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。

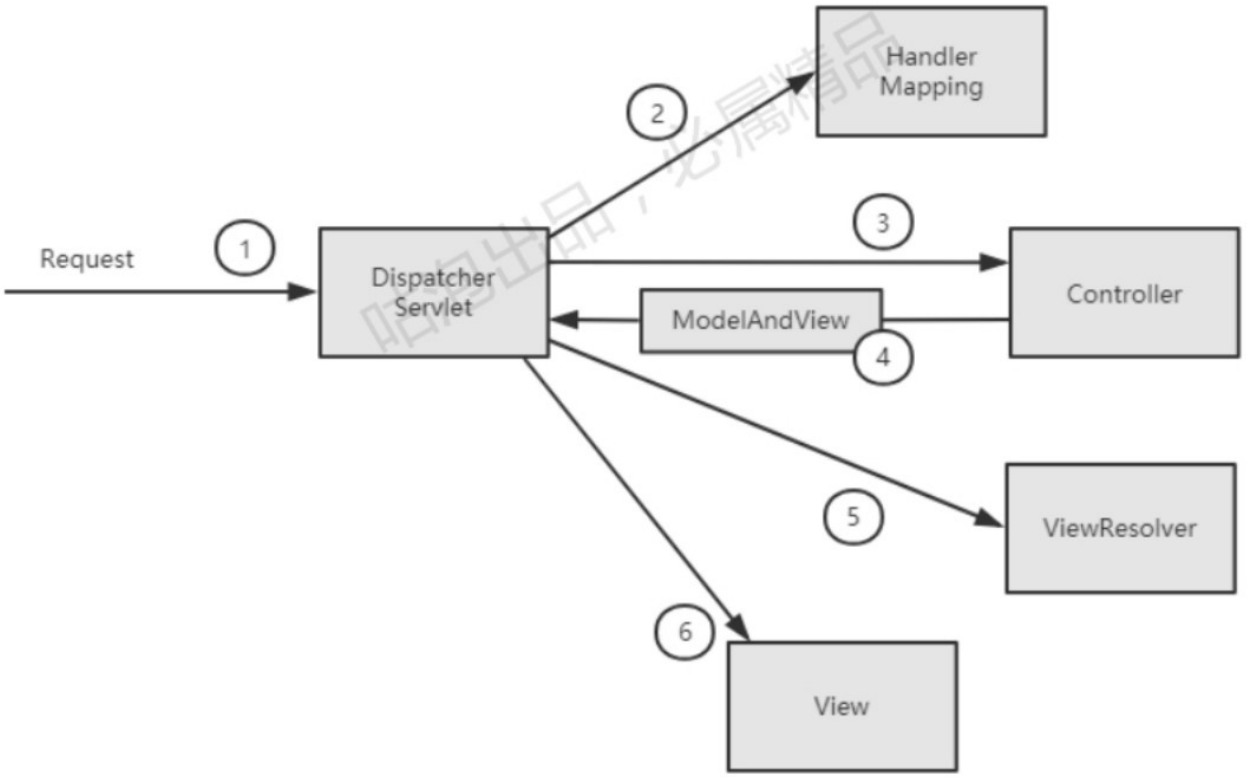
内容定位

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。

1. Spring MVC初体验

1.1. 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、HandlerMapping 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、④、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。

2. Spring MVC 九大组件

2.1. 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。

2.2. 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。

2.3. 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。

2.4. 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。

2.5. 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。

2.6. 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。

2.7. 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。

2.8. 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。

2.9. 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，但是在这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，如果不想把参数写进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。

3. 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。

3.1. 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：

```
    @Override
    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.r
        if (!pvs.isEmpty()) {
            try {
                //定位资源
                BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
                //加载配置信息
                ResourceLoader resourceLoader = new ServletContextResourceLoader(getSer
                bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoad
                initBeanWrapper(bw);
                bw.setPropertyValues(pvs, true);
            }
            catch (BeansException ex) {
                if (logger.isErrorEnabled()) {
                    logger.error("Failed to set bean properties on servlet '" + getServ
                }
                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：

