# 1. 概述

## 1.1 定义

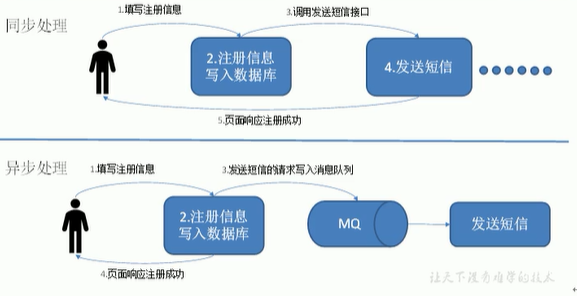
kafka是一个分布式的基于发布/订阅模式的消息队列（消息中间件）。

主要应用于大数据实时处理领域。

## 1.2 消息队列

### 1.2.1 传统消息队列的应用场景

MQ传统应用的场景



### 1.2.2 使用消息队列的好处

1. 解耦（类似Spring的Ioc）

允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

2. 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。

3. 缓冲

有助于控制和优化数据流经过系统的速度， 解决生产消息和消费消息的处理速度不一致的情况。

4. 灵活性（动态增减机器） & 峰值处理能力（削峰）

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

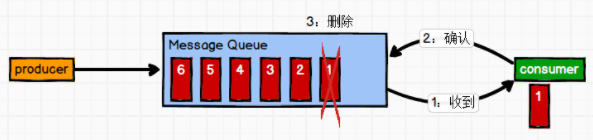
5. 异步通信

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。

### 1.2.3 消息队列的两种模式

#### 1. 点对点模式

一对一，消费者主动拉取数据，消息收到后消息清除



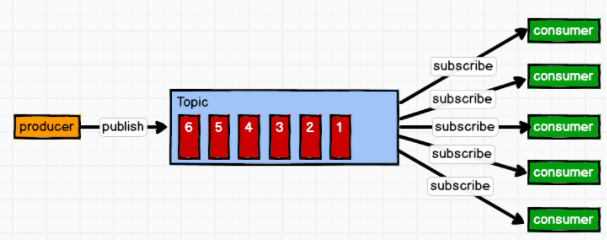
消息生产者生产消息发送到Queue中，然后消息消费者从Queue中取出并且消费消息。

消息被消费以后，Queue中不再有存储，所以消息消费者不可能消费到已经被消费的消息。

Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。

#### 2. 发布/订阅模式

一对多，消费者消费数据之后不会清除消息



消息生产者（发布）将消息发布到topic（Queue）中，同时有多个消息消费者（订阅）消费该消息。

和点对点方式不同，发布到topic的消息会被所有订阅者消费。

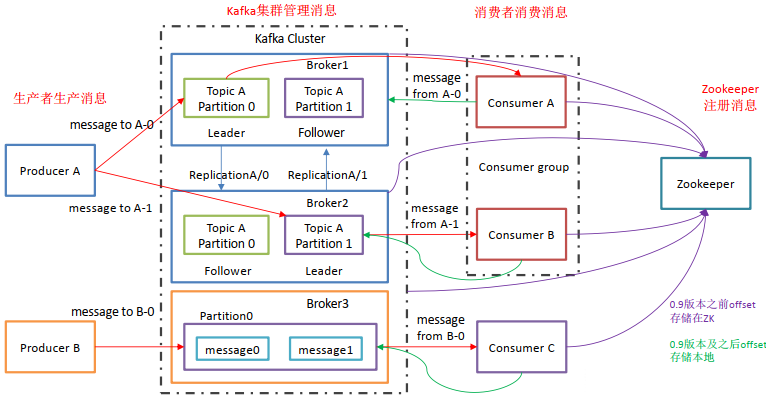
基于发布/定于模式，消费者有2种方式获取消息：

1. 队列主动推送，好处是订阅者可以同时获取到，坏处是由于不同消费者的消费速度是不一样的，但是推送的消息又是一定的，可能会造成消费端压力

2. 消费端主动拉取，好处是主动权在消费者手里，决定何时处理消息，坏处是为了感知到队列中消息的变化，需要保持一个长连接，不断轮询获取消息，有性能损耗

kafka选用的是拉取的模式，同时维护了一个长轮询，可以实时监控到队列中消息的变化。

## 1.3 基础架构



基础概念：

1. Producer

消息生产者。

向Kafka丢数据。

2. Consumer

消息消费者。

从Kafka broker取消息的客户端。

3. Consumer Group（CG）

消费者组。

由多个consumer组成。

消费者组内每个消费者负责消费不同分区的数据，一个分区只能由一个组内消费者消费。消费者组之间互不影响。

所有的消费者都属于某个消费者组，即消费者组是逻辑上的一个订阅者。

1. Broker

经纪人。

一台Kafka服务器就是一个broker（或者说一个kafka server进程就是一个broker）。

一个集群由多个broker组成。

一个broker可以容纳多个topic。

1. Topic

话题。

可以理解为一个队列， 生产者和消费者面向的都是一个topic。

1. Partition

为了实现Topic扩展性。

一个非常大的topic可以分布到多个broker（即服务器）上，一个topic可以分为多个partition，每个partition是一个有序的队列。

1. Replica

副本（Replication）。

为保证集群中的某个节点发生故障时，该节点上的partition数据不丢失，且Kafka仍然能够继续工作，Kafka提供了副本机制。

一个topic的每个分区都有若干个副本，一个leader和若干个follower。

1. Leader

每个分区多个副本的“主”。

生产者发送数据的对象，以及消费者消费数据的对象都是leader。

1. Follower

每个分区多个副本中的“从”。

实时从leader中同步数据，保持和leader数据的同步。

leader发生故障时，某个Follower会成为新的leader。

kafka为啥放弃offset放到zk中？

zk是作为框架之间一个润滑剂的存在。消费者本身消费是很快的，offset快速变化，消费端需要频繁和zk交互，有额外的性能。offset本身可以存到kafka本地上，这样消费端就减少了zk的连接。

# 2. 快速入门

## 2.1 安装部署

### 2.1.1 集群规划

hadoop1

zk1

kafka1

hadoop2

zk2

kafka2

hadoop3

zk3

kafka3

### 2.1.2 jar包下载

http://kafka.apache.org/downloads

官网安装教程

http://kafka.apache.org/0110/documentation/#quickstart

早期版本

kafka\_2.11-0.8.2.1.tgz

2.11是Scala版本，0.8.2.1是kafka版本

```

Step 1: Download the code

Download the 0.11.0.2 release and un-tar it.

> tar -xzf kafka\_2.11-0.11.0.2.tgz -C /opt/tools/

> cd /opt/tools/kafka\_2.11-0.11.0.2

> mv kafka\_2.11-0.11.0.2 kafka

> cd kafka

> mkdir logs

> cd config

> cat server.properties

> mkdir data

Step 2: Start the server

Kafka uses ZooKeeper so you need to first start a ZooKeeper server if you don't already have one. You can use the convenience script packaged with kafka to get a quick-and-dirty single-node ZooKeeper instance.

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

Now start the Kafka server:

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

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

Step 3: Create a topic

Let's create a topic named "test" with a single partition and only one replica:

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

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

新版本用--bootstrap

We can now see that topic if we run the list topic command:

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

Step 4: Send some messages

Kafka comes with a command line client that will take input from a file or from standard input and send it out as messages to the Kafka cluster. By default, each line will be sent as a separate message.

Run the producer and then type a few messages into the console to send to the server.

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

This is a message

This is another message

Step 5: Start a consumer

Kafka also has a command line consumer that will dump out messages to standard output.

> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning

This is a message

This is another message

Step...

更多步骤详情参考

http://kafka.apache.org/0110/documentation/#quickstart

```

编写集群启动脚本

```

#!/bin/bash

case $i in

"start"){

    for i in hadoop1 hadoop2 hadoop3

    do

        echo "\*\*\*\*\*\*\*\*\*$i\*\*\*\*\*\*\*\*\*"

        ssh $i "/opt/tools/kafka/bin/kafka-server-start.sh -daemon /opt/tools/kafka/config/server.properties"

    done

};;

"stop"){

    for i in hadoop1 hadoop2 hadoop3

    do

        echo "\*\*\*\*\*\*\*\*\*$i\*\*\*\*\*\*\*\*\*"

        ssh $i "/opt/tools/kafka/bin/kafka-server-stop.sh"

    done

};;

esac

> cd /bin

> touch kk.sh

> chmod 777 kk.sh

> kk.sh start

> kk.sh stop

> xcall.sh jps

```

windows下安装kafka

```

Binary Downloads

Scala 2.12  - kafka\_2.12-2.8.0.tgz (asc, sha512)

下载kafka 0.11.0.0版本，解压到C:\Kafka\路径下，Kafka主目录文件为C:\Kafka\kafka\_2.11-0.11.0.0（下文用KAFKA\_HOME表示）

Kafka 用到 ZooKeeper 功能，所以要预先运行ZooKeeper。

首先，修改%KAFKA\_HOME%\conf\zookeeper.properties中的dataDir=/tmp/zookeeper，改为dataDir=C:\\Kafka\\data\\zookeeper。

创建新目录C:\\Kafka\\data\\zookeeper。

启动cmd，工作目录切换到%KAFKA\_HOME%，执行命令行：

start bin\windows\zookeeper-server-start.bat config\zookeeper.properties

修改%KAFKA\_HOME%\conf\server.properties中的log.dirs=/tmp/kafka-logs，改为log.dirs=C:\\Kafka\\data\\kafka-logs。

创建新目录C:\\Kafka\\data\\kafka-logs。

另启动cmd，工作目录切换到%KAFKA\_HOME%，执行命令行：

start bin\windows\kafka-server-start.bat config\server.properties

关闭服务，bin\windows\kafka-server-stop.bat和bin\windows\zookeeper-server-stop.bat。

一个问题：

通过kafka-server-stop.bat或右上角关闭按钮来关闭Kafka服务后，马上下次再启动Kafka，抛出异常，说某文件被占用，需清空log.dirs目录下文件，才能重启Kafka。

[2020-07-21 21:43:26,755] ERROR There was an error in one of the threads during logs loading: java.nio.file.FileSystemException: C:\Kafka\data\kafka-logs-0\my-replicated-topic-0\00000000000000000000.timeindex: 另一个程序正在使用此文件，进程无法访问。

 (kafka.log.LogManager)

成功启动kafka，然后在对应的命令行窗口用Ctrl + C结束Kakfa，下次不用清理kafka日志，也能正常启动。

用单一partition和单一replica创建一个名为test的topic:

bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test

查看已创建的topic：

bin\windows\kafka-topics.bat --list --zookeeper localhost:2181

或者配置broker去自动创建未曾发布过的topic，代替手动创建topic。

运行producer，然后输入几行文本，发至服务器：

bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic test

>hello, kafka.

>what a nice day!

>to be or not to be. that' s a question.

运行consumer，将输入的几行句子，标准输出：

bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --from-beginning

hello, kafka.

what a nice day!

to be or not to be. that' s a question

另启cmd，执行命令行：

bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --from-beginning来运行consumer

然后在producer窗口输入一行句子，如I must admit, I can't help but feel a twinge of envy.，两个consumer也会同时输出I must admit, I can't help but feel a twinge of envy.

Setting up a multi-broker cluster

创建有三个节点的集群

首先，在%KAFKA%\config\server.properties的基础上创建两个副本server-1.properties和server-2.properties。

copy config\server.properties config\server-1.properties

copy config\server.properties config\server-2.properties

打开副本，编辑如下属性

#config/server-1.properties:

broker.id=1

listeners=PLAINTEXT://127.0.0.1:9093

log.dir=C:\\Kafka\\data\\kafka-logs-1

#config/server-2.properties:

broker.id=2

listeners=PLAINTEXT://127.0.0.1:9094

log.dir=C:\\Kafka\\data\\kafka-logs-2

这个broker.id属性是集群中每个节点的唯一永久的名称。

必须重写端口和日志目录，因为在同一台机器上运行它们，并且希望阻止brokers试图在同一个端口上注册或覆盖彼此的数据。

已经启动了Zookeeper和单个节点，所以只需要启动两个新节点：

start bin\windows\kafka-server-start.bat config\server-1.properties

start bin\windows\kafka-server-start.bat config\server-2.properties

创建一个replication-factor为3的topic：

bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 3 --partitions 1 --topic my-replicated-topic

怎么知道哪个broker在做什么呢？

运行describe topics命令：

bin\windows\kafka-topics.bat --describe --zookeeper localhost:2181 --topic my-replicated-topic

Topic:my-replicated-topic       PartitionCount:1        ReplicationFactor:3

Configs:

        Topic: my-replicated-topic      Partition: 0    Leader: 0  Replicas: 0,1,2        Isr: 0,1,2

输出：

第一行给出所有Partition的摘要，每一行提供有关一个Partition的信息。

因为这个Topic只有一个Partition，所以只有一行。

"leader"是负责给定Partition的所有读写的节点。每个节点都可能成为Partition随机选择的leader。

"replicas"是复制此Partition日志的节点列表，无论它们是leader还是当前处于存活状态。

"isr"是一组"in-sync" replicas。这是replicas列表的一个子集，它当前处于存活状态，并补充leader。

在示例中，node 0是Topic唯一Partition的leader。

给新Topic发布一些信息：

bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic my-replicated-topic

>There's sadness in your eyes, I don't want to say goodbye to you.

>Love is a big illusion, I should try to forget, but there's something left in m

y head.

>

接收发布的信息：

bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --from-beginning --topic my-replicated-topic

There's sadness in your eyes, I don't want to say goodbye to you.

Love is a big illusion, I should try to forget, but there's something left in my head.

测试容错性：

由上文可知，Broker 0 身为leader，干掉Broker 0对应的进程：

找出Broker 0的进程pid。

杀掉Broker 0的进程。

> wmic process where "caption='java.exe' and commandline like '%server.properties%'" get processid,caption

Caption   ProcessId

java.exe  7528

> taskkill /pid 7528 /f

成功: 已终止 PID 为 7528 的进程。

原leader已被替换成它的flowers中的其中一个，并且 node 0 不在 in-sync replica 集合当中：

> bin\windows\kafka-topics.bat --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: 0,1,2 Isr: 1,2

尽管原先的leader挂掉，但是原来的消息依然可以接收（注意，参数--bootstrap-server localhost:9093，而不是--bootstrap-server localhost:9092）：

> bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9093 --from-beginning --topic my-replicated-topic

There's sadness in your eyes, I don't want to say goodbye to you.

Love is a big illusion, I should try to forget, but there's something left in my head.

I don't forget the way your kissing, the feeling 's so strong which is lasting for so long.

