Kafka

一、常用命令

```
sh /opt/confluent/bin/kafka-console-consumer --bootstrap-server 172.20.3.120:9092 --topic
alert_0109

sh /opt/confluent/bin/kafka-console-consumer --bootstrap-server 172.20.3.151:9092 --topic
alert_1204

sh /opt/confluent/bin/kafka-console-producer --broker-list
172.20.3.120:9092,172.20.3.121:9092,172.20.3.122:9092 --topic aiops_alert_sink

sh /opt/confluent/bin/kafka-console-producer --broker-list 172.20.3.120:9092 --topic
aiops_alert_sink

查看所有的消费者组：
/opt/confluent/bin/kafka-consumer-groups --bootstrap-server 172.20.3.120:9092 -list

查看消费者组下的详细信息：比如有哪些连接
/opt/confluent/bin/kafka-consumer-groups --bootstrap-server 172.20.3.120:9092 --group
aiops_alert_group --describe
```

1.kafka的删除策略

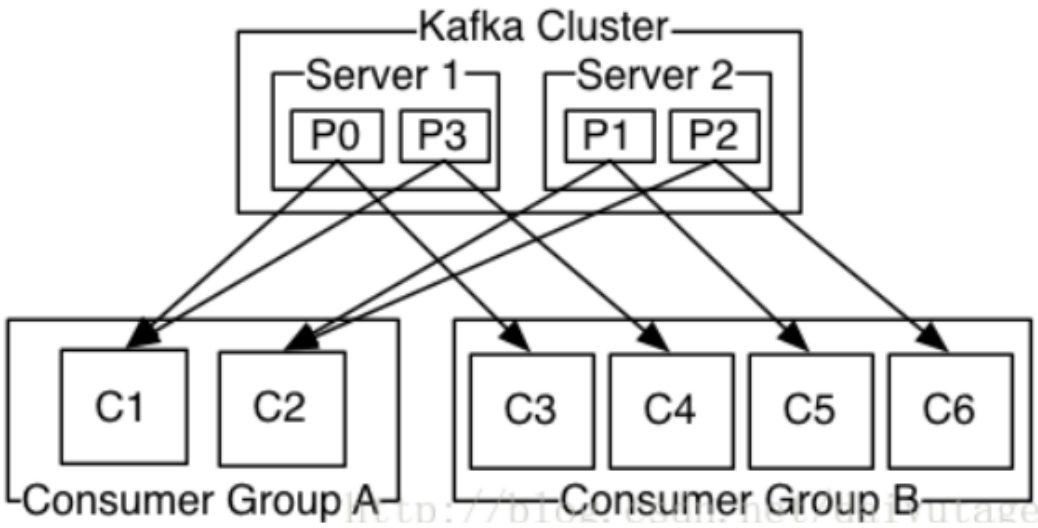
Kafka提供两种策略删除旧数据。

- 基于时间，
- 基于Partition文件大小。

通过配置$KAFKA_HOME/config/server.properties，让Kafka删除一周前的数据，也可在Partition文件超过1GB时删除旧数据，配置如下所示。

> 因为offset由Consumer控制，所以Kafka broker是无状态的，它不需要标记哪些消息被哪些消费过，也不需要通过broker去保证同一个Consumer Group只有一个Consumer能消费某一条消息，因此也就不需要锁机制，这也为Kafka的高吞吐率提供了有力保障。

2.kafka的基本组成



每个partition只能同一个group中的同一个consumer消费

而topic类似一个queue，每个partition中会存储所有消息和索引文件

这样设计的劣势是无法让同一个consumer group里的consumer均匀消费数据，
优势是每个consumer不用都跟大量的broker通信，减少通信开销，同时也降低了分配难度，实现也更简单。
另外，因为同一个partition里的数据是有序的，这种设计可以保证每个partition里的数据也是有序被消费。

3.kafka异步发送消息

```
kafkaProducer.send(record, (metadata, exception) -> {
    if (exception != null) {
        logger.error("send message to kafka failed. \n{}", ExceptionUtil.getMessage(exception));
    }else {
        logger.info(" topic = {},offset = {}",kafkaTopic,metadata.offset());
    }
});
```

二、底层知识

1.producer的幂等性

问题：生产者重复生产消息。生产者进行retry会产生重试时，会重复产生消息。有了幂等性之后，在进行retry重试时，只会生成一个消息。

kafka实现Exactly Once的一种方法是让下游系统具有幂等处理特性，而在 Kafka Stream 中，Kafka Producer 本身就是“下游”系统，因此如果能让 Producer 具有幂等处理特性，那就可以让Kafka Stream在一定程度上支持Exactly once语义。

