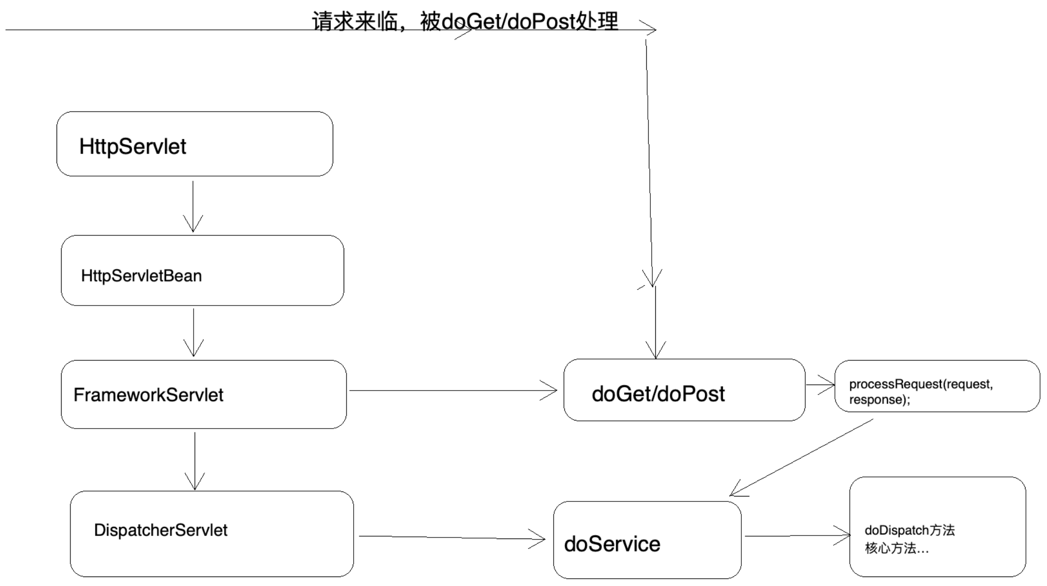
DispatcherServlet继承结构

如下图，HttpServlet时最上层的抽像类，往下依次继承上一层。



doDispatch（）的核心步骤

> DispatcherServlet.doDispatch（）地位相当于spring源码的refresh（）

整体流程：

1）调用getHandler()获取到能够处理当前请求的执行链 HandlerExecutionChain（Handler+拦截器）但是如何去getHandler的?--得到Handler(细节后面分析)

2）调用getHandlerAdapter()；获取能够执行1）中Handler的适配器
但是如何去getHandlerAdapter的？--得到执行Handler的适配器(细节后面分析)

3）适配器调用Handler执行ha.handle（总会返回一个ModelAndView对象）--执行Handler(细节后面分析)

4）调用processDispatchResult()方法完成视图渲染跳转--返回前端

```java
@SuppressWarnings("serial")
public class DispatcherServlet extends FrameworkServlet {
        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);

                // Determine handler for the current request.
                /*
                    2 取得处理当前请求的Controller，这里也称为Handler，即处理器
                      这里并不是直接返回 Controller，而是返回 HandlerExecutionChain 请求处理链对象
                      该对象封装了Handler和Inteceptor
                 */
                mappedHandler = getHandler(processedRequest);
                if (mappedHandler == null) {
                    // 如果 handler 为空，则返回404
                    noHandlerFound(processedRequest, response);
                    return;
                }

                // Determine handler adapter for the current request.
                // 3 获取处理请求的处理器适配器 HandlerAdapter
                HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

                // Process last-modified header, if supported by the handler.
                // 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 (new ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {
                        return;
                    }
                }

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

                // Actually invoke the handler.
                // 4 实际处理器处理请求，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) {
            //最终会调用HandlerInterceptor的afterCompletion 方法
            triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
        }
        catch (Throwable err) {
            //最终会调用HandlerInterceptor的afterCompletion 方法
            triggerAfterCompletion(processedRequest, response, mappedHandler,
                    new NestedServletException("Handler processing failed", err));
        }
    }
}
```

getHandler（）讲解：

主要步骤：

通过HttpServletRequest得到请求中handler对应的url，遍历handlerMappings，获取url对应的handler。

> 该方法主要是通过request的url得到相应handler
>
> 触发时机：容器启动的时候，扫描注解就可以建立url和handler的关系，将handler放入handlerMappings

```
   /**
     * 取得处理当前请求的Controller，这里也称为Handler，即处理器
     * 这里并不是直接返回 Controller，而是返回 HandlerExecutionChain 请求处理链对象
     * 该对象封装了Handler和Inteceptor
     */
   @Nullable
    protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
        if (this.handlerMappings != null) {
            //遍历handlerMappings
            //有两种handlerMappings实现类
            //1.BeanNameUrlHandlerMapping：封装xml配置的handler
            //2.RequestMappingHandlerMapping：封装注解配置的handler（一般都用这个）
            for (HandlerMapping mapping : this.handlerMappings) {
                HandlerExecutionChain handler = mapping.getHandler(request);
                if (handler != null) {
                    return handler;
```

```java
                }
            }
        }
        return null;
    }

public class HandlerExecutionChain {

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

    private final Object handler;

    @Nullable
    private HandlerInterceptor[] interceptors;

    @Nullable
    private List<HandlerInterceptor> interceptorList;

    private int interceptorIndex = -1;
}
```

getHandlerAdapter（）讲解: 为什么要使用不同适配器? ? ?

