# 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

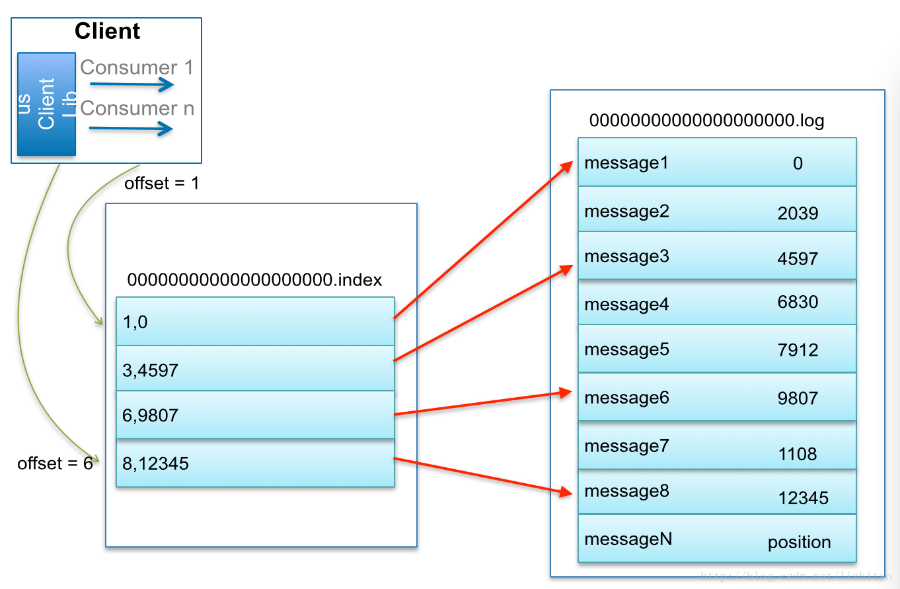
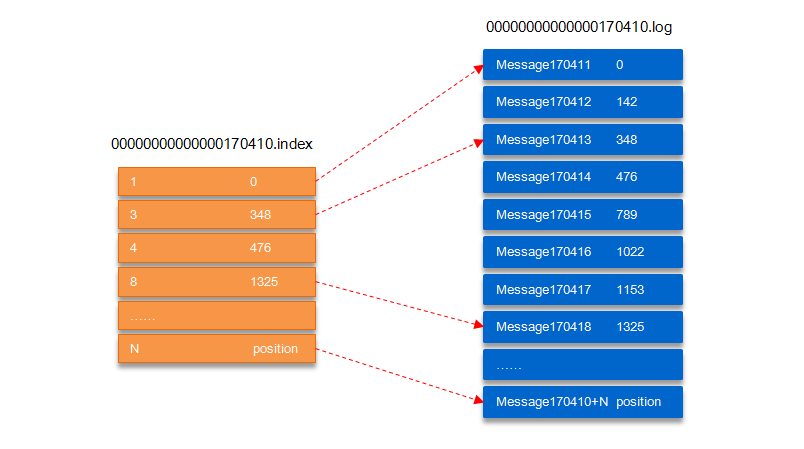
## 存储机制

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

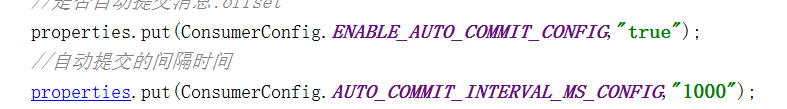


## 查找方式

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

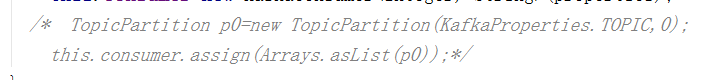
手动异步提交

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｝