KAFKA--事务和幂等(二)

标签： kafka

消息传输保障

一般而言，消息中间件的消息传输保障有3个层级，分别如下。

1. at most once：至多一次。消息可能会丢失，但绝对不会重复传输。
2. at least once：最少一次。消息绝不会丢失，但可能会重复传输。
3. exactly once：恰好一次。每条消息肯定会被传输一次且仅传输一次。

Kafka 的消息传输保障机制非常直观。当生产者向 Kafka 发送消息时，一旦消息被成功提交到日志文件，由于多副本机制的存在，这条消息就不会丢失。如果生产者发送消息到 Kafka 之后，遇到了网络问题而造成通信中断，那么生产者就无法判断该消息是否已经提交。虽然 Kafka 无法确定网络故障期间发生了什么，但生产者可以进行多次重试来确保消息已经写入 Kafka，这个重试的过程中有可能会造成消息的重复写入，所以这里 Kafka 提供的消息传输保障为 at least once。

对消费者而言，消费者处理消息和提交消费位移的顺序在很大程度上决定了消费者提供哪一种消息传输保障。如果消费者在拉取完消息之后，应用逻辑先处理消息后提交消费位移，那么在消息处理之后且在位移提交之前消费者宕机了，待它重新上线之后，会从上一次位移提交的位置拉取，这样就出现了重复消费，因为有部分消息已经处理过了只是还没来得及提交消费位移，此时就对应 at least once。

如果消费者在拉完消息之后，应用逻辑先提交消费位移后进行消息处理，那么在位移提交之后且在消息处理完成之前消费者宕机了，待它重新上线之后，会从已经提交的位移处开始重新消费，但之前尚有部分消息未进行消费，如此就会发生消息丢失，此时就对应 at most once。

Kafka 从 0.11.0.0 版本开始引入了幂等和事务这两个特性，以此来实现 EOS（exactly once semantics，精确一次处理语义）。

幂等

所谓的幂等，简单地说就是对接口的多次调用所产生的结果和调用一次是一致的。生产者在进行重试的时候有可能会重复写入消息，而使用 Kafka 的幂等性功能之后就可以避免这种情况。

开启幂等性功能的方式很简单，只需要显式地将生产者客户端参数 enable.idempotence 设置为 true 即可（这个参数的默认值为 false），参考如下：

```
properties.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
# 或者
properties.put(“enable.idempotence”, true);
```

不过如果要确保幂等性功能正常，还需要确保生产者客户端的 retries、acks、max.in.flight.requests.per.connection 这几个参数不被配置错。实际上在使用幂等性功能的时候，用户完全可以不用配置（也不建议配置）这几个参数。

如果用户显式地指定了 retries 参数，那么这个参数的值必须大于0，否则会报出 ConfigException：

```
che.kafka.common.config.ConfigException: Must set retries to non-zero when using the idempotent producer.
```

如果用户没有显式地指定 retries 参数，那么 KafkaProducer 会将它置为 Integer.MAX_VALUE。同时还需要保证 max.in.flight.requests.per.connection 参数的值不能大于5（这个参数的值默认为5，在 2.2.1 节中有相关的介绍），否则也会报出 ConfigException：

```
org.apache.kafka.common.config.ConfigException: Must set max.in.flight. requests.per.connection to at mos
```

如果用户还显式地指定了 acks 参数，那么还需要保证这个参数的值为 -1（all），如果不为 -1（这个参数的值默认为 1），那么也会报出 ConfigException：

```
org.apache.kafka.common.config.ConfigException: Must set acks to all in order to use the idempotent produ
```

如果用户没有显式地指定这个参数，那么 KafkaProducer 会将它置为-1。开启幂等性功能之后，生产者就可以如同未开启幂等时一样发送消息了。

为了实现生产者的幂等性，Kafka 为此引入了 producer id（以下简称 PID）和***（sequence number）这两个概念，这两个概念其实在第2节中就讲过，分别对应 v2 版的日志格式中 RecordBatch 的 producer id 和 first seqence 这两个字段（参考第2节）。

