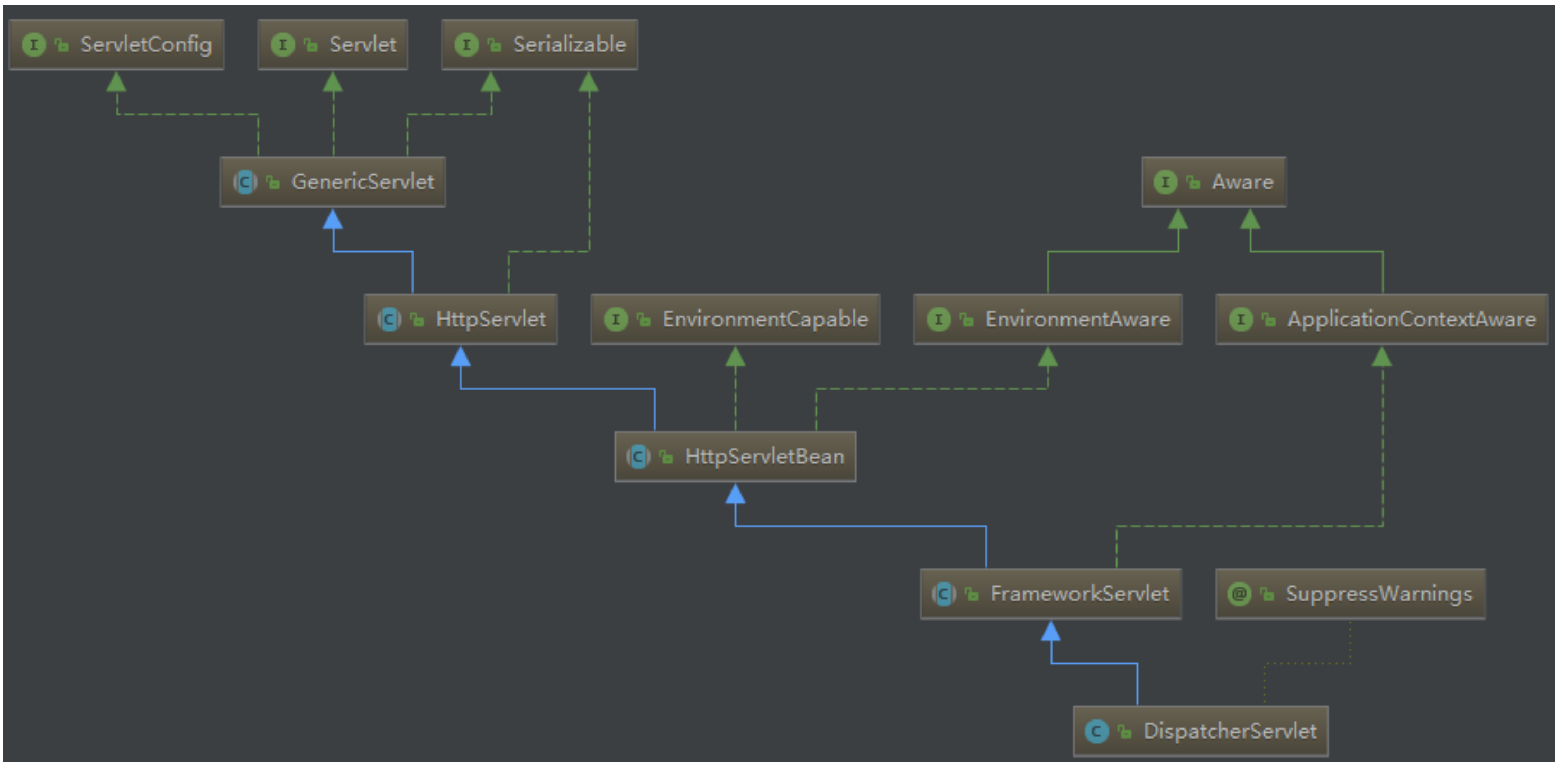
DispatcherServlet是SpringMVC的核心分发器，它实现了请求分发，是处理请求的入口，本篇将深入源码分析它的请求分发过程。

进入主题前，回顾一下DispatcherServlet的继承关系图。



Servlet在service方法中进行请求接收与分发，DispatcherServlet的service方法继承自HttpServlet，具体代码如下图所示。

```
public void service(ServletRequest req, ServletResponse res)
throws ServletException, IOException
{
HttpServletRequest  request;
HttpServletResponse response;

try {
    request = (HttpServletRequest) req;
    response = (HttpServletResponse) res;
} catch (ClassCastException e) {
    throw new ServletException("non-HTTP request or response");
}
service(request, response);
}
```

