目录

[一、 CAP理论 2](#_Toc28100)

[二、 分布式事务解决方案 12](#_Toc20118)

### CAP理论

1. CAP 的由来

要理解 CAP，首先我们要清楚，为何会有人提出 CAP？他提出 CAP 是为了解决什么问题？

时间回到 1985 年，彼时，后来证明了 CAP 理论的 Lynch 教授此时给当时的 IT 界来了一记惊雷：

她通过不可辩驳的证明告诉业界的工程师们，如果在一个不稳定（消息要么乱序要么丢了）的网络环境里（分布式异步模型），想始终保持数据一致是不可能的。

这是个什么概念呢？就是她打破了那些既想提供超高质量服务，又想提供超高性能服务的技术人员的幻想。

这本质是在告诉大家，在分布式系统里，需要妥协。

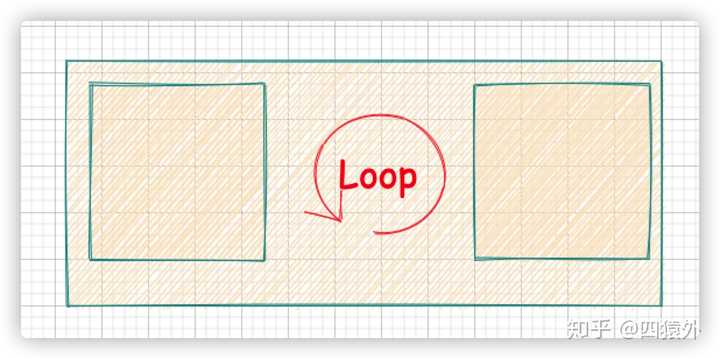
但是，如何妥协？分布式系统里到底应该怎么权衡这种 trade-off？

我们可以想象一下，在 CAP 定理提出之前，没有这些方向性的指引，在设计和实施分布式系统时该有多么混乱。一套分布式系统是由多个模块组成的，这些模块本身可能由不同的开发人员去完成。然而，对于这些人，在公共层面，竟然没有一个原则去指导他们该怎么完成这套功能。

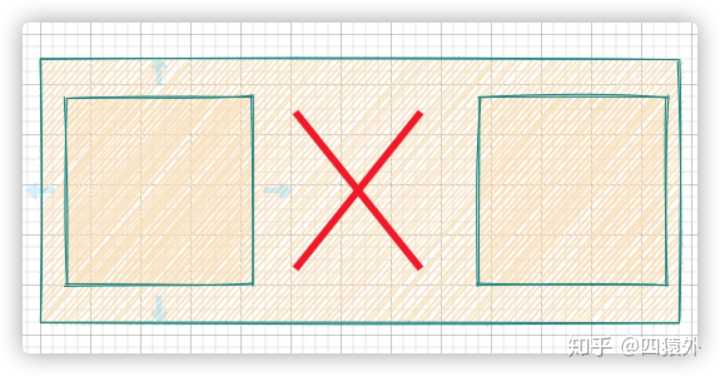
比如，我们在同步两个节点的数据时，如果发生了错误，到底我们应该怎么做呢？如果没有统一的标准和方向，那很可能在一套分布式系统中的不同模块，会出现不同的处理情况。

假设一套系统，由 A、B 两个模块构成。

A 模块的设计理念是：节点间出现了问题，它可能会选择不断的重试，一直等到节点通信恢复。



而 B 的设计理念是：节点间出现了问题，它断开就是了，可能最多就记录下状态，等以后处理。



可是，当 A、B 之间出现了通信怎么办？那会出现 A 往 B 发请求，出问题会不断重试。而 B 往 A 发请求，出问题则直接断开的情况。

当然，在后面我们会说明，CAP 的理念在实际工程中，会允许这种不一致。可是，那种不一致是提前设计好和规划好的，是根据实际数据的重要性和业务需求做的妥协，而不是这种混乱的妥协。

所以，IT 界的人们就一直在摸索，试图找到一些纲领去指导分布式系统的设计，这一找就找了 15 年。

2000 年时，Eric Brewer 教授在 PODC 会议上提出了 CAP 理论，但是由于没有被证明过，所以，当时只能被称为 CAP 猜想。这个猜想引起了巨大的反响，因为 CAP 很符合人们对设计纲领的预期。

在 2002 年后，经过 Seth Gilbert 和 Nancy Lynch 从理论上证明了 CAP 猜想后，CAP 理论正式成为了分布式系统理论的基石之一。

2. CAP 到底是什么

CAP 定理表达了一个分布式系统里不可能同时满足以下的三个特性：

2.1. C：数据一致性

什么是数据一致性？咋一看真的很让人糊涂，一致性是什么？是指数据能一起变化，是能让数据整齐划一。

那么问题又来了，数据何时会变化？数据怎么才能被称为一起变化？我们现在来回答这些问题，当我们搞清楚了这些问题，那么对数据一致性就会有了清晰的理解。

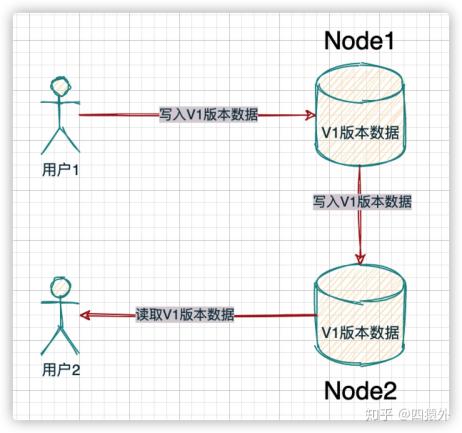
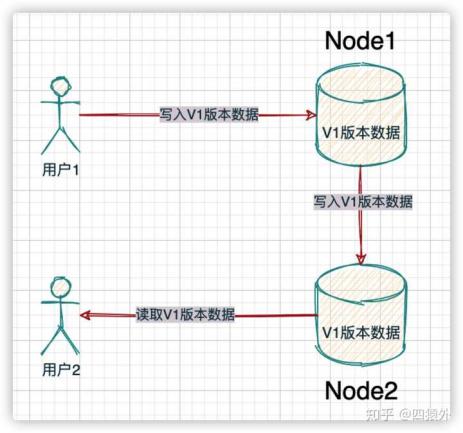
首先第一个问题，数据何时会一起变化？

答案是：仅且仅当包含数据的服务，收到数据更新请求的时候，数据才会发生变化。而数据更新请求则仅包括数据的增、删、改这三种请求，而这三种请求又被统称为写请求。所以，数据只有在写请求的时候才会发生变化。

那我们来回答第二个问题，数据要怎么样才能被称为一起变化了？即谁来判断数据是最终变化了？是服务器对写请求的返回结果吗？告诉写请求成功，数据就一定发生一致性变化了？

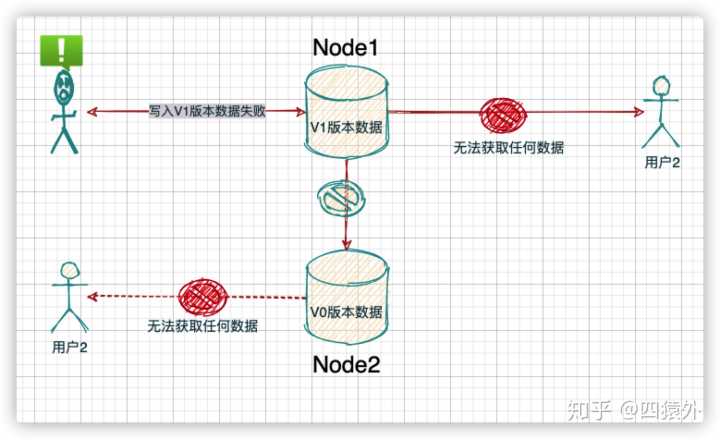
NO，数据发生变化是否一致是需要经过读请求来做检验的。那么读请求判断的依据是什么呢？

假设，我们的分布式存储系统有两个节点，每个节点都包含了一部分需要被变化的数据。如果经过一次写请求后，两个节点都发生了数据变化。然后，读请求把这些变化后的数据都读取到了，我们就把这次数据修改称为数据发生了一致性变化。



但是，这还不是完整的一致性。因为系统不可能永久的正常运行下去。

如果系统内部发生了问题从而导致系统的节点无法发生一致性变化会怎么样呢？当我们这样做的时候，就意味着想看到最新数据的读请求们，很可能会看到旧数据，或者说获取到不同版本的数据。此时，为了保证分布式系统对外的数据一致性，于是选择不返回任何数据。



这里需要注意一下，CAP 定理是在说在某种状态下的选择，和实际工程的理论是有差别的。上面描述的一致性和 ACID 事务中的一致性是两回事。事务中的一致性包含了实际工程对状态的后续处理。但是 CAP 定理并不涉及到状态的后续处理，对于这些问题，后续出现了 BASE 理论等工程结论去处理，目前，只需要明白 CAP 定理主要描述的是状态。

2.2. A：可用性

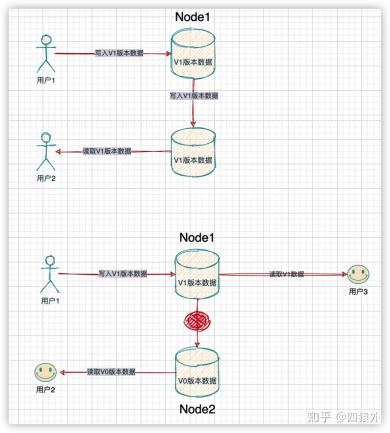
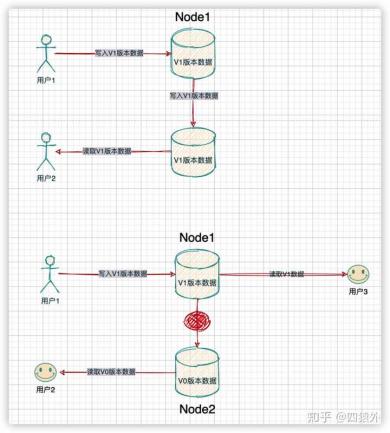
奥维德曾经说过：“行动被人们遗忘，结果却将永存”。

这句话说明了结果的重要性，而可用性在 CAP 里就是对结果的要求。它要求系统内的节点们接收到了无论是写请求还是读请求，都要能处理并给回响应结果。只是它有两点必须满足的条件：

条件 1：返回结果必须在合理的时间以内，这个合理的时间是根据业务来定的。业务说必须 100 毫秒内返回，合理的时间就是 100 毫秒，需要 1 秒内返回，那就是 1 秒，如果业务定的 100 毫秒，结果却在 1 秒才返回，那么这个系统就不满足可用性。

条件 2：需要系统内能正常接收请求的所有节点都返回结果。这包含了两重含义：

1. 如果节点不能正常接收请求了，比如宕机了，系统崩溃了，而其他节点依然能正常接收请求，那么，我们说系统依然是可用的，也就是说，部分宕机没事儿，不影响可用性指标。
2. 如果节点能正常接收请求，但是发现节点内部数据有问题，那么也必须返回结果，哪怕返回的结果是有问题的。比如，系统有两个节点，其中有一个节点数据是三天前的，另一个节点是两分钟前的，如果，一个读请求跑到了包含了三天前数据的那个节点上，抱歉，这个节点不能拒绝，必须返回这个三天前的数据，即使它可能不太合理。



