**常用的 分布式事务 都有哪些？我该用哪个？**

<https://mp.weixin.qq.com/s/IedGC4khj5Ym7AeUPu9ueQ>

分布式的CAP理论应该是人尽皆知了，它描述了一致性（C）、可用性（A）、分区容错性（P）的一系列权衡。很多时候，我们要在一致性和可用性之间权衡，而分布式事务，就是在这个大的前提下，尽可能的达成一致性的要求。

目标很小，问题很大，做法也各有不同。

“如何在微服务中实现分布式事务？”一般在被问到这样的问题时，我都会回答“要尽量避免使用分布式事务”，这也是Martin Fowler所推荐的。但现实总是残酷的，拆分了微服务之后，分布式事务是非常硬核的需求，是绕不开的，我们依然要想办法搞定它。但分布式环境错综复杂，还伴随着网络状况产生的超时，如何让事务达到一致性的状态，难度很大。

分布式事务，由一系列小的子事务组成。这些子事务，同大的分布式事务一样，同样要遵循ACID的原则。在一致性这个属性上，根据达到一致性之前所存在的时间，又分为强一致性和最终一致性（BASE）。

**注意，对于子事务，这里有个小小的误解。并不是只有和数据库打交道的操作，才叫做事务。在微服务环境下，如果你通过RPC调用了另外一个远程接口，并造成了相关数据状态的变化，这个RPC接口，也叫做事务。**

所以，在分布式事务中，我们把这些子事务涉及到的操作，叫做资源。当操作能正常完成的时候，根本不需要什么额外处理。事务主要处理的是发生异常之后的流程。

下面，我们就来看一下常见的分布式事务解决方案。

**1. 一阶段提交（1PC）**

先来看一下最简单的事务提交情况。

如果你的业务，只有一个资源需要协调，那么它可以直接提交。比如，你使用了一个数据库，那么就可以直接使用begin，commit等指令完成事务提交。

在Spring中，通过注解，就可以完成这样的事务。如果发生了嵌套事务，它的实现方式，本质上，是通过ThreadLocal向下传递的。所以如果你的应用中有子线程相关的事务需要管理，它办不到。

