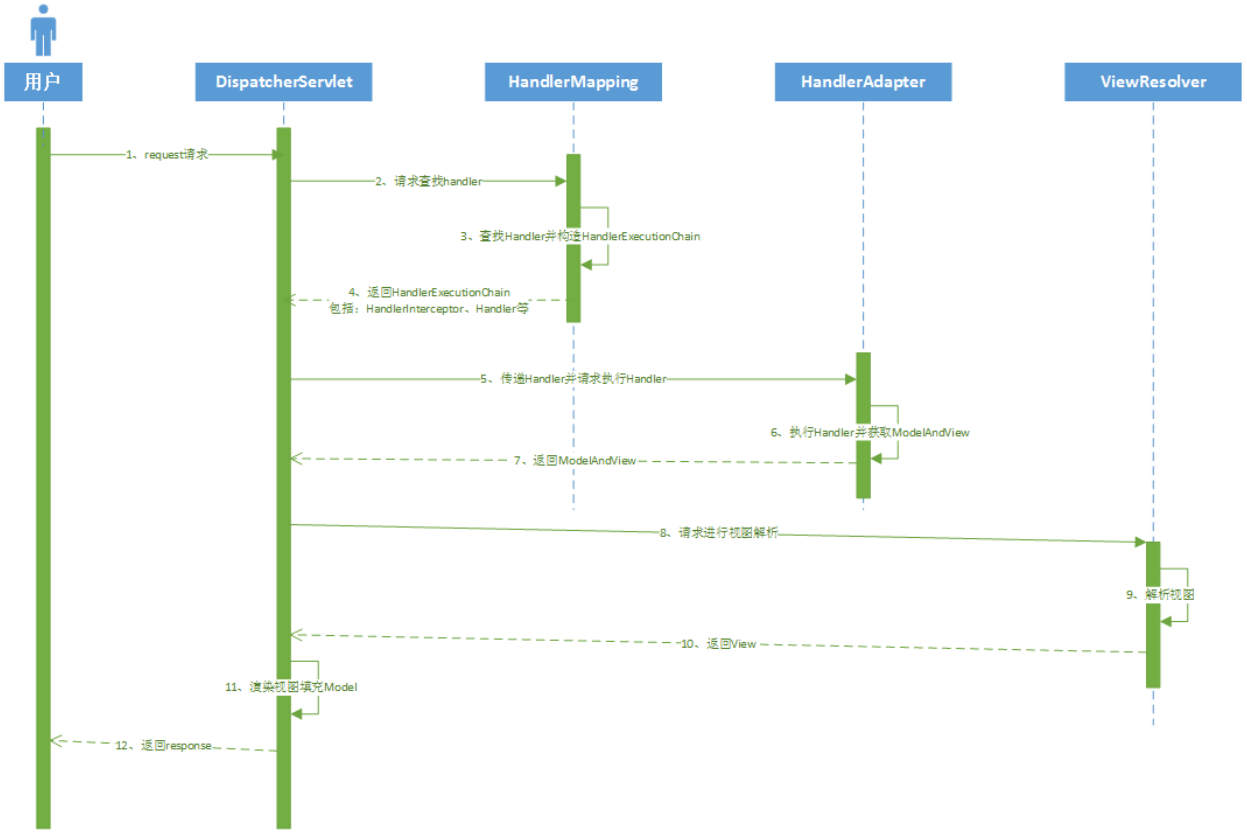
经过前面的启动流程分析，我们也大致清楚了Spring MVC的构成，DispatcherServlet最为重要，然后管理各种组件，比较重要的三个，HandlerMapping，HandlerAdapter，ViewResolver。

该分支我们分析DispatcherServlet 的执行流程，相当于分析了Spring MVC的执行流程，看一下这之中的过程。由于主要的执行过程就涉及到DispatcherServlet 和以上三个组件，我们围绕这个做了一个时序图。



这张图也比较清晰展示了整个流程，我们通过文字一步步解释一下。

1.用户发送request请求，DispatcherServlet 类接收；

2.DispatcherServlet 类遍历所有配置的HandlerMapping类请求查询Handler；

3.HandlerMapping类根据请求的URL信息查到对应的Handler，以及相关的拦截器interceptor并构造HandlerExecutionChain；

