作者：网易云  
链接：https://www.zhihu.com/question/64921387/answer/225784480  
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。

关于分布式事务，工程领域主要讨论的是强一致性和最终一致性的解决方案。典型方案包括：

1. 两阶段提交（2PC, Two-phase Commit）方案
2. eBay 事件队列方案
3. TCC 补偿模式
4. 缓存数据最终一致性

**一、一致性理论**

分布式事务的目的是保障分库数据一致性，而跨库事务会遇到各种不可控制的问题，如个别节点永久性宕机，像单机事务一样的ACID是无法奢望的。另外，业界著名的CAP理论也告诉我们，对分布式系统，需要将数据一致性和系统可用性、分区容忍性放在天平上一起考虑。

两阶段提交协议（简称2PC）是实现分布式事务较为经典的方案，但2PC 的可扩展性很差，在分布式架构下应用代价较大，eBay 架构师Dan Pritchett 提出了BASE 理论，用于解决大规模分布式系统下的数据一致性问题。BASE 理论告诉我们：可以通过放弃系统在每个时刻的强一致性来换取系统的可扩展性。

**1、CAP理论**

在分布式系统中，一致性（Consistency）、可用性（Availability）和分区容忍性（Partition Tolerance）3 个要素最多只能同时满足两个，不可兼得。其中，分区容忍性又是不可或缺的。



* 一致性：分布式环境下多个节点的数据是否强一致。
* 可用性：分布式服务能一直保证可用状态。当用户发出一个请求后，服务能在有限时间内返回结果。
* 分区容忍性：特指对网络分区的容忍性。

举例：Cassandra、Dynamo 等，默认优先选择AP，弱化C；HBase、MongoDB 等，默认优先选择CP，弱化A。

**2、BASE 理论**

核心思想：

* **基本可用（Basically  
  Available）**：指分布式系统在出现故障时，允许损失部分的可用性来保证核心可用。
* **软状态（Soft  
  State）**：指允许分布式系统存在中间状态，该中间状态不会影响到系统的整体可用性。
* **最终一致性（Eventual  
  Consistency）**：指分布式系统中的所有副本数据经过一定时间后，最终能够达到一致的状态。

**二、一致性模型**

数据的一致性模型可以分成以下 3 类：

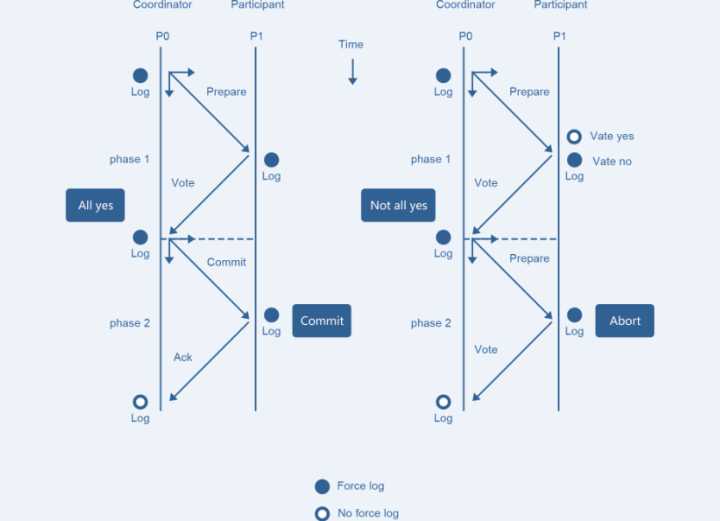
1. **强一致性**：数据更新成功后，任意时刻所有副本中的数据都是一致的，一般采用同步的方式实现。
2. **弱一致性**：数据更新成功后，系统不承诺立即可以读到最新写入的值，也不承诺具体多久之后可以读到。
3. **最终一致性**：弱一致性的一种形式，数据更新成功后，系统不承诺立即可以返回最新写入的值，但是保证最终会返回上一次更新操作的值。

分布式系统数据的强一致性、弱一致性和最终一致性可以通过[Quorum NRW算法](https://link.zhihu.com/?target=https%3A//en.wikipedia.org/wiki/Quorum_%28distributed_computing%29)分析。

**三、分布式事务解决方案**

**1、2PC方案——强一致性**

2PC的核心原理是通过提交分阶段和记日志的方式，记录下事务提交所处的阶段状态，在组件宕机重启后，可通过日志恢复事务提交的阶段状态，并在这个状态节点重试，如Coordinator重启后，通过日志可以确定提交处于Prepare还是PrepareAll状态，若是前者，说明有节点可能没有Prepare成功，或所有节点Prepare成功但还没有下发Commit，状态恢复后给所有节点下发RollBack；若是PrepareAll状态，需要给所有节点下发Commit，数据库节点需要保证Commit幂等。



