`

Flume HDFS Sink使用及源码分析

HDFS Sink介绍

Flume导入数据HDFS，目前只支持创建序列化（sequence）文件和文本（text）文件。还支持这两个文件的压缩。文件可以根据运行的时间，数据的大小和时间的数量来进行周期性的滚动（关闭当前文件产生新的文件）。也可以根据数据属性分区，例如根据时间戳或机器分区。HDFS目录路径可以包含格式化的转义字符，生成目录路径可以通过格式化转移字符（escape sequences），HDFS sink通过这些转义字符生成一个目录或者文件去存储Event。当然在Flume中使用HDFS Sink的话，需要添加HDFS相关的Jar，这样Flume就能使用Hadoop的jar和Hadoop集群交互。注：Hadoop必须支持sync()。

以下是HDFS Sink支持的转义字符：
名称 描述

%{host}	替代Event Header被命名为“host”的值，支持任意的Header name。
%t	Unix毫秒时间
%a	短的周名称，例如：Mon, Tue, ...
%A	周名称全称，例如：Monday, Tuesday, ...
%b	短的月名称，例如：(Jan, Feb, ...
%B	月名称全称，例如：January, February, ...
%c	日期和时间，例如：Thu Mar 3 23:05:25 2005
%d	每个月的某一天，例如：01 - 31
%e	每个月的某一天（没有填充0）例如：1,2,3,4---31
%D	日期；像：%m/%d/%y
%H	小时(00..23)
%I	小时(01..12)
%j	每个年的某一天，例如：001..366
%k	小时，例如：0..23
%m	月份，例如：01..12
%n	月份，例如：1..12
%M	分钟，例如：00..59
%p	am 或 pm
%s	从1970-01-01 00:00:00 UTC到现在的毫秒数
%S	秒，例如：00..59
%y	两位数的年份，例如：00..99
%Y	年份，例如：2010
%z	+hhmm 数字时区，例如：-0400

文件在使用的时候以".tmp"为后缀，一旦文件关闭，扩展名将被移除。
注：跟时间相关的转移序列，Key为“timestamp”必须存在在Event的Headers中（除非hdfs.useLocalTimeStamp设置为true）
NameDefaultDescription

channel	–	
type	–	组件的名称，必须为：HDFS
hdfs.path	–	HDFS目录路径，例如：hdfs://namenode/flume/webdata/
hdfs.filePrefix	FlumeData	HDFS目录中，由Flume创建的文件前缀。
hdfs.fileSuffix	–	追加到文件的后缀，例如：.txt
hdfs.inUsePrefix	–	文件正在写入时的前缀。
hdfs.inUseSuffix	.tmp	文件正在写入时的后缀。
hdfs.rollInterval	30	当前写入的文件滚动间隔，默认30秒生成一个新的文件 (0 = 不滚动)
hdfs.rollSize	1024	以文件大小触发文件滚动，单位字节(0 = 不滚动)
hdfs.rollCount	10	

