芋道源码 —— 知识星球

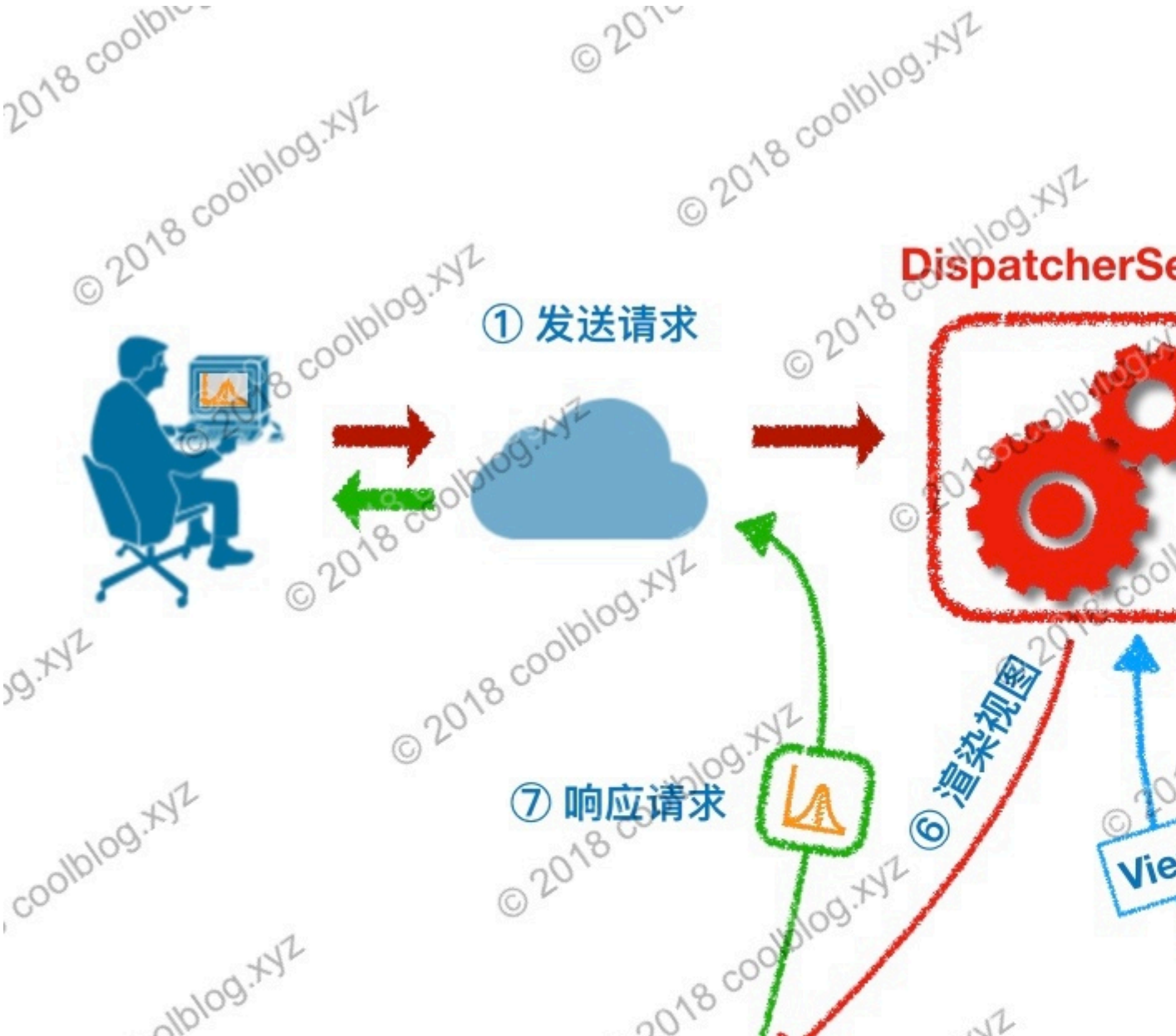
我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2020-02-21
Spring MVC

精尽 Spring MVC 源码分析 —— 请求处理一览

1. 概述

本文，我们来一览一个用户的请求，是如何被 DispatcherServlet 处理的。如下图所示：

FROM 《Spring MVC 原理探秘 —— 一个请求的旅行过程》



整体流程实际不复杂，但是涉及的全部代码会非常多，所以本文重点在于解析整体的流程。特别具体和细节的代码实现，我们会放到后续的文章，一篇一篇细细咀嚼。

1.1 如何调试

芳芳：自我吐槽，写完之后才发现，忘记提供测试示例了。

比较简单，调试 `org.springframework.web.servlet.DispatcherServletTests` 这个单元测试类，可以运行各种单元测试方法，执行各种情况。

2. FrameworkServlet

虽然在 「1. 概述」 的整体流程图，我们看到请求首先是被 DispatcherServlet 所处理，但是实际上，FrameworkServlet 才是真正的入门。FrameworkServlet 会实现

```
#doGet(HttpServletRequest request, HttpServletResponse response)
#doPost(HttpServletRequest request, HttpServletResponse response)
#doPut(HttpServletRequest request, HttpServletResponse response)
#doDelete(HttpServletRequest request, HttpServletResponse response)
#doOptions(HttpServletRequest request, HttpServletResponse response)
#doTrace(HttpServletRequest request, HttpServletResponse response)
#service(HttpServletRequest request, HttpServletResponse response)
```

等方法。而这些实现，最终会调用 `#processRequest(HttpServletRequest request, HttpServletResponse response)` 方法，处理请求。

2.1 不同 HttpMethod 的请求处理

2.1.1 service

`#service(HttpServletRequest request, HttpServletResponse response)` 方法，代码如下：

```
// FrameworkServlet.java

@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> 调用父类，处理其它请求
    } else {
     super.service(request, response);
    }
}
```

<1> 处，获得请求方法。
<2.1> 处，若请求方法是 `HttpMethod.PATCH` ，调用 `#processRequest(HttpServletRequest request, HttpServletResponse response)` 方法，处理请求。因为 HttpServlet 默认没提供 `#doPatch(HttpServletRequest request, HttpServletResponse response)`

方法，所以只能通过父类的 #service(...) 方法，从而实现。另外，关于 processRequest 的详细解析，见 「2.2 processRequest」 。