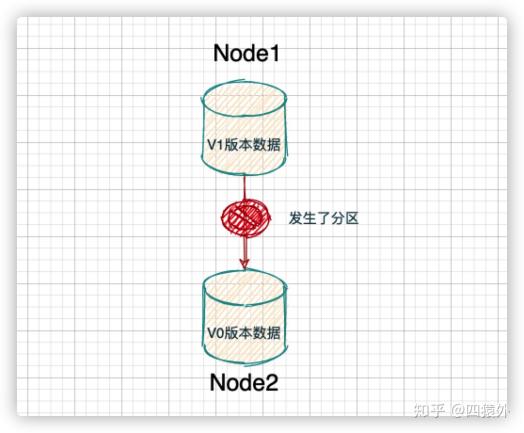
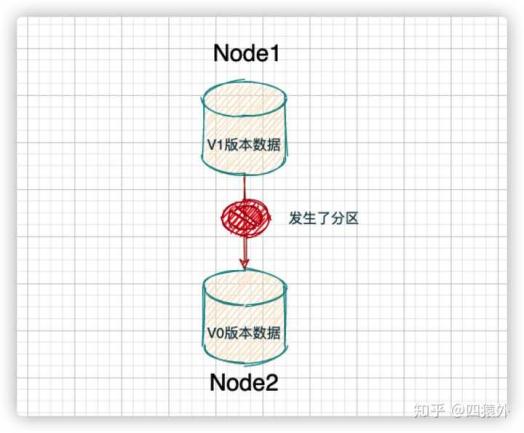
2.3. P：分区容忍性

分布式的存储系统会有很多的节点，这些节点都是通过网络进行通信。而网络是不可靠的，当节点和节点之间的通信出现了问题，此时，就称当前的分布式存储系统出现了分区。但是，值得一提的是，分区并不一定是由网络故障引起的，也可能是因为机器故障。

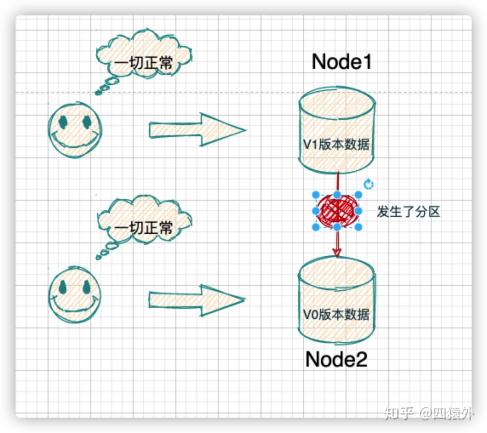
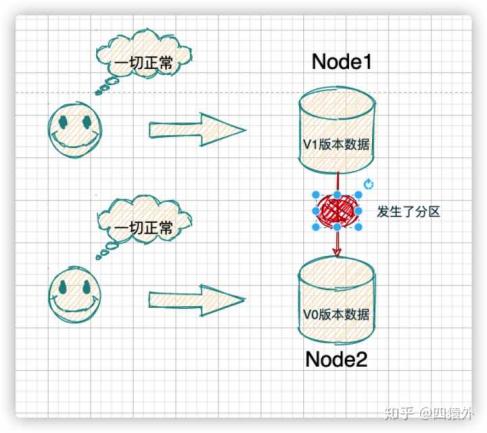
比如，我们的分布式存储系统有 A、B 两个节点。那么，当 A、B 之间由于可能路由器、交换机等底层网络设备出现了故障，A 和 B 通信出现了问题，但是 A、B 依然都在运行，都在对外提供服务。这时候，就说 A 和 B 发生了分区。

还有一种情况也会发生分区，当 A 出现了宕机，A 和 B 节点之间通信也是出现了问题，那么我们也称 A 和 B 发生了分区。

综上，我们可以知道，只要在分布式系统中，节点通信出现了问题，那么就出现了分区。



那么，分区容忍性是指什么？ 它是说，如果出现了分区问题，我们的分布式存储系统还需要继续运行。不能因为出现了分区问题，整个分布式节点全部就熄火了，罢工了，不做事情了。



3. CAP 怎么选择

我们上面已经知道了，在设计分布式系统时，架构师们在 C、A、P 这三种特性里，只能选择两种。

但是，这道 CAP 的选择题，就像别人在问你“小明的父亲有三个孩子，老大叫大朗，老二叫二郎，请问老三叫什么”一样。在以分布式存系统为限定条件的 CAP 世界里，P 是早已经确定的答案，P 是必须的。

因为，在分布式系统内，P 是必然的发生的，不选 P，一旦发生分区错误，整个分布式系统就完全无法使用了，这是不符合实际需要的。所以，对于分布式系统，我们只能能考虑当发生分区错误时，如何选择一致性和可用性。

而根据一致性和可用性的选择不同，开源的分布式系统往往又被分为 CP 系统和 AP 系统。

当一套系统在发生分区故障后，客户端的任何请求都被卡死或者超时，但是，系统的每个节点总是会返回一致的数据，则这套系统就是 CP 系统，经典的比如 Zookeeper。

如果一套系统发生分区故障后，客户端依然可以访问系统，但是获取的数据有的是新的数据，有的还是老数据，那么这套系统就是 AP 系统，经典的比如 Eureka。

说了这么多，其实 CAP 定理本质很简单，它就是一种分布式系统设计的不同理念概括，包括它说的一致性，可用性和分区容错性。这就类似一个大学的校训，是极度概念化的东西。

所以，大白话来形容下 CAP 吧，CAP 就是告诉程序员们当分布式系统出现内部问题了，你要做两种选择：

* 要么迁就外部服务，像外包公司。
* 要么让外部服务迁就你，像银行。

迁就外部服务就是我们不能因为我们自己的问题让外部服务的业务运行受到影响，所以要优先可用性。而让外部服务迁就我们，就要优先一致性。

4. 对 CAP 的常见误解

误解一：分布式系统因为 CAP 定理放弃了 C 或者 A 中的其中一个

很多人在没有对 CAP 做深入了解的情况下，听到很多人说分布式系统必须在 CAP 三个特性里选择两个，就觉得一套分布式系统肯定要么只有可用性要么只有一致性，不存在完整的可用性和一致性功能。

这种理解是大有问题的。因为，P 这种问题发生的概率非常低，所以：

当没有出现分区问题的时候，系统就应该有完美的数据一致性和可用性。

你什么时候见过一个系统，当内部没有问题的时候，会经常让外部请求卡一下的？要么就冷不丁的提供陈旧的老数据？那还能叫系统吗？

误解二：C 和 A 之间的选择是针对整个分布式系统的，只能整体考虑 C 和 A 之间的选择

这个理解也是不对的。当分区发生的时候，其实对一致性和可用性的抉择是局部性的，而不是针对整个系统的。

可能是在一些子系统做一些抉择，甚至很可能只需要对某个事件或者数据，做一致性和可用性的抉择而已。

比如，当我们做一套支付系统的时候，会员的财务相关像账户余额，账务流水是必须强一致性的。这时候，你就要考虑选 C。但是，会员的名字，会员的支付设置就不必考虑强一致性，可以选择可用性 A。

一套分布式系统的运行，就像人生一样，就是一次又一次的选择。在不同阶段，不同的时刻有不同的事件发生的时候，又怎么可能会有完全一样的选择呢？

误解三：CAP 的三个特性只有是和否两种极端选择，而不是一个范围

这种二元性的理解更是极其误导人。

CAP 理论的三种特性不是 Boolean 类型的，不是一致和不一致，可用和不可用，分区和没分区的这类二选一的选项。而是这三种特性都是范围类型。

拿可用性来说，就像我从银行取钱。当我目的是派发压岁钱的时候，我很可能就想全要新票子，但是，新票子很可能就还得多一个步骤，就是需要拿旧票子去换一些新票，此时，我可以多等会儿，能拿到新票子就好。而当我的目的就是做生活花销的时候，票子是新是旧，我根本不那么关心，快点拿到钱就行。这就是可用性的范围需求之一，对时延性的要求。

再比如，分区容错则由于探测机制的问题，可能还得各节点搞投票去协商分区是否存在，当某一台机器出现了问题，可能不影响业务的话，就会被机器投票认为分区不存在。然后一直等到多数机器出现了问题，才会投票确认出现了分区问题。这就好像新冠疫情，还会分低、中、高风险区呢，不是一出现通信故障就都被逻辑认定为分区问题。

作者：四猿外  
链接：https://www.zhihu.com/question/54105974/answer/1643846752  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

5. CAP 理论的一些疑问

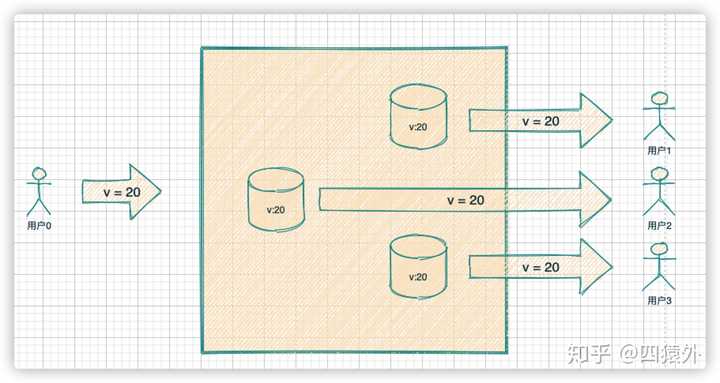
疑问一：在遵从 CAP 定理的系统中是否适合任意的写请求

首先，在 CAP 定理中，关于一致性会有多种说法，但是总的来说，都是在描述数据最新版本的可见性。而这些可见性往往代表的是读请求返回的数据的可见性。

那么问题来了，当我们要求读数据的可见性的时候，对写数据有什么要求吗？

比如，我们系统有三个节点，一个客户端给这个系统发了一个写请求，要求系统写入一个值为 20 的数据。那么，如果要满足 CAP 定理中的一致性，就需要在写完 20 这个数据之后，当其他客户端请求读取这个值为 20 的数据之后，无论请求被转发到系统中任何节点都能返回这个值。

这就要求写入这个值为 20 的写请求必须成功写到三个节点上，此时，系统就满足了写一致性的。所以，我们可以说对于读一致性的要求是同时约束了写一致性的。

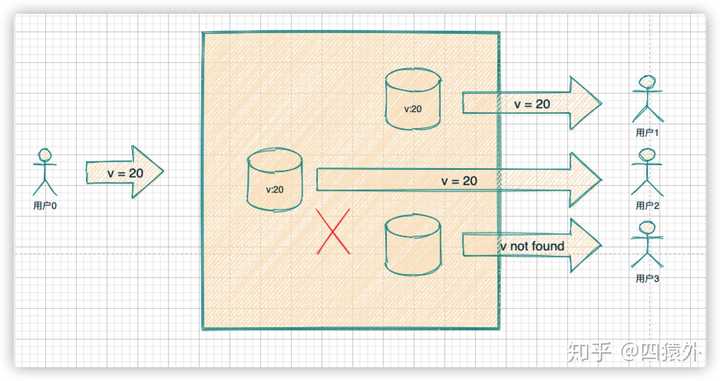


其次，在 CAP 定理中，可用性本身要求对读、写请求都要处理。如果我们以可用性作为标准的时候，在发生分区错误时，由于我们对读请求并没有强行要求返回完全准确的数据，所以，可能在本次读请求之前的最近一次写请求可能是部分失败的。

同样的例子，我们的分布式系统由三个节点组成，最近一次写请求想把值为 20 的数据写到三个节点上。但是，由于发生了分区问题，有一个节点通信故障，写请求写不过去，因此只有两个节点包含了值为 20 的数据。

此时，写请求会返回给客户端一个结果，可能会告诉客户端写入成功了，也可能告诉客户端写入部分成功。

这时候，当后续的读请求恰巧被发送到有通信故障的那个节点，系统可能只能返回一个空的结果。但是，由于系统处理和返回了读写请求，所以，系统是满足了 CAP 中的可用性的。



