

## 了解事务特性

事务是数据库运行的一个逻辑工作单元，在这个工作单元内的一系列SQL命令具有原子性操作的特点，也就是说这一系列SQL指令要么全部执行成功，要么全部回滚不执行。

如果是不执行，那么对于数据库中的数据来说，数据没有发生任何改变。

数据库事务要满足四个需求：

- 原子性 (Atomic)：事务必须是原子工作单元，对数据进行修改，要么全部执行，要么全部都不执行
- 一致性 (Consistent)：事务在完成时，必须使所有数据都保持一致状态，事务结束时所有的内部数据结构都必须是正确的；如果事务是并发多个，系统也必须如同串行事务一样操作。其主要特征是保护性和不变性(Preserving an Invariant)
- 隔离性 (Isolation)：由并发事务所做的修改必须与任何其他并发事务所做的修改隔离
- 持久性 (duration)：事务完成之后，对系统的影响是永久性的，不会被回滚

## X/OpenDTP事务模型

X/Open DTP(X/Open Distributed Transaction Processing Reference Model) 是 X/Open这个组织定义的一套分布式事务的标准，也就是定义了规范和API接口，由各个厂商进行具体的实现。

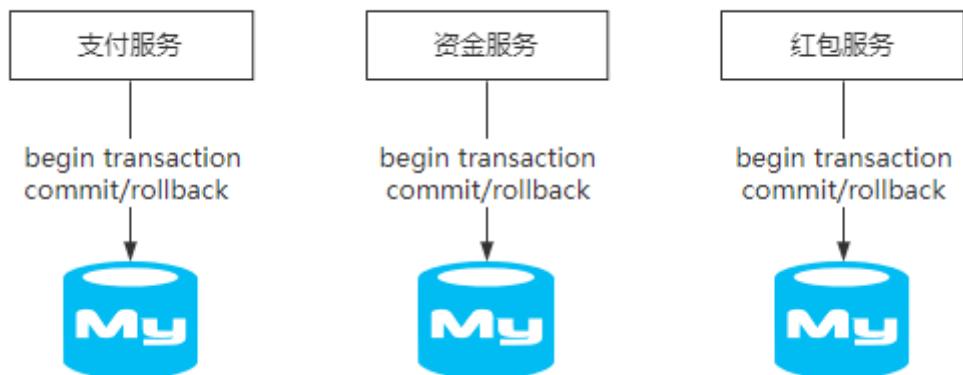
这个标准提出了使用二阶段提交(2PC – Two-Phase-Commit)来保证分布式事务的完整性。

X/Open，即现在的open group，是一个独立的组织，主要负责制定各种行业技术标准。官网地址：<http://www.opengroup.org/>。X/Open组织主要由各大知名公司或者厂商进行支持，这些组织不光遵循X/Open组织定义的行业技术标准，也参与到标准的制定。