<2.2> 处，其它类型的请求方法，还是调用父类的 #service(HttpServletRequest request, HttpServletResponse response) 方法，进行处理。代码如下：

```
// HttpServlet.java

protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    String method = req.getMethod();

    if (method.equals(METHOD_GET)) {
        long lastModified = getLastModified(req);
        if (lastModified == -1) {
            // servlet doesn't support if-modified-since, no reason
            // to go through further expensive logic
            doGet(req, resp);
        } else {
            long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
            if (ifModifiedSince < lastModified) {
                // If the servlet mod time is later, call doGet()
                // Round down to the nearest second for a proper compare
                // A ifModifiedSince of -1 will always be less
                maybeSetLastModified(resp, lastModified);
                doGet(req, resp);
            } else {
                resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
            }
        }
    } else if (method.equals(METHOD_HEAD)) {
        long lastModified = getLastModified(req);
        maybeSetLastModified(resp, lastModified);
        doHead(req, resp);
    } else if (method.equals(METHOD_POST)) {
        doPost(req, resp);
    } else if (method.equals(METHOD_PUT)) {
        doPut(req, resp);
    } else if (method.equals(METHOD_DELETE)) {
        doDelete(req, resp);
    } else if (method.equals(METHOD_OPTIONS)) {
        doOptions(req,resp);
    } else if (method.equals(METHOD_TRACE)) {
        doTrace(req,resp);
    } else {
        //
        // Note that this means NO servlet supports whatever
        // method was requested, anywhere on this server.
        //

        String errMsg = lStrings.getString("http.method_not_implemented");
        Object[] errArgs = new Object[1];
        errArgs[0] = method;
        errMsg = MessageFormat.format(errMsg, errArgs);

        resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED, errMsg);
    }
}
```

可能会有胖友有疑惑，为什么不在 #service(HttpServletRequest request, HttpServletResponse response) 方法，直接调用 #processRequest(HttpServletRequest request, HttpServletResponse response) 方法就好列？因为针对不同的请求方法，处理略微有所不同。

2.1.2 doGet & doPost & doPut & doDelete

这四个方法，都是直接调用 #processRequest(HttpServletRequest request, HttpServletResponse response) 方法，处理请求。代码如下：

```
// FrameworkServlet.java

@Override
protected final void doGet(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {
    processRequest(request, response);
}

@Override
protected final void doPost(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {
    processRequest(request, response);
}

@Override
protected final void doPut(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {
    processRequest(request, response);
}

@Override
protected final void doDelete(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {
    processRequest(request, response);
}
```

2.1.3 doOptions

```
// FrameworkServlet.java

/** Should we dispatch an HTTP OPTIONS request to {@link #doService}?. */
private boolean dispatchOptionsRequest = false;

/**
 * Delegate OPTIONS requests to {@link #processRequest}, if desired.
 * <p>Applies HttpServlet's standard OPTIONS processing otherwise,
 * and also if there is still no 'Allow' header set after dispatching.
 * @see #doService
 */
@Override
protected void doOptions(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {
 // 如果 dispatchOptionsRequest 为 true ，则处理该请求
 if (this.dispatchOptionsRequest || CorsUtils.isPreFlightRequest(request)) {
        // 处理请求
        processRequest(request, response);
     // 如果响应 Header 包含 "Allow" ，则不需要交给父方法处理
```

```
        if (response.containsHeader("Allow")) {
            // Proper OPTIONS response coming from a handler - we're done.
            return;
        }
    }

    // Use response wrapper in order to always add PATCH to the allowed methods
    // 调用父方法，并在响应 Header 的 "Allow" 增加 PATCH 的值
    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);
        }
    });
}
```

选读，因为 OPTIONS 请求方法，实际场景下用的少。
可参考 《HTTP 的请求方法 OPTIONS》 。

2.1.4 doTrace

```
// FrameworkServlet.java

/** Should we dispatch an HTTP TRACE request to {@link #doService}?. */
private boolean dispatchTraceRequest = false;

/**
 * Delegate TRACE requests to {@link #processRequest}, if desired.
 * <p>Applies HttpServlet's standard TRACE processing otherwise.
 * @see #doService
 */
@Override
protected void doTrace(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    // 如果 dispatchTraceRequest 为 true ，则处理该请求
    if (this.dispatchTraceRequest) {
        // 处理请求
        processRequest(request, response);
        // 如果响应的内容类型为 "message/http" ，则不需要交给父方法处理
        if ("message/http".equals(response.getContentType())) {
            // Proper TRACE response coming from a handler - we're done.
            return;
        }
    }

    // 调用父方法
    super.doTrace(request, response);
}
```

选读，因为 TRACE 请求方法，实际场景下用的少。
可参读 《HTTP Method详细解读(GET HEAD POST OPTIONS PUT DELETE TRACE CONNECT)》 的 「9.8 TRACE」 小节。

2.2 processRequest

#processRequest(HttpServletRequest request, HttpServletResponse response) 方法，处理请求。代码如下：

```
// FrameworkServlet.java

protected final void processRequest(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {
 // <1> 记录当前时间，用于计算 web 请求的处理时间
 long startTime = System.currentTimeMillis();
 // <2> 记录异常
    Throwable failureCause = null;

 // <3> TODO 芋艿
    LocaleContext previousLocaleContext = LocaleContextHolder.getLocaleContext();
    LocaleContext localeContext = buildLocaleContext(request);

 // <4> TODO 芋艿
    RequestAttributes previousAttributes = RequestContextHolder.getRequestAttributes();
    ServletRequestAttributes requestAttributes = buildRequestAttributes(request, response, previousAttributes);

 // <5> TODO 芋艿
    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
    asyncManager.registerCallableInterceptor(FrameworkServlet.class.getName(), new RequestBindingInterceptor());

 // <6> TODO 芋艿
    initContextHolders(request, localeContext, requestAttributes);

 try {
     // <7> 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 {
     // <9> TODO 芋艿
        resetContextHolders(request, previousLocaleContext, previousAttributes);
     // <10> TODO 芋艿
     if (requestAttributes != null) {
            requestAttributes.requestCompleted();
        }
     // <11> 打印请求日志，并且日志级别为 DEBUG
        logResult(request, response, failureCause, asyncManager);
     // <12> 发布 ServletRequestHandledEvent 事件
        publishRequestHandledEvent(request, response, startTime, failureCause);
    }
}
```

