Final Phantasy
DeLucia

FLUME 1.9用户手册

- 英文原版文档： Flume 1.9 User Guide
- 中文文档在线查阅： https://flume.liyifeng.org
- 中文文档离线下载：https://flume.liyifeng.org/down
- Flume 1.9新版更新内容请看 Flume v1.9.0 发行日志

简介

概览

Apache Flume 是一个分布式、高可靠、高可用的用来收集、聚合、转移不同来源的大量日志数据到中央数据仓库的工具

Apache Flume是Apache软件基金会（ASF）的顶级项目

系统要求

1. Java运行环境 – Java 1.8或更高版本
2. 内存 – 足够的内存 用来配置Souuces、Channels和Sinks
3. 硬盘空间 – 足够的硬盘用来配置Channels 和 Sinks
4. 目录权限 – Agent用来读写目录的权限

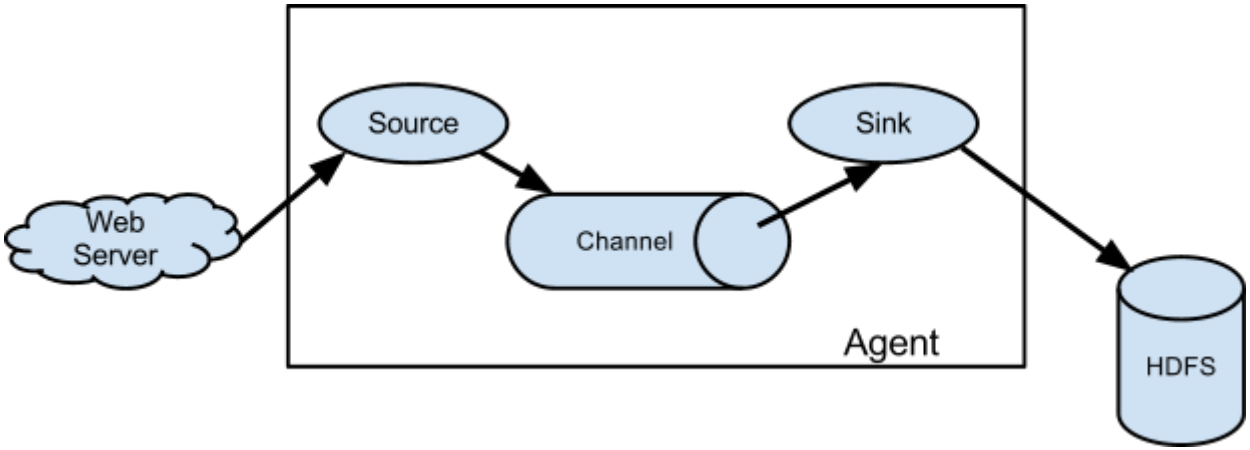
体系结构

数据流模型

Event是Flume定义的一个数据流传输的最小单元。Agent就是一个Flume的实例，本质是一个JVM进程，该JVM进程控制Event数据流从外部日志生产者那里传输到目的地（或者是下一个Agent）。

> *提示*
>
> *学习Flume必须明白这几个概念，Event英文直译是事件，但是在Flume里表示数据传输的一个最小单位（被Flume收集的一条条日志又或者一个个的二进制文件，不管你在外面叫什么，进入Flume之后它就叫event）。参照下图可以看得出Agent就是Flume的一个部署实例， 一个完整的Agent中包含了必须的三个组件*

> *Source、Channel和Sink，Source是指数据的来源和方式，Channel是一个数据的缓冲池，Sink定义了数据输出的方式和目的地（这三个组件是必须有的，另外还有很多可选的组件interceptor、channel selector、sink processor等后面会介绍）。*



Source消耗由外部（如Web服务器）传递给它的Event。外部以Flume Source识别的格式向Flume发送Event。例如，Avro Source 可接收从Avro客户端（或其他FlumeSink）接收Avro Event。用 Thrift Source 也可以实现类似的流程，接收的Event数据可以是任何语言编写的只要符合Thrift协议即可。

当Source接收Event时，它将其存储到一个或多个channel。该channel是一个被动存储器（或者说叫存储池），可以存储Event直到它被Sink消耗。『文件channel』就是一个例子 – 它由本地文件系统支持。sink从channel中移除Event并将其放入外部存储库（如HDFS，通过Flume的 HDFS Sink 实现）或将其转发到流中下一个Flume Agent（下一跳）的Flume Source。

Agent中的source和sink与channel存取Event是异步的。

Flume的Source负责消费外部传递给它的数据（比如web服务器的日志）。外部的数据生产方以Flume Source识别的格式向Flume发送Event。

> *提示*
>
> *“Source消耗由外部传递给它的Event”，这句话听起来好像Flume只能被动接收Event，实际上Flume也有Source是主动收集Event的，比如：Spooling Directory Source 、Taildir Source 。*

复杂流

Flume可以设置多级Agent连接的方式传输Event数据。也支持扇入和扇出的部署方式，类似于负载均衡方式或多点同时备份的方式。

> *提示*
>
> *这里必须解释一下，第一句的意思是可以部署多个Agent组成一个数据流的传输链。第二句要知道扇入（多对一）和扇出（一对多）的概念，就是说Agent可以将数据流发到多个下级Agent，也可以从多个Agent发到一个Agent中。*
>
> *也就是，你可以根据自己的业务需求来任意组合传输日志的Agent流，引用一张后面章节的图，这就是一个扇入方式的Flume部署方式，前三个Agent的数据都汇总到一个Agent4上，最后由Agent4统一存储到HDFS。*

> *提示*
>
> *官方这个图的Agent4的Sink画错了，不应该是 Avro Sink ，应该是 HDFS Sink 。*

可靠性

Event会在每个Agent的Channel上进行缓存，随后Event将会传递到流中的下一个Agent或目的地（比如HDFS）。只有成功地发送到下一个Agent或目的地后Event才会从Channel中删除。这一步保证了Event数据流在Flume Agent中传输时端到端的可靠性。

> *提示*
>
> *Flume的这个channel最重要的功能是用来保证数据的可靠传输的。其实另外一个重要的功能也不可忽视，就是实现了数据流入和流出的异步执行。*
>
> *Flume使用事务来保证Event的 可靠传输。Source和Sink对Channel提供的每个Event数据分别封装一个事务用于存储和恢复，这样就保证了数据流的集合在点对点之间的可靠传输。在多层架构的情况下，来自前一层的sink和来自下一层的Source 都会有事务在运行以确保数据安全地传输到下一层的Channel中。*

可恢复性

Event数据会缓存在Channel中用来在失败的时候恢复出来。Flume支持保存在本地文件系统中的『文件channel』，也支持保存在内存中的『内存Channel』，『内存Channel』显然速度会更快，缺点是万一Agent挂掉『内存Channel』中缓存的数据也就丢失了。

安装

开始安装第一个Agent

Flume Agent的配置是在一个本地的配置文件中。这是一个遵循Java properties文件格式的文本文件。一个或多个Agent配置可放在同一个配置文件里。配置文件包含Agent的source，sink和channel的各个属性以及他们的数据流连接。

第一步：配置各个组件

每个组件（source，sink或者channel）都有一个name，type和一系列的基于其type或实例的属性。例如，一个avro source需要有个hostname（或者ip地址）一个端口号来接收数据。一个内存channel有最大队列长度的属性（capacity）， 一个HDFS sink需要知晓文件系统的URI地址创建文件，文件访问频率（ hdfs.rollInterval ）等等。所有的这些组件属性都需要在Flume配置文件中设置。

第二步：连接各个组件

Agent需要知道加载什么组件，以及这些组件在流中的连接顺序。通过列出在Agent中的source，sink和channel名称，定义每个sink和source的channel来完成。

> *提示*

> *本来上面这段原文中描述了一个例子，可是并不直观，不如直接看下面hello world里面的配置例子。*

第三步：启动Agent

bin目录下的flume-ng是Flume的启动脚本，启动时需要指定Agent的名字、配置文件的目录和配置文件的名称。

比如这样:

```
$ bin/flume-ng agent -n $agent_name -c conf -f conf/flume-conf.properties.template
```

到此，Agent就会运行flume-conf.properties.template里面配置的source和sink了。

一个简单的Hello World

这里给出了一个配置文件的例子，部署一个单节点的Flume，这个配置是让你自己生成Event数据然后Flume会把它们输出到控制台上。

> *提示*
>
> *下面的配置文件中，source使用的是 NetCat TCP Source，这个Source在后面会有专门的一节来介绍，简单说就是监听本机上某个端口上接收到的TCP协议的消息，收到的每行内容都会解析封装成一个Event，然后发送到channel，*
>
> *sink使用的是 Logger Sink，这个sink可以把Event输出到控制台，channel使用的是 Memory Channel ，是一个用内存作为Event缓冲的channel。*

Flume内置了多种多样的source、sink和channel，后面 配置 章节会逐一介绍。

```
<em># example.conf: 一个单节点的 Flume 实例配置</em>

<em># 配置Agent a1各个组件的名称</em>
a1.sources = r1     #Agent a1 的source有一个，叫做r1
a1.sinks = k1       #Agent a1 的sink也有一个，叫做k1
a1.channels = c1    #Agent a1 的channel有一个，叫做c1

<em># 配置Agent a1的source r1的属性</em>
a1.sources.r1.type = netcat       #使用的是NetCat TCP Source，这里配的是别名，Flume内置的一些组件都是有别名的，没有别名填全限定类名
```

```
a1.sources.r1.bind = localhost    #NetCat TCP Source监听的hostname，这个是本机
a1.sources.r1.port = 44444        #监听的端口

<em># 配置Agent a1的sink k1的属性</em>
a1.sinks.k1.type = logger         # sink使用的是Logger Sink，这个配的也是别名

<em># 配置Agent a1的channel c1的属性，channel是用来缓冲Event数据的</em>
a1.channels.c1.type = memory                #channel的类型是内存channel，顾名思义这个channel是使用内存来缓冲数据
a1.channels.c1.capacity = 1000              #内存channel的容量大小是1000，注意这个容量不是越大越好，配置越大一旦Flume挂掉丢失的ev
a1.channels.c1.transactionCapacity = 100    #source和sink从内存channel每次事务传输的event数量

<em># 把source和sink绑定到channel上</em>
a1.sources.r1.channels = c1       #与source r1绑定的channel有一个，叫做c1
a1.sinks.k1.channel = c1          #与sink k1绑定的channel有一个，叫做c1
```

配置文件里面的注释已经写的很明白了，这个配置文件定义了一个Agent叫做a1，a1有一个source监听本机44444端口上接收到的数据、一个缓冲数据的channel还有一个把Event数据输出到控制台的sink。这个配置文件给各个组件命名，并且设置了它们的类型和其他属性。通常一个配置文件里面可能有多个Agent，当启动Flume时候通常会传一个Agent名字来做为程序运行的标记。

> *提示*
>
> *同一个配置文件中如果配置了多个agent流，启动Flume的命令中 --name 这个参数的作用就体现出来了，用它来告诉Flume将要启动该配置文件中的哪一个agent实例。*

用下面的命令加载这个配置文件启动Flume：

```
$ bin/flume-ng agent --conf conf --conf-file example.conf --name a1 -Dflume.root.logger=INFO,console
```

请注意，在完整的部署中通常会包含 –conf=<conf-dir>这个参数，<conf-dir>目录里面包含了flume-env.sh和一个log4j properties文件，在这个例子里面，我们强制Flume把日志输出到了控制台，运行的时候没有任何自定义的环境脚本。

测试一下我们的这个例子吧，打开一个新的终端窗口，用telnet命令连接本机的44444端口，然后输入Hello world！后按回车，这时收到服务器的响应[OK]（这是 NetCat TCP Source 默认给返回的），说明一行数据已经成功发送。

```
$ telnet localhost 44444
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
Hello world! <ENTER>
OK
```

Flume的终端里面会以log的形式输出这个收到的Event内容。

```
12/06/19 15:32:19 INFO source.NetcatSource: Source starting
12/06/19 15:32:19 INFO source.NetcatSource: Created serverSocket:sun.nio.ch.ServerSocketChannelImpl[/127.0.0.1:44444]
12/06/19 15:32:34 INFO sink.LoggerSink: Event: { headers:{} body: 48 65 6C 6C 6F 20 77 6F 72 6C 64 21 0D          Hello
```

恭喜你！到此你已经成功配置并运行了一个Flume Agent，接下来的章节我们会介绍更多关于Agent的配置。

在配置文件里面自定义环境变量

Flume可以替换配置文件中的环境变量，例如：

```
a1.sources = r1
a1.sources.r1.type = netcat
a1.sources.r1.bind = 0.0.0.0
a1.sources.r1.port = ${NC_PORT}
a1.sources.r1.channels = c1
```

> *警告*
>
> *注意了，目前只允许在value里面使用环境变量（也就是说只能在等号右边用，左边不行）*

启动Agent时候加上 *propertiesImplementation = org.apache.flume.node.EnvVarResolverProperties* 就可以了。

例如：

```
$ NC_PORT=44444 bin/flume-ng agent --conf conf --conf-file example.conf --name a1 -Dflume.root.logger=INFO,console -Dpr
```

> *警告*
>
> *上面仅仅是个例子，环境变量可以用其他方式配置，比如在conf/flume-env.sh里面设置。*

输出原始数据到日志

通常情况下在生产环境下记录数据流中的原始数据到日志是不可取的行为，因为可能泄露敏感信息或者是安全相关的配置，比如秘钥之类的。默认情况下Flume不会向日志中输出这些信息，如果Flume出了异常，Flume会尝试提供调试错误的线索。

有一个办法能把原始的数据流都输出到日志，就是配置一个额外的内存Channel（ Memory Channel ） 和 Logger Sink ，Logger Sink可以输出所有的Event数据到Flume的日志，然而这个方法并不是适用所有情况。

为了记录Event和配置相关的数据，必须设置一些java系统属性在log4j配置文件中。

为了记录配置相关的日志，可以通过-Dorg.apache.flume.log.printconfig=true来开启，可以在启动脚本或者flume-env.sh的JAVA_OPTS来配置这个属性。

通过设置-Dorg.apache.flume.log.rawdata=true来开启记录原始日志，对于大多数组件log4j的日志级别需要设置到DEBUG或者TRACE才能保证日志能输出到Flume的日志里面。

下面这个是开启记录Event原始数据，并且设置logj的日志级别为DEBUG的输出到console的脚本

```
$ bin/flume-ng agent --conf conf --conf-file example.conf --name a1 -Dflume.root.logger=DEBUG,console -Dorg.apache.flu
```

基于Zookeeper的配置

Flume支持使用Zookeeper配置Agent。**这是个实验性的功能。**配置文件需要上传到zookeeper中，在一个可配置前缀下。配置文件存储在Zookeeper节点数据里。下面是a1 和 a2 Agent在Zookeeper节点树的配置情况。

```
- /flume
  |- /a1 [Agent config file]
  |- /a2 [Agent config file]
```

上传好了配置文件后，可以使用下面的脚本参数进行启动：

```
$ bin/flume-ng agent --conf conf -z zkhost:2181,zkhost1:2181 -p /flume --name a1 -Dflume.root.logger=INFO,console
```

参数名	默认值	描述
z	–	Zookeeper的连接，hostname:port格式，多个用逗号分开
p	/flume	Zookeeper中存储Agent配置的目录

安装第三方插件

Flume有完整的插件架构。尽管Flume已经提供了很多现成的source、channel、sink、serializer可用。

然而通过把自定义组件的jar包添加到flume-env.sh文件的FLUME_CLASSPATH 变量中使用自定义的组件也是常有的事。现在Flume支持在一个特定的文件夹自动获取组件，这个文件夹就是pluguins.d。这样使得插件的包管理、调试、错误定位更加容易方便，尤其是依赖包的冲突处理。

plugins.d文件夹

`plugins.d` 文件夹的所在位置是 *$FLUME_HOME/plugins.d*，在启动时 *flume-ng* 会启动脚本检查这个文件夹把符合格式的插件添加到系统中。

插件的目录结构

每个插件（也就是 `plugins.d` 下的子文件夹）都可以有三个子文件夹：

1. lib – 插件自己的jar包
2. libext – 插件依赖的其他所有jar包
3. native – 依赖的一些本地库文件，比如 *.so* 文件

下面是两个插件的目录结构例子：

```
plugins.d/
plugins.d/custom-source-1/
plugins.d/custom-source-1/lib/my-source.jar
plugins.d/custom-source-1/libext/spring-core-2.5.6.jar
plugins.d/custom-source-2/
plugins.d/custom-source-2/lib/custom.jar
plugins.d/custom-source-2/native/gettext.so
```

数据获取方式

Flume支持多种从外部获取数据的方式。

RPC

Flume发行版中包含的Avro客户端可以使用avro RPC机制将给定文件发送到Flume Avro Source：

```
$ bin/flume-ng avro-client -H localhost -p 41414 -F /usr/logs/log.10
```

上面的命令会将/usr/logs/log.10的内容发送到监听该端口的Flume Source。

执行命令

Flume提供了一个 Exec Source ，通过执行系统命令来获得持续的数据流，按照\r或者\n或者\r\n（\n\r）来划分数据行，然后把每行解析成为一个Event。

网络流

Flume支持以下比较流行的日志类型读取：

1. Avro
2. Thrift
3. Syslog
4. Netcat

> *提示*
>
> *除了前面的rpc、系统命令、网络流，还有一类很重要的Source就是从文件获取数据，比如 Spooling Directory Source 和 Taildir Source ，可以用它们来监控应用服务产生的日志并进行收集。*

多Agent的复杂流

> *提示*
>
> *这一小节介绍了几种典型的Flume的多Agent以及一个Agent中多路输出等部署方式。*

这个例子里面为了能让数据流在多个Agent之间传输，前一个Agent的sink必须和后一个Agent的source都需要设置为avro类型并且指向相同的hostname（或者IP）和端口。

组合

日志收集场景中比较常见的是数百个日志生产者发送数据到几个日志消费者Agent上，然后消费者Agent负责把数据发送到存储系统。例如从数百个web服务器收集的日志发送到十几个Agent上，然后由十几个Agent写入到HDFS集群。

可以通过使用 Avro Sink 配置多个第一层 Agent（Agent1、Agent2、Agent3），所有第一层Agent的Sink都指向下一级同一个Agent（Agent4）的 Avro Source上（同样你也可以使用 thrift 协议的 Source 和 Sink 来代替）。Agent4 上的 Source 将 Event 合并到一个 channel 中，该 channel中的Event最终由HDFS Sink 消费发送到最终目的地。

> *提示*
>
> *官方这个图的Agent4的Sink画错了，不应该是 Avro Sink ，应该是 HDFS Sink 。*

多路复用流

Flume支持多路复用数据流到一个或多个目的地。这是通过使用一个流的[多路复用器]（multiplexer）来实现的，它可以 **复制** 或者 **选择（多路复用）** 数据流到一个或多个channel上。

> *提示*
>
> *很容易理解， 复制 就是每个channel的数据都是完全一样的，每一个channel上都有完整的数据流集合。 选择（多路复用） 就是通过自定义一个分配机制，把数据流拆分到多个channel上。后面有详细介绍，请参考 Flume Channel Selectors 。*

上图的例子展示了从Agent foo扇出流到多个channel中。这种扇出的机制可以是 **复制** 或者 **选择（多路复用）** 。当配置为复制的时候，每个Event都会被发送到3个channel上。当配置为选择（多路复用）的时候，当Event的某个属性与配置的值相匹配时会被发送到对应的

channel。

例如Event的属性txnType是customer时，Event被发送到channel1和channel3，如果txnType的值是vendor时，Event被发送到channel2，其他值一律发送到channel3，这种规则是可以通过配置来实现的。

> *提示*
>
> *好了做一个总结吧，本章内容是这个文档最重要的一章，让你知道Flume都有哪些组件、配置方式、启动方式、使用第三方插件、以及一些实际使用中的复杂流的部署方案等等。下一章开始逐个介绍每一个组件。*

配置

如前面部分所述，Flume Agent程序配置是从类似于具有分级属性设置的Java属性文件格式的文件中读取的。

> *提示*
>
> *这一章开始详细介绍Flume的source、sink、channel三大组件和其他几个组件channel selector、sink processor、serializer、interceptor的配置、使用方法和各自的适用范围。 如果硬要翻译这些组件的话，三大组件分别是数据源（source）、数据目的地（sink）和缓冲池（channel）。其他几个分别是Event多路复用的channel选择器（channel selector）， Sink组逻辑处理器（sink processor）、序列化器（serializer）、拦截器（interceptor）。*

定义流

要在单个Agent中定义流，你需要通过channel连接source和sink。需要在配置文件中列出所有的source、sink和channel，然后将source和sink指向channel。一个source可以连接多个channel，但是sink只能连接一个channel。格式如下：

> *提示*
>
> *一个agent实例里面可以有多条独立的数据流，一个数据流里必须有且只能有一个Source，但是可以有多个Channel和Sink，Source和Channel是一对多的关系，Channel和Sink也是一对多的关系。*

Source和Channel的一对多就是前面 多路复用流 里面介绍的，Source收集来的数据可以内部拷贝多份到多个Channel上，也可以将event按照某些规则分发到多个Channel上；

Channel和Sink的一对多在后面 Sink组逻辑处理器 有体现，Sink其实就是Channel里面event的消费者，当然就可以创建多个Sink一同消费Channel队列中的数据，并且还能进行自定义这些Sink的工作方式，具体请看该章节内容；

请注意上面这段话里我说的是【一个数据流】，我可没说【一个agent】或【同一个配置文件】里只能有一个Souce，因为一个agent里面可以有多条独立的数据流，多个agent实例的配置又可以都配在同一个配置文件里。

```
<em># 列出Agent实例的所有Source、Channel、Sink</em>
<Agent>.sources = <Source>
<Agent>.sinks = <Sink>
<Agent>.channels = <Channel1> <Channel2>

<em># 设置Channel和Source的关联</em>
<Agent>.sources.<Source>.channels = <Channel1> <Channel2> ...     # 这行配置就是给一个Source配置了多个channel

<em># 设置Channel和Sink的关联</em>
<Agent>.sinks.<Sink>.channel = <Channel1>
```

例如，一个叫做agent_foo的Agent从外部avro客户端读取数据并通过内存channel将其发送到HDFS（准确说并不是通过内存channel发送的数据，而是使用内存channel缓存，然后通过HDFS Sink从channel读取后发送的），它的配置文件应该这样配：

```
<em># 列出Agent的所有source、sink和channel</em>
agent_foo.sources = avro-appserver-src-1
agent_foo.sinks = hdfs-sink-1
agent_foo.channels = mem-channel-1

agent_foo.sources.avro-appserver-src-1.channels = mem-channel-1    # 指定与source avro-appserver-src-1 相连接的channel是me
agent_foo.sinks.hdfs-sink-1.channel = mem-channel-1                # 指定与sink hdfs-sink-1 相连接的channel是mem-channel-1
```

通过上面的配置，就形成了[avro-appserver-src-1]->[mem-channel-1]->[hdfs-sink-1]的数据流，这将使Event通过内存channel（mem-channel-1）从avro-appserver-src-1流向hdfs-sink-1，当Agent启动时，读取配置文件实例化该流。

配置单个组件

定义流后，需要配置source、sink和channel各个组件的属性。配置的方式是以相同的分级命名空间的方式，你可以设置各个组件的类型以及基于其类型特有的属性。

```
<em># properties for sources</em>
<Agent>.sources.<Source>.<someProperty> = <someValue>

<em># properties for channels</em>
<Agent>.channel.<Channel>.<someProperty> = <someValue>

<em># properties for sinks</em>
<Agent>.sources.<Sink>.<someProperty> = <someValue>
```

每个组件都应该有一个 *type* 属性，这样Flume才能知道它是什么类型的组件。每个组件类型都有它自己的一些属性。所有的这些都是根据需要进行配置。在前面的示例中，我们已经构建了一个avro-appserver-src-1到hdfs-sink-1的数据流， 下面的例子展示了如何继续给这几个组件配置剩余的属性。

```
<em># 列出所有的组件</em>
agent_foo.sources = avro-AppSrv-source
agent_foo.sinks = hdfs-Cluster1-sink
agent_foo.channels = mem-channel-1

<em># 将source和sink与channel相连接</em>
<em>#（省略）</em>

<em># 配置avro-AppSrv-source的属性</em>
agent_foo.sources.avro-AppSrv-source.type = avro         # avro-AppSrv-source 的类型是Avro Source
agent_foo.sources.avro-AppSrv-source.bind = localhost    # 监听的hostname或者ip是localhost
agent_foo.sources.avro-AppSrv-source.port = 10000        # 监听的端口是10000

<em># 配置mem-channel-1的属性</em>
agent_foo.channels.mem-channel-1.type = memory                # channel的类型是内存channel
agent_foo.channels.mem-channel-1.capacity = 1000              # channel的最大容量是1000
agent_foo.channels.mem-channel-1.transactionCapacity = 100    # source和sink每次事务从channel写入和读取的Event数量

<em># 配置hdfs-Cluster1-sink的属性</em>
agent_foo.sinks.hdfs-Cluster1-sink.type = hdfs                                    # sink的类型是HDFS Sink
agent_foo.sinks.hdfs-Cluster1-sink.hdfs.path = hdfs://namenode/flume/webdata      # 写入的HDFS目录路径

<em>#...</em>
```

在Agent中增加一个流

一个Flume Agent中可以包含多个独立的流。你可以在一个配置文件中列出所有的source、sink和channel等组件，这些组件可以被连接成多个流：

```
<em># 这样列出Agent的所有source、sink和channel，多个用空格分隔</em>
<Agent>.sources = <Source1> <Source2>
<Agent>.sinks = <Sink1> <Sink2>
<Agent>.channels = <Channel1> <Channel2>
```

然后你就可以给这些source、sink连接到对应的channel上来定义两个不同的流。例如，如果你想在一个Agent中配置两个流，一个流从外部avro客户端接收数据然后输出到外部的HDFS，另一个流从一个文件读取内容然后输出到Avro Sink。配置如下：

```
<em># 列出当前配置所有的source、sink和channel</em>
agent_foo.sources = avro-AppSrv-source1 exec-tail-source2             # 该agent中有2个sourse，分别是：avro-AppSrv-source1
agent_foo.sinks = hdfs-Cluster1-sink1 avro-forward-sink2              # 该agent中有2个sink，分别是：hdfs-Cluster1-sink1 和
agent_foo.channels = mem-channel-1 file-channel-2                     # 该agent中有2个channel，分别是：mem-channel-1 file-

<em># 这里是第一个流的配置</em>
agent_foo.sources.avro-AppSrv-source1.channels = mem-channel-1        # 与avro-AppSrv-source1相连接的channel是mem-channel
agent_foo.sinks.hdfs-Cluster1-sink1.channel = mem-channel-1           # 与hdfs-Cluster1-sink1相连接的channel是mem-channel

<em># 这里是第二个流的配置</em>
agent_foo.sources.exec-tail-source2.channels = file-channel-2         # 与exec-tail-source2相连接的channel是file-channel-
agent_foo.sinks.avro-forward-sink2.channel = file-channel-2           # 与avro-forward-sink2相连接的channel是file-channel
```

配置一个有多Agent的流

要配置一个多层级的流，你需要在第一层Agent的末尾使用Avro/Thrift Sink，并且指向下一层Agent的Avro/Thrift Source。这样就能将第一层Agent的Event发送到下一层的Agent了。例如，你使用avro客户端定期地发送文件（每个Event一个文件）到本地的 Event上，然后本地的Agent可以把Event发送到另一个配置了存储功能的Agent上。

> *提示*
>
> *语言描述似乎不太容易理解，大概是这样的结构[source1]->[channel]->[Avro Sink]->[Avro Source]->[channel2]->[Sink2]*

一个收集web日志的Agent配置：

```
<em># 列出这个Agent的source、sink和channel</em>
agent_foo.sources = avro-AppSrv-source
agent_foo.sinks = avro-forward-sink
agent_foo.channels = file-channel

<em># 把source、channel、sink连接起来，组成一个流</em>
agent_foo.sources.avro-AppSrv-source.channels = file-channel
agent_foo.sinks.avro-forward-sink.channel = file-channel

<em># avro-forward-sink 的属性配置</em>
agent_foo.sinks.avro-forward-sink.type = avro
agent_foo.sinks.avro-forward-sink.hostname = 10.1.1.100
agent_foo.sinks.avro-forward-sink.port = 10000

<em># 其他部分配置（略）</em>
<em>#...</em>
```

存储到HDFS的Agent配置：

```
<em># 列出这个Agent的source、sink和channel</em>
agent_foo.sources = avro-collection-source                                  # 只有一个source叫做：avro-collection-source
agent_foo.sinks = hdfs-sink                                                 # 只有一个sink叫做：hdfs-sink
agent_foo.channels = mem-channel                                            # 只有一个channel叫做：mem-channel

<em># 把source、channel、sink连接起来，组成一个流</em>
agent_foo.sources.avro-collection-source.channels = mem-channel
agent_foo.sinks.hdfs-sink.channel = mem-channel

<em># Avro Source的属性配置</em>
agent_foo.sources.avro-collection-source.type = avro
agent_foo.sources.avro-collection-source.bind = 10.1.1.100
agent_foo.sources.avro-collection-source.port = 10000

<em># 其他部分配置（略）</em>
<em>#...</em>
```

上面两个Agent就这样连接到了一起，最终Event会从外部应用服务器进入，经过第一个Agent流入第二个Agent，最终通过hdfs-sink存储到了HDFS。

> *提示*
>
> *第一个Agent的Avro Sink将Event发送到了10.1.1.100的10000端口上，而第二个Agent的Avro Source从10.1.1.100的10000端口上接收Event，就这样形成了两个Agent首尾相接的多Agent流。*

扇出流

如前面章节所述，Flume支持流的扇出形式配置，就是一个source连接多个channel。有两种扇出模式，**复制** 和 **多路复用** 。在复制模式下，source中的Event会被发送到与source连接的所有channel上。在多路复用模式下，Event仅被发送到 部分channel上。为了分散流量，需要指定好source的所有channel和Event分发的策略。这是通过增加一个复制或多路复用的选择器来实现的，如果是多路复用选择器，还要进一步指定Event分发的规则。 **如果没有配置选择器，默认就是复制选择器** 。

```
<em># 列出这个Agent的source、sink和channel，注意这里有1个source、2个channel和2个sink</em>
<Agent>.sources = <Source1>
<Agent>.sinks = <Sink1> <Sink2>
<Agent>.channels = <Channel1> <Channel2>

<em># 指定与source1连接的channel，这里配置了两个channel</em>
<Agent>.sources.<Source1>.channels = <Channel1> <Channel2>

<em># 将两个sink分别与两个channel相连接</em>
<Agent>.sinks.<Sink1>.channel = <Channel1>
<Agent>.sinks.<Sink2>.channel = <Channel2>

<em># 指定source1的channel选择器类型是复制选择器（按照上段介绍，不显式配置这个选择器的话，默认也是复制）</em>
<Agent>.sources.<Source1>.selector.type = replicating
```

多路复用选择器具有另外一组属性可以配置来分发数据流。这需要指定Event属性到channel的映射，选择器检查Event header中每一个配置中指定的属性值，如果与配置的规则相匹配，则该Event将被发送到规则设定的channel上。如果没有匹配的规则，则Event 会被发送到默认的channel上，具体看下面配置：

```
<em># 多路复用选择器的完整配置如下</em>
<Agent>.sources.<Source1>.selector.type = multiplexing                                  # 选择器类型是多路复用
<Agent>.sources.<Source1>.selector.header = <someHeader>                                # 假如这个<someHeader>值是abc，则选
<Agent>.sources.<Source1>.selector.mapping.<Value1> = <Channel1>                        # 加入这里Value1配置的是3，则Event
<Agent>.sources.<Source1>.selector.mapping.<Value2> = <Channel1> <Channel2>             # 同上，Event header中abc属性等于V
<Agent>.sources.<Source1>.selector.mapping.<Value3> = <Channel2>                        # 同上规则，Event header中abc属性等
<em>#...</em>

<Agent>.sources.<Source1>.selector.default = <Channel2>                                 # Event header读取到的abc属性值不属
```

