#### 分布式事务详细介绍

有人的地方，就有江湖

有江湖的地方，就有纷争

# 普通事务

## 定义

普通事务就是一般所说的数据库事务。事务是数据库管理系统执行过程中的一个逻辑单位，由一个有限的数据库操作序列构成。当事务被提交给了DBMS（数据库管理系统），则DBMS（数据库管理系统）需要确保该事务中的所有操作都成功完成且其结果被永久保存在数据库中，如果事务中有的操作没有成功完成，则事务中的所有操作都需要被回滚，回到事务执行前的状态;同时，该事务对数据库或者其他事务的执行无影响，所有的事务都好像在独立的运行。

## 事务的ACID特性

**原子性（Atomicity）**：原子性就是说，在整个事务中的所有操作，要么全部成功，要么全部失败，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。   
**一致性（Consistency）**：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。   
**隔离性（Isolation）**：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（并发）感知。   
**持久性（Durability）**：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。

Ps:

Sql标准隔离级别：

1. read uncommitted；可以看到未提交的数据，举个例子：别人说的话你都相信了，但是可能他只是说说，并不实际做。幻想读、不可重复读和脏读都允许。
2. read committed(Oracle默认隔离级别)；读取提交的数据。但是，可能多次读取的数据结果不一致。允许幻想读、不可重复读，不允许脏读
3. repeatable read(MySQL默认隔离级别)可以重复读取。允许幻想读，不允许不可重复读和脏读

4.serializable序列化。可读，不可写。幻想读、不可重复读和脏读都不允许

**Oracle数据库支持READ COMMITTED 和 SERIALIZABLE这两种事务隔离级别。所以Oracle不支持脏读**

1. 幻想读：指一个线程中的事务读取到了另外一个线程中提交的insert的数据。

2、不可重复读取：指一个线程中的事务读取到了另外一个线程中提交的update的数据。

3、脏读：指一个线程中的事务读取到了另外一个线程中未提交的数据。

## 事务的实现原理

解决ACID问题的两大技术点是：

预写日志（Write-ahead logging） 保证原子性和持久性

锁（locking） 保证各隔离性

Ps：

这里并没有提到一致性，这是因为一致性是应用相关的话题，它的定义一般由业务系统来定义 - 什么样的状态才是一致的？而实现一致性的代码通常在业务层逻辑的代码中得以体现。

预写日志。因为在数据库的复杂数据结构中更新数据是一个比较慢的操作，保证这种操作的原子性和持久性是很困难的。预写日志的工作模式是这样的：在任何事务操作发生之前，先把所有的变化写入一个日志文件并持久化，然后再开始真实的写数据库操作。如果在接下来的操作中系统崩溃，那么我们可以在系统恢复之后检查日志和数据库中的记录，来决定是继续执行完成未尽的任务还是回滚操作 - 把数据库还原到这次事务之前的一个状态。

预写日志一般采用简单顺序日志(sequential file)的写入格式，这样日志写入速度可以很快。这点很重要，因为一般事务操作的吞吐量往往受到日志系统速度的限制。日志的格式会同时记录redo和undo的信息。

    一. 什么是redo（用于前滚数据）

     redo也就是重做日志文件（redo log file），Oracle维护着两类重做日志文件：在线（online）重做日志文件和归档（archived）重做日志文件。这两类重做日志文件都用于恢复；其主要目的是，万一实例失败或介质失败，它们能够恢复数据。

     二. 什么是undo（用于回滚数据）

     从概念上讲，undo正好与redo相对。你对数据执行修改时，数据库会生成undo信息，这样万一你执行的事务或语句由于某种原因失败了，或者如果你用一条ROLLBACK语句请求回滚，就可以利用这些undo信息将数据放回到修改前的样子。

redo用于在失败时重放事务（即恢复事务），undo则用于取消一条语句或一组语句的作用。

锁是大家熟悉的一个话题，在并发环境中通过读写锁来保证操作的互斥性。

根据隔离程度不同，锁的运用也不同，可能会产生一些问题

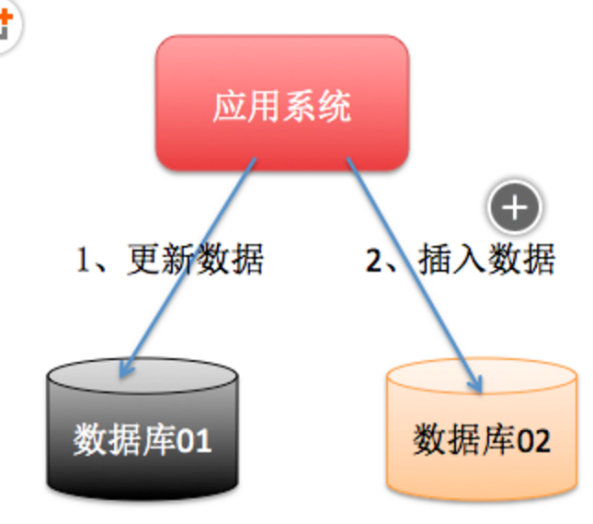
# 分布式事务

分布式事务顾名思义就是在分布式环境下运行的事务，对于分布式事务来说，事务的每个操作步骤是运行在不同机器上的服务的。分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生统一的结果（全部提交或全部回滚）本质上来说，分布式事务就是为了保证不同[数据库](http://lib.csdn.net/base/mysql)的数据一致性。

