Kafka设计解析（八）- Exactly Once语义与事务机制原理

1 写在前面的话

本文所有Kafka原理性的描述除特殊说明外均基于Kafka 1.0.0版本。

2 为什么要提供事务机制

Kafka事务机制的实现主要是为了支持

- `Exactly Once` 即正好一次语义
- 操作的原子性
- 有状态操作的可恢复性

2.1 `Exactly Once`

《Kafka背景及架构介绍》一文中有说明Kafka在0.11.0.0之前的版本中只支持 `At Least Once` 和 `At Most Once` 语义，尚不支持 `Exactly Once` 语义。

但是在很多要求严格的场景下，如使用Kafka处理交易数据， `Exactly Once` 语义是必须的。我们可以通过让下游系统具有幂等性来配合Kafka的 `At Least Once` 语义来间接实现 `Exactly Once` 。但是：

- 该方案要求下游系统支持幂等操作，限制了Kafka的适用场景
- 实现门槛相对较高，需要用户对Kafka的工作机制非常了解
- 对于Kafka Stream而言，Kafka本身即是自己的下游系统，但Kafka在0.11.0.0版本之前不具有幂等发送能力

因此，Kafka本身对 `Exactly Once` 语义的支持就非常必要。

2.2 操作原子性

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。

实现原子性操作的意义在于：

- 操作结果更可控，有助于提升数据一致性
- 便于故障恢复。因为操作是原子的，从故障中恢复时只需要重试该操作（如果原操作失败）或者直接跳过该操作（如果原操作成功），而不需要记录中间状态，更不需要针对中间状态作特殊处理

3 实现事务机制的几个阶段

3.1 幂等性发送

上文提到，实现 `Exactly Once` 的一种方法是让下游系统具有幂等处理特性，而在Kafka Stream中，Kafka Producer本身就是“下游”系统，因此如果能让Producer具有幂等处理特性，那就可以让Kafka Stream在一定程度上支持 `Exactly once` 语义。

为了实现Producer的幂等语义，Kafka引入了 `Producer ID` （即 `PID` ）和 `Sequence Number` 。每个新的Producer在初始化的时候会被分配一个唯一的PID，该PID对用户完全透明而不会暴露给用户。

对于每个PID，该Producer发送数据的每个 `<Topic, Partition>` 都对应一个从0开始单调递增的 `Sequence Number` 。

类似地，Broker端也会为每个 `<PID, Topic, Partition>` 维护一个序号，并且每次Commit一条消息时将其对应序号递增。对于接收的每条消息，如果其序号比Broker维护的序号（即最后一次Commit的消息的序号）大一，则Broker会接受它，否则将其丢弃：

- 如果消息序号比Broker维护的序号大一以上，说明中间有数据尚未写入，也即乱序，此时Broker拒绝该消息，Producer抛出 `InvalidSequenceNumber`
- 如果消息序号小于等于Broker维护的序号，说明该消息已被保存，即为重复消息，Broker直接丢弃该消息，Producer抛出 `DuplicateSequenceNumber`

上述设计解决了0.11.0.0之前版本中的两个问题：

- Broker保存消息后，发送ACK前宕机，Producer认为消息未发送成功并重试，造成数据重复
- 前一条消息发送失败，后一条消息发送成功，前一条消息重试后成功，造成数据乱序

3.2 事务性保证

上述幂等设计只能保证单个Producer对于同一个 `<Topic, Partition>` 的 `Exactly Once` 语义。

另外，它并不能保证写操作的原子性——即多个写操作，要么全部被Commit要么全部不被Commit。

更不能保证多个读写操作的的原子性。尤其对于Kafka Stream应用而言，典型的操作即是从某个Topic消费数据，经过一系列转换后写回另一个Topic，保证从源Topic的读取与向目标Topic的写入的原子性有助于从故障中恢复。

