

课程目标

- 1、通过分析 Spring 源码，深刻掌握核心原理和设计思想。
- 2、通过本课的学习，完全掌握 Spring MVC 的重要细节。
- 3、手绘 Spring MVC 时序图。

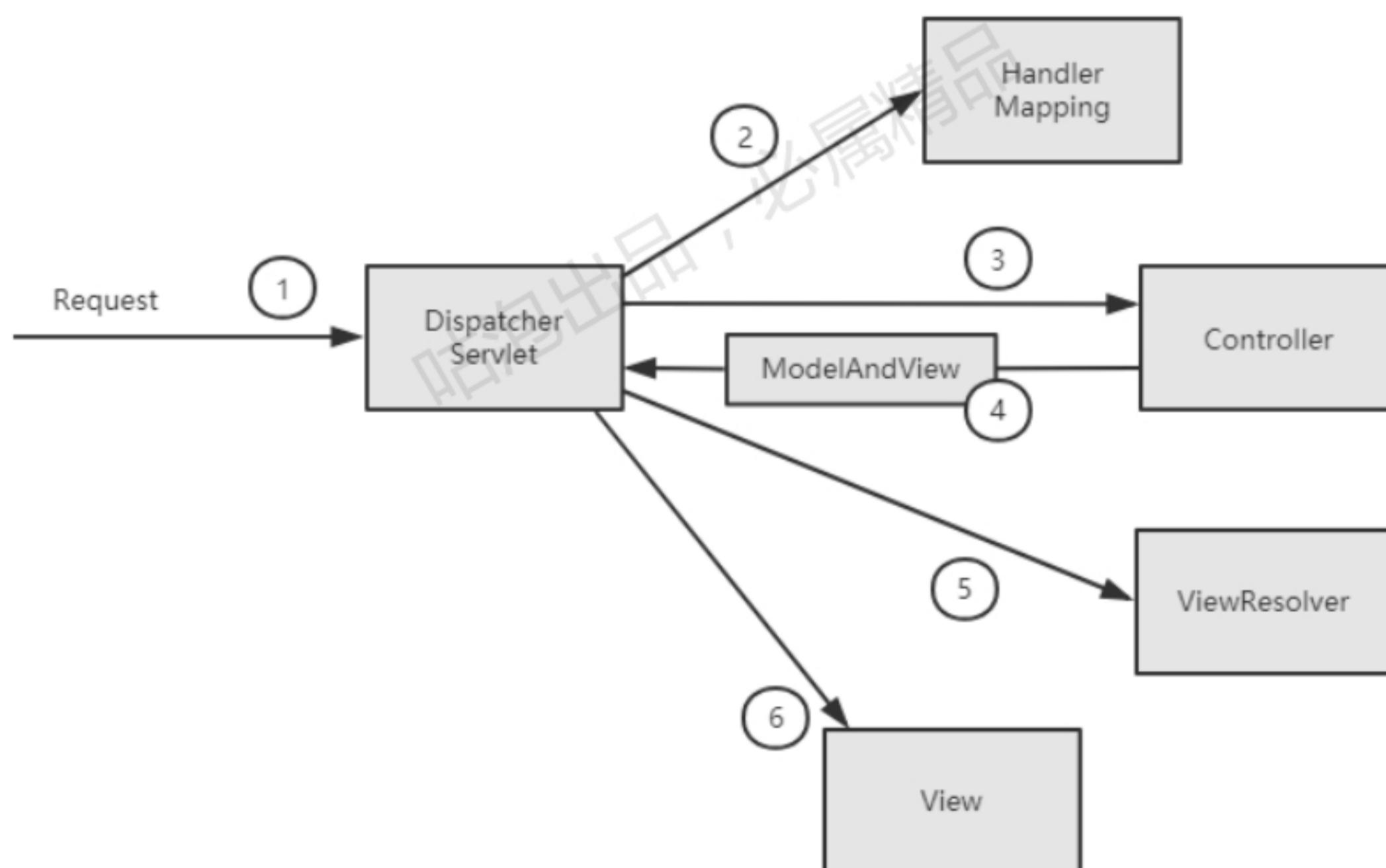
内容定位

- 1、Spring 使用不熟练者不适合学习本章内容。
- 2、先掌握执行流程，再理解设计思想，这个过程至少要花 1 个月。
- 3、Spring 源码非常经典，体系也非常庞大，看一遍是远远不够的。

Spring MVC 初体验

初探 Spring MVC 请求处理流程

Spring MVC 相对于前面的章节算是比较简单的，我们首先引用《Spring in Action》上的一张图来了解 Spring MVC 的核心组件和大致处理流程：



