|  |  |
| --- | --- |
| 参数 | 说明(解释) |
| broker.id =0 | 每一个broker在集群中的唯一表示，要求是正数。当该服务器的IP地址发生改变时，broker.id没有变化，则不会影响consumers的消息情况 |
| log.dirs=/data/kafka-logs | kafka数据的存放地址，多个地址的话用逗号分割/data/kafka-logs-1，/data/kafka-logs-2 |
| port =9092 | broker   server服务端口 |
| message.max.bytes   =6525000 | 表示消息体的最大大小，单位是字节 |
| num.network.threads   =4 | broker处理消息的最大线程数，一般情况下不需要去修改 |
| num.io.threads =8 | broker处理磁盘IO的线程数，数值应该大于你的硬盘数 |
| background.threads   =4 | 一些后台任务处理的线程数，例如过期消息文件的删除等，一般情况下不需要去做修改 |
| queued.max.requests   =500 | 等待IO线程处理的请求队列最大数，若是等待IO的请求超过这个数值，那么会停止接受外部消息，应该是一种自我保护机制。 |
| host.name | broker的主机地址，若是设置了，那么会绑定到这个地址上，若是没有，会绑定到所有的接口上，并将其中之一发送到ZK，一般不设置 |
| socket.send.buffer.bytes=100\*1024 | socket的发送缓冲区，socket的调优参数SO\_SNDBUFF |
| socket.receive.buffer.bytes   =100\*1024 | socket的接受缓冲区，socket的调优参数SO\_RCVBUFF |
| socket.request.max.bytes   =100\*1024\*1024 | socket请求的最大数值，防止serverOOM，message.max.bytes必然要小于socket.request.max.bytes，会被topic创建时的指定参数覆盖 |
| log.segment.bytes   =1024\*1024\*1024 | topic的分区是以一堆segment文件存储的，这个控制每个segment的大小，会被topic创建时的指定参数覆盖 |
| log.roll.hours =24\*7 | 这个参数会在日志segment没有达到log.segment.bytes设置的大小，也会强制新建一个segment会被 topic创建时的指定参数覆盖 |
| log.cleanup.policy   = delete | 日志清理策略选择有：delete和compact主要针对过期数据的处理，或是日志文件达到限制的额度，会被 topic创建时的指定参数覆盖 |
| log.retention.minutes=3days | 数据存储的最大时间超过这个时间会根据log.cleanup.policy设置的策略处理数据，也就是消费端能够多久去消费数据  **log.retention.bytes**和log.retention.minutes任意一个达到要求，都会执行删除，会被topic创建时的指定参数覆盖 |
| **log.retention.bytes**=-1 | topic每个分区的最大文件大小，一个topic的大小限制 =分区数\***log.retention.bytes**。-1没有大小限**log.retention.bytes**和log.retention.minutes任意一个达到要求，都会执行删除，会被topic创建时的指定参数覆盖 |
| log.retention.check.interval.ms=5minutes | 文件大小检查的周期时间，是否处罚 log.cleanup.policy中设置的策略 |
| log.cleaner.enable=**false** | 是否开启日志压缩 |
| log.cleaner.threads   = 2 | 日志压缩运行的线程数 |
| log.cleaner.io.max.bytes.per.second=None | 日志压缩时候处理的最大大小 |
| log.cleaner.dedupe.buffer.size=500\*1024\*1024 | 日志压缩去重时候的缓存空间，在空间允许的情况下，越大越好 |
| log.cleaner.io.buffer.size=512\*1024 | 日志清理时候用到的IO块大小一般不需要修改 |
| log.cleaner.io.buffer.load.factor   =0.9 | 日志清理中hash表的扩大因子一般不需要修改 |
| log.cleaner.backoff.ms   =15000 | 检查是否处罚日志清理的间隔 |
| log.cleaner.min.cleanable.ratio=0.5 | 日志清理的频率控制，越大意味着更高效的清理，同时会存在一些空间上的浪费，会被topic创建时的指定参数覆盖 |
| log.cleaner.delete.retention.ms   =1day | 对于压缩的日志保留的最长时间，也是客户端消费消息的最长时间，同log.retention.minutes的区别在于一个控制未压缩数据，一个控制压缩后的数据。会被topic创建时的指定参数覆盖 |
| log.index.size.max.bytes   =10\*1024\*1024 | 对于segment日志的索引文件大小限制，会被topic创建时的指定参数覆盖 |
| log.index.interval.bytes   =4096 | 当执行一个fetch操作后，需要一定的空间来扫描最近的offset大小，设置越大，代表扫描速度越快，但是也更好内存，一般情况下不需要搭理这个参数 |
