第六章 分布式事务管理

一、事务的基本概念

1.事务的定义

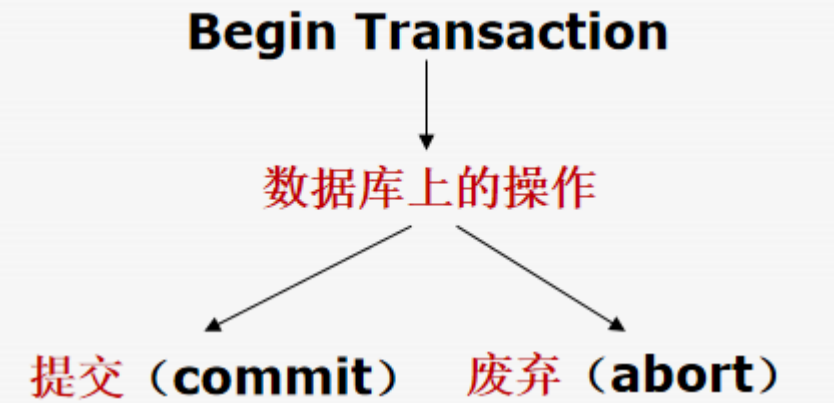
- 任何数据库应用最终都将转换为一系列对数据库进行存取的操作系列
- 为了保证数据库的正确性及操作的有效性，将数据库应用中全部或部分操作序列的执行定义为事务
- 一个事务所包含的所有操作，要么全做，要么全不做，是一个不可分割的整体
- **事务是由若干个为完成某一任务而逻辑相关的操作组成的操作序列，是保证数据库正确性的基本逻辑单元**

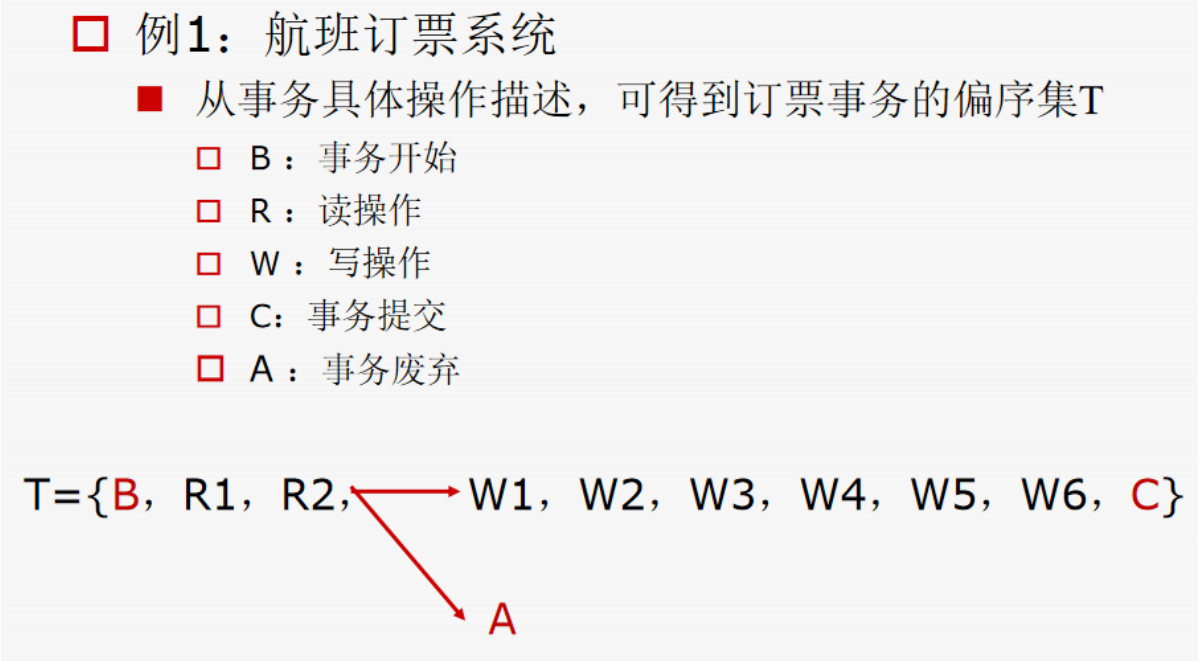
显式声明：应用程序通过事务命令显式地划分事务

隐式声明：系统按照默认规定自动地划分事务

（如果程序中没有显式定义事务，则由系统按照默认规定自动地划分事务，**一句话一个事务**）

2.事务的基本模型





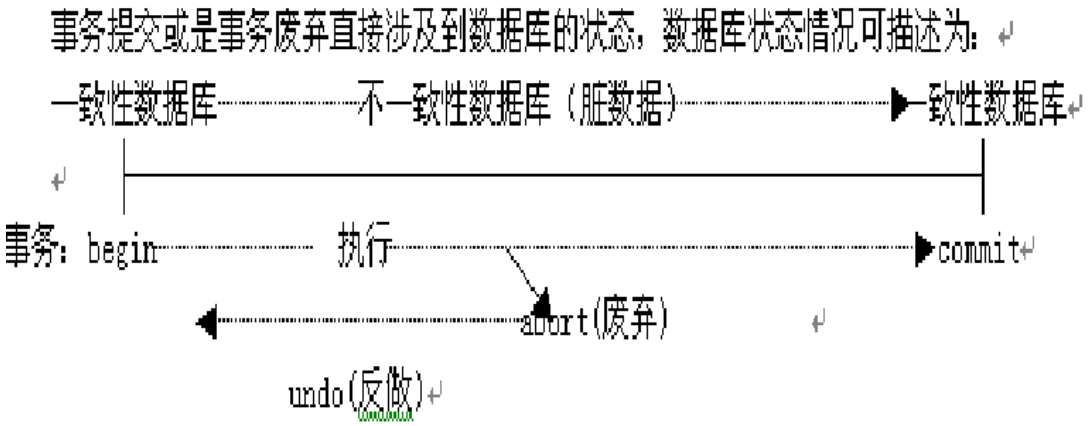
3.事务的基本性质

3.1 原子性

- 事务所包含的操作要么全做，要么全不做
- **事务恢复**：由于输入错误、系统过载、死锁等导致的事务废弃而需要进行的事务的原子性维护
- **故障恢复**：由于系统崩溃（死机、掉电）而导致的事务废弃或者提交结果的丢失而需要进行的原子性维护处理
- **反做**：对于废弃的事务，需要撤销事务已经完成的部分操作，使得数据库恢复到事务执行前的状态
- **重做**：对于已经提交的事务，由于不能保证事务所作的修改已经被更新到外存数据库，因此需要重新执行事务的全部操作

3.2 一致性

- 假如数据库的状态满足所有的完整性约束（参照完整性、实体完整性、用户自定义完整性），则称该数据库是一致的
- 事务的一致性，是**事务执行的结果必须是使数据库从一个一致性状态变化到另一个一致性状态，而不会停留在某种不一致的中间状态上**



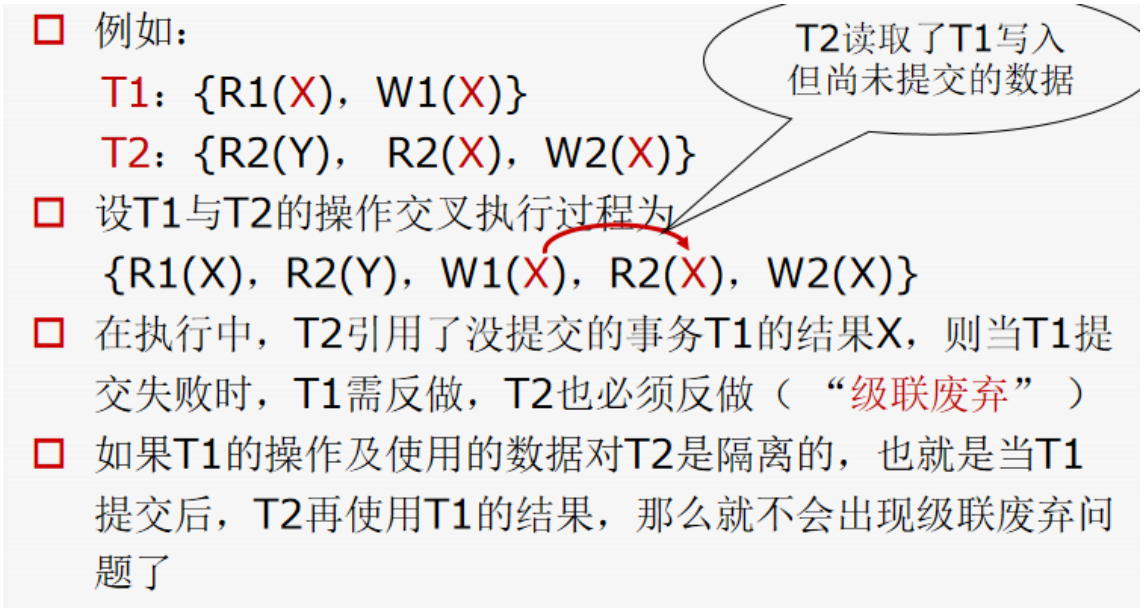
例如在前面的订票事务中，数据库的状态只存在两种情况：

  - 订票事务中的所有操作均被成功执行，使数据库转换为订票后的状态。
  - 由于某种故障使得事务的执行被迫中断，则系统需要撤销订票事务中已执行的操作，也就是对结果部分进行反做，使数据库恢复到订票事务执行前的状态。

在这两种情况下，数据库的状态均为正确的状态，体现了事务操作的一致性。

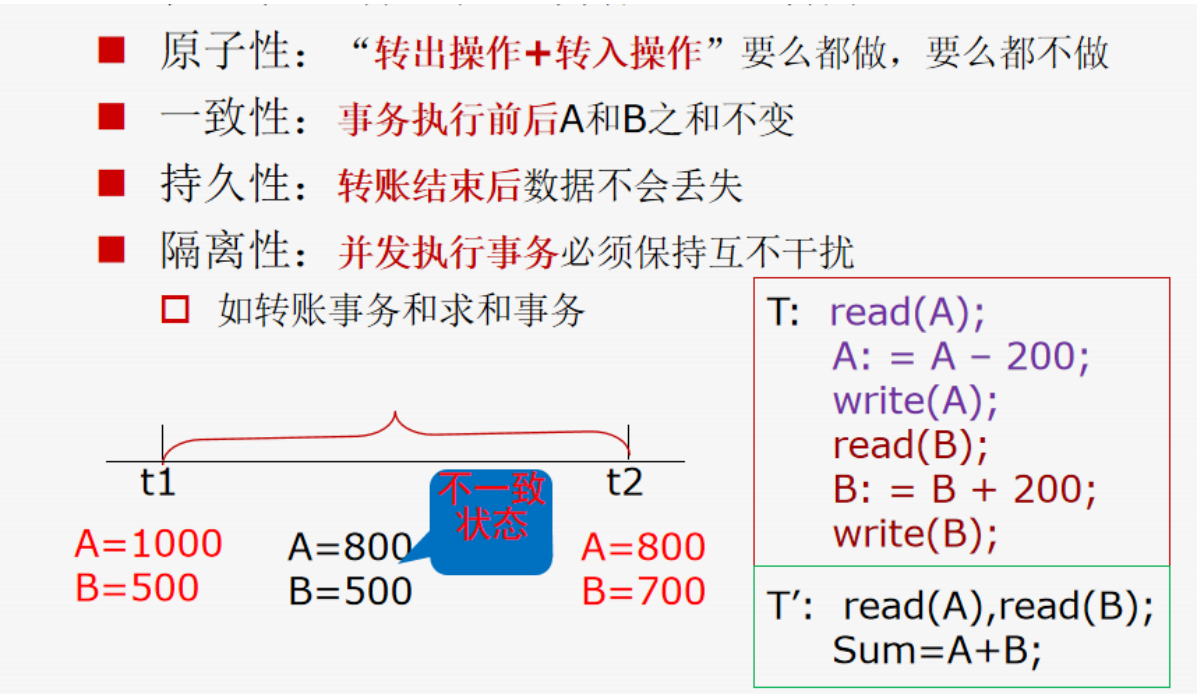
3.3 隔离性

- 当多个事务的操作交叉执行时，若不加控制，一个事务的操作及所使用的数据可能会对其它事务造成影响
- **事务的隔离性是指：一个事务的执行既不能被其它事务所干扰，同时也不能干扰其它事务**
- 具体来讲，一个没有结束的事务在提交之前不允许将其结果暴露给其它事务，因为未提交的事务有可能在以后的执行中废弃
- 若存在其它事务使用了这种无效的数据，则这些事务同样也要进行废弃
- 这种因一个事务的废弃而导致其它事务被牵连地进行废弃的情况称为“级联废弃”