映射的配置允许为每个值配置重复的channel

下面的例子中，一个数据流被分发到了两个路径上。这个叫agent_foo的Agent有一个Avro Source和两个channel，这两个channel分别连接到了两个sink上：

```
<em># 列出了Agent的所有source、 sink 和 channel</em>
agent_foo.sources = avro-AppSrv-source1
agent_foo.sinks = hdfs-Cluster1-sink1 avro-forward-sink2
agent_foo.channels = mem-channel-1 file-channel-2

<em># 让source与两个channel相连接</em>
agent_foo.sources.avro-AppSrv-source1.channels = mem-channel-1 file-channel-2

<em># 分别设定两个sink对应的channel</em>
agent_foo.sinks.hdfs-Cluster1-sink1.channel = mem-channel-1
agent_foo.sinks.avro-forward-sink2.channel = file-channel-2

<em># source的channel选择器配置</em>
agent_foo.sources.avro-AppSrv-source1.selector.type = multiplexing                  # 选择器类型是多路复用，非复制
agent_foo.sources.avro-AppSrv-source1.selector.header = State                       # 读取Event header中名字叫做
agent_foo.sources.avro-AppSrv-source1.selector.mapping.CA = mem-channel-1           # State=CA时，Event发送到me
agent_foo.sources.avro-AppSrv-source1.selector.mapping.AZ = file-channel-2          # State=AZ时，Event发送到fi
agent_foo.sources.avro-AppSrv-source1.selector.mapping.NY = mem-channel-1 file-channel-2  # State=NY时，Event发送到me
agent_foo.sources.avro-AppSrv-source1.selector.default = mem-channel-1              # 如果State不等于上面配置的任
```

上面配置中，选择器检查每个Event中名为“State”的Event header。 如果该值为“CA”，则将其发送到mem-channel-1，如果其为“AZ”，则将其发送到file-channel-2，或者如果其为“NY”则发送到两个channel上。 如果Event header中没有“State”或者与前面三个中任何一个都不匹配，则Event被发送到被设置为default的mem-channel-1上。

多路复用选择器还支持一个 *optional* 属性，看下面的例子：

```
<em># 以下是一个channel选择器的配置</em>
agent_foo.sources.avro-AppSrv-source1.selector.type = multiplexing
agent_foo.sources.avro-AppSrv-source1.selector.header = State
agent_foo.sources.avro-AppSrv-source1.selector.mapping.CA = mem-channel-1                  # CA被第一次映射到mem-
agent_foo.sources.avro-AppSrv-source1.selector.mapping.AZ = file-channel-2
agent_foo.sources.avro-AppSrv-source1.selector.mapping.NY = mem-channel-1 file-channel-2
agent_foo.sources.avro-AppSrv-source1.selector.optional.CA = mem-channel-1 file-channel-2  # 关键看这行，State=CA
agent_foo.sources.avro-AppSrv-source1.selector.mapping.AZ = file-channel-2
agent_foo.sources.avro-AppSrv-source1.selector.default = mem-channel-1
```

> *提示*
>
> *“必需channel”的意思就是被选择器配置里精确匹配到的channel，上面例子里面除了* optional *那一行，剩下的四行映射里面全都是“必需channel”；“可选channel”就是通过* optional *参数配置的映射。*

通常选择器会尝试将匹配到的Event写入指定的所有channel中，如果任何一个channel发生了写入失败的情况，就会导致整个事务的的失败，然后会在所有的channel上重试（不管某一个channel之前成功与否，只有所有channel 都成功了才认为事务成功了）。一旦所有channel写入成功，选择器还会继续将Event写入与之匹配的"可选channel"上，但是"可选channel"如果发生写入失败，选择器会忽略它。

如果"可选channel"与"必需channel"的channel有重叠（上面关于CA的两行配置就有相同的mem-channel-1），则认为该channel是必需的，这个mem-channel-1发生失败时会导致重试所有"必需channel"。上面例子中的mem-channel-1发生失败的话就会导致evnet在所有channel重试。

> *提示*
>
> *这里注意一下，CA这个例子中，"必需channel"失败会导致Event在选择器为它配置的所有通道上重试，是因为第一段中说过"*一旦所有channel写入成功，选择器还会继续将Event写入与之匹配的"可选channel"上*"，依据这个原则，再看CA的例子 必需的mem-channel-1失败后，重试且成功了，然后再把"可选channel"重试一遍，也就是mem-channel-1和file-channel-2*

如果一个Event的header没有找到匹配的"必需channel"，则它会被发送到默认的channel，并且会尝试发送到与这个Event对应的"可选channel"上。无必需，会发送到默认和可选；无必需无默认，还是会发送到可选，这种情况下所有失败都会被忽略。

SSL/TLS 支持

为了方便与其他系统安全地通信，Flume的部分组件从1.9版本开始支持SSL/TLS了。

> *提示*
>
> *这小节是Flume的1.9版本相对于之前版本最重要的一个新增特性，其实在之前的版本已经有部分组件是支持SSL的，从1.9开始将SSL的支持提高到了全局，所有支持SSL的组件配置参数也进行了统一规范命名，可以这么说1.9对SSL的支持更完整、规范和统一了。*

Component	SSL server or client
Avro Source	server
Avro Sink	client
Thrift Source	server

Component	SSL server or client
Thrift Sink	client
Kafka Source	client
Kafka Channel	client
Kafka Sink	client
HTTP Source	server
JMS Source	client
Syslog TCP Source	server
Multiport Syslog TCP Source	server

这些兼容SSL的组件有一些设置SSL的配置参数，比如ssl、keystore/truststore参数（位置、密码、类型）以及其他一些参数（比如禁用的SSL协议等）。

组件是否使用SSL始终是在组件自己的配置中开启或关闭，这样可以任意决定哪些组件使用SSL，哪些组件不使用（即使是相同类型的组件也是单独配置来指定是否开启SSL）

keystore / truststore 这种参数可以放在组件内，也可以使用全局的。

如果是在组件内单独设置，则在配置组件时配置对应的参数就行了。这种方法的优点是每个组件可以使用不同的密钥库（如果需要的话）。缺点是必须为配置文件中的每个组件配置这些参数。 组件内单独设置是可选的，一旦配置了其优先级高于全局参数。

如果使用全局设置，只需要定义一次keystore / truststore的参数就可以了，所有组件使用这同一套配置。

可以通过【系统属性】或【环境变量】来配置这些SSL的全局参数。

System property	Environment variable	**描述**

System property	Environment variable	**描述**
javax.net.ssl.keyStore	FLUME_SSL_KEYSTORE_PATH	Keystore 路径
javax.net.ssl.keyStorePassword	FLUME_SSL_KEYSTORE_PASSWORD	Keystore 密码
javax.net.ssl.keyStoreType	FLUME_SSL_KEYSTORE_TYPE	Keystore 类型 (默认是JKS)
javax.net.ssl.trustStore	FLUME_SSL_TRUSTSTORE_PATH	Truststore 路径
javax.net.ssl.trustStorePassword	FLUME_SSL_TRUSTSTORE_PASSWORD	Truststore 密码
javax.net.ssl.trustStoreType	FLUME_SSL_TRUSTSTORE_TYPE	Truststore 类型 (默认是 JKS)
flume.ssl.include.protocols	FLUME_SSL_INCLUDE_PROTOCOLS	将要使用的SSL/TLS协议版本，多个用逗号分隔，如果一个协议在下面的exclude.protocols中也配置了的话，那么这个协议会被排除，也就是exclude.protocols的优先级更高一些
flume.ssl.exclude.protocols	FLUME_SSL_EXCLUDE_PROTOCOLS	不使用的SSL/TLS协议版本，多个用逗号分隔
flume.ssl.include.cipherSuites	FLUME_SSL_INCLUDE_CIPHERSUITES	将要使用的密码套件，多个用逗号分隔，如果一个套件在下面的exclude.cipherSuites中也配置了的话，那么这个套件会被排除，也就是exclude.cipherSuites的优先级更高一些
flume.ssl.exclude.cipherSuites	FLUME_SSL_EXCLUDE_CIPHERSUITES	不使用的密码套件，多个用逗号分隔

这些SSL的系统属性可以放在启动命令中也可以放在 *conf/flume-env.sh* 文件里面的JVM参数的 `JAVA_OPTS` 中，但是不建议配在命令行中，如果这样的话敏感信息就会被存到操作系统的历史命令中，增加不必要的风险。

如果放在环境变量里面可以像下面这样配置：

```
export JAVA_OPTS="$JAVA_OPTS -Djavax.net.ssl.keyStore=/path/to/keystore.jks"
export JAVA_OPTS="$JAVA_OPTS -Djavax.net.ssl.keyStorePassword=password"
```

Flume使用JSSE（Java安全套接字扩展）中定义的系统属性，因此这是设置SSL的标准方法。 另一方面，在系统属性中指定密码意味着可以在进程列表中能看到密码， 如果在进程列表暴露不能接受，也可以改配在环境变量中，这时候Flume会从内部的相应环境变量中初始化

JSSE系统属性。

环境变量可以放在启动命令中也可以配在 *conf/flume-env.sh* 里面，但是你要知道也不建议配在命令行中，因为敏感信息又会被存到系统的历史命令中。

```
export FLUME_SSL_KEYSTORE_PATH=/path/to/keystore.jks
export FLUME_SSL_KEYSTORE_PASSWORD=password
```

请注意:

- 即使配置了全局的SSL，想要让具体某个组件使用SSL必须要在具体的组件内配置来开启才行，只配置全局的SSL不起任何作用。
- 如果SSL相关的参数被多次重复配置，那么遵从下面的优先级（从高到低）：
  - 组件自己的参数中的配置
  - 系统属性
  - 环境变量
- 如果某个组件的SSL开启，但是其他SSL参数没有配置（组件自己没配、全局的系统属性和环境变量里面也都没配）那么：
  - keystores的情况: 报错configuration error
  - truststores的情况: 使用默认的truststore (jssecacerts / cacerts in Oracle JDK)
- 在所有情况下，trustore密码都是可选的。如果未指定，那么当JDK打开信任库时，将不会在信任库上执行完整性检查。

Source、Sink组件的 batchSizes与channel的单个事务容量兼容要求

基本上Source和Sink都可以配置batchSize来指定一次事务写入/读取的最大event数量，对于有event容量上限的Channel来说，这个batchSize必须要小于这个上限。

Flume只要读取配置，就会检查这个数量设置是否合理以防止设置不兼容。

提示

看发布日志具体指的是文件Channel（File Channel）和内存Channel（Memory Channel），Source和Sink的配置的batchSize数量不应该超过channel中配置的transactionCapacity。

Flume Sources

Avro Source

Avro Source监听Avro端口，接收从外部Avro客户端发送来的数据流。如果与上一层Agent的 Avro Sink 配合使用就组成了一个分层的拓扑结构。 必需的参数已用 **粗体** 标明。

属性	**默认值**	**解释**
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： avro
bind	–	监听的服务器名hostname或者ip
port	–	监听的端口
threads	–	生成的最大工作线程数量
selector.type		可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器的相关属性
compression-type	none	可选值： none 或 deflate 。这个类型必须跟Avro Source相匹配

属性	默认值	解释
ssl	false	设置为 true 启用SSL加密，如果为true必须同时配置下面的 *keystore* 和 *keystore-password* 或者配置了全局的SSL参数也可以，想了解更多请参考 SSL/TLS 支持 。
keystore	–	SSL加密使用的Java keystore文件路径，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
keystore-password	–	Java keystore的密码，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
keystore-type	JKS	Java keystore的类型. 可选值有 JKS 、 PKCS12 ，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
exclude-protocols	SSLv3	指定不支持的协议，多个用空格分开，SSLv3不管是否配置都会被强制排除
include-protocols	–	可使用的SSL/TLS协议的以空格分隔的列表。 最终程序启用的协议将是本参数配置的协议并且排除掉上面的排除协议。 如果本参数为空，则包含所有受支持的协议。
exclude-cipher-suites	–	不使用的密码套件，多个用空格分隔
include-cipher-suites	–	使用的密码套件，多个用空格分隔。最终程序使用的密码套件就是配置的使用套件并且排除掉上面的排除套件，如果本参数为空，则包含所有受支持的密码套件。
ipFilter	false	设置为true可启用ip过滤
ipFilterRules	–	netty ipFilter的配置（参考下面的ipFilterRules详细介绍和例子）

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = avro
a1.sources.r1.channels = c1
a1.sources.r1.bind = 0.0.0.0
a1.sources.r1.port = 4141
```

ipFilterRules**格式详解**

ipFilterRules 可以配置一些允许或者禁止的ip规则，它的配置格式是：allow/deny:ip/name:pattern

第一部分只能是[allow]或[deny]两个词其中一个，第二部分是[ip]或[name]的其中一个，第三部分是正则，每个部分中间用“:”分隔。

比如可以配置成下面这样：

```
ipFilterRules=allow:ip:127.*,allow:name:localhost,deny:ip:*
```

注意，最先匹配到的规则会优先生效，看下面关于localhost的两个配置的不同

```
<em>#只允许localhost的客户端连接，禁止其他所有的连接</em>
ipFilterRules=allow:name:localhost,deny:ip:

<em>#允许除了localhost以外的任意的客户端连接</em>
ipFilterRules=deny:name:localhost,allow:ip:
```

Thrift Source

监听Thrift 端口，从外部的Thrift客户端接收数据流。如果从上一层的Flume Agent的 Thrift Sink 串联后就创建了一个多层级的Flume架构（同 Avro Source 一样，只不过是协议不同而已）。Thrift Source可以通过配置让它以安全模式（kerberos authentication）运行，具体的配置看下表。 必需的参数已用 **粗体** 标明。

> *提示*
>
> *同Avro Source十分类似，不同的是支持了 kerberos 认证。*

属性	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： thrift
bind	–	监听的 hostname 或 IP 地址
port	–	监听的端口
threads	–	生成的最大工作线程数量
selector.type		可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器的相关属性
ssl	false	设置为 true 启用SSL加密，如果为true必须同时配置下面的 *keystore* 和 *keystore-password* 或者配置了全局的SSL参数也可以，想了解更多请参考 SSL/TLS 支持 。
keystore	–	SSL加密使用的Java keystore文件路径，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
keystore-password	–	Java keystore的密码，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
keystore-type	JKS	Java keystore的类型. 可选值有 JKS 、 PKCS12 ，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
exclude-protocols	SSLv3	排除支持的协议，多个用空格分开，SSLv3不管是否配置都会被强制排除

属性	默认值	解释
include-protocols	–	可使用的SSL/TLS协议的以空格分隔的列表。 最终程序启用的协议将是本参数配置的协议并且排除掉上面的排除协议。 如果本参数为空，则包含所有受支持的协议。
exclude-cipher-suites	–	不使用的密码套件，多个用空格分隔
include-cipher-suites	–	使用的密码套件，多个用空格分隔。最终程序使用的密码套件就是配置的使用套件并且排除掉上面的排除套件，如果本参数为空，则包含所有受支持的密码套件。
kerberos	false	设置为 true ，开启kerberos 身份验证。在kerberos模式下，成功进行身份验证需要 *agent-principal* 和 *agent-keytab* 。 安全模式下的Thrift仅接受来自已启用kerberos且已成功通过kerberos KDC验证的Thrift客户端的连接。
agent-principal	–	指定Thrift Source使用的kerberos主体用于从kerberos KDC进行身份验证。
agent-keytab	—-	Thrift Source与Agent主体结合使用的keytab文件位置，用于对kerberos KDC进行身份验证。

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = thrift
a1.sources.r1.channels = c1
a1.sources.r1.bind = 0.0.0.0
a1.sources.r1.port = 4141
```

Exec Source

这个source在启动时运行给定的Unix命令，并期望该进程在标准输出上连续生成数据（stderr 信息会被丢弃，除非属性 *logStdErr* 设置为 true ）。 如果进程因任何原因退出， 则source也会退出并且不会继续生成数据。 综上来看cat [named pipe]或tail -F [file]这两个命令符合要求可以产生所需的结果，而date这种命令可能不会，因为前两个命令（tail 和 cat）能产生持续的数据流，而后者（date这种命令）只会产生单个Event并退出。

> *提示*
>
> *cat [named pipe]和tail -F [file]都能持续地输出内容，那些不能持续输出内容的命令不可以。这里注意一下cat命令后面接的参数是命名管道（named pipe）不是文件。*

必需的参数已用 **粗体** 标明。

属性	**默认值**	**解释**
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： exec
command	–	所使用的系统命令，一般是cat 或者tail
shell	–	设置用于运行命令的shell。 例如 / bin / sh -c。 仅适用于依赖shell功能的命令，如通配符、后退标记、管道等。
restartThrottle	10000	尝试重新启动之前等待的时间（毫秒）
restart	false	如果执行命令线程挂掉，是否重启
logStdErr	false	是否会记录命令的stderr内容
batchSize	20	读取并向channel发送数据时单次发送的最大数量
batchTimeout	3000	向下游推送数据时，单次批量发送Event的最大等待时间（毫秒），如果等待了batchTimeout毫秒后未达到一次批量发送数量，则仍然执行发送操作。
selector.type	replicating	可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用

属性	默认值	解释
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配置

警告

ExecSource相比于其他异步source的问题在于，如果无法将Event放入Channel中，ExecSource无法保证客户端知道它。在这种情况下数据会丢失。例如，最常见的用法是用tail -F [file]这种，应用程序负责向磁盘写入日志文件， Flume 会用tail命令从日志文件尾部读取，将每行作为一个Event发送。这里有一个明显的问题：如果channel满了然后无法继续发送Event，会发生什么？由于种种原因，Flume无法向输出日志文件的应用程序指示它需要保留日志或某些Event尚未发送。 总之你需要知道：当使用ExecSource等单向异步接口时，您的应用程序永远无法保证数据已经被成功接收！作为此警告的延伸，此source传递Event时没有交付保证。为了获得更强的可靠性保证，请考虑使用 Spooling Directory Source， Taildir Source 或通过SDK直接与Flume集成。

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = exec
a1.sources.r1.command = tail -F /var/log/secure
a1.sources.r1.channels = c1
```

*shell*属性是用来配置执行命令的shell（比如Bash或者Powershell）。command 会作为参数传递给 shell 执行，这使得command可以使用shell中的特性，例如通配符、后退标记、管道、循环、条件等。如果没有 shell 配置， 将直接调用 command 配置的命令。shell 通常配置的值有：“/bin/sh -c”、“/bin/ksh -c”、“cmd /c”、“powershell -Command”等。

```
a1.sources.tailsource-1.type = exec
a1.sources.tailsource-1.shell = /bin/bash -c
a1.sources.tailsource-1.command = for i in /path/*.txt; do cat $i; done
```

JMS Source

JMS Source是一个可以从JMS的队列或者topic中读取消息的组件。按理说JMS Source作为一个JMS的应用应该是能够与任意的JMS消息队列无缝衔接工作的，可事实上目前仅在ActiveMQ上做了测试。 JMS Source支持配置batch size、message selector、user/pass和Event数据的转换器（converter）。 注意所使用的JMS队列的jar包需要在Flume实例的classpath中，建议放在专门的插件目录plugins.d下面，或者启动时候用-classpath指定，或者编辑flume-env.sh文件的FLUME_CLASSPATH来设置。

必需的参数已用 **粗体** 标明。

属性	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： jms
initialContextFactory	–	初始上下文工厂类，比如： org.apache.activemq.jndi.ActiveMQInitialContextFactory
connectionFactory	–	连接工厂应显示为的JNDI名称
providerURL	–	JMS 的连接URL
destinationName	–	目的地名称
destinationType	–	目的地类型， queue 或 topic
messageSelector	–	创建消费者时使用的消息选择器
userName	–	连接JMS队列时的用户名
passwordFile	–	连接JMS队列时的密码文件，注意是文件名不是密码的明文
batchSize	100	消费JMS消息时单次发送的Event数量
converter.type	DEFAULT	用来转换JMS消息为Event的转换器类，参考下面参数。
converter.*	–	转换器相关的属性

属性	默认值	解释
converter.charset	UTF-8	转换器把JMS的文本消息转换为byte arrays时候使用的编码，默认转换器的专属参数
createDurableSubscription	false	是否创建持久化订阅。 持久化订阅只能在 *destinationType* = topic 时使用。 如果为 true ，则必须配置 *clientId* 和 *durableSubscriptionName*。
clientId	–	连接创建后立即给JMS客户端设置标识符。持久化订阅必配参数。
durableSubscriptionName	–	用于标识持久订阅的名称。持久化订阅必配参数。

JMS消息转换器

JMS source可以插件式配置转换器，尽管默认的转换器已经足够应付大多数场景了，默认的转换器可以把字节、文本、对象消息转换为Event。不管哪种类型消息中的属性都会作为headers被添加到Event中。

字节消息：JMS消息中的字节会被拷贝到Event的body中，注意转换器处理的单个消息大小不能超过2GB。

文本消息：JMS消息中的文本会被转为byte array拷贝到Event的body中。默认的编码是UTF-8，可自行配置编码。

对象消息：对象消息会被写出到封装在ObjectOutputStream中的ByteArrayOutputStream里面，得到的array被复制到Event的body。

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = jms
a1.sources.r1.channels = c1
```

```
a1.sources.r1.initialContextFactory = org.apache.activemq.jndi.ActiveMQInitialContextFactory
a1.sources.r1.connectionFactory = GenericConnectionFactory
a1.sources.r1.providerURL = tcp://mqserver:61616
a1.sources.r1.destinationName = BUSINESS_DATA
a1.sources.r1.destinationType = QUEUE
```

JMS Source的SSL配置

> *提示*
>
> *JMS的SSL配置有些特殊，所以放在了一节单独说。*

JMS客户端实现通常支持通过JSSE（Java安全套接字扩展）定义的某些Java系统属性来配置SSL。为Flume的JVM指定这些系统属性后，JMS Source（或更确切地说是JMS Source使用的JMS客户端实现）可以通过SSL连接到JMS服务器（当然，只有在JMS服务器也已设置为使用SSL的情况下）。 理论上它应该能在任何一个JMS服务上正常使用，目前已经通过ActiveMQ，IBM MQ和Oracle WebLogic的测试。

以下几段仅介绍Flume这边的SSL配置步骤。您可以在Flume Wiki上找到有关不同JMS服务的服务端设置的更详细描述，以及完整的工作示例。

SSL传输/服务端身份验证：

如果JMS服务端使用自签名证书，或者说其证书是由不受信任的CA（例如公司自己的CA）签名的，则需要设置信任库（包含正确的证书）并将其传递给Flume。可以通过全局SSL参数来完成。有关全局SSL设置的更多详细信息，请参见 SSL/TLS 支持 部分。

有些JMS服务端在使用SSL时需要SSL特定的JNDI Initial Context Factory 和（或）服务的URL来指定使用SSL（例如ActiveMQ使用 ssl:// 的URL前缀而不是 tcp:// ）。 在这种情况下，必须在agent配置文件中调整属性（ initialContextFactory 和（或） providerURL ）。

客户端证书认证（双向SSL）：

JMS Source可以通过客户端证书认证而不是通常的用户名/密码登录（当使用SSL并且JMS服务器配置为接受这种认证时）通过JMS服务器进行认证。

需要再次通过全局SSL参数配置包含用于身份验证的Flume密钥的密钥库。有关全局SSL设置的更多详细信息，请参见 SSL/TLS 支持 部分。

密钥库应仅包含一个密钥（如果存在多个密钥，则将使用第一个密钥）。密钥密码必须与密钥库密码相同。

如果进行客户端证书认证，则无需在Flume agent配置文件中为JMS Source配置 userName 、 passwordFile 属性。

请注意：

与其他组件不同，没有用于JMS Source的组件级配置参数。也没有启用SSL的标志。 SSL设置由 JNDI/Provider URL设置（其实还是根据JMS服务端设置）和是否存在truststore/keystore的配置。

Spooling Directory Source

这个Source允许你把要收集的文件放入磁盘上的某个指定目录。它会将监视这个目录中产生的新文件，并在新文件出现时从新文件中解析数据出来。数据解析逻辑是可配置的。 在新文件被完全读入Channel之后默认会重命名该文件以示完成（也可以配置成读完后立即删除、也可以配置trackerDir来跟踪已经收集过的文件）。

> *提示*
>
> *使用trackerDir跟踪收集的文件是通过1.9新增加了一个参数trackingPolicy，跟原有的参数组合后新增了一个使用的场景：标记已经被收集完成的文件，但是又不想对原文件做任何改动。*

与 Exec Source 不同，Spooling Directory Source是可靠的，即使Flume重新启动或被kill，也不会丢失数据。同时作为这种可靠性的代价，指定目录中的被收集的文件必须是不可变的、唯一命名的。Flume会自动检测避免这种情况发生，如果发现问题，则会抛出异常：

1. 如果文件在写入完成后又被再次写入新内容，Flume将向其日志文件（这是指Flume自己logs目录下的日志文件）打印错误并停止处理。
2. 如果在以后重新使用以前的文件名，Flume将向其日志文件打印错误并停止处理。

为了避免上述问题，生成新文件的时候文件名加上时间戳是个不错的办法。

尽管有这个Source的可靠性保证，但是仍然存在这样的情况，某些下游故障发生时会出现重复Event的情况。这与其他Flume组件提供的保证是一致的。

属性名	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： spooldir .
spoolDir	–	Flume Source监控的文件夹目录，该目录下的文件会被Flume收集
fileSuffix	.COMPLETED	被Flume收集完成的文件被重命名的后缀。1.txt被Flume收集完成后会重命名为1.txt.COMPLETED
deletePolicy	never	是否删除已完成收集的文件，可选值: never 或 immediate
fileHeader	false	是否添加文件的绝对路径名（绝对路径+文件名）到header中。
fileHeaderKey	file	添加绝对路径名到header里面所使用的key（配合上面的fileHeader一起使用）
basenameHeader	false	是否添加文件名（只是文件名，不包括路径）到header 中
basenameHeaderKey	basename	添加文件名到header里面所使用的key（配合上面的basenameHeader一起使用）
includePattern	^.*$	指定会被收集的文件名正则表达式，它跟下面的ignorePattern不冲突，可以一起使用。如果一个文件名同时被这两个正则匹配到，则会被忽略，换句话说ignorePattern的优先级更高

属性名	默认值	解释
ignorePattern	^$	指定要忽略的文件名称正则表达式。它可以跟 *includePattern* 一起使用，如果一个文件被 *ignorePattern* 和 *includePattern* 两个正则都匹配到，这个文件会被忽略。
trackerDir	.flumespool	用于存储与文件处理相关的元数据的目录。如果配置的是相对目录地址，它会在spoolDir中开始创建
trackingPolicy	rename	这个参数定义了如何跟踪记录文件的读取进度，可选值有： rename 、 tracker_dir ，这个参数只有在 *deletePolicy* 设置为 never 的时候才生效。 当设置为 rename ，文件处理完成后，将根据 *fileSuffix* 参数的配置将其重命名。 当设置为 tracker_dir ，文件处理完成后不会被重命名或其他任何改动，会在 *trackerDir* 配置的目录中创建一个新的空文件，而这个空文件的文件名就是原文件 + *fileSuffix* 参数配置的后缀
consumeOrder	oldest	设定收集目录内文件的顺序。默认是“先来先走”（也就是最早生成的文件最先被收集），可选值有： oldest 、 youngest 和 random 。当使用oldest和youngest这两种选项的时候，Flume会扫描整个文件夹进行对比排序，当文件夹里面有大量的文件的时候可能会运行缓慢。 当使用random时候，如果一直在产生新的文件，有一部分老文件可能会很久才会被收集
pollDelay	500	Flume监视目录内新文件产生的时间间隔，单位：毫秒
recursiveDirectorySearch	false	是否收集子目录下的日志文件
maxBackoff	4000	等待写入channel的最长退避时间，如果channel已满实例启动时会自动设定一个很低的值，当遇到ChannelException异常时会自动以指数级增加这个超时时间，直到达到设定的这个最大值为止。

属性名	默认值	解释
batchSize	100	每次批量传输到channel时的size大小
inputCharset	UTF-8	解析器读取文件时使用的编码（解析器会把所有文件当做文本读取）
decodeErrorPolicy	FAIL	当从文件读取时遇到不可解析的字符时如何处理。 FAIL ： 抛出异常，解析文件失败； REPLACE ： 替换掉这些无法解析的字符，通常是用U+FFFD； IGNORE ： 忽略无法解析的字符。
deserializer	LINE	指定一个把文件中的数据行解析成Event的解析器。默认是把每一行当做一个Event进行解析，所有解析器必须实现EventDeserializer.Builder接口
deserializer.*		解析器的相关属性，根据解析器不同而不同
bufferMaxLines	–	（已废弃）
bufferMaxLineLength	5000	（已废弃）每行的最大长度。改用 *deserializer.maxLineLength* 代替
selector.type	replicating	可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配置

配置范例：

```
a1.channels = ch-1
a1.sources = src-1

a1.sources.src-1.type = spooldir
a1.sources.src-1.channels = ch-1
```

```
a1.sources.src-1.spoolDir = /var/log/apache/flumeSpool
a1.sources.src-1.fileHeader = true
```

Event反序列化器

下面是Flume内置的一些反序列化工具

LINE

这个反序列化器会把文本数据的每行解析成一个Event

属性	默认值	解释
deserializer.maxLineLength	2048	每个Event数据所包含的最大字符数，如果一行文本字符数超过这个配置就会被截断，剩下的字符会出现再后面的Event数据里
deserializer.outputCharset	UTF-8	解析Event所使用的编码

> *提示*
>
> deserializer.maxLineLength *的默认值是2048，这个数值对于日志行来说有点小，如果实际使用中日志每行字符数可能超过2048，超出的部分会被截断，千万记得根据自己的日志长度调大这个值。*

AVRO

这个反序列化器能够读取avro容器文件，并在文件中为每个Avro记录生成一个Event。每个Event都会在header中记录它的模式。Event的body是二进制的avro记录内容，不包括模式和容器文件元素的其余部分。

注意如果Spooling Directory Source发生了重新把一个Event放入channel的情况（比如，通道已满导致重试），则它将重置并从最新的Avro容器文件同步点重试。 为了减少此类情况下的潜在Event重复，请在Avro输入文件中更频繁地写入同步标记。

属性名	默认值	解释
deserializer.schemaType	HASH	如何表示模式。 默认或者指定为 HASH 时，会对Avro模式进行哈希处理，并将哈希值存储在Event header中以“flume.avro.schema.hash”这个key。 如果指定为 LITERAL ，则会以JSON格式的模式存储在Event header中以“flume.avro.schema.literal”这个key。 与HASH模式相比，使用LITERAL模式效率相对较低。

BlobDeserializer

这个反序列化器可以反序列化一些大的二进制文件，一个文件解析成一个Event，例如pdf或者jpg文件等。**注意这个解析器不太适合解析太大的文件，因为被反序列化的操作是在内存里面进行的。**

属性	默认值	解释
deserializer	–	这个解析器没有别名缩写，需要填类的全限定名： org.apache.flume.sink.solr.morphline.BlobDeserializer$Builder
deserializer.maxBlobLength	100000000	每次请求的最大读取和缓冲的字节数，默认这个值大概是95.36MB

