RocketMQ事务消息学习及刨坑过程 - 清茶豆奶 - 博客园

笔记本： A1-Tech
创建时间： 2020/4/15 11:10
标签： 消息队列
URL： https://www.cnblogs.com/huangying2124/p/11702761.html

RocketMQ事务消息学习及刨坑过程

一、背景

MQ组件是系统架构里必不可少的一门利器，设计层面可以降低系统耦合度，高并发场景又可以起到削峰填谷的作用，从单体应用到集群部署方案，再到现在的微服务架构，MQ凭借其优秀的性能和高可靠性，得到了广泛的认可。
随着数据量增多，系统压力变大，开始出现这种现象：数据库已经更新了，但消息没发出来，或者消息先发了，但后来数据库更新失败了，结果研发童鞋各种数据修复，这种生产问题出现的概率不大，但让人很郁闷。这个其实就是数据库事务与MQ消息的一致性问题，简单来讲，数据库的事务跟普通MQ消息发送无法直接绑定与数据库事务绑定在一起，例如上面提及的两种问题场景：

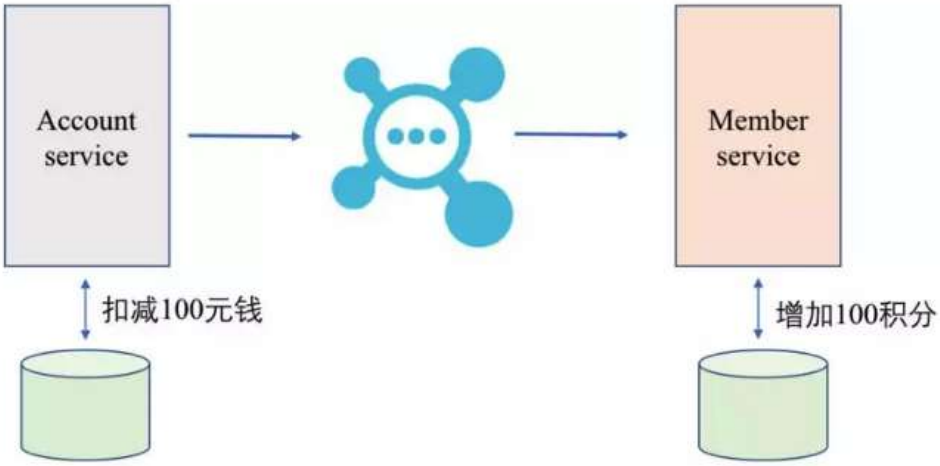
1. 数据库事务提交后发送MQ消息；
2. MQ消息先发，然后再提交数据库事务。

场景1的问题是数据库事务可能刚刚提交，服务器就宕机了，MQ消息没发出去，场景2的问题就是MQ消息发送出去了，但数据库事务提交失败，又没办法追加已经发出去的MQ消息，结果导致数据没更新，下游已经收到消息，最终事务出现不一致的情况。

二、事务消息的引出

我们以微服务架构的购物场景为例，参照一下RocketMQ官方的例子，用户A发起订单，支付100块钱操作完成后，能得到100积分，账户服务和会员服务是两个独立的微服务模块，有各自的数据库，按照上文提及的问题可能性，将会出现这些情况：

- 如果先扣款，再发消息，可能钱刚扣完，宕机了，消息没发出去，结果积分没增加。
- 如果先发消息，再扣款，可能积分增加了，但钱没扣掉，白送了100积分给人家。
- 钱正常扣了，消息也发送成功了，但会员服务实例消费消息出现问题，结果积分没增加。

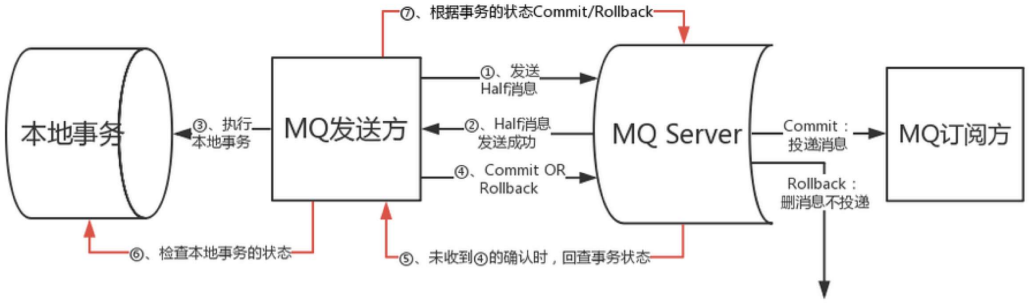


由此引出的是数据库事务与MQ消息的事务一致性问题，rocketmq事务消息解决的问题：解决本地事务执行与消息发送的原子性问题。这里界限一定要明白，是确保MQ生产端正确无误地将消息发送出来，没有多发，也不会漏发。但至于发送后消费端有没有正常的消费掉（如上面提及的第三种情况，钱正常扣了，消息也发了，但下游消费出问题导致积分不对），这种异常场景将由MQ消息消费失败重试机制来保证，不在此次的讨论范围内。