```

server.properties常见配置

```

#broker 的全局唯一编号，不能重复

broker.id=0

#删除 topic 功能使能

delete.topic.enable=true

#处理网络请求的线程数量

num.network.threads=3

#用来处理磁盘 IO 的现成数量

num.io.threads=8

#发送套接字的缓冲区大小

socket.send.buffer.bytes=102400

#接收套接字的缓冲区大小

socket.receive.buffer.bytes=102400

#请求套接字的缓冲区大小

socket.request.max.bytes=104857600

#kafka 运行日志存放的路径（实际就是kafka暂存数据的地方）

log.dirs=/opt/module/kafka/logs

#topic 在当前 broker 上的分区个数

num.partitions=1

#用来恢复和清理 data 下数据的线程数量

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

#segment 文件保留的最长时间，超时将被删除

log.retention.hours=168

#配置连接 Zookeeper 集群地址

zookeeper.connect=hadoop102:2181,hadoop103:2181,hadoop104:2181

```

注意：

1. 删除topic，需要在server.properties中设置delete.topic.enable=true，否则只是标记删除。

2. --from-beginning，会把topic以往所有的数据读取出来。

## 2.2 命令行操作

1. 查看当前服务器中的所有 topic

> bin\windows\kafka-topics.bat --list --zookeeper localhost:2181

2. 创建 topic

> bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 3 --partitions 1 --topic my-replicated-topic

选项说明：

--topic 定义topic名称

--replication-factor 定义副本数

--partitions 定义分区数

为了实现扩展性，一个非常大的topic可以分布到多个broker（即服务器）上，一个topic可以分为多个partition，每个partition是一个有序的队列。

a broker = a kafka server a broker can contain N topic a topic can contain N partition a broker can contain a part of a topic (a broker can contain M(N>M) partition)

3. 删除 topic

> bin\windows\kafka-topics.bat --zookeeper localhost:2181 --delete --topic my-replicated-topic

需要 server.properties 中设置 delete.topic.enable=true 否则只是标记删除。

4. 查看某个Topic的详情

> bin\windows\kafka-topics.bat --zookeeper localhost:2181 --describe --topic first

5. 修改分区数

> bin\windows\kafka-topics.bat --zookeeper localhost:2181 --alter --topic first --partitions 6

6. 发送消息

> bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic test

>hello, kafka.

>what a nice day!

>to be or not to be. that' s a question.

7. 消费消息

> bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --from-beginning

hello, kafka.

what a nice day!

to be or not to be. that' s a question.

--from-beginning：

会把主题中以往所有的数据都读取出来。

## 2.3 数据日志分离

kafka默认会把运行的日志存到/opt/tools/kafka/logs目录里

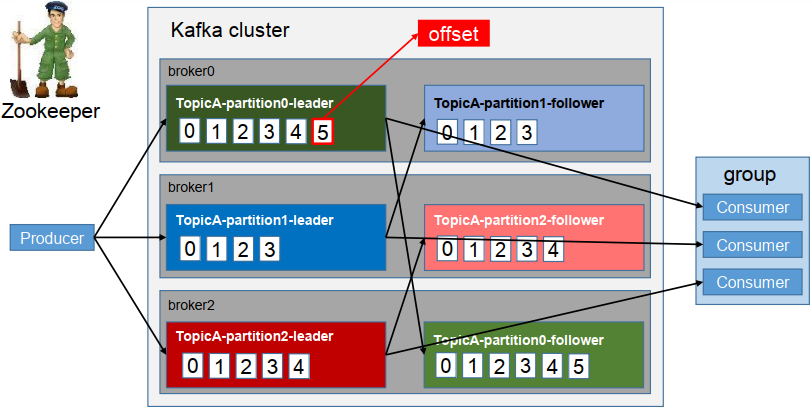
server.properties里的logs.dir指的是数据暂存的目录,和logs目录是不一样的。

一般配置为:

logs.dir=/opt/tools/kafka/data

# 3. 架构深入

## 3.1 工作流程和文件存储机制



Kafka的消息都是以topic进行分类的。producer生产消息，consumer消费消息，都是面向topic的。(从命令行操作看出)

> bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic test

> bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --from-beginning

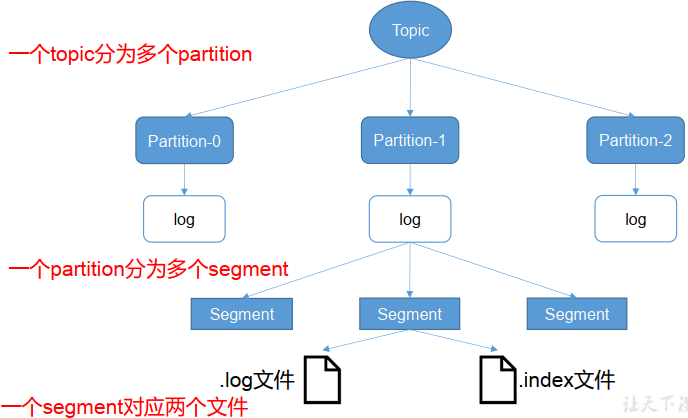
Topic是逻辑上的概念，而partition是物理上的概念。

每个partition对应于一个log文件，该log文件中存储的就是producer生产的数据。（topic = N partition，partition = log）

Producer生产的数据会被不断追加到该log文件末端，且每条数据都有自己的offset。

consumer组中的每个consumer，都会实时记录自己消费到了哪个offset，以便出错恢复时，从上次的位置继续消费。（producer -> log with offset -> consumer(s)）

Kafka文件存储机制



由于生产者生产的消息会不断追加到log文件末尾，为防止log文件过大导致数据定位效率低下，Kafka采取了分片（默认1g）和索引机制，将每个 partition分为多个segment。

每个segment对应两个文件，“.index”文件和“.log”文件。

这些文件位于一个文件夹下，该文件夹的命名规则为： topic名称+分区序号。

例如，first这个topic 有三个分区，则其对应的文件夹为first-0,first-1,first-2。

00000000000000000000.index

00000000000000000000.log

00000000000000170410.index

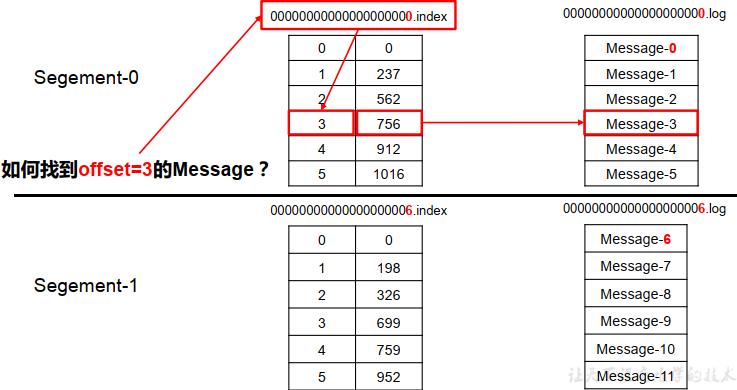
00000000000000170410.log

00000000000000239430.index

00000000000000239430.log

index和log文件以当前segment的第一条消息的offset命名。

下图为index文件和 log文件的结构示意图。



“.index”文件存储大量的索引信息，“.log”文件存储大量的数据。

索引文件中的元数据指向对应数据文件中message的物理偏移地址。

## 3.2 Kafka生产者

### 3.2.1 生产者分区策略

分区的原因：

1. 方便在集群中扩展。

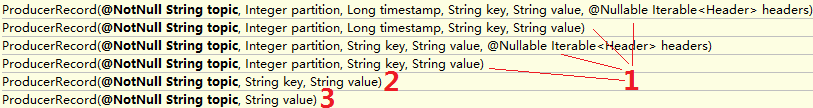
每个Partition可以通过调整以适应它所在的机器，而一个topic又可以由多个Partition组成，因此整个集群就可以适应任意大小的数据了。

2. 可以提高并发。

因为可以以Partition为单位读写了。（联想到ConcurrentHashMap在高并发环境下读写效率比HashTable的高效）

分区的原则：

需要将producer发送的数据封装成一个ProducerRecord对象。



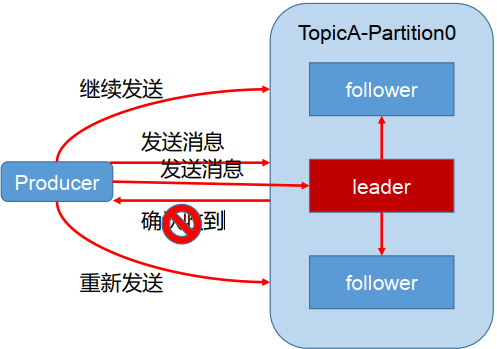
1. 指明partition的情况下，直接将指明的值直接作为partiton值。

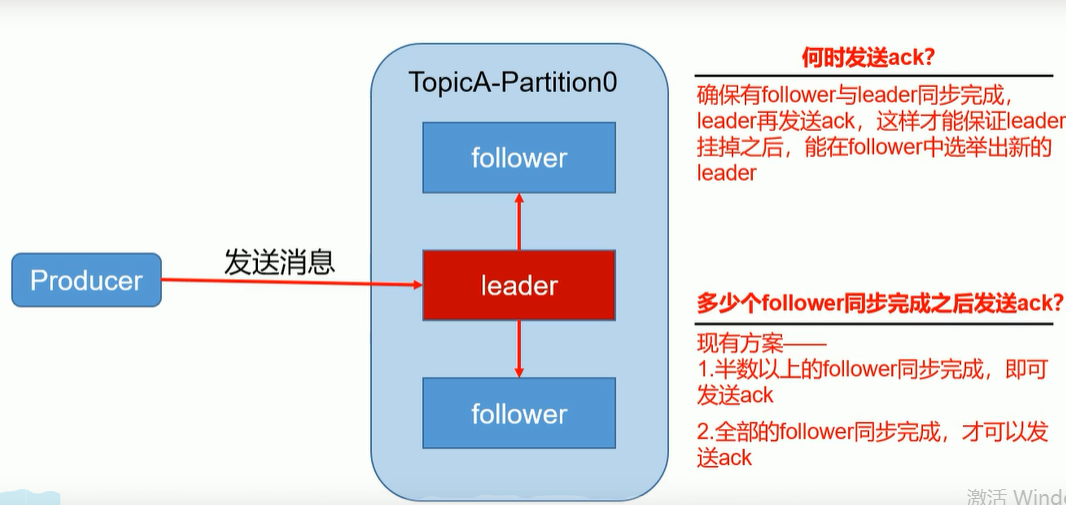
2. 没有指明partition值但有key的情况下，将key的hash值与topic的partition数进行取余得到partition值。

