生产者-客户端开发

从编程的角度而言，生产者就是负责向 Kafka 发送消息的应用程序。在 Kafka 的历史变迁中，一共有两个大版本的生产者客户端：第一个是于 Kafka 开源之初使用 Scala 语言编写的客户端，我们可以称之为旧生产者客户端（Old Producer）或 Scala 版生产者客户端；第二个是从 Kafka 0.9.x 版本开始推出的使用 Java 语言编写的客户端，我们可以称之为新生产者客户端（New Producer）或 Java 版生产者客户端，它弥补了旧版客户端中存在的诸多设计缺陷。

虽然 Kafka 是用 Java/Scala 语言编写的，但这并不妨碍它对于多语言的支持，在 Kafka 官网中，“CLIENTS (https://cwiki.apache.org/confluence/display/KAFKA/Clients)”入口提供了一份多语言的支持列表，其中包括常用的 C/C++、Python、Go 等语言，不过这些其他类语言的客户端并非由 Kafka 社区维护，如果使用则需要另行下载。本章主要针对现下流行的新生产者（Java 语言编写的）客户端做详细介绍，而旧生产者客户端已被淘汰，故不再做相应的介绍了。

客户端开发

一个正常的生产逻辑需要具备以下几个步骤：

1. 配置生产者客户端参数及创建相应的生产者实例。
2. 构建待发送的消息。
3. 发送消息。
4. 关闭生产者实例。

代码清单2-1中已经简单对生产者客户端的编码做了一个基本演示，本节对其修改以做具体的分析，如代码清单3-1所示。

```
//代码清单3-1 生产者客户端示例代码
```

```
public class KafkaProducerAnalysis {
    public static final String brokerList =
"localhost:9092";
    public static final String topic = "topic-
demo";

    public static Properties initConfig(){
        Properties props = new Properties();
        props.put("bootstrap.servers",
brokerList);
        props.put("key.serializer",

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

"org.apache.kafka.common.serialization.StringSeri
alizer");
        properties.put("client.id",
"producer.client.id.demo");
        return props;
    }

    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);
        } catch (Exception e) {
```

```
                e.printStackTrace();
            }
        }
}
```

相比代码清单2-1而言，这里仅仅是让编码的逻辑显得更加“正统”一些，也更加方便下面内容的陈述。

这里有必要单独说明的是构建的消息对象 ProducerRecord，它并不是单纯意义上的消息，它包含了多个属性，原本需要发送的与业务相关的消息体只是其中的一个 value 属性，比如“Hello, Kafka!”只是 ProducerRecord 对象中的一个属性。ProducerRecord 类的定义如下（只截取成员变量）：

```
public class ProducerRecord<K, V> {
    private final String topic; //主题
    private final Integer partition; //分区号
    private final Headers headers; //消息头部
    private final K key; //键
    private final V value; //值
    private final Long timestamp; //消息的时间戳
    //省略其他成员方法和构造方法
}
```

其中 topic 和 partition 字段分别代表消息要发往的主题和分区号。headers 字段是消息的头部，Kafka 0.11.x 版本才引入这个属性，它大多用来设定一些与应用相关的信息，如无需要也可以不用设置。key 是用来指定消息的键，它不仅是消息的附加信息，还可以用来计算分区号进而可以让消息发往特定的分区。前面提及消息以主题为单位进行归类，而这个 key 可以让消息再进行二次归类，同一个 key 的消息会被划分到同一个分区中，详情参见第4节中的分区器。

有 key 的消息还可以支持日志压缩的功能。value 是指消息体，一般不为空，如果为空则表示特定的消息—墓碑消息。timestamp 是指消息的时间戳，它有 CreateTime 和 LogAppendTime 两种类型，前者表示消息创建的时间，后者表示消息追加到日志文件的时间。以上这些深入原理性的东西都会在《图解Kafka之核心原理》(https://juejin.im/book/5c7d270ff265da2d89634e9e)中呈现给大家。

接下来我们将按照生产逻辑的各个步骤来一一做相应分析。

必要的参数配置

在创建真正的生产者实例前需要配置相应的参数，比如需要连接的 Kafka 集群地址。参照代码清单3-1中的 initConfig()方法，在 Kafka 生产者客户端 KafkaProducer 中有3个参数是必填的。

- **bootstrap.servers**：该参数用来指定生产者客户端连接 Kafka 集群所需的 broker 地址清单，具体的内容格式为 host1:port1,host2:port2，可以设置一个或多个地址，中间以逗号隔开，此参数的默认值为“”。注意这里并非需要所有的 broker 地址，因为生产者会从给定的 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。
- **key.serializer** 和 **value.serializer**：broker 端接收的消息必须以字节数组（byte[]）的形式存在。代码清单3-1中生产者使用的 KafkaProducer<String, String>和 ProducerRecord<String, String> 中的泛型 <String, String> 对应的就是消息中 key 和 value 的类型，生产者客户端使用这种方式可以让代码具有良好的可读性，不过在发往 broker 之前需要将消息中对应的 key 和 value 做相应的序列化操作来转换成字节数组。key.serializer 和 value.serializer 这两个参数分别用来指定 key 和 value 序列化操作的序列化