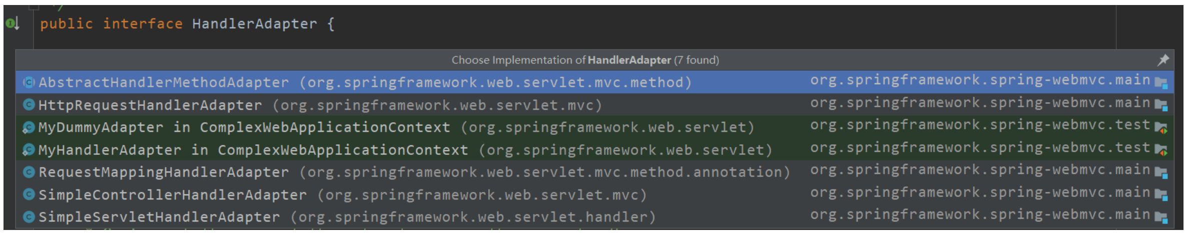
主要步骤:

不同适配器的判断方法不同，只有返回true才返回该类型适配器。

> 该方法也是容器初始化的时候调用的

```java
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException 
{
        if (this.handlerAdapters != null) {
            for (HandlerAdapter adapter : this.handlerAdapters) {
                //根据getHandler（）类型，遍历判断该类型是否实现某个接口判断适配器类型
                if (adapter.supports(handler)) {
                    return adapter;
                }
            }
        }
        throw new ServletException("No adapter for handler this handler");
    }
```

HandlerAdapter实现类:



```
public interface HandlerAdapter {
```

Choose Implementation of **HandlerAdapter** (7 found)

```
AbstractHandlerMethodAdapter (org.springframework.web.servlet.mvc.method)                org.springframework.spring-webmvc.main
HttpRequestHandlerAdapter (org.springframework.web.servlet.mvc)                          org.springframework.spring-webmvc.main
MyDummyAdapter in ComplexWebApplicationContext (org.springframework.web.servlet)         org.springframework.spring-webmvc.test
MyHandlerAdapter in ComplexWebApplicationContext (org.springframework.web.servlet)       org.springframework.spring-webmvc.test
RequestMappingHandlerAdapter (org.springframework.web.servlet.mvc.method.annotation)     org.springframework.spring-webmvc.main
SimpleControllerHandlerAdapter (org.springframework.web.servlet.mvc)                     org.springframework.spring-webmvc.main
SimpleServletHandlerAdapter (org.springframework.web.servlet.handler)                    org.springframework.spring-webmvc.main
```

handle方法执行方法讲解:

ha.handle(processedRequest, response, mappedHandler.getHandler());

主要步骤:

1. 解析参数（不同参数使用不同解析器处理）
2. 获取执行方法（getBean（）获取）
3. 反射调用执行invoke方法，执行过程结束。

> 根据session判断执行方法

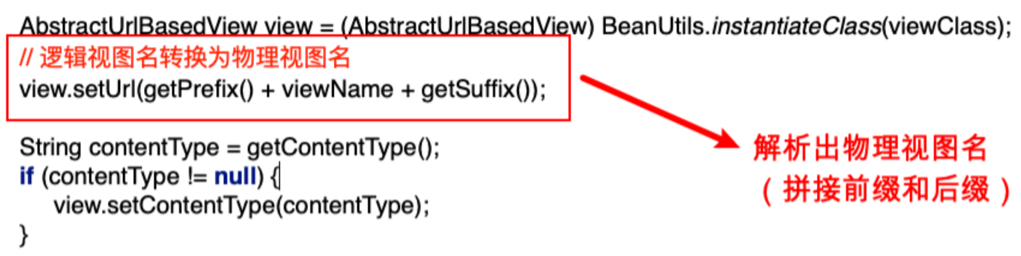
```
RequestMappingHandlerAdapter.java
    //该方法时ha.handle执行过程中的方法，主要是判断session的线程问题，具体需进一步了解；
    //核心方法是invokeHandlerMethod(request, response, handlerMethod);下面会讲解。
    @Override
    protected ModelAndView handleInternal(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws
Exception {

        ModelAndView mav;
        checkRequest(request);

        // 判断当前是否需要支持在同一个session中只能线性地处理请求
        if (this.synchronizeOnSession) {
            // 获取当前请求的session对象
            HttpSession session = request.getSession(false);
            if (session != null) {
                // 为当前session生成一个唯一的可以用于锁定的key
                Object mutex = WebUtils.getSessionMutex(session);
                synchronized (mutex) {
                    // 对HandlerMethod进行参数等的适配处理，并调用目标handler
                    mav = invokeHandlerMethod(request, response, handlerMethod);
                }
            }
            else {
                // 如果当前不存在session，则直接对HandlerMethod进行适配
                mav = invokeHandlerMethod(request, response, handlerMethod);
            }
        }
        else {
            // 如果当前不需要对session进行同步处理，则直接对HandlerMethod进行适配
            mav = invokeHandlerMethod(request, response, handlerMethod);
        }

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

        return mav;
    }
```

