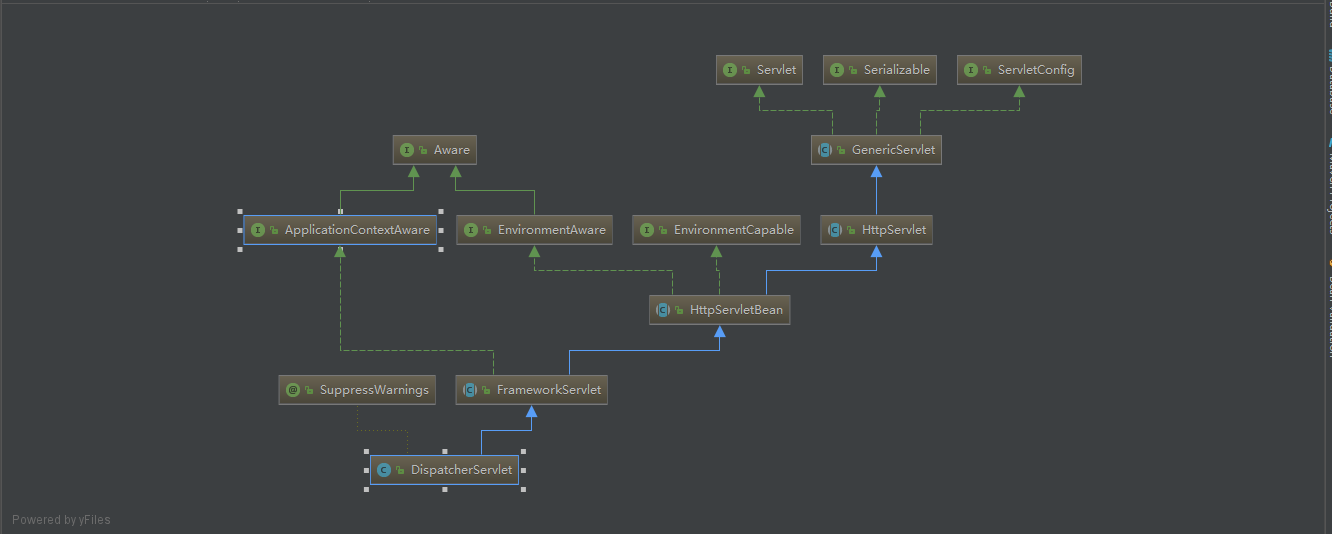
# 深入理解SpringMVC源码

## DispatcherServlet



HttpServletBean实现了：EnvironmentCapable和EnvironmentAware

FrameworkServlet实现了ApplictionContexgtAware接口

XXXAware：在Spring里表示对XXX可以感知，通俗点是：如果在某个类里面想要使用Spring的一些东西，

就可以通过实现XXXAware接口告诉Spring，Spring看到后就会自动给你传送过来，而接受方只需要通过实现接口唯一方法setXXX。

EnvironmentCapable：具有Environment的能力，也就是可以提供Environment。

### HttpServletBean

Servlet创建的时候可以直接调用无参的init方法。

**Ps:我将代码中一些重要的代码删除了只留下核心代码。**

public final void init() throws ServletException

{  
 try {

//将servelet中配置的参数封装到pvs中。  
 PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.requiredProperties);  
 BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);  
 ResourceLoader resourceLoader = new ServletContextResourceLoader(getServletContext());  
 bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, getEnvironment()));  
 //模板方法，可以在子类中调用，做一些初始化工作。Bw代表这DispatcherServlet

initBeanWrapper(bw);

//将设置的初始值赋予DispatcherServlet  
 bw.setPropertyValues(pvs, true);  
 }  
 catch (BeansException ex) {  
 logger.error("Failed to set bean properties on servlet '" + getServletName() + "'", ex);  
 throw ex;  
 }  
 // 模板方法，子类初始化的入口方法  
 initServletBean();  
}

### FrameworkServlet

初始化入口方法：initServletBean

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 {

//初始化WebApplicationContext  
 this.webApplicationContext = initWebApplicationContext();

//初始化FrameworkServlet，该方法为模板方法，子类必须实现  
 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");  
 }  
}

初始化WebApplicationContext方法：

protected WebApplicationContext initWebApplicationContext()

{  
 WebApplicationContext rootContext =  
 WebApplicationContextUtils.getWebApplicationContext(getServletContext());  
 WebApplicationContext wac = null;  
//通过构造方法设置了webApplicationContext  
 if (this.webApplicationContext != null) {  
 // A context instance was injected at construction time -> use it  
 wac = this.webApplicationContext;  
 if (wac instanceof ConfigurableWebApplicationContext) {  
 ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) wac;  
 if (!cwac.isActive()) {  
 // The context has not yet been refreshed -> provide services such as  
 // setting the parent context, setting the application context id, etc  
 if (cwac.getParent() == null) {  
 // The context instance was injected without an explicit parent -> set  
 // the root application context (if any; may be null) as the parent  
 cwac.setParent(rootContext);  
 }  
 configureAndRefreshWebApplicationContext(cwac);  
 }  
 }  
 }  
 if (wac == null) {  
 / /当webApplicationContext已经在ServletContext中时，通过配置在Servlet中的contextAttribute参数获取  
 wac = findWebApplicationContext();  
 }  
 if (wac == null) {  
 // 如果还是没有创建，则创建一个  
 wac = createWebApplicationContext(rootContext);  
 }  
  
 if (!this.refreshEventReceived) {  
 //ContextRefreshedEvent事件没有触发调用该方法，模板方法，子类实现

onRefresh(wac);  
 }  
  
 if (this.publishContext) {  
 //将webApplicationContext保存到ServletContext中

String attrName = getServletContextAttributeName();  
 getServletContext().setAttribute(attrName, wac);  
 if (this.logger.isDebugEnabled()) {  
 this.logger.debug("Published WebApplicationContext of servlet '" + getServletName() +  
 "' as ServletContext attribute with name [" + attrName + "]");  
 }  
 }  
  
 return wac;  
}

该方法一共做了三件事：1.获取Spring的根容器rootContext

2.设置webApplictionContext并根据情况调用onRefresh方法

3.将webApplicatinContext设置到ServletContext中

**获取Spring容器的根容器rootContext：**默认情况下Spring会将自己的容器设置成为ServletContext的属性，默认根容器的key为org.springframework.web.context.WebApplicationContext.ROOT。

在WebApplicationContext中定义：

String *ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE* = WebApplicationContext.class.getName() + ".ROOT";

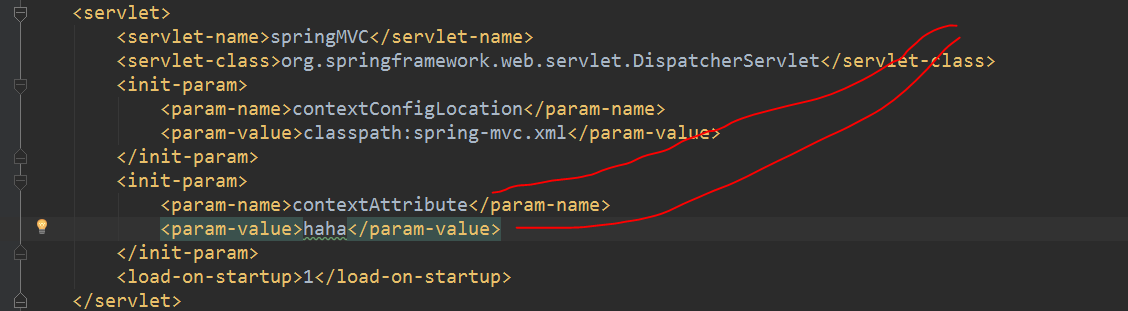
**设置webApplicationContext并根据情况调用onRefresh方法：**

