RocketMQ Broker端事务消息源码解读

BY：whr

基于realease-4.9.3版本

初始化

在broker启动时，`org.apache.rocketmq.broker.BrokerController#initialTransaction`和`org.apache.rocketmq.broker.BrokerController#registerProcessor`两个代码段启动了和事务消息处理相关的服务，注册了处理事务消息提交的处理器。

和事务消息相关的服务有`transactionalMessageService`，`transactionalMessageCheckService`和`transactionalMessageCheckListener`。

1. `transactionalMessageService`不继承`ServiceThread`,其并不是定时调度的服务，它等待其他服务的调用。
2. `transactionalMessageCheckService`由`start()`方法调用`startProcessorByHa()`启动。它的工作是每经过`checkInterval`(默认为1分钟)，调用一次`transactionalMessageService`的`check()`方法。
3. `transactionalMessageCheckListener`实现了向生产者回查事务消息状态的方法。

```
//org.apache.rocketmq.broker.transaction.TransactionalMessageCheckService
public void run() {
    log.info("Start transaction check service thread!");
    long checkInterval =
brokerController.getBrokerConfig().getTransactionCheckInterval();
    while (!this.isStopped()) {
        this.waitForRunning(checkInterval);
    }
    log.info("End transaction check service thread!");
}
protected void onWaitEnd() {
    long timeout =
brokerController.getBrokerConfig().getTransactionTimeOut();
    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);
}
```

4. `transactionalMessageCheckListener`是`transactionalMessageCheckService`调用`transactionalMessageService`的`check()`方法时传入的参数。其默认实现`DefaultTransactionalMessageCheckListener`具有两个方法：`resolveDiscardMsg`和`toMessageExtBrokerInner`。`resolveDiscardMsg`负责将超时的消息发送到一个特殊的消息队

列`TRANS_CHECK_MAXTIME_TOPIC`；`toMessageExtBrokerInner`仅由`resolveDiscardMsg`所调用，其将消息包装为特殊的消息队列`TRANS_CHECK_MAXTIME_TOPIC`主题的消息。

Broker端事务消息处理流程

事务消息和其他种类的消息处理逻辑的分流从这里开始

org.apache.rocketmq.broker.processor.SendMessageProcessor#asyncSendMessage

```
private CompletableFuture<RemotingCommand>
asyncSendMessage(ChannelHandlerContext ctx, RemotingCommand request,
SendMessageContext mqtraceContext, SendMessageRequestHeader requestHeader) {
    ...

    CompletableFuture<PutMessageResult> putMessageResult = null;
    String transFlag =
origProps.get(MessageConst.PROPERTY_TRANSACTION_PREPARED);
    if (transFlag != null && Boolean.parseBoolean(transFlag)) {
        if
(this.brokerController.getBrokerConfig().isRejectTransactionMessage()) {
            response.setCode(ResponseCode.NO_PERMISSION);
            response.setRemark(
                "the broker[" +
this.brokerController.getBrokerConfig().getBrokerIP1()
                + "] sending transaction message is forbidden");
            return CompletableFuture.completedFuture(response);
        }
        putMessageResult =
this.brokerController.getTransactionalMessageService().asyncPrepareMessage(msgIn
ner);
    } else {
        putMessageResult =
this.brokerController.getMessageStore().asyncPutMessage(msgInner);
    }
    return handlePutMessageResultFuture(putMessageResult, response, request,
msgInner, responseHeader, mqtraceContext, ctx, queueIdInt);
}
```

其中`origProps`是原消息的属性。如果原消息携带`PROPERTY_TRANSACTION_PREPARED`属性，则交由`TransactionalMessageService`来处理；否则直接调用`MessageStore`的`asyncPutMessage()`方法放入消息队列。

`org.apache.rocketmq.broker.transaction.queue.TransactionalMessageServiceImpl`是默认的`TransactionalMessageService`的实现类。

```
public CompletableFuture<PutMessageResult>
asyncPrepareMessage(MessageExtBrokerInner messageInner) {
    return transactionalMessageBridge.asyncPutHalfMessage(messageInner);
}
public CompletableFuture<PutMessageResult>
asyncPutHalfMessage(MessageExtBrokerInner messageInner) {
    return store.asyncPutMessage(parseHalfMessageInner(messageInner));
}
```

```
private MessageExtBrokerInner parseHalfMessageInner(MessageExtBrokerInner
msgInner) {
    //保存原始Topic信息
    MessageAccessor.putProperty(msgInner, MessageConst.PROPERTY_REAL_TOPIC,
msgInner.getTopic());
    MessageAccessor.putProperty(msgInner, MessageConst.PROPERTY_REAL_QUEUE_ID,
                                String.valueOf(msgInner.getQueueId()));
    //设置Transaction状态位
    msgInner.setSysFlag(
        MessageSysFlag.resetTransactionValue(msgInner.getSysFlag(),
MessageSysFlag.TRANSACTION_NOT_TYPE));
    //设置Topic为半消息队列的主题
    msgInner.setTopic(TransactionalMessageUtil.buildHalfTopic());
    msgInner.setQueueId(0);
    //保存消息原始属性

 msgInner.setPropertiesString(MessageDecoder.messageProperties2String(msgInner.g
etProperties()));
    return msgInner;
}
```