<1> 处，记录当前时间，用于计算 web 请求的处理时间。
<2> 处，记录异常。
<3> 处，TODO 1001 Locale
<4> 处，TODO 1002 RequestAttributes
<5> 处，TODO 1003 Asyn
<6> 处，TODO 1001 + 1002

【重要】 <7> 处，调用 #doService(HttpServletRequest request, HttpServletResponse response) 抽象方法，执行真正的逻辑。代码如下：

```
// FrameworkServlet.java

protected abstract void doService(HttpServletRequest request, HttpServletResponse response)
      throws Exception;
```

- 该抽象方法由 DispatcherServlet 实现，所以这就是 DispatcherServlet 处理请求的真正入口。详细解析，见 「3. DispatcherServlet」 。

<8> 处，记录抛出的异常，最终在 finally 的代码段中使用。

<9> 处，TODO 1001 + 1002
<10> 处，TODO 1001 + 1002
<11> 处，调用 #logResult(HttpServletRequest request, HttpServletResponse response, Throwable failureCause, WebAsyncManager asyncManager) 方法，打印请求日志，并且日志级别为 DEBUG 。这个方法，感兴趣的胖友，点击 传送门 查看。

<12> 处，调用 #publishRequestHandledEvent(HttpServletRequest request, HttpServletResponse response, long startTime, Throwable failureCause) 方法，发布 org.springframework.web.context.support.ServletRequestHandledEvent 事件。代码如下：

```
// FrameworkServlet.java

/** Should we publish a ServletRequestHandledEvent at the end of each request?. */
private boolean publishEvents = true;

private void publishRequestHandledEvent(HttpServletRequest request, HttpServletResponse response,
      long startTime, @Nullable Throwable failureCause) {
  // 如果开启发布事件
  if (this.publishEvents && this.webApplicationContext != null) {
      // Whether or not we succeeded, publish an event.
      long processingTime = System.currentTimeMillis() - startTime;
      // 创建 ServletRequestHandledEvent 事件，并进行发布
      this.webApplicationContext.publishEvent(
              new ServletRequestHandledEvent(this,
                        request.getRequestURI(), request.getRemoteAddr(),
                        request.getMethod(), getServletConfig().getServletName(),
                        WebUtils.getSessionId(request), getUsernameForRequest(request),
                        processingTime, failureCause, response.getStatus()));
    }
}
```

- 关于 ServletRequestHandledEvent 的监听使用，可参考 《使用 Spring ApplicationListener 容器监听器来记录请求信息》 。

芋艿：好像到了此处，一直没写到 DispatcherServlet 。 有种在啰嗦的感觉，嘿嘿。

3. DispatcherServlet

3.1 doService

#doService(HttpServletRequest request, HttpServletResponse response) 方法，DispatcherServlet 的处理请求的入口方法，代码如下：

```
// DispatcherServlet.java

@Override
protected void doService(HttpServletRequest request, HttpServletResponse response) throws Exception {
 // <1> 打印请求日志，并且日志级别为 DEBUG
    logRequest(request);

 // Keep a snapshot of the request attributes in case of an include,
 // to be able to restore the original attributes after the include.
 // <2> TODO 芋艿
    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));
            }
        }
    }

 // Make framework objects available to handlers and view objects.
 // <3> 设置 Spring 框架中的常用对象到 request 属性中
    request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE, getWebApplicationContext());
    request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);
    request.setAttribute(THEME_RESOLVER_ATTRIBUTE, this.themeResolver);
    request.setAttribute(THEME_SOURCE_ATTRIBUTE, getThemeSource());

 // <4> TODO 芋艿 flashMapManager
 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 {
     // <5> 执行请求的分发
        doDispatch(request, response);
    } finally {
     // <6> TODO 芋艿
     if (!WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
         // Restore the original attribute snapshot, in case of an include.
         if (attributesSnapshot != null) {
                restoreAttributesAfterInclude(request, attributesSnapshot);
            }
        }
    }
}
```