每个新的生产者实例在初始化的时候都会被分配一个 PID，这个 PID 对用户而言是完全透明的。对于每个 PID，消息发送到的每一个分区都有对应的***，这些***从0开始单调递增。生产者每发送一条消息就会将 <PID，分区> 对应的***的值加1。

broker 端会在内存中为每一对 <PID，分区> 维护一个***。对于收到的每一条消息，只有当它的***的值（SN_new）比 broker 端中维护的对应的***的值（SN_old）大1（即 SN_new = SN_old + 1）时，broker 才会接收它。如果 SN_new< SN_old + 1，那么说明消息被重复写入，broker 可以直接将其丢弃。如果 SN_new> SN_old + 1，那么说明中间有数据尚未写入，出现了乱序，暗示可能有消息丢失，对应的生产者会抛出 OutOfOrderSequenceException，这个异常是一个严重的异常，后续的诸如 send()、beginTransaction()、commitTransaction() 等方法的调用都会抛出 IllegalStateException 的异常。

引入***来实现幂等也只是针对每一对 <PID，分区> 而言的，也就是说，Kafka 的幂等只能保证单个生产者会话（session）中单分区的幂等。

```
ProducerRecord<String, String> record
      = new ProducerRecord<>(topic, "key", "msg");
producer.send(record);
producer.send(record);
```

注意，上面示例中发送了两条相同的消息，不过这仅仅是指消息内容相同，但对 Kafka 而言是两条不同的消息，因为会为这两条消息分配不同的***。Kafka 并不会保证消息内容的幂等。

事务

幂等性并不能跨多个分区运作，而事务可以弥补这个缺陷。事务可以保证对多个分区写入操作的原子性。操作的原子性是指多个操作要么全部成功，要么全部失败，不存在部分成功、部分失败的可能。

对流式应用（Stream Processing Applications）而言，一个典型的应用模式为“consume-transform-produce”。在这种模

```
properties.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "transactionId");
# 或者
properties.put("transactional.id", "transactionId");
```

事务要求生产者开启幂等特性，因此通过将 transactional.id 参数设置为非空从而开启事务特性的同时需要将 enable.idempotence 设置为 true（如果未显式设置，则 KafkaProducer 默认会将它的值设置为 true），如果用户显式地将 enable.idempotence 设置为 false，则会报出 ConfigException：

```
org.apache.kafka.common.config.ConfigException: Cannot set a transactional.id without also enabling idemp
```

transactionalId 与 PID 一一对应，两者之间所不同的是 transactionalId 由用户显式设置，而 PID 是由 Kafka 内部分配的。另外，为了保证新的生产者启动后具有相同 transactionalId 的旧生产者能够立即失效，每个生产者通过 transactionalId 获取 PID 的同时，还会获取一个单调递增的 producer epoch（对应下面要讲述的 KafkaProducer.initTransactions() 方法）。如果使用同一个 transactionalId 开启两个生产者，那么前一个开启的生产者会报出如下的错误：

```
org.apache.kafka.common.errors.ProducerFencedException: Producer attempted an operation with an old epoch
```

producer epoch 同 PID 和***一样在第2节中就讲过了，对应v2版的日志格式中 RecordBatch 的 producer epoch 字段。

从生产者的角度分析，通过事务，Kafka 可以保证跨生产者会话的消息幂等发送，以及跨生产者会话的事务恢复。前者表示具有相同 transactionalId 的新生产者实例被创建且工作的时候，旧的且拥有相同 transactionalId 的生产者实例将不再工作。后者指当某个生产者实例宕机后，新的生产者实例可以保证任何未完成的旧事务要么被提交（Commit），要么被中止（Abort），如此可以使新的生产者实例从一个正常的状态开始工作。

而从消费者的角度分析，事务能保证的语义相对偏弱。出于以下原因，Kafka 并不能保证已提交的事务中的所有消息都能够被消费：

- 对采用日志压缩策略的主题而言，事务中的某些消息有可能被清理（相同key的消息，后写入的消息会覆盖前面写入的消息）。
- 事务中消息可能分布在同一个分区的多个日志分段（LogSegment）中，当老的日志分段被删除时，对应的消息可能会丢失。
- 消费者可以通过 seek() 方法访问任意 offset 的消息，从而可能遗漏事务中的部分消息。
- 消费者在消费时可能没有分配到事务内的所有分区，如此它也就不能读取事务中的所有消息。

