**基本用法**

**实例化KafkaProducer**

一个简单的生产端代码如下：

public class KafkaProducerDemo {

private static final String brokerlist = "10.128.123.250:9092";

private static final String topic = "topic-demo";

public static void main(String[] args) {

Properties props = initConfig();

KafkaProducer<String, String> producer = new KafkaProducer<>(props);

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

try {

producer.send(record);

Thread.sleep(500L);

} catch (Exception e) {

e.printStackTrace();

}

}

public static Properties initConfig() {

Properties props = new Properties();

props.put("bootstrap.servers", brokerlist);

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

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

return props;

}

}

上面初始化Kafka配置的代码，为防止字符串的变量因为书写错误造成不能及时发现，可使用如下进行优化

public static Properties initConfig() {

Properties props = new Properties();

props.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, brokerlist);

props.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class.getName());

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

return props;

}

其中的bootstrap.servers不必配置所有的broker地址，生产者会从给定的broker里查找到其他broker的信息。不过建议至少要设置两个以上的broker 地址信息，当其中任意一个宕机时，生产者仍然可以连接到 Kafka集群上。

KafkaProducer源码中有多个构造函数，如果在创建KafkaProducer时没有设置key.serializer和value.serializer，那么也可以直接通过构造函数传入

KafkaProducer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new StringSerializer());

KafkaProducer是线程安全的，可以在多个线程中共享单个KafkaProducer实例，比使用多实例更快。

官网文档描述：

The producer is *thread safe* and sharing a single producer instance across threads will generally be faster than having multiple instances.

**创建ProducerRecord**

public ProducerRecord(String topic, V value)

public ProducerRecord(String topic, K key, V value)

public ProducerRecord(String topic, Integer partition, K key, V value)

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

public ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)

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

* 若指定了partition，则发送至指定的partition.
* 若没指定partition，但指定了key，则根据key和分区规则指定partition。
* 若既没有指定partition也没有指定key，则round-robin模式发送到每个partition。
* 若既指定了partition又指定了key，则根据partition参数发送到指定partition，key不起作用。

**发送模式**

发送消息主要有三种模式：发后即忘（fire-and-forget）、同步（sync）及异步（async）

**发后即忘**

不关心发送的消息是否到达，对返回结果不作任何处理。

本质上是一种异步发送，性能最高，但可靠性最差。

KafkaProducer<String, String> producer = new KafkaProducer<>(props);

ProducerRecord<String, String> record = new ProducerRecord<>(topic, "hello, Kafka4 !");

try {

producer.send(record);

} catch (Exception e) {

e.printStackTrace();

}

**同步**

在执行send（）方法返回Future对象，并调用了get（）方法来阻塞等待Kafka的响应，直到消息发送成功，或者发生异常。如果发生异常，那么就需要捕获异常并交由外层逻辑处理。

KafkaProducer<String, String> producer = new KafkaProducer<>(props);

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

try {

Future<RecordMetadata> future = producer.send(record);

RecordMetadata metadata = future.get();

} catch (Exception e) {

e.printStackTrace();

}

**异步回调**

在send()方法里指定一个Callback回调函数

producer.send(record, new Callback() {

@Override

public void onCompletion(RecordMetadata metadata, Exception exception) {

if (null != exception) {

exception.printStackTrace();

} else {

System.out.println(metadata.topic() + "-" + metadata.partition() + ":" + metadata.offset());

}

}

});

对于同一个分区，假设record1比record2先发送，那么callback1也会在callback2前先调用。

**发送重试**

KafkaProducer中一般会发生两种类型的异常：可重试的异常和不可重试的异常。常见的可重试异常有：NetworkException、LeaderNotAvailableException、UnknownTopicOrPartitionException、NotEnoughReplicasException、NotCoordinatorException 等。比如NetworkException 表示网络异常，这个有可能是由于网络瞬时故障而导致的异常，可以通过重试解决；又比如LeaderNotAvailableException表示分区的leader副本不可用，这个异常通常发生在leader副本下线而新的 leader 副本选举完成之前，重试之后可以重新恢复。不可重试的异常，比如RecordTooLargeException异常，暗示了所发送的消息太大，KafkaProducer对此不会进行任何重试，直接抛出异常。

对于可重试的异常，如果配置了 retries 参数，那么只要在规定的重试次数内自行恢复了，就不会抛出异常。retries参数的默认值为0，配置方式参考如下

props.put(ProducerConfig.RETRIES\_CONFIG, 10);

**序列化器**

消息的生产者需要使用序列化器将消息转换为字节数组才能通过网络发送给kafka，而消费者则使用反序列化器将从kafka接收到的字节数组转换成相应的对象。

除了自带的org.apache.kafka.common.serialization.StringSerializer外，还有ByteArray、ByteBuffer、Bytes、Double、Integer、Long这几种类型，它们都实现了org.apache.kafka.common.serialization.Serializer接口，此接口有3个方法

/\*\*

\* An interface for converting objects to bytes.

