Log4j Fundamentals

28/02/2011

Author

Narasimha Sharma M L
(narasimha.ml@tcs.com)

Log4J Introduction

Every application requires storing and analyzing of output. Most frequently used output destination in our applications is 'Console'. In java, we output to console using 'System.out' which returns a PrintStream object. To collect the output into a File, we need to use either a FileWriter or FileOutputStream or some other File I/O Object. Writing code to different outputstreams is simplified by the use of log4j libraries.

log4j is a logging tool for java used to log the output of an application into different destinations. log4j, with its vast API, offers different types of output destinations such as Console, File, Sockets, Mail, Syslog daemon, etc.

Using log4j, it is possible to log into any number of these output destinations simultaneously. The main components of log4j library are **Logger**, **Appender** and **Layout** classes.

Log4j configuration can be done through any of the following

- Code
- log4j.properties file
- log4j.xml file.

If configuration is done through log4j properties file or XML file, it is applicable for the entire runtime unless overridden through the code. In general, default logging is performed in either properties file or Xml file and dynamic logging is configured through code.

LOGGER

Logger class uses message levels to store logs. Message levels indicate the priority of the logging message. This level of logging can be controlled in such a way that only certain message levels can be enabled for logging.

log4j Message levels: trace, debug, warn, info, error, fatal

These message levels are sorted by their priority with trace having the lowest priority and fatal having the highest priority. Hence, Fatal is used to represent the most important messages like a system crash or unrecoverable errors whereas Trace is used to represent a detailed or a fine grained message.

To print output using log4j library, we use one of these methods defined in Logger class.

```
log.trace(“This is my trace message”);
log.debug("This is my debug message.");
log.info("This is my info message.");
log.warn("This is my warn message.");
log.error("This is my error message.");
log.fatal("This is my fatal message.");
```

The output of each of these methods appears like this in your console:

```
TRACE [Message] hyd.home.MyWorld – This is my trace message.
DEBUG [Message] hyd.home.MyWorld - This is my debug message.
WARN [Message] hyd.home.MyWorld - This is my warn message.
INFO [Message] hyd.home.MyWorld - This is my info message.
ERROR [Message] hyd.home.MyWorld - This is my error message.
FATAL [Message] hyd.home.MyWorld - This is my fatal message.
```

Each of these methods defines a different level of Logging. With log4j you can enable or disable logging done by these methods in the output.

The output message can be formatted in the logger class. The logger class uses a layout object to format the output messages which will be discussed shortly.

Why do you want to filter a message?

When you're developing a code, you would want to know the lower level trace details of every step of execution to debug any errors. So, at some steps, we use the debug() method to print the details. Once the code is successfully developed and delivered, it is not necessary to know the lower level trace details often.

So, when developing code, you write some of the logging in debug mode and some of the logging in warning or error mode. But, instead of removing these logger statements while delivering the code, you can just set the logging level to disable viewing of debug mode logging. These statements can be enabled once again whenever errors are detected.

In Log4j, Category class is deprecated from version 1.2 and is replaced with Logger class which extends the Category class.

To instantiate a Logger class,

```
/***Code***/
import org.apache.log4j.Logger;
//To get the Root Logger
Logger logger = Logger.getRootLogger();
// To get the logger for a specific class
Logger logger = Logger.getLogger(MyClass.class);

/***PropertiesFile****/
log4j.rootLogger=INFO, A1, A2

/***XmlFile***/
    <root>
        <level value="INFO" />
        <appender-ref ref="A1" />
        <appender-ref ref="A2" />
    </root>

/***Code***/
Logger logger=Logger.getRootLogger();
logger.addAppender(A1);
logger.addAppender(A2);
logger.setLevel(Level.INFO);
```

A Root Logger is the default Logger of Log4J. However, Root logger has to be initialized with Appender and allowed Message Level values. The above example ensures that the root logger filters message levels below the INFO level (DEBUG, TRACE). It uses the Appenders A1 and A2 to log the messages. Logger can also be defined for an entire package.

Logger class allows for filtering of message levels by use of ‘level‘ attribute. This attribute is assigned one of the available levels (trace, debug, info, warn, error, fatal). It disables logging of message levels with priority lesser than the assigned value. To define appenders and message levels to a class ’MyWorld’ in ’hyd.home’ package,

```
/***PropertiesFile****/
Log4j.logger.hyd.home.MyWorld=INFO, A1

/***XmlFile***/
   <logger name="hyd.home.MyWorld">
        <level value="INFO"/>
        <appender-ref ref="A1"/>
   </logger>
```

```
/***Code***/
Import org.apache.log4j.Logger;
Import org.apache.log4j.Level;
Logger logger=Logger.getLogger(MyWorld.class) ;
Logger.setLevel(Level.INFO);
```

In the above example, it filters all the message levels below INFO level and allows for logging of levels INFO and above.