我们再来看分布式事务。所谓的分布式事务，就是协调2个或者多个资源，达到共同提交或者共同失败的效果，也就是分布式的ACID。

## 2. 两阶段提交（2PC）

在一阶段提交的概念扩展下，最简单的分布式事务解决方案，就是二阶段提交。二阶段提交不是指有两个参与资源，而是说有两个分布式的协调阶段，它可能有多个资源需要协调。

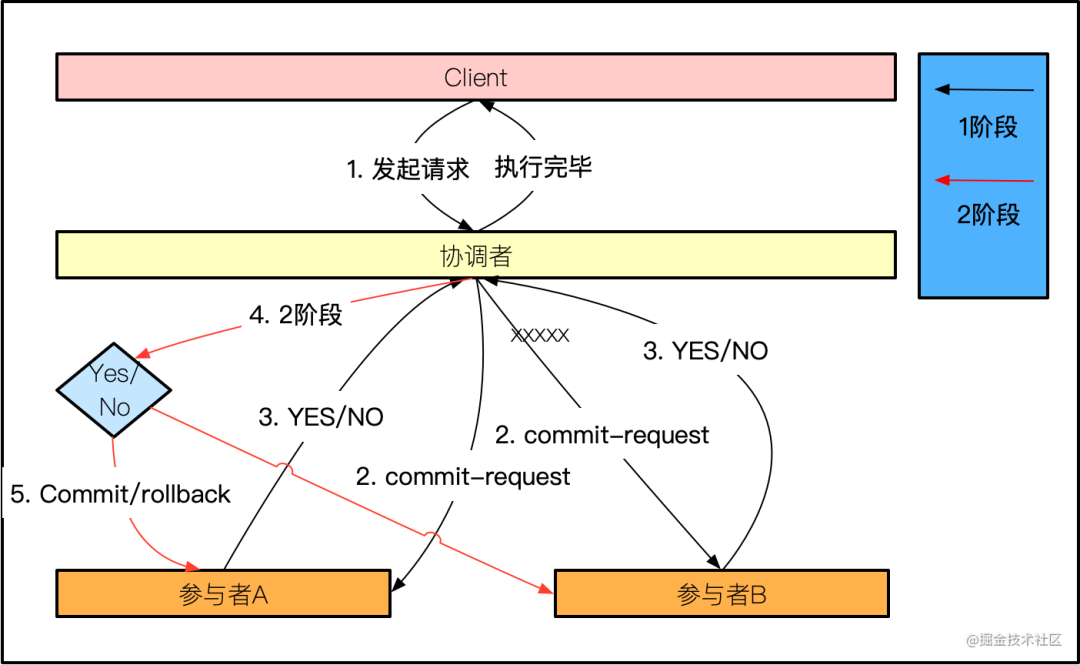
### 2.1 重要参与者

1. 协调者（coordinator），也就是我们需要自建事务管理器，通常在整个系统中只有一个
2. 事务参与者（participants），就是指的我们所说的资源，通常情况下会有多个，否则也称不上分布式事务了

### 2.2 过程

广义上的2PC（two phase commit），有哪两阶段呢？

1. client 分布式事务发起者
2. commit-request/voting  准备阶段
3. commit/rollback 提交或者回滚



准备阶段，也叫做voting阶段。所谓的voting，就是参与者告知协调者，自己的资源到底是能够提交（代表它准备好了），还是取消本次事务（比如发生异常）。

这个投票比较有意思，只要有一个参与者返回了false，本次事务就需要终止，然后执行rollback。只有全票通过，才会正常commit。协调者将这个结果，周知所有参与者的这个过程，就是二阶段。

二阶段提交其实非常容易理解。你可以把每个参与者的执行，想象成正常的SQL更新语句。它们一直挂在那里等待，直到协调者给出确切的commit或者rollback消息，才会正常往下执行。

### 2.3 问题

1. 阻塞问题。两阶段提交最大的问题，就是它是一个阻塞的协议，效率低。如果协调器永久失败，一些参与者，将永远无法完成它的事务
2. 单点故障问题。由于协调者在整个环节中有着非常重要的作用，所以一旦它发生了SPOF，整个系统将变的不可用，这是不能忍受的
3. 事务完整性问题。在某些情况下，比如协调者发送commit指令后，发生异常，有一部分执行成功了，会造成整个事务不一致。因为能不能提交，第一阶段就决定了，第二阶段只是通知而已，你就是死也要给我提交
4. 并不是所有的资源都支持2PC（或者XA）

对于第三点，我们举个例子。比如你的commit-request阶段全部返回了yes，然后协调者发送了commit指令。但这时候，有一台服务器A宕机了，无法执行这个commit。这时候，我们的client也会收到成功的消息。A机器重启之后，要有能力来恢复、继续执行commit指令，这些都是工程上必须要处理的。

### 2.4 框架

2PC也叫做XA事务，大多数数据库如MySQL，都支持XA协议。在Java中，JTA（不是什么JPA哦）是XA协议的实现。Spring也有JTA的事务管理器。

* Atomikos、bitronix实现了JTA，它们只需要提供jar包就可以了。实现了XA协议的数据库或者消息队列，已经能够具备了准备、提交、回滚的各种能力
* 使用在seata等框架，需要启动一个独立的seata服务协调者节点。seata使用的AT，借助于外部事务管理器，概念与XA类似

## 3. 三阶段提交（3PC）

相比较二阶段提交，三阶段提交最典型的特点是加入了超时机制。当然，3阶段证明了它有三个阶段，这个差别更显著。它本质上只是2PC的一些改进，所以身上完全充满了2PC的影子。

