面试必问：分布式事务六种解决方案

敖丙
职场领域创作者
关注他

> 点赞再看，养成习惯，微信搜一搜【**三太子敖丙**】关注这个喜欢写情怀的程序员。
> 本文 **GitHub** https://github.com/JavaFamily 已收录，有一线大厂面试完整考点、资料以及我的系列文章。

前言

上一篇文章已经讲完分布式了，那暖男说要讲分布式事务那就一定会讲，只是我估计大家没料到暖男这么快就肝好了吧？

事务想必大家并不陌生，至于什么是 ACID，也是老生常谈了。不过暖男为了保证文章的完整性确保所有人都听得懂，我还是得先说说 ACID，然后再来介绍下什么是分布式事务和常见的分布式事务包括 2PC、3PC、TCC、本地消息表、消息事务、最大努力通知。

事务

严格意义上的事务实现应该是**具备原子性、一致性、隔离性和持久性**，简称 ACID。

- 原子性（Atomicity），可以理解为一个事务内的所有操作要么都执行，要么都不执行。
- 一致性（Consistency），可以理解为数据是满足完整性约束的，也就是不会存在中间状态的数据，比如你账上有400，我账上有100，你给我打200块，此时你账上的钱应该是200，我账上的钱应该是300，不会存在我账上钱加了，你账上钱没扣的**中间状态**。
- 隔离性（Isolation），指的是多个事务并发执行的时候不会互相干扰，即一个事务内部的数据对于其他事务来说是隔离的。
- 持久性（Durability），指的是一个事务完成了之后数据就被永远保存下来，之后的其他操作或故障都不会对事务的结果产生影响。

而通俗意义上事务就是为了使得一些更新操作要么都成功，要么都失败。

说到这里可能有人会说，不对啊 Redis 的事务不能保证所有操作要么都执行要么都不执行，为什么它也叫事务啊？

一般而言他们既然敢说出他们实现了什么什么，要么是真的实现了，要么是在某种特殊、定制或者极端的条件下才能满足功能。

我们来看看 Redis 怎么说的。

> It's important to note that **even when a command fails, all the other commands in the queue are processed** – Redis will *not* stop the processing of commands.

这句话就是告诉大家事务中的某个命令失败了，之后的命令还是会被处理，Redis 不会停止命令，意味着也不会回滚。

你说这不是扯嘛？这都偏离事务最核心的本意了啊。

别急，咱们来看看 Redis 怎么解释的。

> - Redis commands can fail only if called with a wrong syntax (and the problem is not detectable during the command queueing), or against keys holding the wrong data type: this means that in practical terms a failing command is the result of a programming errors, and a kind of error that is very likely to be detected during development, and not in production.
> - Redis is internally simplified and faster because it does not need the ability to roll back.

Redis 官网解释了为什么不支持回滚，他们说首先如果命令出错那都是语法使用错误，**是你们自己编程出错**，而且这种情况应该在开发的时候就被检测出来，不应在生产环境中出现。

然后 Redis 就是为了快！不需要提供回滚。

下面还有一段话我就不截图了，就是说就算提供回滚也没用，你这代码都写错了，**回滚并不能使你免于编程错误**。而且一般这种错也不可能进入到生产环境，所以选择更加简单、快速的方法，我们不支持回滚。

你看看这说的好像很有道理，我们不提供回滚，因为我们不需要为你的编程错误买单！

但好像哪里不对劲？角度、立场不同，大家自己品。

就下来就开始分布式事务。

分布式事务

分布式事务顾名思义就是要在分布式系统中实现事务，它其实是由多个本地事务组合而成。

对于分布式事务而言几乎满足不了 ACID，其实对于单机事务而言大部分情况下也没有满足 ACID，不然怎么会有四种隔离级别呢？所以更别说分布在不同数据库或者不同应用上的分布式事务了。

我们先来看下 2PC。

2PC

2PC（Two-phase commit protocol），中文叫二阶段提交。 **二阶段提交是一种强一致性设计**，2PC 引入一个事务协调者的角色来协调管理各参与者（也可称之为各本地资源）的提交和回滚，二阶段分别指的是准备（投票）和提交两个阶段。