调用内部的service方法

```
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) {
    //...
    doGet(req, resp);
    } else {
    long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
    if (ifModifiedSince < (lastModified / 1000 * 1000)) {
        //...
        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 {
```

获取请求方式

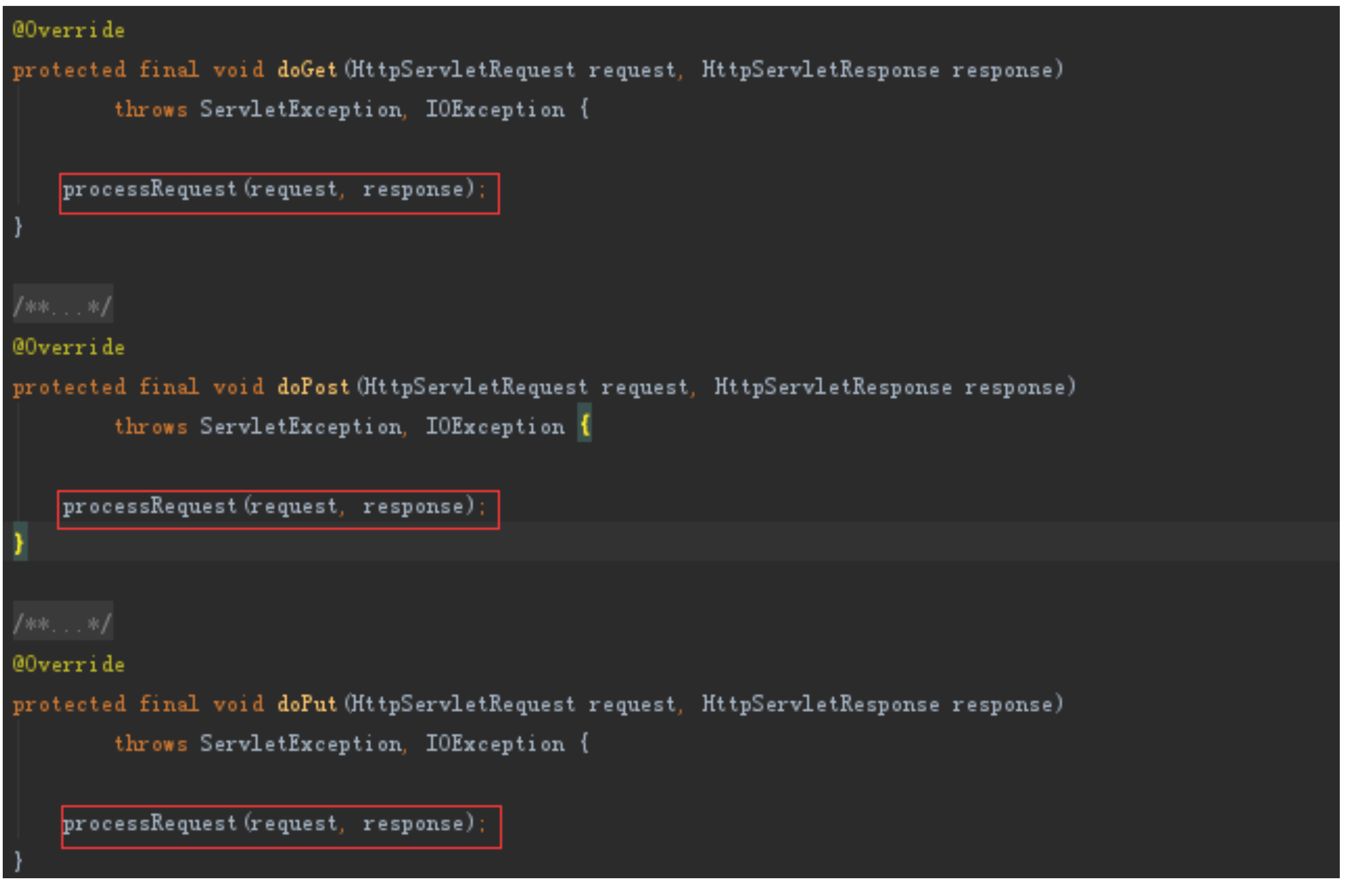
在FrameworkServlet中对这个protected修饰的service方法进行了重写，重写的目的是支持PATCH方式请求，具体代码如下图所示。

```
protected void service(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    String method = request.getMethod();
    if (method.equalsIgnoreCase(RequestMethod.PATCH.name())) {
        processRequest(request, response);
    }
    else {
        super.service(request, response);
    }
}
```