这里的处理也不复杂，只是调用了`parseHalfMessageInner`方法对消息进行了一次包装。将真实的Topic存储为`PROPERTY_REAL_TOPIC`属性，将真实的QueueId存储为`PROPERTY_REAL_QUEUE_ID`属性，重设MessageSysFlag的Transaction相关位；并设置新的Topic为半消息的主题`RMQ_SYS_TRANS_HALF_TOPIC`，新的消息队列ID为0。然后调用`store`的方法将包装过的消息放入半消息队列，等待检查。

生产者主动提交本地事务状态

当生产者执行完成本地事务后，可以发送一条EndTransaction消息。这里的消息的请求码和普通消息不同，为`RequestCode.END_TRANSACTION`，由`EndTransactionProcessor`类进行处理。

其主要的处理流程在`org.apache.rocketmq.broker.processor.EndTransactionProcessor#processRequest`方法里。

```
//如果是提交消息（TRANSACTION_COMMIT_TYPE）
if (MessageSysFlag.TRANSACTION_COMMIT_TYPE ==
requestHeader.getCommitOrRollback()) {
    //这里的方法名虽然是commitMessage，但是查看其默认实现可以知道只是取得对应的半消息，没有任
何修改操作
    result =
this.brokerController.getTransactionalMessageService().commitMessage(requestHead
er);
    if (result.getResponseCode() == ResponseCode.SUCCESS) {
        //检查消息合法性
        RemotingCommand res = checkPrepareMessage(result.getPrepareMessage(),
requestHeader);
        if (res.getCode() == ResponseCode.SUCCESS) {
            //取得半消息队列中对应的消息，调用endMessageTransaction恢复原始消息信息
（Topic，QueueID等等）
            //注意这里result.getPrepareMessage()所取得的的消息Topic仍为半消息的主题
`RMQ_SYS_TRANS_HALF_TOPIC`
```

```
            MessageExtBrokerInner msgInner =
endMessageTransaction(result.getPrepareMessage());

 msgInner.setSysFlag(MessageSysFlag.resetTransactionValue(msgInner.getSysFlag(),
requestHeader.getCommitOrRollback()));
            msgInner.setQueueOffset(requestHeader.getTranStateTableOffset());

 msgInner.setPreparedTransactionOffset(requestHeader.getCommitLogOffset());

 msgInner.setStoreTimestamp(result.getPrepareMessage().getStoreTimestamp());
            MessageAccessor.clearProperty(msgInner,
MessageConst.PROPERTY_TRANSACTION_PREPARED);
            //将恢复后的事务消息投放到原始消息队列中供消费者使用
            RemotingCommand sendResult = sendFinalMessage(msgInner);
            //丢弃半消息队列中的原始消息
            if (sendResult.getCode() == ResponseCode.SUCCESS) {

 this.brokerController.getTransactionalMessageService().deletePrepareMessage(res
ult.getPrepareMessage());
            }
            return sendResult;
        }
        return res;
    }
}
//如果是回滚消息（TRANSACTION_ROLLBACK_TYPE）
else if (MessageSysFlag.TRANSACTION_ROLLBACK_TYPE ==
requestHeader.getCommitOrRollback()) {
    //这里的方法名虽然是rollbackMessage，但是查看其默认实现可以知道只是取得对应的半消息，没有
任何修改操作
    result =
this.brokerController.getTransactionalMessageService().rollbackMessage(requestHe
ader);
    if (result.getResponseCode() == ResponseCode.SUCCESS) {
        //注意这里result.getPrepareMessage()所取得的的消息Topic仍为半消息的主题
`RMQ_SYS_TRANS_HALF_TOPIC`
        RemotingCommand res = checkPrepareMessage(result.getPrepareMessage(),
requestHeader);
        //丢弃半消息队列中的原始消息
        if (res.getCode() == ResponseCode.SUCCESS) {

 this.brokerController.getTransactionalMessageService().deletePrepareMessage(res
ult.getPrepareMessage());
        }
        return res;
    }
}
```

