**struts2的核心和工作原理**

原创 2014年05月30日 16:49:59

* 328593

    在学习struts2之前，首先我们要明白使用struts2的目的是什么？它能给我们带来什么样的好处？

设计目标

    Struts设计的第一目标就是使MVC模式应用于web程序设计。在这儿MVC模式的好处就不在提了。

技术优势

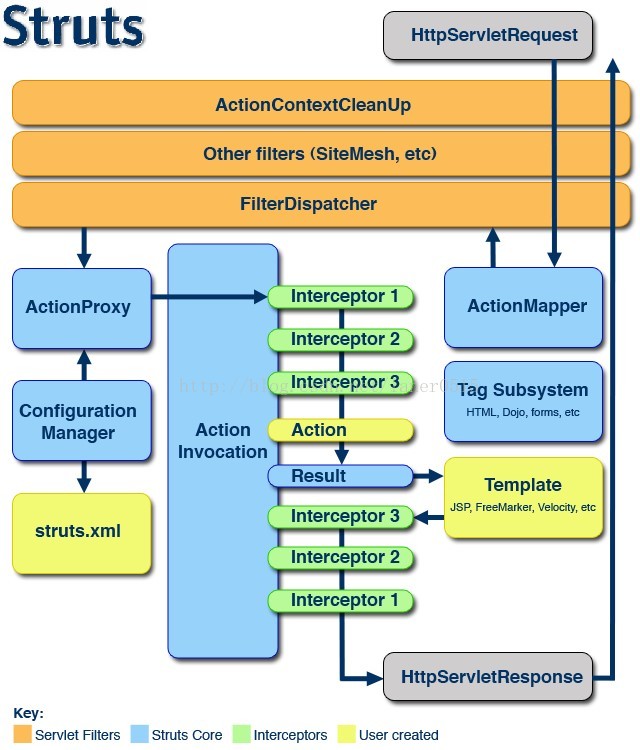
    Struts2有两方面的技术优势，一是所有的Struts2应用程序都是基于client/server HTTP交换协议，The Java Servlet API揭示了Java Servlet只是Java API的一个很小子集，这样我们可以在业务逻辑部分使用功能强大的Java语言进行程序设计。

    二是提供了对MVC的一个清晰的实现，这一实现包含了很多参与对所以请求进行处理的关键组件，如：拦截器、OGNL表达式语言、堆栈。

    因为struts2有这样目标，并且有这样的优势，所以，这是我们学习struts2的理由，下面，我们在深入剖析一下struts的工作原理。

工作原理

    Suruts2的工作原理可以用下面这张图来描述，下面我们分步骤介绍一下每一步的核心内容



    一个请求在Struts2框架中的处理大概分为以下几个步骤

    1、客户端初始化一个指向Servlet容器（例如Tomcat）的请求

    2、这个请求经过一系列的过滤器（Filter）（这些过滤器中有一个叫做ActionContextCleanUp的可选过滤器，这个过滤器对于Struts2和其他框架的集成很有帮助，例如：SiteMesh Plugin）

    3、接着FilterDispatcher被调用，FilterDispatcher询问ActionMapper来决定这个请是否需要调用某个Action

       FilterDispatcher是控制器的核心，就是mvc中c控制层的核心。下面粗略的分析下我理解的FilterDispatcher工作流程和原理：FilterDispatcher进行初始化并启用核心doFilter

**[html]** [view plain](http://blog.csdn.net/laner0515/article/details/27692673/) [copy](http://blog.csdn.net/laner0515/article/details/27692673/)