从上图中看到①、DispatcherServlet 是 SpringMVC 中的前端控制器(Front Controller),负责接收 Request 并将 Request 转发给对应的处理组件。

②、HanlerMapping 是 SpringMVC 中完成 url 到 Controller 映射的组件。DispatcherServlet 接收 Request, 然后从 HandlerMapping 查找处理 Request 的 Controller。

③、Controller 处理 Request,并返回 ModelAndView 对象,Controller 是 SpringMVC 中负责处理 Request 的组件(类似于 Struts2 中的 Action),ModelAndView 是封装结果视图的组件。

④、⑤、⑥视图解析器解析 ModelAndView 对象并返回对应的视图给客户端。

在前面的章节中我们已经大致了解到，容器初始化时会建立所有 url 和 Controller 中的 Method 的对应关系，保存到 HandlerMapping 中，用户请求是根据 Request 请求的 url 快速定位到 Controller 中的某个方法。在 Spring 中先将 url 和 Controller 的对应关系,保存到 Map<url,Controller>中。Web 容器启动时会通知 Spring 初始化容器(加载 Bean 的定义信息和初始化所有单例 Bean),然后 SpringMVC 会遍历容器中的 Bean , 获取每一个 Controller 中的所有方法访问的 url 然后将 url 和 Controller 保存到一个 Map 中；这样就可以根据 Request 快速定位到 Controller ，因为最终处理 Request 的是 Controller 中的方法，Map 中只保留了 url 和 Controller 中的对应关系，所以要根据 Request 的 url 进一步确认 Controller 中的 Method ，这一步工作的原理就是拼接 Controller 的 url(Controller 上 @RequestMapping 的值)和方法的 url(Method 上 @RequestMapping 的值)，与 request 的 url 进行匹配，找到匹配的那个方法；确定处理请求的 Method 后，接下来的任务就是参数绑定，把 Request 中参数绑定到方法的形式参数上，这一步是整个请求处理过程中最复杂的一个步骤。

Spring MVC 九大组件

HandlerMappings

HandlerMapping 是用来查找 Handler 的，也就是处理器，具体的表现形式可以是类也可以是方法。比如，标注了@RequestMapping 的每个 method 都可以看成是一个 Handler，由 Handler 来负责实际的请求处理。HandlerMapping 在请求到达之后，它的作用便是找到请求相应的处理器 Handler 和 Interceptors。

HandlerAdapters

从名字上看，这是一个适配器。因为 Spring MVC 中 Handler 可以是任意形式的，只要能够处理请求便行，但是把请求交给 Servlet 的时候，由于 Servlet 的方法结构都是如 doService(HttpServletRequest req, HttpServletResponse resp) 这样的形式，让固定的 Servlet 处理方法调用 Handler 来进行处理，这一步工作便是 HandlerAdapter 要做的事。

HandlerExceptionResolvers

从这个组件的名字上看，这个就是用来处理 Handler 过程中产生的异常情况的组件。具体来说，此组件的作用是根据异常设置 ModelAndView，之后再交给 render 方法进行渲染，而 render 便将 ModelAndView 渲染成页面。不过有一点，HandlerExceptionResolver 只是用于解析对请求做处理阶段产生的异常，而渲染阶段的异常则不归他管了，这也是 Spring MVC 组件设计的一大原则分工明确互不干涉。

ViewResolvers

视图解析器，相信大家对这个应该都很熟悉了。因为通常在 SpringMVC 的配置文件中，都会配上一个该接口的实现类来进行视图的解析。这个组件的主要作用，便是将 String

类型的视图名和 Locale 解析为 View 类型的视图。这个接口只有一个 resolveViewName() 方法。从方法的定义就可以看出，Controller 层返回的 String 类型的视图名 viewName，最终会在这里被解析成为 View. View 是用来渲染页面的，也就是说，它会将程序返回的参数和数据填入模板中，最终生成 html 文件。ViewResolver 在这个过程中，主要做两件大事，即，ViewResolver 会找到渲染所用的模板（使用什么模板来渲染？）和所用的技术（其实也就是视图的类型，如 JSP 啊还是其他什么 Blabla 的）填入参数。默认情况下，Spring MVC 会为我们自动配置一个 InternalResourceViewResolver，这个是针对 JSP 类型视图的。

RequestToViewNameTranslator

这个组件的作用，在于从 Request 中获取 viewName. 因为 ViewResolver 是根据 ViewName 查找 View, 但有的 Handler 处理完成之后，没有设置 View 也没有设置 ViewName，便要通过这个组件来从 Request 中查找 viewName。

