分布式事务解决方案

### 两段式提交

两段式提交主要分为两个投票和提交两个部分；角色有参与者和协调者，参与者是真正操作数据库单点事务的服务(多个)，协调者主要是决定各个事务提交的中间服务(1个)。其中体的执行过程如下：

1. 投票

投票过程主要分为下面三步：

首先协调者向所有参与者发送事务执行请求，并等待参与者反馈事务执行结果

其次参与者收到请求之后，执行事务但不提交，并记录事务日志

最后参与者将执行结果返回给协调者

图示, 示意图

描述已自动生成

图示

描述已自动生成

1. 提交

提交阶段会根据投票结果做不同的操作

2.1如果参与者都回复正常则给所有参与者发送提交commit请求，请求提交事务，参与者收到通知后执行commit操作释放所有资源，给协调者返回事务commit信息

2.2如果一个或者多个消费者失败了，协调者给所有参与者发送rollback通知，请求回滚，参与者收到实物回滚通知后执行rollback操作，释放资源；给协调者返回

2.3如果超时了操作同2.2

图示, 示意图

描述已自动生成

提交阶段会出现和投票阶段相同的三种情况，如果提交成功则事务顺利执行，如果提交失败或者协调者宕机则会引入以下几个问题：

单点问题：如果协调者服务挂了就会引入整个集群的正常运行（这个可以通过watchdog解决）。

同步阻塞：所有参与者都由协调者调度，等待响应的期间，参与者处于阻塞状态不能执行其他操作，效率上达不到很好的效果（采用超时机制可以避免长时间等待阻塞）

数据不一致：提交阶段一部分参与者接收到提交指令，一部分没有收到则会产生数据不一致（互询机制可以解决这个问题）

但是存在一种情况是无法避免的，协调者在通知一个参与者后，协作者和参与者同时挂了，这种情况无法确定当初协作者的请求是提交还是回滚。

### 三段式提交

三段式操作和二段式操作类似，在二段式的基础上增加了一个预询盘阶段，所以总共有以下三个阶段：预询盘(can commit) ，预提交(pre\_commit)，事务提交阶段。

预询盘阶段

1. 协调者会询问参与者是否可以执行事务操作并等待回复
2. 参与者会根据自身情况确认是否能否进入预备状态看，如果可以则进入预备状态并且返回确认信息，否者返回否定信息。

预提交阶段会根据预询盘阶段的结果采取相应操作：

1. 如果参与者都确认则执行与二段式预提交阶段一致的操作
2. 如果参与者超时或者等待请求超时则给所有参与者发送abort指令，请求推出预备状态。

事务提交阶段：

同二段式预提交类似，但是等待超时会默认commit操作，可以解决二段式的等待超时阻塞问题，但是还是无法避免数据的不一致。

### XA事务

XA事务是基于X/Open DTP事务模型开发的，这个模型主要使用了两段提交(2PC - Two-Phase-Commit)来保证分布式事务的完整性。在这个模型里面，有三个角色：

    AP: Application，应用程序。也就是业务层。哪些操作属于一个事务，就是AP定义的。

    TM: Transaction Manager，事务管理器。接收AP的事务请求，对全局事务进行管理，管理事务分支状态，协调RM的处理，通知RM哪些操作属于哪些全局事务以及事务分支等等。这个也是整个事务调度模型的核心部分。

    RM：Resource Manager，资源管理器。一般是数据库，也可以是其他的资源管理器，如消息队列(如JMS数据源)，文件系统等。

是X/OPEN 提出的分布式事务处理规范。XA则规范了TM与RM之间的通信接口，在TM与多个RM之间形成一个双向通信桥梁，从而在多个数据库资源下保证ACID四个特性。目前知名的数据库，如Oracle, DB2,mysql等，都是实现了XA接口的，都可以作为RM。

XA是数据库的分布式事务，强一致性，在整个过程中，数据一张锁住状态，即从prepare到commit、rollback的整个过程中，TM一直把持折数据库的锁，如果有其他人要修改数据库的该条数据，就必须等待锁的释放，存在长事务风险。

**以下的函数使事务管理器可以对资源管理器进行的操作**：  
1）xa\_open,xa\_close：建立和关闭与资源管理器的连接。  
2）xa\_start,xa\_end：开始和结束一个本地事务。  
3）xa\_prepare,xa\_commit,xa\_rollback：预提交、提交和回滚一个本地事务。  
4）xa\_recover：回滚一个已进行预提交的事务。  
5）ax\_开头的函数使资源管理器可以动态地在事务管理器中进行注册，并可以对XID(TRANSACTION IDS)进行操作。  
6）ax\_reg,ax\_unreg；允许一个资源管理器在一个TMS(TRANSACTION MANAGER SERVER)中动态注册或撤消注册。

**XA各个阶段的Mysql处理流程**

图示

描述已自动生成

### TCC事务

Try阶段：

    完成所有业务检查（一致性），预留业务资源(准隔离性)

