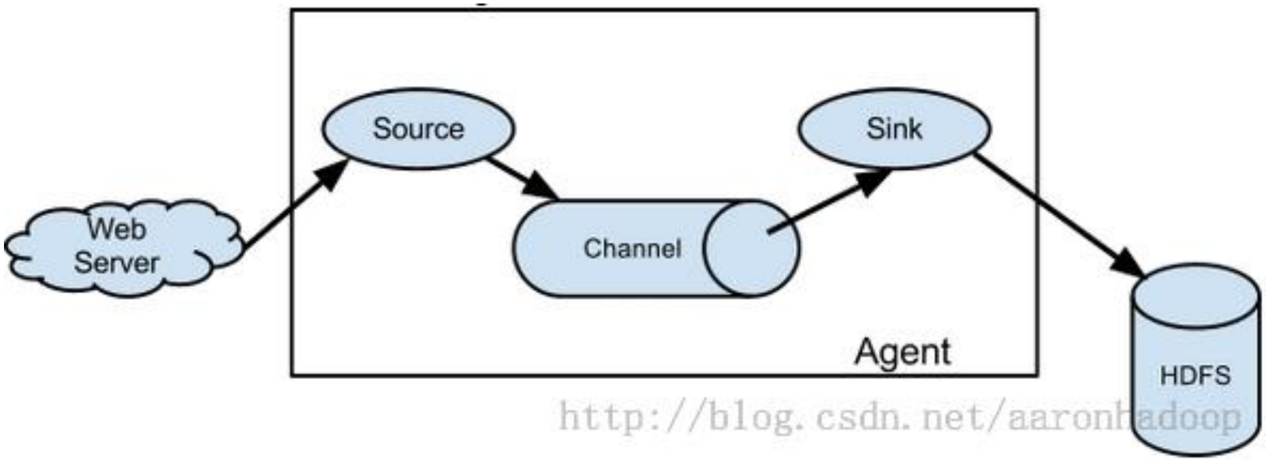
Flume+Kafka+日志分析demo

董炫辰
IBM Platform Computing
xcdong@cn.ibm.com

Agenda（Flume）

- 基本组件介绍
- Event
- Transaction
- interceptor
- selector（多路复用，多路分发）

基本组件介绍



- Flume是一个海量日志采集、聚合和传输的系统。轻量，配置简单，使用灵活。Flume支持在日志系统中定制各类数据发送方， 同时，Flume提供对数据进行简单处理，并写到各种数据接受方的能力。

- Flume的核心是**agent**。Agent是一个java进程，运行在日志收集端。

- Agent里面包含3个核心组件：**source**、**channel**、**sink**。

  - Source组件是用于收集日志的，可以处理各种类型各种格式的日志数据,包括avro、thrift、**exec**、jms、**spooling directory**、netcat、sequence generator、syslog、http、legacy、自定义。source组件把数据收集来以后，临时存放在channel中。

  - Channel组件是在agent中专用于临时存储数据的，可以把Channel看作是一个缓冲区。可以存放在**memory**、jdbc、**file**、自定义。channel中的数据只有在sink发送成功之后才会被删除。

  - Sink组件是用于把数据发送到目的地的组件，目的地包括hdfs、logger、avro、thrift、ipc、file、null、hbase、solr、自定义。

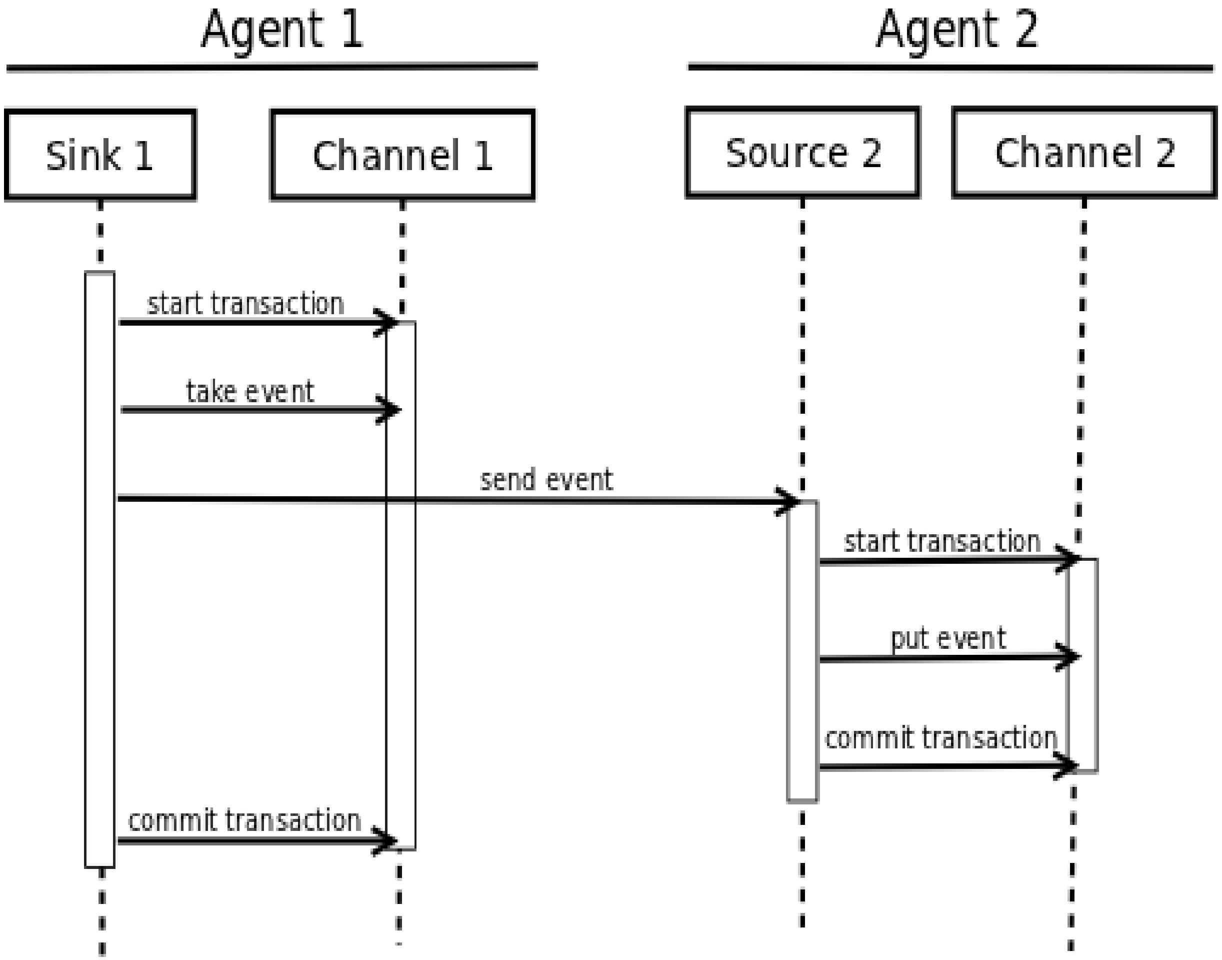
Event

- Flume的数据流由事件(Event)贯穿始终。
- Event是Flume的基本数据单位
- 它包括Header集合和Body两部分组成

```
public interface Event {
    public Map<String, String> getHeaders();
    public void setHeaders(Map<String, String> headers);
    public byte[] getBody();
    public void setBody(byte[] body);
}
```