		以写入的事件数触发文件滚动。(0 = 不滚动)
hdfs.idleTimeout	0	超时多久以后关闭无效的文件。(0 = 禁用自动关闭的空闲文件)但是还是可能因为网络等多种原因导致，正在写的文件始终没有关闭，从而产生tmp文件
hdfs.batchSize	100	有多少Event后，写到文件才刷新到HDFS。
hdfs.codeC	–	压缩编解码器，可以使用：gzip, bzip2, lzo, lzop, snappy
hdfs.fileType	SequenceFile	文件格式：通常使用SequenceFile（默认）,DataStream或者CompressedStream (1)DataStream不能压缩输出文件，请不用设置hdfs.codeC编码解码器。 (2)CompressedStream要求设置hdfs.codeC来制定一个有效的编码解码器。
hdfs.maxOpenFiles	5000	HDFS中允许打开文件的数据，如果数量超过了，最老的文件将被关闭。
hdfs.callTimeout	10000	允许HDFS操作的毫秒数，例如：open，write, flush, close。如果很多HFDS操作超时，这个配置应该增大。
hdfs.threadsPoolSize	10	每个HDFS sink的HDFS的IO操作线程数（例如：open，write）
hdfs.rollTimerPoolSize	1	每个HDFS sink调度定时文件滚动的线程数。
hdfs.kerberosPrincipal	–	安全访问HDFS Kerberos的主用户。
hdfs.kerberosKeytab	–	安全访问HDFSKerberos keytab
hdfs.proxyUser		
hdfs.round	false	时间戳应该被四舍五入。(如果为true，会影响所有的时间，除了t%)
hdfs.roundValue	1	四舍五入的最高倍数（单位配置在hdfs.roundUnit），但是要小于当前时间。
hdfs.roundUnit	second	四舍五入的单位，包含：second,minuteorhour.
hdfs.timeZone	Local Time	时区的名称，主要用来解决目录路径。例如：America/Los_Angeles
hdfs.useLocalTimeStamp	false	使用本地时间替换转义字符。（而不是event header的时间戳)
hdfs.closeTries	0	在发起一个关闭命令后，HDFS sink必须尝试重命名文件的次数。如果设置为1，重命名失败后，HDFS sink不会再次尝试重命名该文件，这个文件处于打开状态，并且用.tmp作为扩展名。如果为0，Sink会一直尝试重命名，直至重命名成功。如果文件 失败，这个文件可能一直保持打开状态，但是这种情况下数据是完整的。文件将会在Flume下次重启时被关闭。
hdfs.retryInterval	180	在几秒钟之间连续尝试关闭文件。每个关闭请求都会有多个RPC往返Namenode，因此设置的太低可能导致Namenode超负荷，如果设置0或者更小，如果第一次尝试失败的话，该Sink将不会尝试关闭文件。并且把文件打开，或者用“.tmp”作为扩展名。
serializer	TEXT	可能的选项包括avro_event或继承了EventSerializer.Builder接口的类名。
serializer.*		

关于round：

a1.sinks.k1.hdfs.round=true

a1.sinks.k1.hdfs.roundValue=10

a1.sinks.k1.hdfs.roundUnit=minute

上面的配置将四舍五入配置到10分钟，例如：一个事件的时间戳是11:54:34 AM, June 12, 2012 将导致hdfs的路径变为：/flume/events/2012-06-12/1150/00

