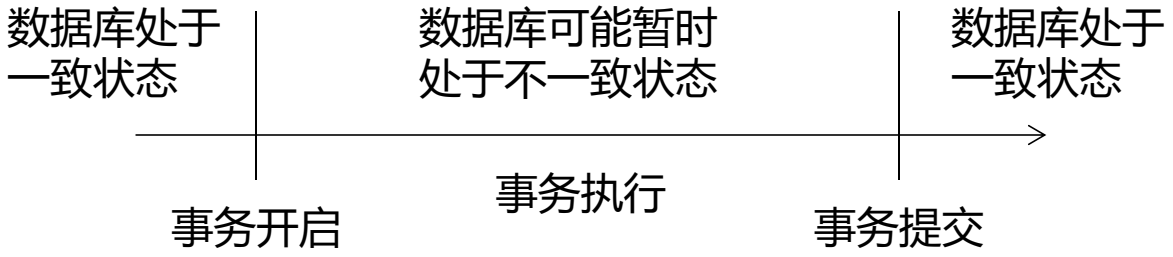
分布式事务原理及应用

郑叔亮

2021年5月

事务（Transaction）的概念

- 数据库事务是指作为单个逻辑工作单元执行的一系列操作（对数据库的相关增删改查的操作），要么完全地执行，要么完全地不执行；事务是数据库系统中保证一致性与执行可靠计算的基本单位
- 无论执行的过程成功与否，事务总应该能够终结：
  - 如果事务成功地完成了它的任务，就称这个事务已**提交**（commit），提交的内涵有两个方面：一是使其所操作的数据项对其他事务可见，二是所有执行结果都会永久保留在数据库中即持久化
  - 如果事务没有完成任务却停止，就称已**中止**（abort），这时所有正在执行的动作都会停止，所有已经执行过的动作都将**反做**（undo），数据库会回退到执行该事务之前的状态，这一过程称为**回滚**（rollback）



事务的形式化定义

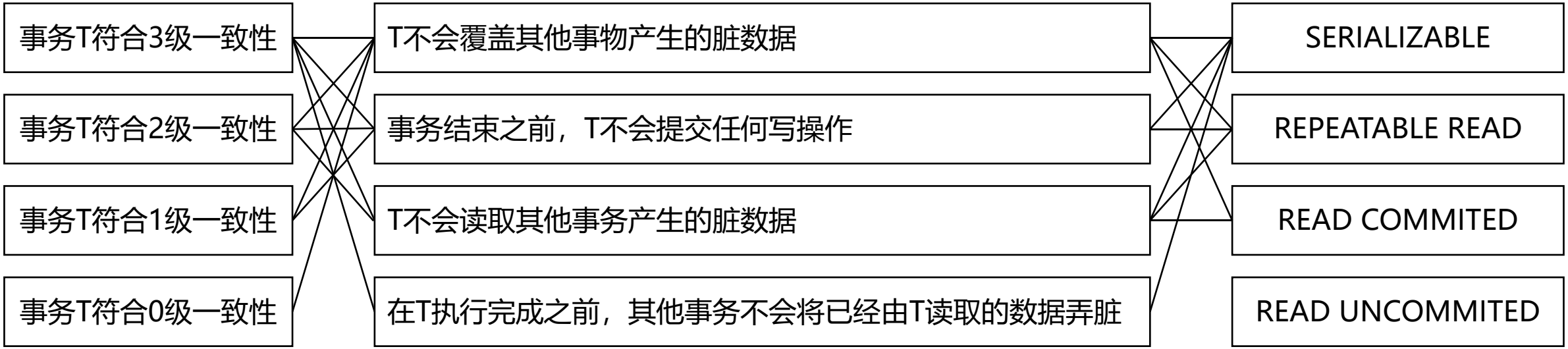
- 将事务T_i作用于数据库实体x的一系列操作O_j记为$O_{ij}(x)$，$O_{ij}(x) \in \{R, W\}$，其中R和W分别表示读和写的原子操作，用$OS_i = \cup_j O_{ij}$表示T_i中的所有操作，用$N_i \in \{abort, commit\}$来表示$T_i$的终结条件
- 进一步地，将T_i定义为偏序集$T_i = \{\sum_i, <_i\}$，其中：
  - $\sum_i = OS_i \cup \{N_i\}$
  - 对于任意两个操作O_{ij}和$O_{ik} \in OS_i$，如果对任意数据项x有$O_{ij} = R(x)|W(x)$，并且$O_{ik} = W(x)$，那么就有$O_{ij} <_i O_{ik}$或者$O_{ik} <_i O_{ij}$，意味着冲突操作的顺序必须在$<_i$中存在，两个操作是冲突的条件是至少一个操作是写操作，且访问同一个数据项
  - $\forall O_{ik} \in OS_i$，满足$O_{ij} <_i N_i$
- 将事务定义为偏序关系的好处是，它可以与**有向无环图（DAG）**关联起来

事务的ACID特性

- 原子性（Atomicity）
  - 事务的所有动作要么全被执行，要么一个都不执行，all-or-nothing
  - 事务恢复和瘫痪恢复
- 一致性（Consistency）
  - 即正确性，能够正确地将数据库从一个一致状态变换到另一个一致状态
  - 完整性实施
- 隔离性（Isolation）
  - 任何事务在任何时候所见到的数据库都是一致的，即在提交之前，一个事务不能向其他并发事务透露其执行结果
- 持久性（Duration）
  - 如果一个事务已经提交，那么它产生的结果就是永久的，这一结果不能从数据库中抹去

事务的一致性等级和隔离级别

- 事务的4个一致性等级和隔离级别



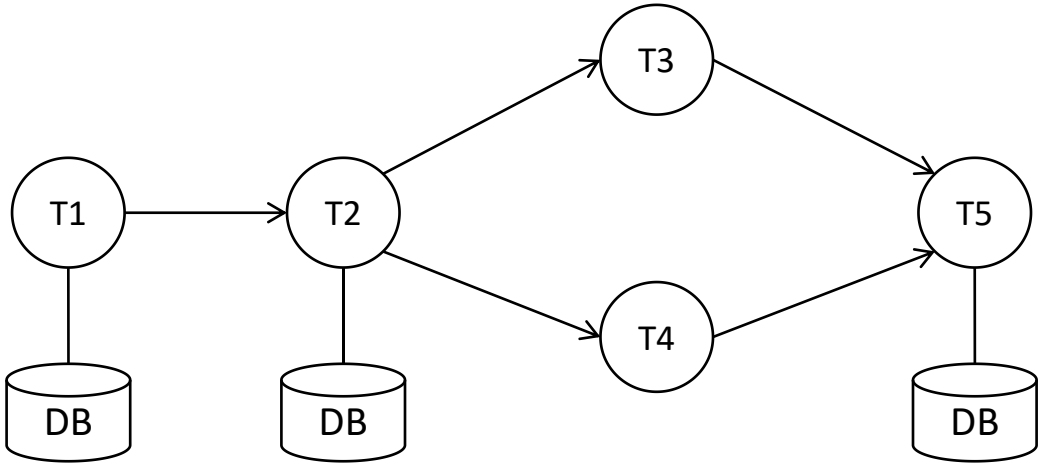
相关概念：遗失更新，游标稳定性，级联式取消，脏读，不可重复读，模糊读，幻读

参考资料：

Gray et al. 1976. Granularity of Locks and Degrees of Consistency in a Shared Data Base.
ANSI 1992. Database Language SQL, ansi x3. 135-1992 edition.
Berenson et al. 1995. A Critique of ANSI SQL Isolation Levels.

事务的结构类型

- 平面事务：有一个起点和一个终点
- 嵌套事务：包含其他具有单独起点和终点事务的事务
  - 封闭式嵌套事务：自底向上的方式提交，子事务的提交是双亲事务提交的先决条件
  - 开放式嵌套事务：允许它的未执行完全的部分结果可以在事务之外被观察到（Saga、分离事务）
- 工作流：一个具有开放式嵌套语义的活动，这个活动由一系列明确了优先关系的任务组成，这里的任务可以是其他活动，也可以是封闭式嵌套事务

