二、Kafka客户端消息流转流程

-- 楼兰

这一章节将重点介绍Kafka的HighLevel API使用，并通过这些API，构建起Kafka整个消息发送以及消费的主线流程。

> Kafka提供了两套客户端API，HighLevel API和LowLevel API。 HighLevel API封装了kafka的运行细节，使用起来比较简单，是企业开发过程中最常用的客户端API。 而LowLevel API则需要客户端自己管理Kafka的运行细节，Partition，Offset这些数据都由客户端自行管理。这层API功能更灵活，但是使用起来非常复杂，也更容易出错。只在极少数对性能要求非常极致的场景才会偶尔使用。我们的重点是HighLeve API 。

一、从基础的客户端说起

Kafka提供了非常简单的客户端API。只需要引入一个Maven依赖即可：

```
<dependency>
 <groupId>org.apache.kafka</groupId>
 <artifactId>kafka_2.13</artifactId>
 <version>3.8.0</version>
</dependency>
```

1、消息发送者主流程

然后可以使用Kafka提供的Producer类，快速发送消息。

> 发送消息前，Topic需要提前创建。建议创建指令： bin/kafka-topics.sh --bootstrap-server worker1:9092 --create --topic Topic --partitions 3 --replication-factor 2
>
> 或者参见配套案例。

```
public class MyProducer {
    private static final String BOOTSTRAP_SERVERS = "worker1:9092,worker2:9092,worker3:9092";
    private static final String TOPIC = "disTopic";

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        //PART1:设置发送者相关属性
        Properties props = new Properties();
        // 此处配置的是kafka的端口
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        // 配置key的序列化类
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,"org.apache.kafka.common.serialization.StringSerializer");
        // 配置value的序列化类
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,"org.apache.kafka.common.serialization.StringSerializer");

        Producer<String,String> producer = new KafkaProducer<>(props);
        CountDownLatch latch = new CountDownLatch(5);
        for(int i = 0; i < 5; i++) {
            //Part2:构建消息
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, Integer.toString(i), "MyProducer" + i);
            //Part3:发送消息
            //单向发送：不关心服务端的应答。
            producer.send(record);
            System.out.println("message "+i+" sended");
            //同步发送：获取服务端应答消息前，会阻塞当前线程。
            RecordMetadata recordMetadata = producer.send(record).get();
            String topic = recordMetadata.topic();
            int partition = recordMetadata.partition();
            long offset = recordMetadata.offset();
            String message = recordMetadata.toString();
            System.out.println("message:["+ message+"] sended with topic:"+topic+"; partition:"+partition+ ";offset:"+offset);
            //异步发送：消息发送后不阻塞，服务端有应答后会触发回调函数
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata recordMetadata, Exception e) {
                    if(null != e){
                        System.out.println("消息发送失败,"+e.getMessage());
                        e.printStackTrace();
                    }else{
                        String topic = recordMetadata.topic();
                        long offset = recordMetadata.offset();
                        String message = recordMetadata.toString();
                        System.out.println("message:["+ message+"] sended with topic:"+topic+";offset:"+offset);
                    }
                    latch.countDown();
                }
            });
        }
        //消息处理完才停止发送者。
        latch.await();
        producer.close();
    }
}
```

整体来说，构建Producer分为三个步骤：

1. **设置Producer核心属性**： Producer可选的属性都可以由ProducerConfig类管理。比如ProducerConfig.BOOTSTRAP_SERVERS_CONFIG属性，显然就是指发送者要将消息发到哪个Kafka集群上。这是每个Producer必选的属性。在ProducerConfig中，对于大部分比较重要的属性，都配置了对应的DOC属性进行描述。
2. **构建消息**：Kafka的消息是一个Key-Value结构的消息。其中，key和value都可以是任意对象类型。其中，key主要是用来进行Partition分区的，业务上更关心的是value。
3. **使用Producer发送消息。**： 通常用到的就是单向发送、同步发送和异步发送者三种发送方式。

2、消息消费者主流程

接下来可以使用Kafka提供的Consumer类，快速消费消息。

```
public class MyConsumer {
    private static final String BOOTSTRAP_SERVERS = "worker1:9092,worker2:9092,worker3:9092";
    private static final String TOPIC = "disTopic";

    public static void main(String[] args) {
        //PART1:设置发送者相关属性
        Properties props = new Properties();
        //kafka地址
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        //每个消费者要指定一个group
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "test");
        //key序列化类
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringDeserializer");
        //value序列化类
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringDeserializer");
        Consumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList(TOPIC));
        while (true) {
            //PART2:拉取消息
            // 100毫秒超时时间
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofNanos(100));
            //PART3:处理消息
            for (ConsumerRecord<String, String> record : records) {
                System.out.println("offset = " + record.offset() + ";key = " + record.key() + "; value= " + record.value());
            }
            //提交offset，消息就不会重复推送。
            consumer.commitSync(); //同步提交，表示必须等到offset提交完毕，再去消费下一批数据。
//            consumer.commitAsync(); //异步提交，表示发送完提交offset请求后，就开始消费下一批数据了。不用等到Broker的确认。
        }
    }
}
```

整体来说，Consumer同样是分为三个步骤：

1. **设置Consumer核心属性**：可选的属性都可以由ConsumerConfig类管理。在这个类中，同样对于大部分比较重要的属性，都配置了对应的DOC属性进行描述。同样BOOTSTRAP_SERVERS_CONFIG是必须设置的属性。
2. **拉取消息**：Kafka采用Consumer主动拉取消息的Pull模式。consumer主动从Broker上拉取一批感兴趣的消息。
3. **处理消息，提交位点**：消费者将消息拉取完成后，就可以交由业务自行处理对应的这一批消息了。只是消费者需要向Broker提交偏移量offset。如果不提交Offset，Broker会认为消费者端消息处理失败了，还会重复进行推送。

Kafka的客户端基本就是固定的按照这三个大的步骤运行。在具体使用过程中，最大的变数基本上就是给生产者和消费者的设定合适的属性。这些属性极大的影响了客户端程序的执行方式。

> 改改配置就学会Kafka了？kafka官方配置：https://kafka.apache.org/documentation/#configuration。看看你晕不晕。

二、从客户端属性来梳理客户端工作机制

其实Kafka的设计精髓，是在网络不稳定，服务也随时会崩溃的这些作死的复杂场景下，如何保证消息的高并发、高吞吐，那才是Kafka最为精妙的地方。但是要理解那些复杂的问题，都是需要建立在这个基础模型基础上的。

1、消费者分组消费机制

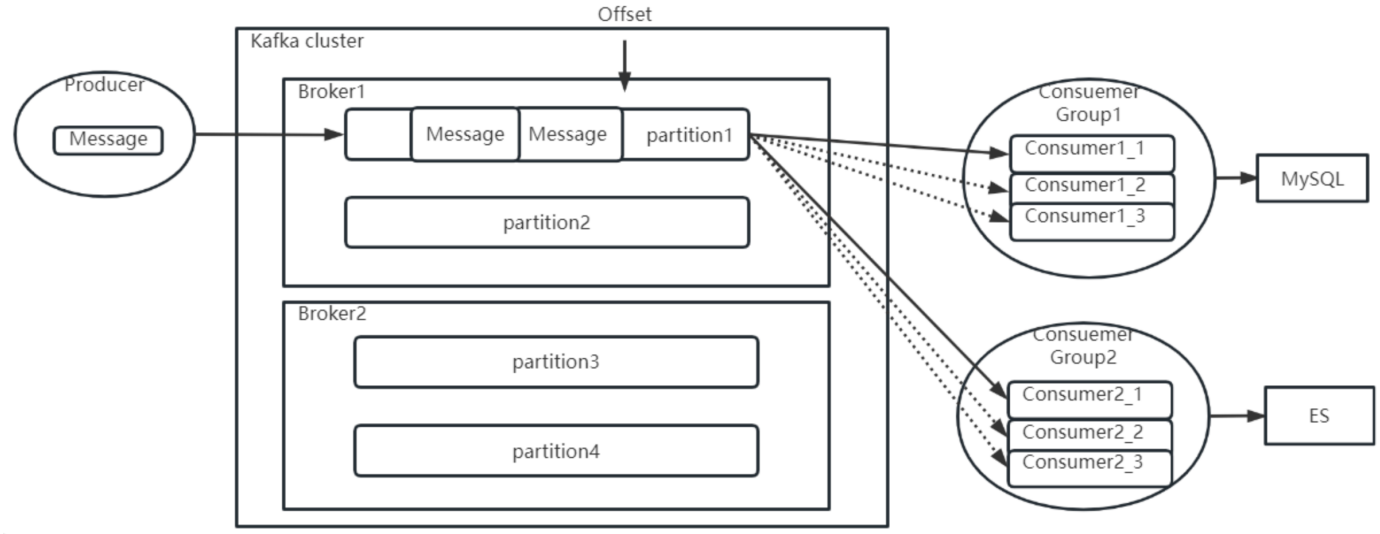
这是我们在使用kafka时，最为重要的一个机制，因此最先进行梳理。

在Consumer中，都需要指定一个GROUP_ID_CONFIG属性，这表示当前Consumer所属的消费者组。他的描述是这样的：

```
  public static final String GROUP_ID_CONFIG = "group.id";
    public static final String GROUP_ID_DOC = "A unique string that identifies the consumer group this consumer belongs to.
This property is required if the consumer uses either the group management functionality by using <code>subscribe(topic)
</code> or the Kafka-based offset management strategy.";
```

> 既然这里提到了kafka-based offset management strategy，那是不是也有非Kafka管理Offset的策略呢？
>
> 另外，还有一个相关的参数GROUP_INSTANCE_ID_CONFIG，可以给组成员设置一个固定的instanceId，这个参数通常可以用来减少Kafka不必要的rebalance。