在现如今的大型互联网平台中，基本上都是采用分布式的SOA架构，所以分布式事务是非常常见的。比如一个电商平台的下单场景，一般对于用户下单会有两个步骤，一是订单业务采取下订单操作，二是库存业务采取减库存操作，但在大型电子商务平台上这两个业务一般会运行在不同的机器上，这就是一个典型的分布式事务场景。还有一个常见的场景就是支付宝向余额宝转账，而支付宝和余额宝不是一个系统，怎么保证这两个系统之间的一致性就是分布式事务所关注的问题。

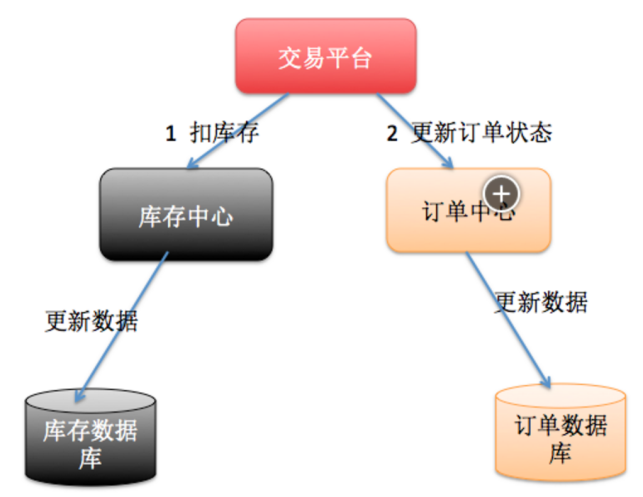
## 分布式事务产生的原因

* 数据库分库



* 应用SOA化

业务的服务化



本质：都是因为要操作的数据库变多了！

## 分布式事务特性

### 分布式系统CAP定律

为了更方便的理解分布式事务，这里插一个分布式系统的CAP定律。

在分布式系统里面有一个CAP定律，这个定理的内容是指的是在一个分布式系统中， Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。

一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）

可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）

分区容错性（P）：分布式系统在遇到任何网络分区故障的时候，仍然需要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。以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。

是不能同时成立的 - 在任意时刻，只有两项能同时成立。对于高流量的网站来说，为了提高系统伸缩性，一般都会牺牲一致性。

### 一致性理论

**强一致：**当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。根据 CAP 理论，这种实现需要牺牲可用性。

**弱一致性：**系统并不保证续进程或者线程的访问都会返回最新的更新过的值。系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。

**最终一致性：**弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。在没有故障发生的前提下，不一致窗口的时间主要受通信延迟，系统负载和复制副本的个数影响。

### 分布式事务特性—最终一致性

在互联网大型分布式平台场景中，为了保障系统的可用性，他们一般会把强一致性的需求转换成最终一致性的需求。所以，对于大部分分布式事务场景，我们仅需要保证最终一致性即可。

### BASE理论

BASE是一种尝试通过牺牲一部分一致性而达到高可用性的原则，ACID原则中要求系统的每个操作之后都是连续的，但是BASE认为系统是可容忍局部的/短时间的不一致。

在基于ACID的事务中，事务是简单可靠的，为了达成这种效果，往往会造成耗尽整个系统的资源造成整体不可用。而BASE的实现是复杂的和业务紧密相关的。BASE 原则体现在（采用这种原则意味着放弃ACID）：

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

### 一致性模型

从这三种一致型的模型上来说，我们可以看到，Weak和Eventually一般来说是异步冗余的，而Strong一般来说是同步冗余的，异步的通常意味着更好的性能，但也意味着更复杂的状态控制；同步意味着简单，但也意味着性能下降。让我们由浅入深，一步一步地来看有哪些技术：

Master-Slave

首先是Master-Slave结构，对于这种结构，Slave一般是Master的备份。在这样的系统中，一般是如下设计的：读写请求都由Master负责。写请求写到Master上后，由Master同步到Slave上。

从Master同步到Slave上，可以使用异步，也可以使用同步，可以使用Master来push，也可以使用Slave来pull。 通常来说是Slave来周期性的pull，所以是最终一致性。这个设计的问题是，如果Master在pull周期内垮掉了，那么会导致这个时间片内的数据丢失。如果你不想让数据丢掉，Slave只能成为Read-Only的方式等Master恢复。

