

通过编译如下的程序来测试我们的安装是否正确。

import org.apache.log4j.Logger;

import org.apache.log4j.BasicConfigurator;

public class Hello {

static Logger logger = Logger.getLogger(Hello.class);

public static void main(String argv[]) {

BasicConfigurator.configure();

logger.debug("Hello world.");

}

}

BasicConfigurator类。

package org.apache.log4j;

/\*\*

\*使用这个类可以快速地配置log4j包，如果需要基于properties文件的配置，使用PropertyConfigurator类；如果需要xml文件的配置，使用DOMConfigurator类。

\*/

public class BasicConfigurator {

/\*\*

\*添加一个使用PatternLayout布局的ConsoleAppender，并且将其添加到root日志记录器。

\*/

public static void configure() {

Logger root = Logger.getRootLogger();

root.addAppender(

new ConsoleAppender(

new PatternLayout(PatternLayout.TTCC\_CONVERSION\_PATTERN)));

}

}

Logger类。

/\*\*

\* 这是log4j的核心类。大部分的日志操作，除了配置之外，都是通过这个类来实现的。

\*/

public class Logger extends Category {

}

Logger的继承类Category类。

/\*\*

\* Logger是Category的一个子类，它继承自Category。

\*/

public class Category implements ULogger, AppenderAttachable {

/\*\*

\* Category类的全名。

\*/

private static final String FQCN = Category.class.getName();

/\*\*

\* category的名字。

\*/

protected String name;

/\*\*

\* category的级别。

\*/

protected volatile Level level;

/\*\*

\* category的父类。

\*/

protected volatile Category parent;

// 输出目的地。

AppenderAttachableImpl aai;

/\*\*

\* 添加新的输出目的地到Category的目的地列表。如果该输出目的地已经存在于该列表，则不会再次新增。

\* </p>

\*/

public void addAppender(Appender newAppender) {

try {

if (aai == null) {

aai = new AppenderAttachableImpl();

}

aai.addAppender(newAppender);

} finally {

}

}

/\*\*

\* 记录调试级别的日志。

\*/

public void debug(Object message)

forcedLog(FQCN, Level.DEBUG, message, null);

}

/\*\*

\* 该方法创建一个新的日志上下文，并且记录相关日志信息。

\*/

protected void forcedLog(String fqcn, Priority level, Object message, Throwable t) {

callAppenders(new LoggingEvent(fqcn, (Logger) this, level, message, t));

}

/\*\*

\* 调用输出目的地进行日志记录。

\*/

public void callAppenders(LoggingEvent event) {

int writes = 0;

for (Category c = this; c != null; c = c.parent) {

try {

if (c.aai != null) {

writes += c.aai.appendLoopOnAppenders(event);

}

} finally {

}

}

}

}

AppenderAttachableImpl类。

/\*\*

\*AppenderAttachable类的实现类。

\*/

public class AppenderAttachableImpl implements AppenderAttachable {

/\*\* 输出目的地列表 \*/

protected Vector appenderList;

/\*\*

\*调用所有输出目的地的doAppend方法。

\*/

public int appendLoopOnAppenders(LoggingEvent event) {

int size = 0;

Appender appender;

if (appenderList != null) {

size = appenderList.size();

for (int i = 0; i < size; i++) {

appender = (Appender) appenderList.elementAt(i);

appender.doAppend(event);

}

}

return size;

}

}

ConsoleAppender类。

/\*\*

\* ConsoleAppender使用用户指定的格式将日志输出到System.out或者System.err。

\*/

public class ConsoleAppender extends WriterAppender {

/\*\*

\* 构造一个输了目的地。

\*/

public ConsoleAppender(final Layout layout) {

setLayout(layout);

}

}

ConsoleAppender类的父类WriterAppender类。

/\*\*

\* WriterAppender输出日志到java.io.Writer或者java.io.OutputStream，这依赖于用户的选择。

\*/

public class WriterAppender extends AppenderSkeleton {

/\*\*

\* 这是一个我们将要记录日志的地方QuietWriter。

\*/

protected QuietWriter qw;

/\*\*

This method is called by the {@link AppenderSkeleton#doAppend}

method.