支持PATCH请求

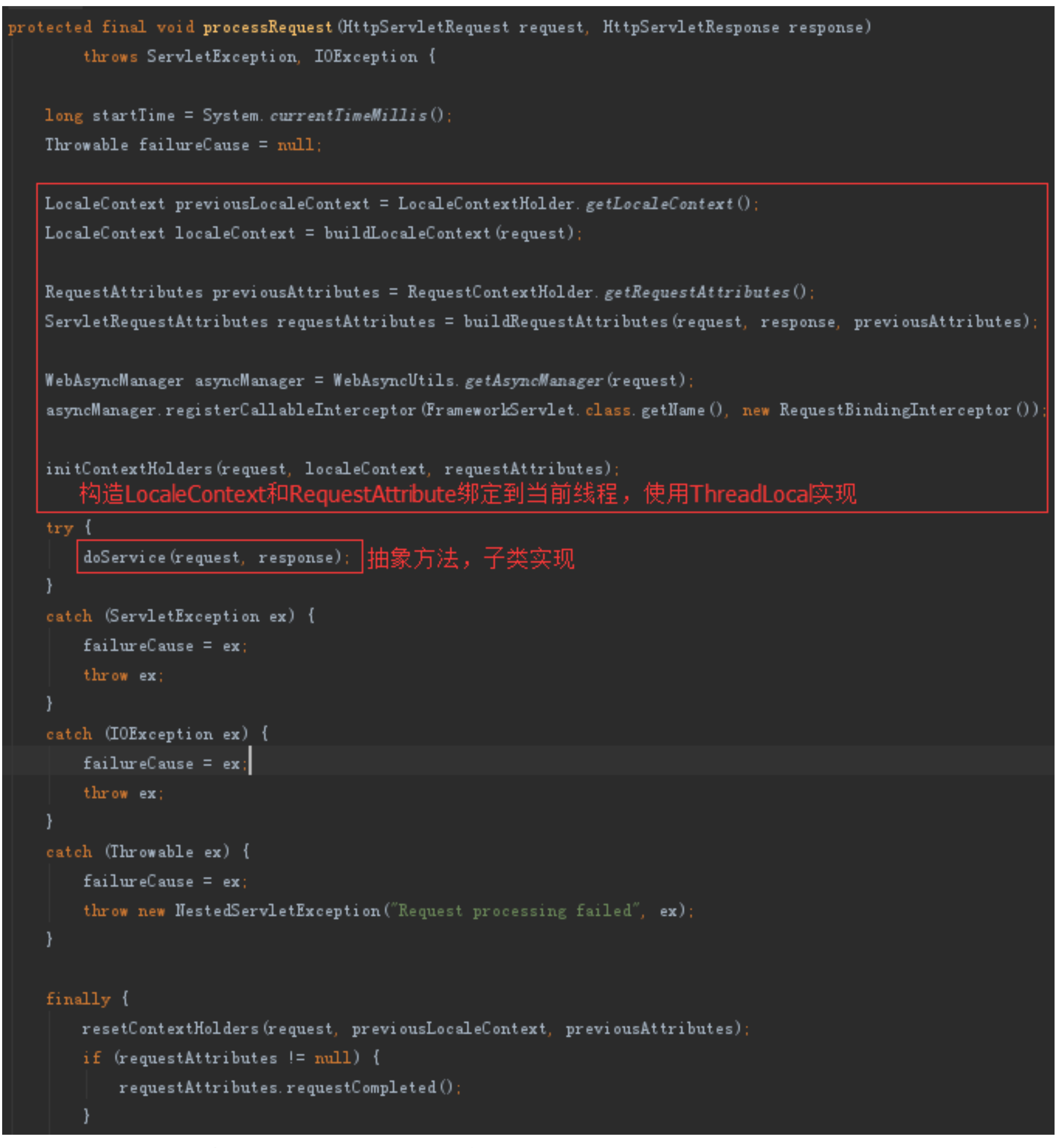
调用HttpServlet中的service方法

上述分析中的doGet、doPost等方法在HttpServlet中没有实际可用的实现，如果要使用这些方法，子类需要重写这些方法，DispatcherServlet没有重写这些方法，在DispatcherServlet的父类FrameworkServlet中进行了重写，看几个重写后的方法代码。



```
@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);
}
```

