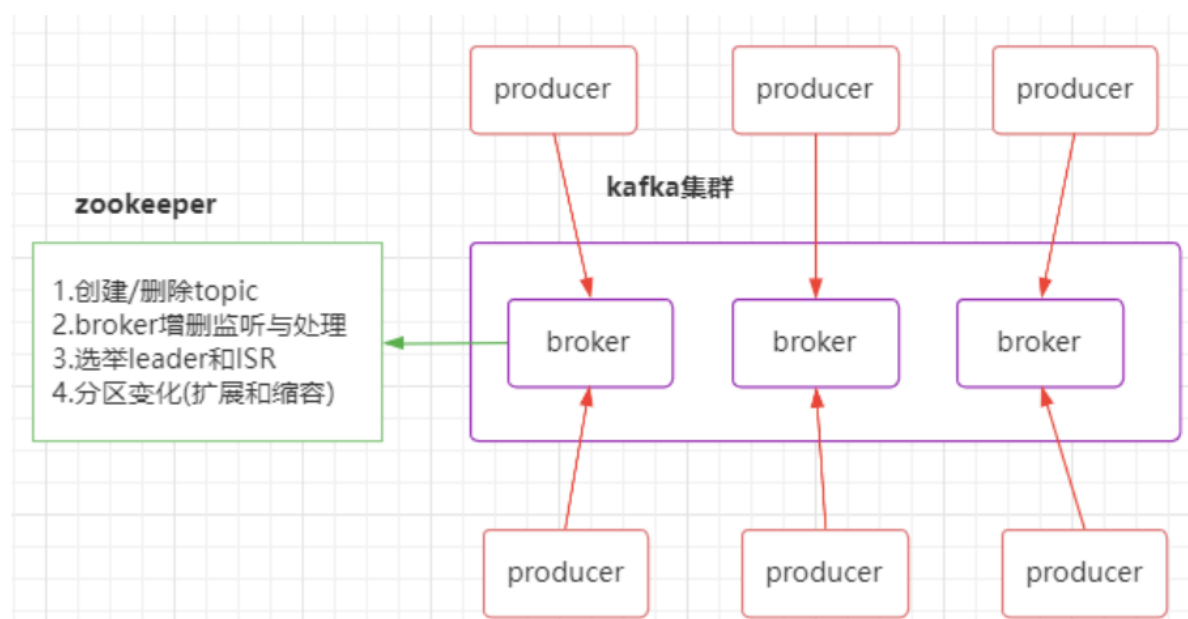


1 集群



1. Kafka架构是由producer（消息生产者）、consumer（消息消费者）、broker(kafka集群的server，负责处理消息读、写请求，存储消息，在kafka cluster这一层这里，其实里面是有很多个broker)、topic（消息队列/分类相当于队列，里面有生产者和消费者模型）、zookeeper 这些部分组成。
2. kafka里面的消息是有topic来组织的，简单的我们可以想象为一个队列，一个队列就是一个topic，然后它把每个topic又分为很多个partition，这个是为了做并行的，在每个partition内部消息强有序，相当于有序的队列，其中每个消息都有个序号offset，比如0到12，从前面读往后面写。一个partition对应一个broker，一个broker可以管多个partition，比如说，topic有6个partition，有两个broker，那每个broker就管3个partition。这个partition可以很简单想象为一个文件，当数据发过来的时候它就往这个partition上面append，追加就行，消息不经过内存缓冲，直接写入文件，kafka和很多消息系统不一样，很多消息系统是消费完了我就把它删掉，而kafka是根据时间策略删除，而不是消费完就删除，在kafka里面没有一个消费完这么个概念，只有过期这样一个概念。
3. producer自己决定往哪个partition里面去写，这里有一些的策略，譬如如果hash，不用多个partition之间去join数据了。consumer自己维护消费到哪个offset，**每个consumer都有对应的group**，group内是queue消费模型（各个consumer消费不同的partition，**因此一个消息在group内只消费一次**），**group间是publish-subscribe消费模型，各个group各自独立消费，互不影响**，因此一个消息在被每个group消费一次。

1.1 搭建两台服务器

ip2: 192.168.31.36

1.2 zookeeper部署

zookeeper还是先只部署一台，在ip2: 192.168.31.36 上启动zookeeper

1.3 启动broker ip1: 192.168.186.138

- 修改broker.id（也可以改为-1，自动分配）

broker.id=0

- 修改server.properties（在config目录），增加zookeeper的配置，

要配置对应的zookeeper ip地址。

```
zookeeper.connect=192.168.186.138:2181
```

- 启动kafka

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

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

默认端口为：9092，可以通过命令**lsof -i:9092**查看kafka是否启动成功。

1.4 启动broker ip2: 192.168.31.36

- 修改broker.id（也可以改为-1，自动分配）

broker.id=1

- 修改server.properties（在config目录），增加zookeeper的配置，

要配置对应的zookeeper ip地址。

```
zookeeper.connect=192.168.186.138:2181
```

- 启动kafka

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

默认端口为：9092，可以通过命令**lsof -i:9092**查看kafka是否启动成功。

1.5 查看kafka集群

创建主题

```
$ sh kafka-topics.sh --create --zookeeper 192.168.186.138:2181 -replication-factor 2 --partitions 2 --topic kafka-2
```

查看主题

```
sh kafka-topics.sh --describe --zookeeper 192.168.186.138:2181 --topic kafka-
```

2 显示信息

```
Topic:kafka-2    PartitionCount:2    ReplicationFactor:2 Configs:
  Topic: kafka-2  Partition: 0    Leader: 1    Replicas: 1,0    Isr: 1,0
  Topic: kafka-2  Partition: 1    Leader: 0    Replicas: 0,1    Isr: 0
```

1.6 测试集群

开三个终端：开启一个生产者，两个消费者

生产者：

```
sh kafka-console-producer.sh --broker-list 192.168.186.138:9092 --topic test_topic
```

消费者：

```
sh kafka-console-consumer.sh --bootstrap-server 192.168.186.138:9092 --topic test_topic --group 0 --from-beginning
```

```
sh kafka-console-consumer.sh --bootstrap-server 192.168.186.138:9092 --topic test_topic --group 0 --from-beginning
```

当两个消费者同属一个消费组开启后，消费者轮流收到发送者的数据。

kafka-console-consumer.sh部分支持的参数：

参数	值类型	说明	有效值
--topic	string	被消费的topic	
-partition	integer	指定分区 除非指定'-offset'，否则从分区结束 (latest)开始消费	
--offset	string	执行消费的起始offset位置 默认值:latest	latest earliest
--consumer-property	string	将用户定义的属性以key=value的形式传递给使用者	
--consumer.config	string	消费者配置属性文件 请注意，[consumer-property]优先于此配置	
--from-beginning		从存在的最早消息开始，而不是从最新消息开始	

