目录

[一、 基础篇 1](#_Toc415491916)

[1. 开篇说明 1](#_Toc415491917)

[2. 概念说明 1](#_Toc415491918)

[3. 配置说明 3](#_Toc415491919)

[4. znode分类 17](#_Toc415491920)

[5. kafka协议分类 24](#_Toc415491921)

[6. Kafka线程 29](#_Toc415491922)

[7. 日志存储格式 30](#_Toc415491923)

[8. kakfa架构设计 35](#_Toc415491924)

[二、 流程篇 36](#_Toc415491925)

[1、 kafka启动过程 36](#_Toc415491926)

[2、 日志初始化和清理过程 38](#_Toc415491927)

[3、 选举controller过程 39](#_Toc415491928)

[4、 controller处理broker startup过程 39](#_Toc415491929)

[5、 controller处理broker failure过程 40](#_Toc415491930)

[6、 broker成leader、follower过程 41](#_Toc415491931)

[7、 produce过程 43](#_Toc415491932)

[8、 新建topic-partition过程 44](#_Toc415491933)

[9、 consume过程 46](#_Toc415491934)

[10、 controlled shutdown过程 47](#_Toc415491935)

[11、 preferred election过程 47](#_Toc415491936)

[12、 reassignment过程 47](#_Toc415491937)

[13、 topic config change过程 48](#_Toc415491938)

[三、 工具篇 48](#_Toc415491939)

[四、 FAQ 52](#_Toc415491940)

[五、 监控篇 53](#_Toc415491941)

# 基础篇

### 开篇说明

kafka是一个分布式消息系统，具有高可用、高性能、分布式、高扩展、持久性等特性。学好kafka对于理解分布式精髓意义重大，本文档旨在讲kafka的原理，对于delete topic等未实现的功能不会涉及，对于log compaction因为我没有研究也不会涉及。

### 概念说明

* **Topic**

主题，声明一个主题，producer指定该主题发布消息，订阅该主题的consumer对该主题进行消费

* **Partition**

每个主题可以分为多个分区，每个分区对应磁盘上一个目录，分区可以分布在不同broker上，producer在发布消息时，可以通过指定partition key映射到对应分区，然后向该分区发布消息，在无partition key情况下，随机选取分区，一段时间内触发一次(比如10分钟)，这样就保证了同一个producer向同一partition发布的消息是顺序的

消费者消费时，可以指定partition进行消费，也可以使用high-level-consumer api,自动进行负载均衡，并将partition分给consumer，一个partition只能被一个consumer进行消费。

* **Producer**

生产者，可以多实例部署，可以批量和单条发送，可以同步、异步(多个线程，1-N个线程做生产消息并放入队列,1个线程做发送消息)发送。无论是异步还是同步发送，producer对于一个broker只用一个连接(符合kafka保证消息顺序的设计)，由于只用一个连接，所以发送线程只有一个，尝试用多线程send都是徒劳的。

* **Consumer**

消费者,可以多实例部署,可以批量拉取,有两类API可供选择,一个simpleConsumer,暴露所有的操作给用户，可以提交offset、fetch offset、指定partition fetch message；另外一个high-level-consumer(ZookeeperConsumerConnector),帮助用户做基于partition自动分配的负载均衡,定期提交offset,建立消费队列等。simpleConsumer相当于手动挡，high-level-consumer相当于自动挡。

simpleConsumer无需像high-level-consumer那样向zk注册brokerid、owner，甚至不需要提交offset到zk，可以将offset提交到任意地方比如(mysql,本地文件)。

high-level-consumer，一个进程中可以启多个消费线程，一个消费线程即是一个consumer，假设A进程里有2个线程(consumerid分别为1,2)，B线程有2个线程(consumerid分别为1,2)，topic1的partition有5个，那么partition分配是这样的：

partition1--->A进程consumerid1

partition2--->A进程consumerid1

partition3--->A进程consumerid2

partition4--->B进程consumer1

partition5--->B进程consumer2

* **Group**

High-level-consumer可以声明group，每个group可以有多个consumer，每group

各自管理各自的消费offset，各个不同group之间互不关联影响。

由于目前版本消费的offset、owner、group都是consumer自己通过zk管理，所以group对于broker和producer并不关心，一些监控工具需要通过group来监控，simpleComsumer无需声明group

* **Leader**

每个partition，都有一个leader，0-N个follower，N=ReplicationFactor(复制系数)-1，leader+follower=replica,in-sync-replica=isr

Leader负责该partition的client的读(fetch)请求和写(send)请求以及follower的读(fetch)请求

Leader处理follower的fetch请求和producer client的produce请求(仅当ack=-1或N)使用到了Delayed response机制(client端长polling，broker delay response)。

当fetch请求先到来，事先hold住fetch请求，有produce请求并写入日志时通知队列释放fetch请求的response，同时produce请求也因follower同步了消息数据而得到响应

Leader负责管理ISR的状态，当follower所同步的消息赶上或者落后与leader某个固定阀值，leader将该follower拉进isr(更新到zk上)，长时间未发同步fetch请求或者落后offset差值大于阀值，leader就会将该follower从isr中移除

* **Follower**

follower负责当leader失联后做故障恢复参与选主(leader)和备份用,成为follower后，就启动fetch线程(一个broker(leader)一个)不停的向leader同步消息到本地

* **ISR**

处于同步状态的follower列表，是in-sync-replica的缩写，replica分为二类：一：处于同步状态的replica即isr，二：处于离线状态的replica

* **Replica**

Topic-partition创建的时候分配的replica，不管被分配为replica的broker有没有topic消息数据，它始终都是replica，保存在zk /brokers/topics/[topic]节点中，除非做reassign

* **Controller**

Kafka集群里面有一台broker作为Controller，controller检测broker failure、进行选leader操作、管理和同步topic-partition元数据和replica状态到各个broker、

Zk上的topic state节点leader项及leader epoch完全由controller控制、在某个broker挂掉后，会做移除isr操作，reassign和preferred elec及delete topic都由controller来做。

每台broker启动时会竞争参选controller，当发现已经有controller时，会自动放弃，并监控/controller节点，当/controller session 过期时再次竞争参选

* **Offset**

Offset是相对于第一条消息的位移，第一条消息的offset是0，在log文件中，offset字段被定义为相当于当前segment的位移，比如当前segment的起始offset是00000006，那么第一消息的offset就是00000001。

Consumer和follower会传递offset字段给leader，来获取offset之后的消息，consumer会将offset提交到zk上

**endlogOffset**--->指topic-partition log目录里面最后的一条消息的offset

* **HighWatermark**

一个partition的isr列表中，所有isr列表里broker中同步的最低的那条消息offset。和木桶原理一样，水位取决于最低那块短板，即highwatermark取决于最低的那条offset。

那highwatermark在什么时候使用呢，在重新选leader的时候，follower会将日志trucate至highwatermark，然后再去主同步数据，这样能保证数据一致性，但是有可能消息数据会丢失

* **controllerEpoch**

为了防止先发的请求后到来导致broker数据不一致，所以使用版本管理数据，每次更换controller，epoch加1，所以broker永远只响应本次请求中epoch>=上次请求epoch的请求。

* **leaderEpoch**

为了防止先发的请求后到来导致broker数据不一致，所以使用版本管理数据，每次选主更换leader，epoch加1，所以broker永远只响应本次请求中epoch>=上次请求epoch的请求

### 配置说明

* **Server端配置**

目前对topic单独配置，除了partition和replication.factor，就只有logconfig只影响log。

其余的配置不支持动态修改，对于topic可以在创建的时候和修改的时候修改log相关config，也可以通过kafka提供的脚本工具修改针对某个topic的replication.factor和partition

[**https://cwiki.apache.org/confluence/display/KAFKA/Dynamic+Topic+Config**](https://cwiki.apache.org/confluence/display/KAFKA/Dynamic+Topic+Config)