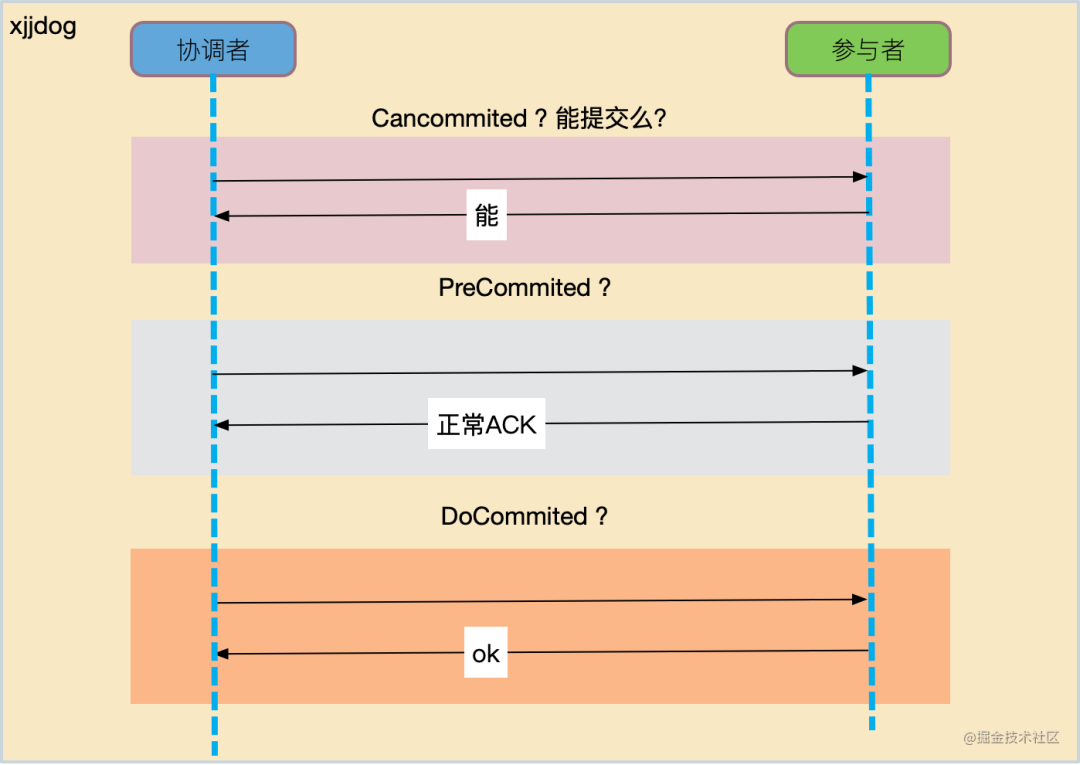
### 3.1 重要参与者

3PC和2PC是一样的。

### 3.2 过程

3PC比2PC多了一个步骤，那就是询问阶段。

1. CanCommit 询问阶段
2. PreCommit 准备阶段
3. DoCommit 提交阶段



提交阶段，无非就是发送个commit或者rollback指令，重要的处理还是在准备阶段，3PC把它一拆为2。

注意下面这个对应关系哦，2PC和3PC都有一个准备阶段，但它们的作用是不同的。

3PC     2PC  
CanCommit   commit-request/voting  
PreCommit   
DoCommit    commit

3PC的询问阶段，对应的才是2PC的准备阶段，都是ask一下参与者是否准备好了，但执行过程会有一些区别。

为什么要这么做？因为2PC有效率问题。2PC的执行过程是阻塞的，一个资源在进入准备阶段之后，必须等待所有的资源准备完毕才能进行下一步，在这个过程中，它们对全局一无所知。

比如，有ABCDE等5个参与者，E其实是一个有问题的参与者资源。但2PC每次都会执行ABCD的预提交，当询问到E的时候，发现是有问题的，再依次执行ABCD等参与者的rollback。在这种情况下，ABCD执行了无用的事务预处理和rollback，是非常浪费资源的。

3PC通过拆分这个询问阶段，在确保所有参与者建康良好的情况下，才会发起真正的事务处理，在效率和容错性上更胜一筹。从概率上来讲，由于commit之前粒度变小了，commit阶段出问题的几率就变小，能省下不少事。

另外，3PC引入了超时机制。在PreCommit阶段，如果超时，就认为失败；而在DoCommit阶段，如果超时还会继续执行下去。但不论怎样，整个事务并不会一直等待下去。

### 3.3 问题

3PC理论上是比较优秀的，还能够避免阻塞问题，但它多了一次网络通信。如果参与者的数量比较多，网络质量比较差的情况下，这个开销非常可观。它的实现也比较复杂，在实际应用中，是不太多的。