```
    @Override
    protected final void initServletBean() throws ServletException {
        getServletContext().log("Initializing Spring FrameworkServlet '" + getServletNa
        if (this.logger.isInfoEnabled()) {
            this.logger.info("FrameworkServlet '" + getServletName() + "': initializati
        }
        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() + "': initializati
                    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DisptcherServlet中实现，来看源码：

```
    /**
     * This implementation calls {@link #initStrategies}.
     */
    @Override
    protected void onRefresh(ApplicationContext context) {
        initStrategies(context);
    }
```

```
8
9       /**
10       * Initialize the strategy objects that this servlet uses.
11       * <p>May be overridden in subclasses in order to initialize further strategy objec
12       */
13      //初始化策略
14      protected void initStrategies(ApplicationContext context) {
15          //多文件上传的组件
16          initMultipartResolver(context);
17          //初始化本地语言环境
18          initLocaleResolver(context);
19          //初始化模板处理器
20          initThemeResolver(context);
21          //handlerMapping
22          initHandlerMappings(context);
23          //初始化参数适配器
24          initHandlerAdapters(context);
25          //初始化异常拦截器
26          initHandlerExceptionResolvers(context);
27          //初始化视图预处理器
28          initRequestToViewNameTranslator(context);
29          //初始化视图转换器
30          initViewResolvers(context);
31          //初始化FlashMap管理器
32          initFlashMapManager(context);
33      }
```

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。

```
1       /**
2        * Calls the {@link #detectHandlers()} method in addition to the
3        * superclass's initialization.
4        */
5       @Override
6       public void initApplicationContext() throws ApplicationContextException {
7           super.initApplicationContext();
8           detectHandlers();
9       }
10
```

```
    /**
     * Register all handlers found in the current ApplicationContext.
     * <p>The actual URL determination for a handler is up to the concrete
     * {@link #determineUrlsForHandler(String)} implementation. A bean for
     * which no such URLs could be determined is simply not considered a handler.
     * @throws org.springframework.beans.BeansException if the handler couldn't be regi
     * @see #determineUrlsForHandler(String)
     */
    /**
     * 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: " + applicat
        }
        // 获取ApplicationContext容器中所有bean的Name
        String[] beanNames = (this.detectHandlersInAncestorContexts ?
                BeanFactoryUtils.beanNamesForTypeIncludingAncestors(applicationContext,
                applicationContext.getBeanNamesForType(Object.class));

        // Take any bean name that we can determine URLs for.
        // 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)) {
                // URL paths found: Let's consider it a handler.
                // 保存urls和beanName的对应关系,put it to Map<urls,beanName>,该方法在父
                registerHandler(urls, beanName);
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Rejected bean name '" + beanName + "': no URL paths i
                }
            }
        }
    }


    /**
     * Determine the URLs for the given handler bean.
     * @param beanName the name of the candidate bean
```

```
     * @return the URLs determined for the bean, or an empty array if none
     */
    /** 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。

```
    @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。

3.2. 运行调用阶段

这一步步是由请求触发的，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)
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        boolean multipartRequestParsed = false;

        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
```

```
8
9           try {
10              ModelAndView mv = null;
11              Exception dispatchException = null;
12
13              try {
14                  // 1.检查是否是文件上传的请求
15                  processedRequest = checkMultipart(request);
16                  multipartRequestParsed = (processedRequest != request);
17
18                  // Determine handler for the current request.
19                  // 2.取得处理当前请求的controller,这里也称为hanlder,处理器,
20                  //      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,
21                  //     而是返回的HandlerExecutionChain请求处理器链对象,
22                  //     该对象封装了handler和interceptors.
23                  mappedHandler = getHandler(processedRequest);
24                  // 如果handler为空,则返回404
25                  if (mappedHandler == null) {
26                      noHandlerFound(processedRequest, response);
27                      return;
28                  }
29
30                  // Determine handler adapter for the current request.
31                  //3. 获取处理request的处理器适配器handler adapter
32                  HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
33
34                  // Process last-modified header, if supported by the handler.
35                  // 处理 last-modified 请求头
36                  String method = request.getMethod();
37                  boolean isGet = "GET".equals(method);
38                  if (isGet || "HEAD".equals(method)) {
39                      long lastModified = ha.getLastModified(request, mappedHandler.getHa
40                      if (logger.isDebugEnabled()) {
41                          logger.debug("Last-Modified value for [" + getRequestUri(reques
42                      }
43                      if (new ServletWebRequest(request, response).checkNotModified(lastM
44                          return;
45                      }
46                  }
47
48                  if (!mappedHandler.applyPreHandle(processedRequest, response)) {
49                      return;
50                  }
```

