

Flume 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

Java 运行环境 - Java 1.8 或更高版本

内存 - 足够的内存 用来配置 Sources、Channels 和 Sinks

硬盘空间 - 足够的硬盘用来配置 Channels 和 Sinks

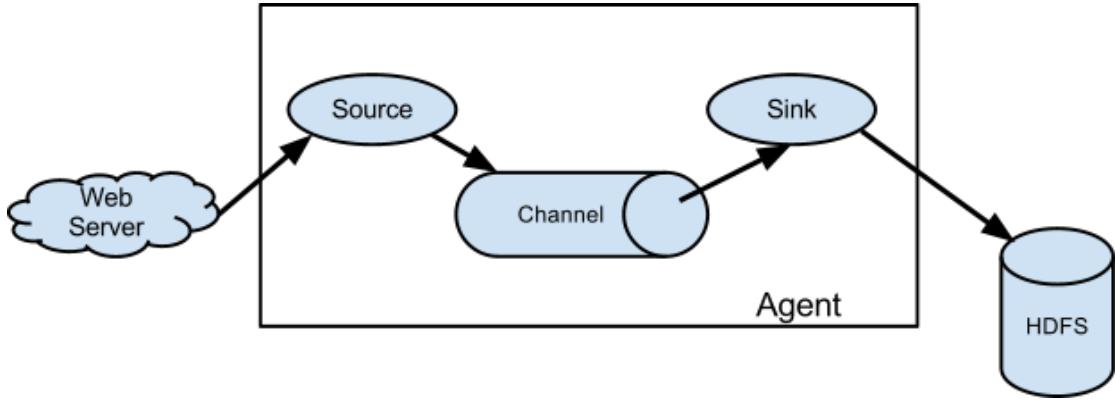
目录权限 - Agent 用来读写目录的权限

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 Event 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 。

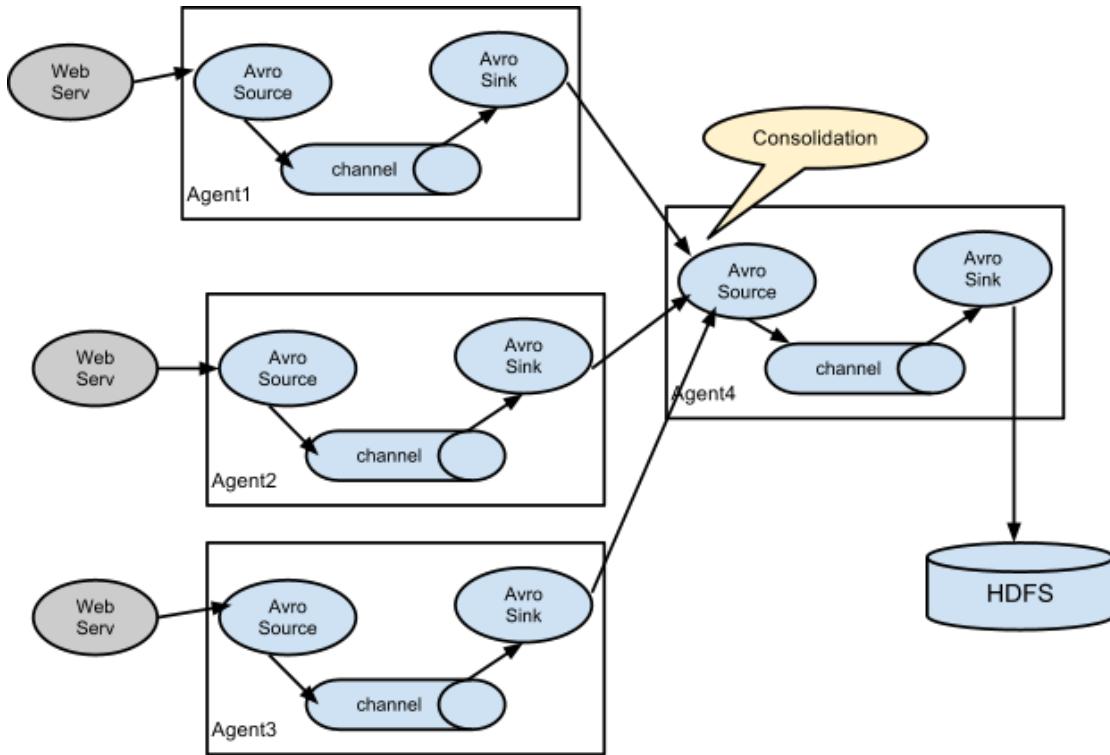
复杂流

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 需要知道文件系统的 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。

```
# example.conf: 一个单节点的 Flume 实例配置
```

```
# 配置 Agent a1 各个组件的名称
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

# 配置 Agent a1 的 source r1 的属性
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

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

# 配置 Agent a1 的 channel c1 的属性，channel 是用来缓冲 Event 数据的
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 event 也就越多
a1.channels.c1.transactionCapacity = 100    #source 和 sink 从内存 channel 每次事务传输的 event 数量

# 把 source 和 sink 绑定到 channel 上
```

```
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 -  
1 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 world!. }
```

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：

```
1 a1.sources = r1
2 a1.sources.r1.type = netcat
3 a1.sources.r1.bind = 0.0.0.0
4 a1.sources.r1.port = ${NC_PORT}
5 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 --
1 name a1 -Dflume.root.logger=INFO,console -
DpropertiesImplementation=org.apache.flume.node.EnvVarResolverProperties
```

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.flume.log.printconfig=true -Dorg.apache.flume.log.rawdata=true
```

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。

```
1 - /flume  
2 |- /a1 [Agent config file]  
3 |- /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：

```
1 $ 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 `plugins.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：

`lib` – 插件自己的 jar 包

`libext` – 插件依赖的其他所有 jar 包

`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:

```
1 $ 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\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 支持以下比较流行日志类型读取：

Avro

Thrift

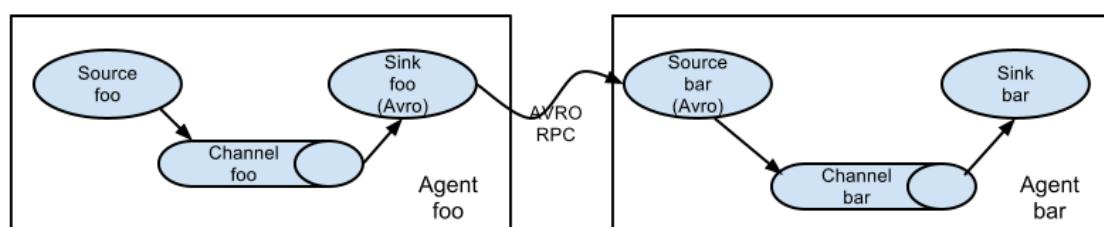
Syslog

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

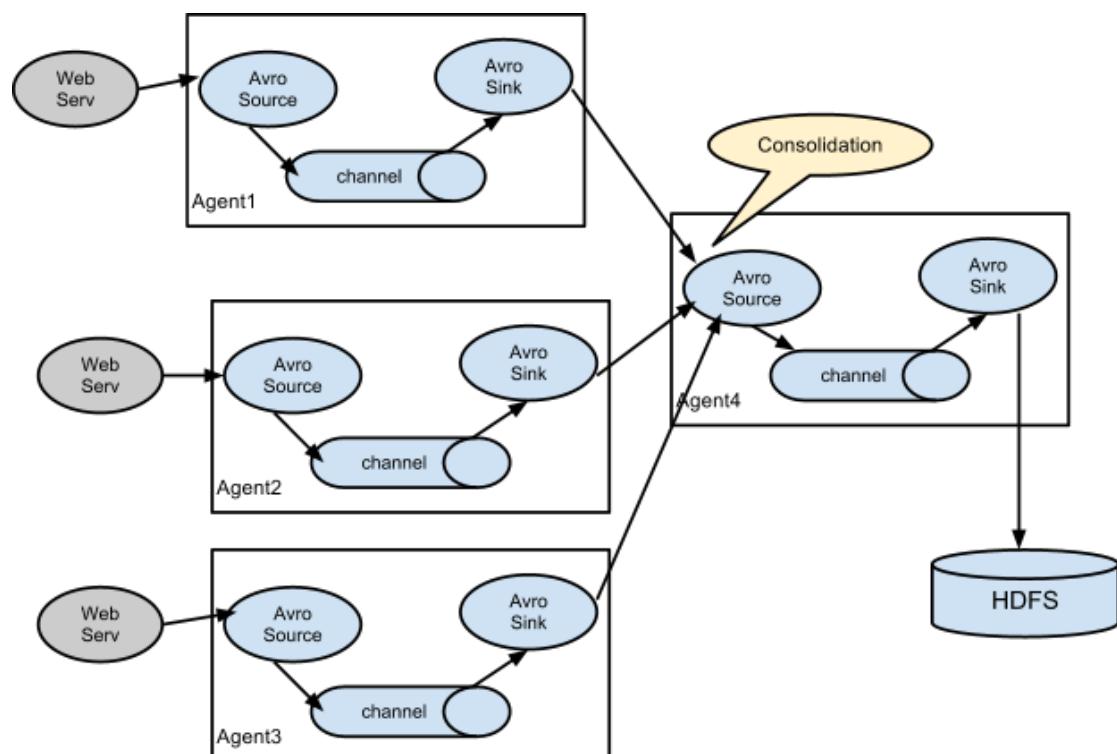
提示：一小节介绍了几种典型的 Flume 的多 Agent 以及一个 Agent 中多路输出等部署方式。



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。

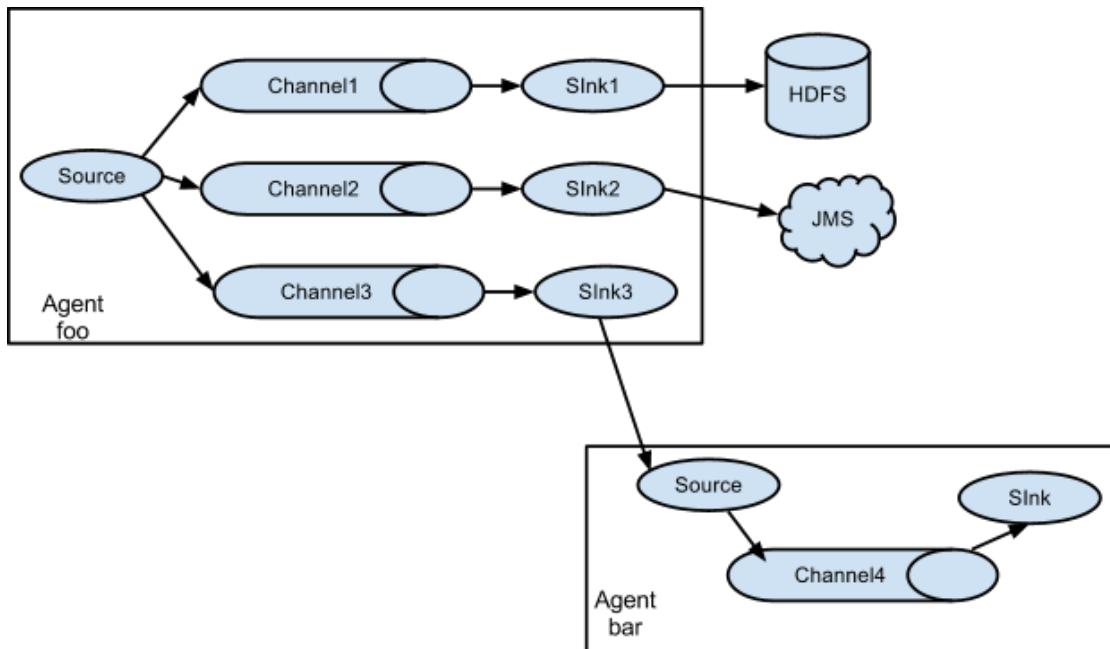
提示

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。

1 # 列出 Agent 实例的所有 Source、Channel、Sink

2 <Agent>.sources = <Source>

3 <Agent>.sinks = <Sink>

4 <Agent>.channels = <Channel1> <Channel2>

```

6 # 设置 Channel 和 Source 的关联

7 <Agent>.sources.<Source>.channels = <Channel1> <Channel2> ...           # 这行
配置就是给一个 Source 配置了多个 channel

8

9
# 设置 Channel 和 Sink 的关联

10 <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>

1
agent_foo.sources = avro-appserver-src-1

2
agent_foo.sinks = hdfs-sink-1

3
agent_foo.channels = mem-channel-1

4

5
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 mem-channel-1
6

7 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>

1
2
3 # properties for channels
4
5 <Agent>.channel.<Channel>.<someProperty> = <someValue>
6
7
8

# properties for sinks

<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。

```
1 <em># 列出所有的组件</em>
2
3 agent_foo.sources = avro-AppSrv-source
4
5 agent_foo.sinks = hdfs-Cluster1-sink
6
7
8 agent_foo.channels = mem-channel-1
9
10
11
12 # 将 source 和 sink 与 channel 相连接
13
14
15 # (省略)
16
```

```
17
18
19 # 配置 avro-AppSrv-source 的属性
20
21
22 agent_foo.sources.avro-AppSrv-source.type = avro          # avro-
23 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

# 配置 mem-channel-1 的属性

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

# 配置 hdfs-Cluster1-sink 的属性

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
```

```
#...
```

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：

```
1 <em># 这样列出 Agent 的所有 source、sink 和 channel，多个用空格分隔</em>
```

```
2 <Agent>.sources = <Source1> <Source2>
```

```
3 <Agent>.sinks = <Sink1> <Sink2>
```

```
4 <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：

```
1 <em># 列出当前配置所有的 source、sink 和 channel</em>
```

```
2 agent_foo.sources = avro-AppSrv-source1 exec-tail-
    source2 # 该 agent 中有 2 个 source，分别是：avro-
3 AppSrv-source1 和 exec-tail-source2
```

```
4 agent_foo.sinks = hdfs-Cluster1-sink1 avro-forward-
    sink2 # 该 agent 中有 2 个 sink，分别是：hdfs-
5 Cluster1-sink1 和 avro-forward-sink2
```

```
6 agent_foo.channels = mem-channel-1 file-channel-
    2 # 该 agent 中有 2 个 channel，分别
7 是：mem-channel-1 file-channel-2
```

```
8
```

```

9 # 这里是第一个流的配置

10 agent_foo.sources.avro-AppSrv-source1.channels = mem-channel-
    1           # 与 avro-AppSrv-source1 相连接的 channel 是 mem-channel-1

11
agent_foo.sinks.hdfs-Cluster1-sink1.channel = mem-channel-
12 1           # 与 hdfs-Cluster1-sink1 相连接的 channel 是 mem-channel-
    1

# 这里是第二个流的配置

agent_foo.sources.exec-tail-source2.channels = file-channel-
2           # 与 exec-tail-source2 相连接的 channel 是 file-channel-2

agent_foo.sinks.avro-forward-sink2.channel = file-channel-2

```

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：

```

1 <em># 列出这个 Agent 的 source、sink 和 channel</em>

2 agent_foo.sources = avro-AppSrv-source

```

```
3 agent_foo.sinks = avro-forward-sink  
  
4 agent_foo.channels = file-channel  
  
5  
  
6 # 把 source、channel、sink 连接起来，组成一个流  
  
7 agent_foo.sources.avro-AppSrv-source.channels = file-channel  
  
8 agent_foo.sinks.avro-forward-sink.channel = file-channel  
  
9  
  
10 # avro-forward-sink 的属性配置  
  
11 agent_foo.sinks.avro-forward-sink.type = avro  
  
12 agent_foo.sinks.avro-forward-sink.hostname = 10.1.1.100  
  
13 agent_foo.sinks.avro-forward-sink.port = 10000  
  
14  
  
15 # 其他部分配置（略）  
  
16 #...
```

存储到 HDFS 的 Agent 配置：

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

```
4 agent_foo.channels = mem-channel          # 只有一个 channel 叫做:  
mem-channel  
5  
  
6          # 把 source、channel、sink 连接起来，组成一个流  
7  
agent_foo.sources.avro-collection-source.channels = mem-channel  
8  
agent_foo.sinks.hdfs-sink.channel = mem-channel  
9  
  
10         # Avro Source 的属性配置  
11  
agent_foo.sources.avro-collection-source.type = avro  
12  
agent_foo.sources.avro-collection-source.bind = 10.1.1.100  
13  
agent_foo.sources.avro-collection-source.port = 10000  
14  
  
15         # 其他部分配置（略）  
16  
#...
```

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  
1 channel 和 2 个 sink</em>  
  
2 <Agent>.sources = <Source1>  
  
3 <Agent>.sinks = <Sink1> <Sink2>  
  
4 <Agent>.channels = <Channel1> <Channel2>  
  
5  
  
6 # 指定与 source1 连接的 channel，这里配置了两个 channel  
  
7 <Agent>.sources.<Source1>.channels = <Channel1> <Channel2>  
  
8  
  
9 # 将两个 sink 分别与两个 channel 相连接  
  
10 <Agent>.sinks.<Sink1>.channel = <Channel1>  
  
11 <Agent>.sinks.<Sink2>.channel = <Channel2>  
  
12  
  
13 # 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）  
14  
<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，则选择器会读取 Event header 中的 abc 属性来作为分发的依据
<Agent>.sources.<Source1>.selector.mapping.<Value1> = <Channel1> # 加入这里 Value1 配置的是 3，则 Event header 中 abc 属性的值等于 3 的 Event 会被发送到 channel1 上
<Agent>.sources.<Source1>.selector.mapping.<Value2> = <Channel1>
<Channel2> # 同上，Event header 中 abc 属性等于 Value2 的 Event 会被发送到 channel1 和 channel2 上
<Agent>.sources.<Source1>.selector.mapping.<Value3> = <Channel2> # 同上规则，Event header 中 abc 属性等于 Value3 的 Event 会被发送到 channel2 上
#...

<Agent>.sources.<Source1>.selector.default = <Channel2> #
Event header 读取到的 abc 属性值不属于上面配置的任何一个的话，默认就会发送到这个 channel2 上

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：

```
1 <em># 列出了 Agent 的所有 source、sink 和 channel</em>
2 agent_foo.sources = avro-AppSrv-source1
3 agent_foo.sinks = hdfs-Cluster1-sink1 avro-forward-sink2
4 agent_foo.channels = mem-channel1-1 file-channel1-2
5
6 # 让 source 与两个 channel 相连接
7 agent_foo.sources.avro-AppSrv-source1.channels = mem-channel1-1 file-channel1-2
8
9
10 # 分别设定两个 sink 对应的 channel
11 agent_foo.sinks.hdfs-Cluster1-sink1.channel = mem-channel1-1
12 agent_foo.sinks.avro-forward-sink2.channel = file-channel1-2
```

```

13
14 # source 的 channel 选择器配置
15 agent_foo.sources.avro-AppSrv-source1.selector.type = 
16 multiplexing # 选择器类
17 型是多路复用，非复制
18 agent_foo.sources.avro-AppSrv-source1.selector.header = 
19 State # 读取
Event header 中名字叫做 State 的属性值，以这个值作为分发的映射依据
agent_foo.sources.avro-AppSrv-source1.selector.mapping.CA = mem-channel-
1 # State=CA 时，Event 发送到 mem-
channel-1 上
agent_foo.sources.avro-AppSrv-source1.selector.mapping.AZ = file-channel-
2 # State=AZ 时，Event 发送到 file-channel-
2 上
agent_foo.sources.avro-AppSrv-source1.selector.mapping.NY = mem-channel-
file-channel-2 # State=NY 时，Event 发送到 mem-channel-1 和 file-
channel-2 上
agent_foo.sources.avro-AppSrv-source1.selector.default = mem-channel-
1 # 如果 State 不等于上面配置的任
何一个值，则 Event 会发送到 mem-channel-1 上

```

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
1 agent_foo.sources.avro-AppSrv-source1.selector.mapping.CA = mem-channel-
2 1 # CA 被第一次映射到 mem-
3 channel-1
4 agent_foo.sources.avro-AppSrv-source1.selector.mapping.AZ = file-channel-2
5 agent_foo.sources.avro-AppSrv-source1.selector.mapping.NY = mem-channel-1
6 file-channel-2
7 agent_foo.sources.avro-AppSrv-source1.selector.optional.CA = mem-channel-1
8 file-channel-2 # 关键看这行，State=CA 的映射在上面本来已经
9 指定到 mem-channel-1 了，这里又另外配置了两个 channel
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
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
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

System property	Environment variable	描述
		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命令历史中，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：

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

```
2 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命令历史中。

```
1 export FLUME_SSL_KEYSTORE_PATH=/path/to/keystore.jks
```

```
2 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

属性	默认值	解释
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
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-	-	Java keystore 的密码, 如果此参数未配置就会默认使

属性	默 认 值	解 释
password		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 <i>keystore</i> 和 <i>keystore-password</i> 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
include-	-	可使用的 SSL/TLS 协议的以空格分隔的列表。最终程序

属性	默认值	解释
protocols		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 <i>agent-principal</i> 和 <i>agent-keytab</i> 。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
selector.*		channel 选择器的相关属性，具体属性根据设定的 <code>selector.type</code> 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，Tailldir 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	DEFAUL T	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
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

属性	默 认 值	解 释
		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：

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。

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

属性名	默认值	解释
		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
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

属性名	默认值	解释
		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

属性名	默认值	解释
		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，直到达到设定的最大值为止。
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

属性名	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

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

属性名	默认值	解释
		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

属性名	默认值	解释
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。
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

属性名	默认值	解释
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

属性名	默认值	解释
		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

属性名	默认值	解释
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.r1.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
consumerSecret	-	OAuth consumer secret
accessToken	-	OAuth access token
accessTokenSecret	-	OAuth token secret
maxBatchSize	100 0	每次获取 twitter 数据的数据集大小，简单说就是一次取多少
maxBatchDurationMillis	100 0	每次批量获取数据的最大等待时间（毫秒）

配置范例：

```

1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = org.apache.flume.source.twitter.TwitterSource
4 a1.sources.r1.channels = c1
5 a1.sources.r1.consumerKey = YOUR_TWITTER_CONSUMER_KEY
6 a1.sources.r1.consumerSecret = YOUR_TWITTER_CONSUMER_SECRET

```

```

7 a1.sources.r1.accessToken = YOUR_TWITTER_ACCESS_TOKEN
8 a1.sources.r1.accessTokenSecret = YOUR_TWITTER_ACCESS_TOKEN_SECRET
9 a1.sources.r1.maxBatchSize = 10
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
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。

属性名	默认值	解释
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。
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。

属性名	默认值	解释
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

属性名	默认值	解释
		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：

```

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

```

正则表达式 topic 订阅的示例：

```

1 tier1.sources.source1.type = org.apache.flume.source.kafka.KafkaSource
2 tier1.sources.source1.channels = channel1
3 tier1.sources.source1.kafka.bootstrap.servers = localhost:9092
4 tier1.sources.source1.kafka.topics.regex = ^topic[0-9]$
5 <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](#) 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 或密钥库类型。

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-13:9093
2 a1.sources.source1.kafka.topics = mytopic
3 a1.sources.source1.kafka.consumer.group.id = flume-consumer
4 a1.sources.source1.kafka.consumer.security.protocol = SSL
5 <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
6 truststore，就可以把这两个参数加上。</em>
7 a1.sources.source1.kafka.consumer.ssl.truststore.location=/path/to/truststore.j
8 ks
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：

```
1 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。

```
1 <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
```

2 必再重复配置

```
3 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：

```
1 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：

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

使用 `SASL_PLAINTEXT` 的示例安全配置：

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

使用 `SASL_SSL` 的安全配置范例：

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

```
9 <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  
1 必再重复配置</em>  
0 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。

```
1 Client {  
2     com.sun.security.auth.module.Krb5LoginModule required  
3         useKeyTab=<strong>true</strong>  
4         storeKey=<strong>true</strong>  
5         keyTab="/path/to/keytabs/flume.keytab"  
6         principal="flume/flumehost1.example.com@YOURKERBEROSREALM";  
7 };  
8  
9 KafkaClient {  
10    com.sun.security.auth.module.Krb5LoginModule required  
11    useKeyTab=<strong>true</strong>  
12    storeKey=<strong>true</strong>  
13    keyTab="/path/to/keytabs/flume.keytab"  
14    principal="flume/flumehost1.example.com@YOURKERBEROSREALM";  
15 };
```

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 channel1 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：

```

1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = netcat
4 a1.sources.r1.bind = 0.0.0.0
5 a1.sources.r1.port = 6666
6 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：

```
1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = netcatudp
4 a1.sources.r1.bind = 0.0.0.0
5 a1.sources.r1.port = 6666
6 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：

```
1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = seq
4 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是，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： <code>syslogtcp</code>
host	-	要监听的 hostname 或者 IP 地址
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 <code>all</code> 、 <code>none</code> 或自定义指定保留的字段。如果设置为 <code>all</code> ，则会保留 <code>Priority</code> 、 <code>Timestamp</code> 和 <code>Hostname</code> 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： <code>priority</code> 、 <code>version</code> 、 <code>timestamp</code> 、 <code>hostname</code> ），用空格分开即可。现在已经不建议使用 <code>true</code> 和 <code>false</code> ，建议改用 <code>all</code> 和 <code>none</code> 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

属性	默认值	解释
		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
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

属性	默认值	解释
		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：

```

1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = syslogtcp
4 a1.sources.r1.port = 5140
5 a1.sources.r1.host = localhost
6 a1.sources.r1.channels = c1

```

Multipoint 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。
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

属性	默认值	解释
		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
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

属性	默认值	解释
		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：

```

1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = multiport_syslogtcp
4 a1.sources.r1.channels = c1
5 a1.sources.r1.host = 0.0.0.0
6 a1.sources.r1.ports = 10001 10002 10003
7 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 和 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。
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