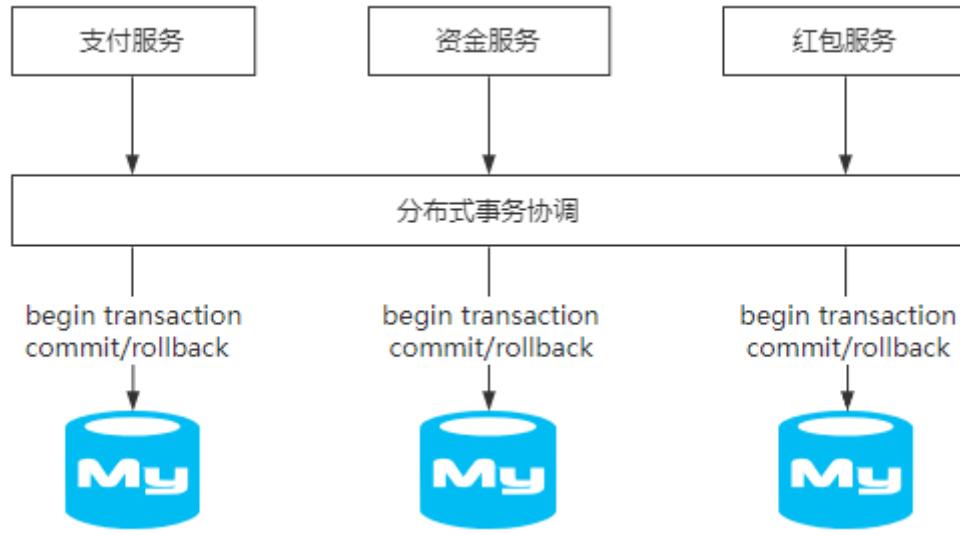
## 二阶段提交模型

了解X/OpenDTP事务模型之前，我们先来了解一下什么是2PC协议

如下图所示，我们知道，在分布式事务中，多个小事务的提交与回滚，只有当前进程知道，其他进程是不清楚的。



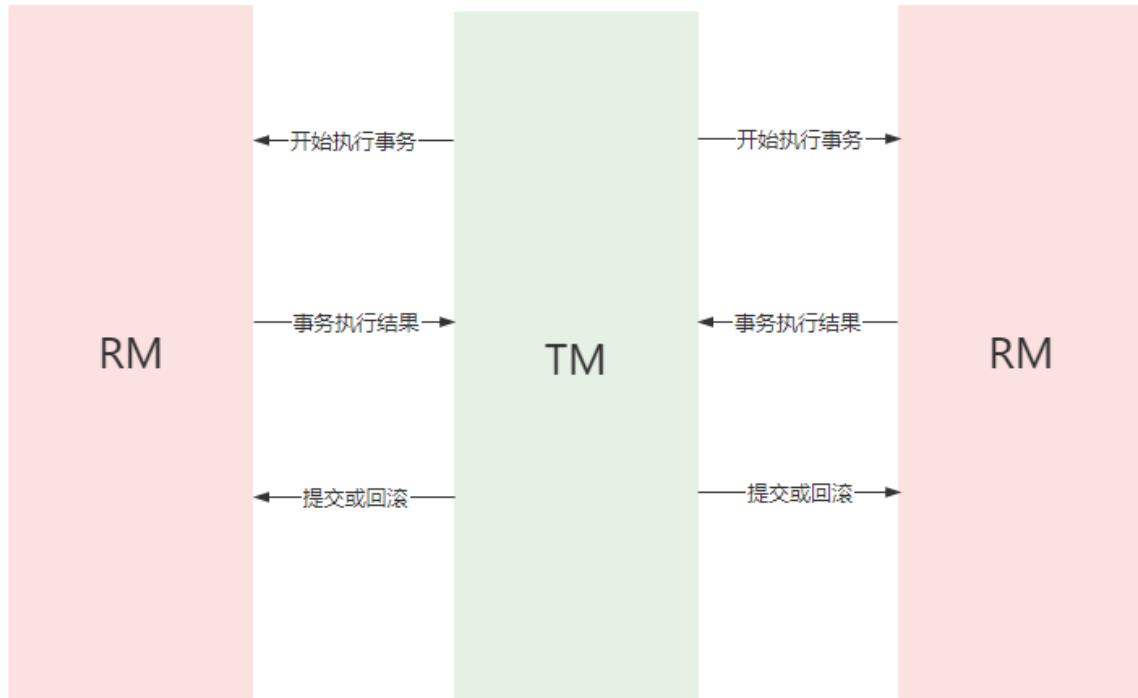
而为了实现多个数据库的事务一致性，就必然需要引入第三方节点来进行事务协调，如下图所示。