Taildir Source

> *注解*
>
> *Taildir Source目前只是个预览版本，还不能运行在windows系统上。*

Taildir Source监控指定的一些文件，并在检测到新的一行数据产生的时候几乎实时地读取它们，如果新的一行数据还没写完，Taildir Source会等到这行写完后再读取。

Taildir Source是可靠的，即使发生文件滚动（译者注1）也不会丢失数据。它会定期地以JSON格式在一个专门用于定位的文件上记录每个文件的最后读取位置。如果Flume由于某种原因停止或挂掉，它可以从文件的标记位置重新开始读取。

Taildir Source还可以从任意指定的位置开始读取文件。默认情况下，它将从每个文件的第一行开始读取。

文件按照修改时间的顺序来读取。修改时间最早的文件将最先被读取（简单记成：先来先走）。

Taildir Source不重命名、删除或修改它监控的文件。当前不支持读取二进制文件。只能逐行读取文本文件。

> *提示*
>
> *译者注1：文件滚动（file rotate）就是我们常见的log4j等日志框架或者系统会自动丢弃日志文件中时间久远的日志，一般按照日志文件大小或时间来自动分割或丢弃的机制。*

属性名	**默认值**	**解释**
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： TAILDIR .
filegroups	–	被监控的文件夹目录集合，这些文件夹下的文件都会被监控，多个用空格分隔
filegroups.<filegroupName>	–	被监控文件夹的绝对路径。正则表达式（注意不会匹配文件系统的目录）只是用来匹配文件名
positionFile	~/.flume/taildir_position.json	用来设定一个记录每个文件的绝对路径和最近一次读取位置inode的文件，这个文件是JSON格式。

属性名	默认值	解释
headers.<filegroupName>.<headerKey>	–	给某个文件组下的Event添加一个固定的键值对到header中，值就是value。一个文件组可以配置多个键值对。
byteOffsetHeader	false	是否把读取数据行的字节偏移量记录到Event的header里面，这个header的key是byteoffset
skipToEnd	false	如果在 *positionFile* 里面没有记录某个文件的读取位置，是否直接跳到文件末尾开始读取
idleTimeout	120000	关闭非活动文件的超时时间（毫秒）。如果被关闭的文件重新写入了新的数据行，会被重新打开
writePosInterval	3000	向 *positionFile* 记录文件的读取位置的间隔时间（毫秒）
batchSize	100	一次读取数据行和写入channel的最大数量，通常使用默认值就很好
maxBatchCount	Long.MAX_VALUE	控制从同一文件连续读取的行数。 如果数据来源是通过tail多个文件的方式，并且其中一个文件的写入速度很快，则它可能会阻止其他文件被处理，因为这个繁忙文件将被无休止地读取。 在这种情况下，可以调低此参数来避免被一直读取一个文件
backoffSleepIncrement	1000	在最后一次尝试未发现任何新数据时，重新尝试轮询新数据之前的时间延迟增量（毫秒）
maxBackoffSleep	5000	每次重新尝试轮询新数据时的最大时间延迟（毫秒）
cachePatternMatching	true	对于包含数千个文件的目录，列出目录并应用文件名正则表达式模式可能非常耗时。 缓存匹配文件列表可以提高性能。 消耗文件的顺序也将被缓存。 要求文件系统支持以至少秒级跟踪修改时间。
fileHeader	false	是否在header里面存储文件的绝对路径
fileHeaderKey	file	文件的绝对路径存储到header里面使用的key

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = TAILDIR
a1.sources.r1.channels = c1
a1.sources.r1.positionFile = /var/log/flume/taildir_position.json
a1.sources.r1.filegroups = f1 f2
a1.sources.r1.filegroups.f1 = /var/log/test1/example.log
a1.sources.r1.headers.f1.headerKey1 = value1
a1.sources.r1.filegroups.f2 = /var/log/test2/.*log.*
a1.sources.r1.headers.f2.headerKey1 = value2
a1.sources.r1.headers.f2.headerKey2 = value2-2
a1.sources.r1.fileHeader = true
a1.sources.ri.maxBatchCount = 1000
```

Twitter 1% firehose Source (实验性的)

> *警告*
>
> *这个source 纯粹是实验性的，之后的版本可能会有改动，使用中任何风险请自行承担。*
>
> *提示*
>
> *从Google上搜了一下twitter firehose，找到了这个 What is Twitter firehose and who can use it?，类似于Twitter提供的实时的消息流服务的API，只有少数的一些合作商公司才能使用，对于我们普通的使用者来说没有任何意义。本节可以跳过不用看了。*

这个Source通过流API连接到1%的样本twitter信息流并下载这些tweet，将它们转换为Avro格式，并将Avro Event发送到下游Flume。使用者需要有Twitter开发者账号、访问令牌和秘钥。 必需的参数已用 **粗体** 标明。

属性	**默认值**	**解释**
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： org.apache.flume.source.twitter.TwitterSource
consumerKey	–	OAuth consumer key

属性	默认值	解释
consumerSecret	–	OAuth consumer secret
accessToken	–	OAuth access token
accessTokenSecret	–	OAuth token secret
maxBatchSize	1000	每次获取twitter数据的数据集大小，简单说就是一次取多少
maxBatchDurationMillis	1000	每次批量获取数据的最大等待时间（毫秒）

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = org.apache.flume.source.twitter.TwitterSource
a1.sources.r1.channels = c1
a1.sources.r1.consumerKey = YOUR_TWITTER_CONSUMER_KEY
a1.sources.r1.consumerSecret = YOUR_TWITTER_CONSUMER_SECRET
a1.sources.r1.accessToken = YOUR_TWITTER_ACCESS_TOKEN
a1.sources.r1.accessTokenSecret = YOUR_TWITTER_ACCESS_TOKEN_SECRET
a1.sources.r1.maxBatchSize = 10
a1.sources.r1.maxBatchDurationMillis = 200
```

Kafka Source

Kafka Source就是一个Apache Kafka消费者，它从Kafka的topic中读取消息。 如果运行了多个Kafka Source，则可以把它们配置到同一个消费者组，以便每个source都读取一组唯一的topic分区。

目前支持Kafka 0.10.1.0以上版本，最高已经在Kafka 2.0.1版本上完成了测试，这已经是Flume 1.9发行时候的最高的Kafka版本了。

属性名	默认值	解释

属性名	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： org.apache.flume.source.kafka.KafkaSource
kafka.bootstrap.servers	–	Source使用的Kafka集群实例列表
kafka.consumer.group.id	flume	消费组的唯一标识符。如果有多个source或者Agent设定了相同的ID，表示它们是同一个消费者组
kafka.topics	–	将要读取消息的目标 Kafka topic 列表，多个用逗号分隔
kafka.topics.regex	–	会被Kafka Source订阅的 topic 集合的正则表达式。这个参数比 kafka.topics 拥有更高的优先级，如果这两个参数同时存在，则会覆盖kafka.topics的配置。
batchSize	1000	一批写入 channel 的最大消息数
batchDurationMillis	1000	一个批次写入 channel 之前的最大等待时间（毫秒）。达到等待时间或者数量达到 batchSize 都会触发写操作。
backoffSleepIncrement	1000	当Kafka topic 显示为空时触发的初始和增量等待时间（毫秒）。等待时间可以避免对Kafka topic的频繁ping操作。默认的1秒钟对于获取数据比较合适， 但是对于使用拦截器时想达到更低的延迟可能就需要配置更低一些。
maxBackoffSleep	5000	Kafka topic 显示为空时触发的最长等待时间（毫秒）。默认的5秒钟对于获取数据比较合适，但是对于使用拦截器时想达到更低的延迟可能就需要配置更低一些。

属性名	默认值	解释
useFlumeEventFormat	false	默认情况下，从 Kafka topic 里面读取到的内容直接以字节数组的形式赋值给Event。如果设置为true，会以Flume Avro二进制格式进行读取。与Kafka Sink上的同名参数或者 Kafka channel 的parseAsFlumeEvent参数相关联，这样以对象的形式处理能使生成端发送过来的Event header信息得以保留。
setTopicHeader	true	当设置为 true 时，会把存储Event的topic名字存储到header中，使用的key就是下面的 *topicHeader* 的值。
topicHeader	topic	如果 *setTopicHeader* 设置为 true ，则定义用于存储接收消息的 topic 使用header key。注意如果与 Kafka Sink 的 topicHeader 参数一起使用的时候要小心，避免又循环将消息又发送回 topic。
kafka.consumer.security.protocol	PLAINTEXT	设置使用哪种安全协议写入Kafka。可选值：SASL_PLAINTEXT 、 SASL_SSL 和 SSL ,有关安全设置的其他信息，请参见下文。
more consumer security props		如果使用了SASL_PLAINTEXT、SASL_SSL 或 SSL 等安全协议，参考 Kafka security 来为消费者增加安全相关的参数配置
Other Kafka Consumer Properties	–	其他一些 Kafka 消费者配置参数。任何 Kafka 支持的消费者参数都可以使用。唯一的要求是使用“kafka.consumer.”这个前缀来配置参数，比如： kafka.consumer.auto.offset.reset

> *注解*
>
> *Kafka Source 覆盖了两个Kafka 消费者的参数：auto.commit.enable 这个参数被设置成了false，Kafka Source 会提交每一个批处理。Kafka Source 保证至少一次消息恢复策略。 Source 启动时可以存在重复项。Kafka Source 还提供了key.deserializer（org.apache.kafka.common.serialization.StringSerializer） 和 value.deserializer（org.apache.kafka.common.serialization.ByteArraySerializer）的默认值，不建议修改这些参数。*

已经弃用的一些属性：

属性名	默认值	解释
topic	–	改用 kafka.topics
groupId	flume	改用 kafka.consumer.group.id
zookeeperConnect	–	自0.9.x起不再受kafka消费者客户端的支持。以后使用kafka.bootstrap.servers与kafka集群建立连接
migrateZookeeperOffsets	true	如果找不到Kafka存储的偏移量，去Zookeeper中查找偏移量并将它们提交给 Kafka 。 它应该设置为true以支持从旧版本的FlumeKafka客户端无缝迁移。 迁移后，可以将其设置为false，但通常不需要这样做。 如果在Zookeeper未找到偏移量，则可通过kafka.consumer.auto.offset.reset配置如何处理偏移量。可以从 Kafka documentation 查看更多详细信息。

通过逗号分隔的 topic 列表进行 topic 订阅的示例：

```
tier1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
tier1.sources.source1.channels = channel1
tier1.sources.source1.batchSize = 5000
tier1.sources.source1.batchDurationMillis = 2000
tier1.sources.source1.kafka.bootstrap.servers = localhost:9092
tier1.sources.source1.kafka.topics = test1, test2
tier1.sources.source1.kafka.consumer.group.id = custom.g.id
```

正则表达式 topic 订阅的示例：

```
tier1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
tier1.sources.source1.channels = channel1
tier1.sources.source1.kafka.bootstrap.servers = localhost:9092
tier1.sources.source1.kafka.topics.regex = ^topic[0-9]$
<em># the default kafka.consumer.group.id=flume is used</em>
```

安全与加密： Flume 和 Kafka 之间通信渠道是支持安全认证和数据加密的。对于身份安全验证，可以使用 Kafka 0.9.0版本中的 SASL、GSSAPI （Kerberos V5） 或 SSL （虽然名字是SSL，实际是TLS实现）。

截至目前，数据加密仅由SSL / TLS提供。

当你把 *kafka.consumer.security.protocol* 设置下面任何一个值的时候意味着：

- SASL_PLAINTEXT – 无数据加密的 Kerberos 或明文认证
- SASL_SSL – 有数据加密的 Kerberos 或明文认证
- SSL – 基于TLS的加密，可选的身份验证。

> *警告*
>
> *启用SSL时性能会下降，影响大小取决于 CPU 和 JVM 实现。参考 Kafka security overview 和 KAFKA-2561 。*

使用TLS：

请阅读 Configuring Kafka Clients SSL SSL 中描述的步骤来了解用于微调的其他配置设置，例如下面的几个例子：启用安全策略、密码套件、启用协议、truststore或秘钥库类型。

服务端认证和数据加密的一个配置实例：

```
a1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
a1.sources.source1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
a1.sources.source1.kafka.topics = mytopic
a1.sources.source1.kafka.consumer.group.id = flume-consumer
a1.sources.source1.kafka.consumer.security.protocol = SSL
<em># 如果在全局配置了SSL下面两个参数可省略，但是如果想使用自己独立的truststore，就可以把这两个参数加上。</em>
a1.sources.source1.kafka.consumer.ssl.truststore.location=/path/to/truststore.jks
a1.sources.source1.kafka.consumer.ssl.truststore.password=<password to access the truststore>
```

1.9版本开始增加了全局的ssl配置，因此这里的truststore的配置是可选配置，不配置会使用全局参数来代替，想了解更多可以参考 SSL/TLS 支持 。

注意，默认情况下 *ssl.endpoint.identification.algorithm* 这个参数没有被定义，因此不会执行主机名验证。如果要启用主机名验证，请加入以下配置：

```
a1.sources.source1.kafka.consumer.ssl.endpoint.identification.algorithm=HTTPS
```

开启后，客户端将根据以下两个字段之一验证服务器的完全限定域名（FQDN）：

1. Common Name (CN) https://tools.ietf.org/html/rfc6125#section-2.3
2. Subject Alternative Name (SAN) https://tools.ietf.org/html/rfc5280#section-4.2.1.6

如果还需要客户端身份验证，则还需要在 Flume 配置中添加以下内容，或者使用全局的SSL配置也可以，参考 SSL/TLS 支持 。 每个Flume实例都必须拥有其客户证书，来被Kafka 实例单独或通过其签名链来信任。 常见示例是由 Kafka 信任的单个根CA签署每个客户端证书。

```
<em># 下面两个参数1.9版本开始不是必须配置，如果配置了全局的keystore，这里就不必再重复配置</em>
a1.sources.source1.kafka.consumer.ssl.keystore.location=/path/to/client.keystore.jks
a1.sources.source1.kafka.consumer.ssl.keystore.password=<password to access the keystore>
```

如果密钥库和密钥使用不同的密码保护，则 *ssl.key.password* 属性将为消费者密钥库提供所需的额外密码：

```
a1.sources.source1.kafka.consumer.ssl.key.password=<password to access the key>
```

Kerberos**安全配置：**

要将Kafka Source 与使用Kerberos保护的Kafka群集一起使用，请为消费者设置上面提到的consumer.security.protocol 属性。 与Kafka实例一起使用的Kerberos keytab和主体在JAAS文件的“KafkaClient”部分中指定。 “客户端”部分描述了Zookeeper连接信息（如果需要）。 有关JAAS文件内容的信息，请参阅 Kafka doc 。 可以通过flume-env.sh中的JAVA_OPTS指定此JAAS文件的位置以及系统范围的 kerberos 配置：

```
JAVA_OPTS="$JAVA_OPTS -Djava.security.krb5.conf=/path/to/krb5.conf"
JAVA_OPTS="$JAVA_OPTS -Djava.security.auth.login.config=/path/to/flume_jaas.conf"
```

使用 SASL_PLAINTEXT 的示例安全配置：

```
a1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
a1.sources.source1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
a1.sources.source1.kafka.topics = mytopic
a1.sources.source1.kafka.consumer.group.id = flume-consumer
a1.sources.source1.kafka.consumer.security.protocol = SASL_PLAINTEXT
a1.sources.source1.kafka.consumer.sasl.mechanism = GSSAPI
a1.sources.source1.kafka.consumer.sasl.kerberos.service.name = kafka
```

使用 SASL_SSL 的安全配置范例：

```
a1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
a1.sources.source1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
a1.sources.source1.kafka.topics = mytopic
a1.sources.source1.kafka.consumer.group.id = flume-consumer
a1.sources.source1.kafka.consumer.security.protocol = SASL_SSL
a1.sources.source1.kafka.consumer.sasl.mechanism = GSSAPI
a1.sources.source1.kafka.consumer.sasl.kerberos.service.name = kafka
<em># 下面两个参数1.9版本开始不是必须配置，如果配置了全局的keystore，这里就不必再重复配置</em>
a1.sources.source1.kafka.consumer.ssl.truststore.location=/path/to/truststore.jks
a1.sources.source1.kafka.consumer.ssl.truststore.password=<password to access the truststore>
```

JAAS 文件配置示例。有关其内容的参考，请参阅Kafka文档 SASL configuration 中关于所需认证机制（GSSAPI/PLAIN）的客户端配置部分。由于Kafka Source 也可以连接 Zookeeper 以进行偏移迁移， 因此"Client"部分也添加到此示例中。除非您需要偏移迁移，否则不必要这样做，或者您需要此部分用于其他安全组件。 另外，请确保Flume进程的操作系统用户对 JAAS 和 keytab 文件具有读权限。

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=<strong>true</strong>
  storeKey=<strong>true</strong>
  keyTab="/path/to/keytabs/flume.keytab"
  principal="flume/flumehost1.example.com@YOURKERBEROSREALM";
};

KafkaClient {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=<strong>true</strong>
  storeKey=<strong>true</strong>
  keyTab="/path/to/keytabs/flume.keytab"
  principal="flume/flumehost1.example.com@YOURKERBEROSREALM";
};
```

NetCat TCP Source

这个source十分像nc -k -l [host] [port]这个命令，监听一个指定的端口，把从该端口收到的TCP协议的文本数据按行转换为Event，它能识别的是带换行符的文本数据，同其他Source一样，解析成功的Event数据会发送到channel中。

> *提示*
>
> *常见的系统日志都是逐行输出的，Flume的各种Source接收数据也基本上以行为单位进行解析和处理。不论是 NetCat TCP Source ，还是其他的读取文本类型的Source比如：Spooling Directory Source 、 Taildir Source 、 Exec Source 等也都是一样的。*

必需的参数已用 **粗体** 标明。

属性	**默认值**	**解释**
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： netcat
bind	–	要监听的 hostname 或者IP地址
port	–	监听的端口
max-line-length	512	每行解析成Event 消息体的最大字节数
ack-every-event	true	对收到的每一行数据用“OK”做出响应
selector.type	replicating	可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配置

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = netcat
a1.sources.r1.bind = 0.0.0.0
```

```
a1.sources.r1.port = 6666
a1.sources.r1.channels = c1
```

NetCat UDP Source

看名字也看得出，跟 NetCat TCP Source 是一对亲兄弟，区别是监听的协议不同。这个source就像是 nc -u -k -l [host] [port]命令一样， 监听一个端口然后接收来自于这个端口上UDP协议发送过来的文本内容，逐行转换为Event发送到channel。

必需的参数已用 **粗体** 标明。

属性	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： netcatudp
bind	–	要监听的 hostname 或者IP地址
port	–	监听的端口
remoteAddressHeader	–	UDP消息源地址（或IP）被解析到Event的header里面时所使用的key名称
selector.type	replicating	可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = netcatudp
a1.sources.r1.bind = 0.0.0.0
a1.sources.r1.port = 6666
a1.sources.r1.channels = c1
```

Sequence Generator Source

这个Source是一个序列式的Event生成器，从它启动就开始生成，总共会生成totalEvents个。它并不是一个日志收集器，它通常是用来测试用的。它在发送失败的时候会重新发送失败的Event到channel， 保证最终发送到channel的唯一Event数量一定是 *totalEvents* 个。 必需的参数已用 **粗体** 标明。

> *提示*
>
> *记住Flume的设计原则之一就是传输过程的『可靠性』，上面说的失败重试以及最终的数量问题，这是毫无疑问的。*

属性	**默认值**	**解释**
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： seq
selector.type		可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*	replicating	channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配
batchSize	1	每次请求向channel发送的 Event 数量
totalEvents	Long.MAX_VALUE	这个Source会发出的Event总数，这些Event是唯一的

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = seq
a1.sources.r1.channels = c1
```

Syslog Sources

这个Source是从syslog读取日志并解析为 Event，同样也分为TCP协议和UDP协议的，TCP协议的Source会按行（\n）来解析成 Event，UDP协议的Souce会把一个消息体解析为一个 Event。

> *提示*
>
> *这三个Syslog Sources里面的* clientIPHeader *和* clientHostnameHeader *两个参数都不让设置成Syslog header中的标准参数名，我之前并不熟悉Syslog协议，特地去搜了Syslog的两个主要版本的文档 RFC 3164 和 RFC 5424，并没有看到有 叫做_host_的标准header参数名，两个协议里面关于host的字段都叫做： HOSTNAME，不知道是不是官方文档编写者没有描述准确，总之大家如果真的使用这个组件，设置一个带个性前缀的值肯定不会有问题，比如：lyf_ip、lyf_host_name。*

Syslog TCP Source

> *提示*
>
> *这个Syslog TCP Source在源码里面已经被@deprecated了，推荐使用 Multiport Syslog TCP Source 来代替。*

必需的参数已用 **粗体** 标明。

属性	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： syslogtcp
host	–	要监听的hostname或者IP地址

属性	默认值	解释
port	–	要监听的端口
eventSize	2500	每行数据的最大字节数
keepFields	none	是否保留syslog消息头中的一些属性到Event中，可选值 all 、 none 或自定义指定保留的字段。如果设置为 all ，则会保留Priority， Timestamp 和 Hostname三个属性到Event中。 也支持单独指定保留哪些属性（支持的属性有：priority， version， timestamp， hostname），用空格分开即可。现在已经不建议使用 true 和 false，建议改用 all 和 none 了。
clientIPHeader	–	如果配置了该参数，那么客户端的IP会被自动添加到event的header中，这个参数值就是存储IP地址时的key，这样可以方便拦截器（interceptor）或channel选择器（channel selector）作为根据客户端的IP来路由分发event的依据。 注意不要设置成Syslog header的标准参数名，比如 _host_ ，这样会导致该参数被覆盖。
clientHostnameHeader	–	如果配置了该参数，那么客户端的host name会被自动添加到event的header中，这个参数值就是存储host name时的key，这样可以方便拦截器（interceptor）或channel选择器（channel selector）作为根据客户端的host name来路由分发event的依据。 检索主机名可能涉及名称服务反向查找,这可能会影响性能。注意不要设置成Syslog header的标准参数名，比如 _host_ ，这样会导致该参数被覆盖。
selector.type	replicating	可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开

属性	默认值	解释
interceptors.*		拦截器相关的属性配
ssl	false	设置为 true 启用SSL加密，如果为true必须同时配置下面的 *keystore* 和 *keystore-password* 或者配置了全局的SSL参数也可以，想了解更多请参考 SSL/TLS 支持 。
keystore	–	SSL加密使用的Java keystore文件路径，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
keystore-password	–	Java keystore的密码，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
keystore-type	JKS	Java keystore的类型. 可选值有 JKS 、 PKCS12 ，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
exclude-protocols	SSLv3	指定不支持的协议，多个用空格分开，SSLv3不管是否配置都会被强制排除
include-protocols	–	可使用的SSL/TLS协议的以空格分隔的列表。 最终程序启用的协议将是本参数配置的协议并且排除掉上面的排除协议。 如果本参数为空，则包含所有受支持的协议。
exclude-cipher-suites	–	不使用的密码套件，多个用空格分隔
include-cipher-suites	–	使用的密码套件，多个用空格分隔。最终程序使用的密码套件就是配置的使用套件并且排除掉上面的排除套件，如果本参数为空，则包含所有受支持的密码套件。

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = syslogtcp
a1.sources.r1.port = 5140
a1.sources.r1.host = localhost
a1.sources.r1.channels = c1
```

Multiport Syslog TCP Source

这是一个增强版的 Syslog TCP Source，它更新、更快、支持监听多个端口。因为支持了多个端口，port参数已经改为了ports。这个Source使用了Apache mina（一个异步通信的框架，同netty类似）来实现。 提供了对RFC-3164和许多常见的RFC-5424格式消息的支持。支持每个端口配置不同字符集。

属性	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： multiport_syslogtcp
host	–	要监听的hostname或者IP地址
ports	–	一个或多个要监听的端口，多个用空格分开
eventSize	2500	解析成Event的每行数据的最大字节数
keepFields	none	是否保留syslog消息头中的一些属性到Event中，可选值 all 、 none 或自定义指定保留的字段，如果设置为all，则会保留Priority， Timestamp 和Hostname三个属性到Event中。 也支持单独指定保留哪些属性（支持的属性有：priority， version， timestamp，hostname），用空格分开即可。现在已经不建议使用 true 和 false ，建议改用 all 和 none 了。
portHeader	–	如果配置了这个属性值，端口号会被存到每个Event的header里面用这个属性配置的值当key。这样就可以在拦截器或者channel选择器里面根据端口号来自定义路由Event的逻辑。

属性	默认值	解释
clientIPHeader	–	如果配置了该参数，那么客户端的IP会被自动添加到event的header中，这个参数值就是存储IP地址时的key，这样可以方便拦截器（interceptor）或channel选择器（channel selector）作为根据客户端的IP来路由分发event的依据。 注意不要设置成Syslog header的标准参数名，比如 _host_ ，这样会导致该参数被覆盖。
clientHostnameHeader	–	如果配置了该参数，那么客户端的host name会被自动添加到event的header中，这个参数值就是存储host name时的key，这样可以方便拦截器（interceptor）或channel选择器（channel selector）作为根据客户端的host name来路由分发event的依据。 检索主机名可能涉及名称服务反向查找,这可能会影响性能。注意不要设置成Syslog header的标准参数名，比如 _host_ ，这样会导致该参数被覆盖。
charset.default	UTF-8	解析syslog使用的默认编码
charset.port.<port>	–	针对具体某一个端口配置编码
batchSize	100	每次请求尝试处理的最大Event数量，通常用这个默认值就很好。
readBufferSize	1024	内部Mina通信的读取缓冲区大小，用于性能调优，通常用默认值就很好。
numProcessors	（自动分配）	处理消息时系统使用的处理器数量。 默认是使用Java Runtime API自动检测CPU数量。 Mina将为每个检测到的CPU核心生成2个请求处理线程，这通常是合理的。
selector.type	replicating	可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*	–	channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同

属性	默认值	解释
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配
ssl	false	设置为 true 启用SSL加密，如果为true必须同时配置下面的 *keystore* 和 *keystore-password* 或者配置了全局的SSL参数也可以，想了解更多请参考 SSL/TLS 支持 。
keystore	–	SSL加密使用的Java keystore文件路径，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
keystore-password	–	Java keystore的密码，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
keystore-type	JKS	Java keystore的类型. 可选值有 JKS 、 PKCS12 ，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
exclude-protocols	SSLv3	指定不支持的协议，多个用空格分开，SSLv3不管是否配置都会被强制排除
include-protocols	–	可使用的SSL/TLS协议的以空格分隔的列表。 最终程序启用的协议将是本参数配置的协议并且排除掉上面的排除协议。 如果本参数为空，则包含所有受支持的协议。
exclude-cipher-suites	–	不使用的密码套件，多个用空格分隔
include-cipher-suites	–	使用的密码套件，多个用空格分隔。最终程序使用的密码套件就是配置的使用套件并且排除掉上面的排除套件，如果本参数为空，则包含所有受支持的密码套件。

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = multiport_syslogtcp
a1.sources.r1.channels = c1
a1.sources.r1.host = 0.0.0.0
a1.sources.r1.ports = 10001 10002 10003
a1.sources.r1.portHeader = port
```

Syslog UDP Source

属性	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个是： syslogudp
host	–	要监听的hostname或者IP地址
port	–	要监听的端口
keepFields	false	设置为 true 后，解析syslog时会保留Priority，Timestamp and Hostname这些属性到Event的消息体中（查看源码发现，实际上保留了priority、version、timestamp、hostname这四个字段在消息体的前面）
clientIPHeader	–	如果配置了该参数，那么客户端的IP会被自动添加到event的header中，这个参数值就是存储IP地址时的key，这样可以方便拦截器（interceptor）或channel选择器（channel selector）作为根据客户端的IP来路由分发event的依据。 注意不要设置成Syslog header的标准参数名，比如 _host_ ，这样会导致该参数被覆盖。

属性	默认值	解释
clientHostnameHeader	–	如果配置了该参数，那么客户端的host name会被自动添加到event的header中，这个参数值就是存储host name时的key，这样可以方便拦截器（interceptor）或channel选择器（channel selector）作为根据客户端的host name来路由分发event的依据。 检索主机名可能涉及名称服务反向查找,这可能会影响性能。注意不要设置成Syslog header的标准参数名，比如 _host_，这样会导致该参数被覆盖。
selector.type	replicating	可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = syslogudp
a1.sources.r1.port = 5140
a1.sources.r1.host = localhost
a1.sources.r1.channels = c1
```

HTTP Source

这个Source从HTTP POST 和 GET请求里面解析 Event，GET方式目前还只是实验性的。把HTTP请求解析成Event是通过配置一个“handler”来实现的，这个“handler”必须实现 *HTTPSourceHandler* 接口， 这个接口其实就一个方法，收到一个HttpServletRequest后解析出一个

Event 的List。从一次请求解析出来的若干个Event会以一个事务提交到channel， 从而在诸如『文件channel』的一些channel上提高效率。如果handler抛出异常，这个HTTP的响应状态码是400。如果channel满了或者无法发送Event到channel，此时会返回HTTP状态码503（服务暂时不可用）。

在一个POST请求中发送的所有 Event 视为一个批处理，并在一个事务中插入到 channel。

这个Source是基于Jetty 9.4实现的，并且提供了配置Jetty参数的能力。

属性	**默认值**	**解释**
channels	–	与Source绑定的channel，多个用空格分开
type		组件类型，这个是： http
port	–	要监听的端口
bind	0.0.0.0	要监听的hostname或者IP地址
handler	org.apache.flume.source.http.JSONHandler	所使用的handler，需填写handler的全限定类名
handler.*	–	handler的一些属性配置
selector.type	replicating	可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配
ssl	false	设置为 true 启用SSL加密，HTTP Source强制不支持SSLv3协议
exclude-protocols	SSLv3	指定不支持的协议，多个用空格分开，SSLv3不管是否配置都会被强制排除

属性	默认值	解释
include-protocols	–	可使用的SSL/TLS协议的以空格分隔的列表。 最终程序启用的协议将是本参数配置的协议并且排除掉上面的排除协议。 如果本参数为空，则包含所有受支持的协议。
exclude-cipher-suites	–	不使用的密码套件，多个用空格分隔
include-cipher-suites	–	使用的密码套件，多个用空格分隔。最终程序使用的密码套件就是配置的使用套件并且排除掉上面的排除套件。
keystore		SSL加密使用的Java keystore文件路径，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
keystore-password		Java keystore的密码，如果此参数未配置就会默认使用全局的SSL的配置，如果全局的也未配置就会报错
keystore-type	JKS	Java keystore的类型. 可选值有 JKS 、 PKCS12
QueuedThreadPool.*		作用在Jetty org.eclipse.jetty.util.thread.QueuedThreadPool上的一些特定设置 注意：至少要给QueuedThreadPool配置一个参数，QueuedThreadPool才会被使用。
HttpConfiguration.*		作用在Jetty org.eclipse.jetty.server.HttpConfiguration上的一些特定设置
SslContextFactory.*		作用在Jetty org.eclipse.jetty.util.ssl.SslContextFactory上的一些特定设置（只有 *ssl* 设置为true的时候才生效）
ServerConnector.*		作用在Jetty org.eclipse.jetty.server.ServerConnector上的一些特定设置

> *提示*

> *Flume里面很多组件都明确表示强制不支持SSLv3协议，是因为SSLv3协议的不安全，各大公司很早就表示不再支持了。*

弃用的一些参数

属性	默认值	解释
keystorePassword	–	改用 *keystore-password* 。弃用的参数会被新的参数覆盖
excludeProtocols	SSLv3	改用 *exclude-protocols* 。 弃用的参数会被新的参数覆盖
enableSSL	false	改用 *ssl* ，弃用的参数会被新的参数覆盖

注意 Jetty的参数实际上是通过上面四个类里面的set方法来设置的，如果想知道这四个类里面都有哪些属性可以配置，请参考Jetty中关于这4个类的Javadoc文档 (QueuedThreadPool, HttpConfiguration, SslContextFactory and ServerConnector).

> *提示*
>
> *官方文档里面给出的四个类的Javadoc地址已经404了，现在的是我去eclipse官网找的，如果哪天这几个链接也失效了麻烦通知我一下，我再去更新。*

当给Jetty配置了一些参数的时候，组件参数优先级是高于Jetty的参数的（比如exclude-protocols 的优先级高于SslContextFactory.ExcludeProtocols），Jetty的参数均以小写字母开头。

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = http
a1.sources.r1.port = 5140
a1.sources.r1.channels = c1
a1.sources.r1.handler = org.example.rest.RestHandler
a1.sources.r1.handler.nickname = random props
a1.sources.r1.HttpConfiguration.sendServerVersion = false     # sendServerVersion是HttpConfiguration的参数，点击上面的文档链
a1.sources.r1.ServerConnector.idleTimeout = 300               # idleTimeout是ServerConnector的参数，点击上面的文档链接就能找到
```

