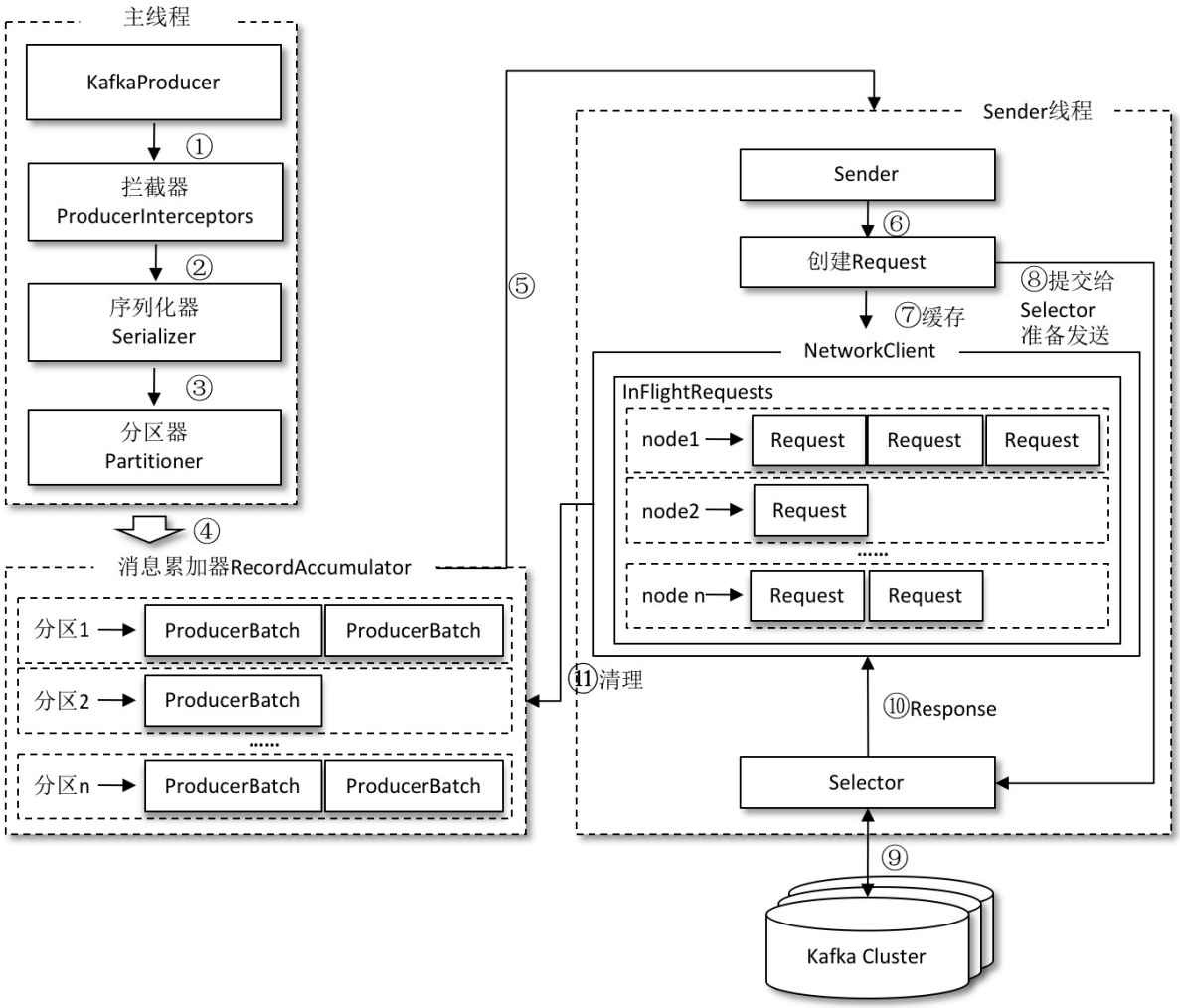
生产者客户端原理分析

在前面的章节中，我们已经了解了 KafkaProducer 的具体使用方法，而本节的内容主要是对 Kafka 生产者客户端的内部原理进行分析，通过了解生产者客户端的整体脉络可以让我们更好地使用它，避免因为一些理解上的偏差而造成使用上的错误。

整体架构

在上一节中提及了消息在真正发往 Kafka 之前，有可能需要经历拦截器（Interceptor）、序列化器（Serializer）和分区器（Partitioner）等一系列的作用，那么在此之后又会发生什么呢？下面我们来看一下生产者客户端的整体架构，如下图所示。