可以看到这些请求都会进入当前FrameworkServlet类的processRequest方法进行处理，具体代码如下图所示。

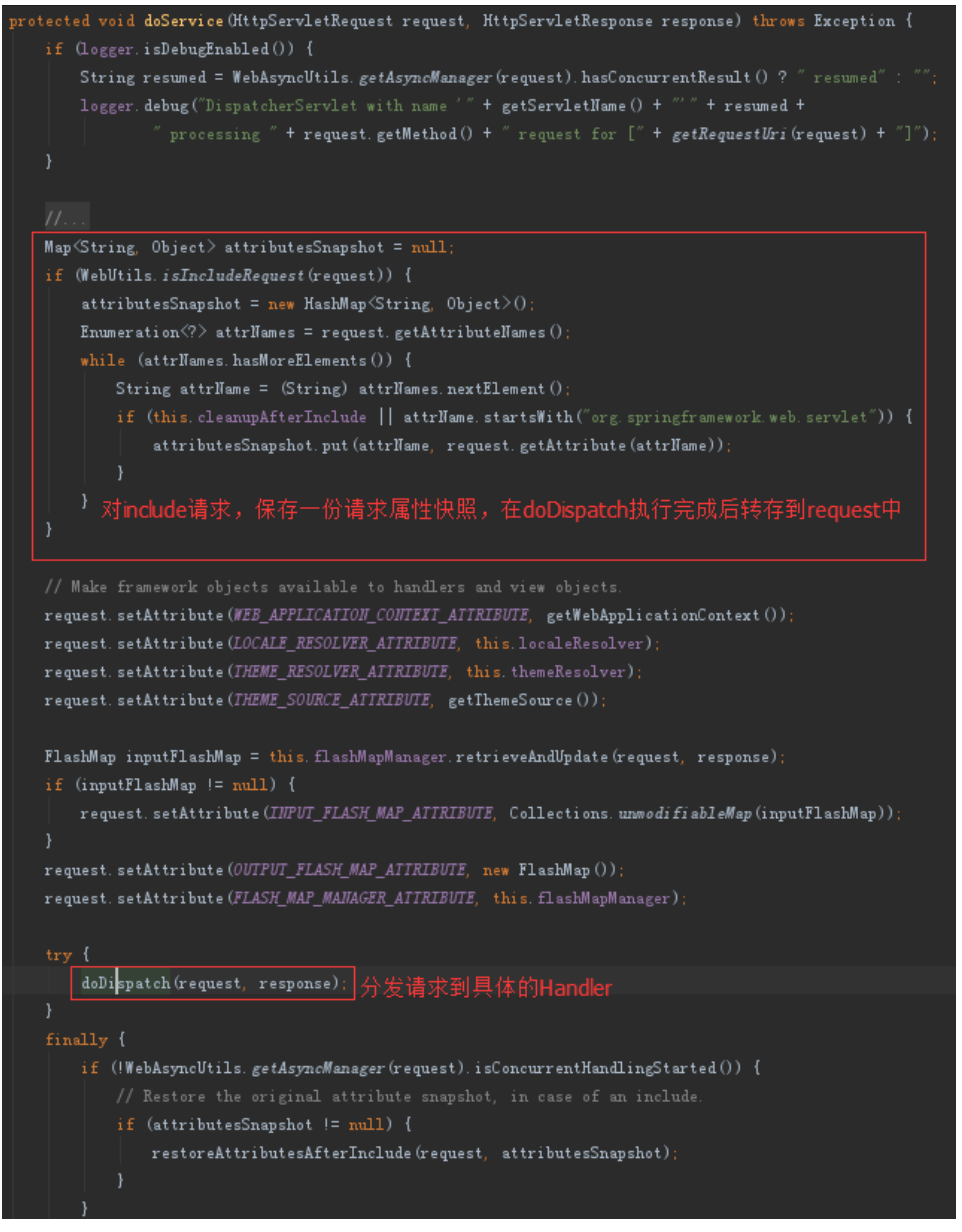


```
protected final void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    long startTime = System.currentTimeMillis();
    Throwable failureCause = null;

    LocaleContext previousLocaleContext = LocaleContextHolder.getLocaleContext();
    LocaleContext localeContext = buildLocaleContext(request);

    RequestAttributes previousAttributes = RequestContextHolder.getRequestAttributes();
    ServletRequestAttributes requestAttributes = buildRequestAttributes(request, response, previousAttributes);

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
    asyncManager.registerCallableInterceptor(FrameworkServlet.class.getName(), new RequestBindingInterceptor());

    initContextHolders(request, localeContext, requestAttributes);
    try {
        doService(request, response);
    }
    catch (ServletException ex) {
        failureCause = ex;
        throw ex;
    }
    catch (IOException ex) {
        failureCause = ex;
        throw ex;
    }
    catch (Throwable ex) {
        failureCause = ex;
        throw new NestedServletException("Request processing failed", ex);
    }

    finally {
        resetContextHolders(request, previousLocaleContext, previousAttributes);
        if (requestAttributes != null) {
            requestAttributes.requestCompleted();
        }
```

