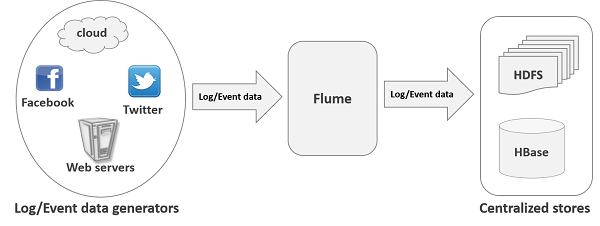
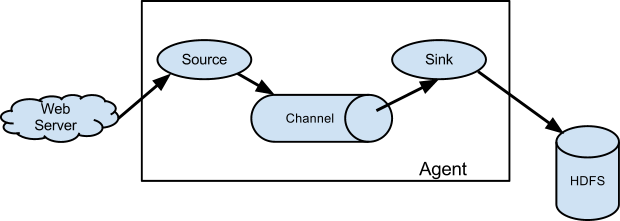
# Apache Flume源码分析

Apache Flume是可以收集日志、事件等数据资源，并将这些数据从各种数据源集中起来存储的工具/服务，其结构如下图：



其设计的原理是基于数据流，将日志数据从各种网站服务器上汇聚起来存储到HDFS/HBase等集中式存储器中。其系统架构如下：



其核心概念如下：

1. Events，数据传输的基本单位，其包括两个部分：header和body

* header，k/v结构数据，用来数据传输路由决策和其他结构化数据，如事件的时间戳或者数据源的服务器主机名，可以类比于HTTP Header的功能
* Body，字节数组，传输的实际数据

Flume为不同的Source生成的Evevent添加不同的Header

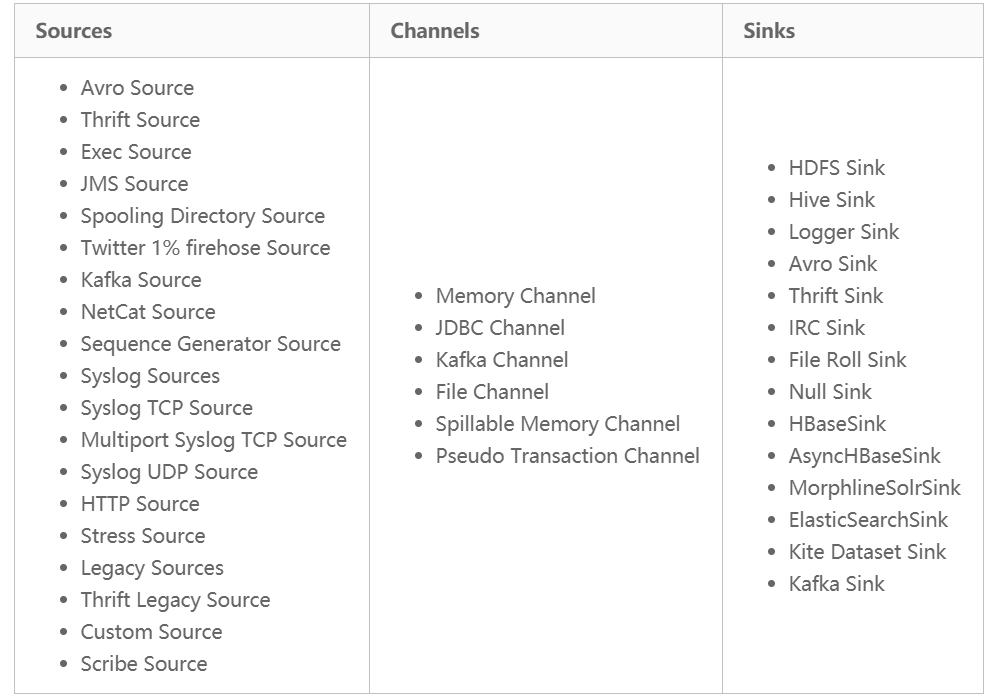
1. Agent, Flume代理，唯一启动的JVM进程，内部承载了从外部数据源事件流转到下一个目的地的过程，内部包含了Source/Channel/Sink
2. Source，从数据源收集数据，并传递给Channel，目前Source支持多种收集方式

