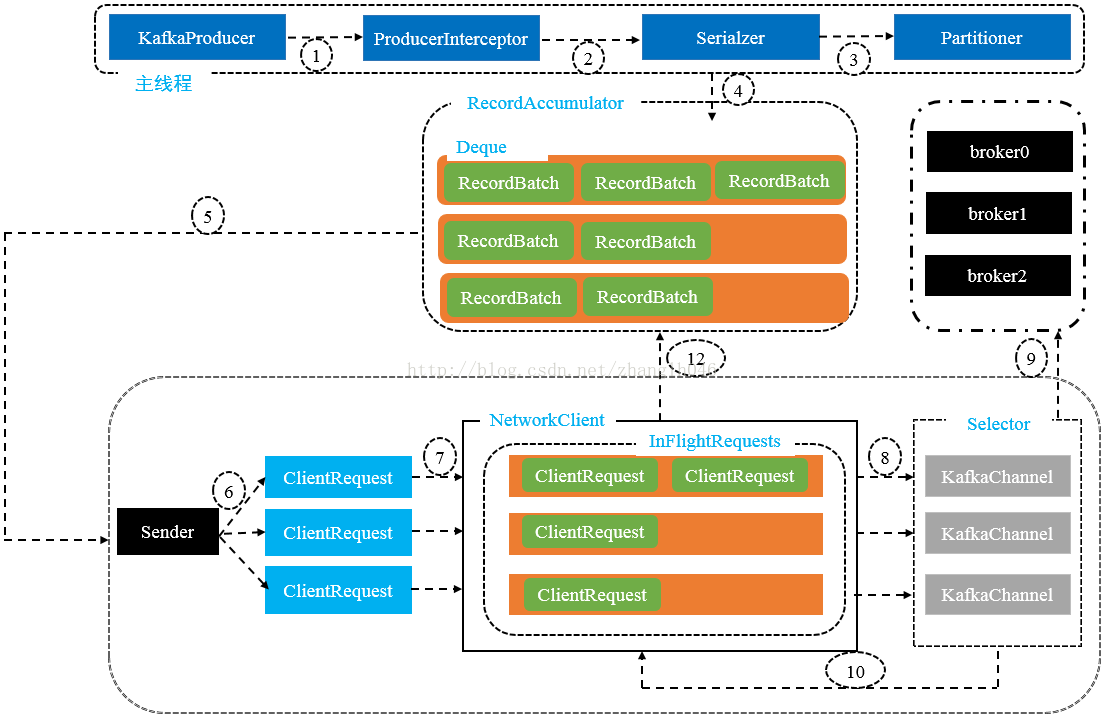
Kafka Producer

Producer将消息发布到指定的Topic中，同时决定该消息属于哪个Partition（基于Round-Robin方式或者其他算法），生产者发送消息到broker的流程如下图所示：



