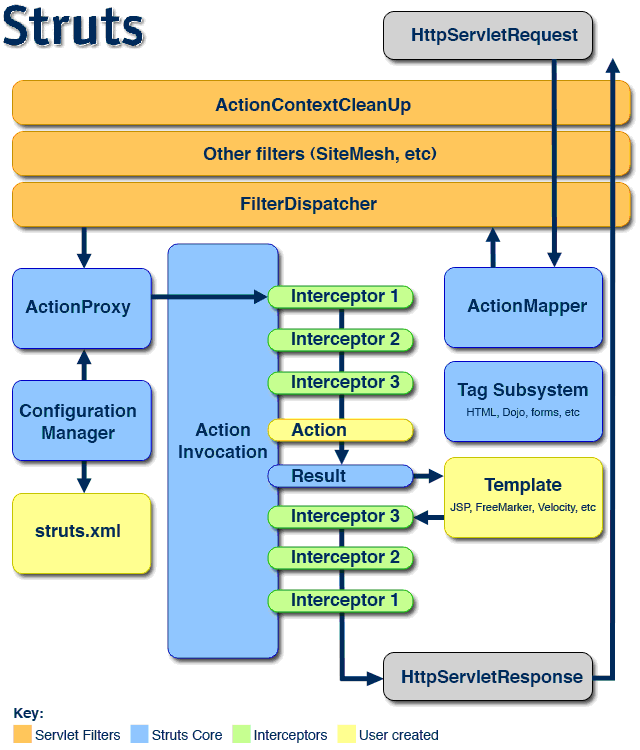
**Struts2源码阅读(一)-Struts2框架流程概述**

**1. Struts2架构图**



请求首先通过Filter chain，Filter主要包括ActionContextCleanUp，它主要清理当前线程的ActionContext和 Dispatcher；FilterDispatcher主要通过AcionMapper来决定需要调用哪个Action。 ActionMapper取得了ActionMapping后，在Dispatcher的serviceAction方法里创建 ActionProxy，ActionProxy创建ActionInvocation，然后ActionInvocation调用 Interceptors，执行Action本身，创建Result并返回，当然，如果要在返回之前做些什么，可以实现 PreResultListener。   
  
**2. Struts2部分类介绍**   
这部分从Struts2参考文档中翻译就可以了。   
ActionMapper   
        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，它自己也有Restful的实现，具体可以参考文档的docs/actionmapper.html。   
ActionProxy&ActionInvocation   
        Action的一个代理，由ActionProxyFactory创建，它本身不包括Action实例，默认实现DefaultActionProxy是 由ActionInvocation持有Action实例。ActionProxy作用是如何取得Action，无论是本地还是远程。而 ActionInvocation的作用是如何执行Action，拦截器的功能就是在ActionInvocation中实现的。   
ConfigurationProvider&Configuration   
        ConfigurationProvider就是Struts2中配置文件的解析器，Struts2中的配置文件主要是尤其实现类 XmlConfigurationProvider及其子类StrutsXmlConfigurationProvider来解析。   
  
**3. Struts2请求流程**   
1、客户端发送请求   
2、请求先通过ActionContextCleanUp-->FilterDispatcher   
3、FilterDispatcher通过ActionMapper来决定这个Request需要调用哪个Action   
4、如果ActionMapper决定调用某个Action，FilterDispatcher把请求的处理交给ActionProxy，这儿已经转到它的Delegate--Dispatcher来执行   
5、ActionProxy根据ActionMapping和ConfigurationManager找到需要调用的Action类   
6、ActionProxy创建一个ActionInvocation的实例   
7、ActionInvocation调用真正的Action，当然这涉及到相关拦截器的调用   
8、Action执行完毕，ActionInvocation创建Result并返回，当然，如果要在返回之前做些什么，可以实现PreResultListener。添加PreResultListener可以在Interceptor中实现。

[**Struts2源码阅读(二)\_ActionContext及CleanUP Filter**](http://blog.csdn.net/wl_ldy/article/details/5948899)

/\*\*

\* The ActionContext is the context in which an {@link Action} is executed. Each context is basically a

\* container of objects an action needs for execution like the session, parameters, locale, etc. <p>

\* <p/>

\* The ActionContext is thread local which means that values stored in the ActionContext are

\* unique per thread. See the {@link ThreadLocal} class for more information. The benefit of

\* this is you don't need to worry about a user specific action context, you just get it:

\* <p/>

\* <ul><code>ActionContext context = ActionContext.getContext();</code></ul>

\* <p/>

\* Finally, because of the thread local usage you don't need to worry about making your actions thread safe.

\*

\* @author Patrick Lightbody

\* @author Bill Lynch (docs)

\*/

ActionContext 是一个执行action的上下文，每个上下文基本上就是一个集装箱的对象需要执行一个动作像session、技术参数、现场等.

这个actionContext 是一个ThreadLocal 这意味着数据存储在ActionContext

独特的每个线程中。

**1. ActionContext**

ActionContext是被存放在当前线程中的，获取ActionContext也是从ThreadLocal中获取的。所以在执行拦 截器、 action和result的过程中，由于他们都是在一个线程中按照顺序执行的，所以可以可以在任意时候在ThreadLocal中获取 ActionContext。

ActionContext包括了很多信息，比如Session、Application、Request、Locale、ValueStack等，其中 ValueStack可以解析ognl表达式，来动态后去一些值，同时可以给表达式提供对象。

ActionContext(com.opensymphony.xwork.ActionContext)是Action执行时的上下文,上下文 可以看作是一个容器 (其实我们这里的容器就是一个Map而已),它存放的是Action在执行时需要用到的对象. 一般情况, 我们的ActionContext都是通过: ActionContext context = (ActionContext) actionContext.get(); 来获取的.我们再来看看这里的actionContext对象的创建:

static ThreadLocal actionContext = new ActionContextThreadLocal();

ActionContextThreadLocal是实现ThreadLocal的一个内部类.ThreadLocal可以命名为"线程局部变 量",它为每一个使用该变量的线程都提供一个变量值的副本,使每一个线程都可以独立地改变自己的副本,而不会和其它线程的副本冲突.这样,我们 ActionContext里的属性只会在对应的当前请求线程中可见,从而保证它是线程安全的.