**第一种方法：**在构造方法中传递webApplicationContext参数。这是只需要进行一些设置即可。

**第二种方法：**webApplicationContext已经在ServletContext中。这时只需要在配置Servlet的时候将ServletContext中的

webApplicationContext的name配置到contextAttribut属性就可以了。

如：在ServletContext中配置一个叫wap的webApplicationContext



**第三种方式：**自己创建一个。正常情况下使用该方式。

这里我对核心代码进行的简化，保留了核心代码：

protected WebApplicationContext createWebApplicationContext(ApplicationContext parent) {

//获取创建的类型，这里是XmlWebApplicationContext

Class<?> contextClass = getContextClass();

//检查创建的类型

if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {}

//具体的创建

ConfigurableWebApplicationContext wac =  
 (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);  
 wac.setEnvironment(getEnvironment());  
 wac.setParent(parent);

//将设置的contextConfigLocation参数传递给wac，默认情况下WEB-INF/[ServletNam]-Servlet.xml

wac.setConfigLocation(getContextConfigLocation());  
 configureAndRefreshWebApplicationContext(wac);  
 return wac;  
}

protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac) {  
 if (ObjectUtils.identityToString(wac).equals(wac.getId())) {  
 // The application context id is still set to its original default value  
 // -> assign a more useful id based on available information  
 if (this.contextId != null) {  
 wac.setId(this.contextId);  
 }  
 else {  
 // Generate default id...  
 wac.setId(ConfigurableWebApplicationContext.APPLICATION\_CONTEXT\_ID\_PREFIX +  
 ObjectUtils.getDisplayString(getServletContext().getContextPath()) + "/" + getServletName());  
 }  
 }  
 wac.setServletContext(getServletContext());  
 wac.setServletConfig(getServletConfig());  
 wac.setNamespace(getNamespace());

//添加监听器  
 wac.addApplicationListener(new SourceFilteringListener(wac, new ContextRefreshListener()));  
 ConfigurableEnvironment env = wac.getEnvironment();  
 if (env instanceof ConfigurableWebEnvironment) {  
 ((ConfigurableWebEnvironment) env).initPropertySources(getServletContext(), getServletConfig());  
 }  
  
 postProcessWebApplicationContext(wac);  
 applyInitializers(wac);  
 wac.refresh();  
}

SourceFilteringListener可以根据传入的参数进行选择，所以实际监听的是ContextRefreshListener所监听的事件，

ContextRefreshListener是FrameworkServlet的内部类，监听ContextRefreshEvent事件，当接受到消息时调用FrameworkServlet的onApplicationEvent方法，在onApplicationEvent中会调用一次onRefresh方法，并将refreshEventReceived

标志设置为true，表示已经refresh过。

private class ContextRefreshListener implements ApplicationListener<ContextRefreshedEvent> {  
  
 @Override  
 public void onApplicationEvent(ContextRefreshedEvent event) {  
 FrameworkServlet.this.onApplicationEvent(event);  
 }  
}

public void onApplicationEvent(ContextRefreshedEvent event) {  
 this.refreshEventReceived = true;  
 onRefresh(event.getApplicationContext());  
}

我们可以知道在第三种方法进行创建WebApplicaitonContext时已经进行了refresh，所在在initWebApplicationContext中

if (!this.refreshEventReceived) {  
 onRefresh(wac);  
}

不需要调用onRefresh方法，并且第一种方法也调用了configureAndRefreshWebApplicationContext方法，也进行了refresh，所以只有第二种方法创建WebApplicationContext的时候才会调用onRefresh方法。最终onRefresh方法只会调用一次。而且

DispatcherServlet正式通过重写这个模板方法来实现初始化的。

**将WebApplicationContext设置到ServletContext中：**

根据publishContext标志判断是否将创建出来的webApplicationContext设置到ServletContext的属性中，publicContext标志可以通过在配置Servlet是通过init-param参数进行设置，HttpServletBean在初始化时会将其设置的publishContext参数

之所以将webApplicaitonContext设置到ServletContext属性中，主要是为了方便获取。

我们可以在配制Servlet时设置一些初始化参数：

* contextAttribute：在ServletContext中，要用作WebApplicationContext的属性名称
* contextClass：创建WebApplicationContext的类型
* contextConfigLocation：Spring MVC配置文件所在的位置
* publishContext：是否将webApplicationContext设置到ServletContext的属性。

### DispatcherServlet

onRefresh方法时DispatcherServlet的入口方法。

@Override  
protected void onRefresh(ApplicationContext context) {  
 initStrategies(context);  
}  
protected void initStrategies(ApplicationContext context) {  
 initMultipartResolver(context);  
 initLocaleResolver(context);  
 initThemeResolver(context);  
 initHandlerMappings(context);  
 initHandlerAdapters(context);  
 initHandlerExceptionResolvers(context);  
 initRequestToViewNameTranslator(context);  
 initViewResolvers(context);  
 initFlashMapManager(context);  
}

为啥不讲initStrategies方法实现不直接写入到onRefresh中？

答：onRefresh方法主要用来刷新容器的，initStrategies方法用来初始化一些策略组件。如果吧initStrategies实现直接写入到onRefresh中也不会有特殊影响，不过这样一来，如果想要在onRefresh方法中添加别的功能，就会没有将其单独写入一个方法出来逻辑清晰，不过这不是最重要的，如果在其他地方也需要调用initStrategies方法，但initStrategies方法没有被独立出来，就会调用onRefresh方法，那么onRefresh增加了新功能就麻烦了。另外单独将initStragegies写出来可以被子类覆盖，使用新的模式进行初始化就比较简单。

#### initStrategies方法详解

protected void initStrategies(ApplicationContext context) {  
 initMultipartResolver(context);  
 initLocaleResolver(context);  
 initThemeResolver(context);  
 initHandlerMappings(context);  
 initHandlerAdapters(context);  
 initHandlerExceptionResolvers(context);  
 initRequestToViewNameTranslator(context);  
 initViewResolvers(context);  
 initFlashMapManager(context);  
}

该方法主要用完成DispatcherServlet中变量的初始化工作。

initHandlerMappings：将处理器映射器添加到List<HandlerMapping> handlerMappings变量中。

##### initLocaleResolver

protected <T> List<T> getDefaultStrategies(ApplicationContext context, Class<T> strategyInterface)

{  
 String key = strategyInterface.getName();

//从defaultStrategies中获取所需策略的类型

String value = defaultStrategies.getProperty(key);  
 if (value != null) {

//如果有多个默认值，则以逗号进行分割为数组

String[] classNames = StringUtils.commaDelimitedListToStringArray(value);  
 List<T> strategies = new ArrayList<T>(classNames.length);

//按照获取到的类型初始化策略

for (String className : classNames) {  
 try {  
 Class<?> clazz = ClassUtils.forName(className, DispatcherServlet.class.getClassLoader());  
 Object strategy = createDefaultStrategy(context, clazz);  
 strategies.add((T) strategy);  
 }  
 catch (ClassNotFoundException ex) {}  
 catch (LinkageError err) { }  
 }  
 return strategies;  
 }  
 else {  
 return new LinkedList<T>();  
 }  
}