2PC方案的问题：

1. 同步阻塞。
2. 数据不一致。
3. 单点问题。

升级的3PC方案旨在解决这些问题，主要有两个改进：

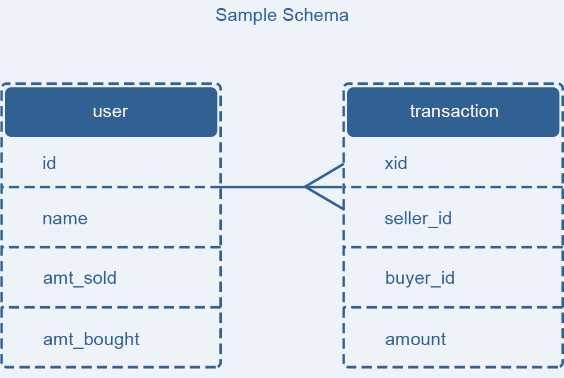
1. 增加超时机制。
2. 两阶段之间插入准备阶段。

但三阶段提交也存在一些缺陷，要彻底从协议层面避免数据不一致，可以采用[Paxos](https://link.zhihu.com/?target=https%3A//en.wikipedia.org/wiki/Paxos_%28computer_science%29)或者[Raft 算法](https://link.zhihu.com/?target=https%3A//raft.github.io/)。

**2、eBay 事件队列方案——最终一致性**

eBay 的架构师Dan Pritchett，曾在一篇解释BASE 原理的论文《[Base：An Acid Alternative](https://link.zhihu.com/?target=http%3A//queue.acm.org/detail.cfm%3Fid%3D1394128)》中提到一个eBay 分布式系统一致性问题的解决方案。它的核心思想是将需要分布式处理的任务通过消息或者日志的方式来异步执行，消息或日志可以存到本地文件、数据库或消息队列，再通过业务规则进行失败重试，它要求各服务的接口是幂等的。

描述的场景为，有用户表user 和交易表transaction，用户表存储用户信息、总销售额和总购买额，交易表存储每一笔交易的流水号、买家信息、卖家信息和交易金额。如果产生了一笔交易，需要在交易表增加记录，同时还要修改用户表的金额。



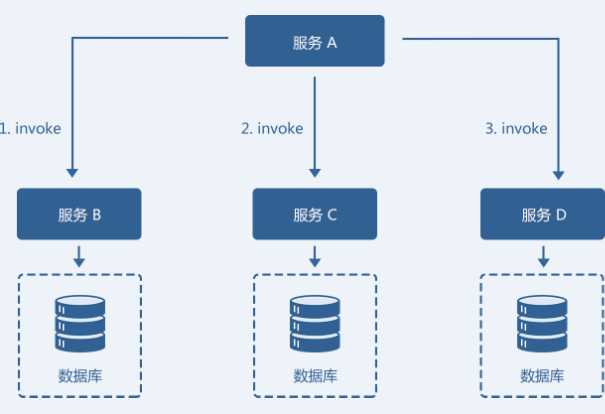
论文中提出的解决方法是将更新交易表记录和用户表更新消息放在一个本地事务来完成，为了避免重复消费用户表更新消息带来的问题，增加一个操作记录表updates\_applied来记录已经完成的交易相关的信息。



这个方案的核心在于第二阶段的重试和幂等执行。失败后重试，这是一种补偿机制，它是能保证系统最终一致的关键流程。

**3、TCC （Try-Confirm-Cancel）补偿模式——最终一致性**

某业务模型如图，由服务 A、服务B、服务C、服务D 共同组成的一个微服务架构系统。服务A 需要依次调用服务B、服务C 和服务D 共同完成一个操作。当服务A 调用服务D 失败时，若要保证整个系统数据的一致性，就要对服务B 和服务C 的invoke 操作进行回滚，执行反向的revert 操作。回滚成功后，整个微服务系统是数据一致的。



实现关键要素：

1. 服务调用链必须被记录下来。
2. 每个服务提供者都需要提供一组业务逻辑相反的操作，互为补偿，同时回滚操作要保证幂等。
3. 必须按失败原因执行不同的回滚策略。