注意这只是协议或者说是理论指导，只阐述了大方向，具体落地还是有会有差异的。

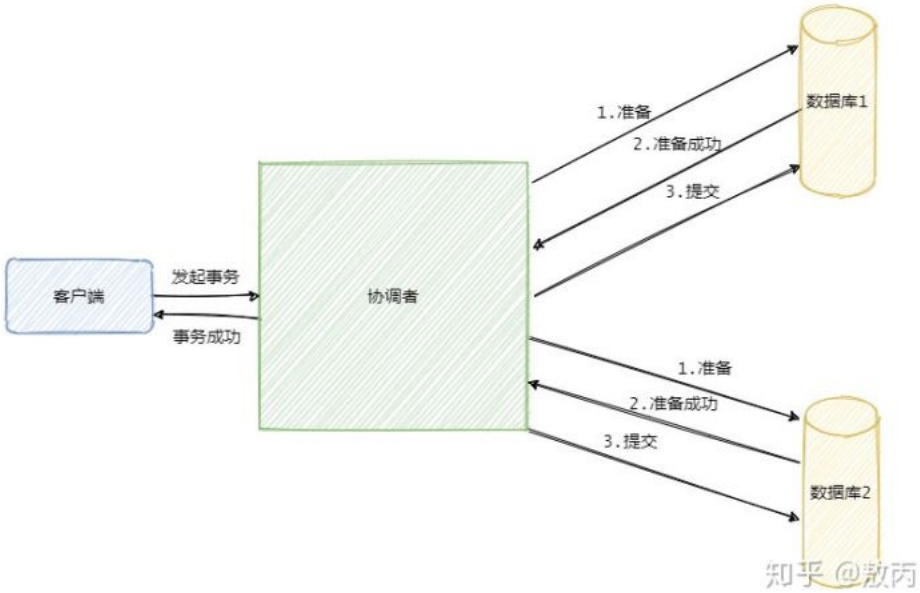
让我们来看下两个阶段的具体流程。

准备阶段协调者会给各参与者发送准备命令，你可以把准备命令理解成除了提交事务之外啥事都做完了。

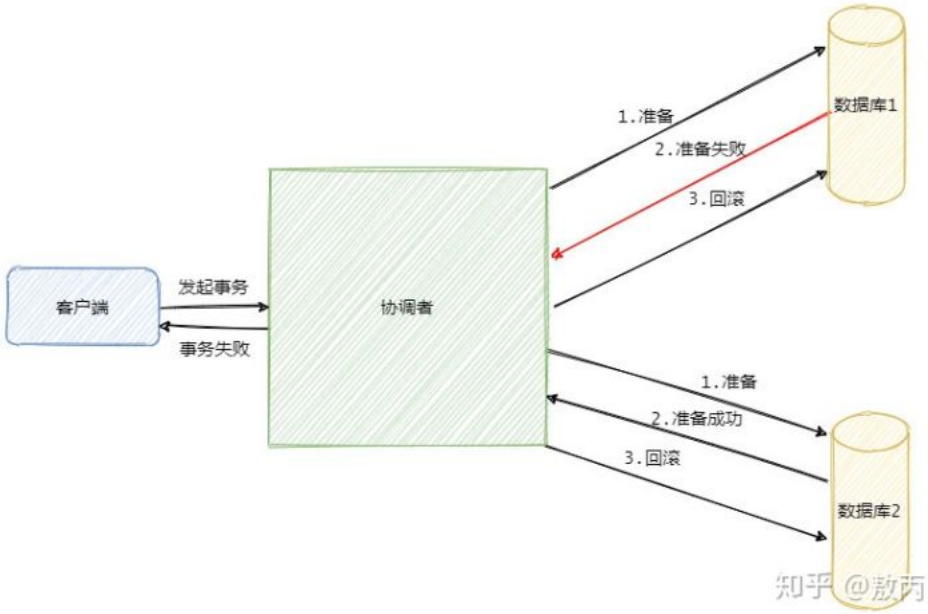
定是提交事务，也可

假如在第一阶段所有参与者都返回准备成功，那么协调者则向所有参与者发送提交事务命令，然后等待所有事务都提交成功之后，返回事务执行成功。

让我们来看一下流程图。



假如在第一阶段有一个参与者返回失败，那么协调者就会向所有参与者发送回滚事务的请求，即分布式事务执行失败。



那可能就有人问了，那第二阶段提交失败的话呢？

这里有两种情况。

第一种是**第二阶段执行的是回滚事务操作**，那么答案是不断重试，直到所有参与者都回滚了，不然那些在第一阶段准备成功的参与者会一直阻塞着。

第二种是**第二阶段执行的是提交事务操作**，那么答案也是不断重试，因为有可能一些参与者的事务已经提交成功了，这个时候只有一条路，就是头铁往前冲，不断的重试，直到提交成功，到最后真的不行只能人工介入处理。