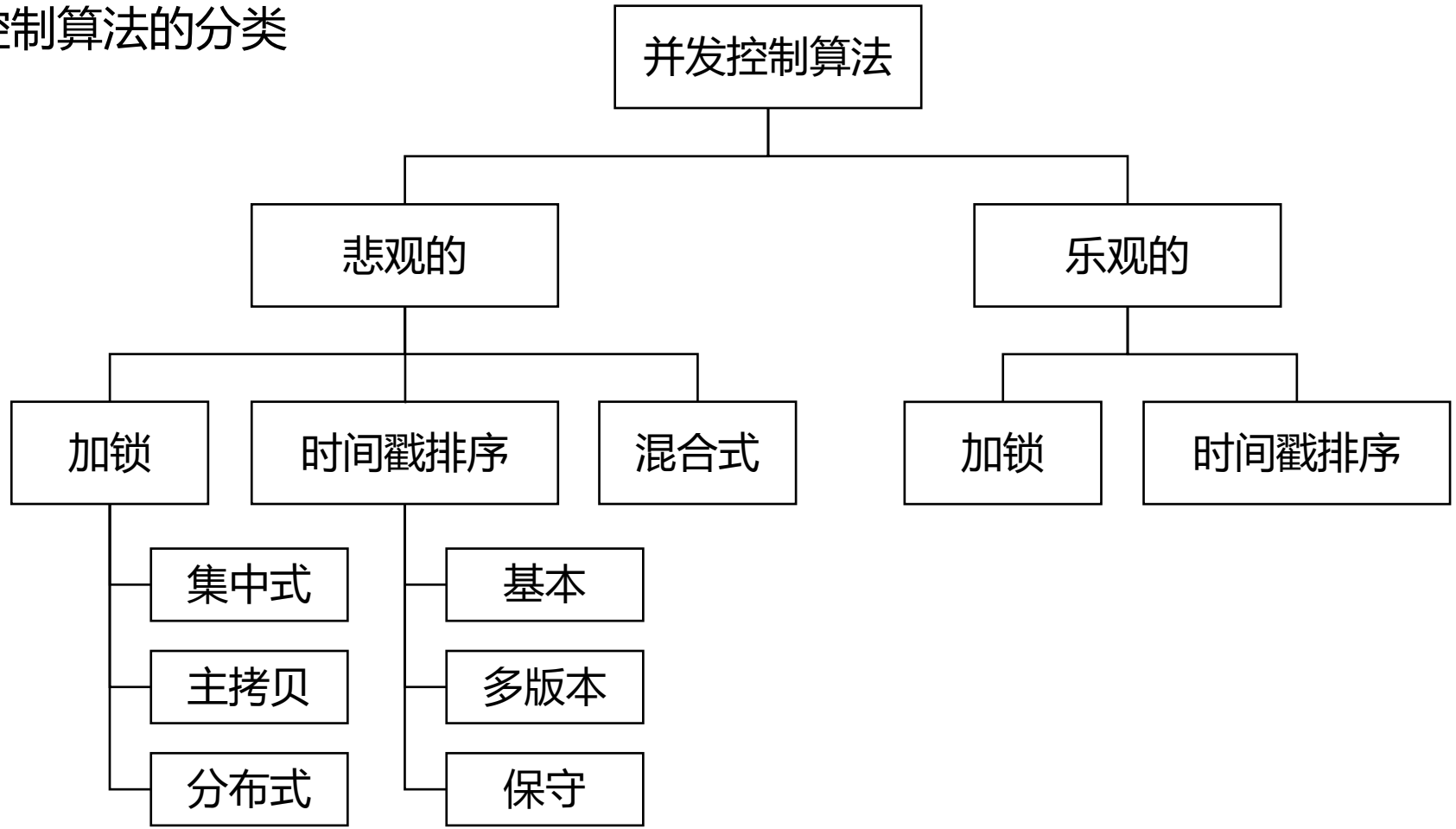


分布式并发控制

- **历史**（也称作**调度**）：定义在一组事务之上，并且指定了这些事务的操作的交错执行顺序
- **完整历史**：为所有操作定义了执行的顺序
- **串行**：在一个**完整历史**中，如果多个事务的操作没有交错，即同一个事务的操作都是连续出现的，就称这个**历史**是串行的
- **冲突等价性**：如果两个**历史**定义在同样的事务集合上，且对于任何两个**相冲突**的操作，在两个**历史**中满足完全相同的偏序关系，那么就说这两个**历史**是**冲突等价**的
- 一个**历史**是**可串行化**的，当且仅当它与一个**串行**的历史**冲突等价**
- 对于非重复的分布式数据库，在每一个站点上的事务执行历史称为**局部历史**，如果数据库没有复制，并且每个**局部历史**都是**可串行化**的，那么它们的并集（称为**全局历史**）也是**可串行化**的
- **并发控制程序的主要功能就是为需要执行的事务产生可串行化的历史**

分布式并发控制

- 并发控制算法的分类

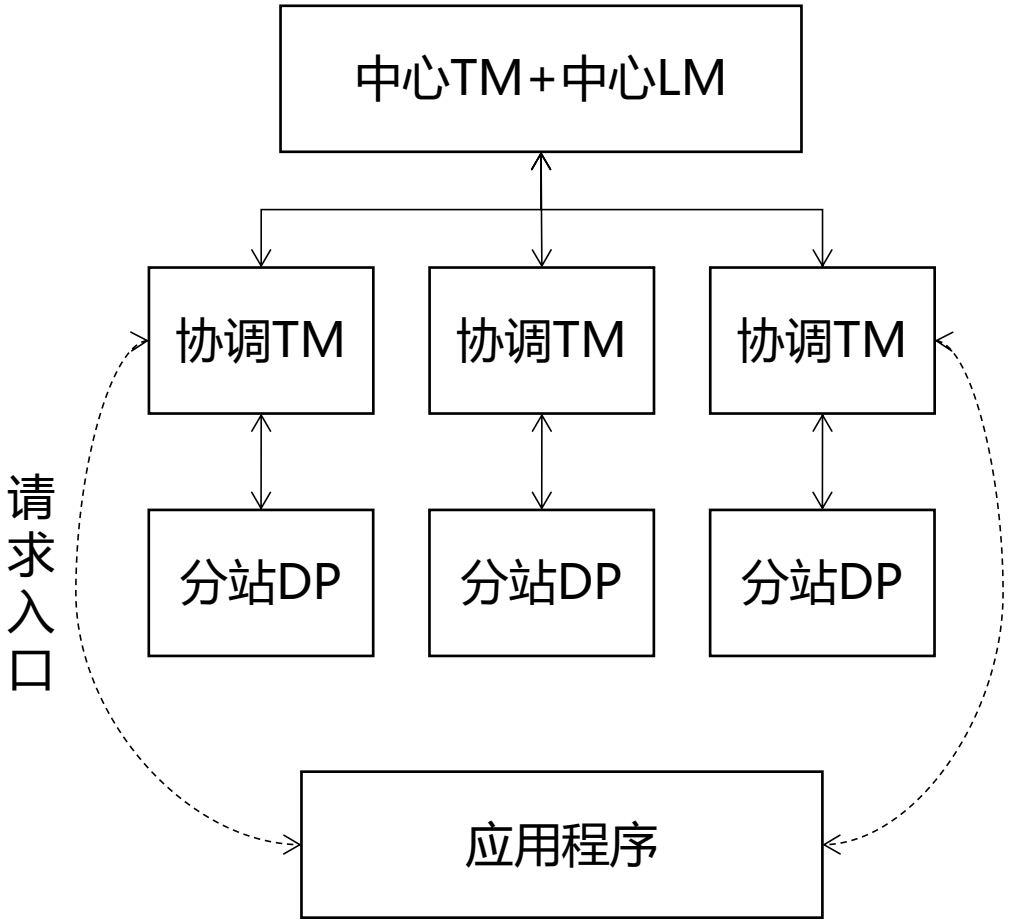


分布式并发控制

- 基于加锁的并发控制算法
  - 作用于每个锁单位的锁有两种类型或模式：**读锁**，**写锁**
  - 如果两个访问相同数据项的事务可以同时申请到相应的锁，就称这两个锁模式是相容的（两个读锁）
  - 两阶段加锁（2PL）：不允许事务在释放了它的任何一个锁之后请求新的锁；或者可以说，直到能够确定一个事务不会再请求新的锁，它已有的锁才可以释放；每一个事务都有两个阶段：增长阶段和收缩阶段
  - 已经证明，任何一个遵守2PL规则的并发控制算法所生成的**历史**都是**可串行化**的
  - 第三种锁模式——**有序共享**锁：两个事务Ti和Tj在同一个数据上的有序共享指的是，给定一个历史允许两个操作o∈Ti和p∈Tj之间的有序共享锁，如果Ti申请o锁的时间位于Tj申请p锁之前，那么o的执行也会在p之前；有序共享锁即**读写锁**
  - **集中式2PL算法**和**分布式2PL算法**

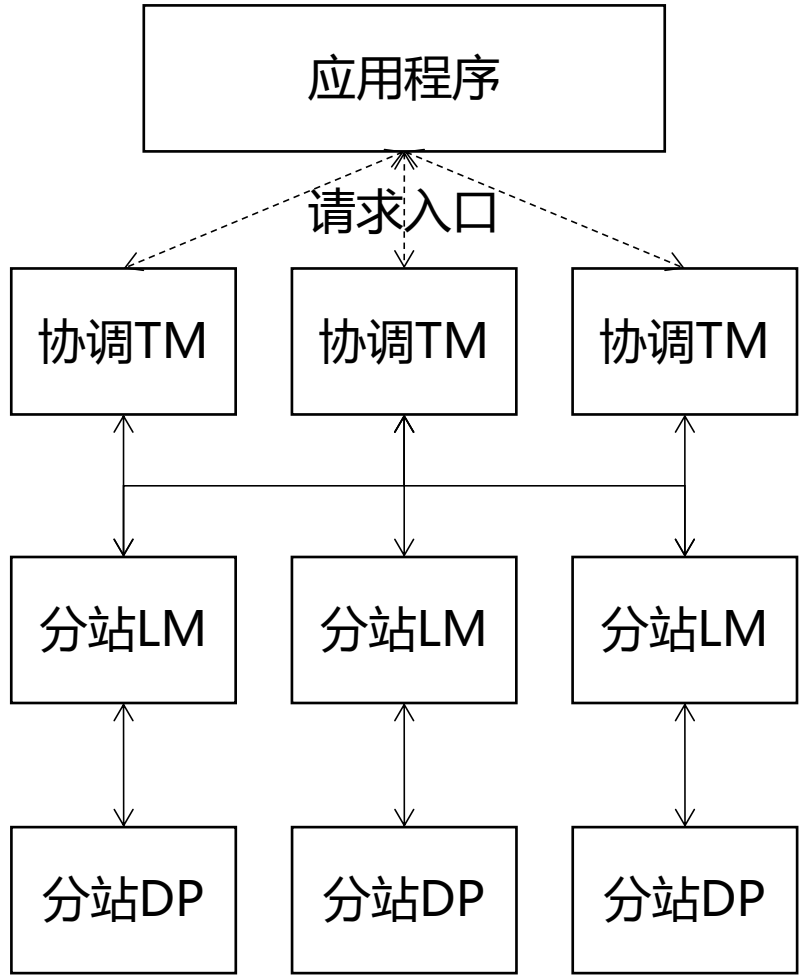
分布式并发控制

- 集中式2PL算法（C2PL）
  - 将锁管理的任务交给一个单独的站点完成
  - 其余站点的事务管理程序（协调TM）与中心事务管理程序通信
  - 算法框架分为三个部分：C2PL-TM，C2PL-LM，DP
  - 消息通信：同步/异步，协议，可靠性
  - 消息结构：{op∈{BT, R, W, A, C}, data, value, tid, result}
  - 锁调度机制：队列
  - 元数据和数据分布管理：统一视图