通过ActionContext取得HttpSession: Map session = ActionContext.getContext().getSession(); (通过Map模拟HttpServlet的对象,操作更方便)

**2. ServletActionContext**

ServletActionContext(com.opensymphony.webwork. ServletActionContext),这个类直接继承了我们上面介绍的ActionContext,它提供了直接与Servlet相关对象访问的 功能,它可以取得的对象有:

(1)javax.servlet.http.HttpServletRequest : HTTPservlet请求对象   
(2)javax.servlet.http.HttpServletResponse : HTTPservlet相应对象   
(3)javax.servlet.ServletContext : Servlet上下文信息   
(4)javax.servlet.ServletConfig : Servlet配置对象   
(5)javax.servlet.jsp.PageContext : Http页面上下文

如何从ServletActionContext里取得Servlet的相关对象:

<1>取得HttpServletRequest对象: HttpServletRequest request = ServletActionContext. getRequest();

<2>取得HttpSession对象: HttpSession session = ServletActionContext. getRequest().getSession();

**3. ServletActionContext和ActionContext联系**

ServletActionContext和ActionContext有着一些重复的功能,在我们的Action中,该如何去抉择呢?我们遵循的 原则是:如果ActionContext能够实现我们的功能,那最好就不要使用ServletActionContext,让我们的Action尽量不要 直接去访问Servlet的相关对象.

注意：在使用ActionContext时有一点要注意: 不要在Action的构造函数里使用ActionContext.getContext(), 因为这个时候ActionContext里的一些值也许没有设置,这时通过ActionContext取得的值也许是null；同 样，HttpServletRequest req = ServletActionContext.getRequest()也不要放在构造函数中，也不要直接将req作为类变量给其赋值。 至于原因，我想是因为前面讲到的static ThreadLocal actionContext = new ActionContextThreadLocal()，从这里我们可以看出ActionContext是线程安全的，而 ServletActionContext继承自ActionContext，所以ServletActionContext也线程安全，线程安全要求每 个线程都独立进行，所以req的创建也要求独立进行，所以ServletActionContext.getRequest()这句话不要放在构造函数 中，也不要直接放在类中，而应该放在每个具体的方法体中(eg：login()、queryAll()、insert()等)，这样才能保证每次产生对象 时独立的建立了一个req。

**4.ActionContextClearUp**

ActionContextClearUp其实是Defer ClearUP.作用就是延长action中属性的生命周期，包括自定义属性，以便在jsp页面中进行访问，让actionContextcleanup 过滤器来清除属性，不让action自己清除。具体看下面的代码，代码很简单：

1. public void doFilter(...){
3. ...
4. try{
5. ...
6. //继续执行所配置的chain中的Filter
7. chain.doFilter(request, response);
8. }finally{
9. //保证在所有动作执行完之后,调用cleanUp
10. ...
11. cleanUp(request);
12. }
13. }
15. protected static void cleanUp(ServletRequest req) {
17. ...
18. ActionContext.setContext(null);//清除ActionContext实例
19. Dispatcher.setInstance(null);//清除Dispatcher实例(Dispatcher主要是完成将url解析成对应的Action)
20. }

另外注明一下UtilTimerStack的push和pop是用来计算调用方法所执行的开始和结束时间，用来做性能测试的。用法如下：

1. String timerKey = "ActionContextCleanUp\_doFilter: ";
3. UtilTimerStack.setActive(true);
5. UtilTimerStack.push(timerKey);
6. //调用要测试的方法。
7. UtilTimerStack.pop(timerKey);

[**Struts2源码阅读(三)\_Dispatcher&ConfigurationProvider**](http://blog.csdn.net/wl_ldy/article/details/5948931)

首先强调一下struts2的线程程安全，在Struts2中大量采用ThreadLocal线程局部变量的方法来保证线程的安全,像Dispatcher等都是通过ThreadLocal来保存变量值,使得每个线程都有自己独立的实例变量,互不相干.

接下来就从Dispatcher开始看起，先看其构造函数：

//创建Dispatcher，此类是一个Delegate，它是真正完成根据url解析转向，读取对应Action的地方

    public Dispatcher(ServletContext servletContext, Map<String, String> initParams) {

        this.servletContext = servletContext;

        //配置在web.xml中的param参数

        this.initParams = initParams;

    }

我们再看在FilterDispatcher创建Dispatcher的:

protected Dispatcher createDispatcher(FilterConfig filterConfig) {

    Map<String, String> params = new HashMap<String, String>();

    for (Enumeration e = filterConfig.getInitParameterNames(); e.hasMoreElements();) {

        String name = (String) e.nextElement();

        String value = filterConfig.getInitParameter(name);

        params.put(name, value);

    }

都可以从FilterConfig中得到

    return new Dispatcher(filterConfig.getServletContext(), params);

}

分七步载入各种配置属性,都是通过ConfigurationProvider接口进行的,这个接口提供init(),destroy(),register()等方法.  
将各种ConfigurationProvider初始化之后将实例添加到ConfigurationManager的List里面.  
最后通过循环调用List里的这些destroy(),register()等方法实现对配置文件的属性进行注册和销毁等功能.  
下面将分析这七层功夫是怎样一步步练成的.

首先是init\_DefaultProperties()

创建Dispatcher之后,来看init()方法  
init()方法是用来Load用户配置文件,资源文件以及默认的配置文件.

主要分七步走,看下面注释

public void init() {

    if (configurationManager == null) {

    //设置ConfigurationManager的defaultFrameworkBeanName.

    //这里DEFAULT\_BEAN\_NAME为struts,这是xwork框架的内容,Framework可以是xwork,struts,webwork等

        configurationManager = new ConfigurationManager(BeanSelectionProvider.DEFAULT\_BEAN\_NAME);

    }

      //读取properties信息,默认的default.properties,

    init\_DefaultProperties(); // [1]

//读取xml配置文件

      init\_TraditionalXmlConfigurations(); // [2]

//读取用户自定义的struts.properties

      init\_LegacyStrutsProperties(); // [3]

//自定义的configProviders

      init\_CustomConfigurationProviders(); // [5]

//载入FilterDispatcher传进来的initParams

      init\_FilterInitParameters() ; // [6]

//将配置文件中的bean与具体的类映射

      init\_AliasStandardObjects() ; // [7]

//构建一个用于依赖注射的Container对象

//在这里面会循环调用上面七个ConfigurationProvider的register方法

//其中的重点就是DefaultConfiguration的#reload()方法

      Container container = init\_PreloadConfiguration();

      container.inject(this);

      init\_CheckConfigurationReloading(container);

      init\_CheckWebLogicWorkaround(container);

      if (!dispatcherListeners.isEmpty()) {

          for (DispatcherListener l : dispatcherListeners) {

              l.dispatcherInitialized(this);

          }

      }

  }

分七步载入各种配置属性,都是通过ConfigurationProvider接口进行的,这个接口提供init(),destroy(),register()等方法.  
将各种ConfigurationProvider初始化之后将实例添加到ConfigurationManager的List里面.  
最后通过循环调用List里的这些destroy(),register()等方法实现对配置文件的属性进行注册和销毁等功能.  
下面将分析这七层功夫是怎样一步步练成的.