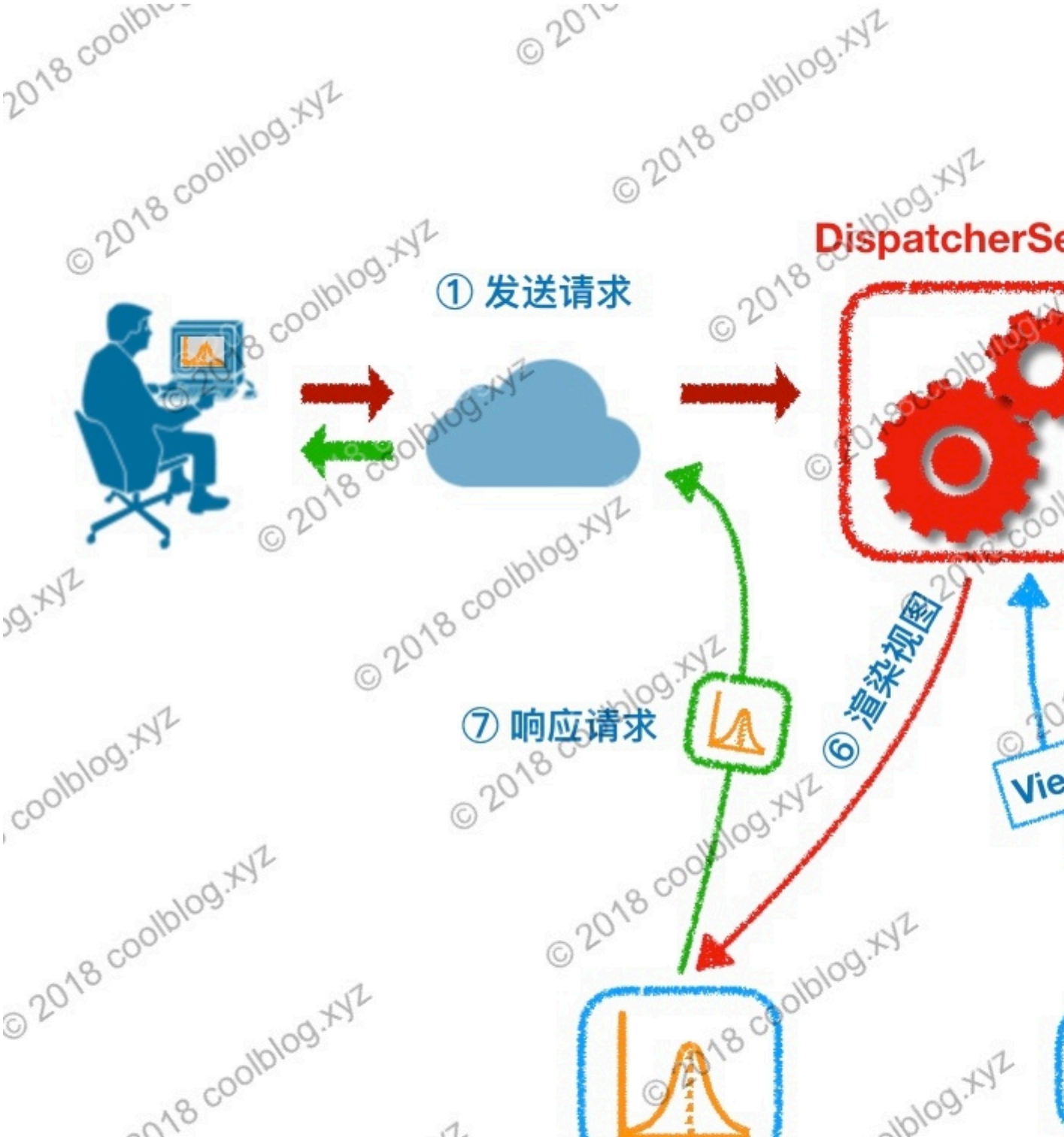
<1>

处，调用 `#logRequest(HttpServletRequest request)` 方法，打印请求日志，并且日志级别为 DEBUG 。这个方法，感兴趣的胖友，点击 传送门 查看。
<2> 处，TODO 1003 芋艿
<3> 处，设置 Spring 框架中的常用对象到 `request` 的属性中。
<4> 处，TODO 1004 芋艿 flashMapManager
<5> 处，调用 `#doDispatch(HttpServletRequest request, HttpServletResponse response)` 方法，执行请求的分发。详细解析，见 「3.2 doDispatch」 。
<6> 处，TODO 1003 芋艿

3.2 doDispatch

`#doDispatch(HttpServletRequest request, HttpServletResponse response)` 方法，执行请求的分发。在开始看具体的代码实现之前，我们在来回味下这张图片：

FROM 《Spring MVC 原理探秘 —— 一个请求的旅行过程》



实际上，这张图，更多的反应的是 DispatcherServlet 的 #DispatcherServlet(...) 方法的核心流程。

代码如下：

```
// DispatcherServlet.java

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

 // <1> TODO 芋艿
    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

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

     try {
         // TODO 芋艿
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

         // Determine handler for the current request.
         // <3> 获得请求对应的 HandlerExecutionChain 对象
            mappedHandler = getHandler(processedRequest);
         if (mappedHandler == null) { // <3.1> 如果获取不到，则根据配置抛出异常或返回 404 错误
                noHandlerFound(processedRequest, response);
             return;
            }

         // Determine handler adapter for the current request.
         // <4> 获得当前 handler 对应的 HandlerAdapter 对象
            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

         // Process last-modified header, if supported by the handler.
         // <4.1> TODO 芋艿 last-modified
            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;
                }
            }

         // <5> 前置处理 拦截器
         if (!mappedHandler.applyPreHandle(processedRequest, response)) {
             return;
            }

         // Actually invoke the handler.
         // <6> 真正的调用 handler 方法，并返回视图
            mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

         // <7> TODO 芋艿
         if (asyncManager.isConcurrentHandlingStarted()) {
             return;
            }
```

```
            // <8> TODO 芋艿 视图
                applyDefaultViewName(processedRequest, mv);
            // <9> 后置处理 拦截器
                mappedHandler.applyPostHandle(processedRequest, response, mv);
            } catch (Exception ex) {
                dispatchException = ex; // <10> 记录异常
            } 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); // <10> 记录异常
            }

         // <11> 处理正常和异常的请求调用结果。
            processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
        } catch (Exception ex) {
         // <12> 已完成 拦截器
            triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
        } catch (Throwable err) {
         // <12> 已完成 拦截器
            triggerAfterCompletion(processedRequest, response, mappedHandler,
                    new NestedServletException("Handler processing failed", err));
        } finally {
         // <13.1> TODO 芋艿，干啥子？
         if (asyncManager.isConcurrentHandlingStarted()) {
             // Instead of postHandle and afterCompletion
             if (mappedHandler != null) {
                    mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
                }
            } else {
             // <13.2> Clean up any resources used by a multipart request.
             if (multipartRequestParsed) {
                    cleanupMultipart(processedRequest);
                }
            }
        }
    }
```

<1> 处，TODO 1003 Async

<2> 处，调用 `#checkMultipart(HttpServletRequest request)` 方法，检查是否是上传请求。如果是，则封装成 MultipartHttpServletRequest 对象。详细解析，见 《精尽 Spring MVC 源码解析 —— MultipartResolver》 中。

<3> 处，调用 `#getHandler(HttpServletRequest request)` 方法，返回请求对应的是 HandlerExecutionChain 对象，它包含处理器（ handler ）和拦截器们（ HandlerInterceptor 数组 ）。代码如下：

```
    // DispatcherServlet.java

    /** List of HandlerMappings used by this servlet. */
    @Nullable
    private List<HandlerMapping> handlerMappings;

    @Nullable
    protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
     if (this.handlerMappings != null) {
         // 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);
            // 获得到，则返回
            if (handler != null) {
                return handler;
               }
           }
       }
     return null;
    }
```

- 详细的解析，见 《精尽 Spring MVC 源码分析 —— HandlerMapping 组件（二）之 HandlerInterceptor》

<3.1> 处，如果获取不到，则调用 #noHandlerFound(HttpServletRequest request, HttpServletResponse response) 根据配置抛出异常或返回 404 错误。 代码比较简单，胖友点击 传送门 自己看该方法。

