Part 2：核心流程设计分析

在Part 1中，我主要是对**Spring MVC核心组件的功能**进行了介绍，并且从“流程化”的角度介绍了一下**Spring MVC的核心流程和核心功能**，并**以DispatcherServlet类为代表进行了功能分析。**

在Part 2中，我将会对Spring MVC**核心流程的设计进行详细的分析介绍**，并分析**类的设计以及类之间的具体关系和交互**，并以此为基础分析**面向对象的具体应用。**

Part 2目录：

一、从不同的HttpMethod的请求处理说起

由Part 1，我们从Spring MVC的核心工作流程中可知客户端向浏览器发送的请求会首先被DispatcherServlet捕获；实际上，**service()方法是Serlvet框架的核心**，用于应答浏览器的请求，而这个方法是在DispatcherServlet的父类FrameworkServlet中实现的；除此之外，不同HttpMethod的请求处理方法doxxx也是在FrameworkServlet中实现的，故在DispatcherServlet的继承关系中，**FrameworkServlet才是真正的请求处理的入口**；

在Part 1中，我们只是从调用关系的角度简要的分析了一下请求处理所用到的函数，在这一Part中，我们将会详细的分析关键的请求处理函数的实现，并在这个过程中对Spring MVC的核心流程和核心组件的类间交互关系进行建模。

进入“Service”阶段后，每一次Http请求到来，容器都会启动一个请求线程，通过service方法，委派doxxxx方法完成请求的处理，下面我们就从Servlet的核心service方法开始对Spring MVC的核心流程进行源码分析；

（一）Servlet的核心：service()方法

在FrameworkServlet中实现，service()方法为传入的请求提供响应，它由容器中的每个请求来调用：

```
@Override
protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
    //(1)获得请求方法
    HttpMethod httpMethod = HttpMethod.resolve(request.getMethod());
    //(2.1)处理PATCH请求
    if (httpMethod == HttpMethod.PATCH || httpMethod == null) {
        processRequest(request, response);
    }
    //(2.2)处理其他请求
    else {
        super.service(request, response);
    }
}
```

对应图上标号进行源码分析(后面的源码分析也是同样的形式)：

(1)获得请求方法；

(2.1)如果请求方法是PATCH，则调用processRequest()方法进行处理；

(2.2)其他请求的处理方法则要调用FrameworkServlet的父类HttpServlet中的service函数进行实现，而父类HttpServlet中的service()方法会调用doxxx方法进行请求处理；

！！面向对象的思想：

这里体现了由Servlet发展到DispatcherServlet中 框架 的作用，直接在servlet-class标签中写上DispatcherServlet即可，不需要自己再重写HttpServlet中的方法，即把一些通用的操作封装起来不需要开发人员自己再去实现。

（二）FrameworkServlet中doxxx方法的重写

1、doGet/doPost/doPut/doDelete

doGet：获取被请求URI指定的信息；

doPost：请求源服务器接受请求中的实体作为请求资源的一个新的从属物；

doPut：请求源服务器把请求中的实体存储在请求URI标识下；

doDelete：请求源服务器删除请求URI指定的资源；

这四种方法均是直接调用processRequest()方法进行请求处理，这里不再粘贴代码，processRequest方法之后会详细讲解；其中doPost和doGet方法是最常用的方法；

2、 doOptions

<1>功能作用

Options方法的主要用途有两个，一是获取服务器支持的HTTP请求方法，二是可以用来检查服务器的性能：

```
@Override
protected void doOptions(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
    //(1)判断是否处理请求
    if (this.dispatchOptionsRequest || CorsUtils.isPreFlightRequest(request)) {
        //(2)处理请求
        processRequest(request, response);
        //(3)判断
        if (response.containsHeader("Allow")) {
            return;
        }
    }

    //(4)调用父类方法
    super.doOptions(request, new HttpServletResponseWrapper(response) {
        @Override
        public void setHeader(String name, String value) {
            if ("Allow".equals(name)) {
                value = (StringUtils.hasLength(value) ? value + ", " : "") + HttpMethod.PATCH.name();
            }
            super.setHeader(name, value);
        }
    });
}
```

