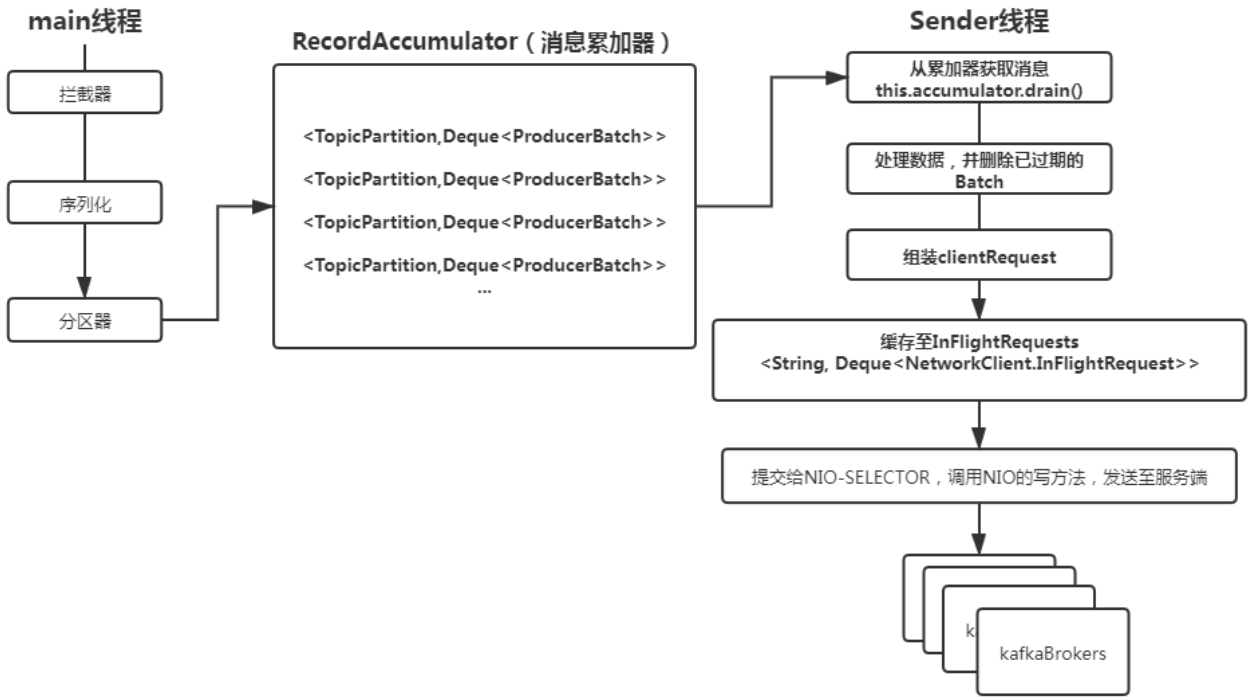
kafka生产者消息发送源码分析

生产者消息发送至网络层之前的处理流程图



kafka的消息发送分为两个部分

main线程向RecordAccumulator添加数据部分

Sender线程从RecordAccumulator获取数据发送部分

main线程

RecordAccumulator译为消息累加器，生产者不停的向消息累加器中添加数据。在RecordAccumulator中为每个分区维护了一个序列`ConcurrentMap<TopicPartition,Deque<ProducerBatch>>`,在经过拦截器，序列化器，和分区器后，消息会被试图添加到RecordAccumulator中。

```
RecordAccumulator.RecordAppendResult result = accumulator.append(tp, timestamp,
serializedKey,serializedValue, headers, interceptCallback, remainingWaitMs);
```

消息在网络之间的传输是通过Byte传输的，消息发送之前，在RecordAccumulator中存储在内存buffer中，kafka客户端中，通过java.io.ByteBuffer实现内存的创建与释放，频繁的创建和释放浪费资源，所以，RecordAccumulator内部通过BufferPool的方式实现ByteBuffer的复用。(类似线程池)

从ConcurrentMap取出对应分区的队列。

当消息到达RecordAccumulator时，会找到对应分区的双端队列：

```
//取出对应分区的队列 （没有则新建）
Deque<ProducerBatch> dq = getOrCreateDeque(tp);
```

试图添加到队列中的RecordBatch里

```
RecordAppendResult appendResult = tryAppend(timestamp, key, value, headers, callback, dq);
```