该方法返回值为List，这是因为HandlerMapping等组件可以有多个，通过仔细的分析，我们可以发现该方法的主要调用了

ClassUtil.forName，它需要接受一个className参数，而className来自于classNames，而classNames来自于value，而value来自于defaultStrategies，defaultStrategies是一个静态属性，在static代码块中进行了初始化。

static {  
 try {  
 ClassPathResource resource = new ClassPathResource(*DEFAULT\_STRATEGIES\_PATH*, DispatcherServlet.class);  
 *defaultStrategies* = PropertiesLoaderUtils.*loadProperties*(resource);  
 }  
 catch (IOException ex) {}  
}

我们可以发现defaultStrategies里面存放的是org.springframework.web.DispatcherServlet.properties里面的定义的键值对。

**分析DispatcherServlet.properties属性文件：**

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.annotation.DefaultAnnotationHandlerMapping  
  
org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\  
 org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\  
 org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter  
  
org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\  
 org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\  
 org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver  
  
org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator  
  
org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver  
  
org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFlashMapManager

该属性文件中定义了八个组件，处理上传的MultipartResolver是没有默认配置的，这是因为并不是每一个应用都需要上传功能，即使需要上传也不一定需要MultipartResolver，所以MultipartResolver不需要默认配制。

注意：默认配置不一定是最优配置，也不是Spring的推荐配制，只是在没有配制的情况下有个默认配制，不至于空着。里面有的配制已经被标注@Deprecated，表示已经被弃用。如：DefaultAnnotationHandlerMapping，AnnotationMethodHandlerAdapter等。

在我们使用<mvc:annotation-driver/>后，并不会全部使用默认配制。因为它配置了HandlerMapping，HandlerAdapter和HandlerExceptionResolver，而且还做了很多别的工作。我们可以通过查看：**org.springframework.scheduling.config.AnnotationDrivenBeanDefinitionParser**

## SpringMVC处理请求分析

该章节主要介绍了Spring MVC如何处理客户端请求：HttpServletBean，FrameworkServlet，DispatcherServlet

HttpServletBean主要参与了创建工作，并没有涉及到请求处理。

Servlet在处理请求时，首先从Service接口的service方法开始，然后在HttpServlet的service方法中根据请求的类型将不同的请求路由到不同的方法中，如doGet，doPost，doHead，doPut，doDelete等。

### FrameworkServlet

FrameworkServlet重写了service，doGet，doHead，doPost，doPut，doDelete，doOptions和doTrace方法。在service方法中添加了PATCH类型请求处理，其他类型的请求直接交给父类进行处理

@Override  
protected void service(HttpServletRequest request, HttpServletResponse response)  
 throws ServletException, IOException {  
  
 if (HttpMethod.*PATCH*.matches(request.getMethod())) {  
 processRequest(request, response);  
 }  
 else {  
 super.service(request, response);  
 }  
}

通过源码，我们发现与HttpServlet将不同类型的请求路由到不同方法进行处理的思路正好相反，这里将所有请求合并到了processRequest方法，不能说明SpringMVC中就不对request的类型进行分类了，而全部执行相同的操作，恰恰相反，SpringMVC中对不同类型的请求支持的非常好，它将不同的请求交给不同的Handle处理。

HttpServer中的service方法将不同的请求路由到不同 的方法中，如果在service方法不调用super.service()方法，而是直接交给processRequest方法进行处理不是更加简单吗？从现有的角度看确实如此，不过那么这样做的话或许存在着一些问题，比如：我们为了某种特殊需求在Post请求处理前对request做一些处理，这时可能会创建一个继承自DispathcerServlet类，然后覆盖doPost方法，在里面先对request进行处理，然后调用super.doPost方法，但是父类根本就没调用doPost，所以这时候就会出问题了。虽然这个问题的解决方法也很简单，但是按照正常的逻辑，调用doPost应该可以完成才合理，而且一般情况下开发者不需要对Spring MVC内部的结构非常了解，所以Spring MVC的这种做法虽然看起来笨重但是必要的。

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();  
 }  
  
 if (logger.isDebugEnabled()) {  
 if (failureCause != null) {  
 this.logger.debug("Could not complete request", failureCause);  
 }  
 else {  
 if (asyncManager.isConcurrentHandlingStarted()) {  
 logger.debug("Leaving response open for concurrent processing");  
 }  
 else {  
 this.logger.debug("Successfully completed request");  
 }  
 }  
 }  
  
 publishRequestHandledEvent(request, response, startTime, failureCause);  
 }  
}

### DispatcherServlet

DispatcherServlet是Spring MVC的核心类，DispatcherServlet执行处理的入口方法时doService

protected void doService(HttpServletRequest request, HttpServletResponse response) throws Exception

{  
   
 Map<String, Object> attributesSnapshot = null;

//判断请求是否为include请求，如果是则对请求的Attribute做个快照  
 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));  
 }  
 }  
 }  
  
 // 对request设置了一些属性  
 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()) {  
 // 还原request快照属性  
 if (attributesSnapshot != null) {  
 restoreAttributesAfterInclude(request, attributesSnapshot);  
 }  
 }  
 }  
}

从该方法中我们可以看出doService并没有直接对请求进行处理，而是交给了doDispatch方法处理，在处理之前对请求进行了一些操作：1.对request设置了webApplicationContext，localResolver，themeResolver和themeSource

2.设置flashMap相关属性，该属性主要用于Redirect转发时参数的传递，

inputFlashMap用来保存上次请求中转发过来的属性，outputFlashMap用来保存本次请求需要转发的属性，FlashMapManager用来管理它们。

#### 小小知识点（FlashMap）

解决表单重复提交问题解决方法之一：在我们处理完表单的post请求之后，我们可以redirect到一个get请求，这样即使用户刷新也不会重复提交的问题，然而在redirect时需要传递一些参数，如果按照普通方式，就只能将参数写入url中，但是url有长度的限制，另外有些场景，我们并不像将参数暴露在url中，这时我们就可以使用FlashMap来进行参数传递，我们只需要在redirect之前将需要传递的参数写入到OUTPUT\_FLASH\_MAP\_ATTRIBUTE中：

model.addAttribute("message","hello world") ;  
((FlashMap)((ServletRequestAttributes)(RequestContextHolder.*getRequestAttributes*()))  
 .getRequest().getAttribute(DispatcherServlet.*OUTPUT\_FLASH\_MAP\_ATTRIBUTE*)).put("name","hao");

这样Spring会自动将其设置到model里

还有一种更加简单的方法，我们只需要在handler方法参数中定义RedirecAttributes类型变量，然后将需要传递的参数设置到里面就可以了，之后Spring会自动完成。RedirectAttributes有两种设置参数的方法：1.addAttribute(key,value) 2.addFlashAttribute(key,value)，用第一个方法设置的参数会拼接到url中，第二个方法将我们设置的参数用FlashMap保存。

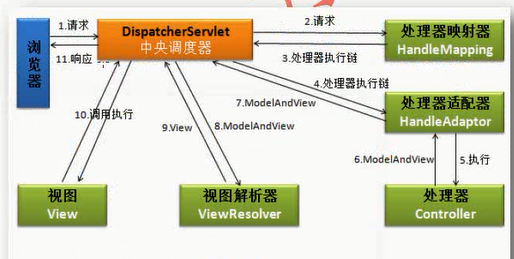
@RequestMapping(value="/submit",method= RequestMethod.*POST*)  
public String submit(RedirectAttributes attr, @RequestParam("username")String username ,  
 @RequestParam("password")String password){  
 //将参数orderId添加到FlashMap中  
 attr.addFlashAttribute("username",username);  
 //将参数拼接到url上  
 attr.addAttribute("password",password);  
 return "redirect:showorders";  
}  
@RequestMapping(value="/showorders",method= RequestMethod.*GET*)  
public String showOrders(Model model){  
 System.*out*.println("username:");  
 return "orders";  
}



