# Struts2架构图

|  |
| --- |
| http://www.blogjava.net/images/blogjava_net/lzhidj/15.PNG |

注:这个是老版本（FilterDispatcher）的架构图,新版本未找到.

# StrutsPrepareAndExecuteFilter

Struts2的核心是一个Filter，Action可以脱离web容器，那么是什么让http请求和action关联在一起的，下面我们深入源码来分析下Struts2是如何工作的。

## **配置文件**

|  |
| --- |
| <filter>  <filter-name>struts2</filter-name>  <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter  </filter-class>  </filter>  <filter-mapping>  <filter-name>struts2</filter-name>  <url-pattern>\*.jsp</url-pattern>  </filter-mapping> |

### FilterDispatcher/StrutsPrepareAndExecuteFilter

FilterDispatcher是struts2.0.x到2.1.2版本的核心过滤器.!  
StrutsPrepareAndExecuteFilter是自2.1.3开始就替代了FilterDispatcher的.!

在老版本的struts2中我们自己定义过滤器的话, 是要放在strtus2的过滤器之前的, 如果放在struts2过滤器之后,你自己的过滤器对action的过滤作用就没有效果!

新版本的strutsPrepareAndExecuteFilter则没有这样的问题

## init

 init是Filter第一个运行的方法，我们看下struts2的核心Filter在调用init方法初始化时做哪些工作：

|  |
| --- |
| **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  //工具类，帮助做初始化操作  InitOperations init = **new** InitOperations();  **try** {  //封装filterConfig，其中有个主要方法getInitParameterNames将参数名字以String格式存储在List中  FilterHostConfig config = **new** FilterHostConfig(filterConfig);  // 初始化 logFactory  init.initLogging(config);  //创建dispatcher ，并初始化  Dispatcher dispatcher = init.initDispatcher(config);  init.initStaticContentLoader(config, dispatcher);  //初始化类属性：prepare 、execute  prepare = **new** PrepareOperations(filterConfig.getServletContext(), dispatcher);  execute = **new** ExecuteOperations(filterConfig.getServletContext(), dispatcher);  } **finally** {  //回调空的postInit方法  init.cleanup();  }  } |

### FilterHostConfig

|  |
| --- |
| **public** **class** FilterHostConfig **implements** HostConfig {  **private** FilterConfig config;  //根据init-param配置的param-name获取param-value的值  **public** FilterHostConfig(FilterConfig config) {  **this**.config = config;  }  //返回初始化参数名的List  **public** String getInitParameter(String key) {  **return** config.getInitParameter(key);  }  **public** Iterator<String> getInitParameterNames() {  **return** MakeIterator.convert(config.getInitParameterNames());  }  **public** ServletContext getServletContext() {  **return** config.getServletContext();  }  } |
| **public** **interface** HostConfig {  /\*\*  \* **@param** key The parameter key  \* **@return** The parameter value  \*/  String getInitParameter(String key);  /\*\*  \* **@return** A list of parameter names  \*/  Iterator<String> getInitParameterNames();  /\*\*  \* **@return** The servlet context  \*/  ServletContext getServletContext();  } |

### InitOperations# initLogging

|  |
| --- |
| **public** **void** initLogging(HostConfig filterConfig) {  String factoryName = filterConfig.getInitParameter("loggerFactory");  **if** (factoryName != **null**) {  **try** {  Class cls = ClassLoaderUtils.loadClass(factoryName, **this**.getClass());  LoggerFactory fac = (LoggerFactory) cls.newInstance();  LoggerFactory.setLoggerFactory(fac);  } **catch** (InstantiationException e) {  System.err.println("Unable to instantiate logger factory: " + factoryName + ", using default");  e.printStackTrace();  } **catch** (IllegalAccessException e) {  System.err.println("Unable to access logger factory: " + factoryName + ", using default");  e.printStackTrace();  } **catch** (ClassNotFoundException e) {  System.err.println("Unable to locate logger factory class: " + factoryName + ", using default");  e.printStackTrace();  }  }  } |

### InitOperations# initDispatcher

|  |
| --- |
| InitOperations# initDispatcher  **public** Dispatcher initDispatcher(HostConfig filterConfig) {  Dispatcher dispatcher = createDispatcher(filterConfig);  dispatcher.init();  **return** dispatcher;  } |
| InitOperations# createDispatcher  **private** Dispatcher createDispatcher(HostConfig filterConfig) {  Map<String, String> params = **new** HashMap<String, String>();  **for** (Iterator e = filterConfig.getInitParameterNames(); e.hasNext();) {  String name = (String) e.next();  String value = filterConfig.getInitParameter(name);  params.put(name, value);  }  **return** **new** Dispatcher(filterConfig.getServletContext(), params);  } |
| InitOperations# initStaticContentLoader  **public** StaticContentLoader initStaticContentLoader(HostConfig filterConfig, Dispatcher dispatcher) {  StaticContentLoader loader = dispatcher.getContainer().getInstance(StaticContentLoader.**class**);  loader.setHostConfig(filterConfig);  **return** loader;  } |
| Dispatcher#init  **public** **void** init() {  **if** (configurationManager == **null**) {  configurationManager = **new** ConfigurationManager(BeanSelectionProvider.*DEFAULT\_BEAN\_NAME*);  }  //加载org/apache/struts2/default.properties  init\_DefaultProperties(); // [1]  //加载struts-default.xml,struts-plugin.xml,struts.xml  init\_TraditionalXmlConfigurations(); // [2]  //  init\_LegacyStrutsProperties(); // [3]  //用户自己实现的ConfigurationProviders类  init\_CustomConfigurationProviders(); // [5]  //Filter的初始化参数  init\_FilterInitParameters() ; // [6]  init\_AliasStandardObjects() ; // [7]  Container container = init\_PreloadConfiguration();  container.inject(**this**);  init\_CheckConfigurationReloading(container);  init\_CheckWebLogicWorkaround(container);  **if** (!*dispatcherListeners*.isEmpty()) {  **for** (DispatcherListener l : *dispatcherListeners*) {  l.dispatcherInitialized(**this**);  }  }  } |
|  |

### PrepareOperations

|  |
| --- |
| **public** **class** PrepareOperations {  **private** ServletContext servletContext;  **private** Dispatcher dispatcher;  **private** **static** **final** String STRUTS\_ACTION\_MAPPING\_KEY = "struts.actionMapping";  **public** **static** **final** String CLEANUP\_RECURSION\_COUNTER = "\_\_cleanup\_recursion\_counter";  **private** Logger log = LoggerFactory.getLogger(PrepareOperations.**class**);  **public** PrepareOperations(ServletContext servletContext, Dispatcher dispatcher) {  **this**.dispatcher = dispatcher;  **this**.servletContext = servletContext;  }  } |

### ExecuteOperations

## doFileter

|  |
| --- |
| **public** **void** doFilter(ServletRequest req, ServletResponse res, FilterChain chain) **throws** IOException, ServletException {  HttpServletRequest request = (HttpServletRequest) req;  HttpServletResponse response = (HttpServletResponse) res;  **try** {    //为当前线程创建一个ActionContext（重点）  prepare.createActionContext(request, response);  //将当前线程  prepare.assignDispatcherToThread();  //设置编码和国际化  prepare.setEncodingAndLocale(request, response);  //包装请求  request = prepare.wrapRequest(request);    //获取ActionMapping  ActionMapping mapping = prepare.findActionMapping(request, response);  **if** (mapping == **null**) {  **boolean** handled = execute.executeStaticResourceRequest(request, response);  **if** (!handled) {  chain.doFilter(request, response);  }  } **else** {  //执行  execute.executeAction(request, response, mapping);  }  } **finally** {  prepare.cleanupRequest(request);  }  } |

### prepare.createActionContext

|  |
| --- |
| PrepareOperations#createActionContext  **public** ActionContext createActionContext(HttpServletRequest request, HttpServletResponse response) {  ActionContext ctx;  Integer counter = 1;  Integer oldCounter = (Integer) request.getAttribute(CLEANUP\_RECURSION\_COUNTER);  **if** (oldCounter != **null**) {  counter = oldCounter + 1;  }    ActionContext oldContext = ActionContext.getContext();  **if** (oldContext != **null**) {  // detected existing context, so we are probably in a forward  ctx = **new** ActionContext(**new** HashMap<String, Object>(oldContext.getContextMap()));  } **else** {  ValueStack stack = dispatcher.getContainer().getInstance(ValueStackFactory.**class**).createValueStack();  stack.getContext().putAll(dispatcher.createContextMap(request, response, **null**, servletContext));  ctx = **new** ActionContext(stack.getContext());  }  request.setAttribute(CLEANUP\_RECURSION\_COUNTER, counter);  ActionContext.setContext(ctx);  **return** ctx;  } |
| ActionContext#getContext  **public** **static** ActionContext getContext() {  **return** (ActionContext) actionContext.get();  } |
| **public** **static** **void** setContext(ActionContext context) {  actionContext.set(context);  } |
| ActionContext# actionContext  **static** ThreadLocal actionContext = **new** ThreadLocal(); |

### prepare.assignDispatcherToThread

|  |
| --- |
| PrepareOperations# assignDispatcherToThread  public void assignDispatcherToThread() {  Dispatcher.setInstance(dispatcher);  } |
| Dispatcher# setInstance  **public** **static** **void** setInstance(Dispatcher instance) {  Dispatcher.*instance*.set(instance);  } |
| Dispatcher# instance  /\*\*  \* Provide a thread local instance.  \*/  **private** **static** ThreadLocal<Dispatcher> *instance* = **new** ThreadLocal<Dispatcher>(); |

### prepare.setEncodingAndLocale

|  |
| --- |
| PrepareOperations# setEncodingAndLocale  **public** **void** setEncodingAndLocale(HttpServletRequest request, HttpServletResponse response) {  dispatcher.prepare(request, response);  } |
| Dispatcher#prepare  **public** **void** prepare(HttpServletRequest request, HttpServletResponse response) {  String encoding = **null**;  **if** (defaultEncoding != **null**) {  encoding = defaultEncoding;  }  Locale locale = **null**;  **if** (defaultLocale != **null**) {  locale = LocalizedTextUtil.*localeFromString*(defaultLocale, request.getLocale());  }  **if** (encoding != **null**) {  **try** {  request.setCharacterEncoding(encoding);  } **catch** (Exception e) {  *LOG*.error("Error setting character encoding to '" + encoding + "' - ignoring.", e);  }  }  **if** (locale != **null**) {  response.setLocale(locale);  }  **if** (paramsWorkaroundEnabled) {  request.getParameter("foo"); // simply read any parameter (existing or not) to "prime" the request  }  } |

### prepare.wrapRequest(request)

|  |
| --- |
| PrepareOperations# wrapRequest  **public** HttpServletRequest wrapRequest(HttpServletRequest oldRequest) **throws** ServletException {  HttpServletRequest request = oldRequest;  **try** {  // Wrap request first, just in case it is multipart/form-data  // parameters might not be accessible through before encoding (ww-1278)  request = dispatcher.wrapRequest(request, servletContext);  } **catch** (IOException e) {  String message = "Could not wrap servlet request with MultipartRequestWrapper!";  **throw** **new** ServletException(message, e);  }  **return** request;  } |
| Dispatcher#wrapRequest  **public** HttpServletRequest wrapRequest(HttpServletRequest request, ServletContext servletContext) **throws** IOException {  // don't wrap more than once  **if** (request **instanceof** StrutsRequestWrapper) {  **return** request;  }  String content\_type = request.getContentType();  **if** (content\_type != **null** && content\_type.indexOf("multipart/form-data") != -1) {  MultiPartRequest multi = getContainer().getInstance(MultiPartRequest.**class**);  request = **new** MultiPartRequestWrapper(multi, request, getSaveDir(servletContext));  } **else** {  request = **new** StrutsRequestWrapper(request);  }  **return** request;  } |