3. 既没有partition值又没有key值的情况下，第一次调用时随机生成一个整数（后面每次调用在这个整数上自增），将这个值与topic可用的partition总数取余得到partition值，也就是常说的round-robin算法。

### 3.2.2 数据可靠性保证

为保证producer发送的数据，能可靠的发送到指定的topic，topic的每个partition收到producer发送的数据后，都需要向producer发送ack（acknowledgement 确认收到），如果producer收到ack，就会进行下一轮的发送，否则重新发送数据。





何时发送ack？

确保有follower与leader同步完成，leader再发送ack，这样才能保证leader挂掉之后，能在follower中选举出新的leader。

多少个follower同步完成之后发送ack？

半数以上的follower同步完成，即可发送ack继续发送重新发送

全部的follower同步完成，才可以发送ack

#### 副本数据同步策略



Kafka 选择了第二种方案，原因如下：

1. 同样为了容忍n台节点的故障，第一种方案需要2n+1个副本，而第二种方案只需要n+1个副本，而Kafka的每个分区都有大量的数据，第一种方案会造成大量数据的冗余。

2. 虽然第二种方案的网络延迟会比较高，但网络延迟对Kafka的影响较小。

#### ISR

采用第二种方案之后，设想以下情景：

leader收到数据，所有follower都开始同步数据，但有一个follower，因为某种故障，迟迟不能与leader进行同步，那leader就要一直等下去，直到它完成同步，才能发送ack。

这个问题怎么解决呢？

Leader维护了一个动态的in-sync replica set (ISR)，意为和leader保持同步的follower集合。

当ISR中的follower完成数据的同步之后，就会给leader发送ack。

如果follower长时间未向leader同步数据，则该follower将被踢出ISR，该时间阈值由replica.lag.time.max.ms参数设定。

Leader发生故障之后，就会从ISR中选举新的leader。

```

replica.lag.time.max.ms

DESCRIPTION: If a follower hasn't sent any fetch requests or hasn't consumed up to the leaders log end offset for at least this time, the leader will remove the follower from isr

TYPE: long

DEFAULT: 10000

Source

http://kafka.apache.org/0110/documentation/#brokerconfigs

```

#### ack应答机制

对于某些不太重要的数据，对数据的可靠性要求不是很高，能够容忍数据的少量丢失，所以没必要等isr中的follower全部接收成功。

所以Kafka为用户提供了三种可靠性级别，用户根据对可靠性和延迟的要求进行权衡，选择以下的配置：

acks参数配置：

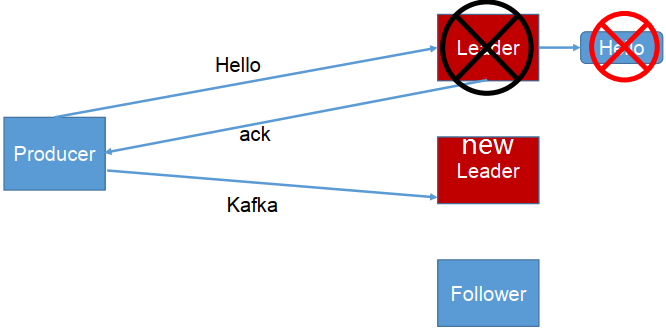
0：

producer不等待broker的ack，这一操作提供了一个最低的延迟，broker一接收到还没有写入磁盘就已经返回，当broker故障时有可能丢失数据。

1：

producer等待broker的ack，partition的leader落盘成功后返回ack，如果在follower同步成功之前leader故障，那么将会丢失数据。

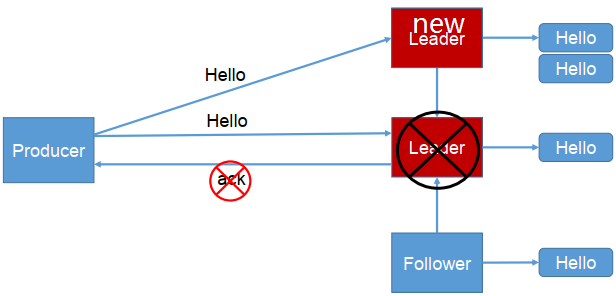
ack=1时的数据丢失案例



-1（all）：

producer等待broker的ack，partition的leader和ISR的follower全部落盘成功后才返回ack。但是如果在follower同步完成后，broker发送ack之前，leader发生故障，那么会造成数据重复。

ack=-1时的数据重复案例



助记：

返ACK前，0无落盘，1落盘，-1全落盘，（落盘：消息存到本地）

```

acks

DESCRIPTION:

The number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent. The following settings are allowed:

acks=0 If set to zero then the producer will not wait for any acknowledgment from the server at all. The record will be immediately added to the socket buffer and considered sent. No guarantee can be made that the server has received the record in this case, and the retries configuration will not take effect (as the client won't generally know of any failures). The offset given back for each record will always be set to -1.

acks=1 This will mean the leader will write the record to its local log but will respond without awaiting full acknowledgement from all followers. In this case should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost.

acks=all This means the leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee. This is equivalent to the acks=-1 setting.

TYPE:string

DEFAULT:1

VALID VALUES:[all, -1, 0, 1]

Source

http://kafka.apache.org/0110/documentation/#producerconfigs

```

#### 故障细节处理

log文件中的HW和LEO



LEO：（Log End Offset）

每个副本的最后一个offset。

HW：（High Watermark）

高水位，指的是消费者能见到的最大的offset，ISR队列中最小的LEO

#### follower故障和leader故障

follower故障：

follower发生故障后会被临时踢出ISR，待该follower恢复后，follower会读取本地磁盘记录的上次的HW，并将log文件高于HW的部分截取掉，从HW开始向leader进行同步。

等该follower的LEO大于等于该Partition的HW，即follower追上leader之后，就可以重新加入ISR了。

leader故障：

leader发生故障之后，会从ISR中选出一个新的leader，之后为保证多个副本之间的数据一致性，其余的follower会先将各自的log文件高于HW的部分截掉，然后从新的leader同步数据。

注意：

这只能保证副本之间的数据一致性，并不能保证数据不丢失或者不重复。

### 3.2.3 Exactly Once语义

将服务器的ACK级别设置为-1（all），可以保证Producer到Server之间不会丢失数据，即At Least Once语义。

相对的，将服务器ACK级别设置为0，可以保证生产者每条消息只会被发送一次，即At Most Once语义。

At Least Once可以保证数据不丢失，但是不能保证数据不重复；相对的，At Most Once可以保证数据不重复，但是不能保证数据不丢失。

但是，对于一些非常重要的信息，比如说交易数据，下游数据消费者要求数据既不重复也不丢失，即Exactly Once语义。

```

At least once（至少一次）：

Messages are never lost but may be redelivered.

At most once（至多一次）：

Messages may be lost but are never redelivered.

Exactly once（精准一次）：

this is what people actually want, each message is delivered once and only once.

Source

http://kafka.apache.org/0110/documentation/#semantics

```

在0.11版本以前的Kafka，对此是无能为力的，只能保证数据不丢失，再在下游消费者对数据做全局去重。

对于多个下游应用的情况，每个都需要单独做全局去重，这就对性能造成了很大影响。

0.11版本的Kafka，引入了一项重大特性：幂等性。

所谓的幂等性就是指Producer不论向Server发送多少次重复数据，Server端都只会持久化一条。

幂等性结合At Least Once语义，就构成了Kafka的Exactly Once语义。即：

At Least Once + 幂等性 = Exactly Once

要启用幂等性，只需要将Producer的参数中enable.idempotence设置为true即可。

Kafka的幂等性实现其实就是将原来下游需要做的去重放在了数据上游。

开启幂等性的Producer在初始化的时候会被分配一个PID，发往同一Partition的消息会附带Sequence Number。

而Broker端会对<PID, Partition, SeqNumber>做缓存，当具有相同主键的消息提交时，Broker只会持久化一条。

但是PID重启就会变化，同时不同的Partition也具有不同主键，所以幂等性无法保证跨分区跨会话的Exactly Once。

```

enable.idempotence

DESCRIPTION:When set to 'true', the producer will ensure that exactly one copy of each message is written in the stream. If 'false', producer retries due to broker failures, etc., may write duplicates of the retried message in the stream. This is set to 'false' by default. Note that enabling idempotence requires max.in.flight.requests.per.connection to be set to 1 and retries cannot be zero. Additionally acks must be set to 'all'. If these values are left at their defaults, we will override the default to be suitable. If the values are set to something incompatible with the idempotent producer, a ConfigException will be thrown.

TYPE:boolean

DEFAULT:false

Source

http://kafka.apache.org/0110/documentation/#producerconfigs

```

## 3.3 消费者

### 3.3.1 消费方式

consumer采用pull（拉）模式从broker中读取数据。

push（推）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。

它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。

pull模式不足之处是，如果kafka没有数据，消费者可能会陷入循环中，一直返回空数据。

针对这一点，Kafka的消费者在消费数据时会传入一个时长参数timeout，如果当前没有数据可供消费，consumer会等待一段时间之后再返回，这段时长即为timeout。

```

Push vs pull

http://kafka.apache.org/0110/documentation/#design\_pull

```

### 3.3.2 分区分配策略

一个consumer group中有多个consumer，一个topic有多个partition，所以必然会涉及到partition的分配问题，即确定那个partition由哪个consumer来消费。

**一个消费组内的不同消费者，可以消费同一个主题，但是不能同时消费同一个主题的一个分区。**

**同一个消费者组中的消费者， 同一时刻只能有一个消费者消费。**

Kafka有两种分配策略：

1. round-robin循环

2. range

```

partition.assignment.strategy

Select between the "range" or "roundrobin" strategy for assigning分配 partitions to consumer streams.

The round-robin partition assignor lays out规划 all the available partitions and all the available consumer threads. It then proceeds to do接着做 a round-robin assignment from partition to consumer thread. If the subscriptions订阅 of all consumer instances are identical完全同样的, then the partitions will be uniformly 均匀地distributed. (i.e.也就是说, the partition ownership counts will be within a delta of exactly one across all consumer threads.) Round-robin assignment is permitted only if:

Every topic has the same number of streams within a consumer instance

The set of subscribed topics is identical for every consumer instance within the group.

Range partitioning works on a per-topic basis. For each topic, we lay out the available partitions in numeric order and the consumer threads in lexicographic词典式的 order. We then divide the number of partitions by the total number of consumer streams (threads) to determine the number of partitions to assign to each consumer. If it does not evenly divide, then the first few consumers will have one extra partition.

DEFAULT:range

Source

http://kafka.apache.org/0110/documentation/#oldconsumerconfigs

```

默认分配策略：Range

RoundRobin是面向消费组的。Range是面向Topic的。

当消费组的消费者个数发生变化时，会触发分配策略进行重新分配。

#### **分区分配策略-RoundRobin**

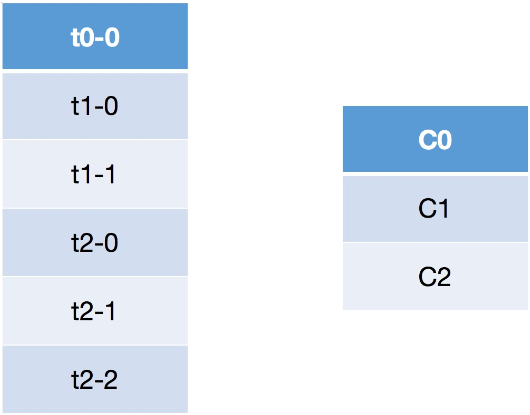
关于Roudn Robin重分配策略，其主要采用的是一种轮询的方式分配所有的分区，该策略主要实现的步骤如下。

