# 1. 日志和Log4J简介

## 1.1 概述

日志可用来记录程序运行的状况，可帮助开发者维护项目，排查错误。比如Tomcat运行时就会输出日志信息，并且会在Tomcat的log目录下生成log日志文件，可用文本编辑器查看。

日志有时显得非常重要，在程序中输出日志主要有以下作用：

（1）监视变量情况，把相关重要数据记录到文件中以便以后进行统计分析；

（2）跟踪代码的运行时轨迹，作为日后审计的依据；

（3）可充当开发或者生产环境中的“调试器”，向文件和控制台输出调试等信息。

因此在以后的项目中最好使用日志。

现在要介绍的Log4J（即Log For Java）是Apache维护的Java开源日志组件，组件对各种日志操作进行了封装，并且提供了多样的配置，方便开发者使用。log4J有两个大版本，即Log4J 1.x和Log4J 2.x，目前推荐使用Log4J 2.x的最新版本，因为Log4J 1.x已经停止维护，并且性能不如Log4J 2.x。

## 1.2 Log4J的三大构件

Log4J的三大构件分别是Logger、Appender和Layout，主要负责的功能如下：

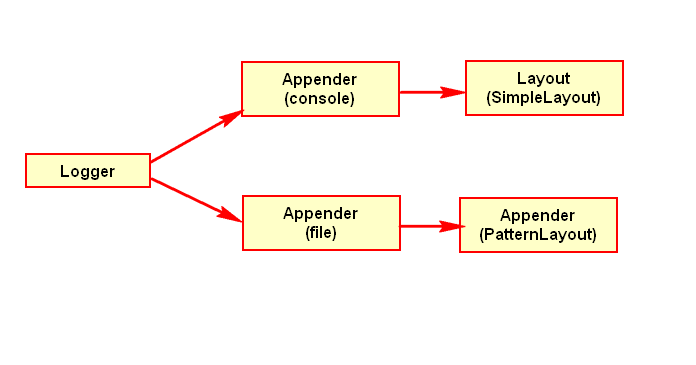
（1）Logger：负责输出日志。由开发者控制日志信息的输出。开发者自行在程序中调用Logger对象决定输出什么内容。

（2）Appender：用于定义日志输出的目的地，即指定日志应该输出到什么地方，这些位置可以是控制台、文件和网络设备等。

（3）Layout：指定日志的输出格式，例如是否显示日志发生的分钟数和秒数等。

这三个组件协同工作，它们之间的关系为：（1）一个Logger可对应多个Appender，这意味着日志可同时输出到多个设备；（2）每个Appender对应一个Layout，Layout决定了输出日志的格式。

假设某程序中的日志需要输出到控制台和指定的文件中，并且输出到控制台使用SimpleLayout布局，输出到文件是时使用PatternLayout布局，那么这时三个组件的关系如图所示：



# 2. Log4J的使用

Log4J的使用很简单，首先在Apache网站上下载Log4J的jar包，引入这两个文件：log4j-api-2.8.1.jar和log4j-core-2.8.1.jar。

首先，用LogManager类提供的静态方法getLogger()获得一个Logger对象。我们只需要用logger对象输出日志消息即可。日志消息分为6个级别，分别是：FATAL(致命错误)、ERROR(错误)、WARN(警告)、INFO(信息)、DEBUG(调试)和TRACE(细节)。其中FATAL的级别最高，TRACE的级别最低。Log4J的logger对象也分别提供了这6类方法来灵活的输出日志，例如：

|  |
| --- |
| **package** com.zhang.test;  **import** org.apache.logging.log4j.LogManager; **import** org.apache.logging.log4j.Logger;;  **public class** Demo {  **public static void** main(String[] args) **throws** Exception {  *// 获得Logger对象* Logger logger = LogManager.*getLogger*();  *// 输出不同级别的日志信息* logger.fatal(**"这是fatal级别的错误"**);  logger.error(**"这是error级别的错误"**);  logger.warn(**"这是warn级别的错误"**);  logger.info(**"这是info级别的错误"**);  logger.debug(**"这是debug级别的错误"**);  logger.trace(**"这是trace级别的错误"**);  } } |