\*

\* A class that implements this interface is expected to have a constructor with no parameter.

\* <p>

\* Implement {@link org.apache.kafka.common.ClusterResourceListener} to receive cluster metadata once it's available. Please see the class documentation for ClusterResourceListener for more information.

\*

\* @param <T> Type to be serialized from.

\*/

public interface Serializer<T> extends Closeable {

/\*\*

\* Configure this class.

\* @param configs configs in key/value pairs

\* @param isKey whether is for key or value

\*/

void configure(Map<String, ?> configs, boolean isKey);

/\*\*

\* Convert {@code data} into a byte array.

\*

\* @param topic topic associated with data

\* @param data typed data

\* @return serialized bytes

\*/

byte[] serialize(String topic, T data);

/\*\*

\* Close this serializer.

\*

\* This method must be idempotent as it may be called multiple times.

\*/

@Override

void close();

}

* configure 方法用于配置当前类，在创建KafkaProducer的时候会被调用。例如在StringSerializer的实现中，读取配置的编码格式key.serializer.encoding、value.serializer.encoding和serializer.encoding，如果都没有配置，则默认使用"UTF-8"。
* serialize 方法非常直观，就是将String类型转为byte[]类型。
* close 一般都是空的实现。源码注解提到该方法必须是幂等的，因为可能被调用多次。

﻿

需要注意的是，生产者使用的序列化器必须与消费者使用的反序列化器一一对应，否则无法解析出想要的数据。

﻿

我们也可以使用Avro、JSON、Thrift、ProtoBuf和Protostuff等通用的序列化工具来实现序列化器，或自定义的序列化器。

**分区器**

如果发送消息的ProducerRecord中已经指定了partition，则无需使用分区器了，因为partition已经指定了需要发送到的分区号。只有在没有指定partition时，分区器才会起作用。

Kafka中提供了默认的分区器org.apache.kafka.clients.producer.internals.DefaultPartitioner，实现了org.apache.kafka.clients.producer.Partitioner接口。

Partitioner还有一个父接口，通过实现configure方法，可通过配置进行一些初始化配置工作。

通过partition方法则可实现分区逻辑。

在默认分区器中，如果key为空，则会计算得到仅为可用分区的分区号中任意一个。如果不为空，可能得到所有分区中的分区号中任意一个。

**拦截器**

拦截器有两种，生产者拦截器和消费者拦截器，这里仅说明生产者拦截器。

生产者拦截器既可以用来在消息发送前做一些准备工作，比如按照某个规则过滤不符合要求的消息、修改消息的内容等，也可以用来在发送回调逻辑前做一些定制化的需求，比如统计类工作。

ProducerInterceptor接口中包含3个方法：

/\*\*

\* A plugin interface that allows you to intercept (and possibly mutate) the records received by the producer before

\* they are published to the Kafka cluster.

\* <p>

\* This class will get producer config properties via <code>configure()</code> method, including clientId assigned

\* by KafkaProducer if not specified in the producer config. The interceptor implementation needs to be aware that it will be

\* sharing producer config namespace with other interceptors and serializers, and ensure that there are no conflicts.

\* <p>

\* Exceptions thrown by ProducerInterceptor methods will be caught, logged, but not propagated further. As a result, if

\* the user configures the interceptor with the wrong key and value type parameters, the producer will not throw an exception,

\* just log the errors.

\* <p>

\* ProducerInterceptor callbacks may be called from multiple threads. Interceptor implementation must ensure thread-safety, if needed.

\* <p>

\* Implement {@link org.apache.kafka.common.ClusterResourceListener} to receive cluster metadata once it's available. Please see the class documentation for ClusterResourceListener for more information.

\*/

public interface ProducerInterceptor<K, V> extends Configurable {

/\*\*

\* This is called from {@link org.apache.kafka.clients.producer.KafkaProducer#send(ProducerRecord)} and

\* {@link org.apache.kafka.clients.producer.KafkaProducer#send(ProducerRecord, Callback)} methods, before key and value

\* get serialized and partition is assigned (if partition is not specified in ProducerRecord).