疑问二：数据分片和数据副本的分布式系统是否都遵守 CAP 定理

我们知道，在一套大规模的分布式系统里，一定是既需要把海量数据做切分，存储到不同的机器上，也需要对这些存储了数据的机器做副本备份的。

那么，如果，一个分布式系统里只有数据分片存储或者只有数据副本存储，他们都会遵守 CAP 定理吗？

答案是当数据分片时，也是要遵守 CAP 定理，但是，是种非常特殊的遵守。

当在一套分布式系统只有分片存储的时候，CAP 理论会表现成什么样？

比如，我们有个分布式系统，由三个节点 a、b、c 组成。其中节点 a 存放了 A 表的数据，b 存放了 B 表的数据，c 存放了 C 表的数据。

如果有一个业务，它的意图是想往 A 表插入一条新数据，在 B 表删除一条已有数据，在 C 表更新一条老数据，这个分布式系统该怎么处理这种业务？

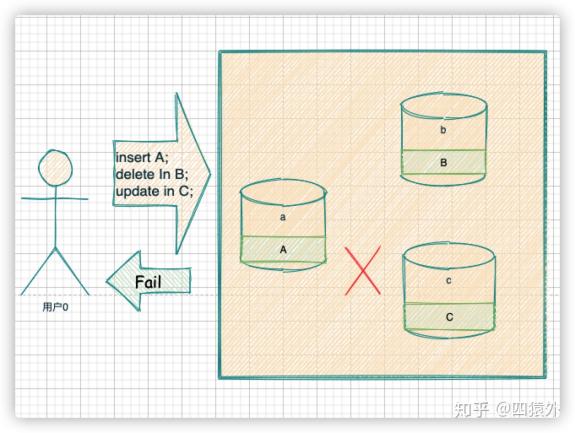
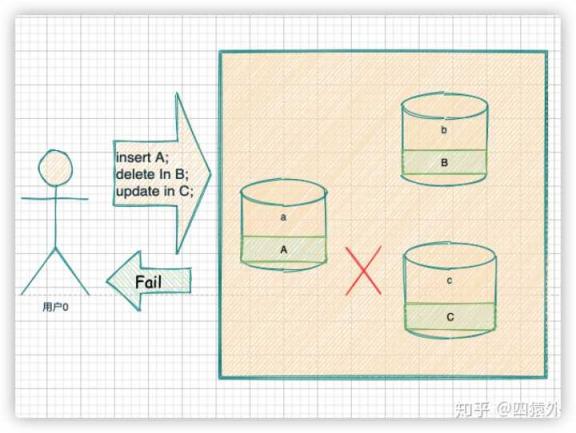
技术上我们对这种一个意图想做多件事的情况往往会包装成一个事务。当我们包装成一个事务以后，我们可能会通过先在 a 节点执行，然后去 b 节点执行，最后去 c 节点执行，等到都成功了，才会返回成功。

但是，发生了分区以后怎么办？当在 a、b 节点都成功了，到 c 发现发生了通信故障？

此时，根据 CAP 定理，你有两个选择，要么就直接返回一个部分成功的结果给客户端，要么直接卡死等客户端超时或者返回失败给客户端。当返回部分成功的时候，这就是选择了可用性（A），当卡死或者返回失败给客户端的时候，就是选择了一致性（C）。

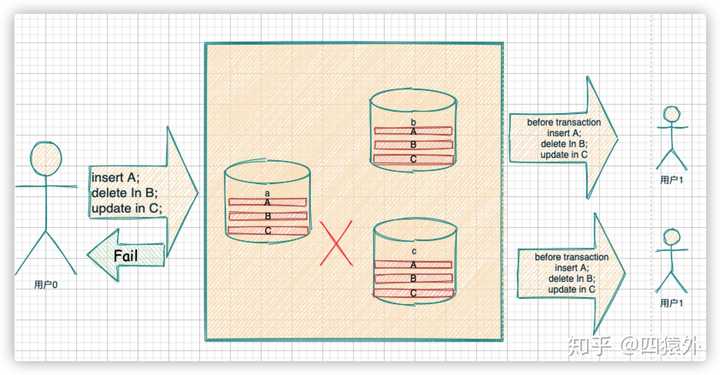
可是，我们将请求包装成了事务，而事务是要求要么都成功，要么都失败……为了遵守这种要求，对于分布式只有分片的情况，迫于客观条件，只能选择C。所以分片的分布式系统，往往都是 CP 的系统。

可选择，但是无法选择是分布式系统只有分片数据存储的情况时，遵守 CAP 定理的特殊表现。

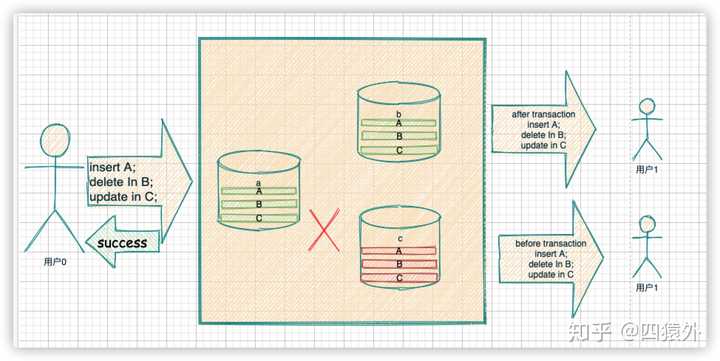


而当分布式系统是多个节点，每个节点存储了完整的一套数据，别的节点只是完整数据的备份的时候，即使事务只在一台机器上成功，当发生分区故障的时候，我们也是可以有充分的余地选择是**单机事务的回退** or **就此认为写成功的**。

单机事务的回退，就可以对外表现为选择了一致性。



就此认为写成功，则可以认为选择了可用性。



疑问三：为何有时候区分一个系统是 AP 还是 CP 是如此之难

因为，就像我们前面讲过的，由于 AP 或者 CP 的选择，可能仅局限为整套系统的局部，甚至某些特殊的数据上，而我们又是用这种局部的特性去描述了整套系统，所以就导致了区分的困难。而这本身其实也日渐成为了 CAP 的一个大问题，从而被人诟病。

6. CAP 的不足

1. CAP 定理本身是没有考虑网络延迟的问题的，它认为一致性是立即生效的，但是，要保持一致性，是需要时间成本的，这就导致往往分布式系统多选择 AP 方式
2. 由于时代的演变，CAP 定理在针对所有分布式系统的时候，出现了一些力不从心的情况，导致很多时候它自己会把以前很严谨的数学定义改成了比较松弛的业务定义，类似于我们看到，CAP 定理把一致性、可用性、分区容错都变成了一个范围属性，而这和 CAP 定理本身这种数学定理般的称呼是有冲突的，出现了不符合数学严谨定义的问题。
3. 在实践中以及后来 CAP 定理的提出者也承认，一致性和可用性并不仅仅是二选一的问题，只是一些重要性的区别，当强调一致性的时候，并不表示可用性是完全不可用的状态。比如，Zookeeper 只是在 master 出现问题的时候，才可能出现几十秒的不可用状态，而别的时候，都会以各种方式保证系统的可用性。而强调可用性的时候，也往往会采用一些技术手段，去保证数据最终是一致的。CAP 定理并没有给出这些情况的具体描述。
4. CAP 理论从工程角度来看只是一种状态的描述，它告诉大家当有错的时候，分布式系统可能处在什么状态。但是，状态是可能变化的。状态间如何转换，如何修补，如何恢复是没有提供方向的。

7. 引申出来的 BASE

正因为 CAP 以上的种种不足，epay 的架构师 Dan Pritchett 根据他自身在大规模分布式系统的实践经验，总结出了 BASE 理论。BASE 理论是对 CAP 理论的延伸，核心思想是即使无法做到强一致性（Strong Consistency），但应用可以采用适合的方式达到最终一致性（Eventual Consitency）。

BASE 理论是实践工程的理论，它弥补了CAP 理论过于抽象的问题，也同时解决了 AP 系统的总体工程实践思想，是分布式系统的核心理论之一，我们将在下一篇文章里，详细的讲解此套理论。

### 分布式事务解决方案

****（讲解1）****

****2PC(二阶段提交)方案：强一致性****

****方案简介****

二阶段提交协议（Two-phase Commit，即 2PC）是常用的分布式事务解决方案，即将事务的提交过程分为两个阶段来进行处理：准备阶段和提交阶段。事务的发起者称协调者，事务的执行者称参与者。

在分布式系统里，每个节点都可以知晓自己操作的成功或者失败，却无法知道其他节点操作的成功或失败。

当一个事务跨多个节点时，为了保持事务的原子性与一致性，而引入一个协调者来统一掌控所有参与者的操作结果，并指示它们是否要把操作结果进行真正的提交或者回滚（rollback）。

****二阶段提交的算法思路可以概括为：****参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。

核心思想就是对每一个事务都采用先尝试后提交的处理方式，处理后所有的读操作都要能获得最新的数据，因此也可以将二阶段提交看作是一个强一致性算法。

****处理流程****

简单一点理解，可以把协调者节点比喻为带头大哥，参与者理解比喻为跟班小弟，带头大哥统一协调跟班小弟的任务执行。

****阶段 1：准备阶段****

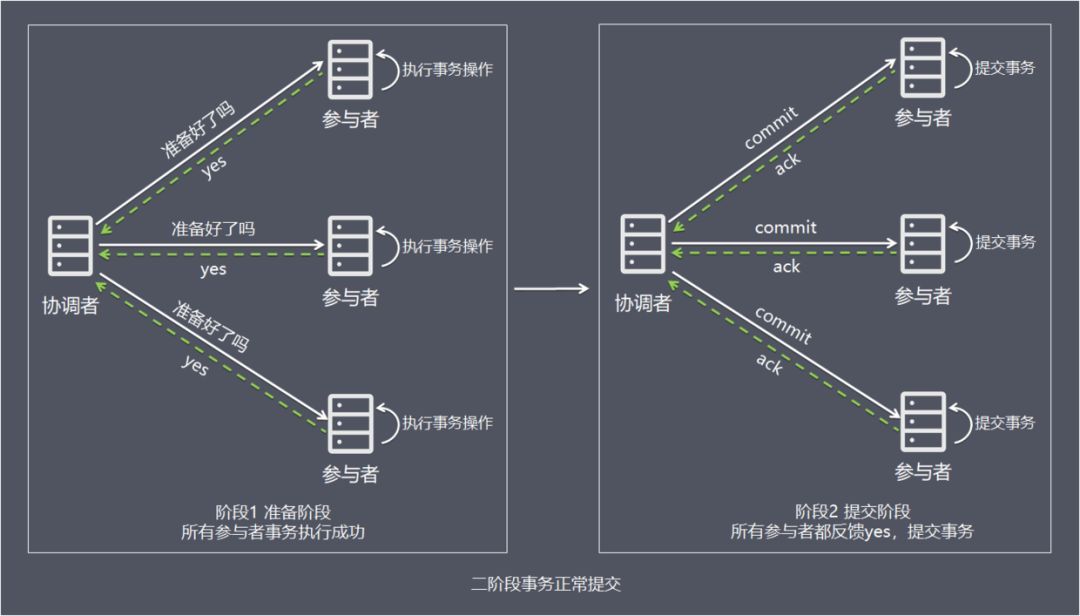
准备阶段有如下三个步骤：

* 协调者向所有参与者发送事务内容，询问是否可以提交事务，并等待所有参与者答复。
* 各参与者执行事务操作，将 undo 和 redo 信息记入事务日志中（但不提交事务）。
* 如参与者执行成功，给协调者反馈 yes，即可以提交；如执行失败，给协调者反馈 no，即不可提交。