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

配置范例：

```

1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = syslogudp
4 a1.sources.r1.port = 5140
5 a1.sources.r1.host = localhost
6 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

属性	默认值	解释
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
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

属性	默认值	解释
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
QueuedThread Pool.*		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。
HttpConfigur ation.*		作用 在 Jetty org.eclipse.jetty.server.HttpCo nfiguration 上的一些特定设置
SslContextFa ctory.*		作用 在 Jetty org.eclipse.jetty.util.ssl.SslC ontextFactory 上的一些特定设置 (只有 ssl 设置为 true 的时候才生效)
ServerConnec tor.*		作用 在 Jetty org.eclipse.jetty.server.Server Connector 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 <i>keystore-password</i> 。弃用的参数会被新的参数覆盖
excludeProtocols	SSLv3	改用 <i>exclude-protocols</i> 。弃用的参数会被新的参数覆盖
enableSSL	false	改用 <i>ssl</i> , 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:

```
1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = http
4 a1.sources.r1.port = 5140
5 a1.sources.r1.channels = c1
6 a1.sources.r1.handler = org.example.rest.RestHandler
7 a1.sources.r1.handler.nickname = random props
8 a1.sources.r1.HttpConfiguration.sendServerVersion      =      false      #
sendServerVersion 是 HttpConfiguration 的参数, 点击上面的文档链接就能找到了
9 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了

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：

```
1 [ {
2     "headers" : {
3         "timestamp" : "434324343",
4         "host" : "random_host.example.com"
5     },
6     "body" : "random_body"
7 },
8 {
9     "headers" : {
10        "namenode" : "namenode.example.com",
11        "datanode" : "random_datanode.example.com"
12    },
13     "body" : "really_random_body"
14 } ]
```

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：

```
1 Type type = <strong>new</strong> TypeToken<List<JSONEvent>>()
2 {}.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：

```

1 a1.sources = stresssource-1
2 a1.channels = memoryChannel-1
3 a1.sources.stresssource-1.type = org.apache.flume.source.StressSource
4 a1.sources.stresssource-1.size = 10240
5 a1.sources.stresssource-1.maxTotalEvents = 1000000

```

```
6 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
type	-	组件类型，这个是：org.apache.flume.source.avroLegacy.AvroLegacySource

属性	默认值	解释
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:

```

1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = org.apache.flume.source.avroLegacy.AvroLegacySource
4 a1.sources.r1.host = 0.0.0.0
5 a1.sources.r1.bind = 6666
6 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
type	-	组件类类型, 这个是: org.apache.flume.source.thriftLegacy.ThriftLegacySource
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
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:

```

1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = org.apache.flume.source.thriftLegacy.ThriftLegacySource
4 a1.sources.r1.host = 0.0.0.0
5 a1.sources.r1.bind = 6666
6 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

属性	默认值	解释
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：

```

1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = org.example.MySource
4 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	1638400	Thrift 默认的 FrameBuffer 大小

属性	默认值	解释
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:

```

1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.type = org.apache.flume.source.scribe.ScribeSource
4 a1.sources.r1.port = 1463
5 a1.sources.r1.workerThreads = 5
6 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

转义符	解释
	aabc 的值
%t	毫秒值的时间戳 (同 <code>System.currentTimeMillis()</code> 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)

转义符	解释
%k	小时 (0 到 23) , 注意跟 %H 的区别
%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

转义符	解释
%[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
<code>channel</code>	-	与 Sink 连接的 channel
<code>type</code>	-	组件类型，这个是： hdfs
<code>hdfs.path</code>	-	HDFS 目录路径（例如： <code>hdfs://namenode/flume/webdata/</code> ）
<code>hdfs.filePrefix</code>	<code>FlumeData</code>	Flume 在 HDFS 文件夹下创建新文件的固定前缀
<code>hdfs.fileSuffix</code>	-	Flume 在 HDFS 文件夹下创建新文件的后缀（比如： <code>.avro</code> ，注意这个“.”不会自动添加，需要显式配置）
<code>hdfs.inUsePrefix</code>	-	Flume 正在写入的临时文件前缀，默认没有
<code>hdfs.inUseSuffix</code>	<code>.tmp</code>	Flume 正在写入的临时文件后缀
<code>hdfs.emptyInUseSuffix</code>	<code>false</code>	如果设置为 <code>false</code> 上面的 <code>hdfs.inUseSuffix</code> 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

属性名	默认值	解释
		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
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.rollTimerPeriod	1	每个 HDFS Sink 实例调度定时文件滚动的线程数

属性名	默认值	解释
olSize		
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 <code>hdfs.roundUnit</code> ）例子：假设当前时间戳是 18:32:01， <code>hdfs.roundUnit = minute</code> 如果 <code>roundValue=5</code> , 则时间戳会取为：18:30 如果 <code>roundValue=7</code> , 则时间戳会取为：18:28 如果 <code>roundValue=10</code> , 则时间戳会取为：18:30
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

属性名	默认值	解释
a1		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:

```

1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = hdfs
4 a1.sinks.k1.channel = c1
5 a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M/%S
6 a1.sinks.k1.hdfs.filePrefix = events-
7 a1.sinks.k1.hdfs.round = true
8 a1.sinks.k1.hdfs.roundValue = 10
9 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
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 string, 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 <i>batchSize</i> 个 Event，这个参数

属性	默认值	解释
		和 <code>batchSize</code> 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
round	false	是否启用时间截舍入机制
roundUnit	minute	舍入值的单位, 可选值: second 、 minute 、 hour

属性	默认值	解释
roundValue	1	舍入到小于当前时间的最高倍数（使用 <code>roundUnit</code> 配置的单位）例子 1: <code>roundUnit=second</code> , <code>roundValue=10</code> , 则 <code>14:31:18</code> 这个时间戳会被舍入到 <code>14:31:10</code> ; 例子 2: <code>roundUnit=second</code> , <code>roundValue=30</code> , 则 <code>14:31:18</code> 这个时间戳会被舍入到 <code>14:31:00</code> , <code>14:31:42</code> 这个时间戳会被舍入到 <code>14:31:30</code> ;
timeZone	Local Time	应用于解析分区中转义序列的时区名称, 比如: <code>America/Los_Angeles</code> 、 <code>Asia/Shanghai</code> 、 <code>Asia/Tokyo</code> 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
<code>serializer.delimiter</code>	,	(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
<code>serializer.fieldnames</code>	-	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。
<code>serializer.serdeSeparator</code>	Ctrl -A	(类型：字符) 自定义底层序列化器的分隔符。如果 <code>serializer.fieldnames</code> 中的字段与 Hive 表列的顺序相同，则 <code>serializer.delimiter</code> 与 <code>serializer.serdeSeparator</code> 相同，并且 <code>serializer.fieldnames</code> 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 <code>serializer.delimiter</code> 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
<code>%{host}</code>	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 <code>%{aabc}</code> 将读取 header 中 key 为 aabc 的值
<code>%t</code>	毫秒值的时间戳（同 <code>System.currentTimeMillis()</code> 方法）
<code>%a</code>	星期的缩写 (Mon、Tue 等)

转义符	解释
%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)
%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

转义符	解释
%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：

```

1 create table weblogs ( id int , msg string )

2         partitioned by (continent string, country string, time string)

3         clustered by (id) into 5 buckets

4         stored as orc;

```

配置范例：

```
1 a1.channels = c1
```

```
2 a1.channels.c1.type = memory

3 a1.sinks = k1

4 a1.sinks.k1.type = hive

5 a1.sinks.k1.channel = c1

6 a1.sinks.k1.hive.metastore = thrift://127.0.0.1:9083

7 a1.sinks.k1.hive.database = logsdb

8 a1.sinks.k1.hive.table = weblogs

9 a1.sinks.k1.hive.partition = asia, {country}, %y-%m-%d-%H-%M

10 a1.sinks.k1.useLocalTimeStamp = false

11 a1.sinks.k1.round = true

12 a1.sinks.k1.roundValue = 10

13 a1.sinks.k1.roundUnit = minute

14 a1.sinks.k1.serializer = DELIMITED

15 a1.sinks.k1.serializer.delimiter = "\t"

16 a1.sinks.k1.serializer.serdeSeparator = '\t'

17 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
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：

```
1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = logger
4 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
<code>channel</code>	-	与 Sink 绑定的 channel
<code>type</code>	-	组件类型，这个是： avro.
<code>hostname</code>	-	监听的服务器名 (hostname) 或者 IP
<code>port</code>	-	监听的端口
<code>batch-size</code>	100	每次批量发送的 Event 数
<code>connect-timeout</code>	20000	第一次连接请求（握手）的超时时间，单位：毫秒
<code>request-timeout</code>	20000	请求超时时间，单位：毫秒
<code>reset-connection-interval</code>	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。
<code>compression-type</code>	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
<code>compression-level</code>	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
<code>ssl</code>	false	设置为 true 表示开启 SSL 下面的 <code>truststore</code> 、 <code>truststore-password</code> 、 <code>truststore-type</code> 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 (<code>trust-all-certs</code>)
<code>trust-all-certs</code>	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

属性	默认值	解释
		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。
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：

```

1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = avro
4 a1.sinks.k1.channel = c1
5 a1.sinks.k1.hostname = 10.10.10.10
6 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
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 <i>truststore</i> 、 <i>truststore-password</i> 、 <i>truststore-type</i> 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

属性	默认值	解释
type		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 <i>client-principal</i> 、 <i>client-keytab</i> 和 <i>server-principal</i> 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 <i>client-principal</i> 结合使用的 keytab 文件路径, 用于对 kerberos KDC 进行身份验证。
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:

```

1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = thrift
4 a1.sinks.k1.channel = c1
5 a1.sinks.k1.hostname = 10.10.10.10
6 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
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 <code>splitlines</code> 设置为 <code>true</code> , 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:

```

1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = irc
4 a1.sinks.k1.channel = c1
5 a1.sinks.k1.hostname = irc.yourdomain.com
6 a1.sinks.k1.nick = flume
7 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.
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：

```
1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = file_roll
4 a1.sinks.k1.channel = c1
5 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：

```
1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = null
4 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 <i>hbase.zookeeper.quorum</i> 的值
znodeParent	/hbase	ZooKeeper 中 HBase 的 Root ZNode 路径, hbase-site.xml 中 <i>zookeeper.znode.parent</i> 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

属性	默认值	解释
		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
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：

```

1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = hbase
4 a1.sinks.k1.table = foo_table
5 a1.sinks.k1.columnFamily = bar_cf
6 a1.sinks.k1.serializer = org.apache.flume.sink.hbase.RegexHbaseEventSerializer
7 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

属性	默认值	解释
		channel
type	-	组件类型，这个是： hbase2
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 <i>hbase.zookeeper.quorum</i> 的值
znodeParent	/hbase	ZooKeeper 中 HBase 的 Root ZNode 路径, hbase-site.xml 中 <i>zookeeper.znode.parent</i> 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.SimpleHBaseEventSerializer	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

属性	默认值	解释
ab		的 Kerberos keytab 文件目录

配置范例：

```

1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = hbase2
4 a1.sinks.k1.table = foo_table
5 a1.sinks.k1.columnFamily = bar_cf
6 a1.sinks.k1.serializer = org.apache.flume.sink.hbase2.RegexHBase2EventSerializer
7 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

属性	默认值	解释
zookeeperQuorum	-	Zookeeper 节点 (host:port 格式, 多个用逗号分隔), hbase-site.xml 中属性 <i>hbase.zookeeper.quorum</i> 的值
znodeParent	/hbase	ZooKeeper 中 HBase 的 Root ZNode 路径, hbase-site.xml 中 <i>zookeeper.znode.parent</i> 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.SimpleASynchHbaseEventSerializer	序列化器
serializer.*	-	序列化器的一些属性
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：

```
1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = asynchbase
4 a1.sinks.k1.table = foo_table
5 a1.sinks.k1.columnFamily = bar_cf
6 a1.sinks.k1.serializer = org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer
7 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： <code>org.apache.flume.sink.solr.morphline.MorphlineSolrSink</code>
morphlineFile	-	morphline 配置文件的相对或者绝对路径，例如： <code>/etc/flume-ng/conf/morphline.conf</code>
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

属性	默认值	解释
batchSize	1000	单个事务操作的最大 Event 数量
batchDurationM illis	1000	事务的最大超时时间（毫秒）。达到这个时间或者达到 <code>batchSize</code> 都会触发提交事物。
handlerClass	org.apache.flume.sink.solr. morphline.MorphlineHandlerI mpl	实现了 <code>org.apache.flume.sink.solr.morphline.MorphlineHandler</code> 接口的实现类的全限定类名
isProductionMo de	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。
recoverableExc eptionClasses	org.apache.solr.client.solr j.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 <code>isProductionMode</code> 标志设置为 <code>true</code> 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。
isIgnoringReco verableExcepti ons	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

属性	默认值	解释
		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：

```

1 a1.channels = c1

2 a1.sinks = k1

3 a1.sinks.k1.type = org.apache.flume.sink.solr.morphline.MorphlineSolrSink

4 a1.sinks.k1.channel = c1

5 a1.sinks.k1.morphlineFile = /etc/flume-ng/conf/morphline.conf

6 <em># a1.sinks.k1.morphlineId = morphline1</em>

7 <em># a1.sinks.k1.batchSize = 1000</em>

8 <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
type	-	组件类型，这个是： <code>org.apache.flume.sink.elasticsearch.ElasticSearchSink</code>
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

属性	默认值	解释
		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.EventLogstashEvent	序列化器必须实现 <i>ElasticSearchEventSerializer</i>

属性	默认值	解释
	Serializer	或 <code>ElasticSearchIndexRequestBuilderFactory</code> 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：

```

1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = elasticsearch
4 a1.sinks.k1.hostNames = 127.0.0.1:9200,127.0.0.2:9300
5 a1.sinks.k1.indexName = foo_index
6 a1.sinks.k1.indexType = bar_type
7 a1.sinks.k1.clusterName = foobar_cluster
8 a1.sinks.k1.batchSize = 500
9 a1.sinks.k1.ttl = 5d
10 a1.sinks.k1.serializer =
11 org.apache.flume.sink.elasticsearch.ElasticSearchDynamicSerializer
12 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
<code>channel</code>	-	与 Sink 绑定的 channel
<code>type</code>	-	组件类型，这个是： <code>org.apache.flume.sink.kite.DatasetSink</code>
<code>kite.dataset.uri</code>	-	要打开的数据集的 URI
<code>kite.repo.uri</code>	-	要打开的存储库的 URI（不建议使用，请改用 <code>kite.dataset.uri</code> ）
<code>kite.dataset.namespace</code>	-	将写入记录的数据集命名空间（不建议使用，请改用 <code>kite.dataset.uri</code> ）
<code>kite.dataset.name</code>	-	将写入记录的数据集名称（不建议使用，请改用 <code>kite.dataset.uri</code> ）
<code>kite.batchSize</code>	100	每批中要处理的记录数

属性	默 认 值	解 释
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 <code>kite.batchSize</code> 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 <code>the kite.flushable.commitOnBatch</code> 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 <code>EntityParser.Builder</code> 接口实现类的全限定类名
kite.failurePolicy	retr y	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 <code>kite.error.dataset.uri</code> 这个数据

属性	默认值	解释
		集。还可以填自定义的处理策略类的全限定类名（需实现 <code>FailurePolicy.Builder</code> 接口）
<code>kite.error.dataset.uri</code>	-	保存失败的 Event 存储的数据集。当上面的参数 <code>kite.failurePolicy</code> 设置为 save 时，此参数必须进行配置。
<code>auth.kerberosPrincipal</code>	-	用于 HDFS 安全身份验证的 Kerberos 用户主体
<code>auth.kerberosKeytab</code>	-	Kerberos 安全验证主体的 keytab 本地文件系统路径
<code>auth.proxyUser</code>	-	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

属性	默认值	解释
		是： org.apache.flume.sink.kafka.KafkaSink
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

属性	默认值	解释
		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 <code>parseAsFlumeEvent</code> 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。
defaultPartitionId	-	指定所有 Event 将要发送到的 Kafka 分区 ID, 除非被 <code>partitionIdHeader</code> 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 <code>kafka.partitionner.class</code> 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 <code>EventDeliveryException</code> 。如果存在标头值，则此设置将覆盖 <code>defaultPartitionId</code> 。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

属性	默认值	解释
		作为分区 ID
allowTopicOverride	true	如果设置为 true, 会读取 Event header 中的名为 <code>topicHeader</code> 的属性值, 用它作为目标 topic。
topicHeader	topic	与上面的 <code>allowTopicOverride</code> 一起使用, <code>allowTopicOverride</code> 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 : <code>kafka.producer.linger.ms</code>

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.serializer (org.apache.kafka.common.serialization.StringSerializer) 和 value.serializer (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。

```
1 a1.sinks.k1.channel = c1
2 a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink
3 a1.sinks.k1.kafka.topic = mytopic
4 a1.sinks.k1.kafka.bootstrap.servers = localhost:9092
5 a1.sinks.k1.kafka.flumeBatchSize = 20
6 a1.sinks.k1.kafka.producer.acks = 1
7 a1.sinks.k1.kafka.producer.linger.ms = 1
8 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 或密钥库类型。

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
1 a1.sinks.sink1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
2 a1.sinks.sink1.kafka.topic = mytopic
3 a1.sinks.sink1.kafka.producer.security.protocol = SSL
4 # 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,
5 就可以把这两个参数加上。
6 a1.sinks.sink1.kafka.producer.ssl.truststore.location = /path/to/truststore.jks
7 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
1 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）：

```
Common Name (CN) https://tools.ietf.org/html/rfc6125#section-2.3  
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。

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。

```
1 a1.sinks.sink1.kafka.producer.ssl.keystore.location =  
2 /path/to/client.keystore.jks  
3 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：

```
1 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：

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

使用 SASL_PLAINTEXT 的示例安全配置：

```
1 a1.sinks.sink1.type = org.apache.flume.sink.kafka.KafkaSink
```

```
2 a1.sinks.sink1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-
3 3:9093
4 a1.sinks.sink1.kafka.topic = mytopic
5 a1.sinks.sink1.kafka.producer.security.protocol = SASL_PLAINTEXT
6 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
1 a1.sinks.sink1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-3:9093
2 a1.sinks.sink1.kafka.topic = mytopic
3 a1.sinks.sink1.kafka.producer.security.protocol = SASL_SSL
4 a1.sinks.sink1.kafka.producer.sasl.mechanism = GSSAPI
5 a1.sinks.sink1.kafka.producer.sasl.kerberos.service.name = kafka
6 <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
7 truststore，就可以把这两个参数加上。</em>
8 a1.sinks.sink1.kafka.producer.ssl.truststore.location = /path/to/truststore.jks
9 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。

```
1 KafkaClient {
2     com.sun.security.auth.module.Krb5LoginModule required
3         useKeyTab=<strong>true</strong>
4         storeKey=<strong>true</strong>
5         keyTab="/path/to/keytabs/flume.keytab"
6         principal="flume/flumehost1.example.com@YOURKERBEROSREALM";
7 };
```

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
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 <code>backoff.CODE</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 <code>rollback.CODE</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

属性	默认值	解释
defaultIncrementMetrics	false	是否默认进行统计计数，如果配置的 <code>incrementMetrics.CODE</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：

```

1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = http
4 a1.sinks.k1.channel = c1
5 a1.sinks.k1.endpoint = http://localhost:8080/someuri
6 a1.sinks.k1.connectTimeout = 2000
7 a1.sinks.k1.requestTimeout = 2000
8 a1.sinks.k1.acceptHeader = application/json

```

```
9 a1.sinks.k1.contentTypeHeader = application/json
10 a1.sinks.k1.defaultBackoff = true
11 a1.sinks.k1.defaultRollback = true
12 a1.sinks.k1.defaultIncrementMetrics = false
13 a1.sinks.k1.backoff.4XX = false
14 a1.sinks.k1.rollback.4XX = false
15 a1.sinks.k1.incrementMetrics.4XX = true
16 a1.sinks.k1.backoff.200 = false
17 a1.sinks.k1.rollback.200 = false
18 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 channel
type	-	组件类型，这个填你自定义 class 的全限定类名

配置范例：

```
1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = org.example.MySink
4 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
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:

```
1 a1.channels = c1
2 a1.channels.c1.type = memory
3 a1.channels.c1.capacity = 10000
4 a1.channels.c1.transactionCapacity = 10000
5 a1.channels.c1.byteCapacityBufferPercentage = 20
6 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

属性	默认值	解释
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：

```
1 a1.channels = c1
2 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：

- 通常与 channel 相关的配置值应用于 channel 配置级别，比如：
`a1.channel.k1.type =`
- 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
- 特定于生产者/消费者的属性以 kafka.producer 或 kafka.consumer 为前缀
- 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： <code>org.apache.flume.channel.kafka.KafkaChannel</code>
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 <code>hostname:port</code> ，多个用逗号分隔
kafka.topic	<code>flume-channel</code>	channel 使用的 Kafka topic
kafka.consumer.group.id	<code>flume</code>	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	<code>true</code>	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 <code>org.apache.flume.source.avro.AvroFlumeEvent</code> 可以在 Kafka 之外解析出 Flume source 的信息。

属性	默认值	解释
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.partitionner.class 指定的分发程序来分发。
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
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：

```

1 a1.channels.channel1.type = org.apache.flume.channel.kafka.KafkaChannel
2 a1.channels.channel1.kafka.bootstrap.servers      =      kafka-1:9092,kafka-
3 2:9092,kafka-3:9092
4 a1.channels.channel1.kafka.topic = channel1
4 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 或密钥库类型。

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
1 a1.channels.channel1.kafka.topic = channel1
2 a1.channels.channel1.kafka.consumer.group.id = flume-consumer
3 a1.channels.channel1.kafka.producer.security.protocol = SSL
4 <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
5 truststore，就可以把这两个参数加上。</em>
6 a1.channels.channel1.kafka.producer.ssl.truststore.location      =
7 /path/to/truststore.jks
8 a1.channels.channel1.kafka.producer.ssl.truststore.password      = <password> to
9 access the truststore>
10 a1.channels.channel1.kafka.consumer.security.protocol = SSL
11 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  
1 = HTTPS  
2 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。

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。

```
1 a1.channels.channel1.kafka.producer.ssl.keystore.location =  
1 /path/to/client.keystore.jks  
2 a1.channels.channel1.kafka.producer.ssl.keystore.password = <password to access  
3 the keystore>  
4 a1.channels.channel1.kafka.consumer.ssl.keystore.location =  
5 /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  
1 the key>  
2 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：

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

使用 SASL_PLAINTEXT 的示例安全配置：

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

使用 SASL_SSL 的安全配置范例：

```
1 a1.channels.channel1.type = org.apache.flume.channel.kafka.KafkaChannel
2 a1.channels.channel1.kafka.bootstrap.servers = kafka-1:9093,kafka-2:9093,kafka-
3:9093
4 a1.channels.channel1.kafka.topic = channel1
5 a1.channels.channel1.kafka.consumer.group.id = flume-consumer
6 a1.channels.channel1.kafka.producer.security.protocol = SASL_SSL
7 a1.channels.channel1.kafka.producer.sasl.mechanism = GSSAPI
8 a1.channels.channel1.kafka.producer.sasl.kerberos.service.name = kafka
9 # 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,
10 就可以把这两个参数加上。
11 a1.channels.channel1.kafka.producer.ssl.truststore.location =
12 /path/to/truststore.jks
13 a1.channels.channel1.kafka.producer.ssl.truststore.password = <password to
14 access the truststore>
15 a1.channels.channel1.kafka.consumer.security.protocol = SASL_SSL
16 a1.channels.channel1.kafka.consumer.sasl.mechanism = GSSAPI
17 a1.channels.channel1.kafka.consumer.sasl.kerberos.service.name = kafka
18 # 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,
19 就可以把这两个参数加上。
20 a1.channels.channel1.kafka.consumer.ssl.truststore.location =
21 /path/to/truststore.jks
22 a1.channels.channel1.kafka.consumer.ssl.truststore.password = <password to
23 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。

```
1 Client {  
2     com.sun.security.auth.module.Krb5LoginModule required  
3     useKeyTab=<strong>true</strong>  
4     storeKey=<strong>true</strong>  
5     keyTab="/path/to/keytabs/flume.keytab"  
6     principal="flume/flumehost1.example.com@YOURKERBEROSREALM";  
7 };  
8  
9 KafkaClient {  
10    com.sun.security.auth.module.Krb5LoginModule required  
11    useKeyTab=<strong>true</strong>  
12    storeKey=<strong>true</strong>  
13    keyTab="/path/to/keytabs/flume.keytab"  
14    principal="flume/flumehost1.example.com@YOURKERBEROSREALM";  
15 };
```

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/check point	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 ， backupCheckp ointDir 参数必须设置。

属性	默认值	解释
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 <code>checkpointDir</code> 相同
dataDirs	<code>~/.flume/file-channel/data</code>	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)
checkpointOnClose	true	channel 关闭时是否创建检查点文件。开启次功能可以避免

属性	默认值	解释
		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：AESCTRNPADDING
encryption.keyProvider	-	key 类型，目前只支持：JCEKSFILE
encryption.keyProvider.keyStoreFile	-	keystore 文件路径
encryption.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
encryption.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：

```

1 a1.channels = c1
2 a1.channels.c1.type = file
3 a1.channels.c1.checkpointDir = /mnt/flume/checkpoint
4 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:

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

使用相同的秘钥库密码生成秘钥 key-1:

```
keytool -genseckeckey -alias key-1 -keyalg AES -keysize 128 -validity 9000  
1 <strong>\</strong>  
2     -keystore      src/test/resources/test.keystore      -storetype      jceks  
3 <strong>\</strong>  
     -storepass keyStorePassword  
  
1 a1.channels.c1.encryption.activeKey = key-0  
1 a1.channels.c1.encryption.cipherProvider = AESCTRNPADDING  
2 a1.channels.c1.encryption.keyProvider = key-provider-0  
3 a1.channels.c1.encryption.keyProvider = JCEKSFILE  
4 a1.channels.c1.encryption.keyProvider.keyStoreFile = /path/to/my.keystore  
5 a1.channels.c1.encryption.keyProvider.keyStorePasswordFile =  
6 /path/to/my.keystore.password  
7 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  
1 a1.channels.c1.encryption.cipherProvider = AESCTRNPADDING  
2 a1.channels.c1.encryption.keyProvider = JCEKSFILE  
3 a1.channels.c1.encryption.keyProvider.keyStoreFile = /path/to/my.keystore  
4 a1.channels.c1.encryption.keyProvider.keyStorePasswordFile =  
5 /path/to/my.keystore.password  
6 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  
1 a1.channels.c1.encryption.cipherProvider = AESCTRNPADDING  
2 a1.channels.c1.encryption.keyProvider = JCEKSFILE  
3 a1.channels.c1.encryption.keyProvider.keyStoreFile = /path/to/my.keystore  
4 a1.channels.c1.encryption.keyProvider.keyStorePasswordFile =  
5 /path/to/my.keystore.password  
6 a1.channels.c1.encryption.keyProvider.keys = key-0 key-1  
7 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
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 <i>byteCapacityBufferPercentage</i> 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

属性	默认值	解释
		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：