跟踪其中丢弃半消息队列中的原始消息所调用的方法`deletePrepareMessage()`，其执行

```
this.transactionalMessageBridge.putOpMessage(msgExt,
TransactionalMessageUtil.REMOVETAG);
```

org.apache.rocketmq.broker.transaction.queue.TransactionalMessageBridge#putOpMessage

```
public boolean putOpMessage(MessageExt messageExt, String opType) {
    //由于Topic为统一的半消息队列，QueueID为0，所以生成的操作队列是同样的
    MessageQueue messageQueue = new MessageQueue(messageExt.getTopic(),
        this.brokerController.getBrokerConfig().getBrokerName(),
messageExt.getQueueId());
    if (TransactionalMessageUtil.REMOVETAG.equals(opType)) {
        return addRemoveTagInTransactionOp(messageExt, messageQueue);
    }
    return true;
}
```

```
/**
* Use this function while transaction msg is committed or rollback write a flag
'd' to operation queue for the msg's offset
*
* @param prepareMessage Half message
* @param messageQueue Half message queue
* @return This method will always return true.
*/
private boolean addRemoveTagInTransactionOp(MessageExt prepareMessage,
MessageQueue messageQueue) {
    Message message = new Message(TransactionalMessageUtil.buildOpTopic(),
TransactionalMessageUtil.REMOVETAG,
String.valueOf(prepareMessage.getQueueOffset()).getBytes(TransactionalMessageUti
l.charset));
    writeOp(message, messageQueue);
    return true;
}
```

到这里可以看出，事务消息的结束（提交或回滚）并不会直接从半消息队列中删除对应的消息，而是在另一个消息队列（操作队列OpQueue，其Topic为RMQ_SYS_TRANS_OP_HALF_TOPIC）中写入一条具有`REMOVETAG`的消息，这条消息的消息体存储了其对应的半消息消费位点，指向对应的半消息。

Broker检查半消息队列RMQ_SYS_TRANS_HALF_TOPIC

我们在初始化的时候提到，Broker在启动时定时调度`transactionalMessageCheckService`。它的工作是每经过`checkInterval`(默认为1分钟)，调用一次`transactionalMessageService`的`check()`方法。半消息队列`RMQ_SYS_TRANS_HALF_TOPIC`的绝大部分处理逻辑都在此时进行。

`check()`方法所做的有：

1. 遍历半消息消息队列
2. 对于每个半消息队列（默认实现中只有一个队列），调用`fillOpRemoveMap()`方法**检查对应的操作队列（OpQueue）中的消息**，将要移除的消息位点存入`HashMap<Long, Long> removeMap`中。
3. 遍历**半消息队列**，
    1. 如果遇到的消息在`HashMap<Long, Long> removeMap`中，则跳过该消息；
    2. 如果该消息是新到达的或者最近被检查过，跳过该消息；
    3. 如果消息回查次数过多（needDiscard返回true）或超时（needSkip为true），丢弃该消息；
    4. 对于其他消息，将消息放回到半消息队列中。注意此时放入的是新的消息，其在队列的最后的位置。此时消费位点正常向后移动。相当于原消息被消费，同时向队列中放入一份新的复制。

