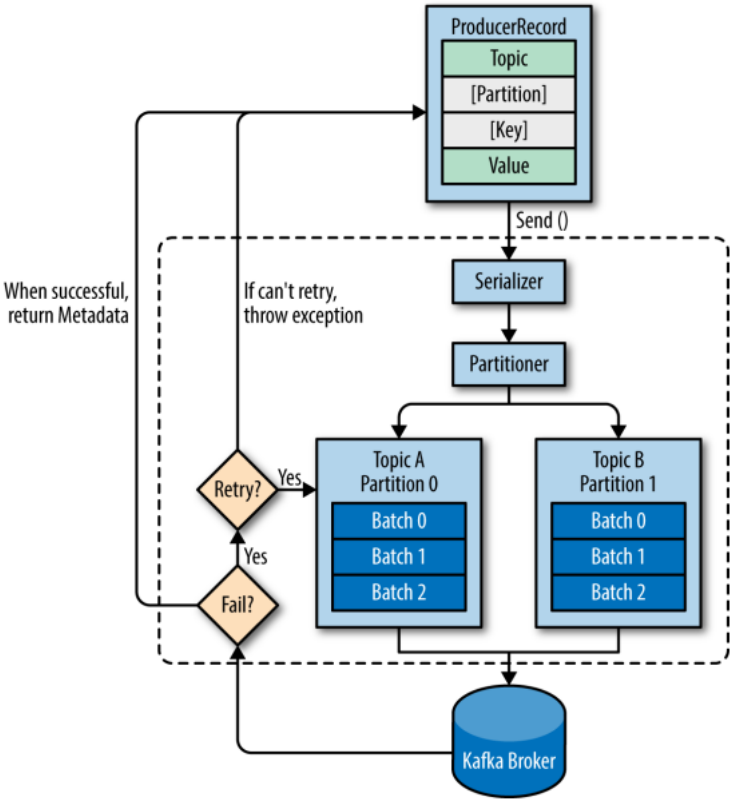
Kafka生产者详解

一、生产者发送消息的过程

首先介绍一下 Kafka 生产者发送消息的过程：

- Kafka 会将发送消息包装为 ProducerRecord 对象， ProducerRecord 对象包含了目标主题和要发送的内容，同时还可以指定键和分区。在发送 ProducerRecord 对象前，生产者会先把键和值对象序列化成字节数组，这样它们才能够在网络上传输。
- 接下来，数据被传给分区器。如果之前已经在 ProducerRecord 对象里指定了分区，那么分区器就不会再做任何事情。如果没有指定分区 ，那么分区器会根据 ProducerRecord 对象的键来选择一个分区，紧接着，这条记录被添加到一个记录批次里，这个批次里的所有消息会被发送到相同的主题和分区上。有一个独立的线程负责把这些记录批次发送到相应的 broker 上。
- 服务器在收到这些消息时会返回一个响应。如果消息成功写入 Kafka，就返回一个 RecordMetaData 对象，它包含了主题和分区信息，以及记录在分区里的偏移量。如果写入失败，则会返回一个错误。生产者在收到错误之后会尝试重新发送消息，如果达到指定的重试次数后还没有成功，则直接抛出异常，不再重试。



二、创建生产者

2.1 项目依赖

本项目采用 Maven 构建，想要调用 Kafka 生产者 API，需要导入 `kafka-clients` 依赖，如下：

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.2.0</version>
</dependency>
```

2.2 创建生产者

创建 Kafka 生产者时，以下三个属性是必须指定的：

- **bootstrap.servers** ： 指定 broker 的地址清单，清单里不需要包含所有的 broker 地址，生产者会从给定的 broker 里查找 broker 的信息。不过建议至少要提供两个 broker 的信息作为容错；
- **key.serializer** ： 指定键的序列化器；
- **value.serializer** ： 指定值的序列化器。

创建的示例代码如下：

```
public class SimpleProducer {

    public static void main(String[] args) {

        String topicName = "Hello-Kafka";

        Properties props = new Properties();
        props.put("bootstrap.servers", "hadoop001:9092");
        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*/
        Producer<String, String> producer = new KafkaProducer<>(props);

        for (int i = 0; i < 10; i++) {
            ProducerRecord<String, String> record = new ProducerRecord<>(topicName, "hello" + i,
                                                                          "world" + i);
            /* 发送消息*/
            producer.send(record);
        }
        /*关闭生产者*/
        producer.close();
    }
}
```

> 本篇文章的所有示例代码可以从 Github 上进行下载：kafka-basis

2.3 测试

1. 启动Kakfa

Kafka 的运行依赖于 zookeeper，需要预先启动，可以启动 Kafka 内置的 zookeeper，也可以启动自己安装的：