事务保证可使得应用程序将生产数据和消费数据当作一个原子单元来处理，要么全部成功，要么全部失败，即使该生产或消费跨多个 `<Topic, Partition>` 。

另外，有状态的应用也可以保证重启后从断点处继续处理，也即事务恢复。

为了实现这种效果，应用程序必须提供一个稳定的（重启后不变）唯一的ID，也即 `Transaction ID` 。 `Transactin ID` 与 `PID` 可能一一对应。区别在于 `Transaction ID` 由用户提供，而 `PID` 是内部的实现对用户透明。

另外，为了保证新的Producer启动后，旧的具有相同 `Transaction ID` 的Producer即失效，每次Producer通过 `Transaction ID` 拿到PID的同时，还会获取一个单调递增的epoch。由于旧的Producer的epoch比新Producer的epoch小，Kafka可以很容易识别出该Producer是老的Producer并拒绝其请求。

有了 `Transaction ID` 后，Kafka可保证：

- 跨Session的数据幂等发送。当具有相同 `Transaction ID` 的新的Producer实例被创建且工作时，旧的且拥有相同 `Transaction ID` 的Producer将不再工作。
- 跨Session的事务恢复。如果某个应用实例宕机，新的实例可以保证任何未完成的旧的事务要么Commit要么Abort，使得新实例从一个正常状态开始工作。

需要注意的是，上述的事务保证是从Producer的角度去考虑的。从Consumer的角度来看，该保证会相对弱一些。尤其是不能保证所有被某事务Commit过的所有消息都被一起消费，因为：

- 对于压缩的Topic而言，同一事务的某些消息可能被其它版本覆盖
- 事务包含的消息可能分布在多个Segment中（即使在同一个Partition内），当老的Segment被删除时，该事务的部分数据可能会丢失
- Consumer在一个事务内可能通过seek方法访问任意Offset的消息，从而可能丢失部分消息
- Consumer可能并不需要消费某一事务内的所有Partition，因此它将永远不会读取组成该事务的所有消息

4 事务机制原理

4.1 事务性消息传递

这一节所说的事务主要指原子性，也即Producer将多条消息作为一个事务批量发送，要么全部成功要么全部失败。

为了实现这一点，Kafka 0.11.0.0引入了一个服务器端的模块，名为 `Transaction Coordinator` ，用于管理Producer发送的消息的事务性。

该 `Transaction Coordinator` 维护 `Transaction Log` ，该log存于一个内部的Topic内。由于Topic数据具有持久性，因此事务的状态也具有持久性。

Producer并不直接读写 `Transaction Log` ，它与 `Transaction Coordinator` 通信，然后由 `Transaction Coordinator` 将该事务的状态插入相应的 `Transaction Log` 。

`Transaction Log` 的设计与 `Offset Log` 用于保存Consumer的Offset类似。

4.2 事务中Offset的提交

许多基于Kafka的应用，尤其是Kafka Stream应用中同时包含Consumer和Producer，前者负责从Kafka中获取消息，后者负责将处理完的数据写回Kafka的其它Topic中。

为了实现该场景下的事务的原子性，Kafka需要保证对Consumer Offset的Commit与Producer对发送消息的Commit包含在同一个事务中。否则，如果在二者Commit中间发生异常，根据二者Commit的顺序可能会造成数据丢失和数据重复：

- 如果先Commit Producer发送数据的事务再Commit Consumer的Offset，即 `At Least Once` 语义，可能造成数据重复。
- 如果先Commit Consumer的Offset，再Commit Producer数据发送事务，即 `At Most Once` 语义，可能造成数据丢失。