当然，如果可以容忍数据丢掉的话，可以马上让Slave代替Master工作（对于只负责计算的结点来说，没有数据一致性和数据丢失的问题，Master-Slave的方式就可以解决单点问题了） 当然，Master Slave也可以是强一致性的， 比如：当写Master的时候，Master负责先备份，等成功后，再写Slave，两者都成功后返回成功，整个过程是同步的，如果写Slave失败了，那么两种方法，一种是标记Slave不可用报错并继续服务（等Slave恢复后同步Master的数据，可以有多个Slave，这样少一个，还有备份，就像前面说的写三份那样），另一种是回滚自己并返回写失败。（注：一般不先写Slave，因为如果写Master自己失败后，还要回滚Slave，此时如果回滚Slave失败，就得手工订正数据了）可以看到，如果Master-Slave需要做成强一致性有多复杂。

Master-Master

Master-Master，又叫[Multi-master](http://en.wikipedia.org/wiki/Multi-master_replication)，是指一个系统存在两个或多个Master，每个Master都提供read-write服务。这个模型是Master-Slave加强版，数据间同步一般是通过Master间异步完成，所以是最终一致性。 Master-Master的好处是一台Master挂了，别的Master可以正常做读写服务，这个和Master-Slave一样，当数据没有被复制到别的Master上时数据会丢失。很多数据库都支持Master-Master的Replication的机制。

另外，如果多个Master对同一个数据进行修改的时候，这个模型的恶梦就出现了——需要对数据间的冲突进行合并，这非常困难。看看Dynamo的Vector Clock的设计（记录数据的版本号和修改者）就知道这个事并不那么简单，而且Dynamo对数据冲突这个事是交给用户自己搞的。就像SVN源码冲突一样，对于同一行代码的冲突，只能交给开发者自己来处理。

### 数据的高可用

数据服务的高可用是所有企业都想拥有的，但是要想让数据有高可用性，就需要冗余数据写多份。写多份的问题会带来一致性的问题，而一致性的问题又会带来性能问题，这就会陷入一个无解的死循环！这里所谓数据一致性，就是当多个用户试图同时访问一个[数据库](http://lib.csdn.net/base/mysql)时，如果它们的事务同时使用相同的数据，可能会发生以下四种情况：丢失更新、未确定的相关性、不一致的分析和幻像读。

数据副本是分布式系统解决数据丢失异常的唯一手段。

在生产线上用一台服务器来提供数据服务的时候，我会遇到如下的两个问题：   
  
　　1）一台服务器的性能不足以提供足够的能力服务于所有的网络请求。   
  
　　2）我们总是害怕我们的这台服务器停机，造成服务不可用或是数据丢失。   
  
于是我们不得不对我们的服务器进行扩展，加入更多的机器来分担性能上的问题，以及来解决单点故障问题。

通常我们会通过两种手段来扩展我们的数据服务：   
  
　　1）数据分区：就是把数据分块放在不同的服务器上（如：uid % 16，一致性哈希等）。   
  
　　2）数据镜像：让所有的服务器都有相同的数据，提供相当的服务。   
  
　　对于第一种情况，我们无法解决数据丢失的问题，单台服务器出问题时，会有部分数据丢失。所以，数据服务的高可用性只能通过第二种方法来完成——数据的冗余存储。 但是，加入更多的机器，会让我们的数据服务变得很复杂，尤其是跨服务器的事务处理，也就是跨服务器的数据一致性。这个是一个很难的问题。   
　　同时，我们还要考虑性能的因素，如果不考虑性能的话，事务得到保证并不困难，系统慢一点就行了。除了考虑性能外，我们还要考虑可用性，也就是说，一台机器没了，数据不丢失，服务可由别的机器继续提供。 于是，我们需要重点考虑下面的这么几个情况：   
  
　　容灾：数据不丢、结点的Failover   
  
　　数据的一致性：事务处理   
  
　　性能：吞吐量 、 响应时间   
  
　　前面说过，要解决数据不丢，只能通过数据冗余的方法，就算是数据分区，每个区也需要进行数据冗余处理。这就是数据副本：当出现某个节点的数据丢失时可以从副本读到，数据副本是分布式系统解决数据丢失异常的唯一手段。所以，在这篇文章中，简单起见，我们只讨论在数据冗余情况下考虑数据的一致性和性能的问题。简单说来：   
  