```
# zookeeper启动命令
bin/zkServer.sh start

# 内置zookeeper启动命令
bin/zookeeper-server-start.sh config/zookeeper.properties
```

启动单节点 kafka 用于测试：

```
# bin/kafka-server-start.sh config/server.properties
```

2. 创建topic

```
# 创建用于测试主题
bin/kafka-topics.sh --create \
                    --bootstrap-server hadoop001:9092 \
                     --replication-factor 1 --partitions 1 \
                     --topic Hello-Kafka

# 查看所有主题
 bin/kafka-topics.sh --list --bootstrap-server hadoop001:9092
```

3. 启动消费者

启动一个控制台消费者用于观察写入情况，启动命令如下：

```
# bin/kafka-console-consumer.sh --bootstrap-server hadoop001:9092 --topic Hello-Kafka --from-beginning
```

4. 运行项目

此时可以看到消费者控制台，输出如下，这里 `kafka-console-consumer` 只会打印出值信息，不会打印出键信息。

```
[root@localhost kafka_2.12-2.2.0]#  bin/kafka-console-consumer.sh \
>                --bootstrap-server hadoop001:9092 \
>                --topic Hello-Kafka --from-beginning
world0
world1
world2
world3
world4
world5
world6
world7
world8
world9
```

2.4 可能出现的问题

在这里可能出现的一个问题是：生产者程序在启动后，一直处于等待状态。这通常出现在你使用默认配置启动 Kafka 的情况下，此时需要对 `server.properties` 文件中的 `listeners` 配置进行更改：

```
# hadoop001 为我启动kafka服务的主机名，你可以换成自己的主机名或者ip地址
listeners=PLAINTEXT://hadoop001:9092
```

二、发送消息

上面的示例程序调用了 `send` 方法发送消息后没有做任何操作，在这种情况下，我们没有办法知道消息发送的结果。想要知道消息发送的结果，可以使用同步发送或者异步发送来实现。

2.1 同步发送

在调用 `send` 方法后可以接着调用 `get()` 方法， `send` 方法的返回值是一个 Future<RecordMetadata>对象，RecordMetadata 里面包含了发送消息的主题、分区、偏移量等信息。改写后的代码如下：

```
for (int i = 0; i < 10; i++) {
    try {
        ProducerRecord<String, String> record = new ProducerRecord<>(topicName, "k" + i, "world" + i);
        /*同步发送消息*/
        RecordMetadata metadata = producer.send(record).get();
        System.out.printf("topic=%s, partition=%d, offset=%s \n",
                metadata.topic(), metadata.partition(), metadata.offset());
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

此时得到的输出如下：偏移量和调用次数有关，所有记录都分配到了 0 分区，这是因为在创建 `Hello-Kafka` 主题时候，使用 `--partitions` 指定其分区数为 1，即只有一个分区。

```
topic=Hello-Kafka, partition=0, offset=40
topic=Hello-Kafka, partition=0, offset=41
topic=Hello-Kafka, partition=0, offset=42
topic=Hello-Kafka, partition=0, offset=43
topic=Hello-Kafka, partition=0, offset=44
topic=Hello-Kafka, partition=0, offset=45
topic=Hello-Kafka, partition=0, offset=46
topic=Hello-Kafka, partition=0, offset=47
topic=Hello-Kafka, partition=0, offset=48
topic=Hello-Kafka, partition=0, offset=49
```

2.2 异步发送

通常我们并不关心发送成功的情况，更多关注的是失败的情况，因此 Kafka 提供了异步发送和回调函数。 代码如下：

```
for (int i = 0; i < 10; i++) {
    ProducerRecord<String, String> record = new ProducerRecord<>(topicName, "k" + i, "world" + i);
    /*异步发送消息，并监听回调*/
    producer.send(record, new Callback() {
        @Override
        public void onCompletion(RecordMetadata metadata, Exception exception) {
            if (exception != null) {
                System.out.println("进行异常处理");
            } else {
                System.out.printf("topic=%s, partition=%d, offset=%s \n",
                        metadata.topic(), metadata.partition(), metadata.offset());
            }
        }
```

```
    });
}
```

三、自定义分区器

Kafka 有着默认的分区机制：

- 如果键值为 null， 则使用轮询 (Round Robin) 算法将消息均衡地分布到各个分区上；
- 如果键值不为 null，那么 Kafka 会使用内置的散列算法对键进行散列，然后分布到各个分区上。

