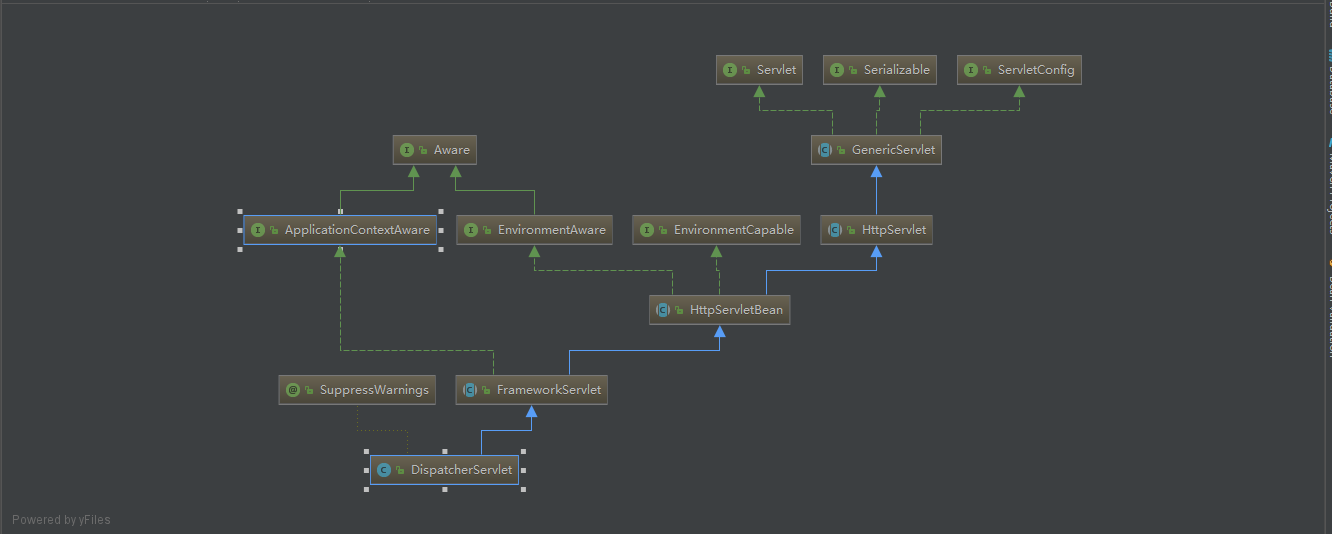
# 深入理解SpringMVC源码

## 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";

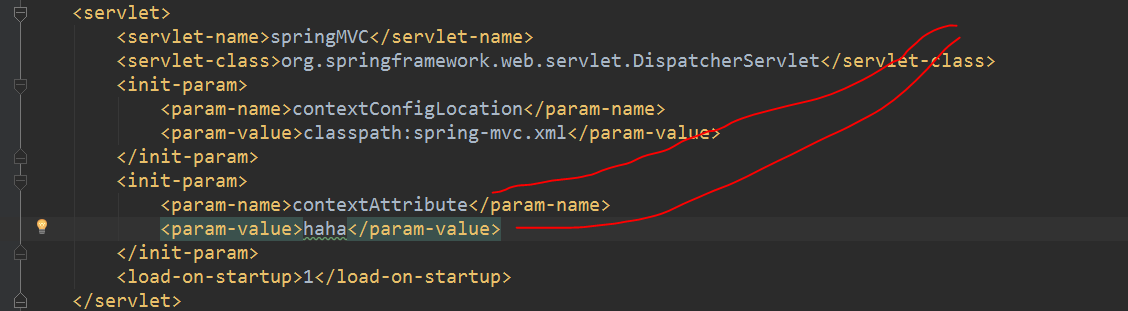
**设置webApplicationContext并根据情况调用onRefresh方法：**

**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

##### 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

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，

#### 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：

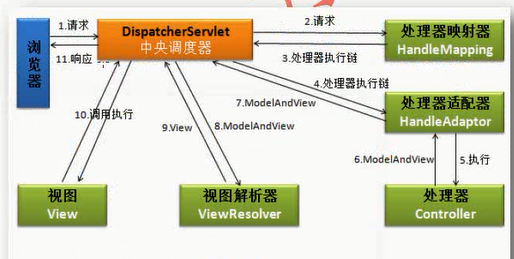
model.addAttribute("message","hello world") ;  
((FlashMap)((ServletRequestAttributes)(RequestContextHolder.*getRequestAttributes*()))  
 .getRequest().getAttribute(DispatcherServlet.*OUTPUT\_FLASH\_MAP\_ATTRIBUTE*)).put("name","hao");

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。

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Handler  
 mappedHandler = getHandler(processedRequest);  
 if (mappedHandler == null || mappedHandler.getHandler() == null) {  
 noHandlerFound(processedRequest, response);  
 return;  
 }  
  
 // 根据请求找到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找到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);

**//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);  
 }  
}

protected ModelAndView processHandlerException(HttpServletRequest request, HttpServletResponse response,  
 Object handler, Exception ex) throws Exception {  
 // Check registered HandlerExceptionResolvers...  
 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;  
 }  
 // We might still need view name translation for a plain error model...  
 if (!exMv.hasView()) {  
 exMv.setViewName(getDefaultViewName(request));  
 }  
 if (logger.isDebugEnabled()) {  
 logger.debug("Handler execution resulted in exception - forwarding to resolved error view: " + exMv, ex);  
 }  
 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，

## 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