3.4 耐久性

- 事务的耐久性体现在：当一个事务提交后，系统保证该事务的结果不会因以后的故障而丢失
- 也就是说，事务一旦被提交，它对数据库的更改将是永久性的
- 即使发生了故障，系统应具备有效的恢复能力，将已提交事务的操作结果恢复过来，即重做（REDO）处理，使这些事务的执行结果不受任何影响



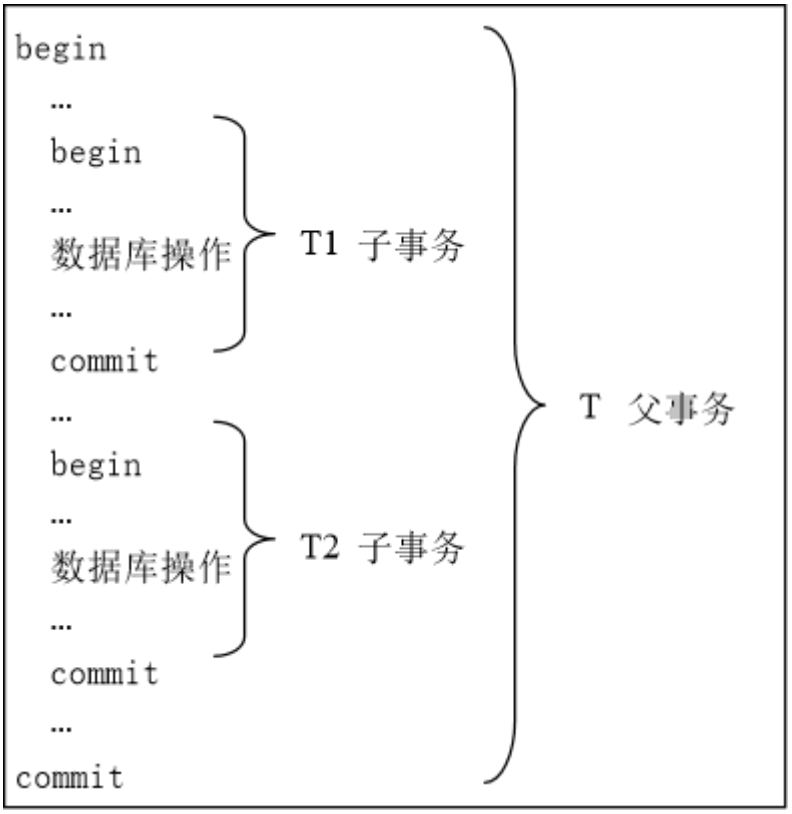
4.事务的种类

4.1 平面事务

- 是指每个事务都与系统中其它事务相分离，并独立于其它事务
- 平面事务是用begin和end括起来的自治执行方式

4.2 嵌套事务

- 是指一个事务的执行包括另一个事务
- 内部事务称为外部事务的**子事务**
- 外部事务称为子事务的**父事务**



提交依赖性：子事务提交，必须等待父事务提交

（准备提交是父事务发起的，他发起后等待子事务回复）

废弃依赖性：父事务废弃，则子事务必须废弃

二、分布式事务

1.分布式事务的定义

- 从宏观上来看，分布式事务是由一系列分布在多个场地上执行的数据库操作所组成的
- 分布式事务：是指分布式数据库应用中的事务，也称为全局事务
- 子事务：一个分布式事务在执行时将被分解为若干个场地上独立执行的操作序列，即一个分布式事务在某个场地上操作的集合
- 显然，分布式事务是典型的**嵌套事务**

2.分布式事务同样具有ACID四个特性

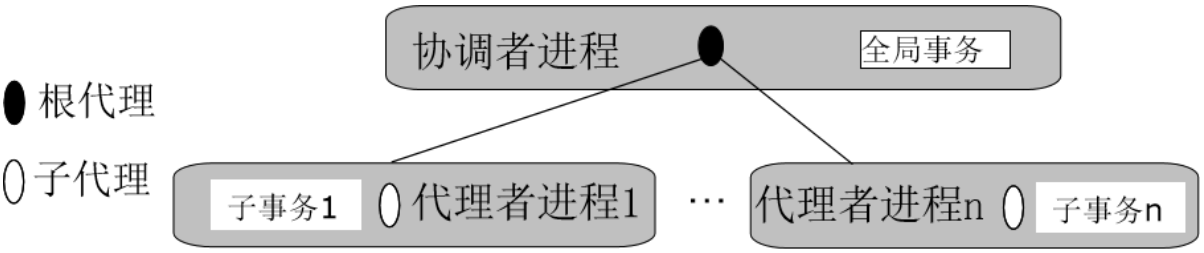
- 由于子事务的分布式执行，因此保证分布式数据库系统的ACID特性更加复杂
- 为了保证全局事务的原子性，必须保证组成该事务的所有子事务要么全部提交，要么全部撤销，不允许出现部分场地上的子事务提交，部分场地上的子事务撤销的情况发生
- 因此，在分布式事务执行过程中，需要对子事务进行协调，从而决定分布式事务的提交与撤销
- 在分布式事务中，除了要考虑对数据的存取操作序列之外，还需要涉及大量的通信原语和控制报文，**通信原语负责在进程间进行数据传送**，**控制报文负责协调各子事务的操作**

3.分布式事务的实现模型

一个分布式事务所要完成的任务是由分布于各个场地上的子事务相互协调合作完成的

3.1 进程模型

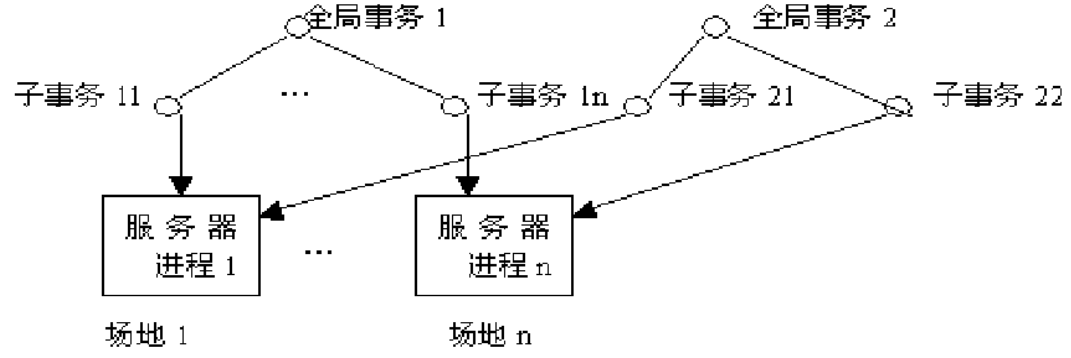
- 全局事务为每一个子事务在相应的场地上创建一个代理者进程（也称局部进程或子进程），由代理者进程执行该场地上的有关操作
- 同时，为协调各子事务的操作，全局事务还要启动一个协调者进程，来进行代理者进程间的通讯，控制和协调各代理者进程的操作
- 发出分布式事务的场地（根代理所在的场地）称为该事务的源场地
- 每个应用均有一个根代理，负责执行全局事务的开始、提交或废弃等命令
- 只有根代理才可以请求创建新的子代理
- 只有当各个子事务均被成功提交后，根代理才能决定在全局上提交该事务，否则根代理将决定废弃该事务



3.2 服务器模型

- 服务器模型要求在事务的**每个执行场地上创建一个服务器进程**，用于执行发生在该场地上的所有子事务
- 每个服务器进程可以交替地为多个事务的子事务服务，即不同全局事务的子事务在同一个场地上共用一个服务器进程

服务器模型可表示为：



4.分布式事务管理的目标

- 使事务的执行具有较高的执行效率，具有较高的可靠性和并发性
- 与事务执行效率有关的因素还包括：
  - CPU和内存利用率
  - 控制报文开销
  - 分布式执行计划
  - 系统可用性

为此，提出分布式事务管理的目标

- 维护分布式事务的ACID性质
- 提高系统的性能，包括CPU、内存等资源的使用效率和数据资源的使用效率，尽量减少控制报文的长度及传送次数，加速事务的响应速度
- 提高系统可靠性和可用性，当系统的一部分或者局部发生故障时，系统仍能正常运转，而不是整个系统瘫痪

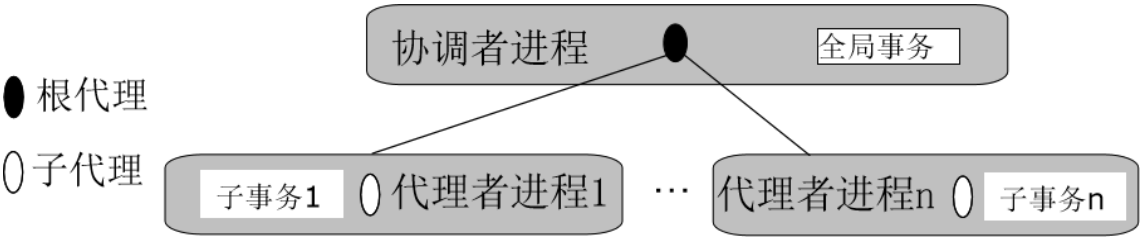
三、分布式事务的提交协议

1.分布式事务

- 分布式数据库中的全局事务由被分解为在各个场地上执行的子事务所组成
- 只有当各个场地上的子事务都正确执行后，全局事务才可以提交
- 只要有一个子事务不能提交，则全局事务应该废弃，接下来所有的子事务也应全部废弃
- 因此，所有子事务均可以正确提交是分布式事务提交的前提
- 上述过程，通过两阶段提交协议来实现，简称`2PC(2-Phase-Commit)`协议

