kafka 学习笔记

1，kafka 产生背景

2，kafka 简介

3，kafka 架构

4，kafka 部署

5，flume+kafka+hdfs+spark 集成

第一部分：kafka 产生背景

Kafka 是一个消息系统，原本开发自 LinkedIn，用作 LinkedIn 的活动流（Activity Stream）和运营数据处理管道（Pipeline）的基础。现在它已被多家不同类型的公司作为多种类型的数据管道和消息系统使用。

活动流数据是几乎所有站点在对其网站使用情况做报表时都要用到的数据中最常规的部分。活动数据包括页面访问量（Page View）、被查看内容方面的信息以及搜索情况等内容。这种数据通常的处理方式是先把各种活动以日志的形式写入某种文件，然后周期性地对这些文件进行统计分析。运营数据指的是服务器的性能数据（CPU、IO 使用率、请求时间、服务日志等等数据)。运营数据的统计方法种类繁多。

近年来，活动和运营数据处理已经成为了网站软件产品特性中一个至关重要的组成部分，这就需要一套稍微更加复杂的基础设施对其提供支持。

第二部分：kafka 简介

Kafka 是一种分布式的，基于发布/订阅的消息系统。主要设计目标如下：

- 以时间复杂度为 O(1)的方式提供消息持久化能力，即使对 TB 级以上数据也能保证常数时间复杂度的访问性能。
- 高吞吐率。即使在非常廉价的商用机器上也能做到单机支持每秒 100K 条以上消息的传输。
- 支持 Kafka Server 间的消息分区，及分布式消费，同时保证每个 Partition 内的消息顺序传输。
- 同时支持离线数据处理和实时数据处理。
- Scale out：支持在线水平扩展。

为何使用消息系统

•冗余
有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的"插入-获取-删除"范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

•可恢复性
系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

•顺序保证
在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。Kafka 保证一个 Partition 内的消息的有序性。

•缓冲
在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行———写入队列的处理会尽可能的快速。该缓冲有助于控制和优化数据流经过系统的速度。

•异步通信
很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

第三部分：kafka 架构

kafka 组件

Broker

Kafka 集群包含一个或多个服务器，这种服务器被称为 broker

•Topic

每条发布到 Kafka 集群的消息都有一个类别，这个类别被称为 Topic。（物理上不同 Topic 的消息分开存储，逻辑上一个 Topic 的消息虽然保存于一个或多个 broker 上但用户只需指定消息的 Topic 即可生产或消费数据而不必关心数据存于何处）

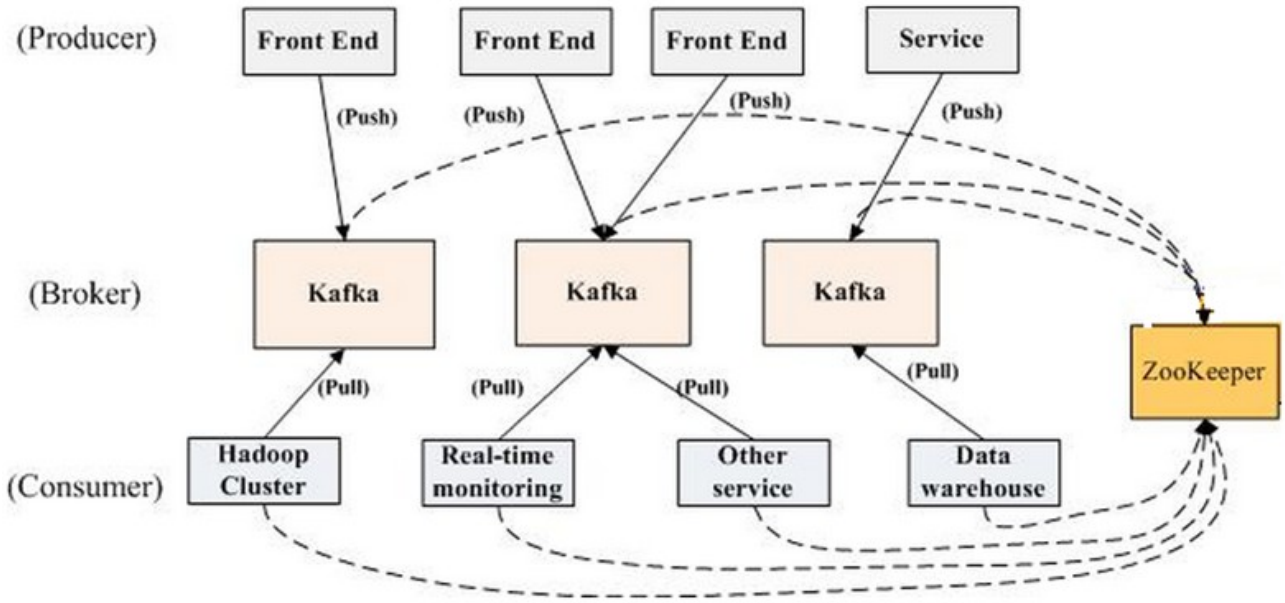
•Partition
Parition 是物理上的概念，每个 Topic 包含一个或多个 Partition.

•Producer
负责发布消息到 Kafka broker

•Consumer
消息消费者，向 Kafka broker 读取消息的客户端。

•Consumer Group
每个 Consumer 属于一个特定的 Consumer Group（可为每个 Consumer 指定 group name，若不指定 group name 则属于默认的 group）。

kafka 拓扑结构



如上图所示，一个典型的 Kafka 集群中包含若干 Producer（可以是 web 前端产生的 Page View，或者是服务器日志，系统 CPU、Memory 等），若干 broker（Kafka 支持水平扩展，一般 broker 数量越多，集群吞吐率越高），若干 Consumer Group，以及一个 Zookeeper 集群。Kafka 通过 Zookeeper 管理集群配置，选举 leader，以及在 Consumer Group 发生变化时进行 rebalance。Producer 使用 push 模式将消息发布到 broker，Consumer 使用 pull 模式从 broker 订阅并消费消息。

Topic & Partition

Topic 在逻辑上可以被认为是一个 queue，每条消费都必须指定它的 Topic，可以简单理解为必须指明把这条消息放进哪个 queue 里。为了使得 Kafka 的吞吐率可以线性提高，物理上把 Topic 分成一个或多个 Partition，每个 Partition 在物理上对应一个文件夹，该文件夹下存储这个 Partition 的所有消息和索引文件。若创建 topic1 和 topic2 两个 topic，且分别有 13 个和 19 个分区，则整个集群上会相应会生成共 32 个文件夹（本文所用集群共 8 个节点，此处 topic1 和 topic2 replication-factor 均为 1），如下图所示。