3PC也并不是完美的，因为PreCommit阶段和DoCommit也并不是原子的，和2PC类似，依然存在一致性问题。

## 4. TCC

TCC是柔性事务，而上面介绍的都是刚性事务。有时候，一个技术问题，可以通过业务建模来实现。

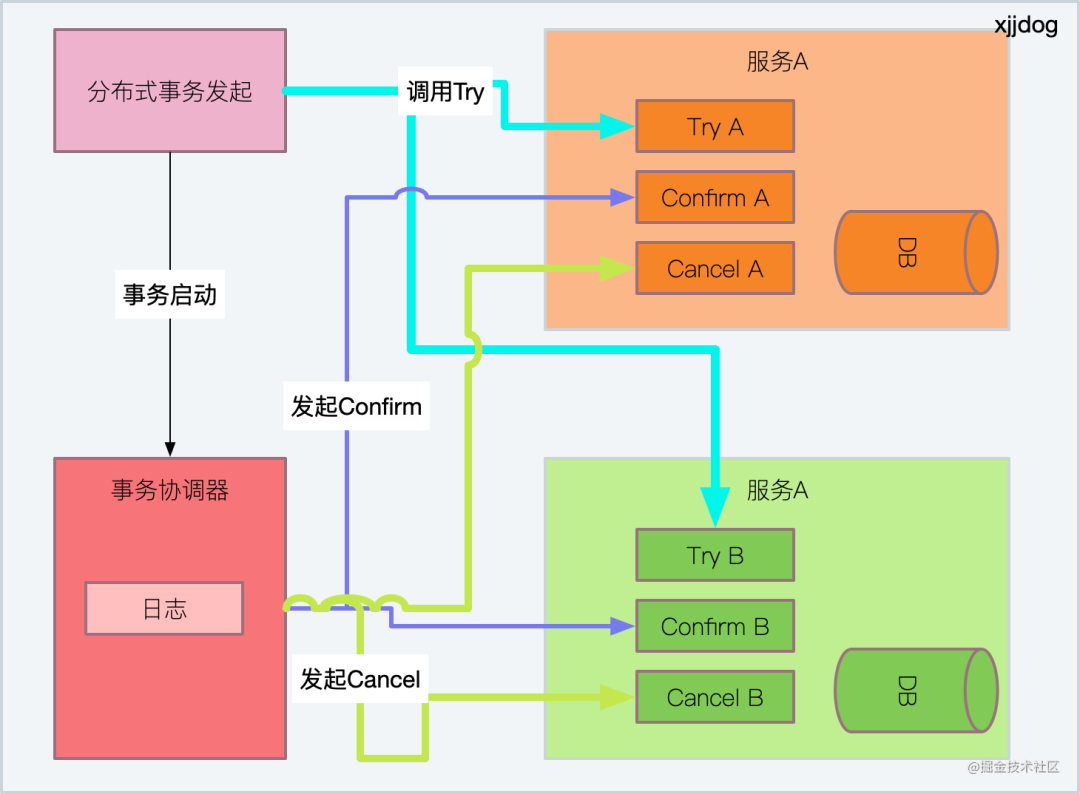
2PC和3PC在概念上看起来虽然简单，但放在分布式环境中，考虑各种超时和宕机问题，如果考虑的周全，那可真是要了老命。

2PC的框架还是比较多的，但3PC全网找了个遍，发现有名的实现几乎没有。

不要伤心，我们有更容易理解，更加直观的分布式事务。那就是TCC，2007年的老古董。

TCC就是大名鼎鼎的补偿事务，是互联网环境最常用的分布式事务。它的核心思想是：为每一个操作，都准备一个确认动作和相应的补偿动作，一共3个方法。

与其靠数据库，不如靠自己的代码！2PC，3PC，都和数据库绑的死死的，TCC才是码农的最爱（意思就是说，你要多写代码）。



如图，TCC同样分为三个阶段，但非常的粗暴！

* try 尝试阶段 尝试锁定资源
* confirm 确认阶段 尝试将锁定的资源进行提交
* cancel 取消阶段 其中某个环节执行失败，将发起事务取消动作

看起来这三个阶段，是2阶段提交的一种？完全不是。但它们的过程可以比较一下。

TCC     2PC  
Try     业务逻辑  
Confirm    commit-request/voting + commit  
Cancel     rollback

从上面可以看出来，2PC是一种对事务过程的划分，而TCC是对正常情况的提交和异常情况的补偿。相对于传统的代码，try和confirm两者加起来，才是真正的业务逻辑。