| log.flush.interval.messages=None | log文件”sync”到磁盘之前累积的消息条数,因为磁盘IO操作是一个慢操作,但又是一个”数据可靠性"的必要手段,所以此参数的设置,需要在**"数据可靠性"**与"性能"之间做必要的权衡.如果此值过大,将会导致每次"fsync"的时间较长(IO阻塞),如果此值过小,将会导致**"fsync"**的次数较多,这也意味着整体的client请求有一定的延迟.物理server故障,将会导致没有fsync的消息丢失. |
| log.flush.scheduler.interval.ms   =3000 | 检查是否需要固化到硬盘的时间间隔 |
| log.flush.interval.ms   = None | 仅仅通过interval来控制消息的磁盘写入时机,是不足的.此参数用于控制**"fsync"**的时间间隔,如果消息量始终没有达到阀值,但是离上一次磁盘同步的时间间隔达到阀值,也将触发. |
| log.delete.delay.ms   =60000 | 文件在索引中清除后保留的时间一般不需要去修改 |
| log.flush.offset.checkpoint.interval.ms   =60000 | 控制上次固化硬盘的时间点，以便于数据恢复一般不需要去修改 |
| auto.create.topics.enable   =**true** | 是否允许自动创建topic，若是**false**，就需要通过命令创建topic |
| **default**.replication.factor =1 | 是否允许自动创建topic，若是**false**，就需要通过命令创建topic |
| num.partitions =1 | 每个topic的分区个数，若是在topic创建时候没有指定的话会被topic创建时的指定参数覆盖 |
|  |  |
| 以下是kafka中Leader,replicas配置参数 |  |
| controller.socket.timeout.ms   =30000 | partition leader与replicas之间通讯时,socket的超时时间 |
| controller.message.queue.size=10 | partition leader与replicas数据同步时,消息的队列尺寸 |
| replica.lag.time.max.ms   =10000 | replicas响应partition leader的最长等待时间，若是超过这个时间，就将replicas列入ISR(in-sync replicas)，并认为它是死的，不会再加入管理中 |
| replica.lag.max.messages   =4000 | 如果follower落后与leader太多,将会认为此follower[或者说partition relicas]已经失效  ##通常,在follower与leader通讯时,因为网络延迟或者链接断开,总会导致replicas中消息同步滞后  ##如果消息之后太多,leader将认为此follower网络延迟较大或者消息吞吐能力有限,将会把此replicas迁移  ##到其他follower中.  ##在broker数量较少,或者网络不足的环境中,建议提高此值. |
| replica.socket.timeout.ms=30\*1000 | follower与leader之间的socket超时时间 |
| replica.socket.receive.buffer.bytes=64\*1024 | leader复制时候的socket缓存大小 |
| replica.fetch.max.bytes   =1024\*1024 | replicas每次获取数据的最大大小 |
| replica.fetch.wait.max.ms   =500 | replicas同leader之间通信的最大等待时间，失败了会重试 |
| replica.fetch.min.bytes   =1 | fetch的最小数据尺寸,如果leader中尚未同步的数据不足此值,将会阻塞,直到满足条件 |
| num.replica.fetchers=1 | leader进行复制的线程数，增大这个数值会增加follower的IO |
| replica.high.watermark.checkpoint.interval.ms   =5000 | 每个replica检查是否将最高水位进行固化的频率 |
| controlled.shutdown.enable   =**false** | 是否允许控制器关闭broker ,若是设置为**true**,会关闭所有在这个broker上的leader，并转移到其他broker |
| controlled.shutdown.max.retries   =3 | 控制器关闭的尝试次数 |
| controlled.shutdown.retry.backoff.ms   =5000 | 每次关闭尝试的时间间隔 |
| leader.imbalance.per.broker.percentage   =10 | leader的不平衡比例，若是超过这个数值，会对分区进行重新的平衡 |
| leader.imbalance.check.interval.seconds   =300 | 检查leader是否不平衡的时间间隔 |
| offset.metadata.max.bytes | 客户端保留offset信息的最大空间大小 |
| kafka中zookeeper参数配置 |  |
| zookeeper.connect   = localhost:2181 | zookeeper集群的地址，可以是多个，多个之间用逗号分割hostname1:port1,hostname2:port2,hostname3:port3 |
| zookeeper.session.timeout.ms=6000 | ZooKeeper的最大超时时间，就是心跳的间隔，若是没有反映，那么认为已经死了，不易过大 |
| zookeeper.connection.timeout.ms   =6000 | ZooKeeper的连接超时时间 |
| zookeeper.sync.time.ms   =2000 | ZooKeeper集群中leader和follower之间的同步实际那 |

