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

// web容器启动时不会进行初始化，因为配置启动时不进行初始化，只有在有请求时才进行初始化（只会加载一次）

dispatchServlet.setLoadOnStartup(-1);

*/\*\*  
 \* 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);

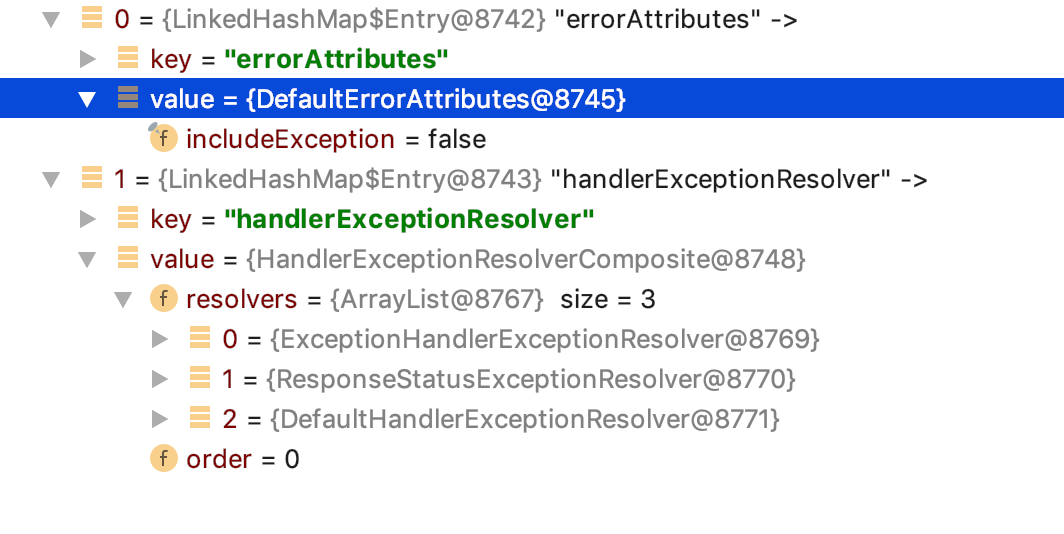
// 添加国际化解析器  
 initLocaleResolver(context);

// 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**;  
 }  
// 根据HandlerMapping获取HandlerAdapter  
 *// Determine handler adapter for the current request.* HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());  
// 如果是GET或者HEAD请求，判断是否被修改过， 如果么有就直接返回  
 *// 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**;  
 }  
// 开始处理请求  
 *// Actually invoke the handler.* mv = ha.handle(processedRequest, response, mappedHandler.getHandler());  
// 当前请求是否是异步任务，finally 中等待结果  
 **if** (asyncManager.isConcurrentHandlingStarted()) {  
 **return**;  
 }  
// 设置默认的视图名称  
 applyDefaultViewName(processedRequest, mv);

// 拦截器后置处理  
 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);  
 }

// 使用视图开始渲染  
 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);  
 }  
 }  
 }  
}