* Avro，侦听Avro端口并从外部Avro客户端接收事件
* Spooling Directory Source，监控配置目录中新增文件，并从文件中读取数据
* NetCat Source，监听指定端口，将接收到的数据每一行转换为一个事件
* HTTP，接收HTTP GET/POST请求作为Flume事件
* Kafka Source，其作为Kafka消费者，从Kafka主题中读取消息
* 其他：Thrift,Exec,JMX,SysLog等

1. Channel，Event中转通道，临时存储Source组件传递过来的Event，其类似于消息队列，用于连接Source和Event
2. Sink，将Event从Channle取出后写入目标存储中，目前支持多种Sink

* HDFS Sink，写入到hadoop文件系统中，支持文件创建/序列文件等
* Hive Sink，将分割文件或者JSON数据的Events直接传送到Hive表或者分区中
* Logger Sink/File Roll Sink
* Avro /Thrift Sink/IRC Sink，写入服务端口
* HBase Sink，将数据写入到HBase中
* Kafka Sink，导出数据到Kafka Topic中
* 其他，ES/HTTP等，用户可以自定义Sink

Flume支持的Source、Sink及Channel类型如下：



可以根据需求搭配组合使用。

Source: https://www.cnblogs.com/swordfall/p/8254271.html

# Flume启动

## 1.1 Flume示例

下面是Flume的示例：



Flume的启动命令如下：

*# bin/flume-ng agent -n a1 -c . -f conf/flume.conf -Dflume.root.logger=DEBUG,console*

在flume.conf中配置信息如下：

*// Agent中各组件名*

*a1.channels = ch1*

*a1.sinks = sink1*

*a1.sources = s1*

*// Source的配置，扫描文件并上传*

*a1.sources.s1.type = spooldir*

*a1.sources.s1.channels = ch1*

*a1.sources.s1.spoolDir = /opt/flume/data*

*//定义channel类型*

*a1.channels.ch1.type = memory*

*a1.channels.ch1.capacity = 1000*

*//描述，Sink，写入到hdfs的/tmp目录下，文件前缀test.txt*

*a1.sinks.sink1.type=hdfs*

*a1.sinks.sink1.hdfs.path=/tmp*

*a1.sinks.sink1.hdfs.filePrefix=test.txt*

*a1.sinks.sink1.channel=ch1*

在flume.conf中配置了Source/Channel/Sink各组件。

## **1.2 Flume入口**

在flume-ng的shell启动脚本中会调用java启动flume Agent，如下所示：

*$EXEC $JAVA\_HOME/bin/java*

*$JAVA\_OPTS $FLUME\_JAVA\_OPTS "*

*${arr\_java\_props[@]}"*

*-cp "$FLUME\_CLASSPATH" \*

*-Djava.library.path=$FLUME\_JAVA\_LIBRARY\_PATH*

*"$FLUME\_APPLICATION\_CLASS" $\* //org.apache.flume.node.Application*

命令执行后，启动进程Application，如下所示：

*# jps*

*20616 Application*

启动入口为：org.apache.flume.node.Application#main，其执行如下：

*List<LifecycleAware> components = Lists.newArrayList();*

*if (reload) { //如果配置reload，在间隔时间内重新加载配置*

*EventBus eventBus = new EventBus(agentName + "-event-bus");*

*PollingPropertiesFileConfigurationProvider configurationProvider =*

*new PollingPropertiesFileConfigurationProvider(*

*agentName, configurationFile, eventBus, 30);*

*components.add(configurationProvider); //监控文件变化的组件Component*

*application = new Application(components);*

*eventBus.register(application);*

*} else {*

*PropertiesFileConfigurationProvider configurationProvider =*

*new PropertiesFileConfigurationProvider(agentName, configurationFile);*

*application = new Application();*

*application.handleConfigurationEvent(configurationProvider.getConfiguration());*

*}*

*application.start();*

1. **handleConfigurationEvent**

在Application.start启动之前调用handleConfigurationEvent，在该过程中

*public void handleConfigurationEvent(MaterializedConfiguration conf) {*

*try {*

*lifecycleLock.lockInterruptibly();*

*stopAllComponents(); //由于配置文件会动态加载这一特性，每次加载之前都要先把所有组件停掉，然后去加载最新配置文件中的配置*

*startAllComponents(conf);*

*}.....*

*}*

1. **startAllComponents**

startAllComponents，配置所有的组件，顺序依次是channel,sink和source，其源码执行如下：

*private void startAllComponents(MaterializedConfiguration materializedConfiguration) {*