KafkaProducer 提供了5个与事务相关的方法，详细如下：

```
void initTransactions();
void beginTransaction() throws ProducerFencedException;
void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets,
                              String consumerGroupId)
        throws ProducerFencedException;
void commitTransaction() throws ProducerFencedException;
void abortTransaction() throws ProducerFencedException;
```

initTransactions() 方法用来初始化事务，这个方法能够执行的前提是配置了 transactionalId，如果没有则会报出 IllegalStateException：

```
java.lang.IllegalStateException: Cannot use transactional methods without enabling transactions by settin
```

beginTransaction() 方法用来开启事务；sendOffsetsToTransaction() 方法为消费者提供在事务内的位移提交的操作；commitTransaction() 方法用来提交事务；abortTransaction() 方法用来中止事务，类似于事务回滚。

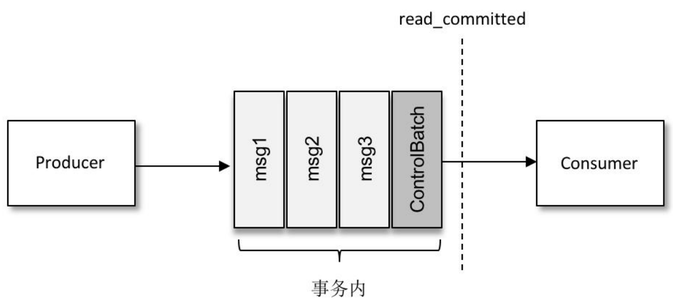
一个典型的事务消息发送的操作如代码清单14-1所示。

```
代码清单14-1 事务消息发送示例
Properties properties = new Properties();
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class.getName());
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class.getName());
properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, brokerList);
properties.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, transactionId);

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

producer.initTransactions();
producer.beginTransaction();

try {
    //处理业务逻辑并创建ProducerRecord
    ProducerRecord<String, String> record1 = new ProducerRecord<>(topic, "msg1");
    producer.send(record1);
    ProducerRecord<String, String> record2 = new ProducerRecord<>(topic, "msg2");
    producer.send(record2);
    ProducerRecord<String, String> record3 = new ProducerRecord<>(topic, "msg3");
    producer.send(record3);
    //处理一些其他逻辑
    producer.commitTransaction();
} catch (ProducerFencedException e) {
    producer.abortTransaction();
}
producer.close();
```

在消费端有一个参数 isolation.level，与事务有着莫大的关联，这个参数的默认值为“read_uncommitted”，意思是说消费端应用可以看到（消费到）未提交的事务，当然对于已提交的事务也是可见的。这个参数还可以设置为“read_committed”，表示消费端应用不可以看到尚未提交的事务内的消息。

举个例子，如果生产者开启事务并向某个分区值发送3条消息 msg1、msg2 和 msg3，在执行 commitTransaction() 或 abortTransaction() 方法前，设置为“read_committed”的消费端应用是消费不到这些消息的，不过在 KafkaConsumer 内部会缓存这些消息，直到生产者执行 commitTransaction() 方法之后它才能将这些消息推送给消费端应用。反之，如果生产者执行了 abortTransaction() 方法，那么 KafkaConsumer 会将这些缓存的消息丢弃而不推送给消费端应用。

日志文件中除了普通的消息，还有一种消息专门用来标志一个事务的结束，它就是控制消息（ControlBatch）。控制消息一共有两种类型：COMMIT 和 ABORT，分别用来表征事务已经成功提交或已经被成功中止。KafkaConsumer 可以通过这个控制消息来判断对应的事务是被提交了还是被中止了，然后结合参数 isolation.level 配置的隔离级别来决定是否将相应的消息返回给消费端应用，如下图所示。注意 ControlBatch 对消费端应用不可见，后面还会对它有更加详细的介绍。



