同济大学

TONGJI UNIVERSITY

OceanBase 事务管理探究

学　　院　电子与信息工程学院

专　　业　计算机科学与技术

学生姓名　龚宣　李元特　司盛宇　顾屹洋　明添识　何诗锟

学　　号　2152095 2150260 2152472 2150259 2151569 2153698

指导教师　关佶红

日　　期　2023 年 12 月 17 日

OceanBase 事务管理探究

摘要

阿里巴巴和蚂蚁集团完全自研的原生分布式关系数据库 OceanBase，是一款极其优秀的开源数据库。在普通硬件上实现金融级高可用，首创"三地五中心"城市级故障自动无损容灾新标准，具备卓越的水平扩展能力，也是全球首家通过 TPC-C 标准测试的分布式数据库，单集群规模超过 1500 节点。OceanBase 具有云原生、强一致性、兼容 Oracle/MySQL 等特性，目前其正承担支付宝 100%核心链路，并且做到了在国内几十家银行、保险公司等金融客户的核心系统中稳定运行。

对此，我们对 OceanBase 数据库的事务处理方面进行了深入的调研和分析，并详细的从 OceanBase 事务的概念、事务管理的特点如并发性、一致性入手进行分析，并且也找到了对应的 OceanBase 源码进行分析，完整的介绍了其功能，解释了其原理、阐释了其应用。

本报告将就调查结果做出完整的叙述，从而揭示 OceanBase 高性能分布式部署下事务处理的原理。

关键词： OceanBase, 事务, 并发性, 一致性

目 录

1 OceanBase 简介

OceanBase，由蚂蚁金服和阿里巴巴于 2010 年完全自主研发的分布式关系型数据库，旨在解决传统关系型数据库在分布式环境下面临的扩展性和一致性挑战。它是一个基于无共享架构设计的关系型数据库管理系统（RDBMS），在能源、政府、交通等多个行业得到广泛应用。OceanBase 的主要产品包括 OceanBase 云服务和同名数据库系统。

OceanBase 在性能方面表现卓越。例如，2019 年在 TPC-C 基准测试中，OceanBase 以其处理速度是第二名 Oracle 的两倍荣获第一名。同年 11 月，该数据库处理了每秒 6100 万次的交易记录。这些成就不仅体现了其在处理大规模、复杂交易方面的先进性和高性能，还展示了其数据强一致性和高可用性的特点。OceanBase 通过分布式事务处理技术，实现了在多个节点上的事务操作，而不会出现数据不一致的情况，确保了数据的准确性和可靠性。

OceanBase 的架构优势包括高可用性和高性能。通过读写分离和数据分级（基线数据与增量数据）的处理方式，OceanBase 实现了数据的快速修改和高效访问。同时，它支持在线扩展，以适应不断增长的数据存储和访问需求。

在兼容性方面，OceanBase 高度符合 SQL 标准和主流关系型数据库，简化了数据库迁移和应用的过程。它还提供了丰富的编程接口和工具，支持多种编程语言，降低了开发和维护的复杂度。

成本效益是 OceanBase 的另一大优势。通过数据编码压缩技术，它实现了高压缩比，降低了存储成本。同时，OceanBase 支持自动化运维和智能监控，进一步减少了人工干预和运营成本。

OceanBase 的实际应用证明了其在核心业务系统中的可靠性和性能优势。它不仅被广泛应用于支付宝和阿里巴巴的多个系统，还服务于南京银行、浙商银行、人保健康险等多个外部客户。

综上可见，OceanBase 凭借其数据强一致性、高可用性、高性能、在线扩展能力、高度兼容性以及低成本等特点，在数据库领域脱颖而出。而本研究报告将聚焦于 OceanBase 的**事务管理技术**，探究其高性能运行原理，并由此洞察一般数据库系统的运作机制。

2 OceanBase 中事务的概念

数据库事务包含了数据库上的一系列操作，事务使得数据库从一个一致的状态转化到另一个一致的状态。

2.1 数据库事务的特性

数据库事务具有 4 个特性：**原子性**、**一致性**、**隔离性**、**持久性**。这四个属性通常称为 ACID 特性。

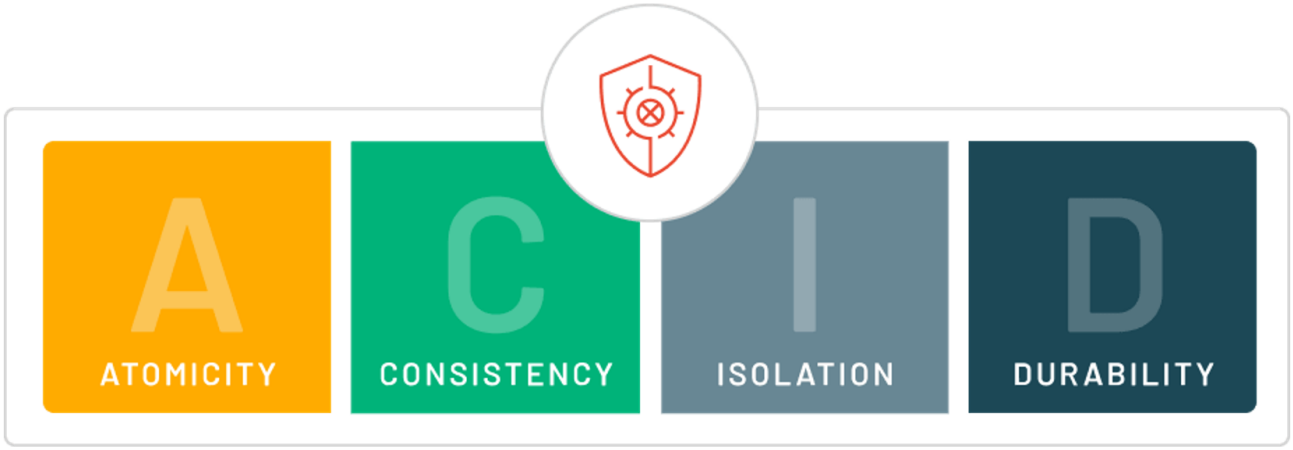

图 2.1 事务“ACID”示意图

2.1.1 原子性（Atomicity）

在 OceanBase 中，原子性是确保每个事务操作要么完全完成，要么完全不发生，从而维护了数据库的完整性。而 OceanBase 是通过分布式系统中的两阶段提交协议（2PC）来保持原子性。根据 2.2.2 章节中介绍，可知传统的标准两阶段提交协议具有状态简单，只依靠协调者状态即可确认和推进整个事务状态的特点，但也存在着相应的缺点，即协调者写日志，commit 延时高。

而 OceanBase 的两阶段提交协议对此进行了优化。首先，协调者不再写日志，变成了一个无持久化状态的状态机。其次，事务的状态由参与者的持久化状态决定。再者，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。此外，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 最后，参与者增加 clear 阶段，用来标记事务状态机是否终止。

2.1.2 一致性（Consistency）

OceanBase 利用 Mulit-Paxos 协议在多个数据副本之间保证一致性。这个协议确保所有副本同步，维持数据库的统一状态。

在实际操作中，Multi-Paxos 协议通过多轮决策和投票过程，确保在分布式环境下的数据一致性。每个决策过程中，提案者提出变更请求，而批准者对这些请求进行投票。只有当大多数 Acceptor 同意时，提案才会被接受，保证了决策的有效性和准确性。这种机制尤其适合于 OceanBase 这样的大规模分布式系统，因为它能有效处理节点故障、网络延迟等常见分布式环境问题，保障数据一致性和系统的高可用性。因此，Multi-Paxos 协议在 OceanBase 中不仅提高了事务处理的可靠性，也优化了系统对高并发和复杂事务的处理能力。

而 OceanBase 中的一致性原则也确保事务将数据库从一个有效状态转移到另一个有效状态，维护了数据的整体完整性。

2.1.3 隔离性（Isolation）

在 OceanBase 中，隔离性是处理并发事务时的关键特性，它通过提供不同级别的隔离选项来管理并发操作的交互。OceanBase 支持与 Oracle 和 MySQL 兼容的模式，并根据这些模式提供不同的隔离级别：

在 Oracle 兼容模式下，OceanBase 支持 Read Committed 和 Serializable 这两种隔离级别。Read Committed 级别保证一个事务不会读取到其他未提交事务的数据，而 Serializable 级别提供严格的事务隔离，确保事务执行的结果就如同它们是顺序执行的一样。

在 MySQL 兼容模式下，OceanBase 除了提供 Read Committed 和 Serializable 隔离级别之外，还支持 Repeatable Read 级别。Repeatable Read 级别保证在同一个事务中多次读取同一数据的结果是一致的，即使在此期间其他事务进行了修改。

通过这些隔离级别，OceanBase 确保了并发事务的正确执行，防止了事务之间的相互干扰和可能导致的数据异常。这种灵活的隔离级别设计使 OceanBase 能够有效地平衡事务的隔离性和系统的性能，适应不同的应用场景和需求。

