# 分布式事务

## 1.什么是分布式事务

分布式事务，是分布式系统中特有的问题之一，在之前的一体式架构系统中，不会存在。随着互联网快速发展，微服务，SOA等服务架构模式正在被大规模的使用，现在分布式系统一般由多个独立的子系统组成，多个子系统通过网络通信互相协作配合完成各个功能有很多用例会跨多个子系统才能完成，比较典型的是电子商务网站的下单支付流程，至少会涉及交易系统和支付系统，而且这个过程中会涉及到事务的概念，即保证交易系统和支付系统的数据一致性，此处我们称这种跨系统的事务为分布式事务，具体一点而言，分布式事务是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。



## 2.分布式事务的难点

 **事务的原子性** 事务操作跨不同节点，当多个节点某一节点操作失败时，需要保证多节点操作的\*\*要么什么都不做，要么做全套（All or Nothing）\*\*的原子性。

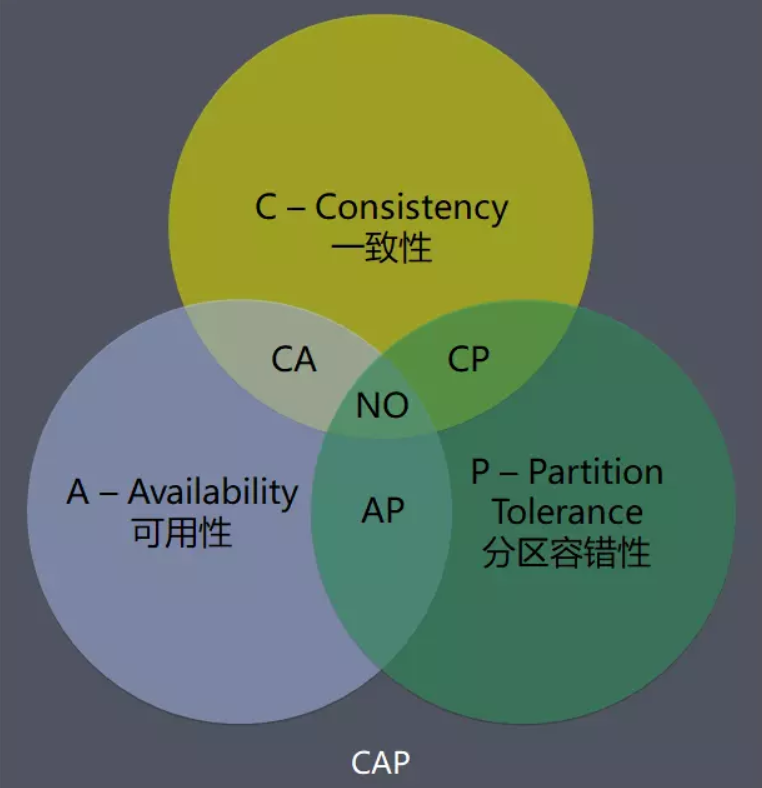
 **事务的一致性** 当发生网络传输故障或者节点故障，节点间数据复制通道中断，在进行事务操作时需要保证数据一致性，保证事务的任何操作都不会使得数据违反数据库定义的约束、触发器等规则。

 **事务的隔离性** 事务隔离性的本质就是如何正确多个并发事务的处理的读写冲突和写写冲突，因为在分布式事务控制中，可能会出现提交不同步的现象，这个时候就有可能出现“部分已经提交”的事务。此时并发应用访问数据如果没有加以控制，有可能出现“脏读”问题。

## 3.分布式系统的一致性CAP

CAP理论

C : 一致性 A: 可读性 P: 分区容错性



**CAP理论的定义** 在一个分布式系统（指互相连接并共享数据的节点的集合）中，当涉及读写操作时，只能保证一致性（Consistence）、可用性（Availability）、分区容错性（PartitionTolerance）三者中的两个，另外一个必须被牺牲。