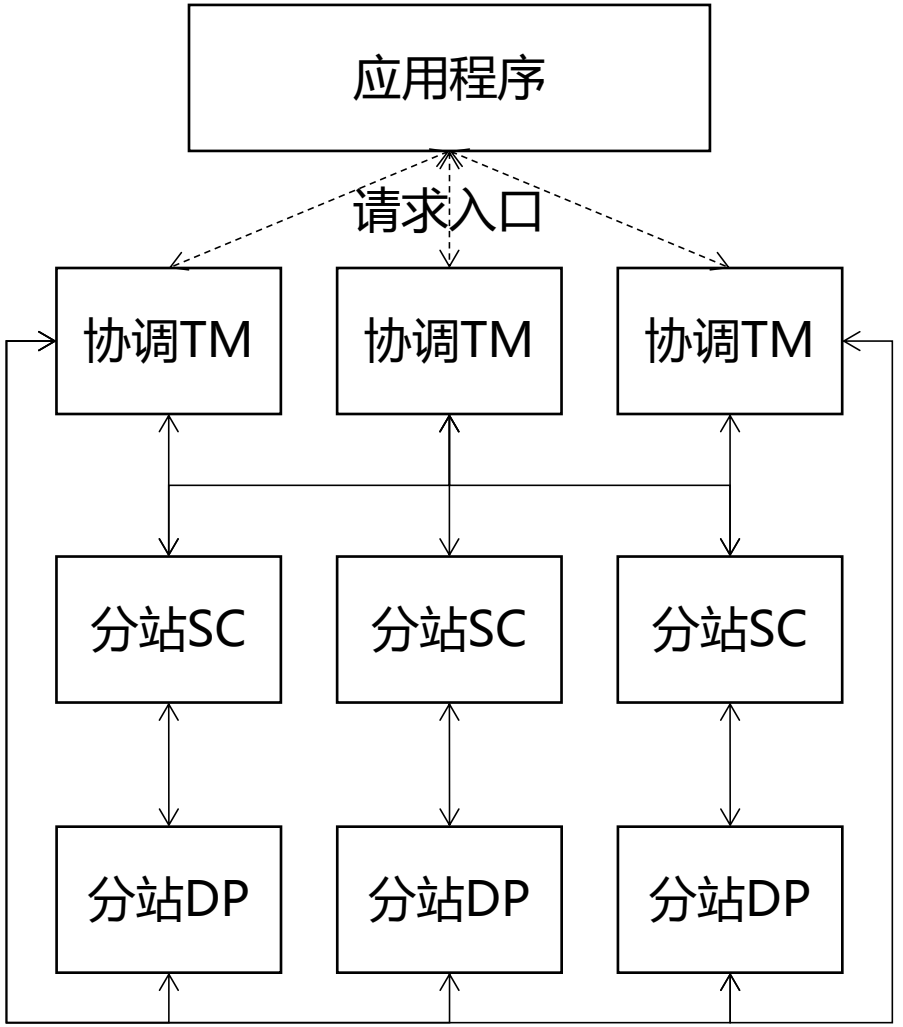
分布式并发控制

- 集中式2PL算法的问题：中心站点会成为瓶颈
- 分布式2PL算法
  - 消息通信：同步/异步，协议，可靠性
  - 锁调度机制：队列
  - 站点的逻辑拓扑结构?
  - 是否需要统一的元数据视图?
  - 消息结构?



分布式并发控制

- 基于时间戳的并发控制算法
  - 时间戳特性：唯一性和单调性
  - 分布式系统时间戳：局部计数值+站点标识号
  - 时间戳排序（TO）规则：给定事务T_i和T_k中的两个冲突操作O_{ij}和O_{kl}，当且仅当$ts(T_i)<ts(T_k)$时，O_{ij}会在O_{kl}之前执行，在这种情况下，T_i成为**较旧的事务**，T_j为**较新的事务**
  - 使用TO规则的调度程序会将每一个操作与已经调度过的冲突的操作进行比较，如果新的操作属于一个较新的事务，那么就接受它；否则的话就拒绝它，然后将相应的事务赋予新的时间戳并重新启动
  - 基本TO算法：TO规则的直接实现，协调TM为每个事务分配时间戳，为每个数据项选定存储站点，并且将相关的数据操作发送到这些站点上



分布式并发控制

- 基本TO算法不会让操作等待，而是直接重启，这样可以有效避免死锁，但是会有性能损耗
- 避免由重启带来开销的算法
  - 各个站点进行时间戳同步，避免部分站点时间戳明显较小，事务被频繁重启
  - 保守TO算法：在SC中维护若干队列作为事务的缓冲区，对列中的事务按照时间戳排序之后再顺序地执行，在边界情况下网络乱序会造成事务重启
  - 多版本TO算法：更新并不改变数据库，每个写操作都建立一个数据项的新版本，每个版本都被标记上一个建立它的事务的时间戳；用户只针对数据项发出事务请求，而不会针对某个特定版本，TM为每个事务分配一个时间戳，这个时间戳被应用于每个版本
    - $R_i(x)$被转化成针对x的某一个版本的读：时间戳小于ts(Ti)中的最大的版本的x_v，$R_i(x)$即转化为$R_i(x_v)$
    - $W_i(x)$会被转化为$W_i(x_w)$，满足$ts(x_w)=ts(T_i)$；它会被发送给数据处理程序，当且仅当没有其他的时间戳大于$ts(T_i)$的事务读取了x的某个版本x_r（$ts(x_r)>ts(x_w)$）;反之，$W_i(x)$就会被拒绝。

分布式并发控制

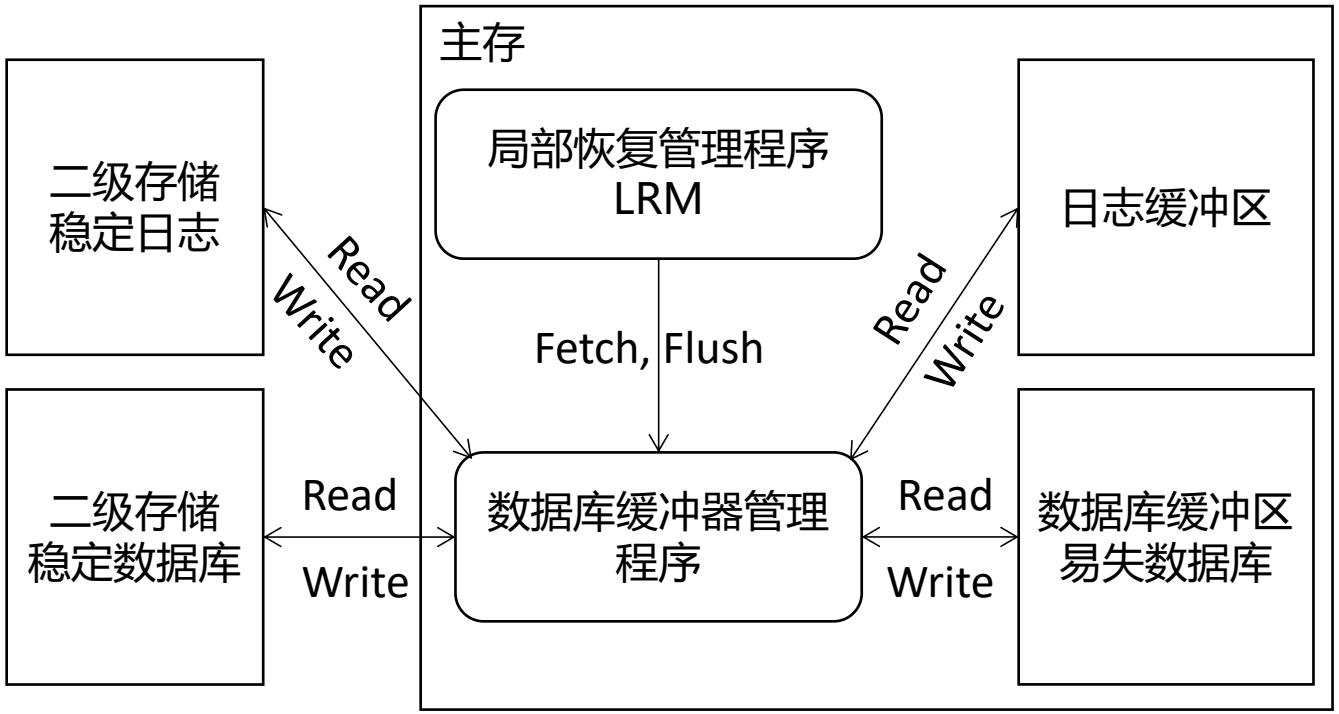
- 乐观并发控制算法
  - 前面的并发控制算法本质上都是悲观的，它们假设事务之间的冲突是比较频繁的，不允许两个事务同时对同一个数据项进行冲突的访问，因此任意一个操作都要按照如下步骤来执行：有效性验证（V），读（R），计算（C），写（W）
  - 乐观算法会将有效性验证这个步骤推迟到写之前，最初每个事务会对数据项的局部拷贝进行更新，然后有效性验证步骤会检查这些更新是否可以保证数据库的一致性，如果是，这个更新就会作用到全局；否则，事务就必须取消并且重启
  - 在有效性验证开始的时候，时间戳会被分配给事务，并且拷贝到事务的各个子事务上去
  - 乐观并发控制算法的一个优点是它可能会允许更高级别的并发性，当事务的冲突非常稀少的时候，乐观算法会比加锁算法更加高效
  - 乐观算法的一个最大问题是，它需要更大的存储代价：为了验证一个事务，乐观算法需要存储其他已经终结的事务的读集和写集

分布式DBMS的可靠性

- 分布式DBMS的可靠性意味着事务的原子性（A）和持久性（D）
- 基本概念：
  - 可靠性、可用性
  - 平均无故障时间、平均修复时间
  - 事务故障、站点（系统）故障、介质故障、通信故障
- 局部可靠性协议
- 分布式可靠性协议