*......*

*//启动所有Channel*

*for (Channel ch : materializedConfiguration.getChannels().values()) {*

*while (ch.getLifecycleState() != LifecycleState.START*

*&& !supervisor.isComponentInErrorState(ch)) {}*

*//启动SinkRunner*

*for (Entry<String, SinkRunner> entry : materializedConfiguration.getSinkRunners().entrySet()) {*

*try {*

*supervisor.supervise(entry.getValue(),*

*new SupervisorPolicy.AlwaysRestartPolicy(), LifecycleState.START);*

*} }*

*//启动SourceRunner*

*for (Entry<String, SourceRunner> entry :*

*materializedConfiguration.getSourceRunners().entrySet()) {*

*try {*

*supervisor.supervise(entry.getValue(),*

*new SupervisorPolicy.AlwaysRestartPolicy(), LifecycleState.START);*

*} }}*

1. **Application.start**

启动其他所有组件，包括PollingPropertiesFileConfigurationProvider，启动过程如下：

*public void start() {*

*try {*

*for (LifecycleAware component : components) {*

*supervisor.supervise(component,*

*new SupervisorPolicy.AlwaysRestartPolicy(), LifecycleState.START);*

*}} }*

components的初始化在main中执行，如下：

*List<LifecycleAware> components = Lists.newArrayList();*

*if (reload) {*

*EventBus eventBus = new EventBus(agentName + "-event-bus");*

*PollingPropertiesFileConfigurationProvider configurationProvider =*

*new PollingPropertiesFileConfigurationProvider(*

*agentName, configurationFile, eventBus, 30);*

*components.add(configurationProvider);*

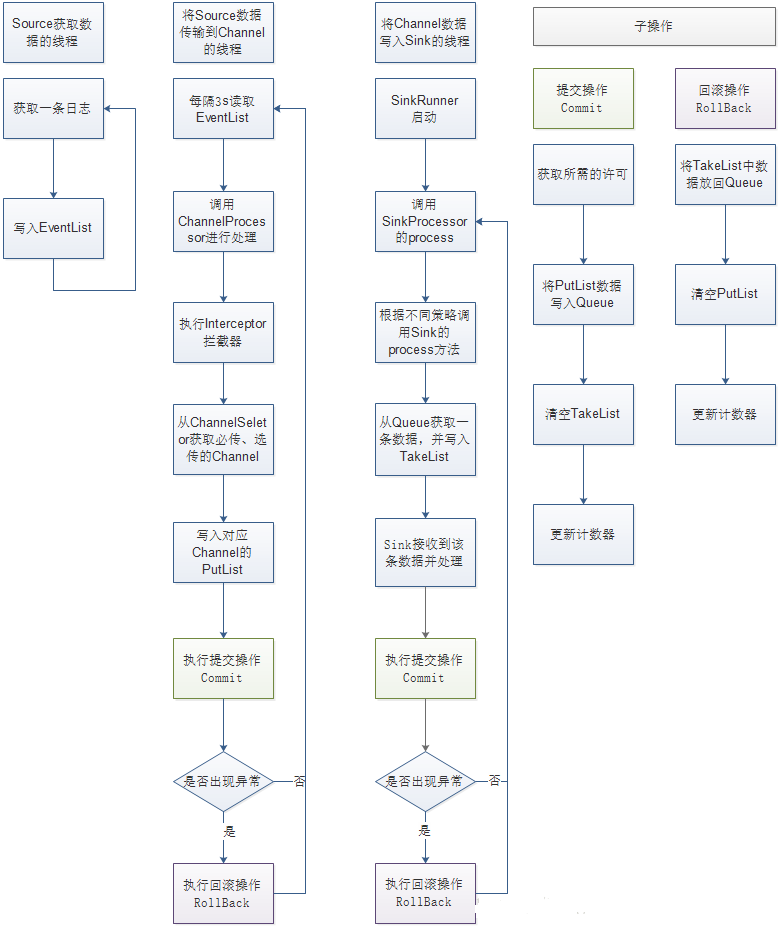
*application = new Application(components);*

