记得录屏

kafka 的版本差异

# 安装kafka

1. tar -zxvf .
2. 进入到config目录下修改server.properties

broker.id

listeners=PLAINTEXT://192.168.11.140:9092

zookeeper.connect

1. 启动

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

sh kafka-server-stop.sh

**zookeeper上注册的节点信息**

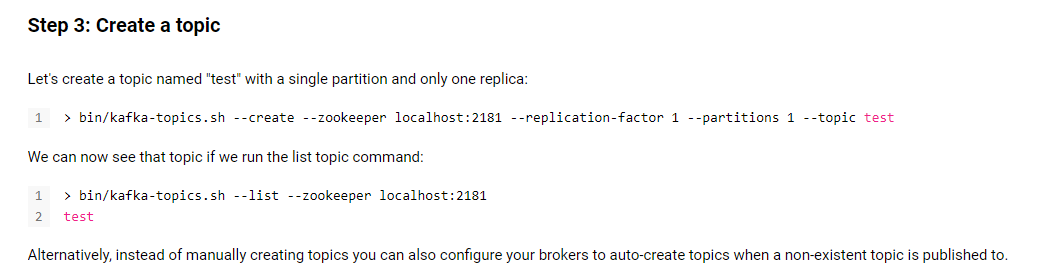
cluster, controller, controller\_epoch, brokers, zookeeper, admin, isr\_change\_notification, consumers, latest\_producer\_id\_block, config

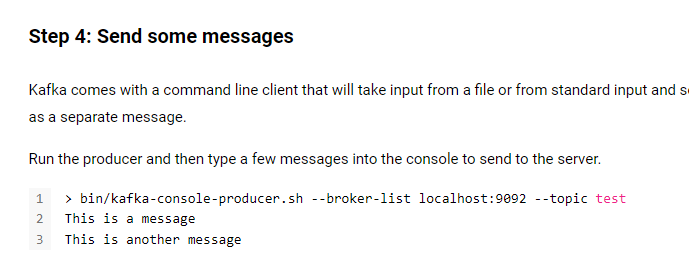
controller – 控制节点

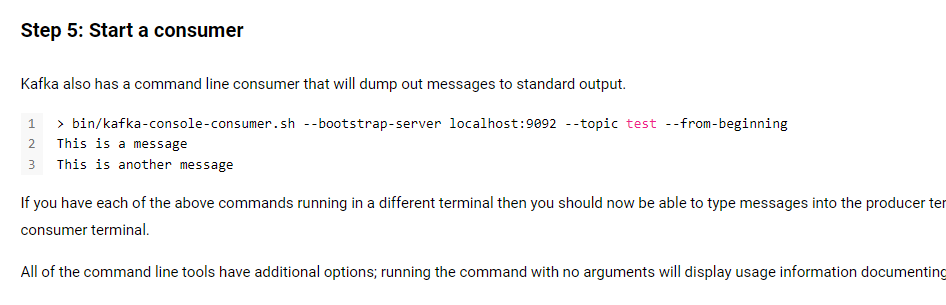
brokers – kafka集群的broker信息 。 topic

consumer ids/owners/offsets

# 基本操作







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

# kafka的实现细节

## 消息

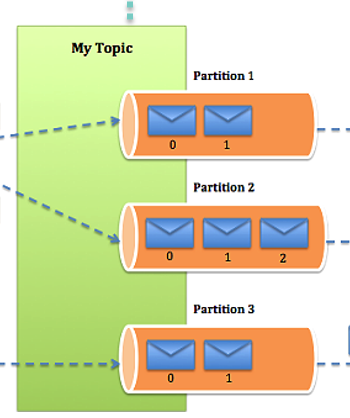
消息是kafka中最基本的数据单元。消息由一串字节构成，其中主要由key和value构成，key和value也都是byte数组。key的主要作用是根据一定的策略，将消息路由到指定的分区中，这样就可以保证包含同一key的消息全部写入到同一个分区中，key可以是null。为了提高网络的存储和利用率，生产者会批量发送消息到kafka，并在发送之前对消息进行压缩

## topic&partition

Topic是用于存储消息的逻辑概念，可以看作一个消息集合。每个topic可以有多个生产者向其推送消息，也可以有任意多个消费者消费其中的消息

每个topic可以划分多个分区（每个Topic至少有一个分区），同一topic下的不同分区包含的消息是不同的。每个消息在被添加到分区时，都会被分配一个offset（称之为偏移量），它是消息在此分区中的唯一编号，kafka通过offset保证消息在分区内的顺序，offset的顺序不跨分区，即kafka只保证在同一个分区内的消息是有序的；