　　1）要想让数据有高可用性，就得写多份数据。   
  
　　2）写多份的问题会导致数据一致性的问题。   
  
　　3）数据一致性的问题又会引发性能问题 

## 分布式事务解决方案

### 两阶段和三阶段提交

要想理解分布式事务，不得不先介绍一下两阶段提交协议。先举个简单但不精准的例子来说明：

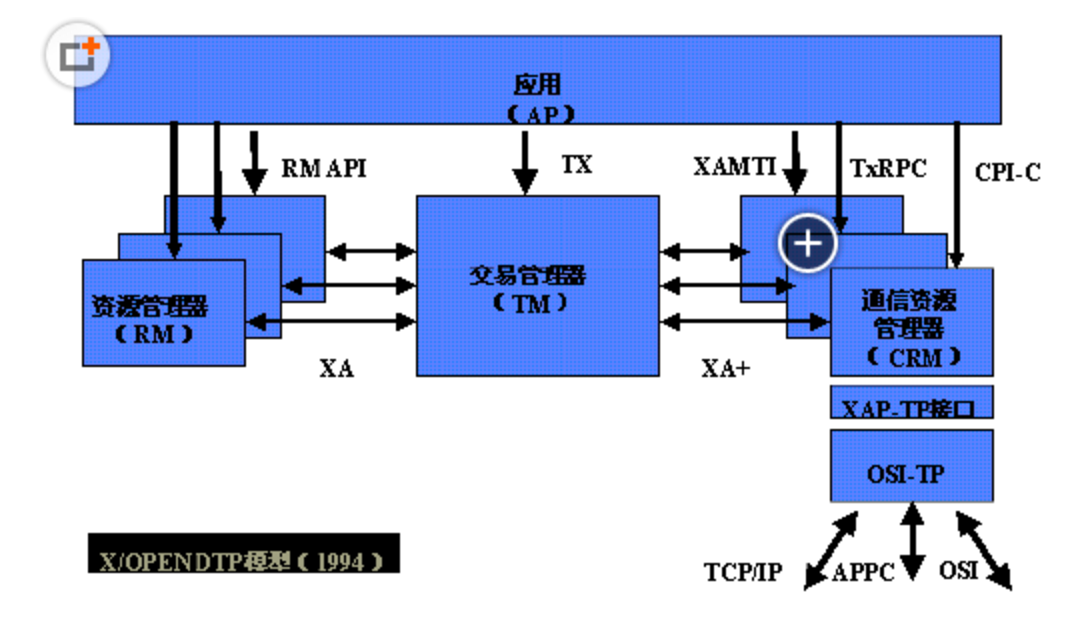
第一阶 段，张老师作为“协调者”，给小强和小明（参与者、节点）发微信，组织他们  
俩明天8 点在学校门口集合，一起去爬山，然后开始等待小强和小明答复。

         第二阶段，如果小强和小明都回答没问题，那么大家如约而至。如果小强或者小明其中  
一人回答说“明天没空，不行”，那么张老师会立即通知小强和小明“爬山活动取消”。  
细心的读者会发现，这个过程中可能有很多问题的。如果小强没看手机，那么张老师会  
一直等着答复，小明可能在家里把爬山装备都准备好了却一直等着张老师确认信息。更严重  
的是，如果到明天8 点小强还没有答复，那么就算“超时”了，那小明到底去还是不去集  
合爬山呢？  
        这就是两阶段提交协议的弊病，所以后来业界又引入了三阶段提交协议来解决该类问题。

所谓的两个阶段是指：

第一阶段：准备阶段(投票阶段)

第二阶段：提交阶段（执行阶段）  
        两阶段提交协议在主流开发语言平台，数据库产品中都有广泛应用和实现的，下面来介绍一下XOpen 组织提供的DTP 模型图：



DTP全称是Distributed Transaction Process，即分布式事务模型。  
TM(Transaction Manager)：事务管理器 协调者  
RM(Resource Manager)：资源管理器 参与者(数据库)  
AP(Application Program)：应用程序  
XA协议：是TM（事务管理器）和RM（资源管理器）之间的接口

目前主流的关系型数据库产品都是实现了XA接口的，JTA(Java Transaction API)是符合X/Open DTP模型的，事务管理器和资源管理器之间也使用了XA协议。

在JavaEE平台下，WebLogic、Webshare等主流商用的应用服务器提供了JTA的实现和支持。而在Tomcat下是没有实现的，这就需要借助第三方的框架Jotm、Automikos等来实现。

两阶段提交协议来实现分布式事务的，看看XA事务成功模型图：



二阶段提交看起来确实能够提供原子性的操作，但是不幸的事，二阶段提交还是有几个缺点的：

1、同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。

2、单点故障。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）

3、数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。

4、二阶段无法解决的问题：协调者再发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

由于二阶段提交存在着诸如同步阻塞、单点问题、脑裂等缺陷，所以，研究者们在二阶段提交的基础上做了改进，提出了三阶段提交。

3PC是2PC的改进版本，将2PC的第一阶段：提交事务阶段一分为二，形成了CanCommit、PreCommit和doCommit三个阶段组成的事务处理协议。

三段提交的核心理念是：在询问的时候并不锁定资源，除非所有人都同意了，才开始锁资源。

在二阶段的基础上加了事务询问的过程(CanCommit)

### CanCommit阶段

3PC的CanCommit阶段其实和2PC的准备阶段很像。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，否则返回No响应。

1.事务询问 协调者向参与者发送CanCommit请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。

2.响应反馈 参与者接到CanCommit请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回Yes响应，并进入预备状态。否则反馈No