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Default** | **Description** | **备注** |
| broker.id |  | Each broker is uniquely identified by a non-negative integer id. This id serves as the broker's "name" and allows the broker to be moved to a different host/port without confusing consumers. You can choose any number you like so long as it is unique. |  |
| log.dirs | /tmp/kafka-logs | A comma-separated list of one or more directories in which Kafka data is stored. Each new partition that is created will be placed in the directory which currently has the fewest partitions. | 当新partition创建时它会进入  包含partition最少的目录中 |
| port | 6667 | The port on which the server accepts client connections. |  |
| zookeeper.connect | null | Specifies the ZooKeeper connection string in the form hostname:port, where hostname and port are the host and port for a node in your ZooKeeper cluster. To allow connecting through other ZooKeeper nodes when that host is down you can also specify multiple hosts in the formhostname1:port1,hostname2:port2,hostname3:port3.  ZooKeeper also allows you to add a "chroot" path which will make all kafka data for this cluster appear under a particular path. This is a way to setup multiple Kafka clusters or other applications on the same ZooKeeper cluster. To do this give a connection string in the formhostname1:port1,hostname2:port2,hostname3:port3/chroot/pathwhich would put all this cluster's data under the path /chroot/path. Note that you must create this path yourself prior to starting the broker and consumers must use the same connection string. | 可以有一个chroot  用于防止和zk其它业务节点冲突 |
| message.max.bytes | 1000000 | The maximum size of a message that the server can receive. It is important that this property be in sync with the maximum fetch size your consumers use or else an unruly producer will be able to publish messages too large for consumers to consume. | 单条message最大字节数，consumers在fetchSize至少要大于该配置，不然有可能会永远拿不到消息 |
| num.network.threads | 3 | The number of network threads that the server uses for handling network requests. You probably don't need to change this. | 做NIO操作，read/write from socket |
| num.io.threads | 8 | The number of I/O threads that the server uses for executing requests. You should have at least as many threads as you have disks. | Request handler，负责处理请求 |
| background.threads | 4 | The number of threads to use for various background processing tasks such as file deletion. You should not need to change this. | 用做定时任务，包括  1、cleanupLogs  2、flushDirtyLogs  3、  checkpointRecoveryPointOffsets  4、  checkpointHighWatermarks  5、maybeShrinkIsr  异步执行一次，包括   1. flush old segment 2. delete segment |
| queued.max.requests | 500 | The number of requests that can be queued up for processing by the I/O threads before the network threads stop reading in new requests. | 异步队列最大容量 |
| host.name | null | Hostname of broker. If this is set, it will only bind to this address. If this is not set, it will bind to all interfaces, and publish one to ZK. | Server socket绑定地址，一般不需要设置，默认绑定所有网卡 |
| advertised.host.name | null | If this is set this is the hostname that will be given out to producers, consumers, and other brokers to connect to. | 通过topicmetadata请求返回，建议producer、consumer用什么host来连接，如果没设置则使用host.name |
| advertised.port | null | The port to give out to producers, consumers, and other brokers to use in establishing connections. This only needs to be set if this port is different from the port the server should bind to. |  |
| socket.send.buffer.bytes | 100 \* 1024 | The SO\_SNDBUFF buffer the server prefers for socket connections. | Send缓冲区，根据业务来定 |
| socket.receive.buffer.bytes | 100 \* 1024 | The SO\_RCVBUFF buffer the server prefers for socket connections. | Receive缓冲区，根据业务来设置 |
| socket.request.max.bytes | 100 \* 1024 \* 1024 | The maximum request size the server will allow. This prevents the server from running out of memory and should be smaller than the Java heap size. | 一次请求能接收的最多字节数，一次请求由多条messages组成，每条又不能超过message.max.bytes |
| num.partitions | 1 | The default number of partitions per topic if a partition count isn't given at topic creation time. | Kafka会默认将partition均匀的分布在各个broker上 |
| log.segment.bytes | 1024 \* 1024 \* 1024 | The log for a topic partition is stored as a directory of segment files. This setting controls the size to which a segment file will grow before a new segment is rolled over in the log. This setting can be overridden on a per-topic basis (see [the per-topic configuration section](https://kafka.apache.org/documentation.html#topic-config)). | 每个segment的最大大小，超过这个大小就会重新生成一个，默认是1G |
| log.roll.hours | 24 \* 7 | This setting will force Kafka to roll a new log segment even if the log.segment.bytes size has not been reached. This setting can be overridden on a per-topic basis (see [the per-topic configuration section](https://kafka.apache.org/documentation.html#topic-config)). | 每个segment的多久才重新生成一个。和上一个配置一样，两者有一个条件满足就会滚动一个新的segment |
| log.cleanup.policy | delete | This can take either the value delete or compact. If delete is set, log segments will be deleted when they reach the size or time limits set. If compact is set [log compaction](https://kafka.apache.org/documentation.html#compaction) will be used to clean out obsolete records. This setting can be overridden on a per-topic basis (see [the per-topic configuration section](https://kafka.apache.org/documentation.html#topic-config)). | 清理日志策略，两个选项delete和compact。Compact我没有研究 |
| log.retention.{minutes,hours} | 7 days | The amount of time to keep a log segment before it is deleted, i.e. the default data retention window for all topics. Note that if both log.retention.minutes and log.retention.bytes are both set we delete a segment when either limit is exceeded. This setting can be overridden on a per-topic basis (see [the per-topic configuration section](https://kafka.apache.org/documentation.html#topic-config)). | 一个segment最大能够多大才会被删除 |
| log.retention.bytes | -1 | The amount of data to retain in the log for each topic-partitions. Note that this is the limit per-partition so multiply by the number of partitions to get the total data retained for the topic. Also note that if both log.retention.hours and log.retention.bytes are both set we delete a segment when either limit is exceeded. This setting can be overridden on a per-topic basis (see [the per-topic configuration section](https://kafka.apache.org/documentation.html#topic-config)). | 每个topic-partition目录大小最大多大才会开始清理单个segment。上述两个条件有一个满足就删除掉 |
| log.retention.check.interval.ms | 5 minutes | The period with which we check whether any log segment is eligible for deletion to meet the retention policies. | 日志留存检测间隔时间 |
| log.cleaner.enable | false | This configuration must be set to true for log compaction to run. |  |
| log.cleaner.threads | 1 | The number of threads to use for cleaning logs in log compaction. |  |
| log.cleaner.io.max.bytes.per.second | None | The maximum amount of I/O the log cleaner can do while performing log compaction. This setting allows setting a limit for the cleaner to avoid impacting live request serving. |  |
| log.cleaner.dedupe.buffer.size | 500\*1024\*1024 | The size of the buffer the log cleaner uses for indexing and deduplicating logs during cleaning. Larger is better provided you have sufficient memory. |  |
| log.cleaner.io.buffer.size | 512\*1024 | The size of the I/O chunk used during log cleaning. You probably don't need to change this. |  |
| log.cleaner.io.buffer.load.factor | 0.9 | The load factor of the hash table used in log cleaning. You probably don't need to change this. |  |
| log.cleaner.backoff.ms | 15000 | The interval between checks to see if any logs need cleaning. |  |
| log.cleaner.min.cleanable.ratio | 0.5 | This configuration controls how frequently the log compactor will attempt to clean the log (assuming [log compaction](https://kafka.apache.org/documentation.html#compaction) is enabled). By default we will avoid cleaning a log where more than 50% of the log has been compacted. This ratio bounds the maximum space wasted in the log by duplicates (at 50% at most 50% of the log could be duplicates). A higher ratio will mean fewer, more efficient cleanings but will mean more wasted space in the log. This setting can be overridden on a per-topic basis (see [the per-topic configuration section](https://kafka.apache.org/documentation.html#topic-config)). |  |
| log.cleaner.delete.retention.ms | 1 day | The amount of time to retain delete tombstone markers for [log compacted](https://kafka.apache.org/documentation.html#compaction) topics. This setting also gives a bound on the time in which a consumer must complete a read if they begin from offset 0 to ensure that they get a valid snapshot of the final stage (otherwise delete tombstones may be collected before they complete their scan). This setting can be overridden on a per-topic basis (see [the per-topic configuration section](https://kafka.apache.org/documentation.html#topic-config)). |  |
| log.index.size.max.bytes | 10 \* 1024 \* 1024 | The maximum size in bytes we allow for the offset index for each log segment. Note that we will always pre-allocate a sparse file with this much space and shrink it down when the log rolls. If the index fills up we will roll a new log segment even if we haven't reached the log.segment.bytes limit. This setting can be overridden on a per-topic basis (see [the per-topic configuration section](https://kafka.apache.org/documentation.html#topic-config)). | Log.index文件最大多少才会重新roll一个，只要index文件roll那么无论怎么样，log都会roll一个 |
| log.index.interval.bytes | 4096 | The byte interval at which we add an entry to the offset index. When executing a fetch request the server must do a linear scan for up to this many bytes to find the correct position in the log to begin and end the fetch. So setting this value to be larger will mean larger index files (and a bit more memory usage) but less scanning. However the server will never add more than one index entry per log append (even if more than log.index.interval worth of messages are appended). In general you probably don't need to mess with this value. | 每隔多少字节kafka会将log文件里面的消息位置记录到index文件，这个值越小，index文件越大，检索越少，由于index文件都被映射到内存里面，所以占用的内存也多。  假如足够小，也不会一条消息记录多次，一般来说一个index文件大小=segment大小/4096\*12 |
| log.flush.interval.messages | None | The number of messages written to a log partition before we force an fsync on the log. Setting this lower will sync data to disk more often but will have a major impact on performance. We generally recommend that people make use of replication for durability rather than depending on single-server fsync, however this setting can be used to be extra certain. | Log积攒多少消息刷到磁盘 |
| log.flush.scheduler.interval.ms | 3000 | The frequency in ms that the log flusher checks whether any log is eligible to be flushed to disk. | 每次检测一次间隔时间 |
| log.flush.interval.ms | None | The maximum time between fsync calls on the log. If used in conjuction with log.flush.interval.messages the log will be flushed when either criteria is met. | 间隔多久被刷到磁盘 |
| log.delete.delay.ms | 60000 | The period of time we hold log files around after they are removed from the in-memory segment index. This period of time allows any in-progress reads to complete uninterrupted without locking. You generally don't need to change this. | 异步延迟删除间隔 |
| log.flush.offset.checkpoint.interval.ms | 60000 | The frequency with which we checkpoint the last flush point for logs for recovery. You should not need to change this. | 一分钟进行一次checkpoint的写入 |
| auto.create.topics.enable | true | Enable auto creation of topic on the server. If this is set to true then attempts to produce, consume, or fetch metadata for a non-existent topic will automatically create it with the default replication factor and number of partitions. | broker接收到不存在  topic metadata请求时，会不会自动创建该topic，不管创建成功与否，本次topic metadata请求都会返回错误码 |
| controller.socket.timeout.ms | 30000 | The socket timeout for commands from the partition management controller to the replicas. | Controller向replica发请求时socket.timeout |
| controller.message.queue.size | 10 | The buffer size for controller-to-broker-channels | Controller对单个broker所发送的数据最多多少条，超过就阻塞 |
| default.replication.factor | 1 | The default replication factor for automatically created topics. | 默认副本个数 |
| replica.lag.time.max.ms | 10000 | If a follower hasn't sent any fetch requests for this window of time, the leader will remove the follower from ISR (in-sync replicas) and treat it as dead. | 如果一个副本落后leader超过这个时间间隔都没有发送同步消息请求，那就将该follower移除isr |
| replica.lag.max.messages | 4000 | If a replica falls more than this many messages behind the leader, the leader will remove the follower from ISR and treat it as dead. | 如果一个副本落后leader超过这个消息数，那就将该follower移除isr |
| replica.socket.timeout.ms | 30 \* 1000 | The socket timeout for network requests to the leader for replicating data. | Follower发送fetch请求的sockettimeout |
| replica.socket.receive.buffer.bytes | 64 \* 1024 | The socket receive buffer for network requests to the leader for replicating data. |  |
| replica.fetch.max.bytes | 1024 \* 1024 | The number of byes of messages to attempt to fetch for each partition in the fetch requests the replicas send to the leader. | 顾名思义，如果单条消息大于此，那么该数值也要增加 |
| replica.fetch.wait.max.ms | 500 | The maximum amount of time to wait time for data to arrive on the leader in the fetch requests sent by the replicas to the leader. | Delay response最大时间，超时则返回响应可以无消息数据 |
| replica.fetch.min.bytes | 1 | Minimum bytes expected for each fetch response for the fetch requests from the replica to the leader. If not enough bytes, wait up to replica.fetch.wait.max.ms for this many bytes to arrive. | 至少要超时或者所得消息达到该配置字节数才能返回给follower响应 |
| num.replica.fetchers | 1 | Number of threads used to replicate messages from leaders. Increasing this value can increase the degree of I/O parallelism in the follower broker. | Follower端使用，对于一个leader broker需要设置几个线程来同步消息，默认一个broker只用一个thread |
| replica.high.watermark.checkpoint.interval.ms | 5000 | The frequency with which each replica saves its high watermark to disk to handle recovery. | 间隔多少时间去做一次保存一次watermark |
| fetch.purgatory.purge.interval.requests | 10000 | The purge interval (in number of requests) of the fetch request purgatory. | 为了不让delay的request过多导致内存溢出，当fetchre quest积累的request达到该数值时，则主动进行expire操作，也就是释放响应 |
| producer.purgatory.purge.interval.requests | 10000 | The purge interval (in number of requests) of the producer request purgatory. | 为了不让delay的request过多导致内存溢出，当produce request积累的request达到该数值时，则主动进行expire操作，也就是释放响应 |
| zookeeper.session.timeout.ms | 6000 | ZooKeeper session timeout. If the server fails to heartbeat to ZooKeeper within this period of time it is considered dead. If you set this too low the server may be falsely considered dead; if you set it too high it may take too long to recognize a truly dead server. | Broker session timeout，最小是2倍的ticktime，最大是20倍的ticktime |
| zookeeper.connection.timeout.ms | 6000 | The maximum amount of time that the client waits to establish a connection to zookeeper. | 连zk超时时间 |
| zookeeper.sync.time.ms | 2000 | How far a ZK follower can be behind a ZK leader. | 只有high-consumer使用这个配置！ |
| controlled.shutdown.enable | false | Enable controlled shutdown of the broker. If enabled, the broker will move all leaders on it to some other brokers before shutting itself down. This reduces the unavailability window during shutdown. | 如果开启，在brokershutdown时候，会发送给controller请求，让其做重新选主，controller发送stopreplica请求给该机器，开启能够降低不可用的时间 |
| controlled.shutdown.max.retries | 3 | Number of retries to complete the controlled shutdown successfully before executing an unclean shutdown. |  |
| controlled.shutdown.retry.backoff.ms | 5000 | Backoff time between shutdown retries. |  |
| auto.leader.rebalance.enable | false | If this is enabled the controller will automatically try to balance leadership for partitions among the brokers by periodically returning leadership to the "preferred" replica for each partition if it is available. | 如果开启，那么在达到一定ratio后，controller会重新分配leader，preferred replica会被分配为主 |
| leader.imbalance.per.broker.percentage | 10 | The percentage of leader imbalance allowed per broker. The controller will rebalance leadership if this ratio goes above the configured value per broker. | 为了不让reblance那么频繁和没必要，设置该值，超过这个值就触发。  错主/所有主=该配置 |
| leader.imbalance.check.interval.seconds | 300 | The frequency with which to check for leader imbalance. | 开启rebalance，就会开启一个线程定期检测，该配置定义检测的间隔时间 |
| offset.metadata.max.bytes | 1024 | The maximum amount of metadata to allow clients to save with their offsets. |  |