大体上二阶段提交的流程就是这样，**我们再来看看细节。**

进行下一步操作，
应或某参与者挂了，

那么超时后就会判断事务失败，向所有参与者发送回滚命令。

在第二阶段协调者的没法超时，因为按照我们上面分析只能不断重试！

协调者故障分析

协调者是一个单点，存在单点故障问题。

假设协调者在**发送准备命令之前**挂了，还行等于事务还没开始。

假设协调者在**发送准备命令之后**挂了，这就不太行了，有些参与者等于都执行了处于事务资源锁定的状态。不仅事务执行不下去，还会因为锁定了一些公共资源而阻塞系统其它操作。

假设协调者在**发送回滚事务命令之前**挂了，那么事务也是执行不下去，且在第一阶段那些准备成功参与者都阻塞着。

假设协调者在**发送回滚事务命令之后**挂了，这个还行，至少命令发出去了，很大的概率都会回滚成功，资源都会释放。但是如果出现网络分区问题，某些参与者将因为收不到命令而阻塞着。

假设协调者在**发送提交事务命令之前**挂了，这个不行，傻了！这下是所有资源都阻塞着。

假设协调者在**发送提交事务命令之后**挂了，这个还行，也是至少命令发出去了，很大概率都会提交成功，然后释放资源，但是如果出现网络分区问题某些参与者将因为收不到命令而阻塞着。

协调者故障，通过选举得到新协调者

因为协调者单点问题，因此我们可以通过选举等操作选出一个新协调者来顶替。

如果处于第一阶段，其实影响不大都回滚好了，在第一阶段事务肯定还没提交。

如果处于第二阶段，假设参与者都没挂，此时新协调者可以向所有参与者确认它们自身情况来推断下一步的操作。

假设有个别参与者挂了！这就有点僵硬了，比如协调者发送了回滚命令，此时第一个参与者收到了并执行，然后协调者和第一个参与者都挂了。

此时其他参与者都没收到请求，然后新协调者来了，它询问其他参与者都说OK，但它不知道挂了的那个参与者到底O不OK，所以它傻了。

问题其实就出在**每个参与者自身的状态只有自己和协调者知道**，因此新协调者无法通过在场的参与者的状态推断出挂了的参与者是什么情况。

求在哪个地方记一

但是就算协调者知道自己该发提交请求，那么在参与者也一起挂了的情况下没用，因为你不知道参与者在挂之前有没有提交事务。

如果参与者在挂之前事务提交成功，新协调者确定存活着的参与者都没问题，那肯定得向其他参与者发送提交事务命令才能保证数据一致。

如果参与者在挂之前事务还未提交成功，参与者恢复了之后数据是回滚的，此时协调者必须是向其他参与者发送回滚事务命令才能保持事务的一致。

所以说极端情况下还是无法避免数据不一致问题。

talk is cheap 让我们再来看下代码，可能更加的清晰。以下代码取自 `<<Distributed System: Principles and Paradigms>>` 。

