**http://flym.iteye.com/blog/417434**

[**Log4j启动过程**](http://flym.iteye.com/blog/417434)

**博客分类：**

* [java](http://flym.iteye.com/category/108860)

[log4j](http://www.iteye.com/blogs/tag/log4j)[配置管理](http://www.iteye.com/blogs/tag/%E9%85%8D%E7%BD%AE%E7%AE%A1%E7%90%86)[JVM](http://www.iteye.com/blogs/tag/JVM)[项目管理](http://www.iteye.com/blogs/tag/%E9%A1%B9%E7%9B%AE%E7%AE%A1%E7%90%86)[框架](http://www.iteye.com/blogs/tag/%E6%A1%86%E6%9E%B6)

    用了好久的log4j，但还是不知道Log4j究竟是基于怎样的原理来进行工作，以及为何在项目中除了Log4j之外，还需要一个common-logging来协同进行日志记录。在网上看了下相应介绍，都说common-logging是一个日志的管理框架，具体的事情还是交由log4j来进行记录。决定从源码出发，看看Log4j如何加载配置文件，并进行日志记录。

    将Log4j从网上down下来，并建立工程，将源代码导进去。从Logger入手，一般来说，取得一个Log都是通过

**Java代码  [收藏代码](javascript:void())**

1. Logger getLogger(Class clazz) {
2. **return** LogManager.getLogger(clazz.getName());
3. }

转入LogManager,首先应该注意的是这个类的static块， 这个块其实就是一个加载log4j配置文件的过程。即在程序启动之初，在JVM需要加载这个类时，这个初始化块会自动运行，并且加载整个配置，以完成log4j的启动。

**Java代码  [收藏代码](javascript:void())**

1. Hierarchy h = **new** Hierarchy(**new** RootLogger((Level) Level.DEBUG));
2. ......
3. OptionConverter.selectAndConfigure(url, configuratorClassName,
4. LogManager.getLoggerRepository());

上面的所有代码均完成两件事，第一件事就是构造一个ROOT的Logger对象，此对象作为所有logger对象的最上层，其它logger的相应属性均从这个对象进行继承或改写，就好像java里的继承一样，这个类是内定的，不能由配置文件直接指定，且root都是在最顶层的，其他logger均在此下，这样形成一个完整的logger树。下层可以引用上层，上层管理下层。    第二件事则是去寻找配置文件的地址信息，通过各种方法都寻找log4j.xml或log4j.properties文件，然后对文件进行解析。（在此处，通过properties文件进行解析）

**Java代码  [收藏代码](javascript:void())**

1. OptionConverter.**void** selectAndConfigure(URL url, String clazz, LoggerRepository hierarchy)

 这个方法会最终通过指定的文件解析类（此处是PropertyConfigurator)进行解析，转入

**Java代码  [收藏代码](javascript:void())**

1. configurator.doConfigure(url, hierarchy)

 这个方法将，url转化成一个properties对象，进行解析。

**Java代码  [收藏代码](javascript:void())**

1. doConfigure(props, hierarchy);

 进入这个方法

**Java代码  [收藏代码](javascript:void())**

1. String value = properties.getProperty(LogLog.DEBUG\_KEY);//即log4j.debug
2. **if**(value == **null**) {
3. value = properties.getProperty("log4j.configDebug");
4. **if**(value != **null**) {
5. LogLog.setInternalDebugging(OptionConverter.toBoolean(value, **true**));
6. }

  上面方法读取一个关于log4j自身的debug Level信息，主要用于log4j内部在解析时调用（因为log4j还不能使用logger对象进行写信息，它用到一个LogLog的类，来模拟logger记录，用于在自身解析的过程中输出一些信息），一般来说，这个都用不到。

    主要的信息集中在以下三句话：

**Java代码  [收藏代码](javascript:void())**

1. configureRootCategory(properties, hierarchy);
2. configureLoggerFactory(properties);
3. parseCatsAndRenderers(properties, hierarchy);

第一句话用于解析root根对象上的相关配置。

第二句话用于解析loggerFactory（factory用于创建logger对象）

第三句话用于解析除rootLogger之外的其他logger以及render信息。

第一句：

**Java代码  [收藏代码](javascript:void())**

1. **void** configureRootCategory(Properties props, LoggerRepository hierarchy) {
2. String effectiveFrefix = ROOT\_LOGGER\_PREFIX;
3. String value = OptionConverter.findAndSubst(ROOT\_LOGGER\_PREFIX, props);
4. **if**(value == **null**) {
5. value = OptionConverter.findAndSubst(ROOT\_CATEGORY\_PREFIX, props);
6. effectiveFrefix = ROOT\_CATEGORY\_PREFIX;
7. }
8. .....
9. Logger root = hierarchy.getRootLogger();
10. **synchronized**(root) {
11. parseCategory(props, root, effectiveFrefix, INTERNAL\_ROOT\_NAME, value);
12. }
13. }

上面语句用于从属性文件中寻找logger.rootLogger或logger.rootCategory属性的值然后，再进行配置rootLogger对象信息（如Level,appender等）