* **Consume 端配置**

|  |  |  |  |
| --- | --- | --- | --- |
| Property | Default | Description |  |
| group.id |  | A string that uniquely identifies the group of consumer processes to which this consumer belongs. By setting the same group id multiple processes indicate that they are all part of the same consumer group. |  |
| zookeeper.connect |  | Specifies the ZooKeeper connection string in the form hostname:portwhere host and port are the host and port of a ZooKeeper server. To allow connecting through other ZooKeeper nodes when that ZooKeeper machine is down you can also specify multiple hosts in the formhostname1:port1,hostname2:port2,hostname3:port3.  The server may also have a ZooKeeperchroot path as part of it'sZooKeeper connection string which puts its data under some path in the global ZooKeeper namespace. If so the consumer should use the same chroot path in its connection string. For example to give a chroot path of /chroot/path you would give the connection string ashostname1:port1,hostname2:port2,hostname3:port3/chroot/path. |  |
| consumer.id | null | Generated automatically if not set. | 每个consumer生成一个默认是  机器名-时间戳-线程id |
| socket.timeout.ms | 30 \* 1000 | The socket timeout for network requests. The actual timeout set will be max.fetch.wait + socket.timeout.ms. | 读超时 |
| socket.receive.buffer.bytes | 64 \* 1024 | The socket receive buffer for network requests |  |
| fetch.message.max.bytes | 1024 \* 1024 | The number of byes of messages to attempt to fetch for each topic-partition in each fetch request. These bytes will be read into memory for each partition, so this helps control the memory used by the consumer. The fetch request size must be at least as large as the maximum message size the server allows or else it is possible for the producer to send messages larger than the consumer can fetch. | 批量日志大小 |
| auto.commit.enable | true | If true, periodically commit to ZooKeeper the offset of messages already fetched by the consumer. This committed offset will be used when the process fails as the position from which the new consumer will begin. | 是否自动提交，默认是，单独线程定期提交，否则只能自己显示提交 |
| auto.commit.interval.ms | 60 \* 1000 | The frequency in ms that the consumer offsets are committed to zookeeper. | 定时任务间隔时间 |
| queued.max.message.chunks | 10 | Max number of message chunks buffered for consumption. Each chunk can be up to fetch.message.max.bytes. | 队列长度 |
| rebalance.max.retries | 4 | When a new consumer joins a consumer group the set of consumers attempt to "rebalance" the load to assign partitions to each consumer. If the set of consumers changes while this assignment is taking place the rebalance will fail and retry. This setting controls the maximum number of attempts before giving up. |  |
| fetch.min.bytes | 1 | The minimum amount of data the server should return for a fetch request. If insufficient data is available the request will wait for that much data to accumulate before answering the request. |  |
| fetch.wait.max.ms | 100 | The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy fetch.min.bytes | Fetch最大等待时间 |
| rebalance.backoff.ms | 2000 | Backoff time between retries during rebalance. | 如果没有就取zookeeper.sync.time.ms |
| refresh.leader.backoff.ms | 200 | Backoff time to wait before trying to determine the leader of a partition that has just lost its leader. |  |
| auto.offset.reset | largest | What to do when there is no initial offset in ZooKeeper or if an offset is out of range: \* smallest : automatically reset the offset to the smallest offset \* largest : automatically reset the offset to the largest offset \* anything else: throw exception to the consumer | 从最后一条消息开始消费 |
| consumer.timeout.ms | -1 | Throw a timeout exception to the consumer if no message is available for consumption after the specified interval | 不要设置这个 |
| client.id | group id value | The client id is a user-specified string sent in each request to help trace calls. It should logically identify the application making the request. |  |
| zookeeper.session.timeout.ms | 6000 | ZooKeeper session timeout. If the consumer fails to heartbeat to ZooKeeper for this period of time it is considered dead and a rebalance will occur. |  |
| zookeeper.connection.timeout.ms | 6000 | The max time that the client waits while establishing a connection to zookeeper. |  |
| zookeeper.sync.time.ms | 2000 | How far a ZK follower can be behind a ZK leader |  |

* **Produce config**

|  |  |  |  |
| --- | --- | --- | --- |
| Property | Default | Description |  |
| metadata.broker.list |  | This is for bootstrapping and the producer will only use it for getting metadata (topics, partitions and replicas). The socket connections for sending the actual data will be established based on the broker information returned in the metadata. The format is host1:port1,host2:port2, and the list can be a subset of brokers or a VIP pointing to a subset of brokers. | 做metadata 请求的broker列表，尽量越多越好，或者使用VIP，或者用代理转发 |
| request.required.acks | 0 | This value controls when a produce request is considered completed. Specifically, how many other brokers must have committed the data to their log and acknowledged this to the leader? Typical values are  0, which means that the producer never waits for an acknowledgement from the broker (the same behavior as 0.7). This option provides the lowest latency but the weakest durability guarantees (some data will be lost when a server fails).  1, which means that the producer gets an acknowledgement after the leader replica has received the data. This option provides better durability as the client waits until the server acknowledges the request as successful (only messages that were written to the now-dead leader but not yet replicated will be lost).  -1, which means that the producer gets an acknowledgement after all in-sync replicas have received the data. This option provides the best durability, we guarantee that no messages will be lost as long as at least one in sync replica remains. | 0是不接收响应，很容易丢消息，因为不知道成功与否  1是leader写入成功就响应，所以未同步到follower的消息有可能丢失  -1或者N是等待所有或N个follower响应才返回client response，这会让响应变慢，但是不会丢消息。  以上所有配置都有可能发送重复消息 |
| request.timeout.ms | 10000 | The amount of time the broker will wait trying to meet the request.required.acks requirement before sending back an error to the client. |  |
| producer.type | sync | This parameter specifies whether the messages are sent asynchronously in a background thread. Valid values are (1) async for asynchronous send and (2) sync for synchronous send. By setting the producer to async we allow batching together of requests (which is great for throughput) but open the possibility of a failure of the client machine dropping unsent data. |  |
| serializer.class | kafka.serializer.DefaultEncoder | The serializer class for messages. The default encoder takes a byte[] and returns the same byte[]. |  |
| key.serializer.class |  | The serializer class for keys (defaults to the same as for messages if nothing is given). |  |
| partitioner.class | kafka.producer.DefaultPartitioner | The partitioner class for partitioning messages amongst sub-topics. The default partitioner is based on the hash of the key. |  |
| compression.codec | none | This parameter allows you to specify the compression codec for all data generated by this producer. Valid values are "none", "gzip" and "snappy". |  |
| compressed.topics | null | This parameter allows you to set whether compression should be turned on for particular topics. If the compression codec is anything other than NoCompressionCodec, enable compression only for specified topics if any. If the list of compressed topics is empty, then enable the specified compression codec for all topics. If the compression codec is NoCompressionCodec, compression is disabled for all topics |  |
| message.send.max.retries | 3 | This property will cause the producer to automatically retry a failed send request. This property specifies the number of retries when such failures occur. Note that setting a non-zero value here can lead to duplicates in the case of network errors that cause a message to be sent but the acknowledgement to be lost. | 建议设置的多些，比如10次，一旦超过这个次数，produce将抛异常 |
| retry.backoff.ms | 100 | Before each retry, the producer refreshes the metadata of relevant topics to see if a new leader has been elected. Since leader election takes a bit of time, this property specifies the amount of time that the producer waits before refreshing the metadata. | 建议设置的稍微大些，  因为最合适的是sesstion time时间，不过sesstion time时间显然太长了，所以只能加大retry次数 |
| topic.metadata.refresh.interval.ms | 600 \* 1000 | The producer generally refreshes the topic metadata from brokers when there is a failure (partition missing, leader not available...). It will also poll regularly (default: every 10min so 600000ms). If you set this to a negative value, metadata will only get refreshed on failure. If you set this to zero, the metadata will get refreshed after each message sent (not recommended). Important note: the refresh happen only AFTER the message is sent, so if the producer never sends a message the metadata is never refreshed |  |
| queue.buffering.max.ms | 5000 | Maximum time to buffer data when using async mode. For example a setting of 100 will try to batch together 100ms of messages to send at once. This will improve throughput but adds message delivery latency due to the buffering. | 文档中说的不对，  实际代码实现是  在上次发送减去该配置时间内返回为null，则需要发送。  如果队列中一直有，  只有满足max.messages才会发送。  该配置的真正作用是  在积攒max.message的过程中，并且超过max.ms时间间隔，有队列为空的情况出现 |
| queue.buffering.max.messages | 10000 | The maximum number of unsent messages that can be queued up the producer when using async mode before either the producer must be blocked or data must be dropped. |  |
| queue.enqueue.timeout.ms | -1 | The amount of time to block before dropping messages when running in async mode and the buffer has reached queue.buffering.max.messages. If set to 0 events will be enqueued immediately or dropped if the queue is full (the producer send call will never block). If set to -1 the producer will block indefinitely and never willingly drop a send. |  |
| batch.num.messages | 200 | The number of messages to send in one batch when using async mode. The producer will wait until either this number of messages are ready to send or queue.buffer.max.ms is reached. |  |
| send.buffer.bytes | 100 \* 1024 | Socket write buffer size |  |
| client.id | "" | The client id is a user-specified string sent in each request to help trace calls. It should logically identify the application making the request. |  |