processDispatchResult()方法完成视图渲染跳转

主要步骤：

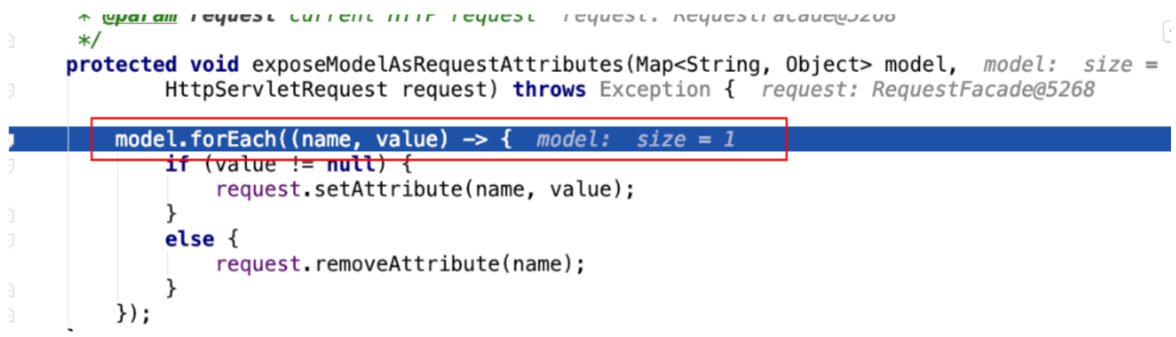
1. 视图解析器解析出View视图对象

2. 在解析出View视图对象的过程中会判断是否重定向、是否转发等，不同的情况封装的是不同的View实现
3. 解析出View视图对象的过程中，要将逻辑视图名解析为物理视图名



```
AbstractUrlBasedView view = (AbstractUrlBasedView) BeanUtils.instantiateClass(viewClass);
// 逻辑视图名转换为物理视图名
view.setUrl(getPrefix() + viewName + getSuffix());

String contentType = getContentType();
if (contentType != null) {
    view.setContentType(contentType);
}
```

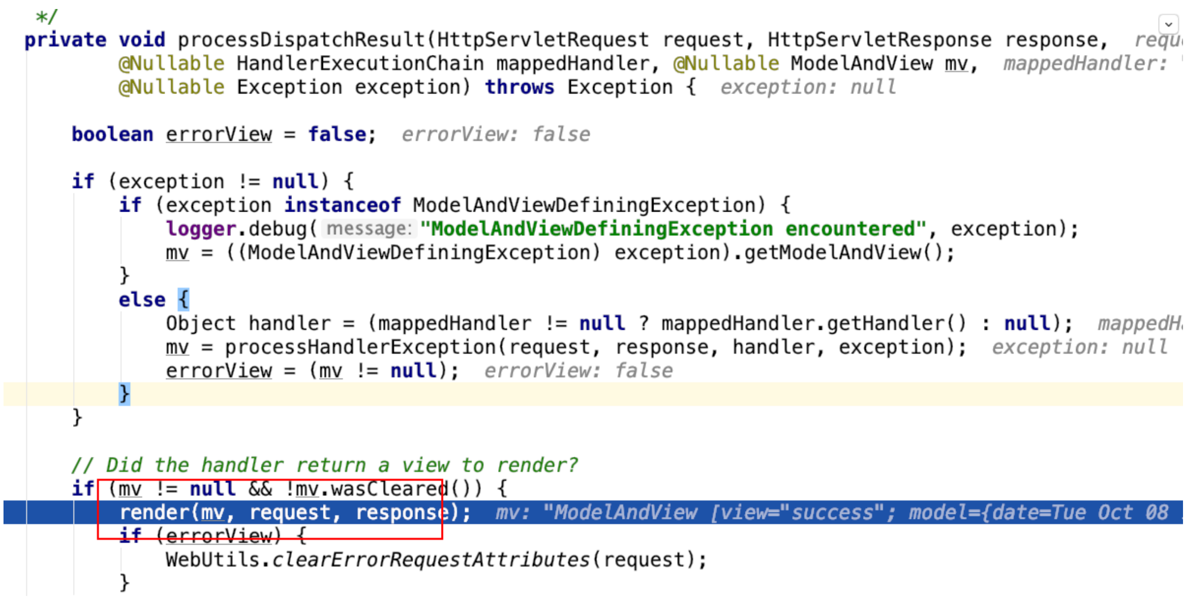
4. 渲染数据
5. 把modelMap中的数据暴露到request域中（这也是为什么后台model.add之后在jsp中可以从请求域取出来的根本原因）
6. 将数据设置到请求域中



```
 * @param request current HTTP request  request: RequestFacade@5268
 */
protected void exposeModelAsRequestAttributes(Map<String, Object> model,  model:  size =
        HttpServletRequest request) throws Exception {  request: RequestFacade@5268

    model.forEach((name, value) -> {  model:  size = 1
        if (value != null) {
            request.setAttribute(name, value);
        }
        else {
            request.removeAttribute(name);
        }
    });
```