常用的MQ组件针对此场景都有自己的实现方案，如ActiveMQ使用AMQP协议(二阶提交方式)保证消息正确发送，这里我们以RocketMQ为重点进行学习。

三、RocketMQ事务消息设计思路

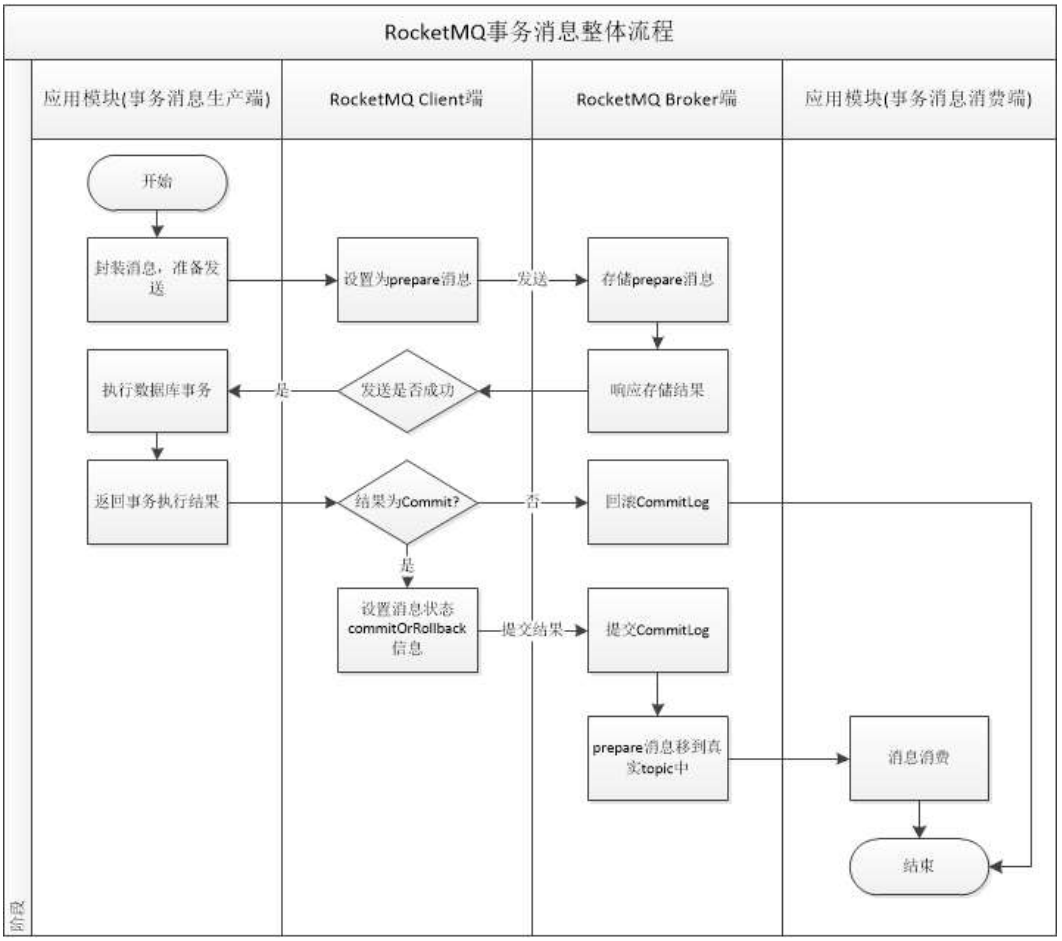
根据CAP理论，RocketMQ事务消息通过异步确保方式，保证事务的最终一致性。设计流程上借鉴两阶段提交理论，流程图如下：



1. 应用模块遇到要发送事务消息的场景时，先发送prepare消息给MQ。
2. prepare消息发送成功后，应用模块执行数据库事务（本地事务）。
3. 根据数据库事务执行的结果，再返回Commit或Rollback给MQ。
4. 如果是Commit，MQ把消息下发给Consumer端，如果是Rollback，直接删掉prepare消息。
5. 第3步的执行结果如果没响应，或是超时的，启动定时任务回查事务状态（最多重试15次，超过了默认丢弃此消息），处理结果同第4步。
6. MQ消费的成功机制由MQ自己保证。