2.协调者与参与者

- 协调者：是协调者进程的执行方，即根代理，负责决定所有子事务的提交或废弃
- 参与者：是代理者进程的执行方，即子代理，负责执行本地数据库操作，负责本地子事务的提交或废弃，并向协调者报告本地子事务的执行状态
- 协调者和参与者都有本地日志，他们在进行操作前都必须将该操作记录到相应的日志文件中（**先写日志再操作**），以进行事务故障恢复和系统故障恢复



3.两阶段提交协议的基本思想

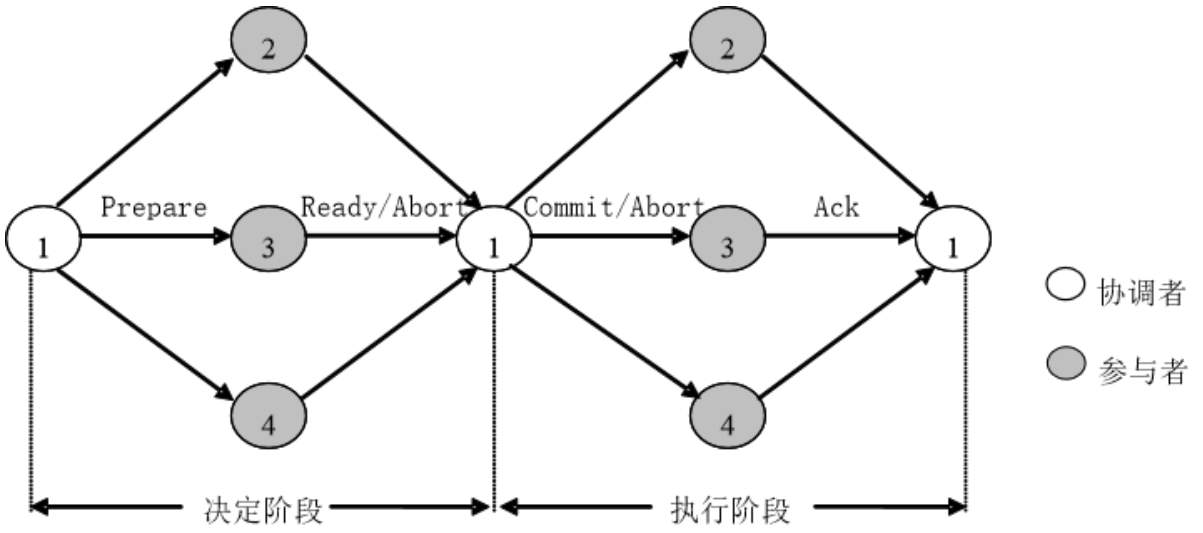
决定阶段：由协调者发送预提交(`prepare`)命令，然后等待参与者的应答

- 如果所有的参与者都返回“准备提交(`ready`)”，那么协调者做出提交决定；如果至少有一个参与者返回“准备废弃”，那么协调者做出废弃决定

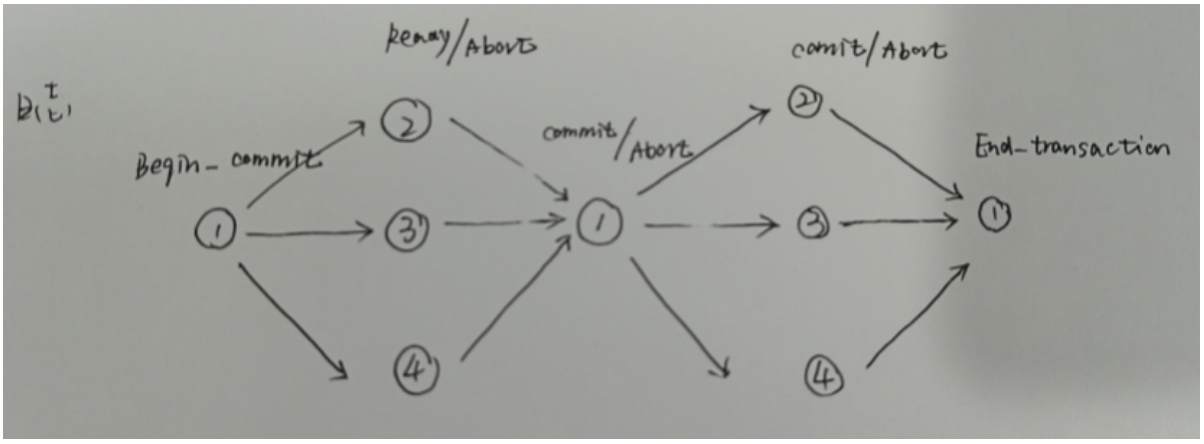
执行阶段：协调者把在决定阶段做出的决定发送给参与者

- 如果协调者向各个参与者发“提交”（`Commit`）命令，各个参与者执行提交；如果协调者向各个参与者发出“废弃”（`Abort`）命令，各个参与者执行废弃，取消对数据库的修改
- 无论是“提交”还是“废弃”，各参与者执行完毕后都要向协调者返回“确认”（`Ack`）应答，通知协调者执行结束

4.两阶段提交的基本流程



先写日志再操作：

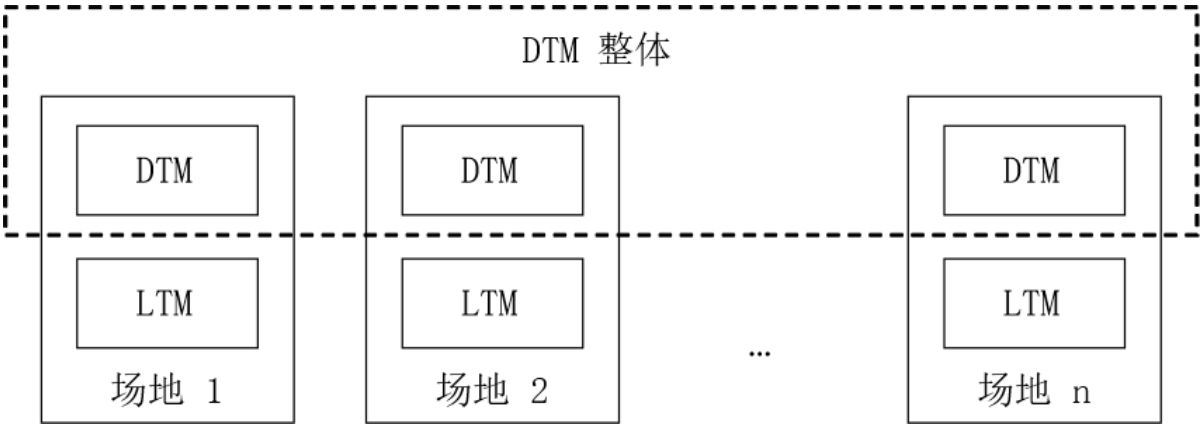


四、分布式事务管理的实现

1.局部事务管理器和分布式事务管理器

分布式事务管理在功能上分为两个层次

- 局部事务管理器（`LTM`）：负责本场地的子事务执行、故障恢复和并发控制；
- 分布式事务管理器（`DTM`）：协同全局事务的执行、物理上多个、逻辑上是一个整体；



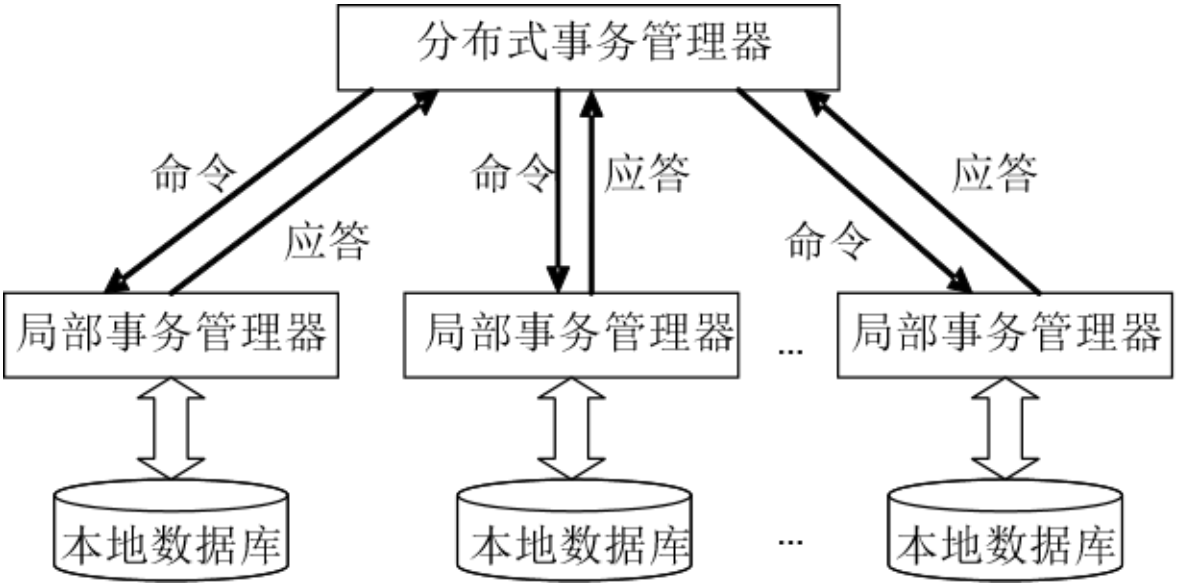
比较：

比较项	局部事务管理器	分布式事务管理器
操作对象	子事务	分布式事务
操作范围	局限在某个场地内	事务所涉及的所有场地
实现目标	保证本地事务的特性	保证全局事务的特性
执行方式	接收命令，发送应答	发送命令，接收应答

2.分布式事务执行的控制模型

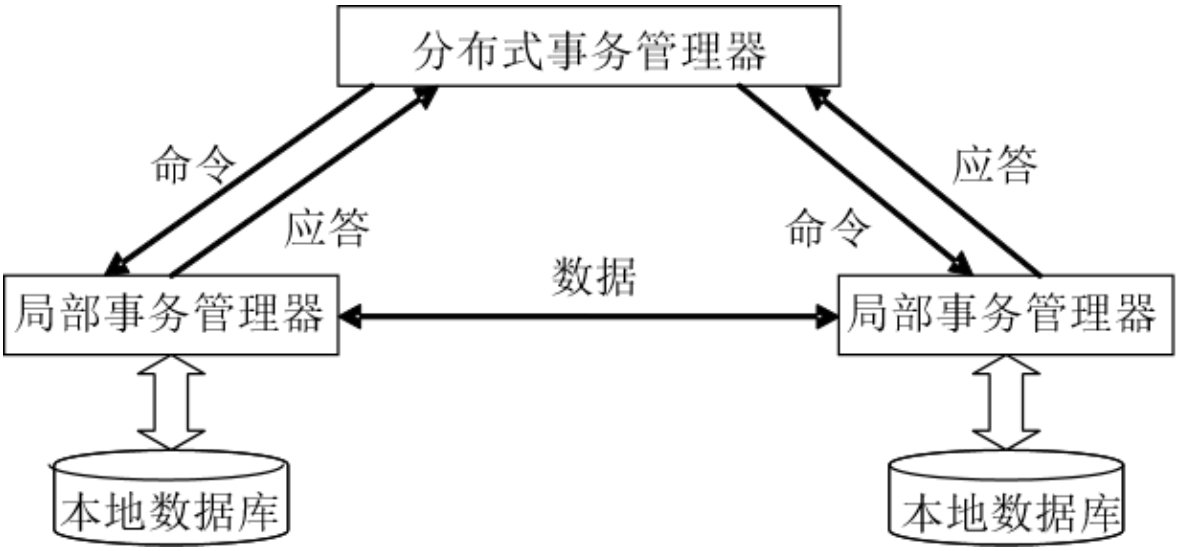
2.1 主从控制模型

- `DTM`作为主控制器，`LTM`作为从属控制器
- 一问一答方式
- `DTM`向`LTM`发送命令并收集应答来实施状态监控
- `LTM`根据`DTM`的命令来执行本场地的子事务，并向`DTM`返回最终结果
- `LTM`之间无直接通信（由`DTM`转发）



