type="java.lang.String" value="yyyy-MM-
                               </bean>
                          </property>  -->
           <!-- 为null字段时不显示 -->
           <property name="serializationInclusion">
             <value type="com.fasterxml.jackson.annotation.JsonInclude.Include">NON_NULL</
           </property>
           <property name="propertyNamingStrategy">
             <!--<bean class="com.xxx.serializer.MyPropertyNamingStrategyBase" />-->
             <bean class="com.fasterxml.jackson.databind.PropertyNamingStrategy.LowerCaseW
           </property>
         </bean>
       </property>
       <property name="supportedMediaTypes">
         <list>
           <value>text/html;charset=UTF-8</value>
           <value>application/json; charset=UTF-8</value>
```

```
        </list>
      </property>
    </bean>
  </mvc:message-converters>
</mvc:annotation-driven>
```

（7）、自定义参数解析器

自定义参数解析 **HandlerMethodArgumentResolver**

springmvc内置了很多解析起器接口，比如参数对象的解析完全可自定义解析器。

```
/**
 * 用于获取当前登陆人信息的注解,配合自定义的参数处理器使用
 */
@Target({ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface CurUser {
}

// 待封装的Vo
@Getter
@Setter
@ToString
public class CurUserVo {
    private Long id;
    private String name;
}
实现参数解析器并注册
public class CurUserArgumentResolver implements HandlerMethodArgumentResolver {

    // 只有标注有CurUser注解，并且数据类型是CurrUserVo/Map/Object的才给与处理
    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        CurrUser ann = parameter.getParameterAnnotation(CurrUser.class);
        Class<?> parameterType = parameter.getParameterType();
        return (ann != null &&
                (CurUserVo.class.isAssignableFrom(parameterType)
                        || Map.class.isAssignableFrom(parameterType)
                        || Object.class.isAssignableFrom(parameterType)));
    }

    @Override
    public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer contai
        HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.class
        // 从请求头中拿到token
        String token = request.getHeader("Authorization");
```

```
        if (StringUtils.isEmpty(token)) {
            return null; // 此处不建议做异常处理，因为校验token的事不应该属于它来做
        }

        // 此处作为测试：new一个处理（写死）
        CurrUserVo userVo = new CurrUserVo();
        userVo.setId(1L);
        userVo.setName("fsx");

        // 判断参数类型进行返回
        Class<?> parameterType = parameter.getParameterType();
        if (Map.class.isAssignableFrom(parameterType)) {
            Map<String, Object> map = new HashMap<>();
            BeanUtils.copyProperties(userVo, map);
            return map;
        } else {
            return userVo;
        }

    }
}

// 注册进springmvc组件内
@Configuration
@EnableWebMvc
public class WebMvcConfig extends WebMvcConfigurerAdapter {

    @Bean
     public CurrUserArgumentResolver currUserArgumentResolver(){
         return new CurrUserArgumentResolver();
     }

     @Override
     public void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolver
         argumentResolvers.add(new CurUserArgumentResolver());
     }
}
```

Controller中的使用：

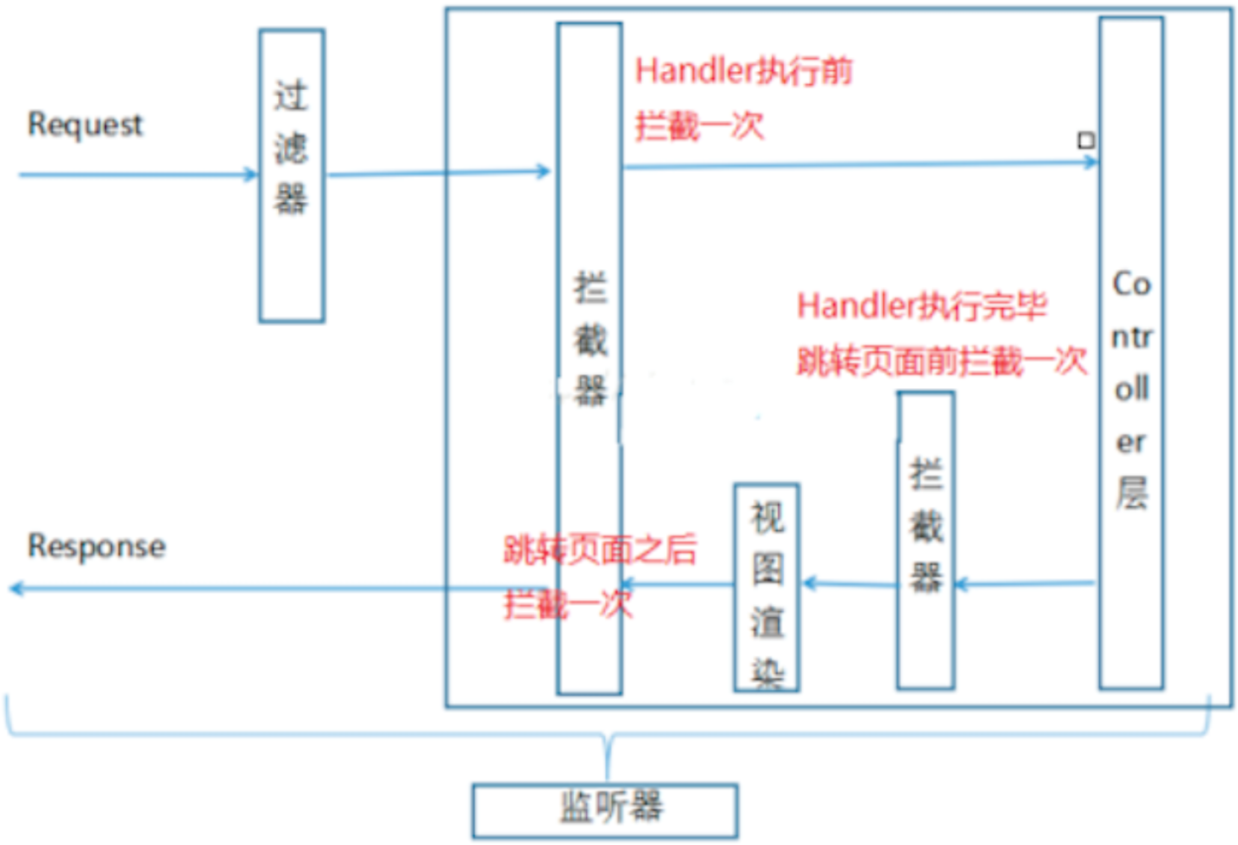
```
@ResponseBody
@GetMapping("/test/curruser")
public Object testCurrUser(@CurUser CurrUserVo currUser) {
  return currUser;
}
@ResponseBody
```