<4> 处，调用 #getHandlerAdapter(Object handler) 方法，获得当前 handler 对应的 HandlerAdapter 对象。代码如下：

```
// DispatcherServlet.java
/** List of HandlerAdapters used by this servlet. */
@Nullable
private List<HandlerAdapter> handlerAdapters;

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)) {
             // 如果支持，则返回
             return adapter;
            }
        }
    }
 // 没找到对应的 HandlerAdapter 对象，抛出 ServletException 异常
 throw new ServletException("No adapter for handler [" + handler +
         "]: The DispatcherServlet configuration needs to include a HandlerAdapter that supports this handler");
}
```

- 详细解析，见 《精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（一）之 HandlerAdapter》

<4.1> 处，见 《精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（一）之 HandlerAdapter》

【拦截器】<5> 处，调用 HandlerExecutionChain#applyPreHandle(HttpServletRequest request, HttpServletResponse response) 方法，拦截器的前置处理，即调用 HandlerInterceptor#preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) 方法。详细解析，见 《精尽 Spring MVC 源码分析 —— HandlerMapping 组件（二）之 HandlerInterceptor》 。

【Controller】<6> 处，调用 HandlerAdapter#handle(HttpServletRequest request, HttpServletResponse response, Object handler) 方法，真正的调用 handler 方法，并返回视图。这里，一般就会调用我们定义的 Controller 的方法。详细解析，见 TODO 。

<7> 处，TODO 1003 Asyn

<8> 处，调用 #applyDefaultViewName(HttpServletRequest request, ModelAndView mv) 方法，当无视图的情况下，设置默认视图。代码如下：

```
// DispatcherServlet.java

/** RequestToViewNameTranslator used by this servlet. */
@Nullable
private RequestToViewNameTranslator viewNameTranslator;

private void applyDefaultViewName(HttpServletRequest request, @Nullable ModelAndView mv) throws Exception {
 if (mv != null && !mv.hasView()) { // 无视图
     // 获得默认视图
        String defaultViewName = getDefaultViewName(request);
     // 设置默认视图
     if (defaultViewName != null) {
            mv.setViewName(defaultViewName);
        }
    }
}

@Nullable
protected String getDefaultViewName(HttpServletRequest request) throws Exception {
 // 从请求中，获得视图
 return (this.viewNameTranslator != null ? this.viewNameTranslator.getViewName(request) : null);
}
```

- 详细解析，见 《精尽 Spring MVC 源码解析 —— RequestToViewNameTranslator》 中。

【拦截器】<9> 处，调用 HandlerExecutionChain#applyPostHandle(HttpServletRequest request, HttpServletResponse response, ModelAndView mv) 方法，拦截器的后置处理，即调用 HandlerInterceptor#postHandle(HttpServletRequest request, HttpServletResponse response, Object handler) 方法。详细解析，见 《精尽 Spring MVC 源码分析 —— HandlerMapping 组件（二）之 HandlerInterceptor》 。

<10> 处，记录异常。注意，此处仅仅记录，不会抛出异常，而是统一交给 <11> 处理。
<11> 处，调用 #processDispatchResult(HttpServletRequest request, HttpServletResponse response, HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception) 方法，处理正常和异常的请求调用结果。注意，正常的、异常的，都会进行处理。详细解析，见 「3.3 processDispatchResult」 。
【拦截器】<12> 处，调用 #triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, HandlerExecutionChain mappedHandler, Exception ex) 方法，拦截器的已完成处理，即调用 HandlerInterceptor#triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, Exception ex) 方法。详细解析，见 《精尽 Spring MVC 源码分析 —— HandlerMapping 组件（二）之 HandlerInterceptor》 。
<13.1> 处，TODO 1003 Asyn
<13.2> 处，如果是上传请求，则调用 #cleanupMultipart(HttpServletRequest request) 方法，清理资源。详细解析，见 《精尽 Spring MVC 源码解析 —— MultipartResolver》 中。

3.3 processDispatchResult

#processDispatchResult(HttpServletRequest request, HttpServletResponse response, HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception) 方法，处理正常和异常的请求调用结果。代码如下：

```
// DispatcherServlet.java

private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
        @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv,
        @Nullable Exception exception) throws Exception {
 // <1> 标记，是否是生成的 ModelAndView 对象
 boolean errorView = false;

 // <2> 如果是否异常的结果
 if (exception != null) {
        // 情况一，从 ModelAndViewDefiningException 中获得 ModelAndView 对象
     if (exception instanceof ModelAndViewDefiningException) {
            logger.debug("ModelAndViewDefiningException encountered", exception);
            mv = ((ModelAndViewDefiningException) exception).getModelAndView();
     // 情况二，处理异常，生成 ModelAndView 对象
        } else {
            Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);
            mv = processHandlerException(request, response, handler, exception);
         // 标记 errorView
            errorView = (mv != null);
        }
    }

 // Did the handler return a view to render?
 if (mv != null && !mv.wasCleared()) {
     // <3.1> 渲染页面
        render(mv, request, response);
     // <3.2> 清理请求中的错误消息属性
     if (errorView) {
            WebUtils.clearErrorRequestAttributes(request);
        }
    } else {
     if (logger.isTraceEnabled()) {
            logger.trace("No view rendering, null ModelAndView returned.");
        }
    }

 // <4> TODO 芋艿
 if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
     // Concurrent handling started during a forward
     return;
    }

 // <5> 已完成处理 拦截器
 if (mappedHandler != null) {
        mappedHandler.triggerAfterCompletion(request, response, null);
    }
}
```