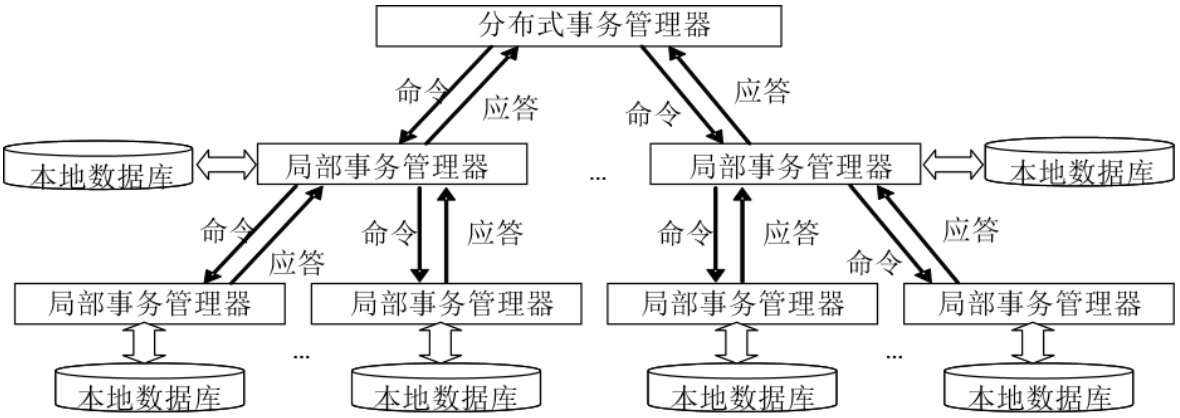
2.2 三角控制模型

- 三角控制模型与主从控制模型相类似，差别是：`LTM`之间可以直接通信，不需要`DTM`作为中介转发
- 因此，同主从控制模型相比，三角控制模型在一定程度上减少了通信代价，但使得`DTM`和`LTM`之间的协作控制变得更加复杂

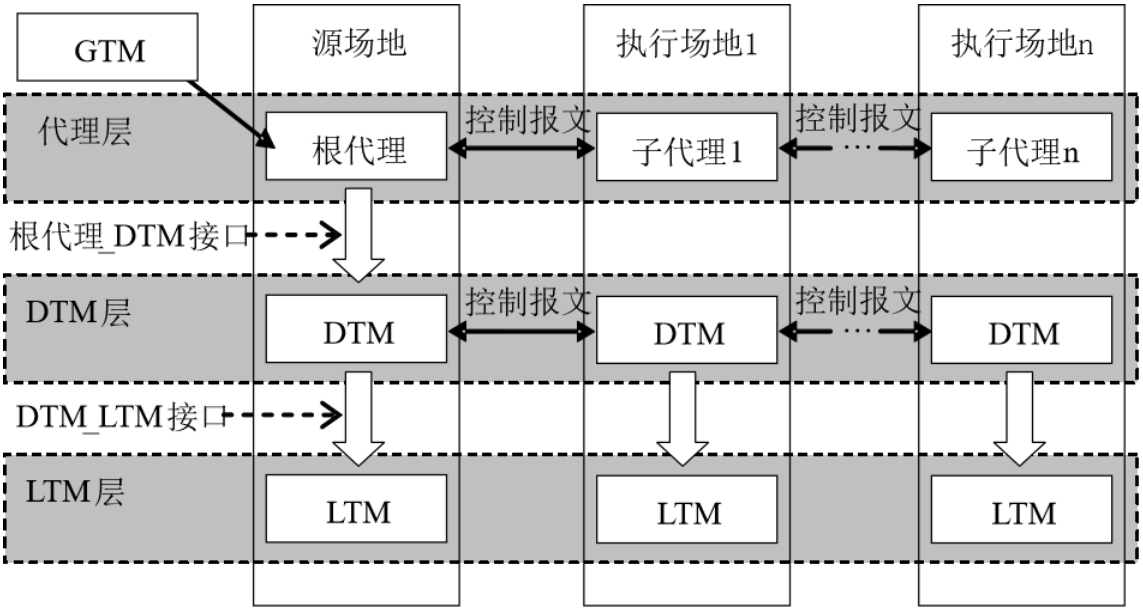


2.3 层次控制模型

- 每个`LTM`可以具有双重角色，除了用来管理其本地的子事务外，还可以将其负责的子事务进一步分解，衍生出一系列更下层的`LTM`，每个分解部分由下一层的`LTM`负责执行
- 这时，`LTM`自身将成为一个新的`DTM`，并控制更下层的`LTM`的执行
- 层次控制模型扩展性好，其层数可以随着任务量的增大而增加，但实现上更加复杂

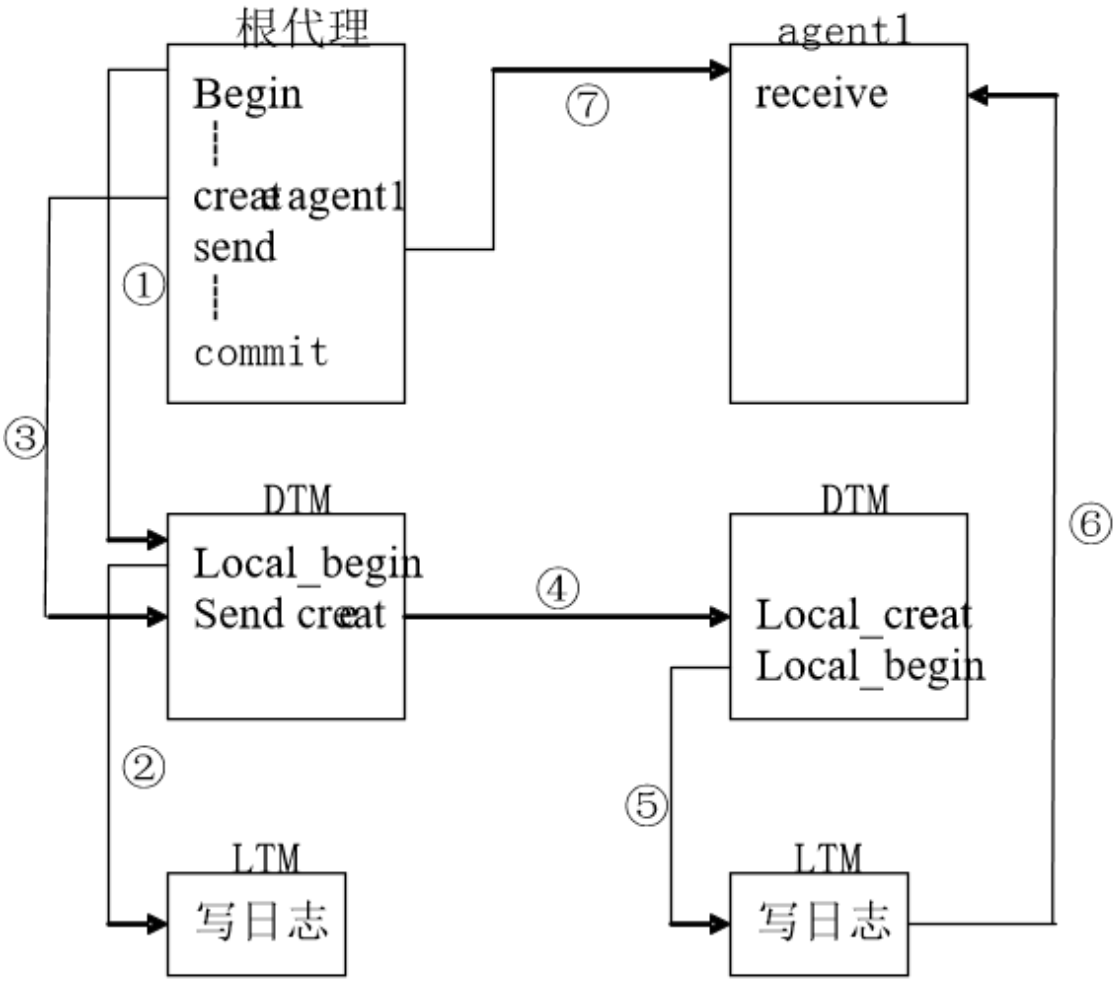


- 实现模型的三层：



  - 代理层：由1个根代理和若干个子代理组成，通过控制报文来协同工作；根代理包含一个`GTM`（全局事务管理器），负责全局事务的划分和调度
  - `DTM`层：由各场地上的`DTM`组成，负责管理分布式事务，调度子事务的执行

    - `LTM`层：由所有场地上的`LTM`组成，负责管理`DTM`交付的子事务的执行
- 三层之间的通信原语
    - 根代理_`DTM`接口
        - begin_transaction：全局事务开始
        - commit：全局事务提交
        - abort：全局事务废弃；
        - create：全局创建子代理。
    - `DTM_LTM`接口
        - local_begin：局部事务开始；
        - local_commit：局部事务提交;
        - local_abort：局部事务废弃；
        - local_create：局部创建子代理。
- 执行过程
    - 根代理向本地`DTM`发送`begin_transaction`命令，全局事务开始；
    - `DTM`向本地`LTM`发送`local_begin`命令，启动局部事务，在局部事务执行前，先将当前局部事务信息写入日志；
    - 根代理通过create原语创建子代理`agent1`；
    - 根代理`DTM`通知子代理`agent1`所在场地`DTM`，子代理`agent1`所在场地`DTM`向本地`LTM`发送`local_begin`命令，建立并启动局部事务；
    - 在局部事务执行前，子代理`agent1`所在场地`LTM`将当前局部事务信息写入日志；
    - 子代理`agent1`所在场地`LTM`通知并激活子代理`agent1`来执行子事务；
    - `agent1`接收参数信息。