LocaleResolver

在上面我们有看到 ViewResolver 的 resolveViewName()方法，需要两个参数。那么第二个参数 Locale 是从哪来的呢，这就是 LocaleResolver 要做的事了。 LocaleResolver 用于从 request 中解析出 Locale, 在中国大陆地区，Locale 当然就会是 zh-CN 之类，用来表示一个区域。这个类也是 i18n 的基础。

ThemeResolver

从名字便可看出，这个类是用来解析主题的。主题，就是样式，图片以及它们所形成的显示效果的集合。Spring MVC 中一套主题对应一个 properties 文件，里面存放着跟当前主题相关的所有资源，如图片，css 样式等。创建主题非常简单，只需准备好资源，然后新建一个 "主题名.properties" 并将资源设置进去，放在 classpath 下，便可以在页面

中使用了。 Spring MVC 中跟主题有关的类有 `ThemeResolver`, `ThemeSource` 和 `Theme`。`ThemeResolver` 负责从 `request` 中解析出主题名，`ThemeSource` 则根据主题名找到具体的主题，其抽象也就是 `Theme`, 通过 `Theme` 来获取主题和具体的资源。

MultipartResolver

其实这是一个大家很熟悉的组件，`MultipartResolver` 用于处理上传请求，通过将普通的 `Request` 包装成 `MultipartHttpServletRequest` 来实现。`MultipartHttpServletRequest` 可以通过 `getFile()` 直接获得文件，如果是多个文件上传，还可以通过调用 `getFileMap` 得到 `Map<FileName, File>` 这样的结构。`MultipartResolver` 的作用就是用来封装普通的 `request`，使其拥有处理文件上传的功能。

FlashMapManager

说到 `FlashMapManager`，就得先提一下 `FlashMap`。

`FlashMap` 用于重定向 `Redirect` 时的参数数据传递，比如，在处理用户订单提交时，为了避免重复提交，可以处理完 `post` 请求后 `redirect` 到一个 `get` 请求，这个 `get` 请求可以用来显示订单详情之类的信息。这样做虽然可以规避用户刷新重新提交表单的问题，但是在哪个页面上要显示订单的信息，那这些数据从哪里去获取呢，因为 `redirect` 重定向是没有传递参数这一功能的，如果不把参数写进 `url`(其实也不推荐这么做，`url` 有长度限制不说，把参数都直接暴露，感觉也不安全)，那么就可以通过 `flashMap` 来传递。只需要在 `redirect` 之前，将要传递的数据写入 `request`（可以通过 `ServletRequestAttributes.getRequest()` 获得）的属性 `OUTPUT_FLASH_MAP_ATTRIBUTE` 中，这样在 `redirect` 之后的 `handler` 中 `spring` 就会自动将其设置到 `Model` 中，在显示订单信息的页面上，就可以直接从 `Model` 中取得数据了。而 `FlashMapManager` 就是用来管理 `FlashMap` 的。

Spring MVC 源码分析

根据上面分析的 Spring MVC 工作机制，从三个部分来分析 Spring MVC 的源代码。

其一，ApplicationContext 初始化时用 Map 保存所有 url 和 Controller 类的对应关系；

其二，根据请求 url 找到对应的 Controller，并从 Controller 中找到处理请求的方法；

其三，Request 参数绑定到方法的形参，执行方法处理请求，并返回结果视图。

初始化阶段

我们首先找到 DispatcherServlet 这个类，必然是寻找 init() 方法。然后，我们发现其 init 方法其实在父类 HttpServletBean 中，其源码如下：

```
public final void init() throws ServletException {
    if (logger.isDebugEnabled()) {
        logger.debug("Initializing servlet '" + getServletName() + "'");
    }

    // Set bean properties from init parameters.
    PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.requiredProperties);
    if (!pvs.isEmpty()) {
        try {
            //定位资源
            BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
            //加载配置信息
            ResourceLoader resourceLoader = new ServletContextResourceLoader(getApplicationContext());
            bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, getEnvironment()));
            initBeanWrapper(bw);
            bw.setPropertyValues(pvs, true);
        }
        catch (BeansException ex) {
            if (logger.isErrorEnabled()) {
                logger.error("Failed to set bean properties on servlet '" + getServletName() + "'", ex);
            }
            throw ex;
        }
    }

    // Let subclasses do whatever initialization they like.
    initServletBean();

    if (logger.isDebugEnabled()) {
        logger.debug("Servlet '" + getServletName() + "' configured successfully");
    }
}
```