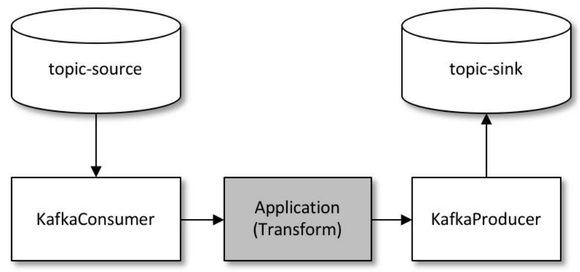
推荐文章

- 数据回显---SPRINGMVC学习笔记（九）
- JUPYTER NOTEBOOK远程连接服务器（WINDOWS浏览器访问）
- SPRING CLOUD OPENFEIGN 源码解析（二）FEIGN CLIENT 的创建
- 提高代码优雅-LOMBOK代替GETTER/SETTER方法
- OKHTTP源码笔记之流程简析
- DIV带箭头提示框实例
- UDP LOCAL SERVER--PYTHON网络编程学习笔记
- LINUX下的PYTHON3.6报错MODULENOTFOUNDERROR: NO MODULE NAMED '_SSL'
- CMD查看局域网所有IP信息 ARP -A(ARP命令介绍)
- 微服务DOCKER容器化自动部署

相关标签

- KAFKA
- 大数据
- 学习笔记
- 一文弄懂系列
- FLINK
- 区块链
- EPOLL
- JVM
- MVP
- KAFKA

sendOffsetsToTransaction() 方法。该模式的具体结构如下图所示。与此对应的应用示例如代码清单14-2所示。

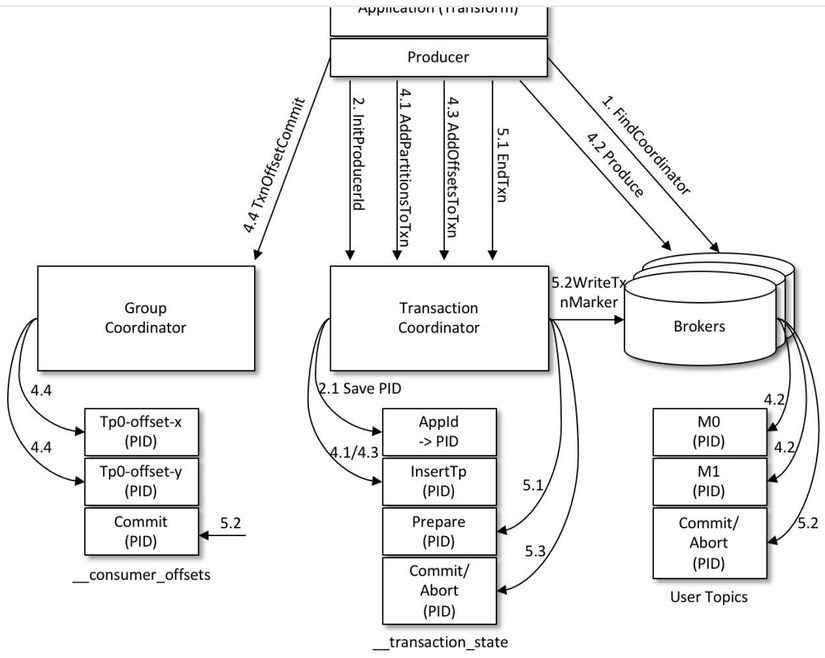


Consume – Transform - Producer

```java
代码清单14-2 消费–转换–生产模式示例
public class TransactionConsumeTransformProduce {
    public static final String brokerList = "localhost:9092";

    public static Properties getConsumerProperties(){
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, brokerList);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class.getName());
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "groupId");
        return props;
    }

    public static Properties getProducerProperties(){
        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());
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "transactionalId");
        return props;
    }

    public static void main(String[] args) {
        //初始化生产者和消费者
        KafkaConsumer<String, String> consumer =
                new KafkaConsumer<>(getConsumerProperties());
        consumer.subscribe(Collections.singletonList("topic-source"));
        KafkaProducer<String, String> producer =
                new KafkaProducer<>(getProducerProperties());
        //初始化事务
        producer.initTransactions();
        while (true) {
            ConsumerRecords<String, String> records =
                    consumer.poll(Duration.ofMillis(1000));
            if (!records.isEmpty()) {
                Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();
                //开启事务
                producer.beginTransaction();
                try {
                    for (TopicPartition partition : records.partitions()) {
                        List<ConsumerRecord<String, String>> partitionRecords
                                = records.records(partition);
                        for (ConsumerRecord<String, String> record :
                                partitionRecords) {
                            //do some logical processing.
                            ProducerRecord<String, String> producerRecord =
                                    new ProducerRecord<>("topic-sink", record.key(),
                                            record.value());
                            //消费–生产模型
                            producer.send(producerRecord);
                        }
                        long lastConsumedOffset = partitionRecords.
                                get(partitionRecords.size() - 1).offset();
                        offsets.put(partition,
                                new OffsetAndMetadata(lastConsumedOffset + 1));
                    }
                    //提交消费位移
                    producer.sendOffsetsToTransaction(offsets,"groupId");
                    //提交事务
                    producer.commitTransaction();
                } catch (ProducerFencedException e) {
                    //log the exception
                    //中止事务
                    producer.abortTransaction();
                }
            }
        }
    }
}
```