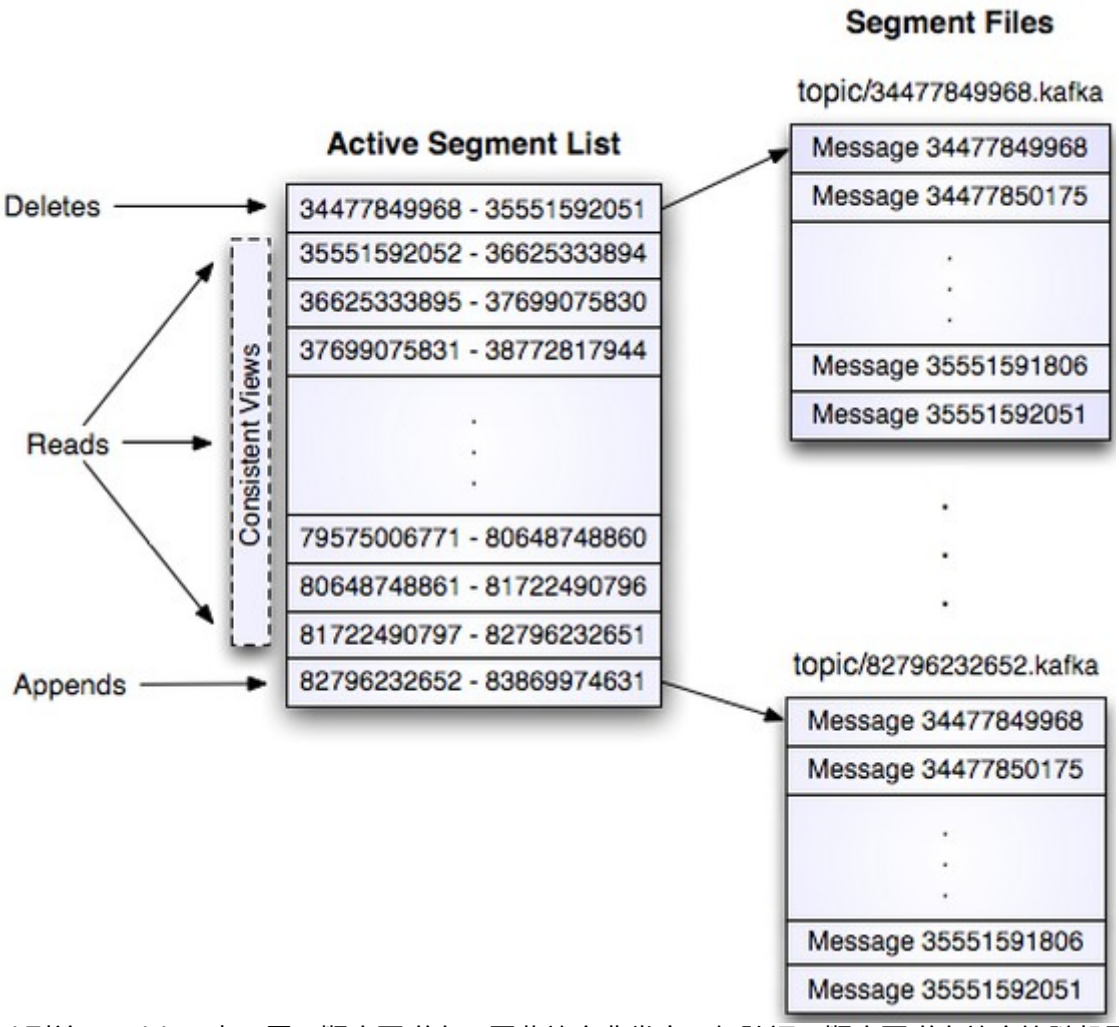
```
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-10
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-2
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-10
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-18
node4: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-2
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-0
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-8
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-0
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-16
node2: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-8
node8: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-6
node8: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-14
node8: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-6
node7: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-5
node7: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-13
node7: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-5
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-1
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-9
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-1
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-17
node3: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-9
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-12
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-4
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-12
node6: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-4
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-11
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-3
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-11
node5: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-3
node1: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic1-7
node1: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-15
node1: drwxr-xr-x 2 root root 4.0K Mar  3 13:01 topic2-7
```

每个日志文件都是一个 log entrie 序列，每个 log entrie 包含一个 4 字节整型数值（值为 N+5），1 个字节的"magic value"，4 个字节的 CRC 校验码，其后跟 N 个字节的消息体。每条消息都有一个当前 Partition 下唯一的 64 字节的 offset，它指明了这条消息的起始位置。磁盘上存储的消息格式如下：

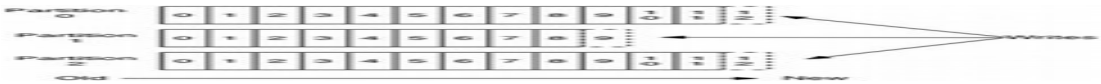
```
message length ： 4 bytes (value: 1+4+n)
"magic" value ： 1 byte
crc ： 4 bytes
payload ： n bytes
```

这个 log entries 并非由一个文件构成，而是分成多个 segment，每个 segment 以该 segment 第一条消息的 offset 命名并以“.kafka”为后缀。另外会有一个索引文件，它标明了每个 segment 下包含的 log entry 的 offset 范围，如下图所示。

因为每条消息都被



append 到该 Partition 中，属于顺序写磁盘，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是 Kafka 高吞吐率的一个很重要的保证）。