图片包含 文字

已生成高可信度的说明  

Partition是以文件的形式存储在文件系统中，存储在kafka-log目录下，命名规则是：<topic\_name>-<partition\_id>

## kafka的高吞吐量的因素

1. 顺序写的方式存储数据 ；
2. 批量发送；在异步发送模式中。kafka允许进行批量发送，也就是先讲消息缓存到内存中，然后一次请求批量发送出去。这样减少了磁盘频繁io以及网络IO造成的性能瓶颈

batch.size 每批次发送的数据大小

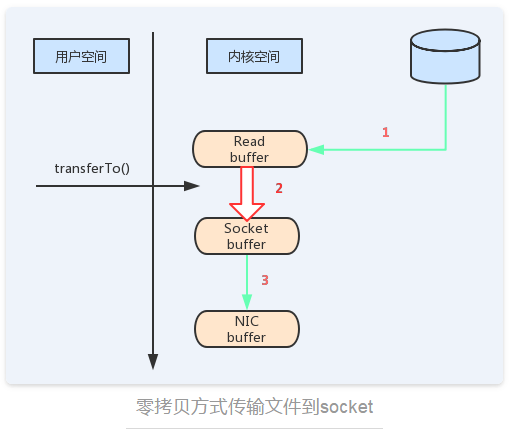
linger.ms 间隔时间

1. 零拷贝

消息从发送到落地保存，broker维护的消息日志本身就是文件目录，每个文件都是二进制保存，生产者和消费者使用相同的格式来处理。在消费者获取消息时，服务器先从硬盘读取数据到内存，然后把内存中的数据原封不懂的通过socket发送给消费者。虽然这个操作描述起来很简单，但实际上经历了很多步骤

|  |  |
| --- | --- |
| ▪ 操作系统将数据从磁盘读入到内核空间的页缓存  ▪ 应用程序将数据从内核空间读入到用户空间缓存中  ▪ 应用程序将数据写回到内核空间到socket缓存中  ▪ 操作系统将数据从socket缓冲区复制到网卡缓冲区，以便将数据经网络发出 | http://rdcqii.hundsun.com/portal/data/upload/201705/f_ea3eaa588e7f176aa71161e0fd08de9e.png |

通过“零拷贝”技术可以去掉这些没必要的数据复制操作，同时也会减少上下文切换次数



## 日志策略

### 日志保留策略

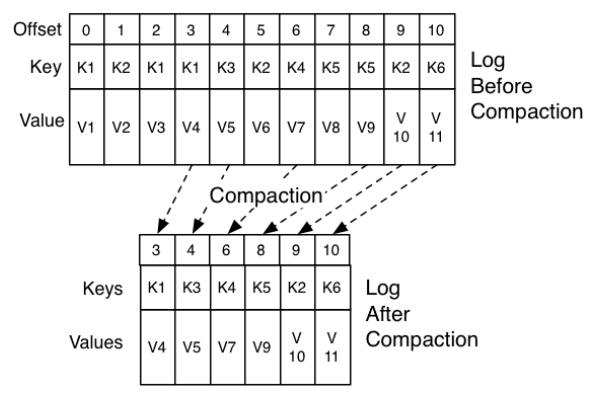
无论消费者是否已经消费了消息，kafka都会一直保存这些消息，但并不会像数据库那样长期保存。为了避免磁盘被占满，kafka会配置响应的保留策略（retention policy），以实现周期性地删除陈旧的消息

kafka有两种“保留策略”：

1. 根据消息保留的时间，当消息在kafka中保存的时间超过了指定时间，就可以被删除；
2. 根据topic存储的数据大小，当topic所占的日志文件大小大于一个阀值，则可以开始删除最旧的消息

### 日志压缩策略

在很多场景中，消息的key与value的值之间的对应关系是不断变化的，就像数据库中的数据会不断被修改一样，消费者只关心key对应的最新的value。我们可以开启日志压缩功能，kafka定期将相同key的消息进行合并，只保留最新的value值



# 消息可靠性机制

## 消息发送可靠性

生产者发送消息到broker，有三种确认方式（request.required.acks）

acks = 0: producer不会等待broker（leader）发送ack 。因为发送消息网络超时或broker crash(1.Partition的Leader还没有commit消息 2.Leader与Follower数据不同步)，既有可能丢失也可能会重发。

acks = 1: 当leader接收到消息之后发送ack，丢会重发，丢的概率很小

acks = -1: 当所有的follower都同步消息成功后发送ack. 丢失消息可能性比较低。

## 消息存储可靠性

每一条消息被发送到broker中，会根据partition规则选择被存储到哪一个partition。如果partition规则设置的合理，所有消息可以均匀分布到不同的partition里，这样就实现了水平扩展。