**4、缓存数据最终一致性**

在我们的业务系统中，缓存（Redis 或者Memcached）通常被用在数据库前面，作为数据读取的缓冲，使得I/O 操作不至于直接落在数据库上。以商品详情页为例，假如卖家修改了商品信息，并写回到数据库，但是这时候用户从商品详情页看到的信息还是从缓存中拿到的过时数据，这就出现了缓存系统和数据库系统中的数据不一致的现象。

要解决该场景下缓存和数据库数据不一致的问题我们有以下两种解决方案：

1. 为缓存数据设置过期时间。当缓存中数据过期后，业务系统会从数据库中获取数据，并将新值放入缓存。这个过期时间就是系统可以达到最终一致的容忍时间。
2. 更新数据库数据后同时清除缓存数据。数据库数据更新后，同步删除缓存中数据，使得下次对商品详情的获取直接从数据库中获取，并同步到缓存。

**四、选择建议**

在面临数据一致性问题的时候，首先要从业务需求的角度出发，确定我们对于3 种一致性模型的接受程度，再通过具体场景来决定解决方案。

从应用角度看，分布式事务的现实场景常常无法规避，在有能力给出其他解决方案前，2PC也是一个不错的选择。

对购物转账等电商和金融业务，中间件层的2PC最大问题在于业务不可见，一旦出现不可抗力或意想不到的一致性破坏，如数据节点永久性宕机，业务难以根据2PC的日志进行补偿。金融场景下，数据一致性是命根，业务需要对数据有百分之百的掌控力，建议使用TCC这类分布式事务模型，或基于消息队列的柔性事务框架，这两种方案都在业务层实现，业务开发者具有足够掌控力，可以结合SOA框架来架构，包括Dubbo、Spring Cloud等（题主的标签写了Dubbo）。

**利益相关：**

以上内容，基本都摘自网易云基础服务架构团队著《[云原生应用架构实践](https://link.zhihu.com/?target=https%3A//item.jd.com/12219496.html)》，更多内容，譬如分布式事务的核心——分布式锁的解析，可以参阅图书。部分内容预览可以参考[这里](https://link.zhihu.com/?target=http%3A//blog.163yun.com/archives/1863)。



## 五、CSDN博客

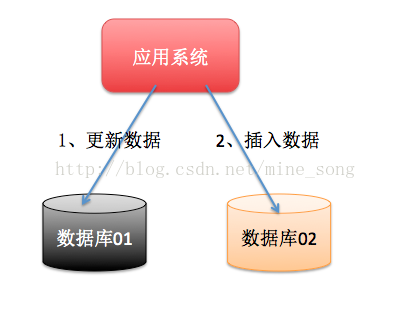
## 1、什么是分布式事务

分布式事务就是指事务的参与者、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。以上是百度百科的解释，简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

## 2、分布式事务的产生的原因

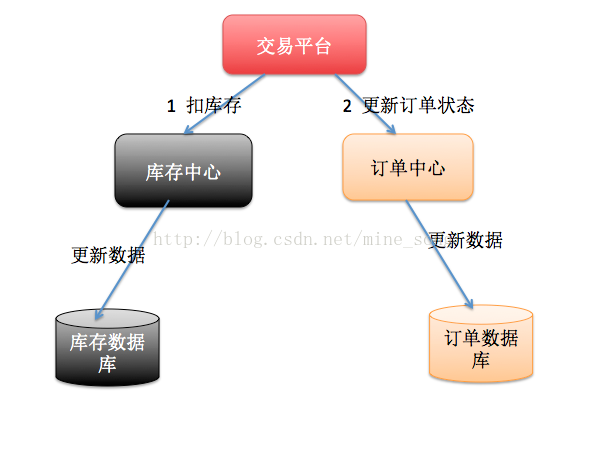
### 2.1、数据库分库分表

当数据库单表一年产生的数据超过1000W，那么就要考虑分库分表，具体分库分表的原理在此不做解释，以后有空详细说，简单的说就是原来的一个数据库变成了多个数据库。这时候，如果一个操作既访问01库，又访问02库，而且要保证数据的一致性，那么就要用到分布式事务。



### 2.2、应用SOA化

所谓的SOA化，就是业务的服务化。比如原来单机支撑了整个电商网站，现在对整个网站进行拆解，分离出了订单中心、用户中心、库存中心。对于订单中心，有专门的数据库存储订单信息，用户中心也有专门的数据库存储用户信息，库存中心也会有专门的数据库存储库存信息。这时候如果要同时对订单和库存进行操作，那么就会涉及到订单数据库和库存数据库，为了保证数据一致性，就需要用到分布式事务。