```

1 a1.channels = c1
2 a1.channels.c1.type = SPILLABLEMEMORY
3 a1.channels.c1.memoryCapacity = 10000
4 a1.channels.c1.overflowCapacity = 1000000
5 a1.channels.c1.byteCapacity = 800000
6 a1.channels.c1.checkpointDir = /mnt/flume/checkpoint
7 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：

```

1 a1.channels = c1
2 a1.channels.c1.type = SPILLABLEMEMORY
3 a1.channels.c1.memoryCapacity = 0
4 a1.channels.c1.overflowCapacity = 1000000
5 a1.channels.c1.checkpointDir = /mnt/flume/checkpoint
6 a1.channels.c1.dataDirs = /mnt/flume/data

```

禁用掉磁盘存储，只使用内存 channel 的例子：

```

1 a1.channels = c1
2 a1.channels.c1.type = SPILLABLEMEMORY
3 a1.channels.c1.memoryCapacity = 100000
4 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： <code>org.apache.flume.channel.PseudoTxnMemoryChannel</code>
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： <code>org.example.myCustomChannel</code>

配置范例：

```
1 a1.channels = c1
2 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 source 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：

1 a1.sources = r1

2 a1.channels = c1 c2 c3

3 a1.sources.r1.selector.type = replicating

4 a1.sources.r1.channels = c1 c2 c3

5 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
1 a1.sources.r1.selector.type = multiplexing
2 a1.sources.r1.selector.header = state          #以每个 Event 的 header 中的
3 state 这个属性的值作为选择 channel 的依据
4 a1.sources.r1.selector.mapping.CZ = c1         #如果 state=CZ, 则选择 c1 这个
5 channel
6 a1.sources.r1.selector.mapping.US = c2 c3      #如果 state=US, 则选择 c2 和 c3
7 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：

```
1 a1.sources = r1
2 a1.channels = c1
3 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：

```
1 a1.sinkgroups = g1
2 a1.sinkgroups.g1.sinks = k1 k2
3 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

属性	默认值	解释
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：

```

1 a1.sinkgroups = g1
2 a1.sinkgroups.g1.sinks = k1 k2
3 a1.sinkgroups.g1.processor.type = failover
4 a1.sinkgroups.g1.processor.priority.k1 = 5
5 a1.sinkgroups.g1.processor.priority.k2 = 10
6 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 <i>AbstractSinkSelector</i>
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

属性	默认值	解释
		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 <code>maxTimeOut</code> 就是退避一个 sink 的最长时间

配置范例：

```

1 a1.sinkgroups = g1
2 a1.sinkgroups.g1.sinks = k1 k2
3 a1.sinkgroups.g1.processor.type = load_balance
4 a1.sinkgroups.g1.processor.backoff = true
5 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。

```

1 <em># test-flume.properties</em>
2 # 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
3 a1.sources = r1
4 a1.sinks = k1 k2
5 a1.channels = c1
6
7 # 使用 Stress Source 来生成测试用的 event
8 a1.sources.r1.type = org.apache.flume.source.StressSource
9 a1.sources.r1.maxEventsPerSecond= 1          # 限制测试的 event 生成速度，让它每
10 秒生成 1 个，便于观察效果
11 a1.sources.r1.batchSize = 1                  # 每次事务向 Channel 写入 1
12 个 event
13 a1.sources.r1.maxTotalEvents = 100         # 总共会生成 100 个 event

```

```

14 a1.sources.r1.channels = c1
15
16 # 两个 Sink 都是 File Roll Sink, 用于把 event 存储到本地文件中
17 a1.sinks.k1.type = file_roll
18 a1.sinks.k1.sink.batchSize = 1
19 # 每次事务从 Channel
20 获取 1 个 event
21 a1.sinks.k1.sink.directory = /Users/liyifeng/testflumek1      # 存储 event 的目录
22 a1.sinks.k1.channel = c1
23
24 # 同上
25 a1.sinks.k2.type = file_roll
26 a1.sinks.k2.sink.batchSize = 1
27 a1.sinks.k2.sink.directory = /Users/liyifeng/testflumek2
28 a1.sinks.k2.channel = c1
29
30 # 用的是内存 Channel, 没什么可说的
31 a1.channels.c1.type = memory
32 a1.channels.c1.capacity = 2
33 a1.channels.c1.transactionCapacity = 1
34
35 # 重点来了, 将两个 Sink 放在一个组 g1 中
36 a1.sinkgroups = g1
37 a1.sinkgroups.g1.sinks = k1 k2
    a1.sinkgroups.g1.processor.type = failover      # 该组的工作方式是故障转移
    # 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
1 $ 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
1 $ 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
1 $ wc -l /Users/liyifeng/testflumek1/*
2 $ 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:

```
1 a1.sinks = k1
2 a1.sinks.k1.type = file_roll
3 a1.sinks.k1.channel = c1
4 a1.sinks.k1.sink.directory = /var/log/flume
5 a1.sinks.k1.sink.serializer = text
6 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:

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

```
5 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
<code>syncIntervalBytes</code>	2048000	Avro 同步间隔，大约的字节数。
<code>compressionCodec</code>	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。
<code>schemaURL</code>	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：

```
1 a1.sinks.k1.type = hdfs
2 a1.sinks.k1.channel = c1
3 a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M%S
4 a1.sinks.k1.serializer =
5 org.apache.flume.sink.hdfs.AvroEventSerializer$Builder
```

```
6 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  
1 a1.sinks = k1  
2 a1.channels = c1  
3 a1.sources.r1.interceptors = i1 i2  
4 a1.sources.r1.interceptors.i1.type =  
5 org.apache.flume.interceptor.HostInterceptor$Builder  
6 a1.sources.r1.interceptors.i1.preserveExisting = false  
7 a1.sources.r1.interceptors.i1.hostHeader = hostname  
8 a1.sources.r1.interceptors.i2.type =  
9 org.apache.flume.interceptor.TimestampInterceptor$Builder  
10 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
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：

```
1 a1.sources = r1  
2 a1.channels = c1  
3 a1.sources.r1.channels = c1  
4 a1.sources.r1.type = seq  
5 a1.sources.r1.interceptors = i1  
6 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：

```
1 a1.sources = r1
2 a1.channels = c1
3 a1.sources.r1.interceptors = i1
4 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

属性	默认值	解释
key	key	写入 header 的 key
value	value	写入 header 的值

配置范例：

```

1 a1.sources = r1

2 a1.channels = c1

3 a1.sources.r1.channels =      c1

4 a1.sources.r1.type = seq

5 a1.sources.r1.interceptors = i1

6 a1.sources.r1.interceptors.i1.type = static

7 a1.sources.r1.interceptors.i1.key = datacenter

8 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 <i>fromListSeparator</i> 指定的分隔符分开
fromListSeparator	\s*, \s*	用来分隔 <i>fromList</i> 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

属性	默认值	解释
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
type	-	组件类型，这个是： <code>org.apache.flume.sink.solr.morphline.MorphlineInterceptor\$Builder</code>

属性	默认值	解释
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：

```

1 a1.sources.avroSrc.interceptors = morphlineinterceptor
2
3   a1.sources.avroSrc.interceptors.morphlineinterceptor.type =
4     org.apache.flume.sink.solr.morphline.MorphlineInterceptor$Builder
5
6   3 a1.sources.avroSrc.interceptors.morphlineinterceptor.morphlineFile =
7     /etc/flume-ng/conf/morphline.conf
8
9   4 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

属性	默认值	解释
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:

```

1 a1.sources.avroSrc.interceptors = search-replace
2   a1.sources.avroSrc.interceptors.search-replace.type = search_replace
3
4   # Remove leading alphanumeric characters in an event body.
5   a1.sources.avroSrc.interceptors.search-replace.searchPattern = ^[A-Za-z0-
6     9_]+

```

再来一个例子:

```

1 a1.sources.avroSrc.interceptors = search-replace
2   a1.sources.avroSrc.interceptors.search-replace.type = search_replace
3
4   # Use grouping operators to reorder and munge words on a line.

```

```
5 a1.sources.avroSrc.interceptors.search-replace.searchPattern = The quick brown  
([a-z]+) jumped over the lazy ([a-z]+)
```

6

```
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

属性	默认值	解释
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： <code>org.apache.flume.interceptor.RegexExtractorInterceptorPassThroughSerializer</code> <code>org.apache.flume.interceptor.RegexExtractorInterceptorMillisSerializer</code>
serializers.<s1>.type	default	可选值： 1: default (default 其实就是这个： <code>org.apache.flume.interceptor.RegexExtractorInterceptorPassThroughSerializer</code>) ; 2: <code>org.apache.flume.interceptor.RegexExtractorInterceptorMillisSerializer</code> ; 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 <code>org.apache.flume.interceptor.RegexExtractorInterceptorSerializer</code> 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”

```

1 a1.sources.r1.interceptors.i1.regex = (\d):(\d):(\d)
2 a1.sources.r1.interceptors.i1.serializers = s1 s2 s3
3 a1.sources.r1.interceptors.i1.serializers.s1.name = one
4 a1.sources.r1.interceptors.i1.serializers.s2.name = two
5 a1.sources.r1.interceptors.i1.serializers.s3.name = three

```

经过这个拦截器后，此时 Event:

```
1 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

```
1 al.sources.r1.interceptors.il.regex = (\d):(\d):(\d)
2 al.sources.r1.interceptors.il.serializers = s1 s2
3 al.sources.r1.interceptors.il.serializers.s1.name = one
4 al.sources.r1.interceptors.il.serializers.s2.name = two
```

执行这个拦截器后，此时 Event:

1 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

运行拦截器后，此时 Event:

1 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：

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。

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。

```
1      <agent_name>.configfilters = <filter_name>
2      <agent_name>.configfilters.<filter_name>.type = <filter_type>
3
4      <agent_name>.sources.<source_name>.parameter =
5          ${<filter_name>['<key_for_sensitive_or_generated_data>']}
6          <agent_name>.sinks.<sink_name>.parameter =
7          ${<filter_name>['<key_for_sensitive_or_generated_data>']}
8          <agent_name>.<component_type>.<component_name>.parameter =
9          ${<filter_name>['<key_for_sensitive_or_generated_data>']}
```

```

10          #or
11      <agent_name>.〈component_type〉.〈component_name〉.parameter =
12      ${<filter_name>["〈key_for_sensitive_or_generated_data〉"]}
           #or
           <agent_name>.〈component_type〉.〈component_name〉.parameter =
           ${<filter_name>[〈key_for_sensitive_or_generated_data〉]}
           #or
           <agent_name>.〈component_type〉.〈component_name〉.parameter =
           some_constant_data${<filter_name>[〈key_for_sensitive_or_generated_data〉]}

```

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。

```

1           a1.sources = r1
2           a1.channels = c1
3
4   a1.configfilters = f1                      # 这里给配置加载器命名为 f1
5
6
7           a1.sources.r1.channels =     c1
8           a1.sources.r1.type = http
9 a1.sources.r1.keystorePassword = ${f1['my_keystore_password']} # 启动 Flume 时如
               果配置了 my_keystore_password=Secret123，这里就能读取到密码了
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
<code>type</code>	-	组件类型，这里只能填 <code>external</code>
<code>command</code>	-	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 <code><command> <key></code> ，它期望的返回结果是个单行数值并且该脚本最后 <code>exit 0</code> 。
<code>charset</code>	<code>UTF-8</code>	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。

```

1           a1.sources = r1
2           a1.channels = c1
3           a1.configfilters = f1
4
5           a1.configfilters.f1.type = external
6           a1.configfilters.f1.command = /usr/bin/passwordResolver.sh      # 外部脚本的
7                           绝对路径
8           a1.configfilters.f1.charset = UTF-8
9
10          a1.sources.r1.channels = c1
11          a1.sources.r1.type = http
12 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。

```
1           a1.sources = r1
2           a1.channels = c1
3           a1.configfilters = f1
4
5           a1.configfilters.f1.type = external
6           a1.configfilters.f1.command = /usr/bin/generateUniqId.sh
7           a1.configfilters.f1.charset = UTF-8
8
9           a1.sinks = k1
10          a1.sinks.k1.type = file_roll
11          a1.sinks.k1.channel = c1
12 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

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
credential.provider.path	-	provider 的路径，参考 hadoop 的文档： https://hadoop.apache.org/docs/stable/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
1 a1.channels = c1
2 a1.configfilters = f1
3
4 a1.configfilters.f1.type = hadoop
5 a1.configfilters.f1.credential.provider.path    =      jceks://file/<path_to_jceks
6 file>
7
8 a1.sources.r1.channels =      c1
9 a1.sources.r1.type = http
10 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

属性	默认值	解释
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:

```

1 <em>#...</em>
2 log4j.appenders.flume =
2 org.apache.flume.clients.log4jappender.Log4jAppender
3
3 log4j.appenders.flume.Hostname = example.com
4
4 log4j.appenders.flume.Port = 41414
5
5 log4j.appenders.flume.UnsafeMode = true
6
7 # 指定一个类的 logger 输出到 Flume appender 上
8 log4j.logger.org.example.MyClass = DEBUG, flume
9 #...

```

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：

```
1 <em>#...</em>
2 log4j.appenders.flume =
3   org.apache.flume.clients.log4jappender.Log4jAppender
4
5   log4j.appenders.flume.Hostname = example.com
6
7   log4j.appenders.flume.Port = 41414
8
9   log4j.appenders.flume.AvroReflectionEnabled = true
10
11  log4j.appenders.flume.AvroSchemaUrl = hdfs://namenode/path/to/schema.avsc
12
13  # 指定一个类的 logger 输出到 flume appender 上
14
15  log4j.logger.org.example.MyClass = DEBUG, flume
16
17  #...
```

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
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 <i>LoadBalancingSelector</i>)
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：

```
1 <em>#...</em>
2 log4j.appenders.out2 =
3   org.apache.flume.clients.log4jappender.LoadBalancingLog4jAppender
4   log4j.appenders.out2.Hosts = localhost:25430 localhost:25431
```

5

6 # configure a class's logger to output to the flume appender

7 log4j.logger.org.example.MyClass = DEBUG,flume

#...

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

1 #...

2 log4j.appenders.out2 =
org.apache.flume.clients.log4jappender.LoadBalancingLog4jAppender

3 log4j.appenders.out2.Hosts = localhost:25430 localhost:25431

4 log4j.appenders.out2.Selector = RANDOM

5

6 # configure a class's logger to output to the flume appender

7 log4j.logger.org.example.MyClass = DEBUG,flume

8 #...

log4j 使用「失败退避」方式的 log4j.properties 配置范例:

1 #...

2 log4j.appenders.out2 =
org.apache.flume.clients.log4jappender.LoadBalancingLog4jAppender
3

4 log4j.appenders.out2.Hosts = localhost:25430 localhost:25431 localhost:25432

```
5 log4j.appenders.out2.Selector = ROUND_ROBIN  
6 log4j.appenders.out2.MaxBackoff = 30000      #最大的退避时长是 30 秒  
  
7  
  
8 # configure a class's logger to output to the flume appender  
  
9 log4j.logger.org.example.MyClass = DEBUG, flume  
  
#...
```

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	Exe c	HTT P	JM S	Kafk a	MultiportSyslog TCP	Scrib e
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

GenericProcessingFailure			x		x		
KafkaCommitTimer				x			
KafkaEmptyCount				x			
KafkaEventGetTimer				x			
OpenConnectionCount	x						

Sources 2

	SequenceGenerator	SpoolDirectory	Syslog Tcp	Syslog UDP	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	HBase 2

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

	Fil e	Kafk a	Memor y	PseudoTxnMem ory	SpillableMem ory
ChannelCapacity	x		x		x
ChannelSize	x		x	x	x
CheckpointBackupWriteError Count	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：

```
1 export JAVA_OPTS="-Dcom.sun.management.jmxremote -  
Dcom.sun.management.jmxremote.port=5445 -  
Dcom.sun.management.jmxremote.authenticate=false -  
Dcom.sun.management.jmxremote.ssl=false"
```

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
<code>type</code>	-	组件类型，这个是： ganglia
<code>hosts</code>	-	hostname:port 格式的 Ganglia 服务列表，多个用逗号分隔
<code>pollFrequency</code>	60	向 Ganglia 服务器报告数据的时间间隔（秒）
<code>isGanglia3</code>	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.example:1234,com.example2:5455  
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：

```
1 {  
2 "typeName1.componentName1": {"metric1": "metricValue1", "metric2": "metricValue2"},  
3 "typeName2.componentName2": {"metric3": "metricValue3", "metric4": "metricValue4"}
```

```
4 }
```

下面是一个具体的报告例子：

```
1 {
2   "CHANNEL.fileChannel": {"EventPutSuccessCount": "468085",
3     "Type": "CHANNEL",
4     "StopTime": "0",
5     "EventPutAttemptCount": "468086",
6     "ChannelSize": "233428",
7     "StartTime": "1344882233070",
8     "EventTakeSuccessCount": "458200",
9     "ChannelCapacity": "600000",
10    "EventTakeAttemptCount": "458288"},
11  "CHANNEL.memChannel": {"EventPutSuccessCount": "22948908",
12    "Type": "CHANNEL",
13    "StopTime": "0",
14    "EventPutAttemptCount": "22948908",
15    "ChannelSize": "5",
16    "StartTime": "1344882209413",
17    "EventTakeSuccessCount": "22948900",
18    "ChannelCapacity": "100",
19    "EventTakeAttemptCount": "22948908"}
20 }
```

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：

```
1 $ 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 `https/<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 -  
1 Dflume.monitoring.type=com.example.reporting.HTTPReporting -  
Dflume.monitoring.node=com.example:332
```

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。

```
1 <strong>public</strong> <strong>class</strong> <strong>SinkCounter</strong> <strong>extends</strong> MonitoredCounterGroup  
2 <strong>implements</strong>  
3   SinkCounterMBean {  
4  
5     <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_CONNECTION_CREATED =  
6     "sink.connection.creation.count";  
7  
8     <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_CONNECTION_CLOSED =  
9     "sink.connection.closed.count";  
10  
11    <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_CONNECTION_FAILED =  
12    "sink.connection.failed.count";  
13  
14    <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_BATCH_EMPTY =
```

```
15 "sink.batch.empty";
16
17 <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_BATCH_UNDERFLOW =
18 "sink.batch.underflow";
19
20 <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_BATCH_COMPLETE =
21 "sink.batch.complete";
22
23 <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_EVENT_DRAIN_ATTEMPT =
24 "sink.event.drain.attempt";
25
26 <strong>private</strong> <strong>static</strong> <strong>final</strong> String COUNTER_EVENT_DRAIN_SUCCESS =
27 "sink.event.drain.sucess";
28
29 <strong>private</strong> <strong>static</strong> <strong>final</strong> String[] ATTRIBUTES = {
30 COUNTER_CONNECTION_CREATED, COUNTER_CONNECTION_CLOSED,
31 COUNTER_CONNECTION_FAILED, COUNTER_BATCH_EMPTY,
32 COUNTER_BATCH_UNDERFLOW, COUNTER_BATCH_COMPLETE,
33 COUNTER_EVENT_DRAIN_ATTEMPT, COUNTER_EVENT_DRAIN_SUCCESS
34 };
35
36 <strong>public</strong> SinkCounter(String name) {
37 <strong>super</strong>(MonitoredCounterGroup.Type.SINK, name, ATTRIBUTES);
38 }
39
40 <strong>@Override</strong>
41 <strong>public</strong> long getConnectionCreatedCount() {
42 <strong>return</strong> get(COUNTER_CONNECTION_CREATED);
43 }
44
45 <strong>public</strong> long incrementConnectionCreatedCount() {
46 <strong>return</strong> increment(COUNTER_CONNECTION_CREATED);
47 }
48
}
```

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：

```
1 $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。

以下是可以选的参数

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 EventValidator 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。

```
1 <strong>public</strong> <strong>static</strong> <strong>class</strong> <strong>MyEventValidator</strong> <strong>implements</strong> EventValidator {  
2  
3   <strong>private</strong> int value = 0;  
4  
5   <strong>private</strong> MyEventValidator(int val) {  
6     value = val;  
7   }  
8  
9   <strong>@Override</strong>  
10  <strong>public</strong> boolean validateEvent(Event event) {  
11    <strong>return</strong> event.getBody() <= value;  
12  }  
13  
14  <strong>public</strong> <strong>static</strong> <strong>class</strong> <strong>Builder</strong> <strong>implements</strong> EventValidator.Builder {  
15  
16    <strong>private</strong> int sizeValidator = 0;  
17  
18    <strong>@Override</strong>  
19    <strong>public</strong> EventValidator build() {  
20      <strong>return</strong> <strong>new</strong> DummyEventVerifier(sizeValidator);  
21    }  
22  }  
23  
24  <strong>class</strong> DummyEventVerifier {  
25    <strong>private</strong> int sizeValidator;  
26  
27    <strong>public</strong> DummyEventVerifier(int sizeValidator) {  
28      this.sizeValidator = sizeValidator;  
29    }  
30  
31    <strong>public</strong> boolean validate(Event event) {  
32      <strong>return</strong> event.getBody() <= sizeValidator;  
33    }  
34  }
```

```
22
23 <strong>@Override</strong>
24 <strong>public</strong> void configure(Context context) {
25     binaryValidator = context.getInteger("maxSize");
26 }
27 }
28 }
```

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，适合于那种例行性、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。

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。

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。

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

组件接口	别名	实现类
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
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

组件接口	别名	实现类
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

组件接口	别名	实现类
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
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

组件接口	别名	实现类
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

组件接口	别名	实现类
org.apache.flume.channel.file. .encryption.KeyProvider\$Builder	jceksfile	org.apache.flume.channel.file.en cryption.JCEFileKeyProvider
org.apache.flume.channel.file. .encryption.KeyProvider\$Builder	-	org.example.MyKeyProvider
org.apache.flume.channel.file. .encryption.CipherProvider	aesctrnop adding	org.apache.flume.channel.file.en cryption.AESCTRNoPaddingProvider
org.apache.flume.channel.file. .encryption.CipherProvider	-	org.example.MyCipherProvider
org.apache.flume.serialization. n.EventSerializer\$Builder	text	org.apache.flume.serialization.B odyTextEventSerializer\$Builder
org.apache.flume.serialization. n.EventSerializer\$Builder	avro_event	org.apache.flume.serialization.F lumeEventAvroEventSerializer\$Bui lder
org.apache.flume.serialization. n.EventSerializer\$Builder	-	org.example.MyEventSerializer\$Bu ilder

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
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

配置 FLUME SOURCE

安装 netcat

Netcat 是一款简单的 Unix 工具，简称 nc，安全界叫它瑞士军刀，使用 UDP 和 TCP 协议。它是一个可靠的容易被其他程序所启用的后台操作工具，同时它也被用作网络的测试工具或黑客工具。 使用它你可以轻易的建立任何连接。内建有很多实用的工具。

```
$> sudo yum install nmap-ncat.x86_64
```

nc 的一些用法：

端口测试

检测主机上 8080 端口服务是否开放

```
telnet 192.168.1.2 8080
```

或者

```
nc -vz 192.168.1.2 8080
```

z 表示不发送数据, v 表示显示额外信息

nc 命令后面的 8080 可以写成一个范围进行扫描：

```
nc -v -v -w3 -z 192.168.1.2 8080-8083
```

两次 -v 是让它报告更详细的内容， -w3 是设置扫描超时时间为 3 秒。

传输测试

A 主机上监听了 8080 端口

```
nc -l -p 8080
```

然后在 B 主机上连接过去：

```
nc 192.168.1.2 8080
```

两边就可以会话了，随便输入点什么按回车，另外一边应该会显示出来。

netcat source

配置

[hello.conf]

```
1 #声明三种组件
2 a1.sources = r1
3 a1.channels = c1
4 a1.sinks = k1
5
6 #定义 source 信息
7 a1.sources.r1.type=netcat
8 a1.sources.r1.bind=localhost
9 a1.sources.r1.port=8888
10
11 #定义 sink 信息
12 a1.sinks.k1.type=logger
13
14 #定义 channel 信息
15 a1.channels.c1.type=memory
16
17 #绑定在一起
18 a1.sources.r1.channels=c1
19 a1.sinks.k1.channel=c1
```

运行

1. 启动 flume agent\$> bin/flume-ng agent -f conf/hello.conf -n a1 –Dflume.root.logger=INFO,console
2. 启动 nc 的客户端\$> nc localhost 8888 \$nc> hello world
3. 在 Flume 的终端输出 hello world.

exec source

4. 实时日志收集, 实时收集日志。

5. 1 17. a1.sources = r1
6. 2 18. a1.sinks = k1
7. 3 19. a1.channels = c1

```
8. 4      20.  
9. 5      21. a1.sources.r1.type=exec  
10. 6     22. a1.sources.r1.command=tail -F /home/centos/test.txt  
11. 7     23.  
12. 8     24. a1.sinks.k1.type=logger  
13. 9     25. a1.channels.c1.type=memory  
14. 10    26.  
15. 11    27. a1.sources.r1.channels=c1  
16. 12    28. a1.sinks.k1.channel=c1
```

spooldir 源

29. 监控一个文件夹，静态文件，批量收集。
收集完之后，会重命名文件成新文件.COMPLETED.