1. public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException, ServletException ...{
2. HttpServletRequest request = (HttpServletRequest) req;
3. HttpServletResponse response = (HttpServletResponse) res;
4. ServletContext servletContext = filterConfig.getServletContext();
5. // 在这里处理了HttpServletRequest和HttpServletResponse。
6. DispatcherUtils du = DispatcherUtils.getInstance();
7. du.prepare(request, response);//正如这个方法名字一样进行locale、encoding以及特殊request parameters设置
8. try ...{
9. request = du.wrapRequest(request, servletContext);//对request进行包装
10. } catch (IOException e) ...{
11. String message = "Could not wrap servlet request with MultipartRequestWrapper!";
12. LOG.error(message, e);
13. throw new ServletException(message, e);
14. }
15. ActionMapperIF mapper = ActionMapperFactory.getMapper();//得到action的mapper
16. ActionMapping mapping = mapper.getMapping(request);// 得到action 的 mapping
17. if (mapping == null) ...{
18. // there is no action in this request, should we look for a static resource?
19. String resourcePath = RequestUtils.getServletPath(request);
20. if ("".equals(resourcePath) && null != request.getPathInfo()) ...{
21. resourcePath = request.getPathInfo();
22. }
23. if ("true".equals(Configuration.get(WebWorkConstants.WEBWORK\_SERVE\_STATIC\_CONTENT))
24. && resourcePath.startsWith("/webwork")) ...{
25. String name = resourcePath.substring("/webwork".length());
26. findStaticResource(name, response);
27. } else ...{
28. // this is a normal request, let it pass through
29. chain.doFilter(request, response);
30. }
31. // WW did its job here
32. return;
33. }
34. Object o = null;
35. try ...{
36. //setupContainer(request);
37. o = beforeActionInvocation(request, servletContext);
38. //整个框架最最核心的方法，下面分析
39. du.serviceAction(request, response, servletContext, mapping);
40. } finally ...{
41. afterActionInvocation(request, servletContext, o);
42. ActionContext.setContext(null);
43. }
44. }
45. du.serviceAction(request, response, servletContext, mapping);
46. //这个方法询问ActionMapper是否需要调用某个Action来处理这个（request）请求，如果ActionMapper决定需要调用某个Action，FilterDispatcher把请求的处理交给ActionProxy
48. public void serviceAction(HttpServletRequest request, HttpServletResponse response, String namespace, String actionName, Map requestMap, Map parameterMap, Map sessionMap, Map applicationMap) ...{
49. HashMap extraContext = createContextMap(requestMap, parameterMap, sessionMap, applicationMap, request, response, getServletConfig());  //实例化Map请求 ，询问ActionMapper是否需要调用某个Action来处理这个（request）请求
50. extraContext.put(SERVLET\_DISPATCHER, this);
51. OgnlValueStack stack = (OgnlValueStack) request.getAttribute(ServletActionContext.WEBWORK\_VALUESTACK\_KEY);
52. if (stack != null) ...{
53. extraContext.put(ActionContext.VALUE\_STACK,new OgnlValueStack(stack));
54. }
55. try ...{
56. ActionProxy proxy = ActionProxyFactory.getFactory().createActionProxy(namespace, actionName, extraContext);
57. //这里actionName是通过两道getActionName解析出来的, FilterDispatcher把请求的处理交给ActionProxy，下面是ServletDispatcher的 TODO:
58. request.setAttribute(ServletActionContext.WEBWORK\_VALUESTACK\_KEY, proxy.getInvocation().getStack());
59. proxy.execute();
60. //通过代理模式执行ActionProxy
61. if (stack != null)...{
62. request.setAttribute(ServletActionContext.WEBWORK\_VALUESTACK\_KEY,stack);
63. }
64. } catch (ConfigurationException e) ...{
65. log.error("Could not find action", e);
66. sendError(request, response, HttpServletResponse.SC\_NOT\_FOUND, e);
67. } catch (Exception e) ...{
68. log.error("Could not execute action", e);
69. sendError(request, response, HttpServletResponse.SC\_INTERNAL\_SERVER\_ERROR, e);
70. }
71. }

    4、如果ActionMapper决定需要调用某个Action，FilterDispatcher把请求的处理交给ActionProxy

    5、ActionProxy通过ConfigurationManager询问框架的配置文件，找到需要调用的Action类 ,这里，我们一般是从struts.xml配置中读取。

    6、ActionProxy创建一个ActionInvocation的实例。

    7、ActionInvocation实例使用命名模式来调用，在调用Action的过程前后，涉及到相关拦截器（Intercepter）的调用。

    下面我们来看看ActionInvocation是如何工作的：

    ActionInvocation是Xworks 中Action 调度的核心。而对Interceptor 的调度，也正是由ActionInvocation负责。ActionInvocation 是一个接口，而DefaultActionInvocation 则是Webwork 对ActionInvocation的默认实现。