****阶段 2：提交阶段****

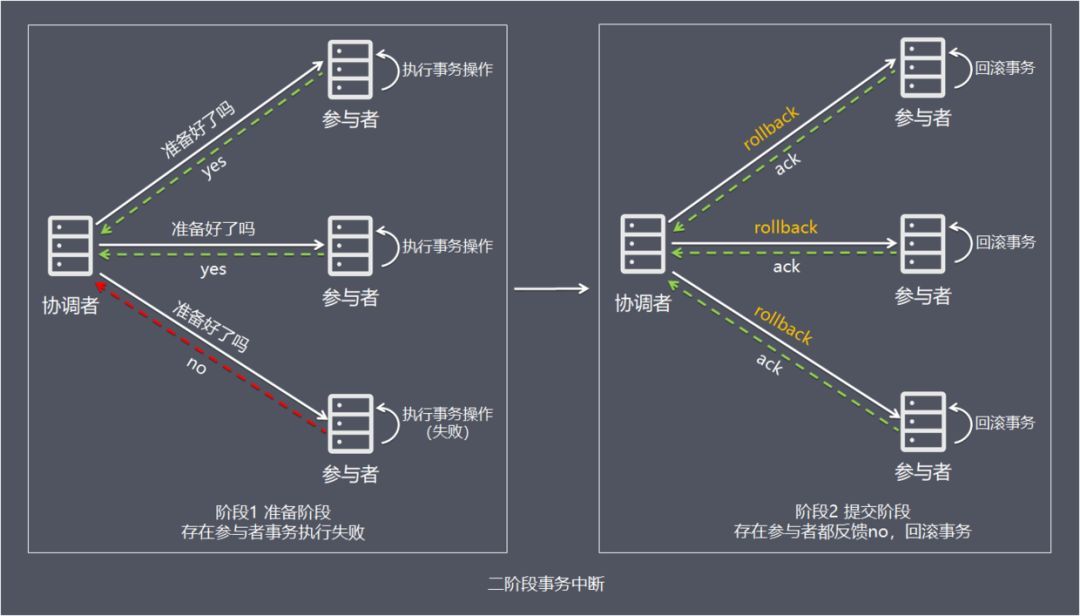
如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(rollback)消息；否则，发送提交(commit)消息。

参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源。(注意：必须在最后阶段释放锁资源) ****接下来分两种情况分别讨论提交阶段的过程。****



情况 1，当所有参与者均反馈 yes，提交事务，如上图：

* 协调者向所有参与者发出正式提交事务的请求（即 commit 请求）。
* 参与者执行 commit 请求，并释放整个事务期间占用的资源。
* 各参与者向协调者反馈 ack(应答)完成的消息。
* 协调者收到所有参与者反馈的 ack 消息后，即完成事务提交。



情况 2，当任何阶段 1 一个参与者反馈 no，中断事务，如上图：

* 协调者向所有参与者发出回滚请求（即 rollback 请求）。
* 参与者使用阶段 1 中的 undo 信息执行回滚操作，并释放整个事务期间占用的资源。
* 各参与者向协调者反馈 ack 完成的消息。
* 协调者收到所有参与者反馈的 ack 消息后，即完成事务中断。

****方案总结****

2PC 方案实现起来简单，实际项目中使用比较少，主要因为以下问题：

* 性能问题：所有参与者在事务提交阶段处于同步阻塞状态，占用系统资源，容易导致性能瓶颈。
* 可靠性问题：如果协调者存在单点故障问题，如果协调者出现故障，参与者将一直处于锁定状态。
* 数据一致性问题：在阶段 2 中，如果发生局部网络问题，一部分事务参与者收到了提交消息，另一部分事务参与者没收到提交消息，那么就导致了节点之间数据的不一致。

****3PC(三阶段提交)方案****

****方案简介****

三阶段提交协议，是二阶段提交协议的改进版本，与二阶段提交不同的是，引入超时机制。同时在协调者和参与者中都引入超时机制。

三阶段提交将二阶段的准备阶段拆分为 2 个阶段，插入了一个 preCommit 阶段，使得原先在二阶段提交中，参与者在准备之后，由于协调者发生崩溃或错误，而导致参与者处于无法知晓是否提交或者中止的“不确定状态”所产生的可能相当长的延时的问题得以解决。

****处理流程****

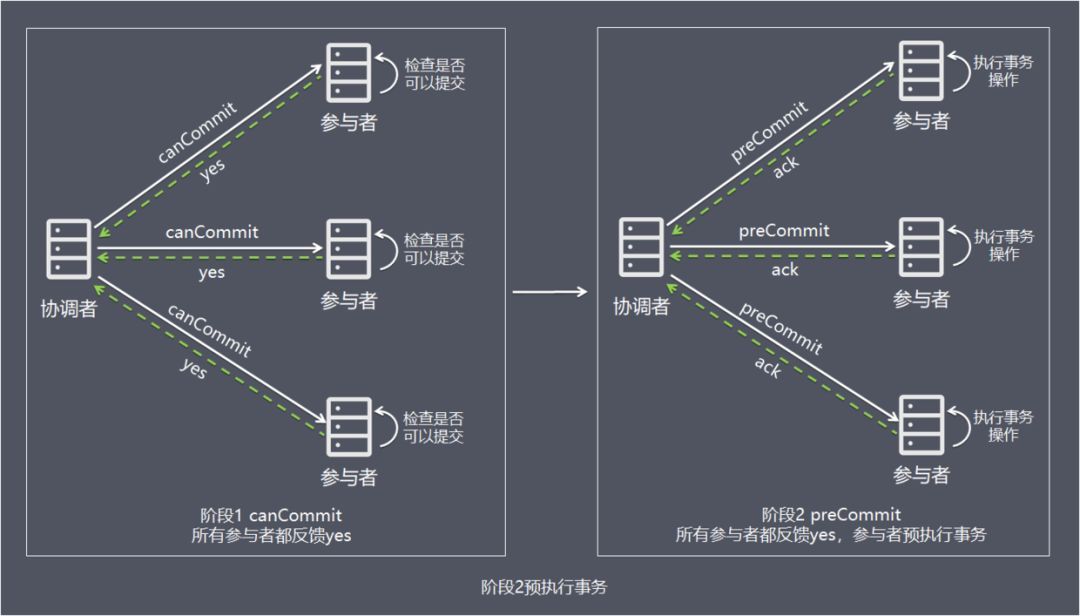
****阶段 1：canCommit****

协调者向参与者发送 commit 请求，参与者如果可以提交就返回 yes 响应(参与者不执行事务操作)，否则返回 no 响应：

* 协调者向所有参与者发出包含事务内容的 canCommit 请求，询问是否可以提交事务，并等待所有参与者答复。
* 参与者收到 canCommit 请求后，如果认为可以执行事务操作，则反馈 yes 并进入预备状态，否则反馈 no。

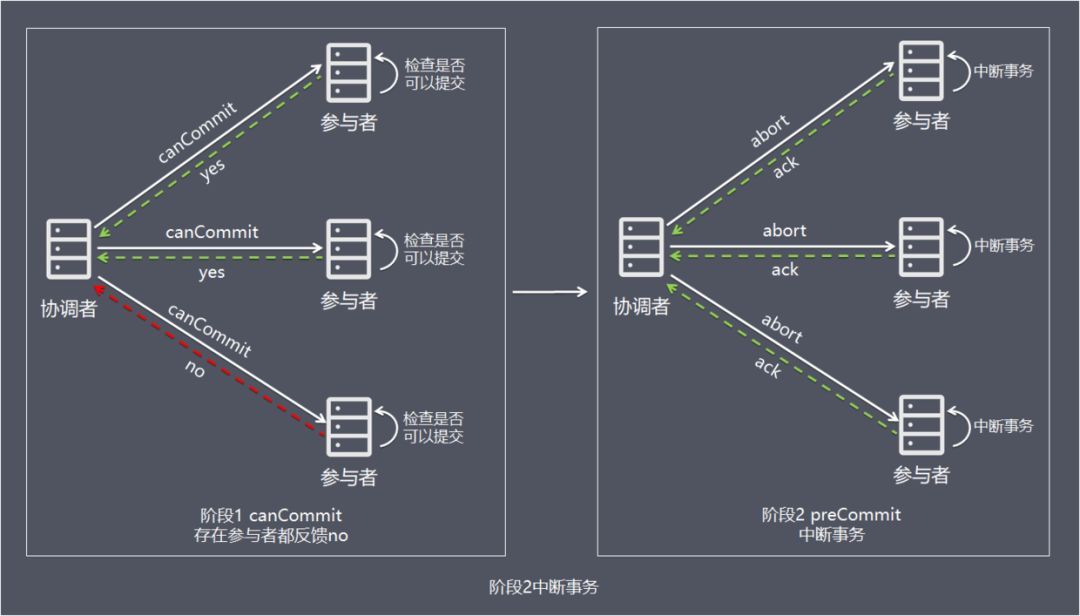
****阶段 2：preCommit****

协调者根据阶段 1 canCommit 参与者的反应情况来决定是否可以进行基于事务的 preCommit 操作。****根据响应情况，有以下两种可能。****



情况 1：阶段 1 所有参与者均反馈 yes，参与者预执行事务，如上图：

* 协调者向所有参与者发出 preCommit 请求，进入准备阶段。
* 参与者收到 preCommit 请求后，执行事务操作，将 undo 和 redo 信息记入事务日志中（但不提交事务）。
* 各参与者向协调者反馈 ack 响应或 no 响应，并等待最终指令。

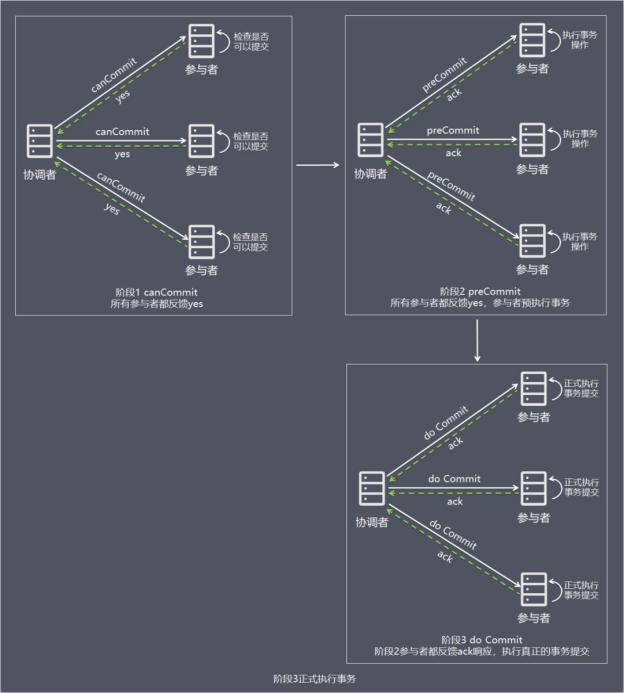


情况 2：阶段 1 任何一个参与者反馈 no，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务，如上图：