--group	string	指定消费者所属组的ID	
---------	--------	-------------	--

1 生产者

代码参考: cplus_kafka/p目录的代码

1.1 客户端开发

正常生产逻辑

- 1.配置生产者客户端参数及创建相应的生产者实例;
- 2.构建待发送的消息;
- 3.发送消息;
- 4.关闭生产者实例;

必要的参数配置

bootstrap.servers

- 指定连接 Kafka 集群所需要的 broker 地址清单。
- 具体的内容格式为 host1:port1,host2:port2, 可以设置一个或者多个地址, 中间以逗号进行隔开, 此参数的默认值为 ""。
- 注意这里并非需要所有的 broker 地址, **因为生产者会从给定的 broker 里查找其他 broker 的信息。**
- 不过建议至少要设置**两个以上的 broker 地址信息**, 当其中任意一个宕机时, 生产者仍然可以连接到 Kafka 集群上。

```
// 创建Kafka Conf对象
m_config = RdKafka::Conf::create(RdKafka::Conf::CONF_GLOBAL);
if (m_config == NULL) {
    std::cout << "Create RdKafka Conf failed." << std::endl;
}
// 创建Topic Conf对象
m_topicConfig = RdKafka::Conf::create(RdKafka::Conf::CONF_TOPIC);
if (m_topicConfig == NULL) {
    std::cout << "Create RdKafka Topic Conf failed." << std::endl;
}
// 设置Broker属性
RdKafka::Conf::ConfResult errCode;
m_dr_cb = new ProducerDeliveryReportCb;
std::string errorStr;
// 传递情况报告
```

```

errCode = m_config->set("dr_cb", m_dr_cb, errorStr);
if (errCode != RdKafka::Conf::CONF_OK) {
    std::cout << "Conf set failed:" << errorStr << std::endl;
}
m_event_cb = new ProducerEventCb;
errCode = m_config->set("event_cb", m_event_cb, errorStr);
if (errCode != RdKafka::Conf::CONF_OK) {
    std::cout << "Conf set failed:" << errorStr << std::endl;
}

// 可以设置自己的分区策略
m_partitioner_cb = new HashPartitionerCb;
errCode = m_topicConfig->set("partitioner_cb", m_partitioner_cb, errorStr);
if (errCode != RdKafka::Conf::CONF_OK) {
    std::cout << "Conf set failed:" << errorStr << std::endl;
}
// 设置broker的地址
errCode = m_config->set("bootstrap.servers", m_brokers, errorStr);
if (errCode != RdKafka::Conf::CONF_OK) {
    std::cout << "Conf set failed:" << errorStr << std::endl;
}

```

消息发送

librdkafka提供的异步的生产接口，**异步的消费接口和同步的消息接口，没有同步的生产接口。**

创建生产者

```

// 创建Producer，可以发布不同的主题
m_producer = RdKafka::Producer::create(m_config, errorStr);
if (m_producer == NULL) {
    std::cout << "Create Producer failed:" << errorStr << std::endl;
}
// 创建Topic对象
m_topic =
    RdKafka::Topic::create(m_producer, m_topicStr, m_topicConfig, errorStr);
if (m_topic == NULL) {
    std::cout << "Create Topic failed:" << errorStr << std::endl;
}

```

发送消息

同一个生产者可以发送多个主题的，在内部处理时根据传入的topic对象发送给对应的主题分区。

```

RdKafka::ErrorCode errorCode = m_producer->produce(
    m_topic, // 指定发送到哪个主题
    RdKafka::Topic::PARTITION_UA, // 指定分区，如果为PARTITION_UA则通过
    // partitioner_cb的回调选择合适的分区
    RdKafka::Producer::RK_MSG_COPY, // 消息拷贝
    payload, // 消息本身
    len, // 消息长度
    &key, // 消息key
    NULL // an optional application-provided per-message opaque pointer
    // that will be provided in the message delivery callback to let
    // the application reference a specific message
);

```

1.2 其他重要的生产者参数

1. acks

用来指定分区中必须要有多少个副本收到这条消息，之后生产者才会认为这条消息是成功写入的。acks 是生产者客户端中一个非常重要的参数，它涉及消息的可靠性和吞吐量之间的权衡。acks 参数有3种类型的值（都是字符串类型）。

- acks = 1。默认值即为 1。生产者发送消息之后，只要分区的 leader 副本成功写入消息，那么它就会收到来自服务端的成功响应。如果消息无法写入 leader 副本，比如在 leader 副本崩溃、重新选举新的 leader 副本的过程中，那么生产者就会收到一个错误的响应，为了避免消息丢失，生产者可以选择重发消息。如果消息写入 leader 副本并返回成功给生产者，且在被其他 follower 副本拉取之前 leader 副本崩溃，那么此时消息还是会丢失，因为新选举的 leader 副本中并没有这条对应的消息。

acks 设置为 1，是消息可靠性和吞吐量之间的折中方案。

- acks = 0。生产者发送消息之后不需要等待任何服务端的响应。

如果在消息从发送到写入 Kafka 的过程中出现了某些异常，导致 Kafka 并没有收到这条消息，那么生产者也无从得知，消息也就丢失了。

在其他配置环境相同的情况下，acks 设置为 0 可以达到最大的吞吐量。

- acks = -1 或 acks = all。生产者在消息发送之后，需要等待 ISR 中的所有副本都成功写入消息之后才能够收到来自服务端的成功响应。

在其他配置环境相同的情况下，acks 设置为 -1 可以达到最强的可靠性。

但是并不意味着消息就一定可靠，因为 ISR 中可能只有 leader 副本，这样就退化成了 acks = 1 的情况。要获得更高的消息可靠性需要配合 min.insync.replicas 等参数的联动。

注意 acks 参数配置的值是一个字符串类型，而不是整数类型。

范例:

```
RdKafka::Conf *conf = RdKafka::Conf::create(RdKafka::Conf::CONF_GLOBAL);
ConfResult ret = conf->set("acks", "1", errstr);
ConfResult ret = conf->set("acks", "0", errstr);
ConfResult ret = conf->set("acks", "all", errstr);
```

2. max.request.size

- 这个参数用来限制生产者客户端能够发送的消息的最大值，默认值为 1048576 B，即 1 MB。
- 一般情况下，这个默认值就可以满足大多数的应用场景了。
- 不建议盲目地增大这个参数的配置值，尤其是在对 Kafka 整体脉络没有足够把控的时候。
- 因为这个参数还涉及一些其他参数的联动，比如 broker 端的 message.max.bytes 参数，如果配置错误可能会引起一些不必要的一场。
- 比如讲 broker 端的 message.max.bytes 参数配置为 10，而 max.request.size 参数配置为 20，那么当我们发送一条消息大小为 15 的消息时，生产者客户端就会报出异常：

The request included a message larger than the max message size the server will accept.