从图中可以看出，通过一个全局的分布式事务协调工具，来实现多个数据库事务的提交和回滚，在这样的架构下，事务的管理方式就变成了两个步骤。

1. 第一个步骤，开启事务并向各个数据库节点写入事务日志。
2. 第二个步骤，根据第一个步骤中各个节点的执行结果，来决定对事务进行提交或者回滚。

这就是所谓的2PC提交协议。



2PC提交流程如下：

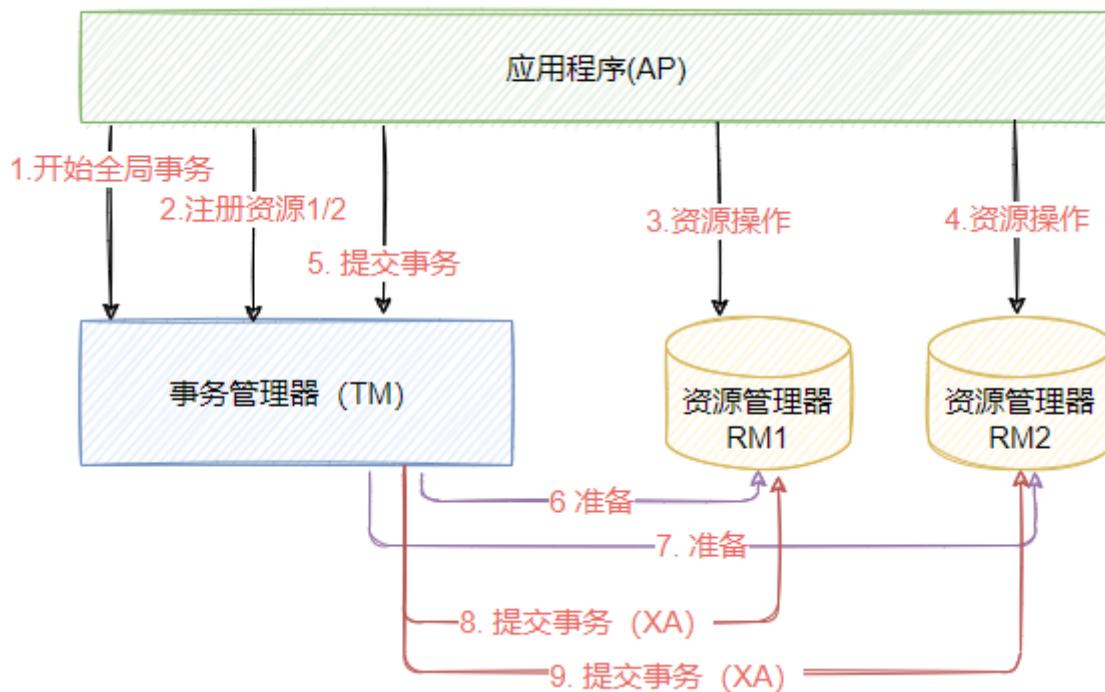
1. 表决阶段：此时 TM（协调者）向所有的参与者发送一个 事务请求，参与者在收到这请求后，如果准备好了(写事务日志) 就会向 TM发送一个 **执行成功** 消息作为回应，告知 TM 自己已经做好了准备，否则会返回一个 **失败** 消息；
2. 提交阶段：TM 收到所有参与者的表决信息，如果所有参与者一致认为可以提交事务，那么 TM就会发送 **提交** 消息，否则发送 **回滚** 消息；对于参与者而言，如果收到 **提交** 消息，就会提交本地事务，否则就会取消本地事务。

## 再次了解X/OpenDTP

基于上述分析，理解了2PC协议后，再来理解X/Open DTP模型。

X/Open DTP (X/Open Distributed Transaction Processing Reference Model)，是 X/Open 这个组织定义的一套分布式事务的标准

X/Open 定义了规范和API接口，由这个厂商进行具体的实现，这个标准提出了使用二阶段提交(2PC - Two-Phase-Commit)来保证分布式事务的完整性，如下图所示。



X/Open DTP模型定义了三个角色和两个协议，其中三个角色分别如下：

- AP (Application Program)，表示应用程序，也可以理解成使用DTP模型的程序
- RM (Resource Manager)，资源管理器，这个资源可以是数据库，应用程序通过资源管理器对资源进行控制，资源管理器必须实现XA定义的接口
- TM (Transaction Manager)，表示事务管理器，负责协调和管理全局事务，事务管理器控制整个全局事务，管理事务的生命周期，并且协调资源。