<2>源码分析

(1)dispatchOptionsRequest初始化为false，此时如果dispatchOptionsRequest为true，则开始处理请求；

(2)调用processRequest方法处理请求；

(3)如果响应Header中包含“Allow”，则不需要调用父类方法，直接返回；

(4)调用父类方法，在响应Header的"Allow"增加PATCH的值；

3、doTrace

doTrace方法用于激发一个远程的、应用层的请求消息回路，即一个请求响应回路；

此方法的实现思路与doOptions方法大体相似，核心均是调用processRequest方法进行请求处理；

（三）processRequest方法

1、源码分析

通过上面对于请求处理的分析我们发现不论是service方法还是doxxx方法，均需要调用FrameworkServlet中的processRequest函数进行请求处理，下面我们就来对该函数进行源码分析：

```
protected final void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
    //(1)获取时间戳
    long startTime = System.currentTimeMillis();
    //(2)申明异常
    Throwable failureCause = null;

    //(3)本地化处理
    LocaleContext previousLocaleContext = LocaleContextHolder.getLocaleContext();
    LocaleContext localeContext = buildLocaleContext(request);

    //(4)获取RequestAttributes
    RequestAttributes previousAttributes = RequestContextHolder.getRequestAttributes();
    //(5)构建ServletRequestAttributes
    ServletRequestAttributes requestAttributes = buildRequestAttributes(request, response, previousAttributes);

                                                    ……

    //(6)将localeContext和requestAttributes放入当前线程中
    initContextHolders(request, localeContext, requestAttributes);

    try {
        //!!(7)进入doService方法，执行真正的逻辑
        doService(request, response);
    }
    catch (ServletException | IOException ex) {
        failureCause = ex;//(8)
        throw ex;
    }
    catch (Throwable ex) {
        failureCause = ex;//(8)
        throw new NestedServletException("Request processing failed", ex);
    }

    finally {
        resetContextHolders(request, previousLocaleContext, previousAttributes);
        if (requestAttributes != null) {
            requestAttributes.requestCompleted();
        }
        //(9)打印请求日志
        logResult(request, response, failureCause, asyncManager);
        //(10)发布事件
        publishRequestHandledEvent(request, response, startTime, failureCause);
    }
```

```
45 }
```

(1)获取系统当前的时间戳，用于计算web请求的处理时间；

(2)记录一个异常顶层对象Throwable；

(3)进行本土化处理，在这里需要获取上一个LocaleContext并且构建一个本土的上下文；

(4)(5)获取当前绑定到线程的RequestAttributes并构建ServletRequestAttributes；

(6)将localeContext和requestAttributes放入当前线程中

(7)进入doService方法，这是真正处理请求的核心，它在DispatcherServlet中实现，在下一个模块中我们会单独拿出来分析；

(8)记录抛出的异常，在finally段代码中使用；

(9)调用logResult方法，打印请求日志，并将日志级别设为DEBUG；

(10)调用publishRequestHandledEvent方法，发布org.springframework.web.context.support.ServletRequestHandledEvent事件；

2、！！功能作用以及及面向对象思想的理解要点

processRequest方法理解的要点是(面向对象的思想)：以***doService()***方法为间隔，前边的6步是将当前请求的Locale对象和属性，分别设置到LocaleContextHolder和RequestContextHolder这两个抽象类的ThreadLocal中，即将Locale队形和属性与请求线程做了绑定；在doService()处理结束之后，后面用于恢复请求前的LocaleContextHolder和RequestContextHolder，即解除线程绑定。而每次请求处理结束后，容器的上下文都会发布一个ServletRequestHandledEvent时间，可以用监听器来监听该事件。

processRequest功能作用：做线程的安全隔离，将请求处理的核心转移到doService()方法中。

（四）doService

上面我们所分析过的流程都是在父类FraneworkServlet中实现的，而FrameworkServlet主要负责的是准备工作，doService方法就是子类DispatcherServlet处理请求的入口方法，在DispatcherServlet中实现；