分布式DBMS的可靠性

- 局部可靠性协议
  - 重做Redo：故障发生的时候，事务更新过的易失数据并未写回稳定数据库中，在恢复的时候就需要重做事务
  - 反做Undo：在事故发生的时候，事务仍然在运行，原子性要求稳定数据库中不能包含事务的任何效果，在恢复的时候就需要反做事务的操作
  - 写在先日志原则WAL：
    - 为了支持Redo，当事务提交时，修改后的数据应该在稳定数据库更新之前写入到稳定日志中；
    - 为了支持Undo，在稳定数据库更新之前，修改前的数据应该写入稳定日志中



基本的LRM结合缓冲管理器的二级存储结构

分布式DBMS的可靠性

- LRM命令的执行决策
  - 固定：缓冲管理程序在事务执行期间自行将更新过的缓冲区页面写入稳定存储
  - 非固定：缓冲管理程序等待LRM的指示才写入稳定存储
  - 强行写：在事务结束时（即在提交点）强制缓冲区管理程序将缓冲区页面强行写到稳定存储
  - 非强行写：根据缓冲区管理算法按照需求外写
  - 四种组合：固定/强行写，固定/非强行写，非固定/强行写，非固定/非强行写
- 建立检查点
  - 避免扫描整个日志
- 处理介质故障
  - 维护数据副本

分布式DBMS的可靠性

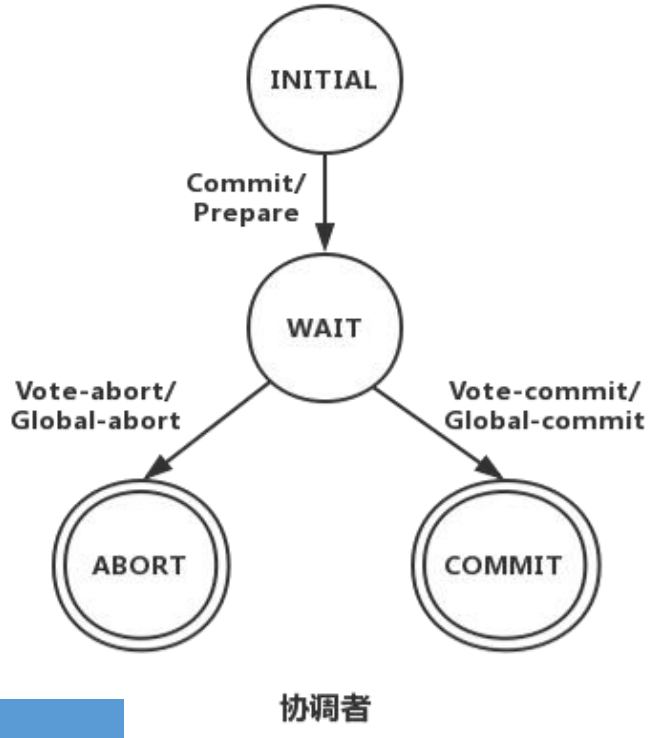
- 两阶段提交协议2PC
  - 最初，协调者会将一个begin_commit记录写到日志中，然后向所有参与站点发送prepare消息，并进入WAIT状态
  - 当一个参与者收到prepare消息时，它会检查是否可以提交事务，如果可以那么参与者就会将一个ready记录写入到日志中，然后向协调者发送一个vote-commit消息，并进入READY状态；否则参与者将一个abort记录写入到日志中，然后向协调者发送一个vote-abort消息
  - 如果参与者站点的决策是取消，那么它仅需将事务丢弃即可，因为取消的决策是具有否决权的
  - 如果有任何一个参与者投了否决票，那么协调者就需要执行一个全局取消，这时，协调者将一个abort记录写到日志中，然后向所有参与者发送global-abort消息并进入ABORT状态
  - 如果所有参与者都发送了提交消息，协调者就将一个commit写入到日志中，然后向所有参与者发送global-commit消息，并进入COMMIT状态
  - 参与者或者提交，或者取消，完全取决于协调者的指示，还要向协调者发送确认消息
  - 在收到确认消息之后，协调者终结事务，并将一个end_of_transaction记录写到日志中
  - **以上是不考虑故障的2PC协议**

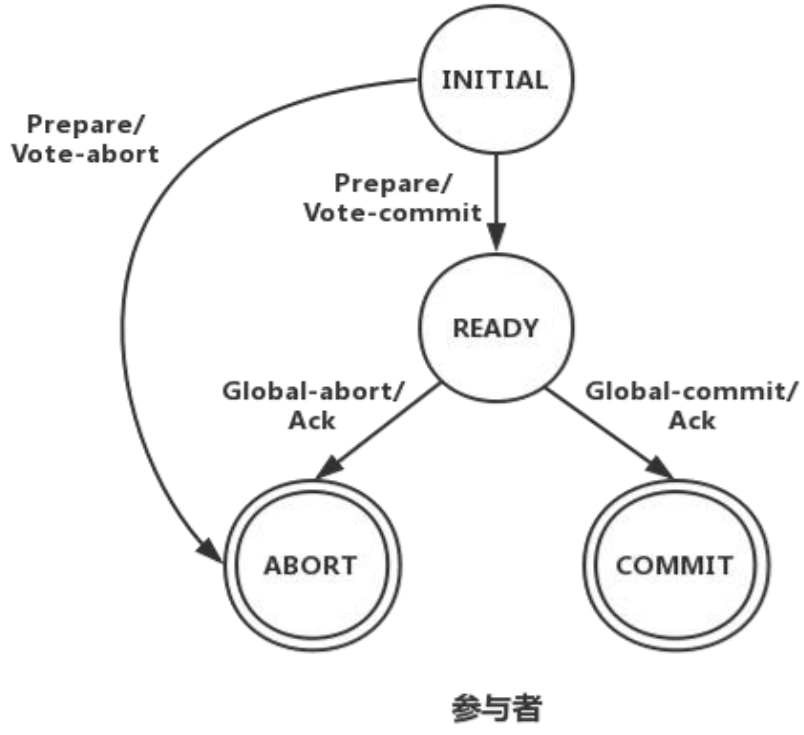
分布式DBMS的可靠性

- 两阶段提交协议2PC的通信结构
  - 集中式2PC
  - 线性2PC或嵌套式2PC
  - 分布式2PC
  - 2PC的变型版本：优化消息数量和写日志的次数
    - 假定取消2PC协议：当一个已经READY的参与者在任何时候向协调者发出对一个事务结果的询问，而在虚拟存储中却没有关于该事务的信息时，那么对这种事务询问的回答一律是将该事务取消
    - 假定提交2PC协议：如果不存在这个事务的信息，那么参与者就认为这个事务已经提交了

分布式DBMS的可靠性

- 分布式可靠性协议
  - 提交协议在分布式事务中的原子性，原子提交
  - 终结协议：阻塞/非阻塞的
  - 恢复协议：独立/非独立的
  - 独立的恢复协议意味着非阻塞终结协议，但反之并不成立
  - 2PC协议是阻塞的





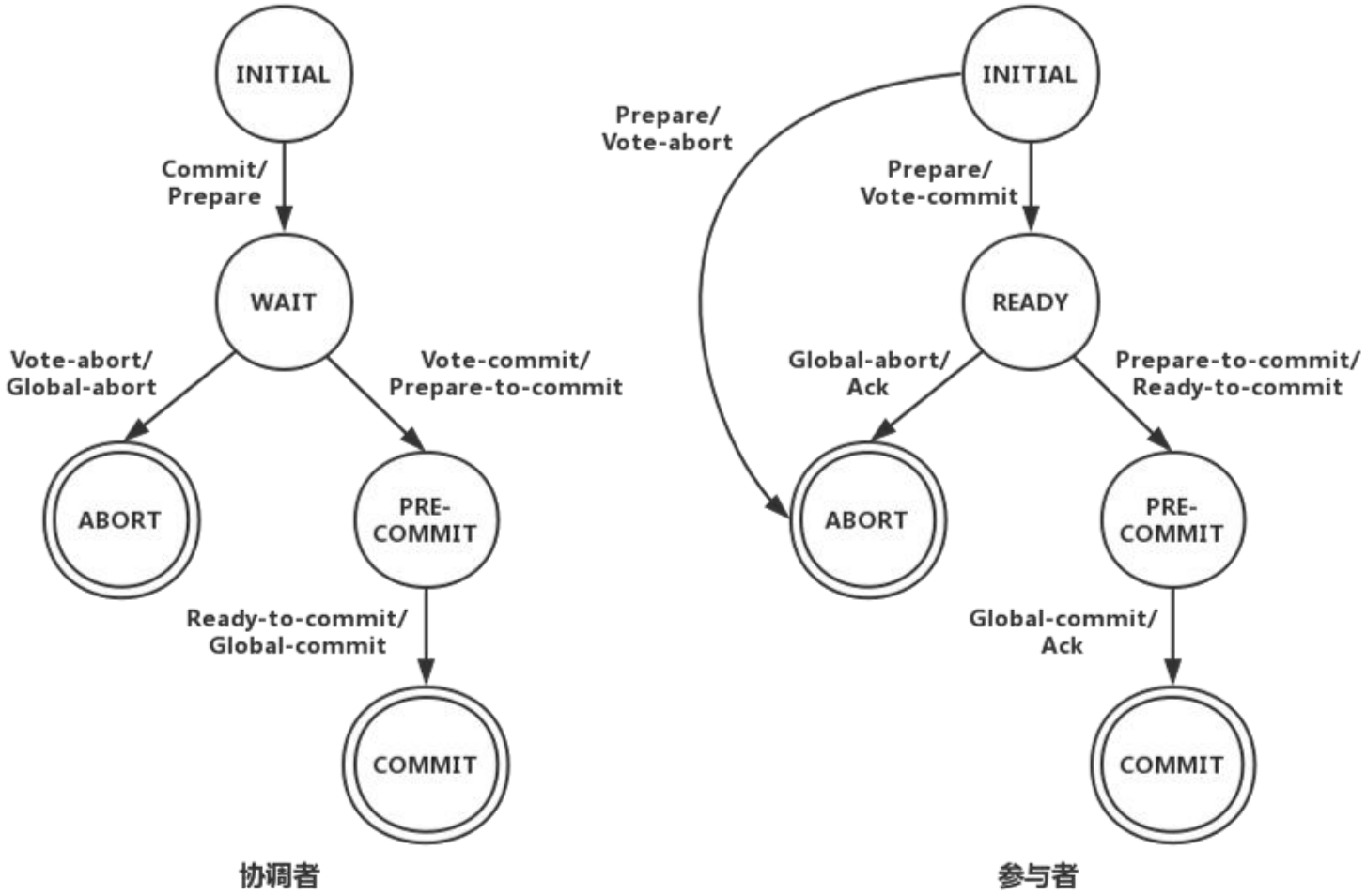
两阶段提交协议的状态转移图

	协调者	参与者
终结协议	WAIT超时	INITIAL超时
	COMMIT和ABORT超时	READY超时
恢复协议	INITIAL故障	INITIAL故障
	WAIT故障	READY故障
	COMMIT和ABORT故障	COMMIT和ABORT故障

