## Apache Log4j里的几个核心类

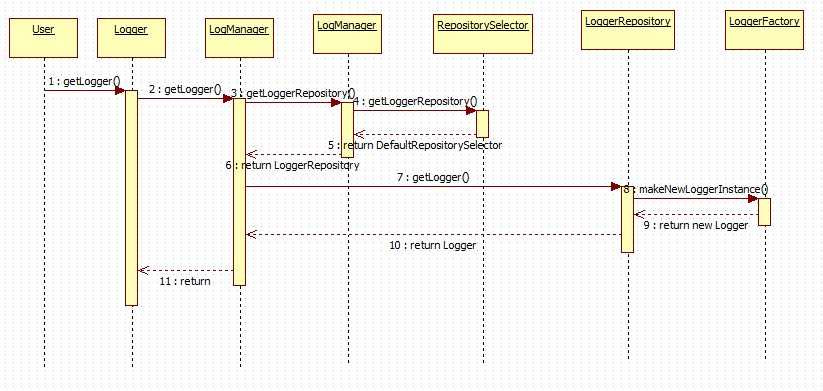
1. Logger：日志类。getLogger()交给LogManager去实现。
2. Level：八个级别。 #OFF,FATAL,ERROR,WARN,INFO,DEBUG,TRACE,ALL。
3. LogManager：Use the LogManager  class to retreive  Logger instances or to operate on the current  
    LoggerRepository。
4. RepositorySelector：该接口只有一个方法LoggerRepository getLoggerRepository()。LogManager有一个RepositorySelector对象。
5. LoggerRepository：创建和管理Logger。
6. Hierarchy：实现LoggerRepository, RendererSupport，是最重要的一个数据结构，维护着整个日志树。调用LoggerFactory产生Logger。
7. LoggerFactory：该接口只有一个方法 Logger makeNewLoggerInstance(String name)。DefaultCategoryFactory实现这个接口，最终就是这里new Logger(name)。
8. ObjectRenderer：该接口只有一个方法String doRender(Object o)。Log4J中，对传入的message实例，如果是非String类型，会先使用注册的ObjectRender（在LogRepository中查找注册的ObjectRender信息）处理成String后返回，若没有找到相应的ObjectRender，则使用默认的ObjectRender，它只是调用该消息实例的toString()方法。**比喻log4j.porperties里的log4j.renderer.com.jyz.study.jdk.logger.User=com.jyz.study.jdk.logger.UserRenderer，UserRenderer就是一个ObjectRender。**但是这个东西实际开发中很少用到。
9. Appender:该接口定义了日志的输出方式，如经常用的两个Appender

## log4j中的自定义

可以通过自定义LoggerFactory来实现创建Logger实例的逻辑,比如在自己的框架中返回一个Logger代理类,这样就可以使自己的框架和Log4j进行集成

## Logger的初始化

 Logger logger = Logger.getLogger(Log4jTest.class)将发生以下事情。



**LogManager静态代码块里：**

1. 以DEBUG等级创建一个RootLogger,然后以RootLogger为参数创建一个Hierarchy类的实例。
2. 以Hierarchy为参数创建一个DefaultRepositorySelector类的实例，就是我们上面说的“LogManager有一个RepositorySelector对象。”
3. 读取配置文件。

**正式进入LogManager.getLogger(name)**

1. 静态代码块完后LogManager.getLogger(name)由getLoggerRepository().getLogger(name)去实现。
2. getLoggerRepository()会调用上面的DefaultRepositorySelector.getLoggerRepository(),也就是Hierarchy。
3. Hierarchy.getLogger(name)最终由LoggerFactory.makeNewLoggerInstance(name)实现。

**配置文件的读取**

配置文件的读取，最终会进入PropertyConfigurator的doConfigure(Properties properties, LoggerRepository hierarchy)方法。

1. 解析log4j.reset，看hierarchy是否需要重置。
2. 解析log4j.threshold。Threshold是个全局的过滤器，它将把低于所设置的level的信息过滤不显示出来。
3. 解析log4j.rootCategory。设置日志级别和初始化所有appender。
4. 解析og4j.loggerFactory。可以自定义lLoggerFactory类，要实现LoggerFactory接口。
5. 解析log4j.category.     log4j.logger.     log4j.renderer.。

## log4j.logger.logger-name

log4j支持为某一个特定的logger实例进行配置,这个logger实例是通过创建logger时传入的name来确定的,比如说Logger.getLogger("mylogger");

我们可以通过mylogger来定位一个logger实例,进而对其进行配置,配置如下:

log4j.logger.mylogger = INFO,studout

上一段配置的意思是名称为mylogger的logger实例采集INFO以上的日志,并且输出到名称为studout的appender中

## log4j中logger的层级关系

在log4j中的logger有继承关系

log4j.rootLogger代表根logger,是所有其他logger的根logger

log4j.logger.com代表名为com的logger

log4j.logger.com.taikang代表名为com.taikang的logger

如果在程序中获取一个名为com.taikang.UserBO的logger,那么这个logger的parent为com.taikang,com.taikang的parent为com,com的parent为root

在调用UserBO的logger的info方法时,会先执行自己绑定的appender,然后调用parent的info方法,parent的info方法同样会执行自己绑定的appender.logger可以不绑定任何appender,交给parent的appender去做日志即可