\* <p>

\* This method is allowed to modify the record, in which case, the new record will be returned. The implication of modifying

\* key/value is that partition assignment (if not specified in ProducerRecord) will be done based on modified key/value,

\* not key/value from the client. Consequently, key and value transformation done in onSend() needs to be consistent:

\* same key and value should mutate to the same (modified) key and value. Otherwise, log compaction would not work

\* as expected.

\* <p>

\* Similarly, it is up to interceptor implementation to ensure that correct topic/partition is returned in ProducerRecord.

\* Most often, it should be the same topic/partition from 'record'.

\* <p>

\* Any exception thrown by this method will be caught by the caller and logged, but not propagated further.

\* <p>

\* Since the producer may run multiple interceptors, a particular interceptor's onSend() callback will be called in the order

\* specified by {@link org.apache.kafka.clients.producer.ProducerConfig#INTERCEPTOR\_CLASSES\_CONFIG}. The first interceptor

\* in the list gets the record passed from the client, the following interceptor will be passed the record returned by the

\* previous interceptor, and so on. Since interceptors are allowed to modify records, interceptors may potentially get

\* the record already modified by other interceptors. However, building a pipeline of mutable interceptors that depend on the output

\* of the previous interceptor is discouraged, because of potential side-effects caused by interceptors potentially failing to

\* modify the record and throwing an exception. If one of the interceptors in the list throws an exception from onSend(), the exception

\* is caught, logged, and the next interceptor is called with the record returned by the last successful interceptor in the list,

\* or otherwise the client.

\*

\* @param record the record from client or the record returned by the previous interceptor in the chain of interceptors.

\* @return producer record to send to topic/partition

\*/

public ProducerRecord<K, V> onSend(ProducerRecord<K, V> record);

/\*\*

\* This method is called when the record sent to the server has been acknowledged, or when sending the record fails before

\* it gets sent to the server.

\* <p>

\* This method is generally called just before the user callback is called, and in additional cases when <code>KafkaProducer.send()</code>

\* throws an exception.

\* <p>

\* Any exception thrown by this method will be ignored by the caller.

\* <p>

\* This method will generally execute in the background I/O thread, so the implementation should be reasonably fast.

\* Otherwise, sending of messages from other threads could be delayed.

\*

\* @param metadata The metadata for the record that was sent (i.e. the partition and offset).

\* If an error occurred, metadata will contain only valid topic and maybe

\* partition. If partition is not given in ProducerRecord and an error occurs

\* before partition gets assigned, then partition will be set to RecordMetadata.NO\_PARTITION.

\* The metadata may be null if the client passed null record to

\* {@link org.apache.kafka.clients.producer.KafkaProducer#send(ProducerRecord)}.

\* @param exception The exception thrown during processing of this record. Null if no error occurred.

\*/

public void onAcknowledgement(RecordMetadata metadata, Exception exception);

/\*\*

\* This is called when interceptor is closed

\*/

public void close();

}

KafkaProducer在将消息序列化和计算分区之前会调用生产者拦截器的onSend（）方法来对消息进行相应的定制化操作。一般来说最好不要修改消息 ProducerRecord 的 topic、key 和partition 等信息，如果要修改，则需确保对其有准确的判断，否则会与预想的效果出现偏差。比如修改key不仅会影响分区的计算，同样会影响broker端日志压缩（Log Compaction）的功能。