4.3 用于事务特性的控制型消息

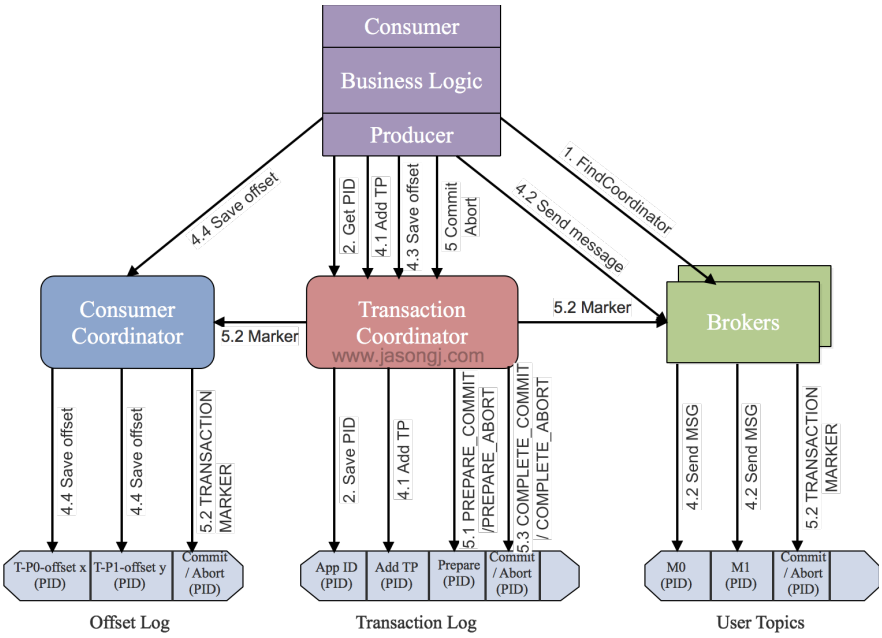
为了区分写入Partition的消息被Commit还是Abort，Kafka引入了一种特殊类型的消息，即 `Control Message` 。该类消息的Value内不包含任何应用相关的数据，并且不会暴露给应用程序。它只用于Broker与Client间的内部通信。

对于Producer端事务，Kafka以Control Message的形式引入一系列的 `Transaction Marker` 。Consumer即可通过该标记判定对应的消息被Commit了还是Abort了，然后结合该Consumer配置的隔离级别决定是否应该将该消息返回给应用程序。

4.4 事务处理样例代码

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

// 初始化事务，包括结束该Transaction ID对应的未完成的事务（如果有）
// 保证新的事务在一个正确的状态下启动
producer.initTransactions();

// 开始事务
producer.beginTransaction();

// 消费数据
ConsumerRecords<String, String> records = consumer.poll(100);

try{
    // 发送数据
    producer.send(new ProducerRecord<String, String>("Topic", "Key", "Value"));

    // 发送消费数据的Offset，将上述数据消费与数据发送纳入同一个Transaction内
    producer.sendOffsetsToTransaction(offsets, "group1");