这个代码就是实现了 2PC，但是相比于2PC增加了写日志的动作、参与者之间还会互相通知、参与者也实现了超时。这里要注意，一般所说的2PC，不含上述功能，这都是实现的时候添加的。

```
协调者:
    write START_2PC to local log; //开始事务
    multicast VOTE_REQUEST to all participants; //广播通知参与者投票
    while not all votes have been collected {
        wait for any incoming vote;
        if timeout { //协调者超时
            write GLOBAL_ABORT to local log; //写日志
            multicast GLOBAL_ABORT to all participants; //通知事务中断
            exit;
        }
        record vote;
    }
    //如果所有参与者都ok
    if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
        write GLOBAL_COMMIT to local log;
        multicast GLOBAL_COMMIT to all participants;
    } else {
        write GLOBAL_ABORT to local log;
        multicast GLOBAL_ABORT to all participants;
    }

参与者:

    write INIT to local log; //写日志
    wait for VOTE_REQUEST from coordinator;
    if timeout { //等待超时
        write VOTE_ABORT to local log;
        exit;
    }
    if participant votes COMMIT {
```

```
        send VOTE_COMMIT to coordinator;
        wait for DECISION from coordinator;
        if timeout {
            multicast DECISION_REQUEST to other participants; //超时通知
            wait until DECISION is received;  /* remain blocked*/
            write DECISION to local log;
        }
        if DECISION == GLOBAL_COMMIT
            write GLOBAL_COMMIT to local log;
        else if DECISION == GLOBAL_ABORT
            write GLOBAL_ABORT to local log;
    } else {
        write VOTE_ABORT to local log;
        send VOTE_ABORT to coordinator;
    }
```

每个参与者维护一个线程处理其它参与者的DECISION_REQUEST请求：

```
    while true {
        wait until any incoming DECISION_REQUEST is received;
        read most recently recorded STATE from the local log;
        if STATE == GLOBAL_COMMIT
            send GLOBAL_COMMIT to requesting participant;
        else if STATE == INIT or STATE == GLOBAL_ABORT;
            send GLOBAL_ABORT to requesting participant;
        else
            skip;  /* participant remains blocked */
    }
```

至此我们已经详细的分析的 2PC 的各种细节，我们来总结一下！

2PC 是一种**尽量保证强一致性**的分布式事务，因此它是**同步阻塞**的，而同步阻塞就导致长久的资源锁定问题，**总体而言效率低**，并且存在**单点故障**问题，在极端条件下存在**数据不一致**的风险。

当然具体的实现可以变形，而且 2PC 也有变种，例如 Tree 2PC、Dynamic 2PC。

还有一点不知道你们看出来没，2PC 适用于**数据库层面的分布式事务场景**，而我们业务需求有时候不仅仅关乎数据库，也有可能是上传一张图片或者发送一条短信。

而且像 Java 中的 JTA 只能解决一个应用下多数据库的分布式事务问题，跨服务了就不能用了。

简单说下 Java 中 JTA，它是基于XA规范实现的事务接口，这里的 XA 你可以简单理解为基于数据库的 XA 规范来实现的 2PC。（至于XA规范到底是啥，篇幅有限，下次有机会再说）

接下来我们再来看看 3PC。

3PC

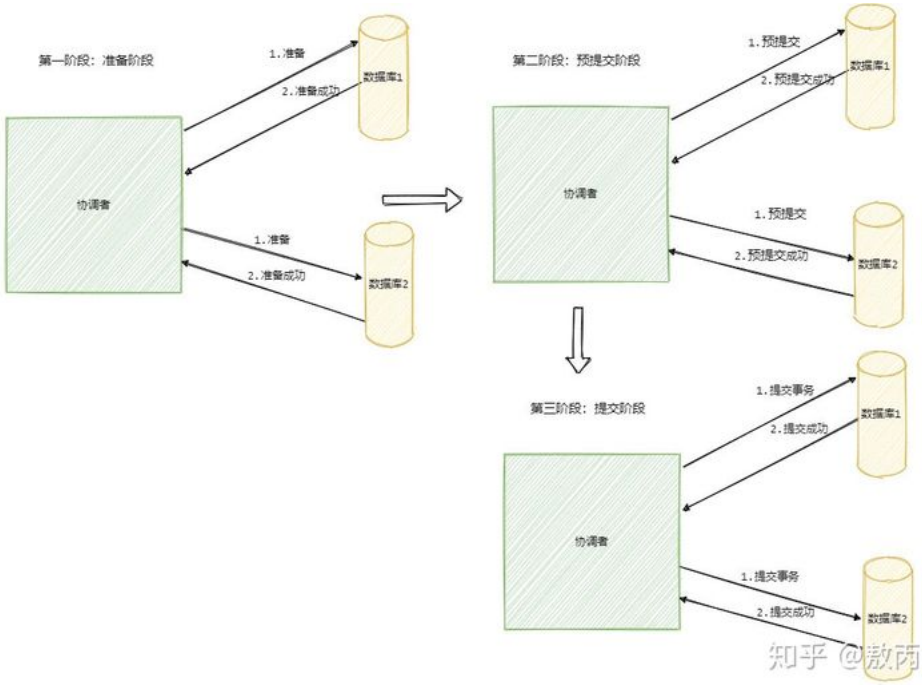
3PC 的出现是为了解决 2PC 的一些问题，相比于 2PC 它在**参与者中也引入了超时机制**，并且**新增了一个阶段**使得参与者可以利用这一个阶段统一各自的状态。