我们看到在这段代码中，又调用了一个重要的 initServletBean() 方法。进入 initServletBean() 方法看到以下源码：

```
protected final void initServletBean() throws ServletException {
```

```

getServletContext().log("Initializing Spring FrameworkServlet '" + getServletName() + "'");
if (this.logger.isInfoEnabled()) {
    this.logger.info("FrameworkServlet '" + getServletName() + "': initialization started");
}
long startTime = System.currentTimeMillis();

try {

    this.webApplicationContext = initWebApplicationContext();
    initFrameworkServlet();
}

catch (ServletException ex) {
    this.logger.error("Context initialization failed", ex);
    throw ex;
}
catch (RuntimeException ex) {
    this.logger.error("Context initialization failed", ex);
    throw ex;
}

if (this.logger.isInfoEnabled()) {
    long elapsedTime = System.currentTimeMillis() - startTime;
    this.logger.info("FrameworkServlet '" + getServletName() + "': initialization completed in " +
        elapsedTime + " ms");
}
}
}

```

这段代码中最主要的逻辑就是初始化 IOC 容器，最终会调用 `refresh()` 方法，前面的章节中对 IOC 容器的初始化细节我们已经详细掌握，在此我们不再赘述。我们看到上面的代码中，IOC 容器初始化之后，最后有调用了 `onRefresh()` 方法。这个方法最终是在 `DispatcherServlet` 中实现，来看源码：

```

@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.
 */
//初始化策略
protected void initStrategies(ApplicationContext context) {
    //多文件上传的组件
    initMultipartResolver(context);
    //初始化本地语言环境
    initLocaleResolver(context);
    //初始化模板处理器
    initThemeResolver(context);
    //handlerMapping
    initHandlerMappings(context);
    //初始化参数适配器
    initHandlerAdapters(context);
    //初始化异常拦截器
    initHandlerExceptionResolvers(context);
    //初始化视图预处理器
    initRequestToViewNameTranslator(context);
    //初始化视图转换器
}

```

```

    initViewResolvers(context);
    //FlashMap 管理器
    initFlashMapManager(context);
}

```

到这一步就完成了 Spring MVC 的九大组件的初始化。接下来 我们来看 url 和 Controller 的 关 系 是 如 何 建 立 的 呢 ？ HandlerMapping 的 子 类 AbstractDetectingUrlHandlerMapping 实现了 initApplicationContext()方法，所以 我们直接看子类中的初始化容器方法。

```

@Override
public void initApplicationContext() throws ApplicationContextException {
    super.initApplicationContext();
    detectHandlers();
}

/**
 * 建立当前 ApplicationContext 中的所有 Controller 和 url 的对应关系
 */
protected void detectHandlers() throws BeansException {
    ApplicationContext applicationContext = obtainApplicationContext();
    if (logger.isDebugEnabled()) {
        logger.debug("Looking for URL mappings in application context: " + applicationContext);
    }
    // 获取 ApplicationContext 容器中所有 bean 的 Name
    String[] beanNames = (this.detectHandlersInAncestorContexts ?
        BeanFactoryUtils.beanNamesForTypeIncludingAncestors(applicationContext, Object.class) :
        applicationContext.getBeanNamesForType(Object.class));

    // 遍历 beanNames，并找到这些 bean 对应的 url
    for (String beanName : beanNames) {
        // 找 bean 上的所有 url(Controller 上的 url+方法上的 url)，该方法由对应的子类实现
        String[] urls = determineUrlsForHandler(beanName);
        if (!ObjectUtils.isEmpty(urls)) {
            // 保存 urls 和 beanName 的对应关系，put it to Map<urls,beanName>,
        // 该方法在父类 AbstractUrlHandlerMapping 中实现
            registerHandler(urls, beanName);
        }
        else {
            if (logger.isDebugEnabled()) {
                logger.debug("Rejected bean name '" + beanName + "' no URL paths identified");
            }
        }
    }
}

/** 获取 Controller 中所有方法的 url，由子类实现，典型的模板模式 */
protected abstract String[] determineUrlsForHandler(String beanName);

```

determineUrlsForHandler(String beanName)方法的作用是获取每个 Controller 中的 url，不同的子类有不同的实现，这是一个典型的模板设计模式。因为开发中我们用的最多的就是用注解来配置 Controller 中的 url，BeanNameUrlHandlerMapping 是

AbstractDetectingUrlHandlerMapping 的子类,处理注解形式的 url 映射.所以我们这里以 BeanNameUrlHandlerMapping 来进行分析。我们看 BeanNameUrlHandlerMapping 是如何查 beanName 上所有映射的 url。

```
/**
 * 获取 Controller 中所有的 url
 */
@Override
protected String[] determineUrlsForHandler(String beanName) {
    List<String> urls = new ArrayList<>();
    if (beanName.startsWith("/")) {
        urls.add(beanName);
    }
    String[] aliases = obtainApplicationContext().getAliases(beanName);
    for (String alias : aliases) {
        if (alias.startsWith("/")) {
            urls.add(alias);
        }
    }
    return StringUtils.toStringArray(urls);
}
```