Transaction

事务保证是在event级别

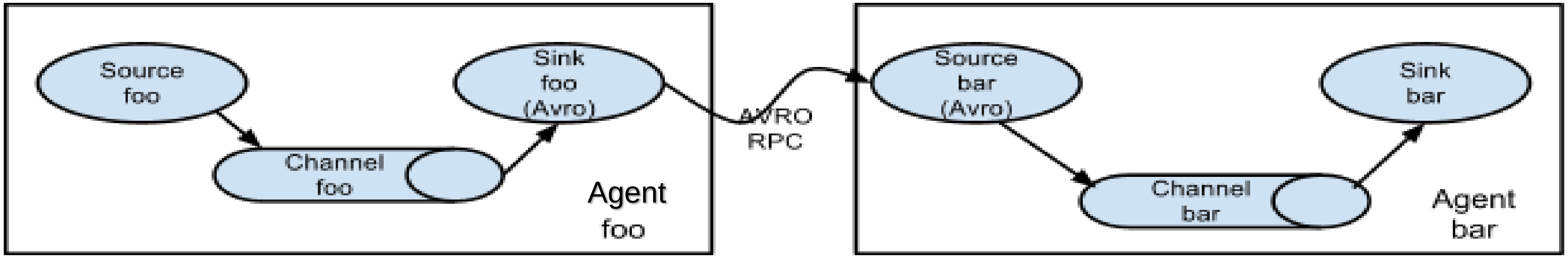


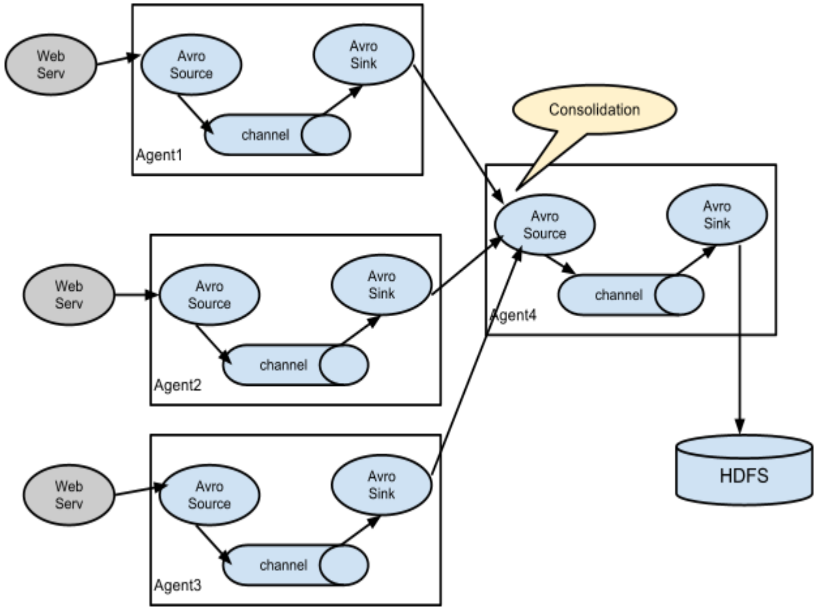
Transaction

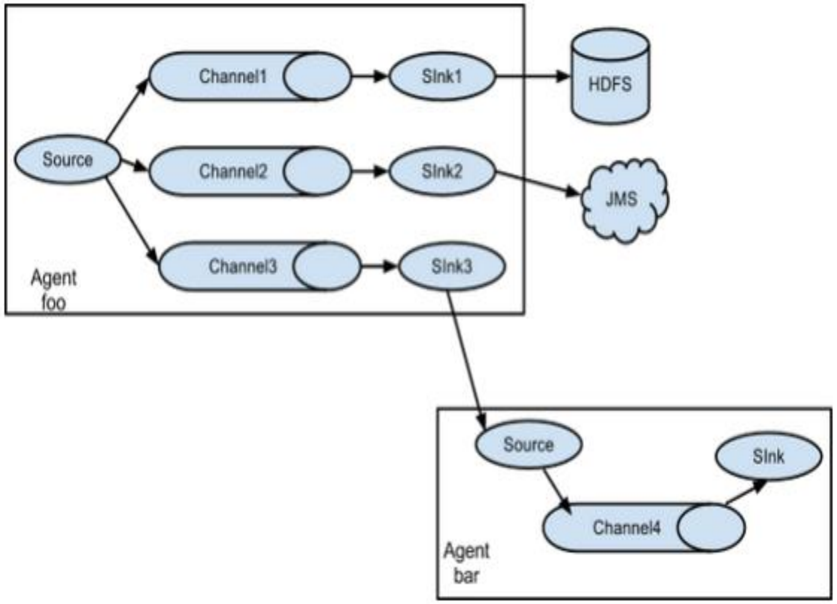
```
 Channel ch = new MemoryChannel();
Transaction txn = ch.getTransaction();
txn.begin();
try {
  Event eventToStage = EventBuilder.withBody("Hello Flume!",
                       Charset.forName("UTF-8"));
  ch.put(eventToStage);
  // Event takenEvent = ch.take();
  // ...
  txn.commit();
} catch (Throwable t) {
  txn.rollback();
  if (t instanceof Error) {
    throw (Error)t;
  }
} finally {
  txn.close();
}
```

multi-agent flow

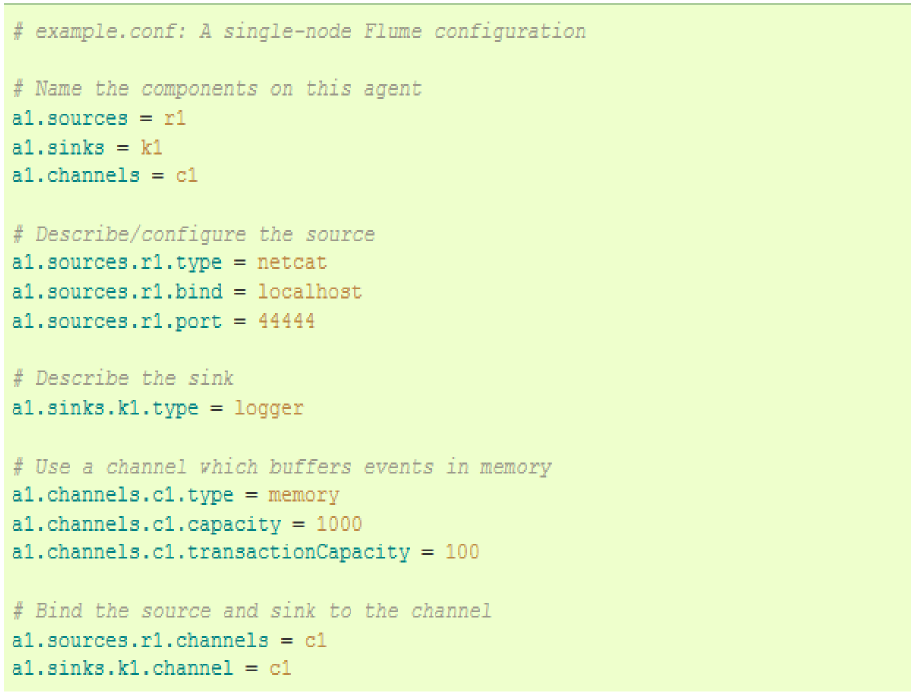
Flume可以支持多级flume的agent，支持扇入(fan-in)、扇出(fan-out)。