构造LocaleContext和RequestAttribute绑定到当前线程，使用ThreadLocal实现

抽象方法，子类实现

FrameworkServlet中的doService是一个抽象方法，DispatcherServlet重写了这个方法，具体代码如下图。



```
protected void doService(HttpServletRequest request, HttpServletResponse response) throws Exception {
    if (logger.isDebugEnabled()) {
        String resumed = WebAsyncUtils.getAsyncManager(request).hasConcurrentResult() ? " resumed" : "";
        logger.debug("DispatcherServlet with name '" + getServletName() + "'" + resumed +
                " processing " + request.getMethod() + " request for [" + getRequestUri(request) + "]");
    }

    //...
    Map<String, Object> attributesSnapshot = null;
    if (WebUtils.isIncludeRequest(request)) {
        attributesSnapshot = new HashMap<String, Object>();
        Enumeration<?> attrNames = request.getAttributeNames();
        while (attrNames.hasMoreElements()) {
            String attrName = (String) attrNames.nextElement();
            if (this.cleanupAfterInclude || attrName.startsWith("org.springframework.web.servlet")) {
                attributesSnapshot.put(attrName, request.getAttribute(attrName));
            }
        }
    }

    // Make framework objects available to handlers and view objects.
    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());

    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 {
        if (!WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
            // Restore the original attribute snapshot, in case of an include.
            if (attributesSnapshot != null) {
                restoreAttributesAfterInclude(request, attributesSnapshot);
            }
        }
```

对include请求，保存一份请求属性快照，在doDispatch执行完成后转存到request中

分发请求到具体的Handler

进入doDispatch方法，这个方法实现了将请求分发到具体Handler、执行拦截器的preHandle方法、调用Handler（编写的Controller）处理具体逻辑、执行拦截器的postHandle方法、处理返回的ModelAndView或异常、执行拦截器的afterCompletion方法，具体代码如下。

```
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);

            // Determine handler for the current request.
            mappedHandler = getHandler(processedRequest);
            if (mappedHandler == null || mappedHandler.getHandler() == 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 (logger.isDebugEnabled()) {
                    logger.debug("Last-Modified value for [" + getRequestUri(request) + "] is: " + lastModified);
                }
                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());
```