让我们来详细看一下。

3PC 包含了三个阶段，分别是**准备阶段、预提交阶段和提交阶段**，对应的英文就是：`CanCommit`、`PreCommit` 和 `DoCommit`。

看起来是**把 2PC 的提交阶段变成了预提交阶段和提交阶段**，但是 3PC 的准备阶段协调者只是询问参与者的自身状况，比如你现在还好吗？负载重不重？这类的。

提交阶段和 2PC 的一样，让我们来看一下图。



不管哪一个阶段有参与者返回失败都会宣布事务失败，这和 2PC 是一样的（当然到最后的提交阶段和 2PC 一样只要是提交请求就只能不断重试）。

我们先来看一下 3PC 的阶段变更有什么影响。

首先**准备阶段的变更成不会直接执行事务**，而是会先去询问此时的参与者是否有条件接这个事务，因此**不会一来就干活直接锁资源**，使得在某些资源不可用的情况下所有参与者都阻塞着。

而**预提交阶段的引入起到了一个统一状态的作用**，它像一道栅栏，表明在预提交阶段前所有参与者其实还未都回应，在预处理阶段表明所有参与者都已经回应了。

假如你是一位参与者，你知道自己进入了预提交状态那你就可以推断出来其他参与者也都进入了预提交状态。

但是多引入一个阶段也多一个交互，因此**性能会差一些**，而且**绝大部分的情况下资源应该都是可用的**，这样等于每次明知可用执行还得询问一次。

我们再来看下参与者超时能带来什么样的影响。

我们知道 2PC 是同步阻塞的，上面我们已经分析了协调者挂在了提交请求还未发出去的时候是最伤的，所有参与者都已经锁定资源并且阻塞等待着。

那么引入了超时机制，参与者就不会傻等了，**如果是等待提交命令超时，那么参与者就会提交事务了**，因为都到了这一阶段了大概率是提交的，**如果是等待预提交命令超时，那该干啥就干啥了，反正本来啥也没干。**

然而超时机制也会带来数据不一致的问题，比如在等待提交命令时候超时了，参与者默认执行的是提交事务操作，但是**有可能执行的是回滚操作，这样一来数据就不一致了。**

当然 3PC 协调者超时还是在的，具体不分析了和 2PC 是一样的。

从维基百科上看，3PC 的引入是为了解决提交阶段 2PC 协调者和某参与者都挂了之后新选举的协调者不知道当前应该提交还是回滚的问题。

新协调者来的时候发现有一个参与者处于预提交或者提交阶段，那么表明已经经过了所有参与者的确认了，所以此时执行的就是提交命令。

所以说 3PC 就是通过引入预提交阶段来使得参与者之间的状态得到统一，也就是留了一个阶段让大家同步一下。

但是这也只能让协调者知道该如果做，但不能保证这样做一定对，这其实和上面 2PC 分析一致，因为挂了的参与者到底有没有执行事务无法断定。

所以说 3PC 通过预提交阶段可以减少故障恢复时候的复杂性，但是不能保证数据一致，除非挂了的那个参与者恢复。

让我们总结一下， 3PC 相对于 2PC 做了一定的改进：引入了参与者超时机制，并且增加了预提交阶段使得故障恢复之后协调者的决策复杂度降低，但整体的交互过程更长了，性能有所下降，并且还是会存在数据不一致问题。

所以 2PC 和 3PC 都不能保证数据100%一致，因此一般都需要有定时扫描补偿机制。

我再说下 3PC 我没有找到具体的实现，所以我认为 3PC 只是纯的理论上的东西，而且可以看到相比于 2PC 它是做了一些努力但是效果甚微，所以只做了解即可。

TCC

2PC 和 3PC 都是数据库层面的，而 TCC 是业务层面的分布式事务，就像我前面说的分布式事务不仅仅包括数据库的操作，还包括发送短信等，这时候 TCC 就派上用场了！