两个协议分别是：

- XA协议：XA 是X/Open DTP定义的资源管理器和事务管理器之间的接口规范，TM用它来通知和协调相关RM事务的开始、结束、提交或回滚。目前Oracle、Mysql、DB2都提供了对XA的支持；XA接口是双向的系统接口，在事务管理器(TM)以及多个资源管理器之间形成通信的桥梁(XA不能自动提交)

<https://dev.mysql.com/doc/refman/8.0/en/xa.html>

<https://dev.mysql.com/doc/refman/8.0/en/xa-statements.html>

XA协议的语法，主流的数据库都支持 XA协议，从而能够实现跨数据库事务。

```

XA {START|BEGIN} xid [JOIN|RESUME]      --负责开启或者恢复一个事务分支，并且管理XID到调用线程

XA END xid [SUSPEND [FOR MIGRATE]]     --负责取消当前线程与事务分支的关联

XA PREPARE xid                         --负责询问RM 是否准备好了提交事务分支

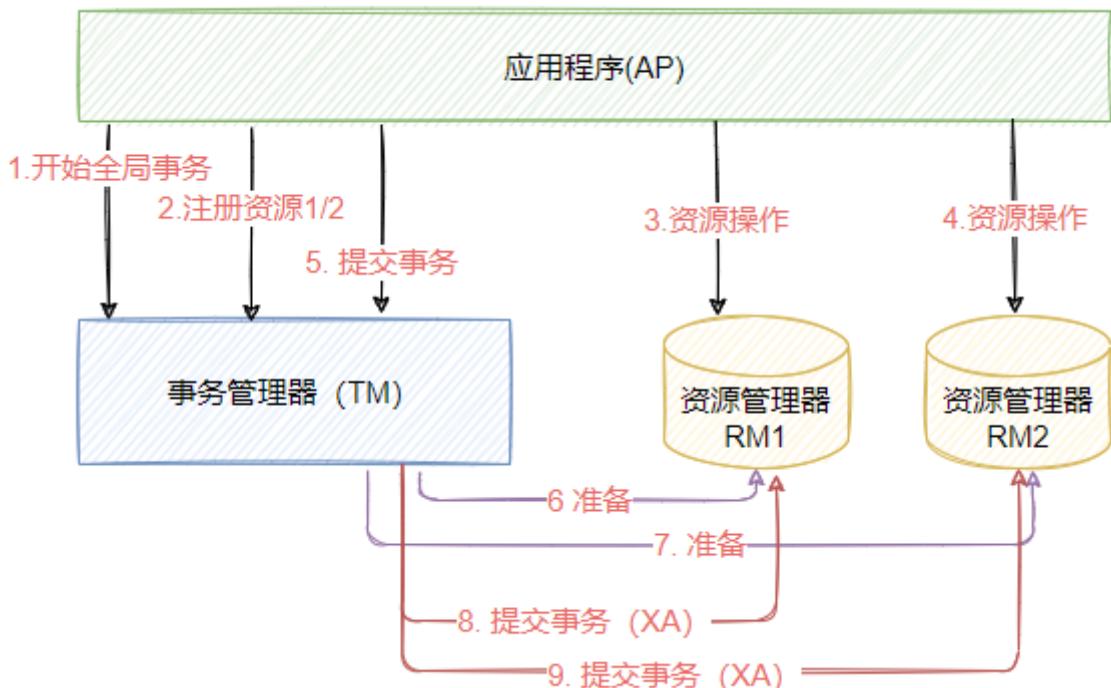
XA COMMIT xid [ONE PHASE]    --告知RM提交事务分支

XA ROLLBACK xid                      --通知RM回滚事务分支

XA RECOVER [CONVERT XID]

```

- TX协议： 全局事务管理器与资源管理器之间通信的接口



## 主流事务框架

- Atomikos, Atomikos是为Java平台提供的开源的事务管理工具，它包含收费和开源两个版本，开源版本基本能满足我们的需求。
- Bitronix, 是一个流行的开源JTA事务管理器实现，你可以使用spring-boot-starter-jta-bitronixstarter为项目添加合适的Bitronix依赖。Bitronix,
- Seata事务，阿里巴巴开源的事务解决方案