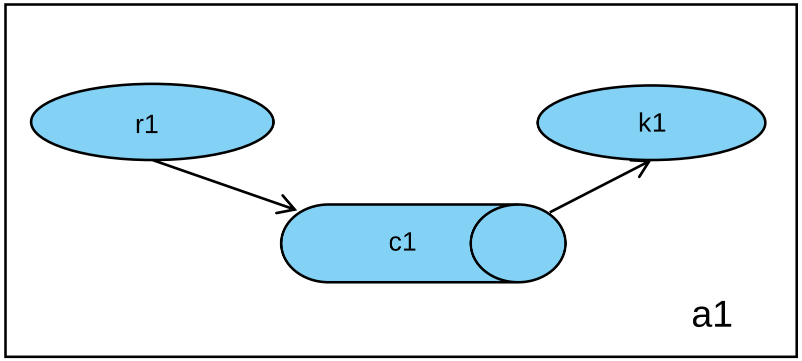




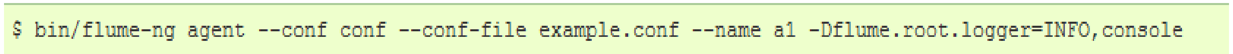
例子



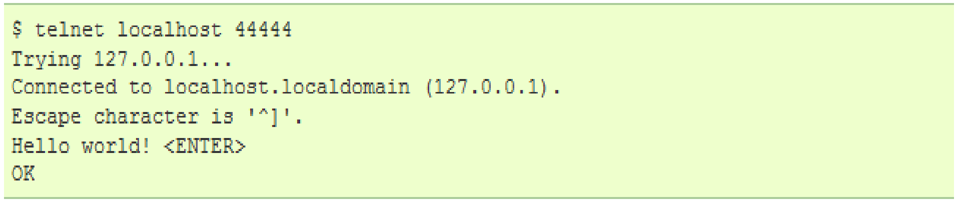
```
# example.conf: A single-node Flume configuration

# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = netcat
a1.sources.r1.bind = localhost
a1.sources.r1.port = 44444

# Describe the sink
a1.sinks.k1.type = logger

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

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



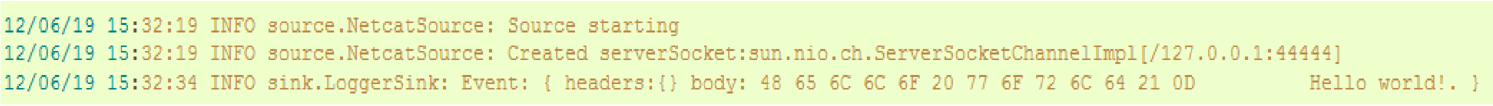
1



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



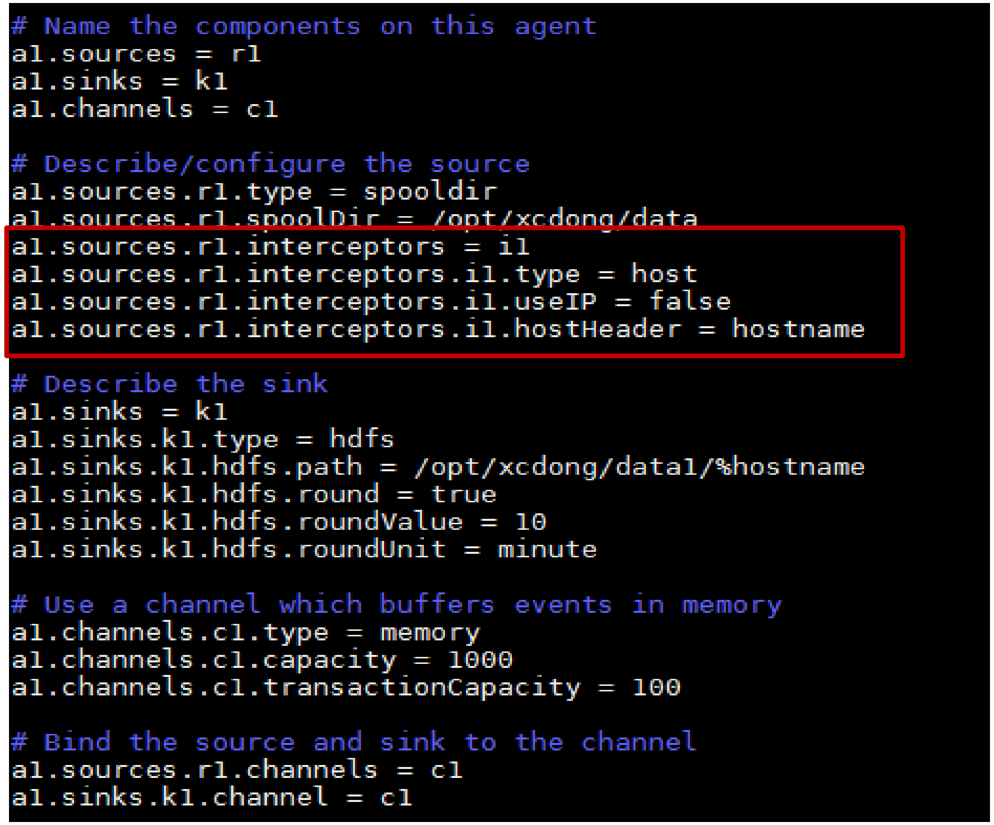
```
$ telnet localhost 44444
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
Hello world! <ENTER>
OK
```



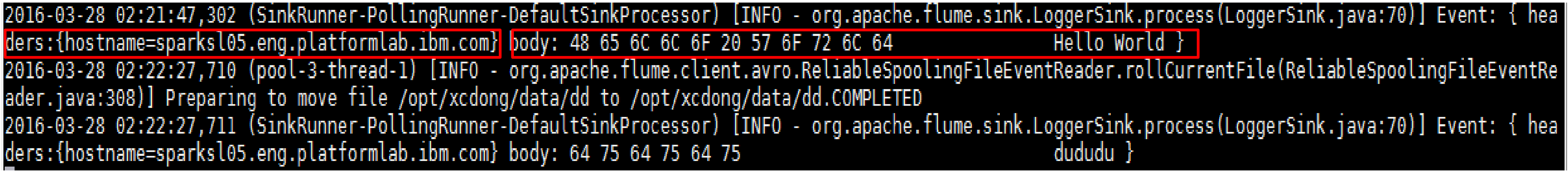
```
12/06/19 15:32:19 INFO source.NetcatSource: Source starting
12/06/19 15:32:19 INFO source.NetcatSource: Created serverSocket:sun.nio.ch.ServerSocketChannelImpl[/127.0.0.1:44444]
12/06/19 15:32:34 INFO sink.LoggerSink: Event: { headers:{} body: 48 65 6C 6C 6F 20 77 6F 72 6C 64 21 0D          Hello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

interceptor

host拦截器



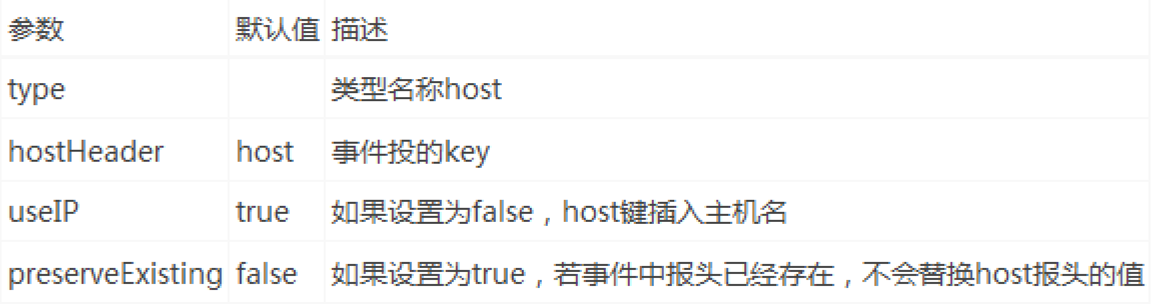
```
# Name the components on this agent
a1.sources = r1
a1.sinks = k1
a1.channels = c1

# Describe/configure the source
a1.sources.r1.type = spooldir
a1.sources.r1.spoolDir = /opt/xcdong/data
a1.sources.r1.interceptors = i1
a1.sources.r1.interceptors.i1.type = host
a1.sources.r1.interceptors.i1.useIP = false
a1.sources.r1.interceptors.i1.hostHeader = hostname

# Describe the sink
a1.sinks = k1
a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.path = /opt/xcdong/data1/%hostname
a1.sinks.k1.hdfs.round = true
a1.sinks.k1.hdfs.roundValue = 10
a1.sinks.k1.hdfs.roundUnit = minute

# Use a channel which buffers events in memory
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100

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



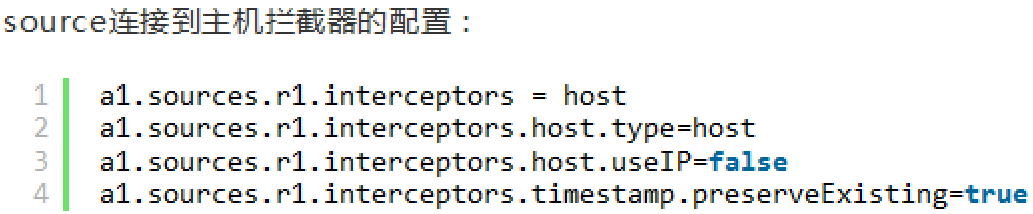
```
2016-03-28 02:21:47,302 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:70)] Event: { hea
ders:{hostname=sparksl05.eng.platformlab.ibm.com} body: 48 65 6C 6C 6F 20 57 6F 72 6C 64                 Hello World }
2016-03-28 02:22:27,710 (pool-3-thread-1) [INFO - org.apache.flume.client.avro.ReliableSpoolingFileEventReader.rollCurrentFile(ReliableSpoolingFileEventRe
ader.java:308)] Preparing to move file /opt/xcdong/data/dd to /opt/xcdong/data/dd.COMPLETED
2016-03-28 02:22:27,711 (SinkRunner-PollingRunner-DefaultSinkProcessor) [INFO - org.apache.flume.sink.LoggerSink.process(LoggerSink.java:70)] Event: { hea
ders:{hostname=sparksl05.eng.platformlab.ibm.com} body: 64 75 64 75 64 75                             dududu }
```

interceptor

主机拦截器：



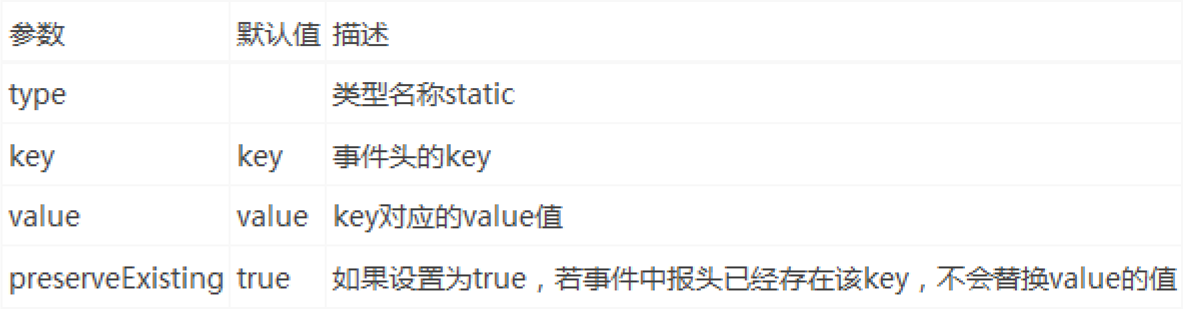
参数	默认值	描述
type		类型名称host
hostHeader	host	事件投的key
useIP	true	如果设置为false，host键插入主机名
preserveExisting	false	如果设置为true，若事件中报头已经存在，不会替换host报头的值



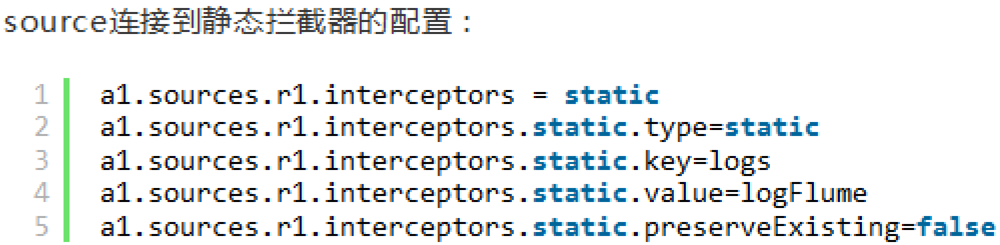
source连接到主机拦截器的配置：