JSONHandler

这是HTTP Source的默认解析器（handler），根据请求所使用的编码把http请求中json格式的数据解析成Flume Event数组（不管是一个还是多个，都以数组格式进行存储）， 如果未指定编码，默认使用UTF-8编码。这个handler支持UTF-8、UTF-16和UTF-32编码。json数据格式如下：

```
[{
  "headers" : {
             "timestamp" : "434324343",
             "host" : "random_host.example.com"
             },
  "body" : "random_body"
  },
  {
  "headers" : {
             "namenode" : "namenode.example.com",
             "datanode" : "random_datanode.example.com"
             },
  "body" : "really_random_body"
  }]
```

HTTP请求中设置编码必须是通过Content type来设置，application/json; charset=UTF-8(UTF-8 可以换成UTF-16 或者 UTF-32)。

一种创建这个handler使用的json格式对象 org.apache.flume.event.JSONEvent 的方法是使用Google Gson 库的Gson#fromJson(Object, Type) 方法创建json格式字符串，这个方法的第二个参数就是类型标记，用于指定Event列表的类型，像下面这样创建：

```
Type type = <strong>new</strong> TypeToken<List<JSONEvent>>() {}.getType();
```

BlobHandler

默认情况下HTTPSource会把json处理成Event。作为一个补充的选项BlobHandler 不仅支持返回请求中的参数也包含其中的二进制数据，比如PDF文件、jpg文件等。这种可以接收附件的处理器不适合处理非常大的文件，因为这些文件都是缓冲在内存里面的。

属性	默认值	解释
handler	–	这里填BlobHandler的全限定类名: org.apache.flume.sink.solr.morphline.BlobHandler
handler.maxBlobLength	100000000	每次请求的最大缓冲字节数

Stress Source

StressSource 是一个内部负载生成Source的实现， **对于压力测试非常有用** 。可以配置每个Event的大小（headers为空）、也可以配置总共发送Event数量以及发送成功的Event最大数量。

> *提示*
>
> *它跟 Sequence Generator Source 差不多，都是用来测试用的。*

必需的参数已用 **粗体** 标明。

属性	默认值	解释
type	–	组件类型，这个是： org.apache.flume.source.StressSource
size	500	每个Event的大小。单位：字节（byte）
maxTotalEvents	-1	总共会发送的Event数量
maxSuccessfulEvents	-1	发送成功的Event最大数量
batchSize	1	每次请求发送Event的数量
maxEventsPerSecond	0	每秒生成event的速度控制，当给此参数设置为一个大于0的值时开始生效

配置范例：

```
a1.sources = stresssource-1
a1.channels = memoryChannel-1
a1.sources.stresssource-1.type = org.apache.flume.source.StressSource
a1.sources.stresssource-1.size = 10240
a1.sources.stresssource-1.maxTotalEvents = 1000000
a1.sources.stresssource-1.channels = memoryChannel-1
```

Legacy Sources

Legacy Sources可以让Flume1.x版本的Agent接收来自于Flume0.9.4版本的Agent发来的Event，可以理解为连接两个版本Flume的一个“桥”。接收到0.9.4版本的Event后转换为1.x版本的Event然后发送到 channel。0.9.4版本的Event属性（timestamp， pri， host， nanos，etc）会被转换到1.xEvent的header中。Legacy Sources支持Avro和Thrift RPC两种方式连接。具体的用法是1.x的Agent可以使用 avroLegacy 或者 thriftLegacy source，然后0.9.4的Agent需要指定sink的host和端口为1.x的 Agent。

> *注解*
>
> *1.x和0.9.x的可靠性保证有所不同。Legacy Sources并不支持0.9.x的E2E和DFO模式。唯一支持的是BE（best effort，尽力而为），尽管1.x的可靠性保证对于从0.9.x传输过来并且已经存在channel里面的Events是有效的。*
>
> *提示*
>
> *虽然数据进入了Flume 1.x的channel之后是适用1.x的可靠性保证，但是从0.9.x到1.x的时候只是BE保证，既然只有BE的保证，也就是说 Legacy Sources 不算是可靠的传输。对于这种跨版本的部署使用行为要慎重。*

必需的参数已用 **粗体** 标明。

Avro Legacy Source

属性	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开

属性	默认值	解释
type	–	组件类型，这个是： org.apache.flume.source.avroLegacy.AvroLegacySource
host	–	要监听的hostname或者IP地址
port	–	要监听的端口
selector.type	replicating	可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = org.apache.flume.source.avroLegacy.AvroLegacySource
a1.sources.r1.host = 0.0.0.0
a1.sources.r1.bind = 6666
a1.sources.r1.channels = c1
```

Thrift Legacy Source

属性	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开

属性	默认值	解释
type	–	组件类型，这个是： org.apache.flume.source.thriftLegacy.ThriftLegacySource
host	–	要监听的hostname或者IP地址
port	–	要监听的端口
selector.type		可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*	replicating	channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = org.apache.flume.source.thriftLegacy.ThriftLegacySource
a1.sources.r1.host = 0.0.0.0
a1.sources.r1.bind = 6666
a1.sources.r1.channels = c1
```

Custom Source

你可以自己写一个Source接口的实现类。启动Flume时候必须把你自定义Source所依赖的其他类配置进Agent的classpath内。custom source在写配置文件的type时候填你的全限定类名。

> *提示*

> *如果前面章节的那些Source都无法满足你的需求，你可以写一个自定义的Source，与你见过的其他框架的自定义组件写法如出一辙，实现个接口而已，然后把你写的类打成jar包，连同依赖的jar包一同配置进Flume的classpath。 后面章节中的自定义Sink、自定义Channel等都是一样的步骤，不再赘述。*

属性	默认值	解释
channels	–	与Source绑定的channel，多个用空格分开
type	–	组件类型，这个填你自己Source的全限定类名
selector.type	replicating	可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同
interceptors	–	该source所使用的拦截器，多个用空格分开
interceptors.*		拦截器相关的属性配

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = org.example.MySource
a1.sources.r1.channels = c1
```

Scribe Source

提示

这里先说一句，Scribe是Facebook出的一个实时的日志聚合系统，我在之前没有听说过也没有使用过它， 从Scribe项目的Github文档里面了解到它在2013年就已经停止更新和支持了，貌似现在已经没有新的用户选择使用它了，所以Scribe Source这一节了解一下就行了。

Scribe 是另外一个类似于Flume的数据收集系统。为了对接现有的Scribe可以使用ScribeSource ， 它是基于Thrift 的兼容传输协议，如何部署Scribe请参考Facebook提供的文档。

必需的参数已用 **粗体** 标明。

属性	默认值	解释
type	–	组件类型，这个是： org.apache.flume.source.scribe.ScribeSource
port	1499	Scribe 的端口
maxReadBufferBytes	16384000	Thrift 默认的FrameBuffer 大小
workerThreads	5	Thrift的线程数
selector.type		可选值： replicating 或 multiplexing ，分别表示： 复制、多路复用
selector.*		channel选择器的相关属性，具体属性根据设定的 *selector.type* 值不同而不同

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.type = org.apache.flume.source.scribe.ScribeSource
a1.sources.r1.port = 1463
a1.sources.r1.workerThreads = 5
a1.sources.r1.channels = c1
```

Flume Sinks

HDFS Sink

这个Sink将Event写入Hadoop分布式文件系统（也就是HDFS）。 目前支持创建文本和序列文件。 它支持两种文件类型的压缩。 可以根据写入的时间、文件大小或Event数量定期滚动文件（关闭当前文件并创建新文件）。 它还可以根据Event自带的时间戳或系统时间等属性对数据进行分区。 存储文件的HDFS目录路径可以使用格式转义符，会由HDFS Sink进行动态地替换，以生成用于存储Event的目录或文件名。 使用此Sink需要安装hadoop， 以便Flume可以使用Hadoop的客户端与HDFS集群进行通信。 注意， **需要使用支持**sync() **调用的** Hadoop**版本** 。

以下是支持的转义符：

转义符	**解释**
%{host}	Event header中key为host的值。这个host可以是任意的key，只要header中有就能读取，比如%{aabc}将读取header中key为aabc的值
%t	毫秒值的时间戳（同 System.currentTimeMillis() 方法）
%a	星期的缩写（Mon、Tue等）
%A	星期的全拼（Monday、 Tuesday等）
%b	月份的缩写（Jan、 Feb等）
%B	月份的全拼（January、February等）
%c	日期和时间（Thu Feb 14 23:05:25 2019）
%d	月份中的天（00到31）
%e	月份中的天（1到31）
%D	日期，与%m/%d/%y相同 ，例如：02/09/19
%H	小时（00到23）
%I	小时（01到12）
%j	年中的天数（001到366）
%k	小时（0到23），注意跟 %H的区别

转义符	解释
%m	月份（01到12）
%n	月份（1到12）
%M	分钟（00到59）
%p	am或者pm
%s	unix时间戳，是秒值。比如2019/2/14 18:15:49的unix时间戳是：1550139349
%S	秒（00到59）
%y	一年中的最后两位数（00到99），比如1998年的%y就是98
%Y	年（2010这种格式）
%z	数字时区（比如：-0400）
%[localhost]	Agent实例所在主机的hostname
%[IP]	Agent实例所在主机的IP
%[FQDN]	Agent实例所在主机的规范hostname

注意，%[localhost], %[IP] 和 %[FQDN]这三个转义符实际上都是用java的API来获取的，在一些网络环境下可能会获取失败。

正在打开的文件会在名称末尾加上".tmp"的后缀。文件关闭后，会自动删除此扩展名。这样容易排除目录中的那些已完成的文件。 必需的参数已用 **粗体** 标明。

> *注解*
>
> *对于所有与时间相关的转义字符，Event header中必须存在带有"timestamp"键的属性（除非* hdfs.useLocalTimeStamp *设置为 true ）。快速自动添加此时间戳的一种方法是使用 时间戳添加拦截器 。*

属性名	默认值	解释
channel	–	与 Sink 连接的 channel
type	–	组件类型，这个是： hdfs

属性名	默认值	解释
hdfs.path	–	HDFS目录路径（例如：hdfs://namenode/flume/webdata/）
hdfs.filePrefix	FlumeData	Flume在HDFS文件夹下创建新文件的固定前缀
hdfs.fileSuffix	–	Flume在HDFS文件夹下创建新文件的后缀（比如：.avro，注意这个“.”不会自动添加，需要显式配置）
hdfs.inUsePrefix	–	Flume正在写入的临时文件前缀，默认没有
hdfs.inUseSuffix	.tmp	Flume正在写入的临时文件后缀
hdfs.emptyInUseSuffix	false	如果设置为 false 上面的 hdfs.inUseSuffix 参数在写入文件时会生效，并且写入完成后会在目标文件上移除 hdfs.inUseSuffix 配置的后缀。如果设置为 true 则上面的 hdfs.inUseSuffix 参数会被忽略，写文件时不会带任何后缀
hdfs.rollInterval	30	当前文件写入达到该值时间后触发滚动创建新文件（0表示不按照时间来分割文件），单位：秒
hdfs.rollSize	1024	当前文件写入达到该大小后触发滚动创建新文件（0表示不根据文件大小来分割文件），单位：字节
hdfs.rollCount	10	当前文件写入Event达到该数量后触发滚动创建新文件（0表示不根据 Event 数量来分割文件）
hdfs.idleTimeout	0	关闭非活动文件的超时时间（0表示禁用自动关闭文件），单位：秒
hdfs.batchSize	100	向 HDFS 写入内容时每次批量操作的 Event 数量
hdfs.codeC	–	压缩算法。可选值：gzip 、 bzip2 、 lzo 、 lzop`、 ``snappy

属性名	默认值	解释
hdfs.fileType	SequenceFile	文件格式，目前支持： SequenceFile 、 DataStream 、 CompressedStream 。 1. DataStream 不会压缩文件，不需要设置hdfs.codeC 2. CompressedStream 必须设置hdfs.codeC参数
hdfs.maxOpenFiles	5000	允许打开的最大文件数，如果超过这个数量，最先打开的文件会被关闭
hdfs.minBlockReplicas	–	指定每个HDFS块的最小副本数。 如果未指定，则使用 classpath 中 Hadoop 的默认配置。
hdfs.writeFormat	Writable	文件写入格式。可选值： Text 、 Writable 。在使用 Flume 创建数据文件之前设置为 Text ，否则 Apache Impala（孵化）或 Apache Hive 无法读取这些文件。
hdfs.threadsPoolSize	10	每个HDFS Sink实例操作HDFS IO时开启的线程数（open、write 等）
hdfs.rollTimerPoolSize	1	每个HDFS Sink实例调度定时文件滚动的线程数
hdfs.kerberosPrincipal	–	用于安全访问 HDFS 的 Kerberos 用户主体
hdfs.kerberosKeytab	–	用于安全访问 HDFS 的 Kerberos keytab 文件
hdfs.proxyUser		代理名
hdfs.round	false	是否应将时间戳向下舍入（如果为true，则影响除 %t 之外的所有基于时间的转义符）
hdfs.roundValue	1	向下舍入（小于当前时间）的这个值的最高倍（单位取决于下面的 *hdfs.roundUnit* ） 例子：假设当前时间戳是18:32:01，*hdfs.roundUnit* = minute 如果roundValue=5，则时间戳会取为：18:30 如果roundValue=7，则时间戳会取为：18:28 如果roundValue=10，则时间戳会取为：18:30

属性名	默认值	解释
hdfs.roundUnit	second	向下舍入的单位，可选值： second 、 minute 、 hour
hdfs.timeZone	Local Time	解析存储目录路径时候所使用的时区名，例如：America/Los_Angeles、Asia/Shanghai
hdfs.useLocalTimeStamp	false	使用日期时间转义符时是否使用本地时间戳（而不是使用 Event header 中自带的时间戳）
hdfs.closeTries	0	开始尝试关闭文件时最大的重命名文件的尝试次数（因为打开的文件通常都有个.tmp的后缀，写入结束关闭文件时要重命名把后缀去掉）。如果设置为1，Sink在重命名失败（可能是因为 NameNode 或 DataNode 发生错误）后不会重试，这样就导致了这个文件会一直保持为打开状态，并且带着.tmp的后缀；如果设置为0，Sink会一直尝试重命名文件直到成功为止；关闭文件操作失败时这个文件可能仍然是打开状态，这种情况数据还是完整的不会丢失，只有在Flume重启后文件才会关闭。
hdfs.retryInterval	180	连续尝试关闭文件的时间间隔（秒）。 每次关闭操作都会调用多次 RPC 往返于 Namenode ，因此将此设置得太低会导致 Namenode 上产生大量负载。 如果设置为0或更小，则如果第一次尝试失败，将不会再尝试关闭文件，并且可能导致文件保持打开状态或扩展名为“.tmp“。
serializer	TEXT	Event 转为文件使用的序列化器。其他可选值有： avro_event 或其他 EventSerializer.Builderinterface 接口的实现类的全限定类名。
serializer.*		根据上面 *serializer* 配置的类型来根据需要添加序列化器的参数

废弃的一些参数：

属性名	默认值	解释
hdfs.callTimeout	10000	允许HDFS操作文件的时间，比如：open、write、flush、close。如果HDFS操作超时次数增加，应该适当调高这个这个值。（毫秒）

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = hdfs
a1.sinks.k1.channel = c1
a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M/%S
a1.sinks.k1.hdfs.filePrefix = events-
a1.sinks.k1.hdfs.round = true
a1.sinks.k1.hdfs.roundValue = 10
a1.sinks.k1.hdfs.roundUnit = minute
```

上面的例子中时间戳会向前一个整10分钟取整。比如，一个 Event 的 header 中带的时间戳是11:54:34 AM, June 12, 2012，它会保存的 HDFS 路径就是/flume/events/2012-06-12/1150/00。

Hive Sink

此Sink将包含分隔文本或JSON数据的 Event 直接流式传输到 Hive表或分区上。 Event 使用 Hive事务进行写入， 一旦将一组 Event 提交给 Hive，它们就会立即显示给Hive查询。 即将写入的目标分区既可以预先自己创建，也可以选择让 Flume 创建它们，如果没有的话。 写入的 Event 数据中的字段将映射到 Hive表中的相应列。

属性	默认值	解释
channel	–	与 Sink 连接的 channel

属性	默认值	解释
type	–	组件类型，这个是： hive
hive.metastore	–	Hive metastore URI (eg thrift://a.b.com:9083)
hive.database	–	Hive 数据库名
hive.table	–	Hive表名
hive.partition	–	逗号分隔的要写入的分区信息。 比如hive表的分区是（continent: string, country 😠tring, time : string），那么“Asia,India,2014-02-26-01-21”就表示数据会写入到continent=Asia,country=India,time=2014-02-26-01-21这个分区。
hive.txnsPerBatchAsk	100	Hive从Flume等客户端接收数据流会使用多次事务来操作，而不是只开启一个事务。这个参数指定处理每次请求所开启的事务数量。来自同一个批次中所有事务中的数据最终都在一个文件中。 Flume会向每个事务中写入 *batchSize* 个 Event，这个参数和 *batchSize* 一起控制着每个文件的大小，请注意，Hive最终会将这些文件压缩成一个更大的文件。
heartBeatInterval	240	发送到 Hive 的连续心跳检测间隔（秒），以防止未使用的事务过期。设置为0表示禁用心跳。
autoCreatePartitions	true	Flume 会自动创建必要的 Hive分区以进行流式传输
batchSize	15000	写入一个 Hive事务中最大的 Event 数量
maxOpenConnections	500	允许打开的最大连接数。如果超过此数量，则关闭最近最少使用的连接。
callTimeout	10000	Hive、HDFS I/O操作的超时时间（毫秒），比如：开启事务、写数据、提交事务、取消事务。
serializer		序列化器负责解析 Event 中的字段并把它们映射到Hive表中的列，选择哪种序列化器取决于 Event 中的数据格式，支持的序列化器有：DELIMITED 和 JSON

属性	默认值	解释
round	false	是否启用时间戳舍入机制
roundUnit	minute	舍入值的单位，可选值：second 、 minute 、 hour
roundValue	1	舍入到小于当前时间的最高倍数（使用 *roundUnit* 配置的单位） 例子1：roundUnit=second，roundValue=10，则14:31:18这个时间戳会被舍入到14:31:10; 例子2：roundUnit=second，roundValue=30，则14:31:18这个时间戳会被舍入到14:31:00，14:31:42这个时间戳会被舍入到14:31:30;
timeZone	Local Time	应用于解析分区中转义序列的时区名称，比如：America/Los_Angeles、Asia/Shanghai、Asia/Tokyo等
useLocalTimeStamp	false	替换转义序列时是否使用本地时间戳（否则使用Event header中的timestamp ）

下面介绍Hive Sink的两个序列化器：

JSON： 处理UTF8编码的 Json 格式（严格语法）Event，不需要配置。 JSON中的对象名称直接映射到Hive表中具有相同名称的列。 内部使用 org.apache.hive.hcatalog.data.JsonSerDe ，但独立于 Hive表的 Serde 。 此序列化程序需要安装 HCatalog。

DELIMITED: 处理简单的分隔文本 Event。 内部使用 LazySimpleSerde，但独立于 Hive表的 Serde。

属性	默认值	解释
serializer.delimiter	,	（类型：字符串）传入数据中的字段分隔符。 要使用特殊字符，请用双引号括起来，例如“\t“

属性	默认值	解释
serializer.fieldnames	–	从输入字段到Hive表中的列的映射。 指定为Hive表列名称的逗号分隔列表（无空格），按顺序标识输入字段。 要跳过字段，请保留未指定的列名称。 例如，'time,,ip,message'表示输入映射到hive表中的 time，ip 和 message 列的第1，第3和第4个字段。
serializer.serdeSeparator	Ctrl-A	（类型：字符）自定义底层序列化器的分隔符。如果 *serializer.fieldnames* 中的字段与 Hive表列的顺序相同， 则 *serializer.delimiter* 与 *serializer.serdeSeparator* 相同， 并且 *serializer.fieldnames* 中的字段数小于或等于表的字段数量，可以提高效率，因为传入 Event 正文中的字段不需要重新排序以匹配 Hive表列的顺序。对于'\t'这样的特殊字符使用单引号，要确保输入字段不包含此字符。 注意：如果 *serializer.delimiter* 是单个字符，最好将本参数也设置为相同的字符。

以下是支持的转义符：

转义符	解释
%{host}	Event header中 key 为 host 的值。这个 host 可以是任意的 key，只要 header 中有就能读取，比如%{aabc}将读取 header 中 key 为 aabc 的值
%t	毫秒值的时间戳（同 System.currentTimeMillis() 方法）
%a	星期的缩写（Mon、Tue等）
%A	星期的全拼（Monday、 Tuesday等）
%b	月份的缩写（Jan、 Feb等）
%B	月份的全拼（January、February等）
%c	日期和时间（Thu Feb 14 23:05:25 2019）

转义符	解释
%d	月份中的天（00到31）
%D	日期，与%m/%d/%y相同，例如：02/09/19
%H	小时（00到23）
%I	小时（01到12）
%j	年中的天数（001到366）
%k	小时（0到23），注意跟 %H 的区别
%m	月份（01到12）
%M	分钟（00到59）
%p	am 或者 pm
%s	unix时间戳，是秒值。比如：2019/4/1 15:12:47 的unix时间戳是：1554102767
%S	秒（00到59）
%y	一年中的最后两位数（00到99），比如1998年的%y就是98
%Y	年（2010这种格式）
%z	数字时区（比如：-0400）

> *注解*
>
> *对于所有与时间相关的转义字符，Event header 中必须存在带有"timestamp"键的属性（除非* useLocalTimeStamp *设置为 true ）。快速添加此时间戳的一种方法是使用 时间戳添加拦截器 （ TimestampInterceptor）。*

假设Hive表如下：

```
create table weblogs ( id int , msg string )
    partitioned by (continent string, country string, time string)
    clustered by (id) into 5 buckets
    stored as orc;
```

配置范例：

```
a1.channels = c1
a1.channels.c1.type = memory
a1.sinks = k1
a1.sinks.k1.type = hive
a1.sinks.k1.channel = c1
a1.sinks.k1.hive.metastore = thrift://127.0.0.1:9083
a1.sinks.k1.hive.database = logsdb
a1.sinks.k1.hive.table = weblogs
a1.sinks.k1.hive.partition = asia,%{country},%y-%m-%d-%H-%M
a1.sinks.k1.useLocalTimeStamp = false
a1.sinks.k1.round = true
a1.sinks.k1.roundValue = 10
a1.sinks.k1.roundUnit = minute
a1.sinks.k1.serializer = DELIMITED
a1.sinks.k1.serializer.delimiter = "\t"
a1.sinks.k1.serializer.serdeSeparator = '\t'
a1.sinks.k1.serializer.fieldnames =id,,msg
```

以上配置会将时间戳向下舍入到最后10分钟。 例如，将时间戳标头设置为2019年4月1日下午15:21:34且“country”标头设置为“india”的Event将评估为分区（continent =’asia’，country =’india’，time =’2019-04-01-15-20’。序列化程序配置为接收包含三个字段的制表符分隔的输入并跳过第二个字段。

Logger Sink

使用INFO级别把Event内容输出到日志中，一般用来测试、调试使用。这个 Sink 是唯一一个不需要额外配置就能把 Event 的原始内容输出的Sink，参照 输出原始数据到日志 。

提示

在 输出原始数据到日志 一节中说过，通常在Flume的运行日志里面输出数据流中的原始的数据内容是非常不可取的，所以 Flume 的组件默认都不会这么做。但是总有特殊的情况想要把 Event 内容打印出来，就可以借助这个Logger Sink了。

必需的参数已用 **粗体** 标明。

属性	默认值	解释

属性	默认值	解释
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： logger
maxBytesToLog	16	Event body 输出到日志的最大字节数，超出的部分会被丢弃

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = logger
a1.sinks.k1.channel = c1
```

Avro Sink

这个Sink可以作为 Flume 分层收集特性的下半部分。发送到此Sink的 Event 将转换为Avro Event发送到指定的主机/端口上。Event 从 channel 中批量获取，数量根据配置的 *batch-size* 而定。 必需的参数已用 **粗体** 标明。

属性	默认值	解释
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： avro .
hostname	–	监听的服务器名（hostname）或者 IP
port	–	监听的端口
batch-size	100	每次批量发送的 Event 数
connect-timeout	20000	第一次连接请求（握手）的超时时间，单位：毫秒
request-timeout	20000	请求超时时间，单位：毫秒

属性	默认值	解释
reset-connection-interval	none	重置连接到下一跳之前的时间量（秒）。 这将强制Avro Sink 重新连接到下一跳。 这将允许Sink在添加了新的主机时连接到硬件负载均衡器后面的主机，而无需重新启动 Agent。
compression-type	none	压缩类型。可选值： none 、 deflate 。压缩类型必须与上一级Avro Source 配置的一致
compression-level	6	Event的压缩级别 0：不压缩，1-9:进行压缩，数字越大，压缩率越高
ssl	false	设置为 true 表示开启SSL 下面的 *truststore* 、 *truststore-password* 、 *truststore-type* 就是开启SSL后使用的参数，并且可以指定是否信任所有证书（ *trust-all-certs* ）
trust-all-certs	false	如果设置为true， 不会检查远程服务器（Avro Source）的SSL服务器证书。不要在生产环境开启这个配置，因为它使攻击者更容易执行中间人攻击并在加密的连接上进行“监听”。
truststore	–	自定义 Java truststore文件的路径。 Flume 使用此文件中的证书颁发机构信息来确定是否应该信任远程 Avro Source 的 SSL 身份验证凭据。 如果未指定，将使用全局的keystore配置，如果全局的keystore也未指定，将使用缺省 Java JSSE 证书颁发机构文件（通常为 Oracle JRE 中的“jssecacerts”或“cacerts”）。
truststore-password	–	上面配置的truststore的密码，如果未配置，将使用全局的truststore配置（如果配置了的话）
truststore-type	JKS	Java truststore的类型。可以配成 JKS 或者其他支持的 Java truststore类型，如果未配置，将使用全局的SSL配置（如果配置了的话）
exclude-protocols	SSLv3	要排除的以空格分隔的 SSL/TLS 协议列表。 SSLv3 协议不管是否配置都会被排除掉。

属性	默认值	解释
maxIoWorkers	2 * 机器上可用的处理器核心数量	I/O工作线程的最大数量。这个是在 NettyAvroRpcClient 的 NioClientSocketChannelFactory 上配置的。

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = avro
a1.sinks.k1.channel = c1
a1.sinks.k1.hostname = 10.10.10.10
a1.sinks.k1.port = 4545
```

Thrift Sink

这个Sink可以作为 Flume 分层收集特性的下半部分。发送到此Sink的 Event 将转换为 Thrift Event 发送到指定的主机/端口上。Event 从 channel 中获取批量获取，数量根据配置的 *batch-size* 而定。 可以通过启用 kerberos 身份验证将 Thrift Sink 以安全模式启动。如果想以安全模式与 Thrift Source 通信，那么 Thrift Sink 也必须以安全模式运行。 *client-principal* 和 *client-keytab* 是 Thrift Sink 用于向 kerberos KDC 进行身份验证的配置参数。 *server-principal* 表示此Sink将要以安全模式连接的 Thrift Source 的主体，必需的参数已用 **粗体** 标明。

属性	默认值	解释
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： thrift .
hostname	–	远程 Thrift 服务的主机名或 IP
port	–	远程 Thrift 的端口
batch-size	100	一起批量发送 Event 数量

属性	默认值	解释
connect-timeout	20000	第一次连接请求（握手）的超时时间，单位：毫秒
request-timeout	20000	请求超时时间，单位：毫秒
reset-connection-interval	none	重置连接到下一跳之前的时间量（秒）。 这将强制 Thrift Sink 重新连接到下一跳。 允许Sink在添加了新的主机时连接到硬件负载均衡器后面的主机，而无需重新启动 Agent。
ssl	false	设置为 true 表示Sink开启 SSL。下面的 *truststore* 、 *truststore-password* 、 *truststore-type* 就是开启 SSL 后使用的参数
truststore	–	自定义 Java truststore文件的路径。 Flume 使用此文件中的证书颁发机构信息来确定是否应该信任远程 Thrift Source的SSL身份验证凭据。 如果未指定，将使用全局的keystore配置， 如果全局的keystore也未指定，将使用缺省 Java JSSE 证书颁发机构文件（通常为 Oracle JRE 中的“jssecacerts”或“cacerts”）。
truststore-password	–	上面配置的truststore的密码，如果未配置，将使用全局的truststore配置（如果配置了的话）
truststore-type	JKS	Java truststore的类型。可以配成 JKS 或者其他支持的 Java truststore类型，如果未配置，将使用全局的SSL配置（如果配置了的话）
exclude-protocols	SSLv3	要排除的以空格分隔的 SSL/TLS 协议列表
kerberos	false	设置为 true 开启 kerberos 身份验证。在 kerberos 模式下，需要 *client-principal* 、 *client-keytab* 和 *server-principal* 才能成功进行身份验证并与启用了 kerberos 的 Thrift Source 进行通信。
client-principal	—-	Thrift Sink 用来向 kerberos KDC 进行身份验证的 kerberos 主体。
client-keytab	—-	Thrift Sink 与 *client-principal* 结合使用的 keytab 文件路径，用于对 kerberos KDC 进行身份验证。

属性	默认值	解释
server-principal	–	Thrift Sink 将要连接到的 Thrift Source 的 kerberos 主体。

提示

官方英文文档 *connection-reset-interval* 这个参数是错误的，在源码里面是 *reset-connection-interval* ，本文档已经纠正。

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = thrift
a1.sinks.k1.channel = c1
a1.sinks.k1.hostname = 10.10.10.10
a1.sinks.k1.port = 4545
```

IRC Sink

IRC sink 从连接的 channel 获取消息然后将这些消息中继到配置的 IRC 目标上。 必需的参数已用 **粗体** 标明。

属性	默认值	解释
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： irc
hostname	–	要连接的服务器名（hostname ）或 IP
port	6667	要连接的远程服务器端口
nick	–	昵称