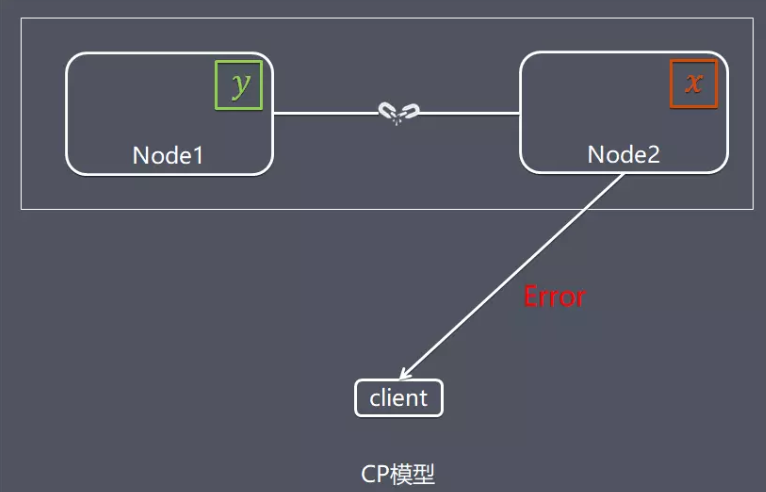
* **C-Consistency 一致性** 对于某个指定客户端来说，读操作保证能够返回最新的写操作结果，这里并不是强调同一时刻拥有相同的数据，对于系统执行事务来说，在事务执行过程中，系统处于一个不一致的状态，不同节点的数据并不完全一致，一致性强调客户端都操作能够获得最新的写操作结果，因为在事务的执行过程中，客户端是无法读取未提交的数据，只有等到事务提交后，客户端才能读取到事务写入的数据，而如果事务失败则会进行回滚，客户端也不会读到事务中写入的数据
* **A-Availability 可用性** 非故障的节点在合理的时间内返回合理的响应（不是错误和超时的响应）。这里强调的是合理的响应，不能超时，不能出错。注意并没有说“正确”的结果，例如，应该返回 100 但实际上返回了 90，肯定是不正确的结果，但可以是一个合理的结果。
* **P - Partition Tolerance 分区容错性** 当出现**网络分区**后，系统能够继续“履行职责”。这里**网络分区**是指： 一个分布式系统里面，节点组成的网络本来应该是连通的。然而可能因为一些故障（节点间网络连接断开、节点宕机），使得有些节点之间不连通了，整个网络就分成了几块区域，数据就散布在了这些不连通的区域中。

### 一致性、可用性、分区容错性的选择

虽然 CAP 理论定义是三个要素中只能取两个，但放到分布式环境下来思考，我们会发现必须选择 P（分区容忍）要素，因为网络本身无法做到 100% 可靠，有可能出故障，所以分区是一个必然的现象。

如果我们选择了 CA（一致性 + 可用性） 而放弃了 P（分区容错性），那么当发生分区现象时，为了保证 C（一致性），系统需要禁止写入，当有写入请求时，系统返回 error（例如，当前系统不允许写入），这又和 A(可用性) 冲突了，因为 A（可用性）要求返回 no error 和 no timeout。

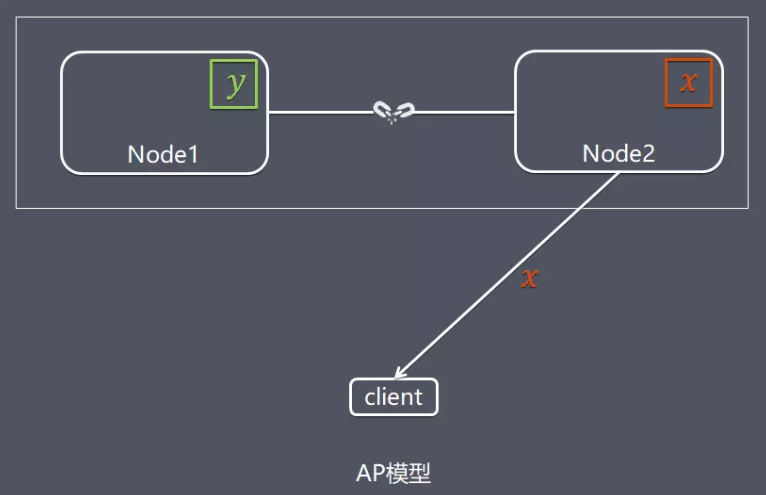
因此，分布式系统理论上不可能选择 CA （一致性 + 可用性）架构，**只能选择 CP（一致性 + 分区容忍性） 或者 AP （可用性 + 分区容忍性）架构，在一致性和可用性做折中选择**。

* **CP - Consistency + Partition Tolerance （一致性 + 分区容忍性）**  
  

如上图所示，因为Node1节点和Node2节点连接中断导致分区现象，Node1节点的数据已经更新到y，但是Node1 和 Node2 之间的复制通道中断，数据 y 无法同步到 Node2，Node2 节点上的数据还是旧数据x。

这时客户端C 访问 Node2 时，Node2 需要返回 Error，提示客户端 “系统现在发生了错误”，这种处理方式违 背了可用性（Availability）的要求，因此 CAP 三者只能满足 CP。