五、两段提交协议(2PC)的实现方法

1.基本思想

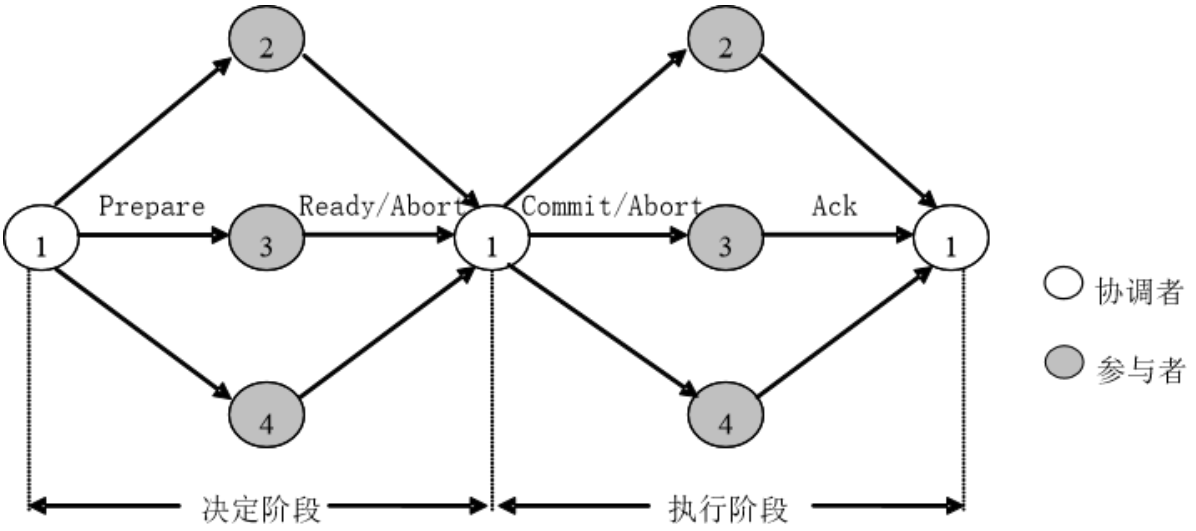
分布式事务提交过程分两个阶段

- 决定阶段
- 执行阶段

2.实现方法

2.1 集中式2PC

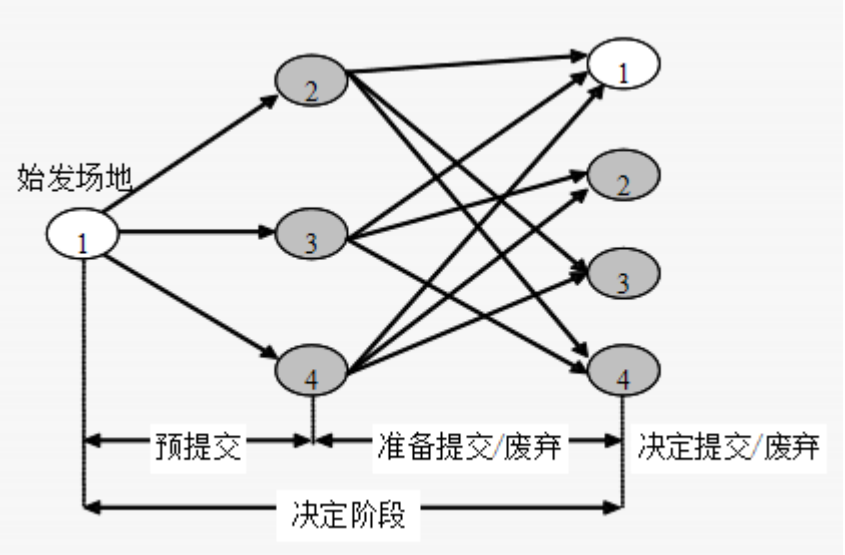
- 首先确定一个协调者场地，通常由事务的发起者场地（源场地）充当，完成事务提交的初始化
- “集中式”主要体现在协调者在整个分布式事务提交过程中始终作为唯一的总控节点（发布命令、搜集状态信息、做出决定）
- 通信只发生在协调者与参与者之间，参与者与参与者之间没有直接通信



- 假设参与者的数目为`N`，则需要传递消息总数为`4N`
- 可见，对于集中式`2PC`，消息传输量随着参与者数目的增加而线性增长
- 假设场地间传输一个消息的平均时间为`T`，则集中式`2PC`传输消息总用时为`4T`（并行）

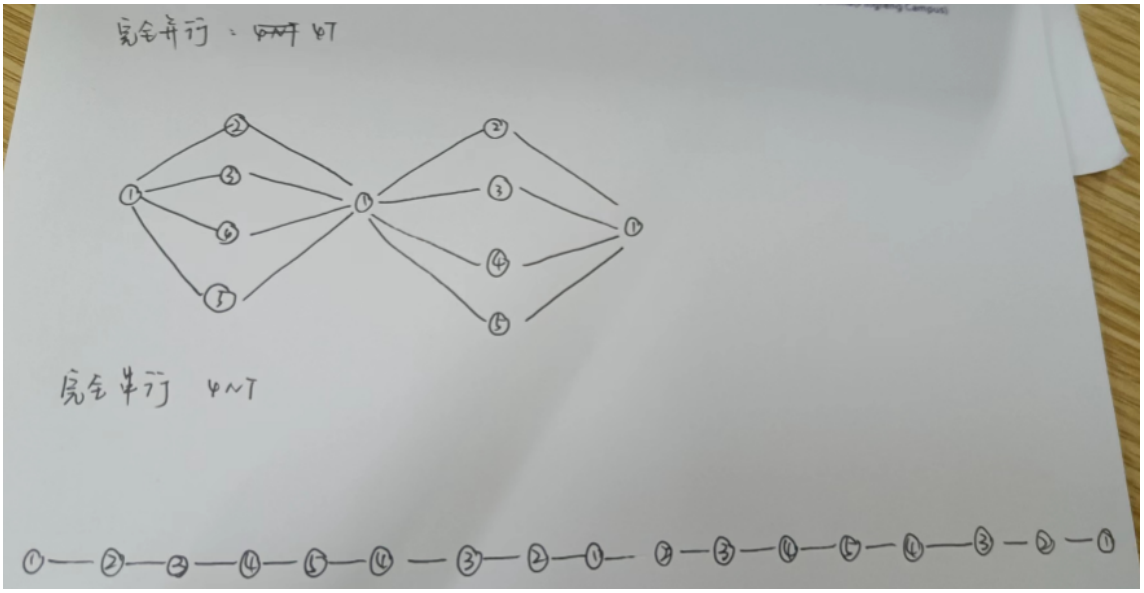
2.2 分布式 2PC

- 分布式是指事务的**所有参与者同时也都是协调者**，都可以决定事务的提交和废弃，提交过程是完全分布地完成。
  - 由事务的始发场地完成提交初始化工作
  - 始发场地初始化完毕后向所有场地广播“预提交”命令。各场地收到“预提交”命令后，做出“准备提交”或“准备废弃”的决定，并向所有场地发送应答信息
  - 每个场地都根据接收到的所有场地的应答独立地做出决定，若存在至少一个 “准备废弃”的应答，则废弃本场地的子事务；否则，各个场地提交子事务
- 分布式`2PC`最大的特点是事务的提交过程只需要一个阶段，即决定阶段
  - 所有场地都可以互相通信，使得各个场地均可以获悉其它场地的当前状态（“准备提交”或“准备废弃”），独立地做出事务是否提交的决定
- 假设有`N`个场地（不含始发场地），则需要传递的消息总数为`N×(N+1)` （N个场地互相通信：N×(N-1)；始发场地与N个场地通信：2×N）。假设场地间传输一个消息的平均时间为`T`，则传输消息总用时为`2T`。由此可见，同集中式`2PC`相比，分布式`2PC`的响应效率高，但通信量大

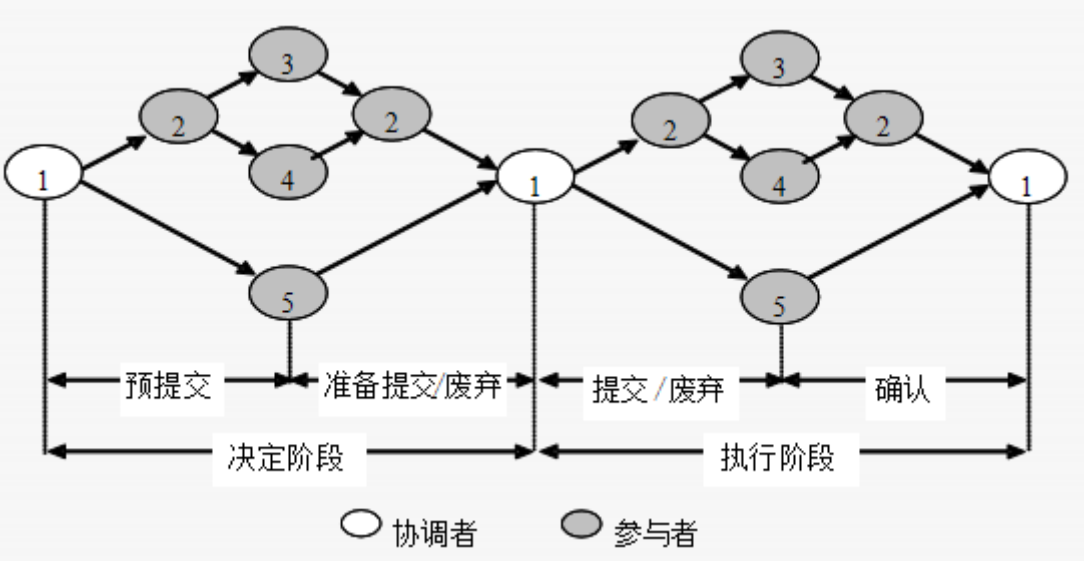


2.3 分层式2PC

- 分层式2PC也称树状实现方法，协调者与参与者间的通讯结构如同一棵树
  - 协调者为根节点，参与者构成树的中间节点或叶子节点
- 根节点向其下层节点下发"预提交"命令；各层节点进行转发，直到叶子节点
- 当叶子节点收到"预提交"命令后，根据自身状态发送应答（"准备提交" / "准备废弃"），应答信息经由中间结点到达根结点；根节点做出决定，下发"提交/废弃"命令；各层节点据此完成提交或废弃，并返回"确认"应答
- 分层式2PC特点是协调者与参与者间的通信没有采用广播方式，而是借助于树型结构将报文逐层传递
- 集中式2PC可以看作是分层式2PC的特例（只有两层）假设参与者的数目为N，则需要传递的消息总数为4N（每一个参与者都有一个入边和一个出边，然后参与者在决定阶段和执行阶段共出现两次）
- 假设场地间传输一个消息的平均时间为T，则传输消息总用时为4T~4NT