四、RocketMQ事务消息实现流程

以RocketMQ 4.5.2版本为例，事务消息有专门的一个队列RMQ_SYS_TRANS_HALF_TOPIC，所有的prepare消息都先往这里放，当消息收到Commit请求后，就把消息再塞到真实的Topic队列里，供Consumer消费，同时向RMQ_SYS_TRANS_OP_HALF_TOPIC塞一条消息。简易流程图如下：

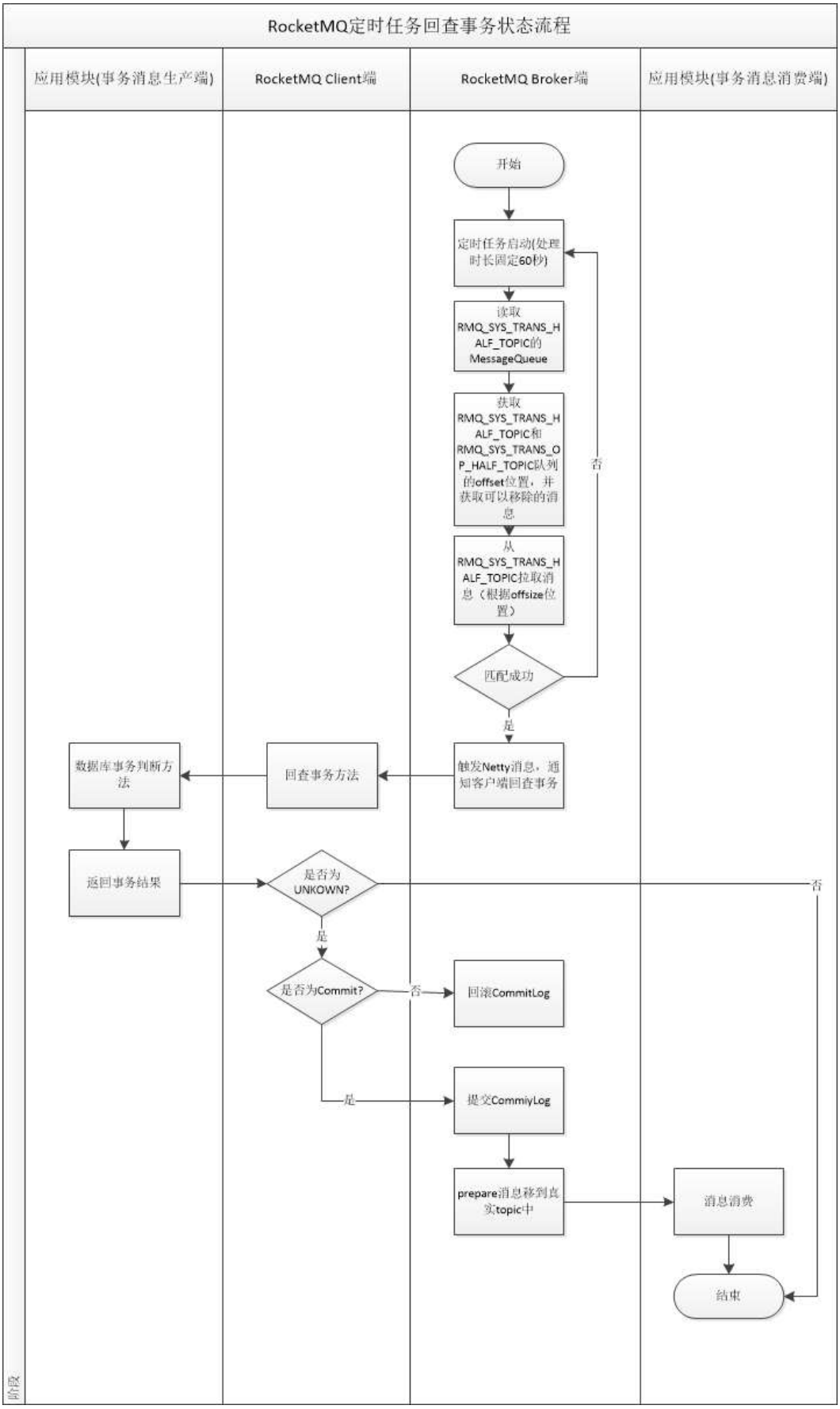


上述流程中，请允许我这样划分模块职责：

1. RocketMQ Client即我们工程中导入的依赖jar包，RocketMQ Broker端即部署的服务端，NameServer暂未体现。
2. 应用模块成对出现，上游为事务消息生产端，下游为事务消息消费端（事务消息对消费端是透明的，与普通消息一致）。

应用模块的事务因为中断，或是其他的网络原因，导致无法立即响应的，RocketMQ当做UNKNOW处理，RocketMQ事务消息还提供了一个补救方案：定时查询事务消息的数据库事务状态