4.将构造的HandlerExecutionChain对象（里面包含了Handler和所有相关拦截器）等返回给DispatcherServlet ；

5.DispatcherServlet遍历所有的HandlerAdapter类并传递Handler请求执行Handler；

6.HandlerAdapter类执行Handler并获取ModelAndView类对象；

7.HandlerAdapter类将ModelAndView对象返回给DispatcherServlet；

8.DispatcherServlet遍历所有的ViewResolver类请求进行视图解析；

9.ViewResolver类进行视图解析并获取View对象；

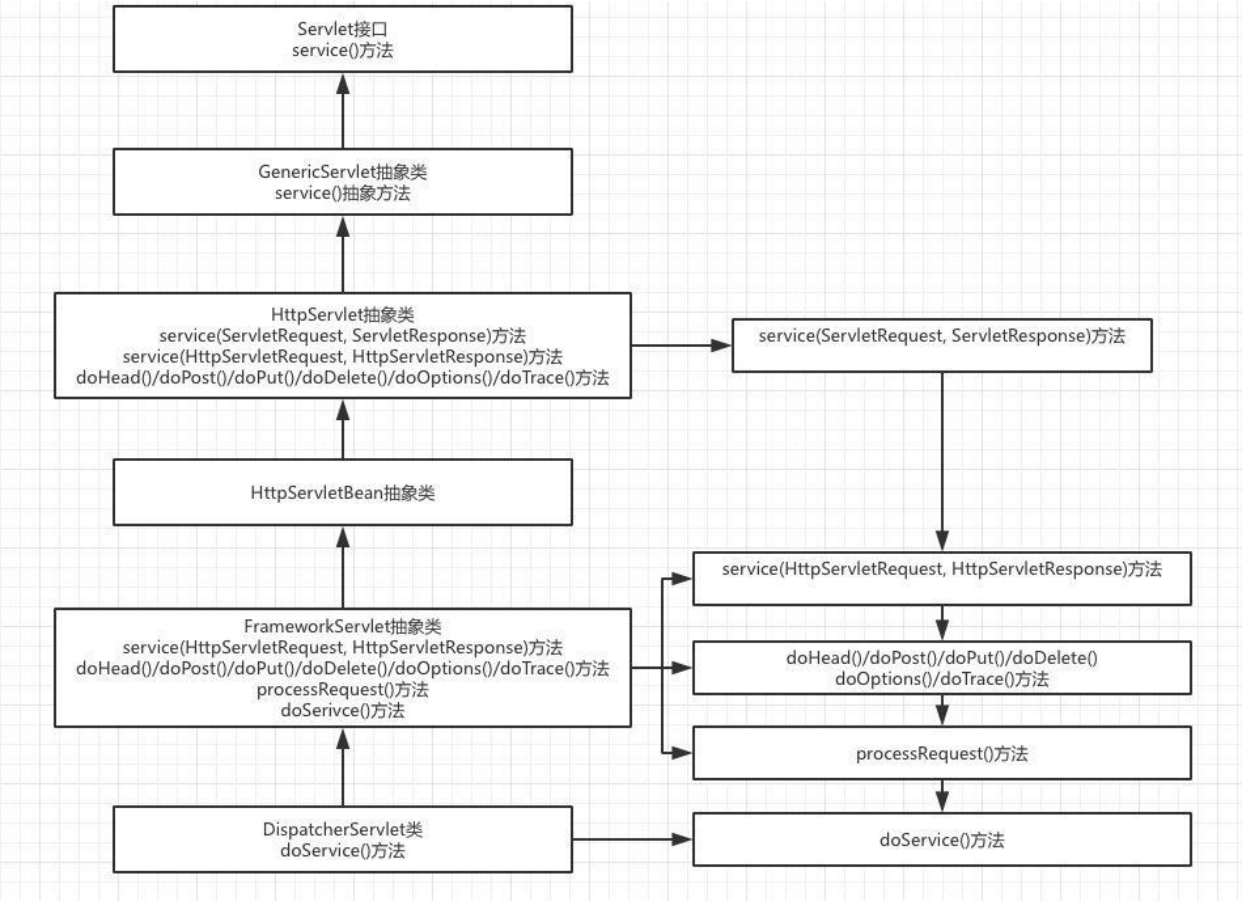
10.ViewResolver向DispatcherServlet返回一个View对象；

11.DispatcherServlet进行视图的渲染填充model；

12.DispatcherServlet向用户返回响应；

整个流程下来大家发现，这和Spring MVC的执行流程不是一样的吗，没错，DispatcherServlet的就是Spring MVC的执行核心，所有东西都围绕其进行。那么就让我们走进DispatcherServlet详细解读执行过程吧。

对用户请求响应本质上是调用了Servlet的service方法，我们也知道DispatcherServlet是Service层层继承下来的具体实现类，我们上一张类图展示下这之中的联系。



鉴于过程中调用实现都比较简单，我们直奔主题（后期针对过程再补充吧），DispatcherServlet的doService方法。

```
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) + "]");
    }

    // 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<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);
    }
```

可以看到给request赋了几个值，最重要的事将当前Servlet的子IoC容器放入request请求中，由此，我们可以访问到当前IoC子容器以及根IoC容器中的Bean。

然后主要点就是doDispatch这个方法，request请求也算是封装完毕了，接下来的调用过程我们就能对比上面的时序图去看了。

```
    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());

                    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);
                    }
                }
            }
```

```
    }
```

我们通读下这个方法，发现了很多熟悉的名词，我们对比之前的步骤来看。

HttpServletRequest processedRequest = request;

该定义的局部变量可以看作第1步，获取用户request请求，当然这一步获取的request已经在上一个方法做过相应的赋值，比如装入子IoC容器。

mappedHandler = getHandler(processedRequest);