```
a1.sources.r1.interceptors = host
a1.sources.r1.interceptors.host.type=host
a1.sources.r1.interceptors.host.useIP=false
a1.sources.r1.interceptors.timestamp.preserveExisting=true
```

静态拦截器：



参数	默认值	描述
type		类型名称static
key	key	事件头的key
value	value	key对应的value值
preserveExisting	true	如果设置为true，若事件中报头已经存在该key，不会替换value的值

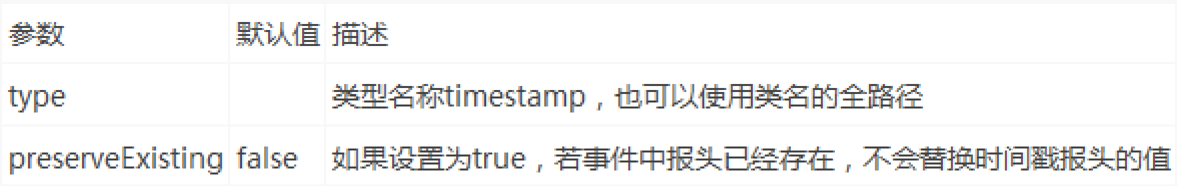


source连接到静态拦截器的配置：

```
a1.sources.r1.interceptors = static
a1.sources.r1.interceptors.static.type=static
a1.sources.r1.interceptors.static.key=logs
a1.sources.r1.interceptors.static.value=logFlume
a1.sources.r1.interceptors.static.preserveExisting=false
```

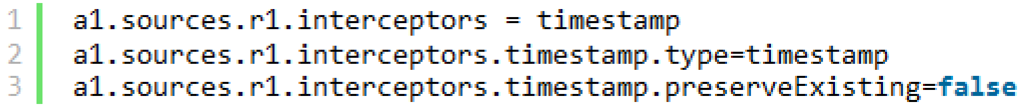
interceptor

时间戳拦截器：



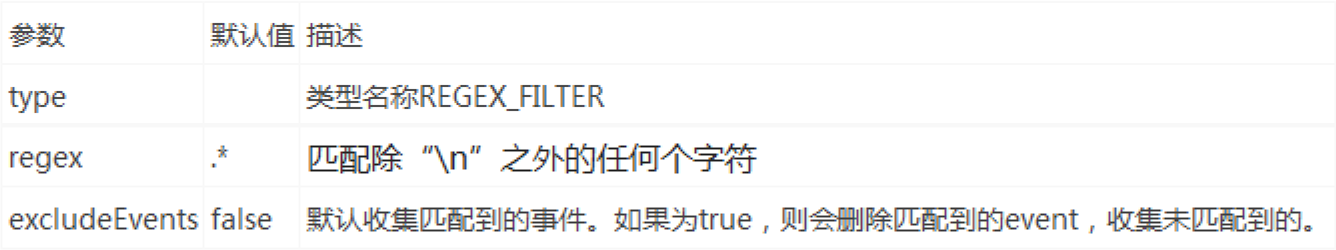
参数	默认值	描述
type		类型名称timestamp，也可以使用类名的全路径
preserveExisting	false	如果设置为true，若事件中报头已经存在，不会替换时间戳报头的值

source连接到时间戳拦截器的配置：



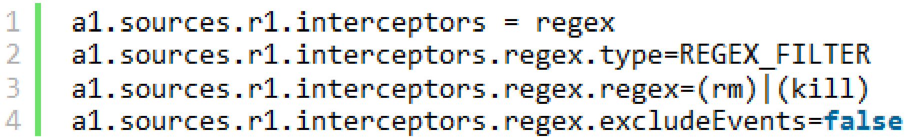
```
a1.sources.r1.interceptors = timestamp
a1.sources.r1.interceptors.timestamp.type=timestamp
a1.sources.r1.interceptors.timestamp.preserveExisting=false
```

正则过滤拦截器：



参数	默认值	描述
type		类型名称REGEX_FILTER
regex	.*	匹配除 “\n” 之外的任何个字符
excludeEvents	false	默认收集匹配到的事件。如果为true，则会删除匹配到的event，收集未匹配到的。

source连接到正则过滤拦截器的配置：



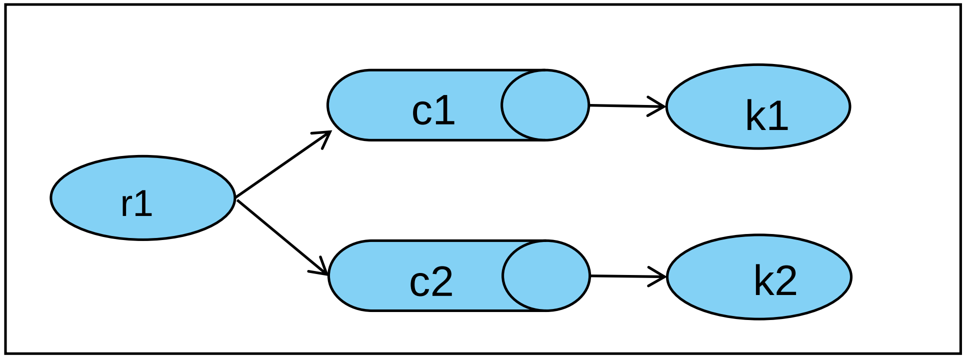
```
a1.sources.r1.interceptors = regex
a1.sources.r1.interceptors.regex.type=REGEX_FILTER
a1.sources.r1.interceptors.regex.regex=(rm)|(kill)
a1.sources.r1.interceptors.regex.excludeEvents=false
```

interceptor

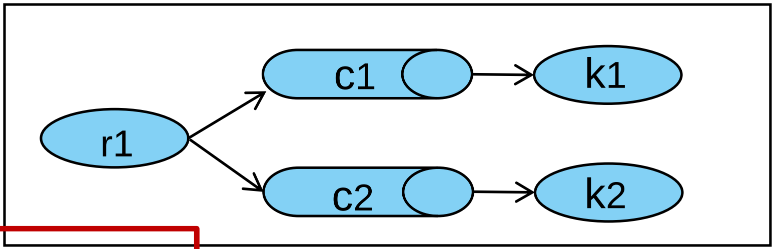
```
tier1.sources=source1
tier1.channels=channel1
tier1.sinks=sink1

tier1.sources.source1.type=exec
tier1.sources.source1.command = tail -f /opt/xcdong/spark-1.4.1-bin-hadoop2.6/app-20160318020019-0000
tier1.sources.source1.channels=channel1

tier1.sources.source1.interceptors=i1 i2
tier1.sources.source1.interceptors.i1.type=regex_filter
tier1.sources.source1.interceptors.i1.regex=\\{.*\\}
tier1.sources.source1.interceptors.i2.type=timestamp

tier1.channels.channel1.type=memory
tier1.channels.channel1.capacity=10000
tier1.channels.channel1.transactionCapacity=1000
tier1.channels.channel1.keep-alive=30

tier1.sinks.sink1.type=hdfs
tier1.sinks.sink1.channel=channel1
tier1.sinks.sink1.hdfs.path=hdfs://master68:8020/flume/events/%y-%m-%d
tier1.sinks.sink1.hdfs.fileType=DataStream
tier1.sinks.sink1.hdfs.writeFormat=Text
tier1.sinks.sink1.hdfs.rollInterval=0
tier1.sinks.sink1.hdfs.rollSize=10240
tier1.sinks.sink1.hdfs.rollCount=0
tier1.sinks.sink1.hdfs.idleTimeout=60
```

Replicating Channel Selector