2.1.4 持久性（Durability）

OceanBase 实现数据持久性的机制主要依赖于重做日志（Redo Log）和预写日志（WAL, Write-Ahead Logging）机制。这意味着在任何操作正式写入数据库之前，OceanBase 首先会将其记录在日志文件中。这样做的好处是，一旦事务被提交，相关的更改就能够被保证持久化，即使发生系统故障也不会丢失这些数据。

此外，OceanBase 在集群级别上通过应用 Paxos 协议来进一步强化数据的持久性。该协议通过将数据复制到多个副本，确保只要大多数副本保持存活，就能保障事务数据的安全，即使部分节点出现故障。这种机制提高了 OceanBase 在处理分布式数据时的可靠性和稳健性。

2.2 数据库事务实现基础

2.2.1 语句级回滚

当一条语句执行过程中没有报错，那么该语句所做的修改都是成功的，如果一条语句执行过程中报错，那么该语句执行的操作都会被回滚，这种情况称为语句级回滚。例如，如果一个事务包含两条 UPDATE 语句，第一条执行成功，第二条执行失败，则只有第二条语句的效果会被回滚。

而语句级回滚的效果相当于该语句未曾执行过。这意味着该语句执行过程中涉及的所有元素，如全局索引、触发器、行锁等，都会回滚到语句执行前的状态。由此，语句级回滚保证了数据库事务的原子性。

2.2.2 两阶段协议

二阶提交协议是最早提出解决分布式一致性的方案。二阶段协议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。

具体来说，在第一阶段，协调者发起提议通知所有的参与者，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。在第二阶段，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。

OceanBase 数据库实现了原生的两阶段提交协议，保证分布式事务的原子性。

2.2.3 Paxos 协议

Paxos 算法由 Leslie Lamport 在 1990 年提出，它是少数在工程实践中被证实的强一致性、高可用、去中心的分布式协议。由于篇幅原因，在这里只能做出简单的介绍。

Paxos 协议包含三类角色：Proposer（提案者）、Acceptor（批准者）和 Learner（学习者）。Proposer 提出议案（value），该议案必须得到超过半数的 Acceptor 批准才能通过。Learner 不参与决策，但会学习被批准的 value。实际应用中，一个节点可以同时充当这三种角色。

Paxos 协议按轮次进行，每轮都有一个编号，并可能批准一个 value。如果某一轮批准了某个 value，则后续的 Paxos 过程只能批准这个 value，这是协议正确性的基础。每轮协议分为准备阶段和批准阶段，Proposer 和 Acceptor 各自执行不同的处理流程。

为解决 Paxos 协议可能引起的“活锁”问题，引入了 Multi-Paxos 算法。Multi-Paxos 在 Basic Paxos 的基础上做了改进，包括针对每个要确定的值运行一个 Paxos 实例，并在所有 Proposers 中选举一个 Leader，由 Leader 提交 Proposal。这样减少了 Proposer 之间的竞争和冲突，提高了系统效率。

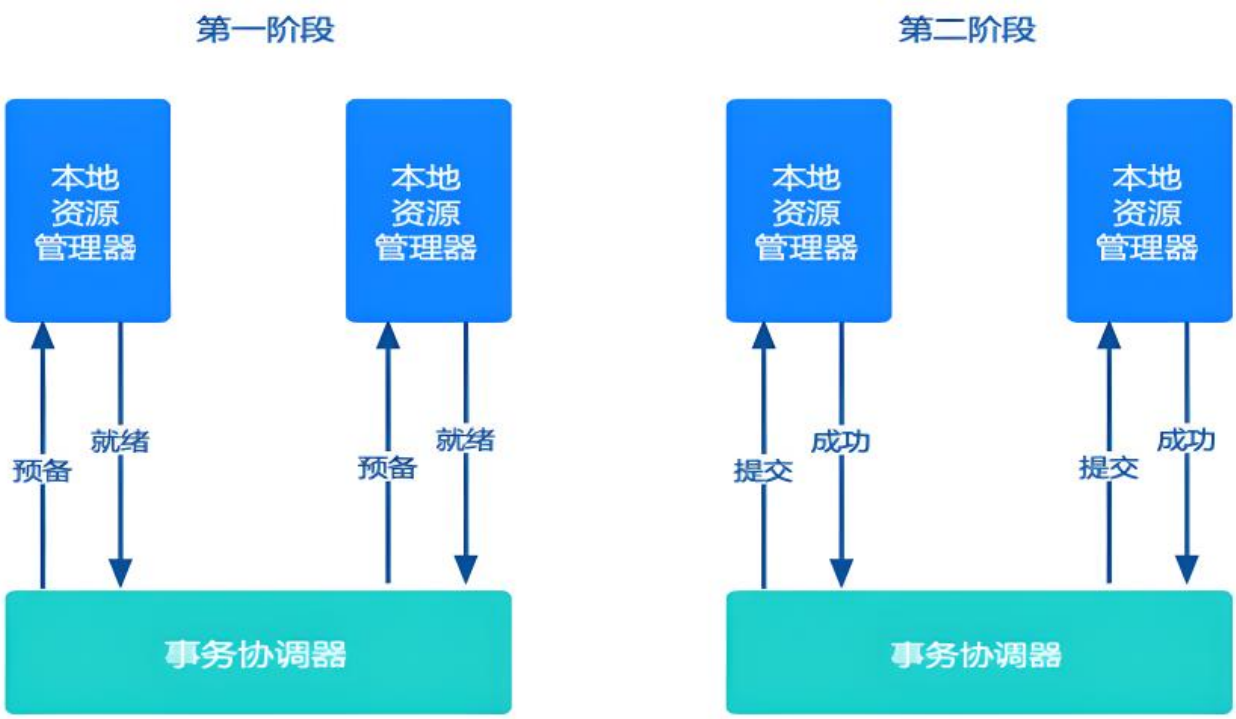

图 2.2 二阶段协议工作示意图

2.3 数据库事务的作用

1. **提高了恢复到正常状态的方法**：数据库事务提供了一种机制，使得一系列操作要么全部成功，要么全部失败，这被称为原子性。在 OceanBase 中，事务的原子性确保了即使在出现系统故障（如断电或系

统崩溃）的情况下，正在进行的事务要么完整地提交，要么完全回滚，从而不会留下未完成的操作。这种机制有效地保护了数据库的完整性和一致性。

2. **保持数据库一致性**：事务还确保了数据库在执行操作后保持一致的状态。即使在异常情况下，如应用程序错误、系统故障或其他问题，事务管理系统都能确保数据库返回到一个安全的、预定义的状态。OceanBase 利用其先进的分布式架构和强一致性协议来维护跨多个节点的数据一致性，即使在分布式环境中也能确保数据的完整性。
3. **处理并发访问**：在多用户环境中，事务为数据库的并发访问提供了隔离。这意味着多个事务可以同时运行，而不会相互影响，防止了“脏读”、“不可重复读”和“幻读”等问题。OceanBase 通过其高级的隔离级别和锁机制来管理并发访问，确保了在多个事务同时操作时数据库的一致性和完整性。这对于需要高并发处理能力的应用（如在线交易处理）尤为重要。
4. **OceanBase 的特定优势**：OceanBase 的事务管理机制特别适用于大规模分布式环境。它采用了高效的分布式协议和算法，优化了跨节点事务的性能和一致性。此外，OceanBase 还提供了灵活的事务隔离级别，允许用户根据具体需求和性能考虑来调整事务的隔离程度。

2.4 事务的结构

OceanBase 的事务结构遵循典型的数据库事务流程，即一个数据库事务包含一条或者多条 DML 语句。整个事务流程包括开启事务、语句执行、和结束事务的步骤。以下是以 mysql 数据库为例，对这些步骤的详细解析。

2.4.1 开启事务

OceanBase 通过执行 `BEGIN`、`START TRANSACTION` 或数据修改语句（如 `INSERT`、`UPDATE`、`DELETE`、`SELECT ... FOR UPDATE`）来开启事务。特别地，通过 `SET autocommit=0;` 关闭自动提交也可以开启事务。一旦事务开始，OceanBase 会为该事务分配一个唯一的事务 ID，用于标识和追踪。例如，将自动提交关闭后执行 `UPDATE t SET c="b" WHERE i=1;` 会开启一个新的事务。

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT /* 在事务开始时创建一个一致性视图 */
  | READ WRITE /* 将事务设置为读写模式 */
  | READ ONLY /* 将事务设置为只读模式 */
}

BEGIN [WORK]