分布式DBMS的可靠性

- 三阶段提交协议
  - 在只有站点故障时，是非阻塞的

	协调者	参与者
终结协议	WAIT超时	INITIAL超时
	PRECOMMIT超时	READY超时
	COMMIT和ABORT超时	PRECOMMIT超时
恢复协议	INITIAL故障	INITIAL故障
	WAIT故障	READY故障
	PRECOMMIT故障	PRECOMMIT故障
	COMMIT和ABORT故障	COMMIT和ABORT故障



三阶段提交协议的状态转移图

分布式DBMS的可靠性

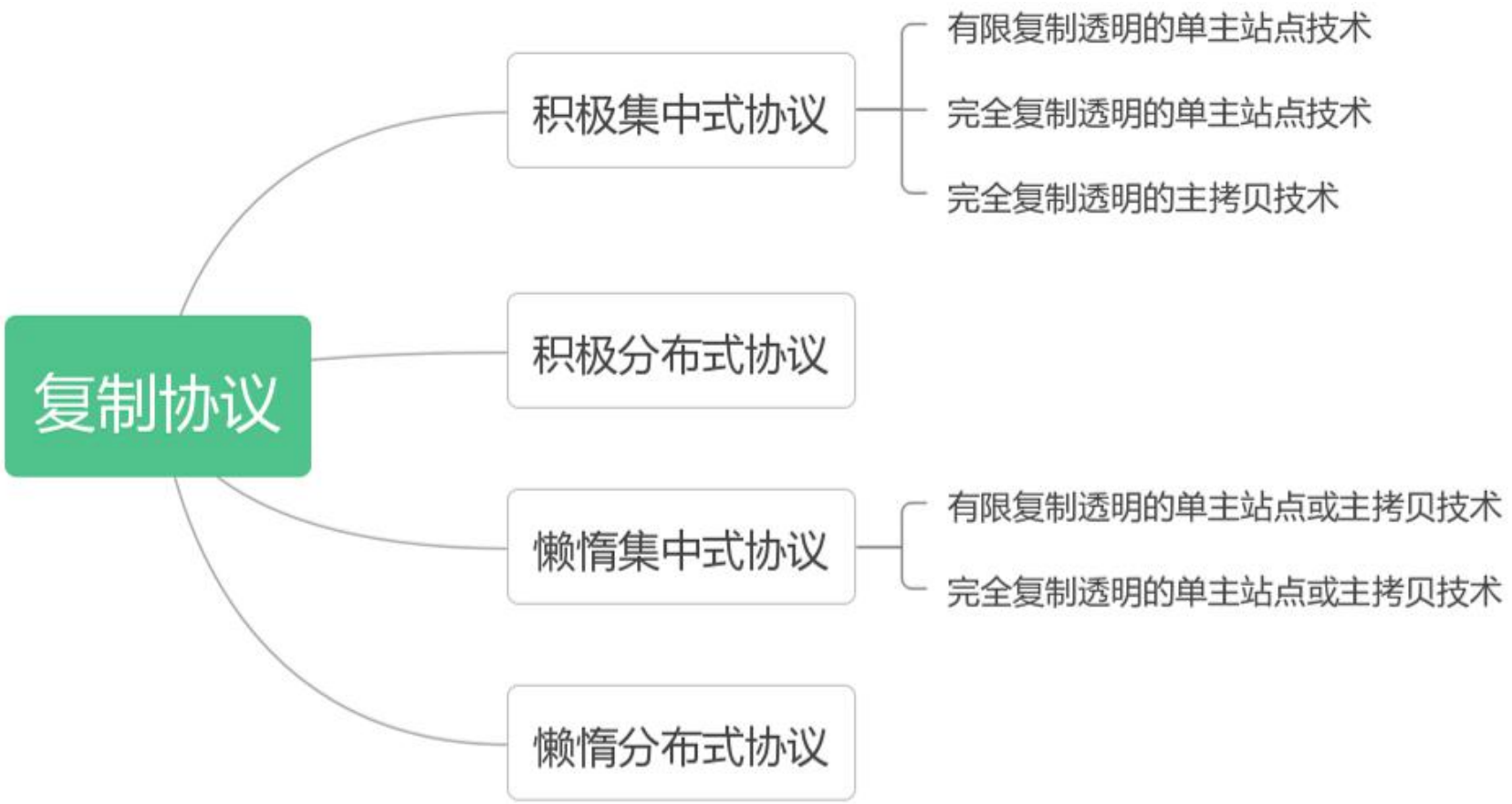
- 网络划分（网络分区）
  - 源于通信线路故障，会依据通信子网的实现方式不同造成不同程度的消息丢失
  - 如果网络被简单地划分成了两个部分，就叫做**简单划分**，否则叫做**多划分**
  - 找到一个网络划分情况下的非阻塞的终结协议是不可能的，就不存在非阻塞的原子提交协议
- 集中式协议
  - 集中式提交：允许包含中心站点的分片保持运行
  - 主站点技术：只有包含写集中的数据项的主站点所在的分片可以执行事务
- 基于投票的协议
  - 基本思想：如果大多数站点都投票执行一个事务，那么这个事务就会被执行
  - 基于限额（quorum）的投票方法，取消限额，提交限额

数据复制

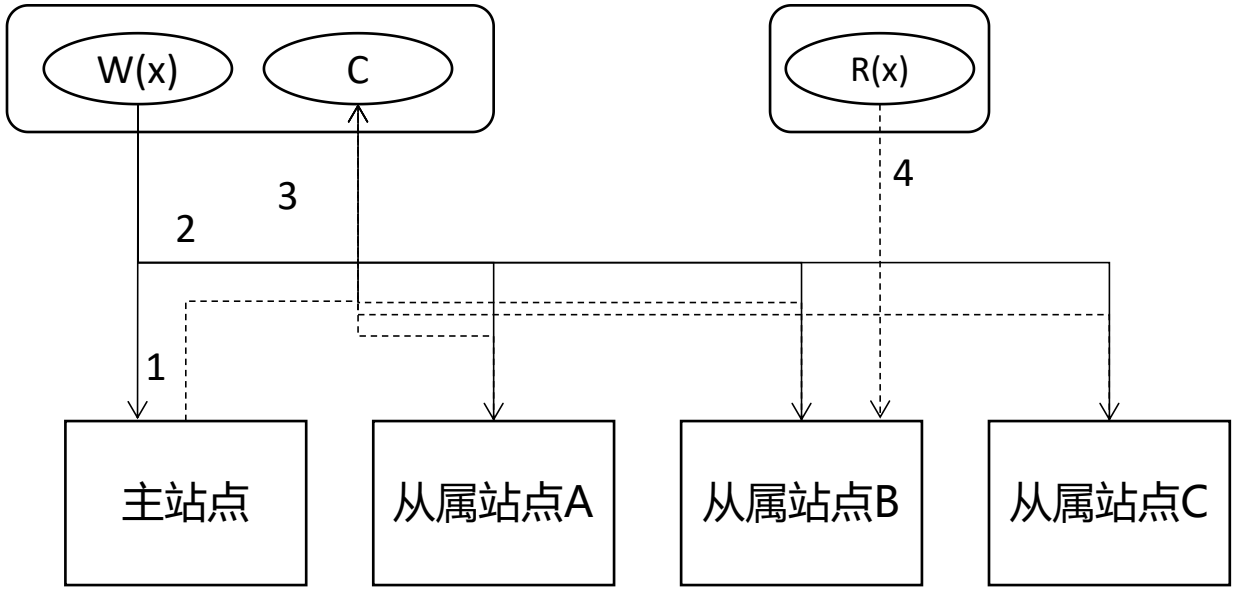
- 数据复制的需要：系统可用性，性能，可扩展性，应用需求.....
- 设计决策因素
  - 事务范围：局部事务/全局事务
  - 数据库一致性，数据项副本同步的紧密程度：强一致性，弱一致性（最终一致性）
  - 在何处进行更新：集中式（主单站点和主拷贝）/分布式
  - 更新传播：积极传播/懒惰传播
  - 复制的透明度：有限复制透明/完全复制透明