配置文件

[spooldir_r.conf]

```
1 a1.sources = r1  
2 a1.channels = c1  
3 a1.sinks = k1  
4  
5 a1.sources.r1.type=spooldir  
6 a1.sources.r1.spoolDir=/home/centos/spool  
7 a1.sources.r1.fileHeader=true  
8  
9 a1.sinks.k1.type=logger  
10  
11 a1.channels.c1.type=memory  
12  
13 a1.sources.r1.channels=c1  
14 a1.sinks.k1.channel=c1
```

```
1 $>mkdir ~/spool
```

启动 flume

```
$>bin/flume-ng agent -f ../conf/helloworld.conf -n a1 -  
Dflume.root.logger=INFO,console
```

seq source

生成事件序列的源，一般用于测试。

[seq]

```
1 a1.sources = r1  
2 a1.channels = c1  
3 a1.sinks = k1  
4  
5 a1.sources.r1.type=seq  
6 a1.sources.r1.totalEvents=1000  
7  
8 a1.sinks.k1.type=logger  
9  
10 a1.channels.c1.type=memory  
11  
12 a1.sources.r1.channels=c1  
13 a1.sinks.k1.channel=c1
```

[运行]

```
$>bin/flume-ng agent -f ../conf/helloworld.conf -n a1 -  
Dflume.root.logger=INFO,console
```

Stress Source

用于压力测试的源。

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

TailDir Source

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。

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。

练习

使用 Flume 监听整个目录的实时追加文件，并上传至 HDFS

```
1 #步骤一: agent Name
2 a1.sources = r1
3 a1.sinks = k1
4 a1.channels = c1
5
6 #步骤二: source
7 # Describe/configure the source
8 a1.sources.r1.type = TAILDIR
9 a1.sources.r1.positionFile = /opt/module/flume/tail_dir.json -- 指定
10 position_file 的位置(记录每次上传后的偏移量，实现断点续传的关键)
11 a1.sources.r1.filegroups = f1 f2 -- 监控的文件目录集合
12 a1.sources.r1.filegroups.f1 = /opt/module/flume/files/.*file.* -- 定义监控的文
13 件目录 1
14 a1.sources.r1.filegroups.f2 = /opt/module/flume/files/.*log.* -- 定义监控的文件
15 目录 2
16
17 #步骤三: channel selector
18 a1.sources.r1.selector.type = replicating
19
20 #步骤四: channel
21 # Describe the channel
22 a1.channels.c1.type = memory
23 a1.channels.c1.capacity = 1000
24 a1.channels.c1.transactionCapacity = 100
25
26 #步骤五: sinkprocessor, 默认配置 defaultsinkprocessor
27 #步骤六: sink
28 a1.sinks.k1.type = hdfs
29 a1.sinks.k1.hdfs.path = hdfs://hadoop102:9820/flume/upload3/%Y%m%d/%H
30
31 #上传文件的前缀
32 a1.sinks.k1.hdfs.filePrefix = upload-
33 #是否按照时间滚动文件夹
34 a1.sinks.k1.hdfs.round = true
35 #多少时间单位创建一个新的文件夹
36 a1.sinks.k1.hdfs.roundValue = 1
37 #重新定义时间单位
38 a1.sinks.k1.hdfs.roundUnit = hour
```

```
39 #是否使用本地时间戳
40 a1.sinks.k1.hdfs.useLocalTimeStamp = true
41 #积攒多少个 Event 才 flush 到 HDFS 一次
42 a1.sinks.k1.hdfs.batchSize = 100
43 #设置文件类型，可支持压缩
44 a1.sinks.k1.hdfs.fileType = DataStream
45 #多久生成一个新的文件
46 a1.sinks.k1.hdfs.rollInterval = 60
47 #设置每个文件的滚动大小大概是 128M
48 a1.sinks.k1.hdfs.rollSize = 134217700
49 #文件的滚动与 Event 数量无关
50 a1.sinks.k1.hdfs.rollCount = 0
#步骤七：连接 source、channel、sink
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

FlumeSinks

Flume Sinks 类型有很多，这里只挑出一些我们常用的 Sink.

1. HDFS Sink
2. Hive Sink
3. Logger Sink
4. Avro Sink
5. HBase Sinks
6. Kafka Sink
7. HTTP Sink
8. File Roll Sink
9. NULL sink
10. Custom SInk

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，%[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。

属性名	默认值	解释
channel	-	与 Sink 连接的 channel

属性名	默认值	解释
type	-	组件类型，这个是： hdfs
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

属性名	默认值	解释
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`
hdfs.fileType	SequenceFile	文件格式，目前支持： <code>SequenceFile</code> 、 <code>DataStream</code> 、 <code>CompressedStream</code> 。 1. <code>DataStream</code> 不会压缩文件，不需要设置 <code>hdfs.codec</code> 2. <code>CompressedStream</code> 必须设置 <code>hdfs.codec</code> 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： <code>Text</code> 、 <code>Writable</code> 。在使用 Flume 创建数据文件之前设置为 <code>Text</code> ，否则 Apache Impala (孵化) 或 Apache Hive 无法读取这些文件。

属性名	默认值	解释
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 <code>hdfs.roundUnit</code> ） 例子：假设当前时间戳是 18:32:01， <code>hdfs.roundUnit = minute</code> 如果 <code>roundValue=5</code> , 则时间戳会取为：18:30 如果 <code>roundValue=7</code> , 则时间戳会取为：18:28 如果 <code>roundValue=10</code> , 则时间戳会取为：18:30
hdfs. roundUnit	second	向下舍入的单位，可选值： second 、 minute 、 hour

属性名	默认值	解释
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：

```

1 a1.channels = c1

2 a1.sinks = k1

3 a1.sinks.k1.type = hdfs

4 a1.sinks.k1.channel = c1

5 a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M%S

6 a1.sinks.k1.hdfs.filePrefix = events-

7 a1.sinks.k1.hdfs.round = true

8 a1.sinks.k1.hdfs.roundValue = 10

9 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
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: string, 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 <i>batchSize</i> 个 Event, 这个参数和 <i>batchSize</i> 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。

属性	默认值	解释
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
round	false	是否启用时间截舍入机制
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 <code>roundUnit</code> 配置的单位）例子 1： <code>roundUnit=second</code> , <code>roundValue=10</code> , 则 14:31:18 这个时间截会被舍入到 14:31:10; 例子 2: <code>roundUnit=second</code> , <code>roundValue=30</code> , 则

属性	默认值	解释
		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”
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 <code>serializer.fieldnames</code> 中的字段与 Hive 表列的顺序相同，

属性	默认值	解释
		则 <code>serializer.delimiter</code> 与 <code>serializer.serdeSeparator</code> 相同，并且 <code>serializer.fieldnames</code> 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 <code>serializer.delimiter</code> 是单个字符，最好将本参数也设置为相同的字符。

为了避免出现找不到类的异常，首先需添加依赖的 jar 包：

将 hive/lib 下面 `hive-hcatalog-core-3.1.2.jar` 拷贝或者软链接到 flume/lib 下

```
$ cp /opt/pkg/hive/lib/hive-hcatalog-core-3.1.2.jar /opt/pkg/flume/lib/
```

接下来我们还需要这个依赖：`hive-hcatalog-streaming-3.1.2.jar`。这个 `hive` 目录是找不到的，需要单独从 mvnrepository 网站搜索下载，下载地址：

<https://mvnrepository.com/artifact/org.apache.hive.hcatalog/hive-hcatalog-streaming/3.1.2>，下载后移动到 flume/lib 下即可。

使用 Hive Sink 还有以下前提条件：

需要开启事务支持

Hive 表必须分区分桶

Hive 表必须是 Acid 表（开启事务支持）

进入 beeline 临时开启事务支持（也可以修改配置文件永久开启，但不建议）

```
3 SET hive.enforce.bucketing = true;  
4 SET hive.exec.dynamic.partition.mode = nonstrict;  
5 SET hive.txn.manager = org.apache.hadoop.hive ql.lockmgr.DbTxnManager;  
6 SET hive.compactor.initiator.on = true;  
7 SET hive.compactor.worker.threads = 1;
```

创建 Hive 表如下：

```
1 create database logsdb;  
2 use logsdb;  
3  
4 create table weblogs ( id int , msg string )  
5  
6 partitioned by (continent string, contry string)  
7 clustered by (id) into 5 buckets  
8 stored as orc  
9 tblproperties(  
10     'transactional'='true',  
11     'transactional_properties'='default'  
12 );
```

Flume Agnet 配置范例：

```
1 $ vi conf/hello-hive.conf  
2  
3 #声明三种组件  
4 a1.sources = r1  
5 a1.channels = c1  
6 a1.sinks = k1  
7  
8 #定义 source 信息  
9 a1.sources.r1.type=netcat  
10 a1.sources.r1.bind=localhost  
11 a1.sources.r1.port=8888  
12  
13 #定义 sink 信息  
14 a1.sinks.k1.type = hive  
15 a1.sinks.k1.hive.metastore = thrift://hadoop100:9083  
16 a1.sinks.k1.hive.database = logsdb  
17 a1.sinks.k1.hive.table = weblogs  
18 a1.sinks.k1.hive.partition = asia, India
```

```
19 a1.sinks.k1.useLocalTimeStamp = false  
20 a1.sinks.k1.batchSize = 50  
21 a1.sinks.k1.round = true  
22 a1.sinks.k1.roundValue = 10  
23 a1.sinks.k1.roundUnit = minute  
24 a1.sinks.k1.serializer = DELIMITED  
25 a1.sinks.k1.serializer.delimiter = "\t"  
26 a1.sinks.k1.serializer.serdeSeparator = '\t'  
27 a1.sinks.k1.serializer.fieldnames =id,msg  
28  
29 #定义 channel 信息  
30 a1.channels.c1.type=memory  
31  
32 #绑定在一起  
33 a1.sources.r1.channels=c1  
34 a1.sinks.k1.channel=c1
```

启动 Flume Agent

```
flume]$ flume-ng agent -c conf/ -f conf/hello-hive.conf -n a1 -  
1 Dflume.root.logger=INFO,console
```

使用 nc 客户端发送数据

```
1 [hadoop@hadoop100 ~]$ nc localhost 8888
```

```
2 1001      hello
```

```
3 OK
```

```
4 1002      world
```

```
5 OK
```

查看 hive 表中是否有数据出现

```
0: jdbc:hive2://localhost:10000> select * from weblogs;
```

```
1
```

```
+-----+-----+-----+-----+
```

```
2
```

```
| weblogs.id    | weblogs.msg     | weblogs.continent   |
 | weblogs.contry|
```

```
3
```

```
4
```

```
+-----+-----+-----+-----+
| 1002          | world        | India           |
| asia          |             |
```

```
5
```

```
+-----+-----+-----+-----+
| 1001          | hello        | India           |
| asia          |             |
```

```
6
```

```
7
```

```
+-----+-----+-----+-----+
```

```
8
```

```
2 rows selected (5.394 seconds)
```

```
9
```

如果对于行级更新删除需求比较频繁的，可以考虑使用事务表，但平常的 hive 表并不建议使用事务表。因为事务表的限制很多，加上由于 hive 表的特性，也很难满足高并发的场

景。另外，如果事务表太多，并且存在大量的更新操作，metastore 后台启动的合并线程会定期的提交 MapReduce Job，也会一定程度上增重集群的负担。

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
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：

```
1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = logger
```

```
4 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
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

属性	默认值	解释
		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）

属性	默认值	解释
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。
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：

```

1 a1.channels = c1

2 a1.sinks = k1

3 a1.sinks.k1.type = avro

4 a1.sinks.k1.channel = c1

5 a1.sinks.k1.hostname = 10.10.10.10

6 a1.sinks.k1.port = 4545

```

利用 AvroSource 和 AvroSink 实现跃点 Agent

配置范例 - Agent #a1:

```
1 #a1
2 a1.sources = r1
3 a1.sinks= k1
4 a1.channels = c1
5
6 a1.sources.r1.type=netcat
7 a1.sources.r1.bind=localhost
8 a1.sources.r1.port=8888
9
10 a1.sinks.k1.type = avro
11 a1.sinks.k1.hostname=localhost
12 a1.sinks.k1.port=9999
13
14 a1.channels.c1.type=memory
15
16 a1.sources.r1.channels = c1
17 a1.sinks.k1.channel = c1
18
```

配置范例 - Agent #a2:

```
1 #a2
2 a2.sources = r2
3 a2.sinks= k2
4 a2.channels = c2
5
6 a2.sources.r2.type=avro
7 a2.sources.r2.bind=localhost
8 a2.sources.r2.port=9999
9
10 a2.sinks.k2.type = logger
11
12 a2.channels.c2.type=memory
13
14 a2.sources.r2.channels = c2
15 a2.sinks.k2.channel = c2
```

启动 Agent #a2

```
$ flume-ng agent -f /soft/flume/conf/avro_hop.conf -n a2 -
1 Dflume.root.logger=INFO,console
```

验证 Agent #a2

```
1 $ netstat -anop | grep 9999
```

启动 Agent #a1

```
1 $ flume-ng agent -f /soft/flume/conf/avro_hop.conf -n a1
```

验证 Agent #a1

```
1 $ netstat -anop | grep 8888
```

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
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 <i>hbase.zookeeper.quorum</i> 的值

属性	默认值	解释
znodeParent	/hbase	ZooKeeper 中 HBase 的 Root ZNode 路径, hbase-site.xml 中 <code>zookeeper.znode.parent</code> 的值
batchSize	100	每个事务写入的 Event 数量
coalesceIncrments	false	每次提交时, Sink 是否合并多个 increment 到一个 cell。如果有有限数量的 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：

```

1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = hbase2
4 a1.sinks.k1.table = foo_table
5 a1.sinks.k1.columnFamily = bar_cf
6 a1.sinks.k1.serializer =
org.apache.flume.sink.hbase2.RegexHBase2EventSerializer
7 a1.sinks.k1.channel = c1

```

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： <code>org.apache.flume.sink.kafka.KafkaSink</code>
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

属性	默认值	解释
		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 <code>partitionIdHeader</code> 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 <code>kafka.partitionner.class</code> 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 <code>EventDeliveryException</code> 。如果存在标头值，则此设置将覆盖 <code>defaultPartitionId</code> 。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 <code>topicHeader</code> 的属性值，用它作为目标 topic。
topicHeader	topic	与上面的 <code>allowTopicOverride</code> 一起使用， <code>allowTopicOverride</code> 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。

属性	默认值	解释
<i>more producer security props</i>		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： <i>kafka.producer.linger.ms</i>

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

属性	默认值	解释
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。

```

1 a1.sinks.k1.channel = c1

2 a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink

3 a1.sinks.k1.kafka.topic = mytopic

4 a1.sinks.k1.kafka.bootstrap.servers = localhost:9092

5 a1.sinks.k1.kafka.flumeBatchSize = 20

6 a1.sinks.k1.kafka.producer.acks = 1

7 a1.sinks.k1.kafka.producer.linger.ms = 1

8 a1.sinks.k1.kafka.producer.compression.type = snappy

```

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
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 <i>backoff.CODE</i> 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 <i>rollback.CODE</i> 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 <i>incrementMetrics.CODE</i> 没有匹

属性	默认值	解释
		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：

```
1 a1.channels = c1
```

```
2 a1.sinks = k1

3 a1.sinks.k1.type = http

4 a1.sinks.k1.channel = c1

5 a1.sinks.k1.endpoint = http://localhost:8080/someuri

6 a1.sinks.k1.connectTimeout = 2000

7 a1.sinks.k1.requestTimeout = 2000

8 a1.sinks.k1.acceptHeader = application/json

9 a1.sinks.k1.contentTypeHeader = application/json

10 a1.sinks.k1.defaultBackoff = true

11 a1.sinks.k1.defaultRollback = true

12 a1.sinks.k1.defaultIncrementMetrics = false

13 a1.sinks.k1.backoff.4XX = false

14 a1.sinks.k1.rollback.4XX = false

15 a1.sinks.k1.incrementMetrics.4XX = true

16 a1.sinks.k1.backoff.200 = false

17 a1.sinks.k1.rollback.200 = false

18 a1.sinks.k1.incrementMetrics.200 = true
```

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.
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 <i>pathManager</i> 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 <i>pathManager</i> 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。

属性	默认值	解释
sink.serializer	TEXT	<p>配置 Event 序列化器，可选值有： text 、 header_and_text 、 avro_event 或者自定义实现</p> <p>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。</p>
sink.batchSize	100	每次事务批处理的 Event 数

配置范例：

```

1 a1.channels = c1

2 a1.sinks = k1

3 a1.sinks.k1.type = file_roll

4 a1.sinks.k1.channel = c1

5 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

属性	默认值	解释
type	-	组件类型, 这个是: null.
batchSize	100	每次批处理的 Event 数量

配置范例:

```

1 a1.channels = c1
2 a1.sinks = k1
3 a1.sinks.k1.type = null
4 a1.sinks.k1.channel = c1

```

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 channel
type	-	组件类型, 这个填你自定义 class 的全限定类名

配置范例:

1 a1. channels = c1

2 a1. sinks = k1

3 a1. sinks. k1. type = org.example.MySink

4 a1. sinks. k1. channel = c1

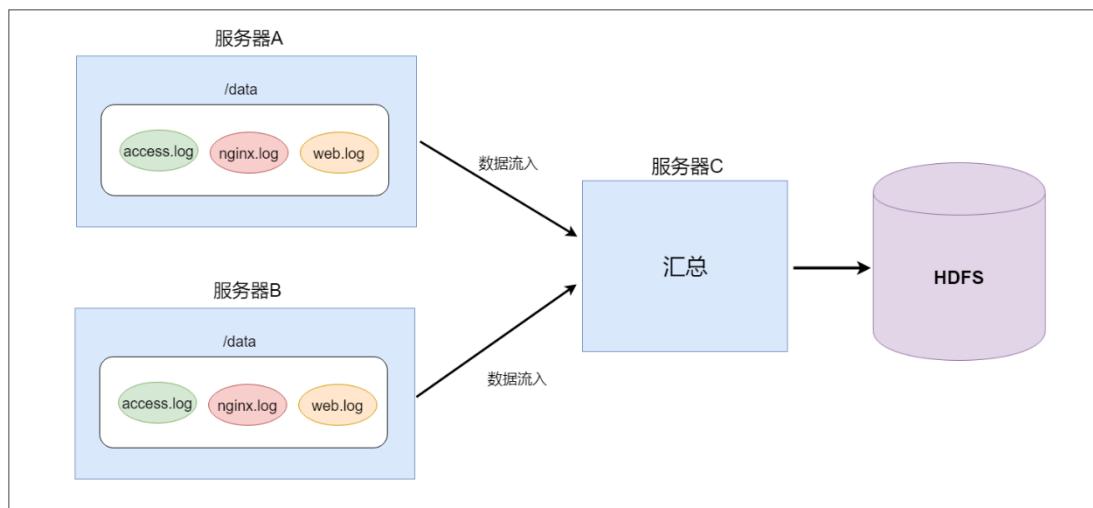
Flume 综合案例之拦截器

Flume 综合案例之静态拦截器使用

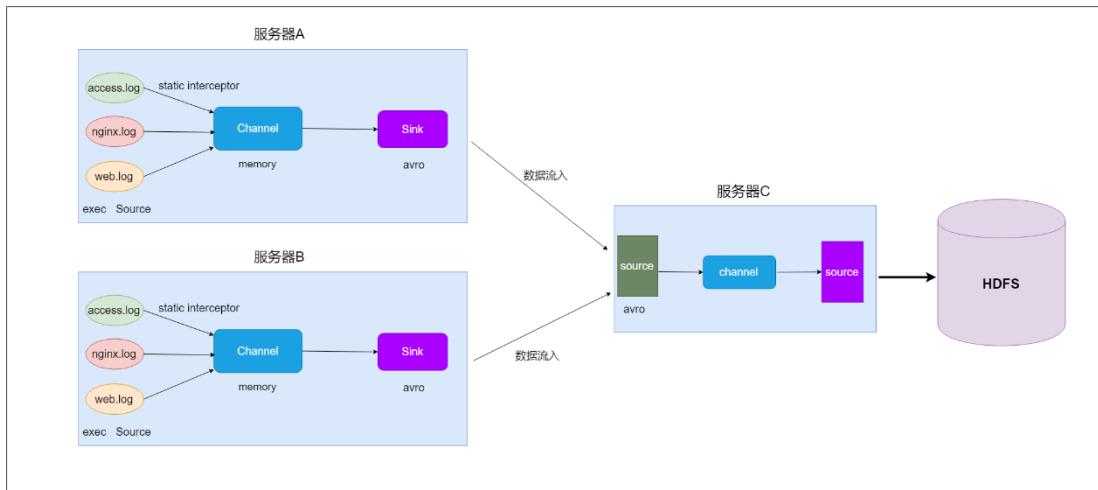
1. 案例场景

- A、B 两台日志服务机器实时生产日志主要类型为 access.log、nginx.log、web.log
- 现在需要把 A、B 机器中的 access.log、nginx.log、web.log 采集汇总到 C 机器上然后统一收集到 hdfs 中。
- 但是在 hdfs 中要求的目录为：
 - 1 • /source/logs/access/20200210/**
 - 2 • /source/logs/nginx/20200210/**
 - 3 • /source/logs/web/20200210/**

2. 场景分析



3. 数据流程处理分析



4. 实现

- 服务器 A 对应的 IP 为 192.168.52.100
- 服务器 B 对应的 IP 为 192.168.52.110
- 服务器 C 对应的 IP 为 192.168.52.120

采集端配置文件开发

- node01 与 node02 服务器开发 flume 的配置文件
 - 1 • cd /kkb/install/apache-flume-1.9.0-bin/conf/
 - 2 • vim exec_source_avro_sink.conf
- 内容如下
 - 1 • # Name the components on this agent
 - 2 • a1.sources = r1 r2 r3
 - 3 • a1.sinks = k1
 - 4 • a1.channels = c1

```
• 5   •
• 6   • # set source
• 7   • a1.sources.r1.type = exec
• 8   • a1.sources.r1.command = tail -F /kkb/install/taillogs/access.log
• 9   • a1.sources.r1.interceptors = i1
• 10  • a1.sources.r1.interceptors.i1.type = static
• 11  • ## static 拦截器的功能就是往采集到的数据的 header 中插入自定义
• 12  的 key-value 对；与 node03 上的 agent 的 sink 中的 type 相呼应
• 13  • a1.sources.r1.interceptors.i1.key = type
• 14  • a1.sources.r1.interceptors.i1.value = access
• 15  •
• 16  • a1.sources.r2.type = exec
• 17  • a1.sources.r2.command = tail -F /kkb/install/taillogs/nginx.log
• 18  • a1.sources.r2.interceptors = i2
• 19  • a1.sources.r2.interceptors.i2.type = static
• 20  • a1.sources.r2.interceptors.i2.key = type
• 21  • a1.sources.r2.interceptors.i2.value = nginx
• 22  •
• 23  • a1.sources.r3.type = exec
• 24  • a1.sources.r3.command = tail -F /kkb/install/taillogs/web.log
• 25  • a1.sources.r3.interceptors = i3
• 26  • a1.sources.r3.interceptors.i3.type = static
• 27  • a1.sources.r3.interceptors.i3.key = type
• 28  • a1.sources.r3.interceptors.i3.value = web
• 29  •
• 30  • # set sink
• 31  • a1.sinks.k1.type = avro
• 32  • a1.sinks.k1.hostname = node03
• 33  • a1.sinks.k1.port = 41415
• 34  •
• 35  • # set channel
• 36  • a1.channels.c1.type = memory
• 37  • a1.channels.c1.capacity = 20000
• 38  • a1.channels.c1.transactionCapacity = 10000
• 39  •
• 40  • # Bind the source and sink to the channel
• 41  • a1.sources.r1.channels = c1
• 42  • a1.sources.r2.channels = c1
• 43  • a1.sources.r3.channels = c1
•     • a1.sinks.k1.channel = c1
```

服务端配置文件开发

- 在 node03 上面开发 flume 配置文件
 - 1 • cd /kkb/install/apache-flume-1.9.0-bin/conf/
 - 2 • vim avro_source_hdfs_sink.conf

- 内容如下

```
1   • # Name the components on this agent
2   • a1.sources = r1
3   • a1.sinks = k1
4   • a1.channels = c1
5   •
6   • #定义 source
7   • a1.sources.r1.type = avro
8   • a1.sources.r1.bind = node03
9   • a1.sources.r1.port =41415
10  •
11  • #定义 channels
12  • a1.channels.c1.type = memory
13  • a1.channels.c1.capacity = 20000
14  • a1.channels.c1.transactionCapacity = 10000
15  •
16  • #定义 sink
17  • a1.sinks.k1.type = hdfs
18  • a1.sinks.k1.hdfs.path=hdfs://node01:8020/source/logs/%{type}/%Y%m%d
19  • a1.sinks.k1.hdfs.filePrefix =events
20  • a1.sinks.k1.hdfs.fileType = DataStream
21  • a1.sinks.k1.hdfs.writeFormat = Text
22  • #时间类型
23  • a1.sinks.k1.hdfs.useLocalTimeStamp = true
24  • #生成的文件不按条数生成
25  • a1.sinks.k1.hdfs.rollCount = 0
26  • #生成的文件按时间生成
27  • a1.sinks.k1.hdfs.rollInterval = 30
28  • #生成的文件按大小生成
29  • a1.sinks.k1.hdfs.rollSize      = 10485760
30  • #批量写入 hdfs 的个数
31  • a1.sinks.k1.hdfs.batchSize = 10000
32  • #flume 操作 hdfs 的线程数（包括新建，写入等）
33  • a1.sinks.k1.hdfs.threadsPoolSize=10
```

```
34   • #操作 hdfs 超时时间  
35   • a1.sinks.k1.hdfs.callTimeout=30000  
36   •  
37   • #组装 source、channel、sink  
38   • a1.sources.r1.channels = c1  
39   • a1.sinks.k1.channel = c1
```

采集端文件生成脚本

- 在 node01 与 node02 上面开发 shell 脚本，模拟数据生成

```
1 cd /kkb/install/shells  
2 vim server.sh
```