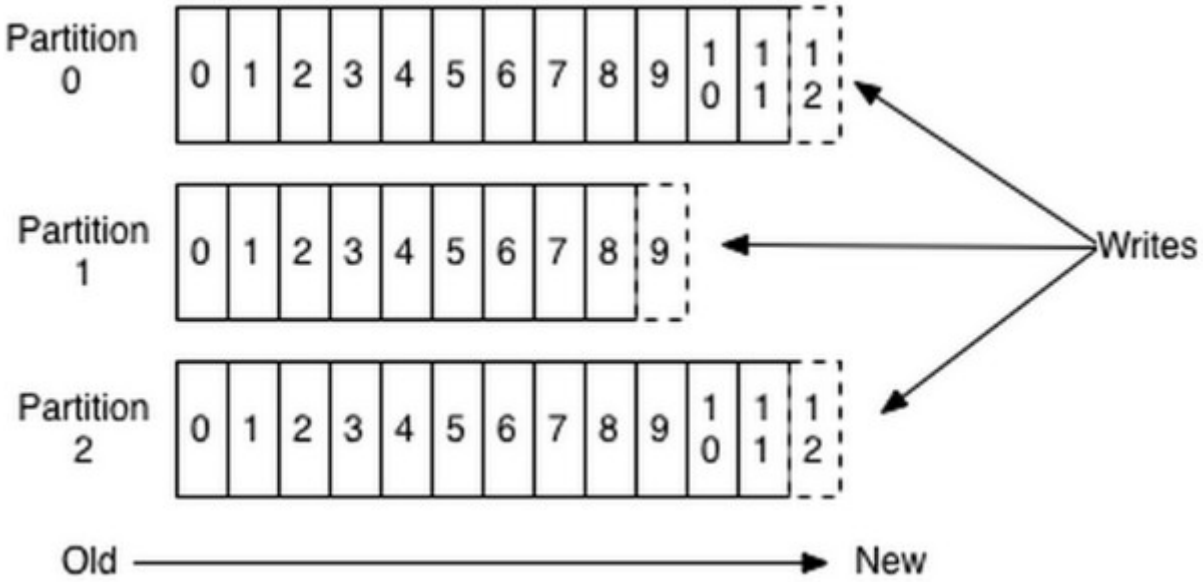


对于传统的 message queue 而言，一般会删除已经被消费的消息，而 Kafka 集群会保留所有的消息，无论其被消费与否。当然，因为磁盘限制，不可能永久保留所有数据（实际上也没必要），因此 Kafka 提供两种策略删除旧数据。一是基于时间，二是基于 Partition 文件大小。例如可以通过配置

$KAFKA_HOME/config/server.properties，让 Kafka 删除一周前的数据，也可在 Partition 文件超过 1GB 时删除旧数据，配置如下所示。

因为每条消息都被 append 到该 Partition 中，属于顺序写磁盘，因此效率非常高（经验证，顺序写磁盘效率比随机写内存还要高，这是 Kafka 高吞吐率的一个很重要的保证）。

对于传统的



message queue 而言，一般会删除已经被消费的消息，而 Kafka 集群会保留所有的消息，无论其被消费与否。当然，因为磁盘限制，不可能永久保留所有数据（实际上也没必要），因此 Kafka 提供两种策略删除旧数据。一是基于时间，二是基于 Partition 文件大小。例如可以通过配置 $KAFKA_HOME/config/server.properties，让 Kafka 删除一周前的数据，也可在 Partition 文件超过 1GB 时删除旧数据，配置如下所示。

```
# The minimum age of a log file to be eligible for deletion

log.retention.hours=168

# The maximum size of a log segment file. When this size is reached a new log segment will be created.

log.segment.bytes=1073741824

# The interval at which log segments are checked to see if they can be deleted according to the retention policies

log.retention.check.interval.ms=300000

# If log.cleaner.enable=true is set the cleaner will be enabled and individual logs can then be marked for log compaction.

log.cleaner.enable=false
```

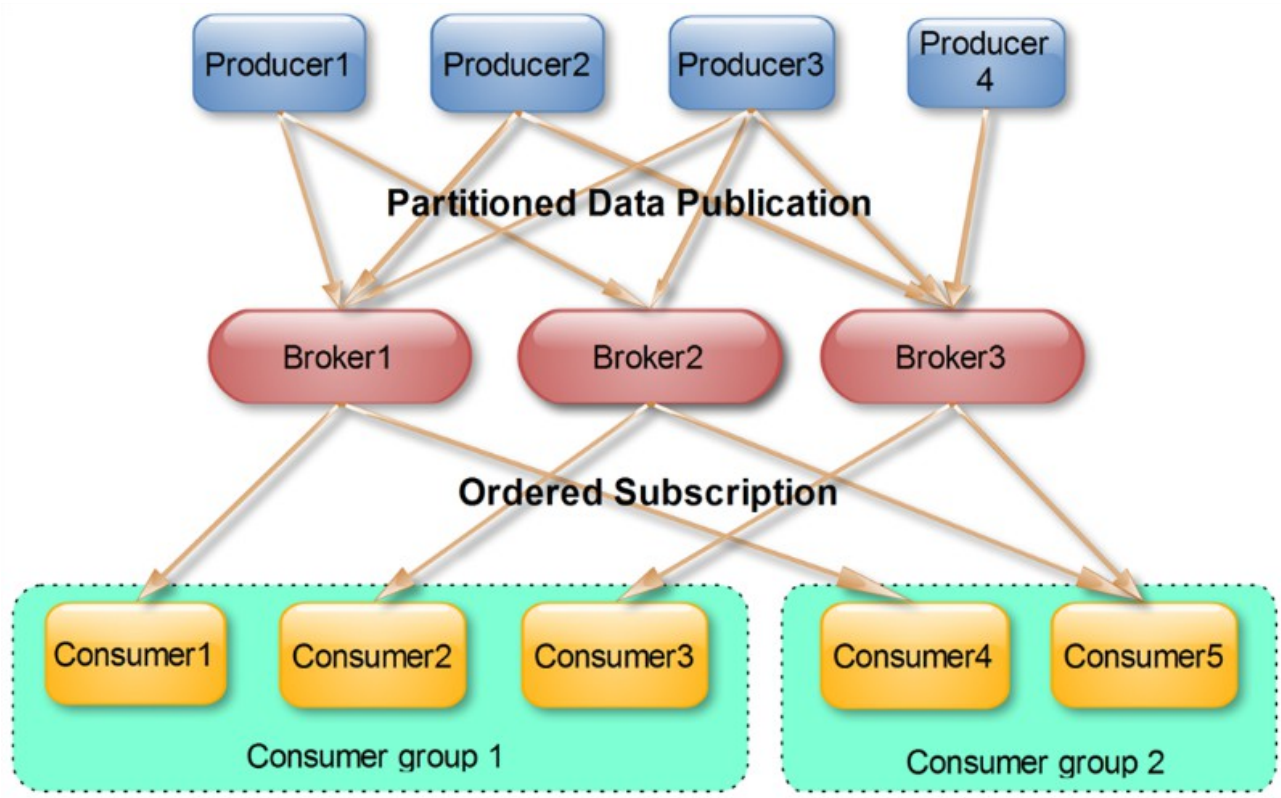
这里要注意，因为 Kafka 读取特定消息的时间复杂度为 O(1)，即与文件大小无关，所以这里删除过期文件与提高 Kafka 性能无关。选择怎样的删除策略只与磁盘以及具体的需求有关。另外，Kafka 会为每一个 Consumer Group 保留一些 metadata 信息——当前消费的消息的 position，也即 offset。这个 offset 由 Consumer 控制。正常情况下 Consumer 会在消费完一条消息后递增该 offset。当然，Consumer 也可将 offset 设成一个较小的值，重新消费一些消息。因为 offet 由 Consumer 控制，所以 Kafka broker 是无状态的，它不需要标记哪些消息被哪些消费过，也不需要通过 broker 去保证同一个 Consumer Group 只有一个 Consumer 能消费某一条消息，因此也就不需要锁机制，这也为 Kafka 的高吞吐率提供了有力保障。