运行程序后，发现默认只会输出error和fatal级别的日志信息。因为程序也提示我们应该为Log4J添加配置文件。这里的Logger就是日志器，我们可以对日志器的Appender和Layout进行配置，以便指定怎样输出日志。

需要注意的是，每个日志器是有名字的，名字默认就是当前类名，即com.zhang.test.Demo，通常这样显式的指定Logger的名称：

|  |
| --- |
| Logger logger = LogManager.*getLogger*(Demo.**class**.getName()); *// 显式指定logger名称是当前类名* |

## 2.1 Log4J配置文件

Log4J支持多种方式写配置文件，包括使用XML和JSON等。这里我们使用XML配置。我们需要在classpath路径下添加一个log4j2.xml文件（在项目的src目录下添加即可），Log4J会自动加载到该配置文件。

默认的配置文件内容和解释如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?> <!-- 根标签是configuration。status配置为OFF表示不输出Log4J本身的日志信息。当然，也可能设置为error、info等，这样会显示Log4J本身运行时的相关信息，建议设置为error级别即可 -->* <**configuration status="OFF"**>  <**appenders**>  *<!-- 在appenders标签中就用于配置多个appender。 -->  <!-- 这里用Console节点定义了一个向控制台输出日志的appender。名字就叫Console，目标就是系统的标准输出，即屏幕 -->* <**Console name="Console" target="SYSTEM\_OUT"**>  *<!-- 里面的节点就可定义Layout格式，这里的pattern就定义了输出格式 -->* <**PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"**/>  </**Console**>  </**appenders**>  <**loggers**>  *<!-- root可以说是所有日志器的“基类”，即root节点可统一配置所有的日志器，除非日志器再重新配置自己 -->  <!-- level设置了输出信息的级别为error。因此我们运行程序时只能看到大于等于error级别的信息 -->* <**root level="error"**>  *<!-- appender-ref用于引用上面定义的appender，注意ref里面写的是appender的name，而不是节点名称。 -->  <!-- logger可配置多个引用，以便同时输出信息到不同地方 -->* <**appender-ref ref="Console"**/>  </**root**>  </**loggers**> </**configuration**> |

有了这个文件，程序运行就不会提示没有配置文件了。为了看到所有的日志信息，我们可以把root节点中的level配置为trace，这样就能看到所有日志了。

下面再看稍微复杂的例子（主要涉及Appender配置）：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**configuration status="error"**>  *<!--先定义所有的appender-->* <**appenders**>  *<!-- 定义一个控制台输出配置 -->* <**Console name="Console" target="SYSTEM\_OUT"**>  *<!-- 该标签可控制输出级别。level表示匹配的级别，onMatch为ACCEPT表示级别匹配则接收，onMisMatch为DENY表示不匹配就拒绝-->  <!-- 这里就是让控制台输出trace级别及以上级别的日志 -->* <**ThresholdFilter level="trace" onMatch="ACCEPT" onMismatch="DENY"**/>  <**PatternLayout pattern="%d{HH:mm:ss.SSS} %-5level %class{36} %L %M - %msg%xEx%n"**/>  </**Console**>  *<!-- 这里定义一个文件输出配置。name表示该appender名称，fileName指定日志文件名，append表示是否追加，这里设置成false表示不追加，那么每次运行程序都会清空之前内容。适合测试用 -->* <**File name="log" fileName="log/test.log" append="false"**>  *<!-- 下面同样是配置输出级别和输出格式 -->* <**ThresholdFilter level="info" onMatch="ACCEPT" onMismatch="DENY"**/>  <**PatternLayout pattern="%d{HH:mm:ss.SSS} %-5level %class{36} %L %M - %msg%xEx%n"**/>  </**File**>  *<!-- 这里定义的是RollingFile的Appender。可进行存档 -->  <!-- 解释：该Appender会打印所有信息，每次当log文件大小超过5M时，就将日志压缩存档到按年份-月份建立的文件夹中 -->* <**RollingFile name="RollingFile" fileName="logs/app.log"  filePattern="log/$${date:yyyy-MM}/app-%d{MM-dd-yyyy}-%i.log.gz"**>  *<!-- 定义size大小 -->* <**SizeBasedTriggeringPolicy size="5MB"**/>  <**PatternLayout pattern="%d{yyyy-MM-dd 'at' HH:mm:ss z} %-5level %class{36} %L %M - %msg%xEx%n"**/>   </**RollingFile**>  </**appenders**>  *<!-- 然后定义logger并引用appender -->* <**loggers**>  *<!--使用默认的root-->* <**root level="trace"**>  <**appender-ref ref="Console"**/>  <**appender-ref ref="log"**></**appender-ref**>  <**appender-ref ref="RollingFile"**/>   </**root**>  </**loggers**> </**configuration**> |

为了证明上述配置的确有效，可以在程序中循环输出日志信息，看是否会生成存档文件。

## 2.2 详细配置和解释

（1）Appender组件中：

除了可以配置Console（控制套）、File（文件）和RollingFile（文件大小超过指定尺寸时就产生新文件）节点外，还能配置DailyRollingFile（每天产生一个日志文件）和Writer（将日志信息以流格式发送到任意指定的地方），可自行研究。注意点：

A：Console中：target默认是“SYSTEM\_OUT”，输出到控制台，也可配置成SYSTEM\_ERR，以err的红色样式输出信息。

B：RollingFile中：SizeBasedTriggeringPolicy的size属性，可以KB、MB或GB作为单位。

（2）Layout组件中：

我们主要使用的就是PatternLayout，可自行探究其他Layout。Layout中的pattern就是控制输出的具体格式的，详细解释如下：

%p：输出日志信息类型，即DEBUG、INFO等。也可以用%level代替。

%d：输出日志产生的时间点，默认格式为ISO8601，也可以在其后指定格式，比如：%d{yyyy年MM月dd日 HH时hh分ss秒SSS毫秒}。

%r：输出自应用启动到输出该log信息所花费的毫秒数。

%c：输出产生该日志事件的类名，可用%class或者%C代替。

%t：输出产生该日志事件的线程名，可用%thread代替。

%L：输出日志事件发生的行号。

%M：输出产生该日志事件的方法名。

%F：输出日志消息产生时所在的文件名称。

%l：输出日志事件发生的位置，相当于%C.%M(%F:%L)的组合，包括类名、线程以及在代码中的行数。例如：Testlog4.main(TestLog4.java:10)开发中可使用这个选项，在IDE中能够直接点击到对应的文件名和行号。

%x：输出和当前线程相关的NDC(嵌套诊断环境)，尤其在Servlet这样的多客户多线程的应用中使用。

%m：输出该日志的消息，可用%msg代替。

%n：输出一个回车换行符。

%%：输出一个"%"字符。

另外，可以在%与字符之间加上“数字”来控制文本的宽度和对齐方式，这和C语言的设计是相似的。比如：

%20c，则显示类名，最小宽度是20。

%-20c，使得类名左对齐。

%.30c，最大的宽度是30，如果名称大于30，则截掉左边多出的字符，但小于30的话也不会有空格。

%20.30c，如果名称小于20就补空格，并且右对齐，如果名称长于30字符，就从左边截掉。

（3）Logger日志器。

为了避免字符串拼接，输出时，可用占位符，然后传入可变参数，如：

|  |
| --- |
| logger.info(**"成功执行：{}，状态码：{}"**, **"更新操作"**, 1); |

之前为什么使用类名来给一个logger设置名字呢，因为日志器也有“继承”关系。我们可以在Loggers节点中再单独给一个节点进行相应配置。而root就是定义所有日志器的“基类”。关于这个自行研究。

主要的还是使用我们最后一个例子中的配置文件，可以按照需求做改动。