在创建topic时可以指定这个topic对应的partition的数量。在发送一条消息时，可以指定这条消息的key，producer根据这个key和partition机制来判断这个消息发送到哪个partition。

kafka的高可靠性的保障来自于另一个叫副本（replication）策略，通过设置副本的相关参数，可以使kafka在性能和可靠性之间做不同的切换。

**高可靠性的副本**

**sh kafka-topics.sh --create --zookeeper 192.168.11.140:2181 --replication-factor 2 --partitions 3 --topic sixsix**

**--replication-factor表示的副本数**

## 副本机制

ISR（副本同步队列）

维护的是有资格的follower节点

1. 副本的所有节点都必须要和zookeeper保持连接状态
2. 副本的最后一条消息的offset和leader副本的最后一条消息的offset之间的差值不能超过指定的阀值，这个阀值是可以设置的（replica.lag.max.messages）

## HW&LEO

关于follower副本同步的过程中，还有两个关键的概念，HW(HighWatermark)和LEO(Log End Offset). 这两个参数跟ISR集合紧密关联。HW标记了一个特殊的offset，当消费者处理消息的时候，只能拉去到HW之前的消息，HW之后的消息对消费者来说是不可见的。也就是说，取partition对应ISR中最小的LEO作为HW，consumer最多只能消费到HW所在的位置。每个replica都有HW，leader和follower各自维护更新自己的HW的状态。对于leader新写入的消息，consumer不能立刻消费，leader会等待该消息被所有ISR中的replicas同步更新HW，此时消息才能被consumer消费。这样就保证了如果leader副本损坏，该消息仍然可以从新选举的leader中获取

LEO 是所有副本都会有的一个offset标记，它指向追加到当前副本的最后一个消息的offset。当生产者向leader副本追加消息的时候，leader副本的LEO标记就会递增；当follower副本成功从leader副本拉去消息并更新到本地的时候，follower副本的LEO就会增加

# 查看kafka数据文件内容

在使用kafka的过程中有时候需要我们查看产生的消息的信息，这些都被记录在kafka的log文件中。由于log文件的特殊格式，需要通过kafka提供的工具来查看

./bin/kafka-run-class.sh kafka.tools.DumpLogSegments --files /tmp/kafka-logs/\*/000\*\*.log --print-data-log {查看消息内容}

# 高可用副本机制回顾

在kfaka0.8版本前，并没有提供这种High Availablity机制，也就是说一旦一个或者多个broker宕机，则在这期间内所有的partition都无法继续提供服务。如果broker无法再恢复，则上面的数据就会丢失。所以在0.8版本以后引入了High Availablity机制

## 关于leader election

在kafka引入replication机制以后，同一个partition会有多个Replica。那么在这些replication之间需要选出一个Leader，Producer或者Consumer只与这个Leader进行交互，其他的Replica作为Follower从leader中复制数据（因为需要保证一个Partition中的多个Replica之间的数据一致性，其中一个Replica宕机以后其他的Replica必须要能继续提供服务且不能造成数据重复和数据丢失）。 如果没有leader，所有的Replica都可以同时读写数据，那么就需要保证多个Replica之间互相同步数据，数据一致性和有序性就很难保证，同时也增加了Replication实现的复杂性和出错的概率。在引入leader以后，leader负责数据读写，follower只向leader顺序fetch数据，简单而且高效

## 如何将所有的Replica均匀分布到整个集群

为了更好的做到负载均衡，kafka尽量会把所有的partition均匀分配到整个集群上。如果所有的replica都在同一个broker上，那么一旦broker宕机所有的Replica都无法工作。kafka分配Replica的算法

1. 把所有的Broker（n）和待分配的Partition排序
2. 把第i个partition分配到 （i mod n）个broker上
3. 把第i个partition的第j个Replica分配到 ( (i+j) mod n) 个broker上

## 如何处理所有的Replica不工作的情况

在ISR中至少有一个follower时，Kafka可以确保已经commit的数据不丢失，但如果某个Partition的所有Replica都宕机了，就无法保证数据不丢失了

1. 等待ISR中的任一个Replica“活”过来，并且选它作为Leader
2. 选择第一个“活”过来的Replica（不一定是ISR中的）作为Leader

这就需要在可用性和一致性当中作出一个简单的折衷。

如果一定要等待ISR中的Replica“活”过来，那不可用的时间就可能会相对较长。而且如果ISR中的所有Replica都无法“活”过来了，或者数据都丢失了，这个Partition将永远不可用。