这里我们首先假设有三个topic：t0、t1和t2，这三个topic拥有的分区数分别为1、2和3，那么总共有六个分区，这六个分区分别为：t0-0、t1-0、t1-1、t2-0、t2-1和t2-2。

这里假设我们有三个consumer：C0、C1和C2，它们订阅情况为：C0订阅t0，C1订阅t0和t1，C2订阅t0、t1和t2。

那么这些分区的分配步骤如下：

首先将所有的partition和consumer按照字典序进行排序，所谓的字典序，就是按照其名称的字符串顺序，那么上面的六个分区和三个consumer排序之后分别为：



然后依次以按顺序轮询的方式将这六个分区分配给三个consumer，如果当前consumer没有订阅当前分区所在的topic，则轮询的判断下一个consumer：

尝试将t0-0分配给C0，由于C0订阅了t0，因而可以分配成功；

尝试将t1-0分配给C1，由于C1订阅了t1，因而可以分配成功；

尝试将t1-1分配给C2，由于C2订阅了t1，因而可以分配成功；

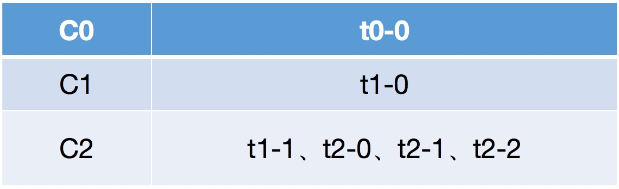
尝试将t2-0分配给C0，由于C0没有订阅t2，因而会轮询下一个consumer；

尝试将t2-0分配给C1，由于C1没有订阅t2，因而会轮询下一个consumer；

尝试将t2-0分配给C2，由于C2订阅了t2，因而可以分配成功；

同理由于t2-1和t2-2所在的topic都没有被C0和C1所订阅，因而都不会分配成功，最终都会分配给C2。

按照上述的步骤将所有的分区都分配完毕之后，最终分区的订阅情况如下：



从上面的步骤分析可以看出，轮询的策略就是简单的将所有的partition和consumer按照字典序进行排序之后，然后依次将partition分配给各个consumer，如果当前的consumer没有订阅当前的partition，那么就会轮询下一个consumer，直至最终将所有的分区都分配完毕。

但是从上面的分配结果可以看出，轮询的方式会导致每个consumer所承载的分区数量不一致，从而导致各个consumer压力不均一。

#### Range

所谓的Range重分配策略，就是首先会计算各个consumer将会承载的分区数量，然后将指定数量的分区分配给该consumer。

这里我们假设有两个consumer：C0和C1，两个topic：t0和t1，这两个topic分别都有三个分区，那么总共的分区有六个：t0-0、t0-1、t0-2、t1-0、t1-1和t1-2。

那么Range分配策略将会按照如下步骤进行分区的分配：

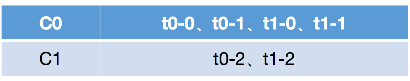
需要注意的是，Range策略是按照topic依次进行分配的，比如我们以t0进行讲解，其首先会获取t0的所有分区：t0-0、t0-1和t0-2，以及所有订阅了该topic的consumer：C0和C1，并且会将这些分区和consumer按照字典序进行排序；

然后按照平均分配的方式计算每个consumer会得到多少个分区，如果没有除尽，则会将多出来的分区依次计算到前面几个consumer。

比如这里是三个分区和两个consumer，那么每个consumer至少会得到1个分区，而3除以2后还余1，那么就会将多余的部分依次算到前面几个consumer，也就是这里的1会分配给第一个consumer，总结来说，那么C0将会从第0个分区开始，分配2个分区，而C1将会从第2个分区开始，分配1个分区；

同理，按照上面的步骤依次进行后面的topic的分配。

最终上面六个分区的分配情况如下：



可以看到，如果按照Range分区方式进行分配，其本质上是依次遍历每个topic，然后将这些topic的分区按照其所订阅的consumer数量进行平均的范围分配。

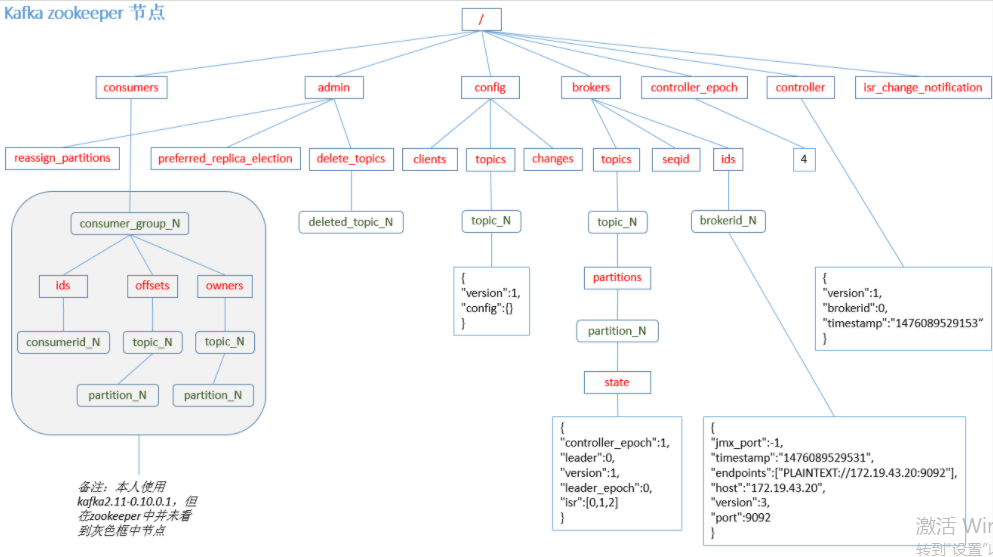
这种方式从计算原理上就会导致排序在前面的consumer分配到更多的分区，从而导致各个consumer的压力不均衡。

#### Kafka再平衡机制详解

https://zhuanlan.zhihu.com/p/86718818

### 3.3.3 消费者offset的维护

由于consumer在消费过程中可能会出现断电宕机等故障，consumer恢复后，需要从故障前的位置的继续消费，所以consumer需要实时记录自己消费到了哪个offset，以便故障恢复后继续消费。



Kafka 0.9版本之前，consumer默认将offset保存在Zookeeper中，从0.9版本开始，consumer默认将offset保存在Kafka一个内置topic中，该topic为\_\_consumer\_offsets。

```

修改配置文件consumer.properties：

exclude.internal.topics=false

读取offset：

0.11.0.0之前版本

> bin/kafka-console-consumer.sh --topic \_\_consumer\_offsets --zookeeper hadoop102:2181 --formatter "kafka.coordinator.GroupMetadataManager\$OffsetsMessageFormatter" --consumer.config config/consumer.properties --from-beginning

0.11.0.0 及之后版本

> bin/kafka-console-consumer.sh --topic \_\_consumer\_offsets --zookeeper hadoop102:2181 --formatter "kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter" --consumer.config config/consumer.properties --from-beginning

```

**offset在0.11.0版本之后是存储在\_\_consumer\_offsets内置topic中，是以键值对的方式来存。**

**其中，键是消费组+Topic名称+分区，值是offset的值。**

**offset是根据消费组+Topic名称+分区来唯一确定。**

### 3.3.4 消费者组案例

需求

测试同一个消费者组中的消费者， 同一时刻只能有一个消费者消费。

（不同消费者组可以同时消费同一个topic）

操作步骤

```

1.修改%KAFKA\_HOME\config\consumer.properties%文件中的group.id属性：

group.id=wh

（如果没有设置，默认起一个消费会话就创建一个组）

2.打开两个cmd，分别启动两个消费者。（以%KAFKA\_HOME\config\consumer.properties%作配置参数）

> bin\windows\kafka-console-consumer.bat --zookeeper 127.0.0.1:2181 --topic test --consumer.config config\consumer.properties

3.再打开一个cmd，启动一个生产者。

> bin\windows\kafka-console-producer.bat --broker-list 127.0.0.1:9092 --topic test

4.在生产者窗口输入消息，观察两个消费者窗口。会发现两个消费者窗口中，只有一个才会弹出消息。

```

## 3.4 高效读写数据

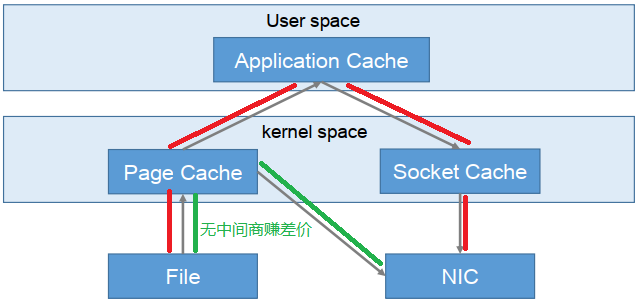
### 1. 顺序写磁盘

Kafka的producer生产数据，要写入到log文件中，写的过程是一直追加到文件末端，为顺序写。

官网有数据表明，同样的磁盘，顺序写能到600M/s，而随机写只有100K/s。

这与磁盘的机械机构有关，顺序写之所以快，是因为其省去了大量磁头寻址的时间。

### 2. 零复制技术



【零拷贝】

```

# cd /opt/tools/zookeeper

# bin/zkCli.sh

[zk: localhost:2181(CONNECTED) 0] ls /

```

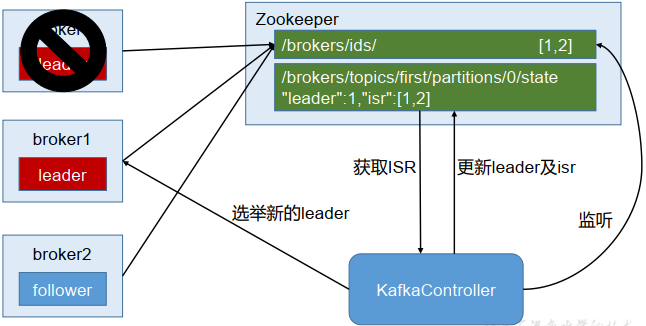
## 3.5 Zookeeper 在 Kafka 中的作用

Kafka集群中有一个broker会被选举为Controller，负责管理集群broker的上下线，所有topic的分区副本分配和leader选举等工作。

[http://kafka.apache.org/0110/documentation/#design\_replicamanagment]

Controller的管理工作都是依赖于Zookeeper的。

以下为partition的leader选举过程：



## 3.6 事务

Kafka从0.11版本开始引入了事务支持。

事务可以保证Kafka在Exactly Once语义的基础上，生产和消费可以跨分区和会话，要么全部成功，要么全部失败。

### Producer事务

为了实现跨分区跨会话的事务，需要引入一个全局唯一的Transaction ID，并将Producer获得的PID和Transaction ID绑定。

这样当Producer重启后就可以通过正在进行的TransactionID获得原来的PID。

为了管理Transaction，Kafka引入了一个新的组件Transaction Coordinator。

Producer就是通过和Transaction Coordinator交互获得Transaction ID对应的任务状态。

Transaction Coordinator还负责将事务所有写入Kafka的一个内部Topic，这样即使整个服务重启，由于事务状态得到保存，进行中的事务状态可以得到恢复，从而继续进行。