* **AP - Availability + Partition Tolerance （可用性 + 分区容忍性）**



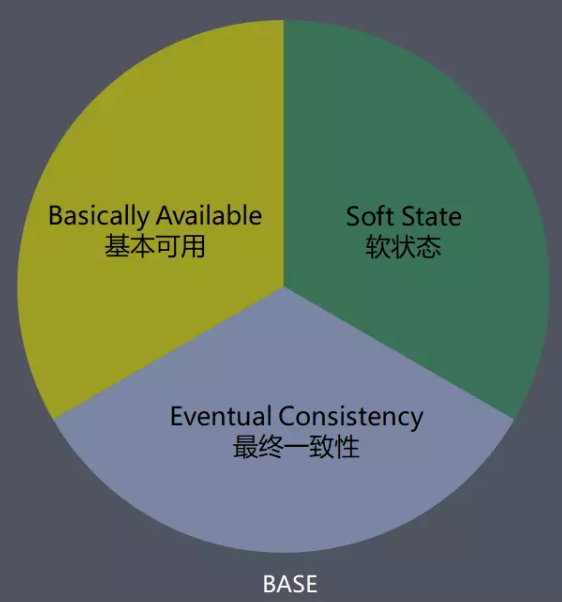
同样是Node2 节点上的数据还是旧数据x，这时客户端C 访问 Node2 时，Node2 将当前自己拥有的数据 x 返回给客户端 了，而实际上当前最新的数据已经是 y 了，这就不满足一致性（Consistency）的要求了，因此 CAP 三者只能满足 AP。

注意：这里 Node2 节点返回 x，虽然不是一个“正确”的结果，但是一个“合理”的结果，因为 x 是旧的数据，并不是一个错乱的值，只是不是最新的数据。

值得补充的是，CAP理论告诉我们**分布式系统只能选择AP或者CP**，但实际上并不是说整个系统只能选择AP或者CP，在 CAP 理论落地实践时，我们需要将系统内的数据按照不同的应用场景和要求进行分类，每类数据选择不同的策略（CP 还是 AP），而不是直接限定整个系统所有数据都是同一策略。

另外，只能选择CP或者AP是指系统发生分区现象时无法同时保证C（一致性）和A（可用性），但不是意味着什么都不做，当分区故障解决后，系统还是要保持保证CA。

## 4.CAP理论的延伸 –BASE理论



BASE 是指基本可用（Basically Available）、软状态（ Soft State）、最终一致性（ Eventual Consistency），核心思想是即使无法做到强一致性（CAP 的一致性就是强一致性），但应用可以采用适合的方式达到最终一致性。

* **BA - Basically Available 基本可用** 分布式系统在出现故障时，允许损失部分可用性，即保证核心可用。

这里的关键词是“**部分**”和“**核心**”，实际实践上，哪些是核心需要根据具体业务来权衡。例如登录功能相对注册功能更加核心，注册不了最多影响流失一部分用户，如果用户已经注册但无法登录，那就意味用户无法使用系统，造成的影响范围更大。

 **S - Soft State 软状态** 允许系统存在中间状态，而该中间状态不会影响系统整体可用性。这里的中间状态就是 CAP 理论中的数据不一致。

 **E - Eventual Consistency 最终一致性** 系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。

这里的关键词是“一定时间” 和 “最终”，“**一定时间**”和数据的特性是强关联的，不同业务不同数据能够容忍的不一致时间是不同的。例如支付类业务是要求秒级别内达到一致，因为用户时时关注；用户发的最新微博，可以容忍30分钟内达到一致的状态，因为用户短时间看不到明星发的微博是无感知的。而“**最终**”的含义就是不管多长时间，最终还是要达到一致性的状态。

BASE 理论本质上是对 CAP 的延伸和补充，更具体地说，是对 CAP 中 AP 方案的一个补充：

* CAP 理论是忽略延时的，而实际应用中延时是无法避免的。 这一点就意味着完美的 CP 场景是不存在的，即使是几毫秒的数据复制延迟，在这几毫秒时间间隔内，系统是不符合 CP 要求的。因此 CAP 中的 CP 方案，实际上也是实现了最终一致性，只是“一定时间”是指几毫秒而已。
* AP 方案中牺牲一致性只是指发生分区故障期间，而不是永远放弃一致性。 这一点其实就是 BASE 理论延伸的地方，分区期间牺牲一致性，但分区故障恢复后，系统应该达到最终一致性。

## 5.数据一致性模型

前面介绍的BASE模型提过“强一致性”和“最终一致性”，下面对这些一致性模型展开介绍。

分布式系统通过复制数据来提高系统的可靠性和容错性，并且将数据的不同的副本存放在不同的机器上，由于维护数据副本的一致性代价很高，因此许多系统采用弱一致性来提高性能，下面介绍常见的一致性模型：