* 协调者向所有参与者发出 abort 请求。
* 无论收到协调者发出的 abort 请求，或者在等待协调者请求过程中出现超时，参与者均会中断事务。

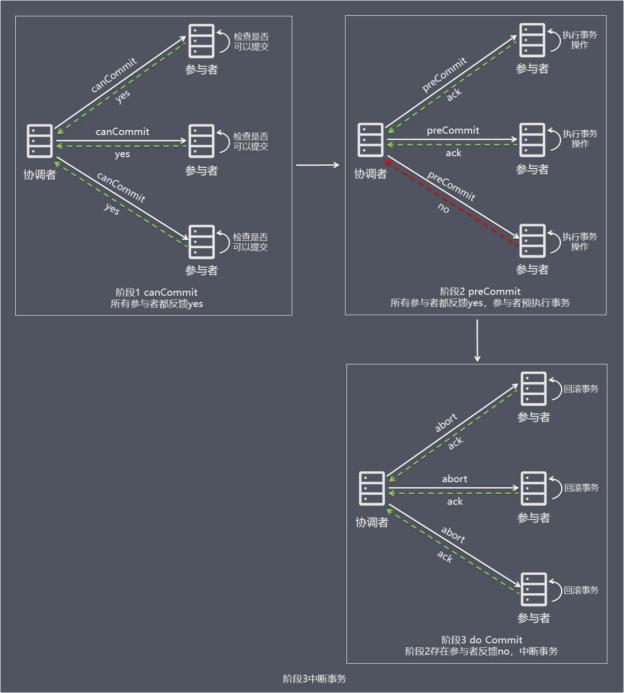
****阶段 3：do Commit****

该阶段进行真正的事务提交，也可以分为以下两种情况。



情况 1：阶段 2 所有参与者均反馈 ack 响应，执行真正的事务提交，如上图：

* 如果协调者处于工作状态，则向所有参与者发出 do Commit 请求。
* 参与者收到 do Commit 请求后，会正式执行事务提交，并释放整个事务期间占用的资源。
* 各参与者向协调者反馈 ack 完成的消息。
* 协调者收到所有参与者反馈的 ack 消息后，即完成事务提交。



情况 2：阶段 2 任何一个参与者反馈 no，或者等待超时后协调者尚无法收到所有参与者的反馈，即中断事务，如上图：

* 如果协调者处于工作状态，向所有参与者发出 abort 请求。
* 参与者使用阶段 1 中的 undo 信息执行回滚操作，并释放整个事务期间占用的资源。
* 各参与者向协调者反馈 ack 完成的消息。
* 协调者收到所有参与者反馈的 ack 消息后，即完成事务中断。

****注意：****进入阶段 3 后，无论协调者出现问题，或者协调者与参与者网络出现问题，都会导致参与者无法接收到协调者发出的 do Commit 请求或 abort 请求。此时，参与者都会在等待超时之后，继续执行事务提交。

****方案总结****

****优点：****相比二阶段提交，三阶段提交降低了阻塞范围，在等待超时后协调者或参与者会中断事务。避免了协调者单点问题，阶段 3 中协调者出现问题时，参与者会继续提交事务。

****缺点：****数据不一致问题依然存在，当在参与者收到 preCommit 请求后等待 do commite 指令时，此时如果协调者请求中断事务，而协调者无法与参与者正常通信，会导致参与者继续提交事务，造成数据不一致。

****TCC 事务：最终一致性****

****方案简介****

TCC（Try-Confirm-Cancel）的概念，最早是由 Pat Helland 于 2007 年发表的一篇名为《Life beyond Distributed Transactions:an Apostate’s Opinion》的论文提出。

TCC 是服务化的二阶段编程模型，其 Try、Confirm、Cancel 3 个方法均由业务编码实现：

* Try 操作作为一阶段，负责资源的检查和预留。
* Confirm 操作作为二阶段提交操作，执行真正的业务。
* Cancel 是预留资源的取消。

TCC 事务的 Try、Confirm、Cancel 可以理解为 SQL 事务中的 Lock、Commit、Rollback。

****处理流程****

为了方便理解，下面以电商下单为例进行方案解析，这里把整个过程简单分为扣减库存，订单创建 2 个步骤，库存服务和订单服务分别在不同的服务器节点上。

****①Try 阶段****

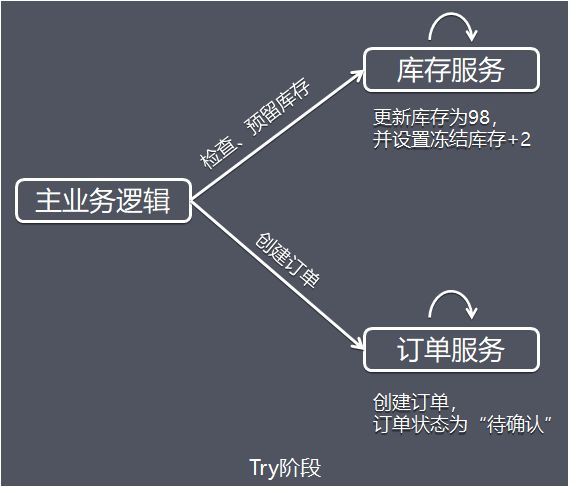
从执行阶段来看，与传统事务机制中业务逻辑相同。但从业务角度来看，却不一样。

TCC 机制中的 Try 仅是一个初步操作，它和后续的确认一起才能真正构成一个完整的业务逻辑，这个阶段主要完成：

* 完成所有业务检查( 一致性 ) 。
* 预留必须业务资源( 准隔离性 ) 。
* Try 尝试执行业务。

TCC 事务机制以初步操作（Try）为中心的，确认操作（Confirm）和取消操作（Cancel）都是围绕初步操作（Try）而展开。

因此，Try 阶段中的操作，其保障性是最好的，即使失败，仍然有取消操作（Cancel）可以将其执行结果撤销。



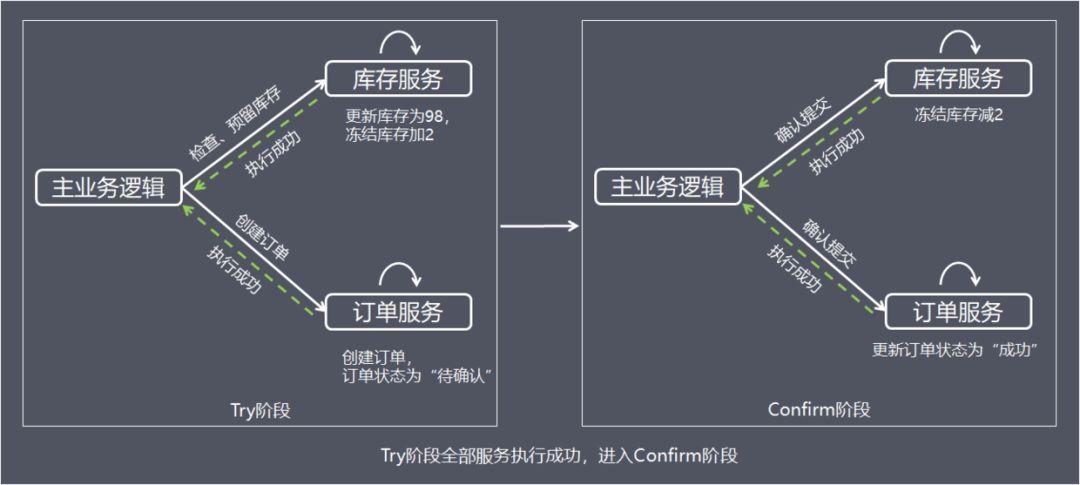
假设商品库存为 100，购买数量为 2，这里检查和更新库存的同时，冻结用户购买数量的库存，同时创建订单，订单状态为待确认。

****②Confirm / Cancel 阶段****

根据 Try 阶段服务是否全部正常执行，继续执行确认操作（Confirm）或取消操作（Cancel）。

Confirm 和 Cancel 操作满足幂等性，如果 Confirm 或 Cancel 操作执行失败，将会不断重试直到执行完成。

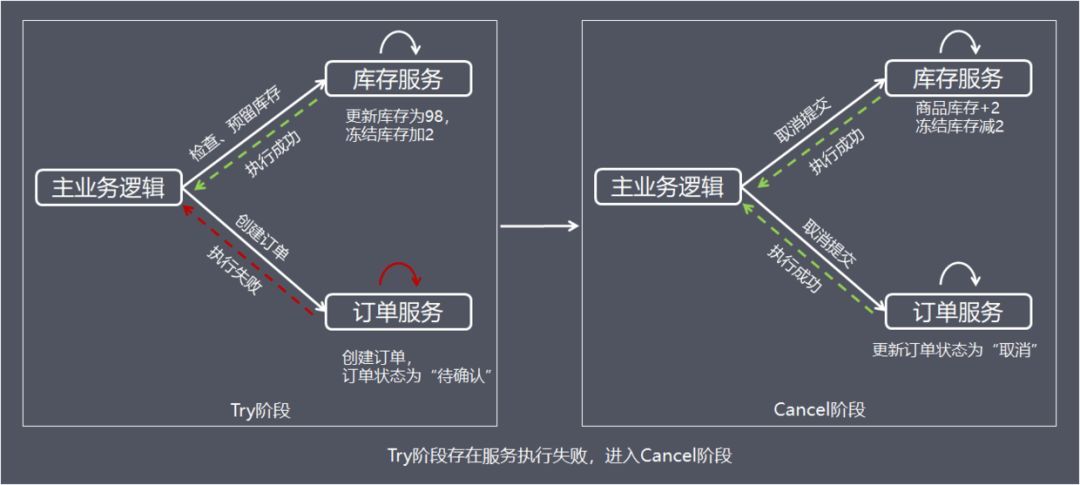
****Confirm：当 Try 阶段服务全部正常执行， 执行确认业务逻辑操作****



这里使用的资源一定是 Try 阶段预留的业务资源。在 TCC 事务机制中认为，如果在 Try 阶段能正常的预留资源，那 Confirm 一定能完整正确的提交。

Confirm 阶段也可以看成是对 Try 阶段的一个补充，Try+Confirm 一起组成了一个完整的业务逻辑。

****Cancel：当 Try 阶段存在服务执行失败， 进入 Cancel 阶段****



Cancel 取消执行，释放 Try 阶段预留的业务资源，上面的例子中，Cancel 操作会把冻结的库存释放，并更新订单状态为取消。

****方案总结****

TCC 事务机制相对于传统事务机制（X/Open XA），TCC 事务机制相比于上面介绍的 XA 事务机制，有以下优点：

* 性能提升：具体业务来实现控制资源锁的粒度变小，不会锁定整个资源。
* 数据最终一致性：基于 Confirm 和 Cancel 的幂等性，保证事务最终完成确认或者取消，保证数据的一致性。
* 可靠性：解决了 XA 协议的协调者单点故障问题，由主业务方发起并控制整个业务活动，业务活动管理器也变成多点，引入集群。

****缺点：****TCC 的 Try、Confirm 和 Cancel 操作功能要按具体业务来实现，业务耦合度较高，提高了开发成本。

****本地消息表：最终一致性****

****方案简介****

本地消息表的方案最初是由 eBay 提出，核心思路是将分布式事务拆分成本地事务进行处理。

方案通过在事务主动发起方额外新建事务消息表，事务发起方处理业务和记录事务消息在本地事务中完成，轮询事务消息表的数据发送事务消息，事务被动方基于消息中间件消费事务消息表中的事务。