### Consumer事务

上述事务机制主要是从Producer方面考虑，对于Consumer而言，事务的保证就会相对较弱，尤其时无法保证Commit的信息被精确消费。

这是由于Consumer可以通过offset访问任意信息，而且不同的Segment File生命周期不同，同一事务的消息可能会出现重启后被删除的情况。

# 4. API

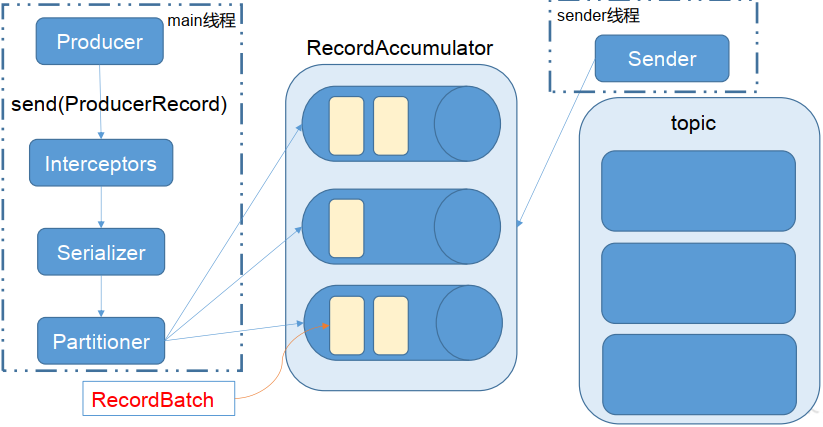
## 4.1 Producer API

### 4.1.1 信息发送流程

Kafka的Producer发送消息采用的是异步发送的方式。

在消息发送的过程中，涉及到了两个线程，main线程和Sender线程，以及一个线程共享变量RecordAccumulator。

main线程将消息发送给RecordAccumulator，Sender线程不断从RecordAccumulator中拉取消息发送到Kafka broker。



相关参数：

```

batch.size：

只有数据积累到batch.size之后，sender才会发送数据

linger.ms：

如果数据迟迟未达到batch.size，sender等待linger.time之后就会发送数据

```

### 4.1.2 异步发送API

```

导入依赖

pom.xml

<dependency>

    <groupId>org.apache.kafka</groupId>

    <artifactId>kafka-clients</artifactId>

    <version>0.11.0.0</version>

</dependency>

编写代码

需要用到的类：

KafkaProducer：需要创建一个生产者对象，用来发送数据

ProducerConfig：获取所需的一系列配置参数

ProducerRecord：每条数据都要封装成一个ProducerRecord对象

CustomProducer.java

import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.Producer;

import org.apache.kafka.clients.producer.ProducerRecord;

public class CustomProducer {

    public static void main(String[] args) {

        // 创建生成者的配置信息

        Properties props = new Properties();

        // kafka 集群， broker-list

        props.put("bootstrap.servers", "127.0.0.1:9092");

        //可用ProducerConfig.ACKS\_CONFIG 代替 "acks"

        //props.put(ProducerConfig.ACKS\_CONFIG, "all");

        props.put("acks", "all");

        // 重试次数

        props.put("retries", 1);

        // 批次大小

        props.put("batch.size", 16384); // 16k

        // 等待时间

        props.put("linger.ms", 1);

        // RecordAccumulator 缓冲区大小

        props.put("buffer.memory", 33554432); // 32m

        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new KafkaProducer<>(props);

        for (int i = 0; i < 100; i++) {

            producer.send(new ProducerRecord<String, String>("test", "test-" + Integer.toString(i),

                    "test-" + Integer.toString(i)));

        }

        producer.close();

    }

}

```

### 4.1.3 异步发送API带回调函数

回调函数会在producer收到ack时调用，为异步调用。

该方法有两个参数，分别是RecordMetadata和Exception，如果Exception为null，说明消息发送成功，如果Exception不为null，说明消息发送失败。

注意：消息发送失败会自动重试，不需要我们在回调函数中手动重试。

```

CallBackProducer.java

import java.util.Properties;

import org.apache.kafka.clients.producer.Callback;

import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.Producer;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.apache.kafka.clients.producer.RecordMetadata;

public class CallBackProducer {

    public static void main(String[] args) {

        Properties props = new Properties();

        props.put("bootstrap.servers", "127.0.0.1:9092");//kafka 集群， broker-list

        props.put("acks", "all");

        props.put("retries", 1);//重试次数

        props.put("batch.size", 16384);//批次大小

        props.put("linger.ms", 1);//等待时间

        props.put("buffer.memory", 33554432);//RecordAccumulator 缓冲区大小

        props.put("key.serializer",

        "org.apache.kafka.common.serialization.StringSerializer");

        props.put("value.serializer",

        "org.apache.kafka.common.serialization.StringSerializer");

        Producer<String, String> producer = new KafkaProducer<>(props);

        for (int i = 0; i < 100; i++) {

            producer.send(new ProducerRecord<String, String>("test",

                "test" + Integer.toString(i)), new Callback() {

                //回调函数， 该方法会在 Producer 收到 ack 时调用，为异步调用

                @Override

                public void onCompletion(RecordMetadata metadata, Exception exception) {

                    if (exception == null) {

                        System.out.println(metadata.partition() + " - " + metadata.offset());

                    } else {

                        exception.printStackTrace();

                    }

                }

            });

        }

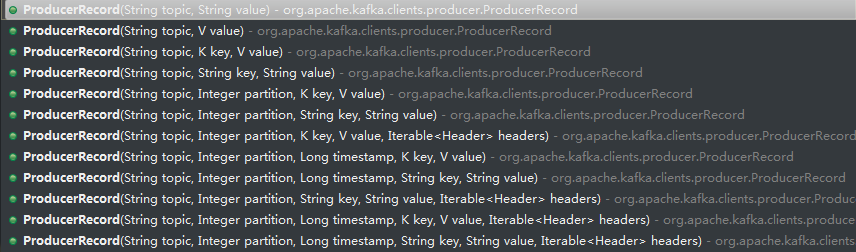
        producer.close();

    }

}

```

ProducerRecord类有许多构造函数，其中一个参数partition可指定分区



### 4.1.4 带自定义分区器的生成者

```

MyPartitioner.java

import org.apache.kafka.clients.producer.Partitioner;

import org.apache.kafka.common.Cluster;

public class MyPartitioner implements Partitioner {

    @Override

    public void configure(Map<String, ?> configs) {

        // TODO Auto-generated method stub

    }

    @Override

    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {

        // TODO Auto-generated method stub

        return 0;

    }

    @Override

    public void close() {

        // TODO Auto-generated method stub

    }

}

具体内容填写可参考默认分区器org.apache.kafka.clients.producer.internals.DefaultPartitioner

然后Producer配置中注册使用

Properties props = new Properties();

props.put(ProducerConfig.PARTITIONER\_CLASS\_CONFIG, MyPartitioner.class);

Producer<String, String> producer = new KafkaProducer<>(props);

```

### 4.1.5 同步发送API

同步发送的意思就是，一条消息发送之后，会阻塞当前线程，直至返回ack。

由于send方法返回的是一个Future对象，根据Futrue对象的特点，我们也可以实现同步发送的效果，只需在调用Future对象的get()方法即可。

```

SyncProducer.java

Producer<String, String> producer = new KafkaProducer<>(props);

for (int i = 0; i < 100; i++) {

    producer.send(new ProducerRecord<String, String>("test",  "test - 1"), new Callback() {

        @Override

        public void onCompletion(RecordMetadata metadata, Exception exception) {

            ...

        }

    }).get();//<----------------------

}

```

## 4.2 Consumer API

KafkaConsumer： 需要创建一个消费者对象，用来消费数据KafkaConsumer： 需要创建一个消费者对象，用来消费数据

ConsumerConfig： 获取所需的一系列配置参数

ConsuemrRecord： 每条数据都要封装成一个ConsumerRecord对象

### 4.2.1 自动提交offset

为了使我们能够专注于自己的业务逻辑， Kafka 提供了自动提交offset的功能。

```

自动提交 offset的相关参数：

enable.auto.commit：

是否开启自动提交offset功能

auto.commit.interval.ms：

自动提交offset的时间间隔

CustomConsumer.java

import java.util.Arrays;

import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerRecord;

import org.apache.kafka.clients.consumer.ConsumerRecords;

import org.apache.kafka.clients.consumer.KafkaConsumer;

public class CustomConsumer {

    public static void main(String[] args) {

        Properties props = new Properties();

        props.put("bootstrap.servers", "127.0.0.1:9092");

        props.put("group.id", "abc"); // 消费组

        props.put("enable.auto.commit", "true"); // 开启自动提交

        props.put("auto.commit.interval.ms", "1000"); // 自动提交时间间隔

        props.put("key.deserializer",

                "org.apache.kafka.common.serialization.StringDeserializer");

        props.put("value.deserializer",

                "org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        consumer.subscribe(Arrays.asList("test"));

        while (true) {

            ConsumerRecords<String, String> records = consumer.poll(100);

            for (ConsumerRecord<String, String> record : records) {

                System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

            }

        }

    }

}

```

消费者重置offset

Consumer消费数据时的可靠性是很容易保证的，因为数据在Kafka中是持久化的，故不用担心数据丢失问题。

由于consumer在消费过程中可能会出现断电宕机等故障，consumer恢复后，需要从故障前的位置的继续消费，所以consumer需要实时记录自己消费到了哪个offset，以便故障恢复后继续消费。

所以offset的维护是Consumer消费数据是必须考虑的问题。

```

public static final String AUTO\_OFFSET\_RESET\_CONFIG = "auto.offset.reset";

Properties props = new Properties();

props.put(ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG, "earliest");

props.put("group.id", "abcd"); // 组id需另设，否则看不出上面一句的配置效果

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

```

从结果看，props.put(ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG, "earliest");

与命令行中bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --from-beginning的--from-beginning拥有相同的作用。

```

--from-beginning

If the consumer does not already have an established offset to consume from, start with the earliest message present in the log rather than the latest message.

From kafka-console-sonsumer.bat

auto.offset.reset

What to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server (e.g. because that data has been deleted):

earliest: automatically reset the offset to the earliest offset

latest: automatically reset the offset to the latest offset

none: throw exception to the consumer if no previous offset is found for the consumer's group

anything else: throw exception to the consumer.

TYPE:string

DEFAULT:latest

VALID VALUES:[latest, earliest, none]

Source

http://kafka.apache.org/0110/documentation/#newconsumerconfigs

```

消费者保存offset读取问题

```

props.put("enable.auto.commit", "true");

enable.auto.commit

If true the consumer's offset will be periodically committed in the background.

TYPE:boolean

DEFAULT:true

Source

http://kafka.apache.org/0110/documentation/#newconsumerconfigs

```

### 4.2.2 手动提交offset

虽然自动提交offset十分便利，但由于其是基于时间提交的，开发人员难以把握offset提交的时机。

因此Kafka还提供了手动提交offset的API。

手动提交offset的方法有两种：

1. commitSync（同步提交）

2. commitAsync（异步提交）

两者的相同点是，都会将本次poll的一批数据最高的偏移量提交；

不同点是，commitSync阻塞当前线程，一直到提交成功，并且会自动失败重试（由不可控因素导致，也会出现提交失败）；

而commitAsync则没有失败重试机制，故有可能提交失败。

同步提交offset

由于同步提交 offset 有失败重试机制，故更加可靠，以下为同步提交offset的示例。

```

SyncCommitOffset.java

public class SyncCommitOffset {

    public static void main(String[] args) {

        Properties props = new Properties();

        ...

        //<-----------------

        //关闭自动提交 offset

        props.put("enable.auto.commit", "false");

        ...

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        consumer.subscribe(Arrays.asList("first"));//消费者订阅主题

        while (true) {

            //消费者拉取数据

            ConsumerRecords<String, String> records =

            consumer.poll(100);

            for (ConsumerRecord<String, String> record : records) {

                System.out.printf("offset = %d, key = %s, value= %s%n", record.offset(), record.key(), record.value());

            }

            //<---------------------------------------

            //同步提交，当前线程会阻塞直到 offset 提交成功

            consumer.commitSync();

        }

    }

}

```

异步提交offset

虽然同步提交offset更可靠一些，但是由于其会阻塞当前线程，直到提交成功。

因此吞吐量会收到很大的影响。因此更多的情况下，会选用异步提交offset的方式。

```

AsyncCommitOffset.java

public class AsyncCommitOffset {

    public static void main(String[] args) {

        Properties props = new Properties();

        ...

        //<--------------------------------------

        //关闭自动提交

        props.put("enable.auto.commit", "false");

        ...

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        consumer.subscribe(Arrays.asList("first"));// 消费者订阅主题

        while (true) {

            ConsumerRecords<String, String> records = consumer.poll(100);// 消费者拉取数据

            for (ConsumerRecord<String, String> record : records) {

                System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

            }

            //<----------------------------------------------

            // 异步提交

            consumer.commitAsync(new OffsetCommitCallback() {

                @Override

                public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception exception) {

                    if (exception != null) {

                        System.err.println("Commit failed for" + offsets);

                    }

                }

            });

        }

    }

}