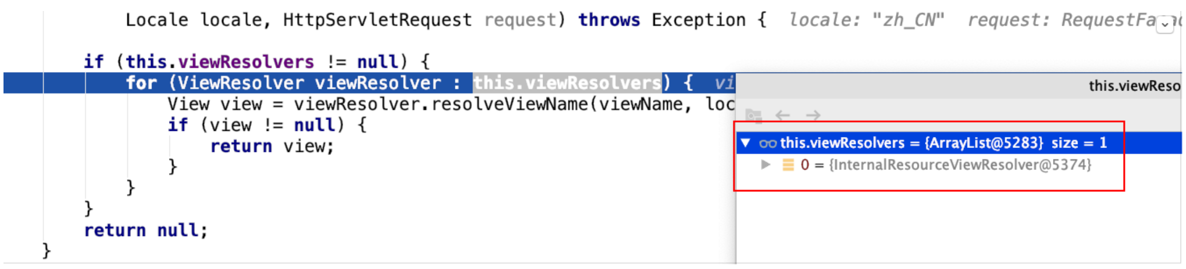
详细步骤：

render方法完成渲染



```
 */
private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,  requ
        @Nullable HandlerExecutionChain mappedHandler, @Nullable ModelAndView mv,  mappedHandler:
        @Nullable Exception exception) throws Exception {  exception: null

    boolean errorView = false;  errorView: false

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

    // Did the handler return a view to render?
    if (mv != null && !mv.wasCleared()) {
        render(mv, request, response);  mv: "ModelAndView [view="success"; model={date=Tue Oct 08
        if (errorView) {
            WebUtils.clearErrorRequestAttributes(request);
        }
```