这里主要涉及到两个线程：

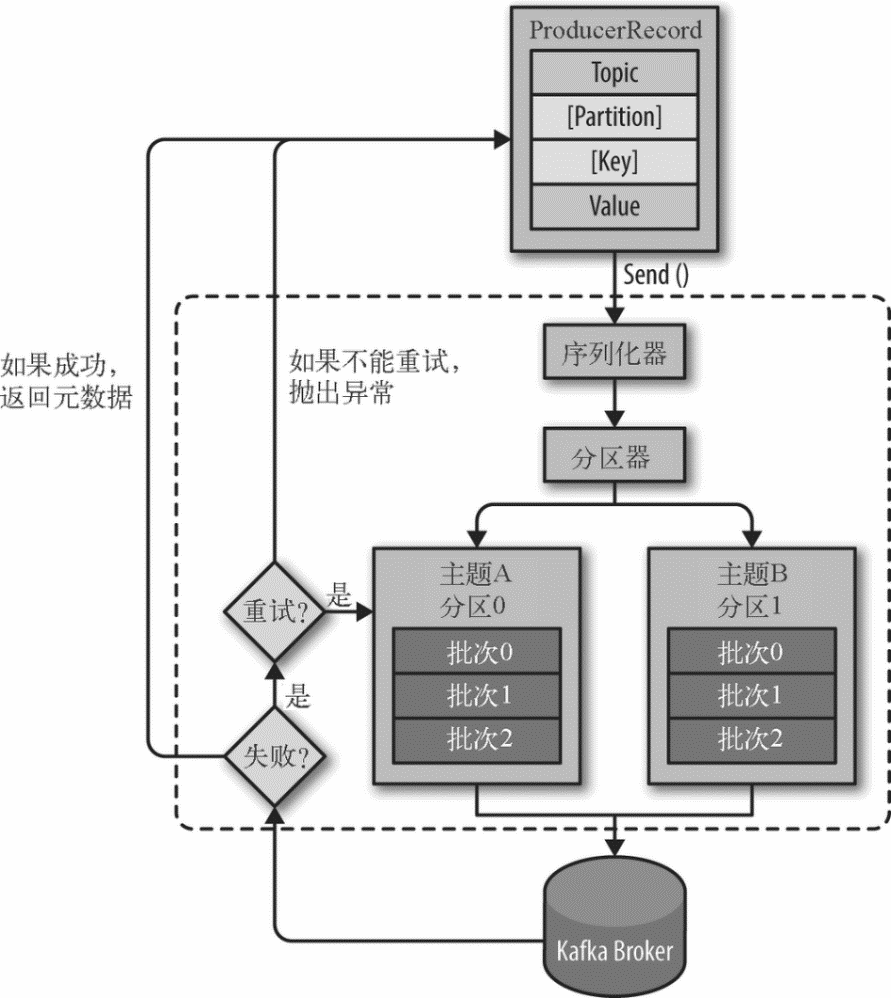
* 主线程，负责封装消息称ProducerRecord对象，之后调用send方法将消息放入Record

Accumulator中暂存

* Sender线程，负责将消息构造成请求，并从RecordAccumulator取出消息并批量发送

# KafkaProducer

KafkaProducer是一个用于向Kafka集群发送数据的Java客户端，发送消息的主要步骤：



1. 创建ProducerRecord对象，包含目标主题和要发送的内容，还可以指定键或分区
2. 生产者在发送ProducerRecord对象时，首先把键值对象序列化字节数组
3. 数据传给分区器（Partitioner），如果在ProducerRecord对象中指定了分区，那么分区器就直接把指定的分区返回，如果没有指定分区，那么分区器根据ProducerRecord对象的键来选择一个分区，选好分区后，生产者就知道往哪个主题和分区发送这条记录
4. 将这条记录添加到记录批次中，这个批次中所有消息都会发送到相同的主题和分区上
5. 启动独立的线程Sender，将记录批次发送到相应的Broker上
6. 服务器收到这条消息后，会返回一个响应，成功则返回RecordMedaData对象（包含主题和分区信息，以及记录在分区里的偏移量）。写入失败，则会返回一个错误，然后生产者会尝试重新发送消息

其使用的示例如下所示：

*Properties props = new Properties();*

*props.put("bootstrap.servers", "192.168.137.200:9092");*

*props.put("acks", "all");*

*props.put("retries", 0);*

*props.put("batch.size", 16384);*

*props.put("linger.ms", 1);*

*props.put("buffer.memory", 33554432);*

*props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");*

*props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");*

*//创建生产者*

*String topic = "mytopic";*

*Producer<String, String> procuder = new KafkaProducer<String,String>(props);*

创建生产者，必须要设置以下属性：

* bootstrap.servers，该属性指定broker的地址清单，地址格式host:port，清单中不需要包含所有的broker地址，生产者会从给定的broker中查找到其他的broker信息，建议提供至少两个broker信息，防止配置的broker宕机
* key.serializer，broker希望接收到的键和值都是字节数组，但是生产者接口允许使用参数化类型，可以把Java对象作为键和值发送给broker。生产者需要知道如何把Java对象序列化为字节数组，key.serializer必须设置为一个实现Serializer接口的类。