属性	默认值	解释
user	–	用户名
password	–	密码
chan	–	频道
name		真实姓名
splitlines	false	是否分割消息后进行发送
splitchars	\n	行分隔符如果上面 *splitlines* 设置为 true ，会使用这个分隔符把消息体先进行分割再逐个发送，如果你要在配置文件中配置默认值，那么你需要一个转义符，像这样：“ \n”

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = irc
a1.sinks.k1.channel = c1
a1.sinks.k1.hostname = irc.yourdomain.com
a1.sinks.k1.nick = flume
a1.sinks.k1.chan = #flume
```

File Roll Sink

把 Event 存储到本地文件系统。 必需的参数已用 **粗体** 标明。

属性	默认值	解释
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： file_roll .

属性	默认值	解释
sink.directory	–	Event 将要保存的目录
sink.pathManager	DEFAULT	配置使用哪个路径管理器，这个管理器的作用是按照规则生成新的存储文件名称，可选值有： default 、 rolltime 。default规则：prefix+当前毫秒值+“-”+文件序号+“.”+extension；rolltime规则：prefix+yyyyMMddHHmmss+“-”+文件序号+“.”+extension；注：prefix 和 extension 如果没有配置则不会附带
sink.pathManager.extension	–	如果上面的 *pathManager* 使用默认的话，可以用这个属性配置存储文件的扩展名
sink.pathManager.prefix	–	如果上面的 *pathManager* 使用默认的话，可以用这个属性配置存储文件的文件名的固定前缀
sink.rollInterval	30	表示每隔30秒创建一个新文件进行存储。如果设置为0，表示所有 Event 都会写到一个文件中。
sink.serializer	TEXT	配置 Event 序列化器，可选值有： text 、 header_and_text 、 avro_event 或者自定义实现了 EventSerializer.Builder 接口的序列化器的全限定类名.。 text 只会把 Event 的 body 的文本内容序列化； header_and_text 会把 header 和 body 内容都序列化。
sink.batchSize	100	每次事务批处理的 Event 数

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = file_roll
a1.sinks.k1.channel = c1
a1.sinks.k1.sink.directory = /var/log/flume
```

Null Sink

丢弃所有从 channel 读取到的 Event。 必需的参数已用 **粗体** 标明。

属性	默认值	解释
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： null .
batchSize	100	每次批处理的 Event 数量

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = null
a1.sinks.k1.channel = c1
```

HBaseSinks

HBaseSink

此Sink将数据写入 HBase。 Hbase 配置是从classpath中遇到的第一个 hbase-site.xml 中获取的。 配置指定的 *HbaseEventSerializer* 接口的实现类用于将 Event 转换为 HBase put 或 increments。 然后将这些 put 和 increments 写入 HBase。 该Sink提供与 HBase 相同的一致性保证，HBase 是当前行的原子性。 如果 Hbase 无法写入某些 Event，则Sink将重试该事务中的所有 Event。

这个Sink支持以安全的方式把数据写入到 HBase。为了使用安全写入模式，运行 Flume 实例的用户必须有写入 HBase 目标表的写入权限。可以在配置中指定用于对 KDC 进行身份验证的主体和密钥表。 Flume 的 classpath 中的 hbase-site.xml 必须将身份验证设置为 kerberos（有关如何执行此操作的详细信息，请参阅HBase文档）。

Flume提供了两个序列化器。第一个序列化器是 SimpleHbaseEventSerializer（*org.apache.flume.sink.hbase.SimpleHbaseEventSerializer*），它把 Event body 原样写入到HBase，并可选增加HBase列，这个实现主要就是提供个例子。 第二个序列化器是 RegexHbaseEventSerializer（*org.apache.flume.sink.hbase.RegexHbaseEventSerializer*），它把 Event body 按照给定的正则进行分割然后写入到不同的列中。

必需的参数已用 **粗体** 标明。

属性	**默认值**	**解释**
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： hbase
table	–	要写入的 Hbase 表名
columnFamily	–	要写入的 Hbase 列族
zookeeperQuorum	–	Zookeeper 节点（host:port格式，多个用逗号分隔），hbase-site.xml 中属性 *hbase.zookeeper.quorum* 的值
znodeParent	/hbase	ZooKeeper 中 HBase 的 Root ZNode 路径，hbase-site.xml中 zookeeper.znode.parent 的值。
batchSize	100	每个事务写入的 Event 数量
coalesceIncrements	false	每次提交时，Sink是否合并多个 increment 到一个 cell。如果有限数量的 cell 有多个 increment ，这样可能会提供更好的性能。
serializer	org.apache.flume.sink.hbase.SimpleHbaseEventSerializer	指定序列化器。默认的increment column = “iCol”，payload column = “pCol”。
serializer.*	–	序列化器的属性

属性	默认值	解释
kerberosPrincipal	–	以安全方式访问 HBase 的 Kerberos 用户主体
kerberosKeytab	–	以安全方式访问 HBase 的 Kerberos keytab 文件目录

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = hbase
a1.sinks.k1.table = foo_table
a1.sinks.k1.columnFamily = bar_cf
a1.sinks.k1.serializer = org.apache.flume.sink.hbase.RegexHbaseEventSerializer
a1.sinks.k1.channel = c1
```

HBase2Sink

> *提示*
>
> *这是Flume 1.9新增的Sink。*

HBase2Sink 是HBaseSink的HBase 2版本。

所提供的功能和配置参数与HBaseSink相同

必需的参数已用 **粗体** 标明。

属性	默认值	解释
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： hbase2

属性	默认值	解释
table	–	要写入的 Hbase 表名
columnFamily	–	要写入的 Hbase 列族
zookeeperQuorum	–	Zookeeper 节点（host:port格式，多个用逗号分隔），hbase-site.xml 中属性 *hbase.zookeeper.quorum* 的值
znodeParent	/hbase	ZooKeeper 中 HBase 的 Root ZNode 路径，hbase-site.xml 中 *zookeeper.znode.parent* 的值
batchSize	100	每个事务写入的Event数量
coalesceIncrements	false	每次提交时，Sink是否合并多个 increment 到一个 cell。如果有限数量的 cell 有多个 increment ，这样可能会提供更好的性能
serializer	org.apache.flume.sink.hbase2.SimpleHBase2EventSerializer	默认的列 increment column = “iCol”, payload column = “pCol”
serializer.*	–	序列化器的一些属性
kerberosPrincipal	–	以安全方式访问 HBase 的 Kerberos 用户主体
kerberosKeytab	–	以安全方式访问 HBase 的 Kerberos keytab 文件目录

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = hbase2
a1.sinks.k1.table = foo_table
a1.sinks.k1.columnFamily = bar_cf
a1.sinks.k1.serializer = org.apache.flume.sink.hbase2.RegexHBase2EventSerializer
a1.sinks.k1.channel = c1
```

AsyncHBaseSink

这个Sink使用异步模型将数据写入HBase。这个Sink使用 *AsyncHbaseEventSerializer* 这个序列化器来转换 Event 为 HBase 的 put 和 increment，然后写入到 HBase。 此Sink使用 Asynchbase API 来写入 HBase。该Sink提供与 HBase 相同的一致性保证，HBase 是当前行的原子性。 如果 Hbase 无法写入某些 Event，则Sink将重试该事务中的所有 Event。

AsyncHBaseSink只能在HBase 1.x版本上使用，因为AsyncHBaseSink使用的async client不兼容HBase 2。

必需的参数已用 **粗体** 标明。

属性	**默认值**	**解释**
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： asynchbase
table	–	要写入的 Hbase 表名
zookeeperQuorum	–	Zookeeper 节点（host:port格式，多个用逗号分隔），hbase-site.xml 中属性 *hbase.zookeeper.quorum* 的值
znodeParent	/hbase	ZooKeeper 中 HBase 的 Root ZNode 路径，hbase-site.xml 中 *zookeeper.znode.parent* 的值
columnFamily	–	要写入的 Hbase 列族
batchSize	100	每个事务写入的Event数量
coalesceIncrements	false	每次提交时，Sink是否合并多个 increment 到一个 cell。如果有限数量的 cell 有多个 increment ，这样可能会提供更好的性能
timeout	60000	Sink为事务中所有 Event 等待来自 HBase 响应的超时时间（毫秒）
serializer	org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer	序列化器
serializer.*	–	序列化器的一些属性

属性	默认值	解释
async.*	–	AsyncHBase库的一些参数配置，这里配置的参数优先于上面的原来的 zookeeperQuorum 和 znodeParent ，你可以在这里查看它支持的参数列表 the documentation page of AsyncHBase。

如果配置文件中没有提供这些参数配置，Sink就会从 classpath 中第一个 hbase-site.xml 中读取这些需要的配置信息。

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = asynchbase
a1.sinks.k1.table = foo_table
a1.sinks.k1.columnFamily = bar_cf
a1.sinks.k1.serializer = org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer
a1.sinks.k1.channel = c1
```

MorphlineSolrSink

此Sink从 Flume的 Event 中提取数据，对其进行转换，并将其近乎实时地加载到 Apache Solr 服务器中，后者又向最终用户或搜索应用程序提供查询服务。

此Sink非常适合将原始数据流式传输到 HDFS（通过HDFS Sink）并同时提取、转换并将相同数据加载到 Solr（通过MorphlineSolrSink）的使用场景。特别是，此Sink可以处理来自不同数据源的任意异构原始数据，并将其转换为对搜索应用程序有用的数据模型。

ETL 功能可使用 morphline 的配置文件进行自定义，该文件定义了一系列转换命令，用于将 Event 从一个命令传递到另一个命令。

Morphlines 可以看作是 Unix 管道的演变，其中数据模型被推广为使用通用记录流，包括任意二进制有效载荷。 morphline 命令有点像 Flume 拦截器。 Morphlines 可以嵌入到 Flume 等 Hadoop 组件中。

用于解析和转换一组标准数据格式（如日志文件，Avro，CSV，文本，HTML，XML，PDF，Word，Excel等）的命令是开箱即用的，还有其他自定义命令和解析器用于其他数据格式可以作为插件添加到 morphline。可以索引任何类型的数据格式， 并且可以生成任何类型的 Solr 模式的任何 Solr 文档，也可以注册和执行任何自定义 ETL 逻辑。

Morphlines 操纵连续的数据流。数据模型可以描述如下：数据记录是一组命名字段，其中每个字段具有一个或多个值的有序列表。值可以是任何Java对象。也就是说，数据记录本质上是一个哈希表， 其中每个哈希表条目包含一个 String 键和一个 Java 对象列表作为值。 （该实现使用 Guava 的 ArrayListMultimap，它是一个 ListMultimap）。请注意，字段可以具有多个值，并且任何两个记录都不需要使用公共字段名称。

此Sink将 Flume Event 的 body 填充到 morphline 记录的 *_attachment_body* 字段中，并将 Flume Event 的 header 复制到同名的记录字段中。然后命令可以对此数据执行操作。

支持路由到 SolrCloud 集群以提高可伸缩性。索引负载可以分布在大量 MorphlineSolrSinks 上，以提高可伸缩性。可以跨多个 MorphlineSolrSinks 复制索引负载以实现高可用性， 例如使用 Flume的负载均衡特性。 MorphlineInterceptor 还可以帮助实现到多个 Solr 集合的动态路由（例如，用于多租户）。

老规矩，morphline 和 solr 的 jar 包需要放在 Flume 的 lib 目录中。

必需的参数已用 **粗体** 标明。

属性	默认值	解释
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： org.apache.flume.sink.solr.morphline.MorphlineSolrSink

属性	默认值	解释
morphlineFile	–	morphline 配置文件的相对或者绝对路径，例如：/etc/flume-ng/conf/morphline.conf
morphlineId	null	如果 morphline 文件里配置了多个 morphline 实例，可以用这个参数来标识 morphline 作为一个可选名字
batchSize	1000	单个事务操作的最大 Event 数量
batchDurationMillis	1000	事务的最大超时时间（毫秒）。达到这个时间或者达到 *batchSize* 都会触发提交事物。
handlerClass	org.apache.flume.sink.solr.morphline.MorphlineHandlerImpl	实现了 org.apache.flume.sink.solr.morphline.MorphlineHandler 接口的实现类的全限定类名
isProductionMode	false	重要的任务和大规模的生产系统应该启用这个模式，这些系统需要在发生不可恢复的异常时不停机来获取信息。未知的 Solr 架构字段相关的错误、损坏或格式错误的解析器输入数据、解析器错误等都会产生不可恢复的异常。
recoverableExceptionClasses	org.apache.solr.client.solrj.SolrServerException	以逗号分隔的可恢复异常列表，这些异常往往是暂时的，在这种情况下，可以进行相应地重试。 比如：网络连接错误，超时等。当 isProductionMode 标志设置为 true 时，使用此参数配置的可恢复异常将不会被忽略，并且会进行重试。
isIgnoringRecoverableExceptions	false	如果不可恢复的异常被意外错误分类为可恢复，则应启用这个标志。 这使得Sink能够取得进展并避免永远重试一个 Event。

配置范例：

```
a1.channels = c1
a1.sinks = k1
```

```
a1.sinks.k1.type = org.apache.flume.sink.solr.morphline.MorphlineSolrSink
a1.sinks.k1.channel = c1
a1.sinks.k1.morphlineFile = /etc/flume-ng/conf/morphline.conf
<em># a1.sinks.k1.morphlineId = morphline1</em>
<em># a1.sinks.k1.batchSize = 1000</em>
<em># a1.sinks.k1.batchDurationMillis = 1000</em>
```

ElasticSearchSink

这个Sink把数据写入到 elasticsearch 集群，就像 logstash 一样把 Event 写入以便 Kibana 图形接口可以查询并展示。

必须将环境所需的 elasticsearch 和 lucene-core jar 放在 Flume 安装的 lib 目录中。 Elasticsearch 要求客户端 JAR 的主要版本与服务器的主要版本匹配，并且两者都运行相同的 JVM 次要版本。如果版本不正确，会报 SerializationExceptions 异常。 要选择所需的版本，请首先确定 elasticsearch 的版本以及目标群集正在运行的 JVM 版本。然后选择与主要版本匹配的 elasticsearch 客户端库。 0.19.x客户端可以与0.19.x群集通信; 0.20.x可以与0.20.x对话，0.90.x可以与0.90.x对话。确定 elasticsearch 版本后， 读取 pom.xml 文件以确定要使用的正确 lucene-core JAR 版本。运行 ElasticSearchSink 的 Flume 实例程序也应该与目标集群运行的次要版本的 JVM 相匹配。

所有的 Event 每天会被写入到新的索引，名称是<indexName>-yyyy-MM-dd的格式，其中<indexName>可以自定义配置。Sink将在午夜 UTC 开始写入新索引。

默认情况下，Event 会被 ElasticSearchLogStashEventSerializer 序列化器进行序列化。可以通过 serializer 参数配置来更改序和自定义列化器。这个参数可以配置 *org.apache.flume.sink.elasticsearch.ElasticSearchEventSerializer* 或 *org.apache.flume.sink.elasticsearch.ElasticSearchIndexRequestBuilderFactory* 接口的实现类，ElasticSearchEventSerializer 现在已经不建议使用了，推荐使用更强大的后者。

必需的参数已用 **粗体** 标明。

属性	默认值	解释
channel	–	与 Sink 绑定的 channel

属性	默认值	解释
type	–	组件类型，这个是： org.apache.flume.sink.elasticsearch.ElasticSearchSink
hostNames	–	逗号分隔的hostname:port列表，如果端口不存在，则使用默认的9300端口
indexName	flume	指定索引名称的前缀。比如：默认是“flume”，使用的索引名称就是 flume-yyyy-MM-dd 这种格式。也支持 header 属性替换的方式，比如%{lyf}就会用 Event header 中的属性名为 lyf 的值。
indexType	logs	文档的索引类型。默认为 log，也支持 header 属性替换的方式，比如%{lyf}就会用 Event header 中的属性名为 lyf 的值。
clusterName	elasticsearch	要连接的 ElasticSearch 集群名称
batchSize	100	每个事务写入的 Event 数量
ttl	–	TTL 以天为单位，设置了会导致过期文档自动删除，如果没有设置，文档将永远不会被自动删除。 TTL 仅以较早的整数形式被接受， 例如 a1.sinks.k1.ttl = 5并且还具有限定符 ms （毫秒）， s （秒）， m （分钟）， h （小时）， d （天）和 w （星期）。 示例 a1.sinks.k1.ttl = 5d 表示将TTL设置为5天。 点击 http://www.elasticsearch.org/guide/reference/mapping/ttl-field/ 了解更多信息。
serializer	org.apache.flume.sink.elasticsearch.ElasticSearchLogStashEventSerializer	序列化器必须实现 *ElasticSearchEventSerializer* 或 *ElasticSearchIndexRequestBuilderFactory* 接口，推荐使用后者。
serializer.*	–	序列化器的一些属性配置

> *注解*
>
> *使用 header 替换可以方便地通过 header 中的值来动态地决定存储 Event 时要时候用的 indexName 和 indexType。使用此功能时应谨慎，因为 Event 提交者可以控制 indexName 和 indexType。 此外，如果使用 elasticsearch REST 客户端，则 Event 提交者可以控制所使用的URL路径。*

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = elasticsearch
a1.sinks.k1.hostNames = 127.0.0.1:9200,127.0.0.2:9300
a1.sinks.k1.indexName = foo_index
a1.sinks.k1.indexType = bar_type
a1.sinks.k1.clusterName = foobar_cluster
a1.sinks.k1.batchSize = 500
a1.sinks.k1.ttl = 5d
a1.sinks.k1.serializer = org.apache.flume.sink.elasticsearch.ElasticSearchDynamicSerializer
a1.sinks.k1.channel = c1
```

Kite Dataset Sink

这是一个将 Event 写入到 Kite 的实验性的Sink。这个Sink会反序列化每一个 Event body，并将结果存储到 Kite Dataset。它通过按URI加载数据集来确定目标数据集。

唯一支持的序列化方式是 avro，并且必须在在 Event header 中传递数据的结构，使用 *flume.avro.schema.literal* 加 json 格式的结构信息表示，或者用 *flume.avro.schema.url* 加一个能够获取到结构信息的URL（比如hdfs:/...这种）。 这与使用deserializer.schemaType = LITERAL 的 Log4jAppender 和 Spooling Directory Source 的 avro 反序列化器兼容。

> *注解*
>
> *1、flume.avro.schema.hash 这个 header 不支持； 2、在某些情况下，在超过滚动间隔后会略微发生文件滚动，但是这个延迟不会超过5秒钟，大多数情况下这个延迟是可以忽略的。*

属性	默认值	解释
channel	–	与 Sink 绑定的 channel

属性	默认值	解释
type	–	组件类型，这个是：org.apache.flume.sink.kite.DatasetSink
kite.dataset.uri	–	要打开的数据集的 URI
kite.repo.uri	–	要打开的存储库的 URI（**不建议使用**，请改用 *kite.dataset.uri*）
kite.dataset.namespace	–	将写入记录的数据集命名空间（**不建议使用**，请改用 *kite.dataset.uri*）
kite.dataset.name	–	将写入记录的数据集名称（**不建议使用**，请改用 *kite.dataset.uri*）
kite.batchSize	100	每批中要处理的记录数
kite.rollInterval	30	释放数据文件之前的最长等待时间（秒）
kite.flushable.commitOnBatch	true	如果为 true，Flume 在每次批量操作 *kite.batchSize* 数据后提交事务并刷新 writer。 此设置仅适用于可刷新数据集。 如果为 true，则可以将具有提交数据的临时文件保留在数据集目录中。 需要手动恢复这些文件，以使数据对 DatasetReaders 可见。
kite.syncable.syncOnBatch	true	Sink在提交事务时是否也将同步数据。 此设置仅适用于可同步数据集。 同步操作能保证数据将写入远程系统上的可靠存储上，同时保证数据已经离开Flume客户端的缓冲区（也就是 channel）。当 *thekite.flushable.commitOnBatch* 属性设置为 false 时，此属性也必须设置为 false 。
kite.entityParser	avro	将 Flume Event 转换为 kite 实体的转换器。取值可以是 avro 或者 *EntityParser.Builder* 接口实现类的全限定类名

属性	默认值	解释
kite.failurePolicy	retry	发生不可恢复的异常时采取的策略。例如 Event header 中缺少结构信息。默认采取重试的策略。 其他可选的值有： save ，这样会把 Event 原始内容写入到 *kite.error.dataset.uri* 这个数据集。还可以填自定义的处理策略类的全限定类名（需实现 *FailurePolicy.Builder* 接口）
kite.error.dataset.uri	–	保存失败的 Event 存储的数据集。当上面的参数 *kite.failurePolicy* 设置为 save 时，此参数必须进行配置。
auth.kerberosPrincipal	–	用于 HDFS 安全身份验证的 Kerberos 用户主体
auth.kerberosKeytab	–	Kerberos 安全验证主体的 keytab 本地文件系统路径
auth.proxyUser	–	HDFS 操作的用户，如果与 kerberos 主体不同的话

Kafka Sink

这个 Sink 可以把数据发送到 Kafka topic上。目的就是将 Flume 与 Kafka 集成，以便基于拉的处理系统可以处理来自各种 Flume Source 的数据。

目前支持Kafka 0.10.1.0以上版本，最高已经在Kafka 2.0.1版本上完成了测试，这已经是Flume 1.9发行时候的最高的Kafka版本了。

必需的参数已用 **粗体** 标明。

属性	默认值	解释
type	–	组件类型，这个是： org.apache.flume.sink.kafka.KafkaSink

属性	默认值	解释
kafka.bootstrap.servers	–	Kafka Sink 使用的 Kafka 集群的实例列表，可以是实例的部分列表。但是更建议至少两个用于高可用（HA）支持。格式为 hostname:port，多个用逗号分隔
kafka.topic	default-flume-topic	用于发布消息的 Kafka topic 名称 。如果这个参数配置了值，消息就会被发布到这个 topic 上。如果Event header中包含叫做“topic”的属性， Event 就会被发布到 header 中指定的 topic 上，而不会发布到 *kafka.topic* 指定的 topic 上。支持任意的 header 属性动态替换， 比如%{lyf}就会被 Event header 中叫做“lyf”的属性值替换（如果使用了这种动态替换，建议将 Kafka 的 *auto.create.topics.enable* 属性设置为 true ）。
flumeBatchSize	100	一批中要处理的消息数。设置较大的值可以提高吞吐量，但是会增加延迟。
kafka.producer.acks	1	在考虑成功写入之前，要有多少个副本必须确认消息。可选值， 0 ： （从不等待确认）； 1 ： 只等待leader确认； -1 ： 等待所有副本确认。 设置为-1可以避免某些情况 leader 实例失败的情况下丢失数据。
useFlumeEventFormat	false	默认情况下，会直接将 Event body 的字节数组作为消息内容直接发送到 Kafka topic 。如果设置为true，会以 Flume Avro 二进制格式进行读取。 与 Kafka Source 上的同名参数或者 Kafka channel 的 *parseAsFlumeEvent* 参数相关联，这样以对象的形式处理能使生成端发送过来的 Event header 信息得以保留。

属性	默认值	解释
defaultPartitionId	–	指定所有 Event 将要发送到的 Kafka 分区ID，除非被 *partitionIdHeader* 参数的配置覆盖。 默认情况下，如果没有设置此参数，Event 会被 Kafka 生产者的分发程序分发，包括 key（如果指定了的话），或者被 *kafka.partitioner.class* 指定的分发程序来分发
partitionIdHeader	–	设置后，Sink将使用 Event header 中使用此属性的值命名的字段的值，并将消息发送到 topic 的指定分区。 如果该值表示无效分区，则将抛出 EventDeliveryException。 如果存在标头值，则此设置将覆盖 *defaultPartitionId* 。假如这个参数设置为“lyf”，这个 Sink 就会读取 Event header 中的 lyf 属性的值，用该值作为分区ID
allowTopicOverride	true	如果设置为 true，会读取 Event header 中的名为 *topicHeader* 的的属性值，用它作为目标 topic。
topicHeader	topic	与上面的 *allowTopicOverride* 一起使用，*allowTopicOverride* 会用当前参数配置的名字从 Event header 获取该属性的值，来作为目标 topic 名称
kafka.producer.security.protocol	PLAINTEXT	设置使用哪种安全协议写入 Kafka。可选值：SASL_PLAINTEXT 、 SASL_SSL 和 SSL， 有关安全设置的其他信息，请参见下文。
more producer security props		如果使用了 SASL_PLAINTEXT 、 SASL_SSL 或 SSL 等安全协议，参考 Kafka security 来为生产者增加安全相关的参数配置
Other Kafka Producer Properties	–	其他一些 Kafka 生产者配置参数。任何 Kafka 支持的生产者参数都可以使用。唯一的要求是使用“kafka.producer.”这个前缀来配置参数，比如：*kafka.producer.linger.ms*

> *注解*
>
> *Kafka Sink使用 Event header 中的 topic 和其他关键属性将 Event 发送到 Kafka。 如果 header 中存在 topic，则会将Event发送到该特定 topic，从而覆盖为Sink配置的 topic。 如果 header 中存在指定分区相关的参数，则Kafka将使用相关参数发送到指定分区。 header中特定参数相同的 Event 将被发送到同一分区。 如果为空，则将 Event 会被发送到随机分区。 Kafka Sink 还提供了key.deserializer（org.apache.kafka.common.serialization.StringSerializer） 和value.deserializer （org.apache.kafka.common.serialization.ByteArraySerializer）的默认值，不建议修改这些参数。*

弃用的一些参数：

属性	默认值	解释
brokerList	–	改用 kafka.bootstrap.servers
topic	default-flume-topic	改用 kafka.topic
batchSize	100	改用 kafka.flumeBatchSize
requiredAcks	1	改用 kafka.producer.acks

下面给出 Kafka Sink 的配置示例。Kafka 生产者的属性都是以 kafka.producer 为前缀。Kafka 生产者的属性不限于下面示例的几个。此外，可以在此处包含您的自定义属性，并通过作为方法参数传入的Flume Context对象在预处理器中访问它们。

```
a1.sinks.k1.channel = c1
a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.k1.kafka.topic = mytopic
a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
a1.sinks.k1.kafka.flumeBatchSize = 20
a1.sinks.k1.kafka.producer.acks = 1
a1.sinks.k1.kafka.producer.linger.ms = 1
a1.sinks.k1.kafka.producer.compression.type = snappy
```

安全与加密

Flume 和 Kafka 之间通信渠道是支持安全认证和数据加密的。对于身份安全验证，可以使用 Kafka 0.9.0版本中的 SASL、GSSAPI （Kerberos V5） 或 SSL （虽然名字是SSL，实际是TLS实现）。

截至目前，数据加密仅由SSL / TLS提供。

Setting kafka.producer.security.protocol to any of the following value means:

当你把 kafka.producer.security.protocol 设置下面任何一个值的时候意味着：

- SASL_PLAINTEXT – 无数据加密的 Kerberos 或明文认证
- SASL_SSL – 有数据加密的 Kerberos 或明文认证
- SSL – 基于TLS的加密，可选的身份验证

> *警告*
>
> *启用 SSL 时性能会下降，影响大小取决于 CPU 和 JVM 实现。参考 Kafka security overview 和 KAFKA-2561 。*

使用TLS

请阅读 Configuring Kafka Clients SSL 中描述的步骤来了解用于微调的其他配置设置，例如下面的几个例子：启用安全策略、密码套件、启用协议、truststore或秘钥库类型。

服务端认证和数据加密的一个配置实例：

```
a1.sinks.sink1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.sink1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
a1.sinks.sink1.kafka.topic = mytopic
a1.sinks.sink1.kafka.producer.security.protocol = SSL
<em># 如果在全局配置了SSL下面两个参数可省略，但是如果想使用自己独立的truststore，就可以把这两个参数加上。</em>
a1.sinks.sink1.kafka.producer.ssl.truststore.location = /path/to/truststore.jks
a1.sinks.sink1.kafka.producer.ssl.truststore.password = <password to access the truststore>
```

如果配置了全局ssl，上面关于ssl的配置就可以省略了，想了解更多可以参考 SSL/TLS 支持 。 注意，默认情况下 ssl.endpoint.identification.algorithm 这个参数没有被定义，因此不会执行主机名验证。如果要启用主机名验证，请加入以下配置：

```
a1.sinks.sink1.kafka.producer.ssl.endpoint.identification.algorithm = HTTPS
```

开启后，客户端将根据以下两个字段之一验证服务器的完全限定域名（FQDN）：

1. Common Name (CN) https://tools.ietf.org/html/rfc6125#section-2.3
2. Subject Alternative Name (SAN) https://tools.ietf.org/html/rfc5280#section-4.2.1.6

如果还需要客户端身份验证，则还应在 Flume 配置中添加以下内容，当然如果配置了全局ssl就不必另外配置了，想了解更多可以参考 SSL/TLS 支持 。 每个Flume实例都必须拥有其客户证书，来被Kafka 实例单独或通过其签名链来信任。 常见示例是由 Kafka 信任的单个根CA签署每个客户端证书。

```
<em># 如果在全局配置了SSL下面两个参数可省略，但是如果想使用自己独立的truststore，就可以把这两个参数加上。</em>
a1.sinks.sink1.kafka.producer.ssl.keystore.location = /path/to/client.keystore.jks
a1.sinks.sink1.kafka.producer.ssl.keystore.password = <password to access the keystore>
```

如果密钥库和密钥使用不同的密码保护，则 *ssl.key.password* 属性将为生产者密钥库提供所需的额外密码：

```
a1.sinks.sink1.kafka.producer.ssl.key.password = <password to access the key>
```

Kerberos安全配置：要将Kafka Sink 与使用 Kerberos 保护的Kafka群集一起使用，请为生产者设置上面提到的 *producer.security.protocol* 属性。 与 Kafka 实例一起使用的 Kerberos keytab 和主体在 JAAS 文件的“KafkaClient”部分中指定。

“客户端”部分描述了 Zookeeper 连接信息（如果需要）。 有关 JAAS 文件内容的信息，请参阅 Kafka doc。 可以通过 flume-env.sh 中的 JAVA_OPTS 指定此 JAAS 文件的位置以及系统范围的 kerberos 配置：

```
JAVA_OPTS="$JAVA_OPTS -Djava.security.krb5.conf=/path/to/krb5.conf"
JAVA_OPTS="$JAVA_OPTS -Djava.security.auth.login.config=/path/to/flume_jaas.conf"
```

使用 SASL_PLAINTEXT 的示例安全配置：

```
a1.sinks.sink1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.sink1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
a1.sinks.sink1.kafka.topic = mytopic
a1.sinks.sink1.kafka.producer.security.protocol = SASL_PLAINTEXT
a1.sinks.sink1.kafka.producer.sasl.mechanism = GSSAPI
a1.sinks.sink1.kafka.producer.sasl.kerberos.service.name = kafka
```

使用 SASL_SSL 的安全配置范例：

```
a1.sinks.sink1.type = org.apache.flume.sink.kafka.KafkaSink
a1.sinks.sink1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
a1.sinks.sink1.kafka.topic = mytopic
a1.sinks.sink1.kafka.producer.security.protocol = SASL_SSL
a1.sinks.sink1.kafka.producer.sasl.mechanism = GSSAPI
a1.sinks.sink1.kafka.producer.sasl.kerberos.service.name = kafka
<em># 如果在全局配置了SSL下面两个参数可省略，但是如果想使用自己独立的truststore，就可以把这两个参数加上。</em>
a1.sinks.sink1.kafka.producer.ssl.truststore.location = /path/to/truststore.jks
a1.sinks.sink1.kafka.producer.ssl.truststore.password = <password to access the truststore>
```

JAAS 文件配置示例。有关其内容的参考，请参阅Kafka文档 SASL configuration 中关于所需认证机制（GSSAPI/PLAIN）的客户端配置部分。 与 Kafka Source 和 Kafka Channel 不同，“Client”部分并不是必须的，除非其他组件需要它，否则不必要这样做。 另外，请确保 Flume 进程的操作系统用户对 JAAS 和 keytab 文件具有读权限。