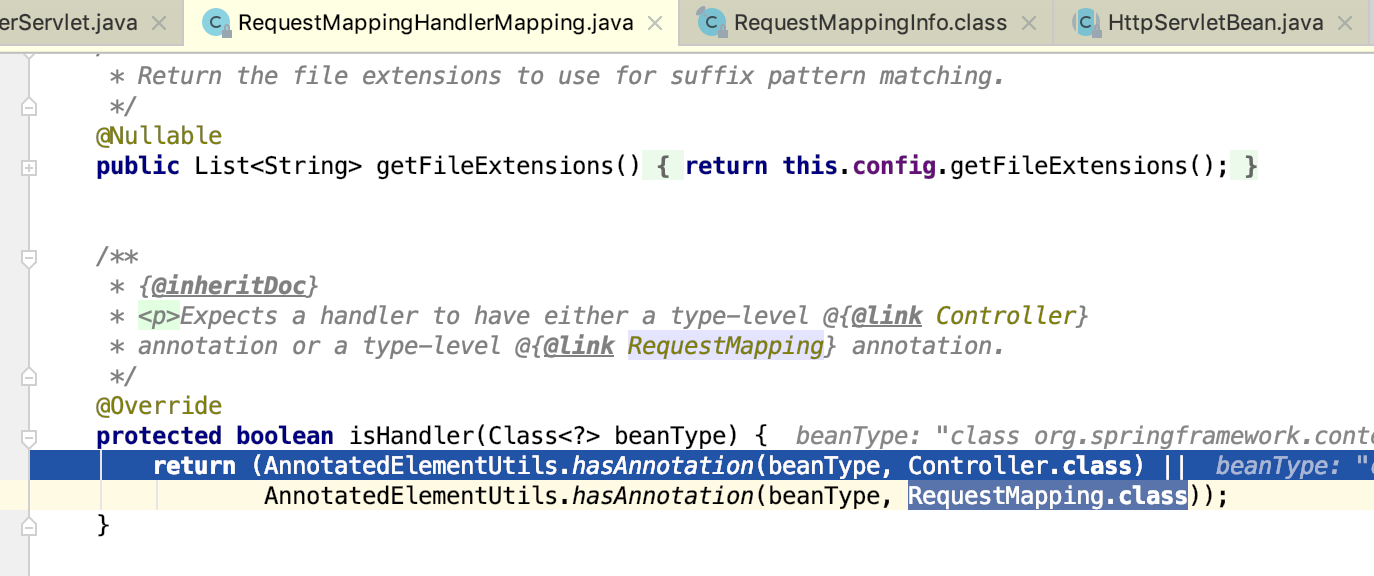
3：RequestMappingHandlerMapping初始化工作

@Override  
**public void** afterPropertiesSet() {

// 设置path解析器  
 **this**.**config** = **new** RequestMappingInfo.BuilderConfiguration();  
 **this**.**config**.setUrlPathHelper(getUrlPathHelper());  
 **this**.**config**.setPathMatcher(getPathMatcher());  
 **this**.**config**.setSuffixPatternMatch(**this**.**useSuffixPatternMatch**);  
 **this**.**config**.setTrailingSlashMatch(**this**.**useTrailingSlashMatch**);  
 **this**.**config**.setRegisteredSuffixPatternMatch(**this**.**useRegisteredSuffixPatternMatch**);  
 **this**.**config**.setContentNegotiationManager(getContentNegotiationManager());  
// 获取springioc容器中包含有Controller.**class**，RequestMapping.**class**

注解的类，并解析RequestMapping注解，获取映射路径，然后保存在MappingRegistry

中，供后续匹配请求路径使用  
 **super**.afterPropertiesSet();  
}



4：开始具体处理

@Override  
**protected** ModelAndView handleInternal(HttpServletRequest request,  
 HttpServletResponse response, HandlerMethod handlerMethod) **throws** Exception {  
  
 ModelAndView mav;

// 校验请求方式是否是我们支持的,默认支持***GET***, ***HEAD***, ***POST***, ***PUT***, ***PATCH***, ***DELETE***, ***OPTIONS***这些  
 checkRequest(request);  
// 判断存在session时是否需要进行同步  
 *// Execute invokeHandlerMethod in synchronized block if required.* **if** (**this**.**synchronizeOnSession**) {  
 HttpSession session = request.getSession(**false**);  
 **if** (session != **null**) {  
 Object mutex = WebUtils.*getSessionMutex*(session);  
 **synchronized** (mutex) {  
 mav = invokeHandlerMethod(request, response, handlerMethod);  
 }  
 }  
 **else** {  
 *// No HttpSession available -> no mutex necessary* mav = invokeHandlerMethod(request, response, handlerMethod);  
 }  
 }  
 **else** {  
 *// No synchronization on session demanded at all...* mav = invokeHandlerMethod(request, response, handlerMethod);  
 }  
// 判断response是否设置了缓存配置，没有设置进行处理，主要设置缓存头相关配置  
 **if** (!response.containsHeader(***HEADER\_CACHE\_CONTROL***)) {  
 **if** (getSessionAttributesHandler(handlerMethod).hasSessionAttributes()) {  
 applyCacheSeconds(response, **this**.**cacheSecondsForSessionAttributeHandlers**);  
 }  
 **else** {  
 prepareResponse(response);  
 }  
 }  
  