客户端默认提供ByteArraySerializer、StringSerializer和IntegerSerializer。

* value.serializer，与key.serializer一样，将value值序列化

其他可选配置：

* acks，指定必须要有多少个分区副本收到消息，生产者才会认为消息写入时成功的。

1. acks=0，生产者在成功写入消息前不会等待任何来自服务器的响应，会达到很高的吞吐量
2. acks=1，集群的Leader节点收到此消息，则会收到服务器的成功响应
3. acks=all，只有当所有与复制的节点全部收到消息时，这种模式是最安全的，保证不止一个服务器收到消息

* buffer.memory，用来设置内存缓冲区的大小，用其缓存其发送到服务器的消息，发送消息的速度超过服务器的速度，就会导致生产者空间不足
* retries，生产者从服务器收到的错误可能是临时性的错误（比如找不到Leader），这种情况下，retries参数的值决定了生产者可以重发消息的次数，达到这个次数，生产者会放弃重试并返回错误。retry.backoff.ms参数来改变这个时间间隔
* batch.size，当有多个消息被发送到同一个分区时，生产者会把他们放在同一个批次里，该参数指定一个批次可以使用的内存大小，按照字节数计算，当批次被填满，批次里所有消息会被发送出去。
* linger.ms，该参数指定生产者在发送批次之前等待更多消息加入批次的时间，KafkaProducer会在批次填满或linger.ms达到上限时把批次发送出去。
* client.id，该参数可以是任意的字符串，服务器会用它来识别消息的来源，还可以用在日志和配额指标里。

# Producer拦截器

Kakfa Producer端的拦截器，主要对消息进行拦截或者修改，也可以用于Producer的Callback回调前进行相应的预处理。拦截器的使用比较简单：

*props.put("interceptor.classes", "T <implements ProducerInterceptor>");*

可以指定多个拦截器，配置时用逗号隔开。

Kakfa Producer拦截器主要是实现ProducerInterceptor接口，此接口包含4个方法：

1. ProducerRecord<K,V> onSend(ProducerRecord<K,V> record)，Producer在将消息序列化和分配分区之前会调用拦截器这个方法对消息进行相应的操作，一般不修改消息的topic,key以及partition等信息
2. void onAckowledgement(RecordMetadata metadata,Exception exception)，在消息被应答之前或者消息发送失败时调用，优先于用户设定的Callback之前执行。这个方法运行在Producer的IO线程中
3. void close,关闭当前拦截器，用于执行一些资源的清理工作
4. configure(Map<String,?> configs)，用来初始化此类方法

一般只需要关注实现onSend和onAckowlegedment方法即可。

# 序列化器

创建生产者必须指定序列化器，可以使用序列化框架如Avro、Thrift或者Protobuf等序列化框架，或者使用自定义序列化器。序列化器要实现Serializer接口，其定义的方法：

1. *public void configure(Map<String, ?> configs, boolean isKey)：用来配置当前类。*
2. *public byte[] serialize(String topic, T data)：用来执行序列化。*
3. *public void close()：用来关闭当前序列化器。一般情况下这个方法都是个空方法，如果实现了此方法，必须确保此方法的幂等性，因为这个方法很可能会被KafkaProducer调用多次。*

例如Kafka中定义的StringSerializer的实现如下

*public class StringSerializer implements Serializer<String> {*

*private String encoding = "UTF8";*

*@Override*

*public void configure(Map<String, ?> configs, boolean isKey) {*

*String propertyName = isKey ? "key.serializer.encoding" : "value.serializer.encoding";*