以上两种情况表象不同，但是本质相同，都是因为要操作的数据库变多了！

## 3、事务的ACID特性

### 3.1、原子性（A）

所谓的原子性就是说，在整个事务中的所有操作，要么全部完成，要么全部不做，没有中间状态。对于事务在执行中发生错误，所有的操作都会被回滚，整个事务就像从没被执行过一样。

### 3.2、一致性（C）

事务的执行必须保证系统的一致性，就拿转账为例，A有500元，B有300元，如果在一个事务里A成功转给B50元，那么不管并发多少，不管发生什么，只要事务执行成功了，那么最后A账户一定是450元，B账户一定是350元。

### 3.3、隔离性（I）

所谓的隔离性就是说，事务与事务之间不会互相影响，一个事务的中间状态不会被其他事务感知。

### 3.4、持久性（D）

所谓的持久性，就是说一单事务完成了，那么事务对数据所做的变更就完全保存在了数据库中，即使发生停电，系统宕机也是如此。

## 4、分布式事务的应用场景

### 4.1、支付

最经典的场景就是支付了，一笔支付，是对买家账户进行扣款，同时对卖家账户进行加钱，这些操作必须在一个事务里执行，要么全部成功，要么全部失败。而对于买家账户属于买家中心，对应的是买家数据库，而卖家账户属于卖家中心，对应的是卖家数据库，对不同数据库的操作必然需要引入分布式事务。

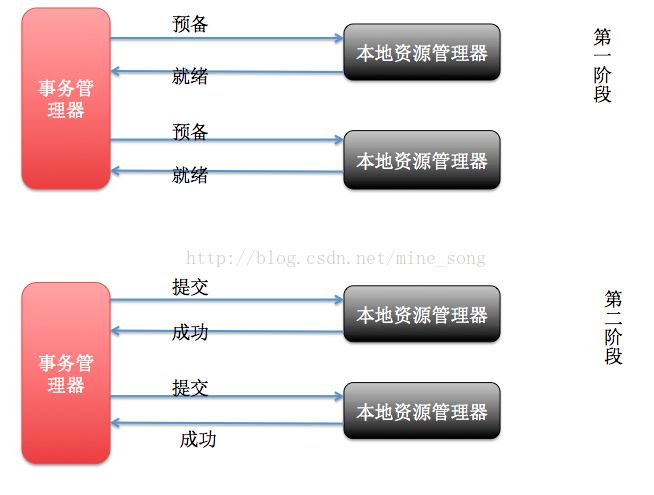
### 4.2、在线下单

买家在电商平台下单，往往会涉及到两个动作，一个是扣库存，第二个是更新订单状态，库存和订单一般属于不同的数据库，需要使用分布式事务保证数据一致性。

## 5、常见的分布式事务解决方案

### 5.1、基于XA协议的两阶段提交

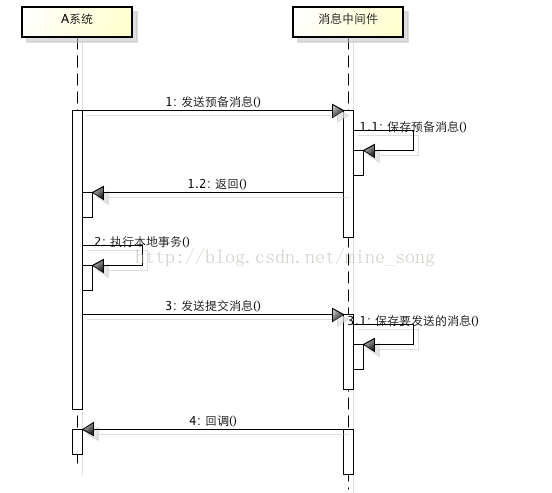
XA是一个分布式事务协议，由Tuxedo提出。XA中大致分为两部分：事务管理器和本地资源管理器。其中本地资源管理器往往由数据库实现，比如Oracle、DB2这些商业数据库都实现了XA接口，而事务管理器作为全局的调度者，负责各个本地资源的提交和回滚。XA实现分布式事务的原理如下：



总的来说，XA协议比较简单，而且一旦商业数据库实现了XA协议，使用分布式事务的成本也比较低。但是，XA也有致命的缺点，那就是性能不理想，特别是在交易下单链路，往往并发量很高，XA无法满足高并发场景。XA目前在商业数据库支持的比较理想，在mysql数据库中支持的不太理想，mysql的XA实现，没有记录prepare阶段日志，主备切换回导致主库与备库数据不一致。许多nosql也没有支持XA，这让XA的应用场景变得非常狭隘。