范例

```
errCode = conf->set("message.max.bytes", "10240000", errorStr);
```

3 retries 和 retry.backoff.ms

retries 重试次数，默认0

retry.backoff.ms 重试间隔，默认100

- retries 参数用来配置生产者重试的次数，默认值为 0，**即发生异常的时候不进行任何的重试动作。**
- 消息在从生产者发出道成功写入服务器之前可能发生一些临时性的异常，比如网络抖动、Leader 副本的选举等，这种异常往往是可以自行恢复的，生产者可以通过配置 retries 大于 0 的值，以此通过内部重试来恢复而不是一味的将异常抛给生产者的应用程序。
- 如果重试达到设定的次数，那么生产者就会放弃重试并返回异常。
- 不过并不是所有的异常都是可以通过重试来解决的，比如消息太大，超过 max.request.size 参数配置的值时，这种方式就不行了。
- 重试还和另一个参数 **retry.backoff.ms** 有关，这个参数的默认值为 100，它用来设定两次重试之间的时间间隔，避免无效的频繁重试。
- 在配置 retries 和 retry.backoff.ms 之前，最好先估算一下可能的异常恢复时间，这样可以设定总的重试时间大于这个异常恢复时间，以此来避免生产者过早地放弃重试。
- Kafka 可以保证同一个分区中的消息时有序的。
- 如果生产者按照一定的顺序发送消息，那么这些消息也会顺序的写入分区，进而消费者也可以按照同样的顺序消费它们。
- 对于某些应用来说，顺序性非常重要，比如 Mysql 的 binlog 传输，如果出现错误就会造成非常严重的后果。
- 如果讲 retries 参数设置为非零值，并且 max.in.flight.requests.per.connection 参数配置为大于 1 的值，那么就会出现错序的现象：如果第一批次消息写入失败，而第二批次消息写入成功，那么生产者会重试发送第一批次的消息，此时如果第一批次的消息写入成功，那么这两个批次的消息就出现了错序。

- 一般而言，在需要保证顺序的场合建议把参数 **max.in.flight.requests.per.connection** 配置为 1，而不是把 **retries** 配置为 0。不过这样也会影响整体的吞吐。

max.in.flight.requests.per.connection = 1 限制客户端在单个连接上能够发送的未响应请求的个数。设置此值是 1 表示 kafka broker 在响应请求之前 client 不能再向同一个 broker 发送请求。注意：设置此参数是为了避免消息乱序

4 compression.type

- 这个参数用来指定消息的压缩方式，默认值为 “none”，即默认情况下，消息不会被压缩。
- 该参数还可以配置为 “gzip”，“snappy”，“lz4”。
- 对消息进行压缩可以极大地减少网络传输量、降低网络 I/O，从而提高整体的性能。
- 消息压缩是一种使用时间换空间的优化方式，如果对时延有一定的要求，则不推荐对消息进行压缩。

5 connection.max.idle.ms

这个参数用来指定在多久之后关闭闲置的连接，默认值为 540000 ms，即 9 分钟。

6 linger.ms

- 这个参数用来指定生产者发送 Producer Batch 之前等待更多消息（ProducerRecord）加入 ProducerBatch 的时间，默认值为 0。
- 生产者客户端会在 ProducerBatch 被填满或等待时间超过 **linger.ms** 值时发送出去。
- **增大这个参数的值会增加消息的延迟，但同时能提升一定的吞吐量。**
- 这个 **linger.ms** 参数与 TCP 协议中的 Nagle 算法有异曲同工之妙。

7 receive.buffer.bytes

- 这个参数用来设置 Socket 接受消息缓冲区（SO_RECVBUF）的大小，默认值为 32768（B），即 32 KB。
- 如果设置为 -1，则使用操作系统的默认值。
- 如果 Producer 与 Kafka 处于不同的机房，则可以适当调大这个参数值。

8 send.buffer.bytes

- 这个参数用来设置 Socket 发送消息缓冲区（SO_SNDBUF）的大小，默认值为 131072（B），即 128 KB。
- 与 **receive.buffer.bytes** 参数一样，如果设置为 -1，则使用操作系统默认值。

9 request.timeout.ms

- 这个参数用来配置 Producer 等待请求响应的最长时间，默认值为 30000 ms。
- 请求超时之后可以选择进行重试。
- 注意这个参数需要比 broker 端参数 **replica.lag.time.max.ms** 的值要大，这样可以减少因客户端重试而引起的消息重复的概率。

11 client.id

用来设定 KafkaProducer 对应的客户端 id。

默认值为 ""。

12 batch.size

batch.size 是 producer 最重要的参数之一！它对于调优 producer 吞吐量和延时性能指标都有着非常重要的作用。

producer 会将发往同一分区的多条消息封装进一个 batch 中，当 batch 满了的时候，producer 会发送 batch 中的所有消息。不过，producer 并不总是等待 batch 满了才发送消息，很有可能当 batch 还有很多空闲空间时 producer 就发送该 batch。显然，batch 的大小就显得非常重要。

通常来说，一个小的 batch 中包含的消息数很少，因而一次发送请求能够写入的消息数也很少，所以 producer 的吞吐量会很低；一个 batch 非常之巨大，那么会给内存使用带来极大的压力，因为不管是否能够填满，producer 都会为该 batch 分配固定大小的内存。

因此 batch.size 参数的设置其实是一种时间与空间权衡的体现。**batch.size 参数默认值是 16384，即 16KB**。这其实是一个非常保守的数字。在实际使用过程中合理地增加该参数值，通常都会发现 producer 的吞吐量得到了相应的增加。