重定向的参数都封装到Model中，我们可以在jsp中直接获得。

完毕

#### doDispatch方法详解



在对request设置完属性后，具体对请求的处理时doDispatch方法进行的：

protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception

{

//实际处理时所用的request，如果不是上传请求，则直接使用接受到的request，否则包装成上传类型的request  
 HttpServletRequest processedRequest = request;

//处理请求的处理器链（包含处理器和对应的Interceptor）  
 HandlerExecutionChain mappedHandler = null;

//是不是上传请求标志  
 boolean multipartRequestParsed = false;  
  
 WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);  
  
 try {

//封装Model和View的容器。  
 ModelAndView mv = null;

//处理请求过程抛出的异常  
 Exception dispatchException = null;  
  
 try {

//检查是否为上传请求  
 processedRequest = checkMultipart(request);  
 multipartRequestParsed = (processedRequest != request);  
  
 // 根据请求找到相应的Handler  
 mappedHandler = getHandler(processedRequest);  
 if (mappedHandler == null || mappedHandler.getHandler() == null) {  
 noHandlerFound(processedRequest, response);  
 return;  
 }  
  
 // 根据Handler找到HandlerAdapter  
 HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());  
  
 // 处理get,head请求的Last-Modifys  
 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;  
 }  
 }  
 //执行相应Interceptor的preHandle  
 if (!mappedHandler.applyPreHandle(processedRequest, response)) {  
 return;  
 }  
  
 //HandlerAdapter使用Handler处理请求

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

//如果需要异步，则直接返回  
  
 if (asyncManager.isConcurrentHandlingStarted()) {  
 return;  
 }  
 //如果view为空时（比如，Handler返回值为void），根据request设置默认view  
 applyDefaultViewName(processedRequest, mv);

//执行相应的Interceptor的postHandle  
 mappedHandler.applyPostHandle(processedRequest, response, mv);  
 }  
 catch (Exception ex) {  
 dispatchException = ex;  
 }

//处理返回结果，包括处理异常，渲染页面，发出完成通知触发Interceptor的afterCompletion  
 processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);  
 }  
 catch (Exception ex) {  
 triggerAfterCompletion(processedRequest, response, mappedHandler, ex);  
 }  
 catch (Error err) {  
 triggerAfterCompletionWithError(processedRequest, response, mappedHandler, err);  
 }  
 finally {

//判断是否执行异步请求  
 if (asyncManager.isConcurrentHandlingStarted()) {  
 // Instead of postHandle and afterCompletion  
 if (mappedHandler != null) {  
 mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);  
 }  
 }  
 else {  
 //删除上传请求的资源

if (multipartRequestParsed) {  
 cleanupMultipart(processedRequest);  
 }  
 }  
 }  
}

**该方法**最核心的代码只有4句，它们的主要任务时：1.根据request找到Handler。2.根据Handler找到对应的HandlerAdapter

3.用HandlerAdapter到Handler。4.调用processDispatcherResult方法处理上面处理之后的结果（包含找到View并渲染输出给用户）

mappedHandler = getHandler(processedRequest);

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

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

processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);

HttpServletRequest processedRequest：实际处理时所用的request，如果不是上传请求，则直接使用接受到的request，否则封装为上传类型的request。

HandlerExecutionChain mappedHandler：处理请求的处理器链（包含了处理器和对应的Interceptor）。

Boolean multipartRequestParsed：是否为上传请求的标志。

ModelAndView mv：封装了Model和View的容器。

Exception dispatchException：处理请求过程抛出的异常。注意：它不包括渲染过程中抛出的异常。

**//获取处理器执行链**

protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception

{  
 for (HandlerMapping hm : this.handlerMappings) {  
 HandlerExecutionChain handler = hm.getHandler(request);  
 if (handler != null) {  
 return handler;  
 }  
 }  
 return null;  
}

**//获取处理器适配器**

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

**该方法的异常处理：**

doDispatch有两层异常捕获，内层是捕获在对请求进行处理的过程中抛出的异常，外层主要是在处理渲染页面时抛出的异常。

内层的异常，也就是执行请求处理时的异常会设置到dispatcheException变量，然后在processDispatchResult方法中进行处理，

外层则是处理processDispatchResult方法抛出异常。

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

{  
 boolean errorView = false;

//如果请求处理的过程中有异常抛出则处理异常  
 if (exception != null) {

//判断异常类型，该异常是人为指定的  
 if (exception instanceof ModelAndViewDefiningException) {  
 logger.debug("ModelAndViewDefiningException encountered", exception);  
 mv = ((ModelAndViewDefiningException) exception).getModelAndView();  
 }  
 else {  
 Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);

//处理其他类型的异常  
 mv = processHandlerException(request, response, handler, exception);  
 errorView = (mv != null);  
 }  
 }  
  
 //渲染页面

if (mv != null && !mv.wasCleared()) {  
 render(mv, request, response);  
 if (errorView) {  
 WebUtils.clearErrorRequestAttributes(request);  
 }  
 }  
 else { }  
 if (WebAsyncUtils.getAsyncManager(request).isConcurrentHandlingStarted()) {  
 return;  
 }  
 if (mappedHandler != null) {  
 mappedHandler.triggerAfterCompletion(request, response, null);  
 }  
}

渲染页面具体在render方法中执行，render方法首先对response设置Locale，然后判断View如果是String类型则调用resolveViewName方法是用ViewResolver得到实际的View，最后调用View的render方法对页面进行具体的渲染。

protected ModelAndView processHandlerException(HttpServletRequest request, HttpServletResponse response,  
 Object handler, Exception ex) throws Exception {  
 ModelAndView exMv = null;  
 for (HandlerExceptionResolver handlerExceptionResolver : this.handlerExceptionResolvers) {  
 exMv = handlerExceptionResolver.resolveException(request, response, handler, ex);  
 if (exMv != null) {  
 break;  
 }  
 }  
 if (exMv != null) {  
 if (exMv.isEmpty()) {  
 request.setAttribute(EXCEPTION\_ATTRIBUTE, ex);  
 return null;  
 }  
 if (!exMv.hasView()) {  
 exMv.setViewName(getDefaultViewName(request));  
 }  
 WebUtils.exposeErrorRequestAttributes(request, ex, getServletName());  
 return exMv;  
 }  
 throw ex;  
}

该方法会遍历所有的HandlerExceptionResolver，判断是否能解决该异常，

### 总结

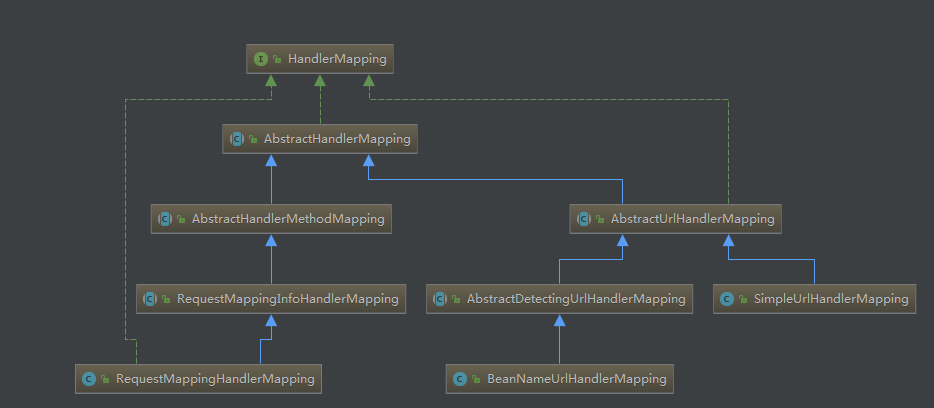
HttpServletBean：没有参与实际请求的处理

FrameworkServlet：将不同类型的请求合并到proecssRequest方法统一处理，该方法具体完成的工作：1.调用doService模板方法具体处理请求。2.将当前请求的LocaleContext和ServletRequestAttributes在处理请求前设置到了LocaleContextHolder和RequestContextHolder中，并在请求处理完成后恢复。3.请求处理后发布了ServletRequestHandledEvent消息。