这样设计可以避免”业务处理成功 + 事务消息发送失败"，或"业务处理失败 + 事务消息发送成功"的棘手情况出现，保证 2 个系统事务的数据一致性。

****处理流程****

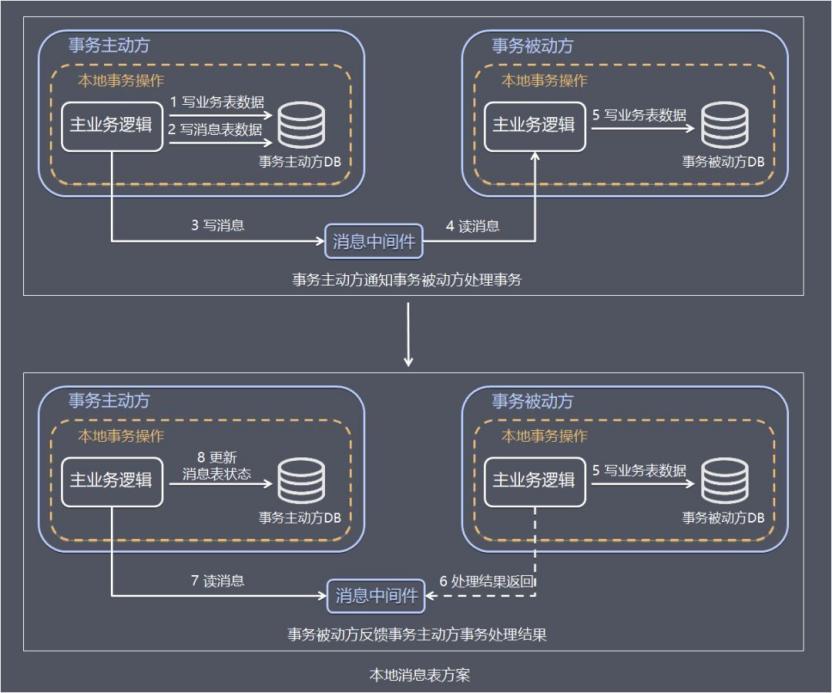
下面把分布式事务最先开始处理的事务方称为事务主动方，在事务主动方之后处理的业务内的其他事务称为事务被动方。

为了方便理解，下面继续以电商下单为例进行方案解析，这里把整个过程简单分为扣减库存，订单创建 2 个步骤。

库存服务和订单服务分别在不同的服务器节点上，其中库存服务是事务主动方，订单服务是事务被动方。

事务的主动方需要额外新建事务消息表，用于记录分布式事务的消息的发生、处理状态。

整个业务处理流程如下：



****步骤1：****事务主动方处理本地事务。

事务主动方在本地事务中处理业务更新操作和写消息表操作。上面例子中库存服务阶段在本地事务中完成扣减库存和写消息表(图中 1、2)。

****步骤 2：****事务主动方通过消息中间件，通知事务被动方处理事务通知事务待消息。

消息中间件可以基于 Kafka、RocketMQ 消息队列，事务主动方主动写消息到消息队列，事务消费方消费并处理消息队列中的消息。

上面例子中，库存服务把事务待处理消息写到消息中间件，订单服务消费消息中间件的消息，完成新增订单（图中 3 - 5）。

****步骤 3：****事务被动方通过消息中间件，通知事务主动方事务已处理的消息。

上面例子中，订单服务把事务已处理消息写到消息中间件，库存服务消费中间件的消息，并将事务消息的状态更新为已完成(图中 6 - 8)。

为了数据的一致性，当处理错误需要重试，事务发送方和事务接收方相关业务处理需要支持幂等。

具体保存一致性的容错处理如下：

* 当步骤 1 处理出错，事务回滚，相当于什么都没发生。
* 当步骤 2、步骤 3 处理出错，由于未处理的事务消息还是保存在事务发送方，事务发送方可以定时轮询为超时消息数据，再次发送到消息中间件进行处理。事务被动方消费事务消息重试处理。
* 如果是业务上的失败，事务被动方可以发消息给事务主动方进行回滚。
* 如果多个事务被动方已经消费消息，事务主动方需要回滚事务时需要通知事务被动方回滚。

****方案总结****

方案的优点如下：

* 从应用设计开发的角度实现了消息数据的可靠性，消息数据的可靠性不依赖于消息中间件，弱化了对 MQ 中间件特性的依赖。
* 方案轻量，容易实现。

缺点如下：

* 与具体的业务场景绑定，耦合性强，不可公用。
* 消息数据与业务数据同库，占用业务系统资源。
* 业务系统在使用关系型数据库的情况下，消息服务性能会受到关系型数据库并发性能的局限。

****MQ 事务：最终一致性****

****方案简介****

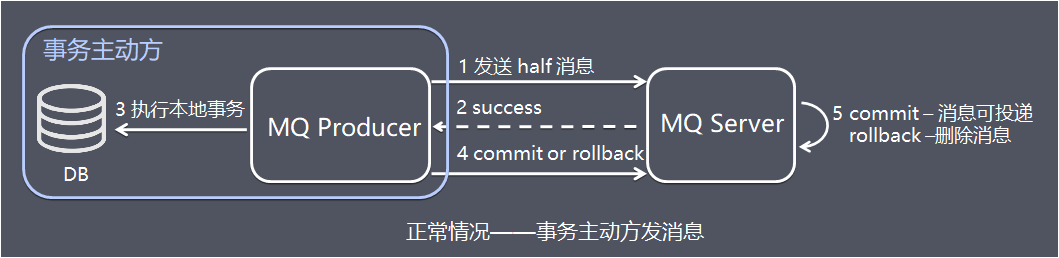
基于 MQ 的分布式事务方案其实是对本地消息表的封装，将本地消息表基于 MQ 内部，其他方面的协议基本与本地消息表一致。

****处理流程****

下面主要基于 RocketMQ 4.3 之后的版本介绍 MQ 的分布式事务方案。

在本地消息表方案中，保证事务主动方发写业务表数据和写消息表数据的一致性是基于数据库事务，RocketMQ 的事务消息相对于普通 MQ，相对于提供了 2PC 的提交接口，方案如下：

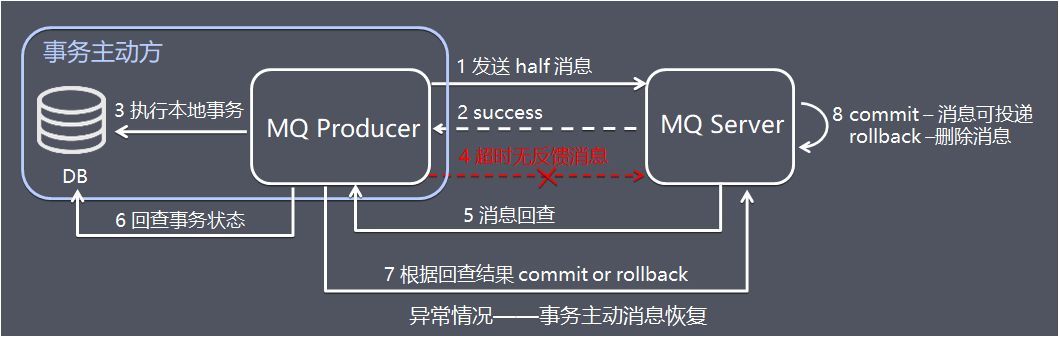
****正常情况：事务主动方发消息****



这种情况下，事务主动方服务正常，没有发生故障，发消息流程如下：

* 图中 1：发送方向 MQ 服务端(MQ Server)发送 half 消息。
* 图中 2：MQ Server 将消息持久化成功之后，向发送方 ack 确认消息已经发送成功。
* 图中 3：发送方开始执行本地事务逻辑。
* 图中 4：发送方根据本地事务执行结果向 MQ Server 提交二次确认（commit 或是 rollback）。
* 图中 5：MQ Server 收到 commit 状态则将半消息标记为可投递，订阅方最终将收到该消息；MQ Server 收到 rollback 状态则删除半消息，订阅方将不会接受该消息。

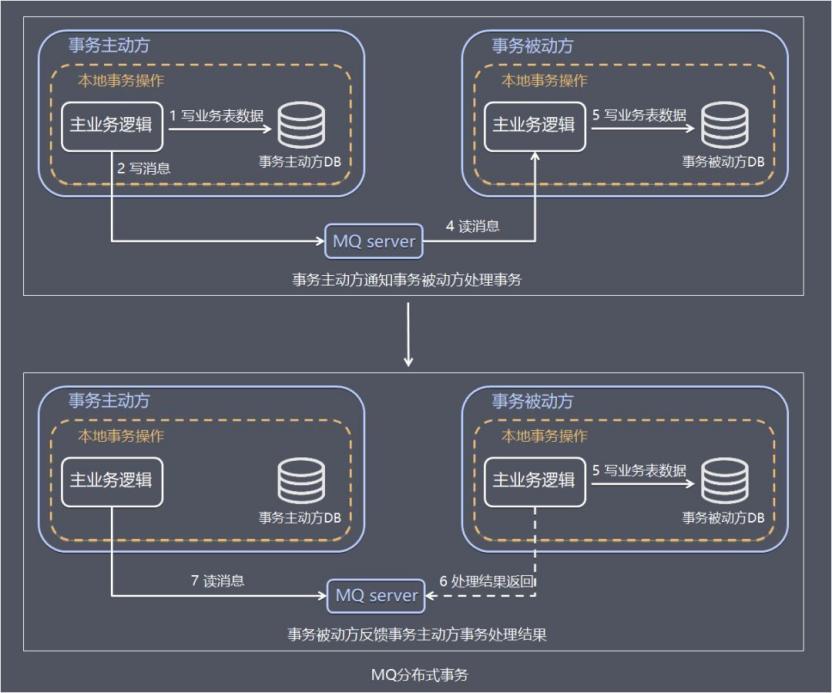
****异常情况：事务主动方消息恢复****



在断网或者应用重启等异常情况下，图中 4 提交的二次确认超时未到达 MQ Server，此时处理逻辑如下：

* 图中 5：MQ Server 对该消息发起消息回查。
* 图中 6：发送方收到消息回查后，需要检查对应消息的本地事务执行的最终结果。
* 图中 7：发送方根据检查得到的本地事务的最终状态再次提交二次确认。
* 图中 8：MQ Server基于 commit/rollback 对消息进行投递或者删除。

介绍完 RocketMQ 的事务消息方案后，由于前面已经介绍过本地消息表方案，这里就简单介绍 RocketMQ 分布式事务：



事务主动方基于 MQ 通信通知事务被动方处理事务，事务被动方基于 MQ 返回处理结果。

如果事务被动方消费消息异常，需要不断重试，业务处理逻辑需要保证幂等。

如果是事务被动方业务上的处理失败，可以通过 MQ 通知事务主动方进行补偿或者事务回滚。

****方案总结****

相比本地消息表方案，MQ 事务方案优点是：

* 消息数据独立存储 ，降低业务系统与消息系统之间的耦合。
* 吞吐量由于使用本地消息表方案。

缺点是：

* 一次消息发送需要两次网络请求(half 消息 + commit/rollback 消息) 。
* 业务处理服务需要实现消息状态回查接口。

****Saga 事务：最终一致性****

****方案简介****

Saga 事务源于 1987 年普林斯顿大学的 Hecto 和 Kenneth 发表的如何处理 long lived transaction（长活事务）论文。

Saga 事务核心思想是将长事务拆分为多个本地短事务，由 Saga 事务协调器协调，如果正常结束那就正常完成，如果某个步骤失败，则根据相反顺序一次调用补偿操作。

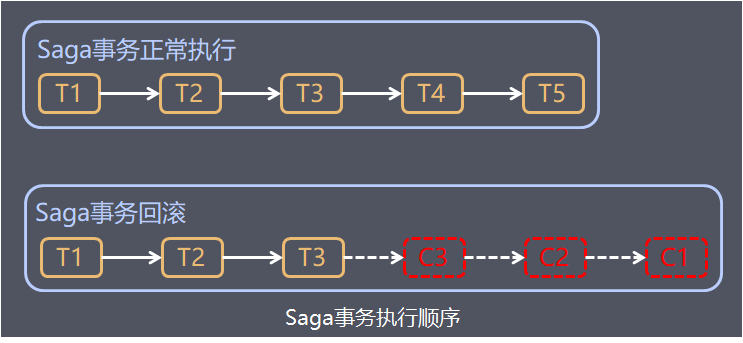
****处理流程****

Saga 事务基本协议如下：

* 每个 Saga 事务由一系列幂等的有序子事务(sub-transaction) Ti 组成。
* 每个 Ti 都有对应的幂等补偿动作 Ci，补偿动作用于撤销 Ti 造成的结果。