SET autocommit = {0 | 1} /* 设置自动提交的状态 */
```

2.4.2 语句执行

在事务的执行过程中，OceanBase 在每个涉及的分区创建事务上下文。这个上下文记录了语句执行过程中的数据快照版本号以及对该分区所做的修改。

2.4.3 结束事务

在 OceanBase 中，结束事务的机制确保了数据的一致性和完整性。具体来说，有以下三种场景。

其一，用户可以通过执行 COMMIT 或 ROLLBACK 语句显式地结束事务。COMMIT 语句会将事务中的所有更改永久保存到数据库中，而 ROLLBACK 语句则会撤销事务中的所有操作。

其二，当用户在事务中执行 DDL（数据定义语言）操作（如 CREATE、DROP 等）时，OceanBase 会隐式地提交当前事务。随后的操作将会在一个新的事务中执行。

其三，如果在事务执行过程中客户端断开连接，OceanBase 会隐式地发起一个 ROLLBACK 请求，回滚当前未完成的事务。

在事务结束时，OceanBase 会收集事务中修改过的所有分区信息，并对这些分区发起提交或回滚事务的请求。这确保了数据的一致性和事务的原子性。

2.5 数据库事务控制

2.5.1 事务控制概述

事务的整个生命周期通常包括开启事务、执行查询和 DML 语句，结束事务等过程。其中，开启事务可以通过 BEGIN、START TRANSACTION 等语句显式开启，也可以通过 DML 语句隐式开启。结束事务通常有两种方式，通过 COMMIT 语句提交事务或者通过 ROLLBACK 语句回滚事务。此外，在一个活跃事务中执行 DDL 语句也会导致隐式地提交事务。

在事务内部可以创建 Savepoint，它标记了事务内部的一个点，您可以在事务后续的执行过程中通过 ROLLBACK TO SAVEPOINT 语句回滚到该点。

1. **事务大小：** OceanBase 数据库 V2.x 版本上单个事务大小有限制，而 OceanBase 数据库 V3.x 版本因支持了大事务，不再受此限制。
2. **语句超时与事务超时：** 系统变量 ob_query_timeout 控制着语句执行时间的上限，语句执行时间超过此值会给应用返回语句超时的错误，错误码为 -6212，并回滚语句，通常该值默认为 10s。系统变量 ob_trx_timeout 控制着事务超时时间，事务执行时间超过此值会给应用返回事务超时的错误，错误码为 -6210，此时需要应用发起 ROLLBACK 语句回滚该事务。系统变量 ob_trx_idle_timeout 表示 Session 上一个事务处于的 IDLE 状态的最长时间，即长时间没有 DML 语句或结束该事务，则超过该时间值后，事务会自动回滚，再执行 DML 语句会给应用返回错误码 -6224，应用需要发起 ROLLBACK 语句清理 Session 状态。
3. **事务查询：** 可以利用虚拟表 __all_virtual_trans_stat 来查询系统中当前所有的活跃事务。

2.5.2 活跃事务

活跃事务是指事务已经开启，但还没有提交或者回滚的事务。活跃事务所做的修改在提交前都是临时的，别的事务无法看到。state 字段对应的值所表示的含义如下表所示。

表 2.1 state 值及其对应说明

state 的值	说明
INIT	表示事务处于活跃状态，所有修改对其他事务不可见。
REDO COMPLETE	表示事务已经将所有数据成功以日志形式持久化。
PREPARE	表示事务已经开始提交，目前处于 PREPARE 状态，读取该事务的修改可能会被卡住（取决于版本号）。
PRECOMMIT	表示事务即将提交，正在同步事务的提交版本号到所有涉及的参与者。
COMMIT	表示事务已经开始提交，且目前处于 COMMIT 状态，其他事务可以看到该事务的修改（取决于版本号）。
ABORT	表示事务已经回滚，处于 ABORT 状态，其他事务不能看到该事务的修改。
CLEAR	表示事务已经提交或回滚结束，处于 CLEAR 状态。

2.5.3 Savepoint

Savepoint 是 OceanBase 数据库提供的可以由用户定义的一个事务内的执行标记。用户可以通过在事务内定义若干标记并在需要时将事务恢复到指定标记时的状态。

例如，当用户在执行过程中在定义了某个 Savepoint 之后执行了一些错误的操作，用户不需要回滚整个事务再重新执行，而是可以通过执行 ROLLBACK TO 命令来将 Savepoint 之后的修改回滚。

如下表所示的示例，用户可以通过创建 Savepoint sp1 来对之后插入的数据执行回滚。

表 2.2 Savepoint 命令

命令	解释
BEGIN;	开启事务
INSERT INTO a VALUE(1);	插入行 1。
SAVEPOINT sp1;	创建名为 sp1 的 Savepoint
INSERT INTO a VALUE(2);	插入行 2。
SAVEPOINT sp2;	创建名为 sp2 的 Savepoint
ROLLBACK TO sp1;	将修改回滚到 sp1
INSERT INTO a VALUE(3);	插入行 3
COMMIT;	提交事务

在 OceanBase 数据库的实现中，事务执行过程中的修改都有一个对应的"sql sequence"，该值在事务执行过程中是递增的（不考虑并行执行的场景），创建 Savepoint 的操作实际上是将用户创建的 Savepoint 名字对应到事务执行的当前"sql sequence"上，当执行 ROLLBACK TO 命令时，OceanBase 数据库内部会执行以下操作：

1. 将事务内的所有大于该 Savepoint 对应"sql sequence"的修改全部回滚，并释放对应的行锁，例如示例中的行 2。

2. 删除该 Savepoint 之后创建的所有 Savepoint，例如示例中的 sp2。

ROLLBACK TO 命令执行成功后，事务仍然可以继续操作。

2.6 Redo 日志

Redo 日志是 OceanBase 数据库用于宕机恢复以及维护多副本数据一致性的关键组件。Redo 日志是一种物理日志，它记录了数据库对于数据的全部修改历史，具体的说记录的是一次写操作后的结果。从某个持久化的数据版本开始逐条回放 Redo 日志可以还原出数据的最新版本。

2.6.1 日志的作用

OceanBase 数据库的 Redo 日志有两个主要作用：

1. **宕机恢复：**与大多数主流数据库相同，OceanBase 数据库遵循 WAL（write-ahead logging）原则，在事务提交前将 Redo 日志持久化，保证事务的原子性和持久性（ ACID 中的 "A" 和 "D"）。如果 observer 进程退出或所在的服务器宕机，重启 OBServer 节点会扫描并回放本地的 Redo 日志用于恢复数据。宕机时未持久化的数据会随着 Redo 日志的回放而重新产生。
2. **多副本数据一致性：**OceanBase 数据库采用 Multi-Paxos 协议在多个副本间同步 Redo 日志。对于事务层来说，一次 Redo 日志的写入只有同步到多数派副本上时才能认为成功。而事务的提交需要所有 Redo 日志都成功写入。最终，所有副本都会收到相同的一段 Redo 日志并回放出数据。这就保证了一个成功提交的事务的修改最终会在所有副本上生效并保持一致。Redo 日志在多个副本上的持久化使得 OceanBase 数据库可以提供更强的容灾能力。

2.6.2 日志文件类型

OceanBase 数据库采用了租户级别的日志流，每个分区的所有日志要求在逻辑上连续有序。机器上同一租户的所有 Tablet 产生的日志会写入到同一个日志流，日志流中的日志存在全序关系。

OceanBase 数据库的 Redo 日志文件称为 Clog，Clog 全称 Commit log，用于记录 Redo 日志的日志内容，位于 store/clog/tenant_xxxx 目录下（其中，xxxx 表示租户 ID），文件编号从 0 开始并连续递增，文件 ID 不会复用，单个日志文件的大小为 64 MB。这些日志文件记录数据库中的数据所做的更改操作，提供数据持久性保证。

2.6.3 日志的产生

OceanBase 数据库的每条 Redo 日志最大为 2 MB。事务在执行过程中会在事务上下文中维护历史操作，包含数据写入、上锁等操作。在 V3.x 之前的版本中，OceanBase 数据库仅在事务提交时才会将事务上下文中保存的历史操作转换成 Redo 日志，以 2 MB 为单位提交到 Clog 模块，Clog 模块负责将日志同步到所有副本并持久化。在 V3.x 及之后的版本中，OceanBase 数据库新增了即时写日志功能，当事务内数据超过 2 MB 时，生成 Redo 日志，提交到 Clog 模块。以 2 MB 为单位主要是出于性能考虑，每条日志提交到 Clog 模块后需要经过 Multi-Paxos 同步到多数派，这个过程需要较多的网络通信，耗时较多。因此，相比于传统数据库，OceanBase 数据库的单条 Redo 日志聚合了多次写操作的内容。

OceanBase 数据库的一个分区可能会有 3 ~ 5 个副本，其中只有一个副本可以作为 Leader 提供写服务，产生 Redo 日志，其它副本都只能被动接收日志。

2.6.4 日志的回放

Redo 日志的回放是 OceanBase 数据库提供高可用能力的基础。日志同步到 Follower 副本后，副本会将日志按照 transaction_id 哈希到同一个线程池的不同任务队列中进行回放。OceanBase 数据库中不同事务的 Redo 日志并行回放，同一事务的 Redo 日志串行回放，在提高回放速度的同时保证了回放的正确性。日志在副本上回放时首先会创建出事务上下文，然后在事务上下文中还原出操作历史，并在回放到 Commit 日志时将事务提交，相当于事务在副本的镜像上又执行了一次。

2.6.5 日志容灾

通过回放 Redo 日志，副本最终会将 Leader 上执行过的事务重新执行一遍，获得和 Leader 一致的数据状态。当某一分区的 Leader 所在的机器发生故障或由于负载过高无法提供服务时，可以重新将另一个机器上的副本选为新的 Leader。因为它们拥有相同的日志和数据，新 Leader 可以继续提供服务。只要发生故障的副本不超过一半，OceanBase 数据库都可以持续提供服务。发生故障的副本在重启后会重新回放日志，还原出未持久化的数据，最终会和 Leader 保持一致的状态。

对于传统数据库来说，无论是故障宕机还是重新选主，正在执行的事务都会伴随内存信息的丢失而丢失状态。之后通过回放恢复出来的活跃事务因为无法确定状态而只能被回滚。从 Redo 日志的角度看就是回放完所有日志后仍然没有 Commit 日志。在 OceanBase 数据库中重新选主会有一段时间允许正在执行的事务将自己的数据和事务状态写成日志并提交到多数派副本，这样在新的 Leader 上事务可以继续执行。

2.6.6 日志的控制与回收

日志文件中记录了数据库的所有修改，因此回收的前提是日志相关的数据都已经成功持久化到磁盘上。如果数据还未持久化就回收了日志，故障后数据就无法被恢复。

当前 OceanBase 数据库的日志回收策略中对用户可见的配置项有 2 个：

clog 磁盘空间使用上限，由全局配置项 log_disk_utilization_limit_threshold 控制，默认值是 95，代表允许 Clog 使用的磁盘空间占总磁盘空间的百分比。这是一个刚性的限制，超过此值后这个 OBServer 节点不再允许任何新事务的写入，同时不允许接收其他 OBServer 节点同步的日志。对外表现是所有访问此 OBServer 节点的读写事务报错 "transaction needs rollback"。

Clog 磁盘复用下限，由配置项 log_disk_utilization_threshold 控制。在系统工作正常时，Clog 会在此水位开始复用最老的日志文件，默认值是 Clog 独立磁盘空间的 80%，不可修改。因此，正常运行的情况下，Clog 磁盘空间占用不会超过 80%，超过则报错 "clog disk is almost full", 提醒 DBA 处理。

3 OceanBase 事务管理的特点

数据并发性和数据一致性两种语义称为并发控制模式，即为 ACID 中的 I(Isolation level). 数据库的并发性体现在数据库允许用户通过事务并发地访问与修改用一个数据，数据一致性用户观察区域内为一致的数据库状态。

通过并发性处理，数据库具有更高的事务处理效率.

3.1 并发性

3.1.1 并发控制模式

在数据库事务管理中，每个事务包括多个读写操作，操作对象为数据据内部不同数据。其中事务读写操作会在事务之间建立依赖关系，关系主要分为：

1. 写写冲突：事务 A 修改数据 X 后，事务 B 再修改同一数据 X，事务 B 依赖事务 A
2. 写读冲突：事务 A 读取数据 X 后，若数据 X 对应是由事务 B 修改的，事务 A 依赖 事务 B
3. 读写冲突：事务 A 读取数据 X 后，事务 B 再修改了同一数据 X，事务 B 依赖事务 A

基于依赖关系可以设计两种基础并发控制模型，分别为串行执行和可串行化执行并发模式控制。

串行执行 在串行执行中，一个进程执行需要在另一个进程操作结束后开始执行。这种机制保证了前一个操作不会影响下一个操作的执行，但在高并发场景下，串行方案明显不能满足事务处理的时效性要求。

可串行化 可以通过并行的防止执行事务内的多个操作，达到和串行执行相同的效果。而通过冲突来定义的可串行化，一般称为冲突可串行化。当事务间的冲突关系没有成环即可以保证冲突可串行化。两阶段和乐观锁是常见的实现机制前者是通过排它地通过加锁限制其他的事务的冲突修改，并通过死锁检测机制回滚产生循环的事务，保证无环；后者通过在提交时的检测阶段，回滚所有可能会导致成环的事务保证不会产生环。

开销分析 在现实应用中，由于实现可串行化隔离级别的商业化数据库较少，且串行执行和实现机制都有极大的性能代价。串行执行会导致资源利用率低下，系统对并发请求的响应时间较长；可串行化在实现过程中可能会因为 等待锁的机制呆滞事务需要等待锁的释放。而在乐观锁的实现中，事务在提交时需要检测可能的冲突，如果检测到冲突，则需要回滚事务。回滚操作会引入一定的开销，尤其在事务冲突较为频繁的情况下开销巨大。

解决方案 在应用中一般会通过允许一些容易接受的成环条件来暴露一些异常，并增加事务的性能和可扩展性。在实际应用中，为了平衡性能和事务的一致性，数据库系统通常提供多个隔离级别，用户可以根据具体需求选择适当的隔离级别。每个隔离级别都对应一定的性能代价。因此一般会通过允许一些容易接受的成环条件来暴露一些异常，并增加事务的性能和可扩展性。其中快照读和读已提交是比较常见的允许异常的并发控制。

其中快照读隔离级别依赖维护多版本数据，并在读取时通过一个固定的读版本号读一个对应版本的数据。因此对于同一事务中的不同数据会产生因为读写冲突导致的环。比如说事务 A 读取数据版本为 1 的 X 后修改产生数据版本 2 的 Y，事务 B 取数据版本为 1 的 Y 后修改产生数据版本 2 的 X。我们可以清楚地发现事务 A 与事务 B 产生了环，这种异常即写偏斜（Write Skew）。这是快照读暴露给用户的异常；而对于读已提交隔离级别，则会暴露不可重复读等异常，即事务内部两次读取结果不同。