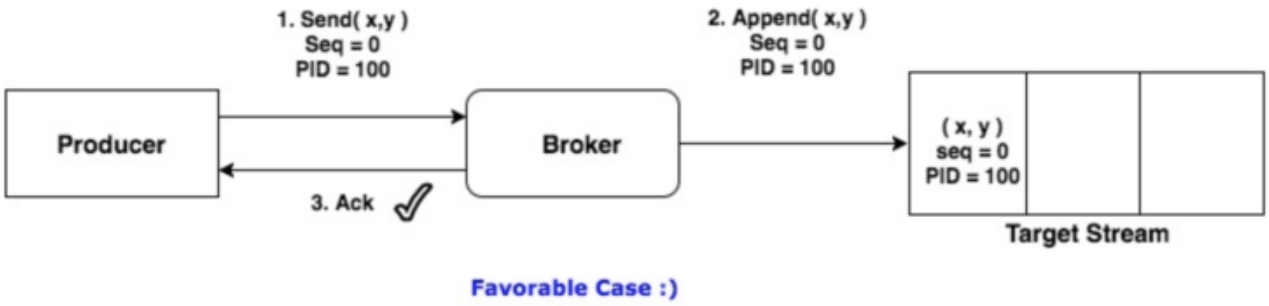
实现原理：

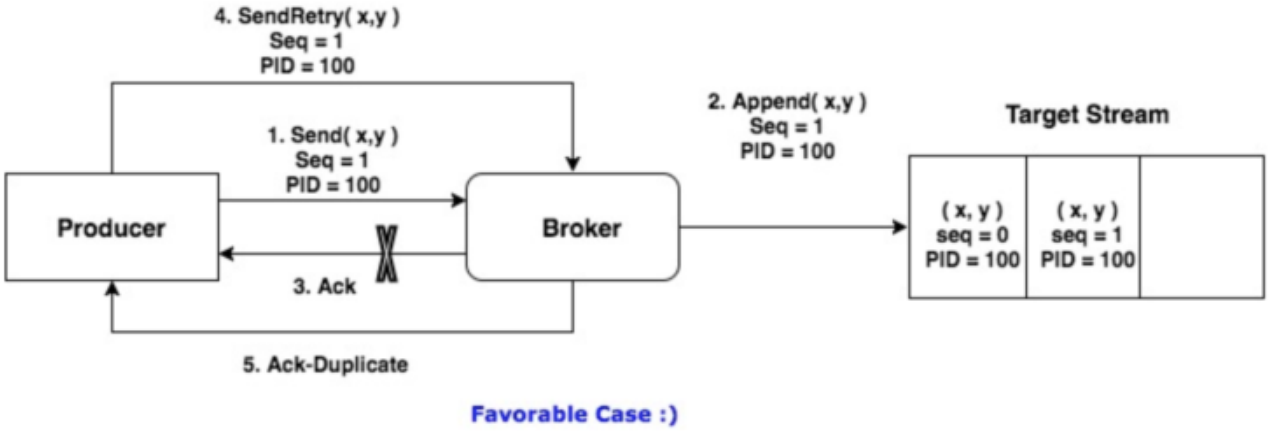
为了实现生产者的幂等性，kafka引入了 `producer ID (PID)` 和 `Sequence Number`。

- PID。每个新的Producer在初始化的时候会被分配一个唯一的PID，这个PID对用户是不可见的。
- Sequence Numbler。（对于每个PID，该Producer发送数据的每个都对应一个从0开始单调递增的Sequence Number。

Broker端在缓存中保存了这seq number，对于接收的每条消息，如果其序号比Broker缓存中序号大于1则接受它，否则将其丢弃。这样就可以实现了消息重复提交了。

```
只能保证单个Producer对于同一个<Topic, Partition>的Exactly Once语义。
不能保证同一个Producer一个topic不同的partion幂等。
```





幂等性实现：

需要将`enable.idempotence = true`,此时就会默认把acks设置为all，所以不需要再设置acks属性了。

```
Properties props = new Properties();
// java中将Prop加上该设置
props.put("enable.idempotence",true);
// Set acknowledgements for producer requests.可不再设置
props.put("acks", "all");
```