```
a3.sources = r1
a3.sinks = k1 k2
a3.channels = c1 c2

a3.sources.r1.type=spooldir
a3.sources.r1.spoolDir = /flume/apache-flume-1.6.0-bin/replicat
a3.sources.r1.fileHeader = true
a3.sources.r1.channels = c1 c2
a3.sources.r1.selector.optional = c2
a3.sources.r1.selector.type=replicating

a3.sinks.k1.type = avro
a3.sinks.k1.channel = c1
a3.sinks.k1.hostname = ip
a3.sinks.k1.port = 4444

a3.sinks.k2.type = avro
a3.sinks.k2.channel = c2
a3.sinks.k2.hostname = ip
a3.sinks.k2.port = 5555

a3.channels.c1.type = memory
a3.channels.c1.capacity=1000
a3.channels.c1.transcationCapacity=100

a3.channels.c2.type = memory
a3.channels.c2.capacity=1000
a3.channels.c2.transcationCapacity=100
```



Multiplexing Channel Selector



```
agent1.sources = seqGenSrc
agent1.channels = memoryChannel1 memoryChannel2
agent1.sinks = msgRollingSink1 msgRollingSink2

# For each one of the sources, the type is defined
agent1.sources.seqGenSrc.type = com.flume.source.NetcatSource
agent1.sources.seqGenSrc.bind = 192.168.19.107
agent1.sources.seqGenSrc.port = 44444
agent1.sources.seqGenSrc.header = LOG_TYPE
agent1.sources.seqGenSrc.selector.type = multiplexing
agent1.sources.seqGenSrc.selector.header = LOG_TYPE
agent1.sources.seqGenSrc.selector.mapping.CREDIT = memoryChannel1
agent1.sources.seqGenSrc.selector.mapping.OTHER = memoryChannel2
agent1.sources.seqGenSrc.selector.default = memoryChannel2
agent1.sources.seqGenSrc.interceptors=i1 i2
agent1.sources.seqGenSrc.interceptors.i1.type=regex_filter
agent1.sources.seqGenSrc.interceptors.i1.regex=\\{.*\\}
agent1.sources.seqGenSrc.interceptors.i2.type=timestamp

# The channel can be defined as follows.
agent1.sources.seqGenSrc.channels = memoryChannel1 memoryChannel2

#Specify the channel the sink should use
agent1.sinks.msgRollingSink1.channel = memoryChannel1
agent1.sinks.msgRollingSink2.channel = memoryChannel2

# Each sink's type must be defined
agent1.sinks.msgRollingSink1.type=hdfs
agent1.sinks.msgRollingSink1.hdfs.path=hdfs://master68:8020/flume/sink1/%y-%m-%d
agent1.sinks.msgRollingSink2.type=hdfs
agent1.sinks.msgRollingSink2.hdfs.path=hdfs://master68:8020/flume/sink2/%y-%m-%d
```

selector

Multiplexing Channel Selector

netcat source开启监听端口，接收发送来的报文消息，
通过memory channel与sink（重写的roll file sink）写到本地磁盘。

将flume提供的NetcatSource中原来生成event的地方修改为：

```
bytes.get(body);
String line = new String(body);
String[] records = line.split("\t", 2);
String header = records[0];
String strBody = records[1];
Map<String, String> headers = new HashMap<String, String>();
headers.put("LOG_FILE", header);
Event event = EventBuilder.withBody(body, headers)
```

Agenda（Kafka）

- 介绍
- 拓扑结构
- topic && partition
- push && pull
- zookeeper
- 配置
- producer && consumer

介绍

- Apache Kafka是分布式发布-订阅消息系统。它最初由LinkedIn公司开发，之后成为Apache项目的一部分。

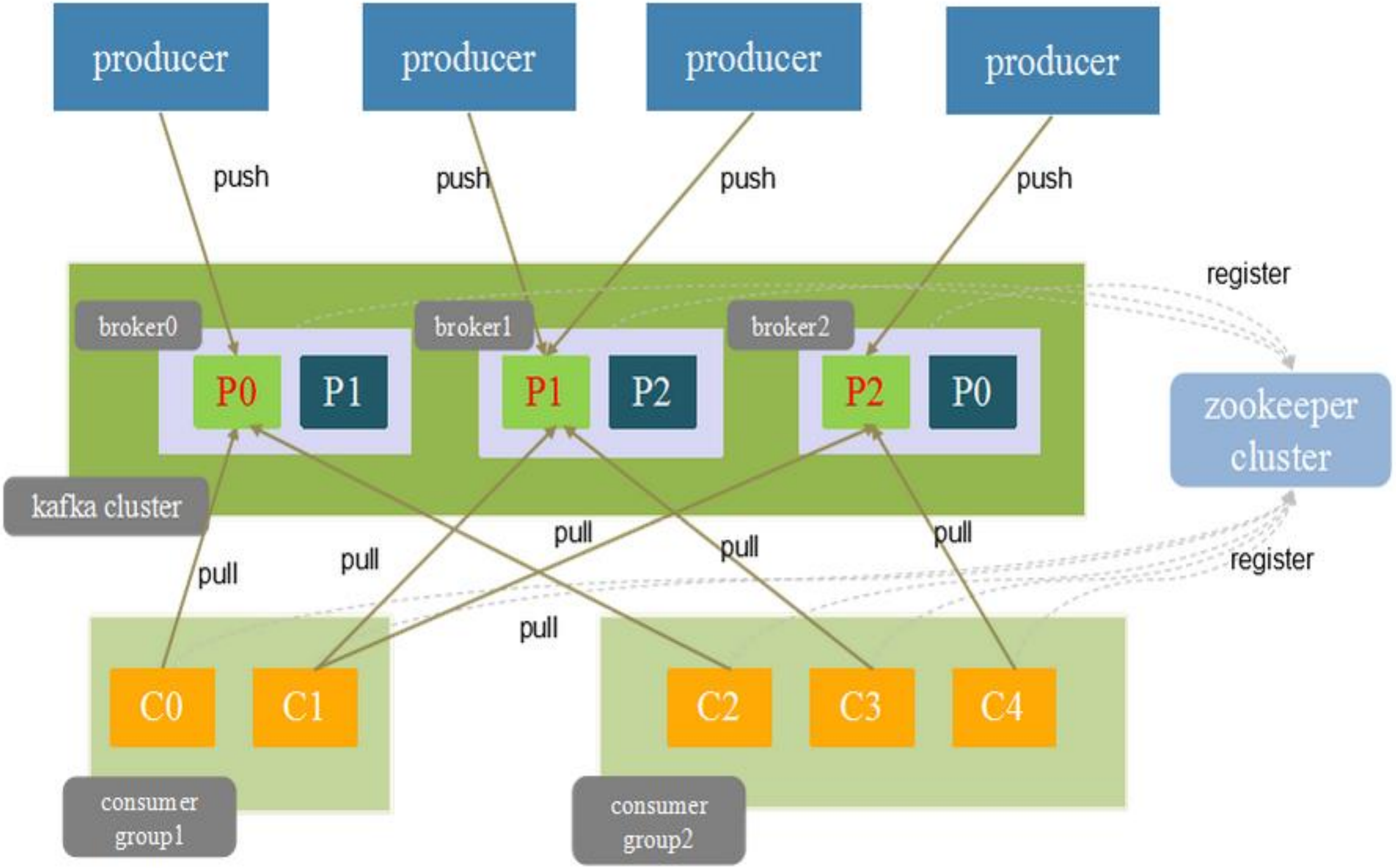
- Kafka是一种快速、可扩展的、设计内在就是分布式的，分区的和可复制的消息系统。

- Apache Kafka与传统消息系统相比，有以下不同：
  - 它被设计为一个分布式系统，易于向外扩展；
  - 它同时为发布和订阅提供高吞吐量；
  - 它支持多订阅者，当失败时能自动平衡消费者；
  - 它将消息持久化到磁盘，因此可用于批量消费，例如ETL，以及实时应用程序。

应用场景

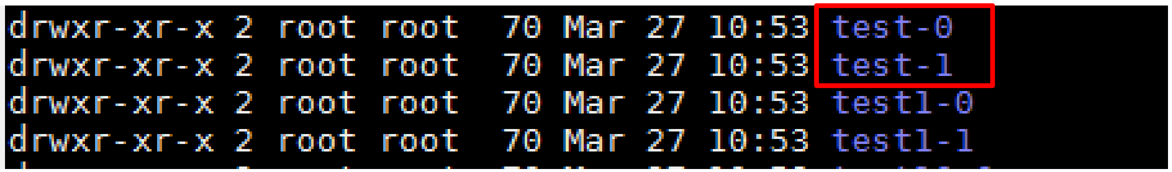
- 日志收集：一个公司可以用Kafka可以收集各种服务的log，通过kafka以统一接口服务的方式开放给各种consumer，例如hadoop、Hbase、Solr等。
- 消息系统：解耦和生产者和消费者、缓存消息等。
- 用户活动跟踪：Kafka经常被用来记录web用户或者app用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到kafka的topic中，然后订阅者通过订阅这些topic来做实时的监控分析，或者装载到hadoop、数据仓库中做离线分析和挖掘。
- 运营指标：Kafka也经常用来记录运营监控数据。包括收集各种分布式应用的数据，生产各种操作的集中反馈，比如报警和报告。
- 流式处理：比如spark streaming和storm

Kafka拓扑结构

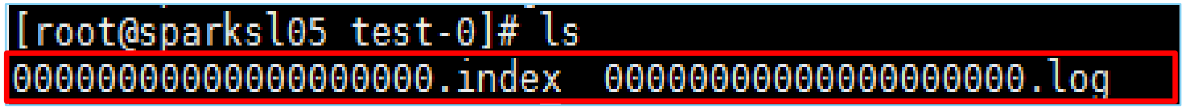

producer
producer
producer
producer
push
push
push
push
register
broker0
broker1
broker2
P0
P1
P1
P2
P2
P0
zookeeper cluster
kafka cluster
pull
pull
pull
pull
pull
register
pull
C0
C1
C2
C3
C4
consumer group1
consumer group2

topic && partition

bin/kafka-topics.sh --zookeeper xxx --create --topic test