    // 数据发送及Offset发送均成功的情况下，提交事务
    producer.commitTransaction();
} catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException
e) {
    // 数据发送或者Offset发送出现异常时，终止事务
    producer.abortTransaction();
} finally {
    // 关闭Producer和Consumer
    producer.close();
    consumer.close();
}
```

4.5 完整事务过程



4.5.1 找到 `Transaction Coordinator`

由于 `Transaction Coordinator` 是分配PID和管理事务的核心，因此Producer要做的第一件事情就是通过向任意一个Broker发送 `FindCoordinator` 请求找到 `Transaction Coordinator` 的位置。

注意：只有应用程序为Producer配置了 `Transaction ID` 时才可使用事务特性，也才需要这一步。另外，由于事务性要求Producer开启幂等特性，因此通过将 `transactional.id` 设置为非空从而开启事务特性的同时也需要通过将 `enable.idempotence` 设置为true来开启幂等特性。

4.5.2 获取PID

找到 `Transaction Coordinator` 后，具有幂等特性的Producer必须发起 `InitPidRequest` 请求以获取PID。

注意：只要开启了幂等特性即必须执行该操作，而无须考虑该Producer是否开启了事务特性。

*** 如果事务特性被开启 ***
`InitPidRequest` 会发送给 `Transaction Coordinator` 。如果 `Transaction Coordinator` 是第一次收到包含有该 `Transaction ID` 的InitPidRequest请求，它将会把该 `<TransactionID, PID>` 存入 `Transaction Log` ，如上图中步骤2.1所示。这样可保证该对应关系被持久化，从而保证即使 `Transaction Coordinator` 宕机该对应关系也不会丢失。

除了返回PID外， `InitPidRequest` 还会执行如下任务：

- 增加该PID对应的epoch。具有相同PID但epoch小于该epoch的其它Producer（如果有）新开启的事务将被拒绝。
- 恢复（Commit或Abort）之前的Producer未完成的事务（如果有）。

注意： `InitPidRequest` 的处理过程是同步阻塞的。一旦该调用正确返回，Producer即可开始新的事务。

另外，如果事务特性未开启， `InitPidRequest` 可发送至任意Broker，并且会得到一个全新的唯一的PID。该Producer将只能使用幂等特性以及单一Session内的事务特性，而不能使用跨Session的事务特性。

4.5.3 开启事务

Kafka从0.11.0.0版本开始，提供 `beginTransaction()` 方法用于开启一个事务。调用该方法后，Producer本地会记录已经开启了事务，但 `Transaction Coordinator` 只有在Producer发送第一条消息后才认为事务已经开启。

4.5.4 Consume-Transform-Produce

这一阶段，包含了整个事务的数据处理过程，并且包含了多种请求。

AddPartitionsToTxnRequest

一个Producer可能会给多个 `<Topic, Partition>` 发送数据，给一个新的 `<Topic, Partition>` 发送数据前，它需要先向 `Transaction Coordinator` 发送 `AddPartitionsToTxnRequest` 。

`Transaction Coordinator` 会将该 `<Transaction, Topic, Partition>` 存于 `Transaction Log` 内，并将其状态置为 `BEGIN` ，如上图中步骤4.1所示。有了该信息后，我们才可以在后续步骤中为每个 `Topic, Partition>` 设置COMMIT或者ABORT标记（如上图中步骤5.2所示）。

另外，如果该 `<Topic, Partition>` 为该事务中第一个 `<Topic, Partition>` ，`Transaction Coordinator` 还会启动对该事务的计时（每个事务都有自己的超时时间）。

ProduceRequest

Producer通过一个或多个 `ProduceRequest` 发送一系列消息。除了应用数据外，该请求还包含了PID，epoch，和 `Sequence Number` 。该过程如上图中步骤4.2所示。

AddOffsetsToTxnRequest