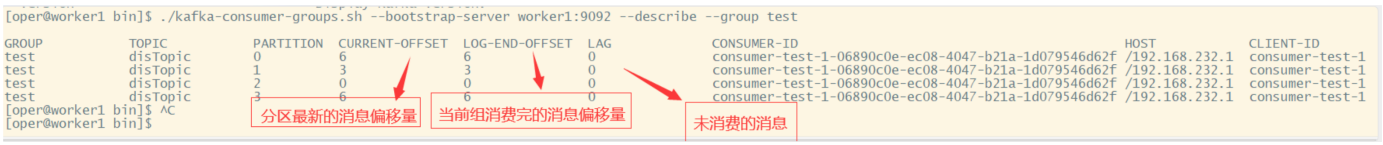
从这段描述中看到，对于Consumer，如果需要在subcribe时使用组管理功能以及Kafka提供的offset管理策略，那就必须要配置GROUP_ID_CONFIG属性。这个分组消费机制简单描述就是这样的：



生产者往Topic下发消息时，会尽量均匀的将消息发送到Topic下的各个Partition当中。而这个消息，会向所有订阅了该Topic的消费者推送。推送时，每个ConsumerGroup中只会推送一份。也就是同一个消费者组中的多个消费者实例，只会共同消费一个消息副本。而不同消费者组之间，会重复消费消息副本。这就是消费者组的作用。

与之相关的还有Offset偏移量。这个偏移量表示每个消费者组在每个Partiton中已经消费处理的进度。在Kafka中，可以看到消费者组的Offset记录情况。

```
[oper@worker1 bin]$ ./kafka-consumer-groups.sh --bootstrap-server worker1:9092 --describe --group test
```



这个Offset偏移量，需要消费者处理完成后主动向Kafka的Broker提交。提交完成后，Broker就会更新消费进度，表示这个消息已经被这个消费者组处理完了。但是如果消费者没有提交Offset，Broker就会认为这个消息还没有被处理过，就会重新往对应的消费者组进行推送，不过这次，一般会尽量推送给同一个消费者组当中的其他消费者实例。

在示例当中，是通过业务端主动调用Consumer的commitAsync方法或者commitSync方法主动提交的，Kafka中自然也提供了自动提交Offset的方式。使用自动提交，只需要在Comsumer中配置ENABLE_AUTO_COMMIT_CONFIG属性即可。

```
 public static final String ENABLE_AUTO_COMMIT_CONFIG = "enable.auto.commit";
    private static final String ENABLE_AUTO_COMMIT_DOC = "If true the consumer's offset will be periodically committed in the background.";
```

从这里可以看到，Offset是Kafka进行消息推送控制的关键之处。这里需要思考两个问题：

一、Offset是根据Group、Partition分开记录的。消费者如果一个Partition对应多个Consumer消费者实例，那么每个Consumer实例都会往Broker提交同一个Partition的不同Offset，这时候Broker要听谁的？所以一个Partition最多只能同时被一个Consumer消费。也就是说，示例中四个Partition的Topic，那么同一个消费者组中最多就只能配置四个消费者实例。

二、这么关键的Offset数据，保存在Broker端，但是却是由"不靠谱"的消费者主导推进，这显然是不够安全的。那么应该如何提高Offset数据的安全性呢？如果你有兴趣自己观察，会发现在Consumer中，实际上也提供了AUTO_OFFSET_RESET_CONFIG参数，来指定消费者组在服务端的Offset不存在时如何进行后续消费。（有可能服务端初始化Consumer Group的Offset失败，也有可能Consumer Group当前的Offset对应的数据文件被过期删除了。）这就相当于服务端做的兜底保障。

> ConsumerConfig.AUTO_OFFSET_RESEWT_CONFIG： 当Server端没有对应的Offset时，要如何处理。 可选项：
>
> - earliest： 自动设置为当前最早的offset
> - latest：自动设置为当前最晚的offset
> - none： 如果消费者组对应的offset找不到，就向Consumer抛异常。
> - 其他选项： 向Consumer抛异常。

有了服务端兜底后，消费者应该要如何保证offset的安全性呢？有两种方式：一种是异步提交。就是消费者在处理业务的同时，异步向Broker提交Offset。这样好处是消费者的效率会比较高，但是如果消费者的消息处理失败了，而offset又成功提交了。这就会造成消息丢失。另一种方式是同步提交。消费者保证处理完所有业务后，再提交Offset。这样的好处自然是消息不会因为offset丢失了。因为如果业务处理失败，消费者就可以不去提交Offset，这样消息还可以重试。但是坏处是消费者处理信息自然就慢了。另外还会产生消息重复。因为Broker端不可能一直等待消费者提交。如果消费者的业务处理时间比较长，这时在消费者正常处理消息的过程中，Broker端就已经等不下去了，认为这个消费者处理失败了。这时就会往同组的其他消费者实例投递消息，这就造成了消息重复处理。

这时，如果采取头疼医头，脚疼医脚的方式，当然都有对应的办法。但是都会显得过于笨重。其实这类问题的根源在于Offset反映的是消息的处理进度。而消息处理进度跟业务的处理进度又是不同步的。所有我们可以换一种思路，将Offset从Broker端抽取出来，放到第三方存储比如Redis里自行管理。这样就可以自己控制用业务的处理进度推进Offset往前更新。

2、生产者拦截器机制

生产者拦截机制允许客户端在生产者在消息发送到Kafka集群之前，对消息进行拦截，甚至可以修改消息内容。

这涉及到Producer中指定的一个参数：INTERCEPTOR_CLASSES_CONFIG

```
  public static final String INTERCEPTOR_CLASSES_CONFIG = "interceptor.classes";
  public static final String INTERCEPTOR_CLASSES_DOC = "A list of classes to use as interceptors. "
                                                       + "Implementing the
<code>org.apache.kafka.clients.producer.ProducerInterceptor</code> interface allows you to intercept (and possibly mutate) the
records "
                                                       + "received by the producer before they are published to the Kafka
cluster. By default, there are no interceptors.";
```

于是，按照他的说明，我们可以定义一个自己的拦截器实现类：

```
public class MyInterceptor implements ProducerInterceptor {
    //发送消息时触发
    @Override
    public ProducerRecord onSend(ProducerRecord producerRecord) {
        System.out.println("prudocerRecord : " + producerRecord.toString());
        return producerRecord;
    }

    //收到服务端响应时触发
    @Override
    public void onAcknowledgement(RecordMetadata recordMetadata, Exception e) {
        System.out.println("acknowledgement recordMetadata:"+recordMetadata.toString());
    }

    //连接关闭时触发
    @Override
    public void close() {
        System.out.println("producer closed");
    }

    //整理配置项
    @Override
    public void configure(Map<String, ?> map) {
        System.out.println("=====config start======");
        for (Map.Entry<String, ?> entry : map.entrySet()) {
            System.out.println("entry.key:"+entry.getKey()+" === entry.value: "+entry.getValue());
        }
        System.out.println("=====config end======");
    }
}
```

然后在生产者中指定拦截器类（多个拦截器类，用逗号隔开）

```
 props.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,"com.roy.kfk.basic.MyInterceptor");
```

拦截器机制一般用得比较少，主要用在一些统一添加时间等类似的业务场景。比如，用Kafka传递一些POJO，就可以用拦截器统一添加时间属性。但是我们平常用Kafka传递的都是String类型的消息，POJO类型的消息，Kafka可以传吗？这就要用到下面的消息序列化机制。

3、消息序列化机制

在之前的简单示例中，Producer指定了两个属性KEY_SERIALIZER_CLASS_CONFIG和VALUE_SERIALIZER_CLASS_CONFIG，对于这两个属性，在ProducerConfig中都有配套的说明属性。

```
 public static final String KEY_SERIALIZER_CLASS_CONFIG = "key.serializer";
    public static final String KEY_SERIALIZER_CLASS_DOC = "Serializer class for key that implements the
<code>org.apache.kafka.common.serialization.Serializer</code> interface.";
    public static final String VALUE_SERIALIZER_CLASS_CONFIG = "value.serializer";
    public static final String VALUE_SERIALIZER_CLASS_DOC = "Serializer class for value that implements the
<code>org.apache.kafka.common.serialization.Serializer</code> interface.";
```

通过这两个参数，可以指定消息生产者如何将消息的key和value序列化成二进制数据。在Kafka的消息定义中，key和value的作用是不同的。

- key是用来进行分区的可选项。Kafka通过key来判断消息要分发到哪个Partition。

如果没有填写key，那么Kafka会自动选择Partition。

如果填写了key，那么会通过声明的Serializer序列化接口，将key转换成一个byte[]数组，然后对key进行hash，选择Partition。这样可以保证key相同的消息会分配到相同的Partition中。

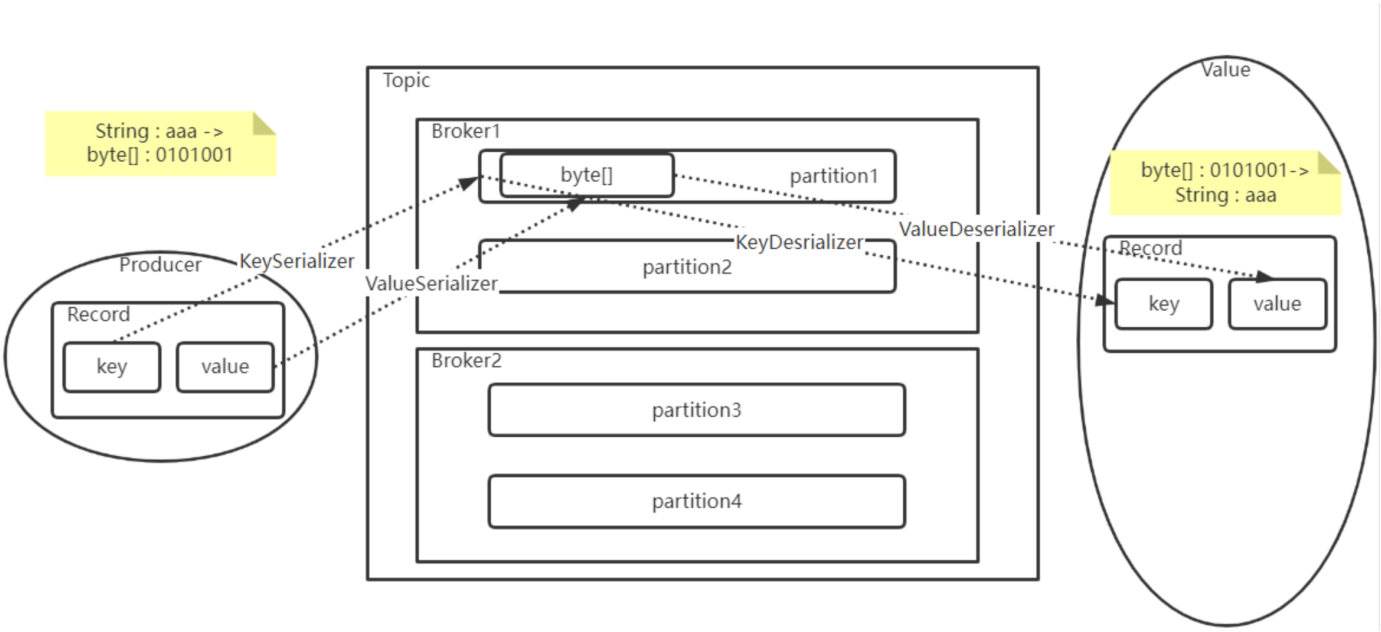
- Value是业务上比较关心的消息。Kafka同样需要将Value对象通过Serializer序列化接口，将Key转换成byte[]数组，这样才能比较好的在网络上传输Value信息，以及将Value信息落盘到操作系统的文件当中。