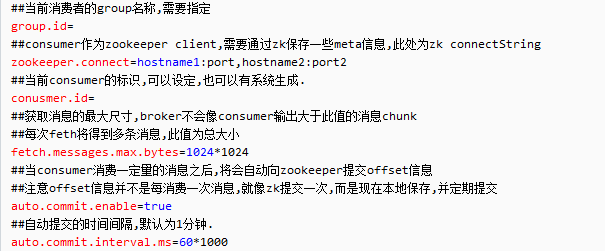
**Broker主要配置**



**Producer 主要配置**



**Consumer 主要配置**



**2.**    **设计原理**

kafka的设计初衷是希望作为一个统一的信息收集平台,能够实时的收集反馈信息,并需要能够支撑较大的数据量,且具备良好的容错能力.

**a、持久性**

    kafka使用文件存储消息,这就直接决定kafka在性能上严重依赖文件系统的本身特性.且无论任何OS下,对文件系统本身的优化几乎没有可能.文件缓存/直接内存映射等是常用的手段.因为kafka是对日志文件进行append操作,因此磁盘检索的开支是较小的;同时为了减少磁盘写入的次数,broker会将消息暂时buffer起来,当消息的个数(或尺寸)达到一定阀值时,再flush到磁盘,这样减少了磁盘IO调用的次数.

**b、性能**

    需要考虑的影响性能点很多,除磁盘IO之外,我们还需要考虑网络IO,这直接关系到kafka的吞吐量问题.kafka并没有提供太多高超的技巧;对于producer端,可以将消息buffer起来,当消息的条数达到一定阀值时,批量发送给broker;对于consumer端也是一样,批量fetch多条消息.不过消息量的大小可以通过配置文件来指定.对于kafka broker端,似乎有个sendfile系统调用可以潜在的提升网络IO的性能:将文件的数据映射到系统内存中,socket直接读取相应的内存区域即可,而无需进程再次copy和交换. 其实对于producer/consumer/broker三者而言,CPU的开支应该都不大,因此启用消息压缩机制是一个良好的策略;压缩需要消耗少量的CPU资源,不过对于kafka而言,网络IO更应该需要考虑.可以将任何在网络上传输的消息都经过压缩.kafka支持gzip/snappy等多种压缩方式.

**c、生产者**

    负载均衡: producer将会和Topic下所有partition leader保持socket连接;消息由producer直接通过socket发送到broker,中间不会经过任何"路由层".事实上,消息被路由到哪个partition上,有producer客户端决定.比如可以采用"random""key-hash""轮询"等,如果一个topic中有多个partitions,那么在producer端实现"消息均衡分发"是必要的.

    其中partition leader的位置(host:port)注册在zookeeper中,producer作为zookeeper client,已经注册了watch用来监听partition leader的变更事件.

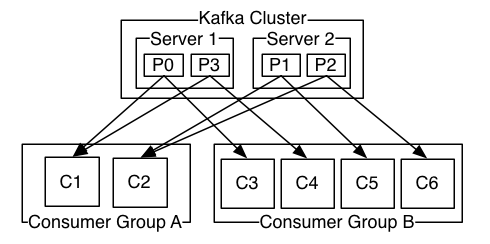
    异步发送：将多条消息暂且在客户端buffer起来，并将他们批量的发送到broker，小数据IO太多，会拖慢整体的网络延迟，批量延迟发送事实上提升了网络效率。不过这也有一定的隐患，比如说当producer失效时，那些尚未发送的消息将会丢失。

**d、消费者**

    consumer端向broker发送"fetch"请求,并告知其获取消息的offset;此后consumer将会获得一定条数的消息;consumer端也可以重置offset来重新消费消息.

    在JMS实现中,Topic模型基于push方式,即broker将消息推送给consumer端.不过在kafka中,采用了pull方式,即consumer在和broker建立连接之后,主动去pull(或者说fetch)消息;这种模式有些优点,首先consumer端可以根据自己的消费能力适时的去fetch消息并处理,且可以控制消息消费的进度(offset);此外,消费者可以良好的控制消息消费的数量,batch fetch.