数据复制

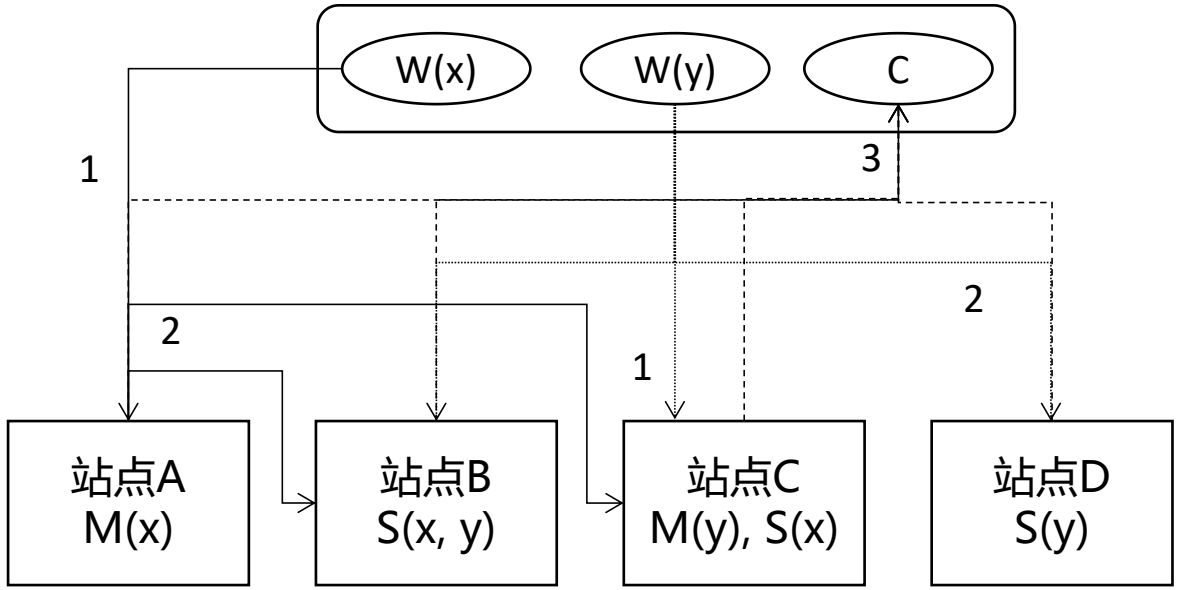
- 复制协议


复制协议
积极集中式协议
有限复制透明的单主站点技术
完全复制透明的单主站点技术
完全复制透明的主拷贝技术
积极分布式协议
懒惰集中式协议
有限复制透明的单主站点或主拷贝技术
完全复制透明的单主站点或主拷贝技术
懒惰分布式协议

数据复制

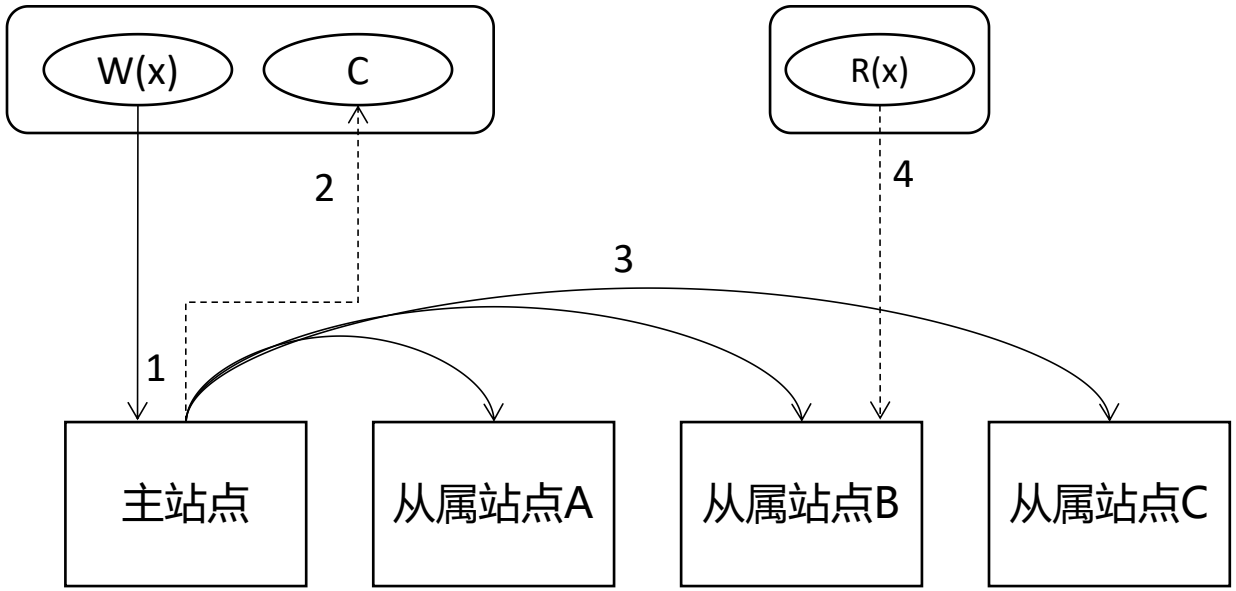


积极单主站点复制协议的操作过程

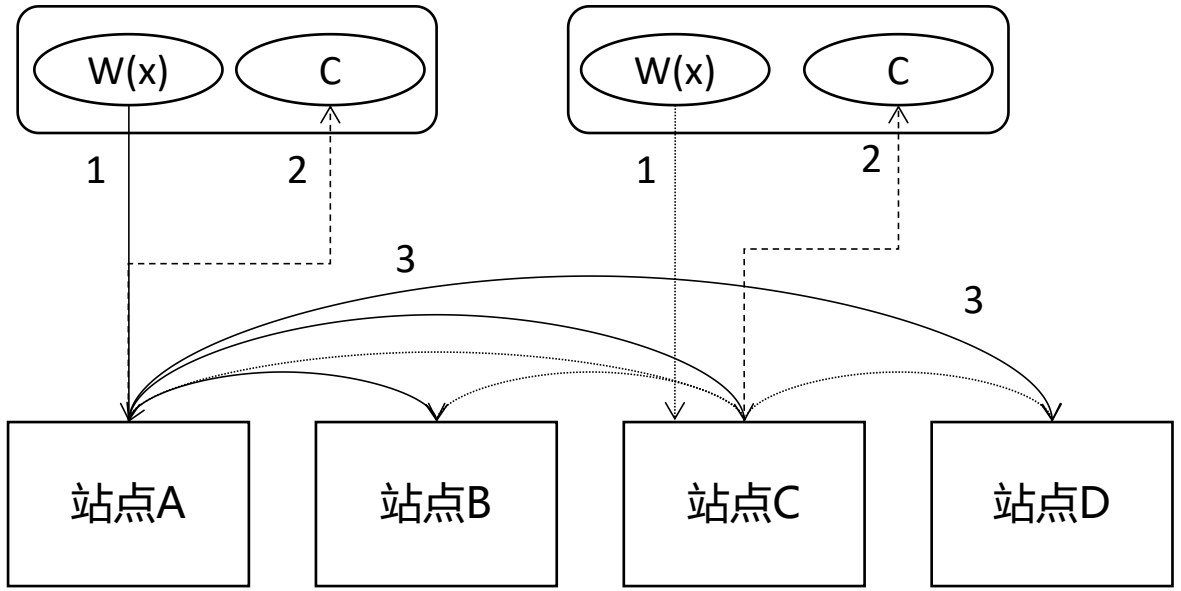


积极主拷贝复制协议的操作过程

数据复制



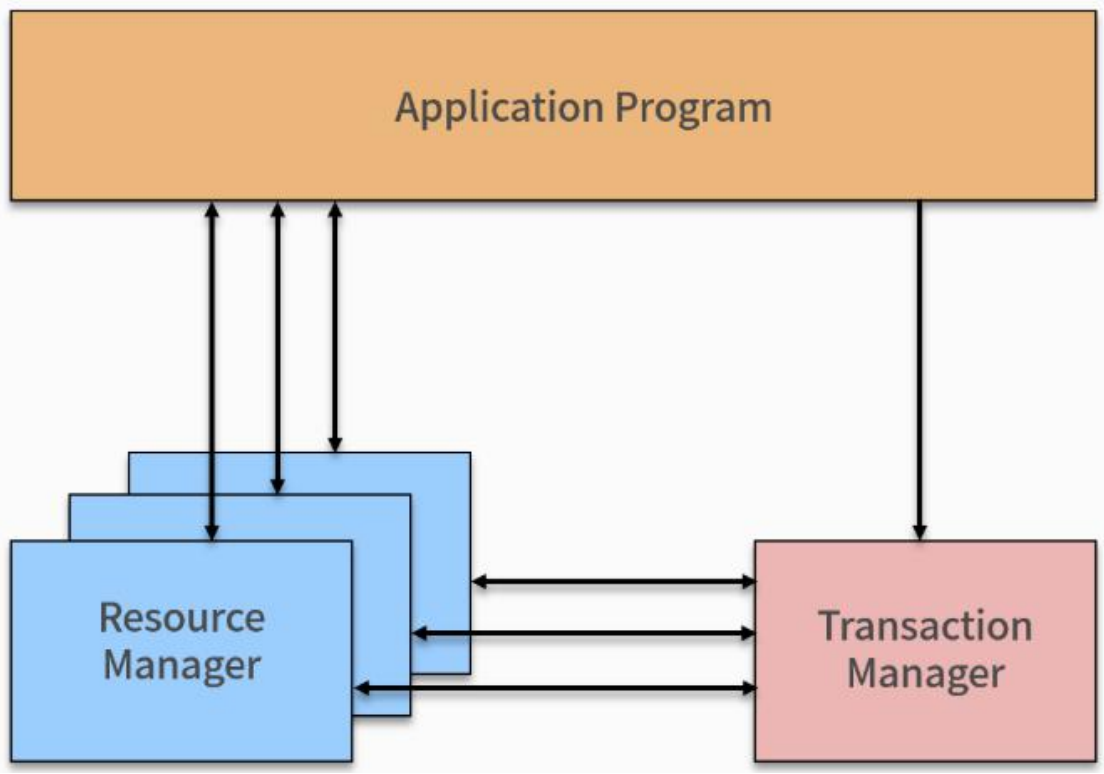
懒惰单主站点复制协议的操作过程



懒惰分布式复制协议的操作过程

应用：XA事务模型

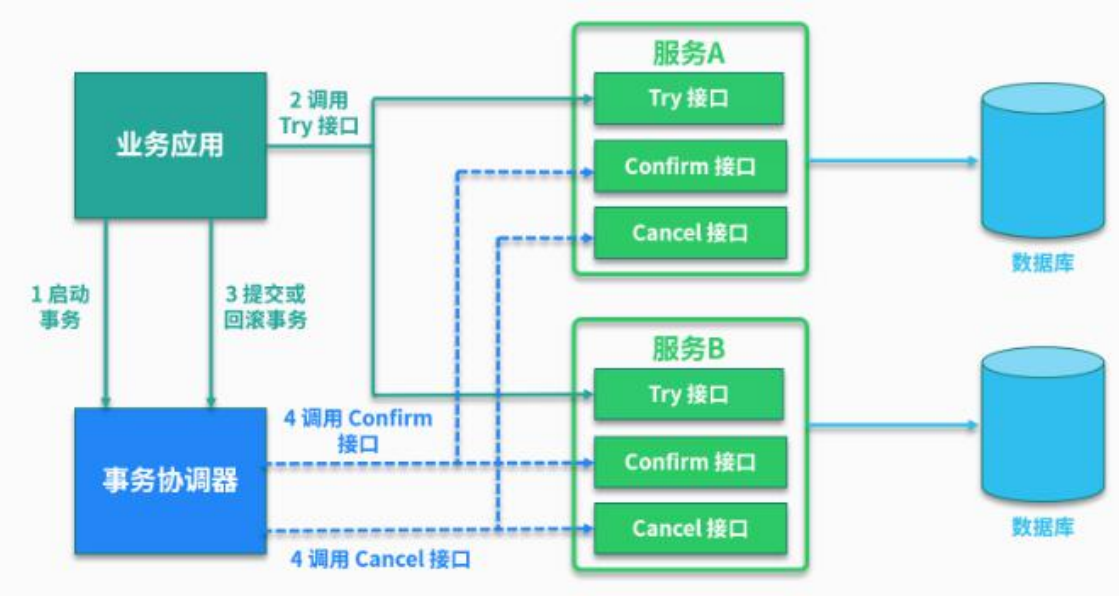
- XA 是由 X/Open 组织提出的分布式事务规范，XA 规范主要定义了事务协调者（Transaction Manager）和资源管理器（Resource Manager）之间的接口
- XA 事务是基于两阶段提交协议的
- 资源管理器负责控制和管理实际资源，比如数据库或 JMS 队列



https://cloud.tencent.com/developer/article/1627837

应用：TCC事务模型

- TCC 提出了一种新的事务模型，基于业务层面的事务定义，锁粒度完全由业务自己控制，目的是解决复杂业务中，跨表跨库等大颗粒度资源锁定的问题
- TCC 把事务运行过程分成 Try、Confirm / Cancel 两个阶段，每个阶段的逻辑由业务代码控制，避免了长事务，可以获取更高的性能
- TCC 解决了跨服务的业务操作原子性问题，比如下订单减库存，多渠道组合支付等场景，通过 TCC 对业务进行拆解，可以让应用自己定义数据库操作的粒度，可以降低锁冲突，提高系统的业务吞吐量



https://cloud.tencent.com/developer/article/1627837

扩展资料：Pat Helland. Life beyond Distributed Transactions: an Apostate’s Opinion. 2007.

应用：Saga分布式模型

- Saga是一个长活事务可被分解成可以交错运行的子事务集合
- 其中每个子事务都是一个保持数据库一致性的真实事务
- Saga不提供ACID保证，因为原子性和隔离性不能得到满足
- Saga协调模式：
  - 编排（Choreography）：在saga参与者中分配决策和排序，他们主要通过交换事件进行沟通
  - 控制（Orchestration）：在saga控制类中集中saga的协调逻辑，一个saga控制者向saga参与者发送命令消息，告诉他们要执行哪些操作
- 可靠的消息中间件是系统中的关键
- https://www.jianshu.com/p/e4b662407c66
- https://docs.microsoft.com/zh-cn/azure/architecture/reference-architectures/saga/saga

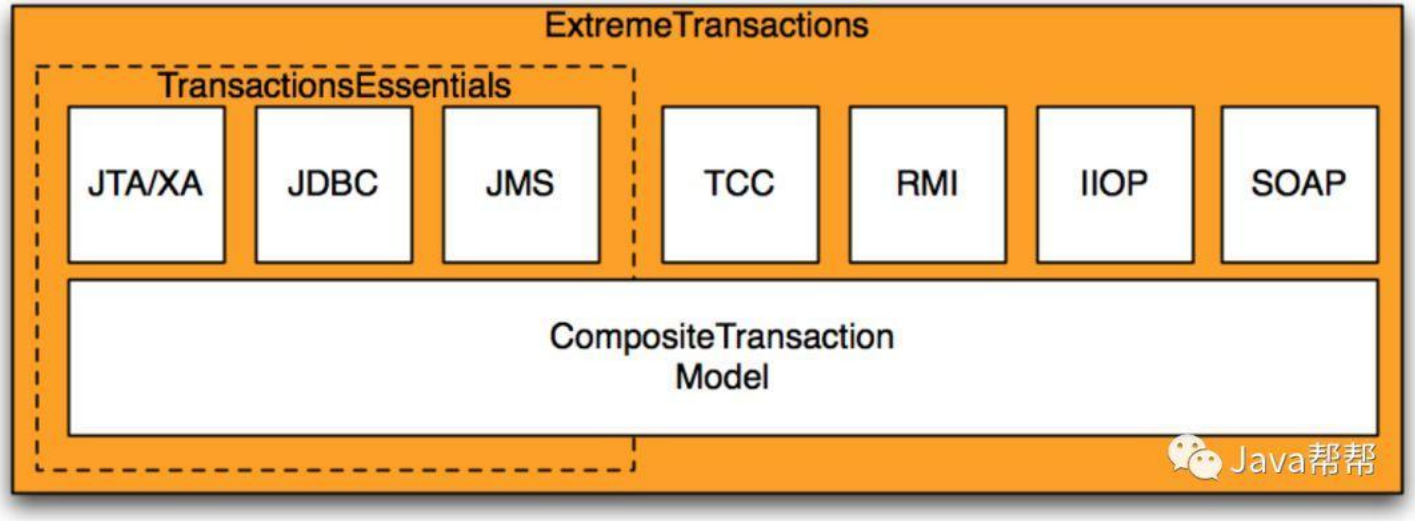
扩展资料：Hector Garcaa-Molrna, Kenneth Salem. SAGAS. 1987.

实现：MySQL XA

- https://dev.mysql.com/doc/refman/8.0/en/xa.html
- https://cloud.tencent.com/developer/article/1378789
- https://www.jianshu.com/p/7003d58ea182
- https://zhuanlan.zhihu.com/p/48586408
- https://mysqlserverteam.com/improvements-to-xa-support-in-mysql-5-7/