首先是init\_DefaultProperties()

private void init\_DefaultProperties() {

    configurationManager.addConfigurationProvider(new DefaultPropertiesProvider());

}

接来看DefaultPropertiesProvider好了,DefaultPropertiesProvider实际上只是实现了register()方法

public void register(ContainerBuilder builder, LocatableProperties props)

        throws ConfigurationException {

    Settings defaultSettings = null;

    try {

        defaultSettings = new PropertiesSettings("org/apache/struts2/default");

    } catch (Exception e) {

        throw new ConfigurationException("Could not find or error in org/apache/struts2/default.properties", e);

    }

    loadSettings(props, defaultSettings);

}

//PropertiesSettings构造方法

    //读取org/apache/struts2/default.properties的配置信息，如果项目中需要覆盖，可以在classpath里的struts.properties里覆写

    public PropertiesSettings(String name) {

        URL settingsUrl = ClassLoaderUtils.getResource(name + ".properties", getClass());

        if (settingsUrl == null) {

            LOG.debug(name + ".properties missing");

            settings = new LocatableProperties();

            return;

        }

        settings = new LocatableProperties(new LocationImpl(null, settingsUrl.toString()));

        // Load settings

        InputStream in = null;

        try {

            in = settingsUrl.openStream();

            settings.load(in);

        } catch (IOException e) {

            throw new StrutsException("Could not load " + name + ".properties:" + e, e);

        } finally {

            if(in != null) {

                try {

                    in.close();

                } catch(IOException io) {

                    LOG.warn("Unable to close input stream", io);

                }

            }

        }

    }

    //loadSettings主要是将progerty的value和Locale从上面PropertiesSettings中取得并存放到LocatableProperties props

    //这个props是register的一个入参.

    protected void loadSettings(LocatableProperties props, final Settings settings) {

        // We are calling the impl methods to get around the single instance of Settings that is expected

        for (Iterator i = settings.listImpl(); i.hasNext(); ) {

            String name = (String) i.next();

            props.setProperty(name, settings.getImpl(name), settings.getLocationImpl(name));

        }

    }

再来看第二步：init\_TraditionalXmlConfigurations()

private void init\_TraditionalXmlConfigurations() {

 //首先读取web.xml中的config初始参数值

    //如果没有配置就使用默认的DEFAULT\_CONFIGURATION\_PATHS:"struts-default.xml,struts-plugin.xml,struts.xml"，

    //这儿就可以看出为什么默认的配置文件必须取名为这三个名称了

    //如果不想使用默认的名称，直接在web.xml中配置config初始参数即可

    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)) {

    //XmlConfigurationProvider负责解析xwork.xml

                configurationManager.addConfigurationProvider(new XmlConfigurationProvider(file, false));

            } else {

    //其它xml都是由StrutsXmlConfigurationProvider来解析

                configurationManager.addConfigurationProvider(new StrutsXmlConfigurationProvider(file, false, servletContext));

            }

        } else {

            throw new IllegalArgumentException("Invalid configuration file name");

        }

    }

}

对于其它配置文件只用StrutsXmlConfigurationProvider，此类继承XmlConfigurationProvider，而XmlConfigurationProvider又实现ConfigurationProvider接口。  
类XmlConfigurationProvider负责配置文件的读取和解析，  
首先通过init()中的loadDocuments(configFileName);利用DomHelper中的  
public static Document parse(InputSource inputSource, Map<String, String> dtdMappings) 将configFileName配置文件通过SAX解析方式按照DtdMappings解析成Document对象.  
然后通过Provider的register()方法加载"bean"和"constant"属性,再通过loadPackages()加载package及package中的属性  
addAction()方法负责读取<action>标签，并将数据保存在ActionConfig中；  
addResultTypes()方法负责将<result-type>标签转化为ResultTypeConfig对象；  
loadInterceptors()方法负责将<interceptor>标签转化为InterceptorConfi对象；  
loadInterceptorStack()方法负责将<interceptor-ref>标签转化为InterceptorStackConfig对象；  
loadInterceptorStacks()方法负责将<interceptor-stack>标签转化成InterceptorStackConfig对象。  
而上面的方法最终会被addPackage()方法调用,addPackage又会被Provider的loadPackages()调用，将所读取到的数据汇集到PackageConfig对象中。

protected PackageConfig addPackage(Element packageElement) throws ConfigurationException {

     PackageConfig.Builder newPackage = buildPackageContext(packageElement);

     if (newPackage.isNeedsRefresh()) {

         return newPackage.build();

     }

     // add result types (and default result) to this package

     addResultTypes(newPackage, packageElement);

     // load the interceptors and interceptor stacks for this package

     loadInterceptors(newPackage, packageElement);

     // load the default interceptor reference for this package

     loadDefaultInterceptorRef(newPackage, packageElement);

     // load the default class ref for this package

     loadDefaultClassRef(newPackage, packageElement);

     // load the global result list for this package

     loadGlobalResults(newPackage, packageElement);

     // load the global exception handler list for this package

     loadGobalExceptionMappings(newPackage, packageElement);

     // get actions

     NodeList actionList = packageElement.getElementsByTagName("action");

     for (int i = 0; i < actionList.getLength(); i++) {

         Element actionElement = (Element) actionList.item(i);

         addAction(actionElement, newPackage);

     }

     // load the default action reference for this package

     loadDefaultActionRef(newPackage, packageElement);

     PackageConfig cfg = newPackage.build();

     configuration.addPackageConfig(cfg.getName(), cfg);

     return cfg;

 }

loadConfigurationFiles解析读取xml中的内容

 private List<Document> loadConfigurationFiles(String fileName, Element includeElement) {

   ...

//通过DomHelper调用SAX进行解析xml

doc = DomHelper.parse(in, dtdMappings);

...