    Interceptor的调度流程大致如下：

    1.ActionInvocation初始化时，根据配置，加载Action相关的所有Interceptor。

    2. 通过ActionInvocation.invoke方法调用Action实现时，执行Interceptor。

    Interceptor将很多功能从我们的Action中独立出来，大量减少了我们Action的代码，独立出来的行为具有很好的重用性。XWork、WebWork的许多功能都是有Interceptor实现，可以在配置文件中组装Action用到的Interceptor，它会按照你指定的顺序，在Action执行前后运行。

    这里，我们简单的介绍一下Interceptor

    在struts2中自带了很多拦截器，在struts2-core-2.1.6.jar这个包下的struts-default.xml中我们可以发现：

**[html]** [view plain](http://blog.csdn.net/laner0515/article/details/27692673/) [copy](http://blog.csdn.net/laner0515/article/details/27692673/)

1. **<interceptors>**
2. **<interceptor** name="alias"class="com.opensymphony.xwork2.interceptor.AliasInterceptor"**/>**
3. **<interceptor** name="autowiring"class="com.opensymphony.xwork2.spring.interceptor.ActionAutowiringInterceptor"**/>**
4. **<interceptor** name="chain"class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"**/>**
5. **<interceptor** name="conversionError"class="org.apache.struts2.interceptor.StrutsConversionErrorInterceptor"**/>**
6. **<interceptor** name="clearSession"class="org.apache.struts2.interceptor.ClearSessionInterceptor"**/>**
7. **<interceptor** name="createSession"class="org.apache.struts2.interceptor.CreateSessionInterceptor"**/>**
8. **<interceptor** name="debugging"class="org.apache.struts2.interceptor.debugging.DebuggingInterceptor"**/>**
9. **<interceptor** name="externalRef"class="com.opensymphony.xwork2.interceptor.ExternalReferencesInterceptor"**/>**
10. **<interceptor** name="execAndWait"class="org.apache.struts2.interceptor.ExecuteAndWaitInterceptor"**/>**
11. **<interceptor** name="exception"class="com.opensymphony.xwork2.interceptor.ExceptionMappingInterceptor"**/>**
12. **<interceptor** name="fileUpload"class="org.apache.struts2.interceptor.FileUploadInterceptor"**/>**
13. **<interceptor** name="i18n"class="com.opensymphony.xwork2.interceptor.I18nInterceptor"**/>**
14. **<interceptor** name="logger"class="com.opensymphony.xwork2.interceptor.LoggingInterceptor"**/>**
15. **<interceptor** name="modelDriven"class="com.opensymphony.xwork2.interceptor.ModelDrivenInterceptor"**/>**
16. **<interceptor** name="scopedModelDriven"class="com.opensymphony.xwork2.interceptor.ScopedModelDrivenInterceptor"**/>**
17. **<interceptor** name="params"class="com.opensymphony.xwork2.interceptor.ParametersInterceptor"**/>**
18. **<interceptor** name="actionMappingParams"class="org.apache.struts2.interceptor.ActionMappingParametersInteceptor"**/>**
19. **<interceptor** name="prepare"class="com.opensymphony.xwork2.interceptor.PrepareInterceptor"**/>**
20. **<interceptor** name="staticParams"class="com.opensymphony.xwork2.interceptor.StaticParametersInterceptor"**/>**
21. **<interceptor** name="scope"class="org.apache.struts2.interceptor.ScopeInterceptor"**/>**
22. **<interceptor** name="servletConfig"class="org.apache.struts2.interceptor.ServletConfigInterceptor"**/>**
23. **<interceptor** name="sessionAutowiring"class="org.apache.struts2.spring.interceptor.SessionContextAutowiringInterceptor"**/>**
24. **<interceptor** name="timer"class="com.opensymphony.xwork2.interceptor.TimerInterceptor"**/>**
25. **<interceptor** name="token"class="org.apache.struts2.interceptor.TokenInterceptor"**/>**
26. **<interceptor** name="tokenSession"class="org.apache.struts2.interceptor.TokenSessionStoreInterceptor"**/>**
27. **<interceptor** name="validation"class="org.apache.struts2.interceptor.validation.AnnotationValidationInterceptor"**/>**
28. **<interceptor** name="workflow"class="com.opensymphony.xwork2.interceptor.DefaultWorkflowInterceptor"**/>**
29. **<interceptor** name="store"class="org.apache.struts2.interceptor.MessageStoreInterceptor"**/>**
30. **<interceptor** name="checkbox"class="org.apache.struts2.interceptor.CheckboxInterceptor"**/>**
31. **<interceptor** name="profiling"class="org.apache.struts2.interceptor.ProfilingActivationInterceptor"**/>**
32. **<interceptor** name="roles"class="org.apache.struts2.interceptor.RolesInterceptor"**/>**
33. **<interceptor** name="jsonValidation"class="org.apache.struts2.interceptor.validation.JSONValidationInterceptor"**/>**
34. **<**interceptorname**interceptorname**="annotationWorkflow"class="com.opensymphony.xwork2.interceptor.annotations.AnnotationWorkflowInterceptor"**/>**