2.kafka的事务

使用kafka的事务api时的一些注意事项：

- 需要消费者的自动模式设置为false,并且不能子再手动的进行执行consumer#commitSync或者consumer#commitAsyc
- 生产者配置transaction.id属性

- 生产者不需要再配置enable.idempotence，因为如果配置了transaction.id，则此时enable.idempotence会被设置为true
- 消费者需要配置Isolation.level。在`consumer-trnasform-producer`模式下使用事务时，必须设置为READ_COMMITTED。`consumer-trnasform-producer是指从kafka获取数据后在发送到下一个kafka topic`

（1）只有写

```
public void run() {
    Properties config = new Properties();
    config.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "172.20.3.151:9092");

    // 设置事务id
    config.put("transactional.id", "first-transactional");
    // 设置幂等性
    config.put("enable.idempotence", true);
    config.put("acks", "all");

    config.put("retries", 2);                 // 重试次数
    config.put("batch.size", 100);            // 批量发送大小
    config.put("buffer.memory", 33554432);    // 缓存大小，根据本机内存大小配置
    config.put("linger.ms", 5000);            // 发送频率，满足任务一个条件发送
    config.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    config.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

    // 初始化
    Producer<String, String> producer = new KafkaProducer<>(config);

    producer.initTransactions();
    try {
        producer.beginTransaction();
        for (int i = 0; i < 2; i++)
            producer.send(new ProducerRecord<>("trans-topic",
                                               Integer.toString(i), // 作为数据路由到哪个partition
                                               "first message:" + Integer.toString(i)));
      // 加上这个总是发送成功。具体问题不明。会不会和linger.ms设置的发送频率有关(5000)
      // Thread.sleep(10000);
        int i = 10 / 0;  // 异常
        producer.commitTransaction();
    } catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
        producer.close();
    } catch (KafkaException e) {
        producer.abortTransaction();
    } catch (Exception e) {
        System.out.println("发生异常");
        producer.abortTransaction();
    }
    producer.close();
}
```

```
ProducerRecord(topic, partition, key, value)
ProducerRecord(topic, key, value)
ProducerRecord(topic, value)

<1> 若指定Partition ID,则PR（ProducerRecord）被发送至指定Partition
<2> 若未指定Partition ID,但指定了Key, PR会按照hasy(key)发送至对应Partition
<3> 若既未指定Partition ID也没指定Key，PR会按照round-robin模式发送到每个Partition
<4> 若同时指定了Partition ID和Key, PR只会发送到指定的Partition (Key不起作用，代码逻辑决定)
```

（2）读完再写：consumer-trnasform-producer

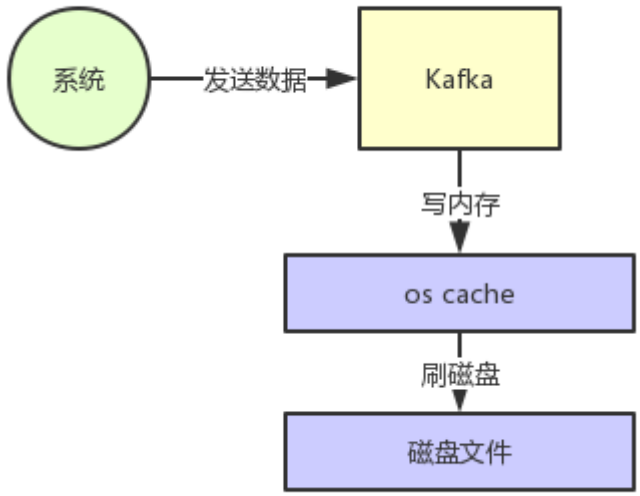
```
public void consumeTransferProduce() {
    // 1.构建上产者
    Producer producer = new KafkaProducer<>(new Properties());
    // 2.初始化事务(生成productId),对于一个生产者,只能执行一次初始化事务操作
    producer.initTransactions();
    // 3.构建消费者和订阅主题
    Consumer consumer = new KafkaConsumer<String, String>(new Properties());
    consumer.subscribe(Arrays.asList("test"));
    while (true) {
        // 4.开启事务
        producer.beginTransaction();
        // 5.1 接受消息
        ConsumerRecords<String, String> records = consumer.poll(500);
        try {
            // 5.2 do业务逻辑;
            System.out.println("customer Message---");
            Map<TopicPartition, OffsetAndMetadata> commits = new HashMap<>();

            for (ConsumerRecord<String, String> record : records) {
                // 5.2.1 读取消息,并处理消息。print the offset,key and value for the consumer
records.
                System.out.printf("offset = %d, key = %s, value = %s\n",
                        record.offset(), record.key(), record.value());
                // 5.2.2 记录提交的偏移量
                commits.put(new TopicPartition(record.topic(), record.partition()),
                            new OffsetAndMetadata(record.offset()));
                // 6.生产新的消息。比如卖订单状态的消息,如果订单成功,则需要发送跟商家结转消息或者派送员
的提成消息
                producer.send(new ProducerRecord<String, String>("test", "data2"));
            }
            // 7.提交偏移量，第二个参数为groupId
            producer.sendOffsetsToTransaction(commits, "group0323");
            // 8.事务提交
            producer.commitTransaction();
        } catch (Exception e) {
            // 7.放弃事务
            producer.abortTransaction();
        }
    }
}
```