```
drwxr-xr-x 2 root root  70 Mar 27 10:53 test-0
drwxr-xr-x 2 root root  70 Mar 27 10:53 test-1
drwxr-xr-x 2 root root  70 Mar 27 10:53 test1-0
drwxr-xr-x 2 root root  70 Mar 27 10:53 test1-1
```



```
[root@sparksl05 test-0]# ls
00000000000000000000.index  00000000000000000000.log
```

num.partitions=3（$KAFKA_HOME/config/server.properties中指定），
表示一个topic由几个partition组成。

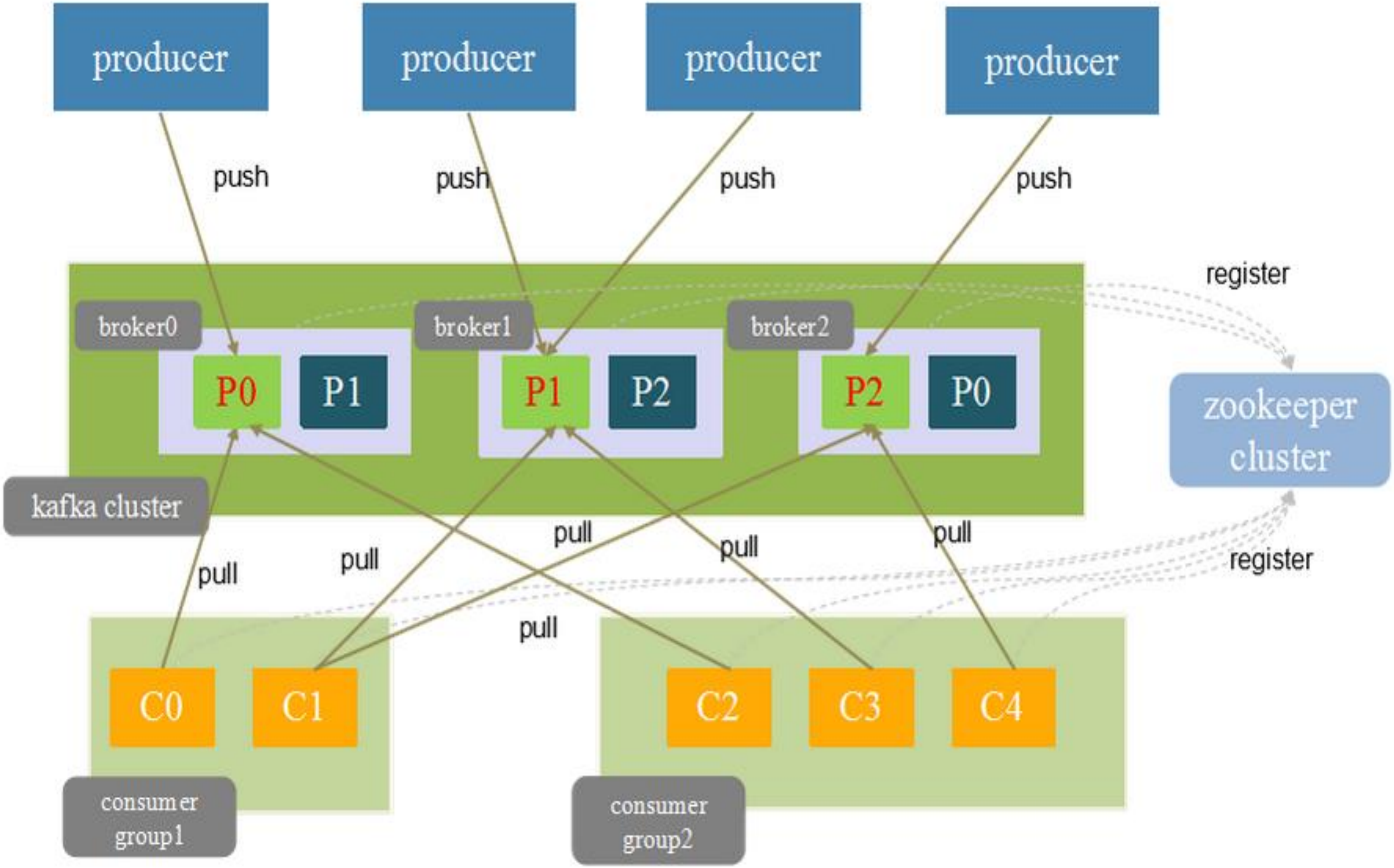
default.replication.factor = 1（$KAFKA_HOME/config/server.properties中指定）
表示一个partition有几个副本partition

bin/kafka-topics.sh --zookeeper 192.168.2.225:2183/config/mobile/mq/mafka02
--create --topic my-topic --partitions 1 --replication-factor 1
--config max.message.bytes=64000（创建）

--alter --topic my-topic --config max.message.bytes=128000（修改）

--alter --topic my-topic --delete-config max.message.bytes（删除）

push && pull

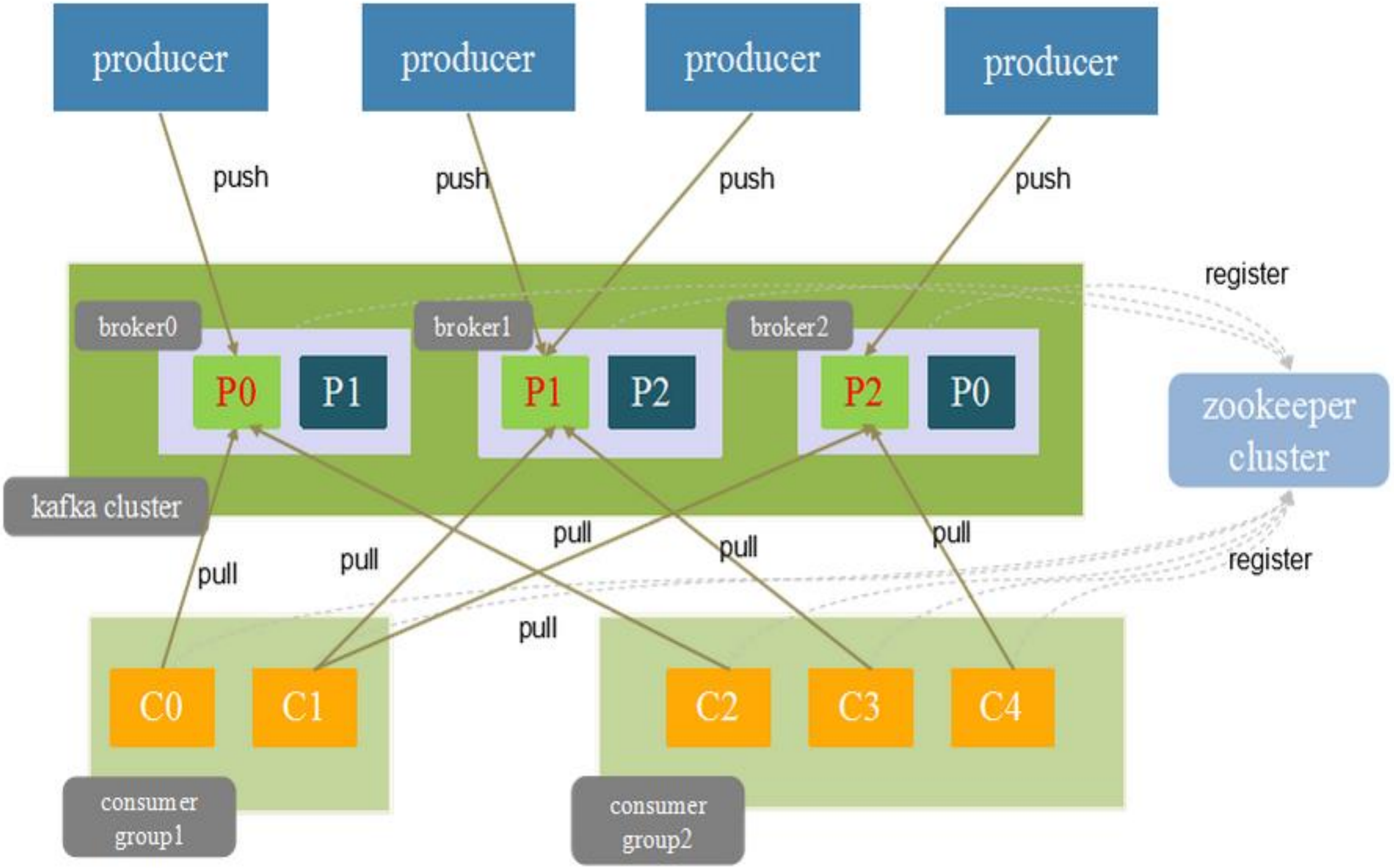

producer
producer
producer
producer
push
push
push
push
register
broker0
broker1
broker2
P0
P1
P1
P2
P2
P0
zookeeper cluster
kafka cluster
pull
pull
pull
pull
pull
pull
register
C0
C1
C2
C3
C4
consumer group1
consumer group2

Partitioner

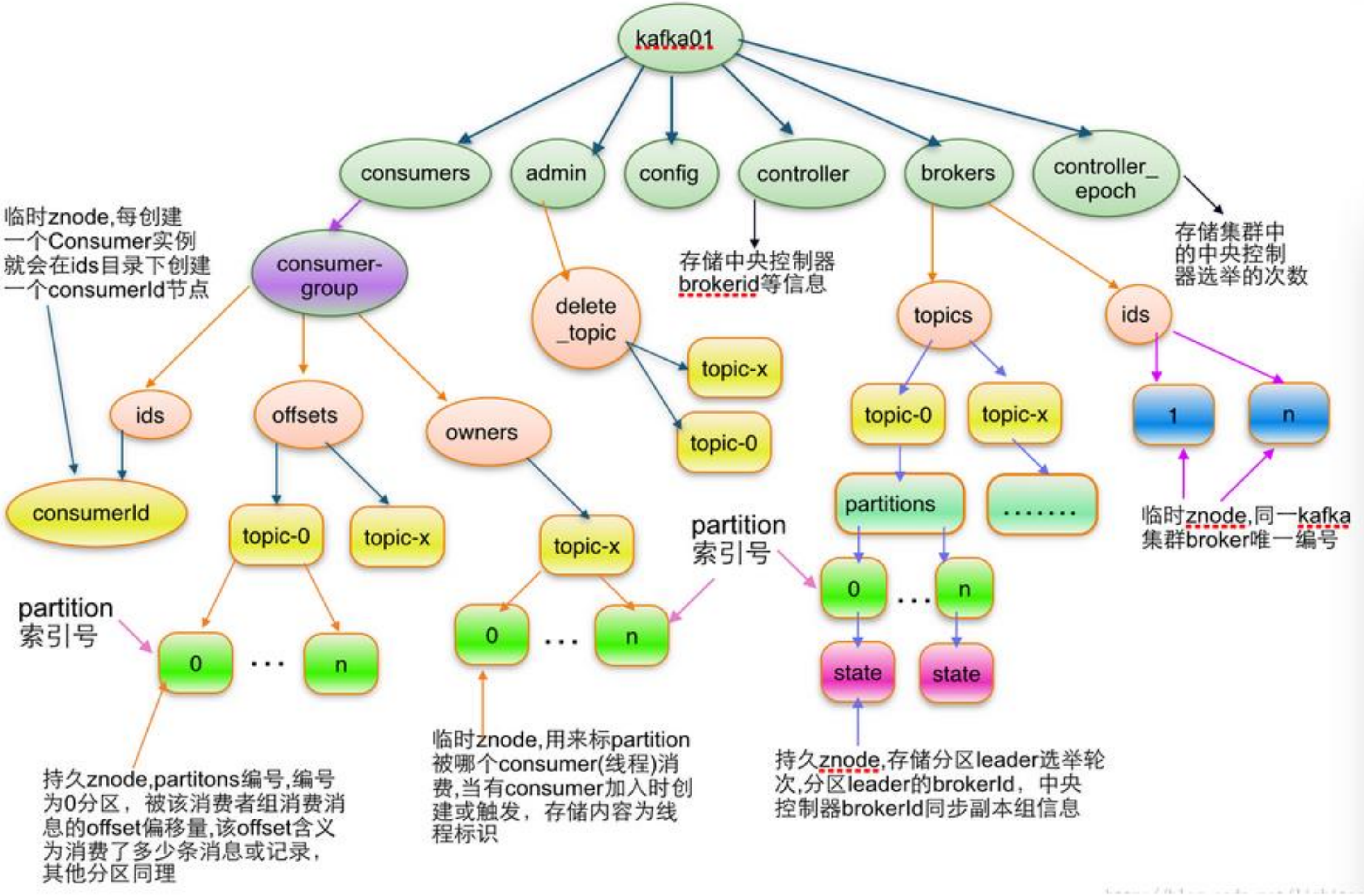
message分发到哪一个partition上，内部默认的是用hash值取余

```
private[kafka] class DefaultPartitioner[T] extends Partitioner[T] {
  private val random = new java.util.Random

  def partition(key: T, numPartitions: Int): Int = {
    if(key == null)
    {
        println("key is null")
        random.nextInt(numPartitions)
    }
    else
    {
        println("key is "+ key + " hashcode is "+key.hashCode)
        math.abs(key.hashCode) % numPartitions