Confirm阶段：

    确认执行业务操作，不做任何业务检查， 只使用Try阶段预留的业务资源。要满足幂等性。

Cancel阶段：

    取消Try阶段预留的业务资源。

TCC与XA两阶段提交有着异曲同工之妙，下图列出了二者之间的对比

图示, 示意图

描述已自动生成

1) 在阶段1：

    在XA中，各个RM准备提交各自的事务分支，事实上就是准备提交资源的更新操作(insert、delete、update等)；

    而在TCC中，是主业务活动请求(try)各个从业务服务预留资源。

2) 在阶段2：

    XA根据第一阶段每个RM是否都prepare成功，判断是要提交还是回滚。如果都prepare成功，那么就commit每个事务分支，反之则rollback每个事务分支。

    TCC中，如果在第一阶段所有业务资源都预留成功，那么confirm各个从业务服务，否则取消(cancel)所有从业务服务的资源预留请求。

TCC两阶段提交与XA两阶段提交的区别是：

    XA是资源层面的分布式事务，强一致性，在两阶段提交的整个过程中，一直会持有资源的锁。

       XA事务中的两阶段提交内部过程是对开发者屏蔽的，开发者从代码层面是感知不到这个过程的。而事务管理器在两阶段提交过程中，从prepare到commit/rollback过程中，资源实际上一直都是被加锁的。如果有其他人需要更新这两条记录，那么就必须等待锁释放。

    TCC是业务层面的分布式事务，最终一致性，不会一直持有资源的锁。

     TCC中的两阶段提交并没有对开发者完全屏蔽，也就是说从代码层面，开发者是可以感受到两阶段提交的存在。try、confirm/cancel在执行过程中，一般都会开启各自的本地事务，来保证方法内部业务逻辑的ACID特性。其中：

    1、try过程的本地事务，是保证资源预留的业务逻辑的正确性。

    2、confirm/cancel执行的本地事务逻辑确认/取消预留资源，以保证最终一致性，也就是所谓的补偿型事务(Compensation-Based Transactions)。由于是多个独立的本地事务，因此不会对资源一直加锁。

另外，这里提到confirm/cancel执行的本地事务是 **补偿性事务：**

补偿是一个独立的支持ACID特性的本地事务，用于在逻辑上取消服务提供者上一个ACID事务造成的影响，对于一个长事务(long-running transaction)，与其实现一个巨大的分布式ACID事务，不如使用基于补偿性的方案，把每一次服务调用当做一个较短的本地ACID事务来处理，执行完就立即提交

## TCC事务模型 VS DTP事务模型

比较一下TCC事务模型和DTP事务模型，如下所示：

图表, 图示

描述已自动生成

这两张图看起来差别较大，实际上很多地方是类似的!

1、TCC模型中的主业务服务 相当于 DTP模型中的AP，TCC模型中的从业务服务 相当于 DTP模型中的RM,在DTP模型中，应用AP操作多个资源管理器RM上的资源；而在TCC模型中，是主业务服务操作多个从业务服务上的资源。例如航班预定案例中，美团App就是主业务服务，而川航和东航就是从业务服务，主业务服务需要使用从业务服务上的机票资源。不同的是DTP模型中的资源提供者是类似于Mysql这种关系型数据库，而TCC模型中资源的提供者是其他业务服务。

2、TCC模型中，从业务服务提供的try、confirm、cancel接口 相当于 DTP模型中RM提供的prepare、commit、rollback接口

    XA协议中规定了DTP模型中定RM需要提供prepare、commit、rollback接口给TM调用，以实现两阶段提交。

    而在TCC模型中，从业务服务相当于RM，提供了类似的try、confirm、cancel接口。

3、事务管理器

     DTP模型和TCC模型中都有一个事务管理器。不同的是：

     在DTP模型中，阶段1的(prepare)和阶段2的(commit、rollback)，都是由TM进行调用的。

     在TCC模型中，阶段1的try接口是主业务服务调用(绿色箭头)，阶段2的(confirm、cancel接口)是事务管理器TM调用(红色箭头)。这就是 TCC 分布式事务模型的二阶段异步化功能，从业务服务的第一阶段执行成功，主业务服务就可以提交完成，然后再由事务管理器框架异步的执行各从业务服务的第二阶段。这里牺牲了一定的隔离性和一致性的，但是提高了长事务的可用性。

### 消息队列

## 本地消息表

将事务拆分为多个小事务，然后存储到本地文本数据库或者消息队列里并发去请求，一旦失败则进行人工重试或者对账修复。如预付费创建机器，生产机器需要进行以下几个步骤：

1.检查是否有资源

2.进行扣费；

3.创建机器以及各种绑定业务

每个业务都部署到不同的机器上，在2步骤扣费后的操作日志存储到数据库或者mq队列里面去执行，如果有失败或者超时的事务则进行重试一般重试需要一个不断变长的时间间隔直到发货成功，如果一直重试不成功则人工根据数据库消息或者mq队列消息进行人工操作或者对账处理，如计费发现创建机器已经多次失败，则取消重试，进行退费。此外，业务接口需要实现幂等性，如创建机器接口就必须要实现幂等性，否则如果第一次请求成功了，但是将本地数据状态修改失败，导致第二次重试则会多生产出一台机器。