！！面向对象思想：

这也是一个抽象方法：**父类抽象处理流程，子类给予具体的实现。**

下面我们正式来看doService方法：

```
1  @Override
2      protected void doService(HttpServletRequest request, HttpServletResponse
   response) throws Exception {
3          //调用logRequest方法，打印请求日志，将日志级别设为DEBUG
4          logRequest(request);
5                                      ……
6      //设置Spring框架中的常用对象到request属性中，使框架队形对处理程序和视图对象可用
7      request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE,
   getWebApplicationContext());
8      request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);
9      request.setAttribute(THEME_RESOLVER_ATTRIBUTE, this.themeResolver);
10     request.setAttribute(THEME_SOURCE_ATTRIBUTE, getThemeSource());
```

```
                        ……//(这部分主要是转发保存数据)。
    RequestPath previousRequestPath = null;
    if (this.parseRequestPath) {
        previousRequestPath = (RequestPath)
request.getAttribute(ServletRequestPathUtils.PATH_ATTRIBUTE);
        ServletRequestPathUtils.parseAndCache(request);
    }

    try {
        //!!调用doDispatch执行请求的分发
        doDispatch(request, response);
    }
    finally {
                                        ……
}
```

Spring MVC不将请求处理过程与Web容器完全隔离，这里的request.setAttribute方法的调用将实例化的对象设置到http请求的属性中以供下一步使用，比如容器的上下文对象、本地化解析器、主题解析器等编程元素。