<1> 处，`errorView` 属性，标记是否是生成的 ModelAndView 对象。

<2> 处，如果是否异常的结果。

- <2.1> 处，情况一，从 ModelAndViewDefiningException 中获得 ModelAndView 对象。
- <2.2> 处，情况二，调用 `#processHandlerException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)` 方法，处理异常，生成 ModelAndView 对象。代码如下：

```
// DispatcherServlet.java

@Nullable
protected ModelAndView processHandlerException(HttpServletRequest request, HttpServletResponse response,
        @Nullable Object handler, Exception ex) throws Exception {
 // Success and error responses may use different content types
    // 移除 PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE 属性
    request.removeAttribute(HandlerMapping.PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE);

 // Check registered HandlerExceptionResolvers...
 // <a> 遍历 HandlerExceptionResolver 数组，解析异常，生成 ModelAndView 对象
    ModelAndView exMv = null;
 if (this.handlerExceptionResolvers != null) {
     // 遍历 HandlerExceptionResolver 数组
     for (HandlerExceptionResolver resolver : this.handlerExceptionResolvers) {
         // 解析异常，生成 ModelAndView 对象
            exMv = resolver.resolveException(request, response, handler, ex);
            // 生成成功，结束循环
         if (exMv != null) {
             break;
            }
        }
    }
 // 情况一，生成了 ModelAndView 对象，进行返回
 if (exMv != null) {
     // ModelAndView 对象为空，则返回 null
     if (exMv.isEmpty()) {
            request.setAttribute(EXCEPTION_ATTRIBUTE, ex); // 记录异常到 request 中
         return null;
        }
     // We might still need view name translation for a plain error model...
     // 设置默认视图
     if (!exMv.hasView()) {
            String defaultViewName = getDefaultViewName(request);
         if (defaultViewName != null) {
                exMv.setViewName(defaultViewName);
            }
        }
     // 打印日志
     if (logger.isTraceEnabled()) {
            logger.trace("Using resolved error view: " + exMv, ex);
        }
     if (logger.isDebugEnabled()) {
            logger.debug("Using resolved error view: " + exMv);
        }
     // 设置请求中的错误消息属性
        WebUtils.exposeErrorRequestAttributes(request, ex, getServletName());
     return exMv;
    }
 // 情况二，未生成 ModelAndView 对象，则抛出异常
 throw ex;
}
```

- <a> 处，遍历 HandlerExceptionResolver 数组，调用 HandlerExceptionResolver#resolveException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) 方法，解析异常，生成 ModelAndView 对象。详细解析，TODO
- <b> 处，情况一，生成了 ModelAndView 对象，进行返回。当然，这里的后续代码还有 10 多行，比较简单，胖友自己瞅瞅就 OK 啦。

    - <c> 处，情况二，未生成 ModelAndView 对象，则抛出异常。

<3.1> 处，调用 #render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response) 方法，渲染页面。详细解析，见 「3.4 render」 。

<3.2> 处，当是 <2> 处的情况二时，则调用 WebUtils#clearErrorRequestAttributes(HttpServletRequest request) 方法，清理请求中的错误消息属性。为什么会有这一步呢？答案在 #processHandlerException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) 方法中，会调用 WebUtils#exposeErrorRequestAttributes(HttpServletRequest request, Throwable ex, String servletName) 方法，设置请求中的错误消息属性。

<4> 处，TODO 1003 芋艿

【拦截器】<5> 处，调用 #triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, HandlerExecutionChain mappedHandler, Exception ex) 方法，拦截器的已完成处理，即调用 HandlerInterceptor#triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, Exception ex) 方法。详细解析，见 《精尽 Spring MVC 源码分析 —— HandlerMapping 组件（二）之 HandlerInterceptor》 。

3.4 render

#render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response) 方法，渲染 ModelAndView 。代码如下：

```
// DispatcherServlet.java

protected void render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response) throws Exception {
 // Determine locale for request and apply it to the response.
 // <1> TODO 芋艿 从 request 中获得 Locale 对象，并设置到 response 中
    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();
 // 情况一，使用 viewName 获得 View 对象
 if (viewName != null) {
     // We need to resolve the view name.
     // <2.1> 使用 viewName 获得 View 对象
        view = resolveViewName(viewName, mv.getModelInternal(), locale, request);
     if (view == null) { // 获取不到，抛出 ServletException 异常
         throw new ServletException("Could not resolve view with name '" + mv.getViewName() +
                 "' in servlet with name '" + getServletName() + "'");
        }
 // 情况二，直接使用 ModelAndView 对象的 View 对象
    } else {
     // No need to lookup: the ModelAndView object contains the actual View object.
     // 直接使用 ModelAndView 对象的 View 对象
        view = mv.getView();
     if (view == null) { // 获取不到，抛出 ServletException 异常
         throw new ServletException("ModelAndView [" + mv + "] neither contains a view name nor a " +
                 "View object in servlet with name '" + getServletName() + "'");
        }
    }

 // Delegate to the View object for rendering.
 // 打印日志
 if (logger.isTraceEnabled()) {
        logger.trace("Rendering view [" + view + "] ");
    }
```

```
try {
    // <3> 设置响应的状态码
    if (mv.getStatus() != null) {
        response.setStatus(mv.getStatus().value());
    }
    // <4> 渲染页面
    view.render(mv.getModelInternal(), request, response);
} catch (Exception ex) {
    if (logger.isDebugEnabled()) {
        logger.debug("Error rendering view [" + view + "]", ex);
    }
    throw ex;
}
```

<1> 处，调用 LocaleResolver#resolveLocale(HttpServletRequest request) 方法，从 request 中获得 Locale 对象，并设置到 response 中。详细解析，见 TODO 1001
<2> 处，获得 View 对象。分成两种情况，代码比较简单，胖友自己瞅瞅。

<2.1> 处，调用 #resolveViewName(String viewName, Map<String, Object> model, Locale locale, HttpServletRequest request) 方法，使用 viewName 获得 View 对象。代码如下：

```
// DispatcherServlet.java

@Nullable
protected View resolveViewName(String viewName, @Nullable Map<String, Object> model,
        Locale locale, HttpServletRequest request) throws Exception {
    if (this.viewResolvers != null) {
        // 遍历 ViewResolver 数组
        for (ViewResolver viewResolver : this.viewResolvers) {
            // 根据 viewName + locale 参数，解析出 View 对象
            View view = viewResolver.resolveViewName(viewName, locale);
            // 解析成功，直接返回 View 对象
            if (view != null) {
                return view;
            }
        }
    }
    // 返回空
    return null;
}
```

- 详细解析，见 《精尽 Spring MVC 源码解析 —— ViewResolver》

<3> 处，设置响应的状态码。
<4> 处，调用 View#render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response) 方法，渲染视图。详细解析，见 《精尽 Spring MVC 源码解析 —— ViewResolver》 。