    对于sturts2自带的拦截器，使用起来就相对比较方便了，我们只需要在struts.xml的action标签中加入<interceptor-ref name="*logger* " />并且struts.xml扩展struts-default，就可以使用，

   如果是要自定义拦截器，首先需要写一个拦截器的类：

**[html]** [view plain](http://blog.csdn.net/laner0515/article/details/27692673/) [copy](http://blog.csdn.net/laner0515/article/details/27692673/)

1. package ceshi;
2. import com.opensymphony.xwork2.ActionInvocation;
3. import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
5. publicclassAuthorizationInterceptor extends AbstractInterceptor {
7. @Override
8. public Stringintercept(ActionInvocation ai)throws Exception {
10. System.out.println("abc");
11. return ai.invoke();
13. }
15. }

并且在struts.xml中进行配置

**[html]** [view plain](http://blog.csdn.net/laner0515/article/details/27692673/) [copy](http://blog.csdn.net/laner0515/article/details/27692673/)

1. <!DOCTYPEstruts PUBLIC
2. "-//Apache SoftwareFoundation//DTD Struts Configuration 2.0//EN"
3. "http://struts.apache.org/dtds/struts-2.0.dtd"**>**

6. **<struts>**
7. **<package** name="test"extends="struts-default"**>**
8. **<interceptors>**
9. **<interceptor** name="abc"class ="ceshi.AuthorizationInterceptor"**/>**
10. **</interceptors>**
11. **<action** name="TestLogger"class="vaannila.TestLoggerAction"**>**
12. **<interceptor-refname**interceptor-refname="abc"**/>**
13. **<result** name="success"**>**/success.jsp**</result>**
14. **</action>**
15. **</package>**
16. **</struts>**

    8、一旦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。返回结果通常是（但不总是，也可能是另外的一个Action链）一个需要被表示的JSP或者FreeMarker的模版。在表示的过程中可以使用Struts2 框架中继承的标签。在这个过程中需要涉及到ActionMapper

在上述过程中所有的对象（Action，Results，Interceptors，等）都是通过ObjectFactory来创建的。

Struts2和struts1的比较

    struts2相对于struts1来说简单了很多，并且功能强大了很多，我们可以从几个方面来看：

    从体系结构来看：struts2大量使用拦截器来出来请求，从而允许与业务逻辑控制器 与 servlet-api分离，避免了侵入性；而struts1.x在action中明显的侵入了servlet-api.

    从线程安全分析：struts2.x是线程安全的，每一个对象产生一个实例，避免了线程安全问题；而struts1.x在action中属于单线程。

    性能方面：struts2.x测试可以脱离web容器，而struts1.x依赖servlet-api，测试需要依赖web容器。

    请求参数封装对比：struts2.x使用ModelDriven模式，这样我们 直接 封装model对象，无需要继承任何struts2的基类，避免了侵入性。

    标签的优势：标签库几乎可以完全替代JSTL的标签库，并且 struts2.x支持强大的ognl表达式。

    当然，struts2和struts1相比，在 文件上传，数据校验 等方面也 方便了好多。在这就不详谈了。

    一个比较优秀的框架可以帮着我们更高效，稳定的开发合格的产品，不过我们也不要依赖框架，我们只要理解了思想，设计模式，我们可以自己扩展功能，不然 就要 永远让别人牵着走了！