## CAP理论和Base理论

大家都听过CAP理论，所谓CAP理论，说的是在分布式架构下的数据一致性问题和性能问题的平衡方案， 它代表三个关键词：

- C: Consistency 一致性 同一数据的多个副本是否实时相同。
- A: Availability 可用性 可用性：一定时间内 & 系统返回一个明确的结果 则称为该系统可用。

- P: Partition tolerance 分区容错性 将同一服务分布在多个系统中，从而保证某一个系统宕机，仍然有其他系统提供相同的服务。

CAP理论告诉我们，在分布式系统中，C、A、P三个条件中我们最多只能选择两个。那么问题来了，究竟选择哪两个条件较为合适呢？

对于一个业务系统来说，可用性和分区容错性是必须要满足的两个条件，并且这两者是相辅相成的。业务系统之所以使用分布式系统，主要原因有两个：

- 提升整体性能 当业务量猛增，单个服务器已经无法满足我们的业务需求的时候，就需要使用分布式系统，使用多个节点提供相同的功能，从而整体上提升系统的性能，这就是使用分布式系统的第一原因。
- 实现分区容错性 单一节点 或 多个节点处于相同的网络环境下，那么会存在一定的风险，万一该机房断电、该地区发生自然灾害，那么业务系统就全面瘫痪了。为了防止这一问题，采用分布式系统，将多个子系统分布在不同的地域、不同的机房中，从而保证系统高可用性。

这说明分区容错性是分布式系统的根本，如果分区容错性不能满足，那使用分布式系统将失去意义。

此外，可用性对业务系统也尤为重要。在大谈用户体验的今天，如果业务系统时常出现“系统异常”、响应时间过长等情况，这使得用户对系统的好感度大打折扣，

在互联网行业竞争激烈的今天，相同领域的竞争者不甚枚举，系统的间歇性不可用会立马导致用户流向竞争对手。因此，我们只能通过牺牲一致性来换取系统的**可用性和分区容错**。

所以这才有了Zookeeper中的基于少数服从多数的2pc落地方案。

因此，也引出了另外一个理论，叫 **Base** 理论。

## Base理论

CAP理论告诉我们一个悲惨但不得不接受的事实—我们只能在C、A、P中选择两个条件。而对于业务系统而言，我们往往选择牺牲一致性来换取系统的可用性和分区容错性。不过这里要指出的是，所谓的“牺牲一致性”并不是完全放弃数据一致性，而是牺牲**强一致性**换取**弱一致性**

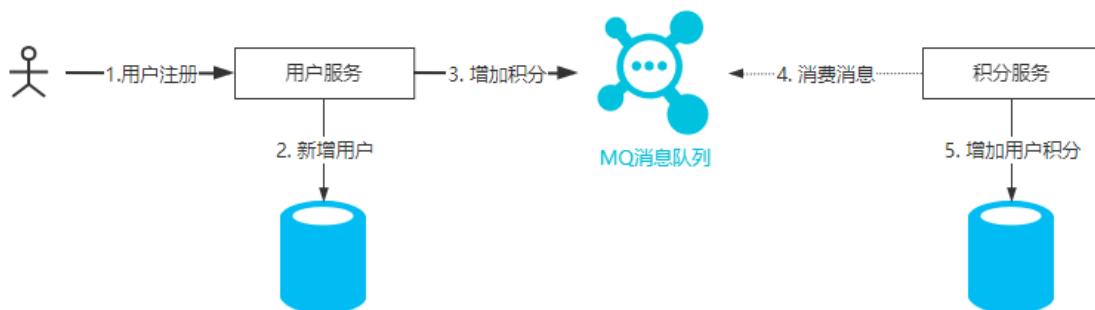
- BA: Basic Available 基本可用
  - 整个系统在某些不可抗力的情况下，仍然能够保证“可用性”，即一定时间内仍然能够返回一个明确的结果。只不过“基本可用”和“高可用”的区别是：
    - “一定时间”可以适当延长 当举行大促时，响应时间可以适当延长
    - 给部分用户返回一个降级页面 给部分用户直接返回一个降级页面，从而缓解服务器压力。但要注意，返回降级页面仍然是返回明确结果。
- S: Soft State: 柔性状态 同一数据的不同副本的状态，可以不需要实时一致。
- E: Eventual Consistency: 最终一致性 同一数据的不同副本的状态，可以不需要实时一致，但一定要保证经过一定时间后仍然是一致的。