Producer

Producer 发送消息到 broker 时，会根据 Paritition 机制选择将其存储到哪一个 Partition。如果 Partition 机制设置合理，所有消息可以均匀分布到不同的 Partition 里，这样就实现了负载均衡。如果一个 Topic 对应一个文件，那这个文件所在的机器 I/O 将会成为这个 Topic 的性能瓶颈，而有了 Partition 后，不同的消息可以并行写入不同 broker 的不同 Partition 里，极大的提高了吞吐率。可以在 $KAFKA_HOME/config/server.properties 中通过配置项 num.partitions 来指定新建 Topic 的默认 Partition 数量，也可在创建 Topic 时通过参数指定，同时也可以在 Topic 创建之后通过 Kafka 提供的工具修改。

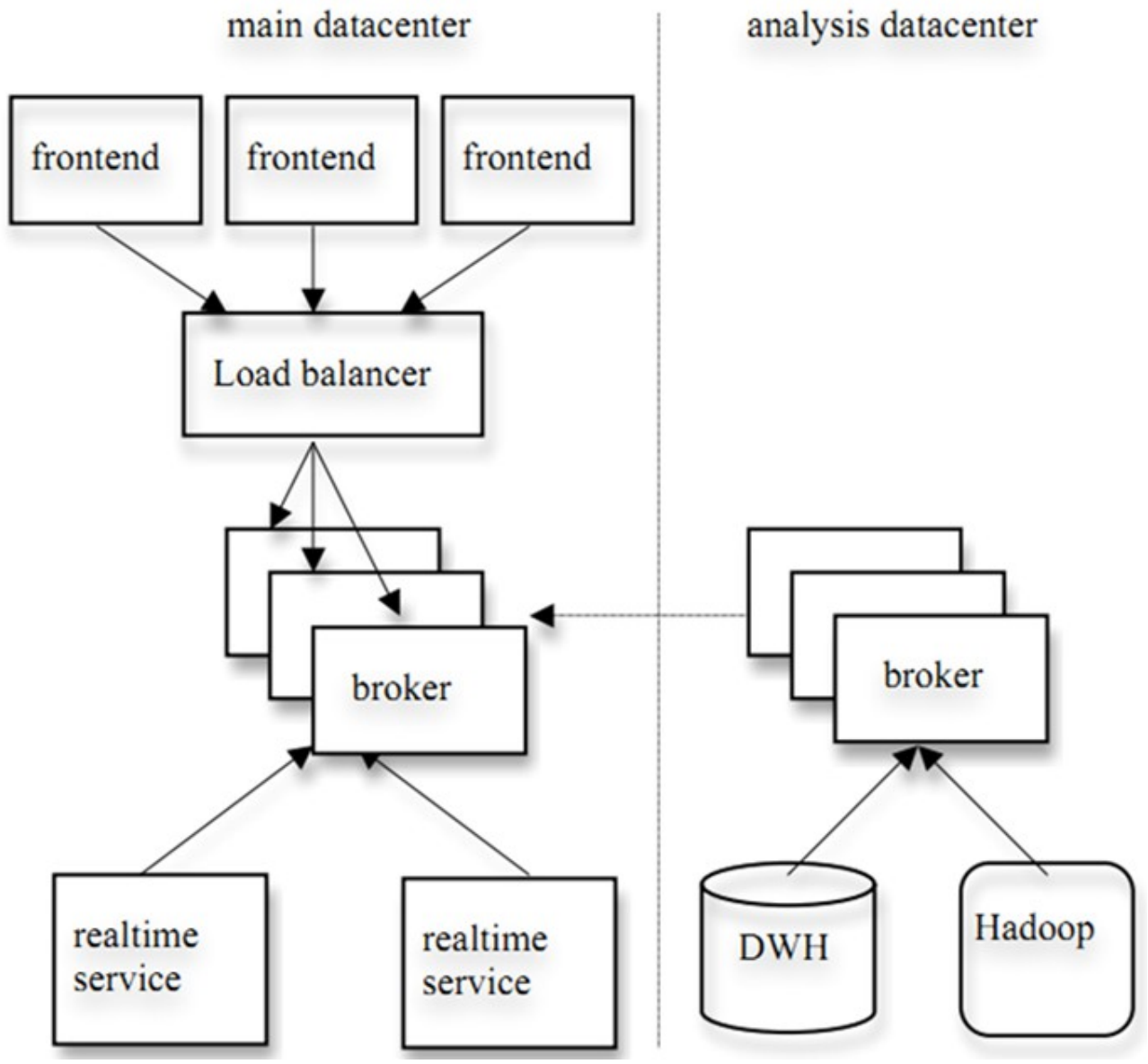
Consumer Group

使用 Consumer high level API 时，同一 Topic 的一条消息只能被同一个 Consumer Group 内的一个 Consumer 消费，但多个 Consumer Group 可同时消费这一消息。