注意：在使用 KafkaConsumer 的时候要将 enable.auto.commit 参数设置为 false，代码里也不能手动提交消费位移。

为了实现事务的功能，Kafka 还引入了事务协调器（TransactionCoordinator）来负责处理事务，这一点可以类比一下组协调器（GroupCoordinator）。每一个生产者都会被指派一个特定的 TransactionCoordinator，所有的事务逻辑包括分派 PID 等都是由 TransactionCoordinator 来负责实施的。TransactionCoordinator 会将事务状态持久化到内部主题 __transaction_state 中。下面就以最复杂的 consume-transform-produce 的流程（参考下图，后面就以“事务流程图”称呼）为例来分析 Kafka 事务的实现原理。



1. 查找TRANSACTIONCOORDINATOR

TransactionCoordinator 负责分配 PID 和管理事务，因此生产者要做的第一件事情就是找出对应的 TransactionCoordinator 所在的 broker 节点。与查找 GroupCoordinator 节点一样，也是通过 FindCoordinatorRequest 请求来实现的，只不过 FindCoordinatorRequest 中的 coordinator_type 就由原来的0变成了1，由此来表示与事务相关联。

Kafka 在收到 FindCoorinatorRequest 请求之后，会根据 coordinator_key（也就是 transactionalId）查找对应的 TransactionCoordinator 节点。如果找到，则会返回其相对应的 node_id、host 和 port 信息。具体查找 TransactionCoordinator 的方式是根据 transactionalId 的哈希值计算主题 __transaction_state 中的分区编号，具体算法如代码清单14-3所示。

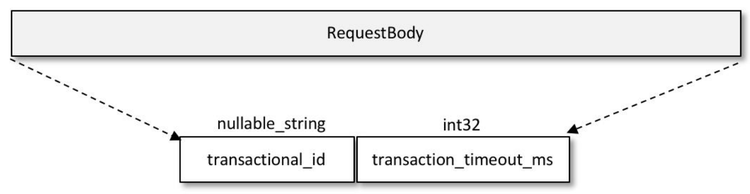
```
代码清单14-3 计算分区编号
Utils.abs(transactionalId.hashCode) % transactionTopicPartitionCount
```

其中 transactionTopicPartitionCount 为主题 __transaction_state 中的分区个数，这个可以通过 broker 端参数 transaction.state.log.num.partitions 来配置，默认值为50。

找到对应的分区之后，再寻找此分区 leader 副本所在的 broker 节点，该 broker 节点即为这个 transactionalId 对应的 TransactionCoordinator 节点。细心的读者可以发现，这一整套的逻辑和查找 GroupCoordinator 的逻辑如出一辙。