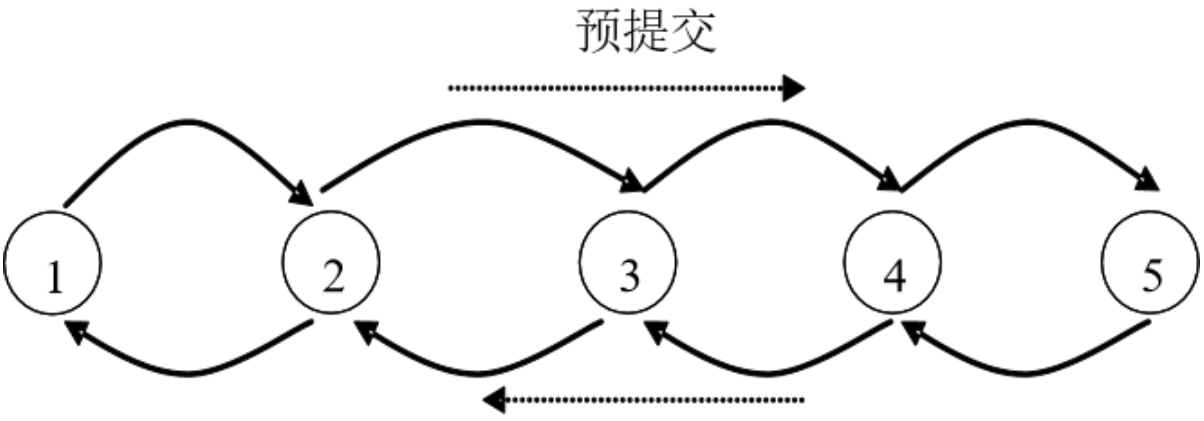


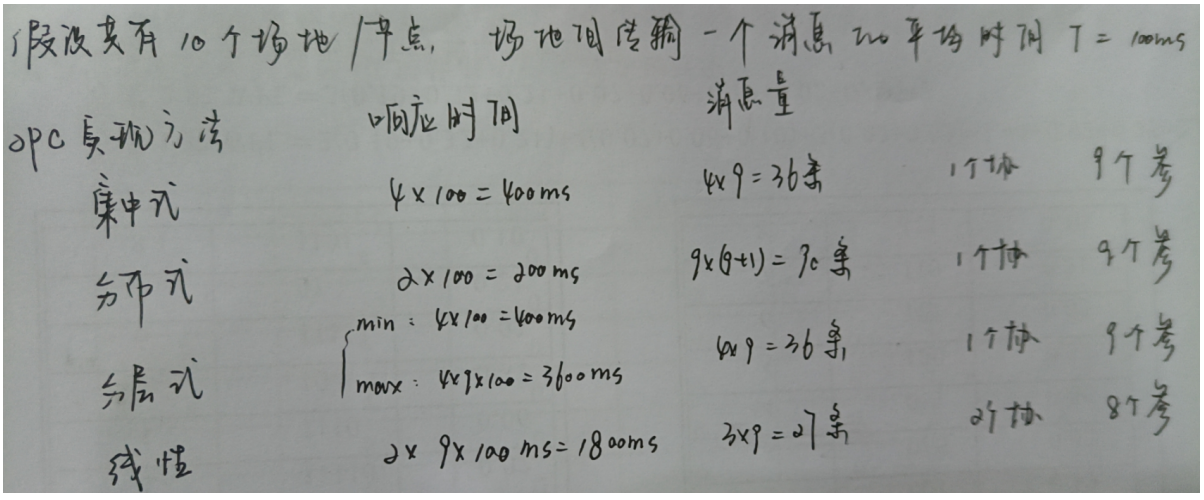
- 由此可见，同分布式方法相比，分层式2PC的报文**传输数量少**，但**响应时间长**



2.4 线形2PC

- 在线性2PC中，允许参与者之间相互通信，由事务的始发场地构造一个线性有序的场地链表
- 表中第一个场地（源场地）和最后一个场地共同担任协调者，在第一个场地后，依次为第一个参与者场地，第二个参与者场地，直到第n个参与者场地
- 事务的源场地进入“预提交”状态，向第一个参与者场地发送“预提交”命令；若第一个参与者场地准备好提交，则向上一个场地发“准备提交”应答，并向下一场地发“预提交”命令；以此类推
- 直到最后一个场地。当最后一个参与者场地收到“预提交”命令，且已准备好提交时，最后一个参与者场地充当协调者，自己首先提交，并向前一场地发提交命令，依此类推直到始发场地，事务提交完成。若任何场地收到“预提交”命令时，没准备好提交，则向前一场地发“废弃”应答，并废弃事务，依次类推直到事务始发场地，事务废弃完成
- 线性2PC的特点是决定阶段和执行阶段都是采用串行方式完成的
- 假设参与者的数目为N，则需要传递的消息总数至多为3×(N-1)， N-1个预提交、N-1个准备提交和N-1个提交/废弃。假设场地间传输一个消息的平均时间为T，传输消息总用时至多为2×(N-1)×T。由此可见，同其它类别的2PC相比，线性2PC的报文传输数量较少，但响应时间最长



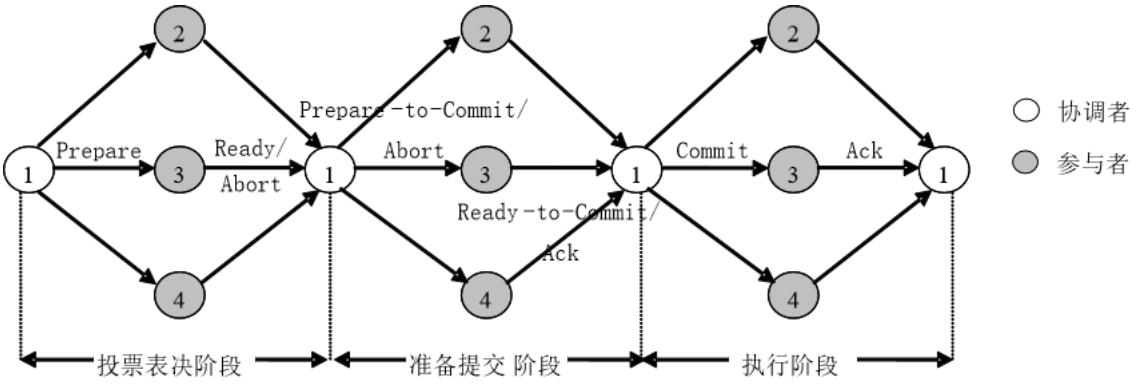


六、非阻塞分布式事务提交协议

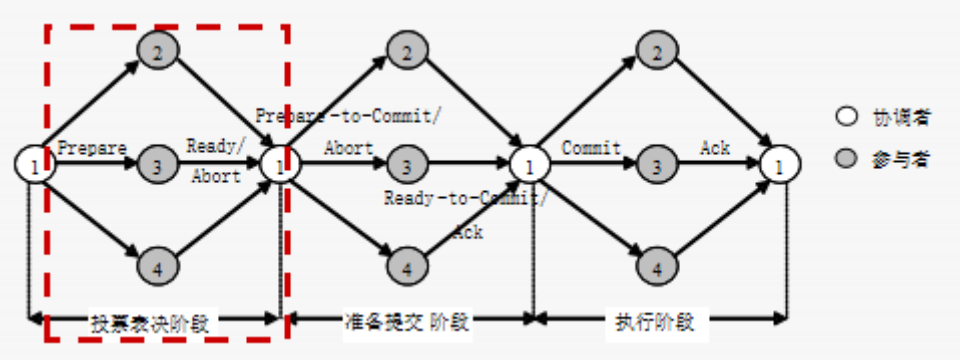
1.两阶段提交协议存在的问题

- 如果在两段提交协议执行的过程中出现协调者故障或网络故障，那么参与者就不能及时收到协调者发出的“提交”命令，参与者处的子事务将处于等待状态

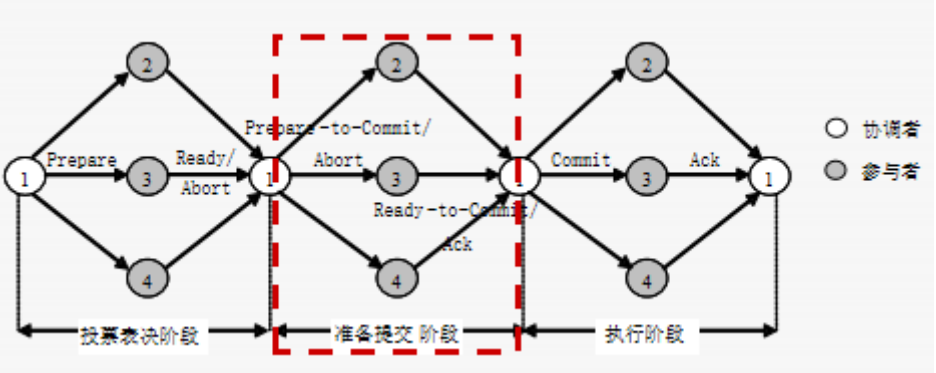
- 在故障恢复前，参与者子事务所占有的系统资源也不能被释放，参与者的子事务进入了阻塞状态。若参与者一直收不到协调者的命令，则子事务将始终处于阻塞状态而挂在相应的执行场地上，所占用的系统资源也不能被其它事务利用
- 三阶段提交协议(`3PC`)



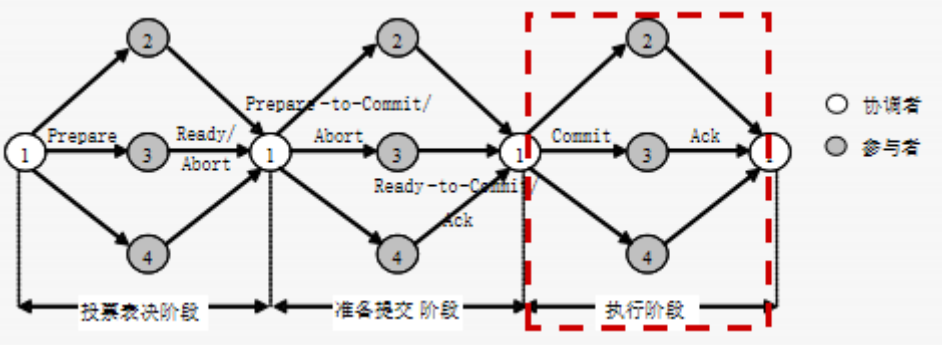
  - 在一定程度上减少了事务阻塞的发生，提高了系统效率
  - `3PC`的基本思想——全局事务的提交分为三个阶段
    - 阶段一：投票表决阶段
    - 阶段二：准备提交阶段
    - 阶段三：执行阶段
- 阶段一：投票表决阶段
  - 由协调者向各个参与者发“预提交”（`Prepare`）命令，然后等待回答
  - 每个参与者根据自己的情况进行投票，若参与者可以提交，则向协调者返回“赞成提交”（`Ready`）应答，否则向协调者发送“准备废弃”（`Abort`）应答



- 阶段二：准备提交阶段
  - 若协调者收到的应答中存在“准备废弃”（`Abort`）应答，则向各个参与者发“全局废弃”（`Abort`）命令，各个参与者执行废弃，执行完毕后向协调者发送“废弃确认”（`Ack`）应答；若协调者收到的应答均为“赞成提交”（`Ready`）应答，则向各个参与者发“准备提交”（`Prepare-to-Commit`）命令，然后等待回答，若参与者已准备就绪，则向协调者返回“准备就绪”（`Ready-to-Commit`）应答



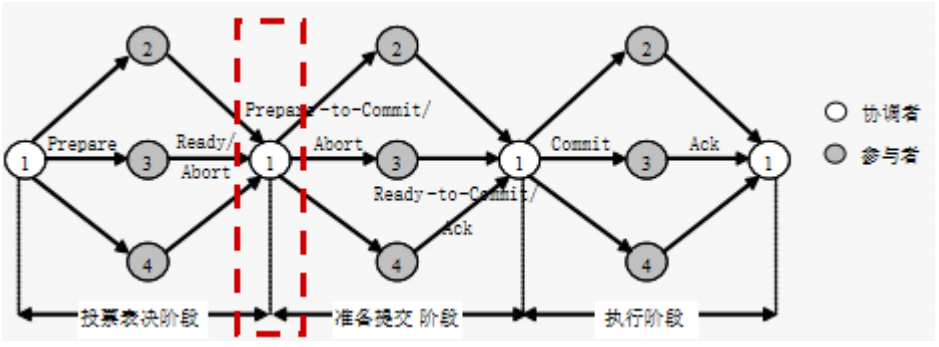
- 阶段三：执行阶段
  - 当协调者收到所有参与者的“准备就绪”（`Ready-to-Commit`）应答后，向所有参与者发送“提交”（`Commit`）命令，此时各个参与者已知道其它参与者均赞成提交，因此可以执行提交，提交后向协调者发送“提交确认”（`Ack`）应答