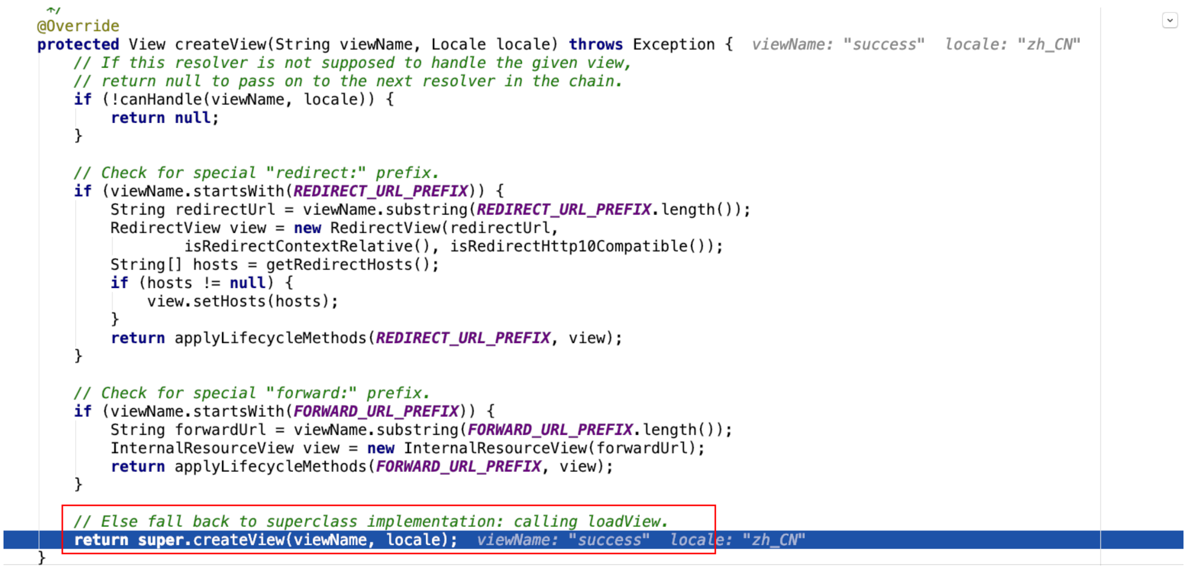
视图解析器解析出View视图对象



```
        Locale locale, HttpServletRequest request) throws Exception {  locale: "zh_CN"   request: RequestFa

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

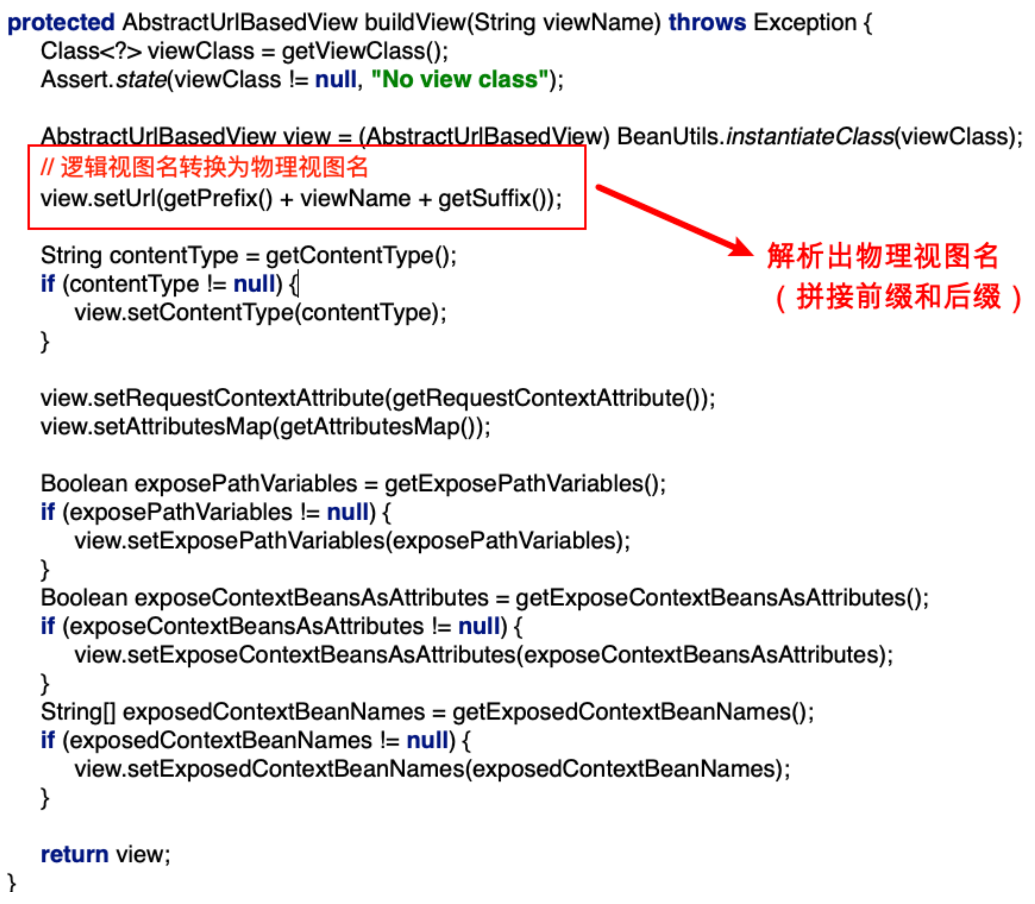
```
this.viewReso
▼ oo this.viewResolvers = {ArrayList@5283}  size = 1
  ▶ 0 = {InternalResourceViewResolver@5374}
```

在解析出View视图对象的过程中会判断是否重定向、是否转发等，不同的情况封装的是不同的View实现



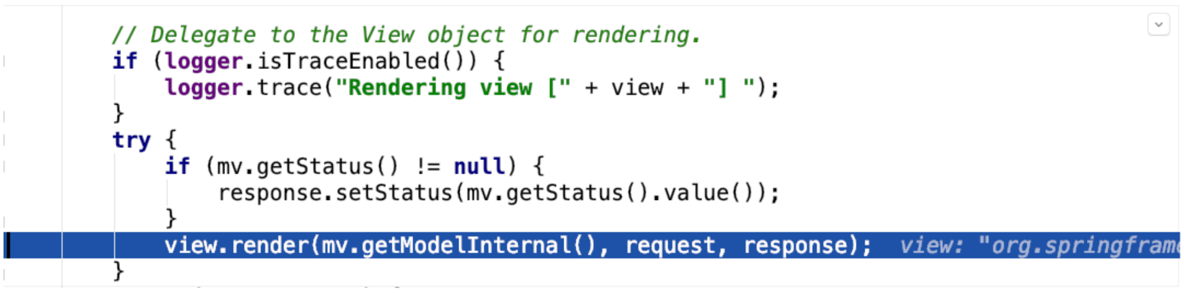
```
 */
@Override
protected View createView(String viewName, Locale locale) throws Exception {  viewName: "success"   locale: "zh_CN"
    // If this resolver is not supposed to handle the given view,
    // return null to pass on to the next resolver in the chain.
    if (!canHandle(viewName, locale)) {
        return null;
    }

    // Check for special "redirect:" prefix.
    if (viewName.startsWith(REDIRECT_URL_PREFIX)) {
        String redirectUrl = viewName.substring(REDIRECT_URL_PREFIX.length());
        RedirectView view = new RedirectView(redirectUrl,
                isRedirectContextRelative(), isRedirectHttp10Compatible());
        String[] hosts = getRedirectHosts();
        if (hosts != null) {
            view.setHosts(hosts);
        }
        return applyLifecycleMethods(REDIRECT_URL_PREFIX, view);
    }