3.kafka如何实现每秒几万、几十万的并发写入

Kafka是基于操作系统的页缓存page cache来实现文件写入的。

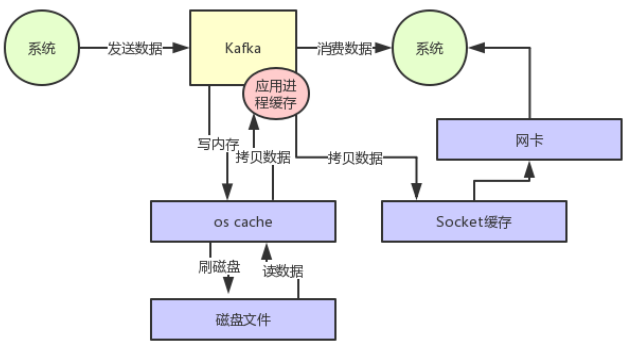
> 操作系统本身有一层缓存，叫做page cache，是在内存里的缓存，也可以称之为os cache，意思就是操作系统自己管理的缓存。
> page cache 是Linux操作系统的一个特色，其中存储的数据在I/O完成后并不回收，而是一直保存在内存中，除非内存紧张，才开始回收占用的内存。
>
> 写入磁盘文件的时候，可以直接写入这个os cache里，也就是仅仅写入内存中，接下来由操作系统自己决定什么时候把os cache里的数据真的刷入磁盘文件中。

接着另外一个就是kafka写数据的时候，非常关键的一点，他是以磁盘顺序写的方式来写的。也就是说，仅仅将数据追加到文件的末尾，不是在文件的随机位置来修改数据。

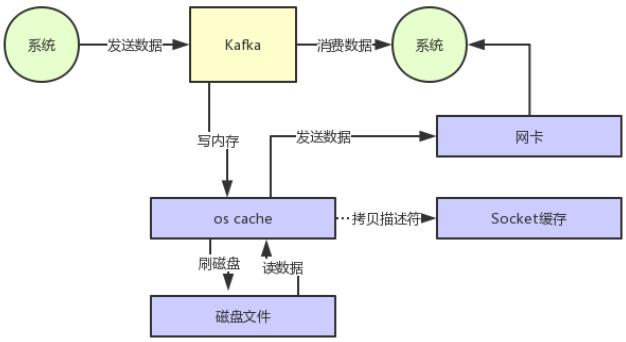


> Kafka在写数据的时候：
> （1）基于了os层面的page cache来写数据，所以性能很高，本质就是在写内存罢了。
> （2）采用磁盘顺序写的方式，所以即使数据刷入磁盘的时候，性能也是极高的，也跟写内存是差不多的。没有磁盘寻址的

4.kafka读取数据



一般读取磁盘文件流程



基于zero_copy的文件读取

通过零拷贝技术，就不需要把os cache里的数据拷贝到应用缓存，再从应用缓存拷贝到Socket缓存了，两次拷贝都省略了，所以叫做零拷贝(在kernel中zero-copy)。

对Socket缓存仅仅就是拷贝数据的描述符过去，然后数据就直接从os cache中发送到网卡上去了，这个过程大大的提升了数据消费时读取文件数据的性能。

而且大家会注意到，在从磁盘读数据的时候，会先看看os cache内存中是否有，如果有的话，其实读数据都是直接读内存的。

如果kafka集群经过良好的调优，大家会发现大量的数据都是直接写入os cache中，然后读数据的时候也是从os cache中读。

相当于是Kafka完全基于内存提供数据的写和读了，所以这个整体性能会极其的高。

5.kafka的数据不丢失与ISR机制

每个Partition有一个leader与多个follower，producer往某个Partition中写入数据是，只会往leader中写入数据，然后数据才会被复制进其他的Replica中。