* **强一致性** 要求无论更新操作是在哪个数据副本上执行，之后所有的读操作都要能获得最新的数据。对于单副本数据来说，读写操作是在同一数据上执行的，容易保证强一致性。对多副本数据来说，则需要使用分布式事务协议。
* **弱一致性** 在这种一致性下，用户读到某一操作对系统特定数据的更新需要一段时间，我们将这段时间称为"不一致性窗口"。
* **最终一致性** 是弱一致性的一种特例，在这种一致性下系统保证用户最终能够读取到某操作对系统特定数据的更新（读取操作之前没有该数据的其他更新操作）。"不一致性窗口"的大小依赖于交互延迟、系统的负载，以及数据的副本数等。

系统选择哪种一致性模型取决于应用对一致性的需求，所选取的一致性模型还会影响到系统如何处理用户的请求以及对副本维护技术的选择等。后面将基于上面介绍的一致性模型分别介绍分布式事务的解决方案。

## 6.柔性事务

### 柔性事务的概念

在电商等互联网场景下，传统的事务在数据库性能和处理能力上都暴露出了瓶颈。在分布式领域基于CAP理论以及BASE理论，有人就提出了**柔性事务**的概念。

基于BASE理论的设计思想，柔性事务下，在不影响系统整体可用性的情况下(Basically Available 基本可用)，允许系统存在数据不一致的中间状态(Soft State 软状态)，在经过数据同步的延时之后，最终数据能够达到一致。**并不是完全放弃了ACID，而是通过放宽一致性要求，借助本地事务来实现最终分布式事务一致性的同时也保证系统的吞吐**

### 实现柔性事务的一些特性

下面介绍的是实现柔性事务的一些常见特性，这些特性在具体的方案中不一定都要满足，因为不同的方案要求不一样。

**可见性(对外可查询)** 在分布式事务执行过程中，如果某一个步骤执行出错，就需要明确的知道其他几个操作的处理情况，这就需要其他的服务都能够提供查询接口，保证可以通过查询来判断操作的处理情况。

为了保证操作的可查询，需要对于每一个服务的每一次调用都有一个全局唯一的标识，可以是业务单据号（如订单号）、也可以是系统分配的操作流水号（如支付记录流水号）。除此之外，操作的时间信息也要有完整的记录。

**操作幂等性** 幂等性，其实是一个数学概念。幂等函数，或幂等方法，是指可以使用相同参数重复执行，并能获得相同结果的函数。幂等操作的特点是其任意多次执行所产生的影响均与一次执行的影响相同。也就是说，同一个方法，使用同样的参数，调用多次产生的业务结果与调用一次产生的业务结果相同。

之所以需要操作幂等性，是因为为了保证数据的最终一致性，很多事务协议都会有很多重试的操作，如果一个方法不保证幂等，那么将无法被重试。幂等操作的实现方式有多种，如在系统中缓存所有的请求与处理结果、检测到重复操作后，直接返回上一次的处理结果等。

## 7.常见分布式事务的解决方案

### 1. 2PC(二阶提交)方案--强一致性

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，即将事务分为准备阶段与提交阶段。事务的发起者成为协调者，事务的执行者称为参与者。

在分布式系统里，每个节点都知道自己做操的成功与失败，却无法知道其他节点的成功与失败。当一个事务跨多个节点时，为了保证事务的一致性，而引入一个协调者来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提交或者回滚。

二阶提交的算法思路可以概况为：**参与者将操作结果交给协调者，协调者根据所有参与者的反馈情报各个参与者是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

**阶段1：准备阶段**

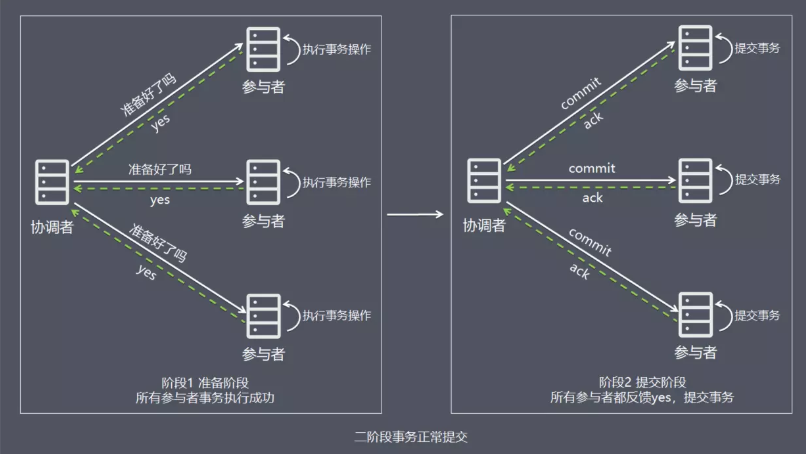
 1、协调者向所有参与者发送事务内容，询问是否可以提交事务，并等待所有参与者答复。