### prepare.findActionMapping

|  |
| --- |
| PrepareOperations# findActionMapping  **public** ActionMapping findActionMapping(HttpServletRequest request, HttpServletResponse response) {  ActionMapping mapping = (ActionMapping) request.getAttribute(STRUTS\_ACTION\_MAPPING\_KEY);  **if** (mapping == **null**) {  **try** {  mapping = dispatcher.getContainer().getInstance(ActionMapper.**class**).getMapping(request, dispatcher.getConfigurationManager());  **if** (mapping != **null**) {  request.setAttribute(STRUTS\_ACTION\_MAPPING\_KEY, mapping);  }  } **catch** (Exception ex) {  dispatcher.sendError(request, response, servletContext, HttpServletResponse.SC\_INTERNAL\_SERVER\_ERROR, ex);  }  }  **return** mapping;  } |
| Dispatcher#getContainer  **public** Container getContainer() {  ConfigurationManager mgr = getConfigurationManager();  **if** (mgr == **null**) {  **throw** **new** IllegalStateException("The configuration manager shouldn't be null");  } **else** {  Configuration config = mgr.getConfiguration();  **if** (config == **null**) {  **throw** **new** IllegalStateException("Unable to load configuration");  } **else** {  **return** config.getContainer();  }  }  }  } |
| Dispatcher# getConfigurationManager  **public** ConfigurationManager getConfigurationManager() {  **return** configurationManager;  }  \* Store ConfigurationManager instance, set on init.  \*/  Dispatcher# ConfigurationManager  **private** ConfigurationManager configurationManager; |

### execute.executeStaticResourceRequest

|  |
| --- |
| ExecuteOperations# executeStaticResourceRequest    **public** **boolean** executeStaticResourceRequest(HttpServletRequest request, HttpServletResponse response) **throws** IOException, ServletException {  // there is no action in this request, should we look for a static resource?  String resourcePath = RequestUtils.getServletPath(request);  **if** ("".equals(resourcePath) && **null** != request.getPathInfo()) {  resourcePath = request.getPathInfo();  }  StaticContentLoader staticResourceLoader = dispatcher.getContainer().getInstance(StaticContentLoader.**class**);  **if** (staticResourceLoader.canHandle(resourcePath)) {  staticResourceLoader.findStaticResource(resourcePath, request, response);  // The framework did its job here  **return** **true**;  } **else** {  // this is a normal request, let it pass through  **return** **false**;  }  } |
|  |

### execute.executeAction

|  |
| --- |
| ExecuteOperations# executeAction  **public** **void** executeAction(HttpServletRequest request, HttpServletResponse response, ActionMapping mapping) **throws** ServletException {  dispatcher.serviceAction(request, response, servletContext, mapping);  } |
| Dispatcher#serviceAction  **public** **void** serviceAction(HttpServletRequest request, HttpServletResponse response, ServletContext context,  ActionMapping mapping) **throws** ServletException {  //封装执行的上下文环境，主要讲相关信息存储入map  Map<String, Object> extraContext = createContextMap(request, response, mapping, context);  // If there was a previous value stack, then create a new copy and pass it in to be used by the new Action  ValueStack stack = (ValueStack) request.getAttribute(ServletActionContext.*STRUTS\_VALUESTACK\_KEY*);  **boolean** nullStack = stack == **null**;  **if** (nullStack) {  ActionContext ctx = ActionContext.*getContext*();  **if** (ctx != **null**) {  stack = ctx.getValueStack();  }  }  **if** (stack != **null**) {  extraContext.put(ActionContext.*VALUE\_STACK*, valueStackFactory.createValueStack(stack));  }  String timerKey = "Handling request from Dispatcher";  **try** {  UtilTimerStack.*push*(timerKey);  //获取命名空间  String namespace = mapping.getNamespace();  //获取action配置的name属性  String name = mapping.getName();  //获取action配置的method属性  String method = mapping.getMethod();  Configuration config = configurationManager.getConfiguration();  //根据执行上下文参数，命名空间，名称等创建用户自定义Action的代理对象  ActionProxy proxy = config.getContainer().getInstance(ActionProxyFactory.**class**).createActionProxy(  namespace, name, method, extraContext, **true**, **false**);  request.setAttribute(ServletActionContext.*STRUTS\_VALUESTACK\_KEY*, proxy.getInvocation().getStack());  // if the ActionMapping says to go straight to a result, do it!  //执行execute方法，并转向结果  **if** (mapping.getResult() != **null**) {    Result result = mapping.getResult();  result.execute(proxy.getInvocation());  } **else** {  proxy.execute();  }  // If there was a previous value stack then set it back onto the request  **if** (!nullStack) {  request.setAttribute(ServletActionContext.*STRUTS\_VALUESTACK\_KEY*, stack);  }  } **catch** (ConfigurationException e) {  // WW-2874 Only log error if in devMode  **if**(devMode) {  *LOG*.error("Could not find action or result", e);  }  **else** {  *LOG*.warn("Could not find action or result", e);  }  sendError(request, response, context, HttpServletResponse.*SC\_NOT\_FOUND*, e);  } **catch** (Exception e) {  sendError(request, response, context, HttpServletResponse.*SC\_INTERNAL\_SERVER\_ERROR*, e);  } **finally** {  UtilTimerStack.*pop*(timerKey);  }  } |

### prepare.cleanupRequest

|  |
| --- |
| **public** **void** cleanupRequest(HttpServletRequest request) {  Integer counterVal = (Integer) request.getAttribute(CLEANUP\_RECURSION\_COUNTER);  **if** (counterVal != **null**) {  counterVal -= 1;  request.setAttribute(CLEANUP\_RECURSION\_COUNTER, counterVal);  **if** (counterVal > 0 ) {  **if** (log.isDebugEnabled()) {  log.debug("skipping cleanup counter="+counterVal);  }  **return**;  }  }  // always clean up the thread request, even if an action hasn't been executed  ActionContext.setContext(**null**);  Dispatcher.setInstance(**null**);  } |

## destroy

|  |
| --- |
| **public** **void** destroy() {  prepare.cleanupDispatcher();  } |

### prepare#cleanupDispatcher

|  |
| --- |
| PrepareOperations#cleanupDispatcher  **public** **void** cleanupDispatcher() {  **if** (dispatcher == **null**) {  **throw** **new** StrutsException("something is seriously wrong, Dispatcher is not initialized (null) ");  } **else** {  **try** {  dispatcher.cleanup();  } **finally** {  ActionContext.setContext(**null**);  }  }  }  } |
| Dispatcher#cleanup  **public** **void** cleanup() {  // clean up ObjectFactory  ObjectFactory objectFactory = getContainer().getInstance(ObjectFactory.**class**);  **if** (objectFactory == **null**) {  *LOG*.warn("Object Factory is null, something is seriously wrong, no clean up will be performed");  }  **if** (objectFactory **instanceof** ObjectFactoryDestroyable) {  **try** {  ((ObjectFactoryDestroyable)objectFactory).destroy();  }  **catch**(Exception e) {  // catch any exception that may occured during destroy() and log it  *LOG*.error("exception occurred while destroying ObjectFactory ["+objectFactory+"]", e);  }  }  // clean up Dispatcher itself for this thread  *instance*.set(**null**);  // clean up DispatcherListeners  **if** (!*dispatcherListeners*.isEmpty()) {  **for** (DispatcherListener l : *dispatcherListeners*) {  l.dispatcherDestroyed(**this**);  }  }  // clean up all interceptors by calling their destroy() method  Set<Interceptor> interceptors = **new** HashSet<Interceptor>();  Collection<PackageConfig> packageConfigs = configurationManager.getConfiguration().getPackageConfigs().values();  **for** (PackageConfig packageConfig : packageConfigs) {  **for** (Object config : packageConfig.getAllInterceptorConfigs().values()) {  **if** (config **instanceof** InterceptorStackConfig) {  **for** (InterceptorMapping interceptorMapping : ((InterceptorStackConfig) config).getInterceptors()) {  interceptors.add(interceptorMapping.getInterceptor());  }  }  }  }  **for** (Interceptor interceptor : interceptors) {  interceptor.destroy();  }  // clean up configuration  configurationManager.destroyConfiguration();  configurationManager = **null**;  } |
| ActionContext#getContext  **public** **static** ActionContext getContext() {  **return** (ActionContext) actionContext.get();  } |
| **public** **static** **void** setContext(ActionContext context) {  actionContext.set(context);  } |
| ActionContext# actionContext  **static** ThreadLocal actionContext = **new** ThreadLocal(); |

# Dispatcher

## Dispatcher#init

|  |
| --- |
| StrutsPrepareAndExecuteFilter#init 🡪  Dispatcher dispatcher = init.initDispatcher(config);  InitOperations# initDispatcher-🡪  **public** Dispatcher initDispatcher(HostConfig filterConfig) {  Dispatcher dispatcher = createDispatcher(filterConfig);  dispatcher.init();  **return** dispatcher;  }  InitOperations# createDispatcher  **private** Dispatcher createDispatcher(HostConfig filterConfig) {  Map<String, String> params = **new** HashMap<String, String>();  **for** (Iterator e = filterConfig.getInitParameterNames(); e.hasNext();) {  String name = (String) e.next();  String value = filterConfig.getInitParameter(name);  params.put(name, value);  }  **return** **new** Dispatcher(filterConfig.getServletContext(), params);  } |

|  |
| --- |
| **public** **void** init() {  **if** (configurationManager == **null**) {  configurationManager = **new** ConfigurationManager(BeanSelectionProvider.*DEFAULT\_BEAN\_NAME*);  }  init\_DefaultProperties(); // [1]  init\_TraditionalXmlConfigurations(); // [2]  init\_LegacyStrutsProperties(); // [3]  init\_CustomConfigurationProviders(); // [5]  init\_FilterInitParameters() ; // [6]  init\_AliasStandardObjects() ; // [7]  Container container = init\_PreloadConfiguration();  container.inject(**this**);  init\_CheckConfigurationReloading(container);  init\_CheckWebLogicWorkaround(container);  **if** (!*dispatcherListeners*.isEmpty()) {  **for** (DispatcherListener l : *dispatcherListeners*) {  l.dispatcherInitialized(**this**);  }  }  } |

### Dispatcher#init\_DefaultProperties

|  |
| --- |
| **private** **void** init\_DefaultProperties() {  configurationManager.addConfigurationProvider(**new** DefaultPropertiesProvider());  } |

### Dispatcher#init\_LegacyStrutsProperties

|  |
| --- |
| **private** **void** init\_LegacyStrutsProperties() {  configurationManager.addConfigurationProvider(**new** LegacyPropertiesConfigurationProvider());  } |