TCC是非常容易理解的，但它有一个大的前提，就是这三个动作必须都是幂等的，对业务有一定的要求。拿资金转账来说，try就是冻结金额；confirm就是完成扣减；cancel就是解冻，只要对应的订单号是一致的，多次执行也不会有任何问题。

由于TCC事务的发起方，直接在业务节点即可完成，和TCC的代码在同一个地方。所以，TCC并不需要一个额外的协调者和事务处理器，它存放在本地表或者资源中即可。

是的，它也要记录一些信息，哪怕是HashMap里，否则它根据啥回滚呢？

### 4.1 问题

TCC事务，需要较多的编码，以及正确的try和confirm划分。由于没有中心协调器，不需要阻塞，TCC的并发量较高，被互联网业务广泛应用。

团队要有能力设计TCC接口，将其拆分成正确的Try和Confirm阶段，实现业务逻辑的分级。

### 4.2 框架

ByteTCC、tcc-transaction、seata等。

## 5. SAGA

SAGA也是一个柔性事务。

saga的历史更久远，要追溯到1987年的一篇论文，可以说是瓶旧酒。它主要处理的是长活事务，但它不保证ACID，只保证最终一致性。

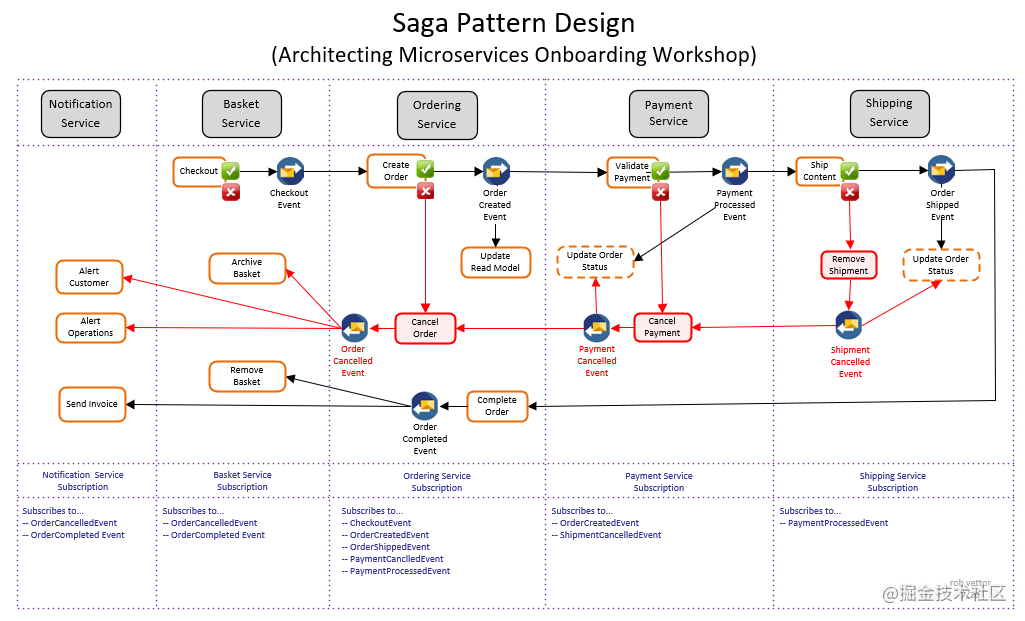
所谓长活事务，可以被分解成交错运行的子事务，它通过消息，来协调一系列的本地子事务，来达到最终的一致性。

我们可以把SAGA编排器，想象成一个状态机。每当处理完一条消息，它就能够知道要执行的下一条消息（子事务）。

比如，我们把事务T，拆分成了T1，T2，T3，T4。那么我们就必须为这些子事务，提供相应的执行逻辑和补偿逻辑。没错，和TCC一样，不过比TCC少了一步Try动作，同样要求这些操作是幂等的。

你瞧瞧，其实SAGA的概念很好理解，你就按照正常的业务逻辑去执行就行了。只不过如果在任何一步发生了异常，就要把前面所提交的数据全部回滚（补偿）。唯一特殊的是，它通常是通过消息驱动来完成事务运转的。