> 数据是由leader push过去还是有flower pull过来？
>
> kafka是由follower周期性或者尝试去pull(拉)过来(其实这个过程与consumer消费过程非常相似)，写是都往leader上写，但是读并不是任意flower上读都行，读也只在leader上读，flower只是数据的一个备份，保证leader被挂掉后顶上来，并不往外提供服务。

数据写入leader后，没有同步到replica 中，此时leader挂了 => 数据丢失

kafka有ISR机制，数据写入leader后，需要有一定数量的replica复制成功，才返回给producer写入成功。否则重试。

6. ISR（In-Sync-Replica）

http://www.importnew.com/25247.html?from=singlemessage

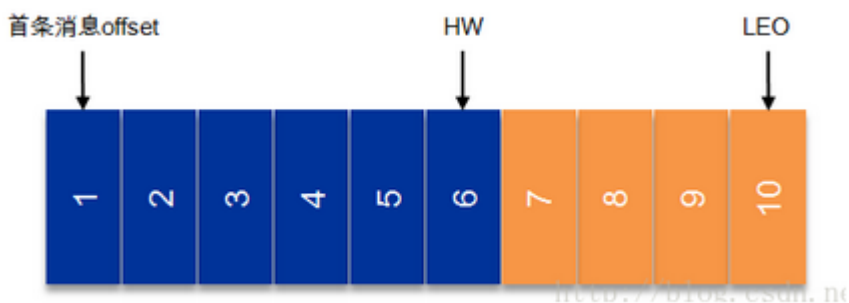
AR：Assigned-Replicas分配的副本数

ISR：是指副本同步队列数

OSR：Outof-Sync-Replica

kafka通过replica.lag.time.max.ms延迟时间作为ISR副本管理的参数，即follower从leader同步数据的时间超过该值，该follower就被踢出ISR到OSR队列中。