为了提供事务性，Producer新增了 `sendOffsetsToTransaction` 方法，该方法将多组消息的发送和消费放入同一批处理内。

该方法先判断在当前事务中该方法是否已经被调用并传入了相同的Group ID。若是，直接跳到下一步；若不是，则向 `Transaction Coordinator` 发送 `AddOffsetsToTxnRequests` 请求，`Transaction Coordinator` 将对应的所有 `<Topic, Partition>` 存于 `Transaction Log` 中，并将其状态记为 `BEGIN` ，如上图中步骤4.3所示。该方法会阻塞直到收到响应。

TxnOffsetCommitRequest

作为 `sendOffsetsToTransaction` 方法的一部分，在处理完 `AddOffsetsToTxnRequest` 后，Producer也会发送 `TxnOffsetCommit` 请求给 `Consumer Coordinator` 从而将本事务包含的与读操作相关的各 `<Topic, Partition>` 的Offset持久化到内部的 `__consumer_offsets` 中，如上图步骤4.4所示。

在此过程中， `Consumer Coordinator` 会通过PID和对应的epoch来验证是否应该允许该Producer的该请求。

这里需要注意：

- 写入 `__consumer_offsets` 的Offset信息在当前事务Commit前对外是不可见的。也即在当前事务被Commit前，可认为该Offset尚未Commit，也即对应的消息尚未被完成处理。
- `Consumer Coordinator` 并不会立即更新缓存中相应 `<Topic, Partition>` 的Offset，因为此时这些更新操作尚未被COMMIT或ABORT。

4.5.5 Commit或Abort事务

一旦上述数据写入操作完成，应用程序必须调用 `KafkaProducer` 的 `commitTransaction` 方法或者 `abortTransaction` 方法以结束当前事务。

EndTxnRequest

`commitTransaction` 方法使得Producer写入的数据对下游Consumer可见。 `abortTransaction` 方法通过 `Transaction Marker` 将Producer写入的数据标记为 `Aborted` 状态。下游的Consumer如果将 `isolation.level` 设置为 `READ_COMMITTED` ，则它读到被Abort的消息后直接将其丢弃而不会返回给客户程序，也即被Abort的消息对应用程序不可见。

无论是Commit还是Abort，Producer都会发送 `EndTxnRequest` 请求给 `Transaction Coordinator` ，并通过标志位标识是应该Commit还是Abort。

收到该请求后， `Transaction Coordinator` 会进行如下操作

1. 将 `PREPARE_COMMIT` 或 `PREPARE_ABORT` 消息写入 `Transaction Log` ，如上图中步骤5.1所示
2. 通过 `WriteTxnMarker` 请求以 `Transaction Marker` 的形式将 `COMMIT` 或 `ABORT` 信息写入用户数据日志以及 `Offset Log` 中，如上图中步骤5.2所示
3. 最后将 `COMPLETE_COMMIT` 或 `COMPLETE_ABORT` 信息写入 `Transaction Log` 中，如上图中步骤5.3所示

补充说明：对于 `commitTransaction` 方法，它会在发送 `EndTxnRequest` 之前先调用flush方法以确保所有发送出去的数据都得到相应的ACK。对于 `abortTransaction` 方法，在发送 `EndTxnRequest` 之前直接将当前Buffer中的事务性消息（如果有）全部丢弃，但必须等待所有被发送但尚未收到ACK的消息发送完成。

上述第二步是实现将一组读操作与写操作作为一个事务处理的关键。因为Producer写入的数据Topic以及记录Comsumer Offset的Topic会被写入相同的 `Transactin Marker` ，所以这一组读操作与写操作要么全部COMMIT要么全部ABORT。

WriteTxnMarkerRequest

上面提到的 `WriteTxnMarkerRequest` 由 `Transaction Coordinator` 发送给当前事务涉及到的每个 `<Topic, Partition>` 的Leader。收到该请求后，对应的Leader会将对应的 `COMMIT(PID)` 或者 `ABORT(PID)` 控制信息写入日志，如上图中步骤5.2所示。

该控制消息向Broker以及Consumer表明对应PID的消息被Commit了还是被Abort了。