生产者要对消息进行序列化，那么消费者拉取消息时，自然需要进行反序列化。所以，在Consumer中，也有反序列化的两个配置

```
  public static final String KEY_DESERIALIZER_CLASS_CONFIG = "key.deserializer";
    public static final String KEY_DESERIALIZER_CLASS_DOC = "Deserializer class for key that implements the
<code>org.apache.kafka.common.serialization.Deserializer</code> interface.";
    public static final String VALUE_DESERIALIZER_CLASS_CONFIG = "value.deserializer";
    public static final String VALUE_DESERIALIZER_CLASS_DOC = "Deserializer class for value that implements the
<code>org.apache.kafka.common.serialization.Deserializer</code> interface.";
```

在Kafka中，对于常用的一些基础数据类型，都已经提供了对应的实现类。但是，如果需要使用一些自定义的消息格式，比如自己定制的POJO，就需要定制具体的实现类了。

在自己进行序列化机制时，需要考虑的是如何用二进制来描述业务数据。例如对于一个通常的POJO类型，可以将他的属性拆分成两种类型：**一种类型是定长的基础类型**，比如Integer,Long,Double等。这些基础类型转化成二进制数组都是定长的。这类属性可以直接转成序列化数组，在反序列化时，只要按照定长去读取二进制数据就可以反序列化了。**另一种是不定长的浮动类型**，比如String，或者基于String的JSON类型等。这种浮动类型的基础数据转化成二进制数组，长度都是不一定的。对于这类数据，通常的处理方式都是先往二进制数组中写入一个定长的数据的长度数据(Integer或者Long类型)，然后再继续写入数据本身。这样，反序列化时，就可以先读取一个定长的长度，再按照这个长度去读取对应长度的二进制数据，这样就能读取到数据的完整二进制内容。



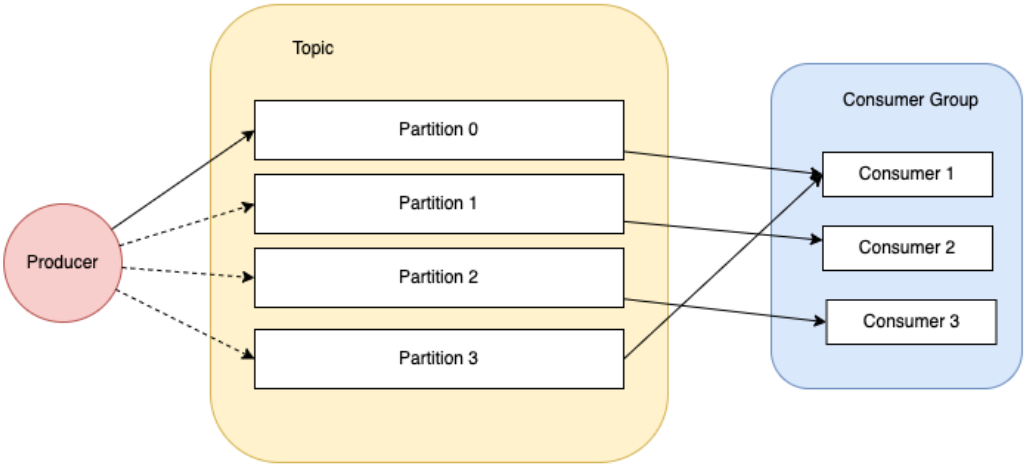
** 渔与鱼** 序列化机制是在高并发场景中非常重要的一个优化机制。高效的系列化实现能够极大的提升分布式系统的网络传输以及数据落盘的能力。例如对于一个User对象，即可以使用JSON字符串这种简单粗暴的序列化方式，也可以选择按照各个字段进行组合序列化的方式。但是显然后者的占用空间比较小，序列化速度也会比较快。而Kafka在文件落盘时，也设计了非常高效的数据序列化实现，这也是Kafka高效运行的一大支撑。

在很多其他业务场景中，也需要我们提供更高效的序列化实现。例如使用MapReduce框架时，就需要自行定义数据的序列化方式。使用Netty框架进行网络调用时，为了防止粘包，也需要定制数据的序列化机制。在这些场景下，进行序列化的基础思想，和我们这里介绍的也是一样的。当然，如果我们可以进一步设计出更简短高效的数据压缩算法，那也就能更进一步提高数据传输的效率。比如对二进制数据进行压缩。而这就是算法最直接的作用。

4、消息分区路由机制

了解前面两个机制后，你自然会想到一个问题。就是消息如何进行路由？也即是两个相关联的问题。

- Producer会根据消息的key选择Partition，具体如何通过key找Partition呢?
- 一个消费者组会共同消费一个Topic下的多个Partition中的同一套消息副本，那Consumer节点是不是可以决定自己消费哪些Partition的消息呢?



这两个问题其实都不难，你只要在几个Config类中稍微找一找就能找到答案。

首先，在Producer中，可以指定一个Partitioner来对消息进行分配。

```
 public static final String PARTITIONER_CLASS_CONFIG = "partitioner.class";
    private static final String PARTITIONER_CLASS_DOC = "A class to use to determine which partition to be send to when
produce the records. Available options are:" +
        "<ul>" +
            "<li>If not set, the default partitioning logic is used. " +
        "This strategy will try sticking to a partition until at least " + BATCH_SIZE_CONFIG + " bytes is produced to the
partition. It works with the strategy:" +
                "<ul>" +
                    "<li>If no partition is specified but a key is present, choose a partition based on a hash of the
key</li>" +
                    "<li>If no partition or key is present, choose the sticky partition that changes when at least " +
BATCH_SIZE_CONFIG + " bytes are produced to the partition.</li>" +
                "</ul>" +
            "</li>" +
            "<li><code>org.apache.kafka.clients.producer.RoundRobinPartitioner</code>: This partitioning strategy is that " +
        "each record in a series of consecutive records will be sent to a different partition(no matter if the 'key' is
provided or not), " +
        "until we run out of partitions and start over again. Note: There's a known issue that will cause uneven distribution
when new batch is created. " +
        "Please check KAFKA-9965 for more detail." +
            "</li>" +
        "</ul>" +
        "<p>Implementing the <code>org.apache.kafka.clients.producer.Partitioner</code> interface allows you to plug in a
custom partitioner.";
```

这里就说明了Kafka是通过一个Partitioner接口的具体实现来决定一个消息如何根据Key分配到对应的Partition上的。你甚至可以很简单的实现一个自己的分配策略。

在之前的3.2.0版本，Kafka提供了三种默认的Partitioner实现类，RoundRobinPartitioner，DefaultPartitioner和UniformStickyPartitioner。目前后面两种实现已经标记为过期，被替换成了默认的实现机制。

对于生产者，默认的Sticky策略在给一个生产者分配了一个分区后，会尽可能一直使用这个分区。等待该分区的batch.size(默认16K)已满，或者这个分区的消息已完成 linger.ms(默认0毫秒，表示如果batch.size迟迟没有满后的等待时间)。RoundRobinPartitioner是在各个Partition中进行轮询发送，这种方式没有考虑到消息大小以及各个Broker性能差异，用得比较少。

另外可以自行指定一个Partitioner实现类，定制分区逻辑。在Partitioner接口中，核心要实现的就是partition方法。根据相关信息，选择一个Partition。比如用key对partition的个数取模之类的。而Topic下的所有Partition信息都在cluster参数中。

```
//获取所有的Partition信息。
List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
```

然后，在Consumer中，可以指定一个PARTITION_ASSIGNMENT_STRATEGY分区分配策略，决定如何在多个Consumer实例和多个Partitioner之间建立关联关系。

```
  public static final String PARTITION_ASSIGNMENT_STRATEGY_CONFIG ="partition.assignment.strategy";
    private static final String PARTITION_ASSIGNMENT_STRATEGY_DOC = "A list of class names or class types, " +
        "ordered by preference, of supported partition assignment strategies that the client will use to distribute " +
        "partition ownership amongst consumer instances when group management is used. Available options are:" +
        "<ul>" +
        "<li><code>org.apache.kafka.clients.consumer.RangeAssignor</code>: Assigns partitions on a per-topic basis.</li>" +
        "<li><code>org.apache.kafka.clients.consumer.RoundRobinAssignor</code>: Assigns partitions to consumers in a round-
robin fashion.</li>" +
        "<li><code>org.apache.kafka.clients.consumer.StickyAssignor</code>: Guarantees an assignment that is " +
        "maximally balanced while preserving as many existing partition assignments as possible.</li>" +
        "<li><code>org.apache.kafka.clients.consumer.CooperativeStickyAssignor</code>: Follows the same StickyAssignor " +
        "logic, but allows for cooperative rebalancing.</li>" +
        "</ul>" +
        "<p>The default assignor is [RangeAssignor, CooperativeStickyAssignor], which will use the RangeAssignor by default,
" +
        "but allows upgrading to the CooperativeStickyAssignor with just a single rolling bounce that removes the
RangeAssignor from the list.</p>" +
        "<p>Implementing the <code>org.apache.kafka.clients.consumer.ConsumerPartitionAssignor</code> " +
        "interface allows you to plug in a custom assignment strategy.</p>";
```