   Element rootElement = doc.getDocumentElement();

   NodeList children = rootElement.getChildNodes();

   int childSize = children.getLength();

   for (int i = 0; i < childSize; i++) {

     Node childNode = children.item(i);

     if (childNode instanceof Element) {

       Element child = (Element) childNode;

       final String nodeName = child.getNodeName();

       if ("include".equals(nodeName)) {

         String includeFileName = child.getAttribute("file");

      //解析每个action配置是，对于include文件可以使用通配符\*来进行配置

         //如Struts.xml中可配置成<include file="actions\_\*.xml"/>

         if (includeFileName.indexOf('\*') != -1) {

           ClassPathFinder wildcardFinder = new ClassPathFinder();

           wildcardFinder.setPattern(includeFileName);

           Vector<String> wildcardMatches = wildcardFinder.findMatches();

           for (String match : wildcardMatches) {

       //递归Load子file中的<include/>

             docs.addAll(loadConfigurationFiles(match, child));

           }

         } else {

           docs.addAll(loadConfigurationFiles(includeFileName, child));

         }

       }

     }

   }

   docs.add(doc);

   loadedFileUrls.add(url.toString());

   ...

   return docs;

 }

接下来第三步:init\_LegacyStrutsProperties()  
调用的是调用的是LegacyPropertiesConfigurationProvider  
通过比较前面DefaultPropertiesProvider与调用的是LegacyPropertiesConfigurationProvider.  
发现DefaultPropertiesProvider继承自后者,但重写了register()方法,主要是生成PropertiesSetting的不同,前者是根据org/apache/struts2/default.properties  
后者是根据struts.properties  
我们展开register()中的Settings.getInstance(),最后是调用getDefaultInstance()

1. private static Settings getDefaultInstance() {
2. if (defaultImpl == null) {
3. // Create bootstrap implementation
4. //不带参数的DefaultSettings(),区别与DefaultPropertiesProvider中直接带default.properties参数
5. //不带参数就是默认为struts.propertes,并且加载struts.custom.properties所定义的properties文件
6. defaultImpl = new DefaultSettings();
8. // Create default implementation
9. try {
10. //STRUTS\_CONFIGURATION为:struts.configuration
11. //在struts.proterties中查找struts.configuration的值,这个值必须是org.apache.struts2.config.Configuration接口的实现类
12. //所以我有个困惑就是在下面的转换当中怎么将Configuration转换成Setting类型的...
13. //这一点先放下了,有时间再研究
14. String className = get(StrutsConstants.STRUTS\_CONFIGURATION);
16. if (!className.equals(defaultImpl.getClass().getName())) {
17. try {
18. // singleton instances shouldn't be built accessing request or session-specific context data
19. defaultImpl = (Settings) ObjectFactory.getObjectFactory().buildBean(Thread.currentThread().getContextClassLoader().loadClass(className), null);
20. } catch (Exception e) {
21. LOG.error("Settings: Could not instantiate the struts.configuration object, substituting the default implementation.", e);
22. }
23. }
24. } catch (IllegalArgumentException ex) {
25. // ignore
26. }

在2.1.6中去掉了第四步:init\_ZeroConfiguration();   
第五步是自定义的configProviders

1. private void init\_CustomConfigurationProviders() {
2. //从这里可以看到可以将自定义的Provider定义在web.xml中FilterDispatcher的param中:configProviders
3. String configProvs = initParams.get("configProviders");
4. if (configProvs != null) {
5. String[] classes = configProvs.split("//s\*[,]//s\*");
6. for (String cname : classes) {
7. try {
8. Class cls = ClassLoaderUtils.loadClass(cname, this.getClass());
9. ConfigurationProvider prov = (ConfigurationProvider)cls.newInstance();
10. configurationManager.addConfigurationProvider(prov);
11. }
12. ...
13. }
14. }
15. }

第六步:init\_FilterInitParameters

1. //从这里可以看出struts.properties中的属性不仅可以在struts.xml中以constant形式定义,而且可以在FilterDispatcher的param中定义
2. private void init\_FilterInitParameters() {
3. configurationManager.addConfigurationProvider(new ConfigurationProvider() {
4. public void destroy() {}
5. public void init(Configuration configuration) throws ConfigurationException {}
6. public void loadPackages() throws ConfigurationException {}
7. public boolean needsReload() { return false; }
9. public void register(ContainerBuilder builder, LocatableProperties props) throws ConfigurationException {
10. props.putAll(initParams);//在这里实现滴~
11. }
12. });
13. }

第七步:init\_AliasStandardObjects,使用BeanSelectionProvider  
这是将配置文件中定义的<bean>与实际的类相映射,就是注入bean的依赖关系,这部分以后有时候再研究Container  
   
接下来是看怎样调用这些ConfigurationProviders  
展开init\_PreloadConfiguration()

1. private Container init\_PreloadConfiguration() {
2. Configuration config = configurationManager.getConfiguration();
3. Container container = config.getContainer();
5. boolean reloadi18n = Boolean.valueOf(container.getInstance(String.class, StrutsConstants.STRUTS\_I18N\_RELOAD));
6. LocalizedTextUtil.setReloadBundles(reloadi18n);
8. return container;
9. }
10. //再看getConfiguration()
11. public synchronized Configuration getConfiguration() {
12. if (configuration == null) {
13. setConfiguration(new DefaultConfiguration(defaultFrameworkBeanName));
14. try {
15. //重点就是这个reloadContainer
16. configuration.reloadContainer(getContainerProviders());
17. } catch (ConfigurationException e) {
18. setConfiguration(null);
19. throw new ConfigurationException("Unable to load configuration.", e);
20. }
21. } else {
22. conditionalReload();
23. }
25. return configuration;
26. }

展开DefaultConfiguration中的reloadContainer

1. public synchronized List<PackageProvider> reloadContainer(List<ContainerProvider> providers) throws ConfigurationException {
2. packageContexts.clear();
3. loadedFileNames.clear();
4. List<PackageProvider> packageProviders = new ArrayList<PackageProvider>();
6. //Struts2(xwork2)用Container来完成依赖注入的功能
7. //首先初始化一个ContainerBuilder,再由builder来保存接口与实现类或工厂类的对应关系
8. //然后通过builder.create(boolean)方法产生container
9. //由container.getInstance(Class);就可以得到接口的实现实例了
10. //这一部分比较复杂,后面研究完成了,会单独拿出来讲,这里先弄清楚Xwork依赖注入的实现步骤就可以了
11. ContainerProperties props = new ContainerProperties();
12. ContainerBuilder builder = new ContainerBuilder();
13. for (final ContainerProvider containerProvider : providers)
14. {
15. //循环调用ConfigurationProvider的init和register方法,明白了吧,在这里统一循环调用
16. containerProvider.init(this);
17. containerProvider.register(builder, props);
18. }
19. props.setConstants(builder);
20. //注入依赖关系,在这里并不产生实例
21. builder.factory(Configuration.class, new Factory<Configuration>() {
22. public Configuration create(Context context) throws Exception {
23. return DefaultConfiguration.this;
24. }
25. });
27. ActionContext oldContext = ActionContext.getContext();
28. try {
29. // Set the bootstrap container for the purposes of factory creation
30. Container bootstrap = createBootstrapContainer();
31. setContext(bootstrap);
32. //create已经注入依赖关系的Container
33. container = builder.create(false);
34. setContext(container);
35. objectFactory = container.getInstance(ObjectFactory.class);
37. // Process the configuration providers first
38. for (final ContainerProvider containerProvider : providers)
39. {
40. if (containerProvider instanceof PackageProvider) {
41. container.inject(containerProvider);
42. //调用PackageProvider的loadPackages()方法,这里主要是针对XmlConfigurationProvider和StrutsXmlConfigurationProvider
43. ((PackageProvider)containerProvider).loadPackages();
44. packageProviders.add((PackageProvider)containerProvider);
45. }
46. }
48. // Then process any package providers from the plugins
49. Set<String> packageProviderNames = container.getInstanceNames(PackageProvider.class);
50. if (packageProviderNames != null) {
51. for (String name : packageProviderNames) {
52. PackageProvider provider = container.getInstance(PackageProvider.class, name);
53. provider.init(this);
54. provider.loadPackages();
55. packageProviders.add(provider);
56. }
57. }
59. rebuildRuntimeConfiguration();
60. } finally {
61. if (oldContext == null) {
62. ActionContext.setContext(null);
63. }
64. }
65. return packageProviders;
66. }