 2、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）。

 3、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。

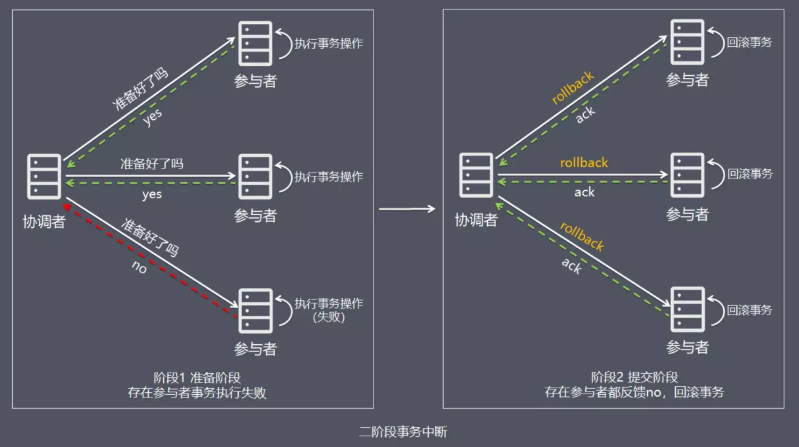
**阶段2：提交阶段** 如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(rollback)消息；否则，发送提交(commit)消息；参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源。(注意:必须在最后阶段释放锁资源) 接下来分两种情况分别讨论提交阶段的过程。

**情况1，当所有参与者均反馈yes，提交事务**：



* 1、协调者向所有参与者发出正式提交事务的请求（即commit请求）。
* 2、参与者执行commit请求，并释放整个事务期间占用的资源。
* 3、各参与者向协调者反馈ack(应答)完成的消息。
* 4、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**：



* 1、协调者向所有参与者发出回滚请求（即rollback请求）。
* 2、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。
* 3、各参与者向协调者反馈ack完成的消息。
* 4、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。

### 2. 3PC(三阶提交)方案

三阶段提交协议，是二阶段提交协议的提升版本，与二阶段提价不同的是，引入了超时机制。同时在协调者与参与者中都引入了超时机制

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，而导致参与者无法知道是否提交或中止的”不确定状态”所产生的可能相当长的延时的问题得以解决

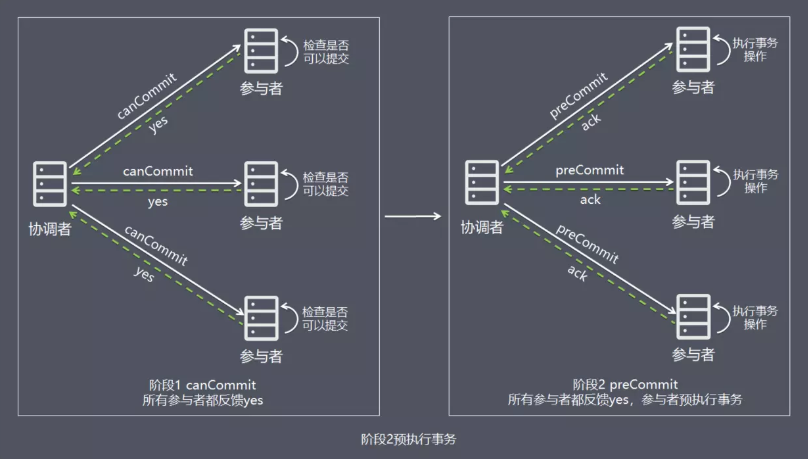
#### 处理流程

**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)，否则则返回no响应：**

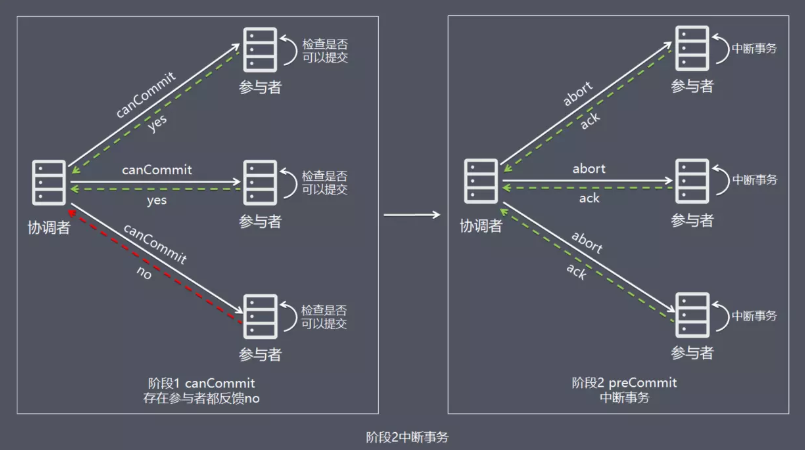
* 1、协调者向所有参与者发出包含事务内容的canCommit请求，询问是否可以提交事务，并等待所有参与者答复。
* 2、参与者收到canCommit请求后，如果认为可以执行事务操作，则反馈yes并进入预备状态，否则反馈no。