内容如下

```
1#!/bin/bash  
2while true  
3do  
4 date >> /kkb/install/taillogs/access.log;  
5 date >> /kkb/install/taillogs/web.log;  
6 date >> /kkb/install/taillogs/nginx.log;  
7 sleep 0.5;  
8done
```

node01、node02 给脚本添加可执行权限

```
1 chmod u+x server.sh
```

顺序启动服务

node03 启动 flume 实现数据收集

```
1 cd /kkb/install/apache-flume-1.9.0-bin/  
2 bin/flume-ng agent -c conf -f conf/avro_source_hdfs_sink.conf -name a1 -  
2 Dflume.root.logger=DEBUG,console
```

- node01 与 node02 启动 flume 实现数据监控

```
1 cd /kkb/install/apache-flume-1.9.0-bin/  
2 bin/flume-ng agent -c conf -f conf/exec_source_avro_sink.conf -name a1 -  
2 Dflume.root.logger=DEBUG,console
```

- node01 与 node02 启动生成文件脚本

1 cd /kkb/install/shells

2 sh server.sh

查看 hdfs 目录/source/logs

Flume 综合案例之自定义拦截器使用

案例需求：

- 在数据采集之后，通过 Flume 的拦截器，实现将无效的 JSON 格式的消息过滤掉。

实现步骤

第一步：创建 maven java 工程，导入 jar 包

```
<dependencies>
    <dependency>
        <groupId>org.apache.flume</groupId>
        <artifactId>flume-ng-core</artifactId>
        <version>1.9.0</version>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.0</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <encoding>UTF-8</encoding>
                <!--          <verbal>true</verbal>-->
```

```

        </configuration>
    </plugin>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.1.1</version>
        <executions>
            <execution>
                <phase>package</phase>
                <goals>
                    <goal>shade</goal>
                </goals>
                <configuration>
                    <filters>
                        <filter>
                            <artifact>*:*</a
rtifact>
                        <excludes>
                            <exclude
>META-INF/*.SF</exclude>
                            <exclude
>META-INF/*.DSA</exclude>
                            <exclude
>META-INF/*.RSA</exclude>
                            </excludes>
                        </filter>
                    </filters>
                    <transformers>
                        <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                            <mainClass></mai
nClass>
                        </transformer>
                    </transformers>
                </configuration>
            </execution>
        <executions>
    </plugin>
</plugins>
</build>

```

第二步：自定义 flume 的拦截器

新建 package com.niit.flume.interceptor
新建类及内部类，分别是 JsonInterceptor、MyBuilder

```
package com.niit.flume.interceptor;

import org.apache.commons.codec.Charsets;
import org.apache.commons.lang.StringUtils;
import org.apache.flume.Context;
import org.apache.flume.Event;
import org.apache.flume.interceptor.Interceptor;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

/**
 * @Author: deLucia
 * @Date: 2021/6/2
 * @Version: 1.0
 * @Description:
 */
public class JsonInterceptor implements Interceptor {

    private static final Logger LOG =
    LoggerFactory.getLogger(JsonInterceptor.class);
    private String tag = "";

    public JsonInterceptor(String tag) {
        this.tag = tag;
    }

    @Override
    public void initialize() {

    }

    /**
     * 过滤 JSON 格式之外的数据 { "key":value, ... }
     * @param event
     * @return
     */
}
```

```
@Override
public Event intercept(Event event) {
    String line = new String(event.getBody(), Charsets.UTF_8);
    line = line.trim();
    if (StringUtils.isBlank(line)) {
        return null;
    }

    // {..}
    if (line.startsWith("{}") && line.endsWith("{}")) {

        Map<String, String> headers = event.getHeaders();
        headers.put("tag", tag);
        return event;

    } else {
        LOG.warn("Not Valid JSON: " + line);
        return null;
    }
}

@Override
public List<Event> intercept(List<Event> list) {
    List<Event> out = new ArrayList<Event>();
    for (Event event : list) {
        Event outEvent = intercept(event);
        if (outEvent != null) {
            out.add(outEvent);
        }
    }
    return out;
}

@Override
public void close() {

}

/**
 * 相当于自定义 Interceptor 的工厂类
 * 在 flume 采集配置文件中通过指定该 Builder 来创建 Interceptor 对象
 *
 * @author
 */
public static class MyBuilder implements Interceptor.Builder {
```

```

private String tag;

@Override
public void configure(Context context) {
    //从 flume 的配置文件中获得拦截器的“tag”属性值
    this.tag = context.getString("tag", "").trim();
}

/*
 * @see
org.apache.flume.interceptor.Interceptor.Builder#build()
*/
@Override
public JsonInterceptor build() {
    return new JsonInterceptor(tag);
}
}
}
}

```

第三步：打包上传服务器

将我们的拦截器打成 jar 包放到主机名为 hadoop100 的服务器上的 flume 安装目录下的 lib 目录下

第四步：开发 flume 的配置文件

开发 flume 的配置文件

```

1 cd /opt/pkg/flume/conf/
2 vim nc-interceptor-logger.conf

```

内容如下

记得将下边的 i1.type 根据自己的实际情况进行替换

```

1 #声明三种组件
2 a1.sources = r1
3 a1.channels = c1
4 a1.sinks = k1
5
6 #定义 source 信息

```

```
7 a1.sources.r1.type=netcat
8 a1.sources.r1.bind=localhost
9 a1.sources.r1.port=8888
10 #定义 source 的拦截器
11 a1.sources.r1.interceptors = i1
12 #根据自定义的拦截器，相应修改全类名及内部类名
13 a1.sources.r1.interceptors.i1.type
14 =com.niit.flume.interceptor.JsonInterceptor$MyBuilder
15 ## tag 的内容将被设置到消息头中
16 a1.sources.r1.interceptors.i1.tag = json_event
17
18 #定义 sink 信息
19 a1.sinks.k1.type=logger
20
21 #定义 channel 信息
22 a1.channels.c1.type=memory
23
24 #绑定在一起
25 a1.sources.r1.channels=c1
    a1.sinks.k1.channel=c1
```

第五步：运行 FLume Agent

同时开启 INFO 日志级别

```
[hadoop@hadoop100 conf]$ bin/flume-ng agent -c conf/ -f
conf/nc_interceptor_logger.conf -n a1 -Dflume.root.logger=INFO,console
```

第六步：简单测试

通过 netcat 客户端发送数据

```
1 [hadoop@hadoop100 ~]$ nc localhost 8888
2 asd
3 dsa
4 {"OK
5 OK
6 OK
```

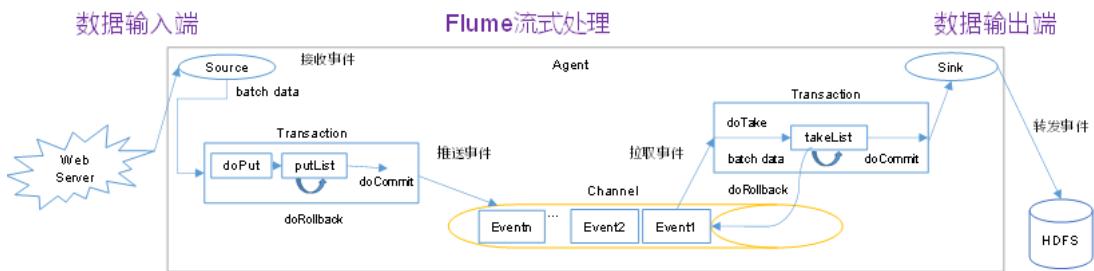
```
7 OK
8 {"key":123}
9 OK
```

查看 flume 的 agent 的日志信息如下

```
2021-05-26 14:40:12,233 (lifecycleSupervisor-1-4) [INFO - org.apache.flume.source.NetcatSource.start(NetcatSource.java:155)] Source starting
1 2021-05-26 14:40:17,326 (lifecycleSupervisor-1-4) [INFO - org.apache.flume.source.NetcatSource.start(NetcatSource.java:166)] Created
2 serverSocket:sun.nio.ch.ServerSocketChannelImpl[/127.0.0.1:8888]
3 2021-05-26 14:40:20,952 (netcat-handler-0) [WARN - com.niit.flume.interceptor.JsonInterceptor.intercept(JsonInterceptor.java:48)] Invalid json format event !
4 2021-05-26 14:40:20,953 (netcat-handler-0) [WARN - com.niit.flume.interceptor.JsonInterceptor.intercept(JsonInterceptor.java:48)] Invalid json format event !
5 2021-05-26 14:40:24,109 (netcat-handler-0) [WARN - com.niit.flume.interceptor.JsonInterceptor.intercept(JsonInterceptor.java:48)] Invalid json format event !
6 2021-05-26 14:40:24,909 (netcat-handler-0) [WARN - com.niit.flume.interceptor.JsonInterceptor.intercept(JsonInterceptor.java:48)] Invalid json format event !
7 2021-05-26 14:40:36,895 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache
```

Flume 进阶

一、flume 事务



put 事务流程

1、doPut

将批量数据先写入临时缓冲区 putList

2、doCommit

检查 Channel 内存队列是否足够，

- (1) 达到一定时间没有数据写入到 putList
- (2) 达到了 putListCapacity 容量

3、doRollback

Channel 内存队列空间不足，回滚数据到 putList，会被 channel 打回来

take 事务流程：

1、doTake

将数据取到临时缓冲区 takeList，并将数据发送到 HDFS

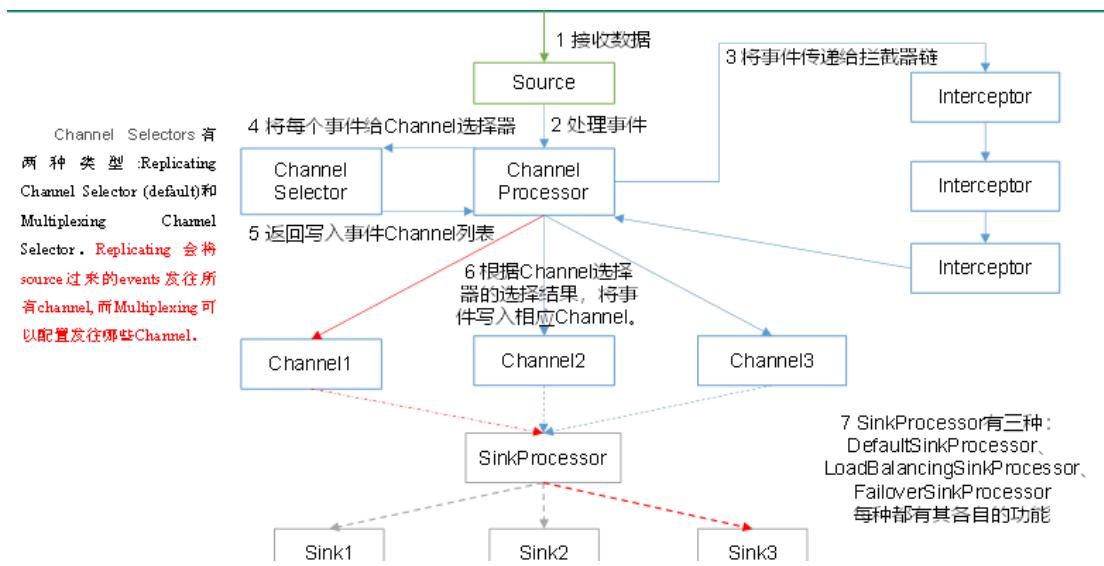
2、doCommit

如果数据全部发送成功，则清除临时缓冲区 takeList

3、doRollback

数据发送过程中如果出现异常，rollback 将临时缓冲区 takeList 中数据全部打回给 Channel 内存队列

二、Flume Agent 内部原理



重要组件：

1) ChannelSelector

ChannelSelector 的作用就是选出 event 将要被发往哪个 Channel。

共有两种类型：Replicating（复制）和 Multiplexing（多路复用）

ReplicatingSelector 会将同一个 event 发往所有的 Channel

MultiplexingSelector 会根据相应的原则，将不同的 event 发往不同的 Channel

2) SinkProcessor

sinkProcessor 共有三种类型：DefaultSinkProcessor、LoadBalancingProcessor 和 FailoverSinkProcessor

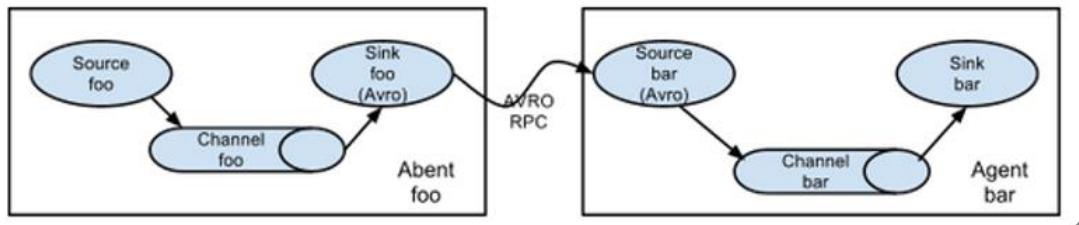
DefaultSinkProcessor：对应的是单个 sink

LoadBalancingProcessor：对应的是 sink group，可以实现负载均衡

FailoverSinkProcessor: 对应的是 sink group, 可以实现故障恢复

三、flume 拓扑结构

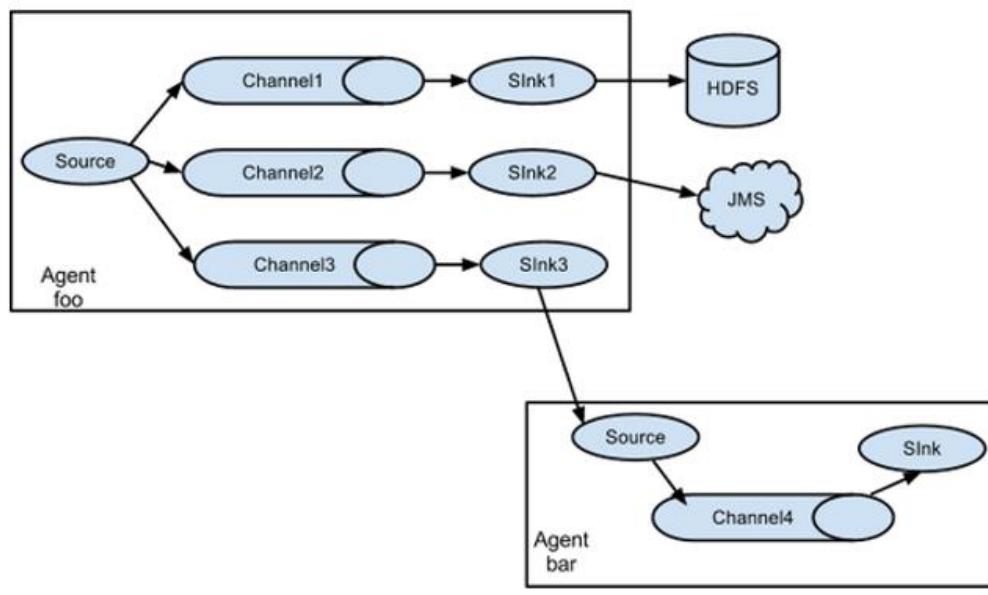
1、简单串联



将多个 flume 顺序连接起来，从最初的 Source 开始到最终 sink 传送的目的存储系统。

此模式不建议桥接过多的 flume 数量，flume 数据过多不仅会影响传输速率，而且一旦传输过程中某个节点 flume 宕机，会影响整个传输系统

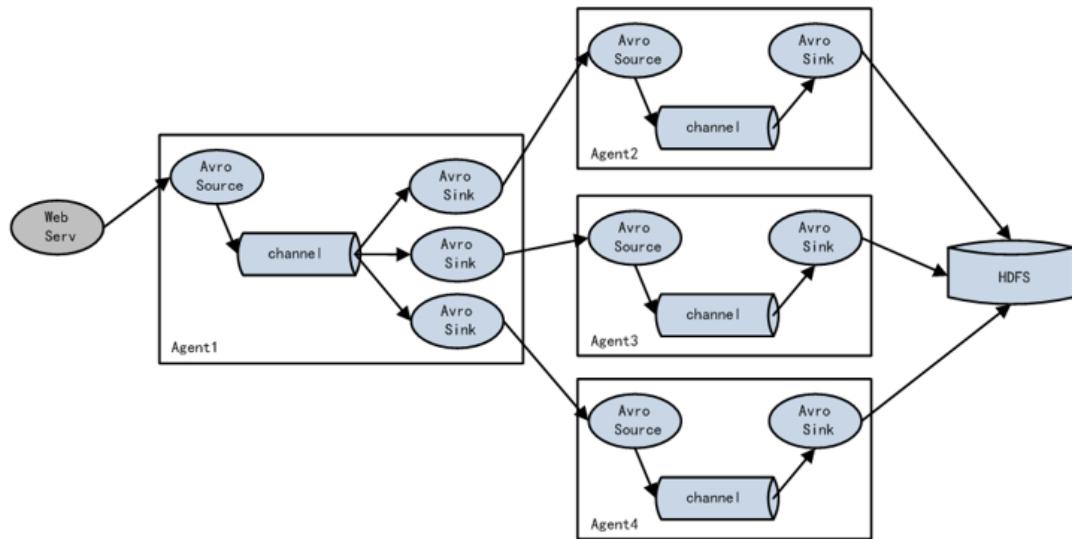
2、复制和多路复用



flume 支持将事件流向一个或者多个目的地。

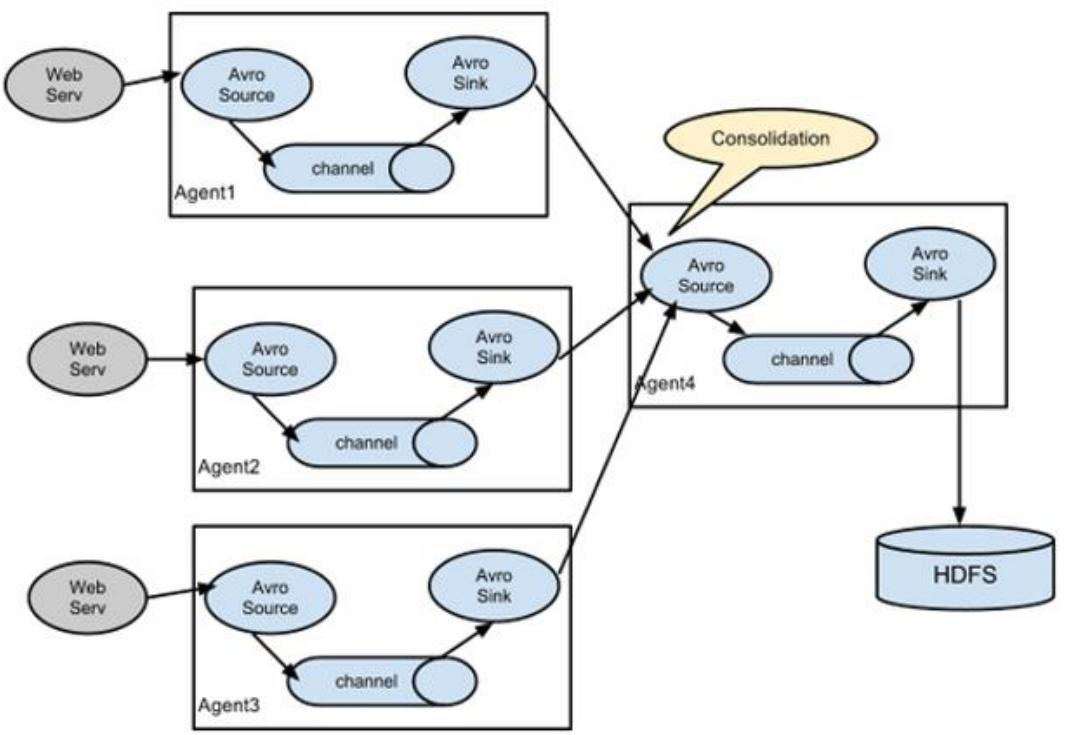
这种模式可以将相同数据复制到多个 Channel 中，或者将不同数据分发到不同的 Channel 中，sink 可以选择传送到不同的目的地

3、负载均衡和故障转移



flume 支持使用将多个 sink 逻辑上分到一个 sink 组，sink 组配合不同的 sinkProcessor 可以实现负载均衡和错误恢复的功能

4、聚合



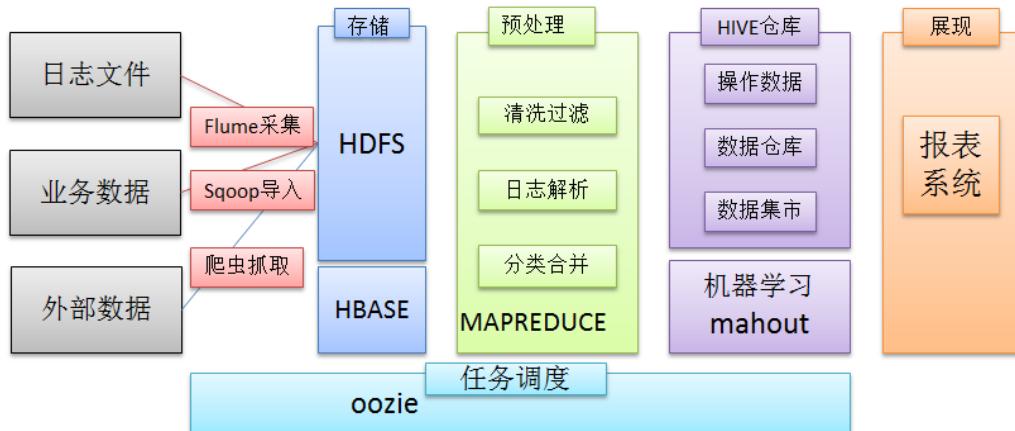
这种模式是我们最常见的，也非常实用，日常 web 应用通常分布在上百个服务器，大者甚至上千个、上万个服务器。产生的日志，处理起来也非常麻烦。

用 flume 的这种组合方式能很好的解决这一问题，每台服务器部署一个 flume 采集日志，传送到一个集中收集日志的 flume，再由此 flume 上传到 hdfs、hive、hbase 等，进行日志分析。

Flume 日志采集

前言

在一个完整的离线大数据处理系统中，除了 hdfs+mapreduce+hive 组成分析系统的核芯之外，还需要数据采集、结果数据导出、任务调度等不可或缺的辅助系统，而这些辅助工具在 hadoop 生态体系中都有便捷的开源框架，如图所示：



Flume 基本介绍

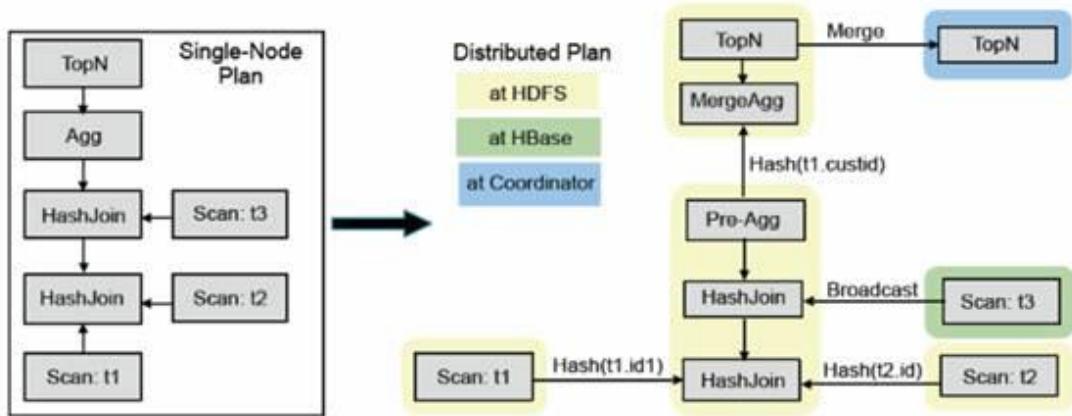
1. 概述

Flume 是一个分布式、可靠、和高可用的海量日志采集、聚合和传输的系统。

- Flume 可以采集文件，socket 数据包、文件、文件夹、kafka 等各种形式源数据，又可以将采集到的数据(下沉 sink)输出到 HDFS、hbase、hive、kafka 等众多外部存储系统中
- 一般的采集需求，通过对 flume 的简单配置即可实现
- Flume 针对特殊场景也具备良好的自定义扩展能力，因此，flume 可以适用于大部分的日常数据采集场景

2. 运行机制

- Flume 分布式系统中最核心的角色是 **agent**, flume 采集系统就是由一个个 agent 所连接起来形成的
- 每一个 agent 相当于一个数据传递员，内部有三个组件：
 - Source: 采集组件，用于跟数据源对接，以获取数据
 - Sink: 下沉组件，用于往下一级 agent 传递数据或者往最终存储系统传递数据
 - Channel: 传输通道组件，用于从 source 将数据传递到 sink