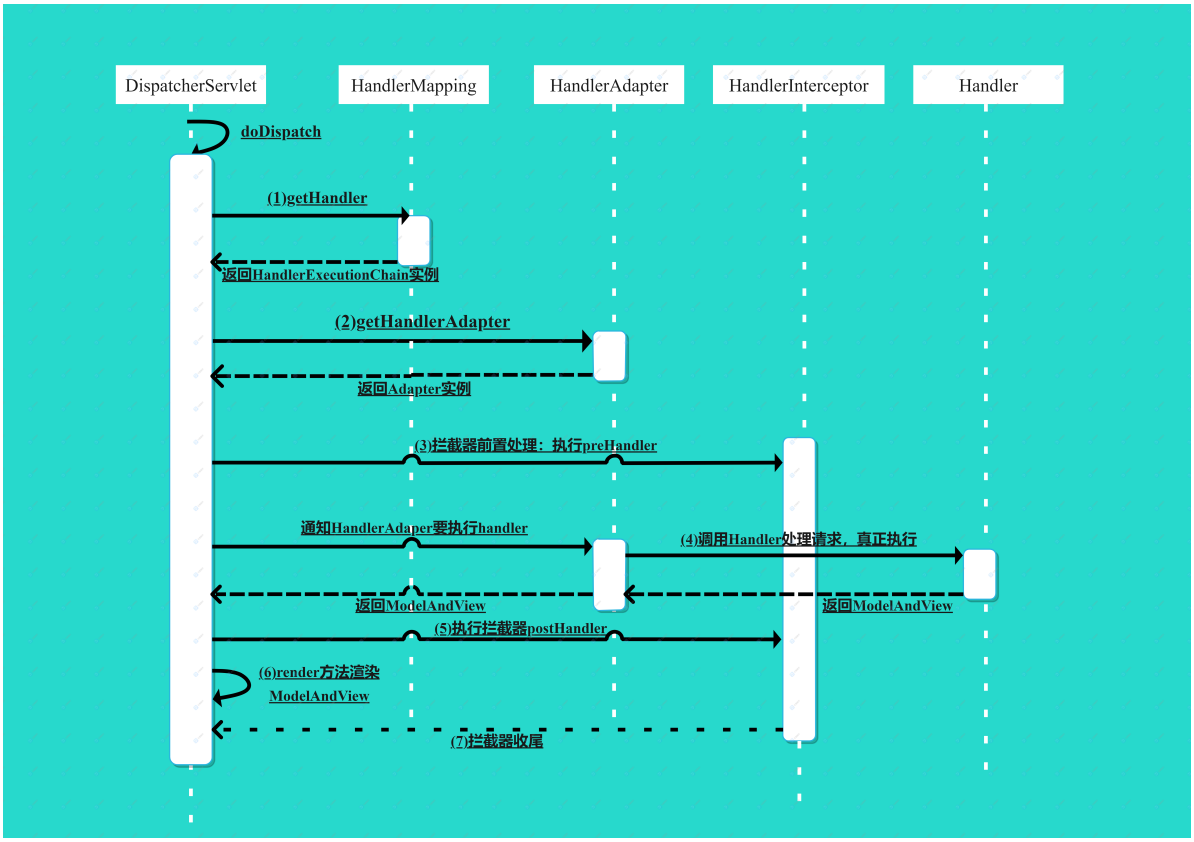
（五）最重要！！！：doDispatch

doDispatch方法用于执行请求的分发，也是实际处理请求的方法，从这里开始，我们也就真正进入到了Spring MVC框架中的最核心的流程部分，在这部分中，我们会涉及到核心流程中的类间关系以及核心组件内部类间关系的建模，于是我们开启一个新的篇章。

二、最重要！！！：doDispatch：进入最核心的流程

（一）doDispatch的时序分析图

首先放一下doDispatch的时序分析图，下面结合这个时序分析图进行核心流程的分析更加有利于理解：



（二）doDispatch的整体流程概览

1、！！面向对象思想

doDispatch()方法，是SpringMVC整个框架的精华所在。它通过高度抽象的接口，描述出了一个MVC（Model-View-Controller）设计模式的实现方案。***Model***、***View***、***Controller***三种层次的编程元素，在SpringMVC中都有大量的实现类，各种处理细节也是千差万别。但是，它们最后都是由doDispatch()方法来统一描述，这就是**接口和抽象**的威力;

2、doDispatch方法本身的源码分析

doDispath是实际处理请求的方法，我们通过源代码来详细分析一下其核心流程：

首先我们可以看到**传入doDispatch的参数是HttpServletRequest和HttpServletResponse对象**，即从request中能获取到一切可以请求的数据，而从response中我们又可以往服务器端输出任何响应，于是Http的请求处理就应该围绕这两个对象来进行设计。

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    try {
        ModelAndView mv = null;
        Exception dispatchException = null;

```

```
        try {
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

            //! (1)获得对应的HandlerExecutionChain对象
            mappedHandler = getHandler(processedRequest);
            if (mappedHandler == null) {
                noHandlerFound(processedRequest, response);
                return;
            }

            //! (2)得到与Handler匹配的HandlerAdapter
            HandlerAdapter ha =
getHandlerAdapter(mappedHandler.getHandler());

            String method = request.getMethod();
            boolean isGet = HttpMethod.GET.matches(method);
            if (isGet || HttpMethod.HEAD.matches(method)) {
                long lastModified = ha.getLastModified(request,
mappedHandler.getHandler());
                if (new ServletWebRequest(request,
response).checkNotModified(lastModified) && isGet) {
                    return;
                }
            }

            //! (3)执行所有注册拦截器的preHandler方法
            if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                return;
            }

            //! (4)这里开始才是Handler方法的真正执行，返回视图
            mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());

            if (asyncManager.isConcurrentHandlingStarted()) {
                return;
            }

            applyDefaultViewName(processedRequest, mv);
            //! (5)执行所有注册拦截器的postHandle方法
            mappedHandler.applyPostHandle(processedRequest, response, mv);
        }
        catch (Exception ex) {
            dispatchException = ex;
        }
        catch (Throwable err) {

            dispatchException = new NestedServletException("Handler dispatch
failed", err);
        }
        //! (6)处理视图，进行页面渲染
        processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);
    }
    catch (Exception ex) {
        //! (7)拦截器收尾
```