 **return** mav;  
}

*/\*\*  
 \* Invoke the {****@link*** *RequestMapping} handler method preparing a {****@link*** *ModelAndView}  
 \* if view resolution is required.  
 \** ***@since*** *4.2  
 \** ***@see*** *#createInvocableHandlerMethod(HandlerMethod)  
 \*/*@Nullable  
**protected** ModelAndView invokeHandlerMethod(HttpServletRequest request,  
 HttpServletResponse response, HandlerMethod handlerMethod) **throws** Exception {  
  
 ServletWebRequest webRequest = **new** ServletWebRequest(request, response);  
 **try** {  
 WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);  
 ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);  
  
 ServletInvocableHandlerMethod invocableMethod = createInvocableHandlerMethod(handlerMethod);  
 **if** (**this**.**argumentResolvers** != **null**) {

// 设置参数解析器  
 invocableMethod.setHandlerMethodArgumentResolvers(**this**.**argumentResolvers**);  
 }  
 **if** (**this**.**returnValueHandlers** != **null**) {

// 设置返回值解析器  
 invocableMethod.setHandlerMethodReturnValueHandlers(**this**.**returnValueHandlers**);  
 }  
 invocableMethod.setDataBinderFactory(binderFactory);  
 invocableMethod.setParameterNameDiscoverer(**this**.**parameterNameDiscoverer**);  
  
 ModelAndViewContainer mavContainer = **new** ModelAndViewContainer();  
 mavContainer.addAllAttributes(RequestContextUtils.*getInputFlashMap*(request));  
 modelFactory.initModel(webRequest, mavContainer, invocableMethod);  
 mavContainer.setIgnoreDefaultModelOnRedirect(**this**.**ignoreDefaultModelOnRedirect**);  
// 创建标准异步请求

AsyncWebRequest asyncWebRequest = WebAsyncUtils.*createAsyncWebRequest*(request, response);

// 设置异步超时时间  
 asyncWebRequest.setTimeout(**this**.**asyncRequestTimeout**);  
// 获取异步任务管理器，并绑定到request上  
 WebAsyncManager asyncManager = WebAsyncUtils.*getAsyncManager*(request);

// 设置异步任务线程池  
 asyncManager.setTaskExecutor(**this**.**taskExecutor**);

// 向异步任务管理器中设置异步请求  
 asyncManager.setAsyncWebRequest(asyncWebRequest);

// 设置callable任务类型的拦截器  
 asyncManager.registerCallableInterceptors(**this**.**callableInterceptors**);  
// 设置deferredResult任务类型的拦截器 asyncManager.registerDeferredResultInterceptors(**this**.**deferredResultInterceptors**);  
// 如果异步管理器有结果，进入下面进行处理  
 **if** (asyncManager.hasConcurrentResult()) {  
 Object result = asyncManager.getConcurrentResult();  
 mavContainer = (ModelAndViewContainer) asyncManager.getConcurrentResultContext()[0];  
 asyncManager.clearConcurrentResult();  
 LogFormatUtils.*traceDebug*(**logger**, traceOn -> {  
 String formatted = LogFormatUtils.*formatValue*(result, !traceOn);  
 **return "Resume with async result ["** + formatted + **"]"**;  
 });  
 invocableMethod = invocableMethod.wrapConcurrentResult(result);  
 }  
// 开始执行controller对应的方法  
 invocableMethod.invokeAndHandle(webRequest, mavContainer);

// 如果异步管理器存在异步任务开始执行，直接返回  
 **if** (asyncManager.isConcurrentHandlingStarted()) {  
 **return null**;  
 }  
  
 **return** getModelAndView(mavContainer, modelFactory, webRequest);  
 }  
 **finally** {  
 webRequest.requestCompleted();  
 }  
}