某些情况下，你可能有着自己的分区需求，这时候可以采用自定义分区器实现。这里给出一个自定义分区器的示例：

3.1 自定义分区器

```
/**
 * 自定义分区器
 */
public class CustomPartitioner implements Partitioner {

    private int passLine;

    @Override
    public void configure(Map<String, ?> configs) {
        /*从生产者配置中获取分数线*/
        passLine = (Integer) configs.get("pass.line");
    }

    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object value,
                         byte[] valueBytes, Cluster cluster) {
        /*key 值为分数，当分数大于分数线时候，分配到 1 分区，否则分配到 0 分区*/
        return (Integer) key >= passLine ? 1 : 0;
    }

    @Override
    public void close() {
        System.out.println("分区器关闭");
```

需要在创建生产者时指定分区器，和分区器所需要的配置参数：

```
public class ProducerWithPartitioner {

    public static void main(String[] args) {

        String topicName = "Kafka-Partitioner-Test";

        Properties props = new Properties();
        props.put("bootstrap.servers", "hadoop001:9092");
        props.put("key.serializer", "org.apache.kafka.common.serialization.IntegerSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        /*传递自定义分区器*/
        props.put("partitioner.class", "com.heibaiying.producers.partitioners.CustomPartitioner");
        /*传递分区器所需的参数*/
        props.put("pass.line", 6);

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

        for (int i = 0; i <= 10; i++) {
            String score = "score:" + i;
            ProducerRecord<Integer, String> record = new ProducerRecord<>(topicName, i, score);
            /*异步发送消息*/
            producer.send(record, (metadata, exception) ->
```

3.2 测试

需要创建一个至少有两个分区的主题：

```
bin/kafka-topics.sh --create \
                    --bootstrap-server hadoop001:9092 \
                     --replication-factor 1 --partitions 2 \
                     --topic Kafka-Partitioner-Test
```

此时输入如下，可以看到分数大于等于 6 分的都被分到 1 分区，而小于 6 分的都被分到了 0 分区。

```
score:6, partition=1,
score:7, partition=1,
score:8, partition=1,
score:9, partition=1,
score:10, partition=1,
score:0, partition=0,
score:1, partition=0,
score:2, partition=0,
score:3, partition=0,
score:4, partition=0,
score:5, partition=0,
分区器关闭
```

四、生产者其他属性

上面生产者的创建都仅指定了服务地址，键序列化器、值序列化器，实际上 Kafka 的生产者还有很多可配置属性，如下：

1. acks

acks 参数指定了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是成功的：

- **acks=0** ： 消息发送出去就认为已经成功了，不会等待任何来自服务器的响应；
- **acks=1** ： 只要集群的首领节点收到消息，生产者就会收到一个来自服务器成功响应；
- **acks=all** ： 只有当所有参与复制的节点全部收到消息时，生产者才会收到一个来自服务器的成功响应。

2. buffer.memory

设置生产者内存缓冲区的大小。

3. compression.type

默认情况下，发送的消息不会被压缩。如果想要进行压缩，可以配置此参数，可选值有 snappy，gzip，lz4。

4. retries

发生错误后，消息重发的次数。如果达到设定值，生产者就会放弃重试并返回错误。

5. batch.size

当有多个消息需要被发送到同一个分区时，生产者会把它们放在同一个批次里。该参数指定了一个批次可以使用的内存大小，按照字节数计算。

6. linger.ms

该参数制定了生产者在发送批次之前等待更多消息加入批次的时间。

7. clent.id

客户端 id,服务器用来识别消息的来源。

8. max.in.flight.requests.per.connection

指定了生产者在收到服务器响应之前可以发送多少个消息。它的值越高，就会占用越多的内存，不过也会提升吞吐量，把它设置为 1 可以保证消息是按照发送的顺序写入服务器，即使发生了重试。

9. timeout.ms, request.timeout.ms & metadata.fetch.timeout.ms

- timeout.ms 指定了 borker 等待同步副本返回消息的确认时间；
- request.timeout.ms 指定了生产者在发送数据时等待服务器返回响应的时间；
- metadata.fetch.timeout.ms 指定了生产者在获取元数据（比如分区首领是谁）时等待服务器返回响应的时间。

10. max.block.ms

指定了在调用 `send()` 方法或使用 `partitionsFor()` 方法获取元数据时生产者的阻塞时间。当生产者的发送缓冲区已满，或者没有可用的元数据时，这些方法会阻塞。在阻塞时间达到 max.block.ms 时，生产者会抛出超时异常。

11. max.request.size

该参数用于控制生产者发送的请求大小。它可以指发送的单个消息的最大值，也可以指单个请求里所有消息总的大小。例如，假设这个值为 1000K ，那么可以发送的单个最大消息为 1000K ，或者生产者可以在单个请求里发送一个批次，该批次包含了 1000 个消息，每个消息大小为 1K。

12. receive.buffer.bytes & send.buffer.byte

这两个参数分别指定 TCP socket 接收和发送数据包缓冲区的大小，-1 代表使用操作系统的默认值。