    其他JMS实现,消息消费的位置是有prodiver保留,以便避免重复发送消息或者将没有消费成功的消息重发等,同时还要控制消息的状态.这就要求JMS broker需要太多额外的工作.在kafka中,partition中的消息只有一个consumer在消费,且不存在消息状态的控制,也没有复杂的消息确认机制,可见kafka broker端是相当轻量级的.当消息被consumer接收之后,consumer可以在本地保存最后消息的offset,并间歇性的向zookeeper注册offset.由此可见,consumer客户端也很轻量级.



**e、消息传送机制**

    对于JMS实现,消息传输担保非常直接:有且只有一次(exactly once).在kafka中稍有不同:

    1) at most once: 最多一次,这个和JMS中"非持久化"消息类似.发送一次,无论成败,将不会重发.

    2) at least once: 消息至少发送一次,如果消息未能接受成功,可能会重发,直到接收成功.

    3) exactly once: 消息只会发送一次.

    at most once: 消费者fetch消息,然后保存offset,然后处理消息;当client保存offset之后,但是在消息处理过程中出现了异常,导致部分消息未能继续处理.那么此后"未处理"的消息将不能被fetch到,这就是"at most once".

    at least once: 消费者fetch消息,然后处理消息,然后保存offset.如果消息处理成功之后,但是在保存offset阶段zookeeper异常导致保存操作未能执行成功,这就导致接下来再次fetch时可能获得上次已经处理过的消息,这就是"at least once"，原因offset没有及时的提交给zookeeper，zookeeper恢复正常还是之前offset状态.

    exactly once: kafka中并没有严格的去实现(基于2阶段提交,事务),我们认为这种策略在kafka中是没有必要的.

    通常情况下"at-least-once"是我们首选.(相比at most once而言,重复接收数据总比丢失数据要好).

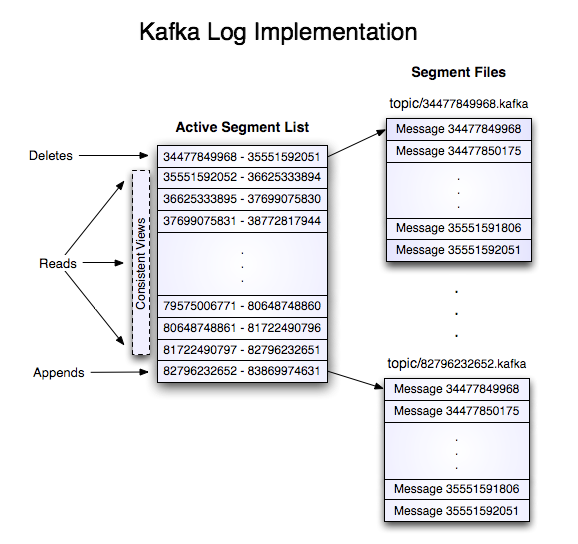
**f、复制备份**

    kafka将每个partition数据复制到多个server上,任何一个partition有一个leader和多个follower(可以没有);备份的个数可以通过broker配置文件来设定.leader处理所有的read-write请求,follower需要和leader保持同步.Follower和consumer一样,消费消息并保存在本地日志中;leader负责跟踪所有的follower状态,如果follower"落后"太多或者失效,leader将会把它从replicas同步列表中删除.当所有的follower都将一条消息保存成功,此消息才被认为是"committed",那么此时consumer才能消费它.即使只有一个replicas实例存活,仍然可以保证消息的正常发送和接收,只要zookeeper集群存活即可.(不同于其他分布式存储,比如hbase需要"多数派"存活才行)

    当leader失效时,需在followers中选取出新的leader,可能此时follower落后于leader,因此需要选择一个"up-to-date"的follower.选择follower时需要兼顾一个问题,就是新leader server上所已经承载的partition leader的个数,如果一个server上有过多的partition leader,意味着此server将承受着更多的IO压力.在选举新leader,需要考虑到"负载均衡".

**g.日志**

    如果一个topic的名称为"my\_topic",它有2个partitions,那么日志将会保存在my\_topic\_0和my\_topic\_1两个目录中;日志文件中保存了一序列"log entries"(日志条目),每个log entry格式为"4个字节的数字N表示消息的长度" + "N个字节的消息内容";每个日志都有一个offset来唯一的标记一条消息,offset的值为8个字节的数字,表示此消息在此partition中所处的起始位置..每个partition在物理存储层面,有多个log file组成(称为segment).segment file的命名为"最小offset".kafka.例如"00000000000.kafka";其中"最小offset"表示此segment中起始消息的offset.