在tryAppend方法中从双端队列尾部试图取出一个RecordBatch。如果可以写入则直接写入。

当双端队列没有RecordBatch时，或者RecordBatch无法继续写入时，会进行新建。

```
// we don't have an in-progress record batch try to allocate a new batch
byte maxUsableMagic = apiVersions.maxUsableProduceMagic();
//对比配置的batch.size和估算的消息大小 取最大  之所以是估算,因为无法计算压缩情况下的开销
int size = Math.max(this.batchSize,
AbstractRecords.estimateSizeInBytesUpperBound(maxUsableMagic, compression, key, value,
headers));
log.trace("Allocating a new {} byte message buffer for topic {} partition {}", size, tp.topic(),
tp.partition());
//如果size超过了缓冲池中的大小,则无法使用缓冲池，即，当前申请的buffer不会进入到buffer中
buffer = free.allocate(size, maxTimeToBlock);
```

消息成功追加到RecordAccumulator后，main线程的任务大致已完成。

Sender线程

Sender线程的任务就是循环获取RecordAccumulator中的RecordBatch进行发送。

核心处理方法sendProducerData()

sendProducerData()方法中，从RecordAccumulator取出数据后，完成了`<TopicPartition,Deque<ProducerBatch>>`到`<Integer(NodeNum),List<ProducerBatch>>`的转换。(对客户端来说，只需要知道发送至哪个节点)。

```
    private long sendProducerData(long now) {
        Cluster cluster = metadata.fetch();
        // 获取准备发送数据的分区列表
        RecordAccumulator.ReadyCheckResult result = this.accumulator.ready(cluster, now);
```

```
        // if there are any partitions whose leaders are not known yet, force metadata update
        //如果分区没有Leader相关信息。则强制更新其元数据
        if (!result.unknownLeaderTopics.isEmpty()) {
            // The set of topics with unknown leader contains topics with leader election
pending as well as
            // topics which may have expired. Add the topic again to metadata to ensure it is
included
            // and request metadata update, since there are messages to send to the topic.
            for (String topic : result.unknownLeaderTopics)
                this.metadata.add(topic);

            log.debug("Requesting metadata update due to unknown leader topics from the batched
records: {}",
                result.unknownLeaderTopics);
            this.metadata.requestUpdate();
        }

        //删除我们不需要发送的节点,也就是与Node没有连接,该node暂时无法发送数据,删除连不上的节点。
        Iterator<Node> iter = result.readyNodes.iterator();
        long notReadyTimeout = Long.MAX_VALUE;
        while (iter.hasNext()) {
            Node node = iter.next();
            if (!this.client.ready(node, now)) {
                iter.remove();
                notReadyTimeout = Math.min(notReadyTimeout, this.client.pollDelayMs(node, now));
            }
        }

        // 获取这些节点对应的ProducerBatch
        // (此处返回的数据 Map<nodes, List<ProducerBatch>> 也就是将要发送的消息,按Broker节点维度进行
划分) ,
        // 并将获取的数据从Deque中删除。
        //
        Map<Integer, List<ProducerBatch>> batches = this.accumulator.drain(cluster,
result.readyNodes, this.maxRequestSize, now);

        //添加到InflightBatches中(飞行中的batches,还未得到响应的),通过此结构可统计消息堆积情况
        addToInflightBatches(batches);

        //如果需要保证消息顺序,记录每个分区号,防止同一时间往同一个分区发送多条未完成状态的消息
        if (guaranteeMessageOrder) {
            // Mute all the partitions drained
            for (List<ProducerBatch> batchList : batches.values()) {
                for (ProducerBatch batch : batchList)
                    this.accumulator.mutePartition(batch.topicPartition);
            }
        }
        //获取超时的消息,并释放空间
        accumulator.resetNextBatchExpiryTime();
        List<ProducerBatch> expiredInflightBatches = getExpiredInflightBatches(now);
        List<ProducerBatch> expiredBatches = this.accumulator.expiredBatches(now);
        expiredBatches.addAll(expiredInflightBatches);
```

```
        // Reset the producer id if an expired batch has previously been sent to the broker.
Also update the metrics
        // for expired batches. see the documentation of @TransactionState.resetProducerId to
understand why
        // we need to reset the producer id here.
        //如果将已经过期的Batch发送到了Broker 需重置pid
        //这种情况为,当数据发送时并未过期,但发送以后过期了。此时,我们不清楚该消息是否已经在broker端被提
交,
        //更新过期Batch数据的指标
        if (!expiredBatches.isEmpty())
            log.trace("Expired {} batches in accumulator", expiredBatches.size());
        for (ProducerBatch expiredBatch : expiredBatches) {
            String errorMessage = "Expiring " + expiredBatch.recordCount + " record(s) for " +
expiredBatch.topicPartition
                + ":" + (now - expiredBatch.createdMs) + " ms has passed since batch creation";
            failBatch(expiredBatch, -1, NO_TIMESTAMP, new TimeoutException(errorMessage),
false);
            if (transactionManager != null && expiredBatch.inRetry()) {
                // This ensures that no new batches are drained until the current in flight
batches are fully resolved.
                transactionManager.markSequenceUnresolved(expiredBatch.topicPartition);
            }
        }
        sensors.updateProduceRequestMetrics(batches);

        // If we have any nodes that are ready to send + have sendable data, poll with 0 timeout
so this can immediately
        // loop and try sending more data. Otherwise, the timeout will be the smaller value
between next batch expiry
        // time, and the delay time for checking data availability. Note that the nodes may have
data that isn't yet
        // sendable due to lingering, backing off, etc. This specifically does not include nodes
with sendable data
        // that aren't ready to send since they would cause busy looping.
        //如果存在待发送的消息，则设置 pollTimeout 等于 0，这样可以立即发送请求，从而能够缩短剩余消息的
缓存时间，避免堆积
        long pollTimeout = Math.min(result.nextReadyCheckDelayMs, notReadyTimeout);
        pollTimeout = Math.min(pollTimeout, this.accumulator.nextExpiryTimeMs() - now);
        pollTimeout = Math.max(pollTimeout, 0);
        if (!result.readyNodes.isEmpty()) {
            log.trace("Nodes with data ready to send: {}", result.readyNodes);
            // if some partitions are already ready to be sent, the select time would be 0;
            // otherwise if some partition already has some data accumulated but not ready yet,
            // the select time will be the time difference between now and its linger expiry
time;
            // otherwise the select time will be the time difference between now and the
metadata expiry time;
            pollTimeout = 0;
        }
        //对每个节点发送数据请求 处理响应
        sendProduceRequests(batches, now);
        return pollTimeout;

    }
```