```
        triggerAfterCompletion(processedRequest, response, mappedHandler,
ex);
    }
    catch (Throwable err) {
        triggerAfterCompletion(processedRequest, response, mappedHandler,
                new NestedServletException("Handler processing failed",
err));
    }
                //后面的代表就不是核心流程的重点了，这里不予贴出
                                        ……
```

下面我们结合核心组件来分析核心流程：

（三）doDispatch结合组件的具体核心流程分析(标号对应源码中注释的标号)

！(1) 调用getHandler方法获得HandlerExecutionChain对象

通过HandlerMapping将请求映射到处理器，调用getHandler方法，获得对应的HandlerExecutionChain对象(处理器执行链），它包含真正的handler对象和Interceptors;

<1>DispatcherServket中的getHandler方法

在DispatcherServket类中找到getHandler方法，根据request获取handler：

```
protected HandlerExecutionChain getHandler(HttpServletRequest request)
throws Exception {
        if (this.handlerMappings != null) {
            //循环遍历HandlerMapping数组
            for (HandlerMapping mapping : this.handlerMappings) {
                //获得请求对应的HandlerExecutionChain对象
                HandlerExecutionChain handler = mapping.getHandler(request);
                if (handler != null) {
                    return handler;
                }
            }
        }
        return null;
    }
```

<2>hm.getHandler：

HandlerMapping是一个映射接口，我们在HandlerMapping中找到getHandler的实现：(只将流程中关键流程部分的代码摘录出来)

```
public final HandlerExecutionChain getHandler(HttpServletRequest request)
throws Exception {
    //<2.1>得到相应的handler
    Object handler = getHandlerInternal(request);
    if (handler == null) {
        handler = getDefaultHandler();
    }
    if (handler == null) {
        return null;
    }
```

```
    if (handler instanceof String handlerName) {
        handler = obtainApplicationContext().getBean(handlerName);
    }
                                                    ……
    //<2.2>实例化HandlerExecutionChain对象并得到Interceptors
    HandlerExecutionChain executionChain = getHandlerExecutionChain(handler,
request);
                                                    ……
    return executionChain;
}
```

<2.1>this.getHandlerInternal：

根据request的url得到相应的handler，此handler为HandlerMethod的实例，这一步的关键函数为getHandlerInternal，getHandlerInternal函数就是要在Map中寻找HandlerMethod方法，而Map中保存的是<url,HhandlerMethod>，在容器初始化的时候会建立所有url和controller的对应关系保存到Map，我们来看一下getHandlerInternal函数的关键部分：