```
KafkaClient {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=<strong>true</strong>
  storeKey=<strong>true</strong>
  keyTab="/path/to/keytabs/flume.keytab"
  principal="flume/flumehost1.example.com@YOURKERBEROSREALM";
};
```

HTTP Sink

HTTP Sink 从 channel 中获取 Event，然后再向远程 HTTP 接口 POST 发送请求，Event 内容作为 POST 的正文发送。

错误处理取决于目标服务器返回的HTTP响应代码。 Sink的 *退避* 和 *就绪* 状态是可配置的，事务提交/回滚结果以及Event是否发送成功在内部指标计数器中也是可配置的。

状态代码不可读的服务器返回的任何格式错误的 HTTP 响应都将产生 *退避* 信号，并且不会从 channel 中消耗该Event。

必需的参数已用 **粗体** 标明。

属性	默认值	解释

属性	默认值	解释
channel	–	与 Sink 绑定的 channel
type	–	组件类型，这个是： http .
endpoint	–	将要 POST 提交数据接口的绝对地址
connectTimeout	5000	连接超时（毫秒）
requestTimeout	5000	一次请求操作的最大超时时间（毫秒）
contentTypeHeader	text/plain	HTTP请求的Content-Type请求头
acceptHeader	text/plain	HTTP请求的Accept 请求头
defaultBackoff	true	是否默认启用退避机制，如果配置的 *backoff.CODE* 没有匹配到某个 http 状态码，默认就会使用这个参数值来决定是否退避
defaultRollback	true	是否默认启用回滚机制，如果配置的 *rollback.CODE* 没有匹配到某个 http 状态码，默认会使用这个参数值来决定是否回滚
defaultIncrementMetrics	false	是否默认进行统计计数，如果配置的 *incrementMetrics.CODE* 没有匹配到某个 http 状态码，默认会使用这个参数值来决定是否参与计数
backoff.CODE	–	配置某个 http 状态码是否启用退避机制（支持200这种精确匹配和2XX一组状态码匹配模式）
rollback.CODE	–	配置某个 http 状态码是否启用回滚机制（支持200这种精确匹配和2XX一组状态码匹配模式）
incrementMetrics.CODE	–	配置某个 http 状态码是否参与计数（支持200这种精确匹配和2XX一组状态码匹配模式）

注意 backoff， rollback 和 incrementMetrics 的 code 配置通常都是用具体的HTTP状态码，如果2xx和200这两种配置同时存在，则200的状态码会被精确匹配，其余200~299（除了200以外）之间的状态码会被2xx匹配。

> *提示*
>
> *Flume里面好多组件都有这个退避机制，其实就是下一级目标没有按照预期执行的时候，会执行一个延迟操作。比如向HTTP接口提交数据发生了错误触发了退避机制生效，系统等待30秒再执行后续的提交操作， 如果再次发生错误则等待的时间会翻倍，直到达到系统设置的最大等待上限。通常在重试成功后退避就会被重置， 下次遇到错误重新开始计算等待的时间。*

任何空的或者为 null 的 Event 不会被提交到HTTP接口上。

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = http
a1.sinks.k1.channel = c1
a1.sinks.k1.endpoint = http://localhost:8080/someuri
a1.sinks.k1.connectTimeout = 2000
a1.sinks.k1.requestTimeout = 2000
a1.sinks.k1.acceptHeader = application/json
a1.sinks.k1.contentTypeHeader = application/json
a1.sinks.k1.defaultBackoff = true
a1.sinks.k1.defaultRollback = true
a1.sinks.k1.defaultIncrementMetrics = false
a1.sinks.k1.backoff.4XX = false
a1.sinks.k1.rollback.4XX = false
a1.sinks.k1.incrementMetrics.4XX = true
a1.sinks.k1.backoff.200 = false
a1.sinks.k1.rollback.200 = false
a1.sinks.k1.incrementMetrics.200 = true
```

Custom Sink

你可以自己写一个 Sink 接口的实现类。启动 Flume 时候必须把你自定义 Sink 所依赖的其他类配置进 classpath 内。custom source 在写配置文件的 type 时候填你的全限定类名。 必需的参数已用 **粗体** 标明。

属性	**默认值**	**解释**
channel	–	与 Sink 绑定的 channe
type	–	组件类型，这个填你自定义class的全限定类名

配置范例：

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = org.example.MySink
a1.sinks.k1.channel = c1
```

Flume Channels

channel 是在 Agent 上暂存 Event 的缓冲池。 Event由source添加，由sink消费后删除。

Memory Channel

内存 channel 是把 Event 队列存储到内存上，队列的最大数量就是 *capacity* 的设定值。它非常适合对吞吐量有较高要求的场景，但也是有代价的，当发生故障的时候会丢失当时内存中的所有 Event。 必需的参数已用 **粗体** 标明。

属性	默认值	解释
type	–	组件类型，这个是： memory
capacity	100	内存中存储 Event 的最大数
transactionCapacity	100	source 或者 sink 每个事务中存取 Event 的操作数量（不能比 *capacity* 大）
keep-alive	3	添加或删除一个 Event 的超时时间（秒）
byteCapacityBufferPercentage	20	指定 Event header 所占空间大小与 channel 中所有 Event 的总大小之间的百分比

属性	默认值	解释
byteCapacity		Channel 中最大允许存储所有 Event 的总字节数（bytes）。默认情况下会使用JVM可用内存的80%作为最大可用内存（就是JVM启动参数里面配置的-Xmx的值）。 计算总字节时只计算 Event 的主体，这也是提供 *byteCapacityBufferPercentage* 配置参数的原因。注意，当你在一个 Agent 里面有多个内存 channel 的时候， 而且碰巧这些 channel 存储相同的物理 Event（例如：这些 channel 通过复制机制（复制选择器）接收同一个 source 中的 Event）， 这时候这些 Event 占用的空间是累加的，并不会只计算一次。如果这个值设置为0（不限制），就会达到200G左右的内部硬件限制。

> *提示*
>
> *举2个例子来帮助理解最后两个参数吧：*
>
> *两个例子都有共同的前提，假设JVM最大的可用内存是100M（或者说JVM启动时指定了-Xmx=100m）。*
>
> *例子1：* byteCapacityBufferPercentage *设置为20，* byteCapacity *设置为52428800（就是50M），此时内存中所有 Event body 的总大小就被限制为50M * （1-20%）=40M，内存channel可用内存是50M。*
>
> *例子2：* byteCapacityBufferPercentage *设置为10，* byteCapacity *不设置，此时内存中所有 Event body 的总大小就被限制为100M * 80% * （1-10%）=72M，内存channel可用内存是80M。*

配置范例：

```
a1.channels = c1
a1.channels.c1.type = memory
a1.channels.c1.capacity = 10000
a1.channels.c1.transactionCapacity = 10000
a1.channels.c1.byteCapacityBufferPercentage = 20
a1.channels.c1.byteCapacity = 800000
```

JDBC Channel

JDBC Channel会通过一个数据库把Event持久化存储。目前只支持Derby。这是一个可靠的channel，非常适合那些注重可恢复性的流使用。 必需的参数已用 **粗体** 标明。

属性	默认值	解释
type	–	组件类型，这个是： jdbc
db.type	DERBY	使用的数据库类型，目前只支持 DERBY.
driver.class	org.apache.derby.jdbc.EmbeddedDriver	所使用数据库的 JDBC 驱动类
driver.url	(constructed from other properties)	JDBC 连接的 URL
db.username	“sa”	连接数据库使用的用户名
db.password	–	连接数据库使用的密码
connection.properties.file	–	JDBC连接属性的配置文件
create.schema	true	如果设置为 true ，没有数据表的时候会自动创建
create.index	true	是否创建索引来加快查询速度
create.foreignkey	true	是否创建外键
transaction.isolation	“READ_COMMITTED”	面向连接的隔离级别，可选值： READ_UNCOMMITTED ， READ_COMMITTED ， SERIALIZABLE ， REPEATABLE_READ
maximum.connections	10	数据库的最大连接数
maximum.capacity	0 (unlimited)	channel 中存储 Event 的最大数
sysprop.*		针对不同DB的特定属性
sysprop.user.home		Derby 的存储主路径

配置范例：

```
a1.channels = c1
a1.channels.c1.type = jdbc
```

Kafka Channel

将 Event 存储到Kafka集群（必须单独安装）。Kafka提供了高可用性和复制机制，因此如果Flume实例或者 Kafka 的实例挂掉，能保证Event数据随时可用。 Kafka channel可以用于多种场景：

1. 与source和sink一起：给所有Event提供一个可靠、高可用的channel。
2. 与source、interceptor一起，但是没有sink：可以把所有Event写入到Kafka的topic中，来给其他的应用使用。
3. 与sink一起，但是没有source：提供了一种低延迟、容错高的方式将Event发送的各种Sink上，比如：HDFS、HBase、Solr。

目前支持Kafka 0.10.1.0以上版本，最高已经在Kafka 2.0.1版本上完成了测试，这已经是Flume 1.9发行时候的最高的Kafka版本了。

配置参数组织如下：

1. 通常与channel相关的配置值应用于channel配置级别，比如：a1.channel.k1.type =
2. 与Kafka相关的配置值或Channel运行的以“kafka.”为前缀（这与CommonClient Configs类似），例如：*a1.channels.k1.kafka.topic*和 *a1.channels.k1.kafka.bootstrap.servers*。 这与hdfs sink的运行方式没有什么不同
3. 特定于生产者/消费者的属性以kafka.producer或kafka.consumer为前缀
4. 可能的话，使用Kafka的参数名称，例如：bootstrap.servers 和 acks

当前Flume版本是向下兼容的，但是第二个表中列出了一些不推荐使用的属性，并且当它们出现在配置文件中时，会在启动时打印警告日志。

必需的参数已用 **粗体** 标明。

属性	默认值	解释
type	–	组件类型，这个是： org.apache.flume.channel.kafka.KafkaChannel
kafka.bootstrap.servers	–	channel使用的Kafka集群的实例列表，可以是实例的部分列表。但是更建议至少两个用于高可用支持。格式为hostname:port，多个用逗号分隔
kafka.topic	flume-channel	channel使用的Kafka topic
kafka.consumer.group.id	flume	channel 用于向 Kafka 注册的消费者群组ID。 多个 channel 必须使用相同的 topic 和 group，以确保当一个Flume实例发生故障时，另一个实例可以获取数据。请注意，使用相同组ID的非channel消费者可能会导致数据丢失。
parseAsFlumeEvent	true	是否以avro基准的 Flume Event 格式在channel中存储Event。 如果是Flume的Source向channel的topic写入Event则应设置为true； 如果其他生产者也在向channel的topic写入Event则应设置为false。 通过使用flume-ng-sdk 中的 *org.apache.flume.source.avro.AvroFlumeEvent* 可以在Kafka之外解析出Flume source的信息。
pollTimeout	500	消费者调用poll()方法时的超时时间（毫秒） https://kafka.apache.org/090/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html#poll(long)
defaultPartitionId	–	指定channel中所有Event将要存储的分区ID，除非被 *partitionIdHeader* 参数的配置覆盖。 默认情况下，如果没有设置此参数，Event 会被Kafka生产者的分发程序分发，包括key（如果指定了的话），或者被 *kafka.partitioner.class* 指定的分发程序来分发。

属性	默认值	解释
partitionIdHeader	–	从Event header中读取要存储Event到目标Kafka的分区的属性名。 如果设置了，生产者会从Event header中获取次属性的值，并将消息发送到topic的指定分区。 如果该值表示的分区无效，则Event不会存入channel。如果该值有效，则会覆盖 *defaultPartitionId* 配置的分区ID。
kafka.consumer.auto.offset.reset	latest	当Kafka中没有初始偏移量或者当前偏移量已经不在当前服务器上时（比如数据已经被删除）该怎么办。 earliest：自动重置偏移量到最早的位置； latest：自动重置偏移量到最新的位置； none：如果没有为消费者的组找到任何先前的偏移量，则向消费者抛出异常； else：向消费者抛出异常。
kafka.producer.security.protocol	PLAINTEXT	设置使用哪种安全协议写入Kafka。可选值： SASL_PLAINTEXT 、 SASL_SSL 和 SSL 有关安全设置的其他信息，请参见下文。
kafka.consumer.security.protocol	PLAINTEXT	与上面的相同，只不过是用于消费者。
more producer/consumer security props		如果使用了 SASL_PLAINTEXT 、 SASL_SSL 或 SSL 等安全协议，参考 Kafka security 来为生产者、消费者增加安全相关的参数配置

下表是弃用的一些参数

属性	默认值	解释
brokerList	–	改用 kafka.bootstrap.servers
topic	flume-channel	改用 kafka.topic
groupId	flume	改用 kafka.consumer.group.id

属性	默认值	解释
readSmallestOffset	false	改用 kafka.consumer.auto.offset.reset
migrateZookeeperOffsets	true	如果找不到Kafka存储的偏移量，去Zookeeper中查找偏移量并将它们提交给 Kafka 。 它应该设置为true以支持从旧版本的FlumeKafka客户端无缝迁移。 迁移后，可以将其设置为false，但通常不需要这样做。 如果在Zookeeper未找到偏移量，则可通过 *kafka.consumer.auto.offset.reset* 配置如何处理偏移量。

> *注解*
>
> *由于channel是负载均衡的，第一次启动时可能会有重复的Event出现。*

配置范例：

```
a1.channels.channel1.type = org.apache.flume.channel.kafka.KafkaChannel
a1.channels.channel1.kafka.bootstrap.servers = kafka-1:9092,kafka-2:9092,kafka-3:9092
a1.channels.channel1.kafka.topic = channel1
a1.channels.channel1.kafka.consumer.group.id = flume-consumer
```

安全与加密：

Flume 和 Kafka 之间通信渠道是支持安全认证和数据加密的。对于身份安全验证，可以使用 Kafka 0.9.0版本中的 SASL、GSSAPI（Kerberos V5） 或 SSL （虽然名字是SSL，实际是TLS实现）。

截至目前，数据加密仅由SSL / TLS提供。

当你把 *kafka.producer（consumer）.security.protocol* 设置下面任何一个值的时候意味着：

- SASL_PLAINTEXT – 无数据加密的 Kerberos 或明文认证

- SASL_SSL – 有数据加密的 Kerberos 或明文认证
- SSL – 基于TLS的加密，可选的身份验证。

> *警告*
>
> *启用 SSL 时性能会下降，影响大小取决于 CPU 和 JVM 实现。参考 Kafka security overview 和 KAFKA-2561 。*

使用TLS：

请阅读 Configuring Kafka Clients SSL 中描述的步骤来了解用于微调的其他配置设置，例如下面的几个例子：启用安全策略、密码套件、启用协议、truststore或秘钥库类型。

服务端认证和数据加密的一个配置实例：

```
a1.channels.channel1.type = org.apache.flume.channel.kafka.KafkaChannel
a1.channels.channel1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
a1.channels.channel1.kafka.topic = channel1
a1.channels.channel1.kafka.consumer.group.id = flume-consumer
a1.channels.channel1.kafka.producer.security.protocol = SSL
<em># 如果在全局配置了SSL下面两个参数可省略，但是如果想使用自己独立的truststore，就可以把这两个参数加上。</em>
a1.channels.channel1.kafka.producer.ssl.truststore.location = /path/to/truststore.jks
a1.channels.channel1.kafka.producer.ssl.truststore.password = <password to access the truststore>
a1.channels.channel1.kafka.consumer.security.protocol = SSL
a1.channels.channel1.kafka.consumer.ssl.truststore.location = /path/to/truststore.jks
a1.channels.channel1.kafka.consumer.ssl.truststore.password = <password to access the truststore>
```

如果配置了全局ssl，上面关于ssl的配置就可以省略了，想了解更多可以参考 SSL/TLS 支持 。 注意，默认情况下 ssl.endpoint.identification.algorithm 这个参数没有被定义，因此不会执行主机名验证。如果要启用主机名验证，请加入以下配置：

```
a1.channels.channel1.kafka.producer.ssl.endpoint.identification.algorithm = HTTPS
a1.channels.channel1.kafka.consumer.ssl.endpoint.identification.algorithm = HTTPS
```

开启后，客户端将根据以下两个字段之一验证服务器的完全限定域名（FQDN）：

1. Common Name (CN) https://tools.ietf.org/html/rfc6125#section-2.3
2. Subject Alternative Name (SAN) https://tools.ietf.org/html/rfc5280#section-4.2.1.6

如果还需要客户端身份验证，则还应在 Flume 配置中添加以下内容，当然如果配置了全局ssl就不必另外配置了，想了解更多可以参考 SSL/TLS 支持 。 每个Flume 实例都必须拥有其客户证书，来被Kafka 实例单独或通过其签名链来信任。 常见示例是由 Kafka 信任的单个根CA签署每个客户端证书。

```
<em># 如果在全局配置了SSL下面几个参数可省略，但是如果想使用自己独立的truststore，就可以把这两个参数加上。</em>
a1.channels.channel1.kafka.producer.ssl.keystore.location = /path/to/client.keystore.jks
a1.channels.channel1.kafka.producer.ssl.keystore.password = <password to access the keystore>
a1.channels.channel1.kafka.consumer.ssl.keystore.location = /path/to/client.keystore.jks
a1.channels.channel1.kafka.consumer.ssl.keystore.password = <password to access the keystore>
```

如果密钥库和密钥使用不同的密码保护，则ssl.key.password 属性将为消费者和生产者密钥库提供所需的额外密码：

```
a1.channels.channel1.kafka.producer.ssl.key.password = <password to access the key>
a1.channels.channel1.kafka.consumer.ssl.key.password = <password to access the key>
```

Kerberos**安全配置：**

要将Kafka channel 与使用Kerberos保护的Kafka群集一起使用，请为生产者或消费者设置上面提到的producer（consumer）.security.protocol属性。 与Kafka实例一起使用的Kerberos keytab和主体在JAAS文件的“KafkaClient”部分中指定。 “客户端”部分描述了Zookeeper连接信息（如果需要）。 有关JAAS文件内容的信息，请参阅 Kafka doc。 可以通过flume-env.sh中的JAVA_OPTS指定此JAAS文件的位置以及系统范围的 kerberos 配置：

```
JAVA_OPTS="$JAVA_OPTS -Djava.security.krb5.conf=/path/to/krb5.conf"
JAVA_OPTS="$JAVA_OPTS -Djava.security.auth.login.config=/path/to/flume_jaas.conf"
```

使用 SASL_PLAINTEXT 的示例安全配置：

```
a1.channels.channel1.type = org.apache.flume.channel.kafka.KafkaChannel
a1.channels.channel1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
a1.channels.channel1.kafka.topic = channel1
a1.channels.channel1.kafka.consumer.group.id = flume-consumer
a1.channels.channel1.kafka.producer.security.protocol = SASL_PLAINTEXT
a1.channels.channel1.kafka.producer.sasl.mechanism = GSSAPI
a1.channels.channel1.kafka.producer.sasl.kerberos.service.name = kafka
a1.channels.channel1.kafka.consumer.security.protocol = SASL_PLAINTEXT
a1.channels.channel1.kafka.consumer.sasl.mechanism = GSSAPI
a1.channels.channel1.kafka.consumer.sasl.kerberos.service.name = kafka
```

使用 SASL_SSL 的安全配置范例：

```
a1.channels.channel1.type = org.apache.flume.channel.kafka.KafkaChannel
a1.channels.channel1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
a1.channels.channel1.kafka.topic = channel1
a1.channels.channel1.kafka.consumer.group.id = flume-consumer
a1.channels.channel1.kafka.producer.security.protocol = SASL_SSL
a1.channels.channel1.kafka.producer.sasl.mechanism = GSSAPI
a1.channels.channel1.kafka.producer.sasl.kerberos.service.name = kafka
<em># 如果在全局配置了SSL下面两个参数可省略，但是如果想使用自己独立的truststore，就可以把这两个参数加上。</em>
a1.channels.channel1.kafka.producer.ssl.truststore.location = /path/to/truststore.jks
a1.channels.channel1.kafka.producer.ssl.truststore.password = <password to access the truststore>
a1.channels.channel1.kafka.consumer.security.protocol = SASL_SSL
a1.channels.channel1.kafka.consumer.sasl.mechanism = GSSAPI
a1.channels.channel1.kafka.consumer.sasl.kerberos.service.name = kafka
<em># 如果在全局配置了SSL下面两个参数可省略，但是如果想使用自己独立的truststore，就可以把这两个参数加上。</em>
a1.channels.channel1.kafka.consumer.ssl.truststore.location = /path/to/truststore.jks
a1.channels.channel1.kafka.consumer.ssl.truststore.password = <password to access the truststore>
```

JAAS 文件配置示例。有关其内容的参考，请参阅Kafka文档 SASL configuration 中关于所需认证机制（GSSAPI/PLAIN）的客户端配置部分。由于Kafka Source 也可以连接 Zookeeper 以进行偏移迁移，因此"Client"部分也添加到此示例中。 除非您需要偏移迁移，否则不必要这样做，或者您需要此部分用于其他安全组件。 另外，请确保Flume进程的操作系统用户对 JAAS 和 keytab 文件具有读权限。

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=<strong>true</strong>
  storeKey=<strong>true</strong>
  keyTab="/path/to/keytabs/flume.keytab"
  principal="flume/flumehost1.example.com@YOURKERBEROSREALM";
};

KafkaClient {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=<strong>true</strong>
  storeKey=<strong>true</strong>
  keyTab="/path/to/keytabs/flume.keytab"
  principal="flume/flumehost1.example.com@YOURKERBEROSREALM";
};
```

File Channel

必需的参数已用 **粗体** 标明。

属性	**默认值**	**解释**
type	–	组件类型，这个是： file .
checkpointDir	~/.flume/file-channel/checkpoint	记录检查点的文件的存储目录
useDualCheckpoints	false	是否备份检查点文件。如果设置为 true ， *backupCheckpointDir* 参数必须设置。
backupCheckpointDir	–	备份检查点的目录。 此目录不能与**数据目录**或检查点目录 *checkpointDir* 相同
dataDirs	~/.flume/file-channel/data	逗号分隔的目录列表，用于存储日志文件。 在不同物理磁盘上使用多个目录可以提高文件channel的性能
transactionCapacity	10000	channel支持的单个事务最大容量
checkpointInterval	30000	检查点的时间间隔（毫秒）
maxFileSize	2146435071	单个日志文件的最大字节数。这个默认值约等于2047MB
minimumRequiredSpace	524288000	最小空闲空间的字节数。为了避免数据损坏，当空闲空间低于这个值的时候，文件channel将拒绝一切存取请求
capacity	1000000	channel的最大容量
keep-alive	3	存入Event的最大等待时间（秒）
use-log-replay-v1	false	（专家）是否使用老的回放逻辑 （Flume默认是使用v2版本的回放方法，但是如果v2版本不能正常工作可以考虑通过这个参数改为使用v1版本，v1版本是从Flume1.2开始启用的，回放是指系统关闭或者崩溃前执行的校验检查点文件和文件channel记录是否一致程序）
use-fast-replay	false	（专家）是否开启快速回放（不适用队列）

属性	默认值	解释
checkpointOnClose	true	channel关闭时是否创建检查点文件。开启次功能可以避免回放提高下次文件channel启动的速度
encryption.activeKey	–	加密数据所使用的key名称
encryption.cipherProvider	–	加密类型，目前只支持： AESCTRNOPADDING
encryption.keyProvider	–	key类型，目前只支持： JCEKSFILE
encryption.keyProvider.keyStoreFile	–	keystore 文件路径
encrpytion.keyProvider.keyStorePasswordFile	–	keystore 密码文件路径
encryption.keyProvider.keys	–	所有key的列表，包含所有使用过的加密key名称
encyption.keyProvider.keys.*.passwordFile	–	可选的秘钥密码文件路径

> *注解*
>
> *默认情况下，文件channel使用默认的用户主目录内的检查点和数据目录的路径（说的就是上面的checkpointDir参数的默认值）。 如果一个Agent中有多个活动的文件channel实例，而且都是用了默认的检查点文件， 则只有一个实例可以锁定目录并导致其他channel初始化失败。 因此，这时候有必要为所有已配置的channel显式配置不同的检查点文件目录，最好是在不同的磁盘上。 此外，由于文件channel将在每次提交后会同步到磁盘，因此将其与将Event一起批处理的sink/source耦合可能是必要的，以便在多个磁盘不可用于检查点和数据目录时提供良好的性能。*

配置范例：

```
a1.channels = c1
a1.channels.c1.type = file
a1.channels.c1.checkpointDir = /mnt/flume/checkpoint
a1.channels.c1.dataDirs = /mnt/flume/data
```

Encryption

下面是几个加密的例子：

用给定的秘钥库密码生成秘钥key-0：

```
keytool -genseckey -alias key-0 -keypass keyPassword -keyalg AES <strong>\</strong>
  -keysize 128 -validity 9000 -keystore test.keystore <strong>\</strong>
  -storetype jceks -storepass keyStorePassword
```

使用相同的秘钥库密码生成秘钥key-1：

```
keytool -genseckey -alias key-1 -keyalg AES -keysize 128 -validity 9000 <strong>\</strong>
  -keystore src/test/resources/test.keystore -storetype jceks <strong>\</strong>
  -storepass keyStorePassword
```

```
a1.channels.c1.encryption.activeKey = key-0
a1.channels.c1.encryption.cipherProvider = AESCTRNOPADDING
a1.channels.c1.encryption.keyProvider = key-provider-0
a1.channels.c1.encryption.keyProvider = JCEKSFILE
a1.channels.c1.encryption.keyProvider.keyStoreFile = /path/to/my.keystore
a1.channels.c1.encryption.keyProvider.keyStorePasswordFile = /path/to/my.keystore.password
a1.channels.c1.encryption.keyProvider.keys = key-0
```

假设你已不再使用key-0，并且已经使用key-1加密新文件：

```
a1.channels.c1.encryption.activeKey = key-1
a1.channels.c1.encryption.cipherProvider = AESCTRNOPADDING
a1.channels.c1.encryption.keyProvider = JCEKSFILE
a1.channels.c1.encryption.keyProvider.keyStoreFile = /path/to/my.keystore
a1.channels.c1.encryption.keyProvider.keyStorePasswordFile = /path/to/my.keystore.password
a1.channels.c1.encryption.keyProvider.keys = key-0 key-1
```

跟上面一样的场景，只不过key-0有自己单独的密码：

```
a1.channels.c1.encryption.activeKey = key-1
a1.channels.c1.encryption.cipherProvider = AESCTRNOPADDING
a1.channels.c1.encryption.keyProvider = JCEKSFILE
a1.channels.c1.encryption.keyProvider.keyStoreFile = /path/to/my.keystore
a1.channels.c1.encryption.keyProvider.keyStorePasswordFile = /path/to/my.keystore.password
a1.channels.c1.encryption.keyProvider.keys = key-0 key-1
a1.channels.c1.encryption.keyProvider.keys.key-0.passwordFile = /path/to/key-0.password
```

Spillable Memory Channel

这个channel会将Event存储在内存队列和磁盘上。 内存队列充当主存储，内存装满之后会存到磁盘。 磁盘存储使用嵌入的文件channel进行管理。 当内存队列已满时，其他传入Event将存储在文件channel中。 这个channel非常适用于需要高吞吐量存储器channel的流，但同时需要更大容量的文件channel，以便更好地容忍间歇性目的地侧（sink）中断或消费速率降低。 在这种异常情况下，吞吐量将大致降低到文件channel速度。 如果Agent程序崩溃或重新启动，只有存储在磁盘上的Event能恢复。 **这个channel目前是实验性的，不建议用于生产环境** 。

> *提示*
>
> *这个channel的机制十分像Windows系统里面的「虚拟内存」。兼顾了内存channel的高吞吐量和文件channel的可靠、大容量优势。*

必需的参数已用 **粗体** 标明。有关其他必需属性，请参阅文件channel。

属性	**默认值**	**解释**
type	–	组件类型，这个是： SPILLABLEMEMORY
memoryCapacity	10000	内存队列存储的Event最大数量。如果设置为0，则会禁用内存队列。
overflowCapacity	100000000	磁盘（比如文件channel）上存储Event的最大数量，如果设置为0，则会禁用磁盘存储
overflowTimeout	3	当内存占满时启用磁盘存储之前等待的最大秒数
byteCapacityBufferPercentage	20	指定Event header所占空间大小与channel中所有Event的总大小之间的百分比

属性	默认值	解释
byteCapacity		内存中最大允许存储Event的总字节数。 默认情况下会使用JVM可用内存的80%作为最大可用内存（就是JVM启动参数里面配置的-Xmx的值）。 计算总字节时只计算Event的主体，这也是提供 *byteCapacityBufferPercentage* 配置参数的原因。注意，当你在一个Agent里面有多个内存channel的时候，而且碰巧这些channel存储相同的物理Event（例如：这些channel通过复制机制（复制选择器）接收同一个source中的Event）， 这时候这些Event占用的空间是累加的，并不会只计算一次。如果这个值设置为0（不限制），就会达到200G左右的内部硬件限制。
avgEventSize	500	估计进入channel的Event的平均大小（单位：字节）
<file channel properties>	see file channel	可以使用除“keep-alive”和“capacity”之外的任何文件channel属性。 文件channel的“keep-alive”由Spillable Memory Channel管理， 而channel容量则是通过使用 *overflowCapacity* 来设置。

如果达到 *memoryCapacity* 或 *byteCapacity* 限制，则内存队列被视为已满。

配置范例：

```
a1.channels = c1
a1.channels.c1.type = SPILLABLEMEMORY
a1.channels.c1.memoryCapacity = 10000
a1.channels.c1.overflowCapacity = 1000000
a1.channels.c1.byteCapacity = 800000
a1.channels.c1.checkpointDir = /mnt/flume/checkpoint
a1.channels.c1.dataDirs = /mnt/flume/data
```

禁用内存channel，只使用磁盘存储（就像文件channel那样）的例子：

```
a1.channels = c1
a1.channels.c1.type = SPILLABLEMEMORY
a1.channels.c1.memoryCapacity = 0
a1.channels.c1.overflowCapacity = 1000000
a1.channels.c1.checkpointDir = /mnt/flume/checkpoint
a1.channels.c1.dataDirs = /mnt/flume/data
```

禁用掉磁盘存储，只使用内存channel的例子：

```
a1.channels = c1
a1.channels.c1.type = SPILLABLEMEMORY
a1.channels.c1.memoryCapacity = 100000
a1.channels.c1.overflowCapacity = 0
```

Pseudo Transaction Channel

> *警告*
>
> *这个伪事务 channel 仅用于单元测试目的，不适用于生产用途。*

必需的参数已用 **粗体** 标明。

属性	默认值	解释
type	–	组件类型，这个是： org.apache.flume.channel.PseudoTxnMemoryChannel
capacity	50	channel中存储的最大Event数
keep-alive	3	添加或删除Event的超时时间（秒）

Custom Channel

可以自己实现Channel接口来自定义一个channel，启动时这个自定义channel类以及依赖必须都放在flume Agent的classpath中。 必需的参数已用 **粗体** 标明。