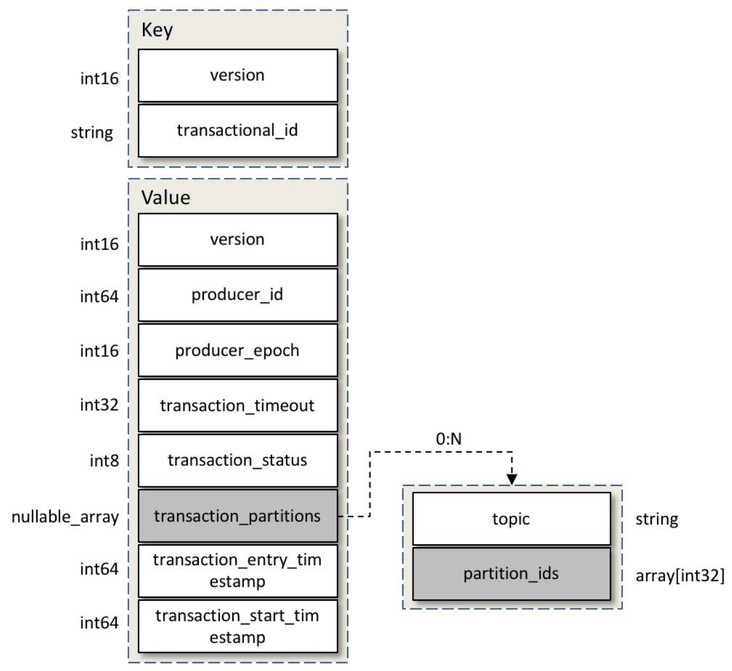
2. 获取PID

在找到 TransactionCoordinator 节点之后，就需要为当前生产者分配一个 PID 了。凡是开启了幂等性功能的生产者都必须执行这个操作，不需要考虑该生产者是否还开启了事务。生产者获取 PID 的操作是通过 InitProducerIdRequest 请求来实现的，InitProducerIdRequest 请求体结构如下图所示，其中 transactional_id 表示事务的 transactionalId，transaction_timeout_ms 表示 TransactionCoordinaor 等待事务状态更新的超时时间，通过生产者客户端参数 transaction.timeout.ms 配置，默认值为60000。

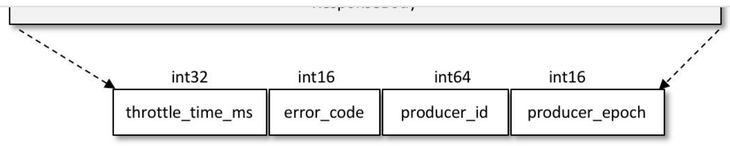


保存PID

生产者的 InitProducerIdRequest 请求会被发送给 TransactionCoordinator。注意，如果未开启事务特性而只开启幂等特性，那么 InitProducerIdRequest 请求可以发送给任意的 broker。当 TransactionCoordinator 第一次收到包含该 transactionalId 的 InitProducerIdRequest 请求时，它会把 transactionalId 和对应的 PID 以消息（我们习惯性地把这类消息称为“事务日志消息”）的形式保存到主题 __transaction_state 中，如事务流程图步骤2.1所示。这样可以保证 <transaction_Id, PID> 的对应关系被持久化，从而保证即使 TransactionCoordinator 宕机该对应关系也不会丢失。存储到主题 __transaction_state 中的具体内容格式如下图所示。



其中 transaction_status 包含 Empty(0)、Ongoing(1)、PrepareCommit(2)、PrepareAbort(3)、CompleteCommit(4)、CompleteAbort(5)、Dead(6) 这几种状态。在存入主题 __transaction_state 之前，事务日志消息同样会根据单独的 transactionalId 来计算要发送的分区，算法同代码清单14-3一样。



与 InitProducerIdRequest 对应的 InitProducerIdResponse 响应体结构如上图所示，除了返回 PID，InitProducerIdRequest 还会触发执行以下任务：

- 增加该 PID 对应的 producer_epoch。具有相同 PID 但 producer_epoch 小于该 producer_epoch 的其他生产者新开启的事务将被拒绝。
- 恢复（Commit）或中止（Abort）之前的生产者未完成的事务。

3. 开启事务

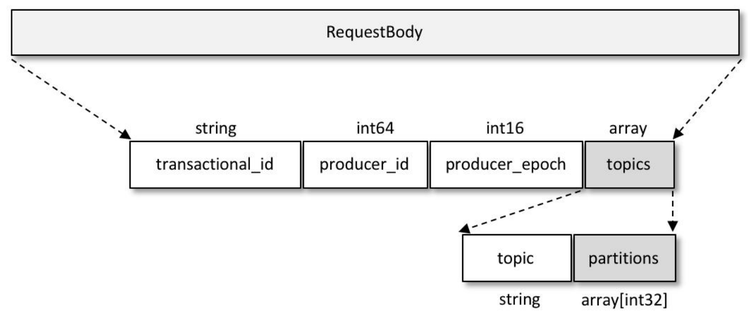
通过 KafkaProducer 的 beginTransaction() 方法可以开启一个事务，调用该方法后，生产者本地会标记已经开启了一个新的事务，只有在生产者发送第一条消息之后 TransactionCoordinator 才会认为该事务已经开启。