简易流程图如下：



五、源码剖析

讲解的思路基本上按照如下流程图，根据模块职责和流程逐一分析。

1. 环境准备

   阅读源码前需要在IDE上获取和调试RocketMQ的源码，这部分请自行查阅方

法。

2. 应用模块（事务消息生产端）核心源码
   创建一个监听类，实现TransactionListener接口，在实现的数据库事务提交方法和回查事务状态方法模拟结果。

```
/**
 * @program: rocket
 * @description: 调试事务消息示例代码
 * @author: Huang
 * @create: 2019-10-16
 **/
public class SelfTransactionListener implements TransactionListener {
   private AtomicInteger transactionIndex = new AtomicInteger(0);
   private AtomicInteger checkTimes = new AtomicInteger(0);

   private ConcurrentHashMap<String, Integer> localTrans = new
ConcurrentHashMap<>();
   /**
    * 执行本地事务
    *
    * @param message
    * @param o
    * @return
    */
   @Override
   public LocalTransactionState executeLocalTransaction(Message
message, Object o) {
      String msgKey = message.getKeys();
      System.out.println("start execute local transaction " + msgKey);
      LocalTransactionState state;
      if (msgKey.contains("1")) {
         // 第一条消息让他通过
         state = LocalTransactionState.COMMIT_MESSAGE;
      } else if (msgKey.contains("2")) {
         // 第二条消息模拟异常，明确回复回滚操作
         state = LocalTransactionState.ROLLBACK_MESSAGE;
      } else {
         // 第三条消息无响应，让它调用回查事务方法
         state = LocalTransactionState.UNKNOW;
         // 给剩下3条消息，放1，2，3三种状态
         localTrans.put(msgKey, transactionIndex.incrementAndGet());
      }
      System.out.println("executeLocalTransaction:" + message.getKeys()
+ ",execute state:" + state + ",current time: " +
System.currentTimeMillis());
      return state;
   }

   /**
    * 回查本地事务结果
    *
    * @param messageExt
    * @return
    */
   @Override
```

```
   public LocalTransactionState checkLocalTransaction(MessageExt
messageExt) {
      String msgKey = messageExt.getKeys();
      System.out.println("start check local transaction " + msgKey);
      Integer state = localTrans.get(msgKey);
      switch (state) {
         case 1:
            System.out.println("check result unknown 回查次数" +
checkTimes.incrementAndGet());
            return LocalTransactionState.UNKNOW;
         case 2:
            System.out.println("check result commit message, 回查次数" +
checkTimes.incrementAndGet());
            return LocalTransactionState.COMMIT_MESSAGE;
         case 3:
            System.out.println("check result rollback message, 回查次数"
+ checkTimes.incrementAndGet());
            return LocalTransactionState.ROLLBACK_MESSAGE;

         default:
            return LocalTransactionState.COMMIT_MESSAGE;
      }
   }
}
```

事务消息生产者代码示例，共发送5条消息，基本上包含全部的场景，休眠时间设置足够的时间，保证回查事务时实例还在运行中，代码如下：

```
/**
 * @program: rocket
 * @description: Rocketmq事务消息
 * @author: Huang
 * @create: 2019-10-16
 **/
public class TransactionProducer {

   public static void main(String[] args) {
      try {
         TransactionMQProducer producer = new
TransactionMQProducer("transactionMQProducer");
         producer.setNamesrvAddr("10.0.133.29:9876");
         producer.setTransactionListener(new
SelfTransactionListener());
         producer.start();
         for (int i = 1; i < 6; i++) {
            Message message = new Message("TransactionTopic",
"transactionTest","msg-" + i, ("Hello" + ":" +  i).getBytes());
            try {
               SendResult result =
producer.sendMessageInTransaction(message, "Hello" + ":" +  i);
               System.out.printf("Topic:%s send success, misId
is:%s%n", message.getTopic(), result.getMsgId());
            } catch (Exception e) {
               e.printStackTrace();
            }
         }
         Thread.sleep(Integer.MAX_VALUE);
```

```
            producer.shutdown();
        } catch (MQClientException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3. RocketMQ Client端代码，代码主要逻辑可以分成三段：第一段为设置消息为prepare消息，并发送给RocketMQ服务端

```
SendResult sendResult = null;
MessageAccessor.putProperty(msg,
MessageConst.PROPERTY_TRANSACTION_PREPARED, "true");
MessageAccessor.putProperty(msg, MessageConst.PROPERTY_PRODUCER_GROUP,
this.defaultMQProducer.getProducerGroup());
try {
    sendResult = this.send(msg);
} catch (Exception e) {
    throw new MQClientException("send message Exception", e);
}
```

第二段：消息发送成功后，调用应用模块数据库事务方法，获取事务结果（为节省篇幅，代码有删节）