这里要注意，如果事务也涉及到 `__consumer_offsets` ，即该事务中有消费数据的操作且将该消费的Offset存于 `__consumer_offsets` 中， `Transaction Coordinator` 也需要向该内部Topic的各Partition的Leader发送 `WriteTxnMarkerRequest` 从而写入 `COMMIT(PID)` 或 `COMMIT(PID)` 控制信息。

写入最终的 `COMPLETE_COMMIT` 或 `COMPLETE_ABORT` 消息

写完所有的 `Transaction Marker` 后， `Transaction Coordinator` 会将最终的 `COMPLETE_COMMIT` 或 `COMPLETE_ABORT` 消息写入 `Transaction Log` 中以标明该事务结束，如上图中步骤5.3所示。

此时， `Transaction Log` 中所有关于该事务的消息全部可以移除。当然，由于Kafka内数据是Append Only的，不可直接更新和删除，这里说的移除只是将其标记为null从而在Log Compact时不再保留。

另外， `COMPLETE_COMMIT` 或 `COMPLETE_ABORT` 的写入并不需要得到所有Rreplica的ACK，因为如果该消息丢失，可以根据事务协议重发。

补充说明，如果参与该事务的某些 `<Topic, Partition>` 在被写入 `Transaction Marker` 前不可用，它对 `READ_COMMITTED` 的Consumer不可见，但不影响其它可用 `<Topic, Partition>` 的COMMIT或ABORT。在该 `<Topic, Partition>` 恢复可用后， `Transaction Coordinator` 会重新根据 `PREPARE_COMMIT` 或 `PREPARE_ABORT` 向该 `<Topic, Partition>` 发送 `Transaction Marker` 。

4.6 总结

- `PID` 与 `Sequence Number` 的引入实现了写操作的幂等性
- 写操作的幂等性结合 `At Least Once` 语义实现了单一Session内的 `Exactly Once` 语义
- `Transaction Marker` 与 `PID` 提供了识别消息是否应该被读取的能力，从而实现了事务的隔离性
- Offset的更新标记了消息是否被读取，从而将对读操作的事务处理转换成了对写（Offset）操作的事务处理
- Kafka事务的本质是，将一组写操作（如果有）对应的消息与一组读操作（如果有）对应的Offset的更新进行同样的标记（即 `Transaction Marker` ）来实现事务中涉及的所有读写操作同时对外可见或同时对外不可见
- Kafka只提供对Kafka本身的读写操作的事务性，不提供包含外部系统的事务性

5 异常处理

5.1 Exception处理

InvalidProducerEpoch

这是一种Fatal Error，它说明当前Producer是一个过期的实例，有 `Transaction ID` 相同但epoch更新的Producer实例被创建并使用。此时Producer会停止并抛出Exception。

InvalidPidMapping

`Transaction Coordinator` 没有与该 `Transaction ID` 对应的PID。此时Producer会通过包含有 `Transaction ID` 的 `InitPidRequest` 请求创建一个新的PID。

NotCorrdinatorForGTransactionalId

该 `Transaction Coordinator` 不负责该当前事务。Producer会通过 `FindCoordinatorRequest` 请求重新寻找对应的 `Transaction Coordinator` 。

InvalidTxnRequest

违反了事务协议。正确的Client实现不应该出现这种Exception。如果该异常发生了，用户需要检查自己的客户端实现是否有问题。

CoordinatorNotAvailable

`Transaction Coordinator` 仍在初始化中。Producer只需要重试即可。

DuplicateSequenceNumber
发送的消息的序号低于Broker预期。该异常说明该消息已经被成功处理过，Producer可以直接忽略该异常并处理下一条消息

InvalidSequenceNumber
这是一个Fatal Error，它说明发送的消息中的序号大于Broker预期。此时有两种可能

- 数据乱序。比如前面的消息发送失败后重试期间，新的消息被接收。正常情况下不应该出现该问题，因为当幂等发送启用时，`max.inflight.requests.per.connection`被强制设置为1，而`acks`被强制设置为all。故前面消息重试期间，后续消息不会被发送，也即不会发生乱序。并且只有ISR中所有Replica都ACK，Producer才会认为消息已经被发送，也即不存在Broker端数据丢失问题。