3.1.2 Oceanbase 数据库的并发控制模型

OceanBase 数据库、支持快照读和读已提交两种隔离级别，并在分布式语义上隔离级别保证外部一致性。

多版本数据和事务表 为了支持读写不互斥，OceanBase 数据库从设计开始就选择了多版本作为存储，并让事务对于全局来说，维护两个版本号，读版本号和提交版本号。如图 3.1 所示内存中存储数据通过版本（ts）、值（val）和事务 id（txn）来维护更新，并将多次更新同时维护来保持多版本；其次，内存中存在一个事务表，事务表中记录了每个事务的 id、状态以及版本。事务开始和提交时会通过全局时间戳缓存服务（Global Timestamp Cache）获取时间戳作为读时间戳和作为提交时间戳参考的一部分。

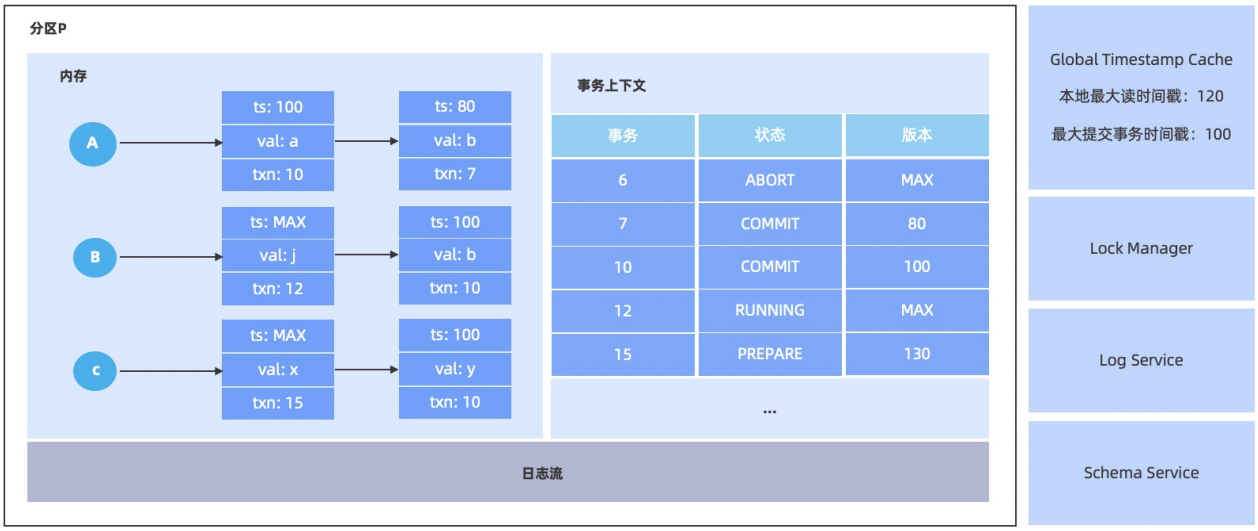

图 3.1 多版本数据和事务表

提交请求处理 OceanBase 的分布式事务有三个状态，RUNNING、PREPARE 和 COMMIT。由于事务状态在分布式场景下无法原子地确认，PREPARE 是两阶段提交所增加的阶段。因此 OceanBase 选择在事务中维护本地提交版本号（local commit version，即 prepare version），事务的全局提交版本号是由所有分区本地提交版本号的最大值决定的。最后事务的全局提交版本号 一定大于等于本地分区的本地提交版本号，从而保证每个分区的数据读写安全，这个保证也是之后我们实现读写请求并发控制的关键之一。事务提交时会走对应的两阶段提交流程。我们对于参与者中的每一个分区取本地最大读时间戳作为本地提交时间戳。此行为可以防止产生单值的读写冲突（anti dependency）。

写请求处理 在写入数据的时候，为了保证不出现写写冲突（write dependency），修改会使用两阶段锁协议，当触发写入请求时，若发现本行的多版本有正在执行的事务，则会把这次请求放入锁管理器中等待，OceanBase 数据库在锁管理器中，由此实现等待队列，通过锁或超时来唤醒此写请求。

读请求处理 OceanBase 读取时会使用读版本号来读取对应的数据，在真正读取时会用读版本号先更新本地最大读时间戳. 当读取到提交或回滚的事务时，可以根据 全局提交时间戳 和状态比较简单地推测出是否需要读到对应数据。当读取到 RUNNING 状态的事务时，由于推高了 本地最大读时间戳，因此之后 RUNNING 状态的事务一定会以更大的 本地时间戳 来进入两阶段提交，根据保证和快照读的概念，可以安全地跳过这个数据。当读取到 PREPARE 状态的事务时，若本地时间戳大于读时间戳则该事务可以保证不用被读取到。而对于本地时间戳 小于读时间戳的情况，那么无法确认这个时间戳最后的 全局提交时间戳 和读时间戳的关系，则此时需要在这行事务上等待。

3.1.3 锁机制

有两种常见的实现机制, 即两阶段锁和乐观锁机制。前者是通过排它地通过加锁限制其他的事务的冲突修改，并通过死锁检测机制回滚产生循环的事务保证无环；后者通过在提交时的检测阶段，回滚所有可能会导致异常的事务保证不会产生异常。

OceanBase 数据库使用了多版本两阶段锁来维护其并发控制模型的正确性,OceanBase 使用了以数据行为级别的锁粒度. 同一行的不同列修改会导致该行对应的锁上的互斥；而不同行修改对应不同行的互斥锁，不构成互斥关系。似于其余的多版本两阶段锁的数据库，OceanBase 数据库的读取是不上锁的，因此可以做到读写不互斥从而提高用户读写事务的并发能力。对于锁的存储模式，OceanBase 选择将锁存储在行上（物理空间可能位于存储在内存与磁盘上）从而避免在内存中维护大量锁的数据结构，同时该策略会在内存中维护锁之间的等待关系，从而在锁释放的时候唤醒等待在锁上面的其余事务。

锁机制的使用 OceanBase 对表中的每一行加锁，具体体现在：当用户使用 UPDATE 对相应的行进行修改时，数据库将隐式地位某一行加上行锁，所有并发的更新都会被阻塞并等待。从而预防并发的修改导致的脏写（Dirty Write)。则用户无需显示地指示锁的范围, 可以依赖 OceanBase 数据库内部的机制做到并发控制的效果。同时用户仍可以显式地指定使用锁机制，在使用 SELECT 语句获取对应行数据时，数据库将为这一数据行加锁，此时所有的并发更新都会被阻塞并等待。

锁机制的粒度 OceanBase 数据库现在不支持表锁，只支持行锁，且只存在互斥行锁。传统数据库中的表锁主要是用来实现一些较为复杂的 DDL 操作，在 OceanBase 数据库中，还未支持一些极度依赖表锁的复杂 DDL，而其余 DDL 通过在线 DDL 变更来实现。在更新同一行的不同列时，事务依旧会互相阻塞，如此选择的原因是为了减小锁数据结构在行上的存储开销。而更新不同行时，事务之间不会有任何影响。

锁机制的互斥 OceanBase 数据库使用了多版本两阶段锁，事务的修改每次并不是原地修改，而是产生新的版本。因此读取可以通过一致性快照获取旧版本的数据，因而不需要行锁依旧可以维护对应的并发控制能力，因此能做到执行中的读写不互斥，这极大提升了 OceanBase 数据库的并发能力。比较特殊的是 SELECT ... FOR UPDATE，此类执行依旧会加上行锁，并与修改或 SELECT ... FOR UPDATE 产生互斥与等待。而修改操作则会与所有需要获取行锁的操作产生互斥。

锁机制的存储 OceanBase 数据库的锁存储在行上，这种存储方式减少了内存中所需要维护的锁数据结构带来的开销。当事务获取到行锁后会在对应的行上设置对应的事务标记，即行锁持有者。当事务尝试获取行锁时，会通过对应的事务标记发现自己不是行锁持有者而放弃并等待或发现自己是行锁持有者后获得行的使用能力。当事务释放行锁后，就会在所有事务涉及的行上解除对应的事务标记，从而允许之后的事务继续尝试获取。

当数据被转储到 SSTable 后，宏块内部的数据上会记录对应的事务标记。其余事务依旧需要通过事务标识来辨识是否可以允许访问对应的数据。与内存中的锁机制不同的是，由于 SSTable 不可变的特性，无法在事务释放行锁后，立即清除宏块内部的数据上的事务标记。当然依旧可以通过事务标识来确认找到对应的事务信息来确认事务是否已经解锁。

锁机制的释放 类似于大部分的两阶段锁实现，OceanBase 数据库的锁在事务结束（提交或回滚）的时候释放的，从而避免数据不一致性的影响。OceanBase 数据库还存在其余的释放时机，即 SAVEPOINT，当用户选择回滚至 SAVEPOINT 后，事务内部会将 SAVEPOINT 及之后所有涉及数据的行锁，全部根据 OceanBase 数据库锁机制的互斥 中介绍的机制进行释放。

锁机制的唤醒 为了唤醒事务，当产生互斥后，会在内存中维护行与事务的等待关系，如图 3.2 所示，行 A 被事务 B 持有，被事务 C 与事务 D 等待。此等待关系的维护，是为了行锁释放的时候可以唤醒对应的事务 C 与 D。当事务 B 释放行 A 后，会根据顺序唤醒事务 C，并依赖事务 C 唤醒事务 D。

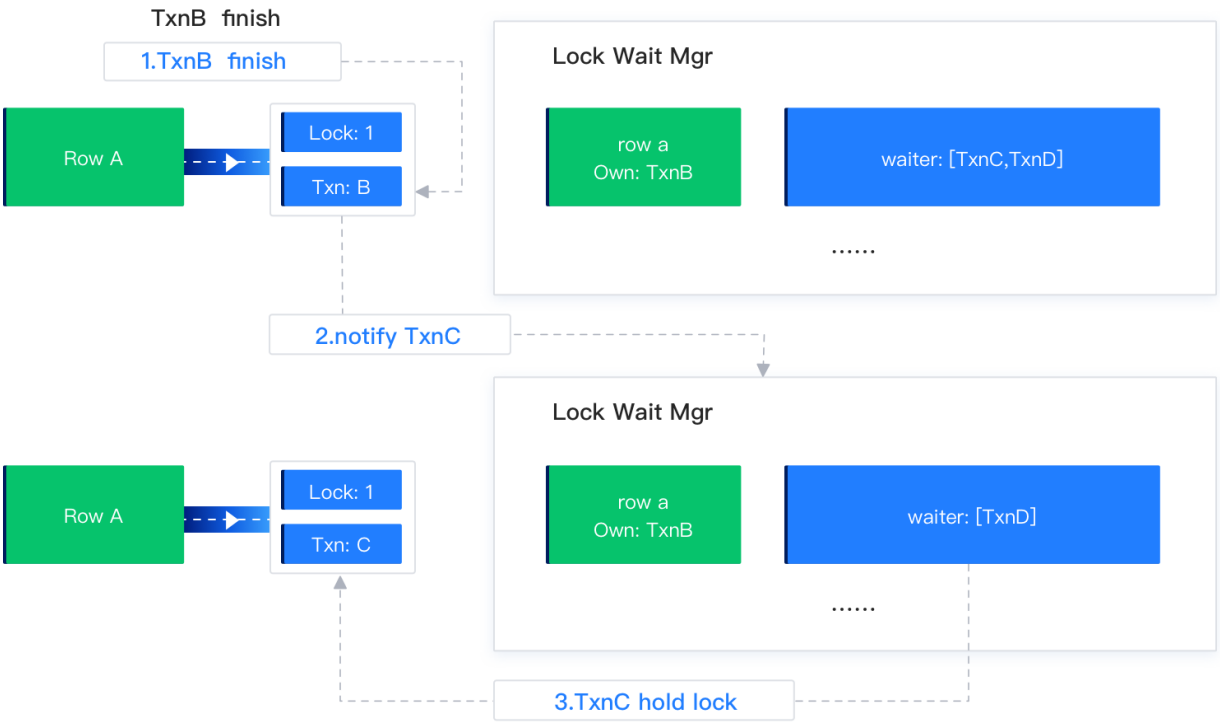

图 3.2 锁机制的唤醒-行与事务的等待关系

主动死锁检测 目前 OceanBase 数据库实现的死锁检测称为 LCL（Lock Chain Length）死锁检测方案，是一种基于优先级的多出度分布式死锁检测方案，OceanBase 数据库的死锁检测算法可以保证不误杀或多杀事务。其中的关键概念为：

1. 基于优先级: 在互相形成死锁的多个事务中，LCL 死锁检测方案总是倾向于杀掉其中优先级最低的事务来解除死锁，目前在死锁检测中事务的优先级指标主要为事务的开启时间，越晚开启的事务具有越低的优先级。
2. 多出度: 每一个事务都可以同时等待超过一个的其他事务。
3. 分布式死锁检测：每一个代表事务进行死锁检测的节点仅知道该节点自身的依赖信息，在不需要全局的锁管理器的情况下即可探测节点间的死锁。

死锁检测原理 一个分布式事务为提高执行效率通常需要同时访问多个分区的数据，并可能同时发现存在多个锁冲突事件，此时为提高死锁检测的效率，可以描述出一个事务同时等待多个事务的单向依赖的有向边，称为多出度。常见的死锁检测方案多采用路径推动算法（path-pushing algorithm），这种算法应用在多出度场景下大多存在多杀以及误杀的问题。

O 测按 Base 所提出的 LCL 死锁检测方案采用的是一种经过特殊设计的边跟踪算法（edge-chasing algorithm），在 LCL 死锁检测方案中，每个节点维护两个状态，分别称为深度值以及令牌值，为防止多杀，一个节

点维护的令牌值数量不能多于一个，令牌值之间可以比较，并使大的令牌值覆盖小的令牌值，如此可以保证在一个环路中，只有最大令牌值的节点可以探测到死锁，可以避免多杀的问题。LCL 算法流程如图 3.3 所示

Algorithm 1: *LCL*-based Distributed Algorithm for Deadlock Detection and Resolution

Input: vertex A, vertex B, A waits for B, $A \neq B$
Output: res

▷ **Stage 1: *LCL* Proliferation**
for $i \leftarrow 1$ *to max(AsgWidth*(scc)*,1)* **do**
  $A.\langle PuAP, PuID\rangle = A.\langle PrAP, \textit{PrID}\rangle$;
  $B.\langle PuAP, PuID\rangle = B.\langle PrAP, \textit{PrID}\rangle$;
  $B.LCLV = \max(B.LCLV, A.LCLV + 1)$;

▷ **Stage 2: *LCL* Spread**
for $i \leftarrow 1$ *to* $2\times$*SccDiam*(scc) **do**
  $B.LCLV = \max(B.LCLV, A.LCLV)$;
  if *B.LCLV == A.LCLV* **then**
    $B.\langle PuAP, PuID\rangle =$
     $\max(B.\langle PuAP, PuID\rangle, A.\langle PuAP, PuID\rangle)$;

▷ **Stage 3: *LCL* Detection**
if *B.LCLV == A.LCLV and*
 $B.\langle PuAP, PuID\rangle == A.\langle PuAP, PuID\rangle$ *and*
 $B.\langle PuAP, PuID\rangle == B.\langle PrAP, \textit{PrID}\rangle$ **then**
  `// B is a detected deadlock vertex`
  $res \leftarrow B$;

return res;

图 3.3 LCL 算法

图 3.4 展示了 LCL 在 OceanBase 中的实现，客户端连接集群中的节点 OBServer1 并启动事务，OBServer1 被称为事务协调者。SQL 语句涉及对数据行的修改会被路由到 Paxos Leader 执行。Leader 上的锁冲突可能导致执行失败，事务协调者从失败的子任务中获取信息并建立依赖关系。LCL 的推演过程会周期性地通过依赖关系传递一个特殊的消息，以发现可能的死锁并进行消除。

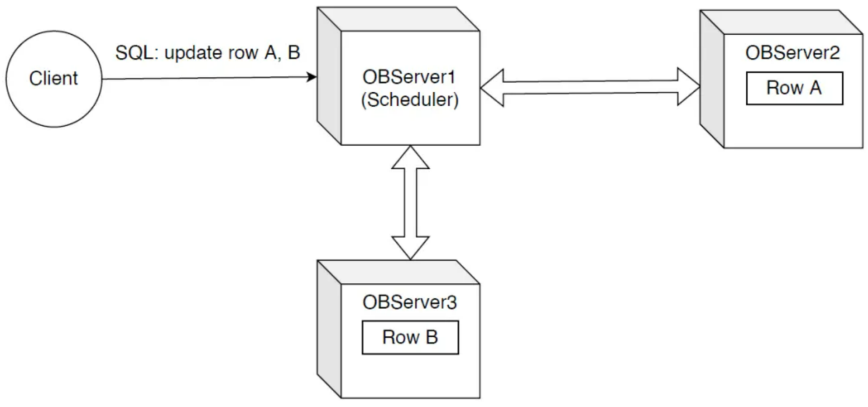

图 3.4 LCL 算法模型

3.2 一致性

3.2.1 数据一致性的定义

数据库通过维护每次更改，产生新的版本，从而做到读写不互斥, 这被称为多版本并发控制（MVCC）。对于不同的事务版本，我们需要为这种数据多版本来定义语义，保证用户看到一个一致的数据库状态，即数据的一致性快照。需要注意的是，此处的数据一致性不等价于 ACID 中的 C (Consistency)。

3.2.2 多版本读一致性介绍

为了支持数据读写不互斥，OceanBase 数据库存储了多个版本的数据。为了处理多版本数据的语义，我们需要维护多版本一致性。OceanBase 数据库的多版本一致性是通过读版本和数据版本来保证的，通过给读取版本号，返回小于读取版本号的所有提交数据，来定义多版本一致性。因此我们需要注意几点:

a、未提交事务: 不能读到非本事务的未提交事务, 否则若对应事务回滚，就会产生脏读（dirty read）。

b、事务一致性快照：要读取小于读取版本号的所有提交数据，来保证一个用户可理解的一致性点，否则我们就会产生会返回一半事务（fractured read）。

c、读写不互斥：在满足未提交事务与事务一致性点的前提下，依旧要保证读写不互斥。

3.2.3 多版本读一致性使用

实现并发控制的核心之一，就是多版本读一致性在数据库内部的普遍适用性，主要包含以下要点：

a、弱一致性读: OceanBase 数据库的弱一致性读依旧提供了事务的一致性快照，不会返回未提交事务和一半事务的情况。

b、强一致性读：OceanBase 数据库的强一致性读分为两种，分别是事务级别读版本号和语句级别版本号，分别提供给快照读和读已提交两个隔离级别使用，需要提供返回事务一致性点的能力。

c、只读事务：OceanBase 数据库的只读语句也是要求提供强一致性读相同的能力，需要提供返回事务一致性点的能力。

d、备份恢复点：OceanBase 数据库需要提供可以备份到一个事务一致性快照，防止备份了多余、未提交的事务或者没有备份需要备份的事务。

3.2.4 多版本读一致性实现

事务表记录着当前副本中正在执行中的事务集合，是一个内存表，根据不同的事务状态，来决定事务在执行过程中是否要读取到对应的数据。其中包含提交（COMMIT）、执行（RUNNING）、回滚（ABORT），四种数据状态。对于执行过程中的事务，可能存在 本地提交版本号（local commit version，即 prepare version）；对于提交的事务, 存在 全局提交版本号（global commit version，即 commit version）。其中 全局提交版本号 记录着事务最终的版本，也是一致性位点的决定因素。

4 OceanBase 事务管理源码探究

注：本报告分析的源码是 OceanBase3.1 版本的源码。

4.1 事务的接口

4.1.1 事务外部接口

事务的外部接口在当前 OceanBase 源码的 src/sql 目录下 ob_sql_trans_control.h 文件和 ob_sql_trans_control.cpp 文件中。这里 OceanBase 中协议层 (处理通信的组件) 对事务层提供的接口进行了封装，从而使得调用 SQL 层更加方便，并且维护了 TransState 的状态 (两位表示一个动作：低位表示是否执行，高位表示是否成功)，语句执行结束后在统一位置根据 TransState 调用正确的事务接口，保证在任何异常状态下事务资源不泄露。

Listing 1 ob_sql_trans_control.h

