4、支持按文件大小滚动的Appender

```
package com.cdsxt.util;
import java.io.File;
import java.io.IOException;
import java.io.Writer;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import java.util.Locale;
import java.util.TimeZone;

import org.apache.log4j.FileAppender;
import org.apache.log4j.Layout;
import org.apache.log4j.helpers.CountingQuietWriter;
import org.apache.log4j.helpers.LogLog;
import org.apache.log4j.helpers.OptionConverter;
import org.apache.log4j.spi.LoggingEvent;

/**
 * <appender name="PROJECT" class="com.bao.logging.MyDailyRollingFileAppender">
        <param name="file" value="e:/test.log"/>
        <param name="DatePattern" value="'.'yyyy-MM-dd'.log'" />
        <param name="append" value="true"/>
        <param name="MaxFileSize" value="500MB"/>
        <param name="MaxBackupIndex" value="20"/>
<!--        <param name="MaxBackupIndex" value="-1"/> --><!-- 无限的文件数量，index顺序按时间顺序递增 -->
        <param name="encoding" value="UTF-8"/>
        <param name="threshold" value="info"/>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="[%d{dd HH:mm:ss,SSS\} %-5p] [%t] %c{2\} - %m%n"/>
        </layout>
    </appender>
 */
public class MyDailyRollingFileAppender extends FileAppender {

    // The code assumes that the following constants are in a increasing
    // sequence.
    static final int TOP_OF_TROUBLE = -1;
    static final int TOP_OF_MINUTE = 0;
    static final int TOP_OF_HOUR = 1;
    static final int HALF_DAY = 2;
    static final int TOP_OF_DAY = 3;
    static final int TOP_OF_WEEK = 4;
    static final int TOP_OF_MONTH = 5;

    /**
     * The default maximum file size is 10MB.
     */
    protected long maxFileSize = 10 * 1024 * 1024;

    /**
     * There is one backup file by default.
     */
    protected int maxBackupIndex = 1;

    /**
     * The date pattern. By default, the pattern is set to "'.'yyyy-MM-dd"
     * meaning daily rollover.
     */
    private String datePattern = "'.'yyyy-MM-dd";

    /**
     * The log file will be renamed to the value of the scheduledFilename
     * variable when the next interval is entered. For example, if the rollover
 * period is one hour, the log file will be renamed to the value of
 * "scheduledFilename" at the beginning of the next hour.
 *
 * The precise time when a rollover occurs depends on logging activity.
 */
private String scheduledFilename;

/**
 * The next time we estimate a rollover should occur.
 */
private long nextCheck = System.currentTimeMillis() - 1;

Date now = new Date();

SimpleDateFormat sdf;

RollingCalendar rc = new RollingCalendar();

int checkPeriod = TOP_OF_TROUBLE;

// The gmtTimeZone is used only in computeCheckPeriod() method.
static final TimeZone gmtTimeZone = TimeZone.getTimeZone("GMT");

/**
 * The default constructor does nothing.
 */
public MyDailyRollingFileAppender() {
}

/**
 * Instantiate a <code>MyDailyRollingFileAppender</code> and open the file
 * designated by <code>filename</code>. The opened filename will become the
 * ouput destination for this appender.
 */
public MyDailyRollingFileAppender(Layout layout, String filename,
        String datePattern) throws IOException {
    super(layout, filename, true);
    this.datePattern = datePattern;
    activateOptions();
}

/**
 * Get the maximum size that the output file is allowed to reach before
 * being rolled over to backup files.
 *
 * @since 1.1
 */
public long getMaximumFileSize() {
    return maxFileSize;
}

/**
 * Set the maximum size that the output file is allowed to reach before
 * being rolled over to backup files.
 *
 * <p>
 * This method is equivalent to {@link #setMaxFileSize} except that it is
 * required for differentiating the setter taking a <code>long</code>
 * argument from the setter taking a <code>String</code> argument by the
 * JavaBeans {@link java.beans.Introspector Introspector}.
 *
 * @see #setMaxFileSize(String)
 */
public void setMaximumFileSize(long maxFileSize) {
    this.maxFileSize = maxFileSize;
}

/**
```

```
 * Set the maximum size that the output file is allowed to reach before
 * being rolled over to backup files.
 *
 * <p>
 * In configuration files, the <b>MaxFileSize</b> option takes an long
 * integer in the range 0 - 2^63. You can specify the value with the
 * suffixes "KB", "MB" or "GB" so that the integer is interpreted being
 * expressed respectively in kilobytes, megabytes or gigabytes. For example,
 * the value "10KB" will be interpreted as 10240.
 */
public void setMaxFileSize(String value) {
    maxFileSize = OptionConverter.toFileSize(value, maxFileSize + 1);
}

/**
 * Returns the value of the <b>MaxBackupIndex</b> option.
 */
public int getMaxBackupIndex() {
    return maxBackupIndex;
}

/**
 * Set the maximum number of backup files to keep around.
 *
 * <p>
 * The <b>MaxBackupIndex</b> option determines how many backup files are
 * kept before the oldest is erased. This option takes a positive integer
 * value. If set to zero, then there will be no backup files and the log
 * file will be truncated when it reaches <code>MaxFileSize</code>.
 */
public void setMaxBackupIndex(int maxBackups) {
    this.maxBackupIndex = maxBackups;
}

/**
 * The <b>DatePattern</b> takes a string in the same format as expected by
 * {@link SimpleDateFormat}. This options determines the rollover schedule.
 */
public void setDatePattern(String pattern) {
    datePattern = pattern;
}

/** Returns the value of the <b>DatePattern</b> option. */
public String getDatePattern() {
    return datePattern;
}

public void activateOptions() {
    super.activateOptions();
    if (datePattern != null && fileName != null) {
        now.setTime(System.currentTimeMillis());
        sdf = new SimpleDateFormat(datePattern);
        int type = computeCheckPeriod();
        printPeriodicity(type);
        rc.setType(type);
        File file = new File(fileName);
        scheduledFilename = fileName
                + sdf.format(new Date(file.lastModified()));

    } else {
        LogLog.error("Either File or DatePattern options are not set for appender ["
                + name + "].");
    }
}

void printPeriodicity(int type) {
    switch (type) {
    case TOP_OF_MINUTE:
        LogLog.debug("Appender [" + name + "] to be rolled every minute.");
        break;
    case TOP_OF_HOUR:
        LogLog.debug("Appender [" + name
                + "] to be rolled on top of every hour.");
        break;
    case HALF_DAY:
        LogLog.debug("Appender [" + name
                + "] to be rolled at midday and midnight.");
        break;
    case TOP_OF_DAY:
        LogLog.debug("Appender [" + name + "] to be rolled at midnight.");
        break;
    case TOP_OF_WEEK:
        LogLog.debug("Appender [" + name
                + "] to be rolled at start of week.");
        break;
    case TOP_OF_MONTH:
        LogLog.debug("Appender [" + name
                + "] to be rolled at start of every month.");
        break;
    default:
        LogLog.warn("Unknown periodicity for appender [" + name + "].");
    }
}

// This method computes the roll over period by looping over the
// periods, starting with the shortest, and stopping when the r0 is
// different from from r1, where r0 is the epoch formatted according
// the datePattern (supplied by the user) and r1 is the
// epoch+nextMillis(i) formatted according to datePattern. All date
// formatting is done in GMT and not local format because the test
// logic is based on comparisons relative to 1970-01-01 00:00:00
// GMT (the epoch).

int computeCheckPeriod() {
    RollingCalendar rollingCalendar = new RollingCalendar(gmtTimeZone,
            Locale.ENGLISH);
    // set sate to 1970-01-01 00:00:00 GMT
    Date epoch = new Date(0);
    if (datePattern != null) {
        for (int i = TOP_OF_MINUTE; i <= TOP_OF_MONTH; i++) {
            SimpleDateFormat simpleDateFormat = new SimpleDateFormat(
                    datePattern);
            simpleDateFormat.setTimeZone(gmtTimeZone); // do all date
                                                        // formatting in GMT
            String r0 = simpleDateFormat.format(epoch);
            rollingCalendar.setType(i);
            Date next = new Date(rollingCalendar.getNextCheckMillis(epoch));
            String r1 = simpleDateFormat.format(next);
            // System.out.println("Type = "+i+", r0 = "+r0+", r1 = "+r1);
            if (r0 != null && r1 != null && !r0.equals(r1)) {
                return i;
            }
        }
    }
    return TOP_OF_TROUBLE; // Deliberately head for trouble...
}

/**
 * Implements the usual roll over behaviour.
 *
 * <p>
 * If <code>MaxBackupIndex</code> is positive, then files {
 * <code>File.1</code>, ..., <code>File.MaxBackupIndex -1</code> are renamed
 * to {<code>File.2</code>, ..., <code>File.MaxBackupIndex</code> .
 * Moreover, <code>File</code> is renamed <code>File.1</code> and closed. A
 * new <code>File</code> is created to receive further log output.
 *
 * <p>
 * If <code>MaxBackupIndex</code> is equal to zero, then the
 * <code>File</code> is truncated with no backup files created.
 */
public// synchronization not necessary since doAppend is alreasy synched
void sizeRollOver() {
    File target;
    File file;

    LogLog.debug("rolling over count="
            + ((CountingQuietWriter) qw).getCount());
    LogLog.debug("maxBackupIndex=" + maxBackupIndex);

    String datedFilename = fileName + sdf.format(now);

    if (maxBackupIndex > 0) {
        // Delete the oldest file, to keep Windows happy.
        file = new File(datedFilename + '.' + maxBackupIndex);
        if (file.exists())
            file.delete();

        // Map {(maxBackupIndex - 1), ..., 2, 1} to {maxBackupIndex, ..., 3,
        // 2}
        for (int i = maxBackupIndex - 1; i >= 1; i--) {
            file = new File(datedFilename + "." + i);
            if (file.exists()) {
                target = new File(datedFilename + '.' + (i + 1));
                LogLog.debug("Renaming file " + file + " to " + target);
                file.renameTo(target);
            }
        }

        // Rename fileName to datedFilename.1
        target = new File(datedFilename + "." + 1);

        this.closeFile(); // keep windows happy.

        file = new File(fileName);
        LogLog.debug("Renaming file " + file + " to " + target);
        file.renameTo(target);
    }else if (maxBackupIndex < 0){//infinite number of files
        //find the max backup index
        for (int i = 1; i < Integer.MAX_VALUE; i++) {
            target = new File(datedFilename + "." + i);
            if (! target.exists()) {//Rename fileName to datedFilename.i
                this.closeFile();
                file = new File(fileName);
                file.renameTo(target);
                LogLog.debug("Renaming file " + file + " to " + target);
                break;
            }
        }
    }

    try {
        // This will also close the file. This is OK since multiple
        // close operations are safe.
        this.setFile(fileName, false, bufferedIO, bufferSize);
    } catch (IOException e) {
        LogLog.error("setFile(" + fileName + ", false) call failed.", e);
    }
    scheduledFilename = datedFilename;
}

public synchronized void setFile(String fileName, boolean append,
        boolean bufferedIO, int bufferSize) throws IOException {
    super.setFile(fileName, append, this.bufferedIO, this.bufferSize);
    if (append) {
        File f = new File(fileName);
        ((CountingQuietWriter) qw).setCount(f.length());
    }
}

protected void setQWForFiles(Writer writer) {
    this.qw = new CountingQuietWriter(writer, errorHandler);
}

/**
 * Rollover the current file to a new file.
 */
void timeRollOver() throws IOException {

    /* Compute filename, but only if datePattern is specified */
    if (datePattern == null) {
        errorHandler.error("Missing DatePattern option in rollOver().");
        return;
    }

    String datedFilename = fileName + sdf.format(now);
    // It is too early to roll over because we are still within the
    // bounds of the current interval. Rollover will occur once the
    // next interval is reached.
    if (scheduledFilename.equals(datedFilename)) {
        return;
    }

    // close current file, and rename it to datedFilename
    this.closeFile();

    File target = new File(scheduledFilename);
    if (target.exists()) {
        target.delete();
    }

    File file = new File(fileName);
    boolean result = file.renameTo(target);
    if (result) {
        LogLog.debug(fileName + " -> " + scheduledFilename);
    } else {
        LogLog.error("Failed to rename [" + fileName + "] to ["
                + scheduledFilename + "].");
    }

    try {
        // This will also close the file. This is OK since multiple
        // close operations are safe.
        super.setFile(fileName, false, this.bufferedIO, this.bufferSize);
    } catch (IOException e) {
        errorHandler.error("setFile(" + fileName + ", false) call failed.");
    }
    scheduledFilename = datedFilename;
}

/**
 * This method differentiates MyDailyRollingFileAppender from its super class.
 *
 * <p>
 * Before actually logging, this method will check whether it is time to do
 * a rollover. If it is, it will schedule the next rollover time and then
 * rollover.
 * */
protected void subAppend(LoggingEvent event) {
    long n = System.currentTimeMillis();

    if (n >= nextCheck) {
            now.setTime(n);
            nextCheck = rc.getNextCheckMillis(now);
            try {
                timeRollOver();
            } catch (IOException ioe) {
                LogLog.error("rollOver() failed.", ioe);
            }
        } else if ((fileName != null)
                && ((CountingQuietWriter) qw).getCount() >= maxFileSize) {
            sizeRollOver();
        }
        super.subAppend(event);

    }
}

/**
 * RollingCalendar is a helper class to MyDailyRollingFileAppender. Given a
 * periodicity type and the current time, it computes the start of the next
 * interval.
 * */
class RollingCalendar extends GregorianCalendar {

    int type = MyDailyRollingFileAppender.TOP_OF_TROUBLE;

    RollingCalendar() {
        super();
    }

    RollingCalendar(TimeZone tz, Locale locale) {
        super(tz, locale);
    }

    void setType(int type) {
        this.type = type;
    }

    public long getNextCheckMillis(Date now) {
        return getNextCheckDate(now).getTime();
    }

    public Date getNextCheckDate(Date now) {
        this.setTime(now);

        switch (type) {
        case MyDailyRollingFileAppender.TOP_OF_MINUTE:
            this.set(Calendar.SECOND, 0);
            this.set(Calendar.MILLISECOND, 0);
            this.add(Calendar.MINUTE, 1);
            break;
        case MyDailyRollingFileAppender.TOP_OF_HOUR:
            this.set(Calendar.MINUTE, 0);
            this.set(Calendar.SECOND, 0);
            this.set(Calendar.MILLISECOND, 0);
            this.add(Calendar.HOUR_OF_DAY, 1);
            break;
        case MyDailyRollingFileAppender.HALF_DAY:
            this.set(Calendar.MINUTE, 0);
            this.set(Calendar.SECOND, 0);
            this.set(Calendar.MILLISECOND, 0);
            int hour = get(Calendar.HOUR_OF_DAY);
            if (hour < 12) {
                this.set(Calendar.HOUR_OF_DAY, 12);
            } else {
                this.set(Calendar.HOUR_OF_DAY, 0);
                this.add(Calendar.DAY_OF_MONTH, 1);
            }
            break;
        case MyDailyRollingFileAppender.TOP_OF_DAY:
            this.set(Calendar.HOUR_OF_DAY, 0);
            this.set(Calendar.MINUTE, 0);
            this.set(Calendar.SECOND, 0);
            this.set(Calendar.MILLISECOND, 0);
            this.add(Calendar.DATE, 1);
            break;
        case MyDailyRollingFileAppender.TOP_OF_WEEK:
            this.set(Calendar.DAY_OF_WEEK, getFirstDayOfWeek());
            this.set(Calendar.HOUR_OF_DAY, 0);
            this.set(Calendar.SECOND, 0);
            this.set(Calendar.MILLISECOND, 0);
            this.add(Calendar.WEEK_OF_YEAR, 1);
            break;
        case MyDailyRollingFileAppender.TOP_OF_MONTH:
            this.set(Calendar.DATE, 1);
            this.set(Calendar.HOUR_OF_DAY, 0);
            this.set(Calendar.SECOND, 0);
            this.set(Calendar.MILLISECOND, 0);
            this.add(Calendar.MONTH, 1);
            break;
        default:
            throw new IllegalStateException("Unknown periodicity type.");
        }
        return getTime();
    }
}
```