```

数据漏消费和重复消费分析

无论是同步提交还是异步提交offset，都有可能会造成数据的漏消费或者重复消费。

先提交offset后消费，有可能造成数据的漏消费；而先消费后提交offset，有可能会造成数据的重复消费。

### 4.2.3 自定义存储offset

Kafka 0.9版本之前， offset存储在zookeeper，0.9版本及之后，默认将offset存储在Kafka的一个内置的topic中。

除此之外，Kafka还可以选择自定义存储offset。

offset的维护是相当繁琐的，因为需要考虑到消费者的Rebalace。

当有新的消费者加入消费者组、已有的消费者退出消费者组或者所订阅的主题的分区发生变化，就会触发到分区的重新分配，重新分配的过程叫做Rebalance。

消费者发生Rebalance之后，每个消费者消费的分区就会发生变化。

因此消费者要首先获取到自己被重新分配到的分区，并且定位到每个分区最近提交的offset位置继续消费。

要实现自定义存储offset，需要借助ConsumerRebalanceListener。

以下为示例代码，其中提交和获取offset的方法，需要根据所选的offset存储系统自行实现。(可将offset存入MySQL数据库)

```

CustomSaveOffset.java

public class CustomSaveOffset {

    private static Map<TopicPartition, Long> currentOffset = new HashMap<>();

    public static void main(String[] args) {

        // 创建配置信息

        Properties props = new Properties();

        ...

        //<--------------------------------------

        // 关闭自动提交 offset

        props.put("enable.auto.commit", "false");

        ...

        // 创建一个消费者

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        // 消费者订阅主题

        consumer.subscribe(Arrays.asList("first"),

            //<-------------------------------------

            new ConsumerRebalanceListener() {

            // 该方法会在 Rebalance 之前调用

            @Override

            public void onPartitionsRevoked(Collection<TopicPartition> partitions) {

                commitOffset(currentOffset);

            }

            // 该方法会在 Rebalance 之后调用

            @Override

            public void onPartitionsAssigned(Collection<TopicPartition> partitions) {

                currentOffset.clear();

                for (TopicPartition partition : partitions) {

                    consumer.seek(partition, getOffset(partition));// 定位到最近提交的 offset 位置继续消费

                }

            }

        });

        while (true) {

            ConsumerRecords<String, String> records = consumer.poll(100);// 消费者拉取数据

            for (ConsumerRecord<String, String> record : records) {

                System.out.printf("offset = %d, key = %s, value = %s%n", record.offset(), record.key(), record.value());

                currentOffset.put(new TopicPartition(record.topic(), record.partition()), record.offset());

            }

            commitOffset(currentOffset);// 异步提交

        }

    }

    // 获取某分区的最新 offset

    private static long getOffset(TopicPartition partition) {

        return 0;

    }

    // 提交该消费者所有分区的 offset

    private static void commitOffset(Map<TopicPartition, Long> currentOffset) {

    }

}

```

## 4.3 自定义Interceptor

### 4.3.1 拦截器原理

Producer拦截器(interceptor)是在Kafka 0.10版本被引入的，主要用于实现clients端的定制化控制逻辑。

对于producer而言，interceptor使得用户在消息发送前以及producer回调逻辑前有机会对消息做一些定制化需求，比如修改消息等。

同时，producer允许用户指定多个interceptor按序作用于同一条消息从而形成一个拦截链(interceptor chain)。

Intercetpor的实现接口是org.apache.kafka.clients.producer.ProducerInterceptor，其定义的方法包括：

configure(configs)：

获取配置信息和初始化数据时调用。

onSend(ProducerRecord)：

该方法封装进KafkaProducer.send方法中，即它运行在用户主线程中。

Producer确保在消息被序列化以及计算分区前调用该方法。

用户可以在该方法中对消息做任何操作，但最好保证不要修改消息所属的topic和分区，否则会影响目标分区的计算。

onAcknowledgement(RecordMetadata, Exception)：

该方法会在消息从RecordAccumulator成功发送到Kafka Broker之后，或者在发送过程中失败时调用。

并且通常都是在producer回调逻辑触发之前。

onAcknowledgement运行在producer的IO线程中，因此不要在该方法中放入很重的逻辑，否则会拖慢producer的消息发送效率。

close()：

关闭interceptor，主要用于执行一些资源清理工作。

如前所述，interceptor可能被运行在多个线程中，因此在具体实现时用户需要自行确保线程安全。

另外倘若指定了多个interceptor，则producer将按照指定顺序调用它们，并仅仅是捕获每个interceptor可能抛出的异常记录到错误日志中而非在向上传递。这在使用过程中要特别留意。

### 4.3.2 拦截器案例

需求

实现一个简单的双interceptor组成的拦截链。

第一个 interceptor 会在消息发送前将时间戳信息加到消息value的最前部

第二个 interceptor 会在消息发送后更新成功发送消息数或失败发送消息数

```

增加时间戳拦截器

TimeInterceptor.java

import java.util.Map;

import org.apache.kafka.clients.producer.ProducerInterceptor;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.apache.kafka.clients.producer.RecordMetadata;

public class TimeInterceptor implements ProducerInterceptor<String, String> {

    @Override

    public void configure(Map<String, ?> configs) {

    }

    @Override

    public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {

        // 创建一个新的 record，把时间戳写入消息体的最前部

        return new ProducerRecord(record.topic(), record.partition(), record.timestamp(), record.key(),

                "TimeInterceptor: " + System.currentTimeMillis() + "," + record.value().toString());

    }

    @Override

    public void close() {

    }

    @Override

    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {

        // TODO Auto-generated method stub

    }

}

```

增加时间戳拦截器

统计发送消息成功和发送失败消息数，并在 producer 关闭时打印这两个计数器

```

CounterInterceptor.java

import java.util.Map;

import org.apache.kafka.clients.producer.ProducerInterceptor;

import org.apache.kafka.clients.producer.ProducerRecord;

import org.apache.kafka.clients.producer.RecordMetadata;

public class CounterInterceptor implements ProducerInterceptor<String, String>{

    private int errorCounter = 0;

    private int successCounter = 0;

    @Override

    public void configure(Map<String, ?> configs) {

        // TODO Auto-generated method stub

    }

    @Override

    public ProducerRecord<String, String> onSend(ProducerRecord<String, String> record) {

        return record;

    }

    @Override

    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {

        // 统计成功和失败的次数

        if (exception == null) {

            successCounter++;

        } else {

            errorCounter++;

        }

    }

    @Override

    public void close() {

        // 保存结果

        System.out.println("Successful sent: " + successCounter);

        System.out.println("Failed sent: " + errorCounter);

    }

}

producer 主程序

InterceptorProducer.java

import java.util.ArrayList;

import java.util.List;

import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.Producer;

import org.apache.kafka.clients.producer.ProducerConfig;

import org.apache.kafka.clients.producer.ProducerRecord;

public class InterceptorProducer {

    public static void main(String[] args) {

        // 1 设置配置信息

        Properties props = new Properties();

        props.put("bootstrap.servers", "127.0.0.1:9092");

        props.put("acks", "all");

        props.put("retries", 3);

        props.put("batch.size", 16384);

        props.put("linger.ms", 1);

        props.put("buffer.memory", 33554432);

        props.put("key.serializer",

                "org.apache.kafka.common.serialization.StringSerializer");

        props.put("value.serializer",

                "org.apache.kafka.common.serialization.StringSerializer");

        //<--------------------------------------------

        // 2 构建拦截链

        List<String> interceptors = new ArrayList<>();

        interceptors.add("com.lun.kafka.interceptor.TimeInterceptor");

        interceptors.add("com.lun.kafka.interceptor.CounterInterceptor");

        props.put(ProducerConfig.INTERCEPTOR\_CLASSES\_CONFIG, interceptors);

        String topic = "test";

        Producer<String, String> producer = new KafkaProducer<>(props);

        // 3 发送消息

        for (int i = 0; i < 10; i++) {

            ProducerRecord<String, String> record = new ProducerRecord<>(topic, "message" + i);

            producer.send(record);

        }

        // 4 一定要关闭 producer，这样才会调用 interceptor 的 close 方法

        producer.close();

    }

}

```

# 5. 监控

## 5.1 Kafka Eagle

https://www.kafka-eagle.org/

https://www.kafka-eagle.org/articles/docs/documentation.html

什么是Kafka Eagle

Kafka Eagle is open source visualization and management software. It allows you to query, visualize, alert on, and explore your metrics no matter where they are stored. In plain English, it provides you with tools to turn your kafka cluster data into beautiful graphs and visualizations.

Kafka Eagle是开源可视化和管理软件。它允许您查询、可视化、提醒和探索您的指标，无论它们存储在哪里。简单地说，它为您提供了将kafka集群数据转换为漂亮的图形和可视化的工具。

一个运行在Tomcat的Web应用。

安装与运行

在Linux上安装与运行

在Windows上安装与运行

```

安装JDK，设置环境变量时，路径最好不要带空格，否则，后序运行ke.bat抛异常。若路径必须带有空格，可以通过小技巧，让ke.bat成功运行。这个技巧是：若你的JAVA\_HOME的变量值为C:\Program Files\Java\jdk1.8.0\_161，则将其改成C:\progra~1\Java\jdk1.8.0\_161。

到Kafka Eagle下载页面下载安装包。也可在网盘下载，链接在本文首部

解压安装包，然后设置环境变量KE\_HOME，其值如C:\Kafka\kafka-eagle-web-1.3.7。若想打开cmd输入命令ke.bat运行Kafka Eagle，在PATH环境变量的值头添加%KE\_HOME%\bin;

修改配置文件%KE\_HOME%\conf\system-config.properties

(可选)Kafka Server的JVM调参，用文本编辑器打开%KAFKA\_HOME%\bin\windows\kafka-server-start.bat，其中的set KAFKA\_HEAP\_OPTS=-Xmx1G -Xms1G改为set KAFKA\_HEAP\_OPTS=-server -Xms2G -Xmx2G -XX:PermSize=128m -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:ParallelGCThreads=8 -XX:ConcGCThreads=5 -XX:InitiatingHeapOccupancyPercent=70

######################################

# multi zookeeper&kafka cluster list

######################################<---设置ZooKeeper的IP地址

kafka.eagle.zk.cluster.alias=cluster1

cluster1.zk.list=localhost:2181

######################################

# zk client thread limit

######################################

kafka.zk.limit.size=25

######################################

# kafka eagle webui port

######################################

kafka.eagle.webui.port=8048

######################################

# kafka offset storage

######################################

cluster1.kafka.eagle.offset.storage=kafka

#cluster2.kafka.eagle.offset.storage=zk

######################################

# enable kafka metrics

######################################<---metrics.charts从false改成true

kafka.eagle.metrics.charts=true

kafka.eagle.sql.fix.error=false

######################################

# kafka sql topic records max

######################################

kafka.eagle.sql.topic.records.max=5000

######################################

# alarm email configure

######################################

kafka.eagle.mail.enable=false

kafka.eagle.mail.sa=alert\_sa@163.com

kafka.eagle.mail.username=alert\_sa@163.com

kafka.eagle.mail.password=mqslimczkdqabbbh

kafka.eagle.mail.server.host=smtp.163.com

kafka.eagle.mail.server.port=25

######################################

# alarm im configure

######################################

#kafka.eagle.im.dingding.enable=true

#kafka.eagle.im.dingding.url=https://oapi.dingtalk.com/robot/send?access\_token=

#kafka.eagle.im.wechat.enable=true

#kafka.eagle.im.wechat.token=https://qyapi.weixin.qq.com/cgi-bin/gettoken?corpid=xxx&corpsecret=xxx

#kafka.eagle.im.wechat.url=https://qyapi.weixin.qq.com/cgi-bin/message/send?access\_token=

#kafka.eagle.im.wechat.touser=

#kafka.eagle.im.wechat.toparty=

#kafka.eagle.im.wechat.totag=

#kafka.eagle.im.wechat.agentid=

######################################

# delete kafka topic token

######################################

kafka.eagle.topic.token=keadmin

######################################

# kafka sasl authenticate

######################################

cluster1.kafka.eagle.sasl.enable=false

cluster1.kafka.eagle.sasl.protocol=SASL\_PLAINTEXT

cluster1.kafka.eagle.sasl.mechanism=PLAIN

cluster1.kafka.eagle.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="admin" password="kafka-eagle";

#cluster2 在此没有用到，将其注释掉

#cluster2.kafka.eagle.sasl.enable=false

#cluster2.kafka.eagle.sasl.protocol=SASL\_PLAINTEXT

#cluster2.kafka.eagle.sasl.mechanism=PLAIN

#cluster2.kafka.eagle.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="admin" password="kafka-eagle";

######################################

# kafka jdbc driver address

######################################

kafka.eagle.driver=org.sqlite.JDBC

#将url设置在本地

kafka.eagle.url=jdbc:sqlite:/C:/Kafka/kafka-eagle-web-1.3.7/db/ke.db

#进入系统需要用到的账号与密码

kafka.eagle.username=root

kafka.eagle.password=123456