同样，Kafka内置了一些实现方式，在通常情况下也都是最优的选择。你也可以实现自己的分配策略。

从上面介绍可以看到Kafka默认提供了三种消费者的分区分配策略

- range策略： 比如一个Topic有10个Partiton(partition 0-9) 一个消费者组下有三个Consumer(consumer1-3)。Range策略就会将分区0-3分给一个Consumer，4-6给一个Consumer，7-9给一个Consumer。
- round-robin策略：轮询分配策略，可以理解为在Consumer中一个一个轮流分配分区。比如0，3，6，9分区给一个Consumer1，1，4，7分区给一个Consumer2，然后2，5，8给一个Consumer3
- sticky策略：粘性策略。这个策略有两个原则：
  - 1、在开始分区时，尽量保持分区的分配均匀。比如按照Range策略分(这一步实际上是随机的)。
  - 2、分区的分配尽可能的与上一次分配的保持一致。比如在range分区的情况下，第三个Consumer的服务宕机了，那么按照sticky策略，就会保持consumer1和consumer2原有的分区分配情况。然后将consumer3分配的7~9分区尽量平均的分配到另外两个consumer上。这种粘性策略可以很好的保持Consumer的数据稳定性。

另外可以通过继承AbstractPartitionAssignor抽象类自定义消费者的订阅方式。

官方默认提供的生产者端的默认分区器以及消费者端的RangeAssignor+CooperativeStickyAssignor分配策略，在大部分场景下都是非常高效的算法。深入理解这些算法，对于你深入理解MQ场景，以及借此去横向对比理解其他的MQ产品，都是非常有帮助的。

那么在哪些场景下我们可以自己来定义分区器呢？例如如果在部署消费者时，如果我们的服务器配置不一样，就可以通过定制消费者分区器，让性能更好的服务器上的消费者消费较多的消息，而其他服务器上的消费者消费较少的消息，这样就能更合理的运用上消费者端的服务器性能，提升消费者的整体消费速度。

5、生产者消息缓存机制

接下来就是如何具体发送消息了。

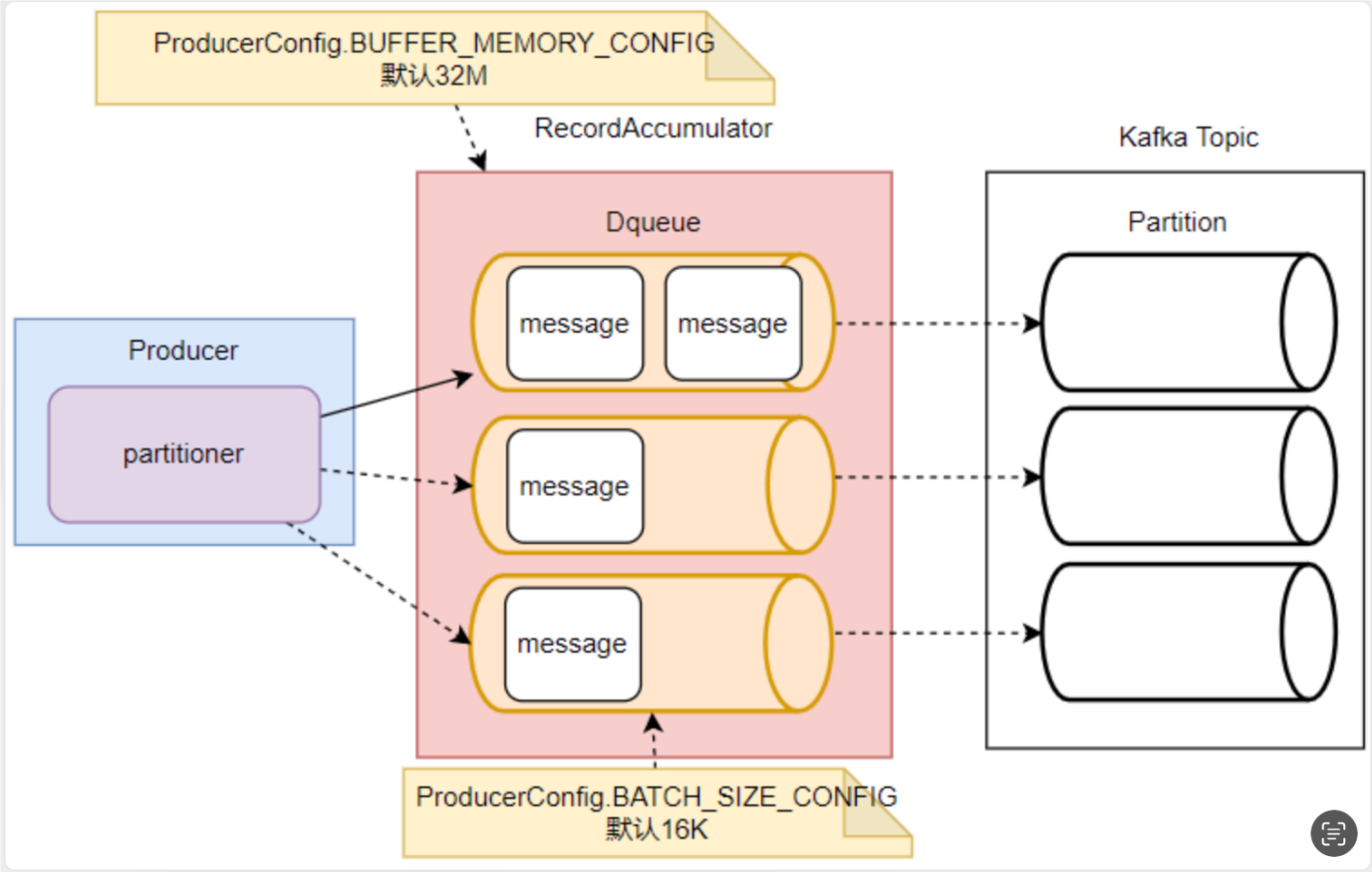
Kafka生产者为了避免高并发请求对服务端造成过大压力，每次发消息时并不是一条一条发往服务端，而是增加了一个高速缓存，将消息集中到缓存后，批量进行发送。这种缓存机制也是高并发处理时非常常用的一种机制。

Kafka的消息缓存机制涉及到KafkaProducer中的两个关键组件： accumulator 和 sender

```
//1.记录累加器
int batchSize = Math.max(1, config.getInt(ProducerConfig.BATCH_SIZE_CONFIG));
this.accumulator
= new RecordAccumulator(logContext,batchSize,this.compressionType,lingerMs(config),retryBackoffMs,deliveryTimeoutMs,
partitionerConfig,metrics,PRODUCER_METRIC_GROUP_NAME,time,apiVersions,transactionManager,new BufferPool(this.totalMemorySize,
batchSize, metrics, time, PRODUCER_METRIC_GROUP_NAME));
//2. 数据发送线程
this.sender = newSender(logContext, kafkaClient, this.metadata);
```

其中RecordAccumulator，就是Kafka生产者的消息累加器。KafkaProducer要发送的消息都会在ReocrdAccumulator中缓存起来，然后再分批发送给kafka broker。

在RecordAccumulator中，会针对每一个Partition，维护一个Deque双端队列，这些Dequeue队列基本上是和Kafka服务端的Topic下的Partition对应的。每个Dequeue里会放入若干个ProducerBatch数据。KafkaProducer每次发送的消息，都会根据key分配到对应的Deque队列中。然后每个消息都会保存在这些队列中的某一个ProducerBatch中。而消息分发的规则，就是由上面的Partitioner组件完成的。