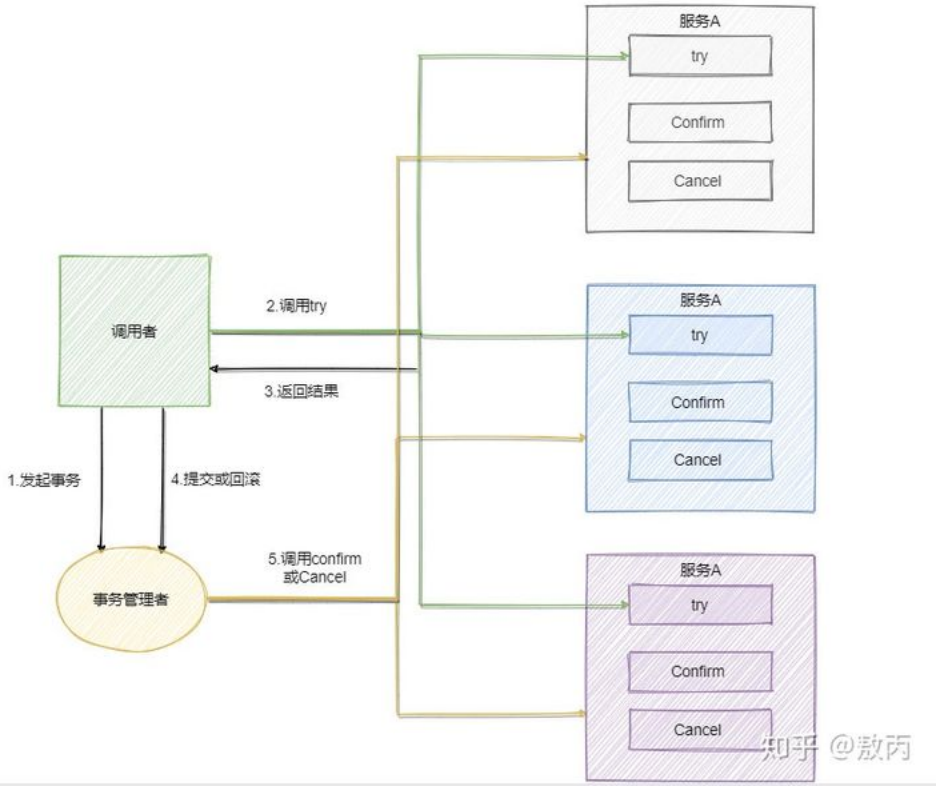
TCC 指的是 `Try - Confirm - Cancel` 。

- Try 指的是预留，即资源的预留和锁定，**注意是预留**。
- Confirm 指的是确认操作，这一步其实就是真正的执行了。
- Cancel 指的是撤销操作，可以理解为把预留阶段的动作撤销了。

其实从思想上看和 2PC 差不多，都是先试探性的执行，如果都可以那就真正的执行，如果不行就回滚。

比如说一个事务要执行A、B、C三个操作，那么先对三个操作执行预留动作。如果都预留成功了那么就执行确认操作，如果有一个预留失败那就都执行撤销动作。

我们来看下流程，TCC模型还有个事务管理者的角色，用来记录TCC全局事务状态并提交或者回滚事务。



可以看到流程还是很简单的，难点在于业务上的定义，对于每一个操作你都需要定义三个动作分别对应 `Try - Confirm - Cancel` 。

因此 **TCC 对业务的侵入较大和业务紧耦合**，需要根据特定的场景和业务逻辑来设计相应的操作。

还有一点要注意，撤销和确认操作的执行可能需要重试，因此还需要保证**操作的幂等**。

相对于 2PC、3PC ，TCC 适用的范围更大，但是开发量也更大，毕竟都在业务上实现，而且有时候你会发现这三个方法还真不好写。不过也因为是在业务上实现的，所以**TCC可以跨数据库、跨不同的业务系统来实现事务**。