    // Check for special "forward:" prefix.
    if (viewName.startsWith(FORWARD_URL_PREFIX)) {
        String forwardUrl = viewName.substring(FORWARD_URL_PREFIX.length());
        InternalResourceView view = new InternalResourceView(forwardUrl);
        return applyLifecycleMethods(FORWARD_URL_PREFIX, view);
    }

    // Else fall back to superclass implementation: calling loadView.
    return super.createView(viewName, locale);  viewName: "success"   locale: "zh_CN"
}
```

解析出View视图对象的过程中，要将逻辑视图名解析为物理视图名

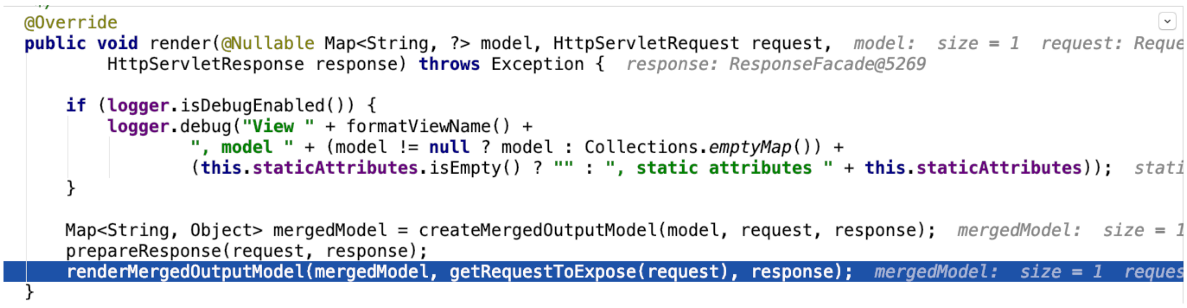


```
protected AbstractUrlBasedView buildView(String viewName) throws Exception {
    Class<?> viewClass = getViewClass();
    Assert.state(viewClass != null, "No view class");

    AbstractUrlBasedView view = (AbstractUrlBasedView) BeanUtils.instantiateClass(viewClass);
    // 逻辑视图名转换为物理视图名
    view.setUrl(getPrefix() + viewName + getSuffix());

    String contentType = getContentType();
    if (contentType != null) {
        view.setContentType(contentType);
    }

    view.setRequestContextAttribute(getRequestContextAttribute());
    view.setAttributesMap(getAttributesMap());

    Boolean exposePathVariables = getExposePathVariables();
    if (exposePathVariables != null) {
        view.setExposePathVariables(exposePathVariables);
    }
    Boolean exposeContextBeansAsAttributes = getExposeContextBeansAsAttributes();
    if (exposeContextBeansAsAttributes != null) {
        view.setExposeContextBeansAsAttributes(exposeContextBeansAsAttributes);
    }
    String[] exposedContextBeanNames = getExposedContextBeanNames();
    if (exposedContextBeanNames != null) {
        view.setExposedContextBeanNames(exposedContextBeanNames);
    }

    return view;
}
```

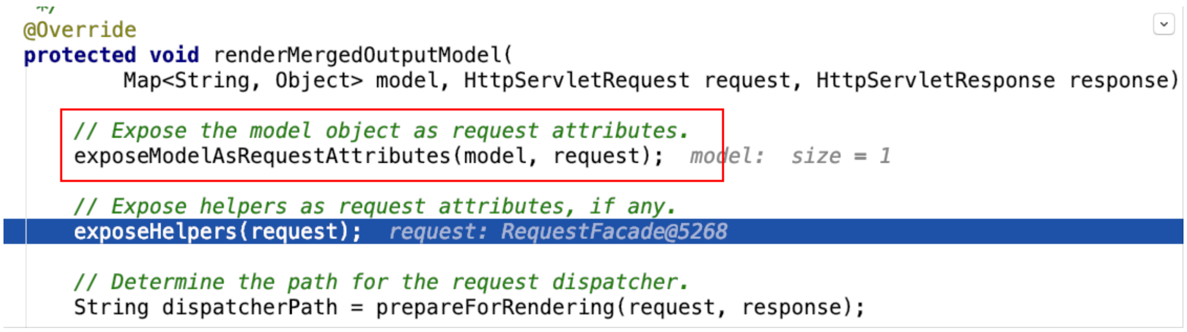
封装View视图对象之后，调用了view对象的render方法