=> Ar = ISR + OSR

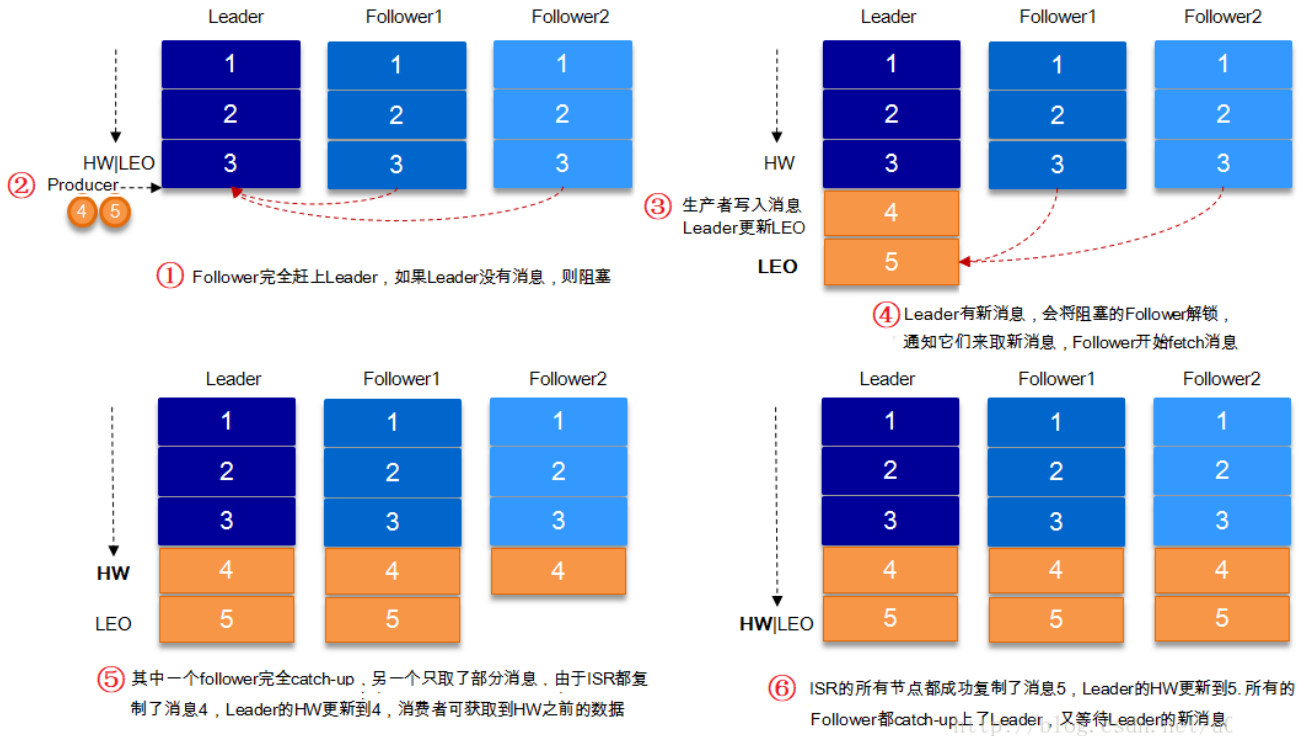


HW：HighWatermark高水位。指consumer能够看到此partition的位置

LEO：log-end-offset，指每个partition最后一条message的位置。

HW和LEO之间的数据属于已经写入leader，但是follower正在同步中。。。

下图详细的说明了当producer生产消息至broker后，ISR以及HW和LEO的流转过程：



同步复制： 只有所有的follower把数据拿过去后才commit，一致性好，可用性不高。
异步复制： 只要leader拿到数据立即commit，等follower慢慢去复制，可用性高，立即返回，一致性差一些。
Commit： 是指leader告诉客户端，这条数据写成功了。

```
kafka尽量保证commit后立即leader挂掉，其他flower都有该条数据。
kafka不是完全同步，也不是完全异步，是一种ISR机制：
        1. leader会维护一个与其基本保持同步的Replica列表，该列表称为ISR(in-sync Replica)，
           每个Partition都会有一个ISR，而且是由leader动态维护
        2. 如果一个flower比一个leader落后太多，或者超过一定时间未发起数据复制请求，则leader将其重ISR中移
除
        3. 当ISR中所有Replica都向Leader发送ACK时，leader才commit
```

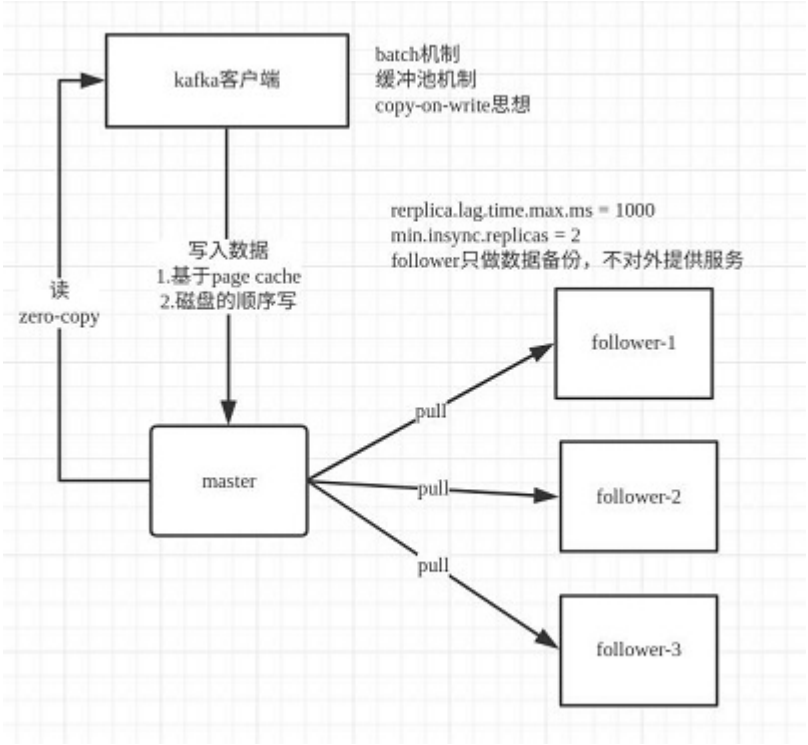
配置参数：

```
# 如果leader发现flower超过1秒没有向它发起fech请求，那么leader考虑这个flower是不是程序出了点问题
# 或者资源紧张调度不过来，它太慢了，不希望它拖慢后面的进度，就把它从ISR中移除。
rerplica.lag.time.max.ms = 1000

# 需要保证ISR中至少有多少个replica
min.insync.replicas = 1
```