本地消息表

本地消息表其实就是利用了 **各系统本地的事务**来实现分布式事务。

本地消息表顾名思义就是会有一张存放本地消息的表，一般都是放在数据库中，然后在执行业务的时候 **将业务的执行和将消息放入消息表中的操作放在同一个事务中**，这样就能保证消息放入本地表中业务肯定是执行成功的。

然后再去调用下一个操作，如果下一个操作调用成功了好说，消息表的消息状态可以直接改成已成功。

如果调用失败也没事，会有 **后台任务定时去读取本地消息表**，筛选出还未成功的消息再调用对应的服务，服务更新成功了再变更消息的状态。

这时候有可能消息对应的操作不成功，因此也需要重试，重试就得保证对应服务的方法是幂等的，而且一般重试会有最大次数，超过最大次数可以记录下报警让人工处理。

可以看到本地消息表其实实现的是**最终一致性**，容忍了数据暂时不一致的情况。

消息事务

RocketMQ 就很好的支持了消息事务，让我们来看一下如何通过消息实现事务。

第一步先给 Broker 发送事务消息即半消息，**半消息不是说一半消息，而是这个消息对消费者来说不可见**，然后**发送成功后发送方再执行本地事务**。

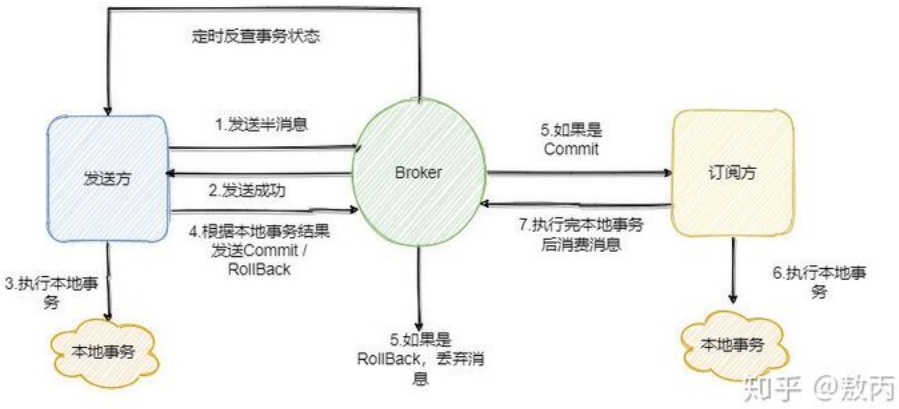
再根据**本地事务的结果向 Broker 发送 Commit 或者 RollBack 命令**。

并且 RocketMQ 的发送方会提供一个**反查事务状态接口**，如果一段时间内半消息没有收到任何操作请求，那么 Broker 会通过反查接口得知发送方事务是否执行成功，然后执行 Commit 或者 RollBack 命令。

如果是 Commit 那么订阅方就能收到这条消息，然后再做对应的操作，做完了之后再消费这条消息即可。

如果是 RollBack 那么订阅方收不到这条消息，等于事务就没执行过。

可以看到通过 RocketMQ 还是比较容易实现的，RocketMQ 提供了事务消息的功能，我们只需要定义好事务反查接口即可。



可以看到消息事务实现的也是最终一致性。

最大努力通知

其实我觉得本地消息表也可以算最大努力，事务消息也可以算最大努力。

就本地消息表来说会有后台任务定时去查看未完成的消息，然后去调用对应的服务，当一个消息多次调用都失败的时候可以记录下然后引入人工，或者直接舍弃。这其实算是最大努力了。

事务消息也是一样，当半消息被commit了之后确实就是普通消息了，如果订阅者一直不消费或者消费不了则会一直重试，到最后进入死信队列。其实这也算最大努力。

所以**最大努力通知其实只是表明了一种柔性事务的思想**：我已经尽力我最大的努力想达成事务的最终一致了。

适用于对时间不敏感的业务，例如短信通知。

总结

可以看出 2PC 和 3PC 是一种强一致性事务，不过还是有数据不一致，阻塞等风险，而且只能用在数据库层面。

而 TCC 是一种补偿性事务思想，适用的范围更广，在业务层面实现，因此对业务的侵入性较大，每一个操作都需要实现对应的三个方法。

本地消息、事务消息和最大努力通知其实都是最终一致性事务，因此适用于一些对时间不敏感的业