## MQ事务

在RocketMQ中实现了分布式事务，实际上其实是对本地消息表的一个封装，将本地消息表移动到了MQ内部，MQ事务的基本流程如下:

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。消息接受者就能使用这个消息。

如果确认消息失败，在RocketMq Broker中提供了定时扫描没有更新状态的消息，如果有消息没有得到确认，会向消息发送者发送消息，来判断是否提交，在rocketmq中是以listener的形式给发送者，用来处理。

如果消费超时，则需要一直重试，消息接收端需要保证幂等。如果消息消费失败，这个就需要人工进行处理，因为这个概率较低，如果为了这种小概率时间而设计这个复杂的流程反而得不偿失

## Saga事务

模型：由一串子事务(本地事务)的事务链Tn以及补偿n组成（Tn和Cn）。执行顺序可能是

T1 🡪 T2 🡪T3…🡪Tn（最理想情况）也可能T1 🡪 T2 🡪T3…🡪Tn-1🡪 Cn-1 … 🡪C2 🡪C1（最差情况）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| T1 | T2 | T3 | … | Tn |
| C1 | C2 | C3 | … | Cn |

C的补偿方式有两种：1.向前恢复(重试失败的业务) 2.向后恢复(回滚之前的事务)。

要达到事务的完整性，saga对与Tn和Cn必须要有以下几个要求：

1. Tn和Cn是幂等的。
2. Cn必须是能够成功的，如果无法成功则需要人工介入。
3. Tn-Cn和Cn-Tn的执行结果必须是一样的：sub-transaction被撤销。

第一点要求Ti和Ci是幂等的，举个例子，假设在执行Ti的时候超时了，此时我们是不知道执行结果的，如果采用forward recovery策略就会再次发送Ti，那么就有可能出现Ti被执行了两次，所以要求Ti幂等。如果采用backward recovery策略就会发送Ci，而如果Ci也超时了，就会尝试再次发送Ci，那么就有可能出现Ci被执行两次，所以要求Ci幂等。

第二点要求Ci必须能够成功，这个很好理解，因为，如果Ci不能执行成功就意味着整个Saga无法完全撤销，这个是不允许的。但总会出现一些特殊情况比如Ci的代码有bug、服务长时间崩溃等，这个时候就需要人工介入了。

第三点乍看起来比较奇怪，举例说明，还是考虑Ti执行超时的场景，我们采用了backward recovery，发送一个Ci，那么就会有三种情况：

Ti的请求丢失了，服务之前没有、之后也不会执行Ti

Ti在Ci之前执行

Ci在Ti之前执行

对于第1种情况，容易处理。对于第2、3种情况，则要求Ti和Ci是可交换的（commutative)，并且其最终结果都是sub-transaction被撤销。

Saga 模型可以满足事务的三个特性：

原子性：Saga 协调器协调事务链中的本地事务要么全部提交，要么全部回滚。

一致性：Saga 事务可以实现最终一致性。

持久性：基于本地事务，所以这个特性可以很好实现。

但是saga模型不支持隔离性，因此需要业务层控制并发，加锁冻结资源等方式去解决问题。

​Saga 事务和 TCC 事务一样，都是强依靠业务改造，所以要求业务方在设计上要遵循四个策略：

**允许空补偿**：网络异常导致事务的参与方只收到了补偿操作指令，因为没有执行过正常操作，因此要进行空补偿。

**保持幂等性**：事务的正向操作和补偿操作都可能被重复触发，因此要保证操作的幂等性。

**防止资源悬挂**：原因是网络异常导致事务的正向操作指令晚于补偿操作指令到达，则要丢弃本次正常操作，否则会出现资源悬挂问题。

**提供隔离性保证**：遵循“宁可长款，不可短款”设计

​虽然 Saga 和 TCC 都是补偿事务，但是由于提交阶段不同，所以两者也是有不同的：

* Saga 没有Try行为，直接Commit，所以会留下原始事务操作的痕迹，Cancel属于不完美补偿，需要考虑对业务上的影响。TCC Cancel是完美补偿的Rollback，补偿操作会彻底清理之前的原始事务操作，用户是感知不到事务取消之前的状态信息的。
* Saga 的补偿操作通常可以异步执行，TCC的Cancel和Confirm可以跟进需要是否异步化。
* Saga 对业务侵入较小，只需要提供一个逆向操作的Cancel即可；而TCC需要对业务进行全局性的流程改造。
* TCC最少通信次数为2n，而Saga为n（n=子事务的数量）。

关于saga事务详细可查看文章：  
<https://www.jianshu.com/p/e4b662407c66>  
https://servicecomb.apache.org/cn/docs/distributed-transactions-saga-implementation/