```
// Delegate to the View object for rendering.
if (logger.isTraceEnabled()) {
    logger.trace("Rendering view [" + view + "] ");
}
try {
    if (mv.getStatus() != null) {
        response.setStatus(mv.getStatus().value());
    }
    view.render(mv.getModelInternal(), request, response);   view: "org.springfram
}
```

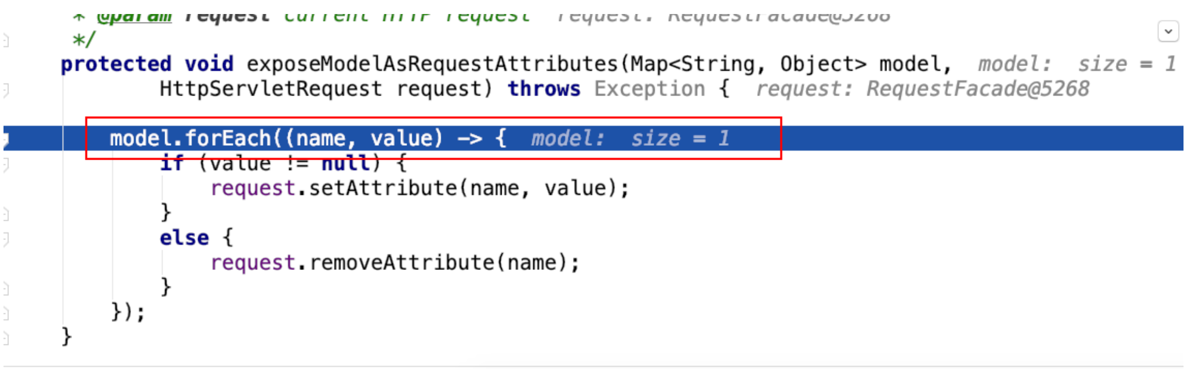
渲染数据


```
@Override
public void render(@Nullable Map<String, ?> model, HttpServletRequest request,   model:  size = 1   request: Reque
        HttpServletResponse response) throws Exception {   response: ResponseFacade@5269

    if (logger.isDebugEnabled()) {
        logger.debug("View " + formatViewName() +
                ", model " + (model != null ? model : Collections.emptyMap()) +
                (this.staticAttributes.isEmpty() ? "" : ", static attributes " + this.staticAttributes));   stati
    }

    Map<String, Object> mergedModel = createMergedOutputModel(model, request, response);   mergedModel:  size = 1
    prepareResponse(request, response);
    renderMergedOutputModel(mergedModel, getRequestToExpose(request), response);   mergedModel:  size = 1   reques
}
```

把modelMap中的数据暴露到request域中，这也是为什么后台model.add之后在jsp中可以从请求域取出来的根本原因


```
@Override
protected void renderMergedOutputModel(
        Map<String, Object> model, HttpServletRequest request, HttpServletResponse response)

    // Expose the model object as request attributes.
    exposeModelAsRequestAttributes(model, request);   model:  size = 1

    // Expose helpers as request attributes, if any.
    exposeHelpers(request);   request: RequestFacade@5268

    // Determine the path for the request dispatcher.
    String dispatcherPath = prepareForRendering(request, response);
```

将数据设置到请求域中


```
 * @param request current HTTP request   request: RequestFacade@5268
 */
protected void exposeModelAsRequestAttributes(Map<String, Object> model,   model:  size = 1
        HttpServletRequest request) throws Exception {   request: RequestFacade@5268

    model.forEach((name, value) -> {   model:  size = 1
        if (value != null) {
            request.setAttribute(name, value);
        }
        else {
            request.removeAttribute(name);
        }
    });
}
```

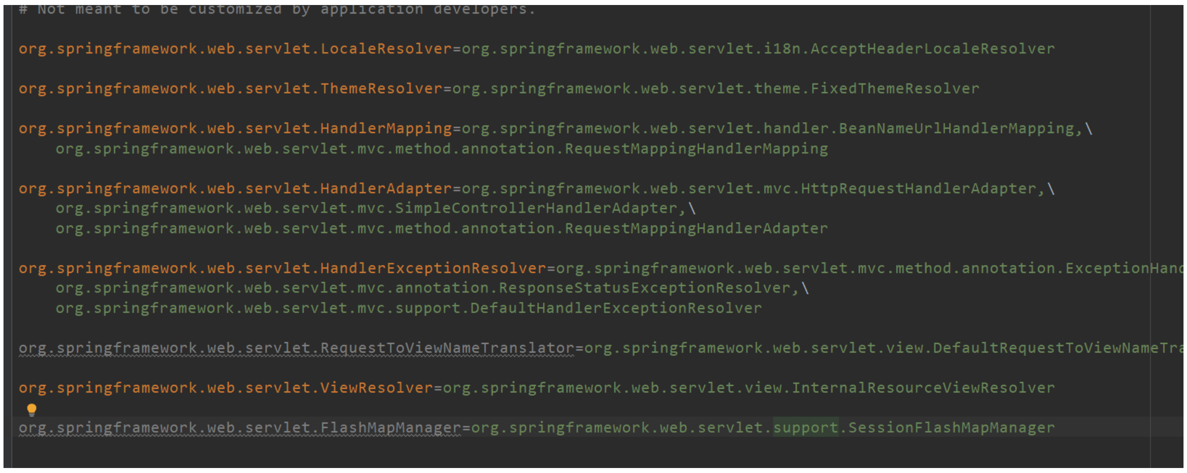
MVC九大组件初始化时机：