并且对该消息发起回查。

```
public void check(long transactionTimeout, int transactionCheckMax,
                  AbstractTransactionalMessageCheckListener listener) {
    try {
        //取得半消息消息队列
        String topic = TopicValidator.RMQ_SYS_TRANS_HALF_TOPIC;
        Set<MessageQueue> msgQueues =
transactionalMessageBridge.fetchMessageQueues(topic);
        if (msgQueues == null || msgQueues.size() == 0) {
            log.warn("The queue of topic is empty :" + topic);
            return;
        }
        log.debug("Check topic={}, queues={}", topic, msgQueues);
        //遍历半消息消息队列
        for (MessageQueue messageQueue : msgQueues) {
            long startTime = System.currentTimeMillis();
            MessageQueue opQueue = getOpQueue(messageQueue);
            long halfOffset =
transactionalMessageBridge.fetchConsumeOffset(messageQueue);
            long opOffset =
transactionalMessageBridge.fetchConsumeOffset(opQueue);
            log.info("Before check, the queue={} msgOffset={} opOffset={}",
messageQueue, halfOffset, opOffset);
            if (halfOffset < 0 || opOffset < 0) {
                log.error("MessageQueue: {} illegal offset read: {}, op offset:
{},skip this queue", messageQueue,
                        halfOffset, opOffset);
                continue;
            }

            List<Long> doneOpOffset = new ArrayList<>();
            HashMap<Long, Long> removeMap = new HashMap<>();
            //检查需要移除的消息，即读取操作队列消息（32条）
            PullResult pullResult = fillOpRemoveMap(removeMap, opQueue,
opOffset, halfOffset, doneOpOffset);
            if (null == pullResult) {
                log.error("The queue={} check msgOffset={} with opOffset={}
failed, pullResult is null",
                        messageQueue, halfOffset, opOffset);
                continue;
            }
            // single thread
            int getMessageNullCount = 1;
            long newOffset = halfOffset;
            //以变量i为索引，遍历半消息队列
            long i = halfOffset;
            while (true) {
                if (System.currentTimeMillis() - startTime >
MAX_PROCESS_TIME_LIMIT) {
                    log.info("Queue={} process time reach max={}", messageQueue,
MAX_PROCESS_TIME_LIMIT);
                    break;
                }
                //如果消息需要移除（已经有相应操作在OpQueue中）
```

```
                if (removeMap.containsKey(i)) {
                    log.debug("Half offset {} has been committed/rolled back",
i);
                    Long removedOpOffset = removeMap.remove(i);
                    doneOpOffset.add(removedOpOffset);
                } else { //消息没有对应的提交/回滚操作，需要回查
                    GetResult getResult = getHalfMsg(messageQueue, i);
                    MessageExt msgExt = getResult.getMsg();

                    //没有取得对应的消息，一些边界处理
                    if (msgExt == null) {
                        ...
                    }

                    //如果消息回查次数过多（needDiscard返回true）或超时（needSkip为
true），丢弃该消息
                    if (needDiscard(msgExt, transactionCheckMax) ||
needSkip(msgExt)) {
                        listener.resolveDiscardMsg(msgExt);
                        newOffset = i + 1;
                        i++;
                        continue;
                    }
                    //新到达的消息跳过回查，此时说明之后的消息都是新到达的，跳过该队列的消息
回查
                    if (msgExt.getStoreTimestamp() >= startTime) {
                        log.debug("Fresh stored. the miss offset={}, check it
later, store={}", i,
                                  new Date(msgExt.getStoreTimestamp()));
                        break;
                    }

                    long valueOfCurrentMinusBorn = System.currentTimeMillis() -
msgExt.getBornTimestamp();
                    long checkImmunityTime = transactionTimeout;
                    String checkImmunityTimeStr =
msgExt.getUserProperty(MessageConst.PROPERTY_CHECK_IMMUNITY_TIME_IN_SECONDS);
                    //消息在回查过后会有一段时间（checkImmunityTime）不会被检查，如果在
ImmunityTime中，跳过该消息的回查
                    if (null != checkImmunityTimeStr) {
                        checkImmunityTime =
getImmunityTime(checkImmunityTimeStr, transactionTimeout);
                        if (valueOfCurrentMinusBorn < checkImmunityTime) {
                            if (checkPrepareQueueOffset(removeMap, doneOpOffset,
msgExt)) {
                                newOffset = i + 1;
                                i++;
                                continue;
                            }
                        }
                    } else {
                        //新到达的消息跳过回查，此时说明之后的消息都是新到达的，跳过该队列的
消息回查
                        //注意这里新到达的消息的定义和上面不相同
```

```
                        if ((0 <= valueOfCurrentMinusBorn) &&
(valueOfCurrentMinusBorn < checkImmunityTime)) {
                            log.debug("New arrived, the miss offset={}, check it
later checkImmunity={}, born={}", i,
                                      checkImmunityTime, new
Date(msgExt.getBornTimestamp()));
                            break;
                        }
                    }
                    List<MessageExt> opMsg = pullResult.getMsgFoundList();
                    //重新检查消息时间，判断消息是否需要回查
                    boolean isNeedCheck = (opMsg == null &&
valueOfCurrentMinusBorn > checkImmunityTime)
                        || (opMsg != null && (opMsg.get(opMsg.size() -
1).getBornTimestamp() - startTime > transactionTimeout))
                        || (valueOfCurrentMinusBorn <= -1);

                    if (isNeedCheck) {
                        //将消息重新放入半消息队列中
                        //注意此时放入的是新的消息，其在队列的最后的位置
                        if (!putBackHalfMsgQueue(msgExt, i)) {
                            continue;
                        }
                        //像生产者发起消息回查
                        listener.resolveHalfMsg(msgExt);
                    } else {
                        //读取操作队列之后32条消息
                        pullResult = fillOpRemoveMap(removeMap, opQueue,
pullResult.getNextBeginOffset(), halfOffset, doneOpOffset);
                        log.debug("The miss offset:{} in messageQueue:{} need to
get more opMsg, result is:{}", i,
                                  messageQueue, pullResult);
                        continue;
                    }
                }
                newOffset = i + 1;
                i++;
            }
            //更新半消息队列及操作队列消费位点
            if (newOffset != halfOffset) {
                transactionalMessageBridge.updateConsumeOffset(messageQueue,
newOffset);
            }
            long newOpOffset = calculateOpOffset(doneOpOffset, opOffset);
            if (newOpOffset != opOffset) {
                transactionalMessageBridge.updateConsumeOffset(opQueue,
newOpOffset);
            }
        }
    } catch (Throwable e) {
        log.error("Check error", e);
    }

}
```