2 消费者

代码参考：cplus_kafka/c 目录的代码

2.1 客户端开发

正常消费逻辑

1. 配置消费者客户端参数及创建相应的消费者实例；
2. 订阅主题；
3. 拉取消息并消费；
4. 提交消息位移；
5. 关闭消费者实例；

必要的参数配置

在创建消费者的时候以下三个选项是必选的：

- `bootstrap.servers`：指定 broker 的地址清单，清单里不需要包含所有的 broker 地址，生产者会从给定的 broker 里查找 broker 的信息。不过建议至少要提供两个 broker 的信息作为容错；
- `group.id`：consumer group 是 kafka 提供的可扩展且具有容错性的消费者机制。既然是一个组，那么组内必然可以有多个消费者或消费者实例(consumer instance)，它们共享一个公共的 ID，即 group ID。组内的所有消费者协调在一起来消费订阅主题(subscribed topics)的所有分区(partition)。
- `auto.offset.reset`：
- 这个参数是针对新的 groupid 中的消费者而言的，当有新 groupid 的消费者来消费指定的 topic 时，对于该参数的配置，会有不同的语义
 - `none`：如果没有为消费者找到先前的offset的值,即没有自动维护偏移量,也没有手动维护偏移量,则抛出异常
 - `earliest`：
 - 在各分区下有提交的offset时：从offset处开始消费
 - 在各分区下无提交的offset时：从头开始消费
 - `latest`：
 - 在各分区下有提交的offset时：从offset处开始消费
 - 在各分区下无提交的offset时：从最新的数据开始消费

```
std::string errorStr;
RdKafka::Conf::ConfResult errorCode;
m_config = RdKafka::Conf::create(RdKafka::Conf::CONF_GLOBAL);

m_event_cb = new ConsumerEventCb;
errorCode = m_config->set("event_cb", m_event_cb, errorStr);
if(errorCode != RdKafka::Conf::CONF_OK)
{
    std::cout << "Conf set failed: " << errorStr << std::endl;
}

m_rebalance_cb = new ConsumerRebalanceCb;
errorCode = m_config->set("rebalance_cb", m_rebalance_cb, errorStr);
if(errorCode != RdKafka::Conf::CONF_OK)
{
    std::cout << "Conf set failed: " << errorStr << std::endl;
}

errorCode = m_config->set("group.id", m_groupID, errorStr);
if(errorCode != RdKafka::Conf::CONF_OK)
{
    std::cout << "Conf set failed: " << errorStr << std::endl;
}

errorCode = m_config->set("bootstrap.servers", m_brokers, errorStr);
if(errorCode != RdKafka::Conf::CONF_OK)
{
    std::cout << "Conf set failed: " << errorStr << std::endl;
}
```

```
// partition.assignment.strategy range,roundrobin
errorCode =
    m_config->set("partition.assignment.strategy", "range", errorStr);
if (errorCode != RdKafka::Conf::CONF_OK) {
    std::cout << "Conf set failed: " << errorStr << std::endl;
}
```

订阅主题和分区

订阅主题，可以订阅多个

ErrorCode subscribe (const std::vector<std::string> &topics)

也可以通过正则表达式方式一次订阅多个主题，比如 “topic.*”，则前缀为“topic-”的主题都被订阅。

```
m_topicConfig = RdKafka::Conf::create(RdKafka::Conf::CONF_TOPIC);
// 获取最新的消息数据
errorCode = m_topicConfig->set("auto.offset.reset", "latest", errorStr);
if (errorCode != RdKafka::Conf::CONF_OK) {
    std::cout << "Topic Conf set failed: " << errorStr << std::endl;
}
errorCode = m_config->set("default_topic_conf", m_topicConfig, errorStr);
if (errorCode != RdKafka::Conf::CONF_OK) {
    std::cout << "Conf set failed: " << errorStr << std::endl;
}
m_consumer = RdKafka::KafkaConsumer::create(m_config, errorStr);
if (m_consumer == NULL) {
    std::cout << "Create kafkaConsumer failed: " << errorStr << std::endl;
}
std::cout << "Created consumer " << m_consumer->name() << std::endl;

// 订阅主题，可以订阅多个主题
RdKafka::ErrorCode errorCode = m_consumer->subscribe(m_topicVector);
```

消息消费

```
void KafkaConsumer::pullMessage() {
    // 订阅Topic
    RdKafka::ErrorCode errorCode = m_consumer->subscribe(m_topicVector);
    if (errorCode != RdKafka::ERR_NO_ERROR) {
        std::cout << "subscribe failed: " << RdKafka::err2str(errorCode)
            << std::endl;
    }
}
```

```

// 消费消息
while (true) {
    RdKafka::Message *msg = m_consumer->consume(1000);
    msg_consume(msg, NULL);
    delete msg;
}

}

void msg_consume(RdKafka::Message *msg, void *opaque) {
    switch (msg->err()) {
    case RdKafka::ERR__TIMED_OUT:
        std::cerr << "Consumer error: " << msg->errstr() << std::endl; // 超时
        break;
    case RdKafka::ERR_NO_ERROR: // 有消息进来
        std::cout << " Message in-> topic:" << msg->topic_name()
            << ", partition:[" << msg->partition() << "]" at offset "
            << msg->offset() << " key: " << msg->key()
            << " payload: " << (char *)msg->payload() << std::endl;

        break;
    default:
        std::cerr << "Consumer error: " << msg->errstr() << std::endl;
        break;
    }
}
}

```

2.2 位移提交

Consumer 需要向 [Kafka](#) 汇报自己的位移数据，这个汇报过程被称为提交位移（Committing Offsets）。因为 Consumer 能够同时消费多个分区的数据，所以位移的提交实际上是在分区粒度上进行的，即 Consumer 需要为分配给它的每个分区提交各自的位移数据。

提交位移主要是为了表征 Consumer 的消费进度，这样当 Consumer 发生故障重启之后，就能够从 **Kafka 中读取之前提交的位移值**，然后从相应的位移处继续消费，从而避免整个消费过程重来一遍。

从用户的角度来说，位移提交分为**自动提交**和**手动提交**；从 Consumer 端的角度来说，位移提交分为**同步提交**和**异步提交**。

自动提交

自动提交默认全部为同步提交

自动提交相关参数

- **enable.auto.commit** (bool) – 如果为True，将自动定时提交消费者offset。默认为True。
- **auto.commit.interval.ms**(int) – 自动提交offset之间的间隔毫秒数。如果enable_auto_commit为true，默认值为：5000。

当设置 **enable.auto.commit** 为 true，**Kafka 会保证在开始调用 poll 方法时，提交上次 poll 返回的所有消息**。从顺序上来说，poll 方法的逻辑是先提交上一批消息的位移，再处理下一批消息，因此它能保证不出现消费丢失的情况。

但自动提交位移的一个问题在于，它可能会出现重复消费。

如果设置 `enable.auto.commit` 为 `true`，Consumer 按照 `auto.commit.interval.ms` 设置的值（默认5秒）自动提交一次位移。我们假设提交位移之后的 3 秒发生了 Rebalance 操作。在 Rebalance 之后，所有 Consumer 从上一次提交的位移处继续消费，但该位移已经是 3 秒前的位移数据了，故在 Rebalance 发生前 3 秒消费的所有数据都要重新再消费一次。虽然你能够通过减少 `auto.commit.interval.ms` 的值来提高提交频率，但这么做只能缩小重复消费的时间窗口，不可能完全消除它。这是自动提交机制的一个缺陷。

在实际测试中，未发现上述情况（kafka 版本 2.11），而是会等待所有消费者消费完当前消息，或者等待消费者超时（等待过程中会报如下 warning），之后才会 rebalance。