*eventBus.register(application);*

*}*

## **1.3启动流程**

程序启动是会在startAllComponent中启动所有的SourceRunner、Channel、SinkRunner，其中Channel的启动就是初始化状态、创建计数器，核心是SourceRunner和SinkRunner，其执行流程图如下：



1. SourceRunner会调用Source的start方法，以ExecSource为例，其start方法就是启动一个线程，去不断获取标准输出流写入列表EventList，同时再启动一个线程去定期批量地把列表中的数据往Channel发
2. SinkRunner则是不断循环调用SinkProcess的process方法，SinkProcess用于决定选择哪个Sink进行存储，选择Sink后调用其process，其主要工作是去Channel中读取数据，并写入对应的存储。

# Flume基础库

Flume基础库是其他模块的运行基础

## **组件库-Lifecycle**

对于生命周期较长的对象，Flume采用了基于LifeCycle的对象模型对其进行管理，该模型主要有以下几个特点：

* 将每个组件的对象分为4个状态：IDLE(被创建，服务处于空闲状态), START

(启动), STOP(停止)及ERROR

* 组件的状态变化都可以触发一些其他动作
* 可通过组合的方式对任务组件进行组合，以便进行统一管理

在Flume中组件模型如下图所示：



在类图中所有组件对象最终均实现了接口LifecycleAware，其定义了最基本的组件启动、停止等操作，其定义如下：

*public interface LifecycleAware {*

*public void start();*

*public void stop();*

*public LifecycleState getLifecycleState();*

*}*

Flume中所有组件通过LifecycleSupervisor进行监管，其执行操作supervisor，执行如下：

*public synchronized void supervise(LifecycleAware lifecycleAware,*

*SupervisorPolicy policy, LifecycleState desiredState) {*

*Supervisoree process = new Supervisoree();*

*process.status = new Status();*

*process.policy = policy;*

*process.status.desiredState = desiredState;*

*process.status.error = false;*

*MonitorRunnable monitorRunnable = new MonitorRunnable();*

*monitorRunnable.lifecycleAware = lifecycleAware;*

*monitorRunnable.supervisoree = process;*

*monitorRunnable.monitorService = monitorService;*

*supervisedProcesses.put(lifecycleAware, process);*

*ScheduledFuture<?> future = monitorService.scheduleWithFixedDelay(*

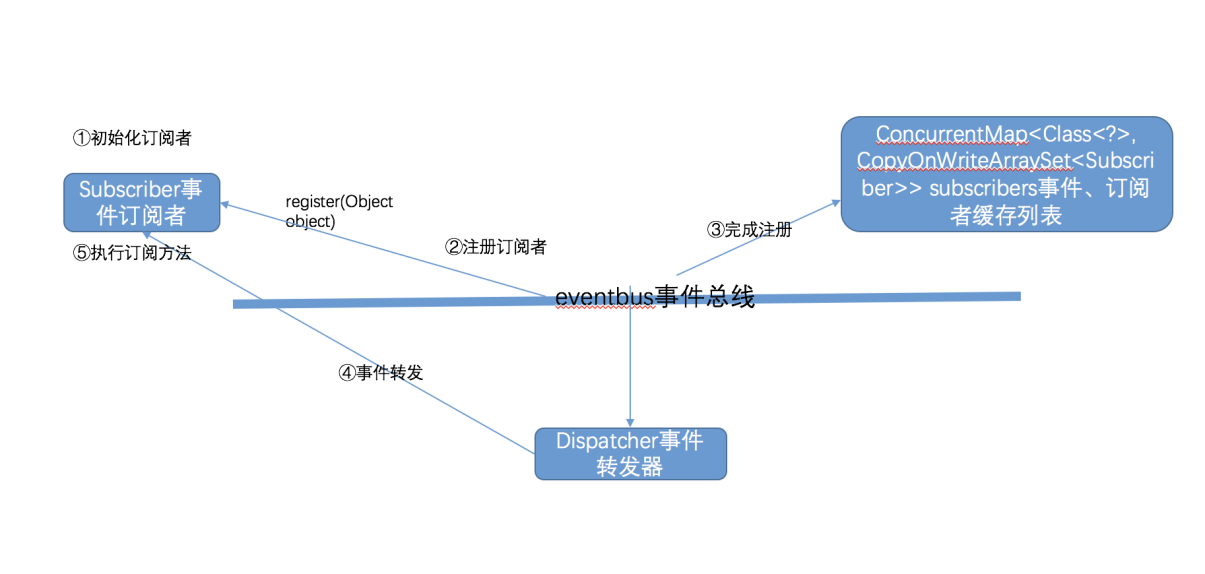
*monitorRunnable, 0, 3, TimeUnit.SECONDS);*