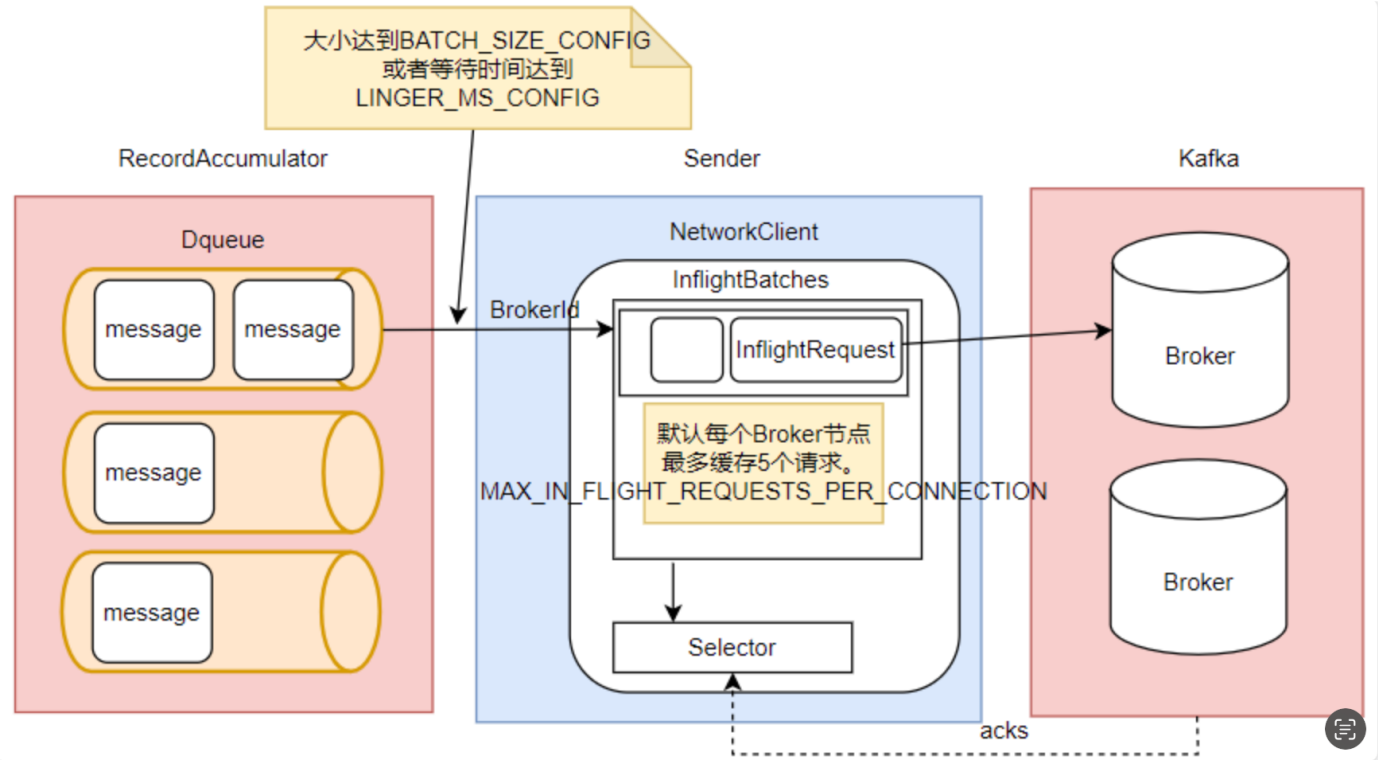


这里主要涉及到两个参数

```
//RecordAccumulator缓冲区大小
public static final String BUFFER_MEMORY_CONFIG = "buffer.memory";
private static final String BUFFER_MEMORY_DOC = "The total bytes of memory the producer can use to buffer records waiting to
be sent to the server. If records are "
                                                + "sent faster than they can be delivered to the server the producer will
block for <code>" + MAX_BLOCK_MS_CONFIG + "</code> after which it will throw an exception."
                                                + "<p>"
                                                + "This setting should correspond roughly to the total memory the producer
will use, but is not a hard bound since "
                                                + "not all memory the producer uses is used for buffering. Some additional
memory will be used for compression (if "
                                                + "compression is enabled) as well as for maintaining in-flight
requests.";

//缓冲区每一个batch的大小
public static final String BATCH_SIZE_CONFIG = "batch.size";
private static final String BATCH_SIZE_DOC = "The producer will attempt to batch records together into fewer requests whenever
multiple records are being sent"
                                             + " to the same partition. This helps performance on both the client and the
server. This configuration controls the "
                                             + "default batch size in bytes. "
                                             + "<p>"
                                             + "No attempt will be made to batch records larger than this size. "
                                             + "<p>"
                                             + "Requests sent to brokers will contain multiple batches, one for each
partition with data available to be sent. "
                                             + "<p>"
                                             + "A small batch size will make batching less common and may reduce
throughput (a batch size of zero will disable "
                                             + "batching entirely). A very large batch size may use memory a bit more
wastefully as we will always allocate a "
                                             + "buffer of the specified batch size in anticipation of additional records."
                                             + "<p>"
                                             + "Note: This setting gives the upper bound of the batch size to be sent. If
we have fewer than this many bytes accumulated "
                                             + "for this partition, we will 'linger' for the <code>linger.ms</code> time
waiting for more records to show up. "
                                             + "This <code>linger.ms</code> setting defaults to 0, which means we'll
immediately send out a record even the accumulated "
                                             + "batch size is under this <code>batch.size</code> setting.";
```

> 这里面也提到了几个其他的参数，比如 MAX_BLOCK_MS_CONFIG ， 默认60秒

接下来，sender就是KafkaProducer中用来发送消息的一个单独的线程。从这里可以看到，每个KafkaProducer对象都对应一个sender线程。他会负责将RecordAccumulator中的消息发送给Kafka。



Sender也并不是一次就把RecordAccumulator中缓存的所有消息都发送出去，而是每次只拿一部分消息。他只获取RecordAccumulator中缓存内容达到BATCH_SIZE_CONFIG大小的ProducerBatch消息。当然，如果消息比较少，ProducerBatch中的消息大小长期达不到BATCH_SIZE_CONFIG的话，Sender也不会一直等待。最多等待LINGER_MS_CONFIG时长。然后就会将ProducerBatch中的消息读取出来。LINGER_MS_CONFIG默认值是0。

然后，Sender对读取出来的消息，会以Broker为key，缓存到一个对应的队列当中。这些队列当中的消息就称为InflightRequest。接下来这些Inflight就会一一发往Kafka对应的Broker中，直到收到Broker的响应，才会从队列中移除。这些队列也并不会无限缓存，最多缓存MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION(默认值为5)个请求。

> 生产者缓存机制的主要目的是将消息打包，减少网络IO频率。所以，在Sender的InflightRequest队列中，消息也不是一条一条发送给Broker的，而是一批消息一起往Broker发送。而这就意味着这一批消息是没有固定的先后顺序的。

其中涉及到的几个主要参数如下：

```
 public static final String LINGER_MS_CONFIG = "linger.ms";
    private static final String LINGER_MS_DOC = "The producer groups together any records that arrive in between request
transmissions into a single batched request. "
                                                + "Normally this occurs only under load when records arrive faster than they
can be sent out. However in some circumstances the client may want to "
                                                + "reduce the number of requests even under moderate load. This setting
accomplishes this by adding a small amount "
                                                + "of artificial delay&mdash;that is, rather than immediately sending out a
record, the producer will wait for up to "
                                                + "the given delay to allow other records to be sent so that the sends can be
batched together. This can be thought "
                                                + "of as analogous to Nagle's algorithm in TCP. This setting gives the upper
bound on the delay for batching: once "
                                                + "we get <code>" + BATCH_SIZE_CONFIG + "</code> worth of records for a
partition it will be sent immediately regardless of this "
                                                + "setting, however if we have fewer than this many bytes accumulated for this
partition we will 'linger' for the "
                                                + "specified time waiting for more records to show up. This setting defaults
to 0 (i.e. no delay). Setting <code>" + LINGER_MS_CONFIG + "=5</code>, "
                                                + "for example, would have the effect of reducing the number of requests sent
but would add up to 5ms of latency to records sent in the absence of load.";



public static final String MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION = "max.in.flight.requests.per.connection";
private static final String MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION_DOC = "The maximum number of unacknowledged requests the
client will send on a single connection before blocking."
                                                                        + " Note that if this configuration is set to be
greater than 1 and <code>enable.idempotence</code> is set to false, there is a risk of"
                                                                        + " message reordering after a failed send due to
retries (i.e., if retries are enabled); "
                                                                        + " if retries are disabled or if
<code>enable.idempotence</code> is set to true, ordering will be preserved."
                                                                        + " Additionally, enabling idempotence requires
the value of this configuration to be less than or equal to " + MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION_FOR_IDEMPOTENCE + "."
                                                                        + " If conflicting configurations are set and
idempotence is not explicitly enabled, idempotence is disabled. ";
```

最后，Sender会通过其中的一个Selector组件完成与Kafka的IO请求，并接收Kafka的响应。

```
//org.apache.kafka.clients.producer.KafkaProducer#doSend
if (result.batchIsFull || result.newBatchCreated) {
                log.trace("Waking up the sender since topic {} partition {} is either full or getting a new batch",
record.topic(), appendCallbacks.getPartition());
                this.sender.wakeup();
            }
```

Kafka的生产者缓存机制是Kafka面对海量消息时非常重要的优化机制。合理优化这些参数，对于Kafka集群性能提升是非常重要的。比如如果你的消息体比较大，那么应该考虑加大batch.size，尽量提升batch的缓存效率。而如果Producer要发送的消息确实非常多，那么就需要考虑加大total.memory参数，尽量避免缓存不够造成的阻塞。如果发现生产者发送消息比较慢，那么可以考虑提升max.in.flight.requests.per.connection参数，这样能加大消息发送的吞吐量。

6、发送应答机制

在Producer将消息发送到Broker后，要怎么确定消息是不是成功发到Broker上了呢?

这是在开发过程中比较重要的一个机制，也是面试过程中最喜欢问的一个机制，被无数教程指导吹得神乎其神。所以这里也简单介绍一下。

其实这里涉及到的，就是在Producer端一个不太起眼的属性ACKS_CONFIG。

```
  public static final String ACKS_CONFIG = "acks";
    private static final String ACKS_DOC = "The number of acknowledgments the producer requires the leader to have received
before considering a request complete. This controls the "
                                        + " durability of records that are sent. The following settings are allowed: "
                                        + " <ul>"
                                        + " <li><code>acks=0</code> If set to zero then the producer will not wait for any
acknowledgment from the"
                                        + " server at all. The record will be immediately added to the socket buffer and
considered sent. No guarantee can be"
                                        + " made that the server has received the record in this case, and the
<code>retries</code> configuration will not"
                                        + " take effect (as the client won't generally know of any failures). The offset
given back for each record will"
                                        + " always be set to <code>-1</code>."
                                        + " <li><code>acks=1</code> This will mean the leader will write the record to its
local log but will respond"
                                        + " without awaiting full acknowledgement from all followers. In this case should
the leader fail immediately after"
                                        + " acknowledging the record but before the followers have replicated it then the
record will be lost."
                                        + " <li><code>acks=all</code> This means the leader will wait for the full set of
in-sync replicas to"
                                        + " acknowledge the record. This guarantees that the record will not be lost as
long as at least one in-sync replica"
                                        + " remains alive. This is the strongest available guarantee. This is equivalent to
the acks=-1 setting."
                                        + "</ul>"
                                        + "<p>"
                                        + "Note that enabling idempotence requires this config value to be 'all'."
                                        + " If conflicting configurations are set and idempotence is not explicitly
enabled, idempotence is disabled.";
```

官方给出的这段解释，同样比任何外部的资料都要准确详细了。如果你理解了Topic的分区模型，这个属性就非常容易理解了。这个属性更大的作用在于保证消息的安全性，尤其在replica-factor备份因子比较大的Topic中，尤为重要。