```
switch (sendResult.getSendStatus()) {
    case SEND_OK: {
        try {
            if (null != localTransactionExecuter) {
                localTransactionState =
localTransactionExecuter.executeLocalTransactionBranch(msg, arg);
            } else if (transactionListener != null) {
                log.debug("Used new transaction API");
                localTransactionState =
transactionListener.executeLocalTransaction(msg, arg);
            }
            if (null == localTransactionState) {
                localTransactionState = LocalTransactionState.UNKNOW;
            }
        } catch (Throwable e) {
            log.info("executeLocalTransactionBranch exception", e);
            log.info(msg.toString());
            localException = e;
        }
    }
    break;
    case FLUSH_DISK_TIMEOUT:
    case FLUSH_SLAVE_TIMEOUT:
    case SLAVE_NOT_AVAILABLE:
        localTransactionState = LocalTransactionState.ROLLBACK_MESSAGE;
        break;
    default:
        break;
}
```

第三段：发送事务结果到RocketMQ端，结束事务，并响应结果给应用模块

```
try {
    this.endTransaction(sendResult, localTransactionState,
localException);
} catch (Exception e) {
    log.warn("local transaction execute " + localTransactionState + ",
but end broker transaction failed", e);
}
```

4. RocketMQ Broker端事务提交/回滚操作（这里取endTransaction部分）

   代码入口：org.apache.rocketmq.broker.processor.EndTransactionProcessor

```
OperationResult result = new OperationResult();
if (MessageSysFlag.TRANSACTION_COMMIT_TYPE ==
requestHeader.getCommitOrRollback()) {
    result =
this.brokerController.getTransactionalMessageService().commitMessage(re
questHeader);
    if (result.getResponseCode() == ResponseCode.SUCCESS) {
        RemotingCommand res =
checkPrepareMessage(result.getPrepareMessage(), requestHeader);
        if (res.getCode() == ResponseCode.SUCCESS) {
            // 修改消息的Topic为由RMQ_SYS_TRANS_HALF_TOPIC改为真实Topic
            MessageExtBrokerInner msgInner =
endMessageTransaction(result.getPrepareMessage());

msgInner.setSysFlag(MessageSysFlag.resetTransactionValue(msgInner.getSy
sFlag(), requestHeader.getCommitOrRollback()));

msgInner.setQueueOffset(requestHeader.getTranStateTableOffset());

msgInner.setPreparedTransactionOffset(requestHeader.getCommitLogOffset(
));

msgInner.setStoreTimestamp(result.getPrepareMessage().getStoreTimestamp
());
            // 将消息存储到真实Topic中，供Consumer消费
            RemotingCommand sendResult = sendFinalMessage(msgInner);
            if (sendResult.getCode() == ResponseCode.SUCCESS) {
                // 将消息存储到RMQ_SYS_TRANS_OP_HALF_TOPIC，标记为删除状态，
事务消息回查的定时任务中会做处理

this.brokerController.getTransactionalMessageService().deletePrepareMes
sage(result.getPrepareMessage());
            }
            return sendResult;
        }
        return res;
    }
} else if (MessageSysFlag.TRANSACTION_ROLLBACK_TYPE ==
requestHeader.getCommitOrRollback()) {
    result =
this.brokerController.getTransactionalMessageService().rollbackMessage(
requestHeader);
    if (result.getResponseCode() == ResponseCode.SUCCESS) {
        RemotingCommand res =
```

```
checkPrepareMessage(result.getPrepareMessage(), requestHeader);
        if (res.getCode() == ResponseCode.SUCCESS) {

this.brokerController.getTransactionalMessageService().deletePrepareMes
sage(result.getPrepareMessage());
        }
        return res;
    }
}
```

5. RocketMQ Broker端定时任务回查数据库事务部分

   方法入口：

   org.apache.rocketmq.broker.transaction.TransactionalMessageCheckService

```
@Override
protected void onWaitEnd() {
    long timeout =
brokerController.getBrokerConfig().getTransactionTimeOut();
    // 超过15次的回查事务状态失败后，默认是丢弃此消息
    int checkMax =
brokerController.getBrokerConfig().getTransactionCheckMax();
    long begin = System.currentTimeMillis();
    log.info("Begin to check prepare message, begin time:{}", begin);

this.brokerController.getTransactionalMessageService().check(timeout,
checkMax,
this.brokerController.getTransactionalMessageCheckListener());
    log.info("End to check prepare message, consumed time:{}",
System.currentTimeMillis() - begin);
}
```

回查事务调用入口：