### PreCommit阶段

协调者根据参与者的反应情况来决定是否可以进行事务的PreCommit操作。根据响应情况，有以下两种可能。

假如协调者从所有的参与者获得的反馈都是Yes响应，那么就会执行事务的预执行。

1.发送预提交请求 协调者向参与者发送PreCommit请求，并进入Prepared阶段。

2.事务预提交 参与者接收到PreCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。

3.响应反馈 如果参与者成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。

假如有任何一个参与者向协调者发送了No响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。

1.发送中断请求 协调者向所有参与者发送abort请求。

2.中断事务 参与者收到来自协调者的abort请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

### doCommit阶段

该阶段进行真正的事务提交，也可以分为以下两种情况。

执行提交

1.发送提交请求 协调接收到参与者发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送doCommit请求。

2.事务提交 参与者接收到doCommit请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。

3.响应反馈 事务提交完之后，向协调者发送Ack响应。

4.完成事务 协调者接收到所有参与者的ack响应之后，完成事务。

中断事务 协调者没有接收到参与者发送的ACK响应（可能是接受者发送的不是ACK响应，也可能响应超时），那么就会执行中断事务。

1.发送中断请求 协调者向所有参与者发送abort请求

2.事务回滚 参与者接收到abort请求之后，利用其在阶段二记录的undo信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。

3.反馈结果 参与者完成事务回滚之后，向协调者发送ACK消息

4.中断事务 协调者接收到参与者反馈的ACK消息之后，执行事务的中断。

三阶段解决了2个问题

1.解决阻塞问题：将2PC中的第一阶段一分为二，提供了一个CanCommit阶段，此阶段并不锁定资源，这样可以大幅降低了阻塞概率  
2.解决单点问题：在参与者这边也引入了超时机制

3PC虽然解决了2PC存在的2个问题，但是不管是2PC还是3PC都存在数据一致性的问题：  
2PC：比如协调者在只给部分参与者发送了Commit请求，那就会出现部分参与者执行了Commit，部分没有提交，出现不一致问题。  
3PC：一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态，但是这种机制也会导致数据一致性问题。

2PC/3PC在某种场合满足了强一致性，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。

### 一阶段提交

不像两阶段提交那样复杂，一阶段提交非常直白，就是从应用程序向数据库发出提交请求到数据库完成提交或回滚之后将结果返回给应用程序的过程。一阶段提交不需要“协调者”角色，各结点之间不存在协调操作，因此其事务执行时间比两阶段提交要短，但是提交的“危险期”是每一个事务的实际提交时间，相比于两阶段提交，一阶段提交出现在“不一致”的概率就变大了。但是我们必须注意到：只有当基础设施出现问题的时候(如网络中断，当机等)，一阶段提交才可能会出现“不一致”的情况，

相比它的性能优势比二阶段强，但是不一致性非常恐怖。

### 分布式系统中常用最终一致性做法

1. 同步调用，实时返回结果，异步callback确认；
2. 发送事务性消息，消息投递（可以错误补偿重试）。
3. 同步调用返回结果，异步get确认。
4. 发送事务性消息。消息投递失败进入补偿任务重试，调用应用接口的幂等性；

### 提供回滚接口

在服务化架构中，功能X，需要去协调后端的A、B甚至更多的原子服务。那么问题来了，假如A和B其中一个调用失败了，那可怎么办呢？

往往提供了一个BFF层来协调调用A、B服务。如果有些是需要同步返回结果的，我会尽量按照“串行”的方式去调用。如果调用A失败，则不会盲目去调用B。如果调用A成功，而调用B失败，会尝试去回滚刚刚对A的调用操作。

当然，有些时候我们不必严格提供单独对应的回滚接口，可以通过传递参数巧妙的实现。

这样的情况，我们会尽量把可提供回滚接口的服务放在前面。举个例子说明：

每天登录成功后会奖励用户5个积分，但是积分和用户又是两套独立的子系统服务，对应不同的DB，这控制起来就比较麻烦了。解决思路：

1. 把登录和加积分的服务调用放在BFF层一个本地方法中。
2. 当用户请求登录接口时，先执行加积分操作，加分成功后再执行登录操作
3. 如果登录成功，那当然最好了，积分也加成功了。如果登录失败，则调用加积分对应的回滚接口（执行减积分的操作）。

总结：这种方式缺点比较多，通常在复杂场景下是不推荐使用的，除非是非常简单的场景，非常容易提供回滚，而且依赖的服务也非常少的情况。

这种实现方式会造成代码量庞大，耦合性高。而且非常有局限性，因为有很多的业务是无法很简单的实现回滚的，如果串行的服务很多，回滚的成本实在太高。

### 本地消息(表)事务

这种实现方式的思路，其实是源于ebay经典的BASE方案。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。

举个例子。假设系统中有以下两个表

user(id, name, amt\_sold, amt\_bought)

transaction(xid, seller\_id, buyer\_id, amount)

其中user表记录用户交易汇总信息，transaction表记录每个交易的详细信息。

begin;