- acks=0，生产者不关心Broker端有没有将消息写入到Partition，只发送消息就不管了。吞吐量是最高的，但是数据安全性是最低的。
- acks=all or -1，生产者需要等Broker端的所有Partiton(Leader Partition以及其对应的Follower Partition都写完了才能得到返回结果，这样数据是最安全的，但是每次发消息需要等待更长的时间，吞吐量是最低的。
- acks设置成1，则是一种相对中和的策略。Leader Partition在完成自己的消息写入后，就向生产者返回结果。

> 在示例代码中可以验证，acks=0的时候，消息发送者就拿不到partition,offset这一些数据。

在生产环境中，acks=0可靠性太差，很少使用。acks=1，一般用于传输日志等，允许个别数据丢失的场景。使用范围最广。acks=-1，一般用于传输敏感数据，比如与钱相关的数据。

如果ack设置为all或者-1 ，Kafka也并不是强制要求所有Partition都写入数据后才响应。在Kafka的Broker服务端会有一个配置参数min.insync.replicas，控制Leader Partition在完成多少个Partition的消息写入后，往Producer返回响应。这个参数可以在broker.conf文件中进行配置。

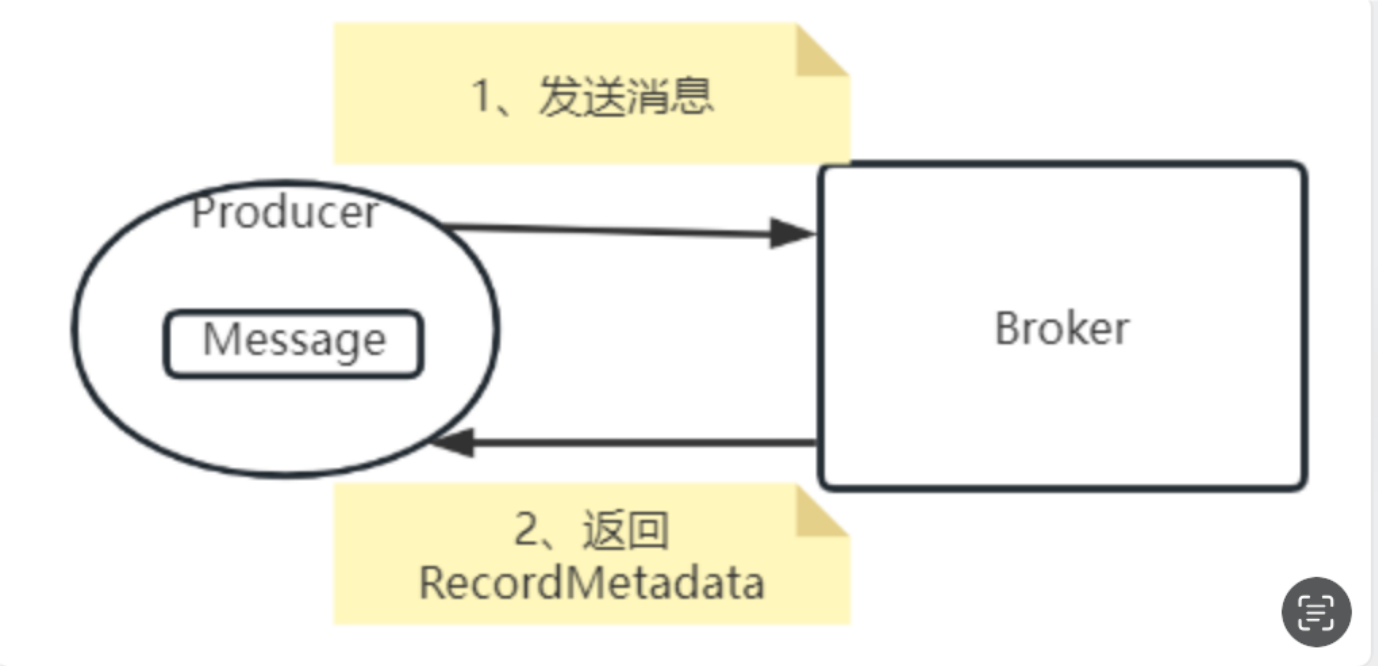
```
min.insync.replicas
When a producer sets acks to "all" (or "-1"), min.insync.replicas specifies the minimum number of replicas that must
acknowledge a write for the write to be considered successful. If this minimum cannot be met, then the producer will raise an
exception (either NotEnoughReplicas or NotEnoughReplicasAfterAppend).
When used together, min.insync.replicas and acks allow you to enforce greater durability guarantees. A typical scenario would
be to create a topic with a replication factor of 3, set min.insync.replicas to 2, and produce with acks of "all". This will
ensure that the producer raises an exception if a majority of replicas do not receive a write.

Type:   int
Default:    1
Valid Values:   [1,...]
Importance: high
Update Mode:    cluster-wide
```

关于消息应答机制，最后强调一点：acks设置成all或者-1，能够有效提高消息的安全性。但是从消息安全性方面考虑，应答机制只是保证Broker可以给Producer一个比较靠谱的响应，但并不代表就保证了消息不丢失。Producer拿到响应后如何进行后续处理，Kafka是不参与的。

7、生产者消息幂等性

当你仔细看下源码中对于acks属性的说明，会看到另外一个单词，idempotence。这个单词的意思就是幂等性。这个幂等性是什么意思呢?

之前分析过，当Producer的acks设置成1或-1时，Producer每次发送消息都是需要获取Broker端返回的RecordMetadata的。这个过程中就需要两次跨网络请求。



如果要保证消息安全，那么对于每个消息，这两次网络请求就必须要求是幂等的。但是，网络是不靠谱的，在高并发场景下，往往没办法保证这两个请求是幂等的。Producer发送消息的过程中，如果第一步请求成功了， 但是第二步却没有返回。这时，Producer就会认为消息发送失败了。那么Producer必然会发起重试。重试次数由参数ProducerConfig.RETRIES_CONFIG，默认值是Integer.MAX。

这时问题就来了。Producer会重复发送多条消息到Broker中。Kafka如何保证无论Producer向Broker发送多少次重复的数据，Broker端都只保留一条消息，而不会重复保存多条消息呢？这就是Kafka消息生产者的幂等性问题。

先来看Kafka中对于幂等性属性的介绍

```
 public static final String ENABLE_IDEMPOTENCE_CONFIG = "enable.idempotence";
    public static final String ENABLE_IDEMPOTENCE_DOC = "When set to 'true', the producer will ensure that exactly one copy of each message is written in the stream. If 'false', producer "
        + "retries due to broker failures, etc., may write duplicates of the retried message in the stream. "
        + "Note that enabling idempotence requires <code>" + MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION + "</code> to be less than or equal to " + MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION_FOR_IDEMPOTENCE
        + " (with message ordering preserved for any allowable value), <code>" + RETRIES_CONFIG + "</code> to be greater than 0, and <code>"
        + ACKS_CONFIG + "</code> must be 'all'. "
        + "<p>"
        + "Idempotence is enabled by default if no conflicting configurations are set. "
        + "If conflicting configurations are set and idempotence is not explicitly enabled, idempotence is disabled. "
        + "If idempotence is explicitly enabled and conflicting configurations are set, a <code>ConfigException</code> is thrown.";
```

这段介绍中涉及到另外两个参数，也一并列出来

```
    // max.in.flight.requests.per.connection should be less than or equal to 5 when idempotence producer enabled to ensure message ordering
    private static final int MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION_FOR_IDEMPOTENCE = 5;

    /** <code>max.in.flight.requests.per.connection</code> */
    public static final String MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION = "max.in.flight.requests.per.connection";
    private static final String MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION_DOC = "The maximum number of unacknowledged requests the client will send on a single connection before blocking."
                                                                            + " Note that if this config is set to be greater than 1 and <code>enable.idempotence</code> is set to false, there is a risk of"
                                                                            + " message re-ordering after a failed send due to retries (i.e., if retries are enabled)."
                                                                            + " Additionally, enabling idempotence requires this config value to be less than or equal to " + MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION_FOR_IDEMPOTENCE + "."
                                                                            + " If conflicting configurations are set and idempotence is not explicitly enabled, idempotence is disabled.";
```

可以看到，Kafka围绕生产者幂等性问题，其实是做了一整套设计的。只是在这些描述中并没有详细解释幂等性是如何实现的。

这里首先需要理解分布式数据传递过程中的三个数据语义：**at-least-once**:至少一次；**at-most-once**:最多一次；**exactly-once**:精确一次。

比如，你往银行存100块钱，这时银行往往需要将存钱动作转化成一个消息，发到MQ，然后通过MQ通知另外的系统去完成修改你的账户余额以及其他一些其他的业务动作。而这个MQ消息的安全性，往往是需要分层次来设计的。首先，你要保证存钱的消息能够一定发送到MQ。如果一次发送失败了，那就重试几次，只到成功为止。这就是at-least-once至少一次。如果保证不了这个语义，那么你肯定不会接受。然后，你往银行存100块，不管这个消息你发送了多少次，银行最多只能记录一次，也就是100块存款，可以少，但决不能多。这就是at-most-once最多一次。如果保证不了这个语义，那么银行肯定也不能接收。最后，这个业务动作要让双方都满意，就必须保证存钱这个消息正正好好被记录一次，不多也不少。这就是Exactly-once语义。