### znode分类

* **zk目录**

|  |  |  |  |
| --- | --- | --- | --- |
| Znode path | 说明 | | |
| /consumers/group/ids/[consumer.id] | 数据 | 节点名是 通过配置 consumer.id得到  Group名-hostname-timestamp-随机串  group\_CY9772-1422585839420-e9aa5caf  数据包括version、subscription、pattern  Timestamp | |
| 例子 | {"version":1,  "subscription":{"topic1\_1":3},  "pattern":"static","timestamp"  :"1422585839563"} | |
| 属性 | **临时节点** | |
| 备注 | Consumer开始消费时，主动注册zk，pattern static还有一种是Wildcard支持对topic进行过滤。  High-level-consumer专有 | |
| Watch | 1、consumer的ZKRebalancerListener(childrenLister)监听该节点，有消失或者有新成员就进行relabance | |
| /consumers/group/offsets/[topic]/[partition] | 数据 | 存放该group 消费到的offsets | |
| 例子 | 5943 | |
| 属性 | **永久节点** | |
| 备注 | 由consumer定期提交offset到该节点，high-level-consumer专有 | |
| Watch | 无 | |
| /consumers/group/owners/[topic]/[partition] | 数据 | Owner是 consumerThreadId  格式是consumer.id--threadid | |
| 例子 | [zk: localhost:2181(CONNECTED) 8] get /consumers/group/owners/topic1\_1/0  group\_CY9772-1422585839420-e9aa5caf-0  [zk: localhost:2181(CONNECTED) 9] get /consumers/group/owners/topic1\_1/1  group\_CY9772-1422585839420-e9aa5caf-1 | |
| 属性 | **临时节点** | |
| 备注 | 每次进行rebance的时候都会抢占分配了的，有一个没抢占成功，都会释放其它所有的，最后尝试N次抢占，抢不成功就剖异常 | |
| Watch | 无 | |
| /brokers/ids/[brokerid] | 数据 | 存放broker host:port信息，这里host是advisehostname | |
| 例子 | {"jmx\_port":-1,  "timestamp":"1418110043379",  "host":"AY140722231919412730Z",  "version":1,"port":9092} | |
| 属性 | **临时节点** | |
| 备注 | Broker启动成功后会将其注册到zk，  该节点被controller监听、consumer监听 | |
| Watch | 1、controller的BrokerChangeListener  2、consumer的ZKRebalancerListener | |
| /brokers/topics/[topic] | 数据 | 该topic下partitions和replica的对应关系 | |
| 例子 | {"version":1,"partitions":  {"1":[0,1],"0":[1,0]}}  表示partition1拥有两个replica，分别是id0和id1 | |
| 属性 | **永久节点** | |
| 备注 | 当topic创建时，该节点先生成，数据也填充好，之后除非通过reassign工具，否则数据不会被修改的 | |
| Watch | 1. controller 的   TopicChangeListener(children listner)  2、/brokers/topics/{your\_topic} （data listener ）AddPartitionsListener(controller的)ZKTopicPartitionChangeListener(包含ZKRebalancerListener） | |
| /brokers/topics/[topic]/partitions/[partition]  /state | 数据 | 包括controller\_epoch,leader,leader\_epoch  isr | |
| 例子 | {"controller\_epoch":1,  "leader":1,"version":1,  "leader\_epoch":0,"isr":[1]} | |
| 属性 | 永久节点 | |
| 备注 | Leader只有controller能修改，isr一般leader修改的最多，只有broker down的时候controller  才会修改isr，将离线的broker移除isr | |
| Watch | 1. ReassignedPartitionsIsrChangeListener   由Controller监听，做reassign的时候  做的监听，用于监听新分配的replica是否已经在ISR中 | |
| /controller | 数据 | 存放controller信息 | |
| 例子 | {"version":1,"brokerid":1,  "timestamp":"1418110043222"} | |
| 属性 | **临时节点** | |
| 备注 | 每次broker启动或者监听到/controller节点失效  就会触发抢占该节点注册的事件，抢占后epoch加1 | |
| Watch | 1、每个broker都会监听LeaderChangeListener | |
| /controller\_epoch | 数据 | Controller\_epoch数字 |
| 例子 | 1 |
| 属性 | **永久节点** |
| 备注 |  |
| Watch | 所有broker监听，ControllerEpochListener  仅仅是将controller\_epoch写入缓存 |
| /config/changes | 数据 | /config/changes/config\_change\_6  "topic1" | |
| 例子 | /config/changes/config\_change\_6  "topic2" | |
| 属性 | 永久节点 | |
| 备注 | 在做alter topic时,会生成类似config\_change\_id的  节点，ConfigChangeListener监听得到节点数据对应的  Topic然后到/config/topics/[topic]拿数据覆盖logmanager里面的配置 | |
| Watch | 1、ConfigChangeListener，所有broker都监听，因为不一定自己的log里面存在这里配的topic | |
| /config/topics | 数据 | /config/topics/[topic] | |
| 例子 | {“version”:1,  ”config”:{“properties1”:”value1”}} | |
| 属性 | **永久节点** | |
| 备注 | 当修改topic时，会将该节点填数据 | |
| Watch |  | |
| /admin/reassign\_partitions | 数据 | 包括topic-partition，replica | |
| 例子 | {"version":1,  "partitions":[{"topic":"foo1","partition":2,"  replicas":[1,2]},  {"topic":"foo1","partition":0,"replicas":[3,4]},  {"topic":"foo2","partition":2,"replicas":[1,2]},  {"topic":"foo2","partition":0,"replicas":[3,4]},  {"topic":"foo1","partition":1,"replicas":[2,3]},  {"topic":"foo2","partition":1,"replicas":[2,3]}] | |
| 属性 | 永久 | |
| 备注 | 启用reassign工具，才会更改该节点，然后controller监听来处理重分配 | |
| Watch | Controller来监听  PartitionsReassignedListener | |
| /admin/delete\_topics |  |  | |
|  |  | |
|  |  | |
| 备注 | 用于删除topic的工具 | |
|  |  | |
| /admin/preferred\_replica\_election | 数据 |  | |
| 例子 | {   "partitions":    [      {"topic": "topic1", "partition": "0"},      {"topic": "topic1", "partition": "1"},      {"topic": "topic1", "partition": "2"},        {"topic": "topic2", "partition": "0"},      {"topic": "topic2", "partition": "1"},    ]  } | |
| 属性 | 永久节点 | |
| 备注 | Preferred工具，controller监听 | |
| Watch | PreferredReplicaElectionListener | |

* **zk监听器**

1. Controller特有的监听器

|  |  |  |  |
| --- | --- | --- | --- |
| 监听器 | 路径&事件 | 作用 | 类型 |
| SessionExpirationListener | 临时路径/controller  事件newSession | 当某个原因失效又恢复后，先shutdown各类controller组件，然后重新参与controller  的选举 | State |
| TopicChangeListener | 路径  /broker/topics  事件  childChange  归属  partitionStateMachine | 当有新topics创建时，controller会触发  onNewTopicCreation  当有topics删除时，controller会将partitionReplicaAssignment中的  Topic删除 | ChildChange |
| BrokerChangeListener | 路径  /broker/ids  事件  childChange  归属  replicaStateMachine | 当有broker失联或者出现时，controller会做如下事情，对于新发现的broker，启动一个线程用来向broker发送请求，并触发onBrokerStartup操作  对于失联的broker关闭掉对应的sendRequest thread，并触发onBrokerFailer操作 | ChildChange |
| AddPartitionsListener | /broker/topics/[topic]  事件  Datachange | 当controller启动时注册，当topic添加时注册，多个topic就多个addPartitionsListener  触发onNewPartitionCreation事件 | datachange |
| PartitionsReassignedListener | 路径/admin/reassign\_partitions  事件  Datachange  Admin工具 | 监听重分配节点，发生变化，就启动重分配 | Datachange |
| PreferredReplicaElectionListener | 路径  /admin/preferred\_replica\_election  事件  Datachange | 监听preferred-replica-elec节点，发生变化就启动preferred-elec | Datachange |
| ReassignedPartitionsIsrChangeListener | 路径  /brokers/topics/{your\_topic}/partitions/{index}/state  事件  Datachange | 只在启动reassign操作时注册，controller监听state节点是否发生变化，如果成功了就触发onReassignMent | datachange |
| DeleteTopicsListener | 路径  /admin/delete\_topics  事件childchange | 当用户通过admin tools，做某个topic进行delete，该节点会增加以topic命名的节点，controller监听该节点触发delete topic操作 | Childchange |

1. 所有broker的监听器

|  |  |  |  |
| --- | --- | --- | --- |
| 监听器 | 路径&事件 | 作用 | 类型 |
| SessionExpireListener | 临时路径  /broker/ids/[brokerid]  事件NewSession | 当某个原因失效又恢复后，重新注册该该临时节点 | State |
| LeaderChangeListener | 路径/controller  事件dataChange  dataDelete | 当/controller数据发生变化，缓存中leaderId(其实写成controllerId更好)要同步修改。  如果节点删除，要重新参与选举，如果自己是controller要首先shutdown一些组件 | DataChange |
| ControllerEpochListener | 路径/controller\_epoch  事件 datachange | 节点数据变化，更新缓存记录controller\_epoch | datachange |
| ConfigChangeListener | 路径/config/changes  事件datachange | 由于topic config change涉及消息文件的配置，所以会和所有broker相关  每个broker会通过logManager将某个topic config 改掉 | Datachange |

1. Consumer的监听器

|  |  |  |  |
| --- | --- | --- | --- |
| 监听器 | 路径&事件 | 作用 | 类型 |
| ZKSessionExpireListener | 临时路径  /consumers/group/[consumer.id]  /consumers/group/owners/  [topic]/[partition]  事件NewSession | 当某个原因失效又恢复后，重新注册consumerid重新进行rebalance，当然也会抢占rebalance后partition的owner | State |
| Kafka.consumer.ZookeeperTopicEventWatcher.  ZkSessionExpireListener(topicEventListener) |  | 当session重新创建时  重新注册ZkTopicEventListener | Children |
| topicEventListener | 监听/brokers/topics/  Children Changed  当topic删除或者增加时  只在Wildcard pattern  Topic使用 | 当topic增加或删除时，增加消费该topic或者取消该topic 的消费 |  |
| ZKTopicPartitionChangeListener( zkRebalancerListener ) | 监听/brokers/topics  事件dataChange | 当/broker/topics的partition有变化时，需要  进行relalance | data |
| ZKRebalancerListener | /consumers/[group]/ids  事件childChange | 当有新成员时，需要进行relalance，因为rebalance是  异步执行，所以有一个单独线程监控一个状态，当状态改变时，执行relance操作 | ChildChange |
|  |  |  |  |

### kafka协议分类

**1)请求汇总**

**请求被接收时会带上requestKeyId，请求处理方根据id来对不同request做处理**

|  |  |  |
| --- | --- | --- |
| **请求key** | **值** | **说明** |
| ProduceKey | 0 | Produce-->leader  用于发送消息给leader |
| FetchKey | 1 | Consumer->leader  Follower-->leader  用于拉取消息、同步消息 |
| OffsetsKey | 2 | Consumer->leader  Follower-->leader  用于获取offset，处理outofrange时需要获取offset |
| MetadataKey | 3 | Consumer-->broker  Producer-->broker  在producer发消息，consumer在rebalance时需要获取topic元数据 |
| LeaderAndIsrKey | 4 | Controller-->broker  (某个partition相关)  用于broker变成leader或者变成follower |
| StopReplicaKey | 5 | Controller--broker  用于停止isr的线程  移除isr，在delete topic和reassign以及  Controllered shutdonw会触发 |
| UpdateMetadataKey | 6 | Controller--broker(所有的)  每次metadata数据更新都会通知所有的broker  让所有的broker都能serve metadata请求 |
| ControlledShutdownKey | 7 | Broker --> controller  Broker接收到shutdown命令后，触发controller shutdown请求给controller |
| OffsetCommitKey | 8 | Consumer-->broker  SimperConsumer专有  借助broker，来提交offset，high-level-consumer可以自己向zk commit |
| OffsetFetchKey | 9 | Consumer-->broker  SimperConsumer专有,  借助broker，来获取zk上的已经消费了的offset  High-level-consumer  可以自己向zk fetch offset |

