**事务**

在数据库系统中，一个事务是指：由一系列数据库操作组成的一个**完整的逻辑过程**。例如银行转帐，从原账户扣除金额，以及向目标账户添加金额，这两个数据库操作的总和，构成一个完整的逻辑过程，不可拆分。这个过程被称为一个事务，具有ACID特性。

**ACID**：是指在数据库管理系统（DBMS）中，事务(transaction)所具有的四个特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation，又称独立性）、持久性（Durability）。  
l  **原子性**：一个事务(transaction)中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。  
l  **一致性**：在事务开始之前和事务结束以后，数据库的完整性限制没有被破坏。  
l  **隔离性**：当两个或者多个事务并发访问（此处访问指查询和修改的操作）数据库的同一数据时所表现出的相互关系。**事务隔离分为不同级别，包括读未提交(Read uncommitted)、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。**  
l  **持久性**：在事务完成以后，该事务对数据库所作的更改便持久地保存在数据库之中，并且是完全的。

**分布式理论**

当我们的单个数据库的性能产生瓶颈的时候，我们可能会对数**据库进行分区**，这里所说的分区指的是物理分区，分区之后可能**不同的库就处于不同的服务器上**了，这个时候**单个数据库的ACID已经不能适应这种情况了**，而在这种ACID的集群环境下，再想保证集群的ACID几乎是很难达到，或者即使能达到那么效率和性能会大幅下降，最为关键的是再很难扩展新的分区了，这个时候如果再追求集群的ACID会导致我们的系统变得很差，这时我们就需要引入一个新的理论原则来适应这种集群的情况，**就是 CAP 原则或者叫CAP定理，**那么CAP定理指的是什么呢？

**CAP**：

CAP原理指的是，一致性(Consistency)可用性(Availability)分区容忍性(Partitiontolerance)这三个要素最多只能同时实现两点，不可能三者兼顾。这是Brewer教授于2000年提出的，后人也论证了CAP理论的正确性。

l  **一致性**（Consistency） ：对于分布式的存储系统，一个数据往往会存在多份。简单的说，一致性会让客户对数据的修改操作（增/删/改），要么在所有的数据副本（replica）全部成功，要么全部失败。即，修改操作对于一份数据的所有副本（整个系统）而言，是原子（atomic）的操作。**如果一个存储系统可以保证一致性，那么则客户读写的数据完全可以保证是最新的。不会发生两个不同的客户端在不同的存储节点中读取到不同副本的情况。**l  **可用性**（Availability） ：可用性很简单，顾名思义，**就是指在客户端想要访问数据的时候，可以得到响应**。但是注意，**系统可用（Available）并不代表存储系统所有节点提供的数据是一致的。**这种情况，我们仍然说系统是可用的。往往我们会对不同的应用设定一个最长响应时间，超过这个响应时间的服务我们仍然称之为不可用的。  
l  **分区容忍性**（Partition Tolerance） ：**如果你的存储系统只运行在一个节点上，要么系统整个崩溃，要么全部运行良好。**比如，两个存储节点之间联通的网络断开（无论长时间或者短暂的），就形成了分区。一般来讲，为了提高服务质量，同一份数据放置在不同城市非常正常的。因此节点之间形成分区也很正常。

**CAP定理总结**

CAP定理是由加州大学伯克利分校Eric Brewer教授提出来的，他指出WEB服务无法同时满足一下3个属性：

* 一致性(Consistency) ： 客户端知道一系列的操作都会同时发生(生效)
* 可用性(Availability) ： 每个操作都必须以可预期的响应结束
* 分区容忍性(Partition tolerance) ： 即使出现单个组件无法可用,操作依然可以完成

具体地讲在分布式系统中，在任何数据库设计中，一个Web应用至多只能同时支持上面的两个属性。显然，任何横向扩展策略都要依赖于数据分区。因此，设计人员必须在**一致性与可用性之间做出选择。**

分布式系统在遇到任何网络分区故障的时候，仍然能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障。

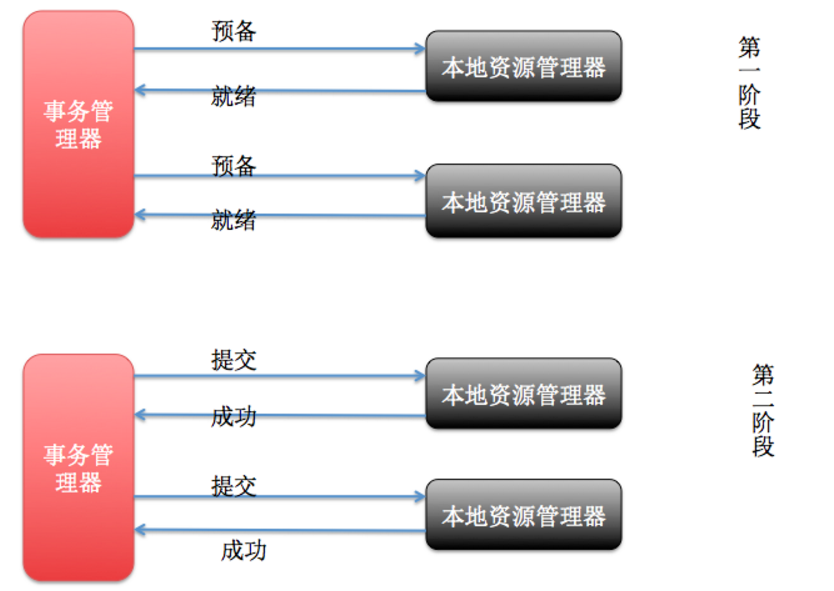
**这个定理在迄今为止的分布式系统中都是适用的！** 为什么这么说呢？

这个时候有同学可能会把数据库的2PC（两阶段提交）搬出来说话了。OK，我们就来看一下数据库的两阶段提交。

对数据库分布式事务有了解的同学一定知道数据库支持的2PC，又叫做 XA Transactions。

MySQL从5.5版本开始支持，SQL Server 2005 开始支持，Oracle 7 开始支持。

其中**，XA 是一个两阶段提交协议，该协议分为以下两个阶段：**

* 第一阶段：**事务协调器**要求**每个涉及到事务的数据库**预提交(precommit)此操作，并反映是否可以提交.
* 第二阶段：**事务协调器**要求**每个数据库**提交数据。
* 

其中，**如果有任何一个数据库否决此次提交**，那么所有数据库都会被**要求回滚它们**在此**事务中的那部分信息**。这样做的缺陷是什么呢?

咋看之下我们可以在数据库分区之间**获得一致性。**如果CAP 定理是对的，**那么它一定会影响到可用性**。（该方案牺牲了一定的可用性换取一致性）

如果说系统的可用性代表的是执行某项操作相关所有组件的可用性的和。那么在两阶段提交的过程中，可用性就代表了涉及到的每一个数据库中可用性的和。

我们假设两阶段提交的过程中每一个数据库都具有99.9%的可用性，那么如果两阶段提交涉及到两个数据库，这个结果就是99.8%。根据系统可用性计算公式，假设每个月43200分钟，**99.9%的可用性就是43157分钟, 99.8%的可用性就是43114分钟，**相当于每个月的宕机时间增加了43分钟。

以上，**可以验证出来，CAP定理从理论上来讲是正确的，CAP我们先看到这里，等会再接着说。**

**BASE**：

在分布式系统中，我们往往追求的是可用性，它的重要性比一致性要高，那么如何实现高可用性呢？ 前人已经给我们提出来了另外一个理论，就是BASE理论，它是用来对CAP定理进行进一步扩充的。BASE理论指的是：

* Basically Available（基本可用）
* Soft state（软状态）
* Eventually consistent（最终一致性）

BASE全称是**BasicallyAvailable（基本可用）, Soft-state（软状态/柔性事务）, Eventually Consistent（最终一致性）。**BASE模型在理论逻辑上是相反于ACID（原子性Atomicity、一致性Consistency、隔离性Isolation、持久性Durability）模型的概念，**它牺牲高一致性，获得可用性和分区容忍性。**

| **基本可用**（Basically Available）。基本可用是指分布式系统在出现故障的时候，允许损失部分可用性，**即保证核心可用**。电商大促时，为了应对访问量激增，部分用户可能会被引导到降级页面，服务层也可能只提供降级服务。这就是损失部分可用性的体现。

| **软状态**（Soft-state）。软状态是指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。分布式存储中一般一份数据至少会有三个副本，**允许不同节点间副本同步的延时就是软状态的体现。**mysql replication的**异步复制也是一种体现。**

l  **最终一致性** （Eventually Consistent）。最终一致性是指：**经过一段时间以后，更新的数据会到达系统中的所有相关节点。**这段时间就被称之为最终一致性的时间窗口  
**BASE理论总结**

BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：**我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性**（Eventual consistency）。

有了以上理论之后，我们来看一下分布式事务的问题。

**分布式事务**

**在分布式系统中，要实现分布式事务，无外乎那几种解决方案。**

**一、两阶段提交（2PC）**

和上一节中提到的数据库XA事务一样，两阶段提交就是使用XA协议的原理，我们可以从下面这个图的流程来很容易的看出中间的一些比如commit和abort的细节。



两阶段提交这种解决方案属于**牺牲了一部分可用性来换取的一致性**。在实现方面，在 .NET 中，可以借助 TransactionScop 提供的 API 来编程实现分布式系统中的两阶段提交，比如WCF中就有实现这部分功能。不过在多服务器之间，需要依赖于DTC来完成事务一致性，Windows下微软搞的有MSDTC服务，Linux下就比较悲剧了。

另外说一句，TransactionScop 默认不能用于异步方法之间事务一致，因为事务上下文是存储于当前线程中的，所以如果是在异步方法，需要显式的传递事务上下文。

**优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。**（其实也不能100%保证强一致）

**缺点： 实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景**，如果分布式系统跨接口调用，目前 .NET 界还没有实现方案。

**二、补偿事务（TCC）**

TCC 其实就是采用的补偿机制，其核心思想是：

**针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。**

它分为三个阶段：

* Try 阶段主要是对业务系统做检测及资源预留
* Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。
* Cancel 阶段主要是在**业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。**

举个例子，假入 Bob 要向 Smith 转账，思路大概是：  
我们有一个本地方法，里面依次调用  
1、首先在 Try 阶段，要先调用远程接口把 Smith 和 Bob 的钱给冻结起来。  
2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。  
3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。

**优点：** 跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

**缺点：** 缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。（在我的理解是，有点try，catch的思想。）

**三、本地消息表（异步确保）**

**本地消息表这种实现方式应该是业界使用最多的，**其核心思想是将**分布式事务拆分成本地事务进行处理，**其基本的设计思想是将远程分布式事务拆分成一系列的本地事务，这种思路是来源于ebay。我们可以从下面的流程图中看出其中的一些细节：



基本思路就是：

消息生产方，需要**额外建一个消息表，并记录消息发送状态**。**消息表和业务数据要在一个事务里提交**，也就是说他们**要在一个数据库**里面。然后消息**会经过MQ发送到消息的消费方。**如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时**如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。**

生产方和消费方定时扫描本地消息表，**把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。**

这种方案遵循BASE理论，采用的是**最终一致性**，笔者认为是这几种方案里面比较适合实际业务场景的，即不会出现像2PC那样复杂的实现**(当调用链很长的时候，2PC的可用性是非常低的)，**也不会像TCC那样**可能出现确认或者回滚不了的情况**。

**优点：** **一种非常经典的实现，避免了分布式事务，实现了最终一致性。**在 .NET中 有现成的解决方案。

**缺点：** **消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理**。

**四、MQ 事务消息**

有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。

以阿里的 RocketMQ 中间件为例，其思路大致为：

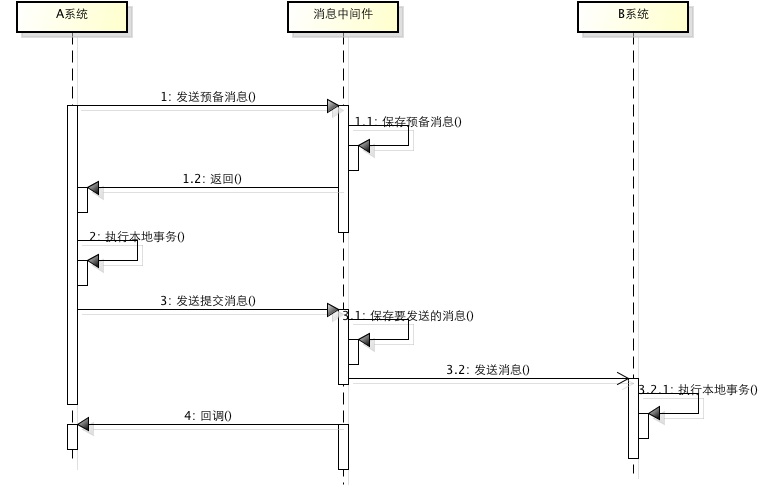
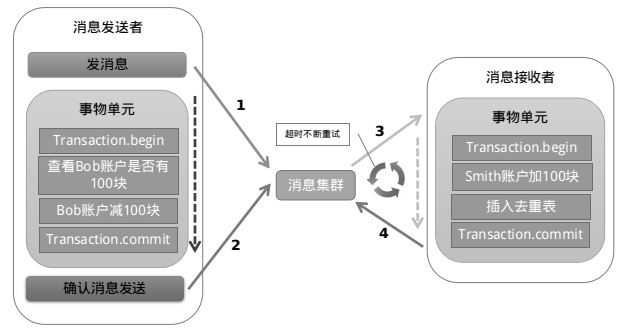
**第一阶段Prepared消息，会拿到消息的地址。  
第二阶段执行本地事务，**

**第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。**

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。

**如果确认消息发送失败了，RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，**

**RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。**这样就保证了消息发送与本地事务同时成功或同时失败。



遗憾的是，RocketMQ并没有 .NET 客户端。有关 RocketMQ的更多消息，大家可以查看[这篇博客](http://www.jianshu.com/p/453c6e7ff81c)

**优点：** **实现了最终一致性，不需要依赖本地数据库事务。**

**缺点：** 实现难度大，主流MQ不支持，没有.NET客户端，RocketMQ事务消息部分代码也未开源。

**总结**

通过本文我们了解到**两个分布式系统的理论，他们分别是CAP和BASE 理论**，同时我们也总结并对比了几种分布式分解方案的优缺点，分布式事务本身是一个技术难题，是没有一种完美的方案应对所有场景的，具体还是要根据业务场景去抉择吧。 然后我们介绍了一种基于本地消息的的分布式事务解决方案CAP。

介绍个场景：

## 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。

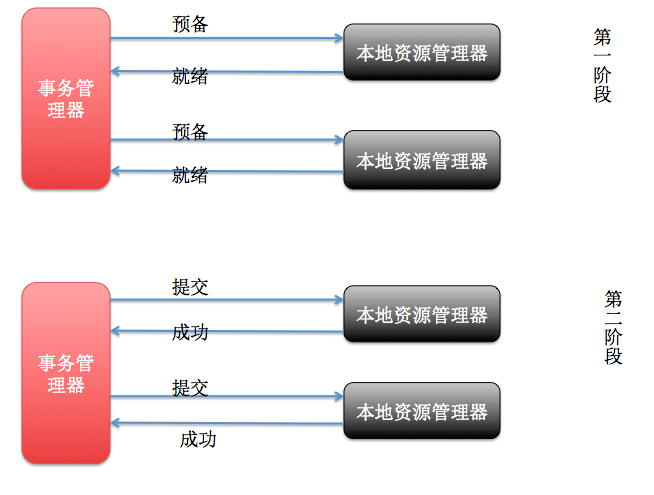
### 4.2、在线下单

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

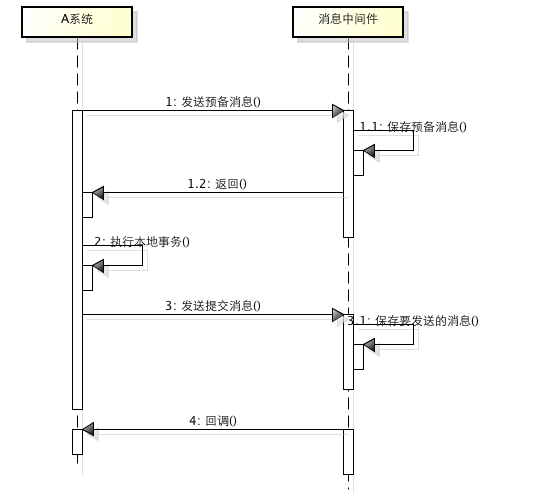
XA是一个分布式事务协议，由Tuxedo提出。XA中大致分为两部分：事务管理器和本地资源管理器。其中**本地资源管理器往往由数据库实现**，比如Oracle、DB2这些商业数据库都实现了XA接口，而事务管理器作为全局的调度者，负责各个本地资源的提交和回滚。XA实现分布式事务的原理如下：



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：

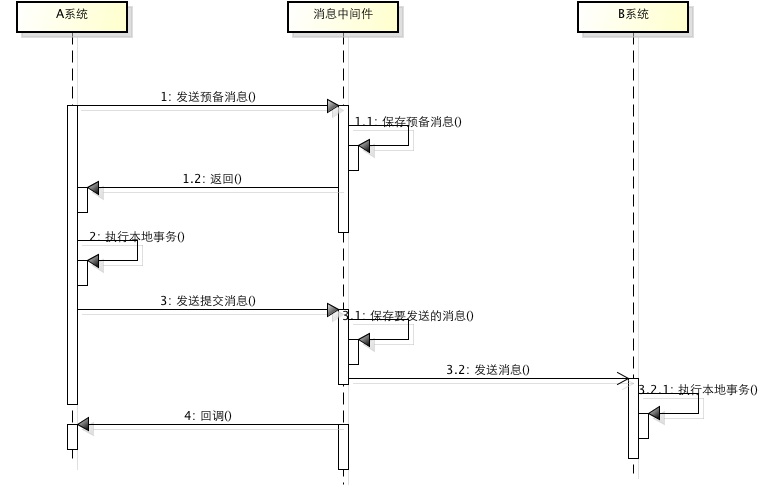


1、A系统向消息中间件发送一条预备消息  
2、消息中间件保存预备消息并返回成功  
3、A执行本地事务  
4、A发送提交消息给消息中间件

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(不是某一时刻就立刻一致的)，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。

举个例子，比如A订单系统，B为库存系统，正常是下单，减库存。那么如果按照二阶段是，如果库存系统因为并发而发生回滚，那么A系统也一并回滚，这样的可用性就很低。

**如果加入这个消息中间件，则认为A系统事务成功了，B系统也会事务成功，B不成功则再发送直到它成功即可。分布式事务是A下单，B减库存，把分布式事务分为，A下单的本地事务+发送消息，然后B再减库存的本地事务。**

### 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！