- 服务器由于日志被Truncate而造成数据丢失。此时应该停止Producer并将此Fatal Error报告给用户。

InvalidTransactionTimeout
`InitPidRequest`调用出现的Fatal Error。它表明Producer传入的timeout时间不在可接受范围内，应该停止Producer并报告给用户。

5.2 处理`Transaction Coordinator`失败

5.2.1 写`PREPARE_COMMIT/PREPARE_ABORT`前失败

Producer通过`FindCoordinatorRequest`找到新的`Transaction Coordinator`，并通过`EndTxnRequest`请求发起`COMMIT`或`ABORT`流程，新的`Transaction Coordinator`继续处理`EndTxnRequest`请求一一写`PREPARE_COMMIT`或`PREPARE_ABORT`，写`Transaction Marker`，写`COMPLETE_COMMIT`或`COMPLETE_ABORT`。

5.2.2 写完`PREPARE_COMMIT/PREPARE_ABORT`后失败

此时旧的`Transaction Coordinator`可能已经成功写入部分`Transaction Marker`。新的`Transaction Coordinator`会重复这些操作，所以部分Partition中可能会存在重复的`COMMIT`或`ABORT`，但只要该Producer在此期间没有发起新的事务，这些重复的`Transaction Marker`就不是问题。

5.2.3 写完`COMPLETE_COMMIT/ABORT`后失败

旧的`Transaction Coordinator`可能已经写完了`COMPLETE_COMMIT`或`COMPLETE_ABORT`但在返回`EndTxnRequest`之前失败。该场景下，新的`Transaction Coordinator`会直接给Producer返回成功。

5.3 事务过期机制

5.3.1 事务超时

`transaction.timeout.ms`

5.3.2 终止过期事务

当Producer失败时，`Transaction Coordinator`必须能够主动的让某些进行中的事务过期。否则没有Producer的参与，`Transaction Coordinator`无法判断这些事务应该如何处理，这会造成：

- 如果这种进行中事务太多，会造成`Transaction Coordinator`需要维护大量的事务状态，大量占用内存
- `Transaction Log`内也会存在大量数据，造成新的`Transaction Coordinator`启动缓慢
- `READ_COMMITTED`的Consumer需要缓存大量的消息，造成不必要的内存浪费甚至是OOM
- 如果多个`Transaction ID`不同的Producer交叉写同一个Partition，当一个Producer的事务状态不更新时，`READ_COMMITTED`的Consumer为了保证顺序消费而被阻塞

为了避免上述问题，`Transaction Coordinator`会周期性遍历内存中的事务状态Map，并执行如下操作

- 如果状态是`BEGIN`并且其最后更新时间与当前时间差大于`transaction.remove.expired.transaction.cleanup.interval.ms`（默认值为1小时），则主动将其终止：1）未避免原Producer临时恢复与当前终止流程冲突，增加该Producer对应的PID的epoch，并确保将该更新的信息写入`Transaction Log`；2）以更新后的epoch回滚事务，从而使得该事务相关的所有Broker都更新其缓存的该PID的epoch从而拒绝旧Producer的写操作
- 如果状态是`PREPARE_COMMIT`，完成后续的COMMIT流程一一一一向各`<Topic, Partition>`写入`Transaction Marker`，在`Transaction Log`内写入`COMPLETE_COMMIT`
- 如果状态是`PREPARE_ABORT`，完成后续ABORT流程

5.3 终止 `Transaction ID`

某 `Transaction ID` 的Producer可能很长时间不再发送数据， `Transaction Coordinator` 没必要再保存该 `Transaction ID` 与 `PID` 等的映射，否则可能会造成大量的资源浪费。因此需要有一个机制探测不再活跃的 `Transaction ID` 并将其信息删除。