**2)请求格式**

说明：对于所有的request，

格式都是dataSize+data(requestId+playloadData)，，dataSize用于告知handler请求数据结束位置，这里只介绍playloadData，灰色是request，绿色是response

* **produceRequest**

非压缩的

requestId +

versionId +

correlationId +

clientId      +

requiredAcks  +

ackTimeoutMs  +

topicCount    +

     topic1\_name + topic1AndPartitionDataSize

       +partition1\_id  +  partition1MessageDataSize + [offset + message.size + message.buffer]

[offset + message.size + message.buffer]

         +partition2\_id  +  partition2MessageDataSize + [offset + message.size + message.buffer]

[offset + message.size + message.buffer]

topic2\_name + topic2AndPartitionDataSize

message的Format(message.buffer)：

1. 4 byte CRC32 of the message

2. 1 byte "magic" identifier to allow format changes, value is 2 currently

3. 1 byte "attributes" identifier to allow annotations on the message independent of the version (e.g. compression enabled, type of codec used)

4. 4 byte key length, containing length K

5. K byte key

6. 4 byte payload length, containing length V

7. V byte payload

对于压缩格式的message会有些区别，[offset + message.size + message.buffer]中的message.buffer可能会包含多条消息,也是[offset + message.size + message.buffer]的格式，最后用最后一个offset进行包装

correlationId

topiccount

topic1

partitionCount1 partition errorCode nextOffset partition errorCode nextOffset

       partitionCount2 partition errorCode nextOffset partition errorCode extOffsets

说明：producer向leader发送写请求，相应里面errorCode用来判断哪些失败，并重试；request里面的offset仅仅只是占位符，leader会重新分配替换offset。

* **TopicMetadataRequest**

versionId

correlationId

clientId

numTopics

topicLength1 topic1 topicLength2 topic2

correlationId

brokerCount

  brokerId brokerHostLenth borkerHost brokerPort

topicCount

  errorCode topicLength topic numPartition

                         errorCode1 partition1 leaderId1 numReplicas

                                                                    replica1-id replica2-id replica3-id

                                                                 numIsr

isr1-id isr2-id

* **fetchRequeset**

versionId 0

correlationId 0

clientId  (-1 or clientId.lenth + clientId.string) 73 group1-ConsumerFetcherThread-group1\_XXXXXXX-1421484727311-b5a58168-0-10

replicaId -1

maxWait 100

minBytes 1

requestInfoGroupedByTopic 2

topic1 partitionCount1 partition1 offset1 fetchSize1 partition2 offset2 fetchSize2

topic2 partitionCount2 partition2\_1 offsett2\_1 fetchSize2\_1

correlationId

topicCount

topic

     partitionCount

partitionId   (error,hw,messageSize,messageBuffer)partitionData

            partitionId2  (error,hw,messageSize,messageBuffer)partitionData2

topic1

     .........

* **offsetRequest**

versionId

correlationId

clientId

replicaId   -1代表来自consumer

topicCount

topic   partitionCount    partition     partitionTime      partitionMaxNumOffsets  (client传的是1)

topic1 partitionCount2  partition2   partitionTime2    partitionMaxNumOffsets2

correlationId

numTopics

 topic

   numPartitions

     partition

     error

     numOffsets

      offset1 offset2 offset3 (倒序排列)

说明：consumer用来获取offset之用，最关键的参数是时间戳，broker会拿收到的时间戳与log文件的lastmodified时间做比较

* **LeaderAndIsrRequest**

versionId

correlationId

clientId

controllerId

controllerEpoch

partitionStateInfoSize

   topic partition controllerEpoch leader leaderEpoch isrSize isr1 isr2 isr3 zkVersion replicationFactor replicas1 replica2

leaderSize leaser1 leader2

correlationId

errorCode

numEntries

   topic partition partitionErrorCode

* **UpdateMetadataRequest**

versionId

correlationId

clientId

controllerId

controllerEpoch

partitionStateSize

    topic partition  controllerEpoch leader leaderEpoch isrSize isr1 isr2 isr3 zkVersion replicationFactor replicas1 replica2

aliveBrokerSize

    brokerId1(id,host,port) brokerId2(id,host,port)

correlationId

errorCode

* **StopReplicaRequest**

requestId 5

versionId

correlationId

clientId

controllerId

contrllerEpoch

deletePartitions (true1false0)

partitionSize

   topic partition

correlationId

errorCode

responseMapSize

   topic partition errorCode

* **ControlledShutdownRequest**

versionId

correlationId

brokerId

correlationId

errorCode

topic-partition-size

topic partition

* **OffsetCommitRequest**
* **offsetFetchRequest**

### Kafka线程

|  |  |  |
| --- | --- | --- |
| 类型 | 线程名 | 说明 |
| Controller特有 | Controller-[brokerid]-to-broker-[brokerId]-  Send-thread | 用于controller向broker发送  UpdateMetadatarequest、  Leaderandisrrequest、  StopReplicaRequest |
| Broker | Zk的线程main-EventThread\main-SendThread  zkClient-EventThread | Send-thread用于与zk发送心跳，接收zk事件响应、main-eventThread用于发布事件 |
| Kafka-acceptor | 接收新请求，kafka没有对socket做  过期清理 |
| Kafka-processor | 做network读写，注册读写事件 |
| Kafka-request-handler | 处理请求 |
| ReplicaFetcherThread-%d-brokerid | Follower去leaderfetch message的线程 |
| Request-expiration-task | 用于delay response的expire操作 |
| Kafka-schedule-0 | 定时延时任务，写checkpoint写  Highwatermark，log清理，log删除，移除isr，加入isr等等 |
| Metrics-meter-tick-thread | 用于记录meter指标 |
| Producer | Producer线程 |  |
| Metrics-meter-tick-thread | 用于记录meter指标 |
| Consumer | Zk 3个线程 | 略 |
| Consumer thread | 消费线程，用户迭代消息队列，消费 |
| Kafka-consumer-schedule | 用于提交offset |
| group-hostname-时间戳-随机串-watcher\_executor | 用于监控rebalance状态，状态位被外界修改，就执行relabance操作 |
| Group-hostname-时间戳-随机串  -leader-finder-thread | 寻找leader的线程，启动fetcher线程  管理topic-partition 元数据，  当fetch抛异常时，将该topic-partition 放入noleader缓存里，重新获取元数据 |
| ConsumerFetchThread-group-hostname-时间戳-随机串-0-brokerid | Fetcher线程，用于consumer拉取消息 |
| Metrics-meter-tick-thread | 用于记录meter指标 |

### 日志存储格式

**1)日志目录**

**[root@AY140722231919412730Z kafka]# tree data/**

data/

├──.lock

├── my-replicated-topic-0

│   ├── 00000000000000000000.index

│   └── 00000000000000000000.log

├── recovery-point-offset-checkpoint

├── replication-offset-checkpoint

├── topic1\_1-0

│   ├── 00000000000000000000.index

│   └── 00000000000000000000.log

├── topic1\_1-1

│   ├── 00000000000000000000.index

│   └── 00000000000000000000.log

├── topic1\_2-0

│   ├── 00000000000000000000.index

│   ├── 00000000000000000000.log

│   ├── 00000000000000027028.index

│   ├── 00000000000000027028.log

│   ├── 00000000000000054056.index

│   ├── 00000000000000054056.log

│   ├── 00000000000000081084.index

│   └── 00000000000000081084.log

└── topic1\_2-1

├── 00000000000000000000.index

├── 00000000000000000000.log

├── 00000000000000027028.index

├── 00000000000000027028.log

├── 00000000000000054056.index

├── 00000000000000054056.log

├── 00000000000000081084.index

├── 00000000000000081084.log

├── 00000000000000108112.index

└── 00000000000000108112.log

消息文件目录可以有多个，以上只列出一个，在列出多个的情况下，kafka会均匀的将partition分散在各个log目录文件中，

data/

topicname-partitionid/

offsetstart1.log (segment)

offsetstart1.index (segment index)

offsetstart2.log (segment)

offsetstart2.index (segment index)

每个segment的文件名都是第一个消息的offset

lock文件,是用来做文件锁用的,当有进程占用时会得到文件锁，防止两个进程对同一目录操作

运行过程会出现一些临时文件

|  |  |  |
| --- | --- | --- |
| 临时文件 | 来源 | 说明 |
| recovery-point-offset-checkpoint.tmp  replication-offset-checkpoint.tmp  cleaner-offset-checkpoint.tmp |  | 每次写到临时文件tmp，然后mv tmp 到源文件，windows下会报错，所以先删除再rename |
| xxx.index.deleted  xxx.log.deleted | Log/index | 对于要删除的log和index文件，先重命名，然后延迟删除 |
| xxx.index.swap | Index | 重建索引时为了不让其它地方使用  所以先命名为swap，重建完再rename回来 |
| .kafka\_cleanshutdown |  | 为了兼容0.8，忽略 |
| xxx.log.cleaned |  | Log clean时使用 |

**2) recovery-point-offset-checkpoint文件格式&**

**replication-offset-checkpoint文件格式**

**cleaner-offset-checkpoint文件格式**

三种格式一致，每次都是覆写文件，recovery文件的topic-partition包含所有的log目录下topic-partition，而replication文件只包含在isr中的topic-partition

* **recovery-point-offset-checkpoint：**

0

5

my-replicated-topic 0 0

topic1\_1 0 0

topic1\_2 0 432448

topic1\_1 1 0

topic1\_2 1 432448

versionId

topic-partition-count

topic1 partition1 last-flushed-offset1

topic2 partition2 last-flushed-offset2

topic3 partition3 last-flushed-offset3

topic4 partition4 last-flushed-offset4

topic5 partition5 last-flushed-offset5

说明：recovery-point-offset-checkpoint用于记录本地完好无损的最后offset

有定时任务定时flush内存里面的消息到磁盘，一般跟系统相关，程序里面可以定义多少消息flush一次，在kafka重启时，会重新对该文件记录的offset以后的消息进行验证(crc验证)，并重做index索引文件，如果遇到数据损坏，将truncate该文件到最后的offset

* **replication-offset-checkpoint**

0

5

my-replicated-topic 0 0

topic1\_1 0 0

topic1\_2 0 432985

topic1\_1 1 0

topic1\_2 1 440728

versionId

topic-partition-count

topic1 partition1 last-highwatermark1

topic2 partition2 last-highwatermark2

topic3 partition3 last-highwatermark3

topic4 partition4 last-highwatermark4

topic5 partition5 last-highwatermark5

说明：记录topic-partition highwatermark，leader和follower都需要记录，leader管理highwatermark，并通过fetch response传递给follower，leader和follower都会定时将highwatermark写入该文件

* **cleaner-offset-checkpoint**

**略**

**2)log文件格式**

* 非压缩格式：
* 格式

8字节offset +

4字节messageSize +

4字节CRC +

1字节magic(现在是0)+

1字节attributes(不压缩0) +

4字节keylength(无key-1) +

K字节key + (用来标识一个message比如用户用uuid来作为key去重)

4字节playloadSize +

V字节playload(比如我们发送”message\_1”，这里的playload就是”message\_1”)