    }
  }
}
```

zookeeper扮演的角色



zookeeper扮演的角色



临时znode,每创建一个Consumer实例就会在ids目录下创建一个consumerId节点

存储中央控制器brokerid等信息

存储集群中的中央控制器选举的次数

partition 索引号

partition 索引号

持久znode,partitons编号,编号为0分区，被该消费者组消费消息的offset偏移量,该offset含义为消费了多少条消息或记录，其他分区同理

临时znode,用来标partition被哪个consumer(线程)消费,当有consumer加入时创建或触发，存储内容为线程标识

持久znode,存储分区leader选举轮次,分区leader的brokerId，中央控制器brokerId同步副本组信息

临时znode,同一kafka集群broker唯一编号

其他

- 压缩

Producer端进行压缩之后，在Consumer端需进行解压。通过消息头部的压缩属性字节标志，这个字节的后两位表示消息是否被压缩，如果后两位为0，则表示消息未被压缩。

- 可靠性的保障
  - 从Producer端看：当一个消息被发送后，Producer会等待broker成功接收到消息的反馈（可通过参数控制等待时间），如果消息在途中丢失或是其中一个broker挂掉，Producer会重新发送
  - 从Consumer端看：zookeeper会记录partition中的一个offset值，这个值表示consumer消费到哪里了。若Consumer收到了消息，但却在处理过程中挂掉，此时Consumer可以通过这个offset值重新找到上一个消息再进行处理。Consumer还有权限控制这个offset值，对持久化到broker端的消息做任意处理。
  - 从broker的角度：一个备份数量为n的集群允许n-1个节点失败。若某个节点down掉，可以很快的切换到其副本所在的节点上，进行message的读写。
- 消息顺序性：

前面讲到过Partition，消息在一个Partition中的顺序是有序的，但是Kafka只保证消息在一个Partition中有序，如果要想使整个topic中的消息有序，那么一个topic仅设置一个Partition即可。

其他配置

Broker配置参数

- log.retention.hours
- log.dir
- log.segment.bytes

Consumer配置参数

- fetch.size
- auto.commit.interval.ms
- rebalance.retries.max

Producer配置参数

- producer.type
- queue.buffering.max.message
- batch.num.messages
- compression.codec
- message.send.max.retries

Producer

Producers可以是各种应用，比如web应用，服务器端应用，代理应用以及log系统等等。当然，Producers现在有各种语言的实现比如Java、C、Python等。

Kafka中和producer相关的API有三个类：

- Producer：最主要的类，用来创建和推送消息
- KeyedMessage：定义要发送的消息对象，比如定义发送给哪个topic，partition key和发送的内容等。
- ProducerConfig: 配置Producer，比如定义要连接的brokers、partition class、serializer class、partition key等

Producer