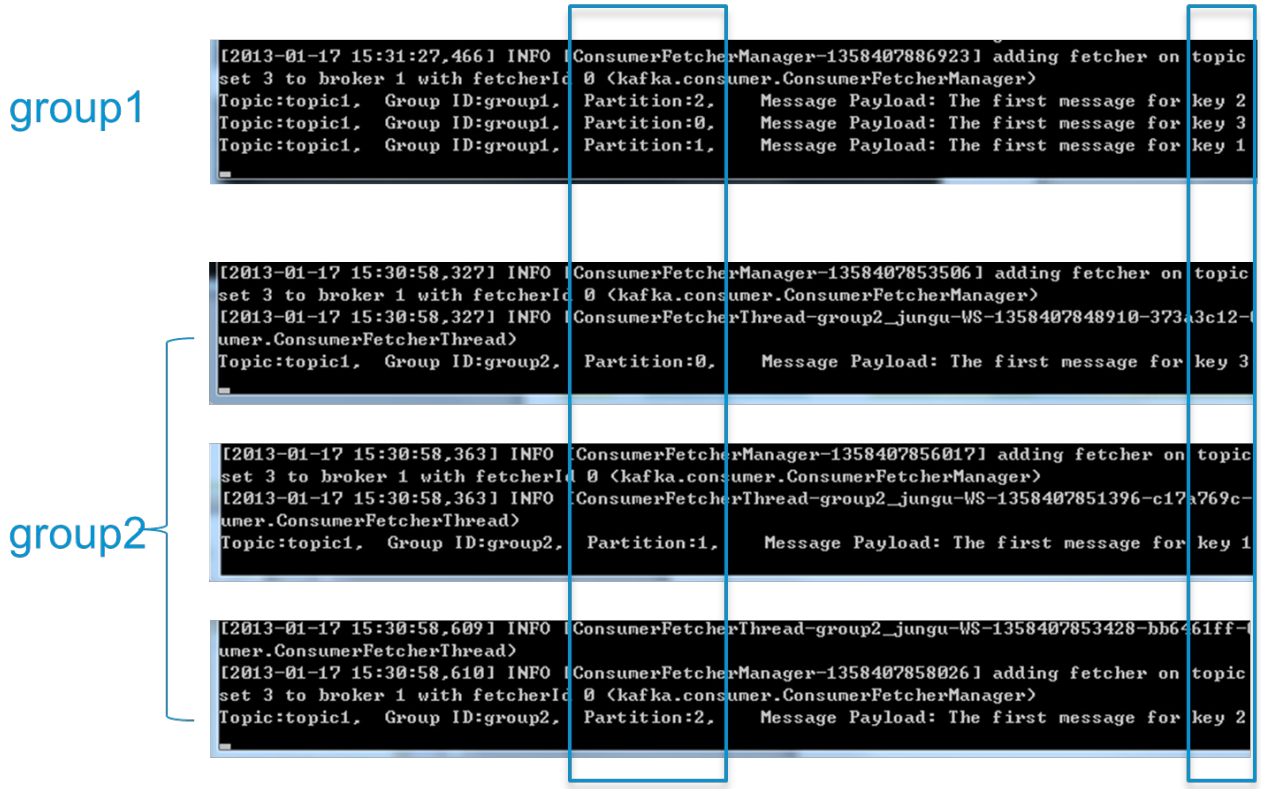


这是 Kafka 用来实现一个 Topic 消息的广播（发给所有的 Consumer）和单播（发给某一个 Consumer）的手段。一个 Topic 可以对应多个 Consumer Group。如果需要实现广播，只要每个 Consumer 有一个独立的 Group 就可以了。要实现单播只要所有的 Consumer 在同一个 Group 里。用 Consumer Group 还可以将 Consumer 进行自由的分组而不需要多次发送消息到不同的 Topic。

实际上，Kafka 的设计理念之一就是同时提供离线处理和实时处理。根据这一特性，可以使用 Storm 这种实时流处理系统对消息进行实时在线处理，同时使用 Hadoop 这种批处理系统进行离线处理，还可以同时将数据实时备份到另一个数据中心，只需要保证这三个操作所使用的 Consumer 属于不同的 Consumer Group 即可。下图是 Kafka 在 Linkedin 的一种简化部署示意图。



下面这个例子更清晰地展示了 Kafka Consumer Group 的特性。首先创建一个 Topic (名为 topic1，包含 3 个 Partition)，然后创建一个属于 group1 的 Consumer 实例，并创建三个属于 group2 的 Consumer 实例，最后通过 Producer 向 topic1 发送 key 分别为 1，2，3 的消息。结果发现属于 group1 的 Consumer 收到了所有的这三条消息，同时 group2 中的 3 个 Consumer 分别收到了 key 为 1，2，3 的消息。如下图所示。



Push vs. Pull

作为一个消息系统，Kafka 遵循了传统的方式，选择由 Producer 向 broker push 消息并由 Consumer 从 broker pull 消息。一些 logging-centric system，比如 Facebook 的 Scribe 和 Cloudera 的 Flume，采用 push 模式。事实上，push 模式和 pull 模式各有优劣。

push 模式很难适应消费速率不同的消费者，因为消息发送速率是由 broker 决定的。push 模式的目标是尽可能以最快速度传递消息，但是这样很容易造成 Consumer 来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而 pull 模式则可以根据 Consumer 的消费能力以适当的速率消费消息。

对于 Kafka 而言，pull 模式更合适。pull 模式可简化 broker 的设计，Consumer 可自主控制消费消息的速率，同时 Consumer 可以自己控制消费方式——即可批量消费也可逐条消费，同时还能选择不同的提交方式从而实现不同的传输语义。

Kafka delivery guarantee

有这么几种可能的 delivery guarantee：

- At most once 消息可能会丢，但绝不会重复传输
- At least one 消息绝不会丢，但可能会重复传输
- Exactly once 每条消息肯定会被传输一次且仅传输一次，很多时候这是用户所想要的。

当 Producer 向 broker 发送消息时，一旦这条消息被 commit，因数 replication 的存在，它就不会丢。但是如果 Producer 发送数据给 broker 后，遇到网络问题而造成通信中断，那 Producer 就无法判断该条消息是否已经 commit。虽然 Kafka 无法确定网络故障期间发生了什么，但是 Producer 可以生成一种类似于主键的东西，发生故障时幂等性的重试多次，这样就做到了 Exactly once。截止到目前(Kafka 0.8.2 版本，2015-03-04)，这一 Feature 还并未实现，有希望在 Kafka 未来的版本中实现。（所以目前默认情况下一条消息从 Producer 到 broker 是确保了 At least once，可通过设置 Producer 异步发送实现 At most once）。