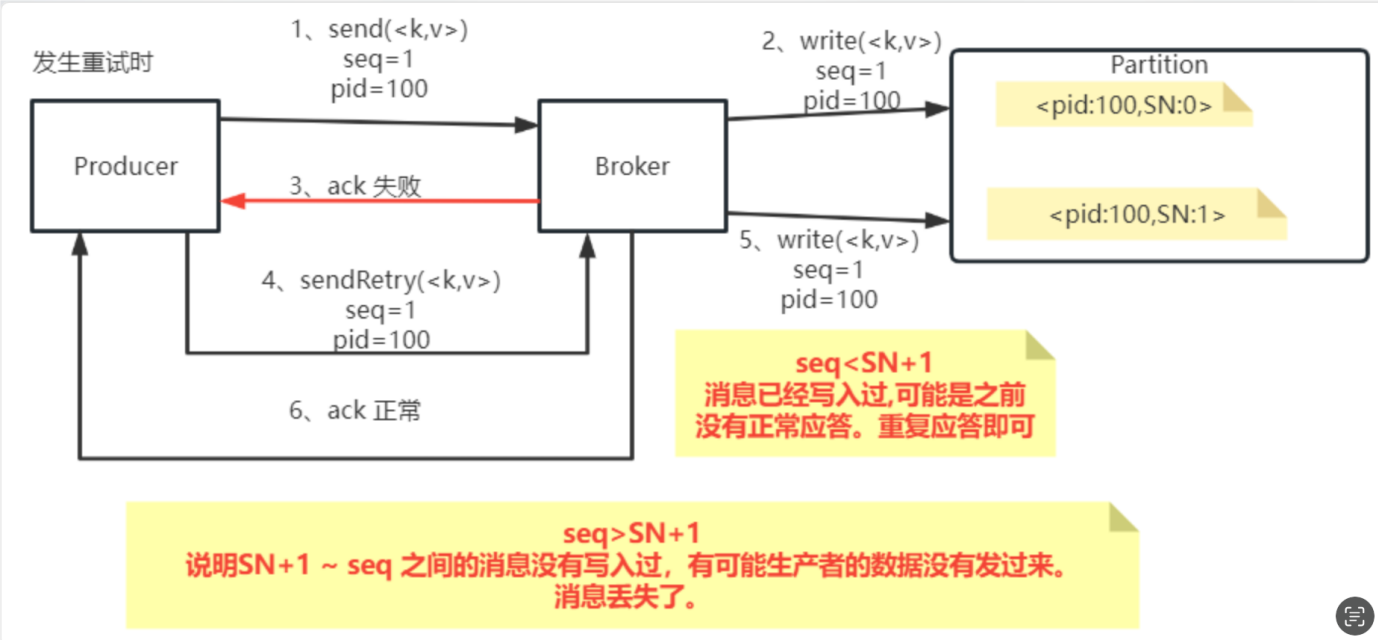
所以，通常意义上，at-least-once可以保证数据不丢失，但是不能保证数据不重复。而at-most-once保证数据不重复，但是又不能保证数据不丢失。这两种语义虽然都有缺陷，但是实现起来相对来说比较简单。但是对一些敏感的业务数据，往往要求数据即不重复也不丢失，这就需要支持Exactly-once语义。而要支持Exactly-once语义，需要有非常精密的设计。

回到Producer发消息给Broker这个场景，如果要保证at-most-once语义，可以将ack级别设置为0即可，此时，是不存在幂等性问题的。如果要保证at-least-once语义，就需要将ack级别设置为1或者-1，这样就能保证Leader Partition中的消息至少是写成功了一次的，但是不保证只写了一次。如果要支持Exactly-once语义怎么办呢？这就需要使用到idempotence幂等性属性了。

Kafka为了保证消息发送的Exactly-once语义，增加了几个概念：

- PID：每个新的Producer在初始化的过程中就会被分配一个唯一的PID。这个PID对用户是不可见的。
- Sequence Numer: 对于每个PID，这个Producer针对Partition会维护一个sequenceNumber。这是一个从0开始单调递增的数字。当Producer要往同一个Partition发送消息时，这个Sequence Number就会加1。然后会随着消息一起发往Broker。
- Broker端则会针对每个<PID,Partition>维护一个序列号（SN），只有当对应的SequenceNumber = SN+1时，Broker才会接收消息，同时将SN更新为SN+1。否则，SequenceNumber过小就认为消息已经写入了，不需要再重复写入。而如果SequenceNumber过大，就会认为中间可能有数据丢失了。对生产者就会抛出一个OutOfOrderSequenceException。

这样，Kafka在打开idempotence幂等性控制后，在Broker端就会保证每条消息在一次发送过程中，Broker端最多只会刚刚好持久化一条。这样就能保证at-most-once语义。再加上之前分析的将生产者的acks参数设置成1或-1，保证at-least-once语义，这样就整体上保证了Exactaly-once语义。



给Producer打开幂等性后，不管Producer往同一个Partition发送多少条消息，都可以通过幂等机制保证消息的Exactly-only语义。但是是不是这样消息就安全了呢?

8、生产者数据压缩机制

当生产者往Broker发送消息时，还会对每个消息进行压缩，从而降低Producer到Broker的网络数据传输压力，同时也降低了Broker的数据存储压力。

具体涉及到ProducerConfig中的 COMPRESSION_TYPE_CONFIG，配置项。

```
    /** <code>compression.type</code> */
    public static final String COMPRESSION_TYPE_CONFIG = "compression.type";
    private static final String COMPRESSION_TYPE_DOC = "The compression type for all data generated by the producer. The
default is none (i.e. no compression). Valid "
                                                       + " values are <code>none</code>, <code>gzip</code>,
<code>snappy</code>, <code>lz4</code>, or <code>zstd</code>. "
                                                       + "Compression is of full batches of data, so the efficacy of batching
will also impact the compression ratio (more batching means better compression).";
```

从介绍中可以看到，Kafka的生产者支持四种压缩算法。这几种压缩算法中，zstd算法具有最高的数据压缩比，但是吞吐量不高。lz4在吞吐量方面的优势比较明显。在实际使用时，可以根据业务情况选择合适的压缩算法。但是要注意下，压缩消息必然增加CPU的消耗，如果CPU资源紧张，就不要压缩了。

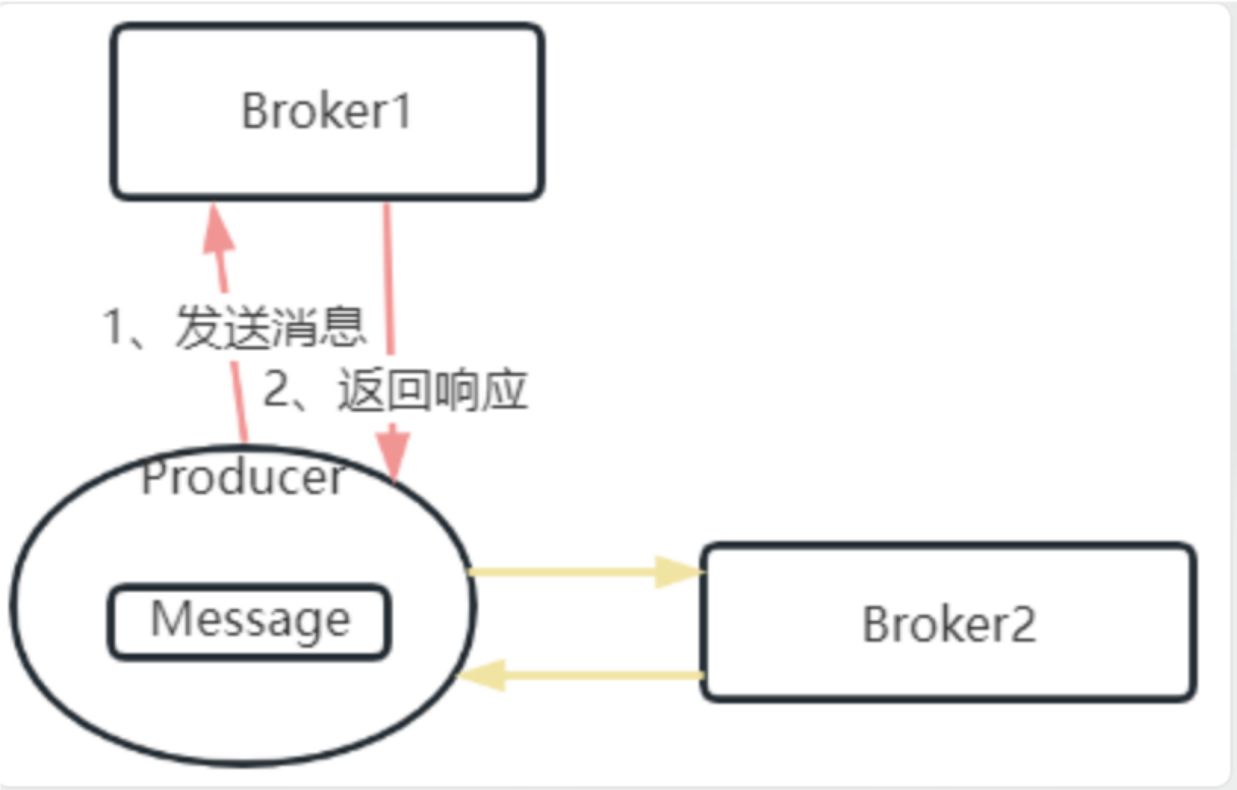
关于数据压缩机制，在Broker端的broker.conf文件中，也是可以配置压缩算法的。正常情况下，Broker从Producer端接收到消息后不会对其进行任何修改，但是如果Broker端和Producer端指定了不同的压缩算法，就会产生很多异常的表现。

```
compression.type
Specify the final compression type for a given topic. This configuration accepts the standard
compression codecs ('gzip', 'snappy', 'lz4', 'zstd'). It additionally accepts 'uncompressed' which is equivalent to no
compression; and 'producer' which means retain the original compression codec set by the producer.

Type:   string
Default:    producer
Valid Values:   [uncompressed, zstd, lz4, snappy, gzip, producer]
Server Default Property:    compression.type
Importance: medium
```

如果开启了消息压缩，那么在消费者端自然是要进行解压缩的。在Kafka中，消息从Producer到Broker再到Consumer会一直携带消息的压缩方式，这样当Consumer读取到消息集合时，自然就知道了这些消息使用的是哪种压缩算法，也就可以自己进行解压了。但是这时要注意的是应用中使用的Kafka客户端版本和Kafka服务端版本是否匹配。

9、生产者消息事务

接下来，通过生产者消息幂等性问题，能够解决单生产者消息写入单分区的的幂等性问题。但是，如果是要写入多个分区呢？比如生产者一次发送多条消息，然后给不同的消息指定不同的key。这批消息就有可能写入多个Partition，而这些Partition是分布在不同Broker上的。这意味着，Producer需要对多个Broker同时保证消息的幂等性。



这时候，通过上面的生产者消息幂等性机制就无法保证所有消息的幂等了。这时候就需要有一个事务机制，保证这一批消息最好同时成功的保持幂等性。或者这一批消息同时失败，这样生产者就可以开始进行整体重试，消息不至于重复。