666. 彩蛋

到此，我们已经对 DispatcherServlet 是如何处理请求已经有了整体的认识。当然，我们对每个 Spring MVC 组件，细节暂时没有进行深扣，正如我们在看本文，会有大量的 TODO 。不要方，后面的每一篇，我们会对每个 Spring MVC 组件逐个解析。这样，我们对 Spring MVC 会更加了解。

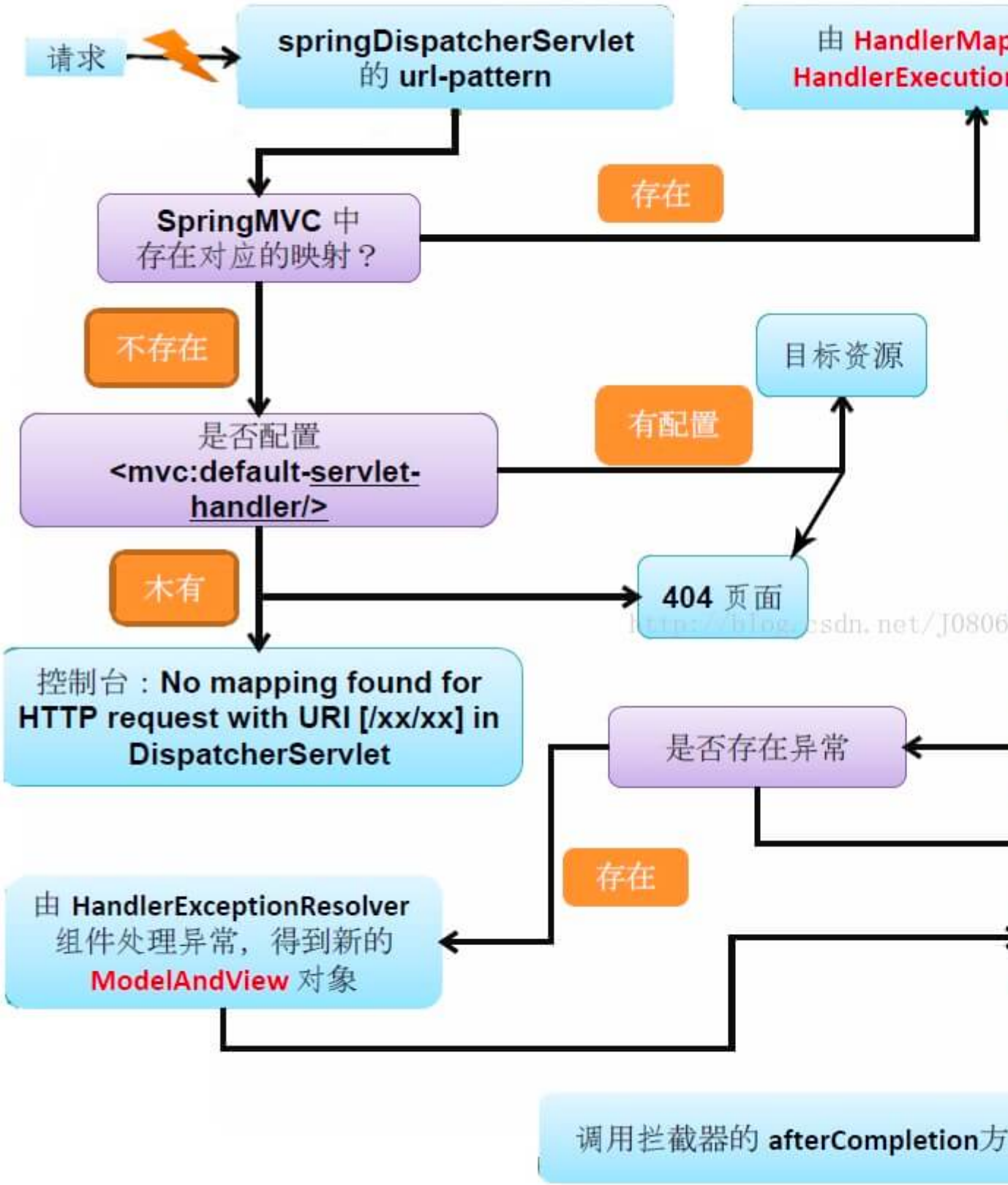
其实，这也是看源码的套路，先整理，后局部，逐步逐步抽丝剥茧，看清理透。

还有，在看完后续的 Spring MVC 的每个组件后，胖友可以在回过头在重新看看这个文章，是否能够将文章更好的串联在一起。

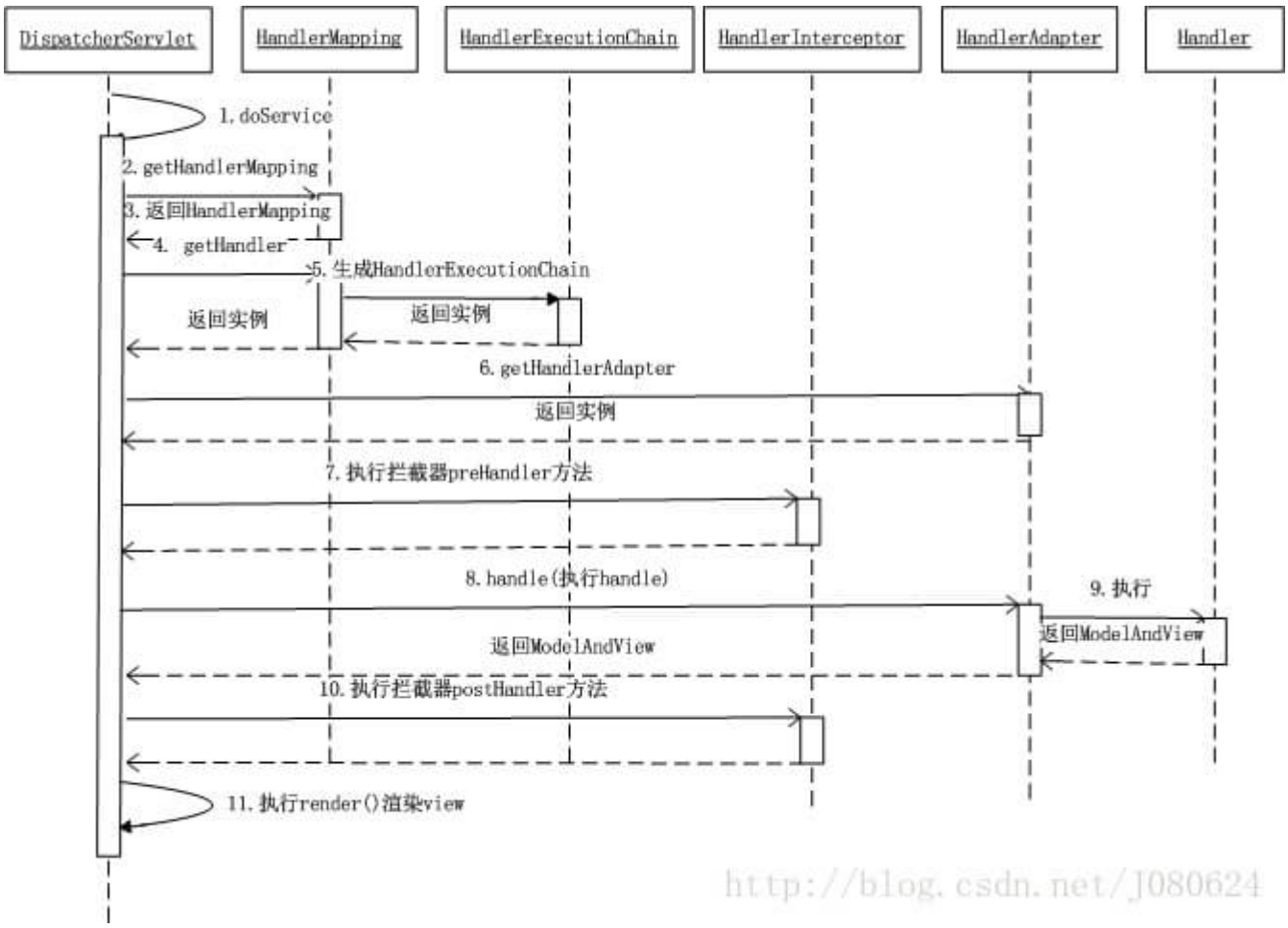