DispacherServlet：doService方法给request方法设置了一些属性并将请求交给doDispatch方法具体处理。

## HanlerMapping

作用：根据request找到相应的处理器Handler和Interceptors。一个处理器处理器总是包装了一个HandlerExecutionChain实例，一个可选的HandlerInterceptor实例。



### AbstractHandlerMapping

AbstractHandlerMapping是HandlerMapping的抽象实现，所有的HandlerMapping都继承自AbstractHandlerMapping。

AbstractHandlerMapping采用模板模式设计了HandlerMapping实现的整体结构，子类只需要通过模板方法提供一些初始值或者具体的算法即可。

## HandlerExceptionResolver

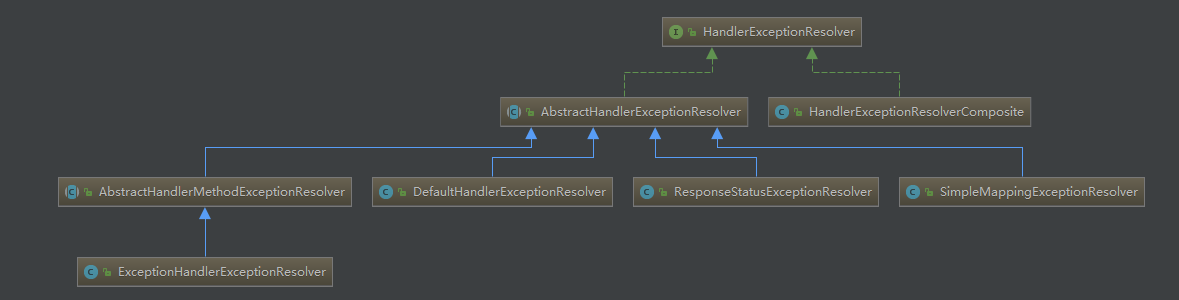
该类的主要作用是根据异常设置的ModelAndView，交个render方法进行渲染。render只负责将ModelAndView渲染成页面，具体ModelAndView是怎么来的render并不关心。

在分析DispacherServlet的doDispatcher方法时，我们分析得到HandlerExceptionResolver只是用来解析对请求做处理过程中产生的 异常，而渲染环节产生的异常并不归它管理。

HandlerExceptionResolver类只有一个方法：

ModelAndView resolveException(  
 HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex);

该方法只需要 从异常解析出ModelAndView就可以了。具体实现可以维护一个异常为key，View为value的Map，解析时直接从Map里获取View，如果在Map里没有相应的异常可以返回默认的View。



HandlerExceptionResolver的主要实现都继承了AbstractHandlerExceptionResolver，它有五个子类，其中AnnotationMethodHandlerExceptionResolver已经被弃用。

AbstractHandlerExceptionResolver和ExceptionHandlerExceptionResolver一起完成了使用@ExceptionHandler注释方法进行异常解析的功能。

DefaultHandlerExceptionResolver按不同类型的异常进行解析。

ResponseStatusExceptionResolver解析有@ResponseStatus注释类型的异常。

SimpleMappingExceptionResolver通过配置的异常类和view的对应关系来解析异常。

异常解析的过程主要包括两部分：给ModelAndView设置相应的内容，设置response的相关属性。在自定义的ExceptionHandler里还可以做更过的事情。

### AbstractHandlerExceptionResolver

该类定义了通用的异常解析流程，并使用了模板模式，子类只需要覆盖相应的方法即可。

private Set<?> mappedHandlers;  
private Class<?>[] mappedHandlerClasses;

这两个属性是在定义HandlerExceptionResolver的时候进行配置的，用于指定可以解析处理器抛出的哪些异常，也就是如果设置了这两个值中的一个，那么这个ExceptionResolver就只能解析所设置的处理器抛出的异常。mappedHandlers用于配置处理器的集合，mappedHandlerClassess用于配置处理器类的集合。

@Override  
public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response,  
 Object handler, Exception ex) {

//判断传入的hangler是否可以解析当前的ExceptionResolver

if (shouldApplyTo(request, handler)) {  
 if (this.logger.isDebugEnabled()) {  
 this.logger.debug("Resolving exception from handler [" + handler + "]: " + ex);  
 }

//

prepareResponse(ex, response);

//模板方法，子类使用它具体完成异常的解析工作

ModelAndView result = doResolveException(request, response, handler, ex);  
 if (result != null) {  
 logException(ex, request);  
 }  
 return result;  
 }  
 else {  
 return null;  
 }  
}

shouldApplyTo方法判断当前ExceptionResolver是否可以解析所传入处理器所抛出的异常，如果不可以则返回null，交给下一个ExceptionResolver解析

protected boolean shouldApplyTo(HttpServletRequest request, Object handler) {  
 if (handler != null) {  
 if (this.mappedHandlers != null && this.mappedHandlers.contains(handler)) {  
 return true;  
 }  
 if (this.mappedHandlerClasses != null) {  
 for (Class<?> handlerClass : this.mappedHandlerClasses) {  
 if (handlerClass.isInstance(handler)) {  
 return true;  
 }  
 }  
 }  
 }  
 return (this.mappedHandlers == null && this.mappedHandlerClasses == null);  
}

protected void prepareResponse(Exception ex, HttpServletResponse response) {  
 if (this.preventResponseCaching) {  
 preventCaching(response);  
 }  
}

根据preventResponseCaching标示判断是否给response设置禁用缓存的属性，默认值为false。

### AbstractHandlerMethodExceptionResolver

该类继承了AbstractHandlerExceptionResolver，并重写了shouldApplyTo方法，

@Override  
protected boolean shouldApplyTo(HttpServletRequest request, Object handler) {  
 if (handler == null) {  
 return super.shouldApplyTo(request, handler);  
 }  
 else if (handler instanceof HandlerMethod) {  
 HandlerMethod handlerMethod = (HandlerMethod) handler;  
 handler = handlerMethod.getBean();  
 return super.shouldApplyTo(request, handler);  
 }  
 else {  
 return false;  
 }  
}

该方法判断handler是否为HandlerMethod，如果是则将处理器设置为其所在的类，然后交给父类判断。

### ExceptionHandlerExceptionResolver

该类继承了AbstractHandlerMethodExceptionResolver，实现了doResolveHandlerMethodException方法，该方法首先根据handlerMethod和exception将其创建出来，然后设置了argumentResolvers和returnValueHanlders，接着调用其invokeAndHandler方法执行处理，最后将处理结果封装成ModelAndView返回。

@Override  
protected ModelAndView doResolveHandlerMethodException(HttpServletRequest request,  
 HttpServletResponse response, HandlerMethod handlerMethod, Exception exception) {  
//找到处理异常的方法