### Dispatcher#init\_TraditionalXmlConfigurations

|  |
| --- |
| **private** **void** init\_TraditionalXmlConfigurations() {  String configPaths = initParams.get("config");  **if** (configPaths == **null**) {  configPaths = *DEFAULT\_CONFIGURATION\_PATHS*;  }  String[] files = configPaths.split("\\s\*[,]\\s\*");  **for** (String file : files) {  **if** (file.endsWith(".xml")) {  **if** ("xwork.xml".equals(file)) {  configurationManager.addConfigurationProvider(**new** XmlConfigurationProvider(file, **false**));  } **else** {  configurationManager.addConfigurationProvider(**new** StrutsXmlConfigurationProvider(file, **false**, servletContext));  }  } **else** {  **throw** **new** IllegalArgumentException("Invalid configuration file name");  }  }  } |

### Dispatcher#init\_CustomConfigurationProviders

|  |
| --- |
| **private** **void** init\_CustomConfigurationProviders() {  String configProvs = initParams.get("configProviders");  **if** (configProvs != **null**) {  String[] classes = configProvs.split("\\s\*[,]\\s\*");  **for** (String cname : classes) {  **try** {  Class cls = ClassLoaderUtils.loadClass(cname, **this**.getClass());  ConfigurationProvider prov = (ConfigurationProvider)cls.newInstance();  configurationManager.addConfigurationProvider(prov);  } **catch** (InstantiationException e) {  **throw** **new** ConfigurationException("Unable to instantiate provider: "+cname, e);  } **catch** (IllegalAccessException e) {  **throw** **new** ConfigurationException("Unable to access provider: "+cname, e);  } **catch** (ClassNotFoundException e) {  **throw** **new** ConfigurationException("Unable to locate provider class: "+cname, e);  }  }  }  } |

### Dispatcher#FilterInitParameters

|  |
| --- |
| **private** **void** init\_FilterInitParameters() {  configurationManager.addConfigurationProvider(**new** ConfigurationProvider() {  **public** **void** destroy() {}  **public** **void** init(Configuration configuration) **throws** ConfigurationException {}  **public** **void** loadPackages() **throws** ConfigurationException {}  **public** **boolean** needsReload() { **return** **false**; }  **public** **void** register(ContainerBuilder builder, LocatableProperties props) **throws** ConfigurationException {  props.putAll(initParams);  }  });  } |

### Dispatcher#init\_AliasStandardObjects

|  |
| --- |
| **private** **void** init\_AliasStandardObjects() {  configurationManager.addConfigurationProvider(**new** BeanSelectionProvider());  } |

### Dispatcher#init\_PreloadConfiguration

|  |
| --- |
| **private** Container init\_PreloadConfiguration() {  Configuration config = configurationManager.getConfiguration();  Container container = config.getContainer();  **boolean** reloadi18n = Boolean.*valueOf*(container.getInstance(String.**class**, StrutsConstants.*STRUTS\_I18N\_RELOAD*));  LocalizedTextUtil.*setReloadBundles*(reloadi18n);  **return** container;  } |

### Dispatcher#init\_ CheckConfigurationReloading

|  |
| --- |
| **private** **void** init\_CheckConfigurationReloading(Container container) {  FileManager.setReloadingConfigs("true".equals(container.getInstance(String.**class**,  StrutsConstants.*STRUTS\_CONFIGURATION\_XML\_RELOAD*)));  } |

### Dispatcher#init\_CheckWebLogicWorkaround

|  |
| --- |
| **private** **void** init\_CheckWebLogicWorkaround(Container container) {  // test whether param-access workaround needs to be enabled  **if** (servletContext != **null** && servletContext.getServerInfo() != **null**  && servletContext.getServerInfo().indexOf("WebLogic") >= 0) {  *LOG*.info("WebLogic server detected. Enabling Struts parameter access work-around.");  paramsWorkaroundEnabled = **true**;  } **else** {  paramsWorkaroundEnabled = "true".equals(container.getInstance(String.**class**,  StrutsConstants.*STRUTS\_DISPATCHER\_PARAMETERSWORKAROUND*));  }  } |

## Dispatcher#createContextMap

|  |
| --- |
| StrutsPrepareAndExcuteFilter#doFilter🡪  prepare.createActionContext(request, response);  PrepareOperations#createActionContext🡪  **public** ActionContext createActionContext(HttpServletRequest request, HttpServletResponse response) {  ActionContext ctx;  Integer counter = 1;  Integer oldCounter = (Integer) request.getAttribute(CLEANUP\_RECURSION\_COUNTER);  **if** (oldCounter != **null**) {  counter = oldCounter + 1;  }    ActionContext oldContext = ActionContext.getContext();  **if** (oldContext != **null**) {  // detected existing context, so we are probably in a forward  ctx = **new** ActionContext(**new** HashMap<String, Object>(oldContext.getContextMap()));  } **else** {  ValueStack stack = dispatcher.getContainer().getInstance(ValueStackFactory.**class**).createValueStack();  stack.getContext().putAll(dispatcher.createContextMap(request, response, **null**, servletContext));  ctx = **new** ActionContext(stack.getContext());  }  request.setAttribute(CLEANUP\_RECURSION\_COUNTER, counter);  ActionContext.setContext(ctx);  **return** ctx;  } |

|  |
| --- |
| **public** Map<String,Object> createContextMap(HttpServletRequest request, HttpServletResponse response,  ActionMapping mapping, ServletContext context) {  // request map wrapping the http request objects  Map requestMap = **new** RequestMap(request);  // parameters map wrapping the http parameters. ActionMapping parameters are now handled and applied separately  Map params = **new** HashMap(request.getParameterMap());  // session map wrapping the http session  Map session = **new** SessionMap(request);  // application map wrapping the ServletContext  Map application = **new** ApplicationMap(context);  Map<String,Object> extraContext = createContextMap(requestMap, params, session, application, request, response, context);  **if** (mapping != **null**) {  extraContext.put(ServletActionContext.*ACTION\_MAPPING*, mapping);  }  **return** extraContext;  }  **public** HashMap<String,Object> createContextMap(Map requestMap,  Map parameterMap,  Map sessionMap,  Map applicationMap,  HttpServletRequest request,  HttpServletResponse response,  ServletContext servletContext) {  HashMap<String,Object> extraContext = **new** HashMap<String,Object>();  extraContext.put(ActionContext.*PARAMETERS*, **new** HashMap(parameterMap));  extraContext.put(ActionContext.*SESSION*, sessionMap);  extraContext.put(ActionContext.*APPLICATION*, applicationMap);  Locale locale;  **if** (defaultLocale != **null**) {  locale = LocalizedTextUtil.*localeFromString*(defaultLocale, request.getLocale());  } **else** {  locale = request.getLocale();  }  extraContext.put(ActionContext.*LOCALE*, locale);  //extraContext.put(ActionContext.DEV\_MODE, Boolean.valueOf(devMode));  extraContext.put(StrutsStatics.*HTTP\_REQUEST*, request);  extraContext.put(StrutsStatics.*HTTP\_RESPONSE*, response);  extraContext.put(StrutsStatics.*SERVLET\_CONTEXT*, servletContext);  // helpers to get access to request/session/application scope  extraContext.put("request", requestMap);  extraContext.put("session", sessionMap);  extraContext.put("application", applicationMap);  extraContext.put("parameters", parameterMap);  AttributeMap attrMap = **new** AttributeMap(extraContext);  extraContext.put("attr", attrMap);  **return** extraContext;  } |

## Dispatcher# wrapRequest

|  |
| --- |
| StrutsPrepareAndExcuteFilter#doFilter🡪  request = prepare.wrapRequest(request);  PrepareOperations# wrapRequest  **public** HttpServletRequest wrapRequest(HttpServletRequest oldRequest) **throws** ServletException {  HttpServletRequest request = oldRequest;  **try** {  // Wrap request first, just in case it is multipart/form-data  // parameters might not be accessible through before encoding (ww-1278)  request = dispatcher.wrapRequest(request, servletContext);  } **catch** (IOException e) {  String message = "Could not wrap servlet request with MultipartRequestWrapper!";  **throw** **new** ServletException(message, e);  }  **return** request;  } |
| Dispatcher#wrapRequest  **public** HttpServletRequest wrapRequest(HttpServletRequest request, ServletContext servletContext) **throws** IOException {  // don't wrap more than once  **if** (request **instanceof** StrutsRequestWrapper) {  **return** request;  }  String content\_type = request.getContentType();  **if** (content\_type != **null** && content\_type.indexOf("multipart/form-data") != -1) {  MultiPartRequest multi = getContainer().getInstance(MultiPartRequest.**class**);  request = **new** MultiPartRequestWrapper(multi, request, getSaveDir(servletContext));  } **else** {  request = **new** StrutsRequestWrapper(request);  }  **return** request;  } |

## Dispatcher#ServiceAction

|  |
| --- |
| StrutsPrepareAndExcuteFilter#doFilter部分代码片段->  execute.executeAction(request, response, mapping);    ExecuteOperations# executeAction  **public** **void** executeAction(HttpServletRequest request, HttpServletResponse response, ActionMapping mapping) **throws** ServletException {  dispatcher.serviceAction(request, response, servletContext, mapping);  }  Dispatcher#ServiceAction 部分代码片段🡪  ActionProxy proxy = config.getContainer().getInstance(ActionProxyFactory.**class**).createActionProxy(  namespace, name, method, extraContext, **true**, **false**); |

|  |
| --- |
| **public** **void** serviceAction(HttpServletRequest request, HttpServletResponse response, ServletContext context,  ActionMapping mapping) **throws** ServletException {  //封装执行的上下文环境，主要讲相关信息存储入map  Map<String, Object> extraContext = createContextMap(request, response, mapping, context);  // If there was a previous value stack, then create a new copy and pass it in to be used by the new Action  ValueStack stack = (ValueStack) request.getAttribute(ServletActionContext.*STRUTS\_VALUESTACK\_KEY*);  **boolean** nullStack = stack == **null**;  **if** (nullStack) {  ActionContext ctx = ActionContext.*getContext*();  **if** (ctx != **null**) {  stack = ctx.getValueStack();  }  }  **if** (stack != **null**) {  extraContext.put(ActionContext.*VALUE\_STACK*, valueStackFactory.createValueStack(stack));  }  String timerKey = "Handling request from Dispatcher";  **try** {  UtilTimerStack.*push*(timerKey);  //获取命名空间  String namespace = mapping.getNamespace();  //获取action配置的name属性  String name = mapping.getName();  //获取action配置的method属性  String method = mapping.getMethod();  Configuration config = configurationManager.getConfiguration();  //根据执行上下文参数，命名空间，名称等创建用户自定义Action的代理对象  ActionProxy proxy = config.getContainer().getInstance(ActionProxyFactory.**class**).createActionProxy(  namespace, name, method, extraContext, **true**, **false**);  request.setAttribute(ServletActionContext.*STRUTS\_VALUESTACK\_KEY*, proxy.getInvocation().getStack());  // if the ActionMapping says to go straight to a result, do it!  //执行execute方法，并转向结果  **if** (mapping.getResult() != **null**) {    Result result = mapping.getResult();  result.execute(proxy.getInvocation());  } **else** {  proxy.execute();  }  // If there was a previous value stack then set it back onto the request  **if** (!nullStack) {  request.setAttribute(ServletActionContext.*STRUTS\_VALUESTACK\_KEY*, stack);  }  } **catch** (ConfigurationException e) {  // WW-2874 Only log error if in devMode  **if**(devMode) {  *LOG*.error("Could not find action or result", e);  }  **else** {  *LOG*.warn("Could not find action or result", e);  }  sendError(request, response, context, HttpServletResponse.*SC\_NOT\_FOUND*, e);  } **catch** (Exception e) {  sendError(request, response, context, HttpServletResponse.*SC\_INTERNAL\_SERVER\_ERROR*, e);  } **finally** {  UtilTimerStack.*pop*(timerKey);  }  } |