```
// 此段代码为TransactionalMessageServiceImpl类中的check方法
List<MessageExt> opMsg = pullResult.getMsgFoundList();
boolean isNeedCheck = (opMsg == null && valueOfCurrentMinusBorn >
checkImmunityTime)
    || (opMsg != null && (opMsg.get(opMsg.size() -
1).getBornTimestamp() - startTime > transactionTimeout))
    || (valueOfCurrentMinusBorn <= -1
);

if (isNeedCheck) {
    if (!putBackHalfMsgQueue(msgExt, i)) {
        continue;
    }
    // 调用AbstractTransactionalMessageCheckListener的
    listener.resolveHalfMsg(msgExt);
} else {
    pullResult = fillOpRemoveMap(removeMap, opQueue,
pullResult.getNextBeginOffset(), halfOffset, doneOpOffset);
    log.info("The miss offset:{} in messageQueue:{} need to get more
opMsg, result is:{}", i,
        messageQueue, pullResult);
    continue;
}
```

```
// 此方法在AbstractTransactionalMessageCheckListener类中
public void resolveHalfMsg(final MessageExt msgExt) {
    executorService.execute(new Runnable() {
        @Override
        public void run() {
            try {
                sendCheckMessage(msgExt);
            } catch (Exception e) {
                LOGGER.error("Send check message error!", e);
            }
        }
    });
}

// 此方法在AbstractTransactionalMessageCheckListener类中
public void sendCheckMessage(MessageExt msgExt) throws Exception {
    CheckTransactionStateRequestHeader
checkTransactionStateRequestHeader = new
CheckTransactionStateRequestHeader();

checkTransactionStateRequestHeader.setCommitLogOffset(msgExt.getCommitL
ogOffset());

checkTransactionStateRequestHeader.setOffsetMsgId(msgExt.getMsgId());

checkTransactionStateRequestHeader.setMsgId(msgExt.getUserProperty(Mess
ageConst.PROPERTY_UNIQ_CLIENT_MESSAGE_ID_KEYIDX));

checkTransactionStateRequestHeader.setTransactionId(checkTransactionSta
teRequestHeader.getMsgId());

checkTransactionStateRequestHeader.setTranStateTableOffset(msgExt.getQu
eueOffset());

msgExt.setTopic(msgExt.getUserProperty(MessageConst.PROPERTY_REAL_TOPIC
));

msgExt.setQueueId(Integer.parseInt(msgExt.getUserProperty(MessageConst.
PROPERTY_REAL_QUEUE_ID)));
    msgExt.setStoreSize(0);
    String groupId =
msgExt.getProperty(MessageConst.PROPERTY_PRODUCER_GROUP);
    Channel channel =
brokerController.getProducerManager().getAvaliableChannel(groupId);
    if (channel != null) {
        // 通过Netty发送请求到RocketMQ Client端，执行checkTransactionState
方法

brokerController.getBroker2Client().checkProducerTransactionState(group
Id, channel, checkTransactionStateRequestHeader, msgExt);
    } else {
        LOGGER.warn("Check transaction failed, channel is null.
groupId={}", groupId);
    }
}
```

RocketMQ Client接收到服务端的请求后，重新调用回查数据库事务方法，并将事务结果再次提交到RocketMQ Broker端

方法入口：org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl类的方法

```
try {
    if (transactionCheckListener != null) {
        localTransactionState =
transactionCheckListener.checkLocalTransactionState(message);
    } else if (transactionListener != null) {
        log.debug("Used new check API in transaction message");
        localTransactionState =
transactionListener.checkLocalTransaction(message);
    } else {
        log.warn("CheckTransactionState, pick transactionListener by
group[{}] failed", group);
    }
} catch (Throwable e) {
    log.error("Broker call checkTransactionState, but
checkLocalTransactionState exception", e);
    exception = e;
}

this.processTransactionState(
    localTransactionState,
    group,
    exception);
```

六、补充一个问题

官网有提及，事务消息是不支持延迟消息和批量消息，我手贱试了一下延迟消息，事务消息设置一个DelayTimeLevel，结果这条消息就一直无法从RMQ_SYS_TRANS_HALF_TOPIC移除掉了，应用模块的日志发现在反复地尝试回查事务，Console界面上RMQ_SYS_TRANS_HALF_TOPIC的消息查询列表很快就超过2000条记录了，为什么?

我们回到代码层面进行分析，过程如下：

1.设置了DelayTimeLevel后，数据事务提交后（或是回查数据库事务完成后），将消息写入目标Topic时，由于DelayTimeLevel的干扰，目标Topic将变成SCHEDULE_TOPIC_XXXX，同时REAL_TOPIC变成RMQ_SYS_TRANS_HALF_TOPIC，真实的Topic在这个环节已经丢失。