```
51
52                  // Actually invoke the handler.
53                  // 4.实际的处理器处理请求,返回结果视图对象
54                  mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
55
56                  if (asyncManager.isConcurrentHandlingStarted()) {
57                      return;
58                  }
59
60                  // 结果视图对象的处理
61                  applyDefaultViewName(processedRequest, mv);
62                  mappedHandler.applyPostHandle(processedRequest, response, mv);
63              }
64              catch (Exception ex) {
65                  dispatchException = ex;
66              }
67              catch (Throwable err) {
68                  // As of 4.3, we're processing Errors thrown from handler methods as we
69                  // making them available for @ExceptionHandler methods and other scenar
70                  dispatchException = new NestedServletException("Handler dispatch failed
71              }
72              processDispatchResult(processedRequest, response, mappedHandler, mv, dispat
73          }
74          catch (Exception ex) {
75              triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
76          }
77          catch (Throwable err) {
78              triggerAfterCompletion(processedRequest, response, mappedHandler,
79                      new NestedServletException("Handler processing failed", err));
80          }
81          finally {
82              if (asyncManager.isConcurrentHandlingStarted()) {
83                  // Instead of postHandle and afterCompletion
84                  if (mappedHandler != null) {
85                      // 请求成功响应之后的方法
86                      mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest,
87                  }
88              }
89              else {
90                  // Clean up any resources used by a multipart request.
91                  if (multipartRequestParsed) {
92                      cleanupMultipart(processedRequest);
93                  }
```

```
            }
        }
    }
```

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);

        // 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);
        }

        if (!response.containsHeader(HEADER_CACHE_CONTROL)) {
            if (getSessionAttributesHandler(handlerMethod).hasSessionAttributes()) {
```

```
                applyCacheSeconds(response, this.cacheSecondsForSessionAttributeHandler
            }
            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 Excep
        // 如果请求url为: http://localhost:8080/web/hello.json,
        // 则 lookupPath为: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() : nu
        }
        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 = createInvocableHandlerMetho
            if (this.argumentResolvers != null) {
                invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolver
            }
            if (this.returnValueHandlers != null) {
                invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHan
            }
            invocableMethod.setDataBinderFactory(binderFactory);
            invocableMethod.setParameterNameDiscoverer(this.parameterNameDiscoverer);

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

            AsyncWebRequest asyncWebRequest = WebAsyncUtils.createAsyncWebRequest(reque
            asyncWebRequest.setTimeout(this.asyncRequestTimeout);

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

            if (asyncManager.hasConcurrentResult()) {
                Object result = asyncManager.getConcurrentResult();
                mavContainer = (ModelAndViewContainer) asyncManager.getConcurrentResult
                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。

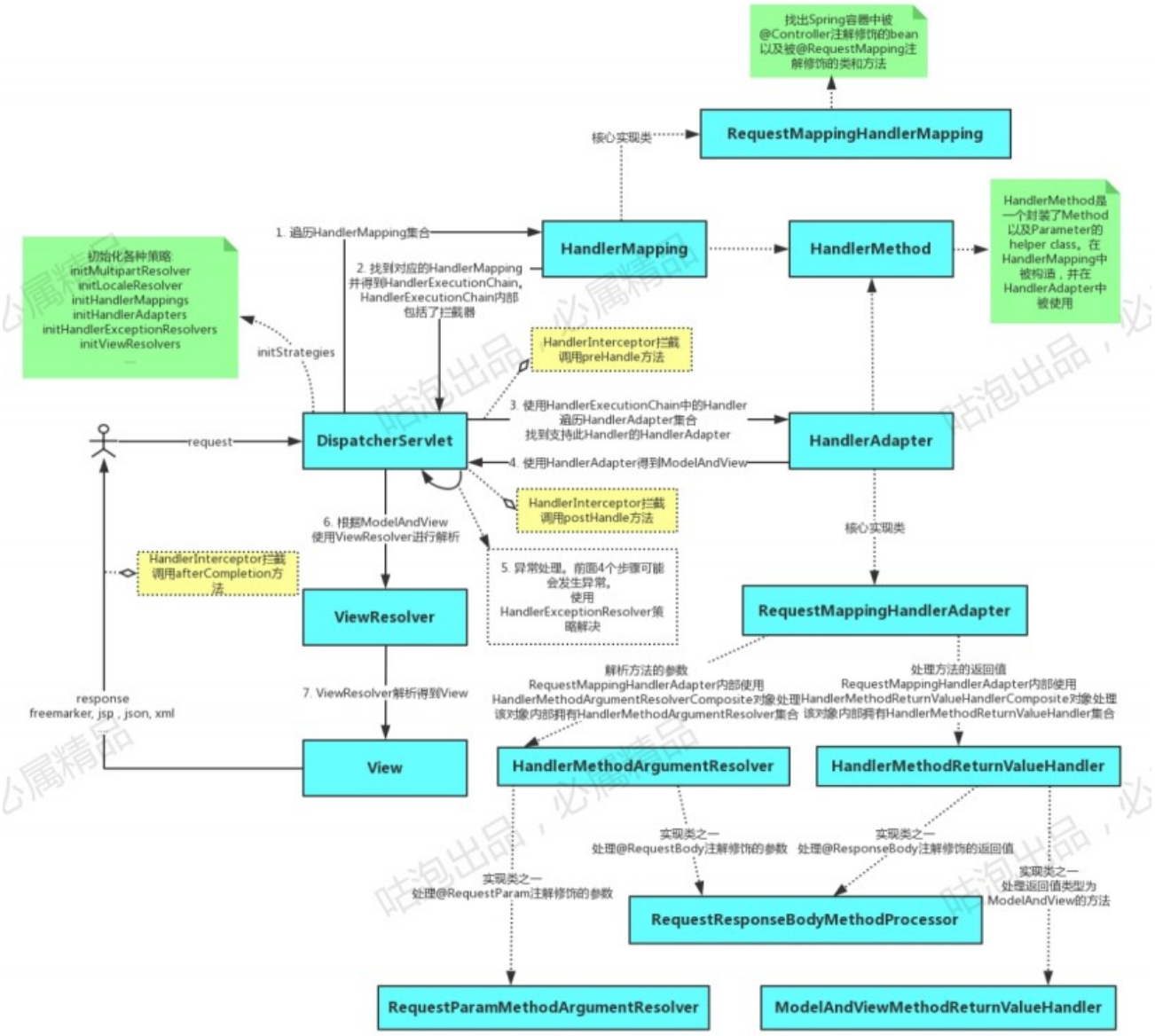
```
    @Nullable
    public Object invokeForRequest(NativeWebRequest request, @Nullable ModelAndViewCont
            Object... providedArgs) throws Exception {

        Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);
        if (logger.isTraceEnabled()) {
            logger.trace("Invoking '" + ClassUtils.getQualifiedMethodName(getMethod(),
                    "' with arguments " + Arrays.toString(args));
        }
        Object returnValue = doInvoke(args);
        if (logger.isTraceEnabled()) {
            logger.trace("Method [" + ClassUtils.getQualifiedMethodName(getMethod(), ge
                    "] returned [" + returnValue + "]");
```

```
        }
        return returnValue;
    }

    private Object[] getMethodArgumentValues(NativeWebRequest request, @Nullable ModelA
            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 resol
                    }
                    throw ex;
                }
            }
            if (args[i] == null) {
                throw new IllegalStateException("Could not resolve method parameter at
                        parameter.getParameterIndex() + " in " + parameter.getExecutabl
                        ": " + getArgumentResolutionErrorMessage("No suitable resolver
            }
        }
        return args;
    }
```

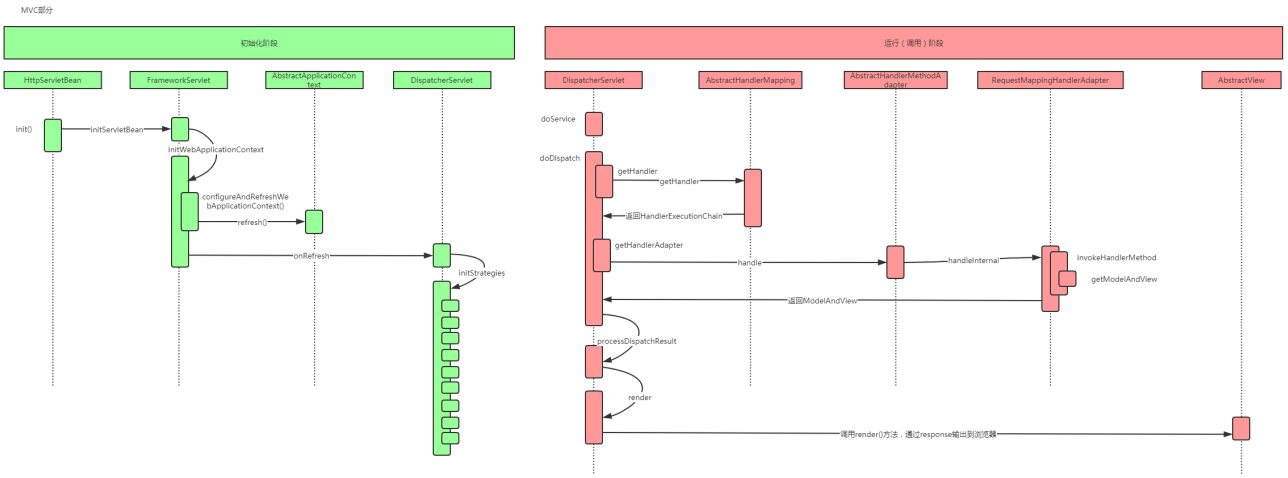
关于asm框架获取方法参数的部分，这里就不再进行分析了。感兴趣的小伙伴可以继续深入了解这个处理过程。

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步就是在这里了。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：



4. 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

4.1. 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。

4.2. 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。即便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。

4.3. 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。