如果你非要追求它的本质，那就是SAGA和TCC一样，都是先记录执行轨迹，然后通过不断地重试达到最终状态。



上图是rob vettor所绘制的一个典型的SAGA事务拆分图。在图中，黑色的线为正常业务流程，红色的线为补偿业务流程。这是一个简单的电子商务结账流程，整个交易跨了5个微服务，可以说是非常大的长事务了。

可以看到，这样的事务流转，靠文字描述已经是不好理解了，所以SAGA通常会配备一个流程编辑器，直接来把事务编排的过程可视化。

### 5.1 问题

那问题就有意思多了。

1. 嵌套问题。SAGA只允许两层嵌套，因为靠消息流转本来就非常复杂了，嵌套层次深在性能和时序上都不允许
2. 如果你的事务包含很多子事务，那么很有可能在某个阶段就执行失败了。但如果补偿操作也发生问题了呢？极端情况下，需要人工参与。在很多时候，需要记录日志（saga log）来配合完成
3. 由于这些小事务并不是同时提交的，所以在执行的过程中，会产生脏数据，这和数据库的read uncommited的概念是一样的

### 5.2 框架

在《微服务架构设计模式》的第四章中，说明了SAGA的具体使用示例，现在网络上的大多数文章都来自于此。但据我所知，使用SAGA的互联网公司并不是很多，倒是使用TCC的比较多一些（可能是遇到的分布式事务都不是长事务）。

seata同样提供了SAGA的方式，主要使用的是状态机驱动的编排模式。为了支持事务的编排，seata提供了一个专用的流程编辑器（在线）。

http:*//seata.io/saga\_designer/index.html*

设计完毕之后，就可以导出为JSON文件，解析之后可以写入到数据库中。

bytetcc虽然叫tcc，它也支持SAGA。

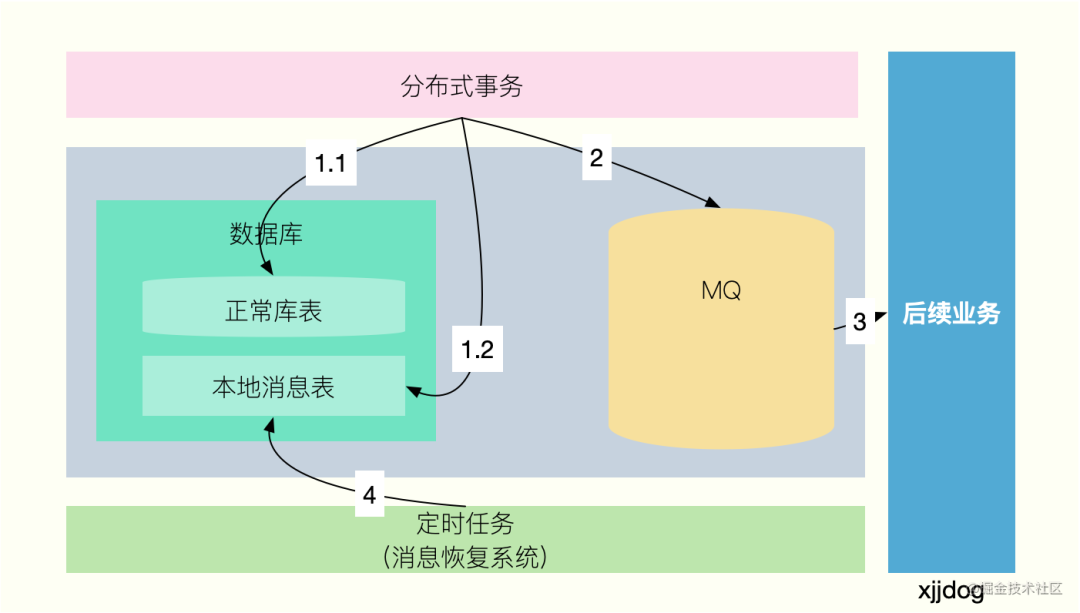
### 5.3 SAGA vs TCC

上面也提到，我在平常工作中，用到TCC比SAGA更多一些，也是由于业务场景确定的。下面简单的对比一下。

1. 开发难度。TCC的开发难度是比SAGA要高的，因为它需要处理Try阶段来冻结资源，而SAGA是直接执行本地事务
2. 脏读问题。TCC不存在脏读，因为try阶段并不影响数据；SAGA会在小事务之间，或者cancel之间出现脏读
3. 效率问题。TCC无论成功失败，都需要和参与方交互两次；SAGA在正常情况下交互一次，异常情况下交互两次，所以效率要高
4. 业务流程。TCC适合少量的分布式事务流程，否则写起来就是噩梦；SAGA适合业务流程长，参与方多的业务，或者遗留系统等无法改造成TCC的业务
5. 手段。TCC是通过业务建模手段解决技术问题；SAGA是通过技术手段解决事务编排