手动提交

手动提交可以自己选择是同步提交（`commitSync`）还是异步提交（`commitAsync`）

`commitAsync` 不能够替代 `commitSync`。`commitAsync` 的问题在于，出现问题时它不会自动重试。因为它是异步操作，倘若提交失败后自动重试，那么它重试时提交的位移值可能早已经“过期”或不是最新值了。因此，异步提交的重试其实没有意义，所以 `commitAsync` 是不会重试的。

手动提交，我们需要将 `commitSync` 和 `commitAsync` 组合使用才能到达最理想的效果，原因有两个：

1. 我们可以利用 `commitSync` 的自动重试来规避那些瞬时错误，比如网络的瞬时抖动，Broker 端 GC 等。因为这些问题都是短暂的，自动重试通常会成功，因此，我们不想自己重试，而是希望 Kafka Consumer 帮我们做这件事。我们不希望程序总处于阻塞状态，影响 TPS。
2. 我们不希望程序总处于阻塞状态，影响 TPS。

同时使用 `commitSync()` 和 `commitAsync()`

- 对于常规性、阶段性的手动提交，我们调用 `commitAsync()` 避免程序阻塞
- 而在 Consumer 要关闭前，我们调用 `commitSync()` 方法执行同步阻塞式的位移提交，以确保 Consumer 关闭前能够保存正确的位移数据。

将两者结合后，我们既实现了异步无阻塞式的位移管理，也确保了 Consumer 位移的正确性。

手动提交和自动提交中的 rebalance 问题

- 如果设置为手动提交，当集群满足 rebalance 的条件时，集群会直接 rebalance，不会等待所有消息被消费完，这会导致所有未被确认的消息会重新被消费，会出现重复消费的问题
- 如果设置为自动提交，当集群满足 rebalance 的条件时，集群不会马上 rebalance，而是会等待所有消费者消费完当前消息，或者等待消费者超时（等待过程中会报如下 warning），之后才会 rebalance。

提交API

- `ErrorCode commitSync();`
提交当前分配分区的位移，同步操作，会阻塞直到位移被提交或提交失败。如果注册了 `RdKafka::OffsetCommitCb` 回调函数，其会在 `KafkaConsumer::consume()` 函数内调用并提交位移。

- ErrorCode **commitAsync()**;
异步提交位移
- ErrorCode commitSync(Message *message);
基于消息对单个topic+partition对象同步提交位移
- virtual ErrorCode commitSync(std::vector<TopicPartition*> &offsets) = 0;
对指定多个TopicPartition同步提交位移
- ErrorCode commitAsync(Message *message);
基于消息对单个TopicPartition异步提交位移
- virtual ErrorCode commitAsync (const std::vector<TopicPartition*> &offsets) = 0;
对多个TopicPartition异步提交位移

```
/* To make sure offset commits are fetched in proper assign sequence
 * we commit an offset that should not be used in the final consume loop.
 * This commit will be overwritten below with another commit. */
vector<RdKafka::TopicPartition *> offsets;
offsets.push_back(RdKafka::TopicPartition::create(toppars1[0]->topic(),
                                                    toppars1[0]->partition(),
                                                    11));

/* This partition should start at this position even though
 * there will be a sub-subsequent commit to overwrite it, that should not
 * be used since this partition is never unassigned. */
offsets.push_back(RdKafka::TopicPartition::create(toppars2[0]->topic(),
                                                    toppars2[0]->partition(),
                                                    22));

pos[Toppar(toppars2[0]->topic(), toppars2[0]->partition())] = 22;

Test::print_TopicPartitions("pre-commit", offsets);

RdKafka::ErrorCode err;
err = consumer->commitSync(offsets);
```

2.3 消费Rebalance机制

当kafka遇到如下四种情况的时候，kafka会触发Rebalance机制：

1. 消费组成员发生了变更，比如有新的消费者加入了消费组或者有消费者宕机
2. 消费者无法在指定的时间之内完成消息的消费
3. 消费组订阅的Topic发生了变化
4. 订阅的Topic的partition发生了变化

后面两个通常都是运维的主动操作，所以它们引发的 Rebalance 大都是不可避免的。接下来，我们主要讲解因为组成员数量变化而引发的 Rebalance 该如何避免。

- **session.timeout.ms** 表示 consumer 向 broker 发送心跳的超时时间。例如 session.timeout.ms = 180000 表示在最长 180 秒内 broker 没收到 consumer 的心跳，那么 broker 就认为该 consumer 死亡了，会启动 rebalance。
- **heartbeat.interval.ms** 表示 consumer 每次向 broker 发送心跳的时间间隔。heartbeat.interval.ms = 60000 表示 consumer 每 60 秒向 broker 发送一次心跳。一般来说，session.timeout.ms 的值是 heartbeat.interval.ms 值的 3 倍以上。

- **max.poll.interval.ms** 表示 consumer 每两次 poll 消息的时间间隔。简单地说，其实就是 consumer 每次消费消息的时长。如果消息处理的逻辑很重，那么时长就要相应延长。否则如果时间到了 consumer 还没消费完，broker 会默认认为 consumer 死了，发起 rebalance。
- **max.poll.records** 表示每次消费的时候，获取多少条消息。获取的消息条数越多，需要处理的时间越长。所以每次拉取的消息数不能太多，需要保证在 max.poll.interval.ms 设置的时间内能消费完，否则会发生 rebalance。

总结两点就是可能会导致崩溃的点就是，消费者心跳超时和消费者消费数据超时

解决方法已经很明朗了，就是适当调参，心跳超时就调整 **session.timeout.ms** 和 **heartbeat.interval.ms**。session.timeout.ms 决定了 Consumer 存活性的时间间隔。

- 这里给出一些推荐数值，可以“无脑”地应用在你的生产环境中。
 - 设置 session.timeout.ms = 6s。
 - 设置 heartbeat.interval.ms = 2s。
 - 要保证 Consumer 实例在被判定为“dead”之前，能够发送至少 3 轮的心跳请求，即 $\text{session.timeout.ms} \geq 3 * \text{heartbeat.interval.ms}$ 。
- 对于消费处理超时问题。一般是增加消费者处理的时间 (max.poll.interval.ms)，减少每次处理的消息数 (max.poll.records)。阿里云官方文档建议 max.poll.records 参数要远小于当前消费组的消费能力 ($\text{records} < \text{单个线程每秒消费的条数} * \text{消费线程的个数} * \text{session.timeout的秒数}$)。

