**Log4j Using Spring (Log4j+Commons Logging):-**

**Jars:**

****

|  |  |  |
| --- | --- | --- |
| **Logger** | **Appender** | **Layout** |

**Logger:**

private final Log log = LogFactory.getLog(this.getClass());

Methods:

**debug < info < warn < error < fatal**

**Appender:**

Appender job is to write the messages into the external file or database or smtp

Logger classes generates some statements under different levels right, this Appender takes these logstatements and stores in some files or database

Appender is an interface

In log4j we have different Appender  implementation classes

* FileAppender [ writing into a file ]
* ConsoleAppender [ Writing into console ]
* JDBCAppender [ For Databases ]
* SMTPAppender [ Mails ]
* SocketAppender [ For remote storage ]
* SocketHubAppender
* SyslogAppendersends
* TelnetAppender

Again in FileAppender we have 2 more

* RollingFileAppender
* DailyRollingFileAppender

LayOut:-

* SimpleLayout
* PatternLayout
* HTMLLayout
* XMLLayout

=========

Maintaining Properties File:

By default the file name would be log4j.properties. This properties file stores data in the form of key, values pairs, in this file keys are fixed but values are our own.  We can include all the log4j related properties into this file.

**log4j.properties:-**

**log4j.rootLogger**=DEBUG,CONSOLE,LOGFILE

log4j.appender.CONSOLE=  
log4j.appender.CONSOLE.layout=  
log4j.appender.CONSOLE.layout.ConversionPattern=

log4j.appender.LOGFILE=  
log4j.appender.LOGFILE.File=  
log4j.appender.LOGFILE.MaxFileSize=  
log4j.appender.LOGFILE.layout=  
log4j.appender.LOGFILE.layout.ConversionPattern=

Example:

**# Root logger option**

log4j.rootLogger=DEBUG, stdout, file

# Redirect log messages to console

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.out

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# Redirect log messages to a log file

log4j.appender.file=org.apache.log4j.RollingFileAppender

#outputs to Tomcat home

log4j.appender.file.File=${catalina.home}/logs/myapp.log

log4j.appender.file.MaxFileSize=5MB

log4j.appender.file.MaxBackupIndex=10

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

**log4j.logger.org.springframework=OFF**

=======

**Using AOP+Stopwatch+Logging :-**

package com.app.aop;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Aspect;

import org.springframework.stereotype.Component;

import org.springframework.util.StopWatch;

@Aspect

@Component

public class LoggingAspect {

private final Log log = LogFactory.getLog(this.getClass());

@Around("execution(\* com.app.\*..\*.\*(..))")

public Object logTimeMethod(ProceedingJoinPoint joinPoint) throws Throwable {

StopWatch stopWatch = new StopWatch();

stopWatch.start();

Object retVal = joinPoint.proceed();

stopWatch.stop();

StringBuffer logMessage = new StringBuffer();

logMessage.append(joinPoint.getTarget().getClass().getName());

logMessage.append(".");

logMessage.append(joinPoint.getSignature().getName());

logMessage.append("(");

// append args

Object[] args = joinPoint.getArgs();

for (int i = 0; i < args.length; i++) {

logMessage.append(args[i]).append(",");

}

if (args.length > 0) {

logMessage.deleteCharAt(logMessage.length() - 1);

}

logMessage.append(")");

logMessage.append(" execution time: ");

logMessage.append(stopWatch.getTotalTimeMillis());

logMessage.append(" ms");

log.info(logMessage.toString());

return retVal;

}

//@Around("execution(\* com.app.\*..\*.\*(..))")

public Object getLogTime(ProceedingJoinPoint jp) throws Throwable{

StopWatch watch=new StopWatch(jp.getTarget().getClass().getName());

watch.start(jp.getSignature().getName());

Object ob=jp.proceed();

watch.stop();

System.out.println(watch.prettyPrint());

return ob;

}

}

**For AOP JARS:**

**http://www.mediafire.com/file/2szvk5sjxgoqkdp/Maven.rar**

**====================**

**Patterns Example notes:**

A flexible layout configurable with pattern string. This code is known to have synchronization and other issues which are not present in org.apache.log4j.EnhancedPatternLayout. EnhancedPatternLayout should be used in preference to PatternLayout. EnhancedPatternLayout is distributed in the log4j extras companion.

The goal of this class is to format a Logging Event and return the results as a String. The results depend on the conversion pattern.

The conversion pattern is closely related to the conversion pattern of the printf function in C. A conversion pattern is composed of literal text and format control expressions called conversion specifiers.

You are free to insert any literal text within the conversion pattern.

Each conversion specifier starts with a percent sign (%) and is followed by optional format modifiers and a conversion character. The conversion character specifies the type of data, e.g. category, priority, date, thread name. The format modifiers control such things as field width, padding, left and right justification. The following is a simple example.

Let the conversion pattern be "%-5p [%t]: %m%n" and assume that the log4j environment was set to use a PatternLayout. Then the statements

Category root = Category.getRoot();

root.debug("Message 1");

root.warn("Message 2");

would yield the output

DEBUG [main]: Message 1

WARN [main]: Message 2

Note that there is no explicit separator between text and conversion specifiers. The pattern parser knows when it has reached the end of a conversion specifier when it reads a conversion character. In the example above the conversion specifier %-5p means the priority of the logging event should be left justified to a width of five characters. The recognized conversion characters are

|  |  |
| --- | --- |
| Conversion Character | Effect |
| c | Used to output the category of the logging event. The category conversion specifier can be optionally followed by precision specifier, that is a decimal constant in brackets.  If a precision specifier is given, then only the corresponding number of right most components of the category name will be printed. By default the category name is printed in full.  For example, for the category name "a.b.c" the pattern %c{2} will output "b.c". |
| C | Used to output the fully qualified class name of the caller issuing the logging request. This conversion specifier can be optionally followed by precision specifier, that is a decimal constant in brackets.  If a precision specifier is given, then only the corresponding number of right most components of the class name will be printed. By default the class name is output in fully qualified form.  For example, for the class name "org.apache.xyz.SomeClass", the pattern %C{1} will output "SomeClass".  WARNING Generating the caller class information is slow. Thus, use should be avoided unless execution speed is not an issue. |
| d | Used to output the date of the logging event. The date conversion specifier may be followed by a date format specifier enclosed between braces. For example, %d{HH:mm:ss,SSS} or %d{dd MMM yyyy HH:mm:ss,SSS}. If no date format specifier is given then ISO8601 format is assumed.  The date format specifier admits the same syntax as the time pattern string of the [SimpleDateFormat](http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html?is-external=true" \o "class or interface in java.text). Although part of the standard JDK, the performance of SimpleDateFormat is quite poor.  For better results it is recommended to use the log4j date formatters. These can be specified using one of the strings "ABSOLUTE", "DATE" and "ISO8601" for specifying [AbsoluteTimeDateFormat](https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/helpers/AbsoluteTimeDateFormat.html" \o "class in org.apache.log4j.helpers), [DateTimeDateFormat](https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/helpers/DateTimeDateFormat.html" \o "class in org.apache.log4j.helpers) and respectively [ISO8601DateFormat](https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/helpers/ISO8601DateFormat.html). For example, %d{ISO8601} or %d{ABSOLUTE}.  These dedicated date formatters perform significantly better than [SimpleDateFormat](http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html?is-external=true" \o "class or interface in java.text). |
| F | Used to output the file name where the logging request was issued.  WARNING Generating caller location information is extremely slow and should be avoided unless execution speed is not an issue. |
| l | Used to output location information of the caller which generated the logging event.  The location information depends on the JVM implementation but usually consists of the fully qualified name of the calling method followed by the callers source the file name and line number between parentheses.  The location information can be very useful. However, its generation is extremely slow and should be avoided unless execution speed is not an issue. |
| L | Used to output the line number from where the logging request was issued.  WARNING Generating caller location information is extremely slow and should be avoided unless execution speed is not an issue. |
| m | Used to output the application supplied message associated with the logging event. |
| M | Used to output the method name where the logging request was issued.  WARNING Generating caller location information is extremely slow and should be avoided unless execution speed is not an issue. |
| n | Outputs the platform dependent line separator character or characters.  This conversion character offers practically the same performance as using non-portable line separator strings such as "\n", or "\r\n". Thus, it is the preferred way of specifying a line separator. |
| p | Used to output the priority of the logging event. |
| r | Used to output the number of milliseconds elapsed from the construction of the layout until the creation of the logging event. |
| t | Used to output the name of the thread that generated the logging event. |
| x | Used to output the NDC (nested diagnostic context) associated with the thread that generated the logging event. |
| X | Used to output the MDC (mapped diagnostic context) associated with the thread that generated the logging event. The X conversion character must be followed by the key for the map placed between braces, as in %X{clientNumber} where clientNumber is the key. The value in the MDC corresponding to the key will be output.  See MDC class for more details. |
| % | The sequence %% outputs a single percent sign. |

By default the relevant information is output as is. However, with the aid of format modifiers it is possible to change the minimum field width, the maximum field width and justification.

The optional format modifier is placed between the percent sign and the conversion character.

The first optional format modifier is the left justification flag which is just the minus (-) character. Then comes the optional minimum field width modifier. This is a decimal constant that represents the minimum number of characters to output. If the data item requires fewer characters, it is padded on either the left or the right until the minimum width is reached. The default is to pad on the left (right justify) but you can specify right padding with the left justification flag. The padding character is space. If the data item is larger than the minimum field width, the field is expanded to accommodate the data. The value is never truncated.

This behavior can be changed using the maximum field width modifier which is designated by a period followed by a decimal constant. If the data item is longer than the maximum field, then the extra characters are removed from the beginning of the data item and not from the end. For example, it the maximum field width is eight and the data item is ten characters long, then the first two characters of the data item are dropped. This behavior deviates from the printf function in C where truncation is done from the end.

Below are various format modifier examples for the category conversion specifier.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Format modifier | left justify | minimum width | maximum width | comment |
| %20c | false | 20 | none | Left pad with spaces if the category name is less than 20 characters long. |
| %-20c | true | 20 | none | Right pad with spaces if the category name is less than 20 characters long. |
| %.30c | NA | none | 30 | Truncate from the beginning if the category name is longer than 30 characters. |
| %20.30c | false | 20 | 30 | Left pad with spaces if the category name is shorter than 20 characters. However, if category name is longer than 30 characters, then truncate from the beginning. |
| %-20.30c | true | 20 | 30 | Right pad with spaces if the category name is shorter than 20 characters. However, if category name is longer than 30 characters, then truncate from the beginning. |

Below are some examples of conversion patterns.

%r [%t] %-5p %c %x - %m%n

This is essentially the TTCC layout.

%-6r [%15.15t] %-5p %30.30c %x - %m%n

Similar to the TTCC layout except that the relative time is right padded if less than 6 digits, thread name is right padded if less than 15 characters and truncated if longer and the category name is left padded if shorter than 30 characters and truncated if longer.