到这里 HandlerMapping 组件就已经建立所有 url 和 Controller 的对应关系。

运行调用阶段

这一步是由请求触发的 , 所以入口为 DispatcherServlet 的核心方法为 doService() , doService()中的核心逻辑由 doDispatch()实现 , 源代码如下 :

```
/** 中央控制器,控制请求的转发 */
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 {
            // 1. 检查是否是文件上传的请求
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

            // 2. 取得处理当前请求的 Controller, 这里也称为 handler, 处理器,
            // 第一个步骤的意义就在这里体现了. 这里并不是直接返回 Controller,
            // 而是返回的 HandlerExecutionChain 请求处理器链对象,
            // 该对象封装了 handler 和 interceptors.
            mappedHandler = getHandler(processedRequest);
            // 如果 handler 为空, 则返回 404
            if (mappedHandler == null) {
                noHandlerFound(processedRequest, response);
                return;
            }
        }
```

```
        }

    //3. 获取处理 request 的处理器适配器 handler adapter
    HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

    // 处理 last-modified 请求头
    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;
    }

    // 4. 实际的处理器处理请求,返回结果视图对象
    mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

    if (asyncManager.isConcurrentHandlingStarted()) {
        return;
    }

    // 结果视图对象的处理
    applyDefaultViewName(processedRequest, mv);
    mappedHandler.applyPostHandle(processedRequest, response, mv);
}

catch (Exception ex) {
    dispatchException = ex;
}
catch (Throwable err) {
    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()) {
        if (mappedHandler != null) {
            // 请求成功响应之后的方法
            mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
        }
    }
    else {
        if (multipartRequestParsed) {
            cleanupMultipart(processedRequest);
        }
    }
}
```

`getHandler(processedRequest)` 方法实际上就是从 `HandlerMapping` 中找到 url 和 Controller 的对应关系。也就是 `Map<url,Controller>`。我们知道，最终处理 Request 的是 Controller 中的方法，我们现在只是知道了 Controller，我们如何确认 Controller 中处理 Request 的方法呢？继续往下看。

从 `Map<urls,beanName>` 中取得 Controller 后，经过拦截器的预处理方法，再通过反射获取该方法上的注解和参数，解析方法和参数上的注解，然后反射调用方法获取 `ModelAndView` 结果视图。最后，调用的就是 `RequestMappingHandlerAdapter` 的 `handle()` 中的核心逻辑由 `handleInternal(request, response, handler)` 实现。

```

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

    ModelAndView mav;
    checkRequest(request);

    if (this.synchronizeOnSession) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            Object mutex = WebUtils.getSessionMutex(session);
            synchronized (mutex) {
                mav = invokeHandlerMethod(request, response, handlerMethod);
            }
        }
        else {
            mav = invokeHandlerMethod(request, response, handlerMethod);
        }
    }
    else {
        mav = invokeHandlerMethod(request, response, handlerMethod);
    }

    if (!response.containsHeader(HEADER_CACHE_CONTROL)) {
        if (getSessionAttributesHandler(handlerMethod).hasSessionAttributes()) {
            applyCacheSeconds(response, this.cacheSecondsForSessionAttributeHandlers);
        }
        else {
            prepareResponse(response);
        }
    }

    return mav;
}

```

整个处理过程中最核心的逻辑其实就是拼接 Controller 的 url 和方法的 url，与 Request 的 url 进行匹配，找到匹配的方法。

```
/** 根据 url 获取处理请求的方法 **/
```

```

@Override
protected HandlerMethod getHandlerInternal(HttpServletRequest request) throws Exception {
    // 如果请求 url 为, http://localhost:8080/web/hello.json, 则 lookupPath=web/hello.json
    String lookupPath = getUrlPathHelper().getLookupPathForRequest(request);
    if (logger.isDebugEnabled()) {
        logger.debug("Looking up handler method for path " + lookupPath);
    }
    this.mappingRegistry.acquireReadLock();
    try {
        // 遍历 Controller 上的所有方法, 获取 url 匹配的方法
        HandlerMethod handlerMethod = lookupHandlerMethod(lookupPath, request);
        if (logger.isDebugEnabled()) {
            if (handlerMethod != null) {
                logger.debug("Returning handler method [" + handlerMethod + "]");
            } else {
                logger.debug("Did not find handler method for [" + lookupPath + "]");
            }
        }
        return (handlerMethod != null ? handlerMethod.createWithResolvedBean() : null);
    }
    finally {
        this.mappingRegistry.releaseReadLock();
    }
}