接下来讨论的是消息从 broker 到 Consumer 的 delivery guarantee 语义。（仅针对 Kafka consumer high level API）。Consumer 在从 broker 读取消息后，可以选择 commit，该操作会在 Zookeeper 中保存该 Consumer 在该 Partition 中读取的消息的 offset。该 Consumer 下一次再读该 Partition 时会从下一条开始读取。如未 commit，下一次读取的开始位置会跟上一次 commit 之后的开始位置相同。当然可以将 Consumer 设置为 autocommit，即 Consumer 一旦读到数据立即自动 commit。如果只讨论这一读取消息的过程，那 Kafka 是确保了 Exactly once。但实际使用中应用程序并非在 Consumer 读取完数据就结束了，而是要进行进一步处理，而数据处理与 commit 的顺序在很大程度上决定了消息从 broker 和 consumer 的 delivery guarantee semantic。

•读完消息先 commit 再处理消息。这种模式下，如果 Consumer 在 commit 后还没来得及处理消息就 crash 了，下次重新开始工作后就无法读到刚刚已提交而未处理的消息，这就对应于 At most once

•读完消息先处理再 commit。这种模式下，如果在处理完消息之后 commit 之前 Consumer crash 了，下次重新开始工作时还会处理刚刚未 commit 的消息，实际上该消息已经被处理过了。这就对应于 At least once。在很多使用场景下，消息都有一个主键，所以消息的处理往往具有幂等性，即多次处理这一条消息跟只处理一次是等效的，那就可以认为是 Exactly once。（笔者认为这种说法比较牵强，毕竟它不是 Kafka 本身提供的机制，主键本身也并不能完全保证操作的幂等性。而且实际上我们说 delivery guarantee 语义是讨论被处理多少次，而非处理结果怎样，因为处理方式多种多样，我们不应该把处理过程的特性——如是否幂等性，当成 Kafka 本身的 Feature）

•如果一定要做到 Exactly once，就需要协调 offset 和实际操作的输出。精典的做法是引入两阶段提交。如果能让 offset 和操作输入存在同一个地方，会更简洁和通用。这种方式可能更好，因为许多输出系统可能不支持两阶段提交。比如，Consumer 拿到数据后可能把数据放到 HDFS，如果把最新的 offset 和数据本身一起写到 HDFS，那就可以保证数据的输出和 offset 的更新要么都完成，

要么都不完成，间接实现 Exactly once。（目前就 high level API 而言，offset 是存于 Zookeeper 中的，无法存于 HDFS，而 low level API 的 offset 是由自己去维护的，可以将之存于 HDFS 中）

总之，Kafka 默认保证 At least once，并且允许通过设置 Producer 异步提交来实现 At most once。而 Exactly once 要求与外部存储系统协作，幸运的是 Kafka 提供的 offset 可以非常直接非常容易得使用这种方式。

来源：http://www.infoq.com/cn/articles/kafka-analysis-part-1

第四部分：kafka 部署

单机环境部署

1，官网下载最新版本

wget https://www.apache.org/dyn/closer.cgi?path=/kafka/0.9.0.1/kafka_2.11-0.9.0.1.tgz

tar -xzvf kafka_2.11-0.9.0.1.tgz

2，启动 zookeeper

如果你集群中本身就有 zookeeper 就不用启动了

bin/zookeeper-server-start.sh config/zookeeper.properties

3，启动 kafka

先修改好配置：

$cat config/server.properties |grep ^[^#]

broker.id=0

listeners=PLAINTEXT://:9092

num.network.threads=3

num.io.threads=8

socket.send.buffer.bytes=102400

socket.receive.buffer.bytes=102400

socket.request.max.bytes=104857600

log.dirs=/tmp/kafka-logs

num.partitions=1

num.recovery.threads.per.data.dir=1

log.retention.hours=168

log.segment.bytes=1073741824

log.retention.check.interval.ms=300000

zookeeper.connect=localhost:2181

zookeeper.connection.timeout.ms=6000

主要是 zookeeper 的配置，如果自己的 zookeeper 集群，则需要修改

然后就是直接启动

bin/kafka-server-start.sh config/server.properties

4，创建 topic

bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test

5，查看所创建的 topic

bin/kafka-topics.sh --list --zookeeper localhost:2181

or

bin/kafka-topics.sh --desdribe --zookeeper localhost:2181 --topic

6，producer 发送消息到 topic

bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test

This is a message

This is another message

7，consumer 消费消息

bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning

This is a message

This is another message

集群环境部署

和上面差不多，就是 zookeeper 部署多台，然后在多台启动 kafka 就 ok 了

注意 broker.id 要集群中唯一

直接从官网拷贝：

First we make a config file for each of the brokers:

```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
```

Now edit these new files and set the following properties:

```
config/server-1.properties:
    broker.id=1
    port=9093
    log.dir=/tmp/kafka-logs-1

config/server-2.properties:
    broker.id=2
    port=9094
    log.dir=/tmp/kafka-logs-2
```

The `broker.id` property is the unique and permanent name of each node in the cluster. We have to override the port and log directory only because we are running these all on the same machine and we want to keep the brokers from all trying to register on the same port or overwrite each others data.

We already have Zookeeper and our single node started, so we just need to start the two new nodes:

```
> bin/kafka-server-start.sh config/server-1.properties &
...
> bin/kafka-server-start.sh config/server-2.properties &
...
```

Now create a new topic with a replication factor of three:

```
> bin/kafka-topics.sh --create --zookeeper localhost:2181
--replication-factor 3 --partitions 1 --topic my-replicated-
topic
```

Okay but now that we have a cluster how can we know which broker is doing what? To see that run the "describe topics" command:

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181
--topic my-replicated-topic
Topic:my-replicated-topic       PartitionCount:1
ReplicationFactor:3     Configs:
      Topic: my-replicated-topic      Partition: 0    Leader:
1       Replicas: 1,2,0 Isr: 1,2,0
```

Here is an explanation of output. The first line gives a summary of all the partitions, each additional line gives information about one partition. Since we have only one partition for this topic there is only one line.

- "leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion of the partitions.
- "replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive.
- "isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-up to the leader.

Note that in my example node 1 is the leader for the only partition of the topic.

We can run the same command on the original topic we created to see where it is:

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181
--topic test
Topic:test      PartitionCount:1        ReplicationFactor:1
Configs:
      Topic: test     Partition: 0    Leader: 0
Replicas: 0     Isr: 0
```

So there is no surprise there—the original topic has no replicas and is on server 0, the only server in our cluster when we created it.