属性	默认值	解释
type	–	你自己实现的channel类的全限定类名，比如：org.example.myCustomChannel

配置范例：

```
a1.channels = c1
a1.channels.c1.type = org.example.MyChannel
```

Flume Channel Selectors

如果没有手动配置，source的默认channel选择器类型是replicating（复制），当然这个选择器只针对source配置了多个channel的时候。

> *提示*
>
> *既然叫做channel选择器，很容易猜得到这是source才有的配置。前面介绍过，一个souce可以向多个channel同时写数据，所以也就产生了以何种方式向多个channel写的问题（比如自带的 复制选择器 ， 会把数据完整地发送到每一个channel，而 多路复用选择器 就可以通过配置来按照一定的规则进行分发，听起来很像负载均衡），channel选择器也就应运而生。*

复制选择器

它是默认的选择器。

必需的参数已用 **粗体** 标明。

属性	默认值	解释
selector.type	replicating	replicating
selector.optional	–	指定哪些channel是可选的，多个用空格分开

配置范例：

```
a1.sources = r1
a1.channels = c1 c2 c3
a1.sources.r1.selector.type = replicating
a1.sources.r1.channels = c1 c2 c3
a1.sources.r1.selector.optional = c3
```

上面这个例子中，c3配置成了可选的。向c3发送数据如果失败了会被忽略。c1和c2没有配置成可选的，向c1和c2写数据失败会导致事务失败回滚。

多路复用选择器

必需的参数已用 **粗体** 标明。

属性	默认值	解释
selector.type	replicating	组件类型，这个是： multiplexing
selector.header	flume.selector.header	想要进行匹配的header属性的名字
selector.default	–	指定一个默认的channel。如果没有被规则匹配到，默认会发到这个channel上
selector.mapping.*	–	一些匹配规则，具体参考下面的例子

配置范例：

```
a1.sources = r1
a1.channels = c1 c2 c3 c4
a1.sources.r1.selector.type = multiplexing
a1.sources.r1.selector.header = state          #以每个Event的header中的state这个属性的值作为选择channel的依据
a1.sources.r1.selector.mapping.CZ = c1         #如果state=CZ，则选择c1这个channel
a1.sources.r1.selector.mapping.US = c2 c3      #如果state=US，则选择c2 和 c3 这两个channel
a1.sources.r1.selector.default = c4             #默认使用c4这个channel
```

自定义选择器

自定义选择器就是你可以自己写一个 *org.apache.flume.ChannelSelector* 接口的实现类。老规矩，你自己写的实现类以及依赖的jar包在启动时候都必须放入Flume的classpath。

属性	默认值	解释
selector.type	–	你写的自定义选择器的全限定类名，比如：org.liyifeng.flume.channel.MyChannelSelector

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.selector.type = org.liyifeng.flume.channel.MyChannelSelector
```

Sink组逻辑处理器

你可以把多个sink分成一个组， 这时候Sink组逻辑处理器（Flume Sink Processors）可以对这同一个组里的几个sink进行负载均衡或者其中一个sink发生故障后将输出Event的任务转移到其他的sink上。

提示

说的直白一些，这N个sink本来是要将Event输出到对应的N个目的地的，通过 Sink组逻辑处理器 就可以把这N个sink配置成负载均衡或者故障转移的工作方式（暂时还不支持自定义的）。 负载均衡就方式是把channel里面的Event按照配置的负载机制（比如轮询）分别发送到sink各自对应的目的地；故障转移就是这N个sink同一时间只有一个在工作，其余的作为备用，工作的sink挂掉之后备用的sink顶上。

必需的参数已用 **粗体** 标明。

属性	默认值	解释
sinks	–	这一组的所有sink名，多个用空格分开
processor.type	default	这个sink组的逻辑处理器类型，可选值 default （默认一对一的）、 failover （故障转移）、 load_balance （负载均衡）

配置范例：

```
a1.sinkgroups = g1
a1.sinkgroups.g1.sinks = k1 k2
a1.sinkgroups.g1.processor.type = load_balance
```

默认

默认的组逻辑处理器就是只有一个sink的情况（准确说这根本不算一个组），所以这种情况就没必要配置sink组了。本文档前面的例子都是 source – channel – sink这种一对一，单个sink的。

故障转移

故障转移组逻辑处理器维护了一个发送Event失败的sink的列表，保证有一个sink是可用的来发送Event。

故障转移机制的工作原理是将故障sink降级到一个池中，在池中为它们分配冷却期（超时时间），在重试之前随顺序故障而增加。 Sink成功发送事件后，它将恢复到实时池。sink具有与之相关的优先级，数值越大，优先级越高。 如果在发送Event时Sink发生故障，会继续尝试下一个具有最高优先级的sink。 例如，在优先级为80的sink之前激活优先级为100的sink。如果未指定优先级，则根据配置中的顺序来选取。

要使用故障转移选择器，不仅要设置sink组的选择器为failover，还有为每一个sink设置一个唯一的优先级数值。 可以使用 *maxpenalty* 属性设置故障转移时间的上限（毫秒）。

必需的参数已用 **粗体** 标明。

属性	**默认值**	**解释**
sinks	–	这一组的所有sink名，多个用空格分开
processor.type	default	组件类型，这个是： failover
processor.priority.<sinkName>	–	组内sink的权重值，<sinkName>必须是当前组关联的sink之一。数值越大越被优先使用
processor.maxpenalty	30000	发生异常的sink最大故障转移时间（毫秒）

配置范例：

```
a1.sinkgroups = g1
a1.sinkgroups.g1.sinks = k1 k2
a1.sinkgroups.g1.processor.type = failover
a1.sinkgroups.g1.processor.priority.k1 = 5
a1.sinkgroups.g1.processor.priority.k2 = 10
a1.sinkgroups.g1.processor.maxpenalty = 10000
```

负载均衡

负载均衡Sink 选择器提供了在多个sink上进行负载均衡流量的功能。 它维护一个活动sink列表的索引来实现负载的分配。 默认支持了轮询（ round_robin ）和随机（ random ）两种选择机制分配负载。 默认是轮询，可以通过配置来更改。也可以从 *AbstractSinkSelector* 继承写一个自定义的选择器。工作时，此选择器使用其配置的选择机制选择下一个sink并调用它。 如果所选sink无法正常工作，则处理器通过其配置的选择机制选择下一个可用sink。 此实现不会将失败的Sink列入黑名单，而是继续乐观地尝试每个可用的Sink。

如果所有sink调用都失败了，选择器会将故障抛给sink的运行器。

如果backoff设置为true则启用了退避机制，失败的sink会被放入黑名单，达到一定的超时时间后会自动从黑名单移除。 如从黑名单出来后sink仍然失败，则再次进入黑名单而且超时时间会翻倍，以避免在无响应的sink上浪费过长时间。 如果没有启用退避机制，在禁用此功能的情况下，发生sink传输失败后，会将本次负载传给下一个sink继续尝试，因此这种情况下是不均衡的。

必需的参数已用 **粗体** 标明。

属性	**默认值**	**解释**
processor.sinks	–	这一组的所有sink名，多个用空格分开
processor.type	default	组件类型，这个是： load_balance
processor.backoff	false	失败的sink是否成倍地增加退避它的时间。 如果设置为false，负载均衡在某一个sink发生异常后，下一次选择sink的时候仍然会将失败的这个sink加入候选队列； 如果设置为true，某个sink连续发生异常时会成倍地增加它的退避时间，在退避的时间内是无法参与负载均衡竞争的。退避机制只统计1个小时发生的异常，超过1个小时没有发生异常就会重新计算
processor.selector	round_robin	负载均衡机制，可选值： round_robin （轮询）、 random （随机选择）、「自定义选择器的全限定类名」：自定义的负载器要继承 *AbstractSinkSelector*

属性	默认值	解释
processor.selector.maxTimeOut	30000	发生异常的sink最长退避时间（毫秒） 如果设置了processor.backoff=true，某一个sink发生异常的时候就会触发自动退避它一段时间，这个 *maxTimeOut* 就是退避一个sink的最长时间

配置范例：

```
a1.sinkgroups = g1
a1.sinkgroups.g1.sinks = k1 k2
a1.sinkgroups.g1.processor.type = load_balance
a1.sinkgroups.g1.processor.backoff = true
a1.sinkgroups.g1.processor.selector = random
```

自定义

目前还不支持自定义Sink组逻辑处理器

例子

> *提示*
>
> *官方没有给出【Sink组逻辑处理器】完整的例子，本小节是我自己写的一个测试【故障转移】机制的例子供参考。*

```
<em># test-flume.properties</em>
<em># 首先定义出该agent实例数据流的所有组件，一个Source、一个Channel和两个Sink</em>
a1.sources = r1
a1.sinks = k1 k2
a1.channels = c1

<em># 使用Stress Source来生成测试用的event</em>
a1.sources.r1.type = org.apache.flume.source.StressSource
```

```
a1.sources.r1.maxEventsPerSecond= 1      # 限制测试的event生成速度，让它每秒生成1个，便于观察效果
a1.sources.r1.batchSize = 1              # 每次事务向Channel写入1个event
a1.sources.r1.maxTotalEvents = 100       # 总共会生成100个event
a1.sources.r1.channels = c1

<em># 两个Sink都是是File Roll Sink，用于把event存储到本地文件中</em>
a1.sinks.k1.type = file_roll
a1.sinks.k1.sink.batchSize = 1                                  # 每次事务从Channel获取1个event
a1.sinks.k1.sink.directory = /Users/liyifeng/testflumek1    # 存储event的目录
a1.sinks.k1.channel = c1

<em># 同上</em>
a1.sinks.k2.type = file_roll
a1.sinks.k2.sink.batchSize = 1
a1.sinks.k2.sink.directory = /Users/liyifeng/testflumek2
a1.sinks.k2.channel = c1

<em># 用的是内存Channel，没什么可说的</em>
a1.channels.c1.type = memory
a1.channels.c1.capacity = 2
a1.channels.c1.transactionCapacity = 1

<em># 重点来了，将两个Sink放在一个组g1中</em>
a1.sinkgroups = g1
a1.sinkgroups.g1.sinks = k1 k2
a1.sinkgroups.g1.processor.type = failover    # 该组的工作方式是故障转移
<em># 下面两个参数是可选的，我这里进行配置的原因是让Flume先使用k2工作，在k2工作的时候让它失败，之后再观察k1是否继续工作</em>
a1.sinkgroups.g1.processor.priority.k1 = 1      # 组内sink的权重值，数值越高越早被激活
a1.sinkgroups.g1.processor.priority.k2 = 10     # 本例中k2会率先工作
```

第一步启动Flume

```
$ bin/flume-ng agent -n a1 -c conf -f conf/test-flume.properties
```

第二步新开终端将k2的目录可写权限移除

```
$ sudo chmod -w  /Users/liyifeng/testflumek2
```

执行移除文件夹写权限步骤前后可以用命令查看两个文件夹下文本的行数来判断哪个Sink在工作中，如果Sink在工作，对应目录下的文本会以一秒一行的速度增加

```
$ wc -l /Users/liyifeng/testflumek1/*
$ wc -l /Users/liyifeng/testflumek2/*
```

Event序列化器

File Roll Sink 和 HDFS Sink 都使用过 *EventSerializer* 接口。下面介绍了随Flume一起提供的Event序列化器的详细信息。

消息体文本序列化器

它的别名是：text。 这个序列化器会把Event消息体里面的内容写到输出流同时不会对内容做任何的修改和转换。Event的header部分会被忽略掉，下面是配置参数：

属性	默认值	解释
appendNewline	true	是否在写入时将换行符附加到每个Event。由于遗留原因，默认值为true假定Event不包含换行符。

配置范例：

```
a1.sinks = k1
a1.sinks.k1.type = file_roll
a1.sinks.k1.channel = c1
a1.sinks.k1.sink.directory = /var/log/flume
a1.sinks.k1.sink.serializer = text
a1.sinks.k1.sink.serializer.appendNewline = false
```

Flume Event的Avro序列化器

别名： avro_event。

这个序列化器会把Event序列化成Avro的容器文件。使用的模式与 Avro RPC 机制中用于Flume Event的模式相同。

这个序列化器继承自 *AbstractAvroEventSerializer* 类。

配置参数：

属性	默认值	解释
syncIntervalBytes	2048000	Avro同步间隔，大约的字节数。
compressionCodec	null	指定 Avro压缩编码器。有关受支持的编码器，请参阅Avro的CodecFactory文档。

配置范例：

```
a1.sinks.k1.type = hdfs
a1.sinks.k1.channel = c1
a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M/%S
a1.sinks.k1.serializer = avro_event
a1.sinks.k1.serializer.compressionCodec = snappy
```

Avro序列化器

别名: 没有别名，只能配成全限定类名： org.apache.flume.sink.hdfs.AvroEventSerializer$Builder 。

这个序列化器跟上面的很像，不同的是这个可以配置记录使用的模式。 记录模式可以指定为Flume配置属性，也可以在Event头中传递。

为了能够配置记录的模式，使用下面 *schemaURL* 这个参数来配置。

如果要在Event头中传递记录模式，请指定包含模式的JSON格式表示的Event头 *flume.avro.schema.literal* 或包含可以找到模式的URL的 *flume.avro.schema.url* （hdfs:// 协议的URI是支持的）。 这个序列化器继承自 *AbstractAvroEventSerializer* 类。

配置参数：

属性	默认值	解释
syncIntervalBytes	2048000	Avro同步间隔，大约的字节数。
compressionCodec	null	指定 Avro压缩编码器。有关受支持的编码器，请参阅Avro的CodecFactory文档。
schemaURL	null	能够获取Avro模式的URL，如果header里面包含模式信息，优先级会高于这个参数的配置

配置范例：

```
a1.sinks.k1.type = hdfs
a1.sinks.k1.channel = c1
a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M/%S
a1.sinks.k1.serializer = org.apache.flume.sink.hdfs.AvroEventSerializer$Builder
a1.sinks.k1.serializer.compressionCodec = snappy
a1.sinks.k1.serializer.schemaURL = hdfs://namenode/path/to/schema.avsc
```

拦截器

Flume支持在运行时对Event进行修改或丢弃，可以通过拦截器来实现。Flume里面的拦截器是实现了 *org.apache.flume.interceptor.Interceptor* 接口的类。拦截器可以根据开发者的意图随意修改甚至丢弃Event， Flume也支持链式的拦截器执行方式，在配置文件里面配置多个拦截器就可以了。拦截器的顺序取决于它们被初始化的顺序（实际也就是配置的顺序），Event就

这样按照顺序经过每一个拦截器，如果想在拦截器里面丢弃Event， 在传递给下一级拦截器的list里面把它移除就行了。如果想丢弃所有的Event，返回一个空集合就行了。拦截器也是通过命名配置的组件，下面就是通过配置文件来创建拦截器的例子。

提示

Event在拦截器之间流动的时候是以集合的形式，并不是逐个Event传输的，这样就能理解上面所说的"从list里面移除"、"返回一个空集合"了。

做过Java web开发的同学应该很容易理解拦截器，Flume拦截器与spring MVC、struts2等框架里面的拦截器思路十分相似。

```
a1.sources = r1
a1.sinks = k1
a1.channels = c1
a1.sources.r1.interceptors = i1 i2
a1.sources.r1.interceptors.i1.type = org.apache.flume.interceptor.HostInterceptor$Builder
a1.sources.r1.interceptors.i1.preserveExisting = false
a1.sources.r1.interceptors.i1.hostHeader = hostname
a1.sources.r1.interceptors.i2.type = org.apache.flume.interceptor.TimestampInterceptor$Builder
a1.sinks.k1.filePrefix = FlumeData.%{CollectorHost}.%Y-%m-%d
a1.sinks.k1.channel = c1
```

拦截器构建器配置在type参数上。 拦截器是可配置的，就像其他可配置的组件一样。 在上面的示例中，Event首先传递给HostInterceptor，然后HostInterceptor返回的Event传递给TimestampInterceptor。 配置拦截器时你可以指定完全限定的类名（FQCN）或别名（timestamp）。 如果你有多个收集器写入相同的HDFS路径下，那么HostInterceptor是很有用的。

时间戳添加拦截器

这个拦截器会向每个Event的header中添加一个时间戳属性进去，key默认是"timestamp "（也可以通过headerName参数来自定义修改），value就是当前的毫秒值（其实就是用System.currentTimeMillis()方法得到的）。 如果Event已经存在同名的属性，可以选择是否保留原始的值。

属性	默认值	解释

属性	默认值	解释
type	–	组件类型，这个是： timestamp
headerName	timestamp	向Event header中添加时间戳键值对的key
preserveExisting	false	是否保留Event header中已经存在的同名（上面header设置的key，默认是timestamp）时间戳

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.channels =  c1
a1.sources.r1.type = seq
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type = timestamp
```

Host添加拦截器

这个拦截器会把当前Agent的hostname或者IP地址写入到Event的header中，key默认是“host”（也可以通过配置自定义key），value可以选择使用hostname或者IP地址。

属性	默认值	解释
type	–	组件类型，这个是： host
preserveExisting	false	如果header中已经存在同名的属性是否保留
useIP	true	true：使用IP地址；false：使用hostname
hostHeader	host	向Event header中添加host键值对的key

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type = host
```

静态属性写入拦截器

静态拦截器可以向Event header中写入一个固定的键值对属性。

这个拦截器目前不支持写入多个属性，但是你可以通过配置多个静态属性写入拦截器来实现。

属性	默认值	解释
type	–	组件类型，这个是： static
preserveExisting	true	如果header中已经存在同名的属性是否保留
key	key	写入header的key
value	value	写入header的值

配置范例：

```
a1.sources = r1
a1.channels = c1
a1.sources.r1.channels =  c1
a1.sources.r1.type = seq
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type = static
a1.sources.r1.interceptors.i1.key = datacenter
a1.sources.r1.interceptors.i1.value = NEW_YORK
```

删除属性拦截器

这个拦截器可以删除Event header里面的属性，可以是一个或多个。支持删除固定的header、固定分隔符分隔的多个header列表，也支持用正则表达式匹配的方式匹配删除。如果这三种方式都没有配置，那么这个拦截器不会对Event做任何修改处理。

如果只有一个header要删除，尽量使用withName方式，它要比另外两种在性能上要好一些。

属性	默认值	解释
type	–	组件类型，这个是： remove_header
withName	–	要删除的header属性名
fromList	–	要删除的header名列表，用下面 *fromListSeparator* 指定的分隔符分开
fromListSeparator	\s*,\s*	用来分隔 *fromList* 里面设置的header名的正则表达式，默认是由任意多个空白字符包围的逗号分隔
matching	–	要删除的header名的正则表达式，符合正则的将被全部删除

添加唯一ID拦截器

此拦截器在所有截获的Event上设置通用唯一标识符。 比如UUID可以是b5755073-77a9-43c1-8fad-b7a586f89757，它是一个128-bit的值。

Event如果没有可用的应用级唯一ID，就可以考虑使用添加唯一ID拦截器自动为Event分配UUID。 Event数据只要进入Flume网络中就给其分配一个UUID是非常重要的，Event进入Flume网络的第一个节点通常就是Flume的第一个source。 这样可以在Flume网络中进行复制和重

新传输以及Event的后续重复数据删除可以实现高可用性和高性能。 如果在应用层有唯一ID的话要比这种自动生成UUID要好一些，因为应用层分配的ID能方便我们在后续的数据存储中心对Event进行集中的更新和删除等操作。

属性	默认值	解释
type	–	组件类型，这个是：org.apache.flume.sink.solr.morphline.UUIDInterceptor$Builder
headerName	id	将要添加或者修改的id名称
preserveExisting	true	如果header中已经存在同名的属性是否保留
prefix	“”	UUID值的固定前缀（每个生成的uuid会在前面拼上这个固定前缀）

Morphline 实时清洗拦截器

此拦截器通过 morphline配置文件 过滤Event，配置文件定义了一系列转换命令，用于将记录从一个命令传递到另一个命令。 例如，morphline可以忽略某些Event或通过基于正则表达式的模式匹配来更改或插入某些Event header， 或者它可以通过Apache Tika在截获的Event上自动检测和设置MIME类型。 例如，这种数据包嗅探可用于Flume拓扑中基于内容的动态路由。 Morphline 实时清洗拦截器还可以帮助实现到多个Apache Solr集合的动态路由（例如，用于multi-tenancy）。

目前存在一个限制，这个拦截器不能输入一个Event然后产生多个Event出来，它不适用于重型的ETL处理，如果有需要，请考虑将ETL操作从Flume source转移到Flume sink中，比如：MorphlineSolrSink 。

必需的参数已用 **粗体** 标明。

属性	默认值	解释

属性	默认值	解释
type	–	组件类型，这个是： org.apache.flume.sink.solr.morphline.MorphlineInterceptor$Builder
morphlineFile	–	morphline配置文件在本地文件系统的绝对目录。比如：/etc/flume-ng/conf/morphline.conf
morphlineId	null	如果在morphline 配置文件里有多个morphline ，可以配置这个名字来加以区分

配置范例：

```
a1.sources.avroSrc.interceptors = morphlineinterceptor
a1.sources.avroSrc.interceptors.morphlineinterceptor.type = org.apache.flume.sink.solr.morphline.MorphlineInterceptor$
a1.sources.avroSrc.interceptors.morphlineinterceptor.morphlineFile = /etc/flume-ng/conf/morphline.conf
a1.sources.avroSrc.interceptors.morphlineinterceptor.morphlineId = morphline1
```

查找-替换拦截器

此拦截器基于Java正则表达式提供对Event消息体简单的基于字符串的搜索和替换功能。 还可以进行Backtracking / group。 此拦截器使用与Java Matcher.replaceAll()方法中的规则相同。

属性	默认值	解释
type	–	组件类型，这个是： search_replace
searchPattern	–	被替换的字符串的正则表达式
replaceString	–	上面正则找到的内容会使用这个字段进行替换
charset	UTF-8	Event body的字符编码，默认是：UTF-8

配置范例：

```
a1.sources.avroSrc.interceptors = search-replace
a1.sources.avroSrc.interceptors.search-replace.type = search_replace

<em># Remove leading alphanumeric characters in an event body.</em>
a1.sources.avroSrc.interceptors.search-replace.searchPattern = ^[A-Za-z0-9_]+
a1.sources.avroSrc.interceptors.search-replace.replaceString =
```

再来一个例子：

```
a1.sources.avroSrc.interceptors = search-replace
a1.sources.avroSrc.interceptors.search-replace.type = search_replace

<em># Use grouping operators to reorder and munge words on a line.</em>
a1.sources.avroSrc.interceptors.search-replace.searchPattern = The quick brown ([a-z]+) jumped over the lazy ([a-z]+)
a1.sources.avroSrc.interceptors.search-replace.replaceString = The hungry $2 ate the careless $1
```

正则过滤拦截器

这个拦截器会把Event的body当做字符串来处理，并用配置的正则表达式来匹配。可以配置指定被匹配到的Event丢弃还是没被匹配到的Event丢弃。

属性	默认值	解释
type	–	组件类型，这个是： regex_filter
regex	".*"	用于匹配Event内容的正则表达式
excludeEvents	false	如果为true，被正则匹配到的Event会被丢弃；如果为false，不被正则匹配到的Event会被丢弃

正则提取拦截器

这个拦截器会使用正则表达式从Event内容体中获取一组值并与配置的key组成n个键值对，然后放入Event的header中，Event的body不会有任何更改。它还支持插件化的方式配置序列化器来格式化从Event body中提取到的值。

属性	默认值	解释
type	–	组件类型，这个是： regex_extractor
regex	–	用于匹配Event内容的正则表达式
serializers	–	被正则匹配到的一组值被逐个添加到header中所使用的key的名字列表，多个用空格分隔 Flume提供了两个内置的序列化器，分别是： *org.apache.flume.interceptor.RegexExtractorInterceptorPassThroughSerializer org.apache.flume.interceptor.RegexExtractorInterceptorMillisSerializer*
serializers.<s1>.type	default	可选值： 1： default（default其实就是这个：*org.apache.flume.interceptor.RegexExtractorInterceptorPassThroughSerializer*）； 2： org.apache.flume.interceptor.RegexExtractorInterceptorMillisSerializer ； 3：自定义序列化器的全限定类名（自定义序列化器需要实现 *org.apache.flume.interceptor.RegexExtractorInterceptorSerializer* 接口）
serializers.<s1>.name	–	指定即将放入header的key，也就是最终写入到header中键值对的key
serializers.*	–	序列化器的一些属性

序列化器是用来格式化匹配到的那些字符串后再与配置的key组装成键值对放入header，默认情况下你只需要制定这些key就行了，Flume默认会使用 org.apache.flume.interceptor.RegexExtractorInterceptorPassThroughSerializer 这个序列化器，这个序列化器只是简单地将提取到的字符串与配置的key映射组装起来。当然也可以配置一个自定义的序列化器，以任意你需要的格式来格式化这些值。

例子 1:

假设Event body中包含这个字符串"1:2:3.4foobar5"

```
a1.sources.r1.interceptors.i1.regex = (\\d):(\\d):(\\d)
a1.sources.r1.interceptors.i1.serializers = s1 s2 s3
a1.sources.r1.interceptors.i1.serializers.s1.name = one
a1.sources.r1.interceptors.i1.serializers.s2.name = two
a1.sources.r1.interceptors.i1.serializers.s3.name = three
```

经过这个拦截器后，此时Event:

```
body: 不变 header增加3个属性: one=>1, two=>2, three=3
```

将上面的例子变动一下

```
a1.sources.r1.interceptors.i1.regex = (\\d):(\\d):(\\d)
a1.sources.r1.interceptors.i1.serializers = s1 s2
a1.sources.r1.interceptors.i1.serializers.s1.name = one
a1.sources.r1.interceptors.i1.serializers.s2.name = two
```

执行这个拦截器后，此时Event:

```
body: 不变 header增加3个属性: one=>1, two=>2
```

例子 2:

假设Event body中的某些行包含2012-10-18 18:47:57,614格式的时间戳，运行下面的拦截器

```
a1.sources.r1.interceptors.i1.regex = ^(?:\\n)?(\\d\\d\\d\\d-\\d\\d-\\d\\d\\s\\d\\d:\\d\\d)
a1.sources.r1.interceptors.i1.serializers = s1
a1.sources.r1.interceptors.i1.serializers.s1.type = org.apache.flume.interceptor.RegexExtractorInterceptorMillisSerial
a1.sources.r1.interceptors.i1.serializers.s1.name = timestamp
a1.sources.r1.interceptors.i1.serializers.s1.pattern = yyyy-MM-dd HH:mm
```

运行拦截器后，此时Event：

```
body不变，header中增加一个新属性：timestamp=>1350611220000
```

自动重载配置

属性	默认值	解释
flume.called.from.service	–	如果设定了这个参数，Agent启动时会轮询地寻找配置文件，即使在预期的位置没有找到配置文件。 如果没有设定这个参数，如果flume Agent在预期的位置没有找到配置文件的话会立即停止。设定这个参数使用的时候无需设定具体的值， 像这样： -Dflume.called.from.service 就可以了。

Property: flume.called.from.service

Flume每隔30秒轮询扫描一次指定配置文件的变动。如果首次扫描现有文件或者上次轮询时的文件「修改时间」发生了变动，Flume Agent就会重新加载新的配置内容。重命名或者移动文件不会更改配置文件的「修改时间」。 当Flume轮询一个不存在的配置文件时，有以下两种情况：

1. 当第一次轮询就不存在配置文件时，会根据flume.called.from.service的属性执行操作。如果设定了这个属性，则继续轮询（固定的时间间隔，30秒轮询一次）；如果未设置这个属性，则Agent会立即终止。
2. 当轮询到一个不存在的配置文件并且不是第一次轮询（也就是说之前轮询的时候有配置文件，但是现在中途没有了），Agent会继续轮询不会停止运行。

配置文件过滤器

> *提示*
>
> *本小节是flume1.9新增，英文名称叫Configuration Filters，感觉翻译成【配置过滤器】不太直观，其实它就是个动态替换配置文件中的占位符，类似于Maven的profile，Spring等各种java框架里面到处都是这种用法。 可以认为这个新特性就是之前 在配置文件里面自定义环境变量 的加强版， 下面这个配置模板看起来很绕，直接看后面例子就容易理解多了。总共有三种用法，第一种是把要替换的内容放在环境变量中（这与之前完全一样）， 第二种是通过执行外部脚本或者命令来动态取值，第三种是将敏感内容存储在Hadoop CredentialProvider。*

Flume提供了一个动态加载配置的功能，用来把那些敏感的数据（比如密码）、或者需要动态获取的信息加载到配置文件中，编写配置文件的时候用类似于EL表达式的${key}占位即可。

用法

具体使用的格式跟EL表达式很像，但是它现在仅仅是像，并不是一个完整的EL表达式解析器。

```
<agent_name>.configfilters = <filter_name>
<agent_name>.configfilters.<filter_name>.type = <filter_type>

<agent_name>.sources.<source_name>.parameter = ${<filter_name>['<key_for_sensitive_or_generated_data>']}
<agent_name>.sinks.<sink_name>.parameter = ${<filter_name>['<key_for_sensitive_or_generated_data>']}
<agent_name>.<component_type>.<component_name>.parameter = ${<filter_name>['<key_for_sensitive_or_generated_data>']}
<em>#or</em>
<agent_name>.<component_type>.<component_name>.parameter = ${<filter_name>["<key_for_sensitive_or_generated_data>"]}
<em>#or</em>
<agent_name>.<component_type>.<component_name>.parameter = ${<filter_name>[<key_for_sensitive_or_generated_data>]}
<em>#or</em>
<agent_name>.<component_type>.<component_name>.parameter = some_constant_data${<filter_name>[<key_for_sensitive_or_gen
```

配到环境变量

属性 默认值		解释
type	–	组件类型，这里只能填 env

例子1

这是一个在配置文件中隐藏密码的例子，密码配置在了环境变量中。

```
a1.sources = r1
a1.channels = c1
a1.configfilters = f1             # 这里给配置加载器命名为f1

a1.configfilters.f1.type = env    # 将配置加载器f1的类型设置为env，表示从环境变量读取参数

a1.sources.r1.channels =  c1
a1.sources.r1.type = http
a1.sources.r1.keystorePassword = ${f1['my_keystore_password']} # 启动Flume时如果配置了my_keystore_password=Secret123，这里
```

这里 a1.sources.r1.keystorePassword 的值就会从环境变量里面获取了，在环境变量里面配置这个my_keystore_password的一种方法就是配置在启动命令前，像下面这样：

```
$ my_keystore_password=Secret123 bin/flume-ng agent --conf conf --conf-file example.conf ...
```

从外部命令获取

属性	默认值	解释
type	–	组件类型，这里只能填 external

属性	默认值	解释
command	–	将要执行的用于获取键值的命令或脚本。这个命令会以这种命令格式调用 `<command> <key>` ，它期望的返回结果是个单行数值并且该脚本最后exit 0。
charset	UTF-8	返回字符串的编码

例子2

这又是一个在配置文件中隐藏密码的例子，这次密码放在了外部脚本中。

```
a1.sources = r1
a1.channels = c1
a1.configfilters = f1

a1.configfilters.f1.type = external
a1.configfilters.f1.command = /usr/bin/passwordResolver.sh    # 外部脚本的绝对路径
a1.configfilters.f1.charset = UTF-8

a1.sources.r1.channels =  c1
a1.sources.r1.type = http
a1.sources.r1.keystorePassword = ${f1['my_keystore_password']} # 用这种类似于EL表达式取值
```

在这个例子里面，flume实际执行的是下面这个命令来取值

```
$ /usr/bin/passwordResolver.sh my_keystore_password
```

这个脚本 `passwordResolver.sh` return了一个密码，假设是 `Secret123` 并且exit code 是0。

例子3

这个例子是通过外部脚本动态生成本地存储event的文件夹路径。

> *提示*
>
> *与上一个例子的使用的配置方式完全相同，其实就是一个配置方式的两种实际应用。这个例子用到了前面的 File Roll Sink ，这个sink会把event全部存储在本地文件系统中，而本地存储的目录生成规则使用了这个新特性。*