[**Struts2源码阅读(五)\_FilterDispatcher核心控制器**](http://blog.csdn.net/wl_ldy/article/details/5948992)

Dispatcher已经在之前讲过，这就好办了。FilterDispatcher是Struts2的核心控制器，首先看一下init()方法。

1. public void init(FilterConfig filterConfig) throws ServletException {
2. try {
3. this.filterConfig = filterConfig;
4. initLogging();
5. //创建dispatcher,前面都已经讲过啰
6. dispatcher = createDispatcher(filterConfig);
7. dispatcher.init();
8. //注入将FilterDispatcher中的变量通过container注入,如下面的staticResourceLoader
9. dispatcher.getContainer().inject(this);
10. //StaticContentLoader在BeanSelectionProvider中已经被注入了依赖关系:DefaultStaticContentLoader
11. //可以在struts-default.xml中的<bean>可以找到
12. staticResourceLoader.setHostConfig(new FilterHostConfig(filterConfig));
13. } finally {
14. ActionContext.setContext(null);
15. }
16. }
17. //下面来看DefaultStaticContentLoader的setHostConfig
18. public void setHostConfig(HostConfig filterConfig) {
19. //读取初始参数pakages，调用parse()，解析成类似/org/apache/struts2/static,/template的数组
20. String param = filterConfig.getInitParameter("packages");
21. //"org.apache.struts2.static template org.apache.struts2.interceptor.debugging static"
22. String packages = getAdditionalPackages();
23. if (param != null) {
24. packages = param + " " + packages;
25. }
26. this.pathPrefixes = parse(packages);
27. initLogging(filterConfig);
28. }

现在回去doFilter的方法,每当有一个Request,都会调用这些Filters的doFilter方法

1. public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException, ServletException {
3. HttpServletRequest request = (HttpServletRequest) req;
4. HttpServletResponse response = (HttpServletResponse) res;
5. ServletContext servletContext = getServletContext();
7. String timerKey = "FilterDispatcher\_doFilter: ";
8. try {
10. // FIXME: this should be refactored better to not duplicate work with the action invocation
11. //先看看ValueStackFactory所注入的实现类OgnlValueStackFactory
12. //new OgnlValueStack
13. ValueStack stack = dispatcher.getContainer().getInstance(ValueStackFactory.class).createValueStack();
14. ActionContext ctx = new ActionContext(stack.getContext());
15. ActionContext.setContext(ctx);
17. UtilTimerStack.push(timerKey);
19. //如果是multipart/form-data就用MultiPartRequestWrapper进行包装
20. //MultiPartRequestWrapper是StrutsRequestWrapper的子类，两者都是HttpServletRequest实现
21. //此时在MultiPartRequestWrapper中就会把Files给解析出来,用于文件上传
22. //所有request都会StrutsRequestWrapper进行包装,StrutsRequestWrapper是可以访问ValueStack
23. //下面是参见Dispatcher的wrapRequest
24. // String content\_type = request.getContentType();
25. //if(content\_type!= null&&content\_type.indexOf("multipart/form-data")!=-1){
26. //MultiPartRequest multi =getContainer().getInstance(MultiPartRequest.class);
27. //request =new MultiPartRequestWrapper(multi,request,getSaveDir(servletContext));
28. //} else {
29. //     request = new StrutsRequestWrapper(request);
30. // }
32. request = prepareDispatcherAndWrapRequest(request, response);
33. ActionMapping mapping;
34. try {
35. //根据url取得对应的Action的配置信息
36. //看一下注入的DefaultActionMapper的getMapping()方法.Action的配置信息存储在 ActionMapping对象中
37. mapping = actionMapper.getMapping(request, dispatcher.getConfigurationManager());
38. } catch (Exception ex) {
39. log.error("error getting ActionMapping", ex);
40. dispatcher.sendError(request, response, servletContext, HttpServletResponse.SC\_INTERNAL\_SERVER\_ERROR, ex);
41. return;
42. }
44. //如果找不到对应的action配置，则直接返回。比如你输入\*\*\*.jsp等等
45. //这儿有个例外，就是如果path是以“/struts”开头，则到初始参数packages配置的包路径去查找对应的静态资源并输出到页面流中，当然.class文件除外。如果再没有则跳转到404
46. if (mapping == null) {
47. // there is no action in this request, should we look for a static resource?
48. String resourcePath = RequestUtils.getServletPath(request);
50. if ("".equals(resourcePath) && null != request.getPathInfo()) {
51. resourcePath = request.getPathInfo();
52. }
54. if (staticResourceLoader.canHandle(resourcePath)) {
55. // 在 DefaultStaticContentLoader 中:return serveStatic && (resourcePath.startsWith("/struts") || resourcePath.startsWith("/static"));
56. staticResourceLoader.findStaticResource(resourcePath, request, response);
57. } else {
58. // this is a normal request, let it pass through
59. chain.doFilter(request, response);
60. }
61. // The framework did its job here
62. return;
63. }
64. //正式开始Action的方法
65. dispatcher.serviceAction(request, response, servletContext, mapping);
67. } finally {
68. try {
69. ActionContextCleanUp.cleanUp(req);
70. } finally {
71. UtilTimerStack.pop(timerKey);
72. }
73. }
74. }
75. //下面是ActionMapper接口的实现类 DefaultActionMapper的getMapping()方法的源代码:
76. public ActionMapping getMapping(HttpServletRequest request,
77. ConfigurationManager configManager) {
78. ActionMapping mapping = new ActionMapping();
79. String uri = getUri(request);//得到请求路径的URI，如：testAtcion.action或testAction.do