Let's publish a few messages to our new topic:

```
> bin/kafka-console-producer.sh --broker-list localhost:9092
--topic my-replicated-topic
...
my test message 1
my test message 2
^C
```

Now let's consume these messages:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181
--from-beginning --topic my-replicated-topic
...
my test message 1
my test message 2
^C
```

Now let's test out fault-tolerance. Broker 1 was acting as the leader so let's kill it:

```
> ps | grep server-1.properties
7564 ttys002    0:15.91
/System/Library/Frameworks/JavaVM.framework/Versions/1.6/Home
/bin/java...
> kill -9 7564
```

Leadership has switched to one of the slaves and node 1 is no longer in the in-sync replica set:

```
> bin/kafka-topics.sh --describe --zookeeper localhost:2181
--topic my-replicated-topic
Topic:my-replicated-topic       PartitionCount:1
ReplicationFactor:3     Configs:
      Topic: my-replicated-topic      Partition: 0    Leader:
2       Replicas: 1,2,0 Isr: 2,0
```

But the messages are still be available for consumption even though the leader that took the writes originally is down:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181
--from-beginning --topic my-replicated-topic
...
my test message 1
my test message 2
^C
```

配置参数介绍

系统参数

```
    #唯一标识在集群中的 ID，要求是正数。
broker.id=0
#服务端口，默认 9092
port=9092
#监听地址
host.name=debugo01

    # 处理网络请求的最大线程数
num.network.threads=2
# 处理磁盘 I/O 的线程数
num.io.threads=8
# 一些后台线程数
background.threads = 4
# 等待 IO 线程处理的请求队列最大数
queued.max.requests = 500

    #  socket 的发送缓冲区（SO_SNDBUF）
socket.send.buffer.bytes=1048576
# socket 的接收缓冲区 (SO_RCVBUF)
socket.receive.buffer.bytes=1048576
# socket 请求的最大字节数。为了防止内存溢出，message.max.bytes 必然要小于
socket.request.max.bytes = 104857600
```

Topic 参数

```
    # 每个 topic 的分区个数，更多的 partition 会产生更多的 segment file
num.partitions=2
# 是否允许自动创建 topic ，若是 false，就需要通过命令创建 topic
auto.create.topics.enable =true
# 一个 topic ，默认分区的 replication 个数 ，不能大于集群中 broker 的个数。
default.replication.factor =1
```

# 消息体的最大大小，单位是字节
message.max.bytes = 1000000

ZooKeeper 参数

# Zookeeper quorum 设置。如果有多个使用逗号分割
zookeeper.connect=debugo01:2181,debugo02,debugo03
# 连接 zk 的超时时间
zookeeper.connection.timeout.ms=1000000
# ZooKeeper 集群中 leader 和 follower 之间的同步实际
zookeeper.sync.time.ms = 2000

日志参数

#日志存放目录，多个目录使用逗号分割
log.dirs=/var/log/kafka

当达到下面的消息数量时，会将数据 flush 到日志文件中。默认 10000
#log.flush.interval.messages=10000
当达到下面的时间(ms)时，执行一次强制的 flush 操作。interval.ms 和 interval.messages 无论哪个达到，都会 flush。默认 3000ms
#log.flush.interval.ms=1000
# 检查是否需要将日志 flush 的时间间隔
log.flush.scheduler.interval.ms = 3000

# 日志清理策略（delete|compact）
log.cleanup.policy = delete
# 日志保存时间 (hours|minutes)，默认为 7 天（168 小时）。超过这个时间会根据 policy 处理数据。bytes 和 minutes 无论哪个先达到都会触发。
log.retention.hours=168
# 日志数据存储的最大字节数。超过这个时间会根据 policy 处理数据。
#log.retention.bytes=1073741824

# 控制日志 segment 文件的大小，超出该大小则追加到一个新的日志 segment 文件中（-1 表示没有限制）
log.segment.bytes=536870912
# 当达到下面时间，会强制新建一个 segment
log.roll.hours = 24*7
# 日志片段文件的检查周期，查看它们是否达到了删除策略的设置（log.retention.hours 或

log.retention.bytes)
log.retention.check.interval.ms=60000

# 是否开启压缩
log.cleaner.enable=false
# 对于压缩的日志保留的最长时间
log.cleaner.delete.retention.ms = 1 day

# 对于 segment 日志的索引文件大小限制
log.index.size.max.bytes = 10 * 1024 * 1024
#y 索引计算的一个缓冲区，一般不需要设置。
log.index.interval.bytes = 4096

副本参数

# partition management controller 与 replicas 之间通讯的超时时间
controller.socket.timeout.ms = 30000
# controller-to-broker-channels 消息队列的尺寸大小
controller.message.queue.size=10
# replicas 响应 leader 的最长等待时间，若是超过这个时间，就将 replicas 排除在管理之外
replica.lag.time.max.ms = 10000
# 是否允许控制器关闭 broker ,若是设置为 true,会关闭所有在这个 broker 上的 leader，并转移到其他 broker
controlled.shutdown.enable = false
# 控制器关闭的尝试次数
controlled.shutdown.max.retries = 3
# 每次关闭尝试的时间间隔
controlled.shutdown.retry.backoff.ms = 5000

# 如果 relicas 落后太多,将会认为此 partition relicas 已经失效。而一般情况下,因为网络延迟等原因,总会导致 replicas 中消息同步滞后。如果消息严重滞后,leader 将认为此 relicas 网络延迟较大或者消息吞吐能力有限。在 broker 数量较少,或者网络不足的环境中,建议提高此值.
replica.lag.max.messages = 4000
#leader 与 relicas 的 socket 超时时间
replica.socket.timeout.ms= 30 * 1000
# leader 复制的 socket 缓存大小
replica.socket.receive.buffer.bytes=64 * 1024