```
// RocketMQ Broker端接受事务提交后的处理
org.apache.rocketmq.broker.processor.EndTransactionProcessor类
OperationResult result = new OperationResult();
if (MessageSysFlag.TRANSACTION_COMMIT_TYPE ==
requestHeader.getCommitOrRollback()) {
    // 这里调用CommitLog的putMessage方法
    result =
this.brokerController.getTransactionalMessageService().commitMessage(re
questHeader);
    if (result.getResponseCode() == ResponseCode.SUCCESS) {
        RemotingCommand res =
```

```
checkPrepareMessage(result.getPrepareMessage(), requestHeader);
        if (res.getCode() == ResponseCode.SUCCESS) {
            // 修改消息的Topic为由RMQ_SYS_TRANS_HALF_TOPIC改为真实Topic
            MessageExtBrokerInner msgInner =
endMessageTransaction(result.getPrepareMessage());

msgInner.setSysFlag(MessageSysFlag.resetTransactionValue(msgInner.getSy
sFlag(), requestHeader.getCommitOrRollback()));

msgInner.setQueueOffset(requestHeader.getTranStateTableOffset());

msgInner.setPreparedTransactionOffset(requestHeader.getCommitLogOffset(
));

msgInner.setStoreTimestamp(result.getPrepareMessage().getStoreTimestamp
());
            // 将消息存储到真实Topic中，此时Topic已经变成SCHEDULE_TOPIC_XXXX
            RemotingCommand sendResult = sendFinalMessage(msgInner);
            if (sendResult.getCode() == ResponseCode.SUCCESS) {
                // 将消息存储到RMQ_SYS_TRANS_OP_HALF_TOPIC，标记为删除状态，
事务消息回查的定时任务中会做处理

this.brokerController.getTransactionalMessageService().deletePrepareMes
sage(result.getPrepareMessage());
            }
            return sendResult;
        }
        return res;
    }
}

// 此段代码在org.apache.rocketmq.store.CommitLog类的putMessage方法中
// 由于DelayTimeLevel的干扰，目标Topic将变成SCHEDULE_TOPIC_XXXX
final int tranType =
MessageSysFlag.getTransactionValue(msg.getSysFlag());
if (tranType == MessageSysFlag.TRANSACTION_NOT_TYPE
    || tranType == MessageSysFlag.TRANSACTION_COMMIT_TYPE) {
    // Delay Delivery
    if (msg.getDelayTimeLevel() > 0) {
        if (msg.getDelayTimeLevel() >
this.defaultMessageStore.getScheduleMessageService().getMaxDelayLevel()
) {

msg.setDelayTimeLevel(this.defaultMessageStore.getScheduleMessageServic
e().getMaxDelayLevel());
        }

        topic = ScheduleMessageService.SCHEDULE_TOPIC;
        queueId =
ScheduleMessageService.delayLevel2QueueId(msg.getDelayTimeLevel());

        // Backup real topic, queueId
        MessageAccessor.putProperty(msg,
MessageConst.PROPERTY_REAL_TOPIC, msg.getTopic());
        MessageAccessor.putProperty(msg,
MessageConst.PROPERTY_REAL_QUEUE_ID, String.valueOf(msg.getQueueId()));

msg.setPropertiesString(MessageDecoder.messageProperties2String(msg.get
```

```
Properties()));

        msg.setTopic(topic);
        msg.setQueueId(queueId);
    }
}
```

打印的日志示例如下：

```
2019-10-17 14\:41\:05 INFO EndTransactionThread_4 - Transaction op
message write successfully. messageId=0A00851D00002A9F0000000000000E09,
queueId=0
msgExt:MessageExt [queueId=0, storeSize=335, queueOffset=5, sysFlag=8,
bornTimestamp=1571293959305, bornHost=/10.0.133.29:54634,
storeTimestamp=1571294460555,
storeHost=/10.0.133.29:10911, msgId=0A00851D00002A9F0000000000000E09,
commitLogOffset=3593, bodyCRC=1849408413, reconsumeTimes=0,
preparedTransactionOffset=0,
toString()=Message{topic='SCHEDULE_TOPIC_XXXX', flag=0, properties=
{REAL_TOPIC=RMQ_SYS_TRANS_HALF_TOPIC, TRANSACTION_CHECK_TIMES=3,
KEYS=msg-test-3,
TRAN_MSG=true, UNIQ_KEY=0A00851D422C18B4AAC25584B0880000, WAIT=false,
DELAY=1, PGROUP=transactionMQProducer, TAGS=transactionTest,
REAL_QID=0},
body=[72, 101, 108, 108, 111, 84, 105, 109, 101, 58, 51],
transactionId='null'}]
```

2.延迟消息是定时任务触发的，我刚刚设置的延迟是1秒，定时任务又把消息重新放回RMQ_SYS_TRANS_HALF_TOPIC中，注意此时只有RMQ_SYS_TRANS_HALF_TOPIC有消息，RMQ_SYS_TRANS_OP_HALF_TOPIC队列是没有这条消息的，如下代码：