器，这两个参数无默认值。注意这里必须填写序列化器的全限定名，如代码清单3-1中的 org.apache.kafka.common.serialization.StringSerializer，单单指定 StringSerializer 是错误的，更多有关序列化的内容可以参考第4节。

注意到代码清单3-1中的 initConfig() 方法里还设置了一个参数 client.id，这个参数用来设定 KafkaProducer 对应的客户端id，默认值为“”。如果客户端不设置，则 KafkaProducer 会自动生成一个非空字符串，内容形式如“producer-1”、“producer-2”，即字符串“producer-”与数字的拼接。

KafkaProducer 中的参数众多，远非示例 initConfig()方法中的那样只有4个，开发人员可以根据业务应用的实际需求来修改这些参数的默认值，以达到灵活调配的目的。一般情况下，普通开发人员无法记住所有的参数名称，只能有个大致的印象。

在实际使用过程中，诸如“key.serializer”、“max.request.size”、“interceptor.classes”之类的字符串经常由于人为因素而书写错误。为此，我们可以直接使用客户端中的 org.apache.kafka.clients.producer.ProducerConfig 类来做一定程度上的预防措施，每个参数在 ProducerConfig 类中都有对应的名称，以代码清单3-1中的 initConfig()方法为例，引入 ProducerConfig 后的修改结果如下：

```
public static Properties initConfig(){
    Properties props = new Properties();

props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG
, brokerList);

props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CON
FIG,

"org.apache.kafka.common.serialization.StringSeri
alizer");

props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_C
ONFIG,

"org.apache.kafka.common.serialization.StringSeri
alizer");
    props.put(ProducerConfig.CLIENT_ID_CONFIG,
"producer.client.id.demo");
    return props;
}
```

注意到上面的代码中 key.serializer 和 value.serializer 参数对应类的全限定名比较长，也比较容易写错，这里通过 Java 中的技巧来做进一步的改进，相关代码如下：

```
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CON
FIG,
        StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_C
ONFIG,
        StringSerializer.class.getName());
```

如此代码便简洁了许多，同时进一步降低了人为出错的可能性。在配置完参数之后，我们就可以使用它来创建一个生产者实例，示例如下：

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

KafkaProducer 是线程安全的，可以在多个线程中共享单个 KafkaProducer 实例，也可以将 KafkaProducer 实例进行池化来供其他线程调用。

KafkaProducer 中有多个构造方法，比如在创建 KafkaProducer 实例时并没有设定 key.serializer 和 value.serializer 这两个配置参数，那么就需要在构造方法中添加对应的序列化器，示例如下：

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

其内部原理和无序列化器的构造方法一样，不过就实际应用而言，一般都选用 public KafkaProducer(Properties properties)这个构造方法来创建 KafkaProducer 实例。

消息的发送

在创建完生产者实例之后，接下来的工作就是构建消息，即创建 ProducerRecord 对象。通过代码清单3-1中我们已经了解了 ProducerRecord 的属性结构，其中 topic 属性和 value 属性是必填项，其余属性是选填项，对应的 ProducerRecord 的构造方法也有多种，参考如下：

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

代码清单3-1中使用的是最后一种构造方法，也是最简单的一种，这种方式相当于将 ProducerRecord 中除 topic 和 value 外的属性全部值设置为 null。在实际的应用中，还会用到其他构造方法，比如要指定 key，或者添加 headers 等。有可能会遇到这些构造方法都不满足需求的情况，需要自行添加更多的构造方法，比如下面的示例：

```
public ProducerRecord(String topic, Long
timestamp,
                      V value, Iterable<Header>
headers)
```

注意，针对不同的消息，需要构建不同的 ProducerRecord 对象，在实际应用中创建 ProducerRecord 对象是一个非常频繁的动作。

创建生产者实例和构建消息之后，就可以开始发送消息了。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）。

代码清单3-1中的这种发送方式就是发后即忘，它只管往 Kafka 中发送消息而并不关心消息是否正确到达。在大多数情况下，这种发送方式没有什么问题，不过在某些时候（比如发生不可重试异常时）会造成消息的丢失。这种发送方式的性能最高，可靠性也最差。

KafkaProducer 的 send()方法并非是 void 类型，而是 Future类型，send()方法有2个重载方法，具体定义如下：

```
public Future<RecordMetadata>
send(ProducerRecord<K, V> record)
public Future<RecordMetadata>
send(ProducerRecord<K, V> record,
                                    Callback
callback)
```