源码分析

configure(Context context)：主要用于加载配置文件。

Java代码

```
public void configure(Context context) {
    this.context = context;
    //HDFS目录路径，例如：hdfs://namenode/flume/webdata/,也可以用/flume/webdata/，这样要把Hadoop的配置文件放到classpath
    filePath = Preconditions.checkNotNull(
        context.getString("hdfs.path"), "hdfs.path is required");
    //HDFS目录中，由Flume创建的文件前缀。
   fileName = context.getString("hdfs.filePrefix", defaultFileName);
    //文件后缀
   this.suffix = context.getString("hdfs.fileSuffix", defaultSuffix);
    //文件正在写入时的前缀。
   inUsePrefix = context.getString("hdfs.inUsePrefix", defaultInUsePrefix);//文件正在写入时的后缀。
```

```
  inUseSuffix = context.getString("hdfs.inUseSuffix", defaultInUseSuffix);
  //时区的名称，主要用来解决目录路径。例如：America/Los_Angeles
 String tzName = context.getString("hdfs.timeZone");
  timeZone = tzName == null ? null : TimeZone.getTimeZone(tzName);
  rollInterval = context.getLong("hdfs.rollInterval", defaultRollInterval);//当前写入的文件滚动间隔，默认30秒生成一个新的文件 (0 = 不滚动)
  rollSize = context.getLong("hdfs.rollSize", defaultRollSize);//以文件大小触发文件滚动，单位字节(0 = 不滚动)
  rollCount = context.getLong("hdfs.rollCount", defaultRollCount);
  //有多少Event后，写到文件才刷新到HDFS。
  batchSize = context.getLong("hdfs.batchSize", defaultBatchSize);
  //超时多久以后关闭无效的文件。(0 = 禁用自动关闭的空闲文件)但是还是可能因为网络等多种原因导致，正在写的文件始终没有关闭，从而产生tmp文件
  idleTimeout = context.getInteger("hdfs.idleTimeout", 0);
  //压缩编解码器，可以使用：gzip, bzip2, lzo, lzop, snappy
  String codecName = context.getString("hdfs.codeC");
 //文件格式：通常使用SequenceFile（默认）, DataStream 或者 CompressedStrea
  //(1)DataStream不能压缩输出文件，请不用设置hdfs.codeC编码解码器。
  //(2)CompressedStream要求设置hdfs.codeC来制定一个有效的编码解码器。
  fileType = context.getString("hdfs.fileType", defaultFileType);
  //HDFS中允许打开文件的数据，如果数量超过了，最老的文件将被关闭。
  maxOpenFiles = context.getInteger("hdfs.maxOpenFiles", defaultMaxOpenFiles);
  //允许HDFS操作的毫秒数，例如：open，write, flush, close。如果很多HFDS操作超时，这个配置应该增大。
  callTimeout = context.getLong("hdfs.callTimeout", defaultCallTimeout);
  //允许HDFS操作的毫秒数，例如：open，write, flush, close。如果很多HFDS操作超时，这个配置应该增大。
  //每个HDFS sink的HDFS的IO操作线程数（例如：open，write）
  threadsPoolSize = context.getInteger("hdfs.threadsPoolSize", defaultThreadPoolSize);
  //每个HDFS sink调度定时文件滚动的线程数。
  rollTimerPoolSize = context.getInteger("hdfs.rollTimerPoolSize", defaultRollTimerPoolSize);
  //每个HDFS sink调度定时文件滚动的线程数。
  String kerbConfPrincipal = context.getString("hdfs.kerberosPrincipal");
  //安全认证
String kerbKeytab = context.getString("hdfs.kerberosKeytab");
String proxyUser = context.getString("hdfs.proxyUser");
tryCount = context.getInteger("hdfs.closeTries", defaultTryCount);
if(tryCount <= 0) {
LOG.warn("Retry count value : " + tryCount + " is not " +
"valid. The sink will try to close the file until the file " +
"is eventually closed.");
tryCount = defaultTryCount;
}
retryInterval = context.getLong("hdfs.retryInterval",
defaultRetryInterval);
if(retryInterval <= 0) {
LOG.warn("Retry Interval value: " + retryInterval + " is not " +
"valid. If the first close of a file fails, " +
"it may remain open and will not be renamed.");
tryCount = 1;
}

Preconditions.checkArgument(batchSize > 0,
"batchSize must be greater than 0");
if (codecName == null) {
codeC = null;
compType = CompressionType.NONE;
} else {
codeC = getCodec(codecName);
// TODO : set proper compression type
compType = CompressionType.BLOCK;
}

// Do not allow user to set fileType DataStream with codeC together
// To prevent output file with compress extension (like .snappy)
if(fileType.equalsIgnoreCase(HDFSWriterFactory.DataStreamType)
&& codecName != null) {
throw new IllegalArgumentException("fileType: " + fileType +
" which does NOT support compressed output. Please don't set codeC" +
" or change the fileType if compressed output is desired.");
}

if(fileType.equalsIgnoreCase(HDFSWriterFactory.CompStreamType)) {
Preconditions.checkNotNull(codeC, "It's essential to set compress codec"
+ " when fileType is: " + fileType);
}

// get the appropriate executor
this.privExecutor = FlumeAuthenticationUtil.getAuthenticator(
kerbConfPrincipal, kerbKeytab).proxyAs(proxyUser);
```

```
   //时间戳应该被四舍五入。(如果为true，会影响所有的时间，除了t%)
 needRounding = context.getBoolean("hdfs.round", false);

 if(needRounding) {
     //四舍五入的单位
 String unit = context.getString("hdfs.roundUnit", "second");
 if (unit.equalsIgnoreCase("hour")) {
 this.roundUnit = Calendar.HOUR_OF_DAY;
 } else if (unit.equalsIgnoreCase("minute")) {
 this.roundUnit = Calendar.MINUTE;
 } else if (unit.equalsIgnoreCase("second")){
 this.roundUnit = Calendar.SECOND;
 } else {
 LOG.warn("Rounding unit is not valid, please set one of" +
 "minute, hour, or second. Rounding will be disabled");
 needRounding = false;
 }
     //四舍五入的最高倍数
 this.roundValue = context.getInteger("hdfs.roundValue", 1);
 if(roundUnit == Calendar.SECOND || roundUnit == Calendar.MINUTE){
 Preconditions.checkArgument(roundValue > 0 && roundValue <= 60,
 "Round value" +
 "must be > 0 and <= 60");
 } else if (roundUnit == Calendar.HOUR_OF_DAY){
 Preconditions.checkArgument(roundValue > 0 && roundValue <= 24,
 "Round value" +
 "must be > 0 and <= 24");
 }
 }

 this.useLocalTime = context.getBoolean("hdfs.useLocalTimeStamp", false);
 if(useLocalTime) {
 clock = new SystemClock();
 }

 if (sinkCounter == null) {
     //<span style="color:#000000;">计数器</span>
 sinkCounter = new SinkCounter(getName());
 }
 }
```