```
  import java.util.Properties;

  import kafka.javaapi.producer.Producer;
  import kafka.producer.KeyedMessage;
  import kafka.producer.ProducerConfig;

  public class SimpleProducer {
      private static Producer<Integer,String> producer;
      private final Properties props=new Properties();
      public SimpleProducer(){
          //定义连接的broker list
          props.put("metadata.broker.list", "192.168.4.31:9092");
          //定义序列化类（Java对象传输前要序列化）
          props.put("serializer.class", "kafka.serializer.StringEncoder");

          producer = new Producer<Integer, String>(new ProducerConfig(props));
          }
   public static void main(String[] args) {
      SimpleProducer sp=new SimpleProducer();
      //定义topic
      String topic="mytopic";
      //定义要发送给topic的消息
      String messageStr = "send a message to broker ";
      //构建消息对象
      KeyedMessage<Integer, String> data = new KeyedMessage<Integer, String>
                                                  (topic, messageStr);
      //推送消息到broker
      producer.send(data);
      producer.close();
   }
}
```

利用producerconfig生成producer，使用keyedmessage产生message，然后producer发送message

Consumer

Consumer是用来消费Producer产生的消息的，当然一个Consumer可以是各种应用，如可以是一个实时的分析系统，也可以是一个数据仓库或者是一个基于发布订阅模式的解决方案等。

与Producer类似，和Consumer相关主要的类：

- ConsumerConfig：定义要连接zookeeper的一些配置信息（Kafka通过zookeeper均衡压力，具体请查阅见面几篇文章），比如定义zookeeper的URL、group id、连接zookeeper过期时间等。
- ConsumerConnector:负责和zookeeper进行连接等工作

另外，可以通过 consumer.createMessageStreams，获取producer push在broker上的数据

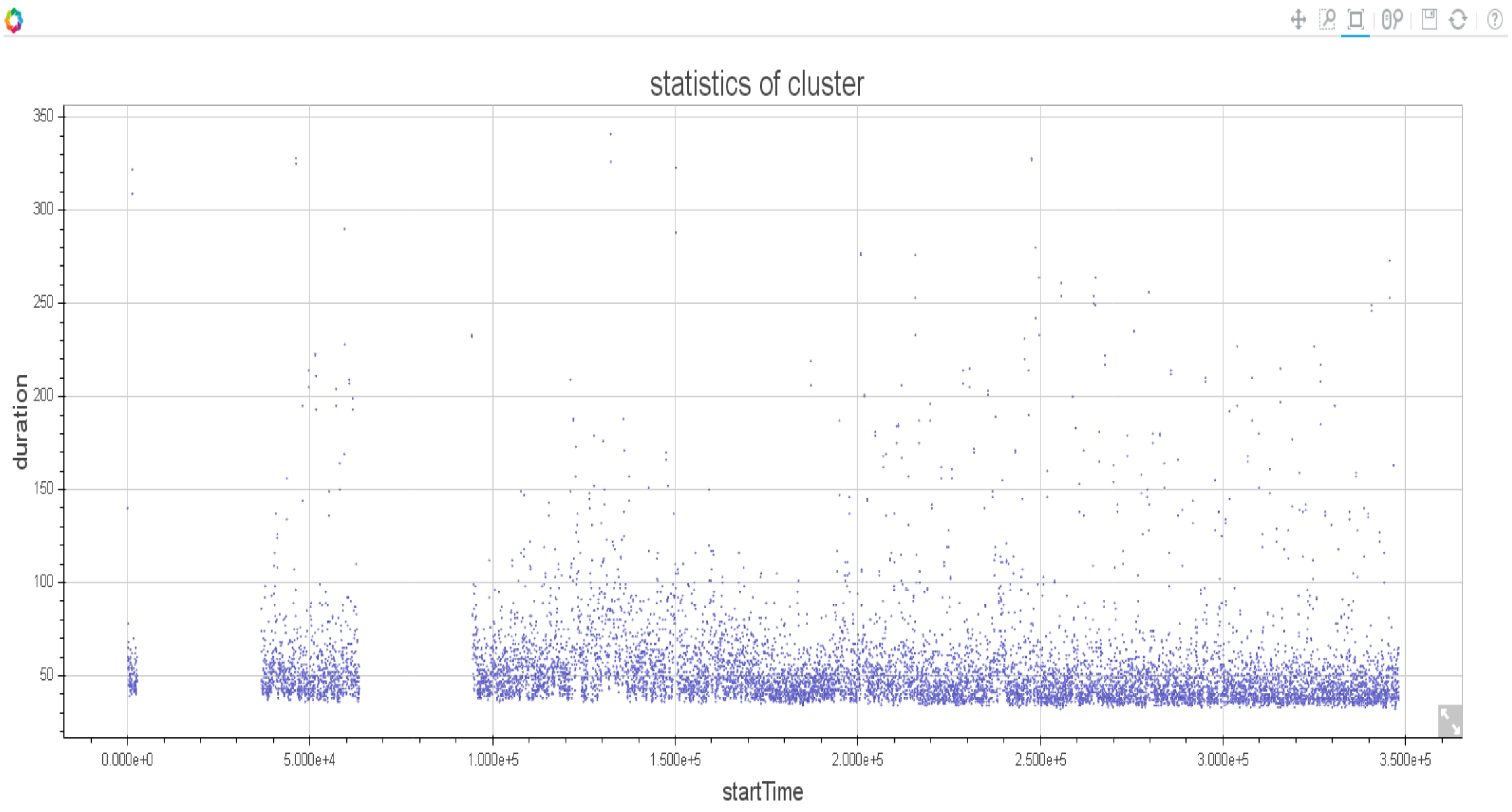
Consumer

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;

import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;

public class SimpleHLConsumer {
    private final ConsumerConnector consumer;
    private final String topic;

    public SimpleHLConsumer(String zookeeper, String groupId, String topic) {
        Properties props = new Properties();
        //定义连接zookeeper信息
        props.put("zookeeper.connect", zookeeper);
        //定义Consumer所有的groupID，关于groupID，后面会继续介绍
        props.put("group.id", groupId);
        props.put("zookeeper.session.timeout.ms", "500");
        props.put("zookeeper.sync.time.ms", "250");
        props.put("auto.commit.interval.ms", "1000");
        consumer = Consumer.createJavaConsumerConnector(new ConsumerConfig(props));
        this.topic = topic;
    }

public void testConsumer() {
    Map<String, Integer> topicCount = new HashMap<String, Integer>();
    //定义订阅topic数量
    topicCount.put(topic, new Integer(1));
    //返回的是所有topic的Map
    Map<String, List<KafkaStream<byte[], byte[]>>> consumerStreams = consumer.createMessageStreams(topicCount);
    //取出我们要需要的topic中的消息流
    List<KafkaStream<byte[], byte[]>> streams = consumerStreams.get(topic);
    for (final KafkaStream stream : streams) {
        ConsumerIterator<byte[], byte[]> consumerIte = stream.iterator();
        while (consumerIte.hasNext())
            System.out.println("Message from Single Topic :: " + new String(consumerIte.next().message()));
    }
    if (consumer != null)
        consumer.shutdown();
}
```

```
public static void main(String[] args) {
        String topic = "mytopic";
        SimpleHLConsumer simpleHLConsumer =
          new SimpleHLConsumer("192.168.4.32:2181", "testgroup", topic);
        simpleHLConsumer.testConsumer();
    }
```

示例

Spark Streaming, Flume and Kafka日志分析demo演示



Q & A & 讨论

董炫辰

IBM Platform Computing

xcdong@cn.ibm.com

谢谢！

To Be Continued

董炫辰

IBM Platform Computing

xcdong@cn.ibm.com