整个生产者客户端由两个线程协调运行，这两个线程分别为主线程和 Sender 线程（发送线程）。在主线程中由 KafkaProducer 创建消息，然后通过可能的拦截器、序列化器和分区器的作用之后缓存到消息累加器（RecordAccumulator，也称为消息收集器）中。Sender 线程负责从 RecordAccumulator 中获取消息并将其发送到 Kafka 中。

RecordAccumulator 主要用来缓存消息以便 Sender 线程可以批量发送，进而减少网络传输的资源消耗以提升性能。
RecordAccumulator 缓存的大小可以通过生产者客户端参数 buffer.memory 配置，默认值为 33554432B，即32MB。如果生产者发送消息的速度超过发送到服务器的速度，则会导致生产者空间

不足，这个时候 KafkaProducer 的 send() 方法调用要么被阻塞，要么抛出异常，这个取决于参数 max.block.ms 的配置，此参数的默认值为60000，即60秒。

主线程中发送过来的消息都会被追加到 RecordAccumulator 的某个双端队列（Deque）中，在 RecordAccumulator 的内部为每个分区都维护了一个双端队列，队列中的内容就是 ProducerBatch，即 Deque。消息写入缓存时，追加到双端队列的尾部；Sender 读取消息时，从双端队列的头部读取。注意 ProducerBatch 不是 ProducerRecord，ProducerBatch 中可以包含一至多个 ProducerRecord。通俗地说，ProducerRecord 是生产者中创建的消息，而 ProducerBatch 是指一个消息批次，ProducerRecord 会被包含在 ProducerBatch 中，这样可以使字节的使用更加紧凑。与此同时，将较小的 ProducerRecord 拼凑成一个较大的 ProducerBatch，也可以减少网络请求的次数以提升整体的吞吐量。ProducerBatch 和消息的具体格式有关，更多的详细内容可以参考《图解Kafka之核心原理》(https://juejin.im/book/5c7d270ff265da2d89634e9e)。如果生产者客户端需要向很多分区发送消息，则可以将 buffer.memory 参数适当调大以增加整体的吞吐量。

消息在网络上都是以字节（Byte）的形式传输的，在发送之前需要创建一块内存区域来保存对应的消息。在 Kafka 生产者客户端中，通过 java.io.ByteBuffer 实现消息内存的创建和释放。不过频繁的创建和释放是比较耗费资源的，在 RecordAccumulator 的内部还有一个 BufferPool，它主要用来实现 ByteBuffer 的复用，以实现缓存的高效利用。不过 BufferPool 只针对特定大小的 ByteBuffer 进行管理，而其他大小的 ByteBuffer 不会缓存进 BufferPool 中，这个特定的大小由 batch.size 参数来指定，默认值为16384B，即16KB。我们可以适当地调大 batch.size 参数以便多缓存一些消息。

ProducerBatch 的大小和 batch.size 参数也有着密切的关系。当一条消息（ProducerRecord）流入 RecordAccumulator 时，会先寻找与消息分区所对应的双端队列（如果没有则新建），再从这个双端队列的尾部获取一个 ProducerBatch（如果没有则新建），查看 ProducerBatch 中是否还可以写入这个 ProducerRecord，如果可以则写入，如果不可以则需要创建一个新的 ProducerBatch。在新建 ProducerBatch 时评估这条消息的大小是否超过 batch.size 参数的大小，如果不超过，那么就以 batch.size 参数的大小来创建 ProducerBatch，这样在使用完这段内存区域之后，可以通过 BufferPool 的管理来进行复用；如果超过，那么就以评估的大小来创建 ProducerBatch，这段内存区域不会被复用。

Sender 从 RecordAccumulator 中获取缓存的消息之后，会进一步将原本<分区, Deque< ProducerBatch>> 的保存形式转变成 <Node, List< ProducerBatch> 的形式，其中 Node 表示 Kafka 集群的 broker 节点。对于网络连接来说，生产者客户端是与具体的 broker 节点建立的连接，也就是向具体的 broker 节点发送消息，而并不关心消息属于哪一个分区；而对于 KafkaProducer 的应用逻辑而言，我们只关注向哪个分区中发送哪些消息，所以在这里需要做一个应用逻辑层面到网络I/O层面的转换。