下面，艿艿整理了一些网络上讲述 Spring MVC 处理请求的一些图，帮助我们更好的理解这个过程。

FROM 《SpringMVC - 运行流程图及原理分析》

流程示意图：

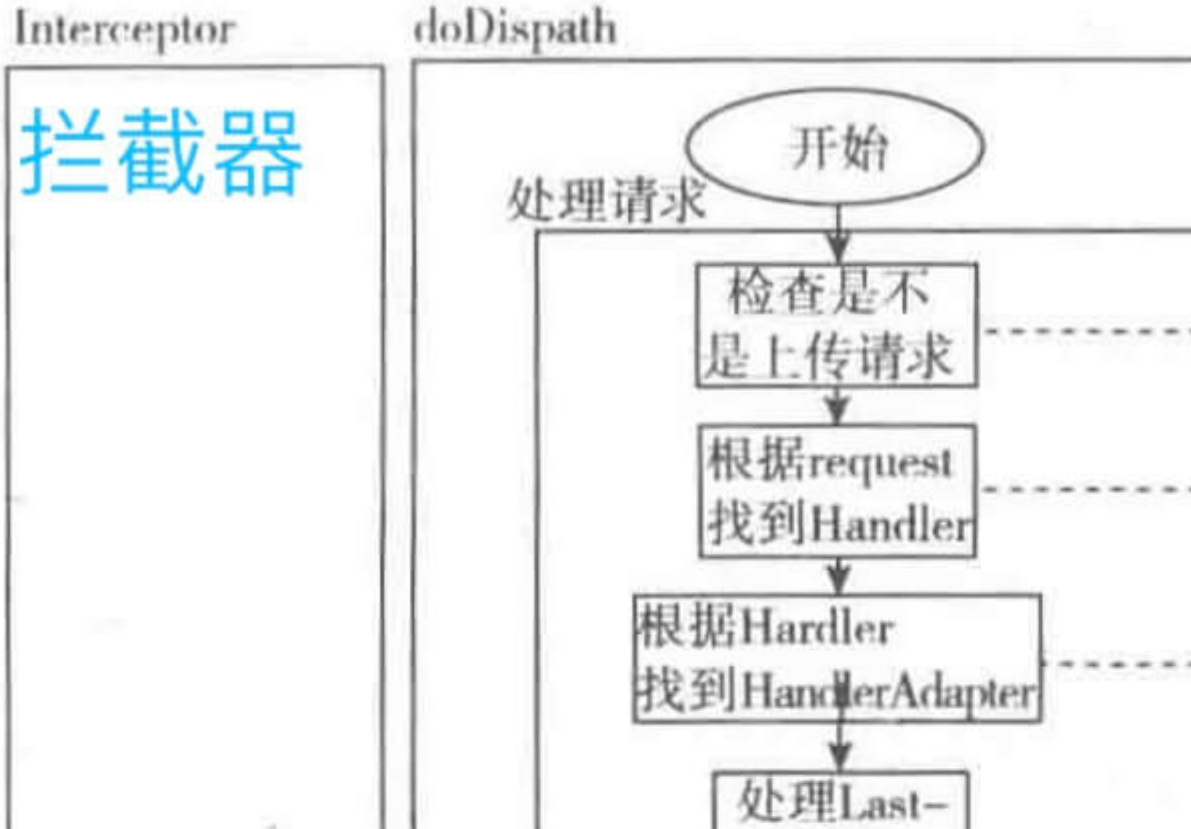


代码序列图：



FROM 《看透 Spring MVC：源代码分析与实践》 P123

流程示意图：



参考和推荐如下文章：

> 田小波 《Spring MVC 原理探秘 - 一个请求的旅行过程》
> 郝佳 《Spring 源码深度解析》 的 「11.3 DispatcherServlet」 小节
> 韩路彪 《看透 Spring MVC：源代码分析与实践》 的 「第10章 Spring MVC 之用」 小节

文章目录