`Transaction Coordinator` 会周期性遍历内存中的 `Transaction ID` 与 `PID` 映射，如果某 `Transaction ID` 没有对应的正在进行中的事务并且它对应的最后一个事务的结束时间与当前时间差大于 `transactional.id.expiration.ms` （默认值是7天），则将其从内存中删除并在 `Transaction Log` 中将其对应的日志的值设置为null从而使得Log Compact可将其记录删除。

6 与其它系统事务机制对比

6.1 PostgreSQL MVCC

Kafka的事务机制与《MVCC PostgreSQL实现事务和多版本并发控制的精华》一文中介绍的PostgreSQL通过MVCC实现事务的机制非常类似，对于事务的回滚，并不需要删除已写入的数据，都是将写入数据的事务标记为Rollback/Abort从而在读数据时过滤该数据。

6.2 两阶段提交

Kafka的事务机制与《分布式事务（一）两阶段提交及JTA》一文中所介绍的两阶段提交机制看似相似，都分PREPARE阶段和最终COMMIT阶段，但又有很大不同。

- Kafka事务机制中，PREPARE时即要指明是 `PREPARE_COMMIT` 还是 `PREPARE_ABORT` ，并且只须在 `Transaction Log` 中标记即可，无须其它组件参与。而两阶段提交的PREPARE需要发送给所有的分布式事务参与方，并且事务参与方需要尽可能准备好，并根据准备情况返回 `Prepared` 或 `Non-Prepared` 状态给事务管理器。
- Kafka事务中，一但发起 `PREPARE_COMMIT` 或 `PREPARE_ABORT` ，则确定该事务最终的结果应该是被 `COMMIT` 或 `ABORT` 。而分布式事务中，PREPARE后由各事务参与方返回状态，只有所有参与方均返回 `Prepared` 状态才会真正执行COMMIT，否则执行ROLLBACK
- Kafka事务机制中，某几个Partition在COMMIT或ABORT过程中变为不可用，只影响该Partition不影响其它Partition。两阶段提交中，若唯一收到COMMIT命令参与者Crash，其它事务参与方无法判断事务状态从而使得整个事务阻塞
- Kafka事务机制引入事务超时机制，有效避免了挂起的事务影响其它事务的问题
- Kafka事务机制中存在多个 `Transaction Coordinator` 实例，而分布式事务中只有一个事务管理器

6.3 Zookeeper

Zookeeper的原子广播协议与两阶段提交以及Kafka事务机制有相似之处，但又有各自的特点

- Kafka事务可COMMIT也可ABORT。而Zookeeper原子广播协议只有COMMIT没有ABORT。当然，Zookeeper不COMMIT某消息也即等效于ABORT该消息的更新。
- Kafka存在多个 `Transaction Coordinator` 实例，扩展性较好。而Zookeeper写操作只能在Leader节点进行，所以其写性能远低于读性能。
- Kafka事务是COMMIT还是ABORT完全取决于Producer即客户端。而Zookeeper原子广播协议中某条消息是否被COMMIT取决于是否有一大半FOLLOWER ACK该消息。

7 Kafka系列文章

- Kafka设计解析（一）- Kafka背景及架构介绍
- Kafka设计解析（二）- Kafka High Availability （上）
- Kafka设计解析（三）- Kafka High Availability （下）
- Kafka设计解析（四）- Kafka Consumer设计解析
- Kafka设计解析（五）- Kafka性能测试方法及Benchmark报告
- Kafka设计解析（六）- Kafka高性能架构之道
- Kafka设计解析（七）- Kafka Stream
- Kafka设计解析（八）- Kafka Exactly Once语义与事务机制原理

分类: Kafka,分布式,集群,消息系统

标签: Kafka, 事务, 大数据

郭俊Jason
关注 - 0
粉丝 - 125
+加关注

2 0

« 上一篇：流式处理的新贵 Kafka Stream - Kafka设计解析（七）
» 下一篇：Java进阶（七）正确理解Thread Local的原理与适用场景

posted on 2017-11-28 21:56 郭俊Jason 阅读(5006) 评论(0) 编辑 收藏

相关博文：
· Kafka设计解析（八）-ExactlyOnce语义与事务机制原理
· Kafka0.11.0.0实现producer的Exactly-once语义（英文）
· kafka 怎么保证的exactly once
· Kafka:Exactly-onceSemantics
· Exactly-Once投递语义

最新新闻：
· 苹果HomePod降价500元：支持24期免息分期
· 三星开始量产5G芯片：要做行业领导者
· 瑞士创企宣称开发出超级电池 电动汽车可续航上千公里
· 纽约时报：改变世界的乔布斯走了 库克忙着卖"糖水"
· 王小川连发三条微博：不姑息不认同搜狗价值观的人
» 更多新闻...