```

运行

运行ZooKeeper

运行Kafka集群，另外运行kafka server前，需设置JMX\_PORT，否则Kafka Eagle 后台提示连接失败。执行命令行set JMX\_PORT=9999 & start bin\windows\kafka-server-start.bat config\server.properties设置JMX\_PORT且运行Kafkaserver。在单节点开启Kafka集群，小心端口号冲突。

点击%KE\_HOME%\bin\ke.bat，运行Kafka Eagle。

打开浏览器，在地址栏输入http://localhost:8048/ke/，然后在登录页面，输入在配置文件%KE\_HOME%\conf\system-config.properties设置的账号与密码。

登录成功，便可进入Kafka Eagle

# 6. Flume对接kafka

Flume是Cloudera提供的一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统，Flume支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。

# 面试

## Kafka中的ISR(InSyncRepli)、OSR(OutSyncRepli)、AR(AllRepli)代表什么？

AR (All Replicas)：

一个分区的所有副本（AR = ISR + OSR）

ISR (In Sync Replicas)：

分区中与leader保持同步的follower集合

OSR (Out Sync Replicas)：

ISR是由leader维护，follower从leader同步数据有一些延迟（包括延迟时间replica.lag.time.max.ms和延迟条数replica.lag.max.messages两个维度, 当前最新的版本0.10.x中只支持replica.lag.time.max.ms这个维度，延迟时间默认10s），任意一个超过阈值都会把follower剔除出ISR，存入OSR（Outof-Sync Replicas）列表，新加入的follower也会先存放在OSR中。

OSR中的副本，如果与leader通信后，会尝试与leader同步，同步的策略是首先将当前记录的hw之后的消息删除，然后与leader同步，当与leader基本同步之后（存储的消息的offset大于当前isr中的hw），就重新回到isr之中。

ISR : 分区follower同步速率和leader相差低于10秒的follower的集合

OSR : 分区follower速率和leader相差大于10秒的follower

AR : 所有分区的follower（副本）

## Kafka中的HW、LEO等分别代表什么？

HW (High Watermark): 高水位，一个分区内所有副本的最小LEO，消费者能见到的最大的offset，ISR队列中最小的LEO

LEO (Log End Offset): 一个分区中每个副本最大的offset

## Kafka中是怎么体现消息顺序性的？

每个分区内，每条消息都有offset，只能保证同一分区内消息有序，不同的分区无法保证消息顺序性。

每个分区内，每条消息都有一个offset，故只能保证单个分区内有序。

kafka每个partition中的消息在写入时都是有序的，消费时每个partition只能被每一个group中的一个消费者消费，保证了消费时也是有序的。

整个topic不保证有序，如果为了保证topic整个有序，那么将partition调整为1。

## Kafka中的分区器、序列化器、拦截器是否了解？它们之间的处理顺序是什么？

分区器：

根据消息的键值确定消息应该处于哪个分区中，默认情况下使用轮询分区（Range），可以自行实现分区器接口自定义分区逻辑。

序列化器：

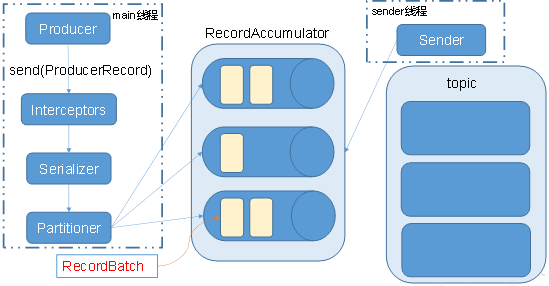
消息键序列化器和值序列化器，将键和值都转为二进制流还有反序列化器将二进制流转为指定类型数据。

拦截器：

实现拦截器接口。两个方法doSend()方法会在序列化之前完成，onAcknowledgement()方法在消息确认或失败时调用可以添加多个拦截器按顺序执行

处理顺序：拦截器 --> 序列化器 --> 分区器

## 5. Kafka生产者客户端的整体结构是什么样子的？使用了几个线程来处理？分别是什么？



使用两个线程:

main线程和sender线程。

main线程负责创建消息，然后依次经过拦截器，序列化器，分区器作用后消息缓存到累加器RecourdAccumlator(线程共享变量)。

再由sender线程从RecourdAccumlator中拉取数据发送到kafka broker。

相关参数：

batch.size：只有数据积累到batch.size之后，sender才会发送数据。

linger.ms：如果数据迟迟未达到batch.size，sender等待linger.time之后就会发送数据。

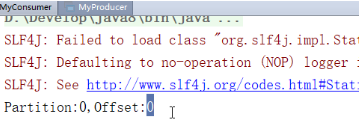
## 6. “消费组中的消费者个数如果超过topic的分区，那么就会有消费者消费不到数据”这句话是否正确？

对的，超过分区数的消费者就不会再接收数据

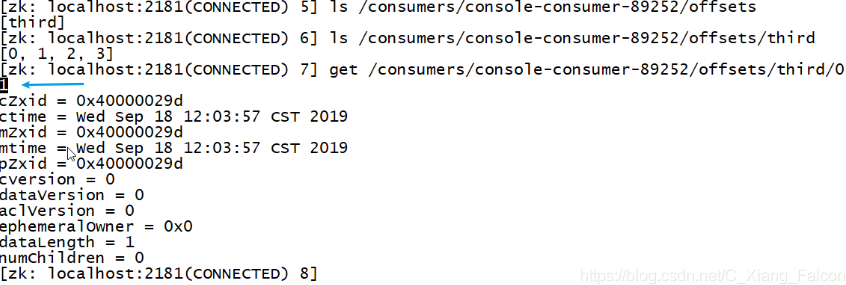
一个分区只能被同一个消费组中的一个消费者消费，如果消费者组中消费者个数超过分区数，那么肯定有一个没有办法消费到数据

## 7. 消费者提交消费位移时提交的是当前消费到的最新消息的offset还是offset+1？

offset + 1



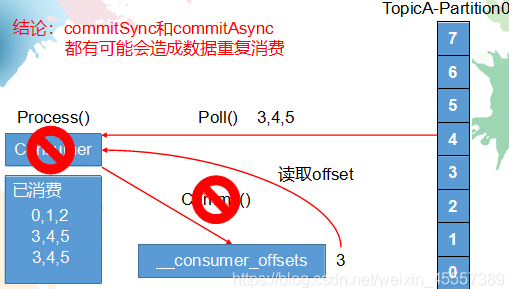
生产者发送数据offset是从0开始的



消费者消费的数据offset是从offset + 1开始的

## 有哪些情形会造成重复消费？

先处理后提交offset，会造成重复消费



1. 当ack=-1时，如果在follower同步完成后，broker发送ack之前，leader发生故障，导致没有返回ack给producer，由于失败重试机制，又会给新选举出来的leader发送数据，造成数据重复

2. 手动管理offset时，先消费后提交offset，消费者消费后没有commit offset（程序崩溃/强行kill/消费耗时/自动提交偏移情况下unscrible）

## 那些情景会造成消息漏消费？

先提交offset后处理，会造成数据丢失

1. 手动管理offset时，先提交offset后消费，有可能造成数据的漏消费

2. 当ack=0时，producer不等待broker的ack，这一操作提供了一个最低的延迟，broker一接收到还没有写入磁盘就已经返回，当broker故障时有可能丢失数据

3. 当ack=1时，producer等待broker的ack，partition的leader落盘成功后返回ack，如果在follower同步成功之前leader故障，而由于已经返回了ack，系统默认新选举的leader已经有了数据，从而不会进行失败重试，那么将会丢失数据

## 10. 当你使用kafka-topics.sh创建（删除）了一个topic之后，Kafka背后会执行什么逻辑？

1. 在zookeeper中的/brokers/topics节点下创建一个新的topic节点，如：/brokers/topics/first

2. 触发Kafka Controller的监听程序

3. kafka Controller负责topic的创建工作，并更新metadata cache

## 11. topic的分区数可不可以增加？如果可以怎么增加？如果不可以，那又是为什么？

可以增加

> bin/kafka-topics.sh --zookeeper localhost:2181/kafka --alter --topic topic-config --partitions 3

也可以通过Kafka Manager等图形化管理工具进行分区的添加

## topic的分区数可不可以减少？如果可以怎么减少？如果不可以，那又是为什么？

不可以，先有的分区数据难以处理，被删除的分区数据难以处理

## Kafka有内部的topic吗？如果有是什么？有什么所用？

有。\_\_consumer\_offsets，以双下划线开头，保存消费组的偏移量。

## Kafka分区分配的概念和目的？

对于kafka集群来说，分区可以做到负载均衡，对于消费者来说，可以提高并发度，提高消息读取效率

一个topic有多个分区，一个消费者组中有多个消费者，故需要将分区分配给不同的消费者（Range、RoundRobin）。

Range是默认策略。Range是对每个topic而言的（即一个topic一个topic的分），首先对同一个topic里面的分区按照序号进行排序，并对消费者按照字母顺序进行排序；然后用partitions分区的个数除以消费者线程的总数来决定每个消费者线程消费几个分区；如果除不尽，那么前面几个消费者线程将会多消费一个分区。

RoundRobin分区分配策略的前提是同一个消费者组里面的所有消费者的num.streams（消费者消费线程数）必须相等；每个消费者订阅的主题必须相同。

首先将所有主题分区组成TopicAndPartition列表，然后对TopicAndPartition列表按照hashCode进行排序，最后按照轮询的方式发给每一个消费线程。

## 简述Kafka的日志目录结构？

每一个分区对应一个目录，在logs.dir属性的目录里，目录名称为：topic名称-分区号，每个目录内有.index和.log文件，index和log文件以当前segment的第一条消息的offset命名。

每个分区对应一个文件夹，文件夹的命名为topic-0，topic-1，内部为.log和.index文件。

每个partition一个文件，包含四类文件：.index、.log、.timeindex、leader-epoch-checkpoint

.index .log .timeindex三个文件成对出现，前缀为上一个segment的最后一个消息的偏移。

log文件中保存了所有的消息。

index文件中保存了稀疏的相对偏移的索引。

timeindex保存的则是时间索引。

leader-epoch-checkpoint中保存了每一任leader开始写入消息时的offset，会定时更新，follower被选为leader时会根据这个确定哪些消息可用。

## 聊一聊Kafka Controller的作用？

负责管理集群broker的上下线，所有topic的分区副本分配和leader选举等工作。

在kafka集群中会有一个或者多个broker，其中有一个broker会被选举为控制器（kafka controller），它负责管理整个集群中所有分区和副本的状态。

当某个分区的leader副本出现故障时，由控制器负责为该分区选举新的leader副本。

当检测到某个分区的ISR集合发生变化时，由控制器负责通知所有broker更新其元数据信息。

当使用kafka-topic.sh脚本为某个topic增加分区数量时，同样还是由控制器负责分区的重新分配。

## Kafka中有那些地方需要选举？这些地方的选举策略又有哪些？

在ISR中需要选举，选举策略为先到先得

partition leader（ISR）

kafka controller（先到先得）

当broker启动时，会尝试去创建/controller节点，创建成功即成为controller；如果该controller死亡，/controller节点会释放，由新的broker创建此节点成为新的controller