# 核心类

## ActionMapper

|  |
| --- |
| **public** **interface** ActionMapper {  //…  } |

|  |
| --- |
| **public** **class** DefaultActionMapper **implements** ActionMapper {  **public** ActionMapping getMapping(HttpServletRequest request,  ConfigurationManager configManager) {  ActionMapping mapping = **new** ActionMapping();  String uri = getUri(request);  **int** indexOfSemicolon = uri.indexOf(";");  uri = (indexOfSemicolon > -1) ? uri.substring(0, indexOfSemicolon) : uri;  uri = dropExtension(uri, mapping);  **if** (uri == **null**) {  **return** **null**;  }  parseNameAndNamespace(uri, mapping, configManager);  handleSpecialParameters(request, mapping);  **if** (mapping.getName() == **null**) {  **return** **null**;  }  parseActionName(mapping);  **return** mapping;  }  **protected** **static** **final** String *METHOD\_PREFIX* = "method:";  **protected** **static** **final** String *ACTION\_PREFIX* = "action:";  **protected** **static** **final** String *REDIRECT\_PREFIX* = "redirect:";  **protected** **static** **final** String *REDIRECT\_ACTION\_PREFIX* = "redirect-action:";  **protected** **boolean** allowDynamicMethodCalls = **true**;  **protected** **boolean** allowSlashesInActionNames = **false**;  **protected** **boolean** alwaysSelectFullNamespace = **false**;  **protected** PrefixTrie prefixTrie = **null**;  **protected** List<String> extensions = **new** ArrayList<String>() {{ add("action"); add("");}};  **protected** Container container;  **public** DefaultActionMapper() {  prefixTrie = **new** PrefixTrie() {  {  //找到action的方法  put(*METHOD\_PREFIX*, **new** ParameterAction() {  **public** **void** execute(String key, ActionMapping mapping) {  mapping  .setMethod(key  .substring(*METHOD\_PREFIX*.length()));  }  });    //找到action的name  put(*ACTION\_PREFIX*, **new** ParameterAction() {  **public** **void** execute(String key, ActionMapping mapping) {  String name = key.substring(*ACTION\_PREFIX*.length());  **if** (allowDynamicMethodCalls) {  **int** bang = name.indexOf('!');  **if** (bang != -1) {  String method = name.substring(bang + 1);  mapping.setMethod(method);  name = name.substring(0, bang);  }  }  mapping.setName(name);  }  });    //找到Result  put(*REDIRECT\_PREFIX*, **new** ParameterAction() {  **public** **void** execute(String key, ActionMapping mapping) {  ServletRedirectResult redirect = **new** ServletRedirectResult();  container.inject(redirect);  redirect.setLocation(key.substring(*REDIRECT\_PREFIX*  .length()));  mapping.setResult(redirect);  }  });  //找到Result  put(*REDIRECT\_ACTION\_PREFIX*, **new** ParameterAction() {  **public** **void** execute(String key, ActionMapping mapping) {  String location = key.substring(*REDIRECT\_ACTION\_PREFIX*  .length());  ServletRedirectResult redirect = **new** ServletRedirectResult();  container.inject(redirect);  String extension = getDefaultExtension();  **if** (extension != **null** && extension.length() > 0) {  location += "." + extension;  }  redirect.setLocation(location);  mapping.setResult(redirect);  }  });  }  };  }    } |

## ActionMapping

ActionMapper其实是HttpServletRequest和Action调用请求的一个映射，它屏蔽了Action对于Request等 java Servlet类的依赖。Struts2中它的默认实现类是DefaultActionMapper，ActionMapper很大的用处可以根据自己的 需要来设计url格式。

|  |
| --- |
| StrutsPrepareAndExcuteFilter#doFilter部分代码片段->  ActionMapping mapping = prepare.findActionMapping(request, response);  PrepareOperations#findActionMapping  **public** ActionMapping findActionMapping(HttpServletRequest request, HttpServletResponse response) {  ActionMapping mapping = (ActionMapping) request.getAttribute(STRUTS\_ACTION\_MAPPING\_KEY);  **if** (mapping == **null**) {  **try** {  mapping = dispatcher.getContainer().getInstance(ActionMapper.**class**).getMapping(request, dispatcher.getConfigurationManager());  **if** (mapping != **null**) {  request.setAttribute(STRUTS\_ACTION\_MAPPING\_KEY, mapping);  }  } **catch** (Exception ex) {  dispatcher.sendError(request, response, servletContext, HttpServletResponse.SC\_INTERNAL\_SERVER\_ERROR, ex);  }  }  **return** mapping;  } |

|  |
| --- |
| **public** **interface** ActionMapper {  //…  } |

|  |
| --- |
| **public** **class** DefaultActionMapper **implements** ActionMapper {  **public** String getUriFromActionMapping(ActionMapping mapping) {  StringBuilder uri = **new** StringBuilder();  **if** (mapping.getNamespace() != **null**) {  uri.append(mapping.getNamespace());  **if** (!"/".equals(mapping.getNamespace())) {  uri.append("/");  }  }  String name = mapping.getName();  String params = "";  **if** (name.indexOf('?') != -1) {  params = name.substring(name.indexOf('?'));  name = name.substring(0, name.indexOf('?'));  }  uri.append(name);  **if** (**null** != mapping.getMethod() && !"".equals(mapping.getMethod())) {  uri.append("!").append(mapping.getMethod());  }  String extension = mapping.getExtension();  **if** (extension == **null**) {  extension = getDefaultExtension();  // Look for the current extension, if available  ActionContext context = ActionContext.getContext();  **if** (context != **null**) {  ActionMapping orig = (ActionMapping) context.get(ServletActionContext.ACTION\_MAPPING);  **if** (orig != **null**) {  extension = orig.getExtension();  }  }  }  **if** (extension != **null**) {  **if** (extension.length() == 0 || (extension.length() > 0 && uri.indexOf('.' + extension) == -1)) {  **if** (extension.length() > 0) {  uri.append(".").append(extension);  }  **if** (params.length() > 0) {  uri.append(params);  }  }  }  **return** uri.toString();  }  //…  } |

## ActionProxy

|  |
| --- |
| StrutsPrepareAndExcuteFilter#doFilter部分代码片段->  execute.executeAction(request, response, mapping);    ExecuteOperations# executeAction  **public** **void** executeAction(HttpServletRequest request, HttpServletResponse response, ActionMapping mapping) **throws** ServletException {  dispatcher.serviceAction(request, response, servletContext, mapping);  }  Dispatcher#ServiceAction 部分代码片段🡪  ActionProxy proxy = config.getContainer().getInstance(ActionProxyFactory.**class**).createActionProxy(  namespace, name, method, extraContext, **true**, **false**);  Dispatcher#ServiceAction 部分代码片段🡪  proxy.execute();    DefaultActionFactoryProxy#createActionProxy🡪  **public** ActionProxy createActionProxy(String namespace, String actionName, String methodName, Map<String, Object> extraContext, **boolean** executeResult, **boolean** cleanupContext) {    ActionInvocation inv = **new** DefaultActionInvocation(extraContext, **true**);  container.inject(inv);  **return** createActionProxy(inv, namespace, actionName, methodName, executeResult, cleanupContext);  }  // executeResult默认true  **public** ActionProxy createActionProxy(ActionInvocation inv, String namespace, String actionName, String methodName, **boolean** executeResult, **boolean** cleanupContext) {  DefaultActionProxy proxy = **new** DefaultActionProxy(inv, namespace, actionName, methodName, executeResult, cleanupContext);  container.inject(proxy);  proxy.prepare();  **return** proxy;  } |

|  |
| --- |
| **public** **interface** ActionProxy {  //…  } |

|  |
| --- |
| **public** **class** DefaultActionProxy **implements** ActionProxy, Serializable {    **protected** ActionInvocation invocation;  **protected** DefaultActionProxy(ActionInvocation inv, String namespace, String actionName, String methodName, **boolean** executeResult, **boolean** cleanupContext) {    **this**.invocation = inv;  **this**.cleanupContext = cleanupContext;  **if** (LOG.isDebugEnabled()) {  LOG.debug("Creating an DefaultActionProxy for namespace " + namespace + " and action name " + actionName);  }  **this**.actionName = actionName;  **this**.namespace = namespace;  **this**.executeResult = executeResult;  **this**.method = methodName;  }  **protected** **void** prepare() {  String profileKey = "create DefaultActionProxy: ";  **try** {  UtilTimerStack.push(profileKey);  config = configuration.getRuntimeConfiguration().getActionConfig(namespace, actionName);    **if** (config == **null** && unknownHandler != **null**) {  config = unknownHandler.handleUnknownAction(namespace, actionName);  }  **if** (config == **null**) {  String message;    **if** ((namespace != **null**) && (namespace.trim().length() > 0)) {  message = LocalizedTextUtil.findDefaultText(XWorkMessages.MISSING\_PACKAGE\_ACTION\_EXCEPTION, Locale.getDefault(), **new** String[]{  namespace, actionName  });  } **else** {  message = LocalizedTextUtil.findDefaultText(XWorkMessages.MISSING\_ACTION\_EXCEPTION, Locale.getDefault(), **new** String[]{  actionName  });  }  **throw** **new** ConfigurationException(message);  }  resolveMethod();    **if** (!config.isAllowedMethod(method)) {  **throw** **new** ConfigurationException("Invalid method: "+method+" for action "+actionName);  }  invocation.init(**this**);  } **finally** {  UtilTimerStack.pop(profileKey);  }  }    **public** String execute() **throws** Exception {  ActionContext nestedContext = ActionContext.getContext();  ActionContext.setContext(invocation.getInvocationContext());  String retCode = **null**;  String profileKey = "execute: ";  **try** {  UtilTimerStack.push(profileKey);    retCode = invocation.invoke();  } **finally** {  **if** (cleanupContext) {  ActionContext.setContext(nestedContext);  }  UtilTimerStack.pop(profileKey);  }  **return** retCode;  }  } |