82. int indexOfSemicolon = uri.indexOf(";");//修正url的带;jsessionid 时找不到而且的bug
83. uri = (indexOfSemicolon > -1) ? uri.substring(0, indexOfSemicolon) : uri;
85. uri = dropExtension(uri, mapping);//删除扩展名，默认扩展名为action
86. if (uri == null) {
87. return null;
88. }
90. parseNameAndNamespace(uri, mapping, configManager);//匹配Action的name和namespace
92. handleSpecialParameters(request, mapping);//去掉重复参数
94. //如果Action的name没有解析出来，直接返回
95. if (mapping.getName() == null) {
96. returnnull;
97. }
98. //下面处理形如testAction!method格式的请求路径
99. if (allowDynamicMethodCalls) {
100. // handle "name!method" convention.
101. String name = mapping.getName();
102. int exclamation = name.lastIndexOf("!");//!是Action名称和方法名的分隔符
103. if (exclamation != -1) {
104. mapping.setName(name.substring(0, exclamation));//提取左边为name
105. mapping.setMethod(name.substring(exclamation + 1));//提取右边的method
106. }
107. }
109. return mapping;
110. }

从代码中看出，getMapping()方法返回ActionMapping类型的对象，该对象包含三个参数：Action的name、namespace和要调用的方法method。  
　 　如果getMapping()方法返回ActionMapping对象为null，则FilterDispatcher认为用户请求不是Action， 自然另当别论，FilterDispatcher会做一件非常有意思的事：如果请求以/struts开头，会自动查找在web.xml文件中配置的 packages初始化参数，就像下面这样:

1. <filter>
2. <filter-name>struts2</filter-name>
3. <filter-class>
4. org.apache.struts2.dispatcher.FilterDispatcher
5. </filter-class>
6. <init-param>
7. <param-name>packages</param-name>
8. <param-value>com.lizanhong.action</param-value>
9. </init-param>
10. </filter>

FilterDispatcher会将com.lizanhong.action包下的文件当作静态资源处理，即直接在页面上显示文件内容，不过会 忽略扩展名为class的文件。比如在com.lizanhong.action包下有一个aaa.txt的文本文件，其内容为“中华人民共和国”，访问 <http://localhost:8081/Struts2Demo/struts/aaa.txt>时会输出txt中的内容  
   FilterDispatcher.findStaticResource()方法

1. protectedvoid findStaticResource(String name, HttpServletRequest request, HttpServletResponse response) throws IOException {
2. if (!name.endsWith(".class")) {//忽略class文件
3. //遍历packages参数
4. for (String pathPrefix : pathPrefixes) {
5. InputStream is = findInputStream(name, pathPrefix);//读取请求文件流
6. if (is != null) {
7. ...
8. // set the content-type header
9. String contentType = getContentType(name);//读取内容类型
10. if (contentType != null) {
11. response.setContentType(contentType);//重新设置内容类型
12. }
13. ...
14. try {
15. //将读取到的文件流以每次复制4096个字节的方式循环输出
16. copy(is, response.getOutputStream());
17. } finally {
18. is.close();
19. }
20. return;
21. }
22. }
23. }
24. }

如果用户请求的资源不是以/struts开头——可能是.jsp文件，也可能是.html文件，则通过过滤器链继续往下传送，直到到达请求的资源为止。  
如 果getMapping()方法返回有效的ActionMapping对象，则被认为正在请求某个Action，将调用 Dispatcher.serviceAction(request, response, servletContext, mapping)方法，该方法是处理Action的关键所在。  
下面就来看serviceAction,这又回到全局变量dispatcher中了

1. //Load Action class for mapping and invoke the appropriate Action method, or go directly to the Result.
2. public void serviceAction(HttpServletRequest request, HttpServletResponse response, ServletContext context,
3. ActionMapping mapping) throws ServletException {
4. //createContextMap方法主要把Application、Session、Request的key value值拷贝到Map中
5. Map<String, Object> extraContext = createContextMap(request, response, mapping, context);
7. // If there was a previous value stack, then create a new copy and pass it in to be used by the new Action
8. ValueStack stack = (ValueStack) request.getAttribute(ServletActionContext.STRUTS\_VALUESTACK\_KEY);
9. boolean nullStack = stack == null;
10. if (nullStack) {
11. ActionContext ctx = ActionContext.getContext();
12. if (ctx != null) {
13. stack = ctx.getValueStack();
14. }
15. }
16. if (stack != null) {
17. extraContext.put(ActionContext.VALUE\_STACK, valueStackFactory.createValueStack(stack));
18. }
20. String timerKey = "Handling request from Dispatcher";
21. try {
22. UtilTimerStack.push(timerKey);
23. String namespace = mapping.getNamespace();
24. String name = mapping.getName();
25. String method = mapping.getMethod();
27. Configuration config = configurationManager.getConfiguration();
28. //创建一个Action的代理对象，ActionProxyFactory是创建ActionProxy的工厂
29. //参考实现类:DefaultActionProxy和DefaultActionProxyFactory
30. ActionProxy proxy = config.getContainer().getInstance(ActionProxyFactory.class).createActionProxy(
31. namespace, name, method, extraContext, true, false);
33. request.setAttribute(ServletActionContext.STRUTS\_VALUESTACK\_KEY, proxy.getInvocation().getStack());
35. // if the ActionMapping says to go straight to a result, do it!
36. //如果是Result,则直接转向,关于Result,ActionProxy,ActionInvocation下一讲中再分析
37. if (mapping.getResult() != null) {
38. Result result = mapping.getResult();
39. result.execute(proxy.getInvocation());
40. } else {
41. //执行Action
42. proxy.execute();
43. }
45. // If there was a previous value stack then set it back onto the request
46. if (!nullStack) {
47. request.setAttribute(ServletActionContext.STRUTS\_VALUESTACK\_KEY, stack);
48. }
49. } catch (ConfigurationException e) {
50. // WW-2874 Only log error if in devMode
51. if(devMode) {
52. LOG.error("Could not find action or result", e);
53. }
54. else {
55. LOG.warn("Could not find action or result", e);
56. }
57. sendError(request, response, context, HttpServletResponse.SC\_NOT\_FOUND, e);
58. } catch (Exception e) {
59. sendError(request, response, context, HttpServletResponse.SC\_INTERNAL\_SERVER\_ERROR, e);
60. } finally {
61. UtilTimerStack.pop(timerKey);
62. }
63. }

### [Struts2源码阅读(六)\_ActionProxy&ActionInvocation](http://blog.csdn.net/wl_ldy/article/details/5949006)

下面开始讲一下主菜ActionProxy了.在这之前最好先去了解一下动态Proxy的基本知识.  
ActionProxy是Action 的一个代理类，也就是说Action的调用是通过ActionProxy实现的，其实就是调用了ActionProxy.execute()方法，而该方 法又调用了ActionInvocation.invoke()方法。归根到底，最后调用的是 DefaultActionInvocation.invokeAction()方法。  
DefaultActionInvocation()->init()->createAction()。   
最 后通过调用 ActionProxy.exute()-->ActionInvocation.invoke()-->Intercepter.intercept()-->ActionInvocation.invokeActionOnly()-->invokeAction()  
这里的步骤是先由ActionProxyFactory创建ActionInvocation和ActionProxy.