**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：**

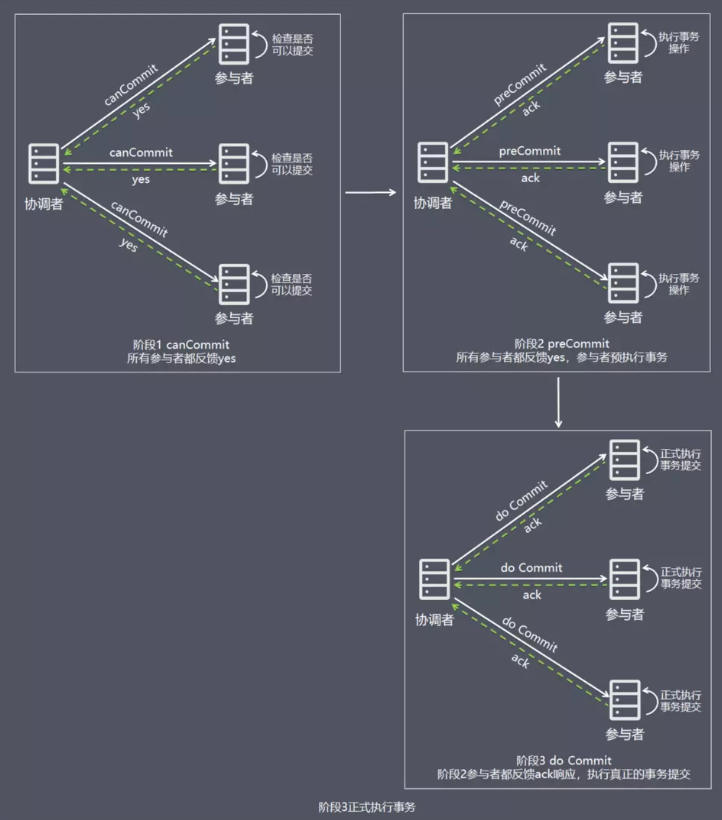


* 1、协调者向所有参与者发出preCommit请求，进入准备阶段。
* 2、参与者收到preCommit请求后，执行事务操作，将undo和redo信息记入事务日志中（但不提交事务）。
* 3、各参与者向协调者反馈ack响应或no响应，并等待最终指令。

**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:** 

* 1、协调者向所有参与者发出abort请求。
* 2、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：