## ActionInvocation

|  |
| --- |
| StrutsPrepareAndExcuteFilter#doFilter部分代码片段->  execute.executeAction(request, response, mapping);    ExecuteOperations# executeAction  **public** **void** executeAction(HttpServletRequest request, HttpServletResponse response, ActionMapping mapping) **throws** ServletException {  dispatcher.serviceAction(request, response, servletContext, mapping);  }  Dispatcher#ServiceAction 部分代码片段🡪  ActionProxy proxy = config.getContainer().getInstance(ActionProxyFactory.**class**).createActionProxy(  namespace, name, method, extraContext, **true**, **false**);  DefaultActionFactoryProxy#createActionProxy🡪  **public** ActionProxy createActionProxy(String namespace, String actionName, String methodName, Map<String, Object> extraContext, **boolean** executeResult, **boolean** cleanupContext) {  ActionInvocation inv = **new** DefaultActionInvocation(extraContext, **true**);  container.inject(inv);  **return** createActionProxy(inv, namespace, actionName, methodName, executeResult, cleanupContext);  }  ActionProxy#perpare代码片段  invocation.init(**this**);  ActionProxy#excute代码片段  retCode = invocation.invoke(); |

|  |
| --- |
| **public** **interface** ActionInvocation **extends** Serializable {  //…  } |

|  |
| --- |
| **public** **class** DefaultActionInvocation **implements** ActionInvocation {  **public** DefaultActionInvocation(**final** Map<String, Object> extraContext, **final** **boolean** pushAction) {  DefaultActionInvocation.**this**.extraContext = extraContext;  DefaultActionInvocation.**this**.pushAction = pushAction;  }  **public** **void** init(ActionProxy proxy) {  **this**.proxy = proxy;  Map<String, Object> contextMap = createContextMap();  // Setting this so that other classes, like object factories, can use the ActionProxy and other  // contextual information to operate  ActionContext actionContext = ActionContext.getContext();  **if** (actionContext != **null**) {  actionContext.setActionInvocation(**this**);  }  createAction(contextMap);  **if** (pushAction) {  stack.push(action);  contextMap.put("action", action);  }  invocationContext = **new** ActionContext(contextMap);  invocationContext.setName(proxy.getActionName());  // get a new List so we don't get problems with the iterator if someone changes the list  List<InterceptorMapping> interceptorList = **new** ArrayList<InterceptorMapping>(proxy.getConfig().getInterceptors());  interceptors = interceptorList.iterator();  }  }  **protected** **void** createAction(Map<String, Object> contextMap) {  // load action  String timerKey = "actionCreate: " + proxy.getActionName();  **try** {  UtilTimerStack.push(timerKey);  //利用反射构造action  action = objectFactory.buildAction(proxy.getActionName(), proxy.getNamespace(), proxy.getConfig(), contextMap);  } **catch** (InstantiationException e) {  **throw** **new** XWorkException("Unable to intantiate Action!", e, proxy.getConfig());  } **catch** (IllegalAccessException e) {  **throw** **new** XWorkException("Illegal access to constructor, is it public?", e, proxy.getConfig());  } **catch** (Exception e) {  String gripe = "";  **if** (proxy == **null**) {  gripe = "Whoa! No ActionProxy instance found in current ActionInvocation. This is bad ... very bad";  } **else** **if** (proxy.getConfig() == **null**) {  gripe = "Sheesh. Where'd that ActionProxy get to? I can't find it in the current ActionInvocation!?";  } **else** **if** (proxy.getConfig().getClassName() == **null**) {  gripe = "No Action defined for '" + proxy.getActionName() + "' in namespace '" + proxy.getNamespace() + "'";  } **else** {  gripe = "Unable to instantiate Action, " + proxy.getConfig().getClassName() + ", defined for '" + proxy.getActionName() + "' in namespace '" + proxy.getNamespace() + "'";  }  gripe += (((" -- " + e.getMessage()) != **null**) ? e.getMessage() : " [no message in exception]");  **throw** **new** XWorkException(gripe, e, proxy.getConfig());  } **finally** {  UtilTimerStack.pop(timerKey);  }  **if** (actionEventListener != **null**) {  action = actionEventListener.prepare(action, stack);  }  }  **public** String invoke() **throws** Exception {  String profileKey = "invoke: ";  **try** {  UtilTimerStack.push(profileKey);  //已经执行完毕？  **if** (executed) {//默认值false  **throw** **new** IllegalStateException("Action has already executed");  }  //调用拦截器  //Interceptor#interceptor 调用的是同一个ActionInvotion实例的invoke方法；  //对于同一ActionInvotion实例的字段：interceptors（Iterator类型）每调用一次//ActionInvotion#invoke方法interceptors 中的interceptor就少一个。  **if** (interceptors.hasNext()) {  **final** InterceptorMapping interceptor = (InterceptorMapping) interceptors.next();  //  String interceptorMsg = "interceptor: " + interceptor.getName();  UtilTimerStack.push(interceptorMsg);  **try** {  //调用interceptor  resultCode = interceptor.getInterceptor().intercept(DefaultActionInvocation.**this**);  }  **finally** {  UtilTimerStack.pop(interceptorMsg);  }  } **else** {  //调用action  resultCode = invokeActionOnly();  }  // this is needed because the result will be executed, then control will return to the Interceptor, which will  // return above and flow through again  //执行完action的方法后 ，紧接着执行下面的代码，且仅执行一次（下面代码只要执行一次excute=true  //了），也就是说后续的拦截器将不再执行下面的代码，也就是说只要执行了下面的代码Result🡪View就已经  //确定，此时你再在后续的拦截器代码中对Action属性做修改没有蛋用。  //**有的时候，before拦截和after拦截（以ActionVocation.invoke为参照点）对我们来说是不够的，因为我**  **//们需要在Action执行完之后，但是还没有回到视图层之前，做一些事情。Struts2同样支持这样的拦截，这**  **//种拦截方式是：PreResultListener接口。**  **if** (!executed) {  **if** (preResultListeners != **null**) {  **for** (Object preResultListener : preResultListeners) {  PreResultListener listener = (PreResultListener) preResultListener;  String \_profileKey = "preResultListener: ";  **try** {  UtilTimerStack.push(\_profileKey);  listener.beforeResult(**this**, resultCode);  }  **finally** {  UtilTimerStack.pop(\_profileKey);  }  }  }  // now execute the result, if we're supposed to  **if** (proxy.getExecuteResult()) {  //执行结果  executeResult();  }  executed = **true**;  }  **return** resultCode;  }  **finally** {  UtilTimerStack.pop(profileKey);  }  }  **public** String invokeActionOnly() **throws** Exception {  **return** invokeAction(getAction(), proxy.getConfig());  }  **private** **void** executeResult() **throws** Exception {  result = createResult();  String timerKey = "executeResult: " + getResultCode();  **try** {  UtilTimerStack.push(timerKey);  **if** (result != **null**) {  result.execute(**this**);  } **else** **if** (resultCode != **null** && !Action.NONE.equals(resultCode)) {  **throw** **new** ConfigurationException("No result defined for action " + getAction().getClass().getName()  + " and result " + getResultCode(), proxy.getConfig());  } **else** {  **if** (LOG.isDebugEnabled()) {  LOG.debug("No result returned for action " + getAction().getClass().getName() + " at " + proxy.getConfig().getLocation());  }  }  } **finally** {  UtilTimerStack.pop(timerKey);  }  }  **public** Result createResult() **throws** Exception {  **if** (explicitResult != **null**) {  Result ret = explicitResult;  explicitResult = **null**;  **return** ret;  }  ActionConfig config = proxy.getConfig();  Map<String, ResultConfig> results = config.getResults();  ResultConfig resultConfig = **null**;  **synchronized** (config) {  **try** {  resultConfig = results.get(resultCode);  } **catch** (NullPointerException e) {  // swallow  }  **if** (resultConfig == **null**) {  // If no result is found for the given resultCode, try to get a wildcard '\*' match.  resultConfig = results.get("\*");  }  }  **if** (resultConfig != **null**) {  **try** {  **return** objectFactory.buildResult(resultConfig, invocationContext.getContextMap());  } **catch** (Exception e) {  LOG.error("There was an exception while instantiating the result of type " + resultConfig.getClassName(), e);  **throw** **new** XWorkException(e, resultConfig);  }  } **else** **if** (resultCode != **null** && !Action.NONE.equals(resultCode) && unknownHandlerManager.hasUnknownHandlers()) {  **return** unknownHandlerManager.handleUnknownResult(invocationContext, proxy.getActionName(), proxy.getConfig(), resultCode);  }  **return** **null**;  }    **public** Result getResult() **throws** Exception {  Result returnResult = result;  // If we've chained to other Actions, we need to find the last result  **while** (returnResult **instanceof** ActionChainResult) {  ActionProxy aProxy = ((ActionChainResult) returnResult).getProxy();  **if** (aProxy != **null**) {  Result proxyResult = aProxy.getInvocation().getResult();  **if** ((proxyResult != **null**) && (aProxy.getExecuteResult())) {  returnResult = proxyResult;  } **else** {  **break**;  }  } **else** {  **break**;  }  }  **return** returnResult;  } |

## Interceptor

|  |
| --- |
| StrutsPrepareAndExcuteFilter#doFilter部分代码片段->  execute.executeAction(request, response, mapping);    ExecuteOperations# executeAction  **public** **void** executeAction(HttpServletRequest request, HttpServletResponse response, ActionMapping mapping) **throws** ServletException {  dispatcher.serviceAction(request, response, servletContext, mapping);  }  Dispatcher#ServiceAction 部分代码片段🡪  ActionProxy proxy = config.getContainer().getInstance(ActionProxyFactory.**class**).createActionProxy(  namespace, name, method, extraContext, **true**, **false**);  DefaultActionFactoryProxy#createActionProxy🡪  **public** ActionProxy createActionProxy(String namespace, String actionName, String methodName, Map<String, Object> extraContext, **boolean** executeResult, **boolean** cleanupContext) {  ActionInvocation inv = **new** DefaultActionInvocation(extraContext, **true**);  container.inject(inv);  **return** createActionProxy(inv, namespace, actionName, methodName, executeResult, cleanupContext);  }  ActionProxy#perpare代码片段  invocation.init(**this**);  ActionProxy#excute代码片段  retCode = invocation.invoke();  ActionInvocation#invoke  resultCode = interceptor.getInterceptor().intercept(DefaultActionInvocation.**this**); |

|  |
| --- |
| **public** **interface** Interceptor **extends** Serializable {  **void** destroy();  **void** init();  //  String intercept(ActionInvocation invocation) **throws** Exception;  } |

## Result

|  |
| --- |
| StrutsPrepareAndExcuteFilter#doFilter部分代码片段->  execute.executeAction(request, response, mapping);    ExecuteOperations# executeAction->  **public** **void** executeAction(HttpServletRequest request, HttpServletResponse response, ActionMapping mapping) **throws** ServletException {  dispatcher.serviceAction(request, response, servletContext, mapping);  }  Dispatcher#ServiceAction 部分代码片段🡪  **if** (mapping.getResult() != **null**) {  Result result = mapping.getResult();  **result.execute(proxy.getInvocation());**  } **else** {  proxy.execute();  }  或者  ActionInvocation#invoke-🡪  executeResult();  ActionInvocation#excuteReesul }  result=createResult();  result.excute(this); |

|  |
| --- |
| **public** **interface** Result **extends** Serializable {  **public** **void** execute(ActionInvocation invocation) **throws** Exception;  } |

|  |
| --- |
| **public** **abstract** **class** StrutsResultSupport **implements** Result, StrutsStatics {  **private** **static** **final** Logger LOG = LoggerFactory.getLogger(StrutsResultSupport.**class**);  //渲染视图  **public** **void** execute(ActionInvocation invocation) **throws** Exception {  lastFinalLocation = conditionalParse(location, invocation);  doExecute(lastFinalLocation, invocation);  }  **protected** **abstract** **void** doExecute(String finalLocation, ActionInvocation invocation) **throws** Exception;  **protected** String conditionalParse(String param, ActionInvocation invocation) {  **if** (parse && param != **null** && invocation != **null**) {  **return** TextParseUtil.translateVariables(param, invocation.getStack(),  **new** TextParseUtil.ParsedValueEvaluator() {  **public** Object evaluate(Object parsedValue) {  **if** (encode) {  **if** (parsedValue != **null**) {  **try** {  // use UTF-8 as this is the recommended encoding by W3C to  // avoid incompatibilities.  **return** URLEncoder.encode(parsedValue.toString(), "UTF-8");  }  **catch**(UnsupportedEncodingException e) {  LOG.warn("error while trying to encode ["+parsedValue+"]", e);  }  }  }  **return** parsedValue;  }  });  } **else** {  **return** param;  }  }  } |