按照Flume的生命周期，先启动start方法：

Java代码

```
@Override
public void start() {
  String timeoutName = "hdfs-" + getName() + "-call-runner-%d";
  //线程池用于event写入HDFS文件
  callTimeoutPool = Executors.newFixedThreadPool(threadsPoolSize,
          new ThreadFactoryBuilder().setNameFormat(timeoutName).build());

  String rollerName = "hdfs-" + getName() + "-roll-timer-%d";
  //该线程池用来滚动文件
  timedRollerPool = Executors.newScheduledThreadPool(rollTimerPoolSize,
          new ThreadFactoryBuilder().setNameFormat(rollerName).build());
  //该LinkedHashMap用来存储文件的绝对路径以及对应的BucketWriter
  this.sfWriters = new WriterLinkedHashMap(maxOpenFiles);
  sinkCounter.start();
  super.start();
}
```

所有的Event，经Source后发送的Channel，再由Channel传入到Sink，主要调用Sink的process方法实现事务：

Java代码

```
public Status process() throws EventDeliveryException {
    Channel channel = getChannel();//获取Channel
    Transaction transaction = channel.getTransaction();//获取事务
    List<BucketWriter> writers = Lists.newArrayList();//初始化BucketWriter列表，BucketWriter是操作HDFS主类。
    transaction.begin();
    try {
      int txnEventCount = 0;
      for (txnEventCount = 0; txnEventCount < batchSize; txnEventCount++) {//批量处理
        Event event = channel.take();//获取Event
        if (event == null) {
          break;
        }
```

```
  // reconstruct the path name by substituting place holders
  String realPath = BucketPath.escapeString(filePath, event.getHeaders(),
      timeZone, needRounding, roundUnit, roundValue, useLocalTime);//格式化HDFS路径，根据转义字符
  String realName = BucketPath.escapeString(fileName, event.getHeaders(),
    timeZone, needRounding, roundUnit, roundValue, useLocalTime);//格式化文件名称，根据转义字符

  //写入HDFS的绝对路径
  String lookupPath = realPath + DIRECTORY_DELIMITER + realName;
  BucketWriter bucketWriter;
  HDFSWriter hdfsWriter = null;
  // Callback to remove the reference to the bucket writer from the
  // sfWriters map so that all buffers used by the HDFS file
  // handles are garbage collected.
  WriterCallback closeCallback = new WriterCallback() {
    @Override
    public void run(String bucketPath) {
      LOG.info("Writer callback called.");
      synchronized (sfWritersLock) {
        sfWriters.remove(bucketPath);
      }
    }
  };
  synchronized (sfWritersLock) {
    //根据HDFS的绝对路径获取对应的BucketWriter对象
    bucketWriter = sfWriters.get(lookupPath);
    // we haven't seen this file yet, so open it and cache the handle
    if (bucketWriter == null) {
      //初始化BuchetWriter对象
      hdfsWriter = writerFactory.getWriter(fileType);
      bucketWriter = initializeBucketWriter(realPath, realName,
        lookupPath, hdfsWriter, closeCallback);
      //放入Map
      sfWriters.put(lookupPath, bucketWriter);
    }
  }

  // track the buckets getting written in this transaction
  if (!writers.contains(bucketWriter)) {
    //如果BucketWriter列表没有正在写的文件——bucketWriter，则加入
    writers.add(bucketWriter);
  }

  // Write the data to HDFS
  try {
    //将event写入bucketWriter对应的文件中
    bucketWriter.append(event);
  } catch (BucketClosedException ex) {
    LOG.info("Bucket was closed while trying to append, " +
      "reinitializing bucket and writing event.");
    hdfsWriter = writerFactory.getWriter(fileType);
    bucketWriter = initializeBucketWriter(realPath, realName,
      lookupPath, hdfsWriter, closeCallback);
    synchronized (sfWritersLock) {
      sfWriters.put(lookupPath, bucketWriter);
    }
    bucketWriter.append(event);
  }
}

if (txnEventCount == 0) {
  //这次事务没有处理任何event
  sinkCounter.incrementBatchEmptyCount();
} else if (txnEventCount == batchSize) {
  //一次处理batchSize个event
  sinkCounter.incrementBatchCompleteCount();
} else {
  //channel中剩余的events不足batchSize
  sinkCounter.incrementBatchUnderflowCount();
}

// flush all pending buckets before committing the transaction
//获取List里面的BucketWriter的所有数据都刷新到HDFS
for (BucketWriter bucketWriter : writers) {
  //如果使用转义字符生成文件名或路径，可能还没有满足其他滚动生成新文件的条件，就有新文件产生，
  //在这种情况下，例如为hdfs.idleTimeout=0，那么就可能会在HDFS中出现很多.tmp后缀的文件。因为调用flush没有关闭该文件。
  bucketWriter.flush();
}
```

```
      //提交事务
      transaction.commit();

      if (txnEventCount < 1) {
        return Status.BACKOFF;
      } else {
        sinkCounter.addToEventDrainSuccessCount(txnEventCount);
        return Status.READY;
      }
    } catch (IOException eIO) {
      transaction.rollback();//事务回滚
      LOG.warn("HDFS IO error", eIO);
      return Status.BACKOFF;
    } catch (Throwable th) {
      transaction.rollback();
      LOG.error("process failed", th);
      if (th instanceof Error) {
        throw (Error) th;
      } else {
        throw new EventDeliveryException(th);
      }
    } finally {
      transaction.close();//关闭事务
    }
  }
```

HDFS Sink流程分析：

1，通过configure(Context context)和start()方法初始化Sink

2，SinkRunner的线程调用process()方法，循环处理批量的Event，如果Event为null，就跳出循环。

3，有Event数据，先格式化HDFS的文件路径和文件名，即：realPath和realName。realPath+realName就是完整HDFS路径：lookupPath，然后根据lookupPath获取BucketWriter对象。

4，BucketWriter对象不存在，则先构建根据fileType构建一个HDFSWriter 对象。然后初始化BucketWriter对象。最后将对象放到sfWriters中，表示正在写的文件。

Java代码

```
public HDFSWriter getWriter(String fileType) throws IOException {
  if (fileType.equalsIgnoreCase(SequenceFileType)) {
    //通过SequenceFile.Writer写入文件
    return new HDFSSequenceFile();
  } else if (fileType.equalsIgnoreCase(DataStreamType)) {
    //通过FSDataOutputStream
    return new HDFSDataStream();
  } else if (fileType.equalsIgnoreCase(CompStreamType)) {
    return new HDFSCompressedDataStream();
  } else {
    throw new IOException("File type " + fileType + " not supported");
  }
}
```

HDFSSequenceFile：configure(context)方法会首先获取写入格式writeFormat即参数"hdfs.writeFormat"，org.apache.flume.sink.hdfs.SequenceFileSerializerType定义了一下三个：

Java代码

```
Writable(HDFSWritableSerializer.Builder.class),//默认的
Text(HDFSTextSerializer.Builder.class),
Other(null);
```

再获取是否使用HDFS本地文件系统"hdfs.useRawLocalFileSystem"，默认是flase不使用；然后获取writeFormat的所有配置信息serializerContext；然后根据writeFormat和serializerContext构造SequenceFileSerializer的对象serializer。

HDFSDataStream：configure(context)方法先获取serializerType类型，默认是TEXT(BodyTextEventSerializer.Builder.class)，其他的还包含：

Java代码

```
public enum EventSerializerType {
  TEXT(BodyTextEventSerializer.Builder.class),
  HEADER_AND_TEXT(HeaderAndBodyTextEventSerializer.Builder.class),
  AVRO_EVENT(FlumeEventAvroEventSerializer.Builder.class),
  OTHER(null);
```

再获取是否使用HDFS本地文件系统"hdfs.useRawLocalFileSystem"，默认是flase不使用；最后获取serializer的所有配置信息serializerContext。serializer的实例化在HDFSDataStream.doOpen(Configuration conf, Path dstPath, FileSystem hdfs)方法中实现的。

HDFSCompressedDataStream：configure和HDFSDataStream.configure(context)类似，serializerType的类型也一样。serializer的实例化是在HDFSCompressedDataStream.open(String filePath, CompressionCodec codec, CompressionType cType)方法中实现。

5，bucketWriter实例化后存放到sfWriters中，并且判断是否在writers变量的List中，如果不存在，就放入List，这样后面就可以对bucketWriter统一flush了。

6，bucketWriter.append(event);