```

通过上面的代码分析，已经可以找到处理 Request 的 Controller 中的方法了，现在看如何解析该方法上的参数，并反射调用该方法。

```

/** 获取处理请求的方法, 执行并返回结果视图 */
@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) {
            invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
        }
        if (this.returnValueHandlers != null) {
            invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
        }
        invocableMethod.setDataBinderFactory(binderFactory);
        invocableMethod.setParameterNameDiscoverer(this.parameterNameDiscoverer);

        ModelAndView mavContainer = new ModelAndView();
        mavContainer.addAllAttributes(RequestContextUtils.getInputFlashMap(request));
        modelFactory.initModel(webRequest, mavContainer, invocableMethod);
        mavContainer.setIgnoreDefaultModelOnRedirect(this.ignoreDefaultModelOnRedirect);

        AsyncWebRequest asyncWebRequest = WebAsyncUtils.createAsyncWebRequest(request, response);
        asyncWebRequest.setTimeout(this.asyncRequestTimeout);

        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
        asyncManager.setTaskExecutor(this.taskExecutor);
        asyncManager.setAsyncWebRequest(asyncWebRequest);
        asyncManager.registerCallableInterceptors(this.callableInterceptors);
    }
}

```

```

        asyncManager.registerDeferredResultInterceptors(this.deferredResultInterceptors);

        if (asyncManager.hasConcurrentResult()) {
            Object result = asyncManager.getConcurrentResult();
            mavContainer = (ModelAndViewContainer) asyncManager.getConcurrentResultContext()[0];
            asyncManager.clearConcurrentResult();
            if (logger.isDebugEnabled()) {
                logger.debug("Found concurrent result value [" + result + "]");
            }
            invocableMethod = invocableMethod.wrapConcurrentResult(result);
        }

        invocableMethod.invokeAndHandle(webRequest, mavContainer);
        if (asyncManager.isConcurrentHandlingStarted()) {
            return null;
        }

        return getModelAndView(mavContainer, modelFactory, webRequest);
    }
    finally {
        webRequest.requestCompleted();
    }
}

```

invocableMethod.invokeAndHandle()最终要实现的目的就是：完成 Request 中的参数和方法参数上数据的绑定。Spring MVC 中提供两种 Request 参数到方法中参数的绑定方式：

- 1、通过注解进行绑定，@RequestParam。
- 2、通过参数名称进行绑定。

使用注解进行绑定，我们只要在方法参数前面声明@RequestParam("name")，就可以将 request 中参数 name 的值绑定到方法的该参数上。使用参数名称进行绑定的前提是必须要获取方法中参数的名称，Java 反射只提供了获取方法的参数的类型，并没有提供获取参数名称的方法。SpringMVC 解决这个问题的方法是用 asm 框架读取字节码文件，来获取方法的参数名称。asm 框架是一个字节码操作框架，关于 asm 更多介绍可以参考其官网。个人建议，使用注解来完成参数绑定，这样就可以省去 asm 框架的读取字节码的操作。

```

@Nullable
public Object invokeForRequest(NativeWebRequest request, @Nullable ModelAndView mavContainer,
    Object... providedArgs) throws Exception {

    Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);
    if (logger.isTraceEnabled()) {