### 5.2、消息事务+最终一致性

所谓的消息事务就是基于消息中间件的两阶段提交，本质上是对消息中间件的一种特殊利用，它是将本地事务和发消息放在了一个分布式事务里，保证要么本地操作成功成功并且对外发消息成功，要么两者都失败，开源的RocketMQ就支持这一特性，具体原理如下：

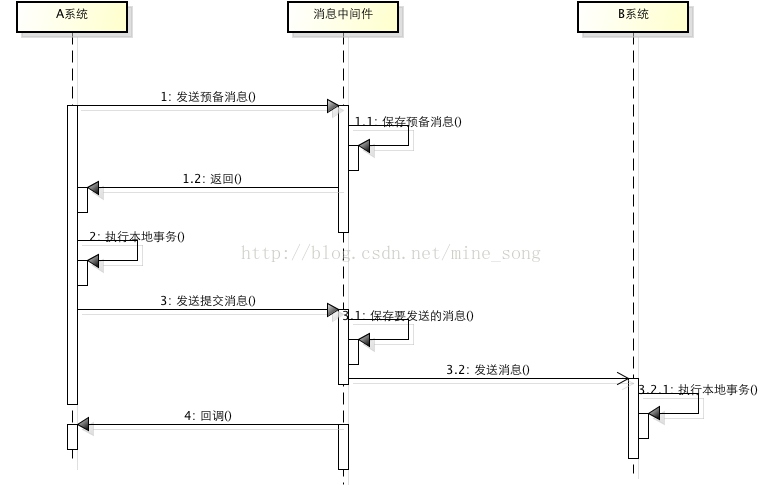


1、A系统向消息中间件发送一条预备消息  
2、消息中间件保存预备消息并返回成功  
3、A执行本地事务  
4、A发送提交消息给消息中间件

通过以上4步完成了一个消息事务。对于以上的4个步骤，每个步骤都可能产生错误，下面一一分析：

* 步骤一出错，则整个事务失败，不会执行A的本地操作
* 步骤二出错，则整个事务失败，不会执行A的本地操作
* 步骤三出错，这时候需要回滚预备消息，怎么回滚？答案是A系统实现一个消息中间件的回调接口，消息中间件会去不断执行回调接口，检查A事务执行是否执行成功，如果失败则回滚预备消息
* 步骤四出错，这时候A的本地事务是成功的，那么消息中间件要回滚A吗？答案是不需要，其实通过回调接口，消息中间件能够检查到A执行成功了，这时候其实不需要A发提交消息了，消息中间件可以自己对消息进行提交，从而完成整个消息事务

基于消息中间件的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A系统的本地操作+发消息）+B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发出来了，这时候B会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到B操作成功，这样就变相地实现了A与B的分布式事务。原理如下：



虽然上面的方案能够完成A和B的操作，但是A和B并不是严格一致的，而是最终一致的，我们在这里牺牲了一致性，换来了性能的大幅度提升。当然，这种玩法也是有风险的，如果B一直执行不成功，那么一致性会被破坏，具体要不要玩，还是得看业务能够承担多少风险。

### 5.3、TCC编程模式

所谓的TCC编程模式，也是两阶段提交的一个变种。TCC提供了一个编程框架，将整个业务逻辑分为三块：Try、Confirm和Cancel三个操作。以在线下单为例，Try阶段会去扣库存，Confirm阶段则是去更新订单状态，如果更新订单失败，则进入Cancel阶段，会去恢复库存。总之，TCC就是通过代码人为实现了两阶段提交，不同的业务场景所写的代码都不一样，复杂度也不一样，因此，这种模式并不能很好地被复用。

## 6、总结

分布式事务，本质上是对多个数据库的事务进行统一控制，按照控制力度可以分为：不控制、部分控制和完全控制。不控制就是不引入分布式事务，部分控制就是各种变种的两阶段提交，包括上面提到的消息事务+最终一致性、TCC模式，而完全控制就是完全实现两阶段提交。部分控制的好处是并发量和性能很好，缺点是数据一致性减弱了，完全控制则是牺牲了性能，保障了一致性，具体用哪种方式，最终还是取决于业务场景。作为技术人员，一定不能忘了技术是为业务服务的，不要为了技术而技术，针对不同业务进行技术选型也是一种很重要的能力

本文链接：<http://www.codeceo.com/article/distributed-transaction.html>