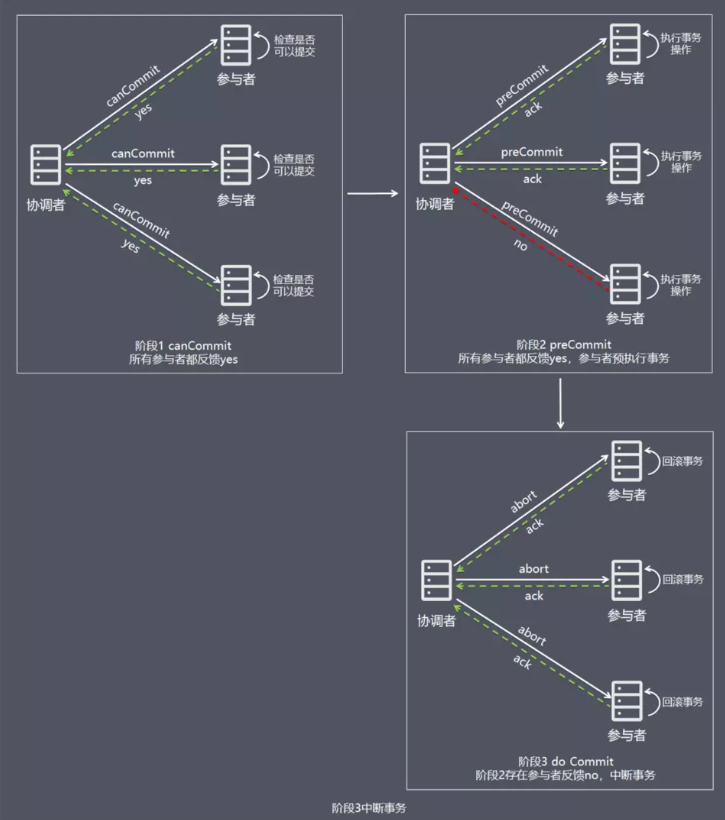
 1、如果协调者处于工作状态，则向所有参与者发出do Commit请求。

 2、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。

 3、各参与者向协调者反馈ack完成的消息。

 4、协调者收到所有参与者反馈的ack消息后，即完成事务提交。

**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：**



* 1、如果协调者处于工作状态，向所有参与者发出abort请求。
* 2、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。
* 3、各参与者向协调者反馈ack完成的消息。
* 4、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，三阶段贴近降低了阻塞范围，在等待超时之后协调者或者参与者都会中断事务。避免了协调者单点问题，阶段3中协调者出现了问题，参与者会继续提交事务

缺点：数据不一致的问题依然存在，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。

### 3. TCC(Try-Confirm-Cancel)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。

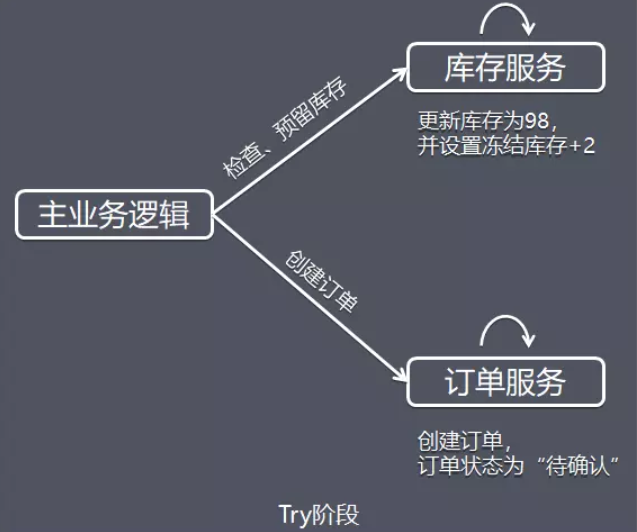
TCC事务的Try、Confirm、Cancel可以理解为SQL事务中的Lock、Commit、Rollback。

#### 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。

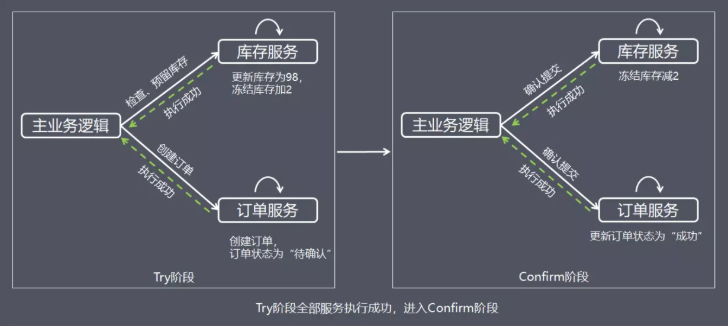
**1、Try 阶段** 从执行阶段来看，与传统事务机制中业务逻辑相同。但从业务角度来看，却不一样。TCC机制中的Try仅是一个初步操作，它和后续的确认一起才能真正构成一个完整的业务逻辑，这个阶段主要完成：

* 完成所有业务检查( 一致性 )
* 预留必须业务资源( 准隔离性 )

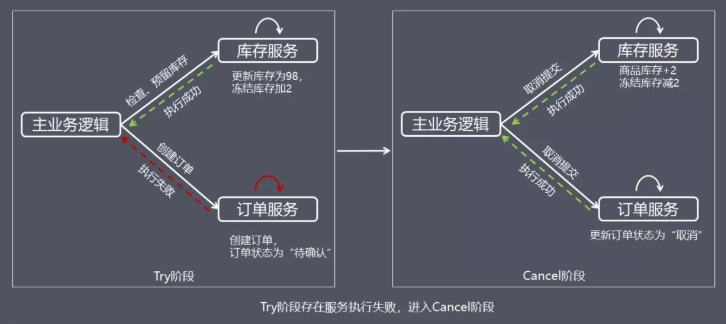
Try 尝试执行业务 TCC事务机制以初步操作（Try）为中心的，确认操作（Confirm）和取消操作（Cancel）都是围绕初步操作（Try）而展开。因此，Try阶段中的操作，其保障性是最好的，即使失败，仍然有取消操作（Cancel）可以将其执行结果撤销。 假设商品库存为100，购买数量为2，这里检查和更新库存的同时，冻结用户购买数量的库存，同时创建订单，订单状态为待确认。