*Object encodingValue = configs.get(propertyName);*

*if (encodingValue == null)*

*encodingValue = configs.get("serializer.encoding");*

*if (encodingValue instanceof String)*

*encoding = (String) encodingValue;*

*}*

*@Override*

*public byte[] serialize(String topic, String data) {*

*try {*

*if (data == null)*

*return null;*

*else*

*return data.getBytes(encoding);*

*} catch (UnsupportedEncodingException e) {*

*… }*

*}*

*@Override*

*public void close() {*

*// nothing to do*

*}}*

自定义序列化器和反序列化器要实现接口：Serializer和Deserializer，下面是实现Company类型的Serializer和Deserializer

1）Company类定义

*public class Company {*

*private String name;*

*private String address;*

*//省略Getter, Setter, Constructor & toString方法*

*}*

2）定义CompanySerializer

*public class CompanySerializer implements Serializer<Company> {*

*@Override*

*public void configure(Map<String, ?> configs, boolean isKey) {*

*//Nothing to do*

*}*

*@Override*

*public byte[] serialize(String topic, Company data) {*

*if (data == null) {*

*return null;*

*}*

*byte[] name, address;*

*try {*

*if(data.getName() != null) {*

*name = data.getName().getBytes("UTF-8");*

*} else {*

*name = new byte[0];*

*}*

*if(data.getAddress() != null) {*

*address = data.getAddress().getBytes("UTF-8");*

*} else {*

*address = new byte[0];*

*}*

*ByteBuffer buffer = ByteBuffer.allocate(4+4+name.length+address.length);*

*buffer.putInt(name.length);*

*buffer.put(name);*

*buffer.putInt(address.length);*

*buffer.put(address);*

*return buffer.array();*

*} catch(UnsupportedEncodingException e) {*

*e.printStackTrace();*

*}*

*return new byte[0];*

*}*

*@Override*

*public void close() { }}*

3）反序列化器

*public class CompanyDeserializer implements Deserializer<Company> {*

*@Override*

*public void configure(Map<String, ?> configs, boolean isKey) { }*

*@Override*

*public Company deserialize(String topic, byte[] data) {*

*if (data == null) {*

*return null;*

*}*

*if (data.length < 8) {*

*throw new SerializationException("Size of data received by CompanyDeserializer is shorter than expected");*

*}*

*ByteBuffer buffer = ByteBuffer.wrap(data);*

*int nameLen, addressLen;*

*String name, address;*

*nameLen = buffer.getInt();*

*byte[] nameBytes = new byte[nameLen];*

*buffer.get(nameBytes);*

*addressLen = buffer.getInt();*

*byte[] addressBytes = new byte[addressLen];*

*buffer.get(addressLen);*

*try {*

*name = new String(nameBytes, "UTF-8");*

*address = new String(addressBytes, "UTF-8");*

*} catch(UnsupportedEncodingException e) {*

*throw new SerializationException("Error occur when deserializing!");*

*}*

*return new Company(name,address);*

*}*

*@Override*

*public void close() {}}*

4）序列化器和反序列化器的使用

使用序列化器，在参数中定义：

*private final KafkaProducer<String,Company> producer;*

*props.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, CompanySerializer.class.getName());*

使用反序列化器，定义如下：

*props.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, CompanyDeserializer.class.getName())*

https://unmi.cc/kafka-produce-consume-avro-data/

# 分区器

ProducerRecord对象包含了目标主题、键和值，Kafka消息是键值对，ProducerRecord对象可以只包含目标主题和值，键可以设置为默认null，不过大多数应用程序会用到键，键的用途：可以作为消息的附加信息，也可以用来决定消息该被写到主题的哪个分区。

拥有相同键的消息被写到同一个分区，也就是说，如果一个进程只从一个主题的分区读取数据，那么具有相同键的所有记录都会被该进程读取，创建一个包含键值的记录，ProducerRecord对象：

*ProducerRecord<Integer,String> record = new ProducerRecord<>(“CustomerCountry”,”Laboratory Equipment”,”USA”);*

1）如果要创建键为null的消息，不指定键就可以了：

*ProducerRecord<Integer,String> record = new ProducerRecord<>(“CustomerCountry” ,”USA”);*