```

```

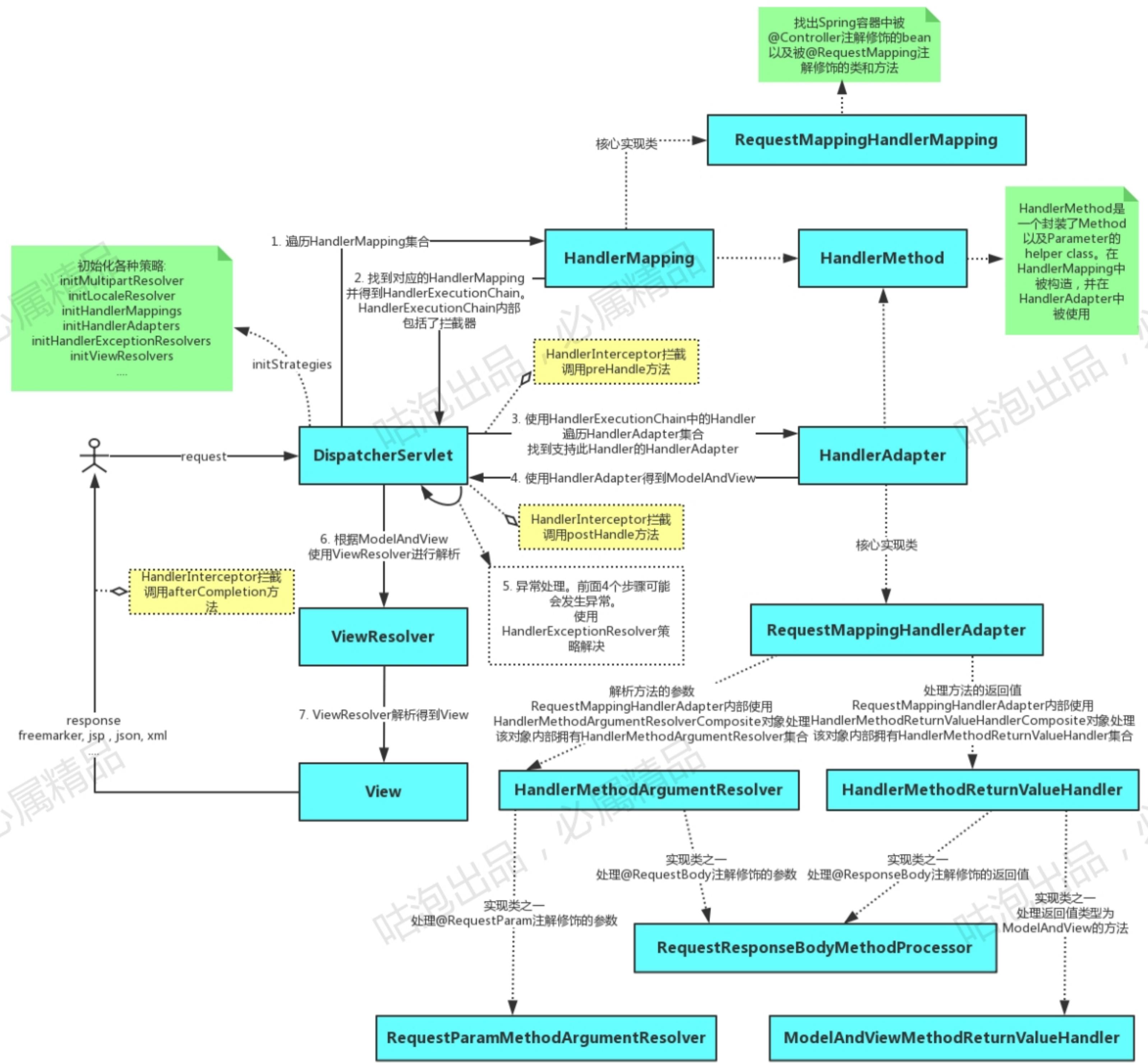
        logger.trace("Invoking '" + ClassUtils.getQualifiedMethodName(getMethod(), getBeanType()) +
                     " with arguments " + Arrays.toString(args));
    }
    Object returnValue = doInvoke(args);
    if (logger.isTraceEnabled()) {
        logger.trace("Method [" + ClassUtils.getQualifiedMethodName(getMethod(), getBeanType()) +
                     "] returned [" + returnValue + "]");
    }
    return returnValue;
}

private Object[] getMethodArgumentValues(NativeWebRequest request, @Nullable ModelAndView mavContainer,
                                         Object... providedArgs) throws Exception {
    MethodParameter[] parameters = getMethodParameters();
    Object[] args = new Object[parameters.length];
    for (int i = 0; i < parameters.length; i++) {
        MethodParameter parameter = parameters[i];
        parameter.initParameterNameDiscovery(this.parameterNameDiscoverer);
        args[i] = resolveProvidedArgument(parameter, providedArgs);
        if (args[i] != null) {
            continue;
        }
        if (this.argumentResolvers.supportsParameter(parameter)) {
            try {
                args[i] = this.argumentResolvers.resolveArgument(
                    parameter, mavContainer, request, this.dataBinderFactory);
                continue;
            }
            catch (Exception ex) {
                if (logger.isDebugEnabled()) {
                    logger.debug(getArgumentResolutionErrorMessage("Failed to resolve", i), ex);
                }
                throw ex;
            }
        }
        if (args[i] == null) {
            throw new IllegalStateException("Could not resolve method parameter at index " +
                parameter.getParameterIndex() + " in " + parameter.getExecutable().toGenericString() +
                ": " + getArgumentResolutionErrorMessage("No suitable resolver for", i));
        }
    }
    return args;
}

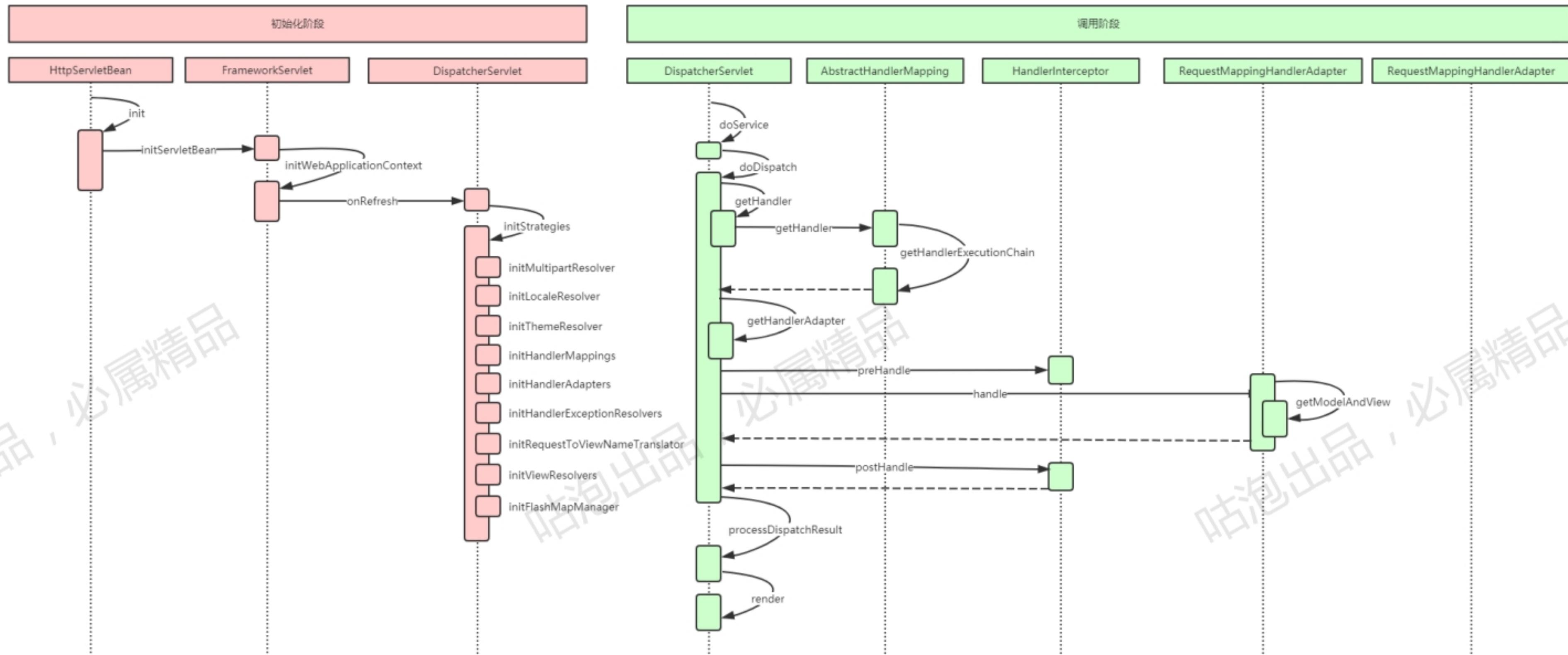
```