选择第一个“活”过来的Replica作为Leader，而这个Replica不是ISR中的Replica，那即使它并不保证已经包含了所有已commit的消息，它也会成为Leader而作为consumer的数据源（前文有说明，所有读写都由Leader完成）。

Kafka0.8.\*使用了第二种方式。Kafka支持用户通过配置选择这两种方式中的一种，从而根据不同的使用场景选择高可用性还是强一致性

# 文件存储机制

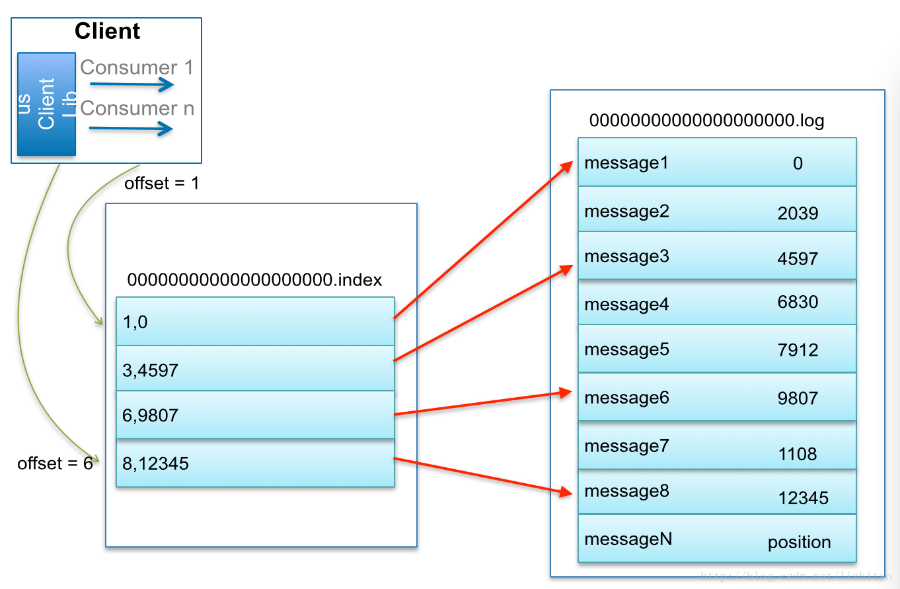
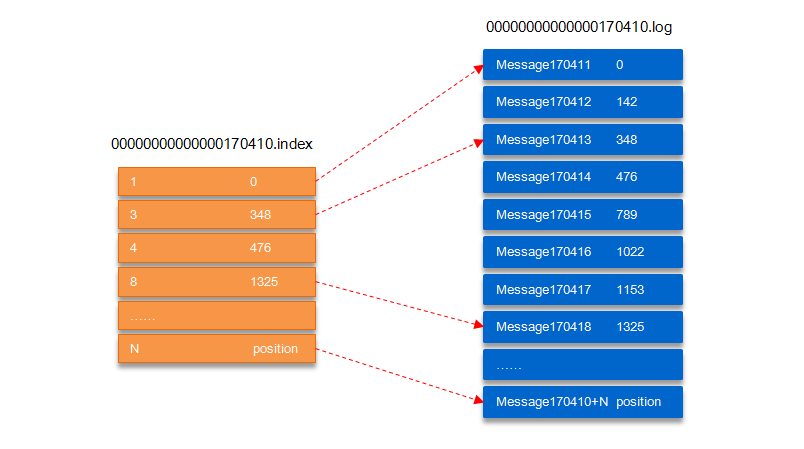
## 存储机制

在kafka文件存储中，同一个topic下有多个不同的partition，每个partition为一个目录，partition的名称规则为：topic名称+有序序号，第一个序号从0开始，最大的序号为partition数量减1，partition是实际物理上的概念，而topic是逻辑上的概念

partition还可以细分为segment，这个segment是什么呢？ 假设kafka以partition为最小存储单位，那么我们可以想象当kafka producer不断发送消息，必然会引起partition文件的无线扩张，这样对于消息文件的维护以及被消费的消息的清理带来非常大的挑战，所以kafka 以segment为单位又把partition进行细分。每个partition相当于一个巨型文件被平均分配到多个大小相等的segment数据文件中（每个setment文件中的消息不一定相等），这种特性方便已经被消费的消息的清理，提高磁盘的利用率