    其中每个partiton中所持有的segments列表信息会存储在zookeeper中.

    当segment文件尺寸达到一定阀值时(可以通过配置文件设定,默认1G),将会创建一个新的文件;当buffer中消息的条数达到阀值时将会触发日志信息flush到日志文件中,同时如果"距离最近一次flush的时间差"达到阀值时,也会触发flush到日志文件.如果broker失效,极有可能会丢失那些尚未flush到文件的消息.因为server意外失败,仍然会导致log文件格式的破坏(文件尾部),那么就要求当server启动时需要检测最后一个segment的文件结构是否合法并进行必要的修复.

    获取消息时,需要指定offset和最大chunk尺寸,offset用来表示消息的起始位置,chunk size用来表示最大获取消息的总长度(间接的表示消息的条数).根据offset,可以找到此消息所在segment文件,然后根据segment的最小offset取差值,得到它在file中的相对位置,直接读取输出即可.

    日志文件的删除策略非常简单:启动一个后台线程定期扫描log file列表,把保存时间超过阀值的文件直接删除(根据文件的创建时间).为了避免删除文件时仍然有read操作(consumer消费),采取copy-on-write方式.

**h、分配**

    kafka使用zookeeper来存储一些meta信息,并使用了zookeeper watch机制来发现meta信息的变更并作出相应的动作(比如consumer失效,触发负载均衡等)

    1) Broker node registry: 当一个kafka broker启动后,首先会向zookeeper注册自己的节点信息(临时znode),同时当broker和zookeeper断开连接时,此znode也会被删除.

    格式: /broker/ids/[0...N]   -->host:port;其中[0..N]表示broker id,每个broker的配置文件中都需要指定一个数字类型的id(全局不可重复),znode的值为此broker的host:port信息.

    2) Broker Topic Registry: 当一个broker启动时,会向zookeeper注册自己持有的topic和partitions信息,仍然是一个临时znode.

    格式: /broker/topics/[topic]/[0...N]  其中[0..N]表示partition索引号.

    3) Consumer and Consumer group: 每个consumer客户端被创建时,会向zookeeper注册自己的信息;此作用主要是为了"负载均衡".

    一个group中的多个consumer可以交错的消费一个topic的所有partitions;简而言之,保证此topic的所有partitions都能被此group所消费,且消费时为了性能考虑,让partition相对均衡的分散到每个consumer上.

    4) Consumer id Registry: 每个consumer都有一个唯一的ID(host:uuid,可以通过配置文件指定,也可以由系统生成),此id用来标记消费者信息.

    格式: /consumers/[group\_id]/ids/[consumer\_id]

    仍然是一个临时的znode,此节点的值为{"topic\_name":#streams...},即表示此consumer目前所消费的topic + partitions列表.

    5) Consumer offset Tracking: 用来跟踪每个consumer目前所消费的partition中最大的offset.

    格式: /consumers/[group\_id]/offsets/[topic]/[broker\_id-partition\_id]-->offset\_value

    此znode为持久节点,可以看出offset跟group\_id有关,以表明当group中一个消费者失效,其他consumer可以继续消费.

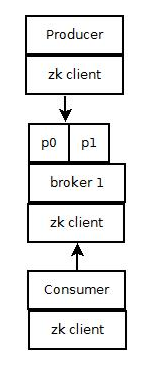
    6) Partition Owner registry: 用来标记partition被哪个consumer消费.临时znode

    格式: /consumers/[group\_id]/owners/[topic]/[broker\_id-partition\_id] -->consumer\_node\_id当consumer启动时,所触发的操作:

    A) 首先进行"Consumer id Registry";

    B) 然后在"Consumer id Registry"节点下注册一个watch用来监听当前group中其他consumer的"leave"和"join";只要此znode path下节点列表变更,都会触发此group下consumer的负载均衡.(比如一个consumer失效,那么其他consumer接管partitions).

    C) 在"Broker id registry"节点下,注册一个watch用来监听broker的存活情况;如果broker列表变更,将会触发所有的groups下的consumer重新balance.



    1) Producer端使用zookeeper用来"发现"broker列表,以及和Topic下每个partition leader建立socket连接并发送消息.

    2) Broker端使用zookeeper用来注册broker信息,已经监测partition leader存活性.

    3) Consumer端使用zookeeper用来注册consumer信息,其中包括consumer消费的partition列表等,同时也用来发现broker列表,并和partition leader建立socket连接,并获取消息.