可以看到，和 TCC 相比，Saga 没有“预留”动作，它的 Ti 就是直接提交到库。

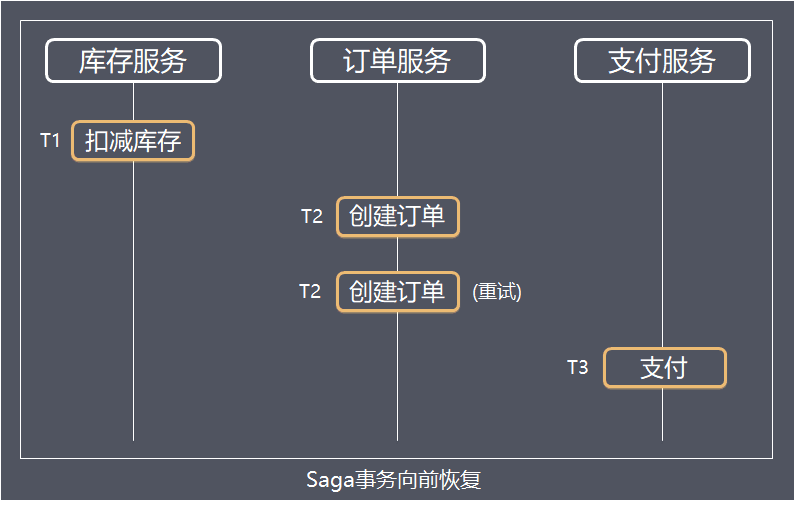
下面以下单流程为例，整个操作包括：创建订单、扣减库存、支付、增加积分。



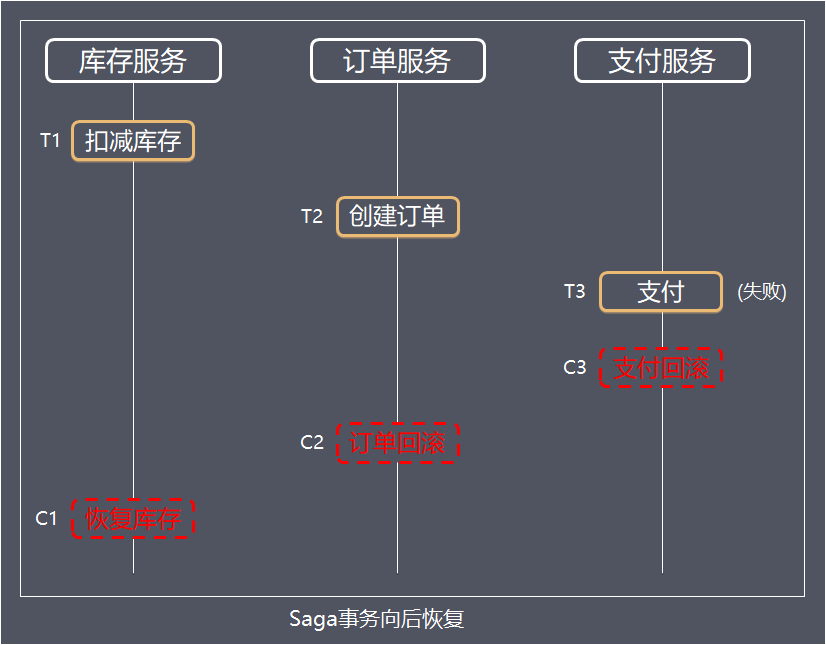
Saga 的执行顺序有两种，如上图：

* 事务正常执行完成：T1, T2, T3, ..., Tn，例如：扣减库存(T1)，创建订单(T2)，支付(T3)，依次有序完成整个事务。
* 事务回滚：T1, T2, ..., Tj, Cj,..., C2, C1，其中 0 < j < n，例如：扣减库存(T1)，创建订单(T2)，支付(T3，支付失败)，支付回滚(C3)，订单回滚(C2)，恢复库存(C1)。

Saga 定义了两种恢复策略：



****向前恢复（forward recovery）：****对应于上面第一种执行顺序，适用于必须要成功的场景，发生失败进行重试，执行顺序是类似于这样的：T1, T2, ..., Tj(失败), Tj(重试),..., Tn，其中j是发生错误的子事务(sub-transaction)。该情况下不需要Ci。

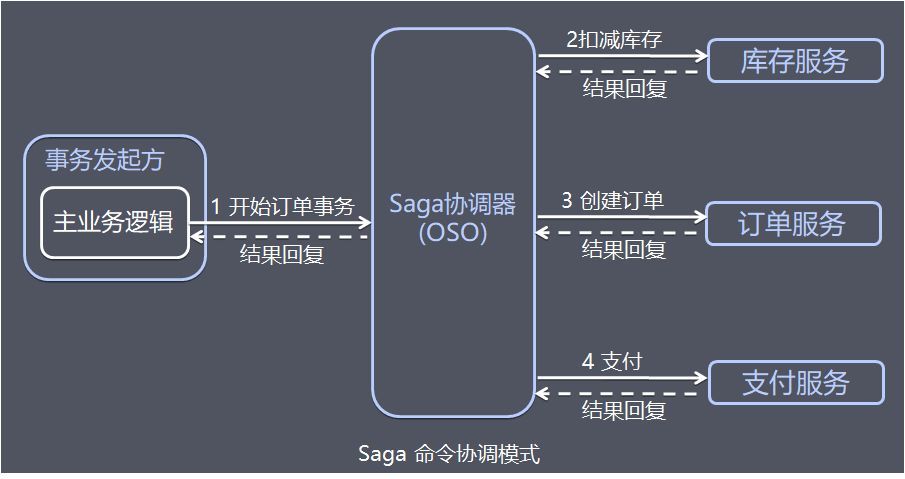


****向后恢复（backward recovery）：****对应于上面提到的第二种执行顺序，其中 j 是发生错误的子事务(sub-transaction)，这种做法的效果是撤销掉之前所有成功的子事务，使得整个 Saga 的执行结果撤销。

Saga 事务常见的有两种不同的实现方式：

****①命令协调（Order Orchestrator）：****中央协调器负责集中处理事件的决策和业务逻辑排序。

中央协调器（Orchestrator，简称 OSO）以命令/回复的方式与每项服务进行通信，全权负责告诉每个参与者该做什么以及什么时候该做什么。



以电商订单的例子为例：

* 事务发起方的主业务逻辑请求 OSO 服务开启订单事务
* OSO 向库存服务请求扣减库存，库存服务回复处理结果。
* OSO 向订单服务请求创建订单，订单服务回复创建结果。
* OSO 向支付服务请求支付，支付服务回复处理结果。
* 主业务逻辑接收并处理 OSO 事务处理结果回复。

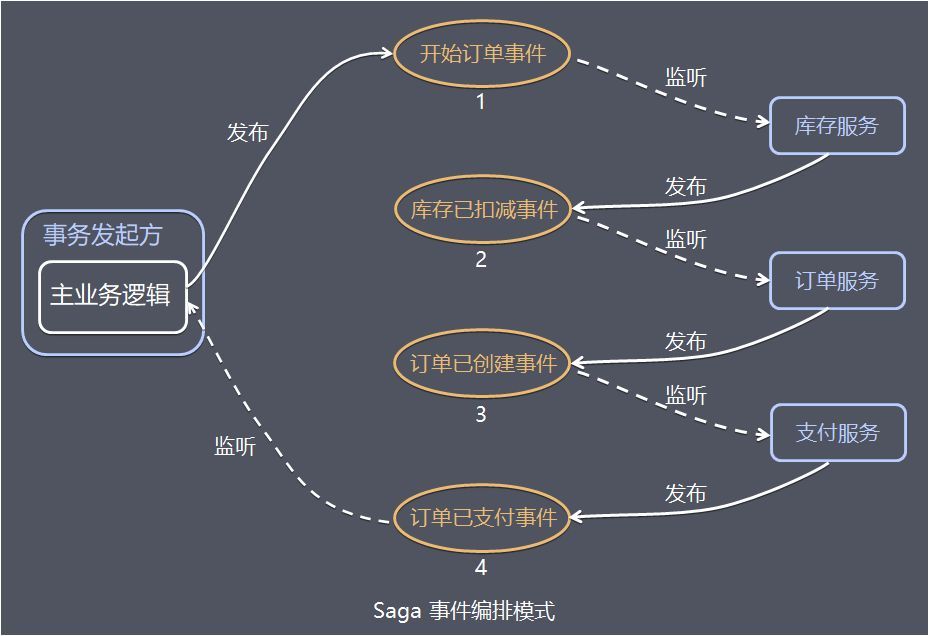
中央协调器必须事先知道执行整个订单事务所需的流程(例如通过读取配置)。如果有任何失败，它还负责通过向每个参与者发送命令来撤销之前的操作来协调分布式的回滚。

基于中央协调器协调一切时，回滚要容易得多，因为协调器默认是执行正向流程，回滚时只要执行反向流程即可。

****②事件编排（Event Choreography0）：****没有中央协调器（没有单点风险）时，每个服务产生并观察其他服务的事件，并决定是否应采取行动。

在事件编排方法中，第一个服务执行一个事务，然后发布一个事件。该事件被一个或多个服务进行监听，这些服务再执行本地事务并发布（或不发布）新的事件。

当最后一个服务执行本地事务并且不发布任何事件时，意味着分布式事务结束，或者它发布的事件没有被任何 Saga 参与者听到都意味着事务结束。



以电商订单的例子为例：

* 事务发起方的主业务逻辑发布开始订单事件。
* 库存服务监听开始订单事件，扣减库存，并发布库存已扣减事件。
* 订单服务监听库存已扣减事件，创建订单，并发布订单已创建事件。
* 支付服务监听订单已创建事件，进行支付，并发布订单已支付事件。
* 主业务逻辑监听订单已支付事件并处理。

事件/编排是实现 Saga 模式的自然方式，它很简单，容易理解，不需要太多的代码来构建。如果事务涉及 2 至 4 个步骤，则可能是非常合适的。

****方案总结****

命令协调设计的优点如下：

* 服务之间关系简单，避免服务之间的循环依赖关系，因为 Saga 协调器会调用 Saga 参与者，但参与者不会调用协调器。
* 程序开发简单，只需要执行命令/回复(其实回复消息也是一种事件消息)，降低参与者的复杂性。
* 易维护扩展，在添加新步骤时，事务复杂性保持线性，回滚更容易管理，更容易实施和测试。

命令协调设计缺点如下：

* 中央协调器容易处理逻辑容易过于复杂，导致难以维护。
* 存在协调器单点故障风险。

事件/编排设计优点如下：

* 避免中央协调器单点故障风险。
* 当涉及的步骤较少服务开发简单，容易实现。

事件/编排设计缺点如下：

* 服务之间存在循环依赖的风险。
* 当涉及的步骤较多，服务间关系混乱，难以追踪调测。

值得补充的是，由于 Saga 模型中没有 Prepare 阶段，因此事务间不能保证隔离性。

当多个 Saga 事务操作同一资源时，就会产生更新丢失、脏数据读取等问题，这时需要在业务层控制并发，例如：在应用层面加锁，或者应用层面预先冻结资源。

总结

****各方案使用场景****



介绍完分布式事务相关理论和常见解决方案后，最终的目的在实际项目中运用，因此，总结一下各个方案的常见的使用场景：

* 2PC/3PC：依赖于数据库，能够很好的提供强一致性和强事务性，但相对来说延迟比较高，比较适合传统的单体应用，在同一个方法中存在跨库操作的情况，不适合高并发和高性能要求的场景。
* TCC：适用于执行时间确定且较短，实时性要求高，对数据一致性要求高，比如互联网金融企业最核心的三个服务：交易、支付、账务。
* 本地消息表/MQ 事务：都适用于事务中参与方支持操作幂等，对一致性要求不高，业务上能容忍数据不一致到一个人工检查周期，事务涉及的参与方、参与环节较少，业务上有对账/校验系统兜底。
* Saga 事务：由于 Saga 事务不能保证隔离性，需要在业务层控制并发，适合于业务场景事务并发操作同一资源较少的情况。 Saga 相比缺少预提交动作，导致补偿动作的实现比较麻烦，例如业务是发送短信，补偿动作则得再发送一次短信说明撤销，用户体验比较差。Saga 事务较适用于补偿动作容易处理的场景。