```
protected HandlerMethod getHandlerInternal(HttpServletRequest request)
throws Exception {
        String lookupPath = initLookupPath(request);
        this.mappingRegistry.acquireReadLock();
        try {
            //!!在Map中寻找handlerMethod
            HandlerMethod handlerMethod = lookupHandlerMethod(lookupPath,
request);
            return (handlerMethod != null ?
handlerMethod.createWithResolvedBean() : null);
        }
        finally {
            this.mappingRegistry.releaseReadLock();
        }
    }
```

<2.2>实例化HandlerExecutionChain对象并得到Interceptors

根据第一步的HandlerMethod参数，利用getHandlerExecutionChain函数实例化一个HandlerExecutionChain对象，得到Interceptors，这个函数主要就是用于获得拦截器，这里我们不展开分析。

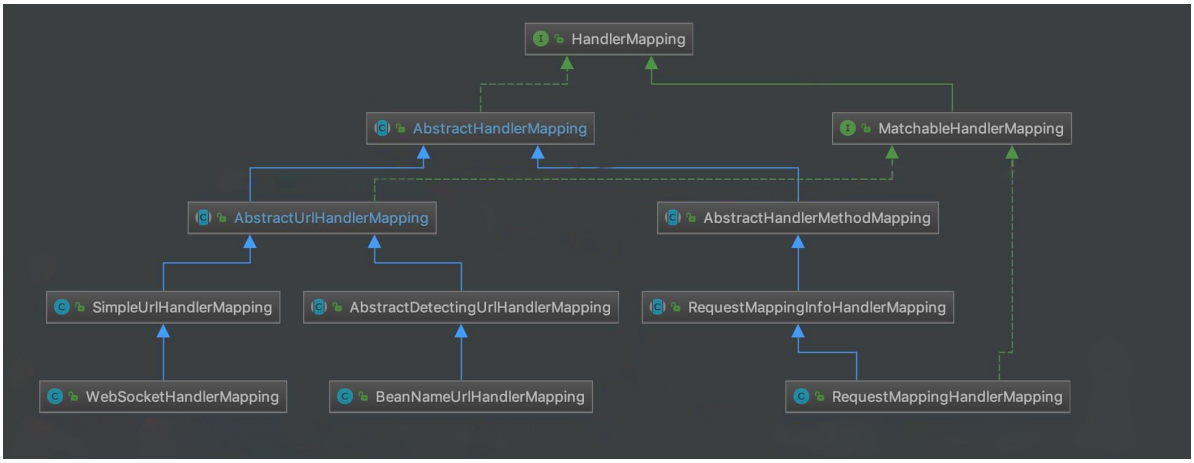
<3> 有关HandlerMapping组件

A. 组件功能

HandlerMapping组件是处理器匹配接口，**它根据请求获得请求对应的处理器(Handler)和拦截器们(Interceptors)**，HandlerExecutionChain是其返回对象，包含了上面处理器和拦截器们。

B. 组件类图

下面是HandlerMapping的整体类图：



C. 组件子类分析

首先是**AbstractHandlerMapping抽象子类**，其实现了获得请求对应的处理器和拦截器们的骨架逻辑，即实现了HandlerMapping接口中的getHandler()方法，而具体的抽象方法getHandlerInternal暴露，交由子类去实现，这里也就涉及到了**面向对象的高级设计意图"模板方法模式"**，我们将会在Part 3中进行解析；

AbstractHandlerMapping的子类分为2类：

I. *AbstractUrlHandlerMapping*：基于Url进行匹配；

II. *AbstractHandlerMethodMapping*：基于Method进行匹配，如RequestMapping的方式；

然后是**MatchableHandlerMethodMapping子类**，是判断请求和指定pattern路径是否匹配的接口方法；

！(2)获得HandlerAdapter对象

调用getHandlerAdapter方法，获得(1)中获得的handler对应的HandlerAdapter对象

<1>在DispatcherServlet类中找到getHandlerAdapter方法：

```
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
        if (this.handlerAdapters != null) {
            //遍历HandlerAdapter数组
            for (HandlerAdapter adapter : this.handlerAdapters) {
                //判断是否支持当前处理器
                if (adapter.supports(handler)) {
                    return adapter;
                }
            }
        }
        //如果没找到对应的HandlerAdapter对象，发出ServletException异常
        throw new ServletException("No adapter for handler [" + handler +
                "]: The DispatcherServlet configuration needs to include a HandlerAdapter that supports this handler");
    }
```

getHandlerAdapter函数能够判断数组HandlerAdapter中哪个Adapter能与这个Handler匹配；

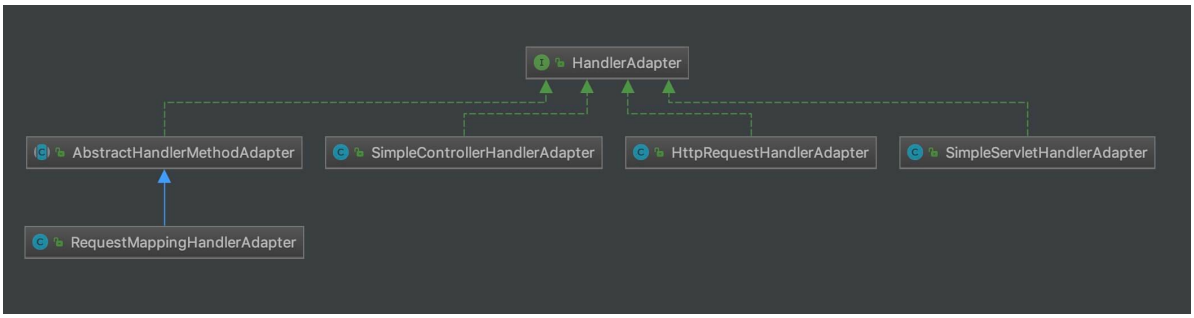
<2> 有关HandlerAdapter组件

A. 组件的功能

HandlerAdapter组件是处理器适配接口，处理器Handler是Object类型，需要有一个调用者来实现handler的执行，这也就引出了HandlerAdapter组件。

B. 组件类图

下面是HandlerAdapter的类图：



C. 组件子类分析

由于我们的重点在于分析Spring MVC的核心流程，所以这里我们不展开分析HanlerAdapter的这些子类，只是简要介绍一下它们之间的交互关系：

SimpleControllerHandlerAdapter子类、HttpRequestHandlerAdapter子类和SimpleServletHandler子类的整体思路是一样的，首先判断handler是什么类型，三个子类分别对应Controller类型、HttpRequestHandler类型和Servlet类型，然后调用handlerRequest方法，分别使用Controller类型、HttpRequestHandler类型和Servlet类型的调用；

AbstractHandlerMethodAdapter子类对应HandlerMapping组件中的AbstractHandlerMethodMapping子类；

RequestMappingHandlerAdapter子类对应HandlerMapping组件中的RequestMappingHandlerMapping子类，是最复杂也是最常用到的一个Adapter；

！(3)前置处理：执行所有注册拦截器的preHandler方法

我们来看一下preHandler方法：