4. CONSUME-TRANSFORM-PRODUCE

这个阶段囊括了整个事务的数据处理过程，其中还涉及多种请求。注：如果没有给出具体的请求体或响应体结构，则说明其并不影响读者对内容的理解，笔者为了缩减篇幅而将其省略。

1）AddPartitionsToTxnRequest

当生产者给一个新的分区（TopicPartition）发送数据前，它需要先向 TransactionCoordinator 发送 AddPartitionsToTxnRequest 请求（AddPartitionsToTxnRequest 请求体结构如下图所示），这个请求会让 TransactionCoordinator 将 <transactionId, TopicPartition> 的对应关系存储在主题 __transaction_state 中，如图事务流程图步骤4.1所示。有了这个对照关系之后，我们就可以在后续的步骤中为每个分区设置 COMMIT 或 ABORT 标记，如事务流程图步骤5.2所示。



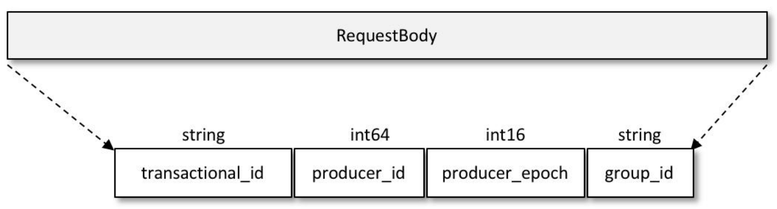
如果该分区是对应事务中的第一个分区，那么此时TransactionCoordinator还会启动对该事务的计时。

2）ProduceRequest

这一步骤很容易理解，生产者通过 ProduceRequest 请求发送消息（ProducerBatch）到用户自定义主题中，这一点和发送普通消息时相同，如事务流程图步骤4.2所示。和普通的消息不同的是，ProducerBatch 中会包含实质的 PID、producer_epoch 和 sequence number。

3）AddOffsetsToTxnRequest

通过 KafkaProducer 的 sendOffsetsToTransaction() 方法可以在一个事务批次里处理消息的消费和发送，方法中包含2个参数：Map<TopicPartition, OffsetAndMetadata> offsets 和 groupId。这个方法会向 TransactionCoordinator 节点发送 AddOffsetsToTxnRequest 请求（AddOffsetsToTxnRequest 请求体结构如下图所示），TransactionCoordinator 收到这个请求之后会通过 groupId 来推导出在 __consumer_offsets 中的分区，之后 TransactionCoordinator 会将这个分区保存在 __transaction_state 中，如事务流程图步骤4.3所示。



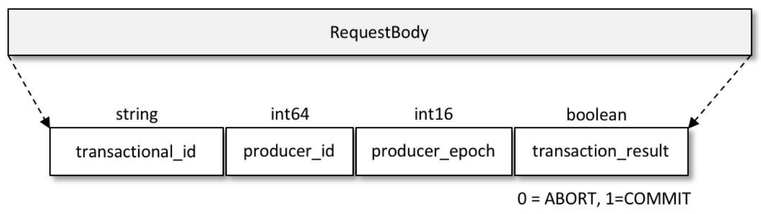
4）TxnOffsetCommitRequest

这个请求也是 sendOffsetsToTransaction() 方法中的一部分，在处理完 AddOffsetsToTxnRequest 之后，生产者还会发送 TxnOffsetCommitRequest 请求给 GroupCoordinator，从而将本次事务中包含的消费位移信息 offsets 存储到主题 __consumer_offsets 中，如事务流程图步骤4.4所示。

5. 提交或者中止事务

一旦数据被写入成功，我们就可以调用 KafkaProducer 的 commitTransaction() 方法或 abortTransaction() 方法来结束当前的事务。

1）EndTxnRequest 无论调用 commitTransaction() 方法还是 abortTransaction() 方法，生产者都会向 TransactionCoordinator 发送 EndTxnRequest 请求（对应的 EndTxnRequest 请求体结构如下图所示），以此来通知它提交（Commit）事务还是中止（Abort）事务。