```
// 此段代码在org.apache.rocketmq.store.schedule.ScheduleMessageService类
executeOnTimeup方法内
try {
    // 消息重新回到RMQ_SYS_TRANS_HALF_TOPIC队列中
    MessageExtBrokerInner msgInner = this.messageTimeup(msgExt);
    PutMessageResult putMessageResult =
        ScheduleMessageService.this.writeMessageStore
            .putMessage(msgInner);

    if (putMessageResult != null
        && putMessageResult.getPutMessageStatus() ==
PutMessageStatus.PUT_OK) {
        continue;
    } else {
        log.error(
            "ScheduleMessageService, a message time up, but reput it
failed, topic: {} msgId {}",
            msgExt.getTopic(), msgExt.getMsgId());
        ScheduleMessageService.this.timer.schedule(
            new DeliverDelayedMessageTimerTask(this.delayLevel,
                nextOffset), DELAY_FOR_A_PERIOD);
        ScheduleMessageService.this.updateOffset(this.delayLevel,
            nextOffset);
        return;
```

```
    }
} catch (Exception e) {
    log.error(
        "ScheduleMessageService, messageTimeup execute error, drop it.
msgExt="
            + msgExt + ", nextOffset=" + nextOffset + ",offsetPy="
            + offsetPy + ",sizePy=" + sizePy, e);
}
```

3.事务消息定时任务启动，查RMQ_SYS_TRANS_HALF_TOPIC有消息，但RMQ_SYS_TRANS_OP_HALF_TOPIC没有消息，为了保证消息顺序写入，又将此消息重新填入RMQ_SYS_TRANS_OP_HALF_TOPIC中，并且触发一次回查事务操作。示例代码如上文回查事务调用入口相同：

```
// 此段代码为TransactionalMessageServiceImpl类中的check方法
List<MessageExt> opMsg = pullResult.getMsgFoundList();
boolean isNeedCheck = (opMsg == null && valueOfCurrentMinusBorn >
checkImmunityTime)
    || (opMsg != null && (opMsg.get(opMsg.size() -
1).getBornTimestamp() - startTime > transactionTimeout))
    || (valueOfCurrentMinusBorn <= -1
);

if (isNeedCheck) {
    if (!putBackHalfMsgQueue(msgExt, i)) {
        continue;
    }
    listener.resolveHalfMsg(msgExt);
} else {
    pullResult = fillOpRemoveMap(removeMap, opQueue,
pullResult.getNextBeginOffset(), halfOffset, doneOpOffset);
    log.info("The miss offset:{} in messageQueue:{} need to get more
opMsg, result is:{}", i,
        messageQueue, pullResult);
    continue;
}
```

这样构成了一个死循环，直到尝试到15次才丢弃此消息（默认最大尝试次数是15次），这个代价有点大。针对此问题的优化，已经提交PR到RocketMQ社区，新版本发布后，事务消息将屏蔽DelayTimeLevel，这个问题就不会再出现了。

在新版本发布之前，我们的解决办法：

1. 明确研发过程中事务消息禁止设置DelayTimeLevel。
   感觉有风险，毕竟新来的童鞋，不是特别了解此部分功能的可能会手抖加上（像我最早那样）。
2. 对RocketMQ Client做一次简单的封装，比如提供一个rocketmq-spring-boot-starter，在提供发送事务消息的方法里不提供设置的入口，如下示例：

```
/**
 * 事务消息发送
 * 不支持延迟发送和批量发送
 */
public void sendMessageInTransaction(String topic, String tag, Object
```

```
message, String requestId) throws Exception {
   TransactionMQProducer producer = annotationScan.getProducer(topic +
"_" + tag);
   producer.sendMessageInTransaction(MessageBuilder.of(topic, tag,
message, requestId).build(), message);
}
```

应该靠谱一些，毕竟从源头杜绝了DelayTimeLevel参数的设置。

七、结束语

本篇简单介绍了事务消息的解决的场景和职责的界限，基本的设计思路和流程，在此借鉴学习了RocketMQ作者的图稿，然后挑了部分代码作简要的讲解，还是自己的刨坑过程，文章内有任何不正确或不详尽之处请留言指导，谢谢。

专注Java高并发、分布式架构，更多技术干货分享与心得，请关注公众号：Java架构社区

标签： Github， RocketMQ

清茶豆奶
关注 - 7
粉丝 - 9
+加关注

0 0

« 上一篇： 使用Spring-boot-starter标准改造项目内的RocketMQ客户端组件
» 下一篇： 一篇文章彻底搞懂snowflake算法及百度美团的最佳实践

posted @ 2019-10-19 08:45 清茶豆奶 阅读(1189) 评论(0) 编辑 收藏