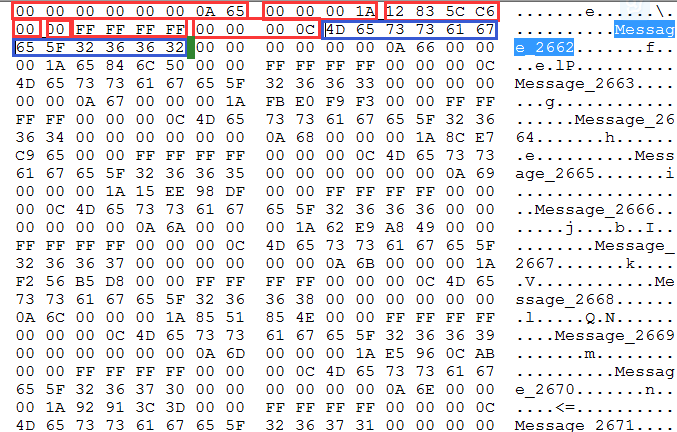
* 例子：

以00000000000000002661.log为例

让我们验证一下

我们拿官网的例子，发送Message\_\*的消息

用Editplus打开，打开16进制视图(16进制视图一个字符代表4bit,两个字符一个字节)



00 00 00 00 00 00 0A 65 --->offset2661

00 00 00 1A --->messageSize即26

12 83 5C C6 --->CRC，用于验证数据完整性(4字节)

00 --->magic num ，0 (1字节)

00 --->attribute,0,未压缩(1字节)

FF FF FF FF --->Key Length,无key，-1 (4字节)

00 00 00 0C --->playload Size,即message\_2662的字节数,即12(4字节)

4D 65 73 73 61 6765 5F 32 36 36 32 ---> message\_2662 (12字节)

* 压缩格式：

**TODO:**

**3) index文件格式**

* 格式

4 byte relative-offset +

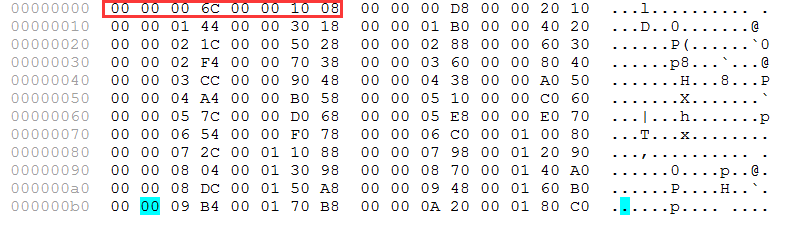
4 byte position

Index文件格式比较简单，每项是4字节的相对位移offset+4字节消息在log文件位置

默认对于log文件每隔4096字节，记录一项，

相对位移offset：比如00000000000000002661.log的第一条假如被记录到index文件中，那么第一条的相对位移offset就是0

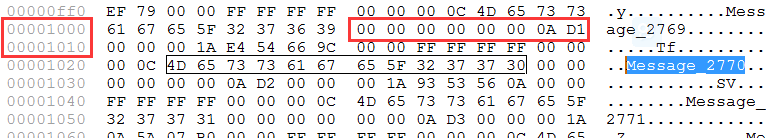
* 例子



00 00 00 6C 是相对offset，即108，加上2661是2769，换算成16进制是0A D1

00 00 10 08 是消息位置

让我们去00000000000000002661.log验证一下



**4）应用日志文件说明**

[root@xxxxxxxxkafka]# tree logs

logs

├── controller.log

├── log-cleaner.log

├── kafka-request.log

├── kafkaServer-gc.log

├── kafkaServer.out

├── server.log

├── state-change.log

controller.logs记录controller的运行日志

kafka-request.log 一直没打出过日志

log-cleaner.log记录log cleaner日志，开启cleaner enable配置才会有

kafkaServer-gc.log 打印kafka gc日志

kafkaServer.out 作为重定向日志包含所有标准输出和错误输出日志，其实我觉得有错误输出即可

server.log记录broker的程序运行日志，这是最主要的日志

state-change.log记录leaderAndIsr相关的操作日志，包括发isr请求，make leader make follower、start fetch、stop fetch等

### kakfa架构设计



1、每个kafka集群都会选择一个controller，controller负责向各个broker(包括自己)发送leaderIsr信息和元数据信息，管理partition和replica状态，broker接收到leaderIsr请求会将自己变成某个topic-partition元组的leader或者follower

2、follower向leader发送fetch请求，来完成数据同步

3、controller和broker监听znode上面的数据，来完成很多操作，比如重选leader，重分配replica，下线replica，修改topic log配置等

4、producer向leader发送produce请求，每隔10分钟抓一次metadata数据，然后根据metadata中的leader信息，随机选取一个partition进行发送produce请求

5、consumer向leader发送fetch请求，同一个group的consumer会自动分配partition，如图所示每个consumer被分配了两个partition，consumer将一些信息比如offsets、owner放到zk上面管理

6、上例中partition为4，replica-factor为2，kafka倾向于将leader平均分布在各个broker上。副本数不得超过可用服务器数，如果超过了会报错，如果在平均分配的情况下副本数-1是kafka集群能允许最多宕掉的服务器数

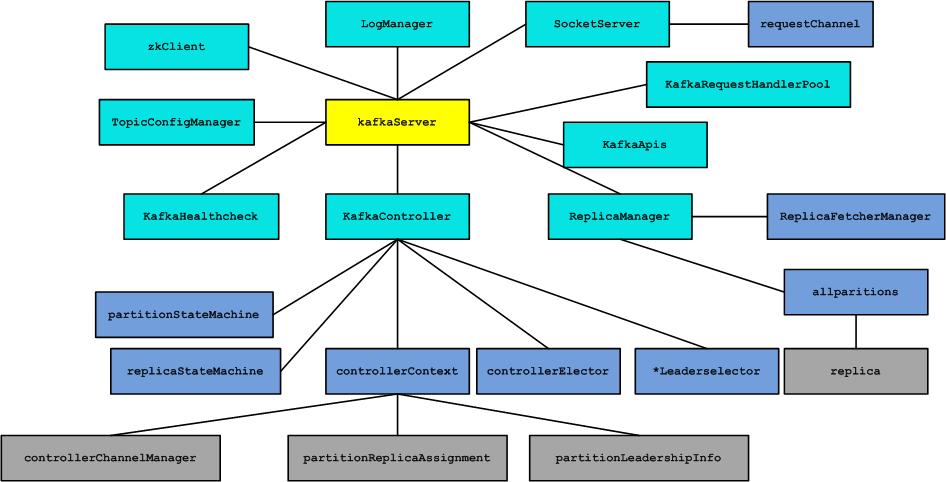
7、leader承受了所有的fetch请求(来自client或者follower)、produce请求

# 流程篇

### kafka启动过程



* **图说明：**红色填充代表其它过程，蓝色代表重要信息
* **简要组件图**

****

* **点评**

1. zkClient用于与zkServer通信的客户端，用于注册当前server、监听节点、创建节点、读取节点、存放数据、竞争controller等用
2. socketServer启动ServerSocket，监听远程连接，接收来自controller以及producer、consumer的请求，controller自己也会使用它来接收controlleredshutdown请求
3. kafkaRequestHandler和kafkapis及socketServer作为请求接收处理的三叉戟，最终交给replicaManager及kafkaController处理，kafkacontroller与kafkaapis的合作只在处理controlleredshutdown请求时出现
4. topicConfigManager除了做监听/config/changes节点之外没有什么太多功能
5. kafkaHealthCheck就是将自己的broker注册到zk上也没有太多可以着墨的地方
6. logManager是非常重要的一个组件，管理着日志文件，包括搜索定位、清理、读写、管理、创建索引等
7. **kafkaController**和**replicaManager**是最重要的两个组件，

前者只有controller broker才会用到，kafkaController下面有几个子组件，

其中controllerContext存放了zk上重要状态数据信息(leader\isr\replica\epoch等)，并通过controllerChannelManager可以对外发请求;

partitionReplicaAssignment存放(topic-partition->replica)

partitionLeadershipInfo存放(topic-partition->leaderAndIsr)

controllerElector用于启动时和/controller节点失效时参与选举controller;

partitionStateMachine管理着topic-partition的状态并通过不同的\*Leaderselector选主，会发出leaderisr和updatemetadata请求

replicaStateMachine用于管理topic-partition-replicaid的状态，会发出ISR和updatemetadata请求

1. replicaManager管理着本地的fetch同步线程，如果是某些topic-partition的leader会管理highwatermark，管理isr

### 日志初始化和清理过程

* **点评**
* 启动过程：

1加载过程----遍历所有data目录，loadlogs扫描所有topic-partition目录，将其加入logmanager，每个tp-log目录，将logsegment load到log里面，index文件通过mmap映射到内存

2恢复过程----加载logsegment的时候会对recovery-checkpoint只有的log进

行恢复，因为这些消息有可能是数据损坏的，重建这些log对应的index文件，然后对这些log里面的消息一一进行CRC验证，遇到损坏的，就截断后续的所有日志

3其它----

log会记录最后一条message的 offset，寻找过程是先找index文件最后一条entry(每个entry8字节，所以很好定位)，然后找到log文件的这条message，然后得到messageset，遍历到最后一条log，得到offset

4开启定时flush操作，将最后一条消息的offset写入recovery-checkpoint文件

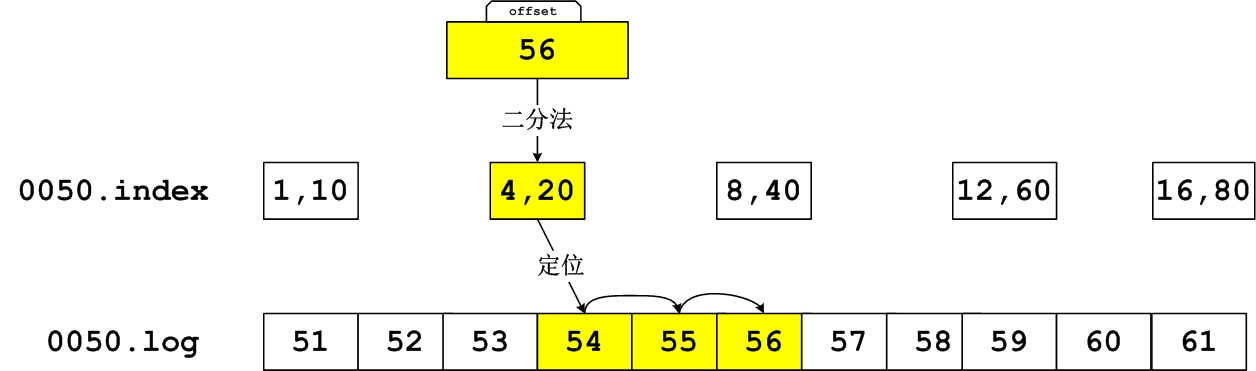
* 清理过程：

一般根据config里面的配置，对log或index文件大小达到阀值进行滚动、对t-p目录总大小进行控制、对过期logsegment的清理

* 读写过程：

通过offset找message的过程如下：

通过offset---->文件名得到开始log文件(floorEntry)--->从index文件找到对应offset的floorEntry(二分法查找)---->从index文件得到的entry(offset,position)去log文件遍历读取,直到读到末尾