```
DispatcherServlet.java
    //多部件解析器
    @Nullable
    private MultipartResolver multipartResolver;

    //国际化解析器
    @Nullable
    private LocaleResolver localeResolver;
```

```java

    //主题解析器
    @Nullable
    private ThemeResolver themeResolver;

    //处理映射器
    @Nullable
    private List<HandlerMapping> handlerMappings;

    //处理适配器组件
    @Nullable
    private List<HandlerAdapter> handlerAdapters;

    //异常解析器组件
    @Nullable
    private List<HandlerExceptionResolver> handlerExceptionResolvers;

    //默认视图名转换器组件
    @Nullable
    private RequestToViewNameTranslator viewNameTranslator;

    //flash属性管理舰
    @Nullable
    private FlashMapManager flashMapManager;

    //视图解析器
    @Nullable
    private List<ViewResolver> viewResolvers;

//上述九大组件接口
```

九大组件初始化细节：

```java
    //该方法spring源码的refresh()方法有调用过。
    protected void onRefresh(ApplicationContext context) {
        // 初始化策略
        initStrategies(context);
    }

    /**
     * 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);
        // 初始化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);
        // 初始化 FlashMap 管理器
        initFlashMapManager(context);
    }

//举例initHandlerMappings，其他都差不多，initMultipartResolver除外（单独讲）。
private void initHandlerMappings(ApplicationContext context) {
        this.handlerMappings = null;

        if (this.detectAllHandlerMappings) {
            // 找到所有实现HandlerMapping接口的类
            Map<String, HandlerMapping> matchingBeans =
                    BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerMapping.class, true, false);
            if (!matchingBeans.isEmpty()) {
                this.handlerMappings = new ArrayList<>(matchingBeans.values());
                AnnotationAwareOrderComparator.sort(this.handlerMappings);
            }
        }
        else {
            try {
                // 否则在ioc中按照固定名称去找
                HandlerMapping hm = context.getBean(HANDLER_MAPPING_BEAN_NAME, HandlerMapping.class);
                this.handlerMappings = Collections.singletonList(hm);
            }
            catch (NoSuchBeanDefinitionException ex) {
            }
        }

        if (this.handlerMappings == null) {
            // 最后还为空则按照默认策略生成
            this.handlerMappings = getDefaultStrategies(context, HandlerMapping.class);
        }
    }

private void initMultipartResolver(ApplicationContext context) {
        try {
            //只能从ioc获取，且配置文件的id一定为multipartResolver才可以。
            this.multipartResolver = context.getBean(MULTIPART_RESOLVER_BEAN_NAME, MultipartResolver.class);//其他不用看，MULTIPART_RESOLVER_BEAN_NAME=“multipartResolver”，该初始化方法只能通过配置文件拿bean，所以在配置文件的id一定为multipartResolver才可以。
            if (logger.isTraceEnabled()) {
                logger.trace("Detected " + this.multipartResolver);
            }
            else if (logger.isDebugEnabled()) {
                logger.debug("Detected " + this.multipartResolver.getClass().getSimpleName());
            }
        }
        catch (NoSuchBeanDefinitionException ex) {
            // Default is no multipart resolver.
            this.multipartResolver = null;
            if (logger.isTraceEnabled()) {
                logger.trace("No MultipartResolver '" + MULTIPART_RESOLVER_BEAN_NAME + "' declared");
            }
        }
    }
```



```
# Not meant to be customized by application developers.

org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeResolver

org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\
    org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\
    org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
    org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.method.annotation.ExceptionHand
    org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
    org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTra

org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver

org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFlashMapManager
```

图1.1

九大组件初始化总结（除了initMultipartResolver）：
  找到实现类；
  如果没找到，在ioc中按照固定名称去找；
  还没找到，如图1.1根据properties文件默认配置去找。

initMultipartResolver初始化：

  只能从ioc获取，且配置文件的id一定为multipartResolver才可以。

session线程不安全，session、cookie了解？？？

springmvc使用session加锁：

实现session同步：给session生成唯一key，对key加锁，里面执行具体逻辑。

```
        // 判断当前是否需要支持在同一个session中只能线性地处理请求
        if (this.synchronizeOnSession) {
            // 获取当前请求的session对象
            HttpSession session = request.getSession(false);
            if (session != null) {
                // 为当前session生成一个唯一的可以用于锁定的key
                Object mutex = WebUtils.getSessionMutex(session);
                synchronized (mutex) {
                    // 对HandlerMethod进行参数等的适配处理，并调用目标handler
                    mav = invokeHandlerMethod(request, response, handlerMethod);
                }
            }
            else {
                // No HttpSession available -> no mutex necessary
                // 如果当前不存在session，则直接对HandlerMethod进行适配
                mav = invokeHandlerMethod(request, response, handlerMethod);
            }
        }
```