<p>If the output stream exists and is writable then write a log

statement to the output stream. Otherwise, write a single warning

message to <code>System.err</code>.

<p>The format of the output will depend on this appender's

layout.

\*/

public void append(LoggingEvent event) {

if (!checkEntryConditions()) {

return;

}

subAppend(event);

}

/\*\*

\* 这是实际进行日志记录的方法。

\* 大部分WriterAppender的子类需要重写这个方法。

\*/

protected void subAppend(LoggingEvent event) {

if(!isActive()) {

return;

}

this.qw.write(this.layout.format(event));

if (layout.ignoresThrowable()) {

String[] s = event.getThrowableStrRep();

if (s != null) {

int len = s.length;

for (int i = 0; i < len; i++) {

this.qw.write(s[i]);

this.qw.write(Layout.LINE\_SEP);

}

}

}

if (this.immediateFlush) {

this.qw.flush();

}

}

}

WriterAppender的父类AppenderSkeleton类。

/\*\*

\* 这是log4j包里其它输出目的地的抽象父类。这个类提供了公用功能的代码。

\*/

public abstract class AppenderSkeleton extends ComponentBase implements Appender, OptionHandler {

/\*\*

\* 布局变量。如果输出目的地实现类有指定布局，则不需要设置该变量。

\*/

protected Layout layout;

/\*\*

\* 输出目的地的名称。

\*/

protected String name;

/\*\*

\* 设置输出目的地的布局。注意：有些输出目的地有它们自己的布局而无需指定该变量。

\*/

public void setLayout(Layout layout) {

this.layout = layout;

}

/\*\*

\* 这个方法执行一些日志记录前的检查然后调用子类的日志记录功能。

\*/

public synchronized void doAppend(LoggingEvent event) {

... ...

this.append(event);

}

}

PatternLayout类。

/\*\*

\* 一个灵活的使用模式字符串的布局。这个类的目标是格式化一个日志上下文并返回结果。

\*

public class PatternLayout extends Layout {

public static final String DEFAULT\_CONVERSION\_PATTERN = "%m%n";

public static final String TTCC\_CONVERSION\_PATTERN = "%r [%t] %p %c %x - %m%n";

/\*\*

\* 模式转换器。

\*/

private PatternConverter head;

/\*\*

\* 转换模式。

\*/

private String conversionPattern;

/\*\*

\* 使用给定的模式字符串构造一个模式布局类。

\* @param pattern conversion pattern.

\*/

public PatternLayout(final String pattern) {

this.conversionPattern = pattern;

head = createPatternParser(

(pattern == null) ? DEFAULT\_CONVERSION\_PATTERN : pattern).parse();

}

/\*\*

\* 返回一个模式转换器的解析器。

\*/

protected org.apache.log4j.helpers.PatternParser createPatternParser(String pattern) {

return new org.apache.log4j.pattern.BridgePatternParser(pattern,repository, getLogger());

}

/\*\*

\* 格式化一个日志上下文到writer。

\*/

public String format(final LoggingEvent event) {

StringBuffer buf = new StringBuffer();

for(PatternConverter c = head;

c != null;

c = c.next) {

c.format(buf, event);

}

return buf.toString();

}

}

解析器BridgePatternParser类。

/\*\*

\* 该类实现于log4j 1.3的org.apache.log4j.helpers.PatternConverter类。

\*/

public final class BridgePatternParser extends org.apache.log4j.helpers.PatternParser {

/\*\*

\* 创建一个新的模式转换器。

\* @return pattern converter.