*monitorFutures.put(lifecycleAware, future);*

*}*

## **事件处理-EventBus**

Flume采用了基于事件驱动的模型，能够增强并发性，从而提高系统整体性能，为了构建该模型使用Guava EventBus来进行事件的订阅和处理，其基于观察者模式（生产者/消费者）的实现。



其处理过程大致为：处理请求将作为事件进入系统，由EventBus负责传递给相应的事件订阅器（Subscriber）。下面是示例：

**

1）定义EventBusCenter

*public class EventBusCenter {*

*private static EventBus eventBus = new EventBus();*

*private EventBusCenter(){}*

*public static EventBus getInstance() {*

*return eventBus;*

*}*

*public static void register(Object obj) {*

*eventBus.register(obj);*

*}*

*public static void unregister(Object obj) {*

*eventBus.unregister(obj);*

*}*

*public static void post(Object obj) {*

*eventBus.post(obj);*

*}*

*}*

1. 定义Observer

*public class DataObserver {*

*@Subscribe*

*public void func(Integer msg) {*

*System.out.println("Integer-MSG: " + msg);*

*}*

*}*

*public class StrObserver {*

*@Subscribe*

*public void func(String msg) {*

*System.out.println("Str-MSG: "+ msg);*

*}*

*}*

1. 测试程序

*DataObserver dataObserver = new DataObserver();*

*StrObserver strObserver = new StrObserver();*

*EventBusCenter.register(dataObserver);*

*EventBusCenter.register(strObserver);*

*System.out.println("========== Start Post MSG ===========");*

*EventBusCenter.post("Post String");*

*EventBusCenter.post(123);*

输出如下：

*========== Start Post MSG ===========*

*Str-MSG: Post String*

*Integer-MSG: 123*

## **EventBus的使用**

对于Flume Agent，其使用EventBus，将主线程注册到EventBus，定义subscribe事件，其定义如下：

*Application#*

*@Subscribe //其接收到MaterializedConfiguration后，进行处理*

*public void handleConfigurationEvent(MaterializedConfiguration conf) {*

*try {*

*lifecycleLock.lockInterruptibly();*

*stopAllComponents();*

*startAllComponents(conf);*

*} ....*

*}*

当配置文件改变后FileWatcherRunnable，通过EventBus触发上面的方法

*@Override*

*public void run() {*

*counterGroup.incrementAndGet("file.checks");*

*long lastModified = file.lastModified();*

*if (lastModified > lastChange) {*

*try {*

*eventBus.post(getConfiguration()); <= MaterializedConfiguration*

*} ......*

*}*

*}*

*}*

# Source

Source是agent来源组件，其将Event传递给Channel，接口如下所示：

*public interface Source extends LifecycleAware, NamedComponent {*

*public void setChannelProcessor(ChannelProcessor channelProcessor);*

*public ChannelProcessor getChannelProcessor();*

*}*

Source通过ChannelProcessor将event传给Channel。在Flume启动流程中可知，Application先启动SourceRunner，再由SourceRunner来启动Source，如下：