写过程：

Append到最后log文件一条，如果达到间隔数，index文件也append(offset\position)，如果文件超大就生成个新的

### 选举controller过程

选controller的过程是在kafkaController这个组件里面做的

先是抢占/controller节点如果发现controllerid是自己，就等一段时间重试，如果别人已经抢占了，说明controller已经了，就返回。

如果竞选controller成功了，会回调kafkacontroller的onControllerFailover方法。

onControllerFailover做了很多事情：初始化controllerContext，选择所有的leader，发送ISR和metadata请求给各个broker，注册各种listener到zk，触发preferred 选举，reassignment，delete topic动作，并开启自动负载均衡检测

当session消息或者controller shutdown时，它会调用onControllerResignation方法

将partitionState和replicaState状态清理掉，controllerChannel关掉，自动负载均衡线程关掉

### controller处理broker startup过程



* 启动一台broker，对controller进行debug
* Zk上/brokers/ids/[brokerid]会消失，zkclient触发ChildChange事件

### controller处理broker failure过程



* kill一台broker，对controller进行debug
* Zk上/brokers/ids/[brokerid]会出现，zkclient触发ChildChange事件

### broker成leader、follower过程



* Controller只发送给broker updateMetaDataRequest，stopReplicaRequest，leaderAndIsrRequest，leaderAndIsrRequest会触发broker 做makeleader或makefollower操作，updateMetaData只会更新kafkaApis的metaCache，供给producer或者consumer做metadata请求使用。由于每次leaderAndIsr变更都会发送一次metadata，所以两者数据会保持同步，但是目前kafka并没有做两个请求的回调操作(只有在删除topic过程接收stopReplicaReponse时才有回调)，不知道发请求是否成功接收，所以应该是kafka的一个缺陷。所以这块要做好监控
* makeLeader过程点评

leader的作用除了接收produce和consume请求，还有一点就是管理ISR以及highwatermark。而makeLeader过程就是为了开启leader的这些功能准备的，首先它要根据topic-partition创建(如果没有)message log目录，然后将自己的endlogoffset作为highwatermark，开启定期检测isr follower是否脱离isr(长时间未发fetch或者落后leader logendoffset太多)。

* makeFollower过程点评

makeFollower的过程比makeLeader的过程要复杂，刚才说了，leader管理ISR和highwatermark(可以看概念说明那节)，那么highwatermark对于Follower可见吗？当然Follower发送fetch请求时会将自身endlogoffset带过去，而返回结果中会有leader返回的

highwatermark。



**为什么要有highwatermark？**

答：看上图，假设某个topic-partition(比如topic1的partition0)的replica list分配在4台机器上，A,B,C,D,produce端设置的ack为1，也就是只要leader 接收处理message成功就返回成功，那么这时replica list的endlogoffset会出现分化。

A作为leader肯定是endlogoffset最高，B紧随其后，C机器由于配置比较低，同步较慢，D机器配置最低，已经被A移除了ISR。

假设这个时候某几个机器出现故障，比如A,C宕机，这时B会成为leader，假如没有highwatermark，在A重启时的时候会做makeFollower操作，在宕机时log文件之后直接追加message，而假如B机器的endlogoffset已经达到A的endlogoffset，会产生数据不一致的情况，所以使用highwatermark来避免这种情况。

在A 做makeFollower操作时，将log文件truncate到highwatermark位置，以防止发生数据不一致情况发生。

还有一种情形会导致数据不一致，那就是unclean leader election，ABC机器都宕机的情况，D机器已经启动，controller会将D作为leader，很明显即便有了highwatermark，也会发生数据不一致，同样消息数据也会丢失。目前kafka 0.8.1.1的版本，没有将unclean election 开关开放给用户，所以这块要做好监控

### produce过程

* 该过程介绍produce发送message过程，leader处理Follower同步message请求过程,leader处理produce请求过程



* 以上图描述了producer发送produce请求到leader，Follower发送fetch请求到

Leader同步消息的过程，这两个请求都是使用长polling机制，当满足条件时才返回，

Fetch请求满足的条件是获取消息字节数达到参数minbytes指定的值，而produce请求满足的条件是同步到消息的replica数量达到ack num指定的数量。

这样做有诸多好处，减少leader的负载，事件通知机制，减少额外的消息接收发送及处理的开销，减少网络带宽，而kafka在长polling基础上做了一次创新，就是两个请求都是长polling，两者互相进行trigger验证是否满足条件的动作，由于produce请求会带来消息这样fetch请求就有可能满足条件返回给Follower，而由于新的fetch请求带来的startoffset已经大于produce请求中最后一个消息offset这表明该Follower已经得到消息同步，即ack数加1。

* produce代码



### 新建topic-partition过程



* **Partition、state的状态转换**
* **partitionState**

|  |  |
| --- | --- |
| **NonExistentPartition** | 这个状态代表该topic-partition被删除了，或者压根没有创建过 |
| **NewPartition** | 当topic-partition刚创建时还没有进行leader选举和isr分配，就处于这个状态 |
| **OnlinePartition** | 一个存活leader被选举了，就处于这个状态 |
| **OfflinePartition** | 当一个leader死掉时，就进入offlinePartition这个状态 |

|  |  |
| --- | --- |
| **NonExistentPartition -> NewPartition** | 新创建topic-partition时触发 |
| **NewPartition -> OnlinePartition** | 分配leader、follower、isr时触发 |
| **OnlinePartition,OfflinePartition**  **-> OnlinePartition** | Offlinepartition->onlinepartition  无非是leader死后又重新选举好，或者在所有broker都重启时触发  Onlinepartitoin->onlinepartition  是在做reassign、preferred eleaction、  Controller shutdown时候触发的 |
| **NewPartition,OnlinePartition -> OfflinePartition** | 做delete topic时触发 |
| **OfflinePartition -> NonExistentPartition** | Delete topic成功后触发 |

* **ReplicaStateChange**

程序里Replica概念是在topic-partition维度下的

|  |  |
| --- | --- |
| **NewReplica** | 当topic-partition创建时，处于zk replica list里面的replica会置为这个状态 |
| **OnlineReplica** | 当发送完leaderandisr之后，replica进入这个状态 |
| **OfflineReplica** | 当replica死掉时，broker节点宕机后，  该replica处于这个状态 |
| **NonExistentReplica** | 当一个replica被移除时，它处于这个状态 |

|  |  |
| --- | --- |
| **NonExistentReplica --> NewReplica** |  |
| **NewReplica-> OnlineReplica** |  |
| **OnlineReplica,OfflineReplica -> OnlineReplica** |  |
| **NewReplica,OnlineReplica -> OfflineReplica** | 当broker死掉时 |
| **OfflineReplica -> NonExistentReplica** | 当一个replica被移除时 |

* 触发partition、replica状态转换的事件列表

自己体会吧

1. **broker startup**
2. **broker shut down**
3. **topic-partition 创建**
4. **alter topic 重置 partition**
5. **preferred election (包括设置了自动负载均衡配置)**
6. **reassignment**
7. **controller shutdown**
8. **delete topic**

### consume过程



* 这里只介绍high-level-consume过程，分为五个类型线程：

1. watcher-executor线程，负责监控rebalance状态，是否需要启动rebalance
2. leader-find-thread，负责处理noleaderpartition，重新获取元数据，关闭该partition老的fetcher线程和空闲，创建新的fetcher线程
3. fetcher-thread负责向leader发fetch请求，获取message，每次获取将fetch offset更新，以供下次fetch请求的使用。除此之外，将FetchedDataChunk放入消费。
4. auto-commit-thread，定时任务，定期将consume offset提交到zk上，来让下次consumer重启时获取最后一次commit-offset来继续消费，0.8.2版本对此做了很好的改进，将consume offset作为一个topic，让consumer作为producer 提交consume-offset
5. consume-thread，处理消息的线程，调用hasnext方法从队列中取出FetchedDataChunk，然后继续从FetchedDataChunk里面的消息进行嵌套迭代。

FetchedDataChunk里面有多条消息，当消息消费完后，会再次从队列中take 新的

FetchedDataChunk。Next方法更新consume offset，这个时候有可能消息还没有处理完。因为默认提交间隔是1分钟，假如已经消费了的消息，还没有提交就已经宕机了，会造成下次消费时重复消费者1分钟内的消息

* high-level-consume会进行自动负载均衡，当新的consumers进入、新的partition被创建，都会触发重新进行负载均衡的动作。
* high-level-consume会自动hand-out-range-fetch，获取log文件最后一条消息进行消费。
* 一个消费线程就是一个consumer，所以不要创建大于partition的数量，否则什么消息都得不到

### controlled shutdown过程

* controlled shutdown是broker手动关闭或者调用钩子回调方法去关闭

步骤如下

1. 钩子方法回调或者执行stop脚本
2. Broker发送controlledshutdownrequest到controller
3. 如果该broker是某个topic-partition的leader，Controller通过controlledshutdownselector重新选举leader，并发送leaderAndIsrRequest给各个broker，如果该broker是follower，发送stopReplica请求到该follower
4. broker处理stopReplica请求(如果还没有shutdown完成)，关闭fetch线程

### preferred election过程

* controller的PreferredReplicaElectionListener，监听

/admin/preferred\_replica\_election，调用onPreferredReplicaElection，使用preferredReplicaPartitionLeaderSelector选举leader，然后发送leaderAndISr请求给各个相关broker

### reassignment过程

* 代码注释最能解释这一过程，确实有点复杂

For example, if OAR = {1, 2, 3} and RAR = {4,5,6}, the values in the assigned replica (AR) and leader/isr path in ZK may go through the following transition.