## log4j中layout中的占位符

%m (message)输出代码中指定的信息

%p (priority)输出优先级,即DEBUG,INFO,WARN,ERROR,FATAL

%r 输出自应用启动到输出该log信息耗费的毫秒值

%c (category)输出所属的类目,通常就是所在类的全名

%t (thread)输出产生该日志时间的线程名

%n 输出一个回车换行符

%d (datetime)输出日志时间点的日期和时间,可以自定义格式如:%d(yyy MMM dd HH:mm:ss,SSS)

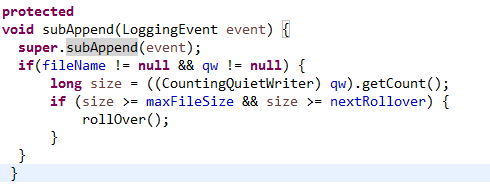
%l (location)输出日志事件的发生位置,包括类目名,发生的线程,以及在代码中的行数

## log4j中RollingFileAppender使用及源码详解

### 配置

|  |
| --- |
| #设置rootLogger的显示级别为DEBUG,Appender为rollfile  log4j.rootLogger=DEBUG,rollfile  #设置rollfile输出媒介为RollingFileAppender  log4j.appender.rollfile=org.apache.log4j.RollingFileAppender  #设置输出文件的路径  log4j.appender.rollfile.File=D:/my.log  #设置内容追加方式,true为将内容附加在原来的文件而不是覆盖原来的文件  log4j.appender.Append=true  #设置日志文件的最大数量  log4j.appender.rollfile.MaxBackupIndex=10  #设置每个日志文件的最大容量  log4j.appender.rollfile.MaxFileSize=10MB  #设置rollfile的输出级别  log4j.appender. rollfile.Threshold=DEBUG  #设置layout策略  log4j.appender. rollfile.layout=org.apache.log4j.PatternLayout  log4j.appender. rollfile.layout.ConversionPattern =[%p] [%d{yyyy-MM-dd HH\:mm\:ss}][%c]%m%n |

### 源码分析



这是RollingFileAppender的subAppend方法,该方法做了两件事,一是调用父类的subAppend方法将信息写入文件中,二是判断写入后文件大小有没有打到限定的最大值,如果超过了最大值就调用rollOver方法,

**下面先看第一步将信息写入文件(下面是父类的subAppend方法)**



在父类的subAppend方法中layout的format方法对信息进行格式化,然后调用write方法将日志信息写入文件

**再来看看rollOver方法的实现**

|  |
| --- |
| **public** // synchronization not necessary since doAppend is alreasy synched  **void** rollOver() {  File target;  File file;  **if** (qw != **null**) {  **long** size = ((CountingQuietWriter) qw).getCount();  LogLog.*debug*("rolling over count=" + size);  // if operation fails, do not roll again until  // maxFileSize more bytes are written  nextRollover = size + maxFileSize;  }  LogLog.*debug*("maxBackupIndex="+maxBackupIndex);  **boolean** renameSucceeded = **true**;  // If maxBackups <= 0, then there is no file renaming to be done.  //先判断最大文件个数是不是大于零,如果等于零,那么就不需要进行文件重命名了  **if**(maxBackupIndex > 0) {  // Delete the oldest file, to keep Windows happy.  //如果当前文件的个数已经达到了最大值,那么先将最老的文件删除  file = **new** File(fileName + '.' + maxBackupIndex);  **if** (file.exists())  renameSucceeded = file.delete();  // Map {(maxBackupIndex - 1), ..., 2, 1} to {maxBackupIndex, ..., 3, 2}  //遍历备份文件,将所有已经存在的文件的文件名中的索引标识加1  **for** (**int** i = maxBackupIndex - 1; i >= 1 && renameSucceeded; i--) {  file = **new** File(fileName + "." + i);  **if** (file.exists()) {  target = **new** File(fileName + '.' + (i + 1));  LogLog.*debug*("Renaming file " + file + " to " + target);  renameSucceeded = file.renameTo(target);  }  }  //将最近操作的文件的文件名最后加.1后缀  **if**(renameSucceeded) {  // Rename fileName to fileName.1  target = **new** File(fileName + "." + 1);  **this**.closeFile(); // keep windows happy.  file = **new** File(fileName);  LogLog.*debug*("Renaming file " + file + " to " + target);  renameSucceeded = file.renameTo(target);  //  // if file rename failed, reopen file with append = true  //  **if** (!renameSucceeded) {  **try** {  **this**.setFile(fileName, **true**, bufferedIO, bufferSize);  }  **catch**(IOException e) {  **if** (e **instanceof** InterruptedIOException) {  Thread.*currentThread*().interrupt();  }  LogLog.*error*("setFile("+fileName+", true) call failed.", e);  }  }  }  }  //  // if all renames were successful, then  //  **if** (renameSucceeded) {  **try** {  // This will also close the file. This is OK since multiple  // close operations are safe.  **this**.setFile(fileName, **false**, bufferedIO, bufferSize);  nextRollover = 0;  }  **catch**(IOException e) {  **if** (e **instanceof** InterruptedIOException) {  Thread.*currentThread*().interrupt();  }  LogLog.*error*("setFile("+fileName+", false) call failed.", e);  }  }  } |

从上面的代码可以看到RollingFileAppender是先将信息写入文件,然后再判断文件大小是否超过了最大值