```
boolean applyPreHandle(HttpServletRequest request, HttpServletResponse
response) throws Exception {
    for (int i = 0; i < this.interceptorList.size(); i++) {
            HandlerInterceptor interceptor = this.interceptorList.get(i);
        if (!interceptor.preHandle(request, response, this.handler)) {
            triggerAfterCompletion(request, response, null);
            return false;
        }
        this.interceptorIndex = i;
    }
    return true;
}
```

在**第！(1)步**中返回的HandlerExecutionChain中封装了Handler和一些拦截器，这里就要调用拦截器；

该函数会调用所有注册拦截器的preHandler方法，如果preHandler方法返回结果为true，则继续运行，否则会调用triggerAfterCopletion方法，该方法会调用所有已经成功执行的拦截器的afterCompletion方法并返回false，则doDiapatcher就不会继续执行下去了，会执行拦截器的afterCompletion方法；

从这里我们可以看出，只有所有注册的拦截器都执行成功，Spring MVC 的核心流程才会继续进行下去。

！(4)真正开始执行用于处理的handler

HandlerAdapter调用Handler处理request和response，目标方法的返回值作为视图名，保存到ModelAndView中，故Handler返回一个ModelAndView对象mv，返回页面自动装配在view属性中。

<1>handle的父类接口：

```
public final ModelAndView handle(HttpServletRequest request,
HttpServletResponse response, Object handler)
            throws Exception {
    return handleInternal(request, response, (HandlerMethod) handler);
}
```

如上，Adapter使用的是其父类的handle接口，父类接口调用子类接口handleInternal作为最终处理的Servlet请求：

<2>handleInternal方法：

```
protected ModelAndView handleInternal(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod)
throws Exception {
    ModelAndView mav;
    //<2.1>检查请求是否合法，对于不支持的请求直接发出异常
    checkRequest(request);

    //<2.2>判断是否要对invokeHandlerMethod加锁实现同步
    if (this.synchronizeOnSession) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            Object mutex = WebUtils.getSessionMutex(session);
            synchronized (mutex) {
                mav = invokeHandlerMethod(request, response, handlerMethod);
            }
        }
        else {
            mav = invokeHandlerMethod(request, response, handlerMethod);
        }
    }
    //TODO<2.3>：执行invokeHandlerMethod并返回视图
    else {
        mav = invokeHandlerMethod(request, response, handlerMethod);
    }

    if (!response.containsHeader(HEADER_CACHE_CONTROL)) {
        if
(getSessionAttributesHandler(handlerMethod).hasSessionAttributes()) {
```