该取值，就完成了第2，3，4步，getHandler方法就是构造了HandlerExecutionChain对象。

```
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    for (HandlerMapping hm : this.handlerMappings) {
        if (logger.isTraceEnabled()) {
            logger.trace(
                    "Testing handler map [" + hm + "] in DispatcherServlet with name '" + getServletName() + "'");
        }
        HandlerExecutionChain handler = hm.getHandler(request);
        if (handler != null) {
            return handler;
        }
    }
    return null;
}
```

我们之前有提过DispatcherServlet会遍历HandlerMappings类就在这里体现，真正构造HandlerExecutionChain对象的是这里的调用的HandlerMapping的实现类AbstractHandlerMapping类的getHandler方法。具体实现就不放在该篇幅讲了。

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

该取值，就完成了第5步，getHandlerAdapter就取到了相应的HandlerAdapter。

```
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
    for (HandlerAdapter ha : this.handlerAdapters) {
        if (logger.isTraceEnabled()) {
            logger.trace("Testing handler adapter [" + ha + "]");
        }
        if (ha.supports(handler)) {
            return ha;
        }
    }
    throw new ServletException("No adapter for handler [" + handler +
            "]: The DispatcherServlet configuration needs to include a HandlerAdapter that supports
}
```

同样也是遍历的，supports方法就是判断当前的handler是否是属于这个HandlerAdapter。

```
if (!mappedHandler.applyPreHandle(processedRequest, response)) {
        return;
}
```

先插一段这个，这个是啥呢，回想以前的知识点，拦截器由三部分，pre，post和after分别对应处理方法的不同阶段，其中pre就是在处理方法之前执行的，而这段代码紧接着就是处理方法了。

```
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
```

该取值，就完成了第6，7步，handle就是去处理了。这里涉及到HandlerAdapter源码解读，就不在该篇幅解答了。

```
mappedHandler.applyPostHandle(processedRequest, response, mv);
```

接下来我们会看到这个，和上面的pre相呼应，在处理方法结束后会调用拦截器的post方法。

再往后我们就要把目光集中到processDispatchResult方法上了，这里我们把以上已经获得的HandlerExecutionChain对象和ModelAndView对象也作为参数参数进行后续操作。

```java
private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,
        HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception) throws Exception {

    boolean errorView = false;

    if (exception != null) {
        if (exception instanceof ModelAndViewDefiningException) {
            logger.debug("ModelAndViewDefiningException encountered", exception);
            mv = ((ModelAndViewDefiningException) exception).getModelAndView();
        }
        else {
            Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);
            mv = processHandlerException(request, response, handler, exception);
            errorView = (mv != null);
        }
    }

    // Did the handler return a view to render?
    if (mv != null && !mv.wasCleared()) {
        render(mv, request, response);
        if (errorView) {
            WebUtils.clearErrorRequestAttributes(request);
        }
    }
    else {
        if (logger.isDebugEnabled()) {
            logger.debug("Null ModelAndView returned to DispatcherServlet with name '" + getServletName() +
                    "': assuming HandlerAdapter completed request handling");
        }
    }

    if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {
        // Concurrent handling started during a forward
        return;
    }

    if (mappedHandler != null) {
        mappedHandler.triggerAfterCompletion(request, response, null);
    }
}
```

我们把目光聚焦到render方法。

```
protected void render(ModelAndView mv, HttpServletRequest request,
HttpServletResponse response) throws Exception {
        // Determine locale for request and apply it to the response.
        Locale locale = this.localeResolver.resolveLocale(request);
        response.setLocale(locale);

        View view;
        if (mv.isReference()) {
            // We need to resolve the view name.
            view = resolveViewName(mv.getViewName(), mv.getModelInternal(), locale,
request);
            if (view == null) {
                throw new ServletException("Could not resolve view with name '" +
mv.getViewName() +
                        "' in servlet with name '" + getServletName() + "'");
            }
        }
        else {
            // No need to lookup: the ModelAndView object contains the actual View
object.
            view = mv.getView();
            if (view == null) {
                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.
            if (logger.isDebugEnabled()) {
                logger.debug("Rendering view [" + view + "] in DispatcherServlet with name
'" + getServletName() + "'");
            }
            try {
                if (mv.getStatus() != null) {
                    response.setStatus(mv.getStatus().value());
                }
                view.render(mv.getModelInternal(), request, response);
            }
            catch (Exception ex) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Error rendering view [" + view + "] in DispatcherServlet
with name '" +
                                    getServletName() + "'", ex);
                }
                throw ex;
            }
        }
```

view = resolveViewName(mv.getViewName(), mv.getModelInternal(), locale, request);

该定义便是第8，9，10步。

```
protected View resolveViewName(String viewName, Map<String, Object> model, Locale locale,
        HttpServletRequest request) throws Exception {

    for (ViewResolver viewResolver : this.viewResolvers) {
        View view = viewResolver.resolveViewName(viewName, locale);
        if (view != null) {
            return view;
        }
    }
    return null;
}
```

遍历视图解析器们获取对应的View对象。

view.render(mv.getModelInternal(), request, response);

该定义算是上面提过的流程的第11，12步，进行了视图的渲染，并最终返回视图响应用户。