*public static SourceRunner forSource(Source source) {*

*SourceRunner runner = null;*

*if (source instanceof PollableSource) {*

*runner = new PollableSourceRunner();*

*((PollableSourceRunner) runner).setSource((PollableSource) source);*

*} else if (source instanceof EventDrivenSource) {*

*runner = new EventDrivenSourceRunner();*

*((EventDrivenSourceRunner) runner).setSource((EventDrivenSource) source);*

*} ....*

*return runner;*

*}*

**1）SourceRunner**

SourceRunner分为EventDrivenSourceRunner和PollableSourceRunner，其核心方法在start中，对比如下：

|  |  |
| --- | --- |
| *public class EventDrivenSourceRunner extends SourceRunner {*  *…*  *@Override*  *public void start() {*  *Source source = getSource();*  *ChannelProcessor cp = source.getChannelProcessor();*  *cp.initialize();*  *source.start();*  *lifecycleState = LifecycleState.START;*  *}*  *…*  *}* | **p***ublic class PollableSourceRunner extends SourceRunner {*    *…*  *@Override*  *public void start() {*  *PollableSource source = (PollableSource) getSource();*  *ChannelProcessor cp = source.getChannelProcessor();*  *cp.initialize();*  *source.start();*  *runner = new PollingRunner();*  *runner.source = source;*  *runner.counterGroup = counterGroup;*  *runner.shouldStop = shouldStop;*  *runnerThread = new Thread(runner);*  *runnerThread.setName(getClass().getSimpleName() + "-" +*  *source.getClass().getSimpleName() + "-" + source.getName());*  *runnerThread.start();*  *lifecycleState = LifecycleState.START;*  *}* |

无论是哪种SourceRunner，都会调用Source的start，其ChannelProcessor都是从LoadSource中进行初始化，核心执行在AbstractConfigurationProvider#loadSource

*ChannelSelector selector = ChannelSelectorFactory.create(sourceChannels, selectorConfig);*

*ChannelProcessor channelProcessor = new ChannelProcessor(selector);*

*Configurables.configure(channelProcessor, config);*

以上是Source的启动流程。

1. **Source Event**

以示例中的SpoolDirectorySource为例，其通过SpoolDirectoryRunner来调用目录查看逻辑，并根据新生成的文件生成Event，其执行如下：