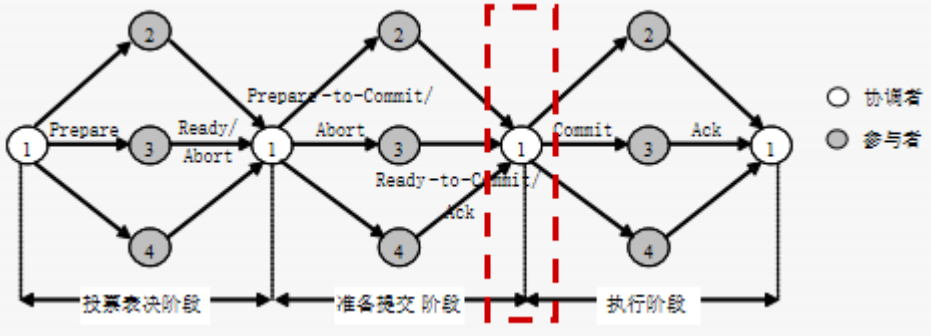
- 参与者的四个状态
  - 参与者发送完“赞成提交”应答后，处于“**赞成提交**”状态
  - 参与者发送完“准备就绪”应答后，处于“**准备就绪**”状态
  - 参与者发送完“提交确认”应答后，处于“**提交**”状态
  - 参与者发送完“废弃确认”应答后，处于“**废弃**”状态

参与者1 \ 参与者2	赞成提交	准备就绪	提交	废弃
赞成提交	相容	相容	不相容	相容
准备就绪	相容	相容	相容	不相容
提交	不相容	相容	相容	不相容
废弃	相容	不相容	不相容	相容

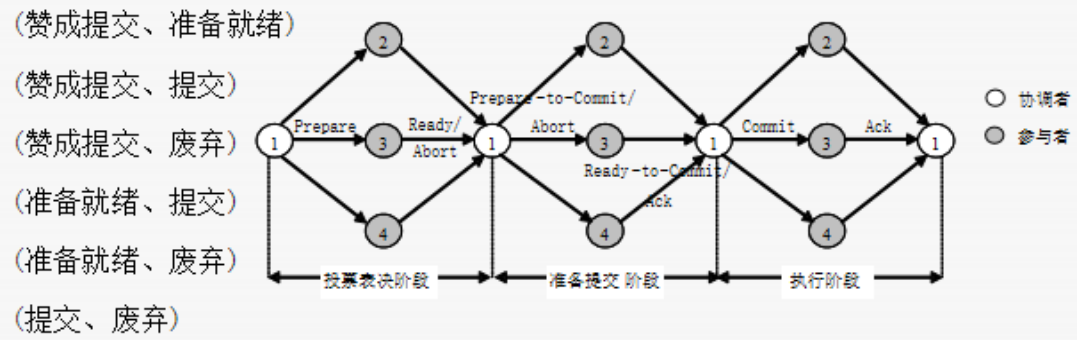
- 两种故障情况下，子事务阻塞问题的解决
  - 参与者发送完“赞成提交”(`Ready`)或者“准备废弃”(`Abort`)的应答后，如果长时间收不到协调者发送来的“全局废弃”(`Abort`)或者“准备提交(`Prepare-to-Commit`)”命令，那么参与者启动恢复处理进程



- 参与者已发送完“准备就绪”(`Ready-to-Commit`)应答后，长时间没有收到协调者发来的“提交”(Commit)命令，参与者启动恢复处理过程

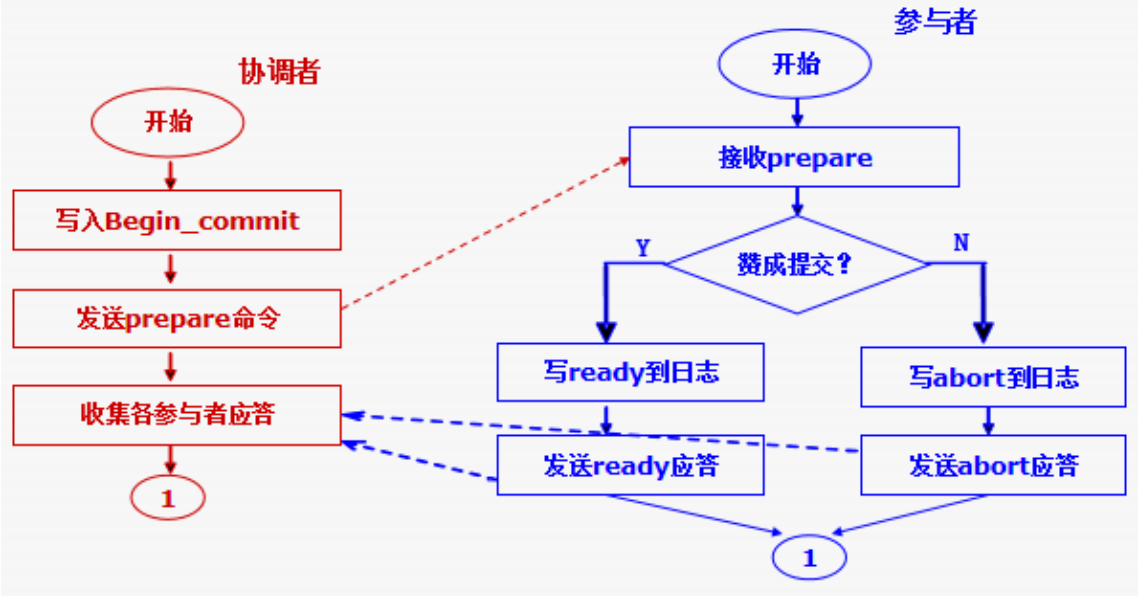


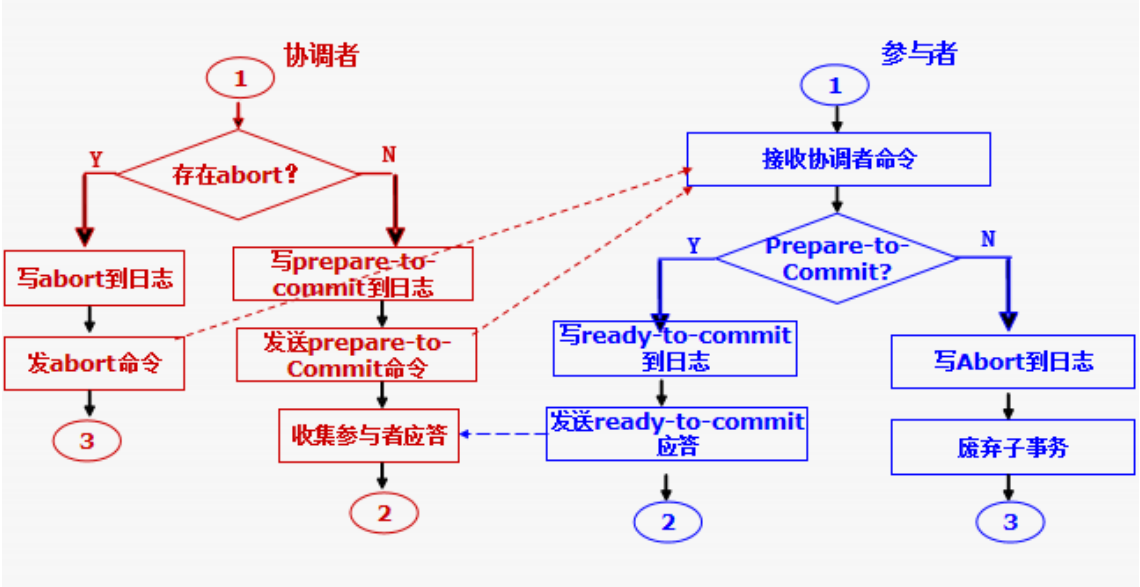
- 恢复处理的具体过程
  - 参与者进入恢复处理后，访问其它参与者的当前状态

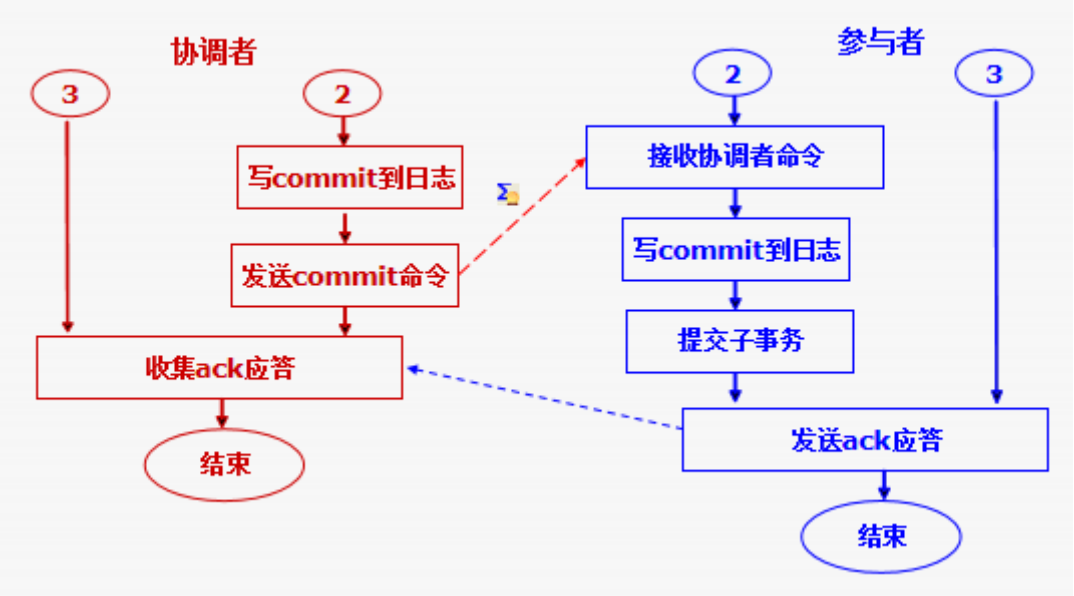


  - (赞成提交、废弃)或者(赞成提交)或者(废弃)
    - 参与者进入恢复处理后，访问其它参与者的当前状态
    - 若某些或者全部参与者处于“赞成提交”或“废弃”状态，则该参与者通知所有参与者进行废弃
  - (赞成提交、准备就绪)或者(准备就绪、提交)或者(准备就绪)
    - 若存在某些参与者处于(赞成提交、准备就绪)或者(准备就绪、提交)状态，或者全部参与者都处于(准备就绪)状态，此时该参与者通知所有参与者提交
    - 注意，若某参与者当前状态为“赞成提交”，则需要先将其转化为“准备就绪”状态，然后再进行提交，进而进入“提交”状态