```
@GetMapping("/test/curruser/map")
public Object testCurrUserMap(@CurUser Map<String,Object> currUser) {
  return currUser;
}
@ResponseBody
@GetMapping("/test/curruser/object")
public Object testCurrUserObject(@CurUser Object currUser) {
  return currUser;
}
```

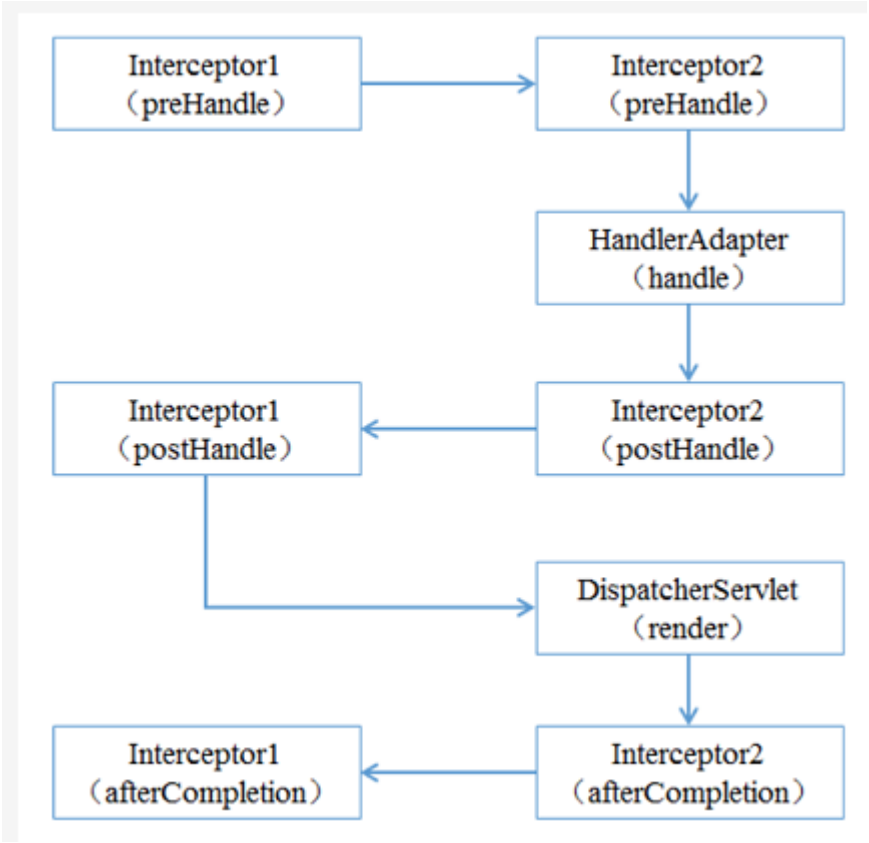
3、拦截器

拦截器使用JDK动态代理实现了AOP的动态增强，拦截器需要实现HandlerInteceptor接口

拦截器拦截时机如下：



多个拦截器的执行顺序如下：


Interceptor1
（preHandle）
Interceptor2
（preHandle）
HandlerAdapter
（handle）
Interceptor1
（postHandle）
Interceptor2
（postHandle）
DispatcherServlet
（render）
Interceptor1
（afterCompletion）
Interceptor2
（afterCompletion）