1. public ActionProxy createActionProxy(String namespace, String actionName, String methodName, Map<String, Object> extraContext, boolean executeResult, boolean cleanupContext) {
3. ActionInvocation inv = new DefaultActionInvocation(extraContext, true);
4. container.inject(inv);
5. return createActionProxy(inv, namespace, actionName, methodName, executeResult, cleanupContext);
6. }

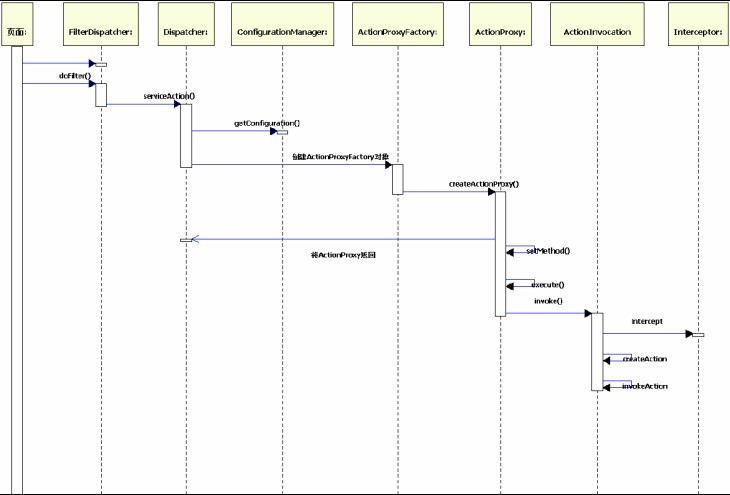
 下面先看DefaultActionInvocation的init方法

1. public void init(ActionProxy proxy) {
2. this.proxy = proxy;
3. Map<String, Object> contextMap = createContextMap();
5. // Setting this so that other classes, like object factories, can use the ActionProxy and other
6. // contextual information to operate
7. ActionContext actionContext = ActionContext.getContext();
9. if (actionContext != null) {
10. actionContext.setActionInvocation(this);
11. }
12. //创建Action,struts2中每一个Request都会创建一个新的Action
13. createAction(contextMap);
15. if (pushAction) {
16. stack.push(action);
17. contextMap.put("action", action);
18. }
20. invocationContext = new ActionContext(contextMap);
21. invocationContext.setName(proxy.getActionName());
23. // get a new List so we don't get problems with the iterator if someone changes the list
24. List<InterceptorMapping> interceptorList = new ArrayList<InterceptorMapping>(proxy.getConfig().getInterceptors());
25. interceptors = interceptorList.iterator();
26. }
28. protected void createAction(Map<String, Object> contextMap) {
29. // load action
30. String timerKey = "actionCreate: " + proxy.getActionName();
31. try {
32. UtilTimerStack.push(timerKey);
33. //默认为SpringObjectFactory:struts.objectFactory=spring.这里非常巧妙,在struts.properties中可以重写这个属性
34. //在前面BeanSelectionProvider中通过配置文件为ObjectFactory设置实现类
35. //这里以Spring为例,这里会调到SpringObjectFactory的buildBean方法,可以通过ApplicationContext的getBean()方法得到Spring的Bean
36. action = objectFactory.buildAction(proxy.getActionName(), proxy.getNamespace(), proxy.getConfig(), contextMap);
37. } catch (InstantiationException e) {
38. throw new XWorkException("Unable to intantiate Action!", e, proxy.getConfig());
39. } catch (IllegalAccessException e) {
40. throw new XWorkException("Illegal access to constructor, is it public?", e, proxy.getConfig());
41. } catch (Exception e) {
42. ...
43. } finally {
44. UtilTimerStack.pop(timerKey);
45. }
47. if (actionEventListener != null) {
48. action = actionEventListener.prepare(action, stack);
49. }
50. }
51. //SpringObjectFactory
52. public Object buildBean(String beanName, Map<String, Object> extraContext, boolean injectInternal) throws Exception {
53. Object o = null;
54. try {
55. //SpringObjectFactory会通过web.xml中的context-param:contextConfigLocation自动注入ClassPathXmlApplicationContext
56. o = appContext.getBean(beanName);
57. } catch (NoSuchBeanDefinitionException e) {
58. Class beanClazz = getClassInstance(beanName);
59. o = buildBean(beanClazz, extraContext);
60. }
61. if (injectInternal) {
62. injectInternalBeans(o);
63. }
64. return o;
65. }
66. //接下来看看DefaultActionInvocation 的invoke方法
67. public String invoke() throws Exception {
68. String profileKey = "invoke: ";
69. try {
70. UtilTimerStack.push(profileKey);
72. if (executed) {
73. throw new IllegalStateException("Action has already executed");
74. }
75. //递归执行interceptor
76. if (interceptors.hasNext()) {
77. //interceptors是InterceptorMapping实际上是像一个像FilterChain一样的Interceptor链
78. //通过调用Invocation.invoke()实现递归牡循环
79. final InterceptorMapping interceptor = (InterceptorMapping) interceptors.next();
80. String interceptorMsg = "interceptor: " + interceptor.getName();
81. UtilTimerStack.push(interceptorMsg);
82. try {
83. //在每个Interceptor的方法中都会return invocation.invoke()
84. resultCode = interceptor.getInterceptor().intercept(DefaultActionInvocation.this);
85. }
86. finally {
87. UtilTimerStack.pop(interceptorMsg);
88. }
89. } else {
90. //当所有interceptor都执行完,最后执行Action,invokeActionOnly会调用invokeAction()方法
91. resultCode = invokeActionOnly();
92. }
94. // this is needed because the result will be executed, then control will return to the Interceptor, which will
95. // return above and flow through again
96. //在Result返回之前调用preResultListeners
97. //通过executed控制,只执行一次
98. if (!executed) {
99. if (preResultListeners != null) {
100. for (Object preResultListener : preResultListeners) {
101. PreResultListener listener = (PreResultListener) preResultListener;
103. String \_profileKey = "preResultListener: ";
104. try {
105. UtilTimerStack.push(\_profileKey);
106. listener.beforeResult(this, resultCode);
107. }
108. finally {
109. UtilTimerStack.pop(\_profileKey);
110. }
111. }
112. }
114. // now execute the result, if we're supposed to
115. //执行Result
116. if (proxy.getExecuteResult()) {
117. executeResult();
118. }
120. executed = true;
121. }
123. return resultCode;
124. }
125. finally {
126. UtilTimerStack.pop(profileKey);
127. }
128. }
130. //invokeAction
131. protected String invokeAction(Object action,ActionConfig actionConfig)throws Exception{
132. String methodName = proxy.getMethod();
134. String timerKey = "invokeAction: " + proxy.getActionName();
135. try {
136. UtilTimerStack.push(timerKey);
138. boolean methodCalled = false;
139. Object methodResult = null;
140. Method method = null;
141. try {
142. //java反射机制得到要执行的方法
143. method = getAction().getClass().getMethod(methodName, new Class[0]);
144. } catch (NoSuchMethodException e) {
145. // hmm -- OK, try doXxx instead
146. //如果没有对应的方法，则使用do+Xxxx来再次获得方法
147. try {
148. String altMethodName = "do" + methodName.substring(0, 1).toUpperCase() + methodName.substring(1);
149. method = getAction().getClass().getMethod(altMethodName, new Class[0]);
150. } catch (NoSuchMethodException e1) {
151. // well, give the unknown handler a shot
152. if (unknownHandlerManager.hasUnknownHandlers()) {
153. try {
154. methodResult = unknownHandlerManager.handleUnknownMethod(action, methodName);
155. methodCalled = true;
156. } catch (NoSuchMethodException e2) {
157. // throw the original one
158. throw e;
159. }
160. } else {
161. throw e;
162. }
163. }
164. }
165. //执行Method
166. if (!methodCalled) {
167. methodResult = method.invoke(action, new Object[0]);
168. }
169. //从这里可以看出可以Action的方法可以返回String去匹配Result,也可以直接返回Result类
170. if (methodResult instanceof Result) {
171. this.explicitResult = (Result) methodResult;
173. // Wire the result automatically
174. container.inject(explicitResult);
175. return null;
176. } else {
177. return (String) methodResult;
178. }
179. } catch (NoSuchMethodException e) {
180. throw new IllegalArgumentException("The " + methodName + "() is not defined in action " + getAction().getClass() + "");
181. } catch (InvocationTargetException e) {
182. // We try to return the source exception.
183. Throwable t = e.getTargetException();
185. if (actionEventListener != null) {
186. String result = actionEventListener.handleException(t, getStack());
187. if (result != null) {
188. return result;
189. }
190. }
191. if (t instanceof Exception) {
192. throw (Exception) t;
193. } else {
194. throw e;
195. }
196. } finally {
197. UtilTimerStack.pop(timerKey);
198. }
199. }