Log4J library offers message levels as constants in the class 'Level' defined in the library. Using the class 'Level', it is also possible to enable or disable logging of all the levels. It is done by using the Level.*ALL,* Level.*OFF* constants defined in the class 'Level'.

If you want to filter only a particular level unlike done as above, it is possible by use of a 'Filter' in appenders. So, at Logger level, you can disable group of levels. To filter only a selected level, it should be done at the lower level, the Appender level.

Logger inheritance

A root logger is always initialized in log4j configurations. If root logger is initialized, it ensures that all the other loggers also write to the same appender. If you define a new appender for a logger, it adds this new appender to the existing appenders inherited from its parent, unless specified not to. Appenders are inherited from the ancestors by default. To stop this additive logging, you have to set the additivity property of Logger to false.

```
/***PropertiesFile***/
log4j.logger.hyd.home.MyWorld.additivity=false

/***XmlFile***/
   <logger name="hyd.home.MyWorld" additivity="false">
        <level value="debug"/>
        <appender-ref ref="A1"/>
   </logger>

/***Code***/
Logger logger=Logger.getLogger(MyWorld.class) ;
Logger.addAppender(new ConsoleAppender(new PatternLayout())) ;
logger.setAdditivity(false);
```

Logger Instances

Any number of Logger instances can be obtained for a class. Since it uses the static method getLogger() of Logger class, all the instances are obtained during initialization and thus each of the instance is a same instance of the logger. And so, each logger can be customized using different appenders and can be used to log messages distinctly.

APPENDER

The other important component of log4j is an appender. Using appender, we can define the type of logging destinations.

The appenders we are going to discuss here are:

- Console Appender
- File Appender
- Rolling File Appender
- SMTP Appender

Console appender is used to log the output to the console just like what System.out.print() does.

```
/***PropertiesFile***/
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c
%x - %m%n

/***XmlFile***/
    <appender name="A1"
class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-4r [%t] %-5p %c
%x - %m%n" />
        </layout>
    </appender>

/***Code***/
Logger shellLogger = Logger.getLogger(MyWorld.class);
String logFile="C:/shellThreadOutput.log";
PatternLayout layout=new PatternLayout("%-4r [%t] %-5p %c %x -
%m%n");
ConsoleAppender  consoleAppender = new ConsoleAppender(layout,
logFile);
shellLogger.addAppender(consoleAppender);
```

File Appender is used to log the output to a File. The output file is defined in one of the appender's attributes. The attributes of a File Appender are

```
/***PropertiesFile****/
log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A2.File=fileAppenderLog.log
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%-4r [%t] %-5p %c
%x - %m%n

/***XmlFile***/
<appender name="A2"
class="org.apache.log4j.RollingFileAppender">
      <param name="File" value="test.log" />
      <layout class="org.apache.log4j.PatternLayout">
       <param name="ConversionPattern" value="%-4r [%t] %-5p %c %x -
%m%n" />
      </layout>
 </appender>

/***Code***/
Logger shellLogger = Logger.getLogger(MyWorld.class);
String logFile="C:/shellThreadOutput.log";
PatternLayout layout=new PatternLayout("%-4r [%t] %-5p %c %x - %m%n
");
FileAppender fileAppender = new FileAppender(layout, logFile);
shellLogger.addAppender(fileAppender);
```

Rolling File Appender is used when you want to control the maximum size of a log file. It is an extension to File Appender. The attributes of a Rolling file Appender are

```
/***PropertiesFile****/
log4j.appender.A3=org.apache.log4j.RollingFileAppender
log4j.appender.A3.maxFileSize=100KB
log4j.appender.A3.maxBackupIndex=10
log4j.appender.A3.File=RollingFileLog.log
log4j.appender.A3.layout=org.apache.log4j.PatternLayout
log4j.appender.A3.layout.ConversionPattern=%-4r [%t] %-5p %c %x -
%m%n

/***XmlFile***/
  <appender name="A3"
class="org.apache.log4j.RollingFileAppender">
       <param name="maxFileSize" value="100KB" />
       <param name="maxBackupIndex" value="10" />
```

```
       <param name="File" value="RollingFileLog.log" />
      <layout class="org.apache.log4j.PatternLayout">
       <param name="ConversionPattern" value="%-4r [%t] %-5p %c %x -
%m%n" />
      </layout>
   </appender>

/***Code***/
Logger shellLogger = Logger.getLogger(MyWorld.class);
String logFile="C:/shellThreadOutput.log”;
PatternLayout layout=new PatternLayout(“%-4r [%t] %-5p %c %x - %m%n
“);
RollingFileAppender rollingFileAppender = new RollingFileAppender
(layout, logFile);
rollingFileAppender.setMaxFileSize(“1MB”);
rollingFileAppender.setMaxBackUpIndex(5);
shellLogger.addAppender(rollingFileAppender);
```