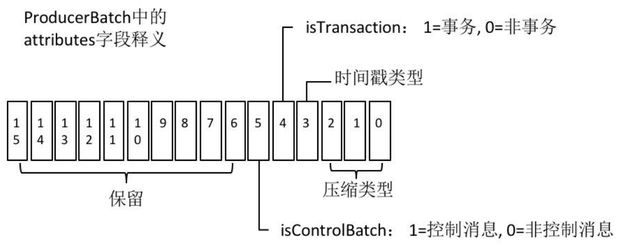


TransactionCoordinator 在收到 EndTxnRequest 请求后会执行如下操作：

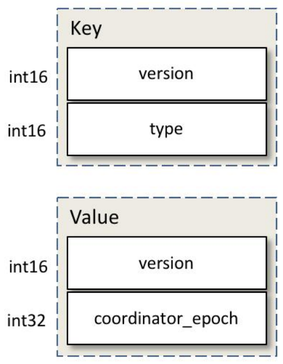
1. 将 PREPARE_COMMIT 或 PREPARE_ABORT 消息写入主题 __transaction_state，如事务流程图步骤5.1所示。
2. 通过 WriteTxnMarkersRequest 请求将 COMMIT 或 ABORT 信息写入用户所使用的普通主题和 __consumer_offsets，如事务流程图步骤5.2所示。

2）WriteTxnMarkersRequest

WriteTxnMarkersRequest 请求是由 TransactionCoordinator 发向事务中各个分区的 leader 节点的，当节点收到这个请求之后，会在相应的分区中写入控制消息（ControlBatch）。控制消息用来标识事务的终结，它和普通的消息一样存储在日志文件中，前面章节中提及了控制消息，RecordBatch 中 attributes 字段的第6位用来标识当前消息是否是控制消息。如果是控制消息，那么这一位会置为1，否则会置为0，如下图所示。



attributes 字段中的第5位用来标识当前消息是否处于事务中，如果是事务中的消息，那么这一位置为1，否则置为0。由于控制消息也处于事务中，所以attributes字段的第5位和第6位都被置为1。ControlBatch 中只有一个 Record，Record 中的 timestamp delta 字段和 offset delta 字段的值都为0，而控制消息的 key 和 value 的内容如下图所示。



就目前的 Kafka 版本而言，key 和 value 内部的 version 值都为0，key 中的 type 表示控制类型：0表示 ABORT，1表示 COMMIT；value 中的 coordinator_epoch 表示 TransactionCoordinator 的纪元（版本），TransactionCoordinator 切换的时候会更新其值。

3）写入最终的COMPLETE_COMMIT或COMPLETE_ABORT

TransactionCoordinator 将最终的 COMPLETE_COMMIT 或 COMPLETE_ABORT 信息写入主题 __transaction_state 以表明当前事务已经结束，此时可以删除主题 __transaction_state 中所有关于该事务的消息。由于主题 __transaction_state 采用的日志清理策略为日志压缩，所以这里的删除只需将相应的消息设置为墓碑消息即可。

哈希数据结构和代码实现

主要结构体： 实现插入、删除、查找、扩容、冲突解决等接口，用于理解哈希这种数据结构 完整代码参见github：https://github.com/jinxiang1224/cpp/tree/master/DataStruct_Algorithm/hash...

解决UBUNTU中解压ZIP文件（提取到此处）中文乱码问题

在Ubuntu系统下，解压zip文件时，使用右键--提取到此处，得到的文件内部文件名中文出现乱码。 导致此问题出现的原因一般为未下载相应的字体。 解决方案： 在终端中使用unar命令。 需要注意的是系统需要包含unar命令，如果没有，采用如下的方式解决： 实例效果展示： 直接提取到此处： 使用 unar filename.zip得到的文件...

CENTOS7安装MYSQL8.0.20单机版详细教程

mysql8.0之后与5.7存在着很大的差异，这些差异不仅仅表现在功能和性能上，还表现在基础操作和设置上。这给一些熟悉mysql5.7的小伙伴带来了很多困扰，下面我们就来详细介绍下8.0的安装和配置过程。 mysql在linux上的多种安装方式： 1.yum安装 由于centos默认的yum源中没有mysql，所以我们要使用yum安装mysql就必须自己指定mysql的yum源。在官网下载mysq...