# replicas 每次获取数据的最大字节数
replica.fetch.max.bytes = 1024 * 1024
# replicas 同 leader 之间通信的最大等待时间，失败了会重试
replica.fetch.wait.max.ms = 500
# 每一个 fetch 操作的最小数据尺寸,如果 leader 中尚未同步的数据不足此值,将会等待直到数据达到这个大小
replica.fetch.min.bytes =1
# leader 中进行复制的线程数，增大这个数值会增加 relipca 的 IO
num.replica.fetchers = 1
# 每个 replica 将最高水位进行 flush 的时间间隔
replica.high.watermark.checkpoint.interval.ms = 5000

# 是否自动平衡 broker 之间的分配策略
auto.leader.rebalance.enable = false
# leader 的不平衡比例，若是超过这个数值，会对分区进行重新的平衡
leader.imbalance.per.broker.percentage = 10
# 检查 leader 是否不平衡的时间间隔
leader.imbalance.check.interval.seconds = 300
# 客户端保留 offset 信息的最大空间大小
offset.metadata.max.bytes = 1024

消费者参数

Consumer 端核心的配置是 group.id、zookeeper.connect
决定该 Consumer 归属的唯一组 ID，By setting the same group id multiple processes indicate that they are all part of the same consumer group.
group.id
消费者的 ID，若是没有设置的话，会自增
consumer.id
# 一个用于跟踪调查的 ID ，最好同 group.id 相同
client.id = <group_id>

# 对于 zookeeper 集群的指定，必须和 broker 使用同样的 zk 配置
zookeeper.connect=debugo01:2182,debugo02:2182,debugo03:2182
# zookeeper 的心跳超时时间，查过这个时间就认为是无效的消费者
zookeeper.session.timeout.ms = 6000
# zookeeper 的等待连接时间
zookeeper.connection.timeout.ms = 6000

```
# zookeeper 的 follower 同 leader 的同步时间
zookeeper.sync.time.ms = 2000
# 当 zookeeper 中没有初始的 offset 时，或者超出 offset 上限时的处理方式 。
# smallest ：重置为最小值
# largest:重置为最大值
# anything else：抛出异常给 consumer
auto.offset.reset = largest

    # socket 的超时时间，实际的超时时间为 max.fetch.wait + socket.timeout.ms.
socket.timeout.ms= 30 * 1000
# socket 的接收缓存空间大小
socket.receive.buffer.bytes=64 * 1024
#从每个分区 fetch 的消息大小限制
fetch.message.max.bytes = 1024 * 1024

    # true 时，Consumer 会在消费消息后将 offset 同步到 zookeeper，这样当 Consumer 失败后，新的 consumer 就能从 zookeeper 获取最新的 offset
auto.commit.enable = true
# 自动提交的时间间隔
auto.commit.interval.ms = 60 * 1000

    # 用于消费的最大数量的消息块缓冲大小，每个块可以等同于 fetch.message.max.bytes 中数值
queued.max.message.chunks = 10

    # 当有新的 consumer 加入到 group 时,将尝试 reblance,将 partitions 的消费端迁移到新的 consumer 中, 该设置是尝试的次数
rebalance.max.retries = 4
# 每次 reblance 的时间间隔
rebalance.backoff.ms = 2000
# 每次重新选举 leader 的时间
refresh.leader.backoff.ms

    # server 发送到消费端的最小数据，若是不满足这个数值则会等待直到满足指定大小。默认为 1 表示立即接收。
fetch.min.bytes = 1
# 若是不满足 fetch.min.bytes 时，等待消费端请求的最长等待时间
fetch.wait.max.ms = 100
```

# 如果指定时间内没有新消息可用于消费，就抛出异常，默认-1 表示不受限
consumer.timeout.ms = -1

生产者参数

核心的配置包括：
metadata.broker.list
request.required.acks
producer.type
serializer.class

消费者获取消息元信息(topics, partitions and replicas)的地址,配置格式是：host1:port1,host2:port2，也可以在外面设置一个 vip
metadata.broker.list

#消息的确认模式
# 0：不保证消息的到达确认，只管发送，低延迟但是会出现消息的丢失，在某个 server 失败的情况下，有点像 TCP
# 1：发送消息，并会等待 leader 收到确认后，一定的可靠性
# -1：发送消息，等待 leader 收到确认，并进行复制操作后，才返回，最高的可靠性
request.required.acks = 0

# 消息发送的最长等待时间
request.timeout.ms = 10000
# socket 的缓存大小
send.buffer.bytes=100*1024
# key 的序列化方式，若是没有设置，同 serializer.class
key.serializer.class
# 分区的策略，默认是取模
partitioner.class=kafka.producer.DefaultPartitioner
# 消息的压缩模式，默认是 none，可以有 gzip 和 snappy
compression.codec = none
# 可以针对默写特定的 topic 进行压缩
compressed.topics=null
# 消息发送失败后的重试次数
message.send.max.retries = 3
# 每次失败后的间隔时间
retry.backoff.ms = 100

# 生产者定时更新 topic 元信息的时间间隔 ，若是设置为 0，那么会在每个消息发送后都去更新数据
topic.metadata.refresh.interval.ms = 600 * 1000
# 用户随意指定，但是不能重复，主要用于跟踪记录消息
client.id=””

# 异步模式下缓冲数据的最大时间。例如设置为 100 则会集合 100ms 内的消息后发送，这样会提高吞吐量，但是会增加消息发送的延时
queue.buffering.max.ms = 5000
# 异步模式下缓冲的最大消息数，同上
queue.buffering.max.messages = 10000
# 异步模式下，消息进入队列的等待时间。若是设置为 0，则消息不等待，如果进入不了队列，则直接被抛弃
queue.enqueue.timeout.ms = -1
# 异步模式下，每次发送的消息数，当 queue.buffering.max.messages 或 queue.buffering.max.ms 满足条件之一时 producer 会触发发送。
batch.num.messages=200

注：本文中的参数不一定是最新版本的 Kafka。

来源：http://debugo.com/kafka-params/

参考：http://kafka.apache.org/documentation.html#configuration

第五部分：flume+kafka+hdfs+spark 集成

agent1.channels.c1.type = file

agent1.channels.c1.checkpointDir = /mnt/flume/checkpoint

agent1.channels.c1.dataDirs = /mnt/flume/data

agent1.channels.c1.transactionCapacity = 10000

agent1.channels.c1.maxFileSize = 2146435071

agent1.channels.c1.capacity = 1000000

agent1.channels.c1.keep-alive = 60

参考文档：

http://kafka.apache.org/documentation.html#majordesignelements

http://www.infoq.com/cn/articles/kafka-analysis-part-1/