**2、Confirm / 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**



**Cancel：当Try阶段存在服务执行失败， 进入Cancel阶段**



#### 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。

### 4. 本地消息列表 – 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，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列表的事务。

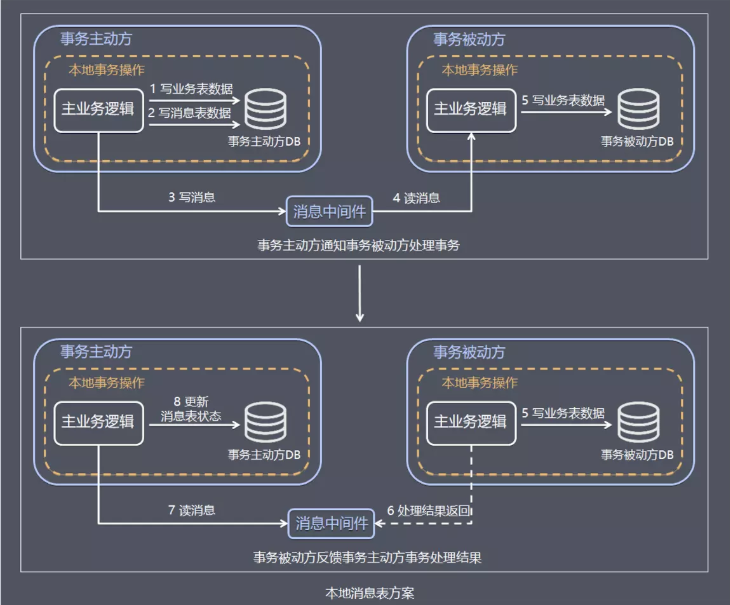
这样设计可以避免”**业务处理成功 + 事务消息发送失败**"，或"**业务处理失败 + 事务消息发送成功**"的棘手情况出现，保证2个系统事务的数据一致性

#### 处理流程

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

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列表，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。**事务主动发在本地事务中处理业务更新操作和写消息表。上面例子中库存服务阶段再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，并将事务消费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：

* 1、当步骤1处理出错，事务回滚，相当于什么都没发生。

2、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，再次发送的消息中间件进行处理。事务被动方消费事务消息重试处理。

* 3、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。
* 4、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。

### 5. 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RocketMQ4.3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。

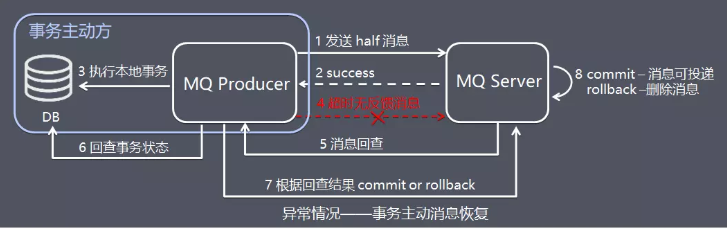
图中2、MQ Server 将消息持久化成功之后，向发送方 ACK 确认消息已经发送成功。

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：



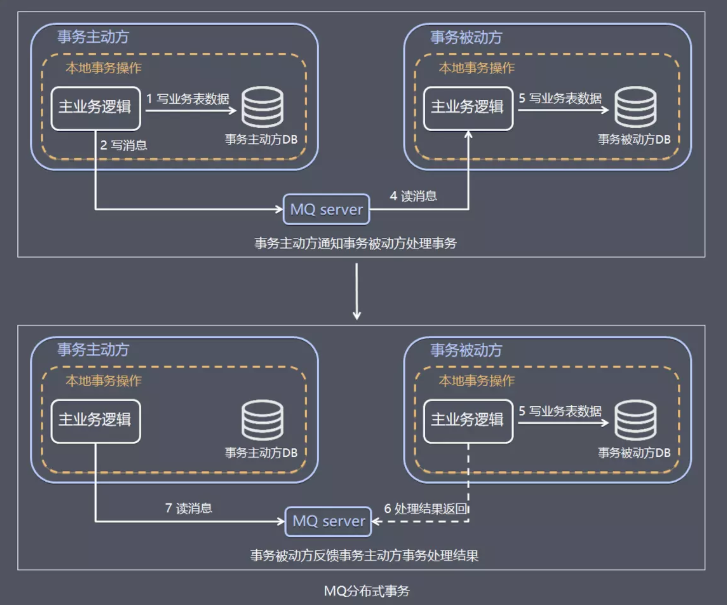
 图中5、MQ Server 对该消息发起消息回查。

 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