ServletInvocableHandlerMethod exceptionHandlerMethod = getExceptionHandlerMethod(handlerMethod, exception);  
 if (exceptionHandlerMethod == null) {  
 return null;  
 }  
 exceptionHandlerMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);  
 exceptionHandlerMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);  
 ServletWebRequest webRequest = new ServletWebRequest(request, response);  
 ModelAndViewContainer mavContainer = new ModelAndViewContainer();  
 try {  
 Throwable cause = exception.getCause();  
 if (cause != null) {  
 //执行ExceptionHandler方法解析异常

exceptionHandlerMethod.invokeAndHandle(webRequest, mavContainer, exception, cause, handlerMethod);  
 }  
 else {  
 // Otherwise, just the given exception as-is  
 exceptionHandlerMethod.invokeAndHandle(webRequest, mavContainer, exception, handlerMethod);  
 }  
 }  
 catch (Throwable invocationEx) {  
 return null;  
 }  
 if (mavContainer.isRequestHandled()) {  
 return new ModelAndView();  
 }  
 else {  
 ModelMap model = mavContainer.getModel();  
 HttpStatus status = mavContainer.getStatus();  
 ModelAndView mav = new ModelAndView(mavContainer.getViewName(), model, status);  
 mav.setViewName(mavContainer.getViewName());  
 if (!mavContainer.isViewReference()) {  
 mav.setView((View) mavContainer.getView());  
 }  
 if (model instanceof RedirectAttributes) {  
 Map<String, ?> flashAttributes = ((RedirectAttributes) model).getFlashAttributes();  
 request = webRequest.getNativeRequest(HttpServletRequest.class);  
 RequestContextUtils.*getOutputFlashMap*(request).putAll(flashAttributes);  
 }  
 return mav;  
 }  
}

### DefaultHandlerExceptionResolver

该类的解析过程是根据异常类型的不同，使用不同的方法进行处理。该类的doResolveException共对16中异常进行了处理，具体的解析方法也非常简单，主要是设置response的相关属性，

response的sendError方法用于设置错误类型，它有两个重载方法sendError(int)和sendError(int ,String)，int参数用于设置404、500等错误类型，String类型的参数用于设置附加的错误信息，可以在页面中获取到。sendError和sendStatus方法的区别时前者会返回web.xml中定义的相应错误页面，后者只是设置了status而不会返回相应错误页面。

### ResponseStatusExceptionResolver

该类用来解析注释了@ResponseStatus异常。

@Override  
protected ModelAndView doResolveException(HttpServletRequest request, HttpServletResponse response,  
 Object handler, Exception ex) {

//找到使用@ResponseStatus注解  
 ResponseStatus responseStatus = AnnotatedElementUtils.*findMergedAnnotation*(ex.getClass(), ResponseStatus.class);  
 if (responseStatus != null) {  
 try {  
 return resolveResponseStatus(responseStatus, request, response, handler, ex);  
 }  
 catch (Exception resolveEx) {  
 }  
 }  
 else if (ex.getCause() instanceof Exception) {  
 ex = (Exception) ex.getCause();  
 return doResolveException(request, response, handler, ex);  
 }  
 return null;  
}

### SimpleMappingExceptionResolver

该类需要提前配置异常类和view的对应关系然后才能使用。

@Override  
protected ModelAndView doResolveException(HttpServletRequest request, HttpServletResponse response,  
 Object handler, Exception ex) {  
 String viewName = determineViewName(ex, request);  
 if (viewName != null) {  
 Integer statusCode = determineStatusCode(request, viewName);  
 if (statusCode != null) {  
 applyStatusCodeIfPossible(request, response, statusCode);  
 }  
 return getModelAndView(viewName, ex, request);  
 }  
 else {  
 return null;  
 }  
}

首先调用determineViewName方法根据异常找到显示异常的逻辑视图，然后调用determineStatusCode方法判断逻辑视图是否对应的statusCode，如果有则调用applyStatusCodeIfPossible方法设置到response，最后调用getModelAndView将异常和解析出的viewName封装成ModelAndView并返回。

**配置选项：**1.exceptionMappings：用于配置异常类（字符串类型）和viewName的对应关系，异常类可以是异常的一部分，还可以是异常父类的一部分。

2.excludedExceptions：用于配置不处理的异常。

3.defaultErrorView：用于配置当无法从exceptionMappings中解析出视图时使用的默认视图。

4.statusCodes：用于配置解析出的viewName和statusCode对应关系。

5.defaultStatusCode：用于配置statusCode中没有配置相应的viewName时使用的默认statusCode。

6.exceptionAttribute：用于配置异常在Model中保存的参数名，默认为exception，如果为null，异常将不保存到Model中。

## MultipartResolver

MultipartResolver是用来处理上传请求的，处理方法是将普通的request封装成MultipartHttpServletRequest，MultipartHttpServletRequest可以调用getFile方法获取到File，如果上传多个文件，还可以调用getFileMap得到FileName🡪File结构的Map，这样就使得上传请求的处理变得非常简单。

是不是上传请求

boolean isMultipart(HttpServletRequest request);

将request包装成MultipartHttpServletRequest

MultipartHttpServletRequest resolveMultipart(HttpServletRequest request) throws MultipartException;

在上传请求处理完后清除上传过程中产生的临时资源。

void cleanupMultipart(MultipartHttpServletRequest request);

### StandardServletMultipartResolver

该类使用了Servelt3.0标准的上传方式，我们只需要调用request的getParts方法就可以获取所有上传的文件。如果想要单独获取某个文件可以使用request.getPart(filename)，获取Part后直接调用它的write(saveFileName)方法就可以将文件保存为isaveFileName的文件名的文件，也可以调用getInputStream获取InputStream。

如果我们希望这种方式需要在配置文件的Servlet时添加multipart-config属性。如：

<servlet>  
 <servlet-name>spring-dispatcher</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:config/applicationContext-\*.xml</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
 <multipart-config>  
 <location>/tmp</location>  
 <max-file-size>-1</max-file-size>  
 <max-request-size>-1</max-request-size>  
 <file-size-threshold>0</file-size-threshold>  
 </multipart-config>  
</servlet>

location：设置上传文件存放的目录，也就是调用Part的write(saveFileName)方法保存文件的根目录。如果saveFileName带了绝对路径，将以savefilename所带路径为准。

max-file-size：设置单个上传文件的最大值，默认值为-1，表示无限制。

max-request-size：设置一次上传的所有文件总和的最大值，默认值为-1，表示无限制。

file-size-threshold：设置不写入磁盘的最大数据量，默认值为0，表示所有上传的文件都会作为一个临时文件写入磁盘。

@Override  
public MultipartHttpServletRequest resolveMultipart(HttpServletRequest request) throws MultipartException {  
 return new StandardMultipartHttpServletRequest(request, this.resolveLazily);  
}

resolveMultipart方法之间将当前请求封装成了StandardMultipartHttpServletRequest并返回。

public StandardMultipartHttpServletRequest(HttpServletRequest request, boolean lazyParsing) throws MultipartException {  
 super(request);  
 if (!lazyParsing) {  
 parseRequest(request);  
 }  
}

private void parseRequest(HttpServletRequest request) {  
 try {  
 Collection<Part> parts = request.getParts();  
 this.multipartParameterNames = new LinkedHashSet<String>(parts.size());  
 MultiValueMap<String, MultipartFile> files = new LinkedMultiValueMap<String, MultipartFile>(parts.size());  
 for (Part part : parts) {  
 String disposition = part.getHeader(*CONTENT\_DISPOSITION*);  
 String filename = extractFilename(disposition);  
 if (filename == null) {  
 filename = extractFilenameWithCharset(disposition);  
 }  
 if (filename != null) {  
 files.add(part.getName(), new StandardMultipartFile(part, filename));  
 }  
 else {  
 this.multipartParameterNames.add(part.getName());  
 }  
 }  
 setMultipartFiles(files);  
 }  
 catch (Throwable ex) {  
 throw new MultipartException("Could not parse multipart servlet request", ex);  
 }  
}