要实现同步的发送方式，可以利用返回的 Future 对象实现，示例如下：

```
try {
    producer.send(record).get();
} catch (ExecutionException |
InterruptedException e) {
    e.printStackTrace();
}
```

实际上 send() 方法本身就是异步的，send() 方法返回的 Future 对象可以使调用方稍后获得发送的结果。示例中在执行 send() 方法之后直接链式调用了 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。

也可以在执行完 send() 方法之后不直接调用 get() 方法，比如下面的一种同步发送方式的实现：

```
try {
    Future<RecordMetadata> future =
producer.send(record);
    RecordMetadata metadata = future.get();
    System.out.println(metadata.topic() + "-" +
            metadata.partition() + ":" +
metadata.offset());
} catch (ExecutionException |
InterruptedException e) {
    e.printStackTrace();
}
```

这样可以获取一个 RecordMetadata 对象，在 RecordMetadata 对象里包含了消息的一些元数据信息，比如当前消息的主题、分区号、分区中的偏移量（offset）、时间戳等。如果在应用代码中需要这些信息，则可以使用这个方式。如果不需要，则直接采用 producer.send(record).get() 的方式更省事。

Future 表示一个任务的生命周期，并提供了相应的方法来判断任务是否已经完成或取消，以及获取任务的结果和取消任务等。既然 KafkaProducer.send() 方法的返回值是一个 Future 类型的对象，那么完全可以用 Java 语言层面的技巧来丰富应用的实现，比如使用 Future 中的 get(long timeout, TimeUnit unit) 方法实现可超时的阻塞。

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，比如第2节中提及的 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);
```

示例中配置了10次重试。如果重试了10次之后还没有恢复，那么仍会抛出异常，进而发送的外层逻辑就要处理这些异常了。

同步发送的方式可靠性高，要么消息被发送成功，要么发生异常。如果发生异常，则可以捕获并进行相应的处理，而不会像“发后即忘”的方式直接造成消息的丢失。不过同步发送的方式的性能会差很多，需要阻塞等待一条消息发送完之后才能发送下一条。

我们再来了解一下异步发送的方式，一般是在 send() 方法里指定一个 Callback 的回调函数，Kafka 在返回响应时调用该函数来实现异步的发送确认。

有读者或许会有疑问，send() 方法的返回值类型就是 Future，而 Future 本身就可以用作异步的逻辑处理。这样做不是不行，只不过 Future 里的 get() 方法在何时调用，以及怎么调用都是需要面对的问题，消息不停地发送，那么诸多消息对应的 Future 对象的处理难免会引起代码处理逻辑的混乱。使用 Callback 的方式非常简洁明了，Kafka 有响应时就会回调，要么发送成功，要么抛出异常。

异步发送方式的示例如下：

```
producer.send(record, new Callback() {
    @Override
    public void onCompletion(RecordMetadata
metadata, Exception exception) {
        if (exception != null) {
            exception.printStackTrace();
        } else {
            System.out.println(metadata.topic() +
"-" +
                    metadata.partition() + ":" +
metadata.offset());
        }
    }
});
```

示例代码中遇到异常时（exception!=null）只是做了简单的打印操作，在实际应用中应该使用更加稳妥的方式来处理，比如可以将异常记录以便日后分析，也可以做一定的处理来进行消息重发。
onCompletion() 方法的两个参数是互斥的，消息发送成功时，metadata 不为 null 而 exception 为 null；消息发送异常时，metadata 为 null 而 exception 不为 null。

```
producer.send(record1, callback1);
producer.send(record2, callback2);
```

对于同一个分区而言，如果消息 record1 于 record2 之前先发送（参考上面的示例代码），那么 KafkaProducer 就可以保证对应的 callback1 在 callback2 之前调用，也就是说，回调函数的调用也可以保证分区有序。

通常，一个 KafkaProducer 不会只负责发送单条消息，更多的是发送多条消息，在发送完这些消息之后，需要调用 KafkaProducer 的 close() 方法来回收资源。下面的示例中发送了100条消息，之后就调用了 close() 方法来回收所占用的资源：

```
int i = 0;
while (i < 100) {
    ProducerRecord<String, String> record =
            new ProducerRecord<>(topic,
"msg"+i++);
    try {
        producer.send(record).get();
    } catch (InterruptedException |
ExecutionException e) {
        e.printStackTrace();
    }
}
producer.close();
```

close() 方法会阻塞等待之前所有的发送请求完成后再关闭 KafkaProducer。与此同时，KafkaProducer 还提供了一个带超时时间的 close() 方法，具体定义如下：

```
public void close(long timeout, TimeUnit
timeUnit)
```

如果调用了带超时时间 timeout 的 close() 方法，那么只会在等待 timeout 时间内来完成所有尚未完成的请求处理，然后强行退出。在实际应用中，一般使用的都是无参的 close() 方法。