SMTP Appender is used to log the output to mail. It requires different attributes required to configure the mail service. The attributes used by an SMTP Appender are

```
/***PropertiesFile****/
log4j.appender.A3=org.apache.log4j.net.SMTPAppender
log4j.appender.A3.BufferSize=1
log4j.appender.A3.SMTPHost=im.tcs.com
log4j.appender.A3.From=sender@tcs.com
log4j.appender.A3.To=receiver@tcs.com
log4j.appender.A3.Subject=LoggingExample
log4j.appender.A3.layout=org.apache.log4j.PatternLayout
log4j.appender.A3.layout.ConversionPattern=%-4r [%t] %-5p %c %x -
%m%n

/***XmlFile***/
<appender name=”A3” class=”org.apache.log4j.net.SMTPAppender”>
     <param name="SMTPHost" value="im.tcs.com " />
     <param name="From" value="sender@tcs.com " />
     <param name="To" value="receiver@tcs.com " />
     <param name="Subject" value="LoggingExample" />
     <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%-4r [%t] %-5p %c %x
- %m%n " />
     </layout>
  </appender>
```

```
/***Code***/
Logger shellLogger = Logger.getLogger(MyWorld.class);
PatternLayout layout=new PatternLayout("%-4r [%t] %-5p %c %x - %m%n
");
SmtpAppender smtpAppender = new smtpAppender (layout);
smtpAppender.setSMTPHost("im.tcs.com ");
smtpAppender.setSetFrom("sender@tcs.com");
smtpAppender.setTo("receiver@tcs.com");
smtpAppender.setSubject("LoggingExample");
smtpAppender.setBufferSize(1);
shellLogger.addAppender(rollingFileAppender);
```

In general, SMTP Appenders are used in logging important messages as there is a significant overhead in sending a mail. By default, SMTP will send an email when a message level of ERROR or FATAL is detected. We can set the parameter 'EvaluatorClass' to the class name which implements 'TriggeringEventEvaluator' interface. This interface has the method 'boolean isTriggeringEvent(LoggingEvent event)' which can be set to return true only when certain number of Errors occur. It is set in the property file as below

```
/***PropertyFile***/
log4j.appender.smtpAppender.evaluatorClass=hyd.home.MyEvaluator
/***XmlFile***/
<appender name="A3" class="org.apache.log4j.net.SMTPAppender">
  <param name="EvaluatorClass" value="hyd.home.MyEvaluator" />
</appender>
```

Threshold

At Appender level, messages can be filtered by setting the threshold parameter of appenders.

```
/***PropertiesFile****/
log4j.appender.A3.threshold=ERROR

/***XmlFile***/
<appender name="A3" class="org.apache.log4j.SmtpAppender">
     <param name="threshold" value="ERROR" />
</appender>

/***Code***/
Logger logger=Logger.getLogger(MyWorld.class) ;
SmtpAppender smtpAppender=new smtpAppender() ;
smtpAppender.setThreshold(Level.ERROR.toString()) ;
Logger.addAppender(smtpAppender) ;
```

The Threshold property in log4j will filter all the messages below the ERROR level and allow messages of ERROR or FATAL level to be logged. So, the Threshold property of appenders is similar to that of Level property of Loggers.

setLevel Vs setThreshold

In Appenders, setThreshold() and addFilter() are used to disable logging of unwanted message levels. In Loggers, setLevel() is used to disable logging of unwanted message levels. So, to disable logging at Appender level, it should be enabled at Logger level. If it isn't enabled at Logger level, disabling at Appender level has no effect as it is already disabled at the Logger.

Filters

To filter out selected message levels unlike following level priorities, we use Filters in appenders. Unlike setLevel() or setThreshold() which disables logging of messages below the specified level, filters allow disabling of the specified level.

```
/***XmlFile***/
    <appender name="CA" class="org.apache.log4j.ConsoleAppender">
      <filter class="org.apache.log4j.varia.LevelMatchFilter">
         <param name="LevelToMatch" value="DEBUG" />
         <param name="AcceptOnMatch" value="false" />
      </filter>
    </appender>

/***Code***/
import org.apache.log4j.varia.LevelMatchFilter;
ConsoleAppender consoleAppender=new ConsoleAppender(new
PatternLayout("%p %c - %m%n"));
LevelMatchFilter levelMatchFilter=new LevelMatchFilter();
levelMatchFilter.setLevelToMatch("INFO");
levelMatchFilter.setAcceptOnMatch(false);
consoleAppender.addFilter(levelMatchFilter);
shellLogger.addAppender(consoleAppender);
```

Filter cannot be configured using a property file. So, we should use an Xml file or write the filter in your code directly.

LAYOUT

Logging of output into a Target can be formatted using log4j's Layout class. Layout class uses the attribute 'Conversion Pattern' in PatternLayout to format the output into the target. If no conversion pattern is defined, layout uses the default pattern of printing just the message.

We discuss the following layouts here.

Simple layout

Conversion pattern can't be set using Simple layout. It displays output as <message level> - <message>.

Eg: `DEBUG – This message is using simple layout.`

```
import org.apache.log4j.SimpleLayout;

FileAppender appender=new FileAppender (new SimpleLayout(),
"c:/fileLog.log");
```

Html layout

Output of Html layout is created in the form of a table. It is recommended that the appenders using HTML layout should use encoding type UTF-16.

```
import org.apache.log4j.HtmlLayout;

FileAppender appender=new FileAppender (new HtmlLayout(),
"c:/fileLog.html");
appender.setEncoding("UTF-16");
```

Pattern layout

Pattern layout allows for formatting of output. Appender object uses the conversion pattern property of Pattern layout object to format the output message.

The conversion pattern "`%d{ABSOLUTE} [%t] %-5p %c - %m%n`" will output

```
11:30:25,459 [main] INFO  hyd.home.MyWorld – This is my message.
```

%d{ABSOLUTE} – Absolute date timestamp
%t – name of the thread
%p – Message level
%c – Fully classified name of calling class
%m – Message
%n – line separator

Object Renderer

If you frequently need to use the attributes of an object in logging, it can be simplified by the use of 'ObjectRenderer' Interface. A Renderer class, which implements the ObjectRenderer interface, is defined for the required object. The Renderer is used to prepare the output string from the object, which is output by the logger methods.

When this Renderer is registered for the required object, it returns the message of the object as a string in the logger methods.

```
/***XmlFile***/
<renderer renderedClass="hyd.home.MyWorld"
                          renderingClass="hyd.home.WorldRenderer" />
 <appender name="A1" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%t %-5p %c{2} -%m%n"/>
    </layout>
 </appender>
<logger name="hyd.home.MyWorld">
    <level value="debug"/>
    <appender-ref  ref="A1"/>
 </logger>

/***PropertiesFile***/
log4j.renderer.hyd.home.MyWorld=hyd.home.WorldRenderer
log4j.appender.A1=org.apache.loog4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%t %-5p %c{2} -%m%n
log4j.logger.hyd.home.MyWorld=DEBUG, A1

/***Code**/
import org.apache.log4j.or.ObjectRenderer;
import org.apache.log4j.Logger;
```

```
class MyWorld{
  private String myName;
  public MyWorld(String name) {
    this.myName = name;
  }
  public String getMyName() {
    return this.myName;
  }
  public void setMyName(String name) {
    this.myName = name;
  }
}

class WorldRenderer implements ObjectRenderer {
private String noMessage = "No Name Found in your world";
private MyWorld world;
public String doRender(Object object) {
  if (object instanceof MyWorld)
  {
      world = (MyWorld) object;
      String message=world.getMyName()+" name found in your
world";
      return message;
   }
  return noMessage;
 }
}
public class mainWorld {
  public static void main(String args[]) {
  Logger logger = Logger.getLogger(mainWorld.class);
  MyWorld myWorld = new MyWorld("INDIA");
  logger.info(myWorld);
  }
}
```

Object rendering is also inherited just like appenders. If a class is registered with a Renderer, all its subclasses inherit the same Renderer unless a new Renderer is registered for the class.