关于 asm 框架获取方法参数的部分,这里就不再进行分析了。感兴趣的小伙伴可以继续深入了解这个处理过程。

到这里,方法的参数值列表也获取到了,就可以直接进行方法的调用了。整个请求过程中最复杂的一歩就是在这里了。到这里整个请求处理过程的关键步骤都已了解。理解了 Spring MVC 中的请求处理流程,整个代码还是比较清晰的。最后我们再来梳理一下 Spring MVC 核心组件的关联关系(如下图) :



最后来一张时序图：



Spring MVC 使用优化建议

上面我们已经对 SpringMVC 的工作原理和源码进行了分析，在这个过程发现了几个优化点：

1、Controller 如果能保持单例，尽量使用单例

这样可以减少创建对象和回收对象的开销。也就是说，如果 Controller 的类变量和实例变量可以以方法形参声明的尽量以方法的形参声明，不要以类变量和实例变量声明，这样可以避免线程安全问题。

2、处理 Request 的方法中的形参务必加上@RequestParam 注解

这样可以避免 SpringMVC 使用 asm 框架读取 class 文件获取方法参数名的过程。即使 SpringMVC 对读取出的方法参数名进行了缓存，如果不要读取 class 文件当然是更好。

3、缓存 URL

阅读源码的过程中，我们发现 Spring MVC 并没有对处理 url 的方法进行缓存，也就是说每次都要根据请求 url 去匹配 Controller 中的方法 url，如果把 url 和 Method 的关系

缓存起来，会不会带来性能上的提升呢？有点恶心的是，负责解析 url 和 Method 对应关系的 `ServletHandlerMethodResolver` 是一个 `private` 的内部类，不能直接继承该类增强代码，必须要该代码后重新编译。当然，如果缓存起来，必须要考虑缓存的线程安全问题。