INSERT INTO transaction VALUES(xid, $seller\_id, $buyer\_id, $amount);

UPDATE user SET amt\_sold = amt\_sold + $amount WHERE id = $seller\_id;

UPDATE user SET amt\_bought = amt\_bought + $amount WHERE id = $buyer\_id;

commit;

即在transaction表中记录交易信息，然后更新卖家和买家的状态。

假设transaction表和user表存储在不同的节点上，那么上述事务就是一个分布式事务。

对于一个分布式事务，我们考虑将其拆分两个独立的子事务，每个子事务都有一张本地消息表。

对于transaction表插入的业务，先启动一个事务，插入transaction表后，并不直接去更新user表，而是将更新表以消息的形式插入到本地消息表message。

begin;

INSERT INTO transaction VALUES(xid, $seller\_id, $buyer\_id, $amount);

put\_to\_queue “update user(“seller”, $seller\_id, amount);

put\_to\_queue “update user(“buyer”, $buyer\_id, amount);

commit;

对于user表更新业务，也需要新建一个message\_applied(msg\_id)表来记录被成功应用的消息，然后发起一个异步任务轮询队列内容进行处理：

for each message in queue

begin;

//先检查此消息是否已处理

SELECT count(\*) as cnt FROM message\_applied WHERE msg\_id = message.id;

if cnt = 0 then

//若没有处理，对user表做更新操作

if message.type = “seller” then

UPDATE user SET amt\_sold = amt\_sold + message.amount WHERE id = message.user\_id;

else

UPDATE user SET amt\_bought = amt\_bought + message.amount WHERE id = message.user\_id;

end

//插入应用的消息，标记此消息已处理

INSERT INTO message\_applied VALUES(message.id);

end

commit;

if 上述事务成功

dequeue message

DELETE FROM message\_applied WHERE msg\_id = message.id;

end

end

我们来仔细分析一下上面代码：

消息队列与transaction使用同一实例，因此第一个事务不涉及分布式操作；

message\_applied与user表在同一个实例中，也能保证一致性；

第二个事务结束后，dequeue message之前系统可能出故障，出故障后系统会重新从消息队列中取出这一消息，但通过message\_applied表可以检查出来这一消息已经被应用过，跳过这一消息实现正确的行为；

最后将已经成功应用，且已经从消息队列中删除的消息从message\_applied表中删除，可以将message\_applied表保证在很小的状态（不清除也是可以的，不影响系统正确性）。由于消息队列与message\_applied在不同实例上，dequeue message之后，将对应message\_applied记录删除之前可能出故障。一但这时出现故障，message\_applied表中会留下一些垃圾内容，但不影响系统正确性，另外这些垃圾内容也是可以正确清理的。

**如果使用表，则对表实时扫描；**

**如果使用消息，则要使用带事务的消息；保证多条消息同时成功；**

**阿里的RocketMQ** [**http://www.jianshu.com/p/453c6e7ff81c**](http://www.jianshu.com/p/453c6e7ff81c)

**顺序问题、重复问题、高可用问题、持久化问题；**

**所以针对消费失败这种情况，最好的办法就是通过报警系统及时发现失败情况然后再人工处理。其实为了交易系统更可靠，我们一般会在类似交易这种高级别的服务代码中，加入详细日志记录的，一旦系统内部引发类似致命异常要及时通过短信通知给业务。同时，应该设计一个报警系统在后台实时扫描和分析此类日志，检查出这种特殊的情况，通过短信及时通知相关人员。**

### 避免分布式--整合业务

适当的业务整合，也是很好的方案；

分清业务数据，将如余额，流水、订单、等信息放到一个库中；将附加信息放到一个库中；

### 消息中间件

重提消息中间件；

如果使用消息要保证接口的幂等性；保证接口的幂等性方式值得研究；

消息重试；消息重试的方式值得研究；

### 对阿里的研究---TCC模型

支付宝分布式事务介绍.ppt

支付宝TCC模型.pdf

alipay\_程立\_分布式事务.pdf

**分布式事务服务简介**

分布式事务服务 (Distributed Transaction Service, DTS) 是一个分布式事务框架，用来保障在大规模分布式环境下事务的最终一致性。DTS 从架构上分为 xts-client 和 xts-server 两部分，前者是一个嵌入客户端应用的 JAR 包，主要负责事务数据的写入和处理；后者是一个独立的系统，主要负责异常事务的恢复。

**核心特性**

传统关系型数据库的事务模型必须遵守 ACID 原则。在单数据库模式下，ACID 模型能有效保障数据的完整性，但是在大规模分布式环境下，一个业务往往会跨越多个数据库，如何保证这多个数据库之间的数据一致性，需要其他行之有效的策略。在 JavaEE 规范中使用 2PC (2 Phase Commit, 两阶段提交) 来处理跨 DB 环境下的事务问题，但是 2PC 是反可伸缩模式，也就是说，在事务处理过程中，参与者需要一直持有资源直到整个分布式事务结束。这样，当业务规模达到千万级以上时，2PC 的局限性就越来越明显，系统可伸缩性会变得很差。基于此，我们采用 BASE 的思想实现了一套类似 2PC 的分布式事务方案，这就是 DTS。DTS在充分保障分布式环境下高可用性、高可靠性的同时兼顾数据一致性的要求，其最大的特点是保证数据最终一致 (Eventually consistent)。