//默认类型

|  |
| --- |
| **public** **class** ServletDispatcherResult **extends** StrutsResultSupport {  **public** **void** doExecute(String finalLocation, ActionInvocation invocation) **throws** Exception {  **if** (LOG.isDebugEnabled()) {  LOG.debug("Forwarding to location " + finalLocation);  }  PageContext pageContext = ServletActionContext.getPageContext();  **if** (pageContext != **null**) {  pageContext.include(finalLocation);  } **else** {  HttpServletRequest request = ServletActionContext.getRequest();  HttpServletResponse response = ServletActionContext.getResponse();  RequestDispatcher dispatcher = request.getRequestDispatcher(finalLocation);  //add parameters passed on the location to #parameters  // see WW-2120  **if** (invocation != **null** && finalLocation != **null** && finalLocation.length() > 0  && finalLocation.indexOf("?") > 0) {  String queryString = finalLocation.substring(finalLocation.indexOf("?") + 1);  Map parameters = (Map) invocation.getInvocationContext().getContextMap().get("parameters");  Map queryParams = UrlHelper.parseQueryString(queryString, **true**);  **if** (queryParams != **null** && !queryParams.isEmpty())  parameters.putAll(queryParams);  }  // if the view doesn't exist, let's do a 404  **if** (dispatcher == **null**) {  response.sendError(404, "result '" + finalLocation + "' not found");  **return**;  }  // If we're included, then include the view  // Otherwise do forward  // This allow the page to, for example, set content type  **if** (!response.isCommitted() && (request.getAttribute("javax.servlet.include.servlet\_path") == **null**)) {  request.setAttribute("struts.view\_uri", finalLocation);  request.setAttribute("struts.request\_uri", request.getRequestURI());  dispatcher.forward(request, response);  } **else** {  dispatcher.include(request, response);  }  }  }  } |

## ActionContext

|  |
| --- |
| StrutsPrepareAndExcuteFilter#doFilter🡪  prepare.createActionContext(request, response);  PrepareOperations#createActionContext🡪  **public** ActionContext createActionContext(HttpServletRequest request, HttpServletResponse response) {  ActionContext ctx;  Integer counter = 1;  Integer oldCounter = (Integer) request.getAttribute(CLEANUP\_RECURSION\_COUNTER);  **if** (oldCounter != **null**) {  counter = oldCounter + 1;  }    ActionContext oldContext = ActionContext.getContext();  **if** (oldContext != **null**) {  // detected existing context, so we are probably in a forward  ctx = **new** ActionContext(**new** HashMap<String, Object>(oldContext.getContextMap()));  } **else** {  ValueStack stack = dispatcher.getContainer().getInstance(ValueStackFactory.**class**).createValueStack();  stack.getContext().putAll(dispatcher.createContextMap(request, response, **null**, servletContext));  ctx = **new** ActionContext(stack.getContext());  }  request.setAttribute(CLEANUP\_RECURSION\_COUNTER, counter);  ActionContext.setContext(ctx);  **return** ctx;  } |

|  |
| --- |
| **public** **class** ActionContext **implements** Serializable {  **static** ThreadLocal *actionContext* = **new** ThreadLocal();  **static** ThreadLocal actionContext = **new** ThreadLocal();  **public** **static** ActionContext getContext() {  **return** (ActionContext) actionContext.get();  }  **public** **static** **void** setContext(ActionContext context) {  actionContext.set(context);  }  **public** ActionContext(Map<String, Object> context) {  **this**.context = context;  }  **public** ValueStack getValueStack() {  **return** (ValueStack) get(*VALUE\_STACK*);  }  } |

|  |
| --- |
| ActionContext#getContext  **public** **static** ActionContext getContext() {  **return** (ActionContext) actionContext.get();  } |
| **public** **static** **void** setContext(ActionContext context) {  actionContext.set(context);  } |
| ActionContext# actionContext  **static** ThreadLocal actionContext = **new** ThreadLocal(); |

## ValueStack

|  |
| --- |
| StrutsPrepareAndExcuteFilter#doFilter🡪  prepare.createActionContext(request, response);  PrepareOperations#createActionContext🡪  **public** ActionContext createActionContext(HttpServletRequest request, HttpServletResponse response) {  ActionContext ctx;  Integer counter = 1;  Integer oldCounter = (Integer) request.getAttribute(CLEANUP\_RECURSION\_COUNTER);  **if** (oldCounter != **null**) {  counter = oldCounter + 1;  }    ActionContext oldContext = ActionContext.getContext();  **if** (oldContext != **null**) {  // detected existing context, so we are probably in a forward  ctx = **new** ActionContext(**new** HashMap<String, Object>(oldContext.getContextMap()));  } **else** {  ValueStack stack = dispatcher.getContainer().getInstance(ValueStackFactory.**class**).createValueStack();  stack.getContext().putAll(dispatcher.createContextMap(request, response, **null**, servletContext));  ctx = **new** ActionContext(stack.getContext());  }  request.setAttribute(CLEANUP\_RECURSION\_COUNTER, counter);  ActionContext.setContext(ctx);  **return** ctx;  }  Dispatcher#ServiceAction 部分代码片段🡪  extraContext.put(ActionContext.*VALUE\_STACK*, valueStackFactory.createValueStack(stack));  OgnlValueStackFactory#createValueStack-🡪  **public** ValueStack createValueStack() {  ValueStack stack = **new** OgnlValueStack(xworkConverter, compoundRootAccessor, textProvider, allowStaticMethodAccess);  container.inject(stack);  stack.getContext().put(ActionContext.*CONTAINER*, container);  **return** stack;  }  **public** ValueStack createValueStack(ValueStack stack) {  ValueStack result = **new** OgnlValueStack(stack, xworkConverter, compoundRootAccessor, allowStaticMethodAccess);  container.inject(result);  stack.getContext().put(ActionContext.*CONTAINER*, container);  **return** result;  } |

|  |
| --- |
| **public** **class** OgnlValueStack **implements** Serializable, ValueStack, ClearableValueStack, MemberAccessValueStack {  **private** **boolean** devMode;  CompoundRoot root;  **transient** Map<String, Object> context;  Class defaultType;  Map<Object, Object> overrides;  **transient** OgnlUtil ognlUtil;  **transient** SecurityMemberAccess securityMemberAccess;  **protected** OgnlValueStack(XWorkConverter xworkConverter, CompoundRootAccessor accessor, TextProvider prov, **boolean** allowStaticAccess) {  setRoot(xworkConverter, accessor, **new** CompoundRoot(), allowStaticAccess);  push(prov);  }  **protected** OgnlValueStack(ValueStack vs, XWorkConverter xworkConverter, CompoundRootAccessor accessor, **boolean** allowStaticAccess) {  setRoot(xworkConverter, accessor, **new** CompoundRoot(vs.getRoot()), allowStaticAccess);  }  **protected** **void** setRoot(XWorkConverter xworkConverter,  CompoundRootAccessor accessor, CompoundRoot compoundRoot, **boolean** allowStaticMethodAccess) {  **this**.root = compoundRoot;  **this**.securityMemberAccess = **new** SecurityMemberAccess(allowStaticMethodAccess);  **this**.context = Ognl.*createDefaultContext*(**this**.root, accessor, **new** OgnlTypeConverterWrapper(xworkConverter),  securityMemberAccess);  context.put(*VALUE\_STACK*, **this**);  Ognl.*setClassResolver*(context, accessor);  ((OgnlContext) context).setTraceEvaluations(**false**);  ((OgnlContext) context).setKeepLastEvaluation(**false**);  }  } |

### OgnlContext

|  |
| --- |
| **public** **class** OgnlContext **extends** Object **implements** Map  {  //…  } |

### Ognl

|  |
| --- |
| **public** **static** Map createDefaultContext(Object root, ClassResolver classResolver,  TypeConverter converter, MemberAccess memberAccess)  {  **return** *addDefaultContext*(root, classResolver, converter, memberAccess, **new** OgnlContext());  }  **public** **static** Map addDefaultContext(Object root, ClassResolver classResolver,  TypeConverter converter, MemberAccess memberAccess, Map context)  {  OgnlContext result;  **if** (!(context **instanceof** OgnlContext)) {  result = **new** OgnlContext();  result.setValues(context);  } **else** {  result = (OgnlContext) context;  }  **if** (classResolver != **null**) {  result.setClassResolver(classResolver);  }  **if** (converter != **null**) {  result.setTypeConverter(converter);  }  **if** (memberAccess != **null**) {  result.setMemberAccess(memberAccess);  }  result.setRoot(root);  **return** result;  }  **public** **static** Object getValue(Object tree, Map context, Object root)  **throws** OgnlException  {  **return** *getValue*(tree, context, root, **null**);  }  **public** **static** **void** setValue(Object tree, Map context, Object root, Object value)  **throws** OgnlException  {  OgnlContext ognlContext = (OgnlContext) *addDefaultContext*(root, context);  Node n = (Node) tree;  **if** (n.getAccessor() != **null**) {  n.getAccessor().set(ognlContext, root, value);  **return**;  }  n.setValue(ognlContext, root, value);  }  } |

# Configuration

ConfigurationManager实例持有一个Configuration实例，Configuration实例持有一个Container实例。所有的配置我们其实都有Container实例管控。Container实例管控的方式：

放入->inject(Object obj);

取出-> getInstance（Class clazz）;

## Configuration

|  |
| --- |
| **public** **interface** Configuration **extends** Serializable {  **void** rebuildRuntimeConfiguration();  PackageConfig getPackageConfig(String name);  Set<String> getPackageConfigNames();  Map<String, PackageConfig> getPackageConfigs();  } |
| **public** **class** DefaultConfiguration **implements** Configuration {  **protected** **static** **final** Logger *LOG* = LoggerFactory.*getLogger*(DefaultConfiguration.**class**);  // Programmatic Action Configurations  **protected** Map<String, PackageConfig> packageContexts = **new** LinkedHashMap<String, PackageConfig>();  **protected** RuntimeConfiguration runtimeConfiguration;  **protected** Container container;  **protected** String defaultFrameworkBeanName;  **protected** Set<String> loadedFileNames = **new** TreeSet<String>();  **protected** List<UnknownHandlerConfig> unknownHandlerStack;  **public** DefaultConfiguration(String defaultBeanName) {  **this**.defaultFrameworkBeanName = defaultBeanName;  }  **public** **synchronized** **void** reload(List<ConfigurationProvider> providers) **throws** ConfigurationException {  // Silly copy necessary due to lack of ability to cast generic lists  List<ContainerProvider> contProviders = **new** ArrayList<ContainerProvider>();  contProviders.addAll(providers);  reloadContainer(contProviders);  }  **public** **synchronized** List<PackageProvider> reloadContainer(List<ContainerProvider> providers) **throws** ConfigurationException {  packageContexts.clear();  loadedFileNames.clear();  List<PackageProvider> packageProviders = **new** ArrayList<PackageProvider>();  ContainerProperties props = **new** ContainerProperties();  ContainerBuilder builder = **new** ContainerBuilder();  **for** (**final** ContainerProvider containerProvider : providers)  {  containerProvider.init(**this**);  containerProvider.register(builder, props);  }  props.setConstants(builder);  builder.factory(Configuration.**class**, **new** Factory<Configuration>() {  **public** Configuration create(Context context) **throws** Exception {  **return** DefaultConfiguration.**this**;  }  });  ActionContext oldContext = ActionContext.*getContext*();  **try** {  // Set the bootstrap container for the purposes of factory creation  Container bootstrap = createBootstrapContainer();  setContext(bootstrap);  container = builder.create(**false**);  setContext(container);  objectFactory = container.getInstance(ObjectFactory.**class**);  // Process the configuration providers first  **for** (**final** ContainerProvider containerProvider : providers)  {  **if** (containerProvider **instanceof** PackageProvider) {  container.inject(containerProvider);  ((PackageProvider)containerProvider).loadPackages();  packageProviders.add((PackageProvider)containerProvider);  }  }  // Then process any package providers from the plugins  Set<String> packageProviderNames = container.getInstanceNames(PackageProvider.**class**);  **if** (packageProviderNames != **null**) {  **for** (String name : packageProviderNames) {  PackageProvider provider = container.getInstance(PackageProvider.**class**, name);  provider.init(**this**);  provider.loadPackages();  packageProviders.add(provider);  }  }  rebuildRuntimeConfiguration();  } **finally** {  **if** (oldContext == **null**) {  ActionContext.*setContext*(**null**);  }  }  **return** packageProviders;  }  //…  } |