# 基于可靠性消息的最终一致性方案

可靠性消息最终一致性方案，是指在多个事务中，当前事务发起方执行完本地事务后，发送一条数据同步消息给到事务参与方。

基于可靠性消息队列，需要保证这个消息一定能够被其他事务参与方接收并执行成功，从而实现多个事务参与者之间的数据最终一致性。

实现原理如下图所示：



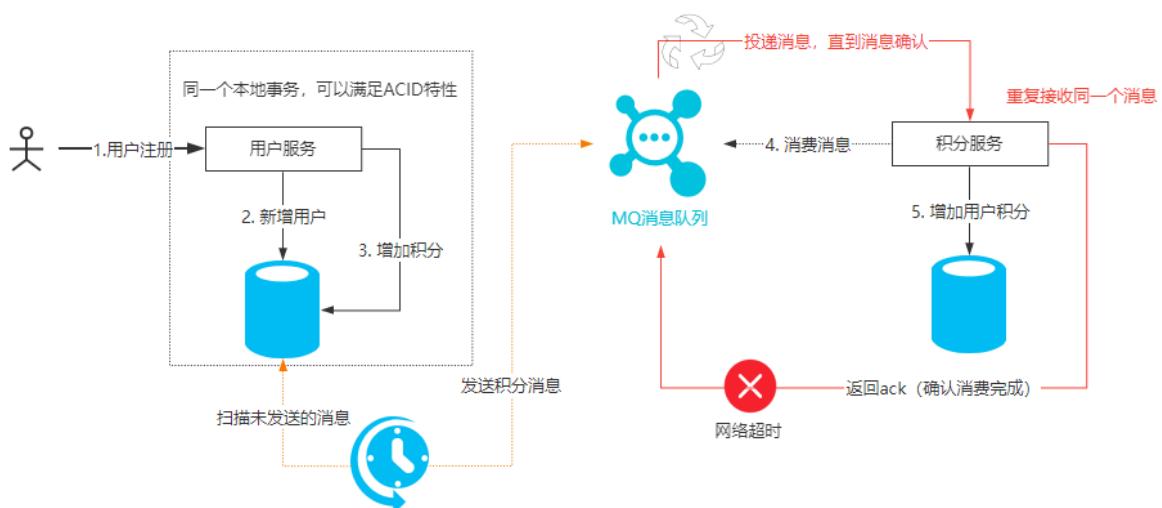
这个方案存在两个问题：

1. 本地事务与消息发送的原子性问题
2. 事务参与方接收消息的可靠性
3. 消息重复消费的问题

## 基于本地消息表实现重发

本地消息表这个方案最初是 eBay 提出的，此方案的核心是通过本地事务保证数据业务操作和消息的一致性，然后通过定时任务将消息发送至消息中间件，待确认消息发送给消费方成功再将消息删除。

下面以注册送积分为例来说明：下例共有两个微服务交互，用户服务和积分服务，用户服务负责添加用户，积分服务负责增加积分。



## 消息重复投递的幂等性保障！

在可靠性消息投递过程中，由于MQ的重试机制，有可能会出现消费者重复收到同一个消息的情况。

因此，我们需要保证消息投递的幂等性，所谓的幂等性，就是MQ重复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；

1. 数据库唯一约束实现幂等
2. 通过tokenid的方式去识别每次请求判断是否重复
3. redis中的setNX
4. 状态机幂等性
5. 本地消息表生成md5实现唯一约束