简单的说，DTS 框架有如下特性：

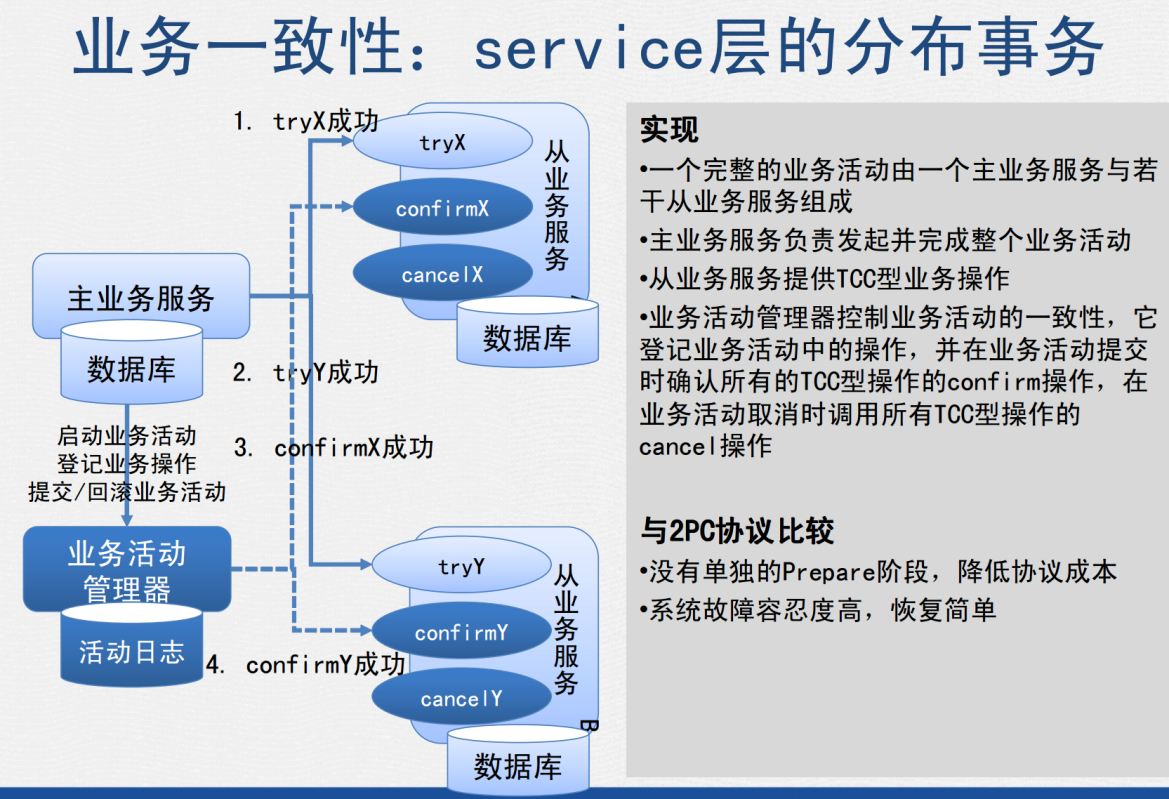
最终一致：事务处理过程中，会有短暂不一致的情况，但通过恢复系统，可以让事务的数据达到最终一致的目标。