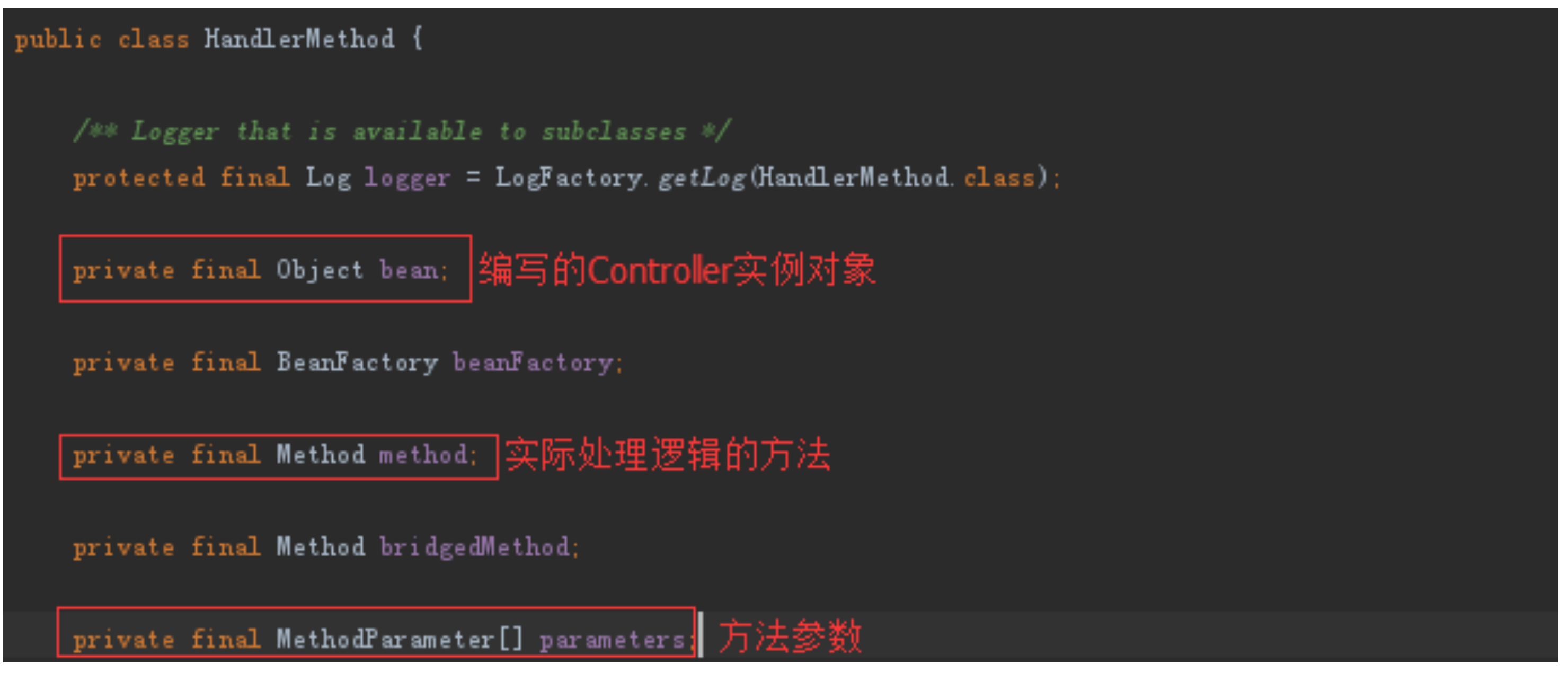
根据请求的url查找实际处理逻辑的Handler，这个Handler被封装在HandlerMethod中，这个HandlerMethod又被封装在HandlerExcutionChain中

遍历HandlerAdapter集合，查找支持这次请求的HandlerAdapter

执行拦截器的preHandle方法，如果有一个拦截器的preHandle方法执行失败，将会执行afterCompletion ，然后直接返回

调用实际编写的Handler处理请求

上图描述中的HandlerMethod和HandlerExecutionChain代码如下所示。



```
public class HandlerMethod {

    /** Logger that is available to subclasses */
    protected final Log logger = LogFactory.getLog(HandlerMethod.class);

    private final Object bean;

    private final BeanFactory beanFactory;

    private final Method method;

    private final Method bridgedMethod;

    private final MethodParameter[] parameters;
```

编写的Controller实例对象

实际处理逻辑的方法

方法参数

```
public class HandlerExecutionChain {

    private static final Log logger = LogFactory.getLog(HandlerExecutionChain.class);

    private final Object handler;

    private HandlerInterceptor[] interceptors;

    private List<HandlerInterceptor> interceptorList;

    private int interceptorIndex = -1;
```

HandlerMethod实例

拦截器们

总结：
首先，SpringMVC框架在启动的时候会遍历Spring容器中的所有bean，对标注了@Controller或@RequestMapping注解的类中方法进行遍历，将类和方法上的@RequestMapping注解值进行合并，使用@RequestMapping注解的相关参数值(如value、method等)封装一个RequestMappingInfo，将这个Controller实例、方法及方法参数信息(类型、注解等)封装到HandlerMethod中，然后以RequestMappingInfo为key，HandlerMethod为value存到一个以Map为结构的handlerMethods中。

接着，将@RequestMapping注解中的value(即请求路径)值取出，即url，然后以url为key，以RequestMappingInfo为value，存到一个以Map为结构的urlMap属性中。

客户端发起请求的时候，根据请求的URL到urlMap中查找，找到RequestMappingInfo，然后根据RequestMappingInfo到handlerMethods中查找，找到对应的HandlerMethod，接着将HandlerMethod封装到HandlerExecutionChain；接着遍历容器中所有HandlerAdapter实现类，找到支持这次请求的HandlerAdapter，如RequestMappingHandlerAdapter，然后执行SpringMVC拦截器的前置方法(preHandle方法)，然后对请求参数解析及转换，然后(使用反射)调用具体Controller的对应方法返回一个ModelAndView对象，执行拦截器的后置方法(postHandle方法)，然后对返回的结果进行处理，最后执行afterCompletion方法。