Example of log4j properties File

log4j.property file contains logging information in the form of "**parameter=value**".

```
log4j.rootLogger=DEBUG, A1, A2

log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c
%x - %m%n

log4j.appender.A2=org.apache.log4j.RollingFileAppender
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%-4r [%t] %-5p %c
%x - %m%n
log4j.appender.A2.maxBackupIndex=5
log4j.appender.A2.maxFileSize=100KB
log4j.appender.A2.file=fileAppender.log
log4j.appender.A2.threshold=INFO

log4j.appender.A3=org.apache.log4j.SmtpAppender
log4j.appender.A3.layout=org.apache.log4j.PatternLayout
log4j.appender.A3.layout.ConversionPattern=%-4r [%t] %-5p %c
%x - %m%n
log4j.appender.A3.SMTPHost=im.tcs.com
log4j.appender.A3.From=sender@tcs.com
log4j.appender.A3.To=receiver@tcs.com
log4j.appender.A3.Subject=LogExample
log4j.appender.A3.BufferSize=1
log4j.appender.A3.threshold=ERROR

log4j.logger.hyd.home.MyWorld =ERROR, A3
log4j.additivity.hyd.home.MyWorld =false
```

Example of log4j.xml File

Log4j can also be configured by log4j.xml. Xml file allows for more readability compared to log4j property file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration
xmlns:log4j='http://jakarta.apache.org/log4j/'>

    <appender name="A1" class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-4r [%t] %-5p %c
%x - %m%n" />
        </layout>
    </appender>
```

```
  <appender name="A2" class="org.apache.log4j.RollingFileAppender">
       <param name="maxFileSize" value="100KB" />
       <param name="maxBackupIndex" value="5" />
       <param name="File" value="test.log" />
      <param name="threshold" value="INFO"/>
      <layout class="org.apache.log4j.PatternLayout">
       <param name="ConversionPattern" value="%-4r [%t] %-5p %c %x -
%m%n" />
      </layout>
   </appender>

 <appender name=”A3”
class=”org.apache.log4j.net.SMTPAppender”>
     <param name="SMTPHost" value="im.tcs.com" />
     <param name="From" value="sender@tcs.com" />
     <param name="To" value="receiver@tcs.com" />
     <param name="Subject" value="LogExample" />
     <param name="BufferSize" value="1" />
     <param name="threshold" value="ERROR" />
     <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%-4r [%t] %-5p %c %x
- %m%n " />
     </layout>
  </appender>

    <root>
        <level value="debug" />
        <appender-ref ref="A1" />
        <appender-ref ref="A2" />
    </root>
   <logger name="hyd.home.MyWorld" additivity="false">
        <level value="ERROR"/>
        <appender-ref ref="A3"/>
   </logger>
</log4j:configuration>
```

What mandatory properties do we need to specify in a log4j configuration file?

RootLogger present in the log4j isn't configured by default. So, we have to set the appender and allowed message levels to the root logger in the configuration file. Otherwise, until a logger is configured in the code, no output is displayed through logger methods. Once the root logger is set, it is atleast assured that all the classes inherit the root logger unless a logger is overridden.

Initializing loggers in Code:

Import Statements

```
import org.apache.log4j.FileAppender;
import org.apache.log4j.Level;
import org.apache.log4j.PatternLayout;
import org.apache.log4j.Logger;
```

Code

```
Logger shellLogger = Logger.getLogger(MyWorld.class);
String logFile="C:/MyWorldOutput.log”;
FileAppender fileAppender = new FileAppender(new PatternLayout(),
logFile);
shellLogger.removeAllAppenders();
shellLogger.addAppender(fileAppender);
shellLogger.setAdditivity(false);
shellLogger.setLevel(Level.INFO);
```

Configuring Loggers through configuration files in Code

Logger configuration files can be changed dynamically in the code by using the configure() method of Configurators. PropertyConfigurator is used to configure property file whereas DOMConfigurator is used to configure an XML file. BasicConfigurator is used to configure the default appenders.

BasicConfigurator

```
import org.apache.log4j.BasicConfigurator;

BasicConfigurator.configure();
```

This static method of BasicConfigurator adds a ConsoleAppender that uses a PatternLayout with ConversionPattern ‘TTCC_CONVERSION_PATTERN’ and prints to console.

PropertyConfigurator

Example-1:

```
import org.apache.log4j.PropertyConfigurator;
import org.apache.log4j.helpers.Loader;
```

```
Properties newProperties = new Properties();
try
{
  // load our log4j properties / configuration file
  newProperties.load(new FileInputStream(new
File("myLog4J.properties"));
  PropertyConfigurator.configure(newProperties);
}
catch(IOException e)
{
  e.printStackTrace();
}
```

Example-2

```
// use the loader helper from log4j
URL url = Loader.getResource("myLog4J.properties");
PropertyConfigurator.configure(url);

// use the same class loader as your class
URL url = MyClass.class.getResource("myLog4J.properties");
PropertyConfigurator.configure(url);
```

DOM Configurator

```
import org.apache.log4j.xml.DOMConfigurator;
import org.apache.log4j.helpers.Loader;

// load custom XML configuration
URL url = Loader.getResource("myLog4J.xml");
DOMConfigurator.configure(url);
```

Reference

- http://logging.apache.org/log4j/1.2/manual.html