所以：kafka维护了一份ISR，只有当ISR里面所有的follower都同步数据成功，才commit。

Kafka的ISR的管理最终都会反馈到Zookeeper节点上。

具体位置为：/brokers/topics/[topic]/partitions/[partition]/state。



7.数据的可靠性和持久性保证

7.1 producer发送数据时，kafka设置可靠性级别

当producer向leader发送数据时，可以通过request.required.acks参数来设置数据可靠性的级别：

- 1（默认）：这意味着producer在ISR中的leader已成功收到的数据并得到确认后发送下一条message。`如果leader宕机了，则会丢失数据`。
- 0：这意味着producer无需等待来自broker的确认而继续发送下一批消息。这种情况下数据传输效率最高，但是数据可靠性确是最低的。
- -1：producer需要等待ISR中的所有follower都确认接收到数据后才算一次发送完成，可靠性最高。**但是这样也不能保证数据不丢失**，比如当ISR中只有leader时（前面ISR那一节讲到，ISR中的成员由于某些情况会增加也会减少，最少就只剩一个leader），这样就变成了acks=1的情况。

如果要提高数据的可靠性：

```
request.required.acks = -1
min.insync.replicas = N
// N > 1, 避免只有一个leader在ISR中。并且只有当acks= -1时，该参数才生效

// 如果ISR中的副本数小于设置的值时，会抛出异常：
org.apache.kafka.common.errors.NotEnoughReplicasExceptoin: Messages are rejected since there are fewer in-sync replicas than required。
```

7.2 所有的replicas都挂了

如果某一个partition的所有replica都挂了，就无法保证数据不丢失了。这种情况下有两种可行的方案：

1. 等待ISR中任意一个replica“活”过来，并且选它作为leader
2. 选择第一个“活”过来的replica（并不一定是在ISR中）作为leader

kafka默认才是第二种方式。

```
unclean.leader.election.enable = false   // 第一种策略
unclean.leader.election.enable = true    // 第二种策略(默认)
```

8.kafka的copy-on-write机制

（1）batch机制：

kafka在发送消息时，会先将消息放到内存缓冲区，最后将其合并成一个batch。

同一个batch必须是发送到同一个topic的同一个partition的。 消息载体是ConcurrentMap<TopicPartition, Deque<RecordBatch>>结构，key值为TopicPartition