**segment file组成**：由2大部分组成，分别为index file和data file，此2个文件一一对应，成对出现，后缀".index"和“.log”分别表示为segment索引文件、数据文件.

**segment文件命名规则**：partion全局的第一个segment从0开始，后续每个segment文件名为上一个segment文件最后一条消息的offset值。数值最大为64位long大小，19位数字字符长度，没有数字用0填充

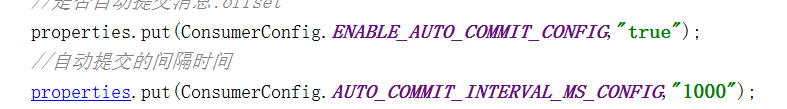


## 查找方式

以上图为例，读取offset=170418的消息，首先查找segment文件，其中00000000000000000000.index为最开始的文件，第二个文件为00000000000000170410.index（起始偏移为170410+1=170411），而第三个文件为00000000000000239430.index（起始偏移为239430+1=239431），所以这个offset=170418就落到了第二个文件之中。其他后续文件可以依次类推，以其实偏移量命名并排列这些文件，然后根据二分查找法就可以快速定位到具体文件位置。其次根据00000000000000170410.index文件中的[8,1325]定位到00000000000000170410.log文件中的1325的位置进行读取。

# 消息确认的几种方式

## 自动提交



## 手动提交

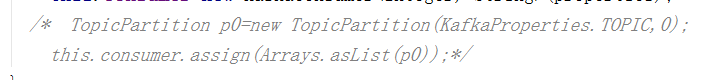
手动异步提交

consumer. commitASync() //手动异步ack

手动同步提交

consumer. commitSync() //手动异步ack

# 指定消费某个分区的消息



# 消息的消费原理

之前Kafka存在的一个非常大的性能隐患就是利用ZK来记录各个Consumer Group的消费进度（offset）。当然JVM Client帮我们自动做了这些事情，但是Consumer需要和ZK频繁交互，而利用ZK Client API对ZK频繁写入是一个低效的操作，并且从水平扩展性上来讲也存在问题。所以ZK抖一抖，集群吞吐量就跟着一起抖，严重的时候简直抖的停不下来。

新版Kafka已推荐将consumer的位移信息保存在Kafka内部的topic中，即\_\_consumer\_offsets topic。通过以下操作来看看\_\_consumer\_offsets\_topic是怎么存储消费进度的，\_\_consumer\_offsets\_topic默认有50个分区

1. 计算consumer group对应的hash值



1. 获得consumer group的位移信息

**bin/kafka-simple-consumer-shell.sh --topic \_\_consumer\_offsets --partition 15 -broker-list 192.168.11.140:9092,192.168.11.141:9092,192.168.11.138:9092 --formatter kafka.coordinator.group.GroupMetadataManager\$OffsetsMessageFormatter**

# kafka的分区分配策略

在kafka中每个topic一般都会有很多个partitions。为了提高消息的消费速度，我们可能会启动多个consumer去消费； 同时，kafka存在consumer group的概念，也就是group.id一样的consumer，这些consumer属于一个consumer group，组内的所有消费者协调在一起来消费消费订阅主题的所有分区。当然每一个分区只能由同一个消费组内的consumer来消费，那么同一个consumer group里面的consumer是怎么去分配该消费哪个分区里的数据，这个就设计到了kafka内部分区分配策略（Partition Assignment Strategy）

在 Kafka 内部存在两种默认的分区分配策略：**Range（默认）** 和 RoundRobin。通过：partition.assignment.strategy指定

## consumer rebalance

当以下事件发生时，Kafka 将会进行一次分区分配：

1. 同一个consumer group内新增了消费者
2. 消费者离开当前所属的consumer group，包括shuts down 或crashes
3. 订阅的主题新增分区（分区数量发生变化）
4. 消费者主动取消对某个topic的订阅
5. 也就是说，把分区的所有权从一个消费者移到另外一个消费者上，这个是kafka consumer 的rebalance机制。如何rebalance就涉及到前面说的分区分配策略。

## 两种分区策略

### Range 策略（默认）

0 ，1 ，2 ，3 ，4，5，6，7，8，9

c0 [0,3] c1 [4,6] c2 [7,9]

10(partition num/3(consumer num) =3

### roundrobin 策略

0 ，1 ，2 ，3 ，4，5，6，7，8，9

c0,c1,c2

c0 [0,3,6,9]

c1 [1,4,7]

c2 [2,5,8]

kafka 的key 为null， 是随机｛一个Metadata的同步周期内，默认是10分钟｝