### CommonsMultipartResolver

该类使用了commons-fileupload来完成文件的上传，所以在使用的时候要添加common-fileupload.jar

@Override  
public MultipartHttpServletRequest resolveMultipart(final HttpServletRequest request) throws MultipartException {  
 Assert.*notNull*(request, "Request must not be null");  
 if (this.resolveLazily) {  
 return new DefaultMultipartHttpServletRequest(request) {  
 @Override  
 protected void initializeMultipart() {  
 MultipartParsingResult parsingResult = parseRequest(request);  
 setMultipartFiles(parsingResult.getMultipartFiles());  
 setMultipartParameters(parsingResult.getMultipartParameters());  
 setMultipartParameterContentTypes(parsingResult.getMultipartParameterContentTypes());  
 }  
 };  
 }  
 else {  
 MultipartParsingResult parsingResult = parseRequest(request);  
 return new DefaultMultipartHttpServletRequest(request, parsingResult.getMultipartFiles(),  
 parsingResult.getMultipartParameters(), parsingResult.getMultipartParameterContentTypes());  
 }  
}

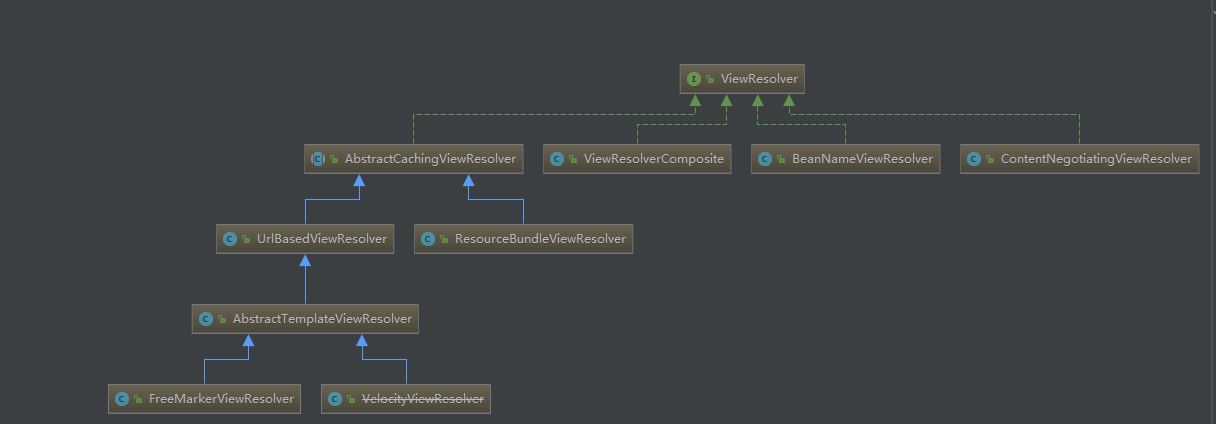
它根据不同的resolveLazily配置使用了两种不同的方法，不过都是将Request转换为DefaultMultipartHttpServlet

如果resolveLazily为true，则会将parsingRequest方法放在DefaultMultipartHttpServletRequest类重写的initializeMultipart方法中，InitializeMultipart方法只有在调用get方法（getMultipartFiles，getMultipartParameters或getMultiParameterContentTypes）时才会调用。

如果为false，则调用parsingRequest方法处理request，然后将处理结构出入DefaultMultipartHttpServletRequest。

protected MultipartParsingResult parseRequest(HttpServletRequest request) throws MultipartException {  
 String encoding = determineEncoding(request);  
 FileUpload fileUpload = prepareFileUpload(encoding);  
 try {  
 List<FileItem> fileItems = ((ServletFileUpload) fileUpload).parseRequest(request);  
 return parseFileItems(fileItems, encoding);  
 }  
 catch (FileUploadBase.SizeLimitExceededException ex) {  
 throw new MaxUploadSizeExceededException(fileUpload.getSizeMax(), ex);  
 }  
 catch (FileUploadBase.FileSizeLimitExceededException ex) {  
 throw new MaxUploadSizeExceededException(fileUpload.getFileSizeMax(), ex);  
 }  
 catch (FileUploadException ex) {  
 throw new MultipartException("Failed to parse multipart servlet request", ex);  
 }  
}

## ViewResolver



ViewResolver用来讲String类型的视图名和Locale解析为View类型的视图，ViewResolver接口只有一个方法。

View resolveViewName(String viewName, Locale locale) throws Exception;

View是用来渲染页面的：1.使用什么模板。2.用什么技术填入参数。

ViewResolver需要找到渲染所用的模板和所用的技术进行渲染，具体的渲染过程交给不同的视图自己完成。如：

InternalResourceViewResolver只针对jsp类型的视图，FreeMarkViewResolver针对FrameMarker，VelocityViewResolver只针对Velocity。

### BeanNameViewResolver

public class BeanNameViewResolver extends WebApplicationObjectSupport implements ViewResolver, Ordered

通过分析BeanNameViewResolver源码，在resolveViewName方法内，通过方法参数viewName，在容器中查找，如果没有找到这返回null。

### ViewResolverComposite

public class ViewResolverComposite implements ViewResolver, Ordered, InitializingBean,  
 ApplicationContextAware, ServletContextAware {

该类封装了多个ViewResolver的容器（List），解析视图时遍历封装的ViewResolver具体解析，该类除了遍历成员解析视图外还给成员进行了必要的初始化，其中包括对实现了ApplicationContextAware接口的ViewResolver设置了ApplicationContext，给事先ServletContextAware接口的ViewResolver设置了ServletContext以及对实现InitializingBean接口的ViewResolver调用afterPropertiesSet方法。

### ContentNegotiatingView

该解析器的作用是在别的解析器解析的结果上增加了对MediaType和后缀的支持，MediaType即媒体类型，有的地方叫做Content-Type。对视图解析并不是自己来完成的而是使用了所封装的视图解析器来进行的：遍历所封装的ViewResolver具体解析视图，可能会解析出多个视图，然后再使用request获取MeditType，也可能会有多个结果，最后对这两个结果进行匹配找出最优的视图。

private List<ViewResolver> viewResolvers;

viewResolvers的初始化有两种方法，第一种是手动设置，第二种如果没有设置则自动获取Spring容器中除了自己的所有ViewResolver并设置到viewResolvers。

@Override  
protected void initServletContext(ServletContext servletContext) {

//从整个Spring容器中获取ViewResolver类型  
 Collection<ViewResolver> matchingBeans =  
 BeanFactoryUtils.beansOfTypeIncludingAncestors(getApplicationContext(), ViewResolver.class).values();

//如果没有手动注册，则将容器中找到的ViewResolver设置给viewResolver  
 if (this.viewResolvers == null) {  
 this.viewResolvers = new ArrayList<ViewResolver>(matchingBeans.size());  
 for (ViewResolver viewResolver : matchingBeans) {  
 if (this != viewResolver) {  
 this.viewResolvers.add(viewResolver);  
 }  
 }  
 }  
 else {