- 三阶段提交协议的基本流程







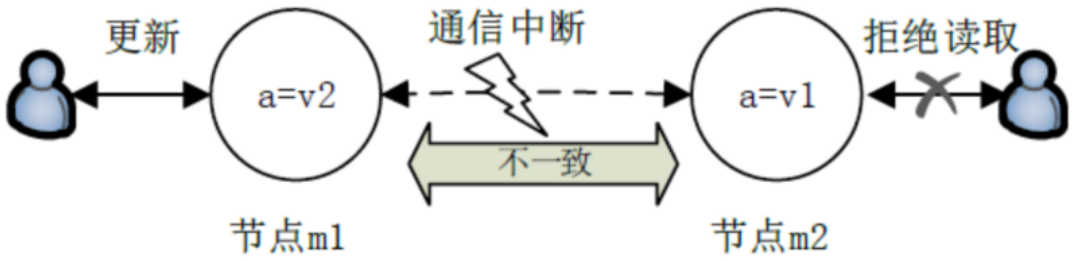
七、大数据库的事务管理

- 大数据库系统是对关系型 SQL 数据系统的补充，是一种分布式、不保证遵循 ACID 特性的数据库设计模式
- 大数据库系统中的事务管理需要解决如下问题
  - **高并发**读写的需求
  - **高可用性**和**高可扩展性**的需求
  - **高效率**读写的需求
- 设计者尝试通过牺牲 ACID 和 SQL 等特性来提升对海量数据的存储管理能力

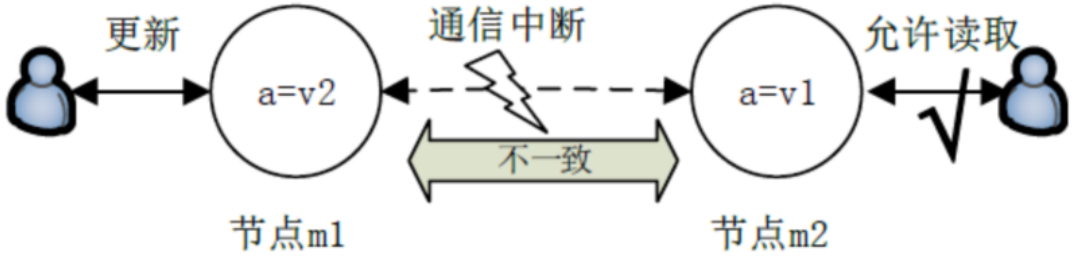
7.1 CAP 理论和 BASE 理论是大数据库系统设计的基石

CAP理论

- 一致性C：即强一致性，要求数据被一致地更新，所有数据变动都是同步的
- 可用性A：系统在面对各种异常时，依然可以响应客户端的读/写请求并提供正常服务



(a) 保证一致性C



(a) 保证可用性A

- 分区容忍性P：分区容忍性是指在网络中断、消息丢失的情况下，系统照样能够工作

`Eric Brewer` 在提出 `CAP` 概念的同时，也证明了 `CAP` 定理：任何分布式系统在可用性、一致性、分区容忍性方面，不可能同时被满足，最多只能得其二

BASE理论

- 基本可用 `Basically Available`：在绝大多数时间内系统处于可用状态，允许偶尔的失败
- 软状态/柔性状态 `Soft-state`：数据状态不要求在任意时刻都完全保持同步，可以有一段时间的不同步
- 最终一致性 `Eventual consistency`：最终一致性是一种弱一致性
- 尽管软状态不要求任意时刻数据保持一致同步，但是最终一致性要求在给定时间窗口内数据会达到一致状态

7.2 弱事务型与强事务型大数据库

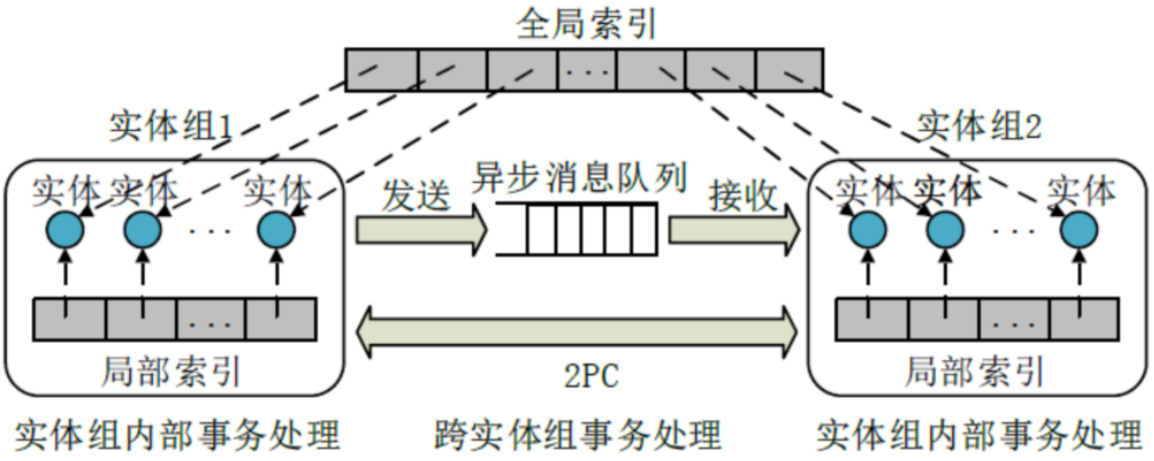
		弱事务型大数据库	强事务型大数据库
特点	优点	复杂性较低； 系统可扩展性好	完整性和实效性较高； 支持跨行跨表事务； 应用程序实现简单
	缺点	不能完全遵循ACID特性； 不支持跨行或跨表事务； 功能简单，应用层负担大	事务执行代价较大； 2PC协议难以实现
应用背景		面向海量交互数据和海量分析数据	面向海量交易数据
设计原则		强调AP，弱化C	强调C，权衡选择CA或CP
应用系统		Bitable、Megastore、Azure等	Spanner、OceanBase等

7.3 大数据库中的事务特性

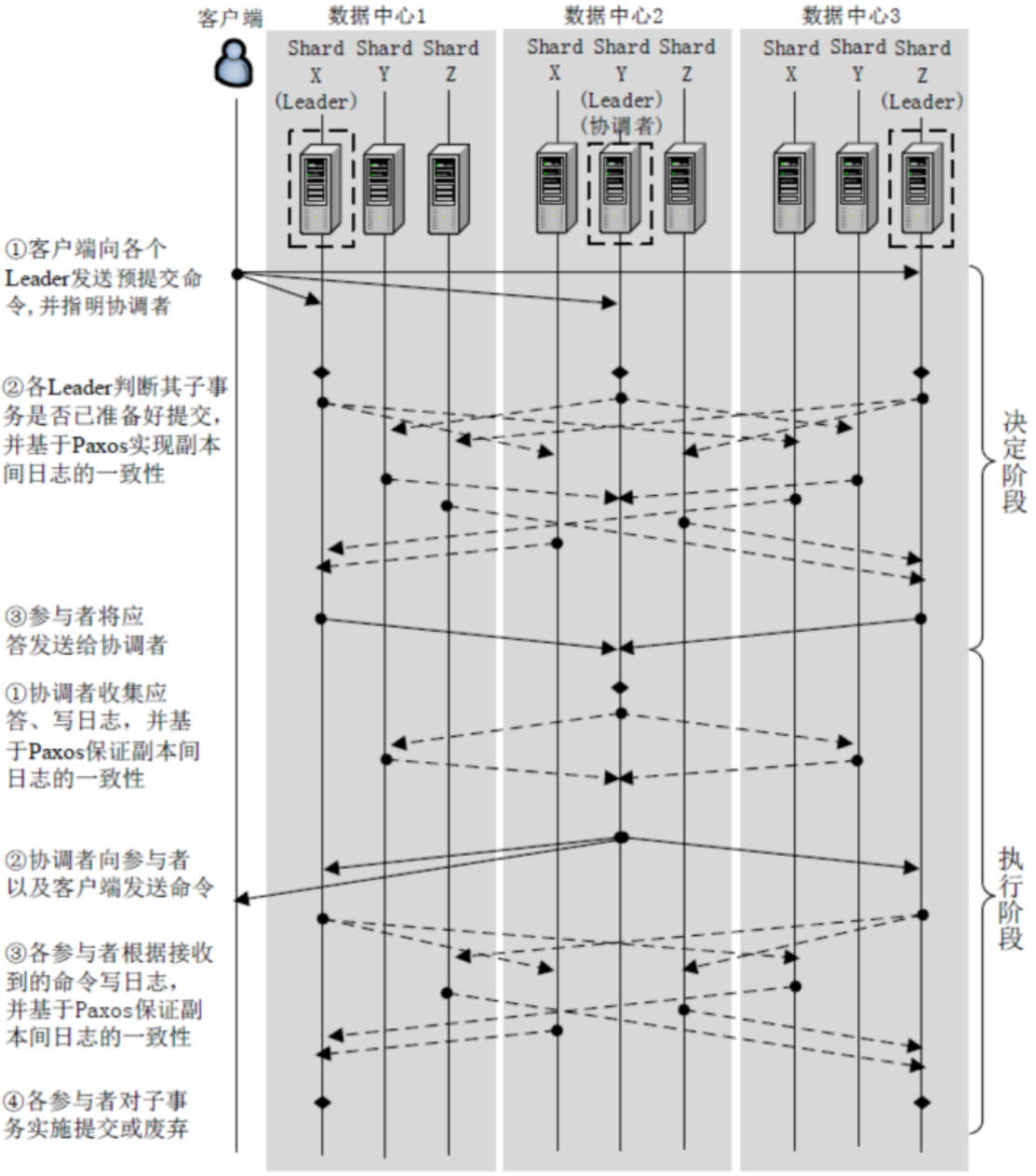
	传统数据库	大数据库	
事务作用域	全局支持	局部支持	组内支持
			分区内支持
			节点内支持
事务的语义	严格支持ACID特性	弱一致性支持	最终一致性
			读自己写一致性
			会话一致性
			单调读一致性
			单调写一致性

7.4 大数据库的事务实现方法

- 基于分组的事务实现方法
- `Megastore`系统的事务实现机制（基于异步消息队列来提供最终一致性）



- `Spanner`系统中`2PC`协议的实现大数据库的事务
  - 假定在`Spanner`数据库系统中某分布式事务涉及数据X、Y、Z，这些数据被存储在不同的`Shard`节点上
  - 它们各有3个副本，形成3个`Paxos`组，分别记作`（X1，X2，X3）`、`(Y1，Y2，Y3)`和`（Z1，Z2，Z3）`，每个组内部通过`Paxos`协议来保证副本的一致性
  - 其中`（X1，Y1，Z1）`隶属于数据中心1，`（X2，Y2，Z2）`隶属于数据中心2，`（X3，Y3，Z3）`隶属于数据中心3
  - 假定`X1`、`Y2`、`Z3`分别为各自`Paxos`组的`Leader`，`Y2`为协调者，`X1`和`Z3`为两个参与者


客户端
数据中心1
数据中心2
数据中心3
Shard X (Leader)
Shard Y
Shard Z
Shard X
Shard Y (Leader) (协调者)
Shard Z
Shard X
Shard Y
Shard Z (Leader)
①客户端向各个Leader发送预提交命令，并指明协调者
②各Leader判断其子事务是否已准备好提交，并基于Paxos实现副本间日志的一致性
③参与者将应答发送给协调者
①协调者收集应答、写日志，并基于Paxos保证副本间日志的一致性
②协调者向参与者以及客户端发送命令
③各参与者根据接收到的命令写日志，并基于Paxos保证副本间日志的一致性
④各参与者对子事务实施提交或废弃
决定阶段
执行阶段