```
int ObSqlTransControl::end_trans(
    storage::ObPartitionService* ps, ObSQLSessionInfo* session, bool& has_called_txs_end_trans)
{
  int ret = OB_SUCCESS;
  int wait_ret = OB_SUCCESS;
  int hook_ret = OB_SUCCESS;
  has_called_txs_end_trans = false;
  if (OB_ISNULL(ps) || OB_ISNULL(session)) {
    ret = OB_INVALID_ARGUMENT;
    LOG_ERROR("invalid argument for point", K(ret), K(ps), K(session));
  } else if (true == session->get_in_transaction()) {
     ......
     会话信息的解析
     ......
  } else {
    session->reset_first_stmt_type();
    has_called_txs_end_trans = true;
  }
  return (OB_SUCCESS != ret) ? ret : hook_ret;
}
```

4.1.2 事务核心接口

对于事务事务内部的接口，放在 storage/trans/ob_trans_service.h 和 cpp 文件中。这里主要有三对接口分别对应事务 (start/end trans)，语句 (start/end stmt)，参与者/执行算子 (start/end participant) 的生命周期。对于事务的生命周期而言，在 start trans 开启事务之后，返回一个 ObTransDesc 对象，它会被保存到 session 对象中，其他事务接口和后续语句执行都需要它。它是事务唯一标识，诊断问题时常用。

Listing 2 ob_trans_service.h

```
//事务开始，创建事务唯一id
int start_trans(const uint64_t tenant_id, const uint64_t cluster_id, const ObStartTransParam& req,
     const int64_t expired_time, const uint32_t session_id, const uint64_t proxy_session_id,
         ObTransDesc& trans_desc){
     if(错误...){...}
     else{
     // reset transaction desc, it may be dirty
      trans_desc.reset();
      有错误打印错误日志
      ...
      判断是否需要更新
      trans_desc.set_can_elr(enable_elr);
      trans_desc.set_tenant_config_refresh_time(cur_time);
      ...
     }
   }
}
//事务结束,提交/回滚
int end_trans(..., ObTransDesc& trans_desc, ...);
//语句开始,生成快照
int start_stmt(..., ObTransDesc& trans_desc, ...);
//语句结束，commit/rollback
int end_stmt(..., ObTransDesc& trans_desc);
//执行算子开始，创建事务上下文
int start_participant(const ObTransDesc& trans_desc, ...);
//执行算子结束
int end_participant(..., const ObTransDesc& trans_desc, ...);
```

4.1.3 事务分区接口

一般的，一个 DQL(查询) 或 DML(操纵) 会产生三类执行计划（ObPhyPlanType）：local 计划表示所有该语句执行需访问分区的 leader 都在本节点；remote 计划表示所有分区 leader 都在远程一个节点上；distributed 计划表示分区 leader 在多节点上的情况。开始执行前，根据语句涉及分区信息从 location cache 中获取 leader 位置，选择对应类别的执行计划。location cache 是一个整体架构上的重要组件，用于高效查找分区副本信息，它对外提供统一接口，屏蔽了很多分布式细节。主体实现位于 share/partition_table，sql/ob_sql_partition_location_cache.h 对它进行了封装，同时统一了虚拟表的处理。

根据语句类型（是否只读），事务类型（是否自动提交，是否弱一致性读），当前事务状态，执行计划类型等决定不同的**事务控制调用**。例如，本地计划自动提交，在本节点执行 start/end trans, start/end stmt, start/end participant；远程计划自动提交，上述接口都在远程节点执行。对分布式计划，start/end stmt 在本节点执行，start/end participant 在每个分区 leader 所在节点上执行。这组接口设计目的是在简化事务接口与分场景优化事

务性能间取得一定的平衡。

4.2 事务与会话

务是与会话密切关联在一起的，在一个会话中，客户端可能会执行多个事务，不过在会话存续的任一时刻仅有一个事务运行（称为活跃事务）。反过来，一个事务必定也仅存在于一个会话中。事务被进一步分解为一个或者多个语句 (Statement)，事务和语句之间是一种一对多的关系，一个事务可以包含多个语句，而每一个语句则仅属于一个事务。OceanBase 管理事务的核心就是维持会话、事务、语句的状态以及它们之间的关联，并根据 SQL 语句在各个节点上的执行情况适时推进它们的状态变化。在会话描述类 ObBasicSessionInfo(src/sql/session/ob_basic_session_info.h) 中有多个属性共同描述了会话中进行着的事务 (图 4.1).

Listing 3 ob_basic_session_info.h

```
public:
    transaction::ObTransDesc trans_desc_;
    TransFlags trans_flags_;
    TransResult trans_result_;
    int64_t nested_count_;
    int32_t trans_consistency_type_;
    ...
    common::ObConsistencyLevel consistency_level_;
```

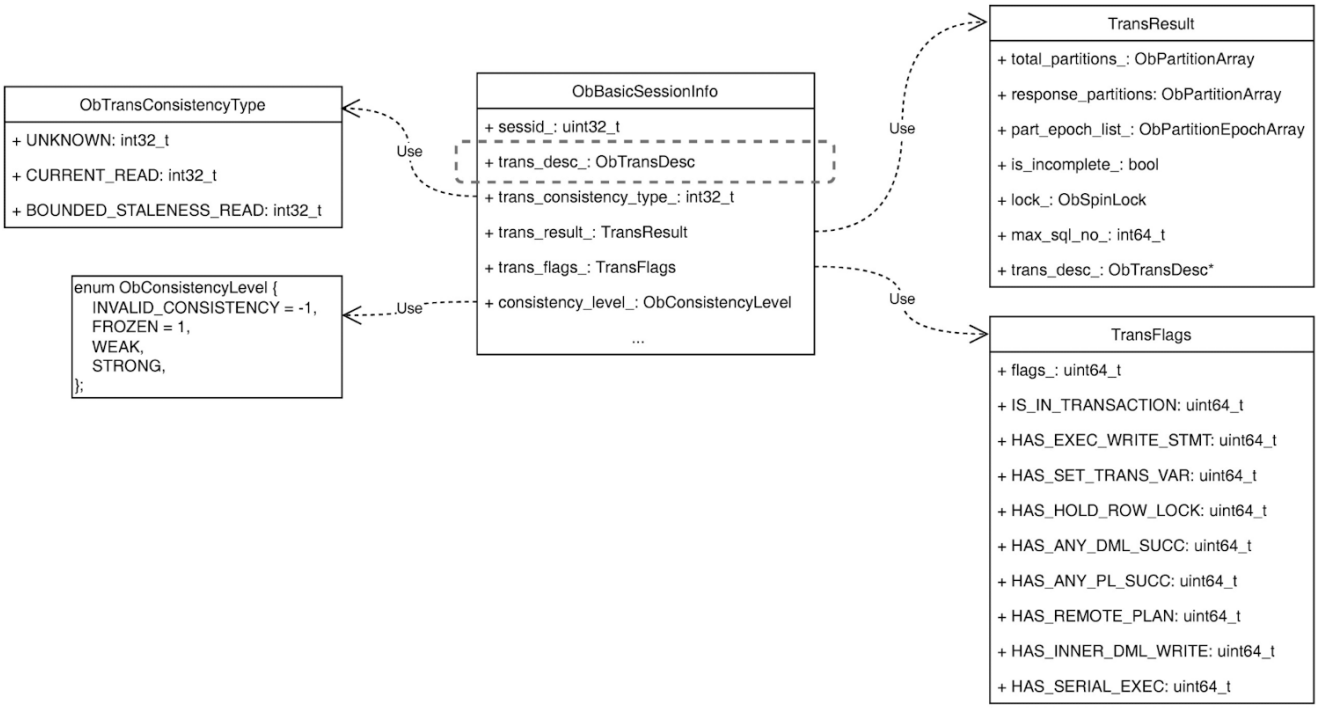

图 4.1 session 与 transaction 有关的类图

1. **trans_consistency_type_**属性描述当前事务的一致性类型，其有两种类型：CURRENT_READ 和 BOUNDED_STALENESS_READ(有界脏读)

2. **consistency_level**_属性描述当前事务的一致性级别, 分为强一致性和弱一致性两种。
3. **trans_flags**_属性中用一个整数的多个位描述了当前事务的一些标志，比如 IS_IN_TRANSACTION 表示会话正在运行一个事务，即有活跃事务存在
4. **trans_results** 描述了会话中当前已执行过的事务的结果状态, 便于事务结束时释放参与结点占用的资源
5. **trans_desc** 是会话中有关事务最核心的部分，它是一个 ObTransDesc 类的对象，它描述了会话中活跃事务的详情，前面已经介绍其重要性

4.3 分布式事务

下面从快照管理、事务提交、并发控制、全局时间戳方面来进行源码分析。

4.3.1 快照管理

对于事务级别的快照，是在 RR 和 Serializable 隔离级别下实现，在事务开启的过程中生成。代码接口在在 storage/trans/ob_trans_service.h 和 cpp 文件

Listing 4 ob_trans_service.cpp

```
/* decide ObTransSnapshotGenType by SQL-Layer's ObTransConsistencyType and ObTransReadSnapshotType
    parameter
 generated at participants side
 */
int ObTransService::decide_read_snapshot_(const ObStmtParam& stmt_param, const
    ObPartitionLeaderArray& pla,
   const ObStmtDesc& stmt_desc, const bool is_external_consistent, ObTransDesc& trans_desc,
   ObPartitionArray& unreachable_partitions)
{
int ret = OB_SUCCESS;
  int32_t snapshot_gene_type = ObTransSnapshotGeneType::UNKNOWN;
  int64_t snapshot_version = ObTransVersion::INVALID_TRANS_VERSION;
  ...
  const int32_t read_snapshot_type = trans_param.get_read_snapshot_type();
  const bool stmt_specific_read_snapshot_is_valid = stmt_desc.has_valid_read_snapshot();

  trans_desc.reset_snapshot_version();
  trans_desc.reset_snapshot_gene_type();
...
}
```

4.3.2 事务提交

多分区事务在 OceanBase 均称为分布式事务，单机多分区事务的提交，本质上仍然是两阶段。对于分布式事务的提交流程，核心路径是：

ObScheTransCtx::end_trans(xxx)->ObCoordTransCtx::handle_message->ObPartTransCtx::handle_message(xxx);

核心逻辑主要在/src/storage/transaction/ob_trans_ctx_mgr.cpp，/src/storage/transaction/ob_trans_ctx_mgr.h,/src/storage/transaction/ob_trans_ctx.cpp，/src/storage/transaction/ob_trans_ctx.h 等文件。

4.3.3 并发控制

OceanBase 并发控制基于 Mvcc+ 行锁的机制来实现，具体而言，读不加锁，读写不相互阻塞、写写并发通过行锁互斥来保证。并发控制的总体实现较为复杂，由于时间紧促，这里不再详细阐述，但是其原码逻辑主要分布在 src/storage/memtable 中的 mvcc 文件夹下和 ob_memtable.cpp/h,ob_memtable_context.cpp/h。

4.3.4 全局时间戳

全局时间戳是分布式事务的核心。为了保持版本之间的可比较性，大部分情况下，这些时间戳的获取渠道是一致的（从同一个时钟获取），在 OceanBase 中这个渠道通过全局时间戳服务实现。这部分代码较多，主要介绍部分 cpp 和 h 文件的作用，细节不再剖析.

1. ob_gts_define.h 定义了一些 GTS 相关的枚举、类型和辅助函数，这些函数主要用于获取 GTS 相关的信息;
2. ob_gts_local_cache.cpp(h) 数据库本地缓存实现，提供了 GTS 的本地缓存类 ObGTSLocalCache，该类维护了关于 GTS 的一系列状态，如本地事务版本、GTS 值、障碍时间戳等；提供了获取本地事务版本和 GTS 的方法，确保数据的一致性和正确性；包含更新 GTS 信息的函数，用于在分布式环境中协调全局事务的时间戳
3. ob_gts_mgr.cpp(h) 实现了全局事务服务的主体功能，包括：初始化、启动、停止和销毁全局时间戳服务；处理来自其他节点的 GTS 请求，通过 RPC 获取全局事务时间戳，并返回结果；处理本地节点发起的 GTS 请求，直接获取本地全局事务时间戳，并返回结果；统计处理请求的性能指标，如请求处理数量和平均处理时间。
4. ob_gts_msg.cpp(h) 定义了用于 GTS 的消息格式 ObGtsRequest 和 ObGtsErrResponse。这些消息用于在 GTS 节点之间进行通信，请求全局事务时间戳或响应错误情况。提供了消息初始化、有效性检查以及序列化和反序列化的操作。
5. ob_gts_response_handler.cpp(h) GTS 中用于处理响应任务的一部分实现.
6. ob_gts_rpc.cpp(h) RPC 通信部分的实现，用于处理 GTS 的请求和错误响应。提供了初始化、启动、停止、等待和销毁等方法，用于管理 RPC 通信的状态.
7. ...

此外还包括其他相关文件如 ob_gts_source.cpp(h)、ob_task_queue.cpp(h) 等, 由于时间精力有限, 这里不再赘述。

4.4 事务日志

OceanBase 的 clog 日志类似于传统数据库的 REDO 日志，这个模块负责在事务提交时持久化事务数据，并实现了基于 Multi-Paxos 的分布式一致性协议。文件主要是/src/storage/transaction/ob_trans_log.cpp(h)，src/storage/transaction/ob_trans_submit_log_cb.cpp(h), 其他地方如/tools/obcdc/src/目录下的含 ob_log_trans 前缀的相关文件等。对于一个分区的一条事务日志，其流程大致如下：

1 事务层调用 log_service->submit_log() 接口提交日志

2 clog 为其分配 log_id 和 submit_timestamp，然后提交到滑动窗口中，生成一个新的 log_task，本地提交写盘，通过 RPC 将日志同步给 follower

3 本地写盘完成时调用 log_service->flush_cb()，更新 log_task 状态，标记本地持久化成功，follower 写盘成功后给 leader 回 ack

4 leader 收到 ack 更新 log_task 的 ack_list

5 leader 在 4、5 两步中会统计多数派，一旦达成 majority，按顺序调用 log_task->on_success() 回调事务，同时发送 confirmed_info 消息发送给 follower 并将此日志从滑动窗口中滑出

6 follower 收到 confirmed_info 消息后尝试将此日志从滑动窗口中滑出，在滑出动作中将日志提交回放

5 总结与收获

综上，我们在本次的调研中深入研究 OceanBase 数据库事务管理，对事务概念及其在系统中的实现进行了深入剖析。首先是事务的基本特性，即原子性、一致性、隔离性和持久性，被细致地探讨，强调了这些特性对数据库系统可靠性和数据完整性的不可或缺贡献。特别是，在事务管理的特点方面，我们针对并发性和一致性进行了深度研究，突出了 OceanBase 在这方面的特点。对于并发性，系统如何处理多个事务同时运行的情况成为关键关注点。调研中更加关注并发控制的实施方式，包括锁定和多版本并发控制（MVCC），以确保在事务同时执行时不会发生干扰，从而维护数据库的稳定性和高效性。

同时，深入研究了 OceanBase 数据库在并发事务处理方面的机制，使我们更好地理解了系统对并发性的精密管理。进一步探究了锁定机制的细节，以及 MVCC 是如何在实际场景中应用的。这不仅有助于我们理解系统的实际运作方式，还为未来的性能优化提供了有力的参考。

在一致性维护方面，本次调用中也着重关注数据库如何保持数据一致性。讨论了事务提交和回滚的过程，并研究了系统如何处理异常情况，确保在各种情境下都能保持数据的一致性。这一方面的深入剖析有助于我们理解数据库系统在面对异常情况时的稳健性和可靠性。

通过源码分析，我们进一步揭示了 OceanBase 数据库事务管理模块的内部结构，关键函数和数据结构被深入挖掘。这样的源码洞察不仅丰富了对系统工作原理的理解，也为未来的系统改进提供了实质性的线索。总体而言，本次调研深入挖掘了 OceanBase 数据库事务管理的方方面面，不仅提供了对系统内部机制的深刻理解，也为未来的系统优化和改进奠定了坚实的基础。

此外，深感这次调研是我们小组成员共同努力和协作的结果，整体而言，小组在这次调研中获得了许多有价值的收获。

首先，小组合作是我们成功的关键。每个小组成员都展现出了高度的责任心和合作精神，共同迎接了调研的挑战。在研究的不同阶段，小组成员能够充分发挥各自的调研能力，形成了良好的协同机制。这种紧密的合作不仅提高了研究效率，每个人负责自己擅长的部分，也为调研提供了更全面的视角。

其次，通过对 OceanBase 数据库事务管理的深入研究，我们对大型数据库系统的内部运作有了更深层次的理解。对事务概念、并发性、一致性等方面的深入研究，使我们在数据库系统设计和优化方面积累了一定的经验。这将对未来的学习和实践提供有力的支持。

在我们每个人的个人成长方面，这次调研也发挥了积极作用。通过源码分析、深入研究数据库内部机制，大家不仅提高了自己的问题解决能力，还培养了对复杂系统的分析和理解能力。这对于我们今后在数据库领域或其他领域的发展都具有重要意义。

通过这次调研，我们不仅对数据库系统的理论知识有了更深刻的认识，也提高了实际解决问题的能力。这次调研过程中遇到的挑战，促使每个人能够更加灵活地应对问题，锻炼了解决实际工程问题的能力。

References

[1] 杨冬青, 李红燕等. 数据库系统概念（第 7 版）. 机械工业出版社, 2021.

[2] 彭煜玮, 杨传辉, 杨志丰. OceanBase 源码解析. 机械工业出版社, 2023.

[3] Zhenkun Yang ,Chuanhui Yang ,Fusheng Han. OceanBase: A 707 Million tpmC Distributed Relational Database System. 2022,PVLDB,15,12,3385-3397.

[4] 李凯, and 韩富晟. "OceanBase 内存事务引擎." 华东师范大学学报 (自然科学版) 2014, no. 5 (2014): 147.

[5] 阳振坤. "OceanBase 关系数据库架构." 华东师范大学学报 (自然科学版) 2014, no. 5 (2014): 141.

[6] CSDN. 第四章：OceanBase 集群技术架构（分布式事务、MVCC、事务隔离级别）[OL].(2023-03-15)[2023-12-10].https://blog.csdn.net/qq_43714918/article/details/129557961.

[7] 周欢, 樊秋实, and 胡华梁. "OceanBase 一致性与可用性分析." 华东师范大学学报 (自然科学版) 2014, no. 5 (2014): 103.

[8] Kai, L. I., and H. A. N. Fu-Sheng. "Memory transaction engine of OceanBase." Journal of East China Normal University (Natural Science) 2014, no. 5 (2014): 147.

[9] Xu, Liang-yu, Xiao-fang Zhang, Li-jun Zhang, and Jin-tao Gao. "Design and Implementation of Trigger on OceanBase." In Advances in Computer Communication and Computational Sciences: Proceedings of IC4S 2017, Volume 2, pp. 121-132. Springer Singapore, 2019.

[10] Huan, Z. H. O. U., F. A. N. Qiu-Shi, and H. U. Hua-Liang. "Consistency and availability in OceanBase." Journal of East China Normal University (Natural Science) 2014, no. 5 (2014): 103.

[11] 祝君, 刘柏众, 余晟隽, 宫学庆, and 周敏奇. "面向 OceanBase 的存储过程设计与实现." 华东师范大学学报 (自然科学版) 2016, no. 5 (2016): 144.

A 小组分工

- 2150260 李元特：1. OceanBase 简介 - 2.4. 事务的结构
- 2150259 顾屹洋：2.5. 数据库事务控制 - 2.6. Redo 日志
- 2151569 明添识：3.1. OceanBase 事务管理并发性
- 2152472 司盛宇：3.2. OceanBase 事务管理一致性
- 2152095 龚　宣：4. OceanBase 事务管理源码探究
- 2153698 何诗锟：5. 总结与收获、摘要、（演讲）