```
a1.sources = r1
a1.channels = c1
a1.configfilters = f1

a1.configfilters.f1.type = external
a1.configfilters.f1.command = /usr/bin/generateUniqId.sh
a1.configfilters.f1.charset = UTF-8

a1.sinks = k1
a1.sinks.k1.type = file_roll
a1.sinks.k1.channel = c1
a1.sinks.k1.sink.directory = /var/log/flume/agent_${f1['agent_name']} # will be /var/log/flume/agent_1234
```

同上一个例子一样，flume实际执行的是下面这个命令来取值

```
$ /usr/bin/generateUniqId.sh agent_name
```

这个脚本 generateUniqId.sh return了一个值，假设是 1234 并且exit code 是0。

使用Hadoop CredentialProvider存储配置信息

使用这种配置方法需要将2.6版本以上的hadoop-common库放到classpath中，如果已经安装了hadoop就不必了，agent会自动把它加到classpath。

属性 默认值		解释
type	–	组件类型，这里只能填 hadoop

属性 默认值		解释
credential.provider.path	–	provider的路径，参考hadoop的文档：https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/CredentialProviderAPI.html#Configuring_the_Provider_Path
credstore.java-keystore-provider.password-file	–	存储CredentialProvider密码的文件名。这个文件必须在classpath下，CredentialProvider的密码可以通过HADOOP_CREDSTORE_PASSWORD环境变量来指定。

例子

通过Hadoop CredentialProvider来实现flume配置文件中隐藏密码

```
a1.sources = r1
a1.channels = c1
a1.configfilters = f1

a1.configfilters.f1.type = hadoop
a1.configfilters.f1.credential.provider.path = jceks://file/<path_to_jceks file>

a1.sources.r1.channels =  c1
a1.sources.r1.type = http
a1.sources.r1.keystorePassword = ${f1['my_keystore_password']} #从hadoop credential获取密码的内容
```

Log4J Appender直接写到Flume

使用log4j Appender输出日志到Flume Agent 的avro source上。使用的时候log4j客户端必须要在classpath引入flume-ng-sdk（比如：flume-ng-sdk-1.9.0.jar）. 必需的参数已用 **粗体** 标明。

> *提示*
>
> *说白了就是用log4j直接将日志内容发送到Flume，省去了一般先写入到日志再由Flume收集的过程。*

属性	**默认值**	**解释**
Hostname	–	远程运行着avro source的Flume Agent的hostname
Port	–	上面这个Flume Agent 的avro source监听的端口
UnsafeMode	false	如果为true，log4j的appender在发送Event失败时不会抛出异常
AvroReflectionEnabled	false	是否使用Avro反射来序列化log4j的Event（当log是字符串时不要开启）
AvroSchemaUrl	–	一个能检索到Avro结构的url

log4j.properties 文件配置范例：

```
<em>#...</em>
log4j.appender.flume = org.apache.flume.clients.log4jappender.Log4jAppender
log4j.appender.flume.Hostname = example.com
log4j.appender.flume.Port = 41414
log4j.appender.flume.UnsafeMode = true

<em># 指定一个类的logger输出到Flume appender上</em>
log4j.logger.org.example.MyClass = DEBUG,flume
<em>#...</em>
```

默认情况下，每一个Event都会通过调用 toString()方法或者log4j的layout（如果配置了的话）转换成一个字符串。

如果Event是 *org.apache.avro.generic.GenericRecord* 或 *org.apache.avro.specific.SpecificRecord* 的一个实例，又或者 *AvroReflectionEnabled* 属性设置为 `true` ，Event会使用Avro序列化器来序列化。

使用Avro序列化每个Event效率很低，因此最好提供一个avro schema的URL，可以被 downstream sink（通常是HDFS sink）从该URL检索schema。 如果未指定 *AvroSchemaUrl*，则schema将作为Flume header包含在内。

使用Avro序列化Event的log4j.properties配置范例：

```
<em>#...</em>
log4j.appender.flume = org.apache.flume.clients.log4jappender.Log4jAppender
log4j.appender.flume.Hostname = example.com
log4j.appender.flume.Port = 41414
log4j.appender.flume.AvroReflectionEnabled = true
log4j.appender.flume.AvroSchemaUrl = hdfs://namenode/path/to/schema.avsc

<em># 指定一个类的logger输出到flume appender上</em>
log4j.logger.org.example.MyClass = DEBUG,flume
<em>#...</em>
```

负载均衡的Log4J Appender

使用log4j Appender发送Event到多个运行着Avro Source的Flume Agent上。使用的时候log4j客户端必须要在classpath引入flume-ng-sdk（比如：flume-ng-sdk-1.9.0.jar）。 这个appender支持轮询和随机的负载方式，它也支持配置一个退避时间，以便临时移除那些挂掉的Flume Agent。 必需的参数已用 **粗体** 标明。

> *提示*
>
> *这是上面Log4j Appender的升级版，支持多个Flume实例的负载均衡发送，配置也很类似。*

属性	默认值	解释
Hosts	–	host:port格式的Flume Agent（运行着Avro Source）地址列表，多个用空格分隔

属性	默认值	解释
Selector	ROUND_ROBIN	appender向Flume Agent发送Event的选择机制。可选值有：ROUND_ROBIN （轮询）、 RANDOM （随机） 或者自定义选择器的全限定类名（自定义选择器必须继承自 *LoadBalancingSelector*）
MaxBackoff	–	一个long型数值，表示负载平衡客户端将无法发送Event的节点退避的最长时间（毫秒）。 默认不启用规避机制
UnsafeMode	false	如果为true，log4j的appender在发送Event失败时不会抛出异常
AvroReflectionEnabled	false	是否使用Avro反射来序列化log4j的Event（当log是字符串时不要开启）
AvroSchemaUrl	–	一个能检索到Avro结构的url

log4j.properties 文件配置范例：

```
<em>#...</em>
log4j.appender.out2 = org.apache.flume.clients.log4jappender.LoadBalancingLog4jAppender
log4j.appender.out2.Hosts = localhost:25430 localhost:25431

<em># configure a class's logger to output to the flume appender</em>
log4j.logger.org.example.MyClass = DEBUG,flume
<em>#...</em>
```

使用随机（ RANDOM ）负载均衡方式的log4j.properties 文件配置范例：

```
<em>#...</em>
log4j.appender.out2 = org.apache.flume.clients.log4jappender.LoadBalancingLog4jAppender
log4j.appender.out2.Hosts = localhost:25430 localhost:25431
log4j.appender.out2.Selector = RANDOM

<em># configure a class's logger to output to the flume appender</em>
log4j.logger.org.example.MyClass = DEBUG,flume
```

```
<em>#...</em>
```

log4j使用「失败退避」方式的log4j.properties配置范例：

```
<em>#...</em>
log4j.appender.out2 = org.apache.flume.clients.log4jappender.LoadBalancingLog4jAppender
log4j.appender.out2.Hosts = localhost:25430 localhost:25431 localhost:25432
log4j.appender.out2.Selector = ROUND_ROBIN
log4j.appender.out2.MaxBackoff = 30000   #最大的退避时长是30秒

<em># configure a class's logger to output to the flume appender</em>
log4j.logger.org.example.MyClass = DEBUG,flume
<em>#...</em>
```

> *提示*
>
> *这种退避机制在其他组件中有过多次应用，比如：Spooling Directory Source 中的* maxBackoff *属性的功能是一样的。*

安全

HDFS Sink 、 HBaseSinks 、 Thrift Source 、 Thrift Sink 和 Kite Dataset Sink 都支持Kerberos认证。 请参考对应组件的文档来配置Kerberos认证的选项。

Flume Agent 会作为一个主体向kerberos KDC认证，给需要kerberos认证的所有组件使用。 HDFS Sink 、 HBaseSinks 、 Thrift Source 、 Thrift Sink 和 Kite Dataset Sink 配置的主体和keytab 文件应该是相同的，否则组件无法启动。

监控

Flume的监控系统的完善仍在进行中，变化可能会比较频繁，有几个Flume组件会向JMX平台MBean服务器报告运行指标。 可以使用Jconsole查询这些指标数据。

当前可用的组件监控指标

下面表中列出了一些组件支持的监控数据，‘x’代表支持该指标，空白的表示不支持。各个指标的具体含义可参阅源码。

提示

Source 1 和Source 2 表格是的指标是一样的，分成两个表是因为全放在一个表里一屏显示不下。Sink1和Sink2同理。

Sources 1

	Avro	Exec	HTTP	JMS	Kafka	MultiportSyslogTCP	Scribe
AppendAcceptedCount	x						
AppendBatchAcceptedCount	x		x	x			
AppendBatchReceivedCount	x		x	x			
AppendReceivedCount	x						
ChannelWriteFail	x		x	x	x	x	x
EventAcceptedCount	x	x	x	x	x	x	x
EventReadFail			x	x	x	x	x
EventReceivedCount	x	x	x	x	x	x	x

GenericProcessingFail			x			x	
KafkaCommitTimer					x		
KafkaEmptyCount					x		
KafkaEventGetTimer					x		
OpenConnectionCount	x						

Sources 2

	SequenceGenerator	SpoolDirectory	SyslogTcp	SyslogUDP	Taildir	Thrift
AppendAcceptedCount						x
AppendBatchAcceptedCount	x	x			x	x
AppendBatchReceivedCount		x			x	x
AppendReceivedCount						x
ChannelWriteFail	x	x	x	x	x	x
EventAcceptedCount	x	x	x	x	x	x
EventReadFail		x	x	x	x	
EventReceivedCount		x	x	x	x	x
GenericProcessingFail		x			x	

KafkaCommitTimer						
KafkaEmptyCount						
KafkaEventGetTimer						
OpenConnectionCount						

Sinks 1

	Avro/Thrift	AsyncHBase	ElasticSearch	HBase	HBase2
BatchCompleteCount	x	x	x	x	x
BatchEmptyCount	x	x	x	x	x
BatchUnderflowCount	x	x	x	x	x
ChannelReadFail	x				x
ConnectionClosedCount	x	x	x	x	x
ConnectionCreatedCount	x	x	x	x	x
ConnectionFailedCount	x	x	x	x	x
EventDrainAttemptCount	x	x	x	x	x
EventDrainSuccessCount	x	x	x	x	x
EventWriteFail	x				x
KafkaEventSendTimer					
RollbackCount					

Sinks 2

	HDFSEvent	Hive	Http	Kafka	Morphline	RollingFile
BatchCompleteCount	x	x			x	
BatchEmptyCount	x	x		x	x	
BatchUnderflowCount	x	x		x	x	
ChannelReadFail	x	x	x	x	x	x
ConnectionClosedCount	x	x				x
ConnectionCreatedCount	x	x				x
ConnectionFailedCount	x	x				x
EventDrainAttemptCount	x	x	x		x	x
EventDrainSuccessCount	x	x	x	x	x	x
EventWriteFail	x	x	x	x	x	x
KafkaEventSendTimer				x		
RollbackCount				x		

Channels

	File	Kafka	Memory	PseudoTxnMemory	SpillableMemory

ChannelCapacity	x		x		x
ChannelSize	x		x	x	x
CheckpointBackupWriteErrorCount	x				
CheckpointWriteErrorCount	x				
EventPutAttemptCount	x	x	x	x	x
EventPutErrorCount	x				
EventPutSuccessCount	x	x	x	x	x
EventTakeAttemptCount	x	x	x	x	x
EventTakeErrorCount	x				
EventTakeSuccessCount	x	x	x	x	x
KafkaCommitTimer		x			
KafkaEventGetTimer		x			
KafkaEventSendTimer		x			
Open	x				
RollbackCounter		x			
Unhealthy	x				

JMX Reporting

MX监控可以通过在flume-env.sh脚本中修改JAVA_OPTS环境变量中的JMX参数来开启，比如这样：

```
export JAVA_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=5445 -Dcom.sun.management.jmxremot
```

> *警告*
>
> *注意：上面的JVM启动参数例子里面没有开启安全验证，如果要开启请参考：http://docs.oracle.com/javase/6/docs/technotes/guides/management/agent.html*

Ganglia Reporting

Flume也可以向Ganglia 3或Ganglia 3.1报告运行指标数据。想要开启这个功能，必须在Agent启动时候指定。Flume Agent在启动时候必须制定下面这些参数并在参数前面加上前缀「flume.monitoring.」来配置，也可以在flume-env.sh中设定这些参数。

属性	默认值	解释
type	–	组件类型，这个是： ganglia
hosts	–	hostname:port 格式的 Ganglia 服务列表，多个用逗号分隔
pollFrequency	60	向Ganglia服务器报告数据的时间间隔（秒）
isGanglia3	false	设置为true后Ganglia的版本兼容为Ganglia3，默认情况下Flume发送的数据是Ganglia3.1格式的

我们可以在启动时这样开启Ganglia支持：

```
$ bin/flume-ng agent --conf-file example.conf --name a1 -Dflume.monitoring.type=ganglia -Dflume.monitoring.hosts=com.ex
```

提示

看上面这个启动脚本，其中 *-Dflume.monitoring.type=ganglia* 以及后面的参数都是按照上面描述的规则配置的，就是「固定的前缀+参数=参数值」的形式。

JSON Reporting

Flume也支持以JSON格式报告运行指标。为了对外提供这些报告数据，Flume会在某个端口（可自定义）上运行一个web服务来提供这些数据，以下面这种格式：

```
{
"typeName1.componentName1" : {"metric1" : "metricValue1", "metric2" : "metricValue2"},
"typeName2.componentName2" : {"metric3" : "metricValue3", "metric4" : "metricValue4"}
}
```

下面是一个具体的报告例子：

```
{
"CHANNEL.fileChannel":{"EventPutSuccessCount":"468085",
                       "Type":"CHANNEL",
                       "StopTime":"0",
                       "EventPutAttemptCount":"468086",
                       "ChannelSize":"233428",
                       "StartTime":"1344882233070",
                       "EventTakeSuccessCount":"458200",
                       "ChannelCapacity":"600000",
                       "EventTakeAttemptCount":"458288"},
"CHANNEL.memChannel":{"EventPutSuccessCount":"22948908",
                      "Type":"CHANNEL",
                      "StopTime":"0",
                      "EventPutAttemptCount":"22948908",
                      "ChannelSize":"5",
                      "StartTime":"1344882209413",
                      "EventTakeSuccessCount":"22948900",
                      "ChannelCapacity":"100",
                      "EventTakeAttemptCount":"22948908"}
}
```

属性	默认值	解释
type	–	组件类型，这个是： http
port	41414	查看json报告的端口

启用JSON报告的启动脚本示例：

```
$ bin/flume-ng agent --conf-file example.conf --name a1 -Dflume.monitoring.type=http -Dflume.monitoring.port=34545
```

启动后可以通过这个地址 http
😔
/<hostname>:<port>/metrics 来查看报告，自定义组件可以报告上面Ganglia部分中提到的指标数据。

Custom Reporting

可以通过编写自己的执行报告服务向其他系统报告运行指标。 报告类必须实现org.apache.flume.instrumentation.MonitorService 接口。 自定义的报告类与GangliaServer的报告用法相同。 他们可以轮询请求mbean服务器获取mbeans的运行指标。 例如，假设一个命名为为HTTPReporting的HTTP监视服务，启动脚本如下所示：

```
$ bin/flume-ng agent --conf-file example.conf --name a1 -Dflume.monitoring.type=com.example.reporting.HTTPReporting -Df
```

属性	默认值	解释
type	–	自定义报告组件的全限定类名

Reporting metrics from custom components

自定义Flume监控组件必须应继承自 *org.apache.flume.instrumentation.MonitoredCounterGroup* 类。 然后， 该类应为其公开的每个度量指标提供getter方法。 请参阅下面的代码。 MonitoredCounterGroup 需要一个此类要提供的监控属性列表。 目前仅支持将监控指标值设置为long型。

```
<strong>public</strong> <strong>class</strong> <strong>SinkCounter</strong> <strong>extends</strong> MonitoredCounterG
    SinkCounterMBean {

  <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_CONNECTION_CREATED =
    "sink.connection.creation.count";

  <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_CONNECTION_CLOSED =
    "sink.connection.closed.count";

  <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_CONNECTION_FAILED =
    "sink.connection.failed.count";

  <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_BATCH_EMPTY =
    "sink.batch.empty";

  <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_BATCH_UNDERFLOW =
      "sink.batch.underflow";

  <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_BATCH_COMPLETE =
    "sink.batch.complete";

  <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_EVENT_DRAIN_ATTEMPT =
    "sink.event.drain.attempt";

  <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_EVENT_DRAIN_SUCCESS =
    "sink.event.drain.sucess";

  <strong>private</strong> <strong>static</strong> <strong>final</strong> String[] ATTRIBUTES = {
    COUNTER_CONNECTION_CREATED, COUNTER_CONNECTION_CLOSED,
    COUNTER_CONNECTION_FAILED, COUNTER_BATCH_EMPTY,
    COUNTER_BATCH_UNDERFLOW, COUNTER_BATCH_COMPLETE,
    COUNTER_EVENT_DRAIN_ATTEMPT, COUNTER_EVENT_DRAIN_SUCCESS
  };


  <strong>public</strong> SinkCounter(String name) {
    <strong>super</strong>(MonitoredCounterGroup.Type.SINK, name, ATTRIBUTES);
  }

  <strong>@Override</strong>
  <strong>public</strong> long getConnectionCreatedCount() {
    <strong>return</strong> get(COUNTER_CONNECTION_CREATED);
  }

  <strong>public</strong> long incrementConnectionCreatedCount() {
    <strong>return</strong> increment(COUNTER_CONNECTION_CREATED);
  }

}
```

工具

文件channel验证工具

文件channel完整性校验工具可验证文件channel中各个Event的完整性，并删除损坏的Event。

这个工具可以通过下面这种方式开启：

```
$bin/flume-ng tool --conf ./conf FCINTEGRITYTOOL -l ./datadir
```

datadir 是即将被校验的用逗号分隔的目录列表。

以下是可选的参数

选项	解释
h/help	显示帮助信息
l/dataDirs	校验工具会校验的目录列表，多个用逗号分隔

Event校验工具

Event验证器工具可用于按照预定好的逻辑验证文件channel中的Event。 该工具会在每个Event上执行用户自定义的验证逻辑，并删除不符合校验逻辑的Event。

提示

简单说就是一个自定义的Event校验器，只能用于验证文件channel中的Event。实现的方式就是实现 EventValidator 接口，没有被校验通过的Event会被丢弃。

多bb一句：目前还没想到这个工具有哪些用途，感觉可以用自定义拦截器来实现这种功能，说起拦截器又很奇怪在拦截器章节中居然没有介绍自定义拦截器。

这个工具可以通过下面这种方式开启：

```
$bin/flume-ng tool --conf ./conf FCINTEGRITYTOOL -l ./datadir -e org.apache.flume.MyEventValidator -DmaxSize 2000
```

datadir 是即将被校验的用逗号分隔的目录列表。

以下是可选的参数

选项	解释
h/help	显示帮助信息
l/dataDirs	校验工具会校验的目录列表，多个用逗号分隔
e/eventValidator	自定义验证工具类的全限定类名，这个类的jar包必须在Flume的classpath中

自定义的Event验证器必须实现 *EventValidator* 接口，建议不要抛出任何异常。其他参数可以通过-D选项传递给EventValitor实现。

让我们看一个基于简单的Event大小验证器的示例，它将拒绝大于指定的最大size的Event。

```
<strong>public</strong> <strong>static</strong> <strong>class</strong> <strong>MyEventValidator</strong> <strong>imple
  <strong>private</strong> int value = 0;
```

```
  <strong>private</strong> MyEventValidator(int val) {
    value = val;
  }

  <strong>@Override</strong>
  <strong>public</strong> boolean validateEvent(Event event) {
    <strong>return</strong> event.getBody() <= value;
  }

  <strong>public</strong> <strong>static</strong> <strong>class</strong> <strong>Builder</strong> <strong>implements</

    <strong>private</strong> int sizeValidator = 0;

    <strong>@Override</strong>
    <strong>public</strong> EventValidator build() {
      <strong>return</strong> <strong>new</strong> DummyEventVerifier(sizeValidator);
    }

    <strong>@Override</strong>
    <strong>public</strong> void configure(Context context) {
      binaryValidator = context.getInteger("maxSize");
    }
  }
}
```

拓扑设计注意事项

Flume非常灵活，可以支持大量的部署方案。如果你打算在大型生产部署中使用Flume，建议你花些时间来思考如何拓扑Flume来解决你的问题。本小节会介绍一些注意事项。

Flume真的适合你吗?

如果你需要将文本日志数据提取到Hadoop / HDFS中，那么Flume最合适不过了。但是，对于其他情况，你最好看看以下建议：

Flume旨在通过相对稳定，可能复杂的拓扑部署来传输和收集定期生成的Event数据。“Event数据”定义非常广泛，对于Flume来说一个Event就是一个普通的字节数组而已。 Event大小有一些限制，它不能比你的内存或者服务器硬盘还大，实际使用中Flume Event可以是文本或图片的任何文件。关键的是这些Event应该是以连续的流的方式不断生成的。 如果你的数据不是定期生成的（比如你将大量的数据批量加载到Hadoop集群中），虽然Flume可以做这个事情，但是有点“杀鸡用牛刀”的感觉，这并不是Flume所擅长和喜欢的工作方式。 Flume喜欢相对稳定的拓扑结构，但也不是说永远一成不变，Flume可以处理拓扑中的更改而又不丢失数据，还可以处理由于故障转移或者配置的定期重新加载。如果你的拓扑结构每天都会变动， 那么Flume可能就无法正常的工作了，毕竟重新配置也是需要一定思考和开销的。

> *提示*
>
> *可以这样理解，Flume就像一个高速公路收费站，适合干那种例行性、重复的工作，别今天还全是人工收费出口，明天加一个ETC出口，后天就全改成ETC出口， 大后天又改回大部分人口收费出口，计费又不准导致大家有ETC也不敢用，整个系统的吞吐能力不升反降，这样的情况就别用Flume了[doge]， 不要总是变来变去，虽然Flume具备一些“随机应变”能力，但是也别太频繁了。*

Flume中数据流的可靠性

Flume 流的可靠性取决于几个因素，通过调整这几个因素，你可以自定这些Flume的可靠性选项。

1. **使用什么类型的channel。** Flume有持久型的channel（将数据保存在磁盘上的channel）和非持久化的channel（如果机器故障就会丢失数据的channel）。持久化的channel使用基于磁盘的存储，存储在这类channel中的数据不受机器重启或其他非磁盘故障影响。
2. **channel是否能充分满足工作负载。** channel在Flume中扮演了数据流传输的缓冲区，这些缓冲区都有固定容量，一旦channel被占满后，就会将压力传播到数据流的前面节点上。如果压力传播到了source节点上，此时Flume将变得不可用并且可能丢失数据。
3. **是否使用冗余拓扑。** 冗余的拓扑可以复制数据流做备份。这样就提供了一个容错机制，并且可以克服磁盘或者机器故障。

在设计一个可靠的Flume拓扑时最好的办法就是把各种故障和故障导致的结果都提前想到。如果磁盘发生故障会怎么样？如果机器出现故障会怎么样？如果你的末端sink（比如HDFS sink）挂掉一段时间遭到背压怎么办？拓扑的设计方案多种多样，但是能想到的常见问题也就这么多。

Flume拓扑设计

拓扑设计第一步就是要确定要使用的 source 和 sink（在数据流中最末端的sink节点）。这些确定了你Flume拓扑集群的边缘。下一个要考虑的因素是是否引入中间的聚合层和Event路由节点。 如果要从大量的source中收集数据，则聚合数据以简化末端Sink的收集挺有帮助的。 聚合层还可以充当缓冲区来缓解突发的 source 流量 和 sink 的不可用情况。如果你想路由不同位置间的数据，你可能还希望在一些点来分割流：这样就会创建本身包含聚合点的子拓扑。

计算Flume部署所需要的节点

一旦你对自己如何拓扑部署Flume集群节点有了大致的方案，下一个问题就是需要多少硬件和网络流量。首先量化你会产生多少要收集的数据，这个不太好计算，因为大多数情况下数据都是突发性的（比如由于昼夜交换）并且可能还不太好预测。 我们可以先确定每个拓扑层的最大吞吐量，包括每秒Event数、每秒字节数，一旦确定了某一层的所需吞吐量，就可以计算这一层所需的最小节点数。要确定可达到的吞吐量，最好使用合成或采样Event数据在你的硬件上测试Flume。通常情况下， 文件channel能达到10MB/s的速率，内存channel应该能达到100MB/s或更高的速率，不过硬件和操作系统不同，性能指标也会有一些差异。

计算聚合吞吐量可以确定每层所需最小节点数，需要几个额外的节点，比如增加冗余和处理突发的流量。

故障排除

处理Agent失败

如果Flume的Agent挂掉，则该Agent上托管的所有流都将中止。 重新启动Agent后，这些流将恢复。 使用文件channel或其他可靠channel的流将从中断处继续处理Event。 如果无法在同一硬件上重新启动Agent，则可以选择将数据库迁移到另一个硬件并设置新的Flume Agent， 该Agent可以继续处理db中保存的Event。 利用数据库的高可用特性将Flume Agent转移到另一个主机。

兼容性

HDFS

Flume目前支持HDFS 0.20.2 和 0.23版本。

AVRO

待完善。（不是没翻译，是原文档上就没有）

Additional version requirements

待完善。（不是没翻译，是原文档上就没有）

Tracing

待完善。（不是没翻译，是原文档上就没有）

More Sample Configs

待完善。（不是没翻译，是原文档上就没有）

内置组件

提示

基本上你能想到的常见的数据来源（source）与目的地（sink）Flume都帮我们实现了，下表是Flume自带的一些组件和它们的别名，这个别名在实际使用的时候非常方便。看一遍差不多也就记住了，记不住也没关系，知道大概有哪些就行了。

这些别名不区分大小写。

组件接口	别名	实现类
org.apache.flume.Channel	memory	org.apache.flume.channel.MemoryChannel
org.apache.flume.Channel	jdbc	org.apache.flume.channel.jdbc.JdbcChannel
org.apache.flume.Channel	file	org.apache.flume.channel.file.FileChannel
org.apache.flume.Channel	–	org.apache.flume.channel.PseudoTxnMemoryChannel
org.apache.flume.Channel	–	org.example.MyChannel
org.apache.flume.Source	avro	org.apache.flume.source.AvroSource
org.apache.flume.Source	netcat	org.apache.flume.source.NetcatSource
org.apache.flume.Source	seq	org.apache.flume.source.SequenceGeneratorSource
org.apache.flume.Source	exec	org.apache.flume.source.ExecSource

组件接口	别名	实现类
org.apache.flume.Source	syslogtcp	org.apache.flume.source.SyslogTcpSource
org.apache.flume.Source	multiport_syslogtcp	org.apache.flume.source.MultiportSyslogTCPSource
org.apache.flume.Source	syslogudp	org.apache.flume.source.SyslogUDPSource
org.apache.flume.Source	spooldir	org.apache.flume.source.SpoolDirectorySource
org.apache.flume.Source	http	org.apache.flume.source.http.HTTPSource
org.apache.flume.Source	thrift	org.apache.flume.source.ThriftSource
org.apache.flume.Source	jms	org.apache.flume.source.jms.JMSSource
org.apache.flume.Source	–	org.apache.flume.source.avroLegacy.AvroLegacySource
org.apache.flume.Source	–	org.apache.flume.source.thriftLegacy.ThriftLegacySource
org.apache.flume.Source	–	org.example.MySource
org.apache.flume.Sink	null	org.apache.flume.sink.NullSink
org.apache.flume.Sink	logger	org.apache.flume.sink.LoggerSink
org.apache.flume.Sink	avro	org.apache.flume.sink.AvroSink
org.apache.flume.Sink	hdfs	org.apache.flume.sink.hdfs.HDFSEventSink
org.apache.flume.Sink	hbase	org.apache.flume.sink.hbase.HBaseSink
org.apache.flume.Sink	hbase2	org.apache.flume.sink.hbase2.HBase2Sink
org.apache.flume.Sink	asynchbase	org.apache.flume.sink.hbase.AsyncHBaseSink
org.apache.flume.Sink	elasticsearch	org.apache.flume.sink.elasticsearch.ElasticSearchSink
org.apache.flume.Sink	file_roll	org.apache.flume.sink.RollingFileSink
org.apache.flume.Sink	irc	org.apache.flume.sink.irc.IRCSink
org.apache.flume.Sink	thrift	org.apache.flume.sink.ThriftSink
org.apache.flume.Sink	–	org.example.MySink

组件接口	别名	实现类
org.apache.flume.ChannelSelector	replicating	org.apache.flume.channel.ReplicatingChannelSelector
org.apache.flume.ChannelSelector	multiplexing	org.apache.flume.channel.MultiplexingChannelSelector
org.apache.flume.ChannelSelector	–	org.example.MyChannelSelector
org.apache.flume.SinkProcessor	default	org.apache.flume.sink.DefaultSinkProcessor
org.apache.flume.SinkProcessor	failover	org.apache.flume.sink.FailoverSinkProcessor
org.apache.flume.SinkProcessor	load_balance	org.apache.flume.sink.LoadBalancingSinkProcessor
org.apache.flume.SinkProcessor	–	
org.apache.flume.interceptor.Interceptor	timestamp	org.apache.flume.interceptor.TimestampInterceptor$Builder
org.apache.flume.interceptor.Interceptor	host	org.apache.flume.interceptor.HostInterceptor$Builder
org.apache.flume.interceptor.Interceptor	static	org.apache.flume.interceptor.StaticInterceptor$Builder
org.apache.flume.interceptor.Interceptor	regex_filter	org.apache.flume.interceptor.RegexFilteringInterceptor$Builder
org.apache.flume.interceptor.Interceptor	regex_extractor	org.apache.flume.interceptor.RegexFilteringInterceptor$Builder
org.apache.flume.channel.file.encryption.KeyProvider$Builder	jceksfile	org.apache.flume.channel.file.encryption.JCEFileKeyProvider
org.apache.flume.channel.file.encryption.KeyProvider$Builder	–	org.example.MyKeyProvider
org.apache.flume.channel.file.encryption.CipherProvider	aesctrnopadding	org.apache.flume.channel.file.encryption.AESCTRNoPaddingProvider
org.apache.flume.channel.file.encryption.CipherProvider	–	org.example.MyCipherProvider

组件接口	别名	实现类
org.apache.flume.serialization.EventSerializer$Builder	text	org.apache.flume.serialization.BodyTextEventSerializer$Builder
org.apache.flume.serialization.EventSerializer$Builder	avro_event	org.apache.flume.serialization.FlumeEventAvroEventSerializer$Builder
org.apache.flume.serialization.EventSerializer$Builder	–	org.example.MyEventSerializer$Builder

配置命名约定

本文档之前给出的例子都按照下面的别名规范来命名，以使所有示例中的名称保持简短和一致。

提示

前面的每个配置范例里面的Agent都叫做a1，就是遵循了下表的约定。

别名	代表组件
a	agent
c	channel
r	source
k	sink
g	sink group
i	interceptor
y	key

别名	代表组件
h	host
s	serializer

提示

下表示译者加的，1.9新增的 配置文件过滤器 这个功能的范例中使用的命名是f，猜测可能因为Flume官方文档不是一个人维护，写某一小节文档的人一时也没想起来要将别名列在这里。

别名	代表组件
f	Configuration Filters

其他关联文章:

1. Flume Sinks
2. 配置 Flume Source

delucia / 2021-05-07 / Flume