### Container

|  |
| --- |
| **public** **interface** Container **extends** Serializable {  String DEFAULT\_NAME = "default";  **void** inject(Object o);  <T> T inject(Class<T> implementation);  <T> T getInstance(Class<T> type, String name);  <T> T getInstance(Class<T> type);  Set<String> getInstanceNames(Class<?> type);  //…  } |
| **class** ContainerImpl **implements** Container {  **final** Map<Key<?>, InternalFactory<?>> factories;  //…  } |

### ContainerBuilder

|  |
| --- |
| **public** **final** **class** ContainerBuilder {  //…  **public** Container create(**boolean** loadSingletons) {  ensureNotCreated();  created = **true**;  **final** ContainerImpl container = **new** ContainerImpl(  **new** HashMap<Key<?>, InternalFactory<?>>(factories));  **if** (loadSingletons) {  container.callInContext(**new** ContainerImpl.ContextualCallable<Void>() {  **public** Void call(InternalContext context) {  **for** (InternalFactory<?> factory : singletonFactories) {  factory.create(context);  }  **return** **null**;  }  });  }  container.injectStatics(staticInjections);  **return** container;  }  //…  } |

## ConfigurationManager

|  |
| --- |
| **public** **class** ConfigurationManager {  **protected** **static** **final** Logger LOG = LoggerFactory.getLogger(ConfigurationManager.**class**);  **protected** Configuration configuration;  **protected** Lock providerLock = **new** ReentrantLock();  **private** List<ContainerProvider> containerProviders = **new** CopyOnWriteArrayList<ContainerProvider>();  **private** List<PackageProvider> packageProviders = **new** CopyOnWriteArrayList<PackageProvider>();  **protected** String defaultFrameworkBeanName;  **public** ConfigurationManager() {  **this**("xwork");  }    //默认是struts  **public** ConfigurationManager(String name) {  **this**.defaultFrameworkBeanName = name;  }  **public** **synchronized** Configuration getConfiguration() {  **if** (configuration == **null**) {  setConfiguration(**new** DefaultConfiguration(defaultFrameworkBeanName));  **try** {  configuration.reloadContainer(getContainerProviders());  } **catch** (ConfigurationException e) {  setConfiguration(**null**);  **throw** **new** ConfigurationException("Unable to load configuration.", e);  }  } **else** {  conditionalReload();  }  **return** configuration;  }  **public** List<ContainerProvider> getContainerProviders() {  providerLock.lock();  **try** {  **if** (containerProviders.size() == 0) {  containerProviders.add(**new** XWorkConfigurationProvider());  containerProviders.add(**new** XmlConfigurationProvider("xwork.xml", **false**));  }  **return** containerProviders;  } **finally** {  providerLock.unlock();  }  }  //…  } |
| BeanSelectionProvider |

## Provider

### PackageProvider

|  |
| --- |
| **public** **interface** PackageProvider {  } |

### ContainerProvider

|  |
| --- |
| **public** **interface** ContainerProvider {  } |

### ConfigurationProvider

|  |
| --- |
| **public** **interface** ConfigurationProvider {  } |

### LegacyPropertiesConfigurationProvider

|  |
| --- |
| **public** **class** LegacyPropertiesConfigurationProvider **implements** ConfigurationProvider {  } |

### DefaultPropertiesProvider

|  |
| --- |
| **public** **class** DefaultPropertiesProvider **extends** LegacyPropertiesConfigurationProvider {  } |

### BeanSelectionProvider

|  |
| --- |
| **public** **class** BeanSelectionProvider **implements** ConfigurationProvider {  **public** **static** **final** String DEFAULT\_BEAN\_NAME = "struts";  //…  } |

# Ognl表达式

OGNL（Object Graph Navigation Language），是一种表达式语言。

使用这种表达式语言，你可以通过某种表达式语法，存取Java对象树中的任意属性、调用Java对象树的方法、同时能够自动实现必要的类型转化。如果我们把表达式看做是一个带有语义的字符串，那么OGNL无疑成为了这个语义字符串与Java对象之间沟通的桥梁。

|  |
| --- |
| **public** **static** Object getValue(Object tree, Map context, Object root)  **throws** OgnlException  {  **return** *getValue*(tree, context, root, **null**);  }  **public** **static** Object getValue(Object tree, Map context, Object root, Class resultType)  **throws** OgnlException  {  Object result;  OgnlContext ognlContext = (OgnlContext) *addDefaultContext*(root, context);  Node node = (Node)tree;  **if** (node.getAccessor() != **null**)  result = node.getAccessor().get(ognlContext, root);  **else**  result = node.getValue(ognlContext, root);  **if** (resultType != **null**) {  result = *getTypeConverter*(context).convertValue(context, root, **null**, **null**, result, resultType);  }  **return** result;  }  **public** **static** **void** setValue(Object tree, Map context, Object root, Object value)  **throws** OgnlException  {  OgnlContext ognlContext = (OgnlContext) *addDefaultContext*(root, context);  Node n = (Node) tree;  **if** (n.getAccessor() != **null**) {  n.getAccessor().set(ognlContext, root, value);  **return**;  }  n.setValue(ognlContext, root, value);  }  } |

## OGNL三要素

我把传入OGNL的API的三个参数，称之为OGNL的三要素。OGNL的操作实际上就是围绕着这三个参数而进行的。

### 表达式（Expression）

表达式是整个OGNL的核心，所有的OGNL操作都是针对表达式的解析后进行的。表达式会规定此次OGNL操作到底要**干什么**。

我们可以看到，在上面的测试中，name、department.name等都是表达式，表示取name或者department中的name的值。OGNL支持很多类型的表达式，之后我们会看到更多。

### 根对象（Root Object）

根对象可以理解为OGNL的**操作对象**。在表达式规定了“干什么”以后，你还需要指定到底**“对谁干”**。

在上面的测试代码中，user就是根对象。这就意味着，我们需要对user这个对象去取name这个属性的值（对user这个对象去设置其中的department中的name属性值）。

### 上下文环境（Context）

有了表达式和根对象，我们实际上已经可以使用OGNL的基本功能。例如，根据表达式对根对象进行取值或者设值工作。

不过实际上，在OGNL的内部，所有的操作都会在一个特定的环境中运行，这个环境就是OGNL的上下文环境（Context）。说得再明白一些，就是这个上下文环境（Context），将规定OGNL的操作**“在哪里干”**。

OGNL的上下文环境是一个Map结构，称之为OgnlContext。上面我们提到的根对象（Root Object），事实上也会被加入到上下文环境中去，并且这将作为一个特殊的变量进行处理，具体就表现为针对根对象（Root Object）的存取操作的表达式是不需要增加#符号进行区分的。

OgnlContext不仅提供了OGNL的运行环境。在这其中，我们还能设置一些自定义的parameter到Context中，以便我们在进 行OGNL操作的时候能够方便的使用这些parameter。不过正如我们上面反复强调的，我们在访问这些parameter时，需要使用#作为前缀才能 进行。

## 常用的OGNL表达式

|  |
| --- |
| struts2的ognlContext |

1. **基本对象树的访问**   
     
   对象树的访问就是通过使用点号将对象的引用串联起来进行。   
     
   例如：name，department.name，user.department.factory.manager.name   
     
   **2. 对容器变量的访问**   
     
   对容器变量的访问，通过#符号加上表达式进行。   
     
   例如：#name，#department.name，#user.department.factory.manager.name   
     
   **3. 使用操作符号**   
     
   OGNL表达式中能使用的操作符基本跟Java里的操作符一样，除了能使用 +, -, \*, /, ++, --, ==, !=, = 等操作符之外，还能使用 mod, in, not in等。   
     
   **4. 容器、数组、对象**   
     
   OGNL支持对数组和ArrayList等容器的顺序访问：   
     
   例如：group.users[0]   
     
   同时，OGNL支持对Map的按键值查找：   
     
   例如：#session['mySessionPropKey']   
     
   不仅如此，OGNL还支持容器的构造的表达式：   
     
   例如：{"green", "red", "blue"}构造一个List，#{"key1" : "value1", "key2" : "value2", "key3" : "value3"}构造一个Map   
     
   你也可以通过任意类对象的构造函数进行对象新建：   
     
   例如：new java.net.URL("http://localhost/")   
     
   **5. 对静态方法或变量的访问**   
     
   要引用类的静态方法和字段，他们的表达方式是一样的@class@member或者@class@method(args)：   
     
   例如：@com.javaeye.core.Resource@ENABLE，@com.javaeye.core.Resource@getAllResources   
     
   **6. 方法调用**   
     
   直接通过类似Java的方法调用方式进行，你甚至可以传递参数：   
     
   例如：user.getName()，group.users.size()，group.containsUser(#requestUser)   
     
   **7. 投影和选择**   
     
     
   OGNL支持类似数据库中的投影（projection） 和选择（selection）。   
     
   投影就是选出集合中每个元素的相同属性组成新的集合，类似于关系数据库的字段操作。投影操作语法为 collection.{XXX}，其中XXX 是这个集合中每个元素的公共属性。   
     
   例如：group.userList.{username}将获得某个group中的所有user的name的列表。   
     
   选择就是过滤满足selection 条件的集合元素，类似于关系数据库的纪录操作。选择操作的语法为：collection.{X YYY}，其中X 是一个选择操作符，后面则是选择用的逻辑表达式。而选择操作符有三种：   
   ? 选择满足条件的所有元素   
   ^ 选择满足条件的第一个元素   
   $ 选择满足条件的最后一个元素   
     
   例如：group.userList.{? #this.name != null}将获得某个group中user的name不为空的user的列表。

## this指针

我们知道，OGNL表达式是以点进行串联的一个字符串链式表达式。而这个表达式在进行计算的时候，从左到右，每一次表达式计算返回的结果成为当前对象，并 继续进行计算，直到得到计算结果。每次计算的中间对象都会放在一个叫做this的变量里面这个this变量就称之为this指针。   
  
例如：group.userList.size().(#this+1).toString()   
  
