DispatcherServlet工作流程

1：前期工作

WebMvcConfigurationSupport springmvc config核心类，该类会往spring ioc 容器中注入RequestMappingHandlerMapping，SimpleUrlHandlerMapping，BeanNameUrlHandlerMapping，ResourceHandlerMapping, DefaultServletHandlerConfigurer, RequestMappingHandlerAdapte。而RequestMappingHandlerMapping，RequestMappingHandlerAdapte这两个是mvc核心处理类

如果只是使用springmvc，我们可以使用DelegatingWebMvcConfiguration其继承WebMvcConfigurationSupport

这个配置类进行委托处理，这也是spring提供给我们的委托类，该类会收集我们自定义实现了WebMvcConfigurer接口的类。事实上springboot的*@EnableWebMvc*

这个注解也是导入了DelegatingWebMvcConfiguration。其的作用就是WebMvcConfigurationSupport这个配置类会往spring ioc容器中导入RequestMappingHandlerMapping，SimpleUrlHandlerMapping，BeanNameUrlHandlerMapping，ResourceHandlerMapping, DefaultServletHandlerConfigurer, RequestMappingHandlerAdapte这些bean。这些bean在创建时，就会从WebMvcConfigurer接口的现实类中获取相关配置。

如果是使用springboot,且不配置*@EnableWebMvc注解，*springboot会自动注入EnableWebMvcConfiguration

2：DispatcherServlet工作部分

1: 初始化

*/\*\*  
 \* This implementation calls {****@link*** *#initStrategies}.  
 \*/*@Override  
**protected void** onRefresh(ApplicationContext context) {  
 initStrategies(context);  
}  
  
*/\*\*  
 \* Initialize the strategy objects that this servlet uses.  
 \* <p>May be overridden in subclasses in order to initialize further strategy objects.  
 \*/***protected void** initStrategies(ApplicationContext context) {

// 添加文件上传解析器，springboot 自动注入StandardServletMultipartResolver

initMultipartResolver(context);

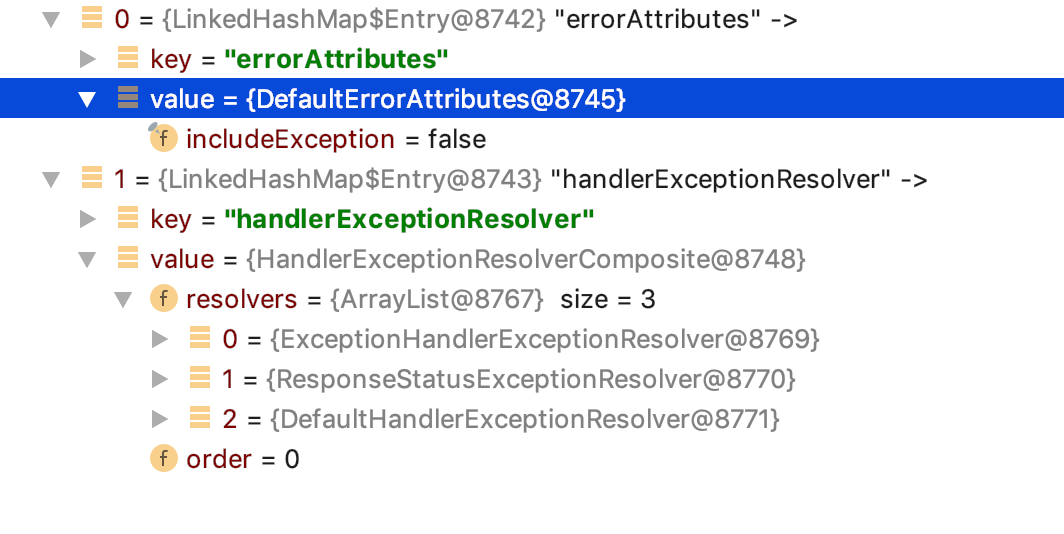
// 添加国际化解析器  
 initLocaleResolver(context);

// 添加主题解析器（不清楚干啥用）  
 initThemeResolver(context);

// 添加HandlerMapping,  
 initHandlerMappings(context);

// 添加HandleMappingAdapter  
 initHandlerAdapters(context);

// 添加异常解析器  
 initHandlerExceptionResolvers(context);



// 添加视图名称解析器，默认添加DefaultRequestToViewNameTranslator  
 initRequestToViewNameTranslator(context);

// 视图解析器，默认添加如下5个解析器

  
 initViewResolvers(context);

// 添加flashmanager(不知道干啥用)  
 initFlashMapManager(context);  
}

2：接受请求

@Override  
**protected void** doService(HttpServletRequest request, HttpServletResponse response) **throws** Exception {

// 根据日志级别打印不同的request的请求信息，debug会打印请求参数，trace会增加

// 请求头的打印信息  
 logRequest(request);

// 保存属性快照，如果是include请求，且属性已**org.springframework.web.servlet**

开头  
 *// Keep a snapshot of the request attributes in case of an include,  
 // to be able to restore the original attributes after the include.* Map<String, Object> attributesSnapshot = **null**;  
 **if** (WebUtils.*isIncludeRequest*(request)) {  
 attributesSnapshot = **new** HashMap<>();  
 Enumeration<?> attrNames = request.getAttributeNames();  
 **while** (attrNames.hasMoreElements()) {  
 String attrName = (String) attrNames.nextElement();  
 **if** (**this**.**cleanupAfterInclude** || attrName.startsWith(***DEFAULT\_STRATEGIES\_PREFIX***)) {  
 attributesSnapshot.put(attrName, request.getAttribute(attrName));  
 }  
 }  
 }  
// 在当前request上设置spring ioc容器