设置为null时，使用默认的分区器，记录被随机发送到主题内各个可用的分区上，分区器使用轮询（Round Bobin）算法将消息均衡的分布到各分区上。

2）键值不为空

使用默认分区器，Kafka会对键进行散列，然后根据散列值把消息映射到特定分区上，关键在于同一个键总是被映射到同一个分区上，所以在进行映射时，会使用主题所有分区，而不仅仅是可用的分区。如果使用键来映射分区，那么最好在创建主题时就把分区规划好。

3）自定义分区策略

除了默认分区器外，有时候也需要对数据进行不一样的分区，其分区实现接口Partitioner，接口中包含：configure/partition和close 三个方法。

*public class DefaultPartitioner implements Partitioner {*

*private final ConcurrentMap<String, AtomicInteger> topicCounterMap =*

*new ConcurrentHashMap<>();*

*public void configure(Map<String, ?> configs) {}*

*public int partition(String topic, Object key, byte[] keyBytes,*

*Object value, byte[] valueBytes, Cluster cluster) {*

*List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);*

*int numPartitions = partitions.size();*

*if (keyBytes == null) {*

*int nextValue = nextValue(topic);*

*List<PartitionInfo> availablePartitions = cluster.availablePartitionsForTopic(topic);*

*if (availablePartitions.size() > 0) {*

*int part = Utils.toPositive(nextValue) % availablePartitions.size();*

*return availablePartitions.get(part).partition();*

*} else {*

*return Utils.toPositive(nextValue) % numPartitions;}*

*} else {*

*// hash the keyBytes to choose a partition*

*return Utils.toPositive(Utils.murmur2(keyBytes)) % numPartitions;*

*}*

*}*

*private int nextValue(String topic) {*

*AtomicInteger counter = topicCounterMap.get(topic);*

*if (null == counter) {*

*counter = new AtomicInteger(ThreadLocalRandom.current().nextInt());*

*AtomicInteger currentCounter = topicCounterMap.putIfAbsent(topic, counter);*

*if (currentCounter != null) {*

*counter = currentCounter;*

*}*

*}*

*return counter.getAndIncrement();*

*}*

*public void close() {}*

*}*

# 写入数据

*//生产者发送消息*

*for (int i = 1; i <= 10; i++) {*

*String value = "value\_" + i;*

*ProducerRecord<String, String> msg = new ProducerRecord<String, String>(topic, value);*

*procuder.send(msg);*

*}*

1. 生产者的send方法将ProducerRecord对象作为参数，所以要先创建一个ProducerRecord对象，其创建构造函数：

*public ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value, Iterable<Header> headers) {*

其中key和value的类型必须与序列化器和生产者对象相匹配。

1. 使用生产者send方法发送ProducerRecord对象，从生产者架构图里可以看出，消息先是被放进缓冲区，然后使用单独的线程发送到服务器端。send方法会返回一个包含RecordMetadata的Future对象。

# RecordAccumulator

生产者调用send方法，不会立即将数据发送到Broker，而是进行Producer缓存。即使用RecordAccumulator接受请求：

*RecordAccumulator.RecordAppendResult result =*

*accumulator.append(tp, timestamp,*

*serializedKey, serializedValue,*

*headers, interceptCallback, remainingWaitMs);*

在RecordAccumulator中，使用队列来存放请求：

*ConcurrentMap<TopicPartition, Deque<ProducerBatch>> batches //根据partition进行分区*

其添加过程如下：

*ProducerBatch batch = new ProducerBatch(tp, recordsBuilder, time.milliseconds());*

*FutureRecordMetadata future = Utils.notNull(batch.tryAppend(timestamp, key, value, headers, callback, time.milliseconds()));*

*dq.addLast(batch);*

至此主线程的信息处理完毕，下一步通过Sender线程将信息发送给Broker。

# Sender线程

Sender线程从RecordAccumulator中获取消息并批量发送，其中Sender中核心成员变量列表如下：

* KafkaClient client，获取每个节点的连接状态
* RecordAccumulator accumulator，待发送的数据
* metadata: Metadata，客户端元数据实例