//手动注册的，并且在容器中不存在，则进行初始化。  
 for (int i = 0; i < viewResolvers.size(); i++) {  
 if (matchingBeans.contains(viewResolvers.get(i))) {  
 continue;  
 }  
 String name = viewResolvers.get(i).getClass().getName() + i;  
 getApplicationContext().getAutowireCapableBeanFactory().initializeBean(viewResolvers.get(i), name);  
 }  
 }  
 if (this.viewResolvers.isEmpty()) {  
 }

//根据Order进行排序。  
 AnnotationAwareOrderComparator.sort(this.viewResolvers);  
 this.cnmFactoryBean.setServletContext(servletContext);  
}

视图解析：

@Override  
public View resolveViewName(String viewName, Locale locale) throws Exception {  
 RequestAttributes attrs = RequestContextHolder.getRequestAttributes();  
 Assert.state(attrs instanceof ServletRequestAttributes, "No current ServletRequestAttributes");  
 List<MediaType> requestedMediaTypes = getMediaTypes(((ServletRequestAttributes) attrs).getRequest());  
 if (requestedMediaTypes != null) {

//获取所有的候选视图，  
 List<View> candidateViews = getCandidateViews(viewName, locale, requestedMediaTypes);

//从多个候选视图中找出最优视图  
 View bestView = getBestView(candidateViews, requestedMediaTypes, attrs);  
 if (bestView != null) {  
 return bestView;  
 }  
 }  
 if (this.useNotAcceptableStatusCode) {  
 return NOT\_ACCEPTABLE\_VIEW;  
 }  
 else {  
 return null;  
 }  
}

private List<View> getCandidateViews(String viewName, Locale locale, List<MediaType> requestedMediaTypes)  
 throws Exception {  
 List<View> candidateViews = new ArrayList<View>();  
 for (ViewResolver viewResolver : this.viewResolvers) {  
 View view = viewResolver.resolveViewName(viewName, locale);  
 if (view != null) {  
 candidateViews.add(view);  
 }  
 for (MediaType requestedMediaType : requestedMediaTypes) {  
 List<String> extensions = this.contentNegotiationManager.resolveFileExtensions(requestedMediaType);  
 for (String extension : extensions) {  
 String viewNameWithExtension = viewName + '.' + extension;  
 view = viewResolver.resolveViewName(viewNameWithExtension, locale);  
 if (view != null) {  
 candidateViews.add(view);  
 }  
 }  
 }  
 }  
 if (!CollectionUtils.isEmpty(this.defaultViews)) {  
 candidateViews.addAll(this.defaultViews);  
 }  
 return candidateViews;  
}

遍历viewResolver进行视图解析，并将解析出的结果添加到候选视图，然后判断有没有默认视图，如果有则也会添加到候选视图。这里不仅直接使用逻辑视图进行解析，还使用了通过遍历requestedMediaTypes获取到所对应的后缀，然后添加到逻辑视图后面作为一个新视图名进行解析。

### AbstractCachingViewResolver

该类提供了统一的缓存功能，当视图解析过一次就被缓存起来，直到缓存被删除前视图的解析都会自动从缓存中获取。

该类的子类有：ResouceBundleViewResolve，XmlViewResolver和UrlBasedViewResolver。

private volatile int cacheLimit = *DEFAULT\_CACHE\_LIMIT*;

设置最大缓存数的，当设置为0时不启用缓存，isCache()方法就是判断它是否大于0，如果设置为一个大于0的数则它表示最多可以缓存视图的数量，如果往里面添加视图时超过这个数那么最前面缓存的值将会删除。

@Override  
public View resolveViewName(String viewName, Locale locale) throws Exception {

//判断是否开启缓存功能  
 if (!isCache()) {

//创建视图  
 return createView(viewName, locale);  
 }  
 else {  
 Object cacheKey = getCacheKey(viewName, locale);

//检查缓存中是否存在  
 View view = this.viewAccessCache.get(cacheKey);  
 if (view == null) {  
 synchronized (this.viewCreationCache) {  
 view = this.viewCreationCache.get(cacheKey);  
 if (view == null) {  
 // Ask the subclass to create the View object.  
 view = createView(viewName, locale);  
 if (view == null && this.cacheUnresolved) {  
 view = UNRESOLVED\_VIEW;  
 }  
 if (view != null) {  
 this.viewAccessCache.put(cacheKey, view);  
 this.viewCreationCache.put(cacheKey, view);  
 if (logger.isTraceEnabled()) {  
 logger.trace("Cached view [" + cacheKey + "]");  
 }  
 }  
 }  
 }  
 }  
 return (view != UNRESOLVED\_VIEW ? view : null);  
 }  
}

### UrlBasedViewResolver

UrlBasedViewResolver是所有直接将逻辑视图作为url查找模文件的ViewResolver的基类，因为它设置了统一的查找模板的规则，所以它的子类只需要确定渲染方式也就是视图的类型就可以了，它的每一个子类对应一种视图类型。

该类重写了getCacheKey、createView和loadView三个方法。

@Override  
protected Object getCacheKey(String viewName, Locale locale) {  
 return viewName;  
}

@Override  
protected View loadView(String viewName, Locale locale) throws Exception {

//创建View  
 AbstractUrlBasedView view = buildView(viewName);

//对创建的View初始化  
 View result = applyLifecycleMethods(viewName, view);

//检查view对应的模板是否存在，如果存在则将初始化的视图返回，否则返回null交给下一个ViewResolver处理  
 return (view.checkResource(locale) ? result : null);  
}

创建View，它的子类只是在这里创建出来的视图的基础上设置了一些属性。该方法为Abs

protected AbstractUrlBasedView buildView(String viewName) throws Exception {

//返回要创建的View  
 AbstractUrlBasedView view = (AbstractUrlBasedView) BeanUtils.instantiateClass(getViewClass());  
 view.setUrl(getPrefix() + viewName + getSuffix());  
 //设置contentType  
 String contentType = getContentType();  
 if (contentType != null) {  
 view.setContentType(contentType);  
 }  
  
 view.setRequestContextAttribute(getRequestContextAttribute());  
 view.setAttributesMap(getAttributesMap());  
 //它用来标示是否让view使用PathVariables，PathVariables就是处理器中@PathVariables  
 Boolean exposePathVariables = getExposePathVariables();  
 if (exposePathVariables != null) {  
 view.setExposePathVariables(exposePathVariables);  
 }

//它用来标示是否可以让view使用容器中注册的bean。  
 Boolean exposeContextBeansAsAttributes = getExposeContextBeansAsAttributes();  
 if (exposeContextBeansAsAttributes != null) {  
 view.setExposeContextBeansAsAttributes(exposeContextBeansAsAttributes);  
 }

//它用于配置view可以使用容器中的哪些bean  
 String[] exposedContextBeanNames = getExposedContextBeanNames();  
 if (exposedContextBeanNames != null) {  
 view.setExposedContextBeanNames(exposedContextBeanNames);  
 }  
  
 return view;  
}

UrlBasedViewResolver的子类主要做三件事：1.重写requiredViewClass方法修改了必须符合的视图类型的值。2.使用setViewClass设置了所用的视图类型。3.给创建出来的视图设置一些属性。

## 附加知识点

### Spring如何解析配制文件中的命名空间

我呢知道在配制文件中，我呢可以使用很多的命名空间来配置，而这些命名空间是如何解析呢？实际上，Spring中把解析标签的类都放到相应的META-INF目录下的spring.handlers文件中，然后从里面找。如：mvc命名空间解析设置在spring-webmvc-xxx.jar中的META-INF/spring.handler文件中

http\://www.springframework.org/schema/mvc=org.springframework.web.servlet.config.MvcNamespaceHandler