2.4 其他重要的消费者参数

在 KafkaConsumer 中，除了前面讲的必要的客户端参数，大部分的参数都有合理的默认值，一般我们也不需要去修改它们。不过了解这些参数可以让我们更好地使用消费者客户端，其中还有一些重要的参数涉及程序的可用性和性能，如果能够熟练掌握它们，也可以让我们在编写相关的程序时能够更好地进行性能调优与故障排查。下面挑选一些重要的参数来做细致的讲解。

1 fetch.min.bytes

该参数用来配置 Consumer 在一次拉取请求（调用 poll() 方法）中能从 Kafka 中拉取的最小数据量，默认值为 1 (B)。Kafka 在收到 Consumer 的拉取请求时，如果返回给 Consumer 的数据量小于这个参数所配置的值，那么它就需要进行等待，直到数据量满足这个参数的配置大小。可以适当调大这个参数的值以提高一定的吞吐量，不过也会造成额外的延迟 (latency)，对于延迟敏感的应用可能就不取了。

2 fetch.max.bytes

该参数与 fetch.min.bytes 参数对应，它用来配置 Consumer 在一次拉取请求中从 Kafka 中拉取的最大数据量，默认值为 52428800 (B)，也就是 50MB。

如果这个参数设置的值比任何一条写入 Kafka 中的消息要小，那么会不会造成无法消费呢？很多资料对此参数的解读认为是无法消费的，比如一条消息的大小为 10B，而这个参数的值是 1 (B)，既然此参数设定的值是一次拉取请求中所能拉取的最大数据量，那么显然 $1B < 10B$ ，所以无法拉取。这个观点是错误的，该参数设定的不是绝对的最大值，如果在第一个非空分区中拉取的第一条消息大于该值，那么该消息将仍然返回，以确保消费者继续工作。也就是说，上面问题的答案是可以正常消费。

与此相关的，Kafka 中所能接收的最大消息的大小通过服务端参数 message.max.bytes（对应于主题端参数 max.message.bytes）来设置。

3 fetch.max.wait.ms

这个参数也和 fetch.min.bytes 参数有关，如果 Kafka 仅仅参考 fetch.min.bytes 参数的要求，那么有可能会一直阻塞等待而无法发送响应给 Consumer，显然这是不合理的。fetch.max.wait.ms 参数用于指定 Kafka 的等待时间，默认值为500（ms）。如果 Kafka 中没有足够多的消息而满足不了 fetch.min.bytes 参数的要求，那么最终会等待500ms。这个参数的设定和 Consumer 与 Kafka 之间的延迟也有关系，如果业务应用对延迟敏感，那么可以适当调小这个参数。

4 max.partition.fetch.bytes

这个参数用来配置从每个分区里返回给 Consumer 的最大数据量，默认值为1048576（B），即1MB。这个参数与 fetch.max.bytes 参数相似，只不过前者用来限制一次拉取中每个分区的信息大小，而后者用来限制一次拉取中整体信息的大小。同样，如果这个参数设定的值比信息的大小要小，那么也不会造成无法消费，Kafka 为了保持消费逻辑的正常运转不会对此做强硬的限制。

5 max.poll.records

这个参数用来配置 Consumer 在一次拉取请求中拉取的最大消息数，默认值为500（条）。如果信息的大小都比较小，则可以适当调大这个参数值来提升一定的消费速度。

如果用户的信息处理逻辑很轻量，默认的 500 条消息通常不能满足实际的信息处理速度。

6 connections.max.idle.ms

这个参数用来指定在多久之后关闭闲置的连接，默认值是540000（ms），即9分钟。

7 exclude.internal.topics

Kafka 中有两个内部的主题：**consumer_offsets** 和 **transaction_state**。exclude.internal.topics 用来指定 Kafka 中的内部主题是否可以向消费者公开，默认值为 true。如果设置为 true，那么只能使用 subscribe(Collection)的方式而不能使用 subscribe(Pattern)的方式来订阅内部主题，设置为 false 则没有这个限制。

8 receive.buffer.bytes

这个参数用来设置 Socket 接收消息缓冲区（SO_RECVBUF）的大小，默认值为65536（B），即64KB。如果设置为-1，则使用操作系统的默认值。如果 Consumer 与 Kafka 处于不同的机房，则可以适当调大这个参数值。

9 send.buffer.bytes

这个参数用来设置Socket发送消息缓冲区（SO_SNDBUF）的大小，默认值为131072（B），即128KB。与receive.buffer.bytes参数一样，如果设置为-1，则使用操作系统的默认值。

10 request.timeout.ms

这个参数用来配置 Consumer 等待请求响应的最长时间，默认值为30000（ms）。

11 metadata.max.age.ms

这个参数用来配置元数据的过期时间，默认值为300000（ms），即5分钟。如果元数据在此参数所限定的时间范围内没有进行更新，则会被强制更新，即使没有任何分区变化或有新的 broker 加入。

12 reconnect.backoff.ms

这个参数用来配置尝试重新连接指定主机之前的等待时间（也称为退避时间），避免频繁地连接主机，默认值为50（ms）。这种机制适用于消费者向 broker 发送的所有请求。

13 retry.backoff.ms

这个参数用来配置尝试重新发送失败的请求到指定的主题分区之前的等待（退避）时间，避免在某些故障情况下频繁地重复发送，默认值为100（ms）。

14 isolation.level

这个参数用来配置消费者的事务隔离级别。字符串类型，有效值为“read_uncommitted”和“read_committed”，表示消费者所消费到的位置，如果设置为“read_committed”，那么消费者就会忽略事务未提交的消息，即只能消费到LSO（LastStableOffset）的位置，默认情况下为“read_uncommitted”，即可以消费到 HW（High Watermark）处的位置。

15 session.timeout.ms

非常重要的参数之一！

session.timeout.ms 是 consumer group 检测组内成员发送崩溃的时间。

假设你设置该参数为 5 分钟，那么当某个 group 成员突然崩溃了（比如被 kill -9 或宕机），管理 group 的 Kafka 组件（即消费者组协调者，也称 group coordinator）有可能需要 5 分钟才能感知到这个崩溃。显然我们想要缩短这个时间，让 coordinator 能够更快地检测到 consumer 失败。

这个参数还有另外一重含义：**consumer 消息处理逻辑的最大时间。**

倘若 consumer 两次 poll 之间的间隔超过了该参数所设置的阈值，那么 coordinator 就会认为这个 consumer 已经追不上组内其他成员的消费进度了，因此会将该 consumer 实例“踢出”组，该 consumer 负责的分区也会被分配给其他 consumer。

在最好的情况下，这会导致不必要的 rebalance，因为 consumer 需要重新加入 group。更糟的是，对于那些在被踢出 group 后处理的消息，consumer 都无法提交位移——这就意味着这些消息在 rebalance 之后会被重新消费一遍。如果一条消息或一组消息总是需要花费很长的时间处理，那么 consumer 甚至无法执行任何消费，除非用户重新调整参数。鉴于以上的“窘境”，Kafka 社区于 0.10.1.0 版本对该参数的含义进行了拆分。在该版本及以后的版本中，session.timeout.ms 参数被明确为“coordinator 检测失败的时间”。