线程启动后，调用run方法sendProducerData

*private long sendProducerData(long now) {*

*//预处理，查看Partition Leader及Broker节点的状态，移除不可用*

*......*

*//创建producer请求*

*Map<Integer, List<ProducerBatch>> batches = this.accumulator.drain(cluster, result.readyNodes,this.maxRequestSize, now);*

*//发送请求*

*sendProduceRequests(batches, now);*

*}*

SendProduceRequest执行如下：

*private void sendProduceRequests(Map<Integer, List<ProducerBatch>> collated, long now) {*

*for (Map.Entry<Integer, List<ProducerBatch>> entry : collated.entrySet())*

*sendProduceRequest(now, entry.getKey(), acks, requestTimeout, entry.getValue());*

*}*

根据entry.getKey即broker ID来进行请求发送。

*private void sendProduceRequest(long now, int destination, short acks, int timeout, List<ProducerBatch> batches) {*

*......*

*for (ProducerBatch batch : batches) {*

*TopicPartition tp = batch.topicPartition;*

*MemoryRecords records = batch.records();*

*//根据partition的进行分组*

*if (!records.hasMatchingMagic(minUsedMagic))*

*records = batch.records().downConvert(minUsedMagic, 0, time).records();*

*produceRecordsByPartition.put(tp, records);*

*recordsByPartition.put(tp, batch);*

*}*

*//创建与broker的连接*

*String nodeId = Integer.toString(destination);*

*ClientRequest clientRequest = client.newClientRequest(nodeId, requestBuilder, now, acks != 0, callback);*

*client.send(clientRequest, now);*

*}*

NetworkClient将clientRequest添加到InFlightRequest中，

*InFlightRequest inFlightRequest = new InFlightRequest(*

*header,*

*clientRequest.createdTimeMs(),*

*clientRequest.destination(),*

*clientRequest.callback(),*

*clientRequest.expectResponse(),*

*isInternalRequest,*

*request,*

*send,*

*now);*

*this.inFlightRequests.add(inFlightRequest);*

*selector.send(inFlightRequest.send);*

KafkaClient与Broker之间的请求发送是通过JDK NIO来实现，其执行器为Selector，根据connectionId获取KafkaChannel，然后通过KafkaChannel发送请求

*public void send(Send send) {*

*String connectionId = send.destination();*

*KafkaChannel channel = openOrClosingChannelOrFail(connectionId);*

*......*

*channel.setSend(send);*

*......*

*}*

Selector进行请求的轮询发送（channel是否有数据需要发送），Sender线程在执行过程中调用Selector.poll，执行请求的发送：

*void run(long now) {*

*......*

*client.poll(pollTimeout, now); //NetworkClient.poll => Selector.poll*

*}*

Selector#pollSelectionKeys

*void pollSelectionKeys(Set<SelectionKey> selectionKeys,*

*boolean isImmediatelyConnected,*

*long currentTimeNanos) {*

*for (SelectionKey key : determineHandlingOrder(selectionKeys)) {*

*KafkaChannel channel = channel(key);*

*......*

*if (channel.ready() && key.isWritable()) {*

*Send send = null;*

*try {*

*send = channel.write();*

*}*

*}*

执行KafkaChannel.write

*public Send write() throws IOException {*

*Send result = null;*

*if (send != null && send(send)) {*

*.....*

*}*

最终的执行者为TransportLayer，基本实现类：PlaintextTransportLayer

*public class PlaintextTransportLayer implements TransportLayer {*

*private final SelectionKey key;*

*private final SocketChannel socketChannel;*

*......*

*public long write(ByteBuffer[] srcs, int offset, int length) throws IOException {*

*return socketChannel.write(srcs, offset, length);*

*}*

*}*

https://blog.csdn.net/zhanglh046/article/details/72845477

https://blog.csdn.net/ronmy/article/details/60467147