而针对这个问题， Kafka就引入了消息事务机制。这涉及到Producer中的几个API：

```
// 1 初始化事务
void initTransactions();
// 2 开启事务
void beginTransaction() throws ProducerFencedException;
// 3 提交事务
void commitTransaction() throws ProducerFencedException;
// 4 放弃事务（类似于回滚事务的操作）
void abortTransaction() throws ProducerFencedException;
```

例如我们可以做个这样的测试：

```
public class TransactionErrorDemo {

    private static final String BOOTSTRAP_SERVERS = "worker1:9092,worker2:9092,worker3:9092";
    private static final String TOPIC = "disTopic";

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        Properties props = new Properties();
        // 此处配置的是kafka的端口
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        // 事务ID
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG,"111");
        // 配置key的序列化类
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,"org.apache.kafka.common.serialization.StringSerializer");
        // 配置value的序列化类
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,"org.apache.kafka.common.serialization.StringSerializer");

        Producer<String,String> producer = new KafkaProducer<>(props);
        producer.initTransactions();
        producer.beginTransaction();
        for(int i = 0; i < 5; i++) {
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, Integer.toString(i), "MyProducer" + i);
            //异步发送。
            producer.send(record);
            if(i == 3){
                //第三条消息放弃事务之后，整个这一批消息都回退了。
                System.out.println("error");
                producer.abortTransaction();
            }
        }
        System.out.println("message sended");
        try {
            Thread.sleep(10000);
        } catch (Exception e) {
            e.printStackTrace();
        }
//        producer.commitTransaction();
        producer.close();
    }
}
```

可以先启动一个订阅了disTopic这个Topic的消费者，然后启动这个生产者，进行试验。在这个试验中，发送到第3条消息时，主动放弃事务，此时之前的消息也会一起回滚。

实际上，Kafka的事务消息还会做两件事情：

1、一个TransactionId只会对应一个PID

如果当前一个Producer的事务没有提交，而另一个新的Producer保持相同的TransactionId，这时旧的生产者会立即失效，无法继续发送消息。

2、跨会话事务对齐

如果某个Producer实例异常宕机了，事务没有被正常提交。那么新的TransactionId相同的Producer实例会对旧的事务进行补齐。保证旧事务要么提交，要么终止。这样新的Producer实例就可以以一个正常的状态开始工作。

> 如果你对消息事务的实现机制比较感兴趣，可以自行参看下Apache下的这篇文章： https://cwiki.apache.org/confluence/display/KAFKA/KIP-98+-+Exactly+Once+Delivery+and+Transactional+Messaging#KIP98ExactlyOnceDeliveryandTransactionalMessaging-AnExampleApplication

所以，如果一个Producer需要发送多条消息，通常比较安全的发送方式是这样的：

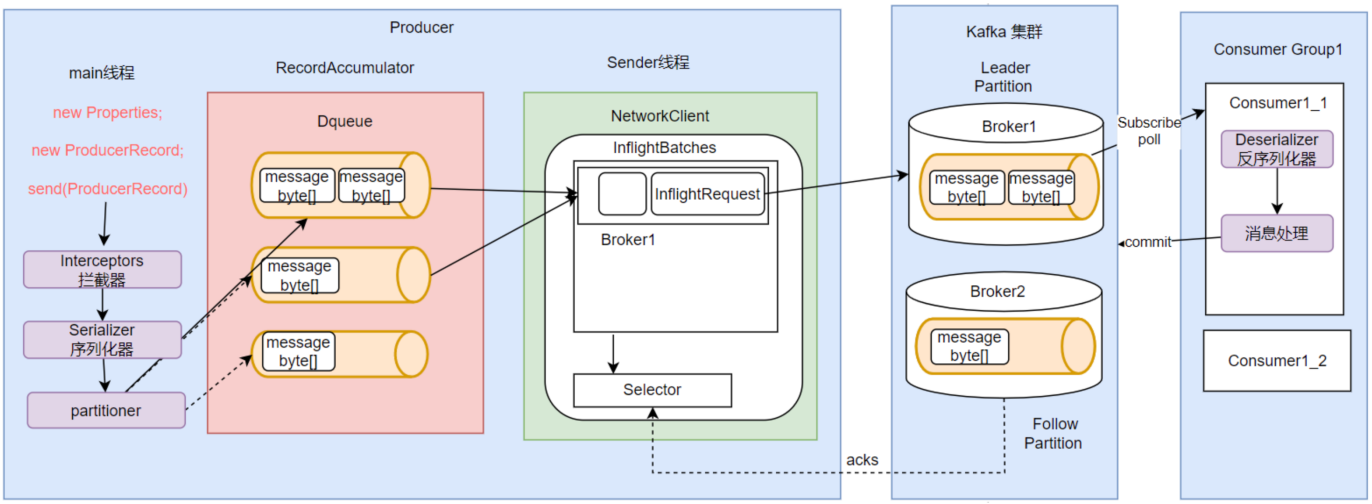
```
public class TransactionProducer {
    private static final String BOOTSTRAP_SERVERS = "worker1:9092,worker2:9092,worker3:9092";
    private static final String TOPIC = "disTopic";
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        Properties props = new Properties();
        // 此处配置的是kafka的端口
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        // 事务ID。
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG,"111");
        // 配置key的序列化类
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,"org.apache.kafka.common.serialization.StringSerializer");
        // 配置value的序列化类
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,"org.apache.kafka.common.serialization.StringSerializer");

        Producer<String,String> producer = new KafkaProducer<>(props);
        producer.initTransactions();
        producer.beginTransaction();
        try{
            for(int i = 0; i < 5; i++) {
                ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, Integer.toString(i), "MyProducer" + i);
                //异步发送。
                producer.send(record);
            }
            producer.commitTransaction();
        }catch (ProducerFencedException e){
            producer.abortTransaction();
        }finally {
            producer.close();
        }
    }
}
```

其中对于事务ID这个参数，可以任意起名，但是建议包含一定的业务唯一性。

生产者的事务消息机制保证了Producer发送消息的安全性，但是，他并不保证已经提交的消息就一定能被所有消费者消费。

三、客户端流程总结

对于这些属性，你并不需要煞有介事的强行去记忆，随时可以根据ProducerConfig和ConsumerConfig以及他们的父类CommonClientConfig去理解，大部分的属性都配有非常简明扼要的解释。但是，你一定需要尝试自己建立一个消息流转模型，理解其中比较重要的过程。然后重点从高可用，高并发的角度去理解Kafka客户端的设计，最后再尝试往其中填充具体的参数。



四、SpringBoot集成Kafka

对于Kafka，你更应该从各个角度建立起一个完整的数据流转的模型，通过这些模型去回顾Kafka的重要设计，并且尝试去验证自己的一些理解。这样才能真正去理解Kafka的强大之处。

当你掌握了Kafka的核心消息流转模型时，也可以帮助你去了解Kafka更多的应用生态。比如SpringBoot集成Kafka，其实非常简单。就分三步

1、在SpringBoot项目中，引入Maven依赖

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

2、在application.properties中配置kafka相关参数 例如

```
###########【Kafka集群】###########
spring.kafka.bootstrap-servers=worker1:9092,worker2:9093,worker3:9093
###########【初始化生产者配置】###########
# 重试次数
spring.kafka.producer.retries=0
# 应答级别:多少个分区副本备份完成时向生产者发送ack确认(可选0、1、all/-1)
spring.kafka.producer.acks=1
# 批量大小
spring.kafka.producer.batch-size=16384
# 提交延时
spring.kafka.producer.properties.linger.ms=0
# 生产端缓冲区大小
spring.kafka.producer.buffer-memory = 33554432
# Kafka提供的序列化和反序列化类
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
###########【初始化消费者配置】###########
# 默认的消费组ID
spring.kafka.consumer.properties.group.id=defaultConsumerGroup
# 是否自动提交offset
spring.kafka.consumer.enable-auto-commit=true
# 提交offset延时(接收到消息后多久提交offset)
spring.kafka.consumer.auto-commit-interval=1000
# 当kafka中没有初始offset或offset超出范围时将自动重置offset
# earliest:重置为分区中最小的offset;
# latest:重置为分区中最新的offset(消费分区中新产生的数据);
# none:只要有一个分区不存在已提交的offset,就抛出异常;
spring.kafka.consumer.auto-offset-reset=latest
# 消费会话超时时间(超过这个时间consumer没有发送心跳,就会触发rebalance操作)
spring.kafka.consumer.properties.session.timeout.ms=120000
# 消费请求超时时间
spring.kafka.consumer.properties.request.timeout.ms=180000
# Kafka提供的序列化和反序列化类
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

这些参数非常多，非常乱，如果你只是靠记忆，是记不住的。但是经过这一轮梳理，有没有觉得这些参数看着眼熟一点了？配的都是Kafka原生的这些参数。如果你真的把上面个模型中的参数补充完整了，SpringBoot框架当中的这些参数就不难整理了。

3、应用中使用框架注入的KafkaTemplate发送消息 例如

```
@RestController
public class KafkaProducer {
    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;
    // 发送消息
    @GetMapping("/kafka/normal/{message}")
    public void sendMessage1(@PathVariable("message") String normalMessage) {
        kafkaTemplate.send("topic1", normalMessage);
    }
}
```

4、使用@KafkaListener注解声明消息消费者 例如：

```
@Component
public class KafkaConsumer {
    // 消费监听
    @KafkaListener(topics = {"topic1"})
    public void onMessage1(ConsumerRecord<?, ?> record){
        // 消费的哪个topic、partition的消息,打印出消息内容
        System.out.println("简单消费："+record.topic()+"-"+record.partition()+"-"+record.value());
    }
}
```

这部分的应用本来就非常简单，而且他的本质也是在框架中构建Producer和Consumer。当你了解了kafka的核心消息流转流程，对这些应用参数就可以进行合理的组装，那么分分钟就可以上手SpringBoot集成Kafka框架的。

> 【有道云笔记】二、Kafka客户端消息流转流程.md
> https://note.youdao.com/s/QXE3n8pc