*public void run() {*

*int backoffInterval = 250;*

*boolean readingEvents = false;*

*try {*

*while (!Thread.interrupted()) {*

*readingEvents = true;*

*List<Event> events = reader.readEvents(batchSize);*

*readingEvents = false;*

*if (events.isEmpty()) {*

*break;*

*}*

*try {*

*getChannelProcessor().processEventBatch(events);*

*reader.commit();*

*}*

*backoffInterval = 250;*

*sourceCounter.addToEventAcceptedCount(events.size());*

*sourceCounter.incrementAppendBatchAcceptedCount();*

*}*

*} .....*

*}*

其核心有两点：

* ReliableSpoolingFileEventReader

其根据新文件生成Event，如下：

*public List<Event> readEvents(int numEvents) throws IOException {*

*if (!committed) {*

*...*

*currentFile.get().getDeserializer().reset();*

*} ......*

*List<Event> events = readDeserializerEvents(numEvents);*

*.....*

*}*

*将文件名，写入到Header中*

*private void fillHeader(List<Event> events) {*

*if (annotateFileName) {*

*String filename = currentFile.get().getFile().getAbsolutePath();*

*for (Event event : events) {*

*event.getHeaders().put(fileNameHeader, filename);*

*}*

*}*

*if (annotateBaseName) {*

*String basename = currentFile.get().getFile().getName();*

*for (Event event : events) {*

*event.getHeaders().put(baseNameHeader, basename);*

*}*

*}*

*}*

* ChannelProcessor，将Event传给Channel

*public void processEventBatch(List<Event> events) {*

*//添加interceptor*

*events = interceptorChain.intercept(events);*

*for (Event event : events) {*

*List<Channel> reqChannels = selector.getRequiredChannels(event);*

*for (Channel ch : reqChannels) {*

*List<Event> eventQueue = reqChannelQueue.get(ch);*

*if (eventQueue == null) {*

*eventQueue = new ArrayList<Event>();*

*reqChannelQueue.put(ch, eventQueue);*

*}*

*eventQueue.add(event);*

*}*

*List<Channel> optChannels = selector.getOptionalChannels(event);*

*for (Channel ch : optChannels) {*

*List<Event> eventQueue = optChannelQueue.get(ch);*

*if (eventQueue == null) {*

*eventQueue = new ArrayList<Event>();*

*optChannelQueue.put(ch, eventQueue);*

*}*

*eventQueue.add(event);*

*}*

*}*

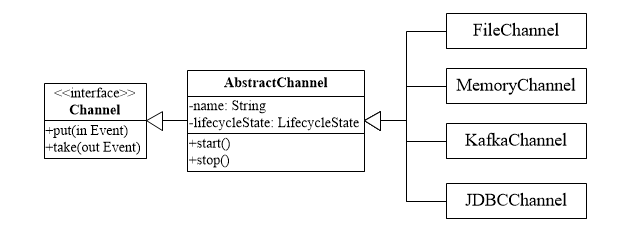
*}*

将event添加到EventQueue中。

Source源码分析：<https://www.2cto.com/kf/201703/610974.html>

# Channel

在Source中，将Event放入到EventQueue中，定义了多种Channel，其类图如下所示：



以MemoryChannel，其定义如下：

*public class MemoryChannel extends BasicChannelSemantics {*

*@GuardedBy(value = "queueLock")*

*private LinkedBlockingDeque<Event> queue;*

*protected void doPut(Event event) throws InterruptedException {*

*channelCounter.incrementEventPutAttemptCount();*

*int eventByteSize = (int) Math.ceil(estimateEventSize(event) / byteCapacitySlotSize);*

*if (!putList.offer(event)) {*

*....*

*}*

*putByteCounter += eventByteSize;*

*}*

*@Override*

*protected Event doTake() throws InterruptedException {*

*Event event;*

*synchronized (queueLock) {*

*event = queue.poll();*

*}*

*takeList.put(event);*

*return event;*

*}*

*}*

# Sink

将Event从Channel取出后写入目标存储中，Sink的接口如下：

*public interface Sink extends LifecycleAware, NamedComponent {*

*public void setChannel(Channel channel);*

*public Channel getChannel();*

*public Status process() throws EventDeliveryException;*

*public static enum Status {*

*READY, BACKOFF*

*}*

*}*

Sink中process方法，从channle中获取Event，并进行处理，下面是自定义Sink的实现

*public class MySink extends AbstractSink implements Configurable {*

*private static final Logger logger = LoggerFactory.getLogger(MySink.class);*

*private static final String PROP\_KEY\_ROOTPATH = "fileName";*

*private String fileName;*

*public void configure(Context context) {*

*fileName = context.getString(PROP\_KEY\_ROOTPATH);*

*}*

*public Status process() throws EventDeliveryException {*

*Channel ch = getChannel();*

*Transaction txn = ch.getTransaction();*

*txn.begin();*

*Event event = null;*

*while(true) {*

*event = ch.take();*

*if(event != null) {*

*break;*

*}*

*}*

*try {*

*String body = new String(event.getBody()); //将Event中的body输出*

*String res = body + ":" + System.currentTimeMillis() + "\r\n";*

*File file = new File(fileName);*

*FileOutputStream fos = new FileOutputStream(file,true);*

*fos.write(res.getBytes());*

*fos.close();*

*txn.commit();*

*return Status.READY;*

*} }*

# Channel Interceptor

Flume中的拦截器（Interceptor），当Source读取event发送到Sink的时候，在Events Header中加入有用的信息，或者对events中的内容进行过滤，完成初步的数据清洗。

在Source中指定拦截器的定义如下：

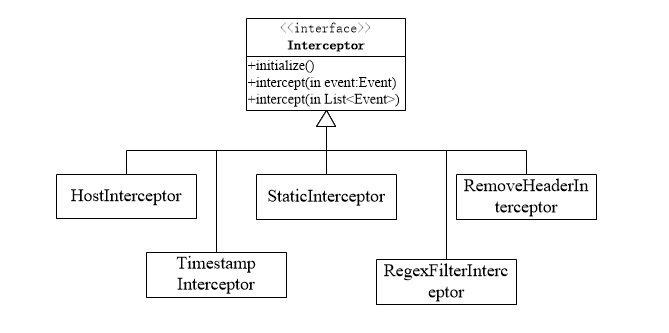
*a1.sources.r1.interceptors=i1 i2*

*a1.sources.r1.interceptors.i1.type=regex\_filter*

*a1.sources.r1.interceptors.i1.regex=\\{.\*\\}*

*a1.sources.r1.interceptors.i2.type=timestamp*

在Flume中目前提供以下拦截器



其使用在ChannelProcessor中，执行如下：

*public void processEventBatch(List<Event> events) {*

*//添加interceptor*

*events = interceptorChain.intercept(events);*

*.......*

*}*

*https://blog.csdn.net/buzaiqq/article/details/72864144*

*https://www.cnblogs.com/chushiyaoyue/p/6207638.html*

*https://blog.csdn.net/lnho2015/article/details/58055919*