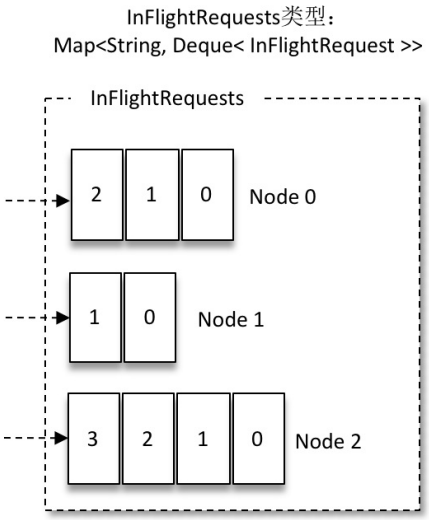
在转换成 <Node, List> 的形式之后，Sender 还会进一步封装成 <Node, Request> 的形式，这样就可以将 Request 请求发往各个 Node 了，这里的 Request 是指 Kafka 的各种协议请求，对于消息发送而言就是指具体的 ProduceRequest。

请求在从 Sender 线程发往 Kafka 之前还会保存到 InFlightRequests 中，InFlightRequests 保存对象的具体形式为 Map<NodeId, Deque>，它的主要作用是缓存了已经发出去但还没有收到响应的请求（NodeId 是一个 String 类型，表示节点的 id 编号）。与此同时，InFlightRequests 还提供了许多管理类的方法，并且通过配置参数还可以限制每个连接（也就是客户端与 Node 之间的连接）最多缓存的请求数。这个配置参数为

max.in.flight.requests. per. connection，默认值为5，即每个连接最多只能缓存5个未响应的请求，超过该数值之后就不能再向这个连接发送更多的请求了，除非有缓存的请求收到了响应（Response）。通过比较 Deque 的 size 与这个参数的大小来判断对应的 Node 中是否已经堆积了很多未响应的消息，如果真是如此，那么说明这个 Node 节点负载较大或网络连接有问题，再继续向其发送请求会增大请求超时的可能。

元数据的更新

前面提及的 InFlightRequests 还可以获得 leastLoadedNode，即所有 Node 中负载最小的那一个。这里的负载最小是通过每个 Node 在 InFlightRequests 中还未确认的请求决定的，未确认的请求越多则认为负载越大。对于下图中的 InFlightRequests 来说，图中展示了三个节点 Node0、Node1和Node2，很明显 Node1 的负载最小。也就是说，Node1 为当前的 leastLoadedNode。选择 leastLoadedNode 发送请求可以使它能够尽快发出，避免因网络拥塞等异常而影响整体的进度。leastLoadedNode 的概念可以用于多个应用场合，比如元数据请求、消费者组播协议的交互。



我们使用如下的方式创建了一条消息 ProducerRecord：

```
ProducerRecord<String, String> record =
new ProducerRecord<>(topic, "Hello, Kafka!");
```

我们只知道主题的名称，对于其他一些必要的信息却一无所知。KafkaProducer 要将此消息追加到指定主题的某个分区所对应的 leader 副本之前，首先需要知道主题的分区数量，然后经过计算得出（或者直接指定）目标分区，之后 KafkaProducer 需要知道目标分区的 leader 副本所在的 broker 节点的地址、端口等信息才能建立连接，最终才能将消息发送到 Kafka，在这一过程中所需要的信息都属于元数据信息。

在第3节中我们了解了 bootstrap.servers 参数只需要配置部分 broker 节点的地址即可，不需要配置所有 broker 节点的地址，因为客户端可以自己发现其他 broker 节点的地址，这一过程也属于元数据相关的更新操作。与此同时，分区数量及 leader 副本的分布都会动态地变化，客户端也需要动态地捕捉这些变化。

元数据是指 Kafka 集群的元数据，这些元数据具体记录了集群中有哪些主题，这些主题有哪些分区，每个分区的 leader 副本分配在哪个节点上，follower 副本分配在哪些节点上，哪些副本在 AR、ISR 等集合中，集群中有哪些节点，控制器节点又是哪一个等信息。

当客户端中没有需要使用的元数据信息时，比如没有指定的主题信息，或者超过 metadata.max.age.ms 时间没有更新元数据都会引起元数据的更新操作。客户端参数 metadata.max.age.ms 的默认值为300000，即5分钟。元数据的更新操作是在客户端内部进行的，对客户端的外部使用者不可见。当需要更新元数据时，会先挑选出 leastLoadedNode，然后向这个 Node 发送 MetadataRequest 请求来获取具体的元数据信息。这个更新操作是由 Sender 线程发起的，在创建完 MetadataRequest 之后同样会存入 InFlightRequests，之后的步骤就和发送消息时的类似。元数据虽然由 Sender 线程负责更新，但是主线程也需要读取这些信息，这里的数据同步通过 synchronized 和 final 关键字来保障。