**Java代码  [收藏代码](javascript:void())**

1. **void** parseCategory(Properties props, Logger logger, String optionKey,
2. String loggerName, String value) {
4. StringTokenizer st = **new** StringTokenizer(value, ",");
5. **if**(!(value.startsWith(",") || value.equals(""))) {
6. **if**(!st.hasMoreTokens())
7. **return**;
9. String levelStr = st.nextToken();
10. **if**(INHERITED.equalsIgnoreCase(levelStr) ||
11. NULL.equalsIgnoreCase(levelStr)) {
12. **if**(loggerName.equals(INTERNAL\_ROOT\_NAME)) {
13. LogLog.warn("The root logger cannot be set to null.");
14. } **else** {
15. logger.setLevel(**null**);
16. }
17. } **else** {
18. logger.setLevel(OptionConverter.toLevel(levelStr, (Level) Level.DEBUG));
19. }
21. logger.removeAllAppenders();
23. Appender appender;
24. String appenderName;
25. **while**(st.hasMoreTokens()) {
26. appenderName = st.nextToken().trim();
27. ......
28. appender = parseAppender(props, appenderName);
29. **if**(appender != **null**) {
30. logger.addAppender(appender);
31. }
32. }
33. }

上面这些内容，即是通过logger属性的值（形如debug,A,R)等，然后将值以，分隔进行解析，将第一个字符串定义为logger的Level信息，后面的值则定义为此logger的appender名称。

    当然上面这个方法，即不是尽为rootLogger服务，对于其他logger也调用这个方法，故上面在解析Level时，对root作了单独判断（因为rootLogger的Level不能为空，至少均需要一个值）。处理Level，即将level值简单设置在logger上即可以了。

   接下来即解析appender信息，通过紧接着level后面的字符串，按列表方式进行解析，然后将appender加入logger的appenderList(即要进行信息处理的监听器表）中。进入parseAddpender(解析appender)

**Java代码  [收藏代码](javascript:void())**

1. String prefix = APPENDER\_PREFIX + appenderName;
2. String layoutPrefix = prefix + ".layout";
4. appender = (Appender) OptionConverter.instantiateByKey(props, prefix,
5. org.apache.log4j.Appender.**class**,
6. **null**);
7. ...
8. appender.setName(appenderName);
10. **if**(appender **instanceof** OptionHandler) {
11. **if**(appender.requiresLayout()) {
12. yout layout = (Layout) OptionConverter.instantiateByKey(props,
13. layoutPrefix,
14. Layout.**class**,
15. **null**);
16. ...
17. appender.setLayout(layout);
18. PropertySetter.setProperties(layout, props, layoutPrefix + ".");
19. LogLog.debug("End of parsing for \"" + appenderName +"\".");
21. }
22. PropertySetter.setProperties(appender, props, prefix + ".");
23. }
24. registryPut(appender);

 上面这个方法，即是解析appender对象，通过log4j.appender.X的前缀（X表示在rootLogger后的appender名称）来取得appender类名，并尝试实例化，然后根据appender来判断是否需要再解析appender的layout(即log4j.appender.X.layout这个键），解析并设置相应属性，最后分别解析appender本身的属性信息和layout的属性信息。（通过ProperSetter这个类，根据javaBean属性映射，将指定后缀后的信息当作一个键，后缀在属性文件中的值作为指定键的值，并将这个键值映射，通过javaBean设置到相应的对象上）

    至此，rootLogger即解析完毕。

   第二句：解析loggerFactory,略。

   第三句：解析其他logger信息。

**Java代码  [收藏代码](javascript:void())**

1. **void** parseCatsAndRenderers(Properties props, LoggerRepository hierarchy) {
2. Enumeration enumeration = props.propertyNames();
3. **while**(enumeration.hasMoreElements()) {
4. String key = (String) enumeration.nextElement();
5. **if**(key.startsWith(CATEGORY\_PREFIX) || key.startsWith(LOGGER\_PREFIX)) {
6. String loggerName = **null**;
7. **if**(key.startsWith(CATEGORY\_PREFIX)) {
8. loggerName = key.substring(CATEGORY\_PREFIX.length());
9. } **else** **if**(key.startsWith(LOGGER\_PREFIX)) {
10. loggerName = key.substring(LOGGER\_PREFIX.length());
11. }
12. String value =  OptionConverter.findAndSubst(key, props);
13. Logger logger = hierarchy.getLogger(loggerName, loggerFactory);
14. **synchronized**(logger) {
15. parseCategory(props, logger, key, loggerName, value);
16. parseAdditivityForLogger(props, logger, loggerName);
17. }
18. } **else** **if**(key.startsWith(RENDERER\_PREFIX)) {
19. ......
20. }
21. }
22. }

这个方法，则主要根据logger.logger为前缀的属性信息进行解析，并根据这个属性信息生成logger,并放在继承树中，设置相应树信息，最后设置其他信息（如appender,level)等，与rootLogger解析基本一致。

至此，整个log4j的配置信息已经完成，而这个配置是由JVM保证线程化的（即只能被加载一次），在使用时整个配置已经加载成功，得到的已经是从配置信息中得到的logger对象了。