Java代码

```
public synchronized void append(final Event event)
        throws IOException, InterruptedException {
```

```
checkAndThrowInterruptedException();//检查当前线程是否被中断
// If idleFuture is not null, cancel it before we move forward to avoid a
// close call in the middle of the append.
if(idleFuture != null) {
  idleFuture.cancel(false);
  // There is still a small race condition - if the idleFuture is already
  // running, interrupting it can cause HDFS close operation to throw -
  // so we cannot interrupt it while running. If the future could not be
  // cancelled, it is already running - wait for it to finish before
  // attempting to write.
  if(!idleFuture.isDone()) {
    try {
      idleFuture.get(callTimeout, TimeUnit.MILLISECONDS);
    } catch (TimeoutException ex) {
      LOG.warn("Timeout while trying to cancel closing of idle file. Idle" +
        " file close may have failed", ex);
    } catch (Exception ex) {
      LOG.warn("Error while trying to cancel closing of idle file. ", ex);
    }
  }
  idleFuture = null;
}

// If the bucket writer was closed due to roll timeout or idle timeout,
// force a new bucket writer to be created. Roll count and roll size will
// just reuse this one
if (!isOpen) {
  if (closed) {
    throw new BucketClosedException("This bucket writer was closed and " +
      "this handle is thus no longer valid");
  }
  open();//一个文件已经完成将isOpen设置为false，则新建一个文件
}

// check if it's time to rotate the file
if (shouldRotate()) {//检查文件的行数及大小，判断是否要关闭文件后重新生成文件。
  boolean doRotate = true;

  if (isUnderReplicated) {
    if (maxConsecUnderReplRotations > 0 &&
        consecutiveUnderReplRotateCount >= maxConsecUnderReplRotations) {
      doRotate = false;
      if (consecutiveUnderReplRotateCount == maxConsecUnderReplRotations) {
        LOG.error("Hit max consecutive under-replication rotations ({}); " +
            "will not continue rolling files under this path due to " +
            "under-replication", maxConsecUnderReplRotations);
      }
    } else {
      LOG.warn("Block Under-replication detected. Rotating file.");
    }
    consecutiveUnderReplRotateCount++;
  } else {
    consecutiveUnderReplRotateCount = 0;
  }

  if (doRotate) {
    close();
    open();//新建一个文件
  }
}

// write the event
try {
  sinkCounter.incrementEventDrainAttemptCount();
  callWithTimeout(new CallRunner<Void>() {
    @Override
    public Void call() throws Exception {
      writer.append(event); // could block 往HDFS写入数据。
      return null;
    }
  });
} catch (IOException e) {
  LOG.warn("Caught IOException writing to HDFSWriter ({}). Closing file (" +
      bucketPath + ") and rethrowing exception.",
      e.getMessage());
  try {
    close(true);
  } catch (IOException e2) {
    LOG.warn("Caught IOException while closing file (" +
        bucketPath + "). Exception follows.", e2);
      }
      throw e;
    }

    // update statistics
    processSize += event.getBody().length;
    eventCounter++;
    batchCounter++;

    if (batchCounter == batchSize) {
      flush();
    }
  }
```

打开新文件分为两类：

第一类不需要压缩

Java代码

```
public void open(String filePath) throws IOException {
  open(filePath, null, CompressionType.NONE);
}
```

第二类要压缩

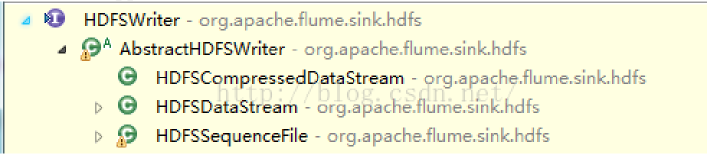
Java代码

```
public void open(String filePath, CompressionCodec codeC,
    CompressionType compType) throws IOException {
  Configuration conf = new Configuration();
  Path dstPath = new Path(filePath);
  FileSystem hdfs = dstPath.getFileSystem(conf);
  open(dstPath, codeC, compType, conf, hdfs);
}
```

注：HDFSDataStream是不支持压缩的，所以直接调用第一类的open方法。

在open方法中，如果按时间滚动的rollInterval不为0，则创建Callable，放入timedRollFuture中rollInterval秒之后关闭文件，默认是30s写一个文件。