```
            applyCacheSeconds(response,
this.cacheSecondsForSessionAttributeHandlers);
        }
        else {
            prepareResponse(response);
        }
    }

    return mav;
}
```

图中的TODO<2.3>：执行invokeHandlerMethod并返回视图

这里我们主要来分析执行*invokeHandlerMethod*的这一步，这里不论是否加锁这个函数都会执行，这个函数的代码比较长，这里直接分析这个函数的功能：

A. 创建视图容器，用于封装视图、数据模型和处理状态等信息；

B. 调用invokeAndHandle()方法执行对应的处理器handler；

C. 封装ModelAndView实例并返回。

！(5)调用所有拦截器的postHandle方法

```
void applyPostHandle(HttpServletRequest request, HttpServletResponse
response, @Nullable ModelAndView mv)
            throws Exception {
    for (int i = this.interceptorList.size() - 1; i >= 0; i--) {
        HandlerInterceptor interceptor = this.interceptorList.get(i);
        interceptor.postHandle(request, response, this.handler, mv);
    }
}
```

在HandlerExecutionChain类中执行，先从interceptorIndex这个索引开始向前遍历拦截器，然后执行postHandle方法；

！(6)处理视图，进行页面渲染

这里会请求ViewReslover对ModelAndView进行解析，然后再利用render方法对ModelAndView进行页面渲染，我们来看一下render方法的源码：

```
protected void render(ModelAndView mv, HttpServletRequest request,
HttpServletResponse response) throws Exception {
    Locale locale =
            (this.localeResolver != null ?
this.localeResolver.resolveLocale(request) : request.getLocale());
        response.setLocale(locale);

    //获得View对象
    View view;
    String viewName = mv.getViewName();
    //使用ViewName获得View对象
    if (viewName != null) {
        // We need to resolve the view name.
```

```
        view = resolveViewName(viewName, mv.getModelInternal(), locale,
request);
        if (view == null) {//获取不到，发出ServletException异常
            throw new ServletException("Could not resolve view with name '"
+ mv.getViewName() +
                    "' in servlet with name '" + getServletName() + "'");
        }
    }
    //直接使用ModelAndView对象的View对象
    else {
        view = mv.getView();
        if (view == null) {//获取不到，发出ServletException异常
            throw new ServletException("ModelAndView [" + mv + "] neither
contains a view name nor a " +
                    "View object in servlet with name '" + getServletName()
+ "'");
        }
    }

    //打印日志
    if (logger.isTraceEnabled()) {
        logger.trace("Rendering view [" + view + "] ");
    }
    try {
        //设置响应的状态码
        if (mv.getStatus() != null) {
            response.setStatus(mv.getStatus().value());
        }
        //渲染视图ModelAndView
        view.render(mv.getModelInternal(), request, response);
    }
    catch (Exception ex) {
        if (logger.isDebugEnabled()) {
            logger.debug("Error rendering view [" + view + "]", ex);
        }
        throw ex;
    }
}
```

！(7)拦截器收尾

```
void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, @Nullable Exception ex) {
        for (int i = this.interceptorIndex; i >= 0; i--) {
            HandlerInterceptor interceptor = this.interceptorList.get(i);
            try {
                interceptor.afterCompletion(request, response, this.handler, ex);
            }
            catch (Throwable ex2) {
                logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);
            }
        }
    }
```

同样在HandlerExecutionChain类中执行，不论是捕获了异常或者错误，还是正常的渲染结束，都会调用拦截器的afterCompletion方法，这里的执行顺序与拦截器的postHandle方法一样。

参考文献：

CSDN：HTTP的请求方法OPTIONS

CSDN：Spring MVC源码分析(五)（HandlerMethod执行过程解析)https://blog.csdn.net/qq_38975553/article/details/103704036

郝佳：《Spring源码深度解析》

韩路彪：《看透Spring MVC：源代码分析与实践》

博客园：https://www.cnblogs.com/machao/p/5788425.html