发送请求方法sendProduceRequests(batches, now);

```
    private void sendProduceRequest(long now, int destination, short acks, int timeout,
List<ProducerBatch> batches) {
        if (batches.isEmpty())
            return;

        Map<TopicPartition, MemoryRecords> produceRecordsByPartition = new HashMap<>
(batches.size());
        final Map<TopicPartition, ProducerBatch> recordsByPartition = new HashMap<>
(batches.size());

        // find the minimum magic version used when creating the record sets
        byte minUsedMagic = apiVersions.maxUsableProduceMagic();
        for (ProducerBatch batch : batches) {
            if (batch.magic() < minUsedMagic)
                minUsedMagic = batch.magic();
        }
        //消息格式向下兼容
        for (ProducerBatch batch : batches) {
            TopicPartition tp = batch.topicPartition;
            MemoryRecords records = batch.records();

            // down convert if necessary to the minimum magic used. In general, there can be a
delay between the time
            // that the producer starts building the batch and the time that we send the
request, and we may have
            // chosen the message format based on out-dated metadata. In the worst case, we
optimistically chose to use
            // the new message format, but found that the broker didn't support it, so we need
to down-convert on the
            // client before sending. This is intended to handle edge cases around cluster
upgrades where brokers may
            // not all support the same message format version. For example, if a partition
migrates from a broker
            // which is supporting the new magic version to one which doesn't, then we will need
to convert.
            if (!records.hasMatchingMagic(minUsedMagic))
                records = batch.records().downConvert(minUsedMagic, 0, time).records();
            produceRecordsByPartition.put(tp, records);
            recordsByPartition.put(tp, batch);
        }

        String transactionalId = null;
        if (transactionManager != null && transactionManager.isTransactional()) {
            transactionalId = transactionManager.transactionalId();
        }
        //组装ProduceRequest请求
        ProduceRequest.Builder requestBuilder = ProduceRequest.Builder.forMagic(minUsedMagic,
acks, timeout,
                produceRecordsByPartition, transactionalId);
        //回调处理响应
```

```
        RequestCompletionHandler callback = new RequestCompletionHandler() {
            public void onComplete(ClientResponse response) {
                handleProduceResponse(response, recordsByPartition, time.milliseconds());
            }
        };

        String nodeId = Integer.toString(destination);
        //组装clientRequest请求
        ClientRequest clientRequest = client.newClientRequest(nodeId, requestBuilder, now, acks
!= 0,
                requestTimeoutMs, callback);

        //发送请求，添加至kafka封装的NIO通道Channel 在此方法中,会将请求对象缓存至InFlightRequests中
        client.send(clientRequest, now);
        log.trace("Sent produce request to {}: {}", nodeId, requestBuilder);
    }
```

发送sendProduceRequests请求之前，会将Request对象缓存至InFlightRequests直到服务端响应。

```
private final Map<String, Deque<NetworkClient.InFlightRequest>> requests = new HashMap<>();
```

InFlightRequests缓存的都是已经发出去，但未得到服务响应的请求。其中，每个链接最大缓存数量默认为5；

```
max.in.flight.requests.per.connection#默认5
```

超过后此链接不能再次发送请求，直到InFlightRequests中缓存的请求收到响应。