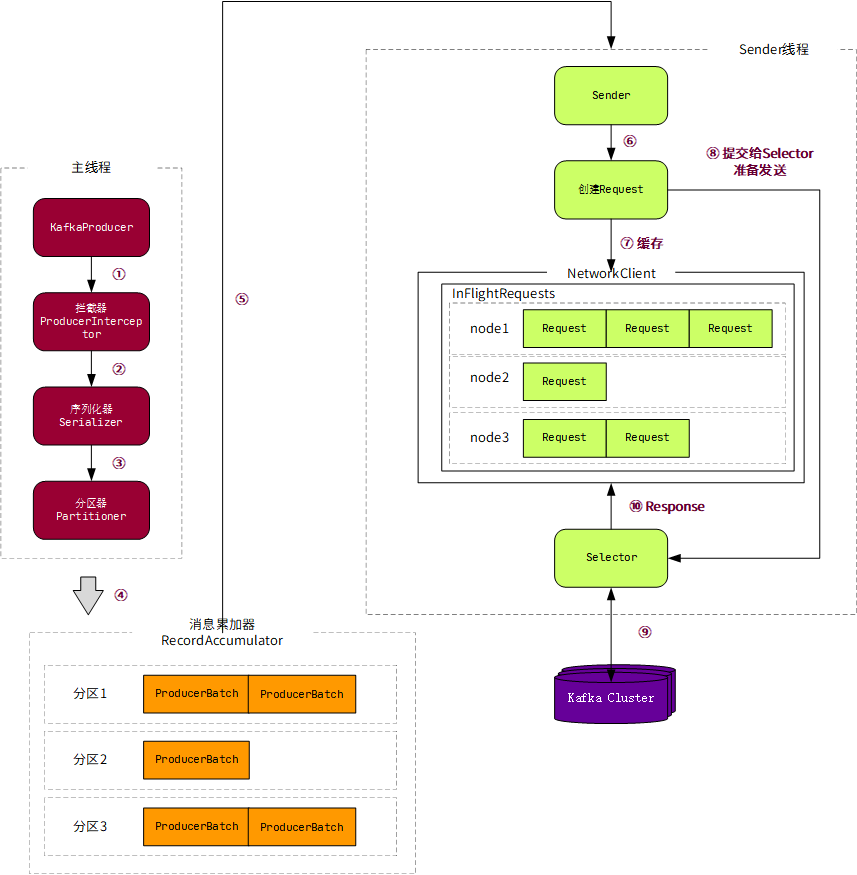
KafkaProducer 会在消息被应答（Acknowledgement）之前或消息发送失败时调用生产者拦截器的onAcknowledgement（）方法，优先于用户设定的 Callback 之前执行。这个方法运行在Producer的 I/O 线程中，所以这个方法中实现的代码逻辑越简单越好，否则会影响消息的发送速度。

Kafka可支持链式的多个拦截器。

**原理分析**

**整体架构**

﻿



﻿

整个生产者客户端由两个线程协调运行。

* 主线程

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。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＜ProducerBatch＞。

﻿

RecordAccumulator用于缓存消息，以便Sender线程能批量发送，进而减少网络传输的资源消耗进而提升性能。

RecordAccumulator缓存的大小可通过生产者客户端参数buffer.memory进行配置，默认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。

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。

﻿

* Sender线程

Sender 线程负责从RecordAccumulator中获取消息并将其发送到Kafka中。

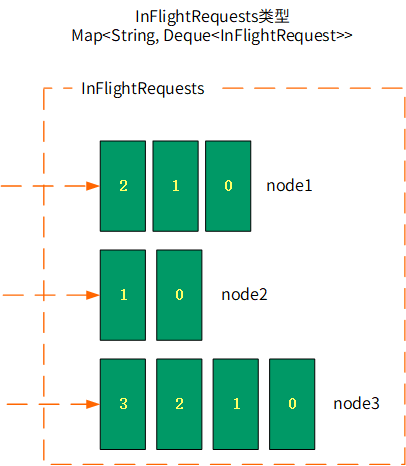
Sender 从 RecordAccumulator 中获取缓存的消息之后，会进一步将原本＜分区，Deque＜ProducerBatch＞＞的保存形式转变成＜Node，List＜ ProducerBatch＞的形式，其中Node表示Kafka集群的broker节点。请求在从Sender线程发往Kafka之前还会保存到InFlightRequests中，InFlightRequests保存对象的具体形式为 Map＜NodeId，Deque＜Request＞＞，它的主要作用是缓存了已经发出去但还没有收到响应的请求（NodeId 是一个 String 类型，表示节点的 id 编号）。

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＜Request＞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**

**leastLoadedNode**

即所有Node中负载最小的那个，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。



如图中所示，负载最小的是节点node2。

因此node2则是leastLoadedNode，如果选择他进行消息发送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。

﻿

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

当我们创建这样一条消息，我们只知道主题和消息内容，对其他信息一无所知。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。

前面介绍了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。

**主要参数**

**acks**

指定分区中必须要有多少个副本收到这条消息，之后生产者才会认为这条消息是成功写入的。

acks 是生产者客户端中一个非常重要的参数，它涉及消息的可靠性和吞吐量之间的权衡。acks参数有3种类型的值（都是字符串类型）。

* acks=1（默认）

生产者发送消息之后，只要分区的leader副本成功写入消息，那么它就会收到来自服务端的成功响应。

如果消息无法写入leader副本，比如在leader 副本崩溃、重新选举新的leader 副本的过程中，那么生产者就会收到一个错误的响应，为了避免消息丢失，生产者可以选择重发消息。