*// Make framework objects available to handlers and view objects.* request.setAttribute(***WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE***, getWebApplicationContext());

// 在当前request上设置国际化资源解析器  
 request.setAttribute(***LOCALE\_RESOLVER\_ATTRIBUTE***, **this**.**localeResolver**);

// 在当前request上设置主题解析器  
 request.setAttribute(***THEME\_RESOLVER\_ATTRIBUTE***, **this**.**themeResolver**);

// 在当前request上设置themeSource  
 request.setAttribute(***THEME\_SOURCE\_ATTRIBUTE***, getThemeSource());  
// **flashMapManager**处理，  
 **if** (**this**.**flashMapManager** != **null**) {  
 FlashMap inputFlashMap = **this**.**flashMapManager**.retrieveAndUpdate(request, response);  
 **if** (inputFlashMap != **null**) {  
 request.setAttribute(***INPUT\_FLASH\_MAP\_ATTRIBUTE***, Collections.*unmodifiableMap*(inputFlashMap));  
 }  
 request.setAttribute(***OUTPUT\_FLASH\_MAP\_ATTRIBUTE***, **new** FlashMap());  
 request.setAttribute(***FLASH\_MAP\_MANAGER\_ATTRIBUTE***, **this**.**flashMapManager**);  
 }  
  
 **try** {  
 doDispatch(request, response);  
 }  
 **finally** {

// 如果异步任务执行完成或者不是异步且数据快照不是null ,重置request属性  
 **if** (!WebAsyncUtils.*getAsyncManager*(request).isConcurrentHandlingStarted()) {  
 *// Restore the original attribute snapshot, in case of an include.* **if** (attributesSnapshot != **null**) {  
 restoreAttributesAfterInclude(request, attributesSnapshot);  
 }  
 }  
 }  
}

**protected void** doDispatch(HttpServletRequest request, HttpServletResponse response) **throws** Exception {  
 HttpServletRequest processedRequest = request;  
 HandlerExecutionChain mappedHandler = **null**;  
 **boolean** multipartRequestParsed = **false**;  
 // 获取异步管理器，并绑定到request上  
 WebAsyncManager asyncManager = WebAsyncUtils.*getAsyncManager*(request);  
  
 **try** {  
 ModelAndView mv = **null**;  
 Exception dispatchException = **null**;  
  
 **try** {

// 判断是否是文件上传类型的请求，判断依据就是获取request的contentType是否已 //“multipart/ 开头“，如果是将request包装在//StandardMultipartHttpServletRequest中，并获取request中的文件数据,并返回这个包装类

processedRequest = checkMultipart(request);  
 multipartRequestParsed = (processedRequest != request);

// 根据request的uri获取指定的HandelMapping，http请求就是返回、、//RequestMappingHandlerMapping  
 *// Determine handler for the current request.* mappedHandler = getHandler(processedRequest);  
 **if** (mappedHandler == **null**) {  
 noHandlerFound(processedRequest, response);  
 **return**;  
 }  
  
 *// Determine handler adapter for the current request.* HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());  
  
 *// Process last-modified header, if supported by the handler.* String method = request.getMethod();  
 **boolean** isGet = **"GET"**.equals(method);  
 **if** (isGet || **"HEAD"**.equals(method)) {  
 **long** lastModified = ha.getLastModified(request, mappedHandler.getHandler());  
 **if** (**new** ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {  
 **return**;  
 }  
 }  
  
 **if** (!mappedHandler.applyPreHandle(processedRequest, response)) {  
 **return**;  
 }  
  
 *// Actually invoke the handler.* mv = ha.handle(processedRequest, response, mappedHandler.getHandler());  
  
 **if** (asyncManager.isConcurrentHandlingStarted()) {  
 **return**;  
 }  
  
 applyDefaultViewName(processedRequest, mv);  
 mappedHandler.applyPostHandle(processedRequest, response, mv);  
 }  
 **catch** (Exception ex) {  
 dispatchException = ex;  
 }  
 **catch** (Throwable err) {  
 *// As of 4.3, we're processing Errors thrown from handler methods as well,  
 // making them available for @ExceptionHandler methods and other scenarios.* dispatchException = **new** NestedServletException(**"Handler dispatch failed"**, err);  
 }  
 processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);  
 }  
 **catch** (Exception ex) {  
 triggerAfterCompletion(processedRequest, response, mappedHandler, ex);  
 }  
 **catch** (Throwable err) {  
 triggerAfterCompletion(processedRequest, response, mappedHandler,  
 **new** NestedServletException(**"Handler processing failed"**, err));  
 }  
 **finally** {  
 **if** (asyncManager.isConcurrentHandlingStarted()) {  
 *// Instead of postHandle and afterCompletion* **if** (mappedHandler != **null**) {  
 mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);  
 }  
 }  
 **else** {  
 *// Clean up any resources used by a multipart request.* **if** (multipartRequestParsed) {  
 cleanupMultipart(processedRequest);  
 }  
 }  
 }  
}