action执行完了，还要根据ResultConfig返回到view，也就是在invoke方法中调用executeResult方法。

1. private void executeResult() throws Exception {
2. //根据ResultConfig创建Result
3. result = createResult();
5. String timerKey = "executeResult: " + getResultCode();
6. try {
7. UtilTimerStack.push(timerKey);
8. if (result != null) {
9. //开始执行Result,
10. //可以参考Result的实现，如用了比较多的ServletDispatcherResult,ServletActionRedirectResult,ServletRedirectResult
11. result.execute(this);
12. } else if (resultCode != null && !Action.NONE.equals(resultCode)) {
13. throw new ConfigurationException("No result defined for action " + getAction().getClass().getName()
14. + " and result " + getResultCode(), proxy.getConfig());
15. } else {
16. if (LOG.isDebugEnabled()) {
17. LOG.debug("No result returned for action " + getAction().getClass().getName() + " at " + proxy.getConfig().getLocation());
18. }
19. }
20. } finally {
21. UtilTimerStack.pop(timerKey);
22. }
23. }
25. public Result createResult() throws Exception {
26. //如果Action中直接返回的Result类型,在invokeAction()保存在explicitResult
27. if (explicitResult != null) {
28. Result ret = explicitResult;
29. explicitResult = null;
31. return ret;
32. }
33. //返回的是String则从config中得到当前Action的Results列表
34. ActionConfig config = proxy.getConfig();
35. Map<String, ResultConfig> results = config.getResults();
37. ResultConfig resultConfig = null;
39. synchronized (config) {
40. try {
41. //通过返回的String来匹配resultConfig
42. resultConfig = results.get(resultCode);
43. } catch (NullPointerException e) {
44. // swallow
45. }
46. if (resultConfig == null) {
47. // If no result is found for the given resultCode, try to get a wildcard '\*' match.
48. //如果找不到对应name的ResultConfig，则使用name为\*的Result
49. //说明可以用\*通配所有的Result
50. resultConfig = results.get("\*");
51. }
52. }
54. if (resultConfig != null) {
55. try {
56. //创建Result
57. return objectFactory.buildResult(resultConfig, invocationContext.getContextMap());
58. } catch (Exception e) {
59. LOG.error("There was an exception while instantiating the result of type " + resultConfig.getClassName(), e);
60. throw new XWorkException(e, resultConfig);
61. }
62. } else if (resultCode != null && !Action.NONE.equals(resultCode) && unknownHandlerManager.hasUnknownHandlers()) {
63. return unknownHandlerManager.handleUnknownResult(invocationContext, proxy.getActionName(), proxy.getConfig(), resultCode);
64. }
65. return null;
66. }
68. public Result buildResult(ResultConfig resultConfig, Map<String, Object> extraContext) throws Exception {
69. String resultClassName = resultConfig.getClassName();
70. Result result = null;
72. if (resultClassName != null) {
73. //buildBean中会用反射机制Class.newInstance来创建bean
74. result = (Result) buildBean(resultClassName, extraContext);
75. Map<String, String> params = resultConfig.getParams();
76. if (params != null) {
77. for (Map.Entry<String, String> paramEntry : params.entrySet()) {
78. try {
79. //reflectionProvider参见OgnlReflectionProvider；
80. //resultConfig.getParams()就是result配置文件里所配置的参数<param></param>
81. //setProperties方法最终调用的是Ognl类的setValue方法
82. //这句其实就是把param名值设置到根对象result上
83. reflectionProvider.setProperty(paramEntry.getKey(), paramEntry.getValue(), result, extraContext, true);
84. } catch (ReflectionException ex) {
85. if (LOG.isErrorEnabled())
86. LOG.error("Unable to set parameter [#0] in result of type [#1]", ex,
87. paramEntry.getKey(), resultConfig.getClassName());
88. if (result instanceof ReflectionExceptionHandler) {
89. ((ReflectionExceptionHandler) result).handle(ex);
90. }
91. }
92. }
93. }
94. }
96. return result;
97. }

最后看一张在网上看到的一个调用流程图作为参考：



ActionMapper Interface:

提供HTTP请求和action行为之间的映射关糸。当给的request，如果没有调用action时，ActionMapper返回值 为NULL时，否则返回一个描绘action行为的ActionMapper。