在这个例子中，#this其实就是group.userList.size()的计算结构。   
  
使用this指针，我们就可以在OGNL表达式中进行一些简单的计算，从而完成我们的计算逻辑，而this指针在lamba表达式的引用中尤为广泛，有兴趣的读者可以深入研究OGNL自带的文档中lamba表达式的章节。

## 默认行为和类型转化

在我们所讲述的所有的OGNL的操作中，实际上，全部都忽略了OGNL内部帮助你完成的很多默认行为和类型转化方面的工作。   
  
我们来看一下OGNL在进行操作初始化时候的一个函数签名：

|  |
| --- |
| public static Map addDefaultContext( Object root, ClassResolver classResolver, TypeConverter converter, MemberAccess memberAccess, Map context ); |

可以看到，在初始化时，OGNL还需要额外初始化一个类型转化的接口和一些其他的信息。只不过这些默认行为，由OGNL的内部屏蔽了。   
  
一旦需要自己定义针对某个特定类型的类型转化方式，你就需要实现TypeConverter接口，并且在OGNL中进行注册。   
  
同时，如果需要对OGNL的许多默认行为做出改变，则需要通过设置OGNL的全局环境变量进行。

## 实战

我们先从一个例子开始，看看数据在Struts2中是如何运转的。

|  |
| --- |
| /\*\*  \* @author Downpour  \*/  public class User {    private Integer id;    private String name;    private Department department = new Department();    // setter and getters  }  //=========================================================================  /\*\*  \* @author Downpour  \*/  public class Department {    private Integer id;    private String name;    // setter and getters  }  //=========================================================================  <form method="post" action="/struts-example/ognl.action">  user name: <input type="text" name="user.name" value="downpour" />  department name: <input type="text" name="department.name" value="dev" />  <input type="submit" value="submit" />  </form>  //=========================================================================  /\*\*  \* @author Downpour  \*/  public class OgnlAction extends ActionSupport {  private static final Log logger = LogFactory.getLog(OgnlAction.class);  private User user;    private Department department;    /\* (non-Javadoc)  \* @see com.opensymphony.xwork2.ActionSupport#execute()  \*/  @Override  public String execute() throws Exception {  logger.info("user name:" + user.getName()); // -> downpour  logger.info("department name:" + department.getName()); // -> dev  return super.execute();  }  // setter and getters  }  //=========================================================================  user name: <s:property value="user.name" />  department name: <s:property value="department.name" />  //========================================================================= |

我们可以看到在JSP中，form中的元素input等，都使用OGNL的表达式作为name的值。而在form提交时，这些值都会被设置到 Action中的Java对象中。而当Action转向到JSP时，Struts2的Tag又可以从Action的Java对象中，通过OGNL进行取 值。   
  
在这里，你看不到任何的OGNL的代码级别操作，因为这些都在Struts2内部进行了封装。而这些封装，都是建立在OGNL的基本概念，也就是根对象和上下文环境之上。下面就分别就这两个方面分别进行讲解。

**提问：在Struts2中，如何使用自身的Tag读取Action中的变量？**   
  
Struts2自身的Tag会根据value中的OGNL表达式，在ValueStack中寻找相应的对象。因为action在 ValueStack的顶部，所以默认情况下，Struts2的Tag中的OGNL表达式将查找action中的变量。请注意，value中的内容直接是 OGNL表达式，无需任何el的标签包装。   
  
例如：<s:property value="user.name" />   
  
**提问：在Struts2中，如何使用自身的Tag读取HttpServletRequest，HttpSession中的变量？**   
  
在上面的知识中，我们知道，Struts2中OGNL的上下文环境中，包含request，session，application等 servlet对象的Map封装。既然这些对象都在OGNL的上下文中，那么根据OGNL的基本知识，我们可以通过在表达式前面加上#符号来对这些变量的 值进行访问。   
  
例如：<s:property value="%{#application.myApplicationAttribute}" />   
<s:property value="%{#session.mySessionAttribute}" />   
<s:property value="%{#request.myRequestAttribute}" />   
<s:property value="%{#parameters.myParameter}" />   
  
在这里啰嗦一句，在Tag的value中包括%{开头和}结尾的字符串，不知道Struts2为什么要做出这样的设置，从源码上看，它似乎没有什么特别额外的作用：

Java代码  [收藏代码](javascript:void())

1. if (value == null) {
2. value = "top";
3. }
4. else if (altSyntax()) {
5. // the same logic as with findValue(String)
6. // if value start with %{ and end with }, just cut it off!
7. if (value.startsWith("%{") && value.endsWith("}")) {
8. value = value.substring(2, value.length() - 1);
9. }
10. }
12. // exception: don't call findString(), since we don't want the
13. //            expression parsed in this one case. it really
14. //            doesn't make sense, in fact.
15. actualValue = (String) getStack().findValue(value, String.class);
17. ......
19. }

有兴趣的朋友可以研究一下，这一对符号的原理究竟是什么。   
  
**提问：在Struts2中，如何使用JSTL来读取Action中的变量？**   
  
这是一个历史悠久的问题。因为事实上，很多朋友（包括我在内）是不使用Struts2自身的标签库，而是使用JSTL的，可能因为JSTL标签库比较少，简单易用的原因吧。   
  
我们知道，JSTL默认是从page，request，session，application这四个Scope逐次查找相应的EL表达式所对应 的对象的值。那么如果要使用JSTL来读取Action中的变量，就需要把Action中的变量，放到request域中才行。所以，早在 Webwork2.1.X的年代，我们会编写一个拦截器来做这个事情的。大致的原理是：在Action执行完返回之前，依次读取Action中的所有的变 量，并依次调用request.setAttribute()来进行设置。具体的整合方式，请参考以下这篇文档：<http://wiki.opensymphony.com/display/WW/Using+WebWork+and+XWork+with+JSP+2.0+and+JSTL+1.1>   
  
不过随着时代的发展，上面的这种方式，已经不再被推荐使用了。（虽然如此，我们依然可以学习它的一个解决问题的思路）目前来说，自从 Webwork2.2以后，包括Struts2，都使用另外一种整合方式：对HttpServletRequest进行装饰。让我们来看一下源码：

Java代码  [收藏代码](javascript:void())

1. public class StrutsRequestWrapper extends HttpServletRequestWrapper {
3. /\*\*
4. \* The constructor
5. \* @param req The request
6. \*/
7. public StrutsRequestWrapper(HttpServletRequest req) {
8. super(req);
9. }
11. /\*\*
12. \* Gets the object, looking in the value stack if not found
13. \*
14. \* @param s The attribute key
15. \*/
16. public Object getAttribute(String s) {
17. if (s != null && s.startsWith("javax.servlet")) {
18. // don't bother with the standard javax.servlet attributes, we can short-circuit this
19. // see WW-953 and the forums post linked in that issue for more info
20. return super.getAttribute(s);
21. }
23. ActionContext ctx = ActionContext.getContext();
24. Object attribute = super.getAttribute(s);
26. boolean alreadyIn = false;
27. Boolean b = (Boolean) ctx.get("\_\_requestWrapper.getAttribute");
28. if (b != null) {
29. alreadyIn = b.booleanValue();
30. }
32. // note: we don't let # come through or else a request for
33. // #attr.foo or #request.foo could cause an endless loop
34. if (!alreadyIn && attribute == null && s.indexOf("#") == -1) {
35. try {
36. // If not found, then try the ValueStack
37. ctx.put("\_\_requestWrapper.getAttribute", Boolean.TRUE);
38. ValueStack stack = ctx.getValueStack();
39. if (stack != null) {
40. attribute = stack.findValue(s);
41. }
42. } finally {
43. ctx.put("\_\_requestWrapper.getAttribute", Boolean.FALSE);
44. }
45. }
46. return attribute;
47. }
48. }

看到了嘛？这个类会在Struts2初始化的时候，替换HttpServletRequest，运行于整个Struts2的运行过程中，当我们试 图调用request.getAttribute()的时候，就会执行上面的这个方法。（这是一个典型的装饰器模式）在执行上面的方法时，会首先调用 HttpServletRequest中原本的request.getAttribute()，如果没有找到，它会继续到ValueStack中去查找， 而action在ValueStack中，所以action中的变量通过OGNL表达式，就能找到对应的值了。   
  
在这里，在el表达式广泛使用的今天，JSTL1.1以后，也支持直接使用el表达式。注意与直接使用struts2的tag的区别，这里需要使用el的表示符号：${}   
  
例如：${user.name}, <c:out value="${department.name}" />   
  
**提问：在Struts2中，如何使用Freemarker等模板来读取Action中的变量以及HttpServletRequest和HttpSession中的变量？**   
  
Freemarker等模板在Struts2中有对应的Result，而在这些Result中，Freemarker等模板会根据 ValueStack和ActionContext中的内容，构造这些模板可识别的Model，从而使得模板可以以他们各自的语法对ValueStack 和ActionContext中的内容进行读取。   
  
有关Freemarker对于变量的读取，可以参考Struts2的官方文档，非常详细：<http://struts.apache.org/2.0.14/docs/freemarker.html>   
  
**设值计算**   
  
Struts2中使用OGNL进行设值计算，就是指View层传递数据到Control层，并且能够设置到相应的Java对象中。这个过程从逻辑上说需要分成两步来完成：   
  
1. 对于每个请求，都建立一个与相应Action对应的ActionContext作为OGNL的上下文环境和ValueStack，并且把Action压入ValueStack   
  
2. 在请求进入Action代码前，通过某种通用的机制，搜集页面上传递过来的参数，并调用OGNL相关的代码，对Action进行设值。   
  
上面的第一个步骤，在处理URL请求时完成，而第二个步骤，则涉及到另外一个XWork的核心知识：拦截器。所以有关Struts2使用OGNL进行设值计算的详细分析，将会在拦截器章节具体给出。

# 类型转换

# 验证框架

# 责任链模式

Chain of Responsibility定义：Chain of Responsibility(CoR) 是**责任链模式是一种对象的行为模式。一个请求有n多个对象来处理，这n多个对象组成一条链，链关系的维护是靠对象持有下个对象的引用来完成的，每个对象的职责都是处理这个请求，当其中一个对象处理了这个请求，则其余的对象不再处理。**

1.CoR的优点：因为无法预知来自外界的请求是属于哪种类型，每个类如果碰到它不能处理的请求只要放弃就可以。无疑这降低了类之间的耦合性。

2.CoR的缺点是效率低，因为一个请求的完成可能要遍历到最后才可能完成，

# Struts2工作流程

①. 请求发送给 StrutsPrepareAndExecuteFilter

②. StrutsPrepareAndExecuteFilter 判定该请求是否是一个 Struts2 请求

③. 若该请求是一个 Struts2 请求，则 StrutsPrepareAndExecuteFilter 把请求的处理交给 ActionProxy

④. ActionProxy 创建一个 ActionInvocation 的实例，并进行初始化

⑤. ActionInvocation 实例在调用 Action 的过程前后，涉及到相关拦截器（Intercepter）的调用。

⑥. Action 执行完毕，ActionInvocation 负责根据 struts.xml 中的配置找到对应的返回结果。调用结果的 execute 方法，渲染结果。

⑦. 执行各个拦截器 invocation.invoke() 之后的代码

⑧. 把结果发送到客户端

Invocation.invoke-执行到中间🡪

interceptor.interceptor1 调用 Invocation.invoke-执行到中间🡪