\*/

public org.apache.log4j.helpers.PatternConverter parse() {

return new BridgePatternConverter(pattern, repository, logger);

}

}

转换器BridgePatternConverter类。

/\*\*

\* The class implements the pre log4j 1.3 org.apache.log4j.helpers.PatternConverter

\* 该类实现了log4j 1.3的org.apache.log4j.helpers.PatternConverter类。

\*/

public final class BridgePatternConverter extends org.apache.log4j.helpers.PatternConverter {

/\*\*

\* 模式转换器数组。

\*/

private LoggingEventPatternConverter[] patternConverters;

/\*\*

\* 日志的长度和排列规则。

\*/

private FormattingInfo[] patternFields;

/\*\*

\* 构造一个BridgePatternConverter。

\*/

public BridgePatternConverter(final String pattern, final LoggerRepository repository,final ULogger logger) {

patternConverters = new LoggingEventPatternConverter[converters.size()];

patternFields = new FormattingInfo[converters.size()];

Iterator converterIter = converters.iterator();

Iterator fieldIter = fields.iterator();

while (converterIter.hasNext()) {

Object converter = converterIter.next();

if (converter instanceof LoggingEventPatternConverter) {

patternConverters[i] = (LoggingEventPatternConverter) converter;

} else {

patternConverters[i] = new org.apache.log4j.pattern.LiteralPatternConverter("");

}

if (fieldIter.hasNext()) {

patternFields[i] = (FormattingInfo) fieldIter.next();

} else {

patternFields[i] = FormattingInfo.getDefault();

}

}

}

/\*\*

Format event to string buffer.

@param sbuf string buffer to receive formatted event, may not be null.

@param e event to format, may not be null.

\*/

public void format(final StringBuffer sbuf, final LoggingEvent e) {

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

int startField = sbuf.length();

patternConverters[i].format(e, sbuf);

patternFields[i].format(startField, sbuf);

}

}

}

LoggingEventPatternConverter类。

/\*\*

\* LoggingEventPatternConverter是一个模式转换器的基础类，可以从日志上下文转换信息。

\*/

public abstract class LoggingEventPatternConverter extends PatternConverter {

/\*\*

\* 格式化一个日志上下文到StringBuffer.

\*/

public abstract void format(final LoggingEvent event, final StringBuffer toAppendTo);

/\*\*

\* {@inheritDoc}

\*/

public void format(final Object obj, final StringBuffer output) {

if (obj instanceof LoggingEvent) {

format((LoggingEvent) obj, output);

}

}

LoggingEventPatternConver的子类LiteralPatternConverter。

/\*\*

\* 格式化一个字符串。

\*/

public final class LiteralPatternConverter extends LoggingEventPatternConverter {

private final String literal;

/\*\*

\* 构造一个新实例。

\*/

public LiteralPatternConverter(final String literal) {

super("Literal", "literal");

this.literal = literal;

}

/\*\*

\* {@inheritDoc}

\*/

public void format(final LoggingEvent event, final StringBuffer toAppendTo) {

toAppendTo.append(literal);

}

}

FormattingInfo类。

/\*\*

\* 根据一个指定的最小和最大宽度和排列来修改模式转换器的输出。

\*/

public final class FormattingInfo {

/\*\*

\* 根据指定的长度和排列方式来调整buffer的内容。

\*/

public final void format(final int fieldStart, final StringBuffer buffer) {

final int rawLength = buffer.length() - fieldStart;

if (rawLength > maxLength) {

buffer.delete(fieldStart, buffer.length() - maxLength);

} else if (rawLength < minLength) {

if (leftAlign) {

final int fieldEnd = buffer.length();

buffer.setLength(fieldStart + minLength);

for (int i = fieldEnd; i < buffer.length(); i++) {

buffer.setCharAt(i, ' ');

}

} else {

int padLength = minLength - rawLength;

for (; padLength > 8; padLength -= 8) {

buffer.insert(fieldStart, SPACES);

}

buffer.insert(fieldStart, SPACES, 0, padLength);

}

}

}

}