AR leader/isr

{1,2,3} 1/{1,2,3} (initial state)

{1,2,3,4,5,6} 1/{1,2,3} (step 2)

{1,2,3,4,5,6} 1/{1,2,3,4,5,6} (step 4)

{1,2,3,4,5,6} 4/{1,2,3,4,5,6} (step 7)

{1,2,3,4,5,6} 4/{4,5,6} (step 8)

{4,5,6} 4/{4,5,6} (step 10)

### topic config change过程

* 所有broker都会注册ConfigChangeListener，监听

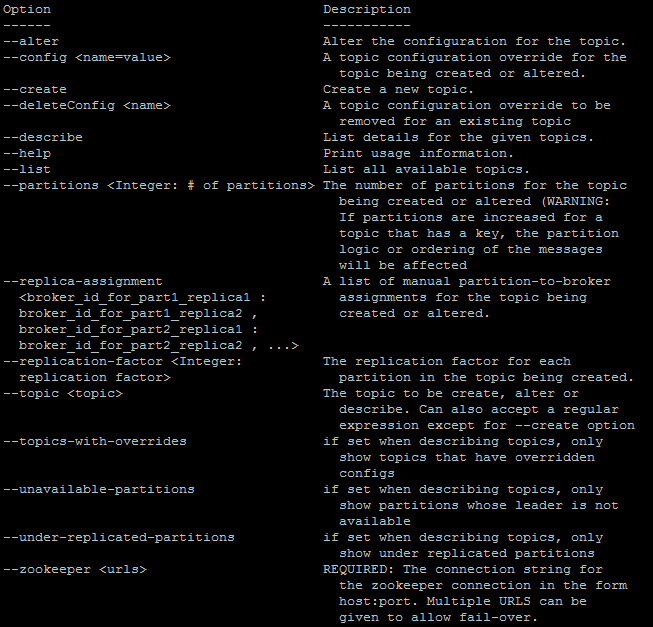
/config/changes,更新log的config，因为/config/changes只会修改log的config

# 工具篇

Kafka提供了不少的工具，利用这些工具，我们可以修改topic的partition数量、replicafactor、可以去迁移topic，可以dump log文件，可以查询leader状态、消费状态等

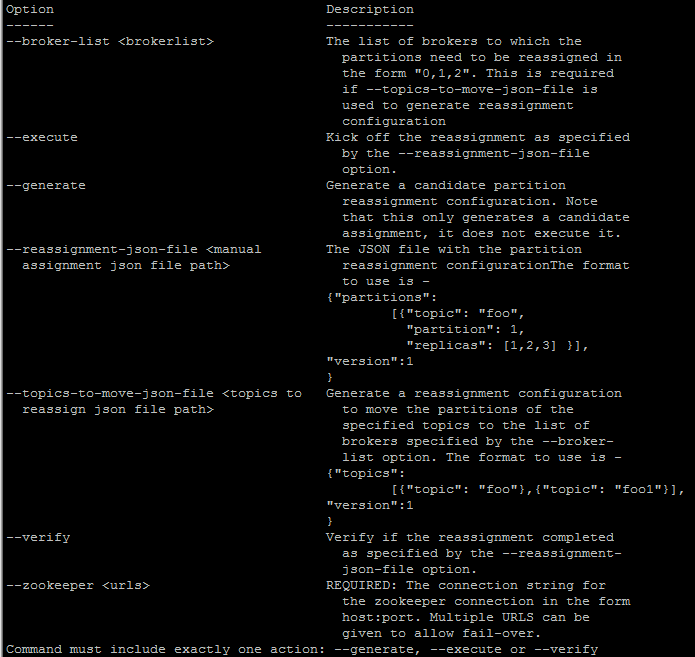
* **kafka-topic.sh**

通过该脚本可以创建、修改topic(不能修改replica 数量)、列出topic



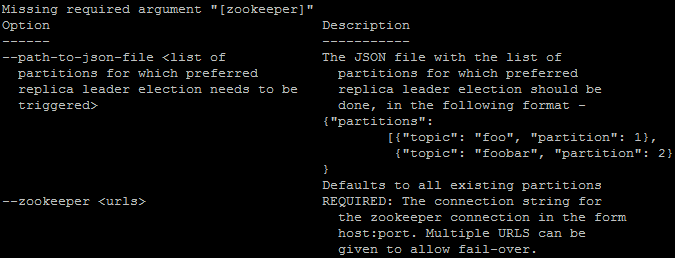
* **kafka-reassign-partitions.sh**

利用该脚本可以迁移topic，同时可以修改topic 的副本数量



* **kafka-preferred-replica-election.sh**

利用该脚本可以进行preferred选举，每个replica lsit选择第一个replica作为leader



* **bin/kafka-run-class.sh kafka.tools.ConsumerOffsetChecker**

查看消费状态：

示例：

[root@AY140722231919412730Z bin]# ./kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --group group --broker-info

Group Topic Pid Offset logSize Lag Owner

SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".

SLF4J: Defaulting to no-operation (NOP) logger implementation

SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.

group topic1\_1 0 0 0 0 none

group topic1\_1 1 0 0 0 none

group topic1\_2 0 454014 454014 0 group\_Cxxxxx-1xxxx17203-16b936a5-0

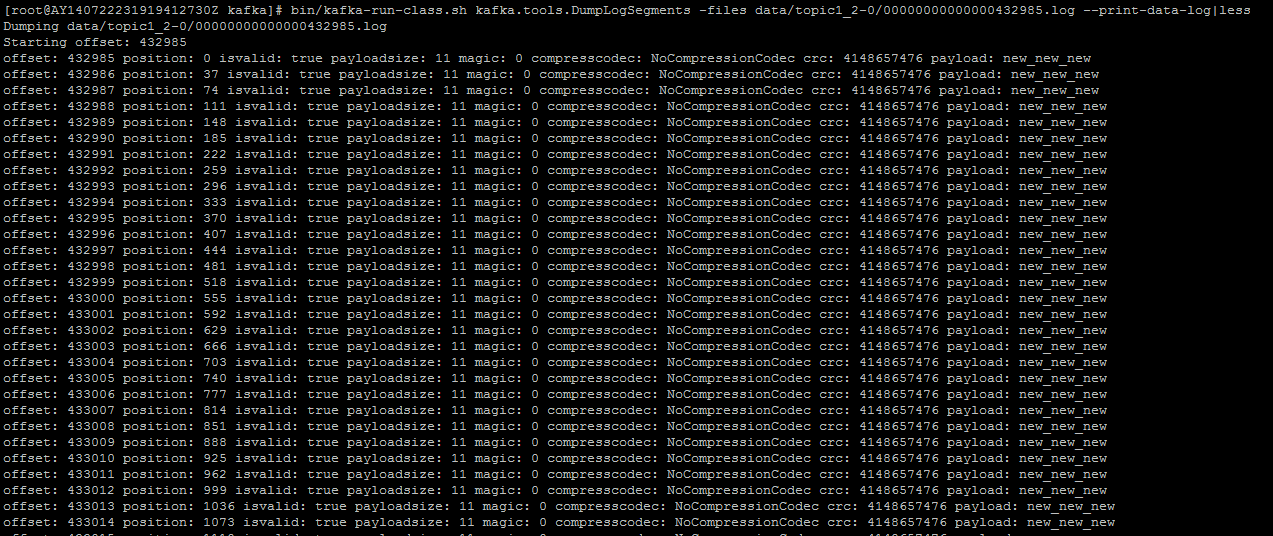
group topic1\_2 1 484775 484775 0 group\_Cxxxxx-1xxxx17203-16b936a5-1

BROKER INFO

1 -> 1xx.xx.1xx.63:9092

* **bin/kafka-run-class.sh kafka.tools.DumpLogSegments**

将log文件dump出来，最有用的工具



* **bin/kafka-run-class.sh kafka.tools.ExportZkOffsets**

将consume的zk信息dump出来

[root@AY140722231919412730Z bin]# ./kafka-run-class.sh kafka.tools.ExportZkOffsets -group group --zkconnect localhost:2181 --output-file ok.txt

[root@AY140722231919412730Z bin]# more ok.txt

/consumers/group/offsets/topic1\_2/1:484775

/consumers/group/offsets/topic1\_2/0:454014

/consumers/group/offsets/topic1\_1/1:0

/consumers/group/offsets/topic1\_1/0:0

* **bin/kafka-run-class.sh kafka.tools.MirrorMaker**

镜像工具

* **bin/kafka-run-class.sh kafka.tools.VerifyConsumerRebalance**

rebalance验证工具

# FAQ

1. **kafka会丢消息吗，如何避免？**

答：kafka能保证at least once，也就是会重复消息，但是可以保证不丢消息；

但是不丢消息需要也是需要client端和server端进行合理配置才可以。

第一：需要在produce端设置request.ack=-1，即每次消息发送需要所有replica确认接收到了。假如设置为1或者N，这样只有leader或者只有N台机器确认接收，当这时leader宕机时，还未同步的消息会丢失。

第二：不能出现unclean election，即当leader宕机并且所有isr都宕机，这时选择了一个不在isr列表里面的replica，这种情况会丢消息也会导致各个replica的数据的不一致。

第三:consume client需要修改，由于client端只要调用consumeIterator.next即会更新consume offset，这时如果没有处理成功也置为已经消费了，就会造成丢失消息的情况。这一点只有client 捕获处理失败的异常，保存该次处理失败的消息，或者打日志后续再进行消费

第四：不要用异步消息

**2、kafka会导致重复消息吗？如何避免？**

答：kafka不能保证消息不重复，如下情况会发生重复消息

1、broker在接收到produce请求，已经将message成功写入到日志之后，发生了异常或者等待其它replica同步消息超时，这个时候producer client会重新发送刚才已经写入leader的消息---发生重复消息.

2、consume client在消费消息时，定期提交消费offset到zk，已经消费的消息还未及时提交到zk，这时下次consumer client再次启动时从最后一个consume offset开始消费，就会出现重复消费的情况

消息重复不容易去避免，官方建议每个消息带上一个唯一message key(比如uuid)，consume在消费的时候进行过滤，或者将消费之后产生的数据在清洗任务的时候滤重。Kafka提供一个log compaction可以进行滤重，但是不能完全避免重复，原因是log compaction也是定时执行的任务。

不知道以后kafka会不会提供滤重消息的策略

**3、kafka有哪些陷阱和bug**

答：kafka目前还不能算作很成熟的消息中间件。

1. kafka不能保证消息重复，kafka一方面保证性能高性能高可用性，如果再去保证消息不重复有点困难
2. consumer client依赖zk提交consume offset，增加耦合性，一旦zk出现不稳定，就会影响consume 的正常消费，长时间提交不了zk就会造成重复消息的发生。0.8.2版本已经对其进行改进，将consume offset作为一个topic供client提交
3. kafka的unclean election不能让用户设置，出现了丢消息都不知道！
4. Kafka目前有很多严重bug，出现了这些错误无法有效通知给使用者，比如出现无controller的情况，[https://issues.apache.org/jira/browse/KAFKA-1451](qq://txfile/)，官方说已经修复了，但是还是会出现。可以模拟重现出来
5. controller发送leaderAndIsr给各个broker，却没有回调方法对来自各个broker响应进行处理，假如broker执行make leader或者make follower失败，controller得不到通知，造成无leader状态
6. consume rebalance出错超过一定次数就会进入假死状态

**4、有些时间窗口会不会导致消息丢失或者状态不一致情况，比如选举leader的时候，选择controller的时候，consume rebalance的时候？**

答：不会

1. produce在推送消息时，遇到错误比如leader改变，会等待一段时间重新获取topic元数据，这个间隔时间至关重要，一般设置为略大于zk session time的时间，这样能保证充足时间让controller重新选leader或者controller选举
2. consume在拉取消息时，也会出现错误，这时也会间隔一段时间获取topic元数据，进行重试，在rebalance出错抢占owner节点也有出错的时候，也停留backoff时间进行重试。
3. 这些backoff时间保证了集群进行恢复，但会将produce过程或者消费过程停止，造成消息积压
4. 除了produce端和consume端的停留backoff重试机制，其它比如broker在controlled shutdown也有重试出错停留backoff时长的机制

# 监控篇

* 基于FAQ里面的kafka有可能出现问题，我们一定要做好kafka的监控：包括controller、各个topic-partition的leader、各个topic-partition的consumer、lagSize、消费offset、各个partition的owner。

防止无controller状态，无leader状态，无对应consumer状态，isr衰减情况，各个isr的logendoffset，consume的lagSize等等

现在没有一款好的监控工具能监控这么全面

* Kafka-web-console

<https://github.com/claudemamo/kafka-web-console>

* Kafka-offset-monitor

通过zk获取数据展现<http://quantifind.com/KafkaOffsetMonitor/>

所以lagSize会比实际的大

* Kafka-manager

雅虎开源的kafka监控工具

<https://github.com/yahoo/kafka-manager>