3. Flume 采集系统结构图

1. 简单结构

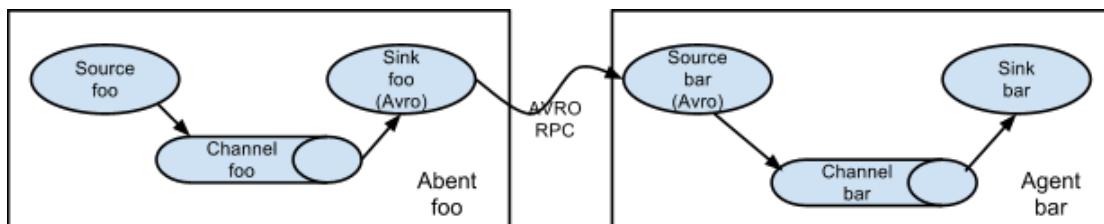
- 单个 agent 采集数据

相关系统对比

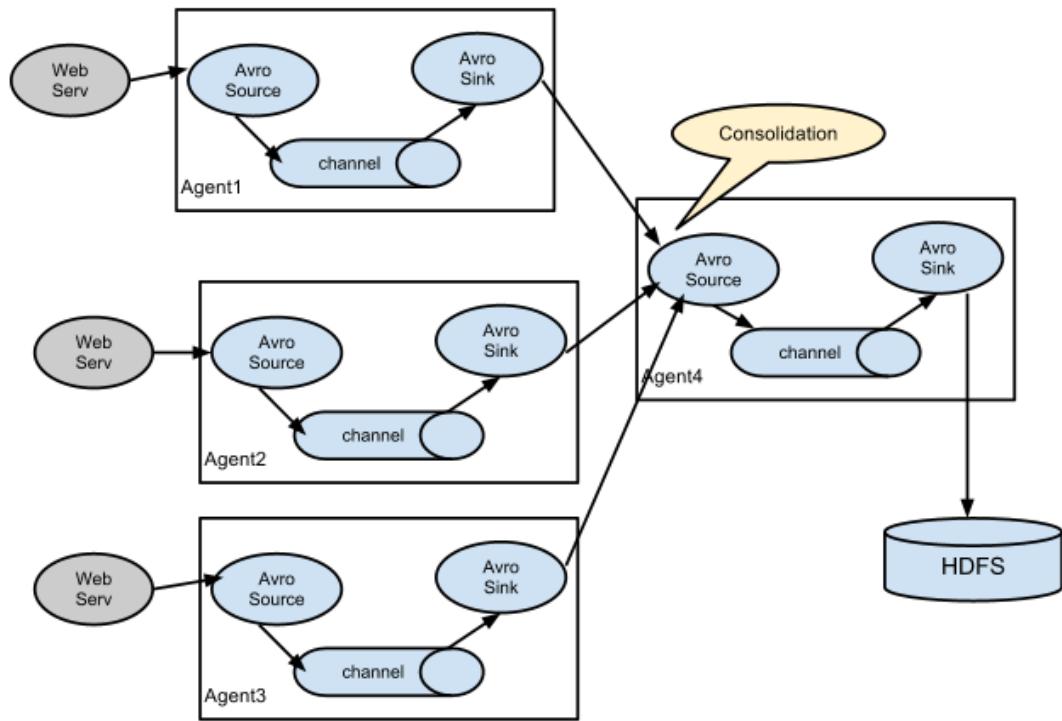
	HIVE	Spark	Impala
概要	Hive是老牌的SQL-on-hadoop解决方案	spark之上的交互式SQL解决方案 提供DataFrame API作为SQL补充，方便实现ETL、分析、机器学习等复杂逻辑。 SparkSQL使用scala定义UDF，非常方便。	hadoop上交互式MPP SQL引擎
容错	强。中间结果落盘，容错能力强，最细粒度容错。	较强。基于数据血缘关系的数据恢复。	弱。遇到问题时，需要重做查询
性能	慢。MR启动慢中间结果落盘	较快。基于RDD模型，DAG执行基于代价的查询优化内存基于列存编译执行(生成java字节码)Java实现	快。MPP架构基于代价的查询优化，JOIN优化LLVM查询编译C++实现向量执行引擎，可使用SSE指令列存IO本地化资源管理器优化（LLAM）

2. 复杂结构

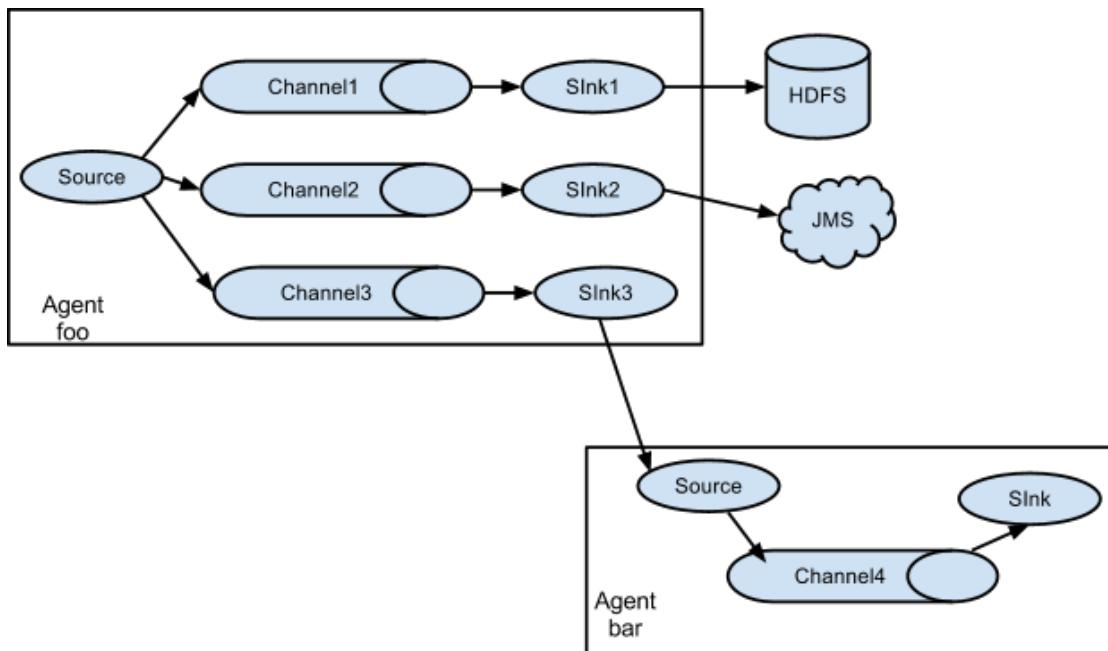
- 两个 agent 之间串联



- 多级 agent 之间串联



- 多级 channel



Flume 实战案例

- 需求：收集网络端口的数据，并将数据打印到 linux 的控制台上面



Flume 的安装部署

第一步：下载解压修改配置文件

- Flume 的安装非常简单，只需要解压即可
- 上传安装包到数据源所在节点上
- 这里我们在第三台机器 node03 来进行安装

- 1 • cd /kkb/soft
 - 2 • tar -xzvf apache-flume-1.9.0-bin.tar.gz -C /kkb/install/
 - 3 • cd /kkb/install/apache-flume-1.9.0-bin/conf/
 - 4 • cp flume-env.sh.template flume-env.sh
 - 5 • vim flume-env.sh
-
- 修改如下内容
 - export JAVA_HOME=/kkb/install/jdk1.8.0_141

解决jar包冲突

apache-flume-1.9.0-bin、hadoop-3.1.4 都有 guava 包，但是版本不一致，会造成冲突

```
java.lang.NoSuchMethodError: com.google.common.base.Preconditions.checkNotNull([Ljava/lang/String;Ljava/lang/Object;)V
    at org.apache.hadoop.conf.Configuration.set(Configuration.java:1357)
    at org.apache.hadoop.conf.Configuration.set(Configuration.java:1338)
    at org.apache.hadoop.conf.Configuration.setBoolean(Configuration.java:1679)
    at org.apache.flume.sink.hdfs.BucketWriter.open(BucketWriter.java:221)
    at org.apache.flume.sink.hdfs.BucketWriter.append(BucketWriter.java:572)
    at org.apache.flume.sink.hdfs.HDFSEventSink.process(HDFSEventSink.java:412)
    at org.apache.flume.sink.DefaultSinkProcessor.process(DefaultSinkProcessor.java:67)
    at org.apache.flume.SinkRunner$PollingRunner.run(SinkRunner.java:145)
    at java.lang.Thread.run(Thread.java:748)
Exception in thread "SinkRunner-PollingRunner-DefaultSinkProcessor" java.lang.NoSuchMethodError: com.google.common.base.Preconditions.checkNotNull([Ljava/lang/String;Ljava/lang/Object;)V
    at org.apache.hadoop.conf.Configuration.set(Configuration.java:1357)
    at org.apache.hadoop.conf.Configuration.set(Configuration.java:1338)
    at org.apache.hadoop.conf.Configuration.setBoolean(Configuration.java:1679)
    at org.apache.flume.sink.hdfs.BucketWriter.open(BucketWriter.java:221)
    at org.apache.flume.sink.hdfs.BucketWriter.append(BucketWriter.java:572)
    at org.apache.flume.sink.hdfs.HDFSEventSink.process(HDFSEventSink.java:412)
    at org.apache.flume.sink.DefaultSinkProcessor.process(DefaultSinkProcessor.java:67)
    at org.apache.flume.SinkRunner$PollingRunner.run(SinkRunner.java:145)
```

解决冲突；将 hadoop 中高版本的 guava 包，替换 flume 中低版本的包

```
1 cd /kkb/install/apache-flume-1.9.0-bin/lib
```

```
2 rm -f guava-11.0.2.jar
```

```
3 cp /kkb/install/hadoop-3.1.4/share/hadoop/common/lib/guava-27.0-jre.jar .
```

第二步：开发配置文件

根据数据采集的需求**配置采集方案**，描述在配置文件中（文件名可任意自定义）

需求：配置我们的网络收集的配置文件；从某 socket 端口采集数据，采集到的数据打印到 console 控制台

在 flume 的 conf 目录下新建一个配置文件（采集方案）

```
1 cd /kkb/install/apache-flume-1.9.0-bin/conf
```

```
2 vim netcat-logger.conf
```

内容如下

```
1 # 定义这个 agent 中各组件的名字
2 a1.sources = r1
3 a1.sinks = k1
4 a1.channels = c1
```

```
5
6 # 描述和配置 source 组件: r1
7 a1.sources.r1.type = netcat
8 # 当前节点的 ip 地址
9 a1.sources.r1.bind = node03
10 a1.sources.r1.port = 44444
11
12 # 描述和配置 sink 组件: k1
13 a1.sinks.k1.type = logger
14
15 # 描述和配置 channel 组件, 此处使用是内存缓存的方式
16 a1.channels.c1.type = memory
17 # channel 中存储的 event 的最大个数
18 a1.channels.c1.capacity = 1000
19 # channel 每次从 source 获得的 event 最多个数或一次发往 sink 的 event 最多个数
20 a1.channels.c1.transactionCapacity = 100
21
22 # 描述和配置 source     channel     sink 之间的连接关系
23 a1.sources.r1.channels = c1
24 a1.sinks.k1.channel = c1
```

对应类型组件的官网文档

netcat-tcp-source

logger-sink

memory-channel

第三步：启动配置文件

指定采集方案配置文件， 在相应的节点上启动 flume agent

先用一个最简单的例子来测试一下程序环境是否正常

启动 agent 去采集数据

```
1 bin/flume-ng agent -c conf -f conf/netcat-logger.conf -n a1 -
Dflume.root.logger=INFO,console
```

-c conf 指定 flume 自身的 conf 目录中的配置文件

-f conf/netcat-logger.conf 指定我们所描述的采集方案

-n a1 指定我们这个 agent 的名字

-Dflume.root.logger=INFO,console 将 info 级别的日志打印到控制台

第四步：安装 telent 准备测试

在 node02 机器上面安装 telnet 客户端，用于模拟数据的发送

```
1 sudo yum -y install telnet  
2 telnet node03 44444 # 使用 telnet 模拟数据发送
```

采集案例

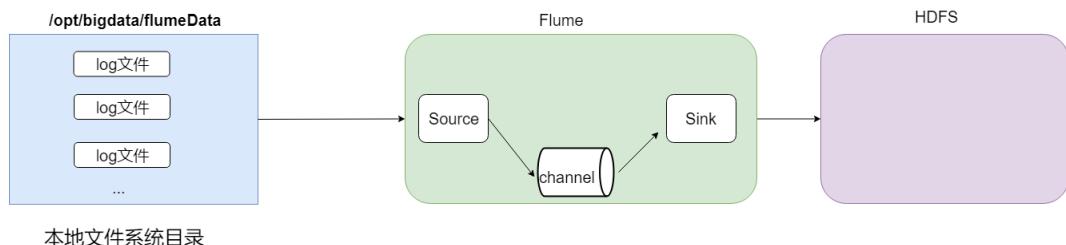
接下来的案例，需要用到 HDFS

先 start-dfs.sh 启动 HDFS

1. 采集目录到 HDFS

需求分析

结构示意图：



采集需求：某服务器的某特定目录下，会不断产生新的文件，每当有==新文件==出现，就需要把文件采集到 HDFS 中去

根据需求，首先定义以下 3 大要素

数据源组件，即 source —— 监控文件目录 : spooldir

spooldir 特性：

1、监视一个目录，只要目录中出现新文件，就会采集文件中的内容

2、采集完成的文件，会被 agent 自动添加一个后缀：COMPLETED

3、此 source 可靠，不会丢失数据；即使 flume 重启或被 kill

注意：

所监视的目录中不允许有同名的文件；且文件被放入 spooldir 后，就不能修改

①如果文件放入 spooldir 后，又向文件写入数据，会打印错误及停止

②如果有同名的文件出现在 spooldir，也会打印错误及停止

下沉组件，即 sink——HDFS 文件系统 : hdfs sink

通道组件，即 channel——可用 file channel 也可以用内存 channel

flume 配置文件开发

配置文件编写：

```
1 cd /kpb/install/apache-flume-1.9.0-bin/conf/  
2 mkdir -p /kpb/install/dirfile  
3 vim spooldir.conf
```

内容如下

```
1 # Name the components on this agent  
2 a1.sources = r1  
3 a1.sinks = k1  
4 a1.channels = c1  
5  
6 # Describe/configure the source  
7 # 注意：不能往监控目中重复丢同名文件  
8 a1.sources.r1.type = spooldir  
9 # 监控的路径  
10 a1.sources.r1.spoolDir = /kpb/install/dirfile
```

```
11 # Whether to add a header storing the absolute path filename
12 #文件绝对路径放到header
13 a1.sources.r1.fileHeader = true
14
15 # Describe the sink
16 a1.sinks.k1.type = hdfs
17 a1.sinks.k1.channel = c1
18 #采集到的数据写入到次路径
19 a1.sinks.k1.hdfs.path =
20 hdfs://node01:8020/spooldir/files/%y-%m-%d/%H%M/
21 # 指定在 hdfs 上生成的文件名前缀
22 a1.sinks.k1.hdfs.filePrefix = events-
23 # timestamp 向下舍 round down
24 a1.sinks.k1.hdfs.round = true
25 # 按 10 分钟, 为单位向下取整; 如 55 分, 舍成 50; 38 -> 30
26 a1.sinks.k1.hdfs.roundValue = 10
27 # round 的单位
28 a1.sinks.k1.hdfs.roundUnit = minute
29 # 每 3 秒滚动生成一个文件; 默认 30; (0 = never roll based on time interval)
30 a1.sinks.k1.hdfs.rollInterval = 3
31 # 每 x 字节, 滚动生成一个文件; 默认 1024; (0: never roll based on file size)
32 a1.sinks.k1.hdfs.rollSize = 20
33 # 每 x 个 event, 滚动生成一个文件; 默认 10; (0 = never roll based on number
34 of events)
35 a1.sinks.k1.hdfs.rollCount = 5
36 # 每 x 个 event, flush 到 hdfs
37 a1.sinks.k1.hdfs.batchSize = 1
38 # 使用本地时间
39 a1.sinks.k1.hdfs.useLocalTimeStamp = true
40 #生成的文件类型, 默认是 Sequencefile; 可选 DataStream, 则为普通文本; 可选
41 CompressedStream 压缩数据
42 a1.sinks.k1.hdfs.fileType = DataStream
43
44 # Use a channel which buffers events in memory
45 a1.channels.c1.type = memory
46 # channel 中存储的 event 的最大数目
47 a1.channels.c1.capacity = 1000
48 # 每次传输数据, 从 source 最多获得 event 的数目或向 sink 发送的 event 的最大的数
49 目
50 a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

组件官网地址：

spooling directory source

hdfs sink

memory channel

Channel 参数解释：

capacity：默认该通道中最大的可以存储的 event 数量

trasactionCapacity：每次最大可以从 source 中拿到或者送到 sink 中的 event 数量

keep-alive：event 添加到通道中或者移出的允许时间

启动 flume

```
cd /kpb/install/apache-flume-1.9.0-bin  
bin/flume-ng agent -c ./conf -f ./conf/spooldir.conf -n a1 -  
Dflume.root.logger=INFO,console
```

上传文件到指定目录

将不同的文件上传到下面目录里面去，注意文件不能重名

```
1 cd /kpb/install/dirfile
```

```
2 mkdir /home/hadoop/datas
```

```
3 cd /home/hadoop/datas
```

```
4 vim a.txt
```

```
5
```

```
6 # 加入如下内容
```

```
7 ab cd ef
```

```
8 english math
```

```
9 hadoop alibaba
```

再执行；

```
1 cp a.txt /kpb/install/dirfile
```

然后观察 flume 的 console 动静、hdfs webui 生成的文件

Hadoop Overview Datanodes Snapshot Startup Progress Utilities

Browse Directory

/spooldir/files/20-07-24/1430 Go!

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	hadoop	supergroup	22 B	Fri Jul 24 14:34:55 +0800 2020	3	128 MB	events-.1595572493403
-rw-r--r--	hadoop	supergroup	15 B	Fri Jul 24 14:34:58 +0800 2020	3	128 MB	events-.1595572493404

Hadoop, 2017.

观察 spooldir 的目标目录

```
[hadoop@node03 ~]$ cd /kkb/install/dirfile
[hadoop@node03 dirfile]$ ll
total 4
-rw-rw-r-- 1 hadoop hadoop 37 Aug 25 11:45 a.txt.COMPLETED
[hadoop@node03 dirfile]$
```

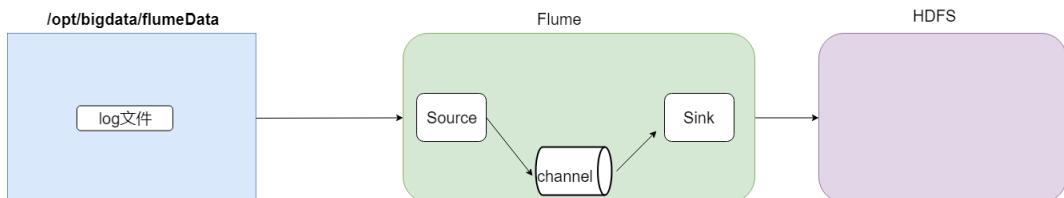
将同名文件再次放到/kkb/install/dirfile 观察现象：

```
1 cp a.txt /kkb/install/dirfile
```

flume 控制台报错

```
2020-07-24 14:45:29,429 [pool-3-thread-1] [ERROR - org.apache.flume.source.SpooldirectorySource$SpooldirectoryRunnable.run(Spooldirectorysource.java:280)] FATAL: Spool directory source r1: { spooldir: /kkb/install/dirfile }: uncaught exception in Spooldirectorysource thread. Restart or reconfigure Flume to continue processing.
java.lang.IllegalStateException: File name has been re-used with different files. Spooling assumptions violated for /kkb/install/dirfile/a.txt.COMPLETED
    at org.apache.flume.client.avro.ReliableSpoolingFileEventReader.rollCurrentFile(ReliableSpoolingFileEventReader.java:463)
    at org.apache.flume.client.avro.ReliableSpoolingFileEventReader.retireCurrentFile(ReliableSpoolingFileEventReader.java:414)
    at org.apache.flume.client.avro.ReliableSpoolingFileEventReader.readEvents(ReliableSpoolingFileEventReader.java:326)
    at org.apache.flume.source.SpooldirectorySource$SpooldirectoryRunnable.run(Spooldirectorysource.java:250)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
    at java.util.concurrent.FutureTask.runAndReset(FutureTask.java:308)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$301(ScheduledThreadPoolExecutor.java:180)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:294)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
    at java.lang.Thread.run(Thread.java:483)
2020-07-24 14:45:33,859 [SinkRunner-pollingrunner-defaultSinkProcessor] [INFO - org.apache.flume.sink.hdfs.HDFSDatamaster.configure(HDFSDataStream.java:57)] serializer = TEXT, useRawLocalFilesystem = false
```

2. 采集文件到 HDFS



该文件内容在不断追加

需求分析：

采集需求：比如业务系统使用 log4j 生成的日志，日志内容不断增加，需要把追加到日志文件中的数据实时采集到 hdfs

根据需求，首先定义以下 3 大要素

采集源，即 source——监控文件内容更新：exec ‘tail -F file’

下沉目标，即 sink——HDFS 文件系统：hdfs sink

Source 和 sink 之间的传递通道——channel，可用 file channel 也可以用 内存 channel

flume 的配置文件开发

- node03 开发配置文件

```
1 cd /kkb/install/apache-flume-1.9.0-bin/conf  
2 vim tail-file.conf
```

- 配置文件内容

```
1 agent1.sources = source1  
2 agent1.sinks = sink1  
3 agent1.channels = channel1  
4  
5 # Describe/configure tail -F source1  
6 agent1.sources.source1.type = exec  
7 agent1.sources.source1.command = tail -F /kkb/install/taillogs/access_log  
8 agent1.sources.source1.channels = channel1  
9  
10 # Describe sink1  
11 agent1.sinks.sink1.type = hdfs  
12 agent1.sinks.sink1.hdfs.path = hdfs://node01:8020/weblog/flume-collection/%y-%m-%d/%H-%M  
13 agent1.sinks.sink1.hdfs.filePrefix = access_log  
14 # 允许打开的文件数：如果超出 5000，老文件会被关闭  
15 agent1.sinks.sink1.hdfs.maxOpenFiles = 5000  
16 agent1.sinks.sink1.hdfs.batchSize= 100  
17 agent1.sinks.sink1.hdfs fileType = DataStream  
18 agent1.sinks.sink1.hdfs.writeFormat =Text  
19 agent1.sinks.sink1.hdfs.rollSize = 102400  
20 agent1.sinks.sink1.hdfs.rollCount = 1000000  
21 agent1.sinks.sink1.hdfs.rollInterval = 60
```

```
22 agent1.sinks.sink1.hdfs.round = true
23 agent1.sinks.sink1.hdfs.roundValue = 10
24 agent1.sinks.sink1.hdfs.roundUnit = minute
25 agent1.sinks.sink1.hdfs.useLocalTimeStamp = true
26
27 # Use a channel which buffers events in memory
28 agent1.channels.channel1.type = memory
29 # 向 channel 添加一个 event 或从 channel 移除一个 event 的超时时间
30 agent1.channels.channel1.keep-alive = 120
31 agent1.channels.channel1.capacity = 500000
32 agent1.channels.channel1.transactionCapacity = 600
33
34 # Bind the source and sink to the channel
35 agent1.sources.source1.channels = channel1
36 agent1.sinks.sink1.channel = channel1
```

启动 flume

```
1 cd /kkb/install/apache-flume-1.9.0-bin
2 bin/flume-ng agent -c conf -f conf/tail-file.conf -n agent1 -
Dflume.root.logger=INFO,console
```

开发 shell 脚本定时追加文件内容

```
1 mkdir -p /kkb/install/shells/
2 cd /kkb/install/shells/
3 vim tail-file.sh
```

内容如下

```
1 #!/bin/bash
2 while true
3 do
4 date >> /kkb/install/taillogs/access_log;
5 sleep 0.5;
6 done
```

创建文件夹: mkdir -p /kkb/install/taillogs

```
启动脚本: chmod u+x tail-file.sh  
sh /kkb/install/shells/tail-file.sh
```

3. 实现断点续传功能

不管是上面的 spoolDir 还是 exec Source dir 都有一个缺陷就是没法实现==断点续传==的功能，为此在 flume1.7 当中特地新增加一个 source 叫做 tail-dir source，专门用于解决断点续传的问题，tail-dirsource 可以监控文件或者文件夹，允许我们使用正则表达式的方式来对我们的文件或者文件夹进行监听

需求分析:

采集需求，使用 tail-dir source 监听某个目录下的多个文件，并且实现文件的断点续传功能

flume 配置文件开发

定义 flume 的配置文件

```
1 cd /kkb/install/apache-flume-1.9.0-bin/conf/
```

```
2 vim tail-dir.conf
```

内容如下

```
1 # Name the components on this agent  
2 a1.sources = r1  
3 a1.sinks = k1  
4 a1.channels = c1  
5  
6 # Describe/configure the source  
7 a1.sources.r1.type = TAILDIR  
8 # 以 json 格式，记录读取的每个文件及读取的 position  
9 a1.sources.r1.positionFile = /kkb/install/apache-flume-1.9.0-bin/taildir_position.json  
10 # 每个 filegroup 代表一系列待 tail 的文件
```

```

11 a1.sources.r1.filegroups = f1
12 # 指定 filegroup 的绝对路径
13 a1.sources.r1.filegroups.f1 = /kkb/install/dirfile/*log/*
14 # 此项用于控制从一个文件连续读取数据的批次；比如有 A、B、C 多个文件，如果向 A 文件写入的频率非常高，导致一直循环的从 A 中采集获取数据，而 B、
15 C 的数据不被处理；可将此值调低；每个批次由属性 batchSize 控制，默认 500 行
16 a1.sources.r1.maxBatchCount = 1000
17
18 # Describe the sink
19 a1.sinks.k1.type = hdfs
20 a1.sinks.k1.channel = c1
21 a1.sinks.k1.hdfs.path = hdfs://node01:8020/taildir/files/%y-%m-%d/%H%M/
22 a1.sinks.k1.hdfs.filePrefix = events-
23 a1.sinks.k1.hdfs.round = true
24 a1.sinks.k1.hdfs.roundValue = 10
25 a1.sinks.k1.hdfs.roundUnit = minute
26 a1.sinks.k1.hdfs.rollInterval = 3
27 a1.sinks.k1.hdfs.rollSize = 5000
28 a1.sinks.k1.hdfs.rollCount = 50000
29 # 每 x 个 event flush 到 hdfs
30 a1.sinks.k1.hdfs.batchSize = 1000
31 a1.sinks.k1.hdfs.useLocalTimeStamp = true
32 #生成的文件类型，默认是 Sequencefile，可用 DataStream，则为普通文本
33 a1.sinks.k1.hdfs fileType = DataStream
34
35 # Use a channel which buffers events in memory
36 a1.channels.c1.type = memory
37 a1.channels.c1.capacity = 1000
38 a1.channels.c1.transactionCapacity = 100
39 # Bind the source and sink to the channel
40 a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1

```

组件官网文档:

[tail dir](#)

启动 flume

node03 执行以下命令启动 flume