如果消息写入leader副本并返回成功响应给生产者，且在被其他follower副本拉取之前leader副本崩溃，那么此时消息还是会丢失，因为新选举的leader副本中并没有这条对应的消息。

**acks设置为1，是消息可靠性和吞吐量之间的折中方案。**

* acks=0

生产者发送消息之后不需要等待任何服务端的响应。如果在消息从发送到写入Kafka的过程中出现某些异常，导致Kafka并没有收到这条消息，那么生产者也无从得知，消息也就丢失了。在其他配置环境相同的情况下，**acks 设置为 0 可以达到最大的吞吐量**。

* acks=-1或acks=all

生产者在消息发送之后，需要等待ISR中的所有副本都成功写入消息之后才能够收到来自服务端的成功响应。在其他配置环境相同的情况下，acks 设置为-1（all）可以达到最强的可靠性。但这并不意味着消息就一定可靠，因为ISR中可能只有leader副本，这样就退化成了acks=1的情况。要获得更高的消息可靠性需要配合 min.insync.replicas 等参数的联动。

**max.request.size**

限制生产者客户端能发送的消息的最大值，默认值为 1048576B，即 1MB。

﻿

不建议盲目地增大这个参数的配置值，尤其是在对Kafka整体脉络没有足够把控的时候。

因为这个参数还涉及一些其他参数的联动，比如broker端的message.max.bytes参数，如果配置错误可能会引起一些不必要的异常。

比如将broker端的message.max.bytes参数配置为10，而max.request.size参数配置为20，那么当我们发送一条大小为15B的消息时，生产者客户端就会报出如下的异常

org.apache.kafka.common.errors.RecordTooLargeException: The request included a message larger than the max message size the server will accept.

**retries和retry.backoff.ms**

retries参数用来配置生产者重试的次数，默认值为0，该参数用于设置在发生**可重试异常**的时候进行重试的次数。

retry.backoff.ms用来设定两次重试之间的时间间隔，避免无效的频繁重试，默认值为100。

﻿

**关于有序消息**

Kafka 可以保证同一个分区中的消息是有序的。如果生产者按照一定的顺序发送消息，那么这些消息也会顺序地写入分区，进而消费者也可以按照同样的顺序消费它们。

如果将acks参数配置为非零值，并且max.in.flight.requests.per.connection参数配置为大于1的值，那么就会出现错序的现象：如果第一批次消息写入失败，而第二批次消息写入成功，那么生产者会重试发送第一批次的消息，此时如果第一批次的消息写入成功，那么这两个批次的消息就出现了错序。

在需要保证消息顺序的场合建议把参数max.in.flight.requests.per.connection配置为1，而不是把acks配置为0，不过这样也会影响整体的吞吐。

**compression.type**

指定消息的压缩方式，默认值为“none”，即默认情况下，消息不会被压缩。

该参数还可以配置为“gzip”“snappy”和“lz4”。对消息进行压缩可以极大地减少网络传输量、降低网络I/O，从而提高整体的性能。消息压缩是一种使用时间换空间的优化方式，如果对时延有一定的要求，则不推荐对消息进行压缩。

**connections.max.idle.ms**

指定在多久之后关闭闲置的连接，默认值是540000（ms），即9分钟。

**linger.ms**

指定生产者发送 ProducerBatch 之前等待更多消息（ProducerRecord）加入ProducerBatch 的时间，默认值为 0。生产者客户端会在 ProducerBatch 被填满或等待时间超过linger.ms 值时发送出去。增大这个参数的值会增加消息的延迟，但是同时能提升一定的吞吐量。

**batch.size**

即上面提到的ProducerBatch一个批次发送消息的大小。在多个消息发送到同一个分区时，生产者将消息打包在一起，以减少网络开销和请求交互，从而提升性能。

**receive.buffer.bytes**

设置Socket接收消息缓冲区（SO\_RECBUF）的大小，默认值为32768（B），即32KB。如果设置为-1，则使用操作系统的默认值。如果Producer与Kafka处于不同的机房，则可以适地调大这个参数值，因为跨数据中心的网络一般都有比较高的延迟和比较低的带宽。

**send.buffer.bytes**

设置Socket发送消息缓冲区（SO\_SNDBUF）的大小，默认值为131072（B），即128KB。与receive.buffer.bytes参数一样，如果设置为-1，则使用操作系统的默认值。

**request.timeout.ms**

配置Producer等待请求响应的最长时间，默认值为30000（ms）。请求超时之后可以选择进行重试。

注意: