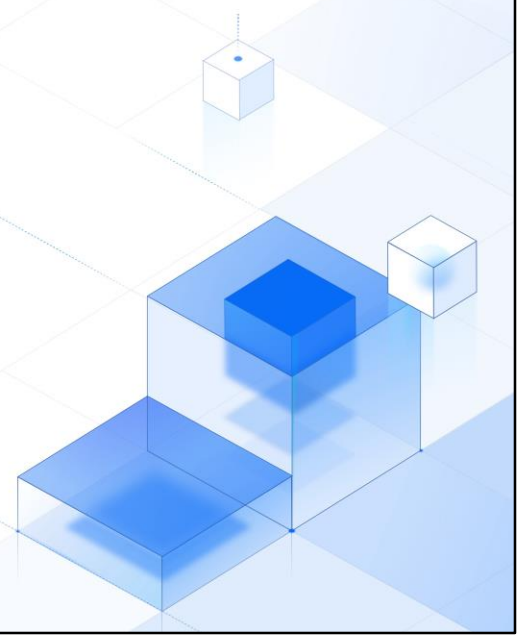
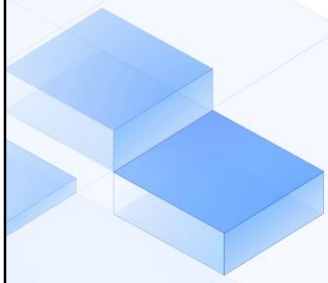
OCEANBASE

OBCP 认证培训

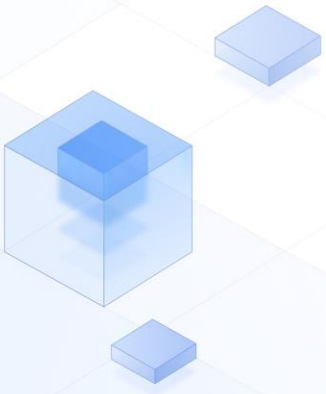


目录



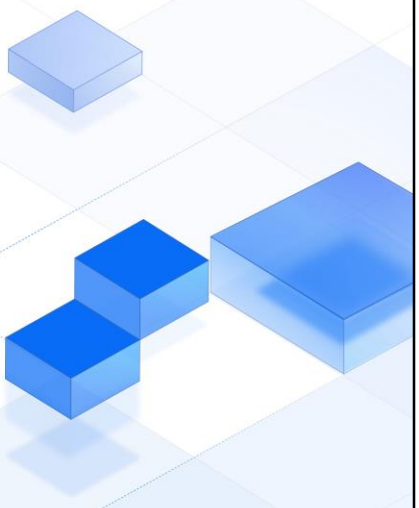
OCEANBASE

目录



OCEANBASE

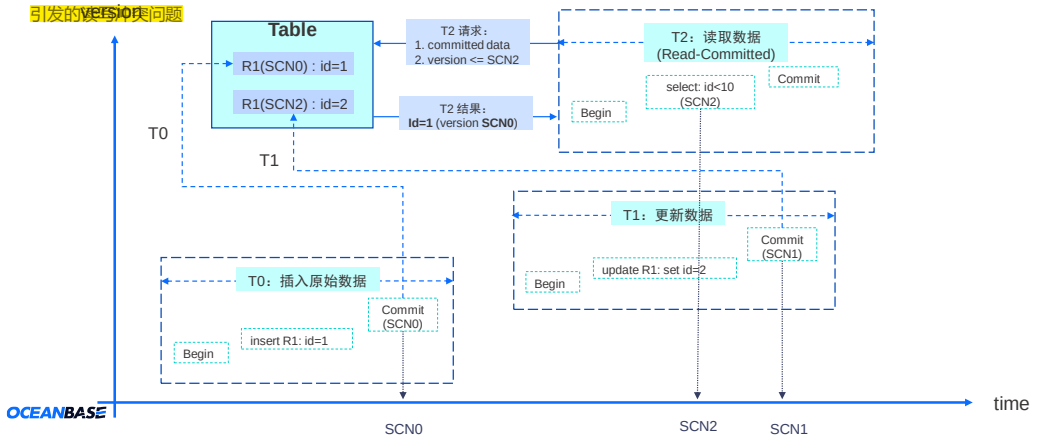
5.1 全局快照及分布式一致性读



OCEANBASE

5.1 传统数据库的实现原理

“快照隔离级别（Snapshot Isolation）”和“多版本并发控制（Multi-VersionConcurrency Control，简称MVCC）”：两种技术的大致含义是：为数据库中的数据维护多个版本号（即多个快照），当数据被修改的时候，可以利用不同的版本号区分出正在被修改的内容和修改之前的内容，以此实现对同一份数据的多个版本做并发访问，避免了经典实现中“锁”机制引发的读写冲突问题



我们来看一下传统数据库中是如何实现“快照隔离级别”和“多版本并发控制”的。以经典的Oracle数据库为例，当数据的更改在数据库中被提交的时候，Oracle会为它分配一个“System Change Number（SCN）”作为版本号。SCN是一个和系统时钟强相关的值，可以简单理解为等同于系统时间戳，不同的SCN代表了数据在不同时间点的“已提交版本（Committed Version）”，由此实现了数据的快照隔离级别。

假设一条记录最初插入时对应的版本号为SCN0，当事务T1正在更改此记录但还未提交的时候（注意：此时T1对应的SCN1尚未生成，需要等到T1的commit阶段），Oracle会将数据更改之前的已提交版本SCN0放到“回滚段（Undo Segments）”中保存起来，此时如果有另外一个并发事务T2要读取这条记录，Oracle会根据当前系统时间戳分配一个SCN2给T2，并按照两个条件去寻找数据：

1）必须是已提交（Committed）的数据；

2）数据的已提交版本（Committed Version）是小于等于SCN2的最大值。

根据上面的条件，事务T2会从回滚段中获取到SCN0版本所对应的数据，并不理会正在同一条记录上进行修改的事务T1。利用这种方法，既避免了“脏读（Dirty Read）”的发生，也不会导致并发的读/写操作之间产生锁冲突，实现了数据的

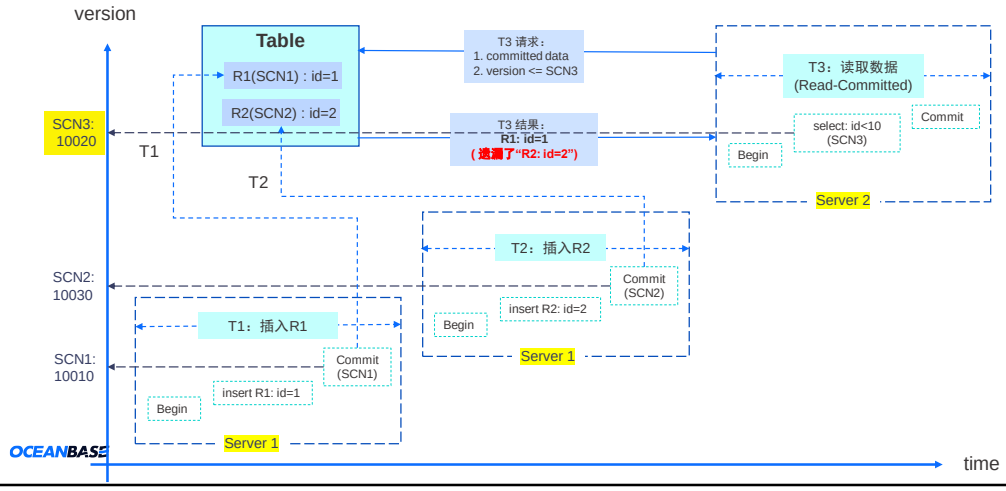
多版本并发控制。

关于“快照隔离级别”和“多版本并发控制”，不同数据库产品的实现机制会有差异，但大多遵循以下原则：

每次数据的更改被提交时，都会为数据分配一个新的版本号。
版本号的变化必须保证“单调向前”。
版本号取自系统时钟里的当前时间戳，或者是一个和当前时间戳强相关的值。
查询数据时，也需要一个最新版本号（同理，为当前时间戳或者和当前时间戳强相关的值），并查找小于等于这个版本号的最近已提交数据。

5.1 分布式数据库面临的挑战

和传统的数据库的单点全共享（即Shared-Everything）架构不同，OceanBase是一个原生的分布式架构，采用了多点无共享（即Shared-Nothing）的架构，在实现全局（跨机器）一致的快照隔离级别和多版本并发控制时会面临分布式架构所带来的技术挑战



关于"多版本并发控制"的描述看上去很完美，但是这里面却有一个隐含的前提条件：**数据库中版本号的变化顺序必须和真实世界中事务发生的时间顺序保持一致**，即：

— 真实世界中较早发生的事务必然获取更小（或者相等）的版本号；

— 真实世界中较晚发生的事务必然获取更大（或者相等）的版本号。

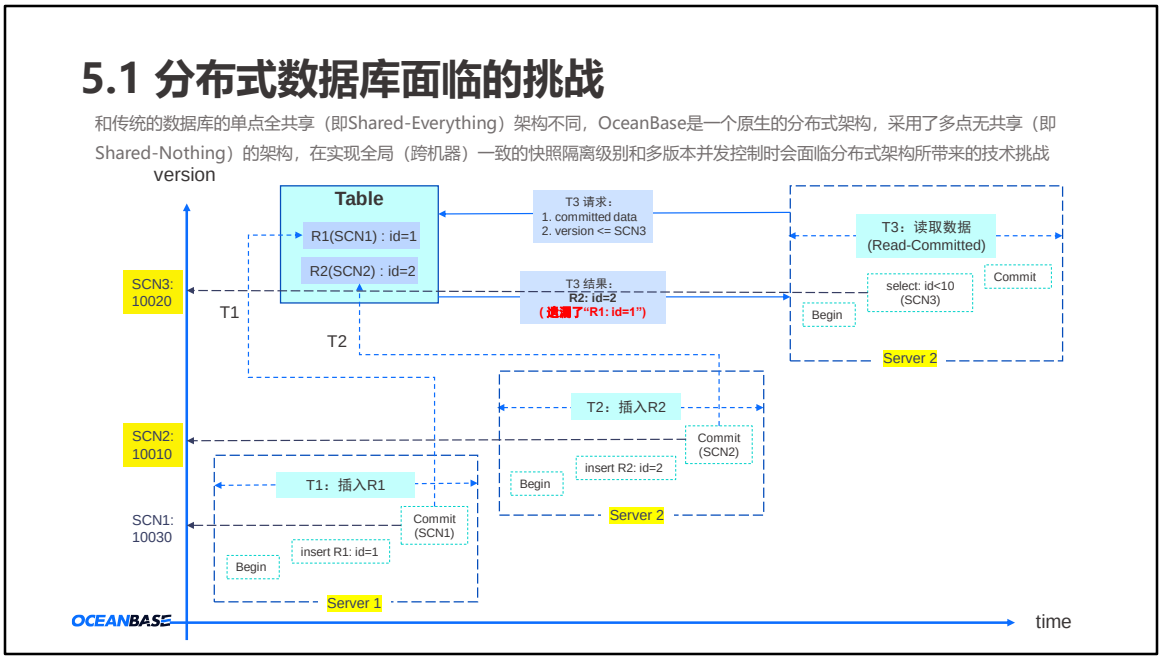
如果不能满足这个一致性，会导致什么结果呢？以下面的场景为例：

1）记录R1首先在事务T1里被插入并提交，对应的SCN1是10010；

2）随后，记录R2在事务T2里被插入并提交，对应的SCN2是10030；

3）随后，事务T3要读取这两条数据，它获取的SCN3为**10020**，因此它只获取到记录R1（SCN1<SCN3，满足条件），而没有获取到R2（SCN2>SCN3，不满足条件）

这对应用来说就是一个逻辑错误：我明明向数据库中插入了两条记录并且都提成功提交了，但却只能读到其中的一条记录。导致这个问题的原因，就是这个场景违反了上面所说的一致性，即**SCN（版本号）的变化顺序没有和真实世界中事务发生的时间顺序保持一致**。



这种一致性还可能引发更极端的情况，考虑下面的场景：
1）记录R1首先在事务T1里被插入并提交，对应的SCN1是10030；
2）随后，记录R2在事务T2里被插入并提交，对应的SCN2是**10010**；
3）随后，事务T3要读取这两条数据，它获取的SCN3为**10020**，因此它只能获取到记录R2（SCN2<SCN3，满足条件），而无法获取到R1（SCN1>SCN3，不满足条件）

对于应用来说，这种结果从逻辑上讲更加难以理解：先插入的数据查不到，后插入的数据反而能查到，完全不合理。
有的朋友可能会说：上面这些情况在实际中是不会发生的，因为系统时间戳永远是单调向前的，因此真实世界中先提交的事务一定有更小的版本号。是的，对于传统数据库来说，由于采用单点全共享（Shared-Everything）架构，数据库只有一个系统时钟来源，因此时间戳（即版本号）的变化的确能做到单调向前，并且一定和真实世界的时间顺序保持一致。
但对于OceanBase这样的分布式数据库来说，由于采用无共享（Shared-Nothing）架构，数据分布和事务处理会涉及不同的物理机器，而多台物理机器之间的系统时钟不可避免存在差异，如果以本地系统时间戳作为版本号，则无法保证不同机器上获取的版本号和真实世界的时间序保持一致。还是以上面的两个场景为例，如果T1、T2和T3分别在不同的物理机器上执行，并且它们都分别以本地

的系统时间戳作为版本号，那么由于机器间的时钟差异，完全可能发生上面所说的两种异常。

5.1 业界常用解决方案

分布式数据库应如何在全局（跨机器）范围内保证外部一致性，进而实现全局一致的快照隔离级别和多版本并发控制呢？大体来说，业界有两种实现方式：

- 利用特殊的硬件设备，如GPS和原子钟（Atomic Clock），使多台机器间的系统时钟保持高度一致，误差小到应用完全无法感知的程度。

- 版本号不再依赖各个机器自己的本地系统时钟，所有的数据库事务通过集中式的服务获取全局一致的版本号，由这个服务来保证版本号的单调向前。

OCEANBASE

第一种方式的典型代表是Google的Spanner数据库。它使用GPS系统在全球的多个机房之间保持时间同步，并使用原子钟确保本地系统时钟的误差一直维持在很小的范围内，这样就能保证全球多个机房的系统时钟能够在一个很高的精度内保持一致，这种技术在Spanner数据库内被称为TrueTime。在此基础上，Spanner数据库就可以沿用传统的方式，以本地系统时间戳作为版本号，而不用担心破坏全局范围内的外部一致性。
这种方式的好处，是软件的实现比较简单，并且避免了采用集中式的服务可能会导致的性能瓶颈。但这种方式也有它的缺点，首先对机房的硬件要求明显提高，其次“GPS+原子钟”的方式也不能100%保证多个机器之间的系统时钟完全一致，如果GPS或者原子钟的硬件偏差导致时间误差过大，还是会出现外部一致性被破坏的问题。根据GoogleSpanner论文中的描述，发生时钟偏差（clock drift）的概率极小，但并不为0。

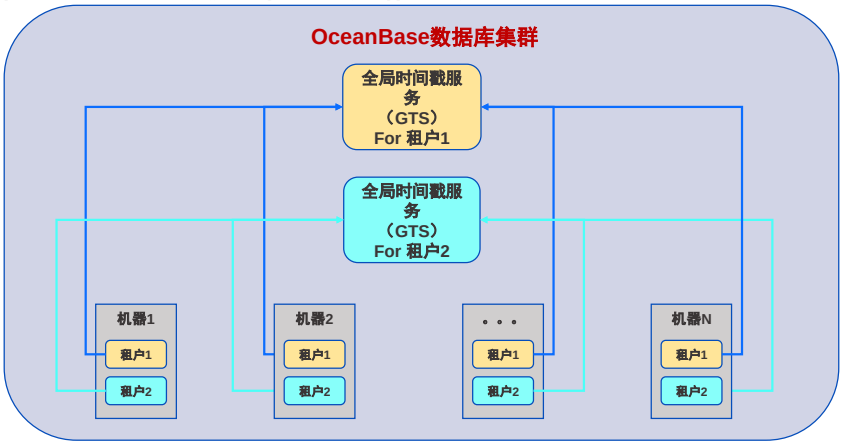
OceanBase则选用了第二种实现方式，即用集中式的服务来提供全局统一的版本号。做这个选择主要是基于以下考虑：

1、可以从逻辑上消除机器间的时钟差异因素，从而彻底避免这个问题。

2、避免了对特殊硬件的强依赖。这对于一个通用数据库产品来说尤其重要，我们不能假设所有使用OceanBase数据库的用户都在机房里部署了“GPS+原子钟”的设备。

5.1 OceanBase全局一致性快照技术

OceanBase数据库是利用一个集中式服务来提供全局一致的版本号。事务在修改数据或者查询数据的时候，无论请求源自哪台物理机器，都会从这个集中式的服务处获取版本号，OceanBase则保证所有的版本号单调向前并且和真实世界的时间顺序保持一致。



有了这样全局一致的版本号，OceanBase就能根据版本号对全局（跨机器）范围内的数据做一致性快照，因此我们把这个技术命名为**“全局一致性快照”**。有了全局一致性快照技术，就能实现全局范围内一致的快照隔离级别和多版本并发控制，而不用担心发生外部一致性被破坏的情况。

但是，相信有些朋友看到这里就会产生疑问了，比如：

这个集中式服务里是如何生成统一版本号的？怎么能保证单调向前？

这个集中式服务的服务范围有多大？整个OceanBase集群里的事务都使用同一个服务吗？

这个集中式服务的性能如何？尤其在高并发访问的情况下，是否会成为性能瓶颈？

如果这个集中式服务发生中断怎么办？

如果在事务获取全局版本号的过程中，发生了网络异常（比如瞬时网络抖动），是否会破坏“外部一致性”？

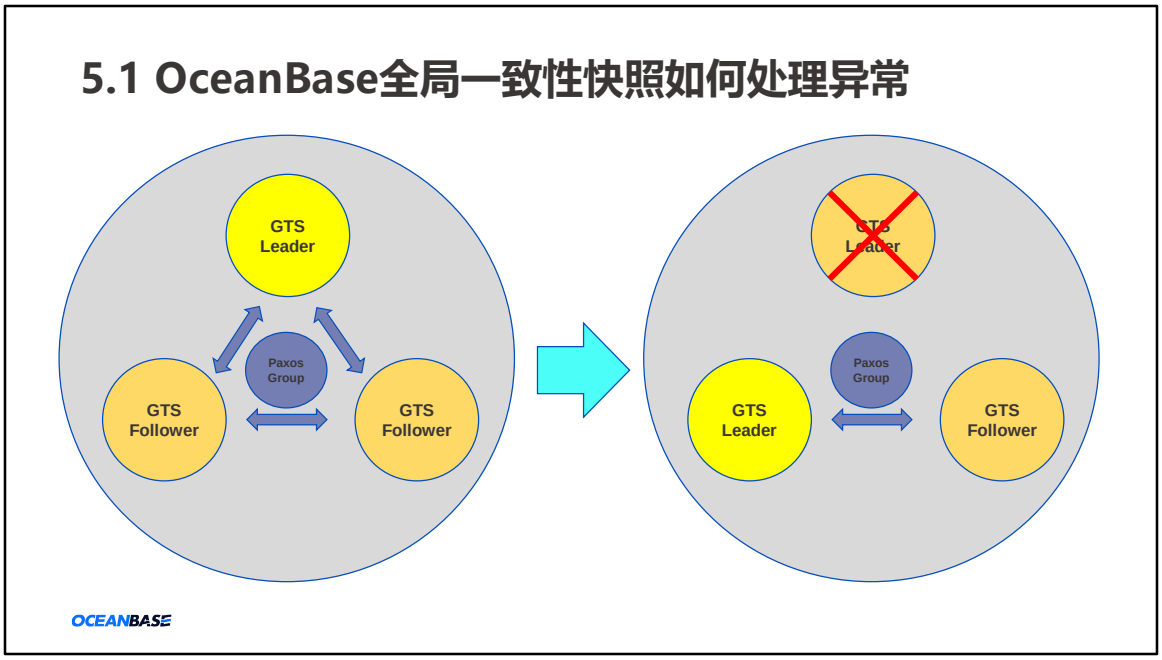
下面针对这些疑问逐一为大家解答。

首先，这个集中式服务所产生的版本号就是本地的系统时间戳，只不过它的服务对象不再只是本地事务，而是全局范围内的所有事务，因此在OceanBase中这个服务被称作**“全局时间戳服务（Global Timestamp Service，简称GTS）”**。由于GTS服务是集中式的，只从一个系统时钟里获取时间戳，因此能保证获取的时间戳（即版本号）一定是单调向前的，并且一定和真实世界的时间顺序保持一

致。

那么，是否一个OceanBase数据库集群中只有一个GTS服务，集群中所有的事务都从这里获取时间戳呢？对OceanBase数据库有了解的朋友都知道，“租户”是OceanBase中实现资源隔离的一个基本单元，比较类似传统数据库中“实例”的概念，不同租户之间的数据完全隔离，没有一致性要求，也无需实现跨租户的全局一致性版本号，因此OceanBase数据库集群中的每一个“租户”都有一个单独的GTS服务。这样做不但使GTS服务的管理更加灵活（以租户为单位），而且也将集群内的版本号请求分流到了多个GTS服务中，大大减少了因单点服务导致性能瓶颈的可能。

说到性能，经过实测，单个GTS服务能够在1秒钟内响应2百万次申请时间戳（即版本号）的请求，因此只要租户内的QPS不超过2百万，就不会遇到GTS的性能瓶颈，实际业务则很难触及这个上限。虽然GTS的性能不是一个问题，但从GTS获取时间戳毕竟比获取本地时间戳有更多的开销，至少网络的时延是无法避免的，对此我们也做过实测，在满负荷压测并且网络正常的情况下，和采用本地时间戳做版本号相比，采用GTS对性能所带来的影响不超过5%，绝大多数应用对此都不会有感知。
除了保证GTS的正常处理性能之外，OceanBase数据库还在**不影响外部一致性**的前提下，对事务获取GTS的流程做了优化，比如：
将某些GTS请求转化为本地请求，从本地控制文件中获取版本号，避免了网络传输的开销；
将多个GTS请求合并为一个批量请求，以提高GTS的全局吞吐量；
利用“本地GTS缓存技术”，减少GTS请求的次数。
这些优化措施进一步提高了GTS的处理效率，因此，使用者完全不用担心GTS的性能。



对于集中式服务来说一定要考虑异常情况。首先就是高可用的问题，GTS服务也像OceanBase中基本的数据服务一样，以Paxos协议实现了高可用，如果GTS服务由于异常情况（比如宕机）而中断，那么OceanBase会根据Paxos协议自动选出一个新的服务节点，整个过程自动而且迅速（1~15秒），无需人工干预。

如果发生网络异常怎么办？比如网络抖动了10秒钟，会不会影响版本号的全局一致性，并进而影响外部一致性？对数据库来说，外部一致性反映的是真实世界中“完整事务”之间的前后顺序，比如前面说到的T1 -> T2-> T3，其实准确来说是T1's Begin-> T1's End -> T2's Begin -> T2's End -> T3's Begin -> T3's End，即任意两个完整的事务窗口之间没有任何重叠。如果发生了重叠，则事务之间并不具备真正的“先后顺序”，外部一致性也就无从谈起

因此，不管网络如何异常，只要真实世界中的“完整事务”满足这种前后顺序，全局版本号就一定会满足外部一致性。
最后，如果事务发现GTS响应过慢，会重新发送GTS请求，以避免由于特殊情况（如网络丢包）而导致事务的处理被GTS请求卡住。
总之，GTS在设计和开发的过程中已经考虑到了诸多异常情况的处理，确保可以提供稳定可靠的服务。

5.1 小结

使用“全局一致性快照”，OceanBase数据库便具备了在全局（跨机器）范围内实现“快照隔离级别”和“多版本并发控制”的能力，可以在全局范围内保证“外部一致性”，并在此基础之上实现众多涉及全局数据一致性的功能，比如全局一致性读、全局索引等。

和传统单点数据库相比，OceanBase在保留分布式架构优势的同时，在全局数据一致性上也没有降级，应用开发者就可以像使用单点数据库一样使用OceanBase，不必担心机器之间的底层数据一致性问题。

OCEANBASE


5.2 分布式两阶段提交
OCEANBASE

5.1 分布式系统中可能会出现的问题

场景 1

- 交易发生时，负责A账户扣100块钱的A机器死机，没有扣成功，而负责给账户B加100块钱的B机器工作正常，加上了100
- 负责给A账户扣100块钱的A机器正常，已经扣掉100，而负责给账户B加100块钱的B机器死机，100块没加上，那么用户就会损失100

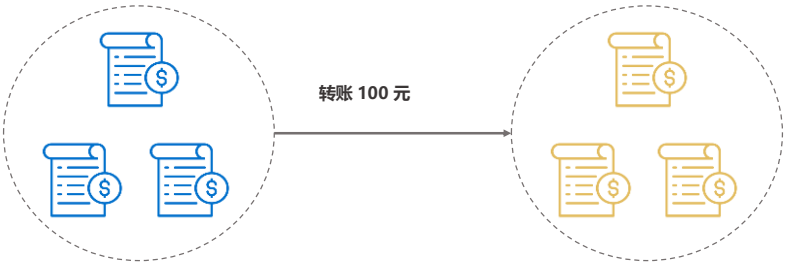
场景 2

- A账户要扣掉100块，但是它的余额只有90块，或者已经达到了今天的转账限额，这种情况下，如果贸然给B账户加了100块，A账户却不够扣，麻烦了
- 如果B账户状态有异常（例如冻结），不能加100块，同样也麻烦了

OCEANBASE

5.1 分布式系统中场景1的解决方案

Paxos 多副本： 账户数据一定同时存在三台机器上。转账时，至少两台机器执行完毕才算转账完成，意味着三台机器里有一台坏掉，并不影响转账的执行



OCEANBASE

5.1 分布式系统中场景2的解决方案

引入裁判员：

1）裁判员问A账户：你的三台机器都没问题吧？A账户说：没问题。 你的账户允许扣100吗？A账户说：允许。

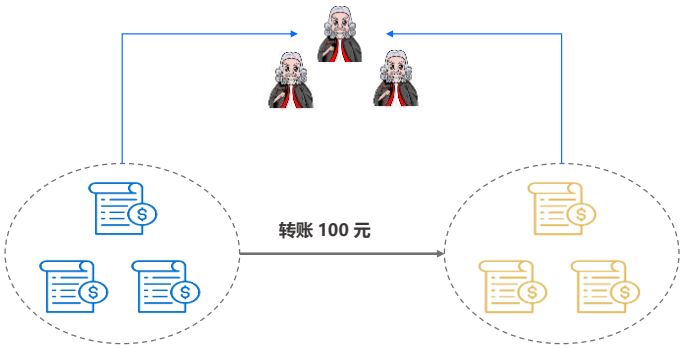
3）裁判员问B账户：你的三台机器都没问题吧？B账户说：没问题。你的账户状态能接受加100吗？B说：允许。

4）这时，裁判员吹哨，A、B账户同时冻结。

5）A扣100，B加100，双方向裁判汇报“成功”。

6）裁判员再吹哨，A、B账户同时解冻。

当然，裁判员也要是多副本才好。



OCEANBASE

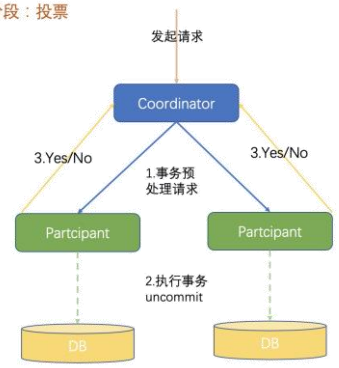
5.1 两阶段协议

2PC是一个非常经典的强一致、中心化的原子提交协议。中心化指的是协调者（Coordinator），强一致性指的是需要所有参与者（partcipant）均要执行成功才算成功，否则回滚。

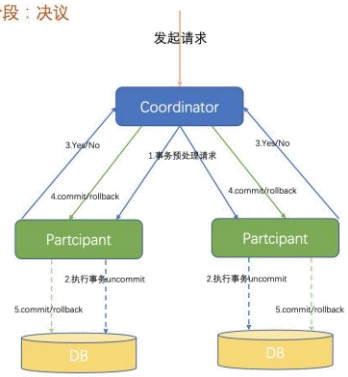
第一阶段：协调者（coordinator)发起提议通知所有的参与者（partcipant），参与者收到提议后，本地尝试执行事务，但并不commit，之后给协调者反馈，反馈可以是yes或者no。

第二阶段：协调者收到参与者的反馈后，决定commit或者rollback，参与者全部同意则commit，如果有一个参与者不同意则rollback。

第一阶段：投票



第二阶段：决议



OCEANBASE

5.2 OceanBase两阶段提交协议

标准两阶段提交协议

- 优点：状态简单，只依靠协调者状态即可确认和推进整个事务状态
- 缺点：协调者写日志，commit延时高

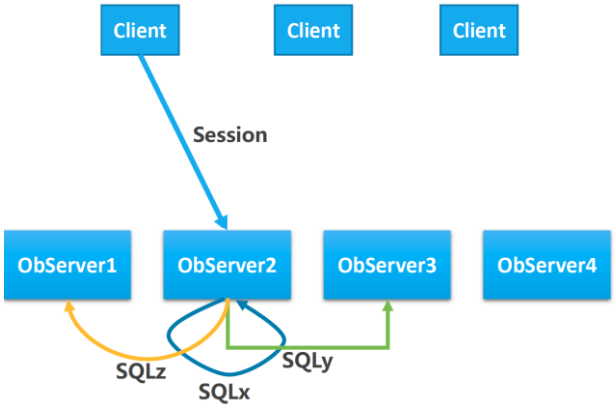
OceanBase两阶段提交协议

- 协调者不写日志，变成了一个无持久化状态的状态机
- 事务的状态由参与者的持久化状态决定
- 所有参与者都prepare成功即认为事务进入提交状态，立即返回客户端commit
- 每个参与者都需要持久化参与者列表，方便异常恢复时构建协调者状态机，推进事务状态
- 参与者增加clear阶段，标记事务状态机是否终止

OCEANBASE

5.2 OB两阶段提交

- 业务不感知是否分布式事务，如果只有一个参与者使用一阶段提交；否则自动使用两阶段提交
- 参与者的实体是 Partition（Px，Py，Pz）
- 制定第一个参与者 Px 作为协调者，发送 end_trans 消息给 Px 并告知参与者列表：Px，Py，Pz

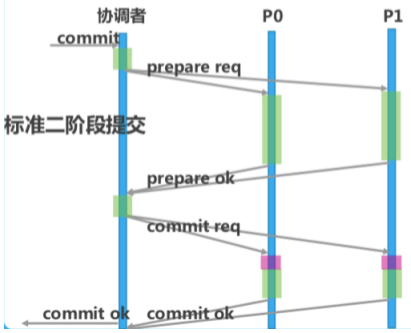


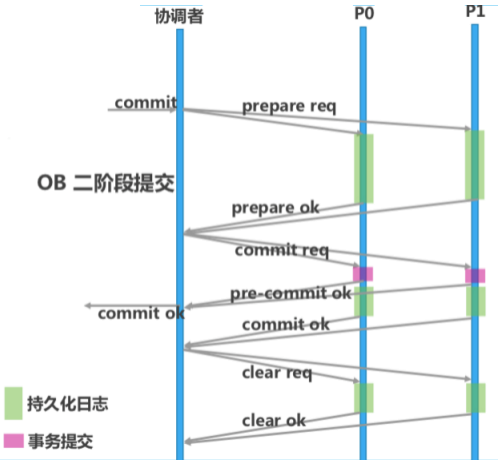
OCEANBASE

5.2 OB两阶段提交延迟分析

用户感知的commit延迟

- 标准：4次日志延迟 + 2次RPC延迟
- OB：1次日志延迟 + 2次RPC延迟

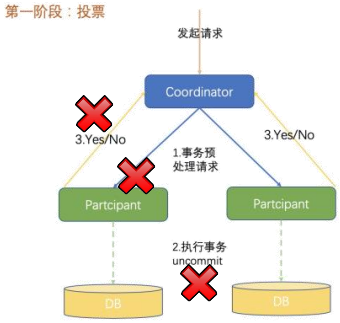




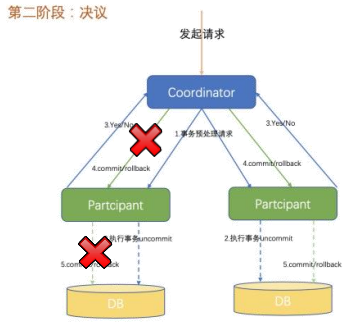
OCEANBASE

5.2 分布式事务的高可用

- 如果事务在prepare状态落盘之前发生宕机，机器恢复后事务会回滚



- 如果事务处于commit阶段，由于clog已经落盘，即使发生宕机场景，事务都会执行完成，只是业务端可能会收到事务unknown的回复，需要业务端confirm事务的状态

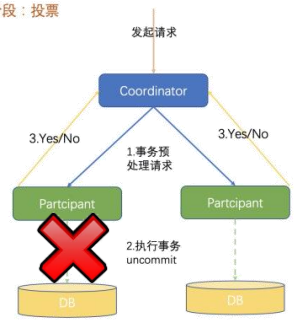


5.2 两阶段提交过程中参与者宕机

还未进入Prepared状态

- 参与者所有事务状态丢失
- 参与者会应答协调者prepare unknown消息
- 事务最终会abort

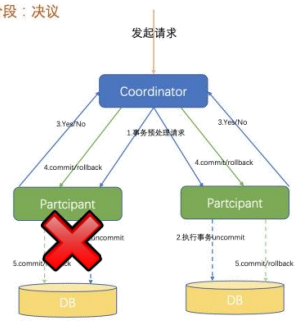
第一阶段：投票



已进入Prepared状态

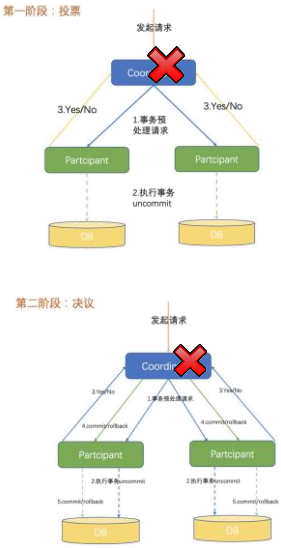
- 状态已经Paxos同步
- 系统会自动选择一个副本，作为新的leader并恢复出prepare状态，协调者继续推进

第二阶段：决议



5.2 两阶段提交过程中协调者宕机

- 协调者与第一个参与者是同一个partition
  - 参与者状态机恢复遵从参与者自己的逻辑
  - 协调者状态机恢复由参与者回复协调者的消息触发
- 参与者发送prepare ok后未收到协调者进一步消息（commit/abort）时，认为上一条回复消息丢失，会定时重新发送上一条消息
- 所有参与者都记录全部参与者列表



5.2 分布式事务调优方法

- 业务数据模型设计原则：尽量避免跨机分布式事务
- 单sql语句不建议跨机器
  - 建议在业务设计阶段，通过table group把相关的表放在同一个机器上
- 单语句读跨机的partition
  - 打开全局快照（OB2.x，GTS）
  - 用弱一致性读 (/*READ_CONSISTENCY(WEAK) */) （OB1.x，OB2.x）


感谢学习
OCEANBASE