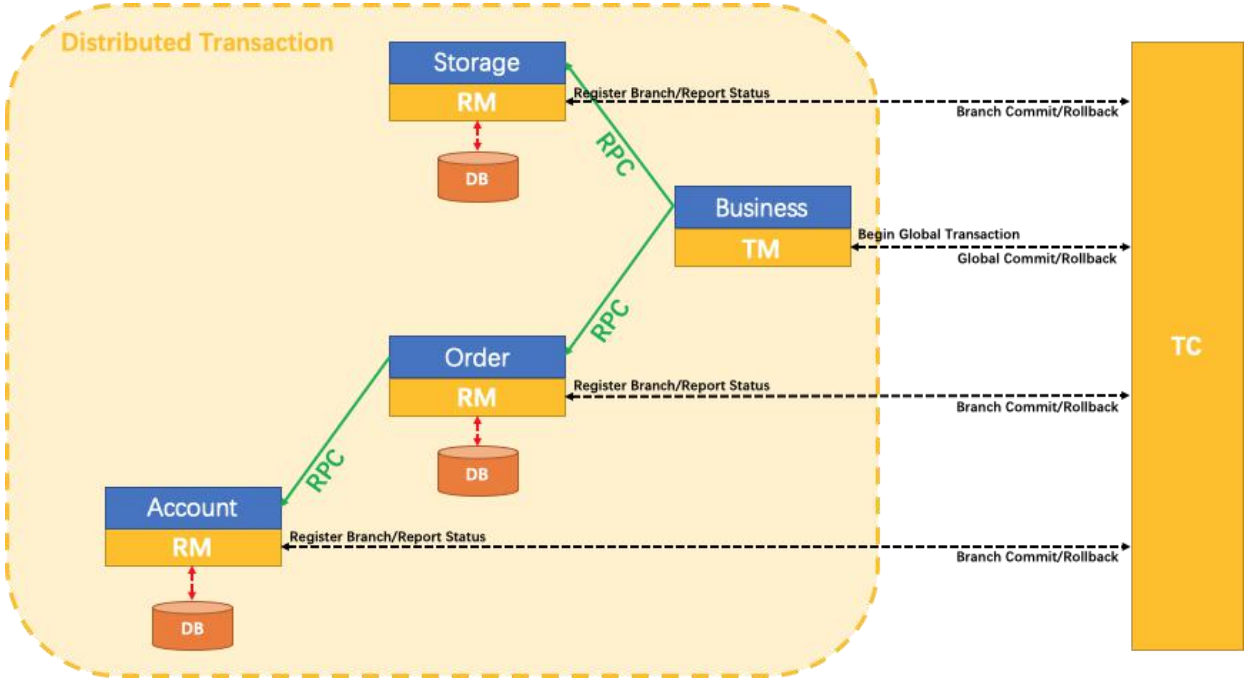
实现：Atomikos

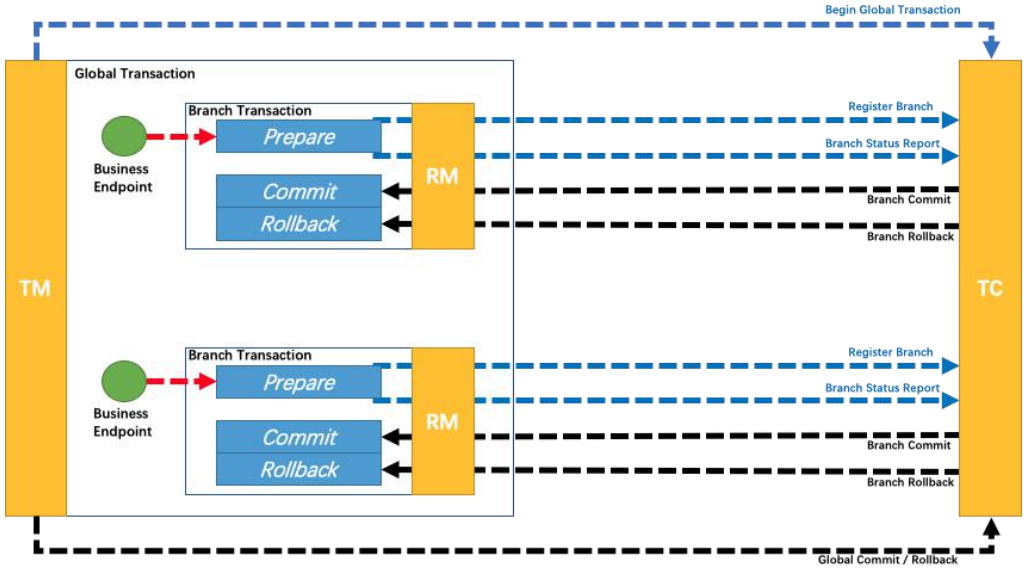
- https://www.atomikos.com/
- https://www.baeldung.com/java-atomikos
- https://blog.minhow.com/articles/java/spring-boot-atomikos/
- https://www.jianshu.com/p/f9bac5822d30
- https://blog.csdn.net/qq_33449307/article/details/102550878



实现：Seata

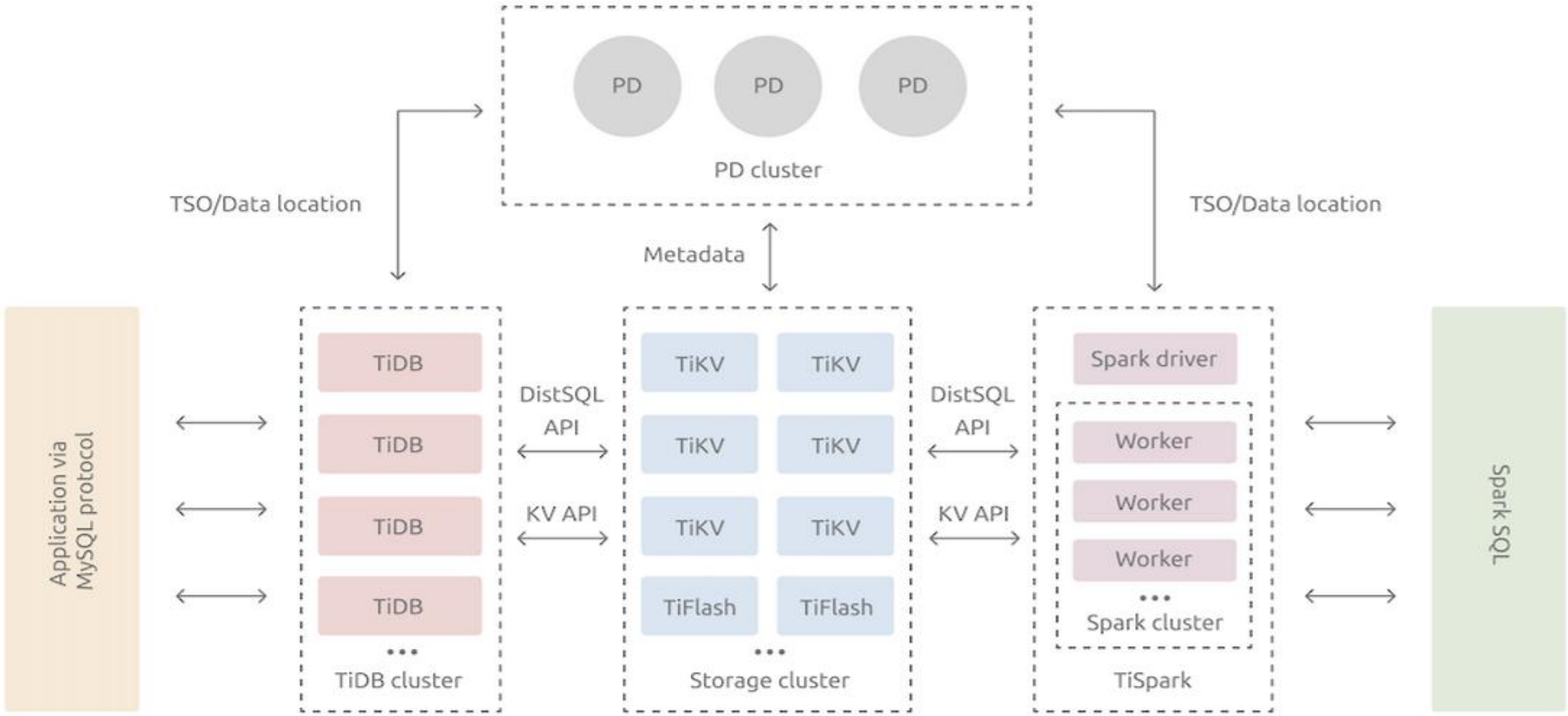
- https://seata.io/
- 支持AT、TCC、SAGA、XA模式
- https://www.cnblogs.com/huanchupkblog/p/12185851.html





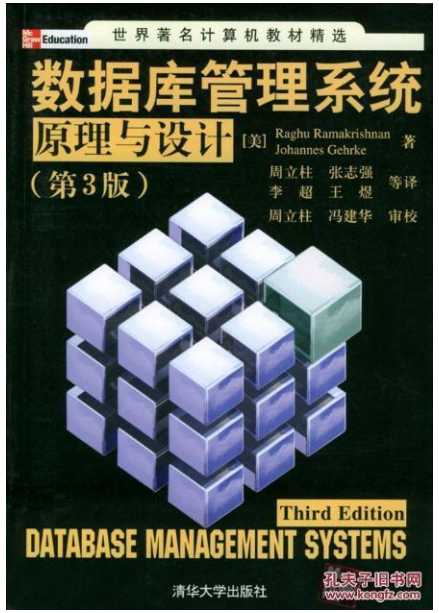
实现：TiDB

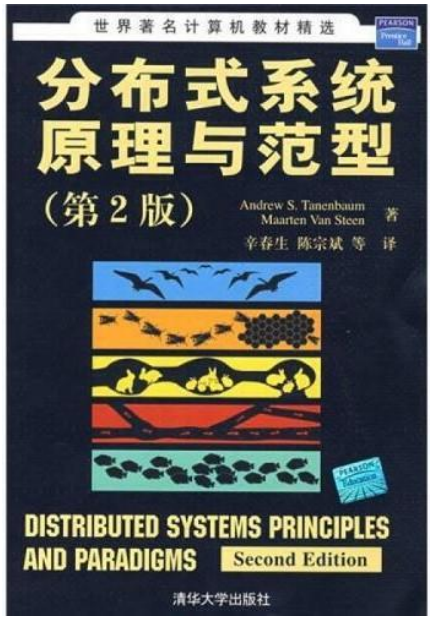
https://pingcap.com/zh/
https://docs.pingcap.com/zh/tidb/stable/transaction-overview

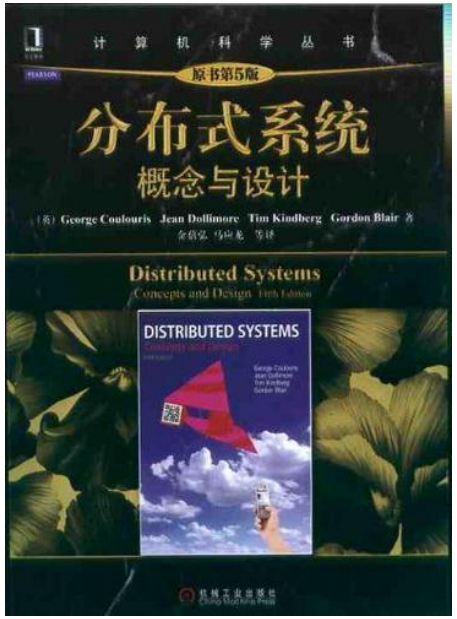


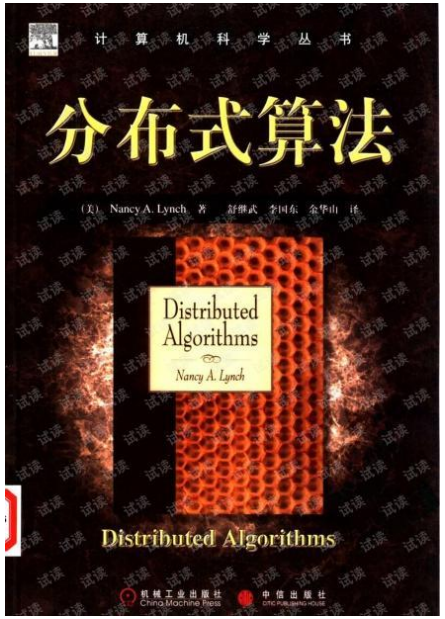
必读书籍


分布式数据库
系统原理（第3版）
PRINCIPLES OF DISTRIBUTED
DATABASE SYSTEMS Third Edition


世界著名计算机教材精选
数据库管理系统
原理与设计
（第3版）
Third Edition
DATABASE MANAGEMENT SYSTEMS
清华大学出版社


世界著名计算机教材精选
分布式系统
原理与范型
（第2版）
DISTRIBUTED SYSTEMS PRINCIPLES
AND PARADIGMS Second Edition
清华大学出版社


计算机科学丛书
原书第5版
分布式系统
概念与设计
Distributed Systems
DISTRIBUTED SYSTEMS


计算机科学丛书
分布式算法
Distributed
Algorithms
Nancy A. Lynch
Distributed Algorithms

谢谢观赏！