## 如果我指定了一个offset， Kafka Controller怎么查找到对应的消息？

二分查找。

1. 先二分查找获取对应index索引文件，获取到对应的物理offset

2. 拿着物理offset去log数据文件顺序查找对应消息

3. 返回查找找到的消息

## 19. 失效副本是指什么？有那些应对措施？

失效副本为速率比leader相差大于10秒的follower。将失效的follower先踢出ISR，等速率接近leader10秒内，再加进ISR。

不能及时与leader同步，暂时踢出ISR，等其追上leader之后再重新加入。

ISR是由leader维护，follower从leader同步数据有一些延迟（包括延迟时间replica.lag.time.max.ms和延迟条数replica.lag.max.messages两个维度, 当前最新的版本0.10.x中移除了replica.lag.max.messages参数，防止服务频繁的进去队列），任意一个超过阈值都会把follower剔除出ISR，存入OSR（Outof-Sync Replicas）列表，新加入的follower也会先存放在OSR中。

OSR中的副本，如果与leader通信后，会尝试与leader同步，同步的策略是首先将当前记录的hw之后的消息删除，然后与leader同步，当与leader基本同步之后（存储的消息的offset大于当前ISR中的HW），就重新回到ISR之中。

## 20. Kafka的哪些设计让它有如此高的性能

1. kafka是分布式的消息队列

2. 对log文件进行了segment，并对segment建立了索引

3. (对于单节点)使用了顺序读写，速度可以达到600M/s

4. 引用了zero拷贝，在os系统就完成了读写操作

1. kafka本身式分布式集群，同时采用分区技术，并发度高

2. 顺序写磁盘

kafka的producer生产数据，要写入到log文件中，写的过程是一直追加到文件末端，为顺序写

官网有数据表明，同样的磁盘，顺序写能到600M/s，而随机写只有100K/s。

3. 零拷贝技术(0-copy)

## 21. Kafka的用途有哪些？使用场景如何？

1. 用户追踪：

根据用户在web或者app上的操作，将这些操作消息记录到各个topic中，然后消费者通过订阅这些消息做实时的分析，或者记录到HDFS，用于离线分析或数据挖掘

2. 日志收集：

通过kafka对各个服务的日志进行收集，再开放给各个consumer

3. 消息队列：

缓存消息

4. 运营指标：

记录运营监控数据，收集操作应用数据的集中反馈，如报错和报告

## 22. Kafka的高可靠性是怎么实现的？

为了实现高可靠性，kafka使用了订阅的模式，并使用ISR和ack应答机制。

能进入isr中的follower和leader之间的速率不会相差10秒

当ack=0时，producer不等待broker的ack，不管数据有没有写入成功，都不再重复发该数据

当ack=1时，broker会等到leader写完数据后，就会向producer发送ack，但不会等follower同步数据，如果这时leader挂掉，producer会对新的leader发送新的数据，在old的leader中不同步的数据就会丢失

当ack=-1或者all时，broker会等到leader和isr中的所有follower都同步完数据，再向producer发送ack，有可能造成数据重复

## 23. 如何解决消费者速率低的问题？

增加分区数和消费者数

## 24. kafka启动不起来的原因？

在关闭kafka时，先关了zookeeper，就会导致kafka下一次启动时，会报节点已存在的错误

只要把zookeeper中的zkdata/version-2的文件夹删除即可

## 25. Kafka消息是采用Pull模式，还是Push模式？

在producer阶段，是向broker用Push模式

在consumer阶段，是向broker用Pull模式

在Pull模式下，consumer可以根据自身速率选择如何拉取数据，避免了低速率的consumer发生崩溃的问题。但缺点是，consumer要时不时的去询问broker是否有新数据，容易发生死循环，内存溢出。

## 26. Kafka创建Topic时如何将分区放置到不同的Broker中？

首先副本数不能超过broker数。

第一分区是随机从Broker中选择一个，然后其他分区相对于0号分区依次向后移。

第一个分区是从nextReplicaShift决定的，而这个数也是随机产生的。

## 27. Kafka中的事务是怎么实现的？

kafka事务有两种：producer事务和consumer事务

producer事务是为了解决kafka跨分区跨会话问题。

kafka不能跨分区跨会话的主要问题是每次启动的producer的PID都是系统随机给的。

所以为了解决这个问题，我们就要手动给producer一个全局唯一的id，也就是transaction id 简称TID。

我们将TID和PID进行绑定，在producer带着TID和PID第一次向broker注册时，broker就会记录TID，并生成一个新的组件\_\_transaction\_state用来保存TID的事务状态信息。

当producer重启后，就会带着TID和新的PID向broker发起请求，当发现TID一致时，producer就会获取之前的PID，将覆盖掉新的PID，并获取上一次的事务状态信息，从而继续上次工作。

consumer事务相对于producer事务就弱一点，需要先确保consumer的消费和提交位置为一致且具有事务功能，才能保证数据的完整，不然会造成数据的丢失或重复。

## 28. kafka单条数据大小是多大？

kafka对于消息的大小默认单条最大值是1M，但是在我们应用场景中，常常会出现一条消息大于1M，如果不对kafka进行配置，则会出现生产者无法将消息推送到kafka或消费者无法去消费kafka里面的数据，这时我们就要对kafka进行以下配置：

```

server.properties

# broker可复制的消息的最大字节数, 默认为1M

replica.fetch.max.bytes: 1048576

# kafka会接收单个消息size的最大限制，默认为1M左右

message.max.bytes: 1000012

```

注意：

message.max.bytes必须小于等于replica.fetch.max.bytes，否则就会导致replica之间数据同步失败

## 29. kafka如何清理过期数据？

保证数据没有被引用（没人消费它）

日志清理保存的策略只有delete和compact两种

```

# 启用删除策略

log.cleanup.policy=delete

# 启用压缩策略

log.cleanup.policy=compact

```

## 30. Kafka可以按照时间消费数据？

Map<TopicPartition, OffsetAndTimestamp> startOffsetMap = KafkaUtil.fetchOffsetsWithTimestamp(topic, sTime, kafkaProp);

## 31. 消费者消费数据 --zookeeper和 --bootstrap-server的区别是什么？

对于消费者，kafka中有两个设置的地方：对于老的消费者，由 --zookeeper参数设置

对于新的消费者，由 --bootstrap-server参数设置

如果使用了 --zookeeper参数，那么consumer的offset信息将会存放在zookeeper之中，查看的方法是使用./zookeeper-client，然后ls /consumer/[group\_id]/offset/[topic]/[broker\_id-part\_id]，这个是查看某个group\_id的某个topic的offset

如果使用了 --bootstrap-server参数，那么consumer的信息将会存放在kafka之中

## 32. Kafka消费者角度考虑是拉取数据还是推送数据？

拉取数据

## 33. Kafka中的数据是有序的么？

只能保证单分区内有序

多分区之间、分区与分区间无序

## 34. Kafka的机器数量如何确定？

Kafka机器数量 = 2 \* (峰值生产速度 \* 副本数 / 100) + 1

## 35. 副本数如何设定？

一般我们设置成2个或3个，很多企业设置为2个

副本的优势：提高可靠性

副本的劣势：增加了网络IO传输

## 36. Kafka如何压测？

Kafka官方自带压力测试脚本（kafka-consumer-perf-test.sh、kafka-producer-perf-test.sh）

Kafka压测时，可以查看到哪个地方出现了瓶颈（CPU、内存、网络IO）

一般都是网络IO达到瓶颈

## 37. Kafka日志保存时间？

默认保存7天，生产环境建议3天

## 38. Kafka中数据量计算？

每天总数据量100g，每天产生1亿条日志，1亿/24/60/60=1150条/s

平均每秒钟：1150条

低谷每秒钟：50条

高峰每秒钟：1150 \* (2-20倍) = 2300-23000条

每条日志大小：0.5-2k（取1k）

每秒多少数据量：2-20M

## 39. Kafka的硬盘大小？

每天的数据量100g \* 2个副本 \* 3天 / 70%

## 40. Kafka监控？

公司自己开发的监控

开源的监控器：Kafka Manager、Kafka Monitor、Kafka Eagle

## 41. Kafka分区？

1. 创建一个只有1个分区的topic

2. 测试这个topic的producer吞吐量和consumer吞吐量

3. 假设它们的值分别是Tp和Tc，单位可以是MB/s

4. 然后假设总的目标吞吐量是Tt，那么分区数 = Tt/min（Tp，Tc）

假如：producer吞吐量=20m/s，consumer吞吐量=50m/s，期望吞吐量100m/s

分区数 = 100 / 20 = 5分区

分区数一般设置为：3-10个

## 42. 多少个topic合适？

通常情况：多少个日志类型就多少个topic，也有对日志类型进行合并的

## 43. Kafka挂掉会怎样？

1. Flume还有记录（Flume --> kafka）

2. 日志有记录

3. 短期没事

## 44. Kafka丢不丢数据？

1. ack=0，相当于异步发送，消息发送完毕即offset增加，继续生产

2. ack=1，leader收到leader replica对一个消息的接受ack才增加offset，然后继续生产

3. ack=-1，leader收到所有replica对一个消息的接受ack才增加offset，然后继续生产

## 45. Kafka数据重复？

幂等性 + ack=-1 + 事务

Kafka数据重复，可以再下一级：SparkStreaming、redis或者hive中dwd层去重

去重的手段：分组、按照id开窗只取第一个值

## 46. Kafka幂等性？

Kafka 0.11版本引入了幂等性，幂等性配合at least once语义可以实现exactly once语义。

但只能保证单次会话的幂等。

## 47. Kafka消息数据积压，Kafka消费能力不足怎么处理？

如果是kafka消费能力不足，则可以考虑增加topic的分区数，并且同时提升消费组的消费者数量，消费者数 = 分区数（两者缺一不可）

如果是下游的数据处理不及时：提升每批次拉取的数量；批次拉取数据过少（拉取数据 / 处理时间 < 生产速度），使处理的数据小于生产的数据，也会造成数据积压

## 48. Kafka事务

Kafka 0.11版本引入Kafka的事务机制，其可以保证生产者发往多个分区的一批数据的原子性

## 49. Kafka参数优化？

1. Broker参数配置（server.properties）

```

日志保留策略配置

# 保留三天，也可以更短（log.cleaner.delete.retention.ms）

log.retention.hours=72

Replica相关配置

# 默认副本1个

default.replication.factor:1

网络通信延时

# 当集群之间网络不稳定时，调大该参数

replica.socket.timeout.ms:30000

# 如果网络不好或者kafka集群压力较大，会出现副本丢失，然后会频繁复制副本，导致集群压力更大，此时可以调大该参数

replica.lag.time.max.ms= 600000

```

2. Producer优化（producer.properties）

```

# 默认发送不进行压缩，推荐配置一种适合的压缩算法，可以大幅度的减缓网络压力和Broker的存储压力

compression.type:none

```

GZIP、Snappy和LZ4，从2.1.0开始，kafka正式支持Zstandard算法（简写zstd），它是Facebook开源的一个压缩算法，能够提供超高的压缩比

对于kafka测试而言

在吞吐方面：LZ4 > Snappy > zstd、GZIP

在压缩比方面：zstd > LZ4 > GZIP > Snappy

具体到物理资源，使用Snappy算法占用的网络带宽资源最多，zstd最少，这是合理的，毕竟zstd就是要提供超高的压缩比

在CPU使用率方面，各个算法表现的差不多，只是在压缩时Snappy使用的CPU较多一些

而在解压缩时GZIP算法则可能使用更多的CPU

3. Kafka内存调整（kafka-server-start.sh）

```

# 默认内存1个G，生产环境尽量不要超过6个G

export KAFKA\_HEAP\_OPTS="-Xms4g -Xmx4g"

```

4. Kafka内存调整（kafka-server-start.sh）

```

# 默认内存1个G，生产环境尽量不要超过6个G

export KAFKA\_HEAP\_OPTS="-Xms4g -Xmx4g"

```