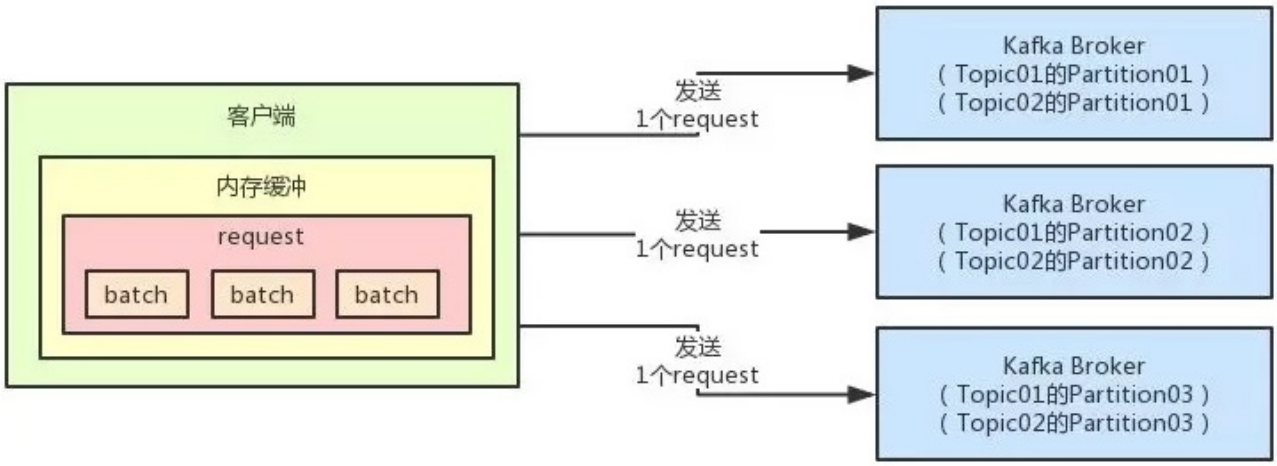
默认batch.size=16kB

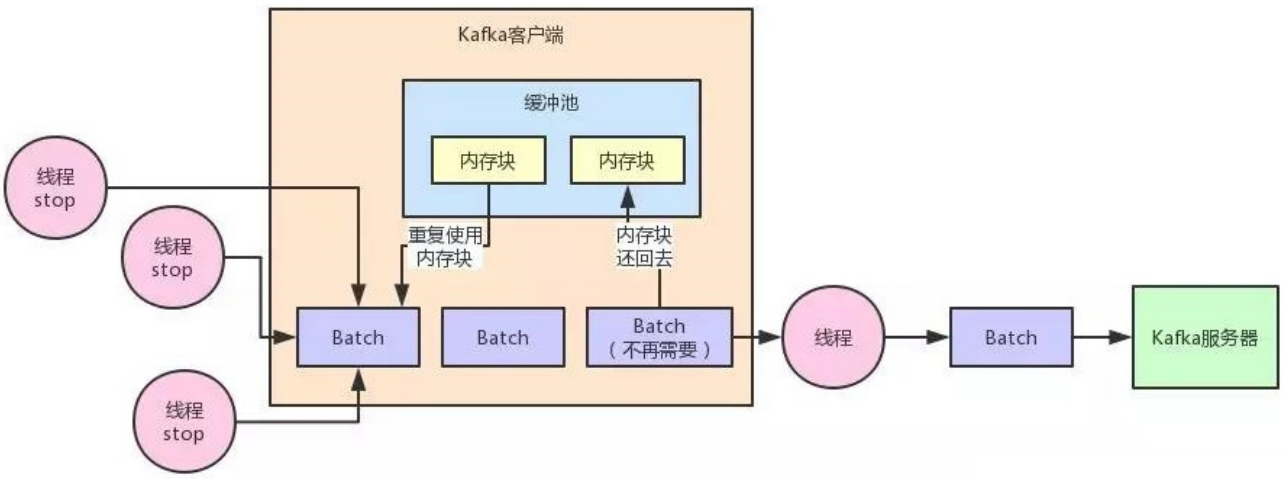
（2）request机制：

将发送往同一个broker的多个batch打包成一个request

（3）缓冲池机制

这个Batch里的数据都发送结束后，不断的清理掉已经发送成功的Batch了会导致频繁GC。因此kafka会有一个缓冲池机制，就是每个Batch底层都对应一块内存空间。数据发送结束后它不会被垃圾回收，而是放回缓冲池。





8.1、ConcurrentMap

kafka的消息载体ConcurrentMap < TopicPartition, Deque< RecordBatch>> ， 是对HashMap的封装（HashMap线程不安全，ConcurrentMap线程安全）

ConcurrentMap是读多写少的操作：

也许你会认为在不停的往Map中添加数据，然后被添加的数据又会读出被送往Server端，读写应该差不多，其实不然，该Map的get操作远远大于put操作，写操作发生于ArrayDeque的addLast操作和pollFirst操作，这些是get取到ArrayDeque之后对ArrayDeque进行的操作，并不属于对map的写操作。因此这里的Map是一个get操作远远大于put的Map。

针对读多写少的数据结构中使用读写锁最大的问题：

偶尔执行一个写操作，会有大量的读操作被阻塞。尤其当这个写操作很费时。

CopyOnWrite思想：

写操作利用copy的副本来执行。并采用volatile关键字修饰，使写线程写入结束后，读线程立刻能感知到变化。

其实是空间换时间。

kafka API实现者在这里对ConcurrentMap 封装了CopyOnWriteMap，

```
// 这个map是核心的，因为用volatile修饰了。
// 只要把最新的数组对他赋值，其他线程立马可以看到最新的数组
private volatile Map<K, V> map;

@Override
public synchronized V put(K k, V v) {
    Map<K, V> copy = new HashMap<K, V>(this.map);
    V prev = copy.put(k, v);
    this.map = Collections.unmodifiableMap(copy);
    return prev;
}

@Override
public synchronized void putAll(Map<? extends K, ? extends V> entries) {
    Map<K, V> copy = new HashMap<K, V>(this.map);
    copy.putAll(entries);
    this.map = Collections.unmodifiableMap(copy);
}
```

9.读写锁最大的问题是什么？

读多写少。

偶尔执行一个写操作的时候，会加上写锁，此时大量的读操作就会被阻塞住。

直接去掉读写锁，直接采用copyonwite思想。