****分布式事务方案设计****

本文介绍的偏向于原理，业界已经有不少开源的或者收费的解决方案，篇幅所限，就不再展开介绍。

实际运用理论时进行架构设计时，许多人容易犯“手里有了锤子，看什么都觉得像钉子”的错误，设计方案时考虑的问题场景过多，各种重试，各种补偿机制引入系统，导致系统过于复杂，落地遥遥无期。

世界上解决一个计算机问题最简单的方法：“恰好”不需要解决它！

—— 阿里中间件技术专家沈询

有些问题，看起来很重要，但实际上我们可以通过合理的设计或者将问题分解来规避。

设计分布式事务系统也不是需要考虑所有异常情况，不必过度设计各种回滚，补偿机制。

如果硬要把时间花在解决问题本身，实际上不仅效率低下，而且也是一种浪费。

如果系统要实现回滚流程的话，有可能系统复杂度将大大提升，且很容易出现 Bug，估计出现 Bug 的概率会比需要事务回滚的概率大很多。

在设计系统时，我们需要衡量是否值得花这么大的代价来解决这样一个出现概率非常小的问题，可以考虑当出现这个概率很小的问题，能否采用人工解决的方式，这也是大家在解决疑难问题时需要多多思考的地方。

参考资料：

* technology-talk —— 事务
* MySQL 中事务的实现
* 分布式一致性算法 2PC 和 3PC
* 分布式开放消息系统(RocketMQ)的原理与实践
* RocketMQ 事务消息入门介绍
* Saga 分布式事务解决方案与实践 —— 姜宁
* 分布式事务 Saga 模式
* 从一笔金币充值去思考分布式事务

****（讲解2）****

## **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。

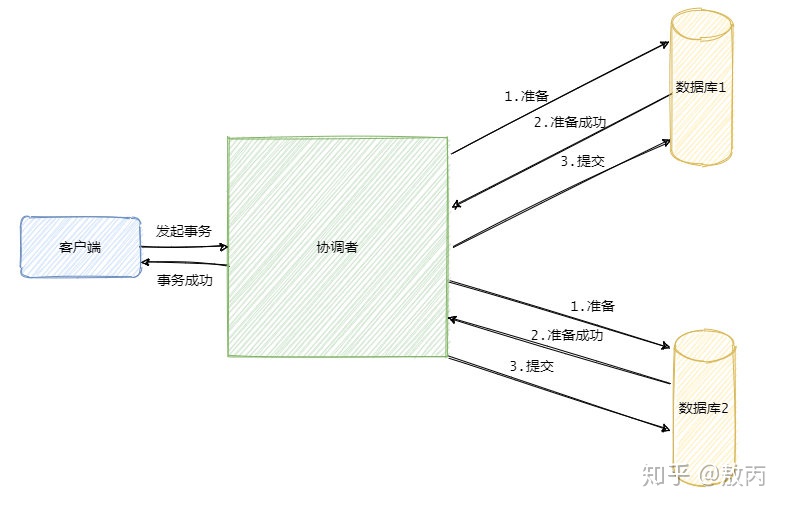
让我们来看下两个阶段的具体流程。

**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。

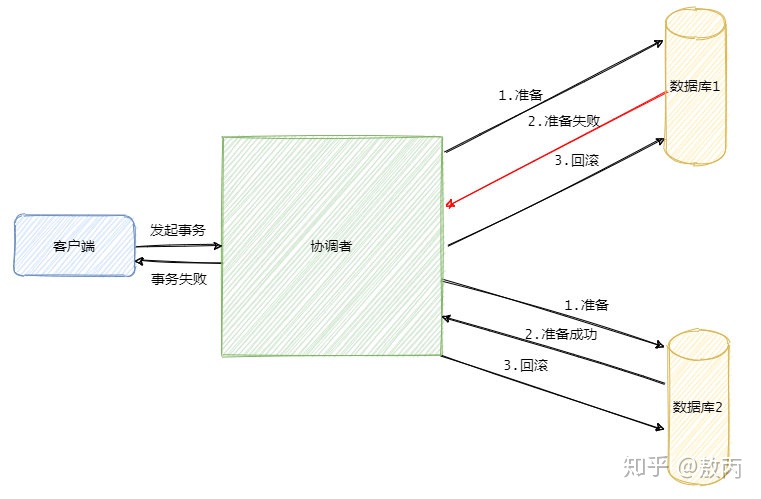
同步等待所有资源的响应之后就进入第二阶段即提交阶段（注意提交阶段不一定是提交事务，也可能是回滚事务）。

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**。

首先 2PC 是一个**同步阻塞协议**，像第一阶段协调者会等待所有参与者响应才会进行下一步操作，当然第一阶段的**协调者有超时机制**，假设因为网络原因没有收到某参与者的响应或某参与者挂了，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。

虽然协议上没说，不过在实现的时候我们可以灵活的让协调者将自己发过的请求在哪个地方记一下，也就是日志记录，这样新协调者来的时候不就知道此时该不该发了？

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 {

write VOTE\_COMMIT to local log; //记录自己的决策

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。

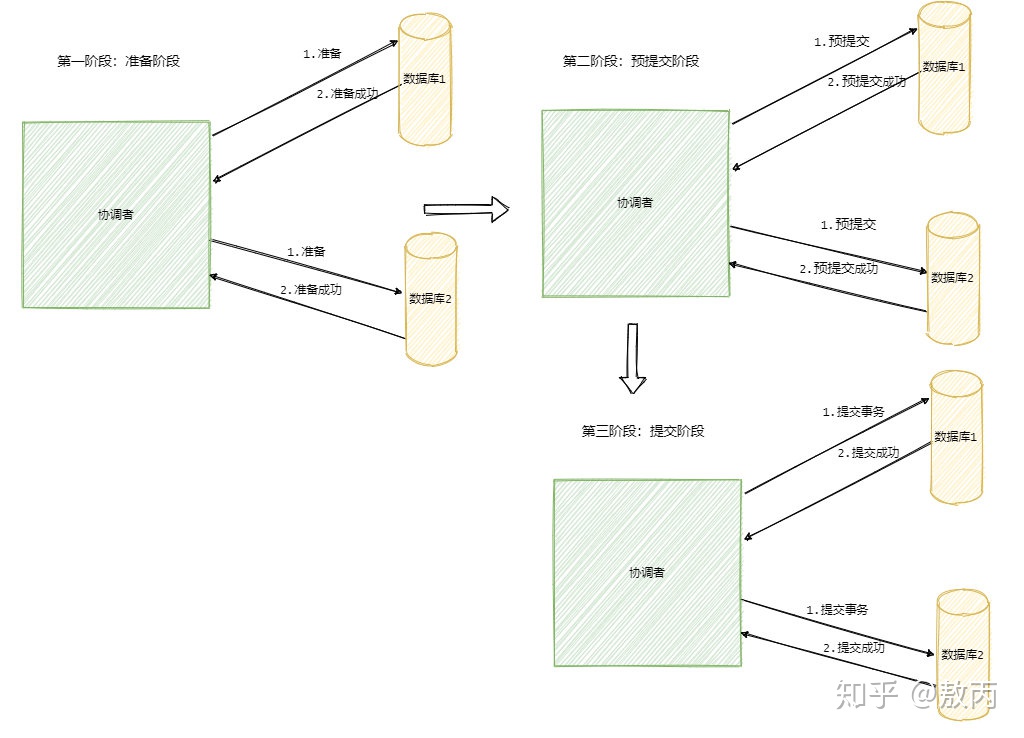
让我们来详细看一下。

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。

而预提交阶段就是和 2PC 的准备阶段一样，除了事务的提交该做的都做了。

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！

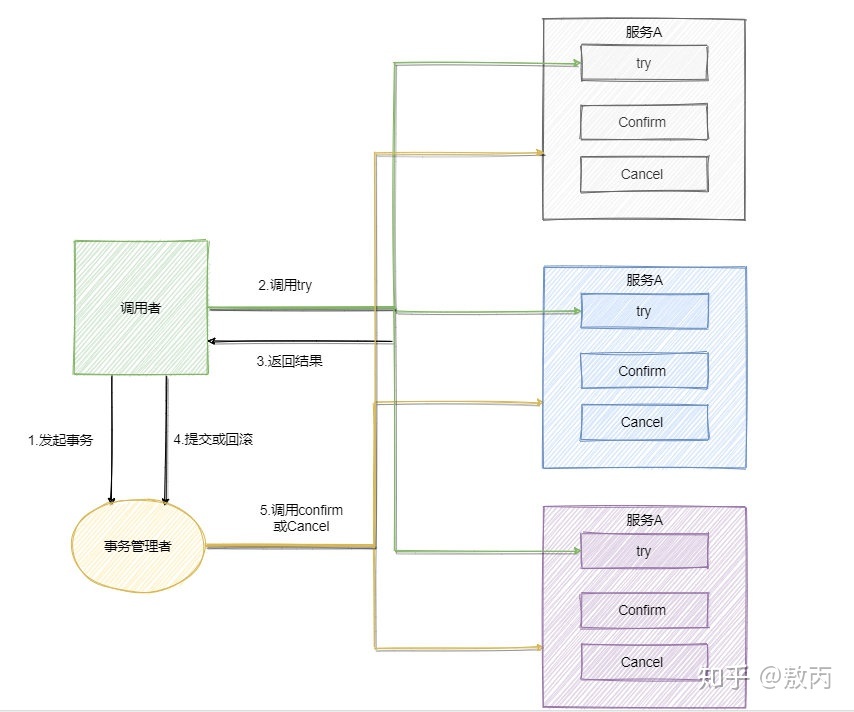
TCC 指的是Try - Confirm - Cancel。

* 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**。

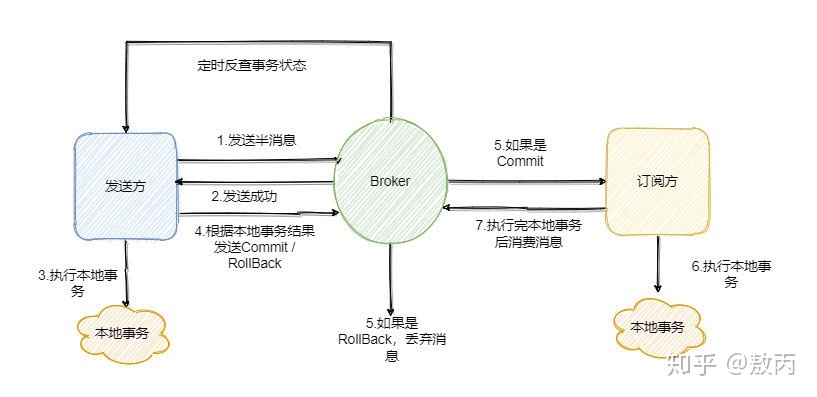
再根据**本地事务的结果向 Broker 发送 Commit 或者 RollBack 命令**。

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务。