```
1 cd /kkb/install/apache-flume-1.9.0-bin/  
2 bin/flume-ng agent -c conf -f conf/tail-dir.conf -n a1 -  
Dflume.root.logger=INFO,console
```

创建文件

node03 执行以下命令创建文件到指定文件夹下

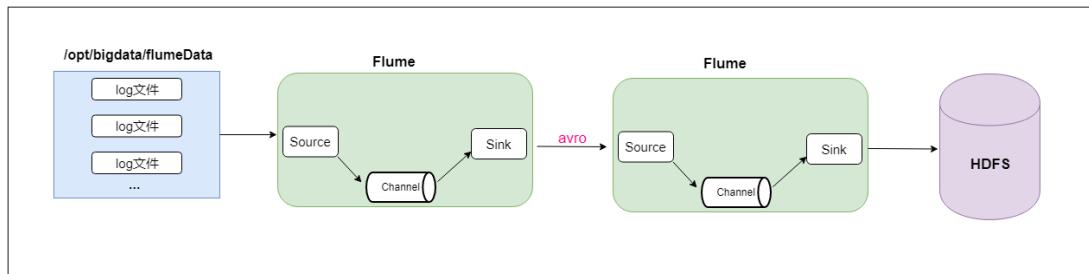
```
1 echo "testlog" >> /kkb/install/dirfile/file.log  
2 echo "testlog2" >> /kkb/install/dirfile/file1.log  
3 echo "testlog3" >> /kkb/install/dirfile/file2.log
```

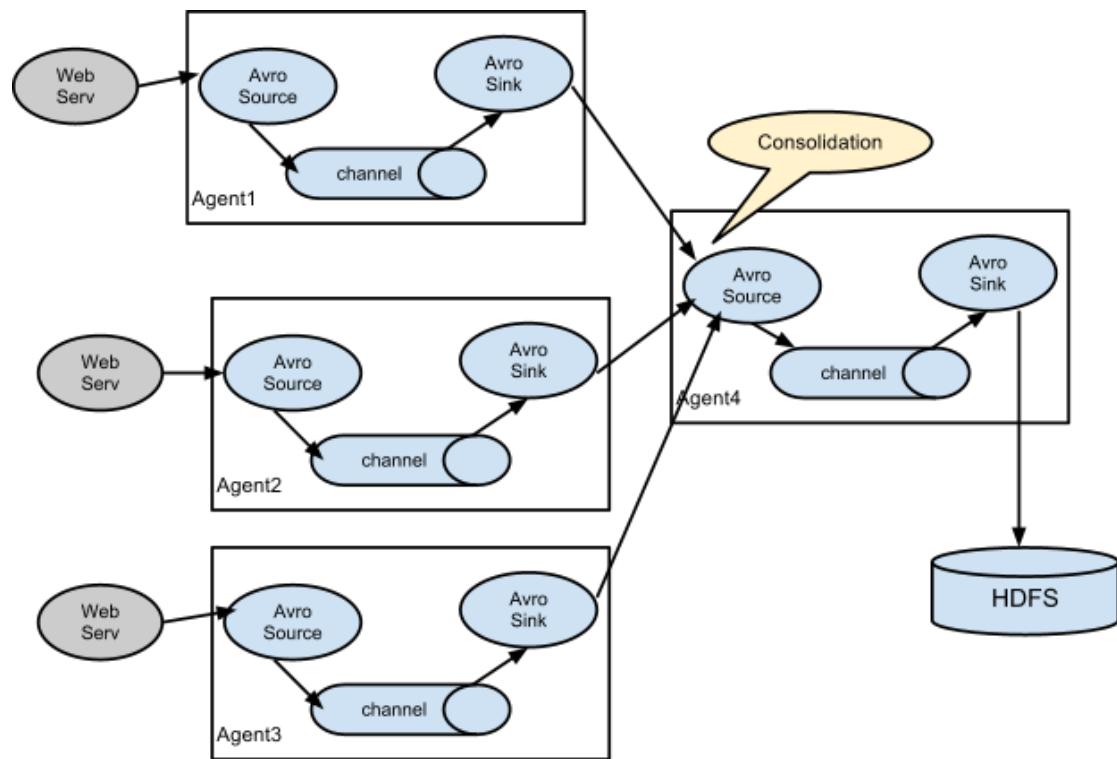
可观察下 taildir_position.json 文件中记录的内容

```
1 cd /kkb/install/apache-flume-1.9.0-bin/  
2 vim taildir_position.json
```

```
[{"inode":1456863,"pos":35,"file":"/kkb/install/dirfile/file1.log"}, {"inode":1456864,"pos":8,"file":"/kkb/install/dirfile/file.log"}, {"inode":1456865,"pos":9,"file":"/kkb/install/dirfile/file2.log"}]
```

4. 两个 agent 级联





需求分析：

第一个 agent 负责收集文件当中的数据，通过网络发送到第二个 agent 当中去；

第二个 agent 负责接收第一个 agent 发送的数据，并将数据保存到 hdfs 上面去

第一步：node02 安装 flume

- 将 node03 机器上面解压后的 flume 文件夹拷贝到 node02 机器上面去

```
1 cd /kkb/install
```

```
2 scp -r apache-flume-1.9.0-bin/ node02:$PWD
```

第二步：node02 配置 flume 配置文件

在 node02 机器配置我们的 flume

```
1 cd /kkb/install/apache-flume-1.9.0-bin/conf/
```

```
2 vim tail-avro-avro-logger.conf
```

内容如下

```
1 # Name the components on this agent
```

```

2 a1.sources = r1
3 a1.sinks = k1
4 a1.channels = c1
5
6 # Describe/configure the source
7 a1.sources.r1.type = exec
8 a1.sources.r1.command = tail -F /kpb/install/taillogs/access.log
9 a1.sources.r1.channels = c1
10
11 # Describe the sink
12 # sink 端的 avro 是一个数据发送者
13 a1.sinks.k1.type = avro
14 a1.sinks.k1.channel = c1
15 a1.sinks.k1.hostname = node03
16 a1.sinks.k1.port = 4141
17 # 每一批次发送的 event 的数目
18 a1.sinks.k1.batch-size = 10
19
20 # Use a channel which buffers events in memory
21 a1.channels.c1.type = memory
22 a1.channels.c1.capacity = 1000
23 a1.channels.c1.transactionCapacity = 100
24
25 # Bind the source and sink to the channel
26 a1.sources.r1.channels = c1
27 a1.sinks.k1.channel = c1

```

[组件官网](#)

[*avro sink*](#)

第三步：node02 开发脚本文件往文件写入数据

直接将 node03 下面的脚本和数据拷贝到 node02 即可， node03 机器上执行以下命令

```

1 cd /kpb/install/shells
2 scp -r    tail-file.sh node02:$PWD

```

第四步：node03 开发 flume 配置文件

在 node03 机器上开发 flume 的配置文件

1 cd /kkb/install/apache-flume-1.9.0-bin/conf/

2 vim avro-hdfs.conf

- 内容如下

```
1 # Name the components on this agent
2 a1.sources = r1
3 a1.sinks = k1
4 a1.channels = c1
5
6 # Describe/configure the source
7 # source 中的 avro 组件是一个接收者服务
8 a1.sources.r1.type = avro
9 a1.sources.r1.channels = c1
10 a1.sources.r1.bind = node03
11 a1.sources.r1.port = 4141
12
13 # Describe the sink
14 a1.sinks.k1.type = hdfs
15 a1.sinks.k1.hdfs.path = hdfs://node01:8020/avro/hdfs/%y-%m-%d/%H%M/
16 a1.sinks.k1.hdfs.filePrefix = events-
17 a1.sinks.k1.hdfs.round = true
18 a1.sinks.k1.hdfs.roundValue = 10
19 a1.sinks.k1.hdfs.roundUnit = minute
20 a1.sinks.k1.hdfs.rollInterval = 3
21 a1.sinks.k1.hdfs.rollSize = 200
22 a1.sinks.k1.hdfs.rollCount = 5
23 a1.sinks.k1.hdfs.batchSize = 1
24 a1.sinks.k1.hdfs.useLocalTimeStamp = true
25 #生成的文件类型， 默认是 Sequencefile， 可用 DataStream，则为普通文本
26 a1.sinks.k1.hdfs.fileType = DataStream
27
28 # Use a channel which buffers events in memory
29 a1.channels.c1.type = memory
30 a1.channels.c1.capacity = 1000
31 a1.channels.c1.transactionCapacity = 100
32
33 # Bind the source and sink to the channel
```

```
34 a1.sources.r1.channels = c1  
35 a1.sinks.k1.channel = c1
```

组件官网

[avro source](#)

第五步：顺序启动

先启动下游 agent，目前数据是从 node02 上的 agent 发往 node03 上的 agent；

所以先启动 node03 的 agent

node03 机器启动 flume 进程

```
1 cd /kkb/install/apache-flume-1.9.0-bin/  
2 bin/flume-ng agent -c conf -f conf/avro-hdfs.conf -n a1 -  
Dflume.root.logger=INFO,console
```

node02 机器启动 flume 进程

```
1 cd /kkb/install/apache-flume-1.9.0-bin//  
2 bin/flume-ng agent -c conf -f conf/tail-avro-logger.conf -n a1 -  
Dflume.root.logger=INFO,console
```

node02 机器启 shell 脚本生成文件

```
1 cd /kkb/install/shells  
2 sh tail-file.sh
```

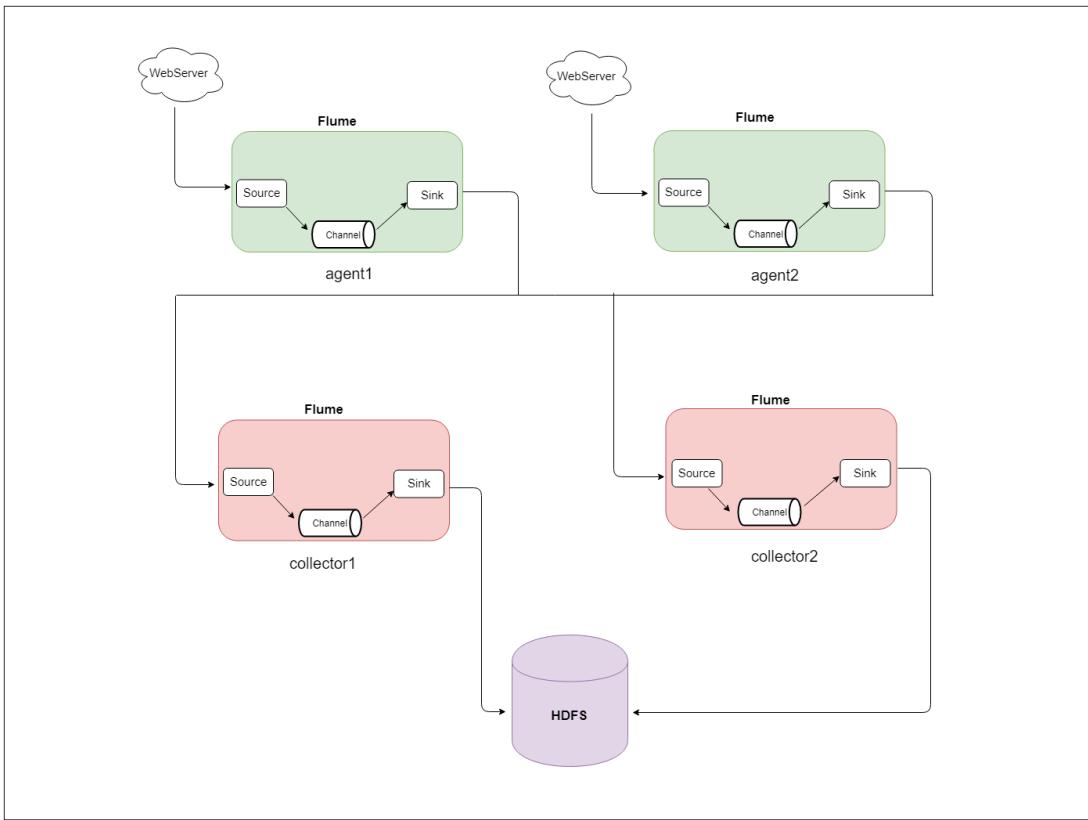
查看 HDFS 目录/avro/hdfs

更多 source 和 sink 组件

Flume 支持众多的 source 和 sink 类型，详细手册可参考官方文档

高可用 Failover 配置案例

在完成单点的 Flume 搭建后，下面我们搭建一个高可用的 Flume 集群，架构图如下所示：



图中，我们可以看出，Flume 的存储可以支持多种，这里只列举了 HDFS 和 Kafka（如：存储最新的一周日志，并给 Storm 系统提供实时日志流）。

1. 角色分配

- Flume 的 Agent 和 Collector 分布如下表所示：

名称	HOST	角色
Agent1	node01	Web Server
Collector1	node02	AgentMstr1
Collector2	node03	AgentMstr2

- 图中所示，Agent1 数据分别流入到 Collector1 和 Collector2，Flume NG 本身提供了 Failover 机制，可以自动切换和恢复。在上图中，有 3 个产生日志服务器分布在不同的机房，要把所有的日志都收集到一个集群中存储。下面我们开发配置 Flume NG 集群

2. node01 安装配置 flume 与拷贝文件脚本

- 将 node03 机器上面的 flume 安装包以及文件生产的两个目录拷贝到 node01 机器上面去
- node03 机器执行以下命令

```

1 cd /kpb/install
2 scp -r apache-flume-1.9.0-bin/ node01:$PWD
3 scp -r shells/ taillogs/ node01:$PWD

```

- node01 机器改 agent 的配置文件

```

1 cd /kpb/install/apache-flume-1.9.0-bin/conf/
2 vim agent.conf

```

- 内容如下

```

1 #agent1 name
2 agent1.channels = c1
3 agent1.sources = r1
4 agent1.sinks = k1 k2
5 ##set group
6 agent1.sinkgroups = g1
7
8 ##set channel
9 agent1.channels.c1.type = memory
10 agent1.channels.c1.capacity = 1000
11 agent1.channels.c1.transactionCapacity = 100
12
13 # 配置 source
14 agent1.sources.r1.channels = c1
15 agent1.sources.r1.type = exec
16 agent1.sources.r1.command = tail -F /kpb/install/taillogs/access_log
17 # interceptor 拦截器；与 source 结合，对 event 进行修改或丢弃
18 agent1.sources.r1.interceptors = i1 i2

```

```

19 # 静态拦截器在所有的 event 的 header 中，增加一个 kv 对，key 是下边属性 key 对应的值，value 是属性 value 对应的值
20 agent1.sources.r1.interceptors.i1.type = static
21 # 被创建的 header 的名字
22 agent1.sources.r1.interceptors.i1.key = Type
23 # 静态的值；key 与 value 对应
24 agent1.sources.r1.interceptors.i1.value = LOGIN
25 # timestamp 拦截器对 event 的 header 中增加 kv 对，key 是 timestamp，value 是对应的时间戳的值
26 agent1.sources.r1.interceptors.i2.type = timestamp
27
28 ## set sink1
29 agent1.sinks.k1.channel = c1
30 agent1.sinks.k1.type = avro
31 agent1.sinks.k1.hostname = node02
32 agent1.sinks.k1.port = 52020
33
34 ## set sink2
35 agent1.sinks.k2.channel = c1
36 agent1.sinks.k2.type = avro
37 agent1.sinks.k2.hostname = node03
38 agent1.sinks.k2.port = 52020
39
40 ##set sink group
41 agent1.sinkgroups.g1.sinks = k1 k2
42
43 ## sink processor 处理器；可用于 sink 的负载均衡或故障转移
44 agent1.sinkgroups.g1.processor.type = failover
45 # priority 值高的 sink，拥有较高的权限；并且必须是唯一不重复的
46 agent1.sinkgroups.g1.processor.priority.k1 = 10
47 agent1.sinkgroups.g1.processor.priority.k2 = 1
48 # maxpenalty 对于故障的节点最大的黑名单时间 (in millis 毫秒)
49 agent1.sinkgroups.g1.processor.maxpenalty = 10000

```

static interceptor 静态拦截器

timestamp 拦截器

sink processor 处理器

3. node02 与 node03 配置 flumecollection

- node02、node03 机器修改配置文件，内容相同

```
1 cd /kkb/install/apache-flume-1.9.0-bin/conf/  
2 vim collector.conf
```

- 内容如下

```
1 #set Agent name  
2 a1.sources = r1  
3 a1.channels = c1  
4 a1.sinks = k1  
5  
6 ##set channel  
7 a1.channels.c1.type = memory  
8 a1.channels.c1.capacity = 1000  
9 a1.channels.c1.transactionCapacity = 100  
10  
11 ## set source  
12 a1.sources.r1.type = avro  
13 a1.sources.r1.bind = 0.0.0.0  
14 a1.sources.r1.port = 52020  
15 a1.sources.r1.channels = c1  
16  
17 # 拦截器  
18 a1.sources.r1.interceptors = i1  
19 #a1.sources.r1.interceptors.i1.type = static  
20 #a1.sources.r1.interceptors.i1.key = Collector  
21 #a1.sources.r1.interceptors.i1.value = node02  
22 # 在 header 中添加的 kv 对的 key 默认是 host  
23 a1.sources.r1.interceptors.i1.type = host  
24 a1.sources.r1.interceptors.i1.hostHeader=hostname  
25  
26 ##set sink to hdfs  
27 a1.sinks.k1.type=hdfs  
28 a1.sinks.k1.hdfs.path= hdfs://node01:8020/flume/failover/%{hostname}  
29 a1.sinks.k1.hdfs.fileType=DataStream  
30 a1.sinks.k1.hdfs.writeFormat=TEXT  
31 a1.sinks.k1.hdfs.rollInterval=10  
32 a1.sinks.k1.channel=c1  
33 a1.sinks.k1.hdfs.filePrefix=%Y-%m-%d
```

4. 顺序启动命令

- node03 机器上面启动 flume

- cd /kkb/install/apache-flume-1.9.0-bin/
- bin/flume-ng agent -n a1 -c conf -f conf/collector.conf -
- Dflume.root.logger=DEBUG, console

- node02 机器上面启动 flume
 - cd /kkb/install/apache-flume-1.9.0-bin/
 - bin/flume-ng agent -n a1 -c conf -f conf/collector.conf -
 - Dflume.root.logger=DEBUG, console

- node01 机器上面启动 flume
 - cd /kkb/install/apache-flume-1.9.0-bin/
 - bin/flume-ng agent -n agent1 -c conf -f conf/agent.conf -
 - Dflume.root.logger=DEBUG, console

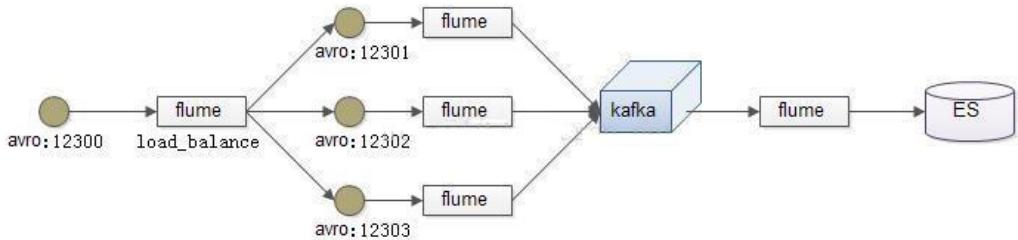
- node01 机器启动文件产生脚本
 - cd /kkb/install/shells
 - sh tail-file.sh

5. 测试 FAILOVER

- 演示:
- 然后去 hdfs 查看生成文件
- 将 node02 的 agent 停掉, 自动切换到 node03 上的 agent
- 再将 node02 的 agent 启动, 由于 node02 的优先级高, 自动切换回 node02 上的 agent
- 下面我们来测试下 Flume NG 集群的高可用 (故障转移)。场景如下: 我们在 Agent1 节点上传文件, 由于我们配置 Collector1 的权重比 Collector2 大, 所以 Collector1 优先采集并上传到存储系统。然后我们 kill 掉 Collector1, 此时有 Collector2 负责日志的采集上传工作, 之后, 我们手动恢复 Collector1 节点的 Flume 服务, 再次在 Agent1 上传文件, 发现 Collector1 恢复优先级别的采集工作。

Flume 的负载均衡 load balancer

- 负载均衡是用于解决一台机器(一个进程)无法解决所有请求而产生的一种算法。Load balancing Sink Processor 能够实现 load balance 功能, 如下图 Agent1 是一个路由节点, 负责将 Channel 暂存的 Event 均衡到对应的多个 Sink 组件上, 而每个 Sink 组件分别连接到一个独立的 Agent 上, 示例配置, 如下所示:



- 在此处我们通过三台机器来进行模拟 flume 的负载均衡
- 三台机器规划如下:
- node01: 采集数据, 发送到 node02 和 node03 机器上去
- node02: 接收 node01 的部分数据
- node03: 接收 node01 的部分数据

第一步：开发 node01 服务器的 flume 配置

- node01 服务器配置:
 - 1 cd /kkb/install/apache-flume-1.9.0-bin/conf/
 - 2 vim load_banlancer_client.conf
- 内容如下

```

1 # agent name
2 a1.channels = c1
3 a1.sources = r1
4 a1.sinks = k1 k2
5
6 # set sink group
7 a1.sinkgroups = g1
8
9 # set channel
10 a1.channels.c1.type = memory
11 a1.channels.c1.capacity = 1000
  
```

```

12 a1.channels.c1.transactionCapacity = 100
13
14 # set source
15 a1.sources.r1.channels = c1
16 a1.sources.r1.type = exec
17 a1.sources.r1.command = tail -F /kpb/install/taillogs/access.log
18
19 # set sink1
20 a1.sinks.k1.channel = c1
21 a1.sinks.k1.type = avro
22 a1.sinks.k1.hostname = node02
23 a1.sinks.k1.port = 52020
24
25 # set sink2
26 a1.sinks.k2.channel = c1
27 a1.sinks.k2.type = avro
28 a1.sinks.k2.hostname = node03
29 a1.sinks.k2.port = 52020
30
31 # set sink group
32 a1.sinkgroups.g1.sinks = k1 k2
33 # set load_balance
34 a1.sinkgroups.g1.processor.type = load_balance
35 # 如果 backoff 被开启, 则 sink processor 会屏蔽故障的 sink
36 a1.sinkgroups.g1.processor.backoff = true
37 # 默认是 round_robin, 还可以选择 random、FQCN
38 a1.sinkgroups.g1.processor.selector = round_robin
39 a1.sinkgroups.g1.processor.selector.maxTimeOut=10000

```

[load balance 处理器文档](#)

第二步：开发 node02、node03 服务器的 flume 配置

- 配置文件名、内容都一样
- node02、node03 都要这样修改

- 1 • cd /kpb/install/apache-flume-1.9.0-bin/conf/
- 2 • vim load_banlancer_server.conf

- 内容如下

```

1 #set Agent name
2 a1.sources = r1
3 a1.channels = c1
4 a1.sinks = k1
5
6 # set channel
7 a1.channels.c1.type = memory
8 a1.channels.c1.capacity = 1000
9 a1.channels.c1.transactionCapacity = 100
10
11 # set source
12 a1.sources.r1.type = avro
13 a1.sources.r1.bind = 0.0.0.0
14 a1.sources.r1.port = 52020
15 a1.sources.r1.channels = c1
16
17 #配置拦截器
18 a1.sources.r1.interceptors = i1 i2
19 a1.sources.r1.interceptors.i1.type = timestamp
20 a1.sources.r1.interceptors.i2.type = host
21 # header 的 key
22 a1.sources.r1.interceptors.i2.hostHeader=hostname
23 # 如果为 false, 那么现实主机名
24 a1.sources.r1.interceptors.i2.useIP=false
25
26 #set sink to hdfs
27 a1.sinks.k1.channel = c1
28 a1.sinks.k1.type=hdfs
29 a1.sinks.k1.hdfs.path=hdfs://node01:8020/loadbalance/logs/%{hostname}
30 a1.sinks.k1.hdfs.filePrefix=%Y-%m-%d
31 a1.sinks.k1.hdfs.round = true
32 a1.sinks.k1.hdfs.roundValue = 10
33 a1.sinks.k1.hdfs.roundUnit = minute
34 a1.sinks.k1.hdfs.rollInterval = 60
35 a1.sinks.k1.hdfs.rollSize = 200
36 a1.sinks.k1.hdfs.rollCount = 10
37 a1.sinks.k1.hdfs.batchSize = 100
38 a1.sinks.k1.hdfs.fileType = DataStream

```

第三步：准备启动 flume 服务

- 启动 node03 的 flume 服务

- cd /kkb/install/apache-flume-1.9.0-bin/
 - 1 • bin/flume-ng agent -n a1 -c conf -f
 - 2 conf/load_banlancer_server.conf -Dflume.root.logger=DEBUG, console
-
- 启动 node02 的 flume 服务
- cd /kkb/install/apache-flume-1.9.0-bin/
 - 1 • bin/flume-ng agent -n a1 -c conf -f
 - 2 conf/load_banlancer_server.conf -Dflume.root.logger=DEBUG, console
-
- 启动 node01 的 flume 服务
- cd /kkb/install/apache-flume-1.9.0-bin/
 - 1 • bin/flume-ng agent -n a1 -c conf -f
 - 2 conf/load_banlancer_client.conf -Dflume.root.logger=DEBUG, console

第四步：node01 服务器运行脚本产生数据

```
1 cd /kkb/install/shells  
2 sh tail-file.sh
```

- 结论：
 - 如果将 node02 上的 agent 杀掉；
 - 再重启，然后 node02 的 agent 仍然可以采集到新的数据

Flume 实现自定义 Source 和 Sink 组件

Flume 自定义 source

- 官方提供的 source 类型已经很多，但是有时候并不能满足实际开发当中的需求，此时我们就需要根据实际需求自定义某些 source。
- 如：实时监控 MySQL，从 MySQL 中获取数据传输到 HDFS 或者其他存储框架，所以此时需要我们自己实现 MySQLSource。
- 官方也提供了自定义 source 的接口：
- [官网文档](#)

需求说明：

- 自定义 flume 的 source，实现从 mysql 数据库当中获取数据，将数据打印到控制台上面来

代码开发步骤：

```
public class MySource extends AbstractSource implements Configurable, PollableSource {
    private String myProp;

    @Override
    public void configure(Context context) {
        String myProp = context.getString("myProp", "defaultValue");

        // Process the myProp value (e.g. validation, convert to another type, ...)