因此在实际使用中，用户可以为该参数设置一个比较小的值，让 coordinator 能够更快地检测 consumer 崩溃的情况，从而更快地开启 rebalance，避免造成更大的消费滞后（consumer lag）。目前该参数的默认值是 10 秒。

16 max.poll.interval.ms

Apache官网[max.poll.interval.ms](https://kafka.apache.org/documentation.html#brokerconfigs)上的解释如下，消费者组中的一员在拉取消息时如果超过了设置的最大拉取时间，则会认为消费者消费消息失败，kafka会重新进行重新负载均衡，以便把消息分配给另一个消费组成员。

在一个典型的 consumer 使用场景中，用户对于消息的处理可能需要花费很长时间。这个参数就是用于设置消息处理逻辑的

最大时间的。假设用户的业务场景中消息处理逻辑是把消息、“落地”到远程数据库中，且这个过程平均处理时间是 2 分钟，那么用户仅需要将 max.poll.interval.ms 设置为稍稍大于 2 分钟的值即可，而不必为 session.timeout.ms 也设置这么大的值。

通过将该参数设置成实际的逻辑处理时间再结合较低的 session.timeout.ms 参数值，consumer group 既实现了快速的 consumer 崩溃检测，也保证了复杂的事件处理逻辑不会造成不必要的 rebalance。

17 heartbeat.interval.ms

该参数和 request.timeout.ms、max.poll.interval.ms 参数是最难理解的 consumer 参数。前面已经讨论了后两个参数的含义，这里解析一下 heartbeat.interval.ms 的含义及用法。

从表面上看，该参数似乎是心跳的间隔时间，但既然已经有了上面的 session.timeout.ms 用于设置超时，为何还要引入这个参数呢？

这里的关键在于要搞清楚 consumer group 的其他成员，如何得知要开启新一轮 rebalance；当 coordinator 决定开启新一轮 rebalance 时，它会将这个决定以 REBALANCE_IN_PROGRESS 异常的形式“塞进”consumer 心跳请求的 response 中，这样其他成员拿到 response 后才能知道它需要重新加入 group。显然这个过程越快越好，而 heartbeat.interval.ms 就是用来做这件事情的。

比较推荐的做法是设置一个比较低的值，让 group 下的其他 consumer 成员能够更快地感知新一轮 rebalance 开启了。

注意，该值必须小于 session.timeout.ms！这很容易理解，毕竟如果 consumer 在 session.timeout.ms 这段时间内都不发送心跳，coordinator 就会认为它已经 dead，因此也就没有必要让它知晓 coordinator 的决定了。

3 broker参数详解

<https://kafka.apache.org/documentation.html#brokerconfigs>

broker 端参数需要在 Kafka 目录下的 config/server.properties 文件中进行设置。当前对于绝大多数的 broker 端参数而言，Kafka 尚不支持动态修改——这就是说，如果要新增、修改，抑或是删除某些 broker 参数的话，需要重启对应的 broker 服务器。

broker.id

Kafka 使用唯一的一个整数来标识每个 broker，这就是 broker.id；该参数默认是-1。如果不指定，Kafka 会自动生成一个唯一值；总之，不管用户指定什么都必须保证该值在 Kafka 集群中是唯一的，不能与其他 broker 冲突。

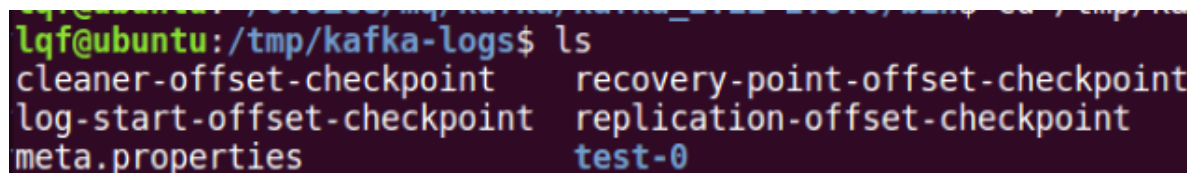
在实际使用中，推荐使用从0开始的数字序列，如0、1、2.....

log.dirs

```
# A comma separated list of directories under which to store log files
log.dirs=/tmp/kafka-logs
```

该参数指定了 Kafka 持久化消息的目录；非常重要的参数！

若不设置该参数，Kafka 默认使用 **tmp/kafka-logs** 作为消息保存的目录(商用一定要自己设置目录)。比如：



```
lqf@ubuntu: /tmp/kafka-logs$ ls
cleaner-offset-checkpoint  recovery-point-offset-checkpoint
log-start-offset-checkpoint  replication-offset-checkpoint
meta.properties           test-0
```

把消息保存在 tmp 目录下，生产环境中是不可取的。若待保存的消息数量非常多，那么最好确保该文件夹下有充足的磁盘空间。

该参数可以设置多个目录，以逗号分隔，比如 **/home/kafka1,/home/kafka2**。在实际使用过程中，指定多个目录的做法通常是被推荐的，因为这样 Kafka 可以把负载均匀地分配到多个目录下。

若用户机器上有N块物理硬盘，那么设置 N 个目录（须挂载在不同磁盘上的目录）是一个很好的选择。N 个磁头可以同时执行写操作，极大地提升了吞吐量。

注意，这里的“均匀”是根据目录下的分区数进行比较的，而不是根据实际的磁盘空间。

zookeeper.connect

同样是非常重要的参数。主要是在 zookeeper 中，kafka 的配置数据，有一个公共开始的跟节点；该参数没有默认的值，如果不配置，则使用 zookeeper 的 / 目录；

例如：zk1:2181,zk2:2181,zk3:2181/kafka_cluster1; /kafka_cluster 就是 kafka 的配置的目录；配置了，可以起到很好的隔离效果。这样管理 Kafka 集群将变得更加容易。

listeners

broker 监听器的 csv (comma-separated values) 列表，格式是 [协议://主机名:端口], [协议]:[/ 主机名:端口]。

该参数主要用于客户端连接 broker 使用，可以认为是 broker 端开放给 clients 的监听端口。如果不指定主机名，则表示绑定默认网卡：如果 0.0.0.0，则表示绑定所有网卡。Kafka 当前支持的协议类型包括 PLAINTEXT、SSL 以及 SASL SSL 等。

advertised.listeners

跟 listeners 类似，该参数也是用于发布给 clients 的监听器；不过该参数主要用于 IaaS 环境，比如云上的机器通常都配有多块网卡（私网网卡和公网网卡）。