## 6. 本地消息表

本地消息表的使用场景比较局限，它要靠MQ去实现，它解决的是数据库事务和MQ之间的事务问题。



如图，有一个分布式事务，在正常落库之后，需要通过MQ来协调后续业务的执行。但是，写DB和写MQ，是无法达成一致性的，就需要加入一个本地消息表来缓存发送到MQ的状态。下面我来描述一下这个过程。

* 1.1 正常写入数据库
* 1.2 在写入数据库的同时，写入一张本地消息表。这张表，用来记录MQ消息处理的状态，可以有发送中和已完成两种状态。由于消息表和正常的业务表在一个DB中，所以可以达成本地事务，确保同时完成
* 2 写入消息表成功之后，可以异步发送MQ消息，且不用关心投递是否成功
* 3 后续业务订阅MQ消息。消费成功之后，将会把执行成功的状态，再通过MQ来发送。本地业务订阅这个执行状态，并把消息表中对应的记录状态，改为已完成；如果消费失败，则不做过多处理
* 4 存在一个定时任务，持续扫描本地消息表中，状态为发送中的消息（注意延时），并再次把这些消息发送到MQ，重复2的过程

通过这样的循环，就可以达到本地DB和MQ消费者状态的一致性，完成最终一致性的分布式事务。

可以看到，我们有重发MQ的过程，所以这种模式要求消费者也要实现幂等的功能，避免重复对业务产生影响。

### 6.1 问题

使用本地消息表方案的系统还是挺多的，但它的弊端也显而易见。

1. 需要开发专用的代码，与业务耦合在一起，无法完成抽象的框架
2. 本地消息表需要写数据库，如果数据库本身的I/O已经比较高了，它会增加数据库的压力

## 7. 最大努力补偿

最大努力补偿，是一种衰减式的补偿机制。

拿个最简单的例子来说吧。如果你是微信支付的接入方，微信支付成功之后，它会将支付结果推送到你指定的接口。

微信支付+你的支付结果处理，就可以算是一个大的分布式事务。涉及到微信的系统还有你的自有系统。

如果你的系统一直处理不成功，那么微信支付就会一直不停的重试。这就叫最大努力补偿，用在系统内和系统间都是可以的。

但也不能无限的重试，重试的间隔通常会随着时间衰减。常用的衰减策略有。

messageDelayLevel = 1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h

上面的公式，意味着如果一直无法处理成功，将在1s...，最大2小时后重试。如果还不成功，就只能进入人工处理通道。

最大努力补偿只是一种思想，实际的应用有多种方式。比如，我首先将事务落地到消息队列，然后依靠消息队列的重试机制，来达到最大努力补偿的效果，这些都是可行的方案。

## 8. 总结

我们在文中，从本地事务谈起，分别聊到了2PC、3PC、TCC、SAGA、本地消息表、最大努力补偿等，也了解到了各种解决方案的一些应用场景和解决方式。

分布式事务框架，在这些理论基础上，都进行了或多或少的修订，也有不少创新。比如LCN框架（lock，confirm，notify），就抽象出了控制方和发起方的概念，感兴趣的可以自行了解。

在互联网公司中，由于高并发量的诉求，在实际应用中，相对于强事务，大家普遍选用软事务进行业务处理。使用最多的，就是TCC、SAGA、本地消息表等解决方案。SAGA应对长事务特别拿手，但隔离性稍差；TCC一直性好并发高，但需要较多编码；本地消息表应用场景有限，耦合业务不能复用。各种解决方案都有它的利弊，一定要结合使用场景进行选择。

在框架方面，阿里的seata（早些年叫fescar），已经得到了广泛应用，XA、TCC、SAGA等模式都支持，如果你需要这方面的功能，可以集成尝试一下。