        // Store myProp for later retrieval by process() method
        this.myProp = myProp;
    }

    @Override
    public void start() {
        // Initialize the connection to the external client
    }

    @Override
    public void stop () {
        // Disconnect from external client and do any additional cleanup
        // (e.g. releasing resources or nulling-out field values) ..
    }

    @Override
    public Status process() throws EventDeliveryException {
        Status status = null;
```

1. 根据官方说明自定义 mysqlsource 需要继承 AbstractSource 类并实现 Configurable 和 PollableSource 接口。
2. 实现对应的方法
3. configure(Context context)
4. 初始化 context
5. process()
6. 从 mysql 表中获取数据，然后把数据封装成 event 对象写入到 channel，该方法被一直调用
 1. stop()
 2. 关闭相关资源

第一步：创建 mysql 数据库表

```
-- 创建一个数据库
CREATE DATABASE IF NOT EXISTS mysqlsource DEFAULT CHARACTER SET utf8;

use mysqlsource;

-- 创建一个表，保存从目标表已经读取数据的位置信息
CREATE TABLE mysqlsource.flume_meta (
    source_tab varchar(255) NOT NULL,
    currentIndex varchar(255) NOT NULL,
    PRIMARY KEY (source_tab)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- 插入数据
insert      into      mysqlsource.flume_meta(source_tab,currentIndex)      values
('student','1');

select * from mysqlsource.flume_meta;

-- 创建要拉取数据的表
CREATE TABLE mysqlsource.student(
    id int(11) NOT NULL AUTO_INCREMENT,
    name varchar(255) NOT NULL,
    PRIMARY KEY (id)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;

-- 向 student 表中添加测试数据
insert      into      mysqlsource.student(id,name)      values
(1,'zhangsan'),(2,'lisi'),(3,'wangwu'),(4,'zhaoliu');

select * from mysqlsource.student;
```

第二步：创建 maven 工程，添加依赖

```
1 <dependencies>
2   <dependency>
3     <groupId>org.apache.flume</groupId>
4     <artifactId>flume-ng-core</artifactId>
5     <version>1.9.0</version>
```

```
6   </dependency>
7
8   <dependency>
9     <groupId>mysql</groupId>
10    <artifactId>mysql-connector-java</artifactId>
11    <version>5.1.38</version>
12  </dependency>
13
14  <dependency>
15    <groupId>org.apache.commons</groupId>
16    <artifactId>commons-lang3</artifactId>
17    <version>3.6</version>
18  </dependency>
19 </dependencies>
20
21 <build>
22   <plugins>
23     <plugin>
24       <groupId>org.apache.maven.plugins</groupId>
25       <artifactId>maven-compiler-plugin</artifactId>
26       <version>3.0</version>
27       <configuration>
28         <source>1.8</source>
29         <target>1.8</target>
30         <encoding>UTF-8</encoding>
31         <!-- <verbal>true</verbal>-->
32       </configuration>
33     </plugin>
34     <plugin>
35       <groupId>org.apache.maven.plugins</groupId>
36       <artifactId>maven-shade-plugin</artifactId>
37       <version>3.1.1</version>
38       <executions>
39         <execution>
40           <phase>package</phase>
41           <goals>
42             <goal>shade</goal>
43           </goals>
44           <configuration>
45             <filters>
46               <filter>
47                 <artifact>*:*</artifact>
48               <excludes>
49                 <exclude>META-INF/*.SF</exclude>
50                 <exclude>META-INF/*.DSA</exclude>
51                 <exclude>META-INF/*.RSA</exclude>
```

```
50      </excludes>
51      </filter>
52      </filters>
53      <transformers>
54          <transformer
55              implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
56              <mainClass></mainClass>
57          </transformer>
58      </transformers>
59      </configuration>
60      </execution>
61      </executions>
62  </plugin>
63 </plugins>
64 </build>
```

第三步：添加配置文件

- 在我们工程的 resources 目录下，添加 jdbc.properties
- 以下属性的值，根据自己的实际情况修改

```
1 dbDriver=com.mysql.jdbc.Driver
2 dbUrl=jdbc:mysql://node03:3306/mysqlsource?useUnicode=true&characterEncoding=ut
3 f-8
4 dbUser=root
4 dbPassword=123456
```

第四步：代码开发

定义查询 mysql 的工具类

```
package com.kkb.flume.source;

import org.apache.flume.Context;
import org.apache.flume.conf.ConfigurationException;
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;

import java.sql.*;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;

public class QueryMysql {
    private static final Logger LOG =
        LoggerFactory.getLogger(QueryMysql.class);

    private int runQueryDelay,           //两次查询的时间间隔
               startFrom,             //开始 id
               currentIndex,          //当前 id
               recordSixe = 0,         //每次查询返回结果的条数
               maxRow;                //每次查询的最大条数

    private String table,              //要操作的表
                 columnsToSelect, //用户传入的查询的列
                 customQuery,       //用户传入的查询语句
                 query,             //构建的查询语句
                 defaultCharsetResultSet; //编码集

    //上下文，用来获取配置文件
    private Context context;

    //为定义的变量赋值（默认值），可在 flume 任务的配置文件中修改
    private static final int DEFAULT_QUERY_DELAY = 10000;
    private static final int DEFAULT_START_VALUE = 0;
    private static final int DEFAULT_MAX_ROWS = 2000;
    private static final String DEFAULT_COLUMNS_SELECT = "*";
    private static final String DEFAULT_CHARSET_RESULTSET = "UTF-8";

    private static Connection conn = null;
    private static PreparedStatement ps = null;
    //连接 mysql 的 url、用户名、密码
    private static String connectionURL,      connectionUserName,
                       connectionPassword;

    //加载静态资源
    static {
        Properties p = new Properties();

```

```

        try {
            p.load(QueryMysql.class.getClassLoader().getResourceAsStream("jdbc.properties"));
            connectionURL = p.getProperty("dbUrl");
            connectionUserName = p.getProperty("dbUser");
            connectionPassword = p.getProperty("dbPassword");
            Class.forName(p.getProperty("dbDriver"));
        } catch (Exception e) {
            LOG.error(e.toString());
        }
    }

    //获取 JDBC 连接 Connection
    private static Connection initConnection(String url, String user, String pw) {
        try {
            Connection conn = DriverManager.getConnection(url, user, pw);
            if (conn == null)
                throw new SQLException();
            return conn;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }

    //构造方法
    public QueryMysql(Context context) throws ParseException {
        //初始化上下文
        this.context = context;

        /**
         * 有默认值参数：获取 flume 任务配置文件中的参数，读不到的采用默认
         * 值
         */
        this.columnsToSelect = context.getString("columns.to.select",
        DEFAULT_COLUMNS_SELECT);
        this.runQueryDelay = context.getInteger("run.query.delay",
        DEFAULT_QUERY_DELAY);
        this.startFrom = context.getInteger("start.from",
        DEFAULT_START_VALUE);
        this.defaultCharsetResultSet =
        context.getString("default.charset.resultset", DEFAULT_CHARSET_RESULTSET);
    }
}

```

```

//无默认值参数：获取 flume 任务配置文件中的参数
this.table = context.getString("table");
this.customQuery = context.getString("custom.query");

connectionURL = context.getString("connection.url");
connectionUserName = context.getString("connection.user");
connectionPassword = context.getString("connection.password");
conn = initConnection(connectionURL, connectionUserName,
connectionPassword);

//校验相应的配置信息，如果没有默认值的参数也没赋值，抛出异常
checkMandatoryProperties();
//从要读取数据的表中，获取之前数据已经读到 id 为几？
currentIndex = getStatusDBIndex(startFrom);
//构建查询语句
query = buildQuery();
}

//校验相应的配置信息（表，查询语句以及数据库连接的参数）
private void checkMandatoryProperties() {
    if (table == null) {
        throw new ConfigurationException("property table not set");
    }
    if (connectionURL == null) {
        throw new ConfigurationException("connection.url property
not set");
    }
    if (connectionUserName == null) {
        throw new ConfigurationException("connection.user property
not set");
    }
    if (connectionPassword == null) {
        throw new ConfigurationException("connection.password
property not set");
    }
}

//构建 sql 语句
private String buildQuery() {
    String sql = "";
    //获取当前 id
    currentIndex = getStatusDBIndex(startFrom);
    LOG.info(currentIndex + "");
}

```

```

        if (customQuery == null) {
            sql = "SELECT " + columnsToSelect + " FROM " + table;
        } else {
            sql = customQuery;
        }
        StringBuilder execSql = new StringBuilder(sql);
        //以 id 作为 offset
        if (!sql.contains("where")) {
            execSql.append(" where ");
            execSql.append("id").append(">").append(currentIndex);
            return execSql.toString();
        } else {
            int length = execSql.toString().length();
            return execSql.toString().substring(0, length -
String.valueOf(currentIndex).length()) + currentIndex;
        }
    }

    //执行查询
    public List<List<Object>> executeQuery() {
        try {
            //每次执行查询时都要重新生成 sql, 因为 id 不同
            customQuery = buildQuery();
            //存放结果的集合
            List<List<Object>> results = new ArrayList<>();
            if (ps == null) {
                //初始化 PreparedStatement 对象
                ps = conn.prepareStatement(customQuery);
            }
            ResultSet result = ps.executeQuery(customQuery);
            while (result.next()) {
                //存放一条数据的集合 (多个列)
                List<Object> row = new ArrayList<>();
                //将返回结果放入集合
                for (int i = 1; i <=
result.getMetaData().getColumnCount(); i++) {
                    row.add(result.getObject(i));
                }
                results.add(row);
            }
            LOG.info("execSql:" + customQuery + "\nresultSize:" +
results.size());
            return results;
        } catch (SQLException e) {

```

```
        LOG.error(e.toString());
        // 重新连接
        conn = initConnection(connectionURL, connectionUserName,
connectionPassword);
    }
    return null;
}
```

//将结果集转化为字符串，每一条数据是一个 list 集合，将每一个小的 list 集合转化为字符串

```
public List<String> getAllRows(List<List<Object>> queryResult) {
    List<String> allRows = new ArrayList<>();
    if (queryResult == null || queryResult.isEmpty())
        return allRows;
    StringBuilder row = new StringBuilder();
    for (List<Object> rawRow : queryResult) {
        Object value = null;
        for (Object aRawRow : rawRow) {
            value = aRawRow;
            if (value == null) {
                row.append(",");
            } else {
                row.append(aRawRow.toString()).append(",");
            }
        }
        allRows.add(row.toString());
        row = new StringBuilder();
    }
    return allRows;
}
```

//更新 offset 元数据状态，每次返回结果集后调用。必须记录每次查询的 offset 值，为程序中断续跑数据时使用，以 id 为 offset

```
public void updateOffset2DB(int size) {
    //以 source_tab 做为 KEY，如果不存在则插入，存在则更新（每个源表对应一条记录）
    String sql = "insert into flume_meta(source_tab,currentIndex)
VALUES('"
        + this.table
        + "','" + (recordSixe += size)
        + "')      on      DUPLICATE      key      update
source_tab=values(source_tab),currentIndex=values(currentIndex)";
    LOG.info("updateStatus Sql:" + sql);
```

```
        execSql(sql);
    }

    //执行 sql 语句
    private void execSql(String sql) {
        try {
            ps = conn.prepareStatement(sql);
            LOG.info("exec::" + sql);
            ps.execute();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    //获取当前 id 的 offset
    private Integer getStatusDBIndex(int startFrom) {
        //从 flume_meta 表中查询出当前的 id 是多少
        String dbIndex = queryOne("select currentIndex from flume_meta
where source_tab='"
+ table + "'");

        if (dbIndex != null) {
            return Integer.parseInt(dbIndex);
        }
        //如果没有数据，则说明是第一次查询或者数据表中还没有存入数据，返回
        最初传入的值
        return startFrom;
    }

    //查询一条数据的执行语句(当前 id)
    private String queryOne(String sql) {
        ResultSet result = null;
        try {
            ps = conn.prepareStatement(sql);
            result = ps.executeQuery();
            while (result.next()) {
                return result.getString(1);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }

    //关闭相关资源
    public void close() {
```

```
        try  {
            ps.close();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    int getCurrentIndex() {
        return currentIndex;
    }

    void setCurrentIndex(int newValue) {
        currentIndex = newValue;
    }

    public int getRunQueryDelay() {
        return runQueryDelay;
    }

    String getQuery() {
        return query;
    }

    String getConnectionURL() {
        return connectionURL;
    }

    private boolean isCustomQuerySet() {
        return (customQuery != null);
    }

    Context getContext() {
        return context;
    }

    public String getConnectionUserName() {
        return connectionUserName;
    }

    public String getConnectionPassword() {
        return connectionPassword;
    }
```

```

        String getDefaultCharsetResultSet() {
            return defaultCharsetResultSet;
        }
    }
}

```

自定义 *mysqlSource* 类

```

1 package com.kkb.flume.source;
2
3 import org.apache.flume.Context;
4 import org.apache.flume.Event;
5 import org.apache.flume.EventDeliveryException;
6 import org.apache.flume.PollableSource;
7 import org.apache.flume.conf.Configurable;
8 import org.apache.flume.event.SimpleEvent;
9 import org.apache.flume.source.AbstractSource;
10 import org.slf4j.Logger;
11 import org.slf4j.LoggerFactory;
12
13 import java.text.ParseException;
14 import java.util.ArrayList;
15 import java.util.HashMap;
16 import java.util.List;
17
18 public class MySqlSource extends AbstractSource implements Configurable, PollableSource {
19
20     //打印日志
21     private static final Logger LOG = LoggerFactory.getLogger(MySqlSource.class);
22     //自定义工具类 QueryMysql 的对象
23     private QueryMysql sqlSourceHelper;
24
25     @Override
26     public long getBackOffSleepIncrement() {
27         return 0;
28     }
29
30     @Override
31     public long getMaxBackOffSleepInterval() {
32         return 0;
33     }
34

```

```
35  @Override
36  public void configure(Context context) {
37      //初始化
38      try {
39          sqlSourceHelper = new QueryMysql(context);
40      } catch (ParseException e) {
41          e.printStackTrace();
42      }
43  }
44
45  /**
46      * 接受 mysql 表中的数据
47      *
48      * @return
49      * @throws EventDeliveryException
50      */
51  @Override
52  public PollableSource.Status process() throws EventDeliveryException {
53      try {
54          //查询数据表
55          List<List<Object>> result = sqlSourceHelper.executeQuery();
56          //存放 event 的集合
57          List<Event> events = new ArrayList<>();
58          //存放 event 头集合
59          HashMap<String, String> header = new HashMap<>();
60          //如果有返回数据，则将数据封装为 event
61          if (!result.isEmpty()) {
62              List<String> allRows = sqlSourceHelper.getAllRows(result);
63              Event event = null;
64              for (String row : allRows) {
65                  event = new SimpleEvent();
66                  event.setBody(row.getBytes());
67                  event.setHeaders(header);
68                  events.add(event);
69              }
70              //将 event 写入 channel
71              this.getChannelProcessor().processEventBatch(events);
72              //更新数据表中的 offset 信息
73              sqlSourceHelper.updateOffset2DB(result.size());
74          }
75          //等待时长
76          Thread.sleep(sqlSourceHelper.getRunQueryDelay());
77          return Status.READY;
78      } catch (InterruptedException e) {
```

```
79     LOG.error("Error procesing row", e);
80     return Status.BACKOFF;
81   }
82 }
83
84 @Override
85 public synchronized void stop() {
86   LOG.info("Stopping sql source {} ...", getName());
87   try {
88     //关闭资源
89     sqlSourceHelper.close();
90   } finally {
91     super.stop();
92   }
93 }
94 }
```

第五步：打包上传到 flume 的 lib 目录下

- 将我们开发的代码，打成 jar 包上传到 flume 的 lib 目录下
- 需要将 mysql 的 connectorjar 包，拷贝到 flume 的 lib 目录下

```
1 cp /kkb/install/apache-hive-3.1.2/lib/mysql-connector-java-5.1.38.jar
  /kkb/install/apache-flume-1.9.0-bin/lib/
```

第六步：开发 flume 的配置文件

- 开发 flume 的配置文件

```
1 cd /kkb/install/apache-flume-1.9.0-bin/conf/
2 vim mysqlsource.conf
```

- 内容如下
- source 中的内容根据自己的实际情况修改；如 MySqlSource 的 FQCN

```
1   • # Name the components on this agent
2   • a1.sources = r1
3   • a1.sinks = k1
4   • a1.channels = c1
```

```
5   •
6   • # set source
7   • # 写上自己的自定义 source 类的 FQCN
8   • a1.sources.r1.type = com.kkb.flume.source.MySqlSource
9   • a1.sources.r1.connection.url = jdbc:mysql://node03:3306/mysqlsource
10  • a1.sources.r1.connection.user = root
11  • a1.sources.r1.connection.password = 123456
12  • a1.sources.r1.table = student
13  • a1.sources.r1.columns.to.select = *
14  • a1.sources.r1.start.from=0
15  • a1.sources.r1.run.query.delay=3000
16  •
17  • # Describe the channel
18  • a1.channels.c1.type = memory
19  • a1.channels.c1.capacity = 1000
20  • a1.channels.c1.transactionCapacity = 100
21  •
22  • # Describe the sink
23  • a1.sinks.k1.type = logger
24  •
25  • # Bind the source and sink to the channel
26  • a1.sources.r1.channels = c1
27  • a1.sinks.k1.channel = c1
```

第七步：启动 flume

```
1 cd /kkb/install/apache-flume-1.9.0-bin/
1 bin/flume-ng agent -n a1 -c conf -f conf/mysqlsource.conf -
2 Dflume.root.logger=info,console
```

第八步：观察 console 结果

```
2020-11-23 16:53:50,932 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 32 2C 6C 69 73 69 2C
2,lisi,}
2020-11-23 16:53:50,932 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 33 2C 77 61 6E 67 77 75 2C
3,wangwu, }
2020-11-23 16:53:50,932 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 34 2C 7A 68 61 6F 6C 69 75 2C
4,zhaoliu, }
```

再次向 mysql 表插入数据

```
1 -- 向 student 表中添加测试数据
2 insert      into          mysqlsource.student(id, name)           values
2 (5, 'tianqi'), (6, 'jack'), (8, 'rousi');

2020-11-23 16:57:54,152 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 35 2C 74 69 61 6E 71 69 2C
5,tianqi, }
2020-11-23 16:57:54,152 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 36 2C 6A 61 63 6B 2C
6,jack, }
2020-11-23 16:57:54,152 (PollableSourceRunner-MySQLSource-r1) [INFO - com.kkb.flume.source.QueryMysql.execSql(QueryMysql.java:208)] exec:::insert into flume_meta(source_tab,currentIndex) VALUES('student','7')
on DUPLICATE key update source_tab=values(source_tab),currentIndex=values(currentIndex)
2020-11-23 16:57:54,152 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:95)] Event: { headers:{} body: 38 2C 72 6F 75 73 69 2C
8,rousi, }
```

Flume 自定义 sink

需求说明:

- 官方提供的 sink 类型已经很多，但是有时候并不能满足实际开发当中的需求，此时我们就需要根据实际需求自定义某些 sink。如：需要把接受到的数据按照规则进行过滤之后写入到某张 mysql 表中，所以此时需要我们自己实现 MySQLSink。
- 官方也提供了自定义 sink 的接口：
- [官网文档](#)

代码开发步骤:

1. 根据官方说明自定义 MysqLSink 需要继承 AbstractSink 类并实现 Configurable 接口
2. 实现对应的方法
 1. configure(Context context)

2. 配置相关
3. start()
4. 启动准备操作
5. process()
6. 从 channel 获取数据，然后解析之后，保存在 mysql 表中
7. stop()
8. 关闭相关资源

第一步：创建 mysql 数据库表

```
1 -- 创建一个数据库
2 CREATE DATABASE IF NOT EXISTS mysqlsource DEFAULT CHARACTER SET utf8;
3
4 USE mysqlsource;
5
6 -- 创建一个表，用户保存拉取目标表位置的信息
7 CREATE TABLE mysqlsource.flume2mysql (
8   id INT(11) NOT NULL AUTO_INCREMENT,
9   createTime VARCHAR(64) NOT NULL,
10  content VARCHAR(255) NOT NULL,
11  PRIMARY KEY (id)
12 ) ENGINE=INNODB DEFAULT CHARSET=utf8;
```

第二步：定义 mysqlSink 类

```
1 package com.kkb.flume.sink;
2
3 import org.apache.flume.conf.Configurable;
4 import org.apache.flume.*;
5 import org.apache.flume.sink.AbstractSink;
6
7 import java.sql.Connection;
8 import java.sql.DriverManager;
9 import java.sql.PreparedStatement;
10 import java.sql.SQLException;
```

```
11 import java.text.SimpleDateFormat;
12 import java.util.Date;
13
14 /**
15  * 自定义 MysqlSink
16 */
17 public class MysqlSink extends AbstractSink implements Configurable {
18     private String mysqlurl = "";
19     private String username = "";
20     private String password = "";
21     private String tableName = "";
22
23     Connection con = null;
24
25     @Override
26     public Status process() {
27         Status status = null;
28         // Start transaction 获得 Channel 对象
29         Channel ch = getChannel();
30         Transaction txn = ch.getTransaction();
31         txn.begin();
32
33         try {
34             Event event = ch.take();
35
36             if (event != null) {
37                 //获取 body 中的数据
38                 String body = new String(event.getBody(), "UTF-8");
39
40                 //如果日志中有以下关键字的不需要保存，过滤掉
41                 if (body.contains("delete") || body.contains("drop") || body.contains("alert")) {
42                     status = Status.BACKOFF;
43                 } else {
44
45                     //存入 Mysql
46                     SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
47                     String createtime = df.format(new Date());
48
49                     PreparedStatement stmt = con.prepareStatement("insert into " + tableName + " (createtime, content) values (?, ?)");
50                     stmt.setString(1, createtime);
51                     stmt.setString(2, body);
52                     stmt.execute();
53                     stmt.close();
54                     status = Status.READY;
55                 }
56             }
57         } catch (Exception e) {
58             e.printStackTrace();
59         }
60     }
61 }
```

```
55     }
56 } else {
57     status = Status.BACKOFF;
58 }
59
60     txn.commit();
61 } catch (Throwable t) {
62     txn.rollback();
63     t.getCause().printStackTrace();
64     status = Status.BACKOFF;
65 } finally {
66     txn.close();
67 }
68
69     return status;
70 }
71
72 /**
73 * 获取配置文件中指定名称的参数值
74 *
75 * @param context
76 */
77 @Override
78 public void configure(Context context) {
79     /*
80         getString 的参数与 agent 配置文件中 sink 组件的自定义的属性名一一对应
81         如 context.getString("mysqlurl");对应
82         a1.sinks.k1.mysqlurl=jdbc:mysql://node03:3306/mysqlsource?useSSL=false
83     */
84     mysqlurl = context.getString("mysqlurl");
85     username = context.getString("username");
86     password = context.getString("password");
87     tableName = context.getString("tablename");
88 }
89
90 @Override
91 public synchronized void start() {
92     try {
93         //初始化数据库连接
94         con = DriverManager.getConnection(mysqlurl, username, password);
95         super.start();
96         System.out.println("finish start");
97     } catch (Exception ex) {
98         ex.printStackTrace();

```

```
99      }
100     }
101
102     @Override
103     public synchronized void stop() {
104         try {
105             con.close();
106         } catch (SQLException e) {
107             e.printStackTrace();
108         }
109         super.stop();
110     }
111
112 }
```

第三步：代码打包上传

- 将我们的代码打成 jar 包上传到 flume 的 lib 目录下

第四步：开发 flume 的配置文件

- 1 • cd /kkb/install/apache-flume-1.9.0-bin/conf/
- 2 • vim mysqlsink.conf

- 内容如下
- 根据自己的实际情况修改 sink

```
1 a1.sources = r1
2 a1.sinks = k1
3 a1.channels = c1
4
5 #配置 source
6 a1.sources.r1.type = exec
7 a1.sources.r1.command = tail -F /kkb/install/flumeData/data.log
8 a1.sources.r1.channels = c1
9
10 #配置 channel
11 a1.channels.c1.type = memory
```

```
12 a1.channels.c1.capacity = 1000
13 a1.channels.c1.transactionCapacity = 100
14
15 #配置 sink
16 a1.sinks.k1.channel = c1
17 a1.sinks.k1.type = com.kkb.flume.sink.MysqlSink
18 a1.sinks.k1.mysqlurl=jdbc:mysql://node03:3306/mysqlsource?useSSL=false
19 a1.sinks.k1.username=root
20 a1.sinks.k1.password=123456
21 a1.sinks.k1.tablename=flume2mysql
```

第五步：启动 flume

```
1 cd /kkb/install/apache-flume-1.9.0-bin/
2 bin/flume-ng agent -n a1 -c conf -f conf/mysqlsink.conf -
Dflume.root.logger=info,console
```

第六步：创建文件验证数据进入 mysql

创建文件，验证数据进入 mysql 数据库

```
1 mkdir -p /kkb/install/flumeData
2 echo "helloworld" >> /kkb/install/flumeData/data.log
3 echo "helloworld 02" >> /kkb/install/flumeData/data.log
```

查看 mysql 表数据

```
mysql> select * from flume2mysql;
+----+-----+-----+
| id | createTime           | content      |
+----+-----+-----+
| 1  | 2020-11-23 17:13:02 | helloworld   |
| 2  | 2020-11-23 17:13:58 | helloworld 02 |
+----+-----+-----+
2 rows in set (0.00 sec)
```

提示：Flume 命令中的 agent 的名称别写错了，后台执行加上 nohup ... &