对于这种机器，用户可以设置该参数绑定公网 IP 供外部 clients 使用，然后配置上面的 listeners 来绑定私网 IP 供 broker 间通信使用。

当然不设置该参数也是可以的，只是云上的机器很容易出现 clients 无法获取数据的问题，原因就是 listeners 绑定的是默认网卡，而默认网卡通常都是绑定私网的。

在实际使用场景中，对于配有多块网卡的机器而言，这个参数通常都是需要配置的。

unclean.leader.election.enable

是否开启 unclean leader 选举。Kafka 社区在1.0.0 版本才正式将该参数默认值调整为 **false** ；

即表明如果发生这种情况，Kafka 不允许从剩下存活的非 ISR 副本中选择一个当 leader。因为如果 true，这样做固然可以让 Kafka 继续提供服务给 clients，但会造成消息数据的丢失，正式环境中，数据不丢失是基本的业务需求；

番外：ISR解释

ISR 全称是**in-sync replica**，翻译过来就是与 leader replica 保持同步的 replica 集合；一个partition 可以配置N个replica，那么这是否就意味着该 partition可以容忍 N-1 replica 失效而不丢失数据呢？答案是“否”！

Kafka partition 动态维护 一个replica 集合。该集合中的所有 replica 保存的消息日志都与leader replica 保持同步状态。只有这个集合中的 replica 才能被选举为 leader，也只有该集合中所有 replica 都接收到了同一条消息，Kafka 才会将该消息置于“已提交”状态，即认为这条消息发送成功。

Kafka 中只要这个集合中至少存在一个 replica，那些“已提交”状态的消息就不会丢失；

这句话的两个关键点：

1. ISR 中至少存在一个“活着的”replica；
2. replica “已提”消息。

Kafka 对于没有提交成功的消息不做任何保证，只保证在 ISR 存活的情况下“已提交”的消息不会丢失。正常情况下，partition 的所有 replica（含 leader replica）都应该与 leader replica 保持同步，即所有 replica 都在 ISR 中。

因为各种各样的原因，一小部分 replica 开始落后于 leader replica的进度。当滞后一定程度时，Kafka 会将这些 replica “踢”出 ISR；相反地，当这replica 重新追上leader进度时候，kafka会再次把它们加回 ISR中。

delete.topic.enable

是否允许 Kafka 删除 topic。

默认情况下，**Kafka 集群允许用户删除topic 及其数据**。这样当用户发起删除 topic 操作时，broker 端会执行 topic 删除逻辑。

在实际生产环境中我们发现允许 Kafka 删除 topic 其实是一个很方便的功能，再加上自Kafka 0.0 新增的 ACL 权限特性，以往对于误操作和恶意操作的担心完全消失了，因此设置该参数为 true 是推荐的做法。

log.retention. {hours | minutes | ms }

这组参数控制了消息数据的留存时间；**默认的留存时间是7天(168小时)**；即 Kafka 只会保存最近 7天的数据，并自动删除 7天前的数据。

当前较新版本的Kafka 会根据消息的时间戳信息进行留存与否的判断；老版本消息格式没有时间戳，Kafka 会根据日志文件的最近修改时间（last modified time）进行判断。

三个参数如果同时设置，优先选取ms的设置，minutes次之，hours最后。

log.retention.bytes

这个参数定义了空间维度上的留存策略；参数默认值是-1，表示 Kafka 永远不会根据消息日志文件总大小来删除日志。

对于大小超过该参数的分区日志而言，Kafka 会自动清理该分区的过期日志段文件。

min.insync.replicas

该参数表示kafka存储的最小副本数，producer发送数据的时候，指定了 broker 端必须成功响应 clients 消息发送的最少副本数；

该参数其实是与 producer 端的 acks 参数配合使用的；并且min.insync.replicas 也只有在 acks=-1 时才有意义；acks=-1 表示 producer端寻求最高等级的持久化保证；

假如 broker 端无法满足该条件，则 clients 的消息发送并不会被视为成功。它与 acks 配合使用可以令 Kafka集群达成最高等级的消息持久化；

举个例子，**假设某个topic 的每个分区的副本数是3，那么推荐设置该参数为 2**，这样我们就能够容忍一台broker 宕机而不影响服务；若设置参数为3，那么只要任何一台 broker 宕机，整个Kafka 集群将无法继续提供服务。

num.network threads (fastdfs 网络数据读取)

一个 broker 在后台用于处理网络请求的线程数，默认是3。

broker启动时会创建多个线程处理来自其他broker和clients 发送过来的各种请求。会将接收到的请求转发到后面的处理线程中。在真实的环境中，用户需要不断地监控NetworkProcessorAvgIdlePercent JMX 指标；如果该指标持续低于0.3 ;建议适当增加该参数的值。

num.io.threads (处理我们读取到网络数据)

这个参数就是控制 broker 端实际处理网络请求的线程数，默认值是8；

Kafka broker 默认创建 8个线程以轮询方式不停地监听转发过来的网络请求并进行实时处理。Kafka 同样也为请求处理提供了一个 JMX 监控指标 RequestHandler AvgIdlePercent。如果发现该指标持续低于0.3，则可以考虑适当增加该参数的值。

message.max. bytes

Kafka broker 能够接收的最大消息大小，默认是 977kB；还不到1MB，可见是非常小的。

在实际使用场景中，突破 1MB 大小的消息十分常见，因此用户有必要综合考虑 Kafka 集群可能处理的最大消息尺寸并设置该参数值

log.segment.bytes (分片的概念)

(默认: 1GB) – kafka数据文件的大小，确保这个数值大于一个消息的长度。一般说来使用默认值即可（一般一个消息很难大于1G，因为这是一个消息系统，而不是文件系统）。

```
# The maximum size of a log segment file. When this size is reached a new log
segment will be created.
#log.segment.bytes=1073741824
# 配置成5M做测试
log.segment.bytes==5242880
```

4 更多配置

<https://kafka.apache.org/documentation.html#topicconfigs>

<https://kafka.apache.org/documentation.html#configuration>

<https://kafka.apachecn.org/documentation.html#majordesignelements>

参考

[Kafka性能优化---1 FD 2013的博客-CSDN博客_rdkafka::deliveryreportcb](#)

[Kafka C++客户端库librdkafka笔记 - 腾讯云开发者社区-腾讯云 \(tencent.com\)](#)

[Kafka组消费之Rebalance机制 - 知乎 \(zhihu.com\)](#)

[\(kafka rebalance与数据重复消费问题hellozhxy的博客-CSDN博客synchronous auto-commit of offsets](#)

[Kafka无消息丢失配置 - huxihx - 博客园 \(cnblogs.com\)](#)