*/\*\*  
 \* Invoke the method and handle the return value through one of the  
 \* configured {****@link*** *HandlerMethodReturnValueHandler HandlerMethodReturnValueHandlers}.  
 \** ***@param webRequest*** *the current request  
 \** ***@param mavContainer*** *the ModelAndViewContainer for this request  
 \** ***@param providedArgs*** *"given" arguments matched by type (not resolved)  
 \*/***public void** invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,  
 Object... providedArgs) **throws** Exception {  
// 执行controller的方法  
 Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);  
 setResponseStatus(webRequest);

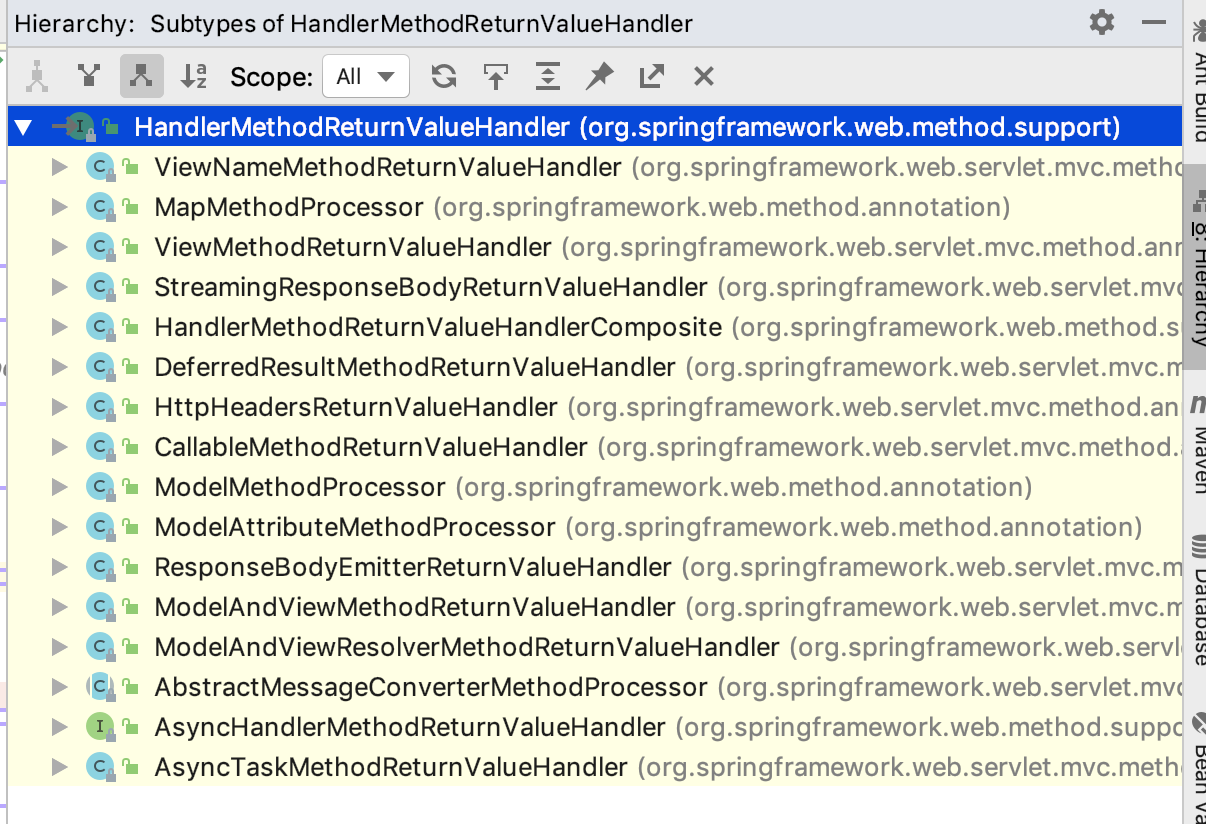
// 没有返回值走下面  
 **if** (returnValue == **null**) {  
 **if** (isRequestNotModified(webRequest) || getResponseStatus() != **null** || mavContainer.isRequestHandled()) {  
 mavContainer.setRequestHandled(**true**);  
 **return**;  
 }  
 }

// 存在Reason走下面的方法  
 **else if** (StringUtils.*hasText*(getResponseStatusReason())) {  
 mavContainer.setRequestHandled(**true**);  
 **return**;  
 }  
  
 mavContainer.setRequestHandled(**false**);  
 Assert.*state*(**this**.**returnValueHandlers** != **null**, **"No return value handlers"**);

// 根据方法值类型，获取相应的返回值类型处理器进行处理  
 **try** {  
 **this**.**returnValueHandlers**.handleReturnValue(  
 returnValue, getReturnValueType(returnValue), mavContainer, webRequest);  
 }  
 **catch** (Exception ex) {  
 **if** (**logger**.isTraceEnabled()) {  
 **logger**.trace(formatErrorForReturnValue(returnValue), ex);  
 }  
 **throw** ex;  
 }  
}

*/\*\*  
 \* Iterate over registered {****@link*** *HandlerMethodReturnValueHandler HandlerMethodReturnValueHandlers} and invoke the one that supports it.  
 \** ***@throws*** *IllegalStateException if no suitable {****@link*** *HandlerMethodReturnValueHandler} is found.  
 \*/*@Override  
**public void** handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,  
 ModelAndViewContainer mavContainer, NativeWebRequest webRequest) **throws** Exception {  
// 根据返回值类型，获取具体的返回值处理器  
 HandlerMethodReturnValueHandler handler = selectHandler(returnValue, returnType);  
 **if** (handler == **null**) {  
 **throw new** IllegalArgumentException(**"Unknown return value type: "** + returnType.getParameterType().getName());  
 }  
 handler.handleReturnValue(returnValue, returnType, mavContainer, webRequest);  
}  
  
@Nullable  
**private** HandlerMethodReturnValueHandler selectHandler(@Nullable Object value, MethodParameter returnType) {  
 **boolean** isAsyncValue = isAsyncReturnValue(value, returnType);  
 **for** (HandlerMethodReturnValueHandler handler : **this**.**returnValueHandlers**) {  
 **if** (isAsyncValue && !(handler **instanceof** AsyncHandlerMethodReturnValueHandler)) {  
 **continue**;  
 }  
 **if** (handler.supportsReturnType(returnType)) {  
 **return** handler;  
 }  
 }  
 **return null**;  
}

返回值解析器如下这些



都是在RequestMappingHandlerAdapter初始化是添加

DispatchServlet初始化：

1：获取文件上传解析器

2：获取国际化解析器，获取不到，取默认

3：获取Theme解析器，获取不到，取默认

4：获取HandlerMapping

5：获取HandAdapter (包含自己指定的参数，返回值解析器)

6：获取HandException(包含自己指定的参数，返回值解析器)

7：获取视图名称解析器(当ModelAndView中没有view名称时，用于获取默认的view名称)

8：获取视图解析器

9：获取flushManager

DispatchServlet工作流程总结：

1：获取WebAsyncManager并绑定到request上

2：检查是否是文件上传请求，是就获取文件相关信息

3：根据请求路径获取对应的MappingHandler,一般就是RequestMappingHandlerMapping

4：根据MappingHandler获取对应的HandlerAdapter

5：拦截器前置处理

6：HandlerAdapter开始处理请求

7：HandlerAdapter处理细节（获取参数，返回值解析器，获取异步任务管理器，并设置异步任务线程池，设置异步超时时间，设置callable类型的拦截器，设置deferredResult类型的拦截器）

8：通过反射调用controller的方法

9：根据返回值获取相应的返回值解析器，并对返回值进行处理

10：如果有requestDestructionCallbacks，则触发相应的回调处理

11：拦截器后置处理

12：有异常，使用异常解析器进行处理

13：渲染view

14：拦截器Completion触发

15：FrameworkServlet.publishRequestHandledEvent(request, response, startTime, failureCause); 派发请求完成事件