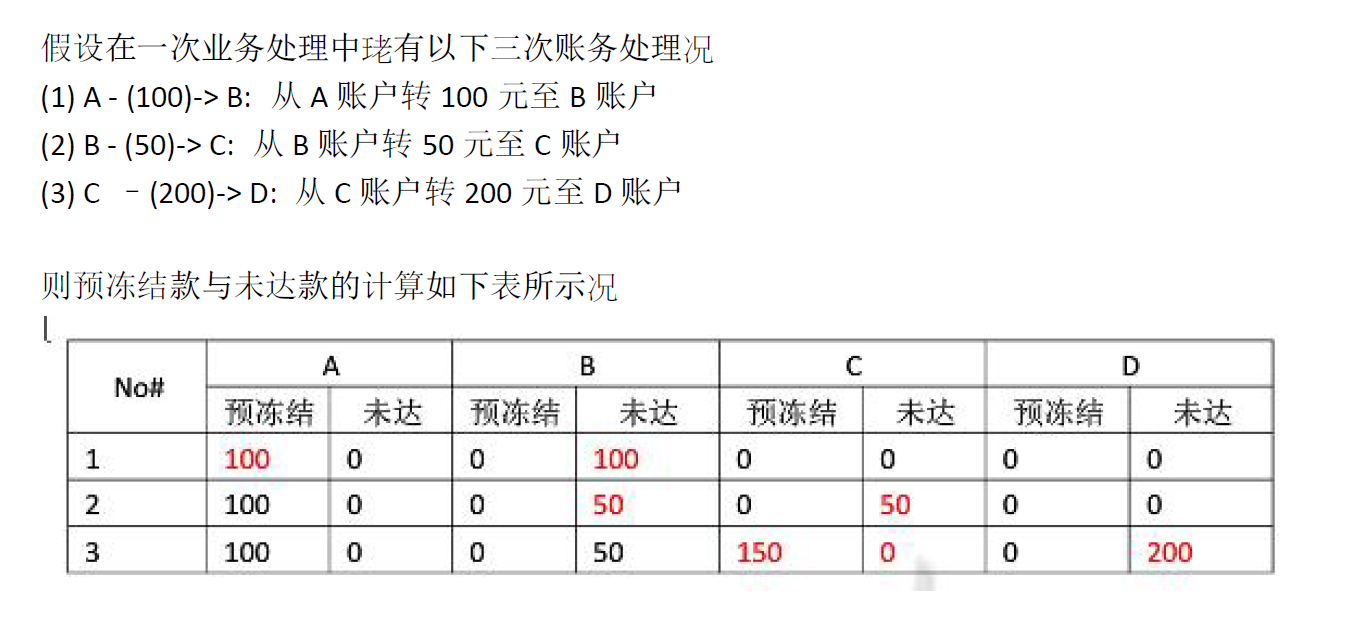
协议简单：DTS 定义了类似 2PC 的标准两阶段接口，业务系统只需要实现对应的接口就可以使用 DTS 的事务功能。

与 RPC 服务协议无关：在 SOA 架构下，一个或多个 DB 操作往往被包装成一个一个的 Service，Service 与 Service 之间通过 RPC 协议通信。DTS 框架构建在 SOA 架构上，与底层协议无关。

与底层事务实现无关： DTS 是一个抽象的基于 Service 层的概念，与底层事务实现无关，也就是说在 DTS 的范围内，无论是关系型数据库 MySQL，[**Oracle**](http://lib.csdn.net/base/oracle)，还是 KV 存储 MemCache，或者列存数据库 [**Hbase**](http://lib.csdn.net/base/hbase)，只要将对其的操作包装成 DTS 的参与者，就可以接入到 DTS 事务范围内。

以下是分布式事务框架的流程图





**实现**

一个完整的业务活动由一个主业务服务与若干从业务服务组成。

主业务服务负责发起并完成整个业务活动。

从业务服务提供 TCC 型业务操作。

业务活动管理器控制业务活动的一致性，它登记业务活动中的操作，并在活动提交时确认所有的两阶段事务的 confirm 操作，在业务活动取消时调用所有两阶段事务的 cancel 操作。”

TCC是分布式事务实现的一种方式  
TRYING 阶段主要是对业务系统做检测及资源预留  
CONFIRMING 阶段主要是对业务系统做确认提交，TRYING阶段执行成功并开始执行CONFIRMING阶段时，默认CONFIRMING阶段是不会出错的。即：只要TRYING成功，CONFIRMING一定成功。  
CANCELING 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。  
而幂等性则是指业务方法调用一次与调用多次的执行返回结果是一样的。

举个支付项目的例子：  
支付系统接收到会员的支付请求后，需要扣减会员账户余额、增加会员积分（暂时假设需要同步实现）增加商户账户余额  
再假设：会员系统、商户系统、积分系统是独立的三个子系统，无法通过传统的事务方式进行处理。  
TRYING阶段：我们需要做的就是会员资金账户的资金预留，即：冻结会员账户的金额（订单金额）  
CONFIRMING阶段：我们需要做的就是会员积分账户增加积分余额，商户账户增加账户余额  
CANCELING阶段：该阶段需要执行的就是解冻释放我们扣减的会员余额  
以上所有的操作需要满足幂等性，幂等性的实现方式可以是：  
1、通过唯一键值做处理，即每次调用的时候传入唯一键值，通过唯一键值判断业务是否被操作，如果已被操作，则不再重复操作  
2、通过状态机处理，给业务数据设置状态，通过业务状态判断是否需要重复执行

# 总结研究方向

服务模式：

可查询操作

幂等性

Tcc操作

可补偿

复核模式：

定期校验

可靠消息

研究代码： tcc-transaction-master、ByteTCC-master

分布式服务对衍生的配套系统要求比较多，特别是我们基于消息、日志的最终一致性方案，需要考虑消息的积压、消费情况、监控、报警等。

《Large-scale Incremental Processing Using Distributed Transactions and Notifications》

翻译为：使用分布式事务和通知的大规模增量处理

提到的 google 的 Percolator在BigTable（非关系型数据库

）基础上，通过两阶段提交实现了跨库，跨表事务以及notifiication框架。