最后writer.append(event)是真正写数据到HDFS，writer分如下三种情况：



```
HDFSWriter - org.apache.flume.sink.hdfs
  AbstractHDFSWriter - org.apache.flume.sink.hdfs
    HDFSCompressedDataStream - org.apache.flume.sink.hdfs
    HDFSDataStream - org.apache.flume.sink.hdfs
    HDFSSequenceFile - org.apache.flume.sink.hdfs
```

HDFSSequenceFile：append(event)方法，会先通过serializer.serialize(e)把event处理成一个Key和一个Value。

serializer为HDFSWritableSerializer：

Key：

Java代码

```
private Object getKey(Event e) {
    String timestamp = e.getHeaders().get("timestamp");//获取header的timesteamp
    long eventStamp;

    if (timestamp == null) {//timestamp不存在就拿系统的当前时间
      eventStamp = System.currentTimeMillis();
    } else {
      eventStamp = Long.valueOf(timestamp);
    }
    return new LongWritable(eventStamp);//将时间封装成LongWritable
  }
```

Value：

Java代码

```
private BytesWritable makeByteWritable(Event e) {
  BytesWritable bytesObject = new BytesWritable();
  bytesObject.set(e.getBody(), 0, e.getBody().length);
  return bytesObject;
}
```

serializer为HDFSTextSerializer：

Key同上，Value：

Java代码

```
private Text makeText(Event e) {
  Text textObject = new Text();
  textObject.set(e.getBody(), 0, e.getBody().length);
  return textObject;
}
```

writer为HDFSDataStream：

直接调用serializer.write(e)，serializer分三种：

org.apache.flume.serialization.BodyTextEventSerializer直接读取body写入OutputStream流中，然后在最后加"\n"。

org.apache.flume.serialization.HeaderAndBodyTextEventSerializer将e.getHeaders() + " " +e.getBody()写入数据流，然后根据配置看是否要加"\n"
org.apache.flume.serialization.AvroEventSerializer将event整体写入dataFileWriter。

然后appned方法更新统计，processSize统计文件大小；eventCounter统计文件行数；batchCounter是统计最近一次flush之后的处理的event数；
如果处理的event数量达到batchSize的大小，则刷新到HDFS，flush()方法会首先执行writer.sync()即写入HDFS，然后将batchCounter置为0，根据fileType的不同writer也会有很多写入类型：

HDFSSequenceFile：sync()方法执行SequenceFile.Writer.syncFs()将数据写入HDFS中；
HDFSDataStream：sync()方法执行
HDFSCompressedDataStream：sync()方法先执行serializer.flush()：只有FlumeEventAvroEventSerializer的flush()方法也有实现dataFileWriter.flush()，其他俩BodyTextEventSerializer和HeaderAndBodyTextEventSerializer均未实现flush()方法。然后执行outStream.flush()和outStream.sync()将数据刷新至HDFS中。

7，回到HDFSEventSink.process()方法中，会根据这次事务处理的event数量更新相应的统计；
8，遍历writers，挨个刷新BucketWriter至HDFS；
9，最后提交事务，异常回滚，关闭事务。
最后停止：
Java代码

```
@Override
public void stop() {
  // do not constrain close() calls with a timeout
  synchronized (sfWritersLock) {//获取对象锁
    //遍历对象锁
    for (Entry<String, BucketWriter> entry : sfWriters.entrySet()) {
      LOG.info("Closing {}", entry.getKey());
      //关闭BucketWriter，flush到HDFS
      try {
        entry.getValue().close();
      } catch (Exception ex) {
        LOG.warn("Exception while closing " + entry.getKey() + ". " +
            "Exception follows.", ex);
        if (ex instanceof InterruptedException) {
          Thread.currentThread().interrupt();
        }
      }
    }
  }

  // shut down all our thread pools
  ExecutorService toShutdown[] = {callTimeoutPool, timedRollerPool};
  for (ExecutorService execService : toShutdown) {
    execService.shutdown();
    try {
      while (execService.isTerminated() == false) {
        execService.awaitTermination(
            Math.max(defaultCallTimeout, callTimeout), TimeUnit.MILLISECONDS);
      }
    } catch (InterruptedException ex) {
      LOG.warn("shutdown interrupted on " + execService, ex);
    }
  }

  callTimeoutPool = null;
  timedRollerPool = null;

  synchronized (sfWritersLock) {
    sfWriters.clear();
    sfWriters = null;
  }
  sinkCounter.stop();
  super.